

به نام خدا

الگوریتم‌های پیشرفته

آرش شفیعی



فهرست مطالب

کوتاه‌ترین مسیر

مسئله‌های انپی کامل

شاربیشینه

تطابق در گراف دو بخشی

الگوریتم‌های تقریبی

برنامه ریزی خطی

الگوریتم‌های موازی

تحلیل احتمالاتی و الگوریتم‌های تصادفی

پیوست شماره یک: مروری بر نظریه زبان‌ها صوری

الگوریتم‌های پیشرفته

کتاب‌های مرجع

- مقدمه‌ای بر الگوریتم‌ها از کرمن، لایرسون، ریوست، و استاین^۱

^۱ Introduction to Algorithms, by Cormen, Leiserson, Rivest, and Stein

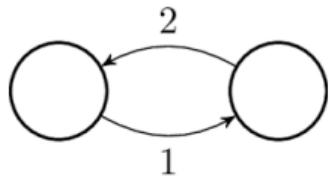
کوتاهترین مسیر

یادآوری

تعریف گراف

- گراف‌ها به دو دسته جهت‌دار و بدون جهت تقسیم می‌شوند. این دسته بندی از این جهت مهم است که گراف‌های جهت‌دار و بدون جهت به شکل مجزا تعریف می‌شوند و تفاوت‌هایی در تعریف آنها وجود دارد؛
 - ۱ گراف بدون جهت: طبق تعریف این گراف نمی‌تواند طوقه یا یال موازی داشته باشد.
 - ۲ گراف جهت‌دار: طبق تعریف این گراف می‌تواند طوقه داشته باشد اما نمی‌تواند یال موازی داشته باشد.

- البته دو یال بین دو رأس یکسان در صورتی که در جهت مخالف یکدیگر باشند موازی محسوب نمی‌شوند.
برای مثال شکل زیر یک گراف فاقد یال موازی است.



- به این یال‌ها پادموازی^۱ گفته می‌شود. بنابراین گراف جهت دار می‌تواند یال پادموازی داشته باشد.
- در هر یک از مسائل بسته به ذات مسئله نوع خاصی از گراف به عنوان ورودی در نظر گرفته می‌شود. برای مثال ورودی مسئله کوتاه ترین مسیر در حالت کلی یک گراف جهت دار و وزن دار است.

^۱ antiparallel

یادآوری

تحلیل الگوریتم‌های گراف

- الگوریتم‌های گراف برخلاف بیشتر الگوریتم‌های دارای دو متغیر تاثیرگذار در اندازه ورودی‌اند: تعداد یال‌ها ($|E|$) و تعداد رئوس ($|V|$).
 - بر اساس یک قرارداد شناخته‌شده می‌توان در نمادهای مجانبی از قرار دادن نماد اندازه در اطراف V و E صرف نظر کرد.
 - الگوریتم‌های گراف به طور معمول در دو حالت بررسی می‌شوند:
- الف زمانی که گراف متراکم باشد: در این حالت فرض می‌کنیم همه رئوس به هم متصل هستند بنابراین تعداد یال‌ها از مرتبه $(V^2)\Theta$ است.
- ب زمانی که گراف خلوت باشد: در این حالت به طور معمول فرض می‌شود که تعداد یال‌ها از مرتبه $(V)\Theta$ است.

یادآوری

تحلیل الگوریتم‌های گراف

- پیچیدگی زمانی ارائه شده برای یک الگوریتم گراف را می‌توان در دو حالت بالا تحلیل کرد. برای مثال تمرين زیر را در نظر بگیرید:

21.2-3

For a sparse graph $G = (V, E)$, where $|E| = \Theta(V)$, is the implementation of Prim's algorithm with a Fibonacci heap asymptotically faster than the binary-heap implementation? What about for a dense graph, where $|E| = \Theta(V^2)$? How must the sizes $|E|$ and $|V|$ be related for the Fibonacci-heap implementation to be asymptotically faster than the binary-heap implementation?

یادآوری

تحلیل الگوریتم‌های گراف

- پیچیدگی زمانی الگوریتم پریم با استفاده از هرم دودویی از مرتبه $O(E \log V + V \lg V)$ و با استفاده از هرم فیبوناچی از مرتبه $O(E + V \log V)$ است. با جایگذاری V به جای E در این دوتابع درمی‌یابیم که در گراف خلوت هر دو پیاده‌سازی از لحاظ مجانبی سرعت یکسانی دارند و از مرتبه $O(V \lg V)$ اند. اما در گراف متراکم پیاده‌سازی با هرم فیبوناچی از لحاظ مجانبی سریع‌تر و از مرتبه $O(\lg V^2)$ است.
- چنین تحلیلی در دیگر الگوریتم‌های گراف هم کاربرد دارد. برای مثال الگوریتم فلوید-وارshall از مرتبه زمانی (V^3) است. از تحلیل این تابع می‌توان نتیجه گرفت خلوت یا متراکم بودن گراف از نظر مجانبی تاثیری بر سرعت این الگوریتم ندارد.

الگوریتم جانسون

- الگوریتم جانسون^۱ مانند الگوریتم فلوید وارشال برای یافتن کوتاه ترین مسیر بین هر دو راس گراف است. این الگوریتم برای گراف‌های خلوت پیچیدگی زمانی کمتری نسبت به بقیه روش‌ها (فلوید وارشال و روش شبیه‌سازی ضرب ماتریسی) دارد.
- الگوریتم جانسون - مانند الگوریتم بلمن فورد- می‌تواند روی گراف دارای یال منفی کوتاه ترین مسیر را به درستی محاسبه کند و وجود دور منفی در گراف را تشخیص و گزارش دهد. (گراف دارای دور منفی در مسئله کوتاه ترین مسیر یک حالت خطأ محسوب می‌شود). ^۲

^۱ Johnson Algorithm

^۲ برای گراف دارای دور منفی یافتن کوتاه ترین مسیر ساده می‌تواند کاربرد داشته باشد. این مسئله انپی-سخت است.

الگوریتم جانسون

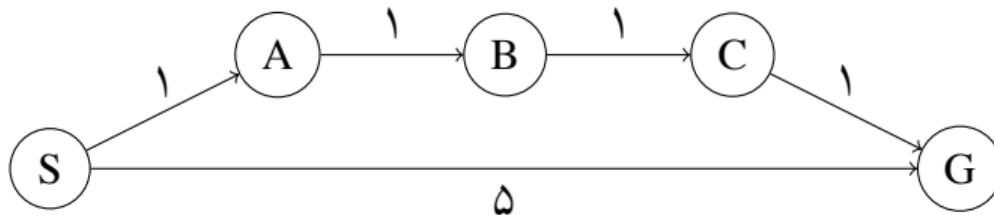
- برای طراحی یک الگوریتم مسئله کوتاهترین مسیر بین هر جفت رأس می‌توان V بار اجرا کردن یکی از الگوریتم‌های کوتاهترین مسیرهای هم مبداء را به عنوان یک حد بالا برای مرتبه زمانی تحلیل کرد.
- برای مثال $|V|$ بار اجرای الگوریتم بلمن فورد مرتبه $O(V^2E)$ و $|V|$ بار اجرای الگوریتم دایجسترا مرتبه $O(VE + V^2\lg V)$ را به دست می‌دهد.
- روش دوم سریع‌تر است اما در گراف‌هایی که یال منفی داشته باشند به درستی کار نمی‌کند.

الگوریتم جانسون

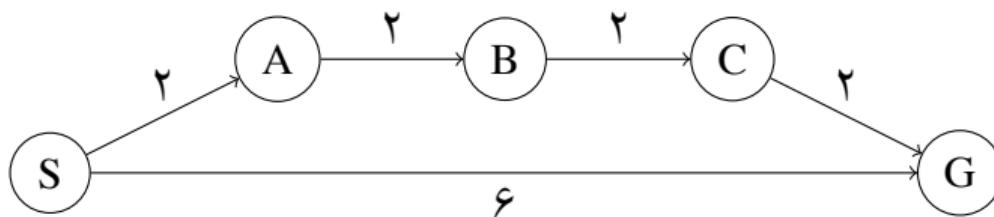
- الگوریتم جانسون نیز از ایده‌ایی مشابه استفاده می‌کند. این الگوریتم هر دو الگوریتم بلمن-فورد و دایجسترا را فراخوانی می‌کند؛
- ١ جانسون در مرحله اول، الگوریتم بلمن-فورد را استفاده می‌کند تا وجود دور منفی را تشخیص دهد و در صورتی که دور منفی وجود داشت آن را گزارش داده و کار تمام می‌شود.
 - ٢ در صورتی که دور منفی وجود نداشت، این الگوریتم با استفاده از روشی که در ادامه بررسی خواهد شد، همه وزن‌ها را به مقادیر غیر منفی نگاشت می‌کند.
 - ٣ حال که همه وزن‌های غیر منفی شده‌اند برای هر رأس یک بار الگوریتم دایجسترا اجرا می‌شود تا کوتاه‌ترین مسیرها از هر رأس به بقیه رئوس محاسبه شوند.

الگوریتم جانسون

- اما چطور باید وزن‌های گراف را به مقادیر غیر منفی نگاشت کرد؟ واضح است که بعد از تغییر وزن‌ها نباید کوتاه‌ترین مسیرها را تغییر کنند. برای مثال اضافه کردن یک مقدار ثابت به همه وزن‌ها نگاشت مناسبی نیست. برای درک بهتر این موضوع به مثال زیر توجه کنید:



- اضافه کردن یک واحد به همه یال‌ها کوتاه‌ترین مسیر از S به G را تغییر می‌دهد:



الگوریتم جانسون

- باید ویژگی‌های این تغییر وزن را به طور دقیق‌تر بررسی کنیم؛
فرض کنید وزن‌های گراف توسط تابعی به نام «تابع وزن»^۱ داده شده. به طوری که وزن یال بین u و v برابر است با $w(u, v)$. آنگاه تابع وزن جدیدی مانند \hat{w} باید این دو ویژگی را داشته باشد:
 - ۱ به ازای هر $E \in E$, $u, v \in E$ مسیری مانند p با استفاده از تابع وزن w یک کوتاه‌ترین مسیر از u به v است اگر و تنها اگر مسیر p با استفاده از تابع وزن \hat{w} نیز یک کوتاه‌ترین مسیر باشد.
 - ۲ به ازای هر $E \in E$, $u, v \in E$ مقدار $(\hat{w}(u, v) - w(u, v))$ غیر منفی باشد.

^۱ weight function

الگوریتم جانسون

- الگوریتم جانسون از این تغییر وزن استفاده می‌کند:

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v)$$

- به طوری که u رأس شروع و v رأس پایان یال است و h تابعی است که به هر رأس یک عدد حقیقی نسبت می‌دهد.
- در ادامه ثابت خواهیم کرد که این تغییر وزن به ازای هر تابع h کوتاهترین مسیرها را تغییر نمی‌دهد اما برای تبدیل همه وزن‌ها به مقادیر غیر منفی، تابع h باید به درستی تعریف شود.

الگوریتم جانسون

- ابتدا باید ثابت کنیم رابطه تغییر وزن ارائه شده ویژگی اول را ارضاء می‌کند.
- فرض کنید $\langle v_0, v_1, \dots, v_k \rangle = p$ یک مسیر از v_0 به v_k باشد. همچنین وزن کل مسیر p با تابع وزن w را با $w(p)$ نشان می‌دهیم. آنگاه \hat{w} برابر است با:

$$w(p) = (\hat{w}(v_0, v_1) + h(v_0) - h(\hat{v}_1)) + (\hat{w}(v_1, v_2) + h(v_1) - h(v_2)) + \\ (\hat{w}(v_2, v_3) + h(v_2) - h(v_3)) + \dots + (\hat{w}(v_{k-1}, v_k) + h(v_{k-1}) - h(v_k))$$

الگوریتم جانسون

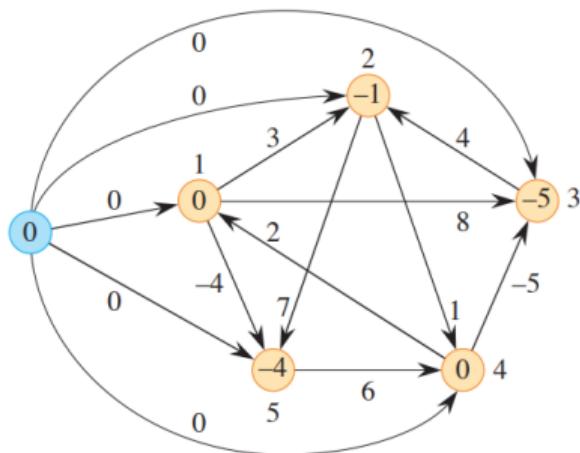
- رابطه بالا نشان می‌دهد برای هر رأس مثل، u مقدار $h(u)$ به وزن یک یال اضافه می‌شود و از یال بعدی کسر می‌شود (به غیر از رأس ابتدا و انتهای مسیر). بعد از ساده‌سازی به این رابطه می‌رسیم:

$$\hat{w}(p) = w(p) + h(h(v_0) - h(v_k))$$

- بنابراین برای هر دو رأس مشخص به همه کوتاهترین مسیرهای بین آن دو رأس یک مقدار ثابت اضافه می‌شود پس این نگاشت وزن کوتاه‌ترین مسیرها را تغییر نمی‌دهد.

الگوریتم جانسون

- هدف بعدی این است که شرایطی فراهم کنیم که ویژگی دوم هم برقرار باشد. برای این کار یک رأس جدید به نام s به گراف اضافه می‌کنیم. سپس این رأس را با یال‌هایی با وزن صفر به همه رئوس دیگر متصل می‌کنیم به طوری که یال‌ها از s خارج شوند.



شکل بالا نمونه‌ایی از این تغییر را نشان می‌دهد (رأس آبی رنگ به گراف اضافه شده است).

الگوریتم جانسون

- سپس تعریف می‌کنیم مقدار h برای هر رأس مثل v برابر است با کوتاه‌ترین مسیر از s به v . در شکل بالا نیز اعداد نوشته شده بر رأس‌ها به همین طریق محاسبه شده‌اند.
- حال با استفاده از نامساوی مثلثاتی می‌دانیم که به ازای هر $(u, v \in E)$ رابطه زیر برقرار است:

$$h(v) \leq h(u) + w(u, v)$$

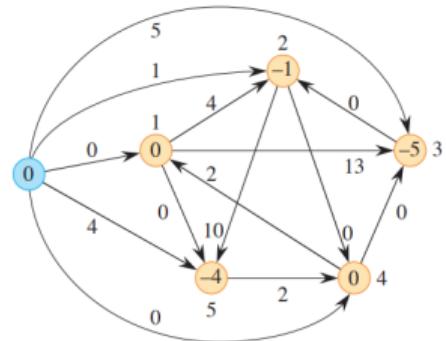
- اگر رابطه بالا برقرار نباشد نتیجه می‌شود که مسیر $h(v)$ کوتاه‌ترین مسیر نبوده و به تناقض می‌رسیم.
- سپس $h(u)$ را به سمت راست نامساوی می‌بریم:

$$0 \leq w(u, v) + h(u) - h(v)$$

- سمت راست این نامساوی همان رابطه‌ایی است که برای \hat{w} تعریف شد. بنابراین ثابت شد که به ازای هر $(u, v \in E)$ مقدار \hat{w} غیر منفی است.

الگوریتم جانسون

- درستی الگوریتم جانسون ثابت شد. شکل زیر یک نمونه از گراف تغییر وزن یافته توسط الگوریتم جانسون است:



- قبل اشاره شد که الگوریتم جانسون از الگوریتم بلمن-فورد برای تشخیص دور منفی استفاده می‌کند. همچنان دیدم که برای محاسبهتابع h نیاز به مقادیر کوتاهترین مسیر از s به بقیه رئوس نیاز داریم. می‌دانیم الگوریتم بلمن-فورد توانایی محاسبه هر دوی این موارد را دارد. بنابراین الگوریتم جانسون با یک بار اجرای بلمن-فورد هم کوتاهترین مسیرها از s را محاسبه می‌کند و هم وجود دور منفی را تشخیص می‌دهد.

الگوریتم جانسون

- برای یافتن وزن اصلی یک مسیر از رابطه زیر استفاده می‌کنیم:

$$w(p(u, v)) = \hat{w}(p(u, v)) + h(v) - h(u)$$

- در نهایت به تحلیل پیچیدگی زمانی الگوریتم جانسون می‌پردازیم:
- اجرای $|V|$ بار الگوریتم دایجسترا زمان‌برترین مرحله الگوریتم جانسون است بنابراین -مانند دایجسترا- باید گراف را در لیست مجاورتی ذخیره کنیم. حال اگر فرض کنیم الگوریتم دایجسترا با استفاده از هرم فیبوناچی پیاده‌سازی شده‌است، پیچیدگی زمانی برابر است با $O(V^2 \lg V + VE)$ که از ضرب یک $|V|$ در مرتبه زمانی دایجسترا به دست آمده است.

- در صورتی که گراف خلوت باشد مرتبه زمانی این الگوریتم برابر است $O(V^2 \lg V)$. در حالی که الگوریتم فلوید-وارشال برای گراف خلوت از مرتبه زمانی $O(V^3)$ است.

مسئله‌های انپی کامل

- تقریباً همه الگوریتم‌هایی تا به حال بررسی کردیم، الگوریتم‌های زمان چند جمله‌ای^۱ بودند بدین معنی که به ازای ورودی با اندازه n ، زمان اجرای آنها در بدترین حالت به ازای هر ثابت k برابر است با $O(n^k)$.
- شاید تصور کنید زمان چند جمله‌ایی مانند $\Theta(n^{100})$ آنقدر بزرگ است که هر مسئله‌ایی در این زمان قابل حل باشد. اما مسائلی وجود دارند که در زمان چند جمله‌ای قابل حل نیستند.
- حتی مسائلی وجود دارند که نه تنها در زمان چند جمله‌ایی، که توسط هیچ الگوریتمی قابل محاسبه نیستند. مانند مسئله توقف ماشین تورینگ^۲ یا مسئله دهم هیلبرت (حل پذیری معادلات سیاله^۳).

¹ polynomial-time algorithm

² halting problem

³ diophantine equations

- وجود راه حل در زمان چند جمله‌ایی برای ما مطلوب است.
- به طوری که مسائلی که در زمان چند جمله‌ای قابل حل‌اند، آسان تلقی می‌شوند و به آنها مسائل قابل کنترل^۱ گفته می‌شود.
- همچنین مسائلی که در این زمان قابل محاسبه نیستند، سخت تلقی می‌شوند و به آنها غیر قابل کنترل^۲ گفته می‌شود.

¹ tractable

² intractable

- اما موضوع این فصل دسته‌ای جالب از مسائل به نام انپی کامل^۱ است که پیچیدگی محاسباتی آن‌ها نامعلوم است.
- هیچ الگوریتم چند جمله‌ای برای این دسته از مسائل تاکنون پیدا نشده است و کسی نتوانسته اثبات کند که هیچ الگوریتم چند جمله‌ای برای آنها وجود ندارد.
- جمله بالا بیانی از مسئله پی در برابر انپی^۲ است؛ یکی از مهمترین مسائل علوم کامپیوتر که در سال ۱۹۷۱ مطرح شده و تا به حال حل نشده باقی مانده است.

^۱ NP-complete problems

^۲ P vs. NP

- برخی از مسائل انپی کامل بسیار شبیه مسائلی هستند که راه حل چند جمله‌ایی برای آنها یافت شده. در ادامه تعدادی از مسائل را می‌بینیم که با وجود شباهت ظاهری‌شان برای یکی از آنها الگوریتم چند جمله‌ای وجود دارد و دیگری انپی کامل است.
- «کوتاهترین مسیر» و «بلندترین مسیر»: با این که این دو مسئله بسیار شبیه یکدیگرند، اما برای مسئله کوتاهترین مسیر در گراف یک الگوریتم چند جمله‌ای وجود دارد که در زمان $O(|V||E|)$ اجرا می‌شود، اما برای مسئله بلندترین مسیر تا به حال الگوریتم چند جمله‌ایی یافته نشده همچنان اثبات نشده که راه حل چند جمله‌ایی برای آن وجود ندارد بنابراین یک مسئله انپی کامل است.

- «دور اویلری» و «دور همیلتونی»: دور اویلری^۱ در یک گراف دوری است که از هر یال دقیقاً یک بار عبور می‌کند. این دور می‌تواند از هر رأس چندبار عبور کند. این مسئله در زمان $O(|E|)$ برای گراف $G(V, E)$ قابل حل است. دور همیلتونی^۲ دوری است که از هر رأس دقیقاً یک بار عبور می‌کند. این مسئله یک مسئله انپی کامل است.

