۲

ویراست سوم

مقدمه‌ای بر الگوریتم‌ها

نویسندگان

توماس کورمن
چارلز لیزرسون
رونالد ریوست
کلیفورد استین

ترجمه
مهندس دهقان طرزه
با مقدمهٔ
دکتر یحیی تابش
هیئت علمی دانشگاه صنعتی شریف

ویراست سوم

مقدمه‌ای بر الگوریتم‌ها

جلد دوم

نویسندگان: کرمن
لیزرسون
ریوست
استین

ترجمهٔ : مهندس علی دهقان طرزه
زیر نظر
دکتر یحیی تابش
(عضو هیئت علمی دانشگاه صنعتی شریف)

ناشر کتب مهندسی، کامپیوتر، دانشگاهی، مدیریت
نشر

عنوان و پدیدآور	: **مقدمه‌ای بر الگوریتم/توماس کورمن...[ودیگران] .ترجمه دهقان.**	
مشخصات نشر	: تهران: نص، ۱۳۹۵.	
مشخصات ظاهری	: ۷۶۰ ص(جلد۲) ،۲ جلدی: مصور، جدول، نمودار	
شابک	: ISBN: 978-964-410-259-2	
وضعیت فهرست نویسی	: فیپا.	
یادداشت	: این کتاب در سالهای مختلف توسط ناشران و مترجمان مختلف به چاپ رسیده است.	
یادداشت	: عنوان اصلی / Introduction to algorithms,	
موضوع	: برنامه‌نویسی	
موضوع	: الگوریتم‌های کامپیوتری	
شناسه افزوده	: کورمن، تامس	cormen, ThomasH
شناسه افزوده	: دهقان، علی، ۱۳۶۴، مترجم	
رده‌بندی کنگره	: الف ۱۳۸۸ ۶۷م/QA۷۶/۳/۶	
رده‌بندی دیویی	: ۰۰۵/ ۱	
شماره کتابشناسی ملی	: ۱۷۸۲۶۰۳	

موسسه علمی فرهنگی

مقدمه‌ای بر الگوریتم (جلد ۲) ویراست سوم

توماس کورمن/ چارلز لیزرسون/ رونالد ریوست

علی دهقان

چاپ: سوم/ تابستان۹۵

تیراژ: ۱۰۰۰

ناشر: «نص»

طراحی، آماده‌سازی: موسسه علمی فرهنگی «نص»

قیمت:۳۲۰۰۰ تومان

دفتر: تهران، میدان انقلاب، خ منیری جاوید، بن بست مبین، شماره

تلفن: ۶۶۴۱۲۳۸۵ – ۶۶۴۶۵۶۷۴ – ۶۶۹۵۳۸۸۳ فاکس: ۶۶۹۵۷۶۹۰

ایمیل : info@nasspub.com وب سایت :www.nasspub.com

فروشگاه: تهران – ضلع جنوب‌شرقی میدان انقلاب، شماره ۱۵۰۰/۳

تلفن: ۶۶۴۰۵۳۷۲

شابک: ۲-۲۵۹-۴۱۰-۹۶۴-۹۷۸ ISBN: 978-964-410-213-4

شابک دوره: ۸-۲۶۰-۴۱۰-۹۶۴-۹۷۸ ISBN:978-964-410-218-9

مقدمهٔ دکتر یحیی تابش

الگوریتم یکی از مهم‌ترین مفاهیم علوم عقلی و ابداعات بشری است. هزاران سال است که ریاضی‌دانان، فلاسفه و دانشمندان علوم مختلف بر تفکر ساختار یافته (یا تفکر الگوریتمی) در توسعه‌ی دانش اتفاق نظر داشته‌اند. در سده‌های اخیر با رشد روزافزون ریاضی و علوم طبیعی، الگوریتم به شاخه‌ای مستقل و بالنده از علوم تبدیل شده‌است. با ابداع ماشین‌های محاسبه و ظهور رایانه، الگوریتم باز هم نقش برجسته‌تری در کاربرد و تکوین علوم پیدا کرده است.

تدریس الگوریتم‌ها به عنوان یک درس پایه برای دانشجویان علوم نظری و کاربردی و کامپیوتر از اواسط سده‌ی بیستم در دانشگاه‌ها هم رواج یافت. به دنبال این موضوع، کتاب‌های متعددی با موضوع «الگوریتم‌ها» تألیف شد و در دسترس دانشگاهیان قرار گرفت. مؤلفین این کتاب‌ها افرادی توانا و شایسته بودند، و کتاب‌هایشان در زمان خود مورد استقبال دانشجویان و اساتید قرار گرفت. با ارائه کتاب «مقدمه‌ای بر الگوریتم‌ها» در سال ۱۹۹۰، نوشته‌ی کورمن و همکاران به سرعت توجه علاقه‌مندان به این اثر جلب شد. در مدت کوتاهی این کتاب به پرفروش‌ترین کتاب الگوریتم‌ها تبدیل شد و به عنوان مرجع درسی در بهترین دانشگاه‌های سراسر دنیا مورد استفاده قرار گرفت. در اکثر دانشگاه‌های ایران هم این کتاب جای خود را به عنوان مرجع دو درس «ساختمان داده‌ها و الگوریتم‌ها» و «طراحی و تحلیل الگوریتم‌ها» باز کرد، که به همین ضرورت از ویرایش‌های قبلی آن چند ترجمه به زبان فارسی صورت گرفته است.

کتابی که در دست دارید، ترجمه‌ای از آخرین ویرایش (ویرایش سوم) کتاب زبان انگلیسی با سال انتشار ۲۰۰۹ می‌باشد. در ترجمه‌ی کتاب سعی شده که در عین دقت در انتقال درست و علمی مطلب، سادگی و رسایی نثر حفظ شود. انتخاب معادل‌ها و واژه‌های مناسب یکی از دغدغه‌های هر مترجم کتاب‌های تخصصی است، که مترجم این کتاب به خوبی از عهده‌ی آن برآمده است.

به علت ماهیت خاص کتاب، مطالب آن نسبتاً سنگین است و در نتیجه خواندن آن می‌تواند برای دانشجو خسته‌کننده باشد. به همین علت با ابتکار ناشر، کتاب به صورت دو رنگ چاپ شده تا به سهولت در خواندن کتاب کمک کند.

طی سال‌ها تدریس در دانشگاه صنعتی شریف، متوجه شده‌ام که کتاب خوب در ارتقاء سطح آموزش از اهمیت ویژه‌ای برخوردار است. امیدوارم این کتاب هم بتواند به نوبه‌ی خود سهمی در بهبود توان علمی کشور داشته باشد.

دکتر یحیی تابش

عضو هیئت علمی دانشگاه صنعتی شریف

مقدمه‌ی مترجم

از ابتدای پیدایش بشر، یکی از دغدغه‌های مهم انسان (و شاید مهم‌ترین آن‌ها) یافتن معنی، هدف، یا انگیزه‌ای برای زیستن روی کره‌ی خاکی بوده است. هدفی که تمامی ادیان بر روی آن تمرکز داشته‌اند، و متفکران مختلف در طول سالیان، راه‌کارهای گوناگونی برای دست‌یابی به آن ارائه کرده‌اند. چیزی که اکثر این ادیان و مکاتب در آن اتفاق نظر داشته‌اند، نقش خود فرد در تعیین هدف زندگی است، که از طرف بسیاری از پیروان این مکاتب نادیده گرفته می‌شود. هیچ مکتبی نمی‌تواند دستورالعملی جامع برای دست‌یابی تک‌تک افراد به تعالی ارائه دهد، چرا که هر کس در آفرینش منحصر به فرد است و باید نقشی متفاوت در این دنیا ایفا کند، و درک این نقش میسر نخواهد شد مگر از طریق تفکر و تعقل خود فرد (حقیقتی که تمامی ادیان الهی به آن تأکید دارند). وظیفه‌ی مکاتب فقط ارائه‌ی رویکرد کلی است، و در نهایت این شمایید که می‌توانید مسیر دقیق رشد خود را تعیین کنید.

حتماً تا به حال با افرادی مواجه شده‌اید که صرفاً به خاطر فشار اطرافیان، احساس ناتوانی یا دلایل دیگر، در زمینه‌ای تحصیل یا کار می‌کنند که به آن علاقه ندارند. این تلاش معمولاً نتیجه‌ای ندارد جز سرخوردگی و هدر رفتن استعدادهای درونی فرد. نیت از نگارش این مقدمه هم چیزی نیست جز واداشتن خودم و شما به لحظه‌ای تأمل. تأمل در مسیری که در پیش گرفته‌ایم و در آن گام برمی‌داریم. تأمل در مورد این که آیا این مسیر ما را به رشد و تعالی فردی نزدیک می‌کند یا نه. این که آیا تا کنون سعی کرده‌ایم به فلسفه‌ی وجودی خود پی ببریم، یا ما هم جزء افرادی هستیم که ناخواسته و بدون هدف در مسیری که دیگران برای ما تعیین کرده‌اند گام برمی‌داریم.

به هیچ وجه انتظار ندارم نظرات این حقیر را بپذیرید، بلکه فقط امیدوارم مشوقی باشم برای تفکر بیشتر در مورد راهی که تا کنون پیموده‌ایم و مسیری که پیش رو داریم، چرا که تفکر لازمه‌ی حصول آگاهی است و حرکت ناآگاهانه به مقصد درستی منتهی نخواهد شد. از این رو تقاضا دارم قبل از خواندن این کتاب (یا انجام هر کار دیگر) لحظه‌ای صادقانه در مورد اهدافی که در زندگی دارید بیاندیشید، و همواره سعی کنید فقط در راستای نیل به آن اهداف حرکت کنید. امیدوارم خواندن این کتاب برای شما گامی باشد در همین راستا.

+++

تمام دغدغه‌ی این‌جانب در برگردان این اثر، انتقال صحیح و روان مفاهیم ارائه شده در کتاب بوده است. در این راستا از ترجمه‌ی کلمه به کلمه به شدت پرهیز کرده، و نهایت تلاش خود را به کار برده‌ام که قبل از آغاز ترجمه‌ی هر بخش، ابتدا به دقت و با جزئیات تمام مطالب آن را مرور کنم تا بتوانم به درستی آن‌ها را به خواننده منتقل کنم. امیدوارم این تلاش نتیجه‌بخش بوده، و گامی باشد هر چند ناچیز در راستای آموزش علاقه‌مندان به مبحث الگوریتم.

در انتها از شما خواننده‌ی عزیز تقاضا دارم که در بهبود این اثر یاری خود را از ما دریغ نکنید. مسلماً این اثر بدون نقص نیست، ولی بهره‌مندی از دیدگاه‌های سازنده‌ی شما خوانندگان ما را به ارتقای سطح کیفی ترجمه در چاپ‌های آتی دلگرم می‌کند. در صورت مشاهده‌ی هر گونه خطا در نگارش یا ترجمه و یا داشتن هر گونه پیشنهاد یا انتقاد، لطفاً به آدرس‌های زیر ارسال فرمائید.

ali.dehghan.tarzeh@gmail.com
info@nasspub.com

علی دهقان طرزه
تابستان ۱۳۸۹

مقدمهٔ نویسندگان

مقدمه

قبل از به وجود آمدن کامپیوترها، الگوریتم‌ها وجود داشتند. اکنون که کامپیوترها به وجود آمده‌اند، وجود الگوریتم‌ها بسیار پررنگ‌تر از قبل شده است، چرا که در قلب محاسبات کامپیوتری قرار دارند.

کتاب حاضر، مقدمه‌ای است جامع برای آموزش مدرن الگوریتم‌های کامپیوتری. این کتاب الگوریتم‌های بسیاری معرفی و آن‌ها را به صورت عمیق بررسی می‌کند، ولی با این حال این معرفی طوری است که طراحی و تحلیل آن‌ها برای خوانندگان تمام سطوح قابل دسترس است. سعی شده است که بدون از دست رفتن عمق بررسی یا دقت ریاضی، توضیحات به صورت مقدماتی باشد تا برای تمام خوانندگان قابل فهم باشد.

هر فصل یک الگوریتم، یک تکنیک طراحی، یک حوزه‌ی کاربرد، یا یک موضوع مربوط را ارائه می‌کند. الگوریتم‌ها به زبان انگلیسی و به شکل سودوکد طراحی شده‌اند تا برای تمام کسانی که به صورت سطحی با برنامه‌نویسی آشنایی دارند، قابل فهم باشد. کل کتاب حاوی بیش از ۲۴۴ شکل است که نحوه‌ی اجرای الگوریتم‌ها را مشخص می‌کنند. از آن‌جایی که *کارایی* را به عنوان یک معیار طراحی می‌شناسیم، تحلیل دقیق زمان اجرای تمام الگوریتم‌ها را در کتاب گنجانده‌ایم.

ساختار متن کتاب طوری است که برای استفاده در واحدهای درسی الگوریتم‌ها یا ساختمان داده‌ها در مقاطع تحصیلی کارشناسی و یا کارشناسی ارشد قابل استفاده باشد. چون در کتاب بحث‌هایی در مورد مطالب مهندسی در طراحی الگوریتم‌ها، و همچنین جنبه‌های ریاضی وجود دارد، می‌توان از آن به عنوان خودآموز هم استفاده کرد.

برای آموزگاران

این کتاب طوری طراحی شده است که هم فراگیر باشد و هم کامل. برای درس‌های مختلفی این کتاب را مفید خواهید یافت، از یک واحد ساختمان داده‌ها در دوره‌ی کارشناسی تا یک واحد الگوریتم در دوره‌ی کارشناسی ارشد. از آن‌جایی که محتویات این کتاب بسیار بیشتر از مطالب مورد نیاز در یک درس در طول یک ترم است، باید به آن به صورت یک «قفسه» نگاه کنید، که می‌توانید مطالبی را که بیشترین همخوانی را با واحد مورد نظر شما دارد، انتخاب کرده و آموزش دهید.

احتمالا مرتب کردن فصول به طور دلخواه، به طوری که با درس شما متناسب باشد، برای شما کار ساده‌ای خواهد بود. فصول نسبتا مستقل هستند، و بنابراین نیازی نیست که نگران وجود وابستگی‌های غیر منتظره از فصلی به فصل دیگر باشید. در هر فصل، ابتدا مطالب ساده‌تر و سپس مطالب سخت‌تر گنجانده شده‌اند، به همراه بخش‌بندی‌هایی که مرز مطالب را مشخص می‌کنند. در یک درس در دوره‌ی کارشناسی، ممکن است بخواهید که فقط از بخش‌های اول یک فصل استفاده کنید، و در یک درس در دوره‌ی کارشناسی ارشد، ممکن است بخواهید تمام فصل را پوشش دهید.

کتاب شامل بیش از ۹۲۰ تمرین و بیش از ۱۴۰ مسئله است. هر بخش با چند تمرین، و هر فصل با چند مسئله به پایان می‌رسد. معمولا تمرین‌ها سؤال‌های کوتاهی هستند که تسلط اولیه بر روی مباحث را آزمایش می‌کنند. بعضی از آن‌ها تمرین‌های خودآموز ساده هستند، در حالی که بعضی دیگر مقداری محکم‌تر هستند، و می‌توان از آن‌ها به عنوان تکلیف استفاده کرد. مسئله‌ها بررسی‌های موردی ظریف‌تری هستند که معمولا مطالب جدیدی را معرفی می‌کنند؛ آن‌ها معمولا شامل چندین سؤال هستند که دانشجو را طی مراحل مورد نیاز برای رسیدن به یک جواب کلی راهنمایی می‌کند.

بخش‌ها و تمرین‌هایی که ستاره (★) دارند، بیشتر برای دانشجویان دوره‌ی کارشناسی ارشد مناسب هستند تا دانشجویان دوره‌ی کارشناسی. یک بخش ستاره‌دار لزوما مشکل‌تر از یک بخش بدون ستاره نیست، ولی ممکن است به درک مطالب پیشرفته‌تری نیاز داشته باشد. متشابها، تمرین‌های ستاره‌دار ممکن است به یک پیش‌زمینه‌ی قبلی و یا خلاقیت بالاتر از سطح متوسط احتیاج داشته باشند.

برای دانشجو

امیدواریم که این کتاب یک معرفی لذت‌بخش در زمینه‌ی الگوریتم‌ها برای شما فراهم کرده باشد. سعی کرده‌ایم که تمام الگوریتم‌ها را قابل فهم و جذاب توصیف کنیم. برای کمک به شما وقتی که به الگوریتم‌های مشکل یا ناآشنا بر می‌خورید، هر کدام از الگوریتم‌ها به صورت قدم به قدم توصیف شده است. همچنین توضیحات دقیقی از ریاضیات مورد نیاز برای تحلیل الگوریتم فراهم شده است. اگر از قبل با یک موضوع آشنایی دارید، ساختار فصل‌ها را طوری خواهید یافت که می‌توانید از بخش‌های مقدماتی صرف نظر کرده و به سرعت به بخش‌های پیش‌رفته‌تر برسید.

حجم این کتاب زیاد است، و احتمالا در کلاس شما فقط بخشی از مطالب آن پوشش داده خواهد شد. با این حال، سعی شده است که کتاب هم اکنون به عنوان یک کتاب درسی، و بعدا به عنوان یک مرجع ریاضی و یا یک راهنمایی مهندسی برای شما قابل استفاده باشد.

پیش‌زمینه‌های مورد نیاز برای این کتاب چیست؟

- باید مقداری تجربه‌ی برنامه‌نویسی داشته باشید. به خصوص، باید با رویه‌های بازگشتی و ساختمان‌های داده‌ی ساده مانند آرایه‌ها و لیست‌های پیوندی آشنا باشید.
- باید تا حدودی در اثبات کردن به وسیله‌ی استقرا مهارت داشته باشید. مقدار کمی از کتاب به پیش‌زمینه‌ای مقدماتی در حسابان نیاز دارد. غیر از آن، قسمت‌های I و VIII تمام تکنیک‌های ریاضی مورد نیاز را به شما آموزش می‌دهند.

برای کاربران

دامنه‌ی وسیع مطالب این کتاب، آن را به یک راهنمای مناسب بر روی الگوریتم‌ها تبدیل کرده است. از آن جایی که مطالب هر فصل نسبتا مستقل است، می‌توانید فقط بر روی مباحثی تمرکز کنید که مورد نیاز شما هستند.

اکثر الگوریتم‌هایی که در مورد آن‌ها بحث می‌کنیم، در عمل کاربردهای فراوانی دارند. بنابراین در کتاب به مسائل پیاده‌سازی و مهندسی هم اشاره شده است. معمولا برای الگوریتم‌های کمی که فقط کاربرد نظری دارند، جایگزین‌های کاربردی هم ارائه خواهیم کرد.

اگر می‌خواهید هر یک از الگوریتم‌ها را پیاده‌سازی کنید، ترجمه‌ی سودوکد به زبان برنامه‌نویسی مورد نظر خود را کاری نسبتا سرراست خواهید یافت. سودوکدها طوری طراحی شده‌اند که الگوریتم‌ها را به صورت واضح و مختصر شرح دهند. به همین دلیل در آن‌ها به مدیریت خطا (error-handling) و مسائل مهندسی نرم‌افزار که به فرض‌های خاص در مورد محیط برنامه‌نویسی نیاز دارند، اشاره نشده است. سعی شده است که توصیف هر الگوریتم ساده و مستقیم باشد تا خصوصیات منحصر به فرد یک زبان برنامه‌نویسی خاص، شفافیت آن را از بین نبرد.

فهرست مطالب

بخش پنجم

ساختمان‌های داده‌ی پیشرفته

شامل فصل‌های :

مقدمه

این بخش به بررسی ساختمان‌های داده‌ای باز می‌گردد که از مجموعه‌های پویا پشتیبانی می‌کنند، ولی با سطح پیشرفته‌تری از بخش ۳. به عنوان مثال، در دو تا از فصل‌ها استفاده‌ی گسترده‌ای از تکنیک‌های تحلیل سرشکن خواهد شد که در فصل ۱۷ دیدیم.

در فصل ۱۸ B–درخت‌ها معرفی خواهند شد، که درخت‌های جستجوی متوازنی هستند که به طور خاص برای ذخیره بر روی دیسک‌های مغناطیسی طراحی شده‌اند. چون دیسک‌های مغناطیسی بسیار کندتر از حافظه‌های با دسترسی تصادفی عمل می‌کنند، کارایی B–درخت‌ها را نه تنها با زمان محاسبات صرف شده توسط اعمال مجموعه‌ها، بلکه با میزان دسترسی‌های حافظه هم می‌سنجیم. برای هر عملیات B–درخت، تعداد دسترسی‌های حافظه با افزایش ارتفاع B–درخت افزایش می‌یابد، که اعمال B–درخت این ارتفاع را تا حد ممکن کم نگه می‌دارند.

در فصل ۱۹ پیاده‌سازی‌ای برای یک هرم قابل ادغام ارائه می‌شود، که از اعمال INSERT، MINIMUM، EXTRACT-MIN، و UNION پشتیبانی می‌کند.[1] عملیات UNION دو هرم را با یکدیگر ترکیب یا ادغام می‌کند. هرم‌های فیبوناچی – ساختمان‌های داده‌ی ارائه شده در این فصل ۱۹ – از اعمال DELETE و DECREASE-KEY هم پشتیبانی می‌کنند. برای اندازه‌گیری کارایی هرم‌های

[1] مانند مسئله‌ی ۱۰-۲، یک هرم قابل ادغام را طوری تعریف کرده‌ایم که از MINIMUM و EXTRACT-MIN پشتیبانی کند، و بنابراین می‌توانیم آن را *هرم کمینه‌ی قابل ادغام* بنامیم. از سوی دیگر، اگر هرم از MAXIMUM و EXTRACT-MAX پشتیبانی کند، آن را یک *هرم بیشینه‌ی قابل ادغام* می‌نامیم. در این کتاب، هرم‌های قابل ادغام، هرم‌های کمینه‌ی قابل ادغام خواهند بود، مگر این که خلاف آن گفته شود.

فیبوناچی از کران‌های سرشکن استفاده می‌کنیم. اعمال INSERT، MINIMUM، و UNION فقط به زمان $O(1)$ برای اجرا بر روی هرم‌های فیبوناچی نیاز دارند، و اعمال EXTRACT-MIN و DELETE در زمان سرشکن $O(\lg n)$ انجام می‌شوند. با این حال مهم‌ترین مزیت هرم‌های فیبوناچی این است که عملیات DECREASE-KEY در حالت سرشکن فقط به زمان $O(1)$ نیاز دارد. زمان سرشکن عملیات DECREASE-KEY بر روی هرم‌های فیبوناچی دلیل اصلی این است که هرم‌های فیبوناچی عناصر اصلی بعضی از سریع‌ترین الگوریتم‌های کنونی برای مسائل گراف هستند.

با توجه به این که وقتی کلیدها، اعداد صحیحی در یک بازه‌ی محدود هستند، می‌توانیم کران پایین $\Omega(n \lg n)$ را برای مرتب‌سازی بشکنیم، فصل ۲۰ می‌پرسد که آیا می‌توانیم یک ساختمان داده طراحی کنیم که وقتی کلیدها در یک بازه‌ی محدود هستند، از اعمال مجموعه‌های پویای SEARCH، INSERT، DELETE، MINIMUM، MAXIMUM، SUCCESSOR، و PREDECESSOR در زمان $o(\lg n)$ پشتیبانی کند؟ معلوم خواهد شد که با استفاده از یک ساختمان داده‌ی بازگشتی با نام درخت‌های van Emde Boas می‌توان به این زمان اجرا دست یافت. اگر کلیدها، اعداد صحیح یکتا از مجموعه‌ی $\{0,1,2,\ldots,u-1\}$ باشند، که در آن u توانی از ۲ است، آن گاه درختان van Emde Boas از هر یک از اعمال بالا در زمان $O(\lg\lg u)$ پشتیبانی می‌کنند.

نهایتاً، فصل ۲۱ ساختمان‌های داده برای مجموعه‌های منفصل را معرفی می‌کند. مجموعه‌ای جهانی از n عنصر داریم که در مجموعه‌های پویا جمع آوری شده‌اند. در ابتدا هر عنصر متعلق به مجموعه‌ی خاص خود است. عملیات UNION دو مجموعه را با یکدیگر متحد می‌کند، و FIND-SET تعیین می‌کند که یک عنصر داده شده در آن لحظه در کدام یک از این مجموعه‌ها است. با نمایش هر مجموعه به وسیله‌ی یک درخت ریشه‌دار ساده، می‌توانیم به اعمال به شدت سریعی دست پیدا کنیم: دنباله‌ای از m عملیات در زمان $O(m\alpha(n))$ انجام می‌شود، که در آن $\alpha(n)$ تابعی است که به شدت کند رشد می‌کند — $\alpha(n)$ در هر کاربرد قابل دستیابی حداکثر ۴ است. تحلیل سرشکنی که این کران زمانی را اثبات می‌کند به همان اندازه‌ای پیچیده است که این ساختمان داده ساده است.

مباحث پوشش داده شده در این بخش به هیچ وجه تنها مثال‌های ساختمان‌های داده‌ی «پیشرفته» نیستند. ساختمان‌های داده‌ی پیشرفته شامل نمونه‌های زیر هم می‌شود:

- درختان پویا که Sleator و Tarjan آن‌ها را معرفی کردند و Tarjan بحث‌هایی بر روی آن‌ها انجام داد. این ساختمان داده، جنگلی از درختان ریشه‌دار منفصل را نگه می‌دارد. هر یال در هر درخت دارای یک هزینه با مقدار حقیقی است. درختان پویا از جستجوهایی برای یافتن پدر گره‌ها، ریشه‌ها، هزینه‌ی یال‌ها، و یال با کم‌ترین هزینه در مسیر از یک گره به ریشه پشتیبانی می‌کنند. اداره کردن درخت‌ها ممکن است به وسیله‌ی قطع کردن یال‌ها، مقدار دهی مجدد تمام یال‌ها در مسیر یک گره به ریشه‌ی درخت، متصل کردن یک ریشه به درختی دیگر، و یا تبدیل یک گره به ریشه‌ی درخت مربوطه انجام شود. یک پیاده‌سازی از درختان پویا از هر یک از اعمال بر روی ساختمان داده در زمان سرشکن $O(\lg n)$ پشتیبانی می‌کند؛ یک پیاده‌سازی

پیچیده‌تر دارای کران‌های $O(\lg n)$ در بدترین حالت است. از درختان پویا در بعضی از سریع‌ترین الگوریتم‌های شار شبکه (network-flow) استفاده می‌شود.

- *درختان Splay*، که Sleator و Tarjan آن‌ها را توسعه دادند و Tarjan بر روی آن‌ها بحث کرد، نوعی از درختان جستجوی دودویی هستند که اعمال استاندارد درختان جستجوی دودویی بر روی آن‌ها در زمان سرشکن $O(\lg n)$ اجرا می‌شوند. یکی از کاربردهای درختان splay، فراهم کردن نوع ساده‌ای از درختان پویا است.
- *ساختمان‌های داده‌ی پایدار* امکان انجام انواع جستجوها، و بعضی مواقع تغییر را برای دیگر ساختمان‌های داده فراهم می‌کنند. Driscoll، Sarnak، Sleator و Tarjan تکنیک‌هایی فراهم کرده‌اند که به کمک آن‌ها می‌توان با هزینه‌ی زمانی و حافظه‌ای کم ساختمان‌های داده‌ی متصل را پایدار کرد. مسئله‌ی ۱۳-۱ مثال ساده‌ای از یک مجموعه‌ی پویای پایدار ارائه می‌کند.
- مانند فصل ۲۰، ساختمان‌های داده‌ی متعددی امکان پیاده‌سازی سریع‌تری از اعمال دیکشنری (INSERT، DELETE، و SEARCH) را برای مجموعه‌ی جهانی محدودی از کلیدها فراهم می‌کنند. این ساختمان‌های داده با استفاده از این محدودیت‌ها قادر هستند که به بدترین حالت زمان اجرای حدی بهتری از ساختمان‌های داده‌ی مقایسه‌ای دست یابند. Fredman و Willard *درختان هم جوش* (fusion tree) را معرفی کردند، که اولین ساختمان داده‌ای است که وقتی مجموعه‌ی جهانی به اعداد صحیح محدود شده است، از اعمال دیکشنری سریع‌تر پشتیبانی می‌کند. آن‌ها نشان دادند که چگونه می‌توان این اعمال را در زمان $O(\lg n/\lg\lg n)$ پیاده‌سازی کرد. بسیاری از ساختمان‌های داده‌ی بعد از آن هم، از جمله *درختان جستجوی نمایی* (exponential search trees) کران‌های بهبود یافته‌ای برای بعضی و یا همه‌ی اعمال دیکشنری ارائه کردند.
- *ساختمان‌های داده‌ی پویای گراف* از جستجوهای مختلفی پشتیبانی می‌کنند، در حالی که اجازه می‌دهند که ساختار گراف تغییر کند، از طریق اعمالی که می‌توانند گره یا یالی را درج یا حذف کنند. مثال‌هایی از جستجوهایی که از آن‌ها پشتیبانی شده است عبارتند از اتصال رأس، اتصال یال، درختان پوشای کمینه، اتصال دوجهته، و بستار تراگذار (transitive closure).

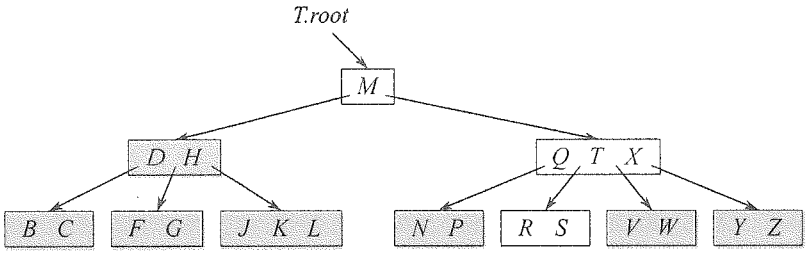
B- درخت‌ها

۱۸-۰ مقدمه

B- درخت‌ها، درختان جستجوی متوازنی هستند که برای کارایی بهتر بر روی دیسک‌های مغناطیسی و دیگر حافظه‌های ثانویه (secondary storage device) با دسترسی مستقیم طراحی شده‌اند. B- درخت‌ها شبیه درختان قرمز- سیاه (فصل ۱۳) هستند، ولی برای کمینه کردن اعمال ورودی/خروجی دیسک کارایی بیشتری دارند. بسیاری از سیستم‌های پایگاه داده‌ای از B- درخت‌ها یا ساختارهای مشابه برای ذخیره‌ی اطلاعات استفاده می‌کنند.

تفاوت B- درخت‌ها با درختان قرمز-سیاه این است که گره‌های B- درخت‌ها می‌توانند فرزندان زیادی داشته باشند، از چند عدد تا چند هزار. یعنی «فاکتور انشعاب» یک B- درخت می‌تواند بسیار بزرگ باشد، و معمولاً به وسیله‌ی خصوصیات دیسک مورد استفاده تعیین می‌شود. B- درخت‌ها از این لحاظ به درختان قرمز-سیاه شبیه هستند که هر B- درخت با n گره دارای ارتفاع $O(\lg n)$ است، هر چند ارتفاع یک B- درخت می‌تواند بسیار کم‌تر از ارتفاع یک درخت قرمز-سیاه باشد چرا که می‌تواند فاکتور انشعاب بسیار بزرگ‌تری داشته باشد. بنابراین B- درخت‌ها هم می‌توانند بسیاری از اعمال مجموعه‌های پویا را در زمان $O(\lg n)$ انجام دهند.

B- درخت‌ها حالت کلی‌تری از درختان جستجوی دودویی هستند. شکل ۱۸-۱ نمونه‌ی ساده‌ای از یک B- درخت را نشان می‌دهد. اگر یک گره‌ی داخلی x در یک B- درخت دارای $x.n$ کلید باشد، آن گاه $x.n+1$ فرزند خواهد داشت. از کلیدهای گره‌ی x به عنوان نقاط جدا کننده‌ای برای دامنه‌ی کلیدهای زیرگره‌های x به $x.n+1$ زیردامنه استفاده می‌شود، که هر یک از این زیردامنه‌ها توسط یکی از فرزندان x اداره می‌شوند. وقتی در یک B- درخت برای کلیدی خاص جستجو می‌کنیم، یک تصمیم $(x.n+1)$جهته بر مبنای مقایسه‌هایی با $x.n$ کلید ذخیره شده در x می‌گیریم. ساختار برگ‌ها با ساختار گره‌های داخلی متفاوت است؛ این تفاوت‌ها را در بخش ۱۸-۱ خواهیم دید.



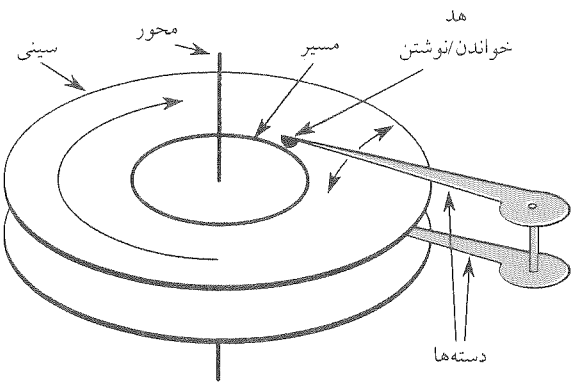
شکل ۱۸-۱ یک B-درخت که کلیدهای آن حروف الفبای انگلیسی هستند. یک گره‌ی داخلی حاوی $x.n$ کلید، $x.n+1$ فرزند دارد. تمام برگ‌ها در درخت عمق یکسانی دارند. گره‌های با سایه‌ی کم‌رنگ آن‌هایی هستند که در یک جستجو برای حرف R بررسی شده‌اند.

در بخش ۱۸-۱ تعریفی دقیق از B-درخت‌ها ارائه می‌شود، و اثبات می‌شود که ارتفاع یک B-درخت بر حسب تعداد کلیدهای درون آن به صورت لگاریتمی افزایش می‌یابد. در بخش ۱۸-۲ توضیح داده می‌شود که چطور می‌توان در یک B-درخت برای یک کلید جستجو کرد و یا گره‌ای در آن درج کرد، و در بخش ۱۸-۳ در مورد حذف بحث خواهد شد. ولی قبل از ادامه، باید به این سؤال جواب دهیم که چرا روش ارزیابی ساختمان‌های داده‌ای که برای کار بر روی دیسک‌های مغناطیسی طراحی شده‌اند با روش ارزیابی ساختمان‌های داده‌ای که برای کار بر روی حافظه‌های اصلی با دسترسی تصادفی طراحی شده‌اند متفاوت است.

ساختمان‌های داده بر روی حافظه‌ی ثانویه

تکنولوژی‌های مختلفی برای فراهم‌سازی فضای حافظه بر روی سیستم‌ها کامپیوتری موجود است. *حافظه‌ی اولیه* (primary memory) (یا *حافظه‌ی اصلی* (main memory)) یک کامپیوتر معمولاً از بردهای حافظه‌ی سیلیکونی تشکیل شده است. این تکنولوژی معمولاً در هر بیت دو رده گران‌تر از تکنولوژی حافظه‌ی مغناطیسی، مانند دیسک‌ها یا نوارها است. اکثر کامپیوترها دارای *حافظه‌ی ثانویه* (secondary storage) هم هستند که بر مبنای دیسک‌های مغناطیسی است؛ میزان چنین حافظه‌ای معمولاً از نظر مقدار دو رده بیشتر از حافظه‌ی اولیه است.

شکل ۱۸-۲ یک دیسک‌خوان معمولی را نشان می‌دهد. این دیسک‌خوان شامل چندین سینی (platter) است، که با سرعت ثابت دور یک *محور* (spindle) مشترک می‌چرخند. سطح هر سینی با ماده‌ای پوشانده شده است که قابل آهنربا شدن است. خواندن/نوشتن از/بر روی سینی‌ها به وسیله‌ی یک *هد* (head) در انتهای یک *دسته* (handle) انجام می‌شود. دسته‌ها می‌توانند هدهای خود را از محور دور و یا به آن نزدیک کنند. وقتی یک هد بی‌حرکت است، به سطحی که از زیر آن عبور می‌کند یک *مسیر* (track) گفته می‌شود. سینی‌های متعدد فقط می‌توانند حجم دیسک را افزایش دهند، و نه قابلیت‌های آن را.



شکل ۱۸-۴ (الف) یک دیسک‌خوان معمولی. این دیسک‌خوان از سینی‌های متعددی تشکیل شده است که دور یک محور می‌چرخند. خواندن/نوشتن از/بر روی هر سینی به وسیله‌ی یک هد در انتهای یک دسته انجام می‌شود. دسته‌ها به یکدیگر متصل شده‌اند تا هدهای خود را به صورت مشترک حرکت دهند. در این جا دسته‌ها به دور یک محور مشترک می‌چرخند. یک مسیر سطحی است که وقتی هد خواندن/نوشتن ثابت است، از زیر آن عبور می‌کند.

با این که دیسک‌ها ارزان‌تر هستند و ظرفیت بیشتری از حافظه‌ی اصلی دارند، بسیار بسیار کندترند، چرا که از بخش‌های مکانیکی متحرک تشکیل شده‌اند.[1] دو عنصر حرکتی مکانیکی در دیسک‌ها وجود دارد: چرخش سینی‌ها و حرکت دسته‌ها. در زمان نوشتن این کتاب سرعت دیسک‌های موجود در بازار ۵۴۰۰–۱۵۰۰۰ دور بر دقیقه (RPM) است. معمولاً سرعت ۱۵،۰۰۰ دور بر دقیقه را در درایوهای کلاس سرور، ۷۲۰۰ دور بر دقیقه را در درایوهای کامپیوترهای رومیزی، و ۵۴۰۰ دور بر دقیقه را در درایوهای لپ‌تاپ‌ها می‌بینیم. با این که ۷۲۰۰ دور بر دقیقه ممکن است سریع به نظر برسد، هر چرخش به ۸٫۳۳ میلی ثانیه نیاز دارد که تقریباً ۵ رده بیشتر از ۵۰ نانو ثانیه‌ی مربوط به زمان دسترسی معمول برای حافظه‌های سیلیکونی است. به عبارت دیگر اگر مجبور باشیم برای دسترسی به یک عنصر خاص در حافظه منتظر باشیم که هد خواندن/نوشتن یک دور کامل بچرخد، در این زمان می‌توانیم تقریباً ۱۰۰،۰۰۰ بار به حافظه‌ی اصلی دسترسی پیدا کنیم! به طور متوسط باید فقط منتظر یک چرخش نیمه باشیم، ولی باز هم تفاوت زمان دسترسی برای حافظه‌های سیلیکونی در مقابل دیسک‌ها هنگفت است. حرکت دادن دسته‌ها هم به زمان نیاز دارد. در زمان نوشتن این کتاب زمان متوسط دسترسی برای دیسک‌های موجود در بازار در بازه‌ی ۸ تا ۱۱ میلی ثانیه است.

برای سرشکن کردن زمان انتظار صرف شده برای حرکات مکانیکی، دیسک‌ها در هر لحظه دسترسی‌های متعددی به عناصر حافظه انجام می‌دهند. اطلاعات به تعدادی *صفحه‌های* (page) هم اندازه از بیت‌ها تقسیم شده‌اند که به صورت پشت سر هم در سیلندرها ظاهر می‌شوند، و خواندن/نوشتن هر دیسک از/بر روی یک یا چند صفحه‌ی کامل انجام می‌شود. برای یک دیسک معمولی،

[1] در زمان نوشتن این کتاب درایوهای ثابت به بازار آمده‌اند. با این که این درایوها سریع‌تر از درایوهای مکانیکی هستند، قیمت بیشتری بر هر گیگابایت، و ظرفیت کم‌تری دارند.

طول یک صفحه ممکن است 2^{11} تا 2^{14} بایت باشد. وقتی که هد خواندن/نوشتن به درستی در مکان مورد نظر قرار گرفت و چرخش دیسک برای دسترسی به ابتدای صفحه‌ی مورد نظر انجام شد، خواندن یا نوشتن بر روی دیسک مغناطیسی کاملاً الکترونیکی است (جدا از چرخش دیسک)، و مقدار زیادی داده می‌تواند به سرعت خوانده یا نوشته شود.

معمولاً زمان دسترسی به صفحه‌ای از اطلاعات و خواندن آن از روی دیسک بیشتر از زمان بررسی تمام آن اطلاعات توسط کامپیوتر است. به این دلیل، در این فصل به طور جداگانه نگاهی می‌اندازیم به دو عنصر اصلی زمان اجرا:

- تعداد دسترسی‌ها به دیسک، و
- زمان CPU (محاسبه).

تعداد دسترسی‌های دیسک بر مبنای تعداد صفحه‌هایی که باید بر روی آن‌ها نوشته و یا از روی آن‌ها خوانده شود، ارزیابی می‌شود. توجه کنید که زمان دسترسی به دیسک ثابت نیست - این زمان به فاصله‌ی مسیر فعلی از مسیر هدف، و همچنین وضعیت چرخشی اولیه‌ی دیسک بستگی دارد. با این حال از تعداد صفحه‌های خوانده یا نوشته شده به عنوان یک تقریب درجه‌ی اول برای کل زمان مصرف شده برای دسترسی به دیسک استفاده می‌کنیم.

در یک کاربرد معمولی B- درخت، میزان داده‌های اداره شده آن قدر زیاد است که تمام داده‌ها را نمی‌توان در حافظه‌ی اصلی جای داد. الگوریتم‌های B- درخت در زمان نیاز صفحه‌های منتخب را از دیسک به حافظه‌ی اصلی کپی می‌کنند، و صفحه‌هایی را که تغییر کرده‌اند دوباره در دیسک کپی می‌کنند. الگوریتم‌های B- درخت طوری طراحی شده‌اند که در هر زمان فقط تعداد ثابتی از صفحه‌ها در حافظه‌ی اصلی باشند؛ بنابراین اندازه‌ی حافظه‌ی اصلی محدودیتی برای اندازه‌ی B-درختی که باید اداره شود ایجاد نمی‌کند.

اعمال دیسک را در شبه‌کد به صورت زیر مدل می‌کنیم. فرض کنید x اشاره‌گری به یک شیئ باشد. اگر این شیئ در همان لحظه در حافظه‌ی اصلی کامپیوتر قرار داشته باشد، آن گاه می‌توانیم به شکل معمول به فیلدهای آن شیئ دسترسی پیدا کنیم: به عنوان مثال $x.key$. ولی اگر شیئ مربوط به x بر روی دیسک باشد، آن گاه قبل از این که بتوانیم به فیلدهای x دسترسی داشته باشیم باید رویه‌ی DISK-READ (x) را فراخوانی کنیم تا x در حافظه کپی شود. (فرض می‌کنیم اگر x در حافظه‌ی اصلی باشد آن گاه DISK-READ (x) هیچ دسترسی به دیسک انجام نمی‌دهد.) به طور مشابه از عملیات DISK-WRITE (x) برای ذخیره‌ی تغییراتی که بر روی فیلدهای شیئ x انجام شده است، استفاده می‌شود. یعنی الگوی معمول برای کار با یک شیئ به صورت زیر است:

- $x =$ یک اشاره‌گر به یک شیئ
- DISK-READ (x)
- انجام اعمالی که به فیلدهای x دسترسی پیدا می‌کنند و/یا آن‌ها را تغییر می‌دهند
- DISK-WRITE (x) // اگر هیچ فیلدی از x تغییر نکرده باشد، از این مرحله صرف نظر می‌شود
- انجام اعمال دیگری که به فیلدهای x دسترسی پیدا می‌کنند، ولی آن‌ها را تغییر نمی‌دهند

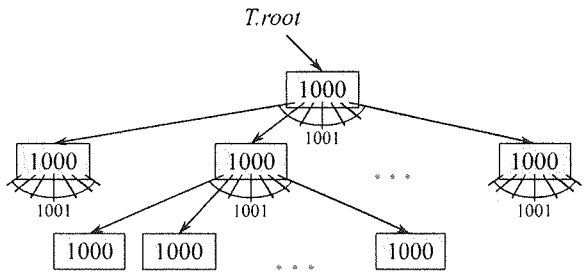
سیستم در هر زمان فقط می‌تواند تعداد محدودی از صفحه‌ها را در حافظه‌ی اصلی نگه دارد. فرض خواهیم کرد صفحه‌هایی که از آن‌ها استفاده نمی‌شود توسط سیستم از حافظه‌ی اصلی خارج می‌شوند؛ در این جا در الگوریتم‌های B- درخت از این مسئله صرف نظر می‌کنیم.

از آن‌جایی که در اکثر سیستم‌ها تعداد اعمال DISK-READ و DISK-WRITE زمان اجرای الگوریتم‌های B-درخت را تعیین می‌کند، کار عاقلانه‌ای است که برای استفاده‌ی بهینه از این اعمال، در هر بار استفاده به هر مقدار ممکن اطلاعات بنویسیم یا بخوانیم. بنابراین معمولاً یک گره‌ی B- درخت به بزرگی یک صفحه‌ی دیسک است، و تعداد فرزندانی که یک گره‌ی B-درخت می‌تواند داشته باشد با اندازه‌ی یک صفحه‌ی دیسک محدود می‌شود.

برای یک B- درخت بزرگ که بر روی دیسک ذخیره شده است معمولاً از فاکتور انشعاب ۵۰ تا ۲۰۰۰ استفاده می‌شود، که مقدار دقیق آن به اندازه‌ی هر کلید نسبت به اندازه‌ی یک صفحه بستگی دارد. فاکتور انشعاب زیاد به شدت ارتفاع درخت و تعداد دسترسی‌های دیسک مورد نیاز برای یافتن یک کلید را کاهش می‌دهد. شکل ۱۸-۳ یک B-درخت با فاکتور انشعاب ۱۰۰۱ و ارتفاع ۲ را نشان می‌دهد که می‌تواند بیش از یک میلیارد کلید را نگه دارد؛ با این حال از آن جایی که می‌توان ریشه را به صورت دائم در حافظه‌ی اصلی نگه داشت، برای یافتن یک کلید در این درخت در بیشترین حالت فقط به *دو* دسترسی به دیسک نیاز داریم!

۱-۱۸ تعریف B-درخت‌ها

برای سادگی، همان طور که برای درخت‌های جستجوی دودویی و درخت‌های قرمز- سیاه فرض کردیم، فرض می‌کنیم که هر گونه «اطلاعات پیرو» مربوط به کلید در همان گره‌ای ذخیره شده است که کلید در آن است. در عمل ممکن است به همراه هر کلید فقط اشاره‌گری به یک صفحه‌ی دیسک دیگر ذخیره شود که حاوی اطلاعات پیرو کلید است. شبه‌کدهای این فصل به صراحت فرض می‌کنند که هر گاه یک کلید از گره‌ای به گره‌ی دیگر جابه‌جا می‌شود، اطلاعات پیرو مربوط به کلید و یا اشاره‌گر این



شکل ۱۸-۳ یک B-درخت با ارتفاع دو و حاوی بیش از یک میلیارد کلید. تعداد کلیدهای هر گره‌ی x درون آن نشان داده شده است. هر گره‌ی داخلی و برگ حاوی ۱۰۰۰ کلید است. ۱۰۰۱ گره در عمق ۱ و بیش از یک میلیون برگ در عمق ۲ وجود دارد.

اطلاعات هم به همراه آن جابه‌جا می‌شود. یک نسخه‌ی معمول از B-درخت‌ها، معروف به B^+-درخت، تمام اطلاعات پیرو را در برگ‌ها ذخیره می‌کند و در گره‌های داخلی فقط کلیدها و اشاره‌گرهایی به فرزندان را نگه می‌دارد، و بدین صورت فاکتور انشعاب گره‌های داخلی را به مقدار بیشینه می‌رساند.

یک *B-درخت* T، یک درخت ریشه‌دار است (که ریشه‌ی آن $T.root$ است) با خصوصیات زیر:

۱. هر گره‌ی x فیلدهای زیر را دارد:

I. $x.n$، تعداد کلیدهایی که در حال حاظر در گره‌ی x ذخیره شده‌اند،

II. خود $x.n$ کلید، که به ترتیب صعودی مرتب شده‌اند، به طوری که $x.key_1 \le x.key_2 \le \ldots \le x.key_{x.n}$،

III. $x.leaf$، یک مقدار بولین که اگر x یک برگ باشد، TRUE و در غیر این صورت FALSE است.

۲. هر گره‌ی داخلی x به علاوه حاوی $n+1$ اشاره‌گر $x.c_1, x.c_2, \ldots, x.c_{x.n+1}$ به فرزندان خود است. برگ‌ها هیچ فرزندی ندارند، بنابراین فیلدهای c_i آن‌ها تعریف نشده است.

۳. کلیدهای $x.key_i$ دامنه‌ی کلیدهای ذخیره شده در هر زیردرخت را جدا می‌کنند: اگر k_iها کلیدهای زیردرخت با ریشه‌ی $x.c_i$ باشد، آن گاه

$$k_1 \le x.key_1 \le k_2 \le x.key_2 \le \ldots \le x.key_{x.n} \le k_{x.n+1}$$

۴. عمق تمام برگ‌ها یکسان، و برابر با ارتفاع درخت است.

۵. کران‌های بالا و پایینی برای تعداد کلیدهایی که یک گره می‌تواند داشته باشد، وجود دارد. این کران‌ها را می‌توان بر حسب یک عدد صحیح ثابت $t \ge 2$ با نام *درجه‌ی کمینه‌ی* B- درخت بیان کرد:

I. هر گره غیر از ریشه باید حداقل $t-1$ کلید داشته باشد. بنابراین هر گره‌ی داخلی غیر از ریشه حداقل t فرزند دارد. اگر درخت ناتهی باشد ریشه باید حداقل یک کلید داشته باشد.

II. هر گره حداکثر می‌تواند $2t-1$ کلید داشته باشد. بنابراین هر گره‌ی داخلی حداکثر می‌تواند $2t$ فرزند داشته باشد. می‌گوییم یک گره **پر** است اگر دقیقاً $2t-1$ کلید داشته باشد.[1]

ساده‌ترین B- درخت زمانی به وجود می‌آید که داشته باشیم $t = 2$. در این صورت هر گره‌ی داخلی ۲، ۳، یا ۴ فرزند دارد، و درخت ما یک *درخت ۲-۳-۴* خواهد بود. با این حال در عمل از مقادیر بسیار بزرگ‌تری برای t استفاده می‌شود.

---

[1] یک نسخه‌ی معمول دیگر از B-درخت‌ها معروف به B^*-درخت‌ها، نیاز دارد که هر گره‌ی داخلی به جای نیمه پر حداقل ۲/۳ پر باشد.

ارتفاع یک B-درخت

برای اکثر اعمال بر روی B-درخت‌ها تعداد دسترسی‌ها به دیسک متناسب است با ارتفاع B-درخت. در این جا ارتفاع بدترین حالت یک B-درخت را تحلیل خواهیم کرد.

قضیه ۱-۱۸ اگر $n \geq ۱$، آن گاه برای هر B-درخت T با n گره و ارتفاع h و $t \geq ۲$ داریم:

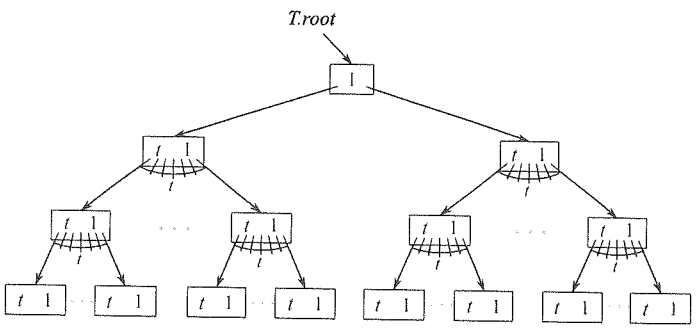
$$h \leq \log_t \frac{n+۱}{۲}$$

اثبات اگر ارتفاع یک B-درخت h باشد، ریشه حداقل یک کلید و بقیه‌ی گره‌ها حداقل $t-۱$ کلید دارند. بنابراین حداقل ۲ گره در عمق ۱ وجود دارد، حداقل $۲t$ گره در عمق ۲، حداقل $۲t^۲$ گره در عمق ۳، و همین طور تا عمق h، که در آن حداقل $۲t^{h-۱}$ گره وجود دارد. شکل ۱۸-۴ چنین درختی را با عمق ۳ نشان می‌دهد. بنابراین n، تعداد کلیدها، نامساوی زیر را ارضا می‌کند:

$$\begin{aligned} n &\geq ۱ + (t-۱)\sum_{i=۱}^{h} ۲t^{i-۱} \\ &= ۱ + ۲(t-۱)\left(\frac{t^h - ۱}{t-۱}\right) \\ &= ۲t^h - ۱ \end{aligned}$$

با محاسبات ساده خواهیم داشت $t^h \leq (n+۱)/۲$. گرفتن لگاریتم با پایه‌ی t از دو طرف قضیه را اثبات خواهد کرد. ■

در این جا قدرت B-درخت‌ها را در مقایسه با درختان قرمز- سیاه می‌بینیم. با این که ارتفاع هر دو نوع درخت با $O(\lg n)$ رشد می‌کند (به خاطر بیاورید که t ثابت است)، برای B-درخت‌ها پایه‌ی لگاریتم می‌تواند بسیار بزرگ‌تر باشد. بنابراین B- درخت‌ها با فاکتوری حدود $\lg t$ نسبت به درختان



عمق	تعداد گره‌ها
0	1
1	2
2	$2t$
3	$2t^2$

شکل ۱۸-۴ یک B-درخت با ارتفاع ۳ حاوی کم‌ترین تعداد کلیدها. عدد نشان داده شده در هر گره‌ی x مقدار $x.n$ است.

قرمز- سیاه در تعداد گره‌های بررسی شده در اکثر اعمال صرفه‌جویی می‌کنند. از آن جایی که بررسی یک گره‌ی دلخواه در یک درخت معمولاً به یک دسترسی به دیسک نیاز دارد، تعداد دسترسی‌ها به دیسک به میزان قابل توجهی کاهش می‌یابد.

تمرین‌ها

۱-۱-۱۸ چرا درجه‌ی کمینه‌ی $t = 1$ مجاز نیست؟

۱-۱-۱۸ درخت شکل ۱-۱۸ برای چه مقادیری از t یک B-درخت مجاز است؟

۱-۱-۱۸ تمام B-درختان مجاز با درجه‌ی کمینه‌ی ۲ را نشان دهید که نشان‌دهنده‌ی مجموعه‌ی $\{1, 2, 3, 4, 5\}$ هستند.

۱-۱-۱۸ بیشینه‌ی تعداد کلیدهایی که در یک B-درخت با ارتفاع h می‌توانند ذخیره شود، به صورت تابعی از درجه‌ی کمینه (t) چیست؟

۱-۱-۱۸ ساختمان داده‌ای را توضیح دهید که از یک درخت قرمز- سیاه نتیجه می‌شود که در آن گره‌های سیاه تمام فرزندان قرمز خود را جذب کرده‌اند، و فرزندان آن‌ها را هم به عنوان فرزندان خود پذیرفته‌اند.

۲-۱۸ اعمال اولیه بر روی B-درخت‌ها

در این بخش جزئیات اعمال B-TREE-SEARCH، B-TREE-CREATE، و B-TREE-INSERT را توضیح خواهیم داد. در این رویه‌ها دو قرارداد را می‌پذیریم:

- ریشه‌ی B-درخت همیشه در حافظه‌ی اصلی است، به طوری که هیچ گاه برای ریشه به DISK-READ نیازی نداریم؛ با این حال در صورتی که ریشه تغییر کند برای آن به DISK-WRITE نیاز پیدا خواهیم کرد.
- هر گاه گره‌ای به عنوان پارامتر ارسال می‌شود، باید قبلاً یک عملیات DISK-READ برای آن انجام شده باشد.

تمام رویه‌هایی که در این جا ارائه خواهیم کرد الگوریتم‌های «یک گذری» هستند که به صورت از بالا به پایین از ریشه‌ی درخت عمل می‌کنند، بدون این که نیازی به بازگشت به بالا باشد.

جستجو در B-درخت‌ها

جستجو در یک B-درخت بسیار شبیه جستجو در یک درخت جستجو دودویی است، غیر از این که به جای انجام یک انتخاب شاخه‌ای دودویی، یا «دو جهته» در هر گره، یک انتخاب شاخه‌ای چند جهته بسته به تعداد فرزندان گره انجام می‌دهیم. به صورت دقیق‌تر در هر گره‌ی داخلی x، یک انتخاب شاخه‌ای $(x.n + 1)$ جهته انجام خواهیم داد.

B-TREE-SEARCH یک کلی‌سازی سرراست از رویه‌ی TREE-SEARCH تعریف شده برای درختان جستجو دودویی است. B-TREE-SERACH یک اشاره‌گر به x (گره‌ی ریشه‌ی یک زیردرخت) و یک کلید k برای جستجو در زیردرخت x را به عنوان پارامتر دریافت می‌کند. بنابراین فراخوانی سطح بالایی به صورت B-TREE-SEARCH $(T.root, k)$ خواهد بود. اگر k در B–درخت وجود داشته باشد، B-TREE-SEARCH جفت مرتب (y, i) را حاوی گره‌ی y و اندیس i بازمی‌گرداند، به طوری که $y.key_i = k$. در غیر این صورت مقدار NIL بازگردانده خواهد شد.

```
B-TREE-SEARCH(x, k)
1  i = 1
2  while i ≤ x.n and k > x.key_i
3      i = i + 1
4  if i ≤ x.n and k == x.key_i
5      return (x, i)
6  if x.leaf
7      return NIL
8  else DISK-READ(x.c_i)
9      return B-TREE-SEARCH(x.c_i, k)
```

با استفاده از یک رویه‌ی جستجو خطی، خطوط ۱–۳ کوچک‌ترین اندیس i را می‌یابند به طوری که $k \le x.k_i$، در غیر این صورت i را برابر با $x.n+1$ قرار می‌دهند. در خطوط ۴–۵ چک می‌شود که آیا کلید را پیدا کرده‌ایم یا نه، و اگر کلید را پیدا کرده باشیم رویه بازگشت می‌کند. در غیر این صورت خطوط ۶–۹ یا جستجو را به صورت ناموفق پایان می‌دهند (اگر x یگ برگ باشد)، و یا به صورت بازگشتی جستجو را در زیردرخت مناسب x انجام می‌دهند، البته بعد از انجام DISK-READ مورد نیاز بر روی آن فرزند.

شکل ۱۸–۱ عملیات B-TREE-SEARCH را مشخص می‌کند؛ گره‌های با سایه‌ی کم‌رنگ آن‌هایی هستند که در جستجو برای کلید R بررسی شده‌اند.

مانند رویه‌ی TREE-SEARCH برای درختان جستجو دودویی، گره‌های بررسی شده در جستجو یک مسیر پایینی را از ریشه‌ی درخت تشکیل می‌دهند. بنابراین تعداد صفحه‌های دیسک مورد دسترسی قرار گرفته در B-TREE-SEARCH برابر خواهد بود با $\theta(h) = \theta(\log_t n)$، که در آن h ارتفاع B–درخت و n تعداد کلیدهای درون B–درخت است. از آن جایی که $x.n \le 2t$، زمان صرف شده توسط حلقه‌ی while در خطوط ۲–۳ در هر گره $O(t)$، و کل زمان CPU برابر $O(th) = O(t \log_t n)$ خواهد بود.

ساختن یک B–درخت تهی

برای ساختن یک B–درخت T، ابتدا از یک B-TREE-CREATE برای ساختن یک گره‌ی ریشه‌ی تهی استفاده کرده و سپس B-TREE-INSERT را برای اضافه کردن کلیدهای جدید به درخت فراخوانی می‌کنیم. هر دوی این رویه‌ها از یک رویه‌ی کمکی ALLOCATE-NODE استفاده می‌کنند، که در زمان $O(1)$ یک صفحه‌ی دیسک را برای استفاده‌ی یک گره‌ی جدید اختصاص می‌دهد. می‌توانیم فرض

کنیم که گره‌ای که توسط ALLOCATE-NODE ساخته شده است به DISK-READ نیازی ندارد، چرا که هنوز هیچ اطلاعات مفیدی در آن ذخیره نشده است.

```
B-TREE-CREATE(T)
1  x = ALLOCATE-NODE()
2  x.leaf = TRUE
3  x.n = 0
4  DISK-WRITE(x)
5  T.root = x
```

B-TREE-CREATE به $O(1)$ عملیات دیسک و $O(1)$ زمان CPU نیاز دارد.

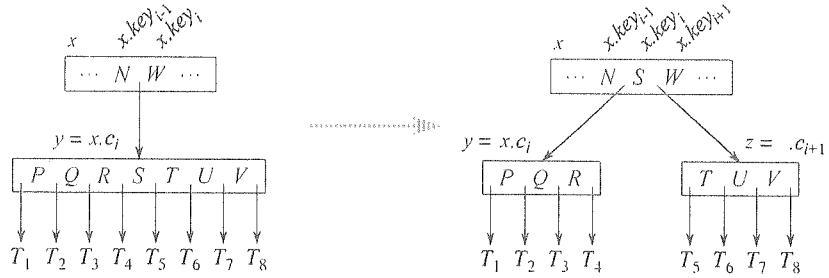
درج یک کلید جدید در B-درخت‌ها

درج یک کلید جدید در B-درخت‌ها به شدت پیچیده‌تر از درج یک کلید جدید در درختان جستجوی دودویی است. در درختان جستجوی دودوی، باید به دنبال جای یک برگ بگردیم که برای درج کلید مناسب باشد. با این حال در یک B-درخت نمی‌توانیم به سادگی یک برگ جدید بسازیم و آن را اضافه کنیم، چرا که درخت حاصل یک B-درخت مجاز نخواهد بود. در عوض کلید جدید را در یک برگ موجود درج می‌کنیم. از آن جایی که نمی‌توانیم یک کلید جدید را در برگی که پر است درج کنیم، یک عملیات جدید معرفی می‌کنیم که یک گره‌ی y پر (دارای $2t-1$ کلید) را حول $y.key_t$، *کلید میانه* (median key) آن به دو گره هر یک با $t-1$ کلید *تقسیم* (split) می‌کند. کلید میانه به سمت بالا حرکت کرده و در پدر y جای می‌گیرد تا نقطه‌ی جدا کننده‌ی دو درخت جدید را مشخص کند. ولی اگر پدر y هم پر باشد، آن هم باید قبل از درج کلید جدید تقسیم شود، و بنابراین تقسیم گره‌های پر می‌تواند تا ریشه‌ی درخت ادامه یابد.

مانند یک درخت جستجو دودویی، در یک B- درخت می‌توانیم درج یک کلید جدید را در یک گذر از ریشه به سمت پایین و تا یک برگ انجام دهیم. برای انجام این کار صبر نمی‌کنیم که ببینیم آیا واقعاً نیازی به تقسیم یک گره‌ی پر داریم یا نه. در عوض همان طور که در درخت به سمت پایین حرکت کرده و به دنبال مکان مناسب برای کلید می‌گردیم، هر گره‌ی پری را که در مسیر می‌یابیم، تقسیم می‌کنیم (شامل خود برگ یافت شده). بنابراین هر گاه می‌خواهیم یک گره‌ی پر y را تقسیم کنیم، مطمئن خواهیم بود که پدر آن پر نیست.

تقسیم یک گره در یک B-درخت

رویه‌ی B-TREE-SPLIT-CHILD به عنوان ورودی یک گره‌ی داخلی غیر پر x (با فرض این که این گره در حافظه‌ی اصلی قرار دارد)، یک اندیس i، و یک گره‌ی y (با فرض این که این گره هم در حافظه‌ی اصلی قرار دارد) را دریافت می‌کند، به طوری که $y = x.c_i$ یک فرزند پر x است. سپس رویه این فرزند را به دو قسمت تقسیم می‌کند و x را طوری اصلاح می‌کند که یک فرزند جدید داشته باشد. برای تقسیم یک ریشه‌ی پر، ابتدا ریشه را تبدیل می‌کنیم به فرزند یک ریشه‌ی جدید و تهی تا



شکل ۱۸-۵ تقسیم یک گره با $t = ۴$. گره‌ی $y = x.c_i$ به دو گره تقسیم می‌شود، y و z، و کلید میانه‌ی y (S) به سمت بالا جابه‌جا شده و در پدر y قرار می‌گیرد.

بتوانیم از B-TREE-SPLIT-CHILD استفاده کنیم. بنابراین ارتفاع درخت یکی افزایش می‌یابد؛ تقسیم کردن تنها کاری است که درخت به وسیله‌ی آن رشد می‌کند.

شکل ۱۸-۵ این روند را نشان می‌دهد. گره‌ی پر y حول کلید میانه‌ی آن (S) تقسیم می‌شود، که این کلید به سمت بالا حرکت کرده و در x، گره‌ی پدر y قرار می‌گیرد. کلیدهایی در y که از کلید میانه بزرگ‌تر هستند در گره‌ی جدید z قرار می‌گیرند، که فرزند جدید x خواهد بود.

```
B-TREE-SPLIT-CHILD(x, i)
1  z = ALLOCATE-NODE()
2  y = x.c_i
3  z.leaf = y.leaf
4  z.n = t – 1
5  for j = 1 to t – 1
6      z.key_j = y.key_{j+t}
7  if not y.leaf
8      for j = 1 to t
9          z.c_j = y.c_{j+t}
10 y.n = t – 1
11 for j = x.n + 1 downto i + 1
12     x.c_{j+1} = x.c_j
13 x.c_{i+1} = z
14 for j = x.n downto i
15     x.key_{j+1} = x.key_j
16 x.key_i = y.key_t
17 x.n = x.n + 1
18 DISK-WRITE(y)
19 DISK-WRITE(z)
20 DISK-WRITE(x)
```

B-TREE-SPLIT-CHILD با «بریدن و چسباندن» سرراست کار می‌کند. x گره‌ای است که تقسیم می‌شود، و y فرزند آن است (خط ۲ را ببینید). گره‌ی y در ابتدا $2t$ فرزند ($2t - 1$ کلید) دارد، ولی

توسط این عملیات فرزندان آن به t (۱ − t کلید) کاهش می‌یابد. t فرزند بزرگ (۱ − t کلید) y توسط z «پذیرفته می‌شوند»، و z فرزند جدید x می‌شود، که در جدول فرزندان x دقیقاً بعد از y قرار می‌گیرد. کلید میانه‌ی S به سمت بالا حرکت می‌کند تا تبدیل به کلیدی شود که y و z را جدا می‌کند.

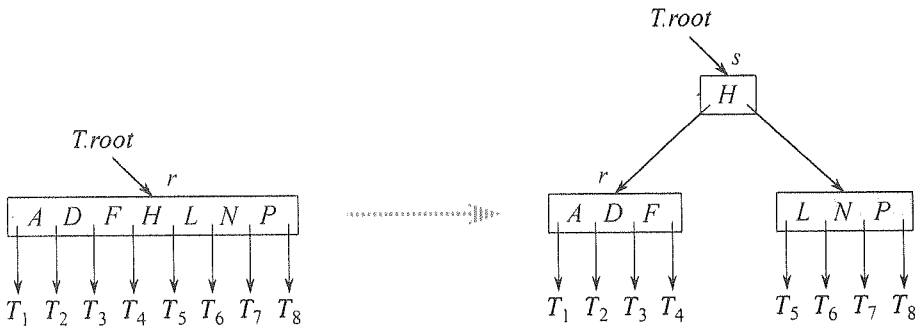
خطوط ۱-۹ گره‌ی z را می‌سازند و ۱ − t کلید بزرگ و t فرزند مربوطه در y را به آن می‌دهند. خط ۱۰ شمارنده‌ی کلیدها را در y اصلاح می‌کند. در نهایت خطوط ۱۱-۱۷ گره‌ی z را به عنوان یک فرزند x درج می‌کنند، کلید میانه را از y به بالا در x جابه‌جا می‌کنند تا y را از z جدا کند، و شمارنده‌ی کلیدهای x را اصلاح می‌کنند. در خطوط ۱۸-۲۰ تمام اصلاحات انجام شده در صفحات دیسک نوشته می‌شود. زمان CPU استفاده شده توسط B-TREE-SPLIT-CHILD برابر با $\theta(t)$ است، به دلیل حلقه‌های خطوط ۵-۶ و ۸-۹. (حلقه‌های دیگر $O(t)$ بار تکرار می‌شوند.) رویه، $O(1)$ عملیات دیسک انجام می‌دهد.

درج یک کلید در B-درخت در یک گذر به سمت پایین در درخت

یک کیلد k را در یک B-درخت T با ارتفاع h در یک گذر به سمت پایین در درخت درج می‌کنیم، که به $O(h)$ دسترسی به دیسک نیاز دارد. زمان CPU مورد نیاز $O(th) = O(t \log_t n)$ است. رویه‌ی B-TREE-INSERT از B-TREE-SPLIT-CHILD استفاده می‌کند تا تضمین کند که بازگشت هیچ گاه به یک گره‌ی پر نمی‌رسد.

```
B-TREE-INSERT(T, k)
1   r = T.root
2   if r.n == 2t - 1
3       s = ALLOCATE-NODE()
4       T.root = s
5       s.leaf = FALSE
6       s.n = 0
7       s.c1 = r
8       B-TREE-SPLIT-CHILD(s, 1)
9       B-TREE-INSERT-NONFULL(s, k)
10  else B-TREE-INSERT-NONFULL(r, k)
```

خطوط ۳-۹ حالتی را اداره می‌کنند که در آن گره‌ی r پر است: ریشه تقسیم می‌شود و یک گره‌ی جدید s (دارای دو فرزند) تبدیل به ریشه خواهد شد. تقسیم ریشه تنها راه افزایش ارتفاع یک B-درخت است. شکل ۱۸-۶ این حالت را مشخص می‌کند. بر خلاف یک درخت جستجوی دودویی، ارتفاع یک B-درخت از بالا اضافه می‌شود، نه از پایین. رویه با فراخوانی B-TREE-INSERT-NONFULL پایان می‌یابد تا درج کلید k در درخت با ریشه‌ی غیر پر جدید انجام شود. B-TREE-NONFULL به صورت بازگشتی به سمت پایین درخت حرکت می‌کند، و در هر زمان با فراخوانی B-TREE-SPLIT-CHILD در صورت نیاز، تضمین می‌کند که گره‌ای که به آن بازگشت انجام می‌شود پر نیست.



شکل ۱۸-۶ تقسیم گره با $t = ۴$. گره‌ی ریشه به دو قسمت تقسیم می‌شود، و یک گره‌ی ریشه‌ی جدید s ساخته می‌شود. ریشه‌ی جدید حاوی کلید میانه‌ی r است و دو قسمت r، فرزندان آن هستند. وقتی ریشه تقسیم می‌شود ارتفاع B-درخت یکی افزایش می‌یابد.

رویه‌ی بازگشتی کمکی B-TREE-INSERT-NONFULL کلید k را در گره‌ی x درج می‌کند، که فرض بر این است که این گره در زمان فراخوانی رویه پر نیست. عملیات B-TREE-INSERT و عملیات بازگشتی B-TREE-NONFULL تضمین می‌کنند که این فرض صحیح است.

```
B-TREE-INSERT-NONFULL(x, k)
1   i = x.n
2   if x.leaf
3       while i ≥ 1 and k < x.key_i
4           x.key_{i+1} = x.key_i
5           i = i − 1
6       x.key_{i+1} = k
7       x.n = x.n + 1
8       DISK-WRITE(x)
9   else while i ≥ 1 and k < x.key_i
10          i = i − 1
11      i = i + 1
12      DISK-READ(x.c_i)
13      if x.c_i.n == 2t − 1
14          B-TREE-SPLIT-CHILD(x, i)
15          if k > x.key_i
16              i = i + 1
17      B-TREE-INSERT-NONFULL(x.c_i, k)
```

رویه‌ی B-TREE-INSERT-NONFULL به صورت زیر کار می‌کند. خطوط ۳–۸ حالتی را اداره می‌کنند که در آن x یک برگ است و کلید k باید در آن درج شود. اگر x یک گره‌ی برگ نباشد، آن گاه باید k را در یک برگ مناسب در زیردرخت با ریشه‌ی گره‌ی داخلی x درج کنیم. در این حالت، خطوط ۹–۱۱ فرزندی از x را تعیین می‌کنند که بازگشت باید بر روی آن انجام شود. خط ۱۳ تشخیص می‌دهد که آیا بازگشت بر روی یک گره‌ی پر انجام می‌شود یا خیر، که در صورت لزوم در خط ۱۴ با استفاده از B-TREE-SPLIT-CHILD این فرزند به دوفرزند غیر پر تقسیم شود، و در خط

۱۵-۱۶ تعیین می‌شود که کدام یک از دو فرزند، فرزندی است که اکنون باید بازگشت بر روی آن انجام شود. (توجه کنید که بعد از این که در خط ۱۶ مقدار i افزایش می‌یابد، نیازی به DISK-READ $(x.c_i)$ نیست، چرا که در این حالت بازگشت بر روی فرزندی انجام خواهد شد که همان موقع توسط B-TREE-SPLIT-CHILD ساخته شده است.) بنابراین کل تأثیر خطوط ۱۳-۱۶ این است که تضمین کند بازگشت هیچ گاه بر روی یک گره‌ی پر انجام نمی‌شود. سپس خط ۱۷ به درج k در زیردرخت مناسب بازگشت می‌کند. شکل ۱۸-۷ حالت‌های مختلف درج یک کلید را در یک B-درخت نشان می‌دهد.

تعداد دسترسی‌های دیسک انجام شده توسط B-TREE-INSERT برای یک B-درخت با ارتفاع h برابر $O(h)$ است، چرا که بین فراخوانی‌های B-TREE-INSERT-NONFULL، تعداد $O(1)$ فراخوانی برای DISK-READ و DISK-WRITE انجام می‌شود. کل زمان CPU مصرف شده $O(th) = O(t \log_t n)$ است. از آن جایی که B-TREE-INSERT-NONFULL بازگشتی دنباله‌ای است می‌توان آن را به صورت یک حلقه‌ی while هم پیاده‌سازی کرد، که روشن می‌کند که تعداد صفحه‌هایی که در هر لحظه باید در حافظه‌ی اصلی باشد، $O(1)$ است.

تمرین‌ها

۱-۲-۱۸ نتیجه‌ی درج کلیدهای

$F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E$

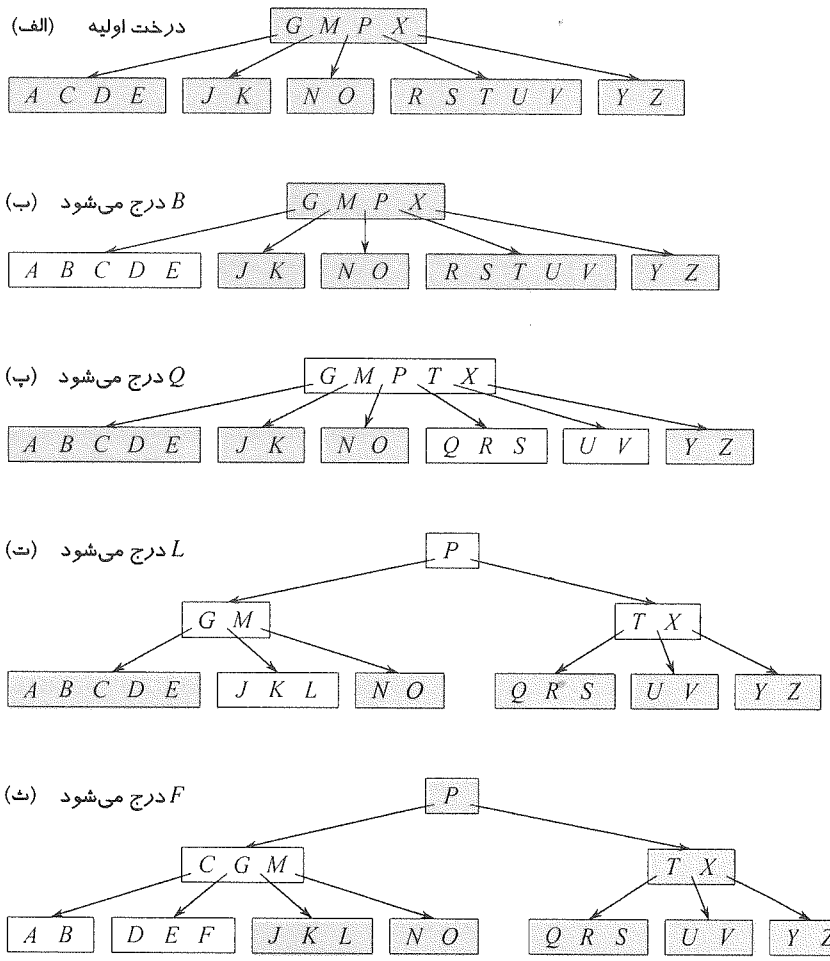
را به ترتیب در یک B-درخت تهی با درجه‌ی کمینه‌ی ۲ نشان دهید. شکل درخت را فقط قبل از تقسیم یک گره نشان دهید، به علاوه‌ی شکل نهایی درخت.

۲-۲-۱۸ توضیح دهید که تحت چه شرایطی حین اجرای B-TREE-INSERT (در صورت وجود) فراخوانی‌های اضافی از DISK-READ یا DISK-WRITE انجام می‌شود. (یک DISK-READ اضافی، یک DISK-READ برای صفحه‌ای است که در حافظه‌ی اصلی قرار دارد. یک DISK-WRITE اضافی صفحه‌ای را در دیسک می‌نویسد که اطلاعات آن دقیقاً مانند همانی است که در دیسک قرار دارد.)

۳-۲-۱۸ توضیح دهید که چگونه می‌توان کلید کمینه و کلید ماقبل یک کلید داده شده را در یک B-درخت یافت.

★ ۴-۲-۱۸ فرض کنید کلیدهای $\{1, 2, \ldots, n\}$ در یک B-درخت تهی با درجه‌ی کمینه‌ی ۲ درج شده‌اند. درخت نهایی چند گره دارد؟

۵-۲-۱۸ از آن جایی که برگ‌ها نیازی به اشاره‌گر برای فرزندان خود ندارند، می‌توانند از یک مقدار t متفاوت (بزرگ‌تر) از گره‌های داخلی برای صفحه‌های دیسک با همان اندازه استفاده کنند. نشان دهید چگونه می‌توان رویه‌های ساختن و درج درون یک B-درخت را برای اداره‌ی این تغییر، اصلاح کرد.



شکل ۱۸-۷ درج کلید در یک B-درخت. درجه‌ی کمینه‌ی t برای این B-درخت ۳ است، بنابراین یک گره می‌تواند حداکثر ۵ کلید نگه دارد. گره‌هایی که توسط رویه‌ی درج اصلاح شده‌اند با سایه‌ی کم‌رنگ مشخص شده‌اند. (الف) درخت اولیه‌ی این مثال. (ب) نتیجه‌ی درج B در درخت اولیه؛ این یک درج ساده در یک گره‌ی برگ است. (پ) نتیجه‌ی درج Q در درخت قبل. گره‌ی $RSTUV$ به دو گره حاوی RS و UV تقسیم شده است، کلید T به سمت بالا و به درون ریشه منتقل شده است، و Q در چپ‌ترین بخش دو قسمت (گره‌ی RS) درج شده است. (ت) نتیجه‌ی درج L در درخت قبل. ریشه در همان ابتدا تقسیم می‌شود، چرا که پر است و ارتفاع B-درخت یکی افزایش یافته است. سپس L در برگ حاوی JK درج شده است. (ث) نتیجه‌ی درج F در درخت قبل. قبل از درج F در سمت راست گره‌ی $ABCDE$، این گره تقسیم شده است.

۱۸-۲-۶ فرض کنید B-TREE-SEARCH طوری پیاده‌سازی می‌شود که به جای جستجوی خطی در هر گره، از جستجوی دودویی استفاده کند. نشان دهید که این تغییر زمان CPU مورد نیاز را به

$O(\lg n)$ کاهش می‌دهد، مستقل از این که t به صورت تابعی از n چگونه انتخاب می‌شود.

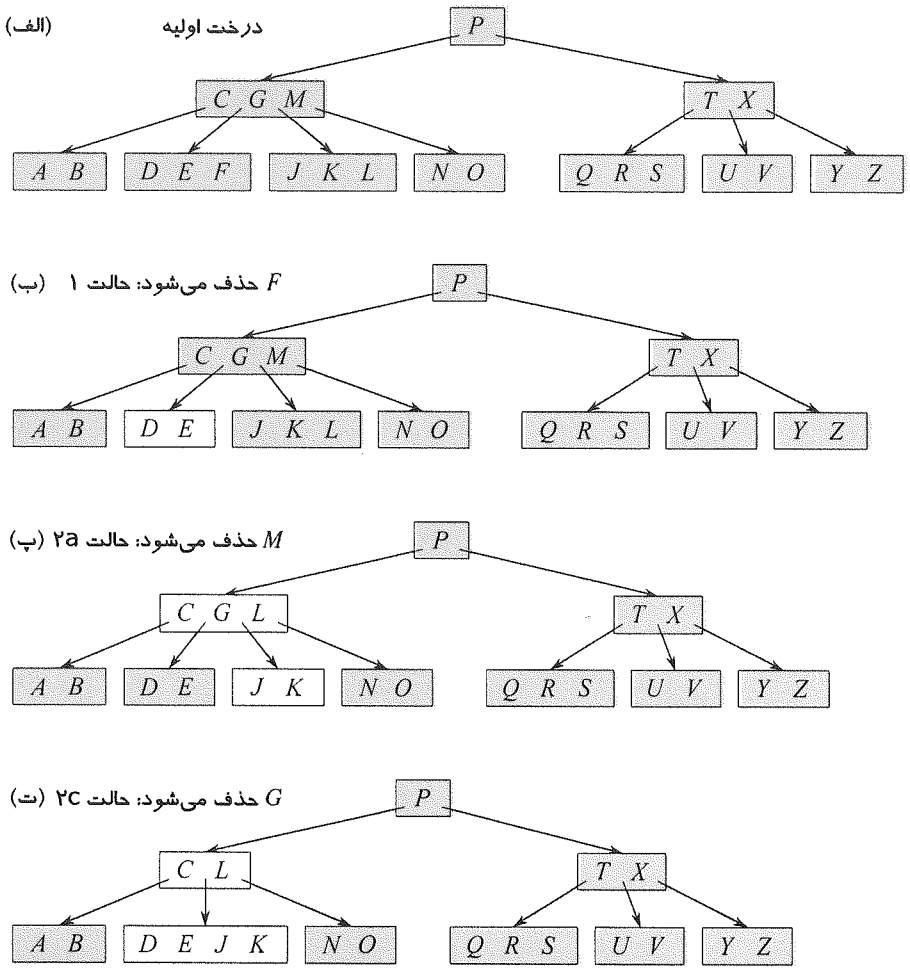
۱۸-۲-۷ فرض کنید یک سخت افزار دیسک به ما اجازه می‌دهد که اندازه‌ی صفحات دیسک را به صورت دلخواه انتخاب کنیم، ولی زمان خواندن یک صفحه‌ی دیسک $a+bt$ است، که در آن a و b ثابت‌های مشخص هستند و t درجه‌ی کمینه‌ی B-درختی است که از این صفحه‌ها با اندازه‌ی انتخاب شده استفاده می‌کند. توضیح دهید که چگونه باید t را انتخاب کنیم تا (تقریباً) زمان جستجوی B-درخت کمینه شود. یک مقدار بهینه برای t برای حالتی که در آن $a = ۵$ و $b = ۱۰$ پیشنهاد دهید.

۱۸-۳ حذف کلید از یک B-درخت

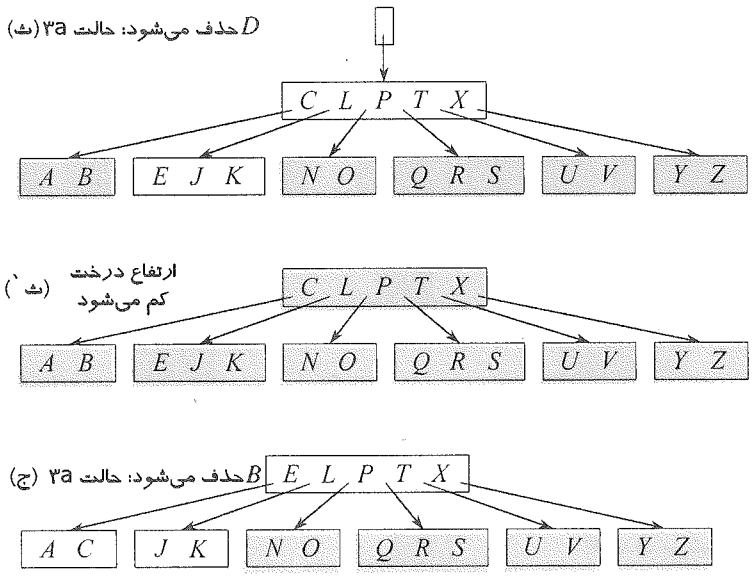
حذف از یک B-درخت مشابه درج در آن است، ولی مقداری پیچیده‌تر، چرا که یک کلید ممکن است از هر گره‌ای حذف شود – نه فقط از یک برگ – و حذف از یک گره‌ی داخلی به این نیاز دارد که فرزندان آن گره دوباره مرتب شوند. مانند درج، باید از درخت در مقابل حذف‌هایی که خصوصیات B-درخت‌ها را نقض می‌کنند، محافظت کنیم. مثل وقتی که باید مطمئن می‌شدیم با درج، گره‌ها بیش از اندازه بزرگ نمی‌شوند، باید مطمئن شویم با حذف گره‌ها بیش از اندازه کوچک نمی‌شوند (به غیر از ریشه که مجاز است کم‌تر از کمینه‌ی $t-۱$ کلید داشته باشد). مانند یک الگوریتم درج ساده که ممکن است در صورت پر بودن گره‌ای در مسیری که به مکان درج کلید ختم می‌شود، عقب‌نشینی کند، یک الگوریتم ساده‌ی حذف هم ممکن است در صورت قرار داشتن یک گره (غیر از ریشه) با کمینه‌ی تعداد کلیدها مجبور باشد عقب نشینی کند.

فرض کنید رویه‌ی B-TREE-DELETE بخواهد کلید k را از زیردرخت با ریشه‌ی x حذف کند. این رویه طوری سازمان‌دهی شده است که تضمین می‌کند هر گاه B-TREE-DELETE به صورت بازگشتی بر روی گره‌ی x فراخوانی می‌شود، تعداد کلیدهای x حداقل برابر است با درجه‌ی کمینه‌ی t. توجه کنید که این موقعیت به یک کلید بیشتر از تعداد کمینه‌ی کلیدهای یک گره نیاز دارد، چرا که بعضی مواقع ممکن است نیاز باشد قبل از رسیدن بازگشت به فرزندان، یک کلید به یکی از فرزندان منتقل شود. این شرط قوی شده به ما اجازه می‌دهد که در یک گذر به پایین و بدون نیاز به «عقب‌نشینی» یک کلید را حذف کنیم (البته با یک استثنا که بعداً توضیح داده خواهد شد). ویژگی‌های زیر مربوط به حذف از B-درخت‌ها باید با این درک تفسیر شوند که اگر ریشه‌ی x گره‌ای شود که هیچ کلیدی ندارد (این حالت ممکن است در حالت‌های III-۲ و II-۳ در زیر رخ دهد)، آن گاه x حذف می‌شود و تنها فرزند x ($c_۱[x]$) تبدیل به ریشه‌ی جدید درخت می‌شود، که در این حالت ارتفاع درخت یکی کاهش می‌یابد و این خصوصیت که ریشه‌ی درخت حداقل یک کلید دارد حفظ می‌شود (مگر این که درخت تهی باشد).

به جای ارائه‌ی شبه‌کد، بر روی شکل نشان می‌دهیم که حذف چگونه کار می‌کند. شکل ۱۸-۸ حالت‌های مختلف حذف یک کلید را از یک B-درخت نشان می‌دهد.



شکل ۱۸-۸ حذف کلید از یک B-درخت. درجه‌ی کمینه‌ی این B-درخت $t = ۳$ است، به طوری که یک گره (غیر از ریشه) نمی‌تواند کم‌تر از ۲ کلید داشته باشد. گره‌هایی که تغییر می‌کنند با سایه‌ی کم‌رنگ مشخص شده‌اند. (الف) B-درخت شکل ۱۸-۷(ث). (ب) حذف F. این حال ۱ است: حذف ساده از یک برگ. (پ) حذف M. این حالت I-۲ است: L، کلید ماقبل M به بالا منتقل می‌شود تا جای M را بگیرد. (ت) حذف G. این حالت III-۲ است: G به سمت پایین حرکت می‌کند تا گره‌ی $DEGJK$ را بسازد، و سپس G از این برگ حذف می‌شود (حالت ۱). (ث) حذف D. این حالت II-۳ است: بازگشت نمی‌تواند به سمت گره‌ی CL حرکت کند چرا که این گره فقط ۲ کلید دارد، بنابراین P به پایین منتقل و با CL و TX ترکیب می‌شود تا $CLPTX$ را تشکیل دهد؛ سپس D از یک گره حذف می‌شود (حالت ۱). (ث′) بعد از (ث)، ریشه حذف می‌شود و ارتفاع ریشه یکی کاهش می‌یابد. (ج) حذف B. این حالت I-۳ است: C جابه‌جا می‌شود تا جای B را پر کند و E جابه‌جا می‌شود تا جای C را پر کند.



شکل ۱۸-۸ (ادامه)

۱. اگر کلید k در گره‌ی x باشد و x یک برگ باشد، k را از x حذف می‌کنیم.

۲. اگر کلید k در گره‌ی x باشد و x یک گره‌ی داخلی باشد، کار زیر را انجام می‌دهیم.

I. اگر فرزند با برچسب y که در گره‌ی x قبل از k می‌آید، حداقل t کلید داشته باشد، آن گاه k'، کلید ماقبل k را در زیردرخت با ریشه‌ی y می‌یابیم. به طور بازگشتی k' را حذف می‌کنیم و در x کلید k را با k' جایگزین می‌کنیم. (یافتن k' و حذف آن می‌تواند در یک گذر پایینی انجام شود.)

II. اگر y کم‌تر از t کلید دارد، آن گاه به طور مشابه، فرزند با برچسب z را که در گره‌ی x بعد از k می‌آید بررسی می‌کنیم. اگر z حداقل t کلید داشته باشد، آن گاه k'، عنصر مابعد k را در زیردرخت با ریشه‌ی z می‌یابیم. به صورت بازگشتی k' را حذف می‌کنیم، و در x کلید k را با k' جایگزین می‌کنیم. (یافتن k' و حذف آن می‌تواند در یک گذر پایینی انجام شود.)

III. در غیر این صورت، اگر هر دوی y و z فقط $t-1$ کلید داشته باشند، k و تمام z را در y ادغام می‌کنیم، به طوری که x هر دوی k و اشاره‌گر z را از دست می‌دهد، و اکنون y حاوی $2t-1$ کلید خواهد بود. سپس z را آزاد کرده و به صورت بازگشتی k را از z حذف می‌کنیم.

۳. اگر کلید k در گره‌ی داخلی x موجود نباشد، $x.c_i$، ریشه‌ی درخت مناسب را که k باید در آن باشد، البته در صورت وجود، تعیین می‌کنیم. اگر $x.c_i$ فقط $t-1$ کلید داشته باشد، در صورت نیاز مرحله‌ی ۳-I یا ۳-II را اجرا می‌کنیم تا تضمین کنیم به سمت گره‌ای حرکت می‌کنیم که حداقل t کلید دارد. آن گاه با بازگشت بر روی فرزند مناسب x به کار پایان می‌دهیم.

I. اگر $x.c_i$ فقط $t-1$ کلید، ولی یک برادر دقیقاً مجاور با حداقل t کلید داشته باشد، با جابه‌جا

کردن یک کلید از x به $x.c_i$، منتقل کردن یک کلید از برادر مجاور سمت چپ یا راست $x.c_i$ به سمت بالا و x، و منتقل کردن اشاره‌گر فرزند مناسب از برادر $x.c_i$ به آن، به $x.c_i$ یک کلید اضافی می‌دهیم.

II. اگر $x.c_i$ و هر دو برادر مجاور $x.c_i$ دارای $t-1$ کلید باشند، $x.c_i$ را با یکی از برادرانش ادغام می‌کنیم، که شامل انتقال یک کلید از x به گره‌ی جدید ادغام شده می‌شود، تا این کلید تبدیل شود به میانه‌ی گره‌ی جدید.

از آن جایی که اکثر کلیدهای یک B–درخت در برگ‌ها هستند، ممکن است انتظار داشته باشیم که در عمل، عملیات حذف اکثراً برای حذف کلیدها از برگ‌ها انجام شود. در این صورت عملیات B-TREE-DELETE در یک گذر پایینی در درخت کار خود را انجام می‌دهد، بدون این که مجبور باشد عقب نشینی کند. با این حال هنگام حذف یک کلید از یک گره‌ی داخلی، رویه یک گذر پایینی بر روی درخت انجام می‌دهد ولی ممکن است مجبور باشد به گره‌ای که کلید از آن حذف شده بازگردد تا آن کلید را با کلید ماقبل یا مابعد آن جایگزین کند (حالت‌های ۲–I و ۲–II).

با این که این رویه ممکن است پیچیده به نظر برسد، فقط حاوی $O(h)$ عملیات دیسک برای یک B–درخت با ارتفاع h است، چرا که بین احضارهای بازگشتی رویه، فقط $O(1)$ فراخوانی برای DISK-READ و DISK-WRITE انجام می‌شود. زمان CPU مورد نیاز $O(th) = O(t \log_t n)$ است.

تمرین‌ها

۱–۳–۱۸ نتیجه‌ی حذف کلیدهای C، P، و V را، به ترتیب، از درخت شکل ۱۸–۸ نشان دهید.

۲–۳–۱۸ برای RB-DELETE یک شبه‌کد بنویسید.

مسائل

۱–۱۸ ***پشته‌ها بر روی حافظه‌ی ثانویه***

پیاده‌سازی یک پشته را بر روی یک کامپیوتر در نظر بگیرید که میزان حافظه‌ی اولیه و پرسرعت آن نسبتاً کم، و میزان حافظه‌ی ثانویه و کند آن نسبتاً زیاد است. اعمال PUSH و POP بر روی مقادیر یک کلمه‌ای پشتیبانی شده‌اند. پشته‌ای که می‌خواهیم از آن پشتیبانی کنیم می‌تواند بسیار بزرگ‌تر از مقدار حافظه‌ی اصلی شود، و بنابراین اکثر آن باید بر روی دیسک ذخیره شود.

یک پیاده‌سازی ساده ولی نابهینه‌ی پشته، کل پشته را در دیسک نگه می‌دارد. در حافظه یک اشاره‌گر پشته نگه می‌داریم که یک آدرس در دیسک به عنصر بالایی پشته است. اگر اشاره‌گر مقدار p داشته باشد، عنصر بالایی، $(p \bmod m)$امین کلمه در صفحه‌ی $\lfloor p/m \rfloor$ام دیسک

است، که در آن m تعداد کلمه‌ها در هر صفحه است.

برای پیاده‌سازی عملیات PUSH، اشاره‌گر پشته را افزایش می‌دهیم، صفحه‌ی مناسب را از دیسک خوانده و در حافظه می‌ریزیم، عنصری را که می‌خواهیم در پشته بنشانیم در کلمه‌ی مناسب در صفحه کپی می‌کنیم، و صفحه را دوباره در دیسک می‌نویسیم. عملیات POP هم مشابه است. اشاره‌گر پشته را کاهش می‌دهیم، صفحه‌ی مناسب را از دیسک می‌خوانیم، و بالای پشته را بازمی‌گردانیم. نیازی به دوباره نوشتن صفحه نداریم، چرا که تغییری در صفحه انجام نشده است.

چون اعمال دیسک نسبتاً پر هزینه هستند، برای هر پیاده‌سازی دو هزینه را محاسبه می‌کنیم: تعداد کل دسترسی‌ها به دیسک و کل زمان CPU مورد نیاز. هر دسترسی به دیسک برای یک صفحه با m کلمه، هزینه‌ای برابر با یک دسترسی به دیسک و $\theta(m)$ زمان CPU دارد.

I. به صورت حدی، بدترین حالت تعداد دسترسی‌های دیسک برای n عملیات پشته با استفاده از این پیاده‌سازی ساده چیست؟ زمان CPU مورد نیاز برای n عملیات پشته چقدر است؟ (در این بخش و بخش‌های بعد، جواب‌های خود را بر حسب m و n توصیف کنید.)

اکنون یک پیاده‌سازی پشته را در نظر بگیرید که در آن یک صفحه از پشته را در حافظه نگه می‌داریم. (همچنین از مقدار کمی از حافظه استفاده می‌کنیم تا مشخص کنیم که کدام صفحه از دیسک در حافظه قرار دارد.) فقط زمانی می‌توانیم یک عملیات پشته انجام دهیم که صفحه‌ی مربوط در حافظه باشد. در صورت لزوم، صفحه‌ای که در حال حاضر در حافظه است می‌تواند بر روی دیسک نوشته شود و صفحه‌ی جدید از دیسک به حافظه‌ی اصلی خوانده شود. اگر صفحه‌ی مربوطه در حافظه باشد، دیگر نیازی به دسترسی به دیسک نیست.

II. بدترین حالت تعداد دسترسی‌های دیسک برای n عملیات PUSH چیست؟ زمان CPU چقدر است؟

III. بدترین حالت تعداد دسترسی‌های دیسک برای n عملیات پشته چیست؟ زمان CPU چقدر است؟

اکنون فرض کنید پشته را با نگه داشتن دو صفحه در حافظه نگه می‌داریم (به علاوه‌ی تعداد کمی کلمه برای ساماندهی صفحه‌ها).

IV. توضیح دهید چگونه می‌توانیم صفحه‌های پشته را اداره کنیم که تعداد سرشکن دسترسی‌های دیسک برای هر عملیات پشته $O(1/m)$ و زمان سرشکن CPU برای هر عملیات پشته $O(1)$ باشد.

۱۸-۲ *اتصال و جداسازی درختان ۲-۳-۴*

عملیات *اتصال* (join) دو مجموعه‌ی پویای S' و S'' و یک عنصر x را می‌گیرد به طوری که برای هر $x' \in S'$ و $x'' \in S''$ داریم $x'.key < x.key < x''.key$، و یک مجموعه‌ی $S = S' \cup \{x\} \cup S''$ را بازمی‌گرداند. عملیات *جداسازی* (split) چیزی مانند «برعکس»

مسائل

اتصال است: با دریافت یک مجموعه‌ی پویای S و یک عنصر $x \in S$، یک مجموعه‌ی S' حاوی تمام عناصر $S - \{x\}$ می‌سازد که تمام کلید آن‌ها کم‌تر از $x.key$ است، و همچنین یک مجموعه‌ی S'' حاوی تمام عناصری از $S - \{x\}$ که کلید آن‌ها بزرگ‌تر از $x.key$ است. در این مسئله خواهیم دید که چطور می‌توان این اعمال را بر روی درختان ۲-۳-۴ پیاده‌سازی کرد. برای سادگی فرض می‌کنیم عناصر فقط حاوی یک کلید بوده و تمام کلیدها یکتا هستند.

I. نشان دهید که چگونه می‌توان برای هر گره‌ی x از یک درخت ۲-۳-۴، ارتفاع زیردخت x را به صورت فیلد $x.height$ نگه داشت. اطمینان حاصل کنید که پیاده‌سازی شما بر روی زمان اجرای حدی جستجو، درج و حذف تأثیری نمی‌گذارد.

II. نشان دهید که چگونه می‌توان عملیات اتصال را پیاده‌سازی کرد. عملیات اتصال باید با دریافت دو درخت ۲-۳-۴ با نام‌های T' و T'' و یک کلید k در زمان $O(1+|h' - h''|)$ اجرا شود، که در آن h' و h'' به ترتیب ارتفاع درخت‌های T' و T'' هستند.

III. مسیر p از ریشه‌ی درخت ۲-۳-۴ T به یک کلید k، مجموعه‌ی S' از کلیدهایی در T که کم‌تر از k هستند، و مجموعه‌ی S'' از کلیدهایی در S که از k بزرگ‌تر هستند را در نظر بگیرید. نشان دهید که p مجموعه‌ی S' را به مجموعه‌ای از درخت‌های $\{T'_0, T'_1, ..., T'_m\}$ و مجموعه‌ای از کلیدهای $\{k'_1, k'_2, ..., k'_m\}$ تقسیم می‌کند، که در آن برای $i = 1, 2, ..., m$، هر کلید $y \in T'_{i-1}$، و $z \in T'_i$، داریم $y < k'_i < z$. ارتباط میان ارتفاع T'_{i-1} و T'_i چیست؟ توضیح دهید که p چگونه S'' را به مجموعه‌هایی از درخت‌ها و کلیدها تقسیم می‌کند.

IV. نشان دهید که چگونه می‌توانیم عملیات جداسازی را بر روی T پیاده‌سازی کنیم. از عملیات اتصال برای سرهم کردن کلیدهای S' در یک درخت ۲-۳-۴ T' و سرهم کردن کلیدهای S'' در یک درخت ۲-۳-۴ T'' استفاده کنید. زمان اجرای عملیات جداسازی باید $O(\lg n)$ باشد، که در آن n تعداد کلیدها در T است. (*راهنمایی:* هزینه‌های اتصال باید به صورت سری تلسکوپی باشد.)

فصل

۱۹

هرم‌های فیبوناچی

۱۹-۰ مقدمه

ساختمان داده‌ی هرم فیبوناچی دو هدف را دنبال می‌کند. اول، از مجموعه‌ای از اعمال پشتیبانی می‌کند که عنصر اصلی تشکیل دهنده‌ی «هرم‌های قابل ادغام» است. دوم، بسیاری از اعمال هرم‌های فیبوناچی در زمان سرشکن ثابت اجرا می‌شوند، که این ساختمان داده را برای کاربردهایی که مکرراً از این اعمال استفاده می‌کنند، مناسب می‌سازد.

هرم‌های قابل ادغام

یک *هرم قابل ادغام* (mergeable heap) ساختمان داده‌ای است که از پنج عمل زیر پشتیبانی می‌کند، که در آن هر عنصر یک *کلید* دارد:

MAKE-HEAP() یک هرم جدید ساخته و آن را (که هیچ عنصری ندارد) بازمی‌گرداند.

INSERT(H, x) عنصر x را که کلید آن قبلاً مقداردهی شده است، در هرم H درج می‌کند.

MINIMUM(H) یک اشاره‌گر به عنصری در H بازمی‌گرداند که کلید آن کمینه است.

EXTRACT-MIN(H) عنصری در H را که کم‌ترین مقدار را دارد، حذف کرده و اشاره‌گری به آن بازمی‌گرداند.

UNION(H_1, H_2) یک هرم جدید ساخته و بازمی‌گرداند که حاوی تمام عناصر H_1 و H_2 است. هرم‌های H_1 و H_2 توسط این عملیات «نابود» می‌شوند.

علاوه بر اعمال هرم‌های قابل ادغام بالا، هرم‌های فیبوناچی از دو عملیات زیر هم پشتیبانی می‌کنند:

DECREASE-KEY(H, x, k) مقدار کلید جدید k را به عنصر x در هرم H نسبت می‌دهد، که

رویه	هرم دودویی (بدترین حالت)	هرم فیبوناچی (سرشکن)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

شکل ۱۹-۱ زمان‌های اجرا برای اعمال دو پیاده‌سازی مختلف از هرم‌های قابل ادغام. تعداد عناصر در هرم(ها) در زمان یک عمل با n نشان داده شده است.

فرض می‌کنیم از مقدار فعلی آن بزرگ‌تر نیست.[1]

DELETE (H, x) عنصر x را از هرم H حذف می‌کند.

همان طور که جدول شکل ۱۹-۱ نشان می‌دهد، اگر نیازی به عملیات UNION نداشته باشیم هرم‌های دودویی معمولی، که در مرتب‌سازی هرمی از آن‌ها استفاده شد (فصل ۶) نسبتاً خوب کار می‌کنند. اعمال غیر از UNION در هرم‌های دودویی در بدترین حالت در زمان $O(\lg n)$ اجرا می‌شوند. ولی اگر نیاز به عملیات UNION باشد هرم‌های دودویی به شدت ضعیف هستند. با اتصال دو آرایه که هرم‌های دودویی را نگه می‌دارند و سپس فراخوانی BUILD-MIN-HEAP (بخش ۶-۳ را ببینید)، عملیات UNION در بدترین حالت به $\theta(n)$ زمان نیاز دارد.

از سوی دیگر، هرم‌های فیبوناچی برای اعمال INSERT، UNION، و DECREASE-KEY زمان‌های حدی بهتری دارند، و زمان حدی آن‌ها برای بقیه‌ی اعمال مانند هرم‌های دودویی است. ولی توجه کنید که زمان‌های اجرای داده شده در شکل ۱۹-۱ برای هرم‌های فیبوناچی، زمان‌های سرشکن هستند و نه بدترین حالت برای هر عمل. عملیات UNION برای هرم‌های فیبوناچی فقط به زمان سرشکن ثابت نیاز دارد، که به شدت از بدترین حالت زمان خطی برای هرم‌های دودویی بهتر است (البته با این فرض که کران سرشکن برای ما کافی باشد).

هرم‌های فیبوناچی در تئوری و عمل

از دید تئوری، هرم‌های فیبوناچی زمانی به صرفه هستند که تعداد اعمال EXTRACT-MIN و DELETE نسبت به تعداد بقیه‌ی اعمال اجرا شده کم باشد. این وضعیت در بسیاری از کاربردها پیش می‌آید. مثلاً

[1] همان طور که در معرفی بخش پنج گفته شد، هرم‌های قابل ادغام پیش‌فرض ما، هرم‌های قابل ادغام کمینه هستند، و بنابراین اعمال MINIMUM، EXTRACT-MIN، و DECREASE-KEY کاربرد دارند. در عوض می‌توانستیم *هرم‌های قابل ادغام بیشینه* را تعریف کنیم، با اعمال MAXIMUM، EXTRACT-MAX، و INCREASE-KEY.

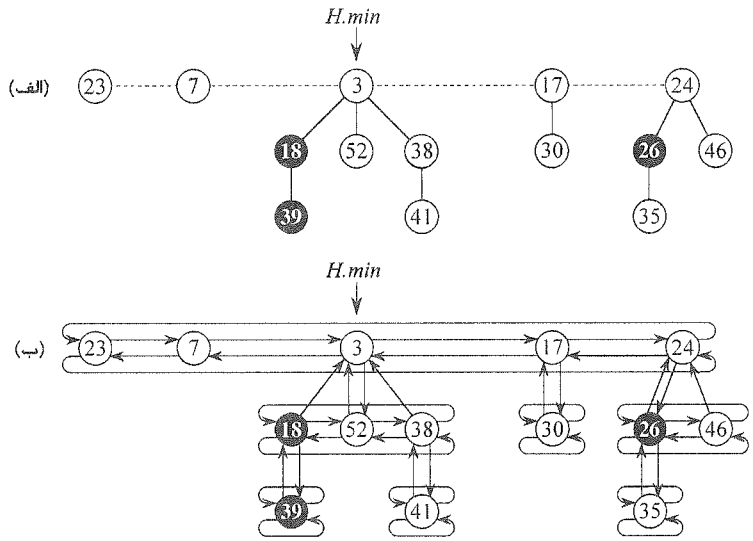
بعضی الگوریتم‌های گراف ممکن است DECREASE-KEY را برای هر یال یک بار فراخوانی کنند. برای گراف‌های شلوغ (dense) که یال‌های زیادی دارند، زمان سرشکن $O(1)$ برای هر بار فراخوانی DECREASE-KEY، بهبود بسیار زیادی نسبت به بدترین حالت زمان اجرای $\theta(\lg n)$ برای هرم‌های دودویی یا دوجمله‌ای ایجاد می‌کند. الگوریتم‌های سریع برای مسائلی مانند محاسبه‌ی درخت‌های پوشای کمینه (فصل ۲۳) و یافتن کوتاه‌ترین مسیرها از یک مبدأ (فصل ۲۴) استفاده‌ی قابل توجهی از هرم‌های فیبوناچی می‌کنند.

با این حال از دید کاربردی، ضرایب ثابت و پیچیدگی برنامه‌نویسی هرم‌های فیبوناچی آن‌ها را نسبت به هرم‌های دودویی (یا k-تایی) معمولی بسیار کم طرفدارتر می‌کنند، مگر برای کاربردهایی که در آن‌ها نیاز به مدیریت مقادیر زیادی از داده‌ها وجود دارد. بنابراین هرم‌های فیبوناچی بیشتر در تئوری کاربرد دارند. اگر یک ساختمان داده‌ی ساده‌تر با کران‌های سرشکن یکسان با هرم‌های فیبوناچی طراحی می‌شد، آن گاه می‌شد از آن در عمل هم استفاده کرد.

هر دو هرم دودویی و فیبوناچی درنحوه‌ی پشتیبانی از عمالیت SEARCH ناکارامد هستند؛ یافتن یک عنصر با یک کلید داده شده می‌تواند زمان زیادی به طول بیانجامد. به همین دلیل اعمالی مانند DECREASE-KEY و DELETE که روی یک عنصر خاص عمل می‌کنند، باید اشاره‌گری به آن عنصر را به عنوان بخشی از ورودی دریافت کنند. مانند بحث صف‌های اولویت در بخش ۶-۵، وقتی از یک هرم قابل ادغام در یک کاربرد استفاده می‌کنیم معمولاً یک دستگیره (اشاره‌گر) به شیئ مورد نظر در کاربرد در هر عنصر هرم قابل ادغام ذخیره می‌کنیم، همچنین یک دستگیره به عنصر هرم قابل ادغام در شیئ متناظر در کاربرد. ساختار دقیق این دستگیره‌ها به کاربرد و نحوه‌ی پیاده‌سازی بستگی دارد.

مانند بسیاری از ساختمان‌های داده‌ی دیگری که دیدیم، هرم‌های فیبوناچی بر پایه‌ی درخت‌های ریشه‌دار بنا شده‌اند. هر عنصر را به کمک یک گره در یک درخت نشان می‌دهیم، و هر گره یک خصیصه‌ی key دارد. در ادامه‌ی این فصل از اصطلاح «گره» به جای «عنصر» استفاده خواهیم کرد. همچنین از پرداختن به مسائل مربوط به اختصاص حافظه به گره‌ها قبل از درج آن‌ها و یا آزاد کردن حافظه بعد از حذف آن‌ها صرف نظر می‌کنیم، با این فرض که کدی که رویه‌های هرم را فراخوانی می‌کند، با این جزئیات سروکار دارد.

در بخش ۱۹-۱ هرم‌های فیبوناچی تعریف می‌شوند، در مورد نمایش آن‌ها بحث شده، و تابع پتانسیل استفاده شده برای تحلیل سرشکن آن‌ها ارائه می‌شود. بخش ۱۹-۲ نشان می‌دهد که چگونه می‌توان اعمال هرم‌های قابل ادغام را بر روی هرم‌های فیبوناچی با کران‌های سرشکن نشان داده شده در شکل ۱۹-۱ پیاده‌سازی کرد. دو عملیات باقی مانده، DECREASE-KEY و DELETE در بخش ۱۹-۳ ارائه خواهند شد. نهایتاً بخش ۱۹-۴ یک بخش کلیدیی تحلیل را پایان داده و در مورد نام عجیب این ساختمان داده توضیح می‌دهد.



شکل ۱۹-۲ (الف) یک هرم فیبوناچی متشکل از پنج درخت هرم کمینه-مرتب و ۱۴ گره. خط نقطه‌چین نشان دهنده‌ی لیست ریشه است. گره‌ی کمینه‌ی هرم گره‌ی حاوی کلید ۳ است. سه تا از گره‌ها با رنگ سیاه علامت گذاری شده‌اند. پتانسیل این هرم فیبوناچی خاص $۱۱ = ۳ \times ۲ + ۵$ است. (ب) یک نمایش کامل‌تر که اشاره‌گرهای p (پیکان‌های بالایی)، *child* (پیکان‌های پایینی)، *left* و *right* (پیکان‌های چپ و راست) را نشان می‌دهد. در شکل‌های باقی مانده‌ی این فصل از این جزئیات صرف نظر شده است، چرا که تمام اطلاعات نشان داده شده در این جا را می‌توان از شکل بخش (الف) تشخیص داد.

۱۹-۱ ساختار هرم‌های فیبوناچی

یک *هرم فیبوناچی* مجموعه‌ای از درخت‌های ریشه‌دار با خصوصیت *هرم کمینه-مرتب* است. یعنی هر درخت از خصوصیت هرم‌های کمینه پیروی می‌کند: کلید یک گره بزرگ‌تر یا مساوی کلید پدر آن گره است. شکل ۱۹-۲(الف) نمونه‌ای از یک هرم فیبوناچی را نشان می‌دهد.

همان طور که شکل ۱۹-۲ (ب) نشان می‌دهد، هر گره‌ی x یک اشاره‌گر $x.p$ به پدر و یک اشاره‌گر $x.child$ به یکی از فرزندان خود دارد. فرزندان x به صورت یک لیست پیوندی دوطرفه و دایره‌ای به یکدیگر متصل شده‌اند، که ما به آن *لیست فرزندان* x می‌گوییم. هر فرزند y در یک لیست فرزندان دارای اشاره‌گرهای $y.left$ و $y.right$ است که به ترتیب به برادران چپ و راست y اشاره می‌کنند. اگر گره‌ی y تنها فرزند باشد، آن گاه $y.left = y.right = y$. ترتیب ظاهر شدن فرزندان در یک لیست فرزندان دلخواه است.

لیست‌های پیوندی دوطرفه‌ی دایره‌ای (بخش ۱۰-۲ را ببینید) دارای دو مزیت برای استفاده در هرم‌های فیبوناچی هستند. اول، می‌توان در زمان $O(۱)$ یک گره را به هرجایی از یک لیست پیوندی

دوطرفه‌ی دایره‌ای اضافه و یا از هر جایی از آن حذف کرد. دوم، با داشتن دو تا از این لیست‌ها می‌توان آن‌ها را به صورت یک لیست پیوندی دوطرفه‌ی دایره‌ای در زمان $O(1)$ به یکدیگر متصل کرد (یا «پیوند» داد). در توضیح اعمال هرم‌های فیبوناچی به صورت غیر رسمی به این اعمال هم اشاره خواهیم کرد، که خواننده می‌تواند جزئیات پیاده‌سازی خود را در آن وارد کند.

دو فیلد دیگر در هر گره می‌تواند مورد استفاده قرار گیرد. تعداد فرزندان درون لیست فرزندان گره‌ی x در فیلد $x.degree$ ذخیره می‌شود. فیلد $x.mark$ با مقدار بولین تعیین می‌کند که آیا از آخرین باری که x فرزند یک گره‌ی دیگر شده است، x فرزندی از دست داده است یا خیر. این فیلد در گره‌های تازه ساخته شده بدون مقدار است، و همچنین هر وقت گره‌ای فرزند یک گره‌ی دیگر می‌شود، این فیلد در آن گره مقدار خود را از دست می‌دهد. تا زمانی که به عملیات DECREASE-KEY در بخش ۱۹-۳ برسیم، برای سادگی تمام فیلدهای $mark$ را با FALSE مقداردهی می‌کنیم.

دسترسی به یک هرم فیبوناچی H توسط یک اشاره‌گر $H.min$ به ریشه‌ی یک درخت حاوی یک کلید کمینه صورت می‌گیرد؛ این گره، *گره‌ی کمینه‌ی* هرم فیبوناچی نام دارد. اگر کلید بیش از یک ریشه، مقدار کمینه داشته باشد، هر کدام از آن‌ها می‌توانند گره‌ی کمینه باشند. اگر یک هرم فیبوناچی H تهی باشد، آن گاه $H.min = \text{NIL}$.

ریشه‌ی تمام درخت‌ها در یک هرم فیبوناچی به وسیله‌ی اشاره‌گرهای $left$ و $right$ آن‌ها و به صورت یک لیست پیوندی دوطرفه‌ی دایره‌ای به نام *لیست ریشه* ی هرم فیبوناچی به یکدیگر متصل شده‌اند. بنابراین اشاره‌گر $H.min$ به گره‌ای در لیست ریشه اشاره می‌کند که کلید آن کمینه است. ترتیب درخت‌ها در یک لیست ریشه دلخواه است.

همچنین به یک خصوصیت دیگر برای هرم فیبوناچی H نیاز داریم: تعداد گره‌هایی که در حال حاضر در H هستند در $H.n$ ذخیره می‌شود.

تابع پتانسیل

همان طور که گفته شد، از متد پتانسیل بخش ۱۷-۳ برای تحلیل کارایی اعمال هرم‌های فیبوناچی استفاده خواهیم کرد. برای یک هرم فیبوناچی داده شده‌ی H، تعداد درخت‌های درون لیست ریشه‌ی H را با $t(H)$ و تعداد گره‌های علامت گذاری شده در H را با $m(H)$ نشان می‌دهیم. در این صورت پتانسیل هرم فیبوناچی H به صورت

$$\Phi(H) = t(H) + 2m(H) \qquad (۱-۱۹)$$

تعریف می‌شود. (در بخش ۱۹-۳ مقداری شهود بر روی این تابع پتانسیل به دست خواهیم آورد.) به عنوان مثال، پتانسیل هرم فیبوناچی نشان داده شده در شکل ۱-۱۹ برابر است با $11 = 3 \times 2 + 5$. پتانسیل مجموعه‌ای از هرم‌های فیبوناچی برابر است با مجموع پتانسیل هرم‌های فیبوناچی آن مجموعه. فرض خواهیم کرد که یک واحد پتانسیل می‌تواند هزینه‌ی مقدار ثابتی کار را پرداخت کند، که این ثابت به اندازه‌ی کافی بزرگ است که هزینه‌ی هر مقدار ثابت کار که ممکن است به آن پرداخت کنیم را می‌پوشاند.

فرض می‌کنیم یک کاربرد از هرم‌های فیبوناچی با مجموعه‌ای تهی از هرم‌ها شروع می‌کند. بنابراین، پتانسیل اولیه ۰ است، و طبق تساوی (۱۹-۱) بعد از آن پتانسیل همیشه نامنفی خواهد بود. بنابراین از تساوی (۱۷-۳)، یک کران بالا برای کل هزینه‌ی سرشکن، کران بالایی است برای کل هزینه‌ی واقعی دنباله‌ای از عملیات.

درجه‌ی بیشینه

در تحلیل سرشکنی که در باقی این فصل انجام می‌دهیم فرض می‌شود که یک کران بالای شناخته شده‌ی $D(n)$ بر روی درجه‌ی بیشینه‌ی هر گره در یک هرم فیبوناچی با n گره وجود دارد. در این جا این را اثبات نمی‌کنیم، ولی وقتی فقط از اعمال هرم‌های قابل ادغام پشتیبانی می‌شود، $D(n) \le \lfloor \lg n \rfloor$. (مسئله‌ی ۱۹-۲(ت) از شما می‌خواهد این را اثبات کنید.) در بخش‌های ۱۹-۳ و ۱۹-۴ نشان خواهیم داد که وقتی که از DECREASE-KEY و DELETE هم پشتیبانی می‌کنیم، $D(n) = O(\lg n)$.

۱۹-۲ اعمال هرم‌های قابل ادغام

ایده‌ی کلیدی در اعمال هرم‌های قابل ادغام بر روی هرم‌های فیبوناچی این است که کارها را تا حد ممکن به تعویق بیندازیم. برای پیاده‌سازی اعمال مختلف یک سبک-سنگین برای کارایی وجود دارد. مثلاً درج یک گره را با اضافه کردن آن به لیست ریشه انجام می‌دهیم، که فقط به زمان ثابت نیاز دارد. اگر با شروع از یک هرم فیبوناچی خالی، k گره را در آن درج کنیم، آن گاه هرم فیبوناچی عبارت خواهد بود از یک لیست ریشه حاوی k گره. سبک-سنگین به این گونه است که اگر بعد از آن یک عملیات EXTRACT-MIN بر روی هرم H انجام دهیم، پس از حذف گره‌ای که $H.min$ به آن اشاره می‌کند، باید تمام $k-1$ گره‌ی لیست ریشه را بررسی کنیم تا گره‌ی کمینه‌ی جدید را بیابیم. وقتی بعد از عملیات EXTRACT-MIN مجبور باشیم کل لیست را بررسی کنیم، می‌توانیم همزمان گره‌ها را به صورت هرم کمینه-مرتب درآوریم تا اندازه‌ی لیست ریشه کوچک شود. خواهیم دید که مستقل از اندازه‌ی لیست ریشه قبل از عملیات EXTRACT-MIN، بعد از آن هر گره در لیست ریشه درجه‌ای دارد که در کل لیست یکتا است، که یک لیست با اندازه‌ی حداکثر $D(n)+1$ به ما می‌دهد.

ساختن یک هرم فیبوناچی جدید

برای ساختن یک هرم فیبوناچی تهی، رویه‌ی MAKE-FIB-HEAP شیئ هرم فیبوناچی H را تخصیص داده و بازمی‌گرداند، که در آن $H.n = 0$ و $H.min = \text{NIL}$؛ هیچ درختی در H وجود ندارد. چون $t(H) = 0$ و $m(H) = 0$، پتانسیل هرم فیبوناچی $\Phi(H) = 0$ خواهد بود. بنابراین هزینه‌ی سرشکن MAKE-FIB-HEAP برابر با هزینه‌ی واقعی $O(1)$ است.

درج یک گره

رویه‌ی زیر گره‌ی x را در هرم فیبوناچی H درج می‌کند، با فرض این که حافظه‌ی گره قبلاً تخصیص

داده شده و فیلد $x.key$ قبلاً مقداردهی شده است.

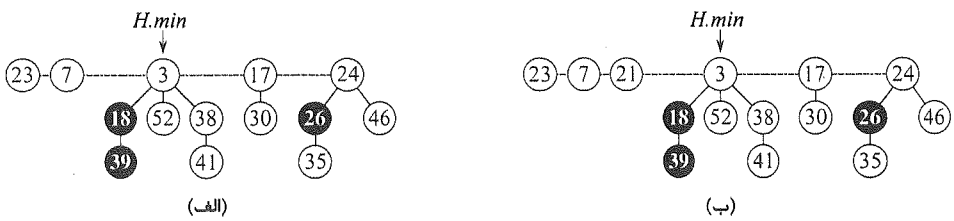
```
FIB-HEAP-INSERT(H, x)
1  x.degree = 0
2  x.p = NIL
3  x.child = NIL
4  x.mark = FALSE
5  if H.min == NIL
6      create a root list for H containing just x
7      H.min = x
8  else insert x into H's root list
9      if x. key < H.min.key
10         H.min = x
11 H.n = H.n + 1
```

خطوط ۱–۴ ساختار فیلدهای x را مقداردهی اولیه می‌کنند. خط ۵ تست می‌کند که آیا هرم فیبوناچی H تهی است یا نه. اگر تهی بود خطوط ۶–۷، x را به عنوان تنها عنصر لیست ریشه قرار می‌دهند، و $H.min$ را طوری مقداردهی می‌کنند که به x اشاره کند. در غیر این صورت، خطوط ۸–۱۰، x را در لیست ریشه‌ی H درج کرده و در صورت نیاز، $H.min$ را اصلاح می‌کنند. نهایتاً در خط ۱۱، $H.n$ افزایش می‌یابد تا تأثیر اضافه کردن گره‌ی جدید را بپذیرد. شکل ۱۹–۳ یک گره با کلید ۲۱ را نشان می‌دهد که در هرم فیبوناچی شکل ۱۹–۲ درج شده است.

برای تعیین هزینه‌ی سرشکن FIB-HEAP-INSERT، فرض کنید H هرم فیبوناچی ورودی و H' هرم فیبوناچی حاصل باشد. آن گاه $t(H') = t(H) + 1$ و $m(H') = m(H)$، و میزان افزایش در پتانسیل برابر است با

$$((t(H)+1)+2m(H))-(t(H)+2m(H))=1$$

از آن جایی که هزینه‌ی واقعی $O(1)$ است، هزینه‌ی سرشکن $O(1)+1=O(1)$ خواهد بود.



شکل ۱۹–۳ درج یک گره در یک هرم فیبوناچی. (الف) یک هرم فیبوناچی H. (ب) هرم فیبوناچی H بعد از درج گره‌ی با کلید ۲۱. این گره خود تبدیل به یک درخت هرم کمینه-مرتب شده و سپس به لیست ریشه اضافه شده، تبدیل به برادر سمت راست ریشه می‌شود.

یافتن گره‌ی کمینه

گره‌ی کمینه‌ی یک هرم فیبوناچی H توسط اشاره‌گر $H.min$ تعیین می‌شود، بنابراین می‌توانیم گره‌ی کمینه را در زمان واقعی $O(1)$ بیابیم. چون پتانسیل H تغییری نمی‌کند، هزینه‌ی سرشکن این عملیات برابر با هزینه‌ی واقعی $O(1)$ است.

اجتماع دو هرم فیبوناچی

رویه‌ی زیر اجتماع دو هرم فیبوناچی H_1 و H_2 را محاسبه کرده و H_1 و H_2 را نابود می‌کند. این رویه به سادگی لیست ریشه‌ی H_1 و H_2 را به هم متصل کرده، سپس گره‌ی کمینه‌ی جدید را تعیین می‌کند.

```
FIB-HEAP-UNION(H1, H2)
1  H = MAKE-FIB-HEAP()
2  H.min = H1.min
3  concatenate the root list of H2 with the root list of H
4  if (H1.min == NIL) or (H2.min ≠ NIL and H2.min < H1.min)
5      H.min = H2.min
6  H.n = H1.n + H2.n
7  return H
```

خطوط ۱-۳ لیست ریشه‌ی H_1 و H_2 را به هم متصل کرده یک لیست جدید H می‌سازند. خطوط ۲، ۴ و ۵ گره‌ی کمینه‌ی H را مقداردهی می‌کنند، و خط ۶، $H.n$ را برابر با تعداد کل گره‌ها قرار می‌دهد. اشیای هرم‌های فیبوناچی H_1 و H_2 در خط ۷ آزاد می‌شوند، و در خط ۸ هرم فیبوناچی حاصل H بازگردانده می‌شود. مانند رویه‌ی FIB-HEAP-INSERT، هیچ مستحکم‌سازی بر روی درخت‌ها انجام نمی‌شود.

میزان تغییر در پتانسیل به صورت

$$\begin{aligned}\Phi(H) - (\Phi(H_1) + \Phi(H_2)) &= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))) \\ &= 0\end{aligned}$$

است، چرا که $t(H) = t(H_1) + t(H_2)$ و $m(H) = m(H_1) + m(H_2)$. بنابراین هزینه‌ی سرشکن FIB-HEAP-UNION برابر با هزینه‌ی واقعی $O(1)$ است.

استخراج گره‌ی کمینه

فرآیند استخراج گره‌ی کمینه پیچیده‌ترین عملیات ارائه شده در این بخش است. همچنین در این جا است که کارهای به تعویق افتاده‌ی مستحکم‌سازی درخت‌ها در لیست ریشه انجام می‌شود. شبه‌کد زیر گره‌ی کمینه را استخراج می‌کند. برای سادگی، در کد فرض می‌شود که وقتی یک گره از یک لیست پیوندی حذف می‌شود، اشاره‌گرهای باقی مانده در لیست به هنگام سازی می‌شوند، ولی اشاره‌گرهای

درون گره‌ی استخراج شده بدون تغییر باقی می‌مانند. این رویه همچنین از رویه‌ی کمکی CONSOLIDATE استفاده می‌کند، که به زودی معرفی خواهد شد.

```
FIB-HEAP-EXTRACT-MIN(H)
1  z = H.min
2  if z ≠ NIL
3      for each child x of z
4          add x to the root list of H
5          x.p = NIL
6      remove z from the root list of H
7      if z == z.right
8          H.min = NIL
9      else H.min = z.right
10         CONSOLIDATE(H)
11     H.n = H.n – 1
12 return z
```

همان طور که در شکل ۱۹-۴ نشان داده شده است، FIB-HEAP-EXTRACT-MIN ابتدا از هر یک از فرزندان گره‌ی کمینه یک ریشه می‌سازد و سپس گره‌ی کمینه را از لیست ریشه حذف می‌کند. سپس با اتصال ریشه‌های با درجه‌ی برابر لیست ریشه را مستحکم‌سازی می‌کند، و این کار را تا زمانی انجام می‌دهد که از هر درجه حداکثر یک ریشه باقی بماند.

در خط ۱ با ذخیره‌ی یک اشاره‌گر z به گره‌ی کمینه شروع می‌کنیم؛ این اشاره‌گر در پایان بازگردانده می‌شود. اگر z برابر با NIL باشد، آن گاه هرم فیبوناچی H خالی است و کار ما در این جا تمام می‌شود. در غیر این صورت، مانند رویه‌ی BINOMIAL-HEAP-EXTRACT-MIN گره‌ی z را از H حذف می‌کنیم، بدین صورت که تمام فرزندان z را در خطوط ۳-۵ تبدیل به ریشه‌هایی در H می‌کنیم (آن‌ها را در لیست ریشه قرار می‌دهیم) و در خط ۶، z را از لیست ریشه حذف می‌کنیم. اگر بعد از خط ۶، z برادر سمت راست خود باشد، آن گاه z تنها گره در لیست ریشه بوده است و هیچ فرزندی ندارد، و بنابراین تنها کار باقی مانده این است که قبل از بازگرداندن z در خط ۸، هرم فیبوناچی را تهی کنیم. در غیر این صورت اشاره‌گر $H.min$ را در لیست ریشه طوری مقداردهی می‌کنیم که به یک گره غیر از z (در این حالت، برادر سمت راست z) اشاره کند، که لزوماً پس از پایان FIB-HEAP-EXTRACT-MIN، گره‌ی کمینه‌ی جدید نخواهد بود. شکل ۱۹-۴ (ب) هرم فیبوناچی شکل ۱۹-۴(الف) را بعد از اجرای خط ۹ نشان می‌دهد.

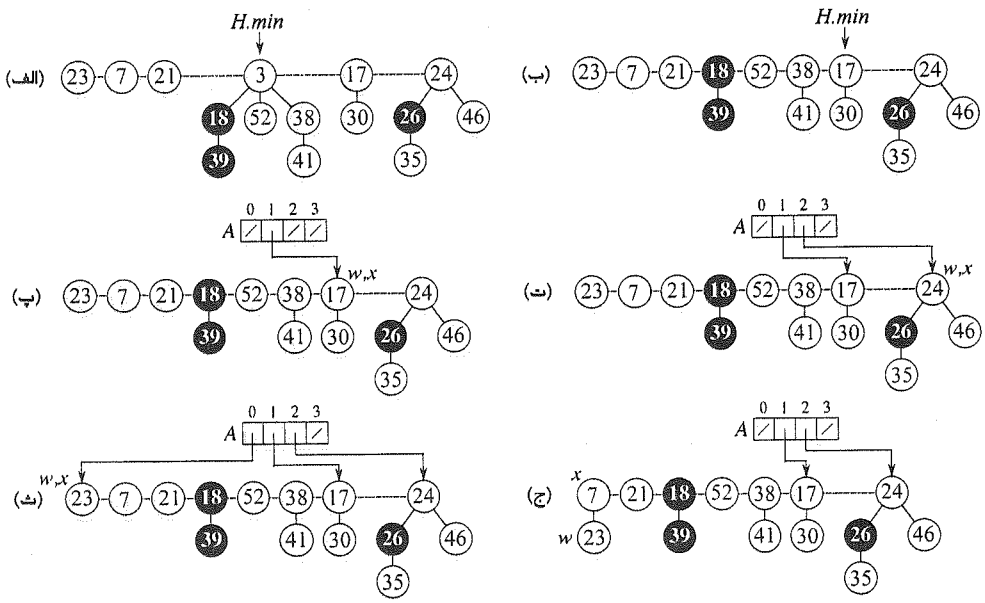
مرحله‌ی بعد، که در آن تعداد درخت‌ها در هرم فیبوناچی را کاهش می‌دهیم، *مستحکم‌سازی* لیست ریشه‌ی H است؛ این کار با فراخوانی CONSOLIDATE(H) انجام می‌شود. مستحکم‌سازی لیست ریشه شامل اجرای مکرر مراحل زیر است، تا زمانی که مقدار *degree* تمام ریشه‌ها در لیست یکتا باشد.

۱. دو ریشه‌ی x و y در لیست ریشه با مقدار درجه‌ی یکسان می‌یابیم. بدون از دست دادن کلیت، فرض می‌کنیم $x.key \le y.key$.

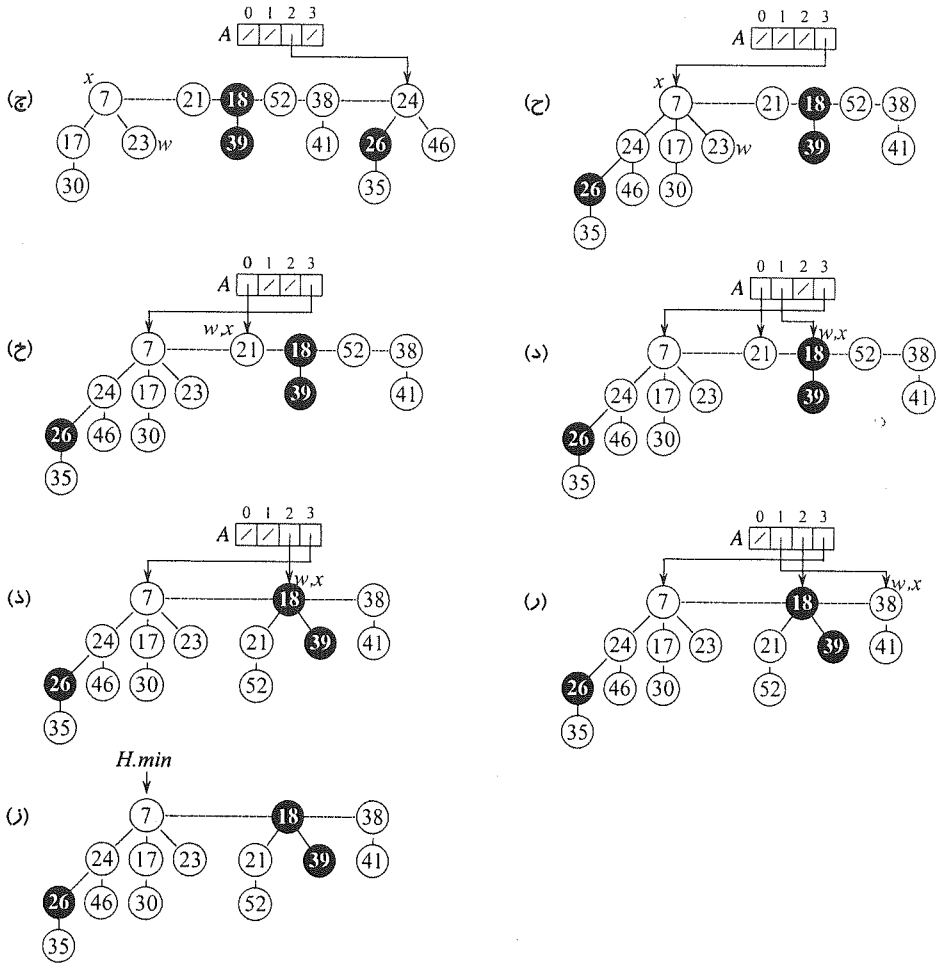
۲. y را به x *پیوند* می‌دهیم: y را از لیست ریشه حذف، و آن را تبدیل به یک فرزند

x می‌کنیم. این عملیات به وسیله‌ی رویه‌ی FIB-HEAP-LINK انجام می‌شود. فیلد $x.degree$ افزایش می‌یابد، و علامت y، در صورت وجود پاک می‌شود.

رویه‌ی CONSOLIDATE از یک آرایه‌ی کمکی $A[0 .. D(H.n)]$ استفاده می‌کند تا اطلاعات ریشه‌ها را بر حسب درجه نگه دارد. اگر $A[i] = y$، آن گاه y یک ریشه با $y.degree = i$ است. مسلماً برای تخصیص دهی آرایه باید بدانیم که چگونه کران بالای $D(H.n)$ را بر روی درجه‌ی بیشینه محاسبه کنیم، ولی روش انجام این کار را در بخش ۱۹-۴ خواهیم دید.



شکل ۱۹-۴ عملیات FIB-HEAP-EXTRACT-MIN. (الف) یک هرم فیبوناچی H. (ب) وضعیت بعد از این که گره‌ی کمینه‌ی z از لیست ریشه حذف شد و فرزندان آن به لیست ریشه اضافه شدند. (پ)-(ث) آرایه‌ی A و درخت‌ها بعد از هر یک از سه تکرار اول حلقه‌ی for در خطوط ۴-۱۴ رویه‌ی CONSOLIDATE. لیست ریشه با شروع از گره‌ای که $H.min$ به آن اشاره می‌کند، پردازش می‌شود، و این پردازش با اشاره‌گرهای *right* بعدی ادامه می‌یابد. در هر بخش مقادیر w و x در انتهای هر تکرار نشان داده شده است. (ج)-(ح) تکرار بعدی حلقه‌ی for، که مقادیر w و x در پایان هر تکرار حلقه‌ی while خطوط ۷-۱۳ نشان داده شده‌اند. قسمت (ج) وضعیت را بعد از اولین عبور از حلقه‌ی while نشان می‌دهد. گره‌ی با کلید ۲۳ به گره‌ی با کلید ۷ متصل شده است، که اکنون x به آن اشاره می‌کند. در بخش (چ)، گره‌ی با کلید ۱۷ به گره‌ی با کلید ۷ متصل شده است، که همچنان x به آن اشاره می‌کند. در بخش (ح)، گره‌ی با کلید ۲۴ به گره‌ی با کلید ۷ متصل شده است. از آن جایی که قبلاً $A[3]$ به هیچ کلیدی اشاره نمی‌کرد، بعد از تکرار حلقه‌ی for، خانه‌ی آرایه‌ی $A[3]$ طوری مقداردهی شده است که به ریشه‌ی درخت حاصل اشاره کند. (خ)-(ر) وضعیت بعد از هر تکرار از چهار تکرار بعدی حلقه‌ی for. (ز) هرم فیبوناچی H بعد از بازسازی لیست ریشه از آرایه‌ی A و تعیین اشاره‌گر جدید $H.min$.



شکل ۱۹–۴ (ادامه)

```
CONSOLIDATE(H)
1   let A[0..D(H.n)] be a new array
2   for i = 0 to D(n[H])
3       A[i] = NIL
4   for each node w in the root list of H
5       x = w
6       d = x.degree
7       while A[d] ≠ NIL
8           y = A[d]    // Another node with the same degree as x.
9           if x.key > y.key
10              exchange x wiht y
11          FIB-HEAP-LINK(H, y, x)
12          A[d] = NIL
```

```
13          d = d + 1
14      A[d] = x
15  H. min = NIL
16  for i = 0 to D(n[H])
17      if A[i] ≠ NIL
18          if H.min == NIL
19              create a root list for H containing just A[i]
20              min[H] = A[i]
21          else insert A[i] into H's root list
22              if A[i].key < H.min.key
23                  H.min = A[i]
```

FIB-HEAP-LINK(H, y, x)

```
1  remove y from the root list of H
2  make y a child of x, incrementing x.degree
3  y.mark = FALSE
```

رویه‌ی CONSOLIDATE به صورت زیر کار می‌کند. خطوط ۱-۳، آرایه‌ی A را تخصیص، و با قرار دادن هر ورودی آن با NIL، مقداردهی اولیه می‌کنند. حلقه‌ی for در خطوط ۴-۱۴ هر ریشه‌ی w را در لیست ریشه پردازش می‌کند. همین طور که ریشه‌ها را به هم متصل می‌کنیم، ممکن است w به یک گره‌ی دیگر متصل شده و دیگر یک ریشه نباشد. در هر حال، w همیشه در درختی است با یک ریشه‌ی x، که ممکن است خود w باشد یا نباشد. از آن جایی که حداکثر یک ریشه با هر درجه می‌خواهیم، آرایه‌ی A را بررسی می‌کنیم تا ببینیم که آیا ریشه‌ای مانند y با درجه‌ی یکسان با x دارد یا نه. اگر چنین ریشه‌ای وجود داشت، ریشه‌های x و y را به یکدیگر متصل می‌کنیم، البته با این تضمین که پس از اتصال، x همچنان ریشه باقی می‌ماند. یعنی در صورتی که کلید y کوچک‌تر از کلید x بود، اتصال را بعد از جابه‌جایی اشاره‌گرهای گره‌های x و y انجام می‌دهیم. پس از اتصال y به x درجه‌ی x یکی افزایش یافته است، و بنابراین این فرایند را ادامه می‌دهیم، که عبارت است از اتصال x و یک ریشه‌ی دیگر که درجه‌ی آن با درجه‌ی جدید x برابر است، تا این که هیچ ریشه‌ی دیگری با درجه‌ی یکسان با x باقی نماند. سپس ورودی مناسب A را مقداردهی می‌کنیم تا به x اشاره کند، به طوری که وقتی بعداً دوباره ریشه‌ها را پردازش می‌کنیم، بدانیم که x یک ریشه با درجه‌ی یکتا است که قبلاً آن را پردازش کرده‌ایم. وقتی این حلقه‌ی for پایان می‌یابد، حداکثر یک ریشه با هر درجه باقی خواهد ماند، و آرایه‌ی A به هر یک از ریشه‌های باقی مانده اشاره می‌کند.

حلقه‌ی while خطوط ۷-۱۳ مکرراً x را که ریشه‌ی درخت شامل گره‌ی w است به یک درخت دیگر که درجه‌ی ریشه‌ی آن با درجه‌ی x برابر است، پیوند می‌زند، تا زمانی که هیچ ریشه‌ی دیگری آن درجه را نداشته باشد. این حلقه‌ی while ثابت حلقه‌ی زیر را حفظ می‌کند:

- در آغاز هر بار تکرار حلقه‌ی while داریم $d = x.degree$.

از این ثابت حلقه به صورت زیر استفاده می‌کنیم:

- آغاز: خط ۶ تضمین می‌کند که اولین بار که وارد حلقه می‌شویم، ثابت حلقه برقرار است.
- ادامه: در هر بار تکرار حلقه‌ی **while**، ورودی آرایه‌ی $A[d]$ به یک ریشه‌ی y اشاره می‌کند. چون $d = x.degree = y.degree$، باید x را به y پیوند دهیم. از x و y هر کدام که کلید کوچک‌تری داشته باشد، بعد از عملیات پیوند پدر دیگری خواهد شد، و بنابراین در خطوط ۹–۱۰ در صورت لزوم جای اشاره‌گرهای x و y عوض می‌شود. سپس در خط ۱۱ با فراخوانی FIB-HEAP-LINK(H, y, x) گره‌ی y را به x پیوند می‌دهیم. این فراخوانی $x.degree$ را افزایش می‌دهد، ولی $y.degree$ را بدون تغییر می‌گذارد تا مقدار آن d باقی بماند. چون گره‌ی y دیگر ریشه نیست، اشاره‌گر آن در خط ۱۲ از آرایه‌ی A حذف می‌شود. از آن جایی که فراخوانی FIB-HEAP-LINK مقدار $x.degree$ را افزایش می‌دهد، خط ۱۳ ثابت $d = x.degree$ را بازسازی می‌کند.
- پایان: حلقه‌ی **while** را تا زمانی تکرار می‌کنیم که $A[d] = \text{NIL}$، که در این حالت هیچ ریشه‌ی دیگری با درجه‌ی یکسان با x وجود ندارد.

پس از پایان حلقه‌ی **while**، در خط ۱۲ ورودی $A[d]$ را با x مقداردهی می‌کنیم، و تکرار بعدی حلقه‌ی **for** را انجام می‌دهیم.

شکل‌های ۱۹–۴(پ)–(ث) آرایه‌ی A و درخت‌های حاصل را بعد از سه تکرار اول حلقه‌ی **for** در خطوط ۴–۱۴ نشان می‌دهند. در تکرار بعدی حلقه‌ی **for** سه پیوند انجام می‌شود؛ نتایج آن‌ها در شکل‌های ۱۹–۴(ج)–(چ) نشان داده شده است. شکل‌های ۱۹–۴(خ)–(ر) نتیجه‌ی چهار تکرار بعدی حلقه‌ی **for** را نشان می‌دهند.

تنها کار باقی مانده جمع بندی است. وقتی حلقه‌ی **for** خطوط ۴–۱۴ پایان می‌یابد، خط ۱۵ لیست ریشه را خالی می‌کند، و خطوط ۱۶–۲۳ آن را از روی آرایه‌ی A بازسازی می‌کند. هرم فیبوناچی حاصل در شکل ۱۹–۴(ز) نشان داده شده است. پس از مستحکم‌سازی لیست ریشه، FIB-HEAP-EXTRACT-MIN با کاهش $H.n$ در خط ۱۱ و بازگرداندن یک اشاره‌گر به گره‌ی حذف شده‌ی z در خط ۱۲ کار را پایان می‌دهد.

اکنون آماده هستیم که نشان دهیم هزینه‌ی سرشکن استخراج گره‌ی کمینه‌ی یک هرم فیبوناچی با n گره، $O(D(n))$ است. فرض کنید H نشان دهنده‌ی هرم فیبوناچی دقیقاً قبل از عملیات FIB-HEAP-EXTRACT-MIN باشد.

هزینه‌ی واقعی استخراج گره‌ی کمینه را می‌توان به صورت زیر محاسبه کرد. یک بخش $O(D(n))$ از این جا می‌آید که حداکثر $D(n)$ فرزند گره‌ی کمینه وجود دارد که در FIB-HEAP-EXTRACT-MIN پردازش می‌شود، به علاوه‌ی کارهای انجام شده در خطوط ۲–۳ و ۱۶–۲۳ رویه‌ی CONSOLIDATE. تحلیل بخش مربوط به حلقه‌ی **for** خطوط ۴–۱۴ باقی می‌ماند. اندازه‌ی لیست ریشه در زمان فراخوانی CONSOLIDATE حداکثر $D(n)+t(H)-1$ است، چرا که این لیست شامل گره‌های لیست ریشه‌ی اولیه‌ی $t(H)$ است، منهای گره‌ی استخراج شده، به علاوه‌ی فرزندان گره‌ی استخراج شده، که تعداد

آن حداکثر $D(n)$ خواهد بود. در یک تکرار خاص از حلقه‌ی for خطوط ۴-۱۴، تعداد تکرارهای حلقه‌ی while خطوط ۷-۱۳ به لیست ریشه بستگی دارد. ولی می‌دانیم که در هر بار عبور از حلقه‌ی while خطوط ۶-۱۲، یکی از ریشه‌ها به یکی دیگر از ریشه‌ها پیوند زده می‌شود، و بنابراین کل کار انجام شده در حلقه‌ی for حداکثر با $D(n)+t(H)$ نسبت خطی دارد. بنابراین کل کار واقعی انجام شده در استخراج گره‌ی کمینه $O(D(n)+t(H))$ است.

پتانسیل قبل از استخراج گره‌ی کمینه $t(H)+2m(H)$ است، و پتانسیل بعد از آن حداکثر $(D(n)+1)+2m(H)$ خواهد بود، چرا که حداکثر $D(n)+1$ ریشه باقی می‌ماند و حین این عملیات هیچ گره‌ای علامت‌گذاری نمی‌شود. بنابراین هزینه‌ی سرشکن حداکثر برابر است با

$$\begin{aligned} &O(D(n)+t(H))+((D(n)+1)+2m(H))-(t(H)+2m(H)) \\ &=O(D(n))+O(t(H))-t(H) \\ &=O(D(n)) \end{aligned}$$

چرا که می‌توانیم واحد پتانسیل را طوری افزایش دهیم که به ثابت‌های مخفی درون $O(t(H))$ غلبه کند. به صورت شهودی، هزینه‌ی انجام هر اتصال توسط کاهش پتانسیل به خاطر کاهش تعداد ریشه‌ها توسط پیوند پرداخت می‌شود. در بخش ۱۹-۴ خواهیم دید که $D(n)=O(\lg n)$، و بنابراین هزینه‌ی سرشکن استخراج گره‌ی کمینه $O(\lg n)$ است.

تمرین‌ها

۱۹-۳-۱ هرم فیبوناچی را نشان دهید که از فراخوانی FIB-EXTRACT-MIN بر روی هرم فیبوناچی نشان داده شده در شکل ۱۹-۳(ز) حاصل می‌شود.

۱۹-۳ کاهش یک کلید و حذف یک گره

در این بخش نشان می‌دهیم که چگونه می‌توان در زمان سرشکن $O(1)$ کلید یک گره را در یک هرم فیبوناچی کاهش داد، و همچنین خواهیم دید که چگونه می‌توان در زمان سرشکن $O(D(n))$ هر گره‌ای را از یک هرم فیبوناچی با n گره حذف کرد. در بخش ۱۹-۴ نشان خواهیم داد که درجه‌ی بیشینه از مرتبه‌ی $O(\lg n)$است، که ایجاب می‌کند که FIB-HEAP-EXTRACT-MIN و FIB-HEAP-DELETE در زمان سرشکن $O(\lg n)$ اجرا شوند.

کاهش یک کلید

در سودوکد زیر برای عملیات FIB-HEAP-DECREASE-KEY، مانند قبل فرض می‌کنیم که حذف یک گره از یک لیست پیوندی هیچ یک از فیلدهای ساختاری را در گره‌ی حذف شده تغییر نمی‌دهد.

```
FIB-HEAP-DECREASE-KEY(H, x, k)
1  if k > x.key
```

```
2       error "new key is greater than current key"
3   x.key = k
4   y = x.p
5   if y ≠ NIL and x.key < y.key
6       CUT(H, x, y)
7       CASCADING-CUT(H, y)
8   if x.key < H.min.key
9       H.min = x

CUT(H, x, y)
1   remove x from the child list of y, decrementing y.degree
2   add x to the root list of H
3   x.p = NIL
4   x.mark = FALSE

CASCADING-CUT(H, y)
1   z = y.p
2   if z ≠ NIL
3       if y.mark = FALSE
4           y.mark = TRUE
5       else CUT(H, y, z)
6           CASCADING-CUT(H, z)
```

رویه‌ی FIB-HEAP-DECREASE-KEY به صورت زیر کار می‌کند. خطوط ۱-۳ اطمینان حاصل می‌کنند که کلید جدید بزرگ‌تر از کلید فعلی گره‌ی x نیست، و سپس کلید جدید را به x نسبت می‌دهند. اگر x یک ریشه باشد و یا $x.key \geq y.key$، که در آن y پدر x است، آن گاه نیازی به هیچ تغییر ساختاری نیست، چرا که ترتیب هرم کمینه نقض نشده است. خطوط ۴-۵ این موقعیت را بررسی می‌کنند.

اگر ترتیب هرم کمینه نقض شده باشد تغییرات بسیاری ممکن است رخ دهد. با برش x در خط ۶ شروع می‌کنیم. رویه‌ی CUT اتصال میان x و پدر آن y را «می‌بُرد»، و x را تبدیل به یک ریشه می‌کند.

از فیلدهای *mark* برای رسیدن به کران‌های زمانی مورد نظر استفاده می‌کنیم. این فیلدها مقدار کمی از گذشته‌ی هر گره را نگه می‌دارند. فرض کنید که رخدادهای زیر برای x اتفاق افتاده باشد:

۱. در یک زمان، x یک ریشه بوده است،
۲. سپس x به یک گره‌ی دیگر متصل (و تبدیل به فرزند آن) شده است،
۳. سپس دو فرزند x به وسیله‌ی برش حذف شده‌اند.

همین که فرزند دوم قطع شد، x را از پدرش می‌بریم و آن را تبدیل به ریشه‌ی جدید می‌کنیم. اگر مرحله‌ی ۱ و ۲ انجام و یکی از فرزندان y بریده شده باشد، فیلد $x.mark$ برابر با TRUE است. بنابراین رویه‌ی CUT در خط ۴ $x.mark$ را پاک می‌کند، چرا که مرحله‌ی ۱ را انجام می‌دهد. (اکنون

می‌توانیم ببینیم که چرا خط ۳ رویه‌ی FIB-HEAP-LINK فیلد $x.mark$ را پاک می‌کند: گره‌ی y به یک گره‌ی دیگر متصل است، و بنابراین مرحله‌ی ۲ انجام می‌شود. دفعه‌ی بعدی که یک فرزند y بریده می‌شود، $y.mark$ با TRUE مقداردهی می‌شود.)

هنوز کار ما تمام نشده است، چرا که ممکن است از زمانی که y به یک گره‌ی دیگر متصل شده بود، x دومین فرزندی باشد که از پدرش y بریده می‌شود. بنابراین، خط ۷ رویه‌ی FIB-HEAP-DECREASE-KEY یک عملیات *برش آبشاری* (cascading-cut) بر روی y انجام می‌دهد. اگر y یک ریشه باشد، آن گاه تست خط ۲ رویه‌ی CASCADING-CUT باعث می‌شود که رویه بازگشت کند. اگر y بدون علامت باشد، رویه آن را در خط ۴ علامت‌گذاری کرده، چرا که اولین فرزند آن تازه بریده شده است، و سپس بازگشت می‌کند. با این حال اگر y علامت گذاری شده باشد تازه دومین فرزند خود را از دست داده است؛ y در خط ۵ بریده می‌شود و CASCADING-CUT در خط ۶ خود را به صورت بازگشتی بر روی z، پدر y فراخوانی می‌کند. رویه‌ی CASCADING-CUT روی درخت در مسیر بالایی آن قدر بازگشت می‌کند که یا یک ریشه و یا یک گره‌ی بدون علامت یافت شود.

وقتی تمام برش‌های آبشاری رخ دادند، خطوط ۸-۹ رویه‌ی FIB-HEA-DECREASE-KEY با به هنگام‌سازی $H.min$ در صورت لزوم، کار را تمام می‌کنند. تنها گره‌ای که کلید آن تغییر کرده، گره‌ی x است که کلید آن کاهش یافته است. بنابراین گره‌ی کمینه‌ی جدید یا گره‌ی کمینه‌ی قبلی است و یا x.

شکل ۱۹-۵ اجرای دو فراخوانی FIB-HEAP-DECREASE-KEY را نشان می‌دهد، که با هرم فیبوناچی نشان داده شده در شکل ۱۹-۵(الف) شروع می‌شود. فراخوانی اول، که در شکل ۱۹-۵(ب) نشان داده شده است، شامل هیچ برش آبشاری نمی‌شود. فراخوانی دوم، که در شکل ۱۹-۵(پ)-(ث) نشان داده شده است، شامل دو برش آبشاری است.

اکنون نشان می‌دهیم که هزینه‌ی سرشکن FIB-HEAP-DECREASE-KEY فقط $O(1)$ است. با تعیین هزینه‌ی واقعی آن شروع می‌کنیم. رویه FIB-HEAP-DECREASE-KEY به زمان $O(1)$ نیاز دارد، به علاوه‌ی زمان انجام برش‌های آبشاری. فرض کنید که CASCADING-CUT به تعداد c بار به صورت بازگشتی از یک احضار خاص FIB-HEAP-DECREASE-KEY فراخوانی شده است (فراخوانی انجام شده توسط خط ۷ رویه‌ی FIB-HEAP-DECREASE-KEY، و پس از آن $c-1$ فراخوانی بازگشتی CASCADING-CUT). هر فراخوانی CASCADING-CUT بدون در نظر گرفتن فراخوانی‌های بازگشتی به زمان $O(1)$ نیاز دارد. بنابراین هزینه‌ی واقعی FIB-HEAP-DECREASE-KEY ، با در نظر گرفتن تمام فراخوانی‌های بازگشتی $O(c)$ است.

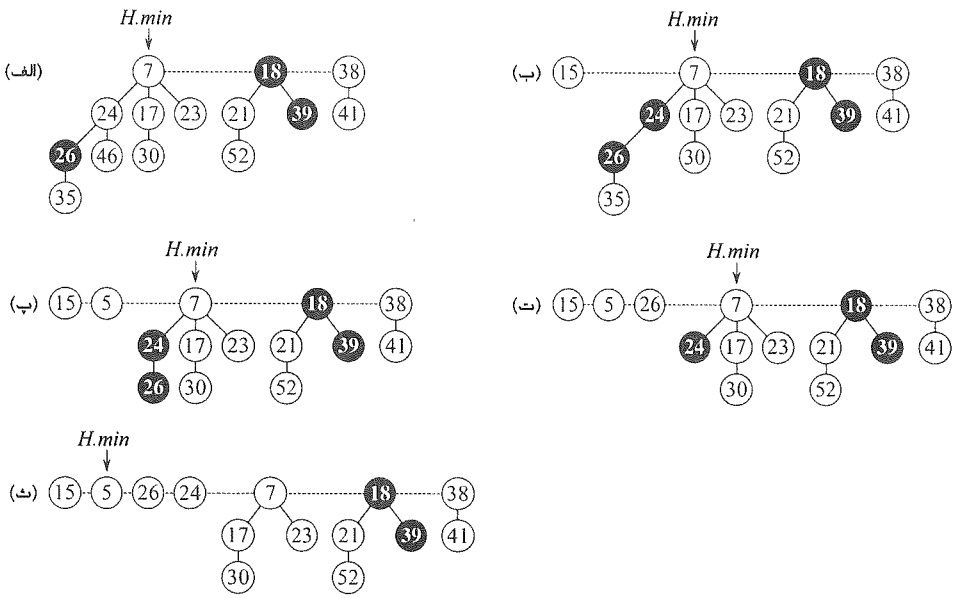
سپس تغییرات پتانسیل را محاسبه می‌کنیم. فرض کنید H نشان دهنده‌ی هرم فیبوناچی دقیقاً قبل از عملیات FIB-HEAP-DECREASE-KEY باشد. فراخوانی CUT در خط ۶ رویه‌ی FIB-HEAP-DECRESE-KEY یک درخت جدید با ریشه‌ی x می‌سازد، و بیت علامت (mark) x را FALSE می‌کند (که ممکن است از قبل FALSE بوده باشد). هر فراخوانی بازگشتی CASCADING-CUT، به جز آخرین فراخوانی، یک گره‌ی علامت‌گذاری شده را بریده و بیت علامت را پاک می‌کند. بعد از آن

$t(H)+c$ درخت ($t(H)$ درخت اولیه، $c-1$ درخت تولید شده توسط برش‌های آبشاری، و درخت با ریشه‌ی x) و حداکثر $m(H)-c+2$ گره‌ی علامت‌دار (علامت $c-1$ تا از آن‌ها توسط برش‌های آبشاری پاک شده است و آخرین فراخوانی CASCADING-CUT احتمالاً یک گره را علامت‌گذاری کرده است) وجود دارد. بنابراین تغییر پتانسیل حداکثر برابر است با:

$$((t(H)+c)+2(m(H)-c+2))-(t(H)+2m(H))=4-c$$

پس هزینه‌ی سرشکن FIB-HEAP-DECREASE-KEY حداکثر برابر است با:

$$O(c)+4-c=O(1)$$



شکل ۱۹-۵ دو فراخوانی FIB-HEAP-DECREASE-KEY. (الف) هرم فیبوناچی اولیه. (ب) کلید با مقدار ۴۶ به ۱۵ کاهش می‌یابد، گره‌ی آن تبدیل به یک ریشه می‌شود، و پدر آن (با کلید ۲۴)، که قبلاً بدون علامت بوده است، علامت‌گذاری می‌شود. (پ)-(ث) کلید ۳۵ به ۵ کاهش می‌یابد. در بخش (پ) گره‌ای که اکنون کلید آن ۵ است، یک ریشه می‌شود. پدر آن با کلید ۲۶ علامت‌گذاری می‌شود، و بنابراین یک برش آبشاری رخ می‌دهد. گره‌ی با کلید ۲۶ از پدر آن بریده و در (ت) تبدیل به یک ریشه‌ی بدون علامت می‌شود. یک برش آبشاری دیگر رخ می‌دهد، چرا که گره‌ی با کلید ۲۴ هم علامت‌گذاری شده است. در بخش (ث) این گره از پدر خود بریده و تبدیل به یک ریشه‌ی بدون علامت می‌شود. برش آبشاری در این نقطه متوقف می‌شود، چرا که گره‌ی با کلید ۷ یک ریشه است. (حتی اگر این گره ریشه نبود برش آبشاری متوقف می‌شد، چرا که این گره علامت ندارد.) نتیجه‌ی عملیات FIB-HEAP-DECREASE-KEY در بخش (ث) نشان داده شده است، که در آن $H.min$ به گره‌ی کمینه‌ی جدید اشاره می‌کند.

چرا که می‌توانیم واحد پتانسیل را طوری بزرگ کنیم که به ثابت‌های مخفی درون $O(c)$ غلبه کند.

اکنون می‌توانیم ببینیم چرا تابع پتانسیل طوری انتخاب شده بود که حاوی یک جمله باشد که دو برابر تعداد گره‌های علامت‌گذاری شده است. وقتی یک گره‌ی علامت‌دار y توسط یک برش آبشاری بریده می‌شود، بیت علامت آن پاک می‌شود، و بنابراین پتانسیل آن ۲ واحد کاهش می‌یابد. یک واحد پتانسیل هزینه‌ی برش و پاک کردن بیت علامت را می‌دهد، و دیگری، افزایش پتانسیل به خاطر تبدیل شدن y به ریشه را جبران می‌کند.

حذف یک گره

حذف یک گره از یک هرم فیبوناچی با n گره در زمان سرشکن $O(D(n))$ بسیار ساده است، همان طور که رویه‌ی زیر این کار را انجام می‌دهد. فرض می‌کنیم هیچ کلیدی با مقدار $-\infty$ در هرم فیبوناچی وجود ندارد.

```
FIB-HEAP-DELETE(H, x)
1  FIB-HEAP-DECREASE-KEY(H, x, -∞)
2  FIB-HEAP-EXTRACT-MIN(H)
```

FIB-HEAP-DELETE مشابه BINOMIAL-HEAP-DELETE است. با دادن مقدار کوچک و یکتای $-\infty$ به x، آن را تبدیل به گره‌ی کمینه در هرم فیبوناچی می‌کند. سپس گره‌ی x توسط رویه‌ی FIB-HEAP-EXTRACT-MIN از هرم فیبوناچی حذف می‌شود. هزینه‌ی سرشکن FIB-HEAP-DELETE مجموع هزینه‌ی سرشکن $O(1)$ مربوط به FIB-HEAP-DECREASE-KEY و هزینه‌ی سرشکن $O(D(n))$ مربوط به FIB-HEAP-EXTRACT-MIN است. از آن جایی که $D(n) = O(\lg n)$، همان طور که در بخش ۱۹-۴ خواهیم دید، هزینه‌ی سرشکن FIB-HEAP-DELETE برابر است با $O(\lg n)$.

تمرین‌ها

۱۹-۳-۱ فرض کنید یک ریشه‌ی x در یک هرم فیبوناچی علامت‌گذاری شده است. توضیح دهید که چگونه x یک ریشه‌ی علامت‌دار شده است. بحث کنید که علامت‌گذاری شدن x در تحلیل اهمیتی ندارد، حتی وقتی که ریشه‌ای نباشد که ابتدا به یک گره‌ی دیگر متصل شده و سپس یک فرزند خود را از دست داده باشد.

۱۹-۳-۲ با استفاده از تحلیل متراکم هزینه‌ی سرشکن $O(1)$ را برای FIB-HEAP-DECREAES-KEY به صورت یک هزینه‌ی متوسط برای هر عملیات توجیه کنید.

۱۹-۴ تعیین کران درجه‌ی بیشینه

برای اثبات این که هزینه‌ی سرشکن FIB-HEAP-EXTRACT-MIN و FIB-HEAP-DELETE برابر $O(\lg n)$ است، باید نشان دهیم که کران بالای $D(n)$ بر روی درجه‌ی هر گره‌ای در یک هرم فیبوناچی با n گره $O(\lg n)$ است. به خصوص نشان خواهیم داد که $D(n) \le \lfloor \log_\phi n \rfloor$، که در آن ϕ

نسبت طلایی است که در تساوی (۳-۲۴) به صورت زیر تعریف شده است:

$$\phi = (1+\sqrt{5})/2 = 1{,}618033\ldots$$

کلید تحلیل به صورت زیر است. برای هر گره‌ی x در هرم فیبوناچی، $size(x)$ را به صورت تعداد گره‌های زیردرخت x، از جمله خود x تعریف می‌کنیم. (توجه کنید که نیازی نیست که x در لیست ریشه باشد – این گره می‌تواند هر گره‌ای باشد.) نشان خواهیم داد که $size(x)$ نسبت به $x.degree$ نمایی است. به خاطر داشته باشید که $x.degree$ همیشه به صورت یک شمارنده‌ی دقیق از درجه‌ی x نگه داری می‌شود.

لم ۱۹-۱ فرض کنید که x هر گره‌ای در هرم فیبوناچی باشد، و همچنین $x.degree = k$. فرض کنید که $y_1, y_2, \ldots, y_k$ نشان‌دهنده‌ی فرزندان x به ترتیبی باشند که به x متصل شده‌اند، از زودترین به دیرترین. در این صورت $y_1.degree \geq 0$، و برای $i = 2,3,\ldots,k$ داریم $y_i.degree \geq i - 2$.

اثبات به وضوح، $y_1.degree \geq 0$.

برای $i \geq 2$، توجه می‌کنیم که وقتی y_i به x متصل شده بود، تمام $y_1, y_2, \ldots, y_k$ فرزندان x بوده‌اند، پس در آن زمان $x.degree = i - 1$. گره‌ی y_i فقط در صورتی (توسط CONSOLIDATE) به x متصل می‌شود که $x.degree = y_i.degree$، بنابراین باید در آن زمان $y_i.degree = i - 1$ هم برقرار بوده باشد. از آن موقع گره‌ی y_i حداکثر یک فرزند را از دست داده است، چرا که اگر دو فرزند را از دست می‌داد (توسط CASCADING-CUT) از x بریده می‌شد. نتیجه می‌گیریم که $y_i.degree \geq i - 2$. ■

بالاخره به بخشی از تحلیل می‌رسیم که نام «هرم‌های فیبوناچی» را توضیح می‌دهد. از بخش ۳-۲ به یاد بیاورید که برای $k = 0, 1, 2, \ldots$، امین عدد فیبوناچی به صورت رابطه‌ی بازگشتی

$$F_k = \begin{cases} 0 & \text{اگر } k = 0 \\ 1 & \text{اگر } k = 1 \\ F_{k-1} + F_{k-2} & \text{اگر } k \geq 2 \end{cases}$$

تعریف شده است. لم زیر روش دیگری برای توصیف F_k می‌دهد.

لم ۱۹-۲ برای تمام اعداد صحیح $k \geq 0$،

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

اثبات اثبات زیر به صورت استقرا بر روی k است. وقتی $k = ۰$،

$$\begin{aligned} ۱ + \sum_{i=۰}^{۰} F_i &= ۱ + F_۰ \\ &= ۱ + ۰ \\ &= ۱ \\ &= F_۲ \end{aligned}$$

اکنون فرض استقرا را به صورت $F_{k+۱} = ۱ + \sum_{i=۰}^{k-۱} F_i$ در نظر می‌گیریم، و داریم

$$\begin{aligned} F_{k+۲} &= F_k + F_{k+۱} \\ &= F_k + \left(۱ + \sum_{i=۰}^{k-۱} F_i \right) \\ &= ۱ + \sum_{i=۰}^{k} F_i \end{aligned}$$

■

لم ۴-۱۹ برای تمام اعداد صحیح $k \geq ۰$، $(k+۲)$مین عدد فیبوناچی رابطه‌ی $F_{k+۲} \geq \phi^k$ را ارضا می‌کند.

اثبات اثبات به کمک استقرا بر روی k انجام می‌شود. حالت‌های پایه عبارتند از $k = ۰$ و $k = ۱$. وقتی $k = ۰$، داریم $F_۲ = ۱ = \phi^۰$، و وقتی $k = ۱$ داریم $F_۳ = ۲ > ۱/۶۱۹ > \phi^۱$. گام استقرا برای $k \geq ۲$ است، و فرض می‌کنیم که $F_{i+۲} > \phi^i$ برای $i = ۰, ۱, \ldots, k-۱$. به خاطر بیاورید که ϕ یک ریشه‌ی مثبت تساوی (۳-۲۳) است، که عبارت است از $x^۲ = x + ۱$. بنابراین داریم

$$\begin{aligned} F_{k+۲} &= F_{k+۱} + F_k \\ &\geq \phi^{k-۱} + \phi^{k-۱} && \text{(طبق فرض استقرا)} \\ &= \phi^{k-۲}(\phi + ۱) \\ &= \phi^{k-۲} \cdot \phi^۲ && \text{(طبق تساوی (۳-۲۳))} \\ &= \phi^k \end{aligned}$$

■

لم زیر و نتیجه‌ی آن تحلیل را کامل می‌کنند.

لم ۴-۱۹ فرض کنید x یک گره در یک هرم فیبوناچی باشد، و فرض کنید $k = x.degree$. آن گاه $size(x) \geq F_{k+۲} \geq \phi^k$، که در آن $\phi = (۱ + \sqrt{۵})/۲$.

اثبات فرض کنید s_k نشان‌دهنده‌ی مقدار کمینه‌ی $size(z)$ بر روی تمام گره‌های z باشد به طوری که $z.degree = k$. به صورت شهودی، $s_0 = 1$، $s_1 = 2$، و $s_2 = 3$. عدد s_k حداکثر $size(x)$ است، و به وضوح مقدار s_k به صورت صعودی با k افزایش می‌یابد. مانند لم ۱۹-۱، فرض کنید $y_1, y_2, ..., y_k$ نشان‌دهنده‌ی فرزندان x به ترتیبی باشند که به x متصل شده‌اند. برای محاسبه‌ی کران پایین $size(x)$، یک واحد برای خود x و یک واحد برای اولین فرزند آن y_1 (که برای آن $size(y_1) \geq 1$) در نظر می‌گیریم، که می‌دهد

$$\begin{aligned} size(x) &\geq s_k \\ &= 2 + \sum_{i=2}^{k} s_{y_i.degree} \\ &\geq 2 + \sum_{i=2}^{k} s_{i-2} \end{aligned}$$

که در آن خط آخر از لم ۱۹-۱ (به طوری که $y_i.degree \geq i - 2$) و یکنواختی s_k (به طوری که $s_{y_i.degree} \geq s_{i-2}$) نتیجه می‌شود.

اکنون به کمک استقرا بر روی k نشان می‌دهیم که برای تمام اعداد صحیح نامنفی k داریم $s_k \geq F_{k+2}$. پایه‌های $k = 0$ و $k = 1$ بدیهی هستند. برای گام استقرا فرض می‌کنیم که برای $i = 0, 1, ..., k-1$، $k \geq 2$ و $s_i \geq F_{i+2}$. داریم

$$\begin{aligned} s_k &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\ &\geq 2 + \sum_{i=2}^{k} F_i \\ &= 1 + \sum_{i=0}^{k} F_i \\ &= F_{k+2} && \text{(طبق لم ۱۹-۲)} \\ &\geq \phi^k && \text{(طبق لم ۱۹-۳)} \end{aligned}$$

بنابراین نشان دادیم که $size(x) \geq s_k \geq F_{k+2} \geq \phi^k$. ■

نتیجه‌ی ۱۹-۵ درجه‌ی بیشینه ($D(n)$) برای هر گره در یک هرم فیبوناچی با n گره، $O(\lg n)$ است.

اثبات فرض کنید x یک گره در یک هرم فیبوناچی با n گره باشد، و $k = x.degree$. طبق لم ۱۹-۳، داریم $n \geq size(x) \geq \phi^k$. گرفتن لگاریتم پایه‌ی ϕ می‌دهد $k \leq \log_\phi n$. (در واقع چون k یک عدد صحیح است، $k \leq \lfloor \log_\phi n \rfloor$.) بنابراین درجه‌ی بیشینه‌ی هر گره‌ای $O(\lg n)$ است. ■

تمرین‌ها

۱-۴-۱۹ پروفسور Pinocchio ادعا می‌کند که ارتفاع یک هرم فیبوناچی با n گره $O(\lg n)$ است. با ارائه‌ی یک دنباله از عملیات که برای هر عدد صحیح n یک هرم فیبوناچی تولید می‌کند که فقط از یک درخت خطی حاوی n گره تشکیل شده است، نشان دهید که پروفسور اشتباه می‌کند.

۲-۴-۱۹ فرض کنید قانون برش آبشاری را طوری اصلاح می‌کنیم که یک گره‌ی x را زمانی از پدر آن می‌برد که kامین فرزند خود را از دست می‌دهد، برای یک ثابت k. (قانون بخش ۱۹-۳ از $k = 2$ استفاده می‌کند.) برای چه مقادیری از k داریم $D(n) = O(\lg n)$؟

مسائل

۱-۱۹ ***پیاده‌سازی جایگزین برای حذف***

پروفسور Pisano نسخه‌ی زیر را از رویه‌ی FIB-HEAP-DELETE پیشنهاد کرده است، با این ادعا که اگر گره‌ای که حذف می‌شود گره‌ای نباشد که $H.min$ به آن اشاره می‌کند، این رویه سریع‌تر اجرا می‌شود.

```
PISANO-DELETE(H, x)
1  if x == H.min
2      FIB-HEAP-EXTRACT-MIN(H)
3  else y = x.p
4      if y ≠ NIL
5          CUT(H, x, y)
6          CASCADING-CUT(H, y)
7      add x's child list to the root list of H
8      remove x from the root list of H
```

I. ادعای پروفسور مبنی بر این که این رویه سریع‌تر اجرا می‌شود، تا حدودی بر پایه‌ی این فرض است که خط ۷ در زمان واقعی $O(1)$ اجرا می‌شود. چه چیزی در مورد این فرض اشتباه است؟

II. یک کران بالای خوب بر روی زمان واقعی PISANO-DELETE برای وقتی که x به $H.min$ اشاره نمی‌کند، بدهید. کران شما باید بر حسب $x.degree$ و c، تعداد فراخوانی‌های رویه‌ی CASCADING-CUT باشد.

III. فرض کنید که PISANO-DELETE (H, x) را فراخوانی می‌کنیم، و فرض کنید H' هرم فیبوناچی حاصل باشد. با فرض این که گره‌ی x ریشه نیست، کران پتانسیل H' را بر حسب $x.degree$، c، $t(H)$، و $m(H)$ تعیین کنید.

IV. نتیجه بگیرید که هزینه‌ی سرشکن PISANO-DELETE به صورت حدی بهتر از FIB-HEAP-DELETE نیست، حتی زمانی که $x \neq H.min$.

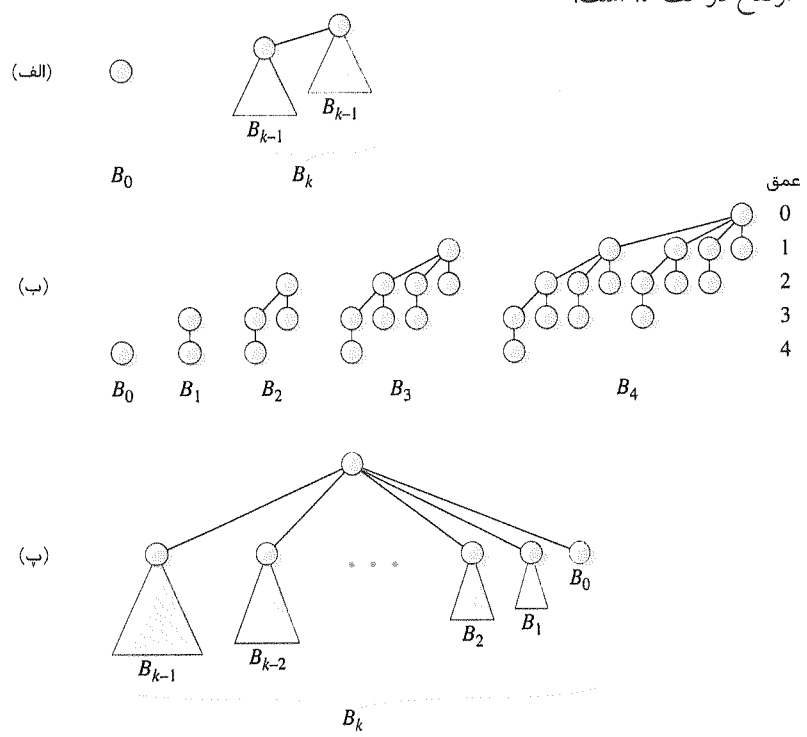
۱۹-۲ درخت‌های دوجمله‌ای و هرم‌های دوجمله‌ای

درخت دوجمله‌ای (binomial tree) B_k یک درخت مرتب است (بخش ب-۵-۲ را ببینید) که به صورت بازگشتی تعریف می‌شود. همان طور که در شکل ۱۹-۶(الف) نشان داده شده است، درخت دوجمله‌ای B_0 از یک گره‌ی تنها تشکیل شده است. درخت دوجمله‌ای B_k از دو درخت دوجمله‌ای B_{k-1} تشکیل می‌شود که به یکدیگر متصل شده‌اند، به طوری که ریشه‌ی یکی، چپ‌ترین فرزند ریشه‌ی دیگری است. شکل ۱۹-۶(ب) درخت‌های دوجمله‌ای B_0 تا B_4 را نشان می‌دهد.

I. نشان دهید که در درخت دوجمله‌ای B_k،

۱. 2^k گره وجود دارد،

۲. ارتفاع درخت k است.



شکل ۱۹-۶ (الف) تعریف بازگشتی درخت دوجمله‌ای B_k. مثلث‌ها نشان‌دهنده‌ی زیردرخت‌های ریشه‌دار هستند. (ب) درخت‌های دوجمله‌ای B_0 تا B_4. عمق گره‌ها در B_4 نشان داده شده است. (پ) روش دیگری برای تصور B_k.

۳. دقیقاً $\binom{k}{i}$ گره در عمق i وجود دارد، برای $i = 0, 1, \ldots, k$، و

۴. درجه‌ی ریشه k است، که از درجه‌ی تمام گره‌های دیگر بزرگ‌تر است؛ به علاوه شکل ۱۹-۶(پ) نشان می‌دهد که، اگر فرزندان ریشه را از چپ به راست با $k-1$، $k-2$، ...، 0 شماره‌گذاری کنیم، آن گاه فرزند i ریشه‌ی یک زیردرخت B_i است.

یک *هرم دوجمله‌ای* (binomial heap) H مجموعه‌ای از درخت‌های دوجمله‌ای است که خصوصیات زیر را ارضا می‌کنند:

۱. هر گره یک کلید دارد (مانند هرم‌های فیبوناچی).

۲. هر درخت دوجمله‌ای در H از خصوصیت هرم‌های کمینه پیروی می‌کند.

۳. برای هر عدد صحیح نامنفی k، حداکثر یک درخت دوجمله‌ای در H وجود دارد که درجه‌ی آن k است.

II. فرض کنید یک هرم دوجمله‌ای H مجموعاً n گره دارد. در مورد رابطه‌ی میان درخت‌های دوجمله‌ای درون H و نمایش دودویی n بحث کنید. نتیجه بگیرید که H حداکثر $\lfloor \lg n \rfloor + 1$ درخت دوجمله‌ای دارد.

فرض کنید یک درخت دوجمله‌ای را به صورت زیر نمایش می‌دهیم. نمایش فرزند چپ-برادر راست از بخش ۱۰-۴، هر یک از درختان دوجمله‌ای درون هرم دوجمله‌ای را نمایش می‌دهد. هر گره حاوی کلید خود است، به همراه اشاره‌گرهایی به پدر، چپ‌ترین فرزند، و برادر سمت راست (این اشاره‌گرها در صورت لزوم NIL هستند)، همچنین درجه‌ی گره (که مانند هرم‌های فیبوناچی، برابر است با تعداد فرزندان گره). ریشه‌ها یک لیست ریشه‌ی یک طرفه را تشکیل می‌دهند که بر حسب درجه‌ی ریشه‌ها مرتب شده‌اند (از چپ به راست)، و دسترسی به هرم دوجمله‌ای توسط یک اشاره‌گر به اولین گره در لیست ریشه انجام می‌شود.

III. توصیف نحوه‌ی نمایش هرم‌های کمینه را کامل کنید (خصیصه‌ها را نام ببرید، توضیح دهید که چه زمانی خصیصه‌ها مقدار NIL دارند، و بگویید که لیست ریشه چگونه سازمان‌دهی می‌شود)، و نشان دهید که چگونه می‌توان هفت عمل ارائه شده بر روی هرم‌های فیبوناچی را بر روی هرم‌های دوجمله‌ای پیاده‌سازی کرد. هر یک از اعمال باید در بدترین حالت در زمان $O(\lg n)$ اجرا شوند، که در آن n تعداد گره‌های هرم دوجمله‌ای (و یا برای عملیات UNION، مجموع تعداد گره‌های دو هرم دوجمله‌ای) است. عملیات MAKE-HEAP باید در زمان ثابت اجرا شود.

IV. فرض کنید فقط نیاز داشتیم که اعمال هرم‌های قابل ادغام را بر روی هرم‌های فیبوناچی پیاده‌سازی کنیم (یعنی نیازی به پیاده‌سازی اعمال DECREASE-KEY و یا DELETE نباشد). درختان هرم فیبوناچی از چه لحاظ مشابه درختان هرم‌های دوجمله‌ای می‌شدند؟ از چه لحاظ با هم تفاوت داشتند؟ نشان دهید که در این حالت، درجه‌ی بیشینه‌ی یک هرم

فیبوناچی با n گره حداکثر $\lfloor \lg n \rfloor$ خواهد بود.

۷ پروفسور McGee یک ساختمان داده‌ی جدید بر مبنای هرم‌های فیبوناچی ابداع کرده است. یک هرم McGee ساختار مشابهی با هرم‌های فیبوناچی دارد و از اعمال هرم‌های قابل ادغام پشتیبانی می‌کند. پیاده‌سازی اعمال مشابه هرم‌های فیبوناچی است، غیر از این که در درج و اجتماع، مستحکم‌سازی در آخرین مرحله انجام می‌شود. بدترین حالت‌های زمان اجرا بر روی هرم‌های McGee چگونه است؟

۱۹-۳ *اعمال بیشتر بر روی هرم‌های فیبوناچی*

می‌خواهیم طوری هرم فیبوناچی H را تکمیل کنیم که از دو عملیات جدید پشتیبانی کند، بدون این که زمان اجرای سرشکن اعمال دیگر هرم‌های فیبوناچی تغییر کند.

ا عملیات FIB-HEAP-CHANGE-KEY (H, x, k) کلید گره‌ی x را به مقدار k تغییر می‌دهد. یک پیاده‌سازی بهینه از FIB-HEAP-CHANGE-KEY بدهید، و زمان اجرای سرشکن پیاده‌سازی خود را برای حالت‌هایی که k بزرگ‌تر، کوچک‌تر، و یا مساوی $x.key$ است، تحلیل کنید.

۱۱ یک پیاده‌سازی بهینه برای FIB-HEAP-PRUNE (H, r) بدهید، که $min(r, H.n)$ گره را از H حذف می‌کند. این که کدام گره‌ها حذف می‌شوند باید دلخواه باشد. زمان اجرای سرشکن پیاده‌سازی خود را تحلیل کنید. (*راهنمایی:* ممکن است نیاز داشته باشید که ساختمان داده و تابع پتانسیل را اصلاح کنید.)

۱۹-۴ *هرم‌های ۲-۳-۴*

در فصل ۱۸ درخت‌های ۲-۳-۴ معرفی شدند، که در آن هر گره‌ی داخلی (احتمالاً غیر از ریشه) دو، سه، یا چهار فرزند دارند، و تمام برگ‌ها عمق یکسانی دارند. در این مسئله هرم‌های ۲-۳-۴ را پیاده‌سازی خواهیم کرد، که از اعمال هرم‌های قابل ادغام پشتیبانی می‌کنند.

هرم‌های ۲-۳-۴ از جهات زیر با درختان ۲-۳-۴ تفاوت دارند. در هرم‌های ۲-۳-۴، فقط برگ‌ها دارای کلید هستند، و هر برگ x دقیقاً یک کلید در خصیصه‌ی $x.key$ نگه می‌دارد. کلیدها در برگ‌ها می‌توانند به هر ترتیبی باشند. هر گره‌ی داخلی x حاوی یک مقدار $x.small$ است که برابر است با کوچک‌ترین کلید ذخیره شده در تمام برگ‌های زیردرخت x. ریشه‌ی r حاوی یک خصیصه‌ی $r.height$ است که ارتفاع درخت را به دست می‌دهد. نهایتاً، هرم‌های ۲-۳-۴ طوری طراحی شده‌اند که در حافظه‌ی اصلی نگه‌داری شوند، و خواندن و نوشتن بر/از روی دیسک‌ها لازم نیست.

اعمال زیر را برای هرم‌های ۲-۳-۴ پیاده‌سازی کنید. در بخش‌های ۱-۷، هر عملیات باید

در زمان $O(\lg n)$ بر روی یک هرم ۲-۳-۴ با n عنصر اجرا شود. عملیات UNION در بخش VI باید در زمان $O(\lg n)$اجرا شود، که در آن n برابر است با مجموع تعداد عناصر در دو هرم ورودی.

I. MINIMUM، که یک اشاره‌گر به برگ با کوچک‌ترین کلید را بازمی‌گرداند.
II. DECREASE-KEY، که کلید یک برگ داده شده‌ی x را به یک مقدار $k \leq x.key$ کاهش می‌دهد.
III. INSERT، که برگ x با کلید k را درج می‌کند.
IV. DELETE، که یک برگ داده شده‌ی x را حذف می‌کند.
V. EXTRACT-MIN، که برگ با کوچک‌ترین کلید را استخراج می‌کند.
VI. UNION، که دو هرم ۲-۳-۴ را با هم ترکیب می‌کند، اجتماع آن‌ها را به صورت یک هرم ۲-۳-۴ بازمی‌گرداند، و هرم‌های ورودی را نابود می‌کند.

فصل

۲۰

درختان van Emde Boas

۲۰-۱ مقدمه

در فصل‌های قبل، ساختمان‌های داده‌ای را دیدیم که از اعمال صف‌های اولویت پشتیبانی می‌کردند – هرم‌های دودویی در فصل ۶، درختان قرمز–سیاه در فصل ۱۳[1]، و هرم‌های فیبوناچی در فصل ۱۹. در هر یک از این ساختمان‌های داده حداقل یک عملیات مهم به زمان $O(\lg n)$ نیاز داشت (بدترین حالت یا سرشکن). در واقع چون در تمام این ساختمان‌های داده تصمیمات بر پایه‌ی مقایسه‌ی کلیدها بود، کران پایین $\Omega(n \lg n)$ بر روی مرتب‌سازی در بخش ۸-۱ به ما می‌گوید که حداقل یکی از اعمال به زمان $\Omega(\lg n)$ نیاز دارد. چرا؟ اگر می‌توانستیم هر دوی INSERT و EXTRACT-MIN را در زمان $o(\lg n)$ پیاده‌سازی کنیم، آن گاه می‌توانستیم با فراخوانی n عملیات INSERT و n عملیات EXTRACT-MIN، مرتب‌سازی n کلید را در زمان $o(n \lg n)$ پیاده‌سازی کنیم.

ولی در فصل ۸ دیدیم که می‌توانیم از اطلاعات دیگری در مورد کلیدها استفاده کرده و آن‌ها را در زمان $o(n \lg n)$ مرتب کنیم. به طور خاص، در مرتب‌سازی شمارشی می‌توانیم n کلید را، که هر یک در بازه‌ی ۰ تا k هستند در زمان $\theta(n+k)$ مرتب کنیم، که برابر است با $\theta(n)$ اگر داشته باشیم $k = O(n)$.

از آن جایی که می‌توانیم کران پایین $\Omega(n \lg n)$ را برای کلیدهای در یک بازه‌ی محدود پشت سر بگذاریم، ممکن است از خود بپرسید که آیا می‌توان در یک حالت مشابه، هر یک از اعمال صف‌های اولویت را در زمان $o(\lg n)$ پیاده‌سازی کرد؟ در این فصل خواهیم دید که در واقع می‌توان این کار را کرد: درختان van Emde Boas از اعمال صف‌های اولویت، و چند عملیات دیگر، در زمان $O(\lg \lg n)$ در بدترین حالت پشتیبانی می‌کنند. تنها مسئله این است که کلیدها باید در بازه‌ی ۰ تا $n-1$ باشند، بدون کلید تکراری.

---

[1] در فصل ۱۳ به صورت صریح در مورد نحوه‌ی پیاده‌سازی EXTRACT-MIN و DECREASE-KEY بحث نشد، ولی به سادگی می‌توان این دو را برای هر ساختمان داده‌ای که از MINIMUM، DELETE، و INSERT پشتیبانی می‌کنند، پیاده‌سازی کرد.

به طور خاص درختان van Emde Boas از هر یک از اعمال ذکر شده در مقدمه‌ی بخش سه — SEARCH، INSERT، DELETE، MINIMUM، MAXIMUM، SUCCESSOR و PREDECESSOR — در زمان $O(\lg\lg n)$ پشتیبانی می‌کنند. در این فصل از بحث از داده‌های پیرو صرف نظر کرده و فقط روی کلیدهای ذخیره شده تمرکز می‌کنیم. چون روی کلیدها تمرکز کرده و کلیدهای تکراری را هم مجاز نمی‌دانیم، به جای توصیف عملیات SEARCH، عملیات ساده‌تر MEMBER(S, x) را پیاده‌سازی می‌کنیم، که یک مقدار بولین بازمی‌گرداند که مشخص می‌کند که آیا مقدار x در مجموعه‌ی پویای S وجود دارد یا نه.

تا این جا از پارامتر n برای دو هدف خاص استفاده کرده‌ایم: تعداد عناصر در مجموعه‌ی پویا، و دامنه‌ی مقادیر ممکن. برای جلوگیری از اشتباه بیشتر، از این به بعد از n برای نشان دادن تعداد عناصر در مجموعه‌ی پویا و از u برای نشان دادن دامنه‌ی مقادیر استفاده خواهیم کرد، به طوری که هر یک از اعمال درختان van Emde Boas در زمان $O(\lg\lg u)$ اجرا می‌شود. به مجموعه‌ی $\{0, 1, 2, \ldots, u-1\}$، *مجموعه‌ی جهانی* مقادیری می‌گوییم که می‌توان آن‌ها را ذخیره کرد، و u *اندازه‌ی مجموعه‌ی جهانی* است. در طول این فصل فرض می‌کنیم u توانی از ۲ است، یعنی $u = 2^k$ برای یک عدد صحیح $k \geq 1$.

بخش ۲۰-۱ با بررسی چند رویکرد ساده آغاز می‌کند که می‌توانند ما را در مسیر درست هدایت کنند. این رویکردها را در بخش ۲۰-۲ به پیش می‌بریم و ساختارهای van Emde Boas بدوی را معرفی می‌کنیم، که بازگشتی هستند و به زمان $O(\lg\lg u)$ برای اعمال درختان دست نمی‌یابند. بخش ۲۰-۳، ساختارهای van Emde Boas بدوی را اصلاح می‌کند تا به درختان van Emde Boas برسد، و نشان می‌دهد که چطور می‌توان تمام اعمال را در زمان $O(\lg\lg u)$ پیاده‌سازی کرد.

۲۰-۱ رویکردهای مقدماتی

در این بخش رویکردهای مختلفی را برای ذخیره‌سازی یک مجموعه‌ی پویا بررسی خواهیم کرد. با این که هیچ کدام از آن‌ها به کران زمانی $O(\lg\lg u)$ که مورد نظر ما است نمی‌رسند، ولی به کمک آن‌ها در ادامه‌ی فصل دیدی برای درک بهتر درختان van Emde Boas خواهیم داشت.

آدرس‌دهی مستقیم

آدرس‌دهی مستقیم، همان طور که در بخش ۱۱-۱ دیدیم، ساده‌ترین رویکرد برای ذخیره‌ی یک مجموعه‌ی پویا است. از آن جایی که در این فصل فقط نیاز به ذخیره‌ی کلیدها داریم، می‌توانیم رویکرد آدرس‌دهی مستقیم را طوری ساده کنیم که مجموعه‌ی پویا را به صورت یک بردار بیتی ذخیره کند، همان طور که در بخش ۱۱-۱-۲ گفته شد. برای ذخیره‌ی یک مجموعه‌ی پویا از مقادیر در مجموعه‌ی جهانی $\{0, 1, 2, \ldots, u-1\}$، یک آرایه‌ی $A[0..u-1]$ از u بیت نگه می‌داریم. خانه‌ی $A[x]$ برابر ۱ است اگر مقدار x در مجموعه‌ی پویا وجود داشته باشد، و در غیر این صورت برابر ۰ است. با

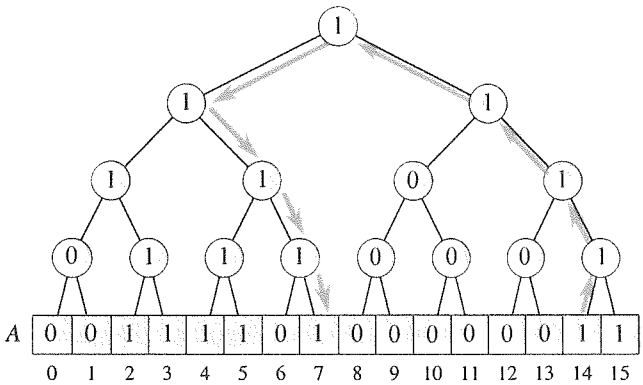
این که می‌توانیم هر یک از اعمال INSERT، DELETE و MEMBER را بر روی یک بردار بیتی در زمان $O(1)$ انجام دهیم، بقیه‌ی اعمال – MINIMUM، MAXIMUM، SUCCESSOR و PREDECESSOR – هر یک در بدترین حالت به زمان $\theta(u)$ نیاز خواهند داشت، چرا که ممکن است نیاز داشته باشیم $\theta(u)$ عنصر را بررسی کنیم.[1] برای مثال اگر یک مجموعه فقط حاوی مقادیر ۰ و $u-1$ باشد، آن گاه برای یافتن عنصر مابعد ۰، عناصر ۱ تا $u-2$ را پویش کنیم تا به مقدار ۱ در $A[u-1]$ برسیم.

تحمیل یک درخت دودویی

می‌توانیم با تحمیل یک درخت دودویی روی بردار بیتی، از پویش‌های طولانی در آن جلوگیری کنیم. شکل ۲۰-۱ یک مثال را نشان می‌دهد. ورودی‌های بردار بیتی برگ‌های درخت را تشکیل می‌دهند، و هر گره‌ی داخلی حاوی یک ۱ است اگر و فقط اگر یکی از برگ‌های زیردرخت مربوط به آن ۱ باشد. به عبارت دیگر، بیت ذخیره شده در یک گره‌ی داخلی برابر است با «یا»ی منطقی دو فرزند آن.

اعمالی که در یک بردار بیتی ساده در بدترین حالت در زمان $O(u)$ اجرا می‌شدند، اکنون از یک ساختار درختی استفاده می‌کنند:

- برای یافتن مقدار کمینه‌ی مجموعه، از ریشه آغاز کرده و به سمت پایین (برگ‌ها) حرکت می‌کنیم، و همیشه به چپ‌ترین گره‌ای می‌رویم که حاوی یک ۱ است.



شکل ۲۰-۱ یک درخت دودویی از بیت‌ها که بر روی یک بردار بیتی سوار شده است. این بردار بیتی نشان‌دهنده‌ی مجموعه‌ی $\{2,3,4,5,7,14,15\}$ است و $u=16$. هر گره‌ی داخلی حاوی یک ۱ است اگر و فقط اگر یکی از برگ‌های زیردرخت آن حاوی ۱ باشد. پیکان‌ها نشان‌دهنده‌ی مسیر طی شده برای یافتن عنصر ماقبل ۱۴ هستند.

[1] در این فصل فرض می‌کنیم که اگر مجموعه‌ی پویا تهی باشد، MINIMUM و MAXIMUM مقدار NIL را بازمی‌گردانند، و در صورتی که عنصر ورودی به ترتیب عنصر مابعد و یا عنصر ماقبل نداشته باشد، SUCCESSOR و PREDECESSOR مقدار NIL را بازمی‌گردانند.

- برای یافتن مقدار بیشینه‌ی مجموعه، از ریشه آغاز کرده و به سمت پایین (برگ‌ها) حرکت می‌کنیم، و همیشه به راست‌ترین گره‌ای می‌رویم که حاوی یک ۱ است.
- برای یافتن عنصر مابعد x از برگ با اندیس x آغاز می‌کنیم، به سمت بالا (ریشه) حرکت می‌کنیم تا به گره‌ای برسیم که فرزند سمت راست آن، z، حاوی ۱ باشد. سپس از طریق گره‌ی z به سمت پایین حرکت می‌کنیم، و همیشه به چپ‌ترین فرزندی می‌رویم که حاوی ۱ است (که متناظر است با یافتن مقدار کمینه در زیردرخت گره‌ی z).
- برای یافتن عنصر ماقبل x از برگ با اندیس x آغاز می‌کنیم، به سمت بالا (ریشه) حرکت می‌کنیم تا به گره‌ای برسیم که فرزند سمت چپ آن، z، حاوی ۱ باشد. سپس از طریق گره‌ی z به سمت پایین حرکت می‌کنیم، و همیشه به راست‌ترین فرزندی می‌رویم که حاوی ۱ است (که متناظر است با یافتن مقدار بیشینه در زیردرخت گره‌ی z).

شکل ۲۰-۱ مسیر طی شده برای یافتن عنصر ماقبل ۱۴ را، که ۷ است، نشان می‌دهد.

همچنین اعمال INSERT و DELETE را به شکل مناسب اصلاح می‌کنیم. هنگام درج یک مقدار، یک ۱ در هر گره بر روی مسیر ساده از برگ مورد نظر به ریشه قرار می‌دهیم. هنگام حذف یک مقدار، از برگ مورد نظر به سمت ریشه حرکت کرده و بیت درون هر گره را به صورت «یا»ی منطقی دو فرزند آن، دوباره محاسبه می‌کنیم.

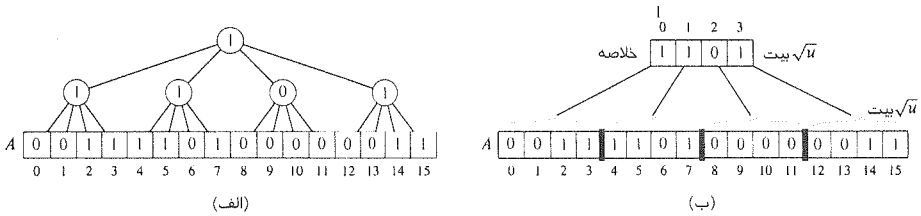
از آن جایی که ارتفاع درخت $\lg u$ است و هر یک از اعمال بالا حداکثر یک عبور به سمت بالای درخت و یک عبور به سمت پایین درخت انجام می‌دهند، هر یک از اعمال در بدترین حالت در زمان $O(\lg u)$ انجام می‌شوند.

این رویکرد چندان از استفاده از یک درخت قرمز-سیاه بهتر نیست. می‌توانیم عملیات MEMBER را در زمان $O(1)$ انجام دهیم، در حالی که جستجو در یک درخت قرمز-سیاه به زمان $O(\lg n)$ نیاز دارد. ولی از طرفی اگر تعداد عناصر ذخیره شده از اندازه‌ی مجموعه‌ی جهانی خیلی کوچک‌تر باشد، یک درخت قرمز- سیاه برای تمام اعمال دیگر سریع‌تر خواهد بود.

تحمیل یک درخت با ارتفاع ثابت

اگر یک درخت با درجه‌ی بیشتر را روی بردار بیتی سوار کنیم چه رخ می‌دهد؟ اجازه دهید فرض کنیم که اندازه‌ی مجموعه‌ی جهانی برابر است با 2^{2k}، برای یک عدد صحیح k، به طوری که $\sqrt{u}$ یک عدد صحیح باشد. به جای سوار کردن یک درخت دودویی بر روی بردار بیتی، یک درخت با درجه‌ی $\sqrt{u}$ بر روی آن سوار می‌کنیم. شکل ۲۰-۲(الف) چنین درختی را برای بردار بیتی شکل ۲۰-۱ نشان می‌دهد. ارتفاع درخت حاصل همیشه برابر با ۲ خواهد بود.

مانند قبل، هر گره‌ی داخلی حاوی مقدار «یا»ی منطقی بیت‌های زیردرخت خود است، به طوری که هر یک از $\sqrt{u}$ گره‌ی داخلی در ارتفاع ۱ نماینده، یا خلاصه‌ی یک گروه از $\sqrt{u}$ مقدار هستند. همان طور که شکل ۲۰-۲(ب) نشان می‌دهد، می‌توانیم این گره‌ها را به صورت یک آرایه‌ی $summary[0 .. \sqrt{u}-1]$ در نظر بگیریم، که در آن $summary[i]$ حاوی ۱ است اگر و فقط اگر



شکل ۲۰-۲ (الف) یک درخت با درجه‌ی $\sqrt{u}$ که بر روی بردار بیتی شکل ۲۰-۱ سوار شده است. هر گره‌ی داخلی حاوی «یا»ی منطقی بیت‌های زیردرخت خود است. (ب) دید دیگری از همان ساختار، که در آن گره‌های داخلی در عمق ۱ به صورت یک آرایه‌ی $summary[0 .. \sqrt{u}-1]$ نشان داده شده‌اند، و $summary[i]$ برابر است با «یا»ی منطقی زیرآرایه‌ی $A[i\sqrt{u} .. (i+1)\sqrt{u}-1]$.

زیرآرایه‌ی $A[i\sqrt{u} .. (i+1)\sqrt{u}-1]$ حاوی یک ۱ باشد. به این زیرآرایه‌ی $\sqrt{u}$ بیتی از A، خوشه‌ی i ام (ith cluster) می‌گوییم. برای یک مقدار داده شده‌ی x، بیت $A[x]$ در خوشه‌ی شماره‌ی $\lfloor x/\sqrt{u} \rfloor$ قرار دارد. اکنون INSERT به یک عملیات $O(1)$ تبدیل می‌شود: برای درج x، هر دوی $A[x]$ و $summary[\lfloor x/\sqrt{u} \rfloor]$ را برابر با ۱ قرار می‌دهیم. می‌توانیم از آرایه‌ی $summary$ برای انجام هر یک از اعمال MINIMUM، MAXIMUM، SUCCESSOR، PREDECESSOR، و DELETE در زمان $O(\sqrt{u})$ استفاده کنیم:

- برای یافتن مقدار کمینه (بیشینه)، چپ‌ترین (راست‌ترین) عنصر در $summary$ را که حاوی ۱ است، مثلاً $summary[i]$، می‌یابیم، و سپس در خوشه‌ی iام یک جستجوی خطی برای چپ‌ترین (راست‌ترین) ۱ انجام می‌دهیم.
- برای یافتن عنصر مابعد (ماقبل) x، ابتدا سمت راست (چپ) خوشه‌ی آن را جستجو می‌کنیم. اگر یک ۱ پیدا کردیم، آن خانه نتیجه‌ای است که به دنبال آن می‌گردیم. در غیر این صورت فرض می‌کنیم $i = \lfloor x/\sqrt{u} \rfloor$ و سمت راست (چپ) اندیس i را در آرایه‌ی $summray$ جستجو می‌کنیم. اولین خانه‌ای که حاوی ۱ باشد، مکان خوشه را به دست می‌دهد. آن خوشه را برای چپ‌ترین (راست‌ترین) ۱ جستجو می‌کنیم. آن خانه حاوی عنصر مابعد (ماقبل) است.
- برای حذف یک مقدار x، فرض می‌کنیم $i = \lfloor x/\sqrt{u} \rfloor$. $A[x]$ را برابر با ۰ قرار می‌دهیم، و همچنین $summary[i]$ را برابر با «یا»ی منطقی بیت‌های iامین خوشه.

در هر یک از اعمال بالا حداکثر درون دو خوشه‌ی $\sqrt{u}$ بیتی را جستجو می‌کنیم، به علاوه‌ی آرایه‌ی $summary$، و بنابراین هر عملیات در زمان $O(\sqrt{u})$ انجام می‌شود.

در نگاه اول به نظر می‌آید که به جای پیش‌رفت، پس‌رفت کرده‌ایم. تحمیل یک درخت دودویی به ما اعمال با زمان اجرای $O(\lg u)$ را داد، که به صورت حدی از $O(\sqrt{u})$ سریع‌تر هستند. ولی معلوم خواهد شد که استفاده از درخت‌هایی با درجه‌ی $\sqrt{u}$، ایده‌ی کلیدی درختان van Emde Boas است. این رویکرد را در بخش بعد ادامه خواهیم داد.

تمرین‌ها

۲۰-۱-۱ ساختمان‌های داده‌ی این بخش را طوری اصلاح کنید که در آن‌ها وجود کلیدهای تکراری هم مجاز باشد.

۲۰-۱-۲ ساختمان‌های داده‌ی این بخش را طوری اصلاح کنید که از کلیدهای با داده‌های پیرو پشتیبانی کنند.

۲۰-۱-۳ مشاهده کنید که با استفاده از ساختارهای توصیف شده در این بخش، روش یافتن عناصر ماقبل یا مابعد x به این که x در آن لحظه در مجموعه وجود دارد یا نه، بستگی ندارد. نشان دهید که چطور می‌توان در یک درخت جستجوی دودویی، وقتی عنصر x در درخت ذخیره نیست، عنصر مابعد آن را پیدا کرد.

۲۰-۱-۴ فرض کنید به جای تحمیل کردن یک درخت با درجه‌ی $\sqrt{u}$، از یک درخت با درجه‌ی $u^{1/k}$ استفاده می‌کردیم، که در آن $k > 1$ یک ثابت است. ارتفاع چنین درختی چقدر خواهد بود، و در این حالت هر یک از اعمال در چه زمانی اجرا می‌شوند؟

۲۰-۲ یک ساختار بازگشتی

در این بخش، ایده‌ی تحمیل یک درخت با درجه‌ی $\sqrt{u}$ بر روی یک بردار بیتی را اصلاح می‌کنیم. در بخش قبل، از یک ساختار خلاصه با اندازه‌ی $\sqrt{u}$ استفاده کردیم، که در آن هر ورودی به یک ساختار دیگر با اندازه‌ی $\sqrt{u}$ اشاره می‌کرد. اکنون، این ساختار را بازگشتی می‌کنیم، و در هر مرحله از بازگشت، اندازه‌ی مجموعه‌ی جهانی را به ریشه‌ی دوم اندازه‌ی اولیه کاهش می‌دهیم. با شروع از یک مجموعه‌ی جهانی با اندازه‌ی u، ساختارهایی می‌سازیم که $\sqrt{u} = u^{1/2}$ عنصر در خود نگه می‌دارند، که هر یک از این عناصر هم $u^{1/4}$ عنصر در خود نگه می‌دارند، که دوباره هر یک از آن‌ها $u^{1/8}$ عنصر در خود دارند، و همین طور تا اندازه‌ی پایه‌ی ۲.

برای سادگی، در این بخش فرض می‌کنیم که $u = 2^{2^k}$ برای یک عدد صحیح k، به طوری که اعداد u، $u^{1/2}$، $u^{1/4}$، ... اعداد صحیح هستند. این بازگشت، در عمل به شدت سخت‌گیر است، و در آن u فقط می‌تواند اعداد دنباله‌ی ۲، ۴، ۱۶، ۲۵۶، ۶۵۵۳۶، ... را بپذیرد. در بخش بعد خواهیم دید که چگونه این فرض را ساده‌تر کرده و فقط فرض کنیم که $u = 2^k$ برای یک عدد صحیح k. از آن جایی که ساختمان داده‌ای که در این بخش آن را بررسی می‌کنیم، فقط یک پیش‌نیاز برای ساختمان‌های van Emde Boas است، این بازگشت را فقط برای کمک به درک بهتر تحمل می‌کنیم.

با یادآوری این که هدف ما رسیدن به زمان اجرای $O(\lg\lg n)$ برای تمام اعمال است، اجازه دهید به این فکر کنیم که چطور می‌توانیم به این زمان دست یابیم. در پایان بخش ۲۰-۳ دیدیم که با زنجیره‌ای کردن متغیرها، می‌توانیم نشان دهیم که جواب رابطه‌ی بازگشتی

$$T(n) = 2T\left(\left\lfloor \sqrt{n} \right\rfloor\right) + \lg n \qquad (۱-۲۰)$$

برابر است با $T(n) = O(\lg n \lg\lg n)$. اجازه دهید یک بازگشت مشابه، ولی ساده‌تر را در نظر بگیریم:

$$T(u) = T\left(\sqrt{u}\right) + O(1) \qquad (۲-۲۰)$$

اگر از همان تکنیک استفاده کرده و متغیرها را تغییر دهیم، می‌توانیم نشان دهیم که جواب رابطه‌ی بازگشتی (۲-۲۰) برابر است با $T(u) = O(\lg\lg u)$. فرض کنید $m = \lg u$، به طوری که $u = 2^m$، و داریم

$$T(2^m) = T(2^{m/2}) + O(1)$$

اکنون با نام‌گذاری مجدد $S(m) = T(2^m)$، رابطه‌ی بازگشتی جدید زیر را داریم:

$$S(m) = S(m/2) + O(1)$$

طبق حالت ۲ از متد اصلی، جواب این رابطه‌ی بازگشتی برابر است با $S(m) = O(\lg m)$. با برگشت از $S(m)$ به $T(u)$ داریم $T(u) = T(2^m) = S(m) = O(\lg m) = O(\lg\lg u)$.

رابطه‌ی بازگشتی (۲-۲۰) به ما برای یافتن یک ساختمان داده کمک خواهد کرد. یک ساختمان داده‌ی بازگشتی طراحی خواهیم کرد که در هر مرحله از بازگشت، به اندازه‌ی یک فاکتور $\sqrt{u}$ کوچک خواهد شد. وقتی یک عملیات این ساختمان داده را بپیماید، در هر سطح به اندازه‌ی یک فاکتور ثابت زمان صرف خواهد کرد. در این صورت، رابطه‌ی (۲-۲۰) زمان اجرای این عملیات را توصیف خواهد کرد.

یک روش دیگر برای درک جواب $\lg\lg u$ به رابطه‌ی (۲-۲۰) بدین صورت است. اگر به اندازه‌ی مجموعه‌ی جهانی در هر سطح از ساختمان داده‌ی بازگشتی نگاه کنیم، دنباله‌ی u، $u^{1/2}$، $u^{1/4}$، $u^{1/8}$، ... را می‌بینیم. اگر تعداد بیت‌های مورد نیاز برای ذخیره‌ی اندازه‌ی مجموعه‌ی جهانی برای هر سطح را بررسی کنیم، می‌بینیم که $\lg u$ بیت در سطح بالا نیاز داریم، و تعداد بیت‌های مورد نیاز برای هر یک از سطوح دیگر برابر است با نصف سطح قبلی. به طور کلی با b بیت آغاز کرده و در هر سطح، تعداد بیت‌ها را نصف می‌کنیم، و بعد از $\lg b$ سطح به پایین‌ترین سطح می‌رسیم که فقط یک بیت نیاز دارد. از آن جایی که $b = \lg u$، می‌بینیم که بعد از $\lg\lg u$ سطح، اندازه‌ی مجموعه‌ی جهانی ۲ خواهد بود.

با بازگشت به ساختمان داده‌ی شکل ۲۰-۲، می‌بینیم که هر مقدار x در خوشه‌ی شماره $\lfloor x/\sqrt{u} \rfloor$ قرار می‌گیرد. اگر x را به صورت یک عدد صحیح دودویی بیتی در نظر بگیریم، شماره‌ی خوشه، $\lfloor x/\sqrt{u} \rfloor$، توسط $(\lg u)/2$ بیت پرارزش x تعیین می‌شود. x در خوشه‌ی خود در مکان $x \bmod \sqrt{u}$ قرار می‌گیرد، که توسط $(\lg u)/2$ بیت کم‌ارزش x مشخص می‌شود. از آن جایی که باید اندیس‌گذاری خود را با عبارات داده شده در بالا انجام دهیم، اجازه دهید برای ساده‌تر شده این کار چند تابع تعریف کنیم:

$$\begin{aligned} \text{high}(x) &= \lfloor x/\sqrt{u} \rfloor, \\ \text{low}(x) &= x \bmod \sqrt{u}, \\ \text{index}(x, y) &= x\sqrt{u} + y. \end{aligned}$$

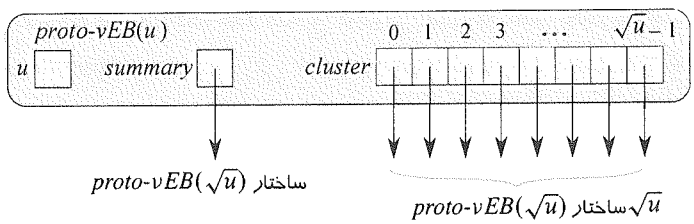
تابع $high(x)$ به ما $\lg u/2$ بیت پرارزش x را می‌دهد، که نشان‌دهنده‌ی شماره‌ی خوشه‌ی x است. تابع $low(x)$، $\lg u/2$ بیت کم‌ارزش x را به دست می‌دهد، که نشان‌دهنده‌ی مکان x در خوشه‌ی خود است. در نهایت تابع $index(x, y)$ یک شماره‌ی عنصر از x و y می‌سازد، که در آن x عبارت است از $\lg u/2$ بیت پرارزش شماره‌ی عنصر، و y عبارت است از $\lg u/2$ بیت کم‌ارزش شماره‌ی عنصر. همچنین رابطه‌ی همانی $x = index(high(x), low(x))$ را داریم. مقدار u که توسط هر یک از این توابع مورد استفاده قرار می‌گیرد، همیشه اندازه‌ی مجموعه‌ی جهانی ساختمان داده‌ای خواهد بود که توابع روی آن فراخوانی می‌شوند، که هم‌زمان با پایین رفتن روی سطوح، تغییر می‌کند.

۲۰-۲-۱ ساختارهای van Emde Boas بدوی

اجازه دهید با استفاده از درکی که از رابطه‌ی (۲۰-۲) به دست آوردیم، یک ساختمان داده برای پشتیبانی از اعمال مورد نظر طراحی کنیم. با این که این ساختمان داده به هدف زمان $O(\lg\lg u)$ برای بعضی از اعمال نمی‌رسد، ولی پایه‌ای خواهد بود برای درختان van Emde Boas که در بخش ۲۰-۳ آن‌ها را خواهیم دید.

برای مجموعه‌ی جهانی $\{0, 1, 2, ..., u-1\}$، یک *ساختار van Emde Boas بدوی*، یا *ساختار vEB بدوی* به صورت بازگشتی به شکل زیر تعریف می‌کنیم، و آن را با $proto\text{-}vEB(u)$ نشان می‌دهیم. هر ساختار $proto\text{-}vEB(u)$ حاوی یک خصیصه‌ی u است، که اندازه‌ی مجموعه‌ی جهانی را به دست می‌دهد. این ساختارها همچنین حاوی خصیصه‌های زیر هستند:

- اگر $u = 2$ آن گاه در سطح اولیه هستیم، و ساختار حاوی یک آرایه‌ی $A[0 .. 1]$ از دو بیت است.
- در غیر این صورت $u = 2^{2^k}$ برای یک عدد صحیح $k \geq 1$، و بنابراین $u \geq 4$. علاوه بر اندازه‌ی مجموعه‌ی جهانی ساختار داده‌ی $proto\text{-}vEB(u)$ حاوی خصیصه‌های زیر است، که در شکل ۲۰-۳ مشخص شده‌اند:
  - یک اشاره‌گر با نام *summary* به یک ساختار $proto\text{-}vEB(\sqrt{u})$، و



شکل ۲۰-۳ اطلاعات درون یک ساختار $proto\text{-}vEB(\sqrt{u})$ وقتی که $u \geq 4$. این ساختار حاوی اندازه‌ی مجموعه‌ی جهانی u، یک اشاره‌گر *summary* به یک ساختار $proto\text{-}vEB(\sqrt{u})$، و یک آرایه‌ی $cluster[0 .. \sqrt{u} - 1]$ از $\sqrt{u}$ اشاره‌گر به ساختارهای $proto\text{-}vEB(\sqrt{u})$ است.

- یک آرایه‌ی $cluster[0 .. \sqrt{u}-1]$ از $\sqrt{u}$ اشاره‌گر، هر کدام به یک ساختار $proto\text{-}vEB(\sqrt{u})$.

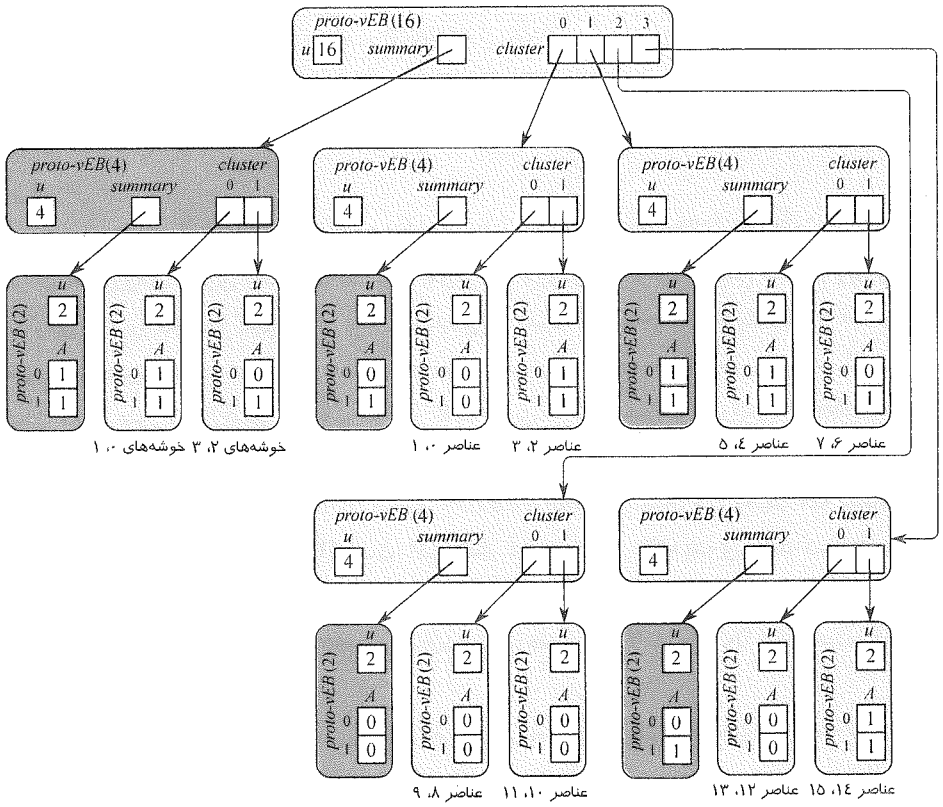
عنصر x، که $0 \le x < u$، به صورت بازگشتی در عنصر شماره $\text{low}(x)$ خوشه‌ی شماره $\text{high}(x)$ ذخیره می‌شود.

در ساختار دو سطحی بخش قبل، هر گره یک آرایه‌ی *summary* با اندازه‌ی $\sqrt{u}$ ذخیره می‌کند، که در آن هر ورودی حاوی یک بیت است. از اندیس هر ورودی می‌توانیم اندیس شروع زیرآرایه‌ای با اندازه‌ی $\sqrt{u}$ را محاسبه کنیم که آن بیت نماینده‌ی آن است. در ساختار vEB بدوی، از اشاره‌گرهای صریح به جای محاسبات اندیسی استفاده خواهیم کرد. آرایه‌ی *summary* حاوی بیت‌های نماینده‌ای است که به صورت بازگشتی در یک ساختار vEB بدوی ذخیره شده‌اند، و آرایه‌ی *cluster* حاوی $\sqrt{u}$ اشاره‌گر است.

شکل ۲۰-۴ یک ساختار کاملاً باز شده‌ی $proto\text{-}vEB(16)$ را نشان می‌دهد، که نشان‌دهنده‌ی مجموعه‌ی $\{2, 3, 4, 5, 7, 14, 15\}$ است. اگر مقدار i در ساختار vEB بدوی باشد که *summary* به آن اشاره می‌کند، آن گاه خوشه‌ی iام حاوی یکی از مقادیری است که در مجموعه‌ی اصلی وجود دارد. مانند درختان با ارتفاع ثابت، $cluster[i]$ نشان‌دهنده‌ی مقدار $i\sqrt{u}$ تا $(i+1)\sqrt{u}-1$ است، که خوشه‌ی iام را تشکیل می‌دهند.

در سطح پایه، عناصر مجموعه‌ی پویای اصلی در بعضی از ساختارهای $proto\text{-}vEB(2)$ ذخیره شده‌اند، و بقیه‌ی ساختارهای $proto\text{-}vEB(2)$ حاوی بیت‌های نماینده هستند. در شکل، بیت‌هایی که هر یک از ساختارهای پایه ذخیره می‌کند (غیر از ساختارهای نماینده) زیر آن ساختار نشان داده شده است. برای مثال ساختار $proto\text{-}vEB(2)$ که با «عناصر ۶، ۷» مشخص شده است، حاوی بیت ۶ (با مقدار ۰، چرا که عنصر ۶ در مجموعه نیست) در $A[0]$، و بیت ۷ (با مقدار ۱، چرا که عنصر ۷ در مجموعه وجود دارد) در $A[1]$ است.

مانند خوشه‌ها، هر نماینده به سادگی یک مجموعه‌ی پویا با مجموعه‌ی جهانی با اندازه‌ی $\sqrt{u}$ است، و بنابراین هر نماینده را به صورت یک $proto\text{-}vEB(\sqrt{u})$ نشان می‌دهیم. چهار بیت نماینده برای ساختار $proto\text{-}vEB(16)$ اصلی در چپ‌ترین ساختار $proto\text{-}vEB(4)$ ذخیره شده‌اند، که در نهایت در دو ساختار $proto\text{-}vEB(2)$ قرار می‌گیرند. برای مثال در ساختار $proto\text{-}vEB(2)$ با برچسب «خوشه‌های ۲، ۳» داریم $A[0]=0$، که نشان می‌دهد که خوشه‌ی شماره‌ی ۲ از ساختار $proto\text{-}vEB(16)$ (حاوی عناصر ۸، ۹، ۱۰، ۱۱) تماماً ۰ است، و $A[1]=1$، که به ما می‌گوید که خوشه‌ی شماره‌ی ۳ (حاوی عناصر ۱۲، ۱۳، ۱۴، ۱۵) حداقل یک ۱ دارد. هر ساختار $proto\text{-}vEB(4)$ به نماینده‌ی خود اشاره می‌کند، که خود در یک ساختار $proto\text{-}vEB(2)$ ذخیره شده است. برای مثال به ساختار $proto\text{-}vEB(2)$ در سمت چپ ساختار با برچسب «عناصر ۰، ۱» نگاه کنید. چون در این ساختار، $A[0]=0$، پس تمام ساختار «عناصر ۰، ۱» برابر ۰ است، و چون $A[1]=1$، خواهیم دانست که ساختار «عناصر ۲، ۳» حاوی حداقل یک ۱ است.



شکل ۲۰-۴ یک ساختار *proto-vEB*(۱۶) نشان دهنده‌ی مجموعه‌ی {۲, ۳, ۴, ۵, ۷, ۱۴, ۱۵}. این ساختار از طریق *cluster*[۳ .. ۰] به چهار ساختار *proto-vEB*(۴) دیگر اشاره می‌کند، و همچنین به یک ساختار نماینده (summary)، که باز هم یک *proto-vEB*(۴) است. هر ساختار *proto-vEB*(۴) از طریق *cluster*[۱ .. ۰] به دو ساختار *proto-vEB*(۲) دیگر اشاره می‌کند، و همچنین به یک نماینده، که باز هم یک *proto-vEB*(۲) است. هر ساختار *proto-vEB*(۲) فقط حاوی یک آرایه‌ی *A*[۱ .. ۰] از دو بیت است. ساختارهای *proto-vEB*(۲) بالای «عناصر *i*، *j*»، بیت‌های *i* و *j* مجموعه‌ی پویای اصلی را ذخیره می‌کنند، و ساختارهای *proto-vEB*(۲) بالای «خوشه‌های *i*، *j*»، بیت‌های نماینده‌ی خوشه‌های *i* و *j* در ساختار *proto-vEB*(۱۶) در بالاترین سطح را ذخیره می‌کنند. برای وضوح، سایه‌ی تیره نشان‌دهنده‌ی سطح بالای یک ساختار vEB بدوی است که اطلاعات خلاصه را برای ساختار پدر خود ذخیره می‌کند؛ چنین ساختاری هیچ تفاوت دیگری با ساختارهای vEB بدوی دیگر با اندازه‌ی مشابه ندارد.

۲۰-۲-۲ اعمال بر روی یک ساختار van Emde Boas بدوی

اکنون توضیح می‌دهیم که چطور می‌توان اعمال مورد نیاز را بر روی یک ساختار vEB بدوی پیاده‌سازی کرد. ابتدا اعمال پرسشی – MEMBER، MINIMUM، و SUCCESSOR – را بررسی خواهیم کرد، که ساختار vEB را تغییر نمی‌دهند. سپس در مورد INSERT و DELETE بحث می‌کنیم. اعمال

MAXIMUM و PREDECESSOR را که به ترتیب مشابه MINIMUM و SUCCESSOR هستند، به عنوان تمرین ۲۰-۲-۱ واگذار می‌کنیم.

هر یک از اعمال MEMBER، SUCCESSOR، PREDECESSOR، INSERT و DELETE یک پارامتر x را دریافت می‌کنند، به همراه یک ساختار vEB بدوی V. تمام این اعمال فرض می‌کنند که $0 \le x < V.u$.

تعیین وجود یک مقدار در مجموعه

برای انجام MEMBER(x)، باید یک بیت متناظر با x در ساختار $proto\text{-}vEB(2)$ مناسب پیدا کنیم. با صرف نظر از تمام ساختارهای *summary*، می‌توانیم این کار را در زمان $O(\lg\lg u)$ انجام دهیم. رویه‌ی زیر یک ساختار vEB بدوی V و یک مقدار x را دریافت می‌کند، و یک بیت بازمی‌گرداند که نشان می‌دهد آیا مقدار x در مجموعه‌ی پویای متناظر با V موجود است یا نه.

```
PROTO-vEB-MEMBER(V, x)
1  if V.u == 2
2      return V.A[x]
3  else return PROTO-vEB-MEMBER(V.cluster[high(x)], low(x))
```

رویه‌ی PROTO-vEB-MEMBER به صورت زیر کار می‌کند. خط ۱ تست می‌کند که آیا در حالت پایه هستیم یا نه، که در آن V یک ساختار $proto\text{-}vEB(2)$ است. خط ۲ عملیات مناسب را برای حالت پایه انجام می‌دهد، که فقط عبارت است از بازگرداندن بیت مناسب آرایه‌ی A. خط ۳ به حالت بازگشتی مربوط می‌شود، که در آن متناسباً در ساختار vEB بدوی کوچک‌تری «فرو می‌رویم». مقدار high(x) می‌گوید کدام ساختار $proto\text{-}vEB(\sqrt{u})$ را ملاقات کرده‌ایم، و low(x) تعیین می‌کند که هم اکنون بر روی کدام یک از عناصر ساختار $proto\text{-}vEB(\sqrt{u})$ قرار داریم.

اجازه دهید ببینیم وقتی PROTO-vEB-MEMBER$(V, 6)$ را روی ساختار $proto\text{-}vEB(16)$ شکل ۲۰-۴ فراخوانی می‌کنیم چه رخ می‌دهد. چون high$(6) = 1$ وقتی $u = 16$، به درون ساختار $proto\text{-}vEB(4)$ در گوشه‌ی بالا و راست بازگشت می‌کنیم، و عنصر low$(6) = 2$ را در آن ساختار بررسی می‌کنیم. در این فراخوانی بازگشتی، $u = 4$، و بنابراین دوباره عمل بازگشت انجام خواهد شد. با $u = 4$، داریم high$(2) = 1$ و low$(2) = 0$، و بنابراین عنصر 0 در ساختار $proto\text{-}vEB(2)$ در بالا و راست را بررسی می‌کنیم. می‌بینیم که در این فراخوانی بازگشتی به حالت پایه رسیده‌ایم، و بنابراین مقدار $A[0] = 0$ در طول زنجیره‌ی فراخوانی بازگشتی به بالا بازگردانده می‌شود. پس فراخوانی PROTO-vEB-MEMBER$(V, 6)$ مقدار 0 را بازخواهد گرداند، که بدین معنی است که مقدار 6 در مجموعه وجود ندارد.

برای تعیین زمان اجرای PROTO-vEB-MEMBER، فرض کنید $T(u)$ نشان‌دهنده‌ی زمان اجرا بر روی یک ساختار $proto\text{-}vEB(u)$ باشد. هر فراخوانی بازگشتی به زمان ثابت نیاز دارد، البته غیر از زمان مورد نیاز برای فراخوانی‌های بازگشتی دیگری که از طریق آن انجام می‌شود. وقتی PROTO-vEB-

MEMBER یک فراخوانی بازگشتی انجام می‌دهد، این فراخوانی را بر روی یک ساختار $proto\text{-}vEB(\sqrt{u})$ انجام می‌دهد. بنابراین می‌توانیم زمان اجرا را به وسیله‌ی رابطه‌ی بازگشتی $T(u) = T(\sqrt{u}) + O(1)$ توصیف کنیم، که قبلاً آن را با عنوان رابطه‌ی (۲۰-۲) دیده‌ایم. جواب این رابطه برابر است با $T(u) = O(\lg\lg u)$، و بنابراین نتیجه می‌گیریم که PROTO-vEB-MEMBER در زمان $O(\lg\lg u)$ اجرا می‌شود.

یافتن عنصر کمینه

اکنون نحوه‌ی پیاده‌سازی عملیات MIMINUM را بررسی می‌کنیم. اگر مجموعه‌ی پویای مورد نظر تهی نباشد، رویه‌ی PROTO-vEB-MINIMUM(V) عنصر کمینه را در ساختار vEB بدوی V بازمی‌گرداند، و اگر مجموعه تهی باشد، مقدار NIL بازگردانده خواهد شد.

```
PROTO-vEB-MINIMUM(V)
1   if V.u == 2
2       if V.A[0] == 1
3           return 0
4       elseif V.A[1] == 1
5           return 1
6       else return NIL
7   else min-cluster = PROTO-vEB-MINIMUM(V.summary)
8       if min-cluster == NIL
9           return NIL
10      else offset = PROTO-vEB-MINIMUM(V.cluster[min-cluster])
11          return index(min-cluster, offset)
```

این رویه بدین صورت کار می‌کند. خط ۱ وجود حالت پایه را چک می‌کند، که در خطوط ۲-۶ به صورت سرراست اداره می‌شود. خطوط ۷-۱۱ مدیریت حالت بازگشتی را بر عهده دارند. ابتدا خط ۷ شماره‌ی اولین خوشه‌ای را که حاوی عنصری از مجموعه است، پیدا می‌کند. این کار با فراخوانی بازگشتی PROTO-vEB-MINIMUM(V) بر روی $V.summary$ انجام می‌شود، که خود یک ساختار $proto\text{-}vEB(\sqrt{u})$ است. خط ۷ شماره‌ی خوشه را به متغیر $min\text{-}cluster$ نسبت می‌دهد. اگر مجموعه تهی باشد، در این صورت فراخوانی بازگشتی NIL را بازگردانده است، و خط ۹، NIL را بازمی‌گرداند. در غیر این صورت عنصر کمینه‌ی مجموعه جایی در خوشه‌ی شماره $min\text{-}cluster$ است. فراخوانی بازگشتی در خط ۱۰ شماره‌ی عنصر کمینه در خوشه را می‌یابد. نهایتاً خط ۱۱ مقدار عنصر کمینه را از شماره‌ی خوشه و اندیس عنصر کمینه‌ی درون آن می‌سازد، و این مقدار را بازمی‌گرداند.

با این که بررسی اطلاعات خلاصه به ما اجازه می‌دهد که به سرعت خوشه‌ی حاوی عنصر کمینه را بیابیم، ولی چون این رویه دو فراخوانی بزاگشتی بر روی ساختار $proto\text{-}vEB(\sqrt{u})$ انجام می‌دهد،

در بدترین حالت در زمان $O(\lg\lg u)$ اجرا نمی‌شود. با فرض این که $T(u)$ نشان دهنده‌ی زمان اجرای بدترین حالت برای PROTO-vEB-MINIMUM بر روی یک ساختار $proto\text{-}vEB(u)$ باشد، رابطه‌ی بازگشتی زیر را داریم:

$$T(u) = 2T(\sqrt{u}) + O(1) \qquad (20\text{-}3)$$

دوباره، از یک تغییر متغیر برای حل رابطه استفاده می‌کنیم، که با قرار دادن $m = \lg u$ به دست می‌دهد

$$T(2^m) = 2T(2^{m/2}) + O(1)$$

نام‌گذاری مجدد $S(m) = T(2^m)$ نتیجه می‌دهد

$$S(m) = 2S(m/2) + O(1)$$

که طبق حالت ۱ از متد اصلی، دارای جواب $S(m) = \theta(m)$ است. با بازگشت از $S(m)$ به $T(u)$ داریم $T(u) = T(2^m) = S(m) = \theta(m) = \theta(\lg u)$. بنابراین می‌بینیم که به خاطر دومین فراخوانی بازگشتی، رویه‌ی PROTO-vEB-MINIMUM، به جای $O(\lg\lg u)$، در زمان $O(\lg u)$ اجرا می‌شود.

یافتن عنصر مابعد

عملیات SUCCESSOR حتی از MINIMUM هم بدتر است. در بدترین حالت این عملیات دو فراخوانی بازگشتی انجام می‌دهد، به همراه یک فراخوانی PROTO-vEB-MINIMUM. رویه‌ی PROTO-vEB-SUCCESSRO (V, x) کوچک‌ترین عنصری را در ساختار vEB بدوی V بازمی‌گرداند که بزرگ‌تر از x است، و NIL اگر هیچ عنصری در V از x بزرگ‌تر نباشد. این رویه نیازی ندارد که x عنصری از مجموعه باشد، ولی فرض می‌کند که $V.u > x \geq 0$.

```
PROTO-vEB-SUCCESSOR(V, x)
1   if V.u == 2
2       if x == 0 and V.A[1] == 1
3           return 1
4       else return NIL
5   else offset = PROTO-vEB-SUCCESSOR(V.cluster[high(x)], low(x))
6       if offset ≠ NIL
7           return index(high(x), offset)
8       else succ-cluster = PROTO-vEB-SUCCESSOR(V.summary, high(x))
9           if succ-cluster == NIL
10              return NIL
11          else offset = PROTO-vEB-MINIMUM(V.cluster[succ-cluster])
12              return index(succ-cluster, offset)
```

رویه‌ی PROTO-vEB-SUCCESSOR به صورت زیر کار می‌کند. مانند قبل خط ۱ وجود حالت پایه را بررسی می‌کند، که در خطوط ۲–۴ به صورت سرراست اداره می‌شوند: تنها حالتی که در آن x می‌تواند در ساختار $proto\text{-}vEB(2)$ یک عنصر مابعد داشته باشد، این است که داشته باشیم $x = 0$ و

$A[1]$ برابر ۱ باشد. خطوط ۵-۱۲ حالت بازگشتی را اداره می‌کنند. خط ۵ در خوشه‌ی x به دنبال یک عنصر مابعد برای x می‌گردد، و نتیجه را به متغیر *offset* نسبت می‌دهد. خط ۶ تعیین می‌کند که آیا x در خوشه‌ی خود یک عنصر مابعد دارد یا نه؛ اگر داشت، خط ۷ آن را محاسبه کرده و مقدار عنصر مابعد را بازمی‌گرداند. در غیر این صورت باید در خوشه‌های دیگر جستجو را ادامه دهیم. خط ۸ شماره‌ی خوشه‌ی ناتهی بعدی را به *succ-cluster* نسبت می‌دهد، که برای یافتن آن از اطلاعات خلاصه استفاده شده است. خط ۹ چک می‌کند که *succ-cluster* برابر با NIL است یا خیر، و اگر تمام خوشه‌های دیگر تهی باشند، خط ۱۰ مقدار NIL را بازمی‌گرداند. اگر *succ-cluster* غیر NIL باشد، خط ۱۱ اولین عنصر درون خوشه را به *offset* نسبت می‌دهد، و خط ۱۲ عنصر کمینه در آن خوشه را محاسبه کرده و بازمی‌گرداند.

در بدترین حالت، PROTO-vEB-SUCCESSOR دو بار خود را به صورت بازگشتی فراخوانی بر روی ساختارهای $proto\text{-}vEB(\sqrt{u})$ فراخوانی می‌کند، و یک فراخوانی از PROTO-vEB-MINIMUM بر روی یک ساختار $proto\text{-}vEB(\sqrt{u})$ انجام می‌دهد. بنابراین رابطه‌ی بازگشتی برای زمان اجرای بدترین حالت PROTO-vEB-SUCCESSOR عبارت است از

$$\begin{aligned} T(u) &= 2T(\sqrt{u}) + \theta(\lg\sqrt{u}) \\ &= 2T(\sqrt{u}) + \theta(\lg u) \end{aligned}$$

می‌توانیم از تکنیکی مشابه چیزی که برای رابطه‌ی (۲۰-۱) استفاده کردیم، بهره برده و نشان دهیم که جواب این رابطه‌ی بازگشتی عبارت است از $T(u) = \theta(\lg u \lg\lg u)$. بنابراین رویه‌ی PROTO-vEB-SUCCESSOR به صورت حدی از PROTO-vEB-MINIMUM کندتر است.

درج یک عنصر

برای درج یک عنصر، باید آن را در خوشه‌ی مناسب درج کنیم، و همچنین بیت نماینده‌ی آن خوشه را برابر با ۱ قرار دهیم. رویه‌ی PROTO-vEB-INSERT(V, x) مقدار x را در ساختار vEB بدوی V درج می‌کند.

```
PROTO-vEB-INSERT(V, x)
1  if V.u == 2
2      V.A[x] = 1
3  else PROTO-vEB-INSERT(V.cluster[high(x)], low(x))
4      PROTO-vEB-INSERT(V.summary, high(x))
```

در حالت پایه، خط ۲ بیت مناسب در آرایه‌ی A را برابر با ۱ قرار می‌دهد. در حالت بازگشتی، فراخوانی بازگشتی در خط ۳ مقدار x را در خوشه‌ی مناسب درج می‌کند، و خط ۴ بیت نماینده‌ی آن خوشه را با ۱ مقداردهی می‌کند.

چون PROTO-vEB-INSERT در بدترین حالت دو فراخوانی بازگشتی انجام می‌دهد، رابطه‌ی بازگشتی (۲۰-۳) زمان اجرای آن را توصیف می‌کند. بنابراین PROTO-vEB-INSERT در زمان $\theta(\lg u)$ اجرا می‌شود.

حذف یک عنصر

عملیات DELETE از درج پیچیده‌تر است. در حالی که هنگام درج همیشه می‌توانیم بیت نماینده را برابر با ۱ قرار دهیم، هنگام حذف همیشه نمی‌توانیم بیت نماینده را ۰ کنیم. باید تعیین کنیم که آیا یک بیت ۱ در خوشه وجود دارد یا نه. طبق تعریف ساختارهای vEB بدوی، باید تمام $\sqrt{u}$ بیت درون خوشه را بررسی کنیم تا ببینیم که آیا هیچ یک از آن‌ها ۱ هستند یا نه. همچنین می‌توانیم یک خصیصه‌ی n به ساختار vEB بدوی اضافه کنیم، که تعداد عناصر آن خوشه را می‌شمارد. پیاده‌سازی PROTO-vEB-DELETE را به عنوان تمرین‌های ۲۰-۲-۲ و ۲۰-۲-۳ واگذار می‌کنیم.

تمرین‌ها

۲۰.۲-۱ شبه‌کدهایی برای رویه‌های PROTO-vEB-MAXIMUM و PROTO-vEB-PREDECESSOR بنویسید.

۲۰.۲-۲ شبه‌کدی برای PROTO-vEB-DELETE بنویسید. در این شبه‌کد باید بیت نماینده‌ی مناسب با استفاده از اطلاعات پویش تمام بیت‌های مربوطه در خوشه، متناسباً به هنگام‌سازی شود. زمان اجرای رویه‌ی شما در بدترین حالت چقدر است؟

۲۰.۲-۳ خصیصه‌ی n را به هر یک از ساختارهای vEB بدوی اضافه کنید، که مشخص کننده‌ی تعداد عناصری از مجموعه است که آن ساختار، نماینده‌ی آن‌ها است. سپس شبه‌کدی برای PROTO-vEB-DELETE بنویسید که از خصیصه‌ی n برای تصمیم‌گیری در مورد ۰ کردن بیت خلاصه استفاده می‌کند. زمان اجرای رویه‌ی شما در بدترین حالت چقدر است؟ چه رویه‌های دیگری به خاطر این خصیصه‌ی جدید باید تغییر کنند؟ آیا این تغییرها بر روی زمان اجرای این رویه‌ها تأثیری می‌گذارند؟

۲۰.۲-۴ ساختار vEB بدوی را طوری تغییر دهید که از مقادیر تکراری برای کلیدها پشتیبانی کند.

۲۰.۲-۵ ساختار vEB بدوی را طوری تغییر دهید که بتواند به همراه کلیدها، داده‌های پیروی متناظر را هم ذخیره کند.

۲۰.۲-۶ شبه‌کدی برای رویه‌ای بنویسید که یک ساختار $proto\text{-}vEB(u)$ می‌سازد.

۲۰.۲-۷ بحث کنید که اگر خط ۹ رویه‌ی PROTO-vEB-MINIMUM اجرا شود، آن گاه ساختار vEB بدوی، تهی است.

۲۰.۲-۸ فرض کنید یک ساختار vEB بدوی طراحی کرده‌ایم، که در آن آرایه‌ی *cluster* فقط $u^{1/4}$ عنصر در خود نگه می‌دارد. در این صورت زمان اجرای هر یک از اعمال چگونه خواهد بود؟

۲۰-۳ درخت van Emde Boas

ساختار vEB بدوی توصیف شده در بخش قبل، نزدیک چیزی است که برای رسیدن به زمان اجرای $O(\lg\lg u)$ نیاز داریم. کم‌بود آن بدین خاطر است که در اکثر رویه‌ها باید بیش از حد مجاز بازگشت انجام دهیم. در این بخش ساختمان داده‌ای طراحی می‌کنیم مشابه ساختار vEB بدوی، ولی اطلاعاتی که ذخیره می‌کند کمی بیشتر است، و بنابراین می‌تواند نیاز به بعضی از بازگشت‌ها را برطرف کند.

همان طور که در بخش ۲۰-۲ مشاهده کردیم، فرض این که در مورد اندازه‌ی مجموعه‌ی جهانی داشتیم – این که $u = 2^{2^k}$ برای یک عدد صحیح k – اضافی و به شدت محدود کننده است، و مقادیر ممکن u را به عناصر یک مجموعه‌ی به شدت خلوت منحصر می‌کند. بنابراین از این به بعد اجازه می‌دهیم که اندازه‌ی مجموعه‌ی جهانی، u، هر توانی از ۲ باشد، و وقتی $\sqrt{u}$ یک عدد صحیح نیست – یعنی وقتی u یک توان فرد از ۲ است ($u = 2^{2^{k+1}}$ برای یک عدد صحیح $k \geq 0$) – آن گاه $\lg u$ بیت یک عدد را به دو دسته‌ی $\lceil (\lg u)/2 \rceil$ بیت پرارزش و $\lfloor (\lg u)/2 \rfloor$ بیت کم‌ارزش تقسیم می‌کنیم. برای سادگی، $2^{\lceil (\lg u)/2 \rceil}$ («ریشه‌ی دوم بالایی» u) را به صورت $\uparrow\sqrt{u}$، و $2^{\lfloor (\lg u)/2 \rfloor}$ («ریشه‌ی دوم پایینی» u) را به صورت $\downarrow\sqrt{u}$ نشان می‌دهیم، به طوری که داریم $u = \uparrow\sqrt{u} \cdot \downarrow\sqrt{u}$، و وقتی u توان زوجی از ۲ است ($u = 2^{2^k}$ برای یک عدد صحیح k) داریم $\uparrow\sqrt{u} = \downarrow\sqrt{u} = \sqrt{u}$. چون اکنون اجازه می‌دهیم u توان فردی از ۲ باشد، باید توابع کمکی خود از بخش ۲۰-۲ را بازتعریف کنیم:

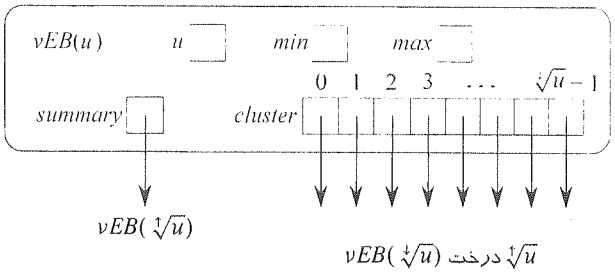
$$\begin{aligned} \text{high}(x) &= \left\lfloor x / \downarrow\sqrt{u} \right\rfloor, \\ \text{low}(x) &= x \mod \downarrow\sqrt{u}, \\ \text{index}(x, y) &= x \downarrow\sqrt{u} + y. \end{aligned}$$

۲۰-۳-۱ درختان van Emde Boas

درخت van Emde Boas یا *درخت vEB*، نسخه‌ی اطلاح شده‌ی ساختار vEB بدوی است. یک درخت vEB با مجموعه‌ی جهانی با اندازه‌ی u را به صورت $vEB(u)$ نشان می‌دهیم، و مگر این که u برابر با حالت پایه (دو) باشد، خصیصه‌ی *summary* به یک درخت $vEB(\uparrow\sqrt{u})$ اشاره می‌کند، و آرایه‌ی $cluster[0 .. \uparrow\sqrt{u} - 1]$ به $\uparrow\sqrt{u}$ درخت $vEB(\downarrow\sqrt{u})$. همان طور که شکل ۲۰-۵ نشان می‌دهد، یک درخت vEB حاوی دو خصیصه است که ساختارهای vEB بدوی فاقد آن‌ها بودند:

- *min* عنصر کمینه را در درخت vEB ذخیره می‌کند، و
- *max* عنصر بیشینه را در درخت vEB ذخیره می‌کند.

به علاوه عنصر ذخیره شده در *min* در هیچ یک از درختان بازگشتی $vEB(\downarrow\sqrt{u})$ که آرایه‌ی *cluster* به آن‌ها اشاره می‌کند، وجود نخواهد داشت. بنابراین عناصر ذخیره شده در یک درخت $vEB(u)$ با نام V عبارت خواهند بود از $V.min$ به علاوه‌ی تمام عناصری که به صورت بازگشتی در درخت $vEB(\downarrow\sqrt{u})$



شکل ۲۰-۵ اطلاعات ذخیره شده در یک درخت $vEB(u)$ که در آن $u > 2$. ساختار، حاوی u (اندازه‌ی مجموعه‌ی جهانی)، عناصر min و max، یک اشاره‌گر $summary$ به یک درخت $vEB(\sqrt[\uparrow]{u})$، و یک آرایه‌ی $cluster[0 .. \sqrt[\uparrow]{u}]$ از $\sqrt[\uparrow]{u}$ اشاره‌گر به درختان $vEB(\sqrt[\downarrow]{u})$ است.

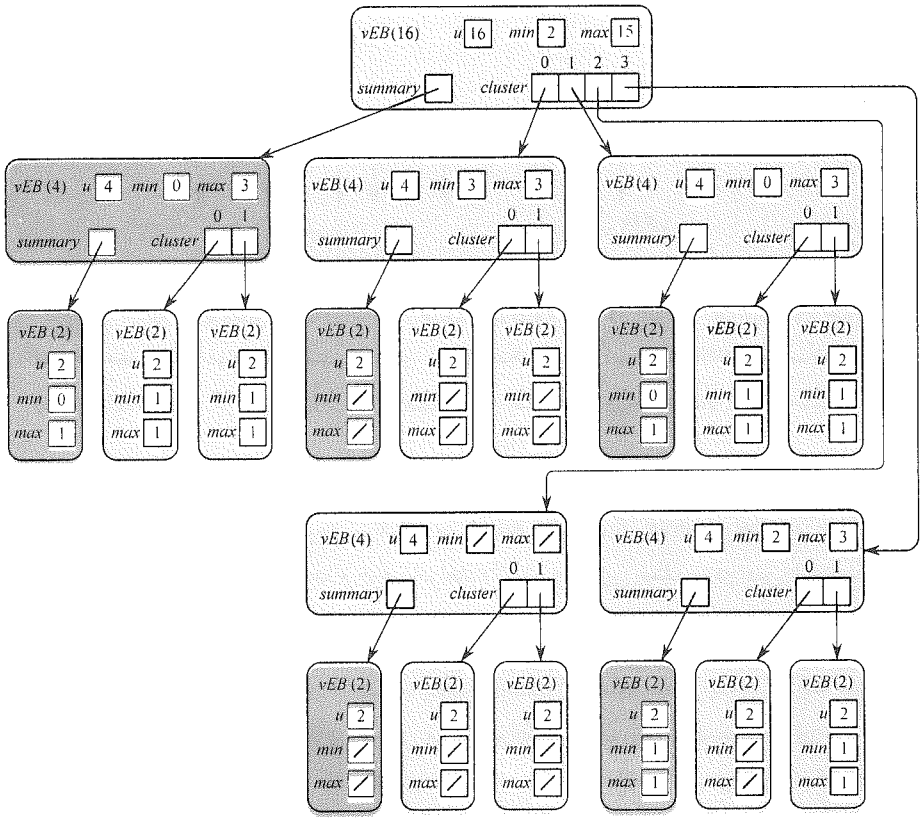
ذخیره شده‌اند و $V.cluster[0 .. \sqrt[\uparrow]{u} - 1]$ به آن‌ها اشاره می‌کند. توجه کنید که وقتی یک درخت vEB حاوی دو عنصر یا بیشتر است، نحوه‌ی برخورد ما با min متفاوت از max خواهد بود: عنصر ذخیره شده در min در هیچ یک از خوشه‌ها وجود نخواهد داشت، ولی این مطلب در مورد عنصر ذخیره شده در max صادق نیست.

از آن جایی که اندازه‌ی پایه ۲ است، یک درخت $vEB(2)$ به آرایه‌ی A که ساختار آن $proto\text{-}vEB(2)$ است، نیازی ندارد. در عوض می‌توانیم عناصر آن را از خصیصه‌های min و max آن تعیین کنیم. در یک درخت vEB که عنصری ندارد، مستقل از اندازه‌ی مجموعه‌ی جهانی آن، هر دو خصیصه‌ی min و max برابر با NIL هستند.

شکل ۲۰-۶ یک درخت $vEB(16)$ را نشان می‌دهد که حاوی مجموعه‌ی $\{2, 3, 4, 5, 7, 14, 15\}$ است. چون کوچک‌ترین عنصر ۲ است، $V.min$ برابر است با ۲، و با این که $\text{high}(2) = 0$، عنصر ۲ در درخت $vEB(2)$ که $V.cluster[0]$ به آن اشاره می‌کند، قرار ندارد: توجه کنید که $V.cluster[0].min$ برابر است با ۳، و بنابراین ۲ در این درخت vEB نیست. به طور مشابه، از آن جایی که $V.cluster[0].min$ برابر ۳ است و ۲ و ۳ تنها عناصر $V.cluster[0]$ هستند، خوشه‌های $vEB(2)$ درون $V.cluster[0]$ تهی هستند.

مشخص خواهد شد که خصیصه‌های min و max کلید کاهش تعداد فراخوانی‌های بازگشتی در اعمال درختان vEB هستند. این خصیصه‌ها به چهار طریق به ما کمک می‌کنند:

- اعمال MINIMUM و MAXIMUM اصلاً نیازی به بازگشت ندارند، چرا که به سادگی می‌توانند مقادیر min و max را بازگردانند.
- عملیات SUCCESSOR می‌تواند از فراخوانی بازگشتی برای تعیین وجود عنصر مابعد x در $\text{high}(x)$ پرهیز کند. این بدین خاطر است که عنصر مابعد یک مقدار x در خوشه‌ی مربوط به همان مقدار است اگر و فقط اگر x اکیداً کوچک‌تر از خصیصه‌ی max در آن خوشه باشد. یک بحث مشابه در مورد PREDECESSOR و min صادق است.



شکل ۲۰-۶ یک درخت $vEB(۱۶)$ متناظر با درخت vEB بدوی شکل ۲۰-۴. در این درخت مجموعه‌ی {۲, ۳, ۴, ۵, ۷, ۱۴, ۱۵} ذخیره شده است. ممیزها نشان‌دهنده‌ی مقادیر NIL هستند. مقدار ذخیره شده در خصیصه‌ی *min* یک درخت vEB در هیچ یک از خوشه‌های آن ظاهر نمی‌شود. کاربرد سایه‌ی تیره در این شکل مشابه شکل ۲۰-۴ است.

۳. با استفاده از مقادیر *min* و *max* می‌توانیم در زمان ثابت تعیین کنیم که یک درخت vEB هیچ عنصری ندارد، یک عنصر دارد، و یا حداقل دو عنصر دارد. این قابلیت در اعمال INSERT و DELETE به کار خواهد آمد. اگر *min* و *max* هر دو NIL باشند آن گاه درخت vEB هیچ عنصری ندارد. اگر *min* و *max* غیر NIL باشند، ولی با هم برابر باشند، آن گاه درخت vEB دقیقاً یک عنصر دارد. در غیر این صورت، هر دوی *min* و *max* غیر NIL هستند، و همچنین نامساوی، و درخت vEB دو عنصر یا بیشتر دارد.

۴. اگر بدانیم یک درخت vEB تهی است، می‌توانیم درج را به سادگی به وسیله‌ی به هنگام‌سازی خصیصه‌های *min* و *max* آن انجام دهیم. بنابراین می‌توانیم درج در یک درخت vEB تهی را در زمان ثابت انجام دهیم. به طور مشابه اگر بدانیم که یک درخت vEB فقط یک عنصر دارد، می‌توانیم در زمان ثابت و با به هنگام‌سازی خصیصه‌های *min* و *max* آن عنصر را حذف کنیم.

این خصوصیات به ما اجازه می‌دهد که زنجیره‌ی فراخوانی‌های بازگشتی را کوتاه کنیم.

حتی اگر u، اندازه‌ی مجموعه‌ی جهانی، توانی فرد از ۲ باشد، تفاوت ایجاد شده در اندازه‌ی درخت vEB نماینده و خوشه‌ها زمان اجرای حدی اعمال درختان vEB را تغییر نخواهد داد. زمان اجرای رویه‌های بازگشتی که اعمال درختان vEB را پیاده‌سازی می‌کنند همگی به وسیله‌ی رابطه‌ی بازگشتی زیر توصیف خواهند شد:

$$T(u) \le T(\sqrt[\uparrow]{u}) + O(1) \qquad (۲۰-۴)$$

این رابطه‌ی بازگشتی شبیه رابطه‌ی (۲۰-۲) است، و ما هم آن را به روشی مشابه حل خواهیم کرد. با فرض این که $m = \lg u$، این رابطه را به صورت زیر بازنویسی می‌کنیم:

$$T(2^m) \le T(2^{\lceil m/2 \rceil}) + O(1)$$

با توجه به این که $\lceil m/2 \rceil \le 2m/3$، برای هر $m \ge 2$ داریم

$$T(2^m) \le T(2^{2m/3}) + O(1)$$

با قرار دادن $S(m) = T(2^m)$، رابطه‌ی بازگشتی آخر را به صورت زیر می‌نویسیم:

$$S(m) \le S(2m/3) + O(1)$$

که طبق حالت ۲ از متد اصلی دارای جواب $S(m) = O(\lg m)$ است. (از دید زمان اجرای حدی کسر ۲/۳ نسبت به کسر ۱/۲ هیچ تفاوتی ایجاد نمی‌کند، زیرا وقتی از متد اصلی استفاده می‌کنیم، می‌بینیم که $\log_{3/2} 1 = \log_2 1 = 0$.) بنابراین داریم $T(u) = T(2^m) = S(m) = O(\lg m) = O(\lg \lg u)$.

قبل از استفاده از یک درخت van Emde Boas باید اندازه‌ی مجموعه‌ی جهانی (u) را بدانیم، تا بتوانیم یک درخت van Emde Boas با اندازه‌ی مناسب بسازیم که در ابتدا نماینده‌ی یک مجموعه‌ی تهی است. همان طور که مسئله‌ی ۲۰-۱ از شما می‌خواهد نشان دهید، کل فضای مورد نیاز برای یک درخت van Emde Boas از مرتبه‌ی $O(u)$ است، و ساختن یک درخت تهی در زمان $O(u)$ کاملاً سرراست است. در مقابل می‌توانیم یک درخت قرمز-سیاه تهی را در زمان ثابت بسازیم. بنابراین وقتی که تعداد اعمالی که می‌خواهیم انجام دهیم کم است، ممکن است بهتر باشد از درختان van Emde Boas استفاده نکنیم، چرا که زمان ساختن ساختمان داده از زمان صرفه‌جویی شده در اعمال مختلف فراتر می‌رود. این مانع در اکثر موارد قابل توجه نیست، چرا که معمولاً برای نمایش یک مجموعه با تعداد عنصر کم، از یک ساختمان داده‌ی ساده مانند یک آرایه یا یک لیست پیوندی استفاده می‌کنیم.

۲۰-۳-۲ اعمال بر روی یک درخت van Emde Boas

اکنون آماده‌ایم که ببینیم چگونه می‌توان اعمال مختلف را بر روی یک درخت van Emde Boas اجرا کرد. همان طور که برای ساختارهای van Emde Boas بدوی این کار را کردیم، در این جا هم ابتدا اعمال پرسشی را بررسی می‌کنیم، و سپس به سراغ INSERT و DELETE می‌رویم. به دلیل ناهماهنگی

کوچک موجود بین عناصر کمینه و بیشینه در یک درخت vEB – وقتی یک vEB حداقل حاوی دو عنصر باشد، عنصر کمینه در هیچ خوشه‌ای حضور نخواهد داشت، ولی عنصر بیشینه در یکی از خوشه‌ها وجود دارد – شبه‌کد هر پنج عمل پرسشی را در این جا ارائه خواهیم کرد. مانند اعمال ساختارهای van Emde Boas بدوی در این جا هم اعمالی که پارامترهای V و x را دریافت می‌کنند، که در آن V یک درخت van Emde Boas است، و x یک عنصر، فرض می‌کنند که $V.u > x \geq 0$.

یافتن عناصر کمینه و بیشینه

از آن جایی که کمینه و بیشینه را در خصیصه‌های min و max ذخیره می‌کنیم، این دو عملیات تک خطی هستند، و در زمان ثابت اجرا می‌شوند:

```
vEB-TREE-MINIMUM(V)
1  return V.min

vEB-TREE-MAXIMUM(V)
1  return V.max
```

تعیین وجود یک مقدار در مجموعه

رویه‌ی vEB-TREE-MEMBER(V, x) یک حالت بازگشتی مشابه حالت بازگشتی PROTO-vEB-MEMBER دارد، ولی حالت پایه‌ی آن کمی متفاوت است. همچنین مستقیماً چک می‌کنیم که آیا x برابر با عناصر کمینه یا بیشینه هست یا نه. چون درخت‌های vEB بیت‌ها را مانند ساختارهای vEB بدوی ذخیره نمی‌کند، vEB-TREE-MEMBER را طوری طراحی می‌کنیم که به جای ۰ یا ۱، TRUE یا FALSE را بازگرداند.

```
vEB-TREE-MEMBER(V, x)
1  if x == V.min or x == V.max
2      return TRUE
3  elseif V.u == 2
4      return FALSE
5  else return vEB-TREE-MEMBER(V.cluster[high(x)], low(x))
```

خط ۱ چک می‌کند که x برابر با هیچ یک از عناصر کمینه یا بیشینه است یا نه. اگر چنین باشد، خط ۲ مقدار TRUE را بازمی‌گرداند. در غیر این صورت خط ۳ وجود حالت پایه را بررسی می‌کند. از آن جایی که یک درخت $vEB(2)$ هیچ عنصری غیر از آن‌هایی که در min و max ذخیره شده‌اند، ندارد، اگر در حالت پایه باشیم خط ۴ مقدار FALSE را بازمی‌گرداند. حالت ممکن دیگر – وقتی در حالت پایه نیستیم و x برابر با هیچ یک از min و max نیست – توسط فراخوانی بازگشتی در خط ۵ اداره می‌شود.

رابطه‌ی بازگشتی (۲۰-۴) زمان اجرای رویه‌ی vEB-TREE-MEMBER را توصیف می‌کند، و بنابراین این رویه در زمان $O(\lg \lg u)$ اجرا خواهد شد.

یافتن عناصر مابعد و ماقبل

اکنون نحوه‌ی پیاده‌سازی عملیات SUCCESSOR را خواهیم دید. به خاطر بیاورید که رویه‌ی PROTO-vEB-SUCCESSOR (V, x) می‌تواند دو فراخوانی بازگشتی انجام دهد: یکی برای تعیین وجود عنصر مابعد x در همان خوشه‌ای که x در آن قرار دارد، و اگر این چنین نبود، یکی هم برای یافتن خوشه‌ی حاوی عنصر مابعد x. از آن جایی که به سرعت می‌توانیم به مقدار بیشینه در یک درخت vEB دسترسی داشته باشیم، می‌توانیم از انجام دو فراخوانی بازگشتی پرهیز کرده و فقط یک فراخوانی بازگشتی بر روی یک خوشه یا روی نماینده انجام دهیم، ولی نه هردو با هم.

```
vEB-TREE-SUCCESSOR(V, x)
1   if V.u == 2
2       if x == 0 and V.max == 1
3           return 1
4       else return NIL
5   elseif V.min ≠ NIL and x < V.min
6       return V.min
7   else max-low = vEB-TREE-MAXIMUM(V.cluster[high(x)])
8       if max-low ≠ NIL and low(x) < max-low
9           offset = vEB-TREE-SUCCESSOR(V.cluster[high(x)], low(x))
10          return index(high(x), offset)
11      else succ-cluster = vEB-TREE-SUCCESSOR(V.summary, high(x))
12          if succ-cluster == NIL
13              return NIL
14          else offset = vEB-TREE-MINIMUM(V.cluster[succ-cluster])
15              return index(succ-cluster, offset)
```

این رویه شش عبارت‌های return و حالت‌های مختلف بسیاری دارد. با حالت پایه در خطوط ۲-۴ آغاز می‌کنیم، که در آن، اگر بخواهیم عنصر مابعد ۰ را پیدا کنیم، و ۱ در مجموعه‌ی ۲ عنصری موجود باشد، در خط ۳ مقدار ۱ بازگردانده خواهد شد؛ در غیر این صورت حالت پایه در خط ۴ مقدار NIL بازخواهد گرداند.

اگر در حالت پایه نباشیم، در خط ۵ چک می‌کنیم که x اکیدا کوچک‌تر از عنصر کمینه هست یا نه. اگر این طور بود، به سادگی در خط ۶ عنصر کمینه را بازمی‌گردانیم.

اگر به خط ۷ برسیم خواهیم دانست که در حالت پایه نیستیم، و x بزرگ‌تر یا مساوی مقدار کمینه در درخت vEB داده‌شده‌ی V است. خط ۷ عنصر بیشینه در خوشه‌ی x را به $max\text{-}low$ نسبت می‌دهد. اگر خوشه‌ی مربوط به x حاوی عنصری بزرگ‌تر از x باشد، آن گاه خواهیم دانست که عنصر مابعد x جایی در خوشه‌ی مربوط به x قرار دارد. خط ۸ وجود این حالت را چک می‌کند. اگر عنصر مابعد x درون خوشه‌ی مربوط به x باشد، آن گاه خط ۹ جای آن در خوشه را مشخص می‌کند، و خط ۱۰ عنصر مابعد را مشابه خط ۷ رویه‌ی PROTO-vEB-SUCCESSOR بازمی‌گرداند.

اگر x بزرگ‌تر یا مساوی بزرگ‌ترین عنصر در خوشه‌ی خود باشد، به خط ۱۱ خواهیم رسید. در این حالت خطوط ۱۱–۱۵ عنصر مابعد را مشابه خطوط ۸–۱۲ رویه‌ی PROTO-vEB-SUCCESSOR پیدا خواهند کرد.

به سادگی می‌توان دید که باز هم رابطه‌ی بازگشتی (۲۰–۴) زمان اجرای رویه‌ی vEB-TREE-SUCCESSOR را توصیف می‌کند. بسته به نتیجه‌ی تست در خط ۷، رویه به صورت بازگشتی یا در خط ۹ (روی یک درخت vEB با مجموعه‌ی جهانی با اندازه‌ی $\sqrt[\downarrow]{u}$) و یا در خط ۱۱ (روی یک درخت vEB با مجموعه‌ی جهانی با اندازه‌ی $\sqrt[\uparrow]{u}$) خود را فراخوانی می‌کند. در هر دو حالت تنها فراخوانی بازگشتی بر روی یک درخت vEB با مجموعه‌ی جهانی با اندازه‌ی حداکثر $\sqrt[\uparrow]{u}$ فراخوانی می‌شود. بقیه‌ی رویه، شامل فراخوانی‌های vEB-TREE-MINIMUM و vEB-TREE-MAXIMUM به زمان $O(1)$ نیاز دارد. بنابراین vEB-TREE-SUCCESSOR در بدترین حالت در زمان $O(\lg\lg u)$ اجرا می‌شود.

رویه‌ی vEB-TREE-PREDECESSOR مشابه رویه‌ی vEB-TREE-SUCCESSOR است، ولی یک حالت بیشتر از آن دارد:

```
vEB-TREE-PREDECESSOR(V, x)
 1  if V.u == 2
 2      if x == 1 and V.min == 0
 3          return 0
 4      else return NIL
 5  elseif V.max ≠ NIL and x > V.max
 6      return V.max
 7  else min-low = vEB-TREE-MINIMUM(V.cluster[high(x)])
 8      if min-low ≠ NIL and low(x) > min-low
 9          offset = vEB-TREE-PREDECESSOR(V.cluster[high(x)], high(x))
10          return index(high(x), offset)
11      else pred-cluster = vEB-TREE-PREDECESSOR(V.summary, high(x))
12          if pred-cluster == NIL
13              if V.min ≠ NIL and x > V.min
14                  return V.min
15              else return NIL
16          else offset = vEB-TREE-MAXIMUM(V.cluster[pred-cluster])
17              return index(pred-cluster, offset)
```

خطوط ۱۳–۱۴ حالت اضافی را تشکیل می‌دهند. این حالت وقتی رخ می‌دهد که عنصر ماقبل x، در صورت وجود، در خوشه‌ی مربوط به x قرار ندارد. در vEB-TREE-SUCCESSOR اطمینان داشتیم که اگر عنصر مابعد x بیرون خوشه‌ی مربوط به آن قرار داشته باشد، آن گاه باید در یک خوشه با شماره‌ی بالاتر باشد. ولی اگر عنصر ماقبل x مقدار کمینه در درخت V باشد، آن گاه عنصر ماقبل در هیچ خوشه‌ای نخواهد بود. خط ۱۳ وجود این وضعیت را چک می‌کند، و خط ۱۴ متناسباً مقدار کمینه

را بازخواهد گرداند.

در مقایسه با vEB-TREE-SUCCESSOR این حالت اضافی تأثیری بر روی زمان اجرای حدی رویه‌ی vEB-TREE-PREDECESSOR ندارد، و بنابراین vEB-TREE-PREDECESSOR در بدترین حالت در زمان $O(\lg\lg u)$ اجرا می‌شود.

درج یک عنصر

اکنون نحوه‌ی درج یک عنصر در یک درخت vEB را بررسی می‌کنیم. به خاطر بیاورید که PROTO-vEB-INSERT دو فراخوانی بازگشتی انجام می‌داد: یکی برای درج خود عنصر و یکی برای درج شماره‌ی خوشه‌ی عنصر در نماینده‌ی ساختار. رویه‌ی vEB-TREE-INSERT فقط یک فراخوانی بازگشتی انجام می‌دهد. چطور می‌توانیم فقط با یک بازگشت این کار را بکنیم؟ وقتی یک عنصر را درج می‌کنیم، خوشه‌ای که این عنصر در آن قرار می‌گیرد یا عنصر دیگری دارد یا ندارد. اگر خوشه از قبل حاوی عنصر دیگری باشد، آن گاه شماره‌ی خوشه هم از قبل در نماینده قرار دارد و دیگر نیازی به فراخوانی بازگشتی نیست. اگر خوشه عنصر دیگری نداشته باشد، عنصر درج شده تنها عنصر خوشه خواهد بود، و بنابراین نیازی به بازگشت برای درج یک عنصر در یک درخت vEB تهی نداریم:

```
vEB-EMPTY-TREE-INSERT(V, x)
1  V.min = x
2  V.max = x
```

با در دسترس داشتن این رویه، در زیر شبه‌کدی برای vEB-TREE-INSERT(V, x) ارائه می‌کنیم که فرض می‌کند x از قبل عنصری از مجموعه‌ی متناظر با درخت vEB نیست:

```
vEB-TREE-INSERT(V, x)
1  if V.min == NIL
2      vEB-EMPTY-TREE-INSERT(V, x)
3  else if x < V.min
4          exchange x with V.min
5      if V.u > 2
6          if vEB-TREE-MINIMUM(V.cluster[high(x)]) == NIL
7              vEB-TREE-INSERT(V.summary, high(x))
8              vEB-TREE-EMPTY-TREE-INSERT(V.cluster[high(x)], low(x))
9          else vEB-TREE-INSERT(V.cluster[high(x)], low(x))
10     if x > V.max
11         V.max = x
```

این رویه به صورت زیر کار می‌کند. خط ۱ چک می‌کند که V یک درخت vEB تهی است یا نه، و اگر بود، خط ۲ به سادگی این حالت را اداره می‌کند. خطوط ۳–۱۱ فرض می‌کنند که V تهی نیست، و بنابراین یک عنصر در یکی از خوشه‌های V درج خواهد شد. ولی عنصری که درج می‌شود لزوماً عنصر x داده شده به vEB-TREE-INSERT نیست. اگر $x < min$، همان طور که در خط ۳ چک

می‌شود، آن گاه x باید تبدیل به min جدید شود. ولی نمی‌خواهیم عنصر min اولیه را از دست بدهیم، و بنابراین باید آن را در یکی از خوشه‌های V درج کنیم. در این حالت خط ۴ عنصر x را با min جابه‌جا می‌کند، تا min اولیه را در یکی از خوشه‌های V درج کنیم.

خطوط ۶-۹ فقط زمانی اجرا می‌شوند که V یک درخت vEB پایه نباشد. خط ۶ تعیین می‌کند که آیا خوشه‌ای که x درون آن قرار خواهد گرفت، تهی است یا نه. اگر چنین بود خط ۷ شماره‌ی خوشه‌ی x را در خلاصه‌ی درخت درج می‌کند، و خط ۸ عملیات ساده‌ی درج x در یک مجموعه‌ی تهی را انجام می‌دهد. اگر خوشه از قبل تهی نباشد، آن گاه خط ۹ عنصر x را در خوشه‌ی مناسب درج می‌کند. در این حالت به هنگام‌سازی خلاصه ضروری نیست، چرا که شماره‌ی خوشه‌ی x از قبل عضوی از خلاصه است.

نهایتاً اگر داشته باشیم $x > max$، خطوط ۱۰-۱۱ به هنگام‌سازی max را بر عهده خواهند داشت. توجه کنید که اگر V یک درخت vEB پایه و ناتهی باشد، آن گاه خطوط ۳-۴ و ۱۰-۱۱ عنصر min و max را متناسباً به هنگام‌سازی می‌کنند.

دوباره، به سادگی می‌توانیم ببینیم که رابطه‌ی بازگشتی (۲۰-۴) زمان اجرا را توصیف می‌کنند. بسته به نتیجه‌ی تست در خط ۶، یا فراخوانی بازگشتی در خط ۷ (بر روی یک درخت vEB با اندازه‌ی مجموعه‌ی جهانی $\uparrow\sqrt{u}$) اجرا خواهد شد، و یا فراخوانی بازگشتی در خط ۹ (بر روی یک درخت vEB با اندازه‌ی مجموعه‌ی جهانی $\downarrow\sqrt{u}$). در هر دو حالت تنها فراخوانی بازگشتی بر روی یک درخت vEB با مجموعه‌ی جهانی با اندازه‌ی حداکثر $\uparrow\sqrt{u}$ انجام خواهد شد. چون بقیه‌ی رویه‌ی vEB-TREE-INSERT به زمان $O(1)$ نیاز دارد، رابطه‌ی بازگشتی (۲۰-۴) برای این رویه صادق است، و بنابراین زمان اجرا برابر است با $O(\lg\lg u)$.

حذف یک عنصر

نهایتاً به نحوه‌ی حذف یک عنصر از یک درخت vEB می‌رسیم. رویه‌ی vEB-TREE-DELETE(V, x) فرض می‌کند که x عضوی از مجموعه‌ی متناظر با درخت vEB دریافت شده‌ی V است.

```
vEB-TREE-DELETE(V, x)
1   if V.min == V.max
2       V.min = NIL
3       V.max = NIL
4   elseif V.u == 2
5       if x == 0
6           V.min = 1
7       else V.min = 0
8       V.max = V.min
9   else if x == V.min
10          first-cluster = vEB-TREE-MINIMUM(V.summary)
11          x = index(first-cluster,
```

```
                vEB-TREE-MINIMUM(V.cluster[first-cluster]))
12          V.min = x
13      vEB-TREE-DELETE(V.cluster[high(x)], low(x))
14      if vEB-TREE-MINIMUM(V.cluster[high(x)]) == NIL
15          vEB-TREE-DELETE(V.summary, high(x))
16          if x == V.max
17              summary-max = vEB-TREE-MAXIMUM(V.summary)
18              if summary-max == NIL
19                  V.max = V.min
20              else V.max = index(summary-max,
                        vEB-TREE-MAXIMUM(V.cluster[summary-max]))
21      elseif x == V.max
22          V.max = index(high(x),
                    vEB-TREE-MAXIMUM(V.cluster[high(x)]))
```

رویه‌ی vEB-TREE-DELETE به صورت زیر کار می‌کند. اگر درخت vEB ارسال شده‌ی V فقط حاوی یک عنصر باشد، آن گاه حذف این عنصر به سادگی درج یک عنصر در یک درخت vEB تهی خواهد بود: فقط باید *min* و *max* را برابر با NIL قرار دهیم. خطوط ۱–۳ با این حالت سروکار دارند. در غیر این صورت V حداقل دو عنصر دارد. خط ۴ پایه بودن یا نبودن درخت vEB داده شده را تعیین می‌کند، و اگر بود، خطوط ۵–۸ هر دو عنصر *min* و *max* را برابر با تنها عنصر باقی‌مانده قرار می‌دهند.

خطوط ۹–۲۲ فرض می‌کنند که V دو عنصر یا بیشتر دارد، و $u \geq 4$. در این حالت باید یک عنصر از یک خوشه حذف کنیم. ولی عنصری که از خوشه حذف می‌کنیم ممکن است x نباشد، چرا که اگر x برابر با *min* باشد، آن گاه وقتی x را حذف کردیم عنصر دیگری در یکی از خوشه‌های V تبدیل به *min* جدید می‌شود، و باید آن عنصر دیگر را از خوشه‌اش حذف کنیم. اگر در تست خط ۹ مشخص شود که در این وضعیت قرار داریم، آن گاه خط ۱۰ متغیر *first-cluster* را برابر با شماره‌ی خوشه‌ای قرار می‌دهد که حاوی پایین‌ترین عنصر غیر از *min* است، و خط ۱۱، x را برابر با مقدار آن پایین‌ترین عنصر در خوشه قرار می‌دهد. این عنصر در خط ۱۲ تبدیل به *min* جدید می‌شود، و چون x را برابر با مقدار آن قرار داده‌ایم، این عنصری است که از خوشه‌اش حذف می‌شود.

وقتی به خط ۱۳ می‌رسیم، می‌دانیم که باید عنصر x را از خوشه‌اش حذف کنیم، چه مقدار x همانی باشد که از ابتدا به vEB-TREE-DELETE ارسال شده است، و چه x عنصری باشد که تبدیل به کمینه‌ی جدید خواهد شد. خط ۱۳ عنصر x را از خوشه‌اش حذف می‌کند. این خوشه ممکن است اکنون تهی شده باشد، که وجود این وضعیت در خط ۱۴ تست می‌شود، و اگر شده باشد، آن گاه باید شماره‌ی خوشه‌ی x را از نماینده حذف کنیم، که در خط ۱۵ انجام می‌شود. پس از به هنگام‌سازی نماینده، ممکن است نیاز داشته باشیم که *max* را هم به هنگام‌سازی کنیم. خط ۱۶ چک می‌کند که آیا داریم عنصر بیشینه در V را حذف می‌کنیم یا نه، و اگر چنین بود، خط ۱۷ متغیر *summary-max* را

برابر با شماره‌ی خوشه‌ی ناتهی با بالاترین شماره قرار می‌دهد. (فراخوانی vEB-TREE-MAXIMUM ($V.summary$) کار می‌کند چرا که از قبل به صورت بازگشتی vEB-TREE-DELETE را روی $V.summary$ فراخوانی کرده‌ایم، و بنابراین $V.summary.max$ در صورت نیاز از قبل به هنگام‌سازی شده است.) اگر تمام خوشه‌های V تهی باشند، آن گاه تنها عنصر باقی‌مانده در V عنصر min است؛ خط ۱۸ وجود این حالت را بررسی می‌کند، و خط ۱۹ عنصر max را به شکل مناسب به هنگام‌سازی می‌کند. در غیر این صورت خط ۲۰ عنصر max را برابر با عنصر بیشینه در خوشه‌ی با بالاترین شماره قرار می‌دهد. (اگر این خوشه همانی باشد که حذف عنصر در آن جا انجام شده است، دوباره به خط ۱۳ اطمینان می‌کنیم که از قبل خصیصه‌ی max خوشه را تصحیح کرده است).

اکنون نشان می‌دهیم که vEB-TREE-DELETE در بدترین حالت در زمان $O(\lg\lg u)$ اجرا می‌شود. در نگاه اول ممکن است فکر کنید که رابطه‌ی (۲۰-۴) همیشه در مورد این رویه صادق نیست، چرا که یک فراخوانی از vEB-TREE-DELETE می‌تواند دو فراخوانی بازگشتی انجام دهد: یکی در خط ۱۳ و یکی در خط ۱۵. با این که رویه می‌تواند هر دو فراخوانی بازگشتی را انجام دهد، اجازه دهید فکر کنیم که اگر این کار را انجام دهد چه رخ می‌دهد. برای این که فراخوانی بازگشتی خط ۱۵ انجام شود، تست خط ۱۴ باید نشان دهد که خوشه‌ی x تهی است. تنها حالتی که در آن خوشه‌ی x می‌تواند تهی باشد این است که x هنگام اجرای فراخوانی بازگشتی خط ۱۳، تنها عنصر خوشه‌ی خود باشد. ولی اگر x تنها عنصر خوشه‌ی خود باشد، آن گاه آن فراخوانی بازگشتی فقط به اندازه‌ی $O(1)$ زمان صرف کرده است، چرا که فقط خطوط ۱-۳ را اجرا می‌کند. بنابراین دو امکان مجزا از یکدیگر داریم:

- فراخوانی بازگشتی خط ۱۳ در زمان ثابت اجرا شده است.
- فراخوانی بازگشتی خط ۱۵ اجرا نشده است.

در هر دو حالت، رابطه‌ی بازگشتی (۲۰-۴) توصیف کننده‌ی زمان اجرای vEB-TREE-DELETE است، و بنابراین زمان اجرای آن در بدترین حالت برابر است با $O(\lg\lg u)$.

تمرین‌ها

۲۰-۳-۱ درختان vEB را طوری اصلاح کنید که از کلیدهای با مقدار تکراری پشتیبانی کنند.

۲۰-۳-۲ درختان vEB را طوری اصلاح کنید که از داده‌های پیرو به همراه کلیدها پشتیبانی کنند.

۲۰-۳-۳ شبه‌کدی ارائه کنید برای رویه‌ای که یک درخت van Emde Boas تهی می‌سازد.

۲۰-۳-۴ چه رخ خواهد داد اگر vEB-TREE-INSERT را با عنصری فراخوانی کنیم که از قبل در درخت vEB قرار دارد؟ اگر vEB-TREE-DELETE را با عنصری فراخوانی کنیم که در درخت vEB وجود ندارد چطور؟ توضیح دهید که چرا این رویه‌ها این رفتارها را از خود نشان می‌دهند. توضیح دهید که چطور می‌توان درختان vEB و اعمال آن‌ها را طوری اصلاح کرد به که بتوانیم در زمان ثابت چک کنیم آیا عنصری در مجموعه قرار دارد یا نه.

۲۰-۳-۵ فرض کنید به جای $\sqrt[\uparrow]{u}$ خوشه هر یک با اندازه‌ی مجموعه‌ی جهانی $\sqrt[\downarrow]{u}$، طوری درختان vEB را می‌ساختیم که $u^{1/k}$ خوشه داشته باشند، هر یک با مجموعه‌ی جهانی با اندازه‌ی $u^{1-1/k}$، که در آن $k > 1$ یک ثابت است. اگر می‌خواستیم اعمال مربوطه را به شکل مناسب اصلاح کنیم، زمان اجرای آن‌ها چقدر می‌شد؟ برای تحلیل ساده‌تر، فرض کنید که $u^{1/k}$ و $u^{1-1/k}$ همیشه اعداد صحیح هستند.

۲۰-۳-۶ ساختن یک درخت vEB با مجموعه‌ی جهانی با اندازه‌ی u به زمان $O(u)$ نیاز دارد. فرض کنید می‌خواهیم صریحاً این زمان را به حساب بیاوریم. کم‌ترین تعداد اعمال که به ازای آن زمان اجرای سرشکن هر یک از اعمال درختان vEB برابر با $O(\lg\lg u)$ باشد چقدر است؟

مسائل

۲۰-۱ ***فضای مورد نیاز برای درختان van Emde Boas***

این مسئله فضای مورد نیاز درختان van Emde Boas را بررسی می‌کند، و روشی پیشنهاد می‌کند برای اصلاح این ساختمان داده به طوری که فضای مورد نیاز آن به جای u، اندازه‌ی مجموعه‌ی جهانی، به n، تعداد عناصری که واقعاً در درخت ذخیره شده‌اند، بستگی داشته باشد. برای سادگی فرض کنید $\sqrt{u}$ همیشه یک عدد صحیح است.

a. توضیح دهید که چرا رابطه‌ی بازگشتی زیر توصیف کننده‌ی فضای مورد نیاز یک درخت van Emde Boas با مجموعه‌ی جهانی با اندازه‌ی u است:

$$P(u) = (\sqrt{u}+1)P(\sqrt{u}) + \theta(\sqrt{u}) \qquad (۲۰-۵)$$

b. اثبات کنید که جواب رابطه‌ی بازگشتی (۲۰-۵) عبارت است از $P(u) = O(u)$.

برای کاهش فضای مورد نیاز، اجازه دهید یک *درخت van Emde Boas با حجم کاهش یافته* (reduced-space van Emde Boas tree)، یا *درخت RS-vEB* را به صورت یک درخت vEB (مثلاً با نام V) تعریف کنیم، ولی با تغییرات زیر:

- خصیصه‌ی $V.cluster$ به جای این که به صورت یک آرایه‌ی ساده از اشاره‌گرها به درختان vEB با مجموعه‌ی جهانی با اندازه‌ی $\sqrt{u}$ ذخیره شده باشد، یک جدول درهم (فصل ۱۱ را ببینید) است که به صورت یک جدول پویا (بخش ۱۷-۴ را ببینید) ذخیره شده است. طبق نسخه‌ی آرایه‌ای $V.cluster$، جدول درهم اشاره‌گرهایی به درختان RS-vEB با مجموعه‌ی جهانی با اندازه‌ی $\sqrt{u}$ در خود ذخیره می‌کند. برای یافتن iامین خوشه، در جدول درهم برای کلید i جستجو می‌کنیم، و بدین صورت می‌توانیم خوشه‌ی iام را با یک جستجو در جدول درهم بیابیم.

- جدول درهم فقط اشاره‌گرهایی به خوشه‌های ناتهی در خود ذخیره می‌کند. جستجو برای یک خوشه‌ی تهی در جدول درهم مقدار NIL را بازمی‌گرداند، که نشان‌دهنده‌ی تهی بودن خوشه است.
- اگر تمام خوشه‌ها تهی باشند خصیصه‌ی $V.summary$ برابر با NIL است. در غیر این صورت $V.summary$ به یک درخت RS-vEB با مجموعه‌ی جهانی با اندازه‌ی $\sqrt{u}$ اشاره می‌کند.

چون جدول درهم با یک جدول پویا پیاده‌سازی شده است، فضای مورد نیاز آن متناسب است با تعداد خوشه‌های ناتهی.

وقتی می‌خواهیم یک عنصر در یک درخت RS-vEB تهی درج کنیم، همان موقع، به کمک رویه‌ی زیر، درخت مورد نظر را می‌سازیم. در رویه‌ی زیر u نشان دهنده‌ی اندازه‌ی مجموعه‌ی جهانی مربوط به درخت RS-vEB است:

```
CREATE-NEW-RS-vEB-TREE(u)
1  allocate a new vEB tree V
2  V.u = u
3  V.min = NIL
4  V.max = NIL
5  V.summary = NIL
6  create V.cluster as an empty dynamic hash table
7  return V
```

III. رویه‌ی vEB-TREE-INSERT را اصلاح کرده و از آن شبه‌کدی برای رویه‌ی RS-vEB-TREE-INSERT(V, x) ارائه کنید، که عنصر x را در درخت RS-vEB داده شده‌ی V درج می‌کند. هنگام نیاز رویه‌ی CREATE-NEW-RS-vEB-TREE را در کد ارائه شده فراخوانی کنید.

IV. رویه‌ی vEB-TREE-SUCCESSOR را اصلاح کرده و از آن شبه‌کدی برای رویه‌ی RS-vEB-TREE-SUCCESSOR(V, x) ارائه کنید، که در صورت وجود، عنصر ماقبل x را در درخت RS-vEB داده شده‌ی V، و در غیر این صورت مقدار NIL را بازمی‌گرداند.

V. اثبات کنید که با فرض درهم‌سازی ساده‌ی متوازن، رویه‌های RS-vEB-TREE-INSERT و RS-vEB-TREE-SUCCESSOR، که در بخش قبل ارائه کردید، با امیدریاضی زمان $O(\lg\lg u)$ اجرا می‌شوند.

VI. با فرض این که عناصر هیچ وقت از یک درخت vEB حذف نمی‌شوند، اثبات کنید که فضای مورد نیاز برای ساختار RS-vEB برابر است با $O(n)$، که در آن n نشان‌دهنده‌ی تعداد عناصری است که واقعاً در درخت RS-vEB ذخیره شده‌اند.

VII. درختان RS-vEB یک مزیت دیگر نسبت به درختان vEB دارند: زمان مورد نیاز ساختن آن‌ها کم‌تر است. ساختن یک درخت RS-vEB چقدر طول می‌کشد؟

۲۰-۳ ترای‌های y-سریع

در این مسئله، «ترای‌های y-سریع» معرفی و بررسی می‌شوند، که D. Willard آن‌ها را ارائه کرده است. در این ساختارها، مانند درختان van Emde Boas، هر یک از اعمال MEMBER، MINIMUM، MAXIMUM، PREDECESSOR، و SUCCESSOR بر روی عناصر استخراج شده از یک مجموعه‌ی جهانی با اندازه‌ی u در بدترین حالت در زمان $O(\lg\lg u)$ اجرا می‌شوند. اعمال INSERT و DELETE به زمان سرشکن $O(\lg\lg u)$ نیاز دارند. مانند درختان van Emde Boas با حجم کاهش یافته (مسئله‌ی ۲۰-۱ را ببینید)، ترای‌های y - سریع برای ذخیره‌ی n عنصر فقط به فضای $O(n)$ نیاز دارند. طراحی ترای‌های y - سریع بر پایه‌ی درهم‌سازی کامل است (بخش ۱۱-۵ را ببینید).

به عنوان یک ساختمان داده‌ی ابتدایی، فرض کنید یک جدول درهم کامل می‌سازیم که علاوه بر خود عناصر مجموعه‌ی پویا، هر پیش‌وندی از نمایش دودویی عناصر را هم در خود ذخیره می‌کند. برای مثال، اگر داشته باشیم $u = ۱۶$، به طوری که $\lg u = ۴$، و $x = ۱۳$ در مجموعه باشد، آن گاه چون نمایش دودویی ۱۳ به صورت ۱۱۰۱ است، جدول درهم کامل حاوی رشته‌های ۱، ۱۱، ۱۱۰، و ۱۱۰۱ خواهد بود. علاوه بر جدول درهم، یک لیست پیوندی دوطرفه از عناصر درون مجموعه به ترتیب صعودی می‌سازیم.

این ساختار به چه مقدار حافظه نیاز دارد؟

نشان دهید که چطور می‌توان اعمال MINIMUM و MAXIMUM را در زمان $O(۱)$، اعمال MEMBER، PREDECESSOR، و SUCCESSOR را در زمان $O(\lg\lg u)$، و اعمال INSERT و DELETE را در زمان $O(\lg u)$ پیاده‌سازی کرد.

برای کاهش فضای مورد نیاز به $O(n)$، تغییرات زیر را روی ساختمان داده اعمال می‌کنیم:

- n عنصر مجموعه را در $n/\lg u$ گروه با اندازه‌ی $\lg u$ خوشه‌بندی می‌کنیم. (فعلاً فرض کنید که n بر $\lg u$ بخش‌پذیر است.) اولین گروه حاوی $\lg u$ عنصر کوچک مجموعه است، گروه دوم حاوی دومین $\lg u$ عنصر کوچک مجموعه، و همین طور تا آخر.
- برای هر گروه یک مقدار «نماینده» در نظر می‌گیریم. نماینده‌ی گروه iام حداقل به بزرگی بزرگ‌ترین عنصر در گروه iام است، و همچنین از تمام عناصر گروه $(i+۱)$ام کوچک‌تر است. (نماینده‌ی آخرین گروه می‌تواند بزرگ‌ترین عنصر ممکن، یعنی $u-۱$ باشد.) توجه کنید که یک نماینده ممکن است مقداری در مجموعه نباشد.
- $\lg u$ عنصر هر گروه را در یک درخت جستجوی دودویی متوازن، مثلاً یک درخت قرمز-سیاه، ذخیره می‌کنیم. هر نماینده به درخت جستجوی دودویی متوازن مربوط به گروه خود اشاره می‌کند، و هر درخت جستجوی دودویی متوازن به نماینده‌ی گروه خود.
- جدول درهم کامل فقط نماینده‌ها را ذخیره می‌کند، که همچنین در یک لیست پیوندی دوطرفه به ترتیب صعودی ذخیره شده‌اند.

این ساختار را یک ترای y - سریع (y-fast trie) می‌نامیم.

III. نشان دهید که یک ترای y - سریع برای ذخیره‌ی n عنصر فقط به فضای $O(n)$ نیاز دارد.

IV. نشان دهید که چطور می‌توان اعمال MINIMUM و MAXIMUM را روی یک ترای y - سریع در زمان $O(\lg\lg u)$ اجرا کرد.

V. نشان دهید که چطور می‌توان عملیات MEMBER را در زمان $O(\lg\lg u)$ اجرا کرد.

VI. نشان دهید که چطور می‌توان اعمال PREDECESSOR و SUCCESSOR را در زمان $O(\lg\lg u)$ اجرا کرد.

VII. توضیح دهید که چرا اعمال INSERT و DELETE به زمان $\Omega(\lg\lg u)$ نیاز دارند.

VIII. نشان دهید که چطور می‌توانیم این شرط را که هر گروه در یک ترای y - سریع باید دقیقاً $\lg u$ عنصر باشد، ملایم‌تر کنیم تا بتوانیم اعمال INSERT و DELETE را در زمان سرشکن $O(\lg\lg u)$ اجرا کنیم، بدون این که زمان اجرای حدی اعمال دیگر تغییر کند.

فصل

۲۱

ساختمان‌های داده برای مجموعه‌های منفصل

بعضی از کاربردها نیاز به دسته‌بندی n عنصر مجزا در گروهی از مجموعه‌های منفصل دارند. در این صورت دو عملیات مهم، یافتن مجموعه‌ای که یک عنصر به آن تعلق دارد و اجتماع دو مجموعه خواهد بود. این فصل متدهایی را برای طراحی ساختمان داده‌ای ارائه خواهد کرد که از این اعمال پشتیبانی کند.

بخش ۲۱-۱ اعمالی را توصیف خواهد کرد که یک ساختمان داده‌ی مجموعه‌های منفصل از آن پشتیبانی می‌کند، و همچنین یک کاربرد ساده در این فصل ارائه خواهد شد. در بخش ۲۱-۲ نگاهی می‌اندازیم بر یک پیاده‌سازی لیست پیوندی برای مجموعه‌های منفصل. یک نمایش بهینه‌تر با استفاده از درختان ریشه‌دار در بخش ۲۱-۳ ارائه شده است. زمان اجرا با استفاده از نمایش درخت، برای تمام اهداف کاربردی خطی است، ولی به صورت تئوری بیش از خطی است. در بخش ۲۱-۴ یک تابع با سرعت رشد بسیار زیاد و همچنین معکوس آن با سرعت رشد بسیار کم معرفی و بحث خواهد شد، که تابع معکوس در زمان اجرای اعمال بر روی پیاده‌سازی بر مبنای درخت ظاهر می‌شود. سپس از تحلیل سرشکن استفاده می‌کنیم تا یک کران بالا بر روی زمان اجرا اثبات کنیم که به سختی از زمان اجرای خطی بیشتر است.

۲۱-۱ اعمال مجموعه‌های منفصل

یک *ساختمان داده‌ی مجموعه‌های منفصل* (disjoint-set data structure)، مجموعه‌ی $\mathcal{S} = \{S_1, S_2, \ldots, S_\ell\}$ را از مجموعه‌های منفصل پویا نگه می‌دارد. هر مجموعه به وسیله‌ی یک *نماینده* (representative) شناسایی می‌شود که عضوی است از مجموعه. در بعضی از کاربردها این که از کدام عضو به عنوان نماینده استفاده شود، اهمیتی ندارد؛ تنها نکته‌ی مهم این است که اگر دو بار نماینده‌ی یک مجموعه‌ی

پویا را فراخوانی کنیم، بدون این که بین این فراخوانی‌ها مجموعه را تغییر دهیم، این نماینده یکسان باشد. در کاربردهای دیگر ممکن است قانون‌های خاصی برای انتخاب نماینده وجود داشته باشد، مانند انتخاب کوچک‌ترین عضو در مجموعه (البته با این فرض که بتوان عناصر را با هم مقایسه کرد).

مانند پیاده‌سازی بقیه‌ی مجموعه‌های پویایی که آموختیم، هر عنصر مجموعه توسط یک شیئ نشان داده می‌شود. با فرض این که x نشان‌دهنده‌ی یک شیئ باشد، می‌خواهیم از اعمال زیر پشتیبانی کنیم:

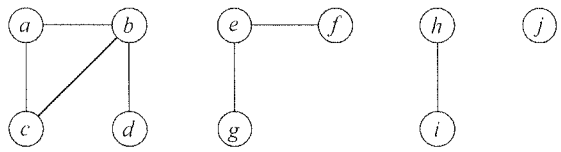
- MAKE-SET(x) یک مجموعه‌ی جدید می‌سازد که تنها عضو آن (و همچنین نماینده‌ی آن) x است. از آن جایی که مجموعه‌ها منفصل هستند، نیاز داریم که x در مجموعه‌ای دیگر نباشد.
- UNION(x, y) اجتماع دو مجموعه‌ای را که حاوی x و y هستند، مثلاً S_x و S_y، در یک مجموعه‌ی جدید تولید می‌کند. فرض بر این است که دو مجموعه قبل از عملیات منفصل هستند. نماینده‌ی مجموعه‌ی حاصل می‌تواند هر عضوی از $S_x \cup S_y$ باشد، با این حال بسیاری از پیاده‌سازی‌های UNION به طور خاص نماینده‌ی S_x یا S_y را به عنوان نماینده‌ی جدید انتخاب می‌کنند. از آن جایی که نیاز داریم مجموعه‌ها در گروه منفصل باشند، مجموعه‌های S_x و S_y را «نابود»، و آن‌ها را از مجموعه‌ی $\mathcal{S}$ حذف می‌کنیم. در عمل معمولاً اعضای یک مجموعه را در مجموعه‌ی دیگر جذب می‌کنیم.
- FIND-SET(x) یک اشاره‌گر به نماینده‌ی مجموعه‌ی (یکتای) حاوی x بازمی‌گرداند.

در طول این فصل زمان اجرای ساختمان‌های داده‌ی مجموعه‌ی منفصل را بر حسب دو پارامتر تحلیل می‌کنیم: n، تعداد اعمال MAKE-SET، و m، کل تعداد اعمال MAKE-SET، UNION، و FIND-SET. از آن جایی که مجموعه‌ها منفصل هستند، هر UNION تعداد مجموعه‌ها را یکی کاهش می‌دهد. بنابراین بعد از $n-1$ عملیات UNION فقط یک مجموعه باقی می‌ماند، و تعداد اعمال UNION حداکثر $n-1$ است. همچنین توجه کنید که چون اعمال MAKE-SET هم در کل تعداد اعمال m محاسبه شده‌اند، داریم $m \geq n$. فرض می‌کنیم که n عملیات MAKE-SET اولین n عملیات اجرا شده هستند.

یک کاربرد از ساختمان داده‌ی مجموعه‌های منفصل

یکی از کاربردهای ساختمان داده‌ی مجموعه‌های منفصل در تعیین مؤلفه‌های همبندی یک گراف بدون جهت است (بخش ب-۴ را ببینید). به عنوان مثال شکل ۲۱-۱(الف) یک گراف با چهار مؤلفه‌ی همبندی را نشان می‌دهد.

رویه‌ی CONNECTED-COMPONENTS که در ادامه می‌آید، از اعمال مجموعه‌های منفصل برای محاسبه‌ی مؤلفه‌های همبندی یک گراف استفاده می‌کند. وقتی CONNECTED-COMPONENTS به عنوان یک مرحله‌ی پردازش اجرا شد، رویه‌ی SAME-COMPONENT پرسش‌هایی را در مورد این که



(الف)

یال پردازش شده	گروه مجموعه‌های منفصل									
مجموعه‌های اولیه	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

(ب)

شکل ۲۱-۲ (الف) یک گراف با چهار مؤلفه‌ی همبندی: $\{a,b,c,d\}$، $\{e,f,g\}$، $\{h,i\}$، و $\{j\}$. (ب) گروه مجموعه‌های منفصل بعد از پردازش بر روی هر یال.

دو رأس در یک مؤلفه‌ی همبندی هستند یا خیر جواب می‌دهد.[1] (مجموعه‌ی رأس‌های گراف G با $G.V$، و مجموعه‌ی یال‌های آن با $G.E$ نشان داده شده است.)

```
CONNECTED-COMPONENTS(G)
1  for each vertex v ∈ G.V
2      MAKE-SET(v)
3  for each edge (u, v) ∈ G.E
4      if FIND-SET(u) ≠ FIND-SET(v)
5          UNION(u, v)

SAME-COMPONENT(u, v)
1  if FIND-SET(u) == FIND-SET(v)
2      return TRUE
3  else return FALSE
```

رویه‌ی CONNECTED-COMPONENTS ابتدا هر رأس را در یک مجموعه قرار می‌دهد. سپس برای هر یال (u,v)، از مجموعه‌های حاوی u و v اجتماع می‌گیرد. طبق تمرین ۲۱-۱-۲، بعد از این

[1] وقتی یال‌های گراف «ایستا»–ثابت در طول زمان–هستند، مؤلفه‌های همبندی را می‌توان با استفاده از جستجوی عمق اول باسرعت بیشتری محاسبه کرد (تمرین ۲۲-۳-۱۲ را ببینید). با این حال بعضی مواقع یال‌ها به صورت «پویا» اضافه می‌شوند و باید مؤلفه‌های همبندی را همزمان با اضافه شدن یال‌ها محاسبه کنیم. در این حالت، پیاده‌سازی داده شده در این جا می‌تواند سریع‌تر از اجرای یک جستجوی عمق اول برای هر یال جدید اجرا شود.

که تمام یال‌ها پردازش شدند، دو رأس در یک مؤلفه‌ی همبندی هستند اگر و تنها اگر اشیای مربوطه در یک مجموعه باشند. بنابراین CONNECTED-COMPONENTS مجموعه‌ها را به صورتی محاسبه می‌کند که رویه‌ی SAME-COMPONENT بتواند تعیین کند کدام دو رأس در یک مؤلفه‌ی همبندی هستند. شکل ۲۱-۱(ب) نشان می‌دهد که مجموعه‌های منفصل چگونه توسط CONNECTED-COMPONENTS محاسبه می‌شوند.

در یک پیاده‌سازی واقعی از این الگوریتم مؤلفه‌ی همبندی، نمایش گراف و ساختمان داده‌ی مجموعه‌های منفصل نیاز دارند که بتوانند به یکدیگر ارجاع دهند. یعنی یک شیئ نماینده‌ی یک رأس حاوی یک اشاره‌گر به شیئ مجموعه‌ی منفصل مربوطه خواهد بود، و بالعکس. این جزئیات برنامه‌نویسی به زبان پیاده‌سازی بستگی دارد، و ما در این جا بیشتر از این به آن‌ها نخواهیم پرداخت.

تمرین‌ها

۲۱-۱-۱ فرض کنید CONNECTED-COMPONENTS بر روی یک گراف بدون جهت $G = (V, E)$ اجرا شده است، که در آن $V = \{a, b, c, d, e, f, g, h, i, j, k\}$ و یال‌های E به ترتیب زیر پردازش می‌شوند: (d, i)، (f, k)، (g, i)، (b, g)، (a, h)، (i, j)، (d, k)، (b, j)، (d, f)، (g, j)، (a, e)، (i, d). رأس‌های درون هر مؤلفه‌ی همبندی را بعد از هر بار اجرای خطوط ۳-۵ لیست کنید.

۲۱-۱-۲ نشان دهید بعد از این که تمام یال‌ها توسط CONNECTED-COMPONENTS پردازش شدند، دو رأس در یک مؤلفه‌ی همبندی هستند اگر و فقط اگر در یک مجموعه باشند.

۲۱-۱-۳ حین اجرای CONNECTED-COMPONENTS بر روی یک گراف بدون جهت $G = (V, E)$ با k مؤلفه‌ی همبندی، FIND-SET چند بار فراخوانی می‌شود؟ UNION چند بار فراخوانی می‌شود؟ جواب‌های خود را برحسب $|V|$، $|E|$، و k ارائه کنید.

۲۱-۲ نمایش لیست پیوندی از مجموعه‌های منفصل

شکل ۲۱-۲(الف) یک راه ساده برای پیاده‌سازی یک ساختمان داده‌ی مجموعه‌های منفصل نشان می‌دهد: نمایش هر مجموعه به وسیله‌ی یک لیست پیوندی. شیئ مربوط به هر مجموعه یک خصیصه‌ی *head* دارد که به اولین عنصر مجموعه اشاره می‌کند، به همراه یک خصیصه‌ی *tail* که به آخرین عنصر اشاره می‌کند. هر شیئ در لیست پیوندی حاوی یک عضو مجموعه، یک اشاره‌گر به عضو بعدی مجموعه، و یک اشاره‌گر عقبی به نماینده است. در هر لیست پیوندی اشیا ممکن است به هر ترتیبی ظاهر شوند. عضو مجموعه در اولین عنصر لیست، نماینده‌ی مجموعه است.

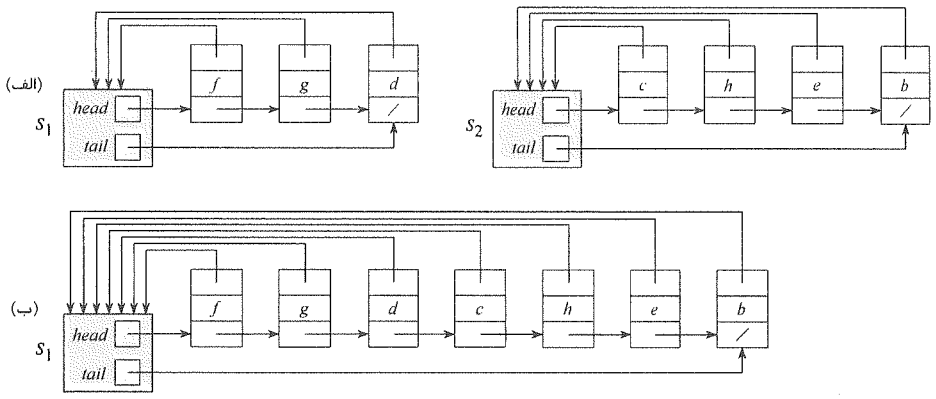
با این نمایش لیست پیوندی، هر دوی اعمال MAKE-SET و FIND-SET ساده هستند و در زمان $O(1)$ اجرا می‌شوند. برای انجام MAKE-SET(x)، یک لیست پیوندی جدید می‌سازیم که تنها

شیئ آن x است. برای FIND-SET(x)، اشاره‌گر درون x که به عنصر مجموعه‌ی آن اشاره می‌کند را دنبال کرده و سپس به شیئی باز می‌گردیم که *head* به آن اشاره می‌کند. برای مثال در شکل ۲۱-۲(الف)، فراخوانی FIND-SET(g) عنصر f را بازمی‌گرداند.

یک پیاده‌سازی ساده از اجتماع

ساده‌ترین پیاده‌سازی عملیات UNION با استفاده از نمایش لیست پیوندی، به شدت زمان بیشتری از MAKE-SET و FIND-SET می‌گیرد. همان طور که شکل ۲۱-۲(ب) نشان می‌دهد، UNION(x, y) را با اتصال لیست x به انتهای لیست y انجام می‌دهیم. نماینده‌ی لیست x تبدیل به نماینده‌ی لیست حاصل می‌شود. از اشاره‌گر *tail* لیست y استفاده می‌کنیم تا به سرعت تشخیص دهیم که لیست x را باید به کجا اضافه کنیم. چون تمام عناصر لیست y به لیست x می‌پیوندند، می‌توانیم شیئ مجموعه‌ی مربوط به لیست y را نابود کنیم. متأسفانه باید برای تمام عناصری که قبلاً در مجموعه‌ی حاوی y بوده‌اند، اشاره‌گر نماینده‌ی مجموعه را به هنگام سازی کنیم، که زمان آن نسبت به تعداد عناصر لیست y خطی است. مثلاً در شکل ۲۱-۲ عملیات UNION(g, e) باعث می‌شود که اشاره‌گر اشیاء b، c، e، و h به هنگام‌سازی شوند.

در واقع اصلاً سخت نیست که دنباله‌ای از m عملیات بر روی n شیئ ارائه کنیم که به زمان $\theta(n^2)$ نیاز داشته باشد. فرض کنید که اشیای $x_1, x_2, ..., x_n$ را داریم. دنباله‌ای از n عملیات



شکل ۲۱-۲ (الف) لیست‌های پیوندی نماینده‌ی دو مجموعه. مجموعه‌ی S_1 حاوی عناصر d، f، و g است با نماینده‌ی f، و مجموعه‌ی S_2 حاوی عناصر b، c، e، و h، با c به عنوان نماینده. هر شیئ درون لیست حاوی یک عضو مجموعه، یک اشاره‌گر به شیئ بعدی در لیست، و یک اشاره‌گر عقبی به اولین شیئ در لیست است، که اولین شیئ همان نماینده‌ی مجموعه می‌باشد. هر لیست شامل اشاره‌گرهای *head* و *tail* به ترتیب به اولین و آخرین عنصر لیست است. (ب) نتیجه‌ی UNION(g, e) که لیست پیوندی حاوی e را به لیست پیوندی حاوی g متصل می‌کند. نماینده‌ی مجموعه‌ی حاصل f است. شیئ مجموعه برای لیست e، یعنی S_2، نابود می‌شود.

عملیات	تعداد اشیای به هنگام سازی شده
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_1, x_2)	1
UNION(x_2, x_3)	2
UNION(x_3, x_4)	3
⋮	⋮
UNION(x_{n-1}, x_n)	$n-1$

شکل ۲۱-۳ دنباله‌ای از $2n-1$ عملیات بر روی n شیئ که با استفاده از نمایش لیست پیوندی و پیاده‌سازی ساده‌ی UNION، در کل به $\theta(n^2)$ زمان، و یا به طور متوسط $\theta(n)$ زمان برای هر عملیات نیاز دارد.

MAKE-SET و پس از آن دنباله‌ای از $n-1$ عملیات UNION را اجرا می‌کنیم، همان طور که در شکل ۲۱-۳ نشان داده شده است، به طوری که داریم $m = 2n-1$. زمان اجرای n عملیات MAKE-SET برابر $\theta(n)$ است. چون iامین عملیات UNION، i شیئ را به هنگام سازی می‌کند، تعداد کل اشیائی که توسط $n-1$ عملیات UNION به هنگام سازی می‌شوند برابر است با

$$\sum_{i=1}^{n-1} i = \theta(n^2)$$

تعداد کل عملیات $2n-1$ است، و بنابراین هر عملیات به طور متوسط به $\theta(n)$ زمان نیاز دارد. یعنی زمان سرشکن هر عملیات $\theta(n)$ است.

یک روش مکاشفه‌ای برای اجتماع وزن‌دار

در بدترین حالت پیاده‌سازی بالا برای رویه‌ی UNION به طور متوسط برای هر فراخوانی به زمان $\theta(n)$ نیاز دارد، چرا که ممکن است یک لیست بلند را به یک لیست کوتاه‌تر متصل کنیم؛ باید اشاره‌گر نماینده را برای هر عضو لیست بلندتر به هنگام سازی کنیم. در عوض فرض کنید که هر لیست حاوی طول لیست نیز هست (که حفظ آن بسیار ساده است) و ما همیشه لیست کوتاه‌تر را به لیست بلندتر متصل می‌کنیم. با این *روش مکاشفه‌ای اجتماع وزن‌دار* (weighted-union heuristic) ساده، در صورتی که هر دو مجموعه $\Omega(n)$ عضو داشته باشند، یک عملیات UNION هنوز می‌تواند $\Omega(n)$ زمان بگیرد. با این حال همان طور که قضیه‌ی زیر نشان می‌دهد، دنباله‌ای از m عملیات MAKE-SET، UNION، و FIND-SET، که n تا از آن‌ها MAKE-SET هستند، به $O(m + n \lg n)$ زمان نیاز دارد.

قضیه ۲۱-۱ با استفاده از نمایش لیست پیوندی مجموعه‌های منفصل و مکاشفه‌ی اجتماع وزن‌دار، دنباله‌ای از m عملیات MAKE-SET، UNION، و FIND-SET، که n تا از آن‌ها MAKE-SET هستند، در زمان $O(m + n \lg n)$ اجرا می‌شود.

اثبات چون هر عملیات UNION دو مجموعه‌ی منفصل را با هم یکی می‌کند، در کل حداثر $n-1$ عملیات انجام می‌دهیم. اکنون کرانی برای کل زمان صرف شده در این اعمال UNION تعیین می‌کنیم. با محاسبه‌ی یک کران بالا برای تعداد دفعاتی که اشاره‌گر نماینده‌ی یک شیئ به هنگام سازی شده است، برای هر شیئ در یک مجموعه با اندازه‌ی n شروع می‌کنیم. یک شیئ x را در نظر بگیرید. می‌دانیم که هر بار اشاره‌گر نماینده‌ی x به هنگام سازی شده است، x در مجموعه‌ی کوچک‌تر بوده است. بنابراین اولین باری که اشاره‌گر نماینده‌ی x به هنگام سازی شده، مجموعه‌ی حاصل حداقل ۲ عضو داشته است. به طور مشابه، دفعه‌ی بعدی که اشاره‌گر x به هنگام سازی شده، مجموعه‌ی حاصل حداقل ۴ عضو داشته است. با ادامه‌ی این روند مشاهده می‌کنیم که برای هر $k \le n$، بعد از این که اشاره‌گر نماینده‌ی x تعداد $\lceil \lg k \rceil$ بار به هنگام سازی شده است، مجموعه‌ی حاصل حداقل k عضو داشته است. از آن جایی که بزرگ‌ترین مجموعه حداکثر n عضو دارد، اشاره‌گر نماینده‌ی هر شیئ در تمام UNION ها حداکثر $\lceil \lg n \rceil$ بار به هنگام سازی شده است. بنابراین کل زمان صرف شده برای به هنگام‌سازی اشاره‌گرهای اشیاء در کل اعمال UNION برابر است با $O(n \lg n)$. باید به هنگام‌سازی اشاره‌گرهای *head* و *tail* را هم در نظر بگیریم، که برای هر عملیات UNION فقط $\theta(1)$ زمان می‌گیرد. بنابراین کل زمان مصرف شده در به هنگام‌سازی n شیئ $O(n \lg n)$ است.