^۱ Eulerian cycle

^۲ Hamiltonian cycle

- «صدق پذیری ۲ تایی» و «صدق پذیری ۳ تایی»: عبارات منطقی شامل متغیرهای منطقی هستند که مقدار آنها می‌تواند صفر یا یک باشد. این متغیرها به عنوان عملوند با تعدادی عملگر منطقی عبارات منطقی را می‌سازند. عملگرهای منطقی شامل عملگر عطف^۱ (۸)، عملگر فصل^۲ (۷) و عملگر نقیض^۳ (~) می‌شوند.
- یک عبارت صدق پذیر^۴ است اگر با انتساب مقادیر صفر و یک به متغیرهای عبارت، مقدار عبارت برابر با ۱ شود. یک عبارت به صورت فرم نرمال عطفی k تایی است اگر آن عبارت از عطف چندین عبارت تشکیل شده باشد که هر یک از آن عبارات از فصل k متغیر یا نقیض متغیر تشکیل شده باشند.
- مسئله صدق پذیری ۲ تایی در واقع تعیین صدق پذیری یک عبارت به صورت فرم نرمال ۲ تایی است. گرچه برای مسئله صدق پذیری ۲ تایی الگوریتم چندجمله‌ای وجود دارد، اما صدق پذیری ۳ تایی انپی کامل است.

¹ conjunction

² disjunction

³ negation

⁴ satisfiable

- در طول این فصل با این سه دسته مسئله سروکار خواهیم داشت: پی، انپی و انپی کامل. در این بخش به صورت مختصر با این کلاس‌های آشنا خواهیم شد. در آینده این کلاس‌ها را دقیق‌تر تعریف و بررسی می‌کنیم.
- پی مخفف زمان چند جمله‌ایی است^۱ و کلاس پی^۲ شامل مسائلی است که در زمان چند جمله‌ای قابل حل هستند.
- انپی مخفف چند جمله‌ایی غیر قطعی^۳ است و کلاس انپی^۴ شامل مسائلی است که در زمان چند جمله‌ای قابل تصدیق^۵ هستند، بدین معنی که اگر یک پاسخ برای مسئله پیشنهاد شود، در زمان چند جمله‌ای می‌توان بررسی کرد پاسخ صحیح است یا خیر. به این پاسخ پیشنهادی گواه^۶ گفته می‌شود.

¹ polynomial time

² class P

³ Non-deterministic Polynomial time

⁴ class NP

⁵ verifiable

⁶ certificate

- برای مثال اگر برای یک گراف یک دور داده شود، می‌توان در زمان چند جمله‌ای بررسی کرد آیا دور داده شده همیلتونی است یا خیر. کافیست بررسی کنیم همه رأس‌ها دقیقاً یک بار در دور وجود داشته باشند. پس مسئله دور همیلتونی انپی است.
- هر مسئله در کلاس انپی به کلاس انپی نیز تعلق دارد، زیرا اگر یک مسئله در زمان چند جمله‌ای قابل حل باشد، در زمان چند جمله تصدیق‌پذیر نیز هست. بنابراین می‌توان بگوییم $P \subseteq NP$.
- مسئله مشهور پی در مقابل انپی^۱ می‌پرسد آیا کلاس پی و انپی برابرند یا خیر. به عبارت دیگر «آیا مسائلی که در زمان چند جمله‌ای قابل تصدیق هستند، در زمان چند جمله‌ای قابل حل هستند؟»

^۱ P vs NP

- یک مسئله به دستهٔ مسائل انپی کامل^۱ تعلق دارد اگر عضو کلاس انپی باشد و جزو سخت‌ترین مسائل این کلاس باشد. در آینده منظور از این تعریف را بیشتر شرح خواهیم داد.
- مسائل NP کامل ویژگی مهمی دارند: اگر یکی از این مسائل در زمان چند جمله‌ایی حل شود نتیجه می‌شود که همه آنها در زمان چند جمله‌ایی حل می‌شوند.
- بنابراین دانشمندان علوم کامپیوتر در بسیاری در طول تاریخ تلاش کردند تا یک راه حل چند جمله‌ایی حداقل برای یکی از این مسائل بیابند اما هیچ الگوریتم چند جمله‌ایی برای این مسائل یافت نشده.
- از طرف دیگر برای هیچ کدام از مسائل ثابت نشده که راه حل چند جمله‌ایی برای آن وجود ندارند.

^۱ NP-complete

- اما شناخت مسائل انپی کامل چه کمکی به ما می‌کند؟
- به عنوان یک طراح الگوریتم، اگر بتوانید ثابت کنید که یک مسئله انپی کامل است، به احتمال زیاد به راحتی برای آن الگوریتم چندجمله‌ای پیدا نخواهید کرد، بنابراین بهتر است تمرکز خود را بر روی پیدا کردن یک الگوریتم تقریبی خوب بگذارید یا مسئله را برای حالت‌های خاص حل کنید.
- به عنوان یک مهندس اگر ثابت کنید یک مسئله انپی کامل است، در واقع می‌توانید سختی آن مسئله را نشان دهید و نشان دهید جستجو برای یک الگوریتم چندجمله‌ای به احتمال بسیار زیاد بی‌ثمر خواهد بود.

- اگر بتوانیم نشان دهیم یک مسئله کامل-NP است، می‌توانیم درباره «سخت بودن» آن مسئله اظهارنظر می‌کنیم. زیرا در این صورت، جستجو برای یافتن الگوریتمی کارا برای آن احتمالاً بی‌ثمر خواهد بود.
- بنابراین اثبات انپی کامل بودن حائز اهمیت است. برای نشان دادن اینکه یک مسئله انپی کامل است، از این سه مفهوم استفاده می‌کنیم: ۱) مسائل تصمیم‌گیری ۲) کاهش ۳) اهمیت یافتن اولین مسئله انپی کامل در ادامه این مفاهیم را معرفی خواهیم کرد.

مسائل تصمیم‌گیری

- بسیاری از مسئله‌ها، مسائل بهینه‌سازی^۱ هستند و هدف از حل این مسائل یافتن مقداری است که از جواب‌های دیگر بهتر (کوچکتر، بزرگتر، ...) است. برای مثال مسئله کوتاهترین مسیر، یک مسئله بهینه‌سازی است، زیرا هدف یافتن مسیری با طول کمینه است.
- مسائل انپی کامل مسائل بهینه‌سازی نیستند، بلکه مسائل تصمیم‌گیری^۲ هستند. در این‌گونه مسائل جواب بله و خیر است.
- با این حال مسائل بهینه‌سازی قابل تبدیل به مسائل تصمیم‌گیری هستند. برای مثال نسخه تصمیم‌گیری مسئله کوتاهترین مسیر از u به v این چنین است: آیا مسیری از u به v با طول کمتر یا مساوی با k وجود دارد یا خیر.
- سختی یک مسئله بهینه‌سازی بیشتر یا مساوی با سختی مسئله تصمیم‌گیری متناظر است. پس اگر نشان دهیم یک مسئله تصمیم‌گیری انپی کامل است، اظهار نظرهای مفیدی در مورد مسئله بهینه‌سازی متناظر نیز انجام داد.

¹ optimization problem

² decision problem

- همینطور که دیدیم مقایسه سختی مسائل حائز اهمیت است. روش کاہش ابزار مناسبی برای مقایسه سختی مسائل تصمیم‌گیری است.
- معمولًاً یک مسئله در حالت کلی توسط تعدادی پارامتر بیان می‌شود. وقتی پارامترهای یک مسئله را مقداردهی می‌کنیم در واقع یک نمونه^۱ از مسئله را به دست آورده‌ایم.
- برای مثال مسئله کوتاهترین مسیر می‌پرسد به ازای یک گراف دلخواه و دو رأس ۱۱ و ۷ چگونه کوتاهترین مسیر را در حالت کلی پیدا کنیم. یک نمونه از مسئله درواقع یک گراف معین و دو رأس ورودی و خروجی معین است.

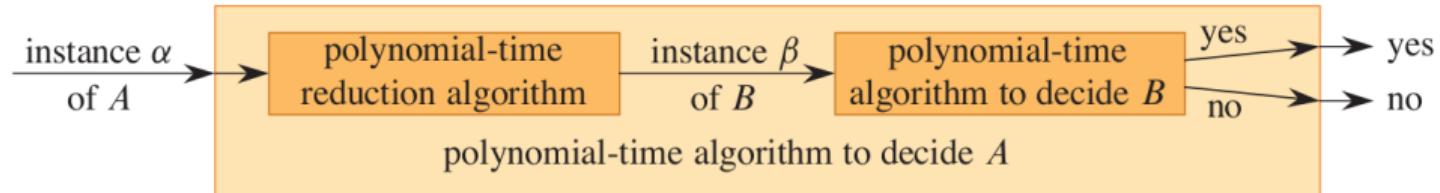
^۱ instance

- مسئله تصمیم‌گیری A را در نظر بگیرید که می‌خواهید آن را در زمان چندجمله‌ای حل کنید.
- فرض کنید می‌دانید چگونه مسئله B را در زمان چندجمله‌ای حل کنید.
- همچنین فرض کنید می‌دانید چگونه هر نمونه α از مسئله A را به نمونه β از مسئله B تبدیل کنید به طوری که:
 - تبدیل نمونه α به نمونه β در زمان چندجمله‌ای انجام شود.
 - جواب α بله است اگر و تنها اگر جواب β بله باشد.
- چنین روندی برای تبدیل مسئله‌های تصمیم‌گیری به یکدیگر را الگوریتم کاهش در زمان چندجمله‌ای^۱ می‌نامیم.
- اگر B در زمان چندجمله‌ای قابل حل باشد و A در زمان چندجمله‌ای قابل کاهش به B باشد، آنگاه برای حل مسئله A در زمان چندجمله‌ای کافی است آن را در زمان چندجمله‌ای به B کاهش دهیم و سپس B را در زمان چندجمله‌ای حل کنیم.

¹ polynominal time reduction algorithm

- الگوریتم کاهش در زمان چندجمله‌ای در واقع روشی برای حل مسئله A در زمان چندجمله‌ای است.
 ۱. به ازای نمونه α از مسئله A ، از الگوریتم کاهش در زمان چندجمله‌ای برای تبدیل آن به نمونه β از مسئله B استفاده می‌کنیم.
 ۲. مسئله تصمیم‌گیری B را برای نمونه β در زمان چندجمله‌ای حل می‌کنیم.
 ۳. جواب نمونه β را به عنوان جواب نمونه α در نظر می‌گیریم.

- شکل زیر الگوریتم کاہش برای حل یک مسئله را نشان می‌دهد.



- پس با استفاده از روش کاهش برای حل مسئله A از الگوریتم چندجمله‌ای مسئله B استفاده می‌کنیم.
- به طور کلی اگر A به B کاهش یابد نتیجه می‌شود مقدار سختی B بیشتر یا مساوی با A است.
- از همین ایده استفاده می‌کنیم برای اینکه نشان دهیم یک مسئله به سختی یک مسئله دیگر است.

- فرض کنید ثابت شده است که برای مسئله A الگوریتم چندجمله‌ای وجود ندارد. حال فرض کنید یک الگوریتم کاهش در زمان چندجمله‌ای برای تبدیل مسئله A به مسئله B وجود دارد. می‌توانیم با استفاده از برهان خلف نشان دهیم که هیچ الگوریتم چندجمله‌ای برای B وجود ندارد.
- برای اثبات این قضیه فرض کنید برای B یک الگوریتم چندجمله‌ای وجود داشته باشد، در آن صورت می‌توانستیم با استفاده از کاهش، یک الگوریتم چندجمله‌ای برای A پیدا کنیم، اما می‌دانیم ثابت شده است که الگوریتم چندجمله‌ای برای A وجود ندارد. پس به تناقض می‌رسیم و در نتیجه فرض اولیه نادرست بوده و الگوریتم چندجمله‌ای برای B وجود ندارد.

یافتن اولین مسئله انپی کامل

- برای اینکه نشان دهیم یک مسئله انپی کامل است نیز از همین روند استفاده می‌کنیم. برای اینکار باید یکی از مسائل انپی کامل را در زمان چندجمله‌ای به مسئله مدنظر کاهش دهیم.
- روش مسقیم اثبات انپی کامل بودن این است که از همه مسائل کلاس انپی به آن کاهش انجام دهیم. می‌توانید حدس بزنید این کار چقدر مشکل است؛ تعداد مسائل انپی کامل نامتناهی است و تبدیل هر کدام به یک مسئله دیگر نیز کار مشکلی است.
- اما اگر تنها یک مسئله ثابت انپی کامل داشته باشیم برای اثبات انپی کامل بودن مسائل بعدی کافیست از مسئله اثبات شده به مسئله جدید یک کاهش بیابیم.

یافتن اولین مسئله انپی کامل

- اولین مسئله‌ای که انپی کامل بودن آن اثبات شده است، مسئله ارضا پذیری است. اثباتی وجود دارد که نشان می‌دهد همه مسائل انپی در زمان چندجمله‌ای قابل کاهش به مسئله صدق‌پذیری هستند. بنابراین اگر مسئله صدق‌پذیری در زمان چندجمله‌ای حل شود، همه مسائل انپی در زمان چندجمله‌ای حل خواهد شد. به عبارت دیگر مسئله صدق‌پذیری به سختی همه مسائل انپی است.
- در واقع هر یک از مسائل انپی کامل به سختی همه مسائل انپی هستند، زیرا حل یکی از آنها در زمان چندجمله‌ای منجر به حل همه مسائل انپی می‌شود.

- در طی این بحث رده‌های مختلفی از مسائل سروکار داریم که ساده‌ترین این رده P است. P مجموعه مسائلی است که در زمان چند جمله‌ایی قابل حل اند. در این بخش می‌خواهیم این رده مسائل را دقیق‌تر بررسی کنیم.
- ابتدا بباید مفهوم قابل حل بودن در زمان چند جمله‌ایی را بررسی کنیم. به‌طور کلی، مسائلی که دارای راه حل‌هایی با زمان چند جمله‌ای هستند را آسان در نظر می‌گیریم.
- اگرچه مسئله‌ای را که زمان حل آن از مرتبه $(n^{100})^{\Omega}$ باشد، نمی‌توان آسان برشمرد اما در عمل اکثر مسائلی که با زمان چند جمله‌ای قابل حل اند، زمان بسیار کمتری می‌طلبند.

- به علاوه تجربه نشان داده که زمانی که اولین الگوریتم چندجمله‌ای برای یک مسئله پیدا می‌شود، معمولاً الگوریتم‌های کارآمدتری نیز در پی آن کشف می‌گردند. به علاوه اگر مسئله‌ایی در زمان چند جمله‌ایی در یکی از مدل‌های محاسباتی حل شود توسط اکثر مدل‌های محاسباتی دیگر نیز در همین زمان حل می‌شود.
- برای مثال کلاس مسائل قابل حل در زمان چند جمله‌ایی توسط ماشین تورینگ و ماشین دارای حافظه دسترسی مستقیم یکسان است. یادآوری می‌شود که برای دسترسی حافظه در ماشین تورینگ باید همه خانه‌های قبلی پیمایش شوند.
- در این بخش هدف بررسی دقیق‌تر کلاس P است. برای اینکار ابتدا باید تعدادی مفاهیم مرتبط را معرفی کنیم.

- یک «مسئله انتزاعی»^۱ یک رابطه بین دو مجموعه I (نمونه‌های مسئله) و S (پاسخ نمونه‌ها) است. همانطور که گفته شد در این مبحث تنها با مسائل تصمیم‌گیری سروکار داریم. در یک مسئله تصمیم‌گیری مجموعه S تنها شامل دو عضو است؛ ° و ۱.
- مسئله انتزاعی در واقع همان چیزیست که ما به عنوان مسئله معمولی می‌شناسیم. از این جهت به آنها انتزاعی می‌گوییم که به خودی خود برای کامپیوتر قابل فهم نیستند. به بیان دیگر هنوز تعریف نکرده‌ایم که اجزای مجموعه I دقیقاً چیستند.

^۱ abstract problem

- برای اینکه یک نمونه مسئله به عنوان ورودی به یک برنامه کامپیوتری داده شود اول باید آن را تبدیل به رشته کنیم که کامپیوتر متوجه آن شود.
- به این کار کد گذاری¹ مسئله گفته می‌شود. به طور دقیق‌تر کد گذاری یک نگاشت نمونه‌های یک مسئله (مجموعه S) به مجموعه‌ایی از رشته‌های باینری است.
- به مسئله‌ایی که اعضای مجموعه S آن مسائل کد گذاری نشده باشند «مسئله‌ی انتزاعی» و زمانی که S را مجموعه‌ایی از رشته‌های باینری قرار دهیم به آن «مسئله‌ی عینی»² گفته می‌شود.
- برای مثال تبدیل اعداد با سیستم دودویی و کارکترها با سیستم ASCII هر کدام از انواع کد گذاری‌اند.

¹ encoding

² concrete problem

- یک مسئله عینی در زمان چند جمله‌ای قابل حل^۱ است، اگر الگوریتمی وجود داشته باشد که در زمان $O(n^k)$ پاسخ درست مسئله را تولید کند. به طوری که k مقداری ثابت و n طول رشته کدگذاری شده است.
- اکنون می‌توانیم کلاس پیچیدگی P رسماً تعریف کنیم؛ مجموعه‌ای از مسائل تصمیم‌گیری عینی که در زمان چندجمله‌ای قابل حل هستند.
- اما برای ما کار کردن با مسائل انتزاعی ساده‌تر است بنابراین می‌خواهیم حل پذیری در زمان چند جمله‌ایی را روی مسائل انتزاعی نیز تعریف کنیم.

^۱ polynomial-time solvable

- بهتر است این تعریف مستقل از نحوه کدگذاری مسئله باشد. همانطور که می‌دانید ما معمولاً در مورد اینکه یک مسئله را می‌توان در چه زمانی حل کرد اظهار نظر می‌کنیم بدون اینکه کدگذاری مدنظر خود را معرفی کنیم.
- متأسفانه این کار چندان هم ممکن نیست. برای مثال فرض کنید یک الگوریتم تنها یک عدد w ورودی می‌گیرد. می‌توانیم این عدد را به صورت یگانی¹ یا به صورت دودویی کدگذاری کنیم. (در کدگذاری یگانی w تا ۱ در رشته قرار می‌گیرد).
- فرض کنید برای هر دو الگوریتمی از مرتبه خطی نسبت به اندازه ورودی موجود باشد بنابراین پیچیدگی زمانی هر دو را با $\Theta(n)$ نشان می‌دهیم. اما این دو پیچیدگی زمانی در واقع برابر نیستند. زیرا اندازه n در کدگذاری یگانی نسبت به کدگذاری دودویی نمایی است.

¹ unary

- دیدیم که تاثیر کدگذاری بر تعریف ما از زمان چند جمله‌ایی اجتناب ناپذیر است. اما در عمل اصلاً کدگذاری‌های بیش از حد طولانی مانند یگانی را در نظر نمی‌گیریم.
- تفاوت بین بقیه کدگذاری‌های معقول هم تأثیر کوچکی بر اینکه آیا مسئله در زمان چند جمله‌ایی حل می‌شود یا خیر می‌گذارد.
- برای مثال نمایش در مبنای ۲ و یا مبنای ۳ تفاوتی در چند جمله‌ایی بودن مسئله نمی‌گذارد زیرا می‌توانیم در زمان چندجمله‌ایی مبنای ۳ را به مبنای ۲ تبدیل کنیم.

- برای یک مجموعه از نمونه‌های مسئله به نام، می‌گوییم دو کدگذاری با هم ارتباط چندجمله‌ای^۱ دارند، اگر دو تابع وجود داشته باشند که در زمان چندجمله‌ای این دو کدگذاری را به یکدیگر تبدیل کنند.
- اگر دو کدگذاری از یک مسئله انتزاعی با یکدیگر ارتباط چندجمله‌ای داشته باشند، آنگاه این که آیا آن مسئله در زمان چندجمله‌ای قابل حل است یا نه، مستقل از انتخاب هریک از این دو کدگذاری خواهد بود.
- به طور کلی فرض می‌کنیم کدگذاری مسئله «استاندارد» است یعنی کدگذاری اعداد صحیح ارتباط چندجمله‌ایی به کدگذاری باینری داشته باشد و کدگذاری لیست‌ها ارتباط مستقیم با کدگذاری به صورت $\{item1, item2, item3, item4\}$ داشته باشد.

^۱ polynomially related

- با داشتن این کدگذاری استاندارد می‌توانیم یک کدگذاری معقول برای بقیه تعاریف ریاضی مانند گراف، تاپل، درخت و غیره نیز در نظر گرفت.
- نماد کدگذاری استاندارد $\langle \rangle$ است. برای مثال اگر G یک گراف باشد $\langle G \rangle$ کدگذاری استاندارد G را نشان می‌دهد.
- وقتی فرض کنیم که کدگذاری استفاده شده ارتباط چند جمله‌ایی با کدگذاری استاندارد دارد، می‌توانیم مستقیماً در مورد مسائل انتزاعی بحث کنیم. از این به بعد فرض می‌کنیم که همه مسائل به صورت استاندارد کدگذاری شده‌اند. همچنین معمولاً از تفاوت بین مسائل انتزاعی و عینی صرف نظر می‌کنیم.

- گفتیم که مسائلی که در این مبحث روی آنها کار می‌کنیم را به مسائل تصمیم‌گیری محدود می‌کنیم. با این فرض می‌توانیم از مفاهیم نظریه زبان‌های صوری^۱ استفاده کنیم. این نظریه تحت عنوان نظریه زبان‌ها و ماشین‌ها^۲ نیز شناخته می‌شود.
- دیدگاه این نظریه نسبت به مسئله متفاوت است. در واقع در این نظریه هر مسئله یک زبان صوری^۳ است و زبان‌ها نیز مجموعه‌ایی از رشته‌ها روی یک الفبای مشخص هستند.
- از این دیدگاه در آینده بهره خواهیم برد. بنابراین اگر با این نظریه آشنایی ندارید بهتر است پیوست شماره یک را مطالعه کنید.