اکنون زمان کل دنباله‌ی m عملیات به سادگی به دست می‌آید. هر عملیات MAKE-SET و FIND-SET، به زمان $O(1)$ نیاز دارد، و $O(m)$ تا از آن‌ها وجود دارد. بنابراین زمان اجرای کل دنباله $O(m + n \lg n)$ است. ■

تمرین‌ها

۲۱-۲-۱ با استفاده از نمایش لیست پیوندی و مکاشفه‌ی اجتماع وزن‌دار برای MAKE-SET، FIND-SET و UNION شبه‌کد بنویسید. حتماً خصیصه‌هایی را که برای اشیاء مجموعه‌ها و اشیاء لیست‌ها در نظر می‌گیرید، مشخص کنید.

۲۱-۲-۲ ساختمان داده و جواب‌های حاصل از فراخوانی عملیات FIND-SET را در برنامه‌ی زیر نشان دهید. از نمایش لیست پیوندی و مکاشفه‌ی اجتماع وزن‌دار استفاده کنید.

```
1  for i = 1 to 16
2      MAKE-SET(x_i)
3  for i = 1 to 15 by 2
4      UNION(x_i, x_{i+1})
5  for i = 1 to 13 by 4
6      UNION(x_i, x_{i+2})
7  UNION(x_1, x_5)
8  UNION(x_11, x_13)
9  UNION(x_1, x_10)
10 FIND-SET(x_2)
11 FIND-SET(x_9)
```

فرض کنید اگر مجموعه‌های حاوی x_i و x_j دارای اندازه‌ی یکسانی باشند، آن گاه عملیات UNION(x_i, x_j)، لیست x_i را به لیست x_j متصل می‌کند.

۲۱-۲-۳ اثبات متراکم قضیه‌ی ۲۱-۱ را طوری تکمیل کنید که با استفاده از نمایش لیست پیوندی و مکاشفه‌ی اجتماع وزن‌دار، به کران‌های زمانی سرشکن $O(1)$ برای MAKE-SET و FIND-SET، و $O(\lg n)$ برای UNION برسد.

۲۱-۲-۴ با فرض استفاده از نمایش لیست پیوندی و مکاشفه‌ی اجتماع وزن‌دار، یک کران حدی نزدیک برای زمان اجرای دنباله‌ی عملیات نشان داده شده در شکل ۲۱-۳ بدهید.

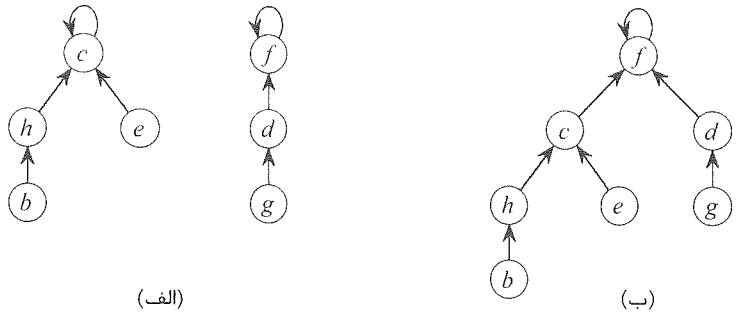
۲۱-۲-۵ پروفسور Gompers فکر می‌کند که احتمالاً می‌توان به جای دو اشاره‌گر (*head* و *tail*)، فقط یک اشاره‌گر در هر شیئ مجموعه نگه داشت، در حالی که تعداد اشاره‌گرهای هر شیئ لیست همان دو باقی بماند. نشان دهید که شک پروفسور به جا است. توضیح دهید که چطور می‌توان هر مجموعه را با یک لیست پیوندی نمایش داد به طوری که زمان اجرای اعمال توصیف شده در این بخش تغییر نکند. همچنین توضیح دهید که این اعمال چگونه کار می‌کنند. مکاشفه‌ی اجتماع وزن‌دار هم باید در رویکرد شما امکان‌پذیر باشد، با همان تأثیری که در این بخش توصیف شد. (*راهنمایی:* از انتهای لیست پیوندی به عنوان نماینده‌ی مجموعه استفاده کنید.)

۲۱-۲-۶ یک تغییر ساده برای رویه‌ی UNION برای نمایش لیست پیوندی پیشنهاد دهید که در آن نیازی به نگه داشتن اشاره‌گر *tail* به آخرین عنصر لیست در هر شیئ نباشد. تغییر شما نباید زمان اجرای حدی عملیات UNION را تغییر دهد، چه در آن از مکاشفه‌ی وزن‌دار استفاده شود و چه نشود. (*راهنمایی:* به جای اتصال یک لیست به دیگری، آن‌ها را به هم «گره بزنید» (splice).)

۲۱-۳ جنگل‌های مجموعه‌های منفصل

در یک پیاده‌سازی سریع‌تر از مجموعه‌های منفصل، مجموعه‌ها را با درختان ریشه‌دار نمایش می‌دهیم که در آن هر گره حاوی یک عضو است و هر درخت نشان دهنده‌ی یک مجموعه. در یک *جنگل مجموعه‌های منفصل* (disjoint-set forest)، که در شکل ۲۱-۴(الف) نشان داده شده است، هر عضو فقط به پدر خود اشاره می‌کند. همان طور که خواهیم دید با این که الگوریتم‌های سرراستی که از این نمایش استفاده می‌کنند سریع‌تر از آن‌هایی نیستند که در نمایش لیست پیوندی از آن‌ها استفاده شد، با معرفی چند مکاشفه – «اجتماع بر حسب رتبه» و «تراکم مسیر» – می‌توانیم به سریع‌ترین ساختمان داده‌ی مجموعه‌های منفصل به صورت حدی دست یابیم.

عملیات درختان مجموعه‌های منفصل را به صورت زیر انجام می‌دهیم. عملیات MAKE-SET به



شکل ۲۱-۴ یک جنگل مجموعه‌های منفصل. (الف) دو درخت نشان دهنده‌ی دو مجموعه‌ی شکل ۲۱-۲. درخت سمت چپ نشان دهنده‌ی مجموعه‌ی $\{b, c, e, h\}$، و درخت سمت راست نشان دهنده‌ی مجموعه‌ی $\{d, f, g\}$ است، که در آن‌ها به ترتیب c و f نماینده‌ی مجموعه‌ها هستند. (ب) نتیجه‌ی UNION(e, g).

سادگی یک درخت با یک گره تولید می‌کند. عملیات FIND-SET را با دنبال کردن اشاره‌گرهای پدر انجام می‌دهیم، تا جایی که به ریشه‌ی درخت برسیم. گره‌های ملاقات شده در این مسیر ساده، *مسیر ریشه* را تشکیل می‌دهند. یک عملیات UNION، که در شکل ۲۱-۴(ب) نشان داده شده است، باعث می‌شود که ریشه‌ی یک درخت به ریشه‌ی دیگری اشاره کند.

مکاشفه‌هایی برای بهبود زمان اجرا

تا به این جا پیاده‌سازی لیست پیوندی را بهبودی نبخشیده‌ایم. دنباله‌ای از $n-1$ عملیات UNION ممکن است درختی تولید کند که فقط زنجیره‌ای خطی از n گره باشد. با این حال با استفاده از دو مکاشفه می‌توانیم به زمان اجرایی برسیم که تقریباً نسبت به m، تعداد کل اعمال انجام شده خطی است.

مکاشفه‌ی اول، *اجتماع بر حسب رتبه* (union by rank) مانند مکاشفه‌ی اجتماع وزن‌دار است که در نمایش لیست پیوندی از آن استفاده کردیم. ایده‌ی بدیهی این است که کاری کنیم ریشه‌ی درختی که گره‌های کم‌تری دارد به ریشه‌ی درختی که گره‌های بیشتری دارد اشاره کند. به جای این که صریحاً اندازه‌ی زیردرخت هر گره را نگه داریم، از رویکردی استفاده می‌کنیم که تحلیل را آسان می‌کند. برای هر گره، یک *رتبه* (rank) نگه می‌داریم که کران بالای ارتفاع گره است. در اجتماع بر حسب رتبه، ریشه‌ی با رتبه‌ی پایین‌تر به ریشه‌ی با رتبه‌ی بالاتر اشاره خواهد کرد.

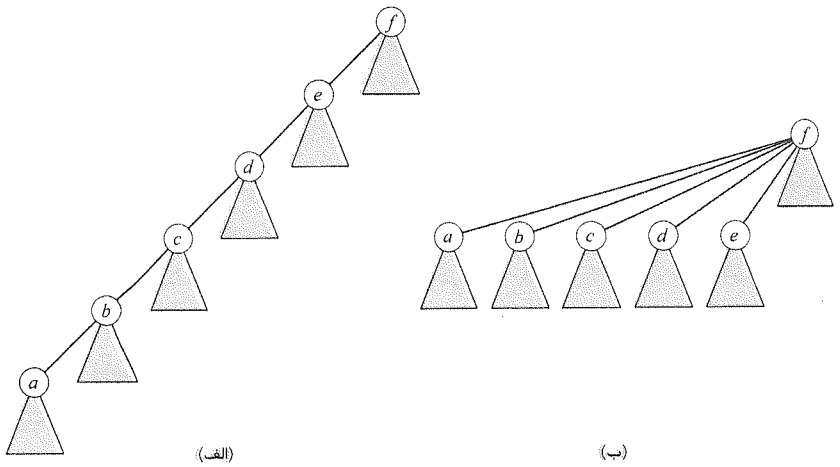
مکاشفه‌ی بعدی، *تراکم مسیر* (path compression)، هم ساده است و هم مؤثر. همان طور که در شکل ۲۱-۵ نشان داده شده است، از آن در عملیات FIND-SET استفاده می‌کنیم تا کاری کنیم که هر گره‌ی در مسیر ریشه مستقیماً به ریشه اشاره کند. تراکم مسیر هیچ رتبه‌ای را تغییر نمی‌دهد.

شبه‌کد برای جنگل‌های مجموعه‌های منفصل

برای پیاده‌سازی یک جنگل از مجموعه‌های منفصل با مکاشفه‌ی اجتماع بر حسب رتبه، باید رتبه‌ها را به درستی حفظ کنیم. با هر گره‌ی x، یک مقدار صحیح $x.rank$ نگه می‌داریم که کران بالایی است بر روی ارتفاع x (تعداد گره‌ها در طولانی‌ترین مسیر از x به یک برگ در نوادگان x). وقتی یک مجموعه‌ی منحصر به فرد توسط MAKE-SET ساخته می‌شود، رتبه‌ی اولیه‌ی تنها گره در درخت مربوطه ۰ است. عملیات FIND-SET تمام رتبه‌ها را بدون تغییر باقی می‌گذارد. وقتی UNION را بر روی دو درخت اعمال می‌کنیم، دو حالت پیش می‌آید، بسته به این که آیا رتبه‌ی ریشه‌ها برابر است یا خیر. اگر رتبه‌ی ریشه‌ها برابر نباشد، ریشه‌ی با رتبه‌ی بالاتر را به عنوان پدر ریشه‌ی با رتبه‌ی پایین‌تر قرار می‌دهیم، ولی خود رتبه‌ها بدون تغییر باقی می‌مانند. در عوض اگر رتبه‌ی ریشه‌ها برابر باشد، به صورت دلخواه یکی از ریشه‌ها را به عنوان پدر انتخاب می‌کنیم و رتبه‌ی آن را یکی افزایش می‌دهیم.

اجازه دهید این متد را به صورت شبه‌کد درآوریم. پدر گره‌ی x را با $x.p$ مشخص می‌کنیم. رویه‌ی LIKN، زیرروالی که توسط UNION فراخوانی می‌شود، اشاره‌گرهایی به دو ریشه را به عنوان ورودی دریافت می‌کند.

```
MAKE-SET(x)
1  x.p = x
2  x.rank = 0

UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))
```



شکل ۲۱-۵ تراکم مسیر هنگام عملیات FIND-SET. از پیکان‌ها و طوقه‌ها در ریشه‌ها صرف نظر شده است. (الف) یک درخت نشان‌دهنده‌ی یک مجموعه قبل از اجرای FIND-SET. مثلث‌ها نشان‌دهنده‌ی زیردرخت‌هایی هستند که ریشه‌ی آن‌ها، گره‌های نشان داده شده است. هر گره یک اشاره‌گر به پدر خود دارد. (ب) همان مجموعه بعد از اجرای FIND-SET. اکنون هر گره در مسیر ریشه مستقیماً به ریشه اشاره می‌کند.

```
LINK(x, y)
1  if x.rank > y.rank
2      y.p = x
3  else x.p = y
4      if x.rank = y.rank
5          y.rank = y.rank + 1
```

رویه‌ی FIND-SET به همراه تراکم مسیر کاملاً ساده است:

```
FIND-SET(x)
1  if x ≠ x.p
2      x.p = FIND-SET(x.p)
3  return x.p
```

رویه‌ی FIND-SET یک متد دو جهته است: این رویه ابتدا یک مسیر به سمت بالا را طی می‌کند تا ریشه را بیابد، و سپس در مسیر دوم به سمت پایین اشاره‌گرهای پدر گره‌ها را طوری به هنگام سازی می‌کند که به ریشه اشاره کنند. هر فراخوانی FIND-SET (x) در خط ۳ مقدار $x.p$ را بازمی‌گرداند. اگر x ریشه باشد، آن گاه خط ۲ اجرا نمی‌شود و $x.p$ بازگردانده می‌شود، که همان x است؛ این حالتی است که در آن بازگشت پایان می‌یابد. در غیر این صورت خط ۲ اجرا شده و فراخوانی بازگشتی روی $x.p$، یک اشاره‌گر به ریشه بازمی‌گرداند. خط ۲ گره‌ی x را طوری به هنگام سازی می‌کند که مستقیماً به ریشه اشاره کند، و این اشاره‌گر در خط ۳ بازگردانده می‌شود.

تأثیر مکاشفه‌ها بر روی زمان اجرا

اجتماع بر حسب رتبه و تراکم مسیر، هر کدام به طور جداگانه زمان اجرای اعمال بر روی جنگل‌های مجموعه‌های منفصل را بهبود می‌بخشند، و این بهبودها زمانی که از این دو مکاشفه به همراه هم استفاده می‌شود بیشتر هم می‌شود. اجتماع بر حسب رتبه به تنهایی به زمان اجرای $O(m \lg n)$ منجر می‌شود (تمرین ۲۱-۴-۴ را ببینید)، و این کران نزدیک است (تمرین ۲۱-۳-۳ را ببینید). با این که این را در این جا اثبات نمی‌کنیم، اگر n عملیات MAKE-SET وجود داشته باشد (و بنابراین حداکثر $n-1$ عملیات UNION) و f عملیات FIND-SET، مکاشفه‌ی تراکم مسیر به تنهایی بدترین حالت زمان اجرای $\theta(n + f \cdot (1+\log_{2+f/n} n))$ را به ما می‌دهد.

وقتی از هر دوی اجتماع بر حسب رتبه و تراکم مسیر استفاده می‌کنیم، بدترین حالت زمان اجرا $O(m\alpha(n))$ است، که در آن $\alpha(n)$ یک تابع با سرعت رشد بسیار کند است، که آن را در بخش ۲۱-۴ تعریف می‌کنیم. در هر کاربرد ممکن از یک ساختمان داده‌ی مجموعه‌های منفصل، داریم $\alpha(n) \le 4$؛ بنابراین می‌توانیم زمان اجرا را در تمام موقعیت‌های کاربردی به صورت خطی نسبت به m ببینیم. هر چند اگر بخواهیم دقیق باشیم، این زمان فرا-خطی است. در بخش ۲۱-۴ این کران بالا را اثبات می‌کنیم.

تمرین‌ها

۲۱-۳-۱ تمرین ۲۱-۲-۲ را با استفاده از جنگل مجموعه‌های منفصل با اجتماع بر حسب رتبه و تراکم مسیر دوباره انجام دهید.

۲۱-۳-۲ یک نسخه‌ی غیر بازگشتی از FIND-SET را با تراکم مسیر بنویسید.

۲۱-۳-۳ دنباله‌ای از m عملیات MAKE-SET، UNION، و FIND-SET بدهید، که n تا از آن‌ها عملیات MAKE-SET هستند، به طوری که این دنباله فقط با استفاده از اجتماع بر حسب رتبه به $\Omega(m \lg n)$ زمان نیاز داشته باشد.

۲۱-۳-۴ فرض کنید می‌خواهیم عملیات PRINT-SET (x) را اضافه کنیم، که یک گره‌ی x را دریافت کرده و تمام اعضای مجموعه‌ی x را چاپ می‌کند (ترتیب چاپ کردن مهم نیست). نشان دهید که چطور می‌توانیم فقط با اضافه کردن یک خصیصه در هر گره‌ی جنگل مجموعه‌های منفصل، کاری کنیم که PRINT-SET (x) در زمان خطی نسبت به تعداد اعضای مجموعه‌ی x اجرا شود، در حالی که مرتبه‌ی زمان اجرای اعمال دیگر تغییر نمی‌کند. فرض کنید می‌توانیم هر عضو مجموعه را در زمان $O(1)$ چاپ کنیم.

★ ۲۱-۳-۵ نشان دهید که هر دنباله‌ای از m عملیات MAKE-SET، UNION، و FIND-SET، که تمام اعمال LINK قبل از اعمال FIND-SET رخ می‌دهند، در صورتی که از هر دوی تراکم مسیر و اجتماع بر حسب رتبه استفاده شود، فقط به زمان $O(m)$ نیاز دارد. در موقعیتی مشابه اگر فقط از مکاشفه‌ی تراکم مسیر استفاده شود، چه اتفاقی می‌افتد؟

★۲۱-۴ تحلیل اجتماع بر حسب رتبه به همراه تراکم مسیر

همان طور که در بخش ۲۱-۳ گفته شد، زمان اجرای ترکیب اجتماع بر حسب رتبه و تراکم مسیر برای m عملیات مجموعه‌های منفصل بر روی n عنصر، $O(m\alpha(n))$ است. در این بخش تابع α را بررسی می‌کنیم تا ببینیم که چقدر کند رشد می‌کند. سپس این زمان اجرا را با استفاده از متد پتانسیل تحلیل سرشکن اثبات می‌کنیم.

یک تابع با سرعت رشد بسیار زیاد و معکوس آن با سرعت رشد بسیار کم

برای اعداد صحیح $k \geq 0$ و $j \geq 1$، تابع $A_k(j)$ را به صورت زیر تعریف می‌کنیم:

$$A_k(j) = \begin{cases} j+1 & \text{اگر } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{اگر } k \geq 1 \end{cases}$$

که در آن عبارت $A_{k-1}^{(j+1)}(j)$ از نماد تکرار توابع داده شده در بخش ۳-۲ استفاده می‌کند. به

خصوص، $A_{k-1}^{(0)}(j) = j$ و $A_{k-1}^{(j)}(j) = A_{k-1}(A_{k-1}^{(j-1)}(j))$ برای $j \geq 1$. از پارامتر k به عنوان *سطح* تابع A یاد خواهیم کرد.

تابع $A_k(j)$ با هر دوی j و k اکیداً افزایش می‌یابد. برای این که ببینیم این تابع با چه سرعتی رشد می‌کند، ابتدا فرم‌های مشخصی برای $A_1(j)$ و $A_2(j)$ به دست می‌آوریم.

لم ۲-۲۸ برای هر عدد صحیح $j \geq 1$ داریم $A_1(j) = 2j + 1$.

اثبات ابتدا از استقرا بر روی i استفاده می‌کنیم تا نشان دهیم $A_0^{(i)}(j) = j + 1$. برای حالت پایه داریم $A_0^{(0)}(j) = j = j + 0$. برای گام استقرا، فرض کنید که $A_0^{(i-1)}(j) = j + (i-1)$. آن گاه $A_0^{(i)} = A_0(A_0^{(i-1)}(j)) = (j + (i-1)) + 1 = j + i$. نهایتاً توجه می‌کنیم که $A_1(j) = A_0^{(j+1)}(j) = j + (j+1) = 2j + 1$. ■

لم ۲-۲۹ برای هر عدد صحیح $j \geq 1$ داریم $A_2(j) = 2^{j+1}(j+1) - 1$.

اثبات ابتدا از استقرا بر روی i استفاده می‌کنیم تا نشان دهیم $A_1^{(i)}(j) = 2^j(j+1) - 1$. برای حالت پایه داریم $A_1^{(0)}(j) = j = 2^0(j+1) - 1$. برای گام استقرا، فرض کنید که $A_1^{(i-1)}(j) = 2^{j-1}(j+1) - 1$. آن گاه:

$$A_1^{(i)}(j) = A_1(A_1^{(i-1)}(j)) = A_1(2^{i-1}(j+1) - 1) = 2 \cdot (2^{i-1}(j+1) - 1) + 1 = 2^i(j+1) - 2 + 1 = 2^i(j+1) - 1$$

نهایتاً توجه می‌کنیم که $A_2(j) = A_1^{(j+1)}(j) = 2^{j+1}(j+1) - 1$. ■

اکنون با بررسی $A_k(1)$ برای سطوح $k = 0, 1, 2, 3, 4$ می‌توانیم به سادگی ببینیم که $A_k(j)$ با چه سرعتی رشد می‌کند. از تعریف $A_0(k)$ و لم‌های بالا، داریم $A_0(1) = 1 + 1 = 2$، $A_1(1) = 2 \times 1 + 1 = 3$، و $A_2(1) = 2^{1+1}.(1+1) - 1 = 7$. همچنین داریم

$$\begin{aligned} A_3(1) &= A_2^{(2)}(1) \\ &= A_2(A_2(1)) \\ &= A_2(7) \\ &= 2^8 \times 8 - 1 \\ &= 2^{11} - 1 \\ &= 2047 \end{aligned}$$

و

$$\begin{aligned}
A_۴(۱) &= A_۳^{(۲)}(۱) \\
&= A_۳(A_۳(۱)) \\
&= A_۳(۲۰۴۷) \\
&= A_۲^{(۲۰۴۸)}(۲۰۴۷) \\
&\gg A_۲(۲۰۴۷) \\
&= ۲^{۲۰۴۸\cdot۲۰۴۸-۱} \\
&> ۲^{۲۰۴۸} \\
&= (۲^۴)^{۵۱۲} \\
&= ۱۶^{۵۱۲} \\
&\gg ۱۰^{۸۰}
\end{aligned}$$

که تعداد تخمین زده شده‌ی اتم‌ها در تمام دنیای شناخته شده است. (نماد "$\gg$" به معنی «بسیار بزرگ‌تر از» می‌باشد.)

برای عدد صحیح $n \geq ۰$، معکوس تابع $A_k(n)$ را به صورت

$$\alpha(n) = \min\{k : A_k(۱) = n\}$$

تعریف می‌کنیم. به صورت توضیحی $\alpha(n)$ پایین‌ترین سطح k است که برای آن $A_k(۱)$ حداقل n است. از مقادیر بالا از $A_k(۱)$، می‌بینیم که

$$\alpha(n) = \begin{cases} ۰ & \text{برای } ۰ \leq n \leq ۲ \\ ۱ & \text{برای } n = ۳ \\ ۲ & \text{برای } ۴ \leq n \leq ۷ \\ ۳ & \text{برای } ۸ \leq n \leq ۲۰۴۷ \\ ۴ & \text{برای } ۲۰۴۷ \leq n \leq A_۴(۱) \end{cases}$$

فقط برای اعداد به صورت غیر طبیعی بزرگ n، که می‌توان واژه‌ی «نجومی» را برای آن‌ها به کار برد (بزرگ‌تر از $A_۴(۱)$، یک عدد بسیار بزرگ)، داریم $\alpha(n) > ۴$، و بنابراین برای تمام اهداف کاربردی $\alpha(n) \leq ۴$.

خصوصیات رتبه‌ها

در ادامه‌ی این بخش اثبات خواهیم کرد که یک کران $O(m\alpha(n))$ بر روی زمان اجرای اعمال مجموعه‌های منفصل با اجتماع بر حسب رتبه و تراکم مسیر وجود دارد. برای اثبات این کران، ابتدا بعضی از خصوصیات ساده‌ی رتبه‌ها را اثبات می‌کنیم.

لم ۲-۲۱ برای تمام گره‌های x، داریم $x.rank \leq x.p.rank$، با نامساوی اکید $x \neq x.p$. مقدار $x.rank$ در ابتدا ۰ است، و در طول زمان تا وقتی که $x \neq x.p$ رشد می‌کند؛ از آن به بعد $x.rank$ تغییر نمی‌کند. مقدار $x.p.rank$ در طول زمان به صورت یکنواخت رشد می‌کند.

اثبات اثبات یک استقرای سرراست بر روی تعداد اعمال است، با استفاده از پیاده‌سازی‌های MAKE-SET، UNION، و FIND-SET که در بخش ۲۱-۳ ارائه شد. این اثبات را به عنوان تمرین ۲۱-۴-۱ واگذار می‌کنیم. ■

نتیجه‌ی ۵-۲۱ وقتی یک مسیر از یک گره تا ریشه را دنبال می‌کنیم، رتبه‌ی گره‌ها به صورت اکید رشد می‌کند.

لم ۳-۲۱ رتبه‌ی هر گره حداکثر $n-1$ است.

اثبات رتبه‌ی هر گره با ۰ شروع می‌شود، و فقط با عملیات LINK افزایش می‌یابد. از آن جایی که حداکثر $n-1$ عملیات UNION انجام خواهد شد، حداکثر $n-1$ عملیات LINK خواهیم داشت. چون هر عملیات LINK یا تمام رتبه‌ها را بدون تغییر باقی می‌گذارد و یا بعضی از آن‌ها را یکی افزایش می‌دهد، تمام رتبه‌ها حداکثر $n-1$ هستند. ■

لم ۲۱-۶ یک کران ضعیف بر روی رتبه‌ها فراهم می‌کند. در واقع رتبه‌ی هر گره حداکثر $\lfloor \lg n \rfloor$ است (تمرین ۲۱-۴-۲ را ببینید). با این حال این کران ضعیف لم ۲۱-۶ برای اهداف ما کافی است.

اثبات کران زمانی

برای اثبات کران زمانی $O(m\alpha(n))$ از متد پتاسیل تحلیل سرشکن (بخش ۱۷-۳ را ببینید) استفاده خواهیم کرد. در انجام تحلیل سرشکن، ساده‌تر است که به جای عملیات UNION فرض کنیم که عملیات LINK انجام شده است. یعنی از آن جایی که پارامترهای رویه‌ی LINK اشاره‌گرهایی به دو ریشه هستند، فرض خواهیم کرد که اعمال FIND-SET مناسب به صورت جداگانه اجرا خواهند شد. لم زیر نشان می‌دهد که حتی اگر اعمال FIND-SET اضافی را که توسط فراخوانی‌های UNION احضار می‌شوند، حساب کنیم، باز هم زمان اجرای حدی بدون تغییر باقی می‌ماند.

لم ۲۱-۷ فرض کنید دنباله‌ی S' از m' عملیات MAKE-SET، UNION، و FIND-SET را به دنباله‌ی S از m عملیات MAKE-SET، LINK، و FIND-SET تبدیل می‌کنیم، بدین صورت که هر عملیات UNION را با دو عملیات FIND-SET و بعد از آن یک عملیات LINK جایگزین می‌کنیم. آن گاه اگر دنباله‌ی S در زمان $O(m\alpha(n))$ اجرا شود، دنباله‌ی S' در زمان $O(m'\alpha(n))$ اجرا خواهد شد.

اثبات چون هر عملیات UNION در دنباله‌ی S' به سه عملیات در S تبدیل می‌شود، داریم $m \le m' \le 3m$. از آن جایی که $m = O(m')$، یک کران زمانی $O(m\alpha(n))$ برای دنباله‌ی تبدیل شده‌ی S، یک کران زمانی $O(m'\alpha(n))$ را برای دنباله‌ی اصلی S' نتیجه می‌دهد. ▩

در ادامه‌ی این بخش فرض خواهیم کرد که دنباله‌ی اصلی متشکل از m' عملیات MAKE-SET، UNION، و FIND-SET به یک دنباله از m عملیات MAKE-SET، LINK، و FIND-SET تبدیل شده است. اکنون یک کران زمانی $O(m\alpha(n))$ را برای دنباله‌ی تبدیل شده اثبات می‌کنیم، و همچنین با استفاده از لم ۲۱-۷، زمان اجرای $O(m'\alpha(n))$ را برای دنباله‌ی اصلی از m' عملیات اثبات می‌کنیم.

تابع پتانسیل

تابع پتانسیلی که در این جا از آن استفاده خواهیم کرد، بعد از q عملیات پتانسیل $\varphi_q(x)$ را به هر گره‌ی x در جنگل مجموعه‌های منفصل نسبت می‌دهد. برای محاسبه‌ی پتاسیل کل جنگل، پتانسیل تک تک گره‌ها را با هم جمع می‌کنیم: $\Phi_q = \Sigma_x \varphi(x)$، که در آن Φ_q نشان‌دهنده‌ی پتانسیل جنگل بعد از q عملیات است. جنگل قبل از انجام اولین عملیات تهی است، و به صورت دلخواه قرار می‌دهیم $\Phi_0 = 0$. هیچ وقت پتانسیل Φ_q منفی نخواهد بود.

مقدار $\varphi_q(x)$ به این بستگی دارد که آیا x بعد از انجام qامین عملیات یک ریشه است یا خیر. اگر این طور نباشد، یا اگر $x.rank = 0$، آن گاه $\varphi_q(x) = \alpha(n) \cdot x.rank$.

اکنون فرض کنید که بعد از qامین عملیات، x ریشه نیست و $x.rank \ge 1$. قبل از تعریف $\varphi_q(x)$ نیاز داریم دو تابع کمکی بر روی x تعریف کنیم. ابتدا تعریف می‌کنیم

$$level(x) = \max\{k : x.p.rank \ge A_k(x.rank)\}$$

یعنی $level(x)$ بزرگ‌ترین سطحی از k است که در آن تابع A_k که برای رتبه‌ی x به کار برده شده است، بزرگ‌تر از رتبه‌ی پدر x نیست.

ادعا می‌کنیم که

$$0 \le level(x) < \alpha(n) \qquad (۱-۲۱)$$

که آن را به صورت زیر خواهیم دید. داریم

$$\begin{aligned} x.p.rank &\geq x.rank + ۱ && \text{(طبق لم ۲۱-۴)} \\ &= A_۰(x.rank) && (A_۰(j) \text{ طبق تعریف}) \end{aligned}$$

که ایجاب می‌کند $level(x) \geq ۰$، و داریم

$$\begin{aligned} A_{\alpha(n)}(x.rank) &\geq A_{\alpha(n)}(۱) && (A_k(j) \text{ چون صعودی اکید است}) \\ &\geq n && (\alpha(n) \text{ طبق تعریف}) \\ &> x.p.rank && \text{(طبق لم ۲۱-۶)} \end{aligned}$$

که ایجاب می‌کند $level(x) < \alpha(n)$. توجه کنید که چون $x.p.rank$ در طول زمان به صورت یکنواخت رشد می‌کند، $level(x)$ هم رشد خواهد کرد.

تابع کمکی دوم زمانی به کار می‌رود که $x.rank \geq ۱$:

$$iter(x) = \max\{i : x.p.rank \geq A^{(i)}_{level(x)}(x.rank)\}$$

یعنی $iter(x)$ بیشترین تعدادی است که ما می‌توانیم با شروع از رتبه‌ی x، $A_{level(x)}$ را مکرراً اعمال کنیم، قبل از این که به عددی بزرگ‌تر از رتبه‌ی پدر x برسیم.

ادعا می‌کنیم که وقتی $x.rank \geq ۱$ داریم:

$$۱ \leq iter(x) \leq x.rank \qquad (۲۱-۲)$$

که آن را به صورت زیر خواهیم دید. داریم

$$\begin{aligned} x.p.rank &\geq A_{level(x)}(x.rank) && (level(x) \text{ طبق تعریف}) \\ &= A^{(۱)}_{level(x)}(x.rank) && \text{(طبق تعریف بازگشت تابعی)} \end{aligned}$$

که ایجاب می‌کند $iter(x) \geq ۱$، و داریم

$$\begin{aligned} A^{(x.rank+۱)}_{level(x)}(x.rank) &= A_{level(x)+۱}(x.rank) && (A_k(j) \text{ طبق تعریف}) \\ &> x.p.rank && (level(x) \text{ طبق تعریف}) \end{aligned}$$

که ایجاب می‌کند $iter(x) \leq x.rank$. توجه کنید که چون $x.p.rank$ به صورت یکنواخت در طول زمان رشد می‌کند، برای این که $iter(x)$ کاهش یابد، $level(x)$ باید افزایش یابد. تا وقتی که $level(x)$ بدون تغییر باقی بماند، $iter(x)$ باید یا افزایش یابد و یا بدون تغییر باقی بماند.

با تعریف این دو تابع کمکی، آماده هستیم که تابع پتانسیل گره‌ی x را بعد از انجام qامین عملیات تعریف کنیم:

$$\varphi_q(x) = \begin{cases} \alpha(n) \cdot x.rank & \text{اگر } x \text{ ریشه باشد یا } x.rank = ۰ \\ (\alpha(n) - level(x)) \cdot x.rank - iter(x) & \text{اگر } x \text{ ریشه نباشد و } x.rank \geq ۱ \end{cases}$$

دو لم بعد خصوصیات مفیدی از پتانسیل گره‌ها به دست می‌دهند.

لم ۲۱-۸ برای هر گره‌ی x و برای تمام اعمال با شماره‌ی q داریم

$$۰ \leq \varphi_q(x) \leq \alpha(n) \cdot x.rank$$

اثبات اگر x یک ریشه باشد و یا $x.rank = 0$، آن گاه طبق تعریف $\varphi_q(x) = \alpha(n) \cdot x.rank$. اکنون فرض کنید x ریشه نباشد و $x.rank \geq 1$. با بیشینه کردن $level(x)$ و $iter(x)$ یک کران پایین برای $\varphi_q(x)$ به دست می‌آوریم. طبق کران (۲۱-۱) داریم $level(x) \leq \alpha(n) - 1$، و طبق کران (۲۱-۲) داریم $iter(x) \leq rank[x]$. بنابراین،

$$\begin{aligned}\varphi_q(x) &= (\alpha(n) - level(x)) \cdot x.rank - iter(x) \\ &\geq (\alpha(n) - (\alpha(n) - 1)) \cdot x.rank - x.rank \\ &= x.rank - x.rank \\ &= 0\end{aligned}$$

به طور مشابه با کمینه کردن $level(x)$ و $iter(x)$ یک کران بالا برای $\varphi_q(x)$ به دست می‌آوریم. طبق کران (۲۱-۱) داریم $level(x) \geq 0$، و طبق کران (۲۱-۲) داریم $iter(x) \geq 1$. بنابراین،

$$\begin{aligned}\varphi(x) &\leq (\alpha(n) - 0) \cdot x.rank - 1 \\ &= \alpha(n) \cdot x.rank - 1 \\ &< \alpha(n) \cdot x.rank\end{aligned}$$

■

نتیجه‌ی ۲۱-۹ اگر گره‌ی x ریشه نباشد و $x.rank > 0$، آن گاه $\varphi_q(x) < \alpha(n) \cdot x.rank$.

■

تغییرات پتانسیل و هزینه‌های سرشکن اعمال

اکنون آماده هستیم بررسی کنیم که اعمال مجموعه‌های منفصل چگونه بر روی پتانسیل گره‌ها تأثیر می‌گذارند. با دانستن تغییرات پتانسیل برای هر عملیات می‌توانیم هزینه‌ی سرشکن آن عملیات را به دست آوریم.

لم ۲۱-۱۰ فرض کنید x نشان‌دهنده‌ی یک گره‌ی غیر ریشه باشد، و همچنین فرض کنید که qامین عملیات نه LINK است و نه FIND-SET. آن گاه بعد از qامین عملیات داریم $\varphi_q(x) \leq \varphi_{q-1}(x)$. به علاوه اگر $x.rank \geq 1$، و هر کدام از $level(x)$ یا $iter(x)$ با عملیات qام تغییر کنند، آن گاه $\varphi_q(x) \leq \varphi_{q-1}(x) - 1$. یعنی پتانسیل x نمی‌تواند افزایش یابد، و اگر x رتبه‌ی مثبت داشته باشد و هر کدام از $iter(x)$ یا $level(x)$ تغییر کنند، آن گاه پتانسیل x حداقل یکی کاهش می‌یابد.

اثبات چون x ریشه نیست، qامین عملیات $x.rank$ را تغییر نمی‌دهد، و چون n بعد از n عملیات اول MAKE-SET تغییر نمی‌کند، $\alpha(n)$ هم بدون تغییر باقی می‌ماند. پس این عناصر در پتانسیل x بعد

از qامین عملیات بدون تغییر باقی می‌مانند. اگر $x.rank = 0$، آن گاه $\varphi_q(x) = \varphi_{q-1}(x) = 0$. اکنون فرض کنید که $x.rank \geq 1$.

به خاطر بیاورید که $level(x)$ در طول زمان به صورت یکنواخت افزایش می‌یابد. اگر qامین عملیات $level(x)$ را بدون تغییر باقی بگذارد، آن گاه $iter(x)$ یا افزایش می‌یابد و یا بدون تغییر باقی می‌ماند. اگر هر دوی $level(x)$ و $iter(x)$ تغییر نکنند، آن گاه $\varphi_q(x) = \varphi_{q-1}(x)$. اگر $level(x)$ تغییر نکند و $iter(x)$ افزایش یابد، آن گاه این افزایش حداقل یکی خواهد بود، و بنابراین $\varphi_q(x) \leq \varphi_{q-1}(x) - 1$.

نهایتاً اگر عملیات qام $level(x)$ را افزایش دهد، این افزایش حداقل یکی خواهد بود، و بنابراین مقدار عبارت $x.rank \cdot (\alpha(n) - level(x))$ حداقل به اندازه‌ی $x.rank$ افت خواهد کرد. چون $level(x)$ افزایش می‌یابد، ممکن است مقدار $iter(x)$ کاهش یابد، ولی طبق کران (۲۱-۲) این کاهش حداکثر به اندازه‌ی $x.rank - 1$ است. بنابراین افزایش در پتانسیل به خاطر تغییر در $iter(x)$ کم‌تر است از کاهش در پتانسیل به خاطر تغییر $level(x)$، و نتیجه می‌گیریم که $\varphi_q(x) \leq \varphi_{q-1}(x) - 1$. ▩

سه لم آخر نشان می‌دهند که هزینه‌ی سرشکن عملیات MAKE-SET، LINK، و FIND-SET برابر $O(\alpha(n))$ است. از تساوی (۱۷-۲) به یاد بیاورید که هزینه‌ی سرشکن هر عملیات برابر است با هزینه‌ی واقعی آن به علاوه‌ی افزایش پتانسیل به خاطر آن عملیات.

لم ۱۱-۲۱ هزینه‌ی سرشکن هر عملیات MAKE-SET برابر است با $O(1)$.

اثبات فرض کنید که qامین عملیات MAKE-SET(x) باشد. این عملیات گره‌ی x را با رتبه‌ی ۰ می‌سازد، و بنابراین $\varphi_q(x) = 0$. هیچ رتبه یا پتانسیل دیگری تغییر نمی‌کند، و بنابراین $\Phi_q = \Phi_{q-1}$. توجه به این که هزینه‌ی واقعی MAKE-SET برابر $O(1)$ است اثبات را کامل می‌کند. ▩

لم ۱۲-۲۱ هزینه‌ی سرشکن عملیات LINK برابر است با $O(\alpha(n))$.

اثبات فرض کنید که qامین عملیات LINK(x, y) باشد. هزینه‌ی واقعی عملیات LINK عبارت است از $O(1)$. بدون از دست دادن کلیت مسئله، فرض می‌کنیم LINK گره‌ی y را پدر گره‌ی x می‌کند.

برای تعیین تغییر پتانسیل به خاطر عملیات LINK، توجه می‌کنیم که تنها گره‌هایی که پتانسیل آن‌ها ممکن است تغییر کند x و y هستند، و همچنین فرزندان y دقیقاً قبل از عملیات. نشان خواهیم داد که تنها گره‌ای که پتانسیل آن ممکن است به خاطر LINK افزایش یابد y است، و این افزایش

حداکثر $\alpha(n)$ است:

- طبق لم ۲۱-۱۰، پتانسیل هر گره‌ای که دقیقاً قبل از LINK فرزند y است، نمی‌تواند به خاطر LINK افزایش یابد.
- طبق تعریف $\varphi_q(x)$، می‌بینیم که از آن جایی که x قبل از qامین عملیات ریشه بوده است، $\varphi_{q-1}(x) = \alpha(n) \cdot x.rank$. اگر $x.rank = 0$، آن گاه $\varphi_q(x) = \varphi_{q-1}(x) = 0$. در غیر این صورت،

$$\varphi_q(x) < \alpha(n) \cdot x.rank \qquad \text{(طبق نتیجه‌ی ۲۱-۹)}$$
$$= \varphi_{q-1}(x)$$

و بنابراین پتانسیل x کاهش یافته است.

- چون y قبل از LINK ریشه بوده است داریم $\varphi_{q-1}(y) = \alpha(n) \cdot y.rank$. عملیات LINK گره‌ی y را ریشه باقی می‌گذارد، و همچنین یا رتبه‌ی y را تغییری نمی‌دهد و یا آن را یکی افزایش می‌دهد. بنابراین یا داریم $\varphi_q(y) = \varphi_{q-1}(y)$ و یا $\varphi_q(y) = \varphi_{q-1}(y) + \alpha(n)$.

از این رو افزایش پتانسیل به خاطر عملیات LINK حداکثر $\alpha(n)$ است. هزینه‌ی سرشکن عملیات LINK برابر خواهد بود با $O(1) + \alpha(n) = O(\alpha(n))$. ■

لم ۲۱-۱۲ هزینه‌ی سرشکن هر عملیات FIND-SET برابر است با $O(\alpha(n))$.

اثبات فرض کنید qامین عملیات FIND-SET باشد، و همچنین مسیر ریشه شامل s گره شود. هزینه‌ی واقعی FIND-SET برابر است با $O(s)$. نشان خواهیم داد که پتانسیل هیچ گره‌ای به خاطر FIND-SET تغییر نخواهد کرد و پتانسیل حداقل $\max(0, s - (\alpha(n) + 2))$ گره در مسیر ریشه یکی کاهش خواهد یافت.

برای این که ببینیم پتانسیل هیچ گره‌ای افزایش نمی‌یابد، ابتدا برای تمام گره‌ها غیر از ریشه به لم ۲۱-۹ رجوع می‌کنیم. اگر x ریشه باشد آن گاه پتانسیل آن $\alpha(n) \cdot x.rank$ است، که تغییری نمی‌کند.

اکنون نشان می‌دهیم که پتانسیل حداقل $\max(0, s - (\alpha(n) + 2))$ گره یکی کاهش می‌یابد. فرض کنید x گره‌ای بر روی مسیر ریشه باشد به طوری که $x.rank > 0$ و گره‌ای مانند y در مسیر ریشه (که ریشه نیست) بعد از x است به طوری که دقیقاً قبل از عملیات FIND-SET داریم $level(y) = level(x)$. (نیازی نیست گره‌ی y در مسیر ریشه *دقیقاً* بعد از x باشد.) تمام گره‌ها غیر از حداکثر $\alpha(n) + 2$ گره در مسیر ریشه این شرط‌ها را برای x ارضا می‌کنند. آن‌هایی که این شرایط را ارضا نمی‌کنند عبارتند از اولین گره در مسیر ریشه (اگر رتبه‌ی آن 0 باشد)، آخرین گره در مسیر (ریشه)، و آخرین گره‌ی w در مسیر به طوری که برای $k = 0, 1, 2, \ldots, \alpha(n) - 1$ داشته باشیم $level(w) = k$.

اجازه دهید گره‌ی x را به طور ثابت تعیین کنیم، و سپس نشان خواهیم داد که پتانسیل x حداقل یکی کاهش خواهد یافت. فرض کنید $level(x) = level(y)$. دقیقاً قبل از تراکم مسیر رخ داده توسط FIND-SET داریم

$$\begin{aligned} x.p.rank &\geq A_k^{iter(x)}(x.rank) && \text{(طبق تعریف } iter(x)\text{)} \\ x.p.rank &\geq A_k(y.rank) && \text{(طبق تعریف } level(x)\text{)} \\ y.rank &\geq x.p.rank && \text{(طبق نتیجه‌ی ۲۱-۵ و چون در مسیر ریشه } y \text{ بعد از } x \text{ می‌آید)} \end{aligned}$$

با قرار دادن این نامساوی‌ها در کنار یکدیگر و با قرار دادن i به عنوان مقدار $iter(x)$ قبل از تراکم، خواهیم داشت

$$\begin{aligned} x.p.rank &\geq A_k(y.rank) \\ &\geq A_k(x.p.rank) && \text{(چون } A_k(j) \text{ صعودی اکید است)} \\ &\geq A_k(A_k^{iter(x)}(x.rank)) \\ &= A_k^{i+1}(x.rank) \end{aligned}$$

چون تراکم مسیر باعث می‌شود پدر x و y یکی شود، می‌دانیم که بعد از تراکم مسیر خواهیم داشت $x.p.rank = y.p.rank$، و تراکم مسیر $y.p.rank$ را کاهش نمی‌دهد. از آن جایی که $x.rank$ تغییری نمی‌کند، بعد از تراکم مسیر داریم $x.p.rank \geq A_k^{i+1}(x.rank)$. بنابراین تراکم مسیر باعث می‌شود که یا $iter(x)$ کاهش یابد (حداقل به اندازه‌ی i) و یا $level(x)$ افزایش یابد (که زمانی اتفاق می‌افتد که $iter(x)$ حداقل به اندازه‌ی $x.rank + 1$ افزایش یابد). در هر دو حالت طبق لم ۲۱-۹ داریم $\varphi_q(x) \leq \varphi_{q-1}(x) - 1$. بنابراین پتانسیل x حداقل یکی کاهش می‌یابد.

هزینه‌ی سرشکن عملیات FIND-SET برابر است با هزینه‌ی واقعی آن به علاوه‌ی تغییرات پتانسیل. هزینه‌ی واقعی $O(s)$ است، و نشان دادیم که کل پتانسیل حداقل به اندازه‌ی $\max(0, s-(\alpha(n)+2))$ کاهش می‌یابد. بنابراین هزینه‌ی سرشکن حداکثر برابر است با $O(s) - (s-(\alpha(n)+2)) = O(s) - s + O(\alpha(n)) = O(\alpha(n))$، چرا که می‌توانیم واحد پتانسیل را به اندازه‌ای بزرگ کنیم که بر ثابت‌های مخفی $O(s)$ غلبه کند. ■

قرار دادن لم‌های بالا در کنار یکدیگر به قضیه‌ی زیر ختم می‌شود.

قضیه‌ی ۲۱-۱۴ دنباله‌ای از m عملیات MAKE-SET، UNION، و FIND-SET را، که n تا از آن‌ها عملیات MAKE-SET هستند، می‌توان به کمک اجتماع بر حسب رتبه و تراکم مسیر بر روی یک جنگل مجموعه‌های منفصل در زمان $O(m\alpha(n))$ انجام داد.

اثبات مستقیماً از لم‌های ۲۱-۷، ۲۱-۱۱، ۲۱-۱۲ و ۲۱-۱۳. ■

تمرین‌ها

۲۱-۴-۱ لم ۲۱-۴ را اثبات کنید.

۲۱-۴-۲ اثبات کنید که رتبه‌ی هر گره حداکثر $\lfloor \lg n \rfloor$ است.

۲۱-۴-۳ به کمک تمرین ۲۱-۴-۲ تعیین کنید که برای ذخیره‌ی $x.rank$ برای هر گره‌ی x چند بیت نیاز داریم.

۲۱-۴-۴ با استفاده از تمرین ۲۱-۴-۲ یک اثبات ساده برای این قضیه ارائه کنید که عملیات بر روی یک جنگل مجموعه‌های منفصل با اجتماع بر حسب رتبه و تراکم مسیر در زمان $O(m \lg n)$ اجرا می‌شوند.

۲۱-۴-۵ پروفسور Dante استدلال می‌کند که چون رتبه‌ی گره‌ها در یک مسیر به ریشه اکیدا افزایش می‌یابد، سطح گره‌ها باید در طول مسیر به طور یکنواخت افزایش یابد. به عبارت دیگر اگر $rank(x) > 0$ و $x.p$ ریشه نباشد، آن گاه $level(x) \le level(x.p)$. آیا پروفسور درست می‌گوید؟

۲۱-۴-۶ ★ تابع $\alpha'(n) = \min\{k : A_k(1) \ge \lg(n+1)\}$ را در نظر بگیرید. نشان دهید که برای تمام مقادیر قابل کاربرد n داریم $\alpha'(n) \le 3$، و با استفاده از تمرین ۲۱-۴-۲ نشان دهید که چگونه می‌توان با تغییر آرگومان تابع پتانسیل، ثابت کرد که دنباله‌ای از m عملیات MAKE-SET، UNION، و FIND-SET را، که n تا از آن‌ها MAKE-SET هستند، به کمک اجتماع بر حسب رتبه و تراکم مسیر می‌توان بر روی یک جنگل مجموعه‌های منفصل در بدترین حالت زمان اجرای $O(m\alpha'(n))$ انجام داد.

مسائل

۲۱-۱ کمینه‌ی آفلاین

مسئله‌ی کمینه‌ی آفلاین این است که یک مجموعه‌ی پویای T از عناصری از دامنه‌ی $\{1, 2, \ldots, n\}$ را تحت اعمال INSERT و EXTRACT-MIN نگه داریم. به ما دنباله‌ی S از n عملیات INSERT و m عملیات EXTRACT-MIN داده شده است، که در آن هر کلید در مجموعه‌ی $\{1, 2, \ldots, n\}$ دقیقاً یک بار درج می‌شود. می‌خواهیم تعیین کنیم که با فراخوانی هر EXTRACT-MIN کدام کلید بازگردانده می‌شود. به خصوص می‌خواهیم آرایه‌ی $extracted[1..m]$ را پر کنیم، که در آن برای $i = 1, 2, \ldots, m$، $extracted[i]$ کلیدی است که توسط iامین EXTRACT-MIN بازگردانده شده است. مسئله «آفلاین» است چون که ما اجازه داریم کل دنباله‌ی S را قبل از تعیین کلیدهای بازگردانده شده اجرا کنیم.

مسائل

I. در نمونه‌ی زیر از مسئله‌ی کمینه‌ی آفلاین، هر INSERT (i) با یک عدد و هر EXTRACT-MIN با یک حرف E نشان داده شده است.

$$۴, ۸, E, ۳, E, ۹, ۲, ۶, E, E, E, ۱, ۷, E, ۵$$

مقادیر صحیح را در آرایه‌ی *extracted* جای گذاری کنید.

برای توسعه‌ی یک الگوریتم برای این مسئله، دنباله‌ی S را به زیردنباله‌های مشابه تقسیم می‌کنیم. یعنی S را توسط

$$I_۱, E, I_۲, E, I_۳, ..., I_m, E, I_{m+۱},$$

نمایش می‌دهیم، که در آن هر E نشان‌دهنده‌ی یک فراخوانی EXTRACT-MIN و هر I_j نشان دهنده‌ی یک دنباله‌ی (احتمالاً تهی) از فراخوانی‌های INSERT است. برای هر زیردنباله‌ی I_j، ابتدا کلیدهای درج شده توسط این اعمال را در یک مجموعه‌ی K_j قرار می‌دهیم، که در صورت خالی بودن I_j تهی است. سپس الگوریتم زیر را اجرا می‌کنیم.

```
OFF-LINE-MINIMUM(m, n)
1  for i = 1 to n
2      determine j such that i ∈ K_j
3      if j ≠ m + 1
4          extracted[j] = i
5          let l be the smallest value greater than j
               for which set K_l exists
6          K_l = K_j ∪ K_l, destroying K_j
7  return extracted
```

II. بحث کنید که آرایه‌ی *extracted* بازگردانده شده توسط OFF-LINE-MINIMU درست است.

III. توضیح دهید که چطور می‌توان OFF-LINE-MINIMUM را با استفاده از یک ساختمان داده‌ی مجموعه‌های منفصل به صورت بهینه پیاده‌سازی کرد. یک کران نزدیک برای بدترین حالت زمان اجرای پیاده‌سازی خود ارائه کنید.

۲۱-۲ تعیین عمق

در مسئله‌ی تعیین عمق، یک جنگل $\mathcal{F} = \{T_i\}$ از درخت‌های ریشه‌دار را تحت سه عملیات زیر نگه می‌داریم:

- MAKE-TREE (v) یک درخت می‌سازد که تنها گره‌ی آن v است.
- FIND-DEPTH (v) عمق گره‌ی v را در درخت آن بازمی‌گرداند.
- GRAFT (r,v) گره‌ی r را، که فرض بر این است که ریشه‌ی درخت است، تبدیل به فرزند گره‌ی v می‌کند، که فرض بر این است که در درختی غیر از درخت r باشد، ولی می‌تواند ریشه باشد یا نباشد.

I. فرض کنید از یک نمایش درخت مانند نمایش استفاده شده در جنگل‌های مجموعه‌های منفصل استفاده می‌کنیم: $v.p$ پدر گره‌ی v است، غیر از حالتی که v ریشه باشد، که در این صورت $v.p = v$. اگر GRAFT(r,v) را با قرار دادن $r.p = v$، و FIND-DEPTH(v) را با دنبال کردن یک مسیر به سمت بالا تا یافتن ریشه و بازگرداندن تعداد گره‌ها غیر از v پیاده‌سازی کنیم، نشان دهید که بدترین حالت زمان اجرای دنباله‌ای از m عملیات MAKE-TREE، FIND-DEPTH، و GRAFT برابر است با $\theta(m^2)$.

با استفاده از مکاشفه‌های اجتماع بر حسب رتبه و تراکم مسیر، می‌توان بدترین حالت زمان اجرا را کاهش داد. از جنگل مجموعه‌های منفصل $S = \{S_i\}$ استفاده می‌کنیم، که در آن هر مجموعه‌ی S_i (که خود یک درخت است) متناظر است با یک درخت در جنگل $\mathcal{F}$. با این حال ساختار درخت درون یک مجموعه‌ی S_i، لزوماً با ساختار درخت T_i یکی نیست. در واقع پیاده‌سازی S_i رابطه‌ی پدر- فرزندی را دقیقاً ذخیره نمی‌کند، ولی به ما اجازه می‌دهد که عمق هر گره‌ای را در T_i تعیین کنیم.

ایده‌ی اصلی این است که در هر گره‌ی v یک «شبه‌فاصله»ی (pseudodistance) $v.d$ نگه داریم، که طوری تعریف شده است مجموعه شبه‌فاصله‌ها در مسیر از v به ریشه‌ی مربوط به مجموعه‌ی S_i برابر است با عمق v در T_i. یعنی اگر مسیر از v به ریشه در S_i، $v_0, v_1, \ldots, v_k$ باشد، که در آن $v = v_0$ و v_k برابر است با ریشه‌ی S_i'، آن گاه عمق v در T_i برابر است با $\sum_{j=0}^{k} v_j.d$.

II. یک پیاده‌سازی برای MAKE-TREE ارائه کنید.

III. نشان دهید که چگونه می‌توان FIND-SET را برای پیاده‌سازی FIND-DEPTH اصلاح کرد. پیاده‌سازی شما باید تراکم مسیر را انجام دهد، و زمان اجرای آن باید نسبت به طول مسیر ریشه خطی باشد. اطمینان حاصل کنید که پیاده‌سازی شما شبه‌فاصله‌ها را به درستی به هنگام‌سازی می‌کند.

IV. نشان دهید که چگونه می‌توان GRAFT(r,v) را با اصلاح رویه‌های UNION و LINK پیاده‌سازی کرد، که این عملیات دو مجموعه شامل r و v را با هم ترکیب می‌کند. اطمینان حاصل کنید که پیاده‌سازی شما شبه‌فاصله‌ها را به درستی به هنگام سازی می‌کند. توجه داشته باشید که ریشه‌ی یک مجموعه‌ی S_i لزوماً ریشه‌ی درخت متناظر T_i نیست.

V. یک کران نزدیک برای بدترین حالت زمان اجرای دنباله‌ای از m عملیات MAKE-TREE، FIND-DEPTH، و GRAFT بدهید، که n تا از آن‌ها عملیات MAKE-TREE هستند.

۲۱-۳ الگوریتم پایین‌ترین جدهای مشترک آفلاین Tarjan

پایین‌ترین جد مشترک دو گره‌ی u و v در درخت ریشه‌دار T گره‌ی w است که جد هر دو گره‌ی u و v است، و بیشترین عمق ممکن را در درخت T دارد. در *مسئله‌ی پایین‌ترین*

مسائل

جدهای مشترک آفلاین، به ما یک درخت ریشه‌دار T و یک مجموعه‌ی دلخواه $P = \{\{u, v\}\}$ از جفت‌های نامرتب از گره‌ها در T داده شده است، و می‌خواهیم پایین‌ترین جد مشترک هر جفت را در P بیابیم.

برای حل مسئله‌ی پایین‌ترین جدهای مشترک آفلاین، شبه‌کد زیر یک پیمایش در درخت T را با فراخوانی اولیه‌ی LCA($T.root$) انجام می‌دهد. فرض می‌شود که قبل از انجام پیمایش، هر گره با WHITE رنگ آمیزی شده است.

```
LCA(u)
1  MAKE-SET(u)
2  FIND-SET(u).ancestor = u
3  for each child v of u in T
4      LCA(v)
5      UNION(u, v)
6      FIND-SET(u).ancestor = u
7  u.color = BLACK
8  for each node v such that {u, v} ∈ P
9      if v.color == BLACK
10         print "The least common ancestor of"
               u "and" v "is" FIND-SET(v).ancestor
```

I. بحث کنید که خط ۱۰ برای هر جفت $\{u, v\} \in P$ دقیقاً یک بار اجرا می‌شود.

II. بحث کنید که در زمان فراخوانی LCA(u)، تعداد مجموعه‌ها در ساختمان داده‌ی مجموعه‌های منفصل برابر است با عمق u در T.

III. اثبات کنید که LCA به درستی پایین‌ترین جد مشترک u و v را برای هر جفت $\{u, v\} \in P$ چاپ می‌کند.

IV. زمان اجرای LCA را تحلیل کنید، با این فرض که از پیاده‌سازی ساختمان داده‌ی مجموعه‌های منفصل ارائه شده در بخش ۲۱-۳ استفاده می‌کنیم.

بخش ششم الگوریتم‌های گراف

شامل فصل‌های :

مقدمه

ساختمان داده‌ی گراف در علم کامپیوتر بسیار فراگیر شده است، و از این رو الگوریتم‌های گراف هم از اهمیت ویژه‌ای برخوردارند. صدها مسئله‌ی محاسباتی جذاب وجود دارند که با استفاده از گراف‌ها تعریف شده‌اند. در این بخش تعداد کمی از الگوریتم‌های مهم گراف را مورد بررسی قرار می‌دهیم.

فصل ۲۲ نشان می‌دهد که چگونه می‌توان یک گراف را بر روی کامپیوتر نمایش داد، و سپس الگوریتم‌هایی را بحث می‌کند که بر مبنای جستجوی گراف (جستجوی سطح اول و یا عمق اول) بنا شده‌اند. دو کاربرد از جستجوی عمق اول داده شده است: مرتب‌سازی توپولوژیکی یک گراف جهت‌دار بدون دور و تجزیه‌ی یک گراف جهت‌دار به مؤلفه‌های قویاً همبند.

فصل ۲۳ توضیح می‌دهد که چگونه می‌توان یک درخت پوشا با وزن کمینه را برای یک گراف محاسبه کرد. چنین درختی برای گرافی که در آن هر یال یک وزن مربوطه دارد، به صورت کم‌وزن‌ترین راه برای اتصال تمام رأس‌ها به یکدیگر تعریف می‌شود. الگوریتم‌های محاسبه‌ی درخت‌های پوشای کمینه مثال‌های خوبی از الگوریتم‌های حریصانه هستند (فصل ۱۶ را ببینید).

فصل‌های ۲۴ و ۲۵ مسئله‌ی محاسبه‌ی کوتاه‌ترین مسیرها را میان رأس‌ها بررسی می‌کنند، که در آن هر یال یک طول یا «وزن» متناظر دارد. فصل ۲۴ محاسبه‌ی کوتاه‌ترین مسیرها از یک رأس مبدأ داده شده به تمام رأس‌های دیگر را بررسی می‌کند، و در فصل ۲۵ راه‌های محاسبه‌ی کوتاه‌ترین مسیرها میان هر دو جفت از رأس‌ها ارائه خواهند شد.

نهایتاً فصل ۲۶ نشان می‌دهد که چگونه می‌توان با داشتن یک منبع از یک ماده‌ی خاص، یک مبدأ مشخص و ظرفیت مشخص شده برای هر یال، بیشینه‌ی شار آن ماده را در یک شبکه (یک گراف

جهت‌دار) محاسبه کرد. این مسئله‌ی کلی به صورت‌های مختلفی پیش می‌آید، و یک الگوریتم خوب برای محاسبه‌ی شار بیشینه می‌تواند در حل مسائل مختلفی به کار آید.

در تعیین زمان اجرای یک الگوریتم گراف بر روی گراف داده شده‌ی $G = (V, E)$، معمولاً اندازه‌ی ورودی را بر حسب $|V|$، تعداد رأس‌ها، و $|E|$، تعداد یال‌های گراف توصیف می‌کنیم. یعنی به جای یک پارامتر، دو پارامتر اندازه‌ی ورودی را تعیین می‌کنند. برای این پارامترها نمادهای قراردادی ساده در نظر می‌گیریم. درون نمادهای حدی (مانند نمادهای O و θ)، و *فقط* درون این نمادها، نماد V نشان‌دهنده‌ی $|V|$ و نماد E نشان‌دهنده‌ی $|E|$ است. مثلاً ممکن است بگوییم «الگوریتم در زمان $O(VE)$ اجرا می‌شود»، که معنی آن این است که الگوریتم در زمان $O(|V||E|)$ اجرا می‌شود. این قرارداد باعث می‌شود بدون ریسک وجود ابهام، خواندن فرمول‌های زمان اجرا ساده‌تر شود.

قرارداد دیگری هم در شبه‌کدها تعریف می‌کنیم. مجموعه‌ی رأس‌های یک گراف G را با $G.V$ و مجموعه‌ی یال‌های آن را با $G.E$ نشان خواهیم داد. یعنی مجموعه‌ی رأس‌ها و یال‌های گراف را به صورت خصیصه‌هایی از گراف می‌بینیم.

الگوریتم‌های اولیه‌ی گراف

در این فصل متدهایی برای نمایش گراف و جستجو در آن معرفی خواهد شد. جستجو در یک گراف، یعنی دنبال کردن یال‌های آن به صورت منظم برای ملاقات (visit) رأس‌ها. یک الگوریتم جستجو در گراف می‌تواند نکات بسیاری را در مورد ساختار آن روشن کند. الگوریتم‌های بسیاری وجود دارند که قبل از هر کار، ابتدا در گراف ورودی خود جستجو می‌کنند تا اطلاعات ساختار آن را به دست آورند، و الگوریتم‌های زیاد دیگری هستند که فقط ترکیب ساده‌ی الگوریتم‌های جستجو در گراف هستند.

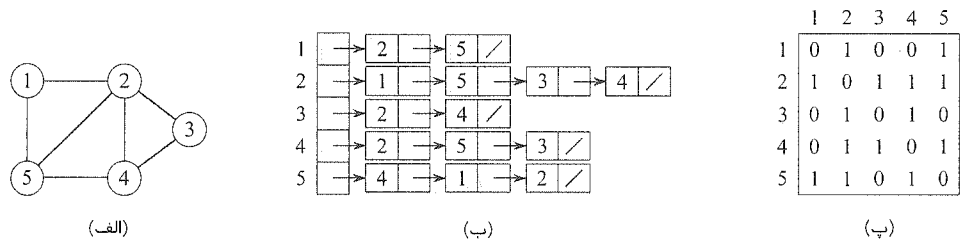
در بخش ۲۲-۱ دو شیوه‌ی مرسوم نمایش گراف‌ها در کامپیوتر معرفی می‌شود: لیست مجاورت و ماتریس مجاورت. در بخش ۲۲-۲ در مورد یک الگوریتم ساده‌ی جستجوی گراف به نام جستجوی سطح اول (BFS)، و روش ساخت درخت سطح اول بحث خواهد شد. در بخش ۲۲-۳ جستجوی عمق اول را معرفی، و چند نتیجه‌ی استاندارد در مورد ترتیب ملاقات رأس‌ها در این روش جستجو را اثبات می‌کنیم. در بخش ۲۲-۴ اولین کاربرد واقعی جستجوی عمق اول معرفی خواهد شد: مرتب‌سازی توپولوژیکی یک گراف جهت‌دار بدون دور. دومین کاربرد جستجوی عمق اول، یافتن بخش‌های قویاً همبند یک گراف جهت‌دار، در بخش ۲۲-۵ معرفی خواهد شد.

۲۲-۱ نمایش گراف‌ها

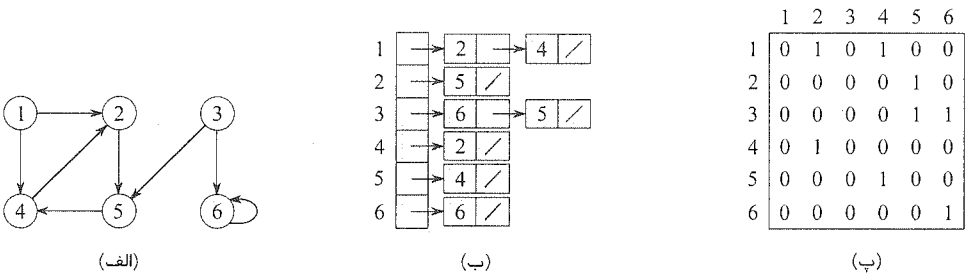
دو راه استاندارد برای نمایش یک گراف $G = (V, E)$ وجود دارد: به صورت مجموعه‌ای از لیست‌های پیوندی (لیست مجاورت) و یا به صورت ماتریس مجاورت. هر دو راه را می‌توان برای هر دو نوع گراف جهت‌دار و بدون جهت به کار برد. معمولاً نمایش لیست پیوندی ترجیح دارد، زیرا روشی فشرده برای نمایش گراف‌های خلوت (sparse) – گراف‌هایی که در آن‌ها $|E|$ بسیار کمتر از $|V|^2$ است – فراهم می‌کند. در اکثر الگوریتم‌های گراف ارائه شده در این کتاب فرض شده است که گراف به

صورت لیست مجاورت نمایش داده می‌شود. با این حال زمانی که گراف شلوغ (dense) – گرافی که در آن $|E|$ به $|V|^2$ نزدیک است – باشد، و یا زمانی که بخواهیم وجود یک یال بین دو رأس را با سرعت بیشتری تشخیص دهیم، روش نمایش ماتریس مجاورت مناسبت خواهد بود. به عنوان مثال در دو تا از الگوریتم‌های «کوتاه‌ترین مسیر بین تمام رأس‌ها» (All pairs shortest path) که در فصل ۲۵ ارائه شده‌اند، فرض شده است که گراف به صورت ماتریس مجاورت نمایش داده می‌شود.

نمایش لیست مجاورت گراف $G = (V, E)$ شامل یک آرایه (که آن را Adj می‌نامیم) از لیست‌های پیوندی با اندازه‌ی V است، یک لیست برای هر رأس. برای هر $u \in V$، لیست $Adj[u]$ شامل تمام رأس‌های v است که $(u, v) \in E$. یعنی $Adj[u]$ از تمام رأس‌هایی تشکیل شده است که همسایه‌ی u هستند (البته ممکن است در لیست پیوندی به جای رأس‌ها، اشاره‌گرهایی به رأس‌ها ذخیره شود). چون لیست‌های پیوندی یال‌های گراف را نشان می‌دهند، در شبه‌کد با آرایه‌ی Adj به صورت خصیصه‌ای از گراف برخورد می‌کنیم، درست مانند مجموعه‌ی یال‌های E. بنابراین نمادهایی مانند $G.Adj[u]$ خواهیم داشت. شکل ۲۲-۱(ب) یک نمایش لیست مجاورت برای گراف شکل ۲۲-۱(الف) است. به طور مشابه، شکل ۲۲-۲(ب) یک نمایش لیست مجاورت برای گراف شکل ۲۲-۲(الف) است.



شکل ۲۲-۱ دو نمایش یک گراف بدون جهت. (الف) یک گراف بدون جهت G شامل پنج رأس و هفت یال. (ب) نمایش لیست مجاورت گراف G. (پ) نمایش ماتریس مجاورت گراف G.



شکل ۲۲-۲ دو نمایش یک گراف جهت‌دار. (الف) یک گراف جهت‌دار G شامل شش رأس و هشت یال. (ب) نمایش لیست مجاورت گراف G. (پ) نمایش ماتریس مجاورت گراف G.

اگر G یک گراف جهت‌دار باشد، مجموع طول تمام لیست‌های پیوندی در نمایش گراف G برابر با $|E|$ خواهد بود، چرا که نمایش یک یال (u,v) بدین صورت است که v در $Adj[u]$ ظاهر شود. اگر G یک گراف بدون جهت باشد، مجموع طول تمام لیست‌های پیوندی برابر با $2|E|$ خواهد بود، زیرا اگر (u,v) یک یال بدون جهت باشد، هم u در لیست پیوندی v ظاهر خواهد شد و هم v در لیست پیوندی u. برای هر دو نوع گراف جهت‌دار و بدون جهت، نمایش لیست پیوندی این خاصیت رضایت بخش را دارد که میزان حافظه‌ی مصرفی آن از مرتبه‌ی $\theta(V+E)$ است.

از لیست پیوندی می‌توان برای نمایش گراف‌های وزن‌دار هم استفاده کرد. گراف‌های وزن‌دار گراف‌هایی هستند که به هر یال یک وزن هم نسبت می‌دهند (معمولاً به کمک یک تابع $w : E \to R$). برای مثال فرض کنید $G = (V,E)$ یک گراف وزن‌دار با تابع وزن w باشد. می‌توان به سادگی وزن $w(u,v)$ را به همراه v در لیست پیوندی u ذخیره کرد. نمایش لیست پیوندی بسیار قدرتمند است، و به کمک آن می‌توان خواص مختلفی از گراف را ذخیره کرد.

یکی از ضعف‌های لیست پیوندی این است که برای فهمیدن این که آیا یال (u,v) در گراف G وجود دارد یا نه، باید تمام لیست پیوندی $Adj[u]$ را جستجو کنیم، و هیچ روش سریع‌تری برای این کار وجود ندارد. این ضعف را می‌توان به کمک نمایش ماتریس مجاورت مرتفع کرد، که هزینه‌ی آن مصرف حافظه‌ی بیشتر است. (در تمرین ۲۲-۸-۱ تغییراتی در لیست پیوندی پیشنهاد خواهد شد که سرعت جستجو را در آن بهبود می‌بخشد.)

برای نمایش *ماتریس مجاورت گراف*، فرض می‌کنیم یال‌ها به صورت $1, 2, \ldots, |V|$ نام‌گذاری شده‌اند. در این صورت نمایش ماتریس مجاورت گراف G یک ماتریس $A = (a_{ij})$ با اندازه‌ی $|V| \times |V|$ خواهد بود که در آن:

$$a_{ij} = \begin{cases} 1 & \text{اگر } (i,j) \in E \\ 0 & \text{در غیر این صورت} \end{cases}$$

شکل‌های ۲۲-۱(پ) و ۲۲-۲(پ)، نمایش ماتریس مجاورت گراف‌های بدون جهت و جهت‌دار شکل‌های ۲۲-۱(الف) و ۲۲-۲(الف) هستند. میزان حافظه‌ی مصرفی نمایش ماتریس مجاورت گراف مستقل از تعداد یال‌های آن، از مرتبه‌ی $\theta(V^2)$ است.

به تقارن موجود در ماتریس مجاورت شکل ۲۲-۱(پ) نسبت به قطر اصلی دقت کنید. ماتریس *ترانهاده‌ی* (transpose) ماتریس $A = (a_{ij})$ را اینگونه تعریف می‌کنیم: $A^T = (a_{ij}^T)$ که در آن $a_{ij}^T = a_{ji}$. از آن جایی که در گراف‌های بدون جهت (u,v) و (v,u) نماینده‌ی یک یال هستند، ترانهاده‌ی ماتریس مجاورت یک گراف بدون جهت با خود آن برابر است: $A = A^T$. در بعضی کاربردها فقط داده‌های روی قطر اصلی و بالای آن را ذخیره می‌کنند، که حافظه‌ی مصرفی را تقریباً به نصف کاهش می‌دهد.

مانند لیست مجاورت، برای ذخیره‌ی گراف‌های وزن‌دار از ماتریس مجاورت هم می‌توان استفاده

کرد. مثلاً اگر G یک گراف وزن‌دار با تابع وزن w باشد، می‌توان به راحتی وزن $w(u,v)$ را در ورودی ماتریس مجاورت گراف در سطر u و ستون v ذخیره کرد. اگر یک یال موجود نباشد، در ورودی مربوط به آن در ماتریس مقدار NIL ذخیره خواهد شد. با این حال در بسیاری مسائل مناسب‌تر است که به جای NIL، مقدار ۰ یا ∞ ذخیره شود.

با این که نمایش لیست مجاورت گراف به صورت حدی حداقل به کارامدی نمایش ماتریس مجاورت است، سادگی استفاده از ماتریس مجاورت باعث می‌شود که زمانی که گراف به حد کافی کوچک باشد، استفاده از ماتریس مجاورت ترجیح داشته باشد. به علاوه اگر گراف بدون وزن باشد، استفاده از ماتریس مجاورت این مزیت را دارد که برای هر ورودی به جای یک کلمه (word) در حافظه‌ی کامپیوتر، تنها یک بیت مصرف شود، که حافظه‌ی مصرفی آن را نیز تا حد قابل توجهی کاهش می‌دهد.

نمایش خصیصه‌ها

اکثر الگوریتم‌هایی که روی گراف‌ها کار می‌کنند نیاز دارند که خصیصه‌هایی از رأس‌ها و/یا یال‌های گراف را ذخیره کنند. این خصیصه‌ها را با نمادهای همیشگی نشان خواهیم داد، مثلاً $v.d$ برای خصیصه‌ی d از رأس v. وقتی یال‌ها را به صورت جفت‌هایی از رأس‌ها در نظر می‌گیریم هم از همان نمادگذاری استفاده می‌کنیم. مثلاً اگر یال‌ها یک خصیصه‌ی f داشته باشند، آن گاه این خصیصه را برای یال (u,v) به صورت $(u,v).f$ نمایش خواهیم داد. در این جا که هدف، نمایش و درک الگوریتم‌ها است، همین نمایش‌ها برای خصیصه‌ها کافی است.

پیاده‌سازی خصیصه‌های رأس‌ها و یال‌ها در برنامه‌های واقعی می‌تواند به کلی داستان متفاوتی باشد. هیچ بهترین روشی برای ذخیره و دسترسی به خصیصه‌های رأس‌ها و یال‌ها وجود ندارد. برای یک موقعیت خاص انتخاب شما احتمالاً به زبان برنامه‌نویسی مورد استفاده، الگوریتم پیاده‌سازی شده، و نحوه‌ی دسترسی بقیه‌ی برنامه به گراف بستگی دارد. اگر گراف را با استفاده از لیست‌های مجاورت نمایش می‌دهید، می‌توان خصیصه‌های رأس‌ها را در آرایه‌ای جداگانه ذخیره کرد، مثلاً آرایه‌ی $d[1..|V|]$ که موازی آرایه‌ی Adj است. اگر رأس‌های مجاور u در $Adj[u]$ باشند، آن گاه چیزی که به آن خصیصه‌ی $u.d$ می‌گوییم در واقع در عنصر آرایه‌ی $d[u]$ خواهد بود. راه‌های بسیار دیگری برای پیاده‌سازی خصیصه‌ها وجود دارد. مثلاً در یک زبان برنامه‌نویسی شیئ‌گرا، خصیصه‌های رأس‌ها می‌توانند به صورت متغیرهای نمونه در یک زیرکلاس از کلاس Vertex قرار داشته باشند.

تمرین‌ها

۲۲-۱-۱ اگر لیست مجاورت یک گراف جهت‌دار را داشته باشیم، چقدر طول می‌کشد درجه‌ی خروجی یک رأس را محاسبه کنیم؟ چقدر طول می‌کشد درجه‌ی ورودی را به دست آوریم؟

۲۲-۱-۲ یک نمایش لیست مجاورت برای یک درخت دودویی کامل با ۷ رأس به دست آورید. همچنین نمایش ماتریس مجاورت معادل آن را رسم کنید. فرض کنید رأس‌ها مانند یک هرم دودویی از ۱ تا ۷ شماره گذاری شده‌اند.

۲۲-۱-۳ ترانهاده‌ی یک گراف جهت‌دار $G = (V, E)$ برابر است با گراف $G^T = (V, E^T)$ که در آن $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$. بنابراین G^T همان G است که جهت تمام یال‌های آن برعکس شده باشد. برای هر دو حالت نمایش ماتریس مجاورت و لیست مجاورت، الگوریتم کارایی ارائه دهید که G^T را از G به دست آورد. مرتبه‌ی زمانی الگوریتم‌های خود را محاسبه کنید.

۲۲-۱-۴ لیست مجاورت یک گراف چندگانه‌ی $G = (V, E)$ را داریم. یک الگوریتم از مرتبه‌ی $O(V + E)$ به دست آورید که گراف بدون جهت $G' = (V, E')$ را محاسبه کند، که E' شامل همه‌ی یال‌های E است که در آن تمام یال‌های چندگانه با یک یال جایگزین شده‌اند و تمام طوقه‌ها حذف شده‌اند.

۲۲-۱-۵ مربع یک گراف جهت‌دار $G = (V, E)$، گراف $G^2 = (V, E^2)$ است که در آن $(u, w) \in E^2$ اگر و فقط اگر در G یک مسیر از u به w شامل حداکثر دو یال موجود باشد. برای هر دو حالت نمایش لیست مجاورت و ماتریس مجاورت، الگوریتم‌های کارایی طراحی کنید که G^2 را از روی G محاسبه کند. مرتبه‌ی زمانی الگوریتم‌های خود را به دست آورید.

۲۲-۱-۶ زمانی که از ماتریس مجاورت استفاده می‌شود، بیشتر الگوریتم‌ها از مرتبه‌ی زمانی $\Omega(V^2)$ هستند. اما چند استثنا هم وجود دارد. نشان دهید که تشخیص وجود یک حفره‌ی جهانی (universal sink) – رأسی که درجه‌ی ورودی آن $|V| - 1$ و درجه‌ی خروجی آن ۰ باشد – در گراف جهت‌دار G با استفاده از ماتریس مجاورت، در زمان $O(V)$ امکان پذیر است.

۲۲-۱-۷ ماتریس برخورد (incidence matrix) یک گراف جهت‌دار و بدون طوقه‌ی $G = (V, E)$، یک ماتریس $B = (b_{ij})$ با اندازه‌ی $|V| \times |E|$ است به طوری که:

$$b_{ij} = \begin{cases} -1 & \text{اگر یال } j \text{ از رأس } i \text{ خارج می‌شود} \\ 1 & \text{اگر یال } j \text{ به رأس } i \text{ وارد می‌شود} \\ 0 & \text{در غیر این صورت} \end{cases}$$

توضیح دهید که درایه‌های ضرب ماتریسی $B \cdot B^T$ (که در آن B^T ترانهاده‌ی ماتریس B است) نماینده‌ی چه چیزی هستند؟

۸-۱-۲۲ فرض کنید به جای لیست پیوندی، آرایه‌ی $Adj[u]$ یک جدول درهم است شامل رأس‌هایی مانند v که داریم $(u,v) \in E$. اگر زمان جستجو برای تمام یال‌ها یکسان باشد، زمان تشخیص وجود یک یال در گراف چقدر خواهد بود؟ کاستی این روش چیست؟ یک ساختمان داده برای لیست یال‌ها پیشنهاد کنید که این کاستی را نداشته باشد. آیا این ساختمان داده‌ی پیشنهادی در مقابل جدول درهم معایبی دارد؟

۲-۲۲ جستجوی سطح اول

جستجوی سطح اول (Breadth-firs search) یکی از ساده‌ترین الگوریتم‌های جستجو در گراف و الگوی اصلی بسیاری از الگوریتم‌های گراف است. الگوریتم درخت پوشای کمینه‌ی Prim (بخش ۲۳-۲) و الگوریتم کوتاه‌ترین مسیر از یک رأس Dijkestra (بخش ۲۴-۳) از ایده‌هایی مانند ایده‌های الگوریتم جستجوی سطح اول استفاده می‌کنند.

الگوریتم جستجوی سطح اول از یک رأس مشخص s (که می‌توان آن را در ورودی مشخص کرد) در گراف $G = (V, E)$ شروع کرده و رأس‌ها را می‌پیماید تا تمام رأس‌های قابل دسترس از s را «کشف» کند. این الگوریتم فاصله(کمترین تعداد رأس میانی)ی هر رأس را از s مشخص می‌کند. همچنین یک «درخت سطح اول» با ریشه‌ی s تشکیل می‌دهد که شامل تمام رأس‌های قابل دسترس از s می‌باشد. برای هررأس قابل دسترس از s مانند v، مسیر ساده‌ی از s به v در درخت سطح اول معادل یکی از کوتاه‌ترین مسیرها از s به v در G می‌باشد، یعنی یک مسیر که شامل کمترین تعداد رأس باشد. این الگوریتم برای هر دو نوع گراف جهت‌دار و بدون جهت قابل استفاده است.

جستجوی سطح اول اینگونه نام‌گذاری شده است چرا که مرز بین رأس‌های کشف شده و رأس‌های کشف نشده را به صورت سطحی می‌پیماید. یعنی این الگوریتم قبل از کشف هر رأس با فاصله‌ی $k+1$ از s، تمام رأس‌های با فاصله‌ی k از s را کشف می‌کند.

جستجوی سطح اول برای نگه داشتن آمار جستجو، رأس‌ها را با سه رنگ سفید، خاکستری و سیاه رنگ‌آمیزی می‌کند. در ابتدا تمام رأس‌ها سفید هستند، و اگر از s قابل دسترس باشند در هنگام جستجو ابتدا خاکستری و سپس سیاه می‌شوند. هر رأس زمانی «کشف» شده است که برای اولین بار توسط الگوریتم جستجو مورد دسترسی قرار می‌گیرد، و در آن زمان است که رنگ آن از سفید به خاکستری و سپس به سیاه تغییر می‌کند. بنابراین رأس‌های خاکستری و سیاه کشف شده‌اند، ولی جستجوی سطح اول بین این دو رنگ تمایز قایل می‌شود تا بتواند خاصیت «سطح اول» بودن جستجو را حفظ کند.[1] اگر $(u,v) \in E$ و رأس u سیاه باشد، آنگاه رأس v یا خاکستری است یا سیاه؛ یعنی

[1] بین رأس‌های خاکستری و سیاه تفاوت قائل می‌شویم تا به ما در درک نحوه‌ی عمل‌کرد جستجوی سطح اول کمک کند. در واقع، همان طور که تمرین ۲۲-۲-۳ نشان می‌دهد، حتی اگر رأس‌های خاکستری و سیاه را یکسان در نظر بگیریم تفاوتی در نتیجه‌ی الگوریتم رخ نخواهد داد.

تمام رأس‌های همسایه‌ی رأس‌های سیاه قبلاً کشف شده‌اند. رأس‌های خاکستری ممکن است چند همسایه‌ی سفید داشته باشند؛ آن‌ها مرز بین رأس‌های کشف شده و کشف نشده را مشخص می‌کنند.

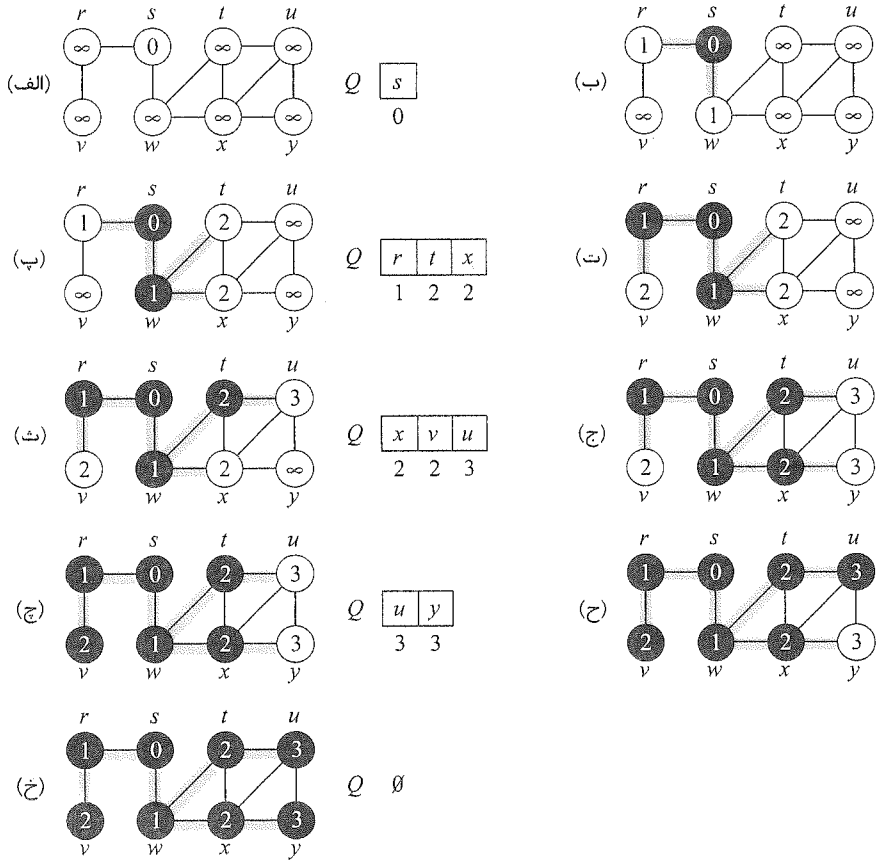
جستجوی سطح اول یک درخت سطح اول تشکیل می‌دهد که در ابتدا فقط شامل ریشه، یا همان رأس s است. هر گاه رأس v در مسیر جستجوی لیست مجاورت رأس فعلی (مثلاً u) کشف شد، به همراه یال (u,v) به درخت اضافه می‌شود. در این حالت می‌گوییم u رأس ماقبل (predecessor) یا پدر (parent) رأس v در درخت سطح اول است. از آن جایی که هر رأس حداکثر یک بار کشف می‌شود، پس حداکثر یک پدر دارد. رابطه‌های جد (ancestor) و نوه (descendant) در درخت سطح اول نسبت به ریشه (s) تعیین می‌شوند: اگر u روی یک مسیر (در واقع تنها مسیر) از s به v در درخت سطح اول باشد، آنگاه u جد v، و v نوه‌ی u محسوب می‌شود.

زیرروال جستجوی سطح اول زیر (BFS) فرض می‌کند که گراف $G = (V, E)$ با استفاده از لیست مجاورت نمایش داده شده است. رنگ هر رأس $u \in V$ در متغیر $u.color$ ذخیره شده است، و پدر u در متغیر $u.\pi$. اگر u پدری نداشته باشد (مثلاً اگر $u = s$، یا u کشف نشده باشد) آنگاه $u.\pi = \text{NIL}$. فاصله‌ی ریشه از رأس u در متغیر $u.d$ ذخیره می‌شود. به علاوه از یک صف (بخش ۱۰-۱ را ببینید) برای مدیریت رأس‌های خاکستری استفاده شده است.

```
BFS(G, s)
1   for each vertex u ∈ G.V − {s}
2       u.color = WHITE
3       u.d = ∞
4       u.π = NIL
5   s.color = GRAY
6   s.d = 0
7   s.π = NIL
8   Q = Ø
9   ENQUEUE(Q, s)
10  while Q ≠ Ø
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

شکل ۲۲-۳ جریان پیشرفت زیرروال BFS را روی یک گراف نمونه نشان می‌دهد.

زیرروال BFS بدین شکل کار می‌کند: در خطوط ۱-۴ غیر از رأس مبدأ s، تمام رأس‌ها با رنگ سفید رنگ‌آمیزی می‌شوند، $u.d$ برای تمام رأس‌های u با بی‌نهایت مقداردهی می‌شود، و پدر تمام



شکل ۲۲-۳ عملیات BFS روی یک گراف بدون جهت. یال‌های درخت پس از ساخته شدن توسط BFS به صورت سایه‌دار نمایش داده شده‌اند، و درون هر رأس u مقدار خصیصه‌ی $u.d$ مشخص شده است. صف Q در شروع هر تکرار حلقه‌ی while (خطوط ۱۰-۱۸) نمایش داده شده است، و درون صف، فاصله‌ی رأس‌ها از مبدأ زیر هر رأس دیده می‌شود.

رأس‌ها برابر با NIL مقداردهی می‌شود. خط ۵ رأس مبدأ (s) را خاکستری می‌کند، چرا که این رأس با شروع زیرروال کشف می‌شود. در خط ۶ خصیصه‌ی $s.d$ مقدار اولیه‌ی ۰ می‌گیرد و در خط ۷، پدر رأس مبدأ مقدار NIL دریافت می‌کند. خطوط ۸ و ۹ به Q مقدار اولیه می‌دهند، که این مقدار اولیه صفی است فقط شامل رأس s.

حلقه‌ی while خطوط ۱۰-۱۸ تا زمانی تکرار می‌شود که رأس خاکستری در گراف باقی مانده باشد، که این رأس‌ها، رأس‌های کشف شده‌ای هستند که هنوز همسایه‌های آن‌ها مورد آزمایش قرار نگرفته‌اند. حلقه‌ی while ثابت حلقه‌ی زیر را حفظ می‌کند:

- زمان تست در خط ۱۰، صف Q مجموعه‌ی رأس‌های خاکستری است.

با این حال که از این ثابت حلقه برای اثبات درستی الگوریتم استفاده نخواهیم کرد،چک کردن این که در زمان شروع حلقه و پس از هر بار تکرار حلقه این خاصیت پابرجاست، کار ساده‌ای است. قبل از اولین تکرار، تنها رأس خاکستری و تنها رأس موجود در صف رأس s است. خط ۱۱، u را برابر با رأس خاکستری در ابتدای Q قرار داده و آن را از Q حذف می‌کند. حلقه‌ی for خطوط ۱۲–۱۷ تمام رأس‌های v درون لیست مجاورت u را در نظر می‌گیرد: اگر v سفید باشد یعنی هنوز کشف نشده است، و الگوریتم با اجرای خطوط ۱۴–۱۷ آن را کشف می‌کند: ابتدا آن را خاکستری می‌کند، سپس $v.d$ را با $u.d+1$ مقداردهی می‌کند، u را به عنوان پدر آن مشخص می‌کند، و آن را در انتهای صف قرار می‌دهد. در نهایت u در انتهای صف قرار می‌گیرد. وقتی تمام رأس‌ها در لیست مجاورت u آزمایش شدند، در خط ۱۸ u سیاه می‌شود. خصوصیت حلقه برقرار می‌ماند چرا که هرگاه یک رأس خاکستری می‌شود (خط ۱۴)، به انتهای صف هم اضافه می‌شود (خط ۱۷)، و هرگاه یک رأس از صف خارج می‌شود (خط ۱۱)، رنگ آن هم به سیاه تغییر پیدا می‌کند (خط ۱۸).

نتیجه‌ی جستجوی سطح اول می‌تواند به ترتیب ملاقات همسایه‌های یک رأس در خط ۱۲ بستگی داشته باشد: درخت سطح اول ممکن است تغییر کند، ولی فاصله‌ی d محاسبه شده توسط الگوریتم همیشه ثابت است. (تمرین ۲۲-۲-۵ را ببینید.)

تحلیل مرتبه‌ی زمانی

قبل از اثبات خصوصیات مختلف جستجوی سطح اول، آنالیز مرتبه‌ی زمانی الگوریتم روی یک گراف $G=(V,E)$ را دنبال خواهیم کرد، که نسبتاً ساده‌تر است. برای این کار از تحلیل متراکم استفاده می‌کنیم، که آن را در بخش ۱۷–۱ دیدیم. بعد از مقداردهی اولیه هیچ وقت رأسی سفید نمی‌شود، که تضمین می‌کند هر رأس حداکثر یک بار در صف قرار می‌گیرد و حداکثر یک بار از صف خارج می‌شود. عملیات درج و حذف از مرتبه‌ی زمانی $O(1)$ هستند، پس کل زمان صرف شده برای عملیات صف $O(V)$ است. از آنجایی که لیست مجاورت هر رأس تنها وقتی پیمایش می‌شود که آن رأس از صف خارج شده باشد، لیست مجاورت حداکثر یک بار پیمایش می‌شود. چون مجموع طول طول تمام لیست‌های پیوندی $O(E)$ است، پس کل زمان پیمایش لیست‌ها از مرتبه‌ی $O(E)$ خواهد بود. سربار مقداردهی اولیه از مرتبه‌ی زمانی $O(V)$ است، بنابراین مجموع زمان اجرای BFS برابر است با $O(E+V)$، و جستجوی سطح اول نسبت به لیست مجاورت گراف، در زمان خطی اجرا می‌شود.

کوتاه‌ترین مسیرها

در ابتدای این بخش فهمیدیم که جستجوی سطح اول فاصله‌ی تمام رأس‌های قابل دسترسی را در یک گراف $G=(V,E)$ از یک مبدأ داده شده‌ی $s \in V$ محاسبه می‌کند. تعریف می‌کنیم *طول کوتاه‌ترین مسیر*، $\delta(s,v)$ از s به v برابر است با کمینه‌ی تعداد یال‌ها در هر مسیری از رأس s به رأس v؛ اگر هیچ مسیری از s به v موجود نباشد، آنگاه $\delta(s,v)=\infty$. یک مسیر با طول $\delta(s,v)$ از s به v را یک

کوتاه‌ترین مسیر[1] از s به v می‌نامیم. قبل از اثبات این که جستجوی سطح اول طول کوتاه‌ترین مسیرها را پیدا می‌کند، یک ویژگی مهم طول کوتاه‌ترین مسیر را بررسی می‌کنیم.

لم ۲۲-۱ فرض کنید $G = (V, E)$ یک گراف جهت‌دار یا بدون جهت باشد، و $s \in V$ یک رأس دلخواه. آنگاه به ازای هر یال $(u, v) \in E$ داریم:

$$\delta(s, v) = \delta(s, u) + 1$$

اثبات اگر u از s قابل دسترس باشد، پس v هم از s قابل دسترس است. در این حالت کوتاه‌ترین مسیر از s به v نمی‌تواند از کوتاه‌ترین مسیر از s به u، به علاوه‌ی (u, v) بلندتر باشد، و بنابراین نامساوی بالا برقرار است. اگر u از s قابل دسترس نباشد، آنگاه $\delta(s, u) = \infty$ و باز هم نامساوی برقرار است. ■

اکنون می‌خواهیم نشان دهیم که BFS به درستی $v.d = \delta(s, v)$ را برای هر رأس $v \in V$ محاسبه می‌کند. ابتدا نشان می‌دهیم که $v.d$ از بالا $\delta(s, v)$ را محدود می‌کند.

لم ۲۲-۲ فرض کنید که $G = (V, E)$ یک گراف جهت‌دار یا بدون جهت باشد، و همچنین فرض کنید که BFS روی G از یک رأس داده شده‌ی $s \in V$ اجرا شده باشد. آنگاه پس از اتمام اجرا به ازای هر $v \in V$، مقدار $v.d$ محاسبه شده توسط BFS در نامساوی $v.d \geq \delta(s, v)$ صدق می‌کند.

اثبات از استقرا روی تعداد عملیات ENQUEUE استفاده می‌کنیم. حکم استقرا این است که به ازای $v \in V$ داریم $v.d \geq \delta(s, v)$.

پایه‌ی استقرا دقیقاً بعد از زمانی است که s در خط ۹ در صف درج می‌شود. حکم استقرا در اینجا برقرار است، چرا که $s.d = 0 = \delta(s, s)$ و به ازای $v \in V - \{s\}$ داریم $v.d = \infty \geq \delta(s, v)$.

در هر مرحله‌ی استقرا، فرض کنید که یک رأس سفید u هنگام جستجوی لیست مجاورت رأس u کشف شده است. بنابر فرض استقرا، $u.d \geq \delta(s, u)$، و بنابر مقداردهی خط ۱۵ و لم ۲۲-۱ داریم:

$$\begin{aligned} v.d &= u.d + 1 \\ &\geq \delta(s, u) + 1 \\ &\geq \delta(s, v) \end{aligned}$$

سپس رأس v در صف درج می‌شود و هرگز دوباره درج نمی‌شود، چرا که قبل از درج شدن خاکستری می‌شود، و بلوک then در خطوط ۱۴-۱۷ فقط برای رأس‌های سفید اجرا می‌شود. بنابراین مقدار $v.d$ دیگر تغییر نخواهد کرد، و حکم استقرا ثابت است. ■

---

[1] در فصل‌های ۲۴ و ۲۵ یادگیری خود از کوتاه‌ترین مسیرها را به گراف‌های وزن‌دار گسترش می‌دهیم، که در آن هر یال یک وزن حقیقی مقدار دارد، و وزن یک مسیر برابر است با مجموع وزن یال‌های تشکیل‌دهنده‌ی آن مسیر. گراف‌های در نظر گرفته شده در فصل حاضر بدون وزن هستند، یا به طور معادل، در آن‌ها وزن تمام یال‌ها واحد است.

برای اثبات این که $v.d = \delta(s, v)$، ابتدا باید به طور دقیق‌تر مشخص کنیم که صف Q در زمان اجرای BFS چگونه عمل می‌کند. لم زیر نشان می‌دهد که در کل زمان اجرا، حداکثر دو مقدار متفاوت d در Q وجود دارد.

لم ۲۲-۴ فرض کنید در زمان اجرای BFS روی یک گراف $G = (V, E)$، صف Q شامل رأس‌های $\langle v_1, v_2, ..., v_r \rangle$ می‌شود، که v_1 سر صف و v_r انتهای صف را نشان می‌دهد. آنگاه $v_r.d \leq v_1.d + 1$ و به ازای $i = 1, 2, ..., r-1$ داریم $v_i.d \leq v_{i+1}.d$.

اثبات اثبات به وسیله‌ی استقرا روی تعداد عملیات صف انجام می‌شود. ابتدا وقتی صف فقط حاوی s است، حکم استقرا برقرار است.

برای اثبات گام استقرا باید نشان دهیم که لم بالا هم پس از درج در صف برقرار باقی می‌ماند و هم پس از حذف از صف. اگر سر صف v_1 از صف خارج شود، بعد از آن v_2 سر صف خواهد بود. (اگر صف خالی شود، لم بالا پابرجا خواهد بود.) بنابر فرض استقرا، $v_1.d \leq v_2.d$. ولی در این صورت داریم $v_r.d \leq v_1.d + 1 \leq v_2.d + 1$، و بقیه‌ی نامساوی‌ها هم بدون تغییر خواهند ماند. پس با حذف v_2 از سر صف، لم همچنان برقرار خواهد بود.

برای این که درک کنیم هنگام درج یک رأس چه رخ می‌دهد، نیاز به توجه دقیق تر به کد داریم. وقتی یک رأس v را در خط ۱۷ در BFS در صف درج می‌کنیم، تبدیل به v_{r+1} می‌شود. در آن زمان، رأس u که لیست مجاورت آن در حال بررسی است را از صف Q خارج کرده‌ایم، و بنابر فرض استقرا برای ابتدای جدید صف داریم $v_1.d \geq u.d$. همچنین بنابر فرض استقرا داریم $v_r.d \leq u.d + 1$. پس $v_{r+1}.d = v.d = u.d + 1 \geq v_r.d$ و بقیه‌ی نامساوی‌ها بدون تغییر باقی می‌مانند. بنابراین زمان درج یک رأس در صف لم بالا برقرار است. ■

نتیجه‌ی زیر نشان می‌دهد که مقدار d زمانی که رأس‌ها در صف درج می‌شوند، صعودی است.

نتیجه‌ی ۲۲-۴ فرض کنید رأس‌های v_i و v_j هنگام اجرای BFS در صف درج شده‌اند، و v_i قبل از v_j درج شده است. آنگاه وقتی v_j درج می‌شود، داریم $v_i.d \leq v_j.d$.

اثبات مستقیماً از لم ۲۲-۳ و این خاصیت که هر رأس حداکثر یک بار در هنگام اجرای BFS یک مقدار غیر بی‌نهایت دریافت می‌کند. ■

اکنون می‌توانیم اثبات کنیم که جستجوی سطح اول به درستی فاصله‌ی کوتاه‌ترین مسیرها را پیدا می‌کند.

قضیه‌ی ۲۲-۵ (درستی جست‌وجوی سطح اول) فرض کنید $G=(V,E)$ یک گراف جهت‌دار یا بدون جهت باشد، و BFS روی G از یک رأس مبدأ $s\in V$ اجرا شده است. آنگاه در زمان اجرای آن BFS تمام رأس‌های $v\in V$ که از مبدأ s قابل دسترس هستند را کشف می‌کند، و پس از اتمام آن برای هر $v\in V$ داریم $\delta(s,v)$. به علاوه، برای تمام رأس‌های $v\neq s$ که از s قابل دسترس هستند، یکی از کوتاه‌ترین مسیرهای از s به v، برابر است با کوتاه‌ترین مسیر از s به $v.\pi$ به علاوه‌ی یال $(v.\pi,v)$.

اثبات فرض کنید این طور نباشد، یعنی رأسی وجود داشته باشد که مقدار دریافتی d آن برابر با فاصله‌ی کوتاه‌ترین مسیر آن نباشد. فرض می‌کنیم v با کوتاه‌ترین مسیر $\delta(s,v)$ رأسی باشد که مقدار دریافتی d آن صحیح نیست؛ بدیهتاً $v\neq s$. بنا بر لم ۲۲-۲، $v.d\geq\delta(s,v)$، پس داریم $v.d>\delta(s,v)$. رأس v باید از s قابل دسترس باشد، چرا که اگر این طور نباشد آنگاه خواهیم داشت $\delta(s,v)=\infty\geq v.d$. فرض می‌کنیم u رأس بلافاصله ماقبل v در کوتاه‌ترین مسیر از s به v باشد، به طوری که $\delta(s,v)=d(s,u)+1$. به خاطر روش انتخاب v و از آنجایی که $\delta(s,u)<\delta(s,v)$، داریم $u.d=\delta(s,u)$. از کنار هم قرار دادن این خصوصیات داریم:

$$v.d>\delta(s,v)=\delta(s,u)+1=u.d+1 \qquad (۱-۲۲)$$

اکنون زمانی را در نظر بگیرید که BFS در خط ۱۱ رأس u را برای حذف از Q انتخاب می‌کند. در این زمان رأس v یا سفید است، یا خاکستری و یا سیاه. نشان خواهیم داد که در هر سه حالت نامساوی ۲۲-۱ به تناقض می‌رسد. اگر v سیاه باشد، یعنی قبلاً از صف حذف شده است و بنابر نتیجه‌ی ۲۲-۴ داریم $v.d\leq u.d$، که دوباره نامساوی ۲۲-۱ را نقض می‌کند. اگر v خاکستری باشد، یعنی هنگام حذف رأسی مانند w از صف رنگ شده است، که w قبل از v از صف حذف شده است و $v.d=w.d+1$. با این حال از نتیجه‌ی ۲۲-۴ داریم $w.d\leq u.d$، پس $v.d\leq u.d+1$، و یک بار دیگر نامساوی ۲۲-۱ نقض می‌شود.

پس نتیجه‌ای که می‌گیریم این است که برای تمام $v\in V$ داریم $v.d=\delta(s,v)$. تمام رأس‌های قابل دسترس از s باید کشف شوند، چرا که در غیر این صورت مقدار d آن‌ها بی‌نهایت خواهد بود. برای روشن‌تر شدن اثبات قضیه توجه کنید که اگر $v.\pi=u$، آنگاه $v.d=u.d+1$. بنابراین می‌توانیم با پیمودن یک کوتاه‌ترین مسیر از s به $v.\pi$ و سپس پیمودن یال $(v.\pi,v)$، به یک کوتاه‌ترین مسیر از s به v برسیم. ■

درخت‌های سطح اول

همان طور که در شکل ۲۲-۳ مشخص شده است، زیرروال BFS هنگام جستجوی گراف یک درخت سطح اول می‌سازد. این درخت توسط فیلد π در هر رأس مشخص شده است. به صورت ساده‌تر برای یک گراف $G=(V,E)$ با مبدأ s، *زیرگراف عناصر ماقبل* (predecessor subgraph) گراف G را به

صورت $G_\pi = (V_\pi, E_\pi)$ تعریف می‌کنیم، که در آن

$$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$$

و به علاوه

$$E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\}$$

زیرگراف عناصر ماقبل G_π، یک *درخت سطح اول* است اگر V_π شامل تمام رأس‌های قابل دسترس از s باشد، و برای تمام $v \in V_\pi$ یک مسیر یکتا از s به v در G_π موجود باشد، که این مسیر همچنین یک کوتاه‌ترین مسیر از s به v در G باشد. در واقع درخت سطح اول یک درخت است، چرا که همبند است و داریم $|E_\pi| = |V_\pi| - ۱$ (قضیه‌ی ب-۲ را ببینید). یال‌های درون E_π *یال‌های درخت* نام دارند.

لم زیر نشان می‌دهد که بعد از اجرای BFS از یک مبدأ s روی یک گراف G، زیرگراف عناصر ماقبل یک درخت سطح اول است.

لم ۲۲-۶ زیرروال BFS پس از اجرا بر روی یک گراف جهت‌دار یا بدون جهت $G = (V, E)$، π را به گونه‌ای می‌سازد که زیرگراف عناصر ماقبل $G_\pi = (V_\pi, E_\pi)$ یک درخت سطح اول باشد.

اثبات خط ۱۶ از BFS خصیصه‌ی $v.\pi$ با u مقداردهی می‌کند اگر و فقط اگر $(u, v) \in E$ و $\delta(s, v) < \infty$ – یعنی اگر v از s قابل دسترس باشد – و بنابراین V_π شامل تمام رأس‌هایی از V می‌شود که از s قابل دسترس باشند. از آنجایی که G_π یک درخت را تشکیل می‌دهد، بنابر قضیه‌ی ب-۲، شامل یک مسیر ساده‌ی یکتا از s به هر یک از رأس‌های درون V_π می‌شود. با به کار بردن قضیه‌ی ۲۲-۵ به صورت استقرایی، نتیجه می‌گیریم که هر یک این مسیرها یک کوتاه‌ترین مسیر هستند. ■

روال زیر رأس‌های روی یک کوتاه‌ترین مسیر از s به v را چاپ می‌کند، با فرض این که BFS به منظور محاسبه‌ی درخت کوتاه‌ترین مسیرها قبلاً اجرا شده است.

```
PRINT-PATH(G, s, v)
1  if v = s
2      print s
3  elseif v.π = NIL
4      print "no path from" s "to" v "exists"
5  else PRINT-PATH(G, s, v.π)
6      print v
```

این روال در زمان خطی نسبت به رأس‌های مسیر چاپ شده اجرا می‌شود، چرا که هر بار تابع روی یک میسر با یک یال کمتر به صورت بازگشتی فراخوانی می‌شود.

تمرین‌ها

۲۲-۲-۱ مقادیر d و π را پس از اجرای جستجوی سطح اول بر روی گراف جهت‌دار شکل ۲۲-۲(الف)، با رأس ۳ به عنوان رأس مبدأ نشان دهید.

۲۲-۲-۲ مقادیر d و π را پس از اجرای جستجوی سطح اول بر روی گراف بدون جهت شکل ۲۲-۳، با رأس u به عنوان رأس مبدأ نشان دهید.

۲۲-۲-۳ نشان دهید که یک بیت برای ذخیره‌ی رنگ هر رأس کافی است، با این بحث که با حذف خطوط ۵ و ۱۴ از رویه‌ی BFS نتیجه‌ی آن تغییری نخواهد کرد.

۲۲-۲-۴ اگر گراف ورودی BFS به صورت ماتریس مجاورت نمایش داده شده باشد، و الگوریتم طوری اصلاح شده باشد که روی این نوع ورودی عمل کند، زمان اجرای آن چقدر خواهد بود؟

۲۲-۲-۵ بحث کنید که در یک جستجوی سطح اول، مقدار $u.d$ نسبت داده شده به رأس u مستقل از ترتیب قرار گرفتن رأس‌ها در لیست‌های مجاورت است. با استفاده از شکل ۲۲-۳ به عنوان نمونه، نشان دهید که درخت سطح اول ساخته شده توسط BFS می‌تواند به ترتیب قرار گرفتن رأس‌ها در لیست‌های مجاورت بستگی داشته باشد.

۲۲-۲-۶ مثالی از یک گراف $G = (V, E)$، یک رأس مبدأ $s \in V$، و یک مجموعه از یال‌های درخت $E_\pi \subseteq E$ بزنید که برای هر رأس $v \in V$، مسیر ساده‌ی یکتا از s به v در گراف (V, E_π)، یک کوتاه‌ترین مسیر در G باشد، ولی با هر ترتیبی که رأس‌های G را در لیست‌های پیوندی قرار دهیم، نتوان مجموعه‌ی یال‌های E_π را با اجرای BFS بر روی G تولید کرد.

۲۲-۲-۷ دو نوع کشتی‌گیر حرفه‌ای وجود دارد: «افراد خوب» و «افراد بد». بین هر جفت از کشتی‌گیران حرفه‌ای، ممکن است مسابقه‌ای انجام گیرد و یا انجام نگیرد. فرض کنید n کشتی‌گیر حرفه‌ای داریم، به علاوه‌ی یک لیست از r جفت کشتی‌گیر که قرار است با یکدیگر مسابقه دهند. یک الگوریتم از مرتبه‌ی زمانی $O(n+r)$ ارائه دهید که تشخیص دهد آیا امکان پذیر است که کشتی‌گیران را به دو دسته‌ی خوب و بد تقسیم کنیم، به طوری که هر مسابقه بین یک کشتی‌گیر خوب و یک کشتی‌گیر بد انجام گیرد. اگر چنین تقسیم‌بندی امکان پذیر باشد، الگوریتم شما باید آن را تولید کند.

★ ۲۲-۲-۸ *قطر* یک درخت $T = (V, E)$ به صورت $\max_{u,v \in V} \delta(u, v)$ تعریف می‌شود، یعنی قطر درخت بلندترین مسیر بین تمام کوتاه‌ترین مسیرها است. یک الگوریتم کارامد ارائه دهید که قطر درخت را محاسبه می‌کند، و سپس زمان اجرای الگوریتم خود را به دست آورید.

۲۲-۲-۹ فرض کنید $G = (V, E)$ یک گراف همبند و بدون جهت باشد. یک الگوریتم از مرتبه‌ی زمانی $O(V + E)$ ارائه دهید که یک مسیر در G پیدا کند که از هر یال E دقیقاً یک بار در هر جهت بگذرد. توضیح دهید که اگر تعداد نامحدودی سکه داشته باشید چطور می‌توانید راه خود را از درون یک مارپیچ (maze) پیدا کنید.

۲۲-۳ جستجوی عمق اول

همان طور که از نام جستجوی عمق اول پیدا است، استراتژی آن این است که تا حد ممکن «عمیق‌تر» در گراف پیش برود. در جستجوی عمق اول، در هر لحظه رأس‌های همسایه‌ی آخرین رأس کشف شده‌ی v مورد بررسی قرار می‌گیرند، و اگر آن رأس همسایه‌ای داشت که کشف نشده بود همین کار در مورد آن رأس کشف نشده انجام می‌شود. وقتی تمام همسایه‌های رأس v کشف شده شدند، الگوریتم به رأسی باز می‌گردد که v از طریق آن کشف شده بود. اگر رأس کشف نشده‌ای باقی مانده باشد، دوباره آن رأس به عنوان مبدأ انتخاب شده و جستجو از آن رأس ادامه می‌یابد. تمام این عملیات آن قدر تکرار می‌شود که تمام رأس‌های قابل دسترس کشف شوند.[۱]

همانند جستجوی سطح اول، هر گاه یک رأس v هنگام بررسی لیست مجاورت یک رأس تازه کشف شده‌ی u، کشف شود، فیلد $v.\pi$ با u مقداردهی می‌شود. ولی بر خلاف جستجوی سطح اول که زیرگراف عناصر ماقبل آن یک درخت را تشکیل می‌دهد، زیرگراف عناصر ماقبل تولید شده توسط جستجوی عمق اول ممکن است از چندین درخت تشکیل شده باشد، چرا که ممکن است جستجو از چند مبدأ مختلف تکرار شود. بنابراین *زیرگراف عناصر ماقبل* (predecessor subgraph) جستجوی عمق اول تاحدودی با زیرگراف عناصر ماقبل جستجوی سطح اول متفاوت است: فرض می‌کنیم $G_\pi = (V, E_\pi)$، که در آن

$$E_\pi = \{(v.\pi, v) : v \in V \text{ and } v.\pi \neq \text{NIL}\}$$

زیرگراف عناصر ماقبل جستجوی عمق اول یک *جنگل عمق اول* (depth-first forest) را تشکیل می‌دهد که از چندین *درخت عمق اول* (depth-frist tree) تشکیل شده است. یال‌های E_π، *یال‌های درخت* نام دارند.

مانند جستجوی سطح اول، رأس‌ها هنگام جستجو رنگ‌آمیزی می‌شوند تا وضعیت آن‌ها مشخص شود. هر رأس در ابتدا سفید رنگ است، بعد از کشف شدن خاکستری می‌شود، و بعد از اتمام کار،

---

[۱] ممکن است تصادفی به نظر برسد که جستجوی سطح اول به یک مبدأ محدود است، ولی جستجوی عمق اول ممکن است از مبدأهای مختلفی جستجو را انجام دهد. با این حال که از نظر مفهومی، جستجوی سطح اول می‌تواند از چند مبدأ جستجو را انجام دهد، و جستجوی عمق اول می‌تواند به یک مبدأ محدود شود، رویکرد ما بر پایه‌ی نحوه‌ی معمول استفاده‌ی نتایج این جستجوها است. از جستجوی سطح اول معمولاً برای یافتن فاصله‌ی کوتاه‌ترین مسیرها (و زیرگراف عناصر ماقبل مربوطه) برای یک مبدأ داده شده استفاده می‌شود. جستجوی عمق اول معمولاً یک زیرروال است که در یک الگوریتم دیگر فراخوانی می‌شود، همان طور که بعداً در این فصل خواهیم دید.

یعنی وقتی تمام لیست پیوندی آن آزمایش شد، سیاه می‌شود. این تکنیک تضمین می‌کند که هر رأس تنها در یک درخت عمق اول ظاهر می‌شود، و درخت‌ها به هم پیوسته نیستند.

جستجوی عمق اول علاوه بر ساختن درخت عمق اول، روی هر رأس *زمان‌گذاری* می‌کند. برای هر رأس v دو زمان نگه داری می‌شود: اولین زمان، $v.d$، زمان کشف (خاکستری شدن) رأس v را نشان می‌دهد، و دومین زمان، $v.f$، زمان کامل شدن آزمایش لیست مجاورت رأس v و سیاه شدن آن را. این زمان‌ها در بسیاری از الگوریتم‌های گراف مورد استفاده قرار می‌گیرند و برای بررسی رفتار جستجوی عمق اول مفید هستند.

زیرروال DFS زیر، زمان کشف رأس u را در $u.d$ و زمان اتمام کار با آن را در $u.f$ نگه‌داری می‌کند. این زمان‌ها بین ۱ و $2|V|$ هستند، چرا که برای هر یک از $|V|$ رأس یک رویداد کشف و یک رویداد اتمام کار داریم. برای هر رأس u داریم

$$u.d < u.f \qquad (۲۲-۲)$$

رأس u قبل از زمان $u.d$، WHITE، بین زمان‌های $u.d$ و $u.f$، GRAY، و بعد از آن BLACK است.

شبه‌کد زیر الگوریتم جستجوی عمق اول اولیه است. گراف ورودی G ممکن است جهت‌دار یا بدون جهت باشد. متغیر *time* یک متغیر جهانی است که از آن برای زمان گذاری استفاده می‌کنیم.

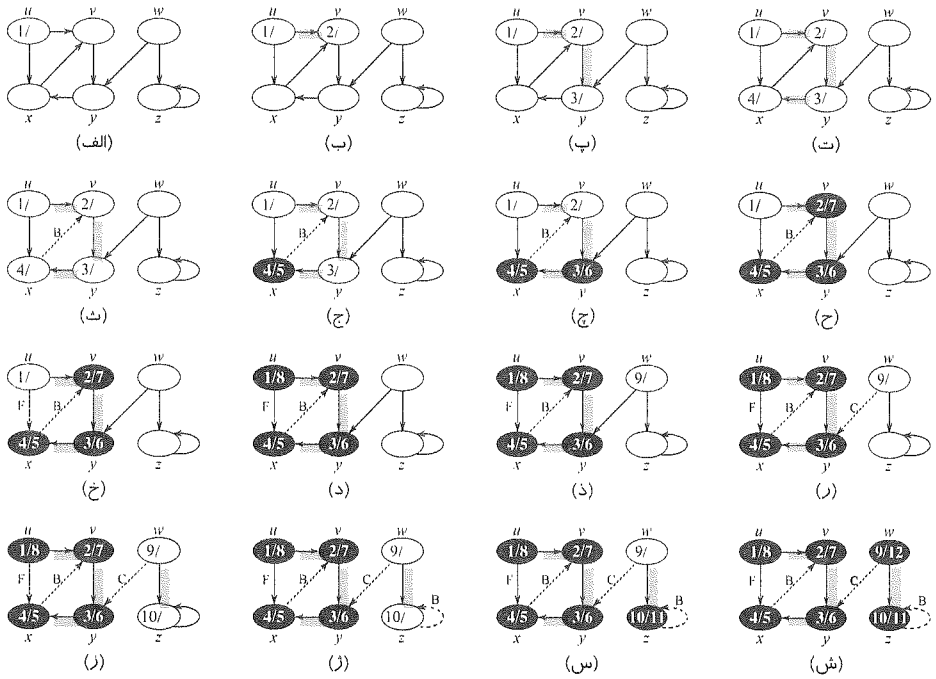
```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color = WHITE
7          DFS-VISIT(G,u)

DFS-VISIT(G,u)
1  time = time +1              // White vertex u has just been discovered.
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]       // Explore edge(u, v).
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G,v)
8  u.color = BLACK             // Blacken u; it is finished.
9  time = time +1
10 u.f = time
```

شکل ۲۲-۴ روند DFS را بر روی گراف شکل ۲۲-۲ نشان می‌دهد.

رویه‌ی DFS به صورت زیر کار می‌کند. خطوط ۱-۳ تمام رأس‌ها را با سفید رنگ‌آمیزی و فیلد π

آن‌ها را با NIL مقداردهی می‌کنند. خط ۴ شمارنده‌ی زمان را بازنشانی می‌کند. خطوط ۵–۷ هر رأس V را چک می‌کنند، و وقتی یک رأس سفید پیدا کردند، با استفاده از DFS-VISIT آن رأس را ملاقات می‌کنند. هر بار که DFS-VISIT(u) در خط ۷ فراخوانی می‌شود، رأس u تبدیل به ریشه‌ی یک درخت جدید در جنگل عمق اول می‌شود. وقتی DFS بازگشت می‌کند، به هر رأس u یک *زمان کشف* $u.d$ و یک *زمان پایان* $u.f$ نسبت داده شده است.

در هر فراخوانی DFS-VISIT(u)، رأس u در ابتدا سفید است. خط ۱ متغیر جهانی *time* را افزایش می‌دهد، خط ۲ مقدار جدید *time* را به عنوان زمان کشف $u.d$ ذخیره می‌کند، و خط ۳ رأس u را با خاکستری رنگ‌آمیزی می‌کند. در خطوط ۴–۷ هر رأس v که مجاور u باشد بررسی می‌شود، و اگر سفید بود به صورت بازگشتی ملاقات بر روی آن رأس انجام می‌گیرد. زمانی که در خط ۴ یک رأس $v \in Adj[u]$ بررسی می‌شود، می‌گوییم یال (u, v) توسط جستجوی عمق اول *کشف* (explore) شده است. نهایتاً بعد از این که تمام یال‌های خروجی u کشف شدند، در خطوط ۸–۱۰ رأس u با سیاه رنگ‌آمیزی شده و زمان پایان $u.f$ ذخیره می‌شود.



شکل ۲۲-۴ روند الگوریتم جستجوی عمق اول DFS بر روی یک گراف جهت‌دار. همین طور که گذر از یال‌ها انجام می‌شود، یا سایه‌دار می‌شوند (اگر یال درخت باشند) و یا نقطه چین می‌شوند (اگر یال درخت نباشند). یال‌های غیر درخت با حروف B، C، یا F برچسب گذاری شده‌اند، بسته به این که به ترتیب یال عقبی، ضربدری، یا جلویی باشند. زمان‌گذاری رأس‌ها به صورت زمان کشف/زمان پایان انجام شده است.

توجه کنید که نتیجه‌ی جستجوی عمق اول ممکن است به ترتیبی که رأس‌ها در خط ۵ رویه‌ی DFS بررسی می‌شوند، و همچنین به ترتیبی که در خط ۴ رویه‌ی DFS-VISIT همسایه‌های یک رأس ملاقات می‌شوند، بستگی داشته باشد. این ترتیب‌های مختلف ملاقات در عمل مشکلی ایجاد نمی‌کنند، چرا که از هر نتیجه‌ای که جستجوی عمق اول تولید می‌کند می‌توان به صورت مؤثر استفاده کرد، و معمولاً نتایج هم یکسان است.

زمان اجرای DFS چقدر است؟ حلقه‌های خطوط ۱-۳ و ۵-۷ رویه‌ی DFS در زمان $\theta(V)$ اجرا می‌شوند، جدا از زمان اجرای فراخوانی‌های DFS-VISIT. همان طور که برای جستجوی سطح اول انجام دادیم، در این جا هم از تحلیل متراکم استفاده می‌کنیم. رویه‌ی DFS-VISIT برای هر رأس $v \in V$ دقیقاً یک بار فراخوانی می‌شود، چرا که DSF-VISIT فقط بر روی رأس‌های سفید احضار می‌شود، و اولین کاری که انجام می‌دهد این است که رأس را با خاکستری رنگ‌آمیزی می‌کند. در طول یک اجرای DFS-VISIT(v)، حلقه‌ی خطوط ۴-۷ به تعداد $|Adj[v]|$ بار اجرا می‌شود. از آن جایی که

$$\sum_{v \in V} |Adj[v]| = \theta(E)$$

کل هزینه‌ی اجرای خطوط ۴-۷ DFS-VISIT برابر است با $\theta(E)$. بنابراین زمان اجرای DFS، $\theta(V+E)$ خواهد بود.

خصوصیات جستجوی عمق اول

جستجوی عمق اول اطلاعات مفیدی در مورد ساختار یک گراف به دست می‌دهد. احتمالاً پایه‌ای‌ترین خصوصیت جستجوی عمق اول این است که زیرگراف ماقبل G_π هم یک جنگل از درخت‌ها را تشکیل می‌دهد، چرا که ساختار درخت‌های عمق اول دقیقاً مشابه ساختار فراخوانی‌های بازگشتی DFS-VISIT است. یعنی $u = v.\pi$ اگر و فقط اگر DFS-VISIT(v) در طول جستجوی لیست مجاورت u فراخوانی شده باشد. به علاوه رأس v در جنگل عمق اول نوه‌ی u است اگر و فقط اگر در زمان کشف v رأس u خاکستری بوده باشد.

یک خصوصیت مهم دیگر جستجوی عمق اول این است که زمان‌های کشف و پایان دارای ساختار پرانتزی هستند. اگر کشف رأس u را با یک پرانتز باز «(»، و پایان کار آن را با یک پرانتز بسته «)» نشان دهیم، آن گاه دنباله‌ی کشف‌ها و پایان‌ها یک عبارت پرانتزی خوش‌فرم را تشکیل می‌دهند، بدین صورت که پرانتزها به شکل مناسب تودرتو هستند. به عنوان مثال جستجوی عمق اول شکل ۲۲-۵ (الف) متناظر است با پرانتز گذاری شکل ۲۲-۵ (ب). روشی دیگر برای تعیین وضعیت پرانتزها در قضیه‌ی ۲۲-۷ داده شده است.

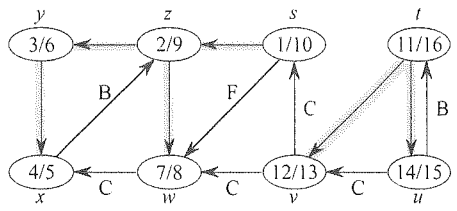
قضیه‌ی ۲۲-۷ (قضیه‌ی پرانتزبندی)

در هر جستجوی عمق اول در یک گراف (جهت‌دار یا بدون جهت) $G = (V, E)$، برای هر دو رأس u و v، دقیقاً یکی از سه حالت زیر برقرار است:

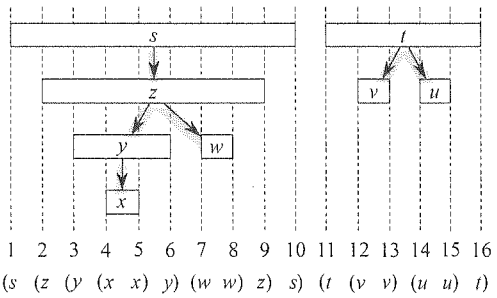
- بازه‌های $[u.d, u.f]$ و $[v.d, v.f]$ کاملاً از هم جدا هستند، و در جنگل عمق اول، u و v هیچ کدام نواده‌ی دیگری نیستند،

- بازه‌ی $[u.d, u.f]$ کاملاً درون بازه‌ی $[v.d, v.f]$ قرار دارد، و در جنگل عمق اول u نواده‌ی v است، یا
- بازه‌ی $[u.d, u.f]$ کاملاً درون بازه‌ی $[v.d, v.f]$ قرار دارد، و در جنگل عمق اول v نواده‌ی u است.

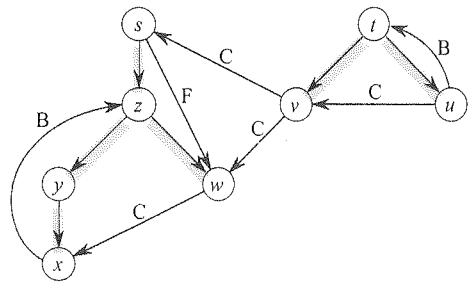
(الف)



(ب)



(پ)



شکل ۲۲-۵ خصوصیات جستجوی عمق اول. (الف) نتیجه‌ی جستجوی عمق اول بر روی یک گراف جهت‌دار. رأس‌ها زمان‌گذاری شده‌اند، و نوع یال‌ها همان طور که در شکل ۲۲-۴ گفته شده، مشخص شده است. (ب) بازه‌هایی برای زمان کشف و پایان هر رأس متناظر است با پرانتزگذاری نشان داده شده. هر مستطیل بازه‌ی متناظر با زمان‌های کشف و پایان رأس مربوطه را مشخص می‌کند. یال‌های درخت نشان داده شده‌اند. اگر دو بازه همپوشانی داشته باشند، آن گاه یکی از آن‌ها درون دیگری قرار خواهد داشت، و رأس مربوط به بازه‌ی کوچک‌تر نواده‌ی رأس مربوط به بازه‌ی بزرگ‌تر است. (پ) گراف بخش (الف) دوباره کشیده شده که در آن تمام یال‌های درخت و یال‌های جلویی به سمت پایین و درون یک درخت عمق اول و تمام یال‌های عقبی به سمت بالا از یک نواده به یک پدر کشیده شده‌اند.

اثبات با حالتی شروع می‌کنیم که در آن $u.d < v.d$. در این حالت دو موقعیت پیش خواهد آمد، بسته به این که $v.d < u.f$ برقرار باشد یا خیر. اگر $v.d < u.f$، آن گاه v زمانی کشف شده است که u هنوز خاکستری بوده است. این ایجاب می‌کند که v نواده‌ی u باشد. به علاوه از آن جایی که v قبل از u کشف شده بود، قبل از این که جستجو به u بازگشته و آن را تمام کند، تمام یال‌های خارجی آن کشف شده‌اند، و جستجوی v پایان یافته است. بنابراین در این حالت بازه‌ی $[v.d, v.f]$ کاملاً درون بازه‌ی $[u.d, u.f]$ قرار دارد. در موقعیت دیگر $u.f < v.d$، و نامساوی (۲۲-۲) ایجاب می‌کند که بازه‌های $[u.d, u.f]$ و $[v.d, v.f]$ کاملاً مستقل از هم باشند. چون بازه‌ها از یکدیگر جدا هستند، هیچ کدام از رأس‌ها در زمانی که دیگری خاکستری بوده است کشف نشده‌اند، و بنابراین هیچ یک نواده‌ی دیگری نیست.

حالت $v.d < u.d$ مشابه است، که در آن نقش u و v در بحث بالا عوض می‌شود. ■

نتیجه‌ی ۸-۲۲ (تودرتگی بازه‌های نوادگان)

رأس v در جنگل عمق اول یک گراف (جهت‌دار یا بدون جهت) G نواده‌ی رأس u است اگر و فقط اگر $u.d < v.d < v.f < u.f$.

اثبات مستقیماً از قضیه‌ی ۲۲-۷. ■

قضیه‌ی بعد خصوصیت مهم دیگری از حالتی را مشخص می‌کند که در آن یک رأس در جنگل عمق اول نواده‌ی رأسی دیگر است.

قضیه‌ی ۹-۲۲ (قضیه‌ی مسیر سفید)

در جنگل عمق اول یک گراف (جهت‌دار یا بدون جهت) $G = (V, E)$، رأس v نواده‌ی رأس u است اگر و فقط اگر در زمان $u.d$ که در آن u کشف شده است، بتوان از یک مسیر تماماً متشکل از رأس‌های سفید از رأس u به رأس v رسید.

اثبات $\Rightarrow$: اگر $v = u$، آن گاه مسیر از u به v فقط حاوی رأس u است، که وقتی به $u.d$ مقدار می‌دهیم هنوز سفید است. اکنون فرض کنید v نواده‌ی u در جنگل عمق اول است. طبق نتیجه‌ی ۸-۲۲ داریم $u.d < v.d$، و بنابراین در زمان $u.d$ رأس v سفید است. از آن جایی که v می‌تواند هر یک از نوادگان u باشد، تمام رأس‌ها بر روی مسیر ساده‌ی یکتای از u به v در جنگل عمق اول در زمان $u.d$ سفید هستند.

$\Leftarrow$: فرض کنید در زمان $u.d$ رأس v از یک مسیر تماماً متشکل از رأس‌های سفید در جنگل عمق اول، از u قابل دسترس است. بدون از دست دادن کلیت، فرض کنید تمام رأس‌های روی مسیر نواده‌های u می‌شوند. (در غیر این صورت، فرض کنید v نزدیک‌ترین رأس به u در مسیر باشد که نواده‌ی u نمی‌شود.) فرض کنید w رأس ماقبل v در مسیر باشد، به طوری که w یک نواده‌ی u است (در واقع w و u ممکن است یکی باشند) و طبق نتیجه‌ی ۸-۲۲، $w.f \leq u.f$. توجه کنید که کشف v

باید بعد از کشف u، ولی قبل از اتمام بررسی w انجام شده باشد. بنابراین $u.d < v.d < w.f \leq u.f$. آن گاه قضیه‌ی ۲۲-۷ ایجاب می‌کند که بازه‌ی $[v.d, v.f]$ کاملاً درون بازه‌ی $[u.d, u.f]$ قرار گیرد. طبق نتیجه‌ی ۲۲-۸ رأس v باید یک نواده‌ی رأس u باشد.

دسته‌بندی یال‌ها

یکی دیگر از خصوصیات جذاب جستجوی عمق اول این است که از این جستجو می‌توان برای دسته‌بندی یال‌های گراف ورودی $G = (V, E)$ استفاده کرد. این دسته‌بندی می‌تواند برای جمع‌آوری اطلاعات مهمی در مورد گراف مورد استفاده قرار گیرد. به عنوان مثال در بخش بعد خواهیم دید که یک گراف جهت‌دار، بدون دور است اگر و فقط اگر یک جستجوی عمق اول هیچ یال «عقبی» تولید نکند (لم ۲۲-۱۱).

برای یک جنگل عمق اول G_π تولید شده توسط یک جستجوی عمق اول بر روی G می‌توان چهار نوع یال تعریف کرد.

۱. *یال‌های درخت* (tree edge) یال‌های درون جنگل عمق اول G_π هستند. یال (u, v) یک یال درخت است اگر v ابتدا با کشف یال (u, v) کشف شده باشد.
۲. *یال‌های عقبی* (back edge) یال‌هایی مانند (u, v) هستند که یک رأس u را به یک جد v در درخت عمق اول متصل می‌کنند. طوقه‌ها، که در گراف‌های جهت‌دار وجود دارند، به عنوان یال‌های عقبی در نظر گرفته می‌شوند.
۳. *یال‌های جلویی* (forward edge) یال‌های غیر درخت (u, v) هستند که یک رأس u را به یک رأس نواده‌ی v در درخت عمق اول متصل می‌کنند.
٤. *یال‌های ضربدری* (cross edge) تمام یال‌های دیگر هستند. آن‌ها ممکن است رأس‌ها را در یک درخت عمق اول به یکدیگر متصل کنند، در صورتی که یکی از رأس‌ها جد دیگری نباشد، و یا ممکن است بین رأس‌هایی در درخت‌های عمق اول مختلف قرار گیرند.

در شکل‌های ۲۲-۴ و ۲۲-۵ بر روی یال‌ها برچسب‌هایی زده شده است تا نوع آن‌ها مشخص شود. شکل ۲۲-۵(پ) هم نشان می‌دهد که چگونه می‌توان گراف شکل ۲۲-۵(الف) را طوری دوباره کشید که تمام یال‌های درخت و یال‌هایی جلویی در درخت عمق اول به سمت پایین، و یال‌های عقبی به سمت بالا کشیده شوند. هر گرافی را می‌توان بدین صورت کشید.

می‌توان الگوریتم DFS را طوری اصلاح کرد که با گذر از روی یال‌ها بعضی از آن‌ها را دسته‌بندی کند. ایده‌ی اصلی این است وقتی برای اولین بار یک یال (u, v) را کشف می‌کنیم، رنگ رأس v اطلاعاتی در مورد آن یال به ما می‌دهد:

۱. WHITE نشان دهنده‌ی یک یال درخت است،
۲. GRAY نشان دهنده‌ی یک یال عقبی است، و
۳. BLACK نشان دهنده‌ی یک یال جلویی یا یک یال ضربدری است.

حالت اول مستقیماً از توصیف الگوریتم نتیجه می‌شود. برای حالت دوم، مشاهده کنید که رأس‌های خاکستری همیشه یک زنجیره‌ی خطی از نوادگان را تشکیل می‌دهند که متناظر است با پشته‌ی فراخوانی‌های فعال DFS-VISIT؛ تعداد یال‌های خاکستری یکی بیشتر است از عمق آخرین رأس کشف شده در جنگل عمق اول. اکتشاف همیشه از عمیق‌ترین رأس خاکستری ادامه می‌یابد، بنابراین یک یال که به یک رأس خاکستری دیگر می‌رسد، به یک جد رسیده است. حالت سوم متناظر است با تمام حالت‌های باقی مانده؛ تمرین ۲۲-۳-۵ از شما می‌خواهد نشان دهید که چنین یالی یال جلویی است اگر $u.d < v.d$ و یال ضربدری است اگر $u.d > v.d$.

در یک گراف بدون جهت ممکن است ابهاماتی در دسته‌بندی وجود داشته باشد، چرا که (u, v) و (v, u) در واقع یک یال هستند. در چنین حالتی نوع یال، اولین نوعی در لیست دسته‌بندی خواهد بود که می‌توان برای آن به کار برد. به عبارت دیگر (تمرین ۲۲-۳-۶ را ببینید) یال بر این مبنا دسته‌بندی خواهد شد که در هنگام اجرای الگوریتم اول به کدام یک از (u, v) یا (v, u) برخورد می‌کنیم.

اکنون نشان می‌دهیم که یال‌های جلویی و ضربدری هیچ وقت در جستجوی عمق اول یک گراف بدون جهت رخ نخواهند داد.

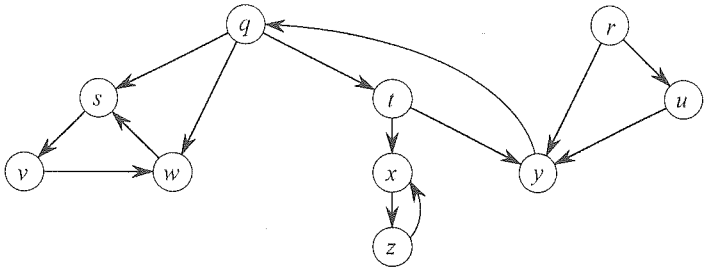
قضیه ۲۲-۱۰ در جستجوی عمق اول یک گراف بدون جهت G، هر یال G یا یک یال درخت است و یا یک یال عقبی.

اثبات فرض کنید (u, v) یک یال دلخواه در G باشد، و همچنین بدون از دست دادن کلیت فرض کنید $u.d < v.d$. آن گاه v باید قبل از این که کار u تمام شود (زمانی که u خاکستری است) کشف شده و پایان یابد، چرا که v در لیست مجاورت u است. اگر یال (u, v) ابتدا در جهت u به v کشف شده باشد، آن گاه v تا آن زمان کشف نشده (سفید) است، چرا که در غیر این صورت باید قبلاً این یال را در جهت v به u کشف کرده باشیم. بنابراین (u, v) به یک یال درخت تبدیل خواهد شد. اگر (u, v) ابتدا در جهت v به u کشف شده باشد، آن گاه (u, v) یک یال عقبی است، چرا که زمانی که (u, v) برای اولین بار کشف می‌شود، u هنوز خاکستری است. ■

در بخش‌های بعد کاربردهای مختلفی از این قضیه‌ها را خواهیم دید.

تمرین‌ها

۲۲-۳-۱ یک جدول ۳ در ۳ با برچسب‌های WHITE، GRAY، و BLACK بر روی ردیف‌ها و ستون‌ها بسازید. در هر خانه‌ی (i, j)، مشخص کنید در هر زمان حین یک جستجوی عمق اول بر روی یک گراف جهت‌دار، آیا ممکن است یک یال از یک رأس با رنگ i به یک رأس با رنگ j وجود داشته باشد یا خیر. برای هر یال ممکن، مشخص کنید که آن یال از کدام نوع می‌تواند باشد. یک جدول مشابه هم برای جستجوی عمق اول بر روی یک گراف بدون جهت بسازید.



شکل ۲۲-۶ یک گراف جهت‌دار برای استفاده در تمرین‌های ۲۲-۳-۲ و ۲۲-۵-۲.

۲۲-۳-۲ نشان دهید که جستجوی عمق اول چگونه بر روی شکل ۲۲-۶ کار می‌کند. فرض کنید حلقه‌ی for خطوط ۵-۷ رویه‌ی DFS رأس‌ها را به ترتیب الفبا در نظر می‌گیرد، و همچنین هر لیست مجاورت به صورت الفبایی مرتب شده است. زمان‌های کشف و پایان را برای هر رأس، و نوع یال‌ها را مشخص کنید.

۲۲-۳-۳ ساختار پرانتزی را برای جستجوی عمق اول نشان داده شده در شکل ۲۲-۴ مشخص کنید.

۲۲-۳-۴ نشان دهید که یک بیت برای ذخیره‌سازی رنگ هر رأس کافی است، بدین صورت که بحث کنید نتیجه‌ی رویه‌ی DFS در صورت حذف خط ۳ از DFS-VISIT تغییری نخواهد کرد.

۲۲-۳-۵ نشان دهید که یال (u, v)

۱. یک یال درخت و یا یک یال جلویی است اگر و فقط اگر $u.d < v.d < v.f < u.f$،
۲. یک یال عقبی است اگر و فقط اگر $v.d < u.d < u.f < v.f$، و
۳. یک یال ضربدری است اگر و فقط اگر $v.d < v.f < u.d < u.f$.

۲۲-۳-۶ نشان دهید که در یک گراف بدون جهت، دسته‌بندی یک یال (u, v) به عنوان یک یال درخت و یا یک یال عقبی بسته به این که در جستجوی عمق اول ابتدا به (u, v) برخورد شده است و یا به (v, u)، معادل است با دسته‌بندی آن بر حسب اولویت انواع در رویکرد دسته‌بندی.

۲۲-۳-۷ با استفاده از یک پشته، رویه‌ی DFS را به صورت غیر بازگشتی بازنویسی کنید.

۲۲-۳-۸ یک مثال نقض برای این حدس ارائه کنید که اگر در یک گراف جهت‌دار G یک مسیر از u به v وجود داشته باشد، و اگر در جستجوی عمق اول G داشته باشیم $u.d < v.d$، آن گاه v در جنگل عمق اول ساخته شده یک نواده‌ی u است.

۲۲-۳-۹ یک مثال نقض برای این حدس ارائه کنید که اگر در یک گراف بدون جهت G یک مسیر از u به v وجود داشته باشد، آن گاه در نتیجه‌ی جستجوی عمق اول باید داشته باشیم $v.d \leq u.f$.

۲۲-۳-۱۰ شبه‌کد جستجوی عمق اول را طوری اصلاح کنید که تمام یال‌ها را در گراف جهت‌دار G به همراه نوع آن‌ها چاپ کند. نشان دهید در صورتی که G بدون جهت باشد، چه اصلاحات دیگری (در صورت وجود) باید انجام شود.

۲۲-۳-۱۱ نشان دهید که چطور ممکن است یک رأس u در یک گراف جهت‌دار در یک درخت عمق اول قرار گیرد که فقط شامل u است، با این که u در G هم یال ورودی دارد و هم یال خروجی.

۲۲-۳-۱۲ نشان دهید که از یک جستجوی عمق اول بر روی یک گراف G می‌توان برای تعیین مؤلفه‌های همبندی G استفاده کرد، و همچنین تعداد درخت‌های عمق اول در جنگل عمق اول G برابر است با تعداد مؤلفه‌های همبندی G. به طور دقیق‌تر، نشان دهید که چگونه می‌توان جستجوی عمق اول طوری را اصلاح کرد که به هر رأس v یک برچسب صحیح $v.cc$ بین ۱ و k نسبت دهد، که در آن k تعداد مؤلفه‌های همبندی G است، و $u.cc = v.cc$ اگر و فقط اگر u و v در یک مؤلفه‌ی همبندی قرار داشته باشند.

★ ۲۲-۳-۱۳ یک گراف جهت‌دار $G = (V, E)$ *تک‌همبند* (singly connected) است اگر $u \rightsquigarrow v$ ایجاب کند که برای تمام رأس‌های $u, v \in V$ حداکثر یک مسیر ساده از u به v وجود دارد. یک الگوریتم بهینه ارائه دهید که تعیین می‌کند یک گراف بدون جهت تک همبند است یا خیر.

۲۲-۴ مرتب‌سازی توپولوژیکی

در این بخش نشان خواهیم داد که چگونه می‌توان از جستجوی عمق اول برای انجام مرتب‌سازی توپولوژیکی بر روی یک گراف جهت‌دار بدون دور، یا یک "dag" استفاده کرد. یک *مرتب‌سازی توپولوژیکی* (topological sort) بر روی گراف $G = (V, E)$ ترتیبی خطی است برای تمام رأس‌های آن به طوری که اگر G حاوی یک یال (u, v) باشد، آن گاه در این ترتیب u قبل از v ظاهر می‌شود. (اگر گراف دور داشته باشد، آن گاه چنین ترتیب خطی ممکن نیست.) می‌توان به مرتب‌سازی توپولوژیکی به صورت یک ترتیب خطی از رأس‌های آن بر روی یک خط افقی نگاه کرد به طوری که تمام یال‌های جهت‌دار از سمت چپ به سمت راست حرکت کنند. بنابراین مرتب‌سازی توپولوژیکی با مرتب‌سازی معمولی که در بخش ۲ آموختیم متفاوت است.

از گراف‌های جهت‌دار بدون دور در بسیاری از کاربردها برای نشان دادن اولویت در رخدادها استفاده می‌شود. شکل ۲۲-۷ مثالی را نشان می‌دهد که در آن پروفسور Bumstead می‌خواهد قبل از رفتن به محل کار لباس خود را بپوشد. پروفسور باید لباس‌هایی را قبل از لباس‌های دیگر بپوشد (مثلاً جوراب قبل از کفش). لباس‌های دیگری هستند که ترتیب پوشیدن آن‌ها مهم نیست (مثلاً جوراب‌ها و شلوار). یک یال جهت‌دار (u, v) در گراف جهت‌دار بدون دور شکل ۲۲-۷(الف) نشان‌دهنده‌ی این

است که لباس u باید قبل از لباس v پوشیده شود. بنابراین یک مرتب‌سازی توپولوژیکی برای این گراف یک ترتیب برای لباس پوشیدن به ما می‌دهد. شکل ۲۲-۷(ب) گراف مرتب‌شده‌ی توپولوژیکی را به صورت ترتیبی از رأس‌ها بر روی یک خط افقی نشان می‌دهد به طوری که تمام یال‌های جهت‌دار از چپ به راست حرکت می‌کنند.

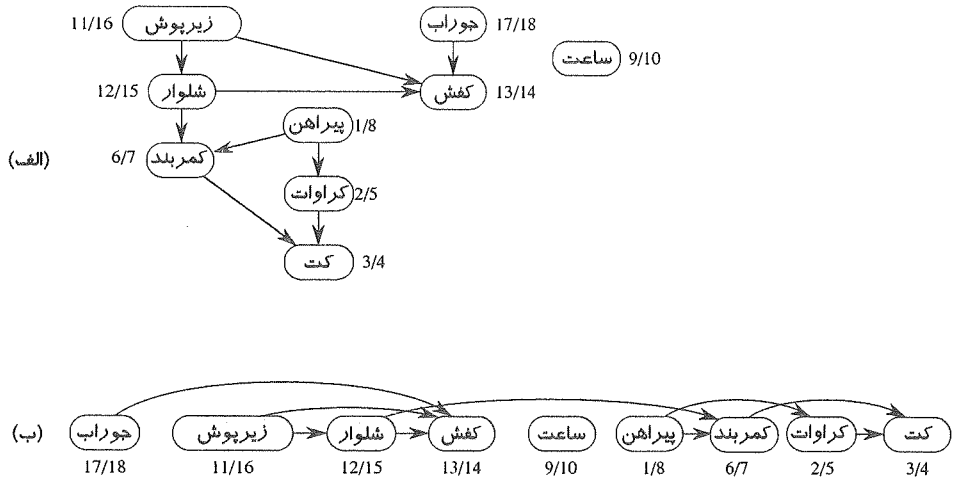
الگوریتم ساده‌ی زیر یک گراف جهت‌دار بدون دور را به صورت توپولوژیکی مرتب می‌کند.

```
TOPOLOGICAL-SORT(G)
1  call DFS(G) to compute finishing times v.f for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

شکل ۲۲-۷ (ب) نشان می‌دهد که رأس‌های مرتب شده‌ی توپولوژیکی چگونه به ترتیب برعکس زمان پایان ظاهر می‌شوند.

مرتب‌سازی توپولوژیکی را می‌توان در زمان $\theta(V+E)$ انجام داد، چرا که جستجوی عمق اول در زمان $\theta(V+E)$ اجرا می‌شود، و در زمان $O(1)$ می‌توان هر یک از $|V|$ رأس را در ابتدای یک لیست پیوندی درج کرد.

با استفاده از لم کلیدی زیر که خصوصیات گراف‌های جهت‌دار بدون دور را مشخص می‌کند، درستی این الگوریتم را اثبات خواهیم کرد.



شکل ۲۲-۷ (الف) پروفسور Bumstead هنگام لباس پوشیدن، لباس‌های خود را به صورت توپولوژیکی مرتب می‌کند. هر یال جهت‌دار (u, v) بدین معنی است که آرگومان u باید قبل از آرگومان v قرار گیرد. زمان کشف و زمان پایان هر رأس در کنار آن نوشته شده است. (ب) همان گراف که به صورت توپولوژیکی مرتب شده است. رأس‌های آن از چپ به راست به ترتیب زمان پایان مرتب شده‌اند. توجه کنید که تمام یال‌های جهت‌دار از چپ به راست هستند.

لم ۲۲-۱۰ یک گراف جهت‌دار G بدون دور است اگر و فقط اگر جستجوی عمق اول G هیچ یال عقبی تولید نکند.

اثبات $\Rightarrow$: فرض کنید یک یال عقبی (u,v) وجود داشته باشد. آن گاه در جنگل عمق اول رأس u جد رأس v است. بنابراین در G یک مسیر از v به u وجود دارد، و یال عقبی (u,v) یک دور را کامل می‌کند.

$\Leftarrow$: فرض کنید G شامل یک دور c باشد. نشان می‌دهیم که جستجوی عمق اول G یک یال عقبی تولید می‌کند. فرض کنید v اولین رأسی باشد که در c کشف می‌شود، و (u,v) یال بعد از آن در c باشد. در زمان $v.d$ رأس‌های c یک مسیر از رأس‌های سفید از v به u تشکیل می‌دهند. طبق قضیه‌ی مسیر سفید، در جنگل عمق اول رأس u یک نواده‌ی رأس v خواهد بود. بنابراین (u,v) یک یال عقبی است. ■

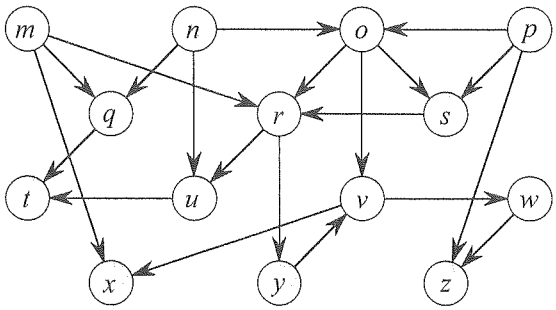
قضیه‌ی ۲۲-۱۲ TOPOLOGICAL-SORT(G) یک مرتب‌سازی توپولوژیکی برای گراف جهت‌دار بدون دور G تولید می‌کند.

اثبات فرض کنید DFS بر روی یک گراف داده شده‌ی $G=(V,E)$ اجرا می‌شود تا زمان پایان را برای رأس‌های آن تعیین کند. کافی است نشان دهیم که برای هر جفت از رأس‌های مجزای $u,v \in V$، اگر در G یک یال از u به v وجود داشته باشد، آن گاه $v.f < u.f$. یک یال (u,v) را در نظر بگیرید که توسط DFS(G) کشف شده است. وقتی این یال کشف می‌شود v نمی‌تواند خاکستری باشد، چرا که در این صورت v یک جد u خواهد بود و (u,v) یک یال عقبی، که لم ۲۲-۱۱ را نقض می‌کند. بنابراین v باید یا سفید باشد و یا سیاه. اگر v سفید باشد، یک نواده‌ی u خواهد شد، و بنابراین $v.f < u.f$. اگر v سیاه باشد، آن گاه کار اکتشاف آن قبلاً پایان یافته است، و بنابراین $v.f$ قبلاً مقداردهی شده است. از آن جایی که ما هنوز در حال اکتشاف از u هستیم، هنوز برچسب زمانی $u.f$ را تعیین نکرده‌ایم، و بنابراین وقتی این کار را بکنیم خواهیم داشت $v.f < u.f$. بنابراین برای هر یال (u,v) در یک گراف جهت‌دار بدون دور، داریم $v.f < u.f$، که اثبات را کامل می‌کند. ■

تمرین‌ها

۲۲-۴-۱ ترتیبی را که TOPOLOGICAL-SORT برای رأس‌های گراف شکل ۲۲-۸ تولید می‌کند نشان دهید، با فرض تمرین ۲۲-۳-۲.

۲۲-۴-۲ یک الگوریتم خطی ارائه کنید که یک گراف جهت‌دار بدون دور $G=(V,E)$ و دو رأس s و t را به عنوان ورودی دریافت کرده و تعداد مسیرها را از s به t به عنوان خروجی بازمی‌گرداند. به عنوان مثال در گراف جهت‌دار بدون دور شکل ۲۲-۸، دقیقاً چهار مسیر از رأس p به رأس v وجود دارد: pov، $poryv$، $posryv$، و $psryv$. (الگوریتم شما فقط باید تعداد مسیرها را بشمارد، و نیازی به لیست کردن آن‌ها نیست.)



شکل ۲۲-۸ یک گراف جهت‌دار بدون دور برای مرتب‌سازی توپولوژیکی.

۳-۴-۲۲ یک الگوریتم ارائه کنید که تعیین می‌کند آیا یک گراف بدون جهت $G = (V, E)$ دور دارد یا خیر. الگوریتم شما باید مستقل از $|E|$ در زمان $O(V)$ اجرا شود.

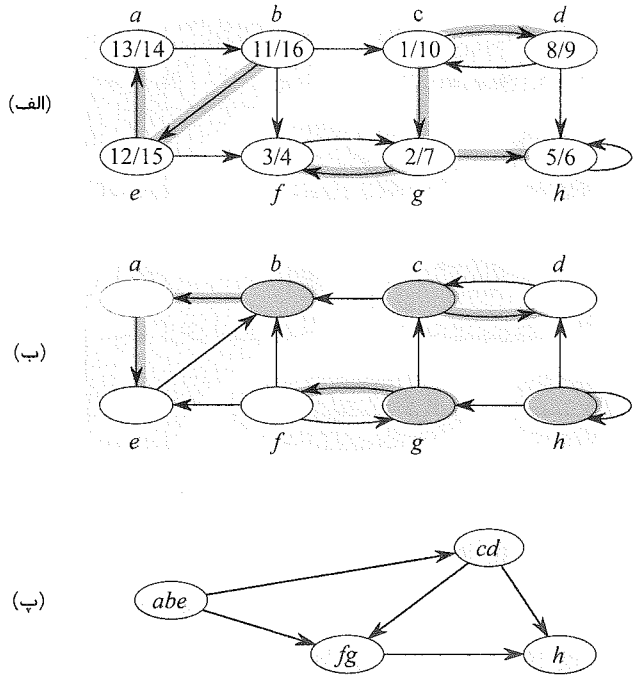
۴-۴-۲۲ اثبات یا رد کنید: اگر یک گراف جهت‌دار G حاوی دور باشد، آن گاه TOPOLOGICAL-SORT(G) یک ترتیب بر روی رأس‌ها می‌دهد که تعداد یال‌های «بد» را که با ترتیب تولید شده نامتناسب هستند، کمینه می‌کند.

۵-۴-۲۲ یک راه دیگر برای انجام مرتب‌سازی توپولوژیکی بر روی یک گراف جهت‌دار $G = (V, E)$ این است که مکرراً یک رأس با درجه‌ی ورودی ۰ بیابیم، آن را به خروجی بدهیم، و سپس آن را به همراه تمام یال‌های خروجی آن از گراف حذف کنیم. توضیح دهید که چگونه می‌توان این ایده را پیاده‌سازی کرد که در زمان $O(V + E)$ اجرا شود. اگر G دور داشته باشد، برای این الگوریتم چه اتفاقی می‌افتد؟

۲۲-۵ مؤلفه‌های قویاً همبند

اکنون یک کاربرد کلاسیک جستجوی عمق اول را بررسی می‌کنیم: تجزیه‌ی یک گراف جهت‌دار به مؤلفه‌های قویاً همبند آن. در این بخش نشان خواهیم داد که چگونه می‌توان با استفاده از دو جستجوی عمق اول، گراف را به مؤلفه‌های قویاً همبند تجزیه کرد. بسیاری از الگوریتم‌ها با گراف‌هایی کار می‌کنند که این تجزیه بر روی آن‌ها انجام شده است. بعد از تجزیه، الگوریتم به صورت جداگانه بر روی هر یک از مؤلفه‌های همبندی اجرا می‌شود. سپس جواب‌ها بر حسب ساختار اتصال‌ها میان مؤلفه‌ها ترکیب می‌شوند.

از پیوست ب به خاطر آورید که یک مؤلفه‌ی قویاً همبند یک گراف $G = (V, E)$، یک مجموعه‌ی حداکثر $C \subseteq V$ از رأس‌ها است به طوری که برای هر جفت u و v در C داشته باشیم $u \rightsquigarrow v$ و $v \rightsquigarrow u$؛ یعنی رأس‌های u و v از یکدیگر قابل دسترس هستند. شکل ۲۲-۹ یک مثال را نشان می‌دهد.



شکل ۲۲-۹ (الف) یک گراف جهت‌دار G. مؤلفه‌های قویاً همبند G به صورت ناحیه‌های سایه‌دار نشان داده شده‌اند. هر رأس با زمان کشف و زمان پایان آن برچسب گذاری شده است. یال‌های درخت سایه زده شده‌اند. (ب) گراف G^T، ترانهاده‌ی گراف G. جنگل عمق اول محاسبه شده در خط ۳ رویه‌ی STRONGLY-CONNECTED-COMPONENTS نشان داده شده است، که در آن یال‌های درخت سایه زده شده‌اند. رأس‌های b، c، g، و h، که با سایه‌ی پررنگ مشخص شده‌اند، ریشه‌های درخت‌های عمق اول ساخته شده توسط جستجوی عمق اول G^T هستند. (پ) گراف مؤلفه‌های بدون دور G^{SCC} که از یکی کردن تمام یال‌ها در هر مؤلفه‌ی همبندی در G به دست می‌آید، به طوری که در هر مؤلفه فقط یک رأس باقی می‌ماند.

الگوریتم ما برای یافتن مؤلفه‌های همبندی یک گراف $G = (V, E)$ از ترانهاده‌ی G استفاده می‌کند که در تمرین ۲۲-۱-۳ تعریف شده است. این ترانهاده عبارت است از گراف $G^T = (V, E^T)$ به طوری که $E^T = \{(u, v) : (v, u) \in E\}$. یعنی E^T شامل یال‌های G است که جهت آن‌ها عوض شده است. با داشتن یک نمایش لیست مجاورت گراف G، زمان ساختن G^T برابر است با $O(V + E)$. جالب است بدانید که مؤلفه‌های قویاً همبند G و G^T دقیقاً یکی هستند: u و v در G از یکدیگر قابل دسترس هستند اگر و فقط اگر در G^T هم از یکدیگر قابل دسترس باشند. شکل ۲۲-۹(ب) ترانهاده‌ی گراف شکل ۲۲-۹(الف) را نشان می‌دهد، که در آن مؤلفه‌های قویاً همبند با سایه مشخص شده‌اند.

الگوریتم خطی زیر (یعنی با زمان $\theta(V+E)$) با استفاده از دو جستجوی عمق اول، یکی بر روی G و یکی بر روی G^T، مؤلفه‌های قویاً همبند را در یک گراف جهت‌دار $G=(V,E)$ محاسبه می‌کند.

```
STRONGLY-CONNECTED-COMPONENTS (G)
1  call DFS (G) to compute finishing times u.f for each vertex u
2  compute G^T
3  call DFS (G^T), but in the main loop of DFS, consider the vertices
       in order of decreasing u.f (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in
       line 3 as a separate strongly connected component
```

ایده‌ی پشت این الگوریتم از خصوصیت کلیدی *گراف مؤلفه‌ای* (component graph) $G^{SCC}=(V^{SCC},E^{SCC})$ سرچشمه می‌گیرد، که آن را به صورت زیر تعریف می‌کنیم. فرض کنید که G از مؤلفه‌های قویاً همبند $C_1, C_2, \ldots, C_k$ تشکیل شده است. مجموعه‌ی رأس‌های V^{SCC} عبارت است از $\{v_1, v_2, \ldots, v_k\}$، و برای هر مؤلفه‌ی همبندی C_i از G حاوی یک رأس v_i است. یک یال $(v_i, v_j) \in E^{SCC}$ وجود دارد اگر G حاوی یک یال جهت‌دار (x, y) از یک $x \in C_i$ به یک $y \in C_j$ باشد. به طور مشابه می‌توان گفت با یکی کردن تمام یال‌هایی که رأس‌های مجاور آن‌ها در یک مؤلفه‌ی قویاً همبند در G هستند، گراف G^{SCC} حاصل می‌شود. شکل ۲۲-۹(پ) گراف مؤلفه‌ای گراف شکل ۲۲-۹ (الف) را نشان می‌دهد.

خصوصیت اصلی این است که گراف مؤلفه‌ای یک گراف جهت‌دار بدون دور است، که لم زیر آن را نشان می‌دهد.

لم ۲۲-۱۳ فرض کنید C و C' دو مؤلفه‌ی قویاً همبند مجزا در گراف جهت‌دار $G=(V,E)$ باشند، و $u, v \in C$ و $u', v' \in C'$، و همچنین فرض کنید که یک مسیر $u \rightsquigarrow u'$ در G وجود دارد. در این صورت هیچ مسیری به شکل $v \rightsquigarrow v'$ در G وجود ندارد.

اثبات اگر یک مسیر $v \rightsquigarrow v'$ در G وجود داشته باشد، آن گاه مسیرهای $u \rightsquigarrow u' \rightsquigarrow v'$ و $u \rightsquigarrow v \rightsquigarrow v'$ هم در G وجود دارند. بنابراین u و v' از یکدیگر قابل دسترس هستند، که با فرض این که C و C' مؤلفه‌های قویاً همبند مجزا هستند تناقض دارد. ■

خواهیم دید که با در نظر گرفتن رأس‌ها در دومین جستجوی عمق اول به ترتیب نزولی زمان پایانی که در اولین جستجوی عمق اول ساخته شده است، در واقع داریم رأس‌های گراف مؤلفه‌ای (که هر کدام از آن‌ها متناظر است با یکی از مؤلفه‌های قویاً همبند G) را به ترتیب توپولوژیکی ملاقات می‌کنیم.

چون STRONGLY-CONNECTED-COMPONENTS دو جستجوی عمق اول انجام می‌دهد، وقتی در مورد $u.d$ و $u.f$ صحبت می‌کنیم احتمال ابهام وجود دارد. در این بخش، این مقادیر همیشه مربوط هستند به زمان‌های کشف و پایان محاسبه شده در اولین فراخوانی DFS در خط ۱.

اکنون نماد زمان‌های کشف و پایان را برای مجموعه‌ای از رأس‌ها گسترش می‌دهیم. اگر $U \subseteq V$، آن گاه تعریف می‌کنیم $d(U) = \min_{u \in U}\{u.d\}$ و $f(U) = \max_{u \in U}\{u.f\}$. یعنی $d(U)$ و $f(U)$ به ترتیب اولین زمان کشف و آخرین زمان پایان در تمام رأس‌های U هستند.

لم زیر و نتیجه‌ی آن یک خصوصیت کلیدی در مورد مؤلفه‌های قویاً همبند و زمان‌های پایان در جستجوی عمق اول به دست می‌دهند.

لم ۲۲-۱۴ فرض کنید C و C' مؤلفه‌های قویاً همبند مجزا در گراف جهت‌دار $G = (V, E)$ باشند، و یک یال $(u, v) \in E$ وجود داشته باشد، که $u \in C$ و $v \in C'$. آن گاه $f(C) > f(C')$.

اثبات بسته به این که از میان مؤلفه‌های قویاً همبند C و C' کدام یک اولین رأس کشف شده را دارد، دو حالت به وجود خواهد آمد.

اگر $d(C') < d(C)$، آن گاه فرض کنید x اولین رأس کشف شده در C باشد. در زمان $x.d$ تمام رأس‌های C و C' سفید هستند. در G یک مسیر از x به تمام رأس‌های C شامل فقط رأس‌های سفید وجود دارد . چون $(u, v) \in E$، برای هر رأس $w \in C'$ در زمان $x.d$ یک مسیر از x به w در G شامل فقط رأس‌های سفید وجود دارد: $x \rightsquigarrow u \rightarrow v \rightsquigarrow w$. طبق قضیه‌ی مسیر سفید، تمام رأس‌ها در C و C' در درخت عمق اول نوادگان x خواهند شد. طبق نتیجه‌ی ۲۲-۸ خواهیم داشت $x.f = f(C) > f(C')$.

در عوض اگر داشته باشیم $d(C) > d(C')$، فرض کنید y اولین رأس کشف شده در C' باشد. در زمان $y.d$ تمام رأس‌ها در C' سفید هستند و در G یک مسیر شامل فقط رأس‌های سفید از y به هر یک از رأس‌ها در C' وجود دارد. طبق قضیه‌ی مسیر سفید، تمام رأس‌ها در C' در درخت عمق اول نوادگان y خواهند شد، و طبق نتیجه‌ی ۲۲-۸ داریم $y.f = f(C')$. در زمان $y.d$، تمام رأس‌ها در C سفید هستند. از آن جایی که یک یال (u, v) از C به C' وجود دارد، لم ۲۲-۱۳ ایجاب می‌کند که مسیری از C' به C وجود نخواهد داشت. بنابراین هیچ رأسی در C از y قابل دسترس نیست. از این رو در زمان $y.f$ تمام رأس‌ها در C هنوز سفید هستند. بنابراین برای هر رأس $w \in C$ داریم $w.f > y.f$، که نتیجه می‌دهد $f(C) > f(C')$. ■

نتیجه‌ی زیر می‌گوید هر یال در G^T که بین مؤلفه‌های قویاً همبند مختلف گذر می‌کنند، از یک مؤلفه با زمان پایان زودتر(در اولین درخت عمق اول) به یک مؤلفه با زمان پایان دیرتر می‌رود.

نتیجه‌ی ۲۲-۱۵ فرض کنید C و C' مؤلفه‌های قویاً همبند مجزا در گراف جهت‌دار $G = (V, E)$ باشند. فرض کنید که یک یال $(u, v) \in E^T$ وجود داشته باشد، که $u \in C$ و $v \in C'$. آن گاه $f(C) < f(C')$.

اثبات از آن جایی که $(u, v) \in E^T$، داریم $(v, u) \in E^T$. چون مؤلفه‌های قویاً همبند G و G^T یکی هستند، لم ۲۲-۱۴ ایجاب می‌کند که $f(C) < f(C')$. ■

نتیجه‌ی ۲۲-۱۵ یک درک کلیدی به ما می‌دهد که چرا رویه‌ی STRONGLY-CONNECTED-COMPONENT به درستی کار می‌کند. اجازه دهید ببینیم که وقتی دومین جستجوی عمق اول را که بر روی G^T است، انجام می‌دهیم چه اتفاقی می‌افتد. با مؤلفه‌ی قویاً همبند C شروع می‌کنیم که زمان پایان آن $f(C)$ بیشینه است. جستجو از یک رأس $x \in C$ آغاز می‌شود، و تمام رأس‌های C را ملاقات می‌کند. طبق نتیجه‌ی ۲۲-۱۵ هیچ یالی در G^T از C به هیچ یک از مؤلفه‌های همبندی دیگر وجود ندارد، و بنابراین جستجو از x به رأس‌های هیچ مؤلفه‌ی دیگری نخواهد رسید. بنابراین درخت با ریشه‌ی x دقیقاً حاوی رأس‌های C است. با پایان ملاقات تمام رأس‌ها در C، جستجو در خط ۳ یک رأس از یک مؤلفه‌ی قویاً همبند دیگر C' را به عنوان ریشه انتخاب خواهد کرد، که زمان پایان $f(C')$ این مؤلفه میان تمام مؤلفه‌های دیگر غیر از C بیشینه است. دوباره جستجو تمام رأس‌ها را در C' ملاقات خواهد کرد، ولی طبق نتیجه‌ی ۲۲-۱۵ تنها یال‌هایی در G^T که از C' به یکی دیگر از مؤلفه‌ها می‌روند، باید به C باشند، که آن را قبلاً ملاقات کرده‌ایم. به طور کلی وقتی جستجوی عمق اول G^T در خط ۳ هر یک از مؤلفه‌های قویاً همبند را ملاقات می‌کند، هر یال خروجی از آن مؤلفه باید به مؤلفه‌هایی باشد که قبلاً ملاقات شده‌اند. بنابراین هر درخت عمق اول دقیقاً متناظر با یک مؤلفه‌ی قویاً همبند خواهد بود. قضیه‌ی زیر این بحث را به صورت رسمی بیان می‌کند.

قضیه‌ی ۲۲-۱۶ STROGLY-CONNECTED-COMPONETNS (G) به درستی مؤلفه‌های قویاً همبند گراف جهت‌دار G را محاسبه می‌کند.

اثبات با استقرا بر روی تعداد درخت‌های عمق اول یافت شده در جستجوی عمق اول G^T در خط ۳، بحث می‌کنیم که رأس‌های هر درخت یک مؤلفه‌ی قویاً همبند را تشکیل می‌دهند. فرض استقرا این است که اولین k درخت ساخته شده در خط ۳ مؤلفه‌های قویاً همبند هستند. پایه‌ی این استقرا، وقتی $k = 0$ بدیهی است.

در گام استقرا فرض می‌کنیم که هر یک از k درخت عمق اول ساخته شده در خط ۳ یک مؤلفه‌ی قویاً همبند است، و $(k+1)$امین درخت ساخته شده را در نظر می‌گیریم. فرض کنید ریشه‌ی این درخت رأس u باشد، و u در مؤلفه‌ی قویاً همبند C. به دلیل روش انتخاب ریشه‌ها در جستجوی عمق اول در خط ۳، برای هر مؤلفه‌ی قویاً همبند C' غیر از C که هنوز باید ملاقات شود،

$u.f = f(C) > f(C')$. طبق فرض استقرا زمانی که جستجو u را ملاقات می‌کند، تمام رأس‌های دیگر C سفید هستند. بنابراین طبق قضیه‌ی مسیر سفید، تمام رأس‌های دیگر C در درخت عمق اول نوادگان u هستند. به علاوه طبق فرض استقرا و نتیجه‌ی ۲۲-۱۵، هر یالی در G^T که از C خارج می‌شود باید به یک مؤلفه‌ی قویاً همبند وارد شود که قبلاً ملاقات شده است. بنابراین هیچ رأسی در هیچ یک از مؤلفه‌های قویاً همبند غیر از رأس‌های C در جستجوی عمق اول G^T نواده‌ی u نخواهد بود. از این رو رأس‌های درخت عمق اول در G^T که ریشه‌ی آن u است دقیقاً یک مؤلفه‌ی قویاً همبند را تشکیل می‌دهند، که گام استقرا و در نتیجه قضیه را اثبات می‌کند. ■

در این جا به شکلی دیگر به عملکرد دومین جستجوی عمق اول نگاه می‌کنیم. گراف مؤلفه‌ای $(G^T)^{SCC}$ مربوط به G^T را در نظر بگیرید. اگر هر مؤلفه‌ی قویاً همبند ملاقات شده در دومین جستجوی عمق اول را به یک رأس در $(G^T)^{SCC}$ نگاشت کنیم، رأس‌های $(G^T)^{SCC}$ به ترتیب برعکس مرتب‌سازی توپولوژیکی ملاقات می‌شوند. اگر رأس‌های $(G^T)^{SCC}$ را برعکس کنیم، گراف $((G^T)^{SCC})^T$ را خواهیم داشت. چون $((G^T)^{SCC})^T = G^{SCC}$ (تمرین ۲۲-۵-۴ را ببینید)، دومین جستجوی عمق اول رأس‌های G^{SCC} را به ترتیب توپولوژیکی ملاقات می‌کند.

تمرین‌ها

۲۲-۵-۱ اگر یک رأس جدید به یک گراف اضافه شود، تعداد مؤلفه‌های قویاً همبند آن چگونه ممکن است تغییر کند؟

۲۲-۵-۲ نشان دهید که رویه‌ی STRONGLY-CONNECTED-COMPONENTS بر روی گراف شکل ۲۲-۶ چگونه عمل می‌کند. به طور خاص، زمان‌های پایان تولید شده در خط ۱ و جنگل عمق اول ساخته شده در خط ۳ را نشان دهید. فرض کنید که حلقه‌ی خطوط ۵-۷ رویه‌ی DFS رأس‌ها را به ترتیب الفبایی در نظر می‌گیرد، و همچنین لیست‌های مجاورت به ترتیب حروف الفبا هستند.

۲۲-۵-۳ پروفسور Deaver ادعا می‌کند که الگوریتم مؤلفه‌های قویاً همبند را می‌توان با استفاده از گراف اصلی (به جای گراف ترانهاده) در دومین جستجوی عمق اول و پویش رأس‌ها به ترتیب *صعودی* زمان‌های پایان ساده‌تر کرد. آیا این الگوریتم ساده‌تر همیشه نتیجه‌ی درستی تولید می‌کند؟

۲۲-۵-۴ اثبات کنید که برای هر گراف G داریم $((G^T)^{SCC})^T = G^{SCC}$. یعنی ترانهاده‌ی گراف مؤلفه‌ای G^T معادل است با گراف مؤلفه‌ای G.

۲۲-۵-۵ یک الگوریتم با زمان $O(V+E)$ برای محاسبه‌ی گراف مؤلفه‌ای گراف جهت‌دار

$G = (V, E)$ ارائه کنید. اطمینان حاصل کنید که میان هر دو رأس گراف مؤلفه‌ای که الگوریتم شما تولید می‌کند، حداکثر یک یال وجود دارد.

۶-۵-۲۲ با داشتن یک گراف جهت‌دار $G = (V, E)$، توضیح دهید که چگونه می‌توان یک گراف دیگر $G' = (V, E')$ ساخت به طوری که (الف) مؤلفه‌های همبندی G' دقیقاً مانند مؤلفه‌های همبندی G باشد، (ب) گراف مؤلفه‌ای G' معادل گراف مؤلفه‌ای G باشد، و (پ) E' تا حد ممکن کوچک باشد. یک الگوریتم سریع برای محاسبه‌ی G' بدهید.

۷-۵-۲۲ یک گراف جهت‌دار $G = (V, E)$ *شبه‌همبند* (semiconnected) است اگر برای تمام جفت رأس‌های $u, v \in V$ داشته باشیم $u \rightsquigarrow v$ یا $v \rightsquigarrow u$. یک الگوریتم بهینه ارائه کنید که تعیین می‌کند آیا G شبه‌همبند است یا خیر. اثبات کنید که الگوریتم شما به درستی کار می‌کند، و زمان اجرای آن را تحلیل کنید.

مسائل

۲۲-۱ *دسته‌بندی یال‌ها توسط جستجوی عمق اول*

یک جنگل عمق اول یال‌های گراف را به دسته‌های درخت، عقبی، جلویی، و ضربدری تقسیم می‌کند. از یک درخت عمق اول هم می‌توان برای دسته‌بندی یال‌های قابل دسترس از مبدأ جستجو به همان چهار دسته استفاده کرد.

I. ثابت کنید که در یک جستجوی عمق اول بر روی یک گراف بدون جهت، خصوصیات زیر برقرار است:

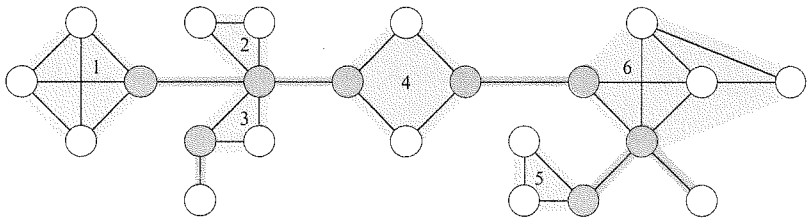
۱. هیچ یال عقبی و هیچ یال جلویی وجود ندارد.
۲. برای هر یال درخت (u, v) داریم $v.d = u.d + ۱$.
۳. برای هر یال ضربدری (u, v) داریم $v.d = u.d$ یا $v.d = u.d + ۱$.

II. اثبات کنید که در یک جستجوی عمق اول بر روی یک گراف جهت‌دار، خصوصیات زیر برقرار است:

۱. هیچ یال جلویی وجود ندارد.
۲. برای هر یال درخت (u, v) داریم $v.d = u.d + ۱$.
۳. برای هر یال ضربدری (u, v) داریم $v.d \leq u.d + ۱$.
۴. برای هر یال عقبی (u, v) داریم $۰ \leq v.d \leq u.d$.

۲۲-۲ *نقاط مَفصَل، پل‌ها، و مؤلفه‌های دوهمبند*

فرض کنید $G = (V, E)$ یک گراف همبند بدون جهت باشد. یک *نقطه‌ی مَفصَل* (articulation



شکل ۲۲-۱۰ نقاط مَفصَل، پل‌ها، و مؤلفه‌های دوهمبند یک گراف بدون جهت همبند برای استفاده در مسئله‌ی ۲۲-۲. نقاط مَفصَل به صورت رأس‌های با سایه‌ی پررنگ، پل‌ها به صورت یال‌های با سایه‌ی پررنگ، و مؤلفه‌های دوهمبند به صورت ناحیه‌های سایه‌دار مشخص شده‌اند، و در هر ناحیه‌ی دوهمبند عدد bcc نشان داده شده است.

(point) در G رأسی است که حذف آن G را ناهمبند می‌کند. یک *پل* (bridge) در G یک یال است که حذف آن G را ناهمبند می‌کند. یک *مؤلفه‌ی دوهمبند* (biconnected component) در G یک مجموعه‌ی بیشینه از یال‌ها است به طوری که هر دو رأس در آن بر روی یک دور ساده‌ی مشترک قرار دارند. شکل ۲۲-۱۰ این تعاریف را نشان می‌دهد. می‌توان نقاط مَفصَل، پل‌ها، و مؤلفه‌های دوهمبند را با استفاده از جستجوی عمق اول تعیین کرد. فرض کنید $G_\pi = (V, E_\pi)$ یک درخت عمق اول G باشد.

I. اثبات کنید که ریشه‌ی G_π یک نقطه‌ی مَفصَل G است اگر و فقط اگر حداقل دو فرزند در G_π داشته باشد.

II. فرض کنید v یک رأس غیر ریشه در G_π باشد. اثبات کنید که v یک نقطه‌ی مَفصَل در G است اگر و فقط اگر v یک فرزند s داشته باشد به طوری که هیچ یال عقبی از s یا هیچ یک از نوادگان s به یکی از اجداد v وجود نداشته باشد.

III. فرض کنید

$$v.low = \min\begin{cases} v.d, \\ w.d : \end{cases}$$

به طوری که (u, v) یک یال عقبی است برای u که یکی از نوادگان v است.

نشان دهید که چگونه می‌توان $v.low$ را برای تمام رأس‌های $v \in V$ در زمان $O(E)$ محاسبه کرد.

IV. نشان دهید که چطور می‌توان تمام نقاط مَفصَل را در زمان $O(E)$ محاسبه کرد.

V. اثبات کنید که یک یال E پل است اگر و فقط اگر بر روی هیچ دور ساده‌ای از G قرار نداشته باشد.

VI. نشان دهید که چطور می‌توان تمام پل‌های G را در زمان $O(E)$ محاسبه کرد.

VII. اثبات کنید که مؤلفه‌های دوهمبند G یال‌های غیر پل را تقسیم بندی می‌کنند.

VIII. یک الگوریتم با زمان $O(E)$ ارائه کنید که هر یال e را در G با یک عدد صحیح مثبت

$e.bcc$ برچسب گذاری کند، به طوری که $e.bcc = e'.bcc$ اگر و فقط اگر e و e' در یک مؤلفه‌ی دوهمبند باشند.

مسایل

۲۲-۳ تور اویلری

یک *تور اویلری* (Euler tour) یک گراف جهت‌دار قویاً همبند $G = (V, E)$ یک دور است که از هر یال G دقیقاً یک بار گذر می‌کند، ولی ممکن است یک رأس را بیش از یک بار ملاقات کند.

I نشان دهید که G یک تور اویلری دارد اگر و فقط اگر برای تمام رأس‌های $v \in V$ داشته باشیم $in\text{-}degree(v) = out\text{-}degree(v)$.

II یک الگوریتم با زمان اجرای $O(E)$ ارائه کنید که تور اویلری G را در صورت وجود می‌یابد. (*راهنمایی:* دورهایی را که یال‌های مجزا دارند با هم ادغام کنید.)

۲۲-۴ قابلیت دسترسی

فرض کنید $G = (V, E)$ یک گراف جهت‌دار باشد به طوری که هر رأس $u \in V$ با یک عدد صحیح یکتای $L(u)$ از مجموعه‌ی $\{1, 2, ..., |V|\}$ برچسب‌گذاری شده است. برای هر رأس $u \in V$ فرض کنید $R(u) = \{v \in V : u \rightsquigarrow v\}$ مجموعه‌ی رأس‌هایی باشند که از u قابل دسترس هستند. $\min(u)$ را به صورت رأسی در $R(u)$ تعریف کنید که برچسب آن کمینه است، یعنی $\min(u)$ رأسی مانند v است به طوری که $L(v) = \min\{L(w) : w \in R(u)\}$. یک الگوریتم با زمان $O(V + E)$ ارائه کنید که $\min(u)$ را برای تمام رأس‌های $u \in V$ محاسبه می‌کند.

فصل

۲۳

درختان پوشای کمینه

۲۳-۰ مقدمه

در طراحی مدارهای الکترونیکی، معمولاً لازم است که پتانسیل سوزن‌های عناصر مختلفی را با متصل کردن آن‌ها با سیم یکی کنیم. برای اتصال مجموعه‌ای از n سوزن، می‌توانیم از $n-1$ تکه سیم استفاده کنیم که هر کدام از آن‌ها دو سوزن را به یکدیگر متصل می‌کنند. از تمام راه‌های ممکن برای این کار، معمولاً روشی که از کم‌ترین میزان سیم استفاده می‌کند روش مورد نظر است.

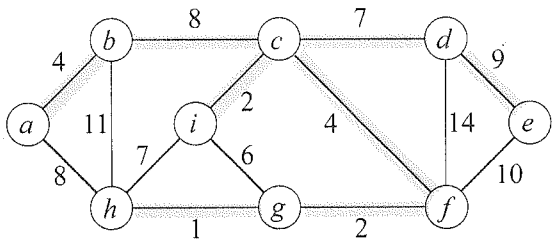
می‌توان این مسئله‌ی سیم‌کشی را با یک گراف بدون جهت همبند $G = (V, E)$ مدل کرد، که در آن V مجموعه‌ی سوزن‌ها و E مجموعه‌ی اتصال‌های ممکن میان جفت سوزن‌ها است، و برای هر یال $(u, v) \in E$ یک وزن $w(u, v)$ داریم که هزینه (میزان سیم مورد نیاز) اتصال u و v را مشخص می‌کند. می‌خواهیم یک زیرمجموعه‌ی بدون دور $T \subseteq E$ بیابیم که تمام رأس‌ها را به هم متصل کند و وزن کل آن

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

کمینه شود. از آن جایی که T بدون دور است و تمام رأس‌ها را به هم متصل می‌کند، باید یک درخت را تشکیل دهد، که ما به آن یک *درخت پوشا* (spanning tree) می‌گوییم، چرا که تمام گراف G را «می‌پوشاند». مسئله‌ی یافتن T *مسئله‌ی درخت پوشای کمینه* نام دارد.[1] شکل ۲۳-۱ یک مثال از یک گراف همبند و درخت پوشای کمینه‌ی آن را نشان می‌دهد.

---

[1] اصطلاح «درخت پوشای کمینه» نسخه‌ی کوتاه شده‌ی عبارت «درخت پوشای با وزن کمینه» است. به عنوان مثال ما نمی‌خواهیم تعداد یال‌ها را در T کمینه کنیم، چرا که طبق قضیه‌ی ب-۲ تمام درختان پوشا دقیقاً $|V| - 1$ یال دارند.



شکل ۲۳-۱ یک درخت پوشای کمینه برای یک گراف همبند. وزن یال‌ها نشان داده شده است، و یال‌های درخت پوشای کمینه با سایه مشخص شده‌اند. کل وزن درخت ۳۷ است. درخت پوشای کمینه یکتا نیست: حذف یال (b, c) و جایگزینی آن با یال (a, h) یک درخت پوشای دیگر با وزن ۳۷ تولید می‌کند.

در این فصل دو الگوریتم برای حل مسئله‌ی درخت پوشای کمینه را بررسی می‌کنیم: الگوریتم کروسکال و الگوریتم پرایم. می‌توان هر یک از آن‌ها را به سادگی و با استفاده از هرم‌های دودویی معمولی در زمان $O(E \lg V)$ پیاده‌سازی کرد. با استفاده از هرم‌های فیبوناچی، سرعت الگوریتم پرایم را می‌توان تا $O(E + V \lg V)$ افزایش داد، که اگر $|V|$ بسیار کوچک‌تر از $|E|$ باشد، بهبود خوبی نسبت به پیاده‌سازی با هرم‌های دودویی دارد.

این دو الگوریتم، الگوریتم‌های حریصانه هستند، همان طور که در فصل ۱۶ توصیف شد. در هر مرحله از یک الگوریتم یکی از گزینه‌های ممکن باید انتخاب شود. استراتژی حریصانه از انتخابی طرفداری می‌کند که در لحظه بهترین به نظر برسد. در حالت کلی چنین استراتژی تضمین نمی‌کند که یک جواب بهینه‌ی کلی برای مسئله بیابیم. با این حال برای مسئله‌ی درخت پوشای کمینه، می‌توانیم اثبات کنیم که استراتژی‌های حریصانه‌ی خاصی به یک درخت پوشا با وزن کمینه ختم می‌شوند. با این که فصل حاضر را می‌توانید مستقل از فصل ۱۶ مطالعه کنید، متدهای حریصانه‌ای که در این جا مطرح می‌شوند، کاربردهای کلاسیکی از افکار تئوری هستند که در آن جا معرفی شدند.

بخش ۲۳-۱ یک متد «کلی» برای یافتن درخت پوشای کمینه معرفی می‌کند، که یک درخت پوشا را هر بار با اضافه کردن یک یال رشد می‌دهد. بخش ۲۳-۲ دو روش برای پیاده‌سازی این متد کلی ارائه می‌کند. الگوریتم اول منسوب به کروسکال (Kruskal)، مشابه الگوریتم مؤلفه‌های همبندی از بخش ۲۱-۱ است. الگوریتم دوم منسوب به پرایم (Prime) مانند الگوریتم کوتاه‌ترین مسیرهای Dijkstra است (بخش ۲۴-۳).

چون درخت نوعی گراف است، برای دقیق بودن باید درخت را نه فقط بر حسب یال‌های آن، که بر حسب یال‌ها و رأس‌های آن تعریف کنیم. با این که این فصل بر روی درخت‌ها و یال‌های آن‌ها تمرکز می‌کند، این کار را با این درک انجام می‌دهیم که رأس‌های یک درخت T آن‌هایی هستند که یک یال مجاور در T دارند.

۲۳-۱ رشد دادن یک درخت پوشای کمینه

فرض کنید یک گراف بدون جهت همبند $G = (V, E)$ داریم، به همراه یک تابع وزن $w : E \to R$، و می‌خواهیم یک درخت پوشای کمینه برای G بسازیم. دو الگوریتمی که در این فصل بررسی می‌کنیم از یک رویکرد حریصانه برای مسئله استفاده می‌کنند، ولی در روش به کار گرفتن این رویکرد با یکدیگر متفاوت هستند.

این استراتژی حریصانه از الگوریتم «کلی» زیر گرفته شده است، که درخت پوشای کمینه را هر بار با اضافه کردن یک یال رشد می‌دهد. الگوریتم یک مجموعه‌ی A از یال‌ها را نگه می‌دارد، که ثابت حلقه‌ی زیر را حفظ می‌کند:

- قبل از هر بار تکرار، A یک زیرمجموعه از یک درخت پوشای کمینه است.

در هر مرحله یک یال (u, v) می‌یابیم که می‌تواند بدون نقض این ثابت حلقه به A اضافه شود، به طوری که $A \cup \{(u, v)\}$ هم زیرمجموعه‌ای از یک درخت پوشای کمینه باشد. به چنین یالی یک *یال ایمن* (safe edge) برای A می‌گوییم، چرا که می‌توان آن را با اطمینان از حفظ ثابت حلقه به A اضافه کرد.

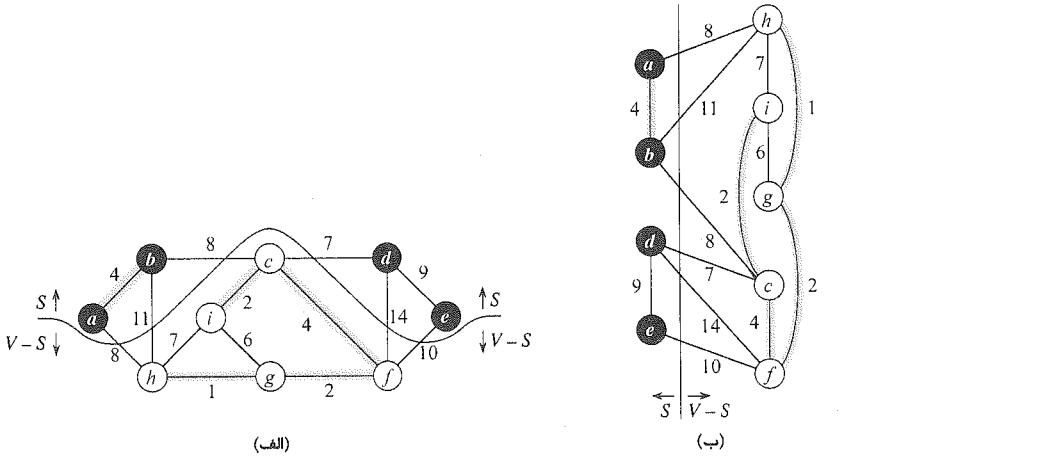
```
GENERIC-MST(G, w)
1  A = ∅
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A ∪ {(u, v)}
5  return A
```

از ثابت حلقه به صورت زیر استفاده می‌کنیم:

- آغاز: بعد از خط ۱ مجموعه‌ی A به صورت بدیهی ثابت حلقه را حفظ می‌کند.
- ادامه: حلقه در خطوط ۲-۴ ثابت حلقه را با اضافه کردن فقط یال‌های ایمن حفظ می‌کند.
- پایان: تمام یال‌هایی که به A اضافه می‌شوند در یک درخت پوشای کمینه هستند، و بنابراین مجموعه‌ی A که در خط ۵ بازگردانده می‌شود باید یک درخت پوشای کمینه باشد.

قسمت سخت، مسلماً یافتن یک یال ایمن در خط ۳ است. حتماً چنین یالی باید وجود داشته باشد، چرا که وقتی خط ۳ اجرا می‌شود ثابت حلقه ایجاب می‌کند که یک درخت پوشای T وجود دارد به طوری که $A \subseteq T$. در بدنه‌ی حلقه‌ی while ، A باید یک زیرمجموعه‌ی سره‌ی T باشد، و بنابراین باید یک یال $(u, v) \in T$ وجود داشته باشد به طوری که $(u, v) \notin A$ و (u, v) برای A ایمن است.

در ادامه‌ی این بخش یک قانون (قضیه‌ی ۲۳-۱) برای تشخیص یال‌های ایمن فراهم می‌کنیم. بخش بعد دو الگوریتم را توصیف می‌کند که از این قانون برای یافتن یال‌های ایمن به صورت بهینه استفاده می‌کنند.

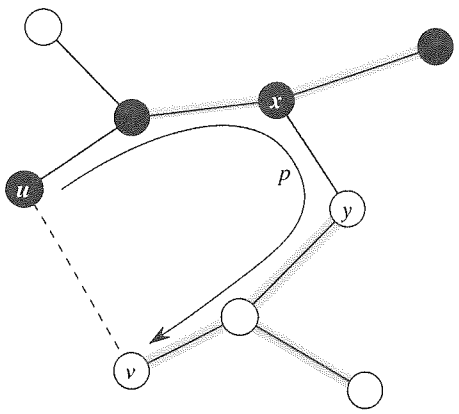


شکل ۲۳-۲ دو روش برای دیدن یک برش $(S, V-S)$ از گراف شکل ۲۳-۱. (الف) رأس‌های مجموعه‌ی S با رنگ تیره، و رأس‌های مجموعه‌ی $V-S$ با رنگ روشن مشخص شده‌اند. یال‌های عبور کننده از برش آن‌هایی هستند که رأس‌های روشن را به رأس‌های تیره متصل می‌کنند. یال (d, c)، یال سبک عبور کننده از برش است. یک زیرمجموعه‌ی A از یال‌ها سایه زده شده است؛ توجه کنید که برش $(S, V-S)$ به A احترام می‌گذارد، چرا که هیچ یالی در A از برش عبور نمی‌کند. (ب) همان گراف که در آن رأس‌های مجموعه‌ی S در سمت چپ و رأس‌های مجموعه‌ی $V-S$ در سمت راست کشیده شده‌اند. یک یال از برش عبور می‌کند اگر یک رأس در سمت چپ را به یک رأس در سمت راست متصل کند.

ابتدا به چند تعریف نیاز داریم. یک *برش* (cut) $(S, V-S)$ از یک گراف بدون جهت $G=(V, E)$ یک تقسیم بندی بر روی V است. شکل ۲۳-۲ این مفهوم را روشن می‌کند. می‌گوییم یک یال $(u, v) \in E$ از برش $(S, V-S)$ *عبور* می‌کند اگر یکی از نقاط پایانی آن در S و دیگری در $V-S$ باشد. می‌گوییم یک برش به مجموعه‌ی یال‌های A *احترام* می‌گذارد اگر هیچ یالی از A از برش عبور نکند. یک یال، *یال سبک* عبور کننده از یک برش است اگر وزن آن میان تمام یال‌های عبور کننده از برش کمینه باشد. توجه داشته باشید که ممکن است بیش از یک یال سبک عبور کننده از یک برش وجود داشته باشد. به طور کلی‌تر می‌گوییم یک یال، یال *سبک ارضا* کننده‌ی یک شرط خاص است اگر وزن آن میان یال‌هایی که آن شرط را ارضا می‌کنند کمینه باشد.

قانونی که برای تشخیص یال‌های ایمن به کار می‌بریم در قضیه‌ی زیر داده شده است.

قضیه‌ی ۲۳-۱ فرض کنید $G=(V, E)$ یک گراف بدون جهت همبند باشد، با یک تابع وزن w با مقدار حقیقی که بر روی E تعریف شده است. فرض کنید A یک زیرمجموعه از E باشد که یک درخت پوشای کمینه در E آن را در بر می‌گیرد، و فرض کنید $(S, V-S)$ یک برش دلخواه باشد که به A احترام می‌گذارد، و (u, v) یک یال سبک عبور کننده از برش $(S, V-S)$ باشد. آن گاه یال (u, v) برای $(S, V-S)$ ایمن است.



شکل ۲۳-۳ اثبات قضیه‌ی ۲۳-۱. رأس‌های S تیره، و رأس‌های $V-S$ روشن هستند. یال‌های درون درخت پوشای کمینه‌ی T نشان داده شده‌اند، ولی یال‌های گراف G نشان داده نشده‌اند. یال‌های درون A با سایه مشخص شده‌اند، و (u,v) یک یال سبک گذرا از برش $(S,V-S)$ است. یال (x,y) یک یال بر روی مسیر یکتای (x,y) در مسیر p از u به v در T است. یک درخت پوشای کمینه‌ی T' که شامل (u,v) است با حذف یال (x,y) از T و اضافه کردن یال سبک (u,v) ساخته می‌شود.

اثبات فرض کنید T یک درخت پوشای کمینه باشد که A را در بر می‌گیرد، و همچنین T شامل (u,v) نمی‌شود، چرا که اگر این طور باشد، اثبات تمام شده است. با استفاده از یک تکنیک برش و چسباندن، یک درخت پوشای کمینه‌ی دیگر T' خواهیم ساخت که $A\cup\{(u,v)\}$ را در بر می‌گیرد، و بدین شکل اثبات می‌کنیم که (u,v) برای A ایمن است.

یال (u,v) به همراه یال‌های بر روی مسیر ساده‌ی p از u به v در T، یک دور را تشکیل می‌دهد، همان طور که در شکل ۲۳-۳ مشخص شده است. چون u و v بر روی جهات مختلف برش $(S,V-S)$ قرار دارند، حداقل یک یال در T بر روی مسیر p قرار دارد که از برش $(S,V-S)$ عبور می‌کند. فرض کنید (x,y) چنین یالی باشد. یال (x,y) در A نیست، چرا که برش به A احترام می‌گذارد. از آن جایی که (x,y) بر روی مسیر یکتای از u به v در T است، حذف یال (x,y) درخت T را به دو مؤلفه می‌شکند. اضافه کردن (u,v) دوباره این دو مؤلفه را به یکدیگر متصل می‌کند و یک درخت پوشای جدید $T'=T-\{(x,y)\}\cup\{(u,v)\}$ می‌سازد.

اکنون نشان می‌دهیم که T' یک درخت پوشای کمینه است. از آن جایی که (u,v) یک یال سبک گذرا از $(S,V-S)$ است، و یال (x,y) هم از این برش عبور می‌کند، $w(u,v)\le w(x,y)$. بنابراین،

$$\begin{aligned} w(T') &= w(T)-w(x,y)+w(u,v) \\ &\le w(T) \end{aligned}$$

ولی T یک درخت پوشای کمینه است، بنابراین $w(T') \leq w(T)$؛ پس T' هم باید یک درخت پوشای کمینه باشد.

این باقی می‌ماند که نشان دهیم (u,v) واقعاً یک یال ایمن برای A است. داریم $A \subseteq T'$، چرا که $A \subseteq T$ و $(x,y) \notin A$؛ بنابراین $A \cup \{(u,v)\} \subseteq T'$. در نتیجه از آن جایی که T' یک درخت پوشای کمینه است، (u,v) برای A ایمن است.

قضیه‌ی ۲۳-۱ درکی بهتر از کارکرد الگوریتم GENERIC-MST بر روی یک گراف همبند $G = (V,E)$ می‌دهد. همان طور که الگوریتم پیش می‌رود، مجموعه‌ی A همیشه بدون دور باقی می‌ماند؛ در غیر این صورت یک درخت پوشای کمینه شامل A حاوی یک دور خواهد بود، که تناقض است. در هر زمانی از اجرای الگوریتم، گراف $G_A = (V, A)$ یک جنگل است، و هر یک از مؤلفه‌های همبندی G_A یک درخت است. (بعضی از درخت‌ها ممکن است فقط شامل یک رأس باشند، به عنوان مثال وقتی الگوریتم تازه آغاز می‌شود، A تهی است و جنگل شامل $|V|$ درخت است، یکی برای هر رأس.) به علاوه هر یال ایمن (u,v) برای A مؤلفه‌های جداگانه‌ای از G_A را به هم متصل می‌کند، چرا که $A \cup \{(u,v)\}$ باید بدون دور باشد.

حلقه‌ی خطوط ۲-۴ GENERIC-MST دقیقاً $|V|-1$ بار اجرا می‌شود، و $|V|-1$ یال درخت پوشای کمینه به صورت پشت سر هم تعیین می‌شوند. در ابتدا وقتی $A = \emptyset$، $|V|$ درخت در G_A وجود دارد، و هر تکرار این تعداد را یکی کاهش می‌دهد. وقتی جنگل فقط حاوی یک درخت است، الگوریتم پایان می‌یابد.

دو الگوریتم بخش ۲۳-۲ از نتیجه‌ی زیر از قضیه‌ی ۲۳-۱ استفاده می‌کنند.

نتیجه‌ی ۲۳-۲ فرض کنید $G = (V,E)$ یک گراف بدون جهت همبند باشد، با یک تابع وزن w با مقدار حقیقی که بر روی E تعریف شده است. فرض کنید A یک زیرمجموعه از E باشد که یک درخت پوشای کمینه در G آن را در بر می‌گیرد، و $C = (V_C, E_C)$ یک مؤلفه‌ی همبندی (درخت) در جنگل $G_A = (V, A)$. اگر (u,v) یک یال سبک باشد که C را به یک مؤلفه‌ی همبندی دیگر در G_A متصل می‌کند، آن گاه (u,v) برای A ایمن است.

اثبات برش $(V_C, V - V_C)$ به A احترام می‌گذارد، و (u,v) یک یال سبک برای این برش است. بنابراین (u,v) برای A ایمن است.

تمرین‌ها

۲۳-۱-۱ فرض کنید (u,v) یک یال با وزن کمینه در گراف G باشد. نشان دهید که (u,v) به یک درخت پوشای کمینه در G تعلق دارد.

۲۳-۱-۲ پروفسور Sabatier عکس قضیه‌ی ۲۳-۱ را به صورت زیر حدس می‌زند. فرض کنید $G=(V,E)$ یک گراف بدون جهت همبند باشد، با یک تابع وزن w با مقدار حقیقی که بر روی E تعریف شده است. فرض کنید A یک زیرمجموعه از E باشد که یک درخت پوشای کمینه در G آن را در بر می‌گیرد، و $(S,V-S)$ یک برش از G باشد که به A احترام می‌گذارد، و (u,v) یک یال ایمن برای A که از $(S,V-S)$ عبور می‌کند. آن گاه (u,v) یک یال سبک برای برش است. با ارائه‌ی یک مثال نقض، نشان دهید که حدس پروفسور درست نیست.

۲۳-۱-۳ نشان دهید که اگر یال (u,v) در یک درخت پوشای کمینه باشد، آن گاه این یال یک یال سبک گذرا از یک برش بر روی گراف است.

۲۳-۱-۴ یک مثال ساده از یک گراف همبند بدهید که مجموعه‌ی یال‌های $\{(u,v)$: یک برش $(S,V-S)$ وجود دارد به طوری که (u,v) یک یال سبک گذرا از $(S,V-S)$ است $\}$ یک درخت پوشای کمینه را تشکیل نمی‌دهد.

۲۳-۱-۵ فرض کنید e یک یال با وزن بیشینه بر روی یک دور از گراف همبند $G=(V,E)$ باشد. اثبات کنید که یک درخت پوشای کمینه بر روی $G'=(V,E-\{e\})$ وجود دارد که یک درخت پوشای کمینه برای G هم هست. یعنی، یک درخت پوشای کمینه برای G وجود دارد که شامل e نیست.

۲۳-۱-۶ نشان دهید که یک گراف یک درخت پوشای کمینه‌ی یکتا دارد اگر برای هر برش گراف، یک یال سبک یکتا وجود داشته باشد که از آن برش عبور می‌کند. با دادن یک مثال نقض نشان دهید که عکس این قضیه درست نیست.

۲۳-۱-۷ بحث کنید اگر وزن تمام یال‌های یک گراف مثبت باشد، آن گاه هر زیرمجموعه‌ای از یال‌ها که تمام رأس‌ها را به هم متصل می‌کند و کم‌ترین وزن ممکن را دارد، باید یک درخت باشد. یک مثال بدهید که نشان می‌دهد اگر وزن منفی هم داشته باشیم این نتیجه‌گیری درست نیست.

۲۳-۱-۸ فرض کنید T یک درخت پوشای کمینه برای گراف G باشد، و L یک لیست مرتب شده از وزن یال‌ها در T. نشان دهید که برای هر درخت پوشای کمینه‌ی T' از G، لیست L لیست مرتب شده‌ی وزن یال‌ها در T' هم هست.

۲۳-۱-۹ فرض کنید T یک درخت پوشای کمینه برای یک گراف $G=(V,E)$ باشد، و V' یک زیرمجموعه از V. فرض کنید T' زیرگرافی از T باشد که توسط V' القا می‌شود، و G' یک زیرگراف از G القا شده توسط V'. نشان دهید که اگر T' همبند باشد، آن گاه T' یک درخت پوشای کمینه برای G' است.

۲۳-۱-۱۰ با داشتن یک گراف G و یک درخت پوشای کمینه‌ی T، فرض کنید وزن یکی از یال‌ها را در T کاهش می‌دهیم. نشان دهید که T همچنان یک درخت پوشای کمینه برای G است. به صورت رسمی‌تر، فرض کنید T یک درخت پوشای کمینه برای G با تابع وزن w باشد. یک یال $(x, y) \in T$ و یک عدد صحیح k انتخاب می‌کنیم، و تابع وزن w' را به صورت

$$w'(u,v) = \begin{cases} w(u,v) & \text{اگر } (u,v) \neq (x,y) \\ w(x,y) - k & \text{اگر } (u,v) = (x,y) \end{cases}$$

تعریف می‌کنیم. نشان دهید که T یک درخت پوشای کمینه برای G با تابع وزن w' است.

★ ۲۳-۱-۱۱ با داشتن یک گراف G و یک درخت پوشای کمینه‌ی T، فرض کنید وزن یکی از یال‌هایی را که در T نیست کاهش می‌دهیم. یک الگوریتم بدهید که درخت پوشای کمینه را برای این گراف اصلاح شده می‌یابد.

۲۳-۲ الگوریتم‌های کروسکال و پرایم

دو الگوریتم درخت پوشای کمینه‌ی ارائه شده در این بخش، نسخه‌های کاملی از الگوریتم کلی مربوطه هستند. هر کدام از آن‌ها از یک قانون خاص برای تعیین یک یال ایمن در خط ۳ رویه‌ی GENERIC-MST استفاده می‌کنند. در الگوریتم کروسکال، مجموعه‌ی A یک جنگل است که رأس‌های آن، همگی جزء رأس‌های گراف داده شده هستند. یال ایمنی که به A اضافه می‌شود همیشه یک یال با وزن کمینه در گراف است که دو مؤلفه‌ی جدا را به هم متصل می‌کند. در الگوریتم پرایم، مجموعه‌ی A همیشه فقط یک درخت را تشکیل می‌دهد. یال ایمنی که به A اضافه می‌شود همیشه یک یال با وزن کمینه است که درخت را به یک رأس که در درخت نیست متصل می‌کند.

الگوریتم کروسکال

الگوریتم کروسکال در هر مرحله یال ایمن (u, v) را با یافتن یک یال با وزن کمینه از میان تمام یال‌هایی که دو درخت مختلف را در جنگل به هم متصل می‌کنند، تعیین می‌کند. فرض کنید C_1 و C_2 نشان‌دهنده‌ی دو درخت باشند که توسط (u, v) به هم متصل می‌شوند. از آن جایی که (u, v) باید یک یال سبک باشد که C_1 را به یک درخت دیگر متصل می‌کند، نتیجه‌ی ۲۳-۲ ایجاب می‌کند که (u, v) برای C_1 یک یال ایمن است. الگوریتم کروسکال یک الگوریتم حریصانه است، چرا که در هر مرحله یک یال با وزن کمینه را به جنگل اضافه می‌کند.

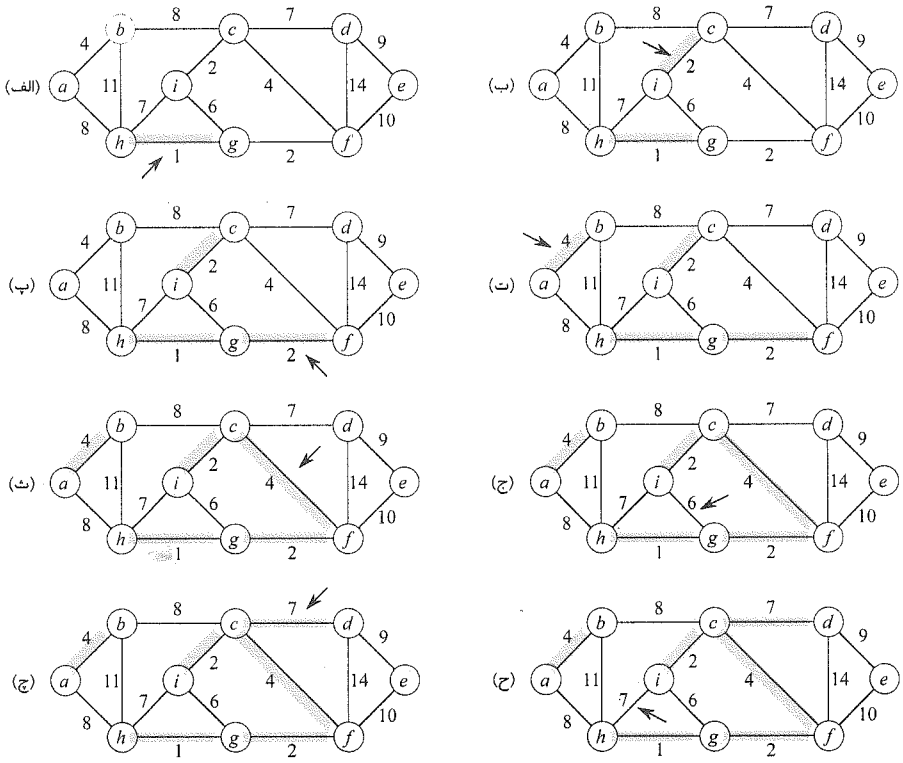
پیاده‌سازی ما از الگوریتم کروسکال مانند الگوریتم محاسبه‌ی مؤلفه‌های همبندی بخش ۲۱-۱ است. این الگوریتم از یک ساختمان داده‌ی مجموعه‌های منفصل برای نگه‌داری مجموعه‌های منفصل مختلفی از عناصر استفاده می‌کند. هر مجموعه حاوی یک درخت از جنگل فعلی است. عملیات

FIND-SET(u) یک عنصر نماینده از مجموعه‌ی حاوی u بازمی‌گرداند. بنابراین می‌توانیم با بررسی این که آیا FIND-SET(u) برابر با FIND-SET(v) هست یا خیر، تعیین کنیم که آیا دو رأس u و v به یک درخت تعلق دارند یا نه. ترکیب درخت‌ها به وسیله‌ی UNION انجام می‌شود.

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  sort the edges of E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  return A
```

کارکرد الگوریتم کروسکال در شکل ۲۳-۴ نشان داده شده است. خطوط ۱-۳ مجموعه‌ی A را به صورت یک مجموعه‌ی تهی مقداردهی اولیه می‌کند و $|V|$ درخت می‌سازد، هر یک حاوی یک رأس. حلقه‌ی for در خطوط ۵-۸ برای هر یال (u,v) چک می‌کند که آیا نقاط پایانی به یک درخت تعلق دارند یا نه. اگر این طور باشد، آن گاه یال (u,v) نمی‌تواند بدون ایجاد یک دور به جنگل اضافه شود، و از این یال صرف نظر می‌شود. در غیر این صورت دو رأس به درخت‌های مختلفی تعلق دارند. در این حالت در خط ۷ یال (u,v) به A اضافه می‌شود، و رأس‌های دو درخت در خط ۸ با هم ادغام می‌شوند.

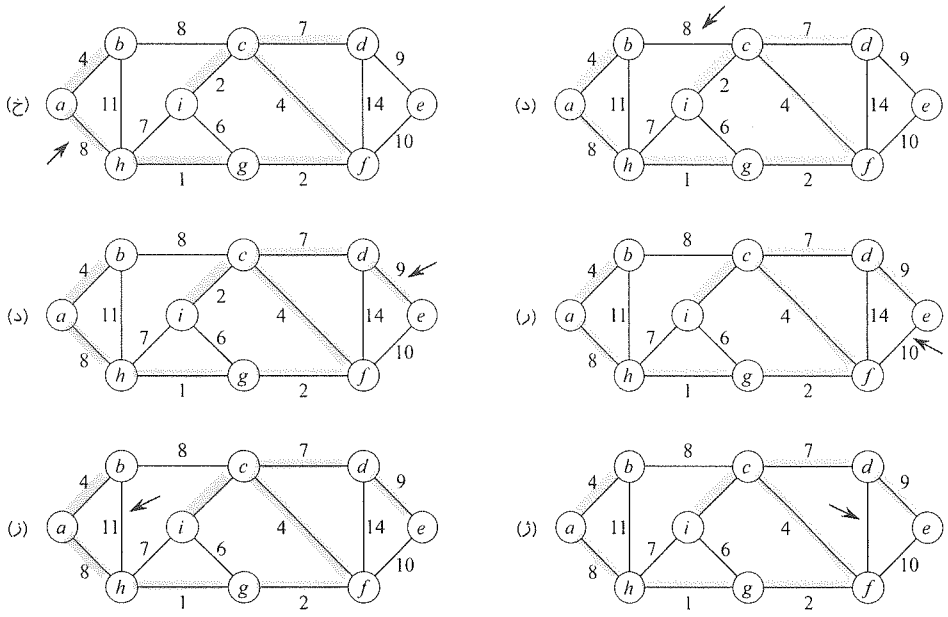
زمان اجرای الگوریتم کروسکال برای یک گراف $G=(V,E)$ به پیاده‌سازی ساختمان داده‌ی مجموعه‌های منفصل بستگی دارد. ما در این جا پیاده‌سازی جنگل مجموعه‌های منفصل بخش ۲۱-۳ را در نظر می‌گیریم، با مکاشفه‌ی اجتماع بر حسب رتبه، چرا که این پیاده‌سازی به صورت حدی سریع‌ترین پیاده‌سازی شناخته شده است. مقداردهی اولیه مجموعه‌ی A در خط ۱ به زمان $O(1)$ نیازدارد، و زمان مرتب سازی یال‌ها در خط ۴ برابر $O(E \lg E)$ است. (هزینه‌ی $|V|$ عملیات MAKE-SET در حلقه‌ی for خطوط ۲-۳ را هم به زودی محاسبه می‌کنیم.) حلقه‌ی for خطوط ۵-۸، $O(E)$ عملیات FIND-SET و UNION بر روی جنگل مجموعه‌های منفصل انجام می‌دهد. به همراه $|V|$ عملیات MAKE-SET. این اعمال در زمان کلی $O((V+E)\alpha(V))$ اجرا می‌شوند، که در آن α یک تابع با سرعت رشد بسیار کم است که در بخش ۲۱-۴ تعریف شده است. چون فرض بر این است که G همبند باشد، داریم $|E| \geq |V|-1$، و بنابراین اعمال مجموعه‌های منفصل $O(E\alpha(V))$ زمان می‌برند. به علاوه از آن جایی که $\alpha(|V|)=O(\lg V)=O(\lg E)$، کل زمان اجرای الگوریتم کروسکال $O(E \lg E)$ است. با توجه به این که $|E|<|V|^2$، داریم $\lg|E|=O(\lg V)$، و بنابراین می‌توانیم زمان اجرای الگوریتم کروسکال را به صورت $O(E \lg V)$ بازنویسی کنیم.



شکل ۲۳-۴ اجرای الگوریتم کروسکال بر روی گراف شکل ۲۳-۱. یال‌های سایه‌دار به جنگل A که در حال رشد است تعلق دارند. در الگوریتم یال‌ها به ترتیب وزن در نظر گرفته می‌شوند. در هر مرحله، یک پیکان به یالی اشاره می‌کند که در حال بررسی است. اگر یال دو درخت مختلف را در جنگل به هم متصل کند، این یال به جنگل اضافه می‌شود، و دو درخت را با هم ادغام می‌کند.

الگوریتم پرایم

الگوریتم پرایم مانند الگوریتم کروسکال حالت خاصی از الگوریتم کلی درخت پوشای کمینه‌ی بخش ۲۳-۱ است. الگوریتم پرایم بسیار مانند الگوریتم Dijkstra برای یافتن کوتاه‌ترین مسیرها در یک گراف عمل می‌کند، که آن را در بخش ۲۴-۳ خواهیم دید. الگوریتم پرایم این خصوصیت را دارد که یال‌های مجموعه‌ی A همیشه یک درخت را تشکیل می‌دهند. همان طور که در شکل ۲۳-۵ نشان داده شده است، این درخت از یک رأس ریشه‌ی دلخواه r آغاز می‌شود و به رشد کردن ادامه می‌دهد تا زمانی که تمام رأس‌های V را بپوشاند. در هر مرحله یک یال سبک که A را به یک رأس تنهای $G_A = (V, A)$ متصل می‌کند به درخت A افزوده می‌شود. طبق نتیجه‌ی ۲۳-۲ این قانون فقط یال‌هایی را به A اضافه می‌کند که برای آن ایمن هستند؛ بنابراین وقتی الگوریتم پایان می‌یابد، یال‌های A یک درخت پوشای کمینه را تشکیل می‌دهند. این استراتژی حریصانه است چرا که در هر مرحله درخت با یک یال تکمیل می‌شود که کم‌ترین مقدار ممکن را به وزن درخت اضافه می‌کند.



شکل ۲۳-۴ (ادامه)

کلید پیاده‌سازی الگوریتم پرایم به صورت بهینه این است که انتخاب یالی را که می‌خواهیم به درخت A اضافه کنیم، تا حد ممکن ساده کنیم. در شبه‌کد زیر گراف همبند G و ریشه‌ی درخت پوشای کمینه‌ی در حال رشد(r) ورودی‌های الگوریتم هستند. حین اجرای الگوریتم تمام رأس‌هایی که در درخت *نیستند* در یک صف اولویت کمینه‌ی Q قرار دارند، که در آن اولویت بر حسب فیلد key تعیین می‌شود. برای هر رأس v خصیصه‌ی $v.key$ یال با وزن کمینه از میان تمام یال‌هایی است که v را به یک رأس در درخت متصل می‌کنند؛ طبق قرارداد اگر چنین یالی وجود نداشته باشد، $v.key = \infty$. فیلد $v.\pi$ پدر v را در درخت مشخص می‌کند. هنگام اجرای الگوریتم، مجموعه‌ی A در GENERIC-MST به صورت ضمنی به شکل

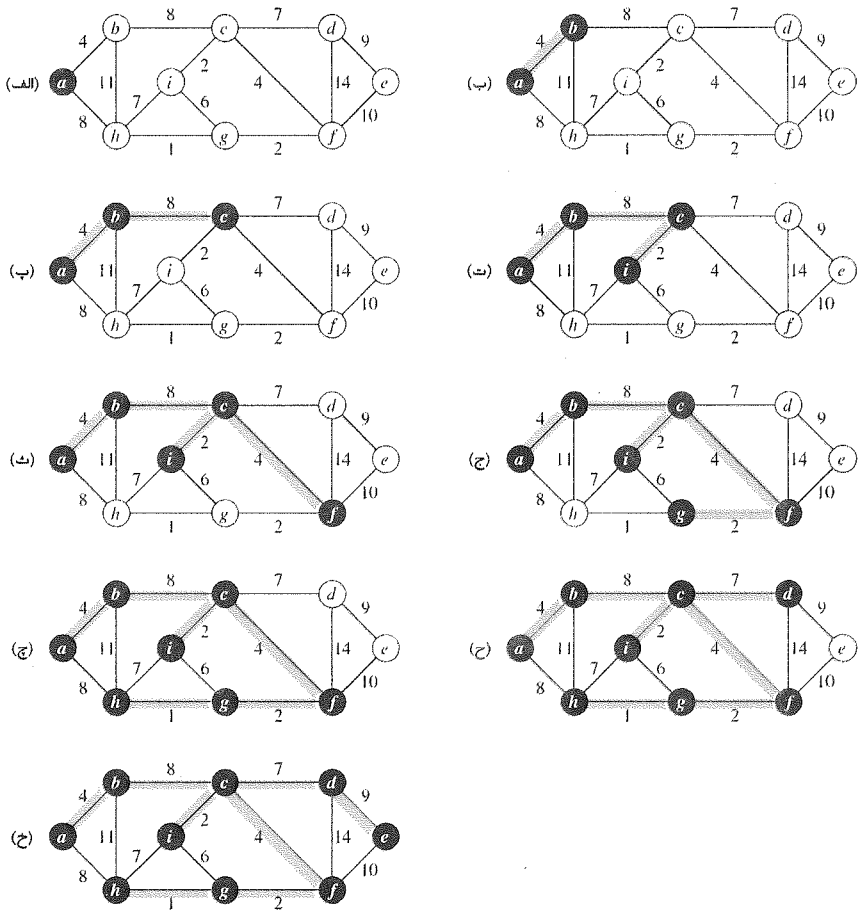
$$A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$$

نگه داشته می‌شود. وقتی الگوریتم پایان می‌یابد، صف اولویت کمینه‌ی Q تهی است؛ بنابراین درخت پوشای کمینه‌ی A برای G برابر است با

$$A = \{(v, v.\pi) : v \in V - \{r\}\}$$

```
MST-PRIM(G, w, r)
1  for each u ∈ G.V
2      u.key = ∞
3      u.π = NIL
4  r.key = 0
```

```
5  Q = G.V
6  while Q ≠ Ø
7      u = EXTRACT-MIN(Q)
8      for each v ∈ G.Adj[u]
9          if v ∈ Q and w(u, v) < v.key
10             v.π = u
11             v.key = w(u, v)
```



شکل ۲۳-۵ اجرای الگوریتم پرایم بر روی گراف شکل ۲۳-۱. رأس a ریشه است. یال‌های سایه‌دار آن‌هایی هستند که در درخت در حال رشد قرار دارند، و رأس‌های درخت با رنگ تیره مشخص شده‌اند. در هر مرحله از الگوریتم، رأس‌های درخت یک برش از گراف را تشکیل می‌دهند، و یک یال سبک گذرا از این برش به درخت اضافه می‌شود. به عنوان مثال، در مرحله‌ی دوم الگوریتم می‌توانند میان یال‌های (b, c) و (a, h) یکی را برای اضافه کردن به درخت اضافه کند چرا که هر دو یال، یال‌های سبکی هستند که از برش عبور می‌کنند.

عملکرد الگوریتم پرایم در شکل ۲۳-۵ نشان داده شده است. خطوط ۱-۵ کلید هر رأس (غیر از ریشه‌ی r که کلید آن با ۰ مقداردهی می‌شود تا اولین رأسی باشد که پردازش می‌شود) را با ∞ و پدر هر رأس را با NIL مقداردهی می‌کنند، و صف پوشای کمینه‌ی Q را طوری مقداردهی می‌کنند تا تمام رأس‌ها را شامل شود. الگوریتم ثابت حلقه‌ی سه بخشی زیر را حفظ می‌کند:

قبل از هر تکرار حلقه‌ی while خطوط ۶-۱۱،

۱. $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$.
۲. رأس‌هایی که در درخت پوشای کمینه قرار گرفته‌اند، رأس‌های $V - Q$ هستند.
۳. برای تمام رأس‌های $v \in Q$، اگر $v.\pi \neq \text{NIL}$ آن گاه $v.key < \infty$، و $v.key$ وزن یک یال سبک $(v, v.\pi)$ است که v را به یک رأس که هم اکنون در درخت پوشای کمینه قرار دارد، متصل می‌کند.

خط ۷ رأس $u \in Q$ را که مجاور یک یال گذرا از برش $(V - Q, Q)$ است، مشخص می‌کند (غیر از اولین تکرار، که به خاطر خط ۴، $u = r$). حذف u از مجموعه‌ی Q آن را به مجموعه‌ی $V - Q$ از رأس‌های درخت اضافه می‌کند، و در نتیجه $(u, u.\pi)$ هم به A اضافه می‌شود. حلقه‌ی for خطوط ۸-۱۱ فیلدهای key و π تمام رأس‌های v مجاور u را که در درخت نیستند به هنگام سازی می‌کند، که این به هنگام سازی قسمت سوم ثابت حلقه را حفظ می‌کند.

کارایی الگوریتم پرایم به پیاده‌سازی صف اولویت کمینه‌ی Q بستگی دارد. اگر Q به صورت یک هرم کمینه‌ی دودویی (فصل ۶ را ببینید) پیاده‌سازی شود، می‌توانیم از رویه‌ی BUILD-MIN-HEAP برای انجام مقداردهی اولیه در خطوط ۱-۵ در زمان $O(V)$ استفاده کنیم. بدنه‌ی حلقه‌ی while، $|V|$ بار اجرا می‌شود، و از آن جایی که هر عملیات EXTRACT-MIN به $O(\lg V)$ زمان نیاز دارد، کل زمان مورد نیاز برای فراخوانی‌های EXTRACT-MIN برابر است با $O(V \lg V)$. حلقه‌ی for خطوط ۸-۱۱ در کل $O(E)$ بار اجرا می‌شود، چرا که مجموع طول تمام لیست‌های مجاورت $|E|$ است. در حلقه‌ی for، تست عضویت در Q را در خط ۹ می‌توان با نگه داشتن یک بیت که می‌گوید هر رأس عضو Q هست یا خیر، در زمان ثابت انجام داد، و این بیت را می‌توان با حذف یک رأس از Q به هنگام سازی کرد. انتساب خط ۱۱ شامل یک عملیات ضمنی DECREASE-KEY بر روی هرم کمینه است، که می‌توان آن را بر روی یک هرم کمینه‌ی دودویی در زمان $O(\lg V)$ پیاده‌سازی کرد. بنابراین کل زمان الگوریتم پرایم $O(V \lg V + E \lg V) = O(E \lg V)$ است، که به صورت حدی با زمان اجرای الگوریتم کروسکال برابر است.

با این حال می‌توان زمان اجرای حدی الگوریتم پرایم را با استفاده از هرم‌های فیبوناچی بهبود بخشید. فصل ۱۹ نشان می‌دهد که اگر $|V|$ عنصر در یک هرم فیبوناچی قرار داشته باشند، می‌توان یک عملیات EXTRACT-MIN را در زمان سرشکن $O(\lg V)$، و یک عملیات DECREASE-KEY را (برای پیاده‌سازی خط ۱۱) در زمان سرشکن $O(1)$ پیاده‌سازی کرد. بنابراین اگر از یک هرم فیبوناچی برای

پیاده‌سازی صف اولویت کمینه‌ی Q استفاده کنیم، زمان اجرای الگوریتم پرایم به $O(E + V \lg V)$ بهبود می‌یابد.

تمرین‌ها

۲۳-۲-۱ الگوریتم کروسکال می‌تواند درخت‌های پوشای کمینه‌ی مختلفی را برای یک گراف بازگرداند، بسته به این که زمانی مرتب‌سازی یال‌ها چگونه تساوی‌ها را از بین می‌بریم. نشان دهید که برای هر درخت پوشای کمینه‌ی T از G، یک راه برای مرتب کردن یال‌های G وجود دارد به طوری که الگوریتم کروسکال T را بازگرداند.

۲۳-۲-۲ فرض کنید گراف $G = (V, E)$ به صورت یک ماتریس مجاورت نمایش داده شده است. یک پیاده‌سازی ساده برای الگوریتم پرایم برای این حالت بدهید که در زمان $O(V^2)$ اجرا می‌شود.

۲۳-۲-۳ آیا برای یک گراف خلوت $G = (V, E)$، که در آن $|E| = \theta(V)$، پیاده‌سازی هرم فیبوناچی برای الگوریتم پرایم به صورت حدی سریع‌تر از پیاده‌سازی آن به کمک یک هرم دودویی است؟ در مورد یک گراف شلوغ، که در آن $|E| = \theta(V^2)$ چطور؟ برای این که پیاده‌سازی هرم فیبوناچی به صورت حدی از پیاده‌سازی هرم دودویی سریع‌تر باشد، رابطه‌ی $|E|$ و $|V|$ باید چگونه باشد؟

۲۳-۲-۴ فرض کنید که وزن تمام یال‌ها در یک گراف اعداد صحیحی در بازه‌ی ۱ تا $|V|$ هستند. سرعت الگوریتم کروسکال را تا چه حد می‌توانید افزایش دهید؟ اگر وزن یال‌ها اعدادی صحیح در بازه‌ی ۱ تا W باشد، که W یک ثابت است، چطور؟

۲۳-۲-۵ فرض کنید که وزن تمام یال‌ها در یک گراف اعداد صحیحی در بازه‌ی ۱ تا $|V|$ هستند. سرعت الگوریتم پرایم را تا چه حد می‌توانید افزایش دهید؟ اگر وزن یال‌ها اعدادی صحیح در بازه‌ی ۱ تا W باشد، که W یک ثابت است، چطور؟

★ ۲۳-۲-۶ فرض کنید که وزن یال‌ها در یک گراف به صورت یکنواخت بر روی یک بازه‌ی نیمه باز $[0,1)$ توزیع شده‌اند. با اصلاح الگوریتم‌های کروسکال و پرایم، کدام یک از آن‌ها را می‌توانید سریع‌تر بر روی این گراف اجرا کنید؟

★ ۲۳-۲-۷ فرض کنید یک گراف G یک درخت پوشای کمینه دارد که قبلاً محاسبه شده است. اگر یک رأس جدید به G اضافه شود، با چه سرعتی می‌توان این درخت پوشای کمینه را به هنگام سازی کرد؟

۲۳-۲-۸ پروفسور Toole یک الگوریتم تقسیم و حل جدید برای محاسبه‌ی درخت‌های پوشای کمینه ارائه کرده است. با داشتن یک گراف $G = (V, E)$، مجموعه‌ی رأس‌های V را به دو مجموعه‌ی V_1 و V_2 تقسیم کنید به طوری که $|V_1|$ و $|V_2|$ حداکثر یکی اختلاف داشته

باشند. فرض کنید E_1 مجموعه‌ی یال‌هایی باشد که فقط با رأس‌های V_1 مجاورند، و E_2 مجموعه‌ی یال‌هایی که فقط با رأس‌های V_2 مجاورند. به صورت بازگشتی مسئله‌ی درخت پوشای کمینه را برای هر یک از دو زیرگراف $G_1 = (V_1, E_1)$ و $G_2 = (V_2, E_2)$ حل کنید. نهایتاً یال با وزن کمینه در E را انتخاب کنید که از برش (V_1, V_2) عبور می‌کند، و از این یال برای اجتماع دو درخت پوشای کمینه‌ی حاصل و تبدیل آن به یک درخت پوشای کمینه استفاده کنید.

بحث کنید که این الگوریتم به درستی یک درخت پوشای کمینه برای G تولید می‌کند، و یا با یک مثال نقض آن را رد کنید.

مسائل

۲۳-۱ دومین درخت پوشای کمینه

فرض کنید $G = (V, E)$ یک گراف بدون جهت همبند باشد، با یک تابع وزن $w : E \to \mathbb{R}$، و همچنین فرض کنید که $|E| \geq |V|$ و وزن تمام یال‌ها یکتا است.

یک دومین درخت پوشای کمینه به صورت زیر تعریف می‌شود. فرض کنید T مجموعه‌ی تمام درخت‌های پوشای کمینه‌ی G باشد، و T' یک درخت پوشا در G. آن گاه یک *دومین درخت پوشای کمینه*، یک درخت پوشای T است به طوری که $w(T) = \min_{T'' \in T - \{T'\}} \{w(T'')\}$.

I. نشان دهید که درخت پوشای کمینه یکتا است، ولی دومین درخت پوشای کمینه لزوماً یکتا نیست.

II. فرض کنید که T یک درخت پوشا بر روی G باشد. اثبات کنید که یال‌های $(u, v) \in T$ و $(x, y) \notin T$ وجود دارند به طوری که $T - \{(u, v)\} \cup \{(x, y)\}$ یک دومین درخت پوشای کمینه بر روی G است.

III. فرض کنید T یک درخت پوشا بر روی G باشد، و برای هر دو رأس $u, v \in V$، فرض کنید $max[u, v]$ یک یال با وزن بیشینه بر روی مسیر یکتای میان u و v در T باشد. یک الگوریتم با زمان $O(V^2)$ ارائه دهید که، با دریافت T، $max[u, v]$ را برای تمام $u, v \in V$ محاسبه می‌کند.

IV. یک الگوریتم بهینه برای محاسبه‌ی دومین درخت پوشای کمینه‌ی G ارائه کنید.

۲۳-۲ درخت پوشای کمینه در گراف‌های خلوت

برای یک گراف همبند بسیار خلوت $G = (V, E)$، می‌توان به کمک هرم‌های فیبوناچی و پیش پردازش G برای کاهش تعداد رأس‌ها قبل از اجرای الگوریتم پرایم، زمان

اجرای $O(E+V \lg V)$ مربوط به این الگوریتم را باز هم بهبود بخشید. به طور خاص برای هر رأس u، یال (u,v) مجاور u با وزن کمینه را انتخاب کرده و آن را به درخت پوشای کمینه‌ی در حال ساخت اضافه می‌کنیم. سپس تمام یال‌های انتخاب شده را کاهش (contract) می‌دهیم (بخش ب-۴ را ببینید). به جای کاهش دادن این یال‌ها به ترتیب، ابتدا مجموعه‌هایی از رأس‌ها را تعیین می‌کنیم که در یک رأس جدید یکسان مجتمع می‌شوند. سپس گرافی را می‌سازیم که از کاهش یال‌ها به صورت به ترتیب به دست می‌آید، ولی این کار را با «نام گذاری دوباره‌ی» یال‌ها بر حسب مجموعه‌هایی که نقاط پایانی آن‌ها در آن قرار دارند، انجام می‌دهیم. یال‌های متعددی از گراف ممکن است با نام یکسانی مشخص شوند. در چنین حالتی فقط یک یال خواهیم داشت، و وزن آن برابر است با وزن کمینه از میان یال‌های اصلی مربوطه.

در ابتدا درخت پوشای کمینه‌ی در حال ساخت را با تهی مقداردهی می‌کنیم، و برای هر یال $(u,v) \in E$ قرار می‌دهیم $(u,v).orig = (u,v)$ و $(u,v).c = w(u,v)$. از خصیصه‌ی $orig$ برای ارجاع یال مربوطه از گراف اصلی به یال گراف کاهش داده شده استفاده می‌کنیم. خصیصه‌ی c وزن یک یال را نگه می‌دارد، و همین طور که یال‌ها کاهش داده می‌شوند، طبق روش بالا برای انتخاب وزن یال‌ها به هنگام‌سازی می‌شود. رویه‌ی MST-REDUCE ورودی‌های G و T را دریافت کرده، یک گراف کاهش یافته‌ی G' را بازمی‌گرداند و خصیصه‌های $orig'$ و c' را برای G' به هنگام‌سازی می‌کند. این رویه همچنین یال‌های G را در درخت پوشای کمینه‌ی T جمع آوری می‌کند.

```
MST-REDUCE(G, T)
 1  for each v ∈ G.V
 2      v.mark = FALSE
 3      MAKE-SET(v)
 4  for each u ∈ G.V
 5      if u.mark == FALSE
 6          choose v ∈ G.Adj[u] such that (u,v).c is minimized
 7          UNION(u, v)
 8          T = T ∪ {(u, v).orig}
 9          u.mark = v.mark = TRUE
10  G'.V = {FIND-SET(v) : v ∈ G.V}
11  G'.E = Ø
12  for each (x, y) ∈ G.E
13      u = FIND-SET(x)
14      v = FIND-SET(y)
15      if (u, v) ∉ G'.E
16          G'.E = G'.E ∪ {(u,v)}
17          (u,v).orig' = (x,y).orig
18          (u,v).c' = (x,y).c
19      elseif (x, y).c < (u,v).c'
```

```
20        (u,v).orig' = (x,y).orig
21        (u,v).c' = (x,y).c
22  construct adjacency lists G'.Adj for G'
23  return G' and T
```

I. فرض کنید T مجموعه‌ی یال‌های بازگردانده شده توسط MST-REDUCE باشد، و A درخت پوشای کمینه‌ی G' تشکیل شده توسط MST-PRIME (G', c', r)، که در آن c' خصیصه‌ی وزن در یال‌های $G'.E$ است و r یک رأس دلخواه در $G'.V$. اثبات کنید که $T \cup \{(x, y).orig' \mid (x, y) \in A\}$ یک درخت پوشای کمینه‌ی G است.

II. بحث کنید که $|G'.V| \le |V|/2$.

III. نشان دهید که چطور می‌توان MST-REDUCE را پیاده‌سازی کرد به گونه‌ای که در زمان $O(E)$ اجرا شود. (*راهنمایی:* از ساختمان‌های داده‌ی ساده استفاده کنید.)

IV. فرض کنید MST-REDUCE را k بار اجرا می‌کنیم، که در آن از خروجی G' تولید شده توسط یک بار اجرا، به عنوان ورودی G اجرای بعد استفاده می‌شود، و یال‌ها درون T جمع‌آوری می‌شوند. بحث کنید که کل زمان k بار اجرا $O(kE)$ است.

V. فرض کنید بعد از k بار اجرای MST-REDUCE، مانند بخش IV، الگوریتم پرایم را با فراخوانی MST-PRIME (G', c', r) اجرا می‌کنیم، که در آن G' با خصیصه‌ی وزن c' توسط آخرین فراخوانی بازگردانده شده است، و r یک رأس دلخواه در $G'.V$ است. نشان دهید که چگونه می‌توان k را انتخاب کرد به طوری که کل زمان اجرا $O(E \lg\lg V)$ باشد. بحث کنید که انتخاب شما برای k زمان اجرا را کمینه می‌کند.

VI. برای چه مقادیری از $|E|$ (نسبت به $|V|$) الگوریتم پرایم به همراه پیش پردازش از الگوریتم پرایم بدون پیش پردازش سریع‌تر عمل می‌کند؟

۲۳-۳ درختان پوشای گلوگاه

یک *درخت پوشای گلوگاه* (bottleneck spanning tree) T برای یک گراف بدون جهت G، یک درخت پوشای G است که وزن بیشینه میان یال‌های آن در تمام درخت‌های پوشای G کمینه است. می‌گوییم مقدار درخت پوشای گلوگاه برابر است با وزن یال بیشینه در T.

I. بحث کنید که یک درخت پوشای کمینه، یک درخت پوشای گلوگاه است.

قسمت I نشان می‌دهد که یافتن یک درخت پوشای گلوگاه سخت‌تر از یافتن یک درخت پوشای کمینه نیست. در قسمت‌های بعدی نشان خواهیم داد که یک درخت پوشای گلوگاه را می‌توان در زمان خطی یافت.

II. یک الگوریتم با زمان خطی ارائه کنید که با دریافت یک گراف G و یک عدد صحیح b، تعیین می‌کند که آیا مقدار درخت پوشای گلوگاه حداکثر b است یا خیر.

III. از الگوریتم خود در بخش II به عنوان یک زیرروال در یک الگوریتم با زمان خطی برای

مسئله‌ی درخت پوشای گلوگاه استفاده کنید. (*راهنمایی:* ممکن است بخواهید از یک زیرروال استفاده کنید که مجموعه‌ی یال‌ها را کاهش می‌دهد، مانند رویه‌ی MST-REDUCE توصیف شده در مسئله‌ی ۲۳-۲.)

۲۳-۴ *الگوریتم‌های درخت پوشای کمینه‌ی جایگزین*

در این مسئله شبه‌کدهایی برای سه الگوریتم مختلف ارائه می‌کنیم. هر یک از این الگوریتم‌ها یک گراف همبند و یک تابع وزن را به عنوان ورودی دریافت کرده و زیرمجموعه‌ی T از یال‌های آن گراف را بازمی‌گرداند. برای هر الگوریتم، شما یا باید اثبات کنید که T یک درخت پوشای کمینه است، و یا اثبات کنید که T یک درخت پوشای کمینه نیست. همچنین بهینه‌ترین پیاده‌سازی را برای هر الگوریتم توصیف کنید، چه این الگوریتم یک درخت پوشای کمینه تولید می‌کند و چه نمی‌کند.

I.

```
MAYBE-MST-A(G, w)
1  sort the edges into nonincreasing order of edge weights w
2  T = E
3  for each edge e, taken in nonincreasing order by weight
4      if T − {e} is a connected graph
5          T = T − {e}
6  return T
```

II.

```
MAYBE-MST-B(G, w)
1  T = ∅
2  for each edge e, taken in arbitrary order
3      if T ∪ {e} has no cycles
4          T = T ∪ e
5  return T
```

III.

```
MAYBE-MST-C(G, w)
1  T = ∅
2  for each edge e, taken in arbitrary order
3      T = T ∪ {e}
4      if T has a cycle c
5          let e′ be the maximum-weight edge on c
6          T = T − {e′}
7  return T
```

فصل

۲۴

کوتاه‌ترین مسیرها از یک مبدأ

۲۴-۰ مقدمه

یک راننده می‌خواهد کوتاه‌ترین مسیر از شیکاگو (Chicago) به بوستون (Boston) را بیابد. با داشتن نقشه‌ی راه‌های ایالات متحده که روی آن فاصله‌ی بین هر جفت تقاطع مجاور نوشته شده است، چگونه می‌توان این مسیر را یافت؟

یک راه ممکن این است که تمام مسیرها از شیکاگو به بوستون را بررسی کنیم، فاصله‌ی هر مسیر را به دست آوریم، و سپس کوتاه‌ترین مسیر را انتخاب کنیم. با این حال به سادگی می‌توان دید که حتی اگر مسیرهایی را که دور دارند حذف کنیم، میلیون‌ها مسیر ممکن وجود دارد، که اکثر آن‌ها حتی ارزش بررسی را هم ندارند. به عنوان مثال یک مسیر از شیکاگو به هوستون (Houston) و سپس به بوستون مسلماً انتخابی ضعیف است، چرا که هوستون تقریباً هزار مایل خارج مسیر است.

در این فصل و فصل ۲۵ نشان خواهیم داد که چگونه می‌توان چنین مسائلی را به صورت بهینه حل کرد. در *یک مسئله‌ی کوتاه‌ترین مسیرها*، به ما یک گراف جهت‌دار و وزن‌دار $G = (V, E)$ داده شده است، به همراه تابع وزن $w : E \to \mathbb{R}$ که یال‌ها را به وزن‌های حقیقی مقدار نگاشت می‌کند. وزن یک مسیر $p = \langle v_0, v_1, \ldots, v_k \rangle$ برابر است با مجموع وزن‌های یال‌های تشکیل‌دهنده‌ی آن:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

وزن کوتاه‌ترین مسیر از u به v را به صورت

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{اگر یک مسیر از } u \text{ به } v \text{ وجود داشته باشد} \\ \infty & \text{در غیر این صورت} \end{cases}$$

تعریف می‌کنیم. در این صورت یک *کوتاه‌ترین مسیر* از رأس u به رأس v به صورت یک مسیر p با وزن $w(p) = \delta(u, v)$ تعریف می‌شود.

در مثال شیکاگو به بوستون، می‌توانیم نقشه‌ی جاده‌ها را به صورت یک گراف مدل کنیم: رأس‌ها نشان‌دهنده‌ی تقاطع‌ها هستند، یال‌ها نشان‌دهنده‌ی جاده‌های میان تقاطع‌ها، و وزن یال‌ها نشان‌دهنده‌ی طول جاده‌ها. هدف یافتن یک کوتاه‌ترین مسیر از یک تقاطع داده شده در شیکاگو به یک تقاطع داده شده در بوستون است.

وزن یال‌ها را می‌توان به عنوان ماتریس‌هایی غیر از فاصله هم در نظر گرفت. از آن‌ها معمولاً برای نمایش زمان، هزینه، جریمه، خسارت، و یا هر کمیت دیگری که در یک مسیر به صورت خطی افزوده می‌شود استفاده می‌شود، که هدف کمینه کردن آن است.

الگوریتم جستجوی سطح اول از بخش ۲۲-۲ یک الگوریتم کوتاه‌ترین مسیر است که برای گراف‌های بدون وزن کار می‌کند، یعنی گراف‌هایی که در آن‌ها وزن تمام یال‌ها واحد است. چون بسیاری از مفاهیم جستجوی سطح اول در یادگیری کوتاه‌ترین مسیرها در گراف‌های وزن‌دار پیش می‌آیند، به خواننده توصیه می‌شود که قبل از ادامه بخش ۲۲-۲ را بازبینی کند.

نسخه‌های مختلف

در این فصل بر روی *مسئله‌ی کوتاه‌ترین مسیرها از یک مبدأ* تمرکز می‌کنیم: با داشتن یک گراف $G = (V, E)$، می‌خواهیم کوتاه‌ترین مسیرها را از یک رأس *مبدأ* داده شده‌ی $s \in V$ به تمام رأس‌های دیگر $v \in V$ بیابیم. بسیاری از مسائل دیگر را می‌توان به کمک الگوریتم مسئله‌ی تک مبدأ حل کرد، شامل مسائل زیر:

- *مسئله‌ی کوتاه‌ترین مسیرها به یک مقصد:* یافتن یک کوتاه‌ترین مسیر از هر رأس v به یک رأس *مقصد* داده شده‌ی t. با برعکس کردن جهت هر یال در گراف، این مسئله به مسئله‌ی تک مبدأ تبدیل می‌شود.
- *مسئله‌ی کوتاه‌ترین مسیر میان دو رأس:* یافتن یک کوتاه‌ترین مسیر از u به v برای رأس‌های داده شده‌ی u و v. اگر مسئله‌ی تک مبدأ را با رأس مبدأ u حل کنیم، این مسئله را هم حل کرده‌ایم. به علاوه هیچ الگوریتمی برای این مسئله شناخته نشده است که در بدترین حالت به صورت حدی از بهترین الگوریتم‌های تک مبدأ سریع‌تر اجرا شود.
- *مسئله‌ی کوتاه‌ترین مسیر میان هر دو رأس:* یافتن یک کوتاه‌ترین مسیر میان u و v برای هر جفت رأس u و v. با این که این مسئله را می‌توان با اجرای الگوریتم تک مبدأ برای هر یک از رأس‌ها حل کرد، معمولاً می‌توان سریع‌تر هم آن را حل کرد. به علاوه ساختار آن به نوبه‌ی خود جذابیت‌های خاصی دارد. فصل ۲۵ مسئله‌ی کوتاه‌ترین مسیرها میان هر دو رأس را با جزئیات بررسی می‌کند.

زیرساختار بهینه‌ی یک کوتاه‌ترین مسیر

الگوریتم‌های کوتاه‌ترین مسیرها معمولاً بر این خصوصیت استوار هستند که یک کوتاه‌ترین مسیر میان دو رأس خود شامل کوتاه‌ترین مسیرهای دیگری است. (الگوریتم شار بیشینه‌ی ادموندز-کارپ در

فصل ۲۶ هم بر پایه‌ی این خصوصیت استوار است.) این خصوصیت زیرساختار بهینه نشانه‌ای است از کاربردی بودن هر دو روش برنامه‌ریزی پویا (فصل ۱۵) و متد حریصانه (فصل ۱۶). الگوریتم Dijkstra که آن را در بخش ۲۴-۳ خواهیم دید، یک الگوریتم حریصانه است، و الگوریتم فلوید-وارشال، که کوتاه‌ترین مسیرها را میان تمام رأس‌ها می‌یابد (فصل ۲۵ را ببینید)، یک الگوریتم برنامه‌ریزی پویا. لم زیر خصوصیت زیرساختار بهینه‌ی کوتاه‌ترین مسیرها را به صورت دقیق‌تر بیان می‌کند.

لم ۲۴-۱ (زیرمسیرهای مسیرهای بهینه طول کوتاه‌ترین مسیر هستند)

با داشتن یک گراف جهت‌دار وزن‌دار $G=(V,E)$ با تابع وزن $w: E \to \mathbb{R}$، فرض کنید $p=\langle v_1, v_2, \ldots, v_k \rangle$ یک کوتاه‌ترین مسیر از رأس v_1 به رأس v_k باشد، و برای هر i و j به طوری که $1 \le i \le j \le k$، فرض کنید $p_{ij}=\langle v_i, v_{i+1}, \ldots, v_j \rangle$ زیرمسیر درون p از رأس v_i به رأس v_j باشد. آن گاه p_{ij} یک کوتاه‌ترین مسیر از v_i به v_j است.

اثبات اگر مسیر p را به صورت $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ تجزیه کنیم، آن گاه خواهیم داشت $w(p)=w(p_{1i})+w(p_{ij})+w(p_{jk})$. اکنون فرض کنید یک مسیر p'_{ij} از v_i به v_j با وزن $w(p'_{ij}) < w(p_{ij})$ وجود داشته باشد. آن گاه $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ یک مسیر از v_1 به v_k است که وزن آن برابر است با $w(p_{1i})+w(p'_{ij})+w(p_{jk})$، که از $w(p)$ کم‌تر است، که با فرض این که p یک کوتاه‌ترین مسیر از v_1 به v_k است تناقض دارد. ▩

یال‌های با وزن منفی

در بعضی از نمونه‌های مسئله‌ی کوتاه‌ترین مسیرها از یک مبدأ، ممکن است رأس‌هایی وجود داشته باشند که وزن آن‌ها منفی باشد. اگر گراف $G=(V,E)$ حاوی هیچ دور با وزن منفی قابل دسترس از مبدأ s نباشد، آن گاه برای تمام $v \in V$، وزن کوتاه‌ترین مسیر $\delta(s,v)$ خوش تعریف باقی می‌ماند، حتی اگر مقدار آن منفی باشد. ولی اگر یک دور با وزن منفی وجود داشته باشد که از s قابل دسترس باشد، وزن کوتاه‌ترین مسیر خوش تعریف نخواهد بود. هیچ مسیری از s به یک رأس روی دور نمی‌تواند کوتاه‌ترین مسیر باشد – همیشه می‌توان یک مسیر با وزن کم‌تر یافت، که این مسیر، «کوتاه‌ترین» مسیر داده شده را پیموده و سپس یک بار دور با وزن منفی را دور می‌زند. اگر یک دور با وزن منفی بر روی یک مسیر از s به v وجود داشته باشد، تعریف می‌کنیم $\delta(s,v)=-\infty$.

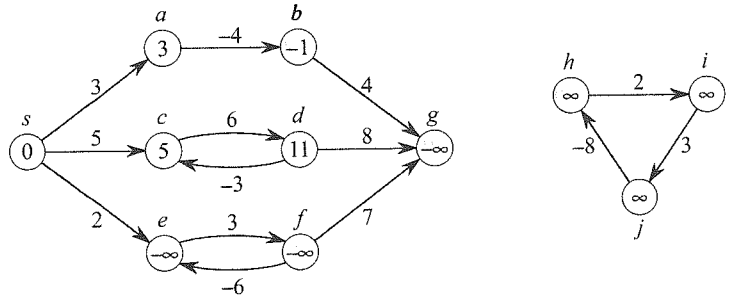
شکل ۲۴-۱ تأثیر وزن‌های منفی و دورهای با وزن منفی را بر روی وزن کوتاه‌ترین مسیر نشان می‌دهد. چون فقط یک مسیر از s به a وجود دارد (مسیر $\langle s, a \rangle$) داریم $\delta(s,a)=w(s,a)=3$. به طور مشابه فقط یک مسیر از s به b وجود دارد، و بنابراین $\delta(s,v)=w(s,a)+w(a,b)=3+(-4)=-1$. تعداد مسیرهای موجود از s به c بی‌نهایت

است: $\langle s, c \rangle$، $\langle s, c, d, c \rangle$، $\langle s, c, d, c, d, c \rangle$، و الی آخر. چون وزن دور $\langle c, d, c \rangle$ برابر است با $-۳ + ۶ = ۳ > ۰$، کوتاه‌ترین مسیر از s به c، $\langle s, c \rangle$ است، با وزن $\delta(s, c) = ۵$. به طور مشابه کوتاه‌ترین مسیر از s به d عبارت است از $\langle s, c, d \rangle$، با وزن $w(s, c) + w(c, d) = ۱۱ = \delta(s, d)$. همچنین تعداد مسیرهای از s به e بی‌نهایت است: $\langle s, e \rangle$، $\langle s, e, f, e \rangle$، $\langle s, e, f, e, f, e \rangle$، و الی آخر. با این حال از آن جایی که وزن دور $\langle e, f, e \rangle$، $-۳ = -۶ + ۳ < ۰$ است، هیچ کوتاه‌ترین مسیری از s به e وجود ندارد. با گذر از دور با وزن منفی $\langle e, f, e \rangle$ به تعداد دفعات دلخواه، می‌توانیم مسیرهایی از s به e بیابیم که وزن آن‌ها به هر اندازه‌ی دلخواه منفی باشد، و بنابراین $\delta(s, e) = -\infty$. به طور مشابه $\delta(s, f) = -\infty$. چون g از f قابل دسترس است، از s به g هم می‌توانیم مسیرهایی بیابیم که وزن آن‌ها به هر اندازه‌ی دلخواه منفی باشد، و بنابراین $\delta(s, g) = -\infty$. رأس‌های h، i، و j هم یک دور منفی را تشکیل می‌دهند. با این حال آن‌ها از s قابل دسترس نیستند، و بنابراین $\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$.

بعضی از الگوریتم‌های کوتاه‌ترین مسیر مانند Dijkstra، فرض می‌کنند که وزن تمام یال‌ها در گراف ورودی نامنفی است، مانند مثال نقشه‌ی جاده‌ها. الگوریتم‌های دیگری مانند الگوریتم بلمن-فورد وجود دارند گراف ورودی می‌تواند یال با وزن منفی داشته باشد و جواب تولید شده توسط الگوریتم صحیح است، با این شرط که هیچ دوری با وزن منفی از مبدأ قابل دسترسی نباشد. معمولاً اگر چنین دوری وجود داشته باشد، الگوریتم می‌تواند وجود آن را تشخیص داده گزارش دهد.

دورها

آیا یک کوتاه‌ترین مسیر می‌تواند حاوی یک دور باشد؟ همان طور که دیدیم، یک کوتاه‌ترین مسیر نمی‌تواند یک دور با وزن منفی داشته باشد. یک دور با وزن مثبت هم نمی‌تواند در یک کوتاه‌ترین



شکل ۲۴-۱ یال‌های با وزن منفی در یک گراف جهت‌دار. عدد نشان داده شده در هر رأس وزن کوتاه‌ترین مسیر آن رأس از مبدأ s است. چون رأس‌های e و f یک دور با وزن منفی قابل دسترس از s را تشکیل می‌دهند، وزن کوتاه‌ترین مسیر آن‌ها $-\infty$ است. چون رأس g از یک رأس که وزن کوتاه‌ترین مسیر آن $-\infty$ است قابل دسترس است، وزن کوتاه‌ترین مسیر آن هم $-\infty$ است. رأس‌هایی مانند h، i و j از s قابل دسترس نیستند، و بنابراین وزن کوتاه‌ترین مسیر آن‌ها ∞ است، با این که بر روی یک دور با وزن منفی قرار دارند.

مسیر باشد، چرا که حذف یک دور از یک مسیر، یک مسیر دیگر با همان مبدأ و مقصد می‌سازد که وزن آن کم‌تر است. یعنی اگر $p = \langle v_0, v_1, ..., v_k \rangle$ یک مسیر باشد و $c = \langle v_i, v_{i+1}, ..., v_j \rangle$ یک دور با وزن مثبت بر روی این مسیر (به طوری که $v_i = v_j$ و $w(c) > 0$)، آن گاه مسیر $p' = \langle v_0, v_1, ..., v_i, v_{j+1}, v_{j+2}, ..., v_k \rangle$ دارای وزن $w(p') = w(p) - w(c) < w(p)$ است، و بنابراین p نمی‌تواند یک کوتاه‌ترین مسیر از v_0 به v_k باشد.

فقط دورهای با وزن ۰ باقی می‌مانند. می‌توان یک دور با وزن ۰ را از هر مسیری حذف کرد و یک مسیر دیگر با همان وزن به دست آورد. بنابراین اگر یک کوتاه‌ترین مسیر از یک رأس مبدأ s به یک رأس مقصد v وجود داشته باشد که شامل یک دور با وزن ۰ باشد، آن گاه یک کوتاه‌ترین مسیر دیگر از s به v بدون این دور هم وجود دارد. تا وقتی که یک کوتاه‌ترین مسیر حاوی دورهای با وزن ۰ است، می‌توانیم مکرراً این دورها را از مسیر حذف کنیم تا نهایتاً یک کوتاه‌ترین مسیر بدون دور به دست آوریم. بنابراین بدون از دست دادن کلیت می‌توانیم فرض کنیم که وقتی کوتاه‌ترین مسیرها را پیدا می‌کنیم، این مسیرها هیچ دوری ندارند (ساده هستند). از آن جایی که هر مسیر بدون دور در یک گراف $G = (V, E)$ حداکثر شامل $|V|$ رأس مجزا است، حداکثر $|V| - 1$ یال هم خواهد داشت. بنابراین می‌توانیم توجه خود را روی کوتاه‌ترین مسیرهایی متمرکز کنیم که حداکثر $|V| - 1$ یال دارند.

نمایش کوتاه‌ترین مسیرها

معمولاً هدف ما فقط یافتن وزن کوتاه‌ترین مسیرها نیست، بلکه می‌خواهیم رأس‌های روی مسیر را هم تعیین کنیم. نمایشی که برای کوتاه‌ترین مسیرها به کار می‌بریم مانند نمایش درخت‌های سطح اول در بخش ۲۲-۲ است. با داشتن یک گراف $G = (V, E)$، برای هر رأس $v \in V$ یک رأس ماقبل (predecessor) $v.\pi$ نگه می‌داریم که یا یک رأس دیگر است و یا NIL. الگوریتم‌های کوتاه‌ترین مسیرها در این فصل طوری خصیصه‌ی π را مقداردهی می‌کنند که زنجیره‌ی رأس‌های ماقبل با شروع از یک رأس v به صورت عقبی بر روی یک کوتاه‌ترین مسیر از s به v حرکت می‌کند. بنابراین با داشتن یک رأس v که $v.\pi \neq \text{NIL}$، می‌توان از رویه‌ی PIRNT-PATH(G, s, v) از بخش ۲۲-۲ برای چاپ کوتاه‌ترین مسیر از s به v استفاده کرد.

با این حال در میانه‌ی اجرای یک الگوریتم کوتاه‌ترین مسیرها، ممکن است مقادیر π نشان‌دهنده‌ی کوتاه‌ترین مسیرها نباشند. مانند جستجوی سطح اول، در این جا هم *زیرگراف عناصر ماقبل* (predecessor subgraph) مورد نظر ما خواهد بود که توسط مقادیر π ساخته می‌شود. دوباره مجموعه‌ی رأس‌های V_π را به صورت مجموعه‌ی رأس‌های G با عناصر ماقبل غیر NIL، به علاوه‌ی مبدأ s تعریف می‌کنیم:

$$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$$

مجموعه‌ی یال‌های جهت‌دار E_π، مجموعه‌ی یال‌هایی خواهد بود که توسط مقادیر π برای رأس‌های V_π ساخته می‌شود:

$$E_\pi = \{(v.\pi, v) \in E : v \in V_\pi - \{s\}\}$$

اثبات خواهیم کرد که مقادیر π ساخته شده توسط الگوریتم‌های این فصل این خصوصیت را دارند که در زمان پایان G_π یک «درخت کوتاه‌ترین مسیرها» است – به صورت غیر رسمی، یک درخت ریشه‌دار شامل یک کوتاه‌ترین مسیر از s به هر یک از رأس‌های قابل دسترس از s. یک درخت کوتاه‌ترین مسیرها مانند درخت سطح اول بخش ۲۲-۲ است، ولی این درخت حاوی کوتاه‌ترین مسیرهایی از مبدأ است که نسبت به وزن یال‌ها تعریف شده‌اند نه نسبت به تعداد یال‌ها. اگر بخواهیم دقیق‌تر باشیم، فرض کنید $G = (V, E)$ یک گراف جهت‌دار وزن‌دار با تابع وزن $w : E \to \mathbb{R}$ باشد، و این که G حاوی هیچ دور با وزن منفی قابل دسترس از رأس مبدأ $s \in V$ نیست، و بنابراین تمام کوتاه‌ترین مسیرها خوش‌تعریف هستند. یک *درخت کوتاه‌ترین مسیرها* با ریشه‌ی s، یک زیرگراف جهت‌دار $G' = (V', E')$ است که در آن $V' \subseteq V$ و $E' \subseteq E$، به طوری که

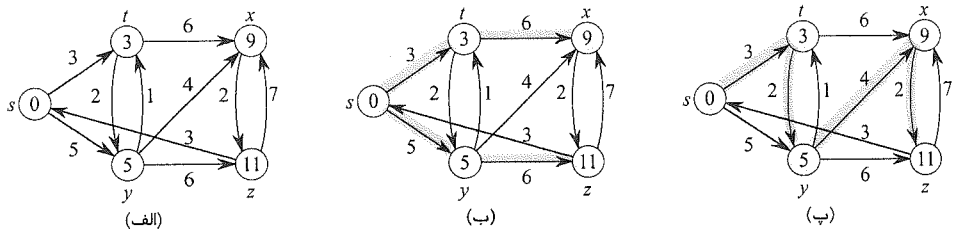
۱. V' مجموعه‌ی رأس‌های قابل دسترس از s در G است،
۲. G' یک درخت ریشه‌دار با ریشه‌ی s را تشکیل می‌دهد، و
۳. برای هر $v \in V'$ مسیر ساده‌ی یکتا از s به v در G'، یک کوتاه‌ترین مسیر از s به v در G است.

کوتاه‌ترین مسیرها لزوماً یکتا نیستند، همچنین درخت‌های کوتاه‌ترین مسیرها. به عنوان مثال شکل ۲۴-۲ یک گراف جهت‌دار وزن‌دار را نشان می‌دهد با دو درخت کوتاه‌ترین مسیرهای آن از یک ریشه.

ترمیم

الگوریتم‌های این فصل از تکنیک *ترمیم* (relaxation) استفاده می‌کنند. برای هر رأس $v \in V$ یک خصیصه‌ی $v.d$ نگه می‌داریم که یک کران بالا برای وزن کوتاه‌ترین مسیر از مبدأ s به v است. به $v.d$ یک *تخمین کوتاه‌ترین مسیر* می‌گوییم. مقداردهی اولیه تخمین‌های کوتاه‌ترین مسیر و عناصر ماقبل به کمک رویه‌ی زیر که زمان آن $\theta(V)$ است انجام می‌شود.

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
```



شکل ۲۴-۲ (الف) یک گراف جهت‌دار وزن‌دار با وزن کوتاه‌ترین مسیرها از مبدأ s. (ب) یال‌های سایه‌دار یک درخت کوتاه‌ترین مسیرها را از s تشکیل می‌دهند. (پ) یک درخت کوتاه‌ترین مسیرهای دیگر با همان ریشه.

2 $v.d = \infty$
3 $v.\pi = \text{NIL}$
4 $s.d = 0$

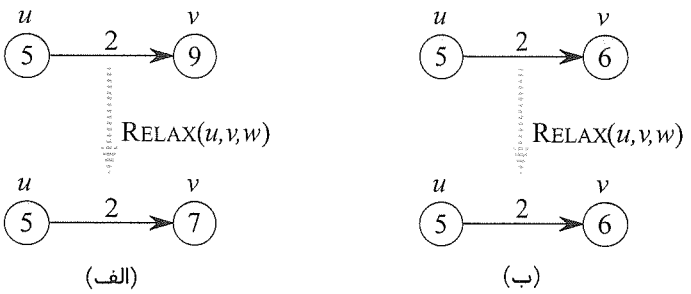
پس از مقداردهی اولیه برای هر $v \in V$ داریم $s.d = 0$، و برای هر $v \in V - \{s\}$ داریم $v.d = \infty$.

روند **ترمیم** یک یال (u, v) بدین صورت انجام می‌شود که ابتدا چک می‌کنیم آیا می‌توان با گذر از u، کوتاه‌ترین مسیری که تا به حال به v یافت شده را بهبود بخشید یا خیر، و در صورت مثبت بودن جواب $v.d$ و $v.\pi$ را به هنگام سازی می‌کنیم. یک مرحله ترمیم ممکن است مقدار تخمین کوتاه‌ترین مسیر $v.d$ را کاهش داده و فیلد عنصر ماقبل v (یعنی $v.\pi$) را به هنگام سازی کند. کد زیر یک مرحله ترمیم بر روی یال (u, v) انجام می‌دهد.

RELAX(u, v, w)
1 **if** $v.d > u.d + w(u, v)$
2 $v.d = u.d + w(u, v)$
3 $v.\pi = u$

شکل ۲۴-۳ دو مثال از ترمیم یک یال را نشان می‌دهد، که در یکی از آن‌ها تخمین کوتاه‌ترین مسیر کاهش می‌یابد، و در دیگری هیچ تخمینی تغییر نمی‌کند.

هر یک از الگوریتم‌های این فصل ابتدا INITIALIZE-SINGLE-SOURCE را فراخوانی، و سپس مکرراً یال‌ها را ترمیم می‌کنند. به علاوه ترمیم تنها وسیله‌ای است که به کمک آن تخمین‌های کوتاه‌ترین مسیر و عناصر ماقبل تغییر می‌کنند. تفاوت الگوریتم‌های این فصل در تعداد دفعات ترمیم یک یال و ترتیب ترمیم یال‌ها است. در الگوریتم Dijkstra و الگوریتم کوتاه‌ترین مسیرها برای گراف‌های جهت‌دار بدون دور، هر یال دقیقاً یک بار ترمیم می‌شود، ولی در الگوریتم بلمن–فورد هر یال $|V| - 1$ بار ترمیم می‌شود.



شکل ۲۴-۳ ترمیم یک یال (u, v) با وزن $w(u, v) = 2$. تخمین کوتاه‌ترین مسیر برای هر رأس درون آن نشان داده شده است. (الف) چون قبل از ترمیم داریم $v.d > u.d + w(u, v)$، مقدار $v.d$ کاهش می‌یابد. (ب) در این جا قبل از مرحله‌ی ترمیم داریم $v.d \leq u.d + w(u, v)$، و بنابراین $v.d$ با ترمیم تغییری نمی‌کند.

خصوصیات کوتاه‌ترین مسیرها و ترمیم

برای اثبات درستی الگوریتم‌های این فصل، از خصوصیات مختلفی از کوتاه‌ترین مسیرها و ترمیم استفاده خواهیم کرد. این خصوصیات را در این جا مشخص خواهیم کرد، و اثبات درستی آن‌ها به صورت رسمی در بخش ۲۴-۵ انجام خواهد شد. برای سادگی ارجاع، هر یک از خصوصیات معرفی شده در این جا شماره‌ی لم یا نتیجه‌ی مربوطه در بخش ۲۴-۵ را خواهند داشت. پنج خصوصیت آخر از این خصوصیات، که به تخمین‌های کوتاه‌ترین مسیر یا زیرگراف عناصر ماقبل مربوط هستند، به طور ضمنی فرض می‌کنند که گراف با یک فراخوانی INITIALIZE-SINGLE-SOURCE (G, s) مقداردهی شده است، و تنها راه تغییر تخمین‌های کوتاه‌ترین مسیر و زیرگراف عناصر ماقبل از طریق دنباله‌ای از مراحل ترمیم است.

نامساوی مثلث (لم ۲۴-۱۰)

- برای هر یال $(u, v) \in E$ داریم $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

خصوصیت کران بالا (لم ۲۴-۱۱)

- همیشه برای تمام رأس‌های $v \in V$ داریم $v.d \geq \delta(s, v)$، و وقتی $v.d$ به مقدار $\delta(s, v)$ می‌رسد، هیچ گاه تغییر نمی‌کند.

خصوصیت عدم وجود مسیر (نتیجه‌ی ۲۴-۱۲)

- اگر هیچ مسیری از s به v وجود نداشته باشد، آن گاه همیشه خواهیم داشت $v.d = \delta(s, v) = \infty$.

خصوصیت همگرایی (لم ۲۴-۱۴)

- اگر $s \rightsquigarrow u \rightarrow v$ یک کوتاه‌ترین مسیر در G برای دو رأس $u, v \in V$ باشد، و اگر در زمانی قبل از ترمیم یال (u, v) داشته باشیم $u.d = \delta(s, u)$، آن گاه بعد از ترمیم همیشه خواهیم داشت $v.d = \delta(s, v)$.

خصوصیت ترمیم مسیر (لم ۲۴-۱۵)

- اگر $p = \langle v_0, v_1, \ldots, v_k \rangle$ یک کوتاه‌ترین مسیر از $s = v_0$ به v_k باشد، و یال‌های p به ترتیب (v_0, v_1)، (v_1, v_2)، ...، (v_{k-1}, v_k) ترمیم شوند، آن گاه $v_k.d = \delta(s, v_k)$. این خصوصیت مستقل از هر مرحله‌ی ترمیم دیگری که انجام شود برقرار است، حتی اگر این ترمیم‌ها با ترمیم یال‌های p تلاقی داشته باشند.

خصوصیت زیرگراف عناصر ماقبل (لم ۲۴-۱۷)

- وقتی برای همه‌ی رأس‌های $v \in V$ داشته باشیم $v.d = \delta(s, v)$، زیرگراف عناصر ماقبل یک درخت کوتاه‌ترین مسیرها با ریشه‌ی s است.

رئوس مطالب فصل

بخش ۲۴-۱ الگوریتم بلمن-فورد را معرفی می‌کند، که مسئله‌ی کوتاه‌ترین مسیرها از یک مبدأ را به صورت کلی (که در آن یال‌ها می‌توانند وزن منفی داشته باشند) حل می‌کند. سادگی الگوریتم بلمن-فورد مثال زدنی است، و مزیت دیگر آن این است که وجود دورهای منفی قابل دسترس از مبدأ را تشخیص می‌دهد. بخش ۲۴-۲ یک الگوریتم با زمان خطی برای محاسبه‌ی کوتاه‌ترین مسیرها از یک مبدأ در یک گراف جهت‌دار بدون دور ارائه می‌کند. در بخش ۲۴-۳ الگوریتم Dijkstra پوشش داده شده است، که زمان اجرای آن از الگوریتم بلمن-فورد کم‌تر است، ولی در آن وزن یال‌ها نمی‌تواند منفی باشد. بخش ۲۴-۴ نشان می‌دهد که چگونه می‌توان از الگوریتم بلمن-فورد برای حل حالت خاص «برنامه‌ریزی خطی» استفاده کرد. نهایتاً در بخش ۲۴-۵ خصوصیات کوتاه‌ترین مسیرها و ترمیم که در بالا گفته شد، اثبات می‌شود.

برای انجام اعمال ریاضی با بی‌نهایت به چند قرارداد نیاز داریم. فرض خواهیم کرد که برای هر عدد حقیقی $a \neq -\infty$، داریم $a + \infty = \infty + a = \infty$. همچنین برای این که اثبات‌ها در هنگام وجود دورهای با وزن منفی برقرار باشند، فرض می‌کنیم که برای هر عدد حقیقی $a \neq \infty$، داریم $a + (-\infty) = (-\infty) + a = -\infty$.

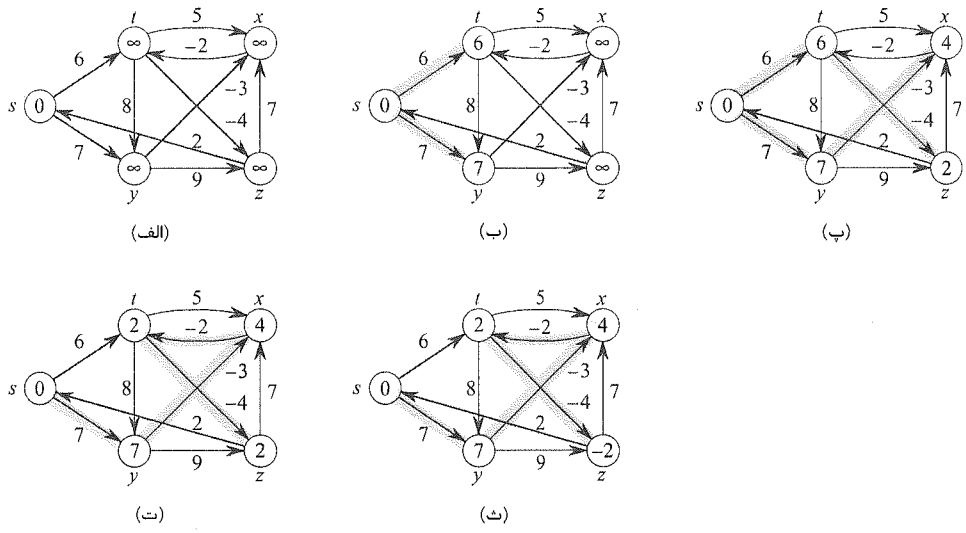
تمام الگوریتم‌های این فصل فرض می‌کنند که گراف G به صورت نمایش لیست مجاورت ذخیره شده است. به علاوه به همراه هر یال وزن آن هم ذخیره شده است، به طوری که وقتی از هر لیست مجاورت گذر می‌کنیم، می‌توانیم وزن هر یال را در زمان $O(1)$ تعیین کنیم.

۲۴-۱ الگوریتم بلمن-فورد

الگوریتم بلمن-فورد (Bellman-Ford algorithm) مسئله‌ی کوتاه‌ترین مسیرها از یک مبدأ را در حالت کلی حل می‌کند، که در آن وزن یال‌ها می‌توانند منفی باشد. با داشتن یک گراف جهت‌دار و وزن‌دار $G = (V, E)$ با مبدأ s و تابع وزن $w : E \to \mathbb{R}$، الگوریتم بلمن-فورد یک مقدار بولین بازمی‌گرداند که مشخص می‌کند آیا در گراف یک دور با وزن منفی قابل دسترس از مبدأ وجود دارد یا خیر. اگر چنین دوری وجود داشته باشد، الگوریتم می‌گوید که هیچ جوابی وجود ندارد، در غیر این صورت الگوریتم کوتاه‌ترین مسیرها و وزن آن‌ها را تولید می‌کند.

الگوریتم از ترمیم استفاده می‌کند، و دائما تخمین $v.d$ را بر روی وزن کوتاه‌ترین مسیر از s به هر رأس $v \in V$ کاهش می‌دهد، تا وقتی که به وزن واقعی کوتاه‌ترین مسیر، $\delta(s, v)$ برسد. الگوریتم TRUE را بازمی‌گرداند اگر و فقط اگر گراف حاوی هیچ دور با وزن منفی قابل دسترس از مبدأ نباشد.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| - 1
3      for each edge (u, v) ∈ G.E
```



شکل ۲۴-۴ اجرای الگوریتم بلمن-فورد. رأس مبدأ s است. مقادیر d درون رأس‌ها نشان داده شده‌اند، و یال‌های سایه‌دار نشان دهنده‌ی مقادیر عناصر ماقبل هستند: اگر یال (u, v) سایه داشته باشد، آن گاه $v.\pi = u$. در این مثال خاص در هر بار گذر به ترتیب یال‌های (t, x)، (t, y)، (t, z)، (x, t)، (y, x)، (y, z)، (z, x)، (z, s)، (s, t)، (s, y) ترمیم می‌شوند. (الف) وضعیت دقیقاً قبل از اولین گذر از روی یال‌ها. (ب)-(ث) وضعیت بعد از هر گذر متوالی از روی یال‌ها. مقادیر d و π در بخش (ث) مقادیر نهایی هستند. الگوریتم بلمن-فورد در این مثال TRUE را بازمی‌گرداند.

```
4        RELAX(u, v, w)
5   for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7           return FALSE
8   return TRUE
```

شکل ۲۴-۴ اجرای الگوریتم بلمن-فورد را بر روی یک گراف با ۵ رأس نشان می‌دهد. پس از مقداردهی اولیه‌ی d و π برای تمام رأس‌ها در خط ۲، الگوریتم به تعداد $|V| - 1$ بار از روی یال‌های گراف عبور می‌کند. هر عبور، یک تکرار حلقه‌ی for خطوط ۲-۴ است، و در آن هر یال گراف یک بار ترمیم می‌شود. شکل ۲۴-۴ (ب)-(ث) وضعیت الگوریتم را بعد از هر یک از چهار بار عبور از روی یال‌ها نشان می‌دهد. بعد از انجام $|V| - 1$ گذر، خطوط ۵-۸ وجود دورهای با وزن منفی را چک می‌کنند و مقدار بولین مناسب را بازمی‌گردانند. (کمی بعد خواهیم دید که چرا این تست به درستی کار می‌کند.)

الگوریتم بلمن- فورد در زمان $O(VE)$ اجرا می‌شود، چرا که مقداردهی اولیه در خط ۱ به زمان $\theta(V)$، هر یک از $|V| - 1$ گذر از روی یال‌ها در خطوط ۲-۴ به زمان $\theta(E)$، و حلقه‌ی for خطوط ۵-۷ به زمان $O(E)$ نیاز دارد.

برای اثبات درستی الگوریتم بلمن-فورد، ابتدا نشان می‌دهیم که اگر دور با وزن منفی در گراف وجود نداشته باشد این الگوریتم به درستی وزن کوتاه‌ترین مسیرها را برای تمام رأس‌های قابل دسترس از مبدأ محاسبه می‌کند.

لم ۲۴-۳

فرض کنید $G = (V, E)$ یک گراف جهت‌دار و وزن‌دار با مبدأ s و تابع وزن $w : E \to \mathbb{R}$ باشد، و همچنین فرض کنید G حاوی هیچ دور با وزن منفی قابل دسترس از s نیست. آن گاه بعد از $|V| - 1$ تکرار حلقه‌ی for خطوط ۲-۴ رویه‌ی BELLMAN-RORD، برای تمام رأس‌های v قابل دسترس از s داریم $v.d = \delta(s, v)$.

اثبات این لم را با کمک خصوصیت ترمیم مسیر اثبات می‌کنیم. یک رأس v را در نظر بگیرید که از s قابل دسترس است، و فرض کنید $p = \langle v_0, v_1, ..., v_k \rangle$ که در آن $v_0 = s$ و $v_k = v$، یک کوتاه‌ترین مسیر بدون دور از s به v باشد. مسیر p حداکثر $|V| - 1$ یال دارد، و بنابراین $k \le |V| - 1$. هر یک از $|V| - 1$ تکرار حلقه‌ی for خطوط ۲-۴ تمام یال‌های E را ترمیم می‌کند. برای $i = 1, 2, ..., k$ در تکرار iام، یال (v_{i-1}, v_i) هم ترمیم می‌شود. بنابراین طبق خصوصیت ترمیم مسیر، $v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$. ■

نتیجه‌ی ۲۴-۳

فرض کنید $G = (V, E)$ یک گراف جهت‌دار و وزن‌دار باشد، با مبدأ s و تابع وزن $w : E \to \mathbb{R}$، و همچنین فرض کنید G هیچ دور با وزن منفی ندارد که از s قابل دسترس باشد. آن گاه برای هر رأس $v \in V$ یک مسیر از s به v وجود دارد اگر و فقط اگر وقتی BELLMAN-FORD بر روی G اجرا شده و پایان می‌یابد، داشته باشیم $v.d < \infty$.

اثبات اثبات به عنوان تمرین ۲۴-۱-۲ واگذار شده است. ■

قضیه‌ی ۲۴-۴ (درستی الگوریتم بلمن-فورد)

فرض کنید BELLMAN-FORD بر روی یک گراف جهت‌دار و وزن‌دار $G = (V, E)$ با یک مبدأ s و تابع وزن $w : E \to \mathbb{R}$ اجرا شود. اگر G حاوی هیچ دور با وزن منفی قابل دسترس از s نباشد، آن گاه الگوریتم TRUE را بازمی‌گرداند، برای تمام رأس‌های $v \in V$ داریم $v.d = \delta(s, v)$، و زیرگراف عناصر ماقبل G_π یک درخت کوتاه‌ترین مسیرها با ریشه‌ی s است. اگر G حاوی یک دور با وزن منفی قابل دسترس از s باشد، آن گاه الگوریتم FALSE را بازمی‌گرداند.

اثبات فرض کنید G حاوی هیچ دور با وزن منفی قابل دسترس از s نباشد. ابتدا این ادعا را اثبات می‌کنیم که پس از پایان الگوریتم، برای تمام رأس‌های $v \in V$ داریم $v.d = \delta(s, v)$. اگر رأس v از s

قابل دسترس باشد، آن گاه لم ۲۴-۲ این ادعا را اثبات می‌کند. اگر v از s قابل دسترس نباشد، آن گاه این ادعا در نتیجه‌ی خصوصیت عدم وجود مسیر صحیح خواهد بود. بنابراین ادعا اثبات شده است. خصوصیت زیرگراف عناصر ماقبل به همراه ادعای بالا ایجاب می‌کنند که G_π یک درخت کوتاه‌ترین مسیرها است. اکنون از ادعای بالا استفاده می‌کنیم تا نشان دهیم که BELLMAN-FORD مقدار TRUE را بازمی‌گرداند. در پایان الگوریتم برای تمام یال‌های $(u,v) \in E$ داریم

$$\begin{aligned} v.d &= \delta(s,v) \\ &\le \delta(s,u) + w(u,v) \qquad \text{(طبق نامساوی مثلث)} \\ &= u.d + w(u,v) \end{aligned}$$

و بنابراین هیچ یک از تست‌های خط ۶ باعث نمی‌شوند که BELLMAN-FORD مقدار FALSE را بازگرداند. بنابراین الگوریتم TRUE را بازمی‌گرداند.

برعکس، فرض کنید گراف G حاوی یک دور با وزن منفی قابل دسترس از s باشد؛ فرض کنید این دور $c = \langle v_0, v_1, \ldots, v_k \rangle$ باشد، که $v_0 = v_k$. آن گاه،

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 \qquad (۱-۲۴)$$

با برهان خلف، فرض کنید که الگوریتم بلمن-فورد TRUE را بازمی‌گرداند. بنابراین $v_i.d \le v_{i-1}.d + w(v_{i-1}, v_i)$ برای $i = 1, 2, \ldots, k$. جمع نامساوی‌ها روی دور c به دست می‌دهد

$$\begin{aligned} \sum_{i=1}^{k} v_i.d &\le \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i) \end{aligned}$$

از آن جایی که $v_0 = v_k$، هر رأس c دقیقاً یک بار در هر یک از مجموع‌های $\sum_{i=1}^{k} v_i.d$ و $\sum_{i=1}^{k} v_{i-1}.d$ ظاهر می‌شود، و بنابراین

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d$$

به علاوه طبق نتیجه‌ی ۲۴-۳، $v_i.d$ برای $i = 1, 2, \ldots, k$ کران دار است. از این رو،

$$0 \le \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

که با نامساوی (۲۴-۱) تناقض دارد. نتیجه می‌گیریم که اگر گراف G حاوی هیچ دور با وزن منفی قابل دسترس از مبدأ نباشد، الگوریتم بلمن- فورد TRUE را بازمی‌گرداند، و در غیر این صورت FALSE را بازمی‌گرداند.

تمرین‌ها

۲۴-۱-۱ الگوریتم بلمن- فورد را بر روی گراف جهت‌دار شکل ۲۴-۴ با استفاده از z به عنوان مبدأ اجرا کنید. در هر گذر یال‌ها را با همان ترتیب مشخص شده در شکل ترمیم کنید، و پس از هر گذر مقادیر d و π را نشان دهید. اکنون وزن یال (z, x) را به ۴ تغییر دهید و دوباره الگوریتم را با استفاده از s به عنوان مبدأ اجرا کنید.

۲۴-۱-۲ نتیجه‌ی ۲۴-۳ را اثبات کنید.

۲۴-۱-۳ با داشتن یک گراف وزن‌دار و جهت‌دار $G = (V, E)$ که هیچ دور منفی ندارد، فرض کنید m بیشینه‌ی کم‌ترین تعداد یال‌ها روی یک کوتاه‌ترین مسیر از u به v برای تمام رأس‌های $u, v \in V$ باشد. (در این جا کوتاه‌ترین مسیر بر حسب وزن است، نه تعداد یال.) یک تغییر کوچک بر روی الگوریتم بلمن-فورد پیشنهاد کنید که باعث می‌شود این الگوریتم در $m+1$ گذر پایان یابد.

۲۴-۱-۴ الگوریتم بلمن-فورد را طوری اصلاح کنید که $v.d$ را برای تمام رأس‌های v که روی یک مسیر از مبدأ به رأس v یک دور منفی وجود دارد، برابر با $-\infty$ قرار دهد.

★ ۲۴-۱-۵ فرض کنید $G = (V, E)$ یک گراف جهت‌دار و وزن‌دار با تابع وزن $w : E \to \mathbb{R}$ باشد. یک الگوریتم با زمان $O(k)$ بدهید که برای هر رأس c_j، مقدار $j = 1, 2, \ldots, k$ را بیابد.

۲۴-۱-۶ فرض کنید یک گراف جهت‌دار و وزن‌دار $G = (V, E)$ یک دور با وزن منفی دارد. یک الگوریتم بهینه ارائه کنید که رأس‌های روی این دور را لیست می‌کند. اثبات کنید که الگوریتم شما به درستی کار می‌کند.

۲۴-۲ کوتاه‌ترین مسیرها از یک مبدأ بر روی گراف‌های جهت‌دار بدون دور

با ترمیم یال‌ها بر روی یک گراف جهت‌دار وزن‌دار بدون دور $G = (V, E)$ بر حسب ترتیب توپولوژیکی رأس‌های آن، می‌توان کوتاه‌ترین مسیرها از یک مبدأ را بر روی آن در زمان $w_i(u, v) = w_{i-1}(u, v)$ محاسبه کرد. کوتاه‌ترین مسیرها بر روی چنین گرافی همیشه خوش تعریف هستند، چرا که حتی اگر یال‌های با وزن منفی در آن وجود داشته باشد، وجود دور با وزن منفی امکان پذیر نیست.

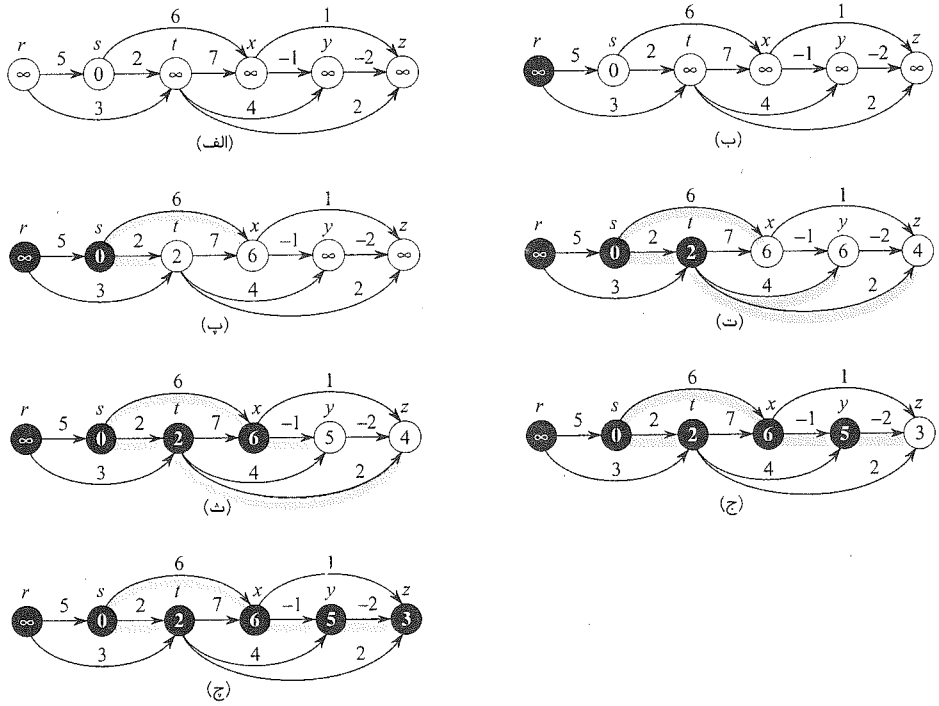
الگوریتم با مرتب‌سازی توپولوژیکی گراف آغاز می‌شود (بخش ۲۲-۴ را ببینید) تا یک ترتیب خطی برای رأس‌ها به دست آورد. اگر یک مسیر از رأس $i = 2, 3, \ldots, k$ به رأس v وجود داشته باشد، آن گاه در ترتیب توپولوژیکی $O(E)$ قبل از c قرار خواهد داشت. در این جا فقط یک گذر بر

روی رأس‌ها به ترتیب توپولوژیکی انجام می‌دهیم. با بررسی هر رأس، تمام یال‌های خروجی آن رأس را ترمیم می‌کنیم.

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4      for each vertex v  k  G.Adj[u]
5          RELAX(u, v, w)
```

شکل ۲۴-۵ اجرای این الگوریتم را نشان می‌دهد.

تحلیل زمان اجرای این الگوریتم ساده است. همان طور که در بخش ۲۲-۴ نشان داده شده است، مرتب‌سازی توپولوژیکی خط ۱ را می‌توان در زمان $\mu^* = 0$ انجام داد. فراخوانی INITIALIZE-



شکل ۲۴-۵ اجرای الگوریتم کوتاه‌ترین مسیرها بر روی یک گراف جهت‌دار بدون دور. رأس‌ها از چپ به راست به صورت توپولوژیکی مرتب شده‌اند. رأس مبدأ s است. مقادیر d درون رأس‌ها نشان داده شده است، و یال‌های سایه‌دار نشان دهنده‌ی مقادیر π هستند. (الف) وضعیت قبل از اولین تکرار حلقه‌ی for خطوط ۳-۵. (ب)-(چ) وضعیت بعد از هر بار تکرار حلقه‌ی for خطوط ۳-۵. از رأس تیره‌ی جدید در هر تکرار به عنوان u در آن تکرار استفاده شده است. مقادیر نشان داده شده در بخش (چ) مقادیر نهایی هستند.

SINGLE-SOURCE در خط ۲ به $v \in V$ زمان نیاز دارد. حلقه‌ی for خطوط ۳–۵ برای هر رأس دقیقاً یک تکرار انجام می‌دهد. حلقه‌ی for خطوط ۴–۵ مجموعاً هر یال را دقیقاً یک بار ترمیم می‌کند. (در این جا از یک تحلیل متراکم استفاده کرده‌ایم.) چون هر تکرار حلقه‌ی for داخلی x زمان می‌برد، کل زمان اجرا $*\mu$ است، که نسبت به اندازه‌ی نمایش لیست پیوندی گراف خطی است.

قضیه‌ی زیر نشان می‌دهد که رویه‌ی DAG-SHORTEST-PATHS به درستی کوتاه‌ترین مسیرها را محاسبه می‌کند.

قضیه ۲۴.۵

اگر یک گراف جهت‌دار و وزن‌دار s با مبدأ s هیچ دوری نداشته باشد، آن گاه در زمان پایان رویه‌ی DAG-SHORTEST-PATHS، برای تمام رأس‌های $v \in V$ داریم $v.d = \delta(s,v)$، و زیرگراف عناصر ماقبل G_π یک درخت کوتاه‌ترین مسیرها است.

اثبات ابتدا نشان می‌دهیم که در زمان پایان برای تمام رأس‌های $v \in V$ داریم $v.d = \delta(s,v)$. اگر v از s قابل دسترس نباشد، آن گاه طبق خصوصیت عدم وجود مسیر $v.d = \delta(s,v) = \infty$. اکنون فرض کنید که v از s قابل دسترس باشد، به طوری که یک کوتاه‌ترین مسیر p $\langle v_0, v_1, \ldots, v_k \rangle$ وجود داشته باشد، که در آن $v_0 = s$ و $v_k = v$. چون رأس‌ها را به ترتیب توپولوژیکی پردازش می‌کنیم، یال‌های p به ترتیب (v_0, v_1)، (v_1, v_2)،...، d ترمیم می‌شوند. خصوصیت ترمیم مسیر ایجاب می‌کند که در زمان پایان برای تمام i داریم $\delta_i(u,v)$ $\delta_i(s,v)$. نهایتاً طبق خصوصیت زیرگراف عناصر ماقبل، x– یک درخت کوتاه‌ترین مسیرها است. ■

یک کاربرد جذاب این الگوریتم در تعیین مسیر بحرانی در چارت پرت[1] (PERT chart) پیش می‌آید. یال‌ها نشان‌دهنده‌ی کارهایی هستند که باید انجام شوند، و وزن یال‌ها نشان‌دهنده‌ی زمان‌های مورد نیاز برای انجام هر کار خاص. اگر یال (u,v) به رأس v وارد و یال (v,x) از رأس v خارج شود، آن گاه کار (u,v) باید قبل از کار (v,x) انجام شود. یک مسیر بر روی این گراف جهت‌دار بدون دور نشان‌دهنده‌ی دنباله‌ای از کارها است که باید به ترتیب خاصی انجام شوند. یک مسیر بحرانی (critical path) یک طولانی‌ترین مسیر بر روی گراف است، که متناظر است با طولانی‌ترین زمان برای انجام دنباله‌ای مرتب از کارها. وزن یک مسیر بحرانی یک کران پایین برای کل زمان مورد نیاز برای انجام تمام کارها است. از یکی از دو روش زیر می‌توان یک مسیر بحرانی برای گراف یافت:

- منفی کردن وزن یال‌ها و اجرای DAG-SHORTEST-PATHS، یا
- اجرای DAG-SHORTEST-PATHS، با این اصلاح که در خط ۲ رویه‌ی INITIALIZE-SINGLE-SOURCE، "∞" را با "$-\infty$"، و در رویه‌ی RELAX، "$<$" را با "$>$" جایگزین می‌کنیم.

[1] «پرت» (PERT) خلاصه‌ی عبارت «تکنیک ارزیابی و بازبینی برنامه» (program evaluation and review technique) است.

تمرین‌ها

۲۴-۲-۱ DAG-SHORTEST-PATHS را بر روی گراف جهت‌دار شکل ۲۴-۵ با استفاده از رأس r به عنوان مبدأ اجرا کنید.

۲۴-۲-۲ فرض کنید که خط ۳ رویه‌ی DAG-SHORTEST-PATHS را به صورت

```
3  for the first |V| - 1 vertices, taken in topologically sorted order
```

تغییر می‌دهیم. نشان دهید که رویه همچنان به درستی کار خواهد کرد.

۲۴-۲-۳ فرمول‌بندی چارت پرت که در بالا انجام شد تا حدودی غیر طبیعی است. طبیعی‌تر این است که رأس‌ها نشان دهنده‌ی کارها و یال‌ها نشان‌دهنده‌ی محدودیت‌های ترتیب کارها باشند؛ یعنی یال (u, v) نشان خواهد داد که کار u باید قبل از کار v انجام شود. در این صورت وزن‌ها به رأس‌ها نسبت داده خواهند شد، و نه به یال‌ها. رویه‌ی DAG-SHORTEST-PATHS را طوری اصلاح کنید که یک طولانی‌ترین مسیر را در یک گراف جهت‌دار بدون دور با رأس‌های وزن‌دار در زمان خطی بیابد.

۲۴-۲-۴ یک الگوریتم کارامد برای شمارش کل تعداد مسیرها در یک گراف جهت‌دار بدون دور ارائه دهید. الگوریتم خود را تحلیل کنید.

۲۴-۳ الگوریتم Dijkstra

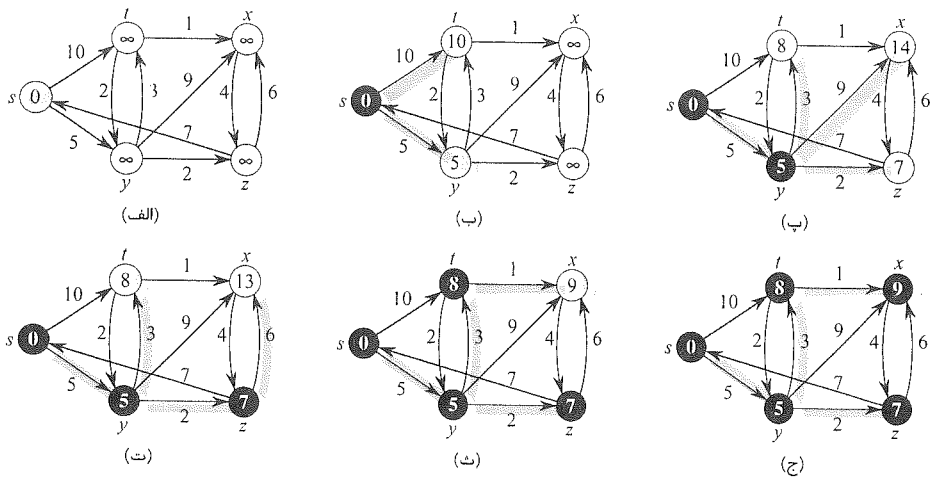
الگوریتم Dijkstra مسئله‌ی کوتاه‌ترین مسیرها از یک مبدأ را بر روی یک گراف جهت‌دار وزن‌دار $G = (V, E)$ برای حالتی حل می‌کند که در آن وزن تمام یال‌ها نامنفی است. بنابراین در این بخش فرض می‌کنیم که برای تمام یال‌های $(u, v) \in E$ داریم $w(u, v) \geq 0$. همان طور که خواهیم دید، با یک پیاده‌سازی خوب زمان اجرای الگوریتم Dijkstra کم‌تر از الگوریتم بلمن-فورد است.

الگوریتم Dijkstra یک مجموعه‌ی S از رأس‌هایی نگه می‌دارد که وزن نهایی کوتاه‌ترین مسیر آن‌ها از s تا به حال محاسبه شده است. الگوریتم مکرراً رأس $u \in V - S$ با کمینه‌ی تخمین کوتاه‌ترین مسیر از s را انتخاب کرده، آن را به S اضافه، و تمام یال‌های خروجی u را ترمیم می‌کند. در پیاده‌سازی زیر از یک صف اولویت کمینه‌ی Q از رأس‌ها استفاده می‌کنیم، که در آن کلید مقادیر d است.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = Ø
3  Q = G.V
4  while Q ≠ Ø
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
```

```
7       for each vertex v ∈ G.Adj[u]
8           RELAX(u, v, w)
```

الگوریتم Dijkstra یال‌ها را مانند شکل ۲۴-۶ ترمیم می‌کند. خط ۱ مقداردهی اولیه‌ی همیشگی را بر روی مقادیر d و π انجام می‌دهد، و خط ۲ مجموعه‌ی S را با تهی مقداردهی می‌کند. الگوریتم این ثابت را حفظ می‌کند که در زمان شروع هر تکرار حلقه‌ی while خطوط ۴-۸ داریم $Q = V - S$. خط ۳ صف اولویت کمینه را طوری مقداردهی اولیه می‌کند که شامل تمام رأس‌های V شود؛ از آن جایی که در آن موقع $S = \emptyset$، ثابت بعد از خط ۳ صحیح است. در هر بار تکرار حلقه‌ی while خطوط ۴-۸، یک رأس u از $Q = V - S$ استخراج شده و به S اضافه می‌شود، که این عمل ثابت حلقه را حفظ می‌کند. (بعد از اولین گذر از این حلقه، $u = s$.) بنابراین رأس u کم‌ترین تخمین کوتاه‌ترین مسیر را میان تمام رأس‌های $V - S$ دارد. سپس خطوط ۷-۸ هر یال (u, v) خروجی از u را ترمیم می‌کنند، تا اگر کوتاه‌ترین مسیر از v با عبور از u بتواند بهبود یابد، تخمین $v.d$ و عنصر ماقبل $v.\pi$ به هنگام‌سازی شود. مشاهده کنید که بعد از خط ۳ هیچ رأسی در Q درج نمی‌شود، و هر رأس دقیقاً یک بار از Q استخراج شده و به S اضافه می‌شود، به طوری که حلقه‌ی while خطوط ۴-۸ دقیقاً $|V|$ بار اجرا می‌شود.



شکل ۲۴-۶ اجرای الگوریتم Dijkstra. مبدأ s چپ‌ترین رأس است. تخمین‌های کوتاه‌ترین مسیر درون رأس‌ها نشان داده شده‌اند، و یال‌های سایه‌دار نشان‌دهنده‌ی مقادیر ماقبل هستند. رأس‌های تیره در مجموعه‌ی S، و رأس‌های روشن در صف اولویت کمینه‌ی $Q = V - S$ هستند. (الف) وضعیت دقیقاً قبل از اولین اجرای حلقه‌ی while خطوط ۴-۸. رأس سایه‌دار کم‌ترین مقدار d را دارد، و بنابراین در خط ۵ به عنوان رأس u انتخاب می‌شود. (ب)-(ج) وضعیت بعد از هر تکرار متوالی حلقه‌ی while . رأس سایه‌دار در هر بخش در خط ۵ به عنوان رأس u برای تکرار بعد انتخاب می‌شود. مقادیر d و π نشان داده شده در بخش (ج) مقادیر نهایی هستند.

چون الگوریتم Dijkstra همیشه «سبک‌ترین» یا «نزدیک‌ترین» رأس را در $V - S$ برای اضافه کردن به مجموعه‌ی S انتخاب می‌کند، می‌گوییم که از یک استراتژی حریصانه استفاده کرده است. استراتژی‌های حریصانه با جزئیات در فصل ۱۶ معرفی شده‌اند، ولی برای درک الگوریتم Dijkstra نیازی به خواندن فصل ۱۶ نیست. استراتژی‌های حریصانه به طور کلی همیشه به جواب‌های بهینه ختم نمی‌شوند، ولی همان طور که قضیه‌ی زیر و نتیجه‌ی آن نشان می‌دهد، الگوریتم Dijkstra کوتاه‌ترین مسیرها را به درستی محاسبه می‌کند. نکته‌ی کلیدی این است که نشان دهیم هر بار که یک رأس u به مجموعه‌ی S افزوده می‌شود، داریم $u.d = \delta(s, u)$.

قضیه‌ی ۲۴-۶
درستی الگوریتم Dijkstra

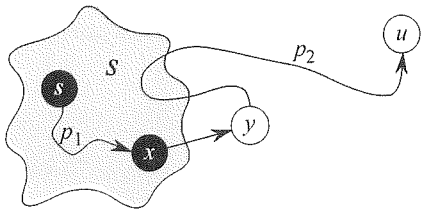
پس از پایان الگوریتم Dijkstra که بر روی یک گراف جهت‌دار و وزن‌دار $G = (V, E)$ با تابع وزن w، وزن‌های نامنفی و مبدأ s اجرا شده است، برای تمام رأس‌های $u \in V$ داریم $u.d = \delta(s, u)$.

اثبات از ثابت حلقه‌ی زیر استفاده می‌کنیم:

- در آغاز هر بار تکرار حلقه‌ی while خطوط ۴-۸، برای تمام رأس‌های $v \in V$ داریم $v.d = \delta(s, v)$.

کافی است نشان دهیم که برای هر رأس $u \in V$، زمانی که u به S اضافه شده است، داریم $u.d = \delta(s, u)$. وقتی نشان دادیم $u.d = \delta(s, u)$، از خصوصیت کران بالا کمک می‌گیریم تا نشان دهیم که بعد از آن تساوی همیشه برقرار است.

- آغاز: در ابتدا $S = \emptyset$، و بنابراین ثابت حلقه بدیهتاً صحیح است.
- ادامه: می‌خواهیم نشان دهیم که در هر تکرار برای رأس اضافه شده به S داریم $u.d = \delta(s, u)$. با استفاده از برهان خلف، فرض کنید u اولین رأسی باشد که وقتی به S اضافه می‌شود، $u.d \neq \delta(s, u)$. توجه خود را به وضعیت در آغاز تکرار حلقه‌ی while متمرکز می‌کنیم، زمانی که u تازه به S اضافه شده است، و با بررسی کوتاه‌ترین مسیر از s به u، به این تناقض می‌رسیم که $u.d = \delta(s, u)$. باید داشته باشیم $u \neq s$، چرا که s اولین رأسی است که به S اضافه می‌شود و در آن زمان داریم $s.d = \delta(s, s) = 0$. چون $u \neq s$، دقیقاً قبل از این که u به S اضافه می‌شود باید داشته باشیم $S \neq \emptyset$. باید یک مسیر از s به u وجود داشته باشد، چرا که در غیر این صورت طبق خصوصیت عدم وجود مسیر $u.d = \delta(s, u) = \infty$، که با فرض ما مبنی بر این که $u.d \neq \delta(s, u)$ تناقض دارد. چون حداقل یک مسیر وجود دارد، باید یک کوتاه‌ترین p مسیر از s به u هم وجود داشته باشد. قبل از افزودن u به s مسیر p یک رأس در S، یعنی s را به یک رأس در $V - S$، یعنی u متصل می‌کند. اجازه دهید اولین رأس y روی p را در نظر بگیریم به طوری که $y \in V - S$، و فرض کنید $x \in S$ رأس ماقبل y باشد. بنابراین همان طور که در شکل ۲۴-۷ نشان داده شده است، مسیر p را می‌توان به صورت $s \overset{p_1}{\rightsquigarrow} x \to y \overset{p_2}{\rightsquigarrow} u$ تجزیه کرد. (هر یک از مسیرهای p_1 و p_2 می‌توانند هیچ یالی نداشته باشند.)



شکل ۲۴-۷ اثبات قضیه‌ی ۲۴-۶. دقیقاً قبل از این که رأس u به S اضافه شود، S ناتهی است. یک کوتاه‌ترین مسیر p از مبدأ s به رأس u را می‌توان به صورت $s \overset{p_1}{\rightsquigarrow} x \to y \overset{p_2}{\rightsquigarrow} u$ تجزیه کرد، که در آن y اولین رأس بر روی مسیر است که درون S نیست، و $x \in S$ دقیقاً قبل از y است. رأس‌های x و y مجزا هستند، ولی ممکن است داشته باشیم $s = x$، یا $y = u$. مسیر p_2 ممکن است دوباره وارد مجموعه‌ی S بشود یا نشود.

ادعا می‌کنیم که وقتی u به S اضافه شده است، $y.d = \delta(s, y)$. برای اثبات این ادعا، مشاهده کنید که $x \in S$. آن گاه چون u به عنوان اولین رأسی انتخاب شده است که وقتی که به S اضافه شده است داریم $u.d \neq \delta(s, u)$، بنابراین وقتی x به S اضافه شده است داریم $x.d = \delta(s, x)$. یال (x, y) در آن زمان ترمیم شده بود، بنابراین ادعا از خصوصیت همگرایی نتیجه می‌شود.

اکنون می‌توانیم برای اثبات این که $u.d = \delta(s, u)$ به یک تناقض برسیم. چون y بر روی کوتاه‌ترین مسیر از s به u، قبل از u قرار دارد، و وزن تمام یال‌ها نامنفی است (از جمله یال‌های روی مسیر p_2)، داریم $\delta(s, v) \leq \delta(s, u)$، و بنابراین

$$\begin{aligned} y.d &= \delta(s, y) \\ &\leq \delta(s, u) \\ &\leq u.d \end{aligned} \qquad (۲-۲۴)$$

(طبق خصوصیت کران بالا)

ولی از آن جایی که وقتی u در خط ۵ انتخاب شده بود، هر دوی u و y در $V - S$ قرار داشتند، داریم $y.d \leq u.d$. بنابراین دو نامساوی داده شده در (۲-۲۴) در واقع تساوی هستند، که به‌دست می‌دهد

$$y.d = \delta(s, y) = \delta(s, u) = u.d$$

در نتیجه $u.d = \delta(s, u)$، که با انتخاب ما از u تناقض دارد. نتیجه می‌گیریم که وقتی u به S اضافه شده است، $u.d = \delta(s, u)$، و بنابراین این تساوی بعد از آن همیشه برقرار است.

* *پایان.* در زمان پایان، $Q = \emptyset$، که به همراه ثابت قبلی $Q = V - S$، ایجاب می‌کند که $S = V$. بنابراین برای تمام رأس‌های $u \in V$، داریم $u \in V$. ■

نتیجه‌ی ۲۴-۷ اگر الگوریتم Dijkstra را بر روی یک گراف جهت‌دار و وزن‌دار $G = (V, E)$ با تابع وزن w، وزن‌های نامنفی و مبدأ s اجرا کنیم، آن گاه در زمان پایان زیرگراف عناصر ماقبل G_π یک درخت کوتاه‌ترین مسیرها با ریشه‌ی s است.

اثبات مستقیماً از قضیه‌ی ۲۴-۶ و خصوصیت زیرگراف عناصر ماقبل. ▒

تحلیل

سرعت الگوریتم Dijkstra چقدر است؟ این الگوریتم صف اولویت کمینه‌ی Q را با فراخوانی سه عملیات صف‌های اولویت نگه داری می‌کند: INSERT (به طور ضمنی در خط ۳)، EXTRACT-MIN (خط ۵)، و DECREASE-KEY (به طور ضمنی در RELAX، که در خط ۸ فراخوانی می‌شود). INSERT یک بار برای هر رأس فراخوانی می‌شود، همین طور است EXTRACT-MIN. چون هر رأس $v \in V$ دقیقاً یک بار به مجموعه‌ی S اضافه می‌شود، هر یال در لیست مجاورت $Adj[v]$ در حلقه‌ی for خطوط ۷-۸ و در کل الگوریتم دقیقاً یک بار بررسی می‌شود. از آن جایی که تعداد یال‌ها در کل لیست‌های مجاورت $|E|$ است، تعداد کل تکرارهای این حلقه‌ی for هم $|E|$ خواهد بود، و همچنین در کل حداکثر $|E|$ عملیات DECREASE-KEY اجرا خواهد شد. (یک بار دیگر مشاهده کنید که از تحلیل متراکم استفاده کرده‌ایم.)

زمان اجرای الگوریتم Dijkstra به روش پیاده‌سازی صف اولویت کمینه بستگی دارد. ابتدا حالتی را در نظر بگیرید که صف اولویت کمینه را با استفاده از شماره گذاری رأس‌ها از ۱ تا $|V|$ پیاده‌سازی می‌کنیم. به سادگی $v.d$ را در vامین ورودی یک آرایه ذخیره می‌کنیم. هر یک از عملیات INSERT و DECREASE-KEY به $O(1)$ زمان نیاز دارند، و هر عملیات EXTRACT-MIN، به زمان $O(V)$ (چرا که باید کل آرایه را جستجو کنیم)، که کل زمان برابر خواهد بود با $O(V^2 + E) = O(V^2)$.

اگر گراف به حد کافی خلوت باشد – به طور خاص، $E = o(V^2/\lg V)$ – پیاده‌سازی صف اولویت کمینه با استفاده از هرم کمینه‌ی دودویی سودمند خواهد بود. (همان طور که در بخش ۶-۵ بحث شد، یک نکته‌ی مهم پیاده‌سازی این است که رأس‌ها و عناصر مربوطه‌ی هرم باید اشاره‌گرهایی به یکدیگر داشته باشند.) در این صورت هر عملیات EXTRACT-MIN به زمان $O(\lg V)$ نیاز خواهد داشت. مانند قبل تعداد $|V|$ از این عملیات وجود دارد. زمان ساختن هرم کمینه دودویی $O(V)$ است. هر عملیات DECREASE-KEY به $O(\lg V)$ زمان نیاز دارد، و حداکثر $|E|$ از این عملیات اجرا خواهد شد. بنابراین کل زمان اجرا $O((V + E)\lg V)$ خواهد بود، که در صورتی که تمام رأس‌ها از مبدأ قابل دسترس باشند، برابر است با $O(E \lg V)$. اگر $E = o(V^2/\lg V)$، این زمان اجرا نسبت به پیاده‌سازی سرراست با زمان $O(V^2)$ یک پیشرفت محسوب می‌شود.

در واقع با پیاده‌سازی صف اولویت کمینه با استفاده از یک هرم فیبوناچی (فصل ۲۰ را ببینید) می‌توانیم به زمان اجرای $O(V \lg V + E)$ دست یابیم. هزینه‌ی سرشکن هر یک از $|V|$ عملیات EXTRACT-MIN برابر $O(\lg V)$ است، و هر یک از فراخوانی‌های DECREASE-KEY، که حداکثر $|E|$ تا از آن‌ها خواهیم داشت، فقط به زمان سرشکن $O(1)$ نیاز دارند. به صورت تاریخی، هرم‌های فیبوناچی این گونه گسترش یافتند که مشاهده شد که در الگوریتم Dijkstra معمولاً تعداد فراخوانی‌های DECREASE-KEY بسیار بیشتر از فراخوانی‌های EXTRACT-MIN است، بنابراین هر متدی که

هزینه‌ی سرشکن DECREASE-KEY را به $o(\lg V)$ کاهش دهد بدون این که هزینه‌ی سرشکن EXTRACT-MIN را افزایش دهد، به یک پیاده‌سازی به صورت حدی سریع‌تر از پیاده‌سازی با استفاده از هرم‌های دودویی ختم خواهد شد.

الگوریتم Dijkstra شباهت‌هایی با هر دو الگوریتم جستجوی سطح اول (بخش ۲۲-۲) و الگوریتم پرایم برای محاسبه‌ی درخت‌های پوشای کمینه (بخش ۲۳-۲) دارد. این الگوریتم از این رو مانند جستجوی سطح اول است که مجموعه‌ی S متناظر است با مجموعه‌ی رأس‌های سیاه در جستجوی سطح اول؛ همان طور رأس‌های S به وزن نهایی کوتاه‌ترین مسیر خود رسیده‌اند، رأس‌های سیاه در جستجوی سطح اول هم فاصله‌ی صحیح سطح اول خود را دارند. الگوریتم Dijkstra مانند الگوریتم پرایم است چون که هر دو الگوریتم از یک صف اولویت کمینه برای یافتن «سبک‌ترین» رأس بیرون یک مجموعه (مجموعه‌ی S در الگوریتم Dijkstra و درخت در حال رشد در الگوریتم پرایم) استفاده می‌کنند، آن رأس را به مجموعه اضافه، و نسبت به وزن آن رأس‌های باقی مانده‌ی بیرون مجموعه را اصلاح می‌کنند.

تمرین‌ها

۲۴-۳-۱ الگوریتم Dijkstra را بر روی گراف جهت‌دار شکل ۲۴-۲ اجرا کنید، ابتدا با استفاده از s به عنوان مبدأ، و سپس با استفاده از z به عنوان مبدأ. به سبک شکل ۲۴-۶، مقادیر d و π و رأس‌های درون مجموعه‌ی S را بعد از هر تکرار حلقه‌ی while نشان دهید.

۲۴-۳-۲ یک مثال ساده از یک گراف جهت‌دار با یال‌هایی با وزن منفی بزنید که الگوریتم Dijkstra برای آن جواب‌های غلط تولید می‌کند. وقتی یال‌های با وزن منفی وجود دارند، چرا اثبات قضیه‌ی ۲۴-۶ کاربرد ندارد؟

۲۴-۳-۳ فرض کنید که خط ۴ الگوریتم Dijkstra را به صورت زیر تغییر می‌دهیم.

4 while $|Q| > 1$

این تغییر باعث می‌شود که حلقه‌ی while به جای این که $|V|-1$ بار اجرا شود، $|V|$ بار اجرا شود. آیا این الگوریتم همچنان به درستی کار می‌کند؟

۲۴-۳-۴ پروفسور Gaedel برنامه‌ای نوشته است که ادعا می‌کند الگوریتم Dijkstra را پیاده‌سازی می‌کند. این برنامه $v.d$ و $v.\pi$ را برای هر رأس $v \in V$ می‌سازد. یک الگوریتم با زمان $O(V+E)$ ارائه کنید که خروجی برنامه‌ی پروفسور را تست کند. برنامه‌ی شما باید تعیین کند که آیا خصیصه‌های d و π با مقادیر مربوطه برای یک درخت کوتاه‌ترین مسیرها مطابقت می‌کند یا خیر. می‌توانید فرض کنید که وزن تمام یال‌ها نامنفی است.

۲۴-۳-۵ پروفسور Newman فکر می‌کند که یک اثبات ساده‌تر برای صحت الگوریتم Dijkstra به

دست آورده است. او ادعا می‌کند که الگوریتم Dijkstra یال‌های هر کوتاه‌ترین مسیر در گراف را به ترتیبی ترمیم می‌کند که در مسیر ظاهر می‌شوند، و بنابراین خصوصیت ترمیم مسیر در مورد تمام رأس‌هایی که از مبدأ قابل دسترس هستند، صادق است. با ساختن یک گراف جهت‌دار که در آن الگوریتم Dijkstra یال‌های یک کوتاه‌ترین مسیر را خارج از ترتیب ترمیم می‌کند، نشان دهید که پروفسور اشتباه می‌کند.

۶-۳-۲۴ یک گراف جهت‌دار $G = (V, E)$ به ما داده شده است، که در آن هر یال $(u, v) \in E$ یک مقدار متناظر $r(u, v)$ دارد، که یک عدد صحیح در بازه‌ی $0 \le r(u, v) \le 1$ است و میزان امنیت یک کانال ارتباطی از رأس u به رأس v را نشان می‌دهد. با $r(u, v)$ به صورت احتمال این که کانال u به v به درستی عمل کند برخورد می‌کنیم، و فرض می‌کنیم که این احتمالات مستقل از یکدیگر هستند. یک الگوریتم کارامد برای یافتن مطمئن‌ترین مسیر بین دو رأس داده شده ارائه کنید.

۷-۳-۲۴ فرض کنید $G = (V, E)$ یک گراف جهت‌دار و وزن‌دار باشد، با تابع وزن $w : E \to \{1, 2, \ldots, W\}$، برای یک عدد صحیح و مثبت W، و همچنین فرض کنید که وزن کوتاه‌ترین مسیر هیچ دو رأسی از رأس مبدأ s یکسان نیست. اکنون فرض کنید که یک گراف جهت‌دار بدون وزن $G' = (V \cup V', E')$ تعریف می‌کنیم، بدین صورت که هر یال $(u, v) \in E$ را با $w(u, v)$ یال با وزن واحد به صورت سری جایگزین می‌کنیم. G' چند رأس دارد؟ اکنون فرض کنید یک جستجوی سطح اول بر روی G' اجرا می‌کنیم. نشان دهید که ترتیب سیاه شدن رأس‌های درون V در جستجوی عمق اول G' با ترتیب استخراج رأس‌ها از صف اولویت کمینه در خط ۵ رویه‌ی DIJKSTRA، وقتی که روی G اجرا می‌شود یکسان است.

۸-۳-۲۴ فرض کنید $G = (V, E)$ یک گراف جهت‌دار وزن‌دار باشد، با تابع وزن $w : E \to \{0, 1, \ldots, W\}$ برای یک عدد صحیح نامنفی W. الگوریتم Dijkstra را طوری اصلاح کنید که کوتاه‌ترین مسیرها را از یک رأس مبدأ s در زمان $O(WV + E)$ محاسبه کند.

۹-۳-۲۴ الگوریتم خود در تمرین ۲۴-۳-۸ را طوری اصلاح کنید که در زمان $O((V + E) \lg W)$ اجرا شود. (*راهنمایی:* در هر لحظه از زمان چند تخمین کوتاه‌ترین مسیر مجزا در $V - S$ می‌تواند وجود داشته باشد؟)

۱۰-۳-۲۴ فرض کنید یک گراف جهت‌دار و وزن‌دار $G = (V, E)$ داریم که در آن یال‌هایی که از رأس مبدأ s خارج می‌شوند ممکن است وزن منفی داشته باشند، وزن تمام یال‌های دیگر نامنفی است، و هیچ دور با وزن منفی وجود ندارد. بحث کنید که الگوریتم Dijkstra در این گراف کوتاه‌ترین مسیرها را از s به درستی محاسبه می‌کند.

۲۴-۴ محدودیت‌های اختلاف و کوتاه‌ترین مسیرها

در فصل ۲۹ مسئله‌ی کلی برنامه‌ریزی خطی بررسی می‌شود، که در آن می‌خواهیم یک تابع خطی را تحت مجموعه‌ای از نامساوی‌های خطی بهینه کنیم. در این بخش حالت خاصی از برنامه‌ریزی خطی را بررسی خواهیم کرد، که می‌توان آن را به مسئله‌ی یافتن کوتاه‌ترین مسیرها از یک مبدأ تبدیل کرد. مسئله‌ی کوتاه‌ترین مسیرها از یک مبدأ حاصل را می‌توانیم با الگوریتم بلمن-فورد حل کنیم، که در نتیجه مسئله‌ی برنامه‌ریزی خطی را هم کرده‌ایم.

برنامه‌ریزی خطی

در مسئله‌ی کلی *برنامه‌ریزی خطی* (linear-programming problem)، به ما یک ماتریس A با اندازه‌ی $m \times n$، یک m-بردار b، و یک n-بردار c داده شده است. می‌خواهیم یک بردار x با n عنصر بیابیم که *تابع هدف* (object function) $\sum_{i=1}^{n} c_i x_i$ را تحت m محدودیت داده شده به صورت $Ax \leq b$ کمینه می‌کند.

با این که الگوریتم سیمپلکس، که موضوع فصل ۲۹ است، همیشه در زمان چند جمله‌ای نسبت به اندازه‌ی ورودی اجرا نمی‌شود، الگوریتم‌های برنامه‌ریزی خطی دیگری هستند که در زمان خطی اجرا می‌شوند. دلایل بسیاری برای درک ساختار مسائل برنامه‌ریزی خطی وجود دارد، که در این جا دو تا از آن‌ها را ذکر می‌کنیم. اول، دانستن این که یک مسئله‌ی داده شده را می‌توان به یک مسئله‌ی برنامه‌ریزی خطی با اندازه‌ی چند جمله‌ای تبدیل کرد، سریعا نتیجه می‌دهد که یک الگوریتم با زمان چند جمله‌ای برای آن مسئله وجود دارد. دوم، حالت‌های خاص بسیاری از برنامه‌ریزی خطی وجود دارد که برای آن‌ها الگوریتم‌های سریع‌تری وجود دارد. به عنوان مثال مسئله‌ی کوتاه‌ترین مسیر بین دو رأس (تمرین ۲۴-۴-۴) و مسئله‌ی شار بیشینه (تمرین ۲۶-۱-۸) حالت‌های خاصی از برنامه‌ریزی خطی هستند.

بعضی مواقع تابع هدف برای ما اهمیتی ندارد؛ هدف ما فقط این است که یک *جواب ممکن* (feasible solution) بیابیم، یعنی هر برداری مانند x که $Ax \leq b$ را ارضا می‌کند، یا فهمیدن این که هیچ جواب ممکنی وجود ندارد. بر روی این *مسئله‌ی امکان‌پذیری* هم تمرکز خواهیم کرد.

سیستم محدودیت‌های اختلاف

در یک *سیستم محدودیت‌های اختلاف* (system of difference constraints)، هر ردیف از ماتریس برنامه‌ریزی خطی A حاوی یک ۱ و یک ۱- است، و تمام ورودی‌های دیگر A ۰ هستند. بنابراین محدودیت‌های داده شده توسط $Ax \leq b$، مجموعه‌ای از m *محدودیت‌های اختلاف* هستند شامل n مجهول، که در آن هر محدودیت یک نامساوی خطی ساده به شکل

$$x_j - x_i \leq b_k$$

است، که در آن $1 \leq i, j \leq n$ و $1 \leq k \leq m$.

برای مثال مسئله‌ی یافتن ۵-بردار $x = (x_i)$ را در نظر بگیرید که

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}$$

را ارضا می‌کند. این مسئله معادل است با یافتن مجهول‌های x_i، برای $i = 1, 2, ..., 5$، به طوری که ۸ محدودیت اختلاف زیر ارضا شوند:

$$x_1 - x_2 \leq 0 \qquad (۳-۲۴)$$

$$x_1 - x_5 \leq -1 \qquad (۴-۲۴)$$

$$x_2 - x_5 \leq 1 \qquad (۵-۲۴)$$

$$x_3 - x_1 \leq 5 \qquad (۶-۲۴)$$

$$x_3 - x_1 \leq 5 \qquad (۷-۲۴)$$

$$x_4 - x_3 \leq -1 \qquad (۸-۲۴)$$

$$x_5 - x_3 \leq -3 \qquad (۹-۲۴)$$

$$x_5 - x_4 \leq -3 \qquad (۱۰-۲۴)$$

یک جواب این مسئله $x = (-5, -3, 0, -1, -4)$ است، که می‌توان صحت آن را مستقیماً با چک کردن هر نامساوی بررسی کرد. در واقع بیش از یک جواب برای این مسئله وجود دارد. یک جواب دیگر $x' = (0, 2, 5, 4, 1)$ است. این دو جواب به یکدیگر مربوط هستند: هر مؤلفه‌ی x'، ۵ واحد بیشتر از مؤلفه‌ی مربوطه در x است. این واقعیت تصادفی نیست.

لم ۲۴-۹ فرض کنید $x = (x_1, x_2, ..., x_n)$ یک جواب سیستم $Ax \leq b$ از محدودیت‌های اختلاف، و همچنین d یک ثابت باشد. آن گاه $x + d = (x_1 + d, x_2 + d, ..., x_n + d)$ هم یک جواب برای $Ax \leq b$ است.

اثبات برای هر x_i و x_j داریم $(x_j + d) - (x_i + d) = x_j - x_i$. بنابراین اگر x نامساوی $Ax \leq b$ را ارضا کند، $x + d$ هم آن را ارضا می‌کند. ■

سیستم محدودیت‌های اختلاف در کاربردهای مختلف بسیاری پیش می‌آید. به عنوان مثال

مجهولات x_i ممکن است زمان‌هایی باشند که در آن‌ها رخدادهایی اتفاق می‌افتند. هر محدودیت را می‌توان به صورت یک شرط دید که مشخص می‌کند باید حداقل (یا حداکثر) یک اختلاف زمانی خاصی بین دو رخداد باشد. ممکن است رخدادها، وظایفی باشند که باید در خط تولید یک محصول انجام شوند. اگر بخواهیم یک بخش در زمان x_1 اضافه کنیم که انجام آن ۲ ساعت طول می‌کشد، و سپس برای شروع نصب یک قطعه در زمان x_2 مجبور باشیم صبر کنیم که این بخش تمام شود، آن گاه محدودیت $x_2 \geq x_1 + 2$، یا معادل آن $x_1 - x_2 \leq -2$ را خواهیم داشت. در عوض ممکن است بخواهیم نصب قطعه بعد از شروع بخش اضافی انجام شود، ولی نباید دیرتر از ۱ ساعت بعد از شروع آن باشد. در این حالت جفت محدودیت $x_1 \leq x_2$ و $x_2 \leq x_1 + 1$، یا معادل آن $x_1 - x_2 \leq 0$ و $x_2 - x_1 \leq 1$ را داریم.

گراف‌های محدودیت

مفید است که با سیستم‌های محدودیت‌های اختلاف از دید تئوری گراف برخورد کنیم. ایده این است که یک سیستم $Ax \leq b$ از محدودیت‌های اختلاف، یعنی ماتریس برنامه‌ریزی خطی A با اندازه‌ی $m \times n$ را می‌توان به صورت ترانهاده‌ی ماتریس مجاورت یک گراف با n رأس و m یال دید (تمرین ۲۲-۱-۷ را ببینید). برای $i = 1, 2, ..., n$، هر رأس v_i در گراف متناظر است با یکی از n مجهول x_i. هر یال جهت‌دار در گراف متناظر است با یکی از m نامساوی شامل دو مجهول.

به صورت رسمی‌تر با داشتن یک سیستم $Ax \leq b$ از محدودیت‌های اختلاف، *گراف محدودیت* (constraint graph) متناظر آن یک گراف جهت‌دار وزن‌دار $G = (V, E)$ است، که در آن

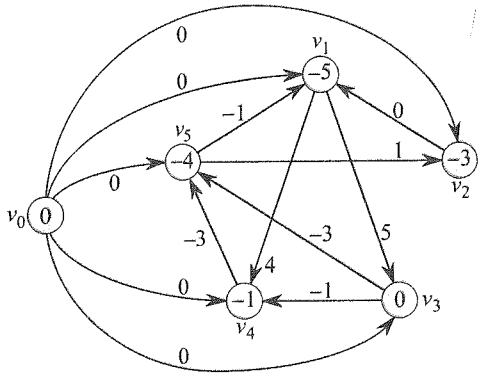
$$V = \{v_0, v_1, ..., v_n\}$$

و

$$E = \{(v_i, v_j) \mid x_j - x_i \leq b_k \text{ است یک محدودیت}\} \cup \{(v_0, v_1), (v_0, v_2), (v_0, v_3), ..., (v_0, v_n)\}$$

همان طور که به زودی خواهیم دید، رأس اضافی v_0 به گراف پیوسته است تا تضمین کند که تمام رأس‌های دیگر از آن قابل دسترس هستند. بنابراین مجموعه‌ی رأس V حاوی یک رأس v_i برای هر مجهول x_i است، به علاوه‌ی یک رأس اضافی v_0. مجموعه‌ی یال E برای هر محدودیت اختلاف حاوی یک یال است، به علاوه‌ی یک یال (v_0, v_i) برای هر مجهول x_i. اگر $x_j - x_i \leq b_k$ یک محدودیت اختلاف باشد، آن گاه وزن یال (v_i, v_j) برابر است با $w(v_i, v_j) = b_k$. وزن هر یال خروجی از v_0 برابر ۰ است. شکل ۲۴-۸ گراف محدودیت سیستم (۲۴-۳) - (۲۴-۱۰) از محدودیت‌های اختلاف را نشان می‌دهد.

قضیه‌ی زیر نشان می‌دهد که می‌توان با یافتن وزن کوتاه‌ترین مسیرها در گراف محدودیت مربوطه یک جواب برای سیستمی از محدودیت‌های اختلاف پیدا کرد.



شکل ۲۴-۸ گراف محدودیت متناظر با سیستم (۲۴-۳) - (۲۴-۱۰) از محدودیت‌های اختلاف. مقدار $\delta(v_0, v_i)$ در هر رأس v_i نشان داده شده است. یک جواب ممکن برای سیستم $x = (-5, -3, 0, -1, -4)$ است.

قضیه‌ی ۲۴-۹ با داشتن یک سیستم $Ax \le b$ از محدودیت‌های اختلاف، فرض کنید $G = (V, E)$ گراف محدودیت متناظر باشد. اگر G حاوی هیچ دور با وزن منفی نباشد، آن گاه

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \ldots, \delta(v_0, v_n)) \qquad (۲۴-۱۱)$$

یک جواب ممکن برای سیستم است. اگر G حاوی یک دور با وزن منفی باشد، آن گاه هیچ جواب ممکنی برای سیستم وجود ندارد.

اثبات ابتدا نشان می‌دهیم که اگر گراف محدودیت دور با وزن منفی نداشته باشد، آن گاه تساوی (۲۴-۱۱) یک جواب ممکن را به دست می‌دهد. یک یال $(v_i, v_j) \in E$ را در نظر بگیرید. طبق نامساوی مثلث، $\delta(v_0, v_j) \le \delta(v_0, v_i) + w(v_i, v_j)$، یا به‌طور معادل $\delta(v_0, v_j) - \delta(v_0, v_i) \le w(v_i, v_j)$. بنابراین قرار دادن $x_i = \delta(v_0, v_i)$ و $x_j = \delta(v_0, v_j)$ محدودیت اختلاف $x_j - x_i \le w(v_i, v_j)$ را که متناظر است با یال (v_i, v_j) ارضا می‌کند.

اکنون نشان می‌دهیم که اگر گراف محدودیت حاوی یک دور با وزن منفی باشد، آن گاه سیستم محدودیت‌های اختلاف هیچ جواب ممکنی ندارد. بدون از دست دادن کلیت فرض کنید دور با وزن منفی $c = \langle v_1, v_2, \ldots, v_k \rangle$ باشد، که در آن $v_1 = v_k$. (رأس v_0 نمی‌تواند روی دور c باشد، چرا که هیچ یال ورودی ندارد.) دور c متناظر است با محدودیت‌های اختلاف زیر:

$$\begin{aligned}
x_2 - x_1 &\le w(v_1, v_2) \\
x_3 - x_2 &\le w(v_2, v_3) \\
&\vdots \\
x_k - x_{k-1} &\le w(v_{k-1}, v_k) \\
x_1 - x_k &\le w(v_k, v_1)
\end{aligned}$$

فرض کنید یک جواب برای x وجود دارد که هر یک از k نامساوی را ارضا می‌کند. این جواب همچنین باید در نامساوی حاصل از جمع این k نامساوی با یکدیگر صدق کند. اگر سمت چپ نامساوی‌ها را با یکدیگر جمع کنیم، هر مجهول x_i یک بار اضافه شده و یک بار کم می‌شود (به خاطر بیاورید که $v_1 = v_k$ ایجاب می‌کند $x_1 = x_k$)، بنابراین سمت چپ مجموع ۰ است. سمت راست نامساوی $w(c)$ است، و بنابراین خواهیم داشت $w(c) \le 0$. ولی از آن جایی که c یک دور با وزن منفی است، $w(c) < 0$، و به تناقض $w(c) \le 0 < 0$ می‌رسیم. ■

حل سیستم‌های محدودیت‌های اختلاف

قضیه‌ی ۲۴-۹ به ما می‌گوید که می‌توانیم از الگوریتم بلمن- فورد برای حل یک سیستم محدودیت‌های اختلاف استفاده کنیم. چون یال‌هایی از رأس مبدأ v_0 به تمام رأس‌های دیگر در گراف محدودیت وجود دارد، هر دور با وزن منفی در گراف محدودیت از v_0 قابل دسترس است. اگر الگوریتم بلمن- فورد TRUE را بازگرداند، آن گاه وزن کوتاه‌ترین مسیرها یک جواب ممکن به سیستم می‌دهند. به عنوان مثال در شکل ۲۴-۸، وزن کوتاه‌ترین مسیرها یک جواب ممکن $x = (-5, -3, 0, -1, -4)$ را فراهم می‌کنند، و طبق لم ۲۴-۸ برای هر ثابت d، $x = (d-5, d-3, d, d-1, d-4)$ هم یک جواب ممکن است. اگر الگوریتم بلمن- فورد FALSE را بازگرداند، هیچ جواب ممکنی برای سیستم محدودیت‌های اختلاف وجود ندارد.

یک سیستم محدودیت‌های اختلاف با m محدودیت و n مجهول یک گراف با $n+1$ رأس و $n+m$ یال می‌سازد. بنابراین با استفاده از الگوریتم بلمن- فورد، می‌توان سیستم را در زمان $O((n+1)(n+m)) = O(n^2 + nm)$ حل کرد. تمرین ۲۴-۴-۵ از شما می‌خواهد الگوریتم را طوری اصلاح کنید که در زمان $O(nm)$ اجرا شود، حتی اگر m بسیار کوچک‌تر از n باشد.

تمرین‌ها

۲۴-۴-۱ یک جواب ممکن برای سیستم محدودیت‌های اختلاف زیر بیابید، و یا تعیین کنید که هیچ جواب ممکنی برای آن وجود ندارد:

$$x_1 - x_2 \le 1,$$
$$x_1 - x_4 \le -4,$$
$$x_2 - x_3 \le 2,$$
$$x_2 - x_5 \le 7,$$
$$x_2 - x_6 \le 5,$$
$$x_3 - x_6 \le 10,$$
$$x_4 - x_2 \le 2,$$
$$x_5 - x_1 \le -1,$$

$$x_۵ - x_۴ \le ۳,$$
$$x_۶ - x_۳ \le -۸,$$

۲-۴-۲۴ یک جواب ممکن برای سیستم محدودیت‌های اختلاف زیر بیابید، و یا تعیین کنید که هیچ جواب ممکنی برای آن وجود ندارد:

$$x_۱ - x_۲ \le ۴,$$
$$x_۱ - x_۵ \le ۵,$$
$$x_۲ - x_۴ \le -۶,$$
$$x_۳ - x_۲ \le ۱,$$
$$x_۴ - x_۱ \le ۳,$$
$$x_۴ - x_۳ \le ۵,$$
$$x_۴ - x_۵ \le ۱۰,$$
$$x_۵ - x_۳ \le -۴,$$
$$x_۵ - x_۴ \le -۸,$$

۳-۴-۲۴ آیا وزن یک کوتاه‌ترین مسیر از رأس جدید v_0 در یک گراف محدودیت می‌تواند مثبت باشد؟ توضیح دهید.

۴-۴-۲۴ مسئله‌ی کوتاه‌ترین مسیرها از یک مبدأ را به صورت یک مسئله‌ی برنامه‌ریزی خطی توصیف کنید.

۵-۴-۲۴ نشان دهید که چگونه می‌توان الگوریتم بلمن-فورد را کمی اصلاح کرد به طوری که وقتی از آن برای حل یک سیستم از محدودیت‌های اختلاف با m نامساوی بر روی n مجهول اجرا می‌شود، زمان اجرا $O(nm)$ باشد.

۶-۴-۲۴ فرض کنید علاوه بر یک سیستم محدودیت‌های اختلاف، می‌خواهیم یک *محدودیت تساوی* به شکل $x_i = x_j + b_k$ را هم ارضا کنیم. نشان دهید که می‌توان الگوریتم بلمن-فورد را طوری توسعه داد که این نوع سیستم محدودیت را هم حل کند.

۷-۴-۲۴ نشان دهید که با استفاده از یک الگوریتم شبیه الگوریتم بلمن-فورد، می‌توان یک سیستم محدودیت اختلاف را بدون رأس اضافه‌ی v_0 حل کرد.

★ ۸-۴-۲۴ فرض کنید $Ax \le b$ یک سیستم با m محدودیت اختلاف و n مجهول باشد. نشان دهید که وقتی الگوریتم بلمن-فورد بر روی گراف محدودیت متناظر اجرا شود، $\sum_{i=1}^{n} x_i$ را برای تمام x_iها تحت $Ax \le b$ و $x_i \le 0$ بیشینه می‌کند.

★ ۹-۴-۲۴ نشان دهید وقتی الگوریتم بلمن-فورد بر روی گراف محدودیت یک سیستم $Ax \le b$ از محدودیت‌های اختلاف اجرا شود، کمیت $(\max\{x_i\} - \min\{x_i\})$ را تحت $Ax \le b$

کمینه می‌کند. توضیح دهید که وقتی از الگوریتم برای برنامه‌ریزی وظایف تولیدی استفاده می‌شود، چگونه ممکن است این مسئله به کار آید.

۲۴-۴-۱۰ فرض کنید که تمام ردیف‌ها در ماتریس A مربوط به یک برنامه‌ریزی خطی $Ax \le b$ متناظر است با یک محدودیت اختلاف، یک محدودیت تک متغیره به شکل $x_i \le b_k$، یا یک محدودیت تک متغیره به شکل $-x_i \le b_k$. نشان دهید که چگونه می‌توان الگوریتم بلمن–فورد را طوری توسعه داد که این نسخه از سیستم محدودیت را حل کند.

۲۴-۴-۱۱ یک الگوریتم بهینه برای حل سیستم $Ax \le b$ از محدودیت‌های اختلاف بدهید، که در آن تمام عناصر b مقادیر حقیقی هستند و تمام مجهول‌های x_i باید عدد صحیح باشند.

★ ۲۴-۴-۱۲ یک الگوریتم کارامد برای حل سیستم $Ax \le b$ از محدودیت‌های اختلاف بدهید، که در آن تمام عناصر b مقادیر حقیقی هستند، و یک زیرمجموعه‌ی خاص از بعضی مجهول‌های x_i، نه لزوماً همه، باید عدد صحیح باشند.

۲۴-۵ اثبات خصوصیت‌های کوتاه‌ترین مسیرها

در طول این فصل بحث‌های درستی ما بر پایه‌ی نامساوی مثلث، خصوصیت کران بالا، خصوصیت عدم وجود مسیر، خصوصیت همگرایی، خصوصیت ترمیم مسیر، و خصوصیت زیرگراف عناصر ماقبل استوار بود. در ابتدای این فصل این خصوصیات بدون اثبات گفته شد. در این بخش آن‌ها را اثبات خواهیم کرد.

نامساوی مثلث

در بحث جستجوی سطح اول (بخش ۲۲-۲)، به عنوان لم ۲۲-۱ یک خصوصیت ساده از کوتاه‌ترین فاصله‌ها را در گراف‌های بدون وزن اثبات کردیم. نامساوی مثلث این خصوصیت را برای گراف‌های وزن‌دار توسعه می‌دهد.

لم ۲۴-۱۰ (نامساوی مثلث)

فرض کنید $G = (V, E)$ یک گراف جهت‌دار و وزن‌دار باشد، با تابع وزن $w : E \to \mathbb{R}$ و رأس مبدأ s. آن گاه برای تمام یال‌های $(u,v) \in E$ داریم

$$\delta(s, v) \le \delta(s, u) + w(u, v)$$

اثبات فرض کنید یک کوتاه‌ترین مسیر p از مبدأ x به رأس v وجود داشته باشد. آن گاه وزن p بیشتر از هیچ مسیر دیگری از s به v نیست. به طور خاص، وزن مسیر p بیشتر از مسیر خاصی که یک کوتاه‌ترین مسیر از مبدأ s به رأس u و سپس یال (u, v) را طی می‌کند نیست.

تمرین ۲۴-۵-۳ از شما می‌خواهد اثبات را برای حالتی انجام دهید که هیچ کوتاه‌ترین مسیری از s به v وجود ندارد.

تأثیرات ترمیم بر روی تخمین‌های کوتاه‌ترین مسیر

گروه بعدی لم‌ها توضیح می‌دهند که تخمین‌های کوتاه‌ترین مسیرها چگونه با اجرای دنباله‌ای از مراحل ترمیم بر روی یال‌های یک گراف جهت‌دار و وزن‌دار که با INITIALIZE-SINGLE-SOURCE مقداردهی اولیه شده است، تحت تأثیر قرار می‌گیرند.

لم ۲۴-۱۱ (خصوصیت کران بالا) فرض کنید $G=(V,E)$ یک گراف جهت‌دار و وزن‌دار با تابع وزن $w: E \to \mathbb{R}$ باشد. فرض کنید $s \in V$ رأس مبدأ بوده و گراف با INITIALIZE-SINGLE-SOURCE (G,s) مقداردهی اولیه شده باشد. آن گاه برای هر $v \in V$ داریم $v.d \geq \delta(s,v)$، و این ثابت در طول هر دنباله‌ای از مراحل ترمیم بر روی یال‌های G حفظ می‌شود. به علاوه وقتی $v.d$ به کران پایین $\delta(s,v)$ رسید، دیگر هیچ گاه تغییر نمی‌کند.

اثبات درستی ثابت $v.d \geq \delta(s,v)$ را برای تمام رأس‌های $v \in V$ به کمک استقرا بر روی تعداد مراحل ترمیم اثبات می‌کنیم.

برای حالت پایه، $v.d \geq \delta(s,v)$ بعد از مقداردهی اولیه بدیهتاً درست است، چرا که $s.d = 0 \geq \delta(s,s)$، و همچنین چون $v.d = \infty$ نتیجه می‌دهد $v.d \geq \delta(s,v)$ برای هر $v \in V - \{s\}$ (توجه کنید که اگر s بر روی یک دور با وزن منفی باشد، $\delta(s,s)$ برابر با $-\infty$ است، و در غیر این صورت 0 خواهد بود).

برای گام استقرا ترمیم یک یال (u,v) را در نظر بگیرید. طبق فرض استقرا قبل از ترمیم برای هر $x \in V$ داریم $x.d \geq \delta(s,x)$. تنها مقدار d که ممکن است تغییر کند $v.d$ است. اگر این مقدار تغییر کند، داریم

$$\begin{aligned} v.d &= u.d + w(u,v) \\ &\geq \delta(s,u) + w(u,v) && \text{(طبق فرض استقرا)} \\ &\geq \delta(s,v) && \text{(طبق نامساوی مثلث)} \end{aligned}$$

و بنابراین ثابت برقرار است.

برای این که ببینیم مقدار $v.d$ پس از رسیدن به $v.d = \delta(s,v)$ هیچ گاه تغییر نمی‌کند، توجه کنید که $v.d$ با رسیدن به کران پایین خود، دیگر نمی‌تواند کاهش یابد چرا که قبلاً نشان دادیم $v.d \geq \delta(s,v)$، و نمی‌تواند افزایش یابد چرا که مراحل ترمیم مقادیر d را تغییر نمی‌دهند.

نتیجه‌ی ۲۴-۱۲ فرض کنید در یک گراف وزن‌دار و جهت‌دار $G=(V,E)$ با تابع وزن $w: E \to \mathbb{R}$ هیچ مسیری رأس مبدأ $s \in V$ را به یک رأس داده شده‌ی $v \in V$ متصل نمی‌کند. آن

(خصوصیت عدم وجود مسیر) گاه بعد از مقداردهی اولیه‌ی گراف با INITIALIZE-SINGLE-SOURCE(G, s) داریم $v.d = (s, v)$، و این تساوی به صورت یک ثابت در تمام دنباله‌های مراحل ترمیم بر روی یال‌های G برقرار می‌ماند.

اثبات طبق خصوصیت کران بالا همیشه داریم $v.d \leq \delta(s, v) = \infty$، و بنابراین $v.d = \infty \delta(s, v)$. ■

لم ۲۴-۱۲ فرض کنید $G = (V, E)$ یک گراف جهت‌دار و وزن‌دار با تابع وزن $w : E \to \mathbb{R}$ باشد، و همچنین $(u, v) \in E$. آن‌گاه دقیقاً بعد از ترمیم یال (u, v) با اجرای RELAX(u, v, w)، داریم $v.d \leq u.d + w(u, v)$.

اثبات اگر دقیقاً قبل از ترمیم یال (u, v) داشته باشیم $v.d > u.d + w(u, v)$، آن گاه بعد از آن خواهیم داشت $v.d = u.d + w(u, v)$. در عوض اگر دقیقاً قبل از ترمیم داشته باشیم $v.d \leq u.d + w(u, v)$، آن‌گاه نه $u.d$ تغییر می‌کند و نه $v.d$، و بنابراین بعد از آن $v.d \leq u.d + w(u, v)$. ■

لم ۲۴-۱۳ (خصوصیت همگرایی) فرض کنید $G = (V, E)$ یک گراف جهت‌دار و وزن‌دار باشد، با تابع وزن $w : E \to \mathbb{R}$، و $s \in V$ یک رأس مبدأ، و $s \rightsquigarrow u \to v$ یک کوتاه‌ترین مسیر در G برای رأس‌های $u, v \in V$. فرض کنید G با INITIALIZE-SINGLE-SOURCE(G, s) مقداردهی اولیه شده باشد، و سپس دنباله‌ای از مراحل ترمیم که شامل فراخوانی RELAX(u, v, w) هستند بر روی یال‌های G اجرا شده‌اند. اگر در هر زمانی قبل از فراخوانی داشته باشیم $u.d = \delta(s, u)$، آن گاه بعد از آن همیشه خواهیم داشت $v.d = \delta(s, v)$.

اثبات طبق خصوصیت کران بالا اگر زمانی قبل از ترمیم یال (u, v) داشته باشیم $u.d = \delta(s, u)$، آن گاه این تساوی بعد از آن برقرار خواهد بود. به طور خاص بعد از ترمیم یال (u, v) داریم

$$\begin{aligned} v.d &\leq u.d + w(u, v) \\ &= \delta(s, u) + w(u, v) && \text{(طبق لم ۲۴-۱۳)} \\ &= \delta(s, v) && \text{(طبق لم ۲۴-۱)} \end{aligned}$$

طبق خصوصیت کران بالا داریم $v.d \geq \delta(s, v)$، که از آن نتیجه می‌گیریم که $v.d = \delta(s, v)$، و این تساوی بعد از آن برقرار است. ■

لم ۲۴.۱۵ (خصوصیت ترمیم بسیط)

فرض کنید $G = (V, E)$ یک گراف جهت‌دار و وزن‌دار باشد، با تابع وزن $w : E \to \mathbb{R}$ و رأس مبدأ $s \in V$. یک مسیر $p = \langle v_0, v_1, ..., v_k \rangle$ از $s = v_0$ به v_k را در نظر بگیرید. اگر G با INITIALIZE-SINGLE-SOURCE(G, s) مقداردهی اولیه شده باشد، و سپس دنباله‌ای از مراحل ترمیم انجام شده باشد، که به ترتیب شامل ترمیم یال‌های (v_0, v_1)، (v_1, v_2)،...، (v_{k-1}, v_k) باشد، آن گاه بعد از این ترمیم‌ها همیشه داریم $v_k.d = \delta(s, v_k)$. این خصوصیت مستقل از این که چه یال‌های دیگری ترمیم شده باشند، حتی شامل یال‌های خود مسیر p، برقرار است.

اثبات به وسیله‌ی استقرا نشان می‌دهیم که بعد از ترمیم iامین یال در مسیر p داریم $v_i.d = \delta(s, v_i)$. برای پایه، $i = 0$، و قبل از این که یالی از p ترمیم شود از مقداردهی اولیه داریم $v_0.d = s.d = 0 = \delta(s, s)$. طبق خصوصیت کران بالا مقدار $s.d$ بعد از مقداردهی اولیه هیچ گاه تغییر نمی‌کند.

برای گام استقرا فرض می‌کنیم که $v_{i-1}.d = \delta(s, v_{i-1})$، و ترمیم یال (v_{i-1}, v_i) را بررسی می‌کنیم. طبق خصوصیت همگرایی بعد از ترمیم داریم $v_i.d = \delta(s, v_i)$، و این تساوی بعد از آن همیشه برقرار است. ▩

ترمیم و درخت‌های کوتاه‌ترین مسیرها

اکنون نشان می‌دهیم که وقتی دنباله‌ای از ترمیم‌ها باعث شد که تخمین‌های کوتاه‌ترین مسیرها به وزن کوتاه‌ترین مسیرها میل کنند، زیرگراف عناصر ماقبل ایجاد شده توسط مقادیر π حاصل یک درخت کوتاه‌ترین مسیرها برای G است. با لم زیر آغاز می‌کنیم، که نشان می‌دهد زیرگراف عناصر ماقبل همیشه یک درخت ریشه‌دار است که ریشه‌ی آن رأس مبدأ است.

لم ۲۴.۱۶

فرض کنید $G = (V, E)$ یک گراف جهت‌دار و وزن‌دار باشد، با تابع وزن $w : E \to \mathbb{R}$ و رأس مبدأ $s \in V$، و فرض کنید G هیچ دوری با وزن منفی ندارد که از s قابل دسترس باشد. آن گاه بعد از این که گراف با INITIALIZE-SINGLE-SOURCE(G, s) مقداردهی اولیه شد، زیرگراف عناصر ماقبل G_π یک درخت ریشه‌دار با ریشه‌ی s است، و هر دنباله‌ای از مراحل ترمیم روی یال‌های G این خصوصیت را به عنوان یک ثابت حفظ می‌کند.

اثبات در ابتدا تنها رأس در G_π رأس مبدأ است، و لم به صورت بدیهی درست است. یک زیرگراف عناصر ماقبل G_π را در نظر بگیرید که بعد از دنباله‌ای از مراحل ترمیم ساخته می‌شود. ابتدا اثبات خواهیم کرد که G_π بدون دور است. از طریق برهان خلف فرض کنید یک مرحله‌ی ترمیم یک دور در گراف G_π ایجاد می‌کند. اگر این دور $c = \langle v_0, v_1, ..., v_k \rangle$ باشد، و $v_k = v_0$، آن گاه

برای $i = ۱, ۲, ..., k$ داریم $v_i.\pi = v_{i-۱}$، و بدون از دست دادن کلیت می‌توانیم فرض کنیم که ترمیم یال $(v_{k-۱}, v_k)$ بوده است که دور را در G_π ساخته است.

ادعا می‌کنیم که تمام رأس‌ها بر روی دور c از رأس s قابل دسترس هستند. چرا؟ هر رأس روی c یک عنصر ماقبل غیر NIL دارد، و بنابراین وقتی به هر رأس روی c یک مقدار π غیر NIL داده شد، یک تخمین کوتاه‌ترین مسیر غیر بی‌نهایت هم نسبت داده شده است. طبق خصوصیت کران بالا هر رأس روی دور c یک وزن کوتاه‌ترین مسیر دارد، که ایجاب می‌کند این رأس از s قابل دسترس باشد.

کوتاه‌ترین مسیرها روی c را دقیقاً قبل از فراخوانی RELAX $(v_{k-۱}, v_k, w)$ بررسی می‌کنیم و نشان می‌دهیم که c یک دور با وزن منفی است، که با این فرض که G هیچ دوری با وزن منفی قابل دسترس از مبدأ ندارد در تناقض است. دقیقاً قبل از فراخوانی، برای $i = ۱, ۲, ..., k-۱$ داریم $v_i.d = v_{i-۱}.d$. بنابراین برای $i = ۱, ۲, ..., k-۱$، آخرین به هنگام‌سازی $v_i.d$ از طریق مقداردهی $v_i.d = v_{i-۱}.d + w(v_{i-۱}, v_i)$ بوده است. اگر $v_{i-۱}.d$ از آن موقع تغییر کرده باشد، این تغییر به صورت کاهش بوده است. از این رو دقیقاً قبل از فراخوانی RELAX $(v_{k-۱}, v_k, w)$ داریم

$$v_i.d \geq v_{i-۱}.d + w(v_{i-۱}, v_i) \qquad \text{برای هر } i = ۱, ۲, ..., k-۱ \qquad (۱۴-۲۴)$$

چون $v_k.\pi$ با این فراخوانی تغییر کرده است، دقیقاً قبل از آن نامساوی اکید

$$v_k.d > v_{k-۱}.d + w(v_{k-۱}, v_k)$$

را داریم. با جمع این نامساوی اکید با $k-۱$ نامساوی (۲۴-۱۴)، مجموع تخمین کوتاه‌ترین مسیرها روی دور c را به دست می‌آوریم:

$$\begin{aligned}\sum_{i=۱}^{k} v_i.d &= \sum_{i=۱}^{k} (v_{i-۱}.d + w(v_{i-۱}, v_i)) \\ &= \sum_{i=۱}^{k} v_{i-۱}.d + \sum_{i=۱}^{k} w(v_{i-۱}, v_i)\end{aligned}$$

ولی

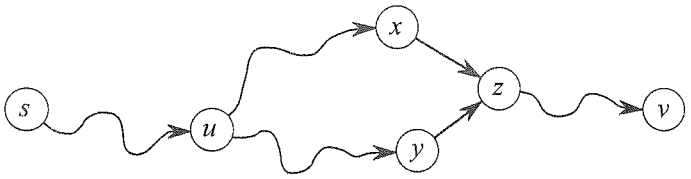
$$\sum_{i=۱}^{k} v_i.d = \sum_{i=۱}^{k} v_{i-۱}.d$$

چرا که هر رأس در دور c دقیقاً یک بار در هر مجموع ظاهر می‌شود. این نامساوی ایجاب می‌کند که

$$۰ > \sum_{i=۱}^{k} w(v_{i-۱}, v_i)$$

بنابراین این جمع وزن‌ها روی دور c منفی است، که تناقض مورد نظر را فراهم می‌کند.

اکنون اثبات کرده‌ایم که G_π یک گراف جهت‌دار بدون دور است. برای این که نشان دهیم این گراف، یک درخت با ریشه‌ی s است، کافی است (تمرین ب-۵-۲ را ببینید) که نشان دهیم برای هر رأس $v \in V_\pi$ یک مسیر یکتا از s به v در G_π وجود دارد.



شکل ۲۴-۹ اثبات این که مسیر از s به رأس v در G_π یکتا است. اگر دو مسیر $p_1(s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v)$ و $p_2(x \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v)$ وجود داشته باشد، که $x \neq y$، آن گاه $z.\pi = x$ و $z.\pi = y$، که تناقض است.

ابتدا باید نشان دهیم که برای هر رأس در V_π یک مسیر از s وجود دارد. رأس‌های درون V_π آن‌هایی هستند که مقدار π آن‌ها غیر NIL است، به علاوه‌ی s. در این جا ایده این است که به کمک استقرا اثبات کنیم که از s یک مسیر به تمام رأس‌ها در V_π وجود دارد. جزئیات به عنوان تمرین ۲۴-۵-۶ واگذار شده‌اند.

برای تکمیل اثبات لم، باید نشان دهیم که برای هر رأس $v \in V_\pi$ حداکثر یک مسیر از s به v در گراف G_π وجود دارد. فرض کنید این طور نباشد. یعنی همان طور که شکل ۲۴-۹ نشان می‌دهد، فرض کنید از s به یک رأس v دو مسیر ساده وجود دارد: p_1، که می‌توان آن را به صورت $s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v$ تجزیه کرد، و p_2، که می‌توان آن را به صورت $s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v$ تجزیه کرد، که در آن $x \neq y$ (ولی u می‌تواند s و z می‌تواند v باشد). ولی در این صورت $z.\pi = x$ و $z.\pi = y$، که این تناقض را به وجود می‌آورد که $x = y$. نتیجه می‌گیریم که در G_π از s به v یک مسیر یکتا وجود دارد، و بنابراین G_π یک درخت با ریشه‌ی s را تشکیل می‌دهد.

اکنون می‌توانیم نشان دهیم که بعد از این که دنباله‌ای از مراحل ترمیم را انجام دادیم، تمام رأس‌ها مقدار وزن کوتاه‌ترین مسیر واقعی خود را دریافت کرده‌اند و زیرگراف عناصر ماقبل G_π یک درخت کوتاه‌ترین مسیرها است. ▒

لم ۲۴-۱۷ (زیرساختار کوتاه‌ترین مسیرها در زیرگراف عناصر ماقبل)

فرض کنید $G = (V, E)$ یک گراف جهت‌دار و وزن‌دار باشد، با تابع وزن $w : E \rightarrow \mathbb{R}$ فرض کنید $G = (V, E)$ یک گراف جهت‌دار و وزن‌دار باشد با تابع وزن $w : E \rightarrow \mathbb{R}$ و مبدأ $s \in V$، و فرض کنید G هیچ دوری با وزن منفی ندارد که از s قابل دسترس باشد. فرض کنید INITIALIZE-SINGLE-SOURCE (G, s) را فراخوانی، و سپس یک دنباله از مراحل ترمیم را بر روی یال‌های G اجرا می‌کنیم، که برای هر $v \in V$ تولید می‌کند $v.d = \delta(s, v)$. آن گاه زیرگراف عناصر ماقبل G_π یک درخت کوتاه‌ترین مسیرها با ریشه‌ی s است.

اثبات باید اثبات کنیم که سه خصوصیت درخت‌های کوتاه‌ترین مسیر داده شده در مقدمه‌ی فصل ۲۴ برای G_π برقرار است. برای نشان دادن خصوصیت اول باید اثبات کنیم که V_π مجموعه‌ای از رأس‌های قابل دسترس از s است. طبق تعریف، یک وزن کوتاه‌ترین مسیر $\delta(s, v)$ غیر بی‌نهایت است اگر و

فقط اگر v از s قابل دسترس باشد، و بنابراین رأس‌های قابل دسترس از s دقیقاً رأس‌هـای بـا مقـدار $v.d$ کران‌دار هستند. ولی به یک رأس $v \in V - \{s\}$ یک مقدار غیر بی‌نهایت برای $v.d$ داده شده است اگر و فقط اگر $v.\pi \neq \text{NIL}$. بنابراین رأس‌های درون V_π دقیقاً آن‌هایی هستند که از s قابل دسترس‌اند.

خصوصیت دوم مستقیماً از لم ۲۴-۱۶ نتیجه می‌شود.

بنابراین، این می‌ماند که خصوصیت آخر درخت‌های کوتاه‌ترین مسیر را اثبات کنیم: برای هر رأس $v \in V_\pi$، مسیر ساده‌ی یکتای $s \overset{p}{\rightsquigarrow} v$ در G_π یک کوتاه‌ترین مسیر از s به v در G است. فـرض کنیـد $p = \langle v_0, v_1, \ldots, v_k \rangle$، که در آن $v_0 = s$ و $v_k = v$. برای هر $i = 1, 2, \ldots, k$ داریـم $v_i.d = \delta(s, v_i)$ و $w(v_{i-1}, v_i) \leq \delta(s, v_i) - \delta(s, v_{i-1})$ که از آن نتیجه مـی‌گیـریم $v_i.d \geq v_{i-1}.d + w(v_{i-1}, v_i)$، که از آن نتیجه مـی‌گیـریم $w(v_{i-1}, v_i)$.

جمع وزن‌ها روی مسیر p می‌دهد

$$\begin{aligned} w(p) &= \sum_{i=1}^{k} w(v_{i-1}, v_i) \\ &\leq \sum_{i=1}^{k} (\delta(s, v_i) - \delta(s, v_{i-1})) \\ &= \delta(s, v_k) - \delta(s, v_0) && \text{(چون سری خاصیت تلسکوپی دارد)} \\ &= \delta(s, v_k) && (\text{چون } \delta(s, v_0) = \delta(s, s) = 0) \end{aligned}$$

بنابراین $w(p) \leq \delta(s, v_k)$. چون $\delta(s, v_k)$ یک کران پایین بر روی وزن هر مسیری از s بـه v_k اسـت، نتیجه می‌گیریم که $w(p) = \delta(s, v_k)$، و بنابراین p یک کوتاه‌ترین مسیر از s به $v = v_k$ است.

تمرین‌ها

۲۴-۵-۱ دو درخت کوتاه‌ترین مسیر برای گراف جهت‌دار شکل ۲۴-۲، غیر از دو درختی که نشـان داده شده است ارائه کنید.

۲۴-۵-۲ یک مثال از یک گراف جهت‌دار و وزن‌دار $G = (V, E)$ با تابع وزن $w : E \to \mathbb{R}$ و رأس مبدأ s بدهید به طوری که G خصوصیت زیر را ارضا می‌کند: بـرای هـر یـال $(u, v) \in E$ یک درخت کوتاه‌ترین مسیر بـا ریشـه‌ی s وجـود دارد کـه حـاوی (u, v) اسـت، و یـک درخت کوتاه‌ترین مسیر با ریشه‌ی s وجود دارد که حاوی (u, v) نیست.

۲۴-۵-۳ لم ۲۴-۱۰ را طوری گسترش دهید که حالت‌هایی را که در آن‌ها وزن کوتاه‌تـرین مسـیرها ∞ یا $-\infty$ است را هم اداره کند.

۲۴-۵-۴ فرض کنید $G = (V, E)$ یک گـراف جهـت‌دار و وزن‌دار بـا رأس مبـدأ s باشـد، کـه بـا INITIALIZE-SINGLE-SOURCE (G, s) مقداردهی اولیه شده است. اثبات کنیـد کـه اگـر دنباله‌ای از مراحل ترمیم $s.\pi$ را برابر با یک مقدار غیر NIL قرار دهـد، آن گـاه G حـاوی یک دور با وزن منفی است.

۲۴-۵-۵ فرض کنید $G = (V, E)$ یک گراف جهت‌دار و وزن‌دار باشد که شامل هیچ یالی با وزن منفی نیست. فرض کنید s رأس مبدأ باشد، و فرض کنید اگر $v \in V - \{s\}$ از s قابل دسترس باشد، اجازه می‌دهیم که $v.\pi$ عنصر ماقبل v در هر کوتاه‌ترین مسیری از s به v باشد، و در غیر این صورت NIL خواهد بود. یک مثال از چنین گراف G و انتساب مقادیر π بدهید که یک دور در G_π می‌سازد. (طبق لم ۲۴-۱۶ چنین مقداردهی نمی‌تواند با دنباله‌ای از مراحل ترمیم انجام شود.)

۲۴-۵-۶ فرض کنید $G = (V, E)$ یک گراف جهت‌دار و وزن‌دار و تابع وزن $w : E \to \mathbb{R}$ باشد که هیچ دوری با وزن منفی ندارد. فرض کنید s رأس مبدأ باشد، و G با INITIALIZE-SINGLE-SOURCE (G, s) مقداردهی اولیه شده باشد. اثبات کنید که برای هر رأس $v \in V_\pi$ یک مسیر از s به v وجود دارد، و این خصوصیت به صورت یک ثابت در طول هر دنباله‌ای از مراحل ترمیم حفظ می‌شود.

۲۴-۵-۷ فرض کنید $G = (V, E)$ یک گراف جهت‌دار و وزن‌دار باشد که هیچ دوری با وزن منفی ندارد. اگر $s \in V$ رأس مبدأ باشد و G به وسیله‌ی INITIALIZE-SINGLE-SOURCE (G, s) مقداردهی اولیه شده باشد، اثبات کنید که دنباله‌ای از $|V| - 1$ مرحله‌ی ترمیم وجود دارد که برای تمام رأس‌های $v \in V$ مقادیر $v.d = \delta(s, v)$ را تولید می‌کند.

۲۴-۵-۸ فرض کنید G یک گراف جهت‌دار و وزن‌دار دلخواه با یک دور منفی قابل دسترس از رأس مبدأ s باشد. نشان دهید که همیشه می‌توان دنباله‌ای بی‌نهایت از ترمیم یال‌های G ساخت به طوری که تمام ترمیم‌ها باعث شوند که یکی از تخمین‌های کوتاه‌ترین مسیر تغییر کند.

مسائل

۲۴-۱ ***بهبود بن برای الگوریتم بلمن-فورد***

فرض کنید که ترمیم یال‌ها در هر یک از گذرهای الگوریتم بلمن-فورد را به صورت زیر مرتب می‌کنیم. قبل از اولین گذر، یک ترتیب خطی دلخواه $v_1, v_2, \ldots, v_{|V|}$ به رأس‌های گراف ورودی $G = (V, E)$ می‌دهیم. آن گاه مجموعه‌ی یال‌ها را به صورت $E_f \cup E_b$ تقسیم بندی می‌کنیم، که در آن $E_f = \{(v_i, v_j) \in E : i < j\}$ و $E_f = \{(v_i, v_j) \in E : i > j\}$. (فرض کنید G حاوی هیچ طوقه‌ای نیست، و از این رو تمام یال‌ها یا درون E_f هستند و یا درون E_b.) تعریف می‌کنیم $G_f = (V, E_f)$ و $G_b = (V, E_b)$.

اثبات کنید که G_f بدون دور است، با ترتیب توپولوژیکی $\langle v_1, v_2, \ldots, v_{|V|} \rangle$، و G_b بدون دور است با ترتیب توپولوژیکی $\langle v_{|V|}, v_{|V|-1}, \ldots, v_1 \rangle$.

مسائل

فرض کنید هر یک از گذرهای الگوریتم بلمن–فورد را به صورت زیر پیاده‌سازی می‌کنیم. هر یک از رأس‌ها را به ترتیب $v_1, \ldots, v_{|V|-1}, v_{|V|}$ ملاقات می‌کنیم، و در هر ملاقات یال‌های E_f را که از رأس خارج می‌شوند، ترمیم می‌کنیم. سپس هر رأس را به ترتیب $v_1, v_2, \ldots, v_{|V|}$ ملاقات می‌کنیم، و در هر ملاقات یال‌های E_b را که از رأس خارج می‌شوند، ترمیم می‌کنیم.

i. اثبات کنید که با این روش، اگر G حاوی هیچ دوری با وزن منفی نباشد که از رأس مبدأ s قابل دسترس باشند، آن گاه پس از فقط $\lceil |V|/2 \rceil$ گذر از روی یال‌ها، برای تمام رأس‌های $v \in V$ داریم $v.d = \delta(s, v)$.

ii. آیا این رویکرد زمان اجرای حدی الگوریتم بلمن–فورد را کاهش می‌دهد؟

۲۴-۲ جعبه‌های تودرتو

یک جعبه‌ی dبعدی با بعدهای $(x_1, x_2, \ldots, x_d)$ درون یک جعبه‌ی دیگر با بعدهای $(y_1, y_2, \ldots, y_d)$ *جای می‌گیرد* اگر یک جایگشت π روی $\{1, 2, \ldots, d\}$ وجود داشته باشد به طوری که $x_{\pi(1)} < y_1$، $x_{\pi(2)} < y_2$، ...، $x_{\pi(d)} < y_d$.

i. بحث کنید که رابطه‌ی جای‌گیری یک رابطه‌ی تراگذری است.

ii. یک متد کارامد ارائه کنید که تعیین می‌کند آیا یک جعبه‌ی dبعدی در یک جعبه‌ی دیگر جای می‌گیرد یا خیر.

iii. فرض کنید به شما مجموعه‌ای از n جعبه‌ی dبعدی $\{B_1, B_2, \ldots, B_n\}$ داده شده است. یک الگوریتم کارامد ارائه کنید که طولانی‌ترین دنباله‌ی $\langle B_{i_1}, B_{i_2}, \ldots, B_{i_k} \rangle$ از جعبه‌ها را تعیین می‌کند به طوری که برای $j = 1, 2, \ldots, k-1$ جعبه‌ی B_{i_j} درون جعبه‌ی $B_{i_{j+1}}$ جای می‌گیرد. زمان اجرای الگوریتم خود را برحسب n و d توصیف کنید.

۲۴-۳ سودآوری

سودآوری (arbitrage) عبارت است از استفاده از اختلافات در نرخ تبدیل واحدهای پول و تبدیل یک واحد پول به بیش از یک واحد از همان پول. مثلاً فرض کنید که ۱ دلار امریکا معادل است با ۴۹ روپیه‌ی هند، ۱ روپیه‌ی هند معادل است با ۲ ین ژاپن، و ۱ ین ژاپن معادل است با ۰٫۰۱۰۷ دلار امریکا. در این صورت یک معامله‌گر، با تبدیل واحدهای پول می‌تواند با ۱ دلار امریکا شروع کرده و $۱٫۰۴۸۶ = ۴۹ \times ۲ \times ۰٫۰۱۰۷$ دلار امریکا خریداری کند، و بدین صورت ۴٫۸۶ درصد سود کند.

فرض کنید که n واحد پول $c_1, c_2, \ldots, c_n$ و یک جدول R با اندازه‌ی $n \times n$ از نرخ‌های تبدیل به ما داده شده است، به طوری که یک واحد از واحد پول c_i معادل است با $R[i, j]$ واحد از پول c_j.

I. یک الگوریتم کارامد ارائه کنید که تعیین می‌کند آیا دنباله‌ی $\langle c_{i_1}, c_{i_2}, ..., c_{i_k} \rangle$ از واحدهای پول وجود دارد به طوری که

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdot ... \cdot R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1$$

زمان اجرای الگوریتم خود را تحلیل کنید.

II. یک الگوریتم کارامد برای چاپ چنین دنباله‌ای در صورت وجود ارائه کنید. زمان اجرای الگوریتم خود را تحلیل کنید.

۲۴-۴ الگوریتم مقیاس دهی گابو برای کوتاه‌ترین مسیرها از یک مبدأ

یک الگوریتم *مقیاس دهی* (scaling) یک مسئله را بدین صورت حل می‌کند که ابتدا فقط پرارزش‌ترین بیت ورودی مربوطه را در نظر می‌گیرد (مثلاً وزن یک یال). سپس جواب اولیه را با بررسی دومین بیت پر ارزش اصلاح می‌کند. این الگوریتم به همین ترتیب بیت‌های پر ارزش بیش‌تری را بررسی می‌کند، و هر بار جواب را اصلاح می‌کند، تا این که تمام بیت‌ها بررسی شوند و جواب درست به دست آید.

در این مسئله یک الگوریتم برای محاسبه‌ی کوتاه‌ترین مسیرها از یک مبدأ را بررسی می‌کنیم که از مقیاس دهی وزن یال‌ها استفاده می‌کند. به ما یک گراف جهت‌دار $G = (V, E)$ با تابع وزن w داده شده است، که وزن‌های آن اعداد صحیح نامنفی هستند. فرض کنید $W = \max_{(u,v) \in E} \{w(u, v)\}$. هدف ما این است که یک الگوریتم طراحی کنیم که در زمان $O(E \lg W)$ اجرا می‌شود. فرض می‌کنیم تمام رأس‌ها از مبدأ قابل دسترس هستند.

الگوریتم بیت‌ها را در نمایش دودویی وزن یال‌ها بررسی می‌کند، به ترتیب از پرارزش‌ترین به کم‌ارزش‌ترین. به طور خاص فرض کنید $k = \lceil \lg(W + 1) \rceil$ تعداد بیت‌ها در نمایش دودویی W باشد، و برای $i = 1, 2, ..., k$ داشته باشیم $w_i(u, v) = \lfloor w(u, v) / 2^{k-i} \rfloor$. یعنی $w_i(u, v)$ نسخه‌ی «کاهش مقیاس» داده شده‌ی $w(u, v)$ است که از i بیت پر ارزش $w(u, v)$ به دست می‌آید. (بنابراین برای هر $(u, v) \in E$ داریم $w_k(u, v) = w(u, v)$.) به عنوان مثال اگر $k = 5$ و $w(u, v) = 25$، که نمایش دودویی آن به صورت $\langle 11001 \rangle$ است، آن گاه $w_3(u, v) = \langle 110 \rangle = 6$. به عنوان یک مثال دیگر با $k = 5$، اگر $w(u, v) = \langle 00100 \rangle = 4$، آن گاه $w_3(u, v) = \langle 001 \rangle = 1$. اجازه دهید $\delta_i(u, v)$ را به صورت وزن کوتاه‌ترین مسیر از رأس u به رأس v با استفاده از تابع وزن w_i تعریف کنیم. بنابراین برای هر $u, v \in V$ داریم $\delta_k(u, v) = \delta(u, v)$. برای یک رأس مبدأ داده شده‌ی s، الگوریتم مقیاس دهی ابتدا $\delta_1(s, v)$ (وزن کوتاه‌ترین مسیرها) را برای $v \in V$ محاسبه می‌کند، سپس $\delta_2(s, v)$ را برای $v \in V$ محاسبه می‌کند، و همین طور تا جایی که به $\delta_k(s, v)$ برسد. فرض می‌کنیم که $|E| \geq |V| - 1$، و به زودی خواهیم دید که محاسبه‌ی δ_i از δ_{i-1} به زمان $O(E)$ نیاز دارد، و بنابراین کل الگوریتم $O(kE) = O(E \lg W)$ زمان خواهد برد.

I. فرض کنید برای تمام رأس‌های $v \in V$ داریم $\delta(s,v) \le |E|$. نشان دهید که می‌توانیم برای هر $v \in V$، مقدار $\delta(s,v)$ را در زمان $O(E)$ محاسبه کنیم.

II. نشان دهید که می‌توان $\delta_1(s,v)$ را برای تمام $v \in V$ در زمان $O(E)$ محاسبه کرد.

اجازه دهید بر روی محاسبه‌ی δ_i از روی δ_{i-1} تمرکز کنیم.

III. اثبات کنید که برای $i = 2, 3, ..., k$، داریم $w_i(u,v) = 2w_{i-1}(u,v)$ یا $w_i(u,v) = 2w_{i-1}(u,v)+1$. سپس اثبات کنید که

$$2\delta_{i-1}(s,v) \le \delta_i(s,v) \le 2\delta_{i-1}(s,v) + |V| - 1$$

برای هر $v \in V$.

IV. برای $i = 1, 2, ..., k$ و هر $(u,v) \in E$ تعریف می‌کنیم

$$\widehat{w_i}(u,v) = w_i(u,v) + 2\delta_{i-1}(s,u) - 2\delta_{i-1}(s,v).$$

اثبات کنید که برای $i = 2, 3, ..., k$ و هر $u, v \in V$، مقدار «وزن دهی مجدد» $\widehat{w_i}(u,v)$ مربوط به یال (u,v) یک عدد صحیح نامنفی است.

V. اکنون $\widehat{\delta_i}(s,v)$ را به صورت وزن کوتاه‌ترین مسیر از s به v با استفاده از تابع وزن $\widehat{w_i}$ تعریف می‌کنیم. اثبات کنید که برای $i = 2, 3, ..., k$ و هر $v \in V$،

$$\delta_i(s,v) = \widehat{\delta_i}(s,v) + 2\delta_{i-1}(s,v)$$

و

$$\widehat{\delta_i}(s,v) \le |E|$$

VI. نشان دهید که چگونه می‌توان برای هر $v \in V$، مقدار $\delta_i(s,v)$ را از $\delta_{i-1}(s,v)$ در زمان $O(E)$ محاسبه کرد، و نتیجه بگیرید که می‌توان $\delta(s,v)$ را برای تمام $v \in V$ در زمان $O(E \lg W)$ محاسبه کرد.

۲۴-۵ الگوریتم کارپ برای دور با کم‌ترین میانگین وزن

فرض کنید $G = (V,E)$ یک گراف جهت‌دار با تابع وزن $w : E \to \mathbb{R}$ باشد، و $n = |V|$. *وزن میانگین* (mean weight) یک دور $c = \langle e_1, e_2, ..., e_k \rangle$ از یال‌های E را به صورت

$$\mu(c) = \frac{1}{k}\sum_{i=1}^{k} w(e_i)$$

تعریف می‌کنیم. فرض کنید $\mu^* = \min_c \mu(c)$، که دامنه‌ی c تمام دورهای جهت‌دار G است. یک دور c که برای آن داریم $\mu(c) = \mu^*$ یک *دور با میانگین وزن کمینه* نام دارد. در این مسئله الگوریتمی برای محاسبه‌ی بهینه‌ی μ^* طراحی می‌شود.

بدون از دست دادن کلیت فرض کنید هر رأس $v \in V$ از یک رأس مبدأ $s \in V$ قابل دسترس است. فرض کنید $\delta(s,v)$ وزن کوتاه‌ترین مسیر از s به v باشد، و $\delta_k(s,v)$ وزن

یک کوتاه‌ترین مسیر از s به v شامل *دقیقاً* k یال. اگر هیچ مسیری از s به v با k یال وجود نداشته باشد، آن گاه $\delta_k(s,v)=\infty$.

I نشان دهید که اگر $\mu^*=0$، آن گاه G هیچ دوری با وزن منفی ندارد، و $\delta(s,v)=\min_{0\le k\le n-1}\delta_k(s,v)$ برای تمام رأس‌های $v\in V$.

II نشان دهید که اگر $\mu^*=0$، آن گاه

$$\max_{0\le k\le n-1}\frac{\delta_n(s,v)-\delta_k(s,v)}{n-k}\ge 0$$

برای تمام رأس‌های $v\in V$. (*راهنمایی:* از هر دو خصوصیت بخش I استفاده کنید.)

III فرض کنید c یک دور با وزن 0 باشد، و u و v رأس‌هایی بر روی c. فرض کنید $\mu^*=0$ و وزن مسیر از u به v روی دور c برابر x است. اثبات کنید که $\delta(s,v)=\delta(s,u)+x$. (*راهنمایی:* وزن مسیر از v به u روی دور $-x$ است.)

IV نشان دهید که اگر $\mu^*=0$، آن گاه روی هر دور با میانگین وزن کمینه یک رأس v وجود دارد به طوری که

$$\max_{0\le k\le n-1}\frac{\delta_n(s,v)-\delta_k(s,v)}{n-k}=0$$

(*راهنمایی:* نشان دهید که یک کوتاه‌ترین مسیر به هر رأسی روی دور با میانگین وزن کمینه می‌تواند روی دور طوری گسترش یابد که یک کوتاه‌ترین مسیر به رأس بعدی روی دور بسازد.)

V نشان دهید که اگر $\mu^*=0$، آن گاه

$$\min_{v\in V}\max_{0\le k\le n-1}\frac{\delta_n(s,v)-\delta_k(s,v)}{n-k}=0$$

VI نشان دهید که اگر یک ثابت t به وزن هر یال در G اضافه کنیم، آن گاه μ^* به اندازه‌ی t افزایش می‌یابد. از این مسئله استفاده کنید تا نشان دهید

$$\mu^*=\min_{v\in V}\max_{0\le k\le n-1}\frac{\delta_n(s,v)-\delta_k(s,v)}{n-k}$$

VII یک الگوریتم با زمان $O(VE)$ برای محاسبه‌ی μ^* ارائه کنید.

۲۴-۶ کوتاه‌ترین مسیرهای دوجهته

یک دنباله دوجته است اگر ابتدا به صورت یکنواخت افزایش یابد و سپس به صورت یکنواخت کاهش یابد، یا این که بتوان آن را به صورت دایره‌ای حرکت داد تا این خصوصیت را پیدا کند. به عنوان مثال دنباله‌های $\langle 2,-3,8,6,4,1\rangle$، $\langle 5,-10,-4,2,9\rangle$، و $\langle 4,3,2,1\rangle$ دوجهته هستند، ولی $\langle 10,2,4,12,3,1\rangle$ دوجهته نیست. (فصل ۲۷ را برای

بحثی در مورد مرتب‌سازهای دوجهته، و مسئله‌ی ۱۵-۱ را برای مسئله‌ی فروشنده‌ی دوره گرد اقلیدسی ببینید.)

فرض کنید به ما یک گراف جهت‌دار $G = (V, E)$ با تابع وزن $w : E \to \mathbb{R}$ داده شده است که در آن وزن تمام یال‌ها یکتا است، و می‌خواهیم کوتاه‌ترین مسیرها را از یک مبدأ s در آن بیابیم. به ما اطلاعات دیگری هم داده شده است: برای هر رأس $v \in V$ وزن یال‌ها روی هر کوتاه‌ترین مسیری از s به v یک دنباله‌ی دوجهته است.

بهینه‌ترین الگوریتمی را که می‌توانید برای حل این مسئله ارائه، و سپس زمان اجرای آن را تحلیل کنید.

فصل

۲۵

کوتاه‌ترین مسیر بین هر دو رأس

۲۵-۰ مقدمه

در این فصل مسئله‌ی یافتن کوتاه‌ترین مسیرها بین هر جفت رأس در یک گراف را بررسی می‌کنیم. این مسئله ممکن است در ساختن یک جدول از فاصله‌ها بین تمام شهرها در یک نقشه‌ی جاده‌ها پیش بیاید. مانند فصل ۲۴، به ما یک گراف جهت‌دار و وزن‌دار با یک تابع وزن $w : E \to \mathbb{R}$ داده شده است، که یال‌ها را به یک تابع حقیقی مقدار نگاشت می‌کند. می‌خواهیم برای هر دو رأس $u, v \in V$ یک کوتاه‌ترین مسیر (مسیر با کم‌ترین وزن) از u به v بیابیم، که در آن وزن یک مسیر برابر است با مجموع وزن یال‌های تشکیل دهنده‌ی آن. معمولاً خروجی را به صورت جدول نیاز داریم: ورودی ردیف u و ستون v باید نشان‌دهنده‌ی کوتاه‌ترین مسیر از u به v باشد.

مسئله‌ی کوتاه‌ترین مسیرها بین هر دو رأس را می‌توان با اجرای یک الگوریتم کوتاه‌ترین مسیرها از یک مبدأ به تعداد $|V|$ بار حل کرد، هر بار با یک رأس به عنوان مبدأ. اگر وزن تمام یال‌ها نامنفی باشد می‌توانیم از الگوریتم Dijkstra استفاده کنیم. اگر صف اولویت کمینه را با یک آرایه‌ی خطی پیاده‌سازی کنیم، زمان اجرا $O(V^3 + VE) = O(V^3)$ خواهد بود. پیاده‌سازی هرم کمینه‌ی دودویی برای صف اولویت کمینه به زمان اجرای $O(VE \lg V)$ منجر می‌شود، که در صورتی که گراف خلوت باشد یک پیشرفت محسوب خواهد شد. در عوض می‌توانیم صف اولویت کمینه را با یک هرم فیبوناچی پیاده‌سازی کنیم، که در این صورت زمان اجرا $O(V^2 \lg V + VE)$ خواهد بود.

اگر یال‌هایی با وزن منفی داشته باشیم، دیگر نمی‌توان از الگوریتم Dijkstra استفاده کرد. در عوض باید الگوریتم کندتر بلمن-فورد را برای هر رأس اجرا کنیم. زمان اجرای حاصل $O(V^2E)$ است، که بر روی یک گراف شلوغ معادل است با $O(V^4)$. در این فصل خواهیم دید که چگونه می‌توان به شکلی بهتر این کار را انجام داد.

همچنین رابطه‌ی میان مسئله‌ی کوتاه‌ترین مسیرها بین هر دو رأس و ضرب زنجیره‌ی ماتریس‌ها را خواهیم دید و ساختار جبری آن را خواهیم آموخت.

برخلاف الگوریتم‌های کوتاه‌ترین مسیرها از یک مبدأ، که نمایش لیست مجاورت گراف را در نظر می‌گیرند، اکثر الگوریتم‌های این فصل از نمایش ماتریس مجاورت استفاده می‌کنند. (الگوریتم جانسون برای گراف‌های خلوت در بخش ۲۵-۳ از لیست مجاورت استفاده می‌کند.) برای سادگی فرض خواهیم کرد رأس‌ها به صورت $1, 2, \ldots, |V|$ شماره گذاری شده‌اند، به طوری که ورودی یک ماتریس W با اندازه‌ی $n \times n$ است که وزن یال‌های یک گراف جهت‌دار n رأسی $G = (V, E)$ را نشان می‌دهد. یعنی $W = (w_{ij})$، که در آن

$$w_{ij} = \begin{cases} 0 & \text{اگر } i = j \\ \text{وزن یال جهت‌دار } (i,j) & \text{اگر } i \neq j \text{ و } (i,j) \in E \\ \infty & \text{اگر } i \neq j \text{ و } (i,j) \notin E \end{cases} \qquad (۱-۲۵)$$

وجود یال‌های با وزن منفی بلامانع است، ولی برای صرفه‌جویی در زمان فرض خواهیم کرد که گراف ورودی هیچ دوری با وزن منفی ندارد.

خروجی جدولی الگوریتم‌های کوتاه‌ترین مسیرها میان هر دو رأس ارائه شده در این فصل، یک ماتریس $D = (d_{ij})$ با اندازه‌ی $n \times n$ است، که در آن ورودی d_{ij} حاوی وزن یک کوتاه‌ترین مسیر از رأس i به رأس j است. یعنی اگر $\delta(i, j)$ نشان‌دهنده‌ی وزن کوتاه‌ترین مسیر از رأس i به رأس j باشد (مانند فصل ۲۴)، آن گاه در پایان الگوریتم داریم $d_{ij} = \delta(i, j)$.

برای حل مسئله‌ی کوتاه‌ترین مسیرها میان هر دو رأس بر روی یک ماتریس مجاورت ورودی، ما نه تنها باید وزن کوتاه‌ترین مسیرها، بلکه *ماتریس عناصر ماقبل* $\Pi = (\pi_{ij})$ را هم باید بیابیم، که در آن π_{ij} برابر با NIL است اگر یا $i = j$ و یا هیچ مسیری از i به j وجود نداشته باشد، و در غیر این صورت π_{ij} عنصر ماقبل j روی یک کوتاه‌ترین مسیر از i است. درست همان طور که زیرگراف عناصر ماقبل G_π از فصل ۲۴ یک درخت کوتاه‌ترین مسیر برای یک رأس مبدأ داده شده است، زیرگراف ساخته شده با iامین ردیف از ماتریس Π باید یک درخت کوتاه‌ترین مسیرها با ریشه‌ی i باشد. برای هر رأس $i \in V$، *زیرگراف عناصر ماقبل* G برای i را به صورت $G_{\pi, i} = (V_{\pi, i}, E_{\pi, i})$ تعریف می‌کنیم، که در آن

$$V_{\pi, i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\}$$

و

$$E_{\pi, i} = \{(\pi_{ij}, j) : j \in V_{\pi, i} - \{i\}\}$$

اگر $G_{\pi, i}$ یک درخت کوتاه‌ترین مسیرها باشد، آن گاه رویه‌ی زیر، که نسخه‌ی اصلاح شده‌ی رویه‌ی PRINT-PATH از فصل ۲۲ است، یک کوتاه‌ترین مسیر از رأس i به رأس j را چاپ می‌کند.

```
PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)
1  if i == j
```

```
2       print i
3   elseif π_ij == NIL
4       print "no path from" i "to" j "exists"
5   else PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_ij)
6       print j
```

برای نمایان شدن خصوصیات اصلی الگوریتم‌های کوتاه‌ترین مسیرها بین هر دو رأس در این فصل، در مورد ساختن و خصوصیات ماتریس‌های عناصر ماقبل با جزئیاتی که در فصل قبل در مورد زیرگراف‌های عناصر ماقبل بحث کردیم، صحبت نخواهیم کرد. نکات اصلی در بعضی از تمرین‌ها پوشش داده شده است.

رئوس مطالب فصل

فصل ۲۵-۱ یک الگوریتم برنامه‌ریزی پویا بر پایه‌ی ضرب ماتریس‌ها برای حل مسئله‌ی کوتاه‌ترین مسیرها بین هر دو رأس ارائه می‌کند. با استفاده از تکنیک «مربع‌گیری مکرر» (repeated squaring)، می‌توان این الگوریتم را طوری پیاده‌سازی کرد که در زمان $\theta(V^3 \lg V)$ اجرا شود. یک الگوریتم برنامه‌ریزی پویای دیگر، الگوریتم فلوید- وارشال، در بخش ۲۵-۲ داده شده است. الگوریتم فلوید- وارشال در زمان $\theta(V^3)$ اجرا می‌شود. در بخش ۲۵-۲ مسئله‌ی یافتن بستار تراگذار یک گراف جهت‌دار هم پوشش داده می‌شود، که به مسئله‌ی کوتاه‌ترین مسیرها بین هر دو رأس مربوط است. نهایتاً در بخش ۲۵-۳ الگوریتم جانسون معرفی می‌شود که مسئله‌ی کوتاه‌ترین مسیرها بین هر دو رأس را در زمان $O(V^2 \lg V + VE)$ حل می‌کند، و برای گراف‌های بزرگ و خلوت بسیار مناسب است.

قبل از ادامه باید چند قرارداد در مورد نمایش‌های ماتریس مجاورت انجام دهیم. اول، به طور کلی فرض می‌کنیم که گراف ورودی n رأس دارد، یعنی $n = |V|$. دوم، به طور قراردادی ماتریس‌ها را با حروف بزرگ نشان می‌دهیم، مانند W، L، یا D، و عناصر آن‌ها را با اندیس‌های حروف کوچک نشان می‌دهیم، مانند w_{ij}، l_{ij}، یا d_{ij}. بعضی از ماتریس‌ها دارای اندیس‌هایی درون پرانتز هستند، مانند $L^{(m)} = (l_{ij}^{(m)})$ یا $D^{(m)} = (d_{ij}^{(m)})$، که این نشان دهنده‌ی تکرار است. نهایتاً، برای یک ماتریس داده شده‌ی A با اندازه‌ی $n \times n$ فرض خواهیم کرد که مقدار n در خصیصه‌ی $A.rows$ ذخیره شده است.

۲۵-۱ کوتاه‌ترین مسیرها و ضرب ماتریس‌ها

در این بخش یک الگوریتم برنامه‌ریزی پویا برای مسئله‌ی کوتاه‌ترین مسیرها بین هر دو رأس بر روی یک گراف جهت‌دار $G = (V, E)$ ارائه می‌شود. هر حلقه‌ی اصلی از برنامه‌ی پویا یک عملیات را فراخوانی خواهد کرد که بسیار مشابه ضرب ماتریسی است، به طوری که الگوریتم شبیه ضرب مکرر ماتریسی خواهد بود. با طراحی یک الگوریتم با زمان $\theta(V^4)$ برای مسئله‌ی کوتاه‌ترین مسیرها بین هر دو رأس آغاز خواهیم کرد، و سپس زمان اجرای آن را به $\theta(V^3 \lg V)$ بهبود خواهیم داد.

قبل از ادامه اجازه دهید مختصراً مراحل توصیف شده در فصل ۱۵ را برای طراحی یک الگوریتم برنامه‌ریزی پویا مرور کنیم.

۱. تعیین ساختار یک جواب بهینه.
۲. تعیین مقدار یک جواب بهینه به صورت بازگشتی.
۳. محاسبه‌ی مقدار یک جواب بهینه به صورت از پایین به بالا.

مرحله‌ی چهارم – ساختن یک جواب بهینه از اطلاعات محاسبه شده – به عنوان تمرین واگذار شده است.)

ساختار یک کوتاه‌ترین مسیر

با تعیین ساختار یک جواب بهینه آغاز می‌کنیم. برای مسئله‌ی کوتاه‌ترین مسیرها بین هر دو رأس روی یک گراف $G = (V, E)$، قبلاً اثبات کردیم (لم ۲۴-۱) که تمام زیرمسیرهای یک کوتاه‌ترین مسیر، خود کوتاه‌ترین مسیر هستند. فرض کنید گراف به صورت یک ماتریس مجاورت $W = (w_{ij})$ نمایش داده شده است. یک کوتاه‌ترین مسیر p را از رأس i به رأس j در نظر بگیرید، و فرض کنید p حداکثر m یال دارد. با فرض این که هیچ دور با وزن منفی در گراف وجود ندارد، m کران‌دار است. اگر $i = j$، آن گاه وزن p برابر ۰ است و هیچ یالی در آن وجود ندارد. اگر رأس‌های i و j مجزا باشند، آن گاه مسیر p را به $i \overset{p'}{\rightsquigarrow} k \to j$ تجزیه می‌کنیم، که اکنون p' حداکثر $m-1$ یال دارد. طبق لم ۲۴-۱ مسیر p' یک کوتاه‌ترین مسیر از i به k است، و بنابراین داریم $\delta(i, j) = \delta(i, k) + w_{kj}$.

یک جواب بازگشتی به مسئله‌ی کوتاه‌ترین مسیرها بین هر دو رأس

اکنون فرض کنید $l_{ij}^{(m)}$ وزن کمینه‌ی هر مسیری از رأس i به رأس j باشد که حداکثر m یال دارد. وقتی $m = ۰$، یک کوتاه‌ترین مسیر از i به j با ۰ یال وجود دارد اگر و فقط اگر $i = j$. بنابراین،

$$l_{ij}^{(۰)} = \begin{cases} ۰ & \text{اگر } i = j \\ \infty & \text{اگر } i \neq j \end{cases}$$

برای $m \geq ۱$، $l_{ij}^{(m)}$ را به صورت کمینه‌ی $l_{ij}^{(m-۱)}$ (وزن کوتاه‌ترین مسیر از i به j شامل حداکثر $m-۱$ یال) و کمینه‌ی وزن هر مسیری از i به j شامل حداکثر m یال محاسبه می‌کنیم، که با مشاهده‌ی تمام عناصر ماقبل ممکن k برای j به دست می‌آید. بنابراین به صورت بازگشتی تعریف می‌کنیم

$$\begin{aligned} l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-۱)}, \min\{l_{ik}^{(m-۱)} + w_{kj}\}\right) \\ &= \min_{۱ \leq k \leq n} \{l_{ik}^{(m-۱)} + w_{kj}\} \end{aligned} \qquad (۲-۲۵)$$

تساوی دوم از این جا نتیجه می‌شود که برای تمام jها داریم $w_{jj} = ۰$.

وزن واقعی کوتاه‌ترین مسیرهای $\delta(i, j)$ چقدر است؟ اگر گراف حاوی هیچ دوری با وزن منفی نباشد، آن گاه برای هر جفت از رأس‌های i و j که برای آن‌ها داریم $\delta(i, j) < \infty$، یک کوتاه‌ترین مسیر از i به j وجود دارد که ساده است، و بنابراین حداکثر $n-۱$ یال دارد. یک مسیر از رأس i به رأس j که بیش از $n-۱$ یال داشته باشد نمی‌تواند وزنی کم‌تر از وزن کوتاه‌ترین مسیر از i به j

داشته باشد. بنابراین وزن واقعی کوتاه‌ترین مسیرها به صورت

$$\delta(i, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \ldots \qquad (۳-۲۵)$$

است.

محاسبه‌ی وزن کوتاه‌ترین مسیرها به صورت از پایین به بالا

با در نظر گرفتن ماتریس $W = (w_{ij})$ به عنوان ورودی، اکنون یک سری از ماتریس‌های $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$ را محاسبه می‌کنیم، که در آن برای $m = 1, 2, \ldots, n-1$ داریم $L^{(m)} = (l_{ij}^{(m)})$. ماتریس نهایی $L^{(n-1)}$ حاوی وزن واقعی کوتاه‌ترین مسیرها است. مشاهده کنید که برای تمام رأس‌های $i, j \in V$ داریم $l_{ij}^{(1)} = w_{ij}$، و بنابراین $L^{(1)} = W$.

قلب الگوریتم رویه‌ی زیر است، که با دریافت ماتریس‌های $L^{(m-1)}$ و W، ماتریس $L^{(m)}$ را بازمی‌گرداند. یعنی کوتاه‌ترین مسیرهای محاسبه شده تا به این جا را به اندازه‌ی یک پال دیگر گسترش می‌دهد.

```
EXTEND-SHORTEST-PATHS(L, W)
1  n = L.rows
2  let L' = (l'_ij) be an n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          l'_ij = ∞
6          for k = 1 to n
7              l'_ij = min(l'_ij, l_ik + w_kj)
8  return L'
```

رویه یک ماتریس $L' = (l'_{ij})$ را محاسبه کرده و در پایان آن را بازمی‌گرداند. این رویه این کار را با محاسبه‌ی تساوی (۲۵-۲) برای تمام iها و jها، با استفاده از L برای $L^{(m-1)}$ و L' برای $L^{(m)}$ انجام می‌دهد. (این اسامی بدون اندیس نوشته شده‌اند تا ماتریس‌های ورودی و خروجی را از m مستقل کنند.) زمان اجرای این رویه $\theta(n^3)$ است، به دلیل سه حلقه‌ی for تودرتو.

اکنون می‌توانیم رابطه‌ی ضرب ماتریسی با کوتاه‌ترین مسیر را ببینیم. فرض کنید می‌خواهیم ضرب ماتریسی $C = A \cdot B$ را برای دو ماتریس A و B با اندازه‌ی $n \times n$ محاسبه کنیم. در این صورت برای $i, j = 1, 2, \ldots, n$ باید

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \qquad (۴-۲۵)$$

را محاسبه کنیم. مشاهده کنید که اگر جایگزینی‌های

$$\begin{aligned} l^{(m-1)} &\to a, \\ w &\to b, \\ l^{(m)} &\to c, \\ \min &\to +, \\ + &\to \cdot \end{aligned}$$

را در تساوی (۲۵-۲) انجام دهیم، به تساوی (۲۵-۴) می‌رسیم. بنابراین اگر این تغییرات را بر روی EXTEND-SHORTEST-PATHS انجام دهیم، و همچنین ∞ (عنصر همانی برای min) را با ۰ (عنصر همانی برای +) جایگزین کنیم، به الگوریتم سرراست با زمان $\theta(n^3)$ برای ضرب ماتریس‌های مربعی که در بخش ۴-۲ دیدیم، می‌رسیم:

```
MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be an n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

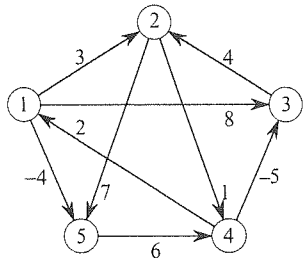
با بازگشت به مسئله‌ی کوتاه‌ترین مسیرها بین هر دو رأس، وزن کوتاه‌ترین مسیرها را با گسترش یکی یکی کوتاه‌ترین مسیرها روی یال‌ها محاسبه می‌کنیم. با فرض این که $A \cdot B$ نشان دهنده‌ی «ضرب» ماتریسی بازگردانده شده توسط EXTEND-SHORTEST-PAHTS (A, B) باشد، دنباله‌ای از $n-1$ ماتریس را به صورت زیر محاسبه می‌کنیم:

$$
\begin{aligned}
L^{(1)} &= L^{(0)} \cdot W = W \\
L^{(2)} &= L^{(1)} \cdot W = W^2 \\
L^{(3)} &= L^{(1)} \cdot W = W^3 \\
&\vdots \\
L^{(n-1)} &= L^{(n-2)} \cdot W = W^{n-1}
\end{aligned}
$$

همان طور که در بالا بحث کردیم، ماتریس $L^{(n-1)} = W^{n-1}$ حاوی وزن کوتاه‌ترین مسیرها است. رویه‌ی زیر این دنباله را در زمان $\theta(n^4)$ محاسبه می‌کند.

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
1  n = W.rows
2  L^(0) = W
3  for m = 2 to n − 1
4      let L^(m) be a new n × n matrix
5      L^(m) = EXTEND-SHORTEST-PATHS(L^(m-1), W)
6  return L^(n-1)
```

شکل ۲۵-۱ یک گراف و ماتریس $L^{(m)}$ محاسبه شده توسط رویه‌ی SLOW-ALL-PAIRS-SHORTEST-PAHTS را نشان می‌دهد.



$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

شکل ۲۵-۱ یک گراف جهت‌دار و دنباله‌ی ماتریس‌های $L^{(m)}$ محاسبه شده توسط SLOW-ALL-PAIRS-SHORTEST-PATHS. خواننده می‌تواند چک کند که $L^{(5)} = L^{(4)}W$ معادل است با $L^{(4)}$، و بنابراین برای $m \geq 4$ داریم $L^{(m)} = L^{(4)}$.

بهبود زمان اجرا

با این حال هدف ما این نیست که *تمام* ماتریس‌های $L^{(m)}$ را محاسبه کنیم: تنها ماتریسی که برای ما مهم است $L^{(n-1)}$ است. به یاد بیاورید که در غیاب دورهای با وزن منفی، تساوی (۲۵-۳) برای تمام اعداد صحیح $m \geq n-1$ ایجاب می‌کند $L^{(m)} = L^{(n-1)}$. درست مانند یک ضرب ماتریسی معمولی که شرکت‌پذیر است، ضرب ماتریسی تعریف شده توسط رویه‌ی EXTEND-SHORTEST-PATHS هم شرکت‌پذیر است (تمرین ۲۵-۱-۴ را ببینید). بنابراین می‌توانیم با محاسبه‌ی دنباله‌ی

$$\begin{aligned} L^{(1)} &= W \\ L^{(2)} &= W^2 = W \cdot W \\ L^{(4)} &= W^4 = W^2 \cdot W^2 \\ L^{(8)} &= W^8 = W^4 \cdot W^4 \\ &\vdots \\ L^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} = W^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot W^{2^{\lceil \lg(n-1) \rceil - 1}} \end{aligned}$$

مقدار $L^{(n-1)}$ را فقط با $\lceil \lg(n-1) \rceil$ ضرب ماتریسی محاسبه کنیم. از آن جایی که $2^{\lceil \lg(n-1) \rceil} \geq n-1$،

حاصل ضرب نهایی $L^{(2^{\lceil \lg(n-1) \rceil})}$ برابر است با $L^{(n-1)}$.

رویه‌ی زیر دنباله‌ی بالا از ضرب ماتریس‌ها را با استفاده از تکنیک مربع گیری مکرر محاسبه می‌کند.

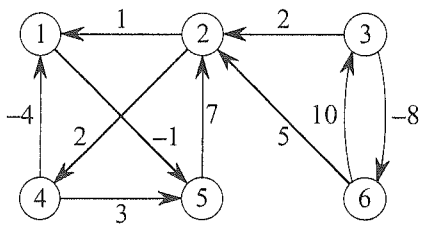
```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
1  n = W.rows
2  L^(1) = W
3  m = 1
4  while m < n − 1
5      let L^(2m) be a new n × n matrix
6      L^(2m) = EXTEND-SHORTEST-PATHS(L^(m), L^(m))
7      m = 2m
8  return L^(m)
```

در هر بار تکرار حلقه‌ی while خطوط ۴-۷، $L^{(2m)} = (L^{(m)})^2$ را محاسبه می‌کنیم، با شروع از $m = 1$. در پایان هر تکرار مقدار m را دو برابر می‌کنیم. تکرار پایانی با محاسبه‌ی $L^{(2m)}$ برای $2m \le 2n - 2$، $n - 1 \le 2m$، مقدار $L^{(n-1)}$ را محاسبه می‌کند. طبق تساوی (۲۵-۳) داریم $L^{(2m)} = L^{(n-1)}$. دفعه‌ی بعدی که تست در خط ۴ انجام می‌شود m دو برابر شده است، و بنابراین اکنون $m \ge n - 1$، و نتیجه‌ی تست منفی است و رویه آخرین ماتریس محاسبه شده را بازمی‌گرداند.

زمان اجرای الگوریتم FASTER-ALL-PAIR-SHORTEST-PATHS برابر $\theta(n^3 \lg n)$ است، چرا که هر یک از $\lceil \lg(n-1) \rceil$ ضرب ماتریسی به $\theta(n^3)$ زمان نیاز دارند. توجه کنید که کد بسیار کوچک است و در آن هیچ ساختمان داده‌ی پیچیده‌ای وجود ندارد، و بنابراین ثابت‌های مخفی در نماد θ کوچک هستند.

تمرین‌ها

۲۵-۱-۱ رویه‌ی SLOW-PAIRS-SHORTEST-PATHS را بر روی گراف جهت‌دار و وزن‌دار شکل ۲۵-۲ اجرا کرده و ماتریس‌های حاصل را در هر تکرار حلقه نشان دهید. سپس همین کار را برای FASTER-ALL-PAIRS-SORTEST-PATHS انجام دهید.



شکل ۲۵-۲ یک گراف جهت‌دار و وزن‌دار برای استفاده در تمرین‌های ۲۵-۱-۱، ۲۵-۲-۱، و ۲۵-۳-۱.

۲۵-۱-۲ چرا برای هر $1 \le i \le n$ نیاز داریم رابطه‌ی $w_{ii} = 0$ برقرار باشد؟

۲۵-۱-۳ ماتریس

$$L^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \cdots & \infty \\ \infty & 0 & \infty & \cdots & \infty \\ \infty & \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \cdots & 0 \end{pmatrix}$$

استفاده شده در الگوریتم کوتاه‌ترین مسیرها، در ضرب ماتریسی معمولی متناظر با چیست؟

۲۵-۱-۴ نشان دهید که ضرب ماتریسی تعریف شده توسط EXTEND-SHORTEST-PATHS شرکت‌پذیر است.

۲۵-۱-۵ نشان دهید که چگونه می‌توان مسئله‌ی کوتاه‌ترین مسیرها از یک مبدأ را به صورت ضرب یک ماتریس و یک بردار نشان داد؟ نشان دهید که محاسبه‌ی این ضرب چگونه متناظر است با یک الگوریتم مشابه الگوریتم بلمن–فورد (بخش ۲۴-۱ را ببینید).

۲۵-۱-۶ فرض کنید در الگوریتم این بخش، علاوه بر وزن کوتاه‌ترین مسیرها می‌خواهیم رأس‌های روی کوتاه‌ترین مسیرها را هم محاسبه کنیم. نشان دهید که چگونه می‌توانیم ماتریس Π را از ماتریس کامل شده‌ی L مربوط به وزن کوتاه‌ترین مسیرها در زمان $O(n^3)$ محاسبه کنیم.

۲۵-۱-۷ رأس‌های روی کوتاه‌ترین مسیرها را هم‌زمان با وزن کوتاه‌ترین مسیرها هم می‌توان محاسبه کرد. اجازه دهید $\pi_{ij}^{(m)}$ را به صورت عنصر ماقبل رأس j روی یک کوتاه‌ترین مسیر از i به j که حداکثر m یال دارد تعریف کنیم. EXTEND-SHORTEST-PATHS و SLOW-ALL-PAIRS-SHORTEST-PATHS را طوری اصلاح کنید که ماتریس‌های $\Pi^{(1)}, \Pi^{(2)}, \ldots, \Pi^{(n-1)}$ را هم‌زمان با ماتریس‌های $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$ محاسبه کنند.

۲۵-۱-۸ رویه‌ی FASTER-ALL-PAIRS-SHORTEST-PATHS، همان طور که گفته شد، نیاز به $\lceil \lg(n-1) \rceil$ ماتریس ذخیره شده دارد هر یک با n^2 عنصر، که کل حافظه‌ی مورد نیاز آن $\theta(n^2 \lg n)$ خواهد بود. این رویه را طوری اصلاح کنید که فقط به $\theta(n^2)$ حافظه نیاز داشته باشد، بدین صورت که فقط از دو ماتریس $n \times n$ استفاده کند.

۲۵-۱-۹ FASTER-ALL-PAIRS-SHORTEST-PATHS را طوری اصلاح کنید که بتواند تعیین کند که گراف، دور وزن منفی دارد یا خیر.

۲۵-۱-۱۰ یک الگوریتم کارامد برای محاسبه‌ی طول (تعداد یال‌های) یک دور با وزن منفی با کم‌ترین طول در یک گراف ارائه کنید.

۲-۲۵ الگوریتم فلوید-وارشال

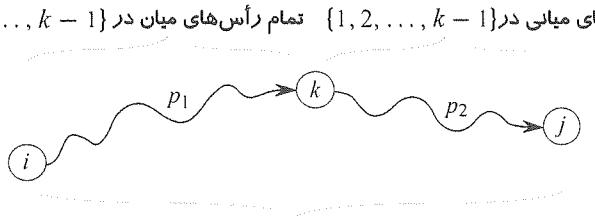
در این بخش از یک فرمول بندی برنامه‌ریزی پویای متفاوت برای حل مسئله‌ی کوتاه‌ترین مسیرها بین هر دو رأس بر روی یک گراف جهت‌دار $G = (V, E)$ استفاده می‌کنیم. الگوریتم حاصل معروف به *الگوریتم فلوید-وارشال* (Floyd-Warshall algorithm) در زمان $\theta(V^3)$ اجرا می‌شود. مانند قبل دورهای با وزن منفی ممکن است وجود داشته باشند، ولی فرض می‌کنیم که این دورها وجود ندارند. مانند بخش ۲۵-۱ از فرایند برنامه‌ریزی پویا برای طراحی الگوریتم استفاده خواهیم کرد. پس از آموختن الگوریتم حاصل، یک متد مشابه ارائه خواهیم کرد برای یافتن بستار تراگذار یک گراف جهت‌دار.

ساختار یک کوتاه‌ترین مسیر

در الگوریتم فلوید- وارشال، توصیفی متفاوت برای ساختار کوتاه‌ترین مسیرها از توصیف بخش ۲۵-۱ ارائه خواهیم کرد. الگوریتم رأس‌های میانی یک کوتاه‌ترین مسیر را در نظر می‌گیرد، که یک رأس *میانی* (intermediate) در یک مسیر ساده‌ی $p = \langle v_1, v_2, \ldots, v_l \rangle$، هر رأسی در p غیر از v_1 یا v_l است، یعنی هر رأسی در مجموعه‌ی $\{v_2, v_3, \ldots, v_{l-1}\}$.

الگوریتم فلوید- وارشال بر پایه‌ی ملاحظات زیر است. تحت این فرض که مجموعه‌ی رأس‌های G به صورت $V = \{1, 2, \ldots, n\}$ است، اجازه دهید زیرمجموعه‌ی $\{1, 2, \ldots, k\}$ از رأس‌ها را برای یک k در نظر بگیریم. برای هر جفت رأس $i, j \in V$، تمام مسیرهای از i به j را در نظر بگیرید که رأس‌های میانی آن‌ها همگی از مجموعه‌ی $\{1, 2, \ldots, k\}$ هستند، و فرض کنید p یک مسیر با کم‌ترین وزن از میان آن‌ها باشد. (مسیر p ساده است.) الگوریتم فلوید- وارشال یک رابطه میان مسیر p و کوتاه‌ترین مسیرها از i به j وقتی تمام رأس‌های میانی از مجموعه‌ی $\{1, 2, \ldots, k-1\}$ هستند برقرار می‌کند. این رابطه به این بستگی دارد که آیا k یک رأس میانی مسیر p هست یا نه.

- اگر k یک رأس میانی در مسیر p نباشد، آن گاه تمام رأس‌های میانی مسیر p در مجموعه‌ی $\{1, 2, \ldots, k-1\}$ هستند. بنابراین یک کوتاه‌ترین مسیر از رأس i به رأس j که تمام رأس‌های میانی آن از مجموعه‌ی $\{1, 2, \ldots, k-1\}$ هستند معادل است با یک کوتاه‌ترین مسیر از i به j که تمام رأس‌های میانی آن از مجموعه‌ی $\{1, 2, \ldots, k\}$ هستند.
- اگر k یک رأس میانی مسیر p باشد، آن گاه p را به صورت $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$ تجزیه می‌کنیم، همان طور که در شکل ۲۵-۳ نشان داده شده است. طبق لم ۲۴-۱، p_1 یک کوتاه‌ترین مسیر از i به k است که تمام رأس‌های میانی آن از مجموعه‌ی $\{1, 2, \ldots, k\}$ هستند. در واقع می‌توانیم یک ادعای کمی قوی‌تر بکنیم. چون رأس k یک رأس میانی مسیر p_1 نیست، تمام رأس‌های میانی p_1 از مجموعه‌ی $\{1, 2, \ldots, k-1\}$ هستند. بنابراین p_1 یک کوتاه‌ترین مسیر از i به k است که تمام رأس‌های میانی آن در مجموعه‌ی $\{1, 2, \ldots, k-1\}$ هستند. به طور مشابه p_2 یک کوتاه‌ترین مسیر از رأس k به رأس j است که تمام رأس‌های میانی آن از مجموعه‌ی $\{1, 2, \ldots, k-1\}$ هستند.



شکل ۲۵-۳ مسیر p یک کوتاه‌ترین مسیر از رأس i به رأس j است، و k رأس میانی این مسیر با بالاترین شماره است. مسیر p_1 (قسمتی از مسیر p که از رأس i تا رأس k است) تماماً از رأس‌های میانی $\{1, 2, \ldots, k-1\}$ تشکیل شده است. مسیر p_2 از رأس k تا رأس j هم به همین صورت است.

یک جواب بازگشتی برای مسئله‌ی کوتاه‌ترین مسیرها میان هر دو رأس

بر مبنای مشاهدات بالا، اکنون یک فرمول بندی بازگشتی برای تخمین کوتاه‌ترین مسیرها تعریف می‌کنیم که با فرمول بندی انجام شده در بخش ۲۵-۱ متفاوت است. فرض کنید $d_{ij}^{(k)}$ وزن یک کوتاه‌ترین مسیر از رأس i به رأس j باشد که تمام رأس‌های میانی آن در مجموعه‌ی $\{1, 2, \ldots, k\}$ هستند. وقتی $k = 0$، یک مسیر از i به j که شماره‌ی هیچ کدام از رأس‌های میانی آن بالاتر از ۰ نیست، در واقع هیچ رأس میانی ندارد. چنین مسیری حداکثر یک یال دارد، و بنابراین $d_{ij}^{(0)} = w_{ij}$. یک تعریف بازگشتی بر مبنای بحث بالا به صورت زیر است:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{اگر } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{اگر } k \geq 1 \end{cases} \qquad (۵-۲۵)$$

چون برای هر مسیری تمام رأس‌های میانی در مجموعه‌ی $\{1, 2, \ldots, n\}$ هستند، ماتریس $D^{(n)} = (d_{ij}^{(n)})$ جواب نهایی $d_{ij}^{(m)} = \delta(i, j)$ را برای تمام $i, j \in V$ می‌دهد.

محاسبه‌ی وزن کوتاه‌ترین مسیرها به صورت از پایین به بالا

بر مبنای رابطه‌ی بازگشتی (۲۵-۵) می‌توان از رویه‌ی از پایین به بالای زیر برای محاسبه‌ی مقادیر $d_{ij}^{(k)}$ به ترتیب افزایشی مقادیر k استفاده کرد. ورودی آن یک ماتریس W با اندازه‌ی $n \times n$ است که مانند تساوی (۲۵-۱) تعریف شده است. این رویه ماتریس وزن کوتاه‌ترین مسیرها ($D^{(n)}$) را بازمی‌گرداند.

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4      let D^(k) = (d_ij^(k)) be a new n × n matrix
5      for i = 1 to n
```

```
6        for j = 1 to n
7            d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1), d_kj^(k-1))
8    return D^(n)
```

شکل ۲۵-۴ ماتریس $D^{(k)}$ را که توسط الگوریتم فلوید-وارشال برای گراف شکل ۲۵-۱ محاسبه شده است، نشان می‌دهد.

زمان اجرای الگوریتم فلوید-وارشال توسط سه حلقه‌ی تودرتوی for در خطوط ۳-۷ تعیین می‌شود. چون اجرای خط ۷ در زمان $O(1)$ انجام می‌شود، کل الگوریتم در زمان $\theta(n^3)$ اجرا می‌شود. مانند الگوریتم نهایی در بخش ۲۵-۱، کد کوتاه است، و در آن از هیچ ساختمان داده‌ی پیچیده‌ای استفاده نشده است. از این رو ضرایب ثابت مخفی درون نماد θ کوچک هستند. بنابراین الگوریتم فلوید-وارشال حتی برای گراف‌های ورودی نسبتاً بزرگ هم الگوریتمی کاربردی است.

ساختن یک کوتاه‌ترین مسیر

متدهای مختلفی برای ساختن کوتاه‌ترین مسیرها در الگوریتم فلوید- وارشال وجود دارد. یک راه این است که ماتریس وزن کوتاه‌ترین مسیرهای D را بسازیم و سپس ماتریس عناصر ماقبل Π را از روی D بسازیم. تمرین ۲۵-۱-۶ از شما می‌خواهد این متد را طوری پیاده‌سازی کنید که در زمان $O(n^3)$ اجرا شود. با داشتن ماتریس عناصر ماقبل Π، می‌توان از رویه‌ی PRINT-ALL-PAIRS-SHORTEST-PATH برای چاپ تمام رأس‌ها روی یک کوتاه‌ترین مسیر داده شده استفاده کرد.

همچنین می‌توانیم ماتریس عناصر ماقبل Π را زمانی محاسبه کنیم که الگوریتم فلوید-وارشال ماتریس‌های $D^{(k)}$ را محاسبه می‌کند. به طور خاص، دنباله‌ای از ماتریس‌های $\Pi^{(0)}$، $\Pi^{(1)}$،...، $\Pi^{(n)}$ را محاسبه می‌کنیم، که در آن $\Pi = \Pi^{(n)}$، و $\pi_{ij}^{(k)}$ به صورت عناصر ماقبل رأس j روی یک کوتاه‌ترین مسیر از رأس i تعریف می‌شود، که در آن تمام رأس‌های میانی از مجموعه‌ی $\{1, 2, ..., k\}$ هستند.

می‌توانیم یک فرمول بندی بازگشتی برای $\pi_{ij}^{(k)}$ ارائه کنیم. وقتی $k = 0$، یک کوتاه‌ترین مسیر از i به j هیچ رأس میانی ندارد. بنابراین،

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{اگر } i = j \text{ یا } w_{ij} = \infty \\ i & \text{اگر } i \neq j \text{ و } w_{ij} < \infty \end{cases} \qquad (۶-۲۵)$$

برای $k \geq 1$، اگر مسیر $i \rightsquigarrow k \rightsquigarrow j$ را داشته باشیم، که در آن $k \neq j$، آن گاه عنصر ماقبل j معادل خواهد بود با عنصر ماقبل j که برای کوتاه‌ترین مسیر از k با رأس‌های میانی $\{1, 2, ..., k-1\}$ انتخاب کردیم. در غیر این صورت، همان عنصر ماقبلی را برای j انتخاب خواهیم کرد که برای کوتاه‌ترین مسیر از i با رأس‌های میانی $\{1, 2, ..., k-1\}$ انتخاب کردیم. به صورت رسمی برای $k \geq 1$ داریم

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{اگر } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{اگر } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases} \qquad (۷-۲۵)$$

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

شکل ۲۵-۴ دنباله‌ی ماتریس‌های $D^{(k)}$ و $\Pi^{(k)}$ که توسط الگوریتم فلوید-وارشال برای گراف شکل ۲۵-۱ محاسبه شده است.

ترکیب محاسبات ماتریس‌های $\Pi^{(k)}$ را با رویه‌ی FLOYD-WARSHALL به عنوان تمرین ۲۵-۲-۳ واگذار می‌کنیم. شکل ۲۵-۴ دنباله‌ی ماتریس‌های $\Pi^{(k)}$ را که الگوریتم حاصل برای گراف شکل ۲۵-۱ محاسبه کرده‌است، نشان می‌دهد. در تمرین‌ها کار مشکل‌تری که باید انجام دهید، اثبات این است که زیرگراف عناصر ماقبل $G_{\pi,i}$ یک درخت کوتاه‌ترین مسیرها با ریشه‌ی i است. همچنین یک راه دیگر برای ساختن کوتاه‌ترین مسیرها در تمرین ۲۵-۲-۷ داده شده است.

بستار تراگذار یک گراف جهت‌دار

با داشتن یک گراف جهت‌دار $G = (V, E)$ با مجموعه‌ی رأس‌های $V = \{1, 2, \ldots, n\}$، ممکن است

بخواهیم بدانیم که آیا برای هر جفت رأس $i, j \in V$، یک مسیر از i به j در G وجود دارد یا خیر. *بستار تراگذار* (transitive closure) G به صورت گراف $G^* = (V, E^*)$ تعریف می‌شود، که در آن:

$$E^* = \{(i, j) : \text{یک مسیر از رأس } i \text{ به رأس } j \text{ در گراف } G \text{ وجود دارد}\}$$

یک راه برای محاسبه‌ی بستار تراگذار یک گراف در زمان $\theta(n^3)$ این است که به تمام یال‌های E وزن ۱ بدهیم و سپس الگوریتم فلوید- وارشال را اجرا کنیم. اگر یک مسیر از رأس i به رأس j وجود داشته باشد، خواهیم داشت $d_{ij} < n$. در غیر این صورت $d_{ij} = \infty$.

یک راه مشابه دیگر برای محاسبه‌ی بستار تراگذار G در زمان $\theta(n^3)$ وجود دارد که می‌تواند در عمل در زمان و حافظه صرفه‌جویی کند. در این متد اعمال ریاضی min و + با اعمال منطقی $\vee$ («یا»ی منطقی) و $\wedge$ («و»ی منطقی) در الگوریتم فلوید-وارشال جایگزین می‌شوند. برای $i, j, k = 1, 2, \ldots, n$، تعریف می‌کنیم $t_{ij}^{(k)}$ برابر است با ۱ اگر یک مسیر از رأس i به رأس j در گراف G وجود داشته باشد که تمام رأس‌های میانی آن در مجموعه‌ی $\{1, 2, \ldots, k\}$ باشند، و در غیر این صورت برابر است با ۰. بستار تراگذار $G^* = (V, E^*)$ را بدین صورت می‌سازیم که یال (i, j) را در E^* قرار می‌دهیم اگر و فقط اگر $t_{ij}^{(n)} = 1$. یک تعریف بازگشتی از $t_{ij}^{(k)}$، مشابه رابطه‌ی بازگشتی (۲۵-۵) عبارت است از

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{اگر } i \neq j \text{ و } (i, j) \notin E \\ 1 & \text{اگر } i = j \text{ یا } (i, j) \in E \end{cases}$$

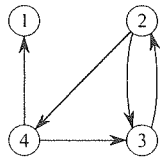
و برای $k \geq 1$،

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}) \qquad (۲۵\text{-}۸)$$

مانند الگوریتم فلوید-وارشال ماتریس‌های $T^{(k)} = (t_{ij}^{(k)})$ را به ترتیب افزایشی k محاسبه می‌کنیم.

```
TRANSITIVE-CLOSURE(G)
1  n = |G.V|
2  for i = 1 to n
3      for j = 1 to n
4          if i == j or (i, j) ∈ G.E
5              t_ij^(0) = 1
6          else t_ij^(0) = 0
7  for k = 1 to n
8      for i = 1 to n
9          for j = 1 to n
10             t_ij^(k) = t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
11 return T^(n)
```

شکل ۲۵-۵ ماتریس‌های $T^{(k)}$ را که توسط رویه‌ی TRANSITIVE-CLOSURE برای یک گراف نمونه محاسبه شده‌اند، نشان می‌دهد. رویه‌ی TRANSITIVE-CLOSURE مانند الگوریتم فلوید-وارشال



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

شکل ۲۵-۵ یک گراف جهت‌دار و ماتریس‌های $T^{(k)}$ که توسط الگوریتم بستار تراگذار محاسبه شده است.

در زمان $\theta(n^3)$ اجرا می‌شود. با این حال بر روی بعضی از کامپیوترها، اعمال منطقی بر روی مقادیر یک بیتی سریع‌تر از اعمال ریاضی بر روی داده‌های اعداد صحیح اجرا می‌شوند. به علاوه از آن جایی که الگوریتم بستار تراگذار مستقیم به جای اعداد صحیح فقط از مقادیر بولین استفاده می‌کند، فضای مورد نیاز آن به نسبت اندازه‌ی کلمه‌ها در حافظه‌ی کامپیوتر کم‌تر از الگوریتم فلوید–وارشال است.

تمرین‌ها

۲۵-۲-۱ الگوریتم فلوید- وارشال را بر روی گراف جهت‌دار و وزن‌دار شکل ۲۵-۲ اجرا کنید. ماتریس $D^{(k)}$ را که از هر بار تکرار حلقه‌ی خارجی حاصل می‌شود، نشان دهید.

۲۵-۲-۲ نشان دهید که چگونه می‌توان با استفاده از تکنیک بخش ۲۵-۱ بستار تراگذار را محاسبه کرد.

۲۵-۲-۳ رویه‌ی FLOYD-WARSHALL را طوری اصلاح کنید که ماتریس‌های $\Pi^{(k)}$ را طبق تساوی‌های (۲۵-۶) و (۲۵-۷) محاسبه کند. به دقت اثبات کنید که برای $i \in V$، زیرگراف عناصر ماقبل $G_{\pi,i}$ یک درخت کوتاه‌ترین مسیرها با ریشه‌ی i است. (*راهنمایی:* برای نشان دادن این که $G_{\pi,i}$ بدون دور است، ابتدا نشان دهید که طبق تعریف $\pi_{ij}^{(k)}$، تساوی $\pi_{ij}^{(k)} = 1$ ایجاب می‌کند که $d_{ij}^{(k)} \geq d_{ij}^{(k)} + w_{ij}$. سپس از اثبات لم ۲۴-۱۶ بهره بگیرید.)

۲۵-۲-۴ همان طور که در بالا مشخص است، الگوریتم فلوید- وارشال به حافظه‌ی $\theta(n^3)$ نیاز دارد، چرا که $d_{ij}^{(k)}$ را برای $i, j, k = 1, 2, \ldots, n$ محاسبه می‌کند. نشان دهید که رویه‌ی زیر که به سادگی تمام اندیس‌ها را حذف می‌کند، صحیح است، و بنابراین فقط به $\theta(n^2)$ حافظه نیاز است.

```
FLOYD-WARSHALL′ (W)
1  n = W.rows
2  D = W
3  for k = 1 to n
4      for i = 1 to n
5          for j = 1 to n
6              d_ij = min (d_ij, d_ik + d_kj)
7  return D
```

۲۵-۲-۵ فرض کنید روش برخورد با حالت‌های مساوی را در تساوی (۲۵-۷) به صورت زیر اصلاح می‌کنیم:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{اگر } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{اگر } d_{ij}^{(k-1)} \geq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

آیا این تعریف جایگزین برای ماتریس عناصر ماقبل Π صحیح است؟

۲۵-۲-۶ چگونه می‌توان از خروجی الگوریتم فلوید- وارشال برای تشخیص وجود یک دور با وزن منفی استفاده کرد؟

۲۵-۲-۷ یک روش دیگر برای بازسازی کوتاه‌ترین مسیرها در الگوریتم فلوید- وارشال، استفاده از مقادیر $\phi_{ij}^{(k)}$ برای $i, j, k = 1, 2, \ldots, n$ است، که $\phi_{ij}^{(k)}$ رأس میانی با بالاترین شماره در یک کوتاه‌ترین مسیر از i به j است که در آن فقط از رأس‌های میانی $\{1, 2, \ldots, k\}$ استفاده شده است. یک فرمول‌بندی بازگشتی برای $\phi_{ij}^{(k)}$ ارائه کنید، رویه‌ی فلوید- وارشال را برای استفاده از مقادیر $\phi_{ij}^{(k)}$ اصلاح کنید، و FLOYD-WARSHALL را طوری بازنویسی کنید که ماتریس $\Phi = (\phi_{ij}^{(n)})$ را به عنوان ورودی دریافت کند. ماتریس Φ از چه لحاظ مشابه جدول s در ضرب زنجیره‌ی ماتریس‌ها در بخش ۱۵-۲ است؟

۲۵-۲-۸ یک الگوریتم با زمان $O(VE)$ ارائه کنید که بستار تراگذار یک گراف $G = (V, E)$ را محاسبه می‌کند.

۲۵-۲-۹ فرض کنید می‌توان بستار تراگذار یک گراف جهت‌دار بدون دور را در زمان $f(|V|, |E|)$ محاسبه کرد، که در آن f یک تابع صعودی یکنواخت از $|V|$ و $|E|$ است. نشان دهید که زمان محاسبه‌ی بستار تراگذار $G^* = (V, E^*)$ برای یک گراف کلی $G = (V, E)$ برابر است با $f(|V|, |E|) + O(V + E^*)$.

۲۵-۳ الگوریتم جانسون برای گراف‌های خلوت

الگوریتم جانسون کوتاه‌ترین مسیرها میان هر دو رأس را در زمان $O(V^2 \lg V + VE)$ محاسبه می‌کند. برای گراف‌های خلوت، این زمان به صورت حدی بهتر از مربع گیری مکرر ماتریس‌ها و یا الگوریتم فلوید-وارشال است. این الگوریتم یا یک ماتریس از کوتاه‌ترین مسیرها برای تمام جهت رأس‌ها بازمی‌گرداند، و یا گزارش می‌کند که گراف ورودی حاوی یک دور با وزن منفی است. الگوریتم جانسون از یک زیرروال استفاده می‌کند که هر دو الگوریتم Dijkstra و بلمن-فورد از آن استفاده می‌کنند، که این زیرروال در فصل ۲۴ توصیف شده است.

الگوریتم جانسون از تکنیک *وزن دهی دوباره* (reweighting) استفاده می‌کند، که روش کار آن به صورت زیر است. اگر وزن تمام یال‌ها در یک گراف $G = (V, E)$ نامنفی باشد، آن گاه می‌توانیم کوتاه‌ترین مسیرها بین تمام جفت رأس‌ها را با اجرای الگوریتم Dijkstra برای هر رأس بیابیم؛ با پیاده‌سازی صف اولویت کمینه به کمک هرم فیبوناچی، زمان اجرای این الگوریتم $O(V^2 \lg V + VE)$ خواهد بود. اگر G یال‌های با وزن منفی داشته باشد ولی دور با وزن منفی نداشته باشد، به سادگی می‌توانیم یک مجموعه‌ی جدید از یال‌های با وزن نامنفی بیابیم که به ما اجازه می‌دهد از متد قبلی استفاده کنیم. مجموعه‌ی جدید وزن یال‌ها، $\hat{w}$، باید دو خصوصیت مهم زیر را داشته باشد.

۱. برای هر جفت رأس $u, v \in V$، یک مسیر p یک کوتاه‌ترین مسیر از u به v با استفاده از تابع وزن w است اگر و فقط اگر p یک کوتاه‌ترین مسیر از u به v با استفاده از تابع وزن $\hat{w}$ باشد.
۲. برای تمام یال‌های (u, v)، وزن جدید $\hat{w}(u, v)$ نامنفی است.

همان طور که به زودی خواهیم دید، پیش پردازش G برای تعیین تابع وزن جدید $\hat{w}$ را می‌توان در زمان $O(VE)$ انجام داد.

حفظ کوتاه‌ترین مسیرها در وزن‌دهی دوباره

همان طور که لم زیر نشان می‌دهد، یافتن یک وزن‌دهی دوباره برای یال‌ها به طوری که اولین خصوصیت بالا را ارضا کند، ساده است. از δ برای نشان دادن وزن کوتاه‌ترین مسیرهای به دست آمده از تابع وزن w، و از $\hat{\delta}$ برای نشان دادن وزن کوتاه‌ترین مسیرها به دست آمده از تابع وزن $\hat{w}$ استفاده می‌کنیم.

لم ۲۵-۱ وزن‌دهی دوباره

با داشتن یک گراف جهت‌دار و وزن‌دار $G = (V, E)$ با تابع وزن $w : E \to \mathbb{R}$، فرض کنید $h : V \to \mathbb{R}$ یک تابع باشد که رأس‌ها را به اعداد حقیقی نگاشت می‌کند. برای هر یال $(u, v) \in E$ تعریف می‌کنیم

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v) \qquad (۹-۲۵)$$

کوتاه‌ترین مسیرها با تغییر متن‌دادم

فرض کنید $p = \langle v_0, v_1, ..., v_k \rangle$ یک مسیر از رأس v_0 به رأس v_k باشد. آن گاه p یک کوتاه‌ترین مسیر از v_0 به v_k با تابع وزن w است اگر و فقط اگر یک کوتاه‌ترین مسیر با تابع وزن $\widehat{w}$ باشد. یعنی، $w(p) = \delta(v_0, v_k)$ اگر و فقط اگر $\widehat{w}(p) = \widehat{\delta}(v_0, v_k)$. همچنین با استفاده از تابع وزن w، گراف G یک دور منفی دارد اگر و فقط اگر با استفاده از تابع وزن $\widehat{w}$ هم یک دور منفی داشته باشد.

اثبات با نشان دادن

$$\widehat{w}(p) = w(p) + h(v_0) - h(v_k) \qquad (۱۰-۲۵)$$

آغاز می‌کنیم. داریم

$$\begin{aligned}
\widehat{w}(p) &= \sum_{i=1}^{k} \widehat{w}(v_{i-1}, v_i) \\
&= \sum_{i=1}^{k} (\widehat{w}(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\
&= \sum_{i=1}^{k} \widehat{w}(v_{i-1}, v_i) + h(v_0) - h(v_k) \qquad \text{(چون سری خاصیت تلسکوپی دارد)} \\
&= w(p) + h(v_0) - h(v_k)
\end{aligned}$$

بنابراین برای هر مسیر p از v_0 به v_k داریم $\widehat{w}(p) = w(p) + h(v_0) - h(v_k)$. چون $h(v_0)$ و $h(v_k)$ به مسیر بستگی ندارند، اگر یک مسیر از v_0 به v_k کوتاه‌تر از مسیری دیگر با استفاده از تابع وزن w باشد، آن گاه با استفاده از $\widehat{w}$ هم کوتاه‌تر خواهد بود. از این رو $w(p) = \delta(v_0, v_k)$ اگر و فقط اگر $\widehat{w}(p) = \widehat{\delta}(v_0, v_k)$.

نهایتاً نشان می‌دهیم که G با استفاده از تابع وزن w یک دور با وزن منفی دارد اگر و فقط اگر G با استفاده از تابع وزن $\widehat{w}$ یک دور با وزن منفی داشته باشد. یک دور دلخواه $c = \langle v_0, v_1, ..., v_k \rangle$ را در نظر بگیرید، که در آن $v_k = v_0$. طبق تساوی (۱۰-۲۵)،

$$\begin{aligned}
\widehat{w}(c) &= w(c) + h(v_0) - h(v_k) \\
&= w(c)
\end{aligned}$$

و بنابراین وزن c با استفاده از w منفی است اگر و فقط اگر وزن آن با استفاده از $\widehat{w}$ هم منفی باشد. ■

ساختن وزن‌های نامنفی با وزن‌دهی دوباره

هدف بعدی ما این است که اطمینان حاصل کنیم خصوصیت دوم برقرار است: می‌خواهیم $\widehat{w}(u, v)$ برای تمام یال‌های $(u, v) \in E$ نامنفی باشد. با داشتن یک گراف جهت‌دار و وزن‌دار $G = (V, E)$ با تابع وزن $w : E \to \mathbb{R}$، یک گراف جدید $G'(V', E')$ می‌سازیم، که در آن برای یک رأس جدید

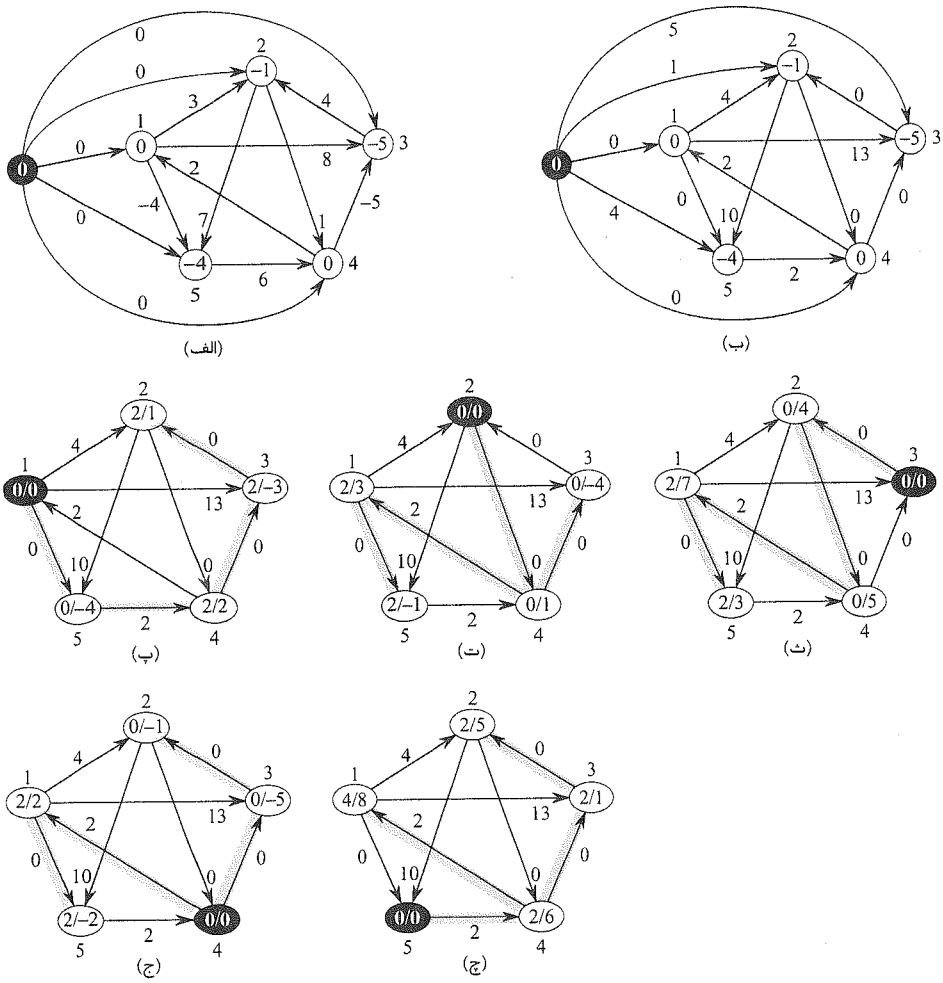
$s \notin V$ داریم $V' = V \cup \{s\}$، و $E' = E \cup \{(s, v) : v \in V\}$. تابع وزن w را طوری گسترش می‌دهیم که برای هر $v \in V$ داشته باشیم $w(s, v) = 0$. توجه کنید که چون هیچ یالی به s وارد نمی‌شود، هیچ کوتاه‌ترین مسیری در G'، غیر از آن‌هایی که از s شروع می‌شوند، شامل s نیستند. به علاوه G' هیچ دوری با وزن منفی ندارد اگر و فقط اگر G هیچ دوری با وزن منفی نداشته باشد. شکل ۲۵-۶(الف) گراف G' متناظر با گراف G شکل ۲۵-۱ را نشان می‌دهد.

اکنون فرض کنید که G و G' هیچ دوری با وزن منفی ندارند. اجازه دهید برای تمام $v \in V'$ تعریف کنیم $h(v) = \delta(s, v)$. طبق نامساوی مثلث (لم ۲۴-۱۰) برای تمام یال‌های $(u, v) \in E'$ داریم $h(v) \le h(u) + w(u, v)$. بنابراین اگر تابع وزن جدید $\hat{w}$ را طبق تساوی (۲۵-۹) تعریف کنیم، داریم $\hat{w}(u, v) = w(u, v) + h(u) - h(v) \ge 0$، و خصوصیت دوم برقرار خواهد بود. شکل ۲۵-۶(ب) گراف G' را از شکل ۲۵-۶(الف) با یال‌های دوباره وزن دهی شده نشان می‌دهد.

محاسبه‌ی کوتاه‌ترین مسیرها بین هر دو رأس

الگوریتم جانسون برای محاسبه‌ی کوتاه‌ترین مسیرها بین هر دو رأس از الگوریتم بلمن- فورد (بخش ۲۴-۱) و الگوریتم Dijkstra (بخش ۲۴-۳) به عنوان زیرروال استفاده می‌کند. این الگوریتم فرض می‌کند که یال‌ها در لیست‌های مجاورت ذخیره شده‌اند، و ماتریس همیشگی $D = d_{ij}$ با اندازه‌ی $|V| \times |V|$ را بازمی‌گرداند، که در آن $d_{ij} = \delta(i, j)$، و یا گزارش می‌کند که گراف حاوی یک دور با وزن منفی است. همان طور که برای تمام الگوریتم‌های کوتاه‌ترین مسیرها بین هر دو رأس فرض کردیم، این بار هم فرض می‌کنیم رأس‌ها از ۱ تا $|V|$ شماره‌گذاری شده‌اند.

```
JOHNSON(G)
1   compute G', where G'.V = G.V ∪ {s},
        G'.E = G.E ∪ {(s, v) : v ∈ G.V}, and
        w(s, v) = 0 for all v ∈ G.V
2   if BELLMAN-FORD(G', w, s) == FALSE
3       print "the input graph contains a negative-weight cycle"
4   else for each vertex v ∈ G'.V
5           set h(v) to the value of δ(s, v)
                computed by the Bellman-Ford algorithm
6       for each edge (u, v) ∈ G'.E
7           ŵ(u, v) = w(u, v) + h(u) − h(v)
8       let D = (d_uv) be a new n × n matrix
9       for each vertex u ∈ G.V
10          run DIJKSTRA(G, ŵ, u) to compute δ̂(u, v) for all v ∈ G.V
11              for each vertex v ∈ G.V
12                  d_uv = δ̂(u, v) + h(v) − h(u)
13  return D
```



شکل ۲۵-۶ الگوریتم کوتاه‌ترین مسیرها بین هر دو رأس جانسون، اجرا شده بر روی گراف شکل ۲۵-۱. شماره‌ی رأس‌ها خارج آن‌ها مشخص شده است. (الف) گراف G' با تابع وزن اصلی w. رأس جدید s تیره است. درون هر رأس v مقدار $h(v) = \delta(s, v)$ نشان داده شده است. (ب) هر یال (u, v) با تابع وزن $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$ دوباره وزن‌دهی شده است. (پ)-(چ) نتیجه‌ی اجرای الگوریتم Dijkstra بر روی هر رأس G با استفاده از تابع وزن $\hat{w}$. در هر قسمت رأس مبدأ u تیره است، و یال‌های سایه‌دار جزئی از درخت کوتاه‌ترین مسیرهای ساخته شده توسط الگوریتم هستند. درون هر رأس v مقادیر $\hat{\delta}(u, v)$ و $\delta(u, v)$ نشان داده شده است، که با یک ممیز از یکدیگر جدا شده‌اند. مقدار $d_{uv} = \delta(u, v)$ برابر است با $\hat{\delta}(u, v) + h(v) - h(u)$.

این کد به سادگی اعمالی را که در بالا گفتیم انجام می‌دهد. خط ۱ گراف G' را تولید می‌کند. خط ۲ الگورتیم بلمن-فورد را بر روی G' با تابع وزن w و رأس مبدأ s اجرا می‌کند. اگر G'، و بنابراین

G، حاوی یک دور منفی باشد، خط ۳ این مسئله را گزارش می‌دهد. در خطوط ۴-۱۲ فرض شده است که G' حاوی هیچ دوری با وزن منفی نیست. خطوط ۴-۵ $h(v)$ را برای هر $v \in V'$ برابر با وزن کوتاه‌ترین مسیر $\delta(s, v)$ محاسبه شده توسط الگوریتم بلمن- فورد قرار می‌دهند. در خطوط ۶-۷ وزن‌های جدید $\widehat{w}$ محاسبه می‌شوند. برای هر جفت رأس $u, v \in V$، حلقه‌ی for خطوط ۹-۱۲ وزن کوتاه‌ترین مسیر $\hat{\delta}(s, v)$ را با فراخوانی الگوریتم Dijkstra برای تک تک رأس‌های V محاسبه می‌کنند.

خط ۱۲ وزن کوتاه‌ترین مسیر $\delta(u, v)$ را در ورودی d_{uv} ماتریس ذخیره می‌کند، که با استفاده از تساوی (۲۵-۱۰) محاسبه شده است. نهایتاً در خط ۱۳ ماتریس کامل D بازگردانده می‌شود. شکل ۲۵-۶ اجرای الگوریتم جانسون را نشان می‌دهد.

اگر صف اولویت کمینه در الگوریتم Dijkstra با استفاده از هرم فیبوناچی پیاده‌سازی شود، زمان اجرای الگوریتم جانسون $O(V^2 \lg V + VE)$ خواهد بود. استفاده از روش ساده‌تر استفاده (پیاده‌سازی به کمک هرم کمینه‌ی دودویی) به زمان اجرای $O(VE \lg V)$ ختم می‌شود، که باز هم برای گراف‌های خلوت به صورت حدی از الگوریتم فلوید- وارشال سریع‌تر است.

تمرین‌ها

۲۵-۳-۱ با استفاده از الگوریتم جانسون، کوتاه‌ترین مسیرها میان هر دو رأس را برای گراف شکل ۲۵-۲ بیابید. مقادیر h و $\widehat{w}$ محاسبه شده توسط الگورتیم را نشان دهید.

۲۵-۳-۲ هدف از اضافه کردن رأس جدید s به V و تولید V' چیست؟

۲۵-۳-۳ فرض کنید برای تمام یال‌های $(u, v) \in E$ داریم $w(u, v) \geq 0$. رابطه‌ی میان تابع وزن w و $\widehat{w}$ چیست؟

۲۵-۳-۴ پروفسور Greenstreet ادعا می‌کند که یک راه ساده‌تر از راه استفاده شده در الگوریتم جانسون برای وزن‌دهی دوباره وجود دارد. با قرار دادن $w^* = \min_{(u,v) \in E} \{w(u, v)\}$، کافی است برای تمام یال‌های $(u, v) \in E$ تعریف کنیم $\widehat{w}(u, v) = w(u, v) - w^*$. در متد پروفسور برای وزن‌دهی دوباره چه اشتباهی وجود دارد؟

۲۵-۳-۵ فرض کنید الگوریتم جانسون را بر روی یک گراف جهت‌دار G با تابع وزن w اجرا می‌کنیم. نشان دهید که اگر G حاوی یک دور c با وزن ۰ باشد، آن گاه برای هر یال (u, v) در c داریم $\widehat{w}(u, v) = 0$.

۲۵-۳-۶ پروفسور Michener ادعا می‌کند که نیازی به ساختن یک رأس جدید در خط ۱ الگوریتم JOHNSON نیست. او ادعا می‌کند که به جای آن می‌توانیم از $G' = G$ استفاده کنیم و فرض کنیم s یک رأس دلخواه باشد. یک مثال از یک گراف جهت‌دار و وزن‌دار G ارائه کنید که برای آن ترکیب ایده‌ی پروفسور در رویه‌ی JOHNSON به جواب‌های اشتباه ختم

می‌شود. سپس نشان دهید که اگر G قویاً همبند باشد (تمام رأس‌ها از یکدیگر قابل دسترس باشند)، آن گاه نتایج بازگردانده شده توسط JOHNSON با اصلاحات پروفسور صحیح هستند.

مسائل

۲۵-۱ بستار تراگذار یک گراف پویا

فرض کنید می‌خواهیم بستار تراگذار یک گراف جهت‌دار $G = (V, E)$ را هم زمان با درج یال‌ها در E داشته باشیم. یعنی می‌خواهیم بعد از این که هر یال درج شد، بستار تراگذار یال‌های درج شده تا کنون را به هنگام سازی کنیم. فرض کنید گراف G در ابتدا هیچ یالی ندارد و بستار تراگذار به صورت یک ماتریس بولین نمایش داده خواهد شد.

I. نشان دهید که با درج یک یال جدید، بستار تراگذار $G^* = (V, E^*)$ برای یک گراف $G = (V, E)$ را می‌توان در زمان $O(V^2)$ به هنگام سازی کرد.

II. یک مثال از یک گراف G و یک یال e بدهید به طوری که برای به هنگام سازی بستار تراگذار پس از درج یال e به زمان $\Omega(V^2)$ نیاز داشته باشیم.

III. یک الگوریتم کارامد برای به هنگام سازی بستار تراگذار به هنگام درج یک یال در گراف ارائه کنید. برای هر دنباله‌ای از n درج، الگوریتم شما باید در زمان کلی $\sum_{i=1}^{n} t_i = O(V^3)$ اجرا شود، که در آن t_i زمان به هنگام سازی بستار تراگذار پس از درج یال iام است. اثبات کنید که الگوریتم شما این کران زمانی را حفظ می‌کند.

۲۵-۲ کوتاه‌ترین مسیرها در یک گراف ε- شلوغ

یک گراف $G = (V, E)$، ε- *شلوغ* است اگر برای یک ثابت ε در بازه‌ی $0 < \varepsilon \le 1$ داشته باشیم $|E| = \theta(V^{1+\varepsilon})$. با استفاده از هرم‌های کمینه‌ی dتایی (مسئله‌ی ۶-۲ را ببینید) در الگوریتم‌های کوتاه‌ترین مسیرها در گراف‌های ε- شلوغ، می‌توانیم به زمان اجرای الگوریتم‌های بر پایه‌ی هرم‌های فیبوناچی برسیم، بدون این که مجبور باشیم از ساختمان‌های داده‌ای با آن میزان پیچیدگی استفاده کنیم.

I. زمان اجرای حدی INSERT، EXTRACT-MIN و DECREASE-KEY، به صورت تابعی از d و n (تعداد عناصر هرم کمینه‌ی dتایی) چیست؟ اگر برای یک ثابت $0 < \alpha \le 1$ از $d = \theta(n^\alpha)$ استفاده کنیم، این زمان‌های اجرا چقدر خواهد بود؟ این زمان‌های اجرا را با هزینه‌های سرشکن اعمال بر روی یک هرم فیبوناچی مقایسه کنید.

II. نشان دهید که چگونه می‌توان کوتاه‌ترین مسیرها را از یک رأس بر روی یک گراف جهت‌دار ε- شلوغ که هیچ یالی با وزن منفی ندارد در زمان $O(E)$ محاسبه کرد. (*راهنمایی:* d را به صورت تابعی از ε انتخاب کنید.)

III. نشان دهید که چگونه می‌توان مسئله‌ی کوتاه‌ترین مسیرها میان هر دو رأس را بر روی یک گراف جهت‌دار ε – شلوغ $G=(V,E)$ که هیچ یالی با وزن منفی ندارد در زمان $O(VE)$ محاسبه کرد.

IV. نشان دهید که چگونه می‌توان در زمان $O(VE)$ مسئله‌ی کوتاه‌ترین مسیرها میان هر دو رأس را برای یک گراف جهت‌دار ε – شلوغ $G=(V,E)$ که ممکن است یال‌های با وزن منفی داشته باشد، ولی دور با وزن منفی ندارد محاسبه کرد.

فصل

۲۶

شار بیشینه

۲۶-۰ مقدمه

همان طور که می‌توانیم برای یافتن کوتاه‌ترین مسیرها از نقطه‌ای به نقطه‌ی دیگر، نقشه‌ی جاده‌ها را به صورت یک گراف جهت‌دار مدل کنیم، می‌توانیم یک گراف جهت‌دار را به صورت «شبکه‌ی شار» (flow network) در نظر گرفته و از آن برای جواب دادن به سؤال‌هایی در مورد شار مواد استفاده کنیم. ماده‌ای را در نظر بگیرید که در یک سیستم از مبدأ، که در آن ماده تولید می‌شود، به چاهک (sink)، که ماده در آن مصرف می‌شود در حرکت است. مبدأ ماده را با نرخ ثابت تولید می‌کند، چاهک هم ماده را با همان نرخ مصرف می‌کند. «شار» ماده در هر نقطه از سیستم به صورت شهودی برابر است با سرعت حرکت ماده در آن نقطه. از شبکه‌های شار می‌توان برای مدل کردن مسئله‌های بسیاری استفاده کرد، از جمله جریان یک مایع درون شبکه‌ای از لوله‌ها، حرکت قطعات در خطوط تولید، جریان در مدارات الکتریکی، و حرکت اطلاعات در شبکه‌های مخابراتی.

هر یال جهت‌دار در شبکه‌ی شار را می‌توان به صورت یک مجرا برای عبور ماده در نظر گرفت. هر مجرا ظرفیت مشخص دارد، که به عنوان نرخ بیشینه‌ی عبور شار ماده تعیین شده است، مانند عبور ۲۰۰ گالن مایع بر ساعت از یک لوله و یا عبور ۲۰ آمپر جریان الکتریکی از یک سیم. رأس‌ها تقاطع مجراها هستند، و به غیر از منبع و چاهک، مواد بدون مصرف شدن از رأس‌ها عبور می‌کنند. به عبارت دیگر نرخ ورود ماده به هر رأس باید برابر باشد با نرخ خروج از همان رأس. به این خصوصیت «بقای شار» می‌گوییم، که معادل است با قانون جریان کیرشهوف برای زمانی که ماده‌ی عبور کننده، جریان الکتریکی است.

در مسئله‌ی شار بیشینه می‌خواهیم بیشترین نرخی را محاسبه کنیم که ماده می‌تواند با آن نرخ از

منبع به چاهک منتقل شود، بدون این که هیچ محدودیت ظرفیتی را نقض کند. این مسئله یکی از ساده‌ترین مسائل مربوط به شبکه‌های شار است، و همان طور که در این فصل خواهیم دید، این مسئله را می‌توان به کمک الگوریتم‌های کارامدی حل کرد. به علاوه از این تکنیک اصلی که در آن از الگوریتم‌های شار بیشینه استفاده می‌شود، می‌توان برای حل دیگر مسائل شار شبکه نیز استفاده کرد.

در این فصل دو متد کلی برای حل مسائل شار بیشینه ارائه می‌شود. بخش ۲۶-۱ مفاهیم شبکه‌ها و شارها را فرمول بندی، و به صورت رسمی مسئله‌ی شار بیشینه را تعریف می‌کند. بخش ۲۶-۲ متد کلاسیک فورد و فولکرسن را برای یافتن شارهای بیشینه توضیح می‌دهد. یک کاربرد این متد، یافتن یک تطابق بیشینه در یک گراف بدون جهت دوبخشی، در بخش ۲۶-۳ داده شده است. بخش ۲۶-۴ متد رانش-برچسب‌دهی مجدد را معرفی می‌کند، که زمینه‌ی بسیاری از سریع‌ترین الگوریتم‌های مربوط به مسائل شبکه‌ی شار است. بخش ۲۶-۵ الگوریتم «برچسب‌دهی مجدد به جلو» را پوشش می‌دهد، یک پیاده‌سازی خاص از متد رانش-برچسب‌دهی مجدد که در زمان $O(V^3)$ اجرا می‌شود. با این که این الگوریتم سریع‌ترین الگوریتم شناخته شده نیست، بعضی از تکنیک‌های استفاده شده در سریع‌ترین الگوریتم‌های حدی را نمایان می‌کند، و در عمل به صورت معقولی کارامد است.

۲۶-۱ شبکه‌های شار

در این بخش یک تعریف از منظر نظریه‌ی گراف برای شبکه‌های شار ارائه می‌کنیم، خصوصیات آن‌ها را بررسی، و به دقت مسئله‌ی شار بیشینه را تعریف می‌کنیم.

شبکه‌های شار و شارها

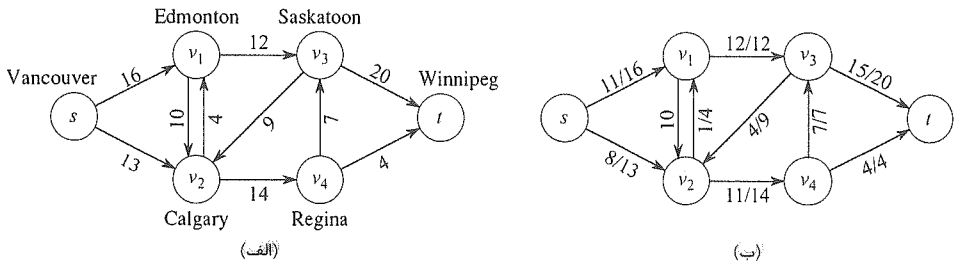
یک *شبکه‌ی شار* (flow network) $G = (V, E)$ گرافی جهت‌دار است که در آن هر یال $(u, v) \in E$ یک *ظرفیت* (capacity) نامنفی $c(u, v) \geq 0$ دارد. به علاوه نیاز داریم که اگر E حاوی یال (u, v) باشد، نباید یال (v, u) در جهت عکس هم در گراف موجود باشد. (به زودی خواهیم دید که چطور این محدودیت را ارضا کنیم.) اگر $(u, v) \notin E$، آن گاه برای سادگی تعریف می‌کنیم $c(u, v) = 0$، و همچنین طوقه‌ها را مجاز نمی‌دانیم. دو رأس خاص در یک شبکه‌ی شار داریم: یک *منبع* (source) s و یک *چاهک* (sink) t. برای سادگی فرض می‌کنیم تمام رأس‌ها بر روی یک مسیر از منبع به چاهک قرار دارند. یعنی برای هر رأس $v \in V$ یک مسیر $s \rightsquigarrow v \rightsquigarrow t$ وجود دارد. بنابراین گراف همبند است، و $|E| \geq |V| - 1$. شکل ۲۶-۱ مثالی از یک شبکه‌ی شار را نشان می‌دهد.

اکنون آماده هستیم که شارها را به صورت رسمی‌تر تعریف کنیم. فرض کنید $G = (V, E)$ یک شبکه‌ی شار با تابع ظرفیت c، منبع s و چاهک t باشد. یک شار در G، یک تابع حقیقی مقدار $f : V \times V \to \mathbb{R}$ است که دو خصوصیت زیر را ارضا می‌کند:

محدودیت ظرفیت: برای هر $u, v \in V$ باید داشته باشیم $f(u, v) \leq c(u, v)$.

بقای شار: برای هر $u \in V - \{s, t\}$، باید داشته باشیم $\sum_{v \in V} f(u, v) = 0$.

وقتی $(u, v) \notin E$، هیچ شاری از u به v نمی‌تواند وجود داشته باشد، و $f(u, v) = 0$.



شکل ۲۶-۱ (الف) یک شبکه‌ی شار $G = (V, E)$ برای مسئله‌ی حمل‌ونقل کمپانی Lucky Puck. کارخانه‌ی ونکوور (Vancouver) منبع s، و انبار وینیپگ (Winnipeg) چاهک t است. کالاها از شهرهای واسط عبور می‌کنند، ولی در هر روز فقط $c(u, v)$ جعبه می‌تواند از شهر u به شهر v حمل شود. ظرفیت هر یال بر روی آن مشخص شده است. (ب) شار f در G با مقدار $|f| = ۱۹$. هر یال (u, v) با برچسب $f(u, v)/c(u, v)$ مشخص شده است. از نماد ممیز صرفا برای جدا کردن شار از ظرفیت استفاده شده است؛ این نماد در این جا نشان دهنده‌ی تقسیم نیست.

کمیت $f(u, v)$، که می‌تواند مثبت، صفر، و یا منفی باشد، *مقدار شار* از رأس u به رأس v نام دارد. مقدار یک شار f به صورت

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) \qquad (۱-۲۶)$$

تعریف شده است، که برابر است با کل شار خروجی از منبع، منهای کل شار ورودی به منبع. (در این جا نماد $|f|$ نشان دهنده‌ی مقدار شار است، نه قدر مطلق یا اندازه.) معمولاً یک شبکه‌ی شار هیچ یال ورودی به منبع ندارد، و بنابراین طبق مجموع $\sum_{v \in V} f(v, s)$ شار ورودی به منبع ۰ خواهد بود. ولی در این جا آن را در نظر می‌گیریم، چرا که وقتی جلوتر در همین فصل شبکه‌های پس‌ماند را معرفی کردیم، شار ورودی به منبع قابل توجه خواهد بود. در *مسئله‌ی شار بیشینه* به ما یک شبکه‌ی شار G با منبع s و چاهک t داده شده است، و می‌خواهیم یک شار با مقدار بیشینه در آن بیابیم.

قبل از دیدن یک مثال از یک مسئله‌ی شار شبکه، اجازه دهید مختصراً تعریف شار و دو خصوصیت شار را بررسی کنیم. محدودیت ظرفیت به سادگی می‌گوید که شار از یک رأس به رأسی دیگر نباید از یک ظرفیت داده شده بیشتر شود. خصوصیت بقای شار می‌گوید که کل شار ورودی به یک رأس غیر از منبع یا چاهک، برابر است با کل شار خروجی از آن رأس – به طور غیر رسمی می‌توانیم بگوییم «شار ورودی برابر است با شار خروجی».

مثالی از شار

یک شبکه‌ی شار می‌تواند مسئله‌ی حمل‌ونقل نشان داده شده در شکل ۲۶-۱(الف) را مدل کند. کمپانی Lucky Puck یک کارخانه (منبع s) در ونکوور (Vancouver) دارد که کالاهای هاکی تولید می‌کند، و یک انبار (چاهک t) در وینیپگ (Winnipeg) دارد که کالاها را در آن جا ذخیره می‌کند. Lucky Puck

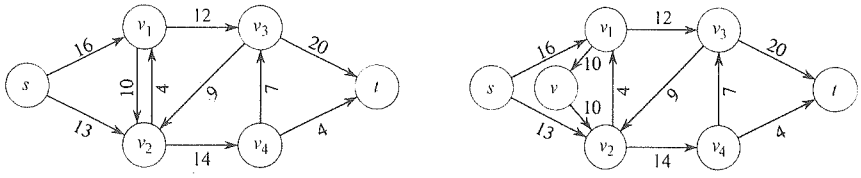
فضای کامیون‌ها را از شرکتی دیگر اجاره می‌کند تا کالاها را از کارخانه به انبار منتقل کند. چون کامیون‌ها در مسیرهای مشخص شده (یال‌ها) بین شهرها (رأس‌ها) حرکت می‌کنند و ظرفیت محدود دارند، Lucky Puck در روز حداکثر می‌تواند $c(u,v)$ جعبه بین هر جفت شهر u و v در شکل ۲۶-۱(الف) منتقل کند. Lucky Puck هیچ کنترلی بر روی این جاده‌ها و ظرفیت آن‌ها ندارد و بنابراین نمی‌تواند شبکه‌ی شار نشان داده شده در شکل ۲۶-۱(الف) را تغییر دهد. هدف آن‌ها این است که بیشترین تعداد جعبه (که آن را با p نشان می‌دهیم) را که در روز می‌توانند حمل کنند تعیین کرده و به همین مقدار تولید کنند، چرا که تولید کالا بیش از مقداری که می‌توانند به انبار حمل کنند فایده‌ای ندارد. مدت زمان رسیدن یک بار از کارخانه به انبار برای Lucky Puck اهمیتی ندارد؛ تنها چیزی که برای آن‌ها مهم است این است که روزانه p جعبه از کارخانه خارج و p جعبه به انبار وارد شود.

می‌توانیم «شار» حمل بار را با یک شار در این شبکه مدل کنیم چون تعداد جعبه‌های حمل شده در روز از یک شهر به شهری دیگر بر مبنای یک محدودیت ظرفیت است. به علاوه این مدل باید بقای شار را باید ارضا کند، چرا که در یک حالت پایدار نرخ ورود کالاها به یک شهر باید با نرخ خروج کالا از آن شهر برابر باشد. در غیر این صورت کالاها در یک شهر انبار می‌شوند.

مدل کردن مسئله‌های با یال‌های خلاف جهت

فرض کنید شرکت حمل و نقل به کمپانی Lucky Puck این امکان را بدهد که به اندازه‌ی ۱۰ کارتن فضا در کامیون‌هایی که از ادمونتون (Edmonton) به کلگری (Calgary) می‌روند، اجاره کند. به نظر طبیعی می‌آید که این امکان را به مثال خود اضافه کرده و شبکه‌ی نشان داده شده در شکل ۲۶-۲(الف) را تشکیل دهیم. ولی این شبکه از یک مشکل رنج می‌برد: در این شبکه این فرض که اگر $(v_1,v_2)\in E$، آن گاه $(v_1,v_2)\notin E$ نقض می‌شود. به دو یال (v_1,v_2) و (v_2,v_1) خلاف جهت، یا ضدموازی (antiparallel) می‌گوییم. بنابراین اگر بخواهیم یک شبکه‌ی شار با یال‌های خلاف جهت را مدل کنیم، باید شبکه را به یک شبکه‌ی معادل تبدیل کنیم که در آن یال‌های خلاف جهت وجود ندارد. شکل ۲۶-۲(ب) این شبکه‌ی معادل را نشان می‌دهد. یکی از دو یال خلاف جهت را انتخاب می‌کنیم (در این جا (v_1,v_2)) و آن را به شکل زیر می‌شکنیم: یک رأس v' اضافه کرده و یال (v_1,v_2) را با یال‌های (v_1,v') و (v',v_2) جایگزین می‌کنیم. همچنین ظرفیت هر دو یال جدید را به اندازه‌ی ظرفیت یال اصلی قرار می‌دهیم. شبکه‌ی حاصل این خصوصیت را ارضا می‌کند که اگر یک یال در شبکه باشد، یال برعکس آن در شبکه نیست. تمرین ۲۶-۱-۱ از شما می‌خواهد اثبات کنید که شبکه‌ی حاصل معادل شبکه‌ی اولیه است.

بنابراین می‌بینیم که ممکن است یک مسئله‌ی شار در دنیای واقعی به صورت طبیعی با شبکه‌ای با یال‌های خلاف جهت مدل شود. ولی مناسب خواهد بود اگر اجازه ندهیم یال‌های خلاف جهت در شبکه وجود داشته باشند، و بنابراین یک روش سرراست داریم برای تبدیل یک شبکه حاوی یال‌های خلاف جهت به یک شبکه‌ی معادل بدون یال‌های خلاف جهت.



شکل ۲۶-۳ تبدیل یک شبکه با یال‌های خلاف جهت به یک شبکه‌ی معادل بدون یال‌های خلاف جهت. (الف) یک شبکه‌ی شار حاوی هر دو یال (v_1, v_2) و (v_2, v_1). (ب) یک شبکه‌ی معادل بدون یال‌های خلاف جهت. در این شبکه رأس جدید v' را اضافه کرده و یال (v_1, v_2) را با دو یال (v_1, v') و (v', v_2) جایگزین می‌کنیم، که ظرفیت هر دو برابر است با ظرفیت (v_1, v_2).

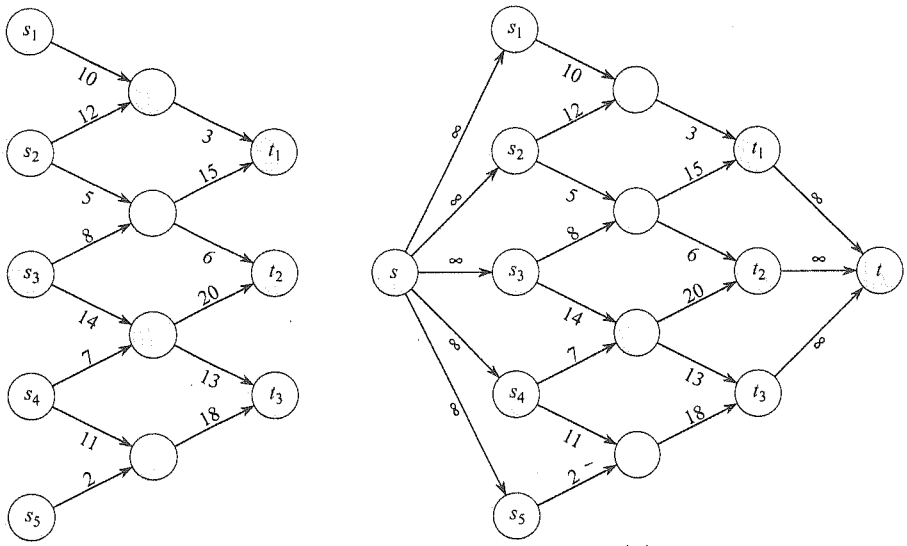
شبکه‌هایی با چند منبع و چاهک

یک مسئله‌ی شار بیشینه ممکن است به جای یک منبع و یک چاهک، چندین منبع و چاهک داشته باشد. مثلاً کمپانی Lucky Puck ممکن است در واقع مجموعه‌ای از m کارخانه‌ی $\{s_1, s_2, ..., s_m\}$ و مجموعه‌ای از n انبار $\{t_1, t_2, ..., t_n\}$ داشته باشد، همان طور که در شکل ۲۶-۳(الف) نشان داده شده است. خوشبختانه این مسئله سخت‌تر از مسئله‌ی شار بیشینه‌ی معمولی نیست.

می‌توانیم مسئله‌ی تعیین یک شار بیشینه در یک شبکه با چند منبع و چند چاهک را به یک مسئله‌ی شار بیشینه‌ی معمولی تبدیل کنیم. شکل ۲۶-۳(ب) نشان می‌دهد که چطور می‌توان شبکه‌ی (الف) را به یک شبکه‌ی شار معمولی که فقط یک منبع و یک چاهک دارد تبدیل کرد. برای این کار یک *ابرمنبع* (supersource) s و یک یال جهت‌دار (s, s_i) با ظرفیت $c(s, s_i) = \infty$ برای هر تمام $i = 1, 2, ..., m$ اضافه می‌کنیم. همچنین یک *ابرچاهک* (supersink) t می‌سازیم و یک یال (t_i, t) با ظرفیت $c(t_i, t) = \infty$ برای هر تمام $i = 1, 2, ..., n$ اضافه می‌کنیم. به صورت شهودی، هر شاری در شبکه‌ی (الف) متناظر است با یک شار در شبکه‌ی (ب) و بالعکس. تک منبع s به سادگی به اندازه‌ی مورد نیاز برای چند منبع s_i شار فراهم می‌کند، و به طور مشابه تک چاهک t به اندازه‌ی موردنیاز چند چاهک t_i شار مصرف می‌کند. تمرین ۲۶-۱-۲ از شما می‌خواهد به صورت رسمی اثبات کنید که این دو مسئله معادل هستند.

تمرین‌ها

۲۶-۱-۱ نشان دهید که تقسیم کردن یک یال در یک شبکه‌ی شار به یک شبکه‌ی شار معادل منجر می‌شود. به طور رسمی‌تر، فرض کنید شبکه‌ی شار G شامل یال (u, v) است، و می‌توانیم با اضافه کردن یک رأس جدید x و جایگزینی (u, v) با یال‌های جدید (u, x) و (x, v) با ظرفیت‌های $c(u, x) = c(x, v) = c(u, v)$، شبکه‌ی شار جدید G' را بسازیم. نشان دهید که مقدار شار بیشینه در G' برابر است با مقدار شار بیشینه در G.



شکل ۲۶-۳ تبدیل یک مسئله‌ی شار بیشینه با چند منبع و چند چاهک به یک مسئله با یک منبع و یک چاهک. (الف) یک شبکه‌ی شار با پنج منبع $S = \{s_1, s_2, s_3, s_4, s_5\}$ و سه چاهک $T = \{t_1, t_2, t_3\}$. (ب) یک شبکه‌ی شار معادل با یک منبع و یک چاهک. برای این تبدیل یک ابرمنبع s و یک یال با ظرفیت بی‌نهایت از s به هر یک از منابع اضافه می‌کنیم. همچنین یک ابرچاهک t با یک یال با ظرفیت بی‌نهایت از هر یک از چاهک‌ها به ابرچاهک اضافه می‌کنیم.

۲۶-۱-۲ خصوصیات و تعاریف شارها را برای مسئله‌ی با چند منبع و چند چاهک گسترش دهید. نشان دهید که هر شاری در یک شبکه با چند منبع و چند چاهک متناظر است با یک شار با مقدار برابر در یک شبکه با یک منبع و یک چاهک، که با اضافه کردن یک ابرمنبع و یک ابرچاهک به شبکه‌ی اصلی به دست آمده است، و برعکس.

۲۶-۱-۳ فرض کنید یک شبکه‌ی شار $G = (V, E)$ این فرض را نقض می‌کند که شبکه شامل یک مسیر $s \rightsquigarrow v \rightsquigarrow t$ برای هر $v \in V$ است. فرض کنید u رأسی باشد که برای آن هیچ مسیر $s \rightsquigarrow u \rightsquigarrow t$ وجود ندارد. نشان دهید که باید یک شار بیشینه‌ی f در G وجود داشته باشد به طوری که در آن برای هر رأس $v \in V$ داشته باشیم $f(u, v) = f(v, u) = 0$.

۲۶-۱-۴ فرض کنید f یک شار در یک شبکه باشد، و α یک عدد حقیقی. *ضرب اسکالر شار* (scalar flow product)، که با αf نشان داده می‌شود، تابعی از $V \times V$ به $\mathbb{R}$ است که به صورت زیر تعریف شده است:

$$(\alpha f)(u, v) = \alpha \cdot f(u, v)$$

اثبات کنید که شارها در یک شبکه، یک *مجموعه‌ی محدب* (convex set) را تشکیل می‌دهند.

یعنی نشان دهید که اگر f_1 و f_2 نشان‌دهنده‌ی شار باشند، آن گاه برای تمام αها در بازه‌ی $0 \le \alpha \le 1$، عبارت $\alpha f_1 + (1-\alpha) f_2$ هم نشان‌دهنده‌ی یک شار است.

۲۶-۱-۵ مسئله‌ی شار بیشینه را به صورت یک مسئله‌ی برنامه‌ریزی خطی بیان کنید.

۲۶-۱-۶ پروفسور آدام (Adam) دو فرزند دارد، که متأسفانه یکدیگر را دوست ندارند. این مسئله آن قدر جدی است که نه تنها این دو حاضر نیستند با یکدیگر به مدرسه بروند، بلکه در واقع هیچ یک حاضر نیست حتی در مکانی قدم بگذارد که دیگری در آن روز قدم گذاشته است. بچه‌ها با این که مسیرهایشان در یک گوشه با یکدیگر برخورد داشته باشد مشکلی ندارند. خوشبختانه هم خانه‌ی پروفسور و هم مدرسه در گوشه قرار دارند، ولی پروفسور مطمئن نیست که آیا می‌تواند فرزندان خود را به یک مدرسه بفرستد یا خیر. پروفسور یک نقشه از شهر خود دارد. نشان دهید که چگونه می‌توان مسئله‌ی تعیین این که آیا هر دو فرزند می‌توانند به یک مدرسه بروند یا نه را به صورت یک مسئله‌ی شار بیشینه مدل کرد.

۲۶-۱-۴ فرض کنید یک شبکه‌ی شار علاوه بر ظرفیت یال‌ها، *ظرفیت رأس* هم دارد. یعنی هر رأس v یک محدودیت $l(v)$ بر روی مقدار شاری که از آن رأس عبور می‌کند، دارد. نشان دهید چطور می‌توان یک شبکه‌ی شار $G = (V, E)$ با ظرفیت رأس را به یک شبکه‌ی شار معادل $G' = (V', E')$ بدون ظرفیت رأس تبدیل کرد، به طوری که مقدار شار بیشینه در G' برابر باشد با مقدار شار بیشینه در G. گراف G' چند رأس و یال دارد؟

۲۶-۲ متد فورد- فولکرسن

در این بخش متد فورد–فولکرسن برای حل مسئله‌ی شار بیشینه معرفی می‌شود. به آن به جای یک «الگوریتم» یک «متد» می‌گوییم، چرا که دارای پیاده‌سازی‌های مختلفی با زمان‌های اجرای مختلف است. متد فورد–فولکرسن به سه ایده‌ی مهم وابسته است که فراتر از متد هستند و با بسیاری از الگوریتم‌ها و مسائل شار ارتباط دارند: شبکه‌های پس‌ماند، مسیرهای تکمیلی، و برش‌ها. این ایده‌ها برای قضیه‌ی مهم شار بیشینه–برش کمینه (قضیه‌ی ۲۶–۷) بسیار حیاتی هستند، که این قضیه مقدار شار بیشینه را برحسب برش‌های شبکه‌ی شار تعیین می‌کند. این بخش را با ارائه‌ی یک پیاده‌سازی از متد فورد– فولکرسن و تحلیل زمان اجرای آن به پایان می‌بریم.

متد فورد– فولکرسن به صورت تکراری (در مقابل بازگشتی) مقدار شار را افزایش می‌دهد. با $f(u, v) = 0$ برای هر $u, v \in V$ آغاز می‌کنیم، که شار اولیه‌ی ۰ را به ما می‌دهد. در هر تکرار، مقدار شار را با یافتن «مسیر تکمیلی» (augmenting path) در یک «شبکه‌ی پس‌ماند» (residual network) G_f متناظر افزایش می‌دهیم. وقتی یال‌های مسیر تکمیلی را در G_f بدانیم، به سادگی می‌توانیم یال‌های خاصی را شناسایی کنیم که با تغییر شار در آن‌ها مقدار کل شار افزایش می‌یابد. با این که هر تکرار متد فورد– فولکرسن مقدار شار کل را افزایش می‌دهد، خواهیم دید که شار هر یال خاص در G ممکن

است افزایش یا کاهش یابد؛ ممکن است کاهش شار در یک یال برای الگوریتم ضروری باشد تا بتواند شار بیشتری از منبع به چاهک بفرستد. فرآیند تکمیل شار را آن قدر تکرار می‌کنیم تا شبکه‌ی پس‌ماند مسیر تکمیلی دیگری نداشته باشد. قضیه‌ی شار بیشینه-برش کمینه نشان می‌دهد که این فرآیند در پایان به یافتن شار بیشینه ختم می‌شود.

```
FORD-FULKERSON-METHOD(G, s, t)
1  initialize flow f to 0
2  while there exists an augmenting path p in the residual network G_f
3      augment flow f along p
4  return f
```

برای پیاده‌سازی و تحلیل متد فورد-فولکرسن، نیاز به معرفی مفاهیم متعدد دیگری داریم.

شبکه‌های پس‌ماند

به صورت شهودی، با داشتن یک شبکه G و یک شار f، شبکه‌ی پس‌ماند G_f شامل یال‌هایی با ظرفیت‌هایی می‌شود که نشان دهنده‌ی امکان تغییر شار در G هستند. یک یال از یک شبکه‌ی شار می‌تواند مقداری شار اضافی بپذیرد که برابر است با ظرفیت یال منهای شار فعلی روی یال. اگر این مقدار مثبت باشد، آن یال را با «ظرفیت پس‌ماند» $c_f(u,v) = c(u,v) - f(u,v)$ در شبکه‌ی پس‌ماند G_f قرار می‌دهیم. تنها یال‌هایی از G که در G_f قرار می‌گیرند، آن‌هایی هستند که می‌توانند شار بیشتری بپذیرند؛ برای یال‌هایی مانند (u,v) که شار آن‌ها برابر ظرفیتشان است، داریم $c_f(u,v) = 0$، که این یال‌ها در G_f نیستند.

ولی شبکه‌ی پس‌ماند G_f ممکن است حاوی یال‌هایی باشد که در G نیستند. همین طور که یک الگوریتم شار را با هدف افزایش آن، دست‌کاری می‌کند، ممکن است نیاز داشته باشد که شار روی یک یال خاص را کاهش دهد. برای نمایش کاهش احتمالی یک شار مثبت $f(u,v)$ بر روی یک یال در G، یک یال (v,u) در G_f قرار می‌دهیم، با ظرفیت پس‌ماند $c_f(v,u) = f(u,v)$ – یعنی یک یال که می‌تواند شار را در خلاف جهت (u,v) حرکت دهد، و مقدار آن حداکثر به اندازه‌ای است که شار روی (u,v) را خنثی می‌کند. یال برعکس در شبکه‌ی پس‌ماند به الگوریتم‌ها اجازه می‌دهد که شاری را که از روی یک یال عبور کرده است، بازگردانند. بازگرداندن شار از روی یک یال معادل است با *کاهش* شار روی آن یال، که یک عملیات لازم در بسیاری از الگوریتم‌ها است.

به صورت رسمی‌تر، فرض کنید یک شبکه‌ی شار $G = (V,E)$ داریم با منبع s و چاهک t. فرض کنید f یک شار در G باشد، و یک جفت رأس $u, v \in V$ را در نظر بگیرید. ظرفیت پس‌ماند c_f را به صورت زیر تعریف می‌کنیم:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{اگر } (u,v) \in E \\ f(u,v) & \text{اگر } (v,u) \in E \\ 0 & \text{در غیر این صورت} \end{cases} \qquad (۲-۲۶)$$

به دلیل فرضی که کردیم، مبنی بر این که $(u,v) \in E$ نتیجه می‌دهد $(v,u) \notin E$، دقیقاً یکی از سه حالت بالا برای هر جفت از رأس‌ها به کار می‌رود.

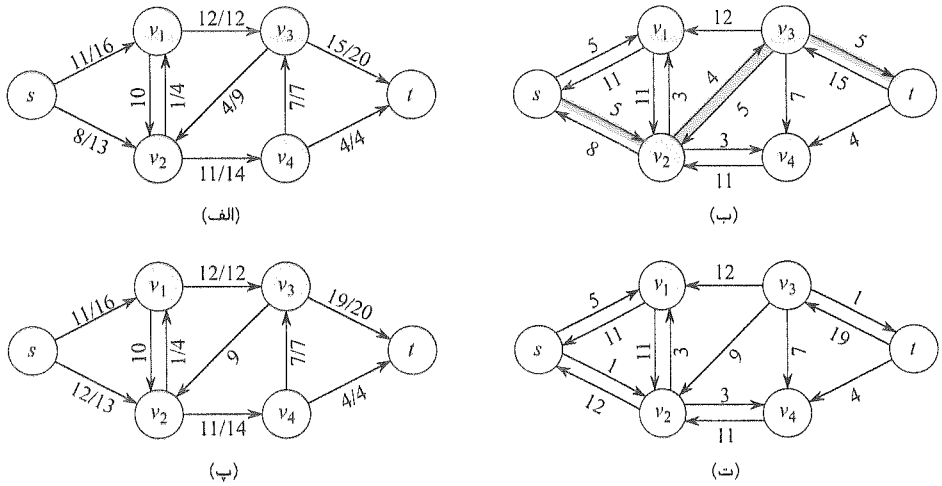
به عنوان یک مثال از تساوی (۲۶-۲)، اگر $c(u,v) = ۱۶$ و $f(u,v) = ۱۱$، آن گاه می‌توانیم $f(u,v)$ را به اندازه‌ی $c_f(u,v) = ۵$ واحد اضافه کنیم بدون این که از محدودیت ظرفیت (u,v) فراتر رویم. همچنین می‌خواهیم به الگوریتم‌ها اجازه دهیم که حداکثر ۱۱ واحد شار را از v به u بازگردانند، و بنابراین $c_f(v,u) = ۱۱$.

با داشتن یک شبکه‌ی شار $G = (V,E)$ و یک شار f، *شبکه‌ی پس‌ماند* G (residual network) حاصل از f برابر است با $G_f = (V,E_f)$، که در آن

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > ۰\}$$

یعنی همان طور که در بالا گفته شد، هر یال در یک شبکه‌ی پس‌ماند، یا هر *یال پس‌ماند*، می‌تواند یک شار اضافی را که بیش از ۰ است بپذیرد. شکل ۲۶-۴(الف) شبکه‌ی شار G و شار f از شکل ۲۶-۱(ب) را تکرار می‌کند، و شکل ۲۶-۴(ب) شبکه‌ی پس‌ماند متناظر G_f را نشان می‌دهد. یال‌های G_f یا یال‌های E هستند و یا یال‌های معکوس، و بنابراین $|E_f| \le ۲|E|$.

مشاهده کنید که شبکه‌ی پس‌ماند G_f مشابه یک شبکه‌ی شار با ظرفیت‌های مشخص شده با c_f



شکل ۲۶-۴ (الف) شبکه‌ی شار G و شار f از شکل ۲۶-۱(ب). (ب) شبکه‌ی پس‌ماند G_f که در آن مسیر تکمیلی p سایه زده شده است؛ ظرفیت پس‌ماند آن $c_f(p) = c(v_۲,v_۳) = ۴$ است. یال‌های با ظرفیت پس‌ماند ۰، مانند $(v_۱,v_۳)$ نشان داده نشده‌اند، که این قرارداد را در ادامه‌ی این بخش حفظ می‌کنیم. (پ) شار G که از اضافه کردن ظرفیت تکمیلی مسیر p که ۴ است حاصل می‌شود. یال‌هایی که شاری حمل نمی‌کنند، مانند $(v_۳,v_۲)$، فقط با ظرفیتشان برچسب‌گذاری شده‌اند؛ قرارداد دیگری که از آن پیروی خواهیم کرد. (ت) شبکه‌ی پس‌ماند حاصل از شار (پ).

است. این شبکه تعریف ما از شبکه‌های شار را ارضا نمی‌کند، چرا که ممکن است هر دو یال (u,v) و معکوس آن (v,u) را داشته باشد. غیر از این تفاوت، یک شبکه‌ی پس‌ماند همان خصوصیات شبکه‌های شار را دارد، و می‌توانیم یک شار در شبکه‌ی پس‌ماند تعریف کنیم که تعریف شار را ارضا می‌کند، ولی نسبت به ظرفیت‌های c_f در G_f.

یک شار در شبکه‌ی پس‌ماند، یک راهنما برای اضافه کردن شار به شبکه‌ی اصلی فراهم می‌کند. اگر f یک شار در G و f' یک شار در شبکه‌ی پس‌ماند متناظر G_f باشد، تکمیل (augmentation) شار f توسط f' را که به صورت $f \uparrow f'$ نشان می‌دهیم، به شکل یک تابع از $V \times V$ به $\mathbb{R}$ تعریف می‌کنیم:

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{اگر } (u,v) \in E \\ 0 & \text{در غیر این صورت} \end{cases} \qquad (۴-۲۶)$$

شهود پشت این تعریف از تعریف شبکه‌های پس‌ماند حاصل می‌شود. شار روی (u,v) را به اندازه‌ی $f'(u,v)$ افزایش داده و به اندازه‌ی $f'(v,u)$ کاهش می‌دهیم، چرا که حمل شار توسط یال معکوس در شبکه‌ی پس‌ماند معادل است با کاهش شار در شبکه‌ی اصلی. این کار (هدایت شار به یال معکوس در شبکه‌ی پس‌ماند) به *خنثی‌سازی* (cancellation) هم معروف است. برای مثال اگر ۵ کارتن لوازم هاکی از u به v و ۲ کارتن از v به u بفرستیم، می‌توانیم به طور معادل (از نظر نتیجه‌ی نهایی) فقط ۳ کارتن از u به v بفرستیم، و چیزی از v به u نفرستیم. خنثی‌سازی بدین شکل برای هر الگوریتم شار بیشینه ضروری است.

لم ۲۶-۱ فرض کنید $G = (V,E)$ یک شبکه‌ی شار با منبع s و چاهک t باشد، و f یک شار در G. فرض کنید G_f شبکه‌ی پس‌ماند G باشد که از شار f حاصل شده است، و f' یک شار در G_f. آن گاه شار مجموع $f + f'$، تعریف شده توسط تساوی (۴-۲۶)، یک شار در G با مقدار $|f| + |f'| = |f \uparrow f'|$ است.

اثبات ابتدا تحقیق می‌کنیم که آیا $f \uparrow f'$ از محدودیت ظرفیت برای هر یال در E و بقای شار برای هر رأس در $V - \{s,t\}$ پیروی می‌کند یا خیر.

برای محدودیت ظرفیت، ابتدا توجه می‌کنیم که اگر $(u,v) \in E$، آن گاه $c_f(u,v) = f(u,v)$. از این رو داریم $f'(u,v) \le c_f(v,u) = f(u,v)$، و بنابراین

$$\begin{aligned} (f \uparrow f')(u,v) &= f(u,v) + f'(u,v) - f'(v,u) && \text{(طبق تساوی (۲-۲۶))} \\ &\ge f(u,v) + f'(u,v) - f(u,v) && (\text{چون } f'(v,u) \le f(u,v)) \\ &= f'(u,v) \\ &\ge 0 \end{aligned}$$

به علاوه،

$$\begin{aligned}(f \uparrow f')(u,v) &= f(u,v)+f'(u,v)-f'(v,u) && \text{(طبق تساوی (۲۶-۲))}\\ &\le f(u,v)+f'(u,v) && \text{(چون شارها نامنفی هستند)}\\ &\le f(u,v)+c_f(u,v) && \text{(محدودیت ظرفیت)}\\ &= f(u,v)+c(u,v)-f(u,v) && (c_f \text{ تعریف})\\ &= c(u,v)\end{aligned}$$

برای بقای شار، توجه کنید که چون f و f' هر دو از بقای شار پیروی می‌کنند، برای هر $u \in V-\{s,t\}$ داریم

$$\begin{aligned}\sum_{v\in V}(f\uparrow f')(u,v) &= \sum_{v\in V}(f(u,v)+f'(u,v)-f'(v,u))\\ &= \sum_{v\in V}f(u,v)+\sum_{v\in V}f'(u,v)-\sum_{v\in V}f'(v,u)\\ &= \sum_{v\in V}f(v,u)+\sum_{v\in V}f'(v,u)-\sum_{v\in V}f'(u,v)\\ &= \sum_{v\in V}f(u,v)+f'(v,u)-f'(u,v)\\ &= \sum_{v\in V}(f\uparrow f')(v,u)\end{aligned}$$

که در آن خط سوم از خط دوم و بقای شار نتیجه می‌شود.

نهایتاً مقدار $|f\uparrow f'|$ را محاسبه می‌کنیم. به خاطر بیاورید که یال‌های خلاف جهت در G مجاز نیستند (هرچند در G' هستند)، و بنابراین برای هر رأس $v \in V$ می‌دانیم که می‌توانیم یک یال (s,v) و یا (v,s) داشته باشیم، ولی نه هر دو. تعریف می‌کنیم $V_1=\{v:(s,v)\in E\}$ مجموعه‌ی رأس‌هایی باشد که یالی از s به آن‌ها وجود دارد، و $V_2=\{v:(v,s)\in E\}$ مجموعه‌ی رأس‌هایی که یالی به s دارند. داریم $V_1\cup V_2\subseteq V$، و چون یال‌های خلاف جهت مجاز نیستند، $V_1\cap V_2=\varnothing$. اکنون داریم

$$\begin{aligned}|f\uparrow f'| &= \sum_{v\in V}(f\uparrow f')(s,v)-\sum_{v\in V}(f\uparrow f')(v,s)\\ &= \sum_{v\in V}(f\uparrow f')(s,v)-\sum_{v\in V}(f\uparrow f')(s,v)\end{aligned} \qquad (۵-۲۶)$$

که در آن خط دوم درست است چرا که $(f\uparrow f')(w,x)$ برابر با ۰ است اگر $(w,x)\notin E$. اکنون تعریف $f\uparrow f'$ را به همراه تساوی (۲۶-۵) به کار برده و سپس عبارت‌ها را بازآرایی می‌کنیم تا به دست آوریم

$$\begin{aligned}&|f\uparrow f'|\\ &= \sum_{v\in V_1}(f(s,v)+f'(s,v)-f'(v,s))-\sum_{v\in V_2}(f(v,s)+f'(v,s)-f'(s,v))\\ &= \sum_{v\in V_1}f(s,v)+\sum_{v\in V_1}f'(s,v)-\sum_{v\in V_1}f'(v,s)\\ &\quad -\sum_{v\in V_2}f(v,s)-\sum_{v\in V_2}f'(v,s)+\sum_{v\in V_2}f'(s,v)\end{aligned}$$

$$
\begin{aligned}
&= \sum_{v \in V_1} f(s,v) - \sum_{v \in V_2} f(v,s) \\
&\quad + \sum_{v \in V_1} f'(s,v) + \sum_{v \in V_2} f'(s,v) - \sum_{v \in V_1} f'(v,s) - \sum_{v \in V_2} f'(v,s) \qquad (۶-۲۶) \\
&= \sum_{v \in V_1} f(s,v) - \sum_{v \in V_2} f(v,s) + \sum_{v \in V_1 \cup V_2} f'(s,v) - \sum_{v \in V_1 \cup V_2} f'(v,s)
\end{aligned}
$$

در تساوی (۲۶-۶) می‌توانیم هر چهار سری را به کل مجموعه‌ی V گسترش دهیم، چرا که تمام جمله‌های اضافی مقدار ۰ دارند. (تمرین ۲۶-۲-۱ از شما می‌خواهد این را به صورت رسمی اثبات کنید.) بنابراین داریم

$$
\begin{aligned}
|f \uparrow f'| &= \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) + \sum_{v \in V} f'(s,v) - \sum_{v \in V} f'(v,s) \\
&= |f| + |f'|
\end{aligned}
$$

مسیرهای تکمیلی

با داشتن یک شبکه‌ی شار $G = (V, E)$ و یک شار f، یک *مسیر تکمیلی* (augmenting path) p یک مسیر ساده از s به t در شبکه‌ی پس‌ماند G_f است. طبق تعریف شبکه‌ی پس‌ماند، یک یال (u, v) روی یک مسیر تکمیلی یک شار مثبت تا سقف $c_f(u, v)$ را بدون نقض محدودیت ظرفیت یال روی هر یک از یال‌های (u, v) یا (v, u) که در شبکه‌ی شار اصلی هستند، می‌پذیرد.

مسیر سایه‌دار در شکل ۲۶-۴(ب) یک مسیر تکمیلی است. با نگاه به شبکه‌ی پس‌ماند G_f در شکل به صورت یک شبکه‌ی شار، می‌توانیم شار روی هر یال این میسر را بدون نقض محدودیت ظرفیت تا ۴ واحد بالا ببریم، چرا که کم‌ترین ظرفیت پس‌ماند روی این مسیر $c_f(v_2, v_3) = 4$ است. بیشترین مقدار شاری را که می‌توان روی تمام یال‌های یک مسیر تکمیلی p افزایش داد، *ظرفیت پس‌ماند* p می‌نامیم، که به صورت زیر تعریف می‌شود:

$$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ روی } p \text{ است}\}$$

لم زیر که اثبات آن به عنوان تمرین ۲۶-۲-۷ واگذار شده است، بحث بالا را به صورت دقیق‌تر بیان می‌کند.

لم ۲۶-۲

فرض کنید $G = (V, E)$ یک شبکه‌ی شار باشد، و f یک شار در G، و فرض کنید p یک مسیر تکمیلی در G_f باشد. تابع $f_p : V \times V \to \mathbb{R}$ را به صورت زیر تعریف می‌کنیم:

$$
f_p(u, v) = \begin{cases} c_f(p) & \text{اگر } (u, v) \text{ بر روی } p \text{ باشد} \\ 0 & \text{در غیر این صورت} \end{cases} \qquad (۸-۲۶)
$$

در این صورت f_p یک شار در G_f با مقدار $|f_p| = c_f(p) > 0$ است.

نتیجه‌ی زیر نشان می‌دهد که اگر f_p را به f اضافه کنیم، یک شار دیگر در G به دست می‌آوریم که مقدار آن به بیشینه نزدیک‌تر است. شکل ۲۶-۴(پ) نتیجه‌ی اضافه کردن f_p در شکل ۲۶-۴(ب) را به f از شکل ۲۶-۴(الف) نشان می‌دهد، و شکل ۲۶-۴(ت) شبکه‌ی پس‌ماند حاصل را نشان می‌دهد.

نتیجه‌ی ۲۶-۳ فرض کنید $G=(V,E)$ یک شبکه‌ی شار، f یک شار در G، و p یک مسیر تکمیلی در G_f باشد. فرض کنید f_p به صورت تساوی (۲۶-۸) تعریف شده باشد، و همچنین فرض کنید f را به اندازه‌ی f_p افزایش می‌دهیم. در این صورت تابع $f \uparrow f_p$ یک شار روی G است با مقدار $|f \uparrow f_p| = |f| + |f_p| > |f|$.

اثبات مستقیماً از لم‌های ۲۶-۱ و ۲۶-۲. ■

برش‌های شبکه‌های شار

متد فورد- فولکرسن مکرراً شار روی مسیرهای تکمیلی را به شار اصلی اضافه می‌کند تا وقتی که به شار بیشینه برسد. چطور باید مطمئن باشیم که وقتی الگوریتم پایان می‌یابد، واقعاً یک شار بیشینه در شکبه پیدا کرده‌ایم؟ قضیه‌ی شار بیشینه-برش کمینه که به زودی آن را اثبات خواهیم کرد، به ما می‌گوید که یک شار، بیشینه است اگر و فقط اگر شبکه‌ی پس‌ماند آن حاوی هیچ مسیر تکمیلی نباشد. با این حال برای اثبات این قضیه باید ابتدا مفهوم برش روی یک شبکه‌ی شار را توصیف کنیم.

یک *برش* (cut) (S,T) روی یک شبکه‌ی شار $G=(V,E)$، یک تقسیم بندی V به S و $T=V-S$ است به طوری که $s \in S$ و $t \in T$. (این تعریف مشابه تعریف «برش» برای درخت‌های پوشای کمینه است که در فصل ۲۳ از آن استفاده کردیم، غیر از این که در این جا به جای یک گراف بدون جهت، بر روی یک گراف جهت‌دار برش ایجاد می‌کنیم، و اصرار داریم که $s \in S$ و $t \in T$.) اگر f یک شار باشد، آن گاه *شار خالص* روی برش (S,T) به صورت $f(S,T)$ تعریف می‌شود، و برابر است با

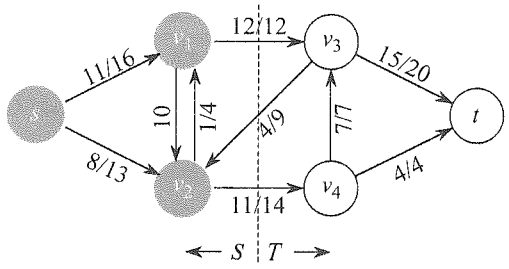
$$f(S,T) = \sum_{v \in S} \sum_{v \in T} f(u,v) - \sum_{v \in S} \sum_{v \in T} f(v,u) \qquad (۹-۲۶)$$

ظرفیت یک برش (S,T) عبارت است از

$$c(S,T) = \sum_{v \in S} \sum_{v \in T} c(u,v) \qquad (۱۰-۲۶)$$

یک *برش کمینه* از یک شبکه عبارت است از یک برش که ظرفیت آن در میان تمام برش‌ها، کمینه است.

عدم تقارن میان تعریف شار و ظرفیت یک برش، عمدی و مهم است. برای ظرفیت، فقط ظرفیت یال‌هایی را در نظر می‌گیریم که از S به T می‌روند، و از یال‌های با جهت عکس صرف نظر می‌کنیم.



شکل ۲۶-۵ یک برش (S,T) در شبکه‌ی شکل ۲۶-۱(ب)، که در آن $S=\{s,v_1,v_2\}$ و $T=\{v_3,v_4,t\}$. رأس‌های S تیره و رأس‌های T روشن هستند. شار خالص روی (S,T) برابر است با $f(S,T)=19$، و ظرفیت آن برابر است با $c(S,T)=26$.

برای شار، چیزی که در نظر می‌گیریم عبارت است از شاری که از S به T می‌رود، منهای شاری که در جهت عکس از T به S می‌رود. دلیل این تفاوت بعداً در همین بخش روشن خواهد شد.

شکل ۲۶-۵ برش $(\{s,v_1,v_2\},\{v_3,v_4,t\})$ را در شبکه‌ی شکل ۲۶-۱(ب) نشان می‌دهد. شار خالص روی این برش برابر است با

$$f(v_1,v_3)+f(v_2,v_3)+f(v_2,v_4)=12+(-4)+11=19$$

و ظرفیت آن برابر است با

$$c(v_1,v_3)+c(v_2,v_4)=12+14=26$$

لم زیر نشان می‌دهد که برای یک شار داده شده‌ی f، شار خالص روی تمام برش‌ها یکسان و برابر با مقدار شار است.

لم ۲۶-۴ فرض کنید f یک شار در شبکه‌ی شار G با منبع s و چاهک t باشد، و (S,T) یک برش روی G. آن گاه شار خالص روی (S,T) برابر است با $f(S,T)=|f|$.

اثبات می‌توانیم شرط بقای شار را برای هر گره‌ی $u\in V-\{s,t\}$ به صورت زیر بازنویسی کنیم:

$$\sum_{v\in V} f(u,v)-\sum_{v\in V} f(v,u)=0 \qquad (۱۱-۲۶)$$

توجه به تعریف $|f|$ از تساوی (۱-۲۶) و اضافه کردن سمت چپ تساوی (۱۱-۲۶)، که برابر است با ۰، و جمع آن برای تمام رأس‌های $S-\{s\}$ به دست می‌دهد

$$|f|=\sum_{v\in V} f(s,v)-\sum_{v\in V} f(v,s)+\sum_{u\in S-\{s\}}\left(\sum_{v\in V} f(u,v)-\sum_{v\in V} f(v,u)\right)$$

باز کردن مجموع سمت راست و گروه‌بندی مجدد عبارت‌ها نتیجه می‌دهد

$$
\begin{aligned}
|f| &= \sum_{v\in V} f(s,v) - \sum_{v\in V} f(v,s) + \sum_{u\in S-\{s\}} \sum_{v\in V} f(u,v) - \sum_{u\in S-\{s\}} \sum_{v\in V} f(v,u) \\
&= \sum_{v\in V} \left(f(s,v) + \sum_{u\in S-\{s\}} f(u,v) \right) - \sum_{v\in V} \left(f(v,s) + \sum_{u\in S-\{s\}} f(v,u) \right) \\
&= \sum_{v\in V} \sum_{u\in S} f(u,v) - \sum_{v\in V} \sum_{u\in S} f(v,u)
\end{aligned}
$$

چون $V = S \cup T$ و $S \cap T = \varnothing$، می‌توانیم هر یک از سری‌های روی V را به دو سری روی S و T بشکنیم و به دست آوریم

$$
\begin{aligned}
|f| &= \sum_{v\in S} \sum_{u\in S} f(u,v) + \sum_{v\in T} \sum_{u\in T} f(u,v) - \sum_{v\in S} \sum_{u\in S} f(v,u) - \sum_{v\in T} \sum_{u\in T} f(v,u) \\
&= \sum_{v\in T} \sum_{u\in S} f(u,v) - \sum_{v\in T} \sum_{u\in S} f(v,u) \\
&\quad + \left(\sum_{v\in S} \sum_{u\in S} f(u,v) - \sum_{v\in S} \sum_{u\in S} f(v,u) \right)
\end{aligned}
$$

دو مجموع درون پرانتز در واقع یکی هستند، چرا که برای تمام رأس‌های $x, y \in V$، عبارت $f(x,y)$ دقیقاً یک بار در هر یک از سری‌ها ظاهر می‌شود. بنابراین این دو سری یکدیگر را خنثی می‌کنند و داریم

$$
\begin{aligned}
|f| &= \sum_{u\in S} \sum_{v\in T} f(u,v) - \sum_{u\in S} \sum_{v\in T} f(v,u) \\
&= f(S,T)
\end{aligned}
$$

■

نتیجه‌ی زیر از لم ۲۶-۴ نشان می‌دهد که چگونه می‌توان از ظرفیت برش‌ها برای تعیین کران مقدار یک شار استفاده کرد.

نتیجه‌ی ۲۶-۵ مقدار هر شار f در یک شبکه‌ی شار G از بالا با ظرفیت هر برشی در G محدود شده است.

اثبات فرض کنید (S,T) یک برش دلخواه در G و f یک شار باشد. طبق لم ۲۶-۵ و محدودیت ظرفیت،

$$
\begin{aligned}
|f| &= f(S,T) \\
&= \sum_{u\in S} \sum_{v\in T} f(u,v) - \sum_{u\in S} \sum_{v\in T} f(v,u) \\
&\le \sum_{u\in S} \sum_{v\in T} f(u,v) \\
&\le \sum_{u\in S} \sum_{v\in T} c(u,v) \\
&= c(S,T)
\end{aligned}
$$

■

یک پی‌آمد نتیجه‌ی ۲۶-۵ این است که شار بیشینه در یک شبکه‌ی شار از بالا با ظرفیت یک برش کمینه در شبکه محدود شده است. قضیه‌ی مهم شار بیشینه-برش کمینه، که در این جا آن را تعریف و اثبات می‌کنیم، می‌گوید که مقدار یک شار بیشینه در واقع برابر است با ظرفیت یک برش کمینه.

قضیه‌ی ۲۶-۶ (قضیه‌ی شار بیشینه-برش کمینه)

اگر f یک شار در یک شبکه‌ی شار G با منبع s و چاهک t باشد، آن گاه وضعیت‌های زیر با یکدیگر معادل هستند:

۱. f یک شار بیشینه در G است.

۲. شبکه‌ی پس‌ماند G_f حاوی هیچ مسیر تکمیلی نیست.

برای یک برش (S,T) در G داریم $|f| = c(S,T)$.

اثبات (۲) $\Rightarrow$ (۱) : از طریق برهان خلف، فرض کنید f یک شار بیشینه در G باشد ولی G_f یک مسیر تکمیلی p داشته باشد. آن گاه طبق نتیجه‌ی ۲۶-۳، شار به دست آمده از اضافه کردن f_p به f، که در آن f_p از تساوی (۲۶-۸) به دست می‌آید، یک شار در G با مقداری اکیداً بیشتر از $|f|$ است، که با فرض بیشینه بودن f تناقض دارد.

(۳) $\Rightarrow$ (۲): فرض کنید G_f هیچ مسیر تکمیلی نداشته باشد، یعنی در G_f هیچ مسیری از s به t وجود نداشته باشد. تعریف می‌کنیم

$$S = \{v \in V \mid \text{یک مسیر از } s \text{ به } v \text{ در } G_f \text{ وجود داشته باشد}\}$$

و $T = V - S$. تقسیم بندی (S,T) یک برش است: بدیهتاً داریم $s \in S$، و $t \notin S$ چرا که در G_f هیچ مسیری از s به t وجود ندارد. اکنون یک جفت رأس u و v را در نظر بگیرید به طوری که $u \in S$ و $v \in T$. اگر $(u,v) \in E$، باید داشته باشیم $f(u,v) = c(u,v)$، چرا که در غیر این صورت $(u,v) \in E_f$، که v را در مجموعه‌ی S قرار خواهد داد. اگر $(v,u) \in E$، باید داشته باشیم $f(v,u) = 0$، چرا که در غیر این صورت $c_f(u,v) = f(v,u)$ باید مثبت باشد، و خواهیم داشت $(u,v) \in E_f$، که v را در S قرار خواهد داد. مسلماً اگر هیچ یک از (u,v) و (v,u) در E نباشند، آن گاه $f(u,v) = f(v,u) = 0$. باید داشته باشیم

$$\begin{aligned} f(S,T) &= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{v \in T} \sum_{u \in S} f(v,u) \\ &= \sum_{u \in S} \sum_{v \in T} c(u,v) - \sum_{v \in T} \sum_{u \in S} 0 \\ &= c(S,T) \end{aligned}$$

بنابراین، طبق لم ۲۶-۴ داریم $|f| = f(S,T) = c(S,T)$.

(۱) $\Rightarrow$ (۳): طبق نتیجه‌ی ۲۶-۶ برای تمام برش‌های (S,T) داریم $|f| \le c(S,T)$. بنابراین شرط $|f| = c(S,T)$ ایجاب می‌کند که f یک شار بیشینه باشد.

الگوریتم فورد–فولکرسن اولیه

در تکرارهای متد فورد- فولکرسن، هر بار یک مسیر تکمیلی p می‌یابیم و شار f را با استفاده از p اصلاح می‌کنیم. همان طور که لم ۲۶-۲ و نتیجه‌ی ۲۶-۳ پیشنهاد می‌کنند، f را با $f \uparrow f_p$ جایگزین می‌کنیم، که یک شار جدید به دست می‌دهد با مقدار $|f| + |f_p|$. پیاده‌سازی زیر از متد، شار بیشینه در یک شبکه‌ی $G = (V, E)$ را با به هنگام سازی خصیصه‌ی شار f برای هر یال $(u, v) \in E$ محاسبه می‌کند.[1] اگر $(u, v) \notin E$، به صورت ضمنی فرض می‌کنیم $f(u, v) = 0$. همچنین فرض می‌کنیم که ظرفیت‌های $c(u, v)$ به همراه گراف داده شده‌اند، و اگر $(u, v) \notin E$ آن گاه $c(u, v) = 0$. ظرفیت پس‌ماند $c_f(u, v)$ طبق فرمول (۲۶-۲) محاسبه شده است. عبارت $c_f(p)$ در کد در واقع فقط یک متغیر موقتی است که ظرفیت پس‌ماند مسیر p را ذخیره می‌کند.

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ G.E
2      (u, v).f = 0
3  while there exists a path p from s to t in the residual network G_f
4      c_f(p) = min {c_f(u, v) : (u, v) is in p}
5      for each edge (u, v) in p
6          if (u,v) ∈ E
7              (u, v).f = (u, v).f + c_f(p)
8          else (v, u).f = (v, u).f – c_f(p)
```

الگوریتم FORD-FULKERSON به سادگی شبه‌کد FORD-FULKERSON-METHOD را که قبل‌تر داده شد، گسترش می‌دهد. شکل ۲۶-۶ نتیجه‌ی هر تکرار الگوریتم را در یک اجرای نمونه نشان می‌دهد. خطوط ۱-۲ شار f را با ۰ مقداردهی اولیه می‌کنند. حلقه‌ی while خطوط ۳-۸ مکرراً یک مسیر تکمیلی p در G_f یافته و شار f را روی p به اندازه‌ی ظرفیت پس‌ماند $c_f(p)$ افزایش می‌دهد. هر یال پس‌ماند در مسیر p یا یک یال در شبکه‌ی اصلی است و یا عکس یک یال در شبکه‌ی اصلی. خطوط ۶-۸ شار را در هر وضعیت به شکل مناسب به هنگام سازی می‌کنند، یعنی وقتی یال پس‌ماند مربوطه یک یال اصلی است، شار را افزایش، و در غیر این صورت آن را کاهش می‌دهند. وقتی هیچ مسیر تکمیلی وجود نداشته باشد شار f یک شار بیشینه است.

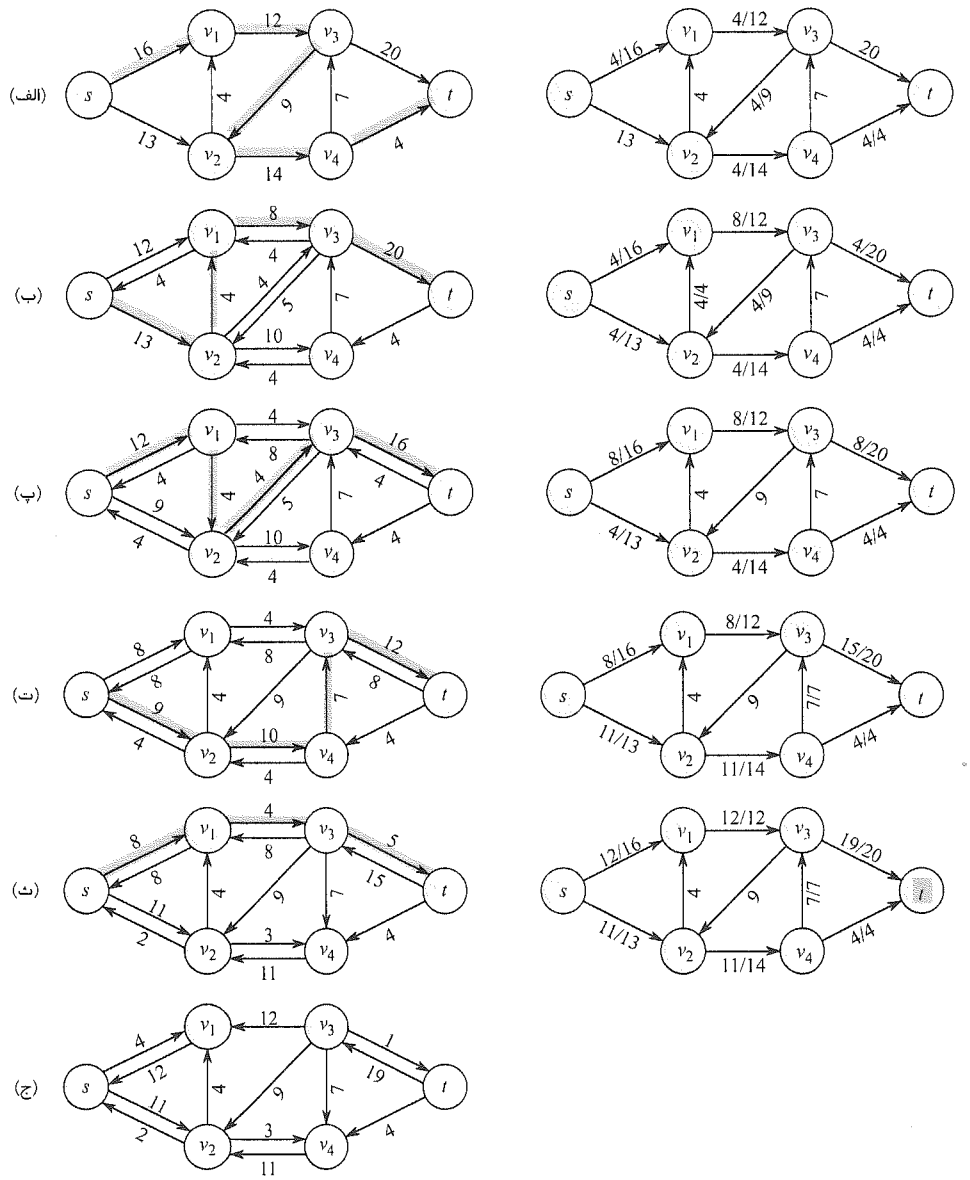
تحلیل فورد- فولکرسن

زمان اجرای FORD-FULKERSON به نحوه‌ی پیدا کردن مسیر تکمیلی p در خط ۳ بستگی دارد. اگر این کار به صورت ضعیف انجام شود، ممکن است الگوریتم حتی پایان هم نیابد: مقدار شار با تکمیل‌های پشت سر هم افزایش می‌یابد، ولی نیازی نیست که به مقدار بیشینه میل کند.[2] با این حال

---

[1] از بخش ۲۲-۱ به خاطر بیاورید که یک خصیصه‌ی f را برای یال (u, v) با همان نمادی $f(u, v)$ – نشان می‌دهیم که خصیصه‌های هر شیئ دیگری را نمایش می‌دهیم.

[2] متد فورد–فولکرسن فقط زمانی ممکن است به پایان نرسد که ظرفیت یال‌ها اعداد گنگ باشند.



شکل ۲۶-۶ اجرای الگوریتم اولیه‌ی فورد- فولکرسن. (الف)-(ث) تکرارهای پشت سر هم حلقه‌ی **while**. سمت چپ هر بخش شبکه‌ی پس‌ماند G_f از خط ۴ را با یک مسیر تکمیلی سایه‌دار p نشان می‌دهد. سمت راست هر بخش شار خالص f را که از اضافه کردن f_p به f حاصل می‌شود، نشان می‌دهد. شبکه‌ی پس‌ماند در (الف) همان شبکه‌ی ورودی G است. (ج) شبکه‌ی پس‌ماند در آخرین تست حلقه‌ی **while**. این شبکه هیچ مسیر تکمیلی ندارد، و بنابراین شار f نشان داده شده در (ث) یک شار بیشینه است. مقدار شار بیشینه‌ی پیدا شده ۲۳ است.

اگر مسیر تکمیلی با استفاده از یک جستجوی سطح اول (که در بخش ۲۲-۲ دیدیم) انتخاب شود، الگوریتم در زمان چندجمله‌ای اجرا می‌شود. در هر حال قبل از اثبات این نتیجه، یک کران ساده برای حالتی که مسیر تکمیلی به صورت دلخواه انتخاب می‌شود و تمام ظرفیت‌ها عدد صحیح هستند، به دست می‌آوریم.

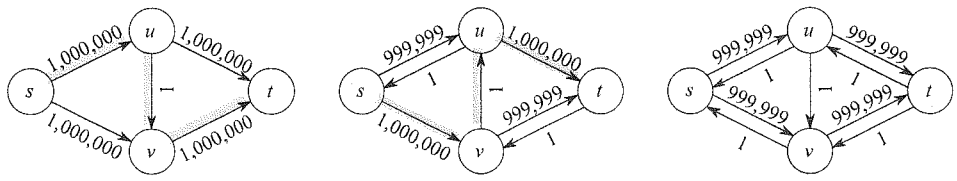
در عمل در اکثر مسئله‌های شار بیشینه ظرفیت‌ها عدد صحیح هستند. اگر ظرفیت‌ها اعداد گویا باشند، می‌توان از یک تبدیل واحد مناسب برای تبدیل همه‌ی آن‌ها به اعداد صحیح استفاده کرد. اگر f^* یک شار بیشینه در شبکه‌ی تبدیل شده باشد، یک پیاده‌سازی سرراست از FORD-FULKERSON، حلقه‌ی while خطوط ۳-۸ را حداکثر $|f^*|$ بار اجرا می‌کند، چرا که مقدار شار در هر تکرار حداقل یک واحد افزایش می‌یابد.

اگر به صورت بهینه ساختمان‌های داده‌ی استفاده شده برای پیاده‌سازی شبکه‌ی $G = (V, E)$ را اداره کنیم و مسیر تکمیلی را با استفاده از یک الگوریتم خطی بیابیم، می‌توانیم کار انجام شده در حلقه‌ی while را بهینه کنیم. اجازه دهید فرض کنیم که یک ساختمان داده متناظر با یک گراف جهت‌دار $G' = (V, E')$ نگه می‌داریم، که در آن

$$E' = \{(u, v) : (u, v) \in E \text{ یا } (v, u) \in E\}$$

یال‌های شبکه‌ی G یال‌های گراف G' هم هستند، و بنابراین نگه‌داری ظرفیت‌ها و شارها در ساختمان داده کار ساده‌ای است. با داشتن یک شار f برای G، شبکه‌ی پس‌ماند G_f شامل تمام یال‌های (u, v) درون G' است به طوری که $c_f(u, v) > 0$، که در آن c_f با تساوی (۲۶-۲) تطابق دارد. بنابراین اگر از جستجوی عمق اول یا جستجوی سطح اول استفاده کنیم، زمان یافتن یک مسیر در شبکه‌ی پس‌ماند $O(V + E') = O(E)$ خواهد بود. پس هر تکرار حلقه‌ی while به $O(E)$ زمان نیاز دارد. همین طور است مقداردهی اولیه‌ی خطوط ۱-۲، که طبق آن کل زمان اجرای FORD-FOLKERSON برابر $O(E|f^*|)$ خواهد شد.

وقتی ظرفیت‌ها صحیح باشند و مقدار شار مؤثر $|f^*|$ کوچک باشد، زمان اجرای الگوریتم فورد-فولکرسن مناسب خواهد بود. شکل ۲۶-۷(الف) مثالی از یک شبکه‌ی شار ساده را نشان می‌دهد که در آن $|f^*|$ بزرگ است. مقدار یک شار بیشینه در این شبکه ۲,۰۰۰,۰۰۰ است: ۱,۰۰۰,۰۰۰ واحد شار از مسیر $s \to u \to t$ می‌گذرد، و ۱,۰۰۰,۰۰۰ واحد دیگر از مسیر $s \to v \to t$. اگر اولین مسیر تکمیلی یافت شده یافته شده توسط FORD-FULKERSON، $s \to u \to v \to t$ باشد، که در شکل ۲۶-۷(الف) نشان داده شده است، مقدار شار پس از اولین تکرار ۱ خواهد بود. شبکه‌ی پس‌ماند حاصل در شکل ۲۶-۷(ب) نشان داده شده است. اگر تکرار دوم مسیر تکمیلی $s \to v \to u \to t$ را پیدا کند، همان طور که در شکل ۲۶-۶(ب) نشان داده شده است، شار مقدار ۲ خواهد داشت. شکل ۲۶-۷(پ) شبکه‌ی پس‌ماند حاصل را نشان می‌دهد. می‌توانیم همین طور با انتخاب مسیر تکمیلی $s \to v \to u \to t$ در تکرارهای با شماره‌ی فرد و انتخاب مسیر تکمیلی $s \to u \to v \to t$ در انتخاب‌های با شماره‌ی زوج ادامه دهیم. در این صورت در کل ۲,۰۰۰,۰۰۰ بار تکمیل شار انجام خواهیم داد، که هر بار شار به اندازه‌ی ۱ واحد افزایش خواهد یافت.



شکل ۲۶-۷ (الف) یک شبکه‌ی شار که در آن FORD-FULKERSON می‌تواند $\theta(E|f^*|)$ زمان بگیرد، که در آن f^* یک شار بیشینه است، که در این جا به صورت $|f^*| = ۲,۰۰۰,۰۰۰$ نشان داده شده است. یک مسیر تکمیلی با ظرفیت پس‌ماند ۱ به صورت سایه‌دار نشان داده شده است. (ب) شبکه‌ی پس‌ماند حاصل با یک مسیر تکمیلی دیگر با ظرفیت پس‌ماند ۱. (پ) شبکه‌ی پس‌ماند حاصل.

الگوریتم ادمونسون-کارپ

اگر محاسبه‌ی مسیر تکمیلی p در خط ۳ رویه‌ی FORD-FULKERSON را با استفاده از جستجوی سطح اول پیاده‌سازی کنیم می‌توانیم زمان اجرای آن را بهبود بخشیم، یعنی اگر مسیر تکمیلی یک *کوتاه‌ترین* مسیر از s به t در شبکه‌ی پس‌ماند باشد، که در آن هر یال فاصله‌ی (وزن) واحد دارد. به متد فورد-فولکرسن که بدین صورت پیاده‌سازی شده باشد، *الگوریتم ادمونسون-کارپ* (Edmonson-Karp algorithm) می‌گوییم. اکنون اثبات می‌کنیم که الگوریتم ادمونسون-کارپ در زمان $O(VE^2)$ اجرا می‌شود.

تحلیل به فاصله‌ی رأس‌ها در شبکه‌ی پس‌ماند G_f بستگی دارد. لم زیر از نماد $\delta_f(u,v)$ برای فاصله‌ی کوتاه‌ترین مسیر از u به v در G_f استفاده می‌کند، که در آن هر یال فاصله‌ی واحد دارد.

لم ۲۶-۷ اگر الگوریتم ادمونسون-کارپ بر روی یک شبکه‌ی شار $G=(V,E)$ با منبع s و چاهک t اجرا شود، آن گاه برای تمام رأس‌های $v \in V - \{s,t\}$، فاصله‌ی کوتاه‌ترین مسیر $\delta_f(u,v)$ در شبکه‌ی پس‌ماند G_f با هر تکمیل شار به صورت یکنواخت افزایش می‌یابد.

اثبات فرض خواهیم کرد که برای یک رأس $v \in V - \{s,t\}$، یک شار تکمیلی وجود دارد که باعث می‌شود فاصله‌ی کوتاه‌ترین مسیر از s به v کاهش یابد، و سپس به یک تناقض خواهیم رسید. فرض کنید f شار مربوط به زمان دقیقاً قبل از اولین تکمیلی باشد که فاصله‌ی یک کوتاه‌ترین مسیر را کاهش می‌دهد، و f' شار مربوط به زمان دقیقاً بعد از آن باشد. فرض کنید v رأس با کمینه‌ی $\delta_{f'}(s,v)$ باشد که فاصله‌ی آن با تکمیل کاهش یافته است، به طوری که $\delta_{f'}(s,v) < \delta_f(s,v)$. فرض کنید $p = s \rightsquigarrow u \rightarrow v$ یک کوتاه‌ترین مسیر از s به v در $G_{f'}$ باشد، به طوری که $(u,v) \in E_{f'}$ و

$$\delta_{f'}(s,u) = \delta_{f'}(s,v) - ۱ \qquad (۱۲-۲۶)$$

به دلیل نحوه‌ی انتخاب v، می‌دانیم که فاصله‌ی رأس u از منبع s کاهش نیافته است، یعنی

$$\delta_{f'}(s,u) \geq \delta_f(s,u) \qquad (۱۳-۲۶)$$

ادعا می‌کنیم که $(u,v) \notin E_f$. چرا؟ اگر داشته باشیم $(u,v) \in E$، آن گاه همچنین خواهیم داشت

$$\begin{aligned}
\delta_f(s,v) &\leq \delta_f(s,u)+1 && \text{(طبق لم ۲۴-۱۰ و نامساوی مثلث)}\\
&\leq \delta_{f'}(s,u)+1 && \text{(طبق نامساوی (۲۶-۱۳))}\\
&= \delta_{f'}(s,v) && \text{(طبق تساوی (۲۶-۱۲))}
\end{aligned}$$

که با فرض $\delta_{f'}(s,v) < \delta_f(s,v)$ تناقض دارد.

چطور می‌توانیم هم داشته باشیم $(u,v) \notin E_f$ و هم $(u,v) \in E_{f'}$؟ تکمیل باید شار را از v به u افزایش داده باشد. الگوریتم ادمونسون-کارپ همیشه شار را در طول کوتاه‌ترین مسیرها تکمیل می‌کند، و بنابراین آخرین یال کوتاه‌ترین مسیر از s به u در G_f یال (v,u) است. پس،

$$\begin{aligned}
\delta_f(s,v) &= \delta_f(s,u)-1\\
&\leq \delta_{f'}(s,u)-2 && \text{(طبق نامساوی (۲۶-۱۳))}\\
&= \delta_{f'}(s,v)-2 && \text{(طبق تساوی (۲۶-۱۲))}
\end{aligned}$$

که با فرض $\delta_{f'}(s,v) < \delta_f(s,v)$ تناقض دارد. نتیجه می‌گیریم که فرض ما که یک رأس v وجود دارد صحیح نیست. ■

قضیه‌ی بعد تعداد تکرارهای الگوریتم ادمونسون-کارپ را محدود می‌کند.

قضیه ۲۶-۸ اگر الگوریتم ادمونسون-کارپ بر روی یک شبکه‌ی شار $G=(V,E)$ با منبع s و چاهک t اجرا شود، آن گاه کل تعداد تکمیل‌های انجام شده بر روی شار توسط الگوریتم، $O(VE)$ است.

اثبات می‌گوییم یک یال (u,v) در یک شبکه‌ی پس‌ماند G_f بر روی مسیر تکمیلی p *بحرانی* (critical) است اگر ظرفیت پس‌ماند p برابر ظرفیت پس‌ماند (u,v) باشد، یعنی اگر $c_f(p)=c_f(u,v)$. پس از این که شار را بر روی یک مسیر تکمیلی افزایش دادیم، هر یال بحرانی روی مسیر از شبکه‌ی پس‌ماند محو می‌شود. به علاوه حداقل یک یال روی هر مسیر تکمیلی باید بحرانی باشد. نشان خواهیم داد که هر یک از $|E|$ یال می‌تواند حداکثر $|V|/2-1$ بار بحرانی شود.

فرض کنید u و v رأس‌هایی در V باشند که با یک یال در E به هم متصل شده‌اند. از آن جایی که مسیرهای تکمیلی کوتاه‌ترین مسیر هستند، وقتی (u,v) برای اولین بار بحرانی شود، داریم

$$\delta_f(u,v) = \delta_f(s,u)+1$$

وقتی شار تکمیل شد، یال (u,v) از شبکه‌ی پس‌ماند حذف می‌شود. این یال دیگر نمی‌تواند در مسیر تکمیلی دیگری ظاهر شود تا وقتی که شار از u به v کاهش یابد، که فقط در صورتی اتفاق می‌افتد که (u,v) بر روی یک مسیر تکمیلی ظاهر شود. اگر f' شار شبکه‌ی G برای زمانی باشد که این اتفاق رخ دهد، آن گاه داریم

$$\delta_{f'}(s,u) = \delta_{f'}(s,v) + 1$$

چون طبق لم ۲۶-۷ ($\delta_f(s,v) \le \delta_{f'}(s,v)$) داریم

$$\begin{aligned}\delta_{f'}(s,u) &= \delta_{f'}(s,v) + 1 \\ &\ge \delta_f(s,v) + 1 \\ &= \delta_f(s,u)\end{aligned}$$

در نتیجه، از زمانی که (u,v) بحرانی می‌شود تا زمانی که دفعه‌ی بعد دوباره بحرانی می‌شود، فاصله‌ی u از منبع حداقل به اندازه‌ی ۲ واحد افزایش می‌یابد. فاصله‌ی u از منبع در ابتدا حداقل ۰ است. رأس‌های میانی بر روی یک کوتاه‌ترین مسیر از s به u نمی‌توانند شامل s، u، یا t شوند (چرا که بودن (u,v) بر روی مسیر تکمیلی ایجاب می‌کند که $u \neq t$). بنابراین تا زمانی که u از منبع غیر قابل دسترس شود، اگر این اتفاق بیفتد، فاصله‌ی آن حداکثر $|V| - ۲$ است. پس بعد از اولین باری که (u,v) بحرانی می‌شود، می‌تواند حداکثر $|V|/۲ - ۱ = (|V| - ۲)/۲$ بار دیگر بحرانی شود، با مجموع حداکثر $|V|/۲$ بار. از آن جایی که $O(E)$ جفت رأس وجود دارد که می‌تواند در گراف پس‌مانده یک یال بین آن‌ها وجود داشته باشد، تعداد کل یال‌های بحرانی در کل زمان اجرای الگوریتم ادمونتون-کارپ $O(VE)$ است. هر مسیر تکمیلی حداقل یک یال بحرانی دارد، و بنابراین قضیه ثابت است. ■

از آن جایی که وقتی مسیرهای تکمیلی را با استفاده از جستجوی سطح اول می‌یابیم، هر تکرار FORD-FULKERSON را می‌توانیم در زمان $O(E)$ پیاده‌سازی کنیم، کل زمان اجرای الگوریتم ادمونسون-کارپ $O(VE^2)$ است. خواهیم دید که الگوریتم‌های رانش-برچسب‌دهی مجدد می‌توانند به زمان اجرای بهتری ختم شوند. الگوریتم بخش ۲۶-۴ یک متد برای رسیدن به زمان اجرای $O(V^2E)$ خواهد داد، که پایه‌ی الگوریتم با زمان $O(V^3)$ بخش ۲۶-۵ خواهد بود.

تمرین‌ها

۲۶-۲-۱ اثبات کنید که سری‌های تساوی (۲۶-۶) برابرند با سری‌های تساوی (۲۶-۷).

۲۶-۲-۲ در شکل ۲۶-۱(ب) شار بر روی برش $(\{s, v_2, v_4\}, \{v_1, v_3, t\})$ چقدر است؟ ظرفیت این برش چقدر است؟

۲۶-۲-۳ اجرای الگوریتم ادمونسون-کارپ را بر روی شبکه‌ی شار شکل ۲۶-۱(الف) نشان دهید.

۲۶-۲-۴ در مثال شکل ۲۶-۶، برش کمینه‌ی متناظر با شار بیشینه نشان داده شده چیست؟ در میان مسیرهای تکمیلی نشان داده شده در این مثال، کدام یک شار را خنثی می‌کند؟

۲۶-۲-۵ به خاطر بیاورید که در بازسازی بخش ۲۶-۱ که یک شبکه‌ی شار با چند منبع و چند چاهک را به یک شبکه‌ی شار با یک منبع و یک چاهک تبدیل می‌کند، یال‌هایی با ظرفیت بی‌نهایت اضافه می‌شوند. اثبات کنید که اگر یال‌های شبکه‌ی اصلی با چند منبع و چند چاهک ظرفیت محدود داشته باشند، هر شاری در شبکه‌ی بازسازی شده هم کران دار است.

۶-۲-۲۶ فرض کنید هر منبع s_i در یک مسئله با چند منبع و چند چاهک دقیقاً p_i واحد شار تولید می‌کند، به طوری که $\sum_{v \in V} f(s_i, v) = p_i$. همچنین فرض کنید که هر چاهک t_j دقیقاً q_j واحد مصرف می‌کند، به طوری که $\sum_{v \in V} f(v, t_j) = q_j$، که در آن $\sum_i p_i = \sum_j q_j$. نشان دهید که چگونه می‌توان مسئله‌ی یافتن یک شار f که از این محدودیت‌های اضافی پیروی می‌کند را به مسئله‌ی یافتن یک شار بیشینه در یک مسئله‌ی با یک منبع و یک چاهک تبدیل کرد.

۷-۲-۲۶ لم ۲۶-۲ را اثبات کنید.

۸-۲-۲۶ فرض کنید شبکه‌های پس‌ماند را طوری بازتعریف می‌کنیم که در آن‌ها یال‌های ورودی به s غیر مجاز هستند. بحث کنید که رویه‌ی FORD-FULKERSON باز هم به درستی یک شار بیشینه را محاسبه می‌کند.

۹-۲-۲۶ فرض کنید که f و f' شارهایی هستند در یک شبکه‌ی G، و ما شار $f \uparrow f'$ را محاسبه می‌کنیم. آیا شار تکمیلی، خاصیت بقای شار را ارضا می‌کند؟ محدودیت ظرفیت را چطور؟

۱۰-۲-۲۶ نشان دهید که چطور می‌توان یک شار بیشینه در یک شبکه‌ی $G = (V, E)$ را با دنباله‌ای از حداکثر $|E|$ مسیر تکمیلی یافت. (*راهنمایی:* مسیرها را بعد از یافتن شار بیشینه تعیین کنید.)

۱۱-۲-۲۶ *همبندی یالی* (edge connectivity) در یک گراف بدون جهت برابر است با کمینه‌ی تعداد k از یال‌ها که باید حذف شوند تا گراف ناهمبند شود. به عنوان مثال، همبندی یالی یک درخت ۱ است، و همبندی یالی یک زنجیره‌ی دایره‌ای از رأس‌ها ۲ است. نشان دهید چگونه می‌توان همبندی یالی یک گراف جهت‌دار $G = (V, E)$ را با استفاده از اجرای یک الگوریتم شار بیشینه بر روی حداکثر $|V|$ شبکه‌ی شار، که هر یک $O(V)$ رأس و $O(E)$ یال دارند تعیین کرد.

۱۲-۲-۲۶ فرض کنید که یک شبکه‌ی شار $G = (V, E)$ یال‌های متقارن دارد، یعنی $(u, v) \in E$ اگر و فقط اگر $(v, u) \in E$. نشان دهید که الگوریتم ادمونسون-کارپ روی این گراف حداکثر بعد از $|V||E|/4$ تکرار پایان می‌یابد. (*راهنمایی:* برای هر یال (u, v)، در نظر بگیرید که $\delta(s, u)$ و $\delta(v, t)$ بین زمان‌هایی که (u, v) بحرانی است چگونه تغییر می‌کنند.)

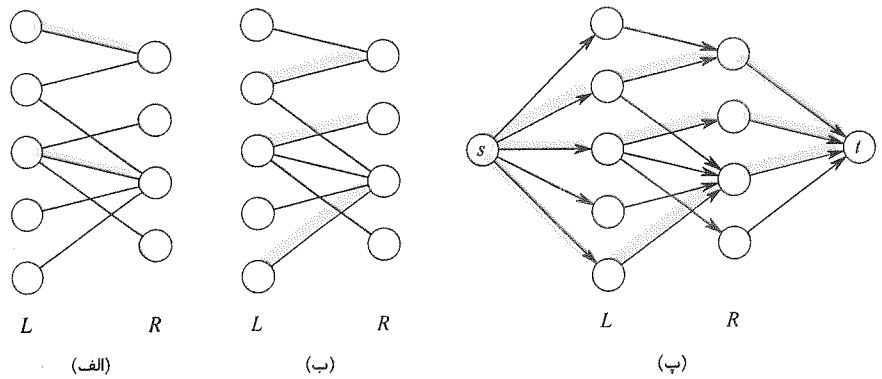
۱۳-۲-۲۶ فرض کنید می‌خواهیم از میان تمام برش‌های کمینه در یک شبکه‌ی شار G، برشی را بیابیم که کم‌ترین تعداد یال‌ها را دارد. نشان دهید که چطور می‌توان ظرفیت‌های G را اصلاح کرد و یک شبکه‌ی شار G' ساخت به طوری که یک برش کمینه در G'، یک برش کمینه با کم‌ترین تعداد یال‌ها در G باشد.

۲۶-۳ تطابق دوبخشی بیشینه

بعضی مسائل ترکیبیاتی وجود دارند که می‌توان به سادگی آن‌ها را به مسئله‌های شار بیشینه تبدیل کرد. مسئله‌ی شار بیشینه با چند منبع و چند چاهک بخش ۲۶-۱ نمونه‌ای از آن‌ها است. مسئله‌های ترکیبیاتی دیگری هستند که در ظاهر به نظر رابطه‌ی چندانی با شبکه‌های شار ندارند، ولی در واقع قابل تبدیل به مسئله‌ی شار بیشینه هستند. در این بخش نمونه‌ای از این مسائل معرفی می‌شود: یافتن یک تطابق بیشینه در یک گراف دوبخشی. برای حل این مسئله از یک خصوصیت صحیح بودن که متد فورد-فولکرسن به وجود می‌آورد، سود خواهیم برد. همچنین خواهیم دید که می‌توان از متد فورد-فولکرسن برای حل مسئله‌ی تطابق دوبخشی کمینه بر روی یک گراف $G = (V, E)$ در زمان $O(VE)$ استفاده کرد.

مسئله‌ی تطابق دوبخشی بیشینه

با داشتن یک گراف بدون جهت $G = (V, E)$، یک *تطابق* (matching) زیرمجموعه‌ای از یال‌های $M \subseteq E$ است به طوری که برای تمام رأس‌های $v \in V$، حداکثر یک یال از M با v مجاور باشد. می‌گوییم یک رأس $v \in V$ توسط تطابق M *مطابقت* یافته است اگر یالی در M وجود داشته باشد که با v مجاور باشد؛ در غیر این صورت v مطابقت نیافته است. یک تطابق بیشینه، تطابقی است که اندازه‌ی آن بیشینه است، یعنی یک تطابق M به طوری که برای هر تطابق M' داشته باشیم $|M| \geq |M'|$. در این بخش توجه خود را به یافتن یک تطابق بیشینه در گراف‌های دوبخشی معطوف می‌کنیم. فرض می‌کنیم مجموعه‌ی رأس‌ها را می‌توان به دو بخش $V = L \cup R$ تقسیم بندی کرد، که در آن L و R مجزا هستند و تمام یال‌های E بین L و R حرکت می‌کنند. همچنین فرض می‌کنیم که هر رأس در V حداقل یک یال مجاور دارد. شکل ۲۶-۸ مفهوم یک تطابق را نشان می‌دهد.



شکل ۲۶-۸ یک گراف دوبخشی $G = (V, E)$ با تقسیم بندی رأس $V = L \cup R$. (الف) یک تطابق با اندازه‌ی ۲، که با یال‌های سایه‌دار مشخص شده است. (ب) یک تطابق بیشینه با اندازه‌ی ۳. (پ) شبکه‌ی شار متناظر G' با یک شار بیشینه. هر یال، ظرفیت واحد دارد. شار یال‌های سایه‌دار ۱، و شار بقیه‌ی یال‌ها ۰ است. یال‌های سایه‌دار از L به R متناظرند با یال‌های تطابق بیشینه در (ب).

مسئله‌ی یافتن یک تطابق بیشینه در یک گراف دوبخشی کاربردهای زیادی دارد. به عنوان یک مثال، ممکن است بخواهیم مجموعه‌ی L از ماشین‌ها را با مجموعه‌ی R از وظایف تطابق دهیم که به صورت هم زمان انجام شوند. وجود یال (u,v) در E را بدین صورت معنی می‌کنیم که یک ماشین خاص $u \in L$ قادر به انجام یک وظیفه‌ی خاص $v \in R$ است. یک تطابق بیشینه برای بیشترین ماشین‌های ممکن کار فراهم می‌کند.

یافتن یک تطابق دوبخشی بیشینه

می‌توانیم از متد فورد- فولکرسن برای یافتن یک تطابق بیشینه در یک گراف بدون جهت دوبخشی $G = (V, E)$ در زمان چند جمله‌ای نسبت به $|V|$ و $|E|$ استفاده کنیم. نکته‌ی مهم ساختن یک شبکه‌ی شار است که در آن شارها متناظر با تطابق‌ها باشند، همان طور که در شکل ۲۶-۸ نشان داده شده است.*شبکه‌ی شار متناظر* $G' = (V', E')$ را برای گراف دوبخشی G به صورت زیر تعریف می‌کنیم. منبع s و چاهک t را رأس‌های جدیدی فرض می‌کنیم که در V نیستند، و قرار می‌دهیم $V' = V \cup \{s, t\}$. اگر تقسیم‌بندی رأس‌ها در G به صورت $V = L \cup R$ باشد، یال‌های جهت‌دار G' همان یال‌های E هستند که جهت آن‌ها از L به R است، به همراه V یال جهت‌دار جدید:

$$E' = \{(s,u) : u \in L\} \cup \{(u,v) : u \in L, v \in R \text{ و } (u,v) \in E\} \cup \{(v,t) : v \in R\}$$

برای تکمیل ساختمان داده به هر یک از یال‌های E' ظرفیت واحد می‌دهیم. از آن جایی که هر رأس در V حداقل یک یال مجاور دارد داریم $|E| \geq |V|/2$. بنابراین $|E| \leq |E'| = |E| + |V| \leq 3|E|$ و $|E'| = \theta(E)$.

لم زیر نشان می‌دهد که یک تطابق در G مستقیماً متناظر است با یک شار در شبکه‌ی شار G، یا همان G'. می‌گوییم یک شار f در یک شبکه‌ی شار $G = (V, E)$ *صحیح مقدار* (integer-valued) است اگر $f(u,v)$ برای تمام $(u,v) \in V \times V$ یک عدد صحیح باشد.

لم ۲۶-۱ فرض کنید $G = (V, E)$ یک گراف دوبخشی با تقسیم‌بندی رأس $V = L \cup R$ باشد، و $G' = (V', E')$ شبکه‌ی شار متناظر آن. اگر M یک تطابق در G باشد، آن گاه یک شار صحیح مقدار f در G' با مقدار $|f| = |M|$ وجود دارد. بالعکس، اگر f یک شار صحیح مقدار در G' باشد، آن گاه یک تطابق M در G با اندازه‌ی $|M| = |f|$ وجود دارد.

اثبات ابتدا نشان می‌دهیم که یک تطابق M در G متناظر است با یک شار صحیح مقدار در G'. f را به صورت زیر تعریف می‌کنیم. اگر $(u,v) \in M$، آن گاه $f(s,u) = f(u,v) = f(v,t) = 1$. برای تمام یال‌های دیگر $(u,v) \in E'$، تعریف می‌کنیم $f(u,v) = 0$. به سادگی می‌توان چک کرد که f خصوصیات محدودیت ظرفیت و بقای شار را ارضا می‌کند.

به صورت شهودی، هر یال $(u,v) \in M$ متناظر است با ۱ واحد شار در G' که مسیر

$s \to u \to v \to t$ را طی می‌کند. به علاوه مسیر ساخته شده توسط یال‌های M مجزای رأسی هستند، غیر از s و t. شار خالص روی برش $(L \cup \{s\}, R \cup \{t\})$ برابر است با $|M|$؛ بنابراین طبق لم ۲۶-۴، مقدار شار برابر است با $|f| = |M|$.

برای اثبات حالت برعکس، فرض کنید f یک شار صحیح مقدار در G' باشد، و فرض کنید

$$M = \{(u, v) : u \in L, v \in R, \text{و } f(u, v) > 0\}$$

هر رأس $u \in L$ فقط یک یال ورودی دارد، که (s, u) است با ظرفیت ۱. بنابراین به هر $u \in L$ حداکثر یک واحد شار مثبت وارد می‌شود، و اگر این یک شار مثبت وارد شود، طبق بقای شار، یک واحد شار مثبت هم باید خارج شود. علاوه بر این از آن جایی که f صحیح مقدار است، برای هر $u \in L$ یک واحد شار حداکثر می‌تواند وارد یک یال شود و حداکثر می‌تواند از یک یال خارج شود. بنابراین یک واحد شار مثبت به u وارد می‌شود اگر و فقط اگر دقیقاً یک رأس $v \in R$ وجود داشته باشد به طوری که $f(u, v) = 1$، و حداکثر یک یال خروجی از $u \in L$ شار مثبت حمل کند. یک بحث متقارن را می‌توان برای هر $v \in R$ انجام داد. بنابراین مجموعه‌ی M تعریف شده در صورت لم یک تطابق است.

برای این که ببینیم $|f| = |M|$، مشاهده کنید که برای هر رأس مطابقت یافته‌ی $u \in L$ داریم $f(s, u) = 1$، و برای هر یال $(u, v) \in E - M$ داریم $f(u, v) = 0$. در نتیجه، $f(L \cup \{s\}, R \cup \{t\})$، شار خالص عبوری از برش $(L \cup \{s\}, R \cup \{t\})$، برابر است با $|M|$. با به کار بردن لم ۲۶-۴ داریم $|f| = f(L \cup \{s\}, R \cup \{t\}) = |M|$. ▒

بر پایه‌ی لم ۲۶-۹، مطلوب است که نتیجه بگیریم یک تطابق بیشینه در یک گراف دوبخشی G متناظر است با یک شار بیشینه در شبکه‌ی شار متناظر G'، و بنابراین می‌توانیم با اجرای یک الگوریتم شار بیشینه روی G'، یک تطابق بیشینه در G پیدا کنیم. تنها مشکل در این استدلال این است که الگوریتم شار بیشینه ممکن است یک شار در G' بازگرداند که $f(u, v)$ یک عدد صحیح نباشد، در حالی که مقدار شار $|f|$ باید یک عدد صحیح باشد. قضیه‌ی زیر نشان می‌دهد که اگر از متد فورد-فولکرسن استفاده کنیم، این مشکل به وجود نخواهد آمد.

قضیه‌ی ۲۶-۱۰ (قضیه‌ی صحیح بودن)

اگر تابع ظرفیت c فقط مقادیر صحیح بگیرد، آن گاه شار بیشینه‌ی f تولید شده توسط متد فورد- فولکرسن این خصوصیت را دارد که $|f|$ صحیح مقدار است. به علاوه برای تمام رأس‌های u و v، مقدار $f(u, v)$ یک عدد صحیح است.

اثبات اثبات به کمک استقرا بر روی تعداد تکرارها است، که آن را به عنوان تمرین ۲۶-۳-۲ واگذار می‌کنیم. ▒

اکنون نتیجه‌ی زیر را برای لم ۲۶-۹ اثبات می‌کنیم.

نتیجه‌ی ۲۶-۱۱ اندازه‌ی یک تطابق بیشینه‌ی M در یک گراف دوبخشی G برابر است با مقدار یک شار بیشینه‌ی f در شبکه‌ی شار متناظر G'.

اثبات از نام گذاری‌های لم ۲۶-۹ استفاده می‌کنیم. فرض کنید M یک تطابق بیشینه در G باشد، ولی شار متناظر f در G' بیشینه نباشد. در این صورت یک شار بیشینه‌ی f' در G' وجود دارد به طوری که $|f'| > |f|$. از آن جایی که ظرفیت‌ها در G' مقادیر صحیح هستند، طبق قضیه‌ی ۲۶-۱۰ می‌توانیم فرض کنیم که f' صحیح مقدار است. بنابراین f' متناظر است با یک تطابق M' در G' با اندازه‌ی $|M| = |f| < |f'| = |M'|$، که با فرض بیشینه بودن تطابق M تناقض دارد. با روشی مشابه می‌توانیم نشان دهیم که اگر f یک شار بیشینه در G' باشد، تطابق متناظر آن یک تطابق بیشینه در G است. ■

بنابراین با داشتن یک گراف بدون جهت دوبخشی در G، می‌توانیم با ساختن شبکه‌ی شار G'، اجرای متد فورد- فولکرسن، و به دست آوردن یک تطابق M از شار بیشینه‌ی صحیح مقدار یافته شده‌ی f ، یک تطابق بیشینه در گراف بیابیم. از آن جایی که اندازه‌ی هر تطابقی در یک گراف دوبخشی حداکثر $\min(L,R) = O(V)$ است، مقدار یک شار بیشینه در G' برابر است با $O(V)$. بنابراین می‌توانیم در یک گراف دوبخشی در زمان $O(VE') = O(VE)$ یک تطابق بیشینه بیابیم، چرا که $|E'| = \theta(E)$.

تمرین‌ها

۲۶-۳-۱ الگوریتم فورد-فولکرسن را بر روی شبکه‌ی شار شکل ۲۶-۸(پ) اجرا کنید و شبکه‌ی پس‌ماند را بعد از هر بار تکمیل شار نشان دهید. رأس‌های L را از بالا به پایین با ۱ تا ۵، و رأس‌های R را از بالا به پایین با ۶ تا ۹ شماره‌گذاری کنید. برای هر تکرار، مسیر تکمیلی را برگزینید که به ترتیب الفبایی کوچک‌تر است.

۲۶-۳-۲ قضیه‌ی ۲۶-۱۰ را اثبات کنید.

۲۶-۳-۳ فرض کنید $G = (V, E)$ یک گراف دوبخشی با تقسیم‌بندی رأس $V = L \cup R$ باشد، و G شبکه‌ی شار متناظر با آن. یک کران بالای خوب برای طول هر مسیر تکمیلی یافت شده در G' در طول اجرای FORD-FULKERSON ارائه کنید.

★ ۲۶-۳-۴ یک *تطابق کامل* (perfect matching)، تطابقی است که در آن تمام رأس‌ها مطابقت یافته‌اند. فرض کنید $G = (V, E)$ یک گراف بدون جهت دوبخشی باشد با تقسیم‌بندی رأس $V = L \cup R$، که در آن $|L| = |R|$. برای هر $X \subseteq V$، *همسایگی* (neighborhood) X را

به صورت

$$N(X) = \{x \in V \mid (x,y) \in E \text{ داریم } x \in X \text{ برای یک }\}$$

تعریف می‌کنیم، یعنی مجموعه‌ی رأس‌هایی که با یکی از اعضای X مجاور هستند. *قضیه‌ی هال* (Hall's theorem) را اثبات کنید: یک تطابق کامل در G وجود دارد اگر و فقط اگر برای تمام زیرمجموعه‌های $A \subseteq L$ داشته باشیم $|A| \le |N(A)|$.

★ ۵-۳-۲۶ می‌گوییم یک گراف دوبخشی $G = (V, E)$، که در آن $V = L \cup R$، *d-منظم* (d-regular) است اگر درجه‌ی تمام رأس‌های $v \in V$ دقیقاً d باشد. در هر گراف دوبخشی d-منظم داریم $|R| = |L|$. اثبات کنید که هر گراف دوبخشی d-منظم یک تطابق با اندازه‌ی $|L|$ دارد. برای این کار، بحث کنید که یک برش کمینه در شبکه‌ی شار متناظر دارای ظرفیت $|L|$ است.

۴-۲۶ الگوریتم‌های رانش-برچسب‌دهی مجدد

در این بخش رویکرد «رانش-برچسب‌دهی مجدد» را برای محاسبه‌ی شار بیشینه ارائه می‌کنیم. امروزه بسیاری از سریع‌ترین الگوریتم‌های حدی شار بیشینه الگوریتم‌های رانش-برچسب‌دهی مجدد، و سریع‌ترین پیاده‌سازی‌های الگوریتم‌های شار بیشینه بر مبنای متد رانش-برچسب‌دهی مجدد هستند. مسئله‌های شار دیگر، مانند مسئله‌ی شار با هزینه‌ی بیشینه را هم می‌توان با استفاده از متدهای رانش-برچسب‌دهی مجدد به صورت بهینه حل کرد. در این بخش الگوریتم «عام» شار بیشینه‌ی گلدبرگ معرفی خواهد شد، که یک پیاده‌سازی ساده دارد که در زمان $O(V^2E)$ اجرا می‌شود، و نسبت به کران $O(VE^2)$ الگوریتم ادمونسون-کارپ یک بهبود محسوب می‌شود. در بخش ۲۶-۵ الگوریتم عام را تصحیح می‌کنیم تا به یک الگوریتم رانش-برچسب‌دهی مجدد دیگر با زمان $O(V^3)$ برسیم.

روش کار الگوریتم‌های رانش-برچسب‌دهی مجدد نسبت به متد فورد-فولکرسن بیشتر محلی است. به جای بررسی کل شبکه‌ی پس‌ماند برای یافتن یک مسیر تکمیلی، الگوریتم‌های رانش-برچسب‌دهی مجدد هر بار بر روی یک رأس کار می‌کنند، و فقط همسایگی آن رأس را در شبکه‌ی پس‌ماند بررسی می‌کنند. به علاوه برخلاف متد فورد-فولکرسن، الگوریتم‌های رانش-برچسب‌دهی مجدد در طول اجرای خود خصوصیت بقای شار را حفظ نمی‌کنند. با این حال یک *پیش‌شار* (preflow) نگه می‌دارند، که یک تابع $f : V \times V \to \mathbb{R}$ است که محدودیت‌های ظرفیت، و نسخه‌ی ساده شده‌ی زیر را از بقای شار حفظ می‌کند:

$$\sum_{v \in V} f(v,u) - \sum_{v \in V} f(u,v) \ge 0$$

برای تمام رأس‌های $u \in V - \{s\}$. یعنی شار ورودی به یک رأس ممکن است از شار خروجی آن رأس فراتر رود. کمیت

$$e(u) = \sum_{v \in V} f(v,u) - \sum_{v \in V} f(u,v)$$

را *شار افزونی* (excess flow) به u می‌نامیم. شار افزونی در یک رأس، مقدار افزونه‌ی شار ورودی نسبت به شار خروجی است. می‌گوییم یک رأس $u \in V - \{s, t\}$ در حال *سرریز* (overflow) است اگر $e(u) > 0$.

این بخش را با توصیف شهود پشت متد رانش- برچسب‌دهی مجدد آغاز می‌کنیم. سپس دو عملیات استفاده شده در این متد را توضیح خواهیم داد: «راندن» پیش‌شار و «برچسب‌دهی مجدد» یک رأس. در نهایت یک الگوریتم رانش-برچسب‌دهی مجدد عام ارائه و زمان اجرای آن را تحلیل خواهیم کرد.

شهود

احتمالاً بهترین روش درک شهود پشت متد رانش-برچسب‌دهی مجدد، توصیف آن برحسب شار مایعات است: یک شبکه‌ی شار $G = (V, E)$ را به صورت سیستمی از لوله‌های متصل به یکدیگر با ظرفیت داده شده در نظر می‌گیریم. با به کار بردن این تشبیه برای متد فورد-فولکرسن، می‌توانیم بگوییم هر مسیر تکمیلی در یک شبکه معادل است با یک جریان اضافی مایع، بدون نقطه‌ی انشعاب، که از منبع به چاهک سرازیر می‌شود. متد فورد-فولکرسن مکرراً جریان‌های بیشتری از شار به سیستم اضافه می‌کند تا زمانی که دیگر نتوان جریان جدیدی اضافه کرد.

الگوریتم عام رانش-برچسب‌دهی مجدد شهود متفاوتی دارد. مانند قبل یال‌های جهت‌دار متناظر با لوله‌ها هستند. رأس‌ها که انشعابات لوله‌ها هستند، دارای دو خصوصیت جالب توجه هستند. اول، برای تطبیق شار افزونی هر رأس یک لوله‌ی خروجی دارد که به یک مخزن به دلخواه بزرگ متصل است و می‌تواند مایع را در خود ذخیره کند. دوم، هر رأس، مخزن آن، و تمام اتصالات لوله‌ی آن بر روی یک زمینه هستند که با پیش روی اجرای الگوریتم ارتفاع آن افزایش می‌یابد.

ارتفاع رأس‌ها نحوه‌ی راندن شار را تعیین می‌کنند: شار را فقط در مسیرهای سرپایینی می‌رانیم، یعنی از یک رأس بالاتر به یک رأس پایین‌تر. شار از یک رأس پایین‌تر به یک رأس بالاتر ممکن است مثبت باشد، ولی اعمالی که شار را می‌رانند فقط در جهت پایین این کار را انجام می‌دهند. ارتفاع منبع $|V|$ است، و ثابت، و ارتفاع چاهک ۰ است. ارتفاع تمام رأس‌های دیگر با ۰ آغاز می‌شود و با زمان افزایش می‌یابد. در ابتدا الگوریتم به اندازه‌ی ممکن شار از منبع به چاهک در مسیر پایینی می‌فرستد. میزان شار فرستاده شده دقیقاً به اندازه‌ای است که تمام لوله‌های خروجی از مبدأ را به اندازه‌ی ظرفیتشان پر کند؛ یعنی به اندازه‌ی ظرفیت برش $(s, V - s)$ شار می‌فرستد. وقتی شار برای اولین بار به یک رأس میانی وارد می‌شود، در مخزن آن رأس جمع آوری می‌شود. سرانجام از آن جا به سمت پایین سرازیر خواهد شد.

ممکن است زمانی اتفاق بیفتد که تنها لوله‌هایی که از u خارج می‌شوند و با شار پر نشده‌اند، به رأس‌هایی متصل باشند که هم سطح با u و یا بالاتر از آن باشند. در این حالت برای رهایی یک رأس در حال سرریز u از شار افزونی آن، باید ارتفاع آن را با یک عملیات به نام «برچسب‌دهی مجدد» رأس u افزایش دهیم. با این عملیات، ارتفاع این رأس به اندازه‌ی یک واحد بیشتر از ارتفاع پایین‌ترین رأس

همسایه‌ای که به آن یک لوله‌ی اشباع نشده دارد، افزایش می‌دهیم. بنابراین پس از برچسب‌دهی مجدد یک رأس، حداقل یک لوله‌ی خروجی وجود دارد که می‌توان شار را از آن لوله به بیرون راند.

نهایتاً کل شاری که می‌تواند به چاهک برسد، به آن رسیده است. امکان رسیدن شار بیشتر نیست، چرا که لوله‌ها از محدودیت ظرفیت پیروی می‌کنند؛ میزان شار در طول هر برش همچنان به وسیله‌ی ظرفیت برش محدود شده است. سپس برای تبدیل پیش‌شار به یک شار «مجاز»، الگوریتم مایع افزونی ذخیره شده در مخزن رأس‌های سرریز شده را با ادامه‌ی برچسب‌دهی مجدد آن‌ها و بالاتر بردن آن‌ها از ارتفاع $|V|$ مربوط به منبع، به منبع بازمی‌گرداند. همان طور که خواهیم دید، وقتی تمام مخازن خالی شدند پیش‌شار نه تنها یک شار «مجاز»، بلکه یک شار بیشینه است.

اعمال اولیه

در بحث قبلی دیدیم دو عملیات اصلی وجود دارند که توسط الگوریتم رانش-برچسب‌دهی مجدد اجرا می‌شوند: راندن شار افزونی از یک رأس به یکی از همسایه‌های آن و برچسب‌دهی مجدد یک رأس. قابلیت کاربرد این اعمال به ارتفاع رأس‌ها بستگی دارد، که اکنون آن را به دقت تعریف می‌کنیم. فرض کنید $G = (V, E)$ یک شبکه‌ی شار با منبع s و چاهک t باشد، و f یک پیش‌شار در G. یک تابع $h: V \to \mathbb{N}$ یک *تابع ارتفاع*[1] است اگر $h(s) = |V|$، $h(t) = 0$، و

$$h(u) \le h(v) + 1$$

برای تمام یال‌های پس‌ماند $(u, v) \in E_f$. بدون معطلی به لم زیر می‌رسیم.

لم ۲۶.۱۶ فرض کنید $G = (V, E)$ یک شبکه‌ی شار باشد، و f یک پیش‌شار G در h، و فرض کنید f یک تابع ارتفاع بر روی V باشد. برای هر دو رأس $u, v \in V$ اگر $h(u) > h(v) + 1$، آن گاه (u, v) یک یال در شبکه‌ی پس‌ماندنیست.

عملیات رانش

عملیات اولیه‌ی PUSH (u, v) را در صورتی می‌توان به کار برد که u یک رأس در حال سرریز باشد و داشته باشیم $c_f(u, v) > 0$ و $h(u) = h(v) + 1$. شبه‌کد زیر پیش‌شار f و شارهای افزونی برای u و v را به هنگام سازی می‌کند. این کد فرض می‌کند که ظرفیت‌های پس‌ماند $c_f(u, v)$ را با داشتن c و f می‌توان در زمان ثابت محاسبه کرد. شار افزونی ذخیره شده در یک رأس u در خصیصه‌ی $u.e$ نگه

[1] در ادبیات کامپیوتر معمولاً به یک تابع ارتفاع، یک «تابع فاصله»، و به ارتفاع یک رأس یک «برچسب فاصله» می‌گویند. ما از اصطلاح «ارتفاع» استفاده می‌کنیم چرا که شهود پشت الگوریتم را بهتر روشن می‌کند. با این حال همچنان به عملیات افزایش ارتفاع یک رأس، «برچسب‌دهی مجدد» می‌گوییم. ارتفاع یک رأس به فاصله‌ی آن از چاهک t مربوط است، همان طور که در جستجوی سطح اول گراف ترانهاده‌ی G^T مشخص خواهد شد.

داشته می‌شود، و ارتفاع u در خصیصه‌ی $u.h$. عبارت $\Delta_f(u,v)$ یک متغیر موقت است که میزان شاری را که می‌تواند از u به v رانده شود نگه می‌دارد.

```
PUSH(u, v)
1  // Applies when: u is overflowing, c_f(u, v) > 0, and u.h = v.h + 1.
2  // Action: Push Δ_f(u, v) = min(u.e, c_f(u, v)) units of flow from u to v.
3  Δ_f(u, v) = min(u.e, c_f(u, v))
4  if (u,v) ∈ E
5      (u, v).f = (u, v).f + Δ_f(u, v)
6  else (v, u).f = (u, v).f − Δ_f(u, v)
7  u.e = u.e − Δ_f(u, v)
8  v.e = v.e + Δ_f(u, v)
```

کد PUSH به صورت زیر کار می‌کند. چون رأس u شار افزونی مثبت $u.e$ دارد، و ظرفیت پس‌ماند (u,v) مثبت است، می‌توانیم شار از u به v را به اندازه‌ی $d_f(u,v) = \min(u.e, c_f(u,v))$ افزایش دهیم بدون این که باعث شویم $u.e$ منفی شود و یا ظرفیت $c(u,v)$ نقض شود. خط ۳ مقدار $\Delta_f(u,v)$ را محاسبه می‌کند، و f را در خطوط ۴-۶ به هنگام سازی می‌کنیم. خط ۵ شار یال (u,v) را افزایش می‌دهد، چرا که داریم شار را روی یک یال پس‌ماند می‌رانیم که یال اصلی هم هست. خط ۶ شار یال (v,u) را کاهش می‌دهد، چون این یال پس‌ماند در واقع معکوس یک یال در شبکه‌ی اصلی است. نهایتاً خطوط ۷-۸ شار افزونی به رأس‌های u و v را به هنگام‌سازی می‌کنند. بنابراین اگر f یک پیش‌شار قبل از فراخوانی PUSH باشد، بعد از آن هم یک پیش‌شار باقی خواهد ماند.

مشاهده کنید که هیچ چیزی در کد PUSH به ارتفاع u و v وابسته نیست، ولی باز هم از فراخوانی آن در صورتی که $u.h = v.h + 1$ برقرار نباشد جلوگیری می‌کنیم. بنابراین شار افزونی فقط با تفاوت ارتفاع ۱ به سمت پایین رانده می‌شود. طبق لم ۲۶-۱۲، هیچ یال پس‌ماندی بین دو رأسی که تفاوت ارتفاع آن‌ها بیش از ۱ است وجود ندارد، و بنابراین تا زمانی که خصیصه‌ی h یک تابع ارتفاع هم باشد، حتی اگر اجازه دهیم که شار به وسیله‌ی تفاوت ارتفاع بیش از ۱ هم به سمت پایین رانده شود، هیچ اتفاقی نمی‌افتد.

به عملیات $\text{PUSH}(u,v)$، یک *رانش* (push) از u به v می‌گوییم. اگر یک عملیات رانش بر روی یک یال (u,v) که از رأس u خارج می‌شود، اعمال شود، همچنین می‌گوییم که عملیات رانش بر روی u اعمال شده است. اگر یال (u,v) با یک راندن *اشباع* (saturate) شود (بعد از رانش داشته باشیم $c_f(u,v) = 0$)، می‌گوییم این یک *رانش اشباع کننده* است؛ در غیر این صورت این رانش، غیر اشباع کننده است. اگر یک یال اشباع شود در شبکه‌ی پس‌ماند حضور نخواهد داشت. یک لم ساده یکی از نتایج رانش‌های غیر اشباع کننده را توصیف می‌کند.

لم ۲۶.۱۴ بعد از یک رانش غیر اشباع کننده از u به v، رأس u دیگر در حال سرریز نخواهد بود.

اثبات چون رانش، غیر اشباع کننده بوده است، $(u,v)\ \Delta_f$ (میزان شاری که واقعاً رانده شده است) باید قبل از راندن برابر $u.e$ باشد. از آن جایی که $u.e$ به این اندازه کاهش یافته است، بعد از راندن ◦ می‌شود. ▒

عملیات برچسب‌دهی مجدد

عملیات اصلی RELABEL (u) زمانی کاربرد دارد که u در حال سرریز باشد و اگر برای تمام یال‌های $(u,v)\in E_f$ داشته باشیم $u.h \le v.h$. به عبارت دیگر، زمانی می‌توانیم یک رأس در حال سرریز u را برچسب‌دهی مجدد کنیم که برای هر رأس v که یک ظرفیت پس‌ماند از u به v وجود دارد، نتوان شار را از u به v راند چرا که v پایین‌تر از u نیست. (به خاطر بیاورید که طبق تعریف، نه منبع s و نه چاهک t نمی‌توانند در حال سرریز باشند، و بنابراین هیچ کدام برچسب‌دهی مجدد هم نخواهند شد.)

```
RELABEL(u)
1  // Applies when: u is overflowing and for all v ∈ V such that (u, v) ∈ E_f,
                    we have u.h ≤ v.h.
2  // Action: Increase the height of u.
3  u.h = 1 + min{v.h : (u, v) ∈ E_f}
```

وقتی عملیات RELABEL (u) را فراخوانی می‌کنیم، می‌گوییم که رأس u برچسب‌دهی مجدد شده است. توجه کنید که وقتی u برچسب‌دهی مجدد شود، E_f باید حداقل شامل یک یال خروجی از u باشد تا کمینه گیری در کد بر روی یک مجموعه‌ی تهی انجام نشود. این خصوصیت از این فرض می‌آید که u در حال سرریز است، که خود به ما می‌گوید

$$u.e = \sum_{v\in V} f(v,u) - \sum_{v\in V} f(u,v) > 0$$

از آن جایی که تمام شارها نامنفی هستند، باید حداقل یک رأس v داشته باشیم به طوری که $(v,u).f > 0$. ولی در این صورت $c_f(u,v) > 0$، که نتیجه می‌دهد $(u,v)\in E_f$. بنابراین عملیات RELABEL (u) به u بالاترین ارتفاع ممکن را می‌دهد که توسط محدودیت‌های روی توابع ارتفاع مجاز باشد.

الگوریتم عام

الگوریتم عام رانش- برچسب‌دهی مجدد از زیرروال زیر استفاده می‌کند تا یک پیش‌شار اولیه در شبکه‌ی شار بسازد.

```
INITIALIZE-PREFLOW(G, s)
1  for each vertex v ∈ G.V
2      v.h = 0
3      v.e = 0
4  for each edge (u, v) ∈ G.E
5      (u,v).f = 0
```

```
6   s.h = |G.V|
7   for each vertex u ∈ s.Adj
8       (s, v).f = c(s, v)
9       v.e = c(s, v)
10      s.e = s.e − c(s, u)
```

INITIALIZE-PREFLOW یک پیش‌شار f می‌سازد که به صورت

$$(u,v).f = \begin{cases} c(u,v) & \text{اگر } u = s \\ 0 & \text{در غیر این صورت} \end{cases} \qquad (۱۵-۲۶)$$

تعریف می‌شود. یعنی هر یال خروجی از منبع s تا ظرفیت خود پر می‌شود، و تمام یال‌های دیگر هیچ شاری حمل نمی‌کنند. برای هر رأس v مجاور با منبع، در ابتدا داریم $v.e = c(s,v)$، و $s.e$ با منفی مجموع این ظرفیت‌ها مقداردهی اولیه شده است. همچنین الگوریتم عام با یک تابع ارتفاع اولیه‌ی h شروع می‌کند، که به صورت زیر است:

$$u.h = \begin{cases} |V| & \text{اگر } u = s \\ 0 & \text{در غیر این صورت} \end{cases} \qquad (۱۶-۲۶)$$

تساوی ۲۶-۱۶ یک تابع ارتفاع تعریف می‌کند، چرا که تنها یال‌های (u,v) که برای آن‌ها داریم $u.h > v.h + 1$، آن‌هایی هستند که $u = s$، و این یال‌ها اشباع شده‌اند، که بدین معنی است که در شبکه‌ی پس‌ماند نیستند.

مقداردهی اولیه، و پس از آن دنباله‌ای از اعمال رانش و برچسب‌دهی مجدد که با هیچ ترتیب خاصی انجام نمی‌شوند، به الگوریتم GENERIC-PUSH-RELABEL ختم می‌شود.

```
GENERIC-PUSH-RELABEL(G)
1  INITIALIZE-PREFLOW(G, s)
2  while there exists an applicable push or relabel operation
3      select an applicable push or relabel operation and perform it
```

لم زیر به ما می‌گوید که تا زمانی که یک رأس در حال سرریز وجود داشته باشد، حداقل یکی از دو عملیات قابل کاربرد است.

لم ۲۶-۱۴
اگر یک رأس در حال سرریز را با هر توان رانده یا برچسب‌دهی مجدد کنیم

فرض کنید $G = (V,E)$ یک شبکه‌ی شار با منبع s و چاهک t باشد، و f یک پیش‌شار در این شبکه، و فرض کنید h یک تابع ارتفاع دلخواه برای f باشد. اگر u یک رأس در حال سرریز باشد، آن گاه یا یک عملیات رانش و یا یک عملیات برچسب‌دهی مجدد برای آن کاربرد دارد.

اثبات برای هر یال پس‌ماند (u,v) داریم $h(u) \le h(v) + 1$ چرا که h یک تابع ارتفاع است. اگر یک عملیات رانش بر روی یک رأس در حال سرریز u کاربرد نداشته باشد، آن گاه برای تمام یال‌های پس‌ماند (u,v) باید داشته باشیم $h(u) < h(v) + 1$، که ایجاب می‌کند $h(u) \le h(v)$. بنابراین می‌توان یک عملیات برچسب‌دهی مجدد بر روی u انجام داد. ■

صحت متد رانش- برچسب‌دهی مجدد

برای این که نشان دهیم الگوریتم عام رانش-برچسب‌دهی مجدد مسئله‌ی شار بیشینه را حل می‌کند، ابتدا اثبات می‌کنیم که اگر این الگوریتم پایان یابد، پیش‌شار f یک شار بیشینه است. سپس اثبات خواهیم کرد که این الگوریتم پایان می‌یابد. با مشاهداتی در مورد تابع ارتفاع h آغاز می‌کنیم.

لم ۲۶-۱۵ در طول اجرای GENERIC-PUSH-RELABEL روی یک شبکه‌ی شار $G=(V,E)$، برای هر رأس $u \in V$، ارتفاع $u.h$ هیچ گاه کاهش نمی‌یابد. علاوه بر این هر گاه که یک عملیات برچسب‌دهی مجدد بر روی یک رأس u اعمال می‌شود، ارتفاع $u.h$ حداقل به اندازه‌ی ۱ واحد افزایش می‌یابد.

اثبات چون ارتفاع رأس‌ها فقط توسط عملیات برچسب‌دهی مجدد تغییر می‌کند، کافی است بخش دوم لم را اثبات کنیم. اگر رأس u در حال برچسب‌دهی مجدد باشد، آن گاه برای تمام رأس‌های v به طوری که $(u,v) \in E_f$، داریم $u.h \le v.h$. بنابراین $u.h < 1 + \min\{v.h : (u,v) \in E_f\}$، و عملیات باید $u.h$ را افزایش دهد.

لم ۲۶-۱۶ فرض کنید $G=(V,E)$ یک شبکه‌ی شار با منبع s و چاهک t باشد. حین اجرای GENERIC-PUSH-RELABEL بر روی G، خصیصه‌ی h به صورت یک تابع ارتفاع باقی می‌ماند.

اثبات اثبات به وسیله‌ی استقرا بر روی تعداد اعمال اصلی انجام شده است. در ابتدا همان طور که قبلاً مشاهده کردیم، h یک تابع ارتفاع است.

ادعا می‌کنیم که اگر h یک تابع ارتفاع باشد، آن گاه یک عملیات RELABEL (u)، h را به صورت یک تابع ارتفاع باقی می‌گذارد. اگر یک یال پس‌ماند $(u,v) \in E_f$ را که u را ترک می‌کند در نظر بگیریم، عملیات RELABEL (u) تضمین می‌کند که بعد از عملیات داریم $u.h \le v.h + 1$. اکنون یک یال پس‌ماند (w,u) را در نظر بگیرید که به u وارد می‌شود. طبق لم ۲۶-۱۶، عبارت $w.h \le u.h + 1$ قبل از عملیات RELABEL (u) ایجاب می‌کند که بعد از آن داشته باشیم $w.h < u.h + 1$. بنابراین عملیات RELABEL (u) تابع h را به صورت یک تابع ارتفاع باقی می‌گذارد.

اکنون یک عملیات PUSH (u,v) را در نظر بگیرید. این عملیات ممکن است یک یال (u,v) را به E_f اضافه کند، و ممکن است (u,v) را از آن حذف کند. در حالت اول داریم $v.h = u.h - 1 < u.h + 1$، و بنابراین h یک تابع ارتفاع باقی می‌ماند. در حالت دوم، حذف (u,v) از شبکه‌ی پس‌ماند محدودیت متناظر با آن را هم حذف می‌کند، و دوباره h یک تابع ارتفاع باقی می‌ماند.

لم زیر یک خصوصیت مهم توابع ارتفاع را ارائه می‌کند.

لم ۲۶-۱۳ فرض کنید $G=(V,E)$ یک شبکه‌ی شار با منبع s و چاهک t باشد، و f یک پیش‌شار در G. همچنین فرض کنید h یک تابع ارتفاع روی V باشد. آن گاه هیچ مسیری از منبع s به چاهک t در شبکه‌ی پس‌ماند G_f وجود ندارد.

اثبات طبق برهان خلف فرض کنید یک مسیر $p=\langle v_0, v_1, ..., v_k\rangle$ از s به t در G_f وجود دارد، که در آن $v_0=s$ و $v_k=t$. بدون از دست دادن کلیت، p یک مسیر ساده است، و بنابراین $k<|V|$. برای $i=0,1,...,k-1$، داریم $(v_i,v_{i+1})\in E_f$. چون h یک تابع ارتفاع است، برای $i=0,1,...,k-1$ داریم $h(v_i)\le h(v_{i+1})+1$. ترکیب این نامساوی‌ها روی مسیر p می‌دهد $h(s)\le h(t)+k$. ولی از آن جایی که $h(t)=0$، داریم $k<|V|$، که با لازمه‌ی $h(s)=|V|$ در یک تابع ارتفاع تناقض دارد. ■

اکنون آماده‌ایم که نشان دهیم اگر الگوریتم عام رانش- برچسب‌دهی مجدد پایان یابد، پیش‌شاری که تولید می‌کند یک شار بیشینه است.

قضیه‌ی ۲۶-۱۴ اگر الگوریتم GENERIC-PUSH-RELABEL وقتی که بر روی یک شبکه‌ی شار $G=(V,E)$ با منبع s و چاهک t اجرا می‌شود، پایان یابد، آن گاه شار f محاسبه شده توسط این الگوریتم یک شار بیشینه برای G است.

اثبات از ثابت حلقه‌ی زیر استفاده می‌کنیم:

- هر بار که تست حلقه‌ی while در خط ۲ رویه‌ی GENERIC-PUSH-RELABEL اجرا می‌شود، f یک پیش‌شار است.
- آغاز: رویه‌ی INITIALIZE-PREFLOW شار f را به صورت یک پیش‌شار می‌سازد.
- ادامه: تنها اعمال درون حلقه‌ی while خطوط ۲-۳ اعمال رانش و برچسب‌دهی مجدد هستند. اعمال برچسب‌دهی مجدد فقط بر روی خصیصه‌های ارتفاع تأثیر می‌گذارند و نه روی مقادیر شار؛ بنابراین بر روی پیش‌شار بودن یا نبودن f تأثیری ندارند. همان طور که در قبل‌تر در همین بخش بحث شد، اگر f قبل از یک عملیات رانش یک پیش‌شار باشد، بعد از آن هم یک پیش‌شار خواهد بود.
- پایان: در پایان، هر رأس درون $V-\{s,t\}$ باید یک شار افزونی ۰ داشته باشد، چرا که طبق لم ۲۶-۱۴ و این ثابت که f همیشه یک پیش‌شار است، هیچ رأس در حال سرریزی وجود ندارد. بنابراین f یک شار است. لم ۲۶-۱۶ نشان می‌دهد که h یک تابع ارتفاع است، و بنابراین لم ۲۶-۱۷ به ما می‌گوید که هیچ مسیری از s به t در شبکه‌ی G_f وجود ندارد. بنابراین طبق قضیه‌ی شار بیشینه- برش کمینه (قضیه‌ی ۲۶-۶)، f یک شار بیشینه است. ■

تحلیل متد رانش- برچسب‌دهی مجدد

برای این که نشان دهیم الگوریتم عام رانش-برچسب‌دهی مجدد پایان می‌یابد، برای تعداد اعمالی که انجام می‌دهد کرانی تعیین می‌کنیم. کران هر یک از سه نوع عملیات – برچسب‌دهی مجدد، رانش‌های

اشباع کننده، و رانش‌های غیر اشباع کننده – به طور جداگانه تعیین می‌شود. با دانستن این کران‌ها ساختن الگوریتمی که در زمان $O(V^2E)$ اجرا می‌شود کار سرراستی است. با این حال قبل از شروع تحلیل، یک لم مهم را اثبات می‌کنیم. به خاطر بیاورید که در شبکه‌ی پس‌ماند وجود یال‌های ورودی به منبع مجاز است.

لم ۲۶-۱۶ فرض کنید $G=(V,E)$ یک شبکه‌ی شار با منبع s و چاهک t، و f یک شار در G باشد. در این صورت برای هر رأس در حال سرریز u، یک مسیر ساده از u به s در شبکه‌ی پس‌ماند G_f وجود دارد.

اثبات برای هر رأس در حال در حال سرریز u، فرض کنید

$U = \{v \mid$ یک مسیر ساده از x به v در G_f وجود دارد $\}$

و برای سادگی فرض کنید $s \notin U$. همچنین $\overline{U} = V - U$.

با رجوع به تعریف شار افزونی از تساوی (۲۶-۱۴)، جمع بر روی تمام رأس‌های U، و توجه به این که $V = U \cup \overline{U}$، به دست می‌آوریم

$$\begin{aligned}
&\sum_{u\in U} e(u) \\
&\quad = \sum_{u\in U}\left(\sum_{v\in V} f(v,u) - \sum_{v\in V} f(u,v)\right) \\
&\quad = \sum_{u\in U}\left(\left(\sum_{v\in U} f(v,u) + \sum_{v\in \overline{U}} f(v,u)\right) - \left(\sum_{v\in U} f(u,v) + \sum_{v\in \overline{U}} f(u,v)\right)\right) \\
&\quad = \sum_{u\in U}\sum_{v\in U} f(v,u) + \sum_{u\in U}\sum_{v\in \overline{U}} f(v,u) - \sum_{u\in U}\sum_{v\in U} f(u,v) - \sum_{u\in U}\sum_{v\in \overline{U}} f(u,v) \\
&\quad = \sum_{u\in U}\sum_{v\in \overline{U}} f(v,u) - \sum_{u\in U}\sum_{v\in \overline{U}} f(u,v)
\end{aligned}$$

می‌دانیم که کمیت $\sum_{u\in U} e(u)$ باید مثبت باشد، چرا که $e(x)>0$، $x \in U$، تمام رأس‌ها غیر از s شار افزونی نامنفی دارند، و طبق فرض، $s \notin U$. بنابراین داریم

$$\sum_{u\in U}\sum_{v\in \overline{U}} f(v,u) - \sum_{u\in U}\sum_{v\in \overline{U}} f(u,v) > 0 \qquad (۲۶-۱۷)$$

شار تمام یال‌ها نامنفی است، و بنابراین برای این که تساوی (۲۶-۱۷) برقرار باشد، باید داشته باشیم $\sum_{u\in U}\sum_{v\in \overline{U}} f(v,u) > 0$. پس باید حداقل یک جفت رأس $u' \in U$ و $v' \in \overline{U}$ وجود داشته باشد که $f(v',u') > 0$. ولی اگر $f(v',u') > 0$ باید یک یال پس‌ماند (u',v') وجود داشته باشد، که بدین معنی است که یک مسیر ساده از x به v' وجود دارد (مسیر $x \rightsquigarrow u' \to v'$) که با تعریف U در تناقض است.

لم بعد ارتفاع رأس‌ها را محدود می‌کند، و نتیجه‌ی آن کرانی برای تعداد اعمال برچسب‌دهی مجدد انجام شده تعیین می‌کند.

لم ۲۶-۲۰ فرض کنید $G=(V,E)$ یک شبکه‌ی شار با منبع s و چاهک t باشد. در هر لحظه در طول اجرای GENERIC-PUSH-RELABEL روی G، برای تمام رأس‌های $u \in V$ داریم $u.h \le 2|V|-1$.

اثبات ارتفاع منبع s و چاهک t هیچ گاه تغییر نمی‌کند چرا که این رأس‌ها طبق تعریف در حال سرریز نیستند. بنابراین همیشه داریم $s.h = |V|$ و $t.h = 0$، که هیچ کدام بزرگ‌تر از $2|V|-1$ نیستند.

اکنون یک رأس $u \in V - \{s,t\}$ را در نظر بگیرید. در ابتدا $u.h = 0 \le 2|V|-1$. نشان خواهیم داد که بعد از هر عملیات برچسب‌دهی مجدد همچنان داریم $u.h \le 2|V|-1$. وقتی u برچسب‌دهی مجدد می‌شود، در حال سرریز است، و لم ۲۶-۲۰ به ما می‌گوید که یک مسیر ساده‌ی p از u به s در G_f وجود دارد. فرض کنید $p = \langle v_0, v_1, \ldots, v_k \rangle$، که در آن $v_0 = u$، $v_k = s$، و $k \le |V|-1$، زیرا p ساده است. برای $i = 0, 1, \ldots, k-1$ داریم $(v_i, v_{i+1}) \in E_f$، و بنابراین طبق لم ۲۶-۱۷، $h + 1 \ge v_i.h$. گسترش این نامساوی‌ها روی مسیر p به ما می‌دهد $u.h = v_0.h \le v_k.h + k \le s.h + (|V|-1) = 2|V|-1$. ■

نتیجه‌ی ۲۶-۲۱ *(کران اعمال برچسب‌دهی مجدد)* فرض کنید $G=(V,E)$ یک شبکه‌ی شار با منبع s و چاهک t باشد. در این صورت حین اجرای GENERIC-PUSH-RELABEL روی G، تعداد اعمال برچسب‌دهی مجدد برای هر رأس حداکثر $2|V|-1$، و در کل حداکثر $(2|V|-1)(|V|-2) < 2|V|^2$ خواهد بود.

اثبات فقط $|V|-2$ رأس در $V - \{s,t\}$ ممکن است برچسب‌دهی مجدد شوند. فرض کنید $u \in V - \{s,t\}$. عملیات RELABEL(u) مقدار $u.h$ را افزایش می‌دهد. مقدار $u.h$ در ابتدا ۰ است، و طبق لم ۲۶-۲۱ حداکثر $2|V|-1$ بار رشد می‌کند. بنابراین هر رأس $u \in V - \{s,t\}$ حداکثر $2|V|-1$ بار برچسب‌دهی مجدد می‌شود، و تعداد کل اعمال برچسب‌دهی مجدد انجام شده حداکثر برابر $(2|V|-1)(|V|-2) < 2|V|^2$ است. ■

لم ۲۶-۲۰ همچنین به ما کمک می‌کند که تعداد رانش‌های اشباع کننده را محدود کنیم.

لم ۲۶-۲۲ *(کران رانش‌های اشباع کننده)* در طول اجرای GENERIC-PUSH-RELABEL روی هر شبکه‌ی شار $G=(V,E)$، تعداد رانش‌های اشباع کننده کم‌تر از $2|V||E|$ است.

اثبات برای هر جفت رأس $u, v \in V$، تعداد رانش‌های اشباع کننده از u به v و از v به u را با هم خواهیم شمارد، که به آن‌ها رانش‌های اشباع کننده میان u و v می‌گوییم. اگر چنین رانشی وجود داشته

باشد، حداقل یکی از (u,v) و (v,u) یالی در E است. اکنون فرض کنید یک رانش اشباع کننده از u به v اتفاق افتاده است. در آن زمان داریم $v.h = u.h - ۱$. برای این که بعداً یک رانش دیگر از u به v رخ دهد، ابتدا الگوریتم باید شار را از v به u براند، که این هم نمی‌تواند اتفاق بیافتد مگر این که $v.h = u.h + ۱$. از آن جایی که $u.h$ هیچ گاه کاهش نمی‌یابد، برای این که $v.h = u.h + ۱$ برقرار باشد، مقدار $v.h$ باید حداقل به اندازه‌ی ۲ واحد افزایش یابد. به همین صورت، $u.h$ هم باید بین رانش‌های اشباع کننده از v به u حداقل ۲ واحد افزایش یابد. ارتفاع‌ها از ۰ شروع می‌شوند، و طبق لم ۲۶-۲۰ هیچ گاه از $۲|V| - ۱$ فراتر نمی‌روند، که ایجاب می‌کند تعداد دفعاتی که ارتفاع هر رأسی می‌تواند ۲ واحد افزایش یابد حداکثر $|V|$ است. چون بین هر دو رانش اشباع کننده میان u و v حداقل یکی از $u.h$ و $v.h$ باید ۲ واحد افزایش یابد، پس کم‌تر از $۲|V|$ رانش اشباع کننده میان u و v خواهیم داشت. ضرب در تعداد یال‌ها کران بالای $۲|V||E|$ را برای تعداد رانش‌های اشباع کننده به دست خواهد داد. ■

لم زیر کران تعداد رانش‌های غیر اشباع کننده را در یک الگوریتم عام رانش- برچسب‌دهی مجدد می‌دهد.

لم ۲۶-۲۳
کران رانش‌های غیر اشباع کننده

در طول اجرای GENERIC-PUSH-RELABEL بر روی یک شبکه‌ی شار $G = (V, E)$، تعداد رانش‌های غیر اشباع کننده کم‌تر از $۴|V|^۲(|V| + |E|)$ است.

اثبات یک تابع پتانسیل $\Phi = \sum_{v: e(v)>۰} v.h$ تعریف می‌کنیم. در ابتدا، $\Phi = ۰$، و مقدار Φ ممکن است بعد از هر برچسب‌دهی مجدد، رانش اشباع کننده، و رانش غیر اشباع کننده تغییر کند. برای میزان افزایشی که رانش‌های اشباع کننده و برچسب‌دهی‌های مجدد می‌توانند برای Φ ایجاد کنند یک کران تعیین می‌کنیم. سپس نشان خواهیم داد که هر رانش غیر اشباع کننده باید Φ را حداقل ۱ واحد افزایش دهد، و از این کران‌ها استفاده می‌کنیم تا به یک کران بالا برای تعداد رانش‌های غیر اشباع کننده برسیم.

اجازه دهید دو راه ممکن برای افزایش Φ را بررسی کنیم. اول، برچسب‌دهی مجدد یک رأس u، Φ را به اندازه‌ی کم‌تر از $۲|V|$ واحد افزایش می‌دهد، چرا که مجموعه‌ای که سری بر روی آن محاسبه می‌شود یکی است و برچسب‌دهی مجدد نمی‌تواند ارتفاع u را به اندازه‌ی بیش از حداکثر ارتفاع بیشینه افزایش دهد، که طبق لم ۲۶-۲۰، حداکثر برابر است با $۲|V| - ۱$. دوم، یک رانش اشباع کننده از یک رأس u به یک رأس v تابع Φ را به اندازه‌ی کم‌تر از $۲|V|$ افزایش می‌دهد، چرا که هیچ ارتفاعی تغییر نمی‌کند و فقط رأس v، که ارتفاع آن حداکثر $۲|V| - ۱$ است ممکن است سرریز شود.

اکنون نشان می‌دهیم که یک رانش غیر اشباع کننده از u به v تابع Φ را حداقل به اندازه‌ی ۱ واحد کاهش می‌دهد. چرا؟ قبل از رانش غیر اشباع کننده u در حال سرریز بوده است، و v ممکن است درحال سرریز بوده یا نبوده باشد. طبق لم ۲۶-۱۳، بعد از رانش، u دیگر در حال سرریز نخواهد بود. به علاوه اگر v منبع نباشد، بعد از رانش ممکن است در حال سرریز باشد یا نباشد. بنابراین تابع پتانسیل Φ دقیقاً به اندازه‌ی $u.h$ کاهش یافته است، و به اندازه‌ی ۰ یا $v.h$ افزایش یافته است. از آن

جایی که $u.h - v.h = 1$، تأثیر خالص این است که تابع پتانسیل حداقل ۱ واحد کاهش یافته است.

بنابراین در حین اجرای الگوریتم کل افزایش در Φ در نتیجه‌ی برچسب‌دهی‌های مجدد و رانش‌های اشباع کننده است، و طبق نتیجه‌ی ۲۶-۲۱ و لم ۲۶-۲۲، این افزایش کم‌تر از $(2|V|)(2|V|^2) + (2|V|)(2|V||E|) = 4|V|^2(|V|+|E|)$ خواهد بود. از آن جایی که $\Phi \geq 0$، میزان کل کاهش، و بنابراین تعداد کل رانش‌های غیر اشباع کننده کم‌تر از $4|V|^2(|V|+|E|)$ است. ■

با تعیین کران برچسب‌دهی‌های مجدد، رانش‌های اشباع کننده و رانش‌های غیر اشباع کننده، زمینه را برای تحلیل زیر بر روی رویه‌ی GENERIC-PUSH-RELABEL، و بنابراین برای هر الگوریتمی بر پایه‌ی متد رانش- برچسب‌دهی مجدد آماده کرده‌ایم.

لم‌ه‌ی ۲۶-۲۳ در طول اجرای GENERIC-PUSH-RELABEL بر روی یک شبکه‌ی شار $G = (V, E)$، تعداد اعمال اصلی $O(V^2E)$ است.

اثبات مستقیماً از نتیجه‌ی ۲۶-۲۱ و لم‌های ۲۶-۲۲ و ۲۶-۲۳. ■

بنابراین الگوریتم بعد از $O(V^2E)$ عملیات پایان می‌یابد. تنها چیزی که باقی می‌ماند، ارائه‌ی یک متد کارامد برای پیاده‌سازی هر یک از اعمال و برای انتخاب یک عملیات مناسب برای اجرا است.

نتیجه‌ی ۲۶-۲۵ یک پیاده‌سازی از الگوریتم عام رانش-برچسب‌دهی مجدد وجود دارد که برای هر شبکه‌ی شار $G = (V, E)$ در زمان $O(V^2E)$ اجرا می‌شود.

اثبات تمرین ۲۶-۴-۲ از شما می‌خواهد نشان دهید که چطور می‌توان الگوریتم عام را با $O(V)$ سربار برای هر عملیات برچسب‌دهی مجدد و $O(1)$ برای هر رانش پیاده‌سازی کرد. این تمرین همچنین از شما می‌خواهد که یک ساختمان داده طراحی کنید که به شما اجازه می‌دهد در زمان $O(1)$ یک عملیات مناسب برای اجرا انتخاب کنید. در این صورت حکم اثبات می‌شود. ■

تمرین‌ها

۲۶-۴-۱ اثبات کنید که بعد از پایان رویه‌ی INITIALIZE-PREFLOW (G, s)، داریم $|f^*| - \leq s.e$، که در آن f^* شار بیشینه روی G است.

۲۶-۴-۲ نشان دهید چگونه می‌توان الگوریتم عام رانش- برچسب‌دهی مجدد را با استفاده از زمان $O(V)$ برای هر عملیات برچسب‌دهی مجدد، زمان $O(1)$ برای هر عملیات رانش، و زمان $O(1)$ برای انتخاب عملیات مناسب برای اجرا، با کل زمان اجرای $O(V^2E)$ پیاده‌سازی کرد.

۲۶-۴-۳ اثبات کنید که الگوریتم عام رانش- برچسب‌دهی مجدد در کل فقط $O(VE)$ زمان برای اجرای تمام $O(V^2)$ عملیات برچسب‌دهی مجدد صرف می‌کند.

۲۶-۴-۴ فرض کنید یک شار بیشینه در یک شبکه‌ی شار $G = (V, E)$ با استفاده از الگوریتم رانش- برچسب‌دهی مجدد یافت شده است. یک الگوریتم سریع برای یافتن یک برش کمینه در G ارائه کنید.

۲۶-۴-۵ یک الگوریتم کارامد رانش- برچسب‌دهی مجدد برای یافتن یک تطابق بیشینه در یک گراف دوبخشی ارائه کنید. الگوریتم خود را تحلیل کنید.

۲۶-۴-۶ فرض کنید ظرفیت تمام یال‌ها در یک شبکه‌ی شار $G = (V, E)$ عضو مجموعه‌ی $\{1, 2, ..., k\}$ هستند. زمان اجرای الگوریتم عام رانش- برچسب‌دهی مجدد را بر حسب $|V|$، $|E|$، و k تحلیل کنید. (راهنمایی: هر یال قبل از اشباع شدن چند بار از رانش‌های غیر اشباع کننده پشتیبانی می‌کند؟)

۲۶-۴-۷ نشان دهید که خط ۷ رویه‌ی INITIALIZE-PREFLOW را می‌توان به صورت

6 $s.h = |G.V| - 2$

تغییر داد، بدون این که درستی و یا کارایی حدی الگوریتم عام رانش- برچسب‌دهی مجدد تغییر کند.

۲۶-۴-۸ فرض کنید $\delta_f(u, v)$ فاصله (تعداد یال‌ها)ی از u به v در شبکه‌ی پس‌ماند G_f باشد. نشان دهید که GENERIC-PUSH-RELABEL این خصوصیات را نگه می‌دارد که $u.h < |V|$ ایجاب می‌کند $u.h < \delta_f(u, t)$، و $u.h \geq |V|$ ایجاب می‌کند $u.h - |V| \leq \delta_f(u, s)$.

★ ۲۶-۴-۹ مانند تمرین قبلی، فرض کنید $\delta_f(u, v)$ فاصله‌ی از u به v در شبکه‌ی پس‌ماند G_f باشد. نشان دهید که الگوریتم عام رانش- برچسب‌دهی مجدد را می‌توان طوری اصلاح کرد که این خصوصیات را نگه دارد، که $u.h < |V|$ نتیجه می‌دهد $u.h = \delta_f(u, t)$، و $u.h \geq |V|$ نتیجه می‌دهد $u.h - |V| = \delta_f(u, s)$. کل زمانی که الگوریتم شما برای حفظ این خصوصیات صرف می‌کند باید $O(VE)$ باشد.

۲۶-۴-۱۰ نشان دهید که تعداد رانش‌های غیر اشباع‌کننده‌ی اجرا شده توسط رویه‌ی GENERIC-PUSH-RELABEL روی یک شبکه‌ی شار $G = (V, E)$ برای $|V| \geq 4$ حداکثر برابر است با $4|V|^2|E|$.

۲۶-۵ الگوریتم برچسب‌دهی مجدد - به جلو

متد رانش- برچسب‌دهی مجدد به ما اجازه می‌دهد که اعمال اصلی را به هر ترتیبی اجرا کنیم. با این حال، با انتخاب مناسب ترتیب اجرا و اداره‌ی ساختمان‌های داده‌ی شبکه به صورت کارامد می‌توانیم مسئله‌ی شار بیشینه را سریع‌تر از کران $O(V^2E)$ داده شده توسط نتیجه‌ی ۲۶-۲۵ حل کنیم. در این جا

الگوریتم برچسب‌دهی مجدد- به جلو را بررسی می‌کنیم، یک الگوریتم رانش- برچسب‌دهی مجدد که زمان اجرای آن $O(V^3)$ است، که به صورت حدی حداقل به خوبی $O(V^2E)$ است، و برای شبکه‌های شلوغ بهتر از آن.

الگوریتم برچسب‌دهی مجدد- به جلو یک لیست از رأس‌ها در شبکه نگه می‌دارد. با شروع از جلو، الگوریتم لیست را پویش کرده و مکرراً یک رأس در حال سرریز u را انتخاب و سپس آن را «تخلیه» می‌کند، یعنی انجام اعمال رانش و برچسب‌دهی مجدد تا زمانی که u دیگر شار افزونی مثبت نداشته باشد. هر گاه یک رأس برچسب‌دهی مجدد می‌شود، به جلوی لیست منتقل می‌شود (نام «برچسب‌دهی مجدد- به جلو» از این جا آمده است) و الگوریتم پویش را از سر می‌گیرد.

یال‌های قابل قبول و شبکه‌ها

اگر $G=(V,E)$ یک شبکه‌ی شار با منبع s و چاهک t، f یک پیش‌شار در G، و h یک تابع ارتفاع باشد، آن گاه می‌گوییم (u,v) یک یال *قابل قبول* (admissible edge) است اگر $c_f(u,v)>0$ و $h(u)=h(v)+1$. در غیر این صورت، (u,v) *غیر قابل قبول* است. $G_{f,h}=(V,E_{f,h})$ یک *شبکه‌ی قابل قبول* است، که در آن $E_{f,h}$ مجموعه‌ی یال‌های قابل قبول می‌باشد.

شبکه‌ی قابل قبول شامل یال‌هایی است که می‌توان شار را در آن‌ها راند.

لم ۲۶.۶

اگر $G=(V,E)$ یک شبکه‌ی شار، f یک پیش‌شار در G، و h یک تابع ارتفاع در G باشد، آن گاه شبکه‌ی قابل قبول $G_{f,h}=(V,E_{f,h})$ بدون دور است.

اثبات اثبات از طریق برهان خلف است. فرض کنید $G_{f,h}$ حاوی یک دور $p=\langle v_0,v_1,\ldots,v_k\rangle$ باشد، که در آن $v_0=v_k$ و $k>0$. از آن جایی که هر یال در p قابل قبول است، برای $i=1,2,\ldots,k$ داریم $h(v_{i-1})=h(v_i)$. جمع بر روی یال‌های دور به دست می‌دهد

$$\begin{aligned}\sum_{k=1}^{k} h(v_{i-1}) &= \sum_{k=1}^{k}(h(v_i)+1)\\ &= \sum_{i=1}^{k} h(v_i)+k\end{aligned}$$

چون هر رأس در دور p یک بار در هر یک از مجموع‌ها ظاهر می‌شود، به این تناقض می‌رسیم که $k=0$. ■

دو لم بعد نشان می‌دهند که اعمال راندن و برچسب‌دهی مجدد، شبکه‌ی قابل قبول را تغییر می‌دهد.

لم ۲۶.۷

فرض کنید $G=(V,E)$ یک شبکه‌ی شار و f یک پیش‌شار در G باشد، و فرض کنید خصیصه‌ی h یک تابع ارتفاع باشد. اگر u یک رأس در حال سرریز و (u,v) یک یال قابل قبول باشد، آن گاه PUHS(u,v) کاربرد خواهد داشت. این عملیات هیچ یال قابل قبول جدیدی نمی‌سازد، ولی ممکن است باعث شود که (u,v) غیر قابل قبول شود.

اثبات طبق تعریف یال‌های قابل قبول، شار را می‌توان از u به v راند. از آن جایی که u در حال سرریز است، عملیات PUSH(u,v) کاربرد خواهد داشت. تنها یال پس‌ماند جدیدی که ممکن است با راندن شار از u به v ساخته شود یال (u,v) است. چون $v.h = u.h - 1$، یال (u,v) نمی‌تواند قابل قبول شود. اگر عملیات یک رانش اشباع کننده باشد، آن گاه بعد از آن خواهیم داشت $c_f(u,v) = 0$ و (u,v) قابل قبول خواهد شد. □

لم ۲۶-۲۱ فرض کنید $G = (V,E)$ یک شبکه‌ی شار باشد، f یک پیش‌شار در G، و خصیصه‌ی h یک تابع ارتفاع. اگر u یک رأس در حال سرریز باشد و هیچ یال قابل قبول خروجی از u وجود نداشته باشد، آن گاه RELABEL(u) کاربرد خواهد داشت. بعد از عملیات برچسب‌دهی مجدد، حداقل یک یال قابل قبول خروجی از u وجود خواهد داشت، ولی هیچ یال قابل قبول ورودی به u وجود نخواهد داشت.

اثبات اگر u در حال سرریز باشد، آن گاه طبق لم ۲۶-۱۴ یا یک عملیات رانش و یا یک عملیات برچسب‌دهی مجدد بر روی آن کاربرد خواهد داشت. اگر هیچ یال قابل قبول خروجی از u وجود نداشته باشد، آن گاه نمی‌توان شار را از u راند، و بنابراین RELABEL(u) کاربرد خواهد داشت. بعد از عملیات برچسب‌دهی مجدد داریم $u.h = 1 + \min\{v.h : (u,v) \in E_f\}$. بنابراین اگر v رأسی باشد که کمینه را در این مجموعه تولید می‌کند، یال (u,v) قابل قبول خواهد شد. از این رو بعد از برچسب‌دهی مجدد حداقل یک یال قابل قبول خروجی از u وجود خواهد داشت.

برای این که نشان دهیم بعد از یک برچسب‌دهی مجدد هیچ یال قابل قبول ورودی به u وجود ندارد، فرض کنید یک رأس (u,v) وجود دارد به طوری که (u,v) قابل قبول است. در این صورت بعد از برچسب‌دهی مجدد خواهیم داشت $v.h = u.h + 1$، و بنابراین دقیقاً قبل از برچسب‌دهی مجدد داریم $v.h > u.h + 1$. طبق لم ۲۶-۱۳ هیچ یال پس‌ماندی بین رأس‌هایی که اختلاف ارتفاع آن‌ها بیش از ۱ است، وجود ندارد. به علاوه برچسب‌دهی مجدد یک رأس شبکه‌ی پس‌ماند را تغییر نمی‌دهد. بنابراین (u,v) در شبکه‌ی پس‌ماند نیست، و نمی‌تواند در شبکه‌ی قابل قبول هم باشد. □

لیست همسایه‌ها

یال‌ها در الگوریتم برچسب‌دهی مجدد-به جلو در «لیست‌های همسایه‌ها» دسته بندی می‌شوند. در یک شبکه‌ی شار $G = (V,E)$، *لیست همسایه‌های* (neighbor list) $u.N$ برای یک رأس $u \in V$، یک لیست پیوندی یک طرفه از همسایه‌های u در G است. بنابراین رأس v در لیست $u.N$ خواهد بود اگر $(u,v) \in E$ یا $(v,u) \in E$. لیست همسایه‌های $u.N$ دقیقاً حاوی رأس‌های v است که ممکن است برای آن‌ها یک یال پس‌ماند (u,v) وجود داشته باشد. رأس اول در $u.N$ توسط اشاره‌گر $u.N.head$ مشخص می‌شود. $v.next\text{-}neightbor$ به رأس بعد از v در لیست همسایه‌ها اشاره می‌کند؛ اگر v آخرین رأس لیست باشد این اشاره‌گر NIL است.

الگوریتم برچسب‌دهی مجدد–به جلو روی هر لیست همسایه به ترتیبی دلخواه، که البته در طول الگوریتم ثابت است، دور می‌زند. برای هر رأس u، فیلد $u.current$ به رأسی که اکنون در حال بررسی در $u.N$ است اشاره می‌کند. در ابتدا، $u.current$ با $u.N.head$ مقداردهی می‌شود.

تخلیه‌ی یک رأس در حال سرریز

یک رأس در حال سرریز u با راندن تمام شار افزونی آن به رأس‌های همسایه از طریق یال‌های قابل قبول، و در صورت لزوم برچسب‌دهی مجدد u برای تبدیل یال‌های خروجی u به یال‌های قابل قبول، تخلیه (discharg) می‌شود. شبه‌کد آن به صورت زیر است.

```
DISCHARGE(u)
1  while u.e > 0
2      v = u.current
3      if v == NIL
4          RELABEL(u)
5          u.current = u.N.head
6      elseif c_f(u, v) > 0 and u.h = v.h + 1
7          PUSH(u, v)
8      else u.current = v.next-neighbor
```

شکل ۲۶–۹ در طول تکرارهای مختلف حلقه‌ی while خطوط ۱–۸ گذر می‌کند. این حلقه تا زمانی اجرا می‌شود که رأس u افزونی مثبت داشته باشد. هر تکرار دقیقاً یکی از سه زیر عملیات را انجام می‌دهد، بسته به رأس فعلی v در لیست همسایه‌های $u.N$.

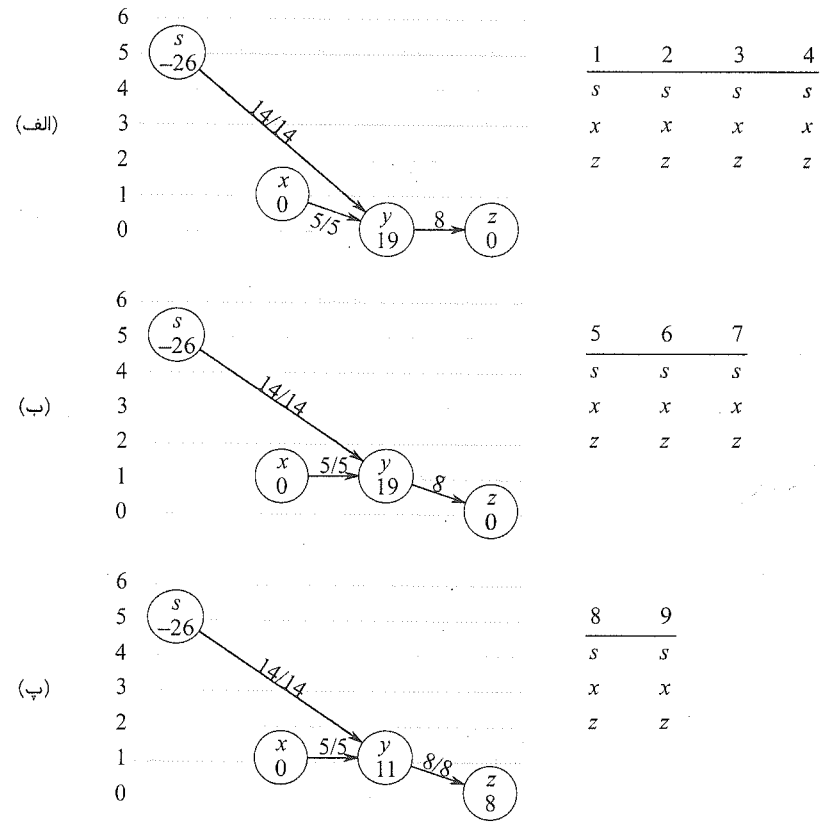
۱. اگر v مقدار NIL داشته باشد، آن گاه به پایان $u.N$ رسیده‌ایم. خط ۴ رأس u را برچسب‌دهی مجدد می‌کند، و سپس خط ۵ همسایه‌ی فعلی u را طوری مقداردهی می‌کند که اولین رأس در $u.N$ باشد. (لم ۲۶–۲۹ در زیر نشان می‌دهد که در این حالت عملیات برچسب‌دهی مجدد کاربرد دارد.)

۲. اگر v غیر NIL باشد و (u,v) یک یال قابل قبول (تعیین شده توسط تست خط ۶)، آن گاه خط ۷ مقداری از (یا همه‌ی) شار افزونی u را به رأس v می‌راند.

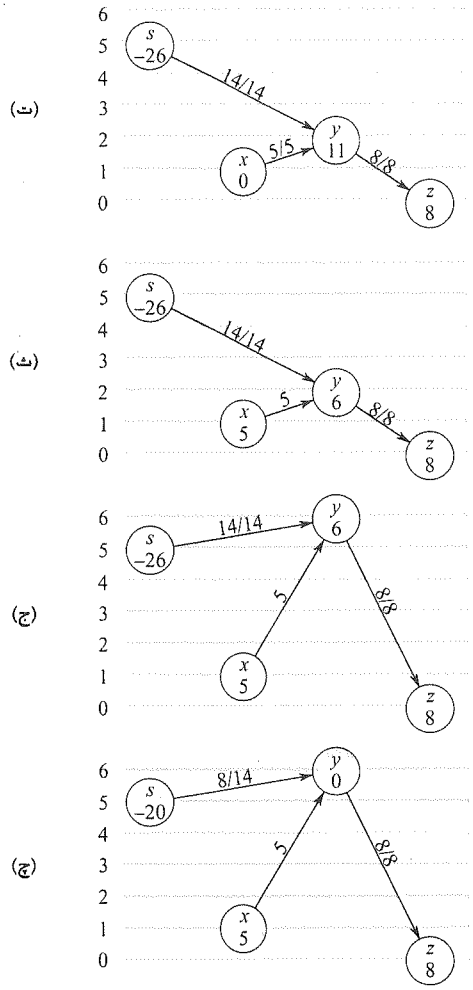
۳. اگر v غیر NIL باشد ولی (u,v) غیر قابل قبول، آن گاه خط ۸ $u.current$ را در لیست همسایه‌های $u.N$ یک خانه به جلوتر می‌برد.

مشاهده کنید که اگر DISCHARGE بر روی یک رأس در حال سرریز u فراخوانی شود، آن گاه حداقل کار انجام شده توسط DISCHARGE باید یک رانش از u باشد. چرا؟ رویه فقط زمانی پایان می‌یابد که $u.e$ صفر شود، و نه برچسب‌دهی مجدد و نه به جلو بردن اشاره‌گر $u.current$ بر روی مقدار $u.e$ تأثیری نمی‌گذارند.

باید اطمینان حاصل کنیم که وقتی PUSH یا RELABEL توسط DISCHARGE فراخوانی شوند که کاربرد داشته باشند. لم بعد این مسئله را اثبات می‌کند.



شکل ۲۶-۹ تخلیه‌ی یک رأس y. برای راندن تمام شار افزونی از y، ۱۵ تکرار از حلقه‌ی while رویه‌ی DISCHARGE اجرا می‌شود. فقط همسایه‌های y و یال‌های خروجی و ورودی آن نشان داده شده‌اند. در هر قسمت، شماره‌ی درون هر رأس شار افزونی آن در آغاز اولین تکرار نشان داده شده در آن بخش است، و هر رأس در آن بخش بر روی ارتفاع آن نشان داده شده است. در سمت راست لیست همسایه‌های y. N در ابتدای هر تکرار، با شماره‌ی تکرار در بالا نشان داده شده است. همسایه‌ی سایه‌دار $y.current$ است. (الف) در ابتدا ۱۹ واحد شار افزونی برای راندن در y وجود دارد، و $y.current = s$. تکرارهای ۱، ۲، و ۳ فقط $y.current$ را به جلو می‌برند، چرا که هیچ یال قابل قبول خروجی از y وجود ندارد. در تکرار ۴ داریم $y.current = \text{NIL}$ (که با قرار گرفتن سایه زیر لیست همسایه نشان داده شده است)، و بنابراین y برچسب‌دهی مجدد شده و $y.current$ به ابتدای لیست همسایه‌ها بازمی‌گردد. (ب) بعد از برچسب‌دهی مجدد، ارتفاع رأس y برابر ۱ است. در تکرار ۵ و ۶ مشخص می‌شود که یال‌های (y, s) و (y, x) غیر قابل قبول هستند، ولی ۸ واحد شار افزونی از y به z در تکرار ۷ رانده شده است. به علت رانش، $y.current$ در این تکرار به جلو برده نمی‌شود. (پ) چون رانش در تکرار ۷ یال (y, z) را اشباع کرده است، این یال در تکرار ۸ غیر قابل قبول است. در تکرار ۹ داریم $y.current = \text{NIL}$، و بنابراین رأس y دوباره برچسب‌دهی مجدد و $y.current$ بازنشانی می‌شود.



شکل ۲۶-۹ (ادامه) (ت) در تکرار ۱۰ یال (y, s) غیر قابل قبول است، ولی در تکرار ۱۱، ۵ واحد شار افزونی از y به x رانده می‌شود. (ث) چون $y.current$ در تکرار ۱۱ به جلو برده نشده است، در تکرار ۱۲ یال (y, x) غیر قابل قبول است. در تکرار ۱۳ یال (y, z) به صورت غیر قابل قبول تشخیص داده می‌شود، و تکرار ۱۴ رأس y را برچسب‌دهی مجدد و $y.current$ را بازنشانی می‌کند. (ج) تکرار ۱۵، ۶ واحد شار افزونی را از y به s می‌راند. (چ) اکنون رأس y هیچ شار افزونی ندارد، و DISCHARGE پایان می‌یابد. در این مثال، هم آغاز و هم پایان DISCHARGE در حالی است که اشاره‌گر فعلی به ابتدای لیست همسایه‌ها اشاره می‌کند، ولی در حالت کلی ممکن است این اتفاق رخ ندهد.

لم ۲۶-۲۹ اگر DISCHARGE در خط ۷ رویه‌ی $\text{PUSH}(u, v)$ را فراخوانی کند، آن گاه یک عملیات رانش برای (u, v) کاربرد دارد. اگر DISCHARGE در خط ۴ عملیات $\text{RELABEL}(u)$ را فراخوانی کند، آن گاه یک برچسب‌دهی مجدد برای u کاربرد دارد.

اثبات تست‌های خطوط ۱ و ۶ اطمینان می‌دهند که یک عملیات رانش فقط زمانی رخ می‌دهد که این عملیات کاربرد داشته باشد، که عبارت اول لم را اثبات می‌کند.

برای اثبات عبارت دوم، طبق تست خط ۱ و لم ۲۶-۲۹ فقط باید نشان دهیم که تمام یال‌های خروجی u غیر قابل قبول هستند. اگر یک فراخوانی DISCHARGE(u) با یک اشاره‌گر $u.current$ در ابتدای لیست همسایه‌های u آغاز شود، و با همان اشاره‌گر در انتهای لیست پایان یابد، آن گاه تمام یال‌های خروجی u غیر قابل قبول هستند، و باید یک عملیات برچسب‌دهی مجدد انجام شود. ولی احتمال دارد که حین فراخوانی DISCHARGE(u)، اشاره‌گر $u.current$ قبل از بازگشت رویه فقط بخشی از لیست را بپیماید. بعد از آن ممکن است فراخوانی‌های DISCHARGE روی رأس‌های دیگر انجام شود، ولی $u.current$ در فراخوانی بعد DISCHARGE(u) به حرکت درون لیست ادامه می‌دهد. اکنون بررسی می‌کنیم که هنگام یک عبور کامل از روی لیست چه رخ می‌دهد، که در ابتدای $u.N$ آغاز و با $u.current = \text{NIL}$ پایان می‌یابد. وقتی $u.current$ به انتهای لیست می‌رسد، رویه u را برچسب‌دهی مجدد کرده و یک پیمایش جدید را آغاز می‌کند. برای این که در طول یک گذر اشاره‌گر $u.current$ از روی یک رأس $v \in u.N$ عبور کند، یال (u,v) باید توسط تست خط ۶ غیر قابل قبول تشخیص داده شود. بنابراین زمانی که گذر کامل می‌شود، تمام یال‌های خروجی u در آن گذر به صورت غیر قابل قبول تشخیص داده شده‌اند. مشاهده‌ی کلیدی این است که در پایان هر گذر، تمام یال‌های خروجی u همچنان غیر قابل قبول هستند. چرا؟ طبق لم ۲۶-۲۷ رانش‌ها نمی‌توانند هیچ یالی را قابل قبول کنند، از جمله یال‌های خروجی u. بنابراین تمامی یال‌های قابل قبول باید در عملیات برچسب‌دهی مجدد ساخته شده باشند. ولی رأس u در طول گذر برچسب‌دهی مجدد نشده است، و طبق لم ۲۶-۲۸، هر رأس دیگر v که در آن گذر برچسب‌دهی مجدد شده باشد، بعد از برچسب‌دهی مجدد هیچ یال ورودی قابل قبولی ندارد. بنابراین در پایان گذر تمام یال‌های خروجی u غیر قابل قبول باقی می‌مانند، و حکم اثبات می‌شود. ■

الگوریتم برچسب‌دهی مجدد-به جلو

در الگوریتم برچسب‌دهی مجدد-به جلو، یک لیست پیوندی L حاوی تمام رأس‌های $V - \{s,t\}$ نگه می‌داریم. یک خصوصیت کلیدی این است که رأس‌های درون L بر حسب شبکه‌ی قابل قبول به صورت توپولوژیکی مرتب شده است، همان طور که در ثابت حلقه‌ی زیر خواهیم دید. (از لم ۲۶-۲۶ به خاطر بیاورید که شبکه‌ی قابل قبول یک گراف جهت‌دار بدون دور است.)

شبه‌کد الگوریتم برچسب‌دهی مجدد-به جلو فرض می‌کند که لیست‌های همسایه‌های $u.N$ قبلاً برای هر رأس u ساخته شده‌اند. همچنین فرض می‌کند که $u.next$ به رأسی که در لیست L بعد از u قرار دارد اشاره می‌کند و مانند قبل، اگر u آخرین رأس در لیست باشد داریم $u.next = \text{NIL}$.