¹ formal-language theory

² formal language and automata theory

³ formal language

کلاس پی

مروری بر نظریه زبان‌ها صوری

- پس از آشنایی با نظریه زبان‌های صوری می‌توانیم به صورت یک کلاس پیچیدگی^۱ را به این صورت تعریف کنیم: مجموعه‌ای از زبان‌ها که عضویت در آن‌ها توسط یک معیار پیچیدگی^۲ (مانند زمان اجرا) برای یک الگوریتم تعیین می‌شود؛ الگوریتمی که مشخص می‌کند آیا یک رشته‌ی داده‌شده x متعلق به یک زبان L هست یا خیر.
- البته تعریف دقیق رده‌ی پیچیدگی، اندکی فنی‌تر از این بیان غیررسمی است.

^۱ complexity class

^۲ complexity measure

- با استفاده از این چارچوب مبتنی بر زبان‌های صوری، می‌توانیم تعریف دیگری از رده پیچیدگی P ارائه دهیم:

$P = \{L \subseteq \{0, 1\}^*: \text{time polynomial in } L \text{ decides that A algorithm an exists there}\}$

- همچنین قضیه‌ایی وجود دارد که تعریف زیر را هم برای P ثابت می‌کند:

$P = \{L : \text{algorithm polynomial-time a by accepted is } L \text{ W}\}$

این تعریف برابر است با مجموعه زبان‌هایی که در زمان چندجمله‌ای پذیرفته می‌شوند.

کلاس انپی

- اکنون بیایید به الگوریتم‌هایی نگاه کنیم که «عضویت در زبان‌ها» را بررسی می‌کنند. برای مثال، فرض کنید برای یک نمونه $\langle G, u, v, k \rangle$ از مسئله‌ی تصمیم‌گیری، PATH یک مسیر p از u به v نیز در اختیار دارد. می‌توانید بررسی کنید که آیا p واقعاً یک مسیر در گراف G است یا نه، و همچنین اینکه آیا طول مسیر p حداقل k است یا خیر.
 - اگر این دو شرط برقرار باشند، می‌توان p را به عنوان یک گواه^۱ در نظر گرفت که نشان می‌دهد این نمونه واقعاً متعلق به زبان PATH است.
 - هرچند برای مسئله‌ای که عضو پی باشد گواه سود چندانی ندارد. در عوض، بیایید به مسئله‌ای بپردازیم که الگوریتم تصمیم‌گیرنده‌ایی با زمان چندجمله‌ای برای آن نمی‌شناسیم. اما اگر یک گواه در اختیار داشته باشیم، کار آسان‌تر است.

1 certificate

- یک دور همیلتونی در یک گراف بدون جهت دوری ساده است که هر رأس از مجموعه V را دقیقاً یک بار طی می‌کند. گرافی که دارای یک دور همیلتونی باشد، گراف همیلتونی^۱ نامیده می‌شود و در غیر این صورت، غیرهمیلتونی^۲ است.
- مسئله دور همیلتونی ذاتاً یک مسئله تصمیم‌گیری است که می‌پرسد: «آیا گراف داده شده دارای دور همیلتونی است؟». تعریف مسئله دور همیلتونی به صورت یک زبان رسمی صوری به این صورت است:

$$\text{HAM-CYCLE} = \{\langle G \rangle : \text{graph hamiltonian a is } G\}$$

^۱ hamiltonian

^۲ nonhamiltonian

کلاس انپی

الگوریتم‌های تصدیق

- حال به مسئله‌ای کمی ساده‌تر توجه کنید. فرض کنید دوستی به شما می‌گوید که گرافی داده‌شده همیلتونی است، و سپس برای اثبات این ادعا، ترتیب رأس‌ها در یک دور همیلتونی را ارائه می‌دهد.
- قطعاً بررسی این اثبات آسان است: کافیست بررسی کنید آیا این دور یک جایگشت از رأس‌های گراف است و آیا هر یک از یال‌های متوالی در دور واقعاً در گراف وجود دارند یا نه.
- می‌توان این الگوریتم بررسی را به گونه‌ای پیاده‌سازی کرد که در زمان $O(n^2)$ اجرا شود.

کلاس انپی

الگوریتم‌های تصدیق

- ما یک الگوریتم تصدیق^۱ را به صورت یک الگوریتم دوپارامتری A تعریف می‌کنیم، که یکی از پارامترهای آن همان رشته‌ای ورودی است که به آن x می‌گوییم و پارامتر دیگر رشته‌ای دودویی به نام گواه^۲ است که با y نشان داده می‌شود.
- در صورتی گواه صحیح باشد الگوریتم A می‌تواند ثابت کند که x عضو آن زبان است و مقدار ۱ را باز می‌گرداند: $A(x, y) = 1$
- زبانی که توسط یک الگوریتم بررسی A تصدیق می‌شود به صورت زیر تعریف می‌شود:

$$L = \{x \in \{0, 1\}^*: \text{there exists } y \in \{0, 1\}^* \text{ such that } A(x, y) = 1\}$$

^۱ verification algorithm

^۲ certificate

- کلاس پیچیدگی انپی شامل زبان‌هایی است که می‌توان آنها را توسط یک الگوریتم با زمان چندجمله‌ای تصدیق کرد. به عبارت دقیق‌تر، یک زبان L متعلق به کلاس NP است اگر و تنها اگر الگوریتمی دوورودی به نام A با زمان اجرای چندجمله‌ای و عدد ثابتی c وجود داشته باشد، به طوری‌که:

$$L = \{x \in \{0,1\}^*: \text{with } y \text{ certificate } a \text{ exists there } |y| = O(|x|^c) \text{ such that } A(x,y) = 1\}$$

- در این صورت می‌گوییم که الگوریتم A زبان L را در زمان چندجمله‌ای «تصدیق» می‌کند.

- از بحث قبلی ما درباره مسئله دور همیلتونی، می‌توان دریافت که:

$$HAM - CYCLE \in NP$$

علاوه بر این، اگر زبانی عضو پی باشد، آنگاه عضو انپی نیز خواهد بود.

- زیرا اگر الگوریتمی با زمان چندجمله‌ای برای تصمیم‌گیری زبان L وجود داشته باشد، می‌توان آن را به یک الگوریتم تصدیق دوورودی تبدیل کرد که گواهی را نادیده می‌گیرد. سپس محاسبات خودش را انجام می‌دهد تا مشخص کند ورودی متعلق به L است یا خیر. سپس دقیقاً رشتۀایی را می‌پذیرد که متعلق به زبان مدنظر باشند.

کلاس انپی

- بنابراین دریافتیم

$$P \subseteq NP$$

اما آیا

$$P \subset NP$$

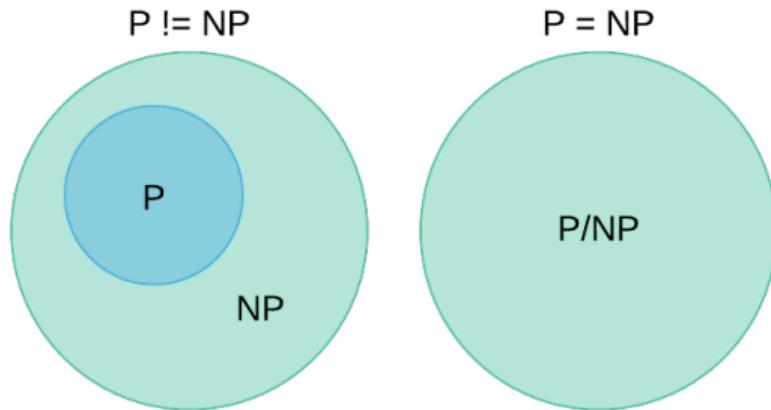
برقرار است یا؟

$$P = NP$$

- این طور به تفاوت این دو کلاس نگاه کنید: کلاس پی شامل مسائلی است که در زمان کوتاه قابل حل و انپی شامل مسائلی است که در زمان کوتاه قابل تصدیق هستند.
- شاید شما طبق تجربه عنوان کنید که تصدیق گواه بسیار از به دست آوردن پاسخ ساده‌تر است. بسیاری از دانشمندان علوم کامپیوتر نیز بر این باوراند.

کلاس انپی

- نکته قابل توجه اینجاست که تا به حال این مسئله ثابت نشده است. یعنی هیچ مسئله‌ای نمی‌شناسیم که در زمان چند جمله‌ایی قابل ارزیابی باشد اما ثابت شود در زمان چند جمله‌ایی قابل حل نیست.



- در صورتی که چنین مسئله‌ایی پیدا شود در فضای $P - NP$ قرار می‌گیرد و ثابت می‌شود $P \neq NP$. به هر حال هر دو حالت شکل بالا در حال حاضر ممکن است صحیح باشد.

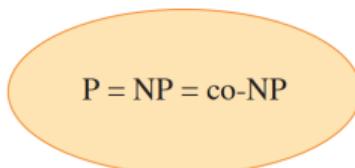
کلاس انپی

- مسئله $P \neq NP$ تنها مسئله حل نشده در این مبحث نیست. برخلاف کارها زیادی که در این زمینه انجام شده هنوز مشخص نیست که آیا کلاس انپی تحت عملگر مکمل بسته است یا خیر. به این معنی که اگر L عضو پی باشد، \bar{L} نیز عضو انپی است.
- دسته مسائل مکمل کلاس انپی به اختصار $co\text{-}NP$ گفته میشود که co مخفف complement است.
- کلاس کو-انپی به این صورت تعریف میشود: همه زبانهایی مانند L به طوری که $\bar{L} \in NP$.

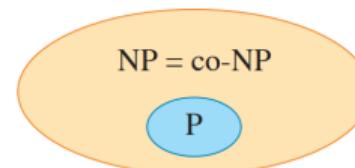
کلاس انپی

- به علاوه می‌دانیم کلاس پی نیز تحت مکمل‌گیری بسته است به این معنی که اگر L در کلاس پی باشد \bar{L} نیز در این کلاس است. بنابراین پی زیر مجموعه انپی نیز هست زیرا مکمل همه اعضای آن در انپی حضور دارند.
- دریافتیم که کلاس پی زیر مجموعه اشتراک انپی و کو-انپی است: $(NP \cup co-NP) \subset P$ ‌ی. اما آیا مجموعه پی کل فضای اشتراک بین انپی و کو-انپی را در بر می‌گیرد؟ به عبارت دیگر آیا زبانی در $(NP \cup co-NP) - P$ وجود دارد؟
- سوال دیگر یکی دیگر از سوالات پاسخ داده نشده است. می‌توانید حدس بزنید با وجود چندین سوال که پاسخ آنها مشخص نباشد می‌بایستی چندین نمایش مجموعه‌ایی (نمودار ون) برای هر یک از نظریه‌های محتمل رسم کرد. در شکل زیر چهار حالت محتمل رسم شده است.

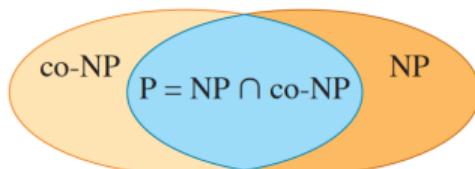
- چهار حالت ممکن برای روابط بین کلاس‌های پیچیدگی زمانی بدین شرح است:



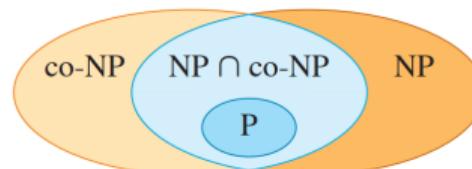
(a)



(b)



(c)



(d)

کلاس انپی

شکل a کلاس P زیرمجموعهٔ محض NP و NP زیرمجموعهٔ محض co-NP است. بیشتر پژوهشگران این احتمال را بسیار بعید می‌دانند.

شکل b اگر NP تحت مکمل بسته باشد، آنگاه NP برابر با co-NP^* است، اما لزوماً P با NP برابر نیست.

شکل c کلاس P زیرمجموعهٔ خاص اشتراک NP و co-NP است، اما NP تحت متمم بسته نیست.

شکل d کلاس NP برابر با co-NP نیست و همچنین P زیر مجموعهٔ محض اشتراک NP و co-NP است. بیشتر پژوهشگران این احتمال را محتمل‌ترین حالت می‌دانند.

- بنابراین، درک ما از رابطه دقیق میان پی و انپی به طور نامیدکننده‌ای ناقص است. اگر یک مسئله در $NP - P$ قرار گیرد، ممکن است نتوانیم ثابت کنیم این مسئله سخت است یا در زمان چند جمله‌ای قابل حل است.
- با این حال اگر بتوانیم ثابت کنیم که آن مسئله انپی کامل است، آنگاه اطلاعات ارزشمندی درباره آن به دست آورده‌ایم. انپی کامل بودن یک مسئله به ما می‌گوید که این مسئله به اندازه همه مسائل موجود در کلاس انپی کامل سخت است.
- بخش آینده در مورد انپی کامل بودن بیشتر صحبت خواهیم کرد.

کلاس انپی کامل

- شاید قانع‌کننده‌ترین دلیلی که می‌بینی بر مخالف بودن کلاس پی و انپی وجود دارد وجود کلاس انپی کامل است. این کلاس ویژگی بسیار جالبی دارد: اگر یکی از اعضای این کلاس در زمان چند جمله‌ایی حل شود آنگاه ثابت می‌شود همه آنها الگوریتم چند جمله‌ایی دارند. و بنابراین پی برابر می‌شود با انپی.
- برخلاف دهه‌ها تلاش محققان برای هیچ کدام از مسائل انپی کامل راه حل چند جمله‌ایی یافت نشده.
- مسائل کلاس انپی کامل «سخت‌ترین» مسائل کلاس انپی است بنابراین اگر ثابت شود که $P = NP$ خالی نیست مسائل دسته انپی کامل به قطع در این فضا قرار می‌گیرند.

کلاس انپی کامل

- در این بخش می‌خواهیم مسائل انپی کامل را به طور دقیق‌تر تعریف کنیم. همانطور که گفته شد این دسته شامل سخت‌ترین مسائل انپی است اما سخت بودن یعنی چه؟ چگونه می‌توان عنوان کرد مسئله‌ایی از دیگری سخت‌تر است؟
- عبارت «سخت بودن»^۱ که چندین بار در این مبحث (از جمله در توضیح کلاس انپی کامل) از آن استفاده کردیم ممکن است مبهم و غیر دقیق به نظر برسد اما این عبارت در واقع تعریف دقیق و مشخصی دارد و هدف آن مقایسه مسائل را از دیدگاه پیچیدگی زمانی است.
- همانطور که قبلًاً اشاره کوتاهی انجام شد، مقایسه سختی مسائل از طریق کاهش به دست می‌آید. بنابراین برای تعریف رسمی کلاس انپی کامل باید ابتدا مبحث کاهش پذیری را دقیق‌تر بررسی کنیم.

¹ hardness

کلاس انپی کامل

کاهش پذیری

- می‌گوییم مسئله A به B کاهش پذیر^۱ است اگر هر نمونه از A مثل a قابلیت تبدیل به یک نمونه از B مانند b را داشته باشد به طوری که با حل نمونه b پاسخ a نیز به دست بیاید.
- برای مثال مسئله حل معادله درجه می‌تواند به حل معادله درجه دو کاهش یابد. فرض کنید $ax + b = 0$ یک نمونه از معادله درجه یک باشد می‌دانیم که پاسخ آن $x = -b/a$ است. آنگاه با حل این معادله $ax^2 + bx + 0 = 0$ می‌توان جواب معادله درجه یک اولیه را به دست آورد. (با توجه به اینکه ۰ صفر است جواب‌های این معادله عبارت‌اند از $(x_1 = 0, x_2 = -b/a)$)
- نتیجه کاهش مسئله A به B در حالت کلی این است که A نمی‌تواند از B سخت‌تر باشد. همانطور که در این مثال از انجام این کاهش می‌توان نتیجه گرفت که حل مسئله معادله درجه یک نمی‌تواند از درجه دو سخت‌تر باشد.

¹ reducible

کلاس انپی کامل

کاہش پذیری

- با استفاده از چارچوبی که در تئوری زبان‌های صوری با آن آشنا شدیم می‌توانیم مفهوم مهمی تحت عنوان «کاہش پذیری در زمان چند جمله‌ایی»^۱ را تعریف کنیم.
- می‌گوییم زبان L_1 در زمان چند جمله‌ایی به L_2 کاہش پذیری است اگر تابعی مثل f وجود داشته باشد که در زمان چند جمله‌ایی نمونه‌های L_1 را به نمونه‌های L_2 تبدیل کند. آنگاه به این تابع، تابع کاہش^۲ و به الگوریتم آن الگوریتم کاہش^۳ گفته می‌شود.
- اگر زبان L_1 در زمان چند جمله‌ایی به L_2 کاہش یابد، آن را به این صورت می‌نویسیم:

$$L_1 \leq_p L_2$$

این نحو نوشتار به خوبی گویای مفهوم کاہش است.

^۱ polynomial-time reducible

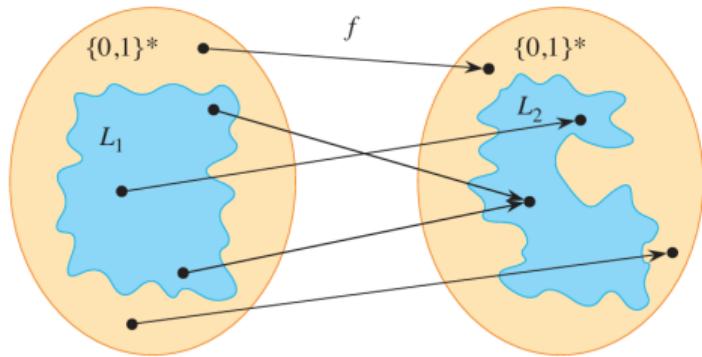
^۲ reduction function

^۳ reduction algorithm

کلاس انپی کامل

کاهش پذیری

- یک تابع کاهش به این صورت زبانی مثل L_1 را به زبان دیگر مانند L_2 کاهش می‌دهد:



- در این شکل تابع f هر عضو از زبان L_1 را به یک عضو از L_2 نگاشت می‌کند و در صورتی که زبانی عضو L_1 نباشد ($f(L_1)$ عضو L_2 نخواهد بود).

کلاس انپی کامل

کاهش پذیری

- کاهش چندجمله‌ای ابزاری قدرتمندی برای اثبات تعلق به کلاس P در اختیار ما قرار می‌دهد.
- به طوری که اگر $L_1, L_2 \subseteq 0, 1^*$ دو زبان باشند به‌طوری‌که $L_1 \leq_P L_2$ ، آنگاه $L_2 \in P$ نتیجه می‌دهد که $L_1 \in P$ نیز برقرار است.

کلاس انپی کامل

تعریف انپی کامل بودن

- با استفاده از کاهش چندجمله‌ای می‌توانیم نشان دهیم یک مسئله حداقل به سختی مسئله‌ای دیگر است، تا حدی که تفاوت سختی آن‌ها تنها در یک چندجمله‌ای باشد.
- به بیان دقیق‌تر، اگر $L_2 \leq_P L_1$ ، آنگاه L_1 حداکثر به اندازه یک مقدار چندجمله‌ای سخت‌تر از L_2 است.
نماد « \leq » در کاهش به خوبی نشان دهنده نسبت سختی دو مسئله است.
- اکنون می‌توانیم مجموعه زبان‌های انپی کامل را تعریف کنیم، که سخت‌ترین مسائل در کلاس NP هستند.
یک زبان مانند L انپی کامل است اگر:

$$L \in NP \quad 1$$

از همه زبان‌های عضو NP به L کاهش وجود داشته باشد.

کلاس انپی کامل

تعریف انپی کامل بودن

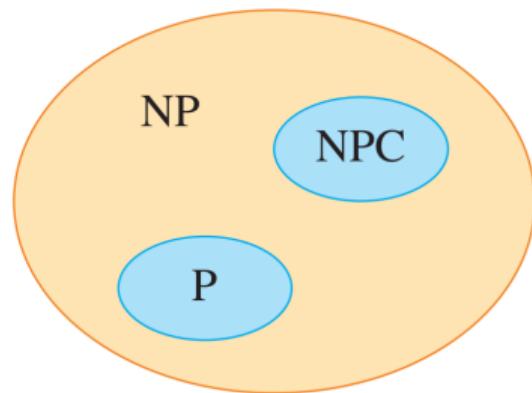
- اگر یک زبان L خاصیت ۲ را داشته باشد، اما الزاماً خاصیت ۱ را نداشته باشد، می‌گوییم که L انپی سخت^۱ است.
- انپی کامل بودن نقش مهمی در مسئله $P = NP$ دارند. در زیر دلایل این مسئله را بررسی می‌کنیم:
 - اگر هرکدام از مسئله انپی کامل در زمان چندجمله‌ای قابل حل باشد، ثابت می‌شود $P = NP$.
 - اگر یک مسئله مانند A در NP وجود داشته باشد که ثابت شود در زمان چندجمله‌ای قابل حل نیست، آنگاه هیچ مسئله انپی کاملی در زمان چندجمله‌ای قابل حل نخواهد بود. زیرا A عضو انپی است پس سختی آن از همه مسائل انپی کامل بیشتر یا مساوی با آنهاست.