```
RELABEL-TO-FRONT(G, s, t)
1  INITIALIZE-PREFLOW(G, s)
2  L = G.V − {s, t}, in any order
```

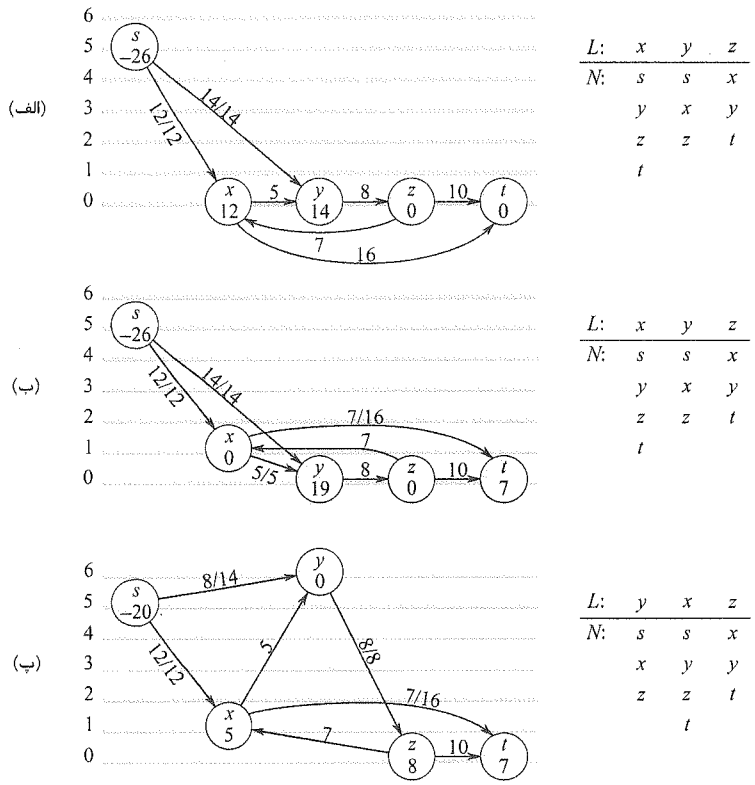
```
3  for each vertex u ∈ G.V − {s, t}
4      u.current = u.N.head
5  u = L.head
6  while u ≠ NIL
7      old-height = u.h
8      DISCHARGE(u)
9      if u.h > old-height
10         move u to the front of list L
11     u = u.next
```

الگوریتم برچسب‌دهی مجدد–به جلو به صورت زیر کار می‌کند. خط ۱ پیش‌شار و ارتفاع‌ها را مانند الگوریتم عام رانش– برچسب‌دهی مجدد مقداردهی اولیه می‌کند. خط ۲ لیست L را طوری مقداردهی می‌کند که حاوی تمام رأس‌هایی باشد که احتمالاً در حال سرریز هستند، به ترتیب. خطوط ۳–۴ اشاره‌گر *current* هر رأس u را با اولین رأس در لیست همسایه‌های u مقداردهی می‌کند.

همان طور که در شکل ۲۶–۱۰ نشان داده شده است، حلقه‌ی while خطوط ۶–۱۱ در طول لیست حرکت، و رأس‌ها را تخلیه می‌کند. خط ۵ باعث می‌شود که این حلقه با اولین رأس در لیست آغاز کند. در هر بار عبور از حلقه، یک رأس u در خط ۸ تخلیه می‌شود. اگر u توسط رویه‌ی DISCHARGE برچسب‌دهی مجدد شده باشد، خط ۱۰ آن را به جلوی لیست L منتقل می‌کند. تشخیص این مسئله توسط ذخیره‌ی ارتفاع u در متغیر *old-height* قبل از عملیات تخلیه (خط ۷) و مقایسه‌ی این ارتفاع ذخیره شده با ارتفاع u بعد از عملیات (خط ۹) انجام می‌شود. خط ۱۱ باعث می‌شود که تکرار بعدی حلقه‌ی while بر روی رأس بعد از u در لیست L انجام شود. اگر u به ابتدای لیست منتقل شده باشد، رأس استفاده شده در تکرار بعدی رأسی است که بعد از مکان جدید u در لیست قرار دارد.

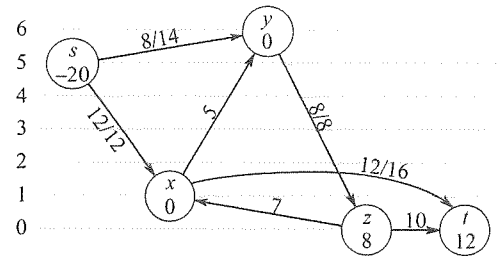
برای این که نشان دهیم RELABEL-TO-FRONT شار بیشینه را محاسبه می‌کند، نشان خواهیم داد که این الگوریتم یک پیاده‌سازی از الگوریتم عام رانش–برچسب‌دهی مجدد است. ابتدا مشاهده کنید که در این رویه اعمال رانش و برچسب‌دهی مجدد فقط زمانی انجام می‌شوند که کاربرد داشته باشند، همان طور که لم ۲۶–۲۹ تضمین می‌کند. این مسئله باقی می‌ماند که نشان دهیم وقتی RELABEL-TO-FRONT پایان می‌یابد، هیچ عملیات اصلی انجام نمی‌شود. باقی بحث درستی بر پایه‌ی ثابت حلقه‌ی زیر است:

- در هر تست در خط ۶ رویه‌ی RELABEL-TO-FRONT، لیست L یک مرتب‌سازی توپولوژیکی از رأس‌ها در شبکه‌ی قابل قبول $G_{f,h} = (V, E_{f,h})$ است، و هیچ رأسی قبل از u در لیست شار افزونی ندارد.
- آغاز: دقیقاً بعد از این که INITIALIZE-PREFLOW اجرا شد، داریم $s.h = |V|$، و برای هر $v \in V - \{s\}$ داریم $v.h = 0$. از آن جایی که $|V| \geq 2$ (چون V حداقل حاوی s و t است)، هیچ یالی نمی‌تواند قابل قبول باشد. بنابراین $E_{f,h} = \emptyset$، و هر ترتیبی از $V - \{s, t\}$ یک مرتب‌سازی توپولوژیکی از $G_{f,h}$ است.



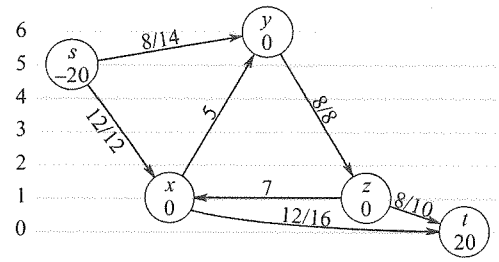
شکل ۲۶-۱۰ عملیات RELABEL-TO-FRONT. (الف) یک شبکه‌ی شار درست قبل از اولین تکرار حلقه‌ی while. در ابتدا ۲۶ واحد شار از منبع s خارج می‌شود. در سمت راست لیست اولیه‌ی $L = \langle x, y, z \rangle$ نشان داده شده است، که در ابتدا $u = x$. زیر هر رأس در لیست L، لیست همسایه‌های آن نشان داده شده است، که در آن همسایه‌ی فعلی سایه دارد. رأس x تخلیه می‌شود، با ارتفاع ۱ برچسب‌دهی مجدد می‌شود، ۵ واحد شار به y رانده می‌شود، و ۷ واحد شار افزونی باقی مانده به چاهک t رانده می‌شود. چون x برچسب‌دهی مجدد شده است، به ابتدای لیست L منتقل می‌شود، که در این حالت ساختار L را تغییر نمی‌دهد. (ب) بعد از x، رأس بعدی در L که تخلیه می‌شود y است. شکل ۲۶-۹ سر لیست L را نشان می‌دهد. (پ) اکنون رأس x در L بعد از y قرار دارد، و بنابراین دوباره تخلیه، و تمام ۵ واحد شار افزونی آن به t رانده می‌شود. چون رأس x در این تخلیه برچسب‌دهی مجدد می‌شود، در لیست L در مکان خود باقی می‌ماند. (ت) از آن جایی که رأس z در لیست L بعد از x قرار دارد، تخلیه می‌شود. این رأس با ارتفاع ۱ برچسب‌دهی مجدد شده و تمام ۸ واحد شار افزونی آن به t رانده می‌شود. چون x برچسب‌دهی مجدد شده است، به ابتدای لیست L منتقل می‌شود. (ث) اکنون رأس y در لیست L بعد از z قرار دارد و بنابراین تخلیه می‌شود. ولی چون y هیچ شار افزونی ندارد، DISCHARGE سریعاً بازگشت می‌کند، و y در L در مکان خود باقی می‌ماند. سپس رأس x تخلیه می‌شود. چون این رأس هم هیچ شار افزونی ندارد رویه‌ی DISCHARGE دوباره سریعاً بازگشت می‌کند، و x در لیست L در مکان خود باقی می‌ماند. RELABEL-TO-FRONT به انتهای لیست L رسیده و بازگشت می‌کند. اکنون هیچ رأس در حال سریزی وجود ندارد، و پیش‌شار همان شار بیشینه است.

(ت)



L:	y	x	z
N:	s	s	x
	x	y	y
	z	z	t
		t	

(ت)



L:	z	y	x
N:	x	s	s
	y	x	y
	t	z	z
			t

شکل ۲۶–۱۰ (ادامه)

چون u در ابتدا سر لیست L است، هیچ رأسی قبل از آن (و در نتیجه هیچ رأسی با شار افزونی قبل از آن) قرار ندارد.

• **ادامه:** برای این که ببینیم مرتب‌سازی توپولوژیکی در هر تکرار حلقه‌ی **while** حفظ می‌شود، ابتدا مشاهده می‌کنیم که شبکه‌ی قابل قبول فقط توسط اعمال رانش و برچسب‌دهی مجدد تغییر می‌کند. طبق لم ۲۶–۲۷، اعمال رانش یال‌ها را به یال قابل قبول تبدیل نمی‌کنند. بنابراین یال‌های قابل قبول فقط توسط اعمال برچسب‌دهی مجدد ساخته می‌شوند. با این حال بعد از این که یک رأس u برچسب‌دهی مجدد می‌شود، لم ۲۶–۲۸ می‌گوید که هیچ یال قابل قبول ورودی به u وجود ندارد، ولی ممکن است یال‌های قابل قبول خروجی از u وجود داشته باشند. بنابراین با انتقال u به ابتدای لیست L، الگوریتم اطمینان حاصل می‌کند که هر یال قابل قبول خروجی از u ترتیب توپولوژیکی را حفظ می‌کند.

برای این که ببینیم هیچ رأسی قبل از u در L شار افزونی ندارد، رأسی که در تکرار بعد u خواهد بود را با u' نشان خواهیم داد. رأس‌هایی که در تکرار بعد قبل از u' خواهند بود عبارتند از uی فعلی (بنا بر خط ۱۱) و، یا هیچ رأس دیگری (اگر u برچسب‌دهی مجدد شده باشد) یا همان رأس‌های قبلی (اگر u برچسب‌دهی مجدد نشده باشد). چون u تخلیه شده است، بعد از این هیچ شار افزونی نخواهد داشت.

بنابراین اگر u حین تخلیه، برچسب‌دهی مجدد شده باشد، هیچ رأسی قبل از u' شار افزونی ندارد. اگر u حین تخلیه برچسب‌دهی مجدد نشده باشد، چون L در کل زمان تخلیه به صورت توپولوژیکی مرتب مانده است (همان طور که در بالا مشخص شده است، یال‌های قابل قبول فقط با برچسب‌دهی مجدد ساخته می‌شوند، و نه با راندن)، بنابراین هر عملیات رانش باعث می‌شود که شار افزونی فقط به سمت رأس‌های پایین‌تر در لیست حرکت کند (یا به s یا t). دوباره، هیچ رأسی قبل از u' شار افزونی ندارد.

• پایان: وقتی حلقه پایان می‌یابد u تازه از انتهای لیست L گذشته است، و بنابراین ثابت حلقه اطمینان می‌دهد که شار افزونی تمام رأس‌ها ۰ است. پس هیچ عملیات اصلی کاربرد ندارد.

تحلیل

اکنون نشان خواهیم داد که RELABEL-TO-FRONT بر روی هر شبکه‌ی شار $G = (V, E)$ در زمان $O(V^3)$ اجرا می‌شود. چون این الگوریتم یک پیاده‌سازی از الگوریتم عام رانش-برچسب‌دهی مجدد است، از نتیجه‌ی ۲۶-۲۱ سود خواهیم برد، که یک کران $O(V)$ بر روی تعداد اعمال برچسب‌دهی مجدد اجرا شده بر روی هر رأس و یک کران $O(V^2)$ بر روی تعداد کل اعمال برچسب‌دهی مجدد فراهم می‌کند. به علاوه تمرین ۲۶-۴-۳ یک کران $O(VE)$ بر روی کل زمان صرف شده بر روی اعمال برچسب‌دهی مجدد به دست می‌دهد، و لم ۲۶-۲۲ یک کران $O(VE)$ بر روی تعداد کل رانش‌های اشباع کننده.

قضیه ۲۶-۳۰ زمان اجرای RELABEL-TO-FRONT بر روی یک شبکه‌ی شار $G = (V, E)$ برابر است با $O(V^3)$.

اثبات اجازه دهید یک «فاز» الگوریتم برچسب‌دهی مجدد-به جلو را به صورت زمان میان دو عملیات پشت سر هم برچسب‌دهی مجدد در نظر بگیریم. $O(V^2)$ فاز وجود دارد، چرا که $O(V^2)$ عملیات برچسب‌دهی مجدد داریم. هر فاز حداکثر شامل $|V|$ فراخوانی DISCHARGE است، که می‌توان آن را به صورت زیر مشاهده کرد. اگر DISCHARGE یک عملیات برچسب‌دهی مجدد انجام ندهد، در این صورت فراخوانی بعد DISCHRAGE در مکانی پایین‌تر در لیست L است، و طول L کم‌تر از $|V|$ است. اگر DISCHARGE یک عملیات برچسب‌دهی مجدد انجام دهد، فراخوانی بعدی DISCHARGE متعلق به یک فاز دیگر است. از آن جایی که هر فاز حداکثر شامل $|V|$ فراخوانی DISCHARGE است و $O(V^2)$ فاز داریم، تعداد دفعاتی که DISCHARGE در خط ۸ رویه‌ی RELABEL-TO-FRONT فراخوانی می‌شود $O(V^3)$ است. بنابراین کل کار انجام شده توسط حلقه‌ی while در RELABEL-TO-FRONT، غیر از کار انجام شده در DISCHARGE، حداکثر $O(V^3)$ است.

اکنون باید کرانی برای کار انجام شده در DISCHARGE حین اجرای الگوریتم تعیین کنیم. هر تکرار حلقه‌ی while در DISCHARGE یکی از سه عملیات را انجام می‌دهد. در این جا کل کار انجام شده در هر یک از این سه حالت را تحلیل خواهیم کرد.

با اعمال برچسب‌دهی مجدد (خطوط ۴-۵) شروع می‌کنیم. تمرین ۲۶-۴-۳ یک کران زمانی $O(VE)$ بر روی تمام $O(V^2)$ عملیات برچسب‌دهی مجدد انجام شده می‌دهد.

اکنون فرض کنید که عملیات انجام شده اشاره‌گر $u.current$ را در خط ۸ به هنگام سازی می‌کند. هر بار که یک رأس u برچسب‌دهی مجدد می‌شود، این عملیات $O(degree(u))$ بار رخ می‌دهد، و در کل $O(V \cdot degree(u))$ بار برای آن رأس. بنابراین برای تمام رأس‌ها، کل کار انجام شده در حرکت دادن

اشاره‌گرها در لیست‌های همسایه‌ها، طبق لم دست دادن (تمرین ب-۴-۱) $O(VE)$ است.

سومین نوع عملیات انجام شده توسط DISCHARGE عملیات رانش (خط ۷) است. از قبل می‌دانیم که تعداد کل رانش‌های اشباع کننده $O(VE)$ است. مشاهده کنید که اگر یک رانش غیر اشباع کننده اجرا شود، DISCHARGE سریعاً بازگشت می‌کند، چرا که رانش شار افزونی را به ۰ کاهش می‌دهد. بنابراین حداکثر می‌توانیم یک رانش غیر اشباع کننده برای هر فراخوانی DISCHARGE داشته باشیم. همان طور که دیدیم رویه‌ی DISCHARGE، $O(V^3)$ بار فراخوانی می‌شود، و بنابراین کل زمان صرف شده برای انجام رانش‌های غیر اشباع کننده $O(V^3)$ است.

پس زمان اجرای RELABEL-TO-FRONT برابر است با $O(V^3+VE)$، که همان $O(V^3)$ است. ■

تمرین‌ها

۲۶-۵-۱ اجرای RELABEL-TO-FRONT را به روش شکل ۲۶-۱۰ برای شبکه‌ی شار شکل ۲۶-۱(الف) مشخص کنید. فرض کنید که ترتیب اولیه‌ی رأس‌ها در L به صورت $\langle v_1, v_2, v_3, v_4 \rangle$ است، و لیست‌های همسایه‌ها عبارتند از

$$v_1.N = \langle s, v_2, v_3 \rangle,$$
$$v_1.N = \langle s, v_1, v_3, v_4 \rangle,$$
$$v_1.N = \langle v_1, v_2, v_4, t \rangle,$$
$$v_1.N = \langle v_2, v_3, t \rangle$$

★ ۲۶-۵-۲ می‌خواهیم یک الگوریتم رانش-برچسب‌دهی مجدد پیاده‌سازی کنیم که در آن یک صف اولین ورودی- اولین خروجی از رأس‌های در حال سرریز نگه می‌داریم. الگوریتم مکرراً رأسی که در ابتدای صف قرار دارد را تخلیه می‌کند، و هر رأسی که قبل از تخلیه در حال سرریز نبوده است ولی بعد از تخلیه در حال سرریز است را در انتهای صف قرار می‌دهد. بعد از این که رأس ابتدای صف تخلیه شد، از صف حذف می‌شود. وقتی صف خالی شود الگوریتم پایان می‌یابد. نشان دهید که چگونه می‌توان این الگوریتم را برای محاسبه‌ی یک شار بیشینه در زمان $O(V^3)$ پیاده‌سازی کرد.

۲۶-۵-۳ نشان دهید که اگر RELABEL، $u.h$ را فقط با محاسبه‌ی $u.h = u.h + 1$ به هنگام سازی کند، الگوریتم عام باز هم کار می‌کند. این تغییر تحلیل RELABEL-TO-FRONT را چگونه تغییر می‌دهد؟

★ ۲۶-۵-۴ نشان دهید که اگر همیشه بالاترین رأس در حال سرریز را تخلیه کنیم، می‌توان متد رانش- برچسب‌دهی مجدد را طوری پیاده‌سازی کرد که در زمان $O(V^3)$ اجرا شود.

۲۶-۵-۵ فرض کنید در لحظه‌ای در طول اجرای یک الگوریتم رانش-برچسب‌دهی مجدد، یک عدد

صحیح $0 < k \leq |V| - 1$ وجود دارد به طوری که برای هیچ رأسی عبارت $v.h = k$ برقرار نیست. نشان دهید که تمام رأس‌های با $v.h > k$ در سمت منبع یک برش کمینه قرار دارند. اگر چنین kای وجود داشته باشد، *مکاشفه‌ی شکاف* (gap heuristic) تمام رأس‌های $v \in V - s$ را که برای آن‌ها داریم $v.h > k$ به صورت $v.h = \max(v.h, |V| + 1)$ به هنگام‌سازی می‌کند. نشان دهید که خصیصه‌ی h حاصل یک تابع ارتفاع است. (مکاشفه‌ی شکاف در پیاده‌سازی کارامد متد رانش- برچسب‌دهی مجدد در عمل تعیین کننده است.)

مسائل

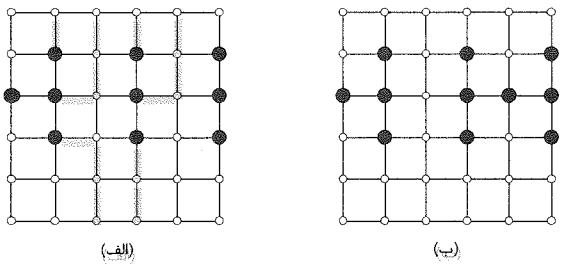
۲۶-۱ مسئله‌ی گریز

یک *شبکه‌ی* (grid) $n \times n$، یک گراف بدون جهت است شامل n ردیف و n ستون از رأس‌ها، همان طور که در شکل ۲۶-۱۱ نشان داده شده است. رأس ردیف iام و ستون jام را با (i, j) نشان می‌دهیم. تمام رأس‌ها در یک شبکه دقیقاً چهار همسایه دارند، به جز رأس‌های مرزی، که برای آن‌ها داریم $i = 1$، $i = n$، $j = 1$، یا $j = n$.

با داشتن $m \leq n^2$ نقطه‌ی شروع (x_1, y_1)، (x_2, y_2)،...، (x_m, y_m) در شبکه، مسئله‌ی گریز از ما می‌خواهد تعیین کنیم که آیا m مسیر مجزای رأسی از نقاط شروع به m نقطه‌ی مرزی مختلف وجود دارد یا خیر. به عنوان مثال، شبکه‌ی شکل ۲۶-۱۱(الف) یک گریز دارد، ولی شبکه‌ی شکل ۲۶-۱۱(ب) ندارد.

I. یک شبکه‌ی شار را در نظر بگیرید که در آن رأس‌ها مانند یال‌ها ظرفیت دارند. یعنی کل شار مثبت ورودی به هر رأس یک محدودیت دارد که با ظرفیت آن رأس تعیین می‌شود. نشان دهید که تعیین شار بیشینه در یک شبکه‌ی شار با رأس‌ها و یال‌های ظرفیت‌دار را می‌توان به یک مسئله‌ی شار بیشینه‌ی معمولی تبدیل کرد.

II. یک الگوریتم کارامد برای حل مسئله‌ی گریز ارائه و زمان اجرای آن را تحلیل کنید.



شکل ۲۶-۱۱ شبکه‌هایی برای مسئله‌ی گریز. نقاط شروع تیره، و بقیه‌ی رأس‌های شبکه روشن هستند. (الف) یک شبکه با یک گریز، که با مسیرهای سایه‌دار نشان داده شده است. (ب) یک شبکه که در آن گریزی وجود ندارد.

۲۶-۳ پوشش مسیر کمینه

یک *پوشش مسیر* (path cover) برای یک گراف جهت‌دار $G=(V,E)$ یک مجموعه‌ی P از مسیرهای مجزای رأسی است به طوری که تمام رأس‌ها در V دقیقاً در یک مسیر در P قرار دارند. شروع و پایان مسیرها می‌تواند هر جایی باشد، و طول آن‌ها هم می‌تواند هر مقداری باشد، از جمله ۰. یک *پوشش مسیر کمینه‌ی* (minimum path cover) G یک پوشش مسیر حاوی کم‌ترین تعداد مسیرهای ممکن است.

I. یک الگوریتم کارآمد برای یافتن یک پوشش مسیر کمینه در یک گراف جهت‌دار بدون دور $G=(V,E)$ بدهید. (*راهنمایی:* فرض کنید $V=\{1,2,...,n\}$، و گراف $G'=(V',E')$ را بسازید که در آن

$$V'=\{x_0,x_1,...,x_n\}\cup\{y_0,y_1,...,y_n\},$$
$$E'=\{(x_0,x_i):i\in V\}\cup\{(y_i,y_0):i\in V\}\cup\{(x_i,y_j):(i,j)\in E\}$$

و سپس الگوریتم شار بیشینه را اجرا کنید.)

II. آیا الگوریتم شما برای گراف‌های جهت‌داری که حاوی دور هستند هم کار می‌کند؟ توضیح دهید.

۲۶-۴ مشاوره‌ی الگوریتمیک

پروفسور Gore می‌خواهد یک مرکز مشاوره‌ی الگوریتمیک باز کند. او n زیرشاخه‌ی مهم از الگوریتم‌ها را شناسایی کرده است (که تقریباً مشابه بخش‌های مختلف این کتاب هستند)، و آن‌ها را با مجموعه‌ی $A=\{A_1,A_2,...,A_n\}$ نشان می‌دهد. برای هر زیرشاخه‌ی A_k، پروفسور می‌تواند با c_k دلار یک متخصص استخدام کند. مرکز مشاوره مجموعه‌ای از $J=\{J_1,J_2,...,J_m\}$ سفارش احتمالی جمع‌آوری کرده است. برای انجام سفارش J_i مرکز باید متخصصانی در زیرمجموعه‌ی $R_i\subseteq A$ از زیرشاخه‌ها استخدام کرده باشد. هر متخصص می‌تواند به طور هم‌زمان بر روی چندین سفارش کار کند. اگر مرکز تصمیم بگیرد که سفارش J_i را بپذیرد، باید حتماً متخصصانی در تمام زیرشاخه‌های R_i استخدام کرده باشد، و پس از انجام سفارش p_i دلار دریافت می‌کند.

وظیفه‌ی پروفسور Gore این است که تعیین کند برای بیشینه کردن سود خاصل، که برابر است با کل درآمد منهای کل هزینه‌ی استخدام متخصصان، مرکز باید چه متخصصانی را استخدام کند و کدام سفارشات را بپذیرد.

شبکه‌ی شار زیر (G) را در نظر بگیرید. این شبکه حاوی یک رأس منبع s است، به علاوه‌ی رأس‌های $A_1,A_2,...,A_n$، رأس‌های $J_1,J_2,...,J_m$، و یک رأس چاهک t. برای $k=1,2,...,n$، شبکه‌ی شار حاوی یک یال (s,A_k) است با ظرفیت $c(s,A_k)=c_k$، و برای $i=1,2,...,m$، حاوی یک یال (J_i,t) با ظرفیت $c(J_i,t)=p_i$. برای $k=1,2,...,n$ و

$i = ۱, ۲, ..., m$، اگر $A_k \in R_i$، آن گاه G حاوی یک یال (A_k, J_i) است با ظرفیت $c(A_k, J_i) = \infty$.

I. نشان دهید اگر برای یک برش (S, T) با ظرفیت محدود در G داشته باشیم $J_i \in T$، آن گاه برای هر $A_k \in R_i$ داریم $A_k \in T$.

II. نشان دهید چگونه می‌توان سود خالص بیشینه را از ظرفیت برش کمینه‌ی G و مقادیر داده شده‌ی p_i به دست آورد.

III. یک الگوریتم کارامد برای تعیین سفارشاتی که باید قبول کنیم و متخصصانی که باید استخدام کنیم ارائه کنید. زمان اجرای الگوریتم خود را بر حسب m، n، و $r = \sum_{i=۱}^{m} |R_i|$ تحلیل کنید.

۲۶-۴ به هنگام سازی شار بیشینه

فرض کنید $G = (V, E)$ یک شبکه‌ی شار با منبع s، چاهک t و ظرفیت‌های صحیح باشد. فرض کنید که یک شار بیشینه در G به ما داده شده است.

I. فرض کنید ظرفیت یک یال $(u, v) \in E$ یکی افزایش می‌یابد. یک الگوریتم با زمان $O(V + E)$ برای به هنگام سازی شار بیشینه ارائه کنید.

II. فرض کنید ظرفیت یک یال $(u, v) \in E$ یکی کاهش می‌یابد. یک الگوریتم با زمان $O(V + E)$ برای به هنگام سازی شار بیشینه ارائه کنید.

۲۶-۵ شار بیشینه با مقیاس‌دهی

فرض کنید $G = (V, E)$ یک شبکه‌ی شار با منبع s، چاهک t، و یک ظرفیت صحیح $c(u, v)$ برای هر یال $(u, v) \in E$ باشد. فرض کنید $C = \max_{(u,v)\in E} c(u, v)$.

I. نشان دهید که ظرفیت یک برش کمینه‌ی G حداکثر $C|E|$ است.

II. برای یک عدد داده شده‌ی K، نشان دهید که چگونه می‌توان در صورت وجود یک مسیر تکمیلی با حداقل ظرفیت K را در زمان $O(E)$ یافت.

از نسخه‌ی زیر از FORD-FULKERSON-METHOD می‌توان برای محاسبه‌ی یک شار بیشینه در G استفاده کرد.

```
MAX-FLOW-BY-SCALING(G, s, t)
1  C = max_{(u,v)∈E} c(u,v)
2  initialize flow f to 0
3  K = 2^⌊lg C⌋
4  while K ≥ 1
5      while there exists an augmenting path p of capacity at least K
6          augment flow f along p
```

7 $K = K/2$
8 return f

III. بحث کنید که MAX-FLOW-BY-SCALING یک شار بیشینه باز می‌گرداند.

IV. نشان دهید که هر بار که خط ۴ اجرا می‌شود، ظرفیت یک برش کمینه در شبکه‌ی پس‌ماند G_f حداکثر $2K|E|$ است.

V. بحث کنید که حلقه‌ی داخلی while در خطوط ۵-۶ برای هر مقدار K، $O(E)$ بار اجرا می‌شود.

VI. نتیجه بگیرید که MAX-FLOW-BY-SCALING را می‌توان طوری پیاده‌سازی کرد که در زمان $O(E^2 \lg C)$ اجرا شود.

۲۶-۶ الگوریتم تطابق دوبخشی هاپکرافت-کارپ

در این مسئله یک الگوریتم سریع‌تر برای یافتن یک تطابق بیشینه در یک گراف دو بخشی توصیف می‌کنیم، که منسوب به هاپکرافت (Hopcroft) و کارپ (Karp) است. الگوریتم در زمان $O(\sqrt{V}E)$ اجرا می‌شود. با داشتن یک گراف بدون جهت دوبخشی $G = (V, E)$، که در آن $V = L \cup R$ و تمام یال‌ها دقیقاً یک نقطه‌ی پایانی در L دارند، فرض کنید M یک تطابق در G باشد. می‌گوییم یک مسیر ساده‌ی P در G نسبت به M یک مسیر تکمیلی (augmenting path) است اگر در یک رأس تطبیق نیافته در L شروع شده، و در یک رأس تطبیق نیافته در R پایان یابد، و یال‌های آن یکی در میان به M و $E - M$ تعلق داشته باشند. (این تعریف از مسیر تکمیلی به تعریف یک مسیر تکمیلی در یک شبکه‌ی شار مربوط است، ولی با آن تفاوت دارد.) در این مسئله با یک مسیر به صورت دنباله‌ای از یال‌ها، به جای دنباله‌ای از رأس‌ها برخورد می‌کنیم. یک کوتاه‌ترین مسیر تکمیلی نسبت به یک تطابق M، یک مسیر تکمیلی با کم‌ترین تعداد یال‌ها است.

با داشتن دو مجموعه‌ی A و B، اختلاف متقارن (symmetric difference) $A \otimes B$ به صورت $(A - B) \cup (B - A)$ تعریف می‌شود، یعنی عناصری که دقیقاً به یکی از دو مجموعه تعلق دارند.

I. نشان دهید که اگر M یک تطابق و P یک مسیر تکمیلی نسبت به M باشند، آن‌گاه اختلاف متقارن $M \otimes P$ یک تطابق است، و $|M \otimes P| = |M| + 1$. نشان دهید که اگر P_1، P_2، ...، P_k مسیرهای تکمیلی مجزای رأسی نسبت به M باشند، آن گاه اختلاف متقارن $M \otimes (P_1 \cup P_2 \cup ... \cup P_k)$ یک تطابق با اندازه‌ی $|M| + k$ است.

ساختار کلی الگوریتم ما به شکل زیر است.

HOPCROFT-KARP(G)
1 M = ∅
2 repeat

```
3      let 𝒫 ← { P_1, P_2, ⋯, P_k} be a maximum set of vertex-disjoint
           shortest augmenting paths with respect to M
4      M = M ⊗ (P_1 ∪ P_2 ∪ ⋯ ∪ P_k)
5      until 𝒫 = ∅
6  return M
```

باقی این مسئله از شما می‌خواهد که تعداد تکرارها در الگوریتم (یعنی تعداد تکرارهای حلقه‌ی **repeat**) را تحلیل، و یک پیاده‌سازی برای خط ۳ توصیف کنید.

II. با داشتن دو تطابق M و M^* در G، نشان دهید که درجه‌ی هر رأس در گراف $G' = (V, M \otimes M^*)$ حداکثر ۲ است. نتیجه بگیرید که G یک اجتماع مجزا از مسیرها یا دورهای ساده است. بحث کنید که یال‌ها در هر مسیر یا دوری مانند بالا یکی در میان به M یا M^* تعلق دارند. اثبات کنید که اگر $|M^*| \le |M|$، آن گاه $M \otimes M^*$ حاوی حداقل $|M^*| - |M|$ مسیر تکمیلی مجزای رأسی نسبت به M است.

فرض کنید l طول کوتاه‌ترین مسیر تکمیلی نسبت به تطابق M باشد، و فرض کنید P_1، P_2،...، P_k یک مجموعه‌ی بیشینه از مسیرهای تکمیلی مجزای رأسی با طول l نسبت به M باشد. فرض کنید $M' = M \otimes (P_1 \cup \ldots \cup P_k)$، و این که P یک کوتاه‌ترین مسیر تکمیلی نسبت به M' است.

III. نشان دهید که اگر P از P_1، P_2،...، P_k مجزای رأسی باشد، آن گاه P بیش از l یال دارد.

IV. اکنون فرض کنید P از P_1، P_2،...، P_k مجزای رأسی نیست. فرض کنید A یک مجموعه از یال‌های $(M \otimes M') \otimes P$ باشد. نشان دهید که $A = (P_1 \cup P_2 \cup \ldots \cup P_k) \otimes P$ و $|A| \ge (k+1)l$. نتیجه بگیرید که P بیش از l یال دارد.

V. اثبات کنید که اگر یک کوتاه‌ترین مسیر تکمیلی برای M طول l داشته باشد، اندازه‌ی تطابق بیشینه حداکثر $|M| + |V|/(l+1)$ است.

VI. نشان دهید که تعداد تکرارهای حلقه‌ی **repeat** در الگوریتم حداکثر $2\sqrt{V}$ است. *(راهنمایی:* بعد از $\sqrt{V}$ امین تکرار، M چقدر می‌تواند رشد کند؟)

VII. یک الگوریتم برای یافتن یک مجموعه‌ی بیشینه از کوتاه‌ترین مسیرهای تکمیلی مجزای رأسی P_1، P_2،...، P_k برای یک تطابق داده شده‌ی M بدهید که در زمان $O(E)$ اجرا می‌شود. نتیجه بگیرید که زمان اجرای HOPCROFT-KARP برابر است با $O(\sqrt{V}E)$.

بخش هفتم مباحث منتخب

شامل فصل‌های :

۲۷ الگوریتم‌های چند ریسمانی
۲۸ اعمال ماتریس‌ها
۲۹ برنامه‌ریزی خطی
۳۰ چندجمله‌ای‌ها و تبدیل تبدیل سریع فوریه
۳۱ الگوریتم‌های نظریه‌ی اعداد
۳۲ تطابق رشته‌ها
۳۳ هندسه‌ی محاسباتی
۳۴ NP - کامل‌ها
۳۵ الگوریتم‌های تقریبی

مقدمه

این بخش حاوی منتخبی است از مباحث مربوط به الگوریتم‌ها که مطالب قبلی این کتاب را افزایش داده و کامل می‌کند. بعضی از فصل‌ها مدل‌های جدیدی از محاسبه ارائه می‌کنند، مانند مدارها و یا محاسبه‌گرهای موازی. در دو فصل آخر، بعضی از محدودیت‌های شناخته نشده‌ی موجود در طراحی الگوریتم‌ها مطرح می‌شوند، به علاوه‌ی چند تکنیک برای کنار آمدن با این محدودیت‌ها.

در فصل ۲۷ یک مدل الگوریتمیک برای محاسبات موازی بر مبنای چندریسمانی پویا (dynamic multithreading) ارائه خواهد شد. این فصل مبانی این مدل را معرفی خواهد کرد، و نشان خواهد داد که چطور می‌توان موازی‌سازی را به کمک معیارهای کار و دهانه، به کمیت تبدیل کرد. سپس الگوریتم‌های چندریسمانی جذاب و متعددی بررسی خواهد شد، از جمله الگوریتم‌هایی برای ضرب ماتریس‌ها و مرتب‌سازی درجی.

در فصل ۲۸، الگوریتم‌های بهینه برای انجام اعمال مختلف بر روی ماتریس‌ها بررسی می‌شوند. بعد از بررسی بعضی از خصوصیات اولیه‌ی ماتریس‌ها، الگوریتم Starssen را خواهیم دید که به کمک آن می‌توان دو ماتریس $n \times n$ را در زمان $O(n^{2/81})$ در هم ضرب کرد. سپس دو متد عام – تجزیه‌ی LU و تجزیه‌ی LUP – برای حل معادلات خطی با حذف گاوسی در زمان $O(n^3)$ معرفی می‌شود. همچنین نشان خواهیم داد که وارون یک ماتریس و ضرب دو ماتریس را می‌توان با سرعت مساوی محاسبه کرد. در پایان این فصل خواهیم دید که وقتی مجموعه‌ای از معادلات خطی جواب دقیق ندارند، چطور می‌توان با استفاده از روش کم‌ترین مربعات یک جواب تقریبی برای آن‌ها محاسبه کرد.

در فصل ۲۹ برنامه‌ریزی خطی بررسی می‌شود، که در آن می‌خواهیم با داشتن منابع محدود و تحت شرایط خاص، یک هدف را کمینه یا بیشینه کنیم. برنامه‌ریزی خطی در کاربردهای عملی مختلفی

پیش می‌آید. این فصل نحوه‌ی فرمول‌بندی و جواب دادن به برنامه‌های خطی را پوشش می‌دهد. راه حل ارائه شده، الگوریتم سیمپلکس است، که در واقع اولین الگوریتم برای حل برنامه‌های خطی می‌باشد. بر خلاف بسیاری از الگوریتم‌های این کتاب، الگوریتم سیمپلکس در بدترین حالت در زمان چندجمله‌ای اجرا نمی‌شود، ولی در حد خوبی کارامد است و در عمل استفاده‌ی فراوانی دارد.

در فصل ۳۰ نحوه‌ی انجام عملیات بر روی چندجمله‌ای‌ها را خواهیم آموخت، به علاوه‌ی یک تکنیک معروف برای پردازش سیگنال – تبدیل سریع فوریه (FFT) – که می‌توان از آن برای ضرب دو چندجمله‌ای درجه n در زمان $O(n \lg n)$ استفاده کرد. همچنین در این فصل پیاده‌سازی‌های بهینه‌ی FFT بررسی می‌شود، از جمله پیاده‌سازی به کمک یک مدار موازی.

در فصل ۳۱ الگوریتم‌های نظریه‌ی اعداد معرفی خواهند شد. پس از مرور نظریه‌ی اعداد پایه، الگوریتم اقلیدس برای محاسبه‌ی بزرگ‌ترین مقسوم‌علیه مشترک ارائه خواهد شد، و پس از آن الگوریتم‌هایی برای حل معادلات خطی پیمانه‌ای و به توان رساندن یک عدد به پیمانه‌ی یک عدد دیگر. سپس یک کاربرد مهم از الگوریتم‌های نظریه‌ی اعداد را خواهیم دید: سیستم رمزنگاری کلید عمومی RSA. این سیستم رمزنگاری نه تنها برای رمزنگاری پیام‌ها کاربرد دارد (به طوری که کسی که استراق سمع می‌کند نتواند آن‌ها را بخواهند) بلکه از آن می‌توان برای امضای دیجیتال هم استفاده کرد. سپس الگوریتم تصادفی Miller-Robin برای تست اول بودن اعداد معرفی خواهد شد، که از آن می‌توان برای یافتن اعداد اول بزرگ به صورت بهینه استفاده کرد – یک نیاز اساسی برای سیستم RSA. نهایتاً مکاشفه‌ی «رو»ی Pollard برای تجزیه‌ی اعداد صحیح را خواهیم دید، و در مورد بهترین روش‌های موجود برای تجزیه‌ی اعداد بحث خواهیم کرد.

در فصل ۳۲ مسئله‌ی یافتن تمام رخ‌دادهای یک رشته‌ی الگوی داده شده را در یک متن خواهیم آموخت، مسئله‌ای که در برنامه‌های ویرایش متن به کرات پیش می‌آید. پس از بررسی رویکرد ساده‌لوحانه، یک رویکرد ظریف منسوب به Rabin و Karp ارائه خواهد شد. سپس، بعد از نشان دادن یک الگوریتم بهینه بر مبنای اتوماتاهای متناهی، الگوریتم Knuth-Morris-Pratt معرفی خواهد شد، که با پردازش هوشمندانه‌ی الگو به کارایی بالایی دست می‌یابد.

هندسه‌ی محاسباتی موضوع فصل ۳۳ است. پس از بحث در مورد اصول اولیه‌ی هندسه‌ی محاسباتی، نشان خواهیم داد که چگونه می‌توان با استفاده از یک متد «جارو کردن»، تعیین کرد که آیا در مجموعه‌ای از پاره‌خط‌ها هیچ برخوردی وجود دارد یا نه. همچنین دو الگوریتم هوشمندانه برای یافتن پوسته‌ی محدب – پویش Graham و راهپیمایی Jarvis – قدرت متدهای جاروکردن را نشان می‌دهند. این فصل با یک الگوریتم بهینه برای یافتن نزدیک‌ترین جفت در میان مجموعه‌ای از نقاط در صفحه پایان می‌یابد.

فصل ۳۴ در مورد مسائل NP-کامل بحث می‌کند. بسیاری از مسائل محاسباتی جذاب، NP-کامل هستند، ولی هیچ الگوریتمی با زمان چندجمله‌ای برای حل هیچ یک از آن‌ها یافت نشده است. در این فصل تکنیک‌هایی ارائه خواهد شد برای تعیین این که یک مسئله NP-کامل هست یا نه. NP-کامل بودن مسائل کلاسیک بسیاری اثبات خواهد شد: تعیین وجود یک دور همیلتونی در یک گراف، تعیین

قابل ارضا بودن یک فرمول بولین، و تعیین این که آیا مجموعه‌ای داده شده از اعداد، زیرمجموعه‌ای دارد که جمع اعضای آن برابر یک مقدار هدف باشد یا خیر. همچنین در این فصل اثبات می‌شود که مسئله‌ی معروف فروشنده‌ی دوره‌گرد، NP–کامل است.

در فصل ۳۵ خواهیم دید که می‌توان با استفاده از الگوریتم‌های تقریبی، به صورت کارامد جواب‌های تقریبی برای مسائل NP–کامل یافت. برای بعضی از مسائل NP–کامل، ساختن جواب‌های تقریبی که به جواب بهینه نزدیک هستند کار ساده‌ای است، ولی برای بعضی دیگر، حتی بهترین الگوریتم‌های تقریبی یافت شده هم با بزرگ شدن اندازه‌ی مسئله، جواب‌های ضعیف‌تری تولید می‌کنند. همچنین مسائلی وجود دارند که با صرف زمان محاسبه‌ی بیشتر می‌توان به جواب‌های تقریبی بهتری برای آن‌ها دست یافت. در این فصل با استفاده از مسئله‌ی پوشش رأسی (نسخه‌های وزن‌دار و بدون وزن)، نسخه‌ی بهینه‌سازی از مسئله‌ی قابلیت ارضای 3-CNF، مسئله‌ی فروشنده‌ی دوره‌گرد، مسئله‌ی پوشش مجموعه، و مسئله‌ی جمع زیرمجموعه، انواع مختلف الگوریتم‌های تقریبی بررسی خواهند شد.

الگوریتم‌های چند ریسمانی

۲۷-۰ مقدمه

اکثریت مطلق الگوریتم‌های این کتاب، *الگوریتم‌های سریال* (serial algorithms) هستند، که برای اجرا روی کامپیوترهای تک‌پردازنده مناسب‌اند، که در آن‌ها در هر لحظه فقط یک دستورالعمل انجام می‌شود. در این فصل، مدل الگوریتمی خود را گسترش می‌دهیم تا الگوریتم‌های موازی (parallel algorithms) را هم در بر بگیرد، که می‌توانند بر روی کامپیوترهای چندپردازنده‌ای اجرا شوند. این کامپیوترها قابلیت اجرای چندین دستورالعمل به صورت هم‌زمان را دارند. به طور خاص، مدل هوشمندانه‌ی الگوریتم‌های چندریسمانی پویا را بررسی خواهیم کرد، که با تحلیل و طراحی الگوریتم‌ها سازگار هستند، همین‌طور با پیاده‌سازی کارامد در عمل.

کامپیوترهای موازی – کامپیوترهایی با چندین واحد پردازش – به سرعت همه‌گیر شده‌اند، و محدوده‌ی گسترده‌ای از قیمت‌ها و کارایی‌ها را پوشش می‌دهند. چندپردازنده‌های تراشه‌ای (chip multiprocessors) حاوی یک تراشه‌ی مدار مجتمع چندهسته‌ای (multi-core) هستند که دارای چندین «هسته»ی پردازنده است. هر یک از این هسته‌ها یک پردازنده‌ی کامل است که به حافظه‌ی اصلی دسترسی دارد. هنگام نیاز به کارایی متوسط، با پرداخت یک هزینه‌ی متوسط می‌توان خوشه‌ای از کامپیوترهای مجزا تهیه و آن‌ها را با یک شبکه‌ی مخصوص به هم متصل کرد. گران‌ترین ماشین‌ها ابرکامپیوترها هستند، که معمولاً از ترکیبی از معماری منحصر به فرد و شبکه‌ی منحصر به فرد بهره می‌برند تا بتوانند بالاترین کارایی ممکن را برحسب تعداد دستورالعمل‌های انجام شده بر هر ثانیه ارائه کنند.

کامپیوترهای چندپردازنده‌ای از چندین دهه پیش به شکل‌های مختلف وجود داشتند. با این که جامعه‌ی محاسباتی، مدل ماشین‌های با دسترسی تصادفی را از مدت‌ها پیش برای محاسبات سریال

پذیرفته بود، هیچ مدل خاصی برای محاسبات موازی این چنین مورد قبول واقع نشده بود. برای مثال بعضی از کامپیوترهای موازی از *حافظه‌ی مشترک* (shared memory) استفاده می‌کردند، که در آن هر پردازنده می‌تواند مستقیماً به هر قسمت از حافظه دسترسی پیدا کند. انواع دیگری از کامپیوترهای موازی از *حافظه‌ی توزیع‌شده* (distributed memory) استفاده می‌کردند، که در آن حافظه‌ی هر پردازنده خصوصی است، و برای این که یک پردازنده به حافظه‌ی پردازنده‌ی دیگر دسترسی داشته باشد، باید یک پیام صریح در این مورد فرستاده شود. ولی با ابتداع تکنولوژی چندهسته‌ای، اکنون هر لپ‌تاپ یا کامپیوتر رومیزی یک ماشین موازی با حافظه‌ی مشترک است، و به نظر می‌آید که گرایش اکثریت به سمت چندپردازشی با حافظه‌ی مشترک باشد. با این که پایان این روند هنوز مشخص نیست، ولی رویکرد ما در این فصل به همین صورت خواهد بود.

یک روش معمول برای برنامه‌ریزی چندپردازنده‌های تراشه‌ای یا کامپیوترهای موازی دیگر با حافظه‌ی مشترک، استفاده از *ریسمان‌سازی ایستا* (static threading) است، که یک تجرید نرم‌افزاری از «پردازنده‌های مجازی»، یا *ریسمان‌ها* (thread) فراهم می‌کند، و این ریسمان‌ها از یک حافظه‌ی مشترک استفاده می‌کنند. هر ریسمان یک شمارنده‌ی برنامه‌ی مختص به خود دارد، و می‌تواند مستقل از بقیه‌ی ریسمان‌ها کد مربوط به خود را اجرا کند. سیستم عامل می‌تواند در هنگام نیاز هر یک از ریسمان‌ها را روی پردازنده بارگذاری کند، یا آن را از پردازنده خارج کند تا ریسمان‌های دیگر روی پردازنده اجرا شوند. با این که سیستم عامل به برنامه‌نویس اجازه می‌دهد ریسمان‌ها را ساخته و یا از بین ببرند، این اعمال نسبتاً کند هستند. بنابراین برای اکثر کاربردها، ریسمان‌ها در طول کل پردازش باقی خواهند ماند، و به همین دلیل است که به آن‌ها «ایستا» می‌گوییم.

متأسفانه برنامه‌ریزی کامپیوتر موازی با حافظه‌ی مشترک، مستقیماً با استفاده از ریسمان‌های ایستا، مشکل و پرخطا است. یک دلیل این است که تقسیم‌بندی کارها به صورت پویا میان ریسمان‌ها به طوری که هر ریسمان تقریباً به یک اندازه کار دریافت کند، کار بسیار پیچیده‌ای است. به غیر از کاربردهای بسیار ساده، برای هر کاربردی برنامه‌نویس باید از پروتکل‌های پیچیده‌ی ارتباطی برای پیاده‌سازی یک برنامه‌ریز برای تقسیم‌بندی عادلانه‌ی کار استفاده کند. این وضعیت به ساخت *چارچوب‌های هم‌زمانی* (concurrency platforms) منجر شد، که لایه‌ای از نرم‌افزار هستند که منابع محاسبات موازی را هماهنگ، برنامه‌ریزی، و مدیریت می‌کنند. بعضی از چارچوب‌های هم‌زمانی به صورت کتاب‌خانه‌های زمان اجرا طراحی شده‌اند، و بعضی دیگر یک زبان برنامه‌نویسی موازی کامل ارائه می‌کنند، به همراه کامپایلر و پشتیبانی زمان اجرا.

برنامه‌نویسی چندریسمانی پویا

یکی از کلاس‌های مهم چارچوب‌های هم‌زمانی، *چندریسمانی پویا* (dynamic multithreading) است، که در این فصل هم از این مدل استفاده خواهد شد. چندریسمانی پویا به برنامه‌نویس اجازه می‌دهد که در کاربردها از موازی‌سازی استفاده کند، بدون این که نگران پروتکل‌های ارتباطی، توازن بار، و یا مشکلات دیگر برنامه‌نویسی چندریسمانی ایستا باشد. این چارچوب هم‌زمانی حاوی یک برنامه‌ریز است که با تقسیم بار محاسبات به صورت متوازن، به میزان قابل توجهی زحمت برنامه‌نویس را کاهش

می‌دهد. با این که کارایی محیط‌های چندریسمانی پویا هنوز در حال تکامل است، تقریباً تمام آن‌ها از دو ویژگی پشتیبانی می‌کنند: موازی‌سازی تودرتو و حلقه‌های موازی. موازی‌سازی تودرتو به زیرروال‌ها اجازه می‌دهد «تکثیر» (spawn) شود، که عبارت است از ادامه‌ی اجرای تابع فراخوانی‌کننده در حالی که زیرروال فراخوانی شده هم‌زمان در حال محاسبه‌ی نتیجه‌ی خود است. یک حلقه‌ی موازی مشابه یک حلقه‌ی for معمولی است، غیر از این که تکرارهای حلقه می‌توانند به صورت هم‌زمان اجرا شوند.

این دو ویژگی پایه‌ی مدل چندریسمانی پویا را تشکیل می‌دهند که ما در این فصل آن را بررسی می‌کنیم. یک ویژگی کلیدی این مدل این است که برنامه‌نویس فقط باید هم‌زمانی منطقی درون محاسبات را مشخص کند، و ریسمان‌های درون چارچوب زیرین، برنامه‌ریزی و تقسیم بار محاسبات را به خودی خود انجام می‌دهند. در این فصل الگوریتم‌های چندریسمانی نوشته شده برای این مدل را بررسی خواهیم کرد، به علاوه‌ی نحوه‌ی برنامه‌ریزی محاسبات توسط چارچوب هم‌زمانی زیرین به صورت کارامد.

مدل ما برای چندریسمانی پویا چندین مزیت مهم دارد:

- این مدل، یک گسترش ساده از مدل برنامه‌نویسی سریالی است که تا این جا از آن استفاده کردیم. به سادگی و فقط با اضافه کردن سه کلمه‌ی کلیدی «هم‌زمانی» به شبه‌کدها، می‌توانیم یک الگوریتم چندریسمانی را توصیف کنیم: parallel، spawn، و sync. به علاوه اگر این کلمات کلیدی هم‌زمانی را از شبه‌کدها حذف کنیم، متن باقی‌مانده یک شبه‌کد سریال برای همان مسئله خواهد بود، که آن را «سریال‌سازی» الگوریتم چندریسمانی می‌نامیم.
- این مدل یک روش نظری واضح برای توصیف کمی موازی‌سازی فراهم می‌کند، که بر پایه‌ی مفاهیم «کار» و دهانه بنا شده است.
- بسیاری از الگوریتم‌های چندریسمانی حاوی موازی‌سازی تودرتو، به صورت طبیعی از رویکرد تقسیم و حل ناشی می‌شوند. به علاوه درست همان طور که الگوریتم‌های تقسیم و حل سریال را می‌توان با استفاده از حل رابطه‌های بازگشتی تحلیل کرد، الگوریتم‌های چندریسمانی هم همین گونه هستند.
- این مدل به روند تکامل محاسبات موازی پایبند است. تعداد زیادی از چارچوب‌های هم‌زمانی از یکی از نسخه‌های چندریسمانی پویا پشتیبانی می‌کنند، از جمله Cilk، Cilk++، OpenMP، Task Parallel Library، و Threading Building Blocks.

بخش ۲۷-۱ مدل چندریسمانی پویا را معرفی و ماتریس‌های کار، دهانه، و هم‌زمانی را ارائه می‌کند، که از آن‌ها برای تحلیل الگوریتم‌های چندریسمانی استفاده خواهیم کرد. در بخش ۲۷-۲ خواهیم دید که چطور با استفاده از موازی‌سازی می‌توان ماتریس‌ها را در هم ضرب کرد، و بخش ۲۷-۳ به مسئله‌ی مشکل‌تر چندریسمانی‌سازی مرتب‌سازی ادغامی می‌پردازد.

۲۷-۱ مبانی چندریسمانی‌سازی پویا

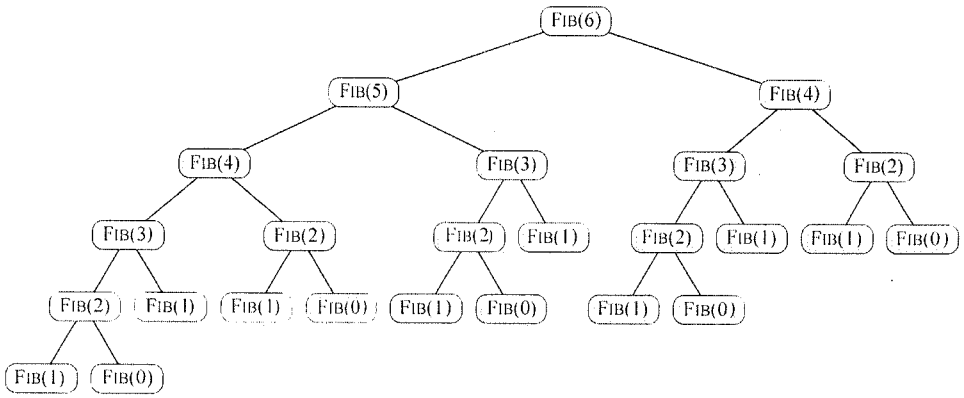
توضیح چندریسمانی‌سازی پویا را با استفاده از مثال محاسبه‌ی اعداد فیبوناچی به صورت بازگشتی آغاز می‌کنیم. به خاطر بیاورید که اعداد فیبوناچی توسط رابطه‌ی بازگشتی (۳-۲۲) تعریف می‌شوند:

$$F_0 = 0,$$
$$F_1 = 1,$$
$$F_i = F_{i-1} + F_{i-2} \quad \text{برای } i \geq 2$$

در زیر یک الگوریتم سریال ساده و بازگشتی را می‌بینیم که n امین عدد فیبوناچی را محاسبه می‌کند:

```
FIB(n)
1  if n ≤ 1
2      return n
3  else x = FIB(n-1)
4       y = FIB(n-2)
5       return x + y
```

محاسبه‌ی اعداد فیبوناچی بزرگ به این روش اصلاً کار مناسبی نیست، چرا که در آن محاسبات تکراری زیادی وجود دارد. شکل ۲۷-۱ درخت رویه‌های بازگشتی را برای محاسبه‌ی F_6 نشان می‌دهد. برای مثال، فراخوانی FIB(6) به صورت بازگشتی FIB(5) و FIB(4) را فراخوانی می‌کند. ولی از طرفی فراخوانی FIB(5) هم FIB(4) را فراخوانی می‌کند. هر دو نمونه‌ی FIB(4) مقدار یکسانی را بازمی‌گردانند ($F_4 = 3$). از آن جایی که رویه‌ی FIB به خاطرسپاری انجام نمی‌دهد، دومین فراخوانی FIB(4) کار انجام شده توسط فراخوانی اول را تکرار می‌کند.



شکل ۲۷-۱ درخت فراخوانی‌های بازگشتی هنگام محاسبه‌ی FIB(6). تمام نمونه‌های FIB با آرگومان ورودی یکسان، کار یکسانی را انجام داده و نتیجه‌ی یکسانی تولید می‌کنند. این روش، روشی ناکارمد ولی جالب برای محاسبه‌ی اعداد فیبوناچی است.

فرض کنید $T(n)$ نشان دهنده‌ی زمان اجرای FIB(n) باشد. چون FIB(n) حاوی دو فراخوانی بازگشتی است به علاوه‌ی مقداری ثابتی کار اضافی، به رابطه‌ی بازگشتی زیر می‌رسیم:

$$T(n) = T(n-1) + T(n-2) + \theta(1)$$

این رابطه‌ی بازگشتی دارای جواب $T(n) = \theta(F_n)$ است، که می‌توانیم آن را به کمک متد جانشین‌سازی اثبات کنیم. به عنوان یک گام استقرا، فرض کنید $T(n) \le aF_n - b$، که در آن $a > 1$ و $b > 0$ ثابت هستند. با جانشین‌سازی به دست می‌آوریم

$$\begin{aligned} T(n) &\le (aF_{n-1} - b) + (aF_{n-2} - b) + \theta(1) \\ &= a(F_{n-1} + F_{n-1}) - 2b + \theta(1) \\ &= aF_n - b - (b - \theta(1)) \\ &\le aF_n - b \end{aligned}$$

که در آن باید b را به اندازه‌ی کافی بزرگ انتخاب کنیم که بر ثابت درون $\theta(1)$ غلبه کند. سپس می‌توانیم a را به اندازه‌ی کافی بزرگ انتخاب کنیم به طوری که شرط اولیه را ارضا کند. اکنون کران تحلیلی

$$T(n) = \theta(\phi^n) \qquad (۱-۲۷)$$

که در آن $\phi = (1+\sqrt{5})/2$ نسبت طلایی است، از تساوی (۳-۲۵) نتیجه می‌شود. چون F_n نسبت به n از مرتبه‌ی نمایی بزرگ می‌شود، این رویه عملاً روشی کند برای محاسبه‌ی اعداد فیبوناچی است. (مسئله‌ی ۳۱-۳ را برای روش‌های بسیار سریع‌تر ببینید.)

با این که رویه‌ی FIB روشی ضعیف برای محاسبه‌ی اعداد فیبوناچی است، ولی مثالی مناسب ارائه می‌کند برای توصیف مفاهیم کلیدی در تحلیل الگوریتم‌های چندریسمانی. مشاهده کنید که در FIB(n)، دو فراخوانی بازگشتی FIB(n-1) و FIB(n-2) در خطوط ۳ و ۴ مستقل از یکدیگر هستند: می‌توان آن دو را به هر ترتیبی فراخوانی کرد، و محاسبه‌ی انجام شده در یکی به هیچ وجه روی دیگری اثری نمی‌گذارد. پس این دو فراخوانی بازگشتی می‌توانند به صورت موازی اجرا شوند.

با اضافه کردن کلمات کلیدی موازی‌سازی **spawn** و **sync** به شبه‌کد قبلی، آن را گسترش می‌دهیم تا از محاسبات موازی پشتیبانی کند. در زیر می‌بینیم که چطور می‌توان رویه‌ی FIB را طوری بازنویسی کرد که از چندریسمانی پویا استفاده کند:

```
P-FIB(n)
1  if n ≤ 1
2      return n
3  else x = spawn P-FIB(n-1)
4       y = P-FIB(n-2)
5       sync
6       return x + y
```

توجه کنید که اگر کلمات کلیدی **spawn** و **sync** را از P-FIB حذف کنیم، شبه‌کد حاصل دقیقاً مشابه FIB خواهد بود (غیر از نام رویه در سرآیند و دو فراخوانی بازگشتی). سریال‌سازی یک الگوریتم

چندریسمانی را به صورت الگوریتم سریالی تعریف می‌کنیم که از حذف کلمات چندریسمانی به دست می‌آید: spawn و sync، و هنگام استفاده از حلقه‌های موازی، parallel. مسلماً شبه‌کد چندریسمانی ما این خصوصیت خوب را دارد که سریال‌سازی آن، همیشه یک الگوریتم معمولی سریال برای حل همان مسئله است.

موازی‌سازی تودرتو (nested parallelism) زمانی رخ می‌دهد که کلمه‌ی کلیدی spawn قبل از فراخوانی یک رویه قرار گیرد، مانند خط ۳. مفهوم تکثیر (spawn) با یک فراخوانی معمولی تفاوت دارد، چرا که رویه‌ای که تکثیر را فراخوانی می‌کند –*پدر* (parent)– می‌تواند به صورت موازی با زیرروال تکثیر شده –*فرزند* (child)– به اجرا ادامه دهد، و برعکس اجرای سریال نیازی ندارد منتظر پایان رویه‌ی فرزند بماند. در این جا در حالی که فرزند تکثیر شده در حال محاسبه‌ی P-FIB(n-1) است، پدر می‌تواند به صورت موازی با فرزند به محاسبه‌ی P-FIB(n-2) در خط ۴ بپردازد. از آن جایی که رویه‌ی P-FIB بازگشتی است، این دو فراخوانی بازگشتی هم موازی‌سازی تودرتو انجام می‌دهند، و همین‌طور فرزندان آن‌ها، که می‌تواند درختی بزرگ از زیرمحاسبات موازی تولید کند.

ولی کلمه‌ی کلیدی spawn نمی‌گوید یک رویه *باید* هم‌زمان با فرزند تکثیر شده اجرا شود، و فقط می‌گوید که این کار ممکن است. کلمه‌های کلیدی هم‌زمانی، *موازی‌سازی منطقی* (logical parallelism) محاسبات را توصیف می‌کنند، که مشخص می‌کند کدام بخش‌های محاسبات می‌توانند به صورت موازی اجرا شوند. در زمان اجرا این *برنامه‌ریز* (scheduler) است که تعیین می‌کند کدام یک از زیرمحاسبات واقعاً به صورت هم‌زمان اجرا شوند، و با پیش رفتن محاسبات، پردازنده‌های موجود را بین آن‌ها تقسیم می‌کند. در مورد مفاهیم نظری پشت برنامه‌ریزها به زودی بحث خواهیم کرد.

یک رویه نمی‌تواند به صورت ایمن از مقادیر بازگردانده شده توسط فرزندان تکثیر شده‌ی خود استفاده کند تا وقتی که عبارت sync را اجرا کند (خط ۵). کلمه‌ی کلیدی sync می‌گوید که رویه باید در صورت نیاز صبر کند تا تمام فرزندان تکثیر شده‌اش کامل شوند، و بعد می‌تواند عبارت بعد از sync را اجرا کند. در رویه‌ی P-FIB، یک sync قبل از عبارت return در خط ۶ نیاز است تا از ناهنجاری ناشی از جمع x و y قبل از اتمام محاسبه‌ی x جلوگیری شود. علاوه بر هم‌زمان‌سازی صریح فراهم شده توسط عبارت sync، هر رویه به صورت ضمنی قبل از خروج یک دستور sync اجرا می‌کند، که تضمین می‌کند تمام فرزندانش قبل از آن پایان یابند.

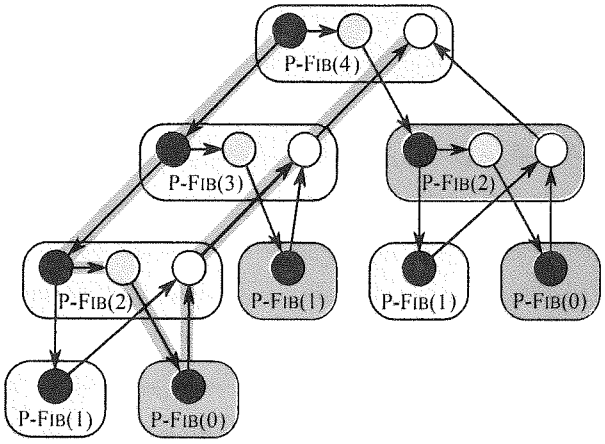
مدلی برای اجرای چندریسمانی

نگاه به یک *محاسبه‌ی چندریسمانی* (multithreaded computation) – مجموعه‌ی دستورالعمل‌های اجرا شده توسط یک پردازنده به جای برنامه‌ی چندریسمانی – به صورت یک گراف جهت‌دار بدون دور $G = (V, E)$، که *دگ محاسبات* (computation dag) نام دارد، به درک آن کمک می‌کند. به عنوان یک مثال، شکل ۲۷-۲ دگ محاسبات حاصل از محاسبه‌ی P-FIB(4) را نشان می‌دهد. از نظر مفهومی رأس‌های V دستورالعمل‌ها هستند، و یال‌های E وابستگی‌های بین دستورالعمل‌ها را نشان می‌دهند،

که در آن $(u,v) \in E$ نشان می‌دهد که دستورالعمل u باید قبل از دستورالعمل v اجرا شود. هرچند برای سادگی، اگر زنجیره‌ای از دستورالعمل‌ها حاوی هیچ کنترل موازی (دستورهای spawn، sync، یا return از یک تکثیر – یا به صورت return صریح، یا به صورت بازگشت ضمنی هنگام رسیدن به انتهای رویه) نباشد، آن‌ها را به صورت یک *طناب* (strand) گروه‌بندی می‌کنیم، که هر کدام از آن‌ها نشان دهنده‌ی یک یا چند دستورالعمل است. دستورالعمل‌های حاوی کنترل‌های موازی درون طناب‌ها قرار نمی‌گیرند، ولی در ساختار دگ نشان داده می‌شوند. برای مثال اگر یک طناب دو عنصر مابعد داشته باشد، یکی از آن‌ها باید تکثیر شده باشد، و یک طناب با چند پردازنده نشان می‌دهد که پردازنده‌ها به خاطر یک عبارت sync به هم متصل شده‌اند. بنابراین در حالت کلی، مجموعه‌ی V، مجموعه‌ی طناب‌ها را تشکیل می‌دهد، و مجموعه‌ی E از یال‌های جهت‌دار، نشان دهنده‌ی وابستگی‌های بین طناب‌ها است که توسط کنترل موازی القا می‌شود. اگر G یک مسیر جهت‌دار از طناب u به طناب v داشته باشد، می‌گوییم که دو طناب *(منطقاً) سری* هستند. در غیر این صورت طناب‌های u و v *(منطقاً) موازی* هستند.

می‌توانیم یک محاسبه‌ی چندریسمانی را به صورت یک دگ از طناب‌ها تصور کنیم که در درختی از نمونه‌های رویه‌ها جاسازی شده است. برای مثال شکل ۲۷–۱ درختی از نمونه‌ی رویه‌ها را برای P-FIB(6) نشان می‌دهد، بدون ساختار جزئی نشان‌دهنده‌ی طناب‌ها. شکل ۲۷–۲ روی یک بخش از آن درخت تمرکز کرده و طناب‌های تشکیل‌دهنده‌ی هر رویه را نشان می‌دهد. تمام یال‌های جهت‌دار متصل‌کننده‌ی طناب‌ها یا درون یک رویه اجرا می‌شوند، یا به صورت هم‌سو با یال‌های بدون جهت در درخت رویه‌ها.

می‌توانیم برای تعیین نوع وابستگی‌های بین طناب‌های مختلف، یال‌های درون دگ محاسبات را دسته‌بندی کنیم. یک *یال ادامه* (continuation edge) (u,u') ، که در شکل ۲۷–۲ به صورت افقی کشیده شده است، یک طناب u را به عنصر ماقبل خود u' در همان نمونه‌ی رویه متصل می‌کند. وقتی یک طناب u یک طناب v را تکثیر می‌کند، دگ حاوی یک *یال تکثیر* (spawn edge) (u,v) خواهد بود، که در شکل به سمت پایین اشاره می‌کند. *یال‌های فراخوانی* (call edge) هم که نشان‌دهنده‌ی فراخوانی‌های معمولی رویه‌ها هستند، به سمت پایین اشاره می‌کنند. تفاوت تکثیر طناب v توسط طناب u با فراخوانی v توسط u در این است که یک تکثیر، یک یال ادامه‌ی افقی از u به طناب u' بعد از خود در رویه به وجود می‌آورد، که مشخص‌کننده‌ی این است که u' می‌تواند هم‌زمان با v اجرا شود، در حالی که یک فراخوانی چنین یالی به وجود نمی‌آورد. وقتی یک طناب u به رویه‌ی فراخوانی کننده‌ی خود بازمی‌گردد، و اگر x طناب دقیقاً بعد از عبارت sync بعدی در رویه‌ی فراخوانی کننده باشد، دگ محاسبات حاوی *یال بازگشت* (return edge) (u,x) خواهد بود، که به سمت بالا اشاره می‌کند. هر محاسبه با یک *طناب آغازین* (initial strand) آغاز می‌شود – رأس تیره در رویه‌ی با برچسب P-FIB(4) در شکل ۲۷–۲ – و با یک *طناب نهایی* (final strand) پایان می‌یابد – رأس سفید در رویه‌ی با برچسب P-FIB(4).



شکل ۲۷-۲ یک گراف جهت‌دار بدون دور نشان دهنده‌ی محاسبه‌ی P-FIB(4). هر دایره یک طناب را نشان می‌دهد، و دایره‌های تیره یا نشان‌دهنده‌ی حالت‌های پایه هستند، و یا مشخص‌کننده‌ی بخشی از (نمونه‌ی) رویه که P-FIB(n-1) را در خط ۳ تکثیر می‌کند. دایره‌های سایه‌دار نشان‌دهنده‌ی بخشی از رویه هستند که P-FIB(n-2) را در خط ۴ فراخوانی می‌کند، تا دستور sync در خط ۵، که رویه در این جا منتظر می‌ماند تا تکثیر P-FIB(n-1) بازگشت کند، و دایره‌های سفید نشان دهنده‌ی قسمتی از رویه هستند که از بعد از sync و هنگام جمع x و y آغاز شده و تا بخش بازگرداندن نتیجه ادامه دارد. هر گروه از طناب‌های متعلق به یک رویه‌ی خاص درون یک شبه مستطیل قرار داده شده‌اند، که برای رویه‌های تکثیر شده با سایه‌ی کم‌رنگ، و برای رویه‌های فراخوانی شده با سایه‌ی پررنگ مشخص شده است. یال‌های تکثیر و یال‌های فراخوانی به سمت پایین اشاره می‌کنند، یال‌های ادامه به صورت افقی به سمت راست اشاره می‌کنند، و یال‌های بازگشت به سمت بالا. با فرض این که هر طناب، یک واحد زمان می‌برد، کار انجام شده برابر است با ۱۷ واحد زمانی، چرا که ۱۷ طناب وجود دارد، و دهانه برابر است با ۸ واحد زمان، چرا که مسیر بحرانی – که با یال‌های سایه‌دار مشخص شده است – حاوی ۸ طناب است.

اجرای الگوریتم‌های چندریسمانی را روی یک *کامپیوتر موازی ایده‌آل* (ideal parallel computer) فراخواهیم گرفت، که حاوی مجموعه‌ای از پردازنده‌ها است، به همراه یک حافظه‌ی مشترک *سازگار ترتیبی* (sequentially consistent). سازگاری ترتیبی بدین معنی است که حافظه‌ی مشترک، که در واقعیت ممکن است در یک لحظه بارگذاری‌ها و ذخیره‌های متعددی از طرف پردازنده‌های مختلف انجام دهد، همان نتیجه‌ای را تولید می‌کند که در صورت انجام دقیقاً یک دستورالعمل در هر مرحله توسط یک پردازنده تولید می‌کرد. به عبارت دیگر حافظه طوری رفتار می‌کند که انگار دستورالعمل‌ها به صورت سریال و بر مبنای یک ترتیب خطی فراگیر اجرا می‌شوند که ترتیب داده شده توسط هر یک از پردازنده‌ها را حفظ می‌کند. برای محاسبات چندریسمانی پویا، که توسط چارچوب هم‌زمانی به صورت خودکار برای پردازنده‌ها برنامه‌ریزی می‌شود، حافظه‌ی مشترک طوری عمل می‌کند که انگار دستورالعمل‌های محاسبات چندریسمانی طوری در میان هم جایگذاری شده‌اند که یک ترتیب خطی

ایجاد کنند که ترتیب درون دَگ محاسبات را حفظ می‌کند. بسته به برنامه‌ریزی، این ترتیب می‌تواند در اجراهای مختلف برنامه متفاوت باشد، ولی با فرض این که دستورالعمل‌ها با یک ترتیب خطی اجرا شده‌اند که با دَگ محاسبات سازگار است، می‌توان رفتار هر اجرایی را تحلیل کرد.

علاوه بر این فرض‌های مفهومی، کامپیوتر موازی ایده‌آل فرض‌هایی هم در مورد کارایی می‌کند. به طور خاص، فرض می‌کند که تمام پردازنده‌های ماشین قدرت محاسباتی یکسانی دارند، و همچنین از هزینه‌ی برنامه‌ریزی صرف نظر می‌کند. با این که این فرض آخر ممکن است خوشبینانه به نظر برسد، مشخص خواهد شد که برای الگوریتم‌های با «موازی‌سازی» (عبارتی که به زودی به طور دقیق تعریف خواهیم کرد) کافی، سربار برنامه‌ریزی معمولاً در عمل بسیار ناچیز است.

معیارهای کارایی

می‌توانیم کارایی تئوری یک الگوریتم چندریسمانی را بـه کمـک دو مقیـاس انـدازه بگیـریم: «کـار» و «دهانه». *کار* (work) انجام شده در یک محاسبه‌ی چندریسمانی برابر است با کل زمان مورد نیاز بـرای اجرای تمام محاسبات بر روی یک پردازنده. به عبارت دیگر، کار برابـر اسـت بـا مجمـوع زمـان‌هـای صرف شده توسط هر یک از طناب‌ها. برای یک دَگ محاسباتی که در آن هر طناب یـک واحـد زمـان می‌گیرد، کار به سادگی برابر است با تعداد رأس‌ها در دَگ. *دهانه* (span) عبارت است از طولانی‌تـرین زمان برای اجرای طناب‌ها روی هر مسیری در دَگ. دوباره، برای دَگی که در آن طناب‌ها به زمان واحد نیاز دارند، دهانه برابر است با تعداد رأس‌ها روی طولانی‌ترین مسیر، یا *مسیر بحرانی* (critical path) در دَگ. (از بخش ۲۴-۲ به خاطر بیاورید که می‌توانیم یک مسیر بحرانی در دَگ $G=(V,E)$ را در زمـان $\theta(V+E)$ بیابیم.) برای مثال، دَگ محاسبات شکل ۲۷-۲ تعداد ۱۷ رأس در کل و ۸ رأس روی مسـیر بحرانی دارد، پس اگر هر طناب به زمان واحد نیاز داشته باشد، کار آن برابر است با ۱۷ واحد زمـان، و دهانه‌ی آن برابر است با ۸ واحد زمان.

زمان اجرای واقعی یک برنامه‌ی چندریسمانی علاوه بر کار و دهانه، به تعداد پردازنده‌های موجـود و نحوه‌ی تخصیص طناب‌ها به پردازنده‌ها توسط برنامه‌ریز هم بسـتگی دارد. بـرای نشـان دادن زمـان اجرای یک محاسبه‌ی چندریسمانی بر روی P پردازنده از زیرنویس P استفاده خـواهیم کـرد. بـرای مثال، ممکن است زمان اجرای یک الگوریتم روی P پردازنده را با T_P نشان دهیم. کار برابر اسـت بـا زمان اجرا بر روی یک پردازنده، یا T_1. دهانه برابر است با زمان اجرا در حالتی که بتوانیم هر طناب را بر روی پردازنده‌ی مختص به خودش اجرا کنیم – به عبارت دیگر، اگـر تعـداد نامحـدودی پردازنـده داشته باشیم – و بنابراین دهانه را با T_∞ نشان می‌دهیم.

به کمک کار و دهانه می‌توان کران‌های پایین برای زمان اجرای یک محاسـبه‌ی چندریسـمانی روی P پردازنده (T_P) ارائه کرد:

- در یک مرحله، یک کامپیوتر موازی ایده‌آل با P پردازنده می‌تواند حداکثر P واحد کار انجـام دهد، و بنابراین در T_P واحد زمان می‌تواند حداکثر PT_P واحد کار انجام دهد. چون تمام کـار مورد نیاز برابر است با T_1، داریم $PT_P \geq T_1$. تقسیم بر P، *قانون کـار* (work law) را بـه مـا

می‌دهد:

$$T_P \geq T_1/P \qquad (۲-۲۷)$$

- یک کامپیوتر موازی ایده‌آل P -پردازنده‌ای نمی‌تواند سریع‌تر از یک ماشین با تعداد نامحدودی پردازنده عمل کند. در نگاهی دیگر، یک ماشین با تعداد نامحدودی پردازنده می‌تواند یک ماشین P پردازنده‌ای را با استفاده از P تا از پردازنده‌هایش شبیه‌سازی کند. بنابراین قانون دهانه را داریم:

$$T_P \geq T_\infty \qquad (۳-۲۷)$$

تسریع (speedup) یک محاسبه بر روی P پردازنده را به صورت نسبت T_1/T_P تعریف می‌کنیم، که می‌گوید محاسبه بر روی P پردازنده چند برابر سریع‌تر از همان محاسبه روی یک پردازنده است. طبق قانون کار داریم $T_P \geq T_1/P$، که ایجاب می‌کند $T_1/T_P \leq P$. بنابراین تسریع بر روی P پردازنده حداکثر می‌تواند P باشد. وقتی تسریع نسبت به تعداد پردازنده‌ها خطی باشد، یعنی وقتی $T_1/T_P = \theta(P)$، محاسبه از *تسریع خطی* (linear speedup) پیروی می‌کند، و وقتی $T_1/T_P = P$ ، *تسریع خطی کامل* (perfect linear speedup) را داریم.

نسبت کار به دهانه، که برابر است با T_1/T_∞ ، میزان *موازات* (parallelism) محاسبه‌ی چندریسمانی را به دست می‌دهد. موازات را می‌توانیم از سه جنبه نگاه کنیم. به عنوان یک نسبت، موازات میزان متوسط کاری را نشان می‌دهد که می‌تواند به صورت موازی برای هر مرحله از مسیر بحرانی انجام شود. به عنوان یک کران بالا، موازات، تسریع بیشینه‌ای را می‌دهد که می‌تواند برای هر تعدادی از پردازده‌ها به آن دست یافت. نهایتاً و احتمالاً مهم‌تر از همه، موازات، حدی می‌دهد برای احتمال رسیدن به تسریع خطی کامل. به طور خاص وقتی تعداد پردازده‌ها از موازات بیشتر می‌شود، امکان رسیدن محاسبه به تسریع خطی کامل وجود نخواهد داشت. برای درک این نکته‌ی آخر، فرض کنید داشته باشیم $P > T_1/T_\infty$ ، که در این حالت قانون دهانه می‌گوید که تسریع باید $T_1/T_P \leq T_1/T_\infty < P$ را ارضا کند. به علاوه اگر P (تعداد پردازنده‌ها در کامپیوتر موازی ایده‌آل) به شدت از موازات فراتر رود – یعنی اگر $P >> T_1/T_\infty$ – آن گاه $T_1/T_P << P$، و تسریع به شدت کم‌تر از تعداد پردازنده‌ها خواهد بود. به عبارت دیگر، هر چه تعداد پردازنده‌هایی که استفاده می‌کنیم از موازات بیشتر باشد، تسریع از خطی کامل بودن دورتر خواهد شد.

به عنوان یک مثال محاسبه‌ی P-FIB(4) را در شکل ۲۷-۲ در نظر بگیرید، و فرض کنید هر طناب به یک واحد زمان نیاز دارد. از آن جایی که کار برابر است با $T_1 = ۱۷$ و دهانه برابر است با $T_\infty = ۸$ ، موازات برابر است با $T_1/T_\infty = ۱۷/۸ = ۲٫۱۲۵$. در نتیجه دست‌یابی به کمی بیش از دو برابر تسریع امکان‌پذیر خواهد بود، و مستقل از این که از چند پردازنده استفاده می‌کنیم، تسریع بیش‌تر از آن ممکن نیست. ولی خواهیم دید که P-FIB(n) برای ورودی‌های با اندازه‌ی بزرگ‌تر از موازات ذاتی پیروی می‌کند.

سستی (موازات) ((parallel) slacknes) برای یک محاسبه‌ی چندریسمانی اجرا شده بر روی یک کامپیوتر موازی ایده‌آل با P پردازنده را به صورت نسبت $(T_1/T_\infty)/P = T_1/(PT_\infty)$ تعریف می‌کنیم، که برابر است با نسبت فراتر رفتن موازات محاسبه از تعداد پردازنده‌های ماشین. بنابراین اگر سستی کم‌تر از ۱ باشد، نمی‌توانیم امیدوار باشیم که به تسریع خطی کامل برسیم، چرا که $T_1/(PT_\infty) < 1$ و قانون دهانه ایجاب می‌کند که تسریع روی P پردازنده عبارت $T_1/T_P \leq T_1/T_\infty < P$ را ارضا کند. مسلماً همین‌طور که سستی از ۱ به سمت ۰ کاهش می‌یابد، تسریع محاسبه بیشتر و بیشتر از تسریع خطی کامل دور می‌شود. ولی اگر سستی بیشتر از ۱ باشد، کار روی هر پردازنده فاکتور محدود کننده خواهد بود. همان طور که خواهیم دید، هر چه سستی از ۱ بیشتر باشد، یک برنامه‌ریز خوب می‌تواند به تسریع خطی کامل نزدیک‌تر و نزدیک‌تر شود.

برنامه‌ریزی

کارایی بالا به فاکتورهایی بیش از کمینه کردن کار و دهانه بستگی دارد. طناب‌ها هم باید بر روی پردازنده‌های ماشین موازی به صورت بهینه برنامه‌ریزی شده باشند. مدل برنامه‌نویسی چندریسمانی ما هیچ روشی برای تخصیص طناب‌ها به پردازنده‌ها ارائه نمی‌کند. در عوض، ما به برنامه‌ریز چارچوب هم‌زمانی تکیه می‌کنیم که محاسبات در حال پیش‌روی را به پردازنده‌های مختلف تخصیص دهد. در عمل برنامه‌ریز، طناب‌ها را به ریسمان‌های ایستا نگاشت می‌کند، و سیستم عامل ریسمان‌ها را برای پردازنده‌ها برنامه‌ریزی می‌کند، ولی این مرحله‌ی اضافی نگاشت برای درک ما از برنامه‌ریزی غیر ضروری است. ما به سادگی می‌توانیم فرض کنیم که برنامه‌ریز چارچوب هم‌زمانی، مستقیماً طناب‌ها را به پردازنده‌ها نگاشت می‌کند.

یک برنامه‌ریز چندریسمانی باید محاسبات را در حالی برنامه‌ریزی کند که از قبل هیچ اطلاعی از زمان تکثیر یا پایان طناب‌ها ندارد – برنامه‌ریز باید به صورت *آنلاین* (on-line) کار کند. به علاوه، یک برنامه‌ریز خوب به صورت توزیع شده کار می‌کند، که در آن ریسمان‌های پیاده‌سازی کننده‌ی خود برنامه‌ریز، بر روی پردازنده‌ها کار می‌کنند تا توازن بار محاسبات را بر روی آن‌ها انجام دهند. برنامه‌ریزهای توزیع شده و آنلاین خوب هم اکنون وجود دارند، ولی تحلیل آن‌ها پیچیده است.

در عوض برای ساده نگه داشتن تحلیل، یک برنامه‌ریز آنلاین *مرکزی* (centralized) را بررسی خواهیم کرد، که حالت کلی محاسبات را در هر زمان می‌داند. به طور خاص *برنامه‌ریزهای حریص* (greedy scheduler) را تحلیل خواهیم کرد، که در هر مرحله‌ی زمانی تا حد ممکن طناب‌ها را به پردازنده‌ها تخصیص می‌دهند. اگر حین یک مرحله‌ی زمانی حداقل P طناب آماده‌ی اجرا باشند، می‌گوییم آن مرحله، یک *مرحله‌ی کامل* (complete step) است، و یک برنامه‌ریز حریص، ممکن است هر زیرمجموعه‌ی P تایی از طناب‌های آماده را به پردازنده‌ها اختصاص دهد. در غیر این صورت کم‌تر از P طناب آماده‌ی اجرا هستند، که در این حالت می‌گوییم یک *مرحله‌ی ناقص* (incomplete step) داریم، و برنامه‌ریز تمام طناب‌های آماده را به پردازنده‌ها تخصیص می‌دهد.

طبق قانون کار، برای P در بهترین حالت می‌توانیم امید داشته باشیم به زمان اجرای $T_P = T_1/P$

برسیم، و طبق قانون دهانه در بهترین حالت زمان اجرای $T_P = T_\infty$ را خواهیم داشت. قضیه‌ی زیر نشان می‌دهد که برنامه‌ریزی حریصانه از نظر تئوری کارامد است، چرا که به مجموع این دو کران پایین به عنوان یک کران بالا دست می‌یابد.

قضیه‌ی ۲۷-۱ روی یک کامپیوتر موازی ایده‌آل با P پردازنده، برنامه‌ریز حریصانه یک محاسبه‌ی چندریسمانی با کار T_1 و دهانه T_∞ را در زمان

$$T_P \le T_1/P + T_\infty \qquad (۴-۲۷)$$

انجام می‌دهد.

اثبات ابتدا مراحل کامل را بررسی می‌کنیم. در هر مرحله‌ی کامل، P پردازنده با هم به اندازه‌ی P واحد کار انجام می‌دهند. طبق برهان خلف، فرض کنید تعداد مراحل کامل اکیداً بیشتر از $\lfloor T_1/P \rfloor$ است. آن گاه کل کار انجام شده در مراحل کامل حداقل برابر است با

$$\begin{aligned} P \cdot (\lfloor T_1/P \rfloor + 1) &= P\lfloor T_1/P \rfloor + P \\ &= T_1 - (T_1 \bmod P) + P && \text{(طبق تساوی (۳-۸))} \\ &> T_1 && \text{(طبق نامساوی (۳-۹))} \end{aligned}$$

پس به این تناقض می‌رسیم که P پردازنده بیش از مقدار مورد نیاز کار انجام می‌دهند، که به ما اجازه می‌دهد فرض کنیم که تعداد مراحل کامل حداکثر برابر است با $\lfloor T_1/P \rfloor$.

اکنون یک مرحله‌ی ناقص را در نظر بگیرید. فرض کنید G نشان‌دهنده‌ی دگ متناظر با کل محاسبه باشد، و بدون از دست دادن کلیت، فرض کنید هر طناب به یک واحد زمان نیاز دارد. (می‌توانیم هر یک از طناب‌های طولانی‌تر را با زنجیره‌ای از طناب‌های تک-واحدی جایگزین کنیم.) فرض کنید G' زیرگرافی از G باشد که در ابتدای مرحله‌ی ناقص هنوز باید انجام شود، و G'' زیرگراف باقی‌مانده بعد از اجرای مرحله‌ی ناقص. یک طولانی‌ترین مسیر در دگ لزوماً باید از رأسی با درجه‌ی ورودی ۰ آغاز شود. چون یک مرحله‌ی ناقص در برنامه‌ریز حریص، تمام طناب‌های با درجه‌ی ورودی ۰ در G' را اجرا می‌کند، طول بلندترین مسیر در G'' باید یکی کم‌تر از طول بلندترین مسیر در G' باشد. به عبارت دیگر، یک مرحله‌ی ناقص دهانه‌ی دگ اجرا نشده را یکی کاهش می‌دهد. بنابراین تعداد مراحل ناقص حداکثر برابر است با T_∞.

از آن جایی که هر مرحله یا کامل است یا ناقص، قضیه اثبات می‌شود. ▒

نتیجه‌ی زیر از قضیه‌ی ۲۷-۱ نشان می‌دهد که برنامه‌ریزهای حریص همیشه به خوبی عمل می‌کنند.

نتیجه‌ی ۲۷-۲ زمان اجرای هر محاسبه‌ی چندریسمانی دلخواه که توسط یک برنامه‌ریز حریص بر روی یک کامپیوتر موازی ایده‌آل با P پردازنده برنامه‌ریزی شده است، در ضریب ۲ از زمان اجرای بهینه قرار دارد.

اثبات فرض کنید T_P زمان اجرای حاصل از برنامه‌ریز حریص، و T_P^* زمان اجرای حاصل از برنامه‌ریز بهینه بر روی یک ماشین با P پردازنده باشد، و فرض کنید T_1 و T_∞، به ترتیب کار و دهانه‌ی محاسبه باشند. قوانین کار و دهانه – نامساوی‌های (۲۷-۲) و (۲۷-۳) – به دست می‌دهند $T_P^* \geq \max(T_1/P, T_\infty)$، و از این رو قضیه‌ی ۲۷-۱ ایجاب می‌کند که

$$\begin{aligned} T_P &\leq T_1/P + T_\infty \\ &\leq 2 \cdot \max(T_1/P, T_\infty) \\ &\leq 2T_P^* \end{aligned}$$

نتیجه‌ی بعد نشان می‌دهد که در واقع، با افزایش سستی، یک برنامه‌ریز حریص برای هر محاسبه‌ی چندریسمانی به تسریعی نزدیک به تسریع خطی کامل می‌رسد.

نتیجه‌ی ۲۷-۳ فرض کنید T_P زمان اجرای محاسبه‌ی چندریسمانی حاصل از یک برنامه‌ریز حریص روی یک کامپیوتر موازی ایده‌آل با P پردازنده باشد، و فرض کنید T_1 و T_∞ به ترتیب کار و دهانه‌ی محاسبه باشند. آن گاه اگر $P << T_1/T_\infty$، خواهیم داشت $T_P \approx T_1/P$، یا به طور معادل تسریع تقریباً برابر خواهد بود با P.

اثبات اگر فرض کنیم $P << T_1/T_\infty$، آن گاه همچنین خواهیم داشت $T_\infty << T_1/P$، و بنابراین قضیه‌ی ۲۷-۱ به دست می‌دهد $T_P \leq T_1/P + T_\infty \approx T_1/P$. چون قانون کار ایجاب می‌کند که $T_P \geq T_1/P$، نتیجه می‌گیریم که $T_P \approx T_1/P$، یعنی تسریع برابر است با $T_1/T_P \approx P$.

نماد >> نشان‌دهنده‌ی «بسیار کوچک‌تر از» می‌باشد، ولی «بسیار کوچک‌تر» چقدر است؟ به عنوان یک قانون سرراست، سستی حداقل ۱۰- یعنی وقتی موازات ۱۰ برابر بیشتر از پردازنده‌ها باشد – معمولاً برای دست‌یابی به تسریع خوب، کافی است. آن گاه دهانه در کران حریصانه (نامساوی ۲۷-۴) کم‌تر از ۱۰٪ کار به ازای هر پردازنده است، که برای اکثر موقعیت‌های مهندسی به اندازه‌ی کافی خوب است. برای مثال، اگر یک محاسبه بر روی فقط ۱۰ یا ۱۰۰ پردازنده انجام شود، عاقلانه نیست که موازات مثلاً ۱,۰۰۰,۰۰۰ را بهتر از موازات ۱۰,۰۰۰ در نظر بگیریم، با این که به اندازه‌ی یک ضریب ۱۰۰ بین این دو تفاوت وجود دارد. همان طور که مسئله‌ی ۲۷-۲ نشان می‌دهد، بعضی مواقع با کاهش موازات بیش از حد زیاد می‌توانیم به الگوریتم‌هایی برسیم که از جهات دیگر بهتر هستند، و هم‌چنان برای تعداد معقولی از پردازنده‌ها تسریع مناسبی دارند.

تحلیل الگوریتم‌های چندریسمانی

اکنون تمام ابزار مورد نیاز برای تحلیل الگوریتم‌های چندریسمانی و ارائه‌ی کران‌های مناسب بر روی زمان اجرای آن‌ها روی چندین پردازنده را در اختیار داریم. تحلیل کار نسبتاً سرراست است، چرا که

تفاوت چندانی با تحلیل زمان اجرای یک الگوریتم سریال – مثلاً سریال شده‌ی همان الگوریتم چندریسمانی – ندارد، که اکنون دیگر باید با آن کاملاً آشنا باشید (چرا که تا این جا اکثر این کتاب در این مورد بوده است!). تحلیل دهانه جذاب‌تر است، ولی وقتی با آن آشنا شدید خواهید دید که چندان مشکل‌تر از تحیل کار نیست. ایده‌های پایه‌ی این کار را با استفاده از برنامه‌ی P-FIB بررسی می‌کنیم.

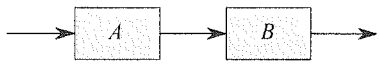
برای تحلیل کار $T_1(n)$ مربوط به P-FIB(n) نیاز به انجام هیچ کاری نداریم، چرا که قبلاً آن را انجام داده‌ایم. برنامه‌ی FIB اصلی در واقع سریال‌سازی شده‌ی P-FIB است، و بنابراین $T_1(n) = T(n) = \theta(\phi^n)$ طبق تساوی (۲۷-۱).

شکل ۲۷-۳ نشان می‌دهد که چطور می‌توان دهانه را تحلیل کرد. اگر دو زیرمحاسبه به صورت سری به هم متصل شوند، دهانه‌ی آن‌ها با هم جمع شده و دهانه‌ی ترکیب آن‌ها را تشکیل می‌دهد، در حالی که اگر به صورت موازی به هم متصل شوند، دهانه‌ی ترکیب برابر است با بیشینه‌ی دهانه‌ی دو زیرمحاسبه. برای P-FIB(n)، فراخوانی تکثیر شده‌ی P-FIB(n-1) در خط ۳ به صورت موازی با فراخوانی P-FIB(n-2) در خط ۴ اجرا می‌شود. بنابراین می‌توانیم دهانه‌ی P-FIB(n) را به صورت رابطه‌ی بازگشتی

$$\begin{aligned} T_\infty(n) &= \max(T_\infty(n-1), T_\infty(n-2)) + \theta(1) \\ &= T_\infty(n-1) + \theta(1) \end{aligned}$$

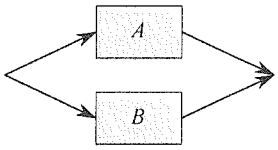
توصیف کنیم، که جواب آن برابر است با $T_\infty(n) = \theta(n)$.

میزان موازات P-FIB(n) برابر است با $T_1(n)/T_\infty(n) = \theta(\phi^n/n)$، که با بزرگ شدن n به طرز قابل توجهی رشد می‌کند. بنابراین حتی بر روی بزرگ‌ترین کامپیوترهای موازی، یک مقدار متوسط n هم برای رسیدن به تسریع خطی کامل برای P-FIB(n) کافی است، چرا که این رویه سستی موازات بالایی دارد.



Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
Span: $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$

(الف)



Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
Span: $T_\infty(A \cup B) = \max(T_\infty(A), T_\infty(B))$

(ب)

شکل ۲۷-۳ کار و دهانه‌ی زیرمحاسبه‌های ترکیب شده. (الف) وقتی دو زیرمحاسبه به صورت سری با هم ترکیب می‌شوند، کار کل محاسبات برابر است با مجموع کار دو زیرمحاسبه، و دهانه‌ی کل برابر است با مجموع دهانه‌ی دو زیرمحسابه. (ب) وقتی دو زیرمحاسبه به صورت موازی به هم متصل می‌شوند، کار کل همچنان برابر است با مجموع کار دو زیرمحاسبه، ولی دهانه‌ی کل برابر است با بیشینه‌ی دهانه‌ی دوزیرمحاسبه.

حلقه‌های موازی

بسیاری از الگوریتم‌ها حلقه‌هایی دارند که تمام تکرارهای آن‌ها می‌توانند به صورت موازی اجرا شوند. همان طور که خواهیم دید، می‌توانیم چنین حلقه‌هایی را با استفاده از کلمه‌های کلیدی **spawn** و **sync** موازی کنیم، ولی بسیار مناسب‌تر خواهد بود اگر تکرارهای چنین حلقه‌هایی بتوانند مستقیماً به صورت هم‌زمان اجرا شوند. شبه‌کدهای ما با استفاده از کلمه‌ی کلیدی **parallel** به این کارایی دست می‌یابند، که در یک حلقه‌ی **for** ، قبل از کلمه‌ی **for** قرار می‌گیرد.

به عنوان یک مثال، برنامه‌ی ضرب یک ماتریس $A = (a_{ij})$ با ابعاد $n \times n$ در یک بردار n تایی $x = (x_j)$ را در نظر بگیرید. بردار n تایی حاصل $y = (y_i)$ با استفاده از تساوی

$$y_i = \sum_{j=1}^{n} a_{ij}$$

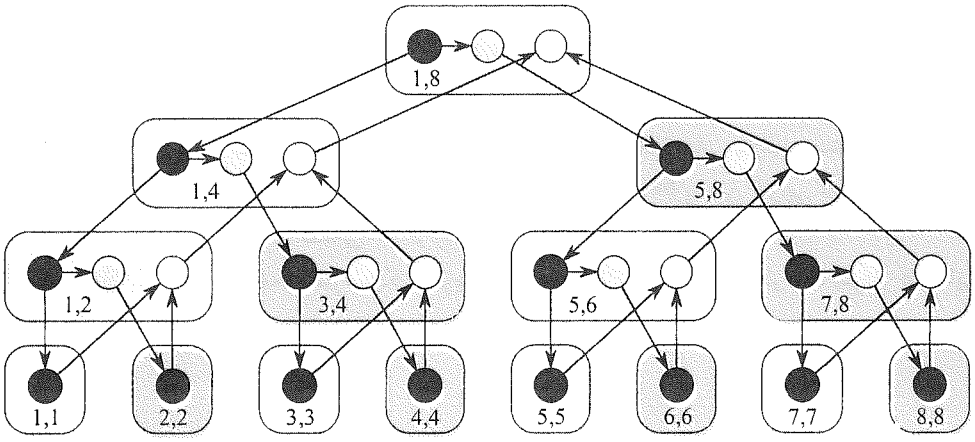
برای $i = 1, 2, \ldots, n$ به دست می‌آید. می‌توانیم به صورت زیر و با محاسبه‌ی تمام ورودی‌های y به صورت موازی، این ضرب ماتریسی را انجام دهیم:

```
MAT-VEC(A, x)
1  n = A.rows
2  let y be a new vector of length n
3  parallel for i = 1 to n
4      y_i = 0
5  parallel for i = 1 to n
6      for j = 1 to n
7          y_i = y_i + a_ij x_j
8  return y
```

در این کد، کلمه‌های کلیدی **parallel for** در خطوط ۳ و ۵ نشان می‌دهند که تکرارهای حلقه‌های متناظر می‌توانند به صورت هم‌زمان اجرا شوند. یک کامپایلر می‌تواند با استفاده از موازی‌سازی تودرتو، هر یک از حلقه‌های **parallel for** را به صورت یک زیرروال تقسیم و حل پیاده‌سازی کند. برای مثال حلقه‌ی **parallel for** در خطوط ۵–۷ می‌تواند با فراخوانی MAT-VEC-MAIN-LOOP$(A, x, y, n, 1, n)$ پیاده‌سازی شود، که در آن کامپایلر، زیرروال کمکی MAT-VEC-MAIN-LOOP را به صورت زیر می‌سازد:

```
MAT-VEC-MAIN-LOOP(A, x, y, n, i, i')
1  if i == i'
2      for j = 1 to n
3          yi = yi + aijxj
4  else mid = ⌊(i + i')/2⌋
5      spawn MAT-VEC-MAIN-LOOP(A, x, y, n, i, mid)
6      MAT-VEC-MAIN-LOOP(A, x, y, n, mid+1, i')
7      sync
```

این کد به صورت بازگشتی، نیمه‌ی اول تکرارهای حلقه را تکثیر می‌کند تا به صورت موازی با نیمه‌ی دوم تکرارها اجرا شوند، و سپس یک **sync** اجرا می‌کند. در نتیجه یک درخت دودویی از اجراها ساخته می‌شوند که در آن برگ‌ها، تکرارهای حلقه هستند، همان طور که در شکل ۲۷–۴ نشان داده شده است.



شکل ۲۷-۴ یک دَگ نشان‌دهنده‌ی محاسبه‌ی MAT-VEC-MAIN-LOOP$(A, x, y, 8, 1, 8)$. دو عدد دورن هر شبه مستطیل، مقدار دو پارامتر آخر (i و i' در سرآیند رویه) را در فراخوانی (یا تکثیر) رویه به دست می‌دهند. دایره‌های تیره نشان‌دهنده‌ی طناب‌هایی هستند که یا با حالت پایه متناظرند یا با ابتدای رویه تا تکثیر MAT-VEC-MAIN-LOOP در خط ۵؛ دایره‌های سایه‌دار نشان دهنده‌ی طناب‌هایی هستند که متناظرند با فراخوانی MAT-VEC-MAIN-LOOP در خط ۶ تا sync در خط ۷، که در آن جا رویه منتظر می‌ماند تا زیرروال تکثیر شده در خط ۵ بازگشت کند؛ و دایره‌های سفید نشان‌دهنده‌ی طناب‌هایی هستند که متناظرند با بخش (قابل چشم‌پوشی) بعد از sync تا جایی که رویه پایان می‌یابد.

برای محاسبه‌ی کار $T_1(n)$ مربوط به MAT-VEC بر روی یک ماتریس $n \times n$ ، به سادگی زمان اجرای نسخه‌ی سریال آن را محاسبه می‌کنیم، که آن را با جایگذاری حلقه‌های for معمولی به جای حلقه‌های parallel for به دست می‌آوریم. بنابراین داریم $T_1(n) = \theta(n^2)$ ، به دلیل زمان اجرای درجه‌ی دوی حلقه‌های دوگانه‌ی تودرتو در خطوط ۵-۷. ولی به نظر می‌آید این تحلیل سربار تکثیر بازگشتی در پیاده‌سازی حلقه‌های موازی را نادیده گرفته است. در واقع سربار تکثیر، کار حلقه‌ی موازی را نسبت به نسخه‌ی سریال افزایش می‌دهد، ولی نه به صورت حدی. برای این که ببینیم چرا، مشاهده کنید که از آن جایی که درخت نمونه‌های رویه‌های بازگشتی، یک درخت دودویی کامل است، تعداد گره‌های داخلی آن یکی کم‌تر است از تعداد برگ‌ها (تمرین ب-۵-۳ را ببینید). هر گره‌ی داخلی به اندازه‌ی ثابتی کار انجام می‌دهد تا دامنه‌ی تکرار را تقسیم کند، و هر برگ متناظر است با یک تکرار از حلقه، که حداقل به زمان ثابت (در این جا $\theta(n)$) نیاز دارد. بنابراین می‌توانیم سربار تکثیرهای بازگشتی را نسبت به کار انجام شده در تکرارها سرشکن کنیم، و نتیجه بگیریم که حداکثر به اندازه‌ی یک ضریب ثابت به کل کار اضافه می‌کنند.

به عنوان یک مسئله‌ی عملی، چارچوب‌های هم‌زمانی چندریسمانی‌سازی پویا، بعضی مواقع برگ‌های درخت بازگشت را تجمیع (coarsen) کرده و چندین تکرار حلقه را در یک برگ انجام

می‌دهند، یا به صورت خودکار و یا تحت کنترل برنامه‌نویس، و بدین صورت سربار تکثیر بازگشتی را کاهش می‌دهند. هرچند این کاهش سربار با هزینه‌ی کاهش موازات انجام می‌شود، ولی اگر محاسبات به اندازه‌ی کافی سستی موازات داشته باشد، تسریع تقریباً خطی کامل فدای این کار نخواهد شد.

هنگام تحلیل دهانه‌ی یک حلقه‌ی موازی هم باید سربار تکثیر بازگشتی را در نظر بگیریم. از آن جایی که عمق بازگشت نسبت به تعداد تکرارها لگاریتمی است، برای یک حلقه‌ی موازی با n تکرار، که در آن دهانه‌ی تکرار i ام $iter_\infty(i)$ است، دهانه برابر است با

$$T_\infty(n) = \theta(\lg n) + \max_{1 \le i \le n} iter_\infty(i)$$

مثلاً برای MAT-VEC روی یک ماتریس $n \times n$، حلقه‌ی موازی مقداردهی اولیه در خطوط ۳-۴ دارای دهانه‌ی $\theta(\lg n)$ است، چرا که تکثیر بازگشتی بر کار ثابت در هر تکرار غلبه می‌کند. دهانه‌ی حلقه‌های تودرتوی دوگانه‌ی خطوط ۵-۷ برابر است با $\theta(n)$، چرا که هر تکرار از حلقه‌ی **parallel for** خارجی حاوی n تکرار از حلقه‌ی **for** (سریال) داخلی است. دهانه‌ی بقیه‌ی رویه ثابت است، و بنابراین دهانه‌ی حلقه‌های تودرتو دهانه‌ی غالب است، و دهانه‌ی کل برابر است با $\theta(n)$ برای تمام رویه. چون کار برابر است با $\theta(n^2)$، موازات برابر است با $\theta(n^2)/\theta(n) = \theta(n)$. (تمرین ۲۷-۱-۶ از شما می‌خواهد یک پیاده‌سازی با موازات بیشتر ارائه کنید.)

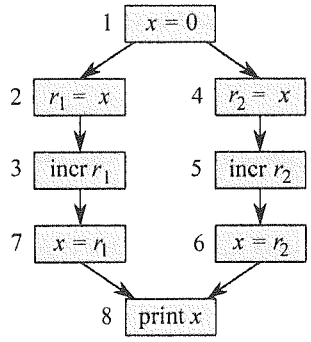
موقعیت‌های چالش‌آمیز

یک الگوریتم چندریسمانی، *قطعی* (deterministic) است اگر همیشه برای یک ورودی خاص، خروجی یکسان تولید کند، مستقل از این که دستورالعمل‌ها چگونه روی پردازنده‌ها برنامه‌ریزی می‌شوند، و *ناقطعی* (nondeterministic) است اگر رفتار آن در اجراهای مختلف با هم تفاوت داشته باشد. خیلی مواقع یک الگوریتم چندریسمانی که باید قطعی باشد نمی‌تواند به این هدف دست یابد، چرا که حاوی یک «چالش قطعیت» است.

موقعیت‌های چالش‌آمیز، مخرب هم‌زمان‌سازی هستند. چند خطاهای معروف ناشی از این موقعیت‌ها عبارتند از دستگاه درمان تابشی Therac-25، که باعث مرگ سه نفر و مجروح شدن چندین نفر دیگر شد، و قطع برق امریکای شمالی در سال ۲۰۰۳، که بیش از ۵۰ میلیون نفر از آن رنج بردند. یافتن این اشکالات مهلک بسیار دشوار است. ممکن است روزهای متوالی در آزمایشگاه برنامه‌ی خود را بدون هیچ مشکلی تست کنید، ولی در نهایت هنگام عمل به خطاهای عجیب و نادر بر بخورید.

یک *چالش قطعیت* (determinacy race) زمانی رخ می‌دهد که دو دستورالعمل منقطا موازی به یک خانه از حافظه دسترسی داشته باشند، و حداقل یکی از آن‌ها در آن خانه‌ی حافظه بازنویسی انجام دهد. رویه‌ی زیر یک حالت چالش‌آمیز را نشان می‌دهد:

```
RACE-EXAMPLE( )
1  x = 0
2  parallel for i = 1 to 2
3      x = x + 1
4  print x
```



(الف)

step	x	r_1	r_2
1	0	–	–
2	0	0	–
3	0	1	–
4	0	1	0
5	0	1	1
6	1	1	1
7	1	1	1

(ب)

شکل ۲۷-۵ نمایش چالش قطعیت در RACE-EXAMPLE. (الف) یک دَگ محاسبات که وابستگی‌های بین دستورالعمل‌های مختلف را نشان می‌دهد. ثبات‌های پردازنده‌ها عبارتند از r_1 و r_2. از دستورالعمل‌های نامربوط به چالش، مانند پیاده‌سازی کنترل‌های حلقه، صرف نظر شده است. (ب) یک دنباله‌ی اجرا که خطای احتمالی را بروز می‌دهد، که در آن مقادیر x در حافظه و r_1 و r_2 در دنباله‌ی اجرا نشان داده شده است.

پس از مقداردهی اولیه‌ی x با ۰ در خط ۱، رویه‌ی RACE-EXAMPLE دو طناب موازی می‌سازد، که هر دوی آن‌ها x را در خط ۳ افزایش می‌دهند. با این که به نظر می‌رسد RACE-EXAMPLE همیشه باید مقدار ۲ را چاپ کند (نسخه‌ی سریال آن قطعا همین کار را می‌کند)، ممکن است مقدار ۱ را چاپ کند. اجازه دهید ببینیم این ناهنجاری چگونه ممکن است رخ دهد.

وقتی یک پردازنده مقدار x را افزایش می‌دهد، این عملیات غیرقابل تقسیم است، ولی خود از چند دستورالعمل تشکیل شده است:

۱. خواندن x از حافظه ونوشتن آن روی یکی از ثبات‌های (register) پردازنده.
۲. افزایش مقدار ثبات.
۳. نوشتن مقدار ثبات در خانه‌ی حافظه‌ی مربوط به x.

در شکل ۲۷-۵(الف) یک دَگ محاسبات نشان‌دهنده‌ی اجرای RACE-EXAMPLE را می‌بینیم که در آن طناب‌ها به دستورالعمل‌های تکی شکسته شده‌اند. به خاطر بیاورید که چون یک کامپیوتر موازی ایده‌آل، سازگاری ترتیبی دارد، می‌توانیم اجرای موازی الگوریتم چندریسمانی را به صورت یک جاسازی دستورالعمل‌ها ببینیم که به وابستگی‌های دَگ احترام می‌گذارد. بخش (ب) شکل، مقادیر را برای اجرایی نشان می‌دهد که در آن ناهنجاری بروز کرده است. مقدار x در حافظه ذخیره شده است، و r_1 و r_2 ثبات‌های پردازنده هستند. در مرحله‌ی ۱، یکی از پردازنده‌ها x را برابر ۰ قرار می‌دهد. در مراحل ۲ و ۳، پردازنده‌ی ۱ مقدار x را از حافظه خوانده و در ثبات r_1 قرار می‌دهد. در این مرحله پردازنده‌ی ۲ وارد عمل می‌شود و دستورالعمل‌های ۴-۶ را انجام می‌دهد. پردازنده‌ی ۲ مقدار x را از حافظه خوانده و در ثبات r_2 می‌ریزد، آن را افزایش می‌دهد، که مقدار ۱ را برای r_1 تولید می‌کند، و

سپس مقدار حاصل را در x می‌ریزد. اکنون پردازنده‌ی ۱ با مرحله‌ی ۷ بازمی‌گردد، که ذخیره‌ی مقدار ۱ مربوط به x_1 در x است، که در واقع x را تغییری نمی‌دهد. بنابراین در مرحله‌ی ۸ مقدار ۱ در خروجی چاپ می‌شود، برعکس حالت سریال که در آن مقدار ۲ را در خروجی خواهیم داشت.

می‌توانیم ببینیم که چه اتفاقی افتاده است. اگر تأثیر اجرای موازی این بود که پردازنده‌ی ۱ تمام دستورالعمل‌های خود را قبل از پردازنده‌ی ۲ اجرا کند، مقدار ۲ در خروجی چاپ می‌شد. برعکس، اگر پردازنده‌ی ۲ قبل از پردازنده‌ی ۱ تمام دستورالعمل‌های خود را انجام می‌داد، باز هم مقدار ۲ را در خروجی داشتیم. ولی وقتی دستورالعمل‌های دو پردازنده با هم اجرا می‌شوند، ممکن است یکی از به‌هنگام‌سازی‌های x از دست برود (مانند حالتی که در بالا گفته شد).

مسلماً بسیاری از اجراها باعث رخ‌داد این خطا نمی‌شوند. مثلاً اگر ترتیب اجرا به صورت $\langle 1, 2, 3, 4, 5, 6, 7, 8 \rangle$ یا به صورت $\langle 1, 4, 5, 6, 2, 3, 7, 8 \rangle$ بود، به نتیجه‌ی صحیح دست پیدا می‌کردیم. این مشکل چالش‌های قطعیت است. معمولاً اکثر ترتیب‌ها نتیجه‌ی درست را تولید می‌کنند – مانند تمام ترتیب‌هایی که در آن‌ها دستورالعمل‌های سمت چپ قبل از دستورالعمل‌های سمت راست اجرا می‌شوند، و یا برعکس. ولی وقتی دستورالعمل‌ها درون یکدیگر جاسازی می‌شوند، بعضی ترتیب‌ها به نتایج نادرست ختم می‌شوند. در نتیجه انجام تست برای تشخیص این چالش‌ها بسیار دشوار است. ممکن است روزها برنامه‌ی خود را تست کرده و هیچ خطایی نیابید، ولی در عمل، زمانی که خروجی برنامه حیاتی است، با توقف ناگهانی سیستم روبه‌رو شوید.

با این که می‌توانیم از طرق مختلف، مثلاً از طریق قفل‌های ممانعت دوجانبه (mutual exclusion) و یا متدهای دیگر هماهنگ‌سازی، از عهده‌ی این چالش‌ها برآییم، ولی در این جا به سادگی اطمینان حاصل می‌کنیم که طناب‌هایی که به صورت موازی کار می‌کنند *مستقل* (independent) از یکدیگرند: یعنی هیچ چالش قطعیتی نسبت به یکدیگر ندارند. بنابراین در یک ساختار **parallel for** تمام تکرارها باید مستقل از یکدیگر باشند. بین یک **spawn** و **sync** متناظر، کد فرزند تکثیر شده باید مستقل از کد پدر باشد، که این شامل کد فرزندان تکثیر یا فراخوانی‌شده‌ی اضافی هم می‌شود. توجه کنید که آرگومان‌های یک فرزند تکثیر شده، قبل از این که واقعاً تکثیر رخ دهد، در پدر ارزیابی (evaluate) می‌شوند، و بنابراین ارزیابی آرگومان‌های یک زیرروال تکثیر شده به صورت سری با دسترسی به همان آرگومان‌ها بعد از تکثیر قرار دارد.

به عنوان یک مثال از میزان سادگی ساختن کد حاوی چالش، در زیر یک پیاده‌سازی ناصحیح از ضرب ماتریس–بردار چندریسمانی را می‌بینیم که با موازی‌سازی حلقه‌ی **for** داخلی به دهانه‌ی $\theta(\lg n)$ دست می‌یابد:

```
MAT-VEC-WRONG(A, x)
1  n = A.rows
2  let y be a new vector of length n
3  parallel for i = 1 to n
4      y_i = 0
5  parallel for i = 1 to n
```

```
6       parallel for j = 1 to n
7           y_i = y_i + a_ij x_j
8   return y
```

متأسفانه این رویه، که برای تمام n مقدار مختلف j به صورت هم‌زمان اجرا می‌شود، به خاطر وجود چالش برای به هنگام‌سازی مقدار y_i در خط ۷، نادرست است. تمرین ۲۷-۱-۶ از شما می‌خواهد یک پیاده‌سازی صحیح با دهانه‌ی $\theta(\lg n)$ ارائه کنید.

یک الگوریتم چندریسمانی حاوی چالش می‌تواند بعضی مواقع درست باشد. به عنوان یک مثال، دو ریسمان موازی ممکن است مقدار یکسانی را در یک متغیر مشترک ذخیره کنند، و اهمیتی نداشته باشد که کدام یک زودتر این مقدار را ذخیره کرده است. ولی به طور کلی کدهای حاوی چالش را نادرست در نظر می‌گیریم.

یک درس از شطرنج

این بخش را با یک داستان واقعی پایان می‌دهیم که حین طراحی برنامه‌ی چندریسمانی جهانی بازی شطرنج با نام Socrates رخ داد (برای وضوح، مقادیر زمانی را ساده کرده‌ایم). طراحی اولیه‌ی این برنامه برای یک کامپیوتر با ۳۲ پردازنده انجام شده بود، ولی در نهایت باید روی یک کامپیوتر با ۵۱۲ پردازنده اجرا می‌شد. در مرحله‌ای از طراحی، برنامه‌نویسان یک بهینه‌سازی بر روی برنامه انجام دادند که زمان اجرای آن را نسبت به یک معیار مهم روی ماشین ۳۲-پردازنده‌ای از $T_{۳۲} = ۶۵$ ثانیه به $T'_{۳۲} = ۴۰$ ثانیه کاهش داد. ولی برنامه‌نویسان با استفاده از معیارهای کار و دهانه نتیجه گرفتند که نسخه‌ی بهینه‌سازی شده، که روی ۳۲ پردازنده سریع‌تر بود، روی ماشین اصلی با ۵۱۲ پردازنده کندتر خواهد بود. در نتیجه از آن «بهینه‌سازی» صرف نظر کردند.

تحلیل آن‌ها به این صورت بود. در نسخه‌ی اصلی برنامه کار برابر با $T_۱ = ۲۰۴۸$ ثانیه و دهانه برابر با $T_\infty = ۱$ ثانیه بود. اگر با نامساوی (۲۷-۴) به صورت یک تساوی برخورد کنیم، داریم $T_P = T_۱/P + T_\infty$، و با استفاده از آن به عنوان یک تقریب برای زمان اجرا بر روی P پردازنده، می‌بینیم که در واقع $T_{۳۲} = ۲۰۴۸/۳۲ + ۱ = ۶۵$. با استفاده از بهینه‌سازی، میزان کار به $T'_۱ = ۱۰۲۴$ ثانیه کاهش و دهانه به $T'_\infty = ۸$ ثانیه افزایش یافت. دوباره با استفاده از همان تقریب، داریم $T'_{۳۲} = ۱۰۲۴/۳۲ + ۸ = ۴۰$. بهینه‌سازی که برنامه را روی ۳۲ پردازنده تسریع کرد، بر روی ۵۱۲ پردازنده آن را دو برابر کندتر می‌کرد! دهانه‌ی ۸ مربوط به نسخه‌ی بهینه‌سازی شده، که بر روی ۳۲ پردازنده عبارت غالب نبود، بر روی ۵۱۲ پردازنده تبدیل به عبارت غالب می‌شد، که مزیت استفاده از پردازنده‌های بیشتر را از بین می‌برد.

نتیجه‌ی اخلاقی این داستان این است که کار و دهانه نسبت به زمان اجرای اندازه‌گیری شده، معیارهای بهتری برای تعیین کارایی هستند.

تمرین‌ها

۲۷-۱-۱ فرض کنید در خط ۴ کد P-FIB، به جای فراخوانی P-FIB(n-2) آن را تکثیر می‌کردیم. تأثیر این تغییر بر روی کار، دهانه، و موازات حدی چگونه است؟

۲۷-۱-۲ دَگ محاسبات حاصل از اجرای P-FIB(5) را بکشید. با فرض این که هر طناب در محاسبات به زمان واحد نیاز دارد، کار، دهانه، و موازات محاسبه چقدر است؟ با برچسب‌گذاری هر طناب با مرحله‌ی زمانی که در آن اجرا می‌شود، نشان دهید که چطور می‌توان با استفاده از برنامه‌ریزی حریصانه، این دَگ را روی سه پردازنده برنامه‌ریزی کرد.

۲۷-۱-۳ اثبات کنید که یک برنامه‌ریز حریص به کران زیر دست می‌یابد، که کمی از کران اثبات شده در قضیه‌ی ۲۷-۱ قوی‌تر است:

$$T_P \le \frac{T_1 - T_\infty}{P} + T_\infty \qquad (۲۷-۵)$$

۲۷-۱-۴ یک دَگ محاسبات بسازید که برای آن، یک اجرا از یک برنامه‌ریز حریص می‌تواند تا دوبرابر اجرای دیگر یک برنامه‌ریز حریص بر روی همان تعداد پردازنده زمان ببرد.

۲۷-۱-۵ پروفرسور Karan الگوریتم چندریسمانی قعطی خود را روی ۴ ، ۱۰ ، و ۶۴ پردازنده‌ی یک کامپیوتر موازی ایده‌آل با استفاده از یک برنامه‌ریز حریص آزمایش می‌کند. او ادعا می‌کند که این سه اجرا به نتایج $T_4 = 80$ ثانیه، $T_{10} = 42$ ثانیه، و $T_{64} = 10$ ثانیه ختم شدند. بحث کنید که پروفسور یا دروغ می‌گوید و یا اشتباه می‌کند. (راهنمایی: از قانون کار (۲۷-۲)، قانون دهانه (۲۷-۳)، و نامساوی (۲۷-۵) از تمرین ۲۷-۱-۳ استفاده کنید.)

۲۷-۱-۶ یک الگوریتم چندریسمانی برای ضرب یک ماتریس $n \times n$ در یک بردار nتایی ارائه کنید که به موازات $\theta(n^2/\lg n)$ دست می‌یابد، در حالی که هم‌چنان کار $\theta(n^2)$ را حفظ می‌کند.

۲۷-۱-۷ می‌توانیم شبه‌کد چندریسمانی زیر را برای تبدیل یک ماتریس A با اندازه‌ی $n \times n$ به ماتریس ترانهاده به صورت درجا در نظر بگیرید:

```
P-TRANSPOSE(A)
1  n = A.rows
2  parallel for j = 2 to n
3      parallel for i = 1 to j-1
4          exchange a_ij with a_ji
```

کار، دهانه، و موازات این الگوریتم را تحلیل کنید.

۲۷-۱-۸ فرض کنید حلقه‌ی parallel for خط ۳ رویه‌ی P-TRANSPOSE (تمرین ۲۷-۱-۷) را با یک for معمولی جایگزین می‌کنیم. کار، دهانه، و موازات الگوریتم حاصل را تحلیل کنید.

۲۷-۱-۹ با فرض این که $T_P = T_1/P + T_\infty$، برای چند پردازنده دو برنامه‌ی شطرنج با سرعت یکسان اجرا می‌شوند؟

۲-۲۷ ضرب چندریسمانی ماتریس‌ها

در این بخش بررسی می‌کنیم که چطور می‌توان ضرب ماتریس‌ها را چندریسمانی کرد (مسئله‌ای که زمان اجرای سریال آن را در بخش ۴-۲ دیدیم). الگوریتم‌هایی خواهیم دید بر پایه‌ی حلقه‌های تودرتوی سه‌گانه و یا الگوریتم‌های تقسیم و حل.

ضرب چندریسمانی ماتریس‌ها

اولین الگوریتمی که خواهیم دید، الگوریتم سرراستی است که از موازی‌سازی حلقه‌های رویه‌ی SQUARE-MATRIX-MULTIPLY در بخش ۴-۲ به دست می‌آید:

```
P-SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  parallel for i = 1 to n
4      parallel for j = 1 to n
5          cij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

برای تحلیل این الگوریتم، مشاهده کنید که نسخه‌ی سریال الگوریتم به سادگی رویه‌ی SQUARE-MATRIX-MULTIPLY است، بنابراین کار برابر است با $T_1(n) = \theta(n^3)$، مشابه زمان اجرای رویه‌ی SQUARE-MATRIX-MULTIPLY. دهانه برابر است با $T_\infty(n) = \theta(n)$، چرا که مسیر بحرانی عبارت است از یک مسیر به سمت پایین روی درخت بازگشتی برای حلقه‌ی parallel for که در خط ۳ آغاز می‌شود، سپس یک مسیر به سمت پایین روی درخت بازگشتی برای حلقه‌ی parallel for که در خط ۴ آغاز می‌شود، و نهایتاً اجرای تمام n تکرار حلقه‌ی for معمولی که در خط ۶ آغاز می‌شود، که دهانه‌ی کلی $\theta(\lg n) + \theta(\lg n) + \theta(n) = \theta(n)$ را نتیجه می‌دهد. بنابراین موازات الگوریتم برابر است با $\theta(n^3)/\theta(n) = \theta(n^2)$. تمرین ۲۷-۲-۳ از شما می‌خواهد حلقه‌ی داخلی را هم موازی‌سازی کرده تا به موازات $\theta(\lg n)$ دست یابید، که نمی‌توانید با استفاده از یک parallel for سرراست این کار را انجام دهید، چرا که چالش به وجود خواهید آورد.

یک الگوریتم چندریسمانی تقسیم و حل برای ضرب ماتریس‌ها

همان طور که در بخش ۴-۲ آموختیم، می‌توان به صورت سریال و با استفاده از استراتژی تقسیم و حل استراسن، ماتریس‌های $n \times n$ را در زمان $\theta(n^{\lg 7}) = \theta(n^{2.81})$ در هم ضرب کرد، که ما را تحریک می‌کند سعی کنیم آن را چندریسمانی کنیم. مانند بخش ۴-۲، با چندریسمانی کردن یک الگوریتم ساده‌تر آغاز می‌کنیم.

به خاطر بیاورید که رویه‌ی SQUARE-MATRIX-MULTIPLY-RECURSIVE، که دو ماتریس A و B با اندازه‌ی $n \times n$ را در هم ضرب کرده تا یک ماتریس C با اندازه‌ی $n \times n$ به دست آورد، بر

پایه‌ی تقسیم هر یک از سه ماتریس به چهار زیرماتریس $n/۲\times n/۲$ بنا شده است:

$$A=\begin{pmatrix}A_{۱۱} & A_{۱۲}\\ A_{۲۱} & A_{۲۲}\end{pmatrix},\quad B=\begin{pmatrix}B_{۱۱} & B_{۱۲}\\ B_{۲۱} & B_{۲۲}\end{pmatrix},\quad C=\begin{pmatrix}C_{۱۱} & C_{۱۲}\\ C_{۲۱} & C_{۲۲}\end{pmatrix}$$

بعد از تقسیم، می‌توانیم ضرب ماتریس را به صورت زیر بازنویسی کنیم:

$$\begin{aligned}\begin{pmatrix}C_{۱۱} & C_{۱۲}\\ C_{۲۱} & C_{۲۲}\end{pmatrix}&=\begin{pmatrix}A_{۱۱} & A_{۱۲}\\ A_{۲۱} & A_{۲۲}\end{pmatrix}\begin{pmatrix}B_{۱۱} & B_{۱۲}\\ B_{۲۱} & B_{۲۲}\end{pmatrix}\\ &=\begin{pmatrix}A_{۱۱}B_{۱۱} & A_{۱۱}B_{۱۲}\\ A_{۲۱}B_{۱۱} & A_{۲۱}B_{۱۲}\end{pmatrix}+\begin{pmatrix}A_{۱۲}B_{۲۱} & A_{۱۲}B_{۲۲}\\ A_{۲۲}B_{۲۱} & A_{۲۲}B_{۲۲}\end{pmatrix}\end{aligned}\qquad(۶-۲۷)$$

بنابراین برای ضرب دو ماتریس $n\times n$ باید هشت ضرب ماتریسی $n/۲\times n/۲$ و یک جمع ماتریسی $n/۲\times n/۲$ انجام دهیم. شبه‌کد زیر با استفاده از موازی‌سازی تودرتو این استراتژی تقسیم و حل را پیاده‌سازی می‌کند. بر خلاف رویه‌ی SQUARE-MATRIX-MULTIPLY-RECURSIVE، رویه‌ی P-MATRIX-MULTIPLY-RECURSIVE، ماتریس خروجی را به عنوان یک پارامتر ورودی دریافت می‌کند تا از تخصیص غیر ضروری حافظه جلوگیری شود.

```
P-MATRIX-MULTIPLY-RECURSIVE(C, A, B)
1   n = A.rows
2   if n == 1
3       c11 = a11 b11
4   else let T be a new n × n matrix
5       partition A, B, C, and T into n/2 × n/2 submatrices
            A11, A12, A21, A22; B11, B12, B21, B22; C11, C12, C21, C22;
            and T11, T12, T21, T22; respectively
6       spawn P-MATRIX-MULTIPLY-RECURSIVE(C11, A11, B11)
7       spawn P-MATRIX-MULTIPLY-RECURSIVE(C12, A11, B12)
8       spawn P-MATRIX-MULTIPLY-RECURSIVE(C21, A21, B11)
9       spawn P-MATRIX-MULTIPLY-RECURSIVE(C22, A21, B12)
10      spawn P-MATRIX-MULTIPLY-RECURSIVE(T11, A12, B21)
11      spawn P-MATRIX-MULTIPLY-RECURSIVE(T12, A12, B22)
12      spawn P-MATRIX-MULTIPLY-RECURSIVE(T21, A22, B21)
13      P-MATRIX-MULTIPLY-RECURSIVE(T22, A22, B22)
14      sync
15      parallel for i = 1 to n
16          parallel for j = 1 to n
17              cij = cij + tij
```

خط ۳ حالت پایه را اداره می‌کند، که در آن ضرب ماتریس‌های ۱×۱ انجام می‌شود. حالت بازگشتی در خطوط ۴–۱۷ اداره می‌شود. در خط ۴ یک ماتریس موقتی T در حافظه تخصیص داده می‌شود، و خط ۵ هر یک از ماتریس‌های A، B، C، و T را به زیرماتریس‌های $n/۲\times n/۲$ تقسیم می‌کند. (مانند رویه‌ی SQUARE-MATRIX-MULTIPLY-RECURSIVE در بخش ۴–۲، از مسئله‌ی کم‌اهمیت استفاده از محاسبات اندیسی برای نمایش بخش‌های زیرماتریس‌های یک ماتریس صرف نظر می‌کنیم.) فراخوانی

بازگشتی در خط ۶ زیرماتریس C_{11} را با ضرب ماتریسی $A_{11} \times B_{11}$ مقداردهی می‌کند، بنابراین C_{11} برابر است با اولین عبارت از دو عبارتی که در تساوی (۲۷-۶) مجموع آن‌ها مقدار نهایی آن را تشکیل می‌دهد. به طور مشابه، خطوط ۷-۹ زیرماتریس‌های C_{12}، C_{21}، و C_{22} را با اولین عبارت از دو عبارت جمع مربوطه در تساوی (۲۷-۶) مقداردهی می‌کنند. خط ۱۰ زیرماتریس T_{11} را برابر با ضرب زیرماتریسی $A_{12}B_{21}$ قرار می‌دهد تا T_{11} برابر با عبارت دوم در مجموع مربوط به C_{11} باشد. خطوط ۱۱-۱۳ زیرماتریس‌های T_{12} ، T_{21} ، و T_{22} را به ترتیب برابر با عبارت دوم در مجموع مربوط به C_{12}، C_{21}، و C_{22} قرار می‌دهند. هفت فراخوانی بازگشتی اول تکثیر شده‌اند، و آخری روی طناب اصلی اجرا می‌شود. عبارت sync در خط ۱۴ اطمینان حاصل می‌کند که ابتدا ضرب زیرماتریس‌های خطوط ۶-۱۳ محاسبه می‌شوند، و بعد از آن حلقه‌های تودرتوی parallel for در خطوط ۱۵-۱۷، ضرب‌های ذخیره شده در T را با C جمع می‌کنند.

ابتدا کار رویه‌ی P-MATRIX-MULTIPLY-RECURSIVE را تحلیل می‌کنیم، که منعکس کننده‌ی تحلیل زمان اجرای سریال جد خود SQUARE-MATRIX-MULTIPLY-RECURSIVE است. در حالت بازگشتی، تقسیم‌بندی را در زمان $\theta(1)$ انجام می‌دهیم، سپس هشت ضرب ماتریسی $n/2 \times n/2$ داریم، و نهایتاً $\theta(n^2)$ کار برای جمع ماتریس‌های $n \times n$ خواهیم داشت. بنابراین رابطه‌ی بازگشتی برای کار، که آن را $M_1(n)$ می‌نامیم، عبارت است از

$$\begin{aligned} M_1(n) &= 8M_1(n/2) + \theta(n^2) \\ &= \theta(n^3) \end{aligned}$$

طبق حالت ۱ از قضیه‌ی اصلی. به عبارت دیگر کار الگوریتم چندریسمانی ما به صورت حدی برابر است با زمان اجرای رویه‌ی SQUARE-MATRIX-MULTIPLY در بخش ۴-۲، با حلقه‌های تودرتوی سه‌گانه.

برای تعیین دهانه‌ی رویه‌ی P-MATRIX-MULTIPLY-RECURSIVE، ابتدا مشاهده می‌کنیم که دهانه‌ی تقسیم‌بندی $\theta(1)$ است، که مغلوب دهانه‌ی $\theta(\lg n)$ مربوط به حلقه‌های تودرتوی دوگانه‌ی parallel for در خطوط ۱۵-۱۷ می‌شود. چون هشت فراخوانی بازگشتی موازی همگی روی ماتریس‌هایی با اندازه‌ی یکسان اجرا می‌شوند، دهانه‌ی بیشینه میان تمام فراخوانی‌های بازگشتی به سادگی برابر است با دهانه‌ی هر کدام از آن‌ها. بنابراین رابطه‌ی بازگشتی دهانه‌ی رویه‌ی P-MATRIX-MULTIPLY-RECURSIVE، که آن را $M_\infty(n)$ می‌نامیم عبارت است از

$$M_\infty(n) = M_\infty(n/2) + \theta(\lg n) \qquad (۲۷-۷)$$

این رابطه‌ی بازگشتی با هیچ یک از حالت‌های قضیه‌ی اصلی سازگار نیست، ولی شروط تمرین ۴-۶-۲ را دارد. بنابراین طبق تمرین ۴-۶-۲، جواب رابطه‌ی بازگشتی (۲۷-۷) برابر است با $M_\infty(n) = \theta(\lg^2 n)$.

اکنون که کار و دهانه‌ی رویه‌ی P-MATRIX-MULTIPLY-RECURSIVE را می‌دانیم، می‌توانیم موازات آن را به صورت $M_1(n)/M_\infty(n) = \theta(n^3/\lg^2 n)$ محاسبه کنیم، که بسیار بالا است.

چندریسمانی کردن متد استراسن

برای چندریسمانی کردن متد استراسن همان رویکرد کلی بخش ۴-۲ را دنبال می‌کنیم، فقط با این تفاوت که در آن از حلقه‌های موازی استفاده خواهیم کرد:

۱. ماتریس‌های ورودی A و B و ماتریس خروجی C را به زیرماتریس‌های $n/2 \times n/2$ تقسیم می‌کنیم، مانند تساوی (۲۷-۶). این مرحله با استفاده از محاسبات اندیسی به کار و دهانه‌ی $\theta(1)$ نیاز خواهد داشت.

۲. ۱۰ ماتریس S_1، S_2، ...، S_{10} با اندازه‌ی $n/2 \times n/2$ می‌سازیم، که هر کدام آن‌ها جمع یا تفریق دو تا از ماتریس‌های ساخته شده در مرحله‌ی ۱ هستند. می‌توانیم با استفاده از حلقه‌های تودرتوی دوگانه‌ی parallel for تمام ۱۰ ماتریس را با صرف کار $\theta(n^2)$ و دهانه‌ی $\theta(\lg n)$ بسازیم.

۳. با استفاده از زیرماتریس‌های ساخته شده در مرحله‌ی ۱ و همچنین ۱۰ ماتریس ساخته شده در مرحله‌ی ۲، به صورت بازگشتی محاسبه‌ی هفت ضرب ماتریسی $P_1, P_2, ..., P_7$ با اندازه‌ی $n/2 \times n/2$ را تکثیر می‌کنیم.

۴. زیرماتریس‌های مورد نظر C_{11}، C_{12}، C_{21}، و C_{22} مربوط به ماتریس نتیجه‌ی C را با جمع و تفریق ترکیب‌های مختلف ماتریس‌های P_i محاسبه می‌کنیم، باز هم با استفاده از حلقه‌های تودرتوی دوگانه‌ی parallel for . تمام چهار زیرماتریس را می‌توانیم با صرف کار $\theta(n^2)$ و دهانه‌ی $\theta(\lg n)$ محاسبه کنیم.

برای تحلیل این الگوریتم، ابتدا مشاهده می‌کنیم که از آن جایی که سریال‌سازی این الگوریتم متناظر است با الگوریتم سریال اصلی، کار الگوریتم برابر است با زمان اجرای الگوریتم سریال، یا $\theta(n^{\lg 7})$. مانند P-MATRIX-RECURSIVE، می‌توانیم یک رابطه‌ی بازگشتی برای دهانه به دست آوریم. در این الگوریتم هفت فراخوانی بازگشتی به صورت موازی اجرا می‌شوند، ولی از آن جایی که همه‌ی آن‌ها روی ماتریس‌هایی با اندازه‌های مشابه کار می‌کنند، همان رابطه‌ی بازگشتی (۲۷-۷) را به دست می‌آوریم که برای P-MATRIX-RECURSIVE به دست آوردیم، که جواب آن $\theta(\lg^2 n)$ است. بنابراین موازات متد استراسن چندریسمانی برابر است با $\theta(n^{\lg 7}/\lg^2 n)$ ، که بالا است، هر چند مقداری از موازات P-MATRIX-RECURSIVE کم‌تر است.

تمرین‌ها

۲۷-۲-۱ دگ محاسبات مربوط به P-SQUARE-MATRIX-MULTIPLY را بر روی ماتریس‌های ۲×۲ بکشید، و رأس‌های شکل خود را طوری برچسب‌گذاری کنید که مشخص شود مربوط به کدام طناب‌ها در اجرای برنامه هستند. از این روش استفاده کنید که یال‌های تکثیر و فراخوانی به سمت پایین، یال‌های ادامه به صورت افقی به سمت راست، و یال‌های بازگشت به سمت بالا اشاره می‌کنند. با فرض این که هر طناب به زمان واحد نیاز دارد، کار، دهانه، و موازات این محاسبه را تحلیل کنید.

۲-۲-۲۷ تمرین ۲۷-۲-۱ را برای P-MATRIX-MULTIPLY-RECURSIVE تکرار کنید.

۳-۲-۲۷ شبه‌کدی برای الگوریتمی چندریسمانی ارائه کنید که دو ماتریس $n \times n$ را با صرف کار $\theta(n^3)$ و دهانه‌ی $\theta(\lg n)$ در هم ضرب می‌کند. الگوریتم خود را تحلیل کنید.

۴-۲-۲۷ شبه‌کدی برای یک الگوریتم چندریسمانی کارامد ارائه کنید که یک ماتریس $p \times q$ را در یک ماتریس $q \times r$ ضرب می‌کند. الگوریتم شما باید موازات بالایی داشته باشد، حتی اگر هر یک از p ، q ، و یا r برابر ۱ باشد. الگوریتم خود را تحلیل کنید.

۵-۲-۲۷ شبه‌کدی برای یک الگوریتم چندریسمانی کارامد ارائه کنید که یک ماتریس $n \times n$ را به صورت درجا به ماتریس ترانهاده تبدیل می‌کند. برای این کار، از یک روش تقسیم و حل استفاده کرده و ماتریس را به صورت بازگشتی به چهار زیرماتریس $n/2 \times n/2$ تبدیل کنید. الگوریتم خود را تحلیل کنید.

۶-۲-۲۷ شبه‌کدی برای یک پیاده‌سازی چندریسمانی کارامد از الگوریتم فلوید- وارشال (بخش ۲۵-۲ را ببینید) ارائه کنید، که کوتاه‌ترین مسیرها را بین تمام رأس‌های یک گراف وزن‌دار محاسبه می‌کند. الگوریتم خود را تحلیل کنید.

۳-۲۷ مرتب‌سازی ادغامی چندریسمانی

اولین بار در بخش ۲-۳-۱ مرتب‌سازی ادغامی را دیدیم، و در بخش ۲-۳-۲ زمان اجرای آن را تحلیل کرده و نشان دادیم که برابر $\theta(n \lg n)$ است. چون مرتب‌سازی ادغامی، خود از رویکرد تقسیم و حل استفاده می‌کند، به نظر می‌آید که یک کاندیدای عالی برای چندریسمانی‌سازی با استفاده از موازی‌سازی تودرتو باشد. به سادگی می‌توانیم شبه‌کد را اصلاح کنیم تا فراخوانی بازگشتی اول، تکثیر شود:

```
MERGE-SORT′(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      spawn MERGE-SORT′(A, p, q)
4      MERGE-SORT′(A, q+1, r)
5      sync
6      MERGE(A, p, q, r)
```

مانند نسخه‌ی سریال، MERGE-SORT′ زیرآرایه‌ی $A[p \,..\, r]$ را مرتب می‌کند. پس از این که دو فراخوانی بازگشتی در خطوط ۳ و ۴ پایان یافتند، که عبارت sync در خط ۵ از آن اطمینان حاصل می‌کند، MERGE-SORT′ همان رویه‌ی MERGE بخش ۲-۳ را فراخوانی می‌کند.

اجازه دهید MERGE-SORT SORT′ را تحلیل کنیم. برای این کار ابتدا باید MERGE را تحلیل کنیم. به خاطر بیاورید که زمان اجرای ادغام n عنصر در حالت سریال $\theta(n)$ است. چون MERGE سریال

است، هم کار آن برابر $\theta(n)$ است و هم دهانه‌ی آن. بنابراین رابطه‌ی بازگشتی زیر برای $MS'_1(n)$ ، کار MERGE-SORT′ را بر روی n عنصر تعیین می‌کند:

$$\begin{aligned} MS'_1(n) &= 2MS'_1(n/2) + \theta(n) \\ &= \theta(n \lg n) \end{aligned}$$

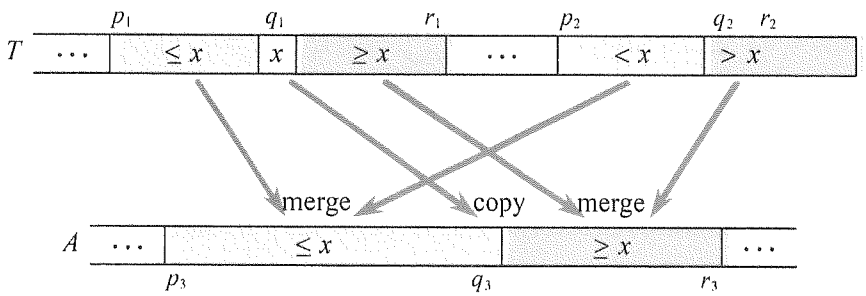
که برابر است با زمان اجرای مرتب‌سازی ادغامی سریال. از آن جایی که دو فراخوانی بازگشتی MERGE-SORT′ می‌توانند به صورت موازی اجرا شوند، دهانه (MS'_∞) توسط رابطه‌ی بازگشتی زیر تعیین می‌شود:

$$\begin{aligned} MS'_\infty(n) &= MS'_\infty(n/2) + \theta(n) \\ &= \theta(n) \end{aligned}$$

بنابراین موازات MERGE-SORT′ برابر است با $MS'_1(n)/MS'_\infty(n) = \theta(\lg n)$ ، که اصلاً قابل توجه نیست. مثلاً برای مرتب‌سازی ۱۰ میلیون عنصر، ممکن است بر روی تعداد کمی پردازنده این الگوریتم به تسریع خطی برسد، ولی برای صدها پردازنده تسریع بسیار ناچیز خواهد بود.

احتمالاً تا الان متوجه شده‌اید که تنگ‌راه موازات در این مرتب‌سازی ادغامی چندریسمانی کجا است: رویه‌ی MERGE سریال. با این که در ابتدا ممکن است به نظر برسد که ادغام ذاتا سریال است، ولی در واقع می‌توانیم با استفاده از موازی‌سازی تودرتو یک نسخه‌ی چندریسمانی برای آن تهیه کنیم.

استراتژی تقسیم و حل ما برای ادغام چندریسمانی، که در شکل ۲۷-۶ نشان داده شده است، روی زیرآرایه‌هایی از یک آرایه‌ی T عمل می‌کند. فرض کنید می‌خواهیم دو زیرآرایه‌ی مرتب‌شده‌ی $T[p_1 .. r_1]$ با طول $n_1 = r_1 - p_1 + 1$ و $T[p_2 .. r_2]$ با طول $n_2 = r_2 - p_2 + 1$ را در یک زیرآرایه‌ی



شکل ۲۷-۶ ایده‌ی پشت ادغام چندریسمانی دو زیرآرایه‌ی مرتب‌شده‌ی $T[p_1 .. r_1]$ و $T[p_2 .. r_2]$ در زیرآرایه‌ی $A[p_3 .. r_3]$. با فرض این که $x = T[q_1]$ میانه‌ی $T[p_1 .. r_1]$ باشد، و q_2 مکانی در $T[p_2 .. r_2]$ به طوری که x بین $T[q_2 - 1]$ و $T[q_2]$ قرار می‌گیرد، تمام عناصر در زیرآرایه‌های $T[p_1 .. q_1 - 1]$ و $T[p_2 .. q_2 - 1]$ (سایه‌ی کم‌رنگ) کم‌تر یا مساوی x هستند، و تمام عنصر در زیرآرایه‌های $T[q_1 + 1 .. r_1]$ و $T[q_2 + 1 .. r_2]$ (سایه‌ی پررنگ) حداقل برابر x اند. برای ادغام، اندیس q_3 را محاسبه می‌کنیم که x در $A[p_3 .. r_3]$ به آن‌جا تعلق دارد، x را در $A[q_3]$ کپی می‌کنیم، و سپس به صورت بازگشتی $T[p_1 .. q_1 - 1]$ را با $T[p_2 .. q_2 - 1]$ در $A[p_3 .. q_3 - 1]$ ، و $T[q_1 + 1 .. r_1]$ را با $T[q_2 .. r_2]$ را در $A[q_3 + 1 .. r_3]$ ادغام می‌کنیم.

$A[p_3 .. r_3]$ با طول $n_3 = r_3 - p_3 + 1$ ادغام کنیم. بدون از دست دادن کلیت و برای سادگی، فرض می‌کنیم $n_1 \geq n_2$.

ابتدا عنصر میانی $x = T[q_1]$ مربوط به زیرآرایه‌ی $T[p_1 .. r_1]$ را می‌یابیم، که در آن $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$. چون زیرآرایه، مرتب شده است، x برابر است با میانه‌ی $T[p_1 .. r_1]$: هیچ عنصری در $T[p_1 .. q_1 - 1]$ بیشتر از x نیست، و هیچ عنصری در $T[q_1 + 1 .. r_1]$ کم‌تر از x نیست. سپس از جستجوی دودویی استفاده می‌کنیم تا اندیس q_2 را در زیرآرایه‌ی $T[p_2 .. r_2]$ بیابیم به طوری که اگر x را بین $T[q_2 - 1]$ و $T[q_2]$ درج کنیم، زیرآرایه هنوز مرتب شده باقی بماند.

سپس به صورت زیر، زیرآرایه‌های اصلی $T[p_1 .. r_1]$ و $T[p_2 .. r_2]$ را در $A[p_3 .. r_3]$ ادغام می‌کنیم:

۱. قرار می‌دهیم $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$.
۲. x را در $A[q_3]$ کپی می‌کنیم.
۳. به صورت بازگشتی $T[p_1 .. q_1 - 1]$ را با $T[p_2 .. q_2 - 1]$ ادغام کرده و نتیجه را در زیرآرایه‌ی $A[p_3 .. q_3 - 1]$ قرار می‌دهیم.
۴. به صورت بازگشتی $T[q_1 + 1 .. r_1]$ را با $T[q_2 .. r_2]$ ادغام کرده و نتیجه را در زیرآرایه‌ی $A[q_3 + 1 .. r_3]$ قرار می‌دهیم.

وقتی q_3 را محاسبه می‌کنیم، کمیت $q_1 - p_1$ برابر است با تعداد عناصر در زیرآرایه‌ی $T[p_1 .. q_1 - 1]$، و کمیت $q_2 - p_2$ برابر است با تعداد عناصر در زیرآرایه‌ی $T[p_2 .. q_2 - 1]$ بنابراین مجموع آن‌ها برابر است با تعداد عناصری که در زیرآرایه‌ی $A[p_3 .. r_3]$ قبل از x قرار می‌گیرند.

حالت پایه زمانی رخ می‌دهد که داشته باشیم $n_1 = n_2 = 0$، که در این حالت لازم نیست برای ادغام دو زیرآرایه‌ی تهی کاری انجام دهیم. از آن جایی که فرض کرده‌ایم طول زیرآرایه‌ی $T[p_1 .. r_1]$ حداقل برابر است با طول $T[p_2 .. r_2]$، یعنی $n_1 \geq n_2$، می‌توانیم حالت پایه را فقط با تست $n_1 = 0$ چک کنیم. همچنین باید اطمینان حاصل کنیم که بازگشت، به درستی با حالتی که در آن فقط یکی از آرایه‌ها تهی است، برخورد می‌کند، که طبق فرض $n_1 \geq n_2$ باید زیرآرایه‌ی $T[p_2 .. r_2]$ باشد.

اکنون اجازه دهید این ایده‌ها را در شبه‌کد پیاده‌سازی کنیم. با جستجوی دودویی آغاز می‌کنیم، که آن را به صورت سریال انجام می‌دهیم. رویه‌ی BINARY-SEARCH(x, T, p, r) یک کلید x و یک زیرآرایه‌ی $T[p .. r]$ را دریافت می‌کند، و یکی از اندیس‌های زیر را بازمی‌گرداند:

- اگر $T[p .. r]$ تهی باشد ($r < p$)، آن گاه اندیس p بازگردانده می‌شود.
- اگر $x \leq T[p]$، و در نتیجه x کوچک‌تر یا مساوی تمام عناصر $T[p .. r]$ باشد، آن گاه اندیس p بازگردانده می‌شود.
- اگر $x > T[p]$، آن گاه بزرگ‌ترین اندیس q در بازه‌ی $p < q \leq r + 1$ بازگردانده می‌شود به طوری که $x > T[q - 1]$.

در زیر شبه‌کد این رویه را می‌بینیم.

BINARY-SEARCH(x, T, p, r)

```
1  low = p
2  high = max(p, r+1)
3  while low < high
4      mid = ⌊(low + high)/2⌋
5      if x ≤ T[mid]
6          high = mid
7      else low = mid + 1
8  return high
```

فراخوانی BINARY-SEARCH(x, T, p, r) در بدترین حالت به $\Theta(\lg n)$ زمان سریال نیاز دارد، که در آن $n = r - p + 1$ اندازه‌ی زیرآرایه‌ای است که رویه روی آن اجرا می‌شود. (تمرین ۲-۳-۵ را ببینید.) چون BINARY-SEARCH یک رویه‌ی سریال است، کار و دهانه‌ی آن در بدترین حالت برابرند با $\Theta(\lg n)$.

اکنون آماده‌ایم که برای ادغام چندریسمانی شبه‌کد بنویسیم. مانند رویه‌ی MERGE در بخش ۲-۳، رویه‌ی P-MERGE فرض می‌کند که دو زیرآرایه‌ای که باید ادغام شوند، درون یک آرایه قرار دارند. ولی برخلاف MERGE، رویه‌ی P-MERGE فرض نمی‌کند که دو زیرآرایه‌ای که باید ادغام شوند درون آرایه با هم مجاور هستند. (یعنی در P-MERGE لازم نیست داشته باشیم $p_2 = r_1 + 1$.) تفاوت دیگر بین MERGE و P-MERGE این است که P-MERGE آرایه‌ی خروجی A را که مقادیر ادغام شده باید در آن ذخیره شوند، به عنوان آرگومان ورودی دریافت می‌کند. فراخوانی MERGE($T, p_1, r_1, p_2, r_2, A, p_3$) زیرآرایه‌های مرتب‌شده‌ی $T[p_1 .. r_1]$ و $T[p_2 .. r_2]$ را در زیرآرایه‌ی $A[p_3 .. r_3]$ ادغام می‌کند، که در آن $r_3 = p_3 + (r_1 - p_1 + 1) + (r_2 - p_2 + 1) - 1 = p_3 + (r_1 - p_1) + (r_2 - p_2) + 1$ و این مقدار به عنوان ورودی ارائه نشده است.

```
P-MERGE(T, p1, r1, p2, r2, A, p3)
1   n1 = r1 − p1 + 1
2   n2 = r2 − p2 + 1
3   if n1 < n2                    // ensure that n1 ≥ n2
4       exchange p1 with p2
5       exchange r1 with r2
6       exchange n1 with n2
7   if n1 == 0                    // both empty?
8       return
9   else q1 = ⌊(p1 + r1)/2⌋
10      q2 = BINARY-SEARCH(T[q1], T, p2, r2)
11      q3 = p3 + (q1 − p1) + (q2 − p2)
12      A[q3] = T[q1]
13      spawn P-MERGE(T, p1, q1 − 1, p2, q2 − 1, A, p3)
14      P-MERGE(T, q1 + 1, r1, q2, r2, A, q3 + 1)
15      sync
```

رویه‌ی P-MERGE به صورت زیر کار می‌کند. خطوط ۱-۲، n_1 و n_2، به ترتیب طول زیرآرایه‌های $T[p_1 .. r_1]$ و $T[p_2 .. r_2]$ را محاسبه می‌کنند. خطوط ۳-۶ فرض $n_1 \geq n_2$ را برقرار می‌سازند. خط ۷ وجود حالت پایه را چک می‌کند، که در آن $T[p_1 .. r_1]$ تهی است (و همچنین $T[p_2 .. r_2]$)، که در این

حالت تابع به سادگی خارج می‌شود. خطوط ۹-۱۵ استراتژی تقسیم و حل را پیاده‌سازی می‌کنند. خط ۹ نقطه‌ی میانی $T[p_1 .. r_1]$ را محاسبه می‌کند، و خط ۱۰ نقطه‌ی q_2 را در $T[p_2 .. r_2]$ می‌یابد به طوری که تمام عناصر $T[p_2 .. q_2 - 1]$ کوچک‌تر از $T[q_1]$ (که متناظر است با x) باشند، و تمام عناصر $T[q_2 .. p_2]$ حداقل برابر با $T[q_1]$ باشند. خط ۱۱ اندیس q_3 را محاسبه می‌کند، که مربوط به عنصری است که آرایه‌ی خروجی $A[p_3 .. r_3]$ را به $A[p_3 .. q_3 - 1]$ و $A[q_3 + 1 .. r_3]$ تقسیم می‌کند، و سپس خط ۱۲، $T[q_1]$ را مستقیماً در $A[q_3]$ کپی می‌کند.

سپس با استفاده از موازی‌سازی تودرتو بازگشت را انجام می‌دهیم. خط ۱۳ اولین زیرمسئله را تکثیر می‌کند، در حالی که خط ۱۴ زیرمسئله‌ی دوم را به صورت موازی فراخوانی می‌کند. عبارت sync در خط ۱۵ اطمینان حاصل می‌کند که قبل از این که از رویه خارج شویم، حل زیرمسئله‌ها کامل شده است. (چون هر رویه‌ای قبل از خروج به صورت ضمنی یک sync انجام می‌دهد، می‌توانستیم از عبارت sync در خط ۱۵ صرف نظر کنیم، ولی استفاده از آن می‌تواند تمرین خوبی برای برنامه‌نویسی باشد.) در این کد، هنگامی که زیرآرایه‌ی $T[p_2 .. r_2]$ تهی است، با مقداری زیرکی عملیات مورد نیاز انجام می‌شود. روش کار آن این گونه است که در هر فراخوانی بازگشتی، یک عنصر میانه‌ی $T[p_1 .. r_1]$ در آرایه‌ی خروجی قرار داده می‌شود، تا زمانی که خود $T[p_1 .. r_1]$ نهایتاً تهی شود، که در این جا حالت پایه فعال می‌شود.

تحلیل ادغام چندریسمانی

ابتدا یک رابطه‌ی بازگشتی برای دهانه‌ی رویه‌ی P-MERGE ($PM_\infty(n)$) ارائه می‌کنیم، که در آن دو زیرآرایه، مجموعاً $n = n_1 + n_2$ عنصر دارند. چون تکثیر خط ۱۳ و فراخوانی خط ۱۴ منطقا موازی عمل می‌کنند، از بین دو فراخوانی فقط باید آن یکی را بررسی کنیم که پرهزینه‌تر است. مهم است درک کنیم که در بدترین حالت، بیشترین تعداد عناصر در این دو فراخوانی بازگشتی حداکثر $3n/4$ است، که به صورت زیر می‌بینیم چرا. چون خطوط ۳-۶ اطمینان حاصل می‌کنند که $n_2 \le n_1$ ، نتیجه می‌شود که $n_2 = 2n_2/2 \le (n_1 + n_2)/2 = n/2$. در بدترین حالت یکی از دو فراخوانی بازگشتی، $\lfloor n_1/2 \rfloor$ عنصر $T[p_1 .. r_1]$ را با n_2 عنصر $T[p_2 .. r_2]$ ادغام می‌کند، و بنابراین در بدترین حالت تعداد عناصر مورد بررسی در آن فراخوانی برابر است با

$$\begin{aligned} \lfloor n_1/2 \rfloor + n_2 &\le n_1/2 + n_2/2 + n_2/2 \\ &= (n_1 + n_2)/2 + n_2/2 \\ &\le n/2 + n/4 \\ &= 3n/4 \end{aligned}$$

با اضافه کردن هزینه‌ی $\theta(\lg n)$ برای فراخوانی BINARY-SEARCH در خط ۱۰، رابطه‌ی بازگشتی زیر را برای دهانه در بدترین حالت به دست می‌آوریم:

$$PM_\infty(n) = PM_\infty(3n/4) + \theta(\lg n) \qquad (۸-۲۷)$$

(برای حالت پایه دهانه برابر است با $\theta(1)$، چرا که خطوط ۱-۸ در زمان ثابت اجرا می‌شوند.) این

رابطه‌ی بازگشتی با هیچ یک از حالت‌های قضیه‌ی اصلی سازگار نیست، ولی شرط تمرین ۴-۶-۲ را ارضا می‌کند. بنابراین جواب رابطه‌ی بازگشتی (۲۷-۸) عبارت است از $PM_{\infty}(n)=\theta(\lg^{2} n)$.

اکنون کار رویه‌ی P-MERGE را روی n عنصر تحلیل می‌کنیم، که خواهیم دید جواب آن $PM_{1}(n)=\theta(n)$ است. چون هر یک از n عنصر باید از آرایه‌ی T به آرایه‌ی A کپی شوند، داریم $PM_{1}(n)=\Omega(n)$. پس فقط این باقی می‌ماند که نشان دهیم $PM_{1}(n)=O(n)$.

ابتدا یک رابطه‌ی بازگشتی برای کار در بدترین حالت به دست می‌آوریم. جستجوی دودویی در خط ۱۰ در بدترین حالت به زمان $\theta(\lg n)$ نیاز دارد، که به کار خارج فراخوانی‌های بازگشتی غلبه می‌کند. برای فراخوانی‌های بازگشتی، مشاهده کنید که با این که فراخوانی‌های بازگشتی خطوط ۱۳ و ۱۴ ممکن است تعداد متفاوتی عنصر را ادغام کنند، ولی روی هم n عنصر را ادغام می‌کنند (در واقع $n-1$ عنصر، چرا که $T[q_{1}]$ در هیچ کدام از فراخوانی‌های بازگشتی شرکت نمی‌کند.) به علاوه در تحلیل دهانه دیدیم که یک فراخوانی بازگشتی حداکثر روی $3n/4$ عنصر کار می‌کند. بنابراین رابطه‌ی بازگشتی

$$PM_{1}(n)=PM_{1}(\alpha n)+PM_{1}((1-\alpha)n)+O(\lg n) \qquad (۹-۲۷)$$

که در آن α در بازه‌ی $1/4 \le \alpha \le 3/4$ قرار می‌گیرد، و همچنین می‌دانیم مقدار واقعی α ممکن است برای هر سطح از بازگشت تغییر کند.

با استفاده از متد جانشین‌سازی اثبات می‌کنیم که جواب رابطه‌ی بازگشتی (۲۷-۹) برابر است با $PM_{1}=O(n)$. فرض کنید برای ثابت‌های مثبت c_{1} و c_{2} داشته باشیم $PM_{1}(n) \le c_{1}n-c_{2}\lg n$. با جانشین‌سازی به دست می‌آوریم

$$\begin{aligned}
PM_{1}(n) &\le (c_{1}\alpha n-c_{2}\lg(\alpha n))+(c_{1}(1-\alpha)n-c_{2}\lg((1-\alpha)n))+\theta(\lg n)\\
&=c_{1}(\alpha+(1-\alpha))n-c_{2}(\lg(\alpha n)+\lg((1-\alpha)n))+\theta(\lg n)\\
&=c_{1}n-c_{2}(\lg\alpha+\lg n+\lg(1-\alpha)+\lg n)+\theta(\lg n)\\
&=c_{1}n-c_{2}\lg n-(c_{2}(\lg n+\lg(\alpha(1-\alpha)))-\theta(\lg n))\\
&\le c_{1}n-c_{2}\lg n
\end{aligned}$$

چرا که می‌توانیم c_{2} را به اندازه‌ی کافی بزرگ انتخاب کنیم به طوری که $c_{2}(\lg n+\lg(\alpha(1-\alpha)))$ به عبارت $\theta(\lg n)$ غلبه کند. به علاوه می‌توانیم c_{1} را به اندازه‌ی کافی بزرگ انتخاب کنیم به طوری که شرط اولیه را برای بازگشت ارضا کند. چون کار $PM_{1}(n)$ برای P-MERGE هم از مرتبه‌ی $\Omega(n)$ است و هم از مرتبه‌ی $O(n)$، داریم $PM_{1}(n)=\theta(n)$.

موازات P-MERGE برابر است با $PM_{1}(n)/PM_{\infty}(n)=\theta(n/\lg^{2} n)$.

مرتب‌سازی ادغامی چندریسمانی

اکنون که یک رویه‌ی چندریسمانی ادغام با موازات خوب داریم، می‌توانیم از آن در مرتب‌سازی ادغامی چندریسمانی استفاده کنیم. این نسخه از مرتب‌سازی ادغامی مشابه رویه‌ی MERGE-SORT′ است که قبلاً دیدیم، ولی برخلاف MERGE-SORT′، این رویه یک زیرآرایه‌ی خروجی B را به عنوان

آرگومان ورودی دریافت می‌کند که نتایج مرتب‌شده در آن ذخیره خواهند شد. به طور خاص فراخوانی P-MERGE-SORT (A, p, r, B, s) عناصر $A[p..r]$ را مرتب کرده و آن‌ها را در $B[s..s+r-p]$ ذخیره می‌کند.

```
P-MERGE-SORT(A, p, r, B, s)
1  n = r − p + 1
2  if n == 1
3      B[s] = A[p]
4  else let T[1..n] be a new array
5      q = ⌊(p + r)/2⌋
6      q' = q − p + 1
7      spqwn P-MERGE-SORT(A, p, q, T, 1)
8      P-MERGE-SORT(A, q + 1, r, T, q' + 1)
9      sync
10     P-MERGE(T, 1, q', q' + 1, n, B, s)
```

بعد از این که خط ۱ تعداد عناصر آرایه‌ی $A[p..r]$ را محاسبه و در متغیر n ذخیره کرد، خطوط ۲-۳ حالت پایه را اداره می‌کنند، که در آن آرایه فقط ۱ عنصر دارد. خطوط ۴-۶ شرایط را برای تکثیر بازگشتی در خط ۷ و فراخوانی بازگشتی در خط ۸ آماده می‌کنند، که به صورت موازی عمل می‌کنند. به طور خاص، خط ۴ یک آرایه‌ی موقت T با n عنصر برای ذخیره‌ی نتایج مرتب‌سازی ادغامی بازگشتی تخصیص‌دهی می‌کند. خط ۵ اندیس q را در $A[p..r]$ محاسبه می‌کند تا به وسیله‌ی آن عناصر به دو زیرآرایه‌ی $A[p..q]$ و $A[q+1..r]$ (که به صورت بازگشتی مرتب خواهند شد) تقسیم شوند. خط ۶، q'، تعداد عناصر درون زیرآرایه‌ی اول ($A[p..q]$) را محاسبه می‌کند، که خط ۸ با استفاده از آن اندیس آغازین در T را محاسبه می‌کند، که نتایج مرتب‌شده‌ی $A[q+1..r]$ باید در آن جا ذخیره شوند. در این لحظه تکثیر و فراخوانی بازگشتی انجام شده‌اند، و عبارت sync در خط ۹ هم اجرا شده است، که رویه را مجبور می‌کند منتظر بماند تا رویه‌ی تکثیر شده پایان یابد. نهایتاً خط ۱۰ رویه‌ی P-MERGE را فراخوانی می‌کند تا زیرآرایه‌های مرتب‌شده را، که اکنون در $T[1..q']$ و $T[q'+1..n]$ قرار دارند، در زیرآرایه‌ی خروجی $B[s..s+r-p]$ ادغام کند.

تحلیل مرتب‌سازی ادغامی چندریسمانی

با تحلیل کار P-MERGE-SORT آغاز می‌کنیم، که بسیار آسان‌تر از تحلیل کار P-MERGE است. اگر کار را $PMS_1(n)$ بنامیم، رابطه‌ی بازگشتی زیر را برای آن داریم:

$$\begin{aligned} PMS_1(n) &= 2PMS_1(n/2) + PM_1(n) \\ &= 2PMS_1(n/2) + \theta(n) \end{aligned}$$

این رابطه‌ی بازگشتی مشابه رابطه‌ی (۴-۴) برای MERGE-SORT معمولی از بخش ۲-۳-۱ است، و طبق حالت ۲ قضیه‌ی اصلی، جواب آن برابر است با $PMS_1(n) = (n \lg n)$.

اکنون یک رابطه‌ی بازگشتی برای بدترین حالت دهانه به دست آورده و آن را تحلیل می‌کنیم. چون دو فراخوانی بازگشتی P-MERGE-SORT در خطوط ۷ و ۸ منطقا به صورت موازی کار می‌کنند،

می‌توانیم یکی از آن‌ها را نادیده گرفته و رابطه‌ی بازگشتی زیر را به دست آوریم:

$$\begin{aligned} PMS_\infty(n) &= PMS_\infty(n/2) + PM_\infty(n) \\ &= PMS_\infty(n/2) + \theta(\lg^2 n) \end{aligned} \qquad (۲۷-۱۰)$$

مانند رابطه‌ی بازگشتی (۲۷-۸)، قضیه‌ی اصلی را برای رابطه‌ی (۲۷-۱۰) هم نمی‌توان به کار برد، ولی تمرین ۴-۶-۲ را می‌توان. جواب برابر است با $PMS_\infty(n) = \theta(\lg^3 n)$، و بنابراین دهانه‌ی رویه‌ی P-MREGE-SORT برابر است با $\theta(\lg^3 n)$.

ادغام موازی نسبت به MERGE-SORT′ پیشرفت موازات قابل ملاحظه‌ای به P-MERGE-SORT می‌دهد. به خاطر بیاورید که موازات MERGE-SORT′، که از رویه‌ی MERGE سریال استفاده می‌کند، فقط $\theta(\lg n)$ است. برای P-MERGE-SORT موازات برابر است با

$$\begin{aligned} PMS_1(n)/PMS_\infty(n) &= \theta(n \lg n)/\theta(\lg^3 n) \\ &= \theta(n/\lg^2 n) \end{aligned}$$

که هم از نظر تئوری بسیار بهتر است و در عمل. در یک پیاده‌سازی خوب در عمل، احتمالاً بهتر است که مقداری موازات را فدا کرده و حالت پایه را کمی سخت‌تر کنیم، به این هدف که ثابت مخفی در نماد حدی را پایین بیاوریم. روش سرراست برای سخت کردن حالت پایه این است که وقتی اندازه‌ی آرایه به اندازه‌ی کافی کوچک شد، از یک مرتب‌سازی سریال معمولی، مثلاً مرتب‌سازی سریع، استفاده کنیم.

تمرین‌ها

۲۷-۳-۱ توضیح دهید که چطور می‌توان حالت پایه‌ی P-MERGE را سخت کرد.

۲۷-۳-۲ به جای یافتن عنصر میانه در زیرآرایه‌ی بزرگ‌تر، همان طور که P-MERGE این کار را انجام می‌دهد، نسخه‌ای را در نظر بگیرید که عنصر میانه‌ی تمام عناصر در دو زیرآرایه‌ی مرتب‌شده را با استفاده از نتیجه‌ی تمرین ۹-۳-۸ می‌یابد. شبه‌کدی ارائه کنید برای یک رویه‌ی ادغام چندریسمانی کارامد که از این رویه‌ی یافتن میانه استفاده می‌کند. الگوریتم خود را تحلیل کنید.

۲۷-۳-۳ یک الگوریتم چندریسمانی کارامد ارائه کنید برای تقسیم‌بندی یک آرایه نسبت به یک عنصر محور، همان طور که رویه‌ی PARTITION در بخش ۷-۱ انجام می‌دهد. نیازی نیست آرایه را درجا تقسیم‌بندی کنید. موازات الگوریتم خود را تا حد ممکن بالا ببرید. الگوریتم خود را تحلیل کنید. (*راهنمایی:* ممکن است به یک آرایه‌ی کمکی نیاز داشته باشید، و ممکن است مجبور باشید بیش از یک بار از روی عناصر آرایه عبور کنید.)

۲۷-۳-۴ یک نسخه‌ی چندریسمانی از رویه‌ی RECURSIVE-FFT در بخش ۳۰-۲ ارائه کنید. موازات پیاده‌سازی خود را تا حد ممکن بالا ببرید. الگوریتم خود را تحلیل کنید.

★ ۲۷-۳-۵ یک نسخه‌ی چندریسمانی از RANDOMIZED-SELECT در بخش ۹-۲ ارائه کنید. موازات پیاده‌سازی خود را تا حد ممکن بالا ببرید. الگوریتم خود را تحلیل کنید. (*راهنمایی:* از الگوریتم تقسیم‌بندی از تمرین ۲۷-۳-۳ استفاده کنید.)

★ ۲۷-۳-۶ نشان دهید که چطور می‌توان رویه‌ی SELECT از بخش ۹-۳ را چندریسمانی کرد. موازات پیاده‌سازی خود را تا حد ممکن بالا ببرید. الگوریتم خود را تحلیل کنید.

مسائل

۲۷-۱ *پیاده‌سازی حلقه‌های موازی با استفاده از موازی‌سازی تودرتو*

الگوریتم چندریسمانی زیر را برای انجام جمع عنصر به عنصر بر روی آرایه‌های nعنصری $A[1..n]$ و $B[1..n]$ در نظر بگیرید، که نتیجه را در آرایه‌ی $C[1..n]$ ذخیره می‌کند:

```
SUM-ARRAYS(A, B, C)
1  parallel for i = 1 to A.length
2      C[i] = A[i] + B[i]
```

I. حلقه‌ی موازی رویه‌ی SUM-ARRAYS را با استفاده از موازی‌سازی تودرتو (spawn و sync) به روش MAT-VEC-MAIN-LOOP بازنویسی کنید. موازات پیاده‌سازی خود را تحلیل کنید.

پیاده‌سازی جایگزین زیر را برای حلقه‌ی موازی در نظر بگیرید، که در آن یک مقدار *grain-size* وجود دارد که باید بعداً تعیین شود:

```
SUM-ARRAYS'(A, B, C)
1  n = A.length
2  grain-size = ?            // to be determined
3  r = ⌈n/grain-size⌉
4  for k = 0 to r - 1
5      spawn ADD-SUBARRAY(A, B, C, k · grain-size + 1,
                          min((k+1) · grain-size, n))
6  sync

ADD-SUBARRAY(A, B, C, i, j)
1  for k = i to j
2      C[k] = A[k] + B[k]
```

II. فرض کنید قرار می‌دهیم $grain\text{-}size = 1$. موازات این پیاده‌سازی چقدر است؟

III. فرمولی ارائه کنید برای دهانه‌ی رویه‌ی SUM-ARRAYS' نسبت به n و $grain\text{-}size$. بهترین مقدار را برای $grain\text{-}size$ به دست آورید به طوری که موازات، بیشینه شود.

۲۷-۲ *صرفه‌جویی در فضای موقت در ضرب ماتریس‌ها*

رویه‌ی P-MATRIX-MULTIPLY-RECURSIVE یک ضعف دارد، و آن این است که باید یک

ماتریس موقت T با اندازه‌ی $n \times n$ در حافظه تخصیص دهد، که بر روی ثابت مخفی در نماد θ تأثیر منفی می‌گذارد. ولی رویه‌ی P-MATRIX-MULTIPLY-RECURSIVE موازات بالایی دارد. برای مثال با نادیده گرفتن ثابت‌های نماد θ، موازات ضرب دو ماتریس ۱۰۰۰×۱۰۰۰ تقریباً برابر است با $۱۰^۳/۱۰^۲ = ۱۰^۷$، چرا که $\lg ۱۰۰۰ \approx ۱۰$. اکثر کامپیوترهای موازی کم‌تر از ۱۰ میلیون پردازنده دارند.

I. یک الگوریتم بازگشتی چندریسمانی ارائه کنید که با هزینه‌ی افزایش دهانه به $\theta(n)$، نیاز به ماتریس موقت T را از بین می‌برد. (*راهنمایی:* طبق استراتژی کلی P-MATRIX-MULTIPLY-RECURSIVE، $C = C + AB$ را محاسبه کنید، ولی C را به صورت موازی مقداردهی اولیه کرده و یک sync در مکانی مناسب قرار دهید.)

II. برای دهانه و کار پیاده‌سازی خود، رابطه‌هایی بازگشتی ارائه کرده و آن‌ها را حل کنید.

III. موازات پیاده‌سازی خود را تحلیل کنید. با نادیده گرفتن ثابت‌های نماد θ، موازات را برای ماتریس‌های ۱۰۰۰×۱۰۰۰ تخمین بزنید. این موازات را با موازات P-MATRIX-MULTIPLY-RECURSIVE مقایسه کنید.

۲۷-۳ الگوریتم‌های چندریسمانی ماتریس

I. رویه‌ی LU-DECOMPOSITION بخش ۲۸-۱ را موازی‌سازی کرده و یک شبه‌کد برای نسخه‌ی چندریسمانی آن ارائه کنید. موازات پیاده‌سازی خود را تا حد ممکن بالا برده و کار، دهانه، و موازات آن را تحلیل کنید.

II. همین کار را برای LUP-DECOMPOSITION در بخش ۲۸-۱ انجام دهید.

III. همین کار را برای LUP-SOLVE در بخش ۲۸-۱ انجام دهید.

IV. همین کار را برای یک الگوریتم چندریسمانی بر مبنای تساوی (۲۸-۱۳) برای معکوس کردن یک ماتریس متقارن مطلقا مثبت بکنید.

۲۷-۴ کاهش‌ها و محاسبات پیشوندی چندریسمانی

یک $\otimes$ – کاهش ($\otimes$ -reduction) از یک آرایه‌ی $x[1..n]$، که در آن $\otimes$ یک عمل‌گر شرکت‌پذیر است، برابر است با مقدار

$$y = x[1] \otimes x[2] \otimes \ldots \otimes x[n]$$

رویه‌ی زیر $\otimes$ – کاهش یک زیرآرایه‌ی $x[i..j]$ را به صورت سریال محاسبه می‌کند.

```
REDUCE(x, i, j)
1  y = x[i]
2  for k = i + 1 to j
3      y = y ⊗ x[k]
4  return y
```

۱. از موازی‌سازی تودرتو استفاده کرده و یک الگوریتم چندریسمانی برای P-REDUCE ارائه کنید، که همان نتیجه را با کار $\theta(n)$ و دهانه‌ی $\theta(\lg n)$ تولید می‌کند. الگوریتم خود را تحلیل کنید.

یک مسئله‌ی مرتبط، یافتن یک $\otimes$- *محاسبه‌ی پیشوندی* (prefix computation -$\otimes$) روی یک آرایه‌ی $x[1..n]$ است، که بعضی مواقع به آن $\otimes$- *پویش* (scan -$\otimes$) می‌گویند، و در آن دوباره $\otimes$ یک عمل‌گر شرکت‌پذیر است. $\otimes$- پویش طبق روابط زیر یک آرایه‌ی $y[1..n]$ تولید می‌کند:

$$\begin{aligned}
y[1] &= x[1], \\
y[2] &= x[1] \otimes x[2], \\
y[3] &= x[1] \otimes x[2] \otimes x[3], \\
&\vdots \\
y[n] &= x[1] \otimes x[2] \otimes x[3] \otimes \ldots \otimes x[n]
\end{aligned}$$

یعنی تمام پیشوندهای آرایه‌ی x با استفاده از عمل‌گر $\otimes$ با هم «جمع» می‌شوند. رویه‌ی سریال SCAN زیر محاسبه‌ی $\otimes$- پویش را انجام می‌دهد:

```
SCAN(x)
1  n = x.length
2  let y[1..n] be a new array
3  y[1] = x[1]
4  for i = 2 to n
5      y[i] = y[i − 1]  ⊗ x[i]
6  return y
```

متأسفانه چندریسمانی‌سازی SCAN چندان سرراست نیست. برای مثال تغییر حلقه‌ی **for** به یک حلقه‌ی **parallel for** چالش به وجود خواهد آورد، چرا که هر تکرار از بدنه‌ی حلقه به تکرارهای قبلی وابسته است. رویه‌ی P-SCAN-1 زیر، هر چند ناکارامد، محاسبه‌ی $\otimes$- پویش را به صورت موازی انجام می‌دهد:

```
P-SCAN-1(x)
1  n = x.length
2  let y[1..n] be a new array
3  P-SCAN-1-AUX(x, y, 1, n)
4  return y

P-SCAN-1-AUX(x, y, i, j)
1  parallel for l = i to j
2      y[l] = P-REDUCE(x, 1, l)
```

۱. کار، دهانه، و موازات P-SCAN-1 را تحلیل کنید.

با استفاده از موازی‌سازی تودرتو می‌توانیم به روش بهینه‌تری برای محاسبه‌ی $\otimes$ -پویش دست یابیم:

```
P-SCAN-2(x)
1  n = x.length
2  let y[1..n] be a new array
3  P-SCAN-2-AUX(x, y, 1, n)
```

```
4  return y

P-SCAN-2-AUX(x, y, i, j)
1  if i == j
2      y[i] = x[i]
3  else k = ⌊(i + j)/2⌋
4      spawn P-SCAN-2-AUX(x, y, i, k)
5      P-SCAN-2-AUX(x, y, k + 1, j)
6      sync
7      parallel for l = k + 1 to j
8          y[l] = y[k] ⊗ y[l]
```

۱۱. بحث کنید که P-SCAN-2 درست عمل می‌کند، و کار، دهانه، و موازات آن را تحلیل کنید.

می‌توانیم با محاسبه‌ی ⊗-پویش در دو عبور مجزا از روی داده‌ها، در هر دوی P-SCAN-1 و P-SCAN-2 بهبود ایجاد کنیم. در عبور اول، عبارت‌های مربوط به زیرآرایه‌های متوالی مختلفی از x را در یک آرایه‌ی موقت t جمع‌آوری می‌کنیم، و در عبور دوم از عبارت‌های ذخیره شده در t استفاده کرده و نتیجه‌ی نهایی y را به دست می‌آوریم. شبه‌کد زیر این استراتژی را پیاده‌سازی می‌کند، ولی در آن بعضی از عبارت‌ها حذف شده‌اند:

```
P-SCAN-3(x)
1  n = x.length
2  let y[1..n] and t[1..n] be new arrays
3  y[1] = x[1]
4  if n > 1
5      P-SCAN-UP(x, t, 2, n)
6      P-SCAN-DOWN(x[1], x, t, y, 2, n)
7  return y

P-SCAN-UP(x, t, i, j)
1  if i == j
2      return x[i]
3  else
4      k = ⌊(i + j)/2⌋
5      t[k] = spawn P-SCAN-UP(x, t, i, k)
6      right = P-SCAN-UP(x, t, k + 1, j)
7      sync
8      return ----                    // fill in the blank

P-SCAN-DOWN(v, x, t, y, i, j)
1  if i == j
2      y[i] = v ⊗ x[i]
3  else
4      k = ⌊(i + j)/2⌋
5      spawn P-SCAN-DOWN(----, x, t, y, i, k)  // fill in the blank
6      P-SCAN-DOWN(----, x, t, y, k + 1, j)    // fill in the balnk
7      sync
```

III. سه جای خالی را در خط ۸ رویه‌ی P-SCAN-UP و خطوط ۵ و ۶ رویه‌ی P-SCAN-DOWN پر کنید. بحث کنید که با عبارت‌هایی که ارائه کرده‌اید، P-SCAN-3 به درستی کار می‌کند. (*راهنمایی:* اثبات کنید که مقدار v ارسال شده به P-SCAN-DOWN(v, x, t, y, i, j) عبارت $v = x[1] \otimes x[2] \otimes ... \otimes x[i-1]$ را ارضا می‌کند.)

IV. کار، دهانه، و موازات P-SCAN-3 را تحلیل کنید.

۲۷-۵ چندریسمانی‌سازی یک محاسبه‌ی ساده‌ی شابلونی

علوم کامپیوتر مملو است از الگوریتم‌هایی که نیاز دارند آرایه‌ای با مقادیری پر شود که به مقادیر ورودی‌هایی در همسایگی (که قبلاً محاسبه شده‌اند) بستگی دارد، همچنین به اطلاعاتی که در طول اجرای برنامه تغییر نمی‌کنند. الگوی ورودی‌های همسایه، که در حین محاسبه تغییر نمی‌کند، *شابلون* (stencil) نام دارد. برای مثال بخش ۱۵-۴ یک الگوریتم شابلونی ارائه می‌کند برای محاسبه‌ی بلندترین زیردنباله‌ی مشترک، که در آن مقدار $c[i, j]$ فقط به مقادیر $c[i-1, j]$، $c[i, j-1]$، و $c[i-1, j-1]$ بستگی دارد، و همچنین به عناصر x_i و y_j در دو دنباله که به عنوان ورودی داده شده‌اند. دنباله‌های ورودی ثابت هستند، ولی الگوریتم طوری آرایه‌ی دوبعدی c را پر می‌کند که ورودی $c[i, j]$ بعد از محاسبه‌ی هر سه ورودی $c[i-1, j]$، $c[i, j-1]$، و $c[i-1, j-1]$ پر شود.

در این مسئله بررسی می‌کنیم که چطور می‌توان با استفاده از موازی‌سازی تودرتو، یک محاسبه‌ی ساده‌ی شابلونی روی یک آرایه‌ی A با اندازه‌ی $n \times n$ را چندریسمانی کرد، که در آن مقدار ورودی $A[i, j]$ فقط به مقادیر ورودی‌های $A[i', j]$ بستگی دارد، به طوری که $i' \le i$ و $j' \le j$ (و مسلماً $i' \ne i$ و $j' \ne j$). به عبارت دیگر، مقدار یک ورودی فقط به مقادیری بستگی دارد که بالا و/یا سمت چپ آن هستند، به علاوه‌ی اطلاعات ثابت خارج از آرایه. به علاوه در طول این مسئله فرض می‌کنیم وقتی ورودی‌هایی را که $A[i, j]$ به آن بستگی دارند پر کردیم، می‌توانیم $A[i, j]$ را در زمان $\theta(1)$ پر کنیم (مانند رویه‌ی LSC-LENGTH در بخش ۱۵-۴).

می‌توانیم به صورت زیر آرایه‌ی A را به چهار زیرآرایه تقسیم کنیم:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad (۱۰-۲۷)$$

اکنون مشاهده کنید که می‌توانیم زیرآرایه‌ی A_{11} را به صورت بازگشتی پر کنیم، چرا که به ورودی‌های سه زیرآرایه‌ی دیگر بستگی ندارد. وقتی A_{11} کامل شد، می‌توانیم A_{12} و A_{21} را به صورت بازگشتی و موازی پر کنیم، چرا که با این که هر دوی آن‌ها به A_{11} بستگی دارند، ولی به یکدیگر بستگی ندارند. در نهایت می‌توانیم A_{22} را به صورت بازگشتی پر کنیم.

I. یک شبه‌کد چندریسمانی ارائه کنید که این محاسبه‌ی ساده‌ی شابلونی را با استفاده از یک الگوریتم تقسیم و حل SIMPLE-STENCIL بر پایه‌ی تجزیه‌ی (۲۷-۱۱) و بحث بالا انجام

می‌دهد. (نگران جزئیات حالت پایه نباشید، چرا که به نوع محاسبه‌ی شابلونی مورد استفاده بستگی دارد.) روابطی بازگشتی برای کار و دهانه‌ی این الگوریتم نسبت به n ارائه کرده و آن‌ها را حل کنید. موازات این الگوریتم چقدر است؟

II. جواب خود به بخش I را طوری اصلاح کنید که یک آرایه‌ی $n \times n$ را به ۹ زیرآرایه‌ی $n/۳ \times n/۳$ تقسیم کند، و دوباره با حداکثر موازات ممکن بازگشت کند. این الگوریتم را تحلیل کنید. موازات این الگوریتم نسبت به الگوریتم بخش I چقدر بیشتر یا کم‌تر است؟

III. جواب خود به بخش‌های I و II را به صورت زیر گسترش دهید. یک عدد صحیح $b \geq ۲$ انتخاب کنید. یک آرایه‌ی $n \times n$ را به $b^۲$ زیرآرایه تقسیم کنید، هر یک با اندازه‌ی $n/b \times n/b$، و با حداکثر موازت ممکن بازگشت کنید. بر حسب n و b، کار، دهانه، و موازات الگوریتم شما چقدر است؟ بحث کنید که با استفاده از این رویکرد، موازات باید برای هر انتخابی از $b \geq ۲$ برابر $o(n)$ باشد. (*راهنمایی:* برای این بحث آخر، نشان دهید که توان n در موازات برای هر انتخابی به صورت $b \geq ۲$ اکیداً کم‌تر از ۱ است.)

IV. شبه‌کدی برای یک الگوریتم چندریسمانی برای این محاسبه‌ی ساده‌ی شابلونی ارائه کنید که به موازات $\theta(n/\lg n)$ دست می‌یابد. با استفاده از مفاهیم کار و دهانه بحث کنید که در واقع این مسئله دارای موازات ذاتی $\theta(n)$ است. همان طور که متوجه شده‌اید، ذات تقسیم و حل این شبه‌کد چندریسمانی به ما اجازه نمی‌دهد که به این موازات بیشینه دست یابیم.

۲۷-۶ الگوریتم‌های چندریسمانی تصادفی

درست مانند الگوریتم‌های سریال معمولی، بعضی مواقع نیاز داریم که الگوریتم‌های چندریسمانی تصادفی تولید کنیم. این مسئله بررسی می‌کند که چطور می‌توان معیارهای مختلف کارایی را طوری اصلاح کرد که امیدریاضی رفتار چنین الگوریتم‌هایی را توصیف کنند. همچنین از شما می‌خواهد یک الگوریتم چندریسمانی برای مرتب‌سازی سریع تصادفی ارائه کرده و آن را تحلیل کنید.

I. توضیح دهید که چطور می‌توان قانون کار (۲۷-۲)، قانون دهانه (۲۷-۳)، و کران برنامه‌ریز حریص (۲۷-۴) را طوری اصلاح کرد که با امیدریاضی کار کنند، در حالتی که T_p، $T_۱$، و T_∞ متغیرهای تصادفی هستند.

II. یک الگوریتم چندریسمانی تصادفی را در نظر بگیرید که در برای آن، در ۱٪ مواقع داریم $T_۱ = ۱۰^۴$ و $T_{۱۰,۰۰۰} = ۱$، ولی برای ۹۹٪ مواقع داریم $T_۱ = T_{۱۰,۰۰۰} = ۱۰^۹$. بحث کنید که *تسریع* یک الگوریتم چندریسمانی تصادفی باید به صورت $E[T_۱]/E[T_p]$ تعریف شود، نه به صورت $E[T_۱/T_p]$.

III. بحث کنید که *موازات* یک الگوریتم چندریسمانی موازی باید به صورت نسبت $E[T_۱]/E[T_\infty]$ تعریف شود.

IV. الگوریتم RANDOMIZED-QUICKSORT بخش ۷-۳ را با استفاده از موازی‌سازی تودرتو،

چندریسمانی کنید. (RANDOMIZED-PARTITION را موازی‌سازی نکنید.) برای الگوریتم P-RANDOMIZED-QUICKSORT خود شبه‌کد ارائه کنید.

۷. الگوریتم چندریسمانی خود برای مرتب‌سازی سریع تصادفی را تحلیل کنید. (*راهنمایی:* تحلیل RANDOMIZED-SELECT در بخش ۹-۲ را بازبینی کنید.)

فصل

۲۸

اعمال ماتریس‌ها

۲۸-۰ مقدمه

از آن جایی که اعمال ماتریس‌ها در قلب محاسبات علمی قرار دارند، الگوریتم‌های کار با ماتریس از نظر عملی بسیار مهم‌اند. این فصل بر روی نحوه‌ی ضرب ماتریس‌ها و حل مجموعه معادلات خطی هم‌زمان تمرکز می‌کند. در پیوست ت مبانی ماتریس‌ها بررسی خواهند شد.

بخش ۲۸-۱ نشان می‌دهد که چگونه می‌توان مجموعه‌ای از معادلات خطی را با استفاده از تجزیه‌ی LUP حل کرد. سپس در بخش ۲۸-۲ رابطه‌ی نزدیک میان مسئله‌ی ضرب ماتریس‌ها و مسئله‌ی پیدا کردن وارون یک ماتریس مشخص می‌شود. نهایتاً در بخش ۲۸-۳ کلاس مهم ماتریس‌های مطلقاً مثبت متقارن بررسی می‌شوند، و نشان داده می‌شود که چگونه می‌توان از آن‌ها برای یافتن یک جواب کم‌ترین مربعات برای مجموعه‌ای از معادلات خطی فرامعین استفاده کرد.

یک مسئله‌ی مهم که در عمل پیش می‌آید، *پایداری عددی* (numerical stability) است. به خاطر دقت محدود نمایش ممیز شناور در کامپیوترهای واقعی، خطاهای گرد کردن در محاسبات عددی ممکن است در جریان یک محاسبه‌ی خاص تقویت شوند، که به نتایج غلط ختم می‌شود؛ می‌گوییم چنین محاسباتی از نظر عددی ناپایدار (numerically unstable) هستند. با این که بعضی مواقع پایداری عددی را به صورت مختصر در نظر می‌گیریم، در این فصل بر روی آن تمرکز نمی‌کنیم. برای یک بحث کامل بر روی مباحث پایداری کتاب جامع نوشته شده توسط Golub و Van Loan را به خواننده پیشنهاد می‌کنیم.

۲۸-۱ حل سیستم‌های معادلات خطی

حل مجموعه‌ای از معادلات خطی هم‌زمان یکی از مسائل مهمی است که در کاربردهای مختلفی پیش می‌آید. یک سیستم خطی را می‌توان به صورت معادله‌ای ماتریسی بیان کرد که در آن هر عنصر ماتریس یا بردار به یک مجموعه تعلق دارد، مثلاً مجموعه‌ی اعداد حقیقی $\mathbb{R}$. این بخش به این می‌پردازد که چگونه می‌توان یک سیستم معادلات خطی را با استفاده از متدی به نام تجزیه‌ی LUP حل کرد.

با مجموعه‌ای از معادلات خطی با n مجهول $x_1, x_2, \ldots, x_n$ شروع می‌کنیم:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_1, \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n. \end{aligned} \tag{۲۸-۱}$$

به مجموعه‌ای از مقادیر $x_1, x_2, \ldots, x_n$ که تمام تساوی‌های (۲۸-۱) را به صورت هم‌زمان ارضا کند، یک *جواب* برای این معادلات گفته می‌شود. در این بخش فقط به حالتی می‌پردازیم که در آن دقیقاً n معادله و n مجهول وجود دارد.

به راحتی می‌توانیم تساوی‌های (۲۸-۱) را به صورت یک تساوی ماتریس-بردار بنویسیم:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

یا به طور مشابه، با قرار دادن $A = (a_{ij})$، $x = (x_i)$، و $b = (b_i)$، به صورت

$$Ax = b \tag{۲۸-۲}$$

اگر A غیرتکین باشد، یک معکوس A^{-1} دارد، و

$$x = A^{-1}b \tag{۲۸-۳}$$

بردار جواب است. به صورت زیر می‌توانیم اثبات کنیم که x جواب یکتای تساوی (۲۸-۲) است. اگر دو جواب x و x' وجود داشته باشد، در این صورت $Ax = Ax' = b$، و با فرض این که I نشان دهنده‌ی ماتریس همانی باشد،

$$\begin{aligned} x &= Ix \\ &= (A^{-1}A)x \\ &= A^{-1}(Ax) \\ &= A^{-1}(Ax') \\ &= (A^{-1}A)x' \\ &= x' \end{aligned}$$

در این بخش به طور عمده به حالتی می‌پردازیم که A غیرتکین است، یا به طور مشابه (طبق قضیه‌ی ت-۱) رتبه‌ی A برابر است با تعداد مجهولات، یعنی n. با این حال وضعیت‌های ممکن دیگری هم وجود دارند، که بد نیست در مورد آن‌ها هم مختصراً بحث کنیم. اگر تعداد معادلات کم‌تر از تعداد مجهولات باشد – یا به طور کلی‌تر، اگر رتبه‌ی A کم‌تر از n باشد – آن گاه سیستم *نامعین* (undetermined) است. یک سیستم نامعین معمولاً بی‌نهایت جواب دارد. البته اگر معادلات ناسازگار باشند سیستم هیچ جوابی نخواهد داشت. اگر تعداد معادلات از تعداد مجهولات بیشتر شود، سیستم *فرامعین* (overdetermined) خواهد بود. ممکن است هیچ جوابی برای یک سیستم فرامعین وجود نداشته باشد. یافتن جواب‌های تقریبی خوب برای سیستم‌های فرامعین از معادلات خطی، مسئله‌ی مهمی است که در بخش ۲۸-۳ در مورد آن بحث خواهد شد.

اجازه دهید به مسئله‌ی حل سیستم $Ax = b$ با n معادله و n مجهول بازگردیم. یک رویکرد این است که A^{-1} را محاسبه کرده و سپس، با استفاده از تساوی (۲۸-۳)، b را در A^{-1} ضرب کنیم، که نتیجه می‌دهد $x = A^{-1}b$. این رویکرد در عمل از ناپایداری عددی رنج می‌برد. خوشبختانه روش دیگری هم وجود دارد – تجزیه‌ی LUP – که هم به صورت عددی پایدار است و هم در عمل سریع‌تر اجرا می‌شود.

مقدمه‌ای بر تجزیه‌ی LUP

ایده‌ی پشت تجزیه‌ی LUP این است که سه ماتریس L، U، و P را با اندازه‌ی $n \times n$ بیابیم به طوری که

$$PA = LU \tag{۲۸-۴}$$

که در آن

- L یک ماتریس یکه‌ی پایین- مثلثی است،
- U یک ماتریس بالا- مثلثی است، و
- P یک ماتریس جایگشت است.

به ماتریس‌های L، U، و P که تساوی (۲۸-۱۹) را ارضا کنند *تجزیه‌ی LUP* ماتریس A می‌گوییم. نشان خواهیم داد که هر ماتریس غیرتکین A را می‌توان بدین صورت تجزیه کرد.

مزیت محاسبه‌ی تجزیه‌ی LUP برای ماتریس A این است که در حالتی که سیستم‌های خطی مثلثی هستند، مانند هر دو ماتریس L و U، ساده‌تر می‌توان آن‌ها را حل کرد. با یافتن یک تجزیه‌ی LUP برای A می‌توانیم تساوی (۲۸-۲)، یعنی $Ax = b$ را با حل کردن فقط سیستم‌های خطی مثلثی حل کنیم، به صورت زیر. ضرب هر دو طرف $Ax = b$ در P تساوی $PAx = Pb$ را نتیجه می‌دهد، که طبق تمرین ت-۱-۴ با بازآرایی تساوی (۲۸-۱) به دست می‌آید. با استفاده از تجزیه‌ی (۲۸-۴) به دست می‌آوریم

$$LUx = Pb$$

اکنون می‌توانیم با حل دو سیستم خطی مثلثی، این معادله را حل کنیم. اجازه دهید تعریف کنیم $y = Ux$، که در آن x بردار جواب مورد نظر است. ابتدا سیستم پایین مثلثی

$$Ly = Pb \qquad (۲۸-۵)$$

را برای بردار مجهول y توسط متدی با نام «جایگزینی جلویی» (forward substitution) حل می‌کنیم. پس از یافتن y، سیستم بالا-مثلثی

$$Ux = y \qquad (۲۸-۶)$$

را برای مجهول x توسط متدی با نام «جایگزینی عقبی» حل می‌کنیم. از آن جایی که ماتریس جایگشت P معکوس‌پذیر است (تمرین ت-۲-۳)، ضرب دو طرف تساوی (۲۸-۴) در P^{-1} به دست می‌دهد $P^{-1}PA = P^{-1}LU$، به طوری که

$$A = P^{-1}LU \qquad (۲۸-۷)$$

بنابراین بردار x جواب تساوی $Ax = b$ است:

$$\begin{aligned}
Ax &= P^{-1}LUx && \text{(طبق تساوی (۲۸-۷))}\\
&= P^{-1}Ly && \text{(طبق تساوی (۲۸-۶))}\\
&= P^{-1}Pb && \text{(طبق تساوی (۲۸-۵))}\\
&= b
\end{aligned}$$

مرحله‌ی بعدی این است که نشان دهیم جایگزینی‌های جلویی و عقبی کار می‌کنند، و سپس به سراغ مسئله‌ی یافتن تجزیه‌ی LUP برویم.

جایگزینی‌های جلویی و عقبی

جایگزینی جلویی (forward substitution) می‌تواند با دریافت L، P و b، سیستم پایین-مثلثی (۲۸-۵) را در زمان $\theta(n^2)$ حل کند. برای سادگی، جایگشت P را به صورت فشرده به کمک یک آرایه‌ی $\pi[1..n]$ نشان می‌دهیم. برای $i = 1, 2, \ldots, n$، ورودی $\pi[i]$ نشان می‌دهد که $P_{i,\pi[i]} = 1$ و $P_{ij} = 0$ برای $j \neq \pi[i]$. بنابراین مقدار PA در سطر i و ستون j برابر $a_{\pi[i],j}$ است، و مقدار عنصر iام Pb برابر است با $b_{\pi[i]}$. از آن جایی که L پایین- مثلثی و یکه است، تساوی (۲۸-۵) را می‌توان به صورت

$$\begin{aligned}
y_1 &= b_{\pi[1]},\\
l_{21}y_1 + y_2 &= b_{\pi[2]},\\
l_{31}y_1 + l_{32}y_2 + y_3 &= b_{\pi[3]},\\
&\vdots\\
l_{n1}y_1 + l_{n2}y_2 + l_{n3}y_3 + \ldots + y_n &= b_{\pi[n]}
\end{aligned}$$

بازنویسی کرد. اولین تساوی به ما می‌گوید که $y_1 = b_{\pi[1]}$. با دانستن مقدار y_1 می‌توانیم آن را در تساوی دوم جایگزین کنیم، که به دست می‌دهد

$$y_2 = b_{\pi[2]} - l_{21} y_1$$

اکنون می‌توانیم هر دوی y_1 و y_2 را در تساوی سوم جایگزین کنیم، که خواهیم داشت

$$y_3 = b_{\pi[3]} - (l_{31} y_1 + l_{32} y_2)$$

به طور کلی می‌توانیم $y_{i-1}, \ldots, y_2, y_1$ را به «جلو» و در تساوی iام جایگزین کنیم و معادله رانسبت به y_i حل کنیم:

$$y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij} y_j$$

پس از حل نسبت به y، تساوی (۲۸-۶) را با استفاده از *جایگزینی عقبی* (back substitution) نسبت به x حل می‌کنیم، که مشابه جایگزینی جلویی انجام می‌شود. با داشتن U و y، ابتدا nامین تساوی را حل می‌کنیم و سپس به سمت عقب و تساوی اول حرکت می‌کنیم. مانند جایگزینی جلویی این فرآیند در زمان $\theta(n^2)$ انجام می‌شود. از آن جایی که U بالا-مثلثی است، می‌توانیم سیستم (۲۸-۶) را به صورت

$$\begin{aligned}
u_{11} x_1 + u_{12} x_2 + \ldots + u_{1,n-2} x_{n-2} + u_{1,n-1} x_{n-1} + u_{1n} x_n &= y_1, \\
u_{22} x_2 + \ldots + u_{2,n-2} x_{n-2} + u_{2,n-1} x_{n-1} + u_{2n} x_n &= y_2, \\
&\vdots \\
u_{n-2,n-2} x_{n-2} + u_{n-2,n-1} x_{n-1} + u_{n-2,n} x_n &= y_{n-2}, \\
u_{n-1,n-1} x_{n-1} + u_{n-1,n} x_n &= y_{n-1}, \\
u_{n,n} x_n &= y_n
\end{aligned}$$

بازنویسی کنیم. بنابراین می‌توانیم $x_1, \ldots, x_{n-1}, x_n$ را به ترتیب و به صورت زیر به دست آوریم:

$$\begin{aligned}
x_n &= y_n / u_{n,n}, \\
x_{n-1} &= (y_{n-1} - u_{n-1,n} x_n) / u_{n-1,n-1}, \\
x_{n-2} &= (y_{n-2} - (u_{n-2,n-1} x_{n-1} + u_{n-2,n} x_n)) / u_{n-2,n-2} \\
&\vdots
\end{aligned}$$

یا به طور کلی

$$x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij} x_j \right) \Big/ u_{ii}$$

با داشتن P، L، U، و b، رویه‌ی LUP-SOLVE با ترکیب جایگزینی جلویی و عقبی، x را به دست می‌آورد. شبه‌کد زیر فرض می‌کند که بُعد nام در خصیصه‌ی $L.rows$ وجود دارد و ماتریس جایگشت P در آرایه‌ی π.

```
LUP-SOLVE(L, U, π, b)
1  n = L.rows
2  let x be a new vector of length n
3  for i = 1 to n
4       y_i = b_{π[i]} − Σ_{j=1}^{i−1} l_ij y_i
5  for i = n downto 1
6       x_i = (y_i − Σ_{j=i+1}^{n} u_ij x_j) / u_ii
7  return x
```

رویه‌ی LUP-SOLVE در خطوط ۳-۴ با استفاده از جایگزینی جلویی y را بـه دسـت مـی‌آورد، و سپس در خطوط ۵-۶ با استفاده از جایگزینی عقبی، x را. از آن جایی که در سری‌هـا در هـر یـک از حلقه‌های for یک حلقه‌ی ضمنی وجود دارد، زمان اجرا برابر است با $\theta(n^2)$.

به عنوان یک مثال از این متدها، سیستم معادلات خطی توصیف شده توسط

$$\begin{pmatrix} ۱ & ۲ & ۰ \\ ۳ & ۴ & ۴ \\ ۵ & ۶ & ۳ \end{pmatrix} x = \begin{pmatrix} ۳ \\ ۷ \\ ۸ \end{pmatrix}$$

را در نظر بگیرید، که در آن

$$A = \begin{pmatrix} ۱ & ۲ & ۰ \\ ۳ & ۴ & ۴ \\ ۵ & ۶ & ۳ \end{pmatrix},$$

$$b = \begin{pmatrix} ۳ \\ ۷ \\ ۸ \end{pmatrix}$$

و می‌خواهیم مجهول x را به دست آوریم. تجزیه‌ی LUP به صورت زیر است:

$$L = \begin{pmatrix} ۱ & ۰ & ۰ \\ ۰/۲ & ۱ & ۰ \\ ۰/۶ & ۰/۵ & ۱ \end{pmatrix},$$

$$U = \begin{pmatrix} ۵ & ۶ & ۳ \\ ۰ & ۰/۸ & -۰/۶ \\ ۰ & ۰ & ۲/۵ \end{pmatrix},$$

$$P = \begin{pmatrix} ۰ & ۰ & ۱ \\ ۱ & ۰ & ۰ \\ ۰ & ۱ & ۰ \end{pmatrix}$$

(خواننده می‌تواند چک کند که $PA = LU$.) با استفاده از جایگزینی جلویی $Ly = Pb$ را نسبت بـه y حل می‌کنیم:

$$\begin{pmatrix} ۱ & ۰ & ۰ \\ ۰/۲ & ۱ & ۰ \\ ۰/۶ & ۰/۵ & ۱ \end{pmatrix} \begin{pmatrix} y_۱ \\ y_۲ \\ y_۳ \end{pmatrix} = \begin{pmatrix} ۸ \\ ۳ \\ ۷ \end{pmatrix}$$

که به دست می‌دهد

$$y = \begin{pmatrix} ۸ \\ ۱/۴ \\ ۱/۵ \end{pmatrix}$$

با ابتدا محاسبه‌ی $y_۱$، سپس $y_۲$ و نهایتاً $y_۳$. با استفاده از جایگزینی عقبی، معادله‌ی $Ux = y$ را نسبت به x حل می‌کنیم:

$$\begin{pmatrix} ۵ & ۶ & ۳ \\ ۰ & ۰/۸ & -۰/۶ \\ ۰ & ۰ & ۲/۵ \end{pmatrix} \begin{pmatrix} x_۱ \\ x_۲ \\ x_۳ \end{pmatrix} = \begin{pmatrix} ۸ \\ ۱/۴ \\ ۱/۵ \end{pmatrix}$$

که جواب مورد نظر

$$x = \begin{pmatrix} -۱/۴ \\ ۲/۲ \\ ۰/۶ \end{pmatrix}$$

را به دست می‌دهد، که در آن ابتدا $x_۳$، سپس $x_۲$، و نهایتاً $x_۱$ محاسبه شده است.

محاسبه‌ی تجزیه‌ی LU

اکنون نشان داده‌ایم که اگر بتوان یک تجزیه‌ی LUP برای یک ماتریس غیرتکین A محاسبه کرد، آن گاه می‌توان از جایگزینی‌های جلویی و عقبی برای حل سیستم معادلات خطی $Ax = b$ استفاده کرد. این باقی می‌ماند که نشان دهیم می‌توان به صورت بهینه یک تجزیه‌ی LUP برای A پیدا کرد. با حالتی آغاز می‌کنیم که در آن A یک ماتریس غیرتکین $n \times n$ است و P وجود ندارد (یا به طور معادل، $P = I_n$). در این صورت باید به دنبال یک حالت فاکتور گرفته شده‌ی $A = LU$ بگردیم. به دو ماتریس L و U یک *تجزیه‌ی LU* از ماتریس A می‌گوییم.

به فرآیندی که در این جا به کمک آن تجزیه‌ی LU را انجام می‌دهیم، *حذف گاوسی* (Gaussian elimination) می‌گوییم. با تفریق ضریب‌هایی از تساوی اول از بقیه‌ی تساوی‌ها شروع می‌کنیم، طوری که متغیر اول از آن تساوی‌ها حذف شود. سپس، ضریب‌هایی از تساوی دوم را از تساوی‌های سوم و پایین‌تر کم می‌کنیم تا متغیر دوم هم از آن‌ها حذف شود. این فرآیند را آن قدر ادامه می‌دهیم تا سیستمی که باقی می‌ماند شکل بالا–مثلثی داشته باشد – در واقع، این ماتریس همان ماتریس U است. ماتریس L از ضرایب سطرها ساخته می‌شود که قبلاً باعث شد متغیرها حذف شوند.

الگوریتم ما برای پیاده‌سازی این استراتژی، بازگشتی است. می‌خواهیم یک تجزیه‌ی LU برای یک

ماتریس غیرتکین A با اندازه‌ی $n \times n$ بیابیم. اگر $n = 1$ آن گاه کار تمام است، چرا که می‌توانیم قرار دهیم $L = I_1$ و $U = A$. برای $n > 1$ ماتریس A را به چهار قسمت تقسیم می‌کنیم:

$$A = \left(\begin{array}{c|ccc} a_{11} & a_{12} & \cdots & a_{1n} \\ \hline a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array}\right) = \begin{pmatrix} a_{11} & w^T \\ v & A' \end{pmatrix}$$

که در آن v یک بردار ستونی با اندازه‌ی $n-1$ است، w^T یک بردار سطری با اندزه‌ی $n-1$، و A' یک ماتریس $(n-1)\times(n-1)$. سپس با استفاده از جبر ماتریس‌ها (درستی تساوی‌ها را به سادگی و با ضرب کردن می‌توان چک کرد)، می‌توانیم A را تجزیه کنیم:

$$\begin{aligned} A &= \begin{pmatrix} a_{11} & w^T \\ v & A' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & A' - vw^T/a_{11} \end{pmatrix} \end{aligned} \qquad (۸-۲۸)$$

۰ها در ماتریس اول و دوم تساوی (۸-۲۸) به ترتیب بردارهایی سطری و ستونی با اندازه‌ی $n-1$ هستند. جمله‌ی vw^T/a_{11} که با ضرب خارجی v و w و سپس تقسیم هر یک از عناصر نتیجه به a_{11} به دست می‌آید، یک ماتریس $(n-1)\times(n-1)$ است، که از نظر اندازه با ماتریس A' که باید از آن کم شود، تطابق دارد. ماتریس $(n-1)\times(n-1)$ حاصل

$$A' - vw^T/a_{11} \qquad (۹-۲۸)$$

مکمل Schur ماتریس A نسبت به a_{11} نام دارد.

ادعا می‌کنیم که اگر A غیرتکین باشد، مکمل Schur هم غیرتکین خواهد بود. چرا؟ فرض کنید مکمل Schur، که $(n-1)\times(n-1)$ است، تکین باشد. آن گاه طبق قضیه‌ی ت-۱ رتبه‌ی سطری آن اکیداً کوچک‌تر از $n-1$ است. چون $n-1$ ورودی پایینی در اولین ستون ماتریس

$$\begin{pmatrix} a_{11} & w^T \\ 0 & A' - vw^T/a_{11} \end{pmatrix}$$

همگی ۰ هستند، رتبه‌ی سطری $n-1$ سطر پایینی ماتریس باید اکیداً کوچک‌تر از $n-1$ باشد. بنابراین رتبه‌ی سطری کل ماتریس اکیداً کوچک‌تر از n است. با اعمال تمرین ت-۲-۸ به تساوی (۸-۲۸)، رتبه‌ی A اکیداً کوچک‌تر از n خواهد بود، و از قضیه‌ی ۲۸-۱ به این تناقض می‌رسیم که A تکین است.

چون مکمل Schur غیرتکین است، اکنون می‌توانیم تجزیه‌ی LU آن را بیابیم. اجازه دهید فرض کنیم

$$A' - vw^T / a_{11} = L'U'$$

که در آن L' پایین–مثلثی و یکه است، و U' بالا– مثلثی. سپس با استفاده از جبر ماتریسی خواهیم داشت

$$\begin{aligned} A &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & A' - vw^T/a_{11} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & L'U' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & U' \end{pmatrix} \\ &= LU \end{aligned}$$

که تجزیه‌ی LU را اثبات می‌کند. (توجه کنید که چون L' پایین– مثلثی و یکه است، L هم این گونه خواهد بود، و چون U' بالا– مثلثی است، U هم این گونه خواهد بود.)

مسلماً اگر $a_{11} = 0$ باشد این متد کار نخواهد کرد، چرا که در آن تقسیم بر 0 انجام می‌شود. همچنین اگر ورودی بالا و چپ مکمل Schur، یعنی $A' - vw^T/a_{11}$ برابر 0 باشد، باز هم این متد کار نخواهد کرد، چرا که در مرحله‌ی بعد بازگشت تقسیم بر آن را انجام خواهیم داد. عناصری که در طول محاسبه‌ی تجزیه‌ی LU بر آن‌ها تقسیم انجام می‌دهیم، *محور* (pivot) نامیده می‌شوند، و عناصر قطری ماتریس U را اشغال می‌کنند. دلیل این که در تجزیه‌ی LUP یک ماتریس جایگشت P را هم وارد کار می‌کنیم، این است که از تقسیم بر عناصر صفر دوری کنیم. استفاده از جایگشت‌ها برای دوری از تقسیم بر 0 (یا تقسیم بر اعداد کوچک، که به ناپایداری ختم می‌شود) *محورگیری* (pivoting) نام دارد.

کلاس مهمی از ماتریس‌ها که تجزیه‌ی LU همیشه برای آن‌ها به درستی عمل می‌کند، کلاس ماتریس‌های مطلقاً مثبت متقارن است. چنین ماتریس‌هایی نیازی به محورگیری ندارند، و بنابراین می‌توان برای آن‌ها از استراتژی بازگشتی که در بالا توصیف شد، بدون ترس از تقسیم بر 0 استفاده کرد. این نتیجه را، به همراه نتایج زیاد دیگر در بخش ۲۸–۵ اثبات خواهیم کرد.

کد ما برای محاسبه‌ی تجزیه‌ی LU یک ماتریس A از استراتژی بازگشتی پیروی می‌کند، غیر از این که یک حلقه‌ی تکرار جایگزین بازگشت می‌شود. (این تبدیل یک بهینه‌سازی استاندارد برای یک رویه‌ی «بازگشتی دنباله‌ای» است – رویه‌ای که آخرین عملیات آن، فراخوانی خود رویه است. مسئله‌ی ۷–۴ را ببینید.) در این کد فرض می‌شود که بُعدهای A در خصیصه‌ی $A.rows$ نگه‌داری می‌شود. ماتریس U را با 0 در مکان‌های زیر قطر اصلی، و ماتریس L را با 1 در قطر اصلی و 0 در مکان‌های بالای قطر اصلی، مقداردهی اولیه می‌کنیم.

```
LU-DECOMPOSITION(A)
```

```
1   n = A.rows
2   let L and U be new n × n matrices
3   initialize U with 0s below the diagonal
4   initialize L with 1s on the diagonal and 0s above the diagonal
5   for k = 1 to n
6       u_kk = a_kk
7       for i = k + 1 to n
8           l_ik = a_ik / u_kk          // l_ik holds v_i
9           u_ki = a_ki                 // u_ki holds w_i^T
10      for i = k + 1 to n
11          for j = k + 1 to n
12              a_ij = a_ij − l_ik u_kj
13  return L and U
```

حلقه‌ی for خارجی که در خط ۲ آغاز می‌شود، برای هر مرحله‌ی بازگشت یک بار تکرار می‌شود. درون این حلقه در خط ۳ محور $u_{kk} = a_{kk}$ تعیین می‌شود. حلقه‌ی for خطوط ۷-۹ (که وقتی $k = n$ اجرا نمی‌شود)، از بردارهای v و w^T برای به هنگام سازی L و U استفاده می‌کند. عناصر بردار v خط در ۸ تعیین می‌شوند، و v_i در l_{ik} ذخیره می‌شود. همچنین عناصر بردار w^T در خط ۹ تعیین می‌شوند، که w_i^T در u_{ki} ذخیره می‌شود. در نهایت عناصر مکمل Schur در خطوط ۱۰-۱۲ محاسبه و در ماتریس A ذخیره می‌شوند. (احتیاجی نداریم که در خط ۱۲ بر a_{kk} تقسیم کنیم، چرا که قبلاً در خط ۸ وقتی که l_{ik} را محاسبه کردیم، این کار را انجام دادیم.) چون در خط ۱۲ سه فرورفتگی داریم، LU-DECOMPOSITION در زمان $\theta(n^3)$ اجرا می‌شود.

شکل ۲۸-۱ عملیات LU-DECOMPOSITION را مشخص می‌کند. در این شکل یک بهینه‌سازی استاندارد رویه نشان داده شده است که در آن عناصر مهم L و U به صورت «درجا» در ماتریس A ذخیره شده‌اند. یعنی می‌توانیم یک تناظر میان هر عنصر a_{ij} و l_{ij} (اگر $i > j$) یا u_{ij} (اگر $i \le j$) برقرار کنیم، و ماتریس A را طوری به هنگام سازی کنیم که وقتی رویه پایان می‌یابد، هر دوی L و U را در خود نگه دارد. شبه‌کد این بهینه‌سازی فقط با جایگزینی هر یک از اشاره‌گرهای l و u با a به دست می‌آید؛ چک کردن این که این تغییر درستی رویه را حفظ می‌کند، کار مشکلی نیست.

محاسبه‌ی تجزیه‌ی LUP

به طور کلی در حل سیستم‌های معادلات خطی $Ax = b$، باید عناصری را به عنوان محور در نظر بگیریم که روی قطر اصلی نیستند تا از تقسیم بر ۰ جلوگیری کنیم. نه تنها تقسیم بر ۰، بلکه تقسیم بر اعداد بسیار کوچک هم نامطلوب است، حتی اگر A غیرتکین باشد، چرا که ممکن است در تجزیه ناپایداری عددی به وجود بیاید. بنابراین سعی می‌کنیم اعداد بزرگ را به عنوان محور در نظر بگیریم.

ریاضیات پشت تجزیه‌ی LUP مانند تجزیه‌ی LU است. به خاطر بیاورید که به ما یک ماتریس غیرتکین A با اندازه‌ی $n \times n$ داده شده است، و می‌خواهیم یک ماتریس جایگشت P، یک ماتریس پایین- مثلثی L، و یک ماتریس بالا- مثلثی U بیابیم به طوری که $PA = LU$. قبل از این که ماتریس

A را تقسیم‌بندی کنیم، همان طور که برای تجزیه‌ی LU انجام دادیم، یک عنصر غیر صفر، مثلاً a_{k1} را از جایی در ستون اول به مکان (۱, ۱) ماتریس جابه‌جا می‌کنیم. برای پایداری عددی، a_{k1} را به عنوان عنصری در ستون اول انتخاب می‌کنیم که بیشترین مقدار مطلق را دارد. (اولین ستون ماتریس نمی‌تواند فقط حاوی ۰ باشد، چرا که در این صورت A تکین خواهد بود، چون طبق قضیه‌های ت-۴ و ت-۵ دترمینان آن ۰ است.) برای این که مجموعه‌ی معادلات را حفظ کنیم، جای سطر ۱ را با سطر k عوض می‌کنیم، که معادل است با ضرب A در ماتریس جایگشت Q در سمت چپ (تمرین ت-۱-۴). بنابراین می‌توانیم QA را به صورت

$$QA = \begin{pmatrix} a_{k1} & w^T \\ v & A' \end{pmatrix}$$

بنویسیم، که در آن $v = (a_{21}, a_{31}, ..., a_{n1})^T$، غیر از این که a_{11} جای a_{k1} را می‌گیرد؛ $w^T = (a_{k2}, a_{k3}, ..., a_{kn})$؛ و A' یک ماتریس $(n-1)\times(n-1)$ است. چون $a_{k1} \neq 0$، اکنون می‌توانیم جبری مانند جبر انجام شده در تجزیه‌ی LU انجام دهیم، ولی با این تفاوت که در این جا عدم تقسیم بر ۰ تضمین شده است

2	3	1	5
6	13	5	19
2	19	10	23
4	10	11	31

(الف)

2	3	1	5
3	4	2	4
1	16	9	18
2	4	9	21

(ب)

2	3	1	
3	**4**	2	4
1	4	1	2
2	1	7	17

(پ)

2	3	1	5
3	4	2	4
1	4	**1**	2
2	1	7	3

(ت)

$$\begin{pmatrix} 2 & 3 & 1 & 5 \\ 6 & 13 & 5 & 19 \\ 2 & 19 & 10 & 23 \\ 4 & 10 & 11 & 31 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 2 & 1 & 7 & 1 \end{pmatrix} \begin{pmatrix} 2 & 3 & 1 & 5 \\ 0 & 4 & 2 & 4 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 3 \end{pmatrix}$$

A L U

(ث)

شکل ۲۸-۱ عملیات LU-DECOMPOSITION. (الف) ماتریس A. (ب) عنصر $a_{11} = 2$ در دایره‌ی تیره، محور است، ستون سایه‌دار v/a_{11}، و سطر سایه‌دار w^T است. عناصر U که تا به این جا محاسبه شده‌اند، بالای خط افقی هستند، و عناصر L که تا به این جا محاسبه شده‌اند، در سمت چپ خط عمودی قرار دارند. ماتریس مکمل Schur، $A' - vw^T/a_{11}$، سمت چپ و پایین را اشغال کرده است. (پ) اکنون بر روی ماتریس مکمل Schur ساخته شده در بخش (ب) عمل می‌کنیم. عنصر $a_{22} = 4$ در دایره‌ی تیره، محور است، و ستون و سطر سایه‌دار، به ترتیب v/a_{22} و w^T (در تقسیم بندی مکمل Schur) هستند. خطوط، ماتریس را به عناصر محاسبه شده‌ی U (بالا)، عناصر محاسبه شده‌ی L (چپ)، و مکمل Schur جدید (پایین و راست) تقسیم می‌کنند. (ت) مرحله‌ی بعد، فاکتورگیری را کامل می‌کند. (عنصر ۳ در مکمل Schur جدید، پس از پایان بازگشت به قسمتی از U تبدیل می‌شود.) (ث) فاکتورگیری $A = LU$.

$$QA = \begin{pmatrix} a_{k1} & w^T \\ v & A' \end{pmatrix}$$
$$= \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & A' - vw^T/a_{k1} \end{pmatrix}$$

همان طور که برای تجزیه‌ی LU دیدیم، اگر A غیرتکین باشد مکمل Schur ماتریس $A' - vw^T/a_{k1}$ هم غیرتکین خواهد بود. بنابراین می‌توانیم به صورت استقرایی یک تجزیه‌ی LUP برای آن بیابیم، که در آن ماتریس L' پایین- مثلثی و یکه، U' بالا- مثلثی، و P' یک ماتریس جایگشت است، به طوری که

$$P'(A' - vw^T/a_{k1}) = L'U'$$

تعریف می‌کنیم

$$P = \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} Q$$

که یک ماتریس جایگشت است، چرا که برابر است با ضرب دو ماتریس جایگشت دیگر (تمرین ت-۱-۴). اکنون داریم

$$PA = \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} QA$$
$$= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & A' - vw^T/a_{k1} \end{pmatrix}$$
$$= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & P' \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & A' - vw^T/a_{k1} \end{pmatrix}$$
$$= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & P'(A' - vw^T/a_{k1}) \end{pmatrix}$$
$$= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & L'U' \end{pmatrix}$$
$$= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & L' \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & U' \end{pmatrix}$$
$$= LU$$

که تجزیه‌ی LUP را نتیجه می‌دهد. چون L' پایین- مثلثی و یکه است L هم همین گونه خواهد بود، و چون U' بالا مثلثی است U هم این گونه است.

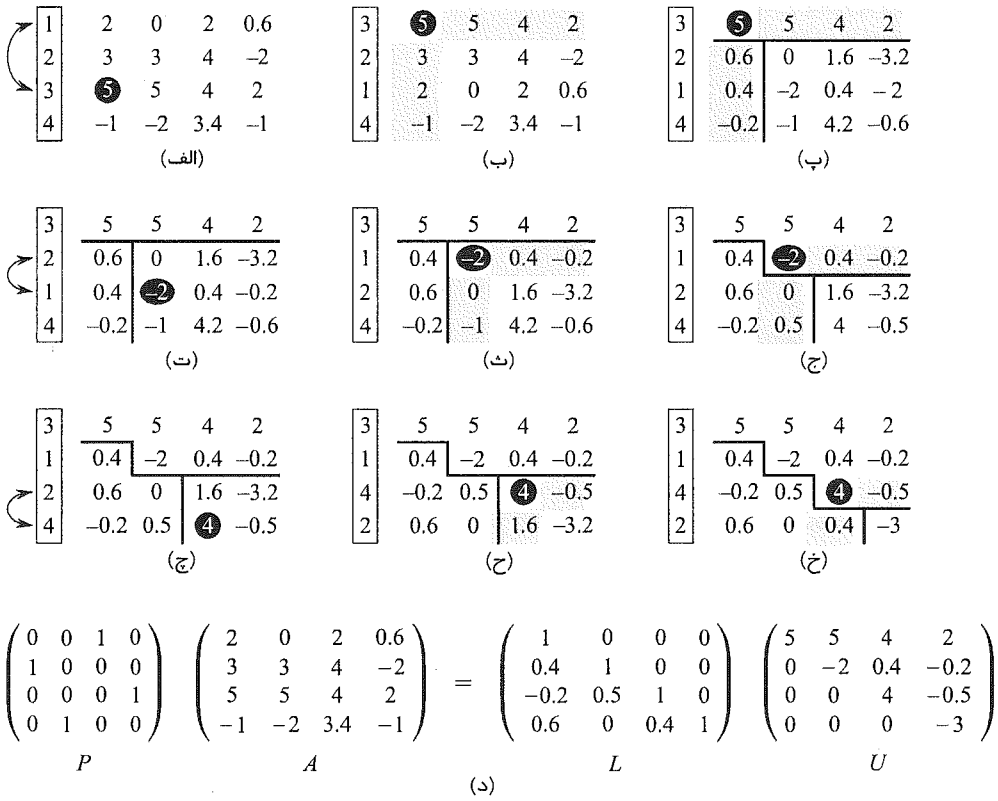
توجه کنید که در این نتیجه‌گیری، برخلاف تجزیه‌ی LU، هم بردار ستونی v/a_{k1} و هم مکمل Schur ماتریس $A' - vw^T/a_{k1}$ باید در ماتریس جایگشت P' ضرب شوند. در زیر شبه‌کد تجزیه‌ی LUP را می‌بینیم:

```
LUP-DECOMPOSITION(A)
1  n = A.rows
2  let π[1..n] be a new array
3  for i = 1 to n
4      π[i] = i
5  for k = 1 to n
6      p = 0
7      for i = k to n
8          if |a_ik| > p
9              p = |a_ik|
10             k' = i
11     if p == 0
12         error "singular matrix"
13     exchange π[k] with π[k']
14     for i = 1 to n
15         exchange a_ki with a_k'i
16     for i = k + 1 to n
17         a_ik = a_ik / a_kk
18         for j = k + 1 to n
19             a_ij = a_ij – a_ik a_kj
```

مانند LU-DECOMPOSITION، شبه‌کد تجزیه‌ی LUP بازگشت را با یک حلقه‌ی تکرار جایگزین می‌کند. به عنوان یک پیشرفت نسبت به پیاده‌سازی مستقیم بازگشت، به صورت پویا ماتریس جایگشت P را به صورت یک آرایه‌ی π نگه می‌داریم، که در آن $\pi[i] = j$، یعنی iامین سطر از P حاوی یک ۱ در ستون j است. همچنین کد را طوری پیاده‌سازی می‌کنیم که محاسبه‌ی L و U را در ماتریس A به صورت «درجا» انجام دهد. بنابراین وقتی رویه پایان می‌یابد داریم:

$$a_{ij} = \begin{cases} l_{ij} & \text{اگر } i > j \\ u_{ij} & \text{اگر } i \le j \end{cases}$$

شکل ۲۸-۲ نشان می‌دهد که LUP-DECOMPOSITION چگونه از یک ماتریس فاکتور می‌گیرد. آرایه‌ی π در خطوط ۳-۴ مقداردهی اولیه می‌شود تا جایگشت همانی را نشان دهد. حلقه‌ی for خارجی که در خط ۵ آغاز می‌شود، بازگشت را پیاده‌سازی می‌کند. با هر بار عبور از روی حلقه‌ی خارجی، خطوط ۶-۱۰ عنصر $a_{k'k}$ را با بزرگ‌ترین مقدار قدرمطلق از میان اولین ستون فعلی (ستون k) از ماتریس $(n-k+1)\times(n-k+1)$ که می‌خواهیم تجزیه‌ی LUP آن را بیابیم، مقداردهی می‌کند. اگر تمام عناصر در اولین ستون فعلی صفر باشند، خطوط ۱۱-۱۲ گزارش می‌دهند که ماتریس تکین است. برای محورگیری، در خط ۱۳ جای $\pi[k]$ و $\pi[k']$ را با هم عوض می‌کنیم، و سپس در خطوط ۱۴-۱۵، جای سطرهای k و k' را در A عوض می‌کنیم، که عنصر a_{kk} را به محور تبدیل می‌کند. (تمام سطرها تعویض می‌شوند چرا که در استنتاج متد بالا، نه تنها $A' - vw^T/a_{k1}$، بلکه v/a_{k1} هم در P' ضرب می‌شود.) در نهایت مکمل Schur در خطوط ۱۶-۱۹ به همان شکلی محاسبه می‌شود که در خطوط ۷-۱۲ رویه‌ی LU-DECOMPOSITION محاسبه شد، با این تفاوت که در این جا عملیات طوری نوشته شده است که «درجا» عمل کند.



شکل ۲۸-۲ عملیات LUP-DECOMPOSITION. (الف) ماتریس ورودی A به همراه جایگشت همانی سطرها در سمت چپ. اولین مرحله‌ی الگوریتم تعیین می‌کند که عنصر ۵ در دایره‌ی تیره در ردیف سوم، محور ستون است. (ب) جای ردیف‌های ۱ و ۳ با هم عوض، و جایگشت به هنگام سازی می‌شود. ستون و سطر سایه‌دار نشان‌دهنده‌ی v و w^T هستند. (پ) بردار v با $v/5$ جایگزین شده است، و سمت راست و پایین ماتریس با مکمل Schur به هنگام سازی شده است. خطوط ماتریس را به سه محدوده تقسیم می‌کند: عناصر U (بالا)، عناصر L (چپ)، و عناصر مکمل Schur (راست و پایین). (ت)-(ج) مرحله‌ی دوم. (چ)-(خ) مرحله‌ی سوم. هیچ تغییر دیگری در مرحله‌های چهارم و آخر اتفاق نمی‌افتد. (د) تجزیه‌ی LUP، $PA = LU$.

زمان اجرای LUP-DECOMPOSITION به خاطر ساختار حلقه‌ی سه‌تایی آن، $\theta(n^3)$ است، که برابر است با زمان اجرای LU-DECOMPOSITION. بنابراین هزینه‌ی محورگیری حداکثر ضریب ثابتی در زمان اجرا خواهد بود.

تمرین‌ها

۲۸-۱-۱ معادله‌ی

$$\begin{pmatrix} ۱ & ۰ & ۰ \\ ۴ & ۱ & ۰ \\ -۶ & ۵ & ۱ \end{pmatrix} \begin{pmatrix} x_۱ \\ x_۲ \\ x_۳ \end{pmatrix} = \begin{pmatrix} ۳ \\ ۱۴ \\ -۷ \end{pmatrix}$$

را با استفاده از جایگزینی جلویی حل کنید.

۲۸-۱-۲ یک تجزیه‌ی LU برای ماتریس

$$\begin{pmatrix} ۴ & -۵ & ۶ \\ ۸ & -۶ & ۷ \\ ۱۲ & -۷ & ۱۲ \end{pmatrix}$$

بیابید.

۲۸-۱-۳ معادله‌ی

$$\begin{pmatrix} ۱ & ۵ & ۴ \\ ۲ & ۰ & ۳ \\ ۵ & ۸ & ۲ \end{pmatrix} \begin{pmatrix} x_۱ \\ x_۲ \\ x_۳ \end{pmatrix} = \begin{pmatrix} ۱۲ \\ ۹ \\ ۵ \end{pmatrix}$$

را با استفاده از تجزیه‌ی LUP حل کنید.

۲۸-۱-۴ تجزیه‌ی LUP یک ماتریس قطری را توصیف کنید.

۲۸-۱-۵ تجزیه‌ی LUP یک ماتریس جایگشت را توصیف کنید، و اثبات کنید که این تجزیه یکتا است.

۲۸-۱-۶ نشان دهید که برای هر $n \geq ۱$، یک ماتریس تکین $n \times n$ وجود دارد که دارای تجزیه‌ی LU است.

۲۸-۱-۷ در LU-DECOMPOSITION آیا وقتی داریم $k = n$، اجرای خارجی‌ترین حلقه‌ی for ضروری است؟ در مورد LUP-DECOMPOSITION چطور؟

۲۸-۲ معکوس کردن ماتریس‌ها

در عمل برای حل سیستم‌های معادلات خطی معمولاً معکوس ماتریس‌ها را محاسبه نمی‌کنیم و ترجیح می‌دهیم در عوض از تکنیک‌هایی که پایداری عددی بیشتری دارند، مانند تجزیه‌ی LUP، استفاده کنیم. با این حال بعضی مواقع محاسبه‌ی معکوس یک ماتریس ضروری است. در این بخش نشان می‌دهیم که چطور می‌توان از تجزیه‌ی LUP برای محاسبه‌ی معکوس یک ماتریس استفاده کرد. همچنین اثبات خواهیم کرد که سختی مسائل ضرب ماتریس‌ها و محاسبه‌ی معکوس آن‌ها با یکدیگر برابر است، به

طوری که (در مباحث تکنیکی) می‌توانیم از یکی از الگوریتم‌ها برای حل مسئله‌ی دیگر با همان زمان اجرا استفاده کنیم. بنابراین می‌توانیم از الگوریتم استراسن (بخش ۴-۲) برای ضرب ماتریس‌ها، برای معکوس کردن یک ماتریس هم استفاده کنیم. در واقع مقاله‌ی اصلی استراسن بر مبنای این مسئله پایه‌ریزی شده بود: اثبات این که می‌توان مجموعه‌ای از معادلات خطی را سریع‌تر از متد معمولی حل کرد.

محاسبه‌ی معکوس یک ماتریس از یک تجزیه‌ی LUP

فرض کنید تجزیه‌ی LUP یک ماتریس A را به صورت سه ماتریس L، U، و P داریم، به طوری که $PA = LU$. با استفاده از رویه‌ی LUP-SOLVE می‌توانیم یک تساوی به شکل $Ax = b$ را در زمان $\theta(n^2)$ حل کنیم. از آن جایی که تجزیه‌ی LUP به A وابسته است، و نه به b، می‌توانیم LUP-SOLVE را بر روی مجموعه‌ی دومی از معادلات به شکل $Ax = b'$، در زمان اضافی $\theta(n^2)$ حل کنیم. به طور کلی وقتی تجزیه‌ی LUP یک ماتریس A را داشته باشیم، می‌توانیم k نسخه‌ی مختلف از تساوی $Ax = b$ را که فقط در b با هم تفاوت دارند، در زمان $\theta(kn^2)$ محاسبه کنیم.

می‌توان تساوی

$$AX = I_n \tag{۲۸-۱۰}$$

را، که ماتریس X (معکوس A) را تعریف می‌کند، به صورت مجموعه‌ای از n تساوی مجزا به شکل $Ax = b$ دید. این تساوی‌ها ماتریس X را به صورت معکوس A تعریف می‌کنند. به صورت دقیق، فرض کنید X_i نشان‌دهنده‌ی iامین ستون X باشد، و به یاد بیاورید که بردار یکه‌ی e_i، iامین ستون I_n است. تساوی (۲۸-۱۰) را می‌توان با استفاده از تجزیه‌ی LUP ماتریس A نسبت به X حل کرد، که معادله‌ی

$$AX_i = e_i$$

را به صورت جداگانه برای X_i حل می‌کند. اگر تجزیه‌ی LUP را داشته باشیم می‌توانیم هر یک از n ستون X_i را در زمان $\theta(n^2)$ بیابیم، و بنابراین محاسبه‌ی X از تجزیه‌ی LUP ماتریس A به $\theta(n^3)$ زمان نیاز دارد. از آن جایی که تجزیه‌ی LUP ماتریس A را می‌توان در زمان $\theta(n^3)$ محاسبه کرد، ماتریس معکوس A، یعنی A^{-1} را می‌توان در زمان $\theta(n^3)$ تعیین کرد.

ضرب و معکوس ماتریس‌ها

اکنون نشان می‌دهیم که تسریع‌های نظری به دست آمده برای ضرب ماتریس‌ها به تسریع‌هایی برای یافتن معکوس ماتریس‌ها تبدیل می‌شوند. در واقع قضیه‌ی قوی‌تری را اثبات می‌کنیم: محاسبه‌ی معکوس ماتریس‌ها معادل است با مسئله‌ی ضرب ماتریس‌ها. برای این کار نشان می‌دهیم که اگر $M(n)$ نشان‌دهنده‌ی زمان مورد نیاز برای ضرب دو ماتریس $n \times n$ باشد، آن گاه روشی برای معکوس کردن یک ماتریس $n \times n$ در زمان $O(M(n))$ وجود دارد. علاوه بر این اگر $I(n)$ نشان‌دهنده‌ی زمان مورد نیاز برای معکوس کردن یک ماتریس غیرتکین $n \times n$ باشد، آن گاه یک روش

برای ضرب دو ماتریس $n \times n$ در زمان $O(I(n))$ وجود دارد. این نتایج را به صورت دو قضیه‌ی مجزا اثبات می‌کنیم.

قضیه‌ی ۲۸.۱
(اگر مسئله‌ی ضرب ماتریس‌ها سخت‌تر از محاسبه‌ی معکوس ماتریس‌ها نیست)

اگر بتوانیم معکوس یک ماتریس $n \times n$ را در زمان $I(n)$ محاسبه کنیم، که در آن $I(n) = \Omega(n^2)$ و $I(n)$ شرط ترتیبی $I(3n) = O(I(n))$ را ارضا می‌کند، آن گاه می‌توانیم دو ماتریس $n \times n$ را در زمان $O(I(n))$ در هم ضرب کنیم.

اثبات فرض کنید A و B دو ماتریس $n \times n$ باشند که محاسبه‌ی ماتریس حاصل ضرب آن‌ها، یعنی C مورد نظر ما است. ماتریس D را با اندازه‌ی $3n \times 3n$ به صورت زیر تعریف می‌کنیم:

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}$$

معکوس ماتریس D، ماتریس

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix}$$

است، و بنابراین می‌توانیم ضرب AB را با در نظر گرفتن زیرماتریس $n \times n$ بالا و راست ماتریس D^{-1} به دست آوریم.

می‌توانیم ماتریس D را در زمان $\theta(n^2)$ بسازیم، که برابر است با $O(I(n))$ چرا که فرض می‌کنیم $I(n) = \Omega(n^2)$، و سپس طبق شرط ترتیبی روی $I(n)$ می‌توانیم D را در زمان $O(I(3n)) = O(I(n))$ معکوس کنیم. بنابراین داریم $M(n) = O(I(n))$. ■

توجه کنید که اگر برای ثابت‌های $c > 0$ و $d \geq 0$ داشته باشیم $I(n) = \theta(n^c \lg^d n)$، آن گاه $I(n)$ شرط ترتیبی را ارضا می‌کند.

اثبات این که محاسبه‌ی معکوس ماتریس سخت‌تر از ضرب ماتریس‌ها نیست به بعضی از خصوصیات ماتریس‌های مطلقاً مثبت و متقارن وابسته است، که در بخش ۲۸-۳ اثبات می‌شوند.

قضیه‌ی ۲۸.۲
(محاسبه‌ی معکوس ماتریس‌ها سخت‌تر از ضرب آن‌ها نیست)

فرض کنید می‌توانیم دو ماتریس $n \times n$ با مقادیر حقیقی را در زمان $M(n)$ در یکدیگر ضرب کنیم، که در آن $M(n) = \Omega(n^2)$ و $M(n)$ دو شرط ترتیبی $M(n+k) = O(M(n))$ برای هر k در بازه‌ی $0 \leq k \leq n$ و $M(n/2) \leq cM(n)$ برای یک ثابت $c < 1/2$ را ارضا می‌کند. در این صورت می‌توانیم معکوس هر ماتریس حقیقی مقدار غیرتکین $n \times n$ را در زمان $O(M(n))$ محاسبه کنیم.

اثبات در این جا این قضیه را برای ماتریس‌های حقیقی اثبات می‌کنیم. تمرین ۲۸-۲-۶ از شما می‌خواهد اثبات را برای ماتریس‌هایی که ورودی‌های آن‌ها اعداد مختلط هستند، گسترش دهید.

می‌توانیم فرض کنیم n توانی از ۲ است، چرا که داریم

$$\begin{pmatrix} A & 0 \\ 0 & I_k \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & I_k \end{pmatrix}$$

برای هر $k > 0$. بنابراین با انتخاب k به طوری که $n+k$ توانی از ۲ باشد، ماتریس را تا توان بعدی ۲ گسترش می‌دهیم و جواب مورد نظر A^{-1} را از جواب مسئله‌ی گسترش داده شده به دست می‌آوریم. اولین شرط ترتیبی بر روی $M(n)$ اطمینان می‌دهد که این گسترش زمان اجرا را بیشتر از یک ضریب ثابت افزایش نمی‌دهد.

فعلاً اجازه دهید فرض کنیم که ماتریس A متقارن و مطلقاً مثبت است. A و معکوس آن A^{-1} را به چهار زیرماتریس $n/2 \times n/2$ تقسیم می‌کنیم:

$$A = \begin{pmatrix} B & C^T \\ C & D \end{pmatrix}, \quad A^{-1} = \begin{pmatrix} R & T \\ U & V \end{pmatrix} \tag{۲۸-۱۱}$$

آن گاه اگر فرض کنیم

$$S = D - CB^{-1}C^T \tag{۲۸-۱۲}$$

یک مکمل Schur ماتریس A نسبت به B باشد (در مورد این نوع از مکمل Schur در بخش ۲۸-۳ بیشتر خواهیم آموخت)، داریم

$$A^{-1} = \begin{pmatrix} B^{-1} + B^{-1}C^T S^{-1} CB^{-1} & -B^{-1}C^T S^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix} \tag{۲۸-۱۳}$$

چرا که $AA^{-1} = I_n$، که می‌توان صحت آن را با ضرب ماتریسی چک کرد. چون A متقارن و مطلقاً مثبت است، لم‌های ۲۸-۴ و ۲۸-۵ در بخش ۲۸-۳ ایجاب می‌کنند که هر دوی B و S متقارن و مطلقاً مثبت باشند. بنابراین طبق لم ۲۸-۳ در بخش ۲۸-۳، معکوس‌های B^{-1} و S^{-1} وجود دارند، و طبق تمرین ت-۲-۶، ماتریس‌های B^{-1} و S^{-1} متقارن هستند، و از این رو داریم $(B^{-1})^T = B^{-1}$ و $(S^{-1})^T = S^{-1}$. بنابراین می‌توانیم زیرماتریس‌های R، T، U، و V مربوط به A^{-1} را به صورت زیر محاسبه کنیم، که در آن تمام ماتریس‌های مورد بحث $n/2 \times n/2$ هستند:

۱. تشکیل زیرماتریس‌های B، C، C^T، و D مربوط به A.
۲. محاسبه‌ی بازگشتی B^{-1}، معکوس B.
۳. محاسبه‌ی ضرب ماتریسی $W = CB^{-1}$، و سپس محاسبه‌ی ماتریس ترانهاده‌ی آن، W^T، که برابر است با $B^{-1}C^T$ (طبق تمرین ت-۱-۲ و این که $(B^{-1})^T = B^{-1}$).
۴. محاسبه‌ی ضرب ماتریسی $X = WC^T$، که برابر است با $CB^{-1}C^T$، و سپس محاسبه‌ی ماتریس

$$\delta_f(s,u) \geq \delta_f(s,u) \qquad (۱۳-۲۶)$$

ادعا می‌کنیم که $(u,v) \notin E_f$. چرا؟ اگر داشته باشیم $(u,v) \in E$، آن گاه همچنین خواهیم داشت

$$\begin{aligned}
\delta_f(s,v) &\leq \delta_f(s,u)+1 && \text{(طبق لم ۲۴-۱۰ و نامساوی مثلث)}\\
&\leq \delta_f(s,u)+1 && \text{(طبق نامساوی (۲۶-۱۳))}\\
&= \delta_f(s,v) && \text{(طبق تساوی (۲۶-۱۲))}
\end{aligned}$$

که با فرض $\delta_f(s,v) < \delta_f(s,v)$ تناقض دارد.

چطور می‌توانیم هم داشته باشیم $(u,v) \notin E_f$ و هم $(u,v) \in E_f$؟ تکمیل باید شار را از v به u افزایش داده باشد. الگوریتم ادمونسون-کارپ همیشه شار را در طول کوتاه‌ترین مسیرها تکمیل می‌کند، و بنابراین آخرین یال کوتاه‌ترین مسیر از s به u در G_f یال (v,u) است. پس،

$$\begin{aligned}
\delta_f(s,v) &= \delta_f(s,u)-1\\
&\leq \delta_f(s,u)-2 && \text{(طبق نامساوی (۲۶-۱۳))}\\
&= \delta_f(s,v)-2 && \text{(طبق تساوی (۲۶-۱۲))}
\end{aligned}$$

که با فرض $\delta_f(s,v) < \delta_f(s,v)$ تناقض دارد. نتیجه می‌گیریم که فرض ما که یک رأس v وجود دارد صحیح نیست. ■

قضیه‌ی بعد تعداد تکرارهای الگوریتم ادمونسون-کارپ را محدود می‌کند.

قضیه‌ی ۲۶-۸ اگر الگوریتم ادمونسون-کارپ بر روی یک شبکه‌ی شار $G=(V,E)$ با منبع s و چاهک t اجرا شود، آن گاه کل تعداد تکمیل‌های انجام شده بر روی شار توسط الگوریتم، $O(VE)$ است.

اثبات می‌گوییم یک یال (u,v) در یک شبکه‌ی پس‌ماند G_f بر روی مسیر تکمیلی p *بحرانی* (critical) است اگر ظرفیت پس‌ماند p برابر ظرفیت پس‌ماند (u,v) باشد، یعنی اگر $c_f(p)=c_f(u,v)$. پس از این که شار را بر روی یک مسیر تکمیلی افزایش دادیم، هر یال بحرانی روی مسیر از شبکه‌ی پس‌ماند محو می‌شود. به علاوه حداقل یک یال روی هر مسیر تکمیلی باید بحرانی باشد. نشان خواهیم داد که هر یک از $|E|$ یال می‌تواند حداکثر $\frac{1}{2}|V|-1$ بار بحرانی شود.

فرض کنید u و v رأس‌هایی در V باشند که با یک یال در E به هم متصل شده‌اند. از آن جایی که مسیرهای تکمیلی کوتاه‌ترین مسیر هستند، وقتی (u,v) برای اولین بار بحرانی شود، داریم

$$\delta_f(u,v) = \delta_f(s,u)+1$$

وقتی شار تکمیل شد، یال (u,v) از شبکه‌ی پس‌ماند حذف می‌شود. این یال دیگر نمی‌تواند در مسیر تکمیلی دیگری ظاهر شود تا وقتی که شار از u به v کاهش یابد، که فقط در صورتی اتفاق می‌افتد که (u,v) بر روی یک مسیر تکمیلی ظاهر شود. اگر f' شار شبکه‌ی G برای زمانی باشد که این اتفاق رخ دهد، آن گاه داریم

اثبات قضیه‌ی ۲۸-۲ یک روش پیشنهاد می‌کند برای حل معادله‌ی $Ax = b$ با استفاده از تجزیه‌ی LU بدون استفاده از محورگیری، البته تا زمانی که A غیرتکین باشد. هر دو طرف تساوی را در A^T ضرب می‌کنیم، که می‌دهد $(A^T A)x = A^T b$. این تبدیل بر روی جواب x تأثیری نمی‌گذارد، چرا که A^T معکوس پذیر است، و بنابراین می‌توانیم از ماتریس متقارن و مطلقاً مثبت $A^T A$ با محاسبه‌ی یک تجزیه‌ی LU فاکتور بگیریم. با این که این متد به صورت تئوری درست است، در عمل رویه‌ی LUP-DECOMPOSITION بسیار بهتر کار می‌کند. تعداد اعمال ریاضی که تجزیه‌ی LUP نیاز دارد به اندازه‌ی یک ضریب ثابت کم‌تر است، و تا حدودی خصوصیات عددی بهتری دارد.

تمرین‌ها

۲۸-۲-۱ فرض کنید $M(n)$ زمان موردنیاز برای ضرب دو ماتریس $n \times n$ باشد، و $S(n)$ نشان‌دهنده‌ی زمان مورد نیاز برای مربع گرفتن از یک ماتریس $n \times n$. نشان دهید که سختی ضرب ماتریس‌ها و مربع‌گیری از آن‌ها ذاتاً برابر است: یک الگوریتم ضرب ماتریس با زمان $M(n)$، یک الگوریتم مربع‌گیری ماتریس با زمان $O(M(n))$ را نتیجه می‌دهد، و یک الگوریتم مربع‌گیری ماتریس با زمان $S(n)$، یک الگوریتم ضرب ماتریس با زمان $O(S(n))$ را.

۲۸-۲-۲ فرض کنید $M(n)$ زمان مورد نیاز برای ضرب دو ماتریس $n \times n$ باشد، و $L(n)$ زمان مورد نیاز برای محاسبه‌ی تجزیه‌ی LUP یک ماتریس $n \times n$. نشان دهید که سختی ضرب ماتریس‌ها و محاسبه‌ی تجزیه‌ی LUP ماتریس‌ها ذاتاً یکسان است: یک الگوریتم ضرب ماتریس با زمان $M(n)$، یک الگوریتم محاسبه‌ی تجزیه‌ی LUP با زمان $O(M(n))$ را نتیجه می‌دهد، و یک الگوریتم محاسبه‌ی تجزیه‌ی LUP با زمان $L(n)$، یک الگوریتم ضرب ماتریس با زمان $O(L(n))$ را.

۲۸-۲-۳ فرض کنید $M(n)$ زمان مورد نیاز برای ضرب دو ماتریس $n \times n$ باشد، و $D(n)$ زمان مورد نیاز برای محاسبه‌ی دترمینان یک ماتریس $n \times n$. نشان دهید که سختی ضرب ماتریس‌ها و محاسبه‌ی دترمینان ماتریس‌ها ذاتاً یکسان است: یک الگوریتم ضرب ماتریس با زمان $M(n)$، یک الگوریتم محاسبه‌ی دترمینان با زمان $O(M(n))$ را نتیجه می‌دهد، و یک الگوریتم محاسبه‌ی دترمینان با زمان $D(n)$، یک الگوریتم ضرب ماتریس با زمان $O(D(n))$ را.

۲۸-۲-۴ فرض کنید $M(n)$ زمان مورد نیاز برای ضرب دو ماتریس دودویی $n \times n$ باشد، و $T(n)$ زمان مورد نیاز برای محاسبه‌ی بستار تراگذار یک ماتریس دودویی $n \times n$ (بخش ۲۵-۲ را ببینید). نشان دهید که یک الگوریتم ضرب ماتریس‌های دودویی با زمان $M(n)$، یک الگوریتم محاسبه‌ی بستار تراگذار ماتریس‌های دودویی با زمان $O(M(n))$ را نتیجه می‌دهد، و یک الگوریتم محاسبه‌ی بستار تراگذار ماتریس‌های دودویی با زمان $T(n)$، یک

الگوریتم ضرب ماتریس‌های دودویی با زمان $O(T(n))$ را.

۵-۲-۲۸ آیا الگوریتم محاسبه‌ی معکوس ماتریس داده شده بر مبنای قضیه‌ی ۲۸-۲، وقتی که عناصر از مجموعه‌ی باقی‌مانده‌ی اعداد صحیح بر ۲ هستند هم کار می‌کند؟ توضیح دهید.

★ ۶-۲-۲۸ الگوریتم محاسبه‌ی معکوس ماتریس مربوط به قضیه‌ی ۲۸-۲ را طوری گسترش دهید که بتواند ماتریس‌هایی از اعداد مختلط را هم معکوس کند، و اثبات کنید که گسترش شما به درستی کار می‌کند. (*راهنمایی:* به جای ترانهاده‌ی A، از *ترانهاده‌ی مزدوج* (conjugate transpose) A^* استفاده کنید، که با جایگزینی هر یک از ورودی‌های ترانهاده‌ی A با مزدوج مختلط[1] آن به دست می‌آید. به جای ماتریس‌های متقارن، ماتریس‌های *هرمیتی* (Hermitian matrix) را در نظر بگیرید، که ماتریس‌هایی مانند A هستند به طوری که $A = A^*$.)

۲۸-۳ ماتریس‌های مطلقاً مثبت متقارن و تقریب کم‌ترین مربعات

ماتریس‌های مطلقاً مثبت متقارن خصوصیات جالب و مفید بسیاری دارند. به عنوان مثال آن‌ها غیرتکین هستند، و می‌توانیم بدون نگرانی برای تقسیم بر ۰ تجزیه‌ی LU را بر روی آن‌ها اجرا کنیم. در این بخش خصوصیات مهم دیگری از ماتریس‌های مطلقاً مثبت متقارن را اثبات می‌کنیم، و یک کاربرد جالب توجه خوراندن منحنی را توسط تقریب کم‌ترین مربعات نشان می‌دهیم.

اولین خصوصیتی که اثبات می‌کنیم، احتمالاً پایه‌ای‌ترین آن‌ها هم هست.

لم ۳-۲۸ هر ماتریس مطلقاً مثبت، غیرتکین است.

اثبات فرض کنید یک ماتریس A تکین است. آن گاه طبق نتیجه‌ی ت-۳، یک بردار غیر صفر x وجود دارد به طوری که $Ax = 0$. بنابراین $x^T Ax = 0$، و A نمی‌تواند مطلقاً مثبت باشد. ■

اثبات این که می‌توانیم بدون تقسیم بر ۰، تجزیه‌ی LU را بر روی یک ماتریس مطلقاً مثبت متقارن A اجرا کنیم مقداری پیچیده‌تر است. با اثبات خصوصیاتی در مورد زیرماتریس‌های خاصی از A آغاز می‌کنیم. kامین *زیرماتریس مقدم* (leading submatrix) A را به صورت A_k تعریف می‌کنیم که شامل تقاطع اولین k سطر A با اولین k ستون A است.

لم ۴-۲۸ اگر A یک ماتریس مطلقاً مثبت متقارن باشد، آن گاه تمام زیرماتریس‌های مقدم A متقارن و مطلقاً مثبت هستند.

[1] عدد مختلطی که بخش حقیقی آن برابر است با بخش حقیقی عدد مختلط داده شده، و بخش موهومی آن برابر است با منفی بخش موهومی عدد مختلط داده شده – م

اثبات این که هر زیرماتریس مقدم A_k متقارن است، بدیهی است. برای اثبات این که A_k مطلقاً مثبت است، فرض می‌کنیم که این گونه نیست و به تناقض می‌رسیم. اگر A_k مطلقاً مثبت نباشد، در این صورت یک بردار $x_k \neq 0$ با اندازه‌ی k وجود دارد به طوری که $x_k^T A x_k \leq 0$. فرض کنید A یک ماتریس $n \times n$ باشد، و

$$A = \begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix} \tag{۱۴-۲۸}$$

برای زیرماتریس‌های B (که $(n-k)\times k$ است) و C (که $(n-k)\times(n-k)$ است). بردار $x = (x_k^T \quad 0)^T$ با اندازه‌ی n را تعریف می‌کنیم، که در آن $n-k$ عدد 0 بعد از x_k قرار دارد. در این صورت داریم

$$\begin{aligned} x^T A x &= (x_k^T \quad 0)\begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix}\begin{pmatrix} x_k \\ 0 \end{pmatrix} \\ &= (x_k^T \quad 0)\begin{pmatrix} A_k x_k \\ B x_k \end{pmatrix} \\ &= x_k^T A_k x_k \\ &\leq 0 \end{aligned}$$

که با مطلقاً مثبت بودن A تناقض دارد. ■

اکنون به بعضی از خصوصیات مهم مکمل Schur توجه می‌کنیم. فرض کنید A یک ماتریس مطلقاً مثبت متقارن باشد، و A_k یک زیرماتریس مقدم $k \times k$ از ماتریس A. دوباره A را به صورت تساوی (۱۴-۲۸) تقسیم‌بندی می‌کنیم. تعریف (۹-۲۸) را طوری گسترش می‌دهیم که مکمل Schur ماتریس A را نسبت به A_k به صورت

$$S = C - BA_K^{-1}B^T \tag{۱۵-۲۸}$$

تعریف کند (طبق لم ۴-۲۸، A_k متقارن و مطلقاً مثبت است؛ بنابراین طبق لم ۳-۲۸، A_k^{-1} وجود دارد و S خوش‌تعریف است.) توجه کنید که تعریف قبلی ما (۹-۲۸) از مکمل Schur با تعریف (۱۵-۲۸) سازگار است، با قرار دادن $k = 1$.

لم بعد نشان می‌دهد که مکمل Schur ماتریس‌های مطلقاً مثبت متقارن، خود مطلقاً مثبت و متقارن هستند. از این نتیجه در قضیه‌ی ۲-۲۸ استفاده شده بود، و نتیجه‌ی آن برای اثبات درستی تجزیه‌ی LU برای ماتریس‌های مطلقاً مثبت متقارن لازم است.

لم ۲۸-۵

اگر A یک ماتریس مطلقاً مثبت متقارن باشد، و A_k یک زیرماتریس مقدم A با اندازه‌ی $k \times k$، آن گاه مکمل Schur ماتریس A نسبت به A_k متقارن و مطلقاً مثبت است.

اثبات چون A متقارن است، زیرماتریس C هم همین گونه است. ماتریس‌های حاصل‌ضرب $BA_k^{-1}B^T$، (طبق تمرین ت-۶-۲) و S (طبق تمرین ت-۱-۱) متقارن هستند.

این باقی می‌ماند که نشان دهیم S مطلقاً مثبت است. تقسیم‌بندی A داده شده در تساوی (۲۸-۱۴) را در نظر بگیرید. برای هر بردار غیر صفر x داریم $x^TAx > 0$، طبق این فرض که A مطلقاً مثبت است. اجازه دهید x را به دو زیربردار y و z بشکنیم، که به ترتیب با A_k و C سازگار هستند. چون A_k^{-1} وجود دارد، داریم

$$\begin{aligned} x^TAx &= (y^T \quad z^T)\begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix}\begin{pmatrix} y \\ z \end{pmatrix} \\ &= (y^T \quad z^T)\begin{pmatrix} A_k y & B^T z \\ By & Cz \end{pmatrix} \\ &= y^TA_k y + y^TB^T z + z^TBy + z^TCz \\ &= (y + A_k^{-1}B^T z)^T A_k (y + A_k^{-1}B^T z) + z^T(C - BA_k^{-1}B^T)z \end{aligned} \qquad (۲۸-۳۰)$$

طبق خصوصیات ماتریس‌ها. (با ضرب می‌توانید درستی تساوی بالا را چک کنید.) این تساوی آخر «کامل کردن مربع» را برای معادله‌ی درجه‌ی دوم انجام می‌دهد. (تمرین ۲۸-۳-۲ را ببینید.)

از آن جایی که $x^TAx > 0$ برای هر x غیر صفر برقرار است، اجازه دهید یک z غیر صفر دلخواه انتخاب کنیم و سپس قرار دهیم $y = -A_k^{-1}B^T z$، که باعث می‌شود جمله‌ی اول در تساوی (۲۸-۱۶) از میان برود، که

$$z^T(C - BA_k^{-1}B^T)z = z^TSz$$

را به عنوان مقدار عبارت باقی می‌گذارد. بنابراین برای هر $z \neq 0$ داریم $z^TSz = x^TAx > 0$، و از این رو S مطلقاً مثبت است. ■

نتیجه‌ی ۲۸-۶ تجزیه‌ی LU یک ماتریس مطلقاً مثبت متقارن، هیچ گاه نمی‌تواند باعث شود که یک تقسیم بر ۰ رخ دهد.

اثبات فرض کنید A یک ماتریس مطلقاً مثبت متقارن باشد. چیزی قوی‌تر از حکم داده شده را اثبات خواهیم کرد: تمام محورها اکیداً مثبت هستند. اولین محور a_{11} است . فرض کنید e_1 اولین بردار یکه باشد، که از آن $a_{11} = e_1^TAe_1 > 0$ را به دست می‌آوریم. از آن جایی که اولین مرحله‌ی تجزیه‌ی LU، مکمل Schur ماتریس A را نسبت به $A_1 = (a_{11})$ تولید می‌کند، لم ۲۸-۵ با استقرا ایجاب می‌کند که تمام محورها مثبت هستند. ■

تقریب کم‌ترین مربعات

خوراندن منحنی‌ها به مجموعه‌های داده شده از نقاط، یک کاربرد مهم ماتریس‌های مطلقاً مثبت متقارن است. فرض کنید به ما مجموعه‌ای از m نقطه‌ی

$$(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)$$

داده شده است، که در آن می‌دانیم ممکن است y_iها دارای خطای اندازه‌گیری باشند. می‌خواهیم یک تابع $F(x)$ بیابیم به طوری که خطاهای تقریب

$$\eta_i = F(x_i) - y_i \qquad (۱۳-۲۸)$$

برای $i = 1, 2, ..., m$ کوچک باشند. شکل تابع F به مسئله‌ی در دست بررسی بستگی دارد. در این جا فرض می‌کنیم شکل یک مجموع وزن‌دار خطی دارد:

$$F(x) = \sum_{j=1}^{n} c_j f_j(x)$$

که در آن تعداد جمله‌های سری n و *توابع پایه‌ی* f_i بر مبنای دانسته‌های مسئله‌ی در دست بررسی انتخاب شده‌اند. یک انتخاب معمول $f_i(x) = x^{j-1}$ است، که بدین معنی است که

$$F(x) = c_1 + c_2 x + c_3 x^2 + ... + c_n x^{n-1}$$

یک چندجمله‌ای از درجه‌ی $n-1$ نسبت به x است. بنابراین با داشتن m نقطه‌ی داده‌ای (x_1, y_1)، (x_2, y_2)، ...، (x_m, y_m)، می‌خواهیم n ضریب $c_1, c_2, ..., c_n$ را بیابیم به طوری که خطاهای تقریب $\eta_1, \eta_2, ..., \eta_m$ کمینه شوند.

با انتخاب $n = m$، می‌توانیم هر y_i را به طور *دقیق* در تساوی (۲۸-۱۷) محاسبه کنیم. با این حال چنین تابع Fای با این درجه‌ی بالا علاوه بر داده‌ها «نویز را هم می‌خوراند»، و معمولاً وقتی برای پیش‌بینی y برای مقادیر نادیده‌ی x به کار می‌رود، نتایج ضعیفی دارد. معمولاً خوب است که n را به مقدار زیادی کوچک‌تر از m انتخاب کنیم و امیدوار باشیم که با انتخاب مناسب ضرایب c_j، می‌توانیم یک تابع F به دست بیاوریم که الگوهای مهم داده‌های موجود (نقاط) را می‌یابد، بدون این که توجه زیادی به نویز بکند. چندین قانون نظری برای انتخاب n وجود دارد، ولی آن‌ها فراتر از حوصله‌ی این کتاب هستند. در هر حالتی، وقتی مقداری برای n انتخاب شد که از m کوچک‌تر است، با مجموعه‌ای از معادلات فرامعین (overdetermined) سروکار خواهیم داشت که می‌خواهیم جواب آن را تخمین بزنیم. اکنون نشان می‌دهیم که چگونه می‌توان این کار را انجام داد.

فرض کنید

$$A = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \cdots & f_n(x_m) \end{pmatrix}$$

نشان دهنده‌ی ماتریس مقادیر تابع پایه در نقاط داده شده باشد؛ یعنی $a_{ij} = f_j(x_i)$. فرض کنید $c = (c_k)$ نشان‌دهنده‌ی بردار n تایی مورد نظر از ضرایب باشد. در این صورت

$$Ac = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \cdots & f_n(x_m) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}$$
$$= \begin{pmatrix} F(x_1) \\ F(x_2) \\ \vdots \\ F(x_m) \end{pmatrix}$$

بردار m تایی از «مقادیر پیش‌بینی شده» برای y است. بنابراین

$$\eta = Ac - y$$

بردار با اندازه‌ی m از *خطاهای تقریب* (approximation error) است.

برای کمینه کردن خطاهای تقریب، سعی می‌کنیم نُرم بردار خطای η را کمینه کنیم، که *جواب کم‌ترین مربعات* (least-square solution) را به ما می‌دهد، چرا که

$$\|\eta\| = \left(\sum_{i=1}^{m} \eta_i^2 \right)^{1/2}$$

از آن جایی که

$$\|\eta\|^2 = \|Ac - y\|^2 = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} c_j - y_i \right)^2$$

می‌توانیم $\|\eta\|$ را با مشتق گرفتن از $\|\eta\|^2$ نسبت به هر c_k و سپس ۰ قرار دادن نتیجه کمینه کنیم:

$$\frac{d\|\eta\|^2}{dc_k} = \sum_{i=1}^{m} 2 \left(\sum_{j=1}^{n} a_{ij} c_j - y_i \right) a_{ik} = 0 \qquad (۱۸-۲۸)$$

n تساوی (۱۸-۲۸) برای $k = 1, 2, ..., n$ با یک تساوی ماتریسی

$$(Ac - y)^T A = 0$$

معادل است، و یا به طور معادل (با استفاده از تمرین ت-۱-۲)، با

$$A^T (Ac - y) = 0$$

که ایجاب می‌کند

$$A^T Ac = A^T y \qquad (۱۹-۲۸)$$

در آمار، به این *معادله‌ی نُرم* گفته می‌شود. ماتریس $A^T A$ طبق تمرین ت-۱-۲ متقارن است، و اگر A

رتبه‌ی ستونی کامل داشته باشد، آن گاه طبق قضیه‌ی ت-۶، $A^T A$ مطلقاً مثبت هم هست. بنابراین $(A^T A)^{-1}$ وجود دارد، و جواب معادله‌ی (۲۸-۱۹) عبارت است از

$$\begin{aligned} c &= ((A^T A)^{-1} A^T) y \\ &= A^+ y \end{aligned} \qquad (۲۸-۲۰)$$

که در آن به ماتریس $A^+ = ((A^T A)^{-1} A^T)$، شبه‌معکوس (pseudoinverse) ماتریس A گفته می‌شود. شبه‌معکوس یک گسترش طبیعی از مفهوم معکوس ماتریس برای حالتی است که A غیر مربعی است. (تساوی (۲۸-۲۰) را به عنوان جواب تقریبی $Ac = y$ با $A^{-1}b$ به عنوان جواب دقیق $Ax = b$ مقایسه کنید.)

به عنوان مثال از ساختن یک تابع کم‌ترین مربعات، فرض کنید پنج نقطه‌ی داده شده داریم

$$\begin{aligned} (x_۱, y_۱) &= (-۱, ۲), \\ (x_۲, y_۲) &= (۱, ۱), \\ (x_۳, y_۳) &= (۲, ۱), \\ (x_۴, y_۴) &= (۳, ۰), \\ (x_۵, y_۵) &= (۵, ۳) \end{aligned}$$

که در شکل ۲۸-۳ به صورت نقاط سیاه نشان داده شده‌اند. می‌خواهیم این نقاط را به یک چندجمله‌ای درجه دوی

$$F(x) = c_۱ + c_۲ x + c_۳ x^۲$$

بخورانیم. با ماتریس مقادیر تابع پایه آغاز می‌کنیم

$$A = \begin{pmatrix} ۱ & x_۱ & x_۱^۲ \\ ۱ & x_۲ & x_۲^۲ \\ ۱ & x_۳ & x_۳^۲ \\ ۱ & x_۴ & x_۴^۲ \\ ۱ & x_۵ & x_۵^۲ \end{pmatrix} = \begin{pmatrix} ۱ & -۱ & ۱ \\ ۱ & ۱ & ۱ \\ ۱ & ۲ & ۴ \\ ۱ & ۳ & ۹ \\ ۱ & ۵ & ۲۵ \end{pmatrix}$$

که شبه‌معکوس آن عبارت است از

$$A^+ = \begin{pmatrix} ۰/۵۰۰ & ۰/۳۰۰ & ۰/۲۰۰ & ۰/۱۰۰ & ۰/۱۰۰ \\ -۰/۳۸۸ & ۰/۰۹۳ & ۰/۱۹۰ & ۰/۱۹۳ & ۰/۰۸۸ \\ ۰/۰۶۰ & -۰/۰۳۶ & -۰/۰۴۸ & -۰/۰۳۶ & ۰/۰۶۰ \end{pmatrix}$$

با ضرب y در A^+، بردار ضرایب

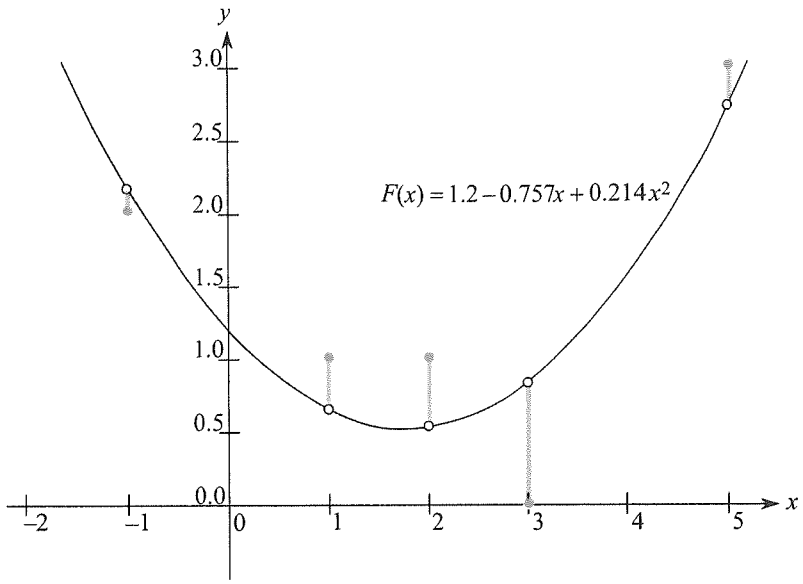
$$c = \begin{pmatrix} ۱/۲۰۰ \\ -۰/۷۵۷ \\ ۰/۲۱۴ \end{pmatrix}$$

را به دست می‌آوریم، که متناظر است با چندجمله‌ای درجه دوی

$$F(x) = ۱٫۲ - ۰٫۷۵۷x + ۰٫۲۱۴x^۲$$

به عنوان نزدیک‌ترین تابع درجه دو به داده‌های داده شده، از نظر روش کم‌ترین مربعات.

به عنوان یک مسئله‌ی کاربردی، تساوی معمولی (۲۸–۱۹) را با ضرب y در A^T و سپس یافتن یک تجزیه‌ی LU برای $A^T A$ حل می‌کنیم. اگر A دارای رتبه‌ی کامل باشد، ماتریس $A^T A$ به صورت تضمین شده غیرتکین است، چرا که متقارن و مطلقاً مثبت است. (تمرین ت–۱–۲ و قضیه‌ی ت–۶ را ببینید.)



شکل ۲۸–۳ خوراندن کم‌ترین مربعات یک چندجمله‌ای درجه دو به مجموعه‌ای از پنج نقطه‌ی داده شده‌ی $\{(۵, ۳), (۳, ۰), (۲, ۱), (۱, ۱), (-۱, ۲)\}$. نقاط تیره نشان دهنده‌ی نقاط داده‌ای هستند، و نقاط سفید مقادیر تخمین زده شده‌ی آن‌ها، که توسط چندجمله‌ای $F(x) = ۱٫۲ - ۰٫۷۵۷x + ۰٫۲۱۴x^۲$ پیش‌بینی شده‌اند، که این تابع، چندجمله‌ای درجه دو ای است که مجموع خطاهای مربع شده را کمینه می‌کند. خطای هر نقطه‌ی داده شده به صورت خط سایه‌دار نشان داده شده است.

تمرین‌ها

۲۸–۳–۱ اثبات کنید که هر عنصر روی قطر اصلی یک ماتریس مطلقاً مثبت متقارن، مثبت است.

۲۸–۳–۲ فرض کنید $A = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$ یک ماتریس مطلقاً مثبت متقارن ۲×۲ باشد. با «کامل کردن مربع» به روشی مشابه آن‌چه در اثبات لم ۲۸–۵ استفاده شد، اثبات کنید که دترمینان این ماتریس، یعنی $ac - b^۲$ مثبت است.

۲۸-۳-۳ اثبات کنید که عنصر بیشینه در یک ماتریس مطلقاً مثبت متقارن، بر روی قطر اصلی قرار دارد.

۲۸-۳-۴ اثبات کنید که دترمینان هر یک از زیرماتریس‌های مقدم یک ماتریس مطلقاً مثبت متقارن، مثبت است.

۲۸-۳-۵ فرض کنید A_k نشان‌دهنده‌ی kامین زیرماتریس مقدم یک ماتریس مطلقاً مثبت متقارن A باشد. ثابت کنید که $\det(A_k)/\det(A_{k-1})$، معادل است با kامین محور در هنگام تجزیه‌ی LU، که در آن برای راحتی قرار داده‌ایم $\det(A_0) = 1$.

۲۸-۳-۶ تابع به شکل

$$F(x) = c_1 + c_2 x \lg x + c_3 e^x$$

را بیابید که بهترین تابع کم‌ترین مربعات برای نقاط

$$(1, 1), (2, 1), (3, 3), (4, 8)$$

است.

۲۸-۳-۷ نشان دهید که شبه‌معکوس A^+ چهار تساوی زیر را ارضا می‌کند:

$$AA^+A = A,$$
$$A^+AA^+ = A^+,$$
$$(AA^+)^T = AA^+,$$
$$(A^+A)^T = A^+A$$

مسائل

۲۸-۱ ***سیستم‌های سه قطری از معادلات خطی***

ماتریس سه قطری

$$A = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix}$$

را در نظر بگیرید.

I. یک تجزیه‌ی LU برای A بیابید.

II. با استفاده از جایگزینی جلویی و عقبی، یک جواب برای معادله‌ی $Ax = (1\ 1\ 1\ 1)^T$ بیابید.

III. معکوس A را بیابید.

مسائل

IV. نشان دهید که برای هر ماتریس سه قطری، مطلقاً مثبت، و متقارن A با اندازه‌ی $n \times n$، و هر بردار b با اندازه‌ی n، می‌توان با انجام یک تجزیه‌ی LU تساوی $Ax = b$ را در زمان $O(n)$ حل کرد. بحث کنید که هر متدی بر پایه‌ی تشکیل A^{-1} به صورت حدی در بدترین حالت پرهزینه‌تر است.

V. نشان دهید که برای هر ماتریس سه‌گانه و غیرتکین A با اندازه‌ی $n \times n$، و هر بردار b با اندازه‌ی n، می‌توان با انجام یک تجزیه‌ی LUP، معادله‌ی $Ax = b$ را در زمان $O(n)$ حل کرد.

۲-۲۸ *spline* ها

یک متد کاربردی برای *درون‌یابی* (interpolating) مجموعه‌ای از نقاط با یک منحنی، استفاده از *spline*های *مکعبی* (cupic spline) است. به ما مجموعه‌ی $\{(x_i, y_i) : i = 0, 1, \ldots, n\}$ از $n+1$ جفت مرتب (نقطه) داده شده است، که در آن $x_0 < x_1 < \ldots < x_n$. می‌خواهیم یک منحنی (spline) قطعه‌ای- مکعبی $f(x)$ برای این نقاط بیابیم. یعنی، منحنی $f(x)$ از n چندجمله‌ای مکعبی $f_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$ برای $i = 0, 1, \ldots, n-1$ تشکیل شده است، که در آن اگر x در بازه‌ی $x_i \le x \le x_{i+1}$ قرار گیرد، آن گاه مقدار منحنی توسط $f(x) = f_i(x - x_i)$ تعیین می‌شود. نقاط x_i که چندجمله‌ای‌های مکعبی در آن‌ها به یکدیگر «چسبیده‌اند»، *گره* (knot) نام دارند. برای سادگی فرض خواهیم کرد که برای $i = 0, 1, \ldots, n$ داریم $x_i = i$.

برای اطمینان از پیوستگی $f(x)$ باید داشته باشیم

$$f(x_i) = f_i(0) = y_i,$$
$$f(x_{i+1}) = f_i(1) = y_{i+1}$$

برای $i = 0, 1, \ldots, n-1$. برای اطمینان از این که $f(x)$ به اندازه‌ی کافی هموار است، همچنین باید مشتق اول در هر یک از گره‌ها پیوسته باشد:

$$f'(x_{i+1}) = f_i'(1) = f_{i+1}'(0)$$

برای $i = 0, 1, \ldots, n-1$.

I. فرض کنید برای $i = 0, 1, \ldots, n$، علاوه بر مقادیر نقاط $\{(x_i, y_i)\}$ مشتق اول $D_i = f'(x_i)$ در هر یک از گره‌ها نیز داده شده است. هر یک از ضرایب a_i، b_i، c_i، و d_i را نسبت به مقادیر y_i، y_{i+1}، D_i، و D_{i+1} توصیف کنید. (به خاطر داشته باشید که $x_i = i$.) با چه سرعتی می‌توان $4n$ ضریب موردنظر را از مقادیر نقاط و مشتق‌های اول محاسبه کرد؟

این سؤال باقی می‌ماند که چگونه باید مشتق اول $f(x)$ را در گره‌ها انتخاب کرد. یک روش این است که مشتق دوم هم در گره‌ها پیوسته باشد:

$$f''(x_{i+1}) = f_i''(1) = f_{i+1}''(0)$$

برای $i = ۰, ۱, ..., n-۱$. در گره‌های اول و آخر، فرض می‌کنیم که $f''(x_۰) = f''_۰(۰) = ۰$ و $f''(x_n) = f''_{n-۱}(۱) = ۰$. این فرض‌ها $f(x)$ را به یک spline مکعبی طبیعی تبدیل می‌کنند.

II. با استفاده از محدودیت‌های پیوستگی مشتق دوم نشان دهید که برای $i = ۰, ۱, ..., n-۱$ داریم

$$D_{i-۱} + ۴D_i + D_{i+۱} = ۳(y_{i+۱} - y_{i-۱}) \qquad (۲۸-۲۱)$$

III. نشان دهید که

$$۲D_۰ + D_۱ = ۳(y_۱ - y_۰) \qquad (۲۸-۲۲)$$
$$D_{n-۱} + ۲D_n = ۳(y_n - y_{n-۱})$$

$$۲D_۰ + D_۱ = ۳(y_۱ - y_۰) \qquad (۲۸-۲۳)$$
$$D_{n-۱} + ۲D_n = ۳(y_n - y_{n-۱})$$

IV. تساوی‌های (۲۸-۲۱)-(۲۸-۲۳) را به صورت یک معادله‌ی ماتریسی شامل بردار $D = \langle D_۰, D_۱, ..., D_n \rangle$ از مجهولات بازنویسی کنید. ماتریس معادله‌ی شما چه خصوصیاتی دارد؟

V. بحث کنید که یک مجموعه از $n+۱$ نقطه را می‌توان در زمان $O(n)$ با یک spline مکعبی طبیعی درون‌یابی کرد (مسئله‌ی ۲۸-۱ را ببینید).

VI. نشان دهید که چگونه می‌توان یک spline مکعبی طبیعی را تعیین کرد، که مجموعه‌ای از $n+۱$ نقطه‌ی (x_i, y_i) را که $x_۰ < x_۱ < ... < x_n$ را ارضا می‌کنند، درون‌یابی می‌کند، در حالتی که x_i لزوماً برابر با i نیست. چه معادله‌ی ماتریسی باید حل شود، و سرعت الگوریتم شما چقدر است؟

فصل
۲۹

برنامه‌ریزی خطی

بسیاری از مسائل را می‌توان به صورت بیشینه یا کمینه کردن یک هدف، که در آن منابع محدود هستند، فرمول‌بندی کرد. اگر بتوان هدف را به صورت یک تابع خطی از متغیرهای خاص توصیف کرد، و اگر بتوان محدودیت‌های منابع را به صورت چند تساوی یا نامساوی بر روی آن متغیرها بیان کرد، در این صورت یک *مسئله‌ی برنامه‌ریزی خطی* (linear-programming problem) داریم. برنامه‌های خطی در کاربردهای مختلفی رخ می‌دهند. با بررسی یک کاربرد در انتخابات سیاسی آغاز می‌کنیم.

یک مسئله‌ی سیاسی

فرض کنید شما یک سیاست‌مدار هستید و می‌خواهید در انتخاباتی پیروز شوید. منطقه‌ی شما دارای سه نوع مختلف بخش است- شهری، برون شهری، و روستایی. این بخش‌ها به ترتیب دارای ۱۰۰,۰۰۰، ۲۰۰,۰۰۰ و ۵۰,۰۰۰ رأی دهنده هستند. با این که تمام واجدین شرایط رأی واقعاً به پای صندوق‌های رأی نمی‌روند، شما می‌خواهید که حداقل نیمی از هر یک از سه بخش به شما رأی دهند. شما به هیچ وجه حاضر نیستید سیاست‌هایی را دنبال کنید که به آن‌ها اعتقاد ندارید. با این حال، متوجه می‌شوید مسائل خاصی هستند که ممکن است در بردن رأی‌ها در مناطقی خاص مؤثرتر باشند. مسائل اصلی مورد نظر شما عبارتند از ساختن جاده‌های بیشتر، کنترل اسلحه، تسهیلات کشاورزی، و مالیات بر روی سوخت برای بهبود حمل‌ونقل عمومی. طبق بررسی ستاد انتخاباتی شما، می‌توان تخمین زد که با هزینه‌ی ۱,۰۰۰ دلار بر روی تبلیغ برای هر یک از مسائل، چند رأی را در هر منطقه از دست می‌دهید یا به دست می‌آورید. این اطلاعات در جدول شکل ۲۹-۱ مشخص شده‌اند. در این جدول، هر ورودی نشان دهنده‌ی تعداد هزار رأی دهنده در هر یک از بخش‌های شهری، برون‌شهری و روستایی است، که می‌توان با صرف ۱,۰۰۰ دلار برای تبلیغ بر روی یک مسئله‌ی خاص، رأی آن‌ها را به دست آورد. ورودی‌های منفی نشان دهنده‌ی رأی‌هایی هستند که از دست خواهید داد. هدف یافتن هزینه‌ی کمینه برای بردن ۵۰,۰۰۰ رأی شهری، ۱۰۰,۰۰۰ رأی برون‌شهری، و ۲۵,۰۰۰ رأی روستایی است.

سیاست	شهری	برون شهری	روستایی
ساختن جاده	−2	5	3
کنترل اسلحه	8	2	−5
تسحیلات کشاورزی	0	0	10
مالیات سوخت	10	0	−2

شکل ۲۹-۱ تأثیر سیاست‌ها بر روی رأی دهنده‌ها. هر ورودی نشان‌دهنده‌ی تعداد هزار رأی‌دهنده‌ی شهری، برون‌شهری و روستایی است که می‌توان با صرف هزینه‌ی ۱,۰۰۰ دلار بر روی تبلیغات برای یک مسئله‌ی سیاسی خاص، رأی آن‌ها را برد. ورودی‌های منفی نشان‌دهنده‌ی رأی‌هایی هستند که از دست خواهند رفت.

با سعی و خطا می‌توان به یک استراتژی برای بردن تعداد رأی مورد نظر دست یافت، ولی ممکن است هزینه‌ی این استراتژی کمینه نباشد. به عنوان مثال، می‌توانید ۲۰,۰۰۰ دلار تبلیغ برای ساختن جاده‌ها، ۰ دلار برای کنترل اسلحه، ۴,۰۰۰ دلار برای تسهیلات کشاورزی، و ۹,۰۰۰ دلار برای مالیات سوخت هزینه کنید. در این صورت $۵۰ = (۱۰)۹ + (۰)۴ + (۸)۰ + (−۲)۲۰$ هزار رأی شهری، $۱۰۰ = (۰)۹ + (۲)۴ + (۵)۰ + (۰)۲۰$ هزار رأی برون‌شهری، و $۸۲ = (−۲)۹ + (۱۰)۴ + (−۵)۰ + (۳)۲۰$ هزار رأی روستایی را خواهید برد. تعداد رأی‌های شهری و برون‌شهری شما دقیقاً برابر با مقدار خواسته شده، و تعداد رأی‌های روستایی بیشتر از مقدار خواسته شده خواهد بود. (در واقع تعداد رأی‌هایی که در مناطق روستایی برده‌اید، بیشتر از تعداد رأی‌های موجود است!) برای به دست آوردن این رأی‌ها، در کل باید هزینه‌ای برابر با $۳۳ = ۹ + ۴ + ۰ + ۲۰$ هزار دلار صرف تبلیغات کنید.

طبیعتا ممکن است بخواهید بدانید که آیا استراتژی شما بهترین استراتژی ممکن است یا خیر. یعنی می‌خواهید بدانید که آیا می‌توانستید با صرف هزینه‌ی کم‌تر بر روی تبلیغات به اهداف خود برسید. سعی و خطای بیشتر ممکن است به شما کمک کند که به این سؤال جواب دهید، ولی ترجیح خواهید داد که یک متد مناسب برای جواب دادن به چنین سؤالاتی داشته باشید. برای انجام این کار این سؤال را به صورت ریاضی فرمول‌بندی می‌کنیم. از ۴ متغیر استفاده خواهیم کرد:

- x_1 نشان‌دهنده‌ی هزینه‌ی صرف شده بر روی تبلیغات برای ساختن جاده‌ها است، بر حسب هزار دلار،
- x_2 نشان‌دهنده‌ی هزینه‌ی صرف شده بر روی تبلیغات برای کنترل اسلحه است، بر حسب هزار دلار،
- x_3 نشان‌دهنده‌ی هزینه‌ی صرف شده بر روی تبلیغات برای تسهیلات کشاورزی است، بر حسب هزار دلار، و
- x_4 نشان دهنده‌ی هزینه‌ی صرف شده بر روی تبلیغات برای مالیات سوخت است، بر حسب هزار دلار.

می‌توانیم بردن حداقل ۵۰,۰۰۰ رأی شهری را به صورت

$$-۲x_1 + ۸x_2 + ۰x_3 + ۱۰x_4 \geq ۵۰ \qquad (۱-۲۹)$$

بنویسیم. به طور مشابه، می‌توانیم بردن حداقل ۱۰۰,۰۰۰ رأی برون شهری و حداقل ۲۵,۰۰۰ رأی روستایی را به صورت

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 \qquad (۲-۲۹)$$

و

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25 \qquad (۳-۲۹)$$

بنویسیم. هر مقداردهی به متغیرهای x_1، x_2، x_3، و x_4 به طوری که نامساوی‌های (۲۹-۱)-(۲۹-۳) ارضا شوند، یک استراتژی است که ما را به بردن تعداد رأی‌های کافی در هر یک از مناطق می‌رساند. برای این که هزینه‌ها را در حد ممکن پایین نگه داریم، می‌خواهیم هزینه‌ی صرف شده بر روی تبلیغات را کمینه کنیم. یعنی می‌خواهیم عبارت

$$x_1 + x_2 + x_3 + x_4 \qquad (۴-۲۹)$$

حداقل شود. با این که تبلیغ منفی در ستادهای انتخاباتی رایج است، ولی چیزی به شکل تبلیغ با هزینه‌ی منفی وجود ندارد. در نتیجه نیاز داریم که

$$x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, x_4 \geq 0 \qquad (۵-۲۹)$$

برقرار باشد. با ترکیب نامساوی‌های (۲۹-۱)-(۲۹-۳) و (۲۹-۵) با هدف کمینه کردن (۲۹-۴)، مجهول مورد نظر خود را به صورت یک «برنامه‌ی خطی» به دست می‌آوریم. این مسئله را به صورت

$$\text{minimize } x_1 + x_2 + x_3 + x_4 \qquad (۶-۲۹)$$

با شرایط

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50 \qquad (۷-۲۹)$$

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 \qquad (۸-۲۹)$$

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25 \qquad (۹-۲۹)$$

$$x_1, x_2, x_3, x_4 \geq 0 \qquad (۱۰-۲۹)$$

استاندارد می‌کنیم. جواب این برنامه‌ی خطی به یک استراتژی بهینه برای مسئله‌ی انتخابات ختم می‌شود.

برنامه‌های خطی کلی

در مسئله‌ی برنامه‌ریزی خطی، به طور کلی می‌خواهیم یک تابع خطی را با ارضای مجموعه‌ای از نامساوی‌ها بهینه کنیم. با داشتن مجموعه‌ای از اعداد حقیقی $a_1, a_2, ..., a_n$ و مجموعه‌ای از متغیرهای $x_1, x_2, ..., x_n$، یک *تابع خطی* f بر روی این متغیرها به صورت

$$f(x_1, x_2, ..., x_n) = a_1 x_1 + a_2 x_2 + ... + a_n x_n = \sum_{j=1}^{n} a_j x_j$$

تعریف می‌شود. اگر b یک عدد حقیقی و f یک تابع خطی باشد، آن گاه تساوی

$$f(x_1, x_2, ..., x_n) = b$$

یک *معادله‌ی خطی* است، و نامساوی‌های

$$f(x_1, x_2, ..., x_n) \leq b$$

و

$$f(x_1, x_2, ..., x_n) \geq b$$

نامساوی‌های خطی هستند. از اصطلاح *شرایط خطی* (linear constraints) برای نشان دادن هر یک از معادلات خطی یا نامساوی‌های خطی استفاده می‌کنیم. در برنامه‌ریزی خطی، نامساوی‌های اکید را مجاز نمی‌دانیم. به صورت رسمی، یک *مسئله‌ی برنامه‌ریزی خطی* عبارت است از مسئله‌ی کمینه یا بیشینه کردن یک تابع خطی با ارضای مجموعه‌ای از شرایط خطی. اگر بخواهیم تابع را کمینه کنیم، آن گاه به برنامه‌ی خطی یک *برنامه‌ی خطی کمینه*، و اگر بخواهیم تابع را بیشینه کنیم، به آن یک *برنامه‌ی خطی بیشینه* می‌گوییم.

باقی این فصل، نحوه‌ی فرمول‌بندی و یافتن جواب برای یک برنامه‌ی خطی را پوشش می‌دهد. با این که الگوریتم‌های با زمان چندجمله‌ای زیادی برای برنامه‌ریزی خطی وجود دارد، آن‌ها را در این فصل نخواهیم آموخت. در عوض الگوریتم سیمپلکس را بررسی خواهیم کرد، که قدیمی‌ترین الگوریتم برنامه‌ریزی خطی است. الگوریتم سیمپلکس در بدترین حالت در زمان چندجمله‌ای اجرا نمی‌شود، ولی نسبتاً کارآمد است، و در عمل موارد استفاده‌ی بسیاری دارد.

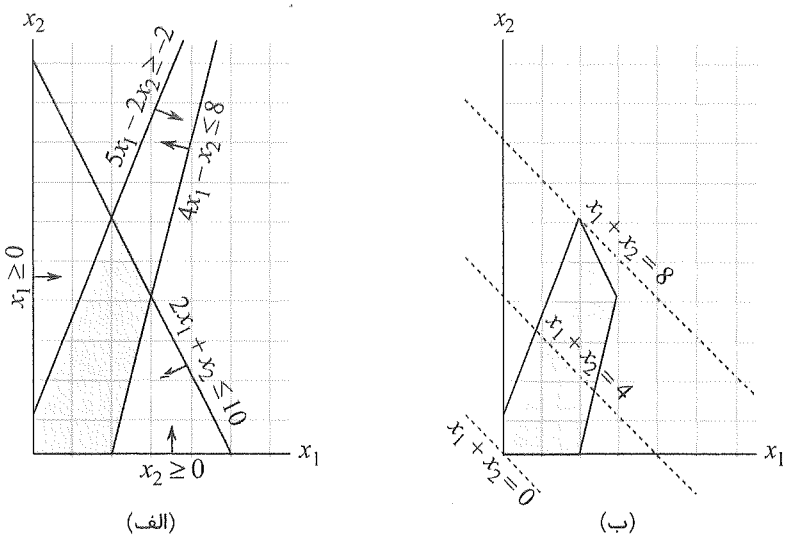
مروری بر روی برنامه‌ریزی خطی

برای توصیف خصوصیات و الگوریتم‌های برنامه‌ریزی‌های خطی، خوب است که شکل‌های متعارفی برای توصیف آن‌ها داشته باشیم. در این فصل از دو شکل استفاده خواهیم کرد، *استاندارد* و *ضعیف* (slack). تعریف دقیق این دو فرم در بخش ۲۹-۱ ارائه خواهد شد. به صورت غیر رسمی، یک برنامه‌ی خطی به شکل استاندارد عبارت است از بیشینه‌سازی یک تابع خطی با حضور شرایطی به شکل *نامساوی‌های* خطی، در حالی که یک برنامه‌ی خطی به شکل ضعیف عبارت است از بیشینه‌سازی یک تابع خطی با حضور شرایطی به شکل *معادلات* خطی. معمولاً از شکل استاندارد برای توصیف برنامه‌های خطی استفاده خواهیم کرد، ولی هنگام توصیف جزئیات الگوریتم سیمپلکس، استفاده از شکل ضعیف ساده‌تر خواهد بود. در حال حاضر، توجه خود را به بیشینه‌سازی یک تابع خطی با n متغیر با حضور مجموعه‌ای از m نامساوی خطی محدود خواهیم کرد.

اجازه دهید برای شروع، برنامه‌ی خطی زیر را با دو متغیر در نظر بگیریم:

$$\text{minimize } x_1 + x_2 \qquad (۲۹-۱۱)$$

با شرایط



شکل ۲۹-۲ (الف) برنامه‌ی خطی داده شده در تساوی‌های (۲۹-۱۲)-(۲۹-۱۵). هر محدودیت به صورت یک خط و یک جهت مشخص شده است. محل برخورد محدودیت‌ها، که همان ناحیه‌ی ممکن است، با سایه مشخص شده است. (ب) خطوط نقطه‌چین به ترتیب نشان دهنده‌ی نقاطی هستند که مقدار هدف در آن‌ها ۰، ۴، و ۸ است. جواب بهینه برای برنامه‌ی خطی عبارت است از $x_1 = ۲$ و $x_2 = ۶$ با مقدار هدف ۸.

$$۴x_1 - x_2 \leq ۸ \qquad (۲۹-۱۲)$$

$$۲x_1 + x_2 \leq ۱۰ \qquad (۲۹-۱۳)$$

$$۵x_1 - ۲x_2 \geq -۲ \qquad (۲۹-۱۴)$$

$$x_1, x_2 \geq ۰ \qquad (۲۹-۱۵)$$

به هر مقداردهی به متغیرهای x_1 و x_2 که تمام شرایط (۲۹-۱۲)-(۲۹-۱۵) را ارضا می‌کند، یک *جواب ممکن* (feasible solution) به برنامه‌ی خطی می‌گوییم. اگر گراف محدودیت‌ها را در سیستم مختصات کارتزین (x_1, x_2) رسم کنیم، مانند شکل ۲۹-۲ (الف)، خواهیم دید که مجموعه‌ی جواب‌های ممکن (که در شکل با سایه مشخص شده‌اند) یک ناحیه‌ی محدب را در فضای دوبعدی تشکیل می‌دهند.[۱] به این ناحیه‌ی محدب، *ناحیه‌ی ممکن* (feasible region) می‌گوییم. تابعی که می‌خواهیم آن را بیشینه کنیم، تابع *هدف* (objective function) نام دارد. از نظر مفهومی باید تابع هدف $x_1 + x_2$ را در هر نقطه‌ی ناحیه‌ی ممکن ارزیابی کنیم؛ به مقدار تابع هدف در یک نقطه‌ی خاص، *مقدار هدف* (objective value) می‌گوییم. در این صورت می‌توانیم نقطه‌ای که بیشترین مقدار هدف را دارد، به عنوان یک جواب بهینه تعیین کنیم. برای این مثال (و برای اکثر برنامه‌های خطی)، ناحیه‌ی

[۱] یک تعریف شهودی برای یک ناحیه‌ی محدب این است که خصوصیت زیر را ارضا می‌کند: برای هر دو نقطه در ناحیه، تمام نقاط روی پاره‌خط متصل‌کننده‌ی این دو نقطه درون ناحیه قرار دارند.

هدف شامل بی‌نهایت نقطه است، و بنابراین باید یک روش بهینه برای یافتن نقطه‌ای به دست بیاوریم که بیشترین مقدار هدف را دارد، بدون این که صریحاً تابع هدف را در تمام نقاط ناحیه‌ی ممکن ارزیابی کنیم.

در فضای دوبعدی می‌توانیم با یک رویه‌ی گرافیکی مقدار بهینه را مشخص کنیم. مجموعه‌ی نقاطی که برای آن‌ها داریم $z = x_1 + x_2$، برای هر z، یک خط است با شیب ۱−. اگر منحنی $x_1 + x_2 = 0$ را رسم کنیم، یک خط به دست می‌آوریم با شیب ۱− که از مبدأ می‌گذرد، مانند شکل ۲۹-۲ (ب). برخورد این خط با ناحیه‌ی ممکن، مجموعه‌ی جواب‌های ممکنی را تشکیل می‌دهد که مقدار هدف آن‌ها ۰ است. در این مثال برخورد خط با ناحیه‌ی ممکن، نقطه‌ی (۰,۰) است. به طور کلی‌تر برای هر z، برخورد خط $x_1 + x_2 = z$ و ناحیه‌ی ممکن عبارت است از مجموعه‌ی جواب‌هایی که مقدار هدف آن‌ها z است. شکل ۲۹-۲ (ب) خطوط $x_1 + x_2 = 0$، $x_1 + x_2 = 4$، و $x_1 + x_2 = 8$ را نشان می‌دهد. چون ناحیه‌ی ممکن در شکل ۲۹-۲ کران‌دار است، باید یک مقدار بیشینه z وجود داشته باشد که برای آن برخورد خط $x_1 + x_2 = z$ با ناحیه‌ی ممکن ناتهی باشد. هر نقطه‌ای که این خاصیت را داشته باشد یک جواب بهینه برای برنامه‌ی خطی است، که در این مثال عبارت است از نقاط $x_1 = 2$ و $x_2 = 6$ با مقدار هدف ۸.

این که یک جواب بهینه برای برنامه‌ی خطی بر روی یکی از رأس‌های ناحیه‌ی ممکن رخ داده است، تصادفی نیست. مقدار بیشینه‌ی z که برای آن خط $x_1 + x_2 = z$ با ناحیه‌ی ممکن برخورد می‌کند، باید بر روی مرز ناحیه‌ی ممکن قرار داشته باشد، و بنابراین برخورد این خط با مرز ناحیه‌ی ممکن یا یک رأس است و یا یک پاره‌خط. اگر ناحیه‌ی برخورد یک رأس باشد، در این صورت فقط یک جواب بهینه وجود دارد، و اگر یک پاره‌خط باشد، تمام نقاط بر روی این پاره‌خط باید مقدار هدف یکسانی داشته باشند؛ به خصوص هر دو نقطه‌ی پایانی پاره خط، جواب‌های بهینه هستند. از آن‌جایی که هر نقطه‌ی پایانی یک پاره‌خط، یک رأس است، در این حالت هم یک جواب بهینه بر روی یک رأس وجود دارد.

با این که نمی‌توانیم به سادگی گراف برنامه‌های خطی با بیش از دو متغیر را رسم کنیم، ولی استدلال شهودی مشابهی برای آن‌ها هم برقرار است. اگر سه متغیر داشته باشیم، هر محدودیت متناظر خواهد بود با یک نیم‌فضا (half-space) در فضای سه بعدی. برخورد این نیم‌فضاها، ناحیه‌ی ممکن را تشکیل می‌دهد. اکنون مجموعه‌ی نقاطی که تابع هدف در آن‌ها یک مقدار خاص z را به خود می‌گیرد، یک صفحه را تشکیل می‌دهند. اگر تمام ضرایب تابع هدف نامنفی باشند، و اگر مبدأ یک جواب ممکن برای برنامه‌ی خطی باشد، در این صورت، همین طور که این صفحه را از مبدأ دور می‌کنیم، مقدار هدف نقاطی که به آن‌ها برخورد می‌کنیم افزایش می‌یابد. (اگر مبدأ در ناحیه‌ی ممکن نباشد و یا اگر تابع هدف ضریب منفی داشته باشد، تصویر شهودی مقداری پیچیده‌تر خواهد شد.) مانند حالت دوبعدی، چون ناحیه‌ی ممکن محدب است، مجموعه‌ی نقاطی که مقدار هدف بهینه را دارند باید شامل یک رأس از ناحیه‌ی ممکن باشند. به طور مشابه اگر n متغیر داشته باشیم، هر محدودیت یک نیم‌فضا را در فضای n بعدی مشخص می‌کند. ناحیه‌ی ممکن تشکیل شده از برخورد

این نیم‌فضاها یک *سیمپلکس* (simplex) نام دارد. اکنون تابع هدف یک سوپرصفحه است، و به خاطر محدب بودن، یک جواب بهینه بر روی یک رأس از سیمپلکس رخ می‌دهد.

الگوریتم سیمپلکس یک برنامه‌ی خطی را به عنوان ورودی دریافت کرده و یک جواب بهینه برای آن برنامه‌ی خطی باز می‌گرداند. این الگوریتم از یک رأس از سیمپلکس آغاز کرده و دنباله‌ای از تکرارها انجام می‌دهد. در هر تکرار، الگوریتم از روی رأس فعلی، یک ضلع را دنبال کرده و به یکی از رأس‌های همسایه که مقدار هدف آن کم‌تر از مقدار هدف رأس فعلی نیست، جابه‌جا می‌شود. الگوریتم سیمپلکس زمانی پایان می‌یابد که به یک نقطه‌ی بیشینه‌ی نسبی دست یابد، که رأسی است که مقدار هدف آن از مقدار هدف تمام همسایه‌های آن بیشتر است. چون ناحیه‌ی ممکن محدب، و تابع هدف خطی است، این مقدار بهینه‌ی نسبی در واقع مقدار بهینه‌ی مطلق است. در بخش ۲۹-۴ از مفهومی به نام «دوگانگی» استفاده می‌کنیم تا نشان دهیم که جواب بازگردانده شده توسط الگوریتم سیمپلکس، بهینه است.

با این که دید هندسی یک شهود مناسب از نحوه‌ی کارکرد الگوریتم سیمپلکس به ما می‌دهد، ولی هنگام توسعه‌ی جزئیات الگوریتم در بخش ۲۹-۳ به طور مستقیم به آن‌ها اشاره نخواهیم کرد. در عوض از دید جبری استفاده خواهیم کرد. ابتدا برنامه‌ی خطی داده شده را به صورت ضعیف بازنویسی می‌کنیم، که مجموعه‌ای است از معادلات خطی. این معادلات خطی بعضی از متغیرها را، که «متغیرهای پایه» نام دارند، برحسب متغیرهای دیگر که «متغیرهای پیرو» نام دارند، توصیف می‌کنند. حرکت از یک رأس به رأس دیگر با تبدیل یک متغیر پایه به یک متغیر پیرو و تبدیل یک متغیر پیرو به یک متغیر پایه انجام می‌شود. به این عملیات یک «چرخش» (pivot) گفته می‌شود، که از نظر جبری چیزی نیست جز بازنویسی برنامه‌ی خطی معادل به صورت ضعیف.

مثال دو متغیری توصیف شده در بالا یک نمونه‌ی ساده بود. جزئیات زیاد دیگری وجود دارند که باید به فکر حل آن‌ها باشیم. این مسائل عبارتند از شناسایی برنامه‌های خطی که هیچ جوابی ندارند، برنامه‌های خطی که هیچ جواب بهینه‌ی کران‌داری ندارند، و برنامه‌های خطی که برای آن‌ها، مبدأ مختصات، یک جواب ممکن نیست.

کاربردهای برنامه‌ریزی پویا

برنامه‌ریزی پویا کاربردهای بسیاری دارد. هر کتابی مربوط به تحقیق در عملیات مملو است از مثال‌هایی از برنامه‌ریزی پویا. به علاوه، امروزه برنامه‌ریزی پویا ابزاری استاندارد است که به دانشجویان اکثر دانشکده‌های مدیریت آموخته می‌شود. سناریوی انتخابات یک مثال معمول است. دو مثال دیگر از برنامه‌ریزی پویا به صورت زیر است:

- یک شرکت هواپیمایی می‌خواهد برنامه‌ی خدمه‌ی پرواز خود را تعیین کند. اداره‌ی امور هوایی فدرال محدودیت‌های بسیاری برای شرکت‌های هواپیمایی تعیین کرده است، از جمله محدودیت تعداد ساعات متوالی که هر یک از خدمه‌ی پرواز می‌تواند داشته باشد، و این که خدمه‌ی خاص در طول یک ماه فقط باید بر روی یک مدل هواپیما کار کند. شرکت هواپیمایی

می‌خواهد برنامه‌ی خدمه‌ی پرواز خود را طوری تعیین کند که از کم‌ترین خدمه‌ی ممکن استفاده کند.

- یک کمپانی نفتی می‌خواهد تصمیم بگیرد که در کجا چاه‌های نفتی خود را بنا کند. بنا کردن یک چاه نفت در منطقه‌ای خاص هزینه‌ای متناسب دارد، و بسته به موقعیت جغرافیایی، بازدهی خاصی را از آن انتظار داریم. بودجه‌ی کمپانی برای بنا کردن یک چاه نفت محدود است و می‌خواهد با این بودجه، میزان نفت مورد انتظار خود را بیشینه کند.

برنامه‌های خطی برای مدل کردن و حل مسائل گراف و ترکیبیاتی هم مفید هستند، مسائلی مانند آن‌هایی که در این کتاب مطرح شدند. پیش از این در بخش ۲۴-۴ نمونه‌ی خاصی از برنامه‌ریزی خطی را که برای حل سیستم‌های محدودیت‌های اختلاف از آن استفاده می‌شود، دیدیم. در بخش ۲۹-۲ خواهیم آموخت که چگونه می‌توان مسائل متعددی از گراف و شبکه‌ی شار را به صورت برنامه‌های خطی فرمول‌بندی کرد. در بخش ۳۵-۴ از برنامه‌ریزی خطی به عنوان یک ابزار برای یافتن یک جواب تقریبی برای یک مسئله‌ی گراف دیگر استفاده خواهیم کرد.

الگوریتم‌های برنامه‌ریزی پویا

در این فصل الگوریتم سیمپلکس را خواهیم آموخت. این الگوریتم، اگر به خوبی پیاده‌سازی شود، معمولاً برنامه‌های خطی کلی را در عمل با سرعت خوبی حل می‌کند. با این حال، با کمی دقت می‌توان ورودی‌هایی به الگوریتم سیمپلکس داد که برای اجرا بر روی آن‌ها به زمان نمایی نیاز داشته باشد. اولین الگوریتم با زمان چندجمله‌ای برای برنامه‌ریزی خطی، *الگوریتم بیضوی* (ellipsoid algorithm) بود، که در عمل به کندی اجرا می‌شود. کلاس دوم از الگوریتم‌های با زمان چندجمله‌ای به *متدهای نقطه‌ی درونی* (interior-point method) معروف‌اند. بر خلاف الگوریتم سیمپلکس، که روی مرز ناحیه‌ی ممکن حرکت کرده و در هر تکرار، یک جواب ممکن نگه می‌دارد که رأسی از سیمپلکس است، این الگوریتم‌ها در نقاط درونی ناحیه‌ی ممکن حرکت می‌کنند. جواب‌های میانی، با این که ممکن هستند، ولی لزوماً رأس‌هایی از سیمپلکس نیستند، ولی جواب نهایی یک رأس است. برای ورودی‌های بزرگ، الگوریتم‌های نقطه‌ی درونی می‌توانند با سرعت الگوریتم سیمپلکس، و یا بعضی مواقع سریع‌تر از آن اجرا شوند.

اگر به یک برنامه‌ی خطی این شرط را اضافه کنیم که تمام متغیرهای آن باید مقادیر صحیح به خود بگیرند، یک *برنامه‌ی خطی صحیح* (integer linear program) خواهیم داشت. تمرین ۳۴-۵-۳ از شما می‌خواهد نشان دهید که یافتن فقط یک جواب ممکن برای این مسئله NP-سخت است؛ چون تا کنون هیچ الگوریتم چندجمله‌ای برای حل مسائل NP-سخت یافت نشده است، هیچ الگوریتمی با زمان چندجمله‌ای برای حل برنامه‌ریزی خطی صحیح وجود ندارد. در مقابل، می‌توانیم یک مسئله‌ی برنامه‌ریزی خطی در حالت کلی را در زمان چندجمله‌ای حل کنیم.

در این فصل، اگر یک برنامه‌ی خطی با متغیرهای $x = (x_1, x_2, ..., x_n)$ داشته باشیم و بخواهیم به یک مقداردهی خاص از متغیرها اشاره کنیم، از نماد $\bar{x} = (\bar{x}_1, \bar{x}_2, ..., \bar{x}_n)$ استفاده خواهیم کرد.

۲۹-۱ شکل‌های استاندارد و ضعیف

در این بخش دو شکل استاندارد و ضعیف تعریف می‌شود، که برای توصیف برنامه‌های خطی و کار با آن‌ها مفید خواهند بود. در شکل استاندارد تمام محدودیت‌ها به صورت نامساوی هستند، در حالی که در شکل ضعیف این محدودیت‌ها به صورت تساوی خواهند بود (غیر از آن‌هایی که نیاز دارند که متغیرها نامنفی باشند).

شکل استاندارد

در *شکل استاندارد* به ما n عدد حقیقی $c_1, c_2, ..., c_n$، m عدد حقیقی $b_1, b_2, ..., b_m$، و mn عدد حقیقی a_{ij} برای $i = 1, 2, ..., m$ و $j = 1, 2, ..., n$ داده شده است. می‌خواهیم n عدد حقیقی $x_1, x_2, ..., x_n$ را بیابیم به طوری که

$$\text{maximize} \quad \sum_{j=1}^{n} c_j x_j \tag{۲۹-۱۶}$$

با ارضای شرایط

$$\sum_{j=1}^{n} a_{ij} x_j \le b_i \qquad \text{برای } i = 1, 2, ..., m \tag{۲۹-۱۷}$$

$$x_j \ge 0 \qquad \text{برای } j = 1, 2, ..., n \tag{۲۹-۱۸}$$

با تعمیم اصطلاحاتی که برای برنامه‌های خطی دو متغیره به کار بردیم، به عبارت (۲۹-۱۶) *تابع هدف*، و به $n + m$ نامساوی در خطوط (۲۹-۱۷) و (۲۹-۱۸) *شرایط* می‌گوییم. n محدودیت خط (۲۹-۱۸) *محدودیت‌های عدم منفی بودن* (nonnegativity constraint) نام دارند. نیازی نیست که یک برنامه‌ی خطی دلخواه لزوماً محدودیت عدم منفی بودن داشته باشد، ولی در شکل استاندارد این محدودیت‌ها اجباری هستند. بعضی مواقع ساده است که یک برنامه‌ی خطی را به صورتی فشرده‌تر بنویسیم. اگر یک ماتریس $A = (a_{ij})$ با اندازه‌ی $m \times n$، یک بردار $b = (b_i)$ با اندازه‌ی m، یک بردار $c = (c_j)$ با اندازه‌ی n، و یک بردار $x = (x_j)$ با اندازه‌ی n بسازیم، می‌توانیم برنامه‌ی خطی توصیف شده در (۲۹-۱۶)–(۲۹-۱۸) را به صورت

$$\text{maximize} \quad c^T x \tag{۲۹-۱۹}$$

با شرایط

$$Ax \le b \tag{۲۹-۲۰}$$

$$x \ge 0 \tag{۲۹-۲۱}$$

بازنویسی کنیم. در خط (۲۹-۱۹)، $c^T x$ ضرب داخلی دو بردار است، در نامساوی (۲۹-۲۰)، Ax یک ضرب ماتریس-بردار، و در نامساوی (۲۹-۲۱)، عبارت $x \ge 0$ بدین معنی است که هر ورودی

بردار x باید نامنفی باشد. می‌بینیم که می‌توانیم یک برنامه‌ی خطی را به صورت استاندارد به وسیله‌ی سه‌تایی (A,b,c) توصیف کنیم، و این قرارداد را خواهیم داشت که A، b، و c همیشه دارای ابعاد داده شده در بالا هستند.

اکنون اصطلاحاتی را برای توصیف جواب‌های برنامه‌های خطی معرفی می‌کنیم. از بعضی از این اصطلاحات در مثال قبلی برنامه‌ی خطی دو متغیره استفاده شده بود. به یک مقداردهی متغیرهای $\bar{x}$ که تمام شرایط را ارضا می‌کند یک *جواب ممکن* می‌گوییم، و یک مقداردهی به متغیرها که حداقل یکی از شرایط را ارضا نمی‌کند، یک *جواب ناممکن* نام دارد. می‌گوییم یک جواب $\bar{x}$ دارای *مقدار هدف* $c^T\bar{x}$ است. یک جواب ممکن $\bar{x}$ که مقدار هدف آن در میان تمام جواب‌های ممکن بیشینه است، یک *جواب بهینه* خواهد بود، و به مقدار هدف آن، $c^T\bar{x}$، *مقدار هدف بهینه* می‌گوییم. اگر یک برنامه‌ی خطی هیچ جواب ممکنی نداشته باشد به آن یک برنامه‌ی خطی *ناممکن* می‌گوییم؛ در غیر این صورت یک برنامه‌ی خطی *ممکن* خواهیم داشت. اگر یک برنامه‌ی خطی دارای جواب ممکن باشد، ولی مقدار هدف بهینه‌ی کران‌دار نداشته باشد، می‌گوییم این برنامه‌ی خطی *بی‌کران* است. تمرین ۲۹-۱-۹ از شما می‌خواهد نشان دهید که یک برنامه‌ی خطی می‌تواند یک مقدار هدف کران‌دار داشته باشد حتی اگر ناحیه‌ی ممکن بی‌کران باشد.

تبدیل برنامه‌های خطی به شکل استاندارد

اگر یک برنامه‌ی خطی داشته باشیم که به صورت بیشینه‌سازی یا کمینه‌سازی یک تابع خطی با حضور چند محدودیت خطی باشد، همیشه می‌توانیم آن را به شکل استاندارد تبدیل کنیم. یک برنامه‌ی خطی به یکی از چهار دلیل زیر ممکن است به شکل استاندارد نباشد:

۱. تابع هدف ممکن است به جای این که به صورت بیشینه باشد، به صورت کمینه باشد.
۲. ممکن است متغیرهایی وجود داشته باشند که محدودیت عدم منفی بودن نداشته باشند.
۳. ممکن است در برنامه‌ی خطی *محدودیت‌های معادله‌ای* (equality constraint) وجود داشته باشد، که به جای علامت کوچک‌تر مساوی، علامت مساوی دارند.
٤. ممکن است در برنامه‌ی خطی *محدودیت‌های نامعادله‌ای* (inequality constraint) وجود داشته باشد که به جای علامت کوچک‌تر مساوی، علامت بزرگ‌تر مساوی دارند.

هنگام تبدیل یک برنامه‌ی خطی L به یک برنامه‌ی خطی دیگر L'، مطلوب است این خصوصیت را داشته باشیم که یک جواب بهینه به L' منجر به یک جواب بهینه به L شود. برای به دست آوردن این ایده، می‌گوییم دو برنامه‌ی خطی بیشینه‌سازی *معادل* (equivalent) هستند اگر برای هر جواب ممکن $\bar{x}$ برای L با مقدار هدف z، یک جواب ممکن متناظر $\bar{x}'$ برای L' با مقدار هدف z وجود داشته باشد، و برای هر جواب ممکن $\bar{x}'$ برای L' با مقدار هدف z، یک جواب ممکن متناظر $\bar{x}$ برای L با مقدار هدف z وجود داشته باشد. (این تعریف به یک تناظر یک به یک بین جواب‌های ممکن منجر نمی‌شود.) یک برنامه‌ی خطی کمینه‌سازی L و یک برنامه‌ی خطی بیشینه‌سازی L' معادل هستند

اگر برای هر جواب ممکن $\bar{x}$ برای L با مقدار هدف z، یک جواب ممکن متناظر $\bar{x}'$ برای L' با مقدار هدف $-z$ وجود داشته باشد، و برای هر جواب ممکن $\bar{x}'$ برای L' با مقدار هدف z، یک جواب ممکن متناظر $\bar{x}$ برای L با مقدار هدف $-z$.

اکنون نشان می‌دهیم که چگونه می‌توان مشکلات احتمالی در لیست بالا را یکی یکی حذف کرد. پس از حذف هر یک، بحث خواهیم کرد که برنامه‌ی خطی حاصل با برنامه‌ی خطی قبلی معادل است.

برای تبدیل یک برنامه‌ی خطی کمینه‌سازی L به یک برنامه‌ی خطی بیشینه‌سای معادل L'، به سادگی ضرایب تابع هدف را منفی می‌کنیم. از آن‌جایی که L و L' دارای مجموعه‌ی یکسانی از جواب‌های ممکن هستند، و برای هر جواب ممکن، مقدار هدف در L برابر است با منفی مقدار هدف در L'، این دو برنامه‌ی خطی با یکدیگر معادل هستند. برای مثال اگر برنامه‌ی خطی زیر را داشته باشیم:

$$\text{minimize} \quad -۲x_۱ + ۳x_۲$$

با شرایط

$$\begin{aligned} x_۱ + x_۲ &= ۷ \\ x_۱ - ۲x_۲ &\geq ۴ \\ x_۱ &\geq ۰ \end{aligned}$$

و ضرایب را در تابع هدف منفی کنیم، به دست می‌آوریم

$$\text{minimize} \quad ۲x_۱ - ۳x_۲$$

با شرایط

$$\begin{aligned} x_۱ + x_۲ &= ۷ \\ x_۱ - ۲x_۲ &\leq ۴ \\ x_۱ &\geq ۰ \end{aligned}$$

سپس نشان می‌دهیم که چگونه می‌توان برنامه‌ی خطی را که در آن بعضی از متغیرها محدودیت عدم منفی بودن ندارند، به یک برنامه‌ی خطی تبدیل کنیم که در آن تمام متغیرها محدودیت عدم منفی بودن دارند. فرض کنید یک متغیر x_j محدودیت عدم منفی بودن ندارد. در این صورت تمام x_jها را با $x_j'' - x_j'$ جایگزین می‌کنیم، و محدودیت‌های $x_j' \geq ۰$ و $x_j'' \geq ۰$ را اضافه می‌کنیم. بنابراین اگر تابع هدف یک جمله‌ی $c_j x_j$ داشته باشد، با عبارت $c_j x_j'' - c_j x_j'$ جایگزین می‌شود، و اگر محدودیت i حاوی جمله‌ی $a_{ij} x_j$ باشد، عبارت $a_{ij} x_j'' - a_{ij} x_j'$ جای آن را خواهد گرفت. هر جواب ممکن $\hat{x}$ به برنامه‌ی خطی جدید متناظر خواهد بود با یک جواب ممکن $\bar{x}$ به برنامه‌ی خطی اولیه، با $\hat{x}_j'' - \hat{x}_j' = \bar{x}_j$ و با مقدار هدف یکسان. همچنین، هر جواب ممکن $\bar{x}$ به برنامه‌ی خطی اولیه متناظر است با یک جواب ممکن $\hat{x}$ به برنامه‌ی خطی جدید با $\hat{x}_j' = \bar{x}_j$ و $\hat{x}_j'' = ۰$ اگر $\bar{x}_j \geq ۰$، یا با $\hat{x}_j'' = \bar{x}_j$ و $\hat{x}_j' = ۰$ اگر $\bar{x}_j < ۰$. دو برنامه‌ی خطی، مستقل از علامت $\bar{x}_j$، مقدار هدف یکسانی دارند، و بنابراین معادل خواهند بود. این روش تبدیل را برای تمام متغیرهایی که محدودیت عدم منفی

بودن ندارند، اعمال می‌کنیم تا به یک برنامه‌ی خطی معادل برسیم که در آن تمام متغیرها محدودیت عدم منفی بودن دارند.

با ادامه‌ی مثال، می‌خواهیم مطمئن شویم که تمام متغیرها دارای یک محدودیت عدم منفی بودن متناظر هستند. متغیر x_1 این محدودیت را دارد، ولی متغیر x_2 خیر. بنابراین x_2 را با دو متغیر x_2' و x_2'' جایگزین می‌کنیم، و برنامه‌ی خطی را اصلاح می‌کنیم تا به دست آوریم

$$\text{maximize} \quad 2x_1 - 3x_2' + 3x_2''$$

با شرایط

$$\begin{aligned} x_1 + x_2' - x_2'' &= 7 \\ x_1 - 2x_2' + 2x_2'' &\le 4 \\ x_1, \quad x_2', \quad x_2'' &\ge 0 \end{aligned} \qquad (۲۲-۲۹)$$

سپس محدودیت‌های معادله‌ای را به محدویت‌های نامعادله‌ای تبدیل می‌کنیم. فرض کنید یک برنامه‌ی خطی یک محدودیت معادله‌ای به شکل $f(x_1, x_2, ..., x_n) = b$ داشته باشد. از آن جایی که $x = y$ اگر و تنها اگر هر دوی $x \ge y$ و $x \le y$ برقرار باشند، می‌توانیم این محدودیت معادله‌ای را با دو محدودیت نامعادله‌ای $f(x_1, x_2, ..., x_n) \le b$ و $f(x_1, x_2, ..., x_n) \ge b$ جایگزین کنیم. با جایگذاری این تبدیل برای هر یک از تساوی‌ها یک برنامه‌ی خطی خواهیم داشت که تمام محدودیت‌های آن به صورت نامعادله هستند.

نهایتاً می‌توانیم با ضرب کردن محدودیت‌ها در ۱-، علامت بزرگ‌تر مساوی را در آن‌ها با علامت کوچک‌تر مساوی جایگزین کنیم. یعنی، هر نامساوی به شکل

$$\sum_{j=1}^{n} a_{ij} x_j \ge b_i$$

معادل است با یک نامساوی به شکل

$$\sum_{j=1}^{n} -a_{ij} x_j \le -b_i$$

بنابراین با جایگزینی هر یک از ضرایب a_{ij} با $-a_{ij}$ و هر مقدار b_i با $-b_i$، یک محدودیت معادل با علامت کوچک‌تر مساوی به دست می‌آوریم.

برای اتمام مثال بالا، معادله‌ی محدودیت (۲۹-۲۲) را با دو نامساوی جایگزین می‌کنیم، که به دست می‌دهد

$$\text{maximize} \quad 2x_1 - 3x_2' + 3x_2''$$

با شرایط

$$\begin{aligned} x_1 + x_2' - x_2'' &\le 7 \\ x_1 + x_2' - x_2'' &\ge 7 \\ x_1 - 2x_2' + 2x_2'' &\le 4 \\ x_1,\quad x_2',\quad x_2'' &\ge 0 \end{aligned} \qquad (۲۹\text{-}۲۳)$$

نهایتاً محدودیت (۲۹-۲۳) را منفی می‌کنیم. برای هماهنگی در نام متغیرها، متغیر x_2' را با x_2 و متغیر x_2'' را با x_3 جایگزین می‌کنیم، که شکل استاندارد

$$\text{maximize} \quad 2x_1 - 3x_2 + 3x_3 \qquad (۲۹\text{-}۲۴)$$

با شرایط

$$x_1 + x_2 - x_3 \le 7 \qquad (۲۹\text{-}۲۵)$$

$$-x_1 - x_2 + x_3 \le -7 \qquad (۲۹\text{-}۲۶)$$

$$x_1 - 2x_2 + 2x_3 \le 4 \qquad (۲۹\text{-}۲۷)$$

$$x_1,\quad x_2,\quad x_3 \ge 0 \qquad (۲۹\text{-}۲۸)$$

را به ما می‌دهد.