^۱ NP-hard

کلاس انپی کامل

تعریف انپی کامل بودن

- به همین دلیل است که پژوهش‌ها درباره پرسش $P \stackrel{?}{=} NP$ عمدتاً بر مسائل انپی کامل متمرکز هستند.
- بیشتر نظریه‌پردازان علوم کامپیوتر بر این باورند که $P \neq NP$ ، بنابراین این سه کلاس به این صورت قرار می‌گیرند: (NPC مخفف انپی کامل است)



کلاس انپی کامل

مسئله ارضاء پذیری مدار

- ما تاکنون مسائل انپی کامل را تعریف کرده‌ایم، اما انپی کامل بودن هیچ مسئله‌ای را اثبات نکرده‌ایم.
- زمانی که انپی کامل بودن یک مسئله را اثبات کنیم، با استفاده از کاهش‌پذیری در زمان چندجمله‌ای ابزاری قدرتمند برای اثبات انپی کامل بودن سایر مسائل در اختیار خواهیم داشت.
- از این‌رو، اکنون تمرکز خود را بر نشان دادن وجود یک مسئله انپی کامل قرار می‌دهیم: مسئله ارضاء‌پذیری مدار^۱.
- اثبات رسمی انپی کامل بودن مسئله رضایت‌پذیری مدار شامل جزئیاتی می‌شود که فراتر از بحث ما است. در عوض، یک اثبات غیر رسمی مبتنی بر مدارات منطقی ترکیبی بیان خواهیم کرد.

^۱ circuit satisfiability

کلاس انپی کامل

مسئله ارضاء پذیری مدار

- می خواهیم ثابت کنیم مسئله ارضاء پذیری مدار متعلق به کلاس انپی است.
- باید یک الگوریتم A با زمان اجرای چند جمله‌ای ارائه دهیم که یک گواه (در اینجا یک مقدار دهی به ورودی‌های مدار ترکیبی) و یک مدار دریافت کند و بررسی کند آیا خروجی مدار ۱ است یا خیر.
- کافیست به ترتیب مقدار خروجی گیت‌های منطقی حساب شود. از آنجا که اندازه ورودی را مجموع سیم‌ها و گیت‌ها تعریف کردیم و این الگوریتم چند جمله‌ایی است.

کلاس انپی کامل

مسئله ارضاء پذیری مدار

- فرض کنید L یک زبان دلخواه در انپی باشد، می‌خواهیم ثابت کنیم هر زبان انپی می‌تواند در زمان چند جمله‌ایی به زبان CIRCUIT-SAT کاهش یابد. فرض کنید تابع کاهش L به CIRCUIT-SAT را f بنامیم.
- بنابراین تابع f هر ورودی x را به یک مدار منطقی ترکیبی C نگاشت می‌کند به نحوی که اگر x عضو L است C هم عضو CIRCUIT-SAT باشد و اگر x عضو L نیست C هم عضو CIRCUIT-SAT نباشد.
- برای مثال فرض کنید می‌خواهیم زبان HAM-CYCLE به زبان CIRCUIT-SAT کاهش دهیم. اگر x را یک کد گذاری از گراف یک در نظر بگیریم تابع f آن را به یک مدار تبدیل می‌کند. این مدار ارضاء پذیر است اگر و تنها اگر گراف x دور همیلتونی داشته باشد.
- بنابراین برای اثبات انپی سخت بودن زبان CIRCUIT-SAT باید یک الگوریتم F با زمان چند جمله‌ایی ارائه دهیم که تابع کاهش f را محاسبه کند.

کلاس انپی کامل

مسئله ارضاء پذیری مدار

- از آنجا که L عضو انپی است، باید الگوریتم A وجود داشته باشد که زبان L را در زمان چند جمله‌ایی تصدیق کند. الگوریتم F که در ادامه خواهیم ساخت، از الگوریتم A برای ساخت مدار استفاده می‌کند.
- ایده اصلی این است که محاسبات را به شکل دنباله‌ایی از پیکربندی‌ها در نظر بگیریم. اگر فرض کنیم مدار ترکیبی M سخت افزار کامپیوتر را شبیه سازی می‌کند، می‌تواند هر پیکربندی c_i را به c_{i+1} نگاشت دهد.
- فرض کنید $T(n)$ زمان اجرای الگوریتم A در بدترین حالت باشد. وظیقه F این است که مدار M را به تعداد $(T(n) + 1)c_i$ کپی کند خروجی مدار ام i که پیکربندی c_i را تولید می‌کند مستقیماً به ورودی مدار $i+1$ ام وارد می‌شود.
- در این پیاده سازی -برخلاف سخت افزار کامپیوتر- پیکربندی‌ها به جای اینکه در حافظه کامپیوتر ذخیره شوند، صرفاً به عنوان مقادیر روی سیم‌های متصل کننده کپی‌های M قرار می‌گیرند.

کلاس انپی کامل

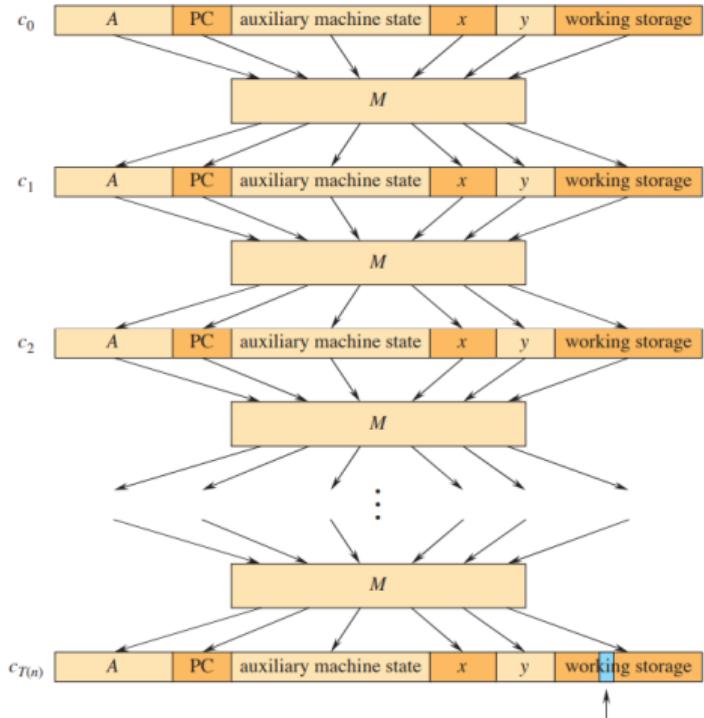
مسئله ارضاء پذیری مدار

- زمانی که تابع F یک ورودی x دریافت می‌کند، ابتدا مقدار $|x| = n$ را محاسبه کرده و سپس یک مدار ترکیبی C' می‌سازد که شامل $T(n)$ نسخه از ماشین M است. (مقدار n برای محاسبه $T(n)$ مورد نیاز است).
- ورودی مدار C' ، یک پیکربندی اولیه است که متناظر با محاسبه‌ای روی $A(x, y)$ می‌باشد، و خروجی آن، روی یک بیت مشخص در آخرین پیکربندی قرار می‌گیرد ($c_{T(n)}$) .

کلاس انپی کامل

مسئله ارضاء پذیری مدار

- در شکل زیر مداری که الگوریتم A را محاسبه می‌کند قابل مشاهده است:



کلاس انپی کامل

مسئله ارضاء پذیری مدار

- تا به اینجا الگوریتم A را با یک مدار پیاده سازی کردیم اما A دو ورودی x و y دارد که باید مشخص شوند. ورودی x توسط مسئله داده شده است. الگوریتم F آن را مستقیماً متصل می‌کند. چندین ورودی دیگر مربوط به وضعیت‌های اولیه حافظه مانند شمارنده برنامه نیز هستند که مقدار آنها مشخص است.
- بنابراین، تنها ورودی‌های باقی‌مانده مدار مربوط به گواه یعنی y خواهد بود.
- در مرحله دوم، الگوریتم تمام خروجی‌های مدار C' را نادیده می‌گیرد، به جز یک بیت از آخرین پیکربندی که مربوط به خروجی برنامه A است.
- بنابراین مدار C ، برای هر ورودی y ، مقدار $C(y)$ به این صورت محاسبه می‌کند:

$$C(y) = A(x, y)$$

کلاس انپی کامل

مسئله ارضاء پذیری مدار

- می خواهیم ثابت کنیم F یک الگوریتم کاهش چند جمله‌ای است بنابراین باید دو ویژگی را اثبات کنیم.
 - اول، باید نشان دهیم که الگوریتم F ، به درستی یک تابع کاهش f را محاسبه می‌کند. یعنی باید نشان دهیم که مدار C ارضاء‌پذیر است اگر و تنها اگر گواهی به نام y وجود داشته باشد به‌طوری‌که
$$A(x, y) = 1$$
.
 - دوم، باید نشان دهیم که الگوریتم F در زمان چند جمله‌ای اجرا می‌شود.

کلاس انپی کامل

مسئله ارضاء پذیری مدار

- برای نشان دادن اینکه F یک تابع کاهاش را به درستی محاسبه می‌کند، ابتدا فرض کنید که گواهی‌ای به نام y با طول وجود دارد به‌گونه‌ای که $A(x, y) = 1$. در این صورت، اگر بیت‌های y را به عنوان ورودی به مدار C اعمال کنیم، خروجی مدار برابر است با:

$$C(y) = A(x, y) = 1$$

پس اگر گواهی‌ای وجود داشته باشد، مدار C ارضاء‌پذیر است.

- در جهت عکس، فرض کنید که مدار C ارضاء‌پذیر است. بنابراین، ورودی‌ای به نام y برای C وجود دارد که: $C(y) = 1$. و از این نتیجه می‌گیریم که:

$$A(x, y) = 1$$

- در نتیجه، الگوریتم F تابع کاهاش را به درستی محاسبه می‌کند.

کلاس انپی کامل

مسئله ارضاء پذیری مدار

- حال باید ثابت کنیم F در زمان چند جمله‌ای نسبت به $|x| = n$ اجرا می‌شود.
- مدار ترکیبی M که سخت‌افزار رایانه را پیاده‌سازی می‌کند، اندازه‌ای چندجمله‌ای نسبت به اندازه یک پیکربندی دارد، ثابت می‌شود که اندازه پیکربندی $O(n^k)$ است. بنابراین، اندازه M نیز نسبت به n چندجمله‌ای خواهد بود.
- مدار C از $O(n^k)$ نسخه از M تشکیل شده است و در نتیجه، اندازه آن نیز چندجمله‌ای در n است. بنابراین الگوریتم کاهش F می‌تواند مدار C را در زمان چندجمله‌ای بسازد.

کلاس انپی کامل

مسئله ارضاء پذیری مدار

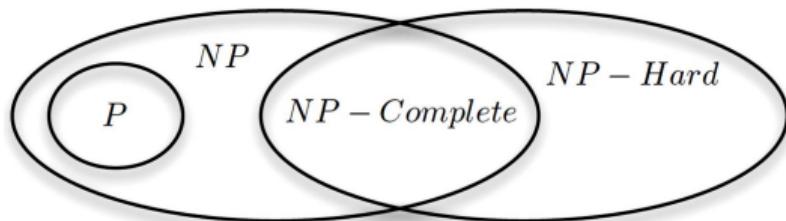
- بنابراین، زبان $CIRCUIT - SAT$ حداقل به سختی هر زبان دیگری در NP است، این یعنی این زبان در انپی سخت است
- به علاوه خود این زبان نیز در NP قرار دارد، پس انپی-کامل است.

اثبات‌های نظریه ان‌پی کامل

- در بخش قبل ثابت کردیم همه زبان‌های ان‌پی به زبان CIRCUIT-SAT کاهش‌پذیر اند.
- در این بخش نشان می‌دهیم که چگونه می‌توان ان‌پی-کامل بودن زبان‌ها را بدون کاهش مستقیم هر زبان در NP به زبان مورد نظر، اثبات کرد.
- لم رو به رو مبنایی برای اثبات ان‌پی-کامل بودن یک زبان فراهم می‌سازد:
اگر از یک زبان ان‌پی-کامل به زبانی مانند L یک کاهش وجود داشته باشد، L ان‌پی سخت است. حال اگر L ان‌پی هم باشد نتیجه می‌شود ان‌پی کامل است.

اثبات‌های نظریه انپی کامل

- به عبارت دیگر برای اثبات انپی کامل بودن یک زبان دو کار باید انجام شود؛ الف) ثابت شود زبان انپی است. ب) یک کاهش از یک زبان انپی کامل به آن پیدا شود.
- برای فهم بهتر این لم به شکل زیر دقت کنید:



اثبات‌های نظریه انپی کامل

- برای مثال حال می‌توانیم به سادگی اثبات کنیم مسئله ارضاءپذیری فرمول^۱ انپی کامل است.
- در مسئله ارضاءپذیری فرمول یک عبارت منطقی داده می‌شود و باید بررسی کنیم که آیا این عبارت ارضاءپذیر است یا خیر. به عبارت دیگر آیا یک مقدار دهی به متغیرهای آن وجود دارد که کل عبارت را برابر با ۱ کند یا خیر.
- این مسئله از نظر تاریخی اولین مسئله‌ایی است که ثابت شد انپی کامل است.

^۱ formula satisfiability

اثبات‌های نظریه ان‌پی‌کامل

- مسئله ارضاء‌پذیری فرمول به بیان زبان‌های صوری چنین تعریف می‌شود:

$$SAT = \{\langle \phi \rangle : \phi \text{ is a satisfiable boolean formula}\}$$

- برای اثبات ان‌پی‌کامل بودن SAT باید نشان دهیم:

الف اگر یک مقداردهی به متغیرهای یک عبارت منطقی داشته باشیم ارزیابی عبارت در زمان چند جمله‌ایی امکان‌پذیر است.

ب مسئله CIRCUIT-SAT در زمان چند جمله‌ایی قابل کاهش به مسئله SAT است.

برای دو حالت بالا الگوریتم چند جمله‌ایی وجود دارد. بنابراین به مسئله ارضاء‌پذیری فرمول، ان‌پی‌کامل است.

اثبات‌های نظریه انپی کامل

- هر چه برای زبان‌های بیشتری ثابت کنیم انپی کامل هستند برای اثبات‌های بعدی ابزارهای بیشتری در اختیار داریم. اما هنوز هم کار زیاد ساده نیست.
- تصور کنید که بخواهید همه نمونه‌های یک مسئله مانند ارضاءپذیری فرمول را به نمونه‌هایی از یک مسئله گراف تبدیل کنید. این کار ممکن است نیازمند بررسی تعداد زیادی از حالات باشد.
- بنابراین بهتر است از یک مسئله با ورودی محدودتر استفاده کنیم. مسئله ارضاءپذیری 3-CNF برای این هدف مناسب است.
- فرم CNF یک فرم استاندارد برای عبارات منطقی است. در درس مدار منطقی این فرم با نام POS نیز شناخته می‌شود.

اثبات‌های نظریه ان‌پی‌کامل

- در اینجا ۳-CNF را بررسی می‌کنیم که در آن عبارت AND تعدادی عبارت کوچک‌تر است و هر عبارت کوچک‌تر دقیقاً شامل ۳ متغیر یا نقیض متغیر است. نمونه‌ایی از عبارت در فرم ۳-CNF به این صورت است:

$$(\neg x_1 \vee \neg x_1 \vee x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

- تعریف می‌کنیم زبان ۳-CNF-SAT شامل کدگذاری عبارات منطقی ارضاء‌پذیر به فرم ۳-CNF است. می‌خواهیم نشان دهیم این زبان ان‌پی‌کامل است.

اثبات‌های نظریه انپی کامل

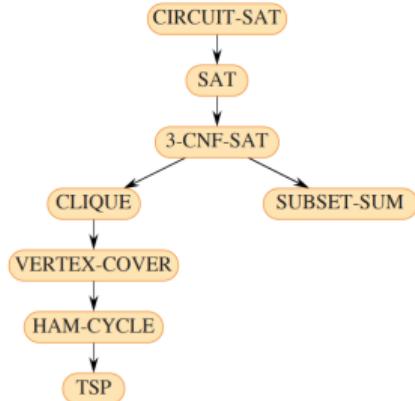
- زبان 3-CNF-SAT حالت ساده‌شده‌ایی از SAT است، پس هر الگوریتمی که SAT را حل کند 3-CNF-SAT را هم حل می‌کند. از آنجا که SAT انپی است پس 3-CNF-SAT هم انپی است.
- اثبات می‌شود که مسئله SAT در زمان چند جمله‌ایی به 3-CNF-SAT کاهش می‌یابد و در نتیجه انپی سخت است. در اینجا به بررسی این اثبات نمی‌پردازیم.
- بنابراین 3-CNF-SAT نیز عضو انپی کامل است.

مسائل انپی کامل

- مسائل انپی کامل در حوزه‌های متنوعی مطرح می‌شوند مانند: نظریه گراف، طراحی شبکه، مجموعه‌ها، ذخیره‌سازی و بازیابی، توالی و زمان‌بندی، جبر و نظریه اعداد، بازی‌ها و معماها، بهینه‌سازی، زیست‌شناسی، شیمی، فیزیک، و حوزه‌های دیگر.
- انپی کامل بودن مسئله ارضاء‌پذیری 3-CNF در بخش قبل اثبات شد.

مسائل انپی کامل

- این مسئله به ما در اثبات انپی کامل بودن بسیاری از مسائل کمک می‌کند. در شکل زیر می‌توانید ترتیب کاهش مسائل به یکدیگر را ببینید.



- انپی کامل بودن هر زبان موجود در این شکل، از طریق کاهش از زبانی که به آن اشاره دارد، اثبات می‌شود.

مسائل انپی کامل

مسئلهی کلیک

- یک کلیک^۱ در یک گراف بدون جهت (V, E) ، زیرمجموعهای مثل V' از رأس‌های V است که هر زوج از آن‌ها توسط یالی در E به یکدیگر متصل هستند. به عبارت دیگر، کلیک یک زیرگراف کامل از G است. اندازهی کلیک برابر با تعداد رأس‌های آن است.
- هدف این مسئلهی بھینه‌سازی یافتن کلیکی با بیشترین اندازه در گراف است. مسئلهی تصمیم‌گیری متناظر می‌پرسد: آیا کلیکی با اندازهی داده شدهی k در گراف وجود دارد یا نه.
- تعریف رسمی آن به صورت زیر است:

$$\text{CLIQUE} = \{\langle G, k \rangle : G \text{ k size of clique a containing graph a is } G\}$$

^۱ clique

مسائل انپی کامل

مسئله‌ی کلیک

- می‌خواهیم ثابت کنیم مسئله‌ی کلیک انپی کامل است.
- اثبات: ابتدا نشان می‌دهیم که کلیک انپی است. فرض کنید یک گراف و یک زیرمجموعه V' به عنوان گواه داده شده.
 - برای بررسی اینکه آیا V' یک کلیک است، در زمان چندجمله‌ای می‌توان بررسی کرد که آیا برای هر زوج $u, v \in V'$ ، یال (u, v) در E وجود دارد یا نه.

مسائل انپی کامل

مسئلهی کلیک

- حال باید نشان دهیم

$3\text{-CNF-SAT} \leq_P CLIQUE$

- ممکن است تعجب کنید که چگونه یک نمونه از 3-CNF-SAT به یک نمونه از $CLIQUE$ کاهش می‌یابد، چراکه در ظاهر، فرمول‌های منطقی ارتباطی با گراف‌ها ندارند.
- الگوریتم کاهش با یک نمونه از 3-CNF-SAT آغاز می‌شود. فرض کنید:

$$\phi = C_1 \wedge C_2 \wedge \dots \wedge C_k$$

یک فرمول بولی در فرم 3-CNF با k جزء^۱ باشد. هر جزء C_r دقیقاً سه لیترال متمایز l_3^r, l_2^r, l_1^r دارد.

^۱ clause

مسائل انپی کامل

مسئله‌ی کلیک

- می‌خواهیم گرافی G بسازیم به‌طوری‌که ϕ ارضاء‌پذیر باشد اگر و تنها اگر G دارای کلیکی با اندازه‌ی k باشد.
- ساخت گراف (V, E) : برای هر جزء $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ در ϕ ، سه رأس v_3^r, v_2^r, v_1^r به V اضافه می‌کنیم.
- سپس یال بین (v_i^r, v_j^s) را به E اضافه می‌کنیم اگر و تنها اگر: v_i^r و v_j^s متعلق به جزء‌های مختلف باشند (یعنی $l_i^r \neq l_j^s$ نباید)، و لیترال‌های متناظر آن‌ها سازگار باشند (یعنی l_i^r نقيض l_j^s نباید).
- این گراف را می‌توان در زمان چندجمله‌ای از ϕ ساخت.