تبدیل برنامه‌های خطی به شکل ضعیف

برای حل بهینه‌ی یک برنامه‌ی خطی به کمک الگوریتم سیمپلکس، بهتر است آن را به شکلی توصیف کنیم که در آن بعضی از محدودیت‌ها به صورت تساوی باشند. به صورت دقیق‌تر، آن را به شکلی تبدیل می‌کنیم که در آن محدودیت‌های عدم منفی بودن، تنها محدودیت‌های به شکل نامساوی هستند، و بقیه‌ی محدودیت‌ها به معادله‌ای هستند. فرض کنید

$$\sum_{j=1}^{n} a_{ij} x_j \le b_i \qquad (۲۹\text{-}۲۹)$$

یک محدودیت نامعادله‌ای باشد. یک متغیر جدید s تعریف و نامساوی (۲۹-۲۹) را به صورت دو محدودیت

$$s = b_i - \sum_{j=1}^{n} a_{ij} x_i \qquad (۲۹\text{-}۳۰)$$

$$s \ge 0 \qquad (۲۹\text{-}۳۱)$$

بازنویسی می‌کنیم. به s یک *متغیر ضعیف* می‌گوییم، چرا که *ضعف*، یا تفاوت میان سمت چپ و سمت راست عبارت (۲۹-۲۹) را تعیین می‌کند. (به زودی خواهیم دید که چرا ترجیح می‌دهیم محدودیت‌ها را طوری بنویسیم که فقط متغیر ضعیف در سمت چپ آن‌ها باشد.) چون نامساوی (۲۹-۲۹) درست است اگر و فقط اگر هر دو تساوی (۲۹-۳۰) و نامساوی (۲۹-۳۱) درست باشند، می‌توانیم این تبدیل را برای هر یک از محدودیت‌های نامعادله‌ای یک برنامه‌ی خطی به کار ببریم، که یک برنامه‌ی خطی

معادل به دست می‌دهد که در آن تنها محدودیت‌های معادله‌ای، محدودیت‌های عدم منفی بودن هستند. هنگام تبدیل فرم استاندارد به فرم ضعیف، از x_{n+i} (به جای s) برای نشان دادن متغیرهای ضعیف مربوط به iامین نامساوی استفاده خواهیم کرد. بنابراین iامین محدودیت عبارت خواهد بود از

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j \qquad (۳۲-۲۹)$$

به همراه محدودیت عدم منفی بودن $x_{n+i} \geq 0$.

با به کار بردن این تبدیل برای هر یک از محدودیت‌ها در یک برنامه‌ی خطی به شکل استاندارد، یک برنامه‌ی خطی با یک شکل متفاوت به دست خواهیم آورد. به عنوان مثال، برای برنامه‌ی خطی توصیف شده در (۲۹-۲۴)-(۲۹-۲۸) متغیرهای ضعیف x_4، x_5، و x_6 را معرفی خواهیم کرد، که داریم

$$\text{maximize} \quad 2x_1 - 3x_2 + 3x_3 \qquad (۳۳-۲۹)$$

با شرایط

$$x_4 = 7 - x_1 - x_2 + x_3 \qquad (۳۴-۲۹)$$

$$x_5 = -7 + x_1 + x_2 - x_3 \qquad (۳۵-۲۹)$$

$$x_6 = 4 - x_1 + 2x_2 - 2x_3 \qquad (۳۶-۲۹)$$

$$x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \qquad (۳۷-۲۹)$$

در این برنامه‌ی خطی، تمام محدودیت‌ها به غیر از محدودیت‌های عدم منفی بودن به شکل نامساوی هستند، و هر متغیر یک محدودیت عدم منفی بودن دارد. هر یک از محدودیت‌های معادله‌ای را به صورتی می‌نویسیم که یکی از متغیرها در سمت چپ تساوی و بقیه در سمت راست آن قرار داشته باشند. به علاوه، متغیرهای سمت راست تمام تساوی‌ها با هم یکسان است، و همچنین این متغیرها تنها متغیرهایی هستند که در تابع هدف ظاهر می‌شوند. متغیرهای سمت چپ تساوی‌ها، *متغیرهای پایه* (basic variable) نام دارند، و متغیرهای سمت راست، *متغیرهای غیرپایه* (nonbasic variable).

برای برنامه‌های خطی که این شرایط را ارضا می‌کنند، بعضی مواقع از اصطلاحات «کمینه‌سازی» و «با شرایط» صرف نظر می‌کنیم، همچنین از ذکر صریح محدودیت‌های عدم منفی بودن. به علاوه از متغیر z برای نشان دادن مقدار تابع هدف استفاده خواهیم کرد. به شکل به دست آمده، *شکل ضعیف* می‌گوییم. اگر برنامه‌ی خطی داده شده در (۲۹-۳۳)-(۲۹-۳۷) را به صورت ضعیف بازنویسی کنیم، خواهیم داشت

$$z = 2x_1 - 3x_2 + 3x_3 \qquad (۳۸-۲۹)$$

$$x_4 = 7 - x_1 - x_2 + x_3 \qquad (۳۹-۲۹)$$

$$x_5 = -7 + x_1 + x_2 - x_3 \qquad (۴۰-۲۹)$$

$$x_6 = 4 - x_1 + 2x_2 - 2x_3 \qquad (۴۱-۲۹)$$

مانند شکل استاندارد، خوب است که یک نماد مختصرتر برای نشان دادن شکل ضعیف هم داشته

باشیم. همان طور که در بخش ۲۹-۳ خواهیم دید، مجموعه‌های متغیرهای پایه و متغیرهای غیرپایه حین اجرای الگوریتم سیمپلکس تغییر خواهند کرد. از N برای نشان دادن مجموعه‌ی اندیس‌های متغیرهای غیرپایه، و از B برای نشان دادن مجموعه‌ی اندیس‌های متغیرهای پایه استفاده خواهیم کرد. همیشه خواهیم داشت $|N| = n$، $|B| = m$، و $N \cup B = \{1,2,\ldots,n+m\}$. تساوی‌ها با ورودی‌های B، و متغیرهای سمت راست با ورودی‌های N اندیس‌گذاری خواهند شد. مانند شکل استاندارد، از b_i، c_j، و a_{ij} برای نشان دادن جمله‌های ثابت و ضرایب استفاده خواهیم کرد. همچنین از v برای نشان دادن یک جمله‌ی ثابت غیرضروری در تابع هدف استفاده می‌کنیم. (کمی بعد خواهیم دید که قرار دادن عبارت ثابت در تابع هدف تعیین مقدار آن را ساده‌تر می‌کند.) بنابراین می‌توانیم به صورت مختصر شکل ضعیف را به صورت چندتایی (N, B, A, b, c, v) توصیف کنیم، که نشان‌دهنده‌ی

$$z = v + \sum_{j \in N} c_j x_j \qquad (۲۹\text{-}۴۲)$$

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \qquad \text{برای } i \in B \qquad (۲۹\text{-}۴۳)$$

است، که در آن تمام متغیرهای x دارای محدودیت عدم منفی بودن هستند. چون سری $\sum_{j \in N} a_{ij} x_j$ را در (۲۹-۴۳) تفریق می‌کنیم، مقادیر a_{ij} در واقع منفی ضرایبی هستند که در شکل ضعیف «ظاهر می‌شوند».

به عنوان مثال، در شکل ضعیف

$$z = ۲۸ - \frac{x_۳}{۶} - \frac{x_۵}{۶} - \frac{۲x_۶}{۳}$$

$$x_۱ = ۸ + \frac{x_۳}{۶} + \frac{x_۵}{۶} - \frac{x_۶}{۳}$$

$$x_۲ = ۴ - \frac{۸x_۳}{۳} - \frac{۲x_۵}{۳} + \frac{x_۶}{۳}$$

$$x_۴ = ۱۸ - \frac{x_۳}{۲} + \frac{x_۵}{۲}$$

داریم $B = \{۱, ۲, ۴\}$، $N = \{۳, ۵, ۶\}$،

$$A = \begin{pmatrix} a_{۱۳} & a_{۱۵} & a_{۱۶} \\ a_{۲۳} & a_{۲۵} & a_{۲۶} \\ a_{۴۳} & a_{۴۵} & a_{۴۶} \end{pmatrix} = \begin{pmatrix} -۱/۶ & -۱/۶ & ۱/۳ \\ ۸/۳ & ۲/۳ & -۱/۳ \\ ۱/۲ & -۱/۲ & ۰ \end{pmatrix}$$

$$b = \begin{pmatrix} b_۱ \\ b_۲ \\ b_۴ \end{pmatrix} = \begin{pmatrix} ۸ \\ ۴ \\ ۱۸ \end{pmatrix}$$

$c = (c_۳ \;\; c_۵ \;\; c_۶)^T = (-۱/۶ \;\; -۱/۶ \;\; -۲/۳)^T$، و $v = ۲۸$. توجه کنید که اندیس‌های A، b، و c لزوماً مجموعه‌هایی از اعداد طبیعی متوالی نیستند؛ آن‌ها به مجموعه اندیس‌های B و N بستگی دارند. به عنوان یک مثال از حالتی که در آن ورودی‌های A منفی ضرایبی هستند که در شکل ضعیف ظاهر

می‌شوند، توجه کنید که تساوی مربوط به x_1 شامل جمله‌ی $x_3/6$ است، با این حال ضریب a_{13} به جای $1/6+$، $1/6-$ است.

تمرین‌ها

۲۹-۱-۱ اگر برنامه‌ی خطی توصیف شده در (۲۹-۲۴)-(۲۹-۲۸) را به شکل مختصر (۲۹-۱۹)-(۲۹-۲۱) بنویسیم، n، m، A، b، و c چه خواهند بود؟

۲۹-۱-۲ سه جواب ممکن برای برنامه‌ی خطی داده شده در (۲۹-۲۴)-(۲۹-۲۸) بدهید. مقدار هدف هر یک از این جواب‌ها چقدر است؟

۲۹-۱-۳ برای شکل ضعیف (۲۹-۳۸)-(۲۹-۴۱)، مقادیر N، B، A، b، c، و v چگونه است؟

۲۹-۱-۴ برنامه‌ی خطی زیر را به شکل استاندارد تبدیل کنید:

$$\text{minimize } 2x_1 + 7x_2 + x_3$$

با شرایط

$$\begin{aligned} x_1 \quad\quad - x_3 &= 7 \\ 3x_1 + x_2 \quad\quad &\geq 24 \\ x_2 \quad\quad &\geq 0 \\ x_3 &\leq 0 \end{aligned}$$

۲۹-۱-۵ برنامه‌ی خطی زیر را به شکل ضعیف تبدیل کنید:

$$\text{minimize } 2x_1 \quad\quad - 6x_3$$

با شرایط

$$\begin{aligned} x_1 + x_2 - x_3 &\leq 7 \\ 3x_1 - x_2 \quad\quad &\geq 8 \\ -x_1 + 2x_2 + 2x_3 &\geq 0 \\ x_1, x_2, x_3 \quad\quad &\geq 0 \end{aligned}$$

متغیرهای پایه و غیرپایه کدامند؟

۲۹-۱-۶ نشان دهید که برنامه‌ی خطی زیر ناممکن است:

$$\text{minimize } 3x_1 - 2x_2$$

با شرایط

$$\begin{aligned} x_1 + x_2 &\leq 2 \\ -2x_1 - 2x_2 &\leq -10 \\ x_1, x_2 &\geq 0 \end{aligned}$$

۲۹-۱-۷ نشان دهید که برنامه‌ی خطی زیر بی‌کران است:

$$\text{minimize} \quad x_1 - x_2$$

با شرایط

$$\begin{aligned} -2x_1 + x_2 &\le -1 \\ -x_1 - 2x_2 &\le -2 \\ x_1, x_2 &\ge 0 \end{aligned}$$

۲۹-۱-۸ فرض کنید یک برنامه‌ی خطی کلی با n متغیر و m محدودیت داریم، و فرض کنید که آن را به شکل استاندارد تبدیل می‌کنیم. یک کران بالا برای تعداد متغیرها و محدودیت‌ها در برنامه‌ی خطی حاصل بدهید.

۲۹-۱-۹ یک مثال از یک برنامه‌ی خطی ارائه کنید که در آن ناحیه‌ی ممکن کران‌دار نیست، ولی مقدار هدف بهینه کران‌دار است.

۲۹-۲ فرمول‌بندی مسائل به صورت برنامه‌های خطی

با این که در این فصل بر روی الگوریتم سیمپلکس تمرکز خواهیم کرد، نکته‌ی مهم دیگر این است که بدانیم چه زمانی می‌توان یک مسئله را به یک برنامه‌ی خطی تبدیل کرد. وقتی یک مسئله به یک برنامه‌ی خطی با اندازه‌ی چندجمله‌ای تبدیل شد، می‌توان آن را به کمک متدهای بیضوی یا نقطه‌ی درونی در زمان چندجمله‌ای حل کرد. نرم‌افزارهای مختلفی هستند که می‌توانند برنامه‌های خطی را به صورت بهینه حل کنند، بنابراین وقتی که مسئله به صورت یک برنامه‌ی خطی توصیف شد، در عمل می‌توان آن را به کمک چنین نرم‌افزارهایی حل کرد.

مسائل برنامه‌ریزی خطی محسوس مختلفی را بررسی خواهیم کرد. با دو مسئله آغاز می‌کنیم که قبلاً آن‌ها را آموخته‌ایم: مسئله‌ی کوتاه‌ترین مسیرها از یک مبدأ (فصل ۲۴) و مسئله‌ی شار بیشینه (فصل ۲۶). سپس مسئله‌ی شار با کم‌ترین هزینه را مطرح خواهیم کرد. با این که یک الگوریتم با زمان چندجمله‌ای برای این مسئله وجود دارد که بر پایه‌ی برنامه‌ریزی خطی نیست، ولی آن را در این جا مطرح نخواهیم کرد. نهایتاً مسئله‌ی شار چند کالایی را توصیف خواهیم کرد، که تنها راه‌حل چندجمله‌ای شناخته شده برای آن بر مبنای برنامه‌ریزی خطی است.

وقتی در بخش شش مسائل گراف را حل می‌کردیم، از نمادهایی برای خصیصه‌ها استفاده کردیم، مانند $v.d$ و $(u,v).f$. ولی برنامه‌های خطی معمولاً از متغیرهای با زیرنویس (اندیس) به جای اشیائی با خصیصه‌های مرتبط استفاده می‌کنند. بنابراین وقتی از متغیرها در برنامه‌های خطی استفاده می‌کنیم، رأس‌ها و یال‌ها را با استفاده از زیرنویس‌ها مشخص می‌کنیم. برای مثال وزن کوتاه‌ترین مسیر را برای رأس v با d_v نشان می‌دهیم، نه با $v.d$. به طور مشابه شار از رأس u به رأس v را با f_{uv} نشان می‌دهیم، نه با $(u,v).f$. برای نامساوی‌هایی که به صورت ورودی به مسئله‌ها داده شده‌اند، مانند وزن یا ظرفیت یال‌ها، همچنان از نمادهایی مانند $w(u,v)$ و $c(u,v)$ استفاده خواهیم کرد.

کوتاه‌ترین مسیرها

مسئله‌ی کوتاه‌ترین مسیرها از یک مبدأ را می‌توان به صورت یک برنامه‌ی خطی توصیف کرد. در این بخش بر روی فرمول‌بندی مسئله‌ی کوتاه‌ترین مسیر بین یک جفت رأس به صورت یک برنامه‌ی خطی تمرکز خواهیم کرد، و گسترش آن به مسئله‌ی کوتاه‌ترین مسیرها از یک مبدأ را به صورت تمرین ۲۹-۲-۳ به خواننده واگذار می‌کنیم.

در مسئله‌ی کوتاه‌ترین مسیر بین یک جفت رأس، به ما یک گراف جهت‌دار و وزن‌دار $G = (V, E)$ با تابع وزن $w : E \to \mathbb{R}$ که یال‌ها را به وزن‌های حقیقی مقدار نگاشت می‌کند، یک رأس مبدأ s، و یک رأس مقصد t داده شده است. می‌خواهیم مقدار d_t را بیابیم، که برابر است با وزن یک کوتاه‌ترین مسیر از s به t. برای توصیف این مسئله به صورت یک برنامه‌ی خطی، باید مجموعه‌ای از متغیرها و محدودیت‌ها توصیف کنیم که تعیین می‌کنند در چه صورت یک کوتاه‌ترین مسیر از s به t خواهیم داشت. خوشبختانه الگوریتم بلمن-فورد دقیقاً همین کار را انجام می‌دهد. وقتی الگوریتم بلمن-فورد پایان می‌یابد، برای هر رأس v یک مقدار d_v (در این جا از نماد زیرنویس به جای نماد خصیصه‌ها استفاده می‌کنیم) محاسبه کرده‌است به طوری که برای هر یال $(u,v) \in E$ داریم $d_v \le d_u + w(u,v)$. رأس مبدأ در ابتدا مقدار $d_s = 0$ را به خود می‌گیرد، که هیچ گاه تغییر نمی‌کند. بنابراین برنامه‌ی خطی زیر را برای محاسبه‌ی وزن کوتاه‌ترین مسیر از s به t خواهیم داشت:

$$\text{minimize} \quad d_t \tag{۲۹-۴۴}$$

با شرایط

$$d_v \le d_u + w(u,v) \qquad \text{برای هر یال } (u,v) \in E \tag{۲۹-۴۵}$$

$$d_s = 0 \tag{۲۹-۴۶}$$

ممکن است تعجب کنید که چرا این برنامه‌ی خطی یک تابع هدف را بیشینه می‌کند، در حالی که باید کوتاه‌ترین مسیرها را بیابد. نباید تابع هدف را کمینه کنیم، چرا که در این صورت با قرار دادن $\bar{d}_v = 0$ برای تمام $v \in V$ به یک جواب بهینه به برنامه‌ی خطی می‌رسیم، بدون این که مسئله‌ی کوتاه‌ترین مسیرها را حل کرده باشیم. بیشینه‌سای را از این رو انجام می‌دهیم که یک جواب بهینه به مسئله‌ی کوتاه‌ترین مسیرها هر یک از $\bar{d}_v$ ها را برابر با $\min_{u:(u,v)\in E}\{\bar{d}_u + w(u,v)\}$ قرار می‌دهد، به طوری که $\bar{d}_v$ بزرگ‌ترین مقداری است که کوچک‌تر یا مساوی تمام مقادیر مجموعه‌ی $\{\bar{d}_u + w(u,v)\}$ خواهد بود. می‌خواهیم $\bar{d}_v$ را برای تمام رأس‌های v روی یک کوتاه‌ترین مسیر از s به t با وجود شرایطی روی تمام رأس‌های v بیشینه کنیم، و بیشینه کردن d_t ما را به این هدف می‌رساند.

در این برنامه‌ی خطی، $|V|$ متغیر d_v وجود دارد، یکی برای هر رأس $v \in V$. تعداد $|E|+1$ محدودیت خواهیم داشت، یکی برای هر یال به علاوه‌ی یک محدودیت که مقدار مبدأ باید همیشه ۰ باشد.

شار بیشینه

مسئله‌ی شار بیشینه را هم می‌توان به صورت یک برنامه‌ی خطی توصیف کرد. به خاطر بیاورید که در این مسئله یک گراف جهت‌دار $G = (V, E)$ داریم که در آن هر یال $(u, v) \in E$ یک ظرفیت نامنفی $c(u, v) \geq 0$ دارد، به علاوه‌ی دو رأس مجزا؛ یک مبدأ s و یک چاهک t. همان طور که در بخش ۲۶-۱ تعریف شد، یک شار، یک تابع حقیقی مقدار $f : V \times V \to \mathbb{R}$ است که سه خصوصیت را ارضا می‌کند: محدودیت ظرفیت، تقارن اریب، و بقای شار. یک شار بیشینه، شاری است که این شرایط را ارضا می‌کند و علاوه بر آن مقدار شار را هم بیشینه می‌کند، که برابر است با کل شار خروجی از منبع. بنابراین یک شار محدودیت‌های خطی را ارضا می‌کند، و مقدار یک شار یک تابع خطی است. با در نظر گرفتن این که $c(u, v) = 0$ اگر $(u, v) \notin E$، می‌توانیم مسئله‌ی شار بیشینه را به صورت برنامه‌ی خطی زیر توصیف کنیم:

$$\text{maximize} \quad \sum_{v \in V} f_{sv} - \sum_{v \in V} f_{vs} \tag{۲۹-۴۷}$$

با شرایط

$$f_{uv} \leq c(u, v) \qquad \text{برای هر } u, v \in V \tag{۲۹-۴۸}$$

$$\sum_{v \in V} f_{vu} = \sum_{v \in V} f_{uv} \qquad \text{برای هر } u, v \in V \tag{۲۹-۴۹}$$

$$f_{uv} \geq 0 \qquad \text{برای هر } u \in V - \{s, t\} \tag{۲۹-۵۰}$$

این برنامه‌ی خطی $|V|^2$ متغیر دارد، که متناظرند با شار میان هر جفت از رأس‌ها، به علاوه‌ی $|V|^2 + |V| - 2$ محدودیت.

معمولاً حل برنامه‌های خطی با اندازه‌ی کوچک‌تر ساده‌تر است. در برنامه‌ی خطی داده شده در (۲۹-۴۷)–(۲۹-۵۰)، برای درک ساده‌تر یک شار و ظرفیت ۰ برای هر جفت از رأس‌های u و v با $(u, v) \notin E$ در نظر گرفته شده است. بهینه‌تر خواهد بود که برنامه‌ی خطی را طوری بازنویسی کنیم که $O(V + E)$ محدودیت داشته باشد. تمرین ۲۹-۲-۵ از شما می‌خواهد این کار را انجام دهید.

شار با کم‌ترین هزینه

در این بخش از برنامه‌ریزی خطی استفاده کردیم، و مسائلی را حل کردیم که قبلاً برای آن‌ها الگوریتم‌های کارامدی داشتیم. در واقع یک الگوریتم بهینه که برای حل یک مسئله طراحی شده است، مانند الگوریتم Dijkstra برای مسئله‌ی کوتاه‌ترین مسیرها از یک مبدأ، و یا متد راندن-برچسب‌دهی مجدد برای شار بیشینه، معمولاً از برنامه‌ریزی خطی بهینه‌تر خواهند بود، چه در تئوری و چه در عمل.

قدرت واقعی برنامه‌ریزی خطی در حل مسائل جدید است. مسئله‌ی پیش روی سیاست‌مدار را که در ابتدای این فصل مطرح شد به خاطر بیاورید. مسئله‌ی به دست آوردن رأی به تعداد کافی، بدون صرف هزینه‌ی بیش از اندازه را به کمک هیچ یک از الگوریتم‌های داده شده در این کتاب نمی‌توان حل کرد، ولی به کمک برنامه‌ریزی خطی می‌توان این کار را کرد. کتاب‌ها مملو از چنین مسائلی در

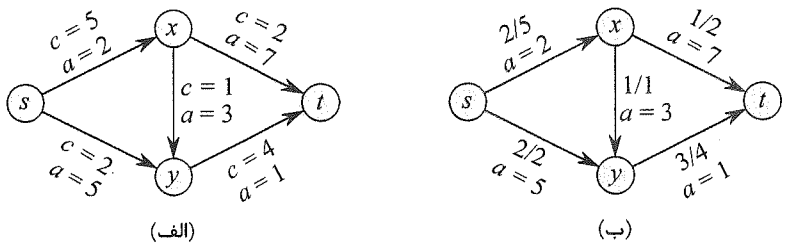
دنیای واقعی هستند که برنامه‌ریزی خطی می‌تواند آن‌ها را حل کند. به علاوه، برنامه‌ریزی خطی برای حل عملی نسخه‌هایی از مسائلی که هنوز یک جواب بهینه برای حل آن‌ها در دسترس نیست، مفید است.

به عنوان مثال، تعمیم زیر را از مسئله‌ی شار بیشینه در نظر بگیرید. فرض کنید هر یال (u,v) علاوه بر یک ظرفیت $c(u,v)$، یک هزینه‌ی حقیقی مقدار $a(u,v)$ هم دارند. مانند مسئله‌ی شار بیشینه فرض می‌کنیم که $c(u,v)=0$ اگر $(u,v)\notin E$، و یال‌های موازی با جهت مخالف در گراف وجود ندارند. اگر f_{uv} واحد شار از یال (u,v) عبور کند، باید $a(u,v)f_{uv}$ هزینه بپردازیم. همچنین یک شار هدف d داریم. می‌خواهیم d واحد شار از s به t بفرستیم، به طوری که کل هزینه‌ای که متحمل می‌شویم، یعنی $\sum_{(u,v)\in E} a(u,v)f_{uv}$، کمینه شود. این مسئله به *مسئله‌ی شار با هزینه‌ی کمینه* معروف است.

شکل ۲۹-۳ (الف) یک مثال از مسئله‌ی شار با هزینه‌ی کمینه را نشان می‌دهد. می‌خواهیم با تحمل کم‌ترین هزینه‌ی ممکن ۴ واحد شار از s به t بفرستیم. هر شار عملی مجاز، که یک تابع f است که شرایط (۲۹-۴۸)-(۲۹-۴۹) را ارضا می‌کند، هزینه‌ی کلی $\sum_{(u,v)\in E} a(u,v)f_{uv}$ را خواهد داشت. می‌خواهیم شار ۴ واحدی خاصی را بیابیم که هزینه را حداقل می‌کند. یک جواب بهینه در شکل ۲۹-۳ (ب) داده شده است، و هزینه‌ی کلی آن برابر است با $\sum_{(u,v)\in E} a(u,v)f_{uv} = (2\times 2)+(5\times 2)+(3\times 1)+(7\times 1)+(1\times 3) = 27$.

الگوریتم‌های چندجمله‌ای وجود دارند که به طور خاص برای حل مسئله‌ی شار با کم‌ترین هزینه طراحی شده‌اند، ولی این الگوریتم‌ها از حوصله‌ی این کتاب خارج‌اند. ولی می‌توانیم این مسئله را به صورت یک برنامه‌ی خطی توصیف کنیم. این برنامه‌ی خطی مشابه برنامه‌ی خطی داده شده برای مسئله‌ی شار بیشینه است، با این محدودیت اضافی که مقدار شار باید دقیقاً d واحد باشد، و تابع هدف جدید که هزینه را کمینه می‌کند:

$$\text{minimize} \quad \sum_{(u,v)\in E} a(u,v)f_{uv} \qquad (۲۹-۵۱)$$



شکل ۲۹-۳ (الف) یک مثال از مسئله‌ی شار با کم‌ترین هزینه. ظرفیت‌ها را با c و هزینه‌ها را با a نشان می‌دهیم. رأس s مبدأ و رأس t چاهک است، و می‌خواهیم ۴ واحد شار را از s به t منتقل کنیم. (ب) یک جواب برای مسئله‌ی شار با کم‌ترین هزینه که در آن ۴ واحد شار از s به t منتقل شده است. برای هر یال، شار و ظرفیت به صورت ظرفیت/شار نوشته شده است.

با شرایط

$$\begin{aligned} f_{uv} &\le c(u,v) && \text{برای هر } u,v \in V \\ \sum_{v\in V} f_{vu} - \sum_{v\in V} f_{uv} &= 0 && \text{برای هر } u \in V - \{s,t\} \\ \sum_{v\in V} f_{sv} - \sum_{v\in V} f_{vs} &= d \\ f_{uv} &\ge 0 && \text{برای هر } u,v \in V \end{aligned} \qquad (۵۲-۲۹)$$

شار چند کالا

به عنوان آخرین مثال یک مسئله‌ی شار دیگر را در نظر می‌گیریم. فرض کنید کـه کمپـانی Lucky Puck که در بخش ۲۶-۱ توصیف شد، تصمیم می‌گیرد خطوط تولید خود را متنوع کنـد و عـلاوه بـر گـوی هاکی، چوب و کلاه هاکی هم تولید کند. هر قطعه‌ی هاکی در کارخانـه‌ی مخصـوص بـه خـود تولیـد می‌شود، انبار مخصوص به خود را دارد، و هر روز باید از کارخانه به انبـار منتقـل شـود. چـوب‌هـای هاکی در Vancouver تولید می‌شوند و باید به Saskatoon منتقل شوند، و کـلاه‌هـا در Edmonton تولیـد شده و باید به Regina منتقل شوند. با این حال ظرفیت شبکه‌های انتقال تغییـر نمـی‌کنـد، و*کـالا* هـای گوناگون باید از یک شبکه‌ی مشترک استفاده کنند.

این مثال، نمونه‌ای از*مسئله‌ی شار چند کالایی* است. در این مسئله هم به ما یـک گـراف جهـت‌دار $G = (V, E)$ داده شده است که در آن هر یـال $(u,v) \in E$ دارای ظرفیـت نـامنفی $c(u,v) \ge 0$ اسـت. مانند مسئله‌ی شار بیشینه، به صورت ضمنی فـرض مـی‌کنـیم کـه $c(u,v) = 0$ بـرای $(u,v) \notin E$، و در گراف، یال‌های موازی و خلاف جهت وجود ندارد. به علاوه k کالای مختلف به ما داده شـده اسـت، $K_1, K_2, \ldots, K_k$، که در آن کالای i با سه‌تایی $K_i = (s_i, t_i, d_i)$ مشخص می‌شود. در ایـن جـا s_i مبدأ کالای i، t_i چاهک آن، و d_i مقداری از آن است که باید از s_i بـه t_i منتقـل شـود. یـک شـار از کالای i را، که با f_i نشان داده می‌شود (به طوری که f_{iuv} برابر است با شـار کـالای i از رأس u بـه رأس v) به صورت یک تابع حقیقی مقدار تعریف می‌کنیم که شرایط بقای شار و محـدودیت ظرفیـت را ارضا می‌کند. اکنون f_{uv}، یا*شار متراکم* (aggregate flow) را به صورت مجموع شار کالاهای مختلف تعریف می‌کنیم، به طوری که $f_{uv} = \sum_{i=1}^{k} f_{iuv}$. شار متراکم روی یال (u,v) نباید بیشتر از ظرفیـت آن باشد. به شکلی که این مسئله توصیف شده است، هیچ چیزی برای کمینه کردن وجود نـدارد؛ فقـط باید تعیین کنیم که آیا یافتن چنین شاری ممکن است یا خیر. بنابراین یک برنامه‌ی خطی با یـک تـابع هدف «پوچ» می‌نویسیم:

minimize 0

با شرایط

$$\sum_{i=1}^{k} f_{iuv} \le c(u,v) \qquad \text{برای هر } u,v \in V$$

$$\sum_{v\in V} f_{iuv} - \sum_{v\in V} f_{ivu} = 0 \qquad \text{برای هر } i = 1,2,\ldots,k \text{ و هر } u \in V - \{s_i, t_i\}$$

$$\sum_{v\in V} f_{i,s_i,v} - \sum_{v\in V} f_{i,v,s_i} = d_i \qquad \text{برای هر } i = 1,2,\ldots,k$$

$$f_{iuv} \ge 0 \qquad \text{برای هر } u,v \in V \text{ و هر } i = 1,2,\ldots,k$$

تنها راه حل چندجمله‌ای شناخته شده برای این مسئله این است که آن را به صورت یک برنامه‌ی خطی توصیف کرده و سپس به کمک یک الگوریتم چندجمله‌ای برنامه‌ریزی خطی آن را حل کنیم.

تمرین‌ها

۲۹-۲-۱ برنامه‌ی خطی مربوط به مسئله‌ی کوتاه‌ترین مسیر بین یک جفت رأس داده شده در (۲۹-۴۴)-(۲۹-۴۶) را به صورت استاندارد تبدیل کنید.

۲۹-۲-۲ برنامه‌ی خطی مربوط به مسئله‌ی یافتن کوتاه‌ترین مسیر از گره‌ی s به گره‌ی y در شکل ۲۴-۲ (الف) را بنویسید.

۲۹-۲-۳ در مسئله‌ی کوتاه‌ترین مسیرها از یک مبدأ، می‌خواهیم وزن کوتاه‌ترین مسیرها از یک رأس مبدأ s را به تمام رأس‌های $v \in V$ بیابیم. با داشتن یک گراف G، یک برنامه‌ی خطی بنویسید که جواب آن این خصوصیت را دارد که d_v وزن کوتاه‌ترین مسیر از s به v برای هر $v \in V$ است.

۲۹-۲-۴ برنامه‌ی خطی مربوط به یافتن شار بیشینه در شکل ۲۶-۱ (الف) را با جزئیات کامل بنویسید.

۲۹-۲-۵ برنامه‌ی خطی مربوط به شار بیشینه‌ی (۲۹-۴۷)-(۲۹-۵۰) را طوری بازنویسی کنید که فقط از $O(V+E)$ محدودیت استفاده کند.

۲۹-۲-۶ یک برنامه‌ی خطی بنویسید که با دریافت یک گراف دوبخشی $G = (V, E)$، مسئله‌ی تطابق دوبخشی بیشینه را بر روی آن حل می‌کند.

۲۹-۲-۷ *در مسئله‌ی شار چند کالایی با کم‌ترین هزینه*، به ما یک گراف جهت‌دار $G = (V, E)$ داده شده است که در آن هر یال $(u,v) \in E$ یک ظرفیت نامنفی $c(u,v) \ge 0$، و یک هزینه‌ی $a(u,v)$ دارد. مانند مسئله‌ی شار چند کالایی، به ما k کالای مختلف $K_1, K_2, \ldots, K_k$ داده شده است، که در آن کالای i به وسیله‌ی سه تایی $K_i = (s_i, t_i, d_i)$ مشخص می‌شود. شار f_i برای کالای i و شار متراکم f_{uv} بر روی یال (u,v) را مانند مسئله‌ی شار چند کالایی تعریف می‌کنیم. یک شار ممکن، شاری است که در آن شار متراکم بر روی هر

یال (u,v) از ظرفیت آن بیشتر نشود. هزینه‌ی یک شار برابر است با $\sum_{u,v \in V} a(u,v) f_{uv}$، و هدف یافتن یک شار با حداقل هزینه است. این مسئله را به صورت یک برنامه‌ی خطی توصیف کنید.

۲۹-۳ الگوریتم سیمپلکس

الگوریتم سیمپلکس، متد کلاسیک حل برنامه‌های خطی است. بر خلاف اکثر الگوریتم‌های دیگر این کتاب، زمان اجرای این الگوریتم در بدترین حالت از مرتبه‌ی چندجمله‌ای نیست. با این حال به بینشی مناسب در برنامه‌های خطی منجر می‌شود، و معمولاً در عمل بسیار سریع است.

الگوریتم سیمپلکس علاوه بر داشتن یک ترجمه‌ی هندسی، که قبل‌تر در همین فصل توصیف شد، شباهت‌هایی هم با حذف گاوسی دارد، که در بخش ۲۸-۱ در مورد آن بحث کردیم. حذف گاوسی با سیستمی از معادلات خطی شروع می‌شود که جواب آن‌ها را نمی‌دانیم. در هر تکرار، این سیستم را به صورتی معادل بازنویسی می‌کنیم که ساختارهایی جدید دارد. پس از تعدادی تکرار، سیستم طوری بازنویسی شده است که یافتن جواب آن بسیار ساده است. الگوریتم سیمپلکس به روشی مشابه عمل می‌کند، و می‌توانیم به آن به چشم یک حذف گاوسی برای نامساوی‌ها نگاه کنیم.

اکنون ایده‌ی اصلی پشت یک تکرار الگوریتم سیمپلکس را توضیح می‌دهیم. هر تکرار، یک «جواب پایه»ی مربوطه دارد که به راحتی از روی فرم ضعیف برنامه‌ی خطی قابل تشخیص است: هر متغیر غیرپایه را با ۰ مقداردهی و مقدار متغیرهای پایه را از روی محدودیت‌های معادله‌ای محاسبه می‌کنیم. از نظر جبری، هر تکرار یک فرم ضعیف را به یک فرم ضعیف دیگر تبدیل می‌کند. مقدار هدف جواب ممکن پایه کم‌تر از همین مقدار در تکرار قبل نخواهد بود (بلکه معمولاً بزرگ‌تر خواهد بود). برای دست‌یابی به این افزایش در مقدار هدف، یک متغیر غیرپایه انتخاب می‌کنیم به طوری که اگر بخواهیم مقدار آن متغیر را از ۰ افزایش دهیم، آن گاه مقدار هدف هم افزایش یابد. میزان افزایشی که می‌توانیم بر روی متغیر اعمال کنیم توسط محدودیت‌های دیگر مشخص می‌شود. به طور خاص، آن را آن قدر افزایش می‌دهیم تا یک متغیر پایه ۰ شود. سپس فرم ضعیف را بازنویسی می‌کنیم، که باعث می‌شود نقش متغیرهای پایه و متغیر غیرپایه‌ی انتخاب شده جابه‌جا شود. با این که از یک مقداردهی خاص متغیرها برای پیش بردن الگوریتم استفاده کردیم، و از آن در اثبات‌های خود هم استفاده می‌کنیم، الگوریتم به طور صریح این جواب را حفظ نمی‌کند، بلکه به سادگی برنامه‌ی خطی را آن قدر بازنویسی می‌کند که جواب «واضح» شود.

یک مثال از الگوریتم سیمپلکس

برنامه‌ی خطی زیر را که در شکل استاندارد است در نظر بگیرید:

$$\text{maximize} \quad 3x_1 + x_2 + 2x_3 \qquad (۲۹-۵۳)$$

با شرایط

$$x_1 + x_2 + 3x_3 \le 30 \quad (29\text{-}54)$$
$$2x_1 + 2x_2 + 5x_3 \le 24 \quad (29\text{-}55)$$
$$4x_1 + x_2 + 2x_3 \le 36 \quad (29\text{-}56)$$
$$x_1, \quad x_2, \quad x_3 \ge 0 \quad (29\text{-}57)$$

برای استفاده از الگوریتم سیمپلکس، ابتدا باید برنامه‌ی خطی را به شکل ضعیف تبدیل کنیم؛ نحوه‌ی انجام این کار را در بخش ۲۹-۱ دیدیم. تبدیل به فرم ضعیف علاوه بر اعمال جبری، یک مفهوم الگوریتمی مفید نیز هست. از بخش ۲۹-۱ به خاطر بیاورید که هر متغیر یک محدودیت عدم منفی بودن متناظر دارد. با توجه به این نکته، می‌گوییم یک محدودیت معادله‌ای برای یک مقداردهی خاص از متغیرهای غیرپایه‌ی آن، *نزدیک* (tight) است اگر این مقداردهی باعث شود که متغیر پایه‌ی آن محدودیت صفر شود. به طور مشابه، یک مقداردهی به متغیرهای غیرپایه که باعث شود متغیر پایه منفی شود، آن محدودیت را *نقض* می‌کند. بنابراین متغیرهای ضعیف صریحاً تعیین می‌کنند که هر محدودیت چقدر با نزدیک بودن فاصله دارد، و بنابراین کمک می‌کنند تشخیص دهیم که چقدر می‌توانیم مقادیر غیرپایه را بدون نقض هیچ محدودیتی افزایش دهیم.

با به کار بردن متغیرهای x_4، x_5، و x_6 به ترتیب برای نامساوی‌های (۲۹-۵۴)-(۲۹-۵۶)، و تبدیل برنامه‌ی خطی به شکل ضعیف خواهیم داشت

$$z = 3x_1 + x_2 + 2x_3 \quad (29\text{-}58)$$
$$x_4 = 30 - x_1 - x_2 - 3x_3 \quad (29\text{-}59)$$
$$x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \quad (29\text{-}60)$$
$$x_6 = 36 - 4x_1 - x_2 - 2x_3 \quad (29\text{-}61)$$

سیستم محدودیت‌های (۲۹-۵۸)-(۲۹-۶۱) سه تساوی و ۶ متغیر دارد. هر مقداردهی به متغیرهای x_1، x_2، و x_3 مقادیری برای x_4، x_5، و x_6 تعیین می‌کند؛ بنابراین بی‌نهایت جواب برای این سیستم معادلات وجود دارد. یک جواب، ممکن است اگر تمام متغیرهای x_1، x_2، ...، x_6 نامنفی باشند، و البته تعداد این جواب‌های ممکن بی‌نهایت است. بی‌نهایت بودن تعداد جواب‌های ممکن برای یک سیستم مانند بالا در اثبات‌های بعدی مفید خواهد بود. بر روی *جواب پایه* تمرکز خواهیم کرد: تمام متغیرهای (غیرپایه‌ی) سمت راست را برابر با ۰ قرار داده و سپس مقدار متغیرهای (پایه‌ی) سمت چپ را محاسبه می‌کنیم. در این مثال، جواب پایه برابر است با $(\bar{x}_1, \bar{x}_2, ..., \bar{x}_6) = (0, 0, 0, 30, 24, 36)$ با مقدار هدف $z = (3\times0) + (1\times0) + (2\times0) = 0$. توجه کنید که این جواب پایه برای هر $i \in B$ قرار می‌دهد $\bar{x}_i = b_i$. یک تکرار از الگوریتم سیمپلکس، مجموعه‌ی معادلات و تابع هدف را طوری بازنویسی می‌کند که مجموعه‌ای دیگر از متغیرها در سمت راست قرار گیرد. بنابراین جواب پایه‌ی مربوط به مسئله‌ی بازنویسی شده متفاوت خواهد بود. تأکید می‌کنیم که این بازنویسی به هیچ وجه مسئله‌ی برنامه‌ریزی خطی مربوطه را تغییر نمی‌دهد؛ مجموعه‌ی جواب‌های ممکن به مسئله‌ی مربوط

به هر تکرار دقیقاً برابر است با مجموعه‌ی جواب‌های ممکن برای مسئله‌ی مربوط به تکرار قبل. با این حال، جواب پایه‌ی مسئله با جواب پایه‌ی مسئله‌ی مربوط به تکرار قبل متفاوت خواهد بود.

اگر یک جواب پایه، یک جواب ممکن هم باشد، به آن یک *جواب پایه‌ی ممکن* می‌گوییم. حین اجرای الگوریتم سیمپلکس یک جواب پایه تقریباً همیشه یک جواب ممکن هم خواهد بود. با این حال در بخش ۲۹–۵ می‌بینیم که برای چند تکرار اول الگوریتم سیمپلکس، این احتمال وجود دارد که جواب پایه، ممکن نباشد.

در هر تکرار، هدف ما این است که برنامه‌ی خطی را طوری بازنویسی کنیم که مقدار هدف جواب پایه بزرگ‌تر از قبل باشد. یک متغیر غیرپایه‌ی x_e انتخاب می‌کنیم که ضریب آن در تابع هدف مثبت است، و مقدار x_e را تا حد ممکن افزایش می‌دهیم، بدون این که هیچ یک از محدودیت‌ها نقض شوند. متغیر x_e تبدیل به یک متغیر پایه، و یک متغیر دیگر x_l تبدیل به یک متغیر غیرپایه می‌شود. ممکن است مقدار متغیرهای پایه‌ی دیگر و یا تابع هدف هم تغییر کند.

برای ادامه‌ی مثال اجازه دهید افزایش مقدار x_1 را در نظر بگیریم. همان طور که x_1 را افزایش می‌دهیم، مقادیر x_4، x_5، و x_6 کاهش می‌یابند. از آن جایی که برای هر متغیر یک محدودیت عدم منفی بودن داریم، نباید اجازه دهیم مقدار هیچ یک از آن‌ها منفی شود. اگر x_1 از ۳۰ فراتر رود x_4 منفی می‌شود، و x_5 و x_6 زمانی منفی می‌شوند که مقدار x_1 به ترتیب از ۱۲ و ۹ بیشتر شود. محدودیت سوم (۲۹–۶۱) نزدیک‌ترین محدودیت است، و محدودیت افزایش x_1 را تعیین می‌کند. بنابراین نقش x_1 و x_6 را عوض می‌کنیم. تساوی (۲۹–۶۱) را نسبت به x_1 حل می‌کنیم و به دست می‌آوریم

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \qquad (۲۹–۶۲)$$

برای بازنویسی بقیه‌ی تساوی‌ها به طوری که x_6 در سمت راست باشد، x_1 را از تساوی (۲۹–۶۲) کم می‌کنیم. با انجام همین کار برای تساوی (۲۹–۵۹) به دست می‌آوریم

$$\begin{aligned} x_4 &= 30 - x_1 - x_2 - 3x_3 \\ &= 30 - (9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}) - x_2 - 3x_3 \\ &= 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \end{aligned} \qquad (۲۹–۶۳)$$

به طور مشابه می‌توانیم تساوی (۲۹–۶۲) را با محدودیت (۲۹–۶۰) و تابع هدف (۲۹–۵۸) ترکیب، و برنامه‌ی خطی را به صورت زیر بازنویسی کنیم:

$$z = 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \qquad (۲۹–۶۴)$$

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \qquad (۲۹–۶۵)$$

$$x_۴ = ۲۱ - \frac{۳x_۲}{۴} - \frac{۵x_۳}{۲} + \frac{x_۶}{۴} \qquad (۶۶-۲۹)$$

$$x_۵ = ۶ - \frac{۳x_۲}{۲} - ۴x_۳ + \frac{x_۶}{۲} \qquad (۶۷-۲۹)$$

به این عملیات یک *چرخش* (pivot) می‌گوییم. همان طور که در بالا مشخص شد، در یک چرخش یک متغیر غیرپایه‌ی x_e، که به آن *متغیر ورودی* (entering variable) می‌گوییم، و یک متغیر پایه‌ی x_l، که به آن *متغیر خروجی* (leaving variable) می‌گوییم، انتخاب شده و نقش آن‌ها عوض می‌شود.

برنامه‌ی خطی توصیف شده در (۲۹-۶۴)-(۲۹-۶۷) معادل است با برنامه‌ی خطی توصیف شده در تساوی‌های (۲۹-۵۸)-(۲۹-۶۱). اعمالی که در الگوریتم سیمپلکس اجرا می‌کنیم چیزی نیستند به جز بازنویسی تساوی‌ها به طوری که متغیرها بین سمت چپ و راست جابه‌جا شوند، و تفریق یک تساوی از تساوی دیگر. عملیات اول بدیهتاً یک مسئله‌ی معادل تولید می‌کند، و دومی، طبق جبر خطی اولیه، باز هم یک مسئله‌ی معادل تولید می‌کند. (تمرین ۲۹-۳-۳ را ببینید.)

برای نشان دادن این معادل بودن، مشاهده کنید که جواب پایه‌ی اولیه‌ی ما، یعنی $(۰, ۰, ۰, ۳۰, ۲۴, ۳۶)$، معادلات جدید (۲۹-۶۵)-(۲۹-۶۷) را ارضا می‌کنند، و مقدار هدف آن‌ها برابر است با $۲۷ + (۱/۴)\times ۰ + (۱/۲)\times ۰ - (۳/۴)\times ۳۶ = ۰$. جواب اولیه‌ی مربوط به مسئله‌ی برنامه‌ی خطی جدید، مقادیر غیرپایه را با صفر مقداردهی می‌کند، و برابر است با $(۹, ۰, ۰, ۲۱, ۶, ۰)$، با مقدار هدف $z = ۲۷$. با ریاضیات ساده می‌توان بررسی کرد که این جواب تساوی‌های (۲۹-۵۹)-(۲۹-۶۱) را هم ارضا می‌کند، و وقتی در تابع هدف (۲۹-۶۱) جایگذاری می‌شود، مقدار هدف $۲۷ = (۲\times ۰) + (۱\times ۰) + (۳\times ۹)$ را دارد.

با ادامه‌ی مثال، می‌خواهیم یک متغیر جدید بیابیم و مقدار آن را افزایش دهیم. نمی‌خواهیم $x_۶$ را افزایش دهیم چرا که با این کار مقدار هدف کاهش می‌یابد. می‌توانیم سعی کنیم $x_۲$ یا $x_۳$ را افزایش دهیم؛ $x_۳$ را انتخاب می‌کنیم. بدون نقض محدودیت‌ها، چقدر می‌توان $x_۳$ را افزایش داد؟ محدودیت‌های (۲۹-۶۵)، (۲۹-۶۶)، و (۲۹-۶۷) به ترتیب آن را به ۱۸، ۴۲/۵ و ۳/۲ محدود می‌کنند. باز هم سومین محدودیت نزدیک‌ترین آن‌ها است، و بنابراین محدودیت سوم را طوری بازنویسی می‌کنیم که $x_۳$ در سمت راست و $x_۵$ در سمت چپ آن قرار گیرد. سپس این تساوی جدید، یعنی $x_۳ = ۳/۲ - ۳x_۲/۸ - x_۵/۴ + x_۶/۸$ را در تساوی‌های (۲۹-۶۴)-(۲۹-۶۶) جایگذاری می‌کنیم و یک سیستم جدید، معادل سیستم قبلی به دست می‌آوریم:

$$z = \frac{۱۱۱}{۴} + \frac{x_۲}{۱۶} - \frac{x_۵}{۸} - \frac{۱۱x_۶}{۱۶} \qquad (۶۸-۲۹)$$

$$x_۱ = \frac{۳۳}{۴} - \frac{x_۲}{۱۶} + \frac{x_۵}{۸} - \frac{۵x_۶}{۱۶} \qquad (۶۹-۲۹)$$

$$x_۳ = \frac{۳}{۲} - \frac{۳x_۲}{۸} - \frac{x_۵}{۴} + \frac{x_۶}{۸} \qquad (۷۰-۲۹)$$

$$x_۴ = \frac{۶۹}{۴} + \frac{۳x_۲}{۱۶} + \frac{۵x_۵}{۸} - \frac{x_۶}{۱۶} \qquad (۷۱-۲۹)$$

جواب اولیه‌ی مربوط به این سیستم (۰, ۰, ۶۹/۴, ۳/۲, ۰, ۳۳/۴) است، با مقدار هدف ۱۱۱/۴. اکنون تنها راه افزایش مقدار هدف، افزایش x_2 است. سه محدودیت، به ترتیب کران‌های بالای ۱۳۲، ۴، و ∞ را به آن می‌دهند. (کران بالای ∞از محدودیت (۲۹-۷۱) به خاطر این است که وقتی x_2 را افزایش می‌دهیم، مقدار متغیر پایه‌ی x_4 هم افزایش می‌یابد. از این رو این محدودیت هیچ کرانی بر روی x_2 تعیین نمی‌کند.) x_2 را تا ۴ افزایش می‌دهیم، و x_2 غیرپایه می‌شود. سپس تساوی (۲۹-۷۰) را بر حسب x_2 حل، و مقدار آن را در تساوی‌های دیگر جایگزین می‌کنیم تا به دست آوریم

$$z = ۲۸ - \frac{x_3}{۶} - \frac{x_5}{۶} - \frac{۲x_6}{۳} \qquad (۲۹-۷۲)$$

$$x_1 = ۸ + \frac{x_3}{۶} + \frac{x_5}{۶} - \frac{x_6}{۳} \qquad (۲۹-۷۳)$$

$$x_2 = ۴ - \frac{۸x_3}{۳} - \frac{۲x_5}{۳} + \frac{x_6}{۳} \qquad (۲۹-۷۴)$$

$$x_4 = ۱۸ - \frac{x_3}{۲} + \frac{x_5}{۲} \qquad (۲۹-۷۵)$$

در این لحظه تمام ضرایب در تابع هدف منفی هستند. همان طور که بعداً در همین فصل خواهیم دید، این وضعیت فقط زمانی اتفاق می‌افتد که برنامه‌ی خطی را طوری بازنویسی کنیم که جواب اولیه، یک جواب بهینه هم باشد. بنابراین برای این مسئله جواب (۰, ۰, ۱۸, ۰, ۴, ۸) با مقدار هدف ۲۸ یک جواب بهینه است. اکنون به برنامه‌ی خطی اولیه‌ی خود در (۲۹-۵۳)-(۲۹-۵۷) بازمی‌گردیم. تنها متغیرها در برنامه‌ی خطی اولیه عبارتند از x_1، x_2، و x_3، و بنابراین جواب ما عبارت است از $x_1 = ۸$، $x_2 = ۴$، و $x_3 = ۰$، با مقدار هدف $۲۸ = (۲\times۰) + (۱\times۴) + (۳\times۸)$. توجه کنید که مقدار متغیرهای ضعیف در جواب نهایی نشان‌دهنده‌ی این است که در هر نامساوی به چه میزان اختلاف وجود دارد. متغیر ضعیف x_4 برابر است با ۱۸، و در نامساوی (۲۹-۵۴)، سمت چپ با مقدار $۱۲ = ۰+۴+۸$ مقدار ۱۸ واحد از سمت راست با مقدار ۳۰ کم‌تر است. متغیرهای x_5 و x_6 صفر هستند، و بنابراین در تساوی‌های (۲۹-۵۵) و (۲۹-۵۶)، سمت چپ و راست برابر هستند. همچنین مشاهده کنید که با این که ضرایب در فرم ضعیف اولیه عدد صحیح هستند، ضرایب در برنامه‌های خطی دیگر لزوماً عدد صحیح نیستند، و جواب‌های میانی هم به همین شکل. به علاوه نیازی نیست که جواب نهایی به برنامه‌ی خطی عدد صحیح باشد؛ این که در این مثال جواب عدد صحیح بوده است، کاملاً اتفاقی است.

چرخش

اکنون رویه‌ی چرخش را تشریح می‌کنیم. رویه‌ی PIVOT فرم ضعیف یک برنامه‌ی خطی را به صورت چند تایی (N, B, A, b, c, v)، یک اندیس l مربوط به متغیر خروجی x_l، و اندیس e مربوط به متغیر ورودی x_e را به عنوان ورودی دریافت می‌کند. مقدار بازگشتی این رویه چندتایی

$(\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v})$ است که فرم ضعیف جدید را توصیف می‌کند. (باز هم به خاطر بیاورید که ورودی‌های ماتریس‌های A و $\hat{A}$ با اندازه‌ی $m \times n$ در واقع منفی ضرایبی هستند که در شکل ضعیف ظاهر می‌شوند.)

```
PIVOT(N, B, A, b, c, v, l, e)
1  // Compute the coefficients of the equation for new basic variable x_e .
2  let Â be a new m×n matrix
3  b̂_e = b_l / a_le
4  for each j ∈ N − {e}
5      â_ej = a_lj / a_le
6  â_el = 1/a_le
7  // Compute the coefficients of the remaining constraints.
8  for each i ∈ B − {l}
9      b̂_i = b_i − a_ie b̂_e
10     for each j ∈ N − {e}
11         â_ij = a_ij − a_ie â_ej
12     â_il = −a_ie â_el
13 // Compute the objective function.
14 v̂ = v + c_e b̂_e
15 for each j ∈ N − {e}
16     ĉ_j = c_j − c_e â_ej
17 ĉ_l = −c_e â_el
18 // Compute new sets of basic and nonbasic variables.
19 N̂ = N − {e} ∪ {l}
20 B̂ = B − {l} ∪ {e}
21 return (N̂, B̂, Â, b̂, ĉ, v̂)
```

PIVOT به صورت زیر کار می‌کند. خطوط ۳-۶ با بازنویسی تساوی که x_l در سمت چپ آن است، به صورتی که x_e در سمت چپ آن باشد، ضرایب را در تساوی جدید برای x_e محاسبه می‌کنند. در خطوط ۸-۱۲ تساوی‌های دیگر با جایگذاری سمت راست این تساوی جدید به جای x_e، بازنویسی می‌شوند. خطوط ۱۴-۱۷ همین جایگذاری را برای تابع هدف انجام می‌دهند، و خطوط ۱۹ و ۲۰ مجموعه‌های متغیرهای پایه و غیرپایه را به هنگام‌سازی می‌کنند. خط ۲۱ فرم ضعیف جدید را بازمی‌گرداند. اگر $a_{le} = 0$، رویه‌ی PIVOT با یک خطای تقسیم بر صفر متوقف می‌شود، ولی همان طور که در اثبات لم‌های ۲۹-۲ و ۲۹-۱۲ خواهیم دید، PIVOT فقط زمانی فراخوانی می‌شود که $a_{le} \neq 0$.

اکنون یک جمع‌بندی بر روی تأثیر PIVOT بر روی مقدار متغیرها در جواب اولیه انجام می‌دهیم.

لم ۲۹-۱

یک فراخوانی PIVOT (N, B, A, b, c, v, l, e) را در نظر بگیرید، که در آن $a_{le} \neq 0$. فرض کنید این فراخوانی مقادیر $(\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v})$ را بازگرداند، و فرض کنید $\bar{x}$ نشان‌دهنده‌ی جواب اولیه بعد از فراخوانی باشد. در این صورت

۱. $\bar{x}_j = 0$ برای هر $j \in \hat{N}$.
۲. $\bar{x}_e = b_l / a_{le}$.
۳. $\bar{x}_i = b_i - a_{ie}\hat{b}_e$ برای هر $i \in \hat{B} - \{e\}$.

اثبات اولین عبارت صحیح است زیرا جواب اولیه همیشه تمام متغیرهای غیرپایه را برابر با ۰ قرار می‌دهد. وقتی که در یک محدودیت

$$x_i = \hat{b}_i - \sum_{j \in N} \hat{a}_{ij} x_j$$

تمام متغیرهای غیرپایه را با ۰ مقداردهی می‌کنیم، برای هر $i \in \hat{B}$ داریم $\bar{x}_i = \hat{b}_i$. از آن جایی که $e \in \hat{B}$، طبق خط ۳ رویه‌ی PIVOT، داریم

$$\bar{x}_e = \hat{b}_e = b_l / a_{le}$$

که عبارت دوم را اثبات می‌کند. به طور مشابه، با استفاده از خط ۸ برای هر $i \in \hat{B} - \{e\}$ داریم

$$\bar{x}_i = \hat{b}_i = b_i - a_{ie}\hat{b}_e$$

که عبارت سوم را اثبات می‌کند. ■

الگوریتم رسمی سیمپلکس

اکنون آماده هستیم که الگوریتم سیمپلکس را که در مثال نشان دادیم، فرمول‌بندی کنیم. آن مثال به طور خاص مثال پیچیده‌ای نبود، و ممکن بود به مشکلات بسیار دیگری برخورد کنیم:

- چطور می‌توان تشخیص داد که یک برنامه‌ی خطی دارای جواب ممکن است؟
- اگر یک برنامه‌ی خطی ممکن باشد ولی جواب پایه‌ی اولیه‌ی آن ممکن نباشد، چه باید بکنیم؟
- چطور می‌توان تشخیص داد که یک برنامه‌ی خطی بی‌کران است؟
- چطور می‌توان متغیرهای ورودی و خروجی را انتخاب کرد؟

در بخش ۲۹-۵ نحوه‌ی تشخیص ممکن بودن یک برنامه‌ی خطی را نشان خواهیم داد، و این که در صورت ممکن بودن برنامه‌ی خطی، چطور می‌توان یک فرم ضعیف برای آن یافت که جواب پایه‌ی اولیه‌ی آن، یک جواب ممکن هم باشد. بنابراین فرض می‌کنیم که یک رویه‌ی INITIALIZE-SIMPLEX (A, b, c) داریم که یک برنامه‌ی خطی به فرم استاندارد، یعنی یک ماتریس $A = (a_{ij})$ با اندازه‌ی m، یک بردار $b = (b_i)$ با اندازه‌ی m، و یک بردار $c = (c_j)$ با اندازه‌ی n را به عنوان ورودی دریافت می‌کند. اگر مسئله ناممکن باشد، این رویه یک پیغام بدین مضمون بازمی‌گرداند

و پایان می‌یابد. در غیر این صورت یک فرم ضعیف بازمی‌گرداند که جواب اولیه‌ی پایه‌ی آن، یک جواب ممکن هم هست.

رویه‌ی SIMPLEX یک برنامه‌ی خطی به شکل استاندارد را به عنوان ورودی دریافت می‌کند، به همان شکلی که در بالا گفته شد. این رویه یک بردار n تایی $\bar{x} = (\bar{x}_j)$ را بازمی‌گرداند که یک جواب بهینه برای برنامه‌ی خطی توصیف شده در (۲۹-۱۹)-(۲۹-۲۱) است.

```
SIMPLEX(A, b, c)
1  (N, B, A, b, c, v) = INITIALIZE-SIMPLEX(A, b, c)
2  let Δ be a new vector of length n
3  while some index j ∈ N has c_j > 0
4      choose an index e ∈ N for which c_e > 0
5      for each index i ∈ B
6          if a_ie > 0
7              Δ_i = b_i / a_ie
8          else Δ_i = ∞
9      choose an index l ∈ B that minimizes Δ_i
10     if Δ_l = ∞
11         return "unbounded"
12     else (N, B, A, b, c, v) = PIVOT(N, B, A, b, c, v, l, e)
13 for i = 1 to n
14     if i ∈ B
15         x̄_i = b_i
16     else x̄_i = 0
17 return (x̄_1, x̄_2, ..., x̄_n)
```

رویه‌ی SIMPLEX به صورت زیر کار می‌کند. در خط ۱، رویه‌ی INITIALIZE-SIMPLEX (A,b,c) که در بالا توصیف شد، فراخوانی می‌شود، که یا تشخیص می‌دهد که برنامه‌ی خطی ناممکن است و یا یک فرم ضعیف بازمی‌گرداند که جواب پایه‌ی آن یک جواب ممکن باشد. بخش اصلی الگوریتم در حلقه‌ی **while** خطوط ۳-۱۲ قرار دارد. اگر تمام ضرایب در تابع هدف منفی باشند، حلقه‌ی **while** پایان می‌یابد. در غیر این صورت، در خط ۴ یک متغیر x_e را که ضریب آن در تابع هدف مثبت است، به عنوان متغیر ورودی انتخاب می‌کنیم. با این که می‌توانیم هر متغیری با این خصوصیت در تابع هدف را به عنوان متغیر ورودی انتخاب کنیم، فرض می‌کنیم که یک قانون از پیش تعیین شده‌ی قطعی برای انجام این کار داریم. سپس در خطوط ۵-۹ هر محدودیت را چک می‌کنیم، و محدودیتی را انتخاب می‌کنیم که میزان افزایش x_e را بیش از بقیه محدود می‌کند؛ متغیر پایه‌ی مربوط به این محدودیت x_l است. دوباره ممکن است این آزادی را داشته باشیم که از میان چند متغیر یکی را به دلخواه به عنوان متغیر خروجی انتخاب کنیم، ولی فرض می‌کنیم که یک قانون از پیش تعیین شده‌ی قطعی برای این کار داریم. اگر هیچ یک از محدودیت‌ها میزان افزایش متغیر ورودی را محدود نمی‌کنند، الگوریتم "unbounded" را در خط ۱۱ بازمی‌گرداند. در غیر این صورت، خط ۱۲ با فراخوانی زیرروال PIVOT (N,B,A,b,c,v,l,e) که در بالا توصیف شد، نقش متغیرهای ورودی و خروجی

را عوض می‌کند. خطوط ۱۳-۱۶ با مقداردهی تمام متغیرهای غیرپایه با ۰ و هر متغیر پایه‌ی $\bar{x}_i$ با b_i، یک جواب برای متغیرهای $\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n$ مربوط به برنامه‌ی خطی اصلی محاسبه می‌کنند، و خط ۱۷ این مقادیر را بازمی‌گرداند.

برای نشان دادن این که SIMPLEX به درستی کار می‌کند، ابتدا نشان می‌دهیم که اگر SIMPLEX یک جواب ممکن اولیه داشته باشد و نهایتاً پایان یابد، در این صورت یا یک جواب ممکن را بازمی‌گرداند و یا تشخیص می‌دهد که برنامه‌ی خطی بی‌کران است. سپس نشان خواهیم داد که SIMPLEX پایان می‌یابد، و نهایتاً، در بخش ۲۹-۴ (قضیه‌ی ۲۹-۱۰) اثبات می‌کنیم که جواب بازگردانده شده بهینه است.

لم ۲۹-۹

فرض کنید یک برنامه‌ی خطی (A, b, c) داریم، و فرض کنید فراخوانی INITIALIZE-SIMPLEX در خط ۱ رویه‌ی SIMPLEX یک فرم ضعیف بازمی‌گرداند که جواب اولیه‌ی آن یک جواب ممکن است. در این صورت اگر SIMPLEX در خط ۱۷ یک جواب بازگرداند، آن جواب یک جواب ممکن برای برنامه‌ی خطی است. اگر SIMPLEX در خط ۱۱ خروجی "unbounded" را بازگرداند، در این صورت برنامه‌ی خطی بی‌کران است.

اثبات از ثابت حلقه‌ی سه بخشی زیر استفاده می‌کنیم:

- در آغاز هر تکرار حلقه‌ی **while** خطوط ۳-۱۲،

۱. فرم ضعیف فعلی معادل است با فرم ضعیف بازگردانده شده توسط فراخوانی INITIALIZE-SIMPLEX.

۲. برای هر $i \in B$ داریم $b_i \geq 0$، و

۳. جواب اولیه‌ی مربوط به فرم ضعیف، یک جواب ممکن است.

- *آغاز:* معادل بودن فرم‌های ضعیف برای اولین تکرار بدیهی است. طبق صورت لم، فرض می‌کنیم که فراخوانی INITIALIZE-SIMPLEX در خط ۱ رویه‌ی SIMPLEX یک فرم ضعیف بازمی‌گرداند که جواب اولیه‌ی آن یک جواب ممکن است. بنابراین قسمت سوم ثابت حلقه هم درست است. به علاوه، از آن جایی که هر متغیر پایه‌ی x_i در جواب اولیه با b_i مقداردهی شده است، و ممکن بودن جواب اولیه ایجاب می‌کند که هر متغیر پایه‌ی x_i نامنفی باشد، داریم $b_i \geq 0$. بنابراین قسمت دوم ثبات حلقه برقرار است.
- *ادامه:* نشان خواهیم داد که با فرض این که عبارت **return** در خط ۱۱ اجرا نمی‌شود، هر تکرار حلقه‌ی **while** ثابت حلقه را حفظ خواهد کرد. حالتی که در آن خط ۱۱ اجرا می‌شود را در بخش پایانی بررسی خواهیم کرد.

یک تکرار از حلقه‌ی **while** با فراخوانی رویه‌ی PIVOT نقش یک متغیر پایه و یک متغیر غیرپایه را

با یکدیگر عوض می‌کند. طبق تمرین ۲۹-۳-۳، فرم ضعیف فعلی معادل است با فرم ضعیف مربوط به تکرار قبلی، که طبق ثابت حلقه معادل است با فرم ضعیف اولیه.

اکنون قسمت دوم ثابت حلقه را ثابت خواهیم کرد. فرض می‌کنیم که در آغاز هر تکرار حلقه‌ی while، برای هر $i \in B$ داریم $b_i \geq 0$، و نشان خواهیم داد که این نامساوی‌ها بعد از فراخوانی PIVOT در خط ۱۲ هم برقرار خواهند بود. از آن جایی که تنها تغییر بر روی متغیرهای b_i و مجموعه‌ی B از متغیرهای پایه در این مقداردهی اتفاق می‌افتد، کافی است نشان دهیم که خط ۱۲، این بخش از ثابت حلقه را حفظ می‌کند. فرض می‌کنیم b_i، a_{ij}، و B نشان دهنده‌ی مقادیر قبل از فراخوانی PIVOT، و $\hat{b}_i$ نشان دهنده‌ی مقادیر بعد بازگردانده شده از PIVOT باشد.

ابتدا، مشاهده می‌کنیم که $\hat{b}_e \geq 0$ چرا که طبق ثابت حلقه، $b_l \geq 0$، طبق خط ۵ رویه‌ی SIMPLEX، $a_{le} > 0$، و طبق خط ۲ رویه‌ی PIVOT، $\hat{b}_e = b_l / a_{le}$.

برای اندیس‌های باقی‌مانده‌ی $i \in B - l$ داریم

$$\begin{aligned} \hat{b}_i &= b_i - a_{ie}\hat{b}_e && \text{(طبق خط ۹ رویه‌ی PIVOT)} \\ &= b_i - a_{ie}(b_l / a_{le}) && \text{(طبق خط ۳ رویه‌ی PIVOT)} \end{aligned} \qquad (۲۹\text{-}۷۶)$$

بسته به این که $a_{ie} > 0$ یا $a_{ie} \leq 0$ باید دو حالت را در نظر بگیریم. اگر $a_{ie} > 0$، آن گاه چون l را طوری انتخاب کرده‌ایم که

$$b_l / a_{le} \leq b_i / a_{ie} \qquad \text{برای هر } i \in B \qquad (۲۹\text{-}۷۷)$$

داریم

$$\begin{aligned} \hat{b}_i &= b_i - a_{ie}(b_l / a_{le}) && \text{(طبق تساوی (۲۹-۷۶))} \\ &\geq b_i - a_{ie}(b_i / a_{ie}) && \text{(طبق تساوی (۲۹-۷۷))} \\ &= b_i - b_i \\ &= 0 \end{aligned}$$

و بنابراین $\hat{b}_i \geq 0$. اگر $a_{ie} = 0$، آن گاه چون a_{le}، b_i، و b_l همگی نامنفی هستند، تساوی (۲۹-۷۶) ایجاب می‌کند که $\hat{b}_i$ هم باید نامنفی باشد.

اکنون بحث می‌کنیم که جواب اولیه باید یک جواب ممکن باشد، یعنی تمام متغیرها باید مقدار نامنفی داشته باشند. متغیرهای غیرپایه با ۰ مقداردهی شده‌اند و بنابراین نامنفی هستند. هر متغیر پایه‌ی x_i طبق تساوی

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j$$

تعریف می‌شود. جواب اولیه قرار می‌دهد $\bar{x}_i = b_i$. با استفاده از بخش دوم ثابت حلقه نتیجه می‌گیریم که هر متغیر پایه‌ی $\bar{x}_i$ نامنفی است.

• پایان: حلقه‌ی while به دو طریق می‌تواند پایان یابد. اگر به خاطر شرط خط ۳ پایان یابد، در این صورت جواب اولیه‌ی فعلی یک جواب ممکن است، و این جواب در خط ۱۷

بازگردانده می‌شود. راه دیگر پایان یافتن حلقه بازگرداندن "unbounded" در خط ۱۱ است. در این حالت، برای هر تکرار از حلقه‌ی for در خطوط ۵-۸، وقتی خط ۶ اجرا می‌شود، متوجه می‌شویم که $a_{ie} \le 0$. جواب $\bar{x}$ را که به صورت

$$\bar{x}_i = \begin{cases} \infty & \text{اگر } i = e \\ 0 & \text{اگر } i \in N - \{e\} \\ b_i - \sum_{j \in N} a_{ij} \bar{x}_j & \text{اگر } i \in B \end{cases}$$

تعریف می‌شود در نظر بگیرید. اکنون نشان می‌دهیم که این جواب یک جواب ممکن است، یعنی تمام متغیرها در آن مقدار نامنفی دارند. متغیرهای غیرپایه، غیر از $\bar{x}_e$، ۰ هستند، و $\bar{x}_e$ مثبت است؛ بنابراین تمام متغیرهای غیرپایه نامنفی هستند. برای هر متغیر پایه‌ی $\bar{x}_i$ داریم

$$\begin{aligned} \bar{x}_i &= b_i - \sum_{j \in N} a_{ij} \bar{x}_j \\ &= b_i - a_{ie} \bar{x}_e \end{aligned}$$

ثابت حلقه ایجاب می‌کند که $b_i \ge 0$، و داریم $a_{ie} \le 0$ و $\bar{x}_e = \infty > 0$. بنابراین $\bar{x}_i \ge 0$.
اکنون نشان می‌دهیم که مقدار هدف برای جواب $\bar{x}$ بی‌کران است. طبق تساوی (۲۹-۴۲) مقدار هدف برابر است با

$$\begin{aligned} z &= v + \sum_{j \in N} c_j \bar{x}_j \\ &= v + c_e \bar{x}_e \end{aligned}$$

از آن جایی که $c_e > 0$ (طبق خط ۴ رویه‌ی SIMPLEX)، و $\bar{x}_e = \infty$، مقدار هدف ∞ است، و بنابراین برنامه‌ی خطی بی‌کران است. ■

این باقی می‌ماند که نشان دهیم SIMPLEX پایان می‌یابد، و پس از پایان جواب بازگردانده شده بهینه است. بخش ۲۹-۴ بهینه‌بودن جواب را بررسی می‌کند، و در این جا در مورد پایان یافتن الگوریتم بحث خواهیم کرد.

پایان یافتن الگوریتم

در مثال ارائه شده در ابتدای این بخش، هر تکرار الگوریتم سیمپلکس مقدار هدف مربوط به جواب اولیه را کمی افزایش می‌داد. همان طور که باید در تمرین ۲۹-۳-۲ نشان دهید، هیچ تکراری از الگوریتم سیمپلکس نمی‌تواند مقدار هدف جواب اولیه را کاهش دهد. متأسفانه ممکن است مقدار هدف در بعضی تکرارها ثابت باقی بماند. این پدیده *انحطاط* (degeneracy) نام دارد، و در این جا با جزئیات بیشتر آن را بررسی می‌کنیم.

مقداردهی انجام شده در خط ۱۴ رویه‌ی PIVOT ($\hat{v} = v + c_e \hat{b}_e$) مقدار هدف را تغییر می‌دهد. از آن جایی که SIMPLEX فقط زمانی PIVOT را فراخوانی می‌کند که داشته باشیم $c_e > 0$، تنها راه ثابت

باقی ماندن مقدار هدف (که در آن داریم $\hat{v} = v$) این است که $\hat{b_e}$ برابر ۰ باشد. این مقدار در خط ۳ رویه‌ی PIVOT به صورت $\hat{b_e} = b_l / a_{le}$ مقداردهی می‌شود. چون همیشه PIVOT را با $a_{le} \neq 0$ فراخوانی می‌کنیم، می‌بینیم که برای این که $\hat{b_e}$ برابر ۰ باشد (و در نتیجه مقدار هدف تغییر نکند) باید داشته باشیم $b_l = 0$.

وقوع این وضعیت کاملاً ممکن است. برنامه‌ی خطی زیر را در نظر بگیرید:

$$\begin{aligned} z &= x_1 + x_2 + x_3 \\ x_4 &= 8 - x_1 - x_2 \\ x_5 &= x_2 - x_3 \end{aligned}$$

فرض کنید x_1 را به عنوان متغیر ورودی و x_4 را به عنوان متغیر خروجی انتخاب می‌کنیم. پس از چرخش به دست می‌آوریم

$$\begin{aligned} z &= 8 + x_3 - x_4 \\ x_1 &= 8 - x_2 - x_4 \\ x_5 &= x_2 - x_3 \end{aligned}$$

در این لحظه، تنها گزینه‌ی موجود انتخاب x_3 به عنوان ورودی و x_5 به عنوان خروجی است. از آن جایی که $b_5 = 0$، مقدار هدف ۸ پس از چرخش بدون تغییر باقی می‌ماند:

$$\begin{aligned} z &= 8 + x_2 - x_4 - x_5 \\ x_1 &= 8 - x_2 - x_4 \\ x_3 &= x_2 - x_5 \end{aligned}$$

مقدار هدف تغییر نکرده است، ولی فرم ضعیف تغییر کرده است. خوشبختانه اگر دوباره با x_2 به عنوان ورودی و x_1 به عنوان خروجی آغاز کنیم، مقدار هدف (تا ۱۶) افزایش می‌یابد، و الگوریتم سیمپلکس می‌تواند ادامه یابد.

انحطاط می‌تواند از پایان یافتن الگوریتم سیمپلکس جلوگیری کند، چرا که ممکن است به پدیده‌ای با نام *دورزدن* (cycling) منجر شود: فرم‌های ضعیف در دو تکرار مختلف SIMPLEX یکسان هستند. به خاطر انحطاط، SIMPLEX ممکن است دنباله‌ای از اعمال چرخش را انتخاب کند که مقدار هدف را تغییر می‌دهد، ولی یک فرم ضعیف را در این دنباله تکرار می‌کند. از آن جایی که SIMPLEX یک الگوریتم قطعی است، اگر یک بار دور بزند، بر روی همان سری برای همیشه دور خواهد زد و هیچ گاه پایان نخواهد یافت.

دور زدن تنها دلیلی است که می‌تواند باعث شود SIMPLEX پایان نیابد. برای نشان دادن این حقیقت ابتدا باید یک مکانیزم جدید طراحی کنیم.

SIMPLEX در هر تکرار، علاوه بر N و B، مقادیر A، b، c، و v را هم نگه می‌دارد. با این که داشتن A، b، c و v برای پیاده‌سازی بهینه‌ی الگوریتم سیمپلکس حیاتی است، ولی در کل این کار ضروری نیست. به عبارت دیگر فرم ضعیف به صورت یکتا توسط مجموعه‌ی متغیرهای پایه و غیرپایه تعیین می‌شود. قبل از اثبات این نکته، یک لم جبری مفید را اثبات می‌کنیم.

لم ۳-۲۹

فرض کنید I مجموعه‌ای از اندیس‌ها باشد. برای هر $j \in I$، فرض کنید α_j و β_j اعداد حقیقی، x_j یک متغیر با مقدار حقیقی، و γ یک عدد حقیقی دلخواه باشد. فرض کنید برای هر مقداردهی به x_j داریم

$$\sum_{j \in I} \alpha_j x_j = \gamma + \sum_{j \in I} \beta_j x_j \qquad (۷۸-۲۹)$$

آن گاه برای هر $j \in I$ داریم $\alpha_j = \beta_j$ و $\gamma = ۰$.

اثبات از آن جایی که تساوی (۲۹-۷۸) برای هر مقداری از x_j برقرار است، می‌توانیم از مقادیر خاص برای نتیجه‌گیری در مورد α، β، و γ استفاده کنیم. اگر قرار دهیم $x_j = ۰$ برای هر $j \in I$، نتیجه می‌گیریم که $\gamma = ۰$. اکنون یک اندیس دلخواه $j \in I$ انتخاب کرده و برای تمام $k \neq j$ قرار می‌دهیم $x_j = ۱$ و $x_k = ۰$. آن گاه باید داشته باشیم $\alpha_j = \beta_j$. از آن جایی که j را به صورت یک اندیس دلخواه در I انتخاب کرده‌ایم، نتیجه می‌گیریم که برای هر $j \in I$ داریم $\alpha_j = \beta_j$. ■

یک برنامه‌ی خطی خاص فرم‌های ضعیف مختلف زیادی دارد؛ به خاطر بیاورید که تمام فرم‌های ضعیف مجموعه‌ی جواب‌های ممکن و بهینه‌ی یکسانی با برنامه‌ی خطی اولیه دارند. اکنون نشان می‌دهیم که فرم ضعیف یک برنامه‌ی خطی را می‌توان به صورت یکتا از روی مجموعه‌ی متغیرهای پایه تعیین کرد. یعنی با داشتن مجموعه‌ی متغیرهای پایه، می‌توان یک فرم ضعیف یکتا (مجموعه‌ای یکتا از ضرایب و سمت راست تساوی‌ها) به آن مجموعه نسبت داد.

لم ۴-۲۹

فرض کنید (A,b,c) یک برنامه‌ی خطی به شکل استاندارد باشد. با داشتن مجموعه‌ی B از متغیرهای پایه، فرم ضعیف متناظر آن را می‌توان به صورت یکتا تعیین کرد.

اثبات از طریق برهان خلف، فرض کنید دو فرم ضعیف مختلف با مجموعه‌ی متغیرهای پایه‌ی یکسان وجود دارند. مجموعه‌ی $N = \{۱, ۲, ..., n+m\} - B$ از متغیرهای غیرپایه‌ی هر دو فرم هم باید یکسان باشد. فرم ضعیف اول را به صورت

$$z = v + \sum_{j \in N} c_j x_j \qquad (۷۹-۲۹)$$

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \qquad \text{برای } i \in B \qquad (۸۰-۲۹)$$

و فرم ضعیف دوم را به صورت

$$z = v' + \sum_{j \in N} c'_j x_j \qquad (۸۱-۲۹)$$

$$x_i = b'_i - \sum_{j \in N} a'_{ij} x_j \qquad \text{برای } i \in B \qquad (۸۲-۲۹)$$

می‌نویسیم. سیستم معادلات تشکیل شده از تفریق هر تساوی در خط (۲۹-۸۲) از تساوی مربوطه در خط (۲۹-۸۰) را در نظر بگیرید. سیستم حاصل به صورت زیر است:

$$0 = (b_i - b_i') - \sum_{j \in N} (a_{ij} - a_{ij}')x_j \qquad \text{برای } i \in B$$

و یا، به طور معادل،

$$\sum_{j \in N} a_{ij} x_j = (b_i - b_i') + \sum_{j \in N} a_{ij}' x_j \qquad \text{برای } i \in B$$

اکنون برای هر $i \in B$، لم ۲۹-۳ را با $\alpha_i = a_{ij}$، $\beta_i = a_{ij}'$، $\gamma = b_i - b_i'$، و $I = N$ به کار می‌بریم. از آن جایی که $\alpha_i = \beta_i$، برای هر $j \in N$ داریم $a_{ij} = a_{ij}'$، و چون $\gamma = 0$، داریم $b_i = b_i'$. بنابراین برای دو فرم ضعیف، A و b معادل هستند با A' و b'. با استفاده از بحثی مشابه، تمرین ۲۹-۳-۱ نشان می‌دهد که باید داشته باشیم $c = c'$ و $v = v'$، و بنابراین دو فرم ضعیف معادل یکدیگر هستند. ■

اکنون نشان می‌دهیم که دور زدن تنها دلیل ممکن برای عدم پایان یافتن SIMPLEX است.

لم ۲۹-۵ اگر SIMPLEX حداکثر در $\binom{n+m}{m}$ تکرار پایان نیابد، در این صورت دور خواهد زد.

اثبات طبق لم ۲۹-۴، مجموعه‌ی B از متغیرهای پایه به صورت یکتا فرم ضعیف را مشخص می‌کند. تعداد کل متغیرها $n+m$ است و داریم $|B| = m$، و بنابراین حداکثر $\binom{n+m}{m}$ روش برای انتخاب B وجود دارد. پس فقط $\binom{n+m}{m}$ فرم ضعیف متفاوت داریم، و اگر SIMPLEX بیشتر از $\binom{n+m}{m}$ تکرار انجام دهد، باید برای همیشه دور بزند. ■

دور زدن از نظر تئوری امکان‌پذیر، ولی در عمل به شدت نادر است. می‌توان با دقت در انتخاب متغیرهای ورودی و خروجی از آن دوری کرد. یک روش این است که ورودی را کمی تغییر دهیم تا داشتن دو جواب با مقدار هدف یکسان غیرممکن باشد. روش دوم حل مسئله‌ی حالت‌های برابر با انتخاب متغیر با اندیس کم‌تر است. این استراتژی آخر به *قانون Bland* معروف است. از اثبات این استراتژی‌های پرهیز از دور زدن صرف نظر می‌کنیم.

لم ۲۹-۶ اگر در خطوط ۴ و ۹ رویه‌ی SIMPLEX، حالت‌های مساوی همیشه با انتخاب متغیر با اندیس کم‌تر حل شود، SIMPLEX همیشه پایان می‌یابد.

■

این بخش را با لم زیر به پایان می‌بریم.

لم ۲-۲۹

با فرض این که INITIALIZE-SIMPLEX یک فرم ضعیف بازمی‌گرداند که جواب اولیه‌ی آن یک جواب ممکن است، در این صورت SIMPLEX یا گزارش می‌دهد که برنامه‌ی خطی بی‌کران است، و یا در حداکثر $\binom{n+m}{m}$ تکرار با یک جواب ممکن پایان می‌یابد.

اثبات لم‌های ۲۹-۲ و ۲۹-۶ نشان می‌دهند که اگر INITIALIZE-SIMPLEX یک فرم ضعیف بازگرداند که جواب اولیه‌ی آن، یک جواب ممکن باشد، SIMPLEX یا گزارش می‌کند که برنامه‌ی خطی بی‌کران است، و یا با یک جواب ممکن پایان می‌یابد. طبق عکس نقیض لم ۲۹-۵، اگر SIMPLEX با یک جواب ممکن پایان یابد، در این صورت حداکثر در $\binom{n+m}{m}$ تکرار پایان می‌یابد. ■

تمرین‌ها

۲۹-۳-۱ با نشان دادن این که باید داشته باشیم $c = c'$ و $v = v'$، اثبات لم ۲۹-۴ را کامل کنید.

۲۹-۳-۲ نشان دهید که فراخوانی PIVOT در خط ۱۲ رویه‌ی SIMPLEX هیچ گاه مقدار v را کاهش نمی‌دهد.

۲۹-۳-۳ اثبات کنید که شکل ضعیف ارسال شده به رویه‌ی PIVOT و فرم ضعیف بازگردانده شده توسط این رویه با هم معادل هستند.

۲۹-۳-۴ فرض کنید که یک برنامه‌ی خطی (A, b, c) را از شکل استاندارد به شکل ضعیف تبدیل می‌کنیم. نشان دهید که جواب اولیه، یک جواب ممکن است اگر و فقط اگر $b_i \geq ۰$ برای $i = ۱, ۲, \ldots, m$.

۲۹-۳-۵ برنامه‌ی خطی زیر را با استفاده از SIMPLEX حل کنید:

$$\text{minimize} \quad ۱۸x_۱ + ۱۲/۵x_۲$$

با شرایط

$$\begin{aligned} x_۱ + x_۲ &\leq ۲۰ \\ x_۱ &\leq ۱۲ \\ x_۲ &\leq ۱۶ \\ x_۱,\ x_۲ &\geq ۰ \end{aligned}$$

۲۹-۳-۶ برنامه‌ی خطی زیر را با استفاده از SIMPLEX حل کنید:

$$\text{minimize} \quad -۵x_۱ - ۳x_۲$$

با شرایط

$$x_1 - x_2 \le 1$$
$$2x_1 + x_2 \le 2$$
$$x_1,\ x_2 \ge 0$$

۷-۳-۲۹ برنامه‌ی خطی زیر را با استفاده از SIMPLEX حل کنید:

$$\text{minimize} \quad x_1 + x_2 + x_3$$

با شرایط

$$2x_1 + 7/5x_2 + 3x_3 \ge 1000$$
$$20x_1 + 5x_2 + 10x_3 \ge 3000$$
$$x_1, \quad x_2, \quad x_3 \ge 0$$

۸-۳-۲۹ در اثبات لم ۲۹-۵، بحث کردیم که حداکثر $\binom{m+n}{n}$ روش برای انتخاب مجموعه‌ی B از متغیرهای پایه وجود دارد. مثالی از یک برنامه‌ی خطی ارائه کنید که در آن اکیداً کم‌تر از $\binom{m+n}{n}$ روش برای انتخاب مجموعه‌ی B وجود دارد.

۲۹-۴ دوگانگی

اثبات کردیم که تحت فرض‌های خاص SIMPLEX پایان می‌یابد، ولی هنوز نشان نداده‌ایم که جواب بهینه‌ی برنامه‌ی خطی را می‌یابد. برای انجام این کار یک مفهوم قدرتمند به نام *دوگانگی برنامه‌ریزی خطی* (linear-programmin duality) را معرفی می‌کنیم.

دوگانگی ما را قادر می‌سازد اثبات کنیم که یک جواب به دست آمده، بهینه است. یک مثال از دوگانگی را در فصل ۲۶ با قضیه‌ی ۲۶-۶ (شار بیشینه-برش کمینه) دیدیم. فرض کنید با داشتن یک نمونه از مسئله‌ی شار بشینیه، یک شار f با مقدار $|f|$ می‌یابیم. چگونه می‌توانیم بفهمیم که f بهینه است یا خیر؟ طبق قضیه‌ی شار بیشینه-برش کمینه، اگر بتوانیم یک برش بیابیم که مقدار آن $|f|$ باشد، آن گاه نشان داده‌ایم که f یک شار بیشینه است. این یک مثال دوگانگی است: با داشتن یک مسئله‌ی بیشینه‌سازی، یک مسئله‌ی کمینه‌سازی متناظر تعریف می‌کنیم به طوری که مقدار هدف بهینه‌ی دو مسئله با هم برابر باشد.

با داشتن یک برنامه‌ی خطی که هدف آن بیشینه‌سازی است، توضیح خواهیم داد که چگونه می‌توان یک برنامه‌ی خطی *دوگان* (dual) برای آن ارائه کرد که هدف آن کمینه‌سازی بوده و مقدار بهینه‌ی آن با برنامه‌ی خطی اولیه برابر باشد. هنگام کار با برنامه‌های خطی دوگان، به برنامه‌ی خطی اصلی، مسئله‌ی *اولیه* (primal) می‌گوییم.

با داشتن یک برنامه‌ی خطی اولیه به شکل استاندارد، مانند (۲۹-۱۶)-(۲۹-۱۸)، برنامه‌ی خطی دوگان را به صورت

$$\text{minimize} \quad \sum_{i=1}^{m} b_i y_i \tag{۲۹-۸۳}$$

با شرایط

$$\sum_{i=1}^{m} a_{ij} y_i \geq c_j \qquad \text{برای } j = 1, 2, \ldots, n \tag{۲۹-۸۴}$$

$$y_i \geq 0 \qquad \text{برای } i = 1, 2, \ldots, m \tag{۲۹-۸۵}$$

تعریف می‌کنیم. برای تشکیل فرم دوگان، بیشینه‌سازی را به کمینه‌سازی تبدیل کرده، نقش سمت راست‌ها و ضرایب تابع هدف را عوض کرده، و علامت کوچک‌تر مساوی را با علامت بزرگ‌تر مساوی جایگزین می‌کنیم. هر یک از m محدودیت در برنامه‌ی اولیه یک متغیر متناظر y_i در دوگان دارد، و هر یک از n محدودیت در دوگان، یک متغیر متناظر x_j در برنامه‌ی اولیه. به عنوان مثال برنامه‌ی خطی داده شده در (۲۹-۵۳)–(۲۹-۵۷) را در نظر بگیرید. دوگان این برنامه‌ی خطی به صورت زیر است:

$$\text{minimize} \quad 30y_1 + 24y_2 + 36y_3 \tag{۲۹-۸۶}$$

با شرایط

$$y_1 + 2y_2 + 4y_3 \geq 3 \tag{۲۹-۸۷}$$

$$y_1 + 2y_2 + y_3 \geq 1 \tag{۲۹-۸۸}$$

$$3y_1 + 5y_2 + 2y_3 \geq 2 \tag{۲۹-۸۹}$$

$$y_1, \quad y_2, \quad y_3 \geq 0 \tag{۲۹-۹۰}$$

در قضیه‌ی ۲۹-۱۰ نشان خواهیم داد که مقدار بهینه در برنامه‌ی خطی دوگان همیشه برابر است با مقدار بهینه برای برنامه‌ی خطی اولیه. به علاوه، الگوریتم سیمپلکس در واقع به طور ضمنی برنامه‌های خطی اولیه و دوگان را هم‌زمان حل می‌کند، که خود اثباتی است برای بهینه بودن.

با دوگانگی ضعیف آغاز می‌کنیم که می‌گوید مقدار یک جواب ممکن برای برنامه‌ی خطی اولیه نمی‌تواند بزرگ‌تر از مقدار یک جواب ممکن برای برنامه‌ی خطی دوگان باشد.

لم ۲۹-۹

فرض کنید $\bar{x}$ یک جواب ممکن برای برنامه‌ی خطی اولیه‌ی داده شده در (۲۹-۱۶)–(۲۹-۱۸) باشد، و $\bar{y}$ یک جواب ممکن برای برنامه‌ی خطی دوگان در (۲۹-۸۳)–(۲۹-۸۵). آن گاه

$$\sum_{j=1}^{n} c_j \bar{x}_j \leq \sum_{i=1}^{m} b_i \bar{y}_i$$

اثبات داریم

$$\sum_{j=1}^{n} c_j \bar{x}_j \le \sum_{j=1}^{n}\left(\sum_{i=1}^{m} a_{ij}\bar{y}_i\right)\bar{x}_j \qquad \text{(طبق (۲۹-۸۴))}$$

$$= \sum_{i=1}^{m}\left(\sum_{j=1}^{n} a_{ij}\bar{x}_j\right)\bar{y}_i$$

$$\le \sum_{i=1}^{m} b_i \bar{y}_i \qquad \text{(طبق (۲۹-۱۷))}$$

■

نتیجه‌ی ۲۹-۹ فرض کنید $\bar{x}$ یک جواب ممکن برای یک برنامه‌ی خطی اولیه به صورت (A,b,c) باشد، و $\bar{y}$ یک جواب ممکن برای برنامه‌ی خطی دوگان متناظر. اگر

$$\sum_{j=1}^{n} c_j \bar{x}_j \le \sum_{i=1}^{m} b_i \bar{y}_i$$

آن گاه $\bar{x}$ و $\bar{y}$ به ترتیب جواب‌های بهینه برای برنامه‌های خطی اولیه و دوگان هستند.

اثبات طبق لم ۲۹-۸ مقدار هدف یک جواب ممکن به برنامه‌ی خطی اولیه نمی‌تواند از مقدار هدف یک جواب ممکن برای دوگان فراتر رود. برنامه‌ی خطی اولیه یک مسئله‌ی بیشینه‌سازی است، و برنامه‌ی خطی دوگان یک مسئله‌ی کمینه‌سازی. بنابراین اگر مقدار هدف جواب‌های ممکن $\bar{x}$ و $\bar{y}$ برابر باشد، مقدار هیچ کدام را نمی‌توان بهبود بخشید. ■

قبل از اثبات این که همیشه یک جواب دوگان وجود دارد که مقدار آن برابر است با مقدار یک جواب بهینه برای برنامه‌ی اولیه، ابتدا توضیح می‌دهیم که چگونه می‌توان چنین جوابی را یافت. وقتی الگوریتم سیمپلکس را بر روی برنامه‌ی خطی (۲۹-۵۳)-(۲۹-۵۷) اجرا کردیم، تکرار آخر به فرم ضعیف (۲۹-۷۲)-(۲۹-۷۵) منجر شد، با هدف $z = ۲۸ - x_۳/۶ - x_۵/۶ - ۲x_۴/۳$، $B = \{۱,۲,۴\}$ و $N = \{۳,۵,۶\}$. همان طور که در ادامه نشان خواهیم داد، جواب اولیه متناظر با فرم ضعیف نهایی یک جواب بهینه برای برنامه‌ی خطی است؛ بنابراین یک جواب بهینه به برنامه‌ی خطی (۲۹-۵۳)-(۲۹-۵۷) عبارت است از $(\bar{x}_۱, \bar{x}_۲, \bar{x}_۳) = (۸, ۴, ۰)$ با مقدار هدف $۲۸ = (۲\times۰) + (۱\times۴) + (۳\times۸)$. همان طور که باز هم در ادامه نشان خواهیم داد، می‌توانیم از روی این جواب یک جواب بهینه برای مسئله‌ی دوگان هم تعیین کنیم: منفی ضرایب تابع هدف اولیه برابر است با مقدار متغیرهای دوگان. به طور دقیق‌تر، فرض کنید فرم ضعیف برنامه‌ی اولیه به صورت زیر است:

$$z = v' + \sum_{j \in N} c'_j x_j$$

$$x_i = b'_i - \sum_{i \in N} a'_{ij} x_j \qquad \text{برای } i \in B$$

در این صورت جواب بهینه‌ی دوگان عبارت است از

$$\bar{y}_i = \begin{cases} -c'_{n+i} & \text{اگر } (n+i) \in N \\ ۰ & \text{در غیر این صورت} \end{cases} \qquad (۹۱-۲۹)$$

بنابراین یک جواب به برنامه‌ی خطی دوگان تعریف شده در (۲۹-۸۶)–(۲۹-۹۰) عبارت است از $\bar{y}_۱ = ۰$ (چرا که $n+۱=۴ \in B$)، $\bar{y}_۲ = -c'_۵ = ۱/۶$، $\bar{y}_۳ = -c'_۶ = ۲/۳$ و $\bar{y}_۴ = -c'_۷ = ۱/۲$. با ارزیابی تابع هدف دوگان (۲۹-۸۹)، یک مقدار هدف ۲۸ $= ((۲/۳)\times۳۶) + ((۱/۶)\times۲۴) + (۰\times۳۰)$ به دست می‌آوریم، که تأیید می‌کند که مقدار هدف برنامه‌ی اولیه برابر است با مقدار هدف برنامه‌ی دوگان. با ترکیب این محاسبات با لم ۲۹-۸ یک اثبات خواهیم داشت که مقدار هدف بهینه‌ی برنامه‌ی خطی اولیه ۲۸ است. اکنون نشان می‌دهیم که به طور کلی می‌توان یک جواب بهینه برای دوگان و یک اثبات برای بهینه بودن یک جواب برای برنامه‌ی اولیه را به این روش به دست آورد.

قضیه‌ی ۲۹-۱۰ (دوگانگی برنامه‌ریزی خطی) فرض کنید SIMPLEX مقادیر $\bar{x} = (\bar{x}_۱, \bar{x}_۲, ..., \bar{x}_n)$ را برای برنامه‌ی خطی اولیه‌ی (A,b,c) بازمی‌گرداند. فرض کنید N و B نشان‌دهنده‌ی متغیرهای غیرپایه و پایه‌ی فرم ضعیف نهایی، و c' نشان‌دهنده‌ی ضرایب در فرم ضعیف نهایی باشد، و $\bar{y} = (\bar{y}_۱, \bar{y}_۲, ..., \bar{y}_m)$ طبق تساوی (۲۹-۹۱) تعریف شده باشد. آن گاه $\bar{x}$ یک جواب بهینه برای برنامه‌ی خطی اولیه است، $\bar{y}$ یک جواب بهینه برای برنامه‌ی خطی دوگان، و

$$\sum_{j=۱}^{n} c_j \bar{x}_j = \sum_{i=۱}^{m} b_i \bar{y}_i \qquad (۹۲-۲۹)$$

اثبات طبق نتیجه‌ی ۲۹-۹ اگر بتوانیم جواب‌های ممکن $\bar{x}$ و $\bar{y}$ را بیابیم به طوری که تساوی (۲۹-۹۲) را ارضا کنند، آن گاه $\bar{x}$ و $\bar{y}$ باید جواب‌های بهینه‌ی اولیه و دوگان باشند. اکنون نشان می‌دهیم که جواب‌های $\bar{x}$ و $\bar{y}$ توصیف شده در صورت قضیه تساوی (۲۹-۹۲) را ارضا می‌کنند.

فرض کنید SIMPLEX را بر روی برنامه‌ی خطی اولیه، همان طور که در (۲۹-۱۶)–(۲۹-۱۸) داده شده است، اجرا می‌کنیم. الگوریتم بر روی یک سری از فرم‌های ضعیف ادامه می‌دهد تا این که با یک فرم ضعیف نهایی با تابع هدف

$$z = v' + \sum_{j \in N} c'_j x_j \qquad (۹۳-۲۹)$$

پایان یابد. چون SIMPLEX با یک جواب پایان می‌یابد، طبق شرط خط ۲ می‌دانیم که

$$c'_j \le ۰ \qquad \text{برای هر } j \in N \qquad (۹۴-۲۹)$$

اگر تعریف کنیم

$$c'_j = ۰ \qquad \text{برای هر } j \in B \tag{۲۹-۹۵}$$

می‌توانیم تساوی (۲۹-۹۳) را به صورت

$$\begin{aligned} z &= v' + \sum_{j \in N} c'_j x_j \\ &= v' + \sum_{j \in N} c'_j + x_j + \sum_{j \in B} c'_j x_j && (\text{چون } j \in B \text{ اگر } c'_j = ۰) \\ &= v' + \sum_{j=۱}^{n+m} c'_j x_j && (\text{چون } N \cup B = \{۱,۲,...,n+m\}) \end{aligned} \tag{۲۹-۹۶}$$

بازنویسی کنیم. برای جواب اولیه‌ی $\bar{x}$ متناظر با این فرم ضعیف نهایی داریم $\bar{x}_j = ۰$ برای هر $j \in N$، و $z = v'$. چون تمام فرم‌های ضعیف با یکدیگر معادل هستند، اگر تابع هدف اصلی را بر روی $\bar{x}$ ارزیابی کنیم، باید همان مقدار هدف را به دست بیاوریم، یعنی

$$\sum_{j=۱}^{n} c_j \bar{x}_j = v' + \sum_{j=۱}^{n+m} c'_j \bar{x}_j \tag{۲۹-۹۷}$$

$$\begin{aligned} &= v' + \sum_{j \in N} c'_j \bar{x}_j + \sum_{j \in B} c'_j \bar{x}_j \\ &= v' + \sum_{j \in N} (c'_j \cdot ۰) + \sum_{j \in B} (۰ \cdot \bar{x}_j) \\ &= v' \end{aligned} \tag{۲۹-۹۸}$$

اکنون نشان می‌دهیم که $\bar{y}$، که طبق تساوی (۲۹-۹۱) تعریف شده است، برای برنامه‌ی خطی دوگان یک جواب ممکن است، و مقدار هدف آن یعنی $\sum_{i=۱}^{m} b_i \bar{y}_i$ برابر است با $\sum_{j=۱}^{n} c_j \bar{x}_j$. تساوی (۲۹-۹۷) می‌گوید که مقدار ارزیابی فرم‌های ضعیف اول و آخر بر روی $\bar{x}$ برابر است. به طور کلی‌تر، معادل بودن تمام فرم‌های ضعیف ایجاب می‌کند که برای هر مجموعه‌ای از مقادیر $x = (x_۱, x_۲, ..., x_n)$ داشته باشیم

$$\sum_{j=۱}^{n} c_j x_j = v' + \sum_{j=۱}^{n+m} c'_j x_j$$

بنابراین برای هر مجموعه‌ی خاص از مقادیر $\bar{x} = (\bar{x}_۱, \bar{x}_۲, ..., \bar{x}_n)$ داریم

$$\begin{aligned} &\sum_{j=۱}^{n} c_j \bar{x}_j \\ &= v' + \sum_{j=۱}^{n+m} c'_j \bar{x}_j \\ &= v' + \sum_{j=۱}^{n} c'_j \bar{x}_j + \sum_{j=n+۱}^{n+m} c'_j \bar{x}_j \end{aligned}$$

$$= v' + \sum_{j=1}^{n} c'_j \bar{x}_j + \sum_{i=1}^{m} c'_{n+i} \bar{x}_{n+i}$$

$$= v' + \sum_{j=1}^{n} c'_j \bar{x}_j + \sum_{i=1}^{m} (-\bar{y}_i) \bar{x}_{n+i} \qquad \text{(طبق تساوی‌های (۲۹-۹۱) و (۲۹-۹۵))}$$

$$= v' + \sum_{j=1}^{n} c'_j \bar{x}_j + \sum_{i=1}^{m} (-\bar{y}_i) \left(b_i - \sum_{j=1}^{n} a_{ij} \bar{x}_j \right) \qquad \text{(طبق تساوی (۲۹-۳۲))}$$

$$= v' + \sum_{j=1}^{n} c'_j \bar{x}_j - \sum_{i=1}^{m} b_i \bar{y}_i + \sum_{i=1}^{m} \sum_{j=1}^{n} (a_{ij} \bar{x}_j) \bar{y}_i$$

$$= v' + \sum_{j=1}^{n} c'_j \bar{x}_j - \sum_{i=1}^{m} b_i \bar{y}_i + \sum_{j=1}^{n} \sum_{i=1}^{m} (a_{ij} \bar{y}_i) \bar{x}_j$$

$$= \left(v' - \sum_{i=1}^{m} b_i \bar{y}_i \right) + \sum_{j=1}^{n} \left(c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i \right) \bar{x}_j$$

به طوری که

$$\sum_{j=1}^{n} c_j \bar{x}_j = \left(v' - \sum_{i=1}^{m} b_i \bar{y}_i \right) + \sum_{j=1}^{n} \left(c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i \right) \bar{x}_j \qquad (۲۹-۹۹)$$

با به کار بردن لم ۲۹-۳ بر روی تساوی (۲۹-۹۹) به دست می‌آوریم

$$v' - \sum_{i=1}^{m} b_i \bar{y}_i = 0, \qquad (۲۹-۱۰۰)$$

$$c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i = c_i \qquad j = 1, 2, ..., n \text{ برای} \qquad (۲۹-۱۰۱)$$

طبق تساوی (۲۹-۱۰۰) داریم $v' = \sum_{i=1}^{m} b_i \bar{y}_i$، و بنابراین مقدار هدف دوگان ($\sum_{i=1}^{m} b_i \bar{y}_i$) برابر است با مقدار هدف برنامه‌ی اولیه (v'). این باقی می‌ماند که نشان دهیم که جواب $\bar{y}$ برای برنامه‌ی دوگان یک جواب ممکن است. از نامساوی‌های (۲۹-۹۴) و تساوی‌های (۲۹-۹۵) داریم $c'_j \leq 0$ برای تمام $j = 1, 2, ..., n+m$. بنابراین برای هر $i = 1, 2, ..., m$، تساوی (۲۹-۱۰۱) نتیجه می‌دهد

$$c_j = c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i$$

$$\leq \sum_{i=1}^{m} a_{ij} \bar{y}_i$$

که محدودیت‌های (۲۹-۸۴) را برای دوگان ارضا می‌کند. نهایتاً از آن جایی که $c'_j \leq 0$ برای هر $j \in N \cup B$، وقتی $\bar{y}$ را طبق تساوی (۲۹-۹۱) مقداردهی می‌کنیم، داریم $\bar{y}_i \geq 0$، و بنابراین محدودیت‌های نامنفی بودن هم ارضا می‌شوند. ■

نشان دادیم که با داشتن یک برنامه‌ی خطی ممکن، اگر INITIALIZE-SIMPLEX یک جواب ممکن بازگرداند، و اگر SIMPLEX پایان یابد بدون این که "unbounded" را بازگرداند، آن گاه جواب بازگردانده شده یک جواب بهینه است. همچنین نشان دادیم که چطور می‌توان یک جواب بهینه برای برنامه‌ی خطی دوگان ساخت.

تمرین‌ها

۲۹-۴-۱ دوگان برنامه‌ی خطی داده شده در تمرین ۲۹-۳-۵ را توصیف کنید.

۲۹-۴-۲ فرض کنید یک برنامه‌ی خطی داریم که به شکل استاندارد نیست. برای محاسبه‌ی دوگان، می‌توانیم ابتدا برنامه‌ی خطی را به فرم استاندارد تبدیل کنیم و سپس دوگان آن را به دست آوریم. با این حال بهتر خواهد بود اگر بتوانیم دوگان را مستقیماً محاسبه کنیم. با داشتن یک برنامه‌ی خطی دلخواه، توضیح دهید که چگونه می‌توان دوگان آن را محاسبه کرد.

۲۹-۴-۳ دوگان برنامه‌ی خطی شار بیشینه را که در خطوط (۲۹-۴۷)-(۲۹-۵۰) داده شده است، به دست آورید. توضیح دهید که چگونه می‌توان این فرمول‌بندی را به صورت یک مسئله‌ی برش کمینه ترجمه کرد.

۲۹-۴-۴ دوگان برنامه‌ی خطی شار با کم‌ترین هزینه را که در خطوط (۲۹-۵۱)-(۲۹-۵۲) داده شده است، به دست آورید. توضیح دهید که چگونه می‌توان این مسئله را بر حسب گراف‌ها و شارها توصیف کرد.

۲۹-۴-۵ نشان دهید که دوگان دوگان یک برنامه‌ی خطی، همان برنامه‌ی خطی اولیه است.

۲۹-۴-۶ کدام نتیجه از فصل ۲۶ را می‌توان به صورت دوگانگی ضعیف مسئله‌ی شار بیشینه ترجمه کرد؟

۲۹-۵ جواب ممکن اولیه

در این بخش، ابتدا توضیح خواهیم داد که چگونه می‌توان فهمید یک برنامه‌ی خطی، ممکن است یا خیر، و سپس در صورت ممکن بودن برنامه‌ی خطی، این که چگونه می‌توان یک فرم ضعیف از برنامه‌ی خطی را تولید کرد که جواب اولیه‌ی آن، یک جواب ممکن باشد. با اثبات قضیه‌ی اصلی برنامه‌ریزی خطی کار را به پایان می‌بریم، که می‌گوید رویه‌ی SIMPLEX همیشه نتیجه‌ی صحیحی تولید می‌کند.

یافتن یک جواب اولیه

در بخش ۲۹-۳ فرض کردیم که یک رویه‌ی INITIALIZE-SIMPLEX داریم که تعیین می‌کند که یک برنامه‌ی خطی جواب ممکن دارد یا خیر، و اگر چنین باشد، یک فرم ضعیف به ما می‌دهد که جواب اولیه‌ی آن یک جواب ممکن است. در این جا این رویه را توضیح خواهیم داد.

یک برنامه‌ی خطی می‌تواند ممکن باشد، و در عین حال جواب اولیه‌ی آن ممکن نباشد. برای مثال برنامه‌ی خطی زیر را در نظر بگیرید:

$$\text{minimize} \quad 2x_1 - x_2 \tag{۲۹-۱۰۲}$$

با شرایط

$$2x_1 - x_2 \le 2 \tag{۲۹-۱۰۳}$$

$$x_1 - 5x_2 \le -4 \tag{۲۹-۱۰۴}$$

$$x_1, \quad x_2 \ge 0 \tag{۲۹-۱۰۵}$$

اگر بخواهیم این برنامه‌ی خطی را به شکل ضعیف تبدیل کنیم، جواب اولیه به صورت $x_1 = 0$ و $x_2 = 0$ خواهد بود. این جواب محدودیت (۲۹-۱۰۴) را نقض می‌کند، و بنابراین یک جواب ممکن نیست. از این رو INITIALIZE-SIMPLEX نمی‌تواند فرم ضعیف بدیهی برنامه‌ی خطی را بازگرداند. برای تعیین این مسئله می‌توانیم یک *برنامه‌ی خطی کمکی* (auxiliary linear program) توصیف کنیم. برای این برنامه‌ی خطی کمکی قادر خواهیم بود (با مقداری تلاش) یک فرم ضعیف به دست آوریم که جواب اولیه‌ی آن، ممکن است. به علاوه، جواب این برنامه‌ی خطی کمکی تعیین می‌کند که آیا برنامه‌ی خطی اولیه ممکن است یا نه، و در صورت ممکن بودن، یک جواب ممکن برای آن ارائه می‌کند که به کمک آن می‌توانیم SIMPLEX را آغاز کنیم.

لم ۲۹-۱۱

فرض کنید L یک برنامه‌ی خطی به فرم استاندارد باشد، که به صورت (۲۹-۱۶)-(۲۹-۱۸) به ما داده شده است. فرض کنید x_0 یک متغیر جدید باشد، و L_{aux} برنامه‌ی خطی زیر با $n+1$ متغیر:

$$\text{minimize} \quad -x_0 \tag{۲۹-۱۰۶}$$

با شرایط

$$\sum_{j=1}^{n} a_{ij} x_j - x_0 \le b_i \qquad \text{برای } i = 1, 2, \ldots, m \tag{۲۹-۱۰۷}$$

$$x_j \ge 0 \qquad \text{برای } j = 0, 1, \ldots, n \tag{۲۹-۱۰۸}$$

در این صورت L ممکن است اگر و فقط اگر مقدار هدف بهینه‌ی L_{aux} برابر با ۰ باشد.

اثبات فرض کنید L یک جواب ممکن $\bar{x} = (\bar{x}_1, \bar{x}_2, ..., \bar{x}_n)$ داشته باشد. در این صورت جواب $\bar{x}_0 = 0$ که با $\bar{x}$ ترکیب شده است، یک جواب ممکن برای L_{aux} با مقدار هدف ۰ است. چون $x_0 \geq 0$ یک محدودیت L_{aux} است، و تابع هدف $-x_0$ را بیشینه می‌کند، این جواب باید برای L_{aux} بهینه باشد. برعکس، فرض کنید که مقدار هدف بهینه‌ی L_{aux} برابر با ۰ باشد. آن گاه $\bar{x}_0 = 0$، و مقادیر متغیرهای باقی مانده‌ی $\bar{x}$ محدودیت‌های L را ارضا می‌کنند. ■

اکنون استراتژی خود را برای یافتن یک جواب ممکن اولیه برای یک برنامه‌ی خطی L به شکل استاندارد توضیح می‌دهیم:

```
INITIALIZE-SIMPLEX(A, b, c)
 1  let k be the index of the minimum b_i
 2  if b_l ≥ 0                    // is the initial basic solution feasible?
 3      return ({1, 2, ..., n}, {n + 1, n + 2, ..., n + m}, A, b, c, 0)
 4  form L_aux by adding -x_0 to the left-hand side of each constraint
        and setting the objective function to -x_0
 5  let (N, B, A, b, c, v) be the resulting slack form for L_aux
 6  l = n + k
 7  // L_aux has n + 1 nonbasic variables and m basic variables
 8  (N, B, A, b, c, v) = PIVOT(N, B, A, b, c, v, l, 0)
 9  // The basic solution is now feasible for L_aux
10  iterate the while loop of lines 3-12 of SIMPLEX until an optimal solution
        To L_aux is found
11  if the optimal solution to L_aux sets x̄_0 to 0
12      if x̄_0 is basic
13          perform one (degenerate) pivot to make it nonbasic
14      from the final slack form of L_aux, remove x_0 from the constraints and
            restore the original objective function of L, but replace each basic
            variable in this objective function by the right-hand side of its
            associated constraint
15      return the modified final slack form
16  else return "infeasible"
```

INITIALIZE-SIMPLEX به صورت زیر کار می‌کند. در خطوط ۱-۳، به صورت ضمنی جواب اولیه‌ی فرم ضعیف L را تست می‌کنیم، که توسط $N = \{1, 2, ..., n\}$، $B = \{n+1, n+2, ..., n+m\}$، $\bar{x}_i = b_i$ برای تمام $i \in B$، و $\bar{x}_j = 0$ برای تمام $j \in N$ داده شده است. (ساختن فرم ضعیف به هیچ تلاش مستقیمی نیاز ندارد، چرا که مقادیر A، b و c در هر دو فرم ضعیف و استاندارد یکی هستند.) اگر در خط ۲ مشخص شود که این جواب اولیه، ممکن هم هست – یعنی داشته باشیم $\bar{x}_i \geq 0$ برای هر $i \in N \cup B$ – آن گاه فرم ضعیف در خط ۳ بازگردانده می‌شود. در غیر این صورت در خط ۴ برنامه‌ی خطی کمکی L_{aux} را مانند لم ۲۹-۱۱ شکل می‌دهیم. چون جواب اولیه‌ی L یک جواب ممکن نیست، جواب اولیه‌ی L_{aux} هم ممکن نخواهد بود. برای یافتن یک جواب اولیه‌ی ممکن یک عملیات چرخش انجام می‌دهیم. خط ۶ مقدار $l = n+k$ را به عنوان اندیس متغیر پایه‌ای انتخاب می‌کند که در این عملیات چرخش، متغیر خروجی خواهد بود. چون متغیرهای اولیه عبارتند از $x_{n+1}, x_{n+2}, ..., x_{n+m}$، متغیر خروجی x_l متغیری است که منفی‌ترین مقدار را دارد. خط ۸

فراخوانی PIVOT را انجام می‌دهد، با x_0 به عنوان ورودی و x_l به عنوان خروجی. به زودی می‌بینیم که جواب اولیه‌ی حاصل از این فراخوانی PIVOT، ممکن خواهد بود. اکنون که یک فرم ضعیف داریم که جواب اولیه‌ی آن، ممکن است، می‌توانیم با فراخوانی مکرر PIVOT در خط ۱۰، برنامه‌ی خطی کمکی را به طور کامل حل کنیم. اگر یک جواب بهینه با مقدار ۰ برای L_{aux} یافته باشیم (که تست خط ۱۱ آن را بررسی می‌کند)، آن گاه در خط ۱۲-۱۴ یک فرم ضعیف برای L می‌سازیم که جواب اولیه‌ی آن، ممکن است. برای این کار، ابتدا در خطوط ۱۲-۱۳ به حالت انحطاط می‌پردازیم، که در آن ممکن است x_0 با مقدار $x_0 = 0$ همچنان متغیر پایه باشد. در این حالت با انجام یک مرحله‌ی چرخش، x_0 را از متغیرهای پایه حذف می‌کنیم. در این چرخش از یک $e \in N$ به عنوان متغیر ورودی استفاده می‌کنیم به طوری که $a_{0e} \neq 0$. جواب اولیه‌ی حاصل همچنان ممکن باقی خواهد ماند؛ چرخش انجام شده مقدار هیچ یک از متغیرها را تغییر نمی‌دهد. سپس تمام عبارات x_0 را از محدودیت‌ها حذف، و تابع هدف ابتدایی را برای L بازیابی می‌کنیم. تابع هدف اولیه ممکن است هم حاوی متغیرهای پایه باشد و هم متغیرهای غیرپایه. بنابراین در تابع هدف هر متغیر پایه را با عبارت سمت راست محدودیت مربوط به آن جایگزین می‌کنیم. سپس خط ۱۵ فرم ضعیف حاصل را بازمی‌گرداند. از طرف دیگر اگر خط ۱۱ دریابد که برنامه‌ی خطی ابتدایی ناممکن است، آن گاه خط ۱۶ این وضعیت را گزارش می‌کند.

اکنون عمل‌کرد INITIALIZE-SIMPLEX را بر روی برنامه‌ی خطی (۲۹-۱۰۲)-(۲۹-۱۰۵) نشان می‌دهیم. این برنامه‌ی خطی، یک برنامه‌ی خطی ممکن است اگر بتوانیم مقادیری نامنفی برای x_1 و x_2 بیابیم به طوری که نامساوی‌های (۲۹-۱۰۳) و (۲۹-۱۰۴) را ارضا کنند. با استفاده از لم ۲۹-۱۱ برنامه‌ی خطی کمکی زیر را تشکیل می‌دهیم:

$$\text{minimize} \quad -x_0 \tag{۲۹-۱۰۹}$$

با شرایط

$$2x_1 - x_2 - x_0 \leq 2 \tag{۲۹-۱۱۰}$$

$$x_1 - 5x_2 - x_0 \leq -4 \tag{۲۹-۱۱۱}$$

$$x_1, \; x_2, \; x_0 \geq 0$$

طبق لم ۲۹-۱۱ اگر مقدار هدف بهینه‌ی این برنامه‌ی خطی کمکی ۰ باشد، آن گاه برنامه‌ی خطی اصلی یک جواب ممکن دارد. اگر مقدار هدف بهینه‌ی این برنامه‌ی خطی کمکی مثبت باشد، آن گاه برنامه‌ی خطی اصلی جواب ممکن ندارد.

این برنامه‌ی خطی را به شکل ضعیف می‌نویسیم، که به دست می‌دهد

$$\begin{aligned} z &= -x_0 \\ x_3 &= 2 - 2x_1 + x_2 + x_0 \\ x_4 &= -4 - x_1 + 5x_2 + x_0 \end{aligned}$$

هنوز مشکل ما حل نشده است چرا که جواب اولیه، که در آن $x_4 = -4$، برای این برنامه‌ی خطی کمکی، ممکن نیست. با این حال می‌توانیم با یک بار فراخوانی PIVOT این فرم ضعیف را به صورتی تبدیل کنیم که جواب اولیه‌ی آن، ممکن است. همان طور که خط ۸ مشخص می‌کند، x_0 را به عنوان

متغیر ورودی انتخاب می‌کنیم. در خط ۶ متغیر x_4 را به عنوان متغیر خروجی انتخاب می‌کنیم، که همان متغیر پایه‌ای است که مقدار آن در جواب اولیه، منفی‌ترین است. پس از چرخش، فرم ضعیف زیر را خواهیم داشت:

$$\begin{aligned} z &= -4 - x_1 + 5x_2 - x_4 \\ x_0 &= 4 + x_1 - 5x_2 + x_4 \\ x_3 &= 6 - x_1 - 4x_2 + x_4 \end{aligned}$$

جواب اولیه‌ی مربوطه عبارت است از $(x_0, x_1, x_2, x_3, x_4) = (4, 0, 0, 6, 0)$، که یک جواب ممکن است. اکنون مکرراً PIVOT را فراخوانی می‌کنیم تا یک جواب برای L_{aux} به دست آوریم. در این حالت، یک فراخوانی PIVOT با x_2 به عنوان ورودی و x_0 به عنوان خروجی به دست می‌دهد:

$$\begin{aligned} z &= -x_0 \\ x_2 &= \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_3 &= \frac{14}{5} + \frac{4x_0}{5} - \frac{9x_1}{5} + \frac{x_4}{5} \end{aligned}$$

این فرم ضعیف، جواب نهایی مسئله‌ی کمکی است. از آن جایی که در جواب داریم $x_0 = 0$، می‌دانیم که مسئله‌ی اولیه‌ی ما ممکن است. به علاوه چون $x_0 = 0$، می‌توانیم به سادگی آن را از مجموعه‌ی محدودیت‌ها حذف کنیم. سپس می‌توانیم از تابع هدف ابتدایی استفاده کنیم، که با جایگزینی‌های مناسب این تابع فقط شامل متغیرهای غیرپایه خواهد بود. در این مثال تابع هدف زیر را خواهیم داشت:

$$2x_1 - x_2 = 2x_1 - \left(\frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5}\right)$$

با قرار دادن $x_0 = 0$ و ساده کردن، تابع هدف زیر را خواهیم داشت:

$$\frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5}$$

و همچنین فرم ضعیف زیر را:

$$\begin{aligned} z &= \frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5} \\ x_2 &= \frac{4}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_3 &= \frac{14}{5} - \frac{9x_1}{5} + \frac{x_4}{5} \end{aligned}$$

این فرم ضعیف یک جواب اولیه‌ی ممکن دارد و می‌توانیم آن را به رویه‌ی SIPLEX بازگردانیم. اکنون به صورت رسمی صحت INITIALIZE-SIMPLEX را نشان می‌دهیم.

لم ۲۹-۱۲ اگر یک برنامه‌ی خطی L جواب ممکن نداشته باشد آن گاه INITIALIZE-SIMPLEX عبارت "infeasible" را بازمی‌گرداند. در غیر این صورت، یک فرم ضعیف بازمی‌گرداند که جواب اولیه‌ی آن یک جواب ممکن است.

اثبات ابتدا فرض کنید که برنامه‌ی خطی L جواب ممکن نداشته باشد. در این صورت طبق لم ۲۹-۱۱ مقدار هدف بهینه‌ی L_{aux}، که در (۲۹-۱۰۶)-(۲۹-۱۰۹) تعریف شده است، غیر صفر است، و طبق محدودیت عدم منفی بودن بر روی x_0، مقدار هدف بهینه باید منفی باشد. به علاوه این مقدار هدف باید کران‌دار باشد، چرا که قرار دادن $x_i = 0$، برای $i = 1, 2, ..., n$، و $x_0 = \left|\min_{i=1}^{m}[b_i]\right|$ یک جواب ممکن به دست می‌دهد، و مقدار هدف این جواب $-\left|\min_{i=1}^{m}[b_i]\right|$ است. بنابراین خط ۱۰ رویه‌ی INITIALIZE-SIMPLEX یک جواب با یک مقدار هدف منفی می‌یابد. فرض کنید $\bar{x}$ جواب اولیه‌ی مربوط به فرم ضعیف نهایی باشد. نمی‌توانیم داشته باشیم $\bar{x}_0 = 0$، چرا که در این صورت L_{aux} مقدار هدف ۰ خواهد داشت، که با منفی بودن مقدار هدف تناقض دارد. بنابراین تست خط ۱۱ باعث می‌شود که "infeasible" در خط ۱۶ بازگردانده شود.

اکنون فرض کنید که برنامه‌ی خطی L یک جواب ممکن داشته باشد. از تمرین ۲۹-۳-۳ می‌دانیم که اگر $b_i \geq 0$، برای $i = 1, 2, ..., m$، آن گاه جواب اولیه‌ی مربوط به فرم ضعیف اولیه، یک جواب ممکن است. در این حالت خطوط ۲-۳ فرم ضعیف مربوط به ورودی را بازمی‌گردانند. (تبدیل فرم استاندارد به فرم ضعیف ساده است، چرا که A، b، و c در هر دو یکی هستند.)

در ادامه‌ی اثبات حالتی را بررسی می‌کنیم که در آن برنامه‌ی خطی ممکن است، ولی در خط ۳ بازگشت انجام نمی‌شود. بحث خواهیم کرد که در این حالت خطوط ۴-۱۰ یک جواب ممکن برای L_{aux} می‌یابند که مقدار هدف آن ۰ است. ابتدا، طبق خطوط ۱-۲ باید داشته باشیم

$$b_l < 0$$

و

$$b_l \leq b_i \qquad \text{برای هر } i \in B \tag{۲۹-۱۱۲}$$

در خط ۸ یک عملیات چرخش انجام می‌دهیم که در آن متغیر خروجی x_l، سمت چپ تساوی با کوچک‌ترین b_i است، و متغیر ورودی x_0 همان متغیر اضافه شده است. اکنون نشان می‌دهیم که بعد از چرخش تمام ورودی‌های b نامنفی هستند، و بنابراین جواب اولیه‌ی L_{aux} یک جواب ممکن است. با فرض این که $\bar{x}$ جواب اولیه بعد از فراخوانی PIVOT باشد، و $\hat{b}$ و $\hat{B}$ مقادیر بازگردانده شده توسط آن، لم ۲۹-۱ ایجاب می‌کند که

$$\bar{x}_i = \begin{cases} b_i - a_{ie}\hat{b}_e & \text{اگر } i \in \hat{B} - \{e\} \\ b_l / a_{le} & \text{اگر } i = e \end{cases} \tag{۲۹-۱۱۳}$$

در فراخوانی PIVOT در خط ۸ داریم $e = 0$. اگر نامساوی‌های (۲۹-۱۰۷) را طوری بازنویسی کنیم که

شامل ضرایب a_{i0} شود،

$$\sum_{j=0}^{n} a_{ij} x_j \le b_i \qquad \text{برای } i = 1, 2, \ldots, m \tag{۲۹-۱۱۴}$$

آن گاه

$$a_{i0} = a_{ie} = -1 \qquad \text{برای هر } i \in B \tag{۲۹-۱۱۵}$$

(توجه کنید که a_{i0} ضریب x_0 در (۲۹-۱۱۰) است، و نه منفی ضریب، چرا که L_{aux} یک فرم استاندارد است نه ضعیف.) چون $l \in B$، همچنین داریم $a_{le} = -1$. بنابراین $b_l / a_{le} > 0$، و از این رو $x_e > 0$. برای متغیرهای پایه‌ی باقی مانده داریم

$$\begin{aligned}
\bar{x}_i &= b_i - a_{ie}\hat{b}_e && \text{(طبق تساوی (۲۹-۱۱۶))} \\
&= b_i - a_{ie}(b_l / a_{le}) && \text{(طبق خط ۲ رویه‌ی PIVOT)} \\
&= b_i - b_l && (a_{le} = -1 \text{ و طبق تساوی (۲۹-۱۱۷))} \\
&\ge 0 && \text{(طبق نامساوی (۲۹-۱۱۵))}
\end{aligned}$$

که اکنون ایجاب می‌کند هر متغیر پایه نامنفی باشد. پس جواب اولیه بعد از فراخوانی PIVOT در خط ۸، ممکن است. سپس خط ۱۰ را اجرا می‌کنیم که L_{aux} را حل می‌کند. از آن جایی که فرض کردیم L یک جواب ممکن دارد، لم ۲۹-۱۱ ایجاب می‌کند که L_{aux} یک جواب بهینه با مقدار هدف ۰ داشته باشد. چون تمام فرم‌های ضعیف با یکدیگر معادل هستند، در جواب نهایی L_{aux} باید داشته باشیم $\bar{x}_0 = 0$، و پس از حذف x_0 از برنامه‌ی خطی، یک فرم ضعیف به دست می‌آوریم که برای L ممکن است. سپس این فرم ضعیف در خط ۱۵ بازگردانده می‌شود.

قضیه‌ی اصلی برنامه‌ریزی خطی

این فصل را با نشان دادن این که رویه‌ی SIMPLEX به درستی کار می‌کند به پایان می‌بریم. به طور خاص، هر برنامه‌ی خطی یا ناممکن است، یا بی‌کران است، و یا یک جواب بهینه با مقدار هدف کران‌دار دارد، که در تمام این حالت‌ها SIMPLEX به درستی عمل می‌کند.

قضیه‌ی ۲۹-۱۳
(قضیه‌ی اصلی برنامه‌ریزی خطی)

هر برنامه‌ی خطی L که به شکل استاندارد داده شده است، یا

۱. یک جواب بهینه با مقدار هدف کران‌دار دارد، یا
۲. ناممکن است، و یا
۳. بی‌کران است.

اگر L ناممکن باشد رویه‌ی SIMPLEX مقدار "infeasible" را بازمی‌گرداند. اگر L بی‌کران باشد SIMIPLEX مقدار "unbounded" را بازمی‌گرداند. در غیر این صورت SIMPLEX یک جواب بهینه با یک مقدار هدف کران‌دار بازمی‌گرداند.

اثبات طبق لم ۲۹-۱۲، اگر برنامه‌ی خطی L ناممکن باشد آن گاه SIMPLEX مقدار "infeasible" را

بازمی‌گرداند. اکنون فرض کنید که برنامه‌ی خطی L ممکن است. طبق لم ۲۹-۱۲، INITIALIZE-SIMPLEX یک فرم ضعیف بازمی‌گرداند که جواب اولیه‌ی آن یک جواب ممکن است. بنابراین طبق لم ۲۹-۷، رویه‌ی SIMPLEX یا مقدار "unbounded" را بازمی‌گرداند و یا با یک جواب ممکن کار را به پایان می‌برد. اگر با یک جواب کران‌دار پایان یابد، آن گاه قضیه‌ی ۲۹-۱۰ می‌گوید که این جواب یک جواب بهینه است. از سوی دیگر اگر SIMPLEX مقدار "unbounded" را بازگرداند، لم ۲۹-۲ می‌گوید که برنامه‌ی خطی L بی‌کران است. از آن جایی که SIMPLEX همیشه به یکی از این روش‌ها پایان می‌یابد، اثبات کامل است. ■

تمرین‌ها

۲۹-۵-۱ یک شبه‌کد با جزئیات کامل برای پیاده‌سازی خط ۵ و ۱۴ رویه‌ی INITIALIZE-SIMPLEX ارائه کنید.

۲۹-۵-۲ نشان دهید که وقتی INITIALIZE-SIMPLEX حلقه‌ی اصلی SIMPLEX را اجرا می‌کند، هیچ‌گاه "unbounded" بازگردانده نمی‌شود.

۲۹-۵-۳ فرض کنید به ما یک برنامه‌ی خطی L به شکل استاندارد داده شده است، و برای L و دوگان L جواب‌های اولیه‌ی مربوط به فرم ضعیف اولیه، ممکن است. نشان دهید که مقدار هدف بهینه‌ی L برابر ۰ است.

۲۹-۵-۴ فرض کنید که اجازه می‌دهیم نامساوی‌های مطلق در یک برنامه‌ی خطی حضور داشته باشند. نشان دهید که در این حالت قضیه‌ی اصلی برنامه‌ریزی خطی برقرار نیست.

۲۹-۵-۵ با استفاده از SIMPLEX برنامه‌ی خطی زیر را حل کنید:

$$\text{minimize} \quad x_1 + 3x_2$$

با شرایط

$$\begin{aligned} -x_1 + x_2 &\le -1 \\ -2x_1 - 2x_2 &\le -6 \\ -x_1 + 4x_2 &\le 2 \\ x_1, \quad x_2 &\ge 0 \end{aligned}$$

۲۹-۵-۶ با استفاده از SIMPLEX برنامه‌ی خطی زیر را حل کنید:

$$\text{minimize} \quad x_1 - 2x_2$$

با شرایط

$$\begin{aligned} x_1 + 2x_2 &\le 4 \\ -2x_1 - 6x_2 &\le -12 \\ x_2 &\le 1 \\ x_1, \quad x_2 &\ge 0 \end{aligned}$$

۲۹-۵-۷ با استفاده از SIMPLEX برنامه‌ی خطی زیر را حل کنید:

minimize $x_1 + 3x_2$

با شرایط

$$\begin{aligned} -x_1 + x_2 &\le -1 \\ -x_1 - x_2 &\le -3 \\ -x_1 + 4x_2 &\le 2 \\ x_1, \quad x_2 &\ge 0 \end{aligned}$$

۲۹-۵-۸ برنامه‌ی خطی داده شده در (۲۹-۶)-(۲۹-۱۰) را حل کنید.

۲۹-۵-۹ برنامه‌ی خطی تک متغیره‌ی زیر را در نظر بگیرید، که آن را P می‌نامیم:

minimize tx

با شرایط

$$\begin{aligned} rx &\le s \\ x &\ge 0 \end{aligned}$$

که در آن r، s، و t اعداد حقیقی دلخواه هستند. فرض کنید D دوگان P باشد. تعیین کنید که برای چه مقادیری از r، s، و t می‌توانید هر یک از عبارت‌های زیر را نتیجه بگیرید:

۱. هر دوی P و D دارای جواب‌های بهینه با مقدار هدف کران‌دار هستند.
۲. P ممکن است، ولی D ناممکن است.
۳. D ممکن است، ولی P ناممکن است.
٤. نه P ممکن است و نه D.

مسائل

۲۹-۱ ***ممکن بودن نامساوی‌های خطی***

با داشتن مجموعه‌ای از m نامساوی بر روی n متغیر $x_1, x_2, ..., x_n$، *مسئله‌ی ممکن بودن نامساوی‌های خطی* عبارت است از تعیین این که آیا مجموعه‌ای از مقادیر برای این متغیرها وجود دارد که تمام نامساوی‌ها را به طور هم زمان ارضا کند یا خیر.

I. نشان دهید که اگر یک الگوریتم برای برنامه‌ریزی خطی داشته باشیم، می‌توانیم از آن برای حل مسئله‌ی ممکن بودن نامساوی‌های خطی استفاده کنیم. تعداد متغیرها و محدودیت‌هایی که در مسئله‌ی برنامه‌ی خطی استفاده می‌کنید باید نسبت به n و m از مرتبه‌ی چندجمله‌ای باشند.

II. نشان دهید که اگر یک الگوریتم برای مسئله‌ی ممکن بودن نامساوی‌های خطی داشته باشیم، می‌توانیم از آن برای حل مسئله‌ی برنامه‌ریزی خطی استفاده کنیم. تعداد متغیرها و محدودیت‌هایی که در مسئله‌ی ممکن بودن نامساوی‌های خطی استفاده می‌کنید باید نسبت به n و m، تعداد متغیرها و محدودیت‌های برنامه‌ی خطی، از مرتبه‌ی چندجمله‌ای باشند.

۲-۲۹ ضعف مکمل

ضعف مکمل (complementary slackness) یک رابطه توصیف می‌کند میان مقدار متغیرهای اولیه و محدودیت‌های دوگان، و همچنین بین مقدار متغیرهای دوگان و محدودیت‌های اولیه. فرض کنید $\bar{x}$ یک جواب بهینه برای برنامه‌ی خطی اولیه‌ی داده شده در (۲۹-۱۶)-(۲۹-۱۸) باشد، و $\bar{y}$ یک جواب بهینه برای برنامه‌ی خطی دوگان داده شده در (۲۹-۸۶)-(۲۹-۸۸). ضعف مکمل می‌گوید که شرایط زیر، لازم و کافی هستند برای این که $\bar{x}$ و $\bar{y}$ بهینه باشند:

$$\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j \text{ یا } \bar{x}_j = 0 \quad \text{برای } j = 1, 2, \ldots, n$$

و

$$\sum_{i=1}^{n} a_{ij}\bar{x}_j = b_i \text{ یا } \bar{y}_i = 0 \quad \text{برای } i = 1, 2, \ldots, m$$

i. نشان دهید که ضعف مکمل برای برنامه‌ی خطی (۲۹-۵۳)-(۲۹-۵۷) برقرار است.

ii. اثبات کنید که ضعف مکمل برای هر برنامه‌ی خطی اولیه و مکمل آن برقرار است.

iii. اثبات کنید که یک جواب ممکن $\bar{x}$ به یک برنامه‌ی خطی اولیه که در خطوط (۲۹-۱۶)-(۲۹-۱۸) داده شده است، بهینه است اگر و فقط اگر مقادیر $\bar{y} = (\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_m)$ موجود باشند به طوری که

۱. $\bar{y}$ یک جواب ممکن برای برنامه‌ی خطی دوگان داده شده در (۲۹-۸۳)-(۲۹-۸۵) باشد،

۲. $\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j$ هر گاه که $\bar{x}_j > 0$، و

۳. $\bar{y}_i = 0$ هر گاه که $\sum_{j=1}^{n} a_{ij}\bar{x}_j < b_i$.

۳-۲۹ برنامه‌ریزی خطی صحیح

یک مسئله‌ی برنامه‌ریزی خطی صحیح، یک مسئله‌ی برنامه‌ریزی خطی است با این محدودیت اضافه که متغیرهای x فقط می‌توانند اعداد صحیح را بپذیرند. تمرین ۳۴-۵-۳ نشان می‌دهد که فقط تعیین این که آیا یک مسئله‌ی برنامه‌ریزی خطی صحیح دارای جواب ممکن است یا خیر، یک مسئله‌ی NP-کامل است، که بدین معنی است که احتمالاً هیچ الگوریتم چندجمله‌ای برای این مسئله وجود ندارد.

i. نشان دهید که دوگانگی ضعیف (لم ۲۹-۸) برای یک برنامه‌ی خطی صحیح برقرار است.

ii. نشان دهید که دوگانگی (قضیه‌ی ۲۹-۱۰) لزوماً برای یک برنامه‌ی خطی صحیح برقرار نیست.

iii. با داشتن یک برنامه‌ی خطی اولیه به شکل استاندارد، اجازه دهید P را به صورت مقدار هدف بهینه برای برنامه‌ی خطی اولیه، D را به صورت مقدار هدف بهینه برای دوگان آن، IP را به صورت مقدار هدف بهینه برای نسخه‌ی صحیح برنامه‌ی خطی اولیه (یعنی، برنامه‌ی خطی اولیه با این محدودیت‌ها که متغیرها در آن مقادیر صحیح به خود می‌گیرند)، و

ID را به صورت مقدار هدف بهینه برای نسخه‌ی صحیح دوگان تعریف کنیم. با فرض این که هر دوی برنامه‌های خطی صحیح اولیه و دوگان، ممکن و کران‌دار هستند، نشان دهید که

$$IP \le P = D \le ID$$

۲۹-۴ لم *Farkas*

فرض کنید A یک ماتریس $m \times n$ و b یک بردار mتایی باشد. در این صورت لم Farkas می‌گوید که دقیقاً یکی از سیستم‌های

$$Ax \le 0,$$
$$bx > 0$$

و

$$yA = b,$$
$$y \ge 0$$

قابل حل هستند، که در آن x یک بردار nتایی و y یک بردار mتایی است. لم Farkas را اثبات کنید.

۲۹-۲ *گردش با کم‌ترین هزینه*

در این مسئله نسخه‌ای از مسئله‌ی شار با کم‌ترین هزینه از بخش ۲۹-۲ را در نظر می‌گیریم، که در آن نه منبع وجود دارد، نه چاهک، و نه مقدار شار مورد نیاز. در عوض مانند قبل یک شبکه‌ی شار داریم، با هزینه‌ی $a(u,v)$ برای یال‌ها. یک شار، ممکن است اگر محدودیت ظرفیت را روی تمام یال‌ها و بقای شار را روی *تمام* رأس‌ها ارضا کند. هدف این است که از میان تمام شارهای ممکن، شار با کم‌ترین هزینه را بیابیم. این مسئله را مسئله‌ی گردش با کم‌ترین هزینه (minimum-cost-circulation problem) می‌نامیم.

I. مسئله‌ی گردش با کم‌ترین هزینه را به صورت یک برنامه‌ی خطی فرمول‌بندی کنید.

II. فرض کنید برای هر یال $(u,v) \in E$ داریم $a(u,v) > 0$. جواب بهینه‌ی مسئله‌ی گردش با کم‌ترین هزینه را توصیف کنید.

III. مسئله‌ی شار بیشینه را به صورت یک برنامه‌ی خطی مربوط به یک مسئله‌ی گردش با کم‌ترین هزینه توصیف کنید. یعنی با داشتن یک نمونه‌ی $G = (V, E)$ از مسئله‌ی شار بیشینه با منبع s، چاهک t و ظرفیت یال c، یک مسئله‌ی چرخش با کم‌ترین هزینه ارائه کنید، با یک شبکه‌ی (نه لزوماً یکسان) $G' = (V', E')$ با ظرفیت یال c' و هزینه‌ی a' به طوری که بتوان از یک جواب برای مسئله‌ی چرخش با کم‌ترین هزینه، به یک جواب به مسئله‌ی شار بیشینه رسید.

IV. مسئله‌ی کوتاه‌ترین مسیرها از یک مبدأ را به صورت یک برنامه‌ی خطی مربوط به یک مسئله‌ی گردش با کم‌ترین هزینه توصیف کنید.

چند جمله‌ای‌ها و تبدیل تبدیل سریع فوریه

متد سرراست جمع دو چندجمله‌ای از درجه‌ی n به زمان $\theta(n)$ نیاز دارد، ولی زمان اجرای متد سرراست ضرب دو چندجمله‌ای از درجه‌ی n از مرتبه‌ی $\theta(n^2)$ است. در این فصل نشان خواهیم داد که چگونه تبدیل سریع فوریه، یا FFT، می‌تواند زمان ضرب دو چندجمله‌ای را به $\theta(n \lg n)$ کاهش دهد.

معمول‌ترین استفاده از تبدیل فوریه، و بنابراین FFT، در پردازش سیگنال‌ها است. یک سیگنال در یک *دامنه‌ی زمانی* (time domain) داده می‌شود؛ یعنی به صورت یک تابع که زمان را به یک مقدار نگاشت می‌کند. تحلیل فوریه به ما اجازه می‌دهد که سیگنال را به صورت مجموع وزن‌دار سینوس‌هایی با شیفت زمانی و فرکانس‌های مختلف نشان دهیم. وزن‌ها و شیفت‌های مربوط به فرکانس‌های مختلف، سیگنال را در *دامنه‌ی فرکانس* (frequency domain) توصیف می‌کنند. از میان کاربردهای روزانه‌ی بسیار FFT می‌توان به تکنیک‌های فشرده‌سازی مورد استفاده برای اطلاعات صوتی و تصویری دیجیتال، از جمله فایل‌های MP3 اشاره کرد. پردازش سیگنال مبحثی غنی است که کتاب‌های خوب مختلفی به آن پرداخته‌اند.

چندجمله‌ای‌ها

یک *چندجمله‌ای* (polynomial) نسبت به متغیر x بر روی یک حوزه‌ی جبری F، نمایشی است از یک تابع $A(x)$ به صورت یک جمع:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

به مقادیر a_0، a_1، ...، a_{n-1}، *ضرایب* (coefficient) چندجمله‌ای گفته می‌شود. ضرایب به یک حوزه‌ی F تعلق دارند، که این حوزه معمولاً مجموعه‌ی اعداد مختلط C است. می‌گوییم یک چندجمله‌ای $A(x)$ دارای درجه‌ی k است اگر بالاترین ضریب غیرصفر آن a_k باشد؛ می‌نویسیم $\text{degree}(A) = k$. هر

عدد صحیحی بزرگ‌تر از درجه‌ی یک چندجمله‌ای، یک *کران درجه* (degree-bound) برای آن چندجمله‌ای است. بنابراین درجه‌ی یک چندجمله‌ای با کران درجه‌ی n می‌تواند هر عدد صحیحی بین (و شامل) ۰ و $n-1$ باشد.

می‌توان اعمال مختلفی بر روی چندجمله‌ای‌ها تعریف کرد. برای *جمع چندجمله‌ای‌ها*، اگر $A(x)$ و $B(x)$ دو چندجمله‌ای با کران درجه‌ی n باشند، می‌گوییم *جمع* آن‌ها یک چندجمله‌ای $C(x)$ با همان کران درجه‌ی n است، به طوری که $C(x)=A(x)+B(x)$ برای تمام xها در حوزه‌ی مربوطه. یعنی اگر

$$A(x)=\sum_{j=0}^{n-1} a_j x^j$$

و

$$B(x)=\sum_{j=0}^{n-1} b_j x^j$$

آن گاه

$$C(x)=\sum_{j=0}^{n-1} c_j x^j$$

که در آن $c_j=a_j+b_j$ برای $j=0,1,\ldots,n-1$. به عنوان مثال اگر $A(x)=6x^3+7x^2-10x+9$ و $B(x)=-2x^3+4x-5$، آن گاه $C(x)=4x^3+7x^2-6x+4$.

برای *ضرب چندجمله‌ای‌ها*، اگر $A(x)$ و $B(x)$ دو چندجمله‌ای با کران درجه‌ی n باشند، می‌گوییم *ضرب* آن‌ها یک چندجمله‌ای $C(x)$ با کران درجه‌ی $2n-1$ است، به طوری که $C(x)=A(x)B(x)$ برای تمام xها در حوزه‌ی مربوطه. احتمالاً شما قبلاً چندجمله‌ای‌ها را در یکدیگر ضرب کرده‌اید، با ضرب هر جمله‌ی $A(x)$ در هر جمله‌ی $B(x)$ و ترکیب جمله‌های با درجه‌ی یکسان. مثلاً می‌توانیم $A(x)=6x^3+7x^2-10x+9$ و $B(x)=-2x^3+4x-5$ را به صورت زیر در یکدیگر ضرب کنیم:

$$\begin{array}{r}
6x^3+7x^2-10x+9\\
-2x^3+4x-5\\
\hline
-30x^3-35x^2+50x-45\\
24x^4+28x^3-40x^2+36x\\
-12x^6-14x^5+20x^4-18x^3\\
\hline
-12x^6-14x^5+44x^4-20x^3-75x^2+86x-45
\end{array}$$

یک روش دیگر برای توصیف ضرب $C(x)$ به صورت زیر است:

$$C(x)=\sum_{j=0}^{2n-2} c_j x^j \qquad (۱-۳۰)$$

که در آن

$$c_j = \sum_{k=0}^{j} a_k b_{j-k} \qquad (۳۰-۲)$$

توجه کنید که $\text{degree}(C) = \text{degree}(A) + \text{degree}(B)$، که ایجاب می‌کند

$$\begin{aligned}\text{degree-bound}(C) &= \text{degree-bound}(A) + \text{degree-bound}(B) - 1 \\ &\le \text{degree-bound}(A) + \text{degree-bound}(B)\end{aligned}$$

با این حال کران درجه‌ی C را به صورت جمع کران درجه‌ی A و B در نظر می‌گیریم، چرا که اگر کران درجه‌ی یک چندجمله‌ای k باشد، $k+1$ هم خواهد بود.

خلاصه‌ی فصل

بخش ۳۰-۱ دو روش برای نمایش چندجمله‌ای‌ها ارائه می‌کند: نمایش ضرایب و نمایش نقطه-مقدار. متد سرراست ضرب چندجمله‌ای‌ها – تساوی‌های (۳۰-۱) و (۳۰-۲) – وقتی چندجمله‌ای‌ها به صورت نمایش ضرایب هستند، به $\theta(n^2)$ زمان نیاز دارد، ولی اگر چندجمله‌ای‌ها به صورت نمایش نقطه- مقدار باشند، فقط $\theta(n)$ زمان خواهد برد. با این حال وقتی که چندجمله‌ای‌ها به صورت نمایش ضرایب هستند، می‌توانیم با استفاده از تبدیل میان نمایش‌ها، آن‌ها را در زمان $\theta(n \lg n)$ ضرب کنیم. برای این که ببینیم چگونه این کار انجام می‌شود، ابتدا باید ریشه‌های مختلط واحد را بیاموزیم، که این کار را در بخش ۳۰-۲ انجام خواهیم داد. سپس از FFT و معکوس آن، که آن‌ها هم در بخش ۳۰-۲ توصیف خواهند شد، استفاده می‌کنیم تا تبدیل‌ها را انجام دهیم. بخش ۳۰-۳ نشان می‌دهد که چطور در هر دو مدل سریال و موازی، FFT را به صورت بهینه پیاده‌سازی کنیم.

در این فصل از اعداد مختلط به صورت وسیع استفاده خواهد شد، و نماد از i منحصرا برای نمایش $\sqrt{-1}$ بهره خواهیم برد.

۳۰-۱ نمایش چندجمله‌ای‌ها

نمایش‌های ضرایب و نقطه- مقدار برای چندجمله‌ای‌ها از یک نظر با هم یکی هستند؛ یک چندجمله‌ای در نمایش نقطه- مقدار یک همتا در نمایش ضرایب دارد. در این بخش این دو نمایش را معرفی می‌کنیم، و نشان می‌دهیم که چگونه می‌توان با ترکیب این دو، ضرب دو چندجمله‌ای با کران درجه‌ی n را در زمان $\theta(n \lg n)$ انجام داد.

نمایش ضرایب

نمایش ضرایب (coefficient representation) یک چندجمله‌ای $A(x) = \sum_{j=0}^{n-1} a_j x^j$ با کران درجه‌ی n، یک بردار از ضرایب $a = (a_0, a_1, \ldots, a_{n-1})$ است. در تساوی‌های ماتریسی در این فصل، به طور کلی بردارها را به صورت بردارهای ستونی در نظر می‌گیریم.

بنابراین ارزیابی و درون‌یابی n نقطه، اعمال معکوس خوش تعریفی هستند که جابه‌جایی بین نمایش‌های ضرایب و نقطه-مقدار را انجام می‌دهند.[1] الگوریتم‌های توصیف شده در بالا برای این مسائل از مرتبه‌ی زمانی $\theta(n^2)$ هستند.

نمایش نقطه- مقدار برای بسیاری از اعمال بر روی چندجمله‌ای‌ها کاملاً مناسب است. برای جمع، اگر $C(x)=A(x)+B(x)$، آن گاه برای هر نقطه‌ی x_k داریم $C(x_k)=A(x_k)+B(x_k)$. به طور دقیق‌تر اگر یک نمایش نقطه- مقدار

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

برای چندجمله‌ای A، و یک نمایش نقطه- مقدار

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{n-1}, y'_{n-1})\}$$

برای B داشته باشیم (توجه کنید که A و B بر روی n نقطه‌ی *یکسان* ارزیابی شده‌اند)، آن گاه یک نمایش نقطه- مقدار برای C می‌تواند

$$\{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \ldots, (x_{n-1}, y_{n-1} + y'_{n-1})\}$$

باشد. بنابراین زمان مورد نیاز برای جمع دو چندجمله‌ای با کران درجه‌ی n در نمایش نقطه-مقدار $\theta(n)$ است.

به طور مشابه، نمایش نقطه- مقدار برای ضرب چندجمله‌ای‌ها هم مناسب است. اگر $C(x)=A(x)B(x)$، آن گاه $C(x_k)=A(x_k)B(x_k)$ برای هر نقطه‌ی x_k، و می‌توان با ضرب جفت‌های متناظر در نمایش نقطه-مقدار A و B، یک نمایش نقطه-مقدار برای C به دست آورد. با این حال مسئله‌ای که وجود دارد این است که کران درجه‌ی C برابر است با مجموع کران درجه‌های A و B. یک نمایش نقطه-مقدار استاندارد برای A و B شامل n جفت نقطه-مقدار برای هر یک از چندجمله‌ای‌ها است. ضرب این دو به ما n جفت نقطه- مقدار برای C خواهد داد، ولی از آن جایی که کران درجه‌ی C برابر $2n$ است، برای یک نمایش نقطه- مقدار از C به $2n$ جفت نقطه-مقدار نیاز داریم. (تمرین ۳۰-۱-۴ را ببینید.) بنابراین باید با نمایش نقطه-مقدار «گسترش یافته»ی A و B آغاز کنیم، که شامل $2n$ جفت نقطه- مقدار برای هر چندجمله‌ای است. با داشتن یک نمایش نقطه-مقدار گسترش یافته برای A،

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{2n-1}, y_{2n-1})\}$$

و یک نمایش نقطه-مقدار گسترش یافته‌ی متناظر برای B،

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{2n-1}, y'_{2n-1})\}$$

یک نمایش نقطه- مقدار برای C عبارت است از

---

[1] مسلماً درون‌یابی از نقطه نظر پایداری عددی مسئله‌ی چالش برانگیزی است. با این که متدهای توصیف شده در این جا از نظر ریاضی صحیح هستند، تفاوت‌های کوچک در ورودی‌ها و یا خطاهای گرد کردن حین محاسبه می‌توانند تفاوت‌های بزرگی در نتیجه ایجاد کند.

$$\{(x_0, y_0 y_0'), (x_1, y_1 y_1'), \ldots, (x_{2n-1}, y_{2n-1} y_{2n-1}')\}$$

با داشتن دو چندجمله‌ای به شکل نمایش نقطه-مقدار گسترش یافته، می‌بینیم که زمان ضرب آن‌ها برای به دست آوردن نمایش نقطه-مقدار حاصل، $\theta(n)$ است، که بسیار کم‌تر است از زمان مشابه در نمایش ضرایب.

در نهایت مسئله‌ی ارزیابی یک چندجمله‌ای داده شده در نمایش نقطه- مقدار را در یک نقطه‌ی جدید در نظر می‌گیریم. ظاهرا برای این مسئله، هیچ روشی ساده‌تر از تبدیل چندجمله‌ای به نمایش ضرایب و سپس ارزیابی نقطه‌ی جدید وجود ندارد.

ضرب سریع چندجمله‌ای‌ها در نمایش ضرایب

آیا می‌توان برای تسریع ضرب چندجمله‌ای‌ها در نمایش ضرایب، از متد ضرب در زمان خطی در نمایش نقطه- مقدار استفاده کرد؟ جواب تماماً به توانایی ما در تبدیل سریع نمایش ضراب به نمایش نقطه- مقدار (ارزیابی) و بالعکس (درون‌یابی) بستگی دارد.

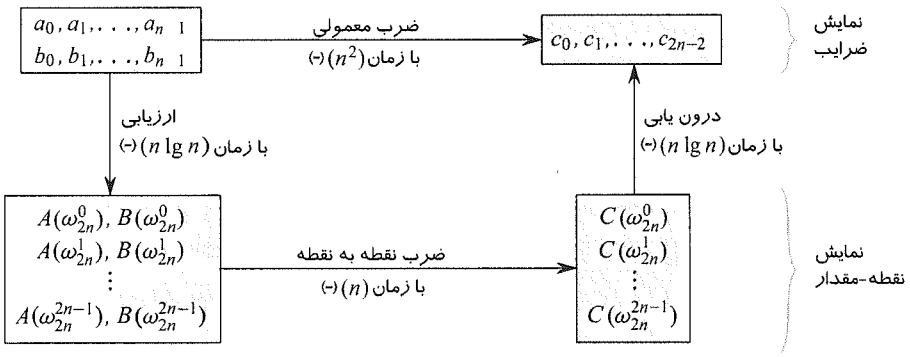
می‌توانیم از هر نقطه‌ی دلخواهی به عنوان نقاط ارزیابی استفاده کنیم، ولی با انتخاب مناسب نقاط ارزیابی می‌توانیم در زمان $\theta(n \lg n)$ تبدیل میان نمایش‌ها را انجام دهیم. همان طور که در بخش ۳۰-۲ خواهیم دید، اگر «ریشه‌های مختلط واحد» را به عنوان نقاط ارزیابی انتخاب کنیم، می‌توانیم با گرفتن تبدیل فوریه‌ی گسسته (یا DFT) یک بردار ضرایب، یک نمایش نقطه- مقدار تولید کنیم. عملیات بازگشت، یعنی درون‌یابی، را می‌توان با گرفتن «معکوس DFT» جفت‌های نقطه- مقدار انجام داد، که بردار ضرایب را نتیجه می‌دهد. بخش ۳۰-۲ نشان خواهد داد که چگونه می‌توانیم اعمال FFT، DFT، و معکوس DFT را در زمان $\theta(n \lg n)$ انجام دهیم.

شکل ۳۰-۱ این استراتژی را به صورت تصویری نشان می‌دهد. یکی از جزئیات فرعی، کران درجه است. کران درجه‌ی حاصل‌ضرب دو چندجمله‌ای با کران درجه‌ی n برابر با $2n$ است. بنابراین قبل از ارزیابی چندجمله‌ای‌های ورودی A و B، ابتدا کران درجه‌ی آن‌ها را با اضافه کردن n ضریب ۰ برای درجه‌های بالا به $2n$ افزایش می‌دهیم. چون بردارها $2n$ عنصر دارند، از « $2n$امین ریشه‌ی مختلط واحد» استفاده می‌کنیم، که با عبارات w_{2n} در شکل ۳۰-۱ نشان داده شده‌اند.

با داشتن FFT، رویه‌ی زیر با زمان $\theta(n \lg n)$ دو چندجمله‌ای $A(x)$ و $B(x)$ با کران درجه‌ی n را در هم ضرب می‌کند، که در آن ورودی و خروجی به صورت نمایش ضرایب هستند. فرض می‌کنیم n توانی از ۲ است؛ همیشه می‌توان با اضافه کردن ضرایب صفر با درجه‌های بالاتر به این خاصیت دست یافت.

۱. *دوبرابر کردن کران درجه:* با اضافه کردن n ضریب صفر با درجه‌ی بالا، کران درجه‌ی نمایش ضرایب $A(x)$ و $B(x)$ را به $2n$ تبدیل می‌کنیم.

۲. *ارزیابی:* نمایش نقطه-مقدار $A(x)$ و $B(x)$ با طول $2n$ را به وسیله‌ی دو بار FFT از مرتبه‌ی $2n$ محاسبه می‌کنیم. این نمایش‌ها شامل مقدار دو چندجمله‌ای در $2n$امین ریشه‌ی واحد هستند.



شکل ۳۰-۱ یک شمای گرافیکی از یک فرآیند بهینه‌ی ضرب چندجمله‌ای‌ها. نمایش‌های بالا بـه فـرم ضرایب هستند، و نمایش‌های پایینی به فرم نقطه-مقدار. پیکان‌های از چپ به راست مربوط بـه عملیـات ضرب هستند، و عبارت‌های w_{2n} نشان‌دهنده‌ی $2n$امین ریشه‌ی مختلط واحد.

۳. *ضرب نقطه به نقطه:* نمایش نقطه-مقدار چندجمله‌ای $C(x) = A(x)B(x)$ را با ضرب نقطه به نقطه‌ی این چندجمله‌ای‌ها در یکدیگر محاسبه می‌کنیم. این نمایش حاوی مقدار $C(x)$ در هـر $2n$امین ریشه‌ی واحد است.

۴. *درون‌یابی:* نمایش ضرایب $C(x)$ را به کمک یک بـار FFT بـر روی $2n$ جفـت نقطـه-مقـدار می‌سازیم تا به کمک آن معکوس تبدیل فوریه‌ی گسسته را محاسبه کنیم.

مراحل (۱) و (۳) در زمان $\Theta(n)$ اجرا می‌شوند، و مراحل (۲) و (۴) در زمـان $\Theta(n \lg n)$. بنـابراین تنها چیزی که برای اثبات قضیه‌ی زیر نیاز داریم، این است که نشـان دهـیم چگونـه مـی‌تـوان از FFT استفاده کرد.

قضیه ۳۰-۱ ضرب دو چندجمله‌ای با کران درجه‌ی n را می‌توان در زمان $\Theta(n \lg n)$ انجـام داد، کـه در آن هم ورودی و هم خروجی به فرم نمایش ضرایب هستند.

تمرین‌ها

۳۰-۱-۱ بـا اسـتفاده از تسـاوی‌هـای (۳۰-۱) و (۳۰-۲) حاصـل ضـرب چندجملـه‌ای‌هـای $A(x) = 7x^3 - x^2 + x - 10$ و $B(x) = 8x^3 - 6x + 3$ را محاسبه کنید.

۳۰-۱-۲ ارزیابی یک چندجمله‌ای $A(x)$ بـا کـران درجـه‌ی n در یـک نقطـه‌ی داده شـده‌ی x_0 را می‌توان با تقسیم $A(x)$ بر $(x - x_0)$ و یافتن یک چندجمله‌ای $q(x)$ با کران درجـه‌ی $n-1$ انجام داد، به طوری که

$$A(x) = q(x)(x - x_0) + r$$

بدیهتاً داریم $A(x) = r$. نشان دهید که چگونه می‌توان r و ضرایب $q(x)$ را در زمان $\theta(n)$ از x_0 و ضرایب A محاسبه کرد.

۳۰-۱-۳ از روی یک نمایش نقطه- مقدار برای $A(x) = \sum_{j=0}^{n-1} a_j x^j$، یک نمایش نقطه مقدار برای $A^{rev}(x) = \sum_{j=0}^{n-1} a_{n-1-j} x^j$ به دست آورید، با این فرض که هیچ کدام از ورودی‌ها ۰ نیستند.

۳۰-۱-۴ اثبات کنید که برای توصیف یک چندجمله‌ای با کران درجه‌ی n به صورت یکتا، به n جفت نقطه-مقدار مجزا نیاز داریم، یعنی اگر تعداد جفت‌های نقطه-مقدار داده شده کم‌تر از n باشد نمی‌توان با استفاده از آن‌ها یک چندجمله‌ای یکتا با کران درجه‌ی n توصیف کرد. (راهنمایی: با استفاده از قضیه‌ی ۳۰-۱، در مورد مجموعه‌ای از $n-1$ جفت نقطه-مقدار که یک جفت نقطه-مقدار دلخواه دیگر به آن اضافه شده است، چه می‌توانید بگویید؟)

۳۰-۱-۵ نشان دهید چگونه می‌توان از تساوی (۳۰-۵) برای درون‌یابی در زمان $\theta(n^2)$ استفاده کرد؟ (راهنمایی: ابتدا نمایش ضرایب چندجمله‌ای $\prod_j (x - x_j)$ را محاسبه، و سپس در صورت لزوم صورت هر عبارت آن را بر $(x - x_k)$ تقسیم کنید؛ تمرین ۳۰-۱-۲ را ببینید. هر یک از n مخرج را می‌توان در زمان $O(n)$ محاسبه کرد.)

۳۰-۱-۶ توضیح دهید که مشکل روش «بدیهی» برای تقسیم چندجمله‌ای‌ها با استفاده از نمایش نقطه- مقدار چیست؟ (منظور از روش بدیهی، تقسیم مقادیر y متناظر بر یکدیگر است.) به صورت جداگانه حالتی که تقسیم در آن به درستی انجام می‌شود و حالتی که در آن تقسیم به درستی انجام نمی‌شود را بررسی کنید.

۳۰-۱-۷ دو مجموعه‌ی A و B را در نظر بگیرید، که هر یک شامل n عدد صحیح در بازه‌ی ۰ تا $10n$ هستند. می‌خواهیم *جمع کارتزین* A و B را محاسبه کنیم، که به صورت زیر تعریف می‌شود:

$$C = \{x + y : x \in A \text{ و } y \in B\}$$

توجه کنید که اعداد C در بازه‌ی ۰ تا $20n$ هستند. می‌خواهیم عناصر C و تعداد حالاتی را که هر یک از عناصر C را می‌توان به صورت مجموع عناصر در A و B در نظر گرفت، بیابیم. نشان دهید که می‌توان این مسئله را در زمان $\theta(n \lg n)$ حل کرد. (راهنمایی: A و B را به صورت چندجمله‌ای‌هایی با درجه‌ی حداکثر $10n$ نمایش دهید.)

۳۰-۲ DFT و FFT

در بخش ۳۰-۱ ادعا کردیم که اگر از ریشه‌های مختلط واحد استفاده کنیم، می‌توانیم در زمان $\theta(n \lg n)$ چندجمله‌ای‌ها را ارزیابی و درون‌یابی کنیم. در این بخش ریشه‌های مختلط واحد را تعریف کرده و خصوصیات مختلف آن‌ها را بررسی می‌کنیم، همچنین DFT را تعریف کرده و سپس نشان می‌دهیم که چگونه FFT، DFT و معکوس آن را فقط در زمان $\theta(n \lg n)$ محاسبه می‌کند.

ریشه‌های مختلط واحد

یک *ریشه‌ی nام مختلط واحد* (complex nth root of unity)، یک عدد مختلط w است به طوری که

$$w^n = 1$$

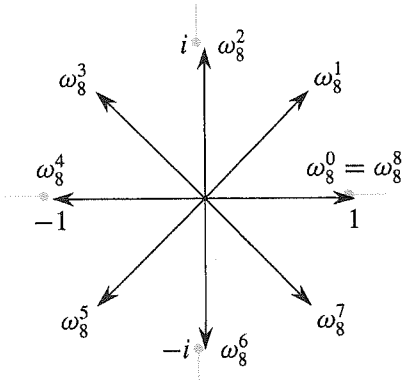
دقیقاً n ریشه‌ی nام مختلط واحد وجود دارد: $e^{2\pi ik/n}$ که در آن $k = 0, 1, \ldots, n-1$. برای تفسیر این فرمول از تعریف توان‌های یک عدد مختلط استفاده می‌کنیم:

$$e^{iu} = \cos(u) + i \sin(u)$$

شکل ۳۰-۲ نشان می‌دهد که n ریشه‌ی مختلط واحد با فاصله‌ی برابر دور دایره‌ی واحد به مرکز مبدأ مختصات صفحه‌ی مختلط پراکنده شده‌اند. مقدار

$$w_n = e^{2\pi i/n} \qquad (۶-۳۰)$$

ریشه‌ی اصلی nام واحد (the principal nth root of unity) نام دارد؛[1] تمام ریشه‌های nام مختلط واحد دیگر، توان‌هایی از w_n هستند.



شکل ۳۰-۲ مقادیر $w_8^0, w_8^1, \ldots, w_8^7$ در صفحه‌ی مختلط، که در آن $w_8 = e^{2\pi i/8}$ ریشه‌ی ۸ام اصلی واحد است.

[1] بسیاری از نویسنده‌های دیگر w_n را به صورتی دیگر تعریف می‌کنند: $w_n = e^{-2\pi i/n}$. این تعریف متفاوت برای کاربردهای پردازش سیگنال استفاده می‌شود. در هر صورت برای هر دو تعریف w_n، ریاضیات مورد استفاده به طور عمده یکسان است.

n ریشه‌ی nام مختلط واحد،

$$w_n^0, w_n^1, \ldots, w_n^{n-1}$$

یک گروه تحت ضرب تشکیل می‌دهند (بخش ۳۱-۳ را ببینید). ساختار این گروه مشابه ساختار گروه جمعی $(\mathbb{Z}_n, +)$ به پیمانه‌ی n است، چرا که $w_n^n = w_n^0 = 1$ نتیجه می‌دهد $w_n^j w_n^k = w_n^{j+k} = w_n^{(j+k) \bmod n}$. به طور مشابه، $w_n^{-1} = w_n^{n-1}$. خصوصیات مهم ریشه‌های nام مختلط در لم‌های زیر آمده‌اند.

لم ۳۰-۶ (لم انحذاف)

برای هر سه عدد صحیح $n \geq 0$، $k \geq 0$، و $d > 0$، داریم

$$w_{dn}^{dk} = w_n^k \tag{۳۰-۷}$$

اثبات این لم مستقیماً از تساوی (۳۰-۶) نتیجه می‌شود، چرا که

$$\begin{aligned} w_{dn}^{dk} &= (e^{2\pi i/dn})^{dk} \\ &= (e^{2\pi i/n})^k \\ &= w_n^k \end{aligned}$$

■

نتیجه‌ی ۳۰-۴

برای هر عدد صحیح زوج $n > 0$،

$$w_n^{n/2} = w_2 = -1$$

اثبات این اثبات به عنوان تمرین ۳۰-۲-۱ به خواننده واگذار شده است. ■

لم ۳۰-۵ (لم اتصال)

اگر $n > 0$ زوج باشد، آن گاه مربعات n ریشه‌ی nام مختلط واحد، $n/2$ ریشه‌ی $(n/2)$ام مختلط هستند.

اثبات طبق لم اتصال برای هر عدد صحیح نامنفی k داریم $(w_n^k)^2 = w_{n/2}^k$. توجه کنید که اگر از تمام n ریشه‌ی nام مختلط واحد مربع گیری کنیم، در این صورت هر ریشه‌ی $(n/2)$ام واحد دقیقاً دو بار تولید می‌شود، چرا که

$$\begin{aligned} (w_n^{k+n/2})^2 &= w_n^{2k+n} \\ &= w_n^{2k} w_n^n \\ &= w_n^{2k} \\ &= (w_n^k)^2 \end{aligned}$$

بنابراین مربع w_n^k و $w_n^{k+n/2}$ یکسان است. این خصوصیت را می‌توان به کمک نتیجه‌ی ۳۰-۴ هم اثبات کرد، چرا که $w_n^{n/2} = -1$ نتیجه می‌دهد $w_n^{k+n/2} = -w_n^k$، و بنابراین $(w_n^{k+n/2})^2 = (w_n^k)^2$. ■

همان طور که خواهیم دید، لم تصنیف برای رویکرد تقسیم و حل ما برای تبدیل میان نمایش‌های ضرایب و نقطه- مقدار چندجمله‌ای‌ها ضروری است، چرا که تضمین می‌کند اندازه‌ی زیرمسئله‌های بازگشتی در هر مرحله نصف می‌شود.

لم ۳۰-۶ (لم حذف)

برای هر عدد صحیح $n \geq 1$ و عدد صحیح نامنفی k که به n تقسیم‌پذیر نیست، داریم

$$\sum_{j=0}^{n-1} (w_n^k)^j = 0$$

اثبات تساوی (الف-۵) علاوه بر اعداد حقیقی، برای اعداد مختلط هم کاربرد دارد، و بنابراین داریم

$$\begin{aligned}\sum_{j=0}^{n-1} (w_n^k)^j &= \frac{(w_n^k)^n - 1}{w_n^k - 1} \\ &= \frac{(w_n^n)^k - 1}{w_n^k - 1} \\ &= \frac{(1)^k - 1}{w_n^k - 1} \\ &= 0\end{aligned}$$

تقسیم‌پذیر نبودن k بر n تضمین می‌کند که مخرج کسر 0 نیست، چرا که $w_n^k = 1$ فقط زمانی برقرار است که k بر n تقسیم‌پذیر باشد.

DFT

به خاطر بیاورید که هدف ما، ارزیابی یک چندجمله‌ای

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

با کران درجه‌ی n در نقاط $w_n^0, w_n^1, w_n^2, ..., w_n^{n-1}$ است (یعنی در n ریشه‌ی nام مختلط واحد).[1] فرض می‌کنیم A به صورت نمایش ضرایب به ما داده شده است: $a = (a_0, a_1, ..., a_{n-1})$. اجازه دهید برای $k = 0, 1, ..., n-1$ نتایج y_k را به صورت زیر تعریف کنیم:

$$\begin{aligned}y_k &= A(w_n^k) \\ &= \sum_{j=0}^{n-1} a_j w_n^{kj}\end{aligned} \qquad (۸-۳۰)$$

بردار $y = (y_0, y_1, ..., y_{n-1})$، *تبدیل فوریه‌ی گسسته‌ی* (Discrete Fourier Transform, DFT) بردار ضرایب $a = (a_0, a_1, ..., a_{n-1})$ است. همچنین می‌نویسیم $y = \mathrm{DFT}_n(a)$.

[1] طول n در واقع چیزی است که در بخش ۳۰-۱ آن را $2n$ نامیدیم، چرا که قبل از ارزیابی چندجمله‌ای‌ها، کران درجه‌ی آن‌ها را دو برابر می‌کنیم. بنابراین در مفهوم ضرب چندجمله‌ای‌ها، در واقع با ریشه‌ی $(2n)$ام مختلط واحد کار می‌کنیم.

FFT

با استفاده از یک متد با نام *تبدیل سریع فوریه* (Fast Fourier Transform, FFT)، که از خصوصیات ویژه‌ی ریشه‌های مختلط واحد استفاده می‌کند، می‌توانیم $\mathrm{DFT}_n(a)$ را در زمان $\theta(n \lg n)$ محاسبه کنیم، که نسبت به زمان $\theta(n^2)$ برای متد سرراست بهبود خوبی است. فرض می‌کنیم n توانی از ۲ است. با این که استراتژی‌هایی برای کار با nهایی که توانی از ۲ نیستند هم وجود دارد، در این جا به آن‌ها نمی‌پردازیم.

متد FFT یک استراتژی تقسیم و حل به کار می‌برد، و با استفاده از ضرایب با اندیس زوج و فرد $A(x)$ به صورت جداگانه، دو چندجمله‌ای $A^{[0]}(x)$ و $A^{[1]}(x)$ با کران درجه‌ی $n/2$ تعریف می‌کند:

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{n/2-1},$$
$$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{n/2-1}$$

توجه کنید که $A^{[0]}$حاوی تمام ضرایب با اندیس زوج A است (نمایش دودویی این ضرایب با ۰ پایان می‌یابد) و $A^{[1]}$ حاوی تمام ضرایب با اندیس فرد A است (نمایش دودویی این ضرایب با ۱ پایان می‌یابد). داریم

$$A(x) = A^{[0]}(x^2) + A^{[1]}(x^2) \qquad (۹-۳۰)$$

و مسئله‌ی ارزیابی $A(x)$ در نقاط $w_n^0, w_n^1, \ldots, w_n^{n-1}$ کاهش می‌یابد به

۱. ارزیابی چندجمله‌ای‌های $A^{[0]}(x)$ و $A^{[1]}(x)$ با کران درجه‌ی $n/2$ در نقاط

$$(w_n^0)^2, (w_n^1)^2, \ldots, (w_n^{n-1})^2 \qquad (۱۰-۳۰)$$

و سپس

۲. ترکیب نتایج طبق تساوی (۳۰-۹).

طبق لم تصنیف، لیست مقادیر (۳۰-۱۰) از n مقدار متمایز تشکیل نشده است، بلکه حاوی $n/2$ ریشه‌ی $(n/2)$ام مختلط واحد است، که در آن هر ریشه دقیقاً دو بار ظاهر می‌شود. بنابراین چندجمله‌ای‌های $A^{[0]}(x)$ و $A^{[1]}(x)$ با کران درجه‌ی $n/2$ به صورت بازگشتی در $n/2$ ریشه‌ی $(n/2)$ام مختلط واحد ارزیابی می‌شوند. شکل این زیرمسئله دقیقاً مانند مسئله‌ی اصلی است، با نصف اندازه‌ی مسئله‌ی اصلی. اکنون محاسبه‌ی یک DFT_n با n عنصر را با موفقیت به محاسبه‌ی دو $\mathrm{DFT}_{n/2}$ با $n/2$ عنصر تبدیل کرده‌ایم. این تجزیه پایه‌ی الگوریتم DFT زیر است که یک بردار n عنصری $a = (a_0, a_1, \ldots, a_{n-1})$ را محاسبه می‌کند، که در آن n توانی از ۲ است.

```
RECURSIVE-FFT(a)
1  n = a.length          // n is a power of 2.
2  if n == 1
3      return a
4  w_n = e^{2πi/n}
5  w = 1
6  a^[0] = (a_0, a_2, ..., a_{n-2})
```

```
7   a^[1] = (a_1, a_3, ..., a_{n-1})
8   y^[0] = RECURSIVE-FFT(a^[0])
9   y^[1] = RECURSIVE-FFT(a^[1])
10  for k = 0 to n/2 - 1
11      y_k = y_k^[0] + w y_k^[1]
12      y_{k+(n/2)} = y_k^[0] - w y_k^[1]
13      w = w w_n
14  return y                    // y is assumed to be a column vector.
```

رویه‌ی RECURSIVE-FFT به صورت زیر کار می‌کند. خطوط ۲-۳ پایه‌ی بازگشت هستند؛ DFT یک عنصر، خود آن عنصر است، چرا که در این حالت

$$\begin{aligned} y_0 &= a_0 w_1^0 \\ &= a_0 \cdot 1 \\ &= a_0 \end{aligned}$$

خطوط ۶-۷، بردارهای ضرایب را برای $A^{[0]}$ و $A^{[1]}$ تعریف می‌کنند. خطوط ۴، ۵، و ۱۳ تضمین می‌کنند که w به درستی به هنگام سازی می‌شود به طوری که هر گاه خطوط ۱۱-۱۲ اجرا می‌شوند، داریم $w = w_n^k$. (با نگه داشتن یک مقدار به هنگام برای w از یک تکرار به تکرار دیگر، به جای محاسبه‌ی w_n^k برای هر تکرار حلقه‌ی for در زمان صرفه‌جویی می‌شود.) خطوط ۸-۹ محاسبات بازگشتی $\mathrm{DFT}_{n/2}$ را انجام می‌دهند، بدین صورت که برای $k = 0, 1, \ldots, n/2 - 1$ قرار می‌دهند

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(w_{n/2}^k), \\ y_k^{[1]} &= A^{[1]}(w_{n/2}^k) \end{aligned}$$

یا، چون طبق لم اتصال داریم $w_{n/2}^k = w_n^{2k}$،

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(w_n^{2k}), \\ y_k^{[0]} &= A^{[0]}(w_n^{2k}) \end{aligned}$$

خطوط ۱۱-۱۲ نتایج محاسبات بازگشتی $\mathrm{DFT}_{n/2}$ را ترکیب می‌کنند. برای $y_0, y_1, \ldots, y_{n/2-1}$ خط ۱۱ نتیجه می‌دهد

$$\begin{aligned} y_k &= y_k^{[0]} + w_n^k y_k^{[1]} \\ &= A^{[0]}(w_n^{2k}) + w_n^k A^{[1]}(w_n^{2k}) \\ &= A(w_n^k) \qquad \text{(طبق تساوی (۳۰-۹))} \end{aligned}$$

برای $y_{n/2}, y_{n/2+1}, \ldots, y_{n-1}$، با فرض $k = 0, 1, \ldots, n/2 - 1$، خط ۱۲ نتیجه می‌دهد

$$y_{k+(n/2)} = y_k^{[0]} - w_n^k y_k^{[1]}$$

$$
\begin{aligned}
&= y_k^{[0]} + w_n^{k+(n/2)} y_k^{[1]} && (\text{چرا که } w_n^{k+(n/2)} = -w_n^k)\\
&= A^{[0]}(w_n^{2k}) + w_n^{k+(n/2)} A^{[1]}(w_n^{2k}) \\
&= A^{[0]}(w_n^{2k+n}) + w_n^{k+(n/2)} A^{[1]}(w_n^{2k+n}) && (\text{چرا که } w_n^{2k+n} = w_n^{2k})\\
&= A(w_n^{k+n(n/2)}) && (\text{طبق تساوی (۳۰-۹)})
\end{aligned}
$$

بنابراین بردار y که توسط RECURSIVE-FFT بازگردانده شده است، همان DFT بردار ورودی a است.

خطوط ۱۱ و ۱۲ هر مقدار $y_k^{[1]}$ را در w_n^k ضرب می‌کنند ($k = 0, 1, \ldots, n/2-1$). خط ۱۱ این حاصل‌ضرب را به $y_k^{[0]}$ اضافه می‌کند، و خط ۱۲ آن را کم می‌کند. چون هر بردار w_n^k به هر دو صورت مثبت و منفی آن مورد استفاده قرار می‌گیرد، فاکتورهای w_n^k به *فاکتورهای گردشی* (twiddle factor) معروف‌اند.

برای تعیین زمان اجرای رویه‌ی RECURSIVE-FFT، توجه می‌کنیم که جدا از فراخوانی‌های بازگشتی، هر فراخوانی به $\theta(n)$ زمان نیاز دارد، که در آن n طول بردار ورودی است. بنابراین رابطه‌ی بازگشتی برای زمان اجرا عبارت است از

$$
\begin{aligned}
T(n) &= 2T(n/2) + \theta(n) \\
&= \theta(n \lg n)
\end{aligned}
$$

پس می‌توانیم با استفاده از تبدیل سریع فوریه، یک چندجمله‌ای با کران درجه‌ی n را در ریشه‌های nام مختلط واحد در زمان $\theta(n \lg n)$ تعیین کنیم.

درون‌یابی در ریشه‌های مختلط واحد

اکنون با نشان دادن این که چگونه می‌توان ریشه‌های مختلط واحد را توسط یک چندجمله‌ای درون‌یابی کرد (که ما را قادر می‌سازد نمایش نقطه-مقدار را دوباره به نمایش ضرایب تبدیل کنیم)، متد ضرب چندجمله‌ای‌ها را کامل می‌کنیم. درون‌یابی را با نوشتن DFT به صورت یک تساوی ماتریسی و سپس توجه به شکل معکوس ماتریس انجام می‌دهیم.

از تساوی (۳۰-۴)، می‌توانیم DFT را به صورت ضرب ماتریسی $y = V_n a$ بنویسیم، که در آن V_n یک ماتریس واندرموند (Vandermonde matrix) است، حاوی توان‌های مناسب از w_n:

$$
\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}
=
\begin{pmatrix}
1 & 1 & 1 & 1 & \cdots & 1 \\
1 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\
1 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\
1 & w_n^3 & w_n^6 & w_n^9 & \cdots & w_n^{3(n-1)} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
1 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)}
\end{pmatrix}
\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}
$$

ورودی (k, j) در ماتریس V_n عبارت است از w_n^{kj}، که $j, k = 0, 1, \ldots, n-1$، و توان‌های ورودی‌های V_n یک جدول ضرب را تشکیل می‌دهند.

برای عملیات معکوس‌گیری، که آن را به صورت $a = \mathrm{DFT}_n^{-1}(y)$ می‌نویسیم، y را در ماتریس V_n^{-1} (معکوس V_n) ضرب می‌کنیم.

قضیه‌ی ۳۰-۷ برای $j,k = 0, 1, \ldots, n-1$، ورودی (j,k) در ماتریس V_n^{-1} عبارت است از w_n^{-kj}/n.

اثبات نشان می‌دهیم که $V_n^{-1}V_n = I_n$، که I_n ماتریس همانی است. ورودی (j,j') ماتریس $V_n^{-1}V_n$ را در نظر بگیرید:

$$\begin{aligned}[V_n^{-1}V_n]_{jj'} &= \sum_{k=0}^{n-1} (w_n^{-kj}/n)(w_n^{kj'}) \\ &= \sum_{k=0}^{n-1} w_n^{k(j'-j)}/n\end{aligned}$$

اگر $j = j'$، این مجموع برابر ۱ می‌شود، و در غیر این صورت، طبق لم جمع (لم ۳۰-۶) برابر ۰ خواهد شد. توجه کنید که باید داشته باشیم $-(n-1) < j' - j < n-1$ تا $j'-j$ بر n بخش‌پذیر نباشد، تا بتوانیم لم جمع را به کار ببریم. ▒

اگر ماتریس معکوس V_n^{-1} را داشته باشیم، می‌دانیم که $\mathrm{DFT}_n^{-1}(y)$ طبق رابطه‌ی زیر به دست می‌آید:

$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k w_n^{-kj} \qquad (۳۰-۱۱)$$

برای $j = 0, 1, \ldots, n-1$. با مقایسه‌ی تساوی‌های (۳۰-۸) و (۳۰-۱۱) می‌بینیم که با اصلاح الگوریتم FFT و تعویض نقش a و y، جایگزینی w_n با w_n^{-1}، و تقسیم هر عنصر در نتیجه بر n، می‌توانیم معکوس DFT را محاسبه کنیم (تمرین ۳۰-۲-۴ را ببینید). پس می‌توان DFT_n^{-1} را هم در زمان $\theta(n \lg n)$ محاسبه کرد.

بنابراین با استفاده از FFT و معکوس آن، می‌توانیم در زمان $\theta(n \lg n)$ نمایش‌های نقطه-مقدار و ضرایب یک چندجمله‌ای با کران درجه‌ی n را به یکدیگر تبدیل کنیم. با بازگشت به مفهوم ضرب چندجمله‌ای‌ها، قضیه‌ی زیر را اثبات کرده‌ایم.

قضیه‌ی ۳۰-۸ (قضیه‌ی کانولوشن) برای هر دو بردار a و b با طول n، که در آن n توانی از ۲ است، داریم

$$a \otimes b = \mathrm{DFT}_{2n}^{-1}(\mathrm{DFT}_{2n}(a) \cdot \mathrm{DFT}_{2n}(b))$$

که در آن a و b با ۰ پر شده تا به اندازه‌ی $2n$ برسند، و $\cdot$ نشان‌دهنده‌ی ضرب عنصر به عنصر دو بردار $2n$ تایی است.

▒

تمرین‌ها

۳۰-۲-۱ نتیجه‌ی ۳۰-۴ را اثبات کنید.

۳۰-۲-۲ DFT بردار $(0, 1, 2, 3)$ را محاسبه کنید.

۳۰-۲-۳ تمرین ۳۰-۱-۱ را با استفاده از رویکرد داده شده با زمان $\theta(n \lg n)$ انجام دهید.

۳۰-۲-۴ شبه‌کدی برای محاسبه‌ی DFT_n^{-1} در زمان $\theta(n \lg n)$ ارائه کنید.

۳۰-۲-۵ گسترش رویه‌ی FFT برای حالتی که n توانی از ۳ است را توضیح دهید. یک رابطه‌ی بازگشتی برای زمان اجرا ارائه کرده و آن را حل کنید.

۳۰-۲-۶ ★ فرض کنید به جای انجام یک FFT با n عنصر در حوزه‌ی اعداد مختلط (که در آن n زوج است)، از حلقه‌ی $\mathbb{Z}_m$ از اعداد به پیمانه‌ی m استفاده می‌کنیم، که در آن $m = 2^{2n/2+1}$ و t یک عدد صحیح مثبت دلخواه است. از $w = 2^t$ به پیمانه‌ی m به جای ریشه‌ی nام اصلی واحد (w_n) استفاده کنید. اثبات کنید که DFT و معکوس DFT در این سیستم خوش‌تعریف هستند.

۳۰-۲-۷ با داشتن لیستی از مقادیر $z_0, z_1, \ldots, z_{n-1}$ (با امکان تکرار)، نشان دهید که چگونه می‌توان ضرایب یک چندجمله‌ای $P(x)$ با کران درجه‌ی $n+1$ را یافت که فقط در نقاط $z_0, z_1, \ldots, z_{n-1}$ صفر می‌شود (با امکان تکرار). رویه‌ی شما باید در زمان $\theta(n \lg^2 n)$ اجرا شود. (*راهنمایی:* چندجمله‌ای $P(x)$ در z_j یک صفر می‌شود اگر و فقط اگر $P(x)$ ضریبی از $(x - z_j)$ باشد.)

۳۰-۲-۸ ★ ***تبدیل chirp*** یک بردار $a = (a_0, a_1, \ldots, a_{n-1})$، بردار $y = (y_0, y_1, \ldots, y_{n-1})$ است، که در آن $y_k = \sum_{j=0}^{n-1} a_j z^{kj}$ و z یک عدد مختلط دلخواه است. بنابراین DFT حالت خاصی است از تبدیل chirp که با قرار دادن $z = w_n$ به دست می‌آید. اثبات کنید که برای هر عدد مختلط z می‌توان تبدیل chirp را در زمان $O(n \lg n)$ ارزیابی کرد. (*راهنمایی:* از تساوی

$$y_k = z^{k^2/2} \sum_{j=0}^{n-1} (a_j z^{j^2/2})(z^{-(k-j)^2/2})$$

استفاده کرده و تبدیل chirp را به صورت یک کانولوشن ببینید.)

۳۰-۳ پیاده‌سازی‌های بهینه‌ی FFT

از آن جایی که کاربردهای عملی FFT مانند پردازش سیگنال به بالاترین سرعت ممکن نیاز دارند، در این بخش دو پیاده‌سازی بهینه‌ی FFT را بررسی می‌کنیم. ابتدا نسخه‌ی تکراری FFT را خواهیم دید که در همان زمان $\theta(n \lg n)$ اجرا می‌شود، ولی ضرایب ثابت مخفی در θ برای آن از نسخه‌ی بازگشتی داده شده در بخش ۳۰-۲ کوچک‌تر است. (بسته به نحوه‌ی پیاده‌سازی، نسخه‌ی بازگشتی ممکن است از کَش سخت‌افزار بهینه‌تر استفاده کند.) سپس از شهودی که ما را به پیاده‌سازی تکراری FFT رسانده است استفاده می‌کنیم تا یک مدار موازی برای FFT طراحی کنیم.

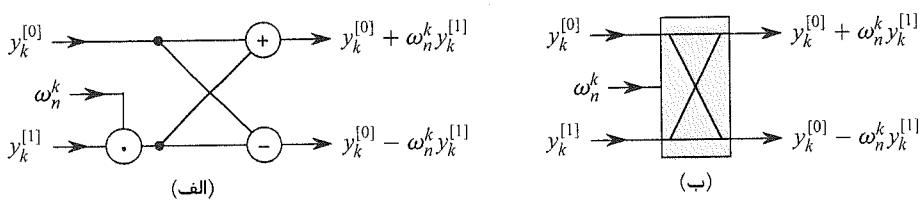
یک پیاده‌سازی بازگشتی برای FFT

ابتدا توجه می‌کنیم که حلقه‌ی for خطوط ۱۰-۱۳ رویه‌ی RECURSIVE-FFT شامل دو بار محاسبه‌ی مقدار $w_n^k y_k^{[1]}$ می‌شود. در ادبیات کامپایلر، این مقدار یک *زیرعبارت مشترک* (common subexpression) نام دارد. می‌توان حلقه را طوری تغییر داد که این مقدار را فقط یک بار محاسبه، و آن را در متغیر کمکی t ذخیره کند.

```
for k = 0 to n/2 − 1
    t = w y_k^[1]
    y_k = y_k^[0] + t
    y_{k+(n/2)} = y_k^[0] − t
    w_n = w w_n
```

عملیات این حلقه، یعنی ضرب فاکتور زائد $w = w_n^k$ در $y_k^{[1]}$، ذخیره‌ی حاصل ضرب در t، و اضافه و کم کردن t از $y_k^{[0]}$، به *عملیات پروانه* (butterfly operation) مشهور است، که در شکل ۳۰-۳ نشان داده شده است.

اکنون نشان می‌دهیم که چگونه می‌توان الگوریتم FFT را به جای بازگشتی به صورت تکراری نوشت. در شکل ۳۰-۴، بردارهای ورودی برای فراخوانی‌های بازگشتی در یک فراخوانی



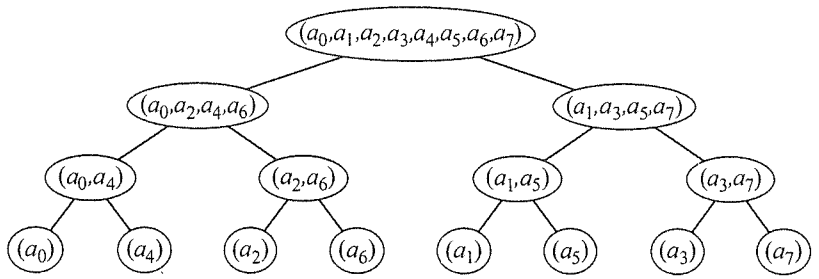
شکل ۳۰-۳ یک عملیات پروانه. (الف) دو مقدار ورودی از سمت چپ وارد می‌شوند، فاکتور زائد w_n^k در $y_k^{[1]}$ ضرب می‌شود، و مجموع و تفاضل، خروجی‌های سمت راست هستند. (ب) یک شکل ساده شده از عملیات پروانه. در مدار موازی FFT از این نمایش استفاده خواهیم کرد.

RECURSIVE-FFT را در یک ساختار درختی مرتب کرده‌ایم، که در آن فراخوانی اولیه برای $n = ۸$ است. این درخت یک گره برای هر فراخوانی رویه دارد، که با بردار ورودی مربوطه علامت‌گذاری شده است. هر فراخوانی RECURSIVE-FFT دو فراخوانی دیگر انجام می‌دهد، مگر این که ورودی آن یک بردار تک عنصری باشد. اولین فراخوانی را بر روی فراخوانی فرزند سمت چپ و دومین فراخوانی را بر روی سمت راست انجام می‌دهیم.

با دقت در درخت، مشاهده می‌کنیم که اگر بتوانیم عناصر بردار اولیه‌ی a را به صورتی مرتب کنیم که در برگ‌های درخت قرار دارند، می‌توانیم به صورت زیر اجرای رویه‌ی RECURSIVE-FFT را دنبال کنیم، البته به صورت پایین به بالا، به جای بالا به پایین. ابتدا عناصر را به صورت جفت جفت در نظر می‌گیریم، DFT هر جفت را با استفاده از عملیات پروانه محاسبه، و آن جفت را با DFT مربوطه جایگزین می‌کنیم. در این صورت بردار حاوی $n/۲$ جفت DFT خواهد بود. سپس دوباره این $n/۲$ جفت DFT را به صورت جفت جفت در نظر می‌گیریم و DFTهای بردارهای چهار عنصری را با استفاده از دو عملیات پروانه محاسبه، و و هر دو DFT دوتایی را با یک DFT چهارتایی جایگزین می‌کنیم. اکنون بردار حاوی $n/۴$ DFT چهاتاریی خواهد بود. این کار را ادامه می‌دهیم تا بردار فقط حاوی دو DFT $n/۲$تایی شود، که می‌توانیم این دو را با $n/۲$ عملیات پروانه در یکدیگر ادغام کرده و DFT نهایی nتایی را بسازیم.

برای تبدیل این عملیات به کد، از یک آرایه‌ی $A[۰ .. n-۱]$ استفاده می‌کنیم که در ابتدا حاوی عناصر بردار ورودی a است، به ترتیبی که در برگ‌های درخت شکل ۳۰-۴ ظاهر می‌شوند. (بعداً نشان خواهیم داد که چگونه می‌توان این ترتیب را تشخیص داد، که جایگشت معکوس بیتی نام دارد.) چون ترکیب باید در هر سطح درخت انجام شود، یک متغیر s برای شمارش سطح‌ها تعریف می‌کنیم که حوزه‌ی تغییرات آن از ۱ (پایین‌ترین سطح، جایی که جفت‌ها را ترکیب می‌کنیم تا DFTهای ۲ عنصری را بسازیم) تا $\lg n$ (در بالاترین سطح، جایی که دو DFT با $n/۲$عنصر را با هم ترکیب می‌کنیم تا نتیجه‌ی نهایی را بسازیم) است. بنابراین ساختار الگوریتم به صورت زیر است:

```
1  for s = 1 to lg n
2      for k = 0 to n − 1 by 2^s
```



شکل ۳۰-۴ درخت بردارهای ورودی برای فراخوانی‌های بازگشتی رویه‌ی RECURSIVE-FFT. اولین فراخوانی برای $n = ۸$ است.

```
3        combine the two 2^(s-1)-element DFT's in
           A[k .. k + 2^(s-1) - 1] and A[k + 2^(s-1) .. k + 2^s - 1]
           into one 2^s -element DFT in A[k .. k + 2^s - 1]
```

می‌توانیم بدنه‌ی حلقه (خط ۳) را به صورت شبه‌کدی دقیق‌تر توصیف کنیم. حلقه‌ی for را از رویه‌ی RECURSIVE-FFT کپی می‌کنیم، که در آن $y^{[0]}$ معادل است با $A[k \,..\, k+2^{s-1}-1]$ و $y^{[1]}$ معادل است با $A[k+2^{s-1} \,..\, k+2^s-1]$. فاکتور زائد استفاده شده در عملیات پروانه به مقدار s بستگی دارد؛ مقدار آن توانی از w_m است، که در آن $m = 2^s$. (متغیر m را فقط برای خوانایی بیشتر تعریف می‌کنیم.) یک متغیر کمکی دیگر u تعریف می‌کنیم که به ما اجازه می‌دهد عملیات پروانه را در جا انجام دهیم. وقتی خط ۳ را در ساختار کلی با بدنه‌ی حلقه جایگزین می‌کنیم، به شبه‌کد زیر می‌رسیم، که مبنای پیاده‌سازی موازی را تشکیل می‌دهد که بعداً آن را ارائه خواهیم کرد. کد زیر ابتدا رویه‌ی کمکی BIT-REVERSE-COPY (a, A) را فراخوانی می‌کند تا مقادیر بردار a به ترتیب مورد نیاز در A کپی شوند.

```
ITERATIVE-FFT (a)
1   BIT-REVERSE-COPY (a, A)
2   n = a.length                    // n is a power of 2
3   for s = 1 to lg n
4       m = 2^s
5       ω_m = e^(2πi/m)
6       for k = 0 to n - 1 by m
7           ω = 1
8               for j = 0 to m/2 - 1
9                   t = ωA[k + j + m/2]
10                  u = A[k + j]
11                  A[k + j] = u + t
12                  A[k + j + m/2] = u - t
13                  ω = ω ω_m
14              return A
```

رویه‌ی BIT-REVERSE-COPY چگونه عناصر بردار ورودی a را به ترتیب مورد نظر در A کپی می‌کند؟ ترتیب ظاهر شدن برگ‌ها در شکل ۳۰-۴ یک جایگشت معکوس بیتی (bit-reversal permutation) است. یعنی اگر فرض کنیم $\text{rev}(k)$ یک عدد صحیح $\lg n$ بیتی باشد که از معکوس کردن بیت‌های نمایش دودویی k به دست می‌آید، در این صورت چیزی که ما نیاز داریم این است که عنصر a_k را در مکان $A[\text{rev}(k)]$ قرار دهیم. به عنوان مثال، در شکل ۳۰-۴ برگ‌ها به ترتیب ۰، ۴، ۲، ۶، ۱، ۵، ۳، ۷ ظاهر می‌شوند؛ شکل دودویی این دنباله به صورت ۰۰۰، ۱۰۰، ۰۱۰، ۱۱۰، ۰۰۱، ۱۰۱، ۰۱۱است، که وقتی بیت‌های آن را معکوس کنیم، به این دنباله می‌رسیم: ۰۰۰، ۰۰۱، ۰۱۰، ۰۱۱، ۱۰۰، ۱۰۱، ۱۱۰، ۱۱۱. برای این که ببینیم این خاصیت در حالت کلی هم برقرار است، توجه می‌کنیم که در بالاترین سطح درخت، اندیس‌هایی که کم‌ارزش‌ترین بیت آن‌ها صفر است در زیردرخت سمت چپ قرار می‌گیرند، و آن‌هایی که کم‌ارزش‌ترین بیت آن‌ها یک است در زیردرخت

سمت راست. با جدا کردن کم‌ارزش‌ترین بیت در هر سطح، همین فرایند در سطوح پایین‌تر درخت هم انجام می‌شود تا جایی که به ترتیب داده شده توسط جایگشت معکوس بیتی در برگ‌ها می‌رسیم.

از آن جایی که محاسبه‌ی تابع $\text{rev}(k)$ بسیار ساده است، می‌توان رویه‌ی BIT-REVERSAL-COPY را به صورت زیر نوشت.

```
BIT-REVERSE-COPY(a, A)
1  n = a.length
2  for k = 0 to n − 1
3      A[rev(k)] = a_k
```

پیاده‌سازی تکراری FFT در زمان $\theta(n \lg n)$ اجرا می‌شود. فراخوانی BIT-REVERSAL-COPY مشخصا به زمان $O(n \lg n)$ نیاز دارد، چرا که n تکرار داریم و می‌توانیم یک عدد بین ۰ و $n-1$ را، که $\lg n$ بیت دارد، در زمان $O(\lg n)$ معکوس کنیم. (در عمل معمولاً مقادیر اولیه‌ی n را از قبل می‌دانیم، و بنابراین احتمالاً یک جدول تهیه می‌کنیم که مقادیر k را به $\text{rev}(k)$ نگاشت می‌کند، و باعث می‌شود که BIT-REVERSAL-COPY در زمان $\theta(n)$ و با یک ضریب ثابت مخفی کوچک اجرا شود. همچنین می‌توانیم از روش شمارنده‌ی دودویی معکوس سرشکن هوشمندانه‌ای که در مسئله‌ی ۱۷-۱ توصیف شد استفاده کنیم.) برای تکمیل اثبات این که ITERATIVE-FFT در زمان $\theta(n \lg n)$ اجرا می‌شود، نشان می‌دهیم که $L(n)$، تعداد دفعاتی که بدنه‌ی داخلی‌ترین حلقه (خطوط ۸-۱۳) اجرا می‌شود، $\theta(n \lg n)$ است. حلقه‌ی for خطوط ۶-۱۳ برای هر مقدار s به تعداد $n/m = n/2^s$ بار اجرا می‌شود، و داخلی‌ترین حلقه‌ی خطوط ۸-۱۳، $m/2 = 2^{s-1}$ بار. بنابراین،

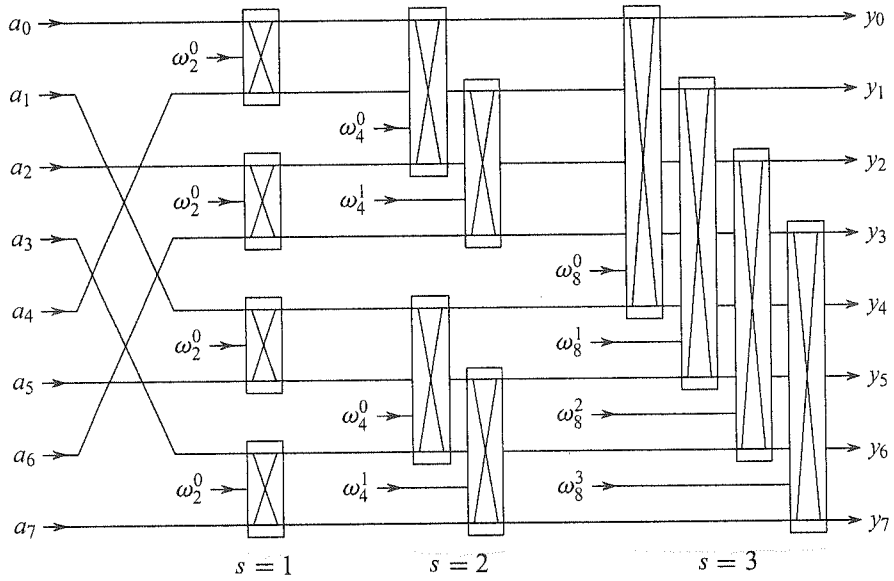
$$\begin{aligned} L(n) &= \sum_{s=1}^{\lg n} \frac{n}{2^s} \cdot 2^{s-1} \\ &= \sum_{s=1}^{\lg n} \frac{n}{2} \\ &= \theta(n \lg n) \end{aligned}$$

یک مدار FFT موازی

بسیاری از خصوصیاتی که در پیاده‌سازی الگوریتم FFT تکراری مفید بودند، می‌توانند در پیاده‌سازی یک الگوریتم FFT موازی هم مفید باشند. الگوریتم موازی FFT را به صورت یک مدار توصیف خواهیم کرد. شکل ۳۰-۵ یک مدار FFT موازی را نشان می‌دهد که FFT را روی n ورودی محاسبه می‌کند، برای $n = 8$. این مدار با یک جایگشت معکوس بیتی بر روی ورودی‌ها آغاز می‌شود، و به دنبال آن $\lg n$ مرحله شامل $n/2$ عملیات پروانه که به صورت موازی اجرا می‌شوند. بنابراین عمق مدار – بیشینه‌ی تعداد عناصر محاسباتی بین هر خروجی و هر ورودی مربوط به آن خروجی – $\theta(\lg n)$ است.

بخش چپی مدار PARALLE-FFT جایگشت معکوس بیتی را انجام می‌دهد، و باقی‌مانده‌ی مدار رویه‌ی ITERATIVE-FFT را شبیه‌سازی می‌کند. چون هر تکرار خارجی‌ترین حلقه‌ی for به تعداد $n/2$ عملیات پروانه‌ی مستقل انجام می‌دهد، این مدار آن‌ها را به صورت موازی پیاده‌سازی می‌کند. مقدار s

در هر تکرار در ITERATIVE-FFT متناظر است با یک مرحله از پروانه‌های نشان داده شده در شکل ۳۰-۵. در مرحله‌ی s، برای $s = ۱, ۲, ..., \lg n$، تعداد $n/۲^s$ گروه از پروانه‌ها وجود دارد (متناظر با هر یک از مقادیر k در ITERATIVE-FFT)، با $۲^{s-۱}$ پروانه در هر گروه (متناظر با هر مقدار j در ITERATIVE-FFT). پروانه‌های نشان داده شده در شکل ۳۰-۵ متناظرند با عملیات پروانه‌ی داخلی‌ترین حلقه (خطوط ۹-۱۲ در رویه‌ی ITERATIVE-FFT). همچنین توجه کنید که فاکتورهای زائد استفاده شده در پروانه‌ها متناظر با آن‌هایی هستند که در ITERATIVE-FFT از آن‌ها استفاده شده است: در مرحله‌ی s از $w_m^{۰}, w_m^{۱}, ..., w_m^{m/۲-۱}$ استفاده می‌کنیم، که در آن $m = ۲^s$.



شکل ۳۰-۵ یک مدار PARALLEL-FFT که FFT را محاسبه می‌کند، که در این جا برای $n = ۸$ ورودی نشان داده شده است. هر عملیات پروانه مقادیر دو سیم را به عنوان ورودی دریافت می‌کند، به همراه فاکتور زائد، و مقادیر دو سیم را به عنوان خروجی تولید می‌کند. مراحل پروانه طوری برچسب‌گذاری شده‌اند که تکرار متناظر آن‌ها در خارجی‌ترین حلقه‌ی رویه‌ی ITERATIVE-FFT مشخص باشد. فقط سیم‌های بالایی و پایینی گذرا از پروانه با آن ارتباط دارند؛ سیم‌هایی که از میانه‌ی پروانه عبور می‌کنند بر روی آن اثری نمی‌گذارند، و مقدار خود سیم‌ها هم توسط پروانه تغییری نمی‌کند. مثلاً پروانه‌ی بالایی در مرحله‌ی ۲ هیچ کاری با سیم ۱ (سیمی که ورودی آن با $y_۱$ علامت‌گذاری شده است) ندارد؛ ورودی و خروجی‌های آن بر روی سیم‌های ۰ و ۲ هستند (که این سیم‌ها با $y_۰$ و $y_۲$ علامت‌گذاری شده‌اند). عمق این مدار $\theta(\lg n)$ است، و مدار در مجموع $\theta(n \lg n)$ عملیات پروانه انجام می‌دهد.

تمرین‌ها

۳۰.۲-۱ نشان دهید که ITERATIVE-FFT چگونه مقدار DFT بردار ورودی $(0, 2, 3, -1, 4, 5, 7, 9)$ را محاسبه می‌کند.

۳۰.۲-۲ نشان دهید که چگونه می‌توان یک الگوریتم FFT پیاده‌سازی کرد که در آن جایگشت معکوس بیتی، به جای ابتدای محاسبه در انتهای محاسبه انجام می‌شود. (راهنمایی: معکوس DFT را در نظر بگیرید.)

۳۰.۲-۳ ITERATIVE-FFT چند بار فاکتورهای زائد را در هر مرحله محاسبه می‌کند؟ ITERATIVE-FFT را طوری بازنویسی کنید که فاکتورهای زائد را فقط 2^{s-1} بار در هر مرحله‌ی s محاسبه کند.

★ ۳۰.۲-۴ فرض کنید جمع‌کننده‌ها در اعمال پروانه‌ی مدار FFT بعضی مواقع اشتباه می‌کنند، به طوری که خروجی آن‌ها همیشه صفر است، مستقل از ورودی. فرض کنید دقیقاً یک جمع کننده این اشتباه را انجام می‌دهد، ولی نمی‌دانیم که کدام یک. توضیح دهید که چگونه می‌توان با دادن یک ورودی به کل مدار FFT و مشاهده‌ی خروجی، جمع‌کننده‌ی خراب را شناسایی کرد. متد شما چقدر کارا است؟

مسائل

۳۰-۱ ضرب به روش تقسیم و حل

i. نشان دهید که چگونه می‌توان حاصل ضرب دو چندجمله‌ای خطی $ax+b$ و $cx+b$ را با استفاده از فقط سه ضرب محاسبه کرد. (راهنمایی: یکی از ضرب‌ها $(a+b)\cdot(c+d)$ است.)

ii. دو الگوریتم تقسیم و حل برای ضرب دو چندجمله‌ای با کران درجه‌ی n بدهید که در زمان $\theta(n^{\lg 3})$ اجرا می‌شوند. الگوریتم اول باید ضرایب چندجمله‌ای ورودی را به دو قسمت نیمه‌ی بالایی و نیمه‌ی پایینی تقسیم کند، و الگوریتم دوم باید تقسیم ضرایب را بر حسب زوج یا فرد بودن اندیس آن‌ها انجام دهد.

iii. نشان دهید که دو عدد صحیح n بیتی را می‌توان در $O(n^{\lg 3})$ مرحله در هم ضرب کرد، که در آن هر مرحله حداکثر بر روی تعداد ثابتی مقدار ۱ بیتی عمل می‌کند.

۳۰-۲ ماتریس‌های Toeplitz

یک ماتریس Toeplitz یک ماتریس $A = (a_{ij})$ با اندازه‌ی $n \times n$ است به طوری که در آن برای $i = 2,3,\ldots,n$ و $j = 2,3,\ldots,n$ داریم $a_{ij} = a_{i-1,j-1}$.

i. آیا جمع دو ماتریس Toeplitz لزوماً یک ماتریس Toeplitz است؟ ضرب آن‌ها چطور؟

II. توضیح دهید که چطور می‌توان یک ماتریس Toeplitz را طوری توصیف کرد که جمع دو ماتریس Toeplitz با اندازه‌ی $n \times n$ در زمان $O(n)$ انجام شود.

III. یک الگوریتم از مرتبه‌ی $O(n \lg n)$ برای ضرب یک ماتریس Toeplitz با اندازه‌ی $n \times n$ در یک بردار به طول n ارائه کنید. از نمایش خود در بخش II استفاده کنید.

IV. یک الگوریتم کارامد برای ضرب دو ماتریس Toeplitz با اندازه‌ی $n \times n$ ارائه و زمان اجرای آن را تحلیل کنید.

۳۰-۳ تبدیل سریع فوریه‌ی چند بعدی

می‌توان تبدیل فوریه‌ی گسسته‌ی یک بعدی را که در تساوی (۳۰-۸) تعریف شده است، به d بعد تعمیم داد. ورودی یک آرایه‌ی d بعدی $A = (a_{i_1}, a_{i_2}, \ldots, a_{id})$ است، که بعدهای آن عبارتند از n_1، n_2،...، n_d، که در آن $n_1 n_2 \ldots n_d = n$. تبدیل سریع فوریه‌ی d بعدی را با تساوی زیر تعریف می‌کنیم:

$$y_{k_1,k_2,\ldots,k_d} = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} a_{j_1,j_2,\ldots,j_d} w_{n_1}^{j_1 k_1} w_{n_2}^{j_2 k_2} \cdots w_{n_d}^{j_d k_d}$$

که در آن $0 \le k_1 < n_1$، $0 \le k_2 < n_2$،...، $0 \le k_d < n_d$.

I. نشان دهید که می‌توانیم یک DFT ی dبعدی را با محاسبه‌ی DFTهای ۱بعدی بر روی هر یک از بعدها محاسبه کنیم. یعنی، ابتدا n/n_1 DFT ی ۱بعدی را در بعد ۱ محاسبه می‌کنیم. سپس با استفاده از نتیجه‌ی DFTهای بعد ۱ به عنوان ورودی، n/n_2 DFT ی ۱بعدی را در بعد ۲ محاسبه می‌کنیم. با استفاده از این نتیجه به عنوان ورودی، n/n_3 DFT ی ۱بعدی را در بعد ۳ محاسبه می‌کنیم، و به همین ترتیب تا بعد d.

II. نشان دهید که ترتیب بعدها مهم نیست، یعنی می‌توانیم یک DFT ی dبعدی را با محاسبه‌ی DFTهای ۱بعدی به هر ترتیبی برای d بعد محاسبه کنیم.

III. نشان دهید که اگر هر یک از DFTهای ۱بعدی را با محاسبه‌ی تبدیل سریع فوریه محاسبه کنیم، کل زمان برای محاسبه‌ی یک DFT ی dبعدی $O(n \lg n)$ است، مستقل از مقدار d.

۳۰-۴ محاسبه‌ی تمام مشتقات یک چندجمله‌ای در یک نقطه

با داشتن یک چندجمله‌ای $A(x)$ با کران درجه‌ی n، مشتق tام آن به صورت

$$A^{(t)}(x) = \begin{cases} A(x) & \text{اگر } t = 0 \\ \dfrac{d}{dx} A^{(t-1)}(x) & \text{اگر } 1 \le t \le n-1 \\ 0 & \text{اگر } t \ge n \end{cases}$$

تعریف می‌شود. می‌خواهیم از روی نمایش ضرایب $(a_0, a_1, \ldots, a_{n-1})$ مربوط به $A(x)$ و یک نقطه‌ی داده شده‌ی x_0، $A^{(t)}(x_0)$ را برای $t = 0, 1, \ldots, n-1$ محاسبه کنیم.

مسائل

I. با داشتن ضرایب $b_0, b_1, \ldots, b_{n-1}$ به طوری که

$$A(x)=\sum_{j=0}^{n-1} b_j (x-x_0)^j$$

نشان دهید که چطور می‌توان $A^{(t)}(x_0)$ را برای $t=0,1,\ldots,n-1$ در زمان $O(n)$ محاسبه کرد.

II. توضیح دهید چگونه می‌توان با داشتن $A(x_0+w_n^k)$ برای $k=0,1,\ldots,n-1$، مقادیر $b_0, b_1, \ldots, b_{n-1}$ را در زمان $O(n\lg n)$ محاسبه کرد.

III. اثبات کنید

$$A(x_0+w_n^k)=\sum_{r=0}^{n-1}\left(\frac{w_n^{kr}}{r!}\sum_{j=0}^{n-1} f(j)g(r-j)\right)$$

که در آن $f(j)=a_j.j!$ و

$$g(l)=\begin{cases} x_0^l/(-l)! & \text{اگر } -(n-1)\le l\le 0 \\ 0 & \text{اگر } 1\le l\le (n-1)\end{cases}$$

IV. توضیح دهید که چگونه می‌توان $A(x_0+w_n^k)$ را برای $k=0,1,\ldots,n-1$ در زمان $O(n\lg n)$ محاسبه کرد. نتیجه بگیرید که مشتقات نابدیهی $A(x)$ را برای x_0 می‌توان در زمان $O(n\lg n)$ محاسبه کرد.

۳۰-۵ *ارزیابی چندجمله‌ای‌ها در چند نقطه*

دیدیم که مسئله‌ی ارزیابی یک چندجمله‌ای با کران درجه‌ی n در یک نقطه‌ی خاص را می‌توان با استفاده از قانون هورنر در زمان $O(n)$ حل کرد. همچنین بحث کردیم که این چندجمله‌ای را می‌توان با استفاده از FFT در تمام n ریشه‌ی مختلط واحد در زمان $O(n\lg n)$ ارزیابی کرد. اکنون نشان خواهیم داد که چگونه می‌توان یک چندجمله‌ای با کران درجه‌ی n را در n نقطه‌ی دلخواه در زمان $O(n\lg^2 n)$ ارزیابی کرد.

برای انجام این کار فرض می‌کنیم که می‌توان باقی‌مانده‌ی تقسیم این چندجمله‌ای بر یک چندجمله‌ای دیگر را در زمان $O(n\lg n)$ محاسبه کرد، نتیجه‌ای که بدون اثبات آن را می‌پذیریم. به عنوان مثال، باقی‌مانده‌ی تقسیم $3x^3+x^2-3x+1$ بر x^2+x+2 برابر است با

$$(3x^3+x^2-3x+1)\bmod(x^2+x+2)=-7x+5$$

با داشتن نمایش ضرایب یک چندجمله‌ای $A(x)=\sum_{k=0}^{n-1}a_k x^k$ و n نقطه‌ی $x_0, x_1, \ldots, x_{n-1}$، می‌خواهیم n مقدار $A(x_0), A(x_1), \ldots, A(x_{n-1})$ را محاسبه کنیم. برای $0\le i\le j\le n-1$، چندجمله‌ای‌های $P_{ij}(x)=\prod_{k=i}^{j}(x-x_k)$ و $Q_{ij}(x)=A(x)\bmod P_{ij}(x)$ را تعریف

می‌کنیم. توجه کنید که درجه‌ی $Q_{ij}(x)$ حداکثر $j-i$ است.

I. اثبات کنید که برای هر نقطه‌ی z داریم $A(x) \bmod (x-z) = A(z)$.

II. اثبات کنید که $Q_{kk}(x) = A(x_k)$ و $Q_{0,n-1}(x) = A(x)$.

III. اثبات کنید که برای $i \le k \le j$ داریم $Q_{ik}(x) = Q_{ij}(x) \bmod P_{ik}(x)$ و $Q_{kj}(x) = Q_{ij}(x) \bmod P_{kj}(x)$.

IV. یک الگوریتم با زمان $O(n \lg^2 n)$ برای ارزیابی $A(x_0)$، $A(x_1)$،...، $A(x_{n-1})$ ارائه کنید.

۳۰-۶ *FFT با استفاده از محاسبات پیمانه‌ای*

همان طور که مشخص شد تبدیل سریع فوریه به استفاده از اعداد مختلط نیاز دارد، که می‌تواند به از دست دادن دقت به خاطر خطای گرد کردن منجر شود. برای بعضی مسائل از پیش می‌دانیم که جواب فقط حاوی اعداد صحیح است، و بنابراین مناسب است که نسخه‌ای از FFT داشته باشیم که بر مبنای محاسبات پیمانه‌ای باشد و تضمین کند که جواب بدون خطا محاسبه می‌شود. مثالی از چنین مسئله‌ای، ضرب دو چندجمله‌ای با ضرایب صحیح است. تمرین ۳۰-۲-۶ یک رویکرد به ما می‌دهد، با استفاده از یک پیمانه با طول $\Omega(n)$ بیت برای یافتن DFT بر روی n نقطه. در این مسئله به یک رویکرد دیگر دست می‌یابیم که از یک پیمانه با طول معقول‌تر $O(\lg n)$ استفاده می‌کند؛ این مسئله نیاز دارد که با مفاهیم فصل ۳۱ آشنا باشید. فرض کنید که n توانی از ۲ باشد.

I. فرض کنید می‌خواهیم کوچک‌ترین k را بیابیم به طوری که $p = kn + 1$ اول باشد. یک بحث مکاشفه‌ای ساده ارائه کنید که چرا انتظار داریم که k تقریباً $\ln n$ باشد. (مقدار k ممکن است بسیار بزرگ‌تر یا کوچک‌تر باشد، ولی به طور معقول انتظار داریم که به طور متوسط $O(\lg n)$ کاندیدا را بررسی کنیم.) امیدریاضی p نسبت به طول n چگونه است؟

فرض کنید g یک تولید کننده‌ی Z_p^* باشد، و $w = g^k \bmod p$.

II. بحث کنید که DFT و معکوس DFT به پیمانه‌ی p اعمال معکوس خوش‌تعریف هستند، که در آن از w به عنوان یک ریشه‌ی اصلی nام واحد استفاده شده است.

III. بحث کنید که می‌توان FFT و معکوس آن را طوری طراحی کرد که به پیمانه‌ی p در زمان $O(n \lg n)$ اجرا شوند، که در آن اعمال بر روی کلمه‌های با $O(\lg n)$ بیت در زمان ثابت اجرا می‌شوند. فرض کنید که در الگوریتم، p و w داده شده‌اند.

IV. DFT بردار $(0, 5, 3, 7, 7, 2, 1, 6)$ را به پیمانه‌ی $p = 17$ محاسبه کنید. توجه کنید که $g = 3$ یک تولید کننده‌ی Z_{17}^* است.

فصل

۳۱

الگوریتم‌های نظریه‌ی اعداد

زمانی وجود داشت که نظریه‌ی اعداد به عنوان یک مبحث زیبا ولی به شدت بی‌فایده در ریاضیات محض دیده می‌شد. امروزه به دلیل ابداء روش‌های رمزنگاری بر پایه‌ی اعداد اول بسیار بزرگ، الگوریتم‌های نظریه‌ی اعداد به صورت وسیع مورد استفاده قرار می‌گیرند. کاربردی بودن این روش‌ها به توانایی ما در یافتن اعداد اول بزرگ بستگی دارد، در حالی که امنیت آن‌ها به عدم توانایی ما در تجزیه‌ی ضرایب این اعداد اول بزرگ وابسته است. در این فصل بعضی از نظریه‌های اعداد و الگوریتم‌های مربوطه‌ی آن‌ها مورد بررسی قرار می‌گیرند که مبنای کاربردهایی مانند آن چه در بالا گفته شد، هستند.

بخش ۳۰-۱ مفاهیم اولیه‌ی نظریه‌ی اعداد را معرفی می‌کند، مانند تقسیم‌پذیری، هم‌ارزی پیمانه‌ای، و تجزیه‌ی یکتا. در بخش ۳۱-۲ یکی از قدیمی‌ترین الگوریتم‌های دنیا را خواهیم آموخت: الگوریتم اقلیدس برای محاسبه‌ی بزرگ‌ترین مقسوم علیه مشترک دو عدد صحیح. در بخش ۳۱-۳ مفاهیم محاسبات پیمانه‌ای بررسی می‌شود. سپس در بخش ۳۱-۴ مجموعه‌ی مضارب یک عدد صحیح a به پیمانه‌ی n را مورد بررسی قرار می‌دهیم، و نشان می‌دهیم که چگونه می‌توان تمام جواب‌های معادله‌ی $ax \equiv b \pmod{n}$ را با استفاده از الگوریتم اقلیدس یافت. قضیه‌ی باقی‌مانده‌ی چینی در بخش ۳۱-۵ معرفی می‌شود. بخش ۳۱-۶ توان‌های یک عدد داده شده‌ی a به پیمانه‌ی n را بررسی می‌کند، و یک الگوریتم مربع‌گیری تکراری برای محاسبه‌ی بهینه‌ی $a^b \bmod n$ با داشتن a، b، و n ارائه می‌کند. این عملیات قلب آزمایش بهینه‌ی اول بودن اعداد، و در نتیجه رمزنگاری مدرن است. در ادامه، بخش

۳۱-۷ سیستم رمزنگاری کلید عمومی RSA را توصیف می‌کند. در بخش ۳۱-۸ یک تست اول بودن تصادفی مورد بررسی قرار می‌گیرد، که از آن می‌توان برای یافتن اعداد اول بسیار بزرگ به صورت بهینه استفاده کرد، یک عملیات حیاتی در ساختن کلید برای سیستم رمزنگاری RSA. نهایتاً، در بخش ۳۱-۹ یک مکاشفه‌ی ساده ولی کارآمد برای تجزیه‌ی اعداد صحیح کوچک مورد بررسی قرار می‌گیرد. واقعیت جالبی است که بسیاری از افراد نمی‌خواهند که تجزیه‌ی اعداد کار ساده‌ای باشد، چرا که امنیت RSA به سختی تجزیه‌ی اعداد صحیح بزرگ بستگی دارد.

اندازه‌ی ورودی‌ها و هزینه‌ی محاسبات ریاضی

چون قرار است با اعداد بزرگ کار کنیم، باید نحوه‌ی تفکر خود در مورد اندازه‌ی یک ورودی و هزینه‌ی اعمال اولیه‌ی ریاضی را تغییر دهیم.

در این فصل، یک «ورودی بزرگ» معمولاً به معنی یک ورودی حاوی «اعداد صحیح بزرگ» است، در مقابل یک ورودی حاوی «اعداد زیاد» (مانند مسئله‌ی مرتب‌سازی). بنابراین اندازه‌ی یک ورودی را بر مبنای *تعداد بیت‌های* مورد نیاز برای نمایش ورودی می‌سنجیم، و نه فقط تعداد اعداد در ورودی. یک الگوریتم با ورودی‌های صحیح a_1، a_2،...، a_k یک *الگوریتم با زمان چندجمله‌ای* است اگر در زمان چندجمله‌ای نسبت به $\lg a_1$، $\lg a_2$،...، $\lg a_k$ اجرا شود، یعنی، چندجمله‌ای نسبت به طول نمایش دودویی ورودی‌ها.

در اکثر این کتاب اعمال اصلی ریاضی (ضرب، تقسیم، یا محاسبه‌ی باقی‌مانده) را به عنوان اعمال اولیه‌ای در نظر می‌گیریم که به یک واحد زمان نیاز دارند. با محاسبه‌ی تعداد این اعمال در یک الگوریتم، یک پایه برای تخمینی مناسب از زمان اجرای واقعی الگوریتم بر روی یک کامپیوتر داریم. با این حال، اگر اندازه‌ی ورودی‌های اعمال اولیه‌ی بزرگ باشد، این اعمال می‌توانند زمان زیادی بگیرند. بنابراین مناسب‌تر است که تعداد *اعمال بیتی* مورد نیاز برای یک الگوریتم نظریه‌ی اعداد را بسنجیم. در این مدل، ضرب دو عدد صحیح β بیتی با متد معمول به $\theta(\beta^2)$ عملیات بیتی نیاز دارد. به طور مشابه، عملیات تقسیم یک عدد صحیح β بیتی بر یک عدد کوچک‌تر، یا عملیات یافتن باقی‌مانده‌ی یک عدد صحیح β بیتی بر یک عدد صحیح کوچک‌تر، با استفاده از الگوریتم‌های ساده می‌تواند در زمان $\theta(\beta^2)$ انجام شود. (تمرین ۳۱-۱-۱۲ را ببینید.) الگوریتم‌های سریع‌تری هم برای این کار وجود دارند. مثلاً یک متد تقسیم و حل ساده برای ضرب دو عدد صحیح β بیتی با زمان $\theta(\beta^{\lg 3})$ وجود دارد، و یک متد سریع‌تر دارای زمان اجرای $\theta(\beta \lg \beta \lg\lg \beta)$ است. با این حال، برای استفاده‌های عملی الگوریتم $\theta(\beta^2)$ معمولاً بهترین انتخاب است، و از همین کران به عنوان پایه‌ی تحلیل‌های خود استفاده خواهیم کرد.

در این فصل، به طور کلی الگوریتم‌ها هم بر حسب اعمال ریاضی تحلیل می‌شوند و هم بر حسب تعداد اعمال بیتی مورد نیاز.

۳۱-۱ مفاهیم اولیه‌ی نظریه‌ی اعداد

این بخش مروری مختصر است بر روی مفاهیم اولیه‌ی نظریه‌ی اعداد مربوط به مجموعه‌ی $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ از اعداد صحیح و مجموعه‌ی $\mathbb{N} = \{0, 1, 2, \ldots\}$ از اعداد طبیعی.

تقسیم‌پذیری و مقسوم‌علیه‌ها

مفهوم تقسیم‌پذیری یک عدد صحیح بر عدد صحیح دیگر، مفهومی کلیدی در نظریه‌ی اعداد است. نماد $d \mid a$ (بخوانید «a، d را می‌شمارد») به این معنی است که $a = kd$ برای یک عدد صحیح k. تمام اعداد صحیح ۰ را می‌شمارند. اگر $a > 0$ و $d \mid a$، آن گاه $|a| \geq |d|$. اگر $d \mid a$، آن گاه همچنین می‌گوییم a *ضریبی* از d است. اگر d، a را نشمارد، می‌نویسیم $d \nmid a$.

اگر $d \mid a$ و $d \geq 0$، می‌گوییم d یک مقسوم‌علیه a است. توجه کنید که $d \mid a$ اگر و فقط اگر $-d \mid a$، بنابراین با تعریف مقسوم‌علیه‌ها به صورت مثبت هیچ کلیتی از دست نمی‌رود، چرا که می‌دانیم منفی هر مقسوم‌علیه a هم a را می‌شمارد. یک مقسوم‌علیه یک عدد صحیح a حداقل برابر است با ۱، ولی نمی‌تواند بزرگ‌تر از $|a|$ باشد. به عنوان مثال مقسوم‌علیه‌های ۲۴ عبارتند از ۱، ۲، ۳، ۴، ۶، ۸، ۱۲، و ۲۴.

هر عدد صحیح a دو *مقسوم‌علیه بدیهی* دارد، که عبارتند از ۱ و a. مقسوم‌علیه‌های نابدیهی a، به *عوامل* a (یا فاکتورهای a) هم مشهورند. به عنوان مثال، عوامل ۲۰ عبارتند از ۲، ۴، ۵، و ۱۰.

اعداد اول و مرکب

یک عدد صحیح $a > 1$ که تنها مقسوم‌علیه‌های آن، مقسوم‌علیه‌های بدیهی ۱ و a هستند، یک عدد *اول* (prime) نام دارد. اعداد اول خصوصیات مهم بسیاری دارند و یک نقش اساسی در نظریه‌ی اعداد بازی می‌کنند. ۲۰ عدد اول، به ترتیب عبارتند از

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71.$$

تمرین ۳۱-۱-۱ از شما می‌خواهد اثبات کنید که تعداد اعداد اول بی‌شمار است. اگر یک عدد صحیح $a > 1$ اول نباشد، به آن یک عدد *مرکب* (composite) می‌گوییم. به عنوان مثال، ۳۹ یک عدد مرکب است چرا که ۳۱۳۹. به عدد ۱، *واحد* (unit) می‌گوییم، و این عدد نه اول است و نه مرکب. به طور مشابه، ۰ و تمام اعداد منفی نه اول هستند و نه مرکب.

قضیه‌ی تقسیم، باقی‌مانده‌ها، و هم‌ارزی پیمانه‌ای

با داشتن یک عدد صحیح n، اعداد صحیح را می‌توان به دو دسته تقسیم کرد، آن‌هایی که مضرب n هستند و آن‌هایی که مضرب n نیستند. بخش اعظمی از نظریه‌ی اعداد بر پایه‌ی پالایش این دسته‌بندی است که با تقسیم اعدادی که مضرب n نیستند بر حسب باقی‌مانده‌ی آن‌ها بر n انجام می‌شود. قضیه‌ی زیر پایه‌ی این پالایش است. اثبات این قضیه در این جا ارائه نخواهد شد.

قضیه‌ی (۳۱-۱) (قضیه‌ی تقسیم) برای هر عدد صحیح a و هر عدد صحیح مثبت n، اعداد صحیح یکتای q و r وجود دارند به طوری که $0 \le r < n$ و $a = qn + r$.

مقدار $q = \lfloor a/n \rfloor$ *خارج قسمت* (quotient) تقسیم است. مقدار $r = a \bmod n$ *باقی‌مانده‌ی* (reminder) تقسیم نام دارد. داریم $n \mid a$ اگر و فقط اگر $a \bmod n = 0$.

اعداد صحیح را می‌توان بسته به باقی‌مانده‌ی آن‌ها بر n، به n کلاس هم‌ارزی تقسیم کرد. *کلاس هم‌ارزی به پیمانه‌ی n* حاوی یک عدد صحیح a به صورت زیر است:

$$[a]_n = \{a + kn : k \in \mathbb{Z}\}$$

به عنوان مثال داریم $[3]_7 = \{..., -11, -4, 3, 10, 17, ...\}$؛ نمادهای دیگر برای این مجموعه عبارتند از $[-4]_7$ و $[10]_7$. با استفاده از نماد تعریف شده در بخش ۳-۲، می‌توانیم بگوییم که نوشتن $a \in [b]_n$ معادل است با نوشتن $a \equiv b \pmod n$. مجموعه‌ی تمام این کلاس‌های هم‌ارزی به صورت

$$\mathbb{Z}_n = \{[a]_n : 0 \le a \le n-1\} \qquad (۳۱-۱)$$

است. معمولاً تعریف

$$\mathbb{Z}_n = \{0, 1, ..., n-1\} \qquad (۳۱-۲)$$

هم دیده می‌شود، که باید معادل با تساوی (۳۱-۱) در نظر گرفته شود، با این درک که 0 نشان‌دهنده‌ی $[0]_n$ است، ۱ نشان‌دهنده‌ی $[1]_n$، و الی آخر؛ هر کلاس با کوچک‌ترین عنصر نامنفی آن کلاس نمایش داده می‌شود. با این حال، باید کلاس هم‌ارزی مربوط به عدد در نظر گرفته شود، و نه خود عدد. به عنوان مثال، یک ارجاع به -1 به عنوان عضوی از $\mathbb{Z}_n$، یک ارجاع به $[n-1]_n$ است، چرا که داریم $-1 \equiv n - 1 \pmod n$.

مقسوم‌علیه‌های مشترک و بزرگ‌ترین مقسوم‌علیه مشترک

اگر d یک مقسوم‌علیه a باشد، و همچنین یک مقسوم‌علیه b، آن گاه d یک *مقسوم‌علیه مشترک* (common divisor) a و b است. مثلاً مقسوم‌علیه‌های ۳۰ عبارتند از ۱، ۲، ۳، ۵، ۶، ۱۰، ۱۵، و ۳۰، و بنابراین مقسوم‌علیه‌های مشترک ۲۴ و ۳۰ عبارتند از ۱، ۲، ۳، و ۶. توجه کنید که ۱ مقسوم‌علیه مشترک هر دو عدد صحیح است.

یک خصوصیت مهم مقسوم‌علیه‌های مشترک این است که

$$d \mid a \text{ و } d \mid b \text{ نتیجه می‌دهد که } d \mid (a+b) \text{ و } d \mid (a-b). \qquad (۳۱-۳)$$

به طور کلی‌تر، برای هر دو عدد صحیح x و y داریم

$$d \mid a \text{ و } d \mid b \text{ نتیجه می‌دهد که } d \mid (ax + by). \qquad (۳۱-۴)$$

همچنین، اگر $a \mid b$، یا داریم $|a| \le |b|$ و یا $b = 0$، که ایجاب می‌کند

بزرگ‌ترین مقسوم‌علیه مشترک (greatest common divisor) دو عدد صحیح a و b (که به آن ب.م.م هم می‌گوییم–م) که هر دو غیر صفر هستند، برابر است با بزرگ‌ترین عدد در بین مقسوم‌علیه‌های مشترک a و b، که آن را با $\gcd(a,b)$ نشان می‌دهیم. مثلاً داریم $\gcd(24,30)=6$، $\gcd(5,7)=1$، و $\gcd(0,9)=9$. اگر a و b هر دو صفر نباشند، آن گاه $\gcd(a,b)$ یک عدد صحیح بین ۱ و $\min(|a|,|b|)$ است. تعریف می‌کنیم $\gcd(0,0)=0$؛ این تعریف برای اعتبار خصوصیات استاندارد تابع ب.م.م (مانند تساوی ۳۱–۹ در زیر) ضروری است.

تساوی‌های زیر، خصوصیات اصلی تابع ب.م.م هستند:

$$\gcd(a,b)=\gcd(b,a) \tag{۳۱–۶}$$

$$\gcd(a,b)=\gcd(-a,b) \tag{۳۱–۷}$$

$$\gcd(a,b)=\gcd(|a|,|b|) \tag{۳۱–۸}$$

$$\gcd(a,0)=|a| \tag{۳۱–۹}$$

$$\gcd(a,ka)=|a| \tag{۳۱–۱۰}$$

قضیه‌ی زیر یک توصیف جایگزین و مفید از $\gcd(a,b)$ به دست می‌دهد.

قضیه‌ی ۳۱–۲ اگر a و b دو عدد صحیح باشند که هر دو صفر نیستند، آن گاه $\gcd(a,b)$ برابر است با کوچک‌ترین عضو مثبت مجموعه‌ی $\{ax+by : x,y \in \mathbb{Z}\}$ از ترکیبات خطی a و b.

اثبات فرض کنید s کوچک‌ترین ترکیب خطی مثبت از a و b باشد، و داشته باشیم $s=ax+by$ برای $x,y \in \mathbb{Z}$. فرض کنید $q=\lfloor a/s \rfloor$. تساوی (۳–۸) ایجاب می‌کند که

$$\begin{aligned} a \bmod s &= a-qs \\ &= a-q(ax+by) \\ &= a(1-qx)+b(-qy) \end{aligned}$$

و بنابراین $a \bmod s$ هم یک ترکیب خطی از a و b است. ولی از آن‌جایی که $0 \le a \bmod s < s$، داریم $a \bmod s = 0$، چرا که s کوچک‌ترین ترکیب خطی مثبت از a و b است. بنابراین $s \mid a$، و به طور مشابه $s \mid b$. پس s یک مقسوم‌علیه مشترک از a و b است، و داریم $s \le \gcd(a,b)$. تساوی (۳۱–۴) ایجاب می‌کند که $\gcd(a,b) \mid s$، چرا که $\gcd(a,b)$ هر دوی a و b را می‌شمارد، و s یک ترکیب خطی از a و b است. ولی $\gcd(a,b) \mid s$ و $s > 0$ نتیجه می‌دهد که $\gcd(a,b) \le s$. با ترکیب $s \le \gcd(a,b)$ و $\gcd(a,b) \le s$ خواهیم داشت $\gcd(a,b)=s$؛ نتیجه می‌گیریم که s برابر است با بزرگ‌ترین مقسوم‌علیه مشترک a و b. ■

نتیجه‌ی ۳۱–۳ برای هر دو عدد صحیح a و b، اگر $d \mid a$ و $d \mid b$، آن گاه $d \mid \gcd(a,b)$.

اثبات این نتیجه از تساوی (۳۱-۴) حاصل می‌شود، چرا که طبق قضیه‌ی ۳۱-۲، $\gcd(a,b)$ یک ترکیب خطی از a و b است. ■

نتیجه‌ی ۳۱-۴ برای تمام اعداد صحیح a و b و هر عدد صحیح نامنفی n،

$$\gcd(an,bn) = n\gcd(a,b)$$

اثبات اگر $n = 0$ حکم بدیهی است. اگر $n > 0$، آن گاه $\gcd(an,bn)$ برابر است با کوچک‌ترین عضو مثبت مجموعه‌ی $\{anx + bny\}$، که برابر است با n برابر کوچک‌ترین عضو مثبت مجموعه‌ی $\{ax + by\}$.

نتیجه‌ی ۳۱-۵ برای تمام اعداد صحیح مثبت n، a، و b، اگر $n \mid ab$ و $\gcd(a,n) = 1$، آن گاه $n \mid b$.

اثبات اثبات به عنوان تمرین ۳۱-۱-۵ واگذار شده است. ■

اعداد اول نسبی

می‌گوییم دو عدد صحیح a و b *نسبت به هم اول* هستند اگر تنها مقسوم‌علیه مشترک آن‌ها ۱ باشد، یعنی، اگر $\gcd(a,b) = 1$. به عنوان مثال ۸ و ۱۵ نسبت به هم اول هستند، چرا که مقسوم‌علیه‌های ۸ عبارتند از ۱، ۲، ۴، و ۸، در حالی که مقسوم‌علیه‌های ۱۵ عبارتند از ۱، ۳، ۵، و ۱۵. قضیه‌ی زیر می‌گوید که اگر دو عدد صحیح نسبت به p اول باشند، آن گاه ضرب آن‌ها هم نسبت به p اول است.

قضیه‌ی ۳۱-۶ برای هر سه عدد صحیح a، b، و p، اگر داشته باشیم $\gcd(a,p) = 1$ و $\gcd(b,p) = 1$، آن گاه $\gcd(ab,p) = 1$.

اثبات از قضیه‌ی ۳۱-۲ نتیجه می‌شود که اعداد صحیح x، y، x'، و y' وجود دارند به طوری که

$$ax + py = 1$$
$$bx' + py' = 1$$

با ضرب این دو تساوی داریم

$$ab(xx') + p(ybx' + y'ax + pyy') = 1$$

چون ۱ ترکیبی خطی از ab و p است، با مراجعه به قضیه‌ی ۳۱-۲ اثبات کامل می‌شود. ■

می‌گوییم اعداد صحیح $n_1, n_2, \ldots, n_k$ *دو به دو نسبت به هم اول* هستند اگر برای $i \neq j$ داشته باشیم $\gcd(n_i, n_j) = 1$.

تجزیه‌ی یکتا

یک حقیقت ابتدایی ولی مهم در مورد تقسیم‌پذیری بر اعداد اول در زیر آمده است.

قضیه‌ی ۳۱-۶ برای تمام اعداد اول p و تمام اعداد صحیح a و b اگر $p \mid ab$ آن گاه $p \mid a$ یا $p \mid b$ (و یا هر دو).

اثبات طبق برهان خلف، فرض کنید که $p \mid ab$ ولی $p \nmid a$ و $p \nmid b$. بنابراین $\gcd(a,p)=1$ و $\gcd(b,p)=1$، چرا که تنها مقسوم‌علیه‌های p عبارتند از ۱ و p، و با این فرض که p نه a را می‌شمارد و نه b را. سپس قضیه‌ی ۳۱-۶ ایجاب می‌کند که $\gcd(ab,p)=1$، که با این فرض که $p \mid ab$ تناقض دارد، چرا که $p \mid ab$ نتیجه می‌دهد $\gcd(ab,p)=p$. این تناقض اثبات را کامل می‌کند. ■

یک نتیجه‌ی قضیه‌ی ۳۱-۷ این است که هر عدد صحیح مرکب یک تجزیه‌ی یکتا به اعداد اول دارد.

قضیه‌ی ۳۱-۷ (تجزیه‌ی یکتا) یک عدد مرکب a را می‌توان دقیقاً به یک شکل به صورت حاصل ضربی به شکل

$$a = p_1^{e_1} p_2^{e_2} \dots p_r^{e_r}$$

نوشت، که در آن p_iها اعداد اول هستند، $p_1 < p_2 < \dots < p_r$، و e_iها اعداد صحیح مثبت هستند.

اثبات اثبات به عنوان تمرین ۳۱-۱-۱۱ واگذار شده است. ■

به عنوان مثال، عدد ۶۰۰۰ را می‌توان به صورت یکتا به شکل $۲^۴ \times ۳ \times ۵$ تجزیه کرد.

تمرین‌ها

۳۱-۱-۱ اثبات کنید اگر $a > b > 0$ و $c = a + b$، آن گاه $c \bmod a = b$.

۳۱-۱-۲ اثبات کنید که تعداد اعداد اول بی‌شمار است. (*راهنمایی:* از این که هیچ یک از اعداد اول $p_1, p_2, \dots, p_k$، عدد $(p_1 p_2 \dots p_k) + 1$ را نمی‌شمارند، چه استفاده‌ای می‌توانید بکنید؟)

۳۱-۱-۳ اثبات کنید که اگر $a \mid b$ و $b \mid c$ آن گاه $a \mid c$.

۳۱-۱-۴ اثبات کنید که اگر p یک عدد اول باشد و $0 < k < p$، آن گاه $\gcd(k,p)=1$.

۳۱-۱-۵ نتیجه‌ی ۳۱-۵ را اثبات کنید.

۳۱-۱-۶ اثبات کنید که اگر p یک عدد اول باشد و $0 < k < p$، آن گاه $p \left| \binom{p}{k} \right.$. نتیجه بگیرید که برای تمام اعداد صحیح a و b و تمام اعداد اول p، داریم

$$(a+b)^p \equiv a^p + b^p \pmod{p}$$

۳۱-۱-۷ اثبات کنید که اگر a و b اعداد صحیح دلخواه باشند به طوری که $a \mid b$ و $b > 0$، آن گاه

$$(x \bmod b) \bmod a = x \bmod a$$

برای هر عدد x. با فرض‌هایی یکسان، اثبات کنید که برای هر دو عدد صحیح x و y، $x \equiv y \pmod{b}$ نتیجه می‌دهد $x \equiv y \pmod{a}$.

۳۱-۱-۸ برای عدد صحیح $k > 0$، می‌گوییم عدد صحیح n یک توان kام است اگر یک عدد صحیح a وجود داشته باشد به طوری که $a^k = n$. می‌گوییم $n > 1$ یک توان نابدیهی است اگر برای یک عدد صحیح $k > 1$ یک توان kام باشد. نشان دهید که چطور می‌توان در زمان چند جمله‌ای نسبت به β تشخیص داد که یک عدد صحیح β بیتی، یک توان نابدیهی است.

۳۱-۱-۹ تساوی‌های (۳۱-۶)-(۳۱-۱۰) را اثبات کنید.

۳۱-۱-۱۰ نشان دهید که عملگر ب.م.م شرکت‌پذیر است. یعنی، اثبات کنید که برای تمام اعدادصحیح a، b، و c، داریم

$$\gcd(a, \gcd(b,c)) = \gcd(\gcd(a,b),c)$$

★ ۳۱-۱-۱۱ قضیه‌ی ۳۱-۸ را اثبات کنید.

۳۱-۱-۱۲ الگوریتم‌هایی کارآمد برای محاسبه‌ی تقسیم یک عدد صحیح β بیتی بر یک عدد صحیح کوچک‌تر و محاسبه‌ی باقی‌مانده‌ی تقسیم یک عدد صحیح β بیتی بر یک عدد صحیح کوچک‌تر ارائه کنید. الگوریتم‌های شما باید در زمان $O(\beta^2)$ اجرا شوند.

۳۱-۱-۱۳ یک الگوریتم کارآمد برای تبدیل یک عدد β بیتی (دودویی) داده شده به نمایش دهدهی ارائه کنید. بحث کنید که اگر ضرب و تقسیم اعداد صحیحی که طول آن‌ها حداکثر β بیت است در زمان $M(\beta)$ اجرا شود، آن گاه می‌توان تبدیل دودویی به دهدهی را در زمان $\theta(M(\beta)\lg\beta)$ انجام داد. (راهنمایی: از یک رویکرد تقسیم و حل استفاده کنید، که نیمه‌های بالایی و پایینی نتیجه را با استفاده از بازگشت‌های مختلف به دست می‌آورد.)

۳۱-۲ [illegible]

در این بخش الگوریتم اقلیدس را برای محاسبه‌ی بهینه‌ی بزرگ‌ترین مقسوم‌علیه مشترک دو عدد صحیح توصیف می‌کنیم. تحلیل زمان اجرا یک رابطه‌ی بسیار جالب توجه با اعداد فیبوناچی دارد: این اعداد ورودی بدترین حالت برای الگوریتم اقلیدس هستند.

در این بخش توجه خود را بر روی اعداد صحیح نامنفی متمرکز می‌کنیم. اعمال این محدودیت مجاز است، چرا که طبق تساوی (۳۱-۸) داریم $\gcd(a,b) = \gcd(|a|,|b|)$.

در عمل می‌توانیم $\gcd(a,b)$ را برای اعداد صحیح a و b از تجزیه‌ی a و b به اعداد اول محاسبه کنیم. در واقع، اگر

$$a = p_1^{e_1} p_2^{e_2} \dots p_r^{e_r} \tag{۳۱-۱۱}$$

$$b = p_1^{f_1} p_2^{f_2} \dots p_r^{f_r} \tag{۳۱-۱۲}$$

که در آن از توان‌های صفر برای یکسان کردن مجموعه‌ی اعداد اول $p_1, p_2, \dots, p_r$ برای a و b استفاده شده است، همان طور که تمرین ۳۱-۲-۱ از شما می‌خواهد نشان دهید، داریم

$$\gcd(a,b) = p_1^{\min(e_1,f_1)} p_2^{\min(e_2,f_2)} \dots p_r^{\min(e_r,f_r)} \tag{۳۱-۱۳}$$

با این حال همان طور که در بخش ۳۱-۹ نشان خواهیم داد، بهترین الگوریتم یافته شده تا کنون برای تجزیه در زمان چندجمله‌ای اجرا نمی‌شود. بنابراین، این رویکرد برای محاسبه‌ی بزرگ‌ترین مقسوم‌علیه مشترک به نظر نمی‌آید که چندان بهینه باشد.

الگوریتم اقلیدس برای محاسبه‌ی بزرگ‌ترین مقسوم‌علیه مشترک بر مبنای قضیه‌ی زیر است.

قضیه‌ی ۳۱-۹ (قضیه‌ی باقیمانده برای بزرگترین مقسوم‌علیه)

برای هر عدد صحیح نامنفی a و هر عدد صحیح مثبت b،

$$\gcd(a,b) = \gcd(b, a \bmod b)$$

اثبات نشان خواهیم داد که $\gcd(a,b)$ و $\gcd(b,a \bmod b)$ یکدیگر را می‌شمارند، و بنابراین طبق تساوی (۳۱-۵) باید با هم برابر باشند (چرا که هر دو نامنفی هستند).

ابتدا نشان می‌دهیم $\gcd(a,b) \mid \gcd(b,a \bmod b)$. اگر قرار دهیم $d = \gcd(a,b)$، آن گاه $d \mid a$ و $d \mid b$. طبق تساوی (۳-۸)، $(a \bmod b) = a - qb$، که در آن $q = \lfloor a/b \rfloor$، و $(a \bmod b)$ ترکیبی خطی از a و b است. از این رو، تساوی (۳۱-۴) می‌گوید که $d \mid (a \bmod b)$. بنابراین، چون $d \mid b$ و $d \mid (a \bmod b)$، نتیجه‌ی ۳۱-۳ ایجاب می‌کند که $d \mid \gcd(b,a \bmod b)$، یا به طور معادل،

$$\gcd(a,b) \mid \gcd(b, a \bmod b) \tag{۳۱-۱۴}$$

نشان دادن این که $\gcd(b,a \bmod b) \mid \gcd(a,b)$ تقریباً به همین صورت است. اگر قرار دهیم $d = \gcd(b,a \bmod b)$، آن گاه $d \mid b$ و $d \mid (a \bmod b)$. چون $a = qb + (a \bmod b)$، که در آن $q = \lfloor a/b \rfloor$، پس a ترکیبی خطی از b و $(a \bmod b)$ است. طبق تساوی (۳۱-۴) نتیجه می‌گیریم که $d \mid a$. چون $d \mid b$ و $d \mid a$، طبق نتیجه‌ی ۳۱-۳ داریم $d \mid \gcd(a,b)$، یا به طور معادل

$$\gcd(b,a \bmod b) \mid \gcd(a,b) \tag{۳۱-۱۵}$$

استفاده از تساوی (۳۱-۵) برای ترکیب تساوی‌های (۳۱-۱۴) و (۳۱-۱۵) اثبات را کامل می‌کند. ■

الگوریتم اقلیدس

کتاب *عناصر* (elements) اقلیدس (حدود ۳۰۰ قبل از میلاد) الگوریتم زیر را برای یافتن ب.م.م توضیح

می‌دهد، با این حال ممکن است ریشه‌ی آن به قبل از اقلیدس بازگردد. الگوریتم اقلیدس به صورت یک برنامه‌ی بازگشتی توصیف شده است که مستقیماً بر پایه‌ی قضیه‌ی ۳۱-۹ است. ورودی‌های a و b اعداد صحیح نامنفی دلخواه هستند.

```
EUCLID(a, b)
1  if b = 0
2      return a
3  else return EUCLID(b, a mod b)
```

به عنوان یک مثال از اجرای EUCLID، محاسبه‌ی gcd(۳۰,۲۱) را در نظر بگیرید:

$$\begin{aligned}\text{EUCLID}(30,21) &= \text{EUCLID}(21,9)\\ &= \text{EUCLID}(9,3)\\ &= \text{EUCLID}(3,0)\\ &= 3\end{aligned}$$

در این محاسبه، سه فراخوانی بازگشتی از EUCLID وجود دارد.

صحت رویه‌ی EUCLID از قضیه‌ی ۳۱-۹ نتیجه می‌شود، و از این واقعیت که اگر الگوریتم در خط ۲، a را بازگرداند، در این صورت داریم $b = 0$، و بنابراین تساوی (۳۱-۹) می‌گوید که $\gcd(a,b) = \gcd(a,0) = 0$. الگوریتم نمی‌تواند تا بی‌نهایت بازگشت کند، چرا که آرگومان دوم در هر فراخوانی بازگشتی کاهش می‌یابد، و همچنین همیشه نامنفی است. بنابراین EUCLID همیشه پایان می‌یابد و جواب صحیح را بازمی‌گرداند.

زمان اجرای الگوریتم اقلیدس

بدترین حالت زمان اجرای EUCLID را به صورت تابعی از اندازه‌ی a و b تحلیل خواهیم کرد. بدون از دست دادن کلیت، فرض می‌کنیم که $a > b \geq 0$. این فرض بدین صورت تأیید می‌شود که اگر داشته باشیم $b > a \geq 0$، آن گاه EUCLID(a,b) بی درنگ به صورت بازگشتی EUCLID(b,a) را فراخوانی می‌کند. یعنی اگر اولین آرگومان کوچک‌تر از دومین آرگومان باشد، EUCLID با یک فراخوانی بازگشتی جای آن دو را عوض می‌کند و سپس ادامه می‌دهد. به طور مشابه، اگر $a = b > 0$، آن گاه رویه بعد از یک فراخوانی بازگشتی پایان می‌یابد، چرا که $a \bmod b = 0$.

زمان اجرای کلی EUCLID متناسب است با تعداد فراخوانی‌های بازگشتی انجام شده. در تحلیل زیر از اعداد فیبوناچی F_k استفاده می‌کنیم، که در رابطه‌ی بازگشتی (۳-۲۱) تعریف شده‌اند.

لم (۳۱-۱۰) اگر $a > b \geq 1$ و فراخوانی EUCLID(a,b) به تعداد $k \geq 1$ فراخوانی بازگشتی انجام دهد، آن گاه $a \geq F_{k+2}$ و $b \geq F_{k+1}$.

اثبات اثبات به وسیله‌ی استقرا بر روی k انجام می‌شود. برای پایه‌ی استقرا، فرض کنید $k = 1$. آن گاه $b \geq 1 = F_2$، و چون $a > b$، باید داشته باشیم $a \geq 2 = F_3$. چون $b > (a \bmod b)$، در هر فراخوانی بازگشتی آرگومان اول اکیداً بزرگ‌تر از دومی است؛ بنابراین فرض این که $a > b$ برای تمام

فراخوانی‌های بازگشتی برقرار است.

به صورت استقرایی فرض کنید که لم برای $k-1$ فراخوانی بازگشتی برقرار است؛ سپس اثبات می‌کنیم که برای k فراخوانی بازگشتی هم درست خواهد بود. چون $k > 0$، داریم $b > 0$ و EUCLID(a,b) به صورت بازگشتی EUCLID$(b, a \bmod b)$ را فراخوانی می‌کند، که این فراخوانی، خود $k-1$ فراخوانی بازگشتی انجام می‌دهد.سپس فرض استقرا ایجاب می‌کند که $b \geq F_{k+1}$ (که بخشی از لم را اثبات می‌کند)، و $(a \bmod b) \geq F_k$. داریم

$$\begin{aligned} b + (a \bmod b) &= b + (a - \lfloor b/a \rfloor) \\ &\leq a \end{aligned}$$

چرا که $a > b > 0$ ایجاب می‌کند $\lfloor a/b \rfloor \geq 1$. بنابراین،

$$\begin{aligned} a &\geq b + (a \bmod b) \\ &\geq F_{k+1} + F_k \\ &= F_{k+2} \end{aligned}$$

قضیه‌ی زیر، نتیجه‌ی مستقیم از این لم است.

قضیه ۳۱-۱۲ برای هر عدد صحیح $k \geq 1$، اگر $a > b \geq 1$ و $b < F_{k+1}$، آن گاه فراخوانی EUCLID(a,b) کم‌تر از k فراخوانی بازگشتی انجام می‌دهد.

می‌توانیم نشان دهیم که کران بالای داده شده در قضیه‌ی ۳۱-۱۱، بهترین کران بالای ممکن است، بدین صورت که نشان می‌دهیم فراخوانی EUCLID(F_{k+1}, F_k) دقیقاً $k-1$ فراخوانی بازگشتی انجام می‌دهد ($k \geq 2$). از استقرا بر روی k استفاده می‌کنیم. برای حالت پایه داریم $k = 2$، و فراخوانی EUCLID(F_3, F_2) دقیقاً یک فراخوانی بازگشتی (EUCLID(1, 0)) انجام می‌دهد. (باید از $k = 2$ شروع کنیم چرا که وقتی $k = 1$، نامساوی $F_2 > F_1$ برقرار نیست.) برای گام استقرا فرض می‌کنیم EUCLID(F_k, F_{k-1}) دقیقاً $k-1$ فراخوانی بازگشتی انجام می‌دهد. برای $k > 2$ داریم $F_k > F_{k-1} > 0$ و $F_{k+1} = F_k + F_{k-1}$، و بنابراین طبق تمرین ۳۱-۱-۱، داریم $F_{k+1} \bmod F_k = F_{k-1}$. از این رو خواهیم داشت

$$\begin{aligned} \gcd(F_{k+1}, F_k) &= \gcd(F_k, (F_{k+1} \bmod F_k)) \\ &= \gcd(F_k, F_{k-1}) \end{aligned}$$

بنابراین EUCLID(F_{k+1}, F_k) یک فراخوانی بیشتر از EUCLID(F_k, F_{k-1})، یا دقیقاً $k-1$ فراخوانی بازگشتی انجام می‌دهد، که برابر است با کران بالای داده شده در قضیه‌ی ۳۱-۱۱.

چون F_k تقریباً برابر است با $\varphi^k/\sqrt{5}$، که در آن φ نسبت طلایی $(1+\sqrt{5})/2$ است، که در تساوی (۳-۲۲) تعریف شده است، تعداد فراخوانی‌های بازگشتی EUCLID از مرتبه‌ی $O(\lg b)$ است. (تمرین

۳۱-۲-۵ را برای یک کران نزدیک‌تر ببینید.) نتیجه این که اگر EUCLID به دو عدد β بیتی اعمال شود، در این صورت $O(\beta)$ عملیات ریاضی و $O(\beta^3)$ عملیات بیتی انجام می‌دهد (با این فرض که ضرب و تقسیم اعداد β بیتی به $O(\beta^2)$ عملیات بیتی نیاز دارند). مسئله‌ی ۳۱-۲ از شما می‌خواهد که یک کران $O(\beta^2)$ بر روی تعداد تعداد عملیات بیتی اثبات کنید.

شکل گسترش یافته‌ی الگوریتم اقلیدس

اکنون الگوریتم اقلیدس را بازنویسی می‌کنیم تا اطلاعات اضافی مفیدی را هم محاسبه کند. به طور خاص، الگوریتم را گسترش می‌دهیم تا ضرایب صحیح x و y را محاسبه کند به طوری که

$$d = \gcd(a,b) = ax + by \tag{۳۱-۱۶}$$

توجه کنید که ممکن است x و y صفر یا منفی باشند. بعداً برای محاسبه‌ی معکوس‌های ضربی پیمانه‌ای این ضرایب را مفید خواهیم یافت. رویه‌ی EXTENDED-EUCLID یک جفت عدد صحیح نامنفی را به عنوان ورودی دریافت می‌کند و یک سه‌تایی به شکل (d, x, y) بازمی‌گرداند که تساوی (۳۱-۱۶) را ارضا می‌کنند.

```
EXTENDED-EUCLID(a, b)
1  if b == 0
2      return (a, 1, 0)
3  else (d', x', y') = EXTENDED-EUCLID(b, a mod b)
4      (d, x, y) = (d', y', x' - ⌊a/b⌋ y')
5      return (d, x, y)
```

شکل ۳۱-۱ اجرای EXTENDED-EUCLID را با محاسبه‌ی gcd(۹۹, ۷۸) نشان می‌دهد.

رویه‌ی EXTENDED-EUCLID نسخه‌ی تغییر یافته‌ای از رویه‌ی EUCLID است. خط ۱ معادل است با تست "$b = 0$" در خط ۱ رویه‌ی EUCLID. اگر $b = 0$، آن گاه EXTENDED-EUCLID در خط

a	b	$\lfloor a/b \rfloor$	d	x	y
99	78	1	3	−11	14
78	21	3	3	3	−11
21	15	1	3	−2	3
15	6	2	3	1	−2
6	3	2	3	0	1
3	0	—	3	1	0

شکل ۳۱-۱ یک مثال از عملیات EXTENDED-EUCLID بر روی ورودی‌های ۹۹ و ۷۸. هر خط نشان‌دهنده‌ی یک سطح از بازگشت است: مقادیر ورودی‌های a و b، مقدار محاسبه شده‌ی $\lfloor a/b \rfloor$ و مقادیر بازگردانده شده‌ی d، x، و y. سه‌تایی (d, x, y) بازگردانده شده، در سطح بالایی بازگشت تبدیل به سه‌تایی (d', x', y') می‌شوند که از آن‌ها در محاسبات استفاده می‌شود. فراخوانی EXTENDED-EUCLID (۹۹, ۷۸)، سه‌تایی (۳, −۱۱, ۱۴) را بازمی‌گرداند، و بنابراین $\gcd(۹۹, ۷۸) = ۳ = ۹۹ \times (-۱۱) + ۷۸ \times ۱۴$.

۲، $d = a$ را بازمی‌گرداند، به همراه ضرایب $x = 1$ و $y = 0$، به طوری که $a = ax + by$. اگر $b \neq 0$، EXTENDED-EUCLID ابتدا (d', x', y') را محاسبه می‌کند به طوری که $d' = \gcd(b, a \bmod b)$ و

$$d' = bx' + (a \bmod b)y' \qquad (۳۱-۱۷)$$

مانند EUCLID، در این حالت داریم $d = \gcd(a,b) = d' = \gcd(b, a \bmod b)$. برای به دست آوردن x و y به طوری که $d = ax + by$، با بازنویسی تساوی (۳۱-۱۷) با استفاده از تساوی $d = d'$ و تساوی (۳-۸) آغاز می‌کنیم:

$$\begin{aligned} d &= bx' + (a - \lfloor a/b \rfloor b)y' \\ &= ay' + b(x' - \lfloor a/b \rfloor y') \end{aligned}$$

بنابراین، انتخاب $x = y'$ و $y = x' - \lfloor a/b \rfloor y'$ تساوی $d = ax + by$ را ارضا، و صحت EXTENDED-EUCLID را اثبات می‌کند.

چون تعداد فراخوانی‌های بازگشتی انجام شده در EUCLID برابر است با تعداد فراخوانی‌های بازگشتی انجام شده در EXTENDED-EUCLID، زمان اجرای EUCLID و EXTENDED-EUCLID برابر است، با اختلاف یک ضریب ثابت. یعنی برای $a > b > 0$ تعداد فراخوانی‌های بازگشتی $O(\lg b)$ است.

تمرین‌ها

۱-۲-۳۱ اثبات کنید که تساوی‌های (۳۱-۱۱) و (۳۱-۱۲)، تساوی (۳۱-۱۳) را نتیجه می‌دهند.

۲-۲-۳۱ مقادیر (d, x, y) را که فراخوانی EXTENDED-EUCLID (۸۹۹, ۴۹۳) بازمی‌گرداند، محاسبه کنید.

۳-۲-۳۱ اثبات کنید که برای تمام اعداد صحیح a، k، و n،

$$\gcd(a,n) = \gcd(a + kn, n)$$

۴-۲-۳۱ EUCLID را به صورت تکراری بازنویسی کنید به طوری که از مقدار ثابتی حافظه استفاده کند (یعنی فقط تعداد ثابتی عدد صحیح را ذخیره کند).

۵-۲-۳۱ اگر $a > b \geq 0$، نشان دهید که فراخوانی EUCLID (a,b) حداکثر $1 + \log_\varphi b$ فراخوانی بازگشتی انجام می‌دهد. این کران را به $1 + \log_\varphi(b/\gcd(a,b))$ بهبود بخشید.

۶-۲-۳۱ EXTENDED-EUCLID (F_{k+1}, F_k) چه چیزی را بازمی‌گرداند؟ درستی جواب خود را اثبات کنید.

۷-۲-۳۱ توابع ب.م.م را برای بیش از دو ورودی با رابطه‌ی بازگشتی $\gcd(a_0, a_1, \ldots, a_n) = \gcd(a_0, \gcd(a_1, a_2, \ldots, a_n))$ تعریف می‌کنیم. نشان دهید که تابع ب.م.م مستقل از ترتیب قرار گرفتن آرگومان‌ها، جواب ثابتی را بازمی‌گرداند. همچنین

نشان دهید که چگونه می‌توان اعداد صحیح $x_0, x_1, ..., x_n$ را یافت به طوری که $\gcd(a_0, a_1, ..., a_n) = a_0 x_0 + a_1 x_1 + ... + a_n x_n$. نشان دهید که تعداد تقسیم‌های انجام شده توسط الگوریتم شما $O(n + \lg(\max\{a_0, a_1, ..., a_n\}))$ است.

۸-۲-۳۱ $\text{lcm}(a_1, a_2, ..., a_n)$ را به صورت *کوچک‌ترین مضرب مشترک* (least common multiple) n عدد صحیح $a_1, a_2, ..., a_n$ تعریف می‌کنیم، یعنی، کوچک‌ترین عدد صحیح نامنفی که مضربی از هر یک از a_iها است. نشان دهید که چطور می‌توان با استفاده از عملیات ب.م.م (دو آرگومانی) به عنوان یک زیرروال، $\text{lcm}(a_1, a_2, ..., a_n)$ را به صورت بهینه محاسبه کرد.

۹-۲-۳۱ اثبات کنید که n_1، n_2، n_3، و n_4 دو به دو نسبت به هم اول هستند اگر و فقط اگر

$$\gcd(n_1 n_2, n_3 n_4) = \gcd(n_1 n_3, n_2 n_4) = 1$$

به صورت کلی‌تر، نشان دهید که $n_1, n_2, ..., n_k$ دو به دو نسبت به هم اول هستند اگر و فقط اگر مجموعه‌ای از $\lceil \lg k \rceil$ جفت از اعداد انتخاب شده از n_iها نسبت به هم اول باشند.

۳۱-۳ محاسبات پیمانه‌ای

به صورت غیررسمی، می‌توانیم محاسبات پیمانه‌ای را به صورت محاسبات ریاضی معمول بر روی اعداد صحیح در نظر بگیریم، با این تفاوت که اگر محاسبات به پیمانه‌ی n باشد، هر نتیجه‌ی x با عنصری از $\{0, 1, ..., n-1\}$ که به پیمانه‌ی n هم‌ارز با x است، جایگزین می‌شود (یعنی x با $x \bmod n$ جایگزین می‌شود). اگر فقط بر روی اعمال جمع، تفریق و ضرب تمرکز کنیم این مدل غیررسمی کافی است. مدل رسمی‌تر برای محاسبات پیمانه‌ای، که اکنون آن را ارائه می‌کنیم، بهتر است که در چارچوب نظریه‌ی گروه‌ها توصیف شود.

گروه‌های متناهی

یک گروه $(S, \oplus)$، یک مجموعه‌ی S به همراه یک عملیات دودویی $\oplus$ است که بر روی S تعریف شده است و خصوصیات زیر را ارضا می‌کند.

۱. *بستار*: برای تمام $a, b \in S$، داریم $a \oplus b \in S$.

۲. *عنصر همانی*: یک عنصر $e \in S$ به نام *عنصر همانی* گروه وجود دارد، به طوری که $e \oplus a = a \oplus e = a$ برای تمام $a \in S$.

۳. *شرکت‌پذیری*: برای هر $a, b, c \in S$، داریم $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.

۴. *معکوس‌ها*: برای هر $a \in S$، یک عنصر یکتای $b \in S$ به نام *معکوس* a وجود دارد به طوری که $a \oplus b = b \oplus a = e$.

به عنوان یک مثال، گروه آشنای $(\mathbb{Z},+)$ را که همان اعداد صحیح $\mathbb{Z}$ و عملیات جمع است در نظر بگیرید: ۰ عنصر همانی است، و معکوس a عبارت است از $-a$. اگر یک گروه $(S,\oplus)$ *قانون جابه‌جایی* (commutative law) $a\oplus b=b\oplus a$ را برای تمام $a,b\in S$ ارضا کند، آن گاه یک *گروه ابلی* (abelian group) است. اگر یک گروه $(S,\oplus)$، $|S|<\infty$ را ارضا کند، آن گاه یک *گروه متناهی* (finite group) است.

گروه‌های تعریف شده توسط جمع و ضرب پیمانه‌ای

با استفاده از جمع و ضرب به پیمانه‌ی n می‌توانیم دو گروه ابلی تعریف کنیم، که در آن n یک عدد صحیح مثبت است. این گروه‌ها بر پایه‌ی کلاس‌های هم‌ارزی اعداد صحیح به پیمانه‌ی n هستند، که در بخش ۳۱-۱ تعریف شده است.

برای تعریف یک گروه بر روی $\mathbb{Z}_n$، باید یک عملیات دودویی مناسب داشته باشیم، که با تعریف دوباره‌ی اعمال ضرب و جمع آن را به دست می‌آوریم. تعریف جمع و ضرب بر روی $\mathbb{Z}_n$ کار ساده‌ای است، چون کلاس هم‌ارزی دو عدد صحیح به صورت یکتا کلاس هم‌ارزی مجموع یا حاصل ضرب آن دو را تعیین می‌کند. یعنی، اگر $a\equiv a'\pmod n$ و $b\equiv b'\pmod n$، آن گاه

$$a+b\equiv a'+b'\pmod n,$$
$$ab\equiv a'b'\pmod n$$

بنابراین جمع و ضرب به پیمانه‌ی n را که با $+_n$ و $\cdot_n$ نشان می‌دهیم، به صورت زیر تعریف می‌کنیم:

$$\begin{aligned}[a]_n +_n [b]_n &= [a+b]_n\\ [a]_n \cdot_n [b]_n &= [ab]_n\end{aligned} \qquad (۳۱\text{-}۱۸)$$

(تفریق را می‌توان به طور مشابه بر روی $\mathbb{Z}_n$ به صورت $[a]_n -_n [b]_n = [a-b]_n$ تعریف کرد، ولی همان طور که خواهیم دید، تقسیم مقداری پیچیده‌تر است.) در این جا می‌توان فهمید که چرا هنگام انجام محاسبات بر روی $\mathbb{Z}_n$، استفاده از آخرین عنصر نامنفی هر کلاس هم‌ارزی به عنوان نماینده‌ی آن کلاس مناسب است. جمع، تفریق و ضرب به صورت معمول بر روی نماینده‌ها انجام می‌شود، با این تفاوت که هر نتیجه‌ی x با نماینده‌ی آن در کلاس مربوطه (یعنی $x \bmod y$) جایگزین می‌شود.

با استفاده از این تعریف برای جمع به پیمانه‌ی n، *گروه جمع به پیمانه‌ی n* را به صورت $(\mathbb{Z}_n,+_n)$ تعریف می‌کنیم. اندازه‌ی این گروه جمع به پیمانه‌ی n، $|\mathbb{Z}_n|=n$ است. شکل ۳۱-۲ (الف) جدول عملیات را برای گروه $(\mathbb{Z}_6,+_6)$ می‌دهد.

قضیه‌ی ۳۱-۱ سیستم $(\mathbb{Z}_n,+_n)$ یک گروه ابلی متناهی است.

اثبات تساوی (۳۱-۱۸) نشان می‌دهد که $(\mathbb{Z}_n,+_n)$ بسته است. شرکت‌پذیری و قابلیت جابه‌جایی برای $+_n$ از شرکت‌پذیری و قابلیت جابه‌جایی $+$ نتیجه می‌شود:

$+_6$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

(الف)

$\cdot_{15}$	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

(ب)

شکل ۳۱-۲ دو گروه متناهی. کلاس‌های هم‌ارزی با عضو نماینده‌ی آن‌ها نشان داده شده‌اند. (الف) گروه $(\mathbb{Z}_6, +_6)$. (ب) گروه $(\mathbb{Z}_{15}^*, \cdot_{15})$.

$$\begin{aligned}([a]_n +_n [b]_n) +_n [c]_n &= [a+b]_n +_n [c]_n \\ &= [(a+b)+c]_n \\ &= [a+(b+c)]_n \\ &= [a]_n +_n [b+c]_n \\ &= [a]_n +_n ([b]_n +_n [c]_n),\end{aligned}$$

$$\begin{aligned}[a]_n +_n [b]_n &= [a+b]_n \\ &= [b+a]_n \\ &= [b]_n +_n [a]_n\end{aligned}$$

عنصر همانی برای $(\mathbb{Z}_n, +_n)$، 0 است (همان $[0]_n$). معکوس (جمعی) یک عنصر a (یعنی معکوس $[a]_n$) عنصر $-a$ است (یعنی $[-a]_n$ یا $[n-a]_n$)، چون $[a-a]_n = [0]_n = [a]_n +_n [-a]_n$. ■

با استفاده از تعریف ضرب به پیمانه‌ی n، گروه ضربی به پیمانه‌ی n را به صورت $(\mathbb{Z}_n^*, \cdot_n)$ تعریف می‌کنیم. عناصر این گروه مجموعه‌ی $\mathbb{Z}_n^*$ از عناصر $\mathbb{Z}_n$ هستند که نسبت به n اول‌اند، به طوری که هر کدام یک معکوس یکتا به پیمانه‌ی n دارند:

$$\mathbb{Z}_n^* = \{[a]_n \in \mathbb{Z}_n : \gcd(a,n) = 1\}$$

برای این که ببینیم $\mathbb{Z}_n^*$ خوش‌تعریف است، توجه کنید که برای $0 \le a < n$ داریم $a \equiv (a+kn) \pmod n$ برای تمام اعداد صحیح k. بنابراین طبق تمرین ۳۱-۲-۳، $\gcd(a,n) = 1$ نتیجه می‌دهد $\gcd(a+kn,n) = 1$ برای تمام اعداد صحیح k. از آن جایی که $[a]_n = \{a+kn : k \in \mathbb{Z}\}$، مجموعه‌ی $\mathbb{Z}_n^*$ خوش‌تعریف است. یک مثال از چنین گروهی به صورت

$$\mathbb{Z}_{15}^* = \{1,2,4,7,8,11,13,14\}$$

است که در آن عملیات گروه، ضرب به پیمانه‌ی ۱۵ است. (در این جا عنصر $[a]_{15}$ را به صورت a نمایش می‌دهیم؛ مثلاً ۷ به معنی $[7]_{15}$ است.) شکل ۳۱-۲ (ب) گروه $(\mathbb{Z}^*_{15}, \cdot_{15})$ را نشان می‌دهد. به عنوان مثال در مجموعه‌ی $\mathbb{Z}^*_{15}$ داریم $8 \cdot 11 \equiv 13 \pmod{15}$. عنصر همانی برای این گروه ۱ است.

قضیه‌ی ۳۱-۱۳ سیستم $(\mathbb{Z}^*_n, \cdot_n)$ یک گروه ابلی متناهی است.

اثبات قضیه‌ی ۳۱-۶ ایجاب می‌کند که $(\mathbb{Z}^*_n, \cdot_n)$ بسته باشد. شرکت‌پذیری و قابلیت جابه‌جایی را می‌توانیم به همان صورتی که در قضیه‌ی ۳۱-۱۲ برای $+_n$ اثبات کردیم، برای $\cdot_n$ هم اثبات کنیم. عنصر همانی $[1]_n$ است. برای این که نشان دهیم معکوس وجود دارد، فرض کنید a یک عنصر از $\mathbb{Z}^*_n$ باشد، و (d, x, y) گروه مربوط به EXTENDED-EUCLID(a, n) در این صورت $d = 1$، چرا که $a \in \mathbb{Z}^*_n$، و

$$ax + ny = 1 \tag{۳۱-۱۹}$$

یا به طور معادل،

$$ax \equiv 1 \pmod{n}$$

بنابراین، $[x]_n$ یک معکوس ضربی $[a]_n$ به پیمانه‌ی n است. به علاوه ادعا می‌کنیم که $[x]_n \in \mathbb{Z}^*_n$. برای این که ببینیم چرا، تساوی (۳۱-۱۹) نشان می‌دهد که کوچک‌ترین کوچک‌ترین ترکیب خطی مثبت از x و n باید ۱ باشد. بنابراین قضیه‌ی ۳۱-۲ ایجاب می‌کند که $\gcd(x, n) = 1$. اثبات این که معکوس‌ها یکتا هستند به نتیجه‌ی ۳۱-۲۶ موکول می‌شود. ■

به عنوان یک مثال از محاسبه‌ی معکوس‌های ضربی، فرض کنید که $a = 5$ و $n = 11$. آن گاه EXTENDED-EUCLID(a, n) سه‌تایی $(d, x, y) = (1, -2, 1)$ را بازمی‌گرداند، به طوری که $11 \cdot 1 + (-2) \cdot 5 = 1$. بنابراین $[-2]_{11}$ (همان $9 \bmod 11$) یک معکوس ضربی از $[5]_{11}$ است.

در ادامه‌ی این فصل، وقتی با گروه‌های $(\mathbb{Z}_n, +_n)$ و $(\mathbb{Z}^*_n, \cdot_n)$ کار می‌کنیم، برای سادگی کلاس‌های هم‌ارزی را با عنصر نماینده‌ی آن‌ها و اعمال $+_n$ و $\cdot_n$ را با نمادهای ریاضی $+$ و $\cdot$ (و یا نماد تهی برای ضرب، به طوری که $ab = a \cdot b$) نشان می‌دهیم. همچنین، هم‌ارزی به پیمانه‌ی n ممکن است به عنوان تساوی در $\mathbb{Z}_n$ تعبیر شود. مثلاً دو عبارت زیر معادل هستند:

$$ax \equiv b \pmod{n}$$
$$[a]_n \cdot_n [x]_n = [b]_n$$

باز هم برای سادگی، بعضی مواقع، وقتی عملیات مورد نظر از روی متن قابل تشخیص است، گروه $(S, \oplus)$ را به صورت S نمایش می‌دهیم. بنابراین ممکن است گروه‌های $(\mathbb{Z}_n, +_n)$ و $(\mathbb{Z}^*_n, \cdot_n)$ را به صورت $\mathbb{Z}_n$ و $\mathbb{Z}^*_n$ نمایش دهیم.

معکوس (ضربی) یک عنصر a به صورت $(a^{-1} \bmod n)$ نمایش داده می‌شود. تقسیم در $\mathbb{Z}^*_n$ به صورت تساوی $a/b \equiv ab^{-1} \pmod{n}$ تعریف می‌شود. مثلاً در $\mathbb{Z}^*_{15}$ داریم $7^{-1} \equiv 13 \pmod{15}$، چرا

که $۷۰۱۳ \equiv ۹۱ \equiv ۱ \pmod{۱۵}$، بنابراین $۴۰۱۳ \equiv ۷ \pmod{۱۵}$ $۴/۷$.

اندازه‌ی $\mathbb{Z}_n^*$ با $\varphi(n)$ نشان داده می‌شود. این تابع، که به *تابع فی اویلر* (Euler's phi function) معروف است، تساوی زیر را ارضا می‌کند:

$$\phi(n) = n\prod_{p|n}\left(۱-\frac{۱}{p}\right) \qquad (۳۱-۲۰)$$

که در آن p بر روی تمام اعداد اولی که n را می‌شمارند (شامل خود n، اگر اول باشد) حرکت می‌کند. این فرمول را در این جا اثبات نخواهیم کرد. به صورت شهودی، با لیستی از n باقی‌مانده‌ی $\{۰, ۱, ..., n-۱\}$ آغاز می‌کنیم، و سپس برای هر عدد اول p که n را می‌شمارد، تمام مضارب p در لیست را خط می‌زنیم. برای مثال، از آن جایی که شمارنده‌های اول ۴۵، ۳ و ۵ هستند،

$$\begin{aligned}\varphi(۴۵) &= ۴۵\left(۱-\frac{۱}{۳}\right)\left(۱-\frac{۱}{۵}\right)\\ &= ۴۵\left(\frac{۲}{۳}\right)\left(\frac{۴}{۵}\right)\\ &= ۲۴\end{aligned}$$

اگر p اول باشد، آن گاه $\mathbb{Z}_p^* = \{۱, ۲, ..., p-۱\}$، و

$$\begin{aligned}\phi(p) &= p\left(۱-\frac{۱}{p}\right) \qquad (۳۱-۲۱)\\ &= p-۱\end{aligned}$$

اگر n مرکب باشد آن گاه $\varphi(n) < n-۱$، ولی می‌توان نشان داد که

$$\phi(n) > \frac{n}{e^{\gamma}\ln\ln n + \frac{۳}{\ln\ln n}} \qquad (۳۱-۲۲)$$

برای $n \geq ۳$، که در آن $\gamma = ۰/۵۷۷۲۱۵۶۶۴۹...$ *ثابت اویلر* (Euler's constant) است. یک کران پایین ساده‌تر (ولی نه چندان نزدیک) برای $n > ۵$ عبارت است از

$$\phi(n) > \frac{n}{۶\ln\ln n} \qquad (۳۱-۲۳)$$

کران پایین (۳۱-۲۲) عملاً بهترین کران ممکن است، چرا که

$$\liminf_{n\to\infty}\frac{\phi(n)}{n/\ln\ln n} = e^{-\gamma} \qquad (۳۱-۲۴)$$

زیرگروه‌ها

اگر $(S, \oplus)$ یک گروه باشد، $S' \subseteq S$، و $(S', \oplus)$ هم یک گروه باشد، آن گاه می‌گوییم $(S', \oplus)$ یک زیرگروه از $(S, \oplus)$ است. به عنوان مثال اعداد زوج یک زیرگروه از اعداد صحیح تحت عملیات جمع هستند. قضیه‌ی زیر یک ابزار مفید برای تشخیص زیرگروه‌ها به دست می‌دهد.

قضیه‌ی ۳۱-۱۳ (زیرگروه‌بودن یک زیرمجموعه‌ی ناتهی)
اگر $(S, \oplus)$ یک گروه متناهی باشد، و S' یک زیرمجموعه‌ی ناتهی از S به طوری که $a \oplus b \in S'$ برای تمام $a, b \in S'$، آن گاه $(S', \oplus)$ یک زیرگروه از $(S, \oplus)$ است.

اثبات اثبات به عنوان تمرین ۳۱-۳-۲ واگذار شده است.

به عنوان مثال، مجموعه‌ی $\{0, 2, 4, 6\}$ یک زیرگروه از $\mathbb{Z}_8$ را تشکیل می‌دهد، چرا که ناتهی، و تحت عملیات + بسته است (یعنی در واقع تحت عملیات $+_8$ بسته است).

قضیه‌ی زیر یک محدودیت بسیار مفید بر روی اندازه‌ی یک زیرگروه به دست می‌دهد؛ در این جا از اثبات صرف نظر می‌کنیم.

قضیه‌ی لاگرانژ ۳۱-۱۵ (قضیه‌ی لاگرانژ)
اگر $(S, \oplus)$ یک گروه متناهی باشد، و $(S', \oplus)$ یک زیرگروه از آن، آن گاه $|S'|$ یک مقسوم‌علیه از $|S|$ است.

می‌گوییم یک زیرگروه S' از گروه S، یک زیرگروه *اکید* (proper) است اگر داشته باشیم $S' \neq S$. از نتیجه‌ی زیر در تحلیل رویه‌ی تست اول بودن به روش میلر–رابین در بخش ۳۱-۸ استفاده خواهد شد.

نتیجه‌ی ۳۱-۱۶ اگر S' یک زیرگروه اکید از یک گروه متناهی باشد، آن گاه $|S'| \leq |S|/2$.

زیرگروه‌های ساخته شده توسط یک عنصر

قضیه‌ی ۳۱-۱۴ یک روش جذاب برای ساختن یک زیرگروه از یک گروه متناهی $(S, \oplus)$ به دست می‌دهد: یک عنصر a را انتخاب کنید و تمام عناصری را که می‌توان با انجام عملیات گروه بر روی a ساخت، در گروه قرار دهید. به طور خاص، برای $k \geq 1$، $a^{(k)}$ را به صورت زیر تعریف می‌کنیم:

$$\bigoplus_{i=1}^{k} a = \underbrace{a \oplus a \oplus \ldots \oplus a}_{k}$$

مثلاً اگر داشته باشیم $a = 2$ در گروه $\mathbb{Z}_6$، دنباله‌ی $a^{(1)}, a^{(2)}, a^{(3)}, \ldots$ عبارت است از $2, 4, 0, 2, 4, 0, 2, 4, 0, \ldots$.

در گروه $\mathbb{Z}_n$، داریم $a^{(k)} = ka \bmod n$، و در گروه $\mathbb{Z}_n^*$، داریم $a^{(k)} = a^k \bmod n$. *زیرگروه ساخته شده توسط* a، که آن را با $\langle a \rangle$ یا $(\langle a \rangle, \oplus)$ نمایش می‌دهیم، به صورت زیر تعریف می‌شود:

$$\langle a\rangle, \{a^{(k)} : k \geq 1\}$$

می‌گوییم a زیرگروه $\langle a\rangle$ را *می‌سازد*، و یا a یک *سازنده‌ی* $\langle a\rangle$ است. از آن‌جایی که S متناهی است، $\langle a\rangle$ یک زیرمجموعه‌ی متناهی از S است، که ممکن است شامل تمام عناصر S باشد. چون خاصیت شرکت‌پذیری $\oplus$ ایجاب می‌کند که

$$a^{(i)} \oplus a^{(j)} = a^{(i+j)}$$

$\langle a\rangle$ بسته است، و بنابراین، طبق قضیه‌ی ۳۱-۱۴، $\langle a\rangle$ یک زیرگروه از S است. مثلاً در $\mathbb{Z}_۶$ داریم

$$\langle ۰\rangle = \{۰\},$$
$$\langle ۱\rangle = \{۰, ۱, ۲, ۳, ۴, ۵\},$$
$$\langle ۲\rangle = \{۰, ۲, ۴\}$$

به طور مشابه در $\mathbb{Z}_۷^*$ داریم

$$\langle ۱\rangle = \{۱\},$$
$$\langle ۲\rangle = \{۱, ۲, ۴\},$$
$$\langle ۳\rangle = \{۱, ۲, ۳, ۴, ۵, ۶\}$$

تعریف می‌کنیم رتبه‌ی a (در گروه S)، که آن را با $\mathrm{ord}(a)$ نمایش می‌دهیم، برابر است با کوچک‌ترین عدد صحیح مثبت t به طوری که $a^{(t)} = e$.

قضیه‌ی ۱۶-۳۱ برای هر گروه متناهی $(S, \oplus)$ و هر $a \in S$، رتبه‌ی یک عنصر برابر است با اندازه‌ی زیرگروهی که می‌سازد، با به عبارت دیگر $\mathrm{ord}(a) = |\langle a\rangle|$.

اثبات فرض کنید $t = \mathrm{ord}(a)$. چون $a^{(t)} = e$ و $a^{(t+k)} = a^{(t)} \oplus a^{(k)} = a^{(k)}$ برای تمام $k \geq ۱$، اگر $i \geq t$، آن گاه $a^{(i)} \geq a^{(j)}$ برای یک $j < i$. بنابراین بعد از $a^{(t)}$ هیچ عنصر جدیدی دیده نمی‌شود، و از این رو $\langle a\rangle = \{a^{(۱)}, a^{(۲)}, \ldots, a^{(t)}\}$ و $|\langle a\rangle| \leq t$. برای این که نشان دهیم $|\langle a\rangle| \geq t$، نشان می‌دهیم که تمام عناصر دنباله‌ی $a^{(۱)}, a^{(۲)}, \ldots, a^{(t)}$ یکتا هستند. طبق برهان خلف فرض کنید که $a^{(i)} = a^{(j)}$ برای i و j که در $۱ \leq i < j \leq t$ صدق می‌کنند. آن گاه $a^{(i+k)} = a^{(j+k)}$ برای $k \geq ۰$. ولی این ایجاب می‌کند که $a^{(i+(t-j))} = a^{(j+(t-j))} = e$، که تناقض است، چرا که $i + (t-j) < t$ ولی t کوچک‌ترین عدد مثبت است به طوری که $a^{(t)} = e$. بنابراین هر عنصر در دنباله‌ی $a^{(۱)}, a^{(۲)}, \ldots, a^{(t)}$ یکتا است، و $|\langle a\rangle| \geq t$. نتیجه می‌گیریم که $\mathrm{ord}(a) = |\langle a\rangle|$.

نتیجه‌ی ۱۸-۳۱ دنباله‌ی $a^{(۱)}, a^{(۲)}, \ldots$ متناوب است، با دوره‌ی تناوب $t = \mathrm{ord}(a)$؛ یعنی $a^{(i)} = a^{(j)}$ اگر و تنها اگر $i \equiv j \pmod t$.

طبق نتیجه‌ی بالا، مناسب است که تعریف کنیم $a^{(0)} = e$ و $a^{(i)} = a^{(i \bmod t)}$، که در آن $t = \text{ord}(a)$، برای تمام اعداد صحیح i.

نتیجه‌ی ۳۱-۱۹ اگر $(S, \oplus)$ یک گروه متناهی باشد، با عنصر همانی e، آن گاه برای تمام $a \in S$،

$$a^{(|S|)} = e$$

اثبات قضیه‌ی لاگرانژ (قضیه‌ی ۳۱-۱۵) ایجاب می‌کند که $\text{ord}(a) \mid |S|$، و بنابراین $|S| \equiv 0 \pmod t$، که در آن $t = \text{ord}(a)$. بنابراین $a^{(|S|)} = a^{(0)} = e$. ■

تمرین‌ها

۳۱-۱-۱ جدول اعمال گروه را برای گروه‌های $(\mathbb{Z}_4, +_4)$ و $(\mathbb{Z}_5^*, \cdot_5)$ بکشید. نشان دهید که این گروه‌ها هم‌ریخت هستند، بدین صورت که یک تناظر یک به یک Σ بین عناصر آن بیابید به طوری که $a + b \equiv c \pmod 4$ اگر و فقط اگر $\Sigma(a) \cdot \Sigma(b) \equiv \Sigma(c) \pmod 5$.

۳۱-۱-۲ تمام زیرگروه‌های $\mathbb{Z}_9$ و $\mathbb{Z}_{13}^*$ را لیست کنید.

۳۱-۱-۳ قضیه‌ی ۳۱-۱۴ را اثبات کنید.

۳۱-۱-۴ نشان دهید که اگر p اول باشد و e یک عدد صحیح مثبت، آن گاه

$$\varphi(p^e) = p^{e-1}(p-1)$$

۳۱-۱-۵ نشان دهید که برای هر $n > 1$ و برای هر $a \in \mathbb{Z}_n^*$، تابع $f_a : \mathbb{Z}_n^* \to \mathbb{Z}_n^*$ که به صورت $f_a(x) = ax \bmod n$ تعریف می‌شود، یک جایگشت بر روی $\mathbb{Z}_n^*$ است.

۳۱-۴ حل معادلات پیمانه‌ای خطی

اکنون مسئله‌ی یافتن جواب‌های معادله‌ی

$$ax \equiv b \pmod n \qquad (۳۱-۲۵)$$

را در نظر می‌گیریم، که در آن $a > 0$ و $n > 0$. کاربردهای زیادی برای این مسئله وجود دارد؛ مثلاً، از آن به عنوان بخشی از رویه‌ی یافتن کلیدها در سیستم رمزنگاری کلید عمومی RSA در بخش ۳۱-۷ استفاده می‌کنیم. فرض می‌کنیم که a، b، و n داده شده‌اند، و می‌خواهیم تمام مقادیر x به پیمانه‌ی n را بیابیم که تساوی (۳۱-۲۵) را ارضا می‌کنند. ممکن است تعداد این جواب‌ها صفر، یک، و یا بیشتر باشد.

فرض کنید $\langle a \rangle$ نشان‌دهنده‌ی زیرگروه $\mathbb{Z}_n$ ساخته شده توسط a باشد. چون $\langle a \rangle = \{a^{(x)} : x > 0\} = \{ax \bmod n : x > 0\}$، تساوی (۳۱-۲۵) یک جواب دارد اگر و فقط اگر

$b \in \langle a \rangle$. قضیه‌ی لاگرانژ (قضیه‌ی ۳۱-۱۵) می‌گوید که $|\langle a \rangle|$ باید n را بشمارد. قضیه‌ی زیر توصیف دقیق $\langle a \rangle$ را به ما می‌دهد.

قضیه‌ی ۳۱-۲۰

برای هر دو عدد صحیح a و n، اگر $d = \gcd(a,n)$، آن گاه

$$\langle a \rangle = \langle d \rangle = \{0, d, 2d, \ldots, ((n/d)-1)d\} \quad (۳۱\text{-}۲۶)$$

در $\mathbb{Z}_n$، و بنابراین

$$|\langle a \rangle| = n/d$$

اثبات با نشان دادن $d \in \langle a \rangle$ شروع می‌کنیم. به خاطر بیاورید که EXTENDED-EUCLID (a,n) اعداد صحیح x' و y' را می‌سازد، به طوری که $ax' + ny' = d$. بنابراین $ax' \equiv d \pmod{n}$، و بنابراین $d \in \langle a \rangle$.

چون $d \in \langle a \rangle$، نتیجه می‌شود که هر مضربی از d متعلق به $\langle a \rangle$ است، زیرا هر مضربی از یک مضرب از a، خود مضربی از a است. بنابراین $\langle a \rangle$ حاوی تمام عناصر $\{0, d, 2d, \ldots, ((n/d)-1)d\}$ است. یعنی $\langle d \rangle \subseteq \langle a \rangle$.

اکنون نشان می‌دهیم که $\langle a \rangle \subseteq \langle d \rangle$. اگر $m \in \langle a \rangle$، آن گاه $m = ax \bmod n$ برای یک عدد صحیح x، و بنابراین $m = ax + ny$ برای یک عدد صحیح y. از طرفی، $d \mid a$ و $d \mid n$، و بنابراین طبق تساوی (۳۱-۴) داریم $d \mid m$. از این رو $m \in \langle d \rangle$.

با ترکیب این نتایج داریم $\langle a \rangle = \langle d \rangle$. برای این که ببینیم $|\langle a \rangle| = n/d$، مشاهده می‌کنیم که دقیقاً n/d مضرب d بین و شامل ۰ و $n-1$ وجود دارد. ▒

نتیجه‌ی ۳۱-۲۱

تساوی $ax \equiv b \pmod{n}$ برای مجهول x جواب دارد اگر و فقط اگر $\gcd(a,n) \mid b$.

اثبات تساوی $ax \equiv b \pmod{n}$ قابل حل خواهد بود اگر و فقط اگر $[b] \in \langle a \rangle$، که طبق قضیه‌ی ۳۱-۲۰ معادل است با

$$(b \bmod n) \in \{0, d, 2d, \ldots, ((n/d)-1)d\}$$

اگر $n > b \geq 0$، آن گاه $b \in \langle b \rangle$ اگر و فقط اگر $d \mid b$، چرا که اعداد $\langle a \rangle$ دقیقاً مضارب d هستند. اگر $b < 0$ یا $b \geq n$، آن گاه نتیجه ثابت است زیرا $d \mid b$ اگر و فقط اگر $d \mid (b \bmod n)$، چرا که b و $b \bmod n$ به اندازه‌ی ضریبی از n با هم اختلاف دارند، که خود ضریبی از d است. ▒

نتیجه‌ی ۳۱-۲۲

تساوی $ax \equiv b \pmod{n}$ یا d جواب متمایز به پیمانه‌ی n دارد، که در آن $d = \gcd(a,n)$، یا هیچ جوابی ندارد.

اثبات اگر $ax \equiv b \pmod n$ جواب داشته باشد، آن گاه $b \in \langle a \rangle$. طبق قضیه‌ی ۳۱-۱۷ داریم $|\langle a \rangle| = \text{ord}(a)$، و بنابراین نتیجه‌ی ۳۱-۱۸ و قضیه‌ی ۳۱-۲۰ ایجاب می‌کنند که دنباله‌ی $ai \bmod n$، برای $i = 0, 1, \ldots, n-1$ متناوب است، با دوره‌ی تناوب $|\langle a \rangle| = n/d$. اگر $b \in \langle a \rangle$، آن گاه b در دنباله‌ی $ai \bmod n$ برای $i = 0, 1, \ldots, n-1$، دقیقاً d بار ظاهر می‌شود، چرا که با افزایش i از ۰ تا $n-1$، بلوک مقادیر $\langle a \rangle$ با طول n/d دقیقاً d بار تکرار می‌شود. اندیس‌های x از d مکانی که برای آن‌ها $ax \bmod n = d$، جواب‌های معادله‌ی $ax \equiv b \pmod n$ هستند. ■

قضیه‌ی ۳۱-۲۱

فرض کنید $d = \gcd(a, n)$، و فرض کنید که $d = ax' + ny'$ برای اعداد صحیح x' و y' (به عنوان مثال اعدادی که EXTENDED-EUCLID محاسبه می‌کند). اگر $d \mid b$، آن گاه یکی از جواب‌های تساوی $ax \equiv b \pmod n$، مقدار x_0 است، که

$$x_0 = x'(b/d) \bmod n$$

اثبات داریم

$$\begin{aligned} ax_0 &\equiv ax'(b/d) \pmod n \\ &\equiv d(b/d) \pmod n \quad (\text{چون } ax' \equiv d \pmod n) \\ &\equiv b \pmod n \end{aligned}$$

و بنابراین x_0 یک جواب برای $ax \equiv b \pmod n$ است. ■

قضیه‌ی ۳۱-۲۲

فرض کنید که معادله‌ی $ax \equiv b \pmod n$ قابل حل است (یعنی $d \mid b$، که در آن $d = \gcd(a, n)$) و x_0 یک جواب دلخواه برای این معادله است. آن گاه این معادله دقیقاً d جواب متمایز به پیمانه‌ی n دارد، که به صورت $x_i = x_0 + i(n/d)$ برای $i = 0, 1, \ldots, d-1$ هستند.

اثبات چون $n/d > 0$ و $0 \le i(n/d) < n$ برای $i = 0, 1, \ldots, d-1$، مقادیر $x_0, x_1, \ldots, x_{d-1}$ همگی به پیمانه‌ی n متمایز هستند. چون x_0 یک جواب برای $ax \equiv b \pmod n$ است، داریم $ax_0 \bmod n = b$. بنابراین برای $i = 0, 1, \ldots, d-1$ داریم

$$\begin{aligned} ax_i \bmod n &= a(x_0 + in/d) \bmod n \\ &= (ax_0 + ain/d) \bmod n \\ &= ax_0 \bmod n \quad (\text{چون } d \mid a \text{ نتیجه می‌دهدکه } ain/d \text{ ضریبی از } n \text{ است}) \\ &= b \pmod n \end{aligned}$$

و بنابراین $ax_i \equiv b \pmod n$، که نتیجه می‌دهد x_i هم یک جواب است. طبق نتیجه‌ی ۳۱-۲۲ تساوی $ax \equiv b \pmod n$ دقیقاً d جواب دارد، و بنابراین $x_0, x_1, \ldots, x_{d-1}$ باید تنها جواب‌های موجود باشند. ■

تا این جا مقدمات ریاضی مورد نیاز را برای حل معادله‌ی $ax \equiv b \pmod{n}$ آماده کرده‌ایم؛ الگوریتم زیر تمام جواب‌های این معادله را در خروجی چاپ می‌کند. ورودی‌های a و n اعداد صحیح مثبت دلخواه هستند، و b یک عدد صحیح دلخواه.

```
MODULAR-LINEAR-EQUATION-SOLVER(a, b, n)
1  (d, x', y') = EXTENDED-EUCLID(a, n)
2  if d | b
3      x_0 = x'(b/d) mod n
4      for i = 0 to d - 1
5          print (x_0 + i(n/d)) mod n
6  else print "no solutions"
```

به عنوان یک مثال از کارکرد این رویه، تساوی $14x \equiv 30 \pmod{100}$ را در نظر بگیرید (در این جا، $a = 14$، $b = 30$، و $n = 100$). با فراخوانی EXTENDED-EUCLID در خط ۱ به دست می‌آوریم $(d, x, y) = (2, -7, 1)$. چون $2 \mid 30$، خط ۲ و خط ۳-۵ اجرا می‌شوند. در خط ۳، $x_0 = (-7)(15) \bmod 100 = 95$ را محاسبه می‌کنیم. حلقه‌ی خط ۴-۵ دو جواب ۹۵ و ۴۵ را چاپ می‌کنند.

رویه‌ی MODULAR-LINEAR-EQUATION-SOLVER به صورت زیر کار می‌کند. خط ۱ مقدار $d = \gcd(a, n)$ را محاسبه می‌کند، به علاوه‌ی مقادیر x' و y' به طوری که $d = ax' + by'$، که نتیجه می‌دهد که x' یک جواب برای معادله‌ی $ax' \equiv d \pmod{n}$ است. اگر d، b را نشمارد، آن گاه طبق نتیجه‌ی ۳۱-۲۱ تساوی $ax \equiv b \pmod{n}$ هیچ جوابی ندارد. خط ۲ چک می‌کند که آیا $d \mid b$؛ اگر خیر، خط ۶ گزارش می‌کند که $ax \equiv b \pmod{n}$ هیچ جوابی ندارد. در غیر این صورت خط ۳ یک جواب x_0 برای $ax \equiv b \pmod{n}$ محاسبه می‌کند، مطابق با قضیه‌ی ۳۱-۲۳. با داشتن یک جواب، قضیه‌ی ۳۱-۲۴ می‌گوید که $d - 1$ جواب دیگر را می‌توان با اضافه کردن مضارب (n/d) به پیمانه‌ی n به دست آورد. حلقه‌ی for خطوط ۴-۵ تمام d جواب را چاپ می‌کنند، که از x_0 آغاز شده و هر یک با بعدی n/d به پیمانه‌ی n فاصله دارد.

MODULAR-LINEAR-EQUATION-SOLVER به تعداد $O(\lg n + \gcd(a, n))$ عملیات ریاضی انجام می‌دهد، چرا که تعداد اعمال ریاضی در EXTENDED-EUCLID برابر است با $O(\lg n)$، و هر تکرار حلقه‌ی for خطوط ۴-۵ تعداد ثابتی عملیات ریاضی انجام می‌دهد.

نتیجه‌های زیر از قضیه‌ی ۳۱-۲۴ حالت‌های خاص و مفیدی را بررسی می‌کنند.

نتیجه‌ی ۳۱-۲۵ برای هر $n > 1$، اگر $\gcd(a, n) = 1$، آن گاه تساوی $ax \equiv b \pmod{n}$ یک جواب یکتا به پیمانه‌ی n دارد.

اگر $b = 1$، که یک حالت معمول و بسیار مفید است، xای که به دنبال آن می‌گردیم یک معکوس ضربی a به پیمانه‌ی n است.

نتیجه‌ی ۳۱-۲۶ برای هر $n > 1$، اگر $\gcd(a,n) = 1$ آن گاه تساوی $ax \equiv 1 \pmod{n}$ یک جواب یکتا به پیمانه‌ی n دارد، و در غیر این صورت هیچ جوابی ندارد.

نتیجه‌ی ۳۱-۲۶ به ما اجازه می‌دهد که از نماد $(a^{-1} \bmod n)$ برای اشاره به معکوس ضربی a به پیمانه‌ی n استفاده کنیم، که در آن a و n نسبت به هم اول هستند. اگر $\gcd(a,n) = 1$، آن گاه یک جواب برای معادله‌ی $ax \equiv 1 \pmod{n}$ عدد صحیح x است که توسط EXTENDED-EUCLID بازگردانده شده است، چرا که تساوی

$$\gcd(a,n) = 1 = ax + ny$$

نتیجه می‌دهد که $ax \equiv 1 \pmod{n}$. بنابراین می‌توانیم با استفاده از EXTENDED-EUCLID، $(a^{-1} \bmod n)$ را به صورت کارآمد محاسبه کنیم.

تمرین‌ها

۳۱-۴-۱ تمام جواب‌های معادله‌ی $35x \equiv 10 \pmod{50}$ را بیابید.

۳۱-۴-۲ اثبات کنید که تساوی $ax \equiv ay \pmod{n}$ نتیجه می‌دهد که $x \equiv y \pmod{n}$ در صورتی که $\gcd(a,n) = 1$. با استفاده از یک مثال نقض که در آن $\gcd(a,n) > 1$، نشان دهید که شرط $\gcd(a,n) = 1$ ضروری است.

۳۱-۴-۳ تغییر زیر را برای خط ۳ رویه‌ی MODULAR-LINEAR-EQUATION-SOLVER در نظر بگیرید:

3 $x_0 = x'(b/d) \bmod (n/d)$

آیا این رویه باز هم کار می‌کند؟ توضیح دهید که چرا.

★ ۳۱-۴-۴ فرض کنید $f(x) \equiv f_0 + f_1 x + \ldots + f_t x^t \pmod{p}$ یک چندجمله‌ای از درجه‌ی t باشد، که در آن ضرایب f_i از مجموعه‌ی $\mathbb{Z}_p$ هستند، و p یک عدد اول است. می‌گوییم $a \in \mathbb{Z}_p$ یک صفر f است اگر $f(a) \equiv 0 \pmod{p}$. اثبات کنید که اگر a یک صفر f باشد، آن گاه $f(x) \equiv (x-a)g(x) \pmod{p}$ برای یک چندجمله‌ای $g(x)$ از درجه‌ی $t-1$. با استقرا بر روی t اثبات کنید که یک چندجمله‌ای $f(x)$ از درجه‌ی t حداکثر می‌تواند t صفر متمایز به پیمانه‌ی یک عدد اول p داشته باشد.

۵-۳۱ قضیه‌ی باقی‌مانده‌ی چینی

حدود سال ۱۰۰ قبل از میلاد، ریاضی‌دان چینی، سان تسو (Sun-Tsu) مسئله‌ی یافتن اعداد صحیحی که باقی‌مانده‌ی آن‌ها بر ۳، ۵، و ۷ به ترتیب ۲، ۳، و ۲ است را حل کرد. یک جواب بدین این مسئله $x = ۲۳$ است؛ تمام جواب‌ها به شکل $۱۰۵k+۲۳$ هستند، برای یک عدد صحیح دلخواه k. «قضیه‌ی باقی‌مانده‌ی چینی» یک تناظر بین یک سیستم از تساوی‌ها به پیمانه‌ی مجموعه‌ای از پیمانه‌های دو به دو نسبت به هم اول (مانند ۳، ۵، و ۷) و یک تساوی به پیمانه‌ی ضرب آن‌ها (در این جا، ۱۰۵) برقرار می‌کند.

قضیه‌ی باقی‌مانده‌ی چینی دو استفاده‌ی اصلی دارد. فرض کنید عدد صحیح n به صورت $n = n_۱n_۲...n_k$ تجزیه شده باشد، که در آن n_iها دو به دو نسبت به هم اول هستند. اول، قضیه‌ی باقی‌مانده‌ی چینی یک «قضیه‌ی ساختاری» توصیفی است که ساختار $\mathbb{Z}_n$ را مشابه ضرب کارتزین $\mathbb{Z}_{n_۱} \times \mathbb{Z}_{n_۲} \times ... \times \mathbb{Z}_{n_k}$ با جمع و ضرب به پیمانه‌ی n_i در عنصر iام توصیف می‌کند. دوم، معمولاً می‌توان با استفاده از این توصیف به الگوریتم‌های کارآمد رسید، چرا که کار با هر یک از سیستم‌های $\mathbb{Z}_{n_i}$ (بر حسب اعمال بیتی) می‌تواند بهینه‌تر از کار به پیمانه‌ی n باشد.

فرض کنید $n = n_۱ n_۲ ... n_k$، که در آن n_iها دو به دو نسبت به هم اول هستند. تناظر زیر را در نظر بگیرید:

$$a \leftrightarrow (a_۱, a_۲, ..., a_k) \qquad (۲۷-۳۱)$$

که در آن $a \in \mathbb{Z}_n$، $a_i \in \mathbb{Z}_{n_i}$، و

$$a_i = a \bmod n_i$$

برای $i = ۱, ۲, ..., k$. آن گاه نگاشت (۲۷-۳۱) یک تناظر یک به یک و پوشا بین $\mathbb{Z}_n$ و ضرب کارتزین $\mathbb{Z}_{n_۱} \times \mathbb{Z}_{n_۲} \times ... \times \mathbb{Z}_{n_k}$ است. اعمال انجام شده بر روی عناصر $\mathbb{Z}_n$ را می‌توان به صورت مشابه بر روی k تایی متناظر آن انجام داد، بدین صورت که اعمال را مستقلا در هر مکان در سیستم مختصاتی مناسب انجام می‌دهیم. یعنی اگر

$$a \leftrightarrow (a_۱, a_۲, ..., a_k),$$
$$b \leftrightarrow (b_۱, b_۲, ..., b_k)$$

آن گاه

$$(a+b) \bmod n \leftrightarrow ((a_۱+b_۱) \bmod n_۱, ..., (a_k+b_k) \bmod n_k), \qquad (۲۸-۳۱)$$
$$(a-b) \bmod n \leftrightarrow ((a_۱-b_۱) \bmod n_۱, ..., (a_k+b_k) \bmod n_k), \qquad (۲۹-۳۱)$$
$$(ab) \bmod n \leftrightarrow (a_۱b_۱ \bmod n_۱, ..., a_kb_k \bmod n_k) \qquad (۳۰-۳۱)$$

اثبات تبدیل بین دو نمایش نسبتاً سرراست است. رفتن از a به $(a_1, a_2, ..., a_k)$ کاملاً ساده است و فقط به k تقسیم نیاز دارد. محاسبه‌ی a از ورودی‌های $(a_1, a_2, ..., a_k)$ مقداری پیچیده‌تر است، و به صورت زیر انجام می‌شود. با تعریف $m_i = n/n_i$ برای $i = 1, 2, ..., k$ شروع می‌کنیم؛ بنابراین m_i برابر است با حاصل ضرب تمام n_jها غیر از n_i: $m_i = n_1 n_2 ... n_{i-1} n_{i+1} ... n_k$. سپس تعریف می‌کنیم

$$c_i = m_i (m_i^{-1} \bmod n_i) \qquad (۳۱-۳۱)$$

برای $i = 1, 2, ..., k$. تساوی (۳۱-۳۱) همیشه خوش‌تعریف است: چون m_i و n_i نسبت به هم اول هستند (طبق قضیه‌ی ۳۱-۶)، نتیجه‌ی ۳۱-۲۶ تضمین می‌کند که $(m_i^{-1} \bmod n_i)$ وجود دارد. نهایتاً می‌توانیم a را به صورت تابعی از $a_1, a_2, ..., a_k$ تعریف کنیم، به صورت زیر:

$$a \equiv (a_1 c_1 + a_2 c_2 + ... + a_k c_k) \pmod n \qquad (۳۱-۳۲)$$

اکنون نشان می‌دهیم که تساوی (۳۱-۳۲) تضمین می‌کند که $a \equiv a_i \pmod{n_i}$ برای $i = 1, 2, ..., k$. توجه کنید که اگر $j \neq i$، آن گاه $m_j \equiv 0 \pmod n$، که نتیجه می‌دهد که $c_j \equiv m_j \equiv 0 \pmod n$. همچنین توجه کنید که $c_i \equiv 1 \pmod{n_i}$، از تساوی (۳۱-۳۱). پس باید تناظر مفید و راضی کننده‌ی

$$c_i \leftrightarrow (0, 0, ..., 0, 1, 0, ..., 0)$$

را داشته باشیم، برداری که در تمام محورها مقدار آن 0 است، غیر از محور iام، که در آن جا مقدار آن 1 است؛ بنابراین c_i به طریقی یک مبنا برای نمایش است. بنابراین برای هر i داریم

$$\begin{aligned} a &\equiv a_i c_i && \pmod{n_i} \\ &\equiv a_i m_i (m_i^{-1} \bmod n_i) && \pmod{n_i} \\ &\equiv a_i && \pmod{n_i} \end{aligned}$$

که همان چیزی است که می‌خواستیم نشان دهیم: متد ما برای محاسبه‌ی a از a_iها، نتیجه‌ای مانند a می‌سازد که محدودیت $a \equiv a_i \pmod{n_i}$ را برای $i = 1, 2, ..., k$ ارضا می‌کند. تناظر، یک به یک است، چرا که می‌توانیم در هر دو جهت تبدیل را انجام دهیم. نهایتاً تساوی‌های (۳۱-۲۸)-(۳۱-۳۰) مستقیماً از تمرین ۳۱-۱-۷ نتیجه می‌شوند، چرا که $x \bmod n_i = (x \bmod n) \bmod n_i$ برای هر x و $i = 1, 2, ..., k$. ■

از نتیجه‌های زیر بعداً در همین فصل استفاده خواهد شد.

نتیجه‌ی ۳۱-۲۸ اگر $n_1, n_2, ..., n_k$ دو به دو نسبت به هم اول باشند و $n = n_1 n_2 ... n_k$، آن گاه برای اعداد صحیح دلخواه $a_1, a_2, ..., a_k$، مجموعه‌ی تساوی‌های هم‌زمان

$$x \equiv a_i \pmod{n_i}$$

برای $i = 1, 2, ..., k$ و مجهول x یک جواب یکتا به پیمانه‌ی n دارد.