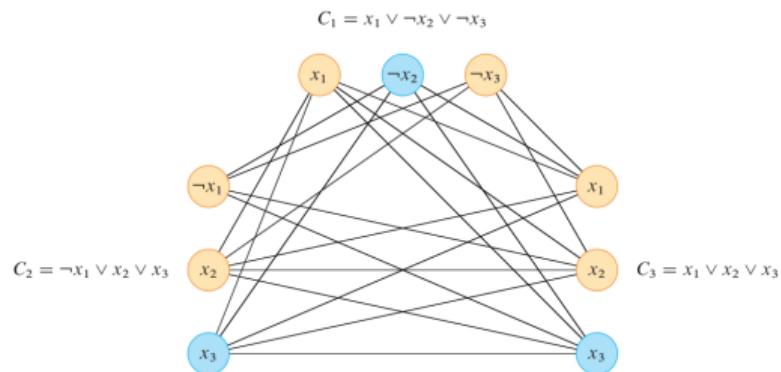
مسائل انپی کامل

مسئله کلیک

• مثال: اگر

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

باشد، گراف G حاصل مطابق شکل زیر ساخته می‌شود.



مسائل انپی کامل

مسئله‌ی کلیک

- اکنون باید نشان دهیم که این تبدیل یک کاهش معتبر است.
- جهت اول: فرض کنید ϕ یک مقداردهی ارضاء‌پذیر دارد. در این صورت، هر بند C_r حداقل یک لیترال l_i^r دارد که مقدار ۱ گرفته است. هر چنین لیترالی متناظر با یک رأس v_i^r است.
- با انتخاب یکی از این لیترال‌های با مقدار ۱ از هر جزء، مجموعه‌ای $V' \subseteq V$ با k رأس به دست می‌آید. ادعا می‌کنیم V' یک کلیک است.
- برای هر دو رأس $v_i^r, v_j^s \in V'$ با $r \neq s$ ، لیترال‌های l_i^r و l_j^s هر دو مقدار ۱ گرفته‌اند، پس نمی‌توانند نقیض یکدیگر باشند. در نتیجه، طبق ساخت گراف، یال $(v_i^r, v_j^s) \in E$ است. پس V' یک کلیک است.

مسائل انپی کامل

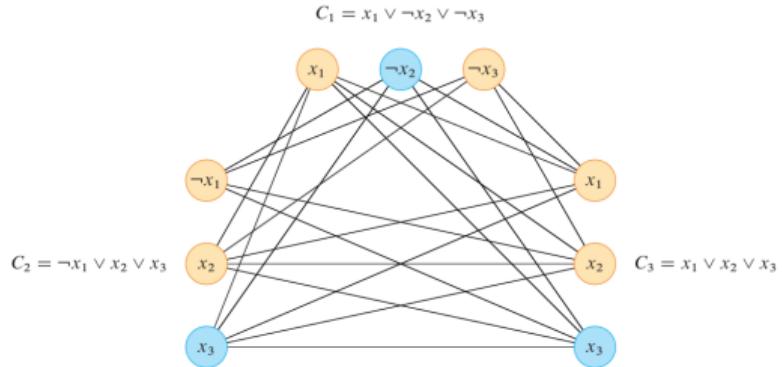
مسئله‌ی کلیک

- جهت دوم: فرض کنید گراف G کلیکی به اندازه k داشته باشد؛ یعنی زیرمجموعه‌ای مانند V' از رأس‌ها با اندازه k که یک کلیک است.
- از آنجا که هیچ یالی بین رأس‌های یک جزء وجود ندارد، مجموعه V' دقیقاً یک رأس از هر بند شامل می‌شود. اگر $V' \in v_i^r$ ، آنگاه مقدار ۱ را به لیترال متناظر l_i^r نسبت می‌دهیم.
- از آنجا که گراف هیچ یالی بین لیترال‌های ناسازگار ندارد، هیچ لیترال و نقیض آن به‌طور همزمان مقدار ۱ نمی‌گیرند. در نتیجه، هر جزء ارضاءپذیر است، و بنابراین ϕ نیز ارضاءپذیر خواهد بود.
- برای متغیرهایی که به هیچ رأسی در کلیک مربوط نمی‌شوند، می‌توان مقداردهی دلخواه انجام داد.

مسائل انپی کامل

مسئله‌ی کلیک

- در مثال شکل زیر، یک تخصیص ارضاء‌پذیر برای ϕ مقدار $x_2 = 0$ و $x_3 = 1$ دارد. یک کلیک متناظر با اندازه $k = 3$ شامل رأس‌هایی است که به ترتیب متناظر با $\neg x_2$ از جزء اول، x_3 از جزء دوم، و x_3 از جزء سوم هستند.



از آنجا که این کلیک هیچ رأسی متناظر با x_1 یا $\neg x_1$ ندارد، این تخصیص می‌تواند مقدار x_1 را به طور دلخواه \circ یا 1 قرار دهد.

مسائل انپی کامل

مسئلهی کلیک

- ممکن است این طور تصور شود که ما تنها نشان داده‌ایم که CLIQUE در گراف‌هایی که رأس‌ها فقط در سه‌تایی‌ها ظاهر می‌شوند و هیچ یالی بین رأس‌های یک سه‌تایی وجود ندارد، انپی سخت است.
- در واقع، ما تنها انپی سخت بودن CLIQUE را در این حالت محدود نشان داده‌ایم، اما همین اثبات برای نشان دادن انپی سخت بودن CLIQUE در گراف‌های کلی کافی است. چرا؟
- اگر الگوریتمی با زمان چندجمله‌ای وجود داشته باشد که مسئلهی CLIQUE را در گراف‌های کلی حل کند، آن الگوریتم می‌تواند همان مسئله را در گراف‌های محدود نیز حل کند.

مسائل انپی کامل

مسئلهی کلیک

- اما جهت مخالف، یعنی کاهش دادن نمونه‌های ۳-CNF-SAT با ساختار خاص به نمونه‌های کلی از CLIQUE کافی نیست. چرا؟ شاید نمونه‌هایی از ۳-CNF-SAT که ما برای کاهش انتخاب کردہ‌ایم، «ساده» باشند، و در نتیجه، ما عملاً یک مسئله‌ی انپی سخت را به CLIQUE کاهش نداده باشیم.
- افزون بر این، کاهش استفاده شده تنها از خود نمونه‌ی ۳-CNF-SAT استفاده می‌کند، نه از پاسخ آن. اگر کاهش چندجمله‌ای به دانستن این‌که آیا فرمول ϕ ارضاء‌پذیر است متکی بود، مرتكب خطأ شده بودیم، زیرا ما نمی‌دانیم که چگونه می‌توان در زمان چندجمله‌ای تعیین کرد که آیا ϕ ارضاء‌پذیر است یا نه.

شار بیشینه

- در دنیای واقعی شبکه‌های بسیاری وجود دارند که با هدف انتقال چیزی ساخته شده‌اند. برای مثال شبکه‌های آب رسانی، مدارات الکتریکی (انتقال جریان الکتریکی)، خطوط راه‌آهن و غیره. به طور معمول در چنین شبکه‌هایی افزایش نرخ انتقال مطلوب است. بنابراین مسئله شار بیشینه می‌پرسد: چگونه می‌توان نرخ انتقال را در این شبکه‌ها بیشینه کرد؟
- برای حل یک مسئله شار بیشینه باید شبکه را توسط گراف مدلسازی کرد. به این گراف «شبکه شار»^۱ و به هر چیزی که در این شبکه در جریان باشد به طور کلی شار^۲ گفته می‌شود.

¹ flow network

² flow

معرفی مسئله

- برای درک بهتر مسئله شار بیشیه ویژگی‌های این مسئله را همراه با مثال شبکه آب رسانی توضیح می‌دهیم؛
- در مسئله شار بیشیه گراف ثابت فرض می‌شود. (مثال: خطوط آبرسانی مثل لوله‌ها و اتصالات از قبل ساخته شده و غیر قابل تغییر است.)
- هر یال یک ظرفیت مشخص برای انتقال شار دارد و نمی‌تواند بیشتر از آن مقدار انتقال دهد. (مثال: هر لوله آب-بسته به قطر لوله- می‌تواند مقدار آبی مشخصی را از خود عبور دهد.)
- سرعت انتقال شار در سراسر شبکه ثابت فرض می‌شود. (مثال: نمی‌توانیم تعیین کنیم آب با فشار بیشتری وارد لوله‌ها شود بنابراین سرعت حرکت آب در لوله‌ها یکسان است.)

- در مسئله شار بیشینه تنها متغیری که ما می‌توانیم تعیین کنیم این است که چقدر از ظرفیت هر یال برای انتقال شار استفاده کنیم. (مثال: می‌توانیم یک شیر آب سر راه هر لوله قرار دهیم و بعد از حل مسئله شار بیشینه تعیین کنیم هر شیر اجازه ورود چه حجمی از آب را بدهد.)
- مجموع شار ورودی به هر رأس باید با مجموع شار خروجی برابر باشد. به عبارت دیگر هیچ شاری نمی‌تواند در رأس‌های گراف ذخیره شود یا از بین برود. به این قانون، قانون بقای شار^۱ گفته می‌شود. (مثال: بدیهیست در اتصالاتی که لوله‌ها را به هم وصل می‌کند آب نمی‌تواند ذخیره شود یا نشت کند. ممکن هر اتصال یک یا چند لوله ورودی و خروجی داشته باشد در هر صورت مجموع آب ورودی و خروجی باید برابر باشد.)

¹ flow conservation

معرفی مسئله

- با این توضیحات شاید به نظر برسد که برای حل این مسئله کافیست به طور حریصانه از همه ظرفیت همه یال‌ها برای انتقال استفاده کنیم. (مثال: همه شیر آب‌هایی که سر راه لوله‌ها قرار گرفته اند را تا بیشترین مقدار بازکنیم تا در صورتی که مقدار کافی آب به آن لوله رسید از تمام ظرفیت آن لوله برای انتقال آب استفاده کنیم.)
- نکته جالب اینجاست که برخلاف انتظار این الگوریتم حریصانه جواب بهینه را تولید نمی‌کند. در ادامه مثال‌های خواهیم دید که ممکن است کاهش مقدار شار گذرنده از یک یال باعث افزایش شار کلی گذرنده از شبکه شود. (مثال: در شبکه آب رسانی ممکن است با بستن شیر، اجازه ورود حجم کمتری از آب به یک لوله را بدهیم و با این کار شار گذرنده از شبکه را بیشتر کنیم.)

- قبل ادامه بحث بهتر است با یک تعریف دقیق و رسمی از شبکه شار آشنا شویم;
- شبکه شار یک گراف جهت دار است. به این معنی که شار نمی تواند در یک یال در دو جهت حرکت کند. شبکه شار همرا با تابع c به ورودی مسئله داده می شود. تابع c یالها و ظرفیت ها را نگاشت می کند به طوری که ظرفیت یال u به v برابر است با $c(u, v)$.
- شبکه شار دارای دو رأس خاص است که ورود شار به شبکه (و یا تولید شار) و خروج از شار شبکه (و یا مصرف شار) را مدل می کنند. به این دو رأس منبع^۱ و مقصد^۲ گفته می شود. به طور معمول به رأس منبع با s و رأس مقصد با t نشان داده می شوند.
- رأس منبع و مقصد تنها رئوسی هستند که از قانون بقای شار پیروی نمی کنند.

¹ source

² sink

شبکه شار

- همانطور که قبلا اشاره شد، گراف جهت دار می تواند طوقه داشته باشد. اما وجود طوقه در شبکه شار غیر مجاز است.
- همچنین دیدیم که وجود یال های پادموازی نیز در گراف جهت دار بلا منع است. اما در شبکه شار یال های پادموازی هم غیر مجاز است. (در بخش یادآوری فصل یال های پادموازی بحث شده است.)
- حذف طوقه (دور به طول ۱) و یال های پادموازی (دور به طول ۲) از پیچیدگی مسئله می کاهد.
- وجود دورهایی با طول بالاتر در شبکه شار مجاز است.

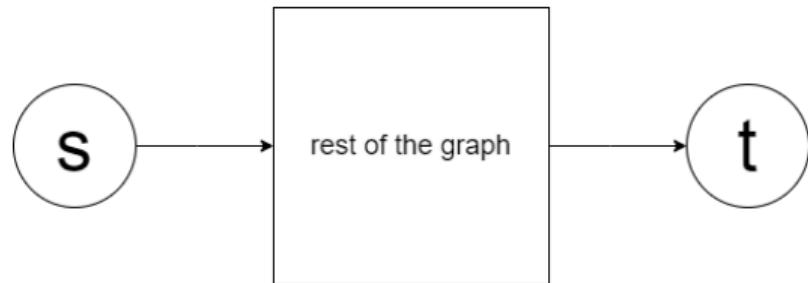
- همچنین باید یک تعریف کمی برای شار کل شبکه ارائه کنیم تا مشخص شود منظور از شار بیشینه چیست؛
- خروجی این مسئله کردن تابع f است که یال‌ها را به شار گذرنده از آنها نگاشت می‌کند. به این صورت که شار گذرنده از یال u به v برابر است با $f(u, v)$.
- تابع f باید دو ویژگی داشته باشد؛ قانون بقای شار را نقض نکند و از مقدار ظرفیت هر یال تجاوز نکند. در صورتی که تابعی مانند f' این ویژگی‌ها را نقض کند یک تابع شار نیست.

- مقدار شار کل شبکه با $|f|$ نشان داده می‌شود و به این صورت تعریف می‌شود:

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

- به زبان ساده این عبارت مجموع شار خالص خروجی از رأس منبع را مشخص می‌کند: مجموع شار خارج شونده از منبع منهای مجموع شار وارد شونده به منبع. (معمولًاً مجموع شار وارد شونده به منبع صفر است).
- قرارداد می‌کنیم که اگر بین u و v یال وجود نداشته باشد $f(u, v)$ برابر صفر است.

- امّا چطور این کمیّت می‌تواند نماینده شارگذرنده از کل شبکه باشد؟
- برای درک بهتر این موضوع به شبکه شار به چشم یک بلوک بزرگ نگاه کنید که از رأس منبع شار به آن وارد و از مقصد خارج می‌شود.



- با توجه به قانون بقای شار، شار نمی‌تواند در این بلوک بزرگ بماند بنابراین با همان نرخی که وارد آن می‌شود باید از آن خارج شود. (فرض کنید پیکان‌ها شار خالص را نشان می‌دهند).
- بنابراین نرخ خالص شار خروجی از منبع نماینده شاری است که در کل شبکه جریان دارد.

مدل‌سازی مسئله شار بیشینه

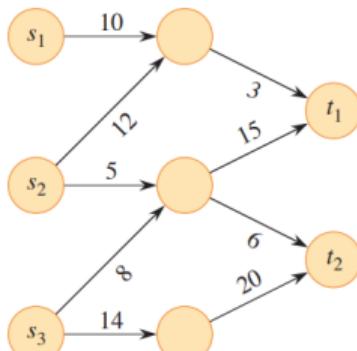
- شاید دقت کرده باشد که در تعریف شبکه شار فرض‌های ساده‌کننده‌ایی در نظر گرفتیم که لزوماً در یک مسئله واقعی بیشینه‌سازی شار، برقرار نیستند. دو فرض به این شکل داشتیم که در زیر آورده شده‌اند؛
 - ۱ ممکن است در یک مسئله واقعی بیشینه‌سازی شار چند منبع و چند مقصد داشته باشیم. در حالی که در تعریف شبکه شار تنها یک منبع و مقصد برای شبکه لحاظ کردیم.
 - ۲ ممکن است در یک مسئله واقعی بیشینه‌سازی شار یال موازی داشته باشیم. در حالی که وجود چنین یالی را در شبکه شار غیر مجاز دانستیم.

مدل‌سازی مسئله شار بیشینه

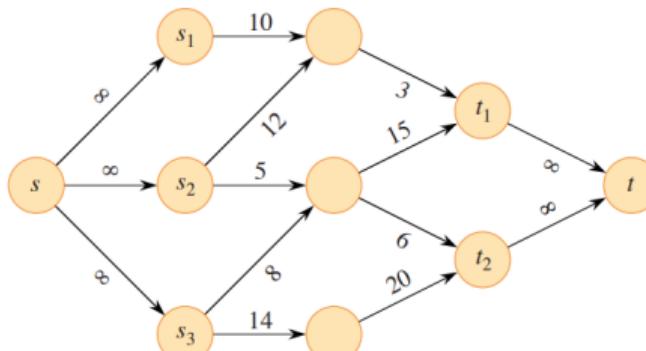
- در ادامه نشان می‌دهیم که چطور یک مسئله واقعی بیشینه‌سازی شار را که دارای چند منبع و مقصد است و یال پادمتقارن دارد را مدل کنیم.

مدل‌سازی مسئله شار بیشینه

- شکل زیر نشان می‌دهد چطور یک شبکه شار با چندین رأس منبع و مقصد را می‌توان به یک شبکه شار با یک منبع و مقصد تبدیل کرد.



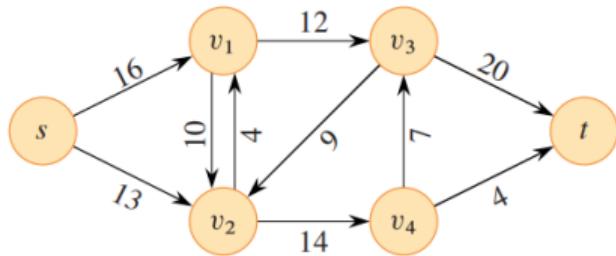
(a)



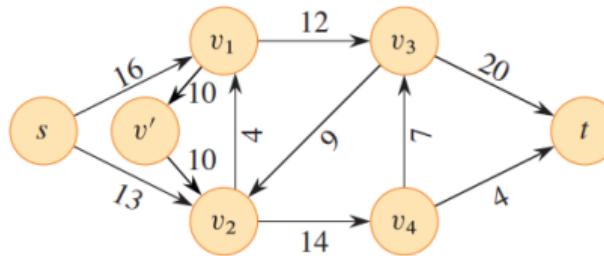
(b)

مدل‌سازی مسئله شار بیشینه

- شکل زیر نشان می‌دهد چطور یک شبکه دارای یال‌های پادمتقارن را می‌توان به یک شبکه شار قابل قبول تبدیل کرد.



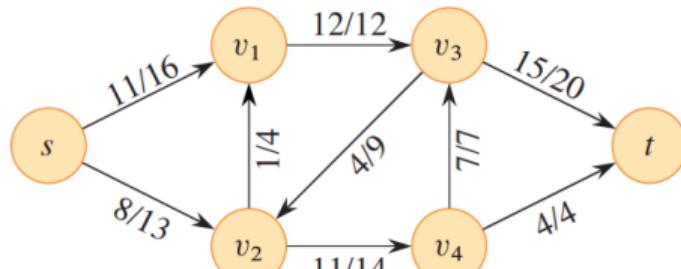
(a)



(b)

مثالی از مسئله شار بیشینه

- شکل زیر یک شبکه شار را به همراه شار موجود در آن نشان می‌دهد:



(a)

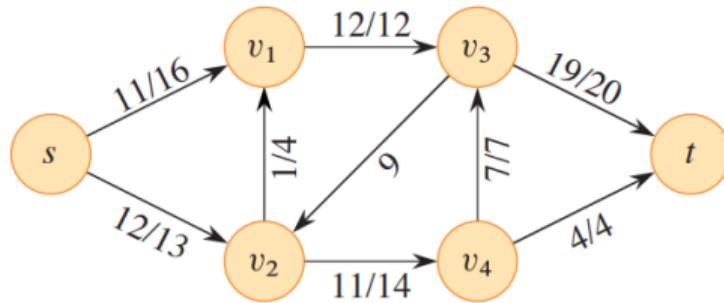
- برای نشان دادن شبکه شار یک قرارداد شناخته شده وجود دارد که در شکل بالا قابل مشاهده است. به این صورت که ابتدا مقدار شار و سپس ظرفیت هر یال نوشته می‌شود و این دو عدد به وسیله خط مورب از هم جدا می‌شوند.

مثالی از مسئله شار بیشینه

- همچنین دقت کنید که چگونه قانون پایستگی شار در این شکل رعایت شده. برای مثال ۱۱ واحد به راس ^v۷ وارد شده و ۴ و ۷ واحد از آن خارج می‌شوند.
- برای درک مسئله شار بیشینه بهتر است سعی کنیم مسئله را صورت دستی حل کنیم.
- برای مثال چطور می‌توان مقدار شار شبکه بالا را افزایش داد؟ به عبارت دیگر باید مقدار شار خروجی از رأس ^v۸ را بیشتر کنید بدون اینکه قانون پایستگی شار نقض شود. (مقدار شار فعلی ۱۹ واحد است).
- اگر موفق به افزایش شار شبکه شدید چطور می‌توان دریافت که این شار بیشینه است یا خیر؟

مثالی از مسئله شار بیشینه

- شکل زیر با تغییر مقدار شار ۳ یال، مقدار شار شبکه را به ۲۳ واحد افزایش داده است. این سه یال عبارت اند از: $(s, v_2), (v_3, v_2), (v_3, t)$



(c)

- نکته جالب توجه اینجاست که با صفر کردن شار یال (v_3, v_2) موفق به افزایش شار کل شبکه شدیم.

آشنایی با روش فورد-فولکرسون

- روش فورد-فولکرسون^۱ برای حل مسئله شار بیشینه طراحی شده است. بهتر است که به فورد-فولکرسون «الگوریتم» گفته نشود زیرا پیاده‌سازی دقیقی را مشخص نمی‌کند. بلکه یک چارچوب کلی برای حل مسئله شار بیشینه ارائه می‌دهد و چند پیاده‌سازی مختلف با زمان‌های اجرای متفاوت دارد.
- در این بخش ابتدا نحوه کار روش فورد-فولکرسون را بررسی می‌کنیم و در بخش آینده درستی این روش را اثبات می‌کنیم.

^۱ Ford-Fulkerson method

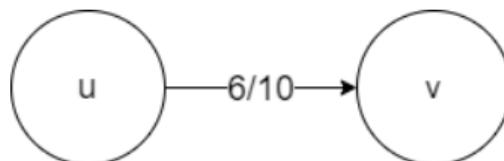
آشنایی با روش فورد-فولکرسون

- مبنای روش فورد-فولکرسون استفاده از یک گراف کمکی به نام گراف متمم^۱ است. گراف متمم بسیار شبیه به شبکه شار اصلی است. رأس‌های این گراف همان رأس‌های شبکه شار است اما یال‌های آن متفاوت است. «ظرفیت یال‌های این گراف نشان می‌دهند که شار در یال‌های شبکه اصلی چطور می‌تواند تغییر کند.»
- گراف متمم برخلاف شبکه شار می‌تواند یال پادمتقارن داشته باشد. اما تفاوت‌های این دو فقط در همین مسئله است. گراف متمم نیز یک گراف جهت‌دار و دارای دو رأس s و t است. همچنین برای این گراف شار تعریف می‌شود و هر یال ظرفیت مشخصی دارد.

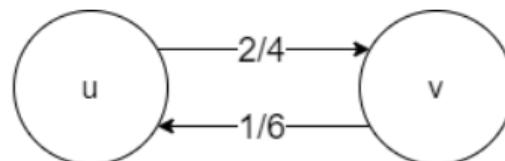
^۱ residual networks

آشنایی با روش فورد-فولکرسون

- همانطور که گفته شد، ظرفیت‌های گراف متمم نشان می‌دهند شار در گراف اصلی چطور می‌تواند تغییر کند. برای مثال به شکل زیر دقت کنید. شکل a یک یال در شبکه شار و شکل b معادل همان یال در گراف متمم است.



a) flow network



b) residual network

- همانطور که مشاهده می‌کنید گراف متمم می‌تواند شامل یالی باشد که در گراف اصلی وجود ندارد.
- ظرفیت یال موافق در گراف متمم نشان می‌دهد که ۴ واحد شار می‌توان به یال متناظر در گراف اصلی اضافه کرد. همچنین ظرفیت یال مخالف در گراف متمم نشان می‌دهد که ۶ واحد شار می‌توان از یال متناظر در گراف اصلی کم کرد.

آشنایی با روش فورد-فولکرسون

- امّا شار در گراف متمم به چه معنیست؟ در روش فورد-فولکرسون شاری که برای گراف متمم قرار می‌دهیم در واقع همان تغییراتی است که می‌خواهیم بر روی شبکه اصلی ایجاد کنیم.
- برای مثال در شکل بالا با اعمال تغییراتی که گراف متمم پیشنهاد می‌دهد یک واحد به شار یال (v, u) در شبکه اصلی اضافه می‌شود. زیرا در گراف متمم ۲ واحد شار در جهت موافق و ۱ واحد در جهت مخالف داریم.
- در اینجا صرفاً برای درک بهتر فقط قسمتی از گراف آورده شده. امّا تنها عملیات تعریف شده برای اعمال تغییرات گراف متمم روی شبکه شار، اعمال کل شار گراف متمم بر کل شار اصلی است.
- دقت کنید یال پادمتقارن در شبکه شار وجود ندارد بنابراین تبدیل به گراف متمم همیشه ممکن است. در بخش آینده نحوه ساخت گراف متمم را به طور دقیق خواهیم دید. در اینجا هدف ایجاد شهود از نحوه کار گراف متمم است.

آشنایی با روش فورد-فولکرسون

- بر روی گراف متمم به هر مسیر ساده از منبع تا مقصد «مسیر افزایشی»^۱ گفته می‌شود.
 - مسیرهای افزایشی ویژگی‌های جالب توجهی دارند:
- اگر در گراف متمم یک مسیر افزایشی داشته باشیم و مقداری شار برای این مسیر در نظر بگیریم و شار بقیه گراف را صفر کنیم، در این صورت اعمال تغییرات گراف متمم روی شبکه اصلی باعث افزایش شار می‌شود.
 - ثابت می‌شود که اگر گراف متمم مسیر افزایشی نداشته باشد، شار گراف اصلی بیشینه است و به جواب رسیده‌ایم.

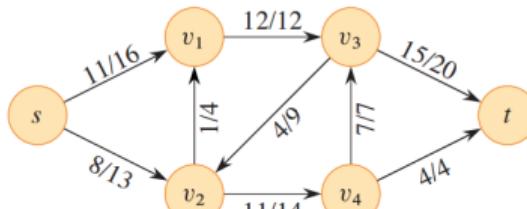
^۱ augmenting path

آشنايی با روش فورد-فولکرسون

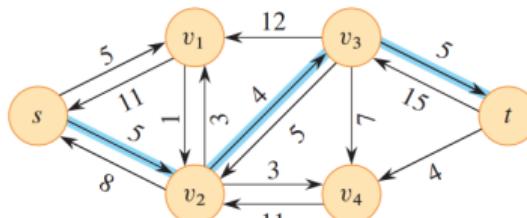
- حال که با گراف متمم آشنا شدیم فهم روش فورد-فولکرسون آسان است. این روش ابتدا کل شار شبکه اصلی را برابر صفر قرار می‌دهد سپس به صورت تکراری اقدام به بهبود مقدار شار می‌کند. مراحل روش فورد-فولکرسون به این ترتیب است:
 - شار همه یال‌های شبکه شار را برابر صفر قرار می‌دهیم.
 - گراف متمم متناظر با شبکه شار را تشکیل می‌دهیم. درصورتی که مسیر افزایشی وجود نداشته باشد، شار بیشینه یافته شده بنابراین کار تمام است.
 - درغیر این صورت یک مسیر افزایشی می‌یابیم و بیشترین شار ممکن را برای یال‌های عضو این مسیر در نظر می‌گیریم. و شار بقیه یال‌های گراف متمم را صفر می‌کنیم.
 - شار گراف متمم را بر شبکه اصلی می‌افزاییم. و به مرحله ۲ بازمی‌گردیم.

آشنایی با روش فورد-فولکرسون

- پیش از این مثالی از مسئله شار بیشینه ارائه شد و سعی کردیم آن را به صورت دستی حل کنیم. در ادامه خواهیم دید که روش فورد-فولکرسون چگونه این مثال را حل می‌کند.



(a)



(b)

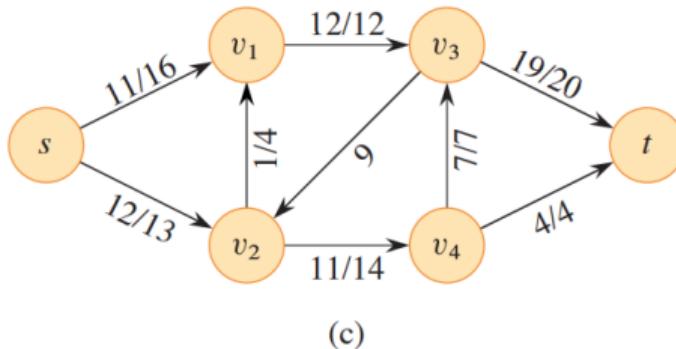
- قسمت a نشان دهنده شبکه شار و b نشان دهنده گراف متمم است و مسیر آبی رنگ یک مسیر افزایشی است.

آشنایی با روش فورد-فولکرسون

- در شکل b شار یال‌ها نشان داده نشده و تنها ظرفیت‌ها نشان داده شده‌اند. اما طبق روش فورد-فولکرسون می‌دانیم این مقدار را در کل گراف صفر و در تمام یال‌های مسیر افزایشی 4 است که بیشترین شار ممکن در این مسیر است. (چرا؟)
- بنابراین مقدار $|f|$ در شبکه شار برابر 19 واحد و در گراف متمم برابر 4 واحد است. (یادآوری می‌شود که برای محاسبه این مقدار باید مجموع شار خروجی از s را در نظر بگیرید.)

آشنایی با روش فورد-فولکرسون

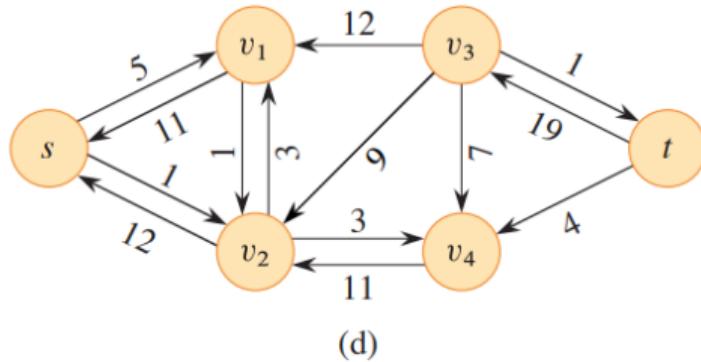
- بیایید شارگراف متمم را بر شبکه اعمال کنیم:



- در مسیر افزایشی شکل b، یال (v_2, v_3) وجود داشت که برخلاف جهت یال متناظر در شکل a است. بنابراین وجود شار در این یال به باعث کاهش شار در شبکه شار می‌شود. به قرار دادن جریان در یال مخالف در گراف متمم cancellation گفته می‌شود.

آشنایی با روش فورد-فولکرسون

- مقدار $|f|_1$ در شبکه شار حاصل ۲۳ واحد شد که این مقدار معادل است با مجموع مقدار $|f|_1$ در شبکه شار و گراف متمم ($4+19$). این مسئله اتفاقی نیست و خواهیم دید که این قائد به طور کلی برقرار است.
- این تنها یک تکرار از روش فورد-فولکرسون بود. حال که شبکه تغییر کرده، گراف متمم هم تغییر می‌کند. شکل زیر گراف متمم در مرحله بعد را نشان می‌دهد:



اثبات روش فورد-فولکرسون

- با نحوه کار روش فورد-فولکرسون آشنا شدیم. اما چرا این روش به درستی کار می‌کند؟
- در این بخش اثبات درستی این روش را خواهیم دید. برای اثبات‌ها نیاز است مفاهیمی که در بخش قبل با آنها آشنا شدیم را به طور رسمی به زبان ریاضی تعریف کنیم.

اثبات روش فورد-فولکرسون

- تعریف رسمی گراف متمم:
- در تعریف شبکه شار گفتیم f تابعیست که شار یال‌ها را بازمی‌گرداند. برای شبکه شار G با تابع ظرفیت c و یک شار مانند f بر روی این شبکه، گراف متمم را با G_f و شار آن را با f' نشان می‌دهیم.
- دقت کنید که f جزئی از تعریف G نیست و یک شبکه شار می‌تواند f های مختلفی داشته باشد. بنابراین این نحوه نوشتار (G_f) مشخص می‌کند گراف متمم متناظر با کدام شبکه و کدام شار از شبکه است.

اثبات روش فورد-فولکرسون

- تابع ظرفیت گراف متمم به این صورت تعریف می‌شود:

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E \\ f(v, u) & \text{if } (v, u) \in E \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

- تعریف می‌کنیم مقدار تابع c برای یال‌هایی که در گراف وجود ندارند صفر است (پیش از این اشاره کردیم که تابع f نیز همین ویژگی را دارد). این مسئله برای فهم فرمول بالا ضروری است. برای درک بهتر فرمول بالا را در حالتهای حدی بررسی کنید. (از ظرفیت یال کاملاً استفاده شود، شار یک یال دارای ظرفیت صفر باشد، بین دو رأس یالی وجود نداشته باشد)

اثبات روش فورد-فولکرسون

- تعریف افزایش شار:
- به اعمال تغییرات گراف متمم بر شبکه شار augmentation گفته می‌شود. در ترجمه فارسی از کلمه «افزایش» استفاده می‌کنیم هرچند منظور از این ترجمه، افزایش شار نیست بلکه افزودن شار گراف متمم بر شار شبکه شار است.
- افزودن شار f به f' را با $f' \uparrow f$ نشان می‌دهیم. و به این صورت تعریف می‌کنیم:

$$f' \uparrow f(u, v) = \begin{cases} f(u, v) + f'(u, v) - f'(v, u) & \text{if } (u, v) \in E \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

اثبات روش فورد-فولکرسون

- $f' \uparrow$ نیز یک تابع است که به هر جفت رأس یک مقدار نسبت می‌دهد. ادعا می‌کنیم که این تابع نیز تابع شار است و می‌توانیم آن را جایگزین f کنیم. برای اثبات این ادعا باید ثابت کنیم $f' \uparrow$ ویژگی‌های تابع شار را دارد. تابع شار تابعی است که به هر جفت رأس یک مقدار نسبت می‌دهد و:
 - ۱ از ظرفیت‌های شبکه شار تجاوز نمی‌کند.
 - ۲ از قانون پایستگی شار پیروی می‌کند.
- بنابراین باید بررسی آیا $f' \uparrow$ این دو ویژگی را دارد یا خیر.

اثبات روش فورد-فولکرسون

- ویژگی اول به سادگی با توجه به فرمول‌های ۱ و ۲ اثبات می‌شوند. به فرمول ۱ دقت کنید؛ شار ظرفیت موافق به دقیقاً برابر با تفاضل شار فعلی و حداکثر ظرفیت تعریف می‌شود بنابراین حتی اگر از تمام ظرفیت یال موافق در گراف متمم استفاده کنیم، شار $f' \uparrow f$ از حداکثر نفوذ نمی‌کند.
- به دلایل مشابه و با توجه به تعاریف می‌توان اثبات کرد که شار $f' \uparrow f$ از صفر کمتر نمی‌شود.

اثبات روش فورد-فولکرسون

- برای اثبات ویژگی دوم (پایستگی شار) ابتدا باید ثابت کنیم این رابطه برای هر u برقرار است:

$$\begin{aligned} \sum_{v \in V} f \uparrow f'(u, v) - \sum_{v \in V} f \uparrow f'(v, u) \\ = \sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) + \sum_{v \in V} f'(u, v) - \sum_{v \in V} f'(v, u) \quad (3) \end{aligned}$$

به زبان ساده این رابطه می‌گوید به ازای هر رأس مثل u

تفاضل شار وارد و خارج شونده به u در $f' \uparrow f'$ $=$

تفاضل شار وارد و خارج شونده به u در f $+$

تفاضل شار وارد و خارج شونده به u در f'

اثبات روش فورد-فولکرسون

- در اینجا تساوی ۳ را اثبات نمی‌کنیم. با اعمال رابطه ۲ به سمت چپ تساوی این اثبات قابل انجام است.
اما درستی این رابطه به صورت شهودی نیز قابل فهم است؛
شارگراف متمم به همه یال‌ها افزوده شده بنابراین انتظار داریم مقدار شار ورودی به هر رأس، به اندازه شار ورودی به همان رأس در f' افزایش داشته باشد. این مسئله برای مجموع شار خروجی نیز صادق است.
- به معادله ۳ دقت کنید. سمت راست این تساوی برابر صفر است زیرا پایستگی در هر دو شار f و f' برقرار است. بنابراین در هر دو، شار وارد شونده به یک رأس با شار خارج شونده از آن برابر است. (به غیر از s و t که پایستگی شار در آنها وجود ندارد.)

اثبات روش فورد-فولکرسون

- حال اگر سمت راست معادله ۳ را صفر بگذاریم نتیجه می‌گیریم:

$$\sum_{v \in V} f \uparrow f'(u, v) = \sum_{v \in V} f \uparrow f'(v, u)$$

- بنابراین پایستگی شار در $f' \uparrow f$ برقرار است. توجه کنید که f و f' هر دو قانون پایستگی شار را رعایت می‌کنند بنابراین قابل انتظار است که شار حاصل افزودن این دو شار نیز چنین باشد.

اثبات روش فورد-فولکرسون

- تا اینجا ثابت کردیم $f' \uparrow f$ یک تابع شار است پس قابلیت اعمال بر شبکه شار را دارد، اما این کافی نیست. می خواهیم انجام این کار باعث افزایش مقدار $|f|$ شود. به عبارت دیگر $|f' \uparrow f|$ بزرگ‌تر از $|f|$ باشد.
- خواهیم دید که اگر f' به این صورت تعریف شود، این اتفاق همیشه می‌افتد:

$$f_p(u, v) = \begin{cases} c_p & \text{if } (u, v) \in P \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

- در تعریف بالا p یک مسیر افزایشی در گراف متمم است. و c_p بیشترین شاری است که می‌تواند در این مسیر جریان یابد. این مقدار برابر است با:

$$c_f(p) = \min\{c_f(u, v) : (u, v) \in p\} \quad (5)$$

اثبات روش فورد-فولکرسون

- برای اینکه ثابت کنیم افزودن f_p به f مقدار $|f|$ را افزایش می‌دهد. باید ثابت کنیم این رابطه برقرار است:

$$|f \uparrow f'| = |f| + |f'| \quad (6)$$

- پیش از این با معادله ۳ آشنا شدیم. دیدیم این معادله به ازای هر $u \in E$ برقرار است. کافیست به جای u رأس s را در آن قرار دهید تا معادله ۶ نتیجه شود. (توجه کنید که در دو رأس s و t پایستگی شار وجود ندارد بنابراین سمت راست تساوی لزوماً برابر صفر نخواهد شد.)
- به زبان ساده این رابطه می‌گوید مقدار شار در شبکه شار افزایش یافته (augmented) برابر است با مجموع شار شبکه شار اولیه و گراف متمم.

اثبات روش فورد-فولکرسون

- بنابراین طبق معادله ۶ می‌دانیم:

$$|f \uparrow f_p| = |f| + |f_p|$$

- کافیست ثابت کنیم شارگراف متمم که از روی یک مسیر افزایشی به دست آمده الزاماً مثبت است و بنابراین

$$|f \uparrow f_p| > |f|$$

- اثبات اینکه $|f_p|$ مثبت است ساده است؛ می‌دانیم p یک مسیر ساده از s به t است بنابراین نمی‌تواند یال ورودی به s داشته باشد و دقیقاً یک یال خروجی از s دارد.

- شارگذرنده از یال خروجی s برابر با کمترین ظرفیت مسیر p است (معادله ۵). می‌دانیم یالی در مسیر p وجود ندارد که ظرفیت ۰ یا منفی داشته باشد. زیرا چنین یالی اصلاً در گراف متمم وجود ندارد (طبق تعریف، ظرفیت مقداری مثبت است و ظرفیت صفر به معنای عدم وجود یال در گراف متمم است).

اثبات روش فورد-فولکرسون

- تا اینجا دیدیم برای افزایش شار کافیست یک مسیر افزایشی در گراف متمم پیدا کنیم و با استفاده از آن عملیات افزایش را روی شبکه شار انجام دهیم و سپس دوباره همین عمل را انجام دهیم. اما شرط خاتمه چیست؟ چطور می‌توانیم از بهنیه بودن جواب اطمینان یابیم؟
- قضیه «شار بیشینه، برش کمینه»^۱ به این سوال اینگونه پاسخ می‌دهد:
در صورتی که هیچ مسیر افزایشی در گراف متمم وجود نداشته باشد شار موجود در شبکه شار بیشینه است.

^۱ Max-fow min-cut theorem

اثبات روش فورد-فولکرسون

- به طور دقیق‌تر قضیه «شار بیشینه، برش کمینه» بیان می‌کند این سه گزاره هم ارز هستند:
 - ۱ یک شار بیشینه در G_f باشد.
 - ۲ گراف متمم G_f مسیر افزایشی نداشته باشد.
 - ۳ مقدار شار برابر با ظرفیت یک برش از گراف باشد.
- برش و ظرفیت برش را در ادامه تعریف خواهیم کرد.

اثبات روش فورد-فولکرسون

- هم ارز بودن گزاره‌های اول و دوم برای اطمینان از صحت الگوریتم کفايت می‌کند اما برای اثبات این قضیه این مسیر را طی می‌کنیم:

$$\{(1 \Rightarrow 2) \wedge (2 \Rightarrow 3) \wedge (3 \Rightarrow 1)\} \Rightarrow \{1 \Leftrightarrow 2 \Leftrightarrow 3\}$$

- بنابراین گزاره سوم در اثبات کمک می‌کند. به اینکه چگونه از سه عبارت سمت چپ هم ارزی دو گزاره را ثابت کردیم دقت کنید. مثلاً برای اثبات همارزی ۳ و ۱ کافیست یک بار از ۳ شروع کنید و ۱ را نتیجه بگیرید و بار دیگر از ۳ شروع و ۱ را نتیجه بگیرید:

$$\{(1 \Rightarrow 2) \wedge (2 \Rightarrow 3)\} \wedge \{(3 \Rightarrow 1)\} \Rightarrow \{(3 \Leftrightarrow 1)\}$$

اثبات روش فورد-فولکرسون

- بنابراین برای اثبات این قضیه باید با برش^۱ و ویژگی‌های آن بیشتر آشنا شویم. ممکن است این مفهوم را در بحث درخت پوشای کمینه دیده باشید. یک برش گراف را به دو مجموعه رأس S و T تقسیم می‌کند. برش در شبکه شار نیز شبیه به گراف معمولی است با این تفاوت که رأس منبع باید در S و رأس مقصد در T باشد. بنابراین هر دو نمی‌توانند در یک بخش باشند.
- کمیتی به نام ظرفیت برش برابر است با کل ظرفیت یال‌هایی که شروع آنها از S است و پایان آنها در T باشد:

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$

^۱ cut

اثبات روش فورد-فولکرسون

- همچنین شار خالصی^۱ که از S به سمت T در جریان است را با $f(S, T)$ نشان داده و به این صورت تعریف می‌کنیم:

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u) \quad (7)$$

- شار خالص ترجمه عبارت flow net است. این عبارت برای کمیتی به کار می‌رود که حاصل کسر شارهای جهت مخالف از شارهای موافق باشد. برای مثال معادله ۷ شار خالص گذرنده از یک برش را نشان می‌دهد.