■

نتیجه‌ی ۳۱-۳۹ اگر $n_1, n_2, ..., n_k$ دو به دو نسبت به هم اول باشند و $n = n_1 n_2 ... n_k$، آن گاه برای تمام اعداد صحیح x و a،

$$x \equiv a \pmod{n_i}$$

برای $i = 1, 2, ..., k$ اگر و فقط اگر

$$x \equiv a \pmod{n}$$

به عنوان یک مثال از کاربرد قضیه‌ی باقی‌مانده‌ی چینی، فرض کنید که به ما دو معادله‌ی

$$a \equiv ۲ \pmod{۵},$$
$$a \equiv ۳ \pmod{۱۳}$$

داده شده است، به طوری که $a_1 = ۲$، $n_1 = m_2 = ۵$، $a_2 = ۳$، و $n_2 = m_1 = ۱۳$، و می‌خواهیم $a \bmod ۶۵$ را محاسبه کنیم، چون $n = ۶۵$. از آن جایی که $۱۳^{-1} \equiv ۲ \pmod{۵}$ و $۵^{-1} \equiv ۸ \pmod{۸}$، داریم

$$c_1 = ۱۳(۲ \bmod ۵) = ۲۶,$$
$$c_2 = ۵(۸ \bmod ۱۳) = ۴۰$$

و

$$\begin{aligned} a &\equiv ۲۰۲۶ + ۲۰۴۰ \pmod{۶۵} \\ &\equiv ۵۲ + ۱۲۰ \pmod{۶۵} \\ &\equiv ۴۲ \pmod{۶۵} \end{aligned}$$

شکل ۳۱-۳ را برای تشریح قضیه‌ی باقی‌مانده‌ی چینی به پیمانه‌ی ۶۵ ببینید.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	40	15	55	30	5	45	20	60	35	10	50	25
1	26	1	41	16	56	31	6	46	21	61	36	11	51
2	52	27	2	42	17	57	32	7	47	22	62	37	12
3	13	53	28	3	43	18	58	33	8	48	23	63	38
4	39	14	54	29	4	44	19	59	34	9	49	24	64

شکل ۳۱-۳ تشریحی از قضیه‌ی باقی‌مانده‌ی چینی برای $n_1 = ۵$ و $n_2 = ۱۳$. برای این مثال، $c_1 = ۲۶$ و $c_2 = ۴۰$. در سطر i و ستون j مقدار a به پیمانه‌ی ۶۵ نشان داده شده است، به طوری که $i = (a \bmod ۵)$ و $j = (a \bmod ۱۳)$. توجه کنید که در سطر ۰ و ستون ۰ مقدار ۰ قرار دارد. به طور مشابه، سطر ۴ و ستون ۱۲ حاوی مقدار ۶۴ (هم‌ارز با -۱) است. چون $c_1 = ۲۶$، حرکت به سمت پایین در یک سطر مقدار a را ۲۶ واحد افزایش می‌دهد. به طور مشابه، $c_2 = ۴۰$ بدین معنی است که حرکت به سمت پایین در یک ستون a را به اندازه‌ی ۴۰ واحد افزایش می‌دهد. یک واحد افزایش در a به معنی حرکت در قطر به سمت پایین و راست است، که در آن آخرین خانه‌ی پایین به اولین خانه‌ی بالا و آخرین خانه‌ی راست به اولین خانه‌ی چپ متصل شده است.

بنابراین برای کار کردن به پیمانه‌ی n، می‌توانیم مستقیماً به پیمانه‌ی n کار کنیم و یا می‌توانیم با استفاده از پیمانه‌های جداگانه‌ی n_i در نمایش تبدیل شده کار کنیم، که ممکن است کار را ساده‌تر کند. محاسبات در دو سیستم کاملاً معادل هستند.

تمرین‌ها

۳۱-۵-۱ تمام جواب‌های معادلات $x \equiv ۴ \pmod ۵$ و $x \equiv ۵ \pmod{۱۱}$ را بیابید.

۳۱-۵-۲ تمام اعداد صحیح x را بیابید که باقی‌مانده‌ی آن‌ها بر ۹، ۸، و ۷ به ترتیب ۱، ۲، و ۳ است.

۳۱-۵-۳ بحث کنید که تحت تعاریف قضیه‌ی ۳۱-۲۷، اگر $\gcd(a,n)=۱$، آن گاه

$$(a^{-۱} \bmod n) \leftrightarrow ((a_۱^{-۱} \bmod n_۱),(a_۲^{-۱} \bmod n_۲),...,(a_k^{-۱} \bmod n_k))$$

۳۱-۵-۴ تحت تعاریف قضیه‌ی ۳۱-۲۷، اثبات کنید که برای هر چندجمله‌ای f، تعداد ریشه‌های $f(x) \equiv ۰ \pmod n$ برابر است با ضرب تعداد ریشه‌های هر یک از معادلات $f(x) \equiv ۰ \pmod{n_۱}$، $f(x) \equiv ۰ \pmod{n_۲}$، ...، $f(x) \equiv ۰ \pmod{n_k}$.

۳۱-۶ توان‌های یک عنصر

همان طور که ممکن است بخواهیم مضارب یک عنصر a به پیمانه‌ی n را بدانیم، بعضی مواقع ممکن است بخواهیم دنباله‌ی توان‌های a به پیمانه‌ی n را به دست آوریم، که در آن $a \in \mathbb{Z}_n^*$:

$$a^۰, a^۱, a^۲, a^۳, ... \qquad (۳۱-۳۳)$$

به پیمانه‌ی n. با اندیس‌گذاری از ۰، ۰امین مقدار در این دنباله $a^۰ \bmod n = ۱$ است، و iامین مقدار $a^i \bmod n$. مثلاً توان‌های ۳ به پیمانه‌ی ۷ عبارتند از

i	۰	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱
$۳^i \bmod ۷$	۱	۳	۲	۶	۴	۵	۱	۳	۲	۶	۴	۵

و توان‌های ۲ به پیمانه‌ی ۷ عبارتند از

i	۰	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱
$۲^i \bmod ۷$	۱	۲	۴	۱	۲	۴	۱	۲	۴	۱	۲	۴

در این بخش، فرض کنید که $\langle a \rangle$ نشان‌دهنده‌ی زیرگروه $\mathbb{Z}_n^*$ ساخته شده توسط a با ضرب مکرر باشد، و $\mathrm{ord}_n(a)$ (مرتبه‌ی a به پیمانه‌ی n) نشان‌دهنده‌ی مرتبه‌ی a در $\mathbb{Z}_n^*$. مثلاً $\langle ۲ \rangle = \{۱,۲,۴\}$ در $\mathbb{Z}_۷^*$، و $\mathrm{ord}_۷(۲) = ۳$. با استفاده از تعریف اویلر از تابع $\varphi(n)$ به صورت اندازه‌ی $\mathbb{Z}_n^*$ (بخش ۳۱-۳ را ببینید)، اکنون نتیجه‌ی ۳۱-۱۹ را به زبان $\mathbb{Z}_n^*$ ترجمه می‌کنیم تا به قضیه‌ی اویلر برسیم و آن را برای $\mathbb{Z}_p^*$ محدود کنیم، که در آن p اول است، و به قضیه‌ی فرما برسیم.

برای هر عدد صحیح $n > ۱$،

برای هر $a \in \mathbb{Z}_n^*$ $a^{\phi(n)} \equiv ۱ \pmod{n}$

اگر p اول باشد، آن گاه

برای هر $a \in \mathbb{Z}_p^*$ $a^{p-۱} \equiv ۱ \pmod{p}$

اثبات طبق تساوی (۳۱-۲۰)، اگر p اول باشد آن گاه $\varphi(p) = p - ۱$.

این نتیجه برای تمام عناصر $\mathbb{Z}_p$ غیر از ۰ کاربرد دارد، چرا که $۰ \notin \mathbb{Z}_p^*$. با این حال، برای هر $a \in \mathbb{Z}_p$ داریم $a^p \equiv a \pmod{p}$ اگر p اول باشد.

اگر $\text{ord}_n(g) = |\mathbb{Z}_n^*|$، آن گاه تمام عناصر در $\mathbb{Z}_n^*$ توانی از g به پیمانه‌ی n هستند، و می‌گوییم g یک *ریشه‌ی اصلی* (primitive root) یا یک *مولد* (generator) $\mathbb{Z}_n^*$ است. مثلاً ۳ به پیمانه‌ی ۷ یک ریشه‌ی اصلی است، ولی ۲ این طور نیست. اگر $\mathbb{Z}_n^*$ شامل یک ریشه‌ی اصلی باشد، می‌گوییم گروه $\mathbb{Z}_n^*$ *تناوبی* (cyclic) است. از اثبات قضیه‌ی زیر صرف نظر می‌کنیم.

مقادیر $n > ۱$ که برای آن‌ها $\mathbb{Z}_n^*$ تناوبی است عبارتند از ۲، ۴، p^e و $۲p^e$ برای تمام اعداد اول $p > ۲$ و تمام اعداد صحیح مثبت e.

اگر g یک ریشه‌ی اصلی $\mathbb{Z}_n^*$ باشد، و a یک عنصر دلخواه در $\mathbb{Z}_n^*$، آن گاه یک عدد صحیح z وجود دارد به طوری که $g^z \equiv a \pmod{n}$. به این z *لوگاریتم گسسته* (discrete logarithm) یا *اندیس* (index) a به پیمانه‌ی n در مبنای g گفته می‌شود؛ این مقدار را با $\text{ind}_{n,g}(a)$ نشان می‌دهیم.

اگر g یک ریشه‌ی اصلی $\mathbb{Z}_n^*$ باشد، آن گاه تساوی $g^x \equiv g^y \pmod{n}$ برقرار است اگر و فقط اگر تساوی $x \equiv y \pmod{\varphi(n)}$ برقرار باشد.

اثبات ابتدا فرض کنید که $x \equiv y \pmod{\varphi(n)}$. آن گاه $x = y + k\varphi(n)$ برای یک عدد صحیح k. بنابراین

$$\begin{aligned} g^x &\equiv g^{y+k\phi(n)} && \pmod n \\ &\equiv g^y \cdot (g^{\phi(n)})^k && \pmod n \\ &\equiv g^y \cdot 1^k && \pmod n \quad \text{(طبق قضیه‌ی اویلر)} \\ &\equiv g^y && \pmod n \end{aligned}$$

برعکس، فرض کنید که $g^x \equiv g^y \pmod n$. چون دنباله‌ی توان‌های g تمام عناصر $\langle g \rangle$ را می‌سازند، و $|\langle g \rangle| = \phi(n)$، نتیجه‌ی ۳۱-۱۸ ایجاب می‌کند که دنباله‌ی توان‌های g متناوب با دوره‌ی تناوب $\phi(n)$ باشد. بنابراین اگر $g^x \equiv g^y \pmod n$، آن گاه باید داشته باشیم $x \equiv y \pmod{\phi(n)}$. ■

اکنون توجه خود را به ریشه‌های دوم ۱ به پیمانه‌ی یک توان اول معطوف می‌کنیم. قضیه‌ی زیر در توسعه‌ی الگوریتم تست اول بودن در بخش ۳۱-۸ مفید خواهد بود.

قضیه‌ی ۳۱-۳۴ اگر p یک عدد اول فرد باشد و $e \geq 1$، آن گاه تساوی

$$x^2 \equiv 1 \pmod{p^e} \qquad (۳۱-۳۴)$$

فقط دو جواب دارد، $x = 1$ و $x = -1$.

اثبات تساوی (۳۱-۳۴) معادل است با

$$p^e \mid (x-1)(x+1)$$

از آن جایی که $p > 2$، می‌توانیم داشته باشیم $p \mid (x-1)$ یا $p \mid (x+1)$، ولی نه هر دو. (در غیر این صورت طبق خصوصیت (۳۱-۳)، p باید اختلاف آن‌ها یعنی $(x+1)-(x-1) = 2$ را هم بشمارد.) اگر $p \nmid (x-1)$، آن گاه $\gcd(p^e, x-1) = 1$، و بنابراین طبق قضیه‌ی ۳۱-۵ خواهیم داشت $p^e \mid (x+1)$. یعنی $x \equiv -1 \pmod{p^e}$. به طور مشابه اگر $p \nmid (x+1)$، آن گاه $\gcd(p^e, x+1) = 1$، و قضیه‌ی ۳۱-۵ ایجاب می‌کند که $p^e \mid (x-1)$، و بنابراین $x \equiv 1 \pmod{p^e}$. بنابراین یا $x \equiv -1 \pmod{p^e}$ یا $x \equiv 1 \pmod{p^e}$. ■

یک عدد x یک *ریشه‌ی دوم نابدیهی ۱ به پیمانه‌ی n* است، اگر تساوی $x^2 \equiv 1 \pmod n$ را ارضا کند، ولی برابر با هیچ یک از ریشه‌های دوم «بدیهی» به پیمانه‌ی n نباشد: ۱ یا ۱-. مثلاً ۶ یک ریشه‌ی دوم نابدیهی ۱ به پیمانه‌ی ۳۵ است. از نتیجه‌ی زیر از قضیه‌ی ۳۱-۳۴ در اثبات درستی رویه‌ی تست اول بودن Miller-Rabin در بخش ۳۱-۸ استفاده خواهد شد.

نتیجه‌ی ۳۱-۳۵ اگر یک ریشه‌ی دوم نابدیهی ۱ به پیمانه‌ی n وجود داشته باشد، آن گاه n مرکب است.

اثبات طبق عکس نقیض قضیه‌ی ۳۱-۳۴، اگر یک ریشه‌ی دوم نابدیهی ۱ به پیمانه‌ی n وجود داشته باشد، آن گاه n نمی‌تواند اول و فرد و یا توانی از یک عدد اول و فرد باشد. اگر $x^2 \equiv 1 \pmod 2$، آن گاه $x \equiv 1 \pmod 2$، و تمام ریشه‌های دوم ۱ به پیمانه‌ی ۲ بدیهی هستند. بنابراین n نمی‌تواند اول باشد.

نهایتاً باید داشته باشیم $n > 1$ تا یک ریشه‌ی دوم نابدیهی ۱ وجود داشته باشد. بنابراین n باید مرکب باشد.

به دست آوردن توان‌ها با مربع‌گیری مکرر

یک عملیات معمول در محاسبات نظریه‌ی اعداد، به توان رساندن یک عدد به پیمانه‌ی عددی دیگر است، که به *توان رسانی پیمانه‌ای* (modular exponentiation) نام دارد. به طور دقیق‌تر، می‌خواهیم یک روش کارآمد بیابیم برای محاسبه‌ی $a^b \bmod n$، که در آن a و b اعداد صحیح نامنفی، و n یک عدد صحیح مثبت است. به توان رسانی پیمانه‌ای عملیاتی است حیاتی در بسیاری از تست‌های اول بودن و همچنین در سیستم رمزنگاری کلید عمومی RSA. متد مربع‌گیری مکرر با استفاده از نمایش دودویی b این مسئله را به صورت بهینه حل می‌کند.

فرض کنید $\langle b_k, b_{k-1}, \ldots, b_1, b_0 \rangle$ نمایش دودویی b باشد. (یعنی نمایش دودویی $k+1$ بیت طول دارد، b_k پرارزش‌ترین بیت است، و b_0 کم‌ارزش‌ترین بیت.) رویه‌ی زیر، c را با دوبرابر کردن و افزایش دادن، از ۰ به b رسانده و $a^c \bmod n$ را محاسبه می‌کند.

```
MODULAR-EXPONENTIATION(a, b, n)
1  c = 0
2  d = 1
3  let ⟨b_k, b_{k-1}, ..., b_0⟩ be the binary representation of b
4  for i = k downto 0
5      c = 2c
6      d = (d · d) mod n
7      if b_i == 1
8          c = c + 1
9          d = (d · a) mod n
10 return d
```

استفاده‌ی مناسب از مربع‌گیری در خط ۶ در هر یک از تکرارها، نام «مربع‌گیری مکرر» را توجیح می‌کند. به عنوان یک مثال، برای $a = 7$، $b = 560$، و $n = 561$، الگوریتم دنباله‌ی مقادیر نشان داده شده در شکل ۳۱-۴ به پیمانه‌ی ۵۶۱ را محاسبه می‌کند؛ دنباله‌ی توان‌های استفاده شده را در سطری از جدول که با c مشخص شده است، می‌بینید.

i	9	8	7	6	5	4	3	2	1	0
b_i	1	0	0	0	1	1	0	0	0	0
c	1	2	4	8	17	35	70	140	280	560
d	7	49	157	526	160	241	298	166	67	1

شکل ۳۱-۴ نتیجه‌ی MODULAR-EXPONENTIATION برای محاسبه‌ی $a^b \pmod n$ وقتی که $a = 7$، $\langle 1000110000 \rangle = 560 = b$، و $n = 561$. مقادیر بعد از هر بار اجرای حلقه‌ی for نشان داده شده‌اند. نتیجه‌ی نهایی ۱ است.

در واقع در این الگوریتم نیازی به متغیر c نیست، ولی برای ثابت حلقه‌ی دوبخشی زیر از آن استفاده شده است:

دقیقاً قبل از هر بار تکرار حلقه‌ی for در خطوط ۴-۹،

۱. مقدار c برابر است با پیشوند $\langle b_k, b_{k-1}, \ldots, b_{i+1}\rangle$ از نمایش دودویی b، و

۲. $d = a^c \bmod n$.

از این ثابت حلقه به صورت زیر استفاده می‌کنیم:

- آغاز: در ابتدا $i = k$، و پیشوند $\langle b_k, b_{k-1}, \ldots, b_{i+1}\rangle$ تهی است، که متناسب است با $c = 0$. به علاوه $d = 1 = a^0 \bmod n$.
- ادامه: فرض کنید c' و d' نشان‌دهنده‌ی مقادیر c و d در انتهای یک تکرار از حلقه‌ی for باشند، و بنابراین مقادیر همین متغیرها در ابتدای تکرار بعد. در هر تکرار به هنگام‌سازی $c' = 2c$ (اگر $b_i = 0$) یا $c' = 2c + 1$ (اگر $b_i = 1$) را انجام می‌دهد، به طوری که c قبل از تکرار بعد صحیح خواهد بود. اگر $b_i = 0$، آن گاه $d' = d^2 \bmod n = (a^c)^2 \bmod n = a^{c'} \bmod n$. اگر $b_i = 1$، آن‌گاه $d' = d^2 a \bmod n = (a^c)^2 a \bmod n = a^{2c+1} \bmod n = a^{c'} \bmod n$. در هر دو حالت قبل از تکرار بعد داریم $d = a^c \bmod n$.
- پایان: در انتها، $i = -1$. بنابراین $c = b$، چون c برابر است با مقدار پیشوند $\langle b_k, b_{k-1}, \ldots, b_0\rangle$ از نمایش دودویی b. بنابراین $d = a^c \bmod n = a^b \bmod n$.

اگر ورودی‌های a، b، و n اعداد β بیتی باشند، آن گاه تعداد کل اعمال ریاضی مورد نیاز $O(\beta)$، و تعداد کل اعمال بیتی مورد نیاز $O(\beta^3)$ است.

تمرین‌ها

۳۱-۶-۱ یک جدول بکشید که مرتبه‌ی هر عنصر در $\mathbb{Z}_{11}^*$ را نشان می‌دهد. کوچک‌ترین ریشه‌ی اصلی g را انتخاب کرده و جدول $\mathrm{ind}_{11,g}(x)$ را برای $x \in \mathbb{Z}_{11}^*$ محاسبه کنید.

۳۱-۶-۲ یک الگوریتم به توان رسانی پیمانه‌ای ارائه کنید که بیت‌های b را به جای از چپ به راست، از راست به چپ بررسی می‌کند.

۳۱-۶-۳ فرض کنید که $\varphi(n)$ را می‌دانیم، توضیح دهید که چگونه می‌توان با استفاده از رویه‌ی MODULAR-EXPONENTIATION، مقدار $a^{-1} \bmod n$ را برای تمام $a \in \mathbb{Z}_n^*$ محاسبه کرد.

۳۱-۷ سیستم رمزنگاری کلید عمومی RSA

از سیستم رمزنگاری کلید عمومی می‌توان برای رمزنگاری پیام‌های فرستاده شده بین دو شخص استفاده کرد به طوری که کسی که استراق سمع می‌کند قادر به رمزگشایی آن نباشد. سیستم رمزنگاری کلید عمومی همچنین یک طرف ارتباط را قادر می‌سازد که یک «امضای الکترونیک» غیر قابل تقلید به انتهای پیام الکترونیکی اضافه کند. چنین امضایی، نسخه‌ی الکترونیکی امضای دست‌نویس بر روی سندهای کاغذی است. این امضا قابل چک کردن توسط هر کسی است، غیر قابل تقلید است، و تغییر یک بیت در پیام اعتبار آن را از بین می‌برد. بنابراین هم هویت امضا کننده را تأیید می‌کند، و هم صحت پیام فرستاده شده را. این امضا، ابزاری است بی‌نقص برای قراردادهای تجاری الکترونیکی، چک‌های الکترونیکی، سفارشات خرید الکترونیکی، و دیگر ارتباطات الکترونیکی که در آن‌ها تشخیص هویت مهم است.

سیستم رمزنگاری کلید عمومی RSA بر پایه‌ی تفاوت وحشتناک میان سادگی یافتن اعداد اول بزرگ و سختی تجزیه‌ی ضرب دو عدد اول بزرگ بنا شده است. بخش ۳۱-۸ یک رویه‌ی کارآمد برای یافتن اعداد اول بزرگ ارائه می‌کند، و در بخش ۳۱-۹ مسئله‌ی تجزیه‌ی ضرب دو عدد اول بزرگ مورد بررسی قرار می‌گیرد.

سیستم رمزنگاری کلید عمومی

در یک سیستم رمزنگاری کلید عمومی، هر طرف ارتباط یک *کلید عمومی* (public key) و یک *کلید اختصاصی* (secret key) دارد. مثلاً در سیستم رمزنگاری RSA، هر کلید شامل یک جفت عدد صحیح است. مرسوم است که در مثال‌های سیستم‌های رمزنگاری، از مکالمه‌کننده‌های «آلیس» و «باب» استفاده کنند؛ کلیدهای عمومی و اختصاصی آن‌ها را با P_A و S_A برای آلیس و P_B و S_B برای باب نشان خواهیم داد.

هر طرف مکالمه، خود کلیدهای عمومی و اختصاصی خود را می‌سازد. هر طرف کلید اختصاصی خود را نزد خود نگه می‌دارد، ولی می‌تواند کلید عمومی را در اختیار هر کسی بگذارد، و یا حتی می‌تواند آن را منتشر کند. در واقع، معمولاً ساده است که فرض کنیم که کلید عمومی هر کس در دسترس همه قرار دارد، و هر کس در صورت لزوم می‌تواند از کلید عمومی هر کس دیگری استفاده کند.

کلیدهای عمومی و اختصاصی مشخص کننده‌ی توابعی هستند که می‌توان آن‌ها را بر روی پیام‌ها اعمال کرد. فرض کنید $\mathcal{D}$ نشان دهنده‌ی مجموعه‌ی پیام‌های مجاز باشد. مثلاً $\mathcal{D}$ ممکن است مجموعه‌ی تمام دنباله‌های بیتی متناهی باشد. در ساده‌ترین حالت، و حالت اصلی رمزنگاری کلید عمومی، لازم است که کلیدهای عمومی و اختصاصی نشان دهنده‌ی توابعی یک به یک از $\mathcal{D}$ به $\mathcal{D}$ باشند. تابع کلید عمومی آلیس با $P_A()$ و تابع مربوط به کلید اختصاصی او با $S_A()$ نشان داده می‌شود. بنابراین توابع $P_A()$ و $S_A()$ جایگشت‌هایی از $\mathcal{D}$ هستند. فرض می‌کنیم که توابع $P_A()$ و $S_A()$ با داشتن کلیدهای P_A و S_A به صورت کارآمد قابل محاسبه هستند.

کلیدهای عمومی و اختصاصی هر شخص یک «جفت متناسب» هستند، به طوری که توابع آن‌ها

معکوس یکدیگرند. یعنی

$$M = S_A(P_A(M)), \tag{۳۱-۳۵}$$

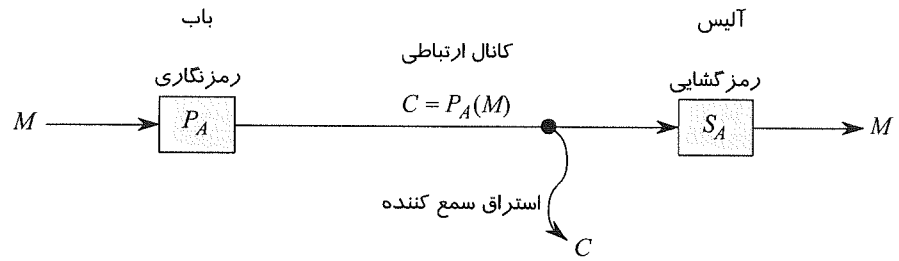
$$M = P_A(S_A(M)) \tag{۳۱-۳۶}$$

برای هر پیام $M \in \mathcal{D}$. تبدیل متوالی M با دو کلید P_A و S_A به هر ترتیبی، دوباره به کلید M منجر می‌شود.

در یک سیستم رمزنگاری کلید عمومی، حیاتی است که هیچ کس غیر از آلیس در مدت زمانی قابل دسترس، قادر به محاسبه‌ی تابع $S_A()$ نباشد. حریم پیامی که رمزنگاری شده و برای آلیس فرستاده شده است و هویت امضای دیجیتال آلیس وابسته به این فرض است که فقط آلیس قادر است $S_A()$ را محاسبه کند. به دلیل این نیاز است که آلیس باید S_A را به صورت اختصاصی نگه دارد؛ اگر این کار را نکند، یکتایی خود را از دست می‌دهد و سیستم نمی‌تواند به او توانایی‌های خاص بدهد. این فرض که فقط آلیس می‌تواند $S_A()$ را محاسبه کند باید در حالی برقرار باشد که همه P_A را می‌دانند و می‌توانند $P_A()$ را محاسبه کنند، که معکوس تابع $S_A()$ است. سختی اصلی در طراحی یک سیستم رمزنگاری کلید عمومی عملی، یافتن روشی است برای ساختن یک سیستم که در آن می‌توانیم $P_A()$ را منتشر کنیم، بدون این که روش محاسبه‌ی معکوس آن $S_A()$ را منتشر کرده باشیم. این کار به نظر بسیار دشوار می‌آید، ولی خواهیم دید که چگونه می‌توان آن را عملی کرد.

در یک سیستم رمزنگاری کلید عمومی، رمزنگاری مانند شکل ۳۱-۵ انجام می‌شود. فرض کنید باب می‌خواهد یک پیام M برای آلیس بفرستد، که طوری رمزنگاری شده است که برای استراق سمع کننده کاملاً بی‌معنی است. سناریوی فرستادن پیام به صورت زیر است.

- باب کلید عمومی آلیس یعنی P_A را به دست می‌آورد (که در اختیار همگان قرار دارد).
- باب متن رمزنگاری شده‌ی $C = P_A(M)$ مربوط به پیام M را محاسبه کرده و C را برای آلیس می‌فرستد.



شکل ۳۱-۵ رمزنگاری در یک سیستم کلید عمومی. باب با استفاده از کلید عمومی آلیس پیام M را رمزنگاری کرده و متن حاصل $C = P_A(M)$ را برای آلیس می‌فرستد. اگر در این میان پیام به دست کسی برسد که قصد استراق سمع دارد، این متن رمزنگاری شده هیچ اطلاعاتی در مورد M به او نخواهد داد. آلیس C را دریافت و با استفاده از کلید اختصاصی خود کلید اصلی $M = S_A(C)$ را به دست می‌آورد.

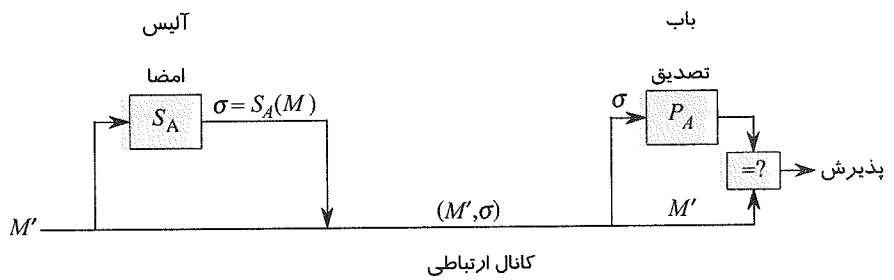
- وقتی آلیس متن رمزنگاری شده‌ی C را دریافت می‌کند، کلید اختصاصی خود، یعنی S_A را به آن اعمال کرده و پیام اصلی را به دست می‌آورد: $S_A(C) = S_A(P_A(M)) = M$.

چون $S_A()$ و $P_A()$ معکوس یکدیگر هستند، آلیس می‌تواند M را از C محاسبه کند. چون فقط آلیس می‌تواند $S_A()$ را محاسبه کند، پس تنها کسی است که می‌تواند از C به M دست یابد. رمزنگاری M با استفاده از $P_A()$، M را از دسترسی هر کس غیر از آلیس محفوظ نگه داشته است.

با استفاده از فرمول‌بندی بالا از سیستم رمزنگاری کلید عمومی، امضای دیجیتال هم به سادگی قابل پیاده‌سازی است. (توجه می‌کنیم که روش‌های دیگری هم برای برخورد با مسئله‌ی ساختن امضای دیجیتال وجود دارد، که در این جا در مورد آن‌ها بحث نخواهد شد.) اکنون فرض کنید که آلیس می‌خواهد یک جواب M' با امضای دیجیتال برای باب بفرستد. سناریوی امضای دیجیتال در شکل ۳۱-۶ نشان داده شده است.

- آلیس با استفاده از کلید اختصاصی خود S_A و تساوی $\Sigma = S_A(M')$، *امضای دیجیتال* خود Σ را برای پیام M' محاسبه می‌کند.
- آلیس جفت پیام/امضای (M', Σ) را برای باب می‌فرستد.
- وقتی باب (M', Σ) را دریافت می‌کند، می‌تواند با استفاده از کلید عمومی آلیس و تساوی $M' = P_A(\Sigma)$ دریابد که آیا آلیس آن را فرستاده است یا خیر. (احتمالاً M' حاوی نام آلیس هم هست، و بنابراین باب می‌داند که از کلید عمومی چه کسی استفاده کند.) اگر تساوی برقرار باشد، باب نتیجه می‌گیرد که پیام M' واقعاً از طرف آلیس آمده است. اگر تساوی برقرار نباشد، باب نتیجه می‌گیرد که یا پیام M' و یا امضای Σ توسط خطای انتقال خراب شده است، و یا جفت (M', Σ) جعلی است.

چون امضای دیجیتال هم هویت فرستنده را تأیید می‌کند و هم صحت محتوای پیام را، متناظر است با امضای دست‌نویس در انتهای اسناد کاغذی.



شکل ۳۱-۶ امضای دیجیتال در سیستم کلید عمومی. آلیس پیام M' را بدین صورت امضا می‌کند که امضای دیجیتال خود $\Sigma = S_A(M')$ را به آن الحاق می‌کند. سپس جفت پیام/امضای (M', Σ) را برای باب می‌فرستد، که او هم می‌تواند با چک کردن درستی رابطه‌ی $M' = P_A(\Sigma)$ صحت آن را بررسی کند. اگر تساوی برقرار باشد، او می‌تواند بپذیرد که پیام (M', Σ) را آلیس فرستاده است.

یک خصوصیت مهم یک امضای دیجیتال این است که صحت توسط هر کس که به کلید عمومی امضا کننده دسترسی دارد قابل بررسی است. یک پیام امضا شده می‌تواند توسط یک نفر چک شده، و سپس برای شخصی دیگر فرستاده شود تا او هم صحت آن را بررسی کند. مثلاً، پیام می‌تواند یک چک الکترونیکی باشد که آلیس برای باب فرستاده است. بعد از این که باب امضای آلیس را بر روی چک بررسی کرد، می‌تواند آن را برای بانک خود بفرستد، که آن‌ها هم می‌توانند صحت آن را بررسی کرده و اعمال مورد نیاز را انجام دهند.

توجه می‌کنیم که خود پیام امضا شده، لزوماً رمزنگاری شده نیست؛ پیام می‌تواند «در دسترش همگان» باشد، و هیچ محافظتی از آن صورت نگیرد. با ترکیبی از پروتکل‌های بالا برای رمزنگاری و امضا، می‌توانیم پیام‌هایی بسازیم که هم رمزنگاری شده‌اند و هم امضا. امضا کننده، ابتدا امضای دیجیتال خود را به پیام الحاق می‌کند و سپس با استفاده از کلید عمومی گیرنده، جفت پیام/امضای حاصل را رمزنگاری می‌کند. گیرنده پیام دریافتی را با استفاده از کلید اختصاصی خود رمزگشایی می‌کند تا به پیام و امضای فرستاده شده دست یابد. سپس می‌تواند با استفاده از کلید عمومی فرستنده، صحت پیام و امضا را بررسی کند. فرآیند ترکیب شده‌ی مربوط در سیستم امضای کاغذی بدین صورت است که فرستنده ابتدا سند مورد نظر را امضا می‌کند، و سپس آن را در یک پاکت قرار می‌دهد، که در نهایت گیرنده آن را باز می‌کند.

سیستم رمزنگاری RSA

در *سیستم رمزنگاری کلید عمومی RSA*، طرف‌های ارتباط کلیدهای عمومی و اختصاصی خود را به روش زیر می‌سازند.

۱. انتخاب دو عدد اول بزرگ p و q به صورت تصادفی، به طوری که $p \neq q$. به عنوان نمونه، اعداد اول p و q می‌توانند هر یک ۱۰۲۴ بیت باشند.

۲. محاسبه‌ی $n = pq$.

۳. انتخاب یک عدد فرد کوچک e که نسبت به $\varphi(n)$ اول است، که طبق تساوی (۳۱-۲۰) برابر است با $(p-1)(q-1)$.

٤. محاسبه‌ی d که معکوس ضربی e به پیمانه‌ی $\varphi(n)$ است. (نتیجه‌ی ۳۱-۲۶ تضمین می‌کند که d وجود دارد و یکتا است. برای محاسبه‌ی d با استفاده از e و $\varphi(n)$ می‌توانیم از تکنیک‌های بخش ۳۱-۴ استفاده کنیم.)

۵. انتشار جفت $P = (e, n)$ به عنوان *کلید عمومی RSA*.

۶. نگه داری از جفت $S = (d, n)$ به عنوان *کلید اختصاصی RSA*.

در این رویکرد، دامنه‌ی $\mathcal{D}$ مجموعه‌ی $\mathbb{Z}_n$ است. تبدیل یک پیام M با یک کلید عمومی $P = (e, n)$ بدین صورت است:

$$P(M) = M^e \pmod{n} \qquad (۳۱-۳۷)$$

و تبدیل یک متن رمزنگاری شده با یک کلید اختصاصی $S = (d, n)$ بدین صورت:

$$S(C) = C^d \pmod{n} \qquad (۳۱-۳۸)$$

این معادلات هم برای رمزنگاری به کار می‌روند و هم برای امضا. برای ساختن یک امضا، امضا کننده کلید اختصاصی خود را به پیامی که می‌خواهد امضا کند، اعمال می‌کند، به جای این که به متن رمزنگاری شده اعمال کند. برای تأیید صحت یک امضا، کلید عمومی امضا کننده به خود آن اعمال می‌شود، به جای این که به پیام اعمال شود.

اعمال کلید عمومی و اختصاصی را می‌توان با استفاده از رویه‌ی MODULAR-EXPONENTIATION که در بخش ۳۱-۶ توصیف شد، پیاده‌سازی کرد. برای تحلیل زمان اجرای این اعمال، فرض کنید که کلید عمومی (e, n) و کلید اختصاصی (d, n) این شرایط را ارضا می‌کنند: $\lg e = O(1)$، $\lg d \le \beta$، و $\lg n \le \beta$. آن گاه اِعمال یک کلید عمومی به $O(1)$ عملیات پیمانه‌ای و $O(\beta^2)$ عملیات بیتی نیاز دارد. اِعمال یک کلید اختصاصی به $O(\beta)$ عملیات پیمانه‌ای نیاز دارد که از $O(\beta^3)$ عملیات بیتی استفاده می‌کند.

نتیجه‌ی ۳۱-۳۶ *(صحت RSA)*

RSA تعریف شده در تساوی‌های (۳۱-۳۷) و (۳۱-۳۸)، تبدیل‌هایی بر روی $\mathbb{Z}_n$ هستند که معکوس یکدیگرند و تساوی‌های (۳۱-۳۵) و (۳۱-۳۶) را ارضا می‌کنند.

اثبات از تساوی‌های (۳۱-۳۷) و (۳۱-۳۸)، داریم برای هر $M \in \mathbb{Z}_n$،

$$P(S(M)) = S(P(M)) = M^{ed} \pmod{n}$$

چون e و d معکوس ضربی یکدیگر به پیمانه‌ی $\varphi(n) = (p-1)(q-1)$ هستند،

$$ed = 1 + k(p-1)(q-1)$$

برای یک عدد صحیح k. ولی در این صورت، اگر $M \not\equiv 0 \pmod{p}$، داریم

$$\begin{aligned} M^{ed} &\equiv M(M^{p-1})^{k(q-1)} && \pmod{p} \\ &\equiv M((M \bmod p)^{p-1})^{k(q-1)} && \pmod{p} \\ &\equiv M(1)^{k(q-1)} && \pmod{p} \quad \text{(طبق قضیه‌ی ۳۱-۳۱)} \\ &\equiv M && \pmod{p} \end{aligned}$$

همچنین، $M^{ed} \equiv M \pmod{p}$ اگر $M \equiv 0 \pmod{p}$. بنابراین

$$M^{ed} \equiv M \pmod{p}$$

برای تمام Mها. پس طبق نتیجه‌ی ۳۱-۲۹ از قضیه‌ی باقی‌مانده‌ی چینی،

$$M^{ed} \equiv M \pmod{n}$$

برای تمام Mها. ▒

امنیت سیستم رمزنگاری RSA به شدت به سختی تجزیه‌ی اعداد صحیح بزرگ وابسته است. اگر کسی بتواند پیمانه‌ی n را در یک کلید عمومی تجزیه کند، آن گاه می‌تواند کلید اختصاصی را از روی کلید عمومی به دست آورد، درست همان‌طوری که سازنده‌ی کلید عمومی با استفاده از p و q این کار را کرده است. بنابراین اگر تجزیه‌ی اعداد بزرگ ساده باشد، آن گاه شکستن سیستم رمزنگاری RSA هم ساده خواهد بود. عبارت برعکس این، که می‌گوید در صورت سخت بودن تجزیه‌ی اعداد بزرگ، شکستن سیستم رمزنگاری RSA هم مشکل خواهد بود، هنوز اثبات نشده است. از طرفی بعد از دو دهه تحقیق، هیچ راهی ساده‌تر از تجزیه‌ی پیمانه‌ی n برای شکستن سیستم رمزنگاری کلید عمومی RSA پیدا نشده است. و همان طور که در بخش ۳۱-۹ خواهیم دید، تجزیه‌ی اعداد بزرگ به طرز باور نکردنی مشکل است. با انتخاب تصادفی دو عدد اول ۱۰۲۴ بیتی و ضرب آن‌ها در یکدیگر، می‌توان یک کلید عمومی ساخت که با تکنولوژی امروز، شکستن آن در زمانی معقول امکان پذیر نیست. در غیاب پیشرفت اساسی در طراحی الگوریتم‌های نظریه‌ی اعداد، و در صورت رعایت استانداردهایی که در ادامه گفته خواهد شد در هنگام پیاده‌سازی، سیستم رمزنگاری RSA می‌تواند در عمل درجه‌ی بالایی از امنیت را فراهم آورد.

با این حال برای دست‌یابی به امنیت بالا در سیستم رمزنگاری RSA، بهتر است که با اعدادی کار کنیم که طول آن‌ها چند صد بیت یا حتی بیش از هزار بیت است، تا در صورت پیشرفت در الگوریتم‌های تجزیه، باز هم امنیت سیستم از دست نرود. در زمان نوشتن این کتاب (۲۰۰۹)، پیمانه‌ی RSA معمولاً بین ۷۶۸ تا ۲۰۴۸ بیت طول دارد. برای ساختن پیمانه‌ای با این اندازه، باید بتوانیم اعداد اول بزرگ تولید کنیم. بخش ۳۱-۸ این مسئله را مورد بحث قرار می‌دهد.

برای کارایی بیشتر، معمولاً از RSA به صورت «دورگه» به همراه سیستم‌های رمزنگاری سریع غیر کلید عمومی استفاده می‌شود. در این سیستم‌ها، کلیدهای مربوط رمزنگاری و رمزگشایی یکی هستند. اگر آلیس بخواهد یک پیام بلند و خصوصی M را برای باب بفرستد، یک کلید تصادفی M برای سیستم رمزنگاری سریع غیر کلید عمومی انتخاب می‌کند، و سپس M را با استفاده از K کد می‌کند، که متن کد شده‌ی C را به دست می‌دهد. در این جا، طول C برابر با طول M است، ولی K بسیار کوتاه‌تر از آن‌ها است. سپس آلیس K را با استفاده از کلید عمومی RSA باب رمزنگاری می‌کند. چون K کوتاه است، محاسبه‌ی $P_B(K)$ سریع انجام خواهد شد (بسیار سریع‌تر از محاسبه‌ی $P_B(M)$). او سپس $(C, P_B(K))$ را برای باب می‌فرستد، که او هم ابتدا $P_B(K)$ را رمزگشایی می‌کند تا K را به دست آورد، و سپس با استفاده از K، متن C را رمزگشایی می‌کند تا M را به دست آورد.

معمولاً از یک رویکرد دورگه‌ی مشابه دیگر هم برای ساختن امضاهای دیجیتال به صورت بهینه استفاده می‌شود. در این رویکرد، RSA با یک *تابع درهم سازی یک طرفه* h ترکیب می‌شود – h تابعی است که محاسبه‌ی آن ساده است ولی از نظر محاسباتی یافتن دو پیام M و M' به طوری که $h(M) = h(M')$ ناممکن است. مقدار $h(M)$ یک «اثر انگشت» کوتاه (مثلاً ۲۵۶ بیتی) از پیام M است. اگر آلیس بخواهد پیام M را امضا کند، ابتدا h را به M اعمال می‌کند تا اثر انگشت $h(M)$ را به دست آورد، که سپس آن را با استفاده از کلید اختصاصی خود رمزنگاری می‌کند. چیزی که آلیس به

عنوان پیام امضا شده برای باب می‌فرستد، $(M, S_A(h(M)))$ است. باب برای تأیید صحت امضا، می‌تواند $h(M)$ را محاسبه کند، و چک کند که آیا با اعمال P_A به $S_A(h(M))$، $h(M)$ به دست می‌آید یا خیر. چون هیچ کس نمی‌تواند دو پیام با اثر انگشت یکسان تولید کند، از نظر محاسباتی تغییر پیام و در عین حال حفظ صحت امضا غیر ممکن است.

نهایتاً، توجه می‌کنیم که استفاده از شناسنامه (certificate) می‌تواند انتشار کلیدهای عمومی را بسیار ساده‌تر کند. به عنوان مثال، فرض کنید یک «مرجع معتبر» T وجود دارد که کلید عمومی آن را همه می‌دانند. آلیس می‌تواند از این مرجع یک پیام امضا شده (که در واقع شناسنامه‌ی اوست) بگیرد که می‌گوید «کلید عمومی آلیس P_A است». از آن جایی که همه P_T را می‌دانند، تعیین هویت این شناسنامه برای همه ممکن است. آلیس می‌تواند شناسنامه‌ی خود را به همراه پیام‌های امضا شده‌ی خود بفرستند تا گیرنده نیازی به جستجو برای کلید عمومی آلیس نداشته باشد و بتواند به سرعت صحت امضای پیام را تأیید کند. چون کلید عمومی آلیس توسط T امضا شده است، گیرنده می‌تواند اطمینان داشته باشد که این کلید عمومی واقعاً متعلق به آلیس است.

تمرین‌ها

۳۱-۷-۱ یک کلید RSA به صورت $p = ۱۱$، $q = ۲۹$، $n = ۳۱۹$، و $e = ۳$ را در نظر بگیرید. از کلید اختصاصی از چه مقداری به عنوان d باید استفاده شود؟ متن رمزنگاری شده‌ی پیام $M = ۱۰۰$ چگونه است؟

۳۱-۷-۲ اثبات کنید که اگر نمای e در کلید عمومی آلیس ۳ باشد و یک دشمن نمای d مربوط به کلید اختصاصی او را به دست آورد، که در آن $۰ < d < \phi(n)$، آن گاه می‌تواند پیمانه‌ی آلیس (n) را در زمان چندجمله‌ای نسبت به تعداد بیت‌های n به دست آورد. (با این که لازم نیست این قضیه را اثبات کنید، ولی ممکن است علاقه‌مند باشید که بدانید حتی اگر شرط $e = ۳$ را حذف کنیم، باز هم این نتیجه صحیح است.)

۳۱-۷-۳★ اثبات کنید که RSA ضربی است، یعنی

$$P_A(M_۱)P_A(M_۲) \equiv P_A(M_۱M_۲) \pmod{n}$$

با استفاده از این مسئله، اثبات کنید که اگر کسی رویه‌ای داشته باشد که به کمک آن بتواند ۱ درصد از پیام‌هایی از $\mathbb{Z}_n$ را که به وسیله‌ی P_A رمزنگاری می‌شوند، رمزگشایی کند، آن گاه می‌تواند با استفاده از یک الگوریتم احتمالاًتی تمام پیام‌های رمزنگاری شده با P_A را با احتمال بالا رمزگشایی کند.

۳۱-۸★ تست اول بودن

در این بخش، مسئله‌ی یافتن اعداد اول بزرگ را در نظر می‌گیریم. با بحثی در مورد تراکم اعداد اول آغاز می‌کنیم، سپس یک رویکرد قابل قبول (ولی ناکامل) را برای تست اول بودن بررسی می‌کنیم، و سپس یک تست تصادفی مؤثر را برای اول بودن که منسوب به Miller و Rabin است، معرفی می‌کنیم.

تراکم اعداد اول

برای بسیاری از کاربردها (مانند رمزنگاری)، نیاز داریم که اعداد اول «تصادفی» بزرگ بیابیم. خوشبختانه، اعداد اول بزرگ چندان کم‌یاب نیستند، و بنابراین تست اعداد تصادفی با اندازه‌ی مناسب برای اول بودن تا یافتن یک عدد اول، کار چندان وقت‌گیری نیست. *تابع توزیع اعداد اول* $\pi(n)$ تعداد اعداد اول کوچک‌تر یا مساوی n را مشخص می‌کند. مثلاً $\pi(10) = 4$، چرا که ۴ عدد اول کوچک‌تر یا مساوی ۱۰ وجود دارد، که عبارتند از ۲، ۳، ۵، و ۷. قضیه‌ی اعداد اول یک تقریب برای $\pi(n)$ به دست می‌دهد.

قضیه‌ی ۳۱-۳۷ (قضیه‌ی اعداد اول)

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1$$

تقریب $n/\ln n$ با دقت معقولی $\pi(n)$ را تخمین می‌زند، حتی برای مقادیر کوچک n. مثلاً خطای این تقریب وقتی $n = 10^9$، کم‌تر از ۶% است، که در آن $\pi(n) = 50,847,534$ و $n/\ln n \approx 48,254,942$. (برای یک نظریه پرداز اعداد، 10^9 یک عدد کوچک است.)

می‌توان فرایند انتخاب یک عدد صحیح n به صورت تصادفی و تعیین اول بودن آن را به صورت یک آزمایش برنولی دید (بخش پ-۴ را ببینید). طبق قضیه‌ی اعداد اول، احتمال موفقیت – احتمال این که n اول باشد – تقریباً برابر است با $1/\ln n$. توزیع هندسی تعداد آزمایش‌های مورد نیاز برای دستیابی به موفقیت را به ما می‌گوید، و طبق تساوی (پ-۳۲)، امیدریاضی آزمایش‌ها تقریباً برابر است با $\ln n$. بنابراین برای یافتن یک عدد اول که هم‌اندازه‌ی n است، نیاز داریم که تقریباً $\ln n$ عدد نزدیک n را به صورت تصادفی تست کنیم. مثلاً برای یافتن یک عدد اول ۱۰۲۴بیتی تقریباً باید $\ln 2^{1024} \approx 710$ عدد ۱۰۲۴ بیتی تصادفی را برای اول بودن تست کنیم. (می‌توان با تست کردن فقط اعداد فرد، این تقریب را نصف کرد.)

در ادامه‌ی این بخش، مسئله‌ی تشخیص این که آیا یک عدد بزرگ n اول است یا خیر را در نظر می‌گیریم. برای سادگی در نمادگذاری، فرض می‌کنیم که تجزیه‌ی n به اعداد اول به صورت

$$n = p_1^{e_1} p_2^{e_2} \dots p_r^{e_r} \tag{۳۱-۳۹}$$

است، که در آن $r \geq 1$، $p_1, p_2, \dots, p_r$ فاکتورهای اول n، و $e_1, e_2, \dots, e_r$ اعداد صحیح هستند. مسلماً n

اول است اگر و فقط اگر $r = 1$ و $e_1 = 1$.

یک رویکرد ساده برای مسئله‌ی تست اول بودن، *تقسیم آزمایشی* (trial division) است. در این روش، سعی می‌کنیم n را به اعداد $2, 3, \ldots, \lfloor \sqrt{n} \rfloor$ تقسیم کنیم. (دوباره، می‌توان از اعداد زوج بزرگ‌تر از ۲ صرف نظر کرد.) به سادگی می‌توان دید که n اول است اگر و فقط اگر n به هیچ یک از این اعداد بخش‌پذیر نباشد. با فرض این که هر یک از این تقسیم‌ها به زمان ثابت نیاز دارد، بدترین حالت زمان اجرا $\theta(\sqrt{n})$ است، که نسبت به طول n از مرتبه‌ی نمایی است. (به خاطر بیاورید که اگر n به صورت دودویی با استفاده از β بیت کد شده باشد، آن گاه $\beta = \lceil \lg(n+1) \rceil$ و بنابراین $\sqrt{n} = \theta(2^{n/2})$.) بنابراین تقسیم آزمایشی فقط زمانی کار می‌کند که n بسیار کوچک باشد، و یا در حالتی که یک فاکتور اول بسیار کوچک داشته باشد. مزیت تقسیم آزمایشی این است که نه فقط تعیین می‌کند که آیا n اول است یا مرکب، بلکه در صورت مرکب بودن یکی از فاکتورهای اول آن را هم مشخص می‌کند.

در این بخش فقط می‌خواهیم اول بودن یک عدد داده شده‌ی n را تعیین کنیم؛ اگر n مرکب باشد پیدا کردن فاکتورهای اول آن برای ما اهمیتی ندارد. همان طور که در بخش ۳۱-۹ خواهیم دید، محاسبه‌ی تجزیه‌ی یک عدد به عوامل اول از نظر محاسباتی بسیار هزینه‌بر است. احتمالاً تعجب‌آور است که تعیین اول بودن یک عدد تا این حد ساده‌تر از تعیین فاکتورهای اول آن (در صورت مرکب بودن) است.

تست شبه اول بودن

اکنون یک متد برای تست اول بودن را در نظر می‌گیریم که «تقریباً کار می‌کند»، و در واقع برای بسیاری از کاربردها به اندازه‌ی کافی خوب است. یک اصلاح از این متد که این خطای کوچک را برطرف می‌کند، بعداً ارائه خواهد شد. فرض کنید $\mathbb{Z}_n^+$ نشان دهنده‌ی عناصر غیر صفر $\mathbb{Z}_n$ باشد:

$$\mathbb{Z}_n^+ = \{1, 2, \ldots, n-1\}$$

اگر n اول باشد، آن گاه $\mathbb{Z}_n^+ = \mathbb{Z}_n^*$.

می‌گوییم n *شبه اول با پایه‌ی* a است، اگر مرکب باشد و

$$a^{n-1} \equiv 1 \pmod{n} \tag{۳۱-۴۰}$$

قضیه‌ی فرما (قضیه‌ی ۳۱-۳۱) می‌گوید که اگر n اول باشد، آن گاه تساوی (۳۱-۳۸) را برای تمام aها در $\mathbb{Z}_n^+$ صدق می‌کند. بنابراین اگر بتوانیم یک $a \in \mathbb{Z}_n^+$ بیابیم به طوری که n تساوی (۳۱-۳۸) را ارضا *نمی‌کند*، آن گاه n حتماً مرکب است. عکس این قضیه به طرز تعجب‌آوری *تقریباً* برقرار است، و بنابراین، این ملاک یک تست تقریباً مناسب برای اول بودن فراهم می‌کند. تست می‌کنیم که آیا n تساوی (۳۱-۳۸) را برای $a = 2$ ارضا می‌کند یا خیر. اگر ارضا نکرد، n مرکب است. در غیر این صورت حدس می‌زنیم که n اول است (در حالی که می‌دانیم در واقع n یا اول است و یا یک شبه اول با پایه‌ی ۲).

رویه‌ی زیر به این روش وانمود می‌کند که اول بودن یا نبودن n را تست می‌کند. در این رویه از MODULAR-EXPONENTIATION از بخش ۳۱-۶ استفاده می‌شود. فرض می‌شود که ورودی n یک

عدد فرد بزرگ‌تر از ۲ است.

```
PSEUDOPRIME(n)
1  if MODULAR-EXPONENTIATION(2, n − 1, n) ≠ 1 (mod n)
2      return COMPOSITE        // definitely.
3  else return PRIME           // we hope!
```

این رویه می‌تواند خطا کند، ولی فقط به یک صورت. یعنی، اگر بگوید n مرکب است، در این صورت n حتماً مرکب است. ولی اگر بگوید n اول است، در این صورت خطا کرده است فقط اگر n یک شبه اول با پایه‌ی ۲ باشد.

این رویه به چه میزان اشتباه می‌کند؟ به شدت کم! فقط ۲۲ مقدار n کوچک‌تر از ۱۰,۰۰۰ وجود دارد که این رویه برای آن‌ها مرتکب خطا می‌شود؛ چهارتای اول این اعداد عبارتند از ۳۴۱، ۵۶۱، ۶۴۵، و ۱۱۰۵. می‌توان نشان داد که احتمال این که این روش بر روی یک عدد β بیتی به صورت تصادفی انتخاب شده وقتی $\beta \to \infty$، به سمت صفر میل می‌کند. با استفاده از تخمین‌های دقیق‌تر منسوب به Pomerance برای تعداد اعداد شبه اول با پایه‌ی ۲ با یک اندازه‌ی خاص، می‌توانیم تخمین بزنیم که یک عدد ۵۱۲ بیتی به صورت تصادفی انتخاب شده، با احتمال کم‌تر از یک در 10^{20} شبه اول با پایه‌ی ۲ است، و یک عدد ۱۰۲۴ بیتی به صورت تصادفی انتخاب شده که به این روش اول اعلام شده است، با احتمال کم‌تر از یک در 10^{41} شبه اول با پایه‌ی ۲ است. بنابراین اگر صرفا می‌خواهید یک عدد اول بزرگ برای یک کاربرد بیابید، برای تمام اهداف کاربردی، اگر اعداد را به صورت تصادفی انتخاب کنید و به PSEUDOPRIME بدهید تا زمانی که PRIME بودن آن را اعلام کند، تقریباً هیچ گاه اشتباه نمی‌کنید. ولی وقتی اعدادی که تست می‌شوند، به صورت تصادفی انتخاب نشده‌اند، به یک رویکرد بهتر برای تست اول بودن نیاز داریم. همان طور که خواهیم دید، مقداری ذکاوت و مقداری تصادفی کردن، به یک روش برای تست اول بودن می‌رسیم که برای تمام اعداد اول به خوبی کار می‌کند.

متأسفانه با چک کردن برای شبه اول بودن با یک پایه‌ی دوم، مثلاً $a = 3$، نمی‌توانیم تمام خطاها را از بین ببریم، چون اعداد مرکبی مانند n وجود دارند که تساوی (۳۱-۴۰) را برای *تمام* $a \in \mathbb{Z}_n^+$ ارضا می‌کنند. این اعداد به *اعداد کارمایکل* (Carmichael number) معروفند. سه عدد اول کارمایکل عبارتند از ۵۶۱، ۱۱۰۵ و ۱۷۲۹. اعداد کارمایکل به شدت کم‌یابند؛ مثلاً فقط ۲۵۵ عدد کارمایکل کوچک‌تر از ۱۰۰,۰۰۰,۰۰۰ وجود دارد. تمرین ۳۱-۸-۲ به شما کمک می‌کند توضیح دهید که چرا آن‌ها این قدر نادرند.

اکنون نشان می‌دهیم چگونه می‌توان تست اول بودن را طوری بهبود بخشیم که توسط اعداد کارمایکل فریب نخورد.

تست اول بودن تصادفی Miller-Rabin

تست اول بودن Miller-Rabin با دو اصلاح بر مشکلات تست PSEUDOPRIME غلبه می‌کند:

- این تست به جای این که فقط یک عدد را به عنوان مقدار پایه آزمایش کند، چندین مقدار تصادفی a را بررسی می‌کند.

- هنگام محاسبه‌ی هر توان پیمانه‌ای، توجه می‌کند که آیا یک ریشه‌ی دوم نابدیهی ۱ به پیمانه‌ی n در میان آخرین مربع‌گیری‌ها یافت شده است یا خیر. اگر این گونه بود، رویه پایان می‌یابد و COMPOSITE را به خروجی می‌دهد. نتیجه‌ی ۳۱-۳۵ از بخش ۳۱-۶ تعیین اعداد مرکب به این روش را تأیید می‌کند.

شبه‌کد تست اول بودن Miller-Rabin در ادامه می‌آید. ورودی $n > ۲$ یک عدد فرد است که باید اول بودن آن بررسی شود، و s تعداد پایه‌های به صورت تصادفی انتخاب شده از $\mathbb{Z}_n^+$ است. این کد از تولید کننده‌ی اعداد تصادفی RANDOM توصیف شده در بخش ۵-۱ استفاده می‌کند: RANDOM $(۱, n-۱)$ یک عدد تصادفی a به دست می‌دهد به طور یکه $۱ \le a \le n-۱$. در کد از یک رویه‌ی کمکی WITNESS استفاده شده است به طوری که WITNESS (a,n) مقدار TRUE را بازمی‌گرداند اگر و فقط اگر a یک «شاهد» (witness) برای مرکب بودن n باشد – یعنی، اگر بتوان با استفاده از a (با استفاده از روشی که خواهیم دید) اثبات کرد که n مرکب است. تست WITNESS (a,n) یک نسخه‌ی گسترش یافته، ولی مؤثرتر از تست

$$a^{n-۱} \not\equiv ۱ \pmod{n}$$

است، که PSEUDOPRIME (با استفاده از $a = ۲$) بر پایه‌ی آن عمل می‌کرد. ابتدا ساختار WITNESS را ارائه کرده و صحت آن را بررسی می‌کنیم، و سپس نشان می‌دهیم که چگونه از آن در تست اول بودن Miller-Rabin استفاده می‌شود. فرض کنید $n-۱ = ۲^t u$، که در آن $t \ge ۱$ و u فرد است؛ یعنی نمایش دودویی $n-۱$ به صورت نمایش دودویی عدد فرد u است، و در ادامه‌ی آن دقیقاً t صفر. بنابراین $a^{n-۱} \equiv (a^u)^{۲^t} \pmod{n}$، به طوری که برای محاسبه‌ی $a^{n-۱} \bmod n$ می‌توانیم ابتدا $a^u \bmod n$ را محاسبه کرده و سپس بر روی نتیجه t بار مربع‌گیری انجام دهیم.

```
WITNESS(a, n)
1  let n − ۱ = ۲^t u , where t ≥ 1 and u is odd
2  x_۰ = MODULAR-EXPONENTIATION(a, u, n)
3  for i = 1 to t
4      x_i = x_{i−۱}^۲ mod n
5      if x_i == ۱ and x_{i−۱} ≠ ۱ and x_{i−۱} ≠ n − ۱
6          return TRUE
7  if x_t ≠ ۱
8      return TRUE
9  return FALSE
```

شبه‌کد WITNESS برای محاسبه‌ی $a^{n-۱} \bmod n$، ابتدا مقدار $x^۰ = a^u \bmod n$ را در خط ۲ محاسبه می‌کند، و سپس در حلقه‌ی for خطوط ۳-۶ از نتیجه t بار پشت سر هم مربع‌گیری می‌کند. طبق استقرا بر روی i، دنباله‌ی $x_۰, x_۱, \ldots, x_t$ از مقادیر محاسبه شده، تساوی $x_i \equiv a^{۲^i u} \pmod{n}$ را برای $i = ۰, ۱, \ldots, t$ ارضا می‌کند، که به طور خاص $x_t \equiv a^{n-۱} \pmod{n}$. با این حال هر گاه یک مرحله‌ی مربع‌گیری در خط ۴ انجام می‌شود، در صورتی که خطوط در ۵-۶ مشخص شود که یک ریشه‌ی دوم

نابدیهی ۱ پیدا شده است، حلقه زود هنگام پایان می‌یابد. (به زودی در مورد این تست‌ها توضیح خواهیم داد.) اگر چنین شود، الگوریتم پایان می‌یابد و TRUE را بازمی‌گرداند. خطوط ۷–۸ مقدار TRUE را بازمی‌گردانند اگر مقدار محاسبه شده برای $x_t \equiv a^{n-1} \pmod n$ برابر با ۱ نباشد، همان طور که رویه‌ی PSEUDOPRIME در این حالت COMPOSITE را بازمی‌گرداند. اگر در خطوط ۶ یا ۸ مقدار TRUE بازگردانده نشده باشد، خط ۹ مقدار FALSE را بازمی‌گرداند.

اکنون بحث می‌کنیم که اگر WITNESS (a,n) مقدار TRUE را بازگرداند، آن گاه می‌توان با استفاده از a اثبات کرد که n مرکب است.

اگر WITNESS در خط ۸ مقدار TRUE را بازگرداند، بدین معنی است که $x_t = a^{n-1} \bmod n \neq 1$. از طرفی اگر n اول باشد، طبق قضیه‌ی فرما (قضیه‌ی ۳۱–۳۱) داریم $a^{n-1} \not\equiv 1 \pmod n$ برای تمام $a \in \mathbb{Z}_n^+$. بنابراین n نمی‌تواند اول باشد، و تساوی $a^{n-1} \bmod n \neq 1$ یک اثبات برای این واقعیت است.

اگر WITNESS در خط ۶ مقدار TRUE را بازگرداند، این بدین معنی است که x_{i-1} یک ریشه‌ی دوم نابدیهی ۱ $x_i = 1$ به پیمانه‌ی n است، چرا که داریم $x_{i-1} \not\equiv \pm 1 \pmod n$ با این که $x_i \equiv x_{i-1}^2 \pmod n$. نتیجه‌ی ۳۱–۳۵ می‌گوید فقط در صورتی که n مرکب باشد، یک ریشه‌ی دوم نابدیهی ۱ به پیمانه‌ی n وجود دارد، پس تشخیص این که x_{i-1} یک ریشه‌ی دوم نابدیهی ۱ به پیمانه‌ی n است، اثباتی است برای مرکب بودن n.

در این جا اثبات درستی WITNESS کامل می‌شود. اگر فراخوانی WITNESS (a,n) مقدار TRUE را بازگرداند، آن‌گاه n مطمئنا مرکب است، و اثبات این قضیه را می‌توان به سادگی به کمک n و a انجام داد.

در این جا مختصراً یک توصیف دیگر از رفتار WITNESS به عنوان تابعی از دنباله‌ی $X = \langle x_0, x_1, \ldots, x_t \rangle$ ارائه می‌کنیم، که بعداً هنگام تحلیل کارایی تست Miller-Rabin مفید خواهد بود. توجه کنید که اگر $x_i = 1$ برای یک $0 \le i < t$، WITNESS ممکن است تست بقیه‌ی دنباله را ادامه ندهد. از طرفی اگر این کار را انجام دهد، آن گاه مقدار هر یک عناصر $x_{i+1}, x_{i+2}, \ldots, x_t$ برابر با ۱ خواهد بود. چهار حالت داریم:

۱. $X = \langle \ldots, d \rangle$، که در آن $d \neq 1$: دنباله‌ی X با ۱ پایان نمی‌یابد. TRUE در خط ۸ بازگردانده می‌شود؛ a شاهدی برای مرکب بودن n است (طبق قضیه‌ی فرما).

۲. $X = \langle 1, 1, \ldots, 1 \rangle$: دنباله‌ی X تماماً ۱ است. FALSE بازگردانده می‌شود؛ a نمی‌تواند شاهدی برای مرکب بودن n باشد.

۳. $X = \langle \ldots, -1, 1, \ldots, 1 \rangle$: دنباله‌ی X با ۱ پایان می‌یابد، و آخرین عنصر غیر ۱ برابر با ۱– است. FALSE در خط ۶ بازگردانده می‌شود؛ a نمی‌تواند شاهدی برای مرکب بودن n باشد.

۴. $X = \langle \ldots, d, 1, \ldots, 1 \rangle$، که در آن $d \neq \pm 1$: دنباله‌ی X با ۱ پایان می‌یابد، ولی آخرین عنصر غیر ۱ برابر با ۱– نیست. TRUE بازگردانده می‌شود؛ a شاهدی است برای مرکب بودن n، چرا که d یک ریشه‌ی دوم نابدیهی ۱ است.

اکنون تست اول بودن Miller-Rabin را که برمبنای رویه‌ی WITNESS است بررسی می‌کنیم. دوباره، فرض می‌کنیم که n یک عدد فرد بزرگ‌تر از ۲ است.

```
MILLER-RABIN(n, s)
1  for j = 1 to s
2      a = RANDOM(1, n − 1)
3      if WITNESS(a, n)
4          return COMPOSITE        // definitely.
5  return PRIME                    // almost surely.
```

رویه‌ی MILLER-RABIN یک جستجوی احتمالاتی برای اثبات مرکب بودن n است. حلقه‌ی اصلی (که در خط ۱ آغاز می‌شود) s مقدار تصادفی a از $\mathbb{Z}_n^+$ (در خط ۲) انتخاب می‌کند. اگر یکی از aهای انتخاب شده، شاهدی برای مرکب بودن n باشد، آن گاه MILLER-RABIN در خط ۴ مقدار COMPOSITE را بازمی‌گرداند. چنین خروجی همیشه درست است، چرا که WITNESS در این حالت درست است. اگر در s تلاش، هیچ شاهدی یافت نشود، MILLER-RABIN فرض می‌کند که این بدین خاطر است که هیچ شاهدی وجود ندارد، و می‌توانیم فرض کنیم که n اول است. خواهیم دید که اگر s به اندازه‌ی کافی بزرگ باشد، این خروجی با احتمال زیاد صحیح است، ولی احتمال بسیار کمی وجود دارد که رویه در انتخاب aها بدشانس باشد و با این که هیچ شاهدی یافت نشده است، ولی این شاهد وجود داشته باشد.

برای مشخص شدن عملیات MILLER-RABIN، فرض کنید n عدد کارمایکل ۵۶۱ باشد، به طوری که $n-1=۵۶۰=۲^۴.۳۵$، $t=۴$ و $u=۳۵$. با فرض این که $a=۷$ به عنوان پایه انتخاب شده است، شکل ۳۱-۴ نشان می‌دهد که WITNESS مقدار $x_۰=a^{۳۵}=۲۴۱ \pmod n$ را محاسبه می‌کند، و همچنین دنباله‌ی $X=\langle ۲۴۱, ۲۹۸, ۱۶۶, ۶۷, ۱\rangle$ را. بنابراین یک ریشه‌ی دوم نابدیهی ۱ در آخرین مرحله‌ی مربع‌گیری یافت می‌شود، چرا که $a^{۲۸۰}\equiv ۶۷ \pmod n$ و $a^{۵۶۰}\equiv ۱ \pmod n$. بنابراین $a=۷$ شاهدی است برای مرکب بودن n، WITNESS $(۷,n)$ مقدار TREU را بازمی‌گرداند، و MILLER-RABIN مقدار COMPOSITE را.

اگر n یک عدد β بیتی باشد MILLER-RABIN به $O(s\beta)$ عملیات ریاضی و $O(s\beta^۳)$ عملیات بیتی نیاز دارد، چرا که به صورت حدی بیش از s عملیات پیمانه‌ای در آن انجام می‌شود.

نرخ خطای تست اول بودن Miller-Rabin

اگر MILLER-RABINN مقدار PRIME را بازگرداند، آن گاه احتمال بسیار کمی وجود دارد که اشتباه کرده باشد. با این حال، برخلاف PSEUDOPRIME، احتمال خطا به n بستگی ندارد؛ هیچ ورودی بدی برای این رویه وجود ندارد. در عوض، این احتمال به اندازه‌ی s و «شانس قرعه کشی» در انتخاب مقادیر a وابسته است. همچنین، چون تست‌های انجام شده دقیق‌تر از چک ساده‌ی تساوی (۳۱-۳۸) است، می‌توانیم انتظار داشته باشیم که در اصول کلی، نرخ خطا برای nهای به صورت تصادفی انتخاب شده کوچک باشد. قضیه‌ی زیر بحثی دقیق‌تر در این مورد ارائه می‌کند.

قضیه‌ی ۳۱-۲۷ اگر n یک عدد مرکب فرد باشد، آن گاه تعداد شاهدها برای مرکب بودن n حداقل $(n-۱)/۲$ است.

اثبات در این اثبات نشان می‌دهیم که تعداد اعداد غیر شاهد حداکثر $(n-1)/2$ است، که قضیه را اثبات می‌کند.

با این ادعا شروع می‌کنیم که هر عدد غیر شاهد باید عضوی از $\mathbb{Z}_n^+$ باشد. چرا؟ یک عدد غیر شاهد a را در نظر بگیرید. این عدد باید در $a^{n-1} \equiv 1 \pmod n$ صدق کند، یا به طور معادل، $a.a^{n-2} \equiv 1 \pmod n$. بنابراین یک جواب برای معادله‌ی $ax \equiv 1 \pmod n$ وجود دارد، که a^{n-2} است. طبق نتیجه‌ی ۳۱-۲۱، $\gcd(a,n) = 1$، که خود نتیجه می‌دهد که $\gcd(a,n) = 1$. بنابراین a عضوی از $\mathbb{Z}_n^+$ است؛ یعنی تمام اعداد غیر شاهد به $\mathbb{Z}_n^+$ تعلق دارند.

برای تکمیل اثبات، نشان می‌دهیم که نه تنها تمام اعداد غیر شاهد در $\mathbb{Z}_n^+$ هستند، بلکه همه در یک زیرگروه اکید B از $\mathbb{Z}_n^+$ قرار دارند (به خاطر بیاورید که می‌گوییم B یک زیرگروه *اکید* از $\mathbb{Z}_n^+$ است اگر B یک زیرگروه از $\mathbb{Z}_n^+$ باشد ولی این دو با هم برابر نباشند). سپس طبق نتیجه‌ی ۳۱-۱۶ خواهیم داشت $|B| \le \left|\mathbb{Z}_n^+\right|/2$. چون $\left|\mathbb{Z}_n^+\right| \le n-1$، به دست می‌آوریم $|B| \le (n-1)/2$. بنابراین تعداد اعداد غیر شاهد حداکثر $(n-1)/2$ است، و از این رو تعداد شاهدها باید حداقل $(n-1)/2$ باشد.

اکنون نشان می‌دهیم که چطور می‌توان یک زیرگروه اکید B از $\mathbb{Z}_n^+$ یافت که حاوی تمام اعداد غیر شاهد باشد. اثبات را به دو حالت تقسیم می‌کنیم.

حالت ۱: یک $x \in \mathbb{Z}_n^+$ وجود دارد به طوری که

$$x^{n-1} \not\equiv 1 \pmod n$$

به عبارت دیگر، n یک عدد کارمایکل نیست. چون، همان طور که قبلاً گفتیم، اعداد کارمایکل بسیار نادر هستند، حالت ۱ حالت اصلی است که «در عمل» پیش می‌آید (یعنی، وقتی n به صورت تصادفی انتخاب و اول بودن آن تست شده است).

فرض کنید $B = \{b \in \mathbb{Z}_n^+ : b^{n-1} \equiv 1 \pmod n\}$. بدیهتاً B تهی نیست، چرا که $1 \in B$. چون B تحت ضرب به پیمانه‌ی n بسته است، پس طبق قضیه‌ی ۳۱-۱۴، B یک زیرگروه از $\mathbb{Z}_n^+$ است. توجه کنید که تمام اعداد غیر شاهد متعلق به B هستند، چرا که یک عدد غیر شاهد a در رابطه‌ی $a^{n-1} \equiv 1 \pmod n$ صدق می‌کند. چون $x \in \mathbb{Z}_n^+ - B$، پس B یک زیرگروه اکید از $\mathbb{Z}_n^+$ خواهد بود.

حالت ۲: برای تمام $x \in \mathbb{Z}_n^+$،

$$x^{n-1} \equiv 1 \pmod n \qquad (۳۱-۴۱)$$

به عبارت دیگر، n یک عدد کارمایکل است. این حالت در عمل بسیار نادر است. با این حال، تست Miller-Rabin (برخلاف تست اعداد شبه اول) می‌تواند به شکلی کارآمد مرکب بودن اعداد کارمایکل را تشخیص دهد، همان طور که در ادامه خواهیم دید.

در این حالت، n نمی‌تواند یک توان اول باشد. برای این که ببینیم چرا، فرض کنید که خلاف این را در نظر بگیریم، یعنی $n = p^e$، که در آن p یک عدد اول است و $e > 1$. به صورت زیر به تناقض می‌رسیم. چون فرض کردیم n فرد است، p هم باید فرد باشد. قضیه‌ی ۳۱-۳۲ ایجاب می‌کند که $\mathbb{Z}_n^+$

یک گروه تناوبی باشد: این گروه حاوی یک سازنده‌ی g است به طوری که $\mathrm{ord}_n(g) = \left|\mathbb{Z}_n^*\right| = \phi(n) =$ $p^e(1-1/p) = (p-1)p^{e-1}$. (فرمول $\phi(n)$ از تساوی (۳۱-۲۰) می‌آید.) طبق تساوی (۳۱-۴۱) داریم $g^{n-1} \equiv 1 \pmod n$. سپس قضیه‌ی لوگاریتم گسسته (قضیه‌ی ۳۱-۳۳، با فرض $y = 0$) می‌گوید $n-1 \equiv 0 \pmod{\varphi(n)}$، یا

$$(p-1)p^{e-1} \mid p^e - 1$$

این برای $e > 1$ تناقض است، چرا که $(p-1)p^e - 1$ بر عدد اول p بخش‌پذیر است، ولی $p^e - 1$ این طور نیست. بنابراین n توانی از یک عدد اول نیست.

چون عدد مرکب و فرد n توانی از یک عدد اول نیست، آن را به ضرب $n_1 n_2$ تجزیه می‌کنیم، که در آن n_1 و n_2 اعداد فرد بزرگ‌تر از ۱ هستند که نسبت به هم اولند. (ممکن است روش‌های زیادی برای این کار باشد، و این که ما کدام یک را انتخاب می‌کنیم اهمیتی ندارد. مثلاً اگر $n = p_1^{e_1} p_2^{e_2} \ldots p_r^{e_r}$، آن گاه می‌توانیم انتخاب کنیم $n_1 = p_1^{e_1}$ و $n_2 = p_2^{e_2} p_3^{e_3} \ldots p_r^{e_r}$.)

به خاطر بیاورید که t و u را طوری تعریف کردیم که $n - 1 = 2^t u$، که در آن $t \geq 1$ و u فرد است، و به طوری که برای یک ورودی a، رویه‌ی WITNESS دنباله‌ی

$$X = \left\langle a^u, a^{2u}, a^{2^2 u}, \ldots, a^{2^t u} \right\rangle$$

را محاسبه می‌کند (تمام محاسبات به پیمانه‌ی n انجام می‌شوند).

اجازه دهید جفتی از اعداد صحیح (v, j) را *قابل قبول* بنامیم اگر $v \in \mathbb{Z}_n^+$، $j \in \{0, 1, \ldots, t\}$، و

$$v^{2^j u} \equiv -1 \pmod n$$

جفت‌های قابل قبول مطمئنا وجود دارد چرا که u فرد است؛ می‌توانیم انتخاب کنیم $v = n - 1$ و $j = 0$، به طوری که به جفت قابل قبول $(n-1, 0)$ برسیم. اکنون بزرگ‌ترین j ممکن را انتخاب می‌کنیم به طوری که یک جفت قابل قبول (v, j) وجود داشته باشد، و سپس v را برای این جفت می‌یابیم. فرض کنید

$$B = \{x \in \mathbb{Z}_n^+ : x^{2^j u} \equiv \pm 1 \pmod n\}$$

چون B تحت ضرب به پیمانه‌ی n بسته است، یک زیرگروه از $\mathbb{Z}_n^+$ خواهد بود. بنابراین طبق قضیه‌ی ۳۱-۱۵، $|B|$، $\left|\mathbb{Z}_n^+\right|$ را می‌شمارد. تمام اعداد غیر شاهد باید عضوی از B باشند، چرا که دنباله‌ی X که توسط یک عدد غیر شاهد ساخته شده است، یا باید تماماً ۱ باشد و یا حاوی یک -1 قبل از مکان jام باشد، طبق ماکسیمم بودن j. (اگر (a, j') قابل قبول باشد، که در آن a یک عدد غیر شاهد است، به دلیل انتخاب خود از j باید داشته باشیم $j' < j$.)

اکنون از وجود v استفاده می‌کنیم تا نشان دهیم که یک $w \in \mathbb{Z}_n^+ - B$ وجود دارد، و بنابراین B یک زیرگروه اکید $\mathbb{Z}_n^*$ است. چون $v^{2^j u} \equiv -1 \pmod n$، داریم $v^{2^j u} \equiv -1 \pmod{n_1}$ طبق نتیجه‌ی ۳۱-۲۹ از قضیه‌ی باقی‌مانده‌ی چینی. طبق نتیجه‌ی ۳۱-۲۸، یک w وجود دارد که هم زمان تساوی‌های زیر را

ارضا می‌کند:

$$w \equiv v \pmod{n_1},$$
$$w \equiv 1 \pmod{n_2}$$

بنابراین

$$w^{2^j u} \equiv -1 \pmod{n_1},$$
$$w^{2^j u} \equiv 1 \pmod{n_2}$$

طبق قضیه‌ی ۳۱-۲۹، $w^{2^j u} \not\equiv 1 \pmod{n_1}$ نتیجه می‌دهد که $w^{2^j u} \not\equiv 1 \pmod{n}$، و $w^{2^j u} \not\equiv -1 \pmod{n_2}$ نتیجه می‌دهد که $w^{2^j u} \not\equiv -1 \pmod{n}$. بنابراین $w^{2^j u} \not\equiv \pm 1 \pmod{n}$، و از این رو $w \notin B$.

این باقی می‌ماند که نشان دهیم $w \in \mathbb{Z}_n^+$، که برای این کار به صورت جداگانه به پیمانه‌های n_1 و n_2 کار می‌کنیم. با کار به پیمانه‌ی n_1، مشاهده می‌کنیم که چون $v \in \mathbb{Z}_n^+$، داریم $\gcd(v, n) = 1$، و همچنین $\gcd(v, n_1) = 1$؛ اگر v هیچ مقسوم‌علیه مشترکی با n نداشته باشد، مطمئنا هیچ مقسوم‌علیه مشترکی با n_1 هم ندارد. چون $w \equiv v \pmod{n_1}$، می‌بینیم که $\gcd(w, n_1) = 1$. با کار به پیمانه‌ی n_2، مشاهده می‌کنیم که $w \equiv 1 \pmod{n_2}$ ایجاب می‌کند که $\gcd(w, n_2) = 1$. برای ترکیب این نتایج از قضیه‌ی ۳۱-۶ استفاده می‌کنیم، که ایجاب می‌کند که $\gcd(w, n_1 n_2) = \gcd(w, n) = 1$. یعنی $w \in \mathbb{Z}_n^+$.

بنابراین $w \in \mathbb{Z}_n^+ - B$، و حالت ۲ را با این نتیجه‌گیری پایان می‌دهیم که B یک زیرگروه اکید از $\mathbb{Z}_n^+$ است.

در هر دو حالت، می‌بینیم که تعداد شاهدها برای مرکب بودن n حداقل $(n-1)/2$ است. □

قضیه‌ی ۳۱-۳۹ برای هر عدد فرد $n > 2$ و عدد صحیح مثبت s، احتمال این که MILLER-RABIN(n, s) مرتکب خطا شود حداکثر 2^{-s} است.

اثبات با استفاده از قضیه‌ی ۳۱-۳۸، می‌بینیم که اگر n مرکب باشد، آن گاه هر اجرای حلقه‌ی for خطوط ۱-۴ با احتمال $1/2$ یک شاهد x برای مرکب بودن n می‌یابد. MILLER-RABIN فقط در صورتی مرتکب خطا می‌شود به حدی بدشانسی بیاورد که در تمام s تکرار حلقه‌ی اصلی، نتواند شاهدی برای مرکب بودن n بیابد. احتمال چنین چیزی حداکثر 2^{-s} است. □

اگر n اول باشد، خروجی MILLER-RABIN همیشه PRIME خواهد بود، و اگر n مرکب باشد احتمال این که MILLER-RABIN مقدار PRIME را به خروجی بدهد حداکثر برابر است با 2^{-s}.

ولی هنگام اعمال MILLER-RABIN به یک عدد تصادفی بزرگ n، باید از قبل احتمال اول بودن n را در نظر بگیریم تا بتوانیم به درستی نتیجه‌ی MILLER-RABIN را تفسیر کنیم. فرض کنید که از قبل یک β در نظر می‌گیریم و سپس به صورت تصادفی یک عدد صحیح n به طول β بیت برای تست اول بودن انتخاب می‌کنیم. فرض کنید A نشان‌دهنده‌ی رخداد اول بودن n باشد. طبق قضیه‌ی اعداد

اول (قضیه‌ی ۳۱-۳۷)، احتمال این که n اول باشد تقریباً برابر است با

$$\begin{aligned}\Pr\{A\} &\approx 1/\ln n \\ &\approx 1{,}4433/\beta\end{aligned}$$

اکنون فرض کنید B نشان‌دهنده‌ی این رخداد باشد که MILLER-RABIN مقدار PRIME را بازگرداند. داریم $\Pr\{\bar{B} \mid A\} = 0$ (یا به طور مشابه، $\Pr\{B \mid A\} = 1$) و $\Pr\{B \mid \bar{A}\} \le 2^{-s}$ (یا به طور مشابه $\Pr\{\bar{B} \mid \bar{A}\} > 1 - 2^{-s}$).

اکنون می‌خواهیم مقدار $\Pr\{A \mid B\}$ را بدانیم (که برابر است با احتمال این که n اول باشد، در حالی که رویه‌ی MILLER-RABIN مقدار PRIME را بازگردانده است). طبق قضیه‌ی بیز (تساوی (پ-۱۸)) داریم

$$\begin{aligned}\Pr\{A \mid B\} &= \frac{\Pr\{A\}\Pr\{B \mid A\}}{\Pr\{A\}\Pr\{B \mid A\} + \Pr\{\bar{A}\}\Pr\{B \mid \bar{A}\}} \\ &\approx \frac{1}{1 + 2^{-s}(\ln n - 1)}\end{aligned}$$

این احتمال از ۱/۲ تجاوز نمی‌کند مگر این که s از $\lg(\ln n - 1)$ فراتر رود. به طور شهودی، به همین مقدار تلاش اولیه‌ی ناموفق در یافتن یک شاهد برای مرکب بودن n نیاز داریم تا بتوانیم به آسودگی مرکب بودن n را رد کنیم. برای یک عدد $\beta = 1024$ بیتی، این تست اولیه حدود

$$\begin{aligned}\lg(\ln n - 1) &\approx \lg(\beta/1{,}4433) \\ &\approx 9\end{aligned}$$

تلاش نیاز دارد. تحت هر شرایطی انتخاب $s = 50$ باید برای هر کاربرد قابل تصوری کافی باشد.

در واقع شرایط از این هم بسیار بهتر است. اگر بخواهیم با استفاده از MILLER-RABIN بر روی اعداد تصادفی بزرگ فرد، اعداد اول بزرگ انتخاب کنیم، آن گاه انتخاب یک مقدار کوچک برای s (مثلاً ۳) با احتمال بسیار کم نتیجه‌ی نادرست تولید می‌کند (هر چند که در این جا آن را اثبات نمی‌کنیم). این بدین خاطر است که برای یک عدد فرد مرکب تصادفی n، امیدریاضی تعداد اعداد غیرشاهد برای مرکب بودن n احتمالاً بسیار کوچک‌تر از $(n-1)/2$ است.

ولی اگر n به صورت تصادفی انتخاب نشده باشد، بهترین چیزی که می‌توان اثبات کرد این است که تعداد اعداد غیر شاهد حداکثر برابر است با $(n-1)/4$ (با استفاده از نسخه‌ای بهبود یافته از قضیه‌ی ۳۱-۳۸). به علاوه اعدادی مانند n وجود دارند که تعداد اعداد غیرشاهد آن‌ها واقعاً برابر با $(n-1)/4$ است.

تمرین‌ها

۳۱-۸-۱ اثبات کنید که اگر یک عدد فرد $n > 1$ اول و یا توانی از یک عدد اول نباشد، آن گاه یک ریشه‌ی دوم نابدیهی ۱ به پیمانه‌ی n وجود دارد.

★ ۳۱-۸-۲ می‌توان قضیه‌ی اویلر را مقداری قوی‌تر کرد، بدین صورت:

$a^{\lambda(n)} \equiv 1 \pmod{n}$ برای تمام $a \in \mathbb{Z}_n^*$

که در آن $n = p_1^{e_1} p_2^{e_2} \ldots p_r^{e_r}$ و $\lambda(n)$ بدین صورت تعریف شده است:

$$\lambda(n) = \text{lcm}(\phi(p_1^{e_1}), \ldots, \phi(p_r^{e_r})) \qquad (۳۱-۴۲)$$

اثبات کنید که $\lambda(n) | \phi(n)$. یک عدد مرکب، کارمایکل است اگر $\lambda(n) | n - 1$. کوچک‌ترین عدد کارمایکل $۵۶۱ = ۳۰ \cdot ۱۱ \cdot ۱۷$ است؛ در این جا $\lambda(n) = \text{lcm}(۲, ۱۰, ۱۶) = ۸۰$، که ۵۶۰ را می‌شمارد. اثبات کنید که اعداد کارمایکل هم باید «بدون مربع» (تقسیم ناپذیر بر مربع هیچ عدد اولی) باشند، و هم حاصل ضرب حداقل سه عدد اول. به همین دلیل، این اعداد چندان رایج نیستند.

۳-۸-۳۱ اثبات کنید اگر x یک ریشه‌ی نابدیهی ۱ به پیمانه‌ی n باشد، آن گاه $\gcd(x - ۱, n)$ و $\gcd(x + ۱, n)$ هر دو مقسوم‌علیه‌های نابدیهی n هستند.

۳۱-۹ تجزیه‌ی اعداد صحیح

فرض کنید یک عدد صحیح n داریم و می‌خواهیم آن را تجزیه کنیم، یعنی، آن را به ضرب اعداد اول تبدیل کنیم. تست اول بودن در بخش قبل می‌تواند به ما بگوید که n مرکب است، ولی معمولاً فاکتورهای اول n را به ما نمی‌گوید. به نظر می‌آید که تجزیه‌ی یک عدد بزرگ n بسیار سخت‌تر از تشخیص اول یا مرکب بودن آن باشد. با ابرکامپیوترهای امروزه و بهترین الگوریتم موجود، می‌توان اعداد ۱۰۲۴ بیتی را هم تجزیه کرد.

مکاشفه‌ی رو-ی پولارد

تقسیم بر تمام اعداد تا R تضمین می‌کند که تجزیه‌ی هر عددی تا R^2 را به ما بدهد. با همین میزان کار، رویه‌ی زیر هر عدد تا R^4 را تجزیه می‌کند (مگر این که بدشانس باشیم). از آن جایی که این رویه فقط یک مکاشفه است، نه زمان اجرای آن تضمین شده است و نه موفقیت آن، ولی با این حال، این رویه در عمل بسیار مؤثر است. یک مزیت دیگر رویه‌ی POLLARD-RHO این است که از مقدار ثابتی حافظه استفاده می‌کند. (می‌توان به سادگی این الگوریتم را بر روی یک ماشین‌حساب جیبی قابل برنامه‌ریزی پیاده و اعداد کوچک را تجزیه کرد.)

```
POLLARD-RHO(n)
1   i = 1
2   x_1 = RANDOM(0, n − 1)
3   y = x_1
4   k = 2
5   while TRUE
6       i = i + 1
7       x_i = (x_{i−1}^2 − 1) mod n
8       d = gcd(y − x_i, n)
9       if d ≠ 1 and d ≠ n
10          print d
```

```
11      if i = k
12          y = x_i
13          k = 2k
```

رویه به صورت زیر کار می‌کند. خطوط ۱-۲ متغیر i را با ۱ و x_1 را با یک مقدار تصادفی در $\mathbb{Z}_n$ مقداردهی اولیه می‌کنند. تکرار حلقه‌ی while که در خط ۵ آغاز می‌شود بدون شرط است (یعنی برای همیشه)، و به دنبال فاکتورهای n می‌گردد. حین هر تکرار حلقه‌ی while از بازگشت

$$x_i = (x_{i-1}^2 - 1) \bmod n \qquad (۳۱-۴۳)$$

در خط ۷ استفاده می‌شود تا مقدار بعدی x_i در دنباله‌ی نامتناهی

$$x_1, x_2, x_3, x_4, \ldots; \qquad (۳۱-۴۴)$$

ساخته شود؛ مقدار i متناظرا در خط ۶ افرایش می‌یابد. در کد برای وضوح بیشتر از مقادیر x_i با اندیس استفاده شده است، ولی اگر تمام اندیس‌ها را برداریم، باز هم برنامه کار می‌کند، چرا که فقط نیاز داریم آخرین مقدار x_i را نگه داریم. با این اصلاح، برنامه فقط از مقدار ثابتی حافظه استفاده می‌کند.

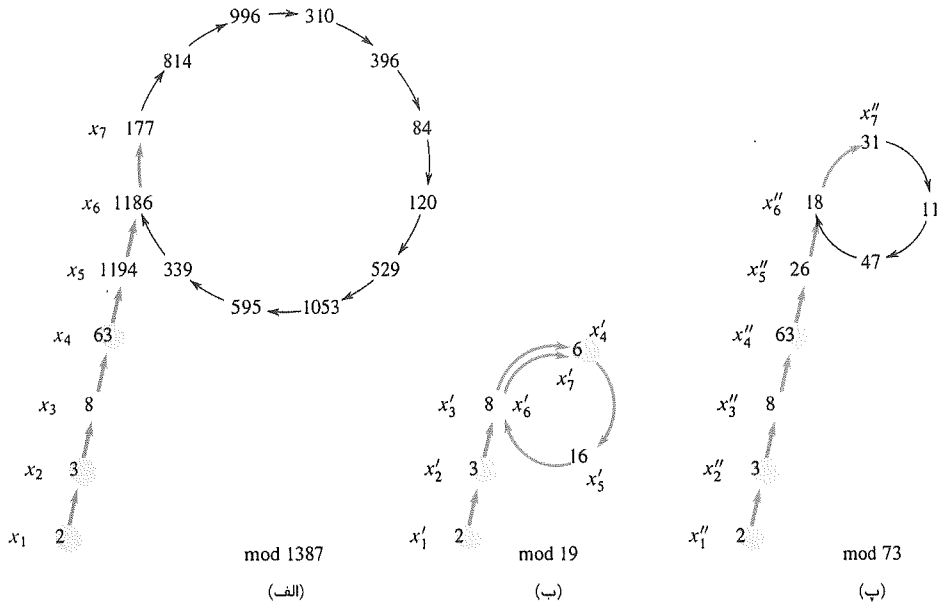
هر از گاهی، برنامه آخرین مقدار ساخته شده‌ی x_i را در متغیر y ذخیره می‌کند. به طور خاص، مقادیری که ذخیره می‌شوند آن‌هایی هستند که اندیس آن‌ها توانی از ۲ است:

$$x_1, x_2, x_4, x_8, x_{16}, \ldots$$

خط ۳ مقدار x_1 را ذخیره می‌کند، و خط ۱۲ مقدار x_i را، هر گاه که i برابر با k باشد. در خط ۲، متغیر k با ۲ مقداردهی اولیه می‌شود، و در خط ۱۳، اگر y به هنگام سازی شود، k هم دو برابر می‌شود. بنابراین k دنباله‌ی $1, 2, 4, 8, \ldots$ را دنبال می‌کند، و همیشه حاوی اندیس مقدار بعدی x_i است که باید در y ذخیره شود.

خطوط ۸-۱۰ سعی می‌کنند که با استفاده از مقدار ذخیره شده‌ی y و مقدار فعلی x_i یک فاکتور از n را بیابند. به طور خاص، خط ۸ بزرگ‌ترین مقسوم‌علیه مشترک $d = \gcd(y - x_i, n)$ را محاسبه می‌کند. اگر d یک مقسوم‌علیه نابدیهی n باشد (که در خط ۹ چک می‌شود)، آن گاه خط ۱۰ مقدار d را چاپ می‌کند.

ممکن است این رویه اول مقداری عجیب به نظر برسد. با این حال، توجه کنید که POLLARD-RHO هیچ گاه یک جواب غلط چاپ نمی‌کند؛ هر عددی که این رویه چاپ می‌کند، یک مقسوم‌علیه نابدیهی n است. ولی ممکن است POLLARD-RHO کلا هیچ چیزی چاپ نکند؛ هیچ تضمینی وجود ندارد که این رویه نتیجه‌ای تولید کند. با این حال، خواهیم دید که دلیل خوبی وجود دارد که انتظار داشته باشیم POLLARD-RHO بعد از $\theta(\sqrt{p})$ تکرار حلقه‌ی while، یک فاکتور p از n را چاپ کند. بنابراین اگر n مرکب باشد، می‌توانیم انتظار داشته باشیم که این رویه بعد از $n^{1/4}$ به هنگام سازی، به تعداد کافی مقسوم‌علیه پیدا کرده باشد تا n را به طور کامل تجزیه کند، چرا که تمام فاکتورهای n، احتمالاً غیر از بزرگ‌ترین آن‌ها، کوچک‌تر از $\sqrt{n}$ هستند.



شکل ۳۱-۷ مکاشفه‌ی رو-ی پولارد. (الف) مقادیر تولید شده توسط دنباله‌ی $x_{i+1} = (x_i^2 - 1) \bmod 1387$، با شروع از $x_1 = 2$ تجزیه‌ی ۱۳۸۷ به اعداد اول به صورت $1387 = 19 \cdot 73$ است. فلش‌های پررنگ نشان‌دهنده‌ی مراحلی از تکرار هستند که قبل از یافتن فاکتور ۱۹انجام شده‌اند. فلش‌های کم‌رنگ به مقادیری اشاره می‌کنند که تکرار به آن‌ها نرسیده است، تا شکل «رو» را کامل کنند. مقادیر سایه‌دار، مقادیر y هستند که در POLLARD-RHO ذخیره شده‌اند. فاکتور ۱۹ با رسیدن به $x_7 = 177$ کشف شده است، که در آن $\gcd(63 - 177, 1387) = 19$ محاسبه می‌شود. اولین مقدار x که تکرار می‌شود ۱۱۸۶ است، ولی فاکتور ۱۹ قبل از تکرار این مقدار یافت می‌شود. (ب) مقادیر تولید شده توسط همان بازگشت به پیمانه‌ی ۱۹. تمام مقادیر x_i در بخش (الف) به پیمانه‌ی ۱۹معادل هستند با مقدار x'_i که در این جا نشان داده شده است. مثلاً هر دو مقدار $x_4 = 63$ و $x_7 = 177$ به پیمانه‌ی ۱۹ معادل هستند با ۶. (پ) مقادیر تولید شده توسط همان بازگشت به پیمانه‌ی ۷۳. تمام مقادیر x_i در بخش (الف) به پیمانه‌ی ۷۳ معادل هستند با مقدار x''_i که در این جا نشان داده شده است. طبق قضیه‌ی باقی‌مانده‌ی چینی، هر گره در بخش (الف) متناظر است با جفتی از گره‌ها، یکی از بخش (ب) و یکی از بخش (پ).

تحلیل خود از رفتار این رویه را با بررسی این مطلب آغاز می‌کنیم که چقدر طول می‌کشد که یک دنباله‌ی تصادفی به پیمانه‌ی n یک مقدار را تکرار کند. چون $\mathbb{Z}_n$ متناهی است، و چون هر مقدار در دنباله‌ی (۳۱-۴۴) فقط به مقدار قبلی وابسته است، دنباله‌ی (۳۱-۴۴) نهایتاً تکرار خواهد شد. وقتی به یک x_i رسیدیم به طوری که $x_i = x_j$ برای یک $j < i$، آن گاه در یک حلقه قرار داریم، چرا که $x_{i+1} = x_{j+1}$، $x_{i+2} = x_{j+2}$، و الی آخر. دلیل نام «مکاشفه‌ی رو» این است که، همان طور که شکل ۳۱-۷ نشان می‌دهد، دنباله‌ی $x_1, x_2, \ldots, x_{j-1}$ را می‌توان به صورت «دسته»ی حرف رو-ی یونانی

کشید، و دور $x_i, x_{i+1}, \ldots, x_j$ را به صورت «بدنه»ی حرف رو.

اجازه دهید بررسی کنیم که چقدر طول می‌کشد که دنباله‌ی x_i تکرار شود. البته این دقیقاً چیزی نیست که ما نیاز داریم، ولی بعداً خواهیم دید که چگونه می‌توانیم با یک اصلاح به چیزی که می‌خواهیم برسیم.

برای به دست آوردن این تخمین، اجازه دهید فرض کنیم تابع

$$f_n(x) = (x^2 - 1) \bmod n$$

به صورت یک تابع «تصادفی» رفتار می‌کند. مسلماً این تابع واقعاً تصادفی نیست، ولی این فرض به نتایجی منجر می‌شود که با رفتار مشاهده شده در POLLARD-RHO هم‌خوانی دارد. سپس می‌توانیم فرض کنیم که هر x_i مستقلا و با یک توزیع یکنواخت از $\mathbb{Z}_n$ انتخاب شده است. طبق تحلیل تناقض تولدها در بخش ۵-۴-۱، امیدریاضی تعداد مراحل طی شده قبل از این که دنباله در حلقه بیافتد، $\theta(\sqrt{n})$ است.

اکنون اصلاح مورد نیاز را انجام می‌دهیم. فرض کنید p یک فاکتور غیر بدیهی از n باشد به طوری که $\gcd(p, n/p) = 1$. مثلاً اگر تجزیه‌ی n به صورت $n = p_1^{e_1} p_2^{e_2} \ldots p_r^{e_r}$ باشد، آن گاه می‌توانیم فرض کنیم p همان $p_1^{e_1}$ است. (اگر $e_1 = 1$، آن گاه p کوچک‌ترین فاکتور اول n خواهد بود، یک مثال خوب برای به خاطر سپردن.)

دنباله‌ی $\langle x_i \rangle$ شامل یک دنباله‌ی مربوطه‌ی $\langle x_i' \rangle$ به پیمانه‌ی p است، که در آن

$$x_i' = x_i \bmod p$$

برای تمام i ها.

به علاوه، چون f_n فقط با استفاده از اعمال ریاضی (مربع‌گیری و تفریق) به پیمانه‌ی n تعریف شده است، خواهیم دید که می‌توان x_{i+1}' را از x_i' محاسبه کرد؛ دید «به پیمانه‌ی p » از دنباله، نسخه‌ی کوچک‌تری است از چیزی که به پیمانه‌ی n رخ می‌دهد:

$$\begin{aligned}
x_{i+1}' &= x_{i+1} \bmod p \\
&= f_n(x_i) \bmod p \\
&= ((x_i^2 - 1) \bmod n) \bmod p \\
&= (x_i^2 - 1) \bmod p \qquad \text{(طبق تمرین ۳۱-۱-۷)} \\
&= ((x_i \bmod p)^2 - 1) \bmod p \\
&= ((x_i')^2 - 1) \bmod p \\
&= f_p(x_i')
\end{aligned}$$

بنابراین، با این که صریحاً دنباله‌ی $\langle x_i' \rangle$ را محاسبه نمی‌کنیم، این دنباله خوش تعریف است و از همان بازگشتی پیروی می‌کند که $\langle x_i \rangle$ پیروی می‌کند.

با استدلال‌های مشابه قبل، می‌بینیم که امیدریاضی تعداد مراحل انجام شده قبل از این که دنباله‌ی

$\langle x'_i \rangle$ تکرار شود، $\theta(\sqrt{p})$ است. اگر p در مقایسه با n کوچک باشد، ممکن است $\langle x'_i \rangle$ بسیار سریع‌تر از دنباله‌ی $\langle x_i \rangle$ تکرار شود. در واقع دنباله‌ی $\langle x'_i \rangle$ صرفا زمانی تکرار می‌شود که دو عنصر از دنباله‌ی $\langle x_i \rangle$ به پیمانه‌ی p با هم برابر شوند، و نه به پیمانه‌ی n. شکل ۳۱-۷، بخش (ب) و (پ) را ببینید.

فرض کنید t نشان‌دهنده‌ی اندیس اولین مقدار تکرار شده در دنباله‌ی $\langle x'_i \rangle$ باشد، و $u > 0$ نشان‌دهنده‌ی طول چرخه‌ای که درست شده است. یعنی، t و $u > 0$ کوچک‌ترین مقادیری هستند به طوری که $x'_{t+i} = x'_{t+u+i}$ برای هر $i \geq 0$. طبق بحث بالا، امیدریاضی مقادیر t و u هر دو $\theta(\sqrt{p})$ است. توجه کنید که اگر $x'_{t+i} = x'_{t+u+i}$، آن گاه $p \mid (x_{t+u+i} - x_{t+i})$. بنابراین $\gcd(x_{t+u+i} - x_{t+i}, n) > 1$.

پس وقتی POLLARD-RHO یک مقدار x_k را در y ذخیره کرد، به طوری که $k \geq t$، آن گاه $y \bmod p$ همیشه در چرخه‌ی به پیمانه‌ی p وجود دارد. (اگر یک مقدار جدید به عنوان y ذخیره شود، آن مقدار هم در چرخه‌ی به پیمانه‌ی p وجود دارد.) در نهایت، k مقداری به خود می‌گیرد که بزرگ‌تر از u است، و سپس رویه یک دور کامل بر روی چرخه‌ی به پیمانه‌ی p می‌زند، بدون این که مقدار y را تغییر دهد. سپس وقتی که x_i به مقدار قبلی ذخیره شده در y «بر می‌خورد»، یعنی وقتی $x_i \equiv y \pmod{p}$، یک فاکتور از n پیدا می‌شود.

احتمالاً فاکتور پیدا شده p است، هر چند ممکن است مضربی از p پیدا شود. چون امیدریاضی هر دوی t و u، $\theta(\sqrt{p})$ است، امیدریاضی تعداد مراحل انجام شده برای ساختن فاکتور p، $\theta(\sqrt{p})$ است. دو دلیل وجود دارد که باعث می‌شود این رویه آن طوری که انتظار می‌رود عمل نکند. اول، تحلیل مکاشفه‌ای زمان اجرا دقیق نیست، و ممکن است که چرخه‌ی مقادیر به پیمانه‌ی p، بسیار بزرگ‌تر از $\sqrt{p}$ باشد. در این حالت الگوریتم به درستی کار می‌کند، ولی بسیار کندتر از میزان مورد انتظار. در عمل به نظر می‌آید که این حالت فقط در حد بحث باشد. دوم مقسوم‌علیه‌های n که توسط این رویه تولید شده‌اند ممکن است همیشه یکی از فاکتورهای بدیهی ۱ یا n باشند. مثلاً فرض کنید که $n = pq$، که در آن p و q اول هستند. ممکن است این اتفاق بیفتد که مقادیر t و u برای p همان مقادیر t و u برای q باشند، و همیشه فاکتور p با همان اعمال ب.م.م-ی کشف شوند که فاکتور q کشف می‌شود. از آن جایی که هر دو فاکتور در یک زمان کشف می‌شوند، فاکتوری که کشف می‌کنیم، فاکتور بدیهی $n = pq$ است، که فایده‌ای ندارد. باز هم این مسئله به نظر می‌رسد که در عمل قابل چشم‌پوشی باشد. در صورت لزوم، می‌توان مکاشفه را دوباره با بازگشتی دیگر به شکل $x_{i+1} = (x_i^2 - c) \bmod n$ آغاز کرد. (به دلایلی که در این جا از آن‌ها صحبت نمی‌کنیم، از مقادیر $c = 0$ و $c = 2$ باید صرف نظر کرد، ولی بقیه‌ی مقادیر مشکلی ندارند.)

مسلماً این تحلیل مکاشفه‌ای است نه دقیق، چرا که بازگشت واقعاً «تصادفی» نیست. با این همه این رویه در عمل به خوبی جواب می‌دهد، و به نظر می‌رسد کارایی آن همان قدری باشد که این تحلیل نشان می‌دهد. برای یافتن فاکتورهای اول کوچک اعداد بزرگ، این متد انتخاب خوبی است.

برای تجزیه‌ی کامل یک عدد مرکب β بیتی n، فقط نیاز داریم تمام فاکتورهای اول کوچک‌تر از $\lfloor n^{1/2} \rfloor$ را بیابیم، و انتظار داریم که POLLARD-RHO حداکثر به $n^{1/4} = 2^{\beta/4}$ عملیات ریاضی و $n^{1/4}\beta = 2^{\beta/4}\beta^2$ عملیات بیتی نیاز داشته باشد. توانایی POLLARD-RHO در یافتن فاکتورهای p کوچک از یک عدد n، با امیدریاضی $\theta(\sqrt{p})$ عملیات ریاضی معمولاً رضایت‌بخش‌ترین مزیت آن است.

تمرین‌ها

۳۱-۹-۱ با مراجعه به دنباله‌ی اجرا، نشان داده شده در شکل ۳۱-۷ (الف)، رویه‌ی POLLARD-RHO چه زمانی فاکتور ۷۳ را برای عدد ۱۳۸۷ چاپ می‌کند؟

۳۱-۹-۲ فرض کنید که تابع $f : \mathbb{Z}_n \to \mathbb{Z}_n$ و یک مقدار اولیه‌ی $x_0 \in \mathbb{Z}_n$ به ما داده شده است. تعریف می‌کنیم $x_i = f(x_{i-1})$ برای $i = 1,2,\ldots$. فرض کنید t و $u > 0$ کوچک‌ترین مقادیری باشند به طوری که $x_{t+i} = x_{t+u+i}$ برای $i = 0,1,\ldots$. به زبان الگوریتم رو-ی پولارد، t طول دسته و u طول چرخه‌ی رو است. یک الگوریتم کارآمد برای تعیین دقیق t و u بدهید و زمان اجرای آن را تحلیل کنید.

۳۱-۹-۳ برای یافتن یک فاکتور به شکل p^e، که در آن p اول است و $e > 1$، انتظار دارید POLLARD-RHO چند مرحله طی کند؟

۳۱-۹-۴★ همان طور که دیدید، یکی از معایب POLLARD-RHO این است که برای هر مرحله در بازگشت نیاز دارد که یک ب.م.م محاسبه کند. یک پیشنهاد این است که ممکن است بتوانیم اعمال ب.م.م را به صورت دسته جمعی انجام دهیم، بدین صورت که ضرب چندین x_i پشت سر هم را محاسبه کنیم و سپس از این حاصل ضرب به جای x_i در محاسبات ب.م.م استفاده کنیم. به دقت توضیح دهید که چگونه این ایده را پیاده‌سازی می‌کنید، چرا این ایده کار می‌کند، و چه اندازه‌ای برای گروه‌های خود انتخاب می‌کنید تا هنگام کار با عدد β بیتی n بهترین کارایی را داشته باشید.

مسائل

۳۱-۱ الگوریتم ب.م.م دودویی

در اکثر کامپیوترها، اعمال تفریق، تست زوج یا فرد بودن یک عدد دودویی، و نصف کردن را می‌توان سریع‌تر از محاسبه‌ی باقی‌مانده‌ی تقسیم انجام داد. این مسئله، *الگوریتم ب.م.م دودویی* را تشریح می‌کند، که در آن از محاسبات باقی‌مانده در الگوریتم اقلیدس دوری شده است.

I. اثبات کنید که اگر a و b هر دو زوج باشند، آن گاه $\gcd(a,b) = 2\gcd(a/2,b/2)$.

II. اثبات کنید که اگر a فرد و b زوج باشد، آن گاه $\gcd(a,b) = \gcd(a,b/2)$.

III. اثبات کنید که اگر a و b هر دو فرد باشند، آن گاه $\gcd(a,b) = \gcd((a-b)/2,b)$.

مسائل

IV. یک الگوریتم ب.م.م دودویی کارآمد برای ورودی‌های a و b، که در آن $a \geq b$، طراحی کنید که در زمان $O(\lg a)$ اجرا می‌شود. فرض کنید که هر تفریق، تست زوج و فرد بودن، و نصف کردن می‌تواند در یک واحد زمان انجام شود.

۳۱-۲ تحلیل اعمال بیتی در الگوریتم اقلیدس

I. الگوریتم «کاغذ و قلم» معمولی را برای تقسیم‌های طولانی در نظر بگیرید: تقسیم a بر b، که به یک خارج‌قسمت q و یک باقی‌مانده‌ی r ختم می‌شود. نشان دهید که این متد به $O((1+\lg q)\lg b)$ عملیات بیتی نیاز دارد.

II. تعریف می‌کنیم $\mu(a,b) = (1+\lg a)(1+\lg b)$. نشان دهید که تعداد اعمال بیتی انجام شده توسط EUCLID در کاهش مسئله‌ی محاسبه‌ی $\gcd(a,b)$ به مسئله‌ی محاسبه‌ی $\gcd(b, a \bmod b)$ حداکثر $c(\mu(a,b) - \mu(b, a \bmod b))$ است، برای یک ثابت به اندازه‌ی کافی بزرگ $c > 0$.

III. نشان دهید که EUCLID (a,b) به طور کلی به $O(\mu(a,b))$ عملیات بیتی و برای دو ورودی β بیتی به $O(\beta^3)$ عملیات بیتی نیاز دارد.

۳۱-۳ سه الگوریتم برای اعداد فیبوناچی

این مسئله کارایی سه متد را برای محاسبه‌ی nامین عدد فیبوناچی با داشتن n، یعنی F_n، بررسی می‌کند. فرض کنید که هزینه‌ی جمع، تفریق، یا ضرب دو عدد $O(1)$ است، مستقل از اندازه‌ی اعداد.

I. نشان دهید که زمان اجرای متد بازگشتی سرراست برای محاسبه‌ی F_n بر مبنای رابطه‌ی بازگشتی (۳-۲۱) نسبت به n نمایی است. (برای مثال، رویه‌ی FIB در بخش ۲۷-۱ را ببینید.)

II. نشان دهید که با استفاده از یادداشت برداری، چگونه می‌توان F_n را در زمان $O(n)$ محاسبه کرد.

III. نشان دهید که چگونه می‌توان فقط با استفاده از جمع و ضرب اعداد صحیح، F_n را در زمان $O(\lg n)$ محاسبه کرد. (راهنمایی: ماتریس

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

و توان‌های آن را در نظر بگیرید.)

IV. اکنون فرض کنید که جمع دو عدد β بیتی به زمان $\theta(\beta)$، و ضرب دو عدد β بیتی به $\theta(\beta^2)$ زمان نیاز دارد. تحت این مقیاس معقول‌تر برای اعمال اولیه‌ی ریاضی، زمان اجرای این سه متد چقدر است؟

۴-۳۱ *باقی‌مانده‌های درجه‌ی دو*

فرض کنید p یک عدد اول باشد. یک عدد $a \in \mathbb{Z}_p^*$ یک *باقی‌مانده‌ی دوجه‌ی دو* است اگر تساوی $x^2 = a \pmod p$ برای مجهول x جواب داشته باشد.

I. نشان دهید که دقیقاً $(p-1)/2$ باقی‌مانده‌ی درجه‌ی دو به پیمانه‌ی p وجود دارد.

II. اگر p اول باشد، تعریف می‌کنیم سمبل لاگرانژ $\left(\frac{a}{p}\right)$ برای $a \in \mathbb{Z}_p^*$ برابر است با ۱ اگر a یک باقی‌مانده‌ی درجه‌ی دو به پیمانه‌ی p باشد، و ۱- در غیر این صورت. اثبات کنید که اگر $a \in \mathbb{Z}_p^*$، آن گاه

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod p$$

یک الگوریتم بهینه برای تعیین این که یک عدد a یک باقی‌مانده‌ی درجه‌ی دو به پیمانه‌ی p است یا خیر، ارائه کنید. کارایی الگوریتم خود را تحلیل کنید.

III. اثبات کنید که اگر p یک عدد اول به شکل $4k+3$ باشد و a یک باقی‌مانده‌ی درجه‌ی دو در $\mathbb{Z}_p^*$، آن گاه $a^{k+1} \bmod p$ یک ریشه‌ی دوم a به پیمانه‌ی p است. برای یافتن یک ریشه‌ی دوم یک باقی‌مانده‌ی درجه‌ی دو به پیمانه‌ی p به چه مقدار زمان نیاز است؟

IV. یک الگوریتم تصادفی کارآمد برای یافتن یک نا-باقی‌مانده‌ی درجه‌ی دو به پیمانه‌ی یک عدد اول دلخواه p ارائه کنید، یعنی عضوی از $\mathbb{Z}_p^*$ که یک باقی‌مانده‌ی درجه‌ی دو نیست. در حالت متوسط الگوریتم شما به چند عملیات ریاضی نیاز دارد؟

فصل

۳۲

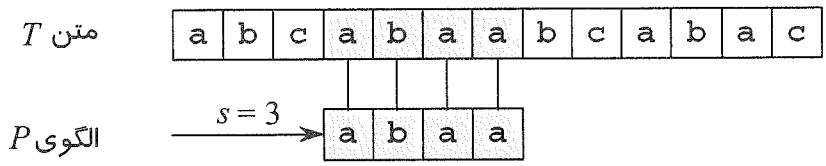
تطابق رشته‌ها

۳۲-۰ مقدمه

یافتن تمام رخدادهای یک الگو در یک متن، مسئله‌ای است که به کرات در برنامه‌های ویرایش متن پیش می‌آید. معمولاً، متن همان سندی است که ویرایش می‌شود، و الگوی مورد جستجو، یک کلمه که از کاربر دریافت می‌شود. الگوریتم‌های کارآمد برای این مسئله – که تطابق رشته نام دارد – می‌توانند به مقدار زیادی سرعت پاسخگویی برنامه‌ی ویرایش متن را افزایش دهند. از الگوریتم‌های تطابق رشته در کاربردهای دیگر، مثلاً جستجوی الگوهای خاص در دنباله‌های DNA هم استفاده می‌شود. همچنین موتورهای جستجوی اینترنتی از آن‌ها برای یافتن صفحه‌های وب مربوط به یک جستجو استفاده می‌کنند.

به طور رسمی *مسئله‌ی تطابق رشته* را به صورت زیر تعریف می‌کنیم. فرض می‌کنیم که متن، یک آرایه‌ی $T[1..n]$ با طول n است، و الگوی مورد جستجو، یک آرایه‌ی $P[1..m]$ با طول $m \le n$. فرض می‌کنیم که عناصر P و T کاراکترهایی هستند از یک الفبای متناهی Σ. مثلاً ممکن است داشته باشیم $\Sigma = \{0, 1\}$ یا $\Sigma = \{a, b, ..., z\}$. معمولاً به آرایه‌های کاراکتری P و T، *رشته* (string)های کاراکتری گفته می‌شود.

می‌گوییم الگوی P *با جابه‌جایی* s در متن T رخ داده است (و یا الگوی P *با شروع از مکان* $s+1$ در متن T رخ داده است) اگر $0 \le s \le n-m$ و $T[s+1..s+m] = P[1..m]$ (یعنی، اگر $T[s+j] = P[j]$ برای $1 \le j \le m$). اگر P با جابه‌جای s در T رخ دهد، آن گاه به s یک *جابه‌جایی معتبر* (valid shift) می‌گوییم؛ در غیر این صورت s یک *جابه‌جایی نامعتبر* (invalid shift) است. مسئله‌ی تطابق رشته، مسئله‌ی یافتن جابه‌جایی‌های متعبری است که برای آن‌ها یک الگوی P در یک متن T رخ می‌دهد. شکل ۳۲-۱ این تعاریف را نمایش می‌دهد.



شکل ۳۲-۱ مسئله‌ی تطابق رشته. هدف یافتن تمام رخ‌دادهای الگوی $P = abaa$ در متن $T = abcabaabcabac$ است. الگو فقط یک بار در متن رخ داده است، در جابه‌جایی $s = 3$. به این جابه‌جایی، یک جابه‌جای معتبر گفته می‌شود. هر کاراکتر در الگو با یک خط عمودی به کاراکتر تطابق یافته در متن متصل شده است، و تمام کاراکترهای تطابق یافته در متن با سایه مشخص شده‌اند.

به جز الگوریتم ساده‌لوحانه، که آن را در بخش ۳۲-۱ مرور می‌کنیم، تمام الگوریتم‌های تطابق رشته در این فصل ابتدا یک پیش‌پردازش بر طبق الگو انجام می‌دهند، و سپس تمام جابه‌جایی‌های معتبر را پیدا می‌کنند؛ به عبارت دوم «تطابق دادن» می‌گوییم. شکل ۳۲-۲ زمان پیش‌پردازش و تطابق را در هر یک از الگوریتم‌های این فصل نشان می‌دهد. زمان اجرای هر الگوریتم برابر است با مجموع زمان پیش‌پردازش و زمان تطابق. بخش ۳۲-۲ یک الگوریتم تطابق رشته‌ی جذاب را که منسوب به رابین (Rabin) و کارپ (Krap) است، معرفی می‌کند. با این که بدترین حالت زمان اجرای $\theta((n-m+1)m)$ این الگوریتم بهتر از متد ساده‌لوحانه نیست، این الگوریتم در حالت متوسط و در عمل بسیار بهتر عمل می‌کند. همچنین این مسئله را می‌توان به خوبی برای مسئله‌های تطابق الگوی دیگر هم گسترش داد. سپس بخش ۳۲-۳ یک الگوریتم تطابق رشته را توضیح می‌دهد که با ساختن یک اتوماتای متناهی آغاز می‌شود که مخصوصاً برای جستجوی رخ‌دادهای یک الگوی داده شده‌ی P در یک متن طراحی شده است. این الگوریتم به $O(m|\Sigma|)$ زمان پیش‌پردازش نیاز دارد، ولی در عوض زمان تطابق آن فقط $\theta(n)$ است. الگوریتم مشابه، ولی بسیار هوشمندانه‌تر Knuth-Morris-Pratt (یا KMP) در بخش ۳۲-۴ معرفی می‌شود؛ زمان تطابق الگوریتم KMP همان $\theta(n)$ است، ولی زمان پیش‌پردازش آن به $\theta(m)$ کاهش یافته است.

نمادها و اصطلاحات فنی

فرض می‌کنیم Σ^* (بخوانید سیگما-ستاره) نشان‌دهنده‌ی مجموعه‌ی تمام رشته‌های با طول متناهی ساخته شده با استفاده از کاراکترهای الفبای Σ باشد. در این فصل فقط رشته‌های با طول متناهی را در نظر می‌گیریم. *رشته‌ی تهی* با طول صفر، که آن را با ε نشان می‌دهیم، هم به Σ^* تعلق دارد. طول یک

الگوریتم	زمان پیش پردازش	زمان تطبیق
ساده‌لوحانه	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
اتوماتای متناهی	$O(m\|\Sigma\|)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

شکل ۳۲-۲ الگوریتم‌های تطابق رشته‌ی ارائه شده در این فصل و زمان پیش‌پردازش و تطابق آن‌ها.

رشته‌ی x با $|x|$ نشان داده می‌شود. *اتصال* (concatenation) دو رشته‌ی x و y، که آن را با xy نشان می‌دهیم، دارای طول $|x|+|y|$ است، و عبارت است از کاراکترهای رشته‌ی x و به دنبال آن کاراکترهای رشته‌ی y.

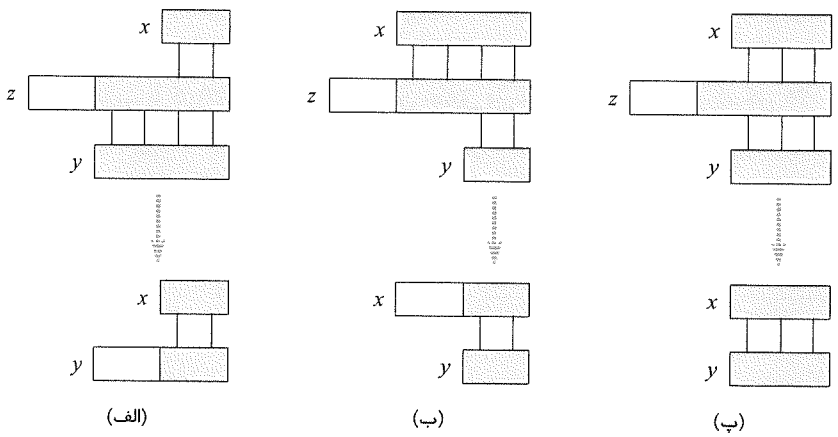
می‌گوییم یک رشته‌ی w *پیشوند* (prefix) یک رشته‌ی x است، و آن را با $w \sqsubset x$ نشان می‌دهیم، اگر $x = wy$ برای یک رشته‌ی $y \in \Sigma^*$. توجه کنید که اگر $w \sqsubset x$، آن گاه $|w| \le |x|$. به طور مشابه می‌گوییم یک رشته‌ی w یک *پسوند* (suffix) یک رشته‌ی x است، و می‌نویسیم $w \sqsupset x$، اگر $x = yw$. از $w \sqsubset x$ نتیجه می‌شود که $|w| \le |x|$. رشته‌ی تهی ε هم پیشوند تمام رشته‌ها است و هم پسوند تمام رشته‌ها. مثلاً داریم $ab \sqsubset abcca$ و $cca \sqsubset abcca$. مفید است توجه کنیم که برای هر رشته‌ی x و y و هر کاراکتر a، داریم $ya \sqsupset x$ اگر و فقط اگر $y \sqsupset x$. همچنین توجه کنید که $\sqsubset$ و $\sqsupset$ رابطه‌های تراگذار (متعدی) هستند. لم زیر بعداً مورد استفاده قرار خواهد گرفت.

لم ۳۲-۱ (لم پیشوندهای همپوشان)

فرض کنید x، y، و z رشته‌هایی هستند به طوری که $z \sqsubset x$ و $z \sqsubset y$. اگر $|y| \le |x|$ آن گاه $y \sqsubset x$. اگر $|x| \ge |y|$ آن گاه $x \sqsubset y$. اگر $|x| = |y|$ آن گاه $x = y$.

اثبات شکل ۳۲-۳ را برای یک اثبات تصویری ببینید. ▩

برای مختصر شدن نمادها، پیشوند k کاراکتری $P[1 .. k]$ از $P[1 .. m]$ را P_k نشان می‌دهیم. بنابراین $P_0 = \varepsilon$ و $P_m = P = P[1 .. m]$. به طور مشابه پیشوند k کاراکتری متن T را با T_k نشان



شکل ۳۲-۳ یک اثبات تصویری برای لم ۳۲-۱. فرض می‌کنیم $z \sqsubset x$ و $z \sqsubset y$. سه بخش شکل نشان‌دهنده‌ی سه حالت لم هستند. خطوط عمودی مناطق دارای تطابق را (که با سایه مشخص شده‌اند) به هم متصل می‌کنند. (الف) اگر $|y| \le |x|$ آن گاه $y \sqsubset x$. (ب) اگر $|x| \ge |y|$ آن گاه $x \sqsubset y$. (پ) اگر $|x| = |y|$ آن گاه $x = y$.

می‌دهیم. با استفاده از این نمادگذاری، می‌توانیم مسئله‌ی تطابق رشته را به صورت یافتن تمام جابه‌جایی‌های s در دامنه‌ی $0 \le s \le n-m$ تعریف کنیم به طوری که $P \sqsupset T_{s+m}$.

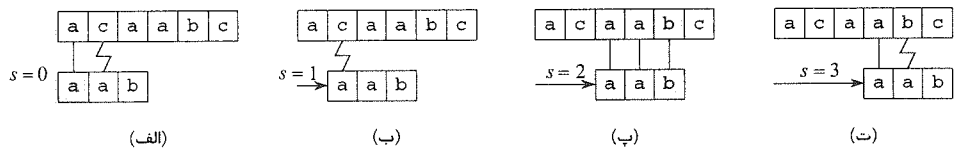
در شبه‌کدهای این فصل، اجازه می‌دهیم که تساوی دو رشته با طول برابر به عنوان یک عملیات اصلی بررسی شود. اگر رشته‌ها از چپ به راست مقایسه شوند، و مقایسه با یافتن یک عدم تطابق متوقف شود، فرض می‌کنیم که زمان صرف شده برای این تست، تابعی خطی از تعداد کاراکترهای تطابق یافته در تست است. به طور دقیق‌تر، فرض می‌شود تست "$x = y$"، $\theta(t+1)$ زمان بگیرد، که در آن t طول بلندترین رشته‌ی z است به طوری که $z \sqsubset x$ و $z \sqsubset y$. (می‌نویسیم $\theta(t+1)$ به جای $\theta(t)$ که حالت $t = 0$ را هم در نظر گرفته باشیم؛ اولین کاراکترهای مقایسه شده با یکدیگر برابر نیستند، ولی برای انجام این مقایسه به زمانی مثبت نیاز داریم.)

۳۲-۱ الگوریتم تطابق رشته‌ی ساده‌لوحانه

الگوریتم ساده‌لوحانه با استفاده از یک حلقه که شرط $P[1..m] = T[s+1..s+m]$ را برای تمام $n-m+1$ مقدار ممکن s چک می‌کند، تمام جابه‌جایی‌های معتبر را می‌یابد.

```
NAIVE-STRING-MATCHER(T, P)
1  n = length[T]
2  m = length[P]
3  for s = 0 to n − m
4      if P[1 .. m] == T[s + 1 .. s + m]
5          print "Pattern occurs with shift" s
```

الگوریتم ساده‌لوحانه را می‌توان به صورت تصویری به شکل یک «قالب» متحرک فرض کرد که حاوی الگوی مورد نظر است و بر روی متن حرکت می‌کند، و دقت می‌کند که برای کدام جابه‌جایی‌ها تمام کاراکترهای قالب با کاراکترهای متناظر در متن تطابق دارند، همان طور که در شکل ۳۲-۴ نشان داده شده است. حلقه‌ی for که در خط ۳ آغاز می‌شود هر جابه‌جایی ممکن را به صورت جداگانه در



شکل ۳۲-۴ عملیات الگوریتم تطابق رشته‌ی ساده‌لوحانه برای الگوی $P = aab$ و متن $T = acaabc$. می‌توانیم الگوی P را به صورت یک «قالب» در نظر بگیریم که بر روی متن حرکت می‌کند. (الف)-(ت) چهار انطباق انجام شده توسط الگوریتم تطابق رشته. در هر بخش، خطوط عمودی کاراکترهای تطابق یافته (سایه‌دار) را به هم متصل می‌کنند، و یک خط موج‌دار اولین کاراکتری را که تطابق ندارد را به کاراکتر متناظر متصل می‌کند، در صورت وجود. یک تطابق در الگو یافت شده است، در جابه‌جایی $s = 2$ که در بخش (پ) نشان داده شده است.

نظر می‌گیرد. تست خط ۴ تعیین می‌کند که آیا جابه‌جایی فعلی معتبر است یا خیر؛ این تست شامل یک حلقه‌ی ضمنی است که کاراکترهای مکان‌های متناظر را چک می‌کند تا زمانی که تمام کاراکترها با هم تطابق داشته باشند، و یا یک عدم تطابق یافت شود. خط ۵ تمام جابه‌جایی‌های معتبر s را چاپ می‌کند.

شبه‌کد NAIVE-STRING-MATCHER به $O((n-m+1)m)$ زمان نیاز دارد، و این کران برای بدترین حالت، نزدیک است. مثلاً متن a^n (رشته‌ای از n کاراکتر a) را در نظر بگیرید، با الگوی a^m. برای هر یک از $n-m+1$ مقدار ممکن برای جابه‌جایی‌های s، حلقه‌ی ضمنی در خط ۴ برای مقایسه‌ی کاراکترهای متناظر باید m بار اجرا شود تا اعتبار جابه‌جایی را تأیید کند. بنابراین بدترین حالت زمان اجرا $\theta((n-m+1)m)$ است، که اگر $m=\lfloor n/2 \rfloor$، برابر است با $\theta(n^2)$. زمان اجرای NAIVE-STRING-MATCHER برابر است با زمان تطابق آن، چرا که در آن هیچ پیش‌پردازشی انجام نمی‌شود.

همان طور که خواهیم دید، NAIVE-STRING-MATCHER رویه‌ای بهینه برای این مسئله نیست. در واقع در این فصل خواهیم دید که الگوریتم Knuth—Morris-Pratt در بدترین حالت بسیار بهتر عمل می‌کند. الگوریتم تطابق رشته‌ی ساده‌لوحانه از این رو غیر بهینه است که از اطلاعات به دست آمده برای یک مقدار s در پیدا کردن مقادیر دیگر s هیچ استفاده‌ای نمی‌کند، با این که چنین اطلاعاتی می‌تواند بسیار مفید باشد. مثلاً اگر $P=aaab$ و دریابیم که $s=0$ معتبر است، در این صورت هیچ یک از جابه‌جایی‌های ۱، ۲، و یا ۳ نمی‌توانند معتبر باشند، چرا که $T[4]=b$. در بخش‌های بعد، راه‌های مختلفی را برای استفاده‌ی مفید از چنین اطلاعاتی بررسی می‌کنیم.

تمرین‌ها

۳۲-۱-۱ مقایسه‌هایی را که الگوریتم تطابق رشته‌ی ساده‌لوحانه برای الگوی $P=0001$ و متن $T=000010001010001$ انجام می‌دهد، نشان دهید.

۳۲-۱-۲ فرض کنید که تمام کاراکترها در یک الگوی P متفاوت هستند. نشان دهید که چگونه می‌توان NAIVE-STRING-MATCHER را طوری اصلاح کرد که بر روی یک متن T با n کاراکتر در زمان $O(n)$ اجرا شود.

۳۲-۱-۳ فرض کنید که الگوی P و متن T به صورت *تصادفی* و به ترتیب با طول‌های m و n، و از الفبای dتایی $\Sigma_d=\{0,1,\ldots,d-1\}$ انتخاب شده‌اند، که در آن $d\geq 2$. نشان دهید که امیدریاضی تعداد مقایسه‌های کاراکتر به کاراکتر انجام شده توسط حلقه‌ی ضمنی خط ۴ در الگوریتم ساده‌لوحانه عبارت است از

$$(n-m+1)\frac{1-d^{-m}}{1-d^{-1}} \leq 2(n-m+1)$$

در تمام تکرارهای انجام شده‌ی حلقه. (فرض کنید که الگوریتم ساده‌لوحانه با یافتن یک عدم تطابق در کاراکترها و یا تطابق یافتن تمام کاراکترهای الگو برای یک جابه‌جایی خاص، متوقف می‌شود.) بنابراین برای رشته‌های به صورت تصادفی انتخاب شده، الگوریتم ساده‌لوحانه کاملاً کارآمد است.

۳۲-۱-۴ فرض کنید که به الگوی P اجازه می‌دهیم حاوی *کاراکتر شکاف* ◊ باشد که می‌تواند با رشته‌ای *دلخواه* از کاراکترها تطابق کند (حتی رشته‌ای با طول صفر). مثلاً، الگوی $ab\lozenge ba\lozenge c$ در متن $cabccbacbacab$ رخ می‌دهد، بدین صورت:

$$\underbrace{c}_{}\ \underbrace{ab}_{ab}\ \underbrace{cc}_{\lozenge}\ \underbrace{ba}_{ba}\ \underbrace{cba}_{\lozenge}\ \underbrace{c}_{c}\ ab$$

و بدین صورت

$$\underbrace{c}_{}\ \underbrace{ab}_{ab}\ \underbrace{ccbac}_{\lozenge}\ \underbrace{ba}_{ba}\ \underbrace{}_{\lozenge}\ \underbrace{c}_{c}\ ab$$

توجه کنید که کاراکتر شکاف می‌تواند به تعدادی دلخواه در الگو رخ دهد، ولی فرض بر این است که در متن رخ نمی‌دهد. یک الگوریتم با زمان چندجمله‌ای بدهید که تعیین می‌کند که آیا الگویی مانند P در یک متن داده شده‌ی T رخ می‌دهد یا خیر، و سپس زمان اجرای الگوریتم خود را تحلیل کنید.

۳۲-۲ الگوریتم رابین-کارپ

رابین و کارپ یک الگوریتم تطابق رشته ارائه کرده‌اند که در عمل به خوبی کار می‌کند، و همچنین می‌توان آن را برای مسئله‌های مشابه هم گسترش داد، مسئله‌هایی مانند تطابق الگوی دو بعدی. الگوریتم رابین-کارپ به $\theta(m)$ زمان پیش‌پردازش نیاز دارد، و بدترین حالت زمان اجرای آن $\theta((n-m+1)m)$ است. با این حال، با فرض‌های خاص زمان اجرای متوسط آن بهتر است.

این الگوریتم از اعمال اولیه‌ی نظریه‌ی اعداد مانند هم‌ارزی دو عدد به پیمانه‌ی یک عدد سوم استفاده می‌کند. ممکن است بخواهید برای تعاریف مربوطه به بخش ۳۱-۱ مراجعه کنید.

برای اهداف توضیحی، اجازه دهید فرض کنیم $\Sigma = \{0, 1, 2, \ldots, 9\}$، به طوری که هر کاراکتر یک رقم است. (در حالت کلی، می‌توانیم فرض کنیم هر کاراکتر یک رقم در مبنای d است، که در آن $d = |\Sigma|$.) سپس می‌توانیم رشته‌ای از k کاراکتر پشت سر هم را به صورت نمایش دهدهی یک عدد با طول k در نظر بگیریم. بنابراین رشته‌ی کاراکتری ۳۱۴۱۵ متناظر است با عدد دهدهی ۳۱,۴۱۵. با ترجمه‌ی دوگانه‌ی کاراکترهای ورودی به هر دو صورت نمادهای تصویری و ارقام، ساده خواهد بود که در این بخش آن‌ها را به همان صورتی معنی کنیم که ارقام را در متن‌های معمولی خود معنی می‌کنیم.

با داشتن یک الگوی $P[1\,..\,m]$، فرض می‌کنیم p نشان‌دهنده‌ی مقدار دهدهی متناظر باشد. به

روشی مشابه، با داشتن یک متن $T[1..n]$، فرض می‌کنیم t_s نشان دهنده‌ی مقدار دهدهی با طول m مربوط به زیررشته‌ی $T[s+1..s+m]$ باشد، برای $s = 0, 1, ..., n-m$. به طور خاص، $t_s = p$ اگر و فقط اگر $T[s+1..s+m] = P[1..m]$؛ بنابراین s یک جابه‌جایی معتبر است اگر و فقط اگر $t_s = p$. اگر بتوانیم p را در زمان $\theta(m)$ و تمام مقادیر t_s را در زمان $\theta(n)$ محاسبه کنیم، آن گاه می‌توانیم با مقایسه‌ی p با هر یک از t_sها، تمام جابه‌جایی‌های معتبر s را در زمان $\theta(n-m+1)$[1] $\theta(n)$ تعیین کنیم. (اجازه دهید در این جا نگران این نباشیم که p و t_sها ممکن است اعدادی بسیار بزرگ باشند.)

می‌توانیم با استفاده از قانون هورنر (بخش ۳۰-۱ را ببینید) p را در زمان $\theta(m)$ محاسبه کنیم:

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + ... + 10(P[2] + 10P[1])...))$$

به طور مشابه می‌توان مقدار t_0 را از $T[1..m]$ در زمان $\theta(m)$ محاسبه کرد.

برای محاسبه‌ی بقیه‌ی مقادیر $t_1, t_2, ..., t_{n-m}$ در زمان $\theta(n-m)$، کافی است مشاهده کنیم که t_{s+1} را می‌توان در زمان ثابت از t_s محاسبه کرد، چرا که

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1] \qquad (۳۲-۱)$$

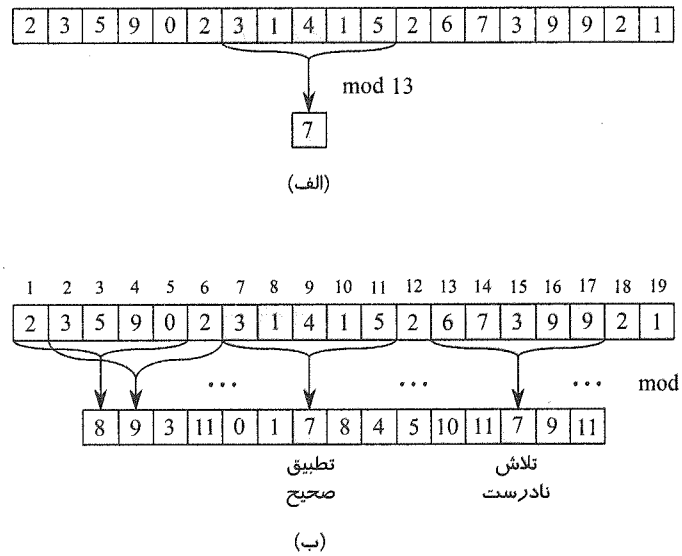
تفریق $10^{m-1}T[s+1]$ رقم پر ارزش را از t_s حذف می‌کند، ضرب در ۱۰ عدد را یک رقم به سمت چپ جابه‌جا می‌کند، و اضافه کردن $T[s+m+1]$ رقم مناسب را در مکان کم ارزش قرار می‌دهد. به عنوان مثال اگر $m = 5$ و $t_s = 31415$، آن گاه می‌خواهیم رقم با ارزش $T[s+1] = 3$ را حذف کرده و یک رقم کم‌ارزش دیگر (فرض کنید این رقم $T[s+5+1] = 2$ است) را به آن اضافه کنیم، و به دست آوریم

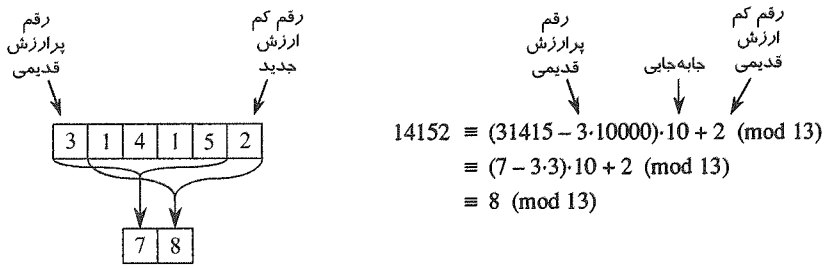
$$t_{s+1} = 10(31415 - 10000 \cdot 3) + 2$$
$$= 14152$$

اگر ثابت 10^{m-1} قبلاً محاسبه شده باشد (که می‌توان این کار را با استفاده از تکنیک‌های بخش ۳۱-۶ در زمان $O(\lg m)$ انجام داد، هر چند برای این کاربرد یک متد سرراست با زمان $O(m)$ هم کافی است)، آن گاه هر اجرای تساوی (۳۲-۱) به تعداد ثابتی عملیات ریاضی نیاز دارد. بنابراین می‌توانیم p را در زمان $O(m)$ و $t_0, t_1, ..., t_{n-m}$ را در زمان $\theta(n-m+1)$ محاسبه کنیم، و همچنین می‌توانیم تمام رخ‌دادهای الگوی $P[1..m]$ را در متن $T[1..n]$ با $\theta(m)$ زمان پیش‌پردازش و $\theta(n-m+1)$ زمان تطابق پیدا کرد.

تنها مشکل این رویه این است که p و t_s ممکن است بسیار بزرگ باشند و نتوان به سادگی با آن‌ها کار کرد. اگر P حاوی m کاراکتر باشد، آن گاه با فرض این که هر عملیات ریاضی بر روی p (که m رقم دارد) در «زمان ثابت» انجام می‌شود، نامعقول است. خوشبختانه یک راه حل ساده برای

[1] می‌نویسیم $\theta(n-m+1)$ به جای $\theta(n-m)$ چرا که $n-m+1$ مقدار متفاوت برای s وجود دارد. مقدار "+۱" از نظر حدی قابل توجه است، چرا که وقتی $m = n$، محاسبه‌ی تنها t_s به $\theta(1)$ زمان نیاز دارد، و نه $\theta(0)$ زمان.





شکل ۳۲-۵ الگوریتم رابین-کارپ. هر کاراکتر یک رقم دهدهی است، و مقادیر را بـه پیمانـه‌ی ۱۳ محاسبه می‌کنیم. (الف) یک متن رشته‌ای. یک پنجره با طول ۵ با سایه مشخص شده است. مقدار عـددی پنجره‌ی سایه‌دار به پیمانه‌ی ۱۳ محاسبه شده است، که به مقدار ۷ ختم می‌شود. (ب) همان متن رشته کـه این بار مقادیر برای هر مکان ممکن از یک پنجره‌ی با طول ۵، به پیمانه‌ی ۱۳ محاسبه شده‌اند. بـا فـرض الگوی $P = ۳۱۴۱۵$، به دنبال پنجره‌هایی می‌گردیم که مقدار آن‌ها به پیمانه‌ی ۱۳ برابر ۷ باشد، چرا کـه $۳۱۴۱۵ \equiv ۷ \pmod{۱۳}$. دو پنجره بدین صورت پیدا شده‌اند، که در شکل با سایه مشخص شده‌اند. اولـی که در متن از مکان ۷ آغاز می‌شود، یک رخ‌داد الگو هم هست، در حالی که دومی، که در متن از مکان ۱۳ آغاز می‌شود، یک آزمایش ناموفق است. (پ) محاسبه‌ی مقدار یک پنجره در زمان ثابت، با داشـتن مقدار پنجره‌ی قبلی. مقدار اولین پنجره ۳۱۴۱۵ است. با حذف رقم با ارزش ۳، جابه‌جایی به چپ (ضرب در ۱۰)، و سپس اضافه کردن رقم کم‌ارزش ۲، مقدار جدید ۴۱۵۱۲ را به دست می‌دهد. با این حال تمـام محاسبات به پیمانه‌ی ۱۳ انجام می‌شوند، به طوری که مقدار اولین پنجره برابر ۷، و مقدار پنجره‌ی جدیـد برابر ۸ خواهد بود.

این مشکل وجود دارد، همان طور که در شکل ۳۲-۵ نشـان داده شـده اسـت: محاسـبه‌ی p و t_s بـه

پیمانه‌ی یک عدد مناسب q. چون محاسبه‌ی p، t_0، و بازگشت (۳۲-۱) را می‌توان به پیمانه‌ی q انجام داد، می‌بینیم که می‌توانیم p به پیمانه‌ی q را در زمان $\theta(m)$ و تمام t_sها را به پیمانه‌ی q در زمان $\theta(n-m+1)$ محاسبه کنیم. پیمانه‌ی q معمولاً به صورت یک عدد اول انتخاب می‌شود به طوری که $10q$ یک کلمه‌ی کامپیوتر را پر کند، که اجازه می‌دهد که تمام محاسبات مورد نیاز با دقت معمولی قابل انجام باشند. به طور کلی، با یک الفبای d تایی $\{0, 1, \ldots, d-1\}$، q را طوری انتخاب می‌کنیم که dq در یک کلمه‌ی کامپیوتر جا شود، و رابطه‌ی بازگشتی (۳۲-۱) را طوری اصلاح می‌کنیم که به پیمانه‌ی q کار کند، که تبدیل می‌شود به:

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod p \qquad (۲-۳۲)$$

که در آن $h = d^{m-1} \pmod q$ مقدار رقم "۱" در با ارزش‌ترین مکان یک پنجره‌ی متن m رقمی است.

با این همه، راه حل کار کردن به پیمانه‌ی q بی‌عیب نیست، چرا که $t_s \equiv p \pmod q$ نتیجه نمی‌دهد که $t_s = p$. از طرف دیگر اگر $t_s \not\equiv p \pmod q$، آن گاه بدون شک داریم $t_s \neq p$، و جابه‌جایی s نامعتبر است. بنابراین می‌توانیم با استفاده از تست $t_s \equiv p \pmod q$ به عنوان یک مکاشفه‌ی سریع برای رد کردن sهای نامعتبر استفاده کنیم. هر جابه‌جایی s که برای آن داریم $t_s \equiv p \pmod q$ باید باز هم تست شود تا ببینیم که واقعاً معتبر است و یا فقط یک *آزمایش ناموفق* انجام داده‌ایم. این تست را می‌توان با چک کردن صریح $P[1..m] = T[s+1..s+m]$ انجام داد. اگر q به اندازه‌ی کافی بزرگ باشد، آن گاه می‌توانیم امیدوار باشیم که آزمایش‌های ناموفق به حد کافی به ندرت اتفاق می‌افتند که هزینه‌ی چک‌های اضافی، کم باشد.

رویه‌ی زیر این ایده‌ها را به صورت دقیق پیاده‌سازی می‌کند. ورودی‌های رویه، متن T، الگوی P، مبنای d برای استفاده (که معمولاً برابر با $|\Sigma|$ در نظر گرفته می‌شود)، و عدد اول q است.

```
RABIN-KARP-MATCHER(T, P, d, q)
1   n = T.length
2   m = P.length
3   h = d^(m-1) mod q
4   p = 0
5   t0 = 0
6   for i = 1 to m              // preprocessing.
7       p = (dp + P[i]) mod q
8       t0 = (dt0 + T[i]) mod q
9   for s = 0 to n - m          // matching.
10      if p = ts
11          if P[1 .. m] = T[s + 1 .. s + m]
12              print "Pattern occurs with shift" s
13          if s < n - m
14              ts+1 = (d(ts - T[s + 1]h) + T[s + m + 1]) mod q
```

رویه‌ی RABIN-KARP-MATCHER به صورت زیر کار می‌کند. تمام کاراکترها به صورت ارقام مبنای d در نظر گرفته می‌شوند. اندیس‌های روی t فقط برای وضوح بیشتر به کار می‌روند؛ اگر تمام اندیس‌ها را حذف کنیم، باز هم برنامه به درستی کار می‌کند. خط ۳، h را با مقدار رقم با ارزش

پنجره‌ی m رقمی مقدار دهی می‌کند. خطوط ۴-۸ مقادیر p و t_0 را که به ترتیب برابر با $P[1..m] \bmod q$ و $T[1..m] \bmod q$ هستند، محاسبه می‌کنند. حلقه‌ی for در خطوط ۹-۱۴ بر روی تمام جابه‌جایی‌های ممکن s تکرار می‌شود، و ثابت حلقه‌ی زیر را حفظ می‌کند:

- هر بار که خط ۱۰ اجرا می‌شود، $t_s = T[s+1..s+m] \bmod q$.

اگر $p = t_s$ در خط ۱۰، آن گاه در خط ۱۱ چک می‌کنیم که آیا $P[1..m] = T[s+1..m+s]$ برقرار است یا خیر. تمام جابه‌جایی‌های معتبر یافت شده در خط ۱۲ چاپ می‌شوند. اگر $s < n-m$ (که در خط ۱۳ چک می‌شود)، آن گاه حلقه‌ی for حداقل یک بار دیگر اجرا می‌شود، و بنابراین ابتدا خط ۱۴ اجرا می‌شود تا تضمین کند که وقتی دوباره به خط ۱۰ می‌رسیم، ثابت حلقه هنوز برقرار است. خط ۱۴ مقدار $t_{s+1} \bmod q$ را از مقدار $t_s \bmod q$ و مستقیماً با استفاده از تساوی (۳۲-۲) در زمان ثابت محاسبه می‌کند.

RABIN-KARP-MATCHER به $\theta(m)$ زمان پیش‌پردازش نیاز دارد، و زمان تطابق آن در بدترین حالت $\theta((n-m)+1)m)$ است، چرا که (مانند الگوریتم تطابق رشته‌ی ساده‌لوحانه) الگوریتم رابین-کارپ تمام جابه‌جایی‌های معتبر را صریحاً چک می‌کند. اگر $P = a^m$ و $T = a^n$، آن گاه این چک کردن به $\theta((n-m+1)m)$ زمان نیاز دارد، چرا که هر یک از $n-m+1$ جابه‌جایی ممکن، معتبر است.

در بسیاری از کاربردها، انتظار داریم که تعداد جابه‌جایی‌های معتبر کم باشد (مثلاً به یک تعداد ثابت c)؛ در این کاربردها، امیدریاضی زمان تطابق برای الگوریتم فقط $O((n-m+1)+cm) = O(n+m)$ است، به علاوه‌ی زمان مورد نیاز برای پردازش آزمایش‌های ناموفق. می‌توانیم یک تحلیل مکاشفه‌ای بر مبنای این فرض که کاهش مقادیر به پیمانه‌ی q مانند نگاشت تصادفی آن‌ها از Σ^* به $\mathbb{Z}_p$ است، انجام دهیم. (بحث مربوط به استفاده از تقسیم برای درهم‌سازی را در بخش ۱۱-۳-۱ ببینید. فرمول‌بندی و اثبات چنین فرضی مشکل است، ولی یک رویکرد قابل قبول این است که فرض کنیم q به صورت تصادفی از اعداد با اندازه‌ی مناسب انتخاب شده است. این فرمول‌بندی را در این جا ادامه نخواهیم داد.) آن گاه می‌توانیم انتظار داشته باشیم که تعداد آزمایش‌های ناموفق $O(n/q)$ باشد، چرا که احتمال این که یک t_s دلخواه به پیمانه‌ی q هم‌ارز با p باشد، $1/q$ تخمین زده می‌شود. چون $O(n)$ مکان وجود دارد که تست خط ۱۰ برای آن‌ها شکست می‌خورد، و برای هر آزمایش $O(m)$ زمان مصرف می‌کنیم، امیدریاضی زمان اجرای صرف شده توسط الگوریتم رابین-کارپ برابر است با

$$O(n) + O(m(v + n/q))$$

که در آن v تعداد جابه‌جایی‌های معتبر است. اگر $v = O(1)$ باشد، این زمان اجرا برابر است با $O(n)$، و انتخاب می‌کنیم $q \geq m$. یعنی، اگر امیدریاضی تعداد جابه‌جایی‌های معتبر کم ($O(1)$) باشد، و عدد اول q طوری انتخاب شود که از طول الگو بیشتر باشد، آن گاه می‌توانیم انتظار داشته باشیم که رویه‌ی رابین-کارپ فقط از $O(n+m)$ زمان برای تطابق استفاده کند. از آن جایی که $m \leq n$، این زمان مورد انتظار تطابق برابر است با $O(n)$.

تمرین‌ها

۳۲-۲-۱ با کار به پیمانه‌ی $q = 11$، الگوریتم رابین-کارپ با متن $T = 3141592653589793$ و الگوی $P = 26$ چند آزمایش ناموفق انجام می‌دهد.

۳۲-۲-۲ برای مسئله‌ی جستجوی یک متن برای رخ‌داد هر یک از k الگو در یک مجموعه‌ی داده شده، چگونه می‌توان متد رابین-کارپ را گسترش داد؟ با این فرض آغاز کنید که تمام k الگو دارای طول یکسان هستند. سپس راه حل خود را کلی‌تر کنید تا با الگوهای با طول‌های مختلف هم کار کند.

۳۲-۲-۳ نشان دهید که چگونه می‌توان الگوریتم رابین-کارپ را گسترش داد که مسئله‌ی جستجو برای یک الگوی $m \times m$ در یک آرایه از کاراکترهای $n \times n$ را حل کند. (الگو می‌تواند به صورت افقی یا عمودی جابه‌جا شود، ولی نمی‌تواند دوران یابد.)

۳۲-۲-۴ آلیس یک کپی از یک فایل طولانی n بیتی $A = \langle a_{n-1}, a_{n-2}, ..., a_0 \rangle$ دارد، و باب هم به طور مشابه یک فایل n بیتی $B = \langle b_{n-1}, b_{n-2}, ..., b_0 \rangle$ دارد. آلیس و باب می‌خواهند بدانند که آیا فایل‌هایی که دارند یکسان است یا خیر. برای جلوگیری از ارسال کل A یا کل B، آن‌ها از تست احتمالاتی سریع زیر استفاده می‌کنند. آن‌ها مشترکا یک عدد اول $q > 1000n$ به صورت تصادفی انتخاب می‌کنند، و همچنین یک عدد صحیح از $\{0, 1, ..., q-1\}$. آن گاه آلیس مقدار

$$A(x) = \left(\sum_{i=0}^{n} a_i x^i \right) \bmod q$$

را محاسبه می‌کند، و باب هم به همین شکل $B(x)$ را. اثبات کنید که اگر $A \neq B$، شانس این که $A(x) = B(x)$ باشد حداکثر یک در ۱۰۰۰ است، در حالی که اگر دو فایل یکسان باشند، $A(x)$ لزوماً برابر است با $B(x)$. (راهنمایی: تمرین ۳۱-۴-۴ را ببینید.)

۳۲-۳ تطابق رشته با اتوماتاهای متناهی

بسیاری از الگوریتم‌های تطابق رشته، یک اتوماتای متناهی می‌سازند که متن رشته‌ای T را برای تمام رخ‌دادهای الگوی P پویش می‌کنند. این بخش متدی برای ساختن چنین اتوماتایی معرفی می‌کند. این اتوماتاهای تطابق رشته بسیار کارآمد هستند: آن‌ها هر کاراکتر متن را *دقیقاً یک بار* بررسی می‌کنند، و برای بررسی هر کاراکتر زمان ثابتی صرف می‌شود. بنابراین زمان تطابق-پس از انجام پیش‌پردازش الگو برای ساختن اتوماتا- $\theta(n)$ خواهد بود. با این حال، زمان ساختن اتوماتا می‌تواند در صورت بزرگ بودن Σ، زیاد باشد. بخش ۳۲-۴ یک روش هوشمندانه برای حل این مسئله ارائه می‌کند.

این بخش را با تعریف یک اتوماتای متناهی آغاز می‌کنیم. یک اتوماتای خاص برای تطابق رشته را

بررسی می‌کنیم، و نشان می‌دهیم که چطور می‌توان از آن برای یافتن رخ‌دادهای یک الگو در یک متن استفاده کرد. نهایتاً نشان خواهیم داد که چگونه می‌توان اتوماتای تطابق رشته را برای یک الگوی ورودی خاص ساخت.

اتوماتای متناهی

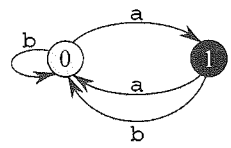
یک اتوماتای متناهی (finite automaton[1]) M، یک ۵ تایی $(Q, q_0, A, \Sigma, \delta)$ است، که در آن

- Q یک مجموعه‌ی متناهی از *حالت* (state)ها است،
- $q_0 \in Q$ یک *حالت شروع* (start state) است،
- $A \subseteq Q$ یک مجموعه‌ی ثابت از *حالت‌های پذیرش* (accepting state) است،
- Σ یک *الفبای ورودی* (input alphabet) متناهی است، و
- δ یک تابع از $Q \times \Sigma$ به Q است، که *تابع انتقال* (transition function) M نام دارد.

اتوماتای متناهی در حالت q_0 آغاز می‌شود و کاراکترهای ورودی را یکی یکی و به صورت یک رشته می‌خواند. اگر یک اتوماتا در حالت q باشد و کاراکتر ورودی a را بخواند، از حالت q به حالت $\delta(q,a)$ حرکت می‌کند (یک «انتقال» انجام می‌دهد). هر گاه حالت فعلی q ماشین M عضوی از A باشد، می‌گوییم M رشته‌ی تا کنون خوانده شده را *می‌پذیرد*. اگر یک ورودی پذیرفته نشود، می‌گوییم *رد* (reject) شده است. شکل ۳۲-۶ این تعریف‌ها را بر روی یک اتوماتای دو حالتی نشان می‌دهد.

وضعیت	ورودی a	ورودی b
0	1	0
1	0	0

(الف)



(ب)

شکل ۳۲-۶ یک اتوماتای ساده‌ی دو حالته با مجموعه حالات $Q = \{0,1\}$، حالت شروع $q_0 = 0$، و الفبای ورودی $\Sigma = \{a,b\}$. (الف) یک نمایش جدولی از تابع انتقال δ. (ب) نمودار حالت-انتقال متناظر. حالت ۱ تنها حالت پذیرش است (که با رنگ سیاه نشان داده شده است). یال‌های جهت‌دار نشان دهنده‌ی انتقال‌ها هستند. مثلاً، یالی که از حالت ۱ به حالت ۰ کشیده شده است و برچسب b دارد، نشان دهنده‌ی $\delta(1,b) = 0$ است. این اتوماتا رشته‌هایی را می‌پذیرد که با تعداد فردی از aها پایان می‌یابند. به طور دقیق‌تر، یک رشته‌ی x پذیرفته می‌شود اگر و فقط اگر $x = yz$، که در آن $y = \varepsilon$ و یا y با یک b تمام می‌شود، و $z = a^k$، که در آن a فرد است. مثلاً دنباله‌ی حالت‌هایی که این اتوماتا برای ورودی *abaaa* طی می‌کند (از جمله حالت شروع) به صورت $\langle 1, 0, 1, 0, 1, 0 \rangle$ است، و بنابراین، این رشته پذیرفته می‌شود. برای ورودی *abbaa*، دنباله‌ی حالت‌ها $\langle 0, 1, 0, 0, 1, 0 \rangle$ است، و بنابراین این ورودی رد می‌شود.

[1] تلفظ صحیح این کلمه اتوماتون است، و اتوماتا (automata) جمع اتوماتون است. ولی در فارسی اصطلاح اتوماتا به صورت مفرد به کار می‌رود، و ما هم در این کتاب این غلط مصطلح را به کار خواهیم برد - م

یک اتوماتای متناهی M حاوی یک تابع φ است، که *تابع حالت نهایی* (final-state function) نام دارد، از Σ^* به Q، به طوری که $\varphi(w)$ حالتی است که M پس از پویش رشته‌ی w در آن قرار می‌گیرد. بنابراین M رشته‌ی w را می‌پذیرد اگر و فقط اگر $\varphi(w) \in A$. تابع φ توسط رابطه‌ی بازگشتی

$$\phi(\varepsilon) = q_0$$
$$\phi(wa) = \delta(\phi(w),a) \quad \text{برای } w \in \Sigma^* \text{ و } a \in \Sigma$$

تعریف می‌شود.

اتوماتای تطابق رشته

یک اتوماتای تطابق رشته برای هر الگوی P وجود دارد؛ این اتوماتا باید در مرحله‌ی پیش‌پردازش از روی الگو ساخته شود. شکل ۳۲-۷ این ساختن را برای الگوی $P = abababaca$ نشان می‌دهد. از این پس، فرض خواهیم کرد که P یک الگوی ثابت است؛ برای اختصار، وابستگی به P را در نمادگذاری‌های خود در نظر نخواهیم گرفت.

برای تعیین اتوماتای تطابق رشته‌ی متناظر با یک الگوی داده شده‌ی $P[1..m]$، ابتدا یک تابع کمکی σ تعریف می‌کنیم، که به آن *تابع پسوند* (suffix function) متناظر با P می‌گوییم. تابع σ یک نگاشت از Σ^* به $\{0, 1, ..., m\}$ است به طوری که $\sigma(x)$ طول بلندترین پیشوند از P است که پسوند x است:

$$\sigma(x) = \max\{k : P_k \sqsupset x\} \tag{۳-۳۲}$$

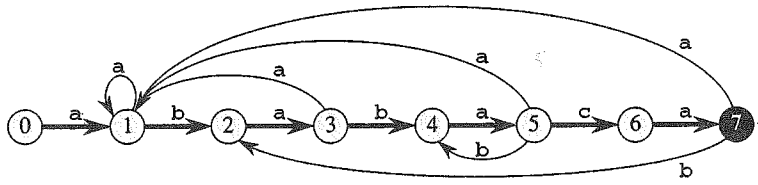
تابع پسوند σ خوش‌تعریف است، چرا که رشته‌ی تهی $P_0 = \varepsilon$ پسوندی برای تمام رشته‌ها است. به عنوان مثال، برای الگوی $P = ab$ داریم $\sigma(\varepsilon) = 0$، $\sigma(ccaca) = 1$، و $\sigma(ccab) = 2$. برای یک الگوی P با طول m داریم $\sigma(x) = m$ اگر و فقط اگر $P \sqsupset x$. از تعریف تابع پسوند نتیجه می‌شود که اگر $x \sqsupset y$، آن گاه $\sigma(x) \le \sigma(y)$.

اتوماتای تطابق رشته‌ی متناظر با یک الگوی $P[1..m]$ را به صورت زیر تعریف می‌کنیم.

- مجموعه‌ی حالت‌های Q برابر است با $\{0,1,...,m\}$. حالت شروع q_0، حالت 0 است، و حالت m تنها حالت پذیرش.
- تابع انتقال به صورت تساوی زیر تعریف می‌شود، برای هر حالت q و کاراکتر a:

$$\delta(q,a) = \sigma(P_q a) \tag{۴-۳۲}$$

تساوی بالا را بدین صورت تعریف می‌کنیم چرا که می‌خواهیم اطلاعات بلندترین پیشوند الگوی P را که تا بدین جا با متن T تطابق یافته است، نگه داریم. در هر لحظه آخرین کاراکترهای خوانده شده از T را در نظر می‌گیریم. برای این که یک زیررشته از T – مثلاً زیررشته‌ای که در $T[i]$ پایان می‌یابد – با یک پیشوند P_j از P تطابق داشته باشد، این پیشوند P_j باید پسوندی از T_i باشد. فرض



(الف)

وضعیت	ورودی a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(ب)

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(پ)

شکل ۳۲-۷ (الف) یک نمودار حالت-انتقال برای اتوماتای تطابق رشته‌ی که تمام رشته‌هایی که با رشته‌ی *ababaca* پایان می‌یابند را می‌پذیرد. حالت ۰ حالت شروع است، و حالت ۷ (که با رنگ سیاه نشان داده شده است) تنها حالت پذیرش. یک یال جهت‌دار که از حالت i به حالت j کشیده شده است، و برچسب a دارد نشان‌دهنده‌ی $\delta(i,a) = j$ است. یال‌های راست رونده‌ای که «ستون فقرات» اتوماتا را تشکیل می‌دهند و در شکل با سایه‌ی پررنگ مشخص شده‌اند، نشان‌دهنده‌ی تطبیق‌های موفق بین الگو و کاراکترهای ورودی هستند. یال‌های چپ رونده نشان‌دهنده‌ی عدم تطابق هستند. بعضی از یال‌های عدم تطابق در این جا نشان داده نشده‌اند؛ برای سادگی، اگر حالت i هیچ یال خروجی با برچسب a، برای یک $a \in \Sigma$ نداشته باشد، آن‌گاه $\delta(i,a) = 0$. (ب) تابع انتقال متناظر δ و الگوی $P = ababaca$. ورودی‌های مربوط به تطابق‌های موفق بین الگو و کاراکترهای ورودی به صورت سایه‌دار مشخص شده‌اند. (پ) عملیات اتوماتا بر روی متن $T = abababacaba$. زیر هر یک از کاراکترهای $T[i]$ حالت $\phi(T_i)$ از اتوماتا بعد از پردازش پسوند T_i داده شده است. یک رخ‌داد از الگو یافت شده است، که در مکان ۹ پایان می‌یابد.

کنید ($q = \phi(T_i)$، به طوری که پس از خواندن T_i، اتوماتا در وضعیت q است. تابع انتقال δ را طوری تعریف می‌کنیم که شماره‌ی وضعیت (q) طول بلندترین پیشوند از P را که با یک پسوند از T_i تطابق دارد، به ما بدهد. به عبارتی، در وضعیت q داریم $P_q \sqsupset T_i$ و $q = \sigma(T_i)$. (هر گاه $q = m$، تمام m کاراکتر P با پسوند T_i تطابق دارند، و بنابراین یک مطابقت یافته‌ایم.) بنابراین چون $\phi(T_i)$ و $\sigma(T_i)$ هر دو برابرند با q، خواهیم دید (در قضیه‌ی ۳۲-۴) که اتوماتا ثابت زیر را حفظ می‌کند:

$$\phi(T_i) = \sigma(T_i) \qquad (۵-۳۲)$$

اگر اتوماتا در وضعیت q باشد و کاراکتر $T[i+1] = a$ را بخواند، آن گاه می‌خواهیم تابع انتقال به وضعیتی برود که مطابق است با بلندترین پیشوندی از P که پسوندی از $T_i a$ است، و این وضعیت عبارت است از $\sigma(P_q a)$. (لم ۳۲-۳ اثبات می‌کند که $\sigma(T_i a) = \sigma(P_q a)$.) بنابراین وقتی اتوماتا در وضعیت q است، می‌خواهیم تابع انتقال کاراکتر a اتوماتا را به وضعیت $\sigma(P_q a)$ ببرد.

دو حالت وجود دارد که باید در نظر بگیریم. در حالت اول $a = P[q+1]$، و کاراکتر a همچنان با الگو مطابقت دارد؛ در این حالت، چون $\delta(q,a) = q+1$، انتقال همچنان روی «ستون» اتوماتا (یال‌های پررنگ در شکل ۳۲-۷) ادامه می‌یابد. در حالت دوم $a \neq P[q+1]$، و a دیگر با الگو تطابق ندارد. در این جا باید یک پیشوند کوچک‌تر در P بیابیم که پسوندی از T_i هم هست. چون در مرحله‌ی پیش‌پردازش، هنگام ساختن اتوماتا، تطابق الگو با خودش بررسی می‌شود، تابع انتقال به سرعت بلندترین پیشوند با شرایط مورد نظر در P را می‌یابد.

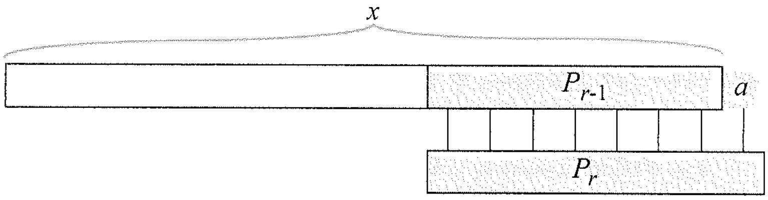
در اتوماتای تطابق رشته‌ی شکل ۳۲-۷، به عنوان مثال $\delta(5, c) = 6$، که نشان‌دهنده‌ی حالت اول است، که در آن تطابق ادامه می‌یابد. برای مشاهده‌ی حالت دوم، دقت کنید که در اتوماتای شکل ۳۲-۷ داریم $\delta(5,b) = 4$. این انتقال را به این خاطر انجام می‌دهیم که اگر اتوماتا در حالت $q = 5$ یک b بخواند، آن گاه $P_q b = ababab$، و طولانی‌ترین پیشوند P که یک پسوند از $ababab$ هم هست، $P_4 = abab$ است.

اکنون برای روشن شدن عملیات یک اتوماتای تطابق رشته، یک برنامه‌ی ساده و کارآمد برای شبیه‌سازی رفتار چنین اتوماتایی (که با تابع انتقال δ مشخص می‌شود) در یافتن رخ‌دادهای یک الگوی P با طول m در یک متن $T[1..n]$ ارائه می‌کنیم. تابع δ را نماینده‌ی اتوماتا در نظر می‌گیریم، چرا که برای هر اتوماتای تطابق رشته برای یک الگو با طول m، مجموعه‌ی حالت‌ها $Q = \{0, 1, ..., m\}$، حالت شروع ۰، و تنها حالت پذیرش m است.

```
FINITE-AUTOMATON-MATCHER(T, δ, m)
1  n = T.length
2  q = 0
3  for i = 1 to n
4      q = δ(q, T[i])
5      if q == m
6          print "Pattern occurs with shift" i – m
```

ساختار ساده‌ی حلقه‌ی FINITE-AUTOMATON-MATCHER ایجاب می‌کند که زمان تطابق آن برای یک رشته به طول n برابر باشد با $\theta(n)$. با این حال، این زمان تطابق شامل زمان پیش‌پردازش مورد نیاز برای محاسبه‌ی تابع انتقال δ نمی‌شود. این مسئله را بعداً مورد بررسی قرار می‌دهیم، بعد از اثبات این که رویه‌ی FINITE-AUTOMATON-MATCHER به درستی کار می‌کند.

عملیات اتوماتا را بر روی یک متن ورودی $T[1..m]$ در نظر بگیرید. می‌خواهیم اثبات کنیم که اتوماتا بعد از پویش کاراکتر $T[i]$ در حالت $\sigma(T_i)$ قرار دارد. چون $\sigma(T_i) = m$ اگر و فقط اگر $P \sqsupset T_i$، ماشین در حالت پذیرش m خواهد بود اگر و فقط اگر در همین لحظه الگوی P پویش شده باشد. برای اثبات این نتیجه، از دو لم زیر در مورد تابع σ استفاده می‌کنیم.



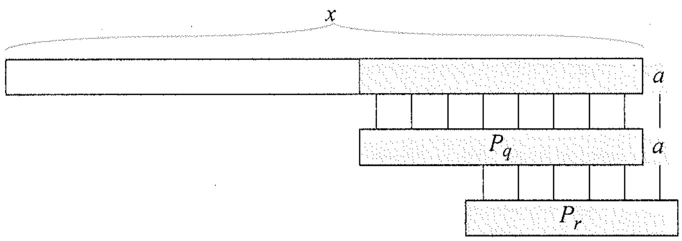
شکل ۳۲-۸ توضیحی برای اثبات لم ۳۲-۲. در شکل نشان داده شده است که $r \le \sigma(x)+1$، که در آن $r = \sigma(xa)$.

لم ۳۲-۲ (نامساوی تابع پسوندی)
برای هر رشته‌ی x و هر کاراکتر a، داریم $\sigma(xa) \le \sigma(x)+1$.

اثبات با مراجعه به شکل ۳۲-۸، فرض کنید $r = \sigma(xa)$. اگر $r = 0$ آن گاه نتیجه‌گیری $\sigma(xa) = r \le \sigma(x)+1$ به صورت بدیهی صحیح است، طبق نامنفی بودن $\sigma(x)$. پس فرض می‌کنیم که $r > 0$. اکنون طبق تعریف σ داریم $P_r \sqsupset xa$. بنابراین با حذف a از انتهای P_r و انتهای xa، داریم $P_{r-1} \sqsupset x$. بنابراین $r-1 \le \sigma(x)$، چرا که $\sigma(x)$ بزرگ‌ترین k است به طوری که $P_k \sqsupset x$، و $\sigma(xa) = r \le \sigma(x)+1$. ■

لم ۳۲-۳ (لم بازگشت تابع پسوند)
برای هر رشته‌ی x و هر کاراکتر a، اگر $q = \sigma(x)$ آن گاه $\sigma(xa) = \sigma(P_q a)$.

اثبات از تعریف σ داریم $P_q \sqsupset x$. همان طور که شکل ۳۲-۹ نشان می‌دهد، همچنین داریم $P_q a \sqsupset xa$. اگر قرار دهیم $r = \sigma(xa)$، آن گاه طبق لم ۳۲-۲ داریم $P_r \sqsupset xa$. چون $P_q a \sqsupset xa$، $P_r \sqsupset xa$، و $|P_r| \le |P_q a|$، لم ۳۲-۱ ایجاب می‌کند که $P_r \sqsupset P_q a$. بنابراین $r \le \sigma(P_q a)$، یعنی، $\sigma(xa) \le \sigma(P_q a)$. ولی داریم $\sigma(P_q a) \le \sigma(xa)$، چرا که $P_q a \sqsupset xa$. بنابراین $\sigma(xa) = \sigma(P_q a)$. ■



شکل ۳۲-۹ توضیحی برای اثبات لم ۳۲-۳. در شکل نشان داده شده است که $r = \sigma(P_q a)$، که در آن $q = \sigma(x)$ و $r = \sigma(xa)$.

اکنون آماده هستیم که قضیه‌ی اصلی خود را که رفتار یک اتوماتای تطابق رشته بر روی یک متن را توصیف می‌کند، اثبات کنیم. همان طور که در بالا گفته شد، این قضیه نشان می‌دهد که اتوماتا صرفا در هر مرحله، اطلاعات بلندترین پیشوند الگو را که پسوندی از آن چه تا کنون خوانده شده است هم هست را نگه می‌دارد. به عبارت دیگر، اتوماتا ثابت (۳۲-۵) را حفظ می‌کند.

قضیه‌ی ۳۲-۱ اگر φ تابع حالت نهایی یک اتوماتای تطابق رشته برای یک الگوی داده شده‌ی P باشد، و $T[1..n]$ یک متن ورودی برای اتوماتا، آن گاه

$$\varphi(T_i) = \sigma(T_i)$$

برای $i = 0, 1, ..., n$.

اثبات اثبات به وسیله‌ی استقرا بر روی i انجام می‌شود. برای $i = 0$، قضیه بدیهتاً درست است، چرا که $T_0 = \varepsilon$. بنابراین $\varphi(T_0) = 0 = \sigma(T_0)$.

اکنون فرض می‌کنیم که $\varphi(T_i) = \sigma(T_i)$ و اثبات می‌کنیم که $\varphi(T_{i+1}) = \sigma(T_{i+1})$. فرض کنید q نشان دهنده‌ی $\varphi(T_i)$ باشد، و a نشان‌دهنده‌ی $T[i+1]$. آن گاه،

$$\begin{aligned}
\varphi(T_{i+1}) &= \varphi(T_i a) && \text{(طبق تعریف } T_{i+1} \text{ و } a\text{)} \\
&= \delta(\varphi(T_i), a) && \text{(طبق تعریف } \varphi\text{)} \\
&= \delta(q, a) && \text{(طبق تعریف } q\text{)} \\
&= \sigma(P_q a) && \text{(طبق تعریف (۳۲-۴) از } \delta\text{)} \\
&= \sigma(T_i a) && \text{(طبق لم ۳۲-۳ و استقرا)} \\
&= \sigma(T_{i+1}) && \text{(طبق تعریف } T_{i+1}\text{)}
\end{aligned}$$

■

طبق قضیه‌ی ۳۲-۴، اگر ماشین در خط ۴ به حالت q وارد شود آن گاه q بزرگ‌ترین مقداری است به طوری که $P_q \sqsupset T_i$. بنابراین در خط ۵ داریم $q = m$ اگر و فقط اگر در همین لحظه رخ‌دادی از الگوی P پویش شده باشد. نتیجه می‌گیریم که FINITE-AUTOMATON-MATCHER به درستی کار می‌کند.

محاسبه‌ی تابع انتقال

رویه‌ی زیر تابع انتقال δ را از روی یک الگوی داده شده‌ی $P[1..m]$ محاسبه می‌کند.

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
1  m = P.length
2  for q = 0 to m
3      for each character a ∈ Σ
4          k = min(m + 1, q + 2)
5          repeat
6              k = k − 1
7          until P_k ⊐ P_q a
8          δ(q, a) = k
9  return δ
```

این رویه $\delta(q, a)$ را از روی تعریف آن در تساوی (۳۲-۴) و به وسیله‌ی یک روش سرراست محاسبه می‌کند. حلقه‌های تودرتو که در خطوط ۲ و ۳ آغاز می‌شوند تمام حالت‌های q و کاراکترهای a را در نظر می‌گیرند، و خطوط ۴-۸، $\delta(q, a)$ را با بزرگ‌ترین k به طوری که $P_k \sqsupset P_q a$ مقداردهی می‌کنند. کد با بزرگ‌ترین kی ممکن آغاز می‌شود، که برابراست با $\min(m, q+1)$، و k را کاهش می‌دهد تا وقتی که $P_k \sqsupset P_q a$، که بالاخره باید رخ دهد، چرا که $P_0 = \varepsilon$ پیشوند تمام رشته‌ها محسوب می‌شود.

زمان اجرای COMPUTE-TRANSITION-FUCTION برابر است با $O(m^3|\Sigma|)$، چرا که حلقه‌ی خارجی به اندازه‌ی یک فاکتور $m|\Sigma|$ در زمان سهم دارد، حلقه‌ی repeat داخلی حداکثر می‌تواند $m+1$ بار اجرا شود، و تست $P_k \sqsupset P_q a$ در خط ۶ می‌تواند حداکثر تا m کاراکتر را با هم مقایسه کند. رویه‌های بسیار سریع‌تری هم وجود دارند؛ با استفاده از محاسبه‌ی اطلاعاتی هوشمندانه در مورد الگو، زمان مورد نیاز برای محاسبه‌ی δ از P می‌تواند تا $O(m|\Sigma|)$ بهبود یابد (تمرین ۳۲-۴-۶ را ببینید). با این رویه‌ی بهبود یافته برای محاسبه‌ی δ، می‌توانیم تمام رخ‌دادهای یک الگو با طول m را در یک متن با طول n بر روی یک الفبای Σ، با $O(m|\Sigma|)$ زمان پیش‌پردازش و $\theta(n)$ زمان مطابقت، بیابیم.

تمرین‌ها

۳۲-۴-۱ اتوماتای تطابق رشته را برای الگوی $P = aabab$ بسازید، و عملیات آن را بر روی متن $T = aaababaabaababaab$ نشان دهید.

۳۲-۴-۲ یک نمودار حالت-انتقال برای اتوماتای تطابق رشته‌ی الگوی $ababbabbababbababbabb$ بر روی الفبای $\Sigma = \{a, b\}$ بکشید.

۳۲-۴-۳ می‌گوییم یک الگوی P *غیر قابل همپوشانی* (nonoverlappable) است اگر $P_k \sqsupset P_q$ نتیجه دهد $k = 0$ یا $k = q$. نمودار حالت-انتقال را برای اتوماتای تطابق رشته‌ی یک الگوی غیر قابل همپوشانی توصیف کنید.

۳۲-۴-۴ ★ با داشتن دو الگوی P و P'، توضیح دهید که چگونه می‌توان یک اتوماتای متناهی ساخت که تمام رخ‌دادهای هر دو الگو را تشخیص دهد. سعی کنید تعداد حالت‌ها را برای اتوماتای خود کمینه کنید.

۳۲-۴-۵ با داشتن یک الگوی P که شامل کاراکترهای شکاف است (تمرین ۳۲-۱-۴ را ببینید)، نشان دهید چگونه می‌توان یک اتوماتای متناهی ساخت که بتواند یک رخ‌داد الگوی P را در متن T در زمان مطابقت $O(n)$ بیابد، که در آن $n = |T|$.

۴-۳۲★ الگوریتم Knuth-Morris-Pratt

اکنون یک الگوریتم تطابق رشته‌ی خطی منسوب به Knuth، Morris، و Pratt ارائه می‌کنیم. این الگوریتم از محاسبه‌ی تابع انتقال δ دوری می‌کند، و زمان تطابق آن $\theta(n)$ است، و فقط از یک تابع کمکی π استفاده می‌کند که در زمان پیش‌پردازش محاسبه و در $\pi[1..m]$ ذخیره شده است. آرایه‌ی π اجازه می‌دهد که تابع انتقال δ به صورت بهینه (با دید سرشکن) در همان زمان نیاز محاسبه شود. به طور کلی، برای هر حالت $q = 0, 1, ..., m$ و هر کاراکتر $a \in \Sigma$، مقدار $\pi[q]$ حاوی اطلاعاتی است که مستقل از a است و برای محاسبه‌ی $\delta(q,a)$ به آن نیاز داریم. چون آرایه‌ی π فقط m ورودی دارد، در حالی که δ، $\theta(m|\Sigma|)$ ورودی دارد، با محاسبه‌ی π به جای δ، به اندازه‌ی یک فاکتور $|\Sigma|$ در زمان پیش‌پردازش صرفه‌جویی می‌کنیم.

تابع پیشوند برای یک الگو

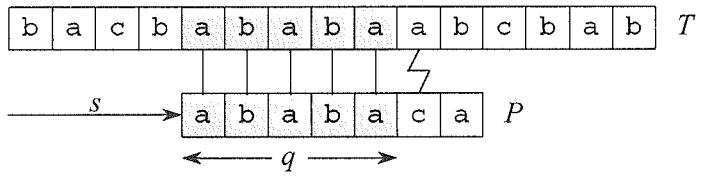
تابع پیشوند π برای یک الگو، اطلاعاتی را در مورد این که الگو چگونه با جابه‌جایی‌هایی از خود مطابقت می‌یابد، ذخیره می‌کند. از این اطلاعات می‌توان برای دوری از جابه‌جایی‌های بی‌مورد در الگوریتم تطابق رشته‌ی ساده‌لوحانه، و یا دوری از پیش‌محاسبه‌ی δ برای یک اتوماتای تطابق رشته استفاده کرد.

عملیات الگوریتم تطابق رشته‌ی ساده‌لوحانه را در نظر بگیرید. شکل ۳۲-۱۰ (الف) یک جابه‌جایی خاص s از یک قالب حاوی الگوی $P = ababaca$ را بر روی متن T نشان می‌دهد. برای این مثال، $q = 5$ تا از کاراکترها با موفقیت مطابقت یافته‌اند، ولی ۶امین کاراکتر الگو با کاراکتر متناظر در متن مطابقت نمی‌یابد. این اطلاعات که q کاراکتر با موفقیت تطابق یافته‌اند، کاراکترهای متناظر متن را تعیین می‌کند. دانستن این q کاراکتر به ما اجازه می‌دهد که سریعا تشخیص دهیم که بعضی از جابه‌جایی‌ها نامعتبر هستند. در مثال شکل، جابه‌جایی $s+1$ لزوماً نامعتبر است، چرا که اولین کاراکتر الگو (a) با کاراکتری از متن متناظر می‌شود که می‌دانیم با دومین کاراکتر الگو (b) تطبیق یافته است. با این حال جابه‌جایی $s' = s+2$ که در بخش (ب) در شکل نشان داده شده است، سه کاراکتر اول الگو را با سه کاراکتر از متن متناظر می‌کند که لزوماً باید با هم مطابقت داشته باشند. به طور کلی، مفید است که جواب سؤال زیر را بدانیم:

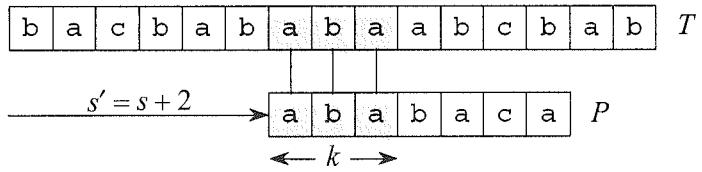
- با دانستن این که کاراکترهای $P[1..q]$ در الگو با کاراکترهای $T[s+1..s+q]$ در متن مطابقت دارند، کوچک‌ترین جابه‌جایی $s' > s$ به طوری که

$$P[1..k] = T[s'+1..s'+k] \qquad (۶-۳۲)$$

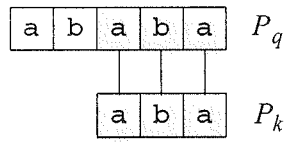
- که در آن $s'+k = s+q$ چیست؟



(الف)



(ب)



(پ)

شکل ۳۲-۱۰ تابع پیشوند π. (الف) الگوی $P = ababaca$ طوری با متن T تراز شده است که اولین $q = 5$ کاراکتر دو رشته با هم تطابق دارند. کاراکترهای مطابقت یافته، که در شکل سایه دارند، با خطوط عمودی به هم متصل شده‌اند. (ب) فقط با استفاده از این اطلاعات که ۵ کاراکتر با هم مطابقت یافته‌اند، می‌توانیم نتیجه بگیریم که یک جابه‌جایی $s+1$ نامعتبر است، ولی جابه‌جایی $s' = s+2$ با تمام اطلاعاتی که تا این جا در مورد متن داریم هم‌خوانی دارد، و بنابراین می‌تواند معتبر باشد. (پ) اطلاعات مفید برای چنین نتیجه‌گیری‌هایی را می‌توان با مقایسه‌ی الگو با خودش به دست آورد. در این جا، می‌بینیم که بلندترین پیشوند P به طوری که یک پشوند اکید از P_5 هم باشد، P_3 است. این اطلاعات از پیش محاسبه و در آرایه‌ی π ذخیره می‌شود، به طوری که $\pi[5] = 3$. با دانستن این که q کاراکتر در جابه‌جایی s با موفقیت مطابقت یافته‌اند، جابه‌جایی بعدی که ممکن است معتبر باشد، $s' = s + (q - \pi[q])$ است، همان طور که در بخش (ب) نشان داده شده است.

به عبارت دیگر با دانستن این که $P_q \sqsupset T_{s+q}$، بلندترین پیشوند اکید P_k از P_q را می‌خواهیم که پسوندی از T_{s+q} هم هست. (از آن جایی که $s+k = s'+q$، اگر s و q به ما داده شده باشد، آن گاه یافتن کوچک‌ترین جابه‌جایی s' معادل است با یافتن بلندترین پیشوند k.) تفاوت $q-k$ در طول این پیشوندهای P را به جابه‌جایی s اضافه می‌کنیم تا به جابه‌جایی جدید s' برسیم، به طوری که $s' = s + (q-k)$. در بهترین حالت، داریم $k = 0$ و $s' = s+q$، و جابه‌جایی‌های $s+1$، $s+2$،...، $s+q-1$ همگی از دور خارج می‌شوند. در هر حالت، در جابه‌جایی جدید s' نیازی نیست که k

کاراکتر اول P را با کاراکترهای متناظر در T مقایسه کنیم، چرا که تساوی (۳۲-۶) تضمین می‌کند که آن‌ها با یکدیگر تطابق دارند.

اطلاعات مورد نیاز را می‌توان با استفاده از مقایسه‌ی الگو با خودش به دست آورد، همان طور که در شکل ۳۲-۱۰ (پ) نشان داده شده است. چون $T[s'+1..s'+k]$ بخشی از متن است که در مورد آن اطلاعات داریم، پسوندی از رشته‌ی P_q است. بنابراین تساوی (۳۲-۵) را می‌توان به صورت جستجو برای بزرگ‌ترین $k < q$ ترجمه کرد، به طوری که $P_k \sqsupset P_q$. در این صورت $s' = s + (q-k)$ جابه‌جایی ممکن بعدی است. خواهیم دید که مناسب است k، تعداد کاراکترهای مطابقت یافته در جابه‌جای جدید s' را نگه داریم، به جای ذخیره‌ی، مثلاً، $s'-s$.

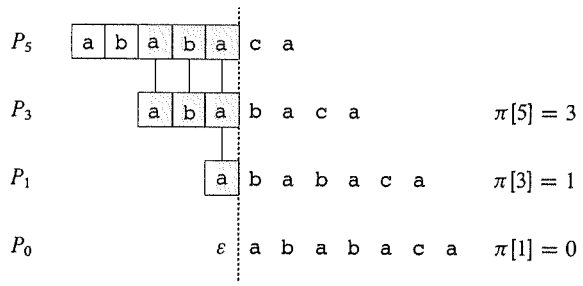
پیش‌پردازش مورد نیاز را به صورت زیر فرمول‌بندی می‌کنیم. با داشتن یک الگوی $P[1..m]$، *تابع پیشوند* (prefix function) برای الگوی P عبارت است از تابع $\pi : \{1, 2, ..., m\} \to \{0, 1, ..., m-1\}$ به طوری که

$$\pi[q] = \max\{k : k < q \text{ و } P_k \sqsupset P_q\}$$

یعنی، $\pi[q]$ برابر است با طول بلندترین پیشوند P که یک پسوند اکید از P_q هم هست. شکل ۳۲-۱۱ (الف) تابع پیشوند π را برای الگوی *abababacba* را به طور کامل نشان می‌دهد.

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

(الف)



(ب)

شکل ۳۲-۱۱ توضیحی از لم ۳۲-۵ برای الگوی $P = ababaca$ و $q = 5$. (الف) تابع π برای الگوی داده شده. چون $\pi[5] = 3$، $\pi[3] = 1$، و $\pi[1] = 0$، با تکرار π به دست می‌آوریم $\pi^*[5] = \{3, 1, 0\}$. (ب) قالب حاوی الگوی P را به سمت راست حرکت می‌دهیم و توجه می‌کنیم که چه زمانی یک پیشوند P_k از P با یک پسوند اکید از P_5 مطابقت پیدا می‌کند؛ این برای $k = 3, 1, 0$ برقرار است. در شکل، سطر اول P را به دست می‌دهد، و خط عمودی نقطه‌چین دقیقاً بعد از P_5 کشیده شده است. ردیف‌های متوالی تمام جابه‌جایی‌هایی از P را نشان می‌دهند که باعث می‌شوند پیشوندی مانند P_k از P با پسوندی از P_5 مطابقت یابد. کاراکترهای مطابقت یافته‌ی پشت سر هم با سایه مشخص شده‌اند. خطوط عمودی کاراکترهای مطابقت یافته را به هم متصل می‌کنند. بنابراین $\{k : k < 5 \text{ و } P_k \sqsupset P_5\} = \{3, 1, 0\}$. لم ادعا می‌کند که $\{k : k < q \text{ و } P_k \sqsupset P_q\} = \pi^*[q]$ برای تمام qها.

الگوریتم تطابق Knuth-Morris-Pratt در شبه‌کد زیر با عنوان رویه‌ی KMP-MATCHER داده شده است. این رویه مدلی دیگر از FINITE-AUTOMATON-MATCHER است، همان طور که خواهیم دید. KMP-MATCHER رویه‌ی کمکی OMPUTE-PREFIX-FUNCIONT را برای محاسبه‌ی π فراخوانی می‌کند.

```
KMP-MATCHER(T, P)
1   n = T.length
2   m = P.length
3   π = COMPUTE-PREFIX-FUNCTION(P)
4   q = 0                                  // number of characters matched.
5   for i = 1 to n                         // scan the text from left to right.
6       while q > 0 and P[q + 1] ≠ T[i]
7           q = π[q]                       // next character does not match.
8       if P[q + 1] == T[i]
9           q = q + 1                      // next character matches.
10          if q == m                      // is all of P matched?
11              print "Pattern occurs with shift" i – m
12              q = π[q]                   // look for the next match.

COMPUTE-PREFIX-FUNCTION(P)
1   m = P.length
2   let π[1 .. m] be a new array
3   π[1] = 0
4   k = 0
5   for q = 2 to m
6       while k > 0 and P[k + 1] ≠P[q]
7           k = π[k]
8       if P[k + 1] == P[q]
9           k = k + 1
10      π[q] = k
11  return π
```

این دو رویه تشابهات زیادی دارند، چرا که هر دو یک رشته را با یک الگوی P مطابقت می‌دهند: KMP-MATCHER متن T را با الگوی P مطابقت می‌دهد، و COMPUTE-PREFIX-FUNCTION الگوی P را با خودش.

با تحلیل زمان اجرای این رویه‌ها آغاز می‌کنیم. اثبات درستی آن‌ها پیچیده‌تر خواهد بود.

تحلیل زمان اجرا

زمان اجرای COMPUTE-PREFIX-FUNCION برابر است با $\theta(m)$، که با استفاده از متد متراکم از تحلیل سرشکن (بخش ۱۷-۱) آن را اثبات می‌کنیم. تنها بخش مشکل، نشان دادن این است که حلقه‌ی while در خطوط ۶-۷ مجموعا به اندازه‌ی $O(m)$ بار اجرا می‌شود. نشان خواهیم داد که این حلقه حداکثر $m-1$ تکرار انجام می‌دهد. با مشاهداتی روی k آغاز می‌کنیم. ابتدا خط ۴، k را با مقدار ۰ آغاز می‌کند، و تنها راه افزایش k از طریق اجرای خط ۹ است، که حداکثر یک بار به ازای هر تکرار حلقه‌ی for خطوط ۵-۱۰ اجرا می‌شود. بنابراین کل افزایش k حداکثر برابر است با $m-1$. دوم، از آن جایی که هنگام آغاز حلقه‌ی for داریم $k < q$ و تمام تکرارهای حلقه q را افزایش می‌دهند، همیشه

داریم $k < q$. بنابراین مقداردهی‌های خطوط ۳ و ۱۰ تضمین می‌کنند که $\pi[q] < q$ برای هر $q = ۱, ۲, ..., m$، که بدین معنی است که هر تکرار حلقه‌ی while مقدار k را کاهش می‌دهد. سوم، k هیچ گاه منفی نمی‌شود. با کنار هم قرار دادن این نکات می‌بینیم که کل کاهش k در حلقه‌ی while از بالا توسط کل افزایش در k در تمام تکرارهای حلقه‌ی for محدود شده است، که برابر است با $m - ۱$. بنابراین حلقه‌ی while حداکثر $m - ۱$ بار تکرار می‌شود، و COMPUTE-PREFIX-FUNCTION در زمان $\theta(m)$ اجرا می‌شود.

تمرین ۳۲-۴-۴ از شما می‌خواهد با استفاده از یک تحلیل متراکم مشابه نشان دهید که زمان تطابق KMP-MATCHER برابر است با $\theta(n)$.

در مقایسه با FINITE-AUTOMATON-MATCHER، با استفاده از π به جای δ، زمان پیش‌پردازش الگو را از $O(m|\Sigma|)$ به $\theta(m)$ کاهش دادیم، در حالی که کران زمان تطبیق $\theta(n)$ باقی مانده است.

درستی محاسبه‌ی تابع پیشوند

به زودی خواهیم دید که تابع پیشوند π به ما کمک می‌کند تابع انتقال δ را در یک اتوماتای تطابق رشته شبیه‌سازی کنیم. ولی ابتدا باید اثبات کنیم که رویه‌ی COMPUTE-PREFIX-FUNCTION به درستی تابع پیشوند را محاسبه می‌کند. برای انجام این کار باید تمام پیشوندهای P_k را بیابیم که پسوندهای اکید یک پیشوند داده شده‌ی P_q هستند. مقدار $\pi[q]$ بلندترین پیشوند با این شرایط را به ما می‌دهد، ولی لم زیر، که در شکل ۳۲-۱۱ نمایش داده شده، نشان می‌دهد که با تکرار تابع پیشوند π می‌توانیم تمام پیشوندهای P_k را که پسوندهای اکید یک پیشوند P_q هستند، بشماریم. فرض کنید

$$\pi^*[q] = \{\pi[q], \pi^{(۲)}[q], \pi^{(۳)}[q], ..., \pi^{(t)}[q]\}$$

که در آن $\pi^{(i)}[q]$ به صورت تکرار تابع تعریف می‌شود، به طوری که $\pi^{(۰)}[q] = q$ و $\pi^{(i+۱)}[q] = \pi[\pi^{(i)}[q]]$ برای $i \geq ۱$، و دنباله‌ی $\pi^*[q]$ وقتی که به $\pi^{(t)}[q] = ۰$ برسیم، متوقف می‌شود.

لم ۳۲-۵ (لم تکرار تابع پسوند)

فرض کنید P یک الگو با طول m باشد، با تابع پیشوند π. آن گاه برای $q = ۱, ۲, ..., m$ داریم

$$\pi^*[q] = \{k : k < q \text{ و } P_k \sqsupset P_q\}$$

اثبات ابتدا اثبات می‌کنیم که $\pi^*[q] \subseteq \{k : k < q \text{ و } P_k \sqsupset P_q\}$، یا به طور مشابه:

$$i \in \pi^*[q] \text{ نتیجه می‌دهد } P_i \sqsupset P_q. \qquad (۷-۳۲)$$

اگر $i \in \pi^*[q]$، آن گاه داریم $i = \pi^{(u)}[q]$ برای یک $u > ۰$. تساوی (۷-۳۲) را به وسیله‌ی استقرا بر روی u اثبات می‌کنیم. برای $u = ۱$، داریم $i = \pi[q]$، و ادعا برقرار است چرا که $i < q$ و $P_{\pi[q]} \sqsupset P_q$.

با استفاده از روابط $\pi[i] < i$ و $P_i \sqsupset P_{\pi[i]}$، و متعدی بودن $<$ و $\sqsupset$، ادعا را برای تمام iها در $\pi^*[q]$ اثبات می‌کند. بنابراین

$$\pi^*[q] \subseteq \{k : k < q \text{ و } P_k \sqsupset P_q\}$$

با استفاده از برهان خلف اثبات می‌کنیم که $\pi^*[q] \subseteq \{k : k < q \text{ و } P_k \sqsupset P_q\}$. فرض کنید که یک عدد صحیح در مجموعه‌ی $\pi^*[q] - \{k : k < q \text{ و } P_k \sqsupset P_q\}$ وجود دارد، و فرض کنید j بزرگ‌ترین مقدار این مجموعه باشد. چون $\pi[q]$ بزرگ‌ترین مقدار در $\{k : k < q \text{ و } P_k \sqsupset P_q\}$ است، و $\pi[q] \in \pi^*[q]$، باید داشته باشیم $j < \pi[q]$، و بنابراین فرض می‌کنیم 'j کوچک‌ترین مقدار در مجموعه‌ی $\pi^*[q]$ باشد که از j بزرگ‌تر است. (اگر هیچ مقداری در مجموعه‌ی $\pi^*[q]$ موجود نباشد که از j بزرگ‌تر باشد، می‌توانیم انتخاب کنیم $j' = \pi[q]$.) داریم $P_j \sqsupset P_q$ چون $P_k \sqsupset P_q$ و $j \in \{k : k < q\}$، و طبق تساوی (۳۲-۷) و چون $j' \in \pi^*[q]$، داریم $P_{j'} \sqsupset P_q$. از این رو طبق لم ۳۲-۱ داریم $P_j \sqsupset P_{j'}$، و j بزرگ‌ترین مقدار کوچک‌تر از 'j با این خصوصیت است. بنابراین باید داشته باشیم $j = \pi[j']$، و از آن جایی که $j' \in \pi^*[q]$، باید همچنین داشته باشیم $j \in \pi^*[q]$. این تناقض لم را اثبات می‌کند.

الگوریتم COMPUTE-PREFIX-FUNCTION مقدار $\pi[q]$ را به ترتیب برای $q = 1, 2, \ldots, m$ محاسبه می‌کند. مقداردهی $\pi[1]$ با ۰ در خط ۳ از COMPUTE-PREFIX-FUNCTION مطمئنا صحیح است، چرا که $\pi[q] < q$ برای تمام qها. از لم زیر و نتیجه‌ی آن برای اثبات این که COMPUTE-PREFIX-FUNCTINO برای $q > 1$ مقدار $\pi[q]$ را به درستی محاسبه می‌کند، استفاده می‌شود.

لم ۳۲-۵ فرض کنید P یک الگو است با طول m، و π تابع پیشوند برای P. برای $q = 1, 2, \ldots, m$، اگر $\pi[q] > 0$ آن گاه $\pi[q] - 1 \in \pi^*[q-1]$.

اثبات اگر $r = \pi[q] > 0$، آن گاه $r < q$ و $P_r \sqsupset P_q$؛ بنابراین $r - 1 < q - 1$ و $P_{r-1} \sqsupset P_{q-1}$ (با حذف آخرین کاراکتر از P_r و P_q، که مجاز است چون.). بنابراین طبق لم ۳۲-۵ داریم $r - 1 \in \pi^*[q-1]$، و بنابراین $\pi[q] - 1 = r - 1 \in \pi^*[q-1]$.

برای $q = 2, 3, \ldots, m$، زیرمجموعه‌ی $E_{q-1} \subseteq \pi^*[q-1]$ را به صورت زیر تعریف می‌کنیم:

$$\begin{aligned} E_{q-1} &= \{k \in \pi^*[q-1] : p[k+1] = P[q]\} \\ &= \{k : k < q-1 \text{ و } P_k \sqsupset P_{q-1} \text{ و } P[k+1] = P[q]\} \quad \text{(طبق لم ۳۲-۵)} \\ &= \{k : k < q-1 \text{ و } P_{k+1} \sqsupset P_q\} \end{aligned}$$

مجموعه‌ی E_{q-1} حاوی مقادیر $k < q-1$ است که برای آن‌ها داریم $P_k \sqsupset P_{q-1}$ و $P_{k+1} \sqsupset P_q$، چرا که $P[k+1] = P[q]$. بنابراین E_{q-1} حاوی مقادیری از $k \in \pi^*[q-1]$ است به طوری که بتوانیم P_k را به P_{k+1} گسترش دهیم و یک پیشوند اکید از P_q داشته باشیم.

نتیجه‌ی ۷-۳۲ فرض کنید P یک الگو با طول m باشد، و π تابع پیشوند P. برای $q = ۲, ۳, ..., m$ داریم

$$\pi[q] = \begin{cases} ۰ & \text{اگر } E_{q-۱} = \varnothing \\ ۱ + \max\{k \in E_{q-۱}\} & \text{اگر } E_{q-۱} \neq \varnothing \end{cases}$$

اثبات اگر $E_{q-۱}$ تهی باشد، هیچ $k \in \pi^*[q-۱]$ (شامل $k = ۰$) وجود ندارد به طوری که بتوانیم P_k را به $P_{k+۱}$ گسترش دهیم به طوری که یک پیشوند اکید از P_q داشته باشیم. بنابراین $\pi[q] = ۰$.

اگر $E_{q-۱}$ ناتهی باشد، آن گاه برای هر $k \in E_{q-۱}$ داریم $k + ۱ < q$ و $P_{k+۱} \sqsupset P_q$. بنابراین طبق تعریف $\pi[q]$ داریم

$$\pi[q] \geq ۱ + \max\{k \in E_{q-۱}\} \qquad (۸-۳۲)$$

توجه کنید که $\pi[q] > ۰$. فرض کنید $r = \pi[q] - ۱$، پس $r + ۱ = \pi[q]$ و بنابراین $P_{r+۱} \sqsupset P_q$. چون $r + ۱ > ۰$ داریم $P[r+۱] = P[q]$. به علاوه طبق لم ۳۲-۶ داریم $r \in \pi^*[q-۱]$. بنابراین $r \in E_{q-۱}$، و $r \leq \max\{k \in E_{q-۱}\}$، یا به طول معادل،

$$\pi[q] \leq ۱ + \max\{k \in E_{q-۱}\} \qquad (۹-۳۲)$$

ترکیب تساوی‌های (۳۲-۸) و (۳۲-۹) اثبات را کامل می‌کند. ■

اکنون اثبات درستی COMPUTE-PREFIX-FUNCTION را کامل می‌کنیم. در رویه‌ی COMPUTE-PREFIX-FUNCTION، در آغاز هر تکرار حلقه‌ی for در خطوط ۵-۱۰، داریم $k = \pi[q-۱]$. درستی این تساوی توسط خطوط ۳ و ۴ برقرار می‌شود، قبل از این که حلقه برای اولین بار اجرا شود، و به خاطر خط ۱۰ در هر تکرار متوالی حلقه برقرار باقی می‌ماند. خطوط ۶-۹، k را طوری اصلاح می‌کنند که برابر با مقدار صحیح $\pi[q]$ باشد. حلقه‌ی خطوط ۶-۷ در میان تمام مقادیر $k \in \pi^*[q-۱]$ جستجو می‌کند تا مقداری بیابد که برای آن $P[k+۱] = P[q]$؛ در آن لحظه k بزرگ‌ترین مقدار در مجموعه‌ی $E_{q-۱}$ است، به طوری که، طبق نتیجه‌ی ۳۲-۷، بتوانیم $\pi[q]$ را برابر با $k + ۱$ قرار دهیم. اگر چنین kای پیدا نشود، در خط ۸ قرار می‌دهیم $k = ۰$. اگر $P[۱] = P[q]$، آن گاه باید هر دوی k و $\pi[q]$ را با ۱ مقداردهی کنیم؛ در غیر این صورت باید بدون تغییر در k، $\pi[q]$ را برابر با ۰ قرار دهیم. خطوط ۸-۱۰ مقداردهی‌های مناسب را برای k و $\pi[q]$ انجام می‌دهند. این اثبات درستی COMPUTE-PREFIX-FUNCTION را کامل می‌کند.

درستی الگوریتم KMP

رویه‌ی KMP-MATCHER را می‌توان به صورت پیاده‌سازی متفاوت رویه‌ی FINITE-AUTOMATON-MATCHER در نظر گرفت، ولی با استفاده از تابع پیشوند π برای محاسبه‌ی انتقال وضعیت‌ها. خصوصا، اثبات خواهیم کرد که در iامین تکرار حلقه‌های for در هر دوی KMP-MATCHER و FINITE-AUTOMATON-MATCHER، وضعیت q هنگام تست برابری با m مقدار یکسانی دارد (در خط ۱۰ در KMP-MATCHER و خط ۵ در FINITE-AUTOMATON-MATCHER). وقتی بحث کردیم

که KMP-MATCHER از روی رفتار FINITE-AUTOMATON-MATCHER تقلید می‌کند، درستی KMP-MATCHER از درستی FINITE-AUTOMATON-MATCHER نتیجه می‌شود (با این حال، به زودی خواهیم دید که چرا خط ۱۲ رویه‌ی KMP-MATCHER ضروری است).

قبل از این که به صورت رسمی اثبات کنیم که KMP-MATCHER به درستی رفتار FINITE-AUTOMATON-MATCHER را شبیه‌سازی می‌کند، اجازه دهید ببینیم که تابع پیشوند π چگونه جایگزین تابع انتقال δ می‌شود. به خاطر بیاورید که وقتی یک اتوماتای تطابق رشته در وضعیت q است و کاراکتر $a = T[i]$ را از ورودی می‌خواند، به یک وضعیت جدید $\delta(q, a)$ منتقل می‌شود. اگر $a = P[q+1]$، یعنی a همچنان با الگو تطابق دارد، و $\delta(q, a) = q+1$. در غیر این صورت $a \neq P[q+1]$، و a دیگر با الگو تطابق ندارد، و $\delta(q, a) \leq q$. در حالت اول، وقتی a همچنان با الگو تطابق دارد، KMP-MATCHER به وضعیت $q+1$ منتقل می‌شود بدون این که به تابع π رجوع کند: نتیجه‌ی تست حلقه‌ی while در خط ۶ اولین بار منفی است، نتیجه‌ی تست خط ۸ مثبت است، و خط ۹ مقدار q را افزایش می‌دهد.

تابع π زمانی وارد عمل می‌شود که کاراکتر a با الگو مطابقت ندارد، و وضعیت جدید $\delta(q,a)$ یا q است و یا سمت چپ q روی ستون اتوماتا قرار دارد. حلقه‌ی while خطوط ۶-۷ رویه‌ی KMP-MATCHER روی وضعیت‌های $\pi^*[q]$ تکرار را انجام می‌دهد، و زمانی متوقف می‌شود که یا به وضعیتی مانند q' برسد به طوری که a با $P[q'+1]$ تطابق داشته باشد، یا q' به ۰ رسیده باشد. اگر a با $P[q'+1]$ مطابقت بیابد، آن گاه خط ۹ وضعیت جدید را با $q'+1$ مقداردهی می‌کند، که برای این که شبیه‌سازی به درستی کار کند، باید برابر با $\delta(q,a)$ باشد. به عبارت دیگر وضعیت جدید $\delta(q,a)$ یا باید وضعیت ۰ باشد و یا وضعیتی باشد که از یک وضعیت در $\pi^*[q]$ بزرگ‌تر است.

اجازه دهید نگاهی بیاندازیم به مثال شکل‌های ۳۲-۷ و ۳۲-۱۱، که مربوط به الگوی $P = ababaca$ هستند. فرض کنید که اتوماتا در وضعیت $q = 5$ است؛ وضعیت‌های $\pi^*[5]$ به ترتیب نزولی عبارتند از ۳، ۱، و ۰. اگر کاراکتر خوانده شده‌ی بعدی c باشد، آن گاه به سادگی می‌توانیم ببینیم که اتوماتا در هر دوی FINITE-AUTOMATON-MATCHER و KMP-MATCHER به وضعیت $\delta(5,c) = 6$ می‌رود. اکنون فرض کنید که کاراکتر خوانده شده‌ی بعدی b باشد، و اتوماتا باید به وضعیت $\delta(5,b) = 4$ برود. حلقه‌ی while رویه‌ی KMP-MATCHER پس از یک بار اجرای خط ۷ متوقف می‌شود، و به وضعیت $q' = \pi[5] = 3$ می‌رسد. چون $P[q'+1] = P[4] = b$، نتیجه‌ی تست خط ۸ مثبت خواهد بود، و KMP-MATCHER به وضعیت جدید $q'+1 = 4 = \delta(5,b)$ منتقل می‌شود. در نهایت، فرض کنید که کاراکتر خوانده شده‌ی بعدی a باشد، و اتوماتا باید به وضعیت $\delta(5,a) = 1$ برود. سه بار اولی که تست خط ۶ اجرا می‌شود، نتیجه‌ی آن مثبت است. دفعه‌ی اول، می‌بینیم که $P[6] = c \neq a$، و KMP-MATCHER به وضعیت $\pi[5] = 3$ (اولین وضعیت درون $\pi^*[5]$) می‌رود. دفعه‌ی دوم، می‌بینیم که $P[4] = b \neq a$ و انتقال به وضعیت $\pi[3] = 1$ (دومین وضعیت در $\pi^*[5]$) انجام می‌شود. دفعه‌ی سوم می‌بینیم که $P[2] = b \neq a$ و انتقال به وضعیت $\pi[1] = 0$ (آخرین وضعیت در $\pi^*[5]$) انجام می‌شود. حلقه‌ی while پس از رسیدن به وضعیت $q' = 0$ متوقف می‌شود. اکنون خط ۸ درمی‌یابد که $P[q'+1] = P[1] = a$، و

خط ۹ اتوماتا را به وضعیت جدید $q' + ۱ = \delta(۵, a) = ۱ + q'$ منتقل می‌کند.

بنابراین شهود ما این است که KMP-MATCHER وضعیت‌های $\pi^*[q]$ را به ترتیب نزولی بررسی می‌کند، روی یک وضعیت q' متوقف می‌شود، و سپس احتمالاً به وضعیت $q' + ۱$ می‌رود. با این که ممکن است فقط شبیه‌سازی محاسبه‌ی $\delta(q, a)$ کار بسیار زمان‌بری به نظر برسد، به خاطر داشته باشید که به صورت حدی، KMP-MATCHER کندتر از FINITE-AUTOMATON-MATCHER نیست.

اکنون آماده‌ایم که به صورت رسمی صحت الگوریتم KMP را اثبات کنیم. طبق قضیه‌ی ۳۲-۵، پس از هر بار اجرای خط ۴ رویه‌ی FINITE-AUTOMATON-MATCHER داریم $q = \sigma(T_i)$. بنابراین کافی است نشان دهیم که همین خصوصیت برای حلقه‌ی for رویه‌ی KMP-MATCHER برقرار است. اثبات با استفاده از استقرا روی تعداد تکرارهای حلقه انجام می‌شود. در ابتدا، هر دو رویه هنگام اولین ورود به حلقه‌های for مربوطه q را با ۰ مقداردهی می‌کنند. تکرار iام حلقه‌ی for در KMP-MATCHER را در نظر بگیرید، و فرض کنید q' وضعیت آغاز این تکرار حلقه باشد. طبق فرض استقرا داریم $q' = \sigma(T_{i-۱})$. باید نشان دهیم که $q = \sigma(T_i)$ در خط ۱۰. (دوباره، خط ۱۲ را به صورت جداگانه بررسی می‌کنیم.)

وقتی کاراکتر $T[i]$ را در نظر می‌گیریم، بلندترین پیشوند P که پسوندی از T_i هم باشد، یا $P_{q'+۱}$ است (اگر $P[q'+۱] = T[i]$) یا پیشوندی از $P_{q'}$ است (نه لزوماً پیشوند اکید یا غیرتهی). سه حالتی که در آن‌ها $\sigma(T_i) = ۰$، $\sigma(T_i) = q' + ۱$، و $\sigma(T_i) \le q' > ۰$ را به صورت جداگانه در نظر می‌گیریم.

- اگر $\sigma(T_i) = ۰$، آن گاه $P_۰ = \varepsilon$ تنها پیشوندی از P است که پسوندی از T_i هم هست. حلقه‌ی while خطوط ۶-۷ روی مقادیر $\pi^*[q]$ حرکت می‌کند، ولی با این که $P_q \sqsupset T_i$ برای هر $q \in \pi^*[q']$، حلقه هیچ گاه qای نمی‌یابد به طوری که $P[q+۱] = T[i]$. وقتی q به ۰ می‌رسد حلقه متوقف می‌شود، و مسلماً خط ۹ اجرا نمی‌شود. بنابراین در خط ۱۰ داریم $q = ۰$، و $q = \sigma(T_i)$.
- اگر $\sigma(T_i) = q' + ۱$، آن گاه $P[q'+۱] = T[i]$، و تست حلقه‌ی while در خط ۶ اولین بار نتیجه‌ی منفی بازمی‌گرداند. خط ۹ اجرا شده و q را افزایش می‌دهد، به طوری که پس از آن داریم $q = q' + ۱ = \sigma(T_i)$.
- اگر $\sigma(T_i) \le q' > ۰$، آن گاه حلقه‌ی while خطوط ۶-۷ حداقل یک تکرار انجام می‌دهد، و به ترتیب نزولی هر مقدار $q \in \pi^*[q']$ را بررسی می‌کند تا روی یک $q < q'$ متوقف شود. بنابراین P_q بلندترین پیشوند $P_{q'}$ است که برای آن داریم $P[q+۱] = T[i]$، به طوری که وقتی حلقه‌ی while پایان می‌یابد، $q + ۱ = \sigma(P_{q'}, T[i])$. چون $q' = \sigma(T_{i-۱})$، لم ۳۲-۳ ایجاب می‌کند که $\sigma(T_{i-۱} T[i]) = \sigma(P_{q'}, T[i])$. بنابراین وقتی حلقه‌ی while پایان می‌یابد داریم

$$\begin{aligned} q + ۱ &= \sigma(P_{q'}, T[i]) \\ &= \sigma(T_{i-۱} T[i]) \\ &= \sigma(T_i) \end{aligned}$$

پس از این که خط ۹ مقدار q را افزایش می‌دهد داریم $q = \sigma(T_i)$.

خط ۱۲ در KMP-MATCHER برای پرهیز از ارجاع‌های ممکن به $P[m+1]$ پس از یافتن یک رخ‌داد P ضروری است. (بحث انجام شده در مورد برقراری $q = \sigma(T_{i-1})$ در تکرار بعدی خط ۶ همچنان معتبر است، طبق راهنمایی داده شده در تمرین ۳۲-۴-۸: $\delta(m,a) = \delta(\pi[m],a)$ ، یا به طور معادل $\sigma(Pa) = \sigma(P_{\pi[m]}a)$ برای $a \in \Sigma$.) بحث باقی مانده برای درستی الگوریتم Knuth-Morris-Pratt از درستی FINITE-AUTOMATON-MATCHER نتیجه می‌شود، چرا که اکنون می‌توانیم ببینیم که KMP-MATCHER رفتار FINITE-AUTOMATON-MATCHER را شبیه‌سازی می‌کند.

تمرین‌ها

۳۲-۴-۱ تابع پیشوند π را برای الگوی *ababbabbabbababbabb* محاسبه کنید.

۳۲-۴-۲ یک کران بالا بر روی اندازه‌ی $\pi^*[q]$ به عنوان تابعی از q بدهید. با یک مثال نشان دهید که کران شما نزدیک است.

۳۲-۴-۳ توضیح دهید که چگونه می‌توان با بررسی تابع π برای رشته‌ی PT (رشته‌ای با طول $m+n$ که از اتصال P و T به دست می‌آید)، رخ‌دادهای الگوی P را در متن T یافت.

۳۲-۴-۴ با استفاده از یک تحلیل متراکم نشان دهید که زمان اجرای KMP-MATHCER برابر است با $\theta(n)$.

۳۲-۴-۵ با استفاده از یک تابع پتانسیل نشان دهید که زمان اجرای KMP-MATCHER برابر است با $\theta(n)$.

۳۲-۴-۶ نشان دهید که چگونه می‌توان با جایگزینی رخ‌دادهای π با π' در خط ۷ (و نه خط ۱۲)، KMP-MATCHER را بهبود بخشید، که در آن π' به صورت بازگشتی برای $q = 1,2,...,m$ و طبق تساوی زیر تعریف می‌شود:

$$\pi'[q] = \begin{cases} 0 & \text{اگر } \pi[q] = 0 \\ \pi'[\pi[q]] & \text{اگر } \pi[q] \neq 0 \text{ و } P[\pi[q]+1] = P[q+1] \\ \pi[q] & \text{اگر } \pi[q] \neq 0 \text{ و } P[\pi[q]+1] \neq P[q+1] \end{cases}$$

توضیح دهید که چرا الگوریتم اصلاح شده صحیح است، و توضیح دهید که چرا این اصلاح یک بهبود برای الگوریتم محسوب می‌شود.

۳۲-۴-۷ یک الگوریتم خطی بدهید که تشخیص دهد که آیا یک رشته‌ی T، جابه‌جایی دورانی یک رشته‌ی T' است یا خیر. مثلاً، *arc* و *car* جابه‌جایی‌های دورانی یکدیگرند.

۳۲-۴-۸★ یک الگوریتم با زمان $O(m|\Sigma|)$ برای محاسبه‌ی تابع انتقال δ برای اتوماتای تطابق رشته برای یک الگوی داده شده‌ی P ارائه کنید. (راهنمایی: اثبات کنید که $\delta(q,a) = \delta(\pi[q],a)$ اگر $q = m$ یا $P[q+1] \neq a$.)

مسائل

۳۲-۱ تطابق رشته بر مبنای فاکتورهای تکرار

فرض کنید y^i نشان‌دهنده‌ی اتصال رشته‌ی y به خودش به تعداد i بار باشد. مثلاً $(ab)^3 = ababab$. می‌گوییم یک رشته‌ی $x \in \Sigma^*$ دارای *فاکتور تکرار* r است اگر $x = y^r$ برای یک رشته‌ی $y \in \Sigma^*$ و یک $r > 0$. فرض کنید $\rho(x)$ نشان دهنده‌ی بزرگ‌ترین r باشد به طوری که x فاکتور تکراری به اندازه‌ی r دارد.

a. یک الگوریتم کارآمد ارائه کنید که با دریافت یک الگوی $P[1..m]$، مقدار $\rho(P_i)$ را برای $i = 1, 2, ..., m$ محاسبه می‌کند. زمان اجرای الگوریتم شما چقدر است؟

b. برای هر الگوی $P[1..m]$، فرض کنید $\rho^*(P)$ به صورت $\max_{1 \le i \le m} \rho(P_i)$ تعریف شود. اثبات کنید که اگر الگوی P به صورت تصادفی از مجموعه‌ی تمام رشته‌های دودویی به طول m انتخاب شود، آن گاه امیدریاضی مقدار $\rho^*(P)$، $O(1)$ است.

c. بحث کنید که الگوریتم تطابق رشته‌ی زیر به درستی تمام رخ‌دادهای الگوی P را در متن $T[1..n]$ در زمان $O(\rho^*(P)n + m)$ می‌یابد.

```
REPETITION-MATCHER(P, T)
1   m = P.length
2   n = T.length
3   k = 1 + ρ*(P)
4   q = 0
5   s = 0
6   while s ≤ n − m
7       if T[s + q + 1] == P[q + 1]
8           q = q + 1
9           if q == m
10              print "Pattern occurs with shift" s
11      if q == m or T[s + q + 1] ≠ P[q + 1]
12          s = s + max(1, ⌈q/k⌉)
13          q = 0
```

این الگوریتم منسوب است به Galil و Seiferas. با گسترش فراوان این ایده‌ها، این دو به یک الگوریتم خطی برای تطابق رشته دست یافته‌اند که فقط از $O(1)$ حافظه علاوه بر حافظه‌ی P و T استفاده می‌کند.

فصل

۳۳

هندسه‌ی محاسباتی

۳۳-۰ مقدمه

هندسه‌ی محاسباتی شاخه‌ای از علوم کامپیوتر است که به بررسی الگوریتم‌هایی برای حل مسائل هندسی می‌پردازد. در مهندسی و ریاضیات مدرن، هندسه‌ی محاسباتی کاربردهای زیادی در بخش‌های مختلف دارد، مانند گرافیک کامپیوتری، روبوتیک، طراحی VLSI، طراحی کامپیوتری (compute-aided design)، مدل‌سازی ملکولی، متالورژی، صنعت، طراحی منسوجات، جنگل‌داری، آمار، و فیلدهای مختلف دیگر. معمولاً ورودی یک مسئله‌ی هندسه‌ی محاسباتی، توصیف مجموعه‌ای از اشیاء هندسی است، مانند مجموعه‌ای از نقاط، پاره‌خط‌ها، و یا رأس‌های یک چندضلعی به ترتیب در جهت چرخش عقربه‌های ساعت. معمولاً خروجی، پاسخی است به یک پرسش در مورد اشیاء، مثلاً این که آیا هیچ یک از خطوط با هم برخورد دارند، و یا یک شیئ هندسی جدید، مثلاً یک پوسته‌ی محدب (کوچک‌ترین چندضلعی محاطی محدب) برای مجموعه‌ای از نقاط.

در این فصل نگاهی خواهیم داشت به چند الگوریتم هندسه‌ی محاسباتی در دو بعد، یا به عبارت دیگر در صفحه. هر شیئ ورودی به صورت مجموعه‌ای از نقاط $\{p_1, p_2, p_3, \ldots\}$ نمایش داده خواهد شد، که در آن $p_i = (x_i, y_i)$ و $x_i, y_i \in \mathbb{R}$. مثلاً یک n ضلعی P به وسیله‌ی یک دنباله‌ی $\langle p_0, p_1, p_2, \ldots, p_{n-1} \rangle$ از رأس‌های آن به ترتیب حضور در مرزهای P توصیف می‌شود. الگوریتم‌های هندسه‌ی محاسباتی در سه بعد، و حتی در فضاهایی با بعدهای بیشتر هم قابل اجرا هستند، ولی تصور چنین مسائلی، و یا جواب‌های آن‌ها، ممکن است بسیار مشکل باشد. از طرفی، حتی در دو بعد هم می‌توانیم مثال‌های خوبی از تکنیک‌های هندسه‌ی محاسباتی را مشاهده کنیم.

بخش ۳۳-۱ نشان می‌دهد که چگونه می‌توان به صورت کارآمد و دقیق به سؤالات ساده در مورد پاره‌خط‌ها پاسخ گفت: سؤالاتی مانند این که آیا یک خط نسبت به یک خط دیگر، که در یک نقطه با آن مشترک است، ساعت‌گرد حرکت می‌کند و یا پادساعت‌گرد، هنگام حرکت از یک پاره‌خط به

پاره‌خط دیگر، جهت چرخش به کدام سمت است، و این که آیا دو پاره‌خط با یکدیگر برخورد دارند یا خیر. بخش ۳۳-۲ تکنیکی معرفی می‌کند به نام «جارو کردن»، که از آن برای توسعه‌ی یک الگوریتم با زمان $O(n \lg n)$ برای تعیین وجود برخورد میان مجموعه‌ای از n پاره‌خط استفاده می‌کنیم. بخش ۳۳-۳ دو الگوریتم «جاروی دورانی» ارائه می‌کند برای محاسبه‌ی پوسته‌ی محدب (کوچک‌ترین چندضلعی محاطی محدب) مجموعه‌ای از n نقطه: پویش Graham، که در زمان $O(n \lg n)$ اجرا می‌شود، و راهپیمایی Jarvis، که در زمان $O(nh)$ اجرا می‌شود، که در آن h تعداد رأس‌های پوسته‌ی محدب است. نهایتاً در بخش ۳۳-۴ یک الگوریتم تقسیم و حل با زمان $O(n \lg n)$ برای یافتن نزدیک‌ترین جفت از نقاط در میان مجموعه‌ای از n نقطه ارائه خواهد شد.

۳۳-۱ خصوصیات پاره‌خط‌ها

بسیاری از الگوریتم‌های هندسه‌ی محاسباتی در این فصل به پاسخ‌هایی در مورد خصوصیات پاره‌خط‌ها نیاز دارند. یک *ترکیب محدب* (convex combination) از دو نقطه‌ی مجزای $p_1 = (x_1, y_1)$ و $p_2 = (x_2, y_2)$ عبارت است از یک نقطه‌ی دلخواه $p_3 = (x_3, y_3)$ به طوری که برای یک α در محدوده‌ی $0 \le \alpha \le 1$ داشته باشیم $x_3 = \alpha x_1 + (1-\alpha) x_2$ و $y_3 = \alpha y_1 + (1-\alpha) y_2$. همچنین می‌نویسیم $p_3 = \alpha p_1 + (1-\alpha) p_2$. به صورت شهودی، p_3 بر روی خط متصل‌کننده‌ی p_1 و p_2، و بین p_1 و p_2 و یا بر روی یکی از خود نقاط p_1 یا p_2 است. با داشتن دو نقطه‌ی مجزای p_1 و p_2، *پاره‌خط* $\overline{p_1 p_2}$ عبارت است از مجموعه‌ی تمام ترکیبات محدب p_1 و p_2. به p_1 و p_2 نقاط انتهایی پاره‌خط $\overline{p_1 p_2}$ می‌گوییم. بعضی مواقع ترتیب p_1 و p_2 اهمیت پیدا می‌کند، و ما از *پاره‌خط جهت‌دار* $\overrightarrow{p_1 p_2}$ صحبت می‌کنیم. اگر p_1 *مبدأ* $(0,0)$ باشد، آن گاه می‌توانیم با پاره‌خط جهت‌دار $\overrightarrow{p_1 p_2}$ مانند *بردار* p_2 برخورد کنیم.

در این بخش سؤالات زیر را بررسی خواهیم کرد:

۱. با داشتن دو پاره‌خط جهت‌دار $\overrightarrow{p_0 p_1}$ و $\overrightarrow{p_0 p_2}$، آیا $\overrightarrow{p_0 p_1}$ نسبت به نقطه‌ی مشترک p_0 در جهت عقربه‌های ساعت از $\overrightarrow{p_0 p_2}$ حرکت می‌کند؟

۲. با داشتن دو پاره‌خط جهت‌دار $\overrightarrow{p_0 p_1}$ و $\overrightarrow{p_1 p_2}$، اگر ابتدا $\overrightarrow{p_0 p_1}$ را طی کنیم و سپس $\overrightarrow{p_1 p_2}$ را، آیا در p_1 یک چرخش چپ انجام می‌دهیم؟

۳. آیا دو پاره‌خط $\overline{p_1 p_2}$ و $\overline{p_3 p_4}$ با یکدیگر برخورد دارند یا خیر؟

برای نقاط داده شده هیچ محدودیتی وجود ندارد.

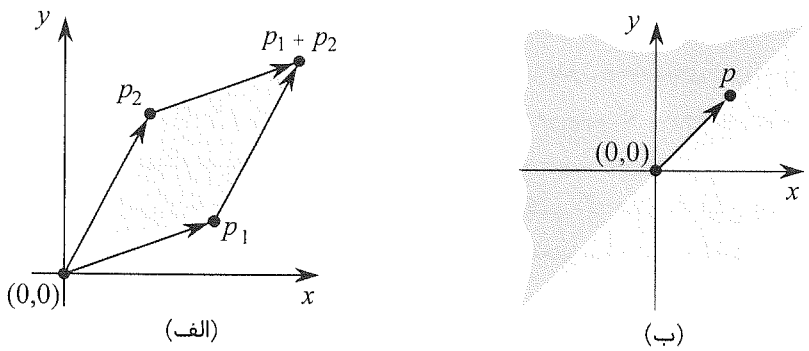
هر یک از این سؤالات را می‌توانیم در زمان $O(1)$ پاسخ دهیم، که تعجبی هم ندارد، چرا که اندازه‌ی ورودی تمام سؤالات $O(1)$ است. به علاوه، در این متدها فقط از جمع، تفریق، ضرب، و مقایسه استفاده می‌شود. برای آن‌ها نه به تقسیم نیاز داریم و نه به توابع مثلثاتی، که هر دو می‌توانند پرهزینه باشند، و باعث به وجود آمدن مشکلاتی در خطای گرد کردن شوند. مثلاً متد «سرراست» برای

تعیین این که دو پاره‌خط با یکدیگر برخورد دارند یا نه – محاسبه‌ی معادله‌ی خط به صورت $y = mx + b$ برای هر پاره‌خط (m شیب است و b عرض از مبدأ y)، یافتن نقطه‌ی برخورد دو خط، و چک کردن این که آیا این نقطه بر روی هر دو پاره‌خط هست یا نه – از تقسیم برای یافتن نقطه‌ی برخورد استفاده می‌کند. وقتی پاره‌خط‌ها تقریباً موازی باشند، این متد به شدت نسبت به دقت تقسیم بر روی کامپیوترهای واقعی حساس است. متد ارائه شده در این بخش، که از تقسیم دوری می‌کند، بسیار دقیق‌تر است.

ضرب خارجی

محاسبه‌ی ضرب خارجی، قلب متدهای مربوط به پاره‌خط در این بخش است. بردارهای p_1 و p_2 را در نظر بگیرید، که در شکل ۳۳-۱ (الف) نشان داده شده‌اند. ضرب خارجی (cross product) $p_1 \times p_2$ را می‌توان به صورت ناحیه‌ی علامت‌دار متواضی‌الاضلاع تشکیل شده از نقاط $(0,0)$، p_1، p_2، و $p_1 + p_2 = (x_1 + x_2, y_1 + y_2)$ در نظر گرفت. یک تعریف معادل، ولی مفیدتر، ضرب خارجی را به صورت دترمینان یک ماتریس ارائه می‌کند[1]:

$$\begin{aligned} p_1 \times p_2 &= \det\begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\ &= x_1 y_2 - x_2 y_1 \\ &= -p_2 \times p_1 \end{aligned}$$



شکل ۳۳-۱ (الف) ضرب خارجی بردارهای p_1 و p_2 عبارت است از ناحیه‌ی علامت‌دار متواضی‌الاضلاع. (ب) ناحیه‌ی با سایه‌ی کم‌رنگ شامل بردارهایی است که نسبت به p در جهت عقربه‌های ساعت قرار دارند، و ناحیه‌ی با سایه‌ی پررنگ شامل بردارهایی است که نسبت به p در جهت عکس عقربه‌های ساعت قرار دارند.

[1] در واقع، ضرب خارجی یک مفهوم سه بعدی است. ضرب خارجی، برداری است عمود بر هر دوی p_1 و p_2 طبق «قانون دست راست»، و اندازه‌ی آن برابر است با $|x_1 y_2 - x_2 y_1|$. با این حال، در این فصل ساده‌تر خواهد بود که ضرب خارجی را به سادگی به صورت مقدار $x_1 y_2 - x_2 y_1$ در نظر بگیریم.

اگر $p_1 \times p_2$ مثبت باشد، آن گاه p_1 نسبت به مبدأ از نقطه‌ی p_2 در جهت عقربه‌های ساعت قرار دارد؛ اگر ضرب خارجی منفی باشد، آن گاه p_1 نسبت به p_2 در جهت عکس عقربه‌های ساعت قرار دارد. (تمرین ۳۳-۱-۱ را ببینید.) شکل ۳۳-۱ (ب) ناحیه‌های در جهت عقربه‌های ساعت، و عکس عقربه‌های ساعت را نسبت به بردار p نشان می‌دهد. یک حالت مرزی زمانی پیش می‌آید که ضرب خارجی ۰ باشد؛ در این حالت بردارها در یک خط هستند، که یا به یک جهت اشاره می‌کنند، و یا به جهت‌های مخالف.

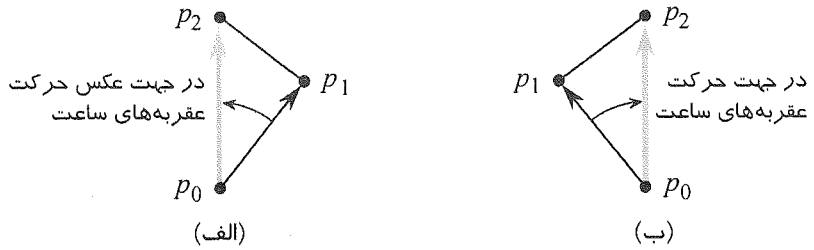
برای تعیین این که یک پاره‌خط $\overrightarrow{p_0p_1}$ نسبت به نقطه‌ی مشترک p_0 از $\overrightarrow{p_0p_2}$ در جهت عقربه‌های ساعت قرار دارد یا خیر، به سادگی p_0 را مبدأ در نظر می‌گیریم. یعنی فرض می‌کنیم $p_1 - p_0$ نشان‌دهنده‌ی بردار $p_1' = (x_1', y_1')$ باشد، که در آن $x_1' = x_1 - x_0$ و $y_1' = y_1 - y_0$، و $p_2 - p_0$ را هم به صورت مشابه تعریف می‌کنیم. سپس ضرب خارجی

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$$

را محاسبه می‌کنیم. اگر این ضرب خارجی مثبت باشد، آن گاه $\overrightarrow{p_0p_1}$ نسبت به $\overrightarrow{p_0p_2}$ در جهت عقربه‌های ساعت قرار دارد؛ در غیر این صورت در نسبت جهت عکس عقربه‌های ساعت است.

تعیین جهت چرخش پاره‌خط‌های متوالی

سؤال بعدی این است که چرخش دو پاره‌خط متوالی $\overline{p_0p_1}$ و $\overline{p_1p_2}$ در نقطه‌ی p_1 به سمت چپ است یا راست. به عبارت دیگر، متدی می‌خواهیم برای تعیین این که چرخش یک زاویه‌ی $\angle p_0p_1p_2$ به کدام جهت است. ضرب خارجی به ما اجازه می‌دهد که این سؤال را بدون محاسبه‌ی زاویه پاسخ دهیم. همان طور که در شکل ۳۳-۲ نشان داده شده است، به سادگی می‌بینیم که پاره‌خط جهت‌دار $\overrightarrow{p_0p_2}$ نسبت به $\overrightarrow{p_0p_1}$ در جهت عقربه‌های ساعت است یا در خلاف جهت عقربه‌های ساعت. برای انجام این کار، ضرب خارجی $(p_1 - p_0) \times (p_2 - p_0)$ را محاسبه می‌کنیم. اگر علامت این ضرب خارجی منفی بود، آن گاه $\overrightarrow{p_0p_2}$ نسبت به $\overrightarrow{p_0p_1}$ در جهت عکس عقربه‌های ساعت قرار دارد، و بنابراین در p_1



شکل ۳۳-۲ استفاده از ضرب خارجی برای تعیین جهت چرخش پاره‌خط‌های متوالی $\overline{p_0p_1}$ و $\overline{p_1p_2}$ در نقطه‌ی p_1. چک می‌کنیم که آیا پاره‌خط جهت‌دار $\overrightarrow{p_0p_2}$ نسبت به پاره‌خط جهت‌دار $\overrightarrow{p_0p_1}$ در جهت عقربه‌های ساعت است یا عکس عقربه‌های ساعت. (الف) اگر در جهت عکس حرکت عقربه‌های ساعت بود، در p_0 یک چرخش چپ خواهیم داشت، و (ب) در غیر این صورت، یک چرخش راست.

یک چرخش چپ انجام می‌شود. یک ضرب خارجی مثبت نشان‌دهنده‌ی نسبت در جهت عقربه‌های ساعت، و در نتیجه چرخش راست است. ضرب خارجی ۰ بدین معنی است که نقاط p_0، p_1، و p_2 بر روی یک خط قرار دارند.

تعیین برخورد یا عدم برخورد دو پاره‌خط

برای تعیین این که دو پاره‌خط با یکدیگر برخورد دارند یا خیر، چک می‌کنیم که آیا هر یک از پاره‌خط‌ها، خط حاوی پاره‌خط دیگر را در بر می‌گیرد یا نه. یک پاره‌خط $\overline{p_1p_2}$ یک خط را *در بر می‌گیرد* (straddle) اگر نقطه‌ی p_1 در یک سمت خط قرار داشته باشد و نقطه‌ی p_2 در سمت دیگر آن. یک حالت مرزی زمانی پیش می‌آید که p_1 یا p_2 دقیقاً روی خط قرار گیرد. دو پاره‌خط با هم برخورد دارند اگر و فقط اگر یکی از (یا هر دوی) شرایط زیر برقرار باشد:

۱. هر پاره‌خط، خط حاوی پاره‌خط دیگر را در بر گیرد.
۲. یک نقطه‌ی پایانی یکی از پاره‌خط‌ها بر روی دیگری قرار داشته باشد. (این شرط در حالت مرزی رخ می‌دهد.)

رویه‌ی زیر این ایده را پیاده‌سازی می‌کند. SEGMENTS-INTERSECT مقدار TRUE را بازمی‌گرداند اگر $\overline{p_1p_2}$ و $\overline{p_3p_4}$ با یکدیگر برخورد داشته باشند، و در غیر این صورت FALSE را بازمی‌گرداند. این رویه، زیرروال‌های DIRECTION و ON-SEGMENT را فراخوانی می‌کند، که اولی جهت‌گیری نسبی دو پاره‌خط را به کمک ضرب خارجی محاسبه می‌کند، و دومی تعیین می‌کند که یک نقطه که می‌دانیم با یک پاره‌خط، در یک امتداد قرار دارد، بر روی پاره‌خط قرار دارد یا خیر.

```
SEGMENTS-INTERSECT(p1, p2, p3, p4)
1   d1 = DIRECTION(p3, p4, p1)
2   d2 = DIRECTION(p3, p4, p2)
3   d3 = DIRECTION(p1, p2, p3)
4   d4 = DIRECTION(p1, p2, p4)
5   if ((d1 > 0 and d2 < 0) or (d1 < 0 and d2 > 0)) and
        ((d3 > 0 and d4 < 0) or (d3 < 0 and d4 > 0))
6       return TRUE
7   elseif d1 == 0 and ON-SEGMENT(p3, p4, p1)
8       return TRUE
9   elseif d2 == 0 and ON-SEGMENT(p3, p4, p2)
10      return TRUE
11  elseif d3 == 0 and ON-SEGMENT(p1, p2, p3)
12      return TRUE
13  elseif d4 = 0 and ON-SEGMENT(p1, p2, p4)
14      return TRUE
15  else return FALSE

DIRECTION(pi, pj, pk)
```

```
1  return (p_k − p_i) × (p_j − p_i)
```

ON-SEGMENT(p_i, p_j, p_k)

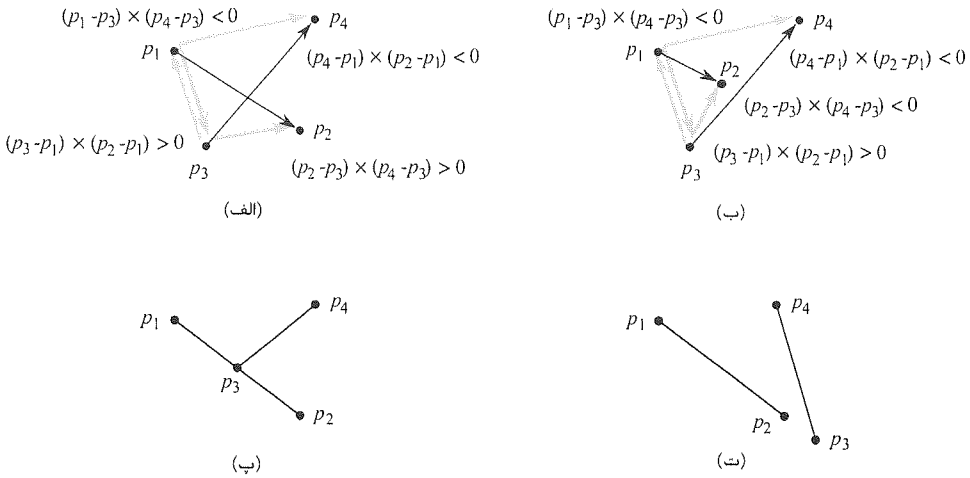
```
1  if min(x_i, x_j) ≤ x_k ≤ max(x_i, x_j) and min(y_i, y_j) ≤ y_k ≤ max(y_i, y_j)
2      return TRUE
3  else return FALSE
```

SEGMENT-INTERSECT به صورت زیر کار می‌کند. خطوط ۱-۴، جهت‌گیری نسبی هر نقطه‌ی پایانی p_i نسبت به خط دیگر را محاسبه می‌کند، که آن را d_i می‌نامیم. اگر تمام جهت‌گیری‌های نسبی غیر صفر باشند، آن گاه به سادگی می‌توانیم تشخیص دهیم که آیا پاره‌خط‌های $\overline{p_1p_2}$ و $\overline{p_3p_4}$ با هم برخورد دارند یا خیر، به صورت زیر. پاره‌خط $\overline{p_1p_2}$ خط حاوی پاره‌خط $\overline{p_3p_4}$ را در بر می‌گیرد اگر جهت‌گیری دو پاره‌خط $\overline{p_3p_1}$ و $\overline{p_3p_2}$ نسبت به $\overline{p_3p_4}$ خلاف هم باشد. در این حالت d_1 و d_2 هم علامت نیستند. به طور مشابه، $\overline{p_3p_4}$ خط حاوی $\overline{p_1p_2}$ را در بر می‌گیرد اگر علامت d_3 و d_4 مخالف باشد. اگر تست خط ۵ درست باشد، در این صورت پاره‌خط‌ها یکدیگر را در بر می‌گیرند، و SEGEMNTS-INTERSECT مقدار TRUE را بازمی‌گرداند. شکل ۳۳-۳ (الف) این حالت را نشان می‌دهد. در غیر این صورت، پاره‌خط‌ها یکدیگر را در بر نمی‌گیرند، ولی ممکن است حالت مرزی پیش آید. اگر تمام جهت‌گیری‌های نسبی غیر صفر باشد، هیچ حالت مرزی پیش نمی‌آید. در این صورت تمام تست‌های ۰ در خطوط ۷-۱۳ رد می‌شوند، و SEGMENTS-INTERSECT در خط ۱۵ مقدار TRUE را بازمی‌گرداند.

حالت مرزی زمانی پیش می‌آید که یکی از جهت‌گیری‌های d_k، ۰ باشد. در این صورت می‌دانیم که p_k با پاره‌خط دیگر در یک خط قرار دارد. حال اگر بین دو نقطه‌ی پایانی پاره‌خط دیگر باشد، پس دقیقاً روی پاره‌خط قرار دارد. رویه‌ی ON-SEGMENT تعیین می‌کند که آیا p_k بین نقاط پایانی پاره‌خط $\overline{p_ip_j}$ قرار دارد یا خیر، که این همان پاره‌خط دیگر است که در خطوط ۷-۱۳ به رویه ارسال می‌شود؛ رویه فرض می‌کند که p_k با پاره‌خط $\overline{p_ip_j}$ در یک خط قرار دارد. شکل ۳۳-۳ (پ) و (ت) حالت‌هایی را که در آن‌ها نقاط روی یک خط قرار دارند نشان می‌دهد. در شکل ۳۳-۳ (پ)، p_3 بر روی $\overline{p_1p_2}$ قرار دارد، و بنابراین SEGMENTS-INTERSECT در خط ۱۲ مقدار TRUE را بازمی‌گرداند. در شکل ۳۳-۳ (ت)، هیچ نقطه‌ی پایانی بر روی پاره‌خط دیگر قرار ندارد، و بنابراین SEGMENTS-INTERSECT در خط ۱۵ مقدار FALSE را بازمی‌گرداند.

کاربردهای دیگر ضرب خارجی

بخش‌های بعدی این فصل، استفاده‌های دیگری از ضرب خارجی را معرفی می‌کنند. در بخش ۳۳-۳، به یک ترتیب برای مجموعه‌ای از نقاط بر مبنای زاویه‌ی قطبی آن‌ها نسبت به یک مبدأ داده شده نیاز خواهیم داشت. همان طور که تمرین ۳۳-۱-۳ از شما می‌خواهد نشان دهید، از ضرب خارجی می‌توان برای انجام مقایسه در رویه‌ی مرتب‌سازی استفاده کرد. در بخش ۳۳-۲، از درخت‌های قرمز-سیاه برای



شکل ۳۳-۳ حالت‌های پیش آمده در رویه‌ی SEGMENTS-INTERSECT. (الف) پاره‌خط‌های $\overline{p_1p_2}$ و $\overline{p_3p_4}$ خطوط حاوی یکدیگر را در بر می‌گیرند. چون $\overline{p_3p_4}$ خط حاوی $\overline{p_1p_2}$ را در بر می‌گیرد، علامت ضرب‌های خارجی $(p_3 - p_1) \times (p_2 - p_1)$ و $(p_4 - p_1) \times (p_2 - p_1)$ متفاوت است. چون $\overline{p_1p_2}$ خط حاوی $\overline{p_3p_4}$ را در بر می‌گیرد، علامت ضرب‌های خارجی $(p_1 - p_3) \times (p_4 - p_3)$ و $(p_2 - p_3) \times (p_4 - p_3)$ متفاوت است. (ب) پاره‌خط $\overline{p_3p_4}$ خط حاوی پاره‌خط $\overline{p_1p_2}$ را در بر می‌گیرد، ولی $\overline{p_1p_2}$ خط حاوی $\overline{p_3p_4}$ را در بر نمی‌گیرد. علامت ضرب‌های خارجی $(p_1 - p_3) \times (p_4 - p_3)$ و $(p_2 - p_3) \times (p_4 - p_3)$ یکی است. (پ) نقطه‌ی p_3 با پاره‌خط $\overline{p_1p_2}$ در یک خط است، و بین p_1 و p_2 قرار دارد. (ت) نقطه‌ی p_3 با پاره‌خط $\overline{p_1p_2}$ در یک خط است، ولی بین p_1 و p_2 نیست. پاره‌خط‌ها با یکدیگر برخورد ندارند.

نگه‌داری ترتیب عمودی مجموعه‌ای از پاره‌خط‌ها استفاده خواهیم کرد. به جای نگه‌داری دقیق مقادیر کلیدها، مقایسه‌ی کلیدها در کد درخت قرمز-سیاه را با یک محاسبه‌ی ضرب خارجی جایگزین خواهیم کرد که تشخیص می‌دهد که از بین دو پاره‌خط که با یک خط عمودی داده شده برخورد دارند، کدام یک بالای دیگری قرار دارد.

تمرین‌ها

۳۳-۱-۱ اثبات کنید که اگر $p_1 \times p_2$ مثبت باشد، آن گاه بردار p_1 از بردار p_2 نسبت به مبدأ $(0,0)$ در جهت عقربه‌های ساعت حرکت می‌کند، و اگر منفی باشد، p_1 از p_2 در جهت خلاف عقربه‌های ساعت حرکت می‌کند.

۳۳-۱-۲ پروفسور Powell ادعا می‌کند که در خط ۱ رویه‌ی ON-SEGMENT، کافی است که بعد x را تست کنیم. نشان دهید که چرا پروفسور اشتباه می‌کند.

۳-۱-۳۳ *زاویه‌ی قطبی* (polar angle) یک نقطه‌ی p_1 نسبت به یک نقطه‌ی مبدأ داده شده‌ی p_0 عبارت است از زاویه‌ی بردار $p_0 - p_1$ در سیستم معمولی محورهای قطبی. مثلاً، زاویه‌ی قطبی (۳,۵) نسبت به (۲,۴) برابر است با زاویه‌ی (۱,۱)، که همان ۴۵ درجه یا $\pi/4$ رادیان درجه است. زاویه‌ی قطبی (۳,۳) نسبت به (۲,۴) برابر است با زاویه‌ی (۱-,۱)، که همان ۳۱۵ درجه، یا $7\pi/4$ رادیان است. شبه‌کدی بنویسید برای مرتب‌سازی دنباله‌ی $\langle p_1, p_2, ..., p_n \rangle$ از n نقطه بر حسب به زاویه‌ی قطبی آن‌ها نسبت به یک مبدأ داده شده‌ی p_0. رویه‌ی شما باید در زمان $O(n \lg n)$ اجرا شده و از ضرب خارجی برای مقایسه‌ی زاویه‌ها استفاده کند.

۴-۱-۳۳ نشان دهید که چگونه می‌توان در زمان $O(n^2 \lg n)$ تعیین کرد که آیا هیچ سه نقطه‌ای در مجموعه‌ای از n نقطه در یک خط قرار دارند یا خیر.

۵-۱-۳۳ یک *چندضلعی* (polygon)، یک منحنی بسته‌ی قطعه‌ای-خطی در صفحه است. یعنی، یک منحنی که بر روی خود پایان می‌یابد، و از دنباله‌ای از پاره‌خط‌ها، با نام *ضلع* (side) تشکیل شده است. نقطه‌ای که دو ضلع متوالی چندضلعی را به هم متصل می‌کند، یک *رأس* (vertex) چندضلعی نام دارد. اگر یک چندضلعی *ساده* (simple) باشد، که معمولاً فرض می‌کنیم این که این طور باشد، با خود برخورد ندارد. مجموعه‌ی نقاطی در صفحه که داخل یک چندضلعی ساده قرار دارند، نقاط *محیطی* (interior) چندضلعی نام دارند، مجموعه‌ی نقاط خود چندضلعی *مرز* (boundary) آن را تشکیل می‌دهند، و مجموعه‌ی نقاط خارجی چندضلعی، نقاط *خارجی* (exterior) آن هستند. یک چندضلعی ساده، *محدب* (convex) است اگر مجموعه‌ی تمام نقاط بر روی پاره خط متصل کننده‌ی هر دو نقطه‌ی محیطی یا مرزی چندضلعی، خود، محیطی یا مرزی باشد. یک رأس یک چندضلعی محدب را نمی‌توان به صورت ترکیبی محدب از دو نقطه‌ی مرزی یا داخلی دیگر در چندضلعی توصیف کرد.
پروفسور Amundsen متد زیر را ارائه کرده‌است برای تعیین این که آیا یک دنباله‌ی $\langle p_0, p_1, ..., p_{n-1} \rangle$ از n نقطه، رأس‌های متوالی یک چندضلعی محدب را تشکیل می‌دهند یا خیر. خروجی "yes" است اگر مجموعه‌ی $\{\angle p_i p_{j+1} p_{j+2} : i = 0, 1, ..., n-1\}$، که در آن جمع اندیس‌ها به پیمانه‌ی n انجام می‌شود، حاوی هر دو چرخش راست و چپ نباشد؛ در غیر این صورت، خروجی "no" است. نشان دهید که با این که این متد در زمان خطی اجرا می‌شود، همیشه جواب صحیح را تولید نمی‌کند. متد پروفسور را طوری اصلاح کنید که همیشه جواب صحیح را در زمان خطی تولید کند.

۶-۱-۳۳ با داشتن یک نقطه‌ی $p_0 = (x_0, y_0)$، پرتوی افقی راست (right horizontal ray) از p_0 عبارت است از مجموعه‌ی نقاط $\{p_i = (x_i, y_i) : x_i \geq x_0 \text{ و } y_i = y_0\}$، یعنی، مجموعه‌ی نقاط سمت راست p_0، به علاوه‌ی خود p_0. نشان دهید چگونه می‌توان در زمان $O(1)$ تشخیص داد که آیا یک پرتوی افقی راست از p_0 با یک پاره‌خط $\overline{p_1 p_2}$ برخورد دارد یا خیر، بدین صورت که مسئله را به مسئله‌ی تعیین وجود برخورد بین دو پاره‌خط کاهش دهید.

۳۳-۱-۷ یک روش برای تعیین این که آیا یک نقطه‌ی p_0 یکی از نقاط محیطی یک چندضلعی ساده ولی نه لزوماً محدب P، هست یا نه، این است که چک کنیم که تعداد برخوردهای یک پرتوی دلخواه از p_0 با نقاط مرزی چندضلعی، فرد باشد، و همچنین p_0 بر روی نقاط مرزی P نباشد. نشان دهید که چگونه می‌توان در زمان $\theta(n)$ تشخیص داد که یک نقطه‌ی p_0 در نقاط محیطی یک nضلعی P هست یا نه. (*راهنمایی:* از تمرین ۳۳-۱-۶ استفاده کنید. اطمینان حاصل کنید که وقتی که پرتو به یکی از رأس‌های چندضلعی برخورد می‌کند، و یا وقتی که پرتو بر روی یک ضلع قرار می‌گیرد، الگوریتم شما به درستی کار می‌کند.)

۳۳-۱-۸ نشان دهید که چگونه می‌توان مساحت یک nضلعی ساده ولی نه لزوماً محدب را در زمان $\theta(n)$ محاسبه کرد. (تمرین ۳۳-۱-۵ را برای تعریف‌های مربوط به چندضلعی‌ها ببینید.)

۳۳-۲ تعیین وجود برخورد میان مجموعه‌ای از پاره‌خط‌ها

این بخش الگوریتمی معرفی خواهد کرد برای تعیین این که آیا در میان مجموعه‌ای از پاره‌خط‌ها، دو پاره‌خط وجود دارد که با هم برخورد داشته باشند یا خیر. این الگوریتم، از تکنیکی استفاده می‌کند به نام «جارو کردن»، که در میان بسیاری از الگوریتم‌های هندسه‌ی محاسباتی معمول است. به علاوه، همان طور که تمرین‌های انتهای این بخش نشان می‌دهند، از این الگوریتم، یا نسخه‌هایی از آن با تغییرات ساده، می‌توان برای حل مسئله‌های دیگری از هندسه‌ی محاسباتی استفاده کرد.

الگوریتم در زمان $O(n \lg n)$ اجرا می‌شود، که در آن n تعداد پاره‌خط‌های مورد نظر است. این الگوریتم فقط مشخص می‌کند که آیا هیچ برخوردی وجود دارد یا نه؛ چاپ برخوردهای موجود در وظایف این الگوریتم نیست. (طبق تمرین ۳۳-۲-۱، در بدترین حالت، برای یافتن *تمام* برخوردهای موجود بین n پاره‌خط به $\Omega(n^2)$ زمان نیاز داریم.)

در *جارو کردن* (sweeping)، یک *خط جاروی* (sweep line) عمودی فرضی از روی مجموعه‌ی داده شده از اشیای هندسی، و معمولاً از چپ به راست، عبور می‌کند. بعد فضایی که خط جارو در طول آن حرکت می‌کند (در این جا بعد x)، به عنوان محور زمان در نظر گرفته می‌شود. جارو کردن، متدی فراهم می‌کند برای مرتب‌کردن عناصر هندسی، معمولاً با قرار دادن آن‌ها در یک ساختمان داده‌ی پویا، و برای سود جستن از رابطه‌ی میان آن‌ها. الگوریتم برخورد پاره‌خط در این بخش، تمام نقاط پایانی پاره‌خط‌ها را به ترتیب از چپ به راست بررسی می‌کند، و هر بار، با برخورد با یک نقطه‌ی پایانی، وجود برخورد میان پاره‌خط‌ها را چک می‌کند.

برای توصیف الگوریتم خود برای تعیین وجود برخورد میان مجموعه‌ای از n پاره‌خط، و اثبات درستی آن، دو فرض را برای سادگی در نظر خواهیم گرفت. اول، فرض می‌کنیم که هیچ پاره‌خط ورودی، عمودی نیست. دوم، فرض می‌کنیم که هیچ سه پاره‌خطی در ورودی، در یک نقطه با یکدیگر برخورد ندارند. تمرین ۳۳-۲-۸ و ۳۳-۲-۹ از شما می‌خواهد نشان دهید که الگوریتم به اندازه‌ی کافی

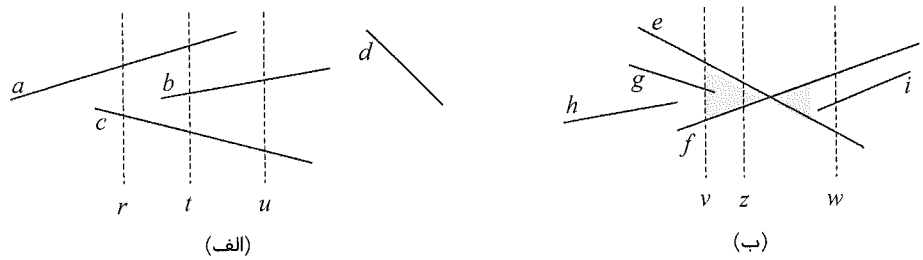
قدرتمند هست که بتوان با تغییری کوچک آن را برای حالت کلی، که ممکن است این فرض‌ها برقرار نباشند، آماده کرد. در واقع، حذف چنین فرض‌هایی که برای سادگی در نظر گرفته می‌شوند، و سروکله زدن با حالت‌های مرزی، معمولاً سخت‌ترین بخش پیاده‌سازی الگوریتم‌های هندسه‌ی محاسباتی و اثبات درستی آن‌ها است.

ترتیب پاره‌خط‌ها

از آن جایی که فرض کرده‌ایم که هیچ پاره‌خط عمودی در ورودی وجود ندارد، هر پاره‌خطی که با یک خط جاروی عمودی برخورد داشته باشد، فقط در یک نقطه با آن برخورد دارد. بنابراین، می‌توانیم پاره‌خط‌هایی را که با یک خط جاروی عمودی برخورد دارند را به ترتیب مختصات y برخورد آن‌ها مرتب کنیم.

به طور دقیق‌تر، دو پاره‌خط s_1 و s_2 را در نظر بگیرید. می‌گوییم این پاره‌خط‌ها در x *قابل مقایسه* (comparable) هستند اگر خط جاروی عمودی گذرا از x با هردوی آن‌ها برخورد داشته باشد. می‌گوییم s_1 در x بالاتر از s_2 است، و می‌نویسیم $s_1 >_x s_2$، اگر s_1 و s_2 در x قابل مقایسه باشند، و نقطه‌ی برخورد s_1 با خط جارو در x بالاتر از نقطه‌ی برخورد s_2 با همان خط جارو باشد. به عنوان مثال، در شکل ۳۳-۴ (الف) رابطه‌های $a >_r c$، $a >_t b$، $b >_t c$، $a >_t c$، و $b >_u c$ را داریم. پاره‌خط d با هیچ پاره‌خط دیگری قابل مقایسه نیست.

برای هر x داده شده، رابطه‌ی "$>_x$" یک ترتیب تام (پیوست ب-۲ را ببینید) بر روی تمام پاره‌خط‌هایی است که با آن خط جارو برخورد دارند. یعنی این رابطه تراگذاری است، و اگر پاره‌خط‌های s_1 و s_2 هر دو در با خط جاروی x برخورد داشته باشند، آن گاه یا $s_1 >_x s_2$ یا $s_2 >_x s_1$، و یا هر دو (اگر s_1 و s_2 روی خط جارو با هم برخورد داشته باشند). (رابطه‌ی $>_x$ بازتابی هم هست، ولی متقارن یا ضدمتقارن نیست.) از طرفی، ممکن است این ترتیب برای مقادیر مختلف x، وقتی پاره‌خط‌ها وارد ترتیب یا از آن خارج می‌شوند، متفاوت باشد. یک پاره‌خط وارد ترتیب می‌شود وقتی



شکل ۳۳-۴ ترتیب پاره‌خط‌ها در خطوط جاروی مختلف. (الف) داریم $a >_r c$، $a >_t b$، $b >_t c$، $a >_t c$، و $b >_u c$. پاره‌خط d با هیچ پاره‌خط دیگری در شکل قابل مقایسه نیست. (ب) وقتی پاره‌خط‌های e و f با یکدیگر برخورد می‌کنند، ترتیب آن‌ها عوض می‌شود: داریم $e >_v f$، ولی $f >_w e$. در ترتیب هر خط جارویی (مانند z) که از ناحیه‌ی سایه‌دار عبور کند، e و f متوالی هستند.

که جارو با نقطه‌ی انتهایی چپ آن برخورد می‌کند، و از ترتیب خارج می‌شود وقتی که جارو با نقطه‌ی نهایی راست آن برخورد می‌کند.

چه اتفاقی رخ می‌دهد وقتی که خط جارو از روی نقطه‌ی برخورد دو پاره‌خط عبور می‌کند؟ همان طور که شکل ۳۳-۴ (ب) نشان می‌دهد، جای آن‌ها در ترتیب تام عوض می‌شود. خط‌های جاروی v و w به ترتیب در سمت چپ و راست نقطه‌ی برخورد پاره‌خط‌های e و f قرار دارند، و داریم $e >_v f$ و $f >_w e$. توجه داشته باشید که چون فرض کردیم که هیچ سه پاره‌خطی در یک نقطه با یکدیگر برخورد ندارند، پس باید یک خط جاروی عمودی x وجود داشته باشد که برای آن، پاره‌خط‌های e و f که با هم برخورد دارند، در ترتیب $>_x$ *متوالی* هستند. در ترتیب تام هر خط جارویی که از ناحیه‌ی سایه‌دار در شکل ۳۳-۴ (ب) عبور می‌کند، مانند z، e و f متوالی هستند.

حرکت‌دادن خط جارو

الگوریتم‌های جارو کردن، معمولاً دو مجموعه‌ی داده را اداره می‌کنند:

۱. *موقعیت خط جارو* (sweep-line status) رابطه‌ی میان اشیائی که با خط جارو برخورد دارند را می‌دهد.

۲. *برنامه‌ی نقاط بررسی* (event-point schedule) دنباله‌ای از مختصات x است، که از چپ به راست مرتب شده‌اند، و موقعیت‌های خط جارو را نشان می‌دهد که در آن‌ها وضعیت بررسی می‌شود. وقتی خط جارو از چپ به راست حرکت می‌کند، هر گاه به یکی از نقاط بررسی برسد متوقف می‌شود، آن نقطه را بررسی می‌کند، و سپس به حرکت ادامه می‌دهد. تغییر موقعیت خط جارو فقط در این نقاط بررسی رخ می‌دهد.

برای بعضی از الگوریتم‌ها (مثلاً الگوریتمی که در تمرین ۳۳-۲-۷ از شما خواسته شده است)، برنامه‌ی نقاط بررسی به صورت پویا در فرآیند اجرای الگوریتم تعیین می‌شود. با این حال الگوریتم حاضر، برنامه‌ی نقاط بررسی را به صورت ایستا و صرفا از روی خصوصیات داده‌های ورودی تعیین می‌کند. به طور خاص، هر نقطه‌ی انتهایی یک پاره‌خط، یک نقطه‌ی بررسی است. نقاط انتهایی را به ترتیب صعودی مختصات x مرتب کرده و از چپ به راست بر روی آن‌ها حرکت می‌کنیم. (اگر دو یا چند نقطه *هم‌عمود* (covertical) باشند، یعنی مختصات x آن‌ها یکی باشد، ابتدا تمام نقاط انتهایی هم‌عمود چپ را قرار می‌دهیم، و بعد نقاط انتهایی هم‌عمود راست را. در میان دسته‌ای از نقاط هم‌عمود چپ (یا راست) نقاط با مختصات y کم‌تر، ابتدا قرار می‌گیرند.) یک پاره‌خط را وقتی در موقعیت خط جارو قرار می‌دهیم که با نقطه‌ی انتهایی چپ آن برخورد کنیم، و وقتی آن را حذف می‌کنیم که با نقطه‌ی انتهایی راست آن برخورد کنیم. اگر دو پاره‌خط در ترتیب تام به صورت متوالی قرار گیرند، بررسی می‌کنیم که آیا این دو با یکدیگر برخورد دارند یا خیر (البته فقط دفعه‌ی اول).

موقعیت خط جارو، یک ترتیب تام T است، که ما نیاز داریم اعمال زیر را بر روی آن انجام دهیم:

- INSERT (T, s): پاره‌خط s را در T درج می‌کند.
- DELETE (T, s): پاره‌خط s را از T حذف می‌کند.

- ABOVE(T, s): پاره‌خطی را باز می‌گرداند که در T دقیقاً بالای s قرار دارد.
- BELOW(T, s): پاره‌خطی را بازمی‌گرداند که در T دقیقاً زیر s قرار دارد.

برای دو پاره‌خط s_1 و s_2 این امکان وجود دارد که در ترتیب تام T هر دو بالای یکدیگر باشند؛ این وضعیت زمانی می‌تواند رخ دهد که s_1 و s_2 دقیقاً روی خط جاروی T با هم برخورد کنند. در چنین حالتی، ترتیب دو پاره‌خط می‌تواند هر یک از دو حالت ممکن باشد.

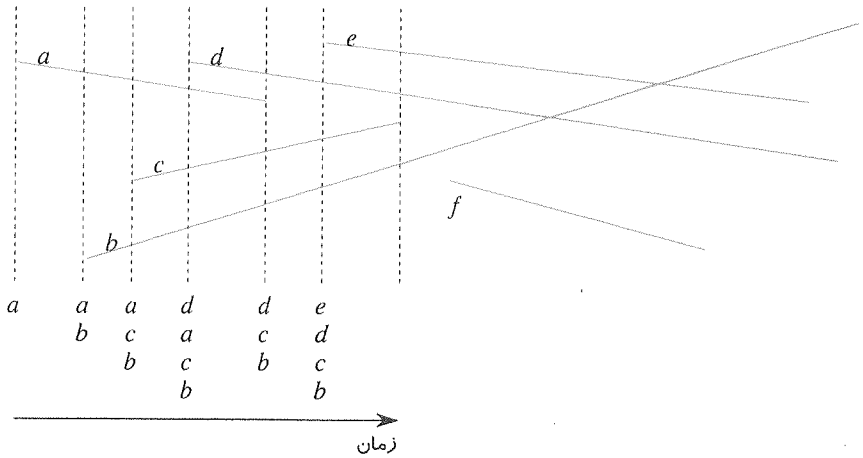
اگر در ورودی n پاره‌خط وجود داشته باشد، می‌توانیم هر یک از اعمال بالا را با استفاده از درختان قرمز-سیاه در زمان $O(\lg n)$ انجام دهیم. به خاطر بیاورید که اعمال درختان قرمز-سیاه در فصل ۱۳ به مقایسه‌ی کلیدها نیاز دارند. می‌توانیم مقایسه‌ی کلیدها را با مقایسه‌هایی جایگزین کنیم که از ضرب خارجی برای تعیین ترتیب نسبی دو پاره‌خط استفاده می‌کنند (تمرین ۳۳-۲-۲ را ببینید).

شبه‌کد برخورد پاره‌خط‌ها

الگوریتم زیر، مجموعه‌ی S از n پاره‌خط را به عنوان ورودی دریافت می‌کند، و در صورتی که یک جفت پاره‌خط در آن وجود داشته باشد که با هم برخورد داشته باشند، مقدار بولین TRUE را بازمی‌گرداند، و در غیر این صورت مقدار FALSE را. ترتیب تام T به کمک یک درخت قرمز-سیاه پیاده‌سازی می‌شود.

```
ANY-SEGMENTS-INTERSECT(S)
1  T = Ø
2  sort the endpoints of the segments in S from left to right,
       breaking ties by putting left endpoints before right endpoints
       and breaking further ties by putting points with lower
       y-coordinates first
3  for each point p in the sorted list of endpoints
4      if p is the left endpoint of a segment s
5          INSERT(T, s)
6          if (ABOVE(T, s) exists and intersects s)
               or (BELOW(T, s) exists and intersects s)
7              return TRUE
8      if p is the right endpoint of a segment s
9          if both ABOVE(T, s) and BELOW(T, s) exist
               and ABOVE(T, s) intersects BELOW(T, s)
10             return TRUE
11         DELETE(T, s)
12 return FALSE
```

شکل ۳۳-۵ اجرای الگوریتم را نشان می‌دهد. خط ۱ ترتیب تام را با مقدار تهی مقداردهی اولیه می‌کند. خط ۲ با مرتب‌سازی $2n$ نقطه‌ی انتهایی از چپ به راست، برنامه‌ی نقاط بررسی را تعیین می‌کند. توجه کنید که خط ۲ را می‌توان به صورت مرتب‌سازی لغت‌نامه‌ای نقاط انتهایی بر روی (x, e, y) هم پیاده‌سازی کرد، که در آن x و y مختصات معمولی هستند، و $e = 0$ برای نقاط انتهایی چپ، و $e = 1$ برای نقاط انتهایی راست.



شکل ۳۳-۵ اجرای ANY-SEGMENTS-INTERSECT. هر خط‌چین نشان دهنده‌ی خط جارو بر روی یک نقطه‌ی بررسی است. غیر از خط جاروی سمت راست، ترتیب نام پاره‌خط‌ها که زیر هر خط جارو قرار دارد، ترتیب تام T در پایان حلقه‌ی for مربوط به همان نقطه‌ی بررسی است. خط جاروی سمت راست مربوط به زمان بررسی نقطه‌ی سمت راست پاره‌خط c است؛ چون خطوط d و b، خط c را احاطه کرده‌اند، و با هم برخورد هم دارند، رویه مقدار TRUE را بازمی‌گرداند.

هر تکرار حلقه‌ی for در خطوط ۳-۱۱ یک نقطه‌ی بررسی p را بررسی می‌کند. اگر p نقطه‌ی انتهایی چپ یک پاره‌خط s باشد، خط ۵، s را به ترتیب تام اضافه می‌کند، و در صورتی که s با یکی از پاره‌خط‌های مجاور خود در ترتیب تام تعریف شده توسط خط جاروی عبور کننده از p برخورد داشته باشد، خطوط ۶-۷ مقدار TRUE را بازمی‌گردانند. (یک حالت مرزی زمانی پیش می‌آید که p بر روی یک پاره‌خط دیگر s' واقع شده باشد. در این حالت، فقط نیاز داریم که s و s' در T متوالی باشند.) اگر p نقطه‌ی انتهایی راست یک پاره‌خط s باشد، آن گاه s باید از ترتیب تام حذف شود. در صورتی که در ترتیب عام تعریف شده توسط خط جاروی عبور کننده از p، برخوردی میان پاره‌خط‌های اطراف s وجود داشته باشد، خطوط ۹-۱۰ مقدار TRUE را بازمی‌گردانند؛ پس از حذف s از ترتیب عام، این خطوط مجاور خواهند شد. اگر این پاره‌خط‌ها با یکدیگر برخورد نداشته باشند، خط ۱۱ پاره‌خط s را از ترتیب عام حذف می‌کند. نهایتاً اگر هیچ برخوردی در بررسی تمام $2n$ نقطه‌ی بررسی وجود نداشته باشد، خط ۱۲ مقدار FALSE را بازمی‌گرداند.

درستی الگوریتم

برای این که نشان دهیم ANY-SEGMENTS-INTERSECT به درستی کار می‌کند، اثبات خواهیم کرد که فراخوانی ANY-SEGMENTS-INTERSECT مقدار TRUE را بازمی‌گرداند اگر برخوردی میان پاره‌خط‌های S وجود داشته باشد.

به سادگی می‌توان دید که ANY-SEGMENTS-INTERSECT مقدار TRUE را (در خطوط ۷ و ۱۰)

بازمی‌گرداند فقط اگر برخوردی میان دو تا از پاره‌خط‌های ورودی بیابد. بنابراین، اگر این رویه مقدار TRUE را بازگرداند، حتماً یک برخورد وجود خواهد داشت.

البته باید عکس این قضیه را هم اثبات کنیم: اگر یک برخورد وجود داشته باشد، آن گاه ANY-SEGMENTS-INTERSECT حتماً مقدار TRUE را بازمی‌گرداند. اجازه دهید فرض کنیم که حداقل یک برخورد در مجموعه‌ی پاره‌خط‌ها وجود دارد. فرض کنید p چپ‌ترین نقطه‌ی برخورد با کم‌ترین مقدار مختصات y باشد. چون هیچ برخوردی در سمت چپ p رخ نمی‌دهد، T در تمام نقاط چپ p صحیح است. چون هیچ سه پاره‌خطی در یک نقطه برخورد ندارند، یک خط جاروی z وجود دارد که برای ترتیب تام آن، a و b متوالی هستند[1]. به علاوه، z در سمت چپ p، و یا روی آن قرار دارد. یک نقطه‌ی انتهایی پاره‌خط، مانند q بر روی خط جاروی z وجود دارد که نقطه‌ی بررسی است که هنگام بررسی آن، a و b در ترتیب تام مجاور می‌شوند. اگر p بر روی خط جاروی z باشد، آن گاه $q = p$. اگر p بر روی خط جاروی z نباشد، آن گاه q در سمت چپ p قرار دارد. در هر دو حالت ترتیب داده شده توسط T دقیقاً قبل از برخورد با q صحیح است. (این جا، جایی است که الگوریتم از ترتیب لغت‌نامه‌ای برای بررسی نقاط بررسی استفاده می‌کند. چون p پایین‌ترین نقطه در میان چپ‌ترین نقاط برخورد است، حتی اگر p بر روی خط جاروی z باشد، و یک نقطه‌ی برخورد دیگر p' بر روی z وجود داشته باشد، نقطه‌ی بررسی $q = p$ قبل از این که نقطه‌ی برخورد دیگر p' بتواند در ترتیب عام T دخالت کند، بررسی می‌شود. به علاوه، حتی اگر p نقطه‌ی انتهایی چپ یک پاره‌خط مانند a، و نقطه‌ی انتهایی راست یک پاره‌خط دیگر مانند b باشد، چون بررسی نقاط انتهایی چپ قبل از بررسی نقاط انتهایی راست انجام می‌شود، بنابراین وقتی اولین بار به پاره‌خط a برخورد می‌کنیم، پاره‌خط b در ترتیب تام T قرار دارد.) نقطه‌ی بررسی q یا توسط ANY-SEGMENTS-INTERSECT بررسی می‌شود و یا نمی‌شود.

اگر q توسط ANY-SEGMENTS-INTERSECT بررسی شود، فقط دو امکان برای عکس‌العمل انجام شده وجود دارد:

۱. یکی از دو پاره‌خط a یا b در T درج می‌شود، و دیگری در ترتیب عام در بالا یا پایین آن قرار دارد. خطوط ۴-۷ این حالت را کشف می‌کنند.

۲. پاره‌خط‌های a و b در T هستند، و یک پاره‌خط در میان آن‌ها در ترتیب عام حذف می‌شود، که باعث می‌شود a و b مجاور شوند. خطوط ۸-۱۱ این حالت را کشف می‌کنند.

در هر دو حالت، نقطه‌ی برخورد p یافت شده و ANY-SEGMENTS-INTERSECT مقدار TRUE را بازمی‌گرداند.

---

[1] اگر اجازه دهیم که سه پاره‌خط در یک نقطه با یکدیگر برخورد داشته باشند، ممکن است یک پاره‌خط مداخله‌گر c وجود داشته باشد که در نقطه‌ی p با a و b برخورد دارد. یعنی ممکن است داشته باشیم $a <_w c$ و $c <_w b$ برای تمام خطوط جاروی w که در سمت چپ p قرار دارند و برای آن‌ها داریم $a <_w b$. تمرین ۳۳-۲-۸ از شما می‌خواهد نشان دهید که حتی اگر سه پاره‌خط در یک نقطه با یکدیگر برخورد داشته باشند، باز هم ANY-SEGMENTS-INTERSECT صحیح است.

اگر نقطه‌ی بررسی q توسط ANY-SEGMETS-INTERSECT بررسی نشود، رویه باید قبل از بررسی تمام نقاط بررسی پایان یافته باشد. این وضعیت فقط زمانی می‌تواند رخ دهد که ANY-SEGMENTS-INTERSECT قبلاً یک نقطه‌ی برخورد یافته و مقدار TRUE را بازگردانده باشد.

بنابراین، اگر یک نقطه‌ی برخورد وجود داشته باشد، ANY-SEGMENTS-INTERSECT مقدار TRUE را بازمی‌گرداند. همان طور که قبلاً دیدیم، اگر ANY-SEGMETS-INTERSECT مقدار TRUE را بازگرداند، حتماً یک برخورد وجود دارد. بنابراین ANY-SEGMENTS-INTERSECT همیشه یک جواب صحیح بازمی‌گرداند.

زمان اجرا

اگر n پاره‌خط در مجموعه‌ی S وجود داشته باشد، آن گاه ANY-SEGMENTS-INTERSECT در زمان $O(n \lg n)$ اجرا می‌شود. خط ۱ به زمان $O(1)$ نیاز دارد. با استفاده از مرتب‌سازی ادغامی و یا مرتب‌سازی هرمی، خط ۲ در زمان $O(n \lg n)$ اجرا می‌شود. چون $2n$ نقطه‌ی بررسی وجود دارد، حلقه‌ی **for** در خطوط ۳–۱۱ حداکثر $2n$ بار تکرار می‌شود. هر تکرار $O(\lg n)$ زمان می‌گیرد، چرا که هر یک از اعمال درختان قرمز–سیاه به $O(\lg n)$ زمان نیاز دارد، و با استفاده از متد بخش ۳۳–۱، هر تست برخورد در زمان $O(1)$ اجرا می‌شود. بنابراین کل زمان عبارت است از $O(n \lg n)$.

تمرین‌ها

۳۳-۲-۱ نشان دهید که ممکن است $\theta(n^2)$ برخورد در مجموعه‌ای از n پاره‌خط وجود داشته باشد.

۳۳-۲-۲ فرض کنید دو پاره‌خط a و b داریم که در x قابل مقایسه هستند. نشان دهید که چطور می‌توانیم در زمان $O(1)$ تعیین کنیم که کدام یک از b $a >_x b$ یا $b >_x a$ برقرار است. فرض کنید که هیچ یک از پاره‌خط‌ها عمودی نیستند. (*راهنمایی:* اگر a و b با یکدیگر برخورد نداشته باشند، می‌توانید فقط از ضرب خارجی استفاده کنید. اگر a و b برخورد داشته باشند – که البته می‌توانید فقط با استفاده از ضرب خارجی آن را تعیین کنید – همچنان می‌توانید فقط از جمع، تفریق، و ضرب استفاده کرده، و از تقسیم پرهیز کنید. البته در کاربرد رابطه‌ی $>_x$ که در این جا از آن استفاده شد، اگر a و b با هم برخورد داشته باشند، می‌توانیم الگوریتم را متوقف کرده و اعلام کنیم که یک برخورد پیدا شده است.)

۳۳-۲-۳ پروفسور Mason پیشنهاد می‌کند که ANY-SEGMENTS-INTERSECT را طوری اصلاح کنیم که به جای بازگشت هنگام یافتن یک برخورد، پاره‌خط‌های برخورد کننده را چاپ کرده و به اجرای تکرار بعدی حلقه‌ی for ادامه دهد. پروفسور نام رویه‌ی حاصل را PRINT-INTERSECTING-SEGMENTS می‌گذارد، و ادعا می‌کند که این رویه تمام برخوردها را به ترتیب از چپ به راست، همان طور که در مجموعه‌ی پاره‌خط‌ها رخ می‌دهند، چاپ می‌کند. پروفسور Dixson مخالف است، و ادعا می‌کند که ایده‌ی پروفسور

Mason صحیح نیست. کدام یک درست می‌گویند؟ آیا PRINT-INTERSECTING-SEGMENTS همیشه اول چپ‌ترین نقطه‌ی برخورد را پیدا می‌کند؟ آیا این رویه همه‌ی نقاط برخورد را پیدا می‌کند؟

۳۳-۲-۴ یک الگوریتم با زمان $O(n \lg n)$ بدهید که تعیین می‌کند که یک n ضلعی ساده است یا خیر.

۳۳-۲-۵ یک الگوریتم با زمان $O(n \lg n)$ بدهید که تعیین می‌کند که آیا دو چندضلعی که مجموعا n ضلع دارند، با یکدیگر برخورد دارند یا خیر.

۳۳-۲-۶ یک *دیسک* (disk) عبارت است از یک دایره به همراه نقاط درونی آن، و با استفاده از مرکز و شعاع آن نمایش داده می‌شود. دو دیسک با یکدیگر برخورد دارند اگر حداقل یک نقطه‌ی مشترک داشته باشند. یک الگوریتم با زمان $O(n \lg n)$ بدهید که تعیین می‌کند که آیا در مجموعه‌ای از n دیسک، دو دیسک وجود دارند که با یکدیگر برخورد داشته باشند یا خیر.

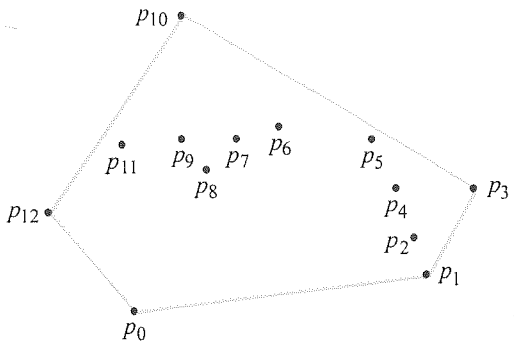
۳۳-۲-۷ با داشتن مجموعه‌ای از n پاره‌خط که در کل شامل k برخورد می‌شود، نشان دهید که چگونه می‌توان در زمان $O((n+k)\lg n)$ تمام k برخورد آن را یافت.

۳۳-۲-۸ بحث کنید که حتی اگر بیش از دو پاره‌خط در یک نقطه با یکدیگر برخورد داشته باشند، باز هم ANY-SEGMENTS-INTERSECT به درستی کار می‌کند.

۳۳-۲-۹ نشان دهید که ANY-SEGMENTS-INTERSECT در حضور پاره‌خط‌های عمودی هم به درستی کار می‌کند، اگر نقطه‌ی انتهایی پایینی آن به صورت نقطه‌ی چپ، و نقطه‌ی انتهایی بالایی آن به صورت نقطه‌ی راست بررسی شود. اگر وجود خطوط عمودی مجاز باشد، جواب شما به تمرین ۳۳-۲-۲ چگونه تغییر می‌کند؟

۳۳-۳ یافتن پوسته‌ی محدب

پوسته‌ی محدب (convex hull) یک مجموعه‌ی Q از نقاط عبارت است از کوچک‌ترین چندضلعی محدب P که هر نقطه‌ی Q درون آن یا روی نقاط مرزی آن قرار گیرد. (تمرین ۳۳-۱-۵ را برای تعریف دقیق یک چندضلعی محدب ببینید.) پوسته‌ی محدب Q را با $CH(Q)$ نشان می‌دهیم. به صورت شهودی، می‌توانیم هر نقطه‌ی Q را به صورت میخی در نظر بگیریم که سر آن از یک تخته بیرون زده است. در این صورت، پوسته‌ی محدب شکلی است که اگر یک کش لاستیکی را به دور میخ‌ها قرار دهیم، تشکیل می‌شود. شکل ۳۳-۶ مجموعه‌ای از نقاط و پوسته‌ی محدب آن را نشان می‌دهد.



شکل ۳۳-۶ مجموعه‌ی نقاط $Q = \{p_0, p_1, \ldots, p_{12}\}$ و پوسته‌ی محدب $CH(Q)$ به رنگ روشن.

در این بخش، دو الگوریتم برای یافتن پوسته‌ی محدب مجموعه‌ای از n نقطه معرفی خواهیم کرد. هر دو الگوریتم، رأس‌های پوسته‌ی محدب را به ترتیب عکس حرکت عقربه‌های ساعت به خروجی می‌دهند. اولی، که به پویش Graham مشهور است، در زمان $O(n \lg n)$ اجرا می‌شود. دومی، که راهپیمایی Jarvis نام دارد، در زمان $O(nh)$ اجرا می‌شود، که در آن h تعداد رأس‌های پوسته‌ی محدب است. همان طور که می‌توان در شکل ۳۳-۶ دید، هر رأس $CH(Q)$ یک نقطه در Q است. هر دو الگوریتم از این خصوصیت استفاده می‌کنند، و تصمیم می‌گیرند که باید کدام یک از نقاط Q را به عنوان رأس‌های پوسته‌ی محدب نگه دارند و کدام یک را رد کنند.

در واقع متدهای مختلفی وجود دارند که پوسته‌ی محدب را در زمان $O(n \lg n)$ محاسبه می‌کنند. هر دو الگوریتم پویش Graham و راهپیمایی Jarvis از تکنیکی به نام «جاروی دورانی» استفاده می‌کنند، که رأس‌ها را به ترتیب زاویه‌ی قطبی آن‌ها از یک رأس مبدأ در نظر می‌گیرند. متدهای دیگری نیز وجود دارند، از جمله:

- در *متد افزایشی* (incremental method)، نقاط از چپ به راست ذخیره می‌شوند، که به یک دنباله‌ی $\langle p_1, p_2, \ldots, p_n \rangle$ ختم می‌شود. در مرحله‌ی iام، پوسته‌ی محدب $i-1$ نقطه‌ی چپ، یعنی $CH(\{p_1, p_2, \ldots, p_{i-1}\})$، بسته به مکان نقطه‌ی iام به هنگام سازی می‌شود، و $CH(\{p_1, p_2, \ldots, p_i\})$ حاصل می‌شود. همان طور که تمرین ۳۳-۳-۶ از شما می‌خواهد نشان دهید، می‌توان این متد را طوری پیاده‌سازی کرد که در زمان $O(n \lg n)$اجرا شود.
- در *متد تقسیم و حل*، در زمان $\theta(n)$ مجموعه‌ی نقاط به دو زیرمجموعه تقسیم می‌شود، که یکی حاوی $\lceil n/2 \rceil$ نقطه‌ی چپ و دیگری حاوی $\lfloor n/2 \rfloor$ نقطه‌ی راست است، پوسته‌ی محدب دو زیرمجموعه به صورت بازگشتی محاسبه می‌شود، و سپس با استفاده از یک متد هوشمندانه، این دو پوسته‌ی محدب در زمان $O(n)$ با هم ترکیب می‌شوند. زمان اجرا به صورت رابطه‌ی بازگشتی آشنای $T(n) = 2T(n/2) + O(n)$ توصیف می‌شود، و بنابراین، زمان اجرای این متد تقسیم و حل $O(n \lg n)$ است.
- *متد هرس و جستجو* (prune-and-search method) مشابه بدترین حالت الگوریتم میانه از بخش ۹-۳

است. این متد بخش بالایی (یا «زنجیر بالایی») پوسته‌ی محدب را بدین صورت می‌یابد که مرتبا کسر ثابتی از نقاط باقی‌مانده را حذف می‌کند تا در نهایت فقط زنجیر بالایی پوسته‌ی محدب باقی بماند. سپس همین کار را برای زنجیر پایینی انجام می‌دهد. این متد به صورت حدی سریع‌ترین است: اگر پوسته‌ی محدب شامل h رأس باشد، الگوریتم در زمان $O(n \lg h)$ اجرا می‌شود.

محاسبه‌ی پوسته‌ی محدب مجموعه‌ای از نقاط در نوع خود مسئله‌ی جذابی است. به علاوه، الگوریتم‌هایی برای مسائل دیگر هندسه‌ی محاسباتی وجود دارند که با محاسبه‌ی پوسته‌ی محدب آغاز می‌شوند. به عنوان مثال، مسئله‌ی *دورترین جفت نقاط* (farthest-pair problem) در دو بعد را در نظر بگیرید: مجموعه‌ای از n نقطه در صفحه داریم، و می‌خواهیم دو نقطه در میان آن‌ها بیابیم که بیشترین فاصله‌ی ممکن را از یکدیگر داشته باشند. همان طور که تمرین ۳۳-۳-۳ از شما می‌خواهد اثبات کنید، این دو نقطه باید رأس‌هایی از پوسته‌ی محدب باشند. با این که این مسئله را در این جا اثبات نمی‌کنیم، می‌توان دورترین جفت نقاط در میان n رأس یک چندضلعی محدب را در زمان $O(n)$ یافت. بنابراین با محاسبه‌ی پوسته‌ی محدب n نقطه‌ی ورودی و در زمان $O(n \lg n)$ و سپس یافتن دورترین جفت نقاط در میان رأس‌های پوسته، می‌توانیم دورترین جفت نقاط در میان n نقطه را در زمان $O(n \lg n)$ بیابیم.

پویش Graham

پویش Graham مسئله‌ی یافتن پوسته‌ی محدب را به کمک یک پشته‌ی S از نقاط کاندیدا حل می‌کند. هر نقطه‌ی مجموعه‌ی ورودی Q یک بار در پشته درج می‌شود، و نقاطی که رأس $CH(Q)$ نیستند از پشته حذف می‌شوند. وقتی الگوریتم پایان می‌یابد، پشته‌ی S دقیقاً حاوی رأس‌های $CH(Q)$ با ترتیب عکس حرکت عقربه‌های ساعت است.

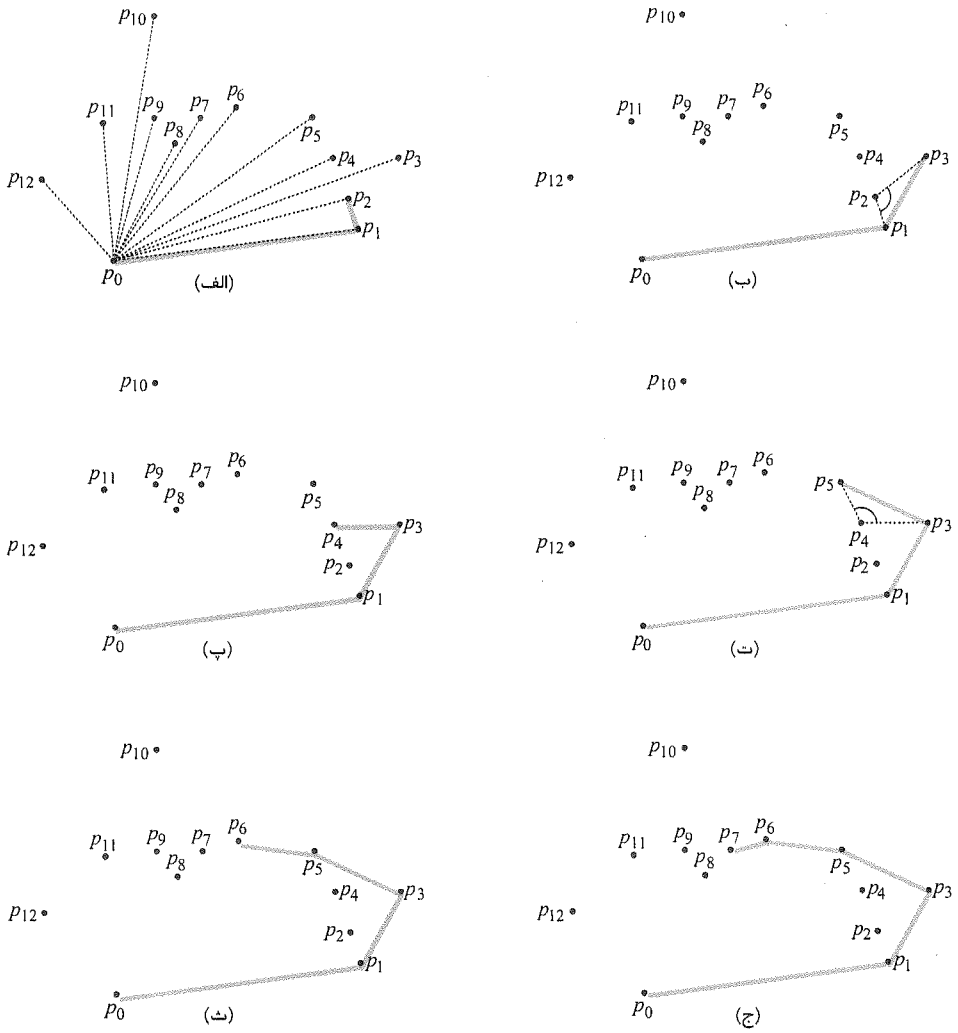
رویه‌ی GRAHAM-SCAN به عنوان ورودی یک مجموعه‌ی Q از نقاط را دریافت می‌کند، که در آن $|Q| \geq ۳$. در رویه از توابع $\text{TOP}(S)$ و $\text{NEXT-TO-TOP}(S)$ استفاده می‌شود، که به ترتیب عنصر بالایی و عنصر قبل از بالایی پشته‌ی S را بازمی‌گردانند، بدون این که پشته را تغییر دهند. همان طور که به زودی اثبات خواهیم کرد، پشته‌ی S بازگردانده شده توسط GRAHAM-SCAN، از بالا به پایین، دقیقاً حاوی رأس‌های $CH(Q)$ به ترتیب عکس حرکت عقربه‌های ساعت است.