^۱ net flow

اثبات روش فورد-فولکرسون

- به برشی که کمترین ظرفیت را در میان تمام برش‌ها داشته باشد «برش کمینه»^۱ گفته می‌شود.
- برش در شبکه شار ویژگی مهمی دارد که در اثبات قضیه شار بیشینه، برش کمینه به ما کمک می‌کند. به این صورت که شار خالص گذرنده از هر برش دلخواه (که در معادله ۷ تعریف کردیم) با α_l برابر است.
- از این ویژگی می‌توان نتیجه گرفت مقدار شار نمی‌تواند از ظرفیت برش کمینه بیشتر باشد. اگر فرض کنیم α_l بیشتر از ظرفیت یک برش باشد به تناقض می‌رسیم. زیرا می‌دانیم شار خالص این برش α_l در حالی که ظرفیت این برش از α_l کمتر است.
- بنابراین ظرفیت هر برش یک حد بالا برای α_l است.

¹ minimum cut

اثبات روش فورد-فولکرسون

- حال که با برش در شبکه شار آشنا شدیم می‌توانیم قضیه «شار بیشینه برش کمینه» را ثابت کنیم. طبق مسیر کلی ایی که پیش از این ارائه شد، ابتدا باید ثابت کنیم در صورتی که مقدار شار بیشینه باشد در گراف متمم مسیر افزایشی وجود ندارد ($2 \Rightarrow 1$).
- فرض کنید f شار بیشینه است و یک مسیر افزایشی مانند p در گراف متمم وجود دارد. پیش از این دیدیم که افزودن شار مسیر افزایشی به شار فعلی باعث افزایش $|f|$ می‌شود. بنابراین $|f_p| > |f|$ از $|f|$ بیشتر است که این با فرض اولیه در تناقض است. بنابراین در صورتی که شار بیشینه باشد مسیر افزایشی نمی‌تواند در گراف متمم وجود داشته باشد.

اثبات روش فورد-فولکرسون

- در قدم بعد باید ثابت کنیم اگر گراف متمم شامل مسیر افزایشی نباشد، مقدار شار با ظرفیت یکی از برش‌ها برابر است ($3 \Rightarrow 2$).
- فرض کنیم گراف متمم مسیر افزایشی نداشته باشد. می‌خواهیم برشی بیابیم که شار خالص آن ظرفیت آن برابر باشد. همچنین می‌دانیم شار خالص هر برش نیز با α برابر است.
- مجموعه S را تمام رأس‌هایی تعریف می‌کنم که در گراف متمم از رأس منبع به آنها مسیر وجود دارد و بقیه رأس‌ها در مجموعه V قرار می‌گیرند. طبق تعریف (معادله ۷) شار خالص این برش برابر است با:

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u)$$

اثبات روش فورد-فولکرسون

- به عبارت زیر که قسمتی از معادله بالاست دقت کنید:

$$\sum_{u \in S} \sum_{v \in T} f(u, v)$$

یادآوری می‌شود که تابع شار برای یال‌هایی که وجود ندارند صفر است. بنابراین در عمل این عبارت مجموع یال‌هایی است که بین دو بخش گراف وجود دارند و جهت آنها از S به T است. ادعا می‌کنیم که شار گذرنده از همه این یال‌ها برابر با ظرفیت آنهاست.

- برای اثبات این ادعا فرض کنید شار یک یال از $S \in T$ به y از ظرفیت آن کمتر باشد بنابراین یک یال از x به y باید در گراف متمم وجود داشته باشد که نشان دهنده امکان افزایش شار است (طبق معادله ۱). می‌دانیم از S به x مسیر وجود دارد زیرا این رأس در مجموعه S است. اگر x به y مسیر داشته باشد پس از S به y یک مسیر وجود دارد اماً y در مجموعه S نیست و اینجا به تناقض می‌رسیم.

اثبات روش فورد-فولکرسون

- با استدلال مشابه می‌توانیم ثابت کنیم که قسمت دوم عبارت برابر صفر است. این عبارت نیز مجموع ظرفیت همان یال‌های بین دو بخش است با این تفاوت که جهت آنها از به S باشد:

$$\sum_{v \in T} \sum_{u \in S} f(v, u)$$

- ادعا می‌کنیم مجموع شار یال‌هایی که از T به S می‌روند صفر است
- برای اثبات این ادعا فرض کنید شار یک یال از $x \in S$ به $y \in T$ بیشتر از صفر باشد بنابراین یک یال از x به y باید در گراف متمم وجود داشته باشد که نشان دهنده امکان کاهش شار است (طبق معادله ۱). حال که نتیجه گرفتیم از x به y مسیر وجود دارد با استدلال مشابه قسمت قبل می‌توان به تناقض رسید.

اثبات روش فورد-فولکرسون

• بنابراین

$$\begin{aligned} f(S, T) &= \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u) \\ &= \sum_{u \in S} \sum_{v \in T} c(u, v) - \sum_{v \in T} \sum_{u \in S} 0 \\ &= c(S, T) \end{aligned}$$

- بنابراین ثابت کردیم که اگر $|f|$ بیشینه باشد یک برش وجود دارد که شار این برش برابر با ظرفیت آن است و می‌دانیم شار یک برش با مقدار $|f|$ برابر است: $|f| = f(S, T) = c(S, T)$

اثبات روش فورد-فولکرسون

- در قدم بعد باید ثابت کنیم اگر مقدار شار با ظرفیت یکی از برش‌ها برابر باشد، مقدار شار بیشینه است.
 $(3 \Rightarrow 1)$.
- این اثبات ساده است. می‌دانیم که مقدار f_1 کوچک‌تر یا مساوی با ظرفیت هر برش دلخواه است. بنابراین اگر f_1 بیابیم که با ظرفیت یک برش برابر باشد می‌دانیم این شار بیشینه است زیرا در غیر این صورت شاری مثل f' وجود دارد که مقدار آن از ظرفیت یک برش بیشتر است و به این صورت به تناقض می‌رسیم.

اثبات روش فورد-فولکرسون

- اثبات قضیه «شار بیشینه، برش کمینه» تکمیل شد. برای درک بهتر این اثبات خوب است مجدداً روند کلی این اثبات را نگاهی بیاندازید.
- به زبان ساده، قضیه «شار بیشینه، برش کمینه» می‌گوید: زمانی که مسیر افزایشی وجود نداشته باشد f_1 بیشینه است و در این حالت حداقل یک برش وجود دارد که ظرفیت آن با f_1 برابر است و این برش کمینه است.

تطابق بیشینه در گراف دو بخشی

مقدمه

- برخی مسائل ترکیبیاتی را می‌توان به صورت مسائل شار بیشینه مدل‌سازی کرد، مانند مسئله شار بیشینه با چندین مبدأ و مقصد که پیش از این مطرح شد.
- برخی دیگر از مسائل ترکیبیاتی در ظاهر ارتباطی با شبکه‌های شار ندارند، اما در واقع می‌توان آن‌ها را به مسائل بیشینه جریان کاهش داد. در ادامه یکی از این مسائل را معرفی می‌کنیم: یافتن یک تطابق بیشینه در یک گراف دو بخشی^۱.
- خواهیم دید که چگونه می‌توان از روش فورد-فالکرسون برای حل مسئله تطابق بیشینه گراف دو بخشی^۲ در زمان استفاده کرد. زمان اجرای آن $O(|V||E|)$ می‌باشد.

¹ bipartite graph

² maximum bipartite matching

تطابق بیشینه در گراف دو بخشی

تعاریف

- با داشتن یک گراف بدون جهت $G = (V, E)$ ، یک تطابق^۱ زیرمجموعه‌ای از یال‌ها به صورت $M \subseteq E$ است به طوری که برای تمام رأس‌های $v \in V$ ، حداکثر یک یال از M به v متصل باشد.
- می‌گوییم یک رأس $v \in V$ توسط تطابق M تطبیق‌یافته^۲ است اگر یالی در M وجود داشته باشد که به v متصل باشد؛ در غیر این صورت، v تطبیق‌نیافته^۳ است.
- یک تطابق بیشینه^۴، تطابقی با بیشترین تعداد یال است؛ M تطابق بیشینه است اگر برای هر تطابق دیگر M' ، داشته باشیم $|M| \geq |M'|$.

^۱ matching

^۲ matched

^۳ unmatched

^۴ maximum matching

تطابق بیشینه در گراف دو بخشی

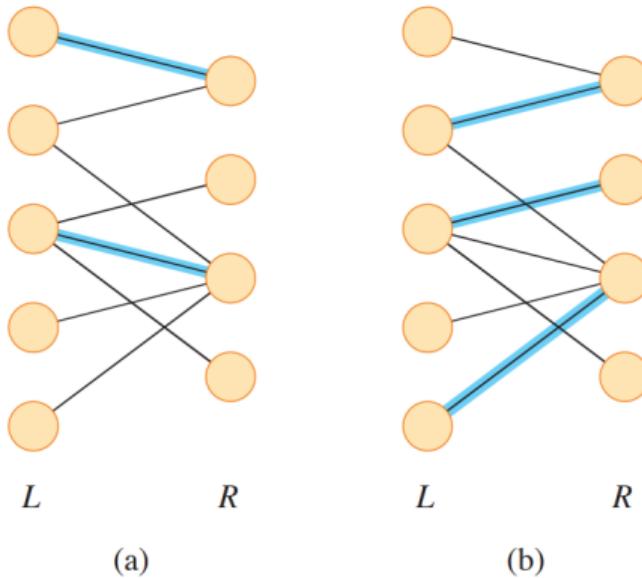
تعاریف

- در این بخش، تمرکز ما بر یافتن تطابق‌های بیشینه در گراف‌های دو بخشی^۱ است: گراف‌هایی که مجموعه رأس‌های آن‌ها را می‌توان به صورت $V = L \cup R$ افزایز کرد، به‌طوری‌که L و R مجزا باشند و تمام یال‌ها یک رأس در L داشته باشند و رأس دیگر آنها در R باشد.
- علاوه بر این، فرض می‌کنیم همه رأس‌های گراف حداقل به یک یال متصل هستند.

^۱ bipartite graphs

تطابق بیشینه در گراف دو بخشی

- شکل زیر دو تطابق مختلف در یک گراف دو بخشی را نشان می‌دهد. یال‌های هر تطابق با رنگ آبی نشان داده شده‌اند.



تطابق بیشینه در گراف دو بخشی

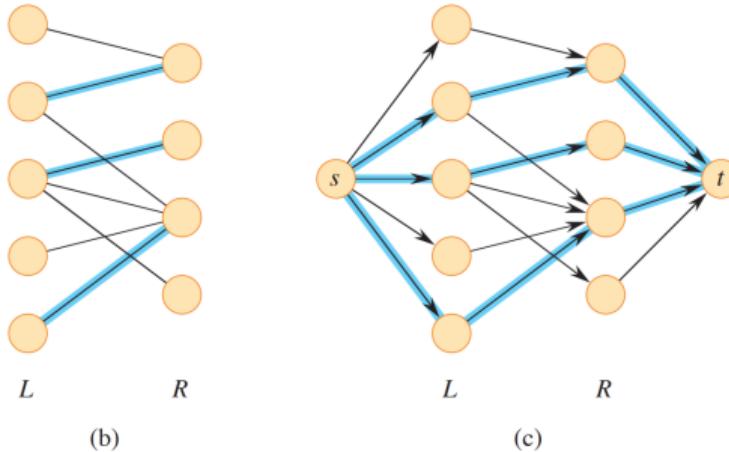
کاربردها

- مسئله یافتن تطابق بیشینه در یک گراف دو بخشی، کاربردهای عملی فراوانی دارد.
- به عنوان مثال، فرض کنید بخواهیم مجموعه‌ای از افراد به نام L را با مجموعه‌ای از وظایف به نام R که باید به صورت همزمان انجام شوند، تطبیق دهیم.
یک یال $(u, v) \in E$ نشان می‌دهد که فرد خاصی قادر به انجام وظیفهٔ خاصی است.
- بنابراین هر تطابق یک انتساب وظایف به افراد است و یک تطابق بیشینه، برای بیشترین تعداد ممکن از افراد کار فراهم می‌کند.

طابق بیشینه در گراف دو بخشی

طابق بیشینه با روش فورد-فولکرسون

- روش فورد-فالکرسون مبنایی برای یافتن طابق بیشینه در یک گراف دوبخشی بدون جهت فراهم می‌کند.
نکته کلیدی در اینجا ساخت یک شبکه شار است به‌گونه‌ای که شار متناظر با تطابق باشد. در شکل زیر ساخت شبکه شار از گراف دوبخشی نشان داده شده است.



تطابق بیشینه در گراف دو بخشی

تطابق بیشینه با روش فورد-فولکرسون

- «شبکه جریان متناظر»^۱ ($G' = (V', E')$) را برای گراف دوبخشی G به صورت زیر تعریف می‌کنیم: علاوه بر رئوس مجموعه V دو رأس جدید مبدأ s و رأس مقصد t نیز به V' اضافه می‌کنیم. اگر افزای رأس‌های گراف G به دو مجموعه L و R باشد، یال‌های جهت‌دار گراف G' به صورت زیر تعریف می‌شوند:

$$E' = \{(s, u) \mid u \in L\} \cup \{(u, v) \mid u \in L, v \in R, (u, v) \in E\} \cup \{(v, t) \mid v \in R\}$$

- در نهایت، به هر یال در E' ظرفیت واحد (مقدار ۱) اختصاص می‌دهیم.
- به سادگی ثابت می‌شود:

$$|E| \leq |E'| \leq 3|E|$$

بنابراین افزایش تعداد یال‌ها از نظر مجانبی مرتبه تعداد یال‌ها را تغییر نمیدهد) ((|E'| = $\Theta(|E|)$).

^۱ corresponding flow network

تطابق بیشینه در گراف دو بخشی

تطابق بیشینه با روش فورد-فولکرسون

- می خواهیم نشان دهیم که یک تطبیق در گراف دو بخشی G با یک شار در شبکه شار' G' متناظر است.
- به یک شار مانند f ، «شار صحیح»^۱ می گوییم اگر برای همهی مقدار شار آن روی همه یال‌ها عدد صحیح باشد.
- اما منظور متناظر بودن شار و تطابق چیست؟ در ادامه یک تعریف رسمی به همین منظور ارائه می‌دهیم.

^۱ integer-valued

تطابق بیشینه در گراف دو بخشی

تطابق بیشینه با روش فورد-فولکرسون

- ادعایی که باید ثابت شود به این ترتیب است:
- فرض کنید $G = (V, E)$ یک گراف دو بخشی با تقسیم رأس‌ها به صورت $V = L \cup R$ باشد، و $G' = (V', E')$ شبکه‌ی جریان متناظر آن باشد. اگر M یک تطبیق در G باشد، آنگاه یک شار صحیح f در G' وجود دارد به‌طوری‌که $|f| = |M|$.
- بر عکس، اگر f یک جریان صحیح در G' باشد، آنگاه تطبیقی مثل M در G با اندازه‌ی $|f|$ وجود دارد که از یال‌های $(u, v) \in E$ با $f(u, v) > 0$ تشکیل شده است.
- منظور از $|M|$ همان تعداد یال‌های انبساطی M است که به آن «کاردینال»^۱ انبساطی گفته می‌شود.

^۱ cardinality

تطابق بیشینه در گراف دو بخشی

تطابق بیشینه با روش فورد-فولکرسون

- ابتدا نشان می‌دهیم که یک تطبیق M در G متناظر با یک جریان صحیح f در G' است. f را به صورت زیر تعریف می‌کنیم:

- اگر $(u, v) \in M$ ، آنگاه شار تمام یال‌های مسیر زیر را برابر ۱ قرار می‌دهیم:

$$s \rightarrow u \rightarrow v \rightarrow t$$

و شار سایر یال‌ها را برابر صفر قرار می‌دهیم.

- مشخص است که f محدودیت ظرفیت و قانون بقای جریان را ارضا می‌کند و هر یال در تطابق مانند (u, v) معادل با یک واحد جریان در G' است که از مسیر $t \rightarrow u \rightarrow v \rightarrow s$ به رأس منبع می‌رسد. علاوه براین، مسیرهای حاصل از یال‌های M به جز در رأس‌های s و t رأس-ناهمپوشان هستند.

- جریان خالص عبوری از برش $(L \cup s, R \cup t)$ برابر با $|M|$ است و بنابراین، مقدار جریان برابر $|f| = |M|$ است. (در بحث شار بیشینه دیدیم که هر شار هر برش با $|f|$ برابر است).

تطابق بیشینه در گراف دو بخشی

تطابق بیشینه با روش فورد-فولکرسون

- جهت عکس اثبات به این صورت است: فرض کنید f یک شار صحیح در G' باشد. ادعا می‌کنیم مجموعه زیر یک تطابق است.

$$M = \{(u, v) \mid u \in L, v \in R, \text{ and } f(u, v) > 0\}$$

بعد از اثبات این ادعا باید ثابت کنیم $|M| = |f|$ می‌باشد.

- برای اثبات این ادعا باید نشان دهیم هر (u, v) که شار داشته باشد قسمتی از مسیر $t \rightarrow u \rightarrow v \rightarrow s$ که از آن دقیقاً یک واحد شار می‌گذرد و بنابراین هر u و v حداقل یک بار در مجموعه M ضاهر می‌شوند.
- می‌دانیم هر رأس $L \in u$ دقیقاً یک یال ورودی دارد که ظرفیت آن ۱ است. اگر به یک $u \in L$ دقیقاً یک واحد شار وارد شود باید همان مقدار نیز خارج شود. از آنجا که f مقادیر صحیح دارد، به ازای هر $u \in L$ تنها یک یال می‌تواند دارای شار باشد. بنابراین، دقیقاً یک رأس $R \in v$ وجود دارد که $f(u, v) = 1$.

تطابق بیشینه در گراف دو بخشی

تطابق بیشینه با روش فورد-فولکرسون

- استدلال مشابهی برای هر رأس $R \in v$ برقرار است. در نتیجه، مجموعه M یک تطابق است.
- برای اینکه نشان دهیم $|M| = |f|$ ، توجه کنید که از میان یال‌های $(u, v) \in E'$ با $u \in L$ و $v \in R$ داریم:

$$f(u, v) = \begin{cases} 1 & \text{if } (u, v) \in M \\ 0 & \text{if } (u, v) \notin M \end{cases}$$

به زبان ساده (u, v) هایی که شار از آنها می‌گذرد در M اند و اگر شار آنها صفر باشد در M نیستند.

- در نتیجه، شار خالص عبوری از برش $(L \cup s, R \cup t)$ برابر $|M|$ است. بنابراین داریم

$$|f| = f(L \cup s, R \cup t) = |M|$$

تطابق بیشینه در گراف دو بخشی

تطابق بیشینه با روش فورد-فولکرسون

- امّا یک مشکل در این استدلال وجود دارد: مسئله شار بیشینه در حالت عادی محدودیتی روی صحیح بودن مقدار شار ندارد. بنابراین یک الگوریتم شار بیشینه در می‌تواند مقادیر غیر صحیح برای شار برخی یال‌ها محاسبه کند. در حالی که می‌دانیم مقدار کلی شار $|f|$ لزوماً صحیح است.
- نشان داده می‌شود که اگر تابع ظرفیت c فقط شامل مقادیر صحیح باشد، آنگاه شار بیشینه‌ی f تولید شده توسط روش فورد-فالکرسون خاصیت‌های زیر را دارد:
 - $|f|$ یک عدد صحیح است.
 - برای همهٔ رأس‌های u و v ، مقدار (u, v) یک عدد صحیح است.

تطابق بیشینه در گراف دو بخشی

تطابق بیشینه با روش فورد-فولکرسون

- از اثبات کردیم که هر تطابق در گراف دو بخشی G متناظر با یک شار در شبکه شار " G' آن است و بلعکس.
- اما آیا اندازه‌ی یک تطبیق بیشینه $(|M|)$ در یک گراف دو بخشی با اندازه یک شار بیشینه $(|f|)$ در شبکه‌ی شار متناظر برابر است؟ به عبارت دیگر اگر یک شار بیشینه در شبکه شار یافت بشود، اندازه آن با بیشینه تطابق در گراف دو بخشی برابر است؟
- برابری این دو نیاز به اثبات دارد. در ادامه اثبات آن را بررسی خواهیم کرد.