```
GRAHAM-SCAN(Q)
1  let p_0 be the point in Q with the minimum y-coordinate,
       or the leftmost such point in case of a tie
2  let ⟨p_1, p_2, ..., p_m⟩ be the remaining points in Q,
       sorted by polar angle in counterclockwise order around p_0
       (if more than one point has the same angle, remove all but
       the one that is farthest from p_0)
3  let S be an empty stack
```

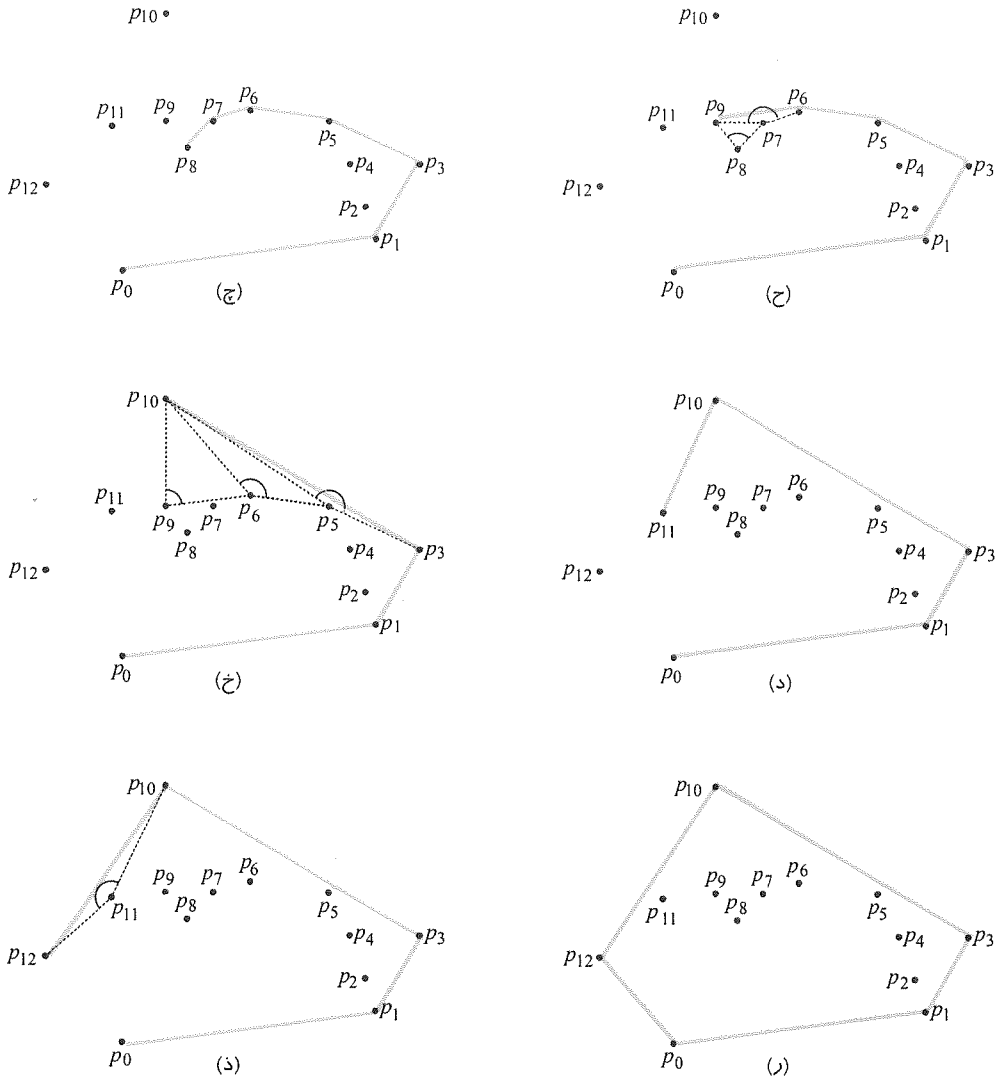
```
4   PUSH( p_0 , S)
5   PUSH( p_1 , S)
6   PUSH( p_2 , S)
7   for i = 3 to m
8       while the angle formed by points NEXT-TO-TOP(S), TOP(S),
                and p_i  makes a nonleft turn
9           POP(S)
10      PUSH( p_i , S)
11  return S
```

شکل ۳۳-۷ روند GRAHAM-SCAN را نشان می‌دهد. خط ۱ نقطه‌ی p_0 را به عنوان ورودی با کم‌ترین مقدار y انتخاب می‌کند، که در میان چنین نقاطی، سمت چپ‌ترین هم هست. از آن جایی که هیچ نقطه‌ی دیگری پایین‌تر از p_0 در Q وجود ندارد، و تمام نقاطی که مختصات y آن‌ها با p_0 برابر است، در سمت راست آن قرار دارند، p_0 رأسی از $CH(Q)$ است. خط ۲ بقیه‌ی نقاط Q را بر حسب زاویه‌ی قطبی نسبت به p_0 مرتب می‌کند، با استفاده از متد استفاده شده در تمرین ۳۳-۱-۳، یعنی مقایسه‌ی ضرب خارجی. اگر دو یا چند نقطه زاویه‌ی قطبی یکسانی نسبت به p_0 داشته باشند، تمام چنین نقاطی، غیر از دورترین نقطه، ترکیب‌های محدبی از p_0 و دورترین نقطه هستند، و بنابراین هیچ کدام از آن‌ها را در نظر نمی‌گیریم. فرض می‌کنیم m نشان‌دهنده‌ی تعداد نقاط باقی‌مانده غیر از p_0 باشد. زاویه‌ی قطبی هر نقطه در Q نسبت به p_0، بر حسب رادیان، در بازه‌ی نیمه‌باز $[0, x)$ است. از آن جایی که نقاط بر حسب زاویه‌ی قطبی مرتب شده‌اند، پس ترتیب آن‌ها نسبت به p_0 در جهت عکس حرکت عقربه‌های ساعت است. این دنباله‌ی مرتب شده از نقاط را با $\langle p_1, p_2, ..., p_m \rangle$ نشان می‌دهیم. توجه کنید که نقاط p_1 و p_m رأس‌هایی از $CH(Q)$ هستند (تمرین ۳۳-۳-۱ را ببینید). شکل ۳۳-۷ (الف) نقاط شکل ۳۳-۶ را نشان می‌دهد که به ترتیب افزایشی زاویه‌ی قطبی نسبت به p_0 شماره‌گذاری شده‌اند.

بقیه‌ی رویه از پشته‌ی S استفاده می‌کند. خطوط ۳-۶ پشته را مقداردهی اولیه می‌کنند، تا از پایین به بالا حاوی سه نقطه‌ی اول p_0، p_1، و p_2 باشد. شکل ۳۳-۷ (الف) پشته‌ی اولیه‌ی S را نشان می‌دهد. حلقه‌ی for در خطوط ۷-۱۰ برای هر یک از نقاط زیردنباله‌ی $\langle p_3, p_4, ..., p_m \rangle$ یک بار تکرار می‌شود. خواهیم دید که پس از پردازش p_i، پشته‌ی S از پایین به بالا حاوی رأس‌های $CH(\{p_0, p_1, ..., p_i\})$ باشد، به ترتیب عکس حرکت عقربه‌های ساعت. در صورتی که نقطه‌ای متعلق به پوسته‌ی محدب نباشد، حلقه‌ی while خطوط ۸-۹ آن را از پشته حذف می‌کند. وقتی پوسته‌ی محدب را در جهت عکس حرکت عقربه‌های ساعت پیمایش می‌کنیم، باید بر روی هر رأس یک چرخش چپ انجام دهیم. بنابراین، هر گاه که حلقه‌ی while رأسی میابد که در آن چرخش غیر چپ انجام می‌شود، آن رأس را از پشته حذف می‌کند. (با چک کردن برای یک چرخش غیر چپ، به جای یک چرخش راست، احتمال وجود یک زاویه‌ی ۱۸۰ درجه را از بین می‌بریم. این کار ضروری است،



شکل ۳۳-۷ اجرای GRAHAM-SCAN بر روی مجموعه‌ی Q از شکل ۳۳-۶. پوسته‌ی محدب فعلی ذخیره شده در پشته‌ی S در هر مرحله با رنگ روشن نشان داده شده است. (الف) دنباله‌ی $\langle p_1, p_2, \ldots, p_{12} \rangle$ از نقاط که به ترتیب افزایشی زاویه‌ی قطبی نسبت به p_0 شماره‌گذاری شده‌اند، و پشته‌ی اولیه‌ی S حاوی نقاط p_0، p_1، و p_2 است. (ب)-(ذ) پشته‌ی S بعد از هر اجرای حلقه‌ی for در خطوط ۶-۹. خط‌چین‌ها، چرخش‌های غیر چپ را نشان می‌دهند، که باعث می‌شوند نقاط از پشته حذف شوند. مثلاً در بخش (ح)، چرخش راست روی زاویه‌ی $\angle p_7 p_8 p_9$ باعث می‌شود که p_8 حذف شود، و سپس چرخش راست روی زاویه‌ی $\angle p_6 p_7 p_9$ نقطه‌ی p_7 را از پشته حذف می‌کند. (ر) پوسته‌ی محدب بازگردانده شده توسط رویه، که با پوسته‌ی شکل ۳۳-۶ مطابقت دارد.



شکل ۳۳-۷ ادامه

چرا که هیچ رأسی از یک چندضلعی محدب نمی‌تواند ترکیب محدبی از رأس‌های دیگر چندضلعی باشد.) وقتی تمام رأس‌هایی را که هنگام حرکت به سمت p_i یک چرخش غیر چپ انجام می‌دهند، حذف کردیم، p_i را در پشته درج می‌کنیم. شکل ۳۳-۷ (ب)-(ذ) وضعیت پشته‌ی S را بعد از هر تکرار حلقه‌ی for نشان می‌دهد. در نهایت، GRAHAM-SCAN پشته‌ی S را در خط ۱۱ بازمی‌گرداند. شکل ۳۳-۷ (ر) پوسته‌ی محدب مربوطه را نشان می‌دهد.

قضیه‌ی زیر به صورت رسمی درستی GRAHAM-SCAN را اثبات می‌کند.

قضیه‌ی ۳۳-۱ (درستی پویش گراهام) اگر پویش گراهام بر روی مجموعه‌ی Q از نقاط اجرا شود، که در آن $|Q| \geq ۳$، هنگام پایان، پشته‌ی S از پایین به بالا دقیقاً حاوی رأس‌های $CH(Q)$ به ترتیب عکس حرکت عقربه‌های ساعت است.

اثبات بعد از خط ۲، دنباله‌ی نقاط $\langle p_۰, p_۱, \ldots, p_m \rangle$ را داریم. برای $i = ۲, ۳, \ldots, m$ زیرمجموعه‌ی نقاط $Q_i = \{p_۰, p_۱, \ldots, p_i\}$ را تعریف می‌کنیم. نقاط در $Q - Q_m$ نقاطی هستند که چون زاویه‌ی قطبی آن‌ها نسبت به $p_۰$ با نقطه‌ای دیگر یکی بوده، حذف شده‌اند؛ این نقاط در $CH(Q)$ نیستند، و بنابراین $CH(Q_m) = CH(Q)$. بنابراین، کافی است نشان دهیم که وقتی GRAHMA-SCAN تمام می‌شود، پشته‌ی S از پایین به بالا دقیقاً حاوی رأس‌های $CH(Q_m)$ به ترتیب عکس حرکت عقربه‌های ساعت است. توجه کنید که همان طور که رأس‌های $p_۰$، $p_۱$، و p_m رأس‌های $CH(Q)$ هستند، نقطه‌های $p_۰$، $p_۱$، و p_i همگی رأس‌های $CH(Q_i)$ هستند.

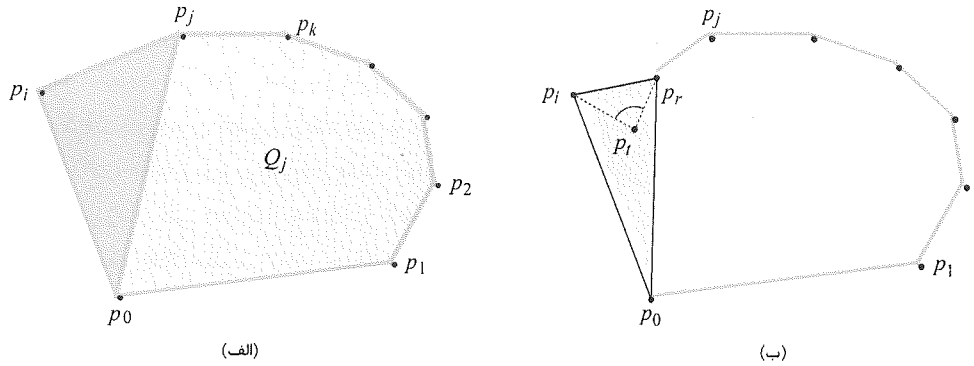
در اثبات از ثابت حلقه‌ی زیر استفاده می‌شود:

در آغاز هر تکرار حلقه‌ی for در خطوط ۷-۱۰، پشته‌ی S از پایین به بالا دقیقاً حاوی رأس‌های $CH(Q_{i-۱})$ به ترتیب عکس حرکت عقربه‌های ساعت است.

- آغاز: ثابت حلقه در اولین اجرای خط ۷ برقرار است، و مجموعه‌ی این سه نقطه پوسته‌ی محدب خود را تشکیل می‌دهند. به علاوه، این نقاط از پایین به بالا به ترتیب عکس حرکت عقربه‌های ساعت قرار دارند.
- ادامه: با ورود به یک تکرار از حلقه‌ی for نقطه‌ی بالایی در پشته‌ی S، $p_{i-۱}$ است، که در پایان تکرار قبلی در پشته درج شده است (یا قبل از تکرار اول، وقتی $i = ۳$). فرض کنید، دقیقاً بعد از اجرای حلقه‌ی while خطوط ۸-۹ و قبل از این که خط ۱۰ نقطه‌ی p_i را درج کند، p_j نقطه‌ی بالایی در پشته‌ی S باشد، و p_k نقطه‌ی زیری p_j در S. در لحظه‌ای که p_j نقطه‌ی بالایی S است، و هنوز p_i را درج نکرده‌ایم، پشته‌ی S دقیقاً حاوی همان نقاطی است که بعد از تکرار jام از حلقه‌ی for در آن قرار داشته‌اند. بنابراین طبق ثابت حلقه، در آن لحظه S دقیقاً حاوی رأس‌های $CH(Q_j)$ است، و آن‌ها از پایین به بالا به ترتیب عکس حرکت عقربه‌های ساعت قرار دارند.

اجازه دهید همچنان بر روی لحظه‌ی قبل از درج p_i تمرکز کنیم. با رجوع به شکل ۳۳-۸ (الف)، چون زاویه‌ی قطبی p_i نسبت به $p_۰$ بزرگ‌تر از زاویه‌ی قطبی p_j است، و چون زاویه‌ی $\angle p_k p_j p_i$ یک چرخش چپ انجام می‌دهد (در غیر این صورت باید p_j را از پشته بازیابی می‌کردیم)، می‌بینیم که چون S دقیقاً حاوی رأس‌های $CH(Q_j)$ است، وقتی p_i را درج می‌کنیم، پشته‌ی S دقیقاً حاوی رأس‌های $CH(Q_j \cup \{p_i\})$ خواهد بود، همچنان به ترتیب عکس حرکت عقربه‌های ساعت.

اکنون نشان می‌دهیم که $CH(Q_j \cup \{p_i\})$ معادل است با مجموعه‌ی نقاط $CH(Q_i)$. یک نقطه‌ی دلخواه p_t را در نظر بگیرید که حین تکرار iام حلقه‌ی for بازیابی شده است، و فرض



شکل ۳۳-۸ اثبات درستی GRAHAM-SCAN. (الف) چون زاویه‌ی قطبی p_i نسبت به p_0 بزرگ‌تر از زاویه‌ی قطبی p_j است، و چون زاویه‌ی $\angle p_k p_j p_i$ یک چرخش چپ انجام می‌دهد، اضافه کردن p_i به $CH(Q_j)$ دقیقاً رأس‌های $CH(Q_j \cup \{p_i\})$ را می‌دهد. (ب) اگر زاویه‌ی $\angle p_r p_t p_i$ یک چرخش غیر چپ انجام دهد، آن گاه p_t یا در نقاط داخلی مثلث تشکیل شده توسط p_0، p_r، و p_i است، و یا بر روی ضلع همان مثلث، و نمی‌تواند رأسی از $CH(Q_i)$ باشد.

کنید p_r نقطه‌ی زیری p_t در پشته‌ی S باشد، وقتی که p_t بازیابی شده است (p_r می‌تواند p_j باشد). زاویه‌ی $\angle p_r p_t p_i$ یک چرخش غیرچپ انجام می‌دهد، و زاویه‌ی قطبی p_t نسبت به p_0 بزرگ‌تر از زاویه‌ی p_r است. همان طور که شکل ۳۳-۸ (ب) نشان می‌دهد، p_t یا باید در فضای داخلی مثلثی باشد که نقاط p_0، p_r، و p_i تشکیل می‌دهند، و یا باید بر روی یک ضلع آن باشد (ولی نمی‌تواند رأس آن باشد). مسلماً چون p_t درون مثلث تشکیل شده توسط سه نقطه‌ی دیگر از Q_i است، نمی‌تواند رأسی از $CH(Q_i)$ باشد. چون p_t رأسی از $CH(Q_i)$ نیست، داریم

$$CH(Q_i - \{p_t\}) = CH(Q_i) \tag{۱-۳۳}$$

فرض کنید P_i مجموعه‌ی نقاطی باشد که هنگام تکرار iام حلقه‌ی for بازیابی شده است. چون تساوی (۳۳-۱) برای تمام نقاط P_i صحت دارد، می‌توانیم مکررا از آن استفاده کرده و نشان دهیم که $CH(Q_i - P_i) = CH(Q_i)$. ولی $Q_i - P_i = Q_j \cup \{p_i\}$، و بنابراین نتیجه می‌گیریم که $CH(Q_j \cup \{p_i\}) = CH(Q_i - P_i) = CH(Q_i)$.

نشان دادیم که وقتی p_i را درج می‌کنیم، پشته‌ی S از پایین به بالا دقیقاً حاوی رأس‌های $CH(Q_i)$ به ترتیب عکس حرکت عقربه‌های ساعت است. بنابراین افزایش i باعث می‌شود که ثابت حلقه همچنان برقرار باقی بماند.

• **پایان:** وقتی حلقه پایان می‌یابد، داریم $i = m + 1$، و بنابراین ثابت حلقه ایجاب می‌کند که پشته‌ی S دقیقاً حاوی رأس‌های $CH(Q_m)$ باشد، که همان $CH(Q)$ است، از پایین به بالا با ترتیب عکس حرکت عقربه‌های ساعت. این اثبات را کامل می‌کند. ■

اکنون نشان می‌دهیم که زمان اجرای GRAHAM-SCAN برابر است با $O(n \lg n)$، که در آن $n = |Q|$. خط ۱ به $\theta(n)$ زمان نیاز دارد. اگر از مرتب‌سازی ادغامی و یا مرتب‌سازی هرمی برای مرتب کردن زاویه‌های قطبی، و متد ضرب خارجی بخش ۳۳-۱ برای مقایسه‌ی زاویه‌ها استفاده کنیم، خط ۲، $O(n \lg n)$ زمان می‌گیرد. (حذف تمام نقاط با زاویه‌ی قطبی برابر غیر از دورترین آن‌ها را می‌توان در زمان کلی $O(n)$ انجام داد.) خطوط ۳-۶ به $O(1)$ زمان نیاز دارند. داریم $m \leq n - 1$، و از این رو حلقه‌ی for خطوط ۷-۱۰ حداکثر $n - 3$ بار تکرار می‌شود. چون PUSH از مرتبه‌ی $O(1)$ است، هر تکرار، غیر از زمان صرف شده در حلقه‌ی while خطوط ۸-۹ به $O(1)$ زمان نیاز دارد، و بنابراین کل حلقه‌ی for (صرف نظر از زمان اجرای حلقه‌ی while) در زمان $O(n)$ اجرا می‌شود.

از تحلیل متراکم استفاده می‌کنیم تا نشان دهیم که حلقه‌ی while در کل به $O(n)$ زمان نیاز دارد. برای $i = 0, 1, \ldots, m$، هر نقطه‌ی p_i دقیقاً یک بار در پشته‌ی S درج می‌شود. مانند تحلیل رویه‌ی MULTIPOP در بخش ۱۷-۱، مشاهده می‌کنیم که حداکثر یک عملیات POP برای هر عملیات PUSH وجود دارد. حداقل سه نقطه — p_0، p_1، و p_m — هیچ وقت از پشته حذف نمی‌شوند، و بنابراین، در حقیقت حداکثر $m - 2$ عملیات POP خواهیم داشت. هر تکرار حلقه‌ی while یک عملیات POP انجام می‌دهد، و بنابراین، این حلقه حداکثر $m - 2$ تکرار خواهد داشت. چون تست خط ۸ به $O(1)$ زمان نیاز دارد، هر فراخوانی POP از مرتبه‌ی $O(1)$ است، و چون $m \leq n - 1$، کل زمان صرف شده توسط حلقه‌ی while برابر است با $O(n)$. بنابراین زمان اجرای GRAHAM-SCAN از مرتبه‌ی $O(n \lg n)$ است.

راهپیمایی Jarvis

راهپیمایی Jarvis (Jarvis's march) یا *بسته‌بندی هدیه* (gift wrapping یا package wrapping) پوسته‌ی محدب مجموعه‌ی Q از نقاط را با استفاده از تکنیکی به نام محاسبه می‌کند. الگوریتم در زمان $O(nh)$ اجرا می‌شود، که h تعداد رأس‌های پوسته‌ی محدب است. اگر h از مرتبه‌ی $o(\lg n)$ باشد، راهپیمایی Jarvis به صورت حدی سریع‌تر از پویش Graham است.

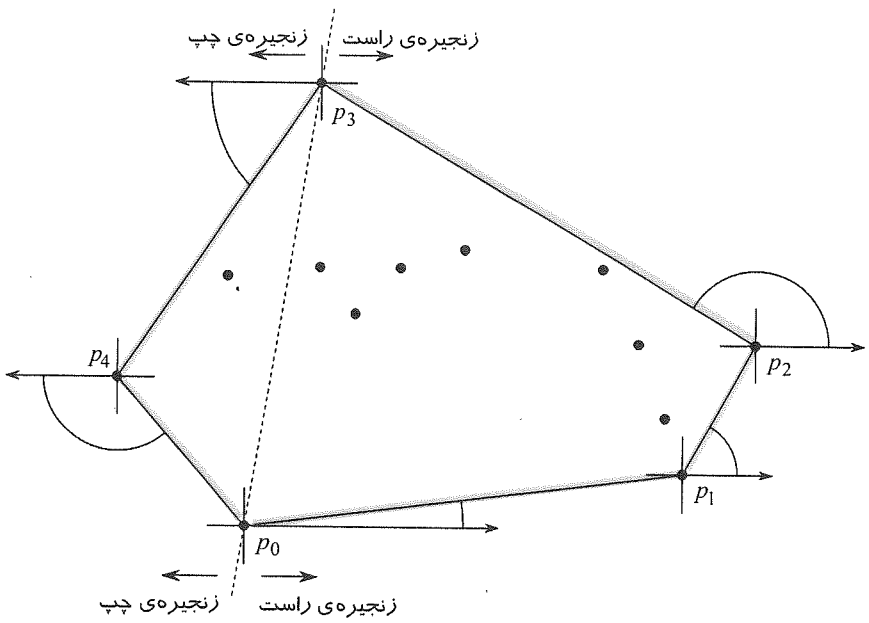
به صورت شهودی، راهپیمایی Jarvis بستن یک قطعه کاغذ محکم را به دور مجموعه‌ی Q شبیه‌سازی می‌کند. با گره زدن انتهای کاغذ به پایین‌ترین نقطه‌ی مجموعه آغاز می‌کنیم، یعنی همان نقطه‌ی p_0 که پویش Graham را با آن آغاز کردیم. این نقطه، یک نقطه از پوسته‌ی محدب است. کاغذ را به سمت راست می‌کشیم تا محکم شود، و سپس آن را به سمت بالا حرکت می‌دهیم تا یک نقطه را لمس کند. این نقطه هم باید نقطه‌ای از پوسته‌ی محدب باشد. با محکم نگه داشتن کاغذ، به همین مسیر دور نقاط ادامه می‌دهیم تا به نقطه‌ی اولیه‌ی p_0 برسیم.

به صورت رسمی‌تر، راهپیمایی Jarvis یک دنباله‌ی $H = \langle p_0, p_1, \ldots, p_{h-1} \rangle$ از رأس‌های $CH(Q)$ می‌سازد. با نقطه‌ی p_0 آغاز می‌کنیم. همان طور که شکل ۳۳-۹ نشان می‌دهد، رأس بعدی پوسته‌ی محدب، یعنی p_1، کوچک‌ترین زاویه‌ی قطبی را نسبت به p_0 دارد. (در حالت‌های مساوی، دورترین نقطه از p_0 را انتخاب می‌کنیم.) به طور مشابه، p_2 کوچک‌ترین زاویه‌ی قطبی نسبت به p_1 را دارد، و به

همین ترتیب تا آخر. وقتی به بالاترین رأس، مثلاً p_k می‌رسیم (که در صورت مساوی بودن ارتفاع چند رأس، دورترین آن‌ها را به عنوان p_k انتخاب می‌کنیم)، همان طور که شکل ۳۳-۹ نشان می‌دهد، *زنجیر راست* (right chain) $CH(Q)$ را ساخته‌ایم. برای ساختن *زنجیر چپ* (left chain)، از p_k آغاز کرده و p_{k+1} را به صورت نقطه‌ی با کم‌ترین زاویه‌ی قطبی نسبت به p_k انتخاب می‌کنیم، ولی در *جهت منفی محور* x. به همین شکل ادامه می‌دهیم، تا در نهایت به رأس اولیه‌ی p_0 برسیم.

راهپیمایی Jarvis را می‌توانیم به کمک مفهوم جارو دور پوسته‌ی محدب پیاده‌سازی کنیم، یعنی بدون ساختن جداگانه‌ی زنجیرهای سمت چپ و راست. معمولاً چنین پیاده‌سازی‌هایی اطلاعات زاویه‌ی آخرین ضلع پوسته‌ی محدب را نگه می‌دارند، و نیاز دارند که زاویه‌ی اضلاع پوسته‌ی محدب به صورت صعودی اکید باشد (در دامنه‌ی ۰ و 2π). مزیت ساختن زنجیرهای جداگانه این است که نیازی نیست که صریحاً زاویه‌ها را محاسبه کنیم؛ تکنیک‌های بخش ۳۳-۱ برای مقایسه‌ی زاویه‌ها کافی هستند.

در صورت پیاده‌سازی بهینه، زمان اجرای راهپیمایی Jarvis برابر است با $O(nh)$. برای هر یک از h رأس $CH(Q)$، رأس با زاویه‌ی قطبی کمینه را پیدا می‌کنیم. با استفاده از تکنیک‌های بخش ۳۳-۱، هر مقایسه بین زاویه‌های قطبی $O(1)$ زمان می‌گیرد. همان طور که بخش ۹-۱ نشان می‌دهد، اگر هر



شکل ۳۳-۹ عملیات راهپیمایی Jarvis اولین رأس انتخاب شده، پایین‌ترین نقطه، یعنی p_0 است. رأس بعدی، p_1، در میان نقاط مجموعه، کم‌ترین زاویه‌ی قطبی را نسبت به p_0 دارد. سپس، p_2 کم‌ترین زاویه‌ی قطبی را نسبت به p_1 دارد. زنجیره‌ی سمت راست تا بالاترین نقطه، یعنی p_3 ادامه می‌یابد. سپس زنجیره‌ی سمت چپ با یافتن کوچک‌ترین زاویه‌ی قطبی نسبت به محور منفی x ساخته می‌شود.

مقایسه در زمان $O(1)$ انجام شود، می‌توانیم کمینه‌ی n مقدار را در زمان $O(n)$ بیابیم. بنابراین راهپیمایی Jarvis به $O(nh)$ زمان نیاز دارد.

تمرین‌ها

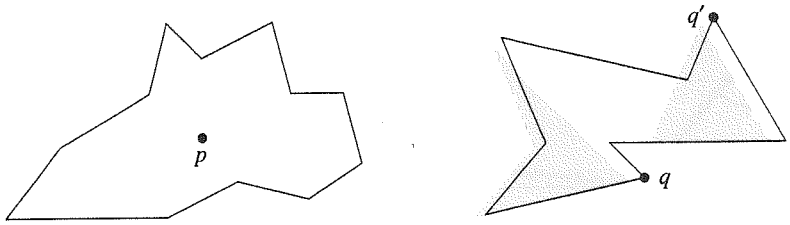
۳۳-۳-۱ اثبات کنید که در رویه‌ی GRAHAM-SCAN، نقاط p_1 و p_m باید رأس‌هایی از $CH(Q)$ باشند.

۳۳-۳-۲ مدلی از محاسبات را در نظر بگیرید که از جمع، مقایسه، و ضرب پشتیبانی می‌کند، و یک کران پایین $\Omega(n \lg n)$ برای مرتب‌سازی n عدد برای آن وجود دارد. اثبات کنید که برای چنین مدلی، $\Omega(n \lg n)$ یک کران پایین برای محاسبه‌ی (به ترتیب) رأس‌های پوسته‌ی محدب یک مجموعه از n نقطه است.

۳۳-۳-۳ با داشتن یک مجموعه‌ی Q از نقاط، اثبات کنید که جفت نقاطی که دورترین فاصله را از یکدیگر دارند، باید رأس‌هایی از $CH(Q)$ باشند.

۳۳-۳-۴ برای یک چندضلعی P و یک نقطه‌ی q بر روی مرز آن، *سایه‌ی* q عبارت است از مجموعه‌ی نقاط r به طوری که پاره‌خط $\overline{qr}$ کاملاً در نقاط داخلی یا نقاط مرزی P قرار دارد.

یک چندضلعی P *ستاره‌شکل* (star-shaped) است اگر یک نقطه‌ی p در نقاط داخلی P وجود داشته باشد که سایه‌ی تمام نقاط مرزی P باشد. به مجموعه‌ی تمام نقاط p مانند این، *هسته‌ی* (kernel) P گفته می‌شود. (شکل ۳۳-۱۰ را ببینید.) با داشتن یک n ضلعی ستاره‌شکل P که توسط رأس‌های آن به ترتیب عکس حرکت عقربه‌های ساعت توصیف شده است، نشان دهید که چطور می‌توان $CH(P)$ را در زمان $O(n)$ محاسبه کرد.



شکل ۳۳-۱۰ تعریف یک چندضلعی ستاره‌شکل، برای استفاده در تمرین ۳۳-۳-۴. (الف) یک چندضلعی ستاره‌شکل. هر پاره‌خط از نقطه‌ی p به هر نقطه‌ی مرزی q، نقاط مرزی را فقط در q قطع می‌کند. (ب) یک چندضلعی غیر ستاره‌شکل. ناحیه‌ی سایه‌دار در سمت چپ سایه‌ی q است، و ناحیه‌ی سایه‌دار در سمت راست سایه‌ی q'. چون این دو ناحیه با یکدیگر اشتراک ندارند، هسته‌ی این چندضلعی تهی است.

۵-۳-۳۳ در *مسئله‌ی پوسته‌ی محدب آنلاین* (on-line convex-hull problem)، مجموعه‌ی Q از n نقطه، به صورت یک نقطه در هر لحظه به ما داده می‌شود. پس از دریافت هر نقطه، باید پوسته‌ی محدب نقاطی را که تا این جا داریم محاسبه کنیم. بدیهتاً می‌توانیم پویش Graham را یک بار برای هر نقطه اجرا کنیم، با زمان اجرای کلی $O(n^2 \lg n)$. نشان دهید که چگونه می‌توان مسئله‌ی پوسته‌ی محدب آنلاین را در زمان $O(n^2)$ حل کرد.

★ ۶-۳-۳۳ نشان دهید که چطور می‌توان متد افزایشی را برای محاسبه‌ی پوسته‌ی محدب n نقطه پیاده‌سازی کرد که در زمان $O(n \lg n)$ اجرا شود.

۴-۳۳ یافتن نزدیک‌ترین جفت نقاط

اکنون مسئله‌ی یافتن نزدیک‌ترین جفت نقاط در میان یک مجموعه‌ی Q از $n \geq 2$ نقطه را در نظر می‌گیریم. «نزدیک‌ترین» به همان فاصله‌ی اقلیدسی معروف اشاره دارد: فاصله‌ی نقطه‌ی $p_1 = (x_1, y_1)$ و $p_2 = (x_2, y_2)$ برابر است با $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. ممکن است دو نقطه در یک مجموعه مختصات یکسانی داشته باشند، که در این صورت فاصله‌ی آن‌ها برابر است با صفر. این مسئله کاربردهایی، مثلاً در سیستم‌های کنترل ترافیک، دارد. یک سیستم برای کنترل ترافیک هوایی یا دریایی ممکن است نیاز داشته باشد بداند که کدام دو وسیله‌ی نقلیه نزدیک‌ترین هستند، تا بتواند تصادم‌های احتمالی را تشخیص دهد.

یک الگوریتم ساده‌لوحانه نزدیک‌ترین جفت به سادگی تمام $\binom{n}{2} = \theta(n^2)$ جفت ممکن را بررسی می‌کند. در این بخش، یک الگوریتم تقسیم و حل برای این مسئله ارائه می‌کنیم که زمان اجرای آن توسط رابطه‌ی بازگشتی آشنای $T(n) = 2T(n/2) + O(n)$ توصیف می‌شود. بنابراین، این الگوریتم فقط به $O(n \lg n)$ زمان نیاز دارد.

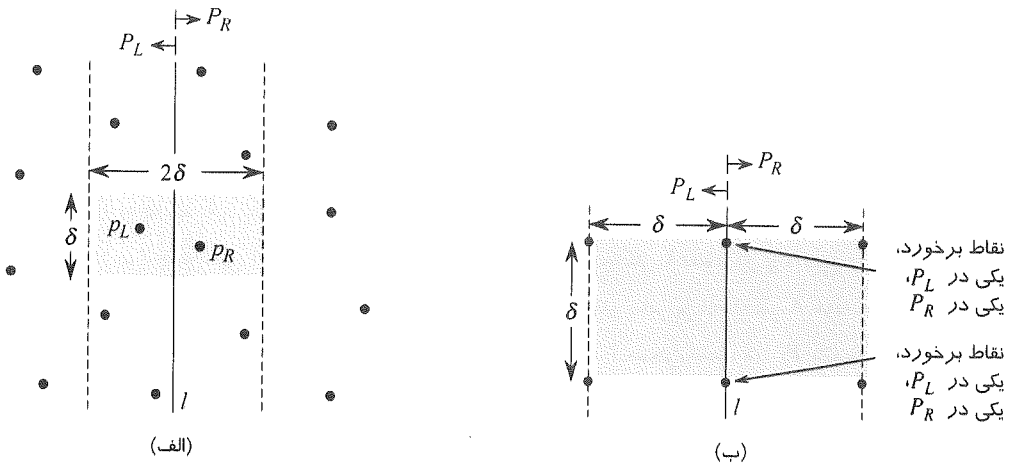
الگوریتم تقسیم و حل

هر فراخوانی بازگشتی الگوریتم، یک زیرمجموعه‌ی $P \subseteq Q$ و آرایه‌های X و Y را که هر یک حاوی تمام نقاط زیرمجموعه‌ی P هستند را به عنوان ورودی دریافت می‌کند. نقاط آرایه‌ی X به ترتیب صعودی مختصات x مرتب شده‌اند، و نقاط آرایه‌ی Y به ترتیب صعودی مختصات y. توجه کنید که برای دست‌یابی به کران زمانی $O(n \lg n)$، نمی‌توانیم هزینه‌ی مرتب‌سازی در هر یک از فراخوانی‌های بازگشتی را تحمل کنیم؛ اگر چنین کاری بکنیم، رابطه‌ی بازگشتی زمان اجرا $T(n) = 2T(n/2) + O(n \lg n)$ خواهد بود، که جواب آن $T(n) = O(n \lg^2 n)$ است. (از نسخه‌ای از متد اصلی که در تمرین ۴-۶-۲ داده شده است، استفاده کنید.) کمی بعد خواهیم دید که چگونه می‌توان از «پیش‌مرتب‌سازی» برای حفظ ترتیب عناصر استفاده کرد، بدون این که در هر فراخوانی بازگشتی یک مرتب‌سازی انجام دهیم.

یک فراخوانی بازگشتی با ورودی‌های P، X، و Y ابتدا رابطه‌ی $|P| \leq ۳$ را چک می‌کند. اگر چنین بود، به سادگی، متد ساده‌لوحانه‌ی توصیف شده در بالا اجرا می‌شود: چک کردن تمام $\binom{|P|}{۲}$ جفت ممکن و بازگرداندن نزدیک‌ترین جفت نقاط. اگر $|P| > ۳$، فراخوانی بازگشتی الگوی تقسیم و حل زیر را پیاده می‌کند.

- تقسیم: یافتن خط عمودی l که نقاط P را به دو مجموعه‌ی P_L و P_R تقسیم می‌کند، به طوری که $|P_L| = \lceil |P|/۲ \rceil$ و $|P_R| = \lfloor |P|/۲ \rfloor$، تمام نقاط P_L بر روی خط l یا در سمت چپ آن قرار دارد، و تمام نقاط P_R بر روی خط l و یا در سمت راست آن قرار دارند. آرایه‌ی X به آرایه‌های X_L و X_R تقسیم می‌شود، که به ترتیب حاوی نقاط P_L و P_R هستند، به ترتیب صعودی مختصات x. به طور مشابه، آرایه‌ی Y به آرایه‌های Y_L و Y_R تقسیم می‌شود، که به ترتیب حاوی نقاط P_L و P_R هستند، به ترتیب صعودی مختصات y.
- حل: پس از تقسیم P به P_L و P_R، دو فراخوانی بازگشتی انجام می‌شود، یکی برای یافتن نزدیک‌ترین جفت نقاط در P_L و دیگری برای یافتن نزدیک‌ترین جفت نقاط در P_R. ورودی‌های فراخوانی اول عبارتند از زیرمجموعه‌ی P_L و آرایه‌های X_L و Y_L؛ فراخوانی دوم ورودی‌های P_R، X_R، و Y_R را دریافت می‌کند. فرض کنید فاصله‌ی نزدیک‌ترین جفت بازگردانده شده برای P_L و P_R به ترتیب δ_L و δ_R باشد، و $\delta = \min(\delta_L, \delta_R)$.
- ترکیب: نزدیک‌ترین جفت یا جفت با فاصله‌ی δ است که توسط یکی از فراخوانی‌های بازگشتی پیدا شده است، و یا جفتی است که یک نقطه از آن در P_L قرار دارد، و دیگری در P_R. الگوریتم تعیین می‌کند که آیا چنین جفتی وجود دارد که فاصله‌ی نقاط آن کم‌تر از δ باشد یا خیر. مشاهده کنید که اگر جفتی از نقاط وجود داشته باشد که فاصله‌ی نقاط آن کم‌تر از δ باشد، هر دوی این نقاط باید در فاصله‌ای کم‌تر از δ نسبت به خط l قرار داشته باشند. بنابراین، همان طور که شکل ۳۳-۱۱ (الف) نشان می‌دهد، هر دوی این نقاط باید در نوار با عرض ۲δ و مرکز l قرار داشته باشند. برای یافتن چنین جفتی، در صورت وجود، الگوریتم عملیات زیر را انجام می‌دهد.

۱. ابتدا یک آرایه‌ی Y' می‌سازد، که همان آرایه‌ی Y است که تمام نقاط خارج از نوار عمودی با عرض ۲δ از آن حذف شده‌اند. آرایه‌ی Y' بر حسب مختصات y مرتب شده است، درست مانند آرایه‌ی Y.
۲. برای هر نقطه‌ی p در آرایه‌ی Y'، الگوریتم سعی می‌کند نقاطی در Y' بیابد که فاصله‌ی آن‌ها کم‌تر از δ واحد از p است. همان طور که به زودی خواهیم دید، تنها ۷ نقطه در Y' که بعد از p قرار دارند باید چک شوند. الگوریتم، فاصله‌ی p از هر یک از این ۷ نقطه را محاسبه می‌کند، و اطلاعات نزدیک‌ترین فاصله‌ی δ' یافته شده در تمام جفت نقاط در Y' را نگه می‌دارد.
۳. اگر $\delta < \delta'$، در این صورت نوار عمودی حاوی یک جفت است که فاصله‌ی آن‌ها از جفت یافته شده توسط فراخوانی‌های بازگشتی کم‌تر است. این جفت و فاصله‌ی δ' بازگردانده



شکل ۳۳-۱۱ مفاهیم کلیدی در اثبات این که الگوریتم نزدیک‌ترین جفت، برای هر نقطه در Y' فقط نیاز دارد ۷ نقطه‌ی بعد از آن را چک کند. (الف) اگر فاصله‌ی $p_L \in P_L$ و $p_R \in P_R$ کم‌تر از δ واحد باشد، این دو باید درون یک مستطیل $\delta \times 2\delta$ با مرکز l باشند. (ب) نحوه‌ی قرار گرفتن ۴ نقطه که فاصله‌ی هر جفت از آن‌ها حداقل δ است، در یک مربع $\delta \times \delta$. در سمت چپ ۴ نقطه در P_L قرار دارند، و در سمت راست، ۴ نقطه در P_R. اگر نقاط نشان داده شده بر روی خط l پر واقع جفتی از نقاط با مختصات یکسان باشند که یکی در P_L قرار دارد و دیگری در P_R می‌توان در یک مستطیل $\delta \times 2\delta$ تعداد ۸ نقطه قرار داد.

می‌شود. در غیر این صورت، نزدیک‌ترین جفت و فاصله‌ی δ یافته شده توسط فراخوانی‌های بازگشتی بازگردانده می‌شود.

توصیف بالا از بعضی جزئیات پیاده‌سازی که برای دست‌یابی به زمان اجرای $O(n \lg n)$ ضروری هستند، صرف نظر می‌کند. پس از اثبات درستی الگوریتم، نشان خواهیم داد که چگونه می‌توان الگوریتم را طوری پیاده‌سازی کرد که به کران زمانی داده شده دست یابیم.

درستی

درستی این الگوریتم نزدیک‌ترین جفت بدیهی است، غیر از دو جنبه‌ی آن. اول، با جمع‌بندی بازگشت وقتی که $|P| \leq 3$، اطمینان حاصل می‌کنیم که هیچ‌گاه سعی نخواهیم کرد که یک زیرمسئله شامل فقط یک نقطه را حل کنیم. جنبه‌ی دوم این است که نیاز داریم فقط ۷ نقطه بعد از هر نقطه‌ی p در Y' را بررسی کنیم؛ در این جا این خصوصیت را اثبات می‌کنیم.

فرض کنید که در مرحله‌ای از بازگشت، نزدیک‌ترین جفت نقاط عبارت است از $p_L \in P_L$ و $p_R \in P_R$. بنابراین، فاصله‌ی δ' بین p_L و p_R اکیداً کم‌تر از δ است. نقطه‌ی p_L باید روی خط l و یا در سمت چپ آن باشد، با فاصله‌ی کم‌تر از δ. به طور مشابه، p_R باید روی خط l و یا در سمت راست آن باشد، با فاصله‌ی کم‌تر از δ. به علاوه، فاصله‌ی عمودی p_L و p_R کم‌تر از δ واحد است.

بنابراین، همان طور که شکل ۳۳-۱۱ (الف) نشان می‌دهد، p_L و p_R درون یک مستطیل $\delta \times 2\delta$ با مرکز l هستند. (ممکن است نقاط دیگری هم در این مستطیل باشند.)

سپس نشان می‌دهیم که حداکثر ۸ نقطه از P می‌توانند در این مستطیل $\delta \times 2\delta$ باشند. مربع $\delta \times \delta$ را در نظر بگیرید که سمت چپ این مستطیل را تشکیل می‌دهد. از آن جایی که تمام نقاط درون P_L حداقل δ واحد با یکدیگر فاصله دارند، حداکثر ۴ تا از آن‌ها می‌توانند در این مربع باشند؛ شکل ۳۳-۱۱ (ب) چگونگی آن را نشان می‌دهد. به طور مشابه، حداکثر ۴ نقطه در P_R می‌توانند در مربع $\delta \times \delta$ تشکیل دهنده‌ی سمت راست مستطیل قرار داشته باشند. بنابراین، حداکثر ۸ نقطه از P می‌توانند در مستطیل $\delta \times 2\delta$ باشند. (توجه کنید که چون نقاط روی خط l ممکن است در P_L باشند و یا در P_R، حداکثر ۴ نقطه می‌تواند روی خط l باشد. در صورتی به این کران می‌رسیم که دو جفت نقطه با مختصات یکسان وجود داشته باشند، که هر جفت شامل یک نقطه از P_L و یک نقطه از P_R باشد، که یک جفت در مکان برخورد l با بالای مستطیل است، و دیگری در مکان برخورد l با خط پایین مستطیل.)

پس از نشان دادن این که حداکثر ۸ نقطه از P می‌توانند درون مستطیل گفته شده قرار گیرند، به سادگی می‌توانیم ببینیم که فقط باید ۷ نقطه‌ی بعد از هر نقطه در Y' را چک کنیم. با فرض این که نزدیک‌ترین جفت، نقاط p_L و p_R هستند. اجازه دهید بدون از دست دادن کلیت، فرض کنیم که p_L در آرایه‌ی Y' قبل از p_R است. در این صورت، حتی اگر p_L در پایین‌ترین مکان ممکن در Y' قرار داشته باشد، و p_R در بالاترین مکان ممکن در Y'، باز هم p_R در میان یکی از ۷ مکان بعد از p_L است. بنابراین، درستی الگوریتم نزدیک‌ترین جفت نقاط را اثبات کرده‌ایم.

پیاده‌سازی و زمان اجرا

همان طور که گفتیم، هدف ما این است که رابطه‌ی بازگشتی زمان اجرا $T(n) = 2T(n/2) + O(n)$ باشد، که در آن $T(n)$ زمان اجرای الگوریتم برای مجموعه‌ای از n نقطه است. سخت‌ترین بخش این است که اطمینان حاصل کنیم که آرایه‌های X_L، X_R، Y_L، و Y_R، که به فراخوانی‌های بازگشتی ارسال می‌شوند، برحسب مختصات مناسب مرتب‌سازی شده‌اند، و همچنین این که آرایه‌ی Y' بر حسب مختصات y مرتب شده است. (توجه کنید که اگر آرایه‌ی X که توسط فراخوانی بازگشتی دریافت می‌شود، از قبل مرتب باشد، آن گاه تقسیم P به P_L و P_R به سادگی در زمان خطی انجام می‌شود.)

نکته‌ی کلیدی این است که در هر فراخوانی، می‌خواهیم یک زیرمجموعه‌ی مرتب از یک آرایه‌ی مرتب داشته باشیم. برای مثال، به یک فراخوانی خاص زیرمجموعه‌ی P و آرایه‌ی Y (مرتب شده برحسب مختصات y)، داده شده است. پس از تقسیم P به P_L و P_R، الگوریتم نیاز دارد که آرایه‌های Y_L و Y_R را تشکیل دهد، که برحسب مختصات y مرتب شده‌اند. به علاوه این آرایه‌ها باید در زمان خطی تشکیل شوند. این متد را می‌توان به صورت معکوس رویه‌ی MERGE از مرتب‌سازی ادغامی در بخش ۲-۳-۱ دید: تقسیم یک آرایه‌ی مرتب شده به دو آرایه‌ی مرتب شده. رویه‌ی زیر این ایده را پیاده‌سازی می‌کند.

```
1  let Y_L[1 .. Y.length] and Y_R[1 .. Y.length] be new arrays
```

```
2  Y_L.length = Y_R.length = 0
3  for i = 1 to Y.length
4      if Y[i] ∈ P_L
5          Y_L.length = Y_L.length + 1
6          Y_L[Y_L.length] = Y[i]
7      else Y_R.length = Y_R.length + 1
8          Y_R[Y_R.length] = Y[i]
```

فقط کافی است نقاط آرایه‌ی Y را به ترتیب بررسی کنیم. اگر یک نقطه‌ی $Y[i]$ در P_L باشد، آن را به انتهای آرایه‌ی Y_L اضافه می‌کنیم؛ در غیر این صورت، آن را به انتهای آرایه‌ی Y_R اضافه می‌کنیم. به کمک رویه‌هایی مشابه، می‌توان X_L، X_R، و Y' را ساخت.

تنها سؤال باقی‌مانده این است که چگونه در ابتدای کار نقاط را مرتب کنیم. این کار را با *پیش‌مرتب‌سازی* (presorting) انجام می‌دهیم؛ فقط یک بار، *قبل* از اولین فراخوانی بازگشتی نقاط را مرتب می‌کنیم. این آرایه‌های مرتب شده به اولین فراخوانی بازگشتی ارسال می‌شوند، و پس از آن در فراخوانی‌های بازگشتی، در صورت نیاز آرایه‌ها تقسیم می‌شوند. پیش‌مرتب‌سازی به اندازه‌ی $O(n \lg n)$ به زمان اجرا اضافه می‌کند، ولی اکنون هر مرحله از بازگشت به زمان خطی نیاز دارد، غیر از زمان فراخوانی‌های بازگشتی. بنابراین، اگر $T(n)$ نشان دهنده‌ی زمان اجرای هر مرحله‌ی بازگشت باشد و $T'(n)$ زمان اجرای کل الگوریتم، خواهیم داشت $T'(n)=T(n)+O(n \lg n)$، و

$$T(n)=\begin{cases} ۲T(n/۲)+O(n) & \text{اگر } n>۳ \\ O(۱) & \text{اگر } n \le ۳ \end{cases}$$

بنابراین، $T(n)=O(n \lg n)$ و $T'(n)=O(n \lg n)$.

تمرین‌ها

۳۳-۴-۱ پروفسور Williams رویکردی در ذهن دارد که اجازه می‌دهد که الگوریتم نزدیک‌ترین جفت فقط ۵ نقطه‌ی بعد از هر نقطه در آرایه‌ی Y' را چک کند. ایده این است که همیشه نقاط روی خط l را در P_L قرار دهیم. در این صورت، دیگر نقاط با مختصات یکسان نمی‌توانند روی l باشند که یکی در P_L باشد و دیگری در P_R. بنابراین، حداکثر ۶ نقطه در مستطیل $\delta \times ۲\delta$ قرار خواهند داشت. نقص ایده‌ی پروفسور در کجاست؟

۳۳-۴-۲ نشان دهید که در واقع برای هر نقطه در آرایه‌ی Y' کافی است فقط ۵ نقطه‌ی بعد از آن را چک کنیم.

۳۳-۴-۳ فاصله‌ی بین دو نقطه را می‌توان با روش‌هایی غیر از روش اقلیدسی هم یافت. در صفحه، *فاصله‌ی* L_m بین نقاط $p_۱$ و $p_۲$ به صورت $(|x_۱-x_۲|^m+|y_۱-y_۲|^m)^{۱/m}$ داده می‌شود. بنابراین، فاصله‌ی اقلیدسی، فاصله $L_۲$ است. الگوریتم نزدیک‌ترین جفت را طوری اصلاح کنید که از فاصله‌ی $L_۱$ استفاده کند، که به *فاصله‌ی منهتن* (Manhattan distance) هم مشهور است.

۴-۴-۳۳ با داشتن دو نقطه‌ی p_1 و p_2 در صفحه، فرض کنید فاصله‌ی L_∞ بین آن‌ها به صورت $\max(|x_1 - x_2|, |y_1 - y_2|)$ تعریف شود. الگوریتم نزدیک‌ترین جفت را طوری اصلاح کنید که از فاصله‌ی L_∞ استفاده کند.

۵-۴-۳۳ فرض کنید که $\Omega(n)$ تا از نقاط داده شده به الگوریتم نزدیک‌ترین جفت، هم‌عمود (covertical) باشند. نشان دهید چطور می‌توان مجموعه‌های P_L و P_R را تشکیل داد، و تعیین کرد که هر نقطه‌ی Y در P_L است یا در P_R، به طوری که زمان اجرای الگوریتم نزدیک‌ترین جفت همچنان $O(n \lg n)$ باقی بماند.

۶-۴-۳۳ یک تغییر در الگوریتم نزدیک‌ترین جفت پیشنهاد کنید که از پیش‌مرتب‌سازی آرایه‌ی Y پرهیز می‌کند، ولی زمان اجرای $O(n \lg n)$ را تغییر نمی‌دهد.(*راهنمایی:* آرایه‌های Y_L و Y_R را مرتب‌سازی ادغامی کنید تا آرایه‌ی مرتب شده‌ی Y را به دست آورید.)

مسائل

۱-۳۳ لایه‌های محدب

با داشتن یک مجموعه‌ی Q از نقاط در یک صفحه، *لایه‌های محدب* Q (convex layer) را به صورت استقرایی تعریف می‌کنیم. اولین لایه‌ی محدب Q تشکیل شده است از نقاطی در Q که رأس‌های $CH(Q)$ هستند. برای $i > 1$، Q_i عبارت است از نقاط Q منهای نقاط لایه‌های محدب $1, 2, \ldots, i-1$. در این صورت، امین لایه‌ی محدب Q برابر است با $CH(Q_i)$ اگر $Q_i \neq \emptyset$، و در غیر این صورت تعریف نشده است.

I. یک الگوریتم با زمان $O(n^2)$ برای یافتن لایه‌های محدب یک مجموعه از n نقطه بدهید.

II. اثبات کنید که برای هر مدلی از محاسبات که به $\Omega(n \lg n)$ زمان برای مرتب‌سازی نیاز داشته باشد، $\Omega(n \lg n)$ زمان هم برای محاسبه‌ی لایه‌های محدب n نقطه نیاز است.

۲-۳۳ لایه‌های ماکسیمال

فرض کنید Q مجموعه‌ای از n نقطه در یک صفحه باشد. می‌گوییم نقطه‌ی (x, y) بر نقطه‌ی (x', y') *مسلط* (dominate) است اگر $x \geq x'$ و $y \geq y'$. یک نقطه در Q که هیچ نقطه‌ی دیگری بر آن مسلط نیست، یک نقطه‌ی *ماکسیمال* (maximal) در Q است. توجه داشته باشید که Q می‌تواند چندین نقطه‌ی ماکسیمال داشته باشد، که می‌توان به صورت زیر آن‌ها را در *لایه‌های ماکسیمال* (maximal layer) دسته‌بندی کرد. اولین لایه‌ی ماکسیمال L_1 برابر است با مجموعه‌ی نقاط ماکسیمال در Q. برای $i > 1$، امین لایه‌ی ماکسیمال L_i برابر است با مجموعه‌ی نقاط ماکسیمال $Q - \bigcup_{j=1}^{i-1} L_j$.

فرض کنید که Q تعداد k لایه‌ی ماکسیمال غیر تهی دارد، و فرض کنید y_i مختصات y

مسائل

چپ‌ترین نقطه در L_i باشد، برای $i = ۱,۲,...,k$. فعلاً فرض کنید که هیچ دو نقطه‌ای با مختصات یکسان x یا y در Q وجود ندارد.

I. نشان دهید که $y_۱ > y_۲ > ... > y_k$.

یک نقطه‌ی (x,y) را در نظر بگیرید که در سمت چپ تمام نقاط Q است، و مختصات y آن با مختصات y تمام نقاط Q متفاوت است. فرض کنید $Q' = Q \cup \{(x,y)\}$.

II. فرض کنید j اندیس کمینه باشد به طوری که $y_j < y$، مگر این که $y < y_k$، که در این صورت قرار می‌دهیم $j = k+۱$. نشان دهید که لایه‌های ماکسیمال Q' به صورت زیر هستند.

- اگر $j \le k$، آن گاه لایه‌های ماکسیمال Q' مشابه لایه‌های ماکسیمال Q هستند، غیر از این که L_j شامل (x,y) هم هست، به عنوان چپ‌ترین نقطه‌ی آن.
- اگر $j = k+۱$، آن گاه k لایه‌ی ماکسیمال اول Q' مانند لایه‌های ماکسیمال Q هستند، ولی علاوه بر آن، Q' یک لایه‌ی ماکسیمال غیر تهی $(k+۱)$ام هم دارد: $L_{k+۱} = \{(x,y)\}$.

III. یک الگوریتم با زمان $O(n \lg n)$ ارائه کنید که لایه‌های ماکسیمال مجموعه‌ی Q شامل n نقطه را محاسبه می‌کند. (*راهنمایی:* یک خط جارو از چپ به راست حرکت دهید.)

IV. اگر اجازه دهیم نقاط دارای مختصات x یا y یکسان باشند، آیا هیچ مشکلی پیش می‌آید؟ یک روش برای حل چنین مشکلاتی ارائه کنید.

۳-۳۳ روح‌ها و روح‌گیرها

گروهی از n روح‌گیر در حال نبرد با n روح هستند. هر روح‌گیر به یک اسلحه‌ی پروتونی مجهز است، که یک اشعه به سمت یک روح شلیک کرده و آن را از بین می‌برد. اشعه در یک خط مستقیم حرکت می‌کند، و زمانی که به یک روح برخورد می‌کند، پایان می‌یابد. روح‌گیرها تصمیم می‌گیرند طبق استراتژی زیر عمل کنند. هر یک از آن‌ها با یک روح جفت می‌شوند، که در کل n جفت روح‌گیر–روح را تشکیل می‌دهد، و سپس به صورت هم‌زمان، تمام روح‌گیرها به سمت روح‌های متناظر شلیک می‌کنند. همان طور که همه می‌دانیم، *بسیار* خطرناک است که اجازه دهیم دو اشعه با یکدیگر برخورد کنند، و بنابراین روح‌گیرها باید طوری جفت‌های خود را انتخاب کنند که هیچ دو اشعه‌ای با یکدیگر برخورد نداشته باشند.

فرض کنید که مکان روح‌گیرها و روح‌ها یک نقطه‌ی ثابت در صفحه است، و هیچ سه مکانی در یک خط قرار ندارند.

I. بحث کنید که یک خط عبور کننده از یک روح‌گیر و یک روح وجود دارد که تعداد روح‌گیرها و روح‌های یک سمت آن با هم برابر است. توضیح دهید که چگونه می‌توان چنین خطی را در زمان $O(n \lg n)$ یافت.

II. یک الگوریتم با زمان $O(n^۲ \lg n)$ برای تعیین جفت‌های روح‌گیر–روح ارائه کنید به صورتی که هیچ دو اشعه‌ای با یکدیگر برخورد نمی‌کنند.

۴-۳۳ برداشتن تکه‌چوب‌ها

پروفسور Charon مجموعه‌ای از n تکه‌چوب دارد، که با ترتیبی خاص بر روی یکدیگر قرار دارند. هر تکه‌چوب به کمک نقاط پایانی آن توصیف می‌شود، و هر نقطه‌ی پایانی عبارت است از یک سه‌تایی مرتب از مختصات (x, y, z). هیچ تکه‌چوبی عمودی نیست. پروفسور می‌خواهد تمام تکه‌چوب‌ها را بردارد، یکی در هر زمان، با این شرط که او فقط زمانی می‌تواند یک تکه‌چوب را بردارد که هیچ تکه‌چوب دیگری بر روی آن نباشد.

I. رویه‌ای ارائه کنید که به عنوان ورودی دو تکه‌چوب a و b را دریافت می‌کند، و گزارش می‌کند که a زیر b، روی b، و یا مستقل از b است.

II. الگوریتمی کارامد ارائه کنید که تشخیص می‌دهد آیا برداشتن تمام تکه‌چوب‌ها ممکن است یا خیر، و در صورت ممکن بودن، دنباله‌ای مجاز از برای برداشتن تکه‌چوب‌ها بازمی‌گرداند.

۵-۳۳ توزیع خلوت-پوسته

مسئله‌ی محاسبه‌ی پوسته‌ی محدب مجموعه‌ای از نقاط را در صفحه در نظر بگیرید که بر حسب یک توزیع تصادفی انتخاب شده‌اند. بعضی مواقع، تعداد نقاط، یا اندازه‌ی پوسته‌ی محدب n نقطه که بدین صورت انتخاب شده‌اند، دارای امیدریاضی $O(n^{1-\epsilon})$ است، برای یک ثابت $\epsilon > 0$. به چنین توزیعی، *خلوت-پوسته* (sparse-hulled) گفته می‌شود. توزیع‌های خلوت-پوسته شامل نمونه‌های زیر هستند:

- نقاط انتخاب شده به صورت یکنواخت از یک دیسک با شعاع واحد. امیدریاضی اندازه‌ی پوسته‌ی محدب $\theta(n^{1/3})$ است.
- نقاط انتخاب شده به صورت یکنواخت از نقاط داخلی یک چندضلعی محدب با k ضلع، برای هر ثابت دلخواه k. امیدریاضی اندازه‌ی پوسته‌ی محدب $\theta(\lg n)$ است.
- نقاط انتخاب شده بر حسب یک توزیع نرمال دو بعدی. امیدریاضی اندازه‌ی پوسته‌ی محدب $\theta(\sqrt{\lg n})$ است.

I. با داشتن دو چندضلعی محدب با n_1 و n_2 ضلع، نشان دهید چگونه می‌توان پوسته‌ی محدب تمام $n_1 + n_2$ نقطه را در زمان $O(n_1 + n_2)$ محاسبه کرد. (چندضلعی‌ها ممکن است با یکدیگر برخورد داشته باشند.)

II. نشان دهید که چگونه می‌توان پوسته‌ی محدب مجموعه‌ای از n نقطه را که به صورت مستقل و متناسب با یک توزیع خلوت-پوسته انتخاب شده‌اند، با امیدریاضی زمان $O(n)$ یافت. (*راهنمایی:* به صورت بازگشتی پوسته‌ی محدب $n/2$ نقطه‌ی اول و $n/2$ نقطه‌ی دوم را بیابید، و سپس نتایج را با هم ترکیب کنید.)

فصل

۳۴

NP- کامل‌ها

۳۴-۰ مقدمه

تقریباً تمام الگوریتم‌هایی که تا این جا آموختیم، *الگوریتم‌های با زمان چندجمله‌ای* (polynomial-time algorithms) بودند: برای یک ورودی با اندازه‌ی n، بدترین حالت زمان اجرای آن‌ها $O(n^k)$ است، برای یک ثابت k. طبیعی است که از خود بپرسیم آیا *تمام* مسئله‌ها را می‌توان در زمان چندجمله‌ای حل کرد؟ جواب خیر است. به عنوان مثال، مسائلی وجود دارند، مانند مسئله‌ی معروف تورینگ، یعنی «مسئله‌ی توقف»، که به کمک هیچ کامپیوتری نمی‌توان آن را حل کرد، صرف نظر از این که چه مقدار زمان در اختیار داریم. همچنین مسائلی وجود دارند که می‌توان آن‌ها را حل کرد، ولی نه در زمان $O(n^k)$ برای یک ثابت k. به طور کلی، مسئله‌هایی را که برای آن‌ها الگوریتم‌های چندجمله‌ای وجود دارد به صورت مسائل رام‌شدنی، یا ساده، در نظر می‌گیریم، و مسائلی را که به زمانی بیش از زمان چندجمله‌ای نیاز دارند، به صورت مسائل رام‌نشدنی، یا سخت.

با این حال، موضوع این فصل کلاسی جذاب از مسئله‌ها است با نام مسائل «NP-کامل»، که وضعیت آن‌ها نامشخص است. هنوز هیچ الگوریتم چندجمله‌ای برای یک مسئله‌ی NP-کامل پیدا نشده است، و همچنین هنوز هیچ کس نتوانسته ثابت کند که هیچ الگوریتم چندجمله‌ای برای آن‌ها وجود ندارد. این سؤال (وجود یا عدم وجود یک راه حل چندجمله‌ای برای مسائل NP-کامل)، که $P \neq NP$ نام دارد، یکی از عمیق‌ترین و پیچیده‌ترین مسائل پیش روی محققان علوم کامپیوتر از سال ۱۹۷۱ بوده است.

یک جنبه‌ی آزاردهنده‌ی مسائل NP-کامل این است که بسیاری از آن‌ها در ظاهر مشابه مسائلی

هستند که جواب چندجمله‌ای دارند. در هر یک از جفت مسائل زیر، یکی در زمان چندجمله‌ای قابل حل، و دیگری NP-کامل است، ولی تفاوت میان مسائل به نظر جزئی می‌آید:

- کوتاه‌ترین مسیر ساده در مقابل بلندترین مسیر ساده: در فصل ۲۴، دیدیم که حتی با وجود وزن‌های منفی، می‌توانیم *کوتاه‌ترین* مسیرها از یک مبدأ در یک گراف جهت‌دار $G = (V, E)$ را در زمان $O(VE)$ بیابیم. با این حال، یافتن بلندترین مسیر ساده بین دو رأس NP-کامل است. در واقع، این مسئله NP-کامل است حتی اگر وزن تمام یال‌ها ۱ باشد.
- تور اویلری در مقابل دور همیلتونی: یک *تور اویلری* (Euler tour) در یک گراف جهت‌دار و همبند $G = (V, E)$، دوری است که از هر *یال* G دقیقاً یک بار عبور می‌کند، ولی ممکن است از یک رأس بیش از یک بار عبور کند. طبق مسئله‌ی ۲۲-۳، در زمان $O(E)$ می‌توانیم تعیین کنیم که یک گراف دارای تور اویلری هست یا نه. در واقع در زمان $O(E)$ می‌توانیم یال‌های تور اویلری را هم تعیین کنیم. یک *دور همیلتونی* (Hamiltonian cycle) در یک گراف جهت‌دار $G = (V, E)$ یک دور ساده است که شامل تمام *رأس‌های* V می‌شود. تعیین این که یک گراف دور همیلتونی دارد یا نه، NP-کامل است. (بعداً در همین فصل، اثبات خواهیم کرد که تعیین این که یک گراف *بدون جهت* دور همیلتونی دارد یا نه NP-کامل است.)
- قابلیت ارضای 2-CNF در مقابل قابلیت ارضای 3-CNF: یک فرمول بولین شامل متغیرهایی است که مقدار آن‌ها ۰ یا ۱ است، به علاوه‌ی عملگرهای بولین مانند $\wedge$ (AND)، $\vee$ (OR)، و $\neg$ (NOT)، و پرانتزها. یک فرمول بولین *قابل ارضا* (satisfiable) است اگر یک مقداردهی ۰ و ۱ به متغیرهای آن وجود داشته باشد که مقدار کل آن را ۱ کند. عبارات مربوط به این مبحث را بعداً در همین فصل تعریف خواهیم کرد، ولی به صورت غیر رسمی، یک فرمول بولین *k-فرم نرمال عطفی* (k-conjunctive normal form)، یا k-CNF است، اگر به صورت AND چند عبارت OR متشکل از دقیقاً k متغیر یا معکوس آن‌ها باشد. برای مثال، فرمول بولین $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ یک 3-CNF است. (مقداردهی ارضا کننده‌ی آن $x_1 = 1$، $x_2 = 0$، $x_3 = 1$ است.) یک الگوریتم با زمان چندجمله‌ای برای تعیین این که آیا یک فرمول 2-CNF قابل ارضا است یا خیر وجود دارد، ولی همان طور که در ادامه‌ی این فصل خواهیم دید، تعیین این که یک فرمول 3-CNF قابل ارضا است یا خیر، NP-کامل است.

NP- کامل‌ها و کلاس‌های P و NP

در این فصل، به سه کلاس از مسائل اشاره خواهیم کرد: P، NP، و NPC، که این آخری همان کلاس مسائل NP-کامل است. در این جا آن‌ها را به صورت غیر رسمی تعریف می‌کنیم، و تعریف دقیق آن‌ها را بعداً ارائه خواهیم کرد.

کلاس P شامل مسائلی است که در زمان چندجمله‌ای قابل حل هستند. به صورت خاص‌تر، این کلاس شامل مسائلی است که می‌توان آن‌ها را در زمان $O(n^k)$ برای یک ثابت k حل کرد، که در آن n

اندازه‌ی ورودی مسئله است. اکثر مسائلی که در فصل‌های قبلی بررسی شدند در کلاس P قرار دارند.

کلاس NP شامل مسائلی است که در زمان چندجمله‌ای «تحقیق‌پذیر» (verifiable) هستند. منظور این است که اگر یک «تصدیق» (certificate) از یک جواب وجود داشته باشد، آن‌گاه می‌توانیم در زمان چندجمله‌ای نسبت به ورودی تحقیق کنیم که آیا این تصدیق صحیح است یا خیر. مثلاً در مسئله‌ی دور همیلتونی، با داشتن یک گراف جهت‌دار $G = (V, E)$، یک تصدیق عبارت خواهد بود از یک دنباله‌ی $\langle v_1, v_2, v_3, \ldots, v_{|V|} \rangle$ از $|V|$ رأس. به سادگی می‌توان در زمان چندجمله‌ای $(v_i, v_{i+1}) \in E$ برای $i = 1, 2, 3, \ldots, |V|$ و $(v_{|V|}, v_1) \in E$ را چک کرد. به عنوان یک مثال دیگر، برای قابلیت راضای 3-CNF، یک تصدیق عبارت خواهد بود از یک مقداردهی به متغیرها. به سادگی می‌توانیم در زمان چندجمله‌ای چک کنیم که این مقداردهی فرمول بولین را ارضا می‌کند یا خیر.

هر مسئله‌ای در P در NP هم هست، چرا که اگر یک مسئله در P باشد، آن گاه حتی بدون دریافت یک تصدیق می‌توانیم آن را در زمان چندجمله‌ای حل کنیم. این مفهوم را بعداً در همین فصل به صورت دقیق‌تر تعریف می‌کنیم، ولی فعلاً می‌توانیم قبول کنیم که $P \subseteq NP$. سؤال باقی‌مانده این است که آیا P یک زیرمجموعه‌ی اکید از NP است یا خیر.

به صورت غیر رسمی، یک مسئله به کلاس NPC تعلق دارد – و ما آن را *NP-کامل* می‌نامیم – اگر در کلاس NP باشد، و به «سختی» هر مسئله‌ی دیگری در NP. «سخت» بودن به اندازه‌ی دیگر مسائل NP را بعداً در همین فصل به صورت رسمی تعریف خواهیم کرد. فعلاً بدون اثبات این مسئله را مطرح خواهیم کرد که اگر بتوان یک مسئله‌ی NP-کامل را در زمان چندجمله‌ای حل کرد، آن گاه *برای تمام* مسائل NP-کامل یک الگوریتم با زمان چندجمله‌ای وجود دارد. اکثر نظریه پردازان علوم کامپیوتر اعتقاد دارند که مسائل NP-کامل رام‌نشدنی هستند، چرا که با وجود گستره‌ی وسیع مسائل NP-کامل که تا کنون مورد مطالعه قرار گرفته‌اند-بدون این که هیچ کس بتواند یک الگوریتم با زمان چندجمله‌ای برای هیچ کدام از آن‌ها بیابد-واقعاً تعجب آور خواهد بود اگر بتوان تمام آن‌ها را در زمان چندجمله‌ای حل کرد. از طرفی، با وجود تلاش‌های فراوانی که تا کنون برای اثبات رام‌نشدنی بودن مسائل NP-کامل صرف شده است-بدون یک نتیجه‌ی قطعی- نمی‌توانیم به سادگی احتمال قابل حل بودن مسائل NP-کامل در زمان چندجمله‌ای را رد کنیم.

برای بدل شدن به یک طراح الگوریتم خوب، شما باید اصول نظریه‌ی NP-کامل‌ها را بدانید. اگر بتوانید یک مسئله را به عنوان NP-کامل تشخیص دهید، آن گاه یک شاهد خوب برای رام‌نشدنی بودن آن ارائه کرده‌اید. به عنوان یک مهندس، آن گاه می‌توانید با صرف زمان خود بر روی یافتن یک الگوریتم تقریبی (فصل ۳۵ را ببینید) یا حل یک حالت خاص رام‌شدنی، به جای سعی در یافتن یک الگوریتم سریع که مسئله را کاملاً حل می‌کند، از زمان خود استفاده‌ی صحیح بکنید. به علاوه، بسیاری از مسائل طبیعی و جذاب که ظاهرا سختی آن‌ها به اندازه‌ی مرتب‌سازی، جستجو در گراف، و یا شار شبکه است، در واقع NP-کامل هستند. بنابراین آشنایی با این کلاس از مسائل از اهمیت خاصی برخوردار است.

مرور روش تشخیص NP-کامل بودن مسائل

تکنیک‌هایی که از آن‌ها برای نشان دادن NP-کامل بودن مسائل استفاده می‌کنیم، با تکنیک‌هایی که در طول این کتاب برای طراحی و تحلیل الگوریتم‌ها از آن‌ها استفاده شد تفاوت اساسی دارند. هنگام نشان دادن NP-کامل بودن یک مسئله، می‌خواهیم میزان سختی مسئله را ارائه کنیم (یا حداقل میزان سختی که ما برای آن حدس می‌زنیم)، نه میزان سادگی آن را. نمی‌خواهیم وجود یک الگوریتم کارامد را برای مسئله اثبات کنیم، بلکه می‌خواهیم نشان دهیم که احتمالاً هیچ الگوریتم کارمدی برای آن وجود ندارد. از این جهت، اثبات NP-کامل بودن شبیه اثبات بخش ۸-۱ برای کران پایین $\Omega(n \lg n)$ برای الگوریتم‌های مرتب‌سازی مقایسه‌ای است؛ با این حال تکنیک خاص استفاده شده برای اثبات NP-کامل بودن با درخت تصمیم استفاده شده در بخش ۸-۱ تفاوت دارد.

در اثبات NP-کامل بودن یک مسئله بر روی سه مفهوم کلیدی تکیه می‌کنیم:

مسائل تصمیمی در مقابل مسائل بهینه سازی

بسیاری از مسائل، *مسائل بهینه‌سازی* (optimization problems) هستند، که در آن هر جواب ممکن (یا «مجاز») یک مقدار متناظر دارد، و ما می‌خواهیم یک جواب ممکن با بهترین مقدار را بیابیم. مثلاً در مسئله‌ای که آن را SHORTEST-PAHT (کوتاه‌ترین مسیر) می‌نامیم، به ما یک گراف بدون جهت G با رأس‌های u و v داده شده است و می‌خواهیم یک مسیر از u به v بیابیم که شامل کم‌ترین رأس‌های ممکن است. (به عبارت دیگر، SHORTEST-PAHT همان مسئله‌ی کوتاه‌ترین مسیرها بین دو رأس در یک گراف بدون جهت و بدون وزن است.) با این حال، NP-کامل بودن مستقیماً برای مسائل بهینه‌سازی قابل کاربرد نیست، بلکه برای *مسائل تصمیمی* (decision problems) به کار می‌رود، که جواب آن‌ها فقط یک «بله» یا «خیر» است (یا به صورت رسمی‌تر، «۱» یا «۰»).

با این که اثبات NP-کامل بودن در محدوده‌ی مسائل تصمیمی جای می‌گیرد، ولی یک رابطه‌ی مفید میان مسائل بهینه‌سازی و مسائل تصمیمی وجود دارد. معمولاً می‌توانیم با تحمیل یک کران بر روی مقدار بهینه، یک مسئله‌ی بهینه‌سازی را به یک مسئله‌ی تصمیمی متناظر تبدیل کنیم. مثلاً یک مسئله‌ی تصمیمی متناظر برای مسئله‌ی SHORTEST-PAHT، که آن را PATH می‌نامیم، این است که برای یک گراف جهت‌دار G، رأس‌های u و v و عدد صحیح k، آیا یک مسیر از u به v با حداکثر k یال وجود دارد یا خیر.

هنگامی که می‌خواهیم نشان دهیم که یک مسئله‌ی بهینه‌سازی «سخت» است، رابطه‌ی بین یک مسئله‌ی بهینه‌سازی و مسئله‌ی تصمیمی متناظر آن به کمک ما می‌آید، چرا که مسئله‌ی تصمیمی از جهاتی ساده‌تر است، و یا حداقل سخت‌تر نیست. به عنوان یک مثال خاص، برای حل کردن PATH، می‌توانیم SHORTEST-PATH را حل کنیم، و سپس تعداد یال‌ها در کوتاه‌ترین مسیر را با پارامتر k از مسئله‌ی تصمیمی مقایسه کنیم. به عبارت دیگر، اگر یک مسئله‌ی بهینه‌سازی آسان باشد، مسئله‌ی تصمیمی متناظر آن هم آسان خواهد بود. به تعبیری دیگر که به NP-کامل بودن مرتبط‌تر است، اگر

بتوانیم نشان دهیم که یک مسئله‌ی تصمیمی سخت است، نشان داده‌ایم که مسئله‌ی بهینه‌سازی مربوط به آن هم سخت است. بنابراین، حتی با این که نظریه‌ی NP-کامل بودن توجه خود را بر روی مسائل تصمیمی متمرکز می‌کند، مفاهیمی برای مسائل بهینه‌سازی هم در بر دارد.

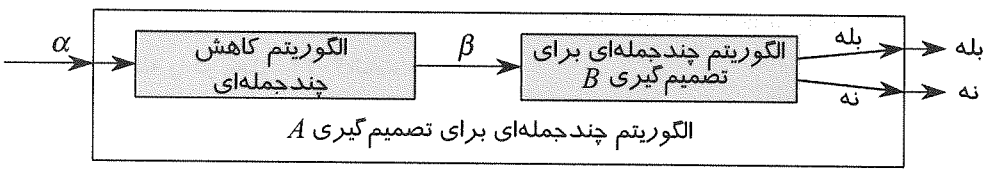
کاهش‌ها

مفهوم بالا در مورد این که یک مسئله سخت‌تر یا ساده‌تر از یک مسئله‌ی دیگر نیست، حتی برای دو مسئله‌ی تصمیمی هم کاربرد دارد. از این ایده تقریباً در تمام اثبات‌های NP-کامل بودن استفاده می‌کنیم، به صورت زیر. اجازه دهید فرض کنیم که یک مسئله‌ی تصمیمی مانند A را در نظر بگیریم، که می‌خواهیم آن را در زمان چندجمله‌ای حل کنیم. به ورودی یک مسئله‌ی خاص، یک *نمونه* (instance) از آن مسئله می‌گوییم؛ مثلاً، در مسئله‌ی PATH، یک نمونه از مسئله عبارت خواهد بود از یک گراف خاص G، رأس‌های خاص u و v در G، و یک عدد صحیح خاص k. اکنون فرض کنید که یک مسئله‌ی تصمیمی دیگر، مانند B وجود دارد که می‌دانیم چطور می‌توان آن را در زمان چندجمله‌ای حل کرد. نهایتاً، فرض کنید که یک رویه داریم که هر نمونه‌ی α از A را با خصوصیات زیر به یک نمونه‌ی β از B تبدیل می‌کند:

۱. تبدیل در زمان چندجمله‌ای انجام می‌شود.
۲. جواب‌ها مشابه هستند. یعنی، جواب α «بله» است اگر و فقط اگر جواب β هم «بله» باشد.

به چنین رویه‌ای یک *الگوریتم کاهش* (reduction algorithm) با زمان چندجمله‌ای می‌گوییم، و همان طور که شکل ۳۴-۱ نشان می‌دهد، این رویه یک روش برای حل مسئله‌ی A در زمان چندجمله‌ای به ما می‌دهد:

۱. با داشتن یک نمونه‌ی α از مسئله‌ی A، با استفاده از الگوریتم کاهش چندجمله‌ای، این نمونه را به یک نمونه‌ی β از مسئله‌ی B تبدیل می‌کنیم.
۲. الگوریتم تصمیم چندجمله‌ای را برای B بر روی β اجرا می‌کنیم.
۳. از جواب β به عنوان جوابی برای α استفاده می‌کنیم.



شکل ۳۴-۱ استفاده از یک الگوریتم کاهش با زمان چندجمله‌ای برای حل یک مسئله‌ی تصمیمی A در زمان چندجمله‌ای، با داشتن یک الگوریتم تصمیم با زمان چندجمله‌ای برای یک مسئله‌ی B. در زمان چندجمله‌ای، نمونه‌ی α از مسئله‌ی A را به یک نمونه‌ی β از مسئله‌ی B تبدیل می‌کنیم، β را در زمان چندجمله‌ای حل می‌کنیم، و سپس از جواب β به عنوان جوابی برای α استفاده می‌کنیم.

اگر تمام این مراحل در زمان چندجمله‌ای انجام شوند، مجموع سه مرحله هم در زمان چندجمله‌ای انجام می‌شود، و بنابراین یک راه برای تصمیم‌گیری در مورد α در زمان چندجمله‌ای وجود دارد. به عبارت دیگر، با «کاهش» مسئله‌ی A به مسئله‌ی B، از «سادگی» B برای اثبات «سادگی» A استفاده می‌کنیم.

با به خاطر آوردن این که NP-کامل بودن در مورد میزان سختی یک مسئله است، نه در مورد میزان سادگی آن، از کاهش در جهت عکس استفاده می‌کنیم تا نشان دهیم یک مسئله NP-کامل است. اجازه دهید ایده را یک قدم پیشتر ببریم، و نشان دهیم که چطور می‌توان با استفاده از کاهش در زمان چندجمله‌ای نشان داد که هیچ الگوریتم چندجمله‌ای نمی‌تواند برای یک مسئله‌ی خاص B وجود داشته باشد. فرض کنید که یک مسئله‌ی تصمیمی A داریم که می‌دانیم هیچ الگوریتم چندجمله‌ای برای آن وجود ندارد. (اجازه دهید فعلاً در مورد این که چگونه مسئله‌ای مانند A بیابیم خود را نگران نکنیم.) همچنین فرض کنید که یک کاهش چندجمله‌ای برای تبدیل نمونه‌های A به نمونه‌های B داریم. اکنون با استفاده از برهان خلف می‌توانیم نشان دهیم که هیچ الگوریتمی با زمان چندجمله‌ای برای B وجود ندارد. فرض کنید که B یک الگوریتم چندجمله‌ای دارد. در این صورت، با استفاده از متد نشان داده شده در شکل ۳۴-۱، یک راه خواهیم داشت برای حل A در زمان چندجمله‌ای، که با این فرض که هیچ الگوریتمی با زمان چندجمله‌ای برای A وجود ندارد، تناقض دارد.

برای NP-کامل‌ها، نمی‌توانیم فرض کنیم که به هیچ وجه، هیچ الگوریتمی با زمان چندجمله‌ای برای A وجود ندارد. با این حال، متدولوژی اثبات مشابه است، بدین صورت که با این فرض که مسئله‌ی A NP-کامل است، اثبات می‌کنیم که مسئله‌ی B هم NP-کامل است.

یک مسئله‌ی NP-کامل اولیه

چون تکنیک کاهش بر این پایه است که ابتدا یک مسئله داشته باشیم که می‌دانیم NP-کامل است، تا بتوانیم اثبات کنیم که یک مسئله‌ی دیگر NP-کامل است، ابتدا به یک مسئله‌ی NP-کامل «اولیه» نیاز داریم. مسئله‌ای که از آن استفاده می‌کنیم، مسئله‌ی قابلیت ارضای مدار است، که در آن به ما یک مدار ترکیبی بولین از گیت‌های AND، OR، و NOT داده شده است، و می‌خواهیم بدانیم که آیا هیچ مجموعه‌ای از مقادیر ورودی برای این مدار وجود دارد که خروجی آن را ۱ کند یا خیر. NP-کامل بودن این مسئله را در بخش ۳۴-۳ اثبات خواهیم کرد.

خلاصه‌ی فصل

این بخش جنبه‌های مختلف NP-کامل بودن را بررسی می‌کند، که اکثراً بر پایه‌ی تحلیل الگوریتم‌ها است. در بخش ۳۴-۱، مفهوم «مسئله» را فرمول‌بندی می‌کنیم، و کلاس پیچیدگی P را از مسائل تصمیمی قابل حل در زمان چندجمله‌ای، تعریف می‌کنیم. همچنین می‌بینیم که این مفاهیم چگونه در چارچوب نظریه‌ی زبان‌های فرمال می‌گنجند. بخش ۳۴-۲ کلاس NP از مسائل تصمیم‌گیری را

تعریف می‌کند، که جواب آن‌ها را می‌توان در زمان چندجمله‌ای تصدیق کرد. همچنین در این بخش سؤال $P \neq NP$ مطرح می‌شود.

بخش ۳۴-۳ نشان می‌دهد که چگونه می‌توان رابطه‌ی بین مسائل را با استفاده از «کاهش» در زمان چندجمله‌ای بررسی کرد. در این بخش NP-کامل بودن تعریف، و اثبات NP-کامل بودن مسئله‌ی «قابلیت ارضای مدار» ارائه می‌شود. با یافتن یک مسئله‌ی NP-کامل، در بخش ۳۴-۴ نشان می‌دهیم که چگونه می‌توان NP-کامل بودن مسائل دیگر را با استفاده از متد کاهش بسیار ساده‌تر اثبات کرد. با نشان دادن این که مسائل ارضاپذیری دوفرموله، NP-کامل هستند، این متدولوژی روشن می‌شود. NP-کامل بودن چندین مسئله‌ی دیگر در بخش ۳۴-۵ نشان داده خواهد شد.

۳۴-۱ زمان چندجمله‌ای

بررسی NP-کامل‌ها را با فرمول‌بندی مفهوم مسائل قابل حل در زمان چندجمله‌ای آغاز می‌کنیم. معمولاً این مسائل به عنوان مسائل رام‌شدنی مطرح می‌شوند، ولی به دلایل فلسفی، نه ریاضی. می‌توانیم سه بحث در تصدیق این مسئله ارائه دهیم.

اول، با این که منطقی است که مسئله‌ای را که به زمان $\theta(n^{100})$ نیاز دارد، به عنوان رام‌نشدنی در نظر بگیریم، تعداد مسائلی که به زمان چندجمله‌ای با چنین درجه‌ی بالایی نیاز دارند، بسیار کم است. مسائل محاسبه‌پذیر با زمان چندجمله‌ای که در عمل به آن‌ها بر می‌خوریم به زمان بسیار کم‌تری نیاز دارند. تجربه نشان داده است که وقتی یک الگوریتم با زمان چندجمله‌ای برای یک مسئله یافت می‌شود، معمولاً الگوریتم‌های کارامدتر هم به دنبال آن کشف می‌شوند. حتی اگر زمان اجرای بهترین الگوریتم کنونی مسئله $\theta(n^{100})$ باشد، احتمالاً یک الگوریتم با زمان اجرای بسیار بهتر به زودی یافت می‌شود.

دوم، برای بسیاری از مدل‌های منطقی محاسبه، یک مسئله را که بتوان آن را در یک مدل زمان چندجمله‌ای حل کرد، در مدل‌های دیگر هم در زمان چندجمله‌ای قابل حل است. مثلاً، کلاس مسائلی که می‌توان آن‌ها را بر روی ماشین‌های با دسترسی تصادفی سری، که در اکثر بخش‌های این کتاب از آن‌ها استفاده شده است، در زمان چندجمله‌ای حل کرد، معادل کلاس مسائلی هستند که می‌توان آن‌ها را بر روی ماشین‌های تورینگ در زمان چندجمله‌ای حل کرد. این کلاس همچنین معادل است با کلاس مسائل قابل حل در زمان چندجمله‌ای بر روی ماشین‌های موازی که در آن‌ها تعداد پردازنده‌ها با اندازه‌ی ورودی به صورت چندجمله‌ای رشد می‌کند.

سوم، کلاس مسائل قابل حل در زمان چندجمله‌ای خصوصیات بستاری جالبی دارند، چرا که چندجمله‌ای‌ها تحت اعمال جمع، ضرب، و ترکیب بسته هستند. مثلاً، اگر خروجی یک الگوریتم چندجمله‌ای به ورودی یکی دیگر داده شود، الگوریتم ترکیبی همچنان چندجمله‌ای است. تمرین ۳۴-۱-۵ از شما می‌خواهد نشان دهید که اگر یک الگوریتم تعداد ثابتی فراخوانی بر روی زیرروال‌های با زمان چندجمله‌ای بکند، و مقداری کار اضافه هم انجام دهد که آن هم در زمان چندجمله‌ای اجرا می‌شود، زمان اجرای الگوریتم حاصل چندجمله‌ای است.

مسائل انتزاعی

برای درک کلاس مسائل قابل حل در زمان چندجمله‌ای، ابتدا باید یک مفهوم از «مسئله» داشته باشیم. یک *مسئله‌ی انتزاعی* (abstract problem) Q را به صورت یک رابطه میان یک مجموعه‌ی I از *نمونه‌های* مسئله و یک مجموعه‌ی S از *جواب‌های* مسئله تعریف می‌کنیم. به عنوان مثال، یک نمونه از SHORTEST-PATH یک سه‌تایی است شامل یک گراف و دو رأس. یک جواب، دنباله‌ای است از رأس‌ها در گراف، و احتمالاً دنباله‌ی تهی که نشان دهنده‌ی عدم وجود یک مسیر است. خود مسئله‌ی SHORTEST-PATH یک رابطه است که هر نمونه از یک گراف و دو رأس را با یک کوتاه‌ترین مسیر در گراف که دو رأس را به هم متصل می‌کند، ارتباط می‌دهد. چون کوتاه‌ترین مسیرها لزوماً یکتا نیستند، یک نمونه‌ی داده شده از یک مسئله ممکن است بیش از یک جواب داشته باشد.

این فرمول‌بندی از مسائل انتزاعی، کلی‌تر از آن است که در این جا به آن نیاز داریم. همان طور که در بالا دیدیم، نظریه‌ی NP-کامل بودن به *مسائل تصمیم‌گیری* محدود می‌شود: مسائلی که جواب آن‌ها به صورت بله/خیر است. در این حالت، می‌توانیم یک مسئله‌ی تصمیم‌گیری انتزاعی را به صورت یک تابع ببینیم که نمونه‌های مجموعه‌ی I را به جواب‌های مجموعه‌ی $\{0,1\}$ نگاشت می‌کند. به عنوان مثال، یک مسئله‌ی تصمیم‌گیری مربوط به SHORTEST-PATH، مسئله PATH است که قبلاً دیدیم. اگر $i = \langle G,u,v,k \rangle$ یک نمونه از مسئله‌ی تصمیم‌گیری PATH باشد، آن گاه $\text{PATH}(i) = 1$ (یا همان «بله») اگر یک کوتاه‌ترین مسیر از u به v حداکثر k یال داشته باشد، و در غیر این صورت $\text{PATH}(i) = 0$ (یا همان «نه»). بسیاری از مسائل انتزاعی مسائل تصمیم‌گیری نیستند، بلکه *مسائل بهینه‌سازی* هستند، که در آن‌ها مقداری وجود دارد که باید کمینه یا بیشینه شود. با این حال همان طور که در بالا دیدیم، معمولاً به سادگی می‌توان یک مسئله‌ی بهینه‌سازی را به یک مسئله‌ی تصمیم‌گیری تبدیل کنیم، که از مسئله‌ی اولیه‌ی سخت‌تر نیست.

کدگذاری‌ها

اگر یک برنامه‌ی کامپیوتری بخواهد یک مسئله‌ی انتزاعی را حل کند، نمونه‌های مسئله باید به شکلی نمایش داده شوند که برنامه آن‌ها را درک کند. یک *کدگذاری* (encoding) از یک مجموعه‌ی S از اشیای انتزاعی، یک نگاشت e از S به مجموعه‌ی رشته‌های دودویی است.[1] مثلاً همه‌ی ما با کدگذاری اعداد طبیعی $N = \{0,1,2,3,4,\ldots\}$ به صورت رشته‌های $\{0,1,10,11,100,\ldots\}$ آشنا هستیم. با استفاده از این کدگذاری داریم $e(17) = 10001$. اگر نمایش کامپیوتری کاراکترهای صفحه‌کلید را دیده باشید احتمالاً با کد ASCII آشنایی دارید، که در آن کدگذاری A به صورت ۱۰۰۰۰۰۱ است. حتی یک شیئ مرکب را می‌توان با ترکیب نمایش بخش‌های تشکیل دهنده‌ی آن به صورت یک رشته‌ی دودویی کد کرد. چندضلعی‌ها، گراف‌ها، توابع، جفت‌های مرتب، برنامه‌ها–همه را می‌توان به صورت رشته‌های دودویی کد کرد.

---

[1] نیازی نیست که هم‌دامنه‌ی e رشته‌های دودویی باشد؛ هر مجموعه‌ای از رشته‌ها برروی یک الفبا که حداقل دو سمبل دارد، مناسب است.

بنابراین، یک الگوریتم کامپیوتری که یک مسئله‌ی تصمیم‌گیری انتزاعی را «حل می‌کند»، در واقع یک کدگذاری از یک نمونه از مسئله را به عنوان ورودی دریافت می‌کند. به مسئله‌ای که مجموعه‌ی نمونه‌های آن مجموعه‌ی رشته‌های دودویی باشد، یک *مسئله‌ی عینی* (concrete problem) می‌گوییم. می‌گوییم یک الگوریتم یک مسئله‌ی عینی را در زمان $O(T(n))$ *حل می‌کند* اگر وقتی که یک نمونه‌ی i از مسئله با طول $n = |i|$ را به آن می‌دهیم، الگوریتم بتواند در زمان $O(T(n))$ جواب را تولید کند.[1] بنابراین یک مسئله‌ی عینی، *قابل حل در زمان چندجمله‌ای* (polynomial-time solvable) است اگر یک الگوریتم با زمان $O(n^k)$ برای یک ثابت k برای حل آن وجود داشته باشد.

اکنون می‌توانیم به صورت رسمی *کلاس پیچیدگی* P را به صورت مجموعه‌ی مسائل تصمیم‌گیری عینی که قابل حل در زمان چندجمله‌ای هستند، تعریف کنیم.

با استفاده از کدگذاری‌ها می‌توانیم مسائل انتزاعی را به مسائل عینی تبدیل کنیم. با داشتن یک مسئله‌ی تصمیم‌گیری انتزاعی Q که یک مجموعه‌ی I از نمونه‌ها را به $\{0,1\}$ نگاشت می‌کند، می‌توان از یک کدگذاری $e : I \to \{0,1\}^*$ استفاده و یک مسئله‌ی تصمیم‌گیری عینی مربوط را تولید کرد، که آن را با $e(Q)$ نشان می‌دهیم[2]. اگر جواب یک نمونه از یک مسئله‌ی انتزاعی $i \in I$، $Q(i) \in \{0,1\}$ باشد، آن گاه جواب نمونه‌ی مسئله‌ی عینی $e(i) \in \{0,1\}^*$ هم $Q(i)$ است. به عنوان یک نکته‌ی فنی، ممکن است رشته‌های دودویی وجود داشته باشند که نشان دهنده‌ی هیچ نمونه‌ی معنی‌داری از یک مسئله‌ی انتزاعی نباشند. برای سادگی، فرض می‌کنیم که تمام چنین رشته‌هایی به ۰ نگاشت شده‌اند. بنابراین، مسئله‌ی عینی بر روی رشته‌های دودویی نمونه‌ها همان جواب‌هایی را تولید می‌کند که مسئله‌ی انتزاعی بر روی نمونه‌های انتزاعی تولید می‌کند.

می‌خواهیم با استفاده از کدگذاری‌ها به عنوان یک پل، تعریف قابلیت حل در زمان چندجمله‌ای را از مسائل عینی به مسائل انتزاعی گسترش دهیم، ولی در عین حال، می‌خواهیم این تعریف مستقل از هر کدگذاری خاص باشد. یعنی، کارایی جواب یک مسئله نباید به نحوه‌ی کدگذاری مسئله وابسته باشد. متأسفانه این وابستگی را به مقدار زیادی می‌توان حس کرد. به عنوان مثال، فرض کنید که باید یک عدد صحیح k را به عنوان تنها ورودی به یک الگوریتم بدهیم، و فرض کنید که زمان اجرای الگوریتم $\theta(k)$ است. اگر عدد k به صورت *یگانی* (unary) – رشته‌ای از k سمبل ۱ – ارائه شده باشد، آن گاه زمان اجرای الگوریتم برای یک ورودی به طول n، $O(n)$ خواهد بود، که یک زمان چندجمله‌ای است. از طرفی اگر از نمایش طبیعی‌تر دودویی عدد صحیح k استفاده کنیم، آن گاه طول ورودی $n = \lfloor \lg k \rfloor + 1$ خواهد بود. در این حالت، زمان اجرای الگوریتم $\theta(k) = \theta(2^n)$ است، که نسبت به طول ورودی از مرتبه‌ی نمایی است. بنابراین بسته به کدگذاری، الگوریتم ممکن است در زمان چندجمله‌ای یا زمان فرا–چندجمله‌ای اجرا شود.

---

[1] فرض می‌کنیم که خروجی الگوریتم جدا از ورودی آن باشد. از آن جایی که برای تولید هر بیت از خروجی حداقل به یک واحد زمان نیاز داریم، و در الگوریتم $O(T(n))$ واحد زمان صرف می‌شود، اندازه‌ی خروجی از مرتبه‌ی $O(T(n))$ است.

[2] $\{0,1\}^*$ نشان‌دهنده‌ی مجموعه‌ی تمام رشته‌های متشکل از عناصر مجموعه‌ی $\{0,1\}$ است.

نحوه‌ی کدگذاری یک مسئله‌ی انتزاعی تأثیر به‌سزایی در درک ما از مفهوم زمان چندجمله‌ای دارد. در واقع نمی‌توانیم از حل یک مسئله‌ی انتزاعی صحبت کنیم بدون این که ابتدا کدگذاری آن را مشخص کرده باشیم. با این حال، در عمل، اگر از کدگذاری‌های «پرهزینه» مانند کدهای یگانی پرهیز کنیم، کدگذاری واقعی یک مسئله تفاوت بسیار کمی در قابلیت حل آن در زمان چندجمله‌ای ایجاد می‌کند. به عنوان مثال، نمایش اعداد در مبنای ۳ به جایی دودویی هیچ تأثیری بر روی این که یک مسئله می‌تواند در زمان چندجمله‌ای حل شود یا خیر ندارد، چرا که می‌توان یک عدد در مبنای ۳ را در زمان چندجمله‌ای به مبنای ۲ تبدیل کرد.

محاسبه‌پذیر در زمان چندجمله‌ای می‌گوییم یک تابع $f : \{0,1\}^* \to \{0,1\}^*$ (polynomial-time computable) است اگر یک الگوریتم A با زمان چندجمله‌ای وجود داشته باشد که با دریافت هر ورودی $x \in \{0,1\}^*$، $f(x)$ را به عنوان خروجی تولید می‌کند. برای بعضی مجموعه‌های I از نمونه مسئله‌ها، می‌گوییم دو کدگذاری e_1 و e_2 *چندجمله‌ای-مرتبط* (polynomially related) هستند اگر دو تابع محاسبه‌پذیر در زمان چندجمله‌ای f_{12} و f_{21} وجود داشته باشد به طوری که برای هر $i \in I$ داشته باشیم $f_{12}(e_1(i)) = e_2(i)$ و $f_{21}(e_2(i)) = e_1(i)$.[1] یعنی می‌توان به کمک یک الگوریتم چندجمله‌ای، کدگذاری $e_2(i)$ را از روی کدگذاری $e_1(i)$ محاسبه کرد، و بالعکس. اگر دو کدگذاری e_1 و e_2 از یک مسئله‌ی انتزاعی، چندجمله‌ای-مرتبط باشند، قابل حل بودن مسئله در زمان چندجمله‌ای بستگی به کدگذاری مورد استفاده (میان e_1 و e_2) نخواهد داشت، همان طور که لم زیر نشان می‌دهد.

لم ۱-۲۲ فرض کنید Q یک مسئله‌ی تصمیم‌گیری انتزاعی باشد، با یک مجموعه نمونه‌ی I، و فرض کنید e_1 و e_2 کدگذاری‌های چندجمله‌ای- مرتبط بر روی I باشند. آن گاه $e_1(Q) \in P$ اگر و فقط اگر $e_2(Q) \in P$.

اثبات فقط نیاز داریم یک جهت از قضیه را اثبات کنیم، چرا که جهت دیگر به صورت مشابه اثبات می‌شود. بنابراین، فرض کنید که $e_1(Q)$ را بتوان در زمان $O(n^k)$ برای یک ثابت k محاسبه کرد. همچنین فرض کنید که برای هر نمونه مسئله‌ی i، کدگذاری $e_1(i)$ را می‌توان در زمان $O(n^c)$ از روی $e_2(i)$ محاسبه کرد، برای یک ثابت c، که در آن $n = |e_2(i)|$. برای حل مسئله‌ی $e_2(Q)$ بر روی ورودی $e_2(i)$، ابتدا $e_1(i)$ را محاسبه، و سپس الگوریتم را برای $e_1(Q)$ بر روی $e_1(i)$ اجرا می‌کنیم. این فرایند چقدر طول می‌کشد؟ تبدیل کدگذاری‌ها به $O(n^c)$ زمان نیاز دارد، و بنابراین $|e_1(i)| = O(n^c)$، چرا که خروجی یک کامپیوتر سری نمی‌تواند بزرگ‌تر از زمان اجرای آن باشد. حل مسئله بر روی $e_1(i)$ به $O(|e_1(i)|^k) = O(n^{ck})$ نیاز دارد، که چندجمله‌ای است، چرا که c و k ثابت هستند. ■

[1] علاوه بر این، از نظر فنی نیاز داریم که توابع f_{12} و f_{21} «غیرنمونه‌ها را به غیرنمونه‌ها تبدیل کنند». یک غیرنمونه از یک کدگذاری e یک رشته‌ی $x \in \{0,1\}^*$ است به طوری که هیچ نمونه‌ای مانند i وجود نداشته باشد که $e(i) = x$. نیاز داریم که برای تمام غیرنمونه‌های x از کدگذاری e_1 و تمام غیرنمونه‌های x' از کدگذاری e_2، داشته باشیم $f_{12}(x) = y$، که در آن y یک غیرنمونه از e_2 است، و $f_{21}(x') = y'$، که در آن y' یک غیرنمونه از e_1 است.

بنابراین، این که یک نمونه‌های مسئله‌ی انتزاعی به صورت دودویی یا مبنای ۳ کدگذاری شده باشند، «پیچیدگی» آن (قابلیت حل آن در زمان چندجمله‌ای) را تغییر نمی‌دهد، ولی اگر نمونه‌ها به صورت یگانی کدگذاری شده باشند، پیچیدگی آن ممکن است تغییر کند. برای این که قادر باشیم به صورت مستقل از کدگذاری بر روی مسائل بحث کنیم، به طور کلی فرض می‌کنیم که مسائل به صورتی منطقی و مختصر کدگذاری شده‌اند، مگر این که خلاف آن ذکر شود. به طور دقیق‌تر، فرض خواهیم کرد که کدگذاری یک عدد صحیح چندجمله‌ای- مرتبط به نمایش دودویی آن است، و این که کدگذاری یک مجموعه‌ی متناهی، چندجمله‌ای- مرتبط است به کدگذاری لیست اعضای آن، که در میان دو آکولاد قرار دارند و با کاما از یکدیگر جدا شده‌اند. (ASCII یک کدگذاری به این روش است.) با در دست داشتن یک کدگذاری «استاندارد» مانند این، می‌توانیم کدگذاری‌هایی منطقی از اشیای ریاضی دیگر، مانند سه‌تایی‌ها، گراف‌ها، و فرمول‌ها هم به دست آوریم. برای نشان دادن کدگذاری استاندارد یک شیئ، آن را در میان آکولادهای زاویه‌دار (علامت‌های بزرگ‌تر و کوچک‌تر) قرار می‌دهیم. بنابراین، $\langle G \rangle$ نشان‌دهنده‌ی کدگذاری استاندارد گراف G است.

تا زمانی که از یک کدگذاری استفاده کنیم که چندجمله‌ای-مرتبط با این کدگذاری استاندارد باشد، می‌توانیم مستقیماً در مورد مسائل انتزاعی صحبت کنیم، بدون این که به هیچ کدگذاری خاصی ارجاع دهیم، با این اطلاع که کدگذاری مورد استفاده هیچ تأثیری بر روی قابل حل بودن مسئله‌ی انتزاعی در زمان چندجمله‌ای ندارد. از این پس به طور کلی فرض می‌کنیم که تمام نمونه‌های مسئله‌ها رشته‌های دودویی هستند که نشان‌دهنده‌ی کدگذاری استاندارد هستند، مگر این که صریحاً خلاف آن گفته شود. همچنین معمولاً از تفاوت میان مسائل انتزاعی و عینی صرف نظر می‌کنیم. با این حال، خواننده باید مواظب مسائلی باشد که در عمل پیش می‌آیند و در آن‌ها کدگذاری مورد استفاده بدیهی نیست، و البته در پیچیدگی مسئله تأثیر می‌گذارد.

چارچوبی برای زبان‌های فرمال

تمرکز بر روی مسائل تصمیم‌گیری این مزیت را دارد که هنگام کار با آن‌ها می‌توان از امکانات نظریه‌ی زبان‌های فرمال استفاده کرد. اجازه دهید بعضی از تعریف‌های اولیه‌ی این نظریه را مرور کنیم. یک *الفبای* (alphabet) Σ مجموعه‌ای متناهی است از سمبل‌ها. یک *زبان* (language) L بر روی Σ هر مجموعه‌ای از رشته‌ها است که با استفاده از سمبل‌های Σ ساخته شده است. برای مثال اگر $\Sigma = \{0,1\}$، آن‌گاه $L = \{10,11,101,111,1011, 1101,10001,\ldots\}$ یک زبان است حاوی نمایش دودویی اعداد اول. *رشته‌ی تهی* (empty string) را با ε و *زبان تهی* (empty language) را با $\emptyset$ نشان می‌دهیم. زبان تمام رشته‌های روی Σ را با Σ^* نشان می‌دهیم. برای مثال اگر $\Sigma = \{0,1\}$، آن گاه $\Sigma^* = \{\varepsilon,0,1,00,01,10,11,000,\ldots\}$ مجموعه‌ی تمام رشته‌های دودویی است. هر زبان L روی Σ زیرمجموعه‌ای از Σ^* است.

اعمال مختلفی بر روی زبان‌ها وجود دارد. اعمال نظریه‌ی مجموعه‌ها، مانند *اجتماع* (union) و *اشتراک* (intersection)، مستقیماً از تعاریف مجموعه‌ها حاصل می‌شوند. مکمل (complement) زبان L

را به صورت $\overline{L} = \Sigma^* - L$ تعریف می‌کنیم. *اتصال* (concatenation) دو زبان L_1 و L_2 عبارت است از زبان

$$L = \{x_1 x_2 : x_1 \in L_1 \text{ و } x_2 \in L_2\} \qquad (۱-۳۴)$$

بستار (closure) یا *ستاره‌ی Kleene* (Kleene star) یک زبان L عبارت است از زبان

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots$$

که در آن L^k از k بار اتصال زبان L به خودش به دست می‌آید.

از دید نظریه‌ی زبان‌ها، مجموعه‌ی نمونه‌های هر مسئله‌ی تصمیم‌گیری Q به سادگی برابر است با مجموعه‌ی Σ^*، که در آن $\Sigma = \{0,1\}$. چون می‌توان Q را به طور کامل با نمونه‌هایی تعریف کرد که جواب ۱ (بله) تولید می‌کنند، می‌توانیم Q را به صورت زبان L بر روی $\Sigma = \{0,1\}$ ببینیم، که در آن

$$L = \{x \in \Sigma^* : Q(x) = 1\}$$

به عنوان مثال، مسئله‌ی تصمیم‌گیری PATH زبان متناظر زیر را دارد:

$$\text{PATH} = \left\{ \langle G, u, v, k \rangle \;\middle|\; \begin{array}{l} G = (V, E) \text{ یک گراف بدون جهت است، } u, v \in V, k \geq 0 \text{ یک عدد} \\ \text{صحیح است، و در } G \text{ یک مسیر از } u \text{ به } v \text{ با حداکثر } k \text{ یال وجود دارد.} \end{array} \right\}$$

(در صورت سادگی، بعضی مواقع از یک نام – در این جا PATH – هم برای نشان دادن یک مسئله‌ی تصمیم‌گیری استفاده خواهیم کرد و هم برای نشان دادن زبان متناظر آن.)

چارچوب زبان‌های فرمال به ما اجازه می‌دهد که رابطه‌ی بین مسائل تصمیم‌گیری و الگوریتم‌هایی که آن‌ها را حل می‌کنند را به صورت مختصر تعریف کنیم. می‌گوییم یک الگوریتم A یک رشته‌ی $x \in \{0,1\}^*$ را *می‌پذیرد* اگر با دریافت ورودی x، خروجی الگوریتم $A(x)$ برابر ۱ باشد. زبان *پذیرفته شده* توسط الگوریتم A برابر است با مجموعه‌ی رشته‌های $L = \{x \in \{0,1\}^* : A(x) = 1\}$، یعنی، مجموعه‌ی رشته‌هایی که الگوریتم می‌پذیرد. یک الگوریتم A یک رشته‌ی x را *رد می‌کند* اگر $A(x) = 0$.

حتی اگر یک زبان L توسط یک الگوریتم A پذیرفته شود، الگوریتم A لزوماً یک رشته‌ی $x \notin L$ را رد نمی‌کند. به عنوان نمونه، الگوریتم ممکن است برای همیشه در یک حلقه باقی بماند. یک الگوریتم A یک زبان L را *تصمیم می‌گیرد* اگر تمام رشته‌های دودویی در L توسط A پذیرفته و تمام رشته‌های دودویی که در L نیستند توسط A رد شوند. یک زبان L توسط یک الگوریتم A *در زمان چندجمله‌ای پذیرفته می‌شود* اگر توسط A پذیرفته شود و به علاوه یک ثابت k وجود داشته باشد به طوری که برای هر رشته‌ی $x \in L$ به طول n، الگوریتم x را در زمان $O(n^k)$ بپذیرد. یک زبان L توسط یک الگوریتم A *در زمان چندجمله‌ای تصمیم‌گیری می‌شود* اگر یک ثابت k وجود داشته باشد به طوری که برای هر رشته‌ی $x \in \{0,1\}^*$ به طول n، الگوریتم به در زمان $O(n^k)$ به درستی تصمیم می‌گیرد که x عضو L است یا خیر. بنابراین برای پذیرش یک زبان، یک الگوریتم فقط نیاز دارد که

برای رشته‌های درون L خروجی صحیح تولید کند، ولی برای تصمیم‌گیری یک زبان، الگوریتم باید تمام رشته‌های درون $\{0,1\}^*$ را به درستی بپذیرد یا رد کند.

به عنوان یک مثال، زبان PATH را می‌توان در زمان چندجمله‌ای پذیرفت. یک الگوریتم پذیرنده در زمان چندجمله‌ای، تأیید می‌کند که G یک گراف بدون جهت را کد می‌کند، تأیید می‌کند که u و v رأس‌هایی در G هستند، و سپس تعداد یال‌های کوتاه‌ترین مسیر را بین آن دو را با k مقایسه می‌کند. اگر G کد یک گراف بدون جهت باشد، و کوتاه‌ترین مسیر از u به v حداکثر k یال داشته باشد، آن گاه الگوریتم خروجی ۱ را بازگردانده و متوقف می‌شود. در غیر این صورت، الگوریتم برای همیشه اجرا می‌شود. با این حال، این الگوریتم PATH را تصمیم‌گیری نمی‌کند، چرا که برای نمونه‌هایی که در آن‌ها تعداد یال‌های کوتاه‌ترین مسیر از k بیشتر است، الگوریتم صریحاً ۰ را بازنمی‌گرداند. یک الگوریتم تصمیم‌گیری برای PATH باید رشته‌های دودویی را که به PATH تعلق ندارند صریحاً رد کند. برای یک مسئله‌ی تصمیم‌گیری مانند PATH، طراحی چنین الگوریتم تصمیم‌گیری بسیار ساده است: وقتی هیچ مسیری از u به v با حداکثر k یال وجود ندارد، به جای اجرا برای همیشه، الگوریتم ۰ را بازگردانده و متوقف می‌شود. (زمانی که کدگذاری ورودی صحیح نیست هم الگوریتم باید همین عکس‌العمل را نشان دهد.) برای مسائل دیگر، مانند مسئله‌ی توقف تورینگ، یک الگوریتم پذیرنده وجود دارد، ولی هیچ الگوریتم تصمیم‌گیرنده‌ای وجود ندارد.

به صورت غیر رسمی می‌توانیم یک *کلاس پیچیدگی* (complexity class) را به صورت مجموعه‌ای از زبان‌ها تعریف کنیم، که در آن عضویت به کمک یک *واحد پیچیدگی* (complexity measure) تعیین می‌شود، مانند زمان اجرای یک الگوریتم که تعیین می‌کند که آیا x به L تعلق دارد یا خیر. تعریف واقعی یک کلاس پیچیدگی تا حدودی فنی‌تر است.

با استفاده از این چارچوب وابسته به نظریه‌ی زبان‌ها، می‌توانیم یک تعریف متفاوت برای کلاس پیچیدگی P ارائه کنیم:

$P = \{L \in \{0,1\}^* :$ یک الگوریتم A وجود دارد که می‌تواند L را در زمان چندجمله‌ای تصمیم‌گیری کند $\}$

در واقع، P کلاس زبان‌های قابل پذیرش در زمان چندجمله‌ای هم هست.

قضیه‌ی ۳۴-۲

$P = \{L :$ توسط یک الگوریتم چندجمله‌ای پذیرفته می‌شود $\}$

اثبات از آن جایی که کلاس زبان‌های تصمیم‌پذیر با الگوریتم‌های چندجمله‌ای، زیرمجموعه‌ای است از کلاس زبان‌های قابل پذیرش با الگوریتم‌های چندجمله‌ای، فقط باید نشان دهیم که اگر L توسط یک الگوریتم چندجمله‌ای پذیرفته می‌شود، توسط یک الگوریتم چندجمله‌ای تصمیم‌گیری هم می‌شود. فرض کنید L زبان پذیرفته شده توسط یک الگوریتم چندجمله‌ای A باشد. از یک بحث «شبیه‌سازی» کلاسیک استفاده کرده و یک الگوریتم چندجمله‌ای دیگر A' می‌سازیم که L را تصمیم‌گیری می‌کند.

چون A زبان L را در زمان $O(n^k)$ می‌پذیرد، برای یک ثابت k، پس همچنین یک ثابت c وجود

دارد به طوری که A زبان L را در حداکثر $T = cn^k$ مرحله می‌پذیرد. برای هر رشته‌ی ورودی x، الگوریتم A' عملیات A را برای زمان T شبیه‌سازی می‌کند. در پایان زمان T، الگوریتم A' رفتار A را بررسی می‌کند. اگر A رشته‌ی x را پذیرفته باشد، آن گاه A' هم x را پذیرفته و ۱ را به خروجی می‌دهد. اگر A رشته‌ی x را نپذیرفته باشد، آن گاه A' رشته‌ی x را رد کرده و مقدار ۰ را به خروجی می‌دهد. سربار شبیه‌سازی A توسط A' بر روی زمان اجرا بیش از یک فاکتور چندجمله‌ای نیست، و بنابراین A' یک الگوریتم چندجمله‌ای است که L را تصمیم‌گیری می‌کند. ■

توجه کنید که اثبات قضیه‌ی ۳۴-۲ سازنده نیست. برای یک زبان داده شده‌ی $L \in P$، ممکن است ما کران زمانی الگوریتم A را که L را می‌پذیرد ندانیم. با این حال، می‌دانیم که چنین کرانی وجود دارد، و بنابراین می‌دانیم که یک الگوریتم A' وجود دارد که می‌تواند آن کران را چک کند، حتی با این که ممکن است نتوانیم A' را به سادگی بیابیم.

تمرین‌ها

۱-۱-۳۴ مسئله‌ی بهینه‌سازی LONGEST-PATH-LENGTH را به صورت یک رابطه تعریف کنید که هر نمونه از یک گراف بدون جهت و دو رأس در آن را به تعداد یال‌های بلندترین مسیر ساده بین دو رأس نسبت می‌دهد. سپس، مسئله‌ی تصمیم‌گیری

$$\text{LONGEST-PATH-LENGTH} = \left\{ \langle G, u, v, k \rangle \;\middle|\; \begin{array}{l} G = (V, E) \text{ یک گراف بدون جهت است، } u, v \in V \text{ و} \\ k \geq 0 \text{ یک عدد صحیح است، و یک مسیر ساده از } u \text{ به } v \\ \text{در } G \text{ با حداقل } k \text{ یال وجود دارد} \end{array} \right\}$$

را تعریف کنید. نشان دهید که مسئله‌ی بهینه‌سازی LONGEST-PATH-LENGTH را می‌توان در زمان چندجمله‌ای حل کرد اگر و فقط اگر $\text{LONGEST-PATH} \in P$.

۲-۱-۳۴ یک تعریف رسمی برای مسئله‌ی یافتن بلندترین دور ساده در یک گراف بدون جهت بدهید. یک مسئله‌ی تصمیم‌گیری متناظر برای این مسئله و زبان مربوط به آن را ارائه کنید.

۳-۱-۳۴ یک کدگذاری رسمی برای گراف‌های جهت‌دار به صورت رشته‌های دودویی با استفاده از نمایش ماتریس مجاورت بدهید. همین کار را برای نمایش نمایش لیست مجاورت بدهید. بحث کنید که این دو نمایش چندجمله‌ای-مرتبط هستند.

۴-۱-۳۴ آیا الگوریتم برنامه‌ریزی پویا برای مسئله‌ی کوله‌پشتی ۰-۱ که در تمرین ۱۶-۲-۲ خواسته شده است، یک الگوریتم چندجمله‌ای است؟ جواب خود را توضیح دهید.

۵-۱-۳۴ نشان دهید که اگر یک الگوریتم حداکثر تعداد ثابتی فراخوانی بر روی زیرروال‌هایی با زمان چندجمله‌ای انجام دهد و علاوه بر آن مقداری کار اضافه انجام دهد که آن هم به زمان چندجمله‌ای نیاز دارد، آن گاه این الگوریتم در زمان چندجمله‌ای اجرا می‌شود.

همچنین نشان دهید که اگر تعداد فراخوانی‌های زیرروال‌های چندجمله‌ای، خود از مرتبه‌ی چندجمله‌ای باشد، ممکن است الگوریتم حاصل نمایی باشد.

۳۴-۱-۶ نشان دهید که کلاس P، اگر به صورت مجموعه‌ای از زبان‌ها دیده شود، تحت اجتماع، اشتراک، اتصال، مکمل، و بستار، بسته است. یعنی، اگر $L_1, L_2 \in P$ آن گاه $L_1 \cup L_2 \in P$، $L_1 \cap L_2 \in P$، $L_1 L_2 \in P$، $\overline{L_1} \in P$، و $L_1^* \in P$.

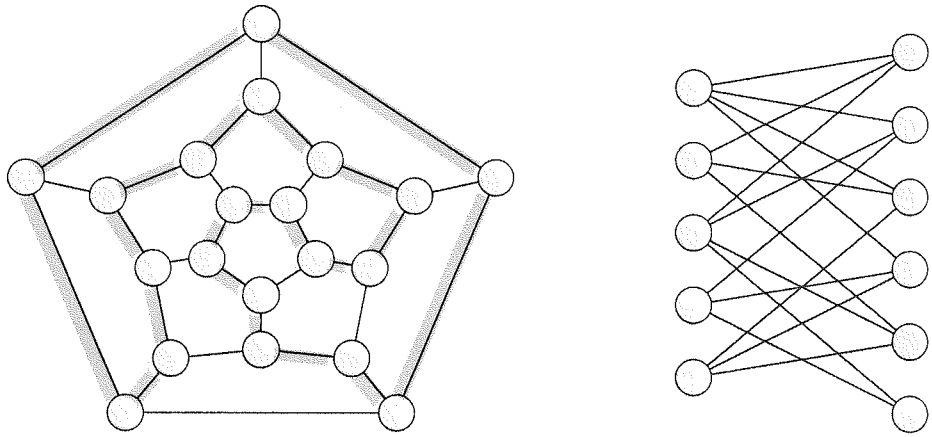
۳۴-۲ تحقیق در زمان چندجمله‌ای

اکنون نگاهی می‌اندازیم به الگوریتم‌هایی که عضویت در زبان‌ها را «تحقیق» می‌کنند. مثلاً، فرض کنید که برای یک نمونه‌ی داده شده‌ی $\langle G, u, v, k \rangle$ از مسئله‌ی تصمیم‌گیری PATH، به ما یک مسیر p از u به v هم داده شده است. به سادگی می‌توانیم چک کنیم که آیا p مسیری در G است یا خیر، و همچنین این که آیا طول p حداکثر k است یا خیر. اگر این گونه بود می‌توانیم p را به صورت یک «تصدیق» برای عضویت نمونه‌ها در PATH در نظر بگیریم. برای مسئله‌ی تصمیم‌گیری PATH، به نظر نمی‌آید که این تصدیق فایده‌ی زیادی داشته باشد. به هر حال PATH به کلاس P تعلق دارد – در واقع می‌توان PATH را در زمان خطی حل کرد – و بنابراین تحقیق عضویت از طریق یک تصدیق داده شده به همان اندازه زمان نیاز دارد که حل مسئله از ابتدا. اکنون مسئله‌ای را بررسی می‌کنیم که هنوز هیچ الگوریتم تصمیم‌گیری چندجمله‌ای برای آن یافت نشده است، ولی با داشتن یک تصدیق، تحقیق آن ساده است.

دورهای همیلتونی

مسئله‌ی یافتن دورهای همیلتونی در یک گراف بدون جهت برای بیش از صد سال مورد مطالعه قرار گرفته است. به صورت رسمی، یک *دور همیلتونی* (Hamiltonian cycle) یک گراف بدون جهت $G = (V, E)$ یک دور ساده است که شامل تمام رأس‌های V می‌شود. به گرافی که یک دور همیلتونی داشته باشد، یک گراف *همیلتونی*، و در غیر این صورت، به آن یک گراف *غیرهمیلتونی* گفته می‌شود. این نام‌گذاری به افتخار W. R. Hamilton انجام شده است که یک بازی ریاضی ارائه کرد که بر روی یک دوازده‌وجهی (شکل ۳۴-۲ (الف)) انجام می‌شد. در این بازی، یکی از بازی‌کنان باید پنج سوزن را در پنج رأس متوالی دلخواه قرار دهد، و بازی‌کن دیگر باید مسیر را طوری کامل کند که یک دور شامل تمام رأس‌ها تشکیل شود[1]. دوازده‌وجهی همیلتونی است، و شکل ۳۴-۲ (ب) یک دور

[1] Hamilton در نامه‌ای به تاریخ ۱۷ اکتبر ۱۸۵۶ به دوست خود John T. Graves می‌نویسد: «به تازگی متوجه شدم که یک بازی ریاضی جدید ذهن بعضی جوانان را مشغول کرده، که در آن یک نفر پنج سوزن در پنج نقطه‌ی پی‌درپی دلخواه فرو می‌کند ... و دیگری باید پانزده سوزن دیگر به ترتیب دایره‌ای اضافه کند به طوری که تمام نقاط دیگر را پوشش دهد، و در انتها، آخرین سوزنی که قرار می‌دهد مجاور سوزنی باشد که با آن شروع کرده است، که طبق نظریه‌ی ارائه شده در این نامه انجام آن همیشه ممکن است.



شکل ۳۴-۲ (الف) یک گراف نشان‌دهنده‌ی رأس‌ها، یال‌ها، و وجه‌های یک دوازده‌وجهی، به همراه یک دور همیلتونی که با یال‌های سایه‌دار نشان داده شده است. (ب) یک گراف دوبخشی با تعداد فردی رأس. هر گرافی مانند این، غیرهمیلتونی است.

همیلتونی آن را نشان می‌دهد. اما تمام گراف‌ها همیلتونی نیستند. به عنوان مثال، شکل ۳۴-۲ (ب) یک گراف دوبخشی را نشان می‌دهد که تعداد رأس‌های آن فرد است. تمرین ۳۴-۲-۲ از شما می‌خواهد نشان دهید که تمام گراف‌های بدین شکل غیرهمیلتونی هستند.

می‌توانیم *مسئله‌ی دور همیلتونی*، «آیا یک گراف G دور همیلتونی دارد؟»، را به صورت یک زبان فرمال بدین شکل تعریف کنیم:

$$\text{HAM-CYCLE} = \{\langle G \rangle : G \text{ یک گراف همیلتونی است}\}$$

یک الگوریتم چگونه می‌تواند زبان HAM-CYCLE را تصمیم‌گیری کند؟ با داشتن یک نمونه مسئله‌ی $\langle G \rangle$، یک الگوریتم تصمیم‌گیری ممکن این است که تمام جایگشت‌های ممکن رأس‌های G را لیست کنیم، و سپس چک کنیم که آیا بین آن‌ها یک مسیر همیلتونی وجود دارد یا خیر. زمان اجرای این الگوریتم چقدر است؟ اگر از کدگذاری «معقول» گراف به صورت ماتریس مجاورت آن استفاده کنیم، m، تعداد رأس‌های گراف، $\Omega(\sqrt{n})$ است، که در آن $n = |\langle G \rangle|$ طول کدگذاری G است. $m!$ جایگشت ممکن از رأس‌ها وجود دارد، و بنابراین زمان اجرا برابر است با $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$، که برای هیچ ثابت kای، $O(n^k)$ نیست. بنابراین، این الگوریتم ساده‌لوحانه در زمان چندجمله‌ای اجرا نمی‌شود. در واقع، مسئله‌ی دور همیلتونی NP-کامل است، که آن را در بخش ۳۴-۵ اثبات خواهیم کرد.

الگوریتم‌های تحقیق

یک مسئله‌ی تا حدودی ساده‌تر را در نظر بگیرید. فرض کنید که یک دوست به شما می‌گوید که یک

گراف داده شده‌ی G همیلتونی است، و سپس پیشنهاد می‌کند که با دادن رأس‌های یک دور همیلتونی به ترتیب حضور، این را به شما اثبات کند. مطمئنا تحقیق این اثبات به اندازه‌ی کافی ساده خواهد بود: به سادگی چک می‌کنیم که آیا این دور همیلتونی داده شده یک جایگشت از رأس‌های V هست یا نه، و سپس چک می‌کنیم که تمام یال‌های متوالی روی دور در گراف وجود داشته باشند. این الگوریتم تحقیق را می‌توان طوری پیاده‌سازی کرد که در زمان $O(n^2)$ اجرا شود، که در آن n طول کدگذاری G است. بنابراین، درستی یک اثبات برای همیلتونی بودن یک گراف را می‌توان در زمان چندجمله‌ای تحقیق کرد.

یک *الگوریتم تحقیق* (verification algorithm) را به صورت یک الگوریتم دو آرگومانی A تعریف می‌کنیم، که در آن یکی از آرگومان‌ها یک رشته‌ی ورودی معمولی x است، و دیگری یک رشته‌ی دودویی y با نام *تصدیق* (certificate). یک الگوریتم دو آرگومانی A یک رشته‌ی ورودی x را *تأیید* می‌کند اگر یک تصدیق y وجود داشته باشد به طوری که $A(x,y)=1$. زبان *تأیید* شده توسط یک الگوریتم تحقیق A عبارت است از

$$L=\{x\in\{0,1\}^*:\ \text{یک } y\in\{0,1\}^* \text{ وجود دارد به طوری که } A(x,y)=1\}$$

به صورت شهودی، یک الگوریتم A یک زبان L را تأیید می‌کند اگر برای هر رشته‌ی $x\in L$، یک تصدیق y وجود داشته باشد که A بتواند با استفاده از آن ثابت کند $x\in L$. به علاوه برای هر رشته‌ی $x\notin L$، باید هیچ تصدیقی وجود نداشته باشد که اثبات کند $x\in L$. برای مثال در مسئله‌ی دور همیلتونی، تصدیق، لیست رأس‌های دور همیلتونی است. اگر یک گراف همیلتونی باشد، خود دور همیلتونی اطلاعات کافی برای تأیید این حقیقت به دست می‌دهد. بالعکس، اگر یک گراف همیلتونی نباشد، هیچ لیستی از رأس‌ها وجود ندارد که بتواند الگوریتم تحقیق را فریب دهد تا الگوریتم تصمیم بگیرد که این گراف همیلتونی است، چرا که الگوریتم تحقیق با دقت «دور» ارائه شده را چک می‌کند تا اطمینان حاصل شود.

کلاس پیچیدگی NP

کلاس پیچیدگی NP عبارت است از کلاس زبان‌هایی که می‌توان آن‌ها را به کمک یک الگوریتم چندجمله‌ای تصدیق کرد.[1] به طور دقیق‌تر، یک زبان L به NP تعلق دارد اگر و فقط اگر یک الگوریتم چندجمله‌ای A با دو ورودی و همچنین یک ثابت c وجود داشته باشد به طوری که

$$L=\{x\in\{0,1\}^*:\ \text{یک تصدیق } y \text{ با } |y|=O(|x|^c) \text{ وجود دارد به طوری که } A(x,y)=1\}$$

می‌گوییم الگوریتم A زبان L را *در زمان چندجمله‌ای تأیید می‌کند*.

از بحث قبلی در مورد مسئله‌ی دور همیلتونی، نتیجه می‌شود که HAM-CYCLE $\in$ NP. (همیشه

---

[1] نام «NP» از «زمان چندجمله‌ای غیرقطعی» (nondeterministic polynomial time) می‌آید. کلاس NP در ابتدا در مبحث عدم قطعیت بررسی شد، ولی این کتاب از مفهوم تا حدودی ساده‌تر ولی متناظر تصدیق استفاده می‌کند. Hopcroft و Ullman یک نمایش مناسب از NP-کامل بودن در مدل محاسباتی عدم قطعیت ارائه می‌کنند.

دانستن این که یک مجموعه‌ی مهم، ناتهی است، آرامش‌بخش است!) به علاوه، اگر $L \in P$، آن گاه $L \in NP$، چرا که اگر یک الگوریتم چندجمله‌ای برای تصمیم L وجود داشته باشد، آن گاه می‌توان به آسانی آن الگوریتم را به یک الگوریتم تحقیق دو آرگومانی تبدیل کرد که به سادگی از تمام تصدیق‌ها صرف نظر می‌کند و دقیقاً آن رشته‌های ورودی را می‌پذیرد که در L هستند. بنابراین، $P \subseteq NP$.

هنوز کسی نمی‌داند که P=NP برقرار است یا خیر، ولی اکثر محققان معتقدند که کلاس‌های P و NP معادل یکدیگر نیستند. به طور شهودی کلاس P شامل مسائلی است که می‌توان آن‌ها را با سرعت بالایی حل کرد. کلاس NP شامل مسائلی است که می‌توان آن‌ها را به سرعت تصدیق کرد. ممکن است از تجربیات خود آموخته باشید که همیشه حل یک مسئله از ابتدا سخت‌تر از تحقیق یک جواب داده شده است، به خصوص وقتی که تحت محدودیت‌های زمانی کار می‌کنیم. نظریه‌پردازان علوم کامپیوتر عموما معتقدند که این قیاس برای کلاس‌های P و NP هم کاربرد دارد، و بنابراین NP شامل مسائل بیشتری از P است.

شواهد متقاعد کننده (هر چند نامطمئن) بیشتری برای $P \neq NP$–وجود زبان‌های «NP–کامل»– وجود دارد. در مورد کلاس NP–کامل در بخش ۳۴–۳ بحث خواهد شد.

فراتر از سؤال $P \neq NP$، بسیارند سؤال‌های پایه‌ای دیگر در همین زمینه که همچنان بی‌پاسخ باقی مانده‌اند. علی‌رغم کارهای فراوان انجام شده توسط محققان، هنوز هیچ کس حتی نمی‌داند که آیا کلاس NP تحت عملیات مکمل بسته است یا خیر. یعنی آیا $L \in NP$ نتیجه می‌دهد $\overline{L} \in NP$؟ می‌توانیم کلاس پیچیدگی co-NP را به صورت زبان‌های L تعریف کنیم که $\overline{L} \in NP$. این سؤال که آیا NP تحت عملیات مکمل بسته است را می‌توان به صورت این سؤال بازنویسی کرد: آیا NP با co-NP معادل است؟ از آن جایی که P تحت مکمل بسته است (تمرین ۳۴–۱–۶)، و از تمرین ۳۴–۲–۹ نتیجه می‌شود که $P \subseteq NP \cap co\text{-}NP$. اما دوباره، هنوز کسی نمی‌داند که آیا $P = NP \cap co\text{-}NP$ یا آیا یک زبان در $NP \cap co\text{-}NP - P$ وجود دارد یا خیر. شکل ۳۴–۳ چهار سناریوی ممکن را نشان می‌دهد.

بنابراین، درک ما از رابطه‌ی میان P و NP به شدت ناقص است. با این همه، هر چند ممکن است نتوانیم اثبات کنیم که یک مسئله رام‌نشدنی است، ولی در صورت اثبات NP–کامل بودن هم اطلاعات بسیار مفیدی در مورد آن کسب کرده‌ایم.

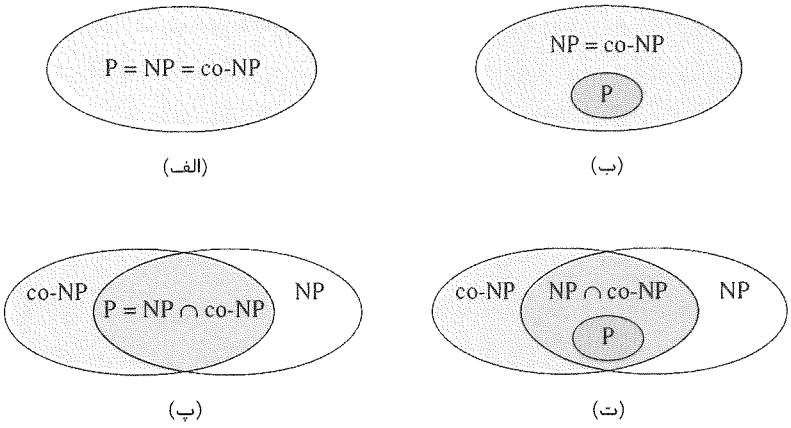
تمرین‌ها

۱–۲–۳۴ زبان زیر را در نظر بگیرید:

$$\text{GRAPH-ISOMORPHISM} = \{\langle G_1, G_2\rangle : G_1 \text{ و } G_2 \text{ گراف‌های هم‌ریخت هستند}\}$$

با توصیف یک الگوریتم چندجمله‌ای برای تحقیق این زبان، اثبات کنید که GRAPH-ISOMORPHISM $\in$ NP.

۲–۲–۳۴ اثبات کنید که اگر G یک گراف دوبخشی بدون جهت با تعداد فردی رأس باشد، آن گاه G غیرهمیلتونی است.



شکل ۳۴-۳ چهار سناریوی ممکن برای رابطه‌ی میان کلاس‌های پیچیدگی. در هر نمودار، یک ناحیه که ناحیه‌ی دیگر را در بر می‌گیرد نشان‌دهنده‌ی رابطه‌ی «زیرمجموعه‌ی اکید» است. (الف) P=NP=co-NP. اکثر محققان احتمال کمی به این حالت نسبت داده و آن را رد می‌کنند. (ب) اگر NP تحت عملیات مکمل بسته باشد، آن گاه co-NP برابر است با NP، ولی نیازی نیست که P=NP برقرار باشد. (پ) $P = NP \cap co\text{-}NP$، ولی NP تحت مکمل بسته نیست. (ت) $NP \neq co\text{-}NP$ و $P \neq NP \cap co\text{-}NP$. اکثر محققان احتمال کمی به این حالت نسبت داده و آن را رد می‌کنند.

۳-۲-۳۴ نشان دهید که اگر HAM-CYCLE $\in P$، آن گاه مسئله‌ی لیست کردن رأس‌های یک دور همیلتونی به ترتیب، قابل حل در زمان چندجمله‌ای است.

۴-۲-۳۴ اثبات کنید که کلاس NP از زبان‌ها تحت عملیات اجتماع، اشتراک، اتصال، و بستار، بسته است. در مورد بسته بودن NP تحت مکمل بحث کنید.

۵-۲-۳۴ نشان دهید که هر زبان در NP را می‌توان با یک الگوریتم که در زمان $2^{O(n^k)}$ اجرا می‌شود، برای یک ثابت k، تصمیم‌گیری کرد.

۶-۲-۳۴ یک *مسیر همیلتونی* در یک گراف یک مسیر ساده است که تمام رأس‌ها را دقیقاً یک بار ملاقات می‌کند. نشان دهید که زبان زیر متعلق به NP است.

$$\text{HAM-PATH} = \{\langle G, u, v\rangle : \text{یک مسیر همیلتونی از } u \text{ به } v \text{ در گراف } G \text{ وجود دارد}\}$$

۷-۲-۳۴ نشان دهید که مسئله‌ی مسیر همیلتونی (تمرین ۳۴-۲-۶) را می‌توان روی گراف‌های جهت‌دار بدون دور در زمان چندجمله‌ای حل کرد. یک الگوریتم کارامد برای این مسئله ارائه کنید.

۸-۲-۳۴ فرض کنید φ یک فرمول بولین باشد که از متغیرهای ورودی‌های بولین $x_1, x_2, \ldots, x_k$، اعمال AND (∧)، OR (∨)، نفی (¬)، و پرانتزها ساخته شده است. فرمول φ *درست‌نما*

(tautology) است اگر مقدار خروجی آن برای هر نوع مقداردهی ۰ و ۱ به ورودی‌ها، ۱ باشد. TAUTOLOGY را به صورت یک زبان از فرمول‌های دودویی تعریف کنید که درست‌نما هستند. نشان دهید که TAUTOLOGY $\in$ co-NP.

۳۴-۲-۹ اثبات کنید که P $\subseteq$ co-NP.

۳۴-۲-۱۰ اثبات کنید که اگر NP $\neq$ co-NP، آن گاه P $\neq$ NP .

۳۴-۲-۱۱ فرض کنید G یک گراف بدون جهت همبند با حداقل ۳ رأس باشد، و فرض کنید G^3 گراف حاصل از اتصال تمام رأس‌هایی باشد که در G با مسیری شامل حداکثر ۳ یال به هم مرتبط‌اند. اثبات کنید که G^3 همیلتونی است. (*راهنمایی:* یک درخت پوشا برای G بسازید، و از یک بحث استقرایی استفاده کنید.)

۳۴-۳ NP-کامل‌ها و قابلیت کاهش

احتمالاً مهم‌ترین دلیلی که نظریه‌پردازان علوم کامپیوتر را متقاعد کرده که P $\neq$ NP، وجود کلاس مسائل «NP-کامل» است. این کلاس این خصوصیت غیرمنتظره را دارد که اگر یک مسئله‌ی NP-کامل را بتوان در زمان چندجمله‌ای حل کرد، آن گاه *تمام* مسائل در NP یک جواب چندجمله‌ای دارند، یعنی P=NP. با این حال، علی‌رغم سال‌های زیاد تحقیق هنوز هیچ الگوریتم چندجمله‌ای برای هیچ یک از مسائل NP-کامل یافت نشده است.

زبان HAM-CYCLE یک مسئله‌ی NP-کامل است. اگر بتوانیم HAM-CYCLE را در زمان چندجمله‌ای تصمیم‌گیری کنیم، آن گاه می‌توانیم تمام مسائل NP را در زمان چندجمله‌ای حل کنیم. در واقع، اگر NP $-$ P غیرتهی باشد، می‌توانیم با اطمینان بگوییم که HAM-CYCLE $\in$ NP $-$ P.

از جهتی، زبان‌های NP-کامل «سخت‌ترین» زبان‌های NP هستند. در این بخش نشان خواهیم داد که چگونه می‌توان «سختی» نسبی زبان‌ها را با استفاده از یک مفهوم دقیق به نام «قابلیت کاهش در زمان چندجمله‌ای» مقایسه کرد. سپس زبان‌های NP-کامل را به صورت رسمی تعریف می‌کنیم، و با ارائه‌ی اثبات NP-کامل بودن یکی از این زبان‌ها به نام CIRCUIT-SAT، کار خود را به پایان می‌بریم. در بخش‌های ۳۴-۴ و ۳۴-۵، از مفهوم قابلیت کاهش استفاده کرده و NP-کامل بودن بسیاری از زبان‌های دیگر را اثبات می‌کنیم.

کاهش‌پذیری

به طور شهودی، یک مسئله‌ی Q را می‌توان به یک مسئله‌ی دیگر Q' کاهش داد اگر بتوان هر نمونه از Q را به سادگی به صورت یک نمونه از Q' «تغییر شکل» داد، به طوری که جواب آن، یک جواب

برای نمونه‌ی متناظر Q هم باشد. به عنوان مثال، مسئله‌ی حل کردن معادلات خطی نسبت به مجهول x به مسئله‌ی حل معادلات درجه‌ی دو کاهش می‌یابد. با داشتن یک نمونه‌ی $ax + b = 0$، آن را به $0x^2 + ax + b = 0$ تبدیل می‌کنیم، که جواب آن یک جواب برای $ax + b = 0$ هم فراهم می‌کند. بنابراین اگر یک مسئله‌ی Q به یک مسئله‌ی Q' کاهش یابد، در این صورت، می‌توان گفت Q «سخت‌تر» از Q' نیست.

با بازگشت به چارچوب زبان‌های فرمال برای مسائل تصمیم‌گیری، می‌گوییم یک زبان L_1 *کاهش‌پذیر در زمان چندجمله‌ای* (polynomial-time reducible) به زبان L_2 است، و آن را به صورت $L_1 \leq_P L_2$ می‌نویسیم، اگر یک تابع محاسبه‌پذیر در زمان چندجمله‌ای $f : \{0,1\}^* \to \{0,1\}^*$ وجود داشته باشد به طوری که برای تمام $x \in \{0,1\}^*$ داشته باشیم

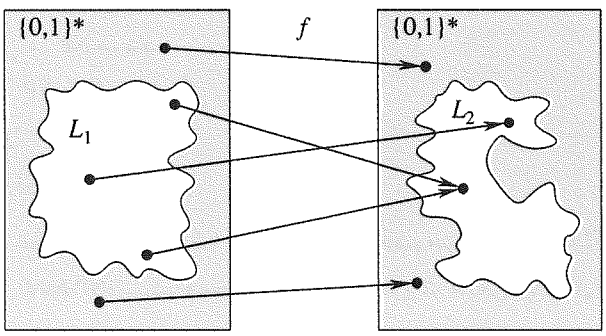
$$x \in L_1 \text{ اگر و فقط اگر } f(x) \in L_2 \qquad (۱-۳۴)$$

به تابع f، *تابع کاهش* (reduction function)، و به الگوریتم چندجمله‌ای F که f را محاسبه می‌کند، *الگوریتم کاهش* (reduction algorithm) می‌گوییم.

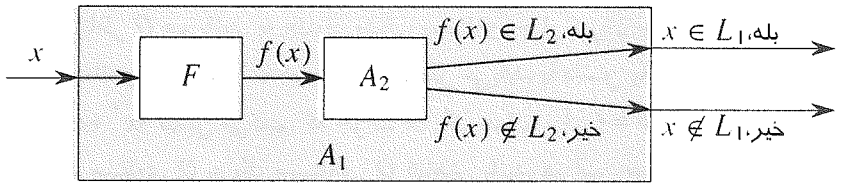
شکل ۳۴-۴ ایده‌ی یک کاهش چندجمله‌ای از زبان L_1 به یک زبان دیگر L_2 را نشان می‌دهد. هر زبان زیرمجموعه‌ای از $\{0,1\}^*$ است. تابع کاهش f یک نگاشت با زمان چندجمله‌ای ارائه می‌کند که اگر $x \in L_1$، آن گاه $f(x) \in L_2$. به علاوه اگر $x \notin L_1$، آن گاه $f(x) \notin L_2$. بنابراین تابع کاهش هر نمونه‌ی x از مسئله‌ی تصمیم‌گیری نشان داده شده توسط L_1 را به یک نمونه‌ی $f(x)$ از مسئله‌ی نشان داده شده توسط L_2 نگاشت می‌کند. ارائه‌ی یک جواب که تعیین می‌کند $f(x) \in L_2$ هست یا نه، درستی $x \in L_1$ را هم مشخص می‌کند.

کاهش‌های چندجمله‌ای به ما یک ابزار قدرتمند برای اثبات تعلق بسیاری از زبان‌ها به P می‌دهند.

لم ۳-۳۴ اگر $L_1, L_2 \subseteq \{0,1\}^*$ زبان‌هایی باشد به طوری که $L_1 \leq_P L_2$، آن گاه $L_2 \in P$ نتیجه می‌دهد $L_1 \in P$.



شکل ۳۴-۴ یک نمایش از یک کاهش چندجمله‌ای از یک زبان L_1 به یک زبان L_2 با استفاده از تابع کاهش f. برای هر ورودی $x \in \{0,1\}^*$، جواب $x \in L_1$ معادل است با جواب $f(x) \in L_2$.



شکل ۳۴-۵ اثبات لم ۳۴-۳. الگوریتم F یک الگوریتم کاهش است که تابع کاهش f از L_1 به L_2 را در زمان چندجمله‌ای محاسبه می‌کند، و A_2 یک الگوریتم با زمان چندجمله‌ای است که L_2 را تصمیم‌گیری می‌کند. الگوریتم A_1 برای تعیین درستی عبارت $x \in L_1$ بدین صورت عمل می‌کند: ابتدا با استفاده از F ورودی x را به $f(x)$ تبدیل می‌کند، و سپس با استفاده از A_2 صحت $f(x) \in L_2$ را تحقیق می‌کند.

اثبات فرض کنید A_2 یک الگوریتم چندجمله‌ای باشد که L_2 را تصمیم‌گیری می‌کند، و F یک الگوریتم کاهش چندجمله‌ای باشد که تابع کاهش f را محاسبه می‌کند. یک الگوریتم چندجمله‌ای A_1 خواهیم ساخت که L_1 را تصمیم‌گیری می‌کند.

شکل ۳۴-۵ ساختن A_1 را نشان می‌دهد. برای یک ورودی داده شده‌ی $x \in \{0,1\}^*$، الگوریتم A_1 از F برای تبدیل x به $f(x)$، و سپس از A_2 برای تست کردن درستی $f(x) \in L_2$ استفاده می‌کند. الگوریتم A_1 خروجی تولید شده توسط الگوریتم A_2 را به عنوان خروجی خود ارائه می‌کند.

درستی A_1 از شرط (۳۴-۱) نتیجه می‌شود. الگوریتم در زمان چندجمله‌ای اجرا می‌شود، چرا که هر دوی F و A_2 چندجمله‌ای هستند (تمرین ۳۴-۱-۵ را ببینید). ■

NP-کامل بودن

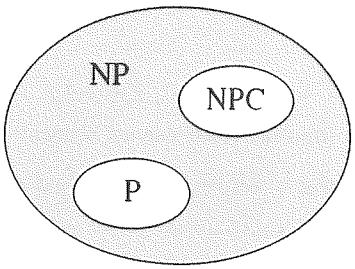
کاهش چندجمله‌ای یک روش فرمال فراهم می‌کند برای نشان دادن این که یک مسئله حداقل به سختی دیگری است، در محدوده‌ی یک فاکتور چندجمله‌ای. یعنی، اگر $L_1 \leq_P L_2$، آن گاه L_1 بیش از یک فاکتور چندجمله‌ای سخت‌تر از L_2 نیست، که به همین دلیل است که نماد «کوچک‌تر یا مساوی» برای به خاطرسپاری کاهش مناسب است. اکنون می‌توانیم کلاس زبان‌های NP-کامل را تعریف کنیم، که سخت‌ترین مسائل در NP هستند.

یک زبان $L \subseteq \{0,1\}^*$ NP-کامل است اگر

۱. $L \in \text{NP}$، و
۲. $L' \leq_P L$ برای هر $L' \in \text{NP}$.

اگر یک زبان L خصوصیت ۲ را ارضا کند، ولی لزوماً خصوصیت ۱ را ارضا نکند، می‌گوییم L یک زبان *NP-سخت* (NP-hard) است. همچنین NPC را به صورت کلاس زبان‌های NP-کامل تعریف می‌کنیم.

همان طور که قضیه‌ی زیر نشان می‌دهد، NP-کامل بودن معمای تساوی P با NP است.



شکل ۳۴-۶ باور اکثر نظریه‌پردازان علوم کامپیوتر در مورد رابطه‌ی میان P، NP، و NPC. هر دوی P و NPC به طور کامل درون NP قرار دارند، و $P \cap NPC = \emptyset$.

قضیه‌ی ۳۴-۶ اگر یک مسئله‌ی NP–کامل قابل حل در زمان چندجمله‌ای باشد، آن گاه P=NP. به طور معادل، اگر یک مسئله در NP قابل حل در زمان چندجمله‌ای نباشد، آن گاه هیچ مسئله‌ی NP–کاملی قابل حل در زمان چندجمله‌ای نیست.

اثبات فرض کنید $L \in P$ و همچنین $L \in NPC$. برای هر $L' \in NP$ ، داریم $L' \leq_P L$ طبق خصوصیت ۲ از تعریف NP–کامل بودن. بنابراین طبق لم ۳۴-۳، همچنین داریم $L' \in P$ ، که بخش اول قضیه را اثبات می‌کند.

برای اثبات بخش دوم، توجه کنید که این بخش، عکس نقیض بخش اول است. ■

به همین دلیل است که تحقیق در مورد $P \neq NP$ حول محور مسائل NP–کامل می‌گردد. اکثر نظریه‌پردازان علوم کامپیوتر معتقدند که $P \neq NP$، که به رابطه‌ی نشان داده شده در شکل ۳۴-۶ میان P، NP، و NPC منجر می‌شود.

ولی تا آن جایی که می‌دانیم، ممکن است یک نفر یک جواب چندجمله‌ای برای یک مسئله‌ی NP–کامل بیابد، و اثبات کند که P=NP. به هر حال، چون هنوز هیچ الگوریتم چندجمله‌ای برای هیچ یک از مسائل NP–کامل یافت نشده است، اثباتی که نشان دهد یک مسئله NP–کامل است، شاهد خوبی است برای رام‌نشدنی بودن آن مسئله.

قابلیت ارضای مدار

تا این جا مفهوم یک مسئله‌ی NP–کامل را تعریف کرده‌ایم، ولی هنوز NP–کامل بودن هیچ مسئله‌ای را اثبات نکرده‌ایم. وقتی اثبات کردیم که حداقل یک مسئله NP–کامل است، می‌توانیم با استفاده از کاهش چندجمله‌ای به عنوان یک ابزار، NP–کامل بودن بقیه‌ی مسائل را اثبات کنیم. بنابراین، اکنون بر روی نشان دادن وجود یک مسئله‌ی NP–کامل تمرکز می‌کنیم: مسئله‌ی قابلیت ارضای مدار.

متأسفانه، اثبات رسمی NP–کامل بودن مسئله‌ی ارضای مدار به جزئیات فنی فراتر از حوصله‌ی این کتاب نیاز دارد. در عوض، یک اثبات غیررسمی ارائه خواهیم کرد که بر پایه‌ی فهم اساسی از مدارهای

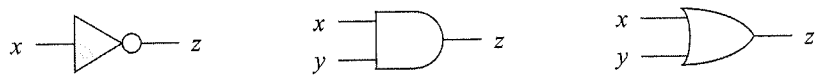
ترکیبی بولین است.

مدارهای ترکیبی بولین از عناصر ترکیبی بولین ساخته می‌شوند که به وسیله‌ی سیم به یکدیگر متصل‌اند. یک *عنصر ترکیبی بولین* (Boolean combinational element) یک عنصر مدار است که تعداد ثابتی ورودی و خروجی بولین دارد و یک تابع خوش‌تعریف را پیاده‌سازی می‌کند. مقادیر بولین از مجموعه‌ی {۰,۱} استخراج می‌شوند، که در آن ۰ نشان‌دهنده‌ی FALSE (نادرست) است، و ۱ نشان‌دهنده‌ی TRUE (درست).

عناصر ترکیبی بولینی که ما از آن‌ها در مسئله‌ی قابلیت ارضای مدار استفاده می‌کنیم، هر یک یک تابع ساده‌ی بولین را محاسبه می‌کنند، و به نام *گیت‌های منطقی* (logic gates) معروفند. شکل ۳۴-۷ سه گیت منطقی اصلی را که ما از آن‌ها در مسئله‌ی قابلیت ارضای مدار استفاده می‌کنیم نشان می‌دهند: *گیت NOT* (یا *وارونگر* (inverter))، *گیت AND*، و *گیت OR*. گیت NOT یک تک *ورودی* دودویی x را دریافت می‌کند، که مقدار آن یا ۰ است و یا ۱، و یک *خروجی* دودویی z را تولید می‌کند، که مقدار آن عکس مقدار ورودی است. هر یک از دو گیت دیگر، دو ورودی دودویی x و y را دریافت می‌کنند، و یک خروجی z تولید می‌کنند.

عملیات هر گیت، و عملیات هر عنصر ترکیبی بولین را می‌توان با استفاده از یک *جدول درستی* (truth table) توصیف کرد، که در شکل ۳۴-۷ زیر هر گیت کشیده شده است. یک جدول درستی خروجی عنصر ترکیبی را برای هر یک از مجموعه ورودی‌های ممکن به دست می‌دهد. به عنوان مثال، جدول درستی گیت OR به ما می‌گوید که وقتی ورودی‌ها به صورت $x = ۰$ و $y = ۱$ هستند، مقدار خروجی $z = ۱$ است. از سمبل $\neg$ برای نشان دادن تابع NOT، از $\wedge$ برای نشان دادن تابع AND، و از $\vee$ برای نشان دادن تابع OR استفاده می‌کنیم. برای مثال، $۰ \vee ۱ = ۱$.

می‌توانیم گیت‌های AND و OR را طوری گسترش دهیم که بیش از دو ورودی دریافت کنند. خروجی یک گیت AND در صورتی ۱ است اگر تمام ورودی‌های آن ۱ باشند، و در غیر این صورت خروجی آن ۰ است، و خروجی یک گیت OR در صورتی ۱ است که حداقل یکی از ورودی‌های آن ۱



x	$\neg x$
0	1
1	0

(الف)

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

(ب)

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

(پ)

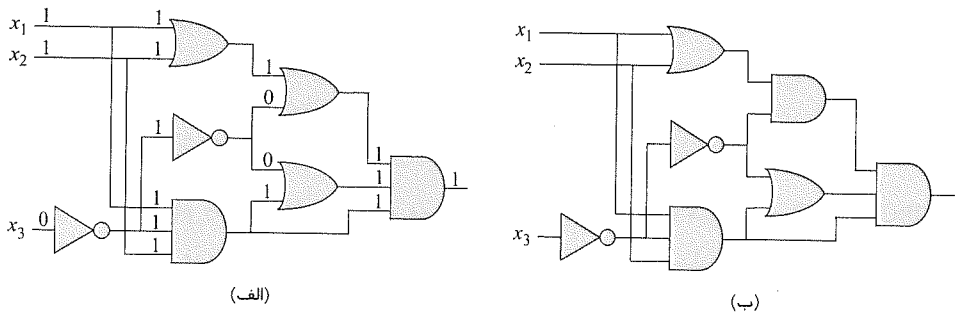
شکل ۳۴-۷ سه گیت منطقی اصلی، با ورودی‌ها و خروجی‌های دودویی. زیر هر گیت، جدول درستی توصیف‌کننده‌ی عملیات آن گیت قرار دارد. (الف) گیت NOT. (ب) گیت AND. (پ) گیت OR.

باشد، و در غیر این صورت خروجی آن ۰ است.

یک *مدار ترکیبی بولین* (Boolean combinational circuit) از یک یا چند عنصر ترکیبی بولین تشکیل شده است که با *سیم* (wire) به یکدیگر متصل شده‌اند. یک سیم می‌تواند خروجی یک عنصر را به ورودی دیگری متصل کند، که مقدار خروجی عنصر اول را به ورودی دومی می‌دهد. شکل ۳۴-۸ دو مدار ترکیبی بولین مشابه را نشان می‌دهد؛ آن‌ها فقط در یک گیت با هم متفاوت هستند. بخش (الف) شکل مقدار سیم‌های مختلف را هم نشان می‌دهد، که مربوط به ورودی $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$ است. با این که یک سیم نمی‌تواند خروجی بیش از یک عنصر را حمل کند، ولی می‌تواند ورودی چندین عنصر را تغذیه کند. به تعداد عناصری که توسط یک سیم تغذیه می‌شوند، *گنجایش خروجی* (fan-out) سیم گفته می‌شود. اگر خروجی هیچ عنصری به سیم متصل نشده باشد، آن سیم یک *ورودی مدار* (circuit input) است، که مقادیر ورودی را از یک منبع خروجی دریافت می‌کند. اگر ورودی هیچ عنصری به یک مدار متصل نباشد، آن سیم یک *خروجی مدار* (circuit output) است، که نتیجه‌ی محاسبات مدار را برای دنیای خارج فراهم می‌کند. (یک سیم داخلی هم می‌تواند به یک خروجی مدار متصل باشد.) برای تعریف مسئله‌ی قابلیت ارضای مدار تعداد خروجی‌های مدار را به ۱ محدود می‌کنیم، ولی در طراحی سخت‌افزار، در عمل یک مدار می‌تواند چندین خروجی داشته باشد.

مدارهای ترکیبی بولین هیچ دوری ندارند. به عبارت دیگر، فرض کنید که یک گراف جهت‌دار $G = (V, E)$ می‌سازیم، با یک رأس برای هر عنصر ترکیبی، و k یال جهت‌دار برای هر سیم با گنجایش خروجی k؛ یک یال جهت‌دار (u, v) خواهیم داشت اگر یک سیم خروجی عنصر u را به یک ورودی عنصر v متصل کند. در این صورت G باید بدون دور باشد.

یک *مقداردهی صحیح* (truth assignment) برای یک مدار ترکیبی، مجموعه‌ای از مقادیر ورودی بولین است. می‌گوییم یک مدار ترکیبی بولین با یک خروجی، *قابل ارضا* (satisfiable) است اگر یک *مقداردهی ارضا کننده* (satisfying assignment) داشته باشد: یک مقداردهی صحیح که خروجی مدار را ۱ می‌کند. برای مثال، مدار شکل ۳۴-۸ (الف) دارای مقداردهی ارضا کننده‌ی $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$،



شکل ۳۴-۸ دو نمونه از مسئله‌ی قابلیت ارضای مدار. (الف) مقداردهی $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$ به ورودی‌های این مدار، خروجی را ۱ می‌کند. بنابراین مدار قابل ارضا است. (ب) هیچ مقداردهی به ورودی‌ها نمی‌تواند خروجی این مدار را ۱ کند، بنابراین مدار غیرقابل ارضا است.

و بنابراین قابل ارضا است. همان طور که تمرین ۳۴-۳-۱ از شما می‌خواهد نشان دهید، هیچ مقداردهی به مقادیر x_1، x_2، و x_3 خروجی مدار شکل ۳۴-۸ (ب) را ۱ نمی‌کند؛ این مدار همیشه مقدار ۰ را به خروجی می‌دهد، و بنابراین غیرقابل ارضا است.

مسئله‌ی قابلیت ارضای مدار (circuit-satisfiability problem) عبارت است از این سؤال: «آیا یک مدار ترکیبی بولین داده شده متشکل از گیت‌های AND، OR، و NOT، قابل ارضا است؟». با این حال برای توصیف این مسئله به صورت رسمی، باید بر روی یک کدگذاری استاندارد برای مدارها توافق کنیم. *اندازه‌ی* (size) یک مدار ترکیبی بولین برابر است با تعداد عناصر ترکیبی بولین آن، به علاوه‌ی تعداد سیم‌ها در مدار. می‌توان یک کدگذاری گراف مانند ارائه کرد که هر مدار C را به یک رشته‌ی دودویی $\langle C \rangle$ نگاشت می‌کند، که طول آن نسبت به اندازه‌ی مدار از مرتبه‌ی چندجمله‌ای است. بنابراین به صورت یک زبان فرمال می‌توانیم تعریف کنیم

$$\text{CIRCUIT-SAT} = \{\langle C \rangle : C \text{ یک مدار ترکیبی دودویی قابل ارضا است}\}$$

مسئله‌ی قابلیت ارضای مدار در زمینه‌ی بهینه‌سازی سخت‌افزار به کمک کامپیوتر پیش می‌آید. اگر یک زیرمدار همیشه ۰ را تولید کند، آن گاه وجود آن غیرضروری است؛ به سادگی می‌توان آن را با یک زیرمدار دیگر جایگزین کرد که از تمام گیت‌ها صرف نظر کرده و همیشه ثابت ۰ را به خروجی می‌دهد. داشتن یک الگوریتم چندجمله‌ای برای این مسئله بسیار کمک‌کننده خواهد بود.

ممکن است به این فکر بیفتیم با آزمایش تمام حالت‌های مختلف برای ورودی‌های یک مدار C، قابلیت ارضای آن را تعیین کنیم. متأسفانه اگر k ورودی داشته باشیم، 2^k مقداردهی مختلف برای ورودی‌ها وجود خواهد داشت. اگر اندازه‌ی C نسبت به k از مرتبه‌ی چندجمله‌ای باشد، چک‌کردن هر یک ($\Omega(2^k)$ زمان خواهد گرفت، که نسبت به اندازه‌ی مدار از درجه‌ی فراچندجمله‌ای است.[1] در واقع، همان طور که قبلاً گفته شد، یک شاهد قوی وجود دارد برای این که هیچ الگوریتم چندجمله‌ای وجود ندارد که مسئله‌ی قابلیت ارضای مدار را حل کند، چرا که این مسئله NP-کامل است. اثبات این مسئله را به دو بخش تقسیم می‌کنیم، بر مبنای دو بخش تعریف NP-کامل بودن.

لم ۳۴.۸ مسئله‌ی قابلیت ارضای مدار به کلاس NP تعلق دارد.

اثبات یک الگوریتم چندجمله‌ای A با دو ورودی ارائه خواهیم کرد که می‌تواند CIRCUIT-SAT را تحقیق کند. یکی از ورودی‌های A (یک کدگذاری استاندارد از) یک مدار ترکیبی بولین C است.

---

[1] از طرف دیگر اگر اندازه‌ی مدار $\theta(2^k)$ باشد، آن گاه زمان اجرای یک الگوریتم که در زمان $O(2^k)$ اجرا شود نسبت به اندازه‌ی مدار از مرتبه‌ی چندجمله‌ای است. حتی اگر $P \neq NP$، این موقعیت با NP-کامل بودن این مسئله تناقض ندارد؛ وجود یک الگوریتم چندجمله‌ای برای یک حالت خاص نتیجه نمی‌دهد که یک الگوریتم چندجمله‌ای برای تمام حالت‌ها وجود دارد.

ورودی دیگر، تصدیق مربوط به یک مقداردهی مقادیر بولین به سیم‌های C است. (تمرین ۳۴-۳-۴ را برای یک تصدیق کوچک‌تر ببینید.)

الگوریتم A به صورت زیر ساخته می‌شود. برای هر گیت منطقی در مدار، این الگوریتم چک می‌کند که مقدار فراهم شده توسط تصدیق برای سیم خروجی، به درستی به صورت تابعی از مقادیر روی سیم‌های ورودی محاسبه شده است یا خیر. آن گاه اگر خروجی تمام مدار ۱ باشد، الگوریتم ۱ را بازمی‌گرداند، چرا که مقدار داده شده به ورودی‌های C یک مقداردهی ارضا کننده است. در غیر این صورت، A مقدار ۰ را بازمی‌گرداند.

هر گاه یک مدار قابل ارضای C به ورودی الگوریتم A داده شود، یک تصدیق وجود دارد که طول آن نسبت به اندازه‌ی C از مرتبه‌ی چندجمله‌ای است، و باعث می‌شود که A مقدار ۱ را به خروجی بدهد. هر گاه ورودی یک مدار غیرقابل ارضا باشد، هیچ تصدیقی نمی‌تواند A را فریب دهد که باور کند مدار قابل ارضا است. الگوریتم A در زمان چندجمله‌ای اجرا می‌شود: با یک پیاده‌سازی خوب، زمان خطی کافی است. بنابراین CIRCUIT-SAT را می‌توان در زمان چندجمله‌ای تحقیق کرد، و $\text{CIRCUIT-SAT} \in \text{NP}$. ▩

در بخش دوم در اثبات NP–کامل بودن CIRCUIT-SAT باید نشان دهیم زبان آن NP–سخت است. یعنی باید نشان دهیم که هر زبانی در NP در زمان چندجمله‌ای کاهش‌پذیر به CIRCUIT-SAT است. اثبات واقعی برای این مسئله مملو است از ریزه‌کاری‌ها، و بنابراین ما سعی می‌کنیم یک شمای کلی از اثبات بر مبنای مفاهیمی از سخت‌افزار کامپیوتر ارائه کنیم.

یک برنامه‌ی کامپیوتری در حافظه‌ی کامپیوتر به صورت دنباله‌ای از دستورالعمل‌ها ذخیره می‌شود. یک دستورالعمل معمولی یک عملیات، آدرس عملوندها در حافظه، و یک آدرس برای ذخیره‌ی خروجی را در خود کد می‌کند. یک مکان خاص در حافظه، با نام *شمارنده‌ی برنامه* (program counter)، شماره‌ی دستورالعمل بعدی که باید اجرا شود را ذخیره می‌کند. هر گاه یک دستورالعمل استخراج می‌شود، شمارنده‌ی برنامه به صورت خودکار یکی افزایش می‌یابد، و باعث می‌شود که کامپیوتر دستورالعمل‌ها را به صورت متوالی اجرا کند. از طرفی یک دستورالعمل قادر است مقدار شمارنده‌ی برنامه را به دلخواه تغییر دهد، که این کار با دست‌کاری روند اجرای متوالی دستورالعمل‌ها به کامپیوتر قابلیت اجرای حلقه‌ها و انشعاب‌های شرطی را می‌دهد.

در هر لحظه حین اجرای یک برنامه، کل وضعیت کامپیوتر توسط حافظه‌ی کامپیوتر مشخص می‌شود. (فرض می‌کنیم حافظه حاوی تمامی این موارد است: خود برنامه، شمارنده‌ی برنامه، اطلاعات عملیاتی ذخیره شده، و هر بیت ذخیره شده‌ی دیگری که کامپیوتر از آن برای حفظ موقعیت فعلی استفاده می‌کند.) به هر وضعیت از از حافظه‌ی کامپیوتر یک *پیکربندی* (configuration) می‌گوییم. می‌توان به اجرای یک دستورالعمل به صورت نگاشت یک پیکربندی به پیکربندی دیگر نگاه کنیم. می‌توان سخت‌افزاری را که این نگاشت را انجام می‌دهد به صورت یک مدار ترکیبی بولین پیاده‌سازی کرد. در اثبات لم زیر این سخت‌افزار را با M نشان می‌دهیم.

لم ۳۴-۶ مسئله‌ی قابلیت ارضای مدار NP-سخت است.

اثبات فرض کنید L یک زبان در NP باشد. یک الگوریتم چندجمله‌ای F را توصیف خواهیم کرد که یک کاهش f را محاسبه می‌کند که تمام رشته‌های دودویی x را به یک مدار $C = f(x)$ نگاشت می‌کند، به طوری که $x \in L$ اگر و فقط اگر $C \in$ CIRCUIT-SAT.

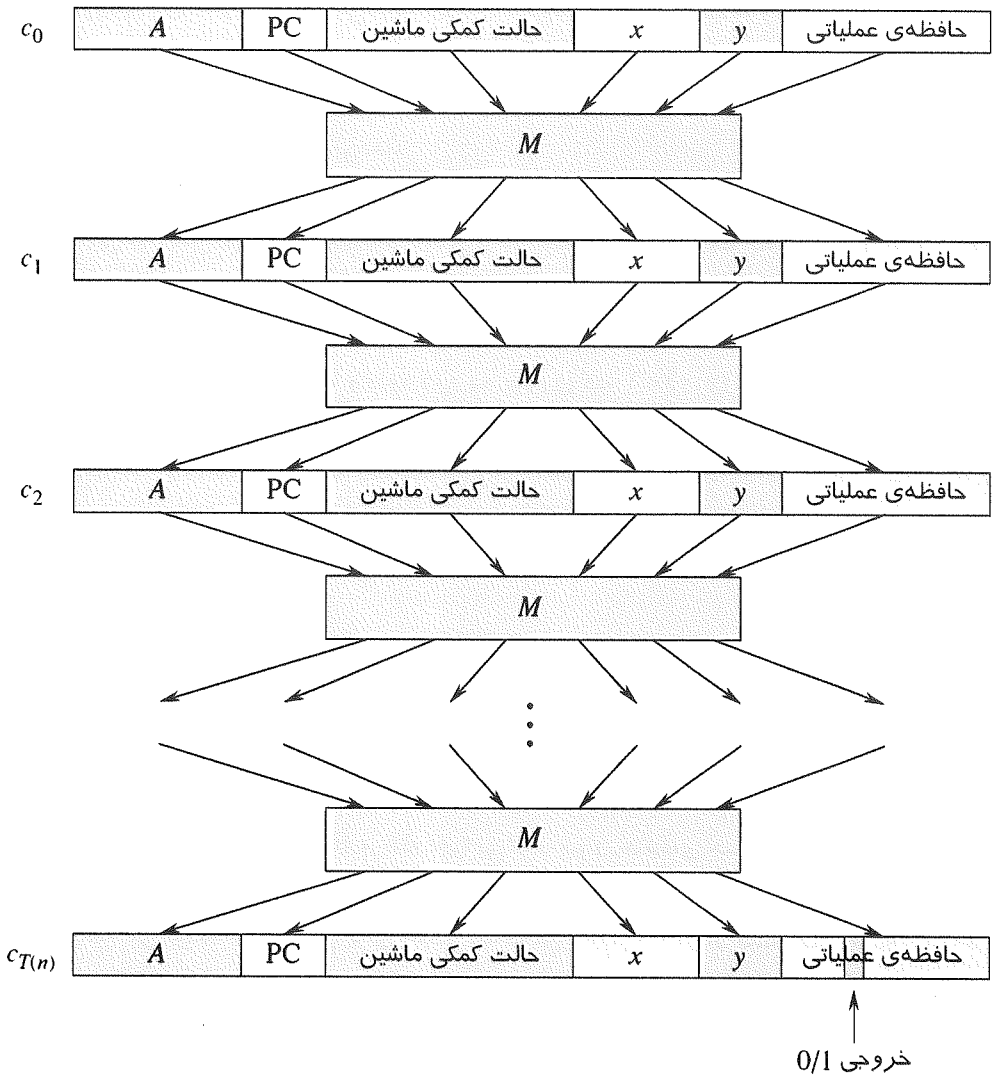
چون $L \in$ NP، باید یک الگوریتم A وجود داشته باشد که L را در زمان چندجمله‌ای تحقیق می‌کند. الگوریتم F که در ادامه توصیف خواهیم کرد، از الگوریتم دو ورودی A برای محاسبه‌ی کاهش f استفاده خواهد کرد.

فرض کنید $T(n)$ نشان‌دهنده‌ی بدترین حالت زمان اجرای الگوریتم A بر روی رشته‌های به طول n باشد، و فرض کنید $k \geq 1$ یک ثابت باشد به طوری که $T(n) = O(n^k)$ و طول تصدیق $O(n^k)$ باشد. (زمان اجرای A در واقع یک چندجمله‌ای نسبت به کل اندازه‌ی ورودی است، که هم شامل رشته‌ی ورودی است و هم شامل تصدیق، ولی چون طول تصدیق نسبت به n - طول رشته‌ی ورودی- از مرتبه‌ی چندجمله‌ای است، زمان اجرا هم نسبت به n از مرتبه‌ی چندجمله‌ای است.)

ایده‌ی اصلی اثبات این است که محاسبه‌ی Aرا به صورت دنباله‌ای از پیکربندی‌ها نشان دهیم. همان طور که در شکل ۳۴-۹ نشان داده شده است، هر پیکربندی را می‌توان به دو بخش تقسیم کرد، شامل برنامه‌ی A، شمارنده‌ی برنامه و حالت کمکی ماشین، ورودی x، تصدیق y، و حافظه‌ی عملیاتی. با شروع از یک پیکربندی اولیه (c_0)، هر پیکربندی c_i توسط مدار ترکیبی M که سخت‌افزار کامپیوتر را پیاده‌سازی می‌کند، به یک پیکربندی بعدی c_{i+1} نگاشت می‌شود. خروجی برنامه‌ی A - که یا ۰ است یا ۱- هنگام پایان A در یک بیت خاص در حافظه‌ی عملیاتی نوشته می‌شود، و اگر فرض کنیم که بعد از آن A متوقف می‌شود، این مقدار هیچ گاه تغییر نمی‌کند. بنابراین اگر الگوریتم حداکثر برای $T(n)$ مرحله اجرا شود، خروجی به عنوان یکی از بیت‌ها در $c_{T(n)}$ نمایان می‌شود.

الگوریتم کاهش F یک مدار ترکیبی می‌سازد که تمام پیکربندی‌های ساخته شده از یک پیکربندی اولیه را محاسبه می‌کند. ایده این است که $T(n)$ کپی از مدار M را به یکدیگر متصل کنیم. خروجی iامین مدار، که پیکربندی c_i را می‌سازد، مستقیماً به ورودی $(i+1)$امین مدار تغذیه می‌شود. بنابراین به جای این که پیکربندی‌ها در حافظه‌ی کامپیوتر ذخیره شوند، به سادگی به صورت مقادیری بر روی سیم‌های متصل‌کننده‌ی کپی‌های M قرار می‌گیرند.

به خاطر بیاورید که الگوریتم چندجمله‌ای کاهش F باید به درستی عمل کند. با داشتن یک ورودی x، این الگوریتم باید یک مدار $C = f(x)$ را محاسبه کند که قابل ارضا است اگر و فقط اگر یک تصدیق y وجود داشته باشد به طوری که $A(x, y) = 1$. وقتی F یک ورودی x را می‌گیرد، ابتدا $n = |x|$ را محاسبه کرده و یک مدار ترکیبی C' می‌سازد شامل $T(n)$ کپی از M. ورودی C' یک پیکربندی اولیه مربوط به یک محاسبه از $A(x, y)$ است، و خروجی، پیکربندی $c_{T(n)}$.



شکل ۳۴-۹ دنباله‌ی پیکربندی‌های ساخته شده توسط الگوریتم A هنگامی که بر روی یک ورودی x و تصدیق y اجرا می‌شود. هر پیکربندی وضعیت کامپیوتر را برای یک مرحله از محاسبه نشان می‌دهد، و علاوه بر A، x، و y، شامل شمارنده‌ی برنامه (PC)، حالت کمکی ماشین، و حافظه‌ی عملیاتی می‌شود. غیر از تصدیق y، پیکربندی اولیه‌ی c_0 ثابت است. یک مدار ترکیبی M هر پیکربندی را به پیکربندی بعدی نگاشت می‌کند. خروجی، یک بیت از پیش تعیین شده در حافظه‌ی عملیاتی است.

الگوریتم F با اصلاح جزئی 'C مدار $C = f(x)$ را می‌سازد. ابتدا ورودی‌های 'C متناظر با برنامه‌ی A، شمارنده‌ی برنامه‌ی اولیه، ورودی x، و وضعیت اولیه‌ی حافظه با سیم مستقیماً به خود این مقادیر متصل می‌شوند. بنابراین تنها ورودی باقی‌مانده برای مدار مربوط به تصدیق y است. دوم، از

تمام خروجی‌های مدار صرف‌نظر می‌شود، غیر از یک بیت $c_{T(n)}$ مربوط به خروجی A. مدار C، که بدین صورت ساخته شده است، $C(y) = A(x, y)$ را برای هر ورودی y با طول $O(n^k)$ محاسبه می‌کند. الگوریتم کاهش F، وقتی که یک رشته‌ی ورودی x به آن داده می‌شود، این مدار C را محاسبه کرده و آن را به خروجی می‌دهد.

اثبات دو خاصیت باقی می‌ماند. اول، باید نشان دهیم که F به درستی یک تابع کاهش f را محاسبه می‌کند. یعنی باید نشان دهیم که C قابل ارضا است اگر و فقط اگر یک تصدیق y وجود داشته باشد به طوری که $A(x, y) = ۱$. دوم، باید نشان دهیم که F در زمان چندجمله‌ای اجرا می‌شود.

برای این که نشان دهیم F به درستی تابع کاهش را محاسبه می‌کند، اجازه دهید فرض کنیم که یک تصدیق y به طول $O(n^k)$ وجود دارد به طوری که $A(x, y) = ۱$. در این صورت، اگر بیت‌های y را به ورودی‌های C اعمال کنیم، خروجی C عبارت خواهد بود از $C(y) = A(x, y) = ۱$. بنابراین اگر یک تصدیق وجود داشته باشد، آن گاه C قابل ارضا است. برای اثبات در جهت عکس، فرض کنید که C قابل ارضا است. بنابراین یک ورودی y برای C وجود دارد به طوری که $C(y) = ۱$، که از آن نتیجه می‌گیریم $A(x, y) = ۱$. بنابراین F به درستی یک تابع کاهش را محاسبه می‌کند.

برای کامل کردن شمای اثبات، فقط نیاز داریم نشان دهیم که F نسبت به $n = |x|$ در زمان چندجمله‌ای اجرا می‌شود. اولین مشاهده‌ای که انجام می‌دهیم این است که تعداد بیت‌های مورد نیاز برای نمایش یک پیکربندی نسبت به n از مرتبه‌ی چندجمله‌ای است. اندازه‌ی خود برنامه‌ی A ثابت است، مستقل از طول ورودی x آن. طول ورودی x، n است، و طول تصدیق y از مرتبه‌ی $O(n^k)$. چون الگوریتم حداکثر برای $O(n^k)$ مرحله اجرا می‌شود، میزان حافظه‌ی عملیاتی مورد نیاز برای A هم نسبت به n از درجه‌ی چندجمله‌ای است. (فرض می‌کنیم که این حافظه متوالی است؛ تمرین ۳۴-۳-۵ از شما می‌خواهد این بحث را برای حالتی گسترش دهید که در آن مکان‌های اشغال شده توسط A در میان ناحیه‌ی بزرگی از حافظه پراکنده شده‌اند، و الگوی خاص این پراکندگی می‌تواند برای هر ورودی x متفاوت باشد.)

اندازه‌ی مدار ترکیبی M که سخت‌افزار کامپیوتر را پیاده‌سازی می‌کند، نسبت به طول یک پیکربندی (که از مرتبه‌ی $O(n^k)$ است) از مرتبه‌ی چندجمله‌ای است. بنابراین اندازه‌ی M نسبت به n از مرتبه‌ی چندجمله‌ای است. (اکثر بخش‌های این مدار، منطق سیستم حافظه را پیاده‌سازی می‌کند.) مدار C حداکثر از $t = O(n^k)$ کپی از M تشکیل شده است، از این رو اندازه‌ی آن نسبت به n از درجه‌ی چندجمله‌ای است. الگوریتم کاهش F می‌تواند در زمان چندجمله‌ای C را از x بسازد، چرا که تک‌تک مراحل ساختن به زمان چندجمله‌ای نیاز دارند.

بنابراین زبان CIRCUIT-SAT حداقل به سختی هر زبانی در NP است، و از آن جایی که این زبان متعلق به NP است، بنابراین NP-کامل هم هست.

قضیه ۳۴-۷ مسئله‌ی قابلیت ارضای مدار NP-کامل است.

اثبات مستقیماً از لم‌های ۳۴-۵ و ۳۴-۶ و از تعریف NP-کامل بودن. ■

تمرین‌ها

۳۴-۳-۱ تحقیق کنید که مدار شکل ۳۴-۸ (ب) غیرقابل ارضا است.

۳۴-۳-۲ نشان دهید که رابطه‌ی $\leq_P$ یک رابطه‌ی تراگذار بر روی زبان‌ها است. یعنی، نشان دهید که اگر $L_1 \leq_P L_2$ و $L_2 \leq_P L_3$، آن گاه $L_1 \leq_P L_3$.

۳۴-۳-۳ اثبات کنید که $L_1 \leq_P \overline{L}$ اگر و فقط اگر $L \leq_P \overline{L}$.

۳۴-۳-۴ نشان دهید که می‌توان از یک مقداردهی ارضا کننده به عنوان یک تصدیق در یک روش دیگر برای اثبات لم ۳۴-۵ استفاده کرد. برای یک اثبات ساده‌تر از چه تصدیقی باید استفاده کرد؟

۳۴-۳-۵ اثبات لم ۳۴-۶ فرض می‌کند که حافظه‌ی عملیاتی برای الگوریتم A یک ناحیه‌ی متوالی با اندازه‌ی چندجمله‌ای را اشغال می‌کند. از این فرض در کدام قسمت اثبات استفاده می‌شود؟ بحث کنید که با این فرض هیچ گونه کلیتی از بین نمی‌رود.

۳۴-۳-۶ یک زبان L برای یک کلاس زبان‌های C و نسبت به کاهش‌های با زمان چندجمله‌ای *کامل* (complete) است اگر $L \in C$ و $L' \leq_P L$ برای تمام $L' \in C$. نشان دهید که $\emptyset$ و $\{0,1\}^*$ تنها زبان‌های P هستند که برای P و نسبت به کاهش‌های چندجمله‌ای کامل نیستند.

۳۴-۳-۷ نشان دهید که، نسبت به کاهش‌های چندجمله‌ای (تمرین ۳۴-۳-۶ را ببینید) زبان L برای کلاس NP، کامل است اگر و فقط اگر $\overline{L}$ برای کلاس co-NP، کامل باشد.

۳۴-۳-۸ الگوریتم کاهش F در اثبات لم ۳۴-۶ مدار $C = f(x)$ را بر مبنای دانسته‌های x، A، و k می‌سازد. پروفسور Statre مشاهده کرده است که x ورودی F است، ولی تنها چیزهایی که F در مورد A، k، و فاکتورهای ثابت مخفی در زمان اجرای $O(n^k)$ می‌داند، وجود آن‌ها است (چرا که زبان L به NP تعلق دارد)، و نه مقدار آن‌ها. بنابراین، پروفسور نتیجه می‌گیرد که F نمی‌تواند مدار C را بسازد، و زبان CIRCUIT-SAT لزوماً NP-سخت نیست. اشکال نتیجه‌گیری پروفسور را توضیح دهید.

۳۴-۴ اثبات‌های NP-کامل بودن

NP-کامل بودن مسئله‌ی قابلیت ارضای مدار بر پایه‌ی اثبات مستقیم $L \leq_p$ CIRCUIT-SAT قرار دارد، برای هر زبان $L \in$ NP. در این بخش، نشان خواهیم داد که چگونه می‌توان بدون کاهش *تمام* زبان‌های NP به یک زبان، NP-کامل بودن آن را اثبات کرد. این متدولوژی را با اثبات NP-کامل بودن چندین مسئله‌ی قابلیت ارضای فرمول روشن خواهیم کرد. بخش ۳۴-۵ مثال‌های زیاد دیگری از این متدولوژی خواهد داد.

لم زیر پایه‌ی متد ما برای اثبات NP-کامل بودن یک زبان است.

لم ۳۴-۸ اگر L یک زبان باشد به طوری که $L' \leq_p L$ برای یک $L' \in$ NPC، آن گاه L NP-سخت است. به علاوه، اگر $L \in$ NP، آن گاه $L \in$ NPC.

اثبات از آن جایی که L' NP-کامل است، برای تمام $L'' \in$ NP، داریم $L'' \leq_p L'$. طبق فرض، $L' \leq_p L$، و بنابراین طبق خاصیت تراگذاری (تمرین ۳۴-۳-۲)، داریم $L'' \leq_p L$، که نشان می‌دهد که L NP-سخت است. اگر $L \in$ NP، همچنین خواهیم داشت $L \in$ NPC. ■

به عبارت دیگر، با کاهش یک زبان L'، که می‌دانیم NP-کامل است، به یک زبان L، به طور ضمنی تمام زبان‌های درون NP را به L کاهش می‌دهیم. بنابراین لم ۳۴-۸ به ما یک متد می‌دهد برای اثبات این که یک زبان L NP-کامل است:

۱. اثبات می‌کنیم که $L \in$ NP.
۲. یک زبان L' انتخاب می‌کنیم که می‌دانیم NP-کامل است.
۳. یک الگوریتم ارائه می‌کنیم که یک تابع f را محاسبه می‌کند که تمام نمونه‌های $x \in \{0,1\}^*$ از L' را به یک نمونه‌ی $f(x)$ از L می‌نگارد.
۴. اثبات می‌کنیم که تابع f، $x \in L'$ را ارضا می‌کند اگر و فقط اگر $f(x) \in L$ برای تمام $x \in \{0,1\}^*$.
۵. اثبات می‌کنیم که الگوریتمی که f را محاسبه می‌کند در زمان چندجمله‌ای اجرا می‌شود.

(مراحل ۲-۵ نشان می‌دهند که L NP-سخت است.) این متدولوژی کاهش از یک زبان که می‌دانیم NP-کامل است بسیار ساده‌تر از این فرآیند است که مستقیماً نشان دهیم که هر زبان در NP را چگونه می‌توان به آن کاهش داد. اثبات این که CIRCUIT-SAT $\in$ NPC به ما یک «نقطه‌ی شروع» می‌دهد. اکنون که می‌دانیم مسئله‌ی قابلیت ارضای مدار NP-کامل است، می‌توانیم اثبات NP-کامل بودن بقیه‌ی مسائل را بسیار ساده‌تر انجام دهیم. به علاوه همین طور که یک کاتالوگ از مسائل NP-کامل توسعه می‌دهیم، انتخاب‌های بیشتری از زبان‌هایی داریم که می‌توانیم کاهش را از آن‌ها انجام دهیم.

قابلیت ارضای فرمول

با اثبات NP–کامل بودن مسئله‌ی تعیین قابل ارضا بودن یک فرمول دودویی، و نه یک مدار، متدولوژی کاهش را بیشتر روشن خواهیم کرد. این مسئله افتخار تاریخی این را دارد که اولین مسئله‌ی NP–کامل شناخته شده است.

مسئله‌ی قابلیت ارضای فرمول را بر حسب زبان‌های SAT این گونه فرمول‌بندی می‌کنیم. یک نمونه از یک SAT یک فرمول دودویی φ است که از عناصر زیر تشکیل شده است:

۱. n متغیر دودویی: $x_1, x_2, \ldots, x_n$.

۲. m رابط دودویی: هر تابع دودویی با یک یا دو ورودی و یک خروجی، مانند $\wedge$ (AND)، $\vee$ (OR)، $\neg$ (NOT)، $\rightarrow$ (ایجاب)، $\leftrightarrow$ (اگر و فقط اگر)؛ و

۳. پرانتزها. (بدون از دست دادن کلیت فرض می‌کنیم هیچ پرانتز اضافی وجود ندارد، یعنی حداکثر یک جفت پرانتز برای هر رابط دودویی وجود دارد.)

به سادگی می‌توان یک فرمول دودویی φ را با طول چندجمله‌ای نسبت به $n+m$ کدگذاری کرد. مانند مدارهای ترکیبی بولین، یک *مقداردهی صحیح* (truth assignment) برای یک فرمول بولین φ عبارت است از یک مجموعه از مقادیر برای متغیرهای φ، و یک *مقداردهی ارضا کننده* (satisfying assignment) یک مقداردهی صحیح است که باعث می‌شود مقدار کل عبارت ۱ شود. یک فرمول با یک مقداردهی ارضا کننده، یک فرمول *قابل ارضا* (satisfiable) است. مسئله‌ی قابلیت ارضا، این است که آیا یک فرمول داده شده قابل ارضا است یا خیر، که به صورت زبان‌های فرمال عبارت است از

$$\text{SAT} = \{\langle\varphi\rangle : \varphi \text{ یک فرمول دودویی قابل ارضا است}\}$$

به عنوان یک مثال، فرمول

$$\varphi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

دارای مقداردهی ارضا کننده‌ی $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1\rangle$ است، چرا که

$$\begin{aligned}\varphi &= ((0 \rightarrow 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 \\ &= (1 \vee \neg(1 \vee 1)) \wedge 1 \\ &= (1 \vee 0) \wedge 1 \\ &= 1\end{aligned} \qquad (۲-۳۴)$$

و بنابراین، این فرمول به SAT تعلق دارد.

الگوریتم ساده‌لوحانه برای تعیین این که یک فرمول دلخواه دودویی قابل ارضا است یا نه، در زمان چندجمله‌ای اجرا نمی‌شود. 2^n مقداردهی ممکن برای یک فرمول φ با n متغیر وجود دارد. اگر طول $\langle\varphi\rangle$ نسبت به n از درجه‌ی چندجمله‌ای باشد، آن گاه چک کردن تمام مقداردهی‌های ممکن به $\Omega(2^n)$ زمان نیاز دارد، که نسبت به طول $\langle\varphi\rangle$ از مرتبه‌ی فراچندجمله‌ای است. همان طور که قضیه‌ی زیر نشان می‌دهد، به احتمال زیاد یک الگوریتم چندجمله‌ای برای این مسئله وجود ندارد.

قضیه‌ی ۹-۳۴ مسئله‌ی قابلیت ارضای یک فرمول NP-کامل است.

اثبات با نشان دادن $SAT \in NP$ آغاز می‌کنیم. سپس اثبات می‌کنیم که CIRCUIT-SAT، NP-سخت است، بدین صورت که نشان می‌دهیم $CIRCUIT\text{-}SAT \leq_P SAT$؛ طبق لم ۳۴-۸ با این کار قضیه اثبات می‌شود.

برای این که نشان دهیم SAT به NP تعلق دارد، نشان می‌دهیم که می‌توان درستی یک تصدیق حاوی یک مقداردهی ارضا کننده برای یک فرمول ورودی φ را در زمان چندجمله‌ای تحقیق کرد. الگوریتم تحقیق به سادگی هر متغیر در فرمول را با مقدار متناظر آن جایگزین، و سپس کل عبارت را ارزیابی می‌کند، همان طور که در تساوی (۳۴-۲) در بالا این کار را انجام دادیم. این کار را به سادگی می‌توان در زمان چندجمله‌ای انجام داد. اگر حاصل عبارت ۱ شود، فرمول قابل ارضا است. بنابراین اولین شرط لم ۳۴-۸ برای NP-کامل بودن برقرار است.

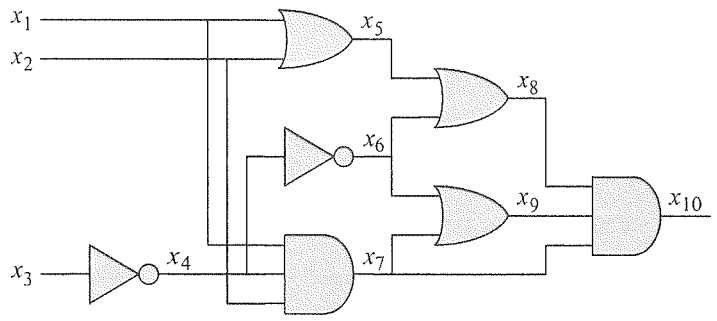
برای اثبات این که SAT، NP-سخت است، نشان می‌دهیم که $CIRCUIT\text{-}SAT \leq_P SAT$. به عبارت دیگر، هر نمونه از مسئله‌ی قابلیت ارضای مدار را می‌توان در زمان چندجمله‌ای به یک نمونه از قابلیت ارضای فرمول کاهش داد. با استفاده از استقرا می‌توان هر مدار ترکیبی دودویی را به صورت یک فرمول دودویی توصیف کرد. به سادگی به گیتی نگاه می‌کنیم که خروجی مدار را تولید می‌کند، و به صورت استقرایی هر یک از ورودی‌های گیت را به صورت یک فرمول توصیف می‌کنیم. سپس فرمول کل مدار از نوشتن یک عبارت که تابع گیت را به فرمول‌های ورودی آن اعمال می‌کند، به دست می‌آید.

متأسفانه، این متد سرراست یک کاهش چندجمله‌ای تولید نمی‌کند. همان طور که تمرین ۳۴-۴-۱ از شما می‌خواهد نشان دهید، زیرفرمول‌های مشترک-که از گیت‌هایی حاصل می‌شود که گنجایش خروجی آن‌ها ۲ یا بیشتر است-ممکن است باعث شوند که اندازه‌ی فرمول کلی به صورت نمایی رشد کند. بنابراین، الگوریتم کاهش باید به طریقی هوشمندانه‌تر باشد.

شکل ۳۴-۱۰ با استفاده از مدار شکل ۳۴-۸(الف) نشان می‌دهد که چطور می‌توانیم این مشکل را حل کنیم. برای هر سیم x_i در مدار C، فرمول φ یک متغیر x_i دارد. اکنون می‌توان نحوه‌ی عمل‌کرد هر گیت را به صورت فرمولی کوچک شامل متغیرهای سیم‌های آن توصیف کرد. مثلاً عملیات گیت AND خروجی عبارت است از $x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)$. هر یک از این فرمول‌های کوچک را یک *عبارت* (caluse) می‌نامیم.

فرمول φ ساخته شده توسط الگوریتم کاهش عبارت است از AND متغیر خروجی مدار با عطف عبارات توصیف کننده‌ی هر یک از گیت‌ها. برای مدار شکل، این فرمول عبارت است از

$$\begin{aligned}\varphi = {} & x_{10} \wedge (x_4 \leftrightarrow \neg x_3) \\ & \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \\ & \wedge (x_6 \leftrightarrow \neg x_4)\end{aligned}$$



شکل ۳۴-۱۰ کاهش قابلیت ارضای مدار به قابلیت ارضای فرمول. فرمول ساخته شده توسط الگوریتم کاهش یک متغیر برای هر سیم در مدار دارد.

$$\wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4))$$
$$\wedge (x_8 \leftrightarrow (x_5 \vee x_6))$$
$$\wedge (x_9 \leftrightarrow (x_6 \vee x_7))$$
$$\wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))$$

با داشتن یک مدار C، ساختن یک فرمول φ مانند این در زمان چندجمله‌ای کاری سرراست است. چرا مدار C دقیقاً وقتی قابل ارضا است که فرمول φ قابل ارضا باشد؟ اگر C یک مقداردهی ارضا کننده داشته باشد، هر سیم در مدار یک مقدار خوش تعریف دارد، و خروجی مدار ۱ است. بنابراین، مقداردهی مقادیر سیم‌ها به متغیرها در φ باعث می‌شود که مقدار هر عبارت در φ ۱ شود، و بنابراین عطف تمام این عبارت‌ها ۱ خواهد شد. بالعکس، اگر یک مقداردهی باعث شود φ مقدار ۱ بگیرد، مدار C طبق یک بحث مشابه قابل ارضا است. بنابراین، نشان داده‌ایم که CIRCUIT-SAT $\leq_P$ SAT، که اثبات را کامل می‌کند. ■

قابلیت ارضای 3-CNF

با کاهش از مسئله‌ی قابلیت ارضای فرمول، می‌توان NP–کامل بودن بسیاری از مسائل را اثبات کرد. ولی الگوریتم کاهش باید بتواند هر فرمول ورودی را اداره کند، و این نیاز می‌تواند به تعداد زیادی از حالات منجر شود که باید در نظر گرفته شوند. معمولاً ترجیح می‌دهیم کاهش را از یک زبان محدود از فرمول‌های بولین انجام دهیم، تا حالت‌هایی که باید در نظر بگیریم کم‌تر شوند. البته نباید محدودیت زبان به اندازه‌ای برسانیم که قابل حل در زمان چندجمله‌ای شود. یک زبان مناسب، قابلیت ارضای 3-CNF، یا 3-CNF-SAT است.

قابلیت ارضای 3-CNF را با استفاده از اصطلاحات زیر تعریف می‌کنیم. یک *لفظ* (literal) در یک فرمول دودویی عبارت است از رخ‌داد یک متغیر یا مکمل آن. یک فرمول بولین به صورت *فرم نرمال عطفی* (conjunctive normal form)، یا *CNF* است، اگر به صورت AND چند *عبارت* (clause) باشد، که هر یک از این عبارت‌ها به صورت OR یک یا چند لفظ هستند. یک فرمول دودویی به صورت *۳-فرم نرمال عطفی*، یا *3-CNF* است، اگر هر عبارت دقیقاً ۳ لفظ متمایز داشته باشد.

برای مثال، فرمول دودویی

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

به صورت 3-CNF است. اولین عبارت در این فرمول $(x_1 \vee \neg x_1 \vee \neg x_2)$ است، که حاوی سه لفظ x_1، $\neg x_1$، و $\neg x_2$ است.

در 3-CNF-SAT، می‌خواهیم بدانیم که آیا یک فرمول φ که به صورت 3-CNF است، قابل ارضا است یا نه. قضیه‌ی زیر نشان می‌دهد که یک الگوریتم چندجمله‌ای که بتواند قابل ارضا بودن یک فرمول دودویی را تعیین کند، احتمالاً وجود ندارد، حتی وقتی که آن‌ها به این فرم استاندارد ساده توصیف می‌شوند.

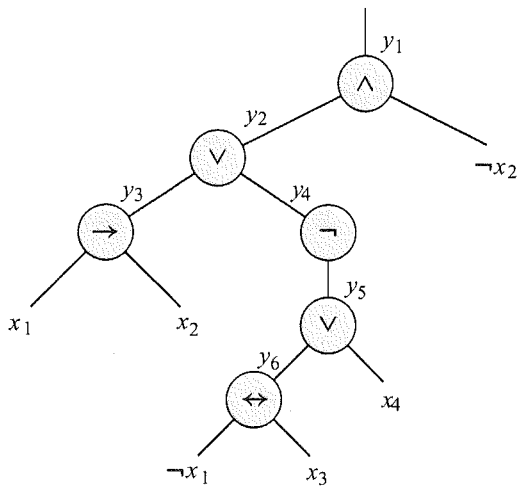
قضیه‌ی ۳۴-۱۰ قابلیت ارضای فرمول‌های دودویی در فرم نرمال عطفی ۳تایی NP-کامل است.

اثبات بحثی که از آن در اثبات قضیه‌ی ۳۴-۹ برای نشان دادن $\text{SAT} \in \text{NP}$ استفاده کردیم، در این جا هم به خوبی برای نشان دادن $\text{3-CNF-SAT} \in \text{NP}$ کاربرد دارد. بنابراین، طبق لم ۳۴-۸ فقط نیاز داریم نشان دهیم که $\text{SAT} \leq_p \text{3-CNF-SAT}$.

الگوریتم کاهش را به سه مرحله‌ی اصلی تقسیم می‌کنیم. هر مرحله فرمول ورودی φ را مقداری به فرم نرمال عطفی ۳تایی مورد نظر نزدیک‌تر می‌کند.

مرحله‌ی اول مشابه اثبات $\text{CIRCUIT-SAT} \leq_p \text{SAT}$ در قضیه‌ی ۳۴-۹ است. ابتدا، یک درخت «تجزیه» (parse) دودویی برای فرمول ورودی φ می‌سازیم، که در آن لفظ‌ها، برگ و عطف‌ها، گره‌های داخلی هستند. شکل ۳۴-۱۱ چنین درخت تجزیه‌ای را برای یک فرمول نشان می‌دهد.

$$\varphi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2 \qquad (۳-۳۴)$$



شکل ۳۴-۱۱ درخت متناظر با فرمول $\varphi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$.

اگر فرمول ورودی حاوی یک عبارت، مثلاً یک OR، باشد که چندین لفظ دارد، از شرکت‌پذیری می‌توان استفاده و عبارت را به طور کامل پرانتزگذاری کرد به طوری که تمام گره‌های داخلی در درخت حاصل ۱ یا ۲ فرزند داشته باشند. اکنون می‌توان به درخت تجزیه‌ی دودویی به صورت یک مدار برای محاسبه‌ی تابع نگاه کرد.

با تقلید از کاهش اثبات قضیه‌ی ۳۴-۹، برای خروجی هر گره‌ی داخلی یک متغیر y_i تعریف می‌کنیم. آن گاه، فرمول اصلی φ را به صورت AND متغیر ریشه و یک عطف از عبارت‌های توصیف‌کننده‌ی هر یک از گره‌ها می‌نویسیم. برای فرمول (۳۴-۳)، عبارت حاصل عبارت است از

$$\begin{aligned}\varphi' = {} & y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ & \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\ & \wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\ & \wedge (y_4 \leftrightarrow \neg y_5) \\ & \wedge (y_5 \leftrightarrow (y_6 \wedge x_4)) \\ & \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))\end{aligned}$$

مشاهده کنید که فرمول φ' که بدین صورت به دست آمده است، عطف عبارت‌های φ_i' است، که هر یک حداکثر ۳ لفظ دارند. تنها نیازی که ممکن است نتوانیم ارضا کنیم این است که هر عبارت به صورت OR سه لفظ باشد.

مرحله‌ی دوم کاهش هر عبارت φ_i' را به یک فرم نرمال عطفی تبدیل می‌کند. با ارزیابی تمام مقداردهی‌های ممکن برای متغیرها، یک جدول درستی برای φ_i' درست می‌کنیم. هر سطر در جدول درستی حاوی یک مقداردهی ممکن برای متغیرهای هر عبارت است، به همراه مقدار عبارت تحت این مقداردهی. با استفاده از ورودی‌های جدول درستی که مقدار آن‌ها ۰ است، یک فرمول در *فرم نرمال فصلی* (disjunctive normal form) (یا DNF)–OR عبارات AND– می‌سازیم که معادل $\neg\varphi_i'$ است. سپس این فرمول را معکوس کرده و با استفاده از *قوانین دمورگان*،

$$\neg(a \wedge b) = \neg a \vee \neg b,$$
$$\neg(a \vee b) = \neg a \wedge \neg b,$$

آن را به یک فرمول CNF (با نام φ_i'') تبدیل می‌کنیم تا تمام تمام لفظ‌ها را کامل، و ORها را به AND و ANDها را به OR تبدیل کنیم.

در مثال بالا، عبارت $\varphi_1' = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ را به صورت زیر به CNF تبدیل می‌کنیم. جدول درستی برای φ_i' در شکل ۳۴-۱۲ داده شده است. فرمول DNF معادل با $\neg\varphi_1'$ عبارت است از

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$$

پس از معکوس کردن و اِعمال قوانین دمورگان، فرمول CNF را به صورت زیر خواهیم داشت:

$$\begin{aligned}\varphi_1'' = {} & (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \\ & \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)\end{aligned}$$

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

شکل ۳۴-۱۲ جدول درستی برای عبارت $((y_2 \wedge \neg x_2) \leftrightarrow y_1)$.

که با عبارت اصلی φ_1' معادل است.

اکنون هر عبارت φ_i' از فرمول φ' به یک فرمول φ_i'' به صورت CNF تبدیل شده است، و بنابراین φ' معادل است با فرمول φ'' شامل عطف عبارات φ_i''، که به صورت CNF است. به علاوه، هر عبارت φ'' حداکثر ۳ لفظ دارد.

مرحله‌ی سوم و آخر کاهش، تبدیل فرمول را ادامه می‌دهد تا هر عبارت *دقیقاً* ۳ لفظ متمایز داشته باشد. فرمول 3-CNF نهایی φ''' از عبارت‌های فرمول φ'' ساخته شده است. این فرمول همچنین از دو متغیر کمکی استفاده می‌کند، که آن‌ها را p و q می‌نامیم. برای هر عبارت C_i از φ''، عبارت‌های زیر را در φ''' اضافه می‌کنیم:

- اگر C_i شامل ۳ لفظ مجزا باشد، آن گاه به سادگی خود C_i را به عنوان یک عبارت در φ''' قرار می‌دهیم.
- اگر C_i شامل ۲ لفظ مجزا باشد، یعنی اگر $C_i = (l_1 \vee l_2)$، که در آن l_1 و l_2 لفظ‌های مجزا هستند، آن گاه $(l_1 \vee l_2 \vee p) \wedge (l \vee \neg p \vee \neg q)$ را به عنوان عبارت‌های φ''' قرار می‌دهیم. لفظ‌های p و $\neg p$ صرفا نیاز وجود دقیقاً ۳ لفظ در هر عبارت را ارضا می‌کنند. صرف نظر از این که $p = 0$ یا $p = 1$، یکی از عبارت‌ها معادل $l_1 \vee l_2$ است، و دیگری همیشه مقدار ۱ دارد، که مقدار همانی برای AND است.
- اگر C_i شامل ۱ لفظ l باشد، آن گاه $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$ را به عنوان عبارت‌های φ''' قرار می‌دهیم. مستقل از مقادیر داده شده به p و q، یکی از چهار عبارت معادل l است، و سه تای دیگر معادل ۱.

با بررسی این سه مرحله می‌توانیم ببینیم که فرمول φ''' قابل ارضا است اگر و تنها اگر φ قابل ارضا باشد. مانند کاهش از CIRCUIT-SAT به SAT، ساختن φ' از φ در مرحله‌ی اول، قابلیت ارضا را حفظ می‌کند. مرحله‌ی دوم یک فرمول φ'' می‌سازد که به صورت جبری معادل φ' است. مرحله‌ی سوم یک فرمول φ''' می‌سازد، که 3-CNF است، و معادل φ''، چرا که هر مقداردهی به متغیرهای p و q یک فرمول می‌سازد که به صورت جبری معادل φ'' است.

همچنین باید نشان دهیم که این کاهش را می‌توان در زمان چندجمله‌ای انجام داد. ساختن φ' از

φ حداکثر ۱ متغیر و ۱ عبارت برای هر حرف عطف اضافه می‌کند. ساختن "φ از 'φ حداکثر می‌تواند ۸ عبارت در "φ برای هر عبارت در 'φ اضافه کند، چرا که هر عبارت 'φ حداکثر ۳ متغیر دارد، و جدول درستی هر عبارت حداکثر $2^3 = 8$ سطر دارد. ساختن ""φ از "φ حداکثر ۴ عبارت جدید برای هر عبارت در "φ معرفی می‌کند. بنابراین اندازه‌ی فرمول حاصل ""φ نسبت به طول فرمول اصلی از درجه‌ی چندجمله‌ای است. هر یک از مراحل را به سادگی می‌توان در زمان چندجمله‌ای انجام داد. ■

تمرین‌ها

۳۴-۴-۱ کاهش سرراست (غیر چندجمله‌ای) را در اثبات قضیه‌ی ۳۴-۹ در نظر بگیرید. یک مدار با اندازه‌ی n توصیف کنید که وقتی طبق این متد به یک فرمول تبدیل می‌شود، فرمولی می‌دهد که اندازه‌ی آن نسبت به n نمایی است.

۳۴-۴-۲ فرمول 3-CNF حاصل از اعمال متد قضیه‌ی ۳۴-۱۰ را بر روی فرمول (۳۴-۳) نشان دهید.

۳۴-۴-۳ پروفسور Jagger می‌خواهد فقط با استفاده از تکنیک جدول درستی در اثبات قضیه‌ی ۳۴-۱۰، و نه مراحل دیگر، نشان دهد که $\text{SAT} \leq_p \text{3-CNF-SAT}$. یعنی، روش پیشنهادی پروفسور بدین صورت است که فرمول دودویی φ را گرفته، جدول درستی متغیرهای آن را تهیه می‌کنیم، از آن یک فرمول به شکل 3-DNF استخراج می‌کنیم که معادل $\neg\varphi$ است، و سپس آن را معکوس کرده و قوانین دمورگان را برای ساختن یک فرمول 3-CNF معادل با φ به کار می‌بریم. نشان دهید که این استراتژی به یک کاهش چندجمله‌ای ختم نمی‌شود.

۳۴-۴-۴ نشان دهید که مسئله‌ی تعیین این که یک فرمول دودویی درست‌نما است یا نه، co-NP است. (*راهنمایی:* تمرین ۳۴-۳-۷ را ببینید.)

۳۴-۴-۵ نشان دهید که مسئله‌ی تعیین قابلیت ارضای یک فرمول دودویی در فرم نرمال فصلی، قابل حل در زمان چندجمله‌ای است.

۳۴-۴-۶ فرض کنید که یک نفر به شما یک الگوریتم چندجمله‌ای برای تعیین قابلیت ارضای فرمول‌ها می‌دهد. توضیح دهید که چگونه می‌توان از این فرمول برای یافتن مقداردهی ارضا کننده در زمان چندجمله‌ای استفاده کرد.

۳۴-۴-۷ فرض کنید 2-CNF-SAT مجموعه‌ی فرمول‌های دودویی قابل ارضا در CNF با دقیقاً دو لفظ در هر عبارت باشد. نشان دهید که $\text{2-CNF-SAT} \in \text{P}$. الگوریتم خود را تا حد ممکن کارامد طراحی کنید. (*راهنمایی:* مشاهده کنید که $x \vee y$ معادل است با $\neg x \to y$. مسئله‌ی 2-CNF-SAT را به مسئله‌ای روی یک گراف جهت‌دار تبدیل کنید که به صورت بهینه قابل حل باشد.)

۳۴-۵ مسائل NP-کامل

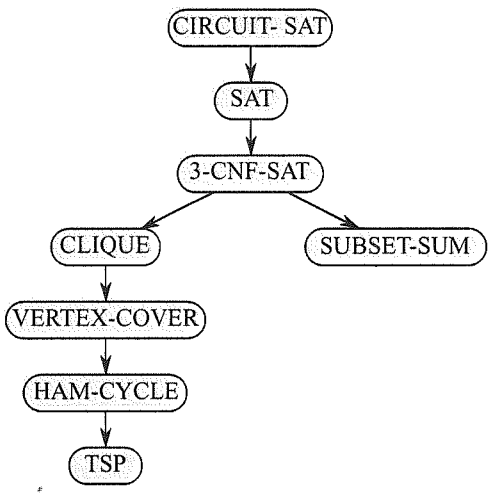
مسائل NP-کامل در زمینه‌های مختلفی پیش می‌آیند: منطق بولین، گراف‌ها، ریاضیات، طراحی شبکه، مجموعه‌ها و افرازها، حافظه و بازیابی اطلاعات، ترتیب‌دهی و برنامه‌ریزی، برنامه‌نویسی ریاضیاتی، جبر و نظریه‌ی اعداد، بازی‌ها و معماها، اتوماتا و نظریه‌ی زبان‌ها، بهینه‌سازی برنامه، زیست‌شناسی، شیمی، فیزیک، و غیره. در این بخش، از متدولوژی کاهش برای ارائه‌ی اثبات NP-کامل بودن مسائل مختلفی در حوزه‌ی نظریه‌ی گراف و افراز مجموعه‌ها استفاده می‌کنیم.

شکل ۳۴-۱۳ ساختار اثبات‌های NP-کامل بودن در این بخش و بخش ۳۴-۴ را نشان می‌دهد. NP-کامل بودن هر زبان در شکل با کاهش از زبانی که به آن اشاره می‌کند، اثبات می‌شود. در ریشه، CIRCUIT-SAT قرار دارد، که NP-کامل بودن آن را در قضیه‌ی ۳۴-۷ اثبات کردیم.

۳۴-۵-۱ مسئله‌ی گروهک

یک *گروهک* (clique) در یک گراف بدون جهت $G = (V, E)$، یک زیرمجموعه‌ی $V' \subseteq V$ از رأس‌ها است، که در آن هر جفت از رأس‌ها با یک یال در E به یکدیگر متصل شده‌اند. به عبارت دیگر یک گروهک، یک زیرگراف کامل از G است. *اندازه‌ی* یک گروهک برابر است با تعداد رأس‌های آن. مسئله‌ی گروهک عبارت است از مسئله‌ی بهینه‌سازی یافتن یک گروهک با اندازه‌ی بیشینه در یک گراف. مانند یک مسئله‌ی تصمیم‌گیری، به سادگی مسئله‌ی وجود یک گروهک با یک اندازه‌ی داده شده‌ی k در یک گراف را در نظر می‌گیریم. تعریف رسمی این مسئله بدین صورت است:

$$\text{CLIQUE} = \{\langle G, k\rangle : G \text{ یک گراف است با یک گروهک با اندازه‌ی } k\}$$



شکل ۳۴-۱۳ ساختار اثبات‌های NP-کامل بودن در بخش‌های ۳۴-۴ و ۳۴-۵. تمام اثبات‌ها نهایتاً با کاهش از NP-کامل بودن CIRCUIT-SAT انجام می‌شود.

یک الگوریتم ساده‌لوحانه برای تعیین این که یک گراف $G=(V,E)$ با $|V|$ رأس، یک گروهک با اندازه‌ی k دارد یا خیر، این است که تمام زیرمجموعه‌های kتایی از V را لیست کنیم، و چک کنیم که آیا هیچ یک از آن‌ها یک گروهک را تشکیل می‌دهند یا خیر. زمان اجرای این الگوریتم $\Omega(k^2\binom{|V|}{k})$ است، که در صورت ثابت بودن k از مرتبه‌ی چندجمله‌ای است. با این حال، به طور کلی k می‌تواند نزدیک $|V|/2$ باشد، که در این حالت الگوریتم در زمان فراچندجمله‌ای اجرا می‌شود. همان طور که حدس زده‌اید، احتمالاً یک الگوریتم کارامد برای مسئله‌ی گروهک وجود ندارد.

قضیه‌ی ۳۴-۱۱ مسئله‌ی گروهک NP-کامل است.

اثبات برای این که نشان دهیم CLIQUE $\in$ NP، برای یک گراف داده شده‌ی $G=(V,E)$، از مجموعه‌ی $V' \subseteq V$ از رأس‌های گروهک به عنوان یک تصدیق برای G استفاده می‌کنیم. چک کردن گروهک بودن V' را می‌توان در زمان چندجمله‌ای انجام داد، بدین صورت که برای هر دو رأس $u,v \in V'$، چک می‌کنیم که آیا یال (u,v) عضو E هست یا نه.

سپس، اثبات می‌کنیم که 3-CNF-SAT $\le_P$ CLIQUE، که نشان می‌دهد که مسئله‌ی گروهک NP-سخت است. ممکن است از این که می‌توانیم چنین نتیجه‌ای را اثبات کنیم تعجب کنید، چرا که در ظاهر، فرمول‌های منطقی رابطه‌ی بسیار کمی با گراف‌ها دارند.

الگوریتم کاهش با یک نمونه از 3-CNF-SAT آغاز می‌شود. فرض کنید $\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ یک فرمول بولین به شکل 3-CNF با k عبارت باشد. برای $r=1,2,\ldots,k$، هر عبارت C_r دقیقاً سه لفظ متمایز l_1^r, l_2^r، و l_3^r دارد. یک گراف G خواهیم ساخت به طوری که φ قابل ارضا باشد اگر و اگر G یک گروهک با اندازه‌ی k داشته باشد.

گراف $G=(V,E)$ به صورت زیر ساخته می‌شود. برای هر عبارت $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ در φ، یک سه‌تایی از رأس‌های v_1^r، v_2^r، و v_3^r در V قرار می‌دهیم. یک یال بین دو رأس v_i^r و v_j^s قرار می‌دهیم اگر هر دو شرط زیر برقرار باشد:

- v_i^r و v_j^s در سه‌تایی‌های مختلف باشند، یعنی $r \neq s$، و
- لفظ‌های متناظر آن‌ها با هم *سازگار* (consistent) باشند، یعنی l_i^r نقیض l_j^s نباشد.

این گراف را به سادگی می‌توان از φ در زمان چندجمله‌ای محاسبه کرد. به عنوان یک مثال، اگر داشته باشیم

$$\varphi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \wedge x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

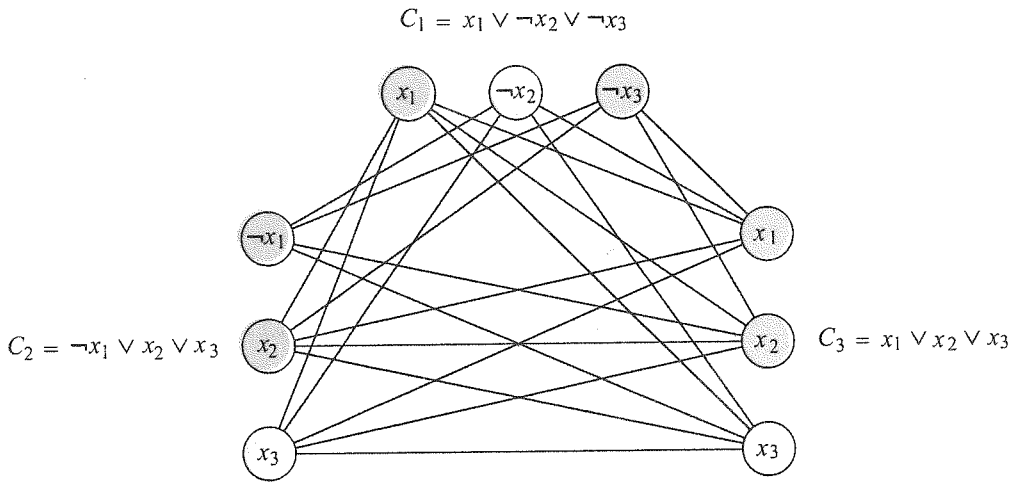
آن گاه G گراف نشان داده شده در شکل ۳۴-۱۴ است.

باید نشان دهیم که این تبدیل از φ به G یک کاهش است. اول، فرض کنید که φ یک مقداردهی ارضا کننده داشته باشد. در این صورت، هر عبارت C_r حداقل شامل یک لفظ l_i^r است که با ۱

مقداردهی شده است، و هر لفظ متناظر است با یک رأس v_i^r. انتخاب یک چنین لفظ «صحیح» از هر عبارت، یک مجموعه‌ی V' از k رأس به دست می‌دهد. ادعا می‌کنیم که V' یک گروهک است. برای هر دو رأس $v_i^r, v_j^s \in V'$، که در آن $r \neq s$، هر دو لفظ متناظر l_i^r و l_j^s با یک مقداردهی ارضا کننده، به ۱ نگاشت شده‌اند، و بنابراین لفظ‌ها نمی‌توانند نقیض یکدیگر باشند. بنابراین طبق ساختار G، یال (v_i^r, v_j^s) متعلق به E است.

برعکس، فرض کنید که G یک گروهک V' با اندازه‌ی k داشته باشد. هیچ یالی در G، رأس‌های یک سه‌تایی را به هم متصل نمی‌کند، و بنابراین V' از هر سه‌تایی دقیقاً یک رأس دارد. می‌توانیم به هر لفظ l_i^r به طوری که $v_i^r \in V'$ مقدار ۱ بدهیم بدون این که نگران این باشیم که به یک لفظ و نقیض آن مقدار ۱ داده باشیم، چرا که G حاوی هیچ یالی بین دو لفظ نقیض نیست. تمام عبارت‌ها ارضا می‌شوند، و همچنین φ. (هر متغیری که با یک رأس در گروهک متناظر نیست، می‌تواند مقداری دلخواه داشته باشد.) ■

در مثال شکل ۳۴-۱۴، یک مقداردهی ارضا کننده برای φ به صورت $x_2 = 0$ و $x_3 = 1$ است. یک گروهک متناظر با اندازه‌ی $k = 3$ شامل رأس‌های متناظر با $\neg x_2$ از عبارت اول، x_3 از عبارت دوم، و x_3 از عبارت سوم است. چون این گروهک شامل هیچ رأسی متناظر با x_1 یا $\neg x_1$ نیست، می‌توانیم به x_1 مقدار ۰ یا ۱ بدهیم.



شکل ۳۴-۱۴ گراف G که در کاهش 3-CNF-SAT به CLIQUE از فرمول $\varphi = C_1 \wedge C_2 \wedge C_3$ گرفته شده است، که در آن $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$، $C_2 = (\neg x_1 \vee x_2 \vee x_3)$، و $C_3 = (x_1 \vee x_2 \vee x_3)$. یک مقداردهی ارضا کننده به فرمول $x_2 = 0$، $x_3 = 1$ است (x_1 می‌تواند ۰ یا ۱ باشد). این مقداردهی، C_1 را با $\neg x_2$، و C_2 و C_3 را با x_3 ارضا می‌کند، که متناظر است با گروهک شامل رأس‌های با سایه‌ی روشن.

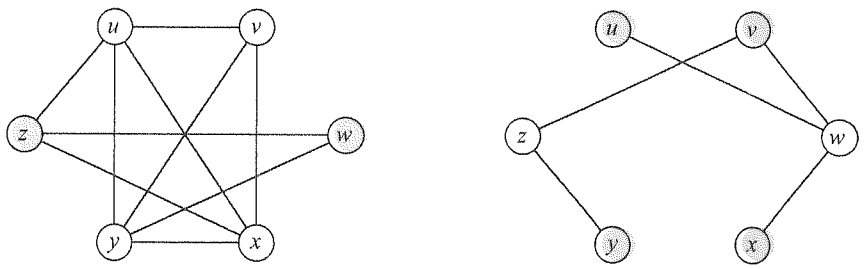
مشاهده کنید که در اثبات قضیه‌ی ۳۴-۱۱، یک نمونه‌ی دلخواه از 3-CNF-SAT را به یک نمونه از CLIQUE با یک ساختار خاص کاهش دادیم. ممکن است به نظر بیاید فقط نشان داده‌ایم که CLIQUE در گراف‌هایی که رأس‌ها فقط می‌توانند در گروه‌های ۳تایی باشند، و یالی بین رأس‌های یک ۳تایی وجود نداشته باشد، NP–سخت است. البته، فقط نشان داده‌ایم که CLIQUE در گراف‌هایی که بدین صورت محدود شده‌اند، NP–سخت است، ولی این اثبات برای نشان دادن این که CLIQUE در حالت کلی NP–سخت است، کفایت می‌کند. چرا؟ اگر ما یک الگوریتم چندجمله‌ای داشته باشیم که CLIQUE را در حالت کلی حل کند، همچنین خواهد توانست آن را در حالت محدود شده حل کند.

با این حال رویکرد معکوس – کاهش نمونه‌های 3-CNF-SAT با ساختاری خاص به نمونه‌های CLIQUE در حالت کلی – کافی نخواهد بود. چرا؟ فرض کنید نمونه‌هایی از 3-CNF-SAT که برای کاهش انتخاب می‌کنیم «آسان» باشند. در این صورت مسئله‌ای که آن را به CLIQUE کاهش داده‌ایم NP–سخت محسوب نخواهد شد.

همچنین مشاهده کنید که این کاهش از نمونه‌ای از 3-CNF-SAT استفاده کرد، و نه از جواب آن. ساختن الگوریتم کاهش چندجمله‌ای بر مبنای دانش این که فرمول φ قابل ارضا است یا خیر، یک اشتباه است، چرا که نمی‌دانیم چطور باید این اطلاعات را در زمان چندجمله‌ای به دست آوریم.

۳۴-۵-۲ مسئله‌ی پوشش رأسی

پوشش رأسی (vertex cover) یک گراف بدون جهت $G = (V, E)$ یک زیرمجموعه‌ی $V' \subseteq V$ است به طوری که اگر $(u, v) \in E$، آن گاه $u \in V'$ یا $v \in V'$ (یا هر دو). یعنی، هر رأس یال‌های مجاور خود را «پوشش» می‌دهد، و یک پوشش رأسی برای G، مجموعه‌ای است از رأس‌ها که تمام یال‌های E را پوشش می‌دهند. *اندازه* ی یک پوشش رأسی برابر است با تعداد رأس‌های آن. به عنوان مثال، گراف شکل ۳۴-۱۵ (ب) یک پوشش رأسی $\{w, z\}$ با اندازه‌ی ۲ دارد.



شکل ۳۴-۱۵ کاهش CLIQUE به VERTEX-COVER. (الف) یک گراف بدون جهت $G = (V, E)$ با گروهک $V' = \{u, v, x, y\}$. (ب) گراف $\overline{G}$ که توسط الگوریتم کاهش ساخته شده است، و دارای پوشش رأسی $V - V' = \{w, z\}$ است.

مسئله‌ی پوشش رأسی (vertex-cover problem) عبارت است از یافتن یک پوشش رأسی با اندازه‌ی کمینه در یک گراف داده شده. تبدیل این مسئله‌ی بهینه‌سازی به یک مسئله‌ی تصمیم‌گیری بدین صورت است: می‌خواهیم تعیین کنیم که آیا یک گراف، یک پوشش رأسی با اندازه‌ی داده شده‌ی k دارد یا خیر. به صورت یک زبان، تعریف می‌کنیم

$$\text{VERTEX-COVER} = \{\langle G, k\rangle : \text{گراف } G \text{ یک پوشش رأسی با اندازه‌ی } k \text{ دارد}\}$$

قضیه‌ی زیر نشان می‌دهد که این مسئله NP-کامل است.

قضیه‌ی ۳۴-۱۲ مسئله‌ی پوشش رأسی NP-کامل است.

اثبات ابتدا نشان می‌دهیم که $\text{VERTEX-COVER} \in \text{NP}$. فرض کنید که به ما یک گراف $G = (V, E)$ و یک عدد صحیح k داده شده است. تصدیقی که انتخاب می‌کنیم، خود پوشش رأسی $V' \subseteq V$ است. الگوریتم تحقیق تأیید می‌کند که $|V'| = k$، و سپس برای هر یال $(u,v) \in E$ چک می‌کند که $u \in V'$ یا $v \in V'$. این تحقیق را می‌توان به سادگی در زمان چندجمله‌ای انجام داد.

اثبات می‌کنیم که مسئله‌ی پوشش رأسی NP- سخت است، بدین صورت که نشان می‌دهیم $\text{CLIQUE} \le_p \text{VERTEX-COVER}$. این کاهش بر مبنای مفهوم «مکمل» یک گراف است. با داشتن یک گراف بدون جهت $G = (V, E)$، *مکمل* (complement) گراف G را به صورت $\overline{G} = (V, \overline{E})$ تعریف می‌کنیم، که در آن

$$\overline{E} = \{(u,v) : (u,v) \notin E \text{ و } u \neq v \text{ ، } u,v \in V\}$$

به عبارت دیگر، $\overline{G}$ گرافی است دقیقاً حاوی آن یال‌هایی که در G نیستند. شکل ۳۴-۱۵ یک گراف و مکمل آن را نشان می‌دهد، و همچنین کاهش از CLIQUE به VERTEX-COVER را.

الگوریتم کاهش یک نمونه‌ی $\langle G, k\rangle$ از مسئله‌ی گروهک را به عنوان ورودی دریافت می‌کند، و مکمل G را محاسبه می‌کند، که به سادگی در زمان چندجمله‌ای انجام می‌شود. خروجی الگوریتم کاهش نمونه‌ی $\langle \overline{G}, |V| - k\rangle$ از مسئله‌ی پوشش رأسی است. برای کامل کردن اثبات، نشان می‌دهیم که این تبدیل در واقع یک کاهش است: گراف G یک گروهک با اندازه‌ی k دارد اگر و فقط اگر گراف $\overline{G}$ یک پوشش رأسی با اندازه‌ی $|V| - k$ داشته باشد.

فرض کنید G یک گروهک $V' \subseteq V$ داشته باشد، که $|V'| = k$. ادعا می‌کنیم که $V - V'$ یک پوشش رأسی در $\overline{G}$ است. فرض کنید (u,v) یک یال در $\overline{E}$ باشد. آن گاه $(u,v) \notin E$، که نتیجه می‌دهد که حداقل یکی از رأس‌های u و v متعلق به V' نباشد، چرا که هر جفت رأس در V' با یک یال از E به هم متصل شده‌اند. به طور معادل، حداقل یکی از رأس‌های u و v در $V - V'$ است، که به این معنی است که یال (u,v) توسط $V - V'$ پوشش داده شده است. چون (u,v) به صورت دلخواه از $\overline{E}$ انتخاب شده است، تمام یال‌های $\overline{E}$ توسط یک رأس در $V - V'$ پوشش داده شده است. بنابراین مجموعه‌ی $V - V'$، که اندازه‌ی آن $|V| - k$ است، یک پوشش رأسی است برای $\overline{G}$.

بالعکس، فرض کنید $\overline{G}$ یک پوشش رأسی $V' \subseteq V$ دارد، که در آن $|V'| = |V| - k$. در این صورت برای تمام $u,v \in V$، اگر $(u,v) \in \overline{E}$، آن گاه $u \in V'$ یا $v \in V'$، یا هر دو. عکس نقیض این استنباط این است که برای تمام $u,v \in V$، اگر $u \notin V'$ و $v \notin V'$، آن گاه $(u,v) \in E$. به عبارت دیگر $V - V'$ یک گروهک است، و اندازه‌ی آن برابر است با $|V| - |V'| = k$. ■

چون VERTEX-COVER، NP–کامل است، انتظار نداریم که بتوانیم یک الگوریتم چندجمله‌ای برای یافتن یک پوشش رأسی با اندازه‌ی کمینه بیابیم. با این حال، بخش ۳۵-۱ یک «الگوریتم تقریبی» چندجمله‌ای ارائه می‌کند، که یک جواب «تقریبی» برای مسئله‌ی پوشش رأسی می‌یابد. اندازه‌ی یک پوشش تقریبی ساخته شده توسط این الگوریتم حداکثر دو برابر اندازه‌ی پوشش رأسی کمینه است.

بنابراین نباید فقط به خاطر این که یک مسئله NP–کامل است، امید خود را از دست بدهیم. ممکن است بتوانیم یک الگوریتم تقریبی چندجمله‌ای طراحی کنیم که جواب‌های نزدیک بهینه تولید می‌کند، در حالی که یافتن جواب بهینه NP–کامل است. فصل ۳۵ الگوریتم‌های تقریبی مختلفی برای مسائل NP–کامل می‌دهد.

۳۴-۵-۳ مسئله‌ی دور همیلتونی

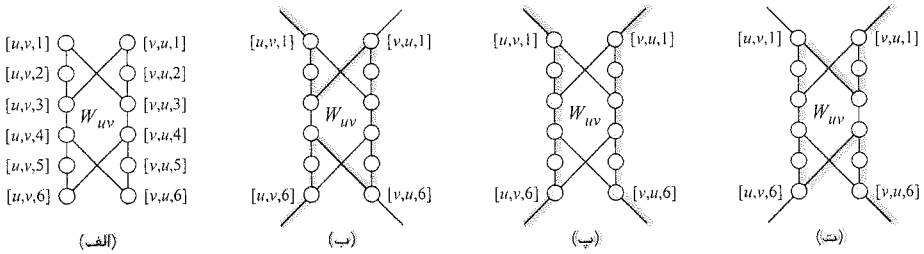
اکنون به مسئله‌ی دور همیلتونی تعریف شده در بخش ۳۴-۲ بازمی‌گردیم.

قضیه‌ی ۳۴-۱۲ مسئله‌ی دو همیلتونی NP–کامل است.

اثبات ابتدا نشان می‌دهیم که HAM-CYCLE به NP تعلق دارد. با داشتن یک گراف $G = (V,E)$، تصدیق ما، دنباله‌ای است از $|V|$ رأس که یک دور همیلتونی را تشکیل می‌دهند. الگوریتم تحقیق، چک می‌کند که آیا تمام رأس‌های V دقیقاً یک بار در این دنباله قرار داشته باشند، و اگر رأس اول در انتها هم تکرار شود، یک دور در G تشکیل شود. یعنی، چک می‌کند که یک یال بین هر دو رأس متوالی، و بین رأس‌های اول و آخر در G قرار دارد یا نه. این تحقیق را می‌توان در زمان چندجمله‌ای انجام داد.

اکنون اثبات می‌کنیم که $\text{VERTEX-COVER} \leq_p \text{HAM-CYCLE}$، که نشان می‌دهد که HAM-CYCLE یک مسئله‌ی NP–کامل است. با داشتن یک گراف $G = (V,E)$ و یک عدد صحیح k، یک گراف بدون جهت $G' = (V',E')$ می‌سازیم که یک دور همیلتونی دارد اگر و فقط اگر G یک پوشش رأسی با اندازه‌ی k داشته باشد.

برای ساختن این گراف از یک *widget* استفاده می‌کنیم، که قطعه‌ای از یک گراف است که ویژگی‌های خاصی دارد. شکل ۳۴-۱۶ (الف) widgetی که از آن استفاده می‌کنیم را نشان می‌دهد. برای هر یال $(u,v) \in E$، گراف G' که می‌سازیم شامل یک کپی از این widget است، که آن را با W_{uv} نشان می‌دهیم. هر یال در W_{uv} را با $[u,v,i]$ یا $[v,u,i]$ نشان می‌دهیم، که در آن $1 \leq i \leq 6$، پس هر widget



شکل ۳۴-۱۶ widget استفاده شده در کاهش مسئله‌ی پوشش رأسی به مسئله‌ی دور همیلتونی. یک یال (u,v) از گراف G متناظر است با W_{uv} در گراف G' که در کاهش ساخته شده است. (الف) widget مورد استفاده به همراه برچسب‌گذاری رأس‌ها. (ب)-(ت) مسیرهای سایه‌دار، تنها مسیرهای ممکن از روی widget هستند که تمام رأس‌ها را در بر می‌گیرند، با این فرض که تنها ارتباط widget با بقیه‌ی G' از طریق رأس‌های $[u,v,1]$، $[u,v,6]$، $[v,u,1]$، و $[v,u,6]$ است.

حاوی ۱۲ رأس است. همچنین، W_{uv} حاوی ۱۴ یال است که در شکل ۳۴-۱۶ (الف) نشان داده شده است.

به همراه ساختار داخلی یک widget، با محدود کردن ارتباط widget با بقیه‌ی گراف G' که می‌سازیم، خصوصیات مورد نظر خود را به دست می‌آوریم. به طور خاص، تنها رأس‌های $[u,v,1]$، $[u,v,6]$، $[v,u,1]$، و $[v,u,6]$ یال مجاور با خارج W_{uv} دارند. هر دور همیلتونی در G' باید یال‌های $W_{u,v}$ را در یکی از مسیرهای نشان داده شده در شکل ۳۴-۱۶ (ب)-(ت) طی کند. اگر دور از طریق رأس $[u,v,1]$ وارد شود، باید از یال $[u,v,6]$ خارج شود، و یا تمام ۱۲ رأس widget (شکل ۳۴-۱۶ (ب)) را ملاقات کند، و یا شش رأس $[u,v,1]$ تا $[u,v,6]$ (شکل ۳۴-۱۶ (پ)). در حالت دوم، دور باید دوباره وارد widget شود تا رأس‌های $[v,u,1]$ تا $[v,u,6]$ را ملاقات کند. به طور مشابه، اگر دور از طریق رأس $[v,u,1]$ وارد شود، باید از طریق رأس $[v,u,6]$ خارج شود، و یا تمام ۱۲ رأس را ملاقات کند (شکل ۳۴-۱۶ (ت)) یا شش رأس $[v,u,1]$ تا $[v,u,6]$ را (شکل ۳۴-۱۶ (پ)). هیچ مسیر دیگری که تمام ۱۲ رأس widget را ملاقات کند وجود ندارد. به طور خاص، ساختن دو مسیر مجزای رأسی که یکی رأس‌های $[u,v,1]$ تا $[v,u,6]$ را ملاقات کند، و دیگری $[v,u,1]$ تا $[u,v,6]$ را، به طوری که اجتماع دو مسیر تمام رأس‌های widget را پوشش دهد، ممکن نیست.

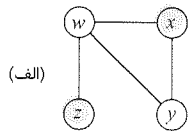
تنها رأس‌های دیگر در V' غیر از رأس‌های widget، *رأس‌های انتخاب‌گر* (selector vertex) $s_1,s_2,\ldots,s_k$ هستند. از یال‌های مجاور با رأس‌های انتخاب‌گر در G' برای انتخاب k رأس در پوشش رأسی در G استفاده خواهیم کرد.

علاوه بر یال‌های درون widgetها، E' حاوی دو نوع یال دیگر هم هست، که در شکل ۳۴-۱۷ نشان داده شده‌اند. ابتدا برای هر رأس $u \in V$ یال‌هایی اضافه می‌کنیم که جفت‌های widget را به هم متصل کنیم تا مسیری تشکیل شود شامل تمام widgetهای متناظر با یال‌های مجاور u در G. به طور

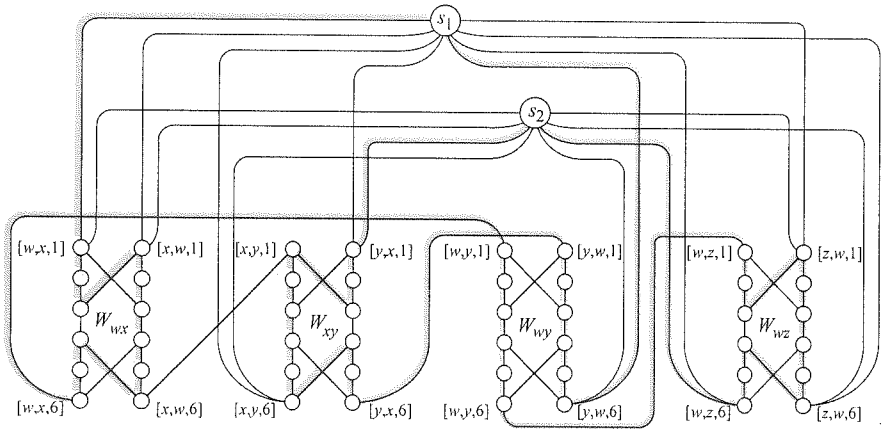
دلخواه رأس‌های مجاور با هر رأس $u \in V$ را به صورت $u^{(1)}$، $u^{(2)}$،...، $u^{(\text{degree}(u))}$ نام‌گذاری می‌کنیم، که در آن degree(u) برابر است با تعداد رأس‌های مجاور با u. یک مسیر در G' می‌سازیم که از تمام widgetهای متناظر با رأس‌های مجاور u می‌گذرد، بدین صورت که یال‌های $\{([u,u^{(i)},۶],[u,u^{(i+1)},۱]) : ۱ \le i \le \text{degree}(u)-۱\}$ را به E' اضافه می‌کنیم. برای مثال در شکل ۳۴-۱۷، رأس‌های مجاور با w را به صورت x، y، z مرتب می‌کنیم، و بنابراین گراف G' در بخش (ب) در شکل شامل یال‌های $([w,x,۶],[w,y,۱])$ و $([w,y,۶],[w,z,۱])$ می‌شود. برای هر رأس $u \in V$، این یال‌ها در G' یک مسیر شامل تمام widgetهای متناظر با یال‌های مجاور با u در G را تشکیل می‌دهند.

شهود پشت انتخاب این یال‌ها این است که اگر یک رأس $u \in V$ را در پوشش رأسی G انتخاب کنیم، می‌توانیم یک مسیر $[u,u^{(1)},۱]$ تا $[u,u^{(\text{degree}(u))},۶]$ در G' تشکیل دهیم که تمام widgetهای متناظر با یال‌های مجاور u را «پوشش» می‌دهد. یعنی برای هر یک از این widgetها، مثلاً $W_{u,u^{(i)}}$، مسیر ساخته شده یا شامل تمام ۱۲ رأس آن می‌شود (اگر u در پوشش رأسی باشد ولی $u^{(i)}$ نباشد) و یا فقط شامل شش رأس $[u,u^{(i)},۱]$، $[u,u^{(i)},۲]$،...، $[u,u^{(i)},۶]$(اگر هم u در پوشش رأسی باشد و هم $u^{(i)}$).

(الف)



(ب)



شکل ۳۴-۱۷ کاهش از یک نمونه از مسئله‌ی پوشش رأسی به یک نمونه از مسئله‌ی دور همیلتونی. (الف) یک گراف بدون جهت G با یک پوشش رأسی با اندازه‌ی ۲، متشکل از رأس‌های با سایه‌ی کم‌رنگ w و y. (ب) گراف بدون جهت G' که توسط کاهش ساخته شده است، با مسیر همیلتونی متناظر با پوشش رأسی سایه‌دار. پوشش رأسی $\{w,y\}$ متناظر است با یال‌های $(s_۱,[w,x,۱])$ و $(s_۲,[y,x,۱])$ که در دور همیلتونی ظاهر شده‌اند.

آخرین نوع یال در E'، اولین رأس $[u,u^{(1)},1]$ و آخرین رأس $[u,u^{(\text{degree}(u))},6]$ از هر یک از این مسیرها را به هر یک از رأس‌های انتخاب‌گر متصل می‌کند. یعنی، یال‌های

$$\{(s_j,[u,u^{(1)},1]) : u \in V \text{ و } 1 \le j \le k\}$$
$$\cup \{(s_j,[u,u^{(\text{degree}(u))},6]) : u \in V \text{ و } 1 \le j \le k\}$$

را هم انتخاب می‌کنیم. سپس، نشان می‌دهیم اندازه‌ی G' نسبت به اندازه‌ی G از درجه‌ی چندجمله‌ای است، و بنابراین می‌توانیم G' را در زمان چندجمله‌ای نسبت به G بسازیم. رأس‌های G'، رأس‌های widgetها هستند، به علاوه‌ی رأس‌های انتخاب‌گر. هر widget شامل ۱۲ رأس می‌شود، و $k \le |V|$ رأس انتخاب‌گر وجود دارد، که در کل برابر خواهد بود با

$$\begin{aligned}|V'| &= 12|E| + k \\ &\le 12|E| + |V|\end{aligned}$$

رأس. یال‌های G' آن‌هایی هستند که در widgetها هستند، به علاوه‌ی آن‌هایی که بین widgetها هستند، به علاوه‌ی آن‌هایی که رأس‌های انتخاب‌گر را به widgetها متصل می‌کنند. ۱۴ یال در هر widget وجود دارد، یا $14|E|$ یال در تمام widgetها. برای هر رأس $u \in V$، $\text{degree}(u)-1$ یال بین widgetها وجود دارد، که با جمع زدن بر روی تمام رأس‌های V

$$\sum_{u \in V} (\text{degree}(u) - 1) = 2|E| - |V|$$

یال بین widgetها خواهیم داشت. نهایتاً، دو یال برای هر جفت شامل یک رأس انتخاب‌گر و یک رأس از V خواهیم داشت، یعنی $2k|V|$ یال از این نوع. بنابراین تعداد کل یال‌ها در G' برابر است با

$$\begin{aligned}|E'| &= (14|E|) + (2|E| - |V|) + (2k|V|) \\ &= 16|E| + (2k-1)|V| \\ &\le 16|E| + (2|V| - 1)|V|\end{aligned}$$

اکنون نشان می‌دهیم که تبدیل از گراف G به G' یک کاهش است. یعنی، باید نشان دهیم که G یک پوشش رأسی با اندازه‌ی k دارد اگر و فقط اگر G' یک دور همیلتونی داشته باشد.

فرض کنید که $G = (V,E)$ یک پوشش رأسی $V^* \subseteq V$ با اندازه‌ی k داشته باشد. فرض کنید $V^* = \{u_1,u_2,\ldots,u_k\}$. همان طور که شکل ۳۴-۱۷ نشان می‌دهد، یک دور همیلتونی در G را با انتخاب یال‌های[1] زیر برای هر رأس $u_j \in V^*$ می‌سازیم: یال‌های $\{([u_j,u_j^{(i)},6],[u_j,u_j^{(i+1)},1]) : 1 \le i \le \text{degree}(u_j)\}$، که تمام widgetهای متناظر با یال‌های مجاور u_j را به هم متصل می‌کنند، به علاوه‌ی یال‌های درون این widgetها، همان طور که شکل ۳۴-۱۶ (ب)-(ت) نشان می‌دهد، بسته به این که آن یال توسط یک یا دو رأس در V^* پوشش داده شده باشد. دور همیلتونی شامل یال‌های زیر هم می‌شود:

---

[1] از نظر فنی، یک دوررا بر حسب رأس‌ها تعریف می‌کنیم، و نه یال‌ها (بخش ب-۴ را ببینید). در این جا برای شفافیت، از این قانون سرپیچی کرده و دور همیلتونی را بر حسب یال‌های آن تعریف می‌کنیم.

$$\{s_j,[u_j,u_j^{(1)},1]):1\le j\le k\}\cup$$
$$\{(s_{j+1},[u_j,u_j^{(\text{degree}(u_j))},6]):1\le j\le k-1\}\cup$$
$$\{(s_1,[u_k,u_k^{(\text{degree}(u_k))},6])\}$$

با بررسی شکل ۳۴-۱۷، خواننده می‌تواند مشاهده کند که این یال‌ها یک دور را تشکیل می‌دهند. دور با s_1 آغاز می‌شود، تمام widgetهای متناظر با یال‌های مجاور u_1 را ملاقات می‌کند، سپس به s_2 می‌رسد، تمام widgetهای متناظر با یال‌های مجاور با u_2 را ملاقات می‌کند، و به همین ترتیب تا آخر، تا زمانی که به s_1 بازگردد. هر widget یا یک بار ملاقات می‌شود و یا دو بار، بسته به این که یک یا دو رأس از V^* یال متناظر با آن را پوشش دهد. چون V^* یک پوشش رأسی برای G است، هر یال در E با یکی از رأس‌های V^* مجاور است، و بنابراین دور مورد نظر تمام رأس‌ها در هر یک از widgetهای G' را ملاقات می‌کند. چون این دور تمام رأس‌های انتخاب‌گر را هم ملاقات می‌کند، پس همیلتونی است.

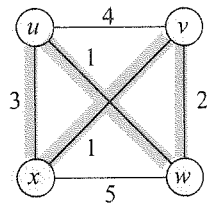
برعکس، فرض کنید که $G'=(V',E')$ یک دور همیلتونی $C\subseteq E'$ دارد. ادعا می‌کنیم که مجموعه‌ی

$$V^*=\{u\in V:\ 1\le j\le k \text{ برای یک } (s_j,[u,u^{(1)},1])\in C\} \qquad (۴-۳۴)$$

یک پوشش رأسی برای G است. برای این که ببینیم چرا، C را به مسیرهای ماکسیمال تقسیم می‌کنیم که از یک رأس انتخاب‌گر s_i آغاز می‌شوند، از یک یال $(s_i,[u,u^{(1)},1])$ برای یک $u\in V$ عبور می‌کنند، و در یک رأس انتخاب‌گر s_j پایان می‌یابند، بدون این که از هیچ رأس انتخاب‌گر دیگری بگذرند. اجازه دهید هر مسیر مانند این را یک «مسیر پوششی» بنامیم. از نحوه‌ی ساخت G'، هر مسیر پوششی باید از یک s_i آغاز شود، از یال $(s_i,[u,u^{(1)},1])$ برای یک $u\in V$ عبور کند، از تمام widgetهای متناظر با یال‌های مجاور u در E بگذرد، و در یک رأس انتخاب‌گر s_j پایان یابد. این مسیر پوششی را p_u می‌نامیم، و طبق تساوی (۴-۳۴)، u را در V^* قرار می‌دهیم. هر widet ملاقات شده توسط p_u یا باید W_{uv} باشد و یا W_{vu} برای یک $v\in V$. برای هر widget ملاقات شده توسط p_u، رأس‌های آن توسط یک یا دو مسیر پوششی، ملاقات می‌شوند. اگر این رأس‌ها توسط یک مسیر پوششی ملاقات شوند، یال $(u,v)\in E$ در G توسط رأس u پوشش داده می‌شود. اگر دو مسیر پوششی widget را ملاقات کنند، آن گاه مسیر پوششی دیگر باید p_v باشد، که نتیجه می‌دهد که $v\in V^*$، و یال $(u,v)\in E$ توسط هر دو رأس u و v پوشش داده شده است. چون هر رأس در هر widget توسط یک مسیر پوششی ملاقات می‌شود، می‌بینیم که هر یال در E توسط یک رأس در V^* پوشش داده خواهد شد. ■

۴-۵-۳۴ مسئله‌ی فروشنده‌ی دوره‌گرد

در *مسئله‌ی فروشنده‌ی دوره‌گرد* (traveling salesman problem)، که رابطه‌ی نزدیکی با مسئله‌ی دور همیلتونی دارد، یک فروشنده باید تمام n شهر را ملاقات کند. با مدل کردن مسئله به صورت یک گراف کامل با n رأس، می‌توانیم بگوییم که فروشنده می‌خواهد یک *تور* (tour)، یا یک دور همیلتونی را تشکیل دهد، که در آن تمام شهرها را دقیقاً یک بار ملاقات کرده و در نهایت بازمی‌گردد به شهری که از آن شروع کرده است. یک هزینه‌ی $c(i,j)$ برای انتقال از شهر i به شهر j وجود دارد، که یک عدد



شکل ۳۴-۱۸ یک نمونه از مسئله‌ی فروشنده‌ی دوره‌گرد. یال‌های سایه‌دار یک تور با هزینه‌ی حداقل را نشان می‌دهند، که هزینه‌ی آن ۱۷ است.

صحیح است، و فروشنده می‌خواهد از توری عبور کند که هزینه‌ی کلی آن حداقل باشد، که در آن هزینه‌ی کل برابر است با مجموع هزینه‌ی یال‌های تور. برای مثال در شکل ۳۴-۱۸، یک تور با هزینه‌ی حداقل عبارت است از $\langle u, w, v, x, u \rangle$، با هزینه‌ی ۷. زبان فرمال برای مسئله‌ی تصمیم‌گیری متناظر عبارت است از

$$\text{TSP} = \left\{ \langle G, c, k \rangle \;\middle|\; \begin{array}{l} G = (V, E) \text{ یک گراف کامل است، } c \text{ یک تابع است} \\ \text{از } V \times V \to \mathbb{Z}, \; k \in \mathbb{Z}, \text{ و } G \text{ یک تور فروشنده‌ی دوره} \\ \text{گرد دارد، با حداکثر هزینه‌ی } k \end{array} \right\}$$

قضیه‌ی زیر نشان می‌دهد که احتمالاً یک الگوریتم سریع برای مسئله‌ی فروشنده‌ی دوره‌گرد وجود ندارد.

قضیه‌ی ۳۴-۱۴ مسئله‌ی فروشنده‌ی دوره‌گرد (TSP)، NP-کامل است.

اثبات ابتدا نشان می‌دهیم که TSP به NP تعلق دارد. با داشتن یک نمونه از مسئله، از یک دنباله از n رأس در تور به عنوان یک تصدیق استفاده می‌کنیم. الگوریتم تحقیق، بررسی می‌کند که هر رأس دقیقاً یک بار در این دنباله ظاهر شود، مجموع هزینه‌ی یال‌ها را با هم جمع می‌کند، و چک می‌کند که این مجموع حداکثر برابر k باشد. این فرآیند را به سادگی می‌توان در زمان چندجمله‌ای انجام داد.

برای اثبات این که مسئله‌ی فروشنده‌ی دوره‌گرد NP- سخت است، نشان می‌دهیم که HAM-CYCLE $\leq_P$ TSP. فرض کنید $G = (V, E)$ یک نمونه از HAM-CYCLE باشد. یک نمونه از TSP را به صورت زیر می‌سازیم. گراف کامل $G' = (V, E')$ را تشکیل می‌دهیم، که در آن $E' = \{(i, j) : i, j \in V \text{ و } i \neq j\}$، و تابع هزینه‌ی c را به صورت زیر تعریف می‌کنیم:

$$c(i, j) = \begin{cases} 0 & \text{اگر } (i, j) \in E \\ 1 & \text{اگر } (i, j) \notin E \end{cases}$$

(توجه کنید که چون G بدون جهت است، هیچ طوقه‌ای ندارد، و بنابراین $c(v, v) = 1$ برای تمام رأس‌های $v \in V$.) بنابراین نمونه‌ی TSP عبارت است از $(G', c, 0)$، که به سادگی در زمان چندجمله‌ای

تشکیل می‌شود.

اکنون نشان می‌دهیم که گراف G یک دور همیلتونی دارد اگر و فقط اگر گراف G' یک تور با هزینه‌ی حداکثر ۰ داشته باشد. فرض کنید که گراف Gیک دور همیلتونی h داشته باشد. هر یال در h به E تعلق دارد، و بنابراین هزینه‌ی آن‌ها ۰ در G' است. بنابراین h یک تور در G' است با هزینه‌ی ۰. بالعکس، فرض کنید که گراف G' یک تور h' با هزینه‌ی حداکثر ۰ داشته باشد. چون هزینه‌ی یال‌ها در E' یا ۰ هستند و یا ۱، هزینه‌ی تور h' دقیقاً ۰ است، و هزینه‌ی هر یال در تور باید ۰ باشد. بنابراین h' تنها شامل یال‌های E است. نتیجه می‌گیریم که h' یک دور همیلتونی در گراف G است. ■

۳۴-۵-۵ مسئله‌ی جمع زیرمجموعه

مسئله‌ی NP-کامل بعدی که در نظر می‌گیریم یک مسئله‌ی ریاضی است. در *مسئله‌ی جمع زیرمجموعه* (subset-sum problem)، به ما یک مجموعه‌ی متناهی $S \subset \mathbb{N}$ و یک هدف $t \in \mathbb{N}$ داده می‌شود. می‌خواهیم بدانیم که آیا زیرمجموعه‌ای مانند $S' \subseteq S$ وجود دارد که مجموع عناصر آن t باشد یا خیر. برای مثال، اگر $S = \{1,2,7,14,49,98,343,686,2409,2793,16808,17206,117705,117993\}$ و $t = 138457$، آن گاه زیرمجموعه‌ی $S' = \{1,2,7,98,343,686,2409,17206,117705\}$ یک جواب است. مانند همیشه، مسئله را به صورت یک زبان تعریف می‌کنیم:

$$\text{SUBSET-SUM} = \{\langle S,t\rangle : t = \sum_{s \in S'} s \text{ که } S' \subseteq S \text{ وجود دارد به طوری } \}$$

مانند هر مسئله‌ی ریاضی، مهم است به خاطر داشته باشیم که کدگذاری استاندارد ما فرض می‌کند که اعداد ورودی به صورت دودویی کد شده‌اند. با در نظر داشتن این فرض، می‌توانیم نشان دهیم که مسئله‌ی جمع زیرمجموعه احتمالاً یک الگوریتم سریع ندارد.

قضیه‌ی ۱۵-۳۴ مسئله‌ی جمع زیرمجموعه NP-کامل است.

اثبات برای نشان دادن این که SUBSET-SUM در NP است، برای یک نمونه‌ی $\langle S,t\rangle$ از مسئله، فرض می‌کنیم زیرمجموعه‌ی S' یک تصدیق باشد. چک کردن $t = \sum_{s \in S'} s$ را می‌توان در زمان چندجمله‌ای انجام داد.

اکنون نشان می‌دهیم که 3-CNF-SAT $\leq_P$ SUBSET-SUM. با داشتن یک فرمول φ که 3-CNF است بر روی متغیرهای $x_1, x_2, \ldots, x_n$ با عبارت‌های $C_1, C_2, \ldots, C_k$ که هر یک دقیقاً سه لفظ مجزا دارند، الگوریتم کاهش یک نمونه‌ی $\langle S,t\rangle$ از مسئله‌ی جمع زیرمجموعه می‌سازد به طوری که φ قابل ارضا باشد اگر و فقط اگر یک زیرمجموعه از S وجود داشته باشد به طوری که مجموع آن دقیقاً t است. بدون از دست دادن کلیت، دو فرض ساده‌کننده در مورد فرمول φ در نظر می‌گیریم. اول این که هیچ عبارتی هم زمان حاوی یک متغیر و نقیض آن نیست، چرا که چنین عبارتی با هر مقداردهی به متغیرها

قابل ارضا است. دوم این که هر متغیر حداقل در یک عبارت ظاهر می‌شود، چرا که در غیر این صورت مقدار داده شده به آن اهمیت نخواهد داشت.

الگوریتم کاهش برای هر متغیر x_i و همچنین برای هر عبارت C_j دو عدد در مجموعه‌ی S می‌سازد. اعداد را در پایه‌ی ۱۰ می‌سازیم، که در آن هر عدد $n+k$ رقم دارد، و هر رقم یا متناظر است با یک متغیر، و یا با یک عبارت. پایه‌ی ۱۰ (و پایه‌های دیگر، همان طور که خواهیم دید) این خاصیت مناسب را دارد که از انتقال رقم نقلی از ارقام پایین‌تر به ارقام بالاتر جلوگیری می‌کند.

همان طور که شکل ۳۴-۱۹ نشان می‌دهد، مجموعه‌ی S و هدف t را به صورت زیر می‌سازیم. مکان هر رقم را یا با یک متغیر و یا با یک عبارت برچسب‌گذاری می‌کنیم. k رقم کم‌ارزش با عبارت‌ها برچسب‌گذاری می‌شود، و n رقم پرارزش با متغیرها.

- در هر مکان از هدف t که با یک متغیر برچسب‌گذاری شده است، یک ۱، و در هر مکان که با یک عبارت برچسب‌گذاری شده است، یک ۴ قرار می‌دهیم.

		x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	=	1	0	0	1	0	0	1
v'_1	=	1	0	0	0	1	1	0
v_2	=	0	1	0	0	0	0	1
v'_2	=	0	1	0	1	1	1	0
v_3	=	0	0	1	0	0	1	1
v'_3	=	0	0	1	1	1	0	0
s_1	=	0	0	0	1	0	0	0
s'_1	=	0	0	0	2	0	0	0
s_2	=	0	0	0	0	1	0	0
s'_2	=	0	0	0	0	2	0	0
s_3	=	0	0	0	0	0	1	0
s'_3	=	0	0	0	0	0	2	0
s_4	=	0	0	0	0	0	0	1
s'_4	=	0	0	0	0	0	0	2
t	=	1	1	1	4	4	4	4

شکل ۳۴-۱۹ کاهش 3-CNF به SUBSET-SUM. فرمول 3-CNF عبارت است از $\varphi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$، که در آن $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$، $C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$، $C_3 = (\neg x_1 \vee \neg x_2 \vee x_3)$، و $C_4 = (x_1 \vee x_2 \vee x_3)$. یک مقداردهی ارضا کننده به φ عبارت است از $\langle x_1 = 0, x_2 = 0, x_3 = 1 \rangle$. مجموعه‌ی S ساخته شده توسط کاهش عبارت است از اعداد پایه‌ی ۱۰ نشان داده شده؛ از بالا به پایین داریم $S = \{1001001, 1000110, 100001, 101110, 10011, 11100, 1000, 2000, 100, 200, 10, 20, 1, 2\}$. مقدار هدف ۱۱۱۴۴۴۴ است. زیرمجموعه‌ی $S' \subseteq S$ با سایه‌ی کم‌رنگ مشخص شده است، و حاوی v'_1، v_2، و v_3 است، متناظر با مقداردهی ارضا کننده، و همچنین شامل متغیرهای قطع s_1، s'_1، s_2، s'_2، s_3، s_4، و s'_4 برای رسیدن به مقدار ۴ در رقم‌های با برچسب C_1 تا C_4.

- برای هر متغیر x_i دو عدد صحیح v_i و v_i' در S وجود دارد. هر یک از این اعداد، در مکان علامت‌گذاری شده با x_i یک ۱ دارند، و ۰ در مکان بقیه‌ی متغیرها. اگر متغیر x_i در عبارت C_j ظاهر شود، آن گاه رقم علامت‌گذاری شده با C_j در v_i حاوی ۱ خواهد بود. اگر لفظ $\neg x_i$ در عبارت C_j ظاهر شود، آن گاه رقم علامت‌گذاری شده با C_j در v_i' حاوی ۱ است. تمام ارقام علامت‌گذاری شده با عبارت‌های دیگر در v_i و v_i'، ۰ هستند.

  تمام مقادیر v_i و v_i' در مجموعه‌ی S یکتا هستند. چرا؟ برای $l \neq i$، هیچ کدام از v_l یا v_l' نمی‌تواند در ارقام پرارزش برابر با v_i یا v_i' باشد. به علاوه با فرض ساده‌کننده‌ی بالا، نمی‌توان یک v_i و v_i' پیدا کرد که تمام k رقم کم‌ارزش آن‌ها با هم برابر باشد. اگر v_i و v_i' برابر باشند، آن گاه x_i و $\neg x_i$ باید دقیقاً در یک مجموعه عبارت وجود داشته باشند. ولی فرض کردیم که هیچ عبارتی نمی‌تواند شامل هر دوی x_i و $\neg x_i$ باشد، و این که یا x_i و یا $\neg x_i$ باید در حداقل یک عبارت ظاهر شود، و بنابراین باید یک عبارت C_j وجود داشته باشد که برای آن مقادیر v_i و v_i' متفاوت است.
- برای هر عبارت C_j دو عدد صحیح s_j و s_j' در S وجود دارد. برای هر یک از این اعداد تمام ارقام ۰ است، غیر از رقمی که با C_j علامت‌گذاری شده است. برای s_j، یک ۱ در قم C_j وجود دارد، و همان رقم در s_j'، ۲ است. این اعداد، «متغیرهای قطع» هستند، که از آن‌ها برای این استفاده می‌کنیم که هر مکان علامت‌گذاری شده با عبارت‌ها را در مقدار هدف به ۴ برسانیم.

با نگاهی به شکل ۳۴-۱۹ می‌بینیم که تمام مقادیر s_j و s_j' در S یکتا هستند.

توجه کنید که بزرگ‌ترین مجموع در هر مکانی ۶ است، که در ارقام با برچسب عبارت‌ها رخ می‌دهد (سه ۱ از مقادیر v_i و v_i'، به علاوه‌ی ۱ و ۲ از مقادیر s_j و s_j'). بنابراین با تفسیر این اعداد در مبنای ۱۰، هیچ رقم نقلی از ارقام پایین به ارقام بالا نخواهیم داشت.[1]

این کاهش را می‌توان در زمان چندجمله‌ای انجام داد. مجموعه‌ی S حاوی $2n+2k$ مقدار است، که هر یک $n+k$ رقم دارند، و زمان تولید هر رقم نسبت به $n+k$ از مرتبه‌ی چندجمله‌ای است. هدف $n+k$ رقم دارد، و الگوریتم کاهش هر یک از این ارقام را در زمان چندجمله‌ای می‌سازد.

اکنون نشان می‌دهیم که فرمول φ قابل ارضا است اگر و فقط اگر یک زیرمجموعه‌ی $S' \subseteq S$ وجود داشته باشد که مجموع آن t باشد. اول، فرض کنید که φ یک مقداردهی ارضا کننده است. برای $i = 1,2,..,n$، اگر $x_i = 1$، آن گاه v_i را در S' قرار می‌دهیم، و در غیر این صورت v_i' را. به عبارت دیگر در S' دقیقاً مقادیر v_i و v_i' را قرار می‌دهیم که متناظر با مقدار ۱ در مقداردهی ارضا کننده هستند. با قرار دادن یکی از v_i یا v_i'، ولی نه هر دو، برای هر i، و همچنین قرار دادن ۰ در ارقام

[1] در واقع هر مبنای b، که $b \geq 7$ برای این کار مناسب خواهد بود. نمونه‌ی ابتدای این زیربخش، مجموعه‌ی S است و هدف t در شکل ۳۴-۱۹ که در مبنای ۷ تفسیر شده است، و S به صورت مرتب لیست شده است.

با برچسب متغیرها در تمام s_j و s'_jها، می‌بینیم که برای هر رقم با برچسب متغیر، مجموع مقادیر S' باید ۱ باشد، که با ارقام متناظر در هدف t یکسان است. چون تمام عبارت‌ها ارضا شده‌اند، پس در هر عبارت حداقل یک لفظ با مقدار ۱ وجود دارد. بنابراین هر رقم که با یک عبارت برچسب‌گذاری شده است، حداقل یک ۱ در مجموع خود دارد، که توسط v_i یا v'_i در S' آورده می‌شود. در واقع در هر عبارت ممکن است ۱، ۲، یا ۳ عبارت ۱ شود، و بنابراین هر رقم با برچسب عبارت در هدف مقدار ۱، ۲، یا ۳ دارد. (مثلاً در شکل ۳۴-۱۹، لفظ‌های $\neg x_1$، $\neg x_2$، و $\neg x_3$ در یک مقداردهی ارضا کننده مقدار ۱ دارند. هر یک از عبارت‌های C_1 و C_4 دقیقاً حاوی یکی از این لفظ‌ها هستند، و بنابراین v'_1، v'_2، و v_3 یک ۱ به مجموع برای C_1 و C_4 اضافه می‌کنند. عبارت C_2 شامل دو تا از این عبارت‌ها است، و v'_1، v'_2، و v_3 یک ۲ به مجموع در ارقام برای C_2 اضافه می‌کنند. عبارت C_3 شامل تمام این سه عبارت است، و v'_1، v'_2، و v_3 یک ۳ به مجموع در ارقام برای C_3 اضافه می‌کنند.) در هر رقم با برچسب C_j، با شامل شدن زیرمجموعه‌ی غیر تهی مناسب از متغیرهای قطع $\{s_j, s'_j\}$، در هدف به مقدار ۴ می‌رسیم. (در شکل ۳۴-۱۹، S' شامل s_1، s'_1، s'_2، s_3، s_4، و s'_4 است.) از آن جایی که تمام ارقام مجموع با هدف یکسان شده است، و هیچ رقم نقلی رخ نمی‌دهد، مجموع مقادیر S' برابر با t خواهد بود.

اکنون فرض کنید که یک زیرمجموعه‌ی $S' \subseteq S$ وجود دارد که مجموع اعضای آن t است. زیرمجموعه‌ی S' باید دقیقاً شامل یکی از v_i و v'_i باشد، برای هر $i = 1,2,...,n$، چرا که در غیر این صورت ارقام با برچسب متغیر برابر با ۱ نخواهند شد. اگر $v_i \in S'$، قرار می‌دهیم $x_i = 1$. در غیر این صورت $v'_i \in S'$، و قرار می‌دهیم $x_i = 0$. ادعا می‌کنیم که تمام عبارت‌های C_j، برای $j = 1,2,...,k$ توسط این مقداردهی ارضا می‌شوند. برای اثبات این ادعا، توجه کنید که برای دست‌یابی به یک مجموع ۴ در رقم با برچسب C_j، زیرمجموعه‌ی S' باید شامل حداقل یکی از مقادیر v_i یا v'_i که رقم با برچسب C_j آن ۱ است، باشد، چرا که مجموع ارقام متغیرهای قطع حداکثر ۳ است. اگر S' شامل یک v_i باشد که مکان مورد نظر آن ۱ است، آن گاه لفظ x_i در عبارت C_j ظاهر می‌شود. چون قرار داده‌ایم $x_i = 1$ وقتی $v_i \in S'$، عبارت C_j ارضا شده است. اگر S' شامل یک v'_i باشد که مکان مورد نظر آن ۱ است، آن گاه جمله‌ی $\neg x_i$ در C_j وجود دارد. چون قرار داده‌ایم $x_i = 0$ وقتی $v'_i \in S'$، عبارت C_jباز هم ارضا می‌شود. بنابراین تمام عبارات φ ارضا می‌شوند، که اثبات را کامل می‌کند. ■

تمرین‌ها

۳۴-۵-۱ در مسئله‌ی هم‌ریختی زیرگراف (subgraph-isomorphism problem) برای دو گراف G_1 و G_2، می‌خواهیم بدانیم که آیا G_1 با یک زیرگراف از G_2 هم‌ریخت است یا نه. نشان دهید که مسئله‌ی هم‌ریختی زیرگراف NP-کامل است.

۳۴-۵-۲ با داشتن یک ماتریس A از اعداد صحیح با اندازه‌ی m در n و یک بردار mتایی b از اعداد صحیح، در *مسئله‌ی برنامه‌ریزی صحیح ۰-۱* (0-1 integer-programming problem) می‌خواهیم بدانیم که آیا یک بردار nتایی x بر روی مجموعه‌ی $\{0,1\}$ وجود دارد به طوری که $Ax \le b$ یا خیر. اثبات کنید که مسئله‌ی برنامه‌ریزی صحیح ۰-۱، NP-کامل است. (*راهنمایی:* کاهش را از 3-CNF انجام دهید.)

۳۴-۵-۳ *مسئله‌ی برنامه‌ریزی خطی صحیح* (integer linear-programming problem) مانند مسئله‌ی برنامه‌ریزی صحیح ۰-۱ داده شده در مسئله‌ی قبل است، غیر از این که مقادیر بردار x می‌توانند هر عدد صحیحی باشند، و نه فقط ۰ و ۱. فرض کنید که مسئله‌ی برنامه‌ریزی صحیح ۰-۱، NP-سخت است. نشان دهید که مسئله‌ی برنامه‌ریزی خطی صحیح NP-کامل است.

۳۴-۵-۴ نشان دهید که چطور می‌توان مسئله‌ی جمع زیرمجموعه را در زمان چندجمله‌ای در حالتی حل کرد که مقدار هدف t به صورت یگانی (مبنای یک) توصیف شده است.

۳۴-۵-۵ *مسئله‌ی تقسیم‌بندی مجموعه* (set-partition problem) به عنوان ورودی یک مجموعه‌ی S از اعداد را دریافت می‌کند. سؤال این است که آیا می‌توان این اعداد را به دو مجموعه‌ی A و $\overline{A} = S - A$ تقسیم کرد به طوری که $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$ یا نه. نشان دهید که مسئله‌ی تقسیم‌بندی مجموعه NP-کامل است.

۳۴-۵-۶ نشان دهید که مسئله‌ی مسیر همیلتونی NP-کامل است.

۳۴-۵-۷ در *مسئله‌ی بلندترین دور ساده* (longest-simple-cycle problem) می‌خواهیم یک دور ساده (بدون رأس تکراری) با طول بیشینه را در یک گراف بیابیم. یک مسئله‌ی تصمیم‌گیری متناظر ارائه کرده و نشان دهید که این مسئله تصمیم‌گیری NP-کامل است.

۳۴-۵-۸ در مسئله‌ی *قابلیت ارضای نیمه‌ی 3-CNF* (half 3-CNF satisfiability)، به ما یک فرمول φ با n متغیر و m عبارت داده می‌شود که 3-CNF است، که در آن m زوج است. می‌خواهیم تعیین کنیم که آیا یک مقداردهی صحیح به متغیرهای φ وجود دارد به طوری که دقیقاً مقدار نیمی از عبارت‌ها ۰ و مقدار نیم دیگر ۱ شود. اثبات کنید که مسئله‌ی قابلیت ارضای نیمه‌ی 3-CNF، NP-کامل است.

مسائل

۳۴-۱ مجموعه‌ی مستقل

یک *مجموعه‌ی مستقل* (independent set) از یک گراف $G = (V, E)$ یک زیرمجموعه‌ی $V' \subseteq V$ از رأس‌ها است به طوری که هر یال در E حداکثر با یک رأس از V' همسایه است. در *مسئله‌ی مجموعه‌ی مستقل* (independent set problem) می‌خواهیم یک زیرمجموعه‌ی مستقل با

اندازه‌ی بیشینه در G بیابیم.

I. یک مسئله‌ی تصمیم‌گیری متناظر برای مسئله‌ی مجموعه‌ی مستقل تعریف کنید، و اثبات کنید که NP-کامل است. (*راهنمایی:* کاهش را از مسئله‌ی گروهک انجام دهید.)

II. فرض کنید که به ما یک زیرروال «جعبه‌ی سیاه» برای حل مسئله‌ی تصمیم‌گیری که در بخش (الف) تعریف کردید، داده شده است. یک الگوریتم برای یافتن مجموعه‌ی مستقل با اندازه‌ی بیشینه بدهید. زمان اجرای الگوریتم شما باید نسبت به $|V|$ و $|E|$ از مرتبه‌ی چندجمله‌ای باشد، که در آن فراخوانی‌های جعبه‌ی سیاه به صورت یک مرحله در نظر گرفته می‌شوند.

با این که مسئله‌ی تصمیم‌گیری مجموعه‌ی مستقل NP-کامل است، حالت‌های خاصی از آن وجود دارد که در زمان چندجمله‌ای قابل حل هستند.

III. یک الگوریتم کارامد برای حل مسئله‌ی مجموعه‌ی مستقل در حالتی ارائه کنید که در آن درجه‌ی هر رأس در G، ۲ است. زمان اجرای الگوریتم خود را تحلیل و درستی آن را اثبات کنید.

IV. یک الگوریتم کارامد برای حل مسئله‌ی مجموعه‌ی مستقل در حالتی ارائه دهید که G دوبخشی است. زمان اجرای الگوریتم خود را تحلیل و درستی آن را اثبات کنید. (*راهنمایی:* از نتایج بخش ۲۶-۳ استفاده کنید.)

۳۴-۲ *Bonnie* و *Clyde*[1]

Bonnie و Clyde از یک بانک سرقت کرده‌اند. آن‌ها یک کیف پر از پول دارند و می‌خواهند آن را تقسیم کنند. برای هر یک از سناریوهای زیر، یا یک الگوریتم چندجمله‌ای ارائه کنید، و یا اثبات کنید که مسئله NP-کامل است. ورودی در هر حالت یک لیست از n شیئ در کیف است، به همراه قیمت هر یک از آن‌ها.

I. n سکه داریم، که فقط از دو نوع مختلف هستند: بعضی x دلاری، و بعضی y دلاری. آن‌ها می‌خواهند پول را به صورت دقیقاً مساوی تقسیم کنند.

II. n سکه داریم، با تعداد دلخواهی نوع مختلف، ولی هر نوع یک توان نامنفی از ۲ است، یعنی انواع مختلفی که داریم عبارتند از ۱ دلاری، ۲ دلاری، ۴ دلاری، و الی آخر. آن‌ها می‌خواهند پول را به صورت دقیقاً مساوی تقسیم کنند.

III. n چک وجود دارد، که به شکلی شگفت‌آور و به صورت تصادفی در وجه «Bonnie یا Clyde» نوشته شده‌اند. آن‌ها می‌خواهند طوری چک‌ها را تقسیم کنند که هر یک دقیقاً به یک مقدار پول دریافت کنند.

IV. n چک داریم، مانند بخش III، با این تفاوت که این بار آن‌ها می‌خواهند طوری چک‌ها را تقسیم کنند که تفاوت میان سهم هر یک بیش از ۱۰۰ دلار نباشد.

---

[1] Bonnie و Clyde دو مجرم معروف آمریکایی بودند که در دهه‌ی ۱۹۲۰ مرتکب خلاف‌های بسیاری از جمله دزدی و قتل قتل شده‌اند - م

۳-۳۴ رنگ‌آمیزی گراف

سازندگان نقشه هنگام رنگ‌آمیزی کشورها روی یک نقشه سعی می‌کنند از کم‌ترین تعداد رنگ ممکن استفاده کنند، ولی هیچ دو کشوری که مرز مشترک دارند نباید رنگ یکسان داشته باشند. می‌توان این مسئله را با یک گراف بدون جهت $G = (V, E)$ مدل کرد که در آن هر رأس نشان دهنده‌ی یک کشور است، و اگر دو کشور مرز مشترک داشته باشند، رأس‌های متناظر آن‌ها همسایه هستند. در این صورت یک *رنگ‌آمیزی k تایی* (k-coloring) یک تابع $c : V \to \{1, 2, ..., k\}$ است به طوری که برای هر یال $(u, v) \in E$ داشته باشیم $c(u) \neq c(v)$. به عبارت دیگر اعداد $1, 2, ..., k$ نشان‌دهنده‌ی k رنگ هستند، و رأس‌های مجاور باید رنگ‌های متفاوت داشته باشند. در *مسئله‌ی رنگ‌آمیزی گراف* می‌خواهیم کم‌ترین تعداد رنگ‌های موردنیاز برای رنگ‌آمیزی یک گراف داده شده را تعیین کنیم.

I. یک الگوریتم کارامد برای تعیین یک رنگ‌آمیزی ۲تایی برای یک گراف (در صورت وجود) طراحی کنید.

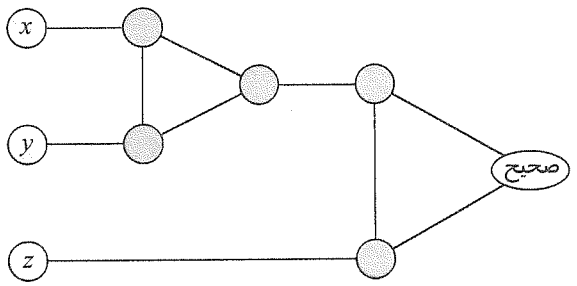
II. مسئله‌ی رنگ‌آمیزی گراف را به صورت یک مسئله‌ی تصمیم‌گیری تعریف کنید. نشان دهید که مسئله‌ی تصمیم‌گیری شما در زمان چندجمله‌ای قابل حل است اگر و فقط اگر مسئله‌ی رنگ‌آمیزی گراف قابل حل در زمان چندجمله‌ای باشد.

III. فرض کنید زبان 3-COLOR مجموعه‌ی گراف‌هایی باشد که بتوان آن‌ها را با سه رنگ، رنگ‌آمیزی کرد. نشان دهید که اگر 3-COLOR، NP–کامل باشد، آن گاه مسئله‌ی تصمیم‌گیری شما در بخش II هم NP–کامل است.

برای اثبات این که 3-COLOR، NP–کامل است، از یک کاهش از 3-CNF-SAT استفاده می‌کنیم. با داشتن یک فرمول φ با m عبارت و n متغیر $x_1, x_2, ..., x_n$، یک گراف $G = (V, E)$ به صورت زیر می‌سازیم. مجموعه‌ی V شامل رأس‌های زیر است: یک رأس برای هر متغیر، یک رأس برای نقیض هر متغیر، ۵ رأس برای هر عبارت، و ۳ رأس ویژه: TRUE، FALSE، و RED. یال‌های گراف از دو نوع هستند: یال‌های «لفظ» مستقل از عبارت‌ها هستند، و یال‌های «عبارت» که وابسته به عبارت‌ها هستند. یال‌های لفظ یک مثلث برای رأس‌های ویژه تشکیل می‌دهند، و همچنین یک مثلث بر روی x_i، $\neg x_i$، و RED برای $i = 1, 2, ..., n$.

IV. بحث کنید که در هر رنگ‌آمیزی سه‌تایی c برای یک گراف حاوی یال‌های لفظ، از بین یک متغیر و نقیض آن، دقیقاً یکی با c(TRUE) رنگ‌آمیزی می‌شود، و دیگری با c(FALSE). بحث کنید که برای هر مقداردهی صحیح به φ، یک رنگ‌آمیزی ۳تایی برای گراف فقط حاوی یال‌های لفظ وجود دارد.

از widget نشان داده شده در شکل ۳۴-۲۰ برای به اجرا در آوردن شرط متناظر با عبارت $(x \vee y \vee z)$ استفاده شده است. هر عبارت به یک کپی یکتا از ۵ رأسی نیاز دارد که در شکل با سایه‌ی پررنگ نشان داده شده‌اند؛ آن‌ها به صورت نشان داده شده به لفظ‌های عبارت و



شکل ۳۴-۳ widget متناظر با عبارت $(x \vee y \vee z)$ که از آن در مسئله‌ی ۳۴-۳ استفاده شده است.

رأس ویژه‌ی TRUE متصل شده‌اند.

۷ بحث کنید که اگر x، y، و z با c(TRUE) یا c(FALSE) رنگ‌آمیزی شوند، آن گاه widget با سه رنگ قابل رنگ‌آمیزی است اگر و فقط اگر حداقل یکی از رأس‌های x، y، یا z با c(TRUE) رنگ‌آمیزی شوند.

اثبات NP-کامل بودن 3-COLOR را کامل کنید.

۳۴-۴ *برنامه‌ریزی با سود و مهلت انجام*

فرض کنید که یک ماشین دارید و یک مجموعه از n وظیفه‌ی $a_1, a_2, \ldots, a_n$ که باید به وسیله‌ی این ماشین انجام شوند. هر وظیفه به t_j واحد زمان روی ماشین نیاز دارد، انجام آن p_j واحد سود دارد، و باید حداکثر تا زمان d_j انجام شود. ماشین در هر زمان فقط می‌تواند یک کار انجام دهد، و وظیفه‌ی a_j باید بدون وقفه برای t_j واحد زمان اجرا شود. اگر وظیفه‌ی a_j را در مهلت d_j انجام دهیم، p_j واحد سود دریافت می‌کنیم، و اگر آن را بعد از مهلت انجام به پایان برسانیم، هیچ سودی دریافت نمی‌کنیم. مانند یک مسئله‌ی بهینه‌سازی، زمان‌های پردازش، سودها، و مهلت‌ها برای مجموعه‌ای از n وظیفه به ما داده شده است، و می‌خواهیم برنامه‌ای بیابیم که تمام وظایف را انجام داده و بیشترین سود را به ما می‌رساند. زمان‌های پردازش، سودها و مهلت‌ها همگی اعداد صحیح نامنفی هستند.

I این مسئله را به صورت یک مسئله‌ی تصمیم‌گیری توصیف کنید.

II نشان دهید که این مسئله‌ی تصمیم‌گیری NP-کامل است.

III یک الگوریتم چندجمله‌ای برای مسئله‌ی تصمیم‌گیری ارائه کنید، با این فرض که تمام زمان‌های پردازش، اعدادی صحیح در بازه‌ی ۱ تا n هستند. (*راهنمایی:* از برنامه‌ریزی پویا استفاده کنید.)

IV یک الگوریتم چندجمله‌ای برای مسئله‌ی بهینه‌سازی ارائه کنید، با این فرض که تمام زمان‌های پردازش، اعدادی صحیح در بازه‌ی ۱ تا n هستند.

فصل

۳۵

الگوریتم‌های تقریبی

بسیاری از مسائل در حوزه‌ی کاربردی NP-کامل هستند، ولی مهم‌تر از آن هستند که کاملاً آن‌ها را رها کنیم چون نمی‌توانیم یک جواب بهینه برای آن‌ها بیابیم. اگر یک مسئله NP-کامل باشد، احتمالاً نمی‌توانیم یک الگوریتم چندجمله‌ای برای حل دقیق آن بیابیم، ولی با این همه، ممکن است بتوانیم امیدوار باشیم. حداقل سه رویکرد برای دست و پنجه نرم کردن با NP-کامل‌ها وجود دارد. اول، اگر در عمل ورودی‌ها کوچک باشند، یک الگوریتم با زمان نمایی ممکن است کاملاً مناسب باشد. دوم، ممکن است بتوانیم حالت‌های خاص و مهمی بیابیم که در زمان چندجمله‌ای قابل حل‌اند. سوم، ممکن است بتوانیم یک جواب *نزدیک بهینه* در زمان چندجمله‌ای (در بدترین حالت یا در حالت متوسط) بیابیم. در عمل، جواب‌های نزدیک بهینه معمولاً به اندازه‌ی کافی خوب هستند. یک الگوریتم که جواب‌های نزدیک بهینه میابد، یک *الگوریتم تقریبی* (approximation algorithm) نام دارد. این فصل برای چند مسئله‌ی NP-کامل الگوریتم‌های تقریبی ارائه می‌کند.

نسبت کارایی برای الگوریتم‌های تقریبی

فرض کنید داریم بر روی یک مسئله‌ی بهینه‌سازی کار می‌کنیم که در آن هر جواب احتمالی یک هزینه‌ی مثبت دارد، و می‌خواهیم یک جواب نزدیک بهینه برای آن بیابیم. بسته به مسئله، یک جواب بهینه ممکن است به صورت جواب با هزینه‌ی بیشینه تعریف شود، و یا به صورت جواب با هزینه‌ی کمینه؛ یعنی، ممکن است یک مسئله‌ی بیشینه‌سازی و یا یک مسئله‌ی کمینه‌سازی داشته باشیم.

می‌گوییم یک الگوریتم برای یک مسئله، یک *نسبت کارایی* (approximation ratio) $\rho(n)$ دارد اگر برای هر ورودی با اندازه‌ی n، هزینه‌ی C مربوط به جواب تولید شده توسط الگوریتم درون یک فاکتور $\rho(n)$ از هزینه‌ی C^* مربوط به جواب بهینه باشد:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n) \qquad (۱-۳۵)$$

همچنین به الگوریتمی که به نسبت کارایی $\rho(n)$ می‌رسد، یک *الگوریتم $\rho(n)$-تقریبی* می‌گوییم. تعریف نسبت کارایی و الگوریتم $\rho(n)$-تقریبی برای هر دو نوع مسئله‌ی بیشینه‌سازی و کمینه‌سازی به کار می‌رود. برای یک مسئله‌ی بیشینه‌سازی، $0 < C \le C^*$، و هزینه‌ی یک جواب بهینه با نسبت C^*/C از هزینه‌ی یک جواب تقریبی بزرگ‌تر است. به طور مشابه برای یک مسئله‌ی کمینه‌سازی، $0 < C^* \le C$، و هزینه‌ی یک جواب تقریبی با نسبت C/C^* از هزینه‌ی یک جواب بهینه بزرگ‌تر است. از آن جایی که فرض می‌کنیم که هزینه‌ی تمام جواب‌ها مثبت است، این نسبت‌ها همیشه تعریف شده‌اند. نسبت کارایی یک الگوریتم تقریبی هیچ‌گاه کوچک‌تر از ۱ نیست، چرا که $C/C^* > 1$ نتیجه می‌دهد $C^*/C < 1$. بنابراین، یک الگوریتم ۱-تقریبی[1] یک جواب بهینه تولید می‌کند، و یک الگوریتم تقریبی با یک نسبت کارایی بزرگ ممکن است جواب‌هایی بازگرداند که با جواب بهینه بسیار فاصله داشته باشند.

برای بسیاری از مسائل، الگوریتم‌های تقریبی با نسبت‌های کارایی ثابت و کوچک توسعه داده شده‌اند، در حالی که برای مسائل دیگر، بهترین الگوریتم‌های تقریبی با زمان چندجمله‌ای، نسبت‌های کارایی دارند که به صورت توابعی با n رشد می‌کنند. یک مثال از چنین مسئله‌ای، مسئله‌ی پوشش مجموعه ارائه شده در بخش ۳۵-۳ است.

بعضی از مسائل NP-کامل، الگوریتم‌های تقریبی دارند که می‌توان با استفاده از زمان محاسبه‌ی بالاتر نسبت کارایی آن‌ها را کوچک‌تر و کوچک‌تر کرد. یعنی، یک سبک-سنگین میان زمان محاسبه و کیفیت تقریب وجود دارد. یک مثال، مسئله‌ی جمع زیرمجموعه است که در بخش ۳۵-۵ بررسی شده است. این حالت به اندازه‌ی کافی مهم است که برای خود نامی جداگانه داشته باشد.

یک *رویکرد تقریب* (approximation scheme) برای یک مسئله‌ی بهینه‌سازی، یک الگوریتم تقریبی است که به عنوان ورودی، علاوه بر نمونه‌ی مسئله، یک مقدار $\varepsilon > 0$ را هم دریافت می‌کند به طوری که برای هر ε ثابت، رویکرد تقریب یک الگوریتم $(1+\varepsilon)$-تقریبی است. به یک رویکرد تقریب، یک *رویکرد تقریب چندجمله‌ای* می‌گوییم اگر برای هر $\varepsilon > 0$ ثابت، رویکرد در زمان چندجمله‌ای نسبت به اندازه‌ی نمونه‌ی ورودی (n) اجرا شود.

زمان اجرای یک رویکرد تقریب چندجمله‌ای ممکن است با کاهش ε به شدت رشد کند. به عنوان مثال زمان اجرای یک رویکرد تقریب چندجمله‌ای ممکن است $O(n^{2/\varepsilon})$ باشد. به صورت ایده‌آل اگر ε با یک فاکتور ثابت رشد کند، زمان اجرای رسیدن به تقریب مورد نظر نباید بیش از یک فاکتور ثابت رشد کند (ولی نه لزوماً همان فاکتور ثابت که سرعت کاهش ε را تعیین می‌کند).

می‌گوییم یک رویکرد تقریب، یک *رویکرد تقریب کاملاً چندجمله‌ای* (fully polynomial-time

[1] وقتی نسبت کارایی مستقل از n باشد، از اصطلاح‌های «نسبت کارایی ρ» و «الگوریتم ρ-تقریبی» استفاده می‌کنیم، که نشان دهنده‌ی عدم وابستگی به n است.

(approximation algorithm) است اگر یک رویکرد تقریب باشد و زمان اجرای آن هم نسبت به $1/\varepsilon$ و هم نسبت به اندازه‌ی ورودی از مرتبه‌ی چندجمله‌ای باشد. به عنوان مثال، ممکن است زمان اجرای یک رویکرد $O((1/\varepsilon)^2 n^3)$ باشد. با چنین رویکردی، هر افزایشی با فاکتور ثابت در ε را می‌توان با یک فاکتور ثابت افزایش در زمان اجرا به دست آورد.

خلاصه‌ی فصل

چهار بخش اول این فصل مثال‌هایی از الگوریتم‌های تقریبی چندجمله‌ای برای مسائل NP-کامل ارائه می‌کنند، و در بخش پنجم یک الگوریتم تقریبی کاملاً چندجمله‌ای معرفی می‌شود. بخش ۳۵-۱ با بررسی مسئله‌ی پوشش رأسی آغاز می‌شود، یک مسئله‌ی کمینه‌سازی NP-کامل که یک الگوریتم تقریبی با یک نسبت کارایی ۲ دارد. بخش ۳۵-۲ یک الگوریتم تقریبی با نسبت کارایی ۲ برای مسئله‌ی فروشنده‌ی دوره‌گرد در حالتی ارائه می‌کند که تابع هزینه نامساوی مثلث را ارضا می‌کند. همچنین در این بخش می‌بینیم که بدون نامساوی مثلث، برای هر ثابت $\rho \geq 1$، یک الگوریتم ρ-تقریبی نمی‌تواند وجود داشته باشد، مگر این که P=NP. در بخش ۳۵-۳ نشان می‌دهیم که چگونه می‌توان از یک متد حریصانه به عنوان یک الگوریتم تقریبی کارآمد برای مسئله‌ی پوشش مجموعه استفاده کرد، و پوششی به دست آورد که در بدترین حالت با یک فاکتور لوگاریتمی از هزینه‌ی بهینه بزرگ‌تر است. بخش ۳۵-۴ دو الگوریتم تقریبی دیگر را ارائه می‌کند. اول نسخه‌ی بهینه‌سازی مسئله‌ی قابلیت ارضای 3-CNF را بررسی می‌کنیم، و یک الگوریتم تصادفی ساده برای آن ارائه می‌کنیم که یک جواب با امیدریاضی نسبت کارایی $8/7$ به دست می‌دهد. سپس یک نسخه‌ی وزن‌دار از مسئله‌ی پوشش رأسی را بررسی می‌کنیم و نشان می‌دهیم که چگونه می‌توان با استفاده از برنامه‌ریزی خطی یک الگوریتم ۲-تقریبی برای آن توسعه داد. نهایتاً، بخش ۳۵-۵ یک رویکرد تقریبی کاملاً چندجمله‌ای برای مسئله‌ی جمع زیرمجموعه ارائه می‌کند.

۳۵-۱ مسئله‌ی پوشش رأسی

مسئله‌ی پوشش رأسی در بخش ۳۴-۵-۲ به طور کامل تعریف، و NP-کامل بودن آن اثبات شد. به خاطر بیاورید که یک *پوشش رأسی* (vertex cover) یک گراف بدون جهت $G = (V, E)$، یک زیرمجموعه‌ی $V' \subseteq V$ است به طوری که اگر (u,v) یک یال در G باشد، آن گاه یا $u \in V'$ و یا $v \in V'$ (یا هر دو). اندازه‌ی یک پوشش رأسی برابر است با تعداد رأس‌های درون آن.

مسئله‌ی پوشش رأسی (vertex cover problem) عبارت است از یافتن یک پوشش رأسی با اندازه‌ی کمینه در یک گراف بدون جهت. به چنین پوشش رأسی، یک *پوشش رأسی بهینه* (optimal vertex cover) می‌گوییم. این مسئله، نسخه‌ی بهینه‌سازی یک مسئله‌ی تصمیم‌گیری NP-کامل است.

با این که نمی‌دانیم چگونه در زمان چندجمله‌ای یک پوشش رأسی بهینه در یک گراف G بیابیم، ولی می‌توانیم به صورت بهینه یک پوشش رأسی نزدیک بهینه بیابیم. یافتن یک پوشش رأسی که

نزدیک بهینه است کار چندان دشواری نیست. الگوریتم تقریبی زیر یک گراف بدون جهت G را به عنوان ورودی دریافت کرده و یک پوشش رأسی بازمی‌گرداند که اندازه‌ی آن بزرگ‌تر از دو برابر یک پوشش رأسی بهینه نخواهد بود.

```
APPROX-VERTEX-COVER(G)
1  C = Ø
2  E' = G.E
3  while E' ≠ Ø
4      let (u, v) be an arbitrary edge of E'
5      C = C ∪ {u, v}
6      remove from E' every edge incident on either u or v
7  return C
```

شکل ۳۵-۱ عملیات APPROX-VERTEX-COVER را نشان می‌دهد. متغیر C حاوی پوشش رأسی در حال ساخت است. خط ۱، C را با یک مجموعه‌ی تهی مقداردهی اولیه می‌کند. در خط ۲، یک مجموعه‌ی E' می‌سازد که یک کپی از مجموعه‌ی یال‌های $G.E$ است. حلقه‌ی خطوط ۳-۶ مکرراً یک یال (u,v) را از E' انتخاب کرده، نقاط پایانی آن را به C اضافه، و تمام یال‌هایی را که توسط u یا v پوشش داده شده‌اند، از E' حذف می‌کند. نهایتاً خط ۷ پوشش رأسی C را بازمی‌گرداند. زمان اجرای این الگوریتم با استفاده از نمایش لیست پیوندی برای E' برابر است با $O(V+E)$.

قضیه‌ی ۳۵-۱ APPROX-VERTEX-COVER یک الگوریتم ۲-تقریبی چندجمله‌ای است.

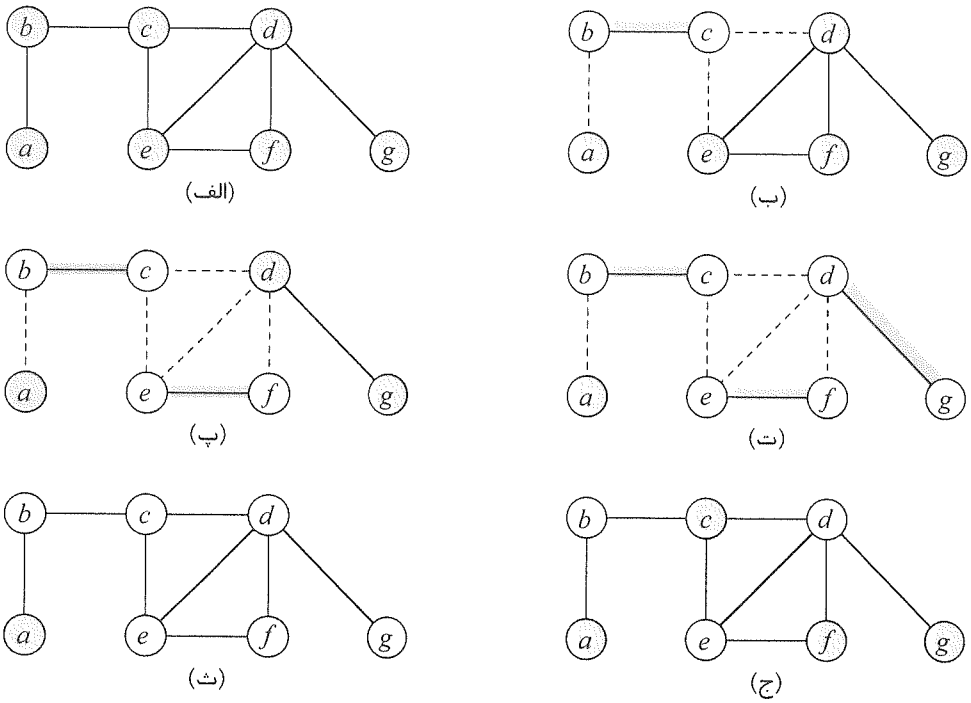
اثبات قبلاً نشان داده‌ایم که APPROX-VERTEX-COVER در زمان چندجمله‌ای اجرا می‌شود.

مجموعه‌ی C از رأس‌ها که توسط APPROX-VERTEX-COVER بازگردانده می‌شود، یک پوشش رأسی است، چرا که الگوریتم آن قدر حلقه را تکرار می‌کند تا تمام یال‌های $G.E$ توسط یک رأس در C پوشش داده شوند.

برای این که ببینیم اندازه‌ی پوشش رأسی بازگردانده شده توسط APPROX-VERTEX-COVER حداکثر دو برابر اندازه‌ی پوشش رأسی بهینه است، فرض کنید A نشان‌دهنده‌ی مجموعه‌ی یال‌هایی باشد که در خط ۴ رویه‌ی APPROX-VERTEX-COVER انتخاب می‌شوند. برای پوشش یال‌های A، هر پوشش رأسی – به طور خاص، پوشش رأسی بهینه‌ی C^* - باید حداقل شامل یک نقطه‌ی پایانی هر یال در A باشد. هیچ دو یالی در A دارای یک نقطه‌ی پایانی مشترک نیستند، چرا که وقتی یک یال در خط ۴ انتخاب می‌شود، تمام یال‌های مجاور با نقاط پایانی آن در خط ۶ از E' حذف می‌شوند. بنابراین، هیچ دو یالی در A توسط یک رأس در C^* پوشش داده نمی‌شوند، و کران پایین

$$|C^*| \geq |A| \qquad (۳۵-۲)$$

را بر روی اندازه‌ی پوشش رأسی بهینه داریم. هر تکرار از خط ۴ یک یال انتخاب می‌کند که هیچ یک از نقاط پایانی آن در C قرار ندارند، که به کران بالای زیر (در واقع، یک کران بالای دقیق) بر روی پوشش رأسی بازگردانده شده منجر می‌شود:



شکل ۳۵-۱ عمل کرد APPROX-VERTEX-COVER. (الف) گراف ورودی G، که ۷ رأس و ۸ یال دارد. (ب) یال (b,c)، که با سایه‌ی پررنگ نشان داده شده است، اولین یالی است که توسط APPROX-VERTEX-COVER انتخاب می‌شود. رأس‌های b و c، که با سایه‌ی کم‌رنگ نشان داده شده‌اند، به مجموعه‌ی C اضافه می‌شوند، که حاوی پوشش رأسی در حال ساخت است. یال‌های (a,b)، (c,e)، و (c,d) که هاشور خورده‌اند، حذف می‌شوند، چرا که اکنون توسط رأسی در C پوشش داده می‌شوند. (پ) یال (e,f) انتخاب می‌شود؛ رأس‌های e و f به C اضافه می‌شوند. (ت) یال (d,g) انتخاب می‌شود؛ رأس‌های d و g به C اضافه می‌شوند. (ث) مجموعه‌ی C، که پوشش رأسی ساخته شده توسط APPROX-VERTEX-COVER است، حاوی شش رأس b، c، d، e، f، و g است. (ج) پوشش رأسی بهینه برای این مسئله فقط شامل سه رأس است: b، d، و e.

$$|C| = ۲|A| \tag{۳۵-۳}$$

با ترکیب تساوی‌های (۳۵-۲) و (۳۵-۳) به دست می‌آوریم

$$\begin{aligned} |C| &= ۲|A| \\ &\le ۲|C^*| \end{aligned}$$

که قضیه را اثبات می‌کند. ■

اجازه دهید این اثبات را بررسی کنیم. ابتدا ممکن است از خود بپرسید که چطور می‌توانیم اثبات

کنیم که اندازه‌ی پوشش رأسی بازگردانده شده توسط APPROX-VERTEX-COVER حداکثر دو برابر اندازه‌ی یک پوشش رأسی بهینه است، در حالی که اندازه‌ی یک پوشش رأسی بهینه را نمی‌دانیم. جواب این است که از یک کران پایین بر روی اندازه‌ی پوشش رأسی بهینه استفاده می‌کنیم. همان طور که تمرین ۳۵-۱-۲ از شما می‌خواهد نشان دهید، مجموعه‌ی A از یال‌هایی که در خط ۴ رویه‌ی APPROX-VERTEX-COVER انتخاب شده‌اند، در واقع یک تطابق ماکسیمال بر روی گراف G است. (یک *تطابق ماکسیمال* (maximal matching)، یک تطابق است که زیرمجموعه‌ی اکید هیچ تطابق دیگری نیست.) اندازه‌ی یک تطابق ماکسیمال، همان طور که در اثبات قضیه‌ی ۳۵-۱ بحث کردیم، یک کران پایین بر روی اندازه‌ی یک پوشش رأسی بهینه است. الگوریتم یک پوشش رأسی بازمی‌گرداند که اندازه‌ی آن حداکثر دو برابر اندازه‌ی تطابق ماکسیمال A است. با مقایسه‌ی اندازه‌ی جواب بازگردانده شده با کران پایین، به نسبت کارایی مورد نظر می‌رسیم. از این متدولوژی در بخش‌های دیگر هم استفاده خواهیم کرد.

تمرین‌ها

۳۵-۱-۱ یک مثال از یک گراف بدهید که در آن APPROX-VERTEX-COVER همیشه به یک جواب غیربهینه ختم می‌شود.

۳۵-۱-۲ اثبات کنید مجموعه‌ی یال‌هایی که در خط ۴ رویه‌ی APPROX-VERTEX-COVER انتخاب شده‌اند یک تطابق ماکسیمال در گراف G را تشکیل می‌دهند.

۳۵-۱-۳★ پروفسور Nixon مکاشفه‌ی زیر را برای حل مسئله‌ی پوشش رأسی ارائه می‌کند. به صورت تکراری یک رأس با بالاترین درجه را انتخاب کرده، تمام یال‌های مجاور آن را حذف می‌کنیم. یک مثال ارائه کنید که نشان می‌دهد که مکاشفه‌ی پروفسور دارای نسبت کارایی ۲ نیست. (*راهنمایی:* یک گراف دوبخشی را امتحان کنید با رأس‌هایی با درجه‌ی یکنواخت در سمت چپ و رأس‌هایی با درجه‌های مختلف در سمت راست.)

۳۵-۱-۴ یک الگوریتم حریصانه‌ی کارآمد ارائه کنید که پوشش رأسی بهینه‌ی یک درخت را در زمان خطی می‌یابد.

۳۵-۱-۵ از قضیه‌ی ۳۴-۱۲ می‌دانیم که مسئله‌ی پوشش رأسی و مسئله‌ی گروهک مکمل یکدیگر هستند، بدین معنی که یک پوشش رأسی بهینه مکمل یک گروهک با اندازه‌ی بیشینه در گراف مکمل است. آیا می‌توان از این رابطه نتیجه گرفت که یک الگوریتم تقریبی چندجمله‌ای با یک نسبت کارایی ثابت برای مسئله‌ی گروهک هم وجود دارد؟ جواب خود را توجیه کنید.

۳۵-۲ مسئله‌ی فروشنده‌ی دوره‌گرد

در مسئله‌ی فروشنده‌ی دوره‌گرد که در بخش ۳۴-۵-۴ معرفی شد، به ما یک گراف بدون جهت و کامل $G = (V, E)$ داده شده است، با هزینه‌ی نامنفی $c(u,v)$ متناظر با هر یال $(u,v) \in E$، و باید یک دور همیلتونی (یک تور) در G با کم‌ترین هزینه‌ی بیابیم. به عنوان توسعه‌ی تعاریف مسئله، فرض کنید $c(A)$ نشان‌دهنده‌ی کل هزینه‌ی یال‌های زیرمجموعه‌ی $A \subseteq E$ باشد:

$$c(A) = \sum_{(u,v)\in A} c(u,v)$$

در بسیاری از موقعیت‌های عملی، کم‌هزینه‌ترین مسیر از یک مکان u به یک مکان w، مسیر مستقیم است، بدون مراحل میانی. به عبارت دیگر، حذف یک توقف‌گاه میانی نمی‌تواند هزینه را افزایش دهد. به صورت رسمی می‌توانیم بگوییم که تابع هزینه‌ی c *نامساوی مثلث* (triangle inequality) را ارضا می‌کند اگر برا تمام رأس‌های $u, v, w \in V$،

$$c(u,w) \le c(u,v) + c(v,w)$$

نامساوی مثلث یک خصوصیت طبیعی است، و در بسیاری از کاربردها خود به خود وجود دارد. مثلاً اگر رأس‌های یک گراف نقاطی در یک صفحه باشند، و هزینه‌ی جابه‌جایی بین دو رأس، فاصله‌ی اقلیدسی معمولی بین آن دو باشد، آن گاه نامساوی مثلث در این گراف ارضا می‌شود. همچنین توابع هزینه‌ی بسیاری غیر از فاصله‌ی اقلیدسی وجود دارند که نامساوی مثلث را ارضا می‌کنند.

همان طور که تمرین ۳۵-۲-۲ نشان می‌دهد، مسئله‌ی فروشنده‌ی دوره‌گرد NP-کامل است، حتی اگر تابع هزینه نامساوی مثلث را ارضا کند. بنابراین احتمالاً نمی‌توانیم یک الگوریتم چندجمله‌ای برای حل دقیق این مسئله بیابیم. بنابراین به دنبال الگوریتم‌های تقریبی مناسب می‌گردیم.

در بخش ۳۵-۲-۱، یک الگوریتم ۲-تقریبی را برای مسئله‌ی فروشنده‌ی دوره‌گرد در حالت ارضای نامساوی مثلث بررسی می‌کنیم. در بخش ۳۵-۲-۲، نشان می‌دهیم که بدون نامساوی مثلث، یک الگوریتم تقریبی چندجمله‌ای با نسبت کارایی ثابت وجود ندارد، مگر این که P=NP.

۳۵-۲-۱ مسئله‌ی فروشنده‌ی دوره‌گرد با نامساوی مثلث

با به کاربردن متدولوژی بخش قبل، ابتدا یک ساختار – یک درخت پوشای کمینه – می‌سازیم که وزن آن یک کران پایین بر روی طول هزینه‌ی یک تور بهینه‌ی فروشنده‌ی دوره‌گرد است. سپس از این درخت پوشای کمینه استفاده می‌کنیم تا یک تور بسازیم که هزینه‌ی آن بیش از دو برابر وزن درخت پوشای کمینه نباشد، البته اگر تابع هزینه نامساوی مثلث را ارضا کند. الگوریتم زیر با فراخوانی الگوریتم درخت پوشای کمینه‌ی MST-PRIM از بخش ۲۳-۲ این رویکرد را پیاده‌سازی می‌کند. پارامتر G یک گراف بدون جهت کامل است، و تابع هزینه‌ی c نامساوی مثلث را ارضا می‌کند.

```
APPROX-TSP-TOUR(G, c)
1  select a vertex r ∈ G.V to be a "root" vertex
2  compute a minimum spanning tree T for G from root r
```

```
        using MST-PRIM(G, c, r)
3       let H be the list of vertices, ordered according to when they are first visited
            in a preorder tree walk of T
4       return the hamiltonian cycle H
```

از بخش ۱۲-۱ به خاطر بیاورید که یک پیمایش پیش‌ترتیبی بر روی درخت به صورت بازگشتی تمام رأس‌های یک درخت را ملاقات می‌کند، و در اولین ملاقات هر رأس، قبل از تمام فرزندان آن رأس، آن را در لیست وارد می‌کند.

شکل ۳۵-۲ عملیات APPROX-TSP-TOUR را نشان می‌دهد. بخش (الف) شکل مجموعه‌ی رأس‌های داده شده را نشان می‌دهد، و بخش (ب) درخت پوشای کمینه‌ی T را که از رأس ریشه‌ی a توسط MST-PRIM ساخته شده است. بخش (پ) رأس‌هایی را نشان می‌دهد که توسط پیمایش پیش‌ترتیبی T ملاقات شده‌اند، و بخش (ت) تور متناظر را نمایش می‌دهد، که تور بازگردانده شده توسط APPROX-TSP-TOUR است. بخش (ث) یک تور بهینه را نشان می‌دهد، که حدودا ۲۳٪ کوتاه‌تر است.

طبق تمرین ۲۳-۲-۲، حتی با یک پیاده‌سازی ساده از MST-PRIM، زمان اجرای APPROX-TSP-TOUR برابر است با $\theta(V^2)$. اکنون نشان می‌دهیم که اگر تابع هزینه برای یک نمونه از مسئله‌ی فروشنده‌ی دوره‌گرد نامساوی مثلث را ارضا کند، آن گاه APPROX-TSP-TOUR توری بازمی‌گرداند که هزینه‌ی آن بیش از دو برابر هزینه‌ی یک تور بهینه نیست.

قضیه‌ی ۳۵-۲ APPROX-TSP-TOUR یک الگوریتم ۲-تقریبی چندجمله‌ای برای مسئله‌ی فروشنده‌ی دوره‌گرد در حالتی است که تابع هزینه نامساوی مثلث را ارضا کند.

اثبات قبلاً نشان دادیم که APPROX-TSP-TOUR در زمان چندجمله‌ای اجرا می‌شود.

فرض کنید H^* نشان‌دهنده‌ی یک تور بهینه برای یک مجموعه‌ی داده شده از رأس‌ها باشد. چون با حذف هر یک از یال‌های تور می‌توانیم یک درخت پوشا بسازیم و هزینه‌ی یال‌ها نامنفی است، وزن درخت پوشای کمینه‌ی T یک کران پایین برای هزینه‌ی یک تور بهینه است، یعنی

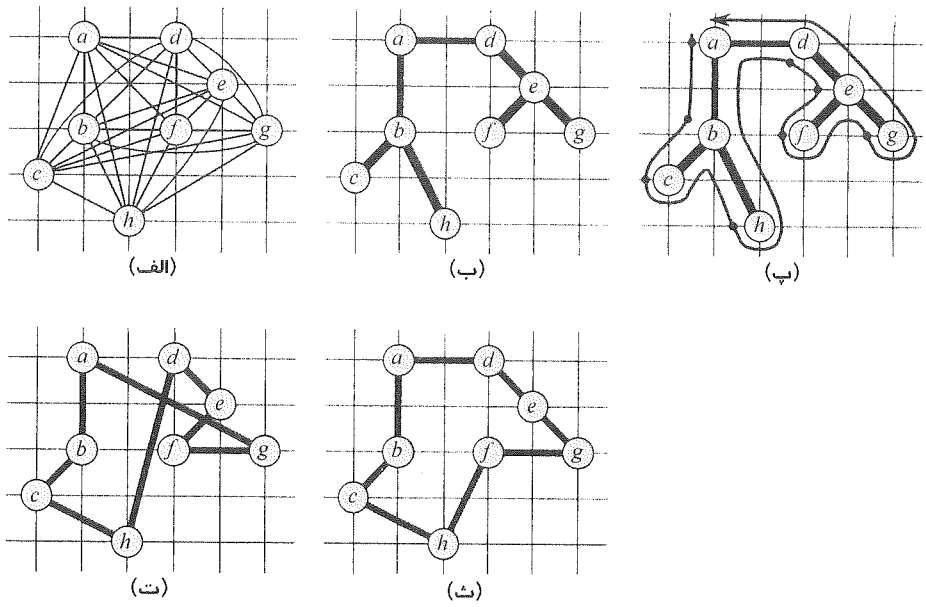
$$c(T) \le c(H^*) \tag{۳۵-۴}$$

یک *پیمایش کامل* (full walk) از T، رأس‌ها را در زمان اولین ملاقات، لیست می‌کند، و همچنین زمانی که بعد از ملاقات یک زیردرخت به آن‌ها بازمی‌گردیم. اجازه دهید این پیمایش را W بنامیم. پیمایش کامل مثال بالا، ترتیب زیر را به ما می‌دهد:

$$a,b,c,b,h,b,a,d,e,f,e,g,e,d,a$$

از آن جایی که پیمایش کامل، از هر یال T دقیقاً دو بار می‌گذرد، (با گسترش تعریف خود از هزینه‌ی c برای حالت مجموعه‌های چندتایی از یال‌ها) داریم

$$c(W) = 2c(T) \tag{۳۵-۵}$$



شکل ۳۵-۲ عملیات APPROX-TSP-TOUR. (الف) یک گراف بدون جهت کامل. رأس‌ها روی تقاطع خطوط شبکه (با مختصات صحیح) قرار دارند. مثلاً f با فاصله‌ی یک واحد در سمت راست و دو واحد در بالای h قرار دارد. از فاصله‌ی اقلیدسی به عنوان تابع هزینه میان دو نقطه استفاده شده است. (ب) یک درخت پوشای کمینه‌ی T از این نقاط که توسط MST-PRIM محاسبه شده است. رأس a ریشه است. فقط یال‌های درخت پوشای کمینه نشان داده شده است. رأس‌ها به صورت اتفاقی به همان ترتیبی برچسب‌گذاری شده‌اند که در MST-PRIM به درخت اضافه می‌شوند. (پ) یک پیمایش بر روی T، که از a آغاز می‌شود. یک پیمایش کامل بر روی درخت رأس‌ها را به ترتیب a، b، c، h، b، a، d، e، f، e، g، e، d، a ملاقات می‌کند. پیمایش پیش‌ترتیبی درخت رأس‌ها را زمانی لیست می‌کند که اولین بار ملاقات می‌شوند، که با نقطه‌ی کنار هر رأس مشخص شده است، که به ترتیب a، b، c، h، d، e، f، g منجر می‌شود. (ت) یک تور از رأس‌ها که با ملاقات رأس‌ها به ترتیب داده شده توسط پیمایش پیش‌ترتیبی به دست می‌آید. این همان تور H است که توسط APPROX-TSP-TOUR بازگردانده می‌شود. هزینه‌ی کلی آن تقریباً برابر است با ۱۹٫۰۷۴. (ث) یک تور بهینه‌ی H^* برای مجموعه‌ی رأس‌های داده شده. هزینه‌ی کلی آن تقریباً برابر است با ۱۴٫۷۱۵.

نامساوی (۳۵-۴) و تساوی (۳۵-۵) نتیجه می‌دهند

$$c(W) \le 2c(H^*) \qquad (۳۵-۶)$$

و بنابراین هزینه‌ی W درون یک فاکتور ۲ از هزینه‌ی یک تور بهینه قرار دارد.

متأسفانه W به طور کلی یک تور نیست، چرا که بعضی از رأس‌ها را بیش از یک بار ملاقات می‌کند. با این حال، طبق نامساوی مثلث می‌توانیم ملاقات یک رأس در W را حذف کنیم بدون این که

هزینه افزایش یابد. (اگر در W یک رأس v در میان ملاقات‌های u و w حذف شود، ترتیب حاصل به معنی حرکت مستقیم از u به w است.) با به کار بردن مکرر این عملیات، می‌توانیم تمام ملاقات‌های هر رأس را، غیر از اولین ملاقات آن رأس، حذف کنیم. با این کار، در مثال بالا ترتیب زیر حاصل می‌شود:

$$a,b,c,g,d,e,f,g$$

این ترتیب، همان ترتیبی است که توسط پیمایش پیش‌ترتیبی درخت T حاصل می‌شود. فرض کنید H دور متناظر با این پیمایش پیش‌ترتیبی باشد. این یک دور همیلتونی است، چرا که در آن هر رأس دقیقاً یک بار ملاقات می‌شود، و در واقع این همان دور محاسبه شده توسط APPROX-TSP-TOUR است. چون H با حذف رأس‌ها از پیمایش کامل W حاصل شده است، داریم

$$c(H) \le c(W) \qquad (۳۵-۷)$$

ترکیب نامساوی‌های (۳۵-۶) و (۳۵-۷) نشان می‌دهد که $c(H) \le 2c(H^*)$، که اثبات را کامل می‌کند. ■

بر خلاف این نسبت کالایی خوب فراهم شده توسط قضیه‌ی ۳۵-۲، APPROX-TSP-TOUR معمولاً در عمل بهترین انتخاب نیست. الگوریتم‌های تقریبی دیگری وجود دارند که معمولاً در عمل بسیار بهتر عمل می‌کنند.

۳۵-۲-۲ مسئله‌ی فروشنده‌ی دوره‌گرد در حالت کلی

اگر از این فرض صرف نظر کنیم که تابع هزینه‌ی c نامساوی مثلث را ارضا می‌کند، نمی‌توان در زمان چندجمله‌ای تورهای تقریبی خوب یافت، مگر این که P=NP.

قضیه‌ی ۳۵-۳ اگر $P \neq NP$، آن گاه برای هیچ ثابت $\rho \ge 1$ یک الگوریتم تقریبی چندجمله‌ای با نسبت کارایی ρ برای مسئله‌ی فروشنده‌ی دوره‌گرد در حالت کلی وجود ندارد.

اثبات اثبات به کمک برهان خلف انجام می‌شود. فرض کنید که برای یک ثابت $\rho \ge 1$، یک الگوریتم تقریبی چندجمله‌ای A با نسبت کارایی ρ وجود داشته باشد. بدون از دست دادن کلیت، فرض می‌کنیم که ρ یک عدد صحیح است، که در صورت لزوم می‌توان با گرد کردن به سمت بالا این فرض را ارضا کرد. سپس نشان می‌دهیم که چگونه با استفاده از A می‌توان نمونه‌های مسئله‌ی دور همیلتونی (تعریف شده در بخش ۳۴-۲) را در زمان چندجمله‌ای حل کرد. چون طبق قضیه‌ی ۳۴-۱۳ مسئله‌ی دور همیلتونی NP-کامل است، قضیه‌ی ۳۴-۴ ایجاب می‌کند که حل آن در زمان چندجمله‌ای مستلزم این است که داشته باشیم P=NP.

فرض کنید $G = (V, E)$ یک نمونه از مسئله‌ی دور همیلتونی باشد. می‌خواهیم به صورت کارآمد و با استفاده از الگوریتم فرضی A تعیین کنیم که آیا G حاوی یک دور همیلتونی است یا نه. به صورت زیر G را به یک نمونه از مسئله‌ی فروشنده‌ی دوره‌گرد تبدیل می‌کنیم. فرض کنید

$G'=(V,E')$ گراف کامل روی V باشد؛ یعنی

$$E'=\{(u,v):u,v\in V \text{ و } u\neq v\}$$

به صورت زیر یک هزینه به هر یال در E' نسبت می‌دهیم:

$$c(u,v)=\begin{cases}1 & \text{اگر } (u,v)\in E\\ \rho|V|+1 & \text{در غیر این صورت}\end{cases}$$

نمایش‌های G' و c را می‌توان در زمان چندجمله‌ای نسبت به $|V|$ و $|E|$ از روی نمایش G ساخت.

اکنون مسئله‌ی فروشنده‌ی دوره‌گرد (G',c) را در نظر بگیرید. اگر گراف اصلی G یک دور همیلتونی H داشته باشد، تابع هزینه‌ی c به هر یال H هزینه‌ی ۱ نسبت می‌دهد، و بنابراین (G',c) حاوی یک تور با هزینه‌ی $|V|$ است. از طرف دیگر اگر G حاوی یک دور همیلتونی نباشد، آن گاه هر توری در G' باید از یالی استفاده کند که در E نیست. ولی هر توری که از یالی استفاده می‌کند که در E نیست، هزینه‌ای دارد حداقل برابر با

$$\begin{aligned}(\rho|V|+1)+(|V|-1)&=\rho|V|+|V|\\ &>\rho|V|\end{aligned}$$

از آن جایی که یال‌هایی که در G نیستند، پرهزینه‌اند، فاصله‌ای حداقل به اندازه‌ی $|V|$ بین هزینه‌ی یک تور که یک دور همیلتونی در G است (هزینه‌ی $|V|$) و هزینه‌ی هر تور دیگر (هزینه‌ی حداقل $\rho|V|+|V|$) وجود دارد. بنابراین هزینه‌ی توری که یک دور همیلتونی در G نیست حداقل به اندازه‌ی یک فاکتور $\rho+1$ بزرگ‌تر از هزینه‌ی یک دور همیلتونی در G است.

اکنون فرض کنید که الگوریتم تقریبی A را به مسئله‌ی فروشنده‌ی دوره‌گرد (G',c) اعمال کنیم. چون تضمین شده است که A توری بازمی‌گرداند که هزینه‌ی آن حداکثر چند برابر هزینه‌ی یک تور بهینه است، اگر G حاوی یک دور همیلتونی باشد، آن گاه A باید آن را بازگرداند. اگر G دور همیلتونی نداشته باشد، آن گاه A یک تور با هزینه‌ای کم‌تر یا مساوی $\rho|V|$ بازمی‌گرداند. بنابراین، می‌توانیم از A برای حل مسئله‌ی دور همیلتونی در زمان چندجمله‌ای استفاده کنیم. ■

اثبات قضیه‌ی ۳۵-۳ مثالی است از تکنیک کلی برای اثبات این که نمی‌توان تقریب خوبی برای یک مسئله یافت. فرض کنید که با داشتن یک مسئله‌ی NP-سخت داده شده‌ی X، می‌توانیم یک مسئله‌ی کمینه‌سازی Y تولید کنیم به طوری که نمونه‌های «بله»ی مسئله‌ی X متناظر هستند با نمونه‌هایی از Y با هزینه‌ی حداکثر k (برای یک k)، ولی نمونه‌های «نه» از مسئله‌ی X متناظرند با نمونه‌هایی از Y که هزینه‌ی آن‌ها بیش از ρk است. آن گاه نشان داده‌ایم که هیچ الگوریتم ρ-تقریبی برای مسئله‌ی Y وجود ندارد، مگر این که P=NP.

تمرین‌ها

۳۵-۲-۱ فرض کنید که یک گراف بدون جهت $G = (V, E)$ با حداقل ۳ رأس، یک تابع هزینه‌ی c دارد که نامساوی مثلث را ارضا می‌کند. اثبات کنید که $c(u,v) \geq 0$ برای تمام $u, v \in V$.

۳۵-۲-۲ نشان دهید که چگونه می‌توانیم در زمان چندجمله‌ای یک نمونه از مسئله‌ی فروشنده‌ی دوره‌گرد را به نمونه‌ای دیگر تبدیل کنیم به طوری که تابع هزینه‌ی آن نامساوی مثلث را ارضا کند. دو نمونه باید مجموعه‌ی یکسانی از تورهای بهینه داشته باشند. توضیح دهید که چرا این تبدیل چندجمله‌ای با قضیه‌ی ۳۵-۳ تناقض ندارد، با این فرض که $P \neq NP$.

۳۵-۲-۳ *مکاشفه‌ی نزدیک‌ترین نقطه* (closest-point heuristic) برای ساختن یک تور تقریبی برای مسئله‌ی فروشنده‌ی دوره‌گرد را در نظر بگیرید. با یک دور بدیهی شامل یک رأس به دلخواه انتخاب شده آغاز می‌کنیم. در هر مرحله، رأس u را طوری میابیم که در دور نباشد ولی فاصله‌ی آن از هر رأس بر روی دور کمینه باشد. فرض کنید که رأسی در دور که کم‌ترین فاصله را با u دارد، رأس v باشد. با اضافه کردن u دقیقاً بعد از v، دور را طوری گسترش می‌دهیم که شامل u هم بشود. این روش را طوری ادامه می‌دهیم تا تمام رأس‌ها در دور باشند. اثبات کنید که این مکاشفه توری بازمی‌گرداند که هزینه‌ی کلی آن بیش از دو برابر هزینه‌ی یک تور بهینه نیست.

۳۵-۲-۴ *مسئله‌ی فروشنده‌ی دوره‌گرد گلوگاهی* (bottleneck traveling-salesman problem) مسئله‌ی یافتن یک دور همیلتونی است به طوری که هزینه‌ی سنگین‌ترین یال کمینه شود. با فرض این که تابع هزینه نامساوی مثلث را ارضا می‌کند، نشان دهید که یک الگوریتم تقریبی چندجمله‌ای با نسبت کارایی ۳ برای این مسئله وجود دارد. (*راهنمایی:* به صورت بازگشتی نشان دهید که می‌توانیم با انجام یک پیمایش کامل بر روی درخت و پرش از روی گره‌ها (بدن پرش از روی بیش از دو گره‌ی میانی متوالی) تمام گره‌های یک درخت پوشای گلوگاه را، همان طور که در مسئله‌ی ۲۳-۳ بحث شد، دقیقاً یک بار ملاقات کنیم. نشان دهید که هزینه‌ی سنگین‌ترین یال در درخت پوشای گلوگاه حداکثر برابر است با هزینه‌ی سنگین‌ترین یال در دور همیلتونی گلوگاه.)

۳۵-۲-۵ فرض کنید که رأس‌های گراف در یک نمونه از مسئله‌ی فروشنده‌ی دوره‌گرد، نقاطی در صفحه هستند و هزینه‌ی $c(u,v)$ همان فاصله‌ی اقلیدسی بین نقاط u و v است. نشان دهید که یک تور بهینه هیچ گاه خود را قطع نمی‌کند.

۳-۳۵ مسئله‌ی پوشش مجموعه

مسئله‌ی بهینه‌سازی پوشش مجموعه، مدلی است برای بسیاری از مسائلی که در آن‌ها تخصیص منبع وجود دارد. مسئله‌ی تصمیم‌گیری متناظر با آن، حالت کلی‌تر مسئله‌ی NP-کامل پوشش رأسی است، و بنابراین NP-سخت است. با این حال الگوریتم تقریبی توسعه داده شده برای مسئله‌ی پوشش رأسی در این جا کاربرد ندارد، و بنابراین باید راه‌های دیگر را امتحان کنیم. در این جا یک مکاشفه‌ی حریصانه‌ی ساده را بررسی می‌کنیم که نسبت کارایی آن لوگاریتمی است. یعنی با بزرگ‌تر شدن اندازه‌ی نمونه‌های مسئله، اندازه‌ی جواب تقریبی نسبت به یک جواب بهینه هم ممکن است افزایش یابد. با این حال چون تابع لوگاریتمی به کندی رشد می‌کند، این الگوریتم تقریبی می‌تواند نتایج خوبی به دست بدهد.

یک نمونه‌ی $(X, \mathcal{F})$ از *مسئله‌ی پوشش مجموعه* (set-covering problem) تشکیل شده است از یک مجموعه‌ی متناهی X و یک خانواده‌ی $\mathcal{F}$ از زیرمجموعه‌های X، به طوری که هر عنصر از X حداقل به یک زیرمجموعه در $\mathcal{F}$ تعلق دارد:

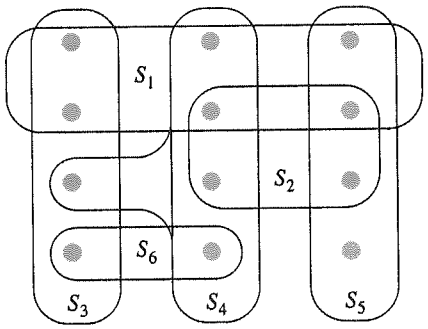
$$X = \bigcup_{S \in \mathcal{F}} S$$

می‌گوییم یک زیرمجموعه‌ی $S \in \mathcal{F}$ عناصر خود را *پوشش* (cover) می‌دهد. مسئله این است که می‌خواهیم یک زیرمجموعه‌ی $\mathcal{C} \in \mathcal{F}$ با اندازه‌ی کمینه بیابیم به طوری که عناصر آن تمام X را پوشش می‌دهند:

$$X = \bigcup_{S \in \mathcal{C}} S \qquad (۸-۳۵)$$

هرگاه یک $\mathcal{C}$ تساوی (۸-۳۵) را ارضا کند، می‌گوییم X را *پوشش می‌دهد*. شکل ۳-۳۵ مسئله‌ی پوشش مجموعه را نشان می‌دهد. اندازه‌ی $\mathcal{C}$ به صورت تعداد مجموعه‌های آن تعریف می‌شود، نه به صورت تعداد عناصر جداگانه‌ی درون مجموعه‌های آن، چرا که هر زیرمجموعه‌ی $\mathcal{C}$ که X را پوشش می‌دهد باید شامل تمام $|X|$ عنصر باشد. در شکل ۳-۳۵، اندازه‌ی پوشش مجموعه‌ی کمینه ۳ است.

مسئله‌ی پوشش مجموعه، نسخه‌ی انتزاعی بسیاری از مسائل پرکاربرد ترکیبیات است. به عنوان یک مثال ساده، فرض کنید که X نشان‌دهنده‌ی یک مجموعه از مهارت‌ها است که برای حل یک مسئله نیاز است، و یک مجموعه از افراد داریم که برای کار بر روی پروژه آماده هستند. می‌خواهیم یک کمیته برای انجام کار تشکیل دهیم متشکل از کم‌ترین تعداد افراد ممکن، به طوری که برای هر مهارت مورد نیاز در X، یک عضو از کمیته باشد که آن مهارت را داشته باشد. در این نسخه‌ی تصمیم‌گیری از مسئله‌ی پوشش مجموعه، می‌خواهیم بدانیم که آیا یک پوشش با اندازه‌ی حداکثر k وجود دارد یا نه، که k یک پارامتر اضافی است که در نمونه‌ی مسئله تعیین می‌شود. نسخه‌ی تصمیم‌گیری مسئله NP-کامل است، همان طور که تمرین ۳-۳-۲ از شما می‌خواهد نشان دهید.



شکل ۳۵-۳ یک نمونه‌ی $(X, \mathcal{F})$ از مسئله‌ی پوشش مجموعه، که در آن X حاوی ۱۲ نقطه‌ی سیاه است، و $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$. یک پوشش کمینه، $\{S_3, S_4, S_5\}$ است با اندازه‌ی ۳. الگوریتم حریصانه، به ترتیب با انتخاب S_1، S_4، S_5، و S_3 و یا انتخاب S_1، S_4، S_5، و S_6 یک پوشش با اندازه‌ی ۴ می‌سازد.

یک الگوریتم تقریبی حریصانه

یک متد حریصانه به این صورت است که در هر مرحله، مجموعه‌ی S را انتخاب می‌کنیم که بیشترین تعداد از عناصر باقی مانده را پوشش می‌دهد.