تطابق بیشینه در گراف دو بخشی

تطابق بیشینه با روش فورد-فولکرسون

- فرض کنید M یک تطابق بیشینه در G باشد و شار متناظر f در G' بیشینه نباشد. آنگاه شار f' وجود دارد به طوری که $|f'| > |f|$.
- از آنجا که ظرفیت‌ها در G' صحیح هستند، f' نیز یک شار صحیح است. در این صورت، f' متناظر با تطابقی مثل M' در G با اندازه‌ی

$$|M'| = |f'| > |f| = |M|$$

است، که با فرض بیشینه بودن M در تناقض است.

- به صورت مشابه می‌توان نشان داد که اگر f یک جریان بیشینه در G' باشد، تطابق متناظر آن نیز یک تطابق بیشینه در G است.

تطابق بیشینه در گراف دو بخشی

تطابق بیشینه با روش فورد-فولکرسون

- بنابراین، برای یافتن یک تطبیق بیشینه در یک گراف دو بخشی بدون جهت G ،
 - شبکه‌ی جریان G' را ایجاد می‌کنیم،
 - روش فورد-فالکرسون را روی G' اجرا می‌کنیم،
 - جریان صحیح بیشینه‌ی حاصل را به یک تطبیق بیشینه در G تبدیل می‌کنیم.
- هر تطبیق در یک گراف دو بخشی حداقل اندازه‌ی $\min(|L|, |R|)$ دارد، بنابراین از مرتبه $O(V)$ است. بنابراین مقدار جریان بیشینه در G' نیز $O(V)$ خواهد بود. پس، یافتن تطبیق بیشینه در یک گراف دو بخشی در زمان

$$O(VE') = O(VE)$$

. $|E'| = \Theta(E)$ انجام می‌شود، زیرا

تطابق در گراف دو بخشی

- بسیاری از مسائل دنیای واقعی را می‌توان به صورت یافتن یک تطابق در یک گراف دو‌بخشی بدون جهت مدل‌سازی کرد.
- برای مثال، فرض کنید شما می‌خواهید یک نفر را استخدام کنید و چندین داوطلب برای مصاحبه وجود دارند. هر داوطلبان تنها در بازه‌های مشخصی در دسترس است. چگونه می‌توانید برنامه‌ی مصاحبه‌ها را طوری تنظیم کنید که در هر بازه‌ی زمانی، حداقل یک داوطلب زمان‌بندی شده باشد و در عین حال، بیشترین تعداد ممکن از داوطلبان را مصاحبه کنید؟
- این مسئله را می‌توان به صورت یک مسئله‌ی تطابق در یک گراف دو‌بخشی مدل کرد، که در آن هر رأس نماینده‌ی یک داوطلب یا یک بازه‌ی زمانی است، و اگر داوطلب در آن زمان در دسترس باشد بین داوطلب و بازه‌ی زمانی یالی وجود دارد.
- اگر یالی در تطابق قرار گیرد، به این معناست که آن داوطلب در آن بازه‌ی زمانی برنامه‌ریزی شده است. هدف شما یافتن یک تطابق بیشینه است: تطابقی با بیشترین تعداد یال ممکن.

- الگوریتم‌های این فصل تطابق را در گراف‌های دوبخشی می‌یابند. هر چند در بخش شار بیشینه مسئله تطابق بیشینه معرفی شد، بهتر است مجدداً مرور مختصری از مسئله و نامگذاری‌ها انجام دهیم.
- ورودی یک گراف بدون جهت $G = (V, E)$ است که در آن $V = L \cup R$ ، به طوری که مجموعه‌های L و R از هم جدا (ناهمپوشان) هستند، و هر یال در E به یک رأس در L و یک رأس در R متصل است. بنابراین، یک تطابق در این گراف، رأس‌های L را به رأس‌های R متصل می‌کند.
- در برخی کاربردها، مجموعه‌های L و R دارای تعداد رأس‌های برابر هستند و در برخی دیگر، الزامی به تساوی اندازه‌ی آن‌ها نیست.
- می‌گوییم یک رأس $v \in V$ توسط تطابق M تطابق‌یافته^۱ است اگر یالی در M وجود داشته باشد که به v متصل باشد؛ در غیر این صورت، v تطابق‌نیافته^۲ است.

¹ matched² unmatched

- یک گراف بدون جهت لزوماً نیازی به دو بخشی بودن ندارد تا بتوان در آن از مفهوم تطابق استفاده کرد. تطابق در گراف‌های بدون جهت عمومی نیز کاربرد دارد.
- مسائل تطابق بیشینه و تطابق با وزن بیشینه در گراف‌های عمومی را می‌توان با الگوریتم‌های زمان چندجمله‌ای حل کرد که زمان اجرای آن‌ها مشابه تطابق در گراف‌های دوبخشی است، اما این الگوریتم‌ها به مراتب پیچیده‌ترند.
- گرچه مفهوم تطابق در گراف‌های عمومی نیز کاربرد دارد، اما ما تنها به گراف‌های دوبخشی می‌پردازیم.

مسیر افزایشی

- قبل از اینکه بر پایهٔ یافتن شار بیشینه برای یافتن تطابق بیشینه در گراف‌های دو بخشی را دیدیم:
- این بخش یک روش کارآمدتر به نام الگوریتم هاپکرft-کارپ^۱ ارائه می‌دهد که در زمان $O(\sqrt{|V|} \cdot |E|)$ اجرا می‌شود.
- برای این منظور ابتدا باید با مفهومی با نام مسیر افزایشی آشنا شویم.

^۱ Hopcroft-Karp

- یک تطابق حداکثری^۱ تطابقی است که نتوان یال دیگری به آن افزود؛ یعنی برای هر یال $e \in E - M$ مجموعه‌ی $e \cup M$ یک تطابق نخواهد بود. یک تطابق بیشینه همواره حداکثری است، اما عکس آن لزوماً برقرار نیست.
- بسیاری از الگوریتم‌ها برای یافتن تطابق بیشینه (از جمله الگوریتم هاپکرفت-کارپ) با افزایش تدریجی اندازه‌ی تطابق عمل می‌کنند.
- فرض کنید تطابق M در گراف بدون جهت $G = (V, E)$ داده شده باشد. یک «مسیر متناوب نسبت به M ^۲
- مسیری ساده است که یال‌های آن به صورت یکی در میان متعلق به M و $E - M$ باشند.
- یک «مسیر افزایشی نسبت به M ^۳
- مسیر افزایشی نسبت به M ، یک مسیر متناوب نسبت به M است که یال‌های ابتدا و انتهای آن در M نباشند. چنین مسیری همیشه دارای تعداد فردی از یال‌ها است.

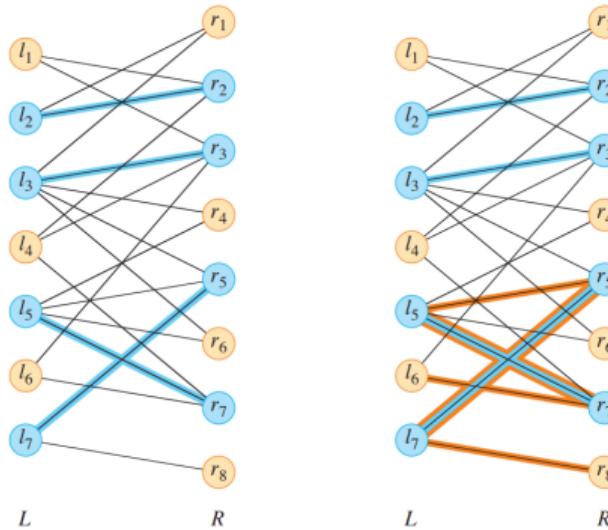
¹ maximal

² M-alternating path

³ M-augmenting path

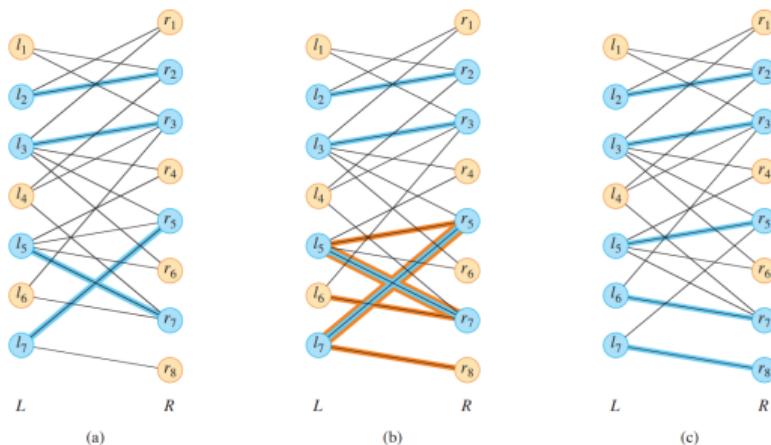
مسیر افزایشی

- در شکل زیر قسمت a یک تطابق با اندازه ۴ را در یک گراف دو بخشی بدون جهت نشان می‌دهد و قسمت b یک مسیر افزایشی نسبت به تطابق شکل قبل که دارای ۵ یال می‌باشد را به تصویر می‌کشد.



مسیر افزایشی

- فرض کنید یک تطابق M داشته باشیم و همچنین یک مسیر افزایشی P نسبت به آن وجود داشته باشد. جالب است بدانید اگر یال‌های مشترک P و M را از M حذف کنیم و بقیه یال‌های P را به M اضافه کنیم، نتیجه یک تطابق می‌شود که از M یک یال بیشتر دارد.
- قسمت c شکل زیر همان تطابق قسمت a را نشان می‌دهد که مسیر افزایشی قسمت b روی آن اعمال شده.



تطابق در گراف دو بخشی

- عملیات اعمال یک مسیر افزایشی به یک تطابق را می‌توان با عملگر تفاضل متقارن^۱ که یکی از عملگرهای تعریف شده بر روی مجموعه‌هاست، نشان داد.
- تفاضل متقارن دو مجموعه به این صورت تعریف می‌شود؛

$$X \oplus Y = X - Y \cup Y - X$$

يعنى عناصرى که به X یا Y تعلق دارند، اما نه به هر دو.

^۱ symmetric difference

- پیشتر دیدیم که بعد از اعمال این عملیات $M' = M \oplus P$ مجموعه M' یک تطابق است که اندازه آن یک واحد از M بیشتر است.
- برای فهم بهتر این مسئله فرض کنید مسیر افزایشی P شامل ۳ یال باشد؛

$$e_1 = (v_1, v_2), e_2 = (v_2, v_3), e_3 = (v_3, v_4)$$

می‌دانیم رأس‌های v_1 و v_4 تطابق‌نیافته هستند و بقیه‌ی رأس‌های آن تطابق‌یافته‌اند. (در غیر این صورت در تطابق M بیش از یک یال به این رأس‌ها متصل می‌شوند.)

- در مسیر افزایشی یال اول و آخر در تطابق قرار ندارند بنابراین e_1 و e_3 در M قرار ندارند، و یال e_2 در M قرار دارد.

- تفاضل متقارن $M' = M \oplus P$, نقش یال‌ها را برعکس می‌کند؛ یال e_2 را در از M' حذف می‌کند و یال‌های e_1 و e_3 در M' قرار می‌دهد.
- با حذف یال e_2 از M رأس‌های v_4, v_2, v_3, v_1 هیچ یال متصلی در M' ندارند بنابراین با افزودن e_1 و e_3 مشکلی ایجاد نمی‌شود.
- از میان سه یال e_1, e_2, e_3 یک یال در M' وجود دارد. بنابراین، تطابق M' نسبت به M یک یال بیشتر دارد و هیچ رأس یا یال دیگری در G تحت تأثیر تغییر M به M' قرار نمی‌گیرد.
- در نتیجه، M' یک تطابق در G است و

$$|M'| = |M| + 1$$

- لم مهمی در بحث تطابق بیشینه وجود دارد که در آینده از آن در اثبات شرط خاتمه الگوریتم استفاده می‌کنیم.
- فرض کنید M^* و M دو تطابق در گراف $G = (V, E)$ باشند، و گراف $G' = (V, E')$ را در نظر بگیرید که یال‌های آن حاصل تقاضل متقارن این دو تطابق باشند. این لم بیان می‌کند که اگر $|M^*| > |M|$ باشد، آنگاه گراف G' حداقل شامل $|M^*| - |M|$ مسیر متناوب نسبت به M است که رأس‌های این مسیرها از یکدیگر مجزا‌اند.
- اثبات: هر رأس گراف G' تنها می‌تواند درجه ۱ یا ۲ داشته باشد زیرا حداکثر یک یال متصل به هر رأس در یک تطابق وجود دارد. بنابراین اگر رأسی با درجه ۲ در G' وجود داشته باشد، قطعاً یکی از یال‌های متصل به آن متعلق به M^* و دیگری متعلق به M است.

مسیر افزایشی

- حال بباید رأس‌های این گراف را بررسی کنیم؛ رأس‌هایی که درجه \circ دارند به هیچ جای دیگر از گراف متصل نیستند. رأس‌های با درجه ۱ و ۲ تنها می‌توانند یک مسیر ساده یا یک دور ساده را تشکیل دهند. همچنین این مسیرها و دورها نمی‌توانند رأس مشترکی داشته باشند زیرا در غیر این صورت حداقل یک رأس باید از درجه ۳ یا بیشتر باشد.
- همچنین دیدیم که هر رأس با درجه ۲ به یک یال از M^* و یک یال از M متصل است و در دور همه رأس‌ها از درجه ۲ هستند. پس دورهای گراف G' نمی‌توانند تعداد فرد یال داشته باشند (چرا؟). همچنین نیمی از یال‌های هر دور به M^* و نیمی دیگر به M تعلق دارند.
- به علاوه یال‌های هر مسیر هم به طور متناوب از دو تطابق است زیرا رأس‌های درونی یک مسیر نیز درجه ۲ هستند.

مسیر افزایشی

- بنابراین، هر مؤلفه‌ی همبند در G' یا یک رأس منفرد، یا یک دور ساده با طول زوج که یال‌های آن به صورت متناوب از M و M^* هستند، و یا یک مسیر ساده با یال‌های متناوب از M و M^* است.
- فرض کردیم M^* از M تعداد یال بیشتری داشته باشد. پس مجموعه‌ی یال‌های E' باید $|M^*| - |M|$ یال بیشتر از M^* نسبت به M داشته باشد (چرا؟).
- دیدیم که تنها مؤلفه‌های همبندی موجود در G' که دارای یال هستند دور و مسیر هستند. پس تعداد یال‌های متعلق به M^* باید در این مؤلفه‌ها بیشتر باشند. اما در هر دور تعداد یال‌های برابری از هر دو تطابق استفاده شده. بنابراین، این مسیرهای ساده در G' هستند که عامل این اختلاف در تعداد یال‌ها از M^* نسبت به M می‌باشند.

مسیر افزایشی

- مسیرهایی که تعداد یال‌های برابری از هر تطابق داشته باشند نیز تأثیری ندارند. اما مسیری که تعداد متفاوتی از یال‌های M و M^* را دارا باشد دو نوع دارد:
 - ۱ یا با یال‌هایی از M شروع و پایان می‌یابد (و یک یال بیشتر از M نسبت به M^* دارد).
 - ۲ یا با یال‌هایی از M^* شروع و پایان می‌یابد (و یک یال بیشتر از M^* نسبت به M دارد).
- از آنجایی که E' شامل $|M^*| - |M|$ یال بیشتر از M^* است، باید حداقل $|M^*| - |M|$ مسیر از نوع دوم وجود داشته باشد، توجه کنید این مسیرها تمام ویژگی‌های یک مسیر افزایشی نسبت به M را دارند و دیدیم تمام این مسیرها از هم مجزا هستند. بنابراین لم بالا ثابت می‌شود.

- حال اگر یک الگوریتم با افزودن تدریجی یال‌ها تطابق بیشینه را بیابد، چگونه تشخیص می‌دهد که باید متوقف شود؟ ثابت خواهیم کرد «زمانی که دیگر هیچ مسیر افزایشی وجود نداشته باشد» الگوریتم باید متوقف شود. به طور دقیق‌تر:
- M یک تطابق بیشینه است ($p \Leftrightarrow$ هیچ مسیر افزایشی نسبت به M در گراف وجود نداشته باشد (q))
- اثبات: باید ثابت کنیم $(q \Rightarrow p)$ (جهت اول) و $(p \Rightarrow q)$ (جهت دوم) برقرار هستند. به جای این کار می‌توانیم عکس و نقیض^۱ هر دو گزاره را اثبات کنیم.

^۱ contrapositive

- عکس و نقیض جهت اول ($p \Rightarrow q$) : اگر یک مسیر افزایشی نسبت به تطابقی وجود داشته باشد، آنگاه آن تطابق بیشینه نیست.
- اثبات جهت اول: اگر یک مسیر افزایشی نسبت به M ، به نام P وجود داشته باشد، آنگاه تطابق $M \oplus P$ یک یال بیشتر از M دارد، بنابراین M نمی‌تواند تطابق بیشینه باشد.
- عکس و نقیض جهت دوم ($q \Rightarrow p$) : اگر تطابقی بیشینه نباشد، آنگاه حداقل یک مسیر افزایشی نسبت به آن در گراف وجود دارد.
- اثبات جهت دوم: اگر M^* یک تطابق بیشینه باشد به طوری که $|M^*| > |M|$. آنگاه طبق لم قبلی، گراف G حداقل شامل $0 < |M^*| - |M|$ مسیر افزایشی با رأس‌های مجزا نسبت به M است.

معرفی الگوریتم هاپکرفت-کارپ

- تا به اینجا می‌توانیم الگوریتمی برای یافتن یک تطابق بیشینه طراحی کنیم که در زمان $O(VE)$ اجرا می‌شود.
- به این صورت که با یک تطابق تهی M آغاز می‌کنیم. سپس به صورت تکراری، یکی از گونه‌های جستجوی اول سطح یا جستجوی اول عمق را از یک رأس تطابق‌نیافته آغاز می‌کنیم تا مسیری متناوب بیابیم که به یک رأس دیگر که تطابق‌نیافته است ختم شود.
- از مسیر افزایشی حاصل شده برای افزودن یک یال به M استفاده می‌کنیم، تا اندازه‌ی آن یک واحد افزایش یابد.
- اما هاپکرفت-کارپ زمان اجرای بهتری از این الگوریتم اولیه‌ایی ارائه می‌دهد. برای معرفی این الگوریتم ابتدا باید با لم دیگری که در این الگوریتم به کار می‌رود آشنا شویم.

معرفی الگوریتم هاپکرفت-کارپ

- ثبت می‌شود که اگر k مسیر افزایشی مجرا داشته باشیم (رأس‌های مشترک نداشته باشند) می‌توان آنها را اجتماع گرفت و نیجه را با تطابق تفاضل متقارن گرفت. در این صورت اندازه تطابق k واحد افزایش می‌یابد:

$$M' = M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)$$

- برای اثبات این ادعا باید استدلال کنید که مجرا بودن مسیرها باعث می‌شود:

$$P_1 \cup P_2 \cup \dots \cup P_k = P_1 \oplus P_2 \oplus \dots \oplus P_k$$

سپس با استفاده از استقراء روی i ، می‌توان نشان داد که تطابق

$$M' = M \oplus (P_1 \cup P_2 \cup \dots \cup P_{i-1})$$

دارای $|M| + (i - 1)$ یال است، و مسیر P_i یک مسیر افزایشی نسبت به این تطابق جدید است.

معرفی الگوریتم هاپکرفت-کارپ

- الگوریتم هاپکرفت-کارپ زمان اجرا را به $O(\sqrt{VE})$ بهبود میبخشد. روال HOPCROFT-KARP یک گراف دوبخشی بدون جهت را دریافت میکند و مکرراً اندازهی تطابق را افزایش میدهد.

Algorithm HOPCROFT-KARP

```
1:  $M \leftarrow \emptyset$ 
2: repeat
3:   let  $P = \{P_1, P_2, \dots, P_k\}$  be a maximal set of vertex-disjoint shortest
    $M$ -augmenting paths
4:    $M \leftarrow M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)$ 
5: until  $P = \emptyset$ 
6: return  $M$ 
```

معرفی الگوریتم هاپکرفت-کارپ

- امّا چطور زمان اجرای الگوریتم برابر با $O(\sqrt{VE})$ است؟
- خواهیم دید که حلقه‌ی `repeat` در خطوط ۲ تا ۵، تعداد $O(\sqrt{V})$ بار تکرار می‌شود و اینکه چگونه خط ۳ را می‌توان به‌گونه‌ای پیاده‌سازی کرد که در هر تکرار، در زمان $O(E)$ اجرا شود.
- باید ابتدا ببینیم چگونه می‌توان یک مجموعه‌ی حداقلی از مسیرهای افزایشی مجزا نسبت به M که اندازه آنها برابر با کوتاهترین مسیر افزایشی موجود است، را در زمان $O(E)$ پیدا کرد. این کار در سه مرحله انجام می‌شود:

معرفی الگوریتم هاپکرفت-کارپ

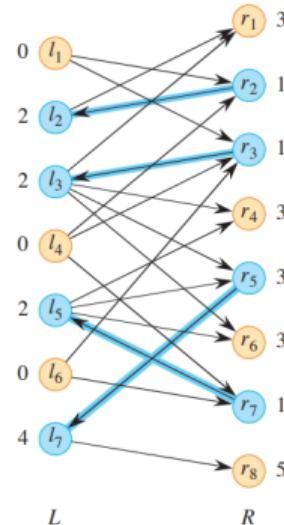