```
GREEDY-SET-COVER(X, )
1  U = X
2  C = ∅
3  while U ≠ ∅
4       select an S ∈ F that maximizes |S ∩ U|
5       U = U - S
6       C = C ∪ {S}
7  return C
```

در مثال شکل ۳۵-۳، رویه‌ی GREEDY-SET-COVER به ترتیب مجموعه‌های S_1، S_4، S_5، و در نهایت S_3 یا S_6 را به C اضافه می‌کند.

الگوریتم به صورت زیر کار می‌کند. در هر مرحله مجموعه‌ی U حاوی مجموعه‌ی عناصر پوشش داده نشده است. مجموعه‌ی C حاوی پوشش در حال ساخته شدن است. خط ۴، مرحله‌ی انتخاب حریصانه است. یک زیرمجموعه‌ی S انتخاب می‌شود که بیشترین تعداد عناصر پوشش داده نشده‌ی ممکن را پوشش می‌دهد (و حالت‌های مساوی به صورت دلخواه شکسته می‌شوند). بعد از انتخاب S خط ۵ عناصر آن از U حذف می‌کند، و خط ۶ S را در C قرار می‌دهد. وقتی الگوریتم پایان می‌یابد، مجموعه‌ی C حاوی یک زیرخانواده از $\mathcal{F}$ است که X را پوشش می‌دهد.

می‌توان به سادگی الگوریتم GREEDY-SET-COVER را در زمان چندجمله‌ای نسبت به $|X|$ و $|\mathcal{F}|$ پیاده‌سازی کرد. چون تعداد تکرارهای حلقه‌ی خطوط ۳-۶ از بالا با $\min(|X|, |\mathcal{F}|)$ محدود شده است، و می‌توان بدنه‌ی حلقه را با زمان $O(|X|, |\mathcal{F}|)$ پیاده‌سازی کرد، یک پیاده‌سازی برای این الگوریتم

وجود دارد که در زمان $O(|X||\mathcal{F}|\min(|X|,|\mathcal{F}|))$ اجرا می‌شود. تمرین ۳۵-۳-۳ از شما می‌خواهد یک الگوریتم با زمان خطی ارائه کنید.

تحلیل

اکنون نشان می‌دهیم که الگوریتم حریصانه پوششی بازمی‌گرداند که بسیار بزرگ‌تر از پوشش بهینه است. برای سادگی، در این فصل dامین عدد هارمونیک $H_d = \sum_{i=1}^{d} 1/i$ (بخش الف–۱را ببینید) را با $H(d)$ نشان می‌دهیم. برای حالت مرزی، تعریف می‌کنیم $H(0) = 0$.

قضیه‌ی ۳۵-۴ GREEDY-SET-COVER یک الگوریتم $\rho(n)$–تقریبی چندجمله‌ای است، که در آن $\rho(n) = H(\max\{|S| : S \in \mathcal{F}\})$

اثبات قبلاً نشان دادیم که GREEDY-SET-COVER در زمان چندجمله‌ای اجرا می‌شود.

برای این که نشان دهیم GREEDY-SET-COVER یک الگوریتم $\rho(n)$–تقریبی است، یک هزینه‌ی ۱ برای هر مجموعه‌ی انتخاب شده توسط الگوریتم تعریف می‌کنیم، این هزینه را بین عناصر پوشش داده شده تقسیم می‌کنیم، و سپس از این هزینه‌ها برای به دست آوردن رابطه‌ی مورد نظر بین اندازه‌ی یک پوشش مجموعه‌ی بهینه‌ی $\mathcal{C}^*$ و اندازه‌ی پوشش مجموعه‌ی $\mathcal{C}$ که توسط الگوریتم بازگردانده می‌شود، استفاده می‌کنیم. فرض کنید S_i نشان دهنده‌ی iامین زیرمجموعه‌ی انتخاب شده توسط GREEDY-SET-COVER باشد؛ الگوریتم هنگام اضافه کردن S_i به $\mathcal{C}$ متحمل یک هزینه‌ی ۱ می‌شود. این هزینه را به صورت مساوی میان عناصری که اولین بار با انتخاب S_i پوشش داده شده‌اند، تقسیم می‌کنیم. فرض کنید c_x نشان دهنده‌ی هزینه‌ی متناظر با عنصر x باشد، برای $x \in X$. به هر عنصر فقط یک بار و در زمان اولین پوشش آن عنصر یک هزینه نسبت داده می‌شود. اگر x اولین بار توسط S_i پوشش داده شود، آن گاه

$$c_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}$$

در هر مرحله‌ی الگوریتم، یک واحد هزینه انتقال داده می‌شود، و بنابراین

$$|\mathcal{C}| = \sum_{x \in X} c_x \qquad (۳۵-۹)$$

هر عنصر $x \in X$ در پوشش بهینه‌ی $\mathcal{C}^*$ حداقل در یکی مجموعه حضور دارد، و بنابراین داریم

$$\sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x \qquad (۳۵-۱۰)$$

با ترکیب تساوی (۳۵-۹) و نامساوی (۳۵-۱۰) خواهیم داشت

$$|\mathcal{C}| \leq \sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x \qquad (۳۵-۱۱)$$

ادامه‌ی این اثبات بر پایه‌ی نامساوی مهم زیر است، که به زودی آن را اثبات خواهیم کرد. برای هر

مجموعه‌ی S متعلق به خانواده‌ی $\mathcal{F}$،

$$\sum_{x \in S} c_x \leq H(|S|) \qquad (۱۲-۳۵)$$

از نامساوی‌های (۳۵-۱۱) و (۳۵-۱۲) نتیجه می‌شود

$$\begin{aligned} |C| &\leq \sum_{S \in C^*} H(|S|) \\ &\leq |C^*| \cdot H(\max\{|S| : S \in \mathcal{F}\}) \end{aligned}$$

که قضیه را اثبات می‌کند.

تنها چیزی که باقی می‌ماند، اثبات نامساوی (۳۵-۱۲) است. یک مجموعه‌ی $S \in \mathcal{F}$ و $i = ۱,۲,...,|C|$ را در نظر بگیرید، و فرض کنید

$$u_i = |S - (S_۱ \cup S_۲ \cup ... \cup S_i)|$$

تعداد عناصر پوشش داده نشده بعد از انتخاب $S_۱, S_۲, ..., S_i$ توسط الگوریتم باشد. تعریف می‌کنیم $u_۰ = |S|$ تعداد عناصر S باشد، که در ابتدا هیچ کدام پوشش داده نشده‌اند. فرض کنید k کوچک‌ترین اندیسی باشد که $u_k = ۰$، به طوری که تمام عناصر S حداقل توسط یکی از مجموعه‌های $S_۱, S_۲, ..., S_k$ پوشش داده شده باشند، ولی عنصری در S وجود داشته باشد که توسط $S_۱ \cup S_۲ \cup ... \cup S_{k-۱}$ پوشش داده نشده است. آن گاه داریم $u_{i-۱} \geq u_i$، و برای $i = ۱,۲,...,k$ تعداد $u_{i-۱} - u_i$ عنصر از S اولین بار توسط S_i پوشش داده می‌شوند. بنابراین

$$\sum_{x \in S} c_x = \sum_{i=۱}^{k} (u_{i-۱} - u_i) \cdot \frac{۱}{|S_i - (S_۱ \cup S_۲ \cup ... \cup S_{i-۱})|}$$

مشاهده کنید که

$$\begin{aligned} |S_i - (S_۱ \cup S_۲ \cup ... \cup S_{i-۱})| &\geq |S - (S_۱ \cup S_۲ \cup ... \cup S_{i-۱})| \\ &= u_{i-۱} \end{aligned}$$

چون که انتخاب حریصانه‌ی S_i تضمین می‌کند که S نمی‌تواند عناصر جدید بیشتری از S_i را پوشش دهد (در غیر این صورت الگوریتم S را به جای S_i انتخاب می‌کرد). در نتیجه به دست می‌آوریم

$$\sum_{x \in S} c_x \leq \sum_{i=۱}^{k} (u_{i-۱} - u_i) \cdot \frac{۱}{u_i - ۱}$$

اکنون به صورت زیر کران این کمیت را تعیین می‌کنیم:

$$\begin{aligned} \sum_{x \in S} c_x &\leq \sum_{i=۱} (u_{i-۱} - u_i) \cdot \frac{۱}{u_{i-۱}} \\ &= \sum_{i=۱}^{k} \sum_{j=u_{i+۱}}^{u_{i-۱}} \frac{۱}{u_{i-۱}} \end{aligned}$$

$$\leq \sum_{i=1}^{k} \sum_{j=u_{i+1}+1}^{u_{i-1}} \frac{1}{j} \qquad (j \leq u_{i-1} \text{ چون})$$

$$= \sum_{i=1}^{k} \left(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right)$$

$$= \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i))$$

$$= H(u_0) - H(u_k) \quad \text{(چون مجموع، تلسکوپی است)}$$

$$= H(u_0) - H(0)$$

$$= H(u_0) \quad (H(0) = 0 \text{ چون})$$

$$= H(|S|)$$

که اثبات نامساوی (۳۵-۱۲) را کامل می‌کند.

نتیجه‌ی ۳۵-۵ GREEDY-SET-COVER یک الگوریتم $(\ln|X|+1)$-تقریبی چندجمله‌ای است.

اثبات از نامساوی (الف-۱۴) و قضیه‌ی ۳۵-۴ استفاده کنید.

در بعضی از کاربردها $\max\{|S| : S \in \mathcal{F}\}$ یک ثابت کوچک است، و بنابراین جواب بازگردانده شده توسط GREEDY-SET-COVER حداکثر به اندازه‌ی یک ضریب ثابت کوچک از جواب بهینه بزرگ‌تر است. یکی از این کاربردها وقتی اتفاق می‌افتد که از این مکاشفه برای به دست آوردن یک پوشش رأسی تقریبی برای یک گراف استفاده می‌شود که درجه‌ی رأس‌های آن حداکثر ۳ است. در این حالت جواب یافته شده توسط GREEDY-SET-COVER بزرگ‌تر از $H(3) = 11/6$ برابر جواب بهینه نیست؛ یک تضمین کارایی که مقداری بهتر از APPROX-VERTEX-COVER است.

تمرین‌ها

۳۵-۳-۱ هر یک از کلمه‌های زیر را به عنوان مجموعه‌ای از حروف در نظر بگیرید: {arid, dash, drain, heard, lost, nose shun, slate, snare, thread}. نشان دهید که اگر در حالت‌های مساوی کلمه‌هایی انتخاب شوند که در مجموعه زودتر ظاهر می‌شوند، رویه‌ی GREEDY-SET-COVER چه پوششی تولید می‌کند.

۳۵-۳-۲ با کاهش از مسئله‌ی پوشش رأسی، نشان دهید که نسخه‌ی تصمیم‌گیری مسئله‌ی پوشش مجموعه NP-کامل است.

۳-۳-۳۵ نشان دهید که چگونه می‌توان GREEDY-SET-COVER را طوری پیاده‌سازی کرد که در زمان $O(\sum_{S \in \mathcal{F}} |S|)$ اجرا شود.

۴-۳-۳۵ نشان دهید که نسخه‌ی ضعیف زیر از قضیه‌ی ۳۵-۴ به صورت بدیهی صحیح است:

$$|\mathcal{C}| \leq |\mathcal{C}^*| \max\{|S| : S \in \mathcal{F}\}$$

۵-۳-۳۵ GREEYD-SET-COVER می‌تواند جواب‌های مختلفی بازگرداند، بسته به این که چگونه در خط ۴ حالت‌های مساوی را می‌شکنیم. یک رویه‌ی BAD-SET-COVER-INSTANCE(n) ارائه کنید که یک نمونه‌ی n عنصری از مسئله‌ی پوشش مجموعه بازمی‌گرداند به طوری که بسته به نحوه‌ی شکستن حالت‌های مساوی در خط ۴، GREEDY-SET-COVER می‌تواند تعدادی جواب مختلف بازگرداند که نسبت به n از مرتبه‌ی نمایی هستند.

۴-۳۵ تصادفی‌سازی و برنامه‌ریزی خطی

در این بخش دو تکنیک مفید برای طراحی الگوریتم‌های تقریبی را می‌آموزیم: تصادفی‌سازی و برنامه‌ریزی خطی. یک الگوریتم تصادفی ساده برای یک نسخه‌ی بهینه‌سازی از قابلیت ارضای 3-CNF ارائه خواهیم کرد، و سپس از برنامه‌ریزی خطی استفاده کرده و یک الگوریتم تقریبی برای یک نسخه‌ی وزن‌دار از مسئله‌ی پوشش رأسی طراحی می‌کنیم. در این بخش فقط مقدمات استفاده از این دو تکنیک قدرتمند را خواهیم دید. در کتاب‌های دیگر می‌توانید در این زمینه بیشتر بیاموزید.

یک الگوریتم تقریبی تصادفی برای قابلیت ارضای MAX-3-CNF

همان طور که الگوریتم‌های تصادفی هستند که جواب‌های دقیق محاسبه می‌کنند، الگوریتم‌های تصافی هم هستند که جواب‌های تقریبی به ما می‌دهند. می‌گوییم یک الگوریتم تصادفی برای یک مسئله، نسبت کارایی $\rho(n)$ دارد اگر برای هر ورودی با اندازه‌ی n، *امیدریاضی* هزینه‌ی C مربوط به جواب تولید شده توسط الگوریتم تصادفی بیش از یک ضریب $\rho(n)$ با هزینه‌ی جواب بهینه‌ی C^* فاصله نداشته باشد:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n) \qquad (۱۳-۳۵)$$

همچنین به یک الگوریتم تصادفی که به نسبت کارایی $\rho(n)$ می‌رسد، یک *الگوریتم $\rho(n)$ - تقریبی تصادفی* می‌گوییم. به عبارت دیگر، یک الگوریتم تقریبی تصادفی مانند یک الگوریتم تقریبی قطعی است، غیر از این که در آن نسبت کارایی مربوط به یک امیدریاضی است.

یک نمونه‌ی خاص از قابلیت ارضای 3-CNF، که در بخش ۳۴-۴ تعریف شد، ممکن است قابل ارضا باشد یا نباشد. برای این که این نمونه قابل ارضا باشد، باید یک مقداردهی به متغیرها وجود

داشته باشد به طوری که مقدار تمام عبارت‌های آن ۱ شود. اگر یک نمونه قابل ارضا نباشد، ممکن است بخواهیم مقدار «نزدیکی» آن به ارضا شدن را تعیین کنیم، یعنی، ممکن است بخواهیم یک مقداردهی به متغیرها بیابیم که بیشترین تعداد عبارت‌های ممکن را ارضا می‌کند. به مسئله‌ی بیشینه‌سازی حاصل *قابلیت ارضای MAX-3-CNF* می‌گوییم. ورودی مسئله‌ی قابلیت ارضای MAX-3-CNF درست مانند ورودی قابلیت ارضای 3-CNF است، و هدف بازگرداندن یک مقداردهی به متغیرها است که تعداد عبارت‌های ۱ را بیشینه می‌کند. اکنون نشان می‌دهیم که مقداردهی ۱ به متغیرها با احتمال ۱/۲ و ۰ با احتمال ۱/۲، یک الگوریتم ۸/۷ - تقریبی تصادفی است. طبق تعریف قابلیت ارضای 3-CNF در بخش ۳۴-۴، باید هر عبارت دقیقاً از سه لفظ متمایز تشکیل شده باشد. همچنین فرض می‌کنیم که هیچ عبارتی حاوی یک متغیر و نقیض آن به صورت هم‌زمان نیست. (تمرین ۳۵-۴-۱ از شما می‌خواهد که این فرض آخر را حذف کنید.)

قضیه‌ی ۳۵-۶

با داشتن یک نمونه از مسئله‌ی قابلیت ارضای MAX-3-CNF با n متغیر $x_1, x_2, \ldots, x_n$ و m عبارت، یک الگوریتم تصادفی که به صورت مستقل به هر متغیر با احتمال ۱/۲ مقدار ۱ و با احتمال ۱/۲ مقدار ۰ می‌دهد، یک الگوریتم ۸/۷-تقریبی تصادفی است.

اثبات فرض کنید که به صورت مستقل، به هر متغیر با احتمال ۱/۲ مقدار ۱ و با احتمال ۱/۲ مقدار ۰ داده‌ایم. برای $i = 1, 2, \ldots, n$، متغیر تصادفی شاخص زیر را تعریف می‌کنیم:

$$Y_i = I\{\text{عبارت } i \text{ ارضا شده است}\}$$

به طوری که $Y_i = 1$ اگر یکی از لفظ‌ها در iامین عبارت با ۱ مقداردهی شده باشد. چون هر لفظ بیش از یک بار در یک عبارت ظاهر نمی‌شود، مقداردهی سه لفظ در هر عبارت مستقل است. یک عبارت فقط در صورتی ارضا نمی‌شود که هر سه لفظ آن مقدار ۰ داشته باشند، و بنابراین

$$\Pr\{\text{عبارت } i \text{ ارضا نشود}\} = (1/2)^3 = 1/8$$

پس،

$$\Pr\{\text{عبارت } i \text{ ارضا شود}\} = 1 - 1/8 = 7/8$$

بنابراین، طبق لم ۵-۱، $E[Y_i] = 7/8$. فرض کنید Y تعداد عبارت‌های ارضا شده باشد، یعنی $Y = Y_1 + Y_2 + \ldots + Y_m$. آن گاه داریم

$$\begin{aligned} E[Y] &= E\left[\sum_{i=1}^{m} Y_i\right] \\ &= \sum_{i=1}^{m} E[Y_i] \qquad \text{(طبق خطی بودن امیدریاضی)} \\ &= \sum_{i=1}^{m} 7/8 \\ &= 7m/8 \end{aligned}$$

به وضوح m یک کران بالا برای تعداد عبارت‌های ارضا شده است، و بنابراین نسبت کارایی حداکثر برابر است با $m/(۲m/۸) = ۸/۲$.

تقریب پوشش رأسی وزن دار با استفاده از برنامه‌ریزی خطی

در *مسئله‌ی پوشش رأسی با وزن کمینه* (minimum-weight vertex-cover problem)، یک گراف بدون جهت $G = (V, E)$ داریم، که در آن هر رأس $v \in V$ یک وزن مثبت $w(v)$ دارد. برای هر پوشش رأسی $V' \subseteq V$، وزن پوشش رأسی را به صورت $w(V') = \sum_{v \in V'} w(v)$ تعریف می‌کنیم. هدف یافتن یک پوشش رأسی با وزن کمینه است.

نمی‌توانیم از الگوریتم مورد استفاده برای پوشش رأسی بدون وزن در این جا استفاده کنیم، همین‌طور از یک جواب تصادفی هم نمی‌توانیم استفاده کنیم؛ هر دو متد ممکن است جواب‌هایی بدهند که با جواب بهینه فاصله‌ی زیادی دارند. در هر صورت می‌توانیم با استفاده از برنامه‌ریزی خطی یک کران پایین برای وزن پوشش رأسی با وزن کمینه بیابیم. سپس این جواب را «گرد» کرده و از آن برای به دست آوردن یک پوشش رأسی استفاده می‌کنیم.

فرض کنید که یک متغیر $x(v)$ به هر رأس $v \in V$ نسبت می‌دهیم، به طوری که $x(v) \in \{۰,۱\}$. رأس v در پوشش رأسی قرار می‌دهیم اگر و فقط اگر $x(v) = ۱$. آن گاه می‌توانیم این محدودیت را که برای هر یال (u,v) حداقل باید یکی از رأس‌های u و v در پوشش رأسی قرار داشته باشد، به صورت $x(u) + x(v) \geq ۱$ بنویسیم. این دید به *برنامه‌ی صحیح ۰-۱* (0-1 integer program) زیر برای یافتن پوشش رأسی با وزن کمینه منجر می‌شود:

کمینه‌سازی

$$\text{minimize} \quad \sum_{v \in V} w(v)x(v) \qquad (۱۴-۳۵)$$

با شرایط

$$x(u) + x(v) \geq ۱ \qquad \text{برای هر } (u,v) \in E \qquad (۱۵-۳۵)$$

$$x(v) \in \{۰,۱\} \qquad \text{برای هر } v \in V \qquad (۱۶-۳۵)$$

در حالت خاصی که در آن تمام وزن‌های $w(v)$ برابرند با ۱، این فرمول‌بندی نسخه‌ی بهینه‌سازی مسئله‌ی NP-سخت پوشش رأسی است. اکنون فرض کنید که محدودیت $x(v) \in \{۰,۱\}$ را برداشته و $۰ \leq x(v) \leq ۱$ را جایگزین می‌کنیم. در این صورت برنامه‌ی خطی زیر را به دست می‌آوریم، که به *ترمیم برنامه‌ریزی خطی* (linear-programming relaxation) معروف است:

$$\text{minimize} \quad \sum_{v \in V} w(v)x(v) \qquad (۱۷-۳۵)$$

با شرایط

$$x(u)+x(v)\geq 1 \qquad \text{برای هر } (u,v)\in E \qquad (۳۵-۱۸)$$

$$x(v)\leq 1 \qquad \text{برای هر } v\in V \qquad (۳۵-۱۹)$$

$$x(v)\geq 0 \qquad \text{برای هر } v\in V \qquad (۳۵-۲۰)$$

هر جواب ممکن برای برنامه‌ی صحیح ۰-۱ در خطوط (۳۵-۱۴)-(۳۵-۱۶)، یک جواب ممکن برای برنامه‌ی خطی در خطوط (۳۵-۱۷)-(۳۵-۲۰) هم هست. بنابراین یک جواب بهینه برای برنامه‌ی خطی، یک کران پایین برای جواب بهینه‌ی برنامه‌ی صحیح ۰-۱ است، و همچنین یک کران پایین برای یک جواب بهینه به مسئله‌ی پوشش رأسی با وزن کمینه.

رویه‌ی زیر از جواب برنامه‌ی خطی بالا برای ساختن یک جواب تقریبی برای مسئله‌ی پوشش رأسی با وزن کمینه استفاده می‌کند:

```
APPROX-MIN-WEIGHT-VC(G, w)
1  C = ∅
2  compute x̄, an optimal solution to the linear program in lines (35.17)-(35.20)
3  for each v ∈ V
4      if x̄(v) ≥ 1/2
5          C = C ∪ {v}
6  return C
```

رویه‌ی APPROX-MIN-WEIGHT-VC به صورت زیر کار می‌کند. خط ۱ پوشش رأسی را با یک مجموعه‌ی تهی مقداردهی اولیه می‌کند. خط ۲ برنامه‌ی خطی خطوط (۳۵-۱۷)-(۳۵-۲۰) را فرمول‌بندی کرده و سپس آن را حل می‌کند. یک جواب بهینه به هر رأس v یک مقدار $\bar{x}(v)$ نسبت می‌دهد به طوری که $0\leq \bar{x}(v)\leq 1$. از این مقدار استفاده می‌کنیم تا بتوانیم در خطوط ۳-۵ رأس مناسب را برای اضافه کردن به پوشش رأسی C انتخاب کنیم. اگر $\bar{x}(v)\geq 1/2$، رأس v را به C اضافه می‌کنیم؛ در غیر این صورت این کار را نمی‌کنیم. در عمل، ما داریم هر مقدار اعشاری در جواب برنامه‌ی خطی را به ۰ یا ۱ «گرد» می‌کنیم تا به یک جواب برای برنامه‌ی صحیح ۰-۱ در خطوط (۳۵-۱۴)-(۳۵-۱۶) برسیم. نهایتاً در خط ۶ پوشش رأسی C بازگردانده می‌شود.

قضیه‌ی ۳۵-۲ الگوریتم APPROX-MIN-WEIGHT-VC یک الگوریتم ۲-تقریبی چندجمله‌ای برای مسئله‌ی پوشش رأسی با وزن کمینه است.

اثبات چون یک الگوریتم چندجمله‌ای برای حل برنامه‌ی خطی در خط ۲ وجود دارد، و چون حلقه‌ی for در خطوط ۳-۵ در زمان چندجمله‌ای اجرا می‌شود، APPROX-MIN-WEIGHT-VC در زمان چندجمله‌ای اجرا می‌شود.

اکنون نشان می‌دهیم که APPROX-MIN-WEIGHT-VC یک الگوریتم ۲-تقریبی است. فرض کنید C^* یک جواب بهینه برای مسئله‌ی پوشش رأسی با وزن کمینه باشد، و z^* مقدار یک جواب بهینه برای برنامه‌ی خطی در خطوط (۳۵-۱۷)-(۳۵-۲۰). چون یک پوشش رأسی بهینه، یک جواب ممکن برای برنامه‌ی خطی است، z^* باید یک کران پایین برای $w(C^*)$ باشد، یعنی

$$z^* \le w(C^*) \tag{۳۵-۲۱}$$

سپس، ادعا می‌کنیم که با گرد کردن مقادیر اعشاری متغیرهای $\bar{x}(v)$، یک مجموعه‌ی C خواهیم ساخت که یک پوشش رأسی است و $w(C) \le ۲z^*$ را ارضا می‌کند. برای این که ببینیم C یک پوشش رأسی است، یک یال دلخواه $(u,v) \in E$ را در نظر بگیرید. طبق محدودیت (۳۵-۱۸)، می‌دانیم که $x(u)+x(v) \ge ۱$، که ایجاب می‌کند که حداقل یکی از $\bar{x}(u)$ و $\bar{x}(v)$ حداقل ۱/۲ است. بنابراین حداقل یکی از رأس‌های u و v در پوشش رأسی قرار دارند، و بنابراین تمام یال‌ها پوشش داده شده‌اند.

اکنون وزن این پوشش را در نظر می‌گیریم. داریم

$$\begin{aligned}
z^* &= \sum_{v \in V} w(v)\bar{x}(v) \\
&\ge \sum_{v \in V : \bar{x}(v) \ge ۱/۲} w(v)\bar{x}(v) \\
&\ge \sum_{v \in V : \bar{x}(v) \ge ۱/۲} w(v) \cdot ۱/۲ \\
&= \sum_{v \in C} w(v) \cdot ۱/۲ \\
&= \frac{۱}{۲} \sum_{v \in C} w(v) \\
&= \frac{۱}{۲} w(C)
\end{aligned} \tag{۳۵-۲۲}$$

ترکیب نامساوی‌های (۳۵-۲۱) و (۳۵-۲۲) به دست می‌دهد

$$w(C) \le ۲z^* \le ۲w(C^*)$$

و بنابراین APPROX-MIN-WEIGHT-VC یک الگوریتم ۲-تقریبی است. ■

تمرین‌ها

۳۵-۴-۱ نشان دهید که حتی اگر اجازه دهیم که یک عبارت هم حاوی خود یک متغیر باشد و هم حاوی نقیض آن، مقداردهی تصادفی متغیرها با ۱ با احتمال ۱/۲ و با ۰ با احتمال ۱/۲ همچنان یک الگوریتم ۸/۷-تقریبی تصادفی خواهد بود.

۳۵-۴-۲ *مسئله‌ی قابلیت ارضای MAX-CNF* مانند قابلیت ارضای MAX-3-CNF است، غیر از این که محدودیت وجود دقیقاً سه لفظ در هر عبارت را ندارد. یک الگوریتم ۲-تقریبی برای مسئله‌ی قابلیت ارضای MAX-CNF بدهید.

۳۵-۴-۳ در مسئله‌ی MAX-CUT، به ما یک گراف بدون جهت و بدون وزن $G = (V, E)$ داده می‌شود. یک برش $(S, V - S)$ را مانند فصل ۲۳، و *وزن* آن را به صورت تعداد یال‌های

عبور کننده از آن برش تعریف می‌کنیم. هدف یافتن یک برش با وزن بیشینه است. فرض کنید که برای هر رأس v، به صورت تصادفی و مستقل، با احتمال ۱/۲ رأس v را در S و با احتمال ۱/۲ آن را در $V - S$ قرار می‌دهیم. نشان دهید که این الگوریتم یک الگوریتم ۲-تقریبی تصادفی است.

۴-۴-۳۵ نشان دهید که محدودیت‌های خط (۳۵-۱۹) اضافه هستند، از این نظر که اگر آن‌ها را از برنامه‌ی خطی در خطوط (۳۵-۱۷)-(۳۵-۲۰) حذف کنیم، هر جواب بهینه به برنامه‌ی خطی حاصل باید $x(v) \leq 1$ را برای $v \in V$ ارضا کند.

۵-۳۵ مسئله‌ی جمع زیرمجموعه

از بخش ۳۴-۵-۵ به خاطر بیاورید که یک نمونه از مسئله‌ی جمع زیرمجموعه، یک جفت (S, t) است که در آن S یک مجموعه‌ی $\{x_1, x_2, \ldots, x_n\}$ از اعداد صحیح مثبت است، و t یک عدد صحیح مثبت. در این مسئله‌ی تصمیم‌گیری می‌خواهیم بدانیم که آیا یک زیرمجموعه از S وجود دارد که جمع اعضای آن دقیقاً برابر t شود یا خیر. همان طور که در بخش ۳۴-۵-۵ دیدیم، این مسئله NP-سخت است.

مسئله‌ی بهینه‌سازی متناظر با این مسئله‌ی تصمیم‌گیری در کاربردهای عملی مختلفی پیش می‌آید. در مسئله‌ی بهینه‌سازی، می‌خواهیم یک زیرمجموعه‌ی $\{x_1, x_2, \ldots, x_n\}$ بیابیم به طوری که جمع آن تا حد ممکن بزرگ باشد، ولی نه بزرگ‌تر از t. به عنوان مثال، ممکن است یک کامیون داشته باشیم که ظرفیت آن حداکثر t پوند است، به همراه n جعبه‌ی مختلف که می‌خواهیم آن‌ها را انتقال دهیم، و iامین جعبه x_i پوند وزن دارد. می‌خواهیم تا حد ممکن جعبه‌ها را در کامیون بار بزنیم، بدون این که از حد وزنی تعیین شده بگذریم.

در این بخش یک الگوریتم با زمان نمایی برای این مسئله‌ی بهینه‌سازی ارائه می‌کنیم، و سپس نشان می‌دهیم که چگونه می‌توان این الگوریتم را طوری اصلاح کرد که تبدیل به یک رویکرد تقریب کاملاً چندجمله‌ای شود. (به خاطر بیاورید که یک رویکرد تقریب کاملاً چندجمله‌ای، الگوریتمی است که زمان اجرای آن علاوه بر اندازه‌ی ورودی، نسبت به $1/\varepsilon$ هم از مرتبه‌ی چندجمله‌ای باشد.)

یک الگوریتم دقیق با زمان نمایی

فرض کنید که برای هر زیرمجموعه‌ی S' از S، مجموع عناصر S' را محاسبه کرده‌ایم، و سپس از میان زیرمجموعه‌هایی که جمع عناصر آن‌ها از t فراتر نمی‌رود، زیرمجموعه‌ای را انتخاب کرده‌ایم که مجموع عناصر آن از بقیه به t نزدیک‌تر است. بدیهتاً این الگوریتم جواب بهینه را بازمی‌گرداند، ولی می‌تواند در زمان نمایی اجرا شود. برای پیاده‌سازی این الگوریتم، می‌توانیم از یک رویه‌ی تکراری استفاده کنیم که در تکرار iام، مجموع تمام زیرمجموعه‌های $\{x_1, x_2, \ldots, x_i\}$ را محاسبه می‌کند، و برای این کار از جمع زیرمجموعه‌های $\{x_1, x_2, \ldots, x_{i-1}\}$ استفاده می‌کند. حین انجام این کار، متوجه

می‌شویم که اگر جمع عناصر یک زیرمجموعه‌ی S' از t فراتر رود، هیچ دلیلی برای نگه‌داری آن وجود ندارد، چرا که هیچ ابرمجموعه‌ای از S' نمی‌تواند جواب بهینه باشد. اکنون یک پیاده‌سازی برای این استراتژی ارائه می‌دهیم.

رویه‌ی EXACT-SUBSET-SUM به عنوان ورودی مجموعه‌ی $S = \{x_1, x_2, ..., x_n\}$ و مقدار هدف t را دریافت می‌کند؛ شبه‌کد آن را به زودی خواهیم دید. این رویه به صورت تکراری L_i را محاسبه می‌کند، که لیست مجموع تمام زیرمجموعه‌های $\{x_1, ..., x_i\}$ است که از t فراتر نمی‌روند، و سپس مقدار بیشینه در L_n را بازمی‌گرداند.

اگر L یک لیست از اعداد صحیح مثبت باشد، و x یک عدد صحیح مثبت دیگر، آن گاه فرض می‌کنیم $L + x$ نشان‌دهنده‌ی لیست اعداد صحیح به دست آمده از افزایش تمام عناصر L به اندازه‌ی x باشد. به عنوان مثال اگر $L = \langle 1, 2, 3, 5, 9 \rangle$، آن گاه $L + 2 = \langle 3, 4, 5, 7, 11 \rangle$. از این نماد برای مجموعه‌ها هم استفاده خواهیم کرد، به طوری که

$$S + x = \{s + x : s \in S\}$$

همچنین از یک رویه‌ی کمکی MERGE-LISTS (L, L') استفاده خواهیم کرد که لیست مرتب‌شده‌ای را بازمی‌گرداند که از ادغام دو لیست مرتب‌شده‌ی ورودی L و L' و حذف مقادیر تکراری به دست می‌آید. مانند رویه‌ی MERGE که از آن در مرتب‌سازی ادغامی استفاده کردیم (بخش ۲-۳-۱)، MERGE-LISTS در زمان $O(|L| + |L'|)$ اجرا می‌شود. از ارائه‌ی شبه‌کد برای MERGE-LISTS صرف نظر می‌کنیم.

```
EXACT-SUBSET-SUM(S, t)
1  n = |S|
2  L_0 = ⟨0⟩
3  for i = 1 to n
4      L_i = MERGE-LISTS(L_{i-1}, L_{i-1} + x_i)
5      remove from L_i every element that is greater than t
6  return the largest element in L_n
```

برای این که ببینیم EXACT-SUBSET-SUM چگونه کار می‌کند، فرض کنید P_i نشان‌دهنده‌ی مجموعه‌ی تمام مقادیری باشد که می‌توان با انتخاب یک زیرمجموعه‌ی (احتمالاً تهی) از $\{x_1, x_2, ..., x_i\}$ و جمع عناصر آن‌ها به دست آورد. به عنوان مثال اگر $S = \{1, 4, 5\}$، آن گاه

$$P_0 = \{0, 1\},$$

$$P_1 = \{0, 1, 4, 5\},$$

$$P_2 = \{0, 1, 4, 5, 6, 9, 10\}$$

با داشتن اتحاد

$$P_i = P_{i-1} \cup (P_{i-1} + x_i) \qquad (۳۵-۲۳)$$

می‌توانیم با استقرا بر روی i (تمرین ۳۵-۵-۱ را ببینید) اثبات کنیم که لیست L_i یک لیست مرتب

شده است، حاوی تمام عناصر P_i که مقدار آن‌ها از t بیشتر نیست. چون طول L_i می‌تواند تا 2^i بزرگ باشد، EXACT-SUBSET-SUM به طور کلی یک الگوریتم نمایی است، هر چند در حالت‌های خاصی که t نسبت به $|S|$ از مرتبه‌ی چندجمله‌ای باشد، یا تمام اعداد درون S از بالا با یک چندجمله‌ای نسبت به $|S|$ محدود شده باشند، این الگوریتم هم از مرتبه‌ی چندجمله‌ای است.

یک رویکرد تقریب کاملاً چندجمله‌ای

می‌توانیم با «هرس» کردن هر لیست L_i پس از ساخته شدن، به یک رویکرد تقریب کاملاً چندجمله‌ای برای مسئله‌ی جمع زیرمجموعه دست یابیم. ایده این است که اگر دو مقدار در L به یکدیگر نزدیک باشند، آن گاه به هدف یافتن یک جواب تقریبی، لزومی ندارد که هر دو مقدار را صریحاً نگه داریم. به طور دقیق‌تر از یک پارامتر هرس δ استفاده می‌کنیم به طوری که $0 < \delta < 1$. *هرس کردن* (trim) یک لیست L با ضریب δ یعنی حذف عناصر L تا جایی که امکان دارد، به طوری که اگر L' نتیجه‌ی هرس کردن L باشد، آن گاه برای هر عنصر y که از L حذف شده است، یک عنصر z هنوز در L' وجود دارد که y را تقریب می‌زند، یعنی

$$\frac{y}{1+\delta} \le z \le y \qquad (۳۵-۲۴)$$

می‌توانیم به z به چشم یک «نماینده»ی y در L' نگاه کنیم. هر عنصر حذف شده‌ی y یک نماینده‌ی z دارد، که عنصری است حذف نشده که در نامساوی (۳۵-۲۴) صدق می‌کند. به عنوان مثال اگر $\delta = 0.1$، و

$$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle$$

آن گاه می‌توانیم L را هرس کرده و به دست آوریم

$$L' = \langle 10, 12, 15, 20, 23, 29 \rangle$$

که در آن نماینده‌ی مقدار حذف شده‌ی ۱۱ مقدار ۱۰ است، نماینده‌ی مقادیر حذف شده‌ی ۲۱ و ۲۲ مقدار ۲۰، و نماینده‌ی مقدار حذف شده‌ی ۲۴ مقدار ۲۳. چون تمام عناصر نسخه‌ی هرس شده‌ی لیست عضوی از خود لیست هم هستند، هرس کردن می‌تواند به شدت تعداد عناصر را کاهش دهد، در حالی که یک مقدار نزدیک (و کمی کوچک‌تر) را به عنوان نماینده‌ی مقدار حذف شده نگه می‌دارد. رویه‌ی زیر با دریافت L و δ لیست $L = \langle y_1, y_2, \ldots, y_m \rangle$ را در زمان $\theta(m)$ هرس می‌کند، با فرض این که L به ترتیب صعودی یکنواخت مرتب شده است. خروجی این رویه یک لیست هرس و مرتب شده است.

```
TRIM(L, δ)
1  m = |L|
2  L' = ⟨y_1⟩
3  last = y_1
4  for i = 2 to m
5      if y_i > last · (1 + δ)        // y_i ≥ last because L is sorted
```

```
6        append y_i onto the end of L'
7        last = y_i
8  return L'
```

رویه عناصر L را به ترتیب صعودی یکنواخت بررسی می‌کند، و یک عدد در لیست بازگشتی L' قرار می‌گیرد فقط اگر اولین عنصر L باشد و یا آخرین عنصر قرار داده شده در L' نتواند نماینده‌ی آن باشد. با داشتن رویه‌ی TRIM می‌توانیم رویکرد تقریب خود را به صورت زیر بسازیم. این رویه به عنوان ورودی یک مجموعه‌ی $S = \{x_1, x_2, \ldots, x_n\}$ از n عدد صحیح (با یک ترتیب دلخواه)، یک هدف t (عدد صحیح)، و یک «پارامتر تقریب» ε را به عنوان ورودی دریافت می‌کند، به طوری که

$$0 < \varepsilon < 1 \qquad (۳۵-۲۵)$$

این رویه یک مقدار z بازمی‌گرداند به طوری که حداکثر به اندازه‌ی یک فاکتور $1+\varepsilon$ با جواب بهینه فاصله دارد.

```
APPROX-SUBSET-SUM(S, t, ε)
1  n = |S|
2  L_0 = ⟨0⟩
3  for i = 1 to n
4        L_i = MERGE-LISTS(L_{i-1}, L_{i-1}+x_i)
5        L_i = TRIM(L_i, ε/2n)
6        remove from L_i every element that is greater than t
7  let z* be the largest value in L_n
8  return z*
```

خط ۲ لیست L_0 را طوری مقداردهی اولیه می‌کند که فقط حاوی عنصر ۰ باشد. حلقه‌ی for در خطوط ۳-۶، L_i را به صورت یک لیست مرتب‌شده حاوی یک نسخه‌ی هرس شده از مجموعه‌ی P_i محاسبه می‌کند، که در آن تمام عناصر بزرگ‌تر از t حذف شده‌اند. چون L_i از L_{i-1} ساخته می‌شود، باید اطمینان حاصل کنیم که هرس مکرر، خطای بیش از اندازه ایجاد نمی‌کند. به زودی خواهیم دید که APPROX-SUBSET-SUM در صورت وجود یک تقریب صحیح بازمی‌گرداند.

به عنوان یک مثال، فرض کنید که نمونه‌ی

$$S = \langle 104, 102, 201, 101 \rangle$$

را داریم، با $t = 308$ و $\varepsilon = 0.40$. پارامتر هرس δ برابر است با $\varepsilon/8 = 0.05$. APPROX-SUBSET-SUM مقادیر زیر را در خطوط مشخص شده محاسبه می‌کند:

line 2: $L_0 = \langle 0 \rangle$,

line 4: $L_1 = \langle 0, 104 \rangle$,

line 5: $L_1 = \langle 0, 104 \rangle$,

line 6: $L_1 = \langle 0, 104 \rangle$,

line 4: $L_2 = \langle 0, 102, 104, 206 \rangle$,

line 5: $L_2 = \langle 0, 102, 206 \rangle$,
line 6: $L_2 = \langle 0, 102, 206 \rangle$,
line 4: $L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$,
line 5: $L_3 = \langle 0, 102, 201, 303, 407 \rangle$,
line 6: $L_3 = \langle 0, 102, 201, 303 \rangle$,
line 4: $L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$,
line 5: $L_4 = \langle 0, 101, 201, 302, 404 \rangle$,
line 6: $L_4 = \langle 0, 101, 201, 302 \rangle$

الگوریتم $z^* = 302$ را به عنوان جواب بازمی‌گرداند، که به خوبی در فاصله‌ی $\varepsilon = 40\%$ از جواب بهینه‌ی $101 + 102 + 104 = 307$ قرار دارد؛ در واقع، این جواب در فاصله‌ی ۲% قرار دارد.

قضیه‌ی ۸-۳۵ APPROX-SUBSET-SUM یک رویکرد تقریب کاملاً چندجمله‌ای برای مسئله‌ی جمع زیرمجموعه است.

اثبات اعمال هرس L_i در خط ۵ و حذف تمام عناصر بزرگ‌تر از t از L_i این خصوصیت را حفظ می‌کنند که تمام عناصر L_i، عنصری از P_i هم هستند. بنابراین مقدار z^* بازگردانده شده در خط ۸ مجموع عناصر یک زیرمجموعه از S است. فرض کنید $y^* \in P_n$ نشان‌دهنده‌ی یک جواب بهینه برای مسئله‌ی جمع زیرمجموعه باشد.

در این صورت، از خط ۶ می‌دانیم که $y^* \le z^*$. طبق نامساوی (۳۵-۱)، باید نشان دهیم که $y^*/z^* \le 1 + \varepsilon$. همچنین باید نشان دهیم که زمان اجرای این الگوریتم هم نسبت به اندازه‌ی ورودی از مرتبه‌ی چندجمله‌ای است و هم نسبت به $1/\varepsilon$.

همان طور که تمرین ۳۵-۵-۲ از شما می‌خواهد نشان دهید، برای هر عنصر y در P_i که حداکثر برابر t باشد یک عنصر $z \in L_i$ وجود دارد به طوری که

$$\frac{y}{(1+\varepsilon/2n)^i} \le z \le y \tag{۳۵-۲۶}$$

نامساوی (۳۵-۲۶) باید برای $y^* \in P_n$ برقرار باشد، و بنابراین یک $z \in L_n$ وجود دارد به طوری که

$$\frac{y^*}{(1+\varepsilon/2n)^n} \le z \le y^*$$

و بنابراین

$$\frac{y^*}{z} \le \left(1 + \frac{\varepsilon}{2n}\right)^n \tag{۳۵-۲۷}$$

چون یک $z \in L_n$ وجود دارد که نامساوی (۳۵-۲۷) را ارضا می‌کند، این نامساوی باید برای z^* هم برقرار باشد، که بزرگ‌ترین مقدار در L_n است؛ یعنی

$$\frac{y^*}{z^*} \le \left(1+\frac{\varepsilon}{2n}\right)^n \qquad (۳۵\text{-}۲۸)$$

این باقی می‌ماند که نشان دهیم $y^*/z^* \le 1+\varepsilon$. این کار را با نشان دادن $(1+\varepsilon/2n)^n \le 1+\varepsilon$ انجام می‌دهیم. طبق تساوی (۳-۱۴) داریم $\lim_{n\to\infty}(1+\varepsilon/2n)^n = e^{\varepsilon/2}$. تمرین ۳۵-۵-۳ از شما می‌خواهد نشان دهید که

$$\frac{d}{dn}\left(1+\frac{\varepsilon}{2n}\right)^n > 0 \qquad (۳۵\text{-}۲۹)$$

بنابراین تابع $(1+\varepsilon/2n)^n$ با رشد n و نزدیک شدن آن به حد $e^{\varepsilon/2}$ رشد می‌کند، و داریم

$$\begin{aligned}\left(1+\frac{\varepsilon}{2n}\right)^n &\le e^{\varepsilon/2} \\ &\le 1+\varepsilon/2+(\varepsilon/2)^2 && \text{(طبق نامساوی (۳-۱۳))} \\ &\le 1+\varepsilon && \text{(طبق نامساوی (۳۵-۲۵))}\end{aligned} \qquad (۳۵\text{-}۳۰)$$

ترکیب نامساوی‌های (۳۵-۲۸) و (۳۵-۳۰) تحلیل نسبت کارایی را کامل می‌کند.

برای این که نشان دهیم APPROX-SUBSET-SUM یک رویکرد تقریب کاملاً چندجمله‌ای است، یک کران بر روی طول L_i به دست می‌آوریم. پس از هرس، عناصر متوالی z و z' در L_i باید در رابطه‌ی $z'/z > 1+\varepsilon/2n$ صدق کنند. یعنی باید حداقل به اندازه‌ی یک فاکتور $1+\varepsilon/2n$ با هم تفاوت داشته باشند. بنابراین هر لیست حتماً حاوی مقدار ۰، احتمالاً مقدار ۱، و حداکثر $\lfloor \log_{1+\varepsilon/2n} t \rfloor$ مقدار دیگر است. تعداد عناصر در هر لیست L_i حداکثر برابر است با

$$\begin{aligned}\log_{1+\varepsilon/2n} t + 2 &= \frac{\ln t}{\ln(1+\varepsilon/2n)} + 2 \\ &\le \frac{2n(1+\varepsilon/2n)\ln t}{\varepsilon} + 2 && \text{(طبق نامساوی (۳-۱۷))} \\ &\le \frac{4n\ln t}{\varepsilon} + 2 && \text{(طبق نامساوی (۳۵-۲۵))}\end{aligned}$$

این کران نسبت به اندازه‌ی ورودی – که برابر است با تعداد بیت‌های مورد نیاز برای نمایش t به علاوه‌ی تعداد بیت‌های مورد نیاز برای نمایش مجموعه‌ی S، که خود نسبت به n از مرتبه‌ی چندجمله‌ای است – و نسبت به $1/\varepsilon$ از مرتبه‌ی چندجمله‌ای است. چون زمان اجرای APPROX-SUBSET-SUM نسبت به طول L_i چندجمله‌ای است، APPROX-SUBSET-SUM یک رویکرد تقریب کاملاً چندجمله‌ای است.

تمرین‌ها

۳۵-۵-۱ تساوی (۳۵-۲۳) را اثبات کنید. سپس نشان دهید که بعد از اجرای خط ۵ رویه‌ی EXACT-SUBSET-SUM، L_i یک لیست مرتب شده از تمام عناصر P_i است که مقدار آن‌ها بیش از t نیست.

۳۵-۵-۲ نامساوی (۳۵-۲۶) را اثبات کنید.

۳۵-۵-۳ نامساوی (۳۵-۲۹) را اثبات کنید.

۳۵-۵-۴ چگونه می‌توان رویکرد تقریب ارائه شده در این بخش را طوری اصلاح کرد که یک تقریب خوب برای کوچک‌ترین مقداری که از t کم‌تر نیست در مجموع عناصر یک زیرمجموعه از یک لیست داده شده بیابد؟

۳۵-۵-۵ رویه‌ی APPROX-SUBSET-SUM را طوری اصلاح کنید که زیرمجموعه‌ای از S که به مجموع z^* می‌رسد را هم بازگرداند.

مسائل

۳۵-۱ ***بار زدن در کیف***

فرض کنید که به ما یک مجموعه از n شیئ داده شده است، که اندازه‌ی s_i مربوط به iامین شیئ در $0 < s_i < 1$ صدق می‌کند. می‌خواهیم تمام این اشیا را با کم‌ترین تعداد کیف‌های با ظرفیت واحد بار بزنیم. هر کیف می‌تواند هر زیرمجموعه‌ی دلخواهی از اشیا را در خود نگه دارد، تا زمانی که مجموع اندازه‌ی آن‌ها از ۱ فراتر نرود.

I. اثبات کنید که مسئله‌ی تعیین کم‌ترین تعداد کیف‌های لازم NP-کامل است. (*راهنمایی:* کاهش را از مسئله‌ی جمع زیرمجموعه انجام دهید.)

مکاشفه‌ی *اولین تناسب* (first-fit) هر شیئ را به ترتیب در نظر گرفته و آن را در اولین کیفی قرار می‌دهد که ظرفیت کافی دارد. فرض کنید $S = \sum_{i=1}^{n} s_i$.

II. بحث کنید که تعداد بهینه‌ی کیف‌های مورد نیاز حداقل برابر است با $\lceil S \rceil$.

III. بحث کنید که مکاشفه‌ی اولین تناسب حداقل یک کیف را کم‌تر از نیمه‌پر رها می‌کند.

IV. اثبات کنید که تعداد کیف‌های استفاده شده در مکاشفه‌ی اولین تناسب هیچ گاه بیش از $\lceil 2S \rceil$ نیست.

V. اثبات کنید که نسبت کارایی مکاشفه‌ی اولین تناسب ۲ است.

VI. یک پیاده‌سازی کارآمد برای مکاشفه‌ی اولین تناسب ارائه و زمان اجرای آن را تحلیل کنید.

۳۵-۲ تقریب اندازه‌ی یک گروهک بیشینه

فرض کنید $G=(V,E)$ یک گراف بدون جهت باشد. برای $k \geq 1$، $G^{(k)}$ را به صورت یک گراف بدون جهت $(V^{(k)},E^{(k)})$ تعریف می‌کند، که در آن $V^{(k)}$ مجموعه‌ی تمام kتایی‌های مرتب از رأس‌های V است، و $E^{(k)}$ طوری تعریف شده است که $(v_1,v_2,...,v_k)$ مجاور $(w_1,w_2,...,w_k)$ است اگر و فقط اگر برای هر $1 \leq i \leq k$، یا رأس v_i در G مجاور w_i است، و یا $v_i=w_i$.

I. اثبات کنید که اندازه‌ی گروهک بیشینه در $G^{(k)}$ برابر است با kامین توان اندازه‌ی گروهک بیشینه در G.

II. بحث کنید که اگر یک الگوریتم تقریبی با نسبت کارایی ثابت برای یافتن یک گروهک بیشینه وجود داشته باشد، آن گاه یک رویکرد تقریب کاملاً چندجمله‌ای برای این مسئله وجود دارد.

۳۵-۳ مسئله‌ی پوشش مجموعه‌ی وزن‌دار

فرض کنید که مسئله‌ی پوشش رأسی را طوری گسترش می‌دهیم که هر مجموعه‌ی S_i در خانواده‌ی $\mathcal{F}$ یک وزن متناظر w_i داشته باشد، و وزن پوشش C را به صورت $\sum_{S_i \in C} w_i$ تعریف می‌کنیم. می‌خواهیم یک پوشش با وزن کمینه بیابیم. (بخش ۳۵-۳ حالتی را بررسی می‌کند که در آن $w_i=1$ برای تمام iها.)

نشان دهید که چطور می‌توان مکاشفه‌ی حریصانه‌ی پوشش رأسی به صورت طبیعی طوری گسترش داد که یک جواب تقریبی برای هر نمونه از مسئله‌ی پوشش مجموعه‌ی وزن‌دار فراهم کند. نشان دهید که نسبت کارایی مکاشفه‌ی شما $H(d)$ است، که در آن d اندازه‌ی بیشینه در میان مجموعه‌های S_i است.

۳۵-۴ تطابق بیشینه

به خاطر بیاورید که برای یک گراف بدون جهت G، یک تطابق عبارت است از مجموعه‌ای از یال‌ها به طوری که هیچ دو یالی در مجموعه با یک رأس مشترک مجاور نباشند. در بخش ۲۶-۳ دیدیم که چگونه می‌توان یک تطابق بیشینه در یک گراف دوبخشی یافت. در این مسئله نگاهی می‌اندازیم به تطابق‌ها در گراف‌های بدون جهت در حالت کلی (گراف‌هایی که لزوماً دوبخشی نیستند).

I. یک *تطابق ماکسیمال* (maximal matching)، تطابقی است که زیرمجموعه‌ی اکید هیچ تطابق دیگری نیست. نشان دهید که یک تطابق ماکسیمال لزوماً یک تطابق بیشینه نیست. برای این کار، باید یک گراف بدون جهت G و یک تطابق ماکسیمال M در G ارائه کنید که بیشینه نباشد. (*راهنمایی*: می‌توانید چنین گرافی بیابید که فقط چهار رأس دارد.)

II. یک گراف بدون جهت $G=(V,E)$ را در نظر بگیرید. یک الگوریتم حریصانه با زمان $O(E)$

برای یافتن یک تطابق ماکسیمال در G ارائه دهید.

در این مسئله، بر روی یک الگوریتم تقریبی چندجمله‌ای برای تطابق بیشینه تمرکز خواهیم کرد. در حالی که سریع‌ترین الگوریتم شناخته شده برای تطابق بیشینه به زمان فراخطی (ولی چندجمله‌ای) نیاز دارد، الگوریتم تقریبی که در این جا خواهیم یافت در زمان خطی اجرا خواهد شد. شما نشان خواهید داد که الگوریتم حریصانه‌ی خطی برای تطابق ماکسیمال در بخش II یک الگوریتم ۲–تقریبی برای تطابق بیشینه است.

III. نشان دهید که اندازه‌ی یک تطابق بیشینه در G یک کران پایین برای اندازه‌ی هر پوشش رأسی در G است.

IV. یک تطابق ماکسیمال M در $G = (V, E)$ را در نظر بگیرید. فرض کنید

$T = \{v \in V :$ یک یال در M با v مجاور است $\}$

در مورد زیرگراف G که از رأس‌هایی در G تشکیل شده است که در T نیستند، چه می‌توانید بگویید؟

V. از بخش IV نتیجه بگیرید که $|M|$ ۲ اندازه‌ی یک پوشش رأسی برای G است.

VI. با استفاده از بخش‌های III و V، اثبات کنید که الگوریتم حریصانه‌ی بخش II یک الگوریتم ۲–تقریبی برای تطابق بیشینه است.

۳۵-۵ برنامه‌ریزی ماشین‌های موازی

در مسئله‌ی برنامه‌ریزی ماشین‌های موازی، به ما n وظیفه‌ی J_1، J_2،...، J_n داده شده است، که در آن هر وظیفه‌ی J_k یک زمان پردازش نامنفی p_k دارد. همچنین m ماشین یکسان M_1، M_2 ،...، M_m داریم. هر وظیفه می‌تواند بر روی هر ماشین دلخواه انجام شود. یک *برنامه* (schedule)، برای هر وظیفه‌ی J_k ماشینی را تعیین می‌کند که این وظیفه باید بر روی آن اجرا شود، و همچنین دوره‌ی اجرای آن را. هر وظیفه‌ی J_k باید بر روی یک ماشین M_i برای p_k واحد زمان متوالی اجرا شود، و در این مدت هیچ وظیفه‌ی دیگری نمی‌تواند بر روی M_i اجرا شود. فرض کنید C_k نشان‌دهنده‌ی *زمان پایان*(completion time) وظیفه‌ی J_k باشد، یعنی، زمانی که پردازش وظیفه‌ی J_k پایان می‌یابد. با داشتن یک برنامه، *گستردگی*(makespan) برنامه را به صورت $C_{\max} = \max_{1 \le j \le n} C_k$ تعریف می‌کنیم. هدف یافتن یک برنامه با گستردگی کمینه است.

به عنوان مثال، فرض کنید که دو ماشین M_1 و M_2 داریم، و چهار وظیفه‌ی J_1، J_2، J_3، و J_4، با $p_1 = 2$، $p_2 = 12$، $p_3 = 4$، و $p_4 = 5$. در این صورت یک برنامه‌ی می‌تواند، بر روی ماشین M_1 ابتدا وظیفه‌ی J_1 را اجرا می‌کند و سپس وظیفه‌ی J_2 را، و بر روی ماشین M_2 ابتدا وظیفه‌ی J_4 و سپس وضیفه‌ی J_3. برای این برنامه، $C_1 = 2$، $C_2 = 14$، $C_3 = 9$، $C_4 = 5$، و $C_{\max} = 14$. یک برنامه‌ی بهینه، وظیفه‌ی J_2 را بر روی ماشین M_1 و وظایف J_1، J_3 و J_4 را بر روی ماشین M_2 اجرا می‌کند. برای این برنامه $C_1 = 2$، $C_2 = 12$، $C_3 = 6$، $C_4 = 11$، و $C_{\max} = 12$.

با داشتن یک مسئله‌ی برنامه‌ریزی ماشین‌های موازی، فرض می‌کنیم $C^*_{\max}$ نشان‌دهنده‌ی گستردگی یک برنامه‌ی بهینه باشد.

I. نشان دهید که گستردگی بهینه حداقل به بزرگی بزرگ‌ترین زمان پردازش است، یعنی

$$C^*_{\max} \geq \max_{1 \leq k \leq n} p_k$$

II. نشان دهید که گستردگی بهینه حداقل به بزرگی بار متوسط ماشین‌ها است، یعنی

$$C^*_{\max} \geq \frac{1}{m} \sum_{1 \leq k \leq n} p_k$$

فرض کنید که از الگوریتم حریصانه‌ی زیر برای برنامه‌ریزی ماشین‌های موازی استفاده می‌کنیم: هر گاه یک ماشین آزاد است، یک وظیفه که هنوز اجرا نشده است را به آن می‌دهیم.

III. یک شبه‌کد برای پیاده‌سازی این الگوریتم حریصانه بنویسید. زمان اجرای الگوریتم شما چقدر است؟

IV. برای برنامه‌ی بازگردانده شده توسط این الگوریتم حریصانه، نشان دهید که

$$C_{\max} \leq \frac{1}{m} \sum_{1 \leq k \leq n} p_k + \max_{1 \leq k \leq n} p_k$$

نتیجه بگیرید که این الگوریتم، یک الگوریتم ۲-تقریبی چندجمله‌ای است.

۳۵-۶ تقریب یک درخت پوشای بیشینه

فرض کنید $G = (V, E)$ یک گراف بدون جهت با وزن یکتای $w(u,v)$ برای هر یال $(u,v) \in E$ باشد. برای هر رأس $v \in V$ فرض کنید $\max(v) = \max_{(u,v) \in E} \{w(u,v)\}$ یک یال مجاور v باشد که وزن آن بیشینه است. فرض کنید $S_G = \{\max(v) : v \in V\}$ مجموعه‌ی یال‌های با وزن بیشینه‌ی مجاور با هر یال باشد، و T_G درخت پوشای بیشینه‌ی G باشد، یعنی یک درخت پوشا با بیشترین وزن ممکن. برای هر زیرمجموعه‌ی $E' \subseteq E$، تعریف می‌کنیم $w(E') = \sum_{(u,v) \in E'} w(u,v)$.

I. مثالی از یک گراف با حداقل ۴ رأس ارائه کنید که برای آن داشته باشیم $S_G = T_G$.

II. مثالی از یک گراف با حداقل ۴ رأس ارائه کنید که برای آن داشته باشیم $S_G \neq T_G$.

III. اثبات کنید که برای هر گراف G داریم $S_G \subseteq T_G$.

IV. اثبات کنید که برای هر گراف G داریم $w(T_G) \geq w(S_G)/2$.

V. الگوریتمی با زمان $O(V + E)$ ارائه کنید که یک ۲-تقریب برای مسئله‌ی درخت پوشای بیشینه محاسبه می‌کند.

۳۵-۷ یک الگوریتم تقریبی برای مسئله‌ی کوله‌پشتی ۰-۱

مسئله‌ی کوله‌پشتی را از بخش ۱۶-۲ به خاطر بیاورید. n عنصر داریم، که ارزش عنصر iام برابر است با v_i دلار و وزن آن برابر است با w_i پوند. همچنین یک کوله‌پشتی داریم که حداکثر ظرفیت آن W پوند است. در این جا این فرض‌ها را هم اضافه می‌کنیم که هر وزن

w_i حداکثر برابر است با W و عناصر به ترتیب نزولی ارزش اندیس‌گذاری شده‌اند: $v_1 \geq v_2 \geq \ldots \geq v_n$.

در مسئله‌ی کوله‌پشتی ۰-۱، می‌خواهیم زیرمجموعه‌ای از این عناصر بیابیم که وزن کل آن‌ها حداکثر W و مجموع ارزش آن‌ها بیشینه باشد. مسئله‌ی کوله‌پشتی کسری مانند مسئله‌ی کوله‌پشتی ۰-۱ است، با این تفاوت که در این مسئله اجازه داریم بخشی از هر عنصر را برداریم، و نیازی نیست که حتماً کل یک عنصر را برداریم یا برنداریم. اگر کسر x_i از عنصر i را برداریم، که در آن $0 \leq x_i \leq 1$، مقدار $x_i w_i$ به وزن کوله‌پشتی، و $x_i v_i$ دلار به ارزش اجناس برداشته شده اضافه کرده‌ایم. در این مسئله هدف ما این است که یک الگوریتم ۲-تقریب با زمان چندجمله‌ای برای مسئله‌ی کوله‌پشتی ۰-۱ بیابیم.

برای طراحی یک الگوریتم چندجمله‌ای، نمونه‌های محدود شده‌ی خاصی از مسئله‌ی کوله‌پشتی ۰-۱ را در نظر می‌گیریم. با داشتن نمونه‌ی I از مسئله، نمونه‌های محدود شده‌ی I_j (برای $j = 1, 2, \ldots, n$) را با حذف عناصر $1, 2, \ldots, j-1$ تشکیل می‌دهیم، با این شرط اضافه که جواب باید حاوی عنصر j باشد (تمام عنصر j در هر دو مسئله‌ی کوله‌پشتی ۰-۱ و کسری). برای نمونه‌ی I_1 هیچ عنصری حذف نمی‌شود. برای نمونه‌ی I_j، فرض کنید P_j نشان‌دهنده‌ی یک جواب بهینه برای مسئله‌ی ۰-۱ باشد، و Q_j یک جواب بهینه برای مسئله‌ی کسری.

I. بحث کنید که جواب بهینه‌ی نمونه‌ی I از مسئله‌ی کوله‌پشتی ۰-۱، یکی از $\{P_1, P_2, \ldots, P_n\}$ است.

II. اثبات کنید که می‌توانیم به روش زیر یک جواب بهینه‌ی Q_j برای نمونه‌ی I_j از مسئله‌ی کسری بیابیم: عنصر j را در کوله‌پشتی قرار داده و سپس از الگوریتم حریصانه استفاده می‌کنیم، که در هر مرحله تا حد ممکن از عنصر انتخاب نشده در $\{j+1, j+2, \ldots, n\}$ با بیشترین ارزش بر واحد وزن (v_i / w_i) برمی‌دارد.

III. اثبات کنید که همیشه می‌توانیم یک جواب بهینه‌ی Q_j برای نمونه‌ی I_j از مسئله‌ی کسری بیابیم که حداکثر یکی از عناصر را به صورت کسری برداشته است. یعنی برای تمام عنصر، غیر از احتمالاً یکی، یا تمام عنصر را در کوله‌پشتی قرار می‌دهیم و یا به کل از آن عنصر صرف نظر می‌کنیم.

IV. با داشتن یک جواب بهینه‌ی Q_j برای نمونه‌ی I_j از مسئله‌ی کسری، جواب R_j را با حذف تمام عناصر کسری از کوله‌پشتی بسازید. فرض کنید $v(S)$ نشان‌دهنده‌ی مجموع ارزش عناصر برداشته شده در جواب S باشد. اثبات کنید که $v(R_j) \geq v(Q_j)/2 \geq v(P_j)/2$.

V. یک الگوریتم چندجمله‌ای ارائه کنید که یک جواب با مقدار بیشینه از مجموعه‌ی $\{R_1, R_2, \ldots, R_n\}$ را بازمی‌گرداند، و اثبات کنید که این الگوریتم، یک ۲-تقریب با زمان چندجمله‌ای برای مسئله‌ی کوله‌پشتی ۰-۱ است.

بخش هشتم
پیوست‌ها:
زمینه‌ی ریاضی

شامل پیوست‌های :

الف سری‌ها

ب مجموعه‌ها و مباحث مربوطه

پ شمارش و احتمالات

ت ماتریس‌ها

مقدمه

معمولاً تحلیل الگوریتم‌ها نیاز به آشنایی با بدنه‌ای از ابزار ریاضی دارد. بعضی از این ابزارها به سادگی جبر دبیرستان هستند، ولی بعضی دیگر ممکن است کاملاً ناآشنا باشند. در بخش I دیدیم که چگونه با نمادهای حدی کار کرده و بازگشت‌ها را حل کنیم. این پیوست خلاصه‌ای است از مفاهیم و متدهای مختلف دیگری که از آن‌ها برای تحلیل الگوریتم‌ها استفاده می‌کنیم. همان طور که در مقدمه‌ی بخش I گفته شد، ممکن است بخش عمده‌ی این پیوست را قبل از خواندن این کتاب دیده باشید (هر چند ممکن است نمادهای خاص استفاده شده در این کتاب با کتاب‌های دیگر متفاوت باشد). بنابراین باید این پیوست را به صورت یک مرجع در نظر بگیرید. با این حال مانند بقیه‌ی این کتاب، در این بخش هم تمارین و مسائلی قرار داده‌ایم که می‌توانید به کمک آن‌ها مهارت خود را در این زمینه افزایش دهید.

در پیوست الف متدهایی ارائه می‌شود برای ارزیابی و تعیین کران سری‌ها، که به کرات در تحلیل الگوریتم‌ها پیش می‌آید. بسیاری از فرمول‌های این فصل را می‌توان در هر کتاب حسابانی یافت، ولی احتمالاً دسترسی به این مطالب به صورت یک‌جا می‌تواند مفید باشد.

پیوست ب حاوی تعاریف و نمادهایی است برای مجموعه‌ها، رابطه‌ها، توابع، گراف‌ها، و درخت‌ها. همچنین در این فصل بعضی از خصوصیات پایه‌ی این عناصر ریاضی را خواهید دید.

پیوست پ با قوانین پایه‌ای شمارش آغاز می‌شود: جایگشت‌ها، ترکیبیات و مانند این. ادامه‌ی فصل حاوی تعاریف و خصوصیات احتمالات مقدماتی است. برای تحلیل اکثر الگوریتم‌های این کتاب نیازی به احتمالات نیست، و بنابراین می‌توانید به سادگی از بخش‌های پایانی این فصل صرف نظر کنید، حتی

بدون مرور. بعداً زمانی که به یک تحلیل احتمالاتی بر می‌خورید که می‌خواهید آن را بهتر درک کنید، پوست پ را مرجعی مناسب و سازمان‌دهی شده خواهید یافت.

پیوست ت ماتریس‌ها را تعریف می‌کند، به علاوه‌ی اعمال مربوطه و بعضی خصوصیات آن‌ها. اگر درسی مشابه جبر خطی گذرانده باشید، احتمالاً با اکثر این مطالب آشنایی دارید، ولی باز هم خوب است که مرجعی داشته باشید برای تعاریف و مفاهیمی که در این کتاب از آن‌ها استفاده کرده‌ایم.

سری‌ها

الف – ۰ مقدمه

وقتی یک الگوریتم حاوی یک ساختار کنترلی تکراری مانند while یا for است، زمان اجرای آن را می‌توان به صورت جمع زمان‌های صرف شده در هر تکرار بدنه‌ی حلقه توصیف کرد. به عنوان مثال، در بخش ۲-۲ دیدیم که jامین تکرار مرتب‌سازی درجی در بدترین حالت به زمانی متناسب با j نیاز دارد. با جمع زمان صرف شده در هر تکرار، به مجموع (یا سری)

$$\sum_{j=۲}^{n} j$$

می‌رسیم. ارزیابی این سری یک کران $\theta(n^۲)$در بدترین حالت برای زمان اجرا به ما داد. این مثال اهمیت کلی درک نحوه‌ی کار و تعیین کران سری‌ها را روشن می‌کند.

بخش الف-۱ فرمول‌های پایه‌ای متعددی را لیست می‌کند که مربوط به سری‌ها می‌شوند. بخش الف-۲ تکنیک‌های مفیدی برای تعیین کران سری‌ها ارائه می‌کند. فرمول‌های بخش الف-۱ بدون اثبات داده شده‌اند، ولی اثبات بعضی از آن‌ها در بخش الف-۲ داده شده است تا نحوه‌ی کار با متدهای این بخش را روشن کند. بسیاری از اثبات‌های دیگر را می‌توانید در تمام کتاب‌های حسابان بیابید.

الف-۱ فرمول‌ها و خصوصیات سری‌ها

با داشتن یک دنباله‌ی $a_1,a_2,...$ از اعداد، مجموع متناهی $a_1+a_2+...+a_n$، که در آن n یک عدد صحیح نامنفی است، را می‌توان به صورت

$$\sum_{k=1}^{n} a_k$$

نوشت. اگر $n=0$، مقدار مجموع به صورت ۰ تعریف می‌شود. مقدار سری‌های متناهی همیشه خوش‌تعریف است، و جمله‌های آن را می‌توان به هر ترتیبی با یکدیگر جمع کرد.

با داشتن یک دنباله‌ی $a_1,a_2,...$ از اعداد، مجموع نامتناهی $a_1+a_2+....$ را می‌توان به صورت

$$\lim_{n\to\infty}\sum_{k=1}^{n} a_k$$

نوشت. اگر این حد وجود نداشته باشد، آن گاه سری *واگرا* است؛ در غیر این صورت، *همگرا* است. نمی‌توان همیشه جمله‌های یک سری همگرا را به ترتیب دلخواه با هم جمع کرد. با این حال، همیشه می‌توانیم جمله‌های یک *سری مطلقا همگرا* (absolutely convergent series) را بازآرایی کنیم، یعنی یک سری $\sum_{k=1}^{\infty} a_k$ به طوری که سری $\sum_{k=1}^{\infty}|a_k|$ هم همگرا باشد.

خطی بودن

برای هر عدد حقیقی c و دنباله‌های $a_1,a_2,...,a_n$ و $b_1,b_2,...,b_n$ داریم

$$\sum_{k=1}^{n}(ca_k+b_k)=c\sum_{k=1}^{n}a_k+\sum_{k=1}^{n}b_k$$

سری‌های نامتناهی همگرا هم از خصوصیت خطی بودن پیروی می‌کنند.

می‌توان از خصوصیت خطی بودن در اداره کردن سری‌های مربوط به نمادهای حدی بهره برد. برای مثال،

$$\sum_{k=1}^{n}\theta(f(k))=\theta\left(\sum_{k=1}^{n}f(k)\right)$$

در این تساوی، نماد θ در سمت چپ مربوط به متغیر k می‌شود، ولی در سمت راست این نماد متناظر با n است. این دستکاری‌ها را می‌توان بر روی سری‌های نامتنهای همگرا هم انجام داد.

سری‌های حسابی

مجموع

$$\sum_{k=1}^{n}k=1+2+...+n$$

یک *سری حسابی* (arighmetic series) است، با مقدار

$$\sum_{k=1}^{n} k = \frac{1}{2} n(n+1) \qquad \text{(الف-۱)}$$

$$= \theta(n^2) \qquad \text{(الف-۲)}$$

مجموع مربع‌ها و مکعب‌ها

سری‌های زیر را برای مربع‌ها و مکعب‌ها داریم:

$$\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \qquad \text{(الف-۳)}$$

$$\sum_{k=0}^{n} k^3 = \frac{n^2(n+1)^2}{4} \qquad \text{(الف-۴)}$$

سری‌های هندسی

برای عدد حقیقی $x \neq 1$، مجموع

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n$$

یک *سری هندسی* (geometric series) یا *نمایی* (exponential) است، با مقدار

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1} \qquad \text{(الف-۵)}$$

وقتی سری نامتناهی باشد و $|x| < 1$، سری هندسی نامتناهی نزولی زیر را داریم:

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \qquad \text{(الف-۶)}$$

سری‌های هارمونیک

برای اعداد صحیح n، *عدد هارمونیک* (harmonic numver) nام عبارت است از

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n}$$

$$= \sum_{k=1}^{n} \frac{1}{k} \qquad \text{(الف-۷)}$$

$$= \ln n + O(1)$$

(این کران را در بخش الف-۲ اثبات خواهیم کرد.)

انتگرال و مشتق سری‌ها

با انتگرال و مشتق گرفتن از فرمول‌های بالا می‌توان به فرمول‌های جدیدی رسید. به عنوان مثال با مشتق گرفتن از دو طرف سری هندسی نامتناهی (الف-۶) و ضرب آن در x، خواهیم داشت

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \qquad \text{(الف-۸)}$$

برای $|x| < 1$.

سری تلسکوپی

برای هر دنباله‌ی $a_0, a_1, \ldots, a_n$، داریم

$$\sum_{k=1}^{n} (a_k - a_{k-1}) = a_n - a_0 \qquad \text{(الف-۹)}$$

چرا که هر یک از جمله‌های $a_1, a_2, \ldots, a_{n-1}$ دقیقاً یک بار به سری اضافه شده و یک بار از آن کم می‌شود. می‌گوییم این سری *تلسکوپی* است. به طور مشابه

$$\sum_{k=0}^{n-1} (a_k - a_{k-1}) = a_0 - a_n$$

به عنوان یک مثال از یک مجموع تلسکوپی، سری

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)}$$

را در نظر بگیرید. از آن جایی که می‌توانیم هر جمله را به صورت

$$\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1}$$

بازنویسی کنیم، خواهیم داشت

$$\begin{aligned}\sum_{k=1}^{n-1} \frac{1}{k(k+1)} &= \sum_{k=1}^{n-1} \left(\frac{1}{k} - \frac{1}{k+1}\right) \\ &= 1 - \frac{1}{n}\end{aligned}$$

حاصل‌ضرب‌ها

حاصل‌ضرب متناهی $a_1 a_2 \ldots a_n$ را می‌توان به صورت

$$\prod_{k=1}^{n} a_k$$

بازنویسی کرد. اگر $n = 0$، مقدار ضرب به صورت ۱ تعریف می‌شود. می‌توانیم به کمک اتحاد زیر، یک فرمول حاوی یک حاصل‌ضرب را به یک فرمول حاوی یک سری تبدیل کنیم:

$$\lg\left(\prod_{k=1}^{n} a_k\right) = \sum_{k=1}^{n} \lg a_k$$

تمرین‌ها

الف-۱-۱ یک فرمول ساده برای $\sum_{k=1}^{n}(2k-1)$ بیابید.

★ الف-۱-۲ با استفاده از سری هارمونیک نشان دهید که $\sum_{k=1}^{n} 1/(2k-1) = \ln(\sqrt{n}) + O(1)$.

الف-۱-۳ نشان دهید که $\sum_{k=0}^{\infty} k^2 x^k = x(1+x)/(1-x)^3$ برای $0 < |x| < 1$.

★ الف-۱-۴ نشان دهید که $\sum_{k=0}^{\infty} (k-1)/2^k = 0$.

★ الف-۱-۵ مجموع $\sum_{k=1}^{\infty}(2k+1)x^{2k}$ را ارزیابی کنید.

الف-۱-۶ با استفاده از خصوصیت خطی بودن سری‌ها اثبات کنید که

$$\sum_{k=1}^{n} O(f_k(n)) = O\left(\sum_{k=1}^{n} f_k(n)\right).$$

الف-۱-۷ حاصل‌ضرب $\prod_{k=1}^{n} 2 \times 4^k$ را ارزیابی کنید.

★ الف-۱-۸ حاصل‌ضرب $\prod_{k=2}^{n}(1-1/k^2)$ را ارزیابی کنید.

الف-۲ تعیین کران سری‌ها

تکنیک‌های بسیاری برای تعیین کران سری‌هایی که زمان اجرای الگوریتم‌ها را توصیف می‌کنند، وجود دارد. در این جا بعضی از پرکاربردترین‌ها را خواهیم دید.

استقرای ریاضی

اساسی‌ترین روش برای ارزیابی یک سری استفاده از استقرای ریاضی است. به عنوان یک مثال، اجازه دهید اثبات کنیم که مقدار سری حسابی $\sum_{k=1}^{n} k$ برابر است با $n(n+1)/2$. می‌توانیم درستی این عبارت را برای $n = 1$ به سادگی بررسی کنیم، پس فرض استقرا را برای n در نظر گرفته و اثبات می‌کنیم که برای $n+1$ هم برقرار است. داریم

$$\begin{aligned}\sum_{k=1}^{n+1} k &= \sum_{k=1}^{n} k + (n+1)\\ &= \frac{1}{2}n(n+1) + (n+1)\\ &= \frac{1}{2}(n+1)(n+2)\end{aligned}$$

برای استفاده از استقرای ریاضی نیازی نیست که مقدار دقیق سری را حدس بزنیم. از استقرا می‌توان برای نشان دادن درستی یک کران هم استفاده کرد. به عنوان یک مثال، اجازه دهید اثبات کنیم که سری هندسی $\sum_{k=0}^{n} 3^k$ از مرتبه‌ی $O(3^n)$ است. به صورت دقیق‌تر، می‌خواهیم اثبات کنیم که $\sum_{k=0}^{n} 3^k \le c3^n$ برای یک ثابت c. برای حالت پایه‌ی $n=0$، داریم $\sum_{k=0}^{0} 3^k = 1 \le c \cdot 1$ تا زمانی که $c \ge 1$ باشد. با فرض این که این کران برای n برقرار است، اثبات می‌کنیم که برای $n+1$ هم برقرار است. داریم

$$\begin{aligned}\sum_{k=0}^{n+1} 3^k &= \sum_{k=0}^{n} 3^k + 3^{n+1}\\ &\le c3^n + 3^{n+1} \qquad \text{(طبق فرض استقرا)}\\ &= \left(\frac{1}{3} + \frac{1}{c}\right) c3^{n+1}\\ &\le c3^{n+1}\end{aligned}$$

در صورتی که $(1/3 + 1/c) \le 1$، یا به طور معادل، $c \ge 3/2$. بنابراین $\sum_{k=0}^{n} 3^k = O(3^k)$، که همان چیزی است که قصد اثبات آن را داشتیم.

هنگام استفاده از نمادهای حدی برای اثبات کران از طریق استقرا باید مواظب باشیم. اثبات سفسطه‌آمیز زیر را برای $\sum_{k=1}^{n} k = O(n)$ را در نظر بگیرید. بدیهتا $\sum_{k=1}^{1} k = O(1)$. با فرض درستی کران برای n، آن را برای $n+1$ اثبات می‌کنیم:

$$\begin{aligned}\sum_{k=1}^{n+1} k &= \sum_{k=1}^{n} k + (n+1)\\ &= O(n) + (n+1) \Leftarrow \text{غلط!!}\\ &= O(n+1)\end{aligned}$$

مشکل این بحث این است که «ثابت» مخفی در نماد «اوی بزرگ» به همراه n رشد می‌کند، و بنابراین ثابت نیست. در این جا نشان نداده‌ایم که ثابت یکسانی برای *تمام* nها کار می‌کند.

تعیین کران جمله‌ها

بعضی مواقع می‌توان با تعیین کران برای جمله‌های یک سری، یک کران بالای خوب برای خود سری یافت، و معمولاً استفاده از کران بزرگ‌ترین جمله برای تمام جمله‌های دیگر هم کافی است. به عنوان

مثال یک کران بالای سریع برای سری حسابی (الف-۱) عبارت است از

$$\begin{aligned}\sum_{k=1}^{n} k &\le \sum_{n=1}^{n} n \\ &= n^2\end{aligned}$$

به طور کلی برای یک سری $\sum_{k=1}^{n} a_k$، اگر قرار دهیم $a_{\max} = \max_{1 \le k \le n} a_k$، آن گاه

$$\sum_{k=1}^{n} a_k \le n \cdot a_{\max}$$

وقتی بتوان یک سری را با یک سری هندسی دیگر کران‌دار کرد، تکنیک تعیین کران هر جمله در یک سری توسط بزرگ‌ترین جمله تکنیک ضعیفی است. با داشتن سری $\sum_{k=0}^{n} a_k$، فرض کنید که برای تمام $k \ge 0$ داریم $a_{k+1}/a_k \le r$، که در آن $0 < r < 1$ یک ثابت است. این مجموع را می‌توان توسط یک سری هندسی نامتنهای نزولی کران‌دار کرد، چرا که $a_k \le a_0 r^k$، و بنابراین

$$\begin{aligned}\sum_{k=0}^{n} a_k &\le \sum_{k=0}^{\infty} a_0 r^k \\ &= a_0 \sum_{k=0}^{\infty} r^k \\ &= a_0 \frac{1}{1-r}\end{aligned}$$

★ **می‌توانیم از این متد برای تعیین کران *الف-۱-۲*:**

استفاده کنیم. برای شروع مجموع از $k = 0$، آن را به صورت $\sum_{k=0}^{\infty} ((k+1)/3^{k+1})$ بازنویسی می‌کنیم. جمله‌ی اول (a_0) برابر است با ۱/۳، و نسبت جمله‌های متوالی (r) برابر است با

$$\begin{aligned}\frac{(k+2)/3^{k+2}}{(k+1)/3^{k+1}} &= \frac{1}{3} \cdot \frac{k+2}{k+1} \\ &\le \frac{2}{3}\end{aligned}$$

برای تمام $k \ge 0$. بنابراین داریم

$$\begin{aligned}\sum_{k=1}^{\infty} \frac{k}{3^k} &= \sum_{k=0}^{\infty} \frac{k+1}{3^{k+1}} \\ &\le \frac{1}{3} \cdot \frac{1}{1-2/3} \\ &= 1\end{aligned}$$

یک خطای معمول در استفاده از این متد این است که نشان دهیم نسبت جمله‌های متوالی کم‌تر از ۱ است و سپس فرض کنیم که مجموع توسط یک سری هندسی محدود شده است. یک مثال، سری

هارمونیک نامتناهی است، که واگرا است، چرا که

$$\begin{aligned}\sum_{k=1}^{\infty}\frac{1}{k} &= \lim_{n\to\infty}\sum_{k=1}^{n}\frac{1}{k}\\ &= \lim_{n\to\infty}\theta(\lg n)\\ &= \infty\end{aligned}$$

نسبت جمله‌ی $(k+1)$ام به جمله‌ی kام در این سری $k/(k+1) < 1$ است، ولی این سری توسط یک سری هندسی محدود نمی‌شود. برای کران‌دار کردن یک سری توسط یک سری هندسی، باید نشان دهیم که یک $r < 1$ وجود دارد، که ثابت است، به طوری که نسبت تمام جفت‌ها از جمله‌های متوالی هیچ گاه از r فراتر نمی‌رود. در سری هارمونیک چنین rی وجود ندارد، چرا که این نسبت را می‌توان به دلخواه به ۱ نزدیک کرد.

تقسیم‌بندی مجموع‌ها

یک روش برای به دست آوردن یک کران برای یک سری این است که آن را به صورت مجموع دو یا چند سری توصیف کنیم، بدین صورت که بازه‌ی اندیس سری را تقسیم‌بندی کرده و کران هر یک از سری‌های حاصل را تعیین کنیم. برای مثال فرض کنید که می‌خواهیم یک کران پایین برای سری حسابی $\sum_{k=1}^{n} k$ بیابیم، که قبلا نشان دادیم که کران بالای آن n^2 است. ممکن است سعی کنیم هر یک از جمله‌ها را از پایین با کوچک‌ترین جمله محدود کنیم، ولی چون این جمله ۱ است، به کران پایین n برای این سری می‌رسیم – بسیار دورتر از کران بالای n^2.

می‌توانیم با قسمت کردن مجموع به یک کران پایین بهتر دست یابیم. برای سادگی فرض کنید که n زوج است. داریم

$$\begin{aligned}\sum_{k=1}^{n} k &= \sum_{k=1}^{n/2} k + \sum_{k=n/2+1}^{n} k\\ &\geq \sum_{k=1}^{n/2} 0 + \sum_{k=n/2+1}^{n} (n/2)\\ &= (n/2)^2\\ &= \Omega(n^2)\end{aligned}$$

که به صورت حدی نزدیک است، چرا که $\sum_{k=1}^{n} k = O(n^2)$.

برای یک سری که از تحلیل یک الگوریتم به دست می‌آید، معمولاً می‌توانیم آن را تقسیم‌بندی کرده و از اعداد ثابت جمله‌های اول صرف نظر کنیم. به طور کلی وقتی که هر جمله‌ی a_k در سری $\sum_{k=0}^{n} a_k$ مستقل از n باشد، این تکنیک کاربرد دارد. در این صورت، برای هر $k_0 > 0$ می‌توانیم بنویسیم

$$\sum_{k=0}^{n} a_k = \sum_{k=0}^{k_0-1} a_k + \sum_{k=k_0}^{n} a_k$$
$$= (1) + \sum_{k=k_0}^{n} a_k$$

چرا که جمله‌های اول سری همگی ثابت هستند، و تعداد ثابتی از آن‌ها وجود دارد. سپس می‌توانیم از متدهای دیگر استفاده کرده و کران $\sum_{k=k_0}^{n} a_k$ را تعیین کنیم. این تکنیک را می‌توان برای سری‌های نامتناهی هم به کار برد. برای مثال برای یافتن یک کران حدی بالا برای

$$\sum_{k=0}^{\infty} \frac{k^2}{2^k}$$

مشاهده می‌کنیم که نسبت جمله‌های متوالی برابر است با

$$\frac{(k+1)^2/2^{k+1}}{k^2/2^k} = \frac{(k+1)^2}{2k^2}$$
$$\leq \frac{8}{9}$$

اگر $k \geq 3$. بنابراین، این سری را می‌توان به صورت زیر تقسیم‌بندی کرد:

$$\sum_{k=0}^{\infty} \frac{k^2}{2^k} = \sum_{k=0}^{2} \frac{k^2}{2^k} + \sum_{k=3}^{\infty} \frac{k^2}{2^k}$$
$$\leq \sum_{k=0}^{2} \frac{k^2}{2^k} + \frac{9}{8} \sum_{k=0}^{\infty} \left(\frac{8}{9}\right)^k$$
$$= O(1)$$

چرا که تعداد جمله‌های سری اول ثابت است، و سری دوم یک سری هندسی نزولی است.

از تکنیک تقسیم‌بندی سری‌ها می‌توان برای تعیین کران‌های حدی در وضعیت‌های بسیار پیچیده استفاده کرد. برای مثال می‌توانیم یک کران $O(\lg n)$ برای سری هارمونیک (الف–۷) بیابیم:

$$H_n = \sum_{k=1}^{n} \frac{1}{k}$$

ایده این است که بازه‌ی ۱ تا n را به $\lfloor \lg n \rfloor + 1$ بخش تقسیم کرده و یک کران بالای ۱ برای هر بخش تعیین کنیم. برای $i = 0, 1, \ldots, \lfloor \lg n \rfloor$ ، بخش iام از جمله‌هایی تشکیل شده است که با $1/2^i$ شروع شده و تا $1/2^{i+1}$ بالا می‌رود (غیر از خود $1/2^{i+1}$)، که به دست می‌دهد

$$\sum_{k=1}^{n} \frac{1}{k} \leq \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i - 1} \frac{1}{2^i + j}$$

$$\le \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i}$$
$$= \sum_{i=0}^{\lfloor \lg n \rfloor} 1$$
$$\le \lg n + 1 \qquad \text{(الف-۱۰)}$$

تقریب به کمک انتگرال

وقتی می‌توان یک سری را به صورت $\sum_{k=m}^{n} f(k)$ توصیف کرد، که در آن $f(k)$ یک تابع صعودی یکنواخت است، می‌توانیم آن را به صورت انتگرال‌های زیر تقریب بزنیم:

$$\int_{m-1}^{n} f(x)dx \le \sum_{k=m}^{n} f(k) \le \int_{m}^{n+1} f(x)dx \qquad \text{(الف-۱۱)}$$

بررسی صحت این تقریب در شکل الف-۱ نشان داده شده است. مجموع به صورت مساحت مستطیل‌های درون شکل نمایش داده شده است، و انتگرال، ناحیه‌ی سایه خورده‌ی زیر منحنی است. وقتی $f(k)$ یک تابع نزولی اکید باشد می‌توانیم از یک متد مشابه برای تعیین کران استفاده کنیم:

$$\int_{m}^{n+1} f(x)dx \le \sum_{k=m}^{n} f(k) \le \int_{m-1}^{n} f(x)dx \qquad \text{(الف-۱۲)}$$

$$\sum_{k=1}^{n} \frac{1}{k} \ge \int_{1}^{n+1} \frac{dx}{x} \qquad \text{(الف-۱۳)}$$
$$= \ln(n+1)$$

برای کران بالا، نامساوی

$$\sum_{k=2}^{n} \frac{1}{k} \le \int_{1}^{n} \frac{dx}{x}$$
$$= \ln n$$

را به دست می‌آوریم، که کران

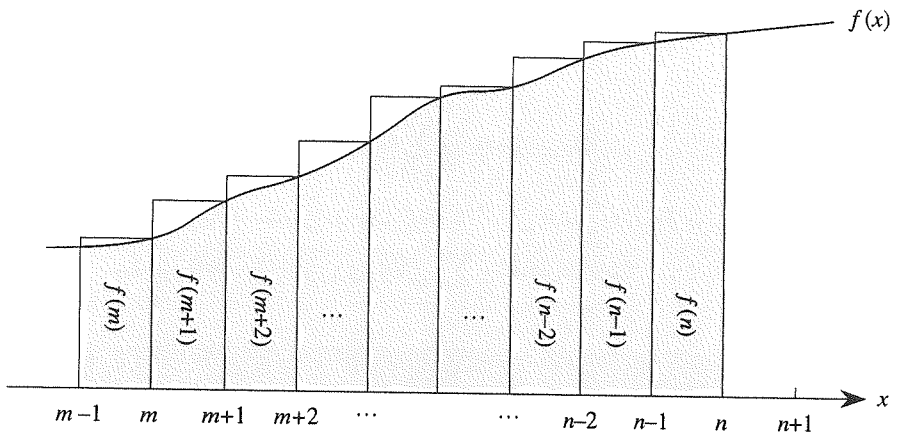
$$\sum_{k=1}^{n} \frac{1}{k} \le \ln n + 1 \qquad \text{(الف-۱۴)}$$

را به ما می‌دهد.

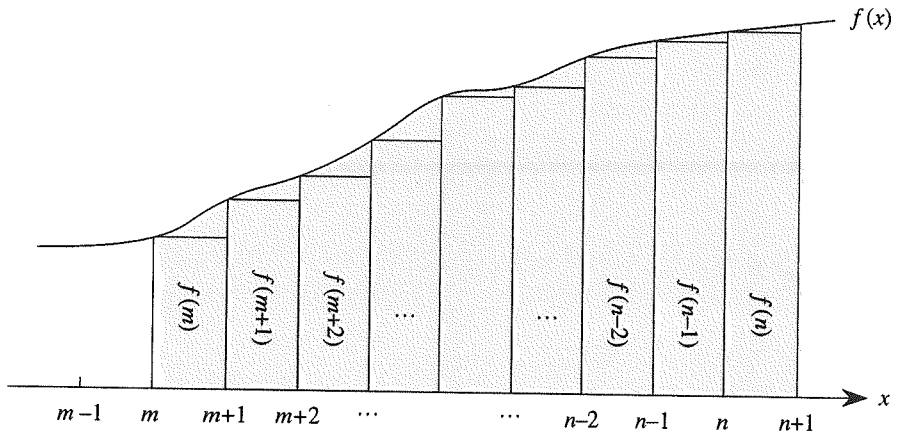
تقریب انتگرال در (الف-۱۲) یک تخمین نزدیک برای nامین سری هارمونیک به ما می‌دهد. برای یک کران پایین، به دست می‌آوریم

تمرین‌ها

نشان دهید که $\sum_{k=1}^{n} 1/k^2$ از بالا با یک ثابت محدود می‌شود.



(الف)



(ب)

شکل الف-۱ تقریب $\sum_{k=m}^{n} f(k)$ به کمک انتگرال. مساحت هر مستطیل درون آن نشان داده شده است، و مجموع مساحت مستطیل‌ها نشان دهنده‌ی مقدار سری است. انتگرال به صورت ناحیه‌ی سایه‌دار زیر منحنی نشان داده شده است. با مقایسه‌ی ناحیه‌ها در (الف) به دست می‌آوریم $\int_{m-1}^{n} f(x)dx \le \sum_{k=m}^{n} f(k)$، و سپس با جابه‌جا کردن مستطیل‌ها به اندازه‌ی یک واحد به راست، در (ب) خواهیم داشت $\sum_{k=m}^{n} f(k) \le \int_{m}^{n+1} f(x)dx$.

الف-۲-۴ یک کران بالای حدی برای سری زیر بیابید:

$$\sum_{k=0}^{\lfloor \lg n \rfloor} \lceil n/2^k \rceil$$

الف-۲-۳ با استفاده از تقسیم‌بندی سری‌ها نشان دهید که nامین سری هارمونیک از مرتبه‌ی $\Omega(\lg n)$ است.

الف-۲-۴ به کمک یک انتگرال، سری $\sum_{k=1}^{n} k^3$ را تخمین بزنید.

الف-۲-۵ چرا برای به دست آوردن یک کران بالا برای nامین عدد هارمونیک مستقیماً از تقریب انتگرالی (الف-۱۲) بر روی $\sum_{k=1}^{n} 1/k$ استفاده نکردیم؟

مسائل

الف-۱ ***تعیین کران مجموع‌ها***

برای مجموع‌های زیر، کران‌های حدی نزدیک ارائه کنید. فرض کنید که $r \geq 0$ و $s \geq 0$ ثابت هستند.

i. $\sum_{k=1}^{n} k^r$

ii. $\sum_{k=1}^{n} \lg^s k$

iii. $\sum_{k=1}^{n} k^r \lg^s k$

پیوست

ب

مجموعه‌ها و مباحث مربوطه

بسیاری از فصل‌های این کتاب برخوردهایی با عناصر ریاضیات گسسته دارند. این فصل به صورت کامل‌تر نمادها، تعاریف، و خصوصیات پایه‌ی مجموعه‌ها، رابطه‌ها، توابع، گراف‌ها، و درخت‌ها را بازبینی می‌کند. خواننده‌هایی که از قبل در این زمینه‌ها تبحر کافی دارند می‌توانند به مرور این فصل اکتفا کنند.

ب-۱ مجموعه‌ها

یک *مجموعه* (set)، گروهی است از اشیاء قابل تشخیص، که به *اعضا* (member) یا *عناصر* (element) آن مجموعه معروفند. اگر یک شیئ x عضو یک مجموعه‌ی S باشد، می‌نویسیم $x \in S$ (بخوانید «x یک عضو از S است»، یا به طور خلاصه‌تر، «x در S است»). اگر x عضوی از S نباشد، می‌نویسیم $x \notin S$. برای نمایش یک مجموعه می‌توانیم تمام اعضای آن را صریحا درون آکولادها لیست کنیم. برای مثال، یک مجموعه‌ی S که حاوی اعداد ۱، ۲، و ۳ است را به صورت $S = \{1,2,3\}$ می‌نویسیم. چون ۲ یک عضو در مجموعه‌ی S است، می‌توانیم بنویسیم $2 \in S$، و چون ۴ عضو آن نیست، داریم $4 \notin S$. یک مجموعه نمی‌تواند بیش از یک بار حاوی یک شیئ خاص باشد[1]، و عناصر آن ترتیب ندارند. دو مجموعه‌ی A و B *معادل* (equal) هستند اگر حاوی عناصر یکسان باشند، که آن را به صورت $A = B$ می‌نویسیم. برای مثال، $\{3,2,1\} = \{1,2,3\} = \{1,2,3,1\}$.

برای مجموعه‌هایی که مکررا با آن‌ها برخورد می‌کنیم، نمادهای خاصی تعریف خواهیم کرد.

- $\varnothing$ نشان‌دهنده‌ی *مجموعه‌ی تهی* (empty set) است، یعنی مجموعه‌ای که حاوی هیچ عنصری نیست.

[1] نسخه‌ای از مجموعه‌ها که می‌توانند بیش از یک نسخه از یک شیئ را داشته باشند، مجموعه‌های چندتایی (multiset) نام دارند.

- $\mathbb{Z}$ نشان‌دهنده‌ی مجموعه‌ی *اعداد صحیح* (integers) است، یعنی مجموعه‌ی $\{\ldots,-۲,-۱,۰,۱,۲,\ldots\}$.
- $\mathbb{R}$ نشان‌دهنده‌ی مجموعه‌ی *اعداد حقیقی* (real numbers) است.
- $\mathbb{N}$ نشان‌دهنده‌ی مجموعه‌ی *اعداد طبیعی* (natural number) است، یعنی مجموعه‌ی $\{۰,۱,۲,\ldots\}$[۱].

اگر تمام عناصر یک مجموعه‌ی A در یک مجموعه‌ی B هم باشند، یعنی اگر $x \in A$ نتیجه دهد $x \in B$، آن گاه می‌نویسیم $A \subseteq B$ و می‌گوییم A یک *زیرمجموعه* (subset) از B است. یک مجموعه‌ی A *زیرمجموعه‌ی اکید* (proper subset) B است، و آن را به صورت $A \subset B$ می‌نویسیم، اگر $A \subseteq B$ ولی $A \neq B$. (بعضی از نویسنده‌ها از سمبل "$\subset$" برای نشان دادن رابطه‌ی زیرمجموعه‌های معمولی استفاده می‌کنند، به جای رابطه‌ی زیرمجموعه‌های اکید.) برای هر مجموعه‌ی A، داریم $A \subseteq A$. برای دو مجموعه‌ی A و B، داریم $A = B$ اگر و فقط اگر $A \subseteq B$ و $B \subseteq A$. برای هر سه مجموعه‌ی A، B، و C، اگر $A \subseteq B$ و $B \subseteq C$، آن گاه $A \subseteq C$. برای هر مجموعه‌ی A داریم $\varnothing \subseteq A$.

بعضی مواقع مجموعه‌ها را بر حسب مجموعه‌های دیگر تعریف می‌کنیم. با داشتن یک مجموعه‌ی A، می‌توانیم یک مجموعه‌ی $B \subseteq A$ را با یک خصوصیت تعریف کنیم که عناصر B را مشخص می‌کند. به عنوان مثال می‌توانیم مجموعه‌ی اعداد زوج را به صورت

$\{x : x \in \mathbb{Z}$ و $x/۲$ یک عدد صحیح است$\}$

تعریف کنیم. دونقطه در این تعریف، «به طوری که» خوانده می‌شود. (بعضی نویسنده‌ها از یک خط عمودی به جای دونقطه استفاده می‌کنند.)

با داشتن دو مجموعه‌ی A و B، می‌توانیم با استفاده از *اعمال مجموعه‌ها* (set operations)، مجموعه‌های جدید تعریف کنیم:

- *اشتراک* (intersection) دو مجموعه‌ی A و B عبارت است از مجموعه‌ی

$$A \cap B = \{x : x \in A \text{ و } x \in B\}$$

- *اجتماع* (union) دو مجموعه‌ی A و B عبارت است از مجموعه‌ی

$$A \cup B = \{x : x \in A \text{ یا } x \in B\}$$

- *تفاضل* (difference) میان دو مجموعه‌ی A و B عبارت است از مجموعه‌ی

$$A - B = \{x : x \in A \text{ و } x \notin B\}$$

اعمال مجموعه‌ها از قوانین زیر پیروی می‌کنند.

---

[۱] بعضی نویسنده‌ها اعداد طبیعی را به جای ۰ با ۱ آغاز می‌کنند. به نظر می‌آید که مسیر مدرن به سمت آغاز این مجموعه با ۰ پیش می‌رود.

قوانین مجموعه‌های تهی:

$$A \cap \varnothing = \varnothing,$$
$$A \cup \varnothing = A$$

قوانین همانی:

$$A \cap A = A,$$
$$A \cup A = A$$

قوانین جابه‌جایی پذیری:

$$A \cap B = B \cap A,$$
$$A \cup B = B \cup A$$

قوانین شرکت‌پذیری:

$$A \cap (B \cap C) = (A \cap B) \cap C,$$
$$A \cup (B \cup C) = (A \cup B) \cup C$$

قوانین توزیع‌پذیری:

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$
$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$

قوانین جذب:

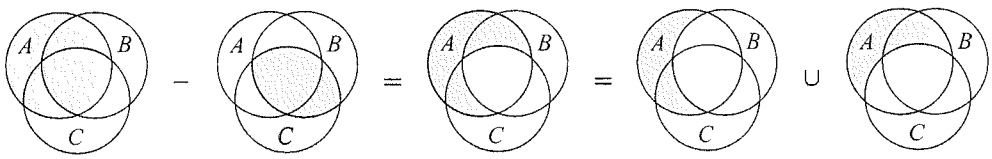
$$A \cap (A \cup B) = A,$$
$$A \cup (A \cap B) = A$$ (ب-۱)

قوانین دمورگان:

$$A - (B \cap C) = (A - B) \cup (A - C)$$
$$A - (B \cup C) = (A - B) \cap (A - C)$$ (ب-۲)

اولین قانون از قوانین دمورگان در شکل ب-۱ با استفاده از یک *نمودار ون* (Venn diagram) نشان داده شده است، که یک نمایش گرافیکی است که در آن مجموعه‌ها به صورت ناحیه‌هایی در صفحه نشان داده می‌شوند.

معمولاً تمام مجموعه‌های مورد نظر، زیرمجموعه‌ی یک مجموعه‌ی بزرگ‌تر U با نام *مجموعه‌ی جهانی* (universe) هستند. به عنوان مثال، اگر با مجموعه‌های مختلفی سروکار داشته باشیم که همگی فقط از اعداد صحیح ساخته شده باشند، آن گاه مجموعه‌ی $\mathbb{Z}$ از اعداد صحیح می‌تواند یک مجموعه‌ی جهانی مناسب باشد. با داشتن یک مجموعه‌ی جهانی U، *مکمل* (complement) یک مجموعه‌ی A را به صورت $\overline{A} = U - A = \{x : x \in U \text{ و } x \notin A\}$ تعریف می‌کنیم. برای هر مجموعه‌ی $A \subseteq U$، قوانین زیر را داریم:



شکل ب-۱ یک نمودار ون که اولین قانون دمورگان (ب-۲) را نشان می‌دهد. هر یک از مجموعه‌های A، B، و C به صورت یک دایره نمایش داده شده‌اند.

$$\overline{\overline{A}} = A,$$
$$A \cap \overline{A} = \emptyset,$$
$$A \cup \overline{A} = U$$

قوانین دمورگان (ب-۲) را می‌توان به کمک مکمل‌ها بازنویسی کرد. برای هر دو مجموعه‌ی $B, C \subseteq U$ داریم

$$\overline{B \cap C} = \overline{B} \cup \overline{C}$$
$$\overline{B \cup C} = \overline{B} \cap \overline{C}$$

دو مجموعه‌ی A و B *ناسازگار* (disjoint) هستند اگر هیچ عنصر مشترکی نداشته باشند، یعنی اگر $A \cap B = \emptyset$. یک گروه $\mathcal{Z} = \{S_i\}$ از مجموعه‌های ناتهی یک *افراز* (partition) از یک مجموعه‌ی S را تشکیل می‌دهند اگر

- مجموعه‌ها دوبه‌دو ناسازگار باشند، یعنی $S_i, S_j \in \mathcal{Z}$ و $i \neq j$ نتیجه دهد $S_i \cap S_j = \emptyset$، و
- اجتماع آن‌ها برابر S باشد، یعنی

$$S = \bigcup_{S_i \in \mathcal{Z}} S_i$$

به عبارت دیگر $\mathcal{Z}$ یک افراز از S را تشکیل می‌دهد اگر هر عنصر S دقیقاً در یک $S_i \in \mathcal{Z}$ ظاهر شود.

به تعداد عناصر در یک مجموعه، *کاردینالیتی* (cardinality) (یا *اندازه*) آن مجموعه گفته، و آن را با $|S|$ نشان می‌دهیم. دو مجموعه، کاردینالیتی‌های برابر دارند اگر بتوانیم یک تناظر یک به یک بین آن دو ایجاد کنیم. کاردینالیتی مجموعه‌ی تهی $|\emptyset| = 0$ است. اگر کاردینالیتی یک مجموعه، یک عدد طبیعی باشد، می‌گوییم مجموعه *متناهی* (finite) است؛ در غیر این صورت به آن مجموعه‌ی *نامتناهی* (infinite) می‌گوییم. اگر بتوان یک تناظر یک به یک بین یک مجموعه‌ی نامتناهی و مجموعه‌ی اعداد طبیعی $\mathbb{N}$ برقرار کرد، این مجموعه *نامتناهی شمارا* (countably infinite) است؛ در غیر این صورت *ناشمارا* (uncountable) است. مجموعه‌ی اعداد صحیح $\mathbb{Z}$ شمارا است، ولی اعداد حقیقی $\mathbb{R}$ ناشمارا هستند.

برای هر دو مجموعه‌ی متناهی A و B اتحاد زیر را داریم:

$$|A \cup B| = |A| + |B| - |A \cap B| \qquad (ب-۳)$$

که از آن می‌توانیم نتیجه بگیریم

$$|A \cup B| \le |A| + |B|$$

اگر A و B ناسازگار باشند، آن گاه $|A \cap B| = ۰$، و بنابراین $|A \cup B| = |A| + |B|$. اگر $A \subseteq B$، آن گاه $|A| \le |B|$.

بعضی مواقع به یک مجموعه‌ی متناهی با n عنصر، یک n-*مجموعه* گفته می‌شود. به یک ۱-مجموعه، یک مجموعه‌ی *یگانه* (singleton) گفته می‌شود. بعضی مواقع به یک زیرمجموعه‌ی k عنصری از یک مجموعه، یک k-*زیرمجموعه* گفته می‌شود.

مجموعه‌ی تمام زیرمجموعه‌های S، شامل مجموعه‌ی تهی و خود مجموعه‌ی S را با ۲^S نشان داده و آن را *مجموعه‌ی توانی* (power set) S می‌نامیم. برای مثال، $۲^{\{a,b\}} = \{\varnothing, \{a\}, \{b\}, \{a,b\}\}$. کاردینالیتی مجموعه‌ی توانی یک مجموعه‌ی متناهی S برابر است با $۲^{|S|}$ (تمرین ب-۱-۵ را ببینید).

بعضی مواقع به ساختارهای شبه مجموعه‌ای نیاز داریم که در آن‌ها عناصر ترتیب دارند. یک *جفت مرتب* (ordered pair) از دو عنصر a و b با (a,b) نشان داده می‌شود، و به صورت رسمی می‌توان آن را به صورت مجموعه‌ی $(a,b) = \{a, \{a,b\}\}$ تعریف کرد. بنابراین جفت مرتب (a,b) با جفت مرتب (b,a) معادل نیست.

ضرب کارتزین (Cartesian product) (یا ضرب دکارتی) دو مجموعه‌ی A و B، که آن را به صورت $A \times B$ نشان می‌دهیم، مجموعه‌ی تمام جفت‌های مرتبی است که عنصر اول آن‌ها از مجموعه‌ی A و عنصر دوم آن‌ها از مجموعه‌ی B است. به صورت رسمی‌تر،

$$A \times B = \{(a,b) : a \in A \text{ و } b \in B\}$$

به عنوان مثال،

$$\{a,b\} \times \{a,b,c\} = \{(a,a),(a,b),(a,c),(b,a),(b,b),(b,c)\}$$

وقتی A و B مجموعه‌های متناهی باشند، کاردینالیتی ضرب کارتزین آن‌ها برابر است با

$$|A \times B| = |A| \cdot |B| \qquad (ب-۴)$$

ضرب کارتزین n مجموعه‌ی $A_1, A_2, \ldots, A_n$ عبارت است از مجموعه‌ی *n تایی‌های*

$$A_1 \times A_2 \times \ldots \times A_n = \{(a_1, a_2, \ldots, a_n) : a_i \in A_i, i = ۱, ۲, \ldots, n\}$$

با کاردینالیتی

$$|A_1 \times A_2 \times \ldots \times A_n| = |A_1| \cdot |A_2| \ldots |A_n|$$

در صورتی که تمام مجموعه‌ها متناهی باشند. ضرب کارتزین nتایی تودرتو از یک مجموعه‌ی A را به

صورت زیر نشان می‌دهیم:

$$A^n = A \times A \times \ldots \times A$$

با کاردینالیتی $|A^n| = |A|^n$ اگر A متناهی باشد. یک nتایی را می‌توان به صورت یک دنباله‌ی متناهی با طول n هم دید (بخش ب-۳ را ببینید).

تمرین‌ها

ب-۱-۱ نمودار ون مربوط به اولین قانون توزیع‌پذیری (ب-۱) را بکشید.

ب-۱-۲ حالت کلی قوانین دمورگان را برای هر گروه متناهی از مجموعه‌ها اثبات کنید:

$$\overline{A_1 \cap A_2 \cap \ldots \cap A_n} = \overline{A_1} \cup \overline{A_2} \cup \ldots \cup \overline{A_n},$$
$$\overline{A_1 \cup A_2 \cup \ldots \cup A_n} = \overline{A_1} \cap \overline{A_2} \cap \ldots \cap \overline{A_n}$$

★ب-۱-۳ حالت کلی تساوی (ب-۳) را اثبات کنید، که به *اصل شمول و عدم شمول* (principle of inclusion and exclusion) معروف است:

$$\begin{aligned}
&|A_1 \cup A_2 \cup \ldots \cup A_n| = \\
&\quad |A_1| + |A_2| + \ldots + |A_n| \\
&\quad -|A_1 \cap A_2| - |A_1 \cap A_3| - \ldots \quad \text{(تمام جفت‌ها)} \\
&\quad +|A_1 \cap A_2 \cap A_3| + \ldots \quad \text{(تمام سه‌تایی‌ها)} \\
&\qquad \vdots \\
&\quad +(-1)^n |A_1 \cap A_2 \cap \ldots \cap A_n|
\end{aligned}$$

ب-۱-۴ نشان دهید که مجموعه‌ی تمام اعداد فرد، شمارا است.

ب-۱-۵ نشان دهید که برای هر مجموعه‌ی متناهی S، مجموعه‌ی توانی 2^S، $2^{|S|}$ عضو دارد (یعنی، $2^{|S|}$ زیرمجموعه‌ی متفاوت از S وجود دارد).

ب-۱-۶ با گسترش تعریف نظریه‌ی مجموعه‌ای جفت مرتب، یک تعریف استقرایی برای یک nتایی بدهید.

ب-۲ رابطه‌ها

یک *رابطه‌ی دودویی* (binary relation) R بر روی دو مجموعه‌ی A و B، یک زیرمجموعه از ضرب کارتزین $A \times B$ است. اگر $(a,b) \in R$، بعضی مواقع می‌نویسیم $a R b$. وقتی می‌گوییم R یک رابطه‌ی دودویی بر روی A است، این بدین معنی است که R یک زیرمجموعه از $A \times A$ است. مثلاً رابطه‌ی

«کوچک‌تر از» بر روی اعداد طبیعی عبارت است از مجموعه‌ی

$$\{(a,b): a,b \in \mathbb{N} \text{ و } a<b\}$$

یک رابطه‌ی nتایی بر روی مجموعه‌های A_1، A_2،...، A_n یک زیرمجموعه از $A_1 \times A_2 \times ... \times A_n$ است.

یک رابطه‌ی دودویی $R \subseteq A \times A$، *انعکاسی* (reflexive) است اگر

$$aRa$$

برای هر $a \in A$. به عنوان مثال "$=$" و "$\leq$" رابطه‌های انعکاسی بر روی $\mathbb{N}$ هستند، ولی "$<$" چنین نیست. رابطه‌ی R *متقارن* (symmetric) است اگر

$$aRb \Rightarrow bRa$$

برای تمام $a,b \in A$. برای مثال "$=$" متقارن است، ولی "$<$" و "$\leq$" این گونه نیستند. رابطه‌ی R *تراگذار* (transitive) است اگر

$$aRb \text{ و } bRc \Rightarrow aRc$$

برای هر $a,b,c \in A$. مثلاً رابطه‌های "$<$"، "$\leq$"، و "$=$" تراگذار هستند، ولی رابطه‌ی

$$R = \{(a,b): a,b \in \mathbb{N} \text{ و } a = b-1\}$$

این گونه نیست، چرا که $3R4$ و $4R5$ نتیجه نمی‌دهد $3R5$.

یک رابطه که انعکاسی، متقارن، و تراگذار باشد، یک *رابطه‌ی هم‌ارزی* (equivalence relation) نام دارد. برای مثال "$=$" یک رابطه‌ی هم‌ارزی بر روی اعداد طبیعی است، ولی "$<$" این گونه نیست. اگر R یک رابطه‌ی هم‌ارزی بر روی یک مجموعه‌ی A باشد، آن گاه برای $a \in A$، *کلاس هم‌ارزی* a عبارت است از مجموعه‌ی $[a] = \{b \in A : aRb\}$، یعنی مجموعه‌ی تمام عناصر معادل با a. مثلاً اگر تعریف کنیم

$$R = \{(a,b): a,b \in \mathbb{N} \text{ و } a+b \text{ یک عدد زوج است}\}$$

آن گاه R یک رابطه‌ی هم‌ارزی است، چرا که $a+a$ زوج است (انعکاسی)، زوج بودن $a+b$ نتیجه می‌دهد که $a+b$ هم زوج است (متقارن)، و زوج بودن $a+b$ و $b+c$ نتیجه می‌دهد که $a+c$ هم زوج است (تراگذار). کلاس هم‌ارزی ۴ عبارت است از $[4] = \{0,2,4,6,...\}$، و کلاس هم‌ارزی ۳ عبارت است از $[3] = \{1,3,5,7,...\}$. یک قضیه‌ی اساسی در مورد کلاس‌های هم‌ارزی، قضیه‌ی زیر است.

قضیه‌ی ب-۱ (یک رابطه‌ی هم‌ارزی معادل یک افراز است)

کلاس‌های هم‌ارزی هر رابطه‌ی هم‌ارزی R بر روی یک مجموعه‌ی A یک افراز بر روی A را شکل می‌دهند، و هر افراز بر روی A، یک رابطه‌ی هم‌ارزی بر روی آن تعیین می‌کنند، که در آن مجموعه‌های حاصل از افراز، همان کلاس‌های هم‌ارزی هستند.

اثبات برای بخش اول اثبات، باید نشان دهیم که کلاس‌های هم‌ارزی R، مجموعه‌های ناتهی و دوبه‌دو ناسازگار هستند که اجتماع آن‌ها برابر است با A. از آن جایی که R انعکاسی است، $a \in [a]$، و کلاس‌های هم‌ارزی ناتهی هستند؛ به علاوه چون هر عنصر $a \in A$ به کلاس هم‌ارزی $[a]$ تعلق دارد، اجتماع کلاس‌های هم‌ارزی برابر A است. این باقی می‌ماند که نشان دهیم کلاس‌های هم‌ارزی، دوبه‌دو ناسازگار هستند، یعنی اگر دو کلاس هم‌ارزی $[a]$ و $[b]$ یک عنصر مشترک مانند c داشته باشند، باید در واقع یک مجموعه باشند. اکنون $a R c$ و $b R c$ که طبق خواص تقارن و تراگذاری نتیجه می‌دهد $a R b$. بنابراین برای هر عنصر دلخواه $x \in [a]$، داریم $x R a$ نتیجه می‌دهد $x R b$، و بنابراین $[b] \subseteq [a]$. به طور مشابه $[a] \subseteq [b]$، و بنابراین $[a] = [b]$.

برای بخش دوم اثبات، فرض کنید $A = A_i$ یک افراز بر روی A باشد، و تعریف می‌کنیم:

$$R = \{(a,b): \quad \text{یک } i \text{ وجود دارد به طوری که } a \in A_i \text{ و } b \in A_i\}$$

ادعا می‌کنیم که R یک رابطه‌ی هم‌ارزی روی A است. انعکاس برقرار است، چرا که $a \in A_i$ نتیجه می‌دهد $a R a$. تقارن برقرار است، چرا که اگر $a R b$، آن گاه a و b هر دو در یک مجموعه مانند A_i هستند، و بنابراین $b R a$. اگر $a R b$ و $b R c$، آن گاه هر سه عنصر در یک مجموعه هستند، و بنابراین $a R c$، و تراگذاری هم برقرار است. برای این که ببینیم مجموعه‌های افراز همان کلاس‌های هم‌ارزی R هستند، مشاهده کنید که اگر $a \in A_i$، آن گاه $x \in [a]$ نتیجه می‌دهد $x \in A_i$، و $x \in A_i$ نتیجه می‌دهد $x \in [a]$. ▒

یک رابطه‌ی دودویی R بر روی یک مجموعه‌ی A *ضد تقارن* (antisymmetric) است اگر $a R b$ و $b R a$ نتیجه دهد $a = b$.

برای مثال رابطه‌ی "$\leq$" بر روی اعداد طبیعی ضد تقارن است، چرا که $a \leq b$ و $b \leq a$ نتیجه می‌دهد $a = b$. یک رابطه که انعکاسی، ضد متقارن، و تراگذار است یک *ترتیب جزئی* (partial order) نام دارد، و به مجموعه‌ای که یک ترتیب جزئی بر روی آن تعریف شده است، یک *مجموعه‌ی مرتب جزئی* (partially ordered set) می‌گوییم. به عنوان مثال رابطه‌ی «نواده»، یک ترتیب جزئی بر روی انسان‌ها است (اگر فرض کنیم هر کس نواده‌ی خود هم هست).

در یک مجموعه‌ی مرتب جزئی، ممکن است یک عنصر «بیشینه‌ی» a وجود نداشته باشد که برای تمام $b \in A$ داشته باشیم $b R a$. در عوض ممکن است عناصر ماکسیمال متعددی مانند a وجود داشته باشد به طوری که برای هیچ $b \in A$، که $b \neq a$، نداشته باشیم $a R b$. برای مثال در یک گروه از جعبه‌های با اندازه‌های مختلف ممکن است جعبه‌های ماکسیمال متعددی وجود داشته باشند که درون هیچ جعبه‌ی دیگری جا نمی‌گیرند، ولی هیچ جعبه‌ی بیشینه‌ای هم وجود نداشته باشد که تمام جعبه‌های دیگر در آن جای گیرند.[1]

[1] اگر بخواهیم دقیق باشیم، برای این که «جای گرفتن در» یک ترتیب جزئی باشد، باید فرض کنیم که یک جعبه درون خود جای می‌گیرد.

یک رابطه‌ی R روی یک مجموعه‌ی A یک *رابطه‌ی تمام* (total relation) است اگر برای هر $a,b \in A$ داشته باشیم aRb یا bRa (یا هر دو)، یعنی اگر هر جفت از عناصر A توسط R با هم رابطه داشته باشند. یک ترتیب جزئی که یک رابطه‌ی تام هم باشد یک *ترتیب تام* (total order) یا *ترتیب خطی* (linear order) است. برای مثال رابطه‌ی "$\leq$" یک ترتیب تام بر روی اعداد طبیعی است، ولی «نواده» یک ترتیب تام بر روی مجموعه‌ی افراد نیست، چرا که افرادی وجود دارند که هیچ یک نواده‌ی دیگری نیستند. یک رابطه‌ی تام که تراگذار است، ولی لزوماً انعکاسی یا ضد متقارن نیست، یک *پیش‌ترتیب تام* (total preorder) نام دارد.

تمرین‌ها

ب-۲-۱ اثبات کنید که رابطه‌ی "$\subseteq$" (زیرمجموعه) بر روی تمام زیرمجموعه‌های $\mathbb{Z}$ یک ترتیب جزئی است، ولی رابطه‌ی تام نیست.

ب-۲-۲ نشان دهید که برای هر عدد صحیح مثبت n، رابطه‌ی «همنهشتی به پیمانه‌ی n» یک رابطه‌ی هم‌ارزی بر روی اعداد صحیح است. (می‌گوییم $a \equiv b \pmod{n}$ اگر یک عدد صحیح q وجود داشته باشد به طوری که $a-b=qn$.) این رابطه، اعداد صحیح را به چه کلاس‌های هم‌ارزی تقسیم می‌کند؟

ب-۲-۳ مثالی از رابطه‌هایی بزنید که خصوصیات زیر را داشته باشند.

I. انعکاسی و متقارن باشد، ولی تراگذار نباشد.

II. انعکاسی و تراگذار باشد، ولی متقارن نباشد.

III. متقارن و تراگذار باشد، ولی انعکاسی نباشد.

ب-۲-۴ فرض کنید S یک مجموعه‌ی متناهی باشد، و R یک رابطه‌ی هم‌ارزی بر روی $S \times S$. نشان دهید که اگر R ضد متقارن هم باشد، آن گاه کلاس‌های هم‌ارزی R بر روی S مجموعه‌های تک عضوی هستند.

ب-۲-۵ پروفسور Narcissus ادعا می‌کند که اگر یک رابطه‌ی R متقارن و تراگذار باشد، آن گاه انعکاسی هم هست. او اثبات زیر را ارائه می‌کند. طبق تقارن، aRb نتیجه می‌دهد bRa. بنابراین تراگذاری ایجاب می‌کند که aRa. آیا پروفسور درست می‌گوید؟

ب-۳ توابع

با داشتن دو مجموعه‌ی A و B، یک *تابع* (function) f یک رابطه‌ی دودویی بر روی $A \times B$ است به طوری که برای تمام $a \in A$ دقیقاً یک $b \in B$ وجود داشته باشد به طوری که $(a,b) \in f$. مجموعه‌ی A *دامنه‌ی* (domain) f می‌گوییم، و مجموعه‌ی B *هم‌دامنه‌ی* (codomain) f نام دارد. بعضی مواقع

می‌نویسیم $f: A \to B$؛ و اگر $(a,b) \in f$ می‌نویسیم $b = f(a)$، چرا که b به صورت یکتا از روی انتخاب a به دست می‌آید.

به صورت شهودی، تابع f یک عنصر از B به هر یک از عناصر A نسبت می‌دهد. به هیچ عنصری از A دو عنصر مختلف از B نسبت داده نمی‌شود، ولی ممکن است یک عنصر از B به دو عنصر از A نسبت داده شود. برای مثال رابطه‌ی دودویی

$$f = \{(a,b) : a,b \in \mathbb{N} \text{ و } b = a \bmod ۲\}$$

یک تابع $f: \mathbb{N} \to \{۰,۱\}$ است، چرا که برای هر عدد طبیعی a، دقیقاً یک مقدار b در $\{۰,۱\}$ وجود دارد به طوری که $b = a \bmod ۲$. برای این مثال، $f(۰) = ۰$، $f(۱) = ۱$، $f(۲) = ۰$، و الی آخر. در مقابل رابطه‌ی دودویی

$$g = \{(a,b) : a,b \in \mathbb{N} \text{ و } a+b \text{ زوج است}\}$$

یک تابع نیست، چرا که (۱,۳) و (۱,۵) هر دو در g هستند، و بنابراین برای انتخاب $a = ۱$، دقیقاً یک b وجود ندارد به طوری که $(a,b) \in g$.

با داشتن تابع $f: A \to B$، اگر $b = f(a)$، می‌گوییم a *آرگومان* (argument) f است، و b *مقدار* (value) f در a. می‌توانیم یک تابع را با تعیین مقدار تمام عناصر آن در دامنه تعریف کنیم. برای مثال ممکن است تعریف کنیم $f(n) = ۲n$ برای $n \in N$، که بدین معنی است که $f = \{(n,۲n) : n \in N\}$. دو تابع f و g *برابر* (equal) هستند اگر دامنه و هم‌دامنه‌ی آن‌ها برابر باشد، و برای هر a در دامنه داشته باشیم $f(a) = g(a)$.

یک *دنباله‌ی متناهی* (finite sequence) به طول n، یک تابع f است که دامنه‌ی آن مجموعه‌ی اعداد صحیح $\{۰,۱,...,n-۱\}$ است. یک دنباله‌ی متناهی را با لیست کردن مقادیر آن نشان می‌دهیم: $\langle f(۰), f(۱), ..., f(n-۱) \rangle$. یک *دنباله‌ی نامتناهی* (infinite sequence) یک تابع است که دامنه‌ی آن مجموعه‌ی $\mathbb{N}$ از اعداد طبیعی است. برای مثال دنباله‌ی فیبوناچی، که طبق بازگشت (۳-۲۱) تعریف می‌شود، یک دنباله‌ی نامتناهی است به شکل $\langle ۰,۱,۱,۲,۳,۵,۸,۱۳,۲۱,... \rangle$.

وقتی دامنه‌ی یک تابع f، یک ضرب کارتزین است، معمولاً از پرانتزهای اضافی دو طرف آرگومان f صرف‌نظر می‌کنیم. برای مثال اگر تابع $f: A_۱ \times A_۲ \times ... \times A_n \to B$ را داشته باشیم، می‌نویسیم $b = f(a_۱, a_۲, ..., a_n)$ به جای این که بنویسیم $b = f((a_۱, a_۲, ..., a_n))$. همچنین به هر a_i، یک آرگومان تابع f می‌گوییم، با این که از نظر فنی (تنها) آرگومان f، n تایی $(a_۱, a_۲, ..., a_n)$ است.

اگر $f: A \to B$ یک تابع باشد و $b = f(a)$، بعضی مواقع می‌گوییم b *تصویر* (image) a تحت f است. تصویر مجموعه‌ی $A' \subseteq A$ تحت f به صورت

$$f(A') = \{b \in B : a \in A' \text{ برای یک } b = f(a)\}$$

تعریف می‌شود. *برد* (range) f عبارت است از تصویر دامنه‌ی آن، یعنی $f(A)$. برای مثال دامنه‌ی تابع $f: N \to N$ که $f(n) = ۲n$ برابر است با

$$f(\mathbb{N}) = \{m : \text{برای یک } n \in \mathbb{N},\ m = ۲n\}$$

که عبارت است از مجموعه‌ی اعداد زوج نامنفی.

یک تابع *پوشا* (surjection) است اگر برد آن با هم‌دامنه برابر باشد. برای مثال تابع $f(n) = \lfloor n/۲ \rfloor$ یک تابع پوشا از $\mathbb{N}$ به $\mathbb{N}$ است، چرا که هر عنصر در $\mathbb{N}$ به عنوان مقدار f برای یک آرگومان ظاهر می‌شود. در مقابل تابع $f(n) = ۲n$ یک تابع پوشا از $\mathbb{N}$ به $\mathbb{N}$ نیست، چرا که هیچ آرگومانی برای f نمی‌تواند مقدار ۳ را تولید کند. با این حال، تابع $f(n) = ۲n$ یک تابع پوشا از مجموعه‌ی اعداد طبیعی به مجموعه‌ی اعداد زوج است. یک تابع پوشای $f : A \to B$ بعضی مواقع یک نگاشت A *روی* B (A onto B) نامیده می‌شود.

یک تابع $f : A \to B$ *یک به یک* (one-to-one) است اگر آرگومان‌های متمایز f مقادیر متمایزی تولید کنند، یعنی اگر $a \neq a'$ ایجاب کند $f(a) \neq f(a')$. برای مثال تابع $f(n) = ۲n$ یک تابع یک به یک از $\mathbb{N}$ به $\mathbb{N}$ است، چرا که هر عدد زوج b تحت f حداکثر تصویر یک عنصر از دامنه است، یعنی $b/۲$. تابع $f(n) = \lfloor n/۲ \rfloor$ یک به یک نیست، چرا که مقدار ۱ توسط دو آرگومان ۲ و ۳ ساخته می‌شود.

یک تابع $f : A \to B$، *یک به یک-پوشا* (bijection) است اگر هم یک به یک باشد و هم پوشا. برای مثال تابع $f(n) = (-۱)^n \lceil n/۲ \rceil$ یک تابع یک به یک-پوشا از $\mathbb{N}$ به $\mathbb{Z}$ است:

$$\begin{aligned}
۰ &\to ۰,\\
۱ &\to -۱,\\
۲ &\to ۱,\\
۳ &\to -۲,\\
۴ &\to ۲,\\
&\vdots
\end{aligned}$$

این تابع یک به یک است چرا که هیچ عنصری از $\mathbb{Z}$ تصویر بیش از یک عنصر از $\mathbb{N}$ نیست، و پوشا است چون تمام عناصر $\mathbb{Z}$ به صورت تصویر یک عنصر از $\mathbb{N}$ ظاهر می‌شوند. بنابراین تابع یک به یک-پوشا است. بعضی مواقع به یک تابع یک به یک-پوشا، یک *تناظر یک به یک* (one-to-one correspondence) هم گفته می‌شود، چرا که تمام عناصر دامنه و هم‌دامنه را با یک دیگر جفت می‌کند. به یک تابع یک به یک-پوشا از یک مجموعه‌ی A به همان مجموعه یک *جایگشت* (permutation) گفته می‌شود.

اگر یک تابع f یک به یک-پوشا باشد، *معکوس* (inverse) آن به صورت

$$f^{-۱}(b) = a \quad \text{اگر و فقط اگر} \quad f(a) = b$$

تعریف می‌شود. برای مثال معکوس تابع $f(n) = (-۱)^n \lceil n/۲ \rceil$ عبارت است از

$$f^{-۱}(m) = \begin{cases} ۲m & \text{اگر } m \geq ۰ \\ -۲m - ۱ & \text{اگر } m < ۰ \end{cases}$$

تمرین‌ها

ب-۳-۱ فرض کنید A و B مجموعه‌های متناهی باشند، و $f: A \to B$ یک تابع باشد. نشان دهید که

i. اگر f یک به یک باشد، آن گاه $|A| \le |B|$؛

ii. اگر f پوشا باشد، آن گاه $|A| \ge |B|$.

ب-۳-۲ اگر دامنه و هم‌دامنه $\mathbb{N}$ باشند، آیا $f(x) = x + 1$ یک به یک-پوشا است؟ اگر دامنه و هم‌دامنه $\mathbb{Z}$ باشند چطور؟

ب-۳-۳ یک تعریف طبیعی برای معکوس یک رابطه‌ی دودویی ارائه دهید به طوری که اگر یک رابطه در واقع یک تابع یک به یک-پوشا باشد، معکوس رابطه‌ای آن با معکوس تابعی آن یکی باشد.

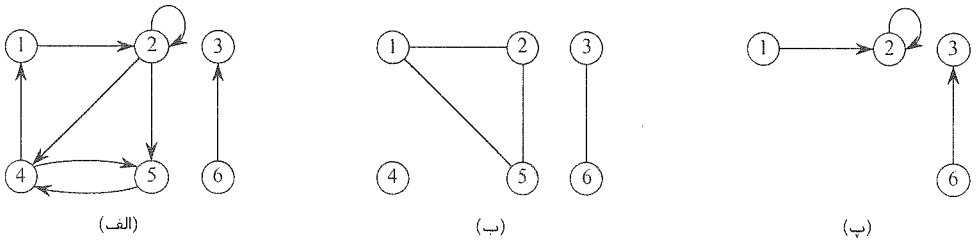
★ب-۳-۴ یک تابع یک به یک-پوشا از $\mathbb{Z}$ به $\mathbb{Z} \times \mathbb{Z}$ ارائه کنید.

ب-۴ گراف‌ها

در این بخش دو نوع گراف معرفی می‌شود: جهت‌دار و بدون جهت. تعریف‌های خاص در ادبیات علوم کامپیوتر با این تعاریف متفاوت‌اند، ولی در اکثر مواقع این تفاوت‌ها مختصر است. بخش ۲۲-۱ نشان می‌دهد که چگونه می‌توان گراف‌ها را در حافظه‌ی کامپیوتر ذخیره کرد.

یک *گراف جهت‌دار* (directed graph) G یک جفت (V, E) است، که در آن V یک مجموعه‌ی متناهی است و E یک رابطه بر روی V. به مجموعه‌ی V *مجموعه‌ی رأس‌های* G گفته می‌شود، و عناصر آن *رأس* (vertex) نام دارند. به مجموعه‌ی E مجموعه‌ی یال‌های G گفته می‌شود، و عناصر آن *یال* (edge) نام دارند. شکل ب-۲ (الف) یک نمایش عملی از یک گراف جهت‌دار بر روی مجموعه رأس‌های $\{1,2,3,4,5,6\}$ است. رأس‌ها به صورت دایره نشان داده شده‌اند، و یال‌ها به صورت فلش. توجه کنید که *طوقه* (self-loop) – یال‌هایی که از یک رأس شروع شده و در همان رأس پایان می‌یابند – هم در این گراف وجود دارد.

در یک *گراف بدون جهت* (undirected graph) $G = (V, E)$، مجموعه یال‌های G حاوی جفت‌های *نامرتبی* است از رأس‌ها، نه جفت‌های مرتب. یعنی، یک یال یک مجموعه‌ی $\{u, v\}$ است که در آن $u, v \in V$ و $u \ne v$. برای سادگی از نماد (u, v) به جای نماد مجموعه‌ای $\{u, v\}$ برای یال‌ها استفاده می‌کنیم، و (u, v) و (v, u) یکسان در نظر گرفته می‌شوند. در یک گراف بدون جهت، طوقه‌ها غیر مجاز هستند، و بنابراین هر یال از دو رأس متمایز تشکیل می‌شود. شکل ب-۲ (ب) یک نمایش عملی از یک گراف بدون جهت است، با مجموعه رأس‌های $\{1,2,3,4,5,6\}$.



شکل ب-۲ گراف‌های جهت‌دار و بدون جهت. (الف) یک گراف جهت‌دار $G=(V,E)$، که در آن $V=\{1,2,3,4,5,6\}$ و $E=\{(1,2),(2,2),(2,4),(2,5),(4,1),(4,5),(5,4),(6,3)\}$. یال (۲,۲) یک طوقه است. (ب) یک گراف بدون جهت $G=(V,E)$ که در آن $V=\{1,2,3,4,5,6\}$ و $E=\{(1,2),(1,5),(2,5),(3,6)\}$. رأس ۴ یک رأس تنها است. (پ) یک زیرگراف از گراف بخش (الف) که مجموعه رأس‌های آن $\{1,2,3,6\}$ است.

بسیاری از تعریف‌ها برای گراف‌های جهت‌دار و بدون جهت یکی هستند، هر چند بعضی عبارت‌های خاص در این دو مفهوم معنی‌های متفاوت دارند. اگر (u,v) یک یال در یک گراف جهت‌دار $G=(V,E)$ باشد، می‌گوییم (u,v) *مجاور* (incident) با رأس u است، یا رأس u را *ترک* (leave) می‌کند، و مجاور با رأس v است، و یا به آن *وارد* (enter) می‌شود. برای مثال یال‌های خروجی از رأس ۲ در شکل ب-۲ (الف) عبارتند از (۲,۲)، (۲,۴)، و (۲,۵). یال‌های ورودی به رأس۲ عبارتند از (۱,۲) و (۲,۲). اگر (u,v) یک یال در یک گراف بدون جهت $G=(V,E)$ باشد، می‌گوییم (u,v) با رأس‌های u و v مجاور است. در شکل ب-۲ (ب) یال‌های مجاور با رأس ۲ عبارتند از (۱,۲) و (۲,۵). اگر (u,v) یک یال در گراف $G=(V,E)$ باشد، می‌گوییم رأس v مجاور رأس u است. اگر گراف بدون جهت باشد، رابطه‌ی مجاورت متقارن است، ولی اگر گراف جهت‌دار باشد، این رابطه لزوماً متقارن نیست. اگر v با u در یک گراف جهت‌دار مجاور باشد، بعضی مواقع می‌نویسیم $u \rightarrow v$. در بخش‌های (الف) و (ب) در شکل ب-۲، رأس ۲ مجاور رأس ۱ است، چرا که یال (۱,۲) متعلق به گراف هر دو شکل است. رأس ۱ در شکل ب-۲ (الف) مجاور رأس ۲ نیست، چرا که رأس (۲,۱) در این گراف نیست.

درجه‌ی (degree) یک رأس در یک گراف بدون جهت برابر است با تعداد یال‌های مجاور با آن. برای مثال رأس ۲ در شکل ب-۲ (ب) دارای درجه‌ی ۲ است. یک رأس با درجه‌ی ۰، مانند رأس ۴ در شکل ب-۲ (ب)، یک رأس *تنها* (isolated) است. در یک گراف جهت‌دار، *درجه‌ی خروجی* (out-degree) یک رأس برابر است با تعداد یال‌های خروجی آن، و *درجه‌ی ورودی* (in-degree) آن برابر است با تعداد یال‌های ورودی به آن. *درجه‌ی* یک رأس در یک گراف جهت‌دار برابر است با درجه‌ی ورودی آن به علاوه‌ی درجه‌ی خروجی آن. رأس ۲ در شکل ب-۲ (ب) دارای درجه‌ی ورودی ۲، درجه‌ی خروجی ۳، و درجه‌ی ۵ است.

یک *مسیر* (path) با *طول* (length) k از یک رأس u به یک رأس u' در یک گراف $G=(V,E)$ عبارت است از یک دنباله‌ی $\langle v_0,v_1,v_2,...,v_k\rangle$ از رأس‌ها، به طوری که $u=v_0$، $u'=v_k$، و $(v_{i-1},v_i)\in E$ برای $i=1,2,...,k$. طول یک مسیر برابر است با تعداد یال‌های آن مسیر. مسیر *شامل* رأس‌های v_0، v_1، ...، v_k و یال‌های (v_0,v_1)، (v_1,v_2)، ...، (v_{k-1},v_k) می‌شود. (همیشه یک مسیر با طول ۰ از u به u وجود دارد.) اگر یک مسیر p از u به u' وجود داشته باشد، می‌گوییم u' از u قابل *دسترس* (reachable) است، از طریق مسیر p، که اگر G جهت‌دار باشد، بعضی مواقع آن را به صورت $u \overset{p}{\rightsquigarrow} u'$ می‌نویسیم. یک مسیر، *ساده* (simple) است اگر تمام رأس‌های آن مجزا باشند. در شکل ب-۲ (الف) مسیر $\langle 1,2,5,4\rangle$ یک مسیر ساده با طول ۳ است. مسیر $\langle 2,5,4,5\rangle$ ساده نیست.

یک *زیرمسیر* (subpath) از مسیر $p=\langle v_0,v_1,...,v_k\rangle$ یک زیردنباله‌ی پیوسته از رأس‌های آن است. یعنی برای $0\le i\le j\le k$، زیردنباله‌ی متشکل از رأس‌های $\langle v_i,v_{i+1},...,v_j\rangle$ یک زیرمسیر از p است.

در یک گراف جهت‌دار، یک مسیر $\langle v_0,v_1,...,v_k\rangle$ یک *دور* (cycle) را تشکیل می‌دهد اگر $v_0=v_k$ و مسیر حداقل حاوی یک یال باشد. یک دور، *ساده* (simple) است اگر رأس‌های $v_1,v_2,...,v_k$ متمایز باشند. یک طوقه، یک دور با طول ۱ است. دو مسیر $\langle v_0,v_1,v_2,...,v_{k-1},v_0\rangle$ و $\langle v'_0,v'_1,v'_2,...,v'_{k-1},v'_0\rangle$ دورهای یکسانی هستند اگر یک عدد صحیح j وجود داشته باشد به طوری که $v'_i=v_{(i+j) \bmod k}$ برای $i=0,1,...k-1$. در شکل ب-۲ (الف) مسیر $\langle 1,2,4,1\rangle$ همان دوری را تشکیل می‌دهد که مسیرهای $\langle 2,4,1,2\rangle$ و $\langle 4,1,2,4\rangle$ تشکیل می‌دهند. این دور ساده است، ولی دور $\langle 1,2,4,5,4,1\rangle$ این گونه نیست. دور $\langle 2,2\rangle$ که توسط یال $(2,2)$ به وجود می‌آید، یک طوقه است. یک گراف جهت‌دار بدون طوقه، یک گراف *ساده* (simple) است. در یک گراف بدون جهت، یک مسیر $\langle v_0,v_1,...,v_k\rangle$ یک *دور* را تشکیل می‌دهد اگر $k\ge 3$ و $v_0=v_k$؛ دور *ساده* است اگر رأس‌های $v_1,v_2,...,v_k$ متمایز باشند. برای مثال در شکل ب-۲ (ب)، مسیر $\langle 1,2,5,1\rangle$ یک دور است. گرافی که دور نداشته باشد، یک گراف *بدون دور* (acyclic) نام دارد.

یک گراف بدون جهت، *همبند* (connected) است اگر بین هر جفت از رأس‌ها یک مسیر وجود داشته باشد. *مؤلفه‌های همبندی* (connected component) یک گراف، کلاس‌های هم‌ارزی از رأس‌ها هستند تحت رابطه‌ی «قابل دسترس بودن». گراف شکل ب-۲ (ب) سه مؤلفه‌ی همبندی دارد: $\{1,2,5\}$، $\{3,6\}$، و $\{4\}$. هر رأس در $\{1,2,5\}$ از هر رأس دیگر در $\{1,2,5\}$ قابل دسترس است. یک گراف بدون جهت، همبند است اگر دقیقاً یک مؤلفه‌ی همبندی داشته باشد. یال‌های مؤلفه‌ی همبندی آن‌هایی هستند که فقط با رأس‌های همان مؤلفه مجاورند؛ به عبارت دیگر (u,v) یک یال مؤلفه‌ی همبندی است تنها اگر هر دوی u و v رأس‌های مؤلفه باشند.

یک گراف جهت دار، *قویا همبند* (strongly connected) است اگر هر دو رأس آن از یکدیگر قابل دسترس باشند. *مؤلفه‌های قویا همبند* (strongly connected component) یک گراف جهت‌دار، کلاس‌های هم‌ارزی از رأس‌ها هستند تحت رابطه‌ی «دسترسی متقابل». یک گراف جهت‌دار، قویا همبند است اگر دقیقاً یک مؤلفه‌ی قویا همبند داشته باشد. گراف شکل ب-۲ (الف) سه مؤلفه‌ی قویا همبند

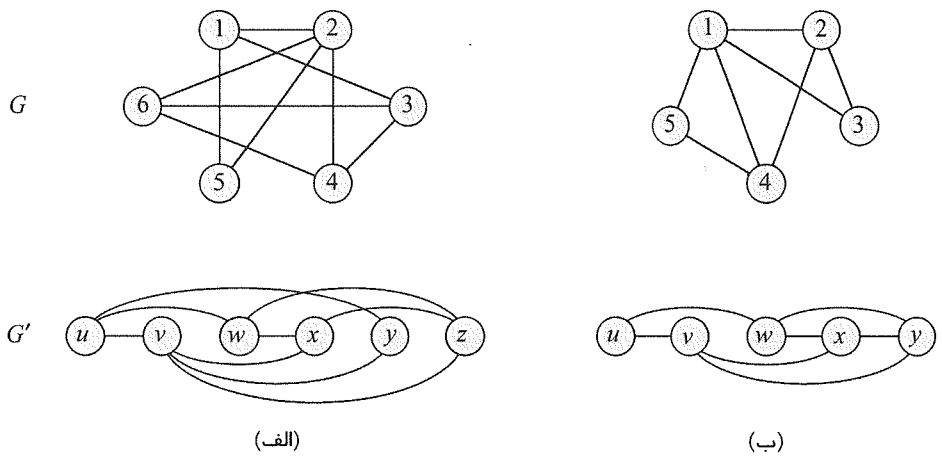
دارد: {۱,۲,۴,۵}، {۳}، و {۶}. هر جفتی از رأس‌ها در {۱,۲,۴,۵} متقابلا از یکدیگر قابل دسترس هستند. رأس‌های {۳,۶} یک مؤلفه‌ی قویا همبند تشکیل نمی‌دهند، چرا که رأس ۶ از رأس ۳ قابل دسترس نیست.

دو گراف $G=(V,E)$ و $G'=(V',E')$ هم‌ریخت هستند اگر یک تابع یک به یک-پوشای $f: V \to V'$ وجود داشته باشد به طوری که $(u,v)\in E$ اگر و فقط اگر $(f(u),f(v))\in E'$. به عبارت دیگر، می‌توانیم رأس‌های G را با نام رأس‌های G' برچسب‌گذاری کنیم، طوری که یال‌های متناظر در G و G' یکسان باشند. شکل ب-۳ (الف) یک جفت از گراف‌های هم‌ریخت G و G' را نسبت به مجموعه رأس‌های $\{1,2,3,4,5,6\}$ و $V'=\{u,v,w,x,y,z\}$ نشان می‌دهد.

نگاشت داده شده توسط $f(1)=u$، $f(2)=v$، $f(3)=w$، $f(4)=x$، $f(5)=y$، و $f(6)=z$ از V به V' همان تابع یک به یک-پوشای مورد نظر است. گراف‌های شکل ب-۳ (ب) هم‌ریخت نیستند. با این که هر دو گراف ۵ رأس و ۷ یال دارند، گراف بالایی یک رأس با درجه‌ی ۴ دارد، در حالی که گراف پایینی چنین رأسی ندارد.

می‌گوییم یک گراف $G'=(V',E')$ یک *زیرگراف* (subgraph) از گراف $G=(V,E)$ است اگر $V'\subseteq V$ و $E'\subseteq E$. با داشتن یک مجموعه‌ی $V'\subseteq V$، زیرگراف G *القا شده* (induced) توسط V' عبارت است از گراف $G'=(V',E')$ که در آن

$$E'=\{(u,v)\in E : u,v\in V'\}$$



شکل ب-۳ (الف) دو گراف هم‌ریخت. رأس‌های گراف بالایی توسط $f(1)=u$، $f(2)=v$، $f(3)=w$، $f(4)=x$، $f(5)=y$، و $f(6)=z$ به رأس‌های گراف پایینی نگاشت شده‌اند. (ب) این دو گراف، هم‌ریخت نیستند، چرا که گراف بالایی یک رأس با درجه‌ی ۴ دارد، در حالی که گراف پایین چنین رأسی ندارد.

زیرگراف القا شده توسط مجموعه رأس‌های {۱,۲,۳,۶} در شکل ب-۲ (الف)، در شکل ب-۲ (پ) نشان داده شده است، و دارای مجموعه یال‌های {(۱,۲),(۲,۲),(۶,۳)} است.

با داشتن یک گراف بدون جهت $G=(V,E)$، *نسخه‌ی جهت‌دار* (directed version) G، گراف جهت‌دار $G'=(V,E')$ است، که در آن $(u,v)\in E'$ اگر و فقط اگر $(u,v)\in E$. یعنی هر یال بدون جهت (u,v) در G در نسخه‌ی جهت‌دار با دو یال جهت‌دار (u,v) و (v,u) جایگزین می‌شود. با داشتن یک گراف جهت‌دار $G=(V,E)$، *نسخه‌ی بدون جهت* (undirected version) G، گراف بدون جهت $G'=(V,E')$ است، که در آن $(u,v)\in E'$ اگر و فقط اگر $u\neq v$ و $(u,v)\in E$. یعنی نسخه‌ی بدون جهت حاوی یال‌های G است که در آن «جهت یال‌ها» و طوقه‌ها حذف شده‌اند. (چون (u,v) و (v,u) در یک گراف بدون جهت یکی هستند، نسخه‌ی بدون جهت یک گراف جهت‌دار فقط یک بار حاوی این یال است، حتی اگر گراف جهت‌دار حاوی هر دو یال (u,v) و (v,u) باشد.) در یک گراف جهت‌دار $G=(V,E)$، یک *همسایه* (neighbor) از یک رأس u هر رأسی است که در نسخه‌ی بدون جهت، با u مجاور باشد. یک *گراف دوبخشی* (bipartite graph) یک گراف بدون جهت $G=(V,E)$ است که در آن می‌توان V را به دو مجموعه‌ی V_1 و V_2 تقسیم کرد، به طوری که $(u,v)\in E$ ایجاب کند $u\in V_1$ و $v\in V_2$ یا $u\in V_2$ و $v\in V_1$. یعنی تمام یال‌ها بین دو مجموعه‌ی V_1 و V_2 حرکت می‌کنند. یک گراف بدون جهت و بدون دور، یک *جنگل* (forest) است، و یک گراف بدون جهت و بدون دور همبند، یک *درخت (آزاد)* (free tree) (بخش ب-۵ را ببینید). معمولاً از اولین حروف «گراف جهت‌دار بدون دور» (directed acyclic graph) استفاده کرده و چنین گرافی را *dag* می‌نامیم.

دو نوع گراف وجود دارد که معمولاً به آن‌ها برخورد می‌کنیم. یک *گراف چندتایی* (multigraph) یک گراف بدون جهت است، ولی ممکن است طوقه و یا چندین یال بین دو رأس داشته باشد. یک *ابرگراف* (hypergraph) مانند یک گراف بدون جهت است، ولی هر *ابریال* (hyperedge)، به جای دو رأس می‌تواند تعداد دلخواهی از رأس‌ها را به هم متصل کند. بسیاری از الگوریتم‌هایی را که برای گراف‌های جهت‌دار و بدون جهت معمولی نوشته شده‌اند، می‌توان طوری اصلاح کرد که بر روی چنین ساختارهای گراف مانندی هم کار کنند.

کاهش (contraction) یک گراف بدون جهت $G=(V,E)$ القا شده توسط $e=(u,v)$ یک گراف $G'=(V',E')$ است، که در آن $V'=V-\{u,v\}\cup\{x\}$ و x یک رأس جدید است. مجموعه‌ی یال‌های E' از حذف یال (u,v) از E، و برای هر رأس w مجاور با u یا v، حذف هر کدام از (u,w) یا (v,w) موجود در E و اضافه کردن یال جدید (x,w) به وجود می‌آید. عملاً u و v به یک رأس «کاهش» یافته‌اند.

تمرین‌ها

ب-۴-۱ شرکت‌کننده‌ها در میهمانی دانشگاهی هنگام احوال‌پرسی با یکدیگر دست می‌دهند، و هر پروفسور تعداد دفعات دست دادن خود را به خاطر می‌سپارد. در پایان میهمانی، رئیس دانشکده تعداد دفعات دست دادن‌های هر پروفسور را با یکدیگر جمع می‌کند. با *اثبات لم دست دادن* (handshaking lemma)، نشان دهید که نتیجه زوج است: اگر $G = (V, E)$ یک گراف بدون جهت باشد، آن گاه

$$\sum_{v \in V} \deg ree(v) = ۲|E|$$

ب-۴-۲ نشان دهید که اگر یک گراف جهت‌دار یا بدون جهت حاوی یک مسیر بین دو رأس u و v باشد، آن گاه حاوی یک مسیر ساده بین این دو رأس هم هست. نشان دهید که اگر یک گراف جهت‌دار حاوی یک دور باشد، آن گاه حاوی یک دور ساده هم هست.

ب-۴-۳ نشان دهید که هر گراف بدون جهت و همبند $G = (V, E)$ رابطه‌ی $|E| \geq |V| - ۱$ را ارضا می‌کند.

ب-۴-۴ تحقیق کنید که در یک گراف بدون جهت، رابطه‌ی «قابل دسترس بودن» یک رابطه‌ی هم‌ارزی بر روی رأس‌های گراف است. در حالت کلی کدام یک از سه خصوصیت رابطه‌های هم‌ارزی برای رابطه‌ی «قابل دسترس بودن» برای رأس‌های یک گراف جهت‌دار برقرار هستند؟

ب-۴-۵ نسخه‌ی بدون جهت گراف جهت‌دار شکل ب-۲ (الف) چگونه است؟ نسخه‌ی جهت‌دار گراف بدون جهت شکل ب-۲ (ب) چگونه است؟

★ب-۴-۶ نشان دهید که یک ابرگراف را می‌توان به صورت یک گراف دو بخشی نشان داد اگر رابطه‌ی مجاور بودن در ابرگراف را با رابطه‌ی همسایگی در گراف دوبخشی نشان دهیم. (*راهنمایی:* فرض کنید یک مجموعه از رأس‌ها در گراف دو بخشی متناظر با رأس‌های ابرگراف باشد، و مجموعه‌ی دیگر نشان‌دهنده‌ی ابریال‌ها.)

ب-۵ درخت‌ها

مانند گراف‌ها، نسخه‌های فراوانی از درخت‌ها وجود دارد، که البته تفاوت‌های کمی با یکدیگر دارند. این بخش تعاریف و خصوصیات ریاضیاتی انواع مختلفی از درخت‌ها را ارائه می‌کند. بخش‌های ۱۰-۴ و ۲۲-۱ توضیح می‌دهند که چگونه می‌توان درخت‌ها را در حافظه‌ی کامپیوتر نمایش داد.

ب-۵-۱ درخت‌های آزاد

همان طور که در بخش ب-۴ تعریف شد، یک *درخت آزاد* (free tree) یک گراف بدون جهت، بدون دور، و همبند است. معمولاً از صفت «آزاد» برای درخت‌ها صرف نظر می‌کنیم. اگر یک گراف، بدون دور باشد، ولی احتمالاً ناهمبند، به آن یک *جنگل* (forest) می‌گوییم. بسیاری از الگوریتم‌های که برای درخت‌ها نوشته شده‌اند بر روی جنگل‌ها هم کار می‌کنند. در شکل ب-۴ (الف) یک درخت آزاد و در شکل ب-۴ (ب) یک جنگل را مشاهده می‌کنیم. جنگل شکل ب-۴ (ب) یک درخت نیست، چرا که ناهمبند است. گراف شکل ب-۴ (پ) همبند است، ولی نه یک درخت است و نه یک جنگل، چرا که حاوی یک دور است.

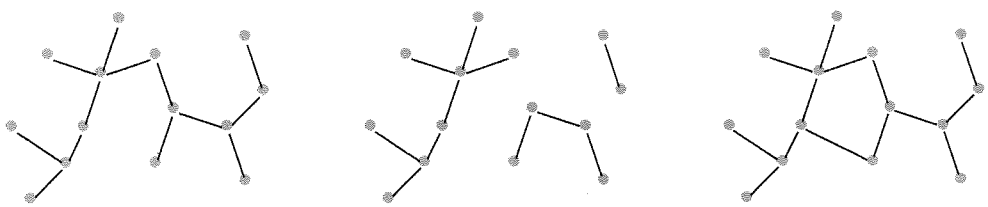
قضیه‌ی زیر نکات بسیاری را در مورد درخت‌های آزاد مشخص می‌کند.

قضیه‌ی ب-۲
ویژگی‌های درخت‌های آزاد

فرض کنید $G=(V,E)$ یک گراف بدون جهت باشد. عبارت‌های زیر یکسان هستند.

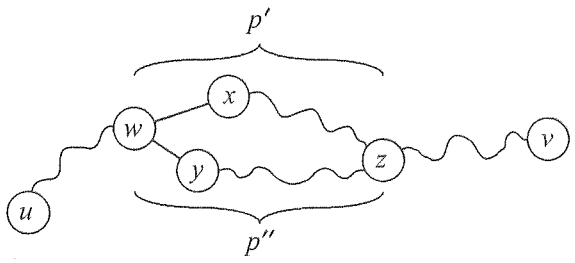
۱. G یک درخت آزاد است.
۲. هر دو رأس در G توسط یک رأس ساده و یکتا به یکدیگر متصل‌اند.
۳. G همبند است، ولی اگر هر رأسی از E حذف شود، گراف حاصل ناهمبند خواهد بود.
۴. G همبند است، و $|E|=|V|-1$.
۵. G بدون دور است، و $|E|=|V|-1$.
۶. G بدون دور است، ولی اگر یک رأس به E اضافه شود، گراف حاصل حاوی یک دور خواهد بود.

اثبات (۲) $\Rightarrow$ (۱): چون درخت‌ها همبند هستند، هر دو رأس در G توسط حداقل یک مسیر ساده به یکدیگر متصل‌اند. فرض کنید u و v رأس‌هایی باشند که دو مسیر ساده و مجزای p_1 و p_2 آن‌ها را به یکدیگر متصل می‌کنند، همان طور که در شکل ب-۵ نشان داده شده است. فرض کنید w رأسی باشد



(الف) (ب) (پ)

شکل ب-۴ (الف) یک درخت آزاد. (ب) یک جنگل. (پ) یک گراف که حاوی یک دور است، و بنابراین نه یک درخت است و نه یک جنگل.



شکل ب-۵ یک مرحله از اثبات قضیه‌ی ب-۲: اگر (۱) G یک درخت آزاد باشد، آن گاه (۲) هر دو رأس در G توسط یک مسیر ساده‌ی یکتا به یکدیگر متصل‌اند. طبق برهان خلف فرض کنید که رأس‌های u و v توسط دو مسیر ساده و مجزای p_1 و p_2 به یکدیگر متصل‌اند. این مسیرها ابتدا در رأس w از یکدیگر جدا شده و سپس در رأس z دوباره به هم می‌پیوندند. از اتصال مسیر p' به معکوس مسیر p'' یک دور تشکیل می‌شود، که تناقض است.

که در آن اولین جدایی مسیرها رخ می‌دهد؛ یعنی w اولین رأس بر روی p_1 و p_2 است که رأس بعدی آن در p_1، x است و در p_2، y، و $x \neq y$. فرض کنید z اولین رأسی باشد که در آن مسیرها دوباره به یکدیگر می‌رسند؛ یعنی z بعد از w اولین رأسی است که هم بر روی p_1 قرار دارد و هم بر روی p_2. فرض کنید p' زیرمسیر p_1 از w تا z باشد، و p'' زیرمسیر p_2 از w تا z. مسیرهای p' و p'' در هیچ رأسی مشترک نیستند، مگر رأس‌های انتهایی. بنابراین مسیر حاصل از اتصال p' و معکوس p'' یک دور است. این با فرض درخت بودن G تناقض دارد. بنابراین، اگر G یک درخت باشد، حداکثر یک مسیر ساده بین هر دو رأس وجود خواهد داشت.

(۳) $\Rightarrow$ (۲): اگر هر دو رأس در G توسط یک مسیر ساده‌ی یکتا به هم متصل شده باشند، آن گاه G همبند است. فرض کنید (u,v) یک یال دلخواه در E باشد. این یال، یک مسیر از u به v است، و بنابراین باید تنها مسیر از u به v باشد. اگر (u,v) را از G حذف کنیم، هیچ مسیری از u به v باقی نمی‌ماند، و بنابراین حذف آن، G را ناهمبند می‌کند.

(۴) $\Rightarrow$ (۳): طبق فرض گراف G همبند است، و طبق تمرین ب-۴-۳ داریم $|E| \geq |V| - ۱$. به وسیله‌ی استقرا اثبات خواهیم کرد $|E| \leq |V| - ۱$. یک گراف همبند با $n = ۱$ یا $n = ۲$ رأس، $n - ۱$ یال دارد. فرض کنید G، $n \geq ۳$ رأس داشته باشد، و تمام گراف‌هایی که (۳) را ارضا می‌کنند و کم‌تر از n رأس دارند، در $|E| \leq |V| - ۱$ صدق می‌کنند. حذف یک یال دلخواه از G، گراف را به $k \geq ۲$ مؤلفه‌ی همبندی تقسیم می‌کند (در واقع $k = ۲$). هر مؤلفه (۳) را ارضا می‌کند، چرا که در غیر این صورت G در (۳) صدق نخواهد کرد. اگر هر مؤلفه‌ی همبندی V_i با مجموعه‌ی یال E_i را به صورت یک درخت آزاد جداگانه در نظر بگیریم، آن گاه چون هر مؤلفه کم‌تر از $|V|$ رأس دارد، طبق فرض استقرا داریم $|E_i| \leq |V_i| - ۱$. بنابراین مجموع تعداد یال‌های تمام مؤلفه‌ها حداکثر $|V| - k \leq |V| - ۲$ است. اضافه کردن یال حذف شده به دست می‌دهد $|E| \leq |V| - ۱$.

(۵) ⇒ (۴): فرض کنید G همبند باشد و $|E|=|V|-1$. باید نشان دهیم که G دور ندارد. فرض کنید G یک دور داشته باشد شامل k رأس $v_1,v_2,\ldots,v_k$، و بدون از دست دادن کلیت، فرض کنید که این دور ساده است. فرض کنید $G_k=(V_k,E_k)$ زیرگراف G شامل دور باشد. توجه کنید که $|V_k|=|E_k|=k$. اگر $k<|V|$، باید یک رأس $v_{k+1}\in V-V_k$ وجود داشته باشد که با یک رأس $v_i\in V_k$ مجاور است، چرا که G همبند است. تعریف می‌کنیم $G_{k+1}=(V_{k+1},E_{k+1})$ زیرگرافی از G باشد که در آن $V_{k+1}=V_k\cup\{v_{k+1}\}$ و $E_{k+1}=E_k\cup\{(v_i,v_{k+1})\}$. توجه کنید که $|V_{k+1}|=|E_{k+1}|=k+1$. اگر $k+1<|V|$، می‌توانیم با تعریف G_{k+2} به همین ترتیب ادامه دهیم، تا زمانی که به $G_n=(V_n,E_n)$ برسیم، که در آن $n=|V|$، و $V_n=V$، و $|E_n|=|V_n|=|V|$. چون G_n یک زیرگراف از G است، داریم $E_n\subseteq E$ و بنابراین $|E|\geq|V|$، که با فرض $|E|=|V|-1$ تناقض دارد. بنابراین G بدون دور است.

(۶) ⇒ (۵): فرض کنید G بدون دور است و $|E|=|V|-1$. فرض می‌کنیم k تعداد مؤلفه‌های همبندی G باشد. طبق تعریف هر مؤلفه‌ی همبندی یک درخت آزاد است، و چون (۱) نتیجه می‌دهد (۵)، مجموع تمام یال‌ها در تمام مؤلفه‌های همبندی G برابر است با $|V|-k$. در نتیجه باید داشته باشیم $k=1$، و G در واقع یک درخت است. چون (۱) نتیجه می‌دهد (۲)، هر دو رأسی در G توسط یک مسیر ساده‌ی یکتا به یکدیگر متصل شده‌اند. بنابراین اضافه کردن هر یالی به G یک دور تولید می‌کند.

(۱) ⇒ (۶): فرض کنید G بدون دور باشد ولی اگر هر یالی به E اضافه شود، یک دور به وجود بیاید. باید نشان دهیم که G همبند است. فرض کنید u و v رأس‌های دلخواه در G باشند. اگر u و v مجاور نباشند، اضافه کردن یال (u,v) یک دور می‌سازد که تمام یال‌های آن غیر از (u,v) به G تعلق دارند. بنابراین یک مسیر از u به v وجود دارد، و چون u و v به صورت دلخواه انتخاب شده بودند، G همبند است. ▒

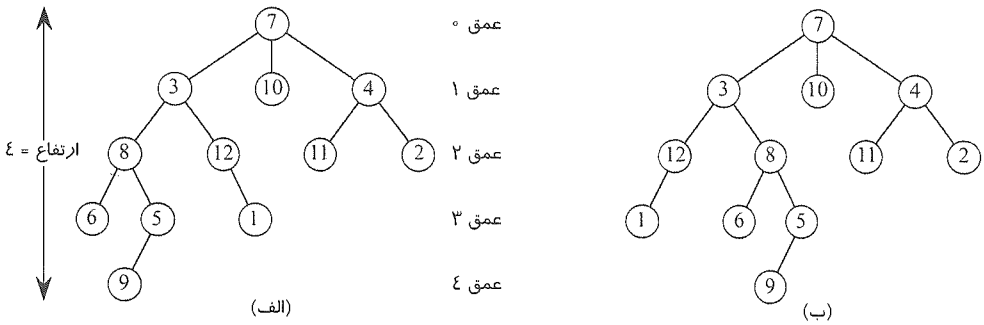
ب-۵-۲ درخت‌های ریشه‌دار و مرتب

یک *درخت ریشه‌دار* (rooted tree) درخت آزادی است که در آن یکی از رأس‌ها از بقیه متمایز است. این رأس متمایز، *ریشه‌ی* (root) درخت نام دارد. معمولاً رأس‌های درخت را گره‌های[1] آن می‌نامیم. شکل ب-۶ (الف) یک درخت ریشه‌دار را نشان می‌دهد با مجموعه‌ای از ۱۲ گره و ریشه‌ی ۷.

یک گره‌ی x را در یک درخت ریشه‌دار T با ریشه‌ی r در نظر بگیرید. هر گره‌ی y بر روی مسیر یکتا از r به x یک *جد* (ancestor) x نام دارد. اگر y یک جد xباشد، آن گاه x یک *نواده‌ی* (descendant) y است. (هر گره هم جد خود محسوب می‌شود و هم نواده‌ی خود.) اگر y یک جد x باشد و $x\neq y$، آن گاه y یک جد اکید (proper ancestor) x است، و x یک *نواده‌ی اکید* (proper descendant) گره‌ی y. *زیردرخت با ریشه‌ی* x درختی است که نوادگان x القا می‌کنند، با ریشه‌ی x. برای مثال، زیردرخت با ریشه‌ی گره‌ی ۸ در شکل ب-۶ (الف) شامل گره‌ی ۸، ۶، ۵، و ۹ است.

---

[1] معمولاً عبارت «گره» در ادبیات نظریه‌ی گراف مترادف با «رأس» در نظر گرفته می‌شود. در این جا از عبارت «گره» برای اشاره به رأس‌های یک درخت ریشه‌دار استفاده خواهیم کرد.



شکل ب-۶ درخت‌های ریشه‌دار و مرتب. (الف) یک درخت ریشه‌دار با ارتفاع ۴. درخت به شکل استاندارد کشیده شده است: ریشه (گره‌ی ۷) در بالا است، فرزندان آن (گره‌های با عمق ۱) زیر آن هستند، و فرزندان آن‌ها (گره‌های با عمق ۲) زیر آن‌ها هستند، و الی آخر. اگر درخت مرتب باشد، ترتیب نسبی چپ به راست فرزندان یک گره اهمیت خواهد داشت؛ در غیر این صورت این ترتیب اهمیت نخواهد داشت. (ب) یک درخت ریشه‌دار دیگر. به عنوان یک درخت ریشه‌دار، این درخت با درخت بخش (الف) معادل است، ولی به عنوان یک درخت مرتب، این دو درخت متفاوت‌اند، چرا که فرزندان گره‌ی ۳ با ترتیبی متفاوت ظاهر می‌شوند.

اگر آخرین یال در مسیر از ریشه‌ی r به یک گره‌ی x در یک درخت T، یال (y, x) باشد، آن گاه y *پدر* (parent) گره‌ی x است، و x یک *فرزند* (child) گره‌ی y. ریشه، تنها گره‌ی T است که پدری ندارد. اگر پدر دو گره مشترک باشد، آن دو گره برادر هستند. یک گره که فرزندی ندارد، یک *گره‌ی خارجی* (external node) و یا یک *برگ* (leaf) است. یک گره‌ی غیر برگ، یک *گره‌ی داخلی* (internal node) نام دارد.

تعداد فرزندان یک گره‌ی x در یک درخت ریشه‌دار T، *درجه‌ی*[1] x (degree) نام دارد. طول مسیر از ریشه‌ی r به گره‌ی x، *عمق* (depth) x در T است. ارتفاع یک گره در یک درخت برابر است با تعداد یال‌های طولانی‌ترین مسیر پایینی از آن گره به یک برگ، و ارتفاع یک درخت برابر است با ارتفاع ریشه‌ی آن. ارتفاع یک درخت همچنین برابر است با بزرگ‌ترین عمق یک گره در آن درخت.

یک *درخت مرتب* (ordered tree)، یک درخت ریشه‌دار است که در آن فرزندان هر گره ترتیب دارند. یعنی اگر یک گره k فرزند داشته باشد، در این صورت یک فرزند اول خواهیم داشت، یک فرزند دوم،...، و یک فرزند kام. دو درخت شکل ب-۶ اگر به صورت درخت مرتب در نظر گرفته شوند، با یکدیگر متفاوت‌اند، ولی اگر به صورت درخت ریشه‌دار در نظر گرفته شوند، یکسان هستند.

---

[1] توجه کنید که درجه‌ی یک گره بستگی به این دارد که T یک درخت ریشه‌دار در نظر گرفته شود یا یک درخت آزاد. درجه‌ی یک رأس در یک درخت آزاد، مانند هر گراف بدون جهتی برابر است با تعداد رأس‌های مجاور آن. با این حال در یک درخت ریشه‌دار درجه برابر است با تعداد فرزندان گره – پدر یک گره در محاسبه‌ی درجه‌ی آن شمرده نمی‌شود.

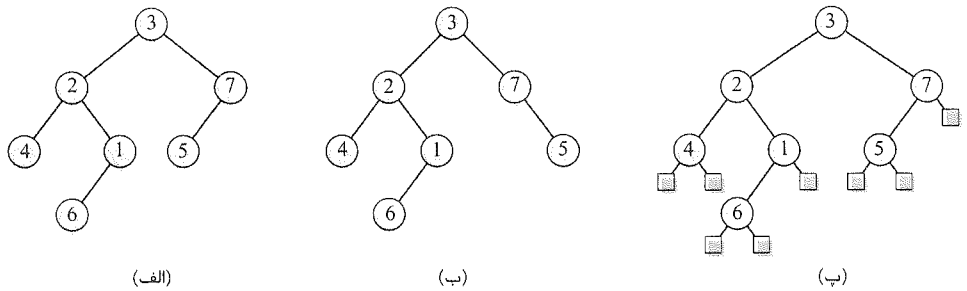
ب-۵-۳ درختان دودویی و درختان مکانی

درختان دودویی به صورت بازگشتی تعریف می‌شوند. یک *درخت دودویی* (binary tree) T ساختاری است که بر روی مجموعه‌ای متناهی از گره‌ها تعریف شده است، و یکی از حالت‌های زیر را دارد:

- حاوی هیچ گره‌ای نیست، و یا
- از سه مجموعه‌ی متفاوت از گره‌ها تشکیل شده است: یک گره‌ی *ریشه* (root)، یک درخت دودویی با نام *زیردرخت سمت چپ* (left subtree)، و یک درخت دودویی دیگر با نام *زیردرخت سمت راست* (right subtree).

درخت دودویی که حاوی هیچ گره‌ای نباشد، *درخت تهی* (empty tree) یا *درخت پوچ* (null tree) نام دارد، و بعضی مواقع با NIL نشان داده می‌شود. اگر زیردرخت سمت چپ ناتهی باشد، ریشه‌ی آن *فرزند چپ* (left child) ریشه‌ی درخت نام دارد. به طور مشابه، ریشه‌ی یک زیردرخت سمت راست ناتهی، *فرزند راست* (right child) ریشه‌ی درخت است. اگر یک زیردرخت، درخت تهی باشد، می‌گوییم آن فرزند *غایب* (absent) است. شکل ب-۷ (الف) یک درخت دودویی را نشان می‌دهد.

یک درخت دودویی فقط یک درخت مرتب نیست که درجه‌ی گره‌های آن حداکثر ۲ باشد. برای مثال، در یک درخت دودویی اگر یک گره فقط یک فرزند داشته باشد، مکان آن فرزند – فرزند چپ یا فرزند راست بودن آن – اهمیت دارد. در یک درخت مرتب، برای یک فرزند تنها، تفاوتی ندارد که سمت چپ باشد یا سمت راست. شکل ب-۷ (ب) یک درخت دودویی را نشان می‌دهد که با درخت



شکل ب-۷ درخت‌های دودویی. (الف) یک درخت دودویی که به صورت استاندارد کشیده شده است. فرزند سمت چپ هر گره، در پایین و سمت چپ آن گره کشیده شده است، و فرزند سمت راست آن، در پایین و راست آن. (ب) یک درخت دودویی متفاوت با درخت دودویی (الف). در (الف)، فرزند سمت چپ ۷، ۵، و فرزند سمت راست آن غایب است. در (ب)، فرزند سمت چپ ۷ غایب، و فرزند سمت راست آن ۵ است. به عنوان درختان مرتب، این دو درخت معادل هستند، ولی به عنوان درختان دودویی، با یکدیگر متفاوت‌اند. (پ) درخت دودویی (الف) که به صورت گره‌های داخلی یک درخت دودویی کامل نشان داده شده است: یک درخت مرتب که در آن درجه‌ی هر گره‌ی داخلی ۲ است. برگ‌های درخت به شکل مربع نشان داده شده‌اند.

دودویی شکل ب–۷ (الف) تفاوت دارد، به خاطر مکان یک گره. با این حال، اگر این دو درخت را به صورت درختان مرتب در نظر بگیریم، با یکدیگر تفاوتی ندارند.

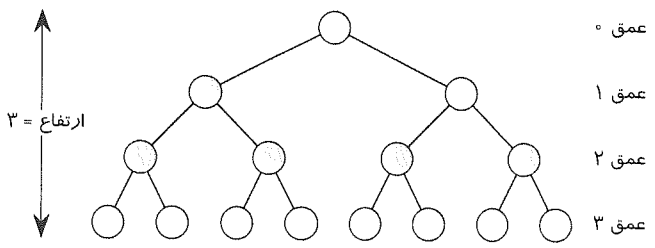
اطلاعات مکانی در یک درخت دودویی را می‌توان به کمک گره‌های داخلی یک درخت مرتب نشان داد، همان طور که در شکل ب–۷ (پ) مشاهده می‌کنید. ایده این است که هر فرزند غایب در درخت دودویی را با یک گره که هیچ فرزندی ندارد، جایگزین کنیم. برگ‌ها به شکل مربع در شکل نشان داده شده‌اند. درخت حاصل، یک *درخت دودویی کامل* (full binary tree) است: هر گره یا یک برگ است، یا درجه‌ی آن دقیقاً ۲ است. هیچ گره‌ای با درجه‌ی ۱ وجود ندارد. بنابراین ترتیب فرزندان یک گره، اطلاعات مکانی آن‌ها را حفظ می‌کند.

اطلاعات مکانی که درختان دودویی را از درختان مرتب جدا می‌کند را می‌توان به درختانی که بیش از دو فرزند برای هر گره دارند، گسترش داد. در یک *درخت مکانی* (positional tree)، فرزندان یک گره با اعداد صحیح مثبت و متمایز شماره‌گذاری می‌شوند. iامین فرزند یک گره *غایب* (absent) است اگر هیچ کدام از فرزندان آن با شماره‌ی i برچسب‌گذاری نشده باشند. یک درخت *k تایی* (k-ary) یک درخت مکانی است که در آن فرزندان با شماره‌های بالاتر از k در هر گره غایب هستند. بنابراین یک درخت دودویی، یک درخت kتایی است، با $k = ۲$.

یک **درخت** *k تایی کامل* (complete k-ary tree) یک درخت kتایی است که در آن عمق تمام برگ‌ها برابر است و درجه‌ی تمام گره‌های داخلی k است. شکل ب–۸ یک درخت دودویی کامل با ارتفاع ۳ را نشان می‌دهد. یک درخت kتایی کامل با ارتفاع h چند برگ دارد؟ ریشه، k فرزند در عمق ۱ دارد، که هر کدام k فرزند در عمق ۲ دارند، و الی آخر. بنابراین تعداد برگ‌ها در عمق h برابر است با k^h. در نتیجه ارتفاع یک درخت کامل kتایی با n برگ برابر است با $\log_k n$. تعداد گره‌های داخلی یک درخت کامل kتایی با ارتفاع h برابر است با

$$۱ + k + k^۲ + \ldots + k^{h-۱} = \sum_{i=۰}^{h-۱} k^i = \frac{k^h - ۱}{k - ۱}$$

طبق تساوی (الف–۵). بنابراین یک درخت دودویی کامل $۲^h - ۱$ گره‌ی داخلی دارد.



شکل ب–۸ یک درخت دودویی کامل با ارتفاع ۳، ۸ برگ، و ۷ گره‌ی داخلی.

تمرین‌ها

ب-۵-۱ تمام درختان آزاد متشکل از سه رأس x، y، و z را بکشید. سپس تمام درختان ریشه‌دار متشکل از گره‌های x، y، و z را بکشید، با x به عنوان ریشه. تمام درختان مرتب متشکل از گره‌های x، y، و z را بکشید، با x به عنوان ریشه. تمام درختان دودویی متشکل از گره‌های x، y، و z را بکشید، با x به عنوان ریشه.

ب-۵-۲ فرض کنید $G=(V,E)$ یک گراف جهت‌دار بدون دور باشد که در آن یک رأس $v_0 \in V$ وجود دارد به طوری که یک مسیر یکتا از v_0 به تمام رأس‌ها وجود دارد. اثبات کنید که نسخه‌ی بدون‌جهت G یک درخت را تشکیل می‌دهد.

ب-۵-۳ به کمک استقرا نشان دهید که تعداد گره‌های با درجه‌ی ۲ در هر درخت دودویی ناتهی، یکی کم‌تر از تعداد برگ‌ها است. نتیجه بگیرید که تعداد گره‌های داخلی در یک درخت دودویی کامل یکی کم‌تر از تعداد برگ‌ها است.

ب-۵-۴ با استفاده از استقرا نشان دهید که ارتفاع یک درخت دودویی ناتهی با n گره حداقل $\lfloor \lg n \rfloor$ است.

★ ب-۵-۵ *طول مسیر داخلی* (internal path length) یک درخت دودویی کامل برابر است با مجموع عمق تمام گره‌های داخلی درخت. به طور مشابه، *طول مسیر خارجی* (external path length) برابر است با مجموع عمق تمام برگ‌های درخت. یک درخت دودویی کامل با n گره، طول مسیر داخلی i، و طول مسیر خارجی e را در نظر بگیرید. اثبات کنید که $e = i + 2n$.

★ ب-۵-۶ اجازه دهید یک وزن $w(x) = 2^{-d}$ به هر برگ x در عمق d در یک درخت دودویی T نسبت دهیم، و فرض کنید L مجموعه‌ی برگ‌های T باشد. اثبات کنید که $\sum_x w(x) \le 1$، که در آن مجموع بر روی تمام برگ‌های x در T گرفته شده است. (این قضیه به *نامساوی کرافت* (Kraft inequality) معروف است.)

★ ب-۵-۷ نشان دهید که هر درخت دودویی با L برگ حاوی یک زیردرخت است که بین (و شامل) $L/3$ و $2L/3$ برگ دارد.

مسائل

ب-۱ رنگ‌آمیزی گراف

با داشتن یک گراف بدون جهت $G=(V,E)$، یک *k-رنگ‌آمیزی* (k-coloring) برای G یک تابع $c: V \to \{0,1,\ldots,k-1\}$ است به طوری که $c(u) \neq c(v)$ برای هر یال $(u,v) \in E$. به عبارت

مسائل

دیگر، اعداد $k-1$ ،...,۱,۰ نشان دهنده‌ی k رنگ هستند، و رأس‌های مجاور باید رنگ‌های متفاوت داشته باشند.

I. نشان دهید که هر درخت را می‌توان با ۲ رنگ، رنگ‌آمیزی کرد.

II. نشان دهید که عبارت‌های زیر معادل هستند:

۱. G دوبخشی است.

۲. G را می‌توان با ۲ رنگ، رنگ‌آمیزی کرد.

۳. G هیچ دوری با طول فرد ندارد.

III. فرض کنید d درجه‌ی بیشینه در یک گراف G باشد. اثبات کنید که می‌توان G را با $d+1$ رنگ، رنگ‌آمیزی کرد.

IV. نشان دهید که اگر G، $O(|V|)$ یال داشته باشد، آن گاه G را می‌توان با $O(\sqrt{|V|})$ رنگ، رنگ‌آمیزی کرد.

ب-۲ گراف‌های دوستانه

هر یک از عبارت‌های زیر را به صورت یک قضیه در مورد گراف‌های بدون جهت بیان کرده و سپس آن را اثبات کنید. فرض کنید که رابطه‌ی دوستی، متقارن است، ولی انعکاسی نیست.

I. در هر گروهی متشکل از $n \geq ۲$ نفر، دو نفر با تعداد یکسانی دوست در گروه وجود دارند.

II. هر گروهی متشکل از شش نفر یا حاوی ۳ نفر است که دو به دو دوست هستند، و یا ۳ نفر که دو به دو غریبه‌اند.

III. هر گروهی از افراد را می‌توان به دو زیرگروه تقسیم کرد به طوری که حداقل نیمی از دوستان هر شخص در زیرگروهی باشد که آن شخص به آن تعلق *ندارد*.

IV. اگر هر کس در گروه با حداقل نیمی از افراد گروه دوست باشد، آن گاه می‌توان این گروه را طوری دور یک میز نشاند که در دو سمت هر کس، دو نفر از دوستان آن شخص قرار گیرند.

ب-۳ درخت‌های دوقسمتی

بسیاری از الگوریتم‌های تقسیم و حل که بر روی گراف‌ها عمل می‌کنند، نیاز دارند که گراف به دو زیرگراف با اندازه‌ی تقریباً مساوی تقسیم شود، که با تقسیم‌بندی رأس‌ها انجام می‌شود. این مسئله تقسیم‌بندی درخت‌ها را بررسی می‌کند، که توسط حذف تعداد کمی از یال‌ها انجام می‌گیرد. در این تقسیم‌بندی، هر گاه پس از حذف یال‌ها دو رأس در یک زیردرخت قرار می‌گیرند، آن گاه این دو رأس در یک قسمت هستند.

I. نشان دهید که با حذف یک یال، می‌توانیم رأس‌های هر درخت دودویی n رأسی را به دو مجموعه‌ی A و B تقسیم کنیم به طوری که $|A| \leq ۳n/۴$ و $|B| \leq ۳n/۴$.

II. نشان دهید که ثابت ۳/۴ در بخش I در بدترین حالت، بهینه است. برای این کار مثالی از یک

درخت دودویی ساده ارائه کنید که در آن در متوازن‌ترین تقسیم‌بندی ممکن از حذف یک یال داریم $|A| = 3n/4$.

III. نشان دهید که با حذف حداکثر $O(\lg n)$ یال، می‌توانیم رأس‌های هر درخت دودویی n رأسی را به دو مجموعه‌ی A و B تقسیم کنیم به طوری که $|A| = \lfloor n/2 \rfloor$ و $|B| = \lceil n/2 \rceil$.

شمارش و احتمالات

در این پیوست ترکیبیات مقدماتی و نظریه‌ی احتمالات را بررسی می‌کنیم. اگر پیش‌زمینه‌ی خوبی در این زمینه‌ها دارید، ممکن بخواهید ابتدای این پیوست را مرور کرده و سپس بر روی بخش‌های پایانی تمرکز کنید. اکثر فصل‌ها به احتمالات نیازی ندارند، ولی برای بعضی از فصل‌ها آموختن احتمالات ضروری است.

بخش پ-۱ نتایج مقدماتی نظریه‌ی شمارش را مرور می‌کند، شامل فرمول‌های استاندارد برای شمارش جایگشت‌ها و ترکیب‌ها. اصول احتمالات و حقایق پایه‌ای مربوط به توزیع احتمالاتی در بخش پ-۲ ارائه شده‌اند. متغیرهای تصادفی در بخش پ-۳ معرفی می‌شوند، به همراه خصوصیات امیدریاضی و واریانس. بخش پ-۴ توزیع‌های دوجمله‌ای و هندسی را بررسی می‌کند که در بحث آزمایش‌های برنولی پیش می‌آیند. بررسی توزیع‌های دوجمله‌ای در بخش پ-۵ ادامه می‌یابد، که بحثی پیشرفته است در مورد «آزمایش»های مربوط به توزیع‌ها.

پ-۱ شمارش

نظریه‌ی شمارش سعی در پاسخ به سؤال «چند تا؟» دارد، بدون این که واقعاً مجبور باشیم تعداد را بشماریم. برای مثال ممکن است بپرسیم «چند عدد مختلف nبیتی وجود دارد؟» یا «چند ترتیب مختلف از n عنصر مجزا وجود دارد؟». در این بخش عناصر نظریه‌ی شمارش را مرور می‌کنیم. چون در بعضی موارد فرض می‌کنیم که خواننده با مفاهیم اولیه‌ی مجموعه‌ها آشنایی دارد، به خواننده توصیه می‌شود که با مرور مطالب بخش ب-۱ کار را آغاز کند.

قوانین جمع و ضرب

بعضی مواقع می‌توان یک مجموعه از عناصر را که می‌خواهیم تعداد آن‌ها را بشماریم، به صورت اجتماع مجموعه‌های ناسازگار و یا ضرب کارتزین مجموعه‌ها توصیف کرد.

قانون جمع (rule of sum) می‌گوید که تعداد راه‌های انتخاب یک عنصر از یکی از دو مجموعه‌ی ناسازگار برابر است با جمع کاردینالیتی مجموعه‌ها. یعنی اگر A و B دو مجموعه‌ی متناهی باشند با اشتراک تهی، آن گاه $|A \cup B| = |A| + |B|$، که از تساوی (ب-۳) نتیجه می‌شود. برای مثال، در هر مکان بر روی پلاک یک اتوموبیل، یا یک حرف نوشته شده است و یا یک رقم. بنابراین تعداد حالت‌های ممکن برای هر مکان عبارت است از $۳۶ = ۱۰ + ۲۶$، چرا که ۲۶ انتخاب برای حروف و ۱۰ انتخاب برای ارقام وجود دارد.

قانون ضرب (rule of product) می‌گوید که تعداد راه‌های انتخاب یک جفت مرتب برابر است با تعداد راه‌های انتخاب عنصر اول ضرب در تعداد راه‌های انتخاب عنصر دوم. یعنی اگر A و B دو مجموعه‌ی متناهی باشند، آن گاه $|A \times B| = |A| \cdot |B|$، که همان تساوی (ب-۴) است. برای مثال، اگر در یک بستنی فروشی ۲۸ طعم از بستنی ارائه شود و ۴ نوع میوه برای روی آن، تعداد کل بستنی‌های ممکن با طعم‌ها و میوه‌های مختلف برابر است با $۱۱۲ = ۴ \times ۲۸$.

رشته‌ها

یک رشته بر روی یک مجموعه‌ی متناهی S، دنباله‌ای است از عناصر S. برای مثال ۸ رشته‌ی دودویی مختلف با طول ۳ وجود دارد:

۰۰۰, ۰۰۱, ۰۱۰, ۰۱۱, ۱۰۰, ۱۰۱, ۱۱۰, ۱۱۱

بعضی مواقع به یک رشته با طول k، یک *k-رشته* (k-string) می‌گوییم. یک *زیررشته‌ی* (substring) s' از یک رشته‌ی s، یک دنباله‌ی مرتب است از عناصر متوالی s. یک *k-زیررشته* (k-substring) از یک رشته عبارت است از یک زیررشته با طول k. برای مثال ۰۱۰ یک ۳-زیررشته از ۰۱۱۰۱۰۰۱ است (۳-زیررشته‌ای که از مکان ۴ آغاز می‌شود)، ولی ۱۱۱ یک زیررشته از ۰۱۱۰۱۰۰۱ نیست.

یک k-رشته بر روی یک مجموعه‌ی S را می‌توان به صورت یک عنصر از ضرب کارتزین S^k از k تایی‌ها دید؛ بنابراین $|S|^k$ رشته با طول k وجود دارد. برای مثال تعداد k-رشته‌های دودویی برابر است با ۲^k. به صورت شهودی برای ساختن یک k-رشته بر روی یک مجموعه‌ی n عضوی، n راه برای انتخاب عنصر اول داریم؛ برای هر یک از این انتخاب‌ها، n راه برای انتخاب عنصر دوم داریم؛ و همین‌طور تا به عنصر kام برسیم. این ساختار به ضرب k تایی $n\,n\,...\,n = n^k$ برای تعداد k-رشته‌ها منجر می‌شود.

جایگشت‌ها

یک *جایگشت* (permutation) از یک مجموعه‌ی متناهی S، یک دنباله‌ی مرتب از تمام عناصر آن است، که در آن هر عنصر دقیقاً یک بار ظاهر می‌شود. برای مثال اگر $S = \{a,b,c\}$، ۶ جایگشت برای S وجود دارد:

$$abc, acb, bac, bca, cab, cba$$

$n!$ جایگشت از یک مجموعه‌ی n عنصری وجود دارد، چرا که اولین عنصر جایگشت را می‌توان به n طریق مختلف انتخاب کرد، دومین عنصر را به $n-1$ طریق مختلف، سومی به $n-2$ طریق، و الی آخر.

یک *k-جایگشت* (k-permutation) از S یک دنباله‌ی مرتب است از k عنصر از S، که در آن هیچ عنصری بیش از یک بار در دنباله ظاهر نمی‌شود. (بنابراین یک جایگشت معمولی، یک n-جایگشت است از یک مجموعه‌ی n عنصری.) تعداد ۲-جایگشت‌های مجموعه‌ی $\{a,b,c,d\}$ ۱۲تا است، که این جایگشت‌ها عبارتند از

$$ab, ac, ad, ba, bc, bd, ca, cb, cd, da, db, dc$$

تعداد k-جایگشت‌های یک مجموعه‌ی n عنصری عبارت است از

$$n(n-1)(n-2)...(n-k+1) = \frac{n!}{(n-k)!} \qquad (پ-1)$$

چرا که به n طریق می‌توان اولین عنصر را انتخاب کرد، به $n-1$ طریق می‌توان دومین عنصر را انتخاب کرد، و همین طور تا زمانی که kامین عنصر از میان $n-k+1$ عنصر انتخاب شود.

ترکیب‌ها

یک *k-ترکیب* (k-combination) از یک مجموعه‌ی n عنصری، یک زیرمجموعه‌ی k عنصری از S است. برای مثال شش ۲-ترکیب از مجموعه‌ی ۴ تایی $\{a,b,c,d\}$ وجود دارد:

$$ab, ac, ad, bc, bd, cd$$

(در این جا برای سادگی برای نشان دادن مجموعه‌ی $\{a,b\}$ از ab استفاده می‌کنیم، و به همین ترتیب برای بقیه‌ی مجموعه‌ها.) برای ساختن یک k-ترکیب از یک مجموعه‌ی n عنصری، k عنصر (متفاوت) از مجموعه انتخاب می‌کنیم. ترتیب انتخاب عناصر اهمیتی ندارد.

تعداد k-ترکیب‌های یک مجموعه‌ی n عنصری را می‌توان برحسب تعداد k-جایگشت‌های یک مجموعه‌ی n عنصری بیان کرد. برای هر k-ترکیب، دقیقاً $k!$ جایگشت از عناصر آن وجود دارد، که هر کدام از آن‌ها یک k-جایگشت متفاوت از مجموعه‌ی nتایی است. بنابراین تعداد k-ترکیب‌های یک مجموعه‌ی nتایی برابر است با تعداد k جایگشت‌ها تقسیم بر $k!$؛ برای تساوی (پ-۱)، این کمیت برابر است با

$$\frac{n!}{k!(n-k)!} \qquad (پ-۲)$$

برای $k = ۰$، این فرمول به ما می‌گوید که تعداد راه‌های انتخاب ۰ عنصر از یک مجموعه‌ی n تایی برابر است با ۱ (نه ۰)، چرا که $۰! = ۱$.

ضرایب دوجمله‌ای

از نماد $\binom{n}{k}$ (بخوانید «انتخاب k از n») برای نشان دادن تعداد k-ترکیب‌های یک مجموعه‌ی n تایی استفاده می‌کنیم. از تساوی (پ-۲) داریم

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

این فرمول نسبت به k و $n-k$ متقارن است:

$$\binom{n}{k} = \binom{n}{n-k} \qquad (پ-۳)$$

این اعداد به *ضرایب دوجمله‌ای* (binomial coefficient) هم معروفند، به دلیل وجود آن‌ها در *توزیع دوجمله‌ای* (binomial expansion):

$$(x+y)^n = \sum_{k=۰}^{n} \binom{n}{k} x^k y^{n-k} \qquad (پ-۴)$$

یک حالت خاص از توزیع دوجمله‌ای زمانی رخ می‌دهد که $x = y = ۱$:

$$۲^n = \sum_{k=۰}^{n} \binom{n}{k}$$

این فرمول مربوط می‌شود به شمارش ۲^n رشته‌ی دودویی با طول n توسط تعداد ۱های درون آن‌ها: $\binom{n}{k}$ رشته‌ی دودویی با طول n وجود دارد که دقیقاً k ۱ دارند، چرا که $\binom{n}{k}$ راه برای انتخاب k مکان از n مکان برای قرار دادن ۱ها در آن‌ها وجود دارد.

کران‌های دوجمله‌ای

بعضی مواقع نیاز داریم برای اندازه‌ی یک ضریب دوجمله‌ای کران تعیین کنیم. برای $۱ \le k \le n$، کران پایین زیر را داریم:

$$\begin{aligned}\binom{n}{k} &= \frac{n(n-۱)\ldots(n-k+۱)}{k(k-۱)\ldots۱} \\ &= \left(\frac{n}{k}\right)\left(\frac{n-۱}{k-۱}\right)\cdots\left(\frac{n-k+۱}{۱}\right) \\ &\ge \left(\frac{n}{k}\right)^k\end{aligned}$$

با بهره بردن از نامساوی $k! \geq (k/e)^k$ که از تقریب استرلینگ (۳–۱۸) به دست می‌آید، کران‌های بالای زیر را به دست می‌آوریم:

$$\begin{aligned}\binom{n}{k} &= \frac{n(n-1)\ldots(n-k+1)}{k(k-1)\ldots 1} \\ &\leq \frac{n^k}{k!} \\ &\leq \left(\frac{en}{k}\right)^k\end{aligned} \qquad (\text{پ–۵})$$

برای تمام $0 \leq k \leq n$ می‌توانیم از استقرا استفاده کرده (تمرین پ–۱–۱۲ را ببینید) و کران زیر را اثبات کنیم:

$$\binom{n}{k} \leq \frac{n^n}{k^k (n-k)^{n-k}} \qquad (\text{پ–۶})$$

که در آن برای سادگی فرض می‌کنیم $0^0 = 1$. برای $k = \lambda n$، که در آن $0 \leq \lambda \leq 1$، این کران را می‌توان به صورت زیر بازنویسی کرد:

$$\begin{aligned}\binom{n}{\lambda n} &\leq \frac{n^n}{(\lambda n)^{\lambda n}((1-\lambda)n)^{(1-\lambda)n}} \\ &= \left(\left(\frac{1}{\lambda}\right)^{\lambda}\left(\frac{1}{1-\lambda}\right)^{1-\lambda}\right)^n \\ &= 2^{nH(\lambda)}\end{aligned}$$

که در آن

$$H(\lambda) = -\lambda \lg \lambda - (1-\lambda)\lg(1-\lambda) \qquad (\text{پ–۷})$$

تابع آنتروپی (entropy function) (دودویی) است، و در آن برای سادگی فرض می‌کنیم $0 \lg 0 = 0$، به طوری که $H(0) = H(1) = 0$.

تمرین‌ها

پ–۱–۱ یک رشته‌ی به طول n چند k–زیررشته دارد؟ (k–زیررشته‌های یکسان در مکان‌های مختلف را متفاوت در نظر بگیرید.) یک رشته‌ی به طول n در کل چند زیررشته دارد؟

پ–۱–۲ یک *تابع دودویی* (Boolean function) با n ورودی و m خروجی، تابعی است از $\{\text{TRUE,FLASE}\}^n$ به $\{\text{TRUE,FLASE}\}^m$. چند تابع دودویی با n ورودی و ۱ خروجی وجود دارد؟ چند تابع دودویی با n ورودی و m خروجی وجود دارد؟

پ-۱-۳ به چند طریق n پروفسور می‌توانند دور یک میز کنفرانس دایره‌ای بنشینند؟ دو وضعیت را یکسان در نظر بگیرید اگر بتوان با چرخیدن یکی را از دیگری به دست آورد.

پ-۱-۴ به چند طریق می‌توان از مجموعه‌ی $\{1, 2, \ldots, 99\}$ سه عدد مختلف انتخاب کرد به طوری که مجموع آن‌ها زوج باشد؟

پ-۱-۵ اتحاد

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} \qquad \text{(پ-۸)}$$

را برای $0 < k \le n$ اثبات کنید.

پ-۱-۶ اتحاد

$$\binom{n}{k} = \frac{n}{n-k}\binom{n-1}{k}$$

را برای $0 < k \le n$ اثبات کنید.

پ-۱-۷ برای انتخاب k شیئ از میان n شیئ، می‌توانیم یکی از عناصر را جدا کرده و حالت‌های انتخاب شدن و انتخاب نشدن آن شیئ را جداگانه در نظر بگیریم. از این رویکرد استفاده کرده و نشان دهید

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

پ-۱-۸ با استفاده از نتیجه‌ی تمرین پ-۱-۷، یک جدول بکشید برای $n = 0,1,\ldots,6$ و $0 \le k \le n$ از ضرایب دوجمله‌ای $\binom{n}{k}$ با $\binom{0}{0}$ در بالا، $\binom{1}{0}$ و $\binom{1}{1}$ در خط بعد، و الی آخر. به چنین جدولی از ضرایب دوجمله‌ای، *مثلث پاسکال* (Pascal's triangle) گویند.

پ-۱-۹ اثبات کنید که

$$\sum_{i=1}^{n} i = \binom{n+1}{2}$$

پ-۱-۱۰ نشان دهید که برای هر $n \ge 0$ و $0 \le k \le n$، مقدار بیشینه‌ی $\binom{n}{k}$ وقتی به دست می‌آید که $k = \lfloor n/2 \rfloor$ یا $k = \lceil n/2 \rceil$.

★پ-۱-۱۱ بحث کنید که برای هر $n \ge 0$، $j \ge 0$، $k \ge 0$، و $j + k \le n$،

$$\binom{n}{j+k} \le \binom{n}{j}\binom{n-j}{k} \qquad \text{(پ-۹)}$$

هم یک اثبات جبری ارائه دهید و هم یک بحث بر مبنای متدی برای انتخاب $j+k$ عنصر در میان n عنصر. مثالی بزنید که در آن حالت مساوی رخ نمی‌دهد.

★پ-۱-۱۲ با استفاده از استقرا بر روی تمام اعداد صحیح $0 \le k \le n/2$ نامساوی (پ-۶) را اثبات کنید، و با استفاده از تساوی (پ-۳) آن را برای تمام $0 \le k \le n$ گسترش دهید.

★پ-۱-۱۳ از تقریب استرلینگ استفاده کرده و اثبات کنید

$$\binom{2n}{n} = \frac{2^{2n}}{\sqrt{\pi n}}(1+O(1/n)) \qquad \text{(پ-۱۰)}$$

★پ-۱-۱۴ با مشتق‌گیری از تابع آنتروپی $H(\lambda)$ نشان دهید که این تابع در $\lambda = 1/2$ به بیشینه‌ی خود می‌رسد. $H(1/2)$ چقدر است؟

★پ-۱-۱۵ نشان دهید که برای هر عدد صحیح $n \ge 0$،

$$\sum_{k=0}^{n}\binom{n}{k}k = n2^{n-1} \qquad \text{(پ-۱۱)}$$

پ-۲ احتمالات

احتمالات یک ابزار اساسی برای طراحی و تحلیل الگوریتم‌های تصادفی و احتمالاتی است. در این بخش مقدمات نظریه‌ی احتمالات بررسی می‌شود.

احتمالات را بر حسب یک *فضای نمونه‌ی* (sample space) S تعریف می‌کنیم، که این فضای نمونه، مجموعه‌ای است که عناصر آن *پیشامدهای پایه* (elementary event) نام دارند. هر پیشامد پایه را می‌توان به صورت نتیجه‌ی احتمالی یک آزمایش در نظر گرفت. برای آزمایش پرتاب دو سکه‌ی متمایز، که هر پرتاب به نتیجه‌ی شیر (H) یا خط (T) منجر می‌شود، می‌توانیم فضای نمونه را به صورت مجموعه‌ی تمام رشته‌های به طول ۲ بر روی {H,T} در نظر بگیریم:

$$S = \{HH, HT, TH, TT\}$$

یک *پیشامد* (event) زیرمجموعه‌ای[1] است از فضای نمونه‌ی S. برای مثال در آزمایش پرتاب دو

[1] برای توزیع احتمالاتی عام، ممکن است زیرمجموعه‌هایی از فضای نمونه‌ی S وجود داشته باشند که پیشامد محسوب نشوند. این وضعیت معمولاً زمانی پیش می‌آید که فضای نمونه نامتناهی و ناشمارا است. شرط اصلی پیشامد بودن زیرمجموعه‌ها این است که مجموعه‌ی پیشامدهای یک فضای نمونه تحت اعمال زیر بسته باشند: مکمل گرفتن از یک پیشامد، اجتماع گرفتن از تعدادی متناهی یا شمارا از پیشامدها، و اشتراک گرفتن از تعدادی متناهی یا شمارا از پیشامدها. اکثر توزیع‌های احتمالاتی که خواهیم دید بر روی فضاهای نمونه‌ی متناهی یا شمارا هستند، و به طور کلی فرض می‌کنیم که تمام زیرمجموعه‌های یک فضای نمونه، پیشامد هستند. یک استثنای قابل توجه، توزیع احتمالاتی یکنواخت پیوسته است، که به زودی معرفی خواهد شد.

سکه، پیشامد به دست آوردن یک شیر و یک خط عبارت است از {HT, TH}. بـه پیشـامد S، *پیشـامد قطعی* (certain event) می‌گوییم، و پیشامد $\varnothing$، *پیشامد پوچ* (null event) نام دارد. می‌گوییم دو پیشامد A و B *دوبه‌دو ناسازگار* (mutually exclusive) هستند اگر $A \cap B = \varnothing$. بعضی مواقـع بـا پیشـامد پایـه‌ی $s \in S$ به صورت پیشامد $\{s\}$ برخورد می‌کنیم. طبق تعریف تمام پیشامدهای پایـه، دوبـه‌دو ناسـازگار هستند.

اصول احتمالات

یک *توزیع احتمالاتی* (probability distribution) $\Pr\{\}$ بر روی یک فضای نمونه‌ی S نگاشتی اسـت از پیشامدهای S به اعداد حقیقی به طوری که *اصول احتمالاتی* (probability axioms) زیر برقرار باشند:

۱. $\Pr\{A\} \geq ۰$ برای هر پیشامد A.

۲. $\Pr\{S\} = ۱$.

۳. $\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\}$ برای هر دو پیشامد ناسازگار A و B. به طور کلـی‌تـر، بـرای هر دنباله (متناهی یا نامتناهی و شمارا) از پیشامدهای $A_۱, A_۲, \ldots$ که دوبه‌دو ناسازگار هستند،

$$\Pr\left\{\bigcup_i A_i\right\} = \sum_i \Pr\{A_i\}$$

به $\Pr\{A\}$ *احتمال* (probability) پیشامد A می‌گوییم. در این جا توجه می‌کنیم که اصل ۲ یک نیاز نرمال کننده است: هیچ نکته‌ی پایه‌ای در مورد انتخاب ۱ به عنوان احتمال پیشامد قطعی وجود ندارد، غیـر از این که این کار طبیعی و ساده است.

نتایج مختلفی مستقیماً از این اصول به همراه نظریه‌ی مجموعه‌ها (بخـش ب-۱ را ببینیـد) حاصـل می‌شوند. احتمال پیشامد پوچ برابر اسـت بـا $\Pr\{\varnothing\} = ۰$. اگـر $A \subseteq B$، آن گـاه $\Pr\{A\} \leq \Pr\{B\}$. بـا استفاده از $\overline{A}$ برای نشان دادن پیشـامد $S - A$ (*مکمـل* A)، داریـم $\Pr\{\overline{A}\} = ۱ - \Pr\{A\}$. بـرای هـر دو پیشامد A و B داریم

$$\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\} - \Pr\{A \cap B\} \qquad (پ-۱۲)$$

$$\leq \Pr\{A\} + \Pr\{B\} \qquad (پ-۱۳)$$

در مثال پرتاب سکه، فرض کنید که هر یک از چهار پیشامد پایـه دارای احتمـال ۱/۴ هسـتند. در ایـن صورت احتمال به دست آوردن حداقل یک شیر برابر است با

$$\Pr\{HH, HT, TH\} = \Pr\{HH\} + \Pr\{HT\} + \Pr\{TH\}$$
$$= ۳/۴$$

به روش دیگر، احتمال به دست آوردن کم‌تر از یک شیر برابر اسـت بـا $\Pr\{TT\} = ۱/۴$، و احتمـال بـه دست آوردن حداقل یک شیر برابر است با $۱ - ۱/۴ = ۳/۴$.

توزیع‌های احتمالاتی گسسته

یک توزیع احتمالاتی *گسسته* (discrete) است اگر بر روی یک فضای نمونه‌ی متناهی یا نامتناهی و شمارا تعریف شده باشد. اگر S فضای نمونه باشد، آن گاه برای هر پیشامد A،

$$\Pr\{A\} = \sum_{s \in A} \Pr\{s\}$$

چرا که پیشامدهای پایه، به طور خاص آن‌ها که در A هستند، دوبه‌دو ناسازگارند. اگر S متناهی باشد و هر پیشامد پایه‌ی $s \in S$ دارای احتمال

$$\Pr\{s\} = 1/|S|$$

باشد، آن گاه یک *توزیع احتمالاتی یکنواخت* (uniform probability distribution) بر روی S داریم. در چنین حالتی معمولاً به آزمایش، «انتخاب یک عنصر از S به صورت تصادفی» گفته می‌شود.

به عنوان یک مثال، فرآیند پرتاب یک *سکه‌ی متقارن* (fair coin) را در نظر بگیرید، که در آن احتمال به دست آوردن شیر و خط هر دو برابر ۱/۲ است. اگر سکه را n بار پرتاب کنیم، توزیع احتمالاتی یکنواختی خواهیم داشت که بر روی فضای نمونه‌ی $S = \{H, T\}^n$ تعریف شده است، که یک مجموعه با اندازه‌ی 2^n است. هر پیشامد پایه در S را می‌توان به صورت یک رشته با طول n بر روی $\{H, T\}$ نشان داد، که هر یک از آن‌ها با احتمال $1/2^n$ رخ می‌دهند. پیشامد

$$A = \{\text{دقیقاً } k \text{ شیر و } k \text{ خط رخ}\}$$

یک زیرمجموعه از S با اندازه‌ی $|A| = \binom{n}{k}$ است، چرا که $\binom{n}{k}$ رشته با طول n بر روی $\{H, T\}$ وجود دارد که دقیقاً k تا H دارند. بنابراین احتمال وقوع A برابر است با $\Pr\{A\} = \binom{n}{k} \Big/ 2^n$.

توزیع احتمالاتی یکنواخت پیوسته

توزیع احتمالاتی یکنواخت پیوسته نمونه‌ای است از یک توزیع احتمالاتی که در آن تمام زیرمجموعه‌های فضای نمونه پیشامد در نظر گرفته می‌شوند. توزیع احتمالاتی یکنواخت پیوسته بر روی یک بازه‌ی بسته $[a,b]$ از اعداد حقیقی تعریف می‌شود، که در آن $a < b$. به طور شهودی می‌خواهیم احتمال وقوع تمام نقاط در بازه‌ی $[a,b]$ یکسان باشد. با این حال، تعداد نقاط ناشمارا است، پس اگر به تمام نقاط یک احتمال مثبت و متناهی یکسان بدهیم، نمی‌توانیم اصول ۲ و ۳ را با هم ارضا کنیم. به همین دلیل فقط به بعضی از زیرمجموعه‌های S یک احتمال نسبت می‌دهیم به طوری که اصول احتمالاتی برای این پیشامدها ارضا شوند.

برای هر بازه‌ی بسته‌ی $[c,d]$ که در آن $a \le c \le d \le b$، *توزیع احتمالاتی یکنواخت پیوسته* (continuous uniform probability distribution) احتمال پیشامد $[c,d]$ را به صورت زیر تعریف می‌کند:

$$\Pr\{[c,d]\} = \frac{d-c}{b-a}$$

توجه کنید که برای هر نقطه‌ی $x = [x,x]$، احتمال وقوع x برابر ۰ است. اگر نقاط انتهایی یک بازه‌ی $[c,d]$ را حذف کنیم، به بازه‌ی (c,d) می‌رسیم. چون $[c,d] = [c,c] \cup (c,d) \cup [d,d]$، اصل ۳ به دست می‌دهد $\Pr\{[c,d]\} = \Pr\{(c,d)\}$. به طور کلی مجموعه‌ی پیشامدها برای توزیع احتمالاتی یکنواخت پیوسته شامل هر زیرمجموعه‌ای از فضای نمونه‌ی $[a,b]$ است که می‌توان آن را از طریق اجتماع تعدادی متناهی یا شمارا از بازه‌های باز یا بسته به دست آورد، به علاوه‌ی چند مجموعه‌ی خاص و پیچیده‌ی دیگر.

احتمال شرطی و استقلال

بعضی مواقع، از قبل اطلاعاتی جزئی در مورد نتیجه‌ی یک آزمایش داریم. برای مثال فرض کنید که دوست شما دو سکه‌ی متقارن را پرتاب کرده است، و به شما گفته است که نتیجه‌ی پرتاب یکی از سکه‌ها، شیر بوده است. احتمال این که هر دو سکه شیر آمده باشند چقدر است؟ این اطلاعات داده شده، احتمال وقوع دو خط را منتفی می‌کند. احتمال وقوع سه پیشامد پایه‌ی دیگر برابر است، پس نتیجه می‌گیریم که هر یک از آن‌ها با احتمال ۱/۳ رخ می‌دهند. چون فقط یکی از این سه پیشامد، آمدن دو شیر است، جواب سؤال ما برابر است با ۱/۳.

احتمال شرطی، مفهوم داشتن اطلاعات جزئی در مورد نتیجه‌ی آزمایش را به صورت رسمی بیان می‌کند. *احتمال شرطی* (conditional probability) یک پیشامد A با این اطلاعات که پیشامد B به وقوع پیوسته است، به صورت زیر تعریف می‌شود:

$$\Pr\{A \mid B\} = \frac{\Pr\{A \cap B\}}{\Pr\{B\}} \qquad (\text{پ-}۱۴)$$

در صورتی که $\Pr\{B\} \neq 0$. ("$\Pr\{A \mid B\}$" را می‌خوانیم «احتمال A به شرط B».) به طور شهودی، چون می‌دانیم که B رخ داده است، پیشامد این که A هم رخ دهد $A \cap B$ است. یعنی $A \cap B$ مجموعه‌ی نتایجی است که در آن هم A رخ می‌دهد و هم B. چون نتیجه، یکی از پیشامدهای پایه در B است، احتمال تمام پیشامدهای پایه در B را با تقسیم کردن بر $\Pr\{B\}$ نرمال می‌کنیم، به طوری که مجموع آن‌ها برابر ۱ شود. بنابراین احتمال شرطی A به شرط B برابر است با نسبت احتمال پیشامد $A \cap B$ به احتمال پیشامد B. در مثال بالا A پیشامد وقوع دو شیر است، و B پیشامد این که حداقل یک شیر آمده باشد. بنابراین $\Pr\{A \mid B\} = (۱/۴)/(۳/۴) = ۱/۳$.

دو پیشامد مستقل (independent) هستند اگر

$$\Pr\{A \cap B\} = \Pr\{A\}\Pr\{B\} \qquad (\text{پ-}۱۵)$$

برای مثال فرض کنید که دو سکه‌ی متقارن پرتاب شده‌اند و نتایج آن‌ها مستقل است. در این صورت احتمال وقوع دو شیر برابر است با $(۱/۲)(۱/۲) = ۱/۴$. اکنون فرض کنید که یک پیشامد این باشد که اولین سکه شیر بیاید و پیشامد دیگر این باشد که سکه‌ی دوم متفاوت از سکه‌ی اول باشد. هر یک

از این پیشامدها با احتمال ۱/۲ رخ می‌دهند، و احتمال رخ‌داد هر دو پیشامد ۱/۴ است؛ بنابراین طبق تعریف استقلال، این پیشامدها مستقل هستند، با این که ممکن است تصور کنید هر دو پیشامد به اولین سکه بستگی دارد. نهایتاً فرض کنید که سکه‌ها طوری به هم متصل شده‌اند که یا هر دو شیر می‌آیند و یا هر دو خط، و این که هر دو احتمال با یکدیگر برابر است. در این صورت احتمال این که هر سکه شیر بیاید ۱/۲ است، ولی احتمال این که هر دو شیر بیایند (۱/۲)(۱/۲) ≠ ۱/۲ است. در نتیجه، پیشامد این که یکی از سکه‌ها شیر بیاید از پیشامد این که دیگری شیر بیاید مستقل نیست.

گروه $A_1, A_2, ..., A_n$ از پیشامدها، *دوبه‌دو مستقل* (pairwise independent) هستند اگر

$$\Pr\{A_i \cap A_j\} = \Pr\{A_i\}\Pr\{A_j\}$$

برای تمام $1 \le i \le j \le n$. می‌گوییم پیشامدهای گروه *(متقابلا) مستقل* (mutually independent) هستند اگر هر k-زیرمجموعه‌ی $A_{i_1}, A_{i_2}, ..., A_{i_k}$ از گروه، که در آن $2 \le k \le n$ و $1 \le i_1 < i_2 < ... < i_k \le n$، در رابطه‌ی

$$\Pr\{A_{i_1} \cap A_{i_2} \cap ... \cap A_{i_k}\} = \Pr\{A_{i_1}\}\Pr\{A_{i_2}\}...\Pr\{A_{i_k}\}$$

صدق کند. برای مثال فرض کنید که دو سکه‌ی متقارن را پرتاب می‌کنیم. فرض می‌کنیم A_1 پیشامد شیر آمدن سکه‌ی اول، A_2 پیشامد شیر آمدن سکه‌ی دوم، و A_3 پیشامد متفاوت بودن دو سکه باشد. داریم

$$\begin{aligned}
\Pr\{A_1\} &= 1/2 \\
\Pr\{A_2\} &= 1/2 \\
\Pr\{A_3\} &= 1/2 \\
\Pr\{A_1 \cap A_2\} &= 1/4 \\
\Pr\{A_1 \cap A_3\} &= 1/4 \\
\Pr\{A_2 \cap A_3\} &= 1/4 \\
\Pr\{A_1 \cap A_2 \cap A_3\} &= 0
\end{aligned}$$

چون برای $1 \le i < j \le 3$ داریم $\Pr\{A_i \cap A_j\} = \Pr\{A_i\}\Pr\{A_j\} = 1/4$، پیشامدهای A_1، A_2، و A_3 دوبه‌دو مستقل‌اند. با این حال پیشامدها متقابلا مستقل نیستند، چرا که $\Pr\{A_1 \cap A_2 \cap A_3\} = 0$ و $\Pr\{A_1\}\Pr\{A_2\}\Pr\{A_3\} = 1/8 \ne 0$.

قضیه‌ی بیز

از تعریف احتمال شرطی (پ-۱۴) و قانون جابه‌جایی $A \cap B = B \cap A$، نتیجه می‌شود که برای دو پیشامد A و B، هر یک با احتمال غیر صفر،

$$\begin{aligned}
\Pr\{A \cap B\} &= \Pr\{B\}\Pr\{A \mid B\} \\
&= \Pr\{A\}\Pr\{B \mid A\}
\end{aligned} \qquad (پ-۱۶)$$

با حل معادله نسبت به $\Pr\{A \mid B\}$ به دست می‌آوریم

$$\Pr\{A \mid B\} = \frac{\Pr\{A\}\Pr\{B \mid A\}}{\Pr\{B\}} \qquad (پ-۱۷)$$

که به *قضیه‌ی بیز* (Bayes's theorem) معروف است. مخرج $\Pr\{B\}$ یک ثابت نرمال کننده است که به صورت زیر آن را توضیح می‌دهیم. چون $B = (B \cap A) \cup (B \cap \overline{A})$ و چون $B \cap A$ و $B \cap \overline{A}$ پیشامدهای متقابلا مستقل هستند،

$$\begin{aligned}\Pr\{B\} &= \Pr\{B \cap A\} + \Pr\{B \cap \overline{A}\} \\ &= \Pr\{A\}\Pr\{B \mid A\} + \Pr\{\overline{A}\}\Pr\{B \mid \overline{A}\}\end{aligned}$$

با جایگذاری در تساوی (پ-۱۷) یک معادل برای قضیه‌ی بیز به دست می‌آوریم:

$$\Pr\{A \mid B\} = \frac{\Pr\{A\}\Pr\{B \mid A\}}{\Pr\{A\}\Pr\{B \mid A\} + \Pr\{\overline{A}\}\Pr\{B \mid \overline{A}\}} \qquad (\text{پ-۱۸})$$

قضیه‌ی بیز می‌تواند محاسبه‌ی احتمالات شرطی را ساده‌تر کند. برای مثال فرض کنید که یک سکه‌ی متقارن داریم و یک سکه‌ی نامتقارن که همیشه شیر می‌آید. یک آزمایش متشکل از سه پیشامد مستقل انجام می‌دهیم: یکی از دو سکه به صورت تصادفی انتخاب می‌شود، یک بار پرتاب می‌شود، و سپس دوباره پرتاب می‌شود. فرض کنید که سکه‌ی انتخاب شده در هر دو بار شیر می‌آید. احتمال این که این سکه، همان سکه‌ی نامتقارن باشد چقدر است؟

این مسئله را با استفاده از قضیه‌ی بیز حل می‌کنیم. فرض کنید A پیشامد انتخاب سکه‌ی نامتقارن باشد، و B پیشامد این که سکه‌ی انتخاب شده هر دو بار شیر بیاید. می‌خواهیم $\Pr\{A \mid B\}$ را تعیین کنیم. داریم $\Pr\{A\} = ۱/۲$، $\Pr\{B \mid A\} = ۱$، $\Pr\{\overline{A}\} = ۱/۲$، و $\Pr\{B \mid \overline{A}\} = ۱/۴$؛ بنابراین

$$\begin{aligned}\Pr\{A \mid B\} &= \frac{(۱/۲)\cdot ۱}{(۱/۲)\cdot ۱ + (۱/۲)\cdot(۱/۴)} \\ &= ۴/۵\end{aligned}$$

تمرین‌ها

پ-۲-۱ پروفسور Rosencrantz یک سکه‌ی متقارن را یک بار پرتاب می‌کند، و پروفسور Guildenstern یک سکه‌ی متقارن را دو بار پرتاب می‌کند. احتمال این که پروفسور Rosencrantz بیشتر از پروفسور Guildenstern شیر به دست آورد چقدر است؟

پ-۲-۲ *نامساوی بول* (Boole's inequality) را اثبات کنید: برای هر دنباله‌ی متناهی یا نامتناهی و شمارا از پیشامدهای $A_۱, A_۲, \ldots$ داریم

$$\Pr\{A_۱ \cup A_۲ \cup \ldots\} \le \Pr\{A_۱\} + \Pr\{A_۲\} + \ldots \qquad (\text{پ-۱۹})$$

پ-۲-۳ دسته‌ای از ۱۰ کارت، هر یک با یک شماره‌ی متمایز از ۱ تا ۱۰، بر زده می‌شود تا کارت‌ها کاملاً مخلوط شوند. سه کارت به صورت هم‌زمان از دسته خارج می‌شوند. احتمال این که سه کارت انتخاب شده به ترتیب صعودی مرتب شده باشند، چقدر است؟

پ-۲-۴ اثبات کنید که

$$\Pr\{A \mid B\} + \Pr\{\overline{A} \mid B\} = ۱$$

پ-۲-۵ اثبات کنید که برای هر مجموعه‌ای از پیشامدهای $A_۱, A_۲, ..., A_n$،

$$\Pr\{A_۱ \cap A_۲ \cap ... \cap A_n\} = \Pr\{A_۱\} \cdot \Pr\{A_۲ \mid A_۱\} \cdot \Pr\{A_۳ \mid A_۱ \cap A_۲\} \cdot ... \cdot \Pr\{A_n \mid A_۱ \cap A_۲ \cap ... \cap A_{n-۱}\}$$

★پ-۲-۶ رویه‌ای را توصیف کنید که دو عدد صحیح a و b را به عنوان ورودی دریافت می‌کند، به طوری که $۰ < a < b$، و با استفاده از پرتاب‌های یک سکه‌ی متقارن، با احتمال a/b شیر را به خروجی می‌دهد و با احتمال $(b-a)/b$ خط را. یک کران برای امیدریاضی تعداد پرتاب‌های سکه ارائه دهید، که باید $O(۱)$ باشد. (راهنمایی: a/b را به صورت دودویی نمایش دهید.)

★پ-۲-۷ نشان دهید چطور می‌توان مجموعه‌ای از n پیشامد دوبه‌دو مستقل ساخت به طوری که هیچ زیرمجموعه‌ای از $k > ۲$ عضو از آن‌ها متقابلا مستقل نباشد.

★پ-۲-۸ با داشتن یک پیشامد C، دو پیشامد A و B مستقل مشروط (conditionally independent) هستند اگر

$$\Pr\{A \cap B \mid C\} = \Pr\{A \mid C\} \Pr\{B \mid C\}$$

یک مثال ساده ولی نابدیهی از دو پیشامد ارائه کنید که مستقل نیستند ولی نسبت به یک پیشامد سوم، مستقل مشروط هستند.

★پ-۲-۹ فرض کنید که در یک مسابقه‌ی تلویزیونی هستید که جایزه‌ی آن پشت یکی از سه پرده پنهان شده است. اگر بتوانید پرده‌ی درست را انتخاب کنید، جایزه را خواهید برد. پس از این که یک پرده را انتخاب کردید، ولی قبل از کنار زدن پرده، مجری یکی از دو پرده‌ی دیگر را که پشت آن جایزه نیست، کنار می‌زند، و از شما می‌پرسد که آیا می‌خواهید انتخاب فعلی خود را با پرده‌ی باقی‌مانده عوض کنید یا خیر. اگر این کار را بکنید، شانس شما چه تغییری می‌کند؟

★پ-۲-۱۰ رئیس یک زندان از میان سه زندانی، به صورت تصادفی یکی را انتخاب کرده که آزاد کند. دو زندانی دیگر اعدام خواهند شد. نگهبان می‌داند که کدام زندانی آزاد خواهد شد، ولی دادن اطلاعات به زندانی‌ها در مورد سرنوشت آن‌ها ممنوع است. اجازه دهید زندانی‌ها را X، Y، و Z بنامیم. زندانی X به صورت خصوصی از نگهبان می‌پرسد که از دو زندانی دیگر کدام یک اعدام خواهد شد، با این ادعا که چون او می‌داند که از آن دو زندانی حتماً یکی اعدام خواهد شد، نگهبان با جواب به این سؤال هیچ اطلاعاتی در مورد سرنوشتش به او نخواهد داد. نگهبان به X می‌گوید که Y اعدام خواهد شد. اکنون

زندانی X احساس بهتری دارد، چرا که به نظر او اکنون یا زندانی Z آزاد خواهد شد و یا خود او، که بدین معنی است که احتمال آزاد شدن او اکنون ۱/۲ است. آیا او درست فکر می‌کند، یا شانس آزاد شدن او همچنان ۱/۳ است؟ توضیح دهید.

پ-۳ متغیرهای تصادفی گسسته

یک *متغیر تصادفی (گسسته)* (discrete random variable) X تابعی است از یک فضای نمونه‌ی متناهی یا نامتناهی و شمارای S به اعداد حقیقی. این تابع یک عدد حقیقی به هر یک از نتایج ممکن برای یک آزمایش نسبت می‌دهد، که به ما اجازه می‌دهد که بر روی توزیع احتمالاتی القا شده توسط مجموعه‌ی اعداد حاصل کار کنیم. متغیرهای تصادفی را می‌توان برای فضاهای نمونه‌ی نامتناهی و ناشمارا هم تعریف کرد، ولی این کار مسائل تکنیکی خاصی را به همراه دارد که در این جا نیازی به بررسی آن‌ها نیست. از این رو فرض خواهیم کرد که متغیرهای تصادفی، گسسته هستند.

برای یک متغیر تصادفی X و یک عدد حقیقی x، پیشامد $X = x$ را به صورت $\{s \in S : X(s) = x\}$ تعریف می‌کنیم؛ بنابراین

$$\Pr\{X = x\} = \sum_{\{s \in S : X(s) = x\}} \Pr\{s\}$$

تابع

$$f(x) = \Pr\{X = x\}$$

تابع چگالی احتمال (probability density function) متغیر تصادفی X است. طبق اصول احتمالات، $\Pr\{X = x\} \geq ۰$ و $\sum_x \Pr\{X = x\} = ۱$.

به عنوان یک مثال، آزمایش انداختن یک جفت تاس معمولی با ۶ جهت را در نظر بگیرید. ۳۶ پیشامد پایه‌ی ممکن در فضای نمونه برای این آزمایش وجود دارد. فرض می‌کنیم که توزیع احتمال یکنواخت باشد، و احتمال وقوع تمام پیشامدهای پایه‌ی $s \in S$ با یکدیگر برابر است: $\Pr\{s\} = ۱/۳۶$. متغیر تصادفی X را به صورت مقدار بیشینه در میان دو تاس تعریف می‌کنیم. داریم $\Pr\{X = ۳\} = ۵/۳۶$، چرا که X به ۵ تا از ۳۶ پیشامد پایه مقدار ۳ نسبت می‌دهد، که این پیشامدها عبارتند از $(۱,۳)$، $(۲,۳)$، $(۳,۳)$، $(۳,۲)$، و $(۳,۱)$.

تعریف چندین متغیر تصادفی بر روی یک فضای نمونه کار معمولی است. اگر X و Y متغیرهای تصادفی باشند، تابع

$$f(x,y) = \Pr\{X = x \text{ و } Y = y\}$$

تابع چگالی احتمال توام (joint probability density function) X و Y است. برای یک مقدار ثابت y،

$$\Pr\{Y = y\} = \sum_x \Pr\{X = x \text{ و } Y = y\}$$

و به طور مشابه برای یک مقدار ثابت x،

$$\Pr\{X = x\} = \sum_y \Pr\{X = x \text{ و } Y = y\}$$

با استفاده از تعریف (پ-۱۴) برای احتمال شرطی، داریم

$$\Pr\{X = x \mid Y = y\} = \frac{\Pr\{X = x \text{ و } Y = y\}}{\Pr\{Y = y\}}$$

می‌گوییم دو متغیر تصادفی X و Y *مستقل* هستند اگر برای تمام x و y، پیشامد $X = x$ و $Y = y$ مستقل باشند، یا به طور معادل اگر برای تمام x، و y داشته باشیم

$$\Pr\{X = x \text{ و } Y = y\} = \Pr\{X = x\}\Pr\{Y = y\}$$

با داشتن مجموعه‌ای از متغیرهای تصادفی که بر روی یک فضای نمونه‌ای تعریف شده‌اند، می‌توان متغیرهای تصادفی جدیدی به صورت مجموع، ضرب، یا توابع دیگری بر روی متغیرهای اصلی تعریف کرد.

امیدریاضی متغیرهای تصادفی

ساده‌ترین و پرکاربردترین نتیجه از توزیع یک متغیر تصادفی «متوسط» مقادیری است که آن متغیر دریافت می‌کند. *امیدریاضی* (expected value) یک متغیر تصادفی گسسته‌ی X برابر است با

$$E[X] = \sum_x x \Pr\{X = x\} \qquad (پ-۲۰)$$

که اگر این سری متناهی باشد، و یا مطلقاً همگرا باشد، خوش‌تعریف است. بعضی مواقع امیدریاضی X به صورت μx، و یا اگر متغیر مورد نظر از محتوای متن، واضح باشد، فقط با μ نشان داده می‌شود.

یک بازی را در نظر بگیرید که در آن باید دو سکه‌ی متقارن را پرتاب کنید. با آمدن هر شیر، شما ۳ دلار دریافت می‌کنید، ولی برای هر خط ۲ دلار از دست می‌دهید. امیدریاضی متغیر تصادفی X که نشان دهنده‌ی مقدار دریافتی شما است، برابر است با

$$\begin{aligned} E[X] &= ۶\Pr\{۲H\} + ۱\Pr\{۱H, ۱T\} - ۴\Pr\{۲T\} \\ &= ۶(۱/۴) + ۱(۱/۲) - ۴(۱/۴) \\ &= ۱ \end{aligned}$$

امیدریاضی مجموع دو متغیر تصادفی برابر است با مجموع امیدریاضی‌های آن دو، یعنی،

$$E[X + Y] = E[X] + E[Y] \qquad (پ-۲۱)$$

در صورتی که $E[X]$ و $E[Y]$ تعریف شده باشند. به این خصوصیت، *خطی بودن امیدریاضی* می‌گوییم، که حتی در صورتی که X و Y مستقل نباشند هم صحیح است. این خصوصیت همچنین برای سری‌های متناهی و مطلقاً همگرا از امیدریاضی‌ها هم برقرار است. خطی بودن امیدریاضی، خصوصیت اصلی است که ما را قادر می‌کند با استفاده از متغیرهای تصادفی شاخص، تحلیل‌های

احتمالاتی انجام دهیم (بخش ۵-۲ را ببینید).

اگر X یک متغیر تصادفی باشد، هر تابع $g(x)$ یک متغیر تصادفی جدید $g(X)$ تعریف می‌کند. اگر امیدریاضی $g(X)$ تعریف شده باشد، آن گاه

$$E[g(X)] = \sum_x g(x) \cdot \Pr\{X = x\}$$

با قرار دادن $g(x) = ax$، برای هر ثابت a داریم

$$E[aX] = a\,E[X] \qquad (\text{پ-۲۲})$$

در نتیجه امیدریاضی خطی است: برای هر دو متغیر تصادفی X و Y و هر ثابت a،

$$E[aX + Y] = aE[X] + E[Y] \qquad (\text{پ-۲۳})$$

وقتی دو متغیر تصادفی X و Y مستقل هستند، و امیدریاضی هر یک تعریف شده است،

$$\begin{aligned} E[XY] &= \sum_x \sum_y xy \Pr\{X = x \text{ و } Y = y\} \\ &= \sum_x \sum_y xy \Pr\{X = x\} \Pr\{Y = y\} \\ &= \left(\sum_x x \Pr\{X = x\}\right)\left(\sum_y y \Pr\{Y = y\}\right) \\ &= E[X]E[Y] \end{aligned}$$

به طور کلی اگر n متغیر تصادفی $X_1, X_2, \ldots, X_n$ متقابلا مستقل باشند،

$$E[X_1 X_2 \ldots X_n] = E[X_1]E[X_2]\ldots E[X_n] \qquad (\text{پ-۲۴})$$

وقتی یک متغیر تصادفی X مقادیر خود را از مجموعه‌ی اعداد طبیعی $\mathbb{N} = \{0,1,2,\ldots\}$ دریافت می‌کند، یک فرمول مناسب برای امیدریاضی آن وجود دارد:

$$\begin{aligned} E[X] &= \sum_{i=0}^{\infty} i \Pr\{X = i\} \qquad (\text{پ-۲۵}) \\ &= \sum_{i=0}^{\infty} i \left(\Pr\{X \geq i\} - \Pr\{X \geq i+1\}\right) \\ &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \end{aligned}$$

چرا که هر عبارت $\Pr\{X \geq i\}$ به تعداد i بار اضافه شده و $i-1$ بار کم می‌شود (غیر از $\Pr\{X \geq 0\}$ که ۰ بار اضافه شده و اصلاً کم نمی‌شود).

وقتی یک تابع محدب $f(x)$ را به یک متغیر تصادفی X اعمال می‌کنیم، نامساوی ینسن (Jensen's inequality) به دست می‌دهد

$$E[f(X)] \geq f(E[X]) \qquad (\text{پ-۲۶})$$

با این فرض که امیدریاضی‌ها وجود داشته و متناهی هستند. (یک تابع $f(x)$ *محدب* (convex) است اگر برای هر x و y و برای تمام $0 \le \lambda \le 1$ داشته باشیم $f(\lambda x + (1-\lambda)y) \le \lambda f(x) + (1-\lambda) f(y)$.)

واریانس و انحراف معیار

امیدریاضی یک متغیر تصادفی به ما نمی‌گوید که مقادیر متغیر چگونه «توزیع» شده‌اند. برای مثال اگر متغیرهای تصادفی X و Y را داشته باشیم به صورتی که $\Pr\{X = 1/4\} = \Pr\{X = 3/4\} = 1/2$ و $\Pr\{Y = 0\} = \Pr\{Y = 1\} = 1/2$، آن گاه هر دو مقدار $E[X]$ و $E[Y]$ برابر $1/2$ است، با این که مقادیر واقعی گرفته شده توسط Y از مقادیر واقعی گرفته شده توسط X از میانگین دورتر هستند.

مفهوم واریانس به صورت ریاضی توضیح می‌دهد که مقادیر اخذ شده توسط متغیرهای تصادفی چقدر از میانگین دور هستند. *واریانس* (variance) یک متغیر تصادفی X با میانگین $E[X]$ برابر است با

$$\begin{aligned} \mathrm{Var}[X] &= E[(X - E[X])^2] \\ &= E[X^2 - 2XE[X] + E^2[X]] \\ &= E[X^2] - 2E[XE[X]] + E^2[X] \\ &= E[X^2] - 2E^2[X] + E^2[X] \\ &= E[X^2] - E^2[X] \end{aligned} \qquad (\text{پ–۲۶})$$

برای تأیید تساوی $E[E^2[X]] = E^2[X]$، توجه کنید که چون $E[X]$ یک عدد حقیقی است و نه یک متغیر تصادفی، $E^2[X]$ هم همین طور است. تساوی $E[XE[X]] = E^2[X]$ از تساوی (پ–۲۲) نتیجه می‌شود، با $a = E[X]$. تساوی (پ–۲۷) را می‌توانیم طوری بازنویسی کنیم که به توصیفی برای امیدریاضی مربع یک متغیر تصادفی برسیم:

$$E[X^2] = \mathrm{Var}[X] + E^2[X] \qquad (\text{پ–۲۸})$$

واریانس یک متغیر تصادفی X و واریانس aX به یکدیگر مرتبط‌اند (تمرین پ–۳–۱۰ را ببینید):

$$\mathrm{Var}[aX] = a^2 \mathrm{Var}[X]$$

اگر X و Y متغیرهای تصادفی مستقل باشند،

$$\mathrm{Var}[X + Y] = \mathrm{Var}[X] + \mathrm{Var}[Y]$$

به طور کلی، اگر n متغیر تصادفی $X_1, X_2, \ldots, X_n$ دوبه‌دو مستقل باشند، آن گاه

$$\mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{Var}[X_i] \qquad (\text{پ–۲۹})$$

انحراف معیار (standard deviation) یک متغیر تصادفی X برابر است با ریشه‌ی مثبت واریانس X. بعضی مواقع انحراف معیار یک متغیر تصادفی X را با σX، و اگر متغیر تصادفی مورد نظر X از

محتوای متن قابل تشخیص باشد، با σ نشان می‌دهند. با این نشان‌گذاری، واریانس X به صورت σ^2 خواهد بود.

تمرین‌ها

پ-۳-۱ دو تاس معمولی با شش جهت انداخته می‌شوند. امیدریاضی مجموع مقادیر تاس‌ها چقدر است؟ امیدریاضی بیشینه‌ی مقادیر تاس‌ها چقدر است؟

پ-۳-۲ یک آرایه‌ی $A[1..n]$ حاوی n عدد متمایز است که به صورت تصادفی مرتب شده‌اند، که در آن احتمال وقوع هر جایگشت ممکن از این n عدد برابر است. امیدریاضی اندیس مقدار بیشینه در آرایه چقدر است؟ امیدریاضی اندیس مقدار کمینه در آرایه چقدر است؟

پ-۳-۳ یک بازی عمومی شامل سه تاس در یک جعبه است. یک بازی‌کن می‌تواند یک دلار بر روی هر یک از اعداد ۱ تا ۶ شرط‌بندی کند. جعبه تکان داده می‌شود، و پرداخت بدین صورت خواهد بود. اگر عدد بازی‌کن بر روی هیچ یک از تاس‌ها نیامده باشد، بازی‌کن یک دلار خود را می‌بازد. در غیر این صورت اگر این عدد بر روی k تا از سه تاس ظاهر شود، برای $k = 1,2,3$، او یک دلار خود را پس گرفته و k دلار دیگر هم می‌برد. امیدریاضی پول دریافتی یک بازی‌کن برای یک بار بازی کردن چقدر است؟

پ-۳-۴ بحث کنید که اگر X و Y متغیرهای تصادفی نامنفی باشند، آن گاه

$$E[\max(X, Y)] \le E[X] + E[Y]$$

★پ-۳-۵ فرض کنید X و Y متغیرهای تصادفی مستقل باشند. اثبات کنید که $f(X)$ و $g(Y)$ برای هر انتخابی از توابع f و g مستقل هستند.

★پ-۳-۶ فرض کنید X یک متغیر تصادفی نامنفی باشد، و فرض کنید $E[X]$ خوش‌تعریف باشد. *نامساوی مارکوف* (Markov's inequality) را اثبات کنید:

$$\Pr\{X \ge t\} \le E[X]/t \qquad (پ-۳۰)$$

برای هر $t > 0$.

★پ-۳-۷ فرض کنید S یک فضای نمونه باشد، و X و X' متغیرهای تصادفی باشند به طوری که $X(s) \ge X'(s)$ برای هر $s \in S$. اثبات کنید که برای هر ثابت حقیقی t،

$$\Pr\{X \ge t\} \ge \Pr\{X' \ge t\}$$

پ-۳-۸ کدام یک بزرگ‌تر است: امیدریاضی مربع یک متغیر تصادفی، و یا مربع امیدریاضی آن؟

پ-۳-۹ نشان دهید که برای هر متغیر تصادفی X که فقط مقادیر ۰ و ۱ را می‌گیرد، داریم $\mathrm{Var}[X] = E[X]E[1-X]$.

پ–۳–۱۰ از تعریف واریانس (پ–۲۷) اثبات کنید که $\mathrm{Var}[aX] = a^2\mathrm{Var}[X]$.

پ–۴ توزیع‌های هندسی و دوجمله‌ای

پرتاب یک سکه، نمونه‌ای است از یک *آزمایش برنولی* (Bernoulli trial)، که به صورت آزمایشی تعریف می‌شود که فقط دو نتیجه‌ی ممکن داشته باشد: *موفقیت*، که با احتمال p رخ می‌دهد، و *شکست*، که با احتمال $q = 1 - p$ رخ می‌دهد. وقتی از *آزمایش‌های برنولی* به صورت گروهی صحبت می‌کنیم، منظور این است که آزمایش‌ها متقابلا مستقل هستند، مگر این که خلاف آن گفته شود، و این که تمام آزمایش‌ها با احتمال p به موفقیت ختم می‌شوند. دو توزیع مهم برای آزمایش‌های برنولی وجود دارد: توزیع هندسی و توزیع دوجمله‌ای.

توزیع هندسی

فرض کنید که دنباله‌ای از آزمایش‌های برنولی داریم، هر یک با احتمال p برای موفقت و $q = 1 - p$ برای شکست. چند آزمایش باید انجام دهیم تا به یک موفقیت برسیم؟ فرض کنید متغیر تصادفی X نشان‌دهنده‌ی تعداد آزمایش‌های مورد نیاز برای به دست آوردن یک موفقیت باشد. در این صورت X مقادیری خواهد گرفت در دامنه‌ی $\{1, 2, \ldots\}$، و برای $k \geq 1$،

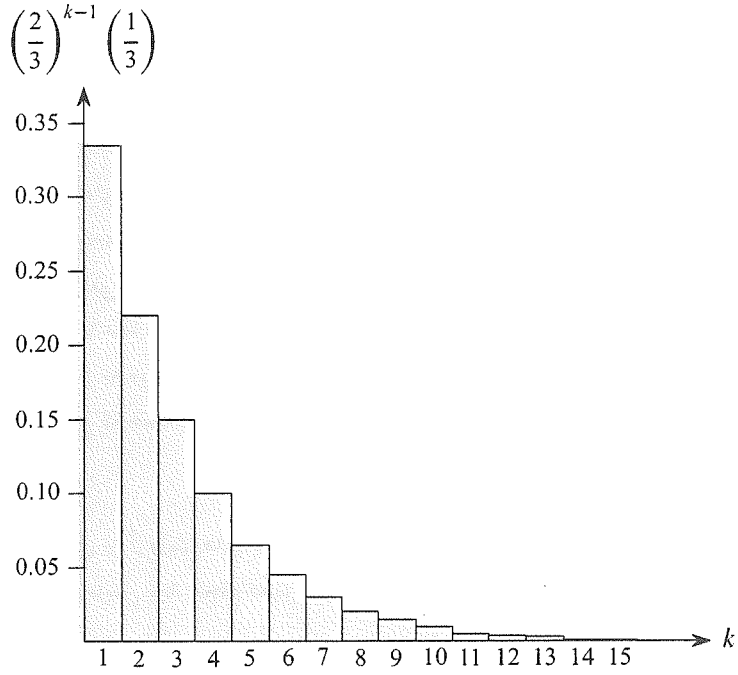
$$\Pr\{X = k\} = q^{k-1}p \qquad \text{(پ–۳۱)}$$

چرا که قبل از رسیدن به موفقت، $k - 1$ شکست خواهیم داشت. یک توزیع احتمالاتی که در رابطه‌ی (پ–۳۰) صدق می‌کند، *توزیع هندسی* (geometric distribution) نام دارد. شکل پ–۱ چنین توزیعی را نشان می‌دهد.

با فرض این که $q < 1$، امیدریاضی توزیع هندسی را می‌توان با استفاده از اتحاد (الف–۸) محاسبه کرد:

$$\begin{aligned} E[X] &= \sum_{k=1}^{\infty} kq^{k-1}p \\ &= \frac{p}{q}\sum_{k=0}^{\infty} kq^k \\ &= \frac{p}{q}\cdot\frac{q}{(1-q)^2} \\ &= \frac{p}{q}\cdot\frac{q}{p^2} \\ &= 1/p \end{aligned} \qquad \text{(پ–۳۲)}$$

بنابراین در حالت متوسط، $1/p$ آزمایش قبل از به دست آوردن یک موفقیت خواهیم داشت، که یک نتیجه‌ی شهودی است. واریانس را می‌توان به طور مشابه محاسبه کرد، ولی با استفاده از تمرین الف–۱–۳ داریم



شکل پ-۱ یک توزیع هندسی با احتمال $p = ۱/۳$ برای موفقیت و احتمال $q = ۱ - p$ برای شکست. امیدریاضی این توزیع برابر است با $۱/p = ۳$.

$$\mathrm{Var}[X] = q/p^{۲} \qquad (پ-۳۳)$$

به عنوان یک مثال، فرض کنید که مکرراً دو تاس را پرتاب می‌کنیم تا یا به یک هفت برسیم و یا به یک یازده. از ۳۶ نتیجه‌ی ممکن، ۶ تای آن‌ها به هفت و ۲ تای آن‌ها به یازده منجر می‌شوند. بنابراین احتمال موفقیت برابر است با $p = ۸/۳۶ = ۲/۹$، و به طور متوسط $۱/p = ۹/۲ = ۴/۵$ پرتاب برای رسیدن به یک هفت و یا یک یازده خواهیم داشت.

توزیع دوجمله‌ای

چند موفقیت در n آزمایش برنولی خواهیم داشت، که در آن موفقیت با احتمال p و شکست با احتمال $q = ۱ - p$ به دست می‌آید؟ متغیر تصادفی X را به صورت تعداد موفقیت‌ها در n آزمایش تعریف می‌کنیم. در این صورت مقادیر X در دامنه‌ی $\{۰,۱,...,n\}$ خواهد بود، و برای $k = ۰,۱,...,n$،

$$\Pr\{X = k\} = \binom{n}{k} p^k q^{n-k} \qquad (پ-۳۴)$$

چرا که $\binom{n}{k}$ راه برای انتخاب k موفقیت از n آزمایش وجود دارد، و احتمال رخداد هر یک برابر

است با $p^k q^{n-k}$. به یک توزیع احتمالاتی که در تساوی (پ-۳۴) صدق می‌کند، *توزیع دوجمله‌ای* (binomial distribution) گفته می‌شود. برای سادگی، خانواده‌ی توزیع‌های دوجمله‌ای را با استفاده از نماد زیر تعریف خواهیم کرد:

$$b(k;n,p)=\binom{n}{k}p^k(1-p)^{n-k} \qquad (پ-۳۵)$$

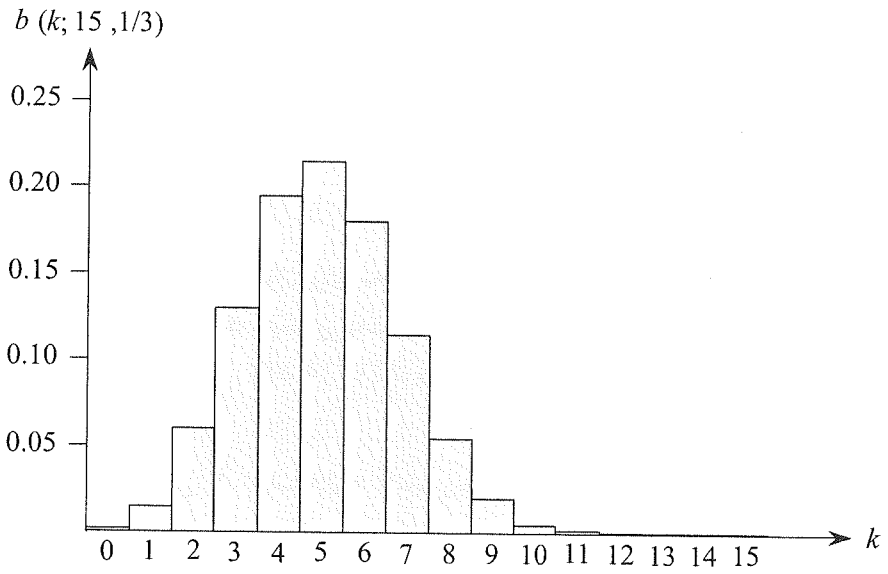
شکل پ-۲ یک توزیع دوجمله‌ای را نشان می‌دهد. نام «دوجمله‌ای» از این جا آمده است که سمت راست تساوی (پ-۳۴)، kامین جمله‌ی بسط $(p+q)^n$ است. در نتیجه، چون $p+q=1$ داریم

$$\sum_{k=0}^{n} b(k;n,p)=1 \qquad (پ-۳۶)$$

همان طور که در اصل ۲ اصول احتمالاتی گفته شده است.

می‌توانیم امیدریاضی یک متغیر تصادفی که توزیع دوجمله‌ای دارد را از تساوی‌های (پ-۸) و (پ-۳۵) محاسبه کنیم. فرض کنید X یک متغیر تصادفی باشد که از توزیع دوجمله‌ای $b(k;n,p)$ پیروی می‌کند، و فرض کنید $q=1-p$. طبق تعریف امیدریاضی، داریم

$$E[X]=\sum_{k=0}^{n} k\,\Pr\{X=k\}$$



شکل پ-۲ توزیع دوجمله‌ای $b(k;15,1/3)$ در نتیجه‌ی $n=15$ آزمایش برنولی، که هر یک با احتمال $p=1/3$ به موفقیت ختم می‌شوند. امیدریاضی توزیع برابر است با $np=5$.

$$
\begin{aligned}
&= \sum_{k=0}^{n} k b(k;n,p) \\
&= \sum_{k=1}^{n} k \binom{n}{k} p^k q^{n-k} \\
&= np \sum_{k=1}^{n} \binom{n-1}{k-1} p^{k-1} q^{n-k} \qquad \text{(طبق تساوی (پ-۸))} \\
&= np \sum_{k=0}^{n-1} \binom{n-1}{k} p^k q^{(n-1)-k} \\
&= np \sum_{k=0}^{n-1} b(k;n-1,p) \\
&= np \qquad \text{(طبق تساوی (پ-۳۶))}
\end{aligned}
\tag{پ-۳۷}
$$

با استفاده از خطی بودن امیدریاضی، می‌توانیم با جبر بسیار کم‌تر به نتیجه‌ی یکسانی برسیم. فرض کنید X_i متغیر تصادفی توصیف‌کننده‌ی تعداد موفقیت‌ها در iامین آزمایش باشد. در این صورت $E[X_i] = p \cdot 1 + q \cdot 0 = p$، و طبق خطی بودن امیدریاضی (تساوی (پ-۲۰))، امیدریاضی تعداد موفقیت‌ها در n آزمایش برابر است با

$$
\begin{aligned}
E[X] &= E\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} E[X_i] \\
&= \sum_{i=1}^{n} p \\
&= np
\end{aligned}
\tag{پ-۳۸}
$$

می‌توان از رویکرد یکسانی می‌توان برای محاسبه‌ی واریانس توزیع استفاده کرد. با استفاده از تساوی (پ-۲۷) داریم $X_i^2 = X_i$ که نتیجه می‌دهد $\mathrm{Var}[X_i] = E[X_i^2] - E^2[X_i]$. چون X_i فقط مقادیر ۰ و ۱ را می‌گیرد، داریم $E[X_i^2] = E[X_i] = p$، و بنابراین

$$
\mathrm{Var}[X_i] = p - p^2 = p(1-p) = pq \tag{پ-۳۹}
$$

برای محاسبه‌ی واریانس X، از استقلال n آزمایش بهره می‌گیریم؛ بنابراین، طبق تساوی (پ-۲۹)،

$$
\begin{aligned}
\mathrm{Var}[X] &= \mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{Var}[X_i]
\end{aligned}
$$

$$= \sum_{i=1}^{n} pq \qquad (پ-۴۰)$$
$$= n\,p\,q$$

همان طور که می‌توان در شکل پ-۲ دید، توزیع دوجمله‌ای $b(k\,;n,\,p)$ با حرکت k از ۰ به سمت n افزایش می‌یابد، تا زمانی که به میانگین np می‌رسد، و سپس کاهش می‌یابد. با نگاهی به جمله‌های متوالی می‌توانیم اثبات کنیم که این توزیع همیشه به همین صورت رفتار می‌کند:

$$\frac{b(k;n,p)}{b(k-1;n,p)} = \frac{\binom{n}{k} p^k q^{n-k}}{\binom{n}{k-1} p^{k-1} q^{n-k+1}}$$
$$= \frac{n!(k-1)!(n-k+1)!p}{k!(n-k)!n!q} \qquad (پ-۴۱)$$
$$= \frac{(n-k+1)p}{kq}$$
$$= 1 + \frac{(n+1)p-k}{kq}$$

این نسبت زمانی بزرگ‌تر از ۱ است که $(n+1)p-k$ مثبت باشد. در نتیجه $b(k;n,p) > b(k-1;n,p)$ برای $k < (n+1)p$ (توزیع افزایش می‌یابد)، و $b(k;n,p) < b(k-1;n,p)$ برای $k > (n+1)p$ (توزیع کاهش می‌یابد). اگر $k = (n+1)p$ یک عدد صحیح باشد، آن گاه $b(k;n,p) = b(k-1;n,p)$، و توزیع دو مقدار بیشینه‌ی محلی دارد: در $k = (n+1)p$ و $k-1 = (n+1)p-1 = np-q$. در غیر این صورت یک بیشینه در عدد صحیح یکتای k دارد که در دامنه‌ی $np-q < k < (n+1)p$ قرار می‌گیرد.

لم زیر یک کران بالا برای توزیع دوجمله‌ای فراهم می‌کند.

لم پ-۱ فرض کنید $n \geq 0$، $0 < p < 1$، $q = 1-p$، و $0 \leq k \leq n$. آن گاه

$$b(k;n,p) \leq \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}$$

اثبات با استفاده از تساوی (پ-۶) داریم

$$b(k;n,p) = \binom{n}{k} p^k q^{n-k}$$
$$\leq \left(\frac{n}{k}\right)^k \left(\frac{n}{n-k}\right)^{n-k} p^k q^{n-k}$$
$$= \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}$$

■

تمرین‌ها

پ-۴-۱ درستی اصل ۲ از اصول احتمالات را برای توزیع هندسی تحقیق کنید.

پ-۴-۲ در حالت متوسط چند بار باید ۶ سکه‌ی متقارن را پرتاب کنیم قبل از این که ۳ شیر و ۳ خط به دست آوریم؟

پ-۴-۳ نشان دهید که $b(k;n,p)=b(n-k;n,q)$، که در آن $q=1-p$.

پ-۴-۴ نشان دهید که مقدار بیشینه‌ی توزیع دوجمله‌ای $b(k;n,p)$ تقریباً برابر است با $1/\sqrt{2\pi n p q}$، که در آن $q=1-p$.

★پ-۴-۵ نشان دهید که احتمال عدم وقوع موفقیت در n آزمایش برنولی، هر یک با احتمال $p=1/n$، تقریباً برابر است با $1/e$. نشان دهید که احتمال دقیقاً یک موفقیت هم تقریباً برابر است با $1/e$.

★پ-۴-۶ پروفسور Rosencrantz یک سکه‌ی متقارن را n بار پرتاب می‌کند، و پروفسور Guildenstern هم همین کار را می‌کند. نشان دهید که احتمال این که تعداد شیرهایی که هر دو به دست می‌آورند برابر باشد، برابر است با $\binom{2n}{n}\Big/4^n$. (راهنمایی: برای پروفسور Rosencrantz، یک شیر را یک موفقیت بنامید؛ برای پروفسور Guildenstern یک خط را یک موفقیت بنامید.) از نتیجه‌ی بحث خود برای تحقیق درستی اتحاد زیر استفاده کنید.

$$\sum_{k=0}^{n}\binom{n}{k}^2=\binom{2n}{n}$$

★پ-۴-۷ نشان دهید که برای $0\le k\le n$،

$$b(k;n,1/2)\le 2^{nH(k/n)-n}$$

که در آن $H(x)$ تابع آنتروپی (پ-۷) است.

★پ-۴-۸ n آزمایش برنولی را در نظر بگیرید، که در آن برای $i=1,2,\ldots,n$، احتمال موفقیت در iامین آزمایش p_i است، و فرض کنید X متغیر تصادفی نشان دهنده‌ی تعداد کل موفقیت‌ها باشد. اگر $p\ge p_i$ برای تمام $i=1,2,\ldots,n$، برای $1\le k\le n$ اثبات کنید که

$$\Pr\{X<k\}\le\sum_{i=0}^{k-1}b(i;n,p)$$

پ-۴-۹ فرض کنید X متغیر تصادفی تعداد کل موفقیت‌ها در مجموعه‌ی A از n آزمایش برنولی باشد، که در آن احتمال موفقیت در iامین آزمایش p_i است، و فرض کنید X' متغیر

تصادفی تعداد کل موفقیت‌ها در یک مجموعه‌ی دوم A' از n آزمایش برنولی باشد، که در آن احتمال موفقیت در iامین آزمایش $p_i' \geq p_i$ است. اثبات کنید که برای $0 \leq k \leq n$،

$$\Pr\{X' \geq k\} \geq \Pr\{X \geq k\}$$

(*راهنمایی:* نشان دهید که چگونه می‌توان با یک آزمایش مربوط شامل آزمایش‌های A به آزمایش‌های برنولی در A' رسید، و از نتیجه‌ی تمرین پ-۳-۷ استفاده کنید.)

★پ-۵ نقاط پایانی در توزیع دوجمله‌ای

معمولاً احتمال به دست آوردن حداقل، یا حداکثر، k موفقیت در n آزمایش برنولی، هر یک با احتمال موفقیت p، بیشتر مورد توجه است تا احتمال به دست آوردن دقیقاً k موفقیت. در این بخش دو *انتهای* (tail) توزیع دوجمله‌ای را بررسی می‌کنیم: دو منطقه از توزیع $b(k;n,p)$ که در دورترین نقطه از میانگین np قرار دارند. کران‌های مهم متعددی را بر روی (مجموع تمام عبارت‌ها در) یک انتها اثبات خواهیم کرد.

ابتدا یک کران برای انتهای سمت راست توزیع $b(k;n,p)$ ارائه خواهیم کرد. کران‌های انتهای سمت چپ را می‌توان با تعویض نقش موفقیت‌ها و شکست‌ها به دست آورد.

قضیه‌ی پ-۳

یک دنباله از n آزمایش برنولی را در نظر بگیرید، که در آن موفقیت با احتمال p رخ می‌دهد. فرض کنید X متغیر تصادفی نشان دهنده‌ی تعداد کل موفقیت‌ها باشد. آن گاه برای $0 \leq k \leq n$ احتمال وقوع حداقل k موفقیت برابر است با

$$\begin{aligned}\Pr\{X \geq k\} &= \sum_{i=k}^{n} b(i;n,p) \\ &\leq \binom{n}{k} p^k\end{aligned}$$

اثبات برای $S \subseteq \{1,2,\ldots,n\}$ فرض می‌کنیم A_S نشان‌دهنده‌ی این پیشامد باشد که iامین آزمایش موفق باشد، برای هر $i \in S$. به وضوح $\Pr\{A_S\} = p^k$ اگر $|S| = k$. داریم

$$\begin{aligned}\Pr\{X \geq k\} &= \Pr\{A_S \text{ و } |S| = k \text{ وجود دارد به طوری که } S \subseteq \{1, 2, \ldots, n\} \text{ (مجموعه‌ی }\} \\ &= \Pr\left\{\bigcup_{S \subseteq \{1, 2, \ldots, n\}: |S| = k} A_S\right\} \\ &\leq \sum_{S \subseteq \{1, 2, \ldots, n\}: |S| = k} \Pr\{A_S\} \qquad \text{(طبق نامساوی (پ-۱۹))} \\ &= \binom{n}{k} p^k\end{aligned}$$

■

نتیجه‌ی زیر، قضیه را برای انتهای سمت چپ توزیع دوجمله‌ای بازگو می‌کند. به طور کلی تبدیل اثبات یک انتها به یک اثبات برای انتهای دیگر را به خواننده واگذار می‌کنیم.

نتیجه‌ی پ-۳

دنباله‌ای از n آزمایش برنولی را در نظر بگیرید، که در آن موفقیت با احتمال p رخ می‌دهد. اگر X متغیر تصادفی نشان‌دهنده‌ی تعداد کل موفقیت‌ها باشد، آن گاه برای $0 \le k \le n$، احتمال به دست آوردن حداکثر k موفقیت برابر است با

$$\begin{aligned}\Pr\{X \le k\} &= \sum_{i=0}^{k} b(i;n,p) \\ &\le \binom{n}{n-k}(1-p)^{n-k} \\ &= \binom{n}{k}(1-p)^{n-k}\end{aligned}$$

کران بعدی ما به انتهای سمت چپ توزیع دوجمله‌ای مربوط می‌شود. نتیجه‌ی آن نشان می‌دهد که، در نقاط بسیار دور از میانگین، انتهای سمت چپ به صورت نمایی تحلیل می‌رود.

لم پ-۴

دنباله‌ای از n آزمایش برنولی را در نظر بگیرید، که در آن موفقیت با احتمال p و شکست با احتمال $q = 1-p$ رخ می‌دهد. فرض کنید X متغیر تصادفی نشان دهنده‌ی تعداد کل موفقیت‌ها باشد. در این صورت برای $0 < k < np$، احتمال به دست آوردن کم‌تر از k موفقیت برابر است با

$$\begin{aligned}\Pr\{X < k\} &= \sum_{i=0}^{k-1} b(i;n,p) \\ &< \frac{kq}{np-k} b(k;n,p)\end{aligned}$$

اثبات با استفاده از تکنیک‌های بخش الف-۲، سری $\sum_{i=0}^{k-1} b(i;n,p)$ را توسط یک سری هندسی محدود می‌کنیم. برای $i = 1,2,\ldots,k$ از تساوی (پ-۴۱) داریم

$$\begin{aligned}\frac{b(i-1;n,p)}{b(i;n,p)} &= \frac{iq}{(n-i+1)p} \\ &< \frac{iq}{(n-i)p} \\ &\le \frac{kq}{(n-k)p}\end{aligned}$$

اگر قرار دهیم

$$
\begin{aligned}
x &= \frac{kq}{(n-k)p} \\
&< \frac{kq}{(n-np)p} \\
&= \frac{kq}{nqp} \\
&= \frac{k}{np} \\
&< ۱
\end{aligned}
$$

که نتیجه می‌دهد

$$b(i-۱;n,p) < xb(i;n,p)$$

برای $۰ < i \le k$. با اعمال مکرر این نامساوی به تعداد $k-i$ بار، به دست می‌آوریم

$$b(i;n,p) < x_{k-i}b(k;n,p)$$

برای $۰ \le i < k$، و بنابراین

$$
\begin{aligned}
\sum_{i=۰}^{k-۱} b(i;n,p) &< \sum_{i=۰}^{k-۱} x^{k-i} b(k;n,p) \\
&< b(k;n,p)\sum_{i=۱}^{\infty} x^{i} \\
&= \frac{x}{۱-x} b(k;n,p) \\
&= \frac{kq}{np-k} b(k;n,p)
\end{aligned}
$$

نتیجه‌ی پ-۵ دنباله‌ای از n آزمایش برنولی را در نظر بگیرید، که در آن موفقیت با احتمال p و شکست با احتمال $q = ۱-p$ رخ می‌دهد. در این صورت برای $۰ < k \le np/۲$، احتمال این که کم‌تر از k موفقیت به دست آوریم کم‌تر است از نصف احتمال این که کم‌تر از $k+۱$ موفقیت به دست آوریم.

اثبات چون $k \le np/۲$، داریم

$$
\begin{aligned}
\frac{kq}{np-k} &\le \frac{(np/۲)q}{np-(np/۲)} \\
&= \frac{(np/۲)q}{np/۲} \\
&\le ۱
\end{aligned}
\qquad (پ-۴۲)
$$

چرا که $q \leq ۱$. با فرض این که X متغیر تصادفی نشان دهنده‌ی تعداد موفقیت‌ها باشد، قضیه‌ی پ-۴ و نامساوی (پ-۴۲) ایجاب می‌کند که احتمال وقوع کم‌تر از k موفقیت برابر است با

$$\Pr\{X < k\} \sum_{i=۰}^{k-۱} b(i;n,p) < b(k;n,p)$$

بنابراین داریم

$$\begin{aligned}\frac{\Pr\{X < k\}}{\Pr\{X < k+۱\}} &= \frac{\sum_{i=۰}^{k-۱} b(i;n,p)}{\sum_{i=۰}^{k-۱} b(i;n,p)} \\ &= \frac{\sum_{i=۰}^{k-۱} b(i;n,p)}{\sum_{i=۰}^{k-۱} b(i;n,p) + b(k;n,p)} \\ &\leq ۱/۲\end{aligned}$$

چرا که $\sum_{i=۰}^{k-۱} b(i;n,p) < b(k;n,p)$. ■

کران‌های انتهای سمت راست را هم می‌توان به صورت مشابه به دست آورد. اثبات آن‌ها به عنوان تمرین پ-۵-۲ واگذار شده است.

نتیجه‌ی پ-۶ دنباله‌ای از n آزمایش برنولی را در نظر بگیرید، که در آن موفقیت با احتمال p رخ می‌دهد. فرض کنید X متغیر تصادفی نشان دهنده‌ی تعداد موفقیت‌ها باشد. در این صورت برای $np < k < n$، احتمال وقوع بیش از k موفقیت برابر است با

$$\begin{aligned}\Pr\{X > k\} &= \sum_{i=k+۱}^{n} b(i;n,p) \\ &< \frac{(n-k)p}{k-np} b(k;n,p)\end{aligned}$$

■

نتیجه‌ی پ-۷ دنباله‌ای از n آزمایش برنولی را در نظر بگیرید، که در آن موفقیت با احتمال p و شکست با احتمال $q = ۱-p$ رخ می‌دهد. در این صورت برای $(np+n)/۲ < k < n$، احتمال وقوع بیش از k موفقیت کم‌تر است از نصف احتمال وقوع بیش از $k-۱$ موفقیت.

■

قضیه‌ی بعد n آزمایش برنولی را در نظر می‌گیرد، هر یک با احتمال موفقیت p_i برای $i = ۱,۲,..,n$. همان طور که نتیجه‌ی بعد از آن نشان می‌دهد، می‌توانیم از این قضیه برای تعیین یک کران بر روی انتهای سمت راست توزیع دوجمله‌ای با قرار دادن $p_i = p$ برای هر آزمایش استفاده کنیم.

قضیه‌ی پ-۸

دنباله‌ای از n آزمایش برنولی را در نظر بگیرید، که در آن در آزمایش iام، برای $i = ۱,۲,...,n$ موفقیت با احتمال p_i و شکست با احتمال $q_i = ۱ - p_i$ رخ می‌دهد. فرض کنید X متغیر تصادفی نشان دهنده‌ی تعداد کل موفقیت‌ها باشد، و فرض کنید $\mu = E[X]$. آن گاه برای $r > \mu$،

$$\Pr\{X - \mu \geq r\} \leq \left(\frac{\mu e}{r}\right)^r$$

اثبات از آن جایی که برای هر $\alpha > ۰$، تابع $e^{\alpha x}$ نسبت به x اکیداً صعودی است،

$$\Pr\{X - \mu \geq r\} = \Pr\{e^{\alpha(X-\mu)} \geq e^{\alpha r}\} \qquad (پ-۴۳)$$

که در آن مقدار α بعداً مشخص می‌شود. با استفاده از نامساوی مارکوف (پ-۳۰)، به دست می‌آوریم

$$\Pr\{e^{\alpha(X-\mu)} \geq e^{\alpha r}\} \leq E[e^{\alpha(X-\mu)}]e^{-\alpha r} \qquad (پ-۴۴)$$

قسمت اعظم اثبات شامل تعیین کران برای $E[e^{\alpha(X-\mu)}]$ و جایگزینی یک مقدار مناسب برای α در نامساوی (پ-۴۴) می‌شود. ابتدا $E[e^{\alpha(X-\mu)}]$ را ارزیابی می‌کنیم. با استفاده از نمادگذاری بخش ۵-۲، فرض کنید

$X_i = I\{i$امین آزمایش برنولی موفقیت آمیز باشد$\}$

برای $i = ۱, ۲, ..., n$؛ یعنی X_i یک متغیر تصادفی است که مقدار آن برابر ۱ است اگر نتیجه‌ی iامین آزمایش موفقیت باشد، و ۰ اگر نتیجه‌ی iامین آزمایش شکست باشد. بنابراین

$$X = \sum_{i=۱}^{n} X_i$$

و طبق خطی بودن امیدریاضی،

$$\mu = E[X] = E\left[\sum_{i=۱}^{n} X_i\right] = \sum_{i=۱}^{n} E[X_i] = \sum_{i=۱}^{n} p_i$$

که نتیجه می‌دهد

$$X - \mu = \sum_{i=۱}^{n} (X_i - p_i)$$

برای ارزیابی $E[e^{\alpha(X-\mu)}]$، $X - \mu$ را جایگزین می‌کنیم، که به دست می‌دهد

$$\begin{aligned} E[e^{\alpha(X-\mu)}] &= E\left[e^{\alpha\sum_{i=۱}^{n}(X_i - p_i)}\right] \\ &= E\left[\prod_{i=۱}^{n} e^{\alpha(X_i - p_i)}\right] \\ &= \prod_{i=۱}^{n} E[e^{\alpha(X_i - p_i)}] \end{aligned}$$

که از (پ-۲۴) نتیجه می‌شود، چرا که استقلال متقابل متغیرهای تصادفی X_i، استقلال متقابل متغیرهای تصادفی $e^{\alpha(X_i - p_i)}$ را نتیجه می‌دهد (تمرین پ-۳-۵ را ببینید). طبق تعریف امیدریاضی،

$$\begin{aligned} E[e^{\alpha(X_i - p_i)}] &= e^{\alpha(1-p_i)}p_i + e^{\alpha(0-p_i)}q_i \\ &= p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \\ &\le p_i e^{\alpha} + 1 \\ &\le \exp(p_i e^{\alpha}) \end{aligned} \qquad \text{(پ-۴۵)}$$

که در آن $\exp(x)$ نشان‌دهنده‌ی تابع نمایی است: $\exp(x) = e^x$. (نامساوی (پ-۴۵) از نامساوی‌های $\alpha > 0$، $q_i \le 1$، $e^{\alpha q_i} \le e^{\alpha}$، و $e^{\alpha p_i} \le 1$ نتیجه می‌شود، و خط آخر از نامساوی (۳-۱۲)). در نتیجه

$$\begin{aligned} E[e^{\alpha(X - \mu)}] &= \prod_{i=1}^{n} E[e^{\alpha(X_i - p_i)}] \\ &\le \prod_{i=1}^{n} \exp(p_i e^{\alpha}) \\ &= \exp\left(\sum_{i=1}^{n} p_i e^{\alpha}\right) \\ &= \exp(\mu e^{\alpha}) \end{aligned} \qquad \text{(پ-۴۶)}$$

چرا که $\mu = \sum_{i=1}^{n} p_i$. بنابراین از تساوی (پ-۴۳) و نامساوی‌های (پ-۴۴) و (پ-۴۶)، نتیجه می‌شود

$$\Pr\{X - \mu \ge r\} \le \exp(\mu e^{\alpha} - \alpha r) \qquad \text{(پ-۴۷)}$$

با انتخاب $\alpha = \ln(r/\mu)$ (تمرین پ-۵-۷ را ببینید)، به دست می‌آوریم

$$\begin{aligned} \Pr\{X - \mu \ge r\} &\le \exp(\mu e^{\ln(r/\mu)} - r\ln(r/\mu)) \\ &= \exp(r - r\ln(r/\mu)) \\ &= \frac{e^r}{(r/\mu)^r} \\ &= \left(\frac{\mu e}{r}\right)^r \end{aligned}$$

■

با به اعمال قضیه‌ی پ-۸ به آزمایش‌های برنولی که در آن‌ها هر آزمایش یک احتمال موفقیت مربوط به خود را دارد، نتیجه‌ی زیر برای تعیین کران انتهای سمت راست توزیع دوجمله‌ای حاصل می‌شود.

نتیجه‌ی پ-۹ دنباله‌ای از n آزمایش برنولی را در نظر بگیرید، که در آن موفقت با احتمال p و شکست با احتمال $q = 1 - p$ رخ می‌دهد. در این صورت برای $r > np$،

$$\Pr\{X - np \geq r\} = \sum_{k=\lceil np+r \rceil}^{n} b(k;n,p) \leq \left(\frac{npe}{r}\right)^r$$

اثبات طبق تساوی (پ–۳۷)، داریم $\mu = E[X] = np$.

تمرین‌ها

★ پ–۵–۱ احتمال وقوع کدام یک کم‌تر است: عدم به دست آوردن شیر در n پرتاب یک سکه‌ی متقارن، یا به دست آوردن کم‌تر از n شیر در $4n$ پرتاب یک سکه؟

★ پ–۵–۲ نتیجه‌های پ–۶ و پ–۷ را اثبات کنید.

★ پ–۵–۳ نشان دهید که

$$\sum_{i=0}^{k-1} \binom{n}{i} a^i < (a+1)^n \frac{k}{na-k(a+1)} b(k;n,a/(a+1))$$

برای هر $a > 0$ و هر k به طوری که $0 < k < na/(a+1)$.

★ پ–۵–۴ اثبات کنید که اگر $0 < k < np$، که در آن $0 < p < 1$ و $q = 1-p$، آن گاه

$$\sum_{i=0}^{k-1} p^i q^{n-i} < \frac{kq}{np-k}\left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}$$

★ پ–۵–۵ نشان دهید که شرایط قضیه‌ی پ–۸ ایجاب می‌کند که

$$\Pr\{\mu - X \geq r\} \leq \left(\frac{(n-\mu)e}{r}\right)^r$$

به طور مشابه، نشان دهید که شرایط نتیجه‌ی پ–۹ ایجاب می‌کند که

$$\Pr\{np - X \geq r\} \leq \left(\frac{nqe}{r}\right)^r$$

★ پ–۵–۶ دنباله‌ای از n آزمایش برنولی را در نظر بگیرید، که در آن در iامین آزمایش، برای $i = 1,2,\ldots,n$، موفقیت با احتمال p_i و شکست با احتمال $q_i = 1-p_i$ رخ می‌دهد. فرض کنید X متغیر تصادفی نشان‌دهنده‌ی تعداد کل موفقیت‌ها باشد، و فرض کنید $\mu = E[X]$. برای $r \geq 0$ نشان دهید که

$$\Pr\{X - \mu \geq r\} \leq e^{-r^2/2n}$$

(*راهنمایی:* اثبات کنید که $p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \le e^{\alpha^2/2}$. سپس خط مشی اثبات قضیه‌ی پ-۸ را دنبال کنید، با جایگزینی این نامساوی به جای نامساوی (پ-۴۵).)

★ پ-۵-۷ نشان دهید که سمت راست نامساوی (پ-۴۷) با انتخاب $\alpha = \ln(r/\mu)$ کمینه می‌شود.

مسائل

پ-۱ *توپ‌ها و سبدها*

در این مسئله تأثیر فرض‌های مختلف را بر روی تعداد روش‌های قرار دادن n توپ در b سبد متفاوت بررسی می‌کنیم.

I. فرض کنید که n توپ متفاوت هستند، و ترتیب آن‌ها در یک سبد اهمیتی ندارد. بحث کنید که تعداد روش‌های قرار دادن توپ‌ها در سبدها برابر است با b^n.

II. فرض کنید که توپ‌ها متفاوت هستند، و توپ‌های درون هر سبد، مرتب شده است. اثبات کنید که دقیقاً $(b+n-1)!/(b-1)!$ روش برای قرار دادن توپ‌ها در سبدها وجود دارد. (*راهنمایی:* تعداد روش‌های قرار دادن n توپ متفاوت و $b-1$ تکه چوب قابل تمیز را در یک ردیف در نظر بگیرید.)

III. فرض کنید که توپ‌ها یکسان هستند، و بنابراین ترتیب آن‌ها در یک سبد هم اهمیتی ندارد. نشان دهید که تعداد روش‌های قرار دادن توپ‌ها در سبدها برابر است با $\binom{b+n-1}{n}$. (*راهنمایی:* از میان ترتیب‌های بخش II، اگر توپ‌ها یکسان در نظر گرفته شوند، چند تا از آن‌ها تکراری می‌شوند؟)

IV. فرض کنید که توپ‌ها یکسان هستند و این که هیچ یک از سبدها نمی‌تواند حاوی بیش از یک توپ باشد، و بنابراین $n \le b$. نشان دهید که تعداد روش‌های قرار دادن توپ‌ها در سبدها برابر است با $\binom{b}{n}$.

V. فرض کنید که توپ‌ها یکسان هستند و هیچ سبدی نباید تهی باشد. با فرض این که $n \ge b$ نشان دهید که تعداد روش‌های قرار دادن توپ‌ها در سبدها برابر است با $\binom{n-1}{b-1}$.

پیوست

ت

ماتریس‌ها

ت-۰ مقدمه

ماتریس‌ها کاربردهای بسیاری در محاسبات علمی و زمینه‌های بسیار دیگری دارند. اگر از قبل با ماتریس‌ها آشنا باشید، اکثر مطالب این پیوست برایتان تکراری خواهد بود، ولی ممکن است مطالب جدیدی هم در آن بیابید. بخش ت-۱ تعریف‌ها و اعمال اولیه‌ی ماتریس‌ها را پوشش می‌دهد، و بخش ت-۲ برخی خصوصیات اساسی آن‌ها را ارائه می‌کند.

ت-۱ خصوصیات ماتریس‌ها

در این بخش بعضی از مفاهیم اصلی نظریه‌ی ماتریس‌ها و بعضی از خصوصیات پایه‌ای ماتریس‌ها را بررسی می‌کنیم.

ماتریس‌ها و بردارها

یک *ماتریس* (matrix) یک آرایه‌ی مستطیلی از اعداد است. به عنوان مثال

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} = \begin{pmatrix} ۱ & ۲ & ۳ \\ ۴ & ۵ & ۶ \end{pmatrix} \qquad (ت-۱)$$

یک ماتریس $A = (a_{ij})$ با اندازه‌ی ۳×۲ است، که در آن برای $i = ۱, ۲$ و $j = ۱, ۲, ۳$، عنصر ماتریس در سطر i و ستون j را با a_{ij} نشان می‌دهیم. از حروف بزرگ برای نشان دادن ماتریس‌ها و حروف کوچک متناظر برای نشان دادن عناصر همان ماتریس‌ها استفاده می‌کنیم. مجموعه‌ی تمام ماتریس‌های $m \times n$ با ورودی‌های حقیقی با $\mathbb{R}^{m \times n}$ نشان داده می‌شود. به طور کلی، مجموعه‌ی تمام ماتریس‌های $m \times n$ که ورودی‌های آن‌ها از مجموعه‌ی S انتخاب شده است، به صورت $S^{m \times n}$ نشان داده می‌شود.

ترانهاده‌ی (transpose) یک ماتریس A، ماتریس A^T است که از تعویض سطرها و ستون‌های A به دست می‌آید. برای ماتریس A از تساوی (ت-۱) داریم

$$A^T = \begin{pmatrix} ۱ & ۴ \\ ۲ & ۵ \\ ۳ & ۶ \end{pmatrix}$$

یک *بردار* (vector) یک آرایه‌ی یک بعدی از اعداد است. مثلاً

$$x = \begin{pmatrix} ۲ \\ ۳ \\ ۵ \end{pmatrix}$$

یک بردار با اندازه‌ی ۳ است. بعضی مواقع یک بردار با طول n را *n-بردار* می‌نامیم. از حروف کوچک برای نشان دادن بردارها استفاده می‌کنیم، و iامین عنصر یک بردار x با اندازه‌ی n را با x_i نشان می‌دهیم ($i = ۱, ۲, \ldots, n$). شکل استاندارد یک بردار را *بردار ستونی* (column vector) در نظر می‌گیریم، که معادل است با یک ماتریس $n \times ۱$؛ *بردار سطری* (row vector) مربوطه با گرفتن ترانهاده به دست می‌آید:

$$x^T = (۲ \quad ۳ \quad ۵)$$

بردار یکه (unit vector) e_i یک بردار است که iامین عنصر آن ۱ و بقیه‌ی عناصر آن ۰ است. معمولاً اندازه‌ی یک بردار یکه از محتوای متن مشخص است.

یک *ماتریس صفر* (zero matrix) ماتریسی است که تمام ورودی‌های آن ۰ هستند. چنین ماتریسی معمولاً با ۰ نشان داده می‌شود، چرا که ابهام میان عدد ۰ و یک ماتریس از ۰ها اغلب از محتوای متن به سادگی قابل برطرف شدن است.

ماتریس‌های مربعی

ماتریس‌های $n \times n$ *مربعی* استفاده‌های زیادی دارند. حالت‌های خاص مختلفی از ماتریس‌های مربعی دارای اهمیت ویژه‌ای هستند:

۱. در یک *ماتریس قطری* (diagonal matrix) داریم $a_{ij} = ۰$ هرگاه $i \neq j$. چون تمام عناصر غیر قطر ۰ هستند، این ماتریس را می‌توان با لیست کردن عناصر روی قطر مشخص کرد:

$$\operatorname{diag}(a_{11}, a_{22}, ..., a_{nn}) = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}$$

۲. *ماتریس همانی* (identity matrix) $n \times n$، که آن را با I_n نشان می‌دهیم، یک ماتریس قطری است که تمام عناصر روی قطر آن ۱ هستند:

$$I_n = \operatorname{diag}(1, 1, ..., 1) = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

وقتی I ذکر می‌شود بدون این که اندیسی داشته باشد، اندازه‌ی آن را می‌توان از محتوای متن تشخیص داد. iامین ستون یک ماتریس همانی، بردار یکه‌ی e_i است.

۳. یک *ماتریس سه‌قطری* (tridiagonal matrix) T، ماتریسی است که در آن $t_{ij} = 0$ اگر $|i - j| > 1$. ورودی‌های غیر صفر فقط بر روی قطر اصلی، دقیقاً بالای قطر اصلی ($t_{i, i+1}$ برای $i = 1, 2, ..., n-1$) یا دقیقاً پایین قطر اصلی ($t_{i+1, i}$ برای $i = 1, 2, ..., n-1$) وجود دارند:

$$T = \begin{pmatrix} t_{11} & t_{12} & 0 & 0 & \cdots & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & \cdots & 0 & 0 & 0 \\ 0 & t_{32} & t_{33} & t_{34} & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & t_{n-2,n-2} & t_{n-2,n-1} & 0 \\ 0 & 0 & 0 & 0 & \cdots & t_{n-1,n-2} & t_{n-1,n-1} & t_{n-1,n} \\ 0 & 0 & 0 & 0 & \cdots & 0 & t_{n,n-1} & t_{nn} \end{pmatrix}$$

۴. یک *ماتریس بالا-مثلثی* (upper-triangular matrix) U ماتریسی است که برای آن داریم $u_{ij} = 0$ اگر $i > j$. تمام ورودی‌های زیر قطر اصلی صفر هستند:

$$U = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{pmatrix}$$

یک ماتریس بالا-مثلثی، *ماتریس بالا-مثلثی یکه* است اگر تمام عناصر روی قطر اصلی آن ۱ باشند.

۵. یک *ماتریس پایین- مثلثی* L (lower-triangular matrix) ماتریسی است که برای آن داریم $l_{ij} = ۰$ اگر $i < j$. تمام ورودی‌های بالای قطر اصلی صفر هستند.

$$L = \begin{pmatrix} l_{۱۱} & ۰ & \cdots & ۰ \\ l_{۲۱} & l_{۲۲} & \cdots & ۰ \\ \vdots & \vdots & \ddots & \vdots \\ l_{n۱} & l_{n۲} & \cdots & l_{nn} \end{pmatrix}$$

یک ماتریس پایین- مثلثی، *ماتریس پایین-مثلثی یکه* است اگر تمام عناصر روی قطر اصلی آن ۱ باشند.

یک *ماتریس جایگشت* P (permutation matrix) دقیقاً یک ۱ در هر سطر یا ستون دارد، و تمام عناصر دیگر آن ۰ است. یک مثال از یک ماتریس جایگشت به صورت زیر است:

$$P = \begin{pmatrix} ۰ & ۱ & ۰ & ۰ & ۰ \\ ۰ & ۰ & ۰ & ۱ & ۰ \\ ۱ & ۰ & ۰ & ۰ & ۰ \\ ۰ & ۰ & ۰ & ۰ & ۱ \\ ۰ & ۰ & ۱ & ۰ & ۰ \end{pmatrix}$$

به چنین ماتریسی یک ماتریس جایگشت گفته می‌شود چرا که ضرب یک ماتریس x در یک ماتریس جایگشت، اثر بازآرایی عناصر x را دارد. تمرین ت-۱-۴ خصوصیات دیگری از ماتریس‌های جایگشت را بیان می‌کند.

۶. یک *ماتریس متقارن* A (symmetric matrix) شرط $A = A^T$ را ارضا می‌کند. به عنوان مثال

$$\begin{pmatrix} ۱ & ۲ & ۳ \\ ۲ & ۶ & ۴ \\ ۳ & ۴ & ۵ \end{pmatrix}$$

یک ماتریس متقارن است.

اعمال اولیه‌ی ماتریس‌ها

عناصر یک ماتریس و یا یک بردار، اعداد گرفته شده از یک سیستم عددی هستند، مانند اعداد حقیقی، اعداد مختلط، و یا باقی‌مانده‌ی اعداد صحیح بر یک عدد اول. سیستم عددی تعیین می‌کند که چگونه باید اعداد را با یکدیگر جمع و یا در هم ضرب کنیم. می‌توانیم این تعاریف را گسترش دهیم تا تعاریفی برای جمع و ضرب ماتریس‌ها به دست بیاوریم.

می‌توانیم *جمع ماتریسی* را به صورت زیر تعریف کنیم. اگر $A = (a_{ij})$ و $B = (b_{ij})$ دو ماتریس $m \times n$ باشند، در این صورت جمع ماتریسی آن‌ها، یعنی $C = (c_{ij}) = A + B$ یک ماتریس $m \times n$ است که به صورت زیر تعریف می‌شود:

$$c_{ij} = a_{ij} + b_{ij}$$

برای $j = ۱, ۲, ..., n$. یعنی جمع ماتریسی به صورت عنصر به عنصر انجام می‌شود. یک ماتریس صفر، عنصر همانی برای جمع ماتریسی است:

$$A + ۰ = A = ۰ + A$$

اگر λ یک عدد باشد و $A = (a_{ij})$ یک ماتریس، در این صورت $\lambda A = (\lambda a_{ij})$ *ضرب اسکالر* ماتریس A است که با ضرب هر یک از عناصر A در λ به دست می‌آید. به عنوان یک حالت خاص، *منفی* یک ماتریس $A = (a_{ij})$ را به صورت $-A = -۱ \cdot A$ تعریف می‌کنیم، به طوری که عنصر ijام ماتریس $-A$ برابر است با $-a_{ij}$. بنابراین

$$A + (-A) = ۰ = (-A) + A$$

با تعاریف بالا می‌توانیم *تفریق ماتریسی* را به صورت جمع منفی یک ماتریس تعریف کنیم:

$$A - B = A + (-B)$$

ضرب ماتریسی را به صورت زیر تعریف می‌کنیم. با دو ماتریس A و B آغاز می‌کنیم که با یکدیگر *سازگار* (compatible) هستند، یعنی تعداد ستون‌های A برابر است با تعداد سطرهای B. (به طور کلی، همیشه فرض بر این است که یک عبارت که شامل ضرب ماتریسی AB است، ایجاب می‌کند که ماتریس‌های A و B سازگار باشند.) اگر $A = (a_{ik})$ یک ماتریس $m \times n$ و $B = (b_{kj})$ یک ماتریس $n \times p$ باشند، در این صورت ضرب ماتریسی آن‌ها $C = AB$، ماتریس $C = (c_{ik})$ با اندازه‌ی $m \times p$ است، که در آن

$$c_{ik} = \sum_{j=۱}^{n} a_{ij} b_{jk} \qquad (ت-۲)$$

برای $i = ۱, ۲, ..., m$ و $k = ۱, ۲, ..., p$. رویه‌ی SQUAR-MATRIX-MULTIPLY در بخش ۴-۲ ضرب ماتریسی را به روش مستقیم و بر مبنای تساوی (ت-۲) پیاده‌سازی می‌کند، با این فرض که ماتریس‌ها مربعی هستند: $m = n = p$. برای ضرب ماتریس‌های $n \times n$، رویه‌ی SQUARE-MATRIX-MULTIPLY تعداد $n^۳$ ضرب و $n^۲(n-۱)$ جمع انجام می‌دهد، و بنابراین زمان اجرای آن برابر است با $\theta(n^۳)$.

ماتریس‌ها، بسیاری از خصوصیات جبری اعداد معمولی را دارند (ولی نه همه‌ی آن‌ها را). ماتریس‌های همانی، عناصر همانی برای ضرب ماتریسی هستند:

$$I_m A = A I_n = A$$

برای هر ماتریس A با اندازه‌ی $m \times n$. ضرب در یک ماتریس صفر، یک ماتریس صفر به ما می‌دهد:

$$A ۰ = ۰$$

ضرب ماتریسی شرکت‌پذیر است:

$$A(BC) = (AB)C$$

برای ماتریس‌های سازگار A، B، و C. ضرب ماتریس‌ها بر روی جمع توزیع پذیر است:

$$A(B+C) = AB + AC$$
$$(B+C)D = BD + CD$$

برای $n > 1$، ضرب ماتریس‌های $n \times n$ خاصیت جابه‌جایی ندارد. به عنوان مثال اگر $A = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$ و $B = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$، در این صورت

$$AB = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

و

$$BA = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

ضرب ماتریس-بردار و یا ضرب بردار-بردار با این فرض انجام می‌شود که بردارها، ماتریس‌های $n \times 1$ (یا $1 \times n$، در حالتی که بردار سطری باشد) هستند. بنابراین اگر A یک ماتریس $m \times n$ و x یک بردار با اندازه‌ی n باشد، آن گاه Ax یک بردار با اندازه‌ی m خواهد بود. اگر x و y بردارهایی با اندازه‌ی n باشند، آن گاه

$$x^T y = \sum_{i=1}^{n} x_i y_i$$

یک عدد (در واقع یک ماتریس 1×1) است، که به آن *ضرب داخلی* (inner product) x و y می‌گوییم. ماتریس xy^T یک ماتریس Z با اندازه‌ی $n \times n$ است که به آن *ضرب خارجی* (outer product) x و y می‌گوییم، که در آن $z_{ij} = x_i y_i$. *نُرم (اقلیدسی)* (Euclidean norm) $\|x\|$ برای یک بردار x با اندازه‌ی n به صورت

$$\|x\| = (x_1^2 + x_2^2 + \ldots + x_n^2)^{1/2}$$
$$= (x^T x)^{1/2}$$

تعریف می‌شود. بنابراین، نُرم x برابر است با طول آن در فضای اقلیدسی n بعدی.

تمرین‌ها

ت-۱-۱ نشان دهید که اگر A و B ماتریس‌های متقارن $n \times n$ باشند، آن گاه $A+B$ و $A-B$ هم این گونه هستند.

ت-۱-۲ اثبات کنید که $(AB)^T = B^T A^T$، و این که $A^T A$ همیشه یک ماتریس متقارن است.

ت-۱-۳ اثبات کنید که ضرب دو ماتریس پایین- مثلثی همیشه یک ماتریس پایین- مثلثی است.

ت-۱-۴ اثبات کنید که اگر P یک ماتریس جایگشت $n\times n$ باشد، و A یک ماتریس $n\times n$، در این صورت حاصل‌ضرب PA را می‌توان از بازآرایی سطرهای A، و حاصل‌ضرب AP را می‌توان از بازآرایی ستون‌های A به دست آورد. اثبات کنید که حاصل ضرب دو ماتریس جایگشت، یک ماتریس جایگشت است.

ت-۲ معکوس، رتبه و دترمینان ماتریس‌ها

در این بخش بعضی خصوصیات اولیه‌ی ماتریس‌ها را بررسی می‌کنیم: معکوس، وابستگی و استقلال خطی، رتبه، و دترمینان. همچنین کلاس ماتریس‌های مطلقاً مثبت را تعریف می‌کنیم.

معکوس، رتبه، و دترمینان ماتریس‌ها

معکوس (inverse) یک ماتریس A با اندازه‌ی $n\times n$ را (در صورت وجود) با A^{-1} نشان می‌دهیم و به صورت ماتریسی تعریف می‌کنیم که برای آن داشته باشیم $AA^{-1}=I_n=A^{-1}A$. به عنوان مثال

$$\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{-1}=\begin{pmatrix}0 & 1\\ 1 & -1\end{pmatrix}$$

بسیاری از ماتریس‌های غیر صفر با اندازه‌ی $n\times n$ ماتریس معکوس ندارند. یک ماتریس که ماتریس معکوس برای آن وجود ندارد، *معکوس ناپذیر* (noninvertible) یا *تکین* (singular) نامیده می‌شود. یک مثال برای یک ماتریس غیر صفر تکین به صورت زیر است:

$$\begin{pmatrix}1 & 0\\ 1 & 0\end{pmatrix}$$

اگر ماتریسی، یک ماتریس معکوس داشته باشد، به این ماتریس *معکوس‌پذیر* یا *غیرتکین* می‌گوییم. معکوس ماتریس‌ها در صورت وجود یکتا هستند. (تمرین ت-۲-۱ را ببینید.) اگر A و B ماتریس‌های غیرتکین $n\times n$ باشند، در این صورت

$$(BA)^{-1}=A^{-1}B^{-1}$$

عملیات محاسبه‌ی معکوس با عملیات محاسبه‌ی ترانهاده قابل جابه‌جایی است:

$$(A^{-1})^T=(A^T)^{-1}$$

بردارهای $x_1, x_2, \ldots, x_n$ *وابسته‌ی خطی* (linearly dependent) هستند اگر ضرایب $c_1, c_2, \ldots, c_n$ که تمام آن‌ها صفر نیستند، وجود داشته باشند به طوری که $c_1x_1+c_2x_2+\ldots+c_nx_n=0$. به عنوان مثال بردارهای سطری $x_1=(1\ 2\ 3)$، $x_2=(4\ 6\ 2)$، و $x_3=(4\ 11\ 9)$ *وابسته‌ی خطی* (linearly dependent) هستند، چرا که $2x_1+3x_2+2x_3=0$. اگر بردارها وابسته‌ی خطی نباشند، *مستقل خطی* (linearly independent) هستند. به عنوان مثال ستون‌های یک ماتریس همانی، مستقل خطی هستند.

رتبه‌ی ستونی (column rank) یک ماتریس غیر صفر A با اندازه‌ی $m \times n$ برابر است با اندازه‌ی بزرگ‌ترین مجموعه‌ی ستون‌های مستقل خطی A. به طور مشابه، *رتبه‌ی سطری* (row rank) یک ماتریس غیر صفر A با اندازه‌ی $m \times n$ برابر است با اندازه‌ی بزرگ‌ترین مجموعه‌ی سطرهای مستقل خطی A. یک خصوصیت اصلی یک ماتریس A این است که رتبه‌ی سطری آن همیشه برابر است با رتبه‌ی ستونی آن، به طوری که می‌توانیم هر دوی آن‌ها را *رتبه‌ی* A بنامیم. رتبه‌ی یک ماتریس $m \times n$ یک عدد صحیح است بین ۰ و $\min(m,n)$. (رتبه‌ی یک ماتریس ۰ برابر است با ۰، و رتبه‌ی یک ماتریس همانی $n \times n$ برابر است با n.) یک تعریف جایگزین، ولی معادل و اکثراً مفیدتر، این است که رتبه‌ی یک ماتریس A با اندازه‌ی $m \times n$ برابر است با کوچک‌ترین عدد r به طوری که ماتریس‌های B و C با اندازه‌های به ترتیب $m \times r$ و $r \times n$ وجود داشته باشند، به طوری که

$$A = BC$$

یک ماتریس مربعی $n \times n$ دارای *رتبه‌ی کامل* (full rank) است اگر رتبه‌ی آن برابر باشد با n. یک ماتریس $m \times n$ دارای *رتبه‌ی ستونی کامل* (full column rank) است اگر رتبه‌ی آن برابر باشد با n. یک خصوصیت مهم رتبه‌ها در قضیه‌ی زیر مشخص می‌شود.

قضیه‌ی ت-۱ یک ماتریس مربعی دارای رتبه‌ی کامل است اگر و فقط اگر غیرتکین باشد.

یک *بردار تهی* (null vector) برای یک ماتریس A، یک بردار غیر صفر x است به طوری که $Ax = 0$. قضیه‌ی زیر (که اثبات آن به عنوان تمرین ت-۲-۷ واگذار شده است) و نتیجه‌ی آن، مفهوم رتبه‌ی ستونی و تکین بودن را به بردارهای تهی مربوط می‌کنند.

قضیه‌ی ت-۲ یک ماتریس A دارای رتبه‌ی ستونی کامل است اگر و فقط اگر بردار تهی نداشته باشد.

نتیجه‌ی ت-۳ ماتریس مربعی A تکین است اگر و فقط اگر یک بردار تهی داشته باشد.

ijامین *ماتریس فرعی* (minor) یک ماتریس A با اندازه‌ی $n \times n$، برای $n > 1$، یک ماتریس $A_{[ij]}$ با اندازه‌ی $(n-1) \times (n-1)$ است که از حذف سطر i و ستون j ماتریس A به دست می‌آید. *دترمینان* (determinant) یک ماتریس A با اندازه‌ی $n \times n$ را می‌توان به صورت بازگشتی نسبت به ماتریس‌های فرعی تعریف کرد:

$$\det(A)=\begin{cases}a_{11} & \text{اگر } n=1\\ \sum_{j=1}^{n}(-1)^{1+j}a_{1j}\det(A_{[1j]}) & \text{اگر } n>1\end{cases}$$

عبارت $(-1)^{i+j}\det(A_{[ij]})$ با نام *کوفاکتور* (cofactor) عنصر a_{ij} شناخته می‌شود.

قضیه‌های زیر، که در این جا از اثبات آن‌ها صرف نظر شده است، خصوصیات اصلی دترمینان‌ها را توصیف می‌کنند.

قضیه‌ی ت-۴ (خصوصیات دترمینان‌ها)

دترمینان یک ماتریس مربعی A خصوصیات زیر را دارد:

- اگر هر یک از سطرها و یا ستون‌های A صفر باشد، آن گاه داریم $\det(A)=0$.
- اگر تمام ورودی‌های هر یک از سطرها (و یا هر یک از ستون‌ها)ی ماتریس A در یک عدد λ ضرب شوند، آن گاه دترمینان ماتریس A هم در λ ضرب می‌شود.
- اگر ورودی‌های یک سطر (ستون) به ورودی‌های متناظر در یک سطر (ستون) دیگر اضافه شوند، دترمینان ماتریس بدون تغییر باقی می‌ماند.
- دترمینان ماتریس A برابر است با دترمینان ماتریس A^T.
- اگر جای دو سطر (یا دو ستون) در ماتریس A با یکدیگر عوض شود، آن گاه دترمینان A در -1 ضرب می‌شود.

همچنین، برای هر ماتریس مربعی A و B داریم $\det(AB)=\det(A)\det(B)$.

قضیه‌ی ت-۵ یک ماتریس A با اندازه‌ی $n\times n$ تکین است اگر و فقط اگر داشته باشیم $\det(A)=0$.

ماتریس‌های مطلقاً مثبت

ماتریس‌های مطلقاً مثبت نقش مهمی در بسیاری از کاربردها بازی می‌کنند. یک ماتریس A با اندازه‌ی $n\times n$ *مطلقاً مثبت* (positive-definite) است اگر $x^TAx>0$ برای تمام بردارهای $x\neq 0$ با اندازه‌ی n. به عنوان مثال ماتریس همانی یک ماتریس مطلقاً مثبت است، چرا که برای هر بردار غیر صفر $x=(x_1\ x_2\ \dots\ x_n)^T$، داریم

$$\begin{aligned}x^T I_n x &= x^T x\\ &=\sum_{i=1}^{n}x_i^2\\ &>0\end{aligned}$$

همان طور که خواهیم دید، ماتریس‌های موجود در کاربردها معمولاً مطلقاً مثبت هستند، بنابر قضیه‌ی زیر.

قضیه‌ی ت-۲ برای هر ماتریس A با رتبه‌ی ستونی کامل، ماتریس $A^T A$ مطلقاً مثبت است.

اثبات باید نشان دهیم که برای هر بردار غیر صفر x داریم $x^T (A^T A)x > 0$. برای هر بردار x،

$$x^T (A^T A)x = (Ax)^T (Ax) \quad \text{(طبق تمرین ت-۱-۲)}$$
$$= \|Ax\|^2$$

توجه کنید که $\|Ax\|^2$ به سادگی برابر است با مجموع مربعات عناصر بردار Ax. بنابراین $\|Ax\|^2 \geq 0$. اگر $\|Ax\|^2 = 0$، هر عنصر Ax برابر است با ۰، که می‌توانیم بگوییم $Ax = 0$. از آن جایی که A رتبه‌ی ستونی کامل دارد، $Ax = 0$ ایجاب می‌کند که $x = 0$، طبق قضیه‌ی ت-۲. بنابراین $A^T A$ مطلقاً مثبت است. ■

تمرین‌ها

ت-۲-۱ اثبات کنید که معکوس ماتریسی همیشه یکتا است، یعنی، اگر B و C معکوس‌های A باشند، آن گاه داریم $B = C$.

ت-۲-۲ اثبات کنید که دترمینان یک ماتریس پایین- مثلثی یا یک ماتریس بالا- مثلثی، برابر است با ضرب عناصر قطری آن. اثبات کنید که معکوس یک ماتریس پایین- مثلثی، در صورت وجود یک ماتریس پایین- مثلثی است.

ت-۲-۳ اثبات کنید که اگر P یک ماتریس جایگشت باشد، آن گاه P معکوس‌پذیر است، معکوس آن P^T است، و P^T یک ماتریس جایگشت است.

ت-۲-۴ فرض کنید A و B ماتریس‌های $n \times n$ باشند به طوری که $AB = I$. اثبات کنید که اگر A' با جمع سطر j ماتریس A با سطر i آن به دست آمده باشد، آن گاه B'، ماتریس معکوس A'، را می‌توان از تفریق ستون i ماتریس B از ستون j آن به دست آورد.

ت-۲-۵ فرض کنید که A یک ماتریس غیرتکین $n \times n$ با ورودی‌های مختلط باشد. نشان دهید که تمام ورودی‌های A^{-1} حقیقی هستند اگر و فقط اگر تمام ورودی‌های A حقیقی باشند.

ت-۲-۶ نشان دهید که اگر A یک ماتریس غیرتکین، متقارن، و $n \times n$ باشد، آن گاه A^{-1} متقارن است. نشان دهید که اگر B یک ماتریس $m \times n$ دلخواه باشد، آن گاه ماتریس $m \times m$ به دست آمده از ضرب BAB^T متقارن است.

ت-۲-۷ قضیه‌ی ت-۲ را اثبات کنید. یعنی نشان دهید که یک ماتریس A رتبه‌ی ستونی کامل دارد

اگر و فقط اگر $Ax = 0$ نتیجه دهد $x = 0$. (*راهنمایی:* وابستگی خطی یک ستون به ستون‌های دیگر را به صورت یک تساوی ماتریس-برداری نشان دهید.)

ت-۲-۸ اثبات کنید که برای هر دو ماتریس سازگار A و B،

$$\text{rank}(AB) \le \min(\text{rank}(A), \text{rank}(B)),$$

که در آن حالت تساوی زمانی برقرار است که یکی از ماتریس‌ها، مربعی و غیرتکین باشد. (*راهنمایی:* از تعریف جایگزین رتبه‌ی یک ماتریس استفاده کنید.)

مسائل

ت-۱ ماتریس Vandermonde

با داشتن اعداد $x_0, x_1, \ldots, x_{n-1}$، اثبات کنید که دترمینان *ماتریس Vandermonde*

$$V(x_0, x_1, \ldots, x_{n-1}) = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix}$$

برابر است با

$$\det(V(x_0, x_1, \ldots, x_{n-1})) = \prod_{0 \le j < k \le n-1} (x_k - x_j)$$

(*راهنمایی:* برای $i = n-1, n-2, \ldots, 1$، ستون i را در $-x_0$ ضرب کرده و آن را به ستون $i+1$ اضافه کنید، و سپس از استقرا استفاده کنید.)

ت-۲ جایگشت‌های تعریف شده توسط ضرب ماتریس-بردار روی GF(2)

یکی از کلاس‌های جایگشت‌های اعداد صحیح در مجموعه‌ی $S_n = \{0, 1, 2, \ldots, 2^n - 1\}$ به وسیله‌ی ضرب ماتریسی روی $GF(2)$ تعریف می‌شود. برای هر عدد صحیح x در S_n، نمایش دودویی آن را به صورت یک بردار n بیتی در نظر می‌گیریم:

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{pmatrix}$$

که در آن $x = \sum_{i=0}^{n-1} x_i 2^i$. اگر A یک ماتریس $n \times n$ باشد که در آن هر ورودی یا 0 باشد و

یا ۱، آن گاه می‌توانیم جایگشتی تعریف کنیم که در آن هر مقدار x در S_n به عددی نگاشت می‌شود که نمایش دودویی آن، ضرب ماتریس-بردار Ax است. در این جا تمام اعمال ریاضی را روی $GF(2)$ انجام می‌دهیم: تمام مقادیر یا ۰ هستند و یا ۱، و با یک استثنا، تمام قوانین معمول جمع و ضرب اعمال می‌شوند. استثنا این است که $1+1=0$. می‌توانید اعمال ریاضی روی $GF(2)$ به صورت اعمال ریاضی معمول در نظر بگیرید، با این تفاوت که در آن فقط از کم‌ارزش‌ترین بیت استفاده می‌شود.

به عنوان یک مثال ماتریس

$$A = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

جایگشت زیر را برای $S_2 = \{0, 1, 2, 3\}$ تعریف می‌کند:

$$\pi_A : \ \pi_A(0) = 0, \ \pi_A(1) = 3, \ \pi_A(2) = 2, \ \pi_A(3) = 1$$

برای این که ببینید چرا $\pi_A(3) = 1$، مشاهده کنید که در فضای $GF(2)$ داریم

$$\begin{aligned} \pi_A(3) &= \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} 1\cdot 1 + 0\cdot 1 \\ 1\cdot 1 + 1\cdot 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{aligned}$$

که نمایش دودویی عدد ۱ است.

در ادامه‌ی این مسئله در فضای $GF(2)$ کار می‌کنیم، و تمام ورودی‌های ماتریس‌ها و بردارهایی که با آن‌ها کار می‌کنیم یا ۰ هستند و یا ۱. رتبه‌ی یک ماتریس $n \times n$ (ماتریسی که تمام ورودی‌های آن یا ۰ است یا ۱) روی $GF(2)$ را مشابه ماتریس‌های معمولی تعریف می‌کنیم، با این تفاوت که تمام اعمال ریاضی که استقلال خطی را تعیین می‌کنند روی $GF(2)$ انجام می‌شوند. اگر A یک ماتریس $n \times n$ با اندازه‌ی $n \times n$ باشد، *دامنه‌ی* (range) آن را به صورت زیر تعریف می‌کنیم:

$$R(A) = \{y : x \in S_n \text{ برای یک } y = Ax\}$$

یعنی $R(A)$ مجموعه‌ی اعدادی از S_n است که می‌توان با ضرب مقادیر x عضو S_n در A به دست آورد.

I. اگر r رتبه‌ی ماتریس A باشد، اثبات کنید که $|R(A)| = 2^r$. نتیجه بگیرید که A یک جایگشت روی S_n ارائه می‌کند فقط اگر A رتبه‌ی کامل داشته باشد.

برای یک ماتریس داده شده‌ی A با اندازه‌ی $n \times n$ و یک مقدار داده شده‌ی $y \in R(A)$، *پیش‌تصویر* (preimage) مقدار y را به صورت زیر تعریف می‌کنیم:

$$P(A, y) = \{x : Ax = y\}$$

یعنی $P(A, y)$ مجموعه‌ی مقادیری در S_n است که با ضرب در A به مقدار y نگاشت می‌شوند.

II. اگر r رتبه‌ی ماتریس A با اندازه‌ی $n \times n$ باشد و $y \in R(A)$، اثبات کنید $|P(A, y)| = ۲^{n-r}$.

فرض کنید $n \leq m \leq ۰$، و فرض کنید مجموعه‌ی S_n را به بلوک‌هایی از اعداد متوالی تقسیم می‌کنیم، که در آن iامین بلوک حاوی ۲^m عدد $i۲^m, i۲^m + ۱, i۲^m + ۲, \ldots, (i+۱)۲^m - ۱$ است. برای هر زیرمجموعه‌ی $S \subseteq S_n$، $B(S, m)$ را به صورت مجموعه‌ای از بلوک‌های با اندازه‌ی ۲^m در S_n تعریف می‌کنیم که حاوی حداقل یکی از عناصر S باشند. مثلاً اگر $n = ۳$، $m = ۱$، و $S = \{۱, ۴, ۵\}$، آن گاه $B(S, m)$ عبارت است از بلوک‌های ۰ (چون ۱ در ۰امین بلوک است) و ۲ (چون ۴ و ۵ در دومین بلوک هستند).

III. فرض کنید r رتبه‌ی زیرماتریس $(n-m) \times m$در پایین و سمت چپ A باشد، یعنی ماتریسی که از تقاطع $n-m$ردیف پایین و m ستون چپ A به دست می‌آید. فرض کنید S یک بلوک دلخواه از S_n با اندازه‌ی ۲^m باشد، و همچنین

$$S' = \{y : x \in S \text{ برای یک } y = Ax\}$$

اثبات کنید که $|B(S', m)| = ۲^r$، و این که برای هر بلوک در $B(S', m)$ دقیقاً ۲^{m-r} عدد در S به آن بلوک نگاشت می‌شوند.

از آن جایی که نتیجه‌ی ضرب بردار صفر در هر ماتریسی، بردار صفر است، مجموعه‌ی جایگشت‌های S_n که به صورت ضرب در ماتریس‌های ۰-۱ با اندازه‌ی $n \times n$ و رتبه‌ی کامل روی $GF(۲)$ تعریف می‌شود، نمی‌تواند حاوی تمام جایگشت‌های S_n باشد. اجازه دهید کلاس جایگشت‌های تعریف شده توسط ضرب ماتریس-بردار را گسترش دهیم به طوری که مجاز باشند از یک عبارت جمع هم استفاده کنند. در این صورت $x \in S_n$ به $Ax + c$ نگاشت می‌شود، که در آن c یک بردار nبیتی است و جمع در فضای $GF(۲)$ انجام می‌شود. مثلاً اگر

$$A = \begin{pmatrix} ۱ & ۰ \\ ۱ & ۱ \end{pmatrix}$$

و

$$c = \begin{pmatrix} ۰ \\ ۱ \end{pmatrix}$$

جایگشت زیر را خواهیم داشت:

$$\pi_{A,c} : \ \pi_{A,c}(۰) = ۲, \ \pi_{A,c}(۱) = ۱, \ \pi_{A,c}(۲) = ۰, \ \pi_{A,c}(۳) = ۳$$

برای یک ماتریس A با اندازه‌ی $n \times n$ و رتبه‌ی کامل و یک بردار nبیتی c، هر جایگشتی که $x \in S_n$ را به $Ax + c$ نگاشت می‌کند، یک *جایگشت خطی* (linear permutation) می‌نامیم.

IV. با استفاده از یک بحث شمارشی نشان دهید که تعداد جایگشت‌های خطی S_n بسیار کم‌تر از تعداد کل جایگشت‌های S_n است.

V. مثالی از یک مقدار n و یک جایگشت برای S_n ارائه کنید که نمی‌توان به کمک جایگشت‌های خطی به آن دست یافت. (*راهنمایی:* برای یک جایگشت داده شده، به رابطه‌ی میان ضرب یک ماتریس در یک بردار یکه و ستون‌های ماتریس توجه کنید.)

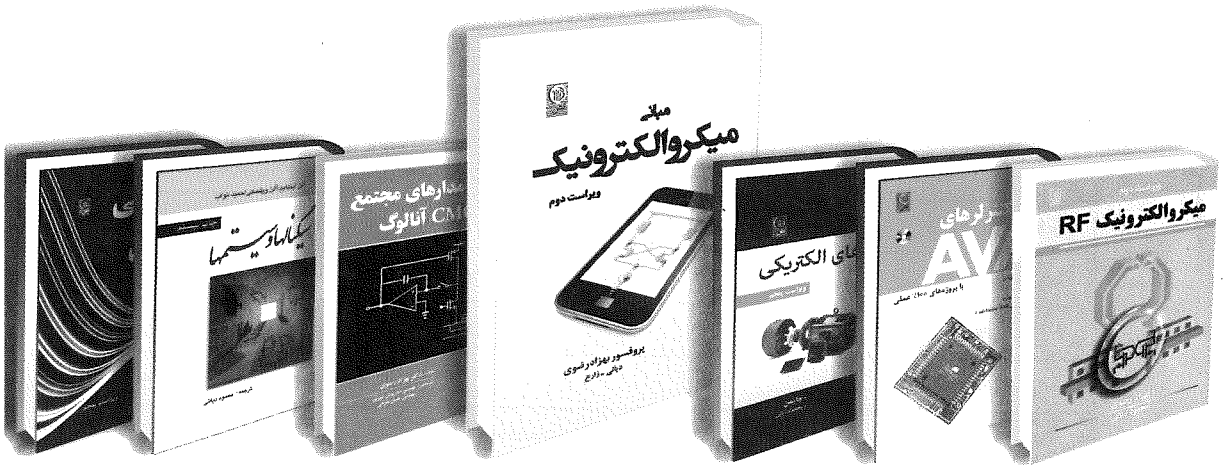
ناشر کتاب‌های فنی، مهندسی، دانشگاهی، کامپیوتر و مدیریت

نص

فهرست بخشی از کتاب‌های «مؤسسه علمی فرهنگی نص»

کتاب‌های مهندسی

ردیف	عنوان	مولف/ مترجم	قیمت به تومان / تعداد صفحات
۱	**مبانی فیزیک(۱) مکانیک**/ ویراست ۷/ چاپ ۳/ تمام رنگی/ با CD	هالیدی- رزنیک/ دیانی	۱۲۰۰۰/ ۴۰۸ ص
۲	**مبانی فیزیک الکتریسیته**/ ویراست ۷/ چاپ ۲/ تمام رنگی/ با CD	هالیدی- رزنیک/ دیانی	۹۵۰۰/ ۴۱۶ ص
۳	**مبانی فیزیک سیالات، موج، حرارت**/ ویراست ۷/ تمام رنگی/ با CD	هالیدی- رزنیک/ دیانی	۴۵۰۰/ ۲۴۸ ص
۴	**مبانی الکترومغناطیس** همراه با میدان‌ها و امواج/ ویراست ۵/ جلد سخت/ دو رنگ/ با CD	ماتیوان.او سدیکو/ دیانی	۱۹۰۰۰/ ۹۱۲ ص
۵	**مبانی الکترومغناطیس**/ جلد نرم/ دو رنگ/ با CD/ جلد نرم/ویراست ۵	ماتیوان.او سدیکو/ دیانی	۱۱۰۰۰/ ۵۶۸ ص
۶	**شیمی**/ جلد ۱/ تمام رنگی/ قطع رحلی	سومدال/ محمود دیانی	۹۵۰۰/ ۴۰۰ ص
۷	**شیمی**/ جلد ۲/ تمام رنگی/ قطع رحلی	سومدال/ محمود دیانی	۹۵۰۰/ ۴۰۸ ص
۸	**الکترونیک** (مدار، طراحی،کاربرد)/ ویراست ۷/ تمام رنگی/ با CD/ قطع رحلی	فلوید/ دیانی	۲۸۰۰۰/ ۸۳۲ ص
۹	**استاتیک** (مکانیک برداری)/ ویراست ۸/ چهاررنگ/ با CD	بیبر- جانسون/ زارع‌پور	۱۸۰۰۰/ ۵۸۴ ص
۱۰	**طراحی اجزاء ماشین**/ جلد ۱/ چاپ ۶/ دورنگ/ با CD/ قطع رحلی	شیگلی/ زارع‌پور	۱۸۰۰۰/ ۸۸۸ ص
۱۱	**طراحی اجزاء ماشین**/ جلد ۲/ چاپ ۴/ دورنگ/ با CD/ قطع رحلی / نایاب	شیگلی/ زارع‌پور	۹۵۰۰۰/ ۳۰۰ ص
۱۲	**تحلیل مهندسی مدار**/ ویراست ۷/ تمام رنگی/ با CD/ قطع رحلی/چاپ ۵	هیت/ دیانی	۴۸۰۰۰/ ۷۲۰ ص
۱۳	**آموزش نرم افزار CFX** با DVD	مهرجوئی- نیک‌خو	۶۰۰۰/ ۲۹۶ ص
۱۴	**مدارهای میکروالکترونیک**/ جلد ۱/ چاپ ۴/ دورنگ/ با CD	سدره اسمیت/ دیانی	۱۵۰۰۰/ ۷۵۲ ص
۱۵	**مدارهای میکروالکترونیک**/ جلد ۲/ چاپ ۳/ دورنگ/ با CD	سدره اسمیت/ دیانی	۱۵۰۰۰/ ۷۲۸ ص
۱۶	**مدارهای میکروالکترونیک/ویراست ۶**/جلد ۱/ دو رنگ/ با DVD	کنت اسمیت/سدره/دیانی	۲۸۰۰۰/ ۵۷۶ ص
۱۷	**مدارهای میکروالکترونیک**/ویراست ۶/ جلد ۲/ دو رنگ/ با DVD	کنت اسمیت/سدره/دیانی	۱۶۰۰۰/ ۶۵۶ ص



همکاران نص در استان ها

فروشگاه کتاب مرکزی اصفهان : خ چهارباغ خ سید علیخان	۰۳۱-۳۲۲۲۳۷۲۵
فروشگاه کتاب گنج دانش اصفهان: خ جهارباغ مقابل هتل عباسی	۰۳۱-۳۲۲۲۲۶۱۲
پخش کتاب کیمیا اصفهان : میدان شهدا خ کاوه	۰۳۱-۳۴۴۸۴۰۲۰
نمایشگاه کتاب رشد اهواز: خ حافظ بین سیروس و نادری	۰۶۱-۳۲۲۱۷۰۰۰
فروشگاه کتاب شرق اهواز: خ نادری بین حافظ و فردوسی	۰۶۱-۳۲۲۳۰۵۵۳
شهر کتاب اهواز: خ نادری بین حافظ و فردوسی	۰۶۱-۳۲۲۱۲۳۱۴
نمایشگاه کتاب شریعتی اردبیل : میدان شریعتی جنب ارس اپتیک	۰۴۵-۳۳۲۴۳۱۳۸
کتابکده خیام اردبیل : خ امام خمینی جنب سرپرستی بانک رفاه	۰۴۵-۳۳۲۳۹۳۱۹
دانش پژوه ارومیه : خ امام خمینی روبروی آموزش و پرورش	۰۴۴-۳۲۲۲۷۵۷۸
کتابسرای پژوهش بابل : خ شریعتی روبروی دانشگاه نوشیروانی	۰۱۱-۳۲۳۳۰۶۳۹
انتشارات آشینا تبریز: خ امام خمینی بازار بزرگ تربیت طبقه پایین	۰۴۱ -۳۵۵۶۸۳۳۴
کتاب‌فروشی شایسته تبریز: خ امام خمینی روبروی مصلا	۰۴۱-۳۵۵۶۵۴۰۵
کتاب‌فروشی علامه تبریز: فلکه دانشگاه اول خ دانشگاه پلاک ۲۳۰	۰۴۱-۳۳۳۴۱۶۶۹
کتاب‌فروشی نگار بهبهان : خ عدالت پاساز مودتی	۰۶۱-۵۲۷۲۰۷۱۵
کتاب‌فروشی ولایت بروجرد: خ شهدا پاساژ آیینه طبقه پایین	۰۶۶- ۴۲۶۲۹۵۵۰
کتاب‌فروشی شریعتی خرم آباد: خ شریعتی میدان بسیج بالاتر میدان بسیج	۰۶۶-۳۲۲۲۸۷۸۴
کتاب‌فروشی دنیای زبان خرم آباد : چهارراه فرهنگ	۰۶۶-۳۳۳۰۳۳۷۶
کتاب‌فروشی معراج دزفول : خ شهید بهشتی نبش خ شریعتی	۰۶۱-۳۲۲۶۱۸۲۱
کتابسرای مژده رشت : خ امام خمینی جنب سینما انقلاب	۰۱۳-۳۳۲۲۳۶۳۷
مجتمع مهران رشت : ابتدای خ سعدی روبروی کلیسا جنب بانک کشاورزی	۰۱۳-۳۳۲۴۴۴۸۹
شهرکتاب زنجان : چهارراه سعدی روبروی بانک پاسارگاد	۰۲۴-۳۳۳۲۴۴۸۱
کتابکده صدرا شاهرود : خ ۲۲ بهمن پاساژ باقری	۰۲۳-۳۲۲۳۸۲۷۹
کتابسرای دنیای خرد شیراز: مشیر فاطمی ابتدای خ معدل ساختمان ۱۱۰	۰۷۱-۳۲۳۳۵۱۶۹
کتاب‌فروشی محمدی شیراز:خ زند - ابتدای خ ملاصدرا	۰۷۱-۳۶۴۷۳۵۳۴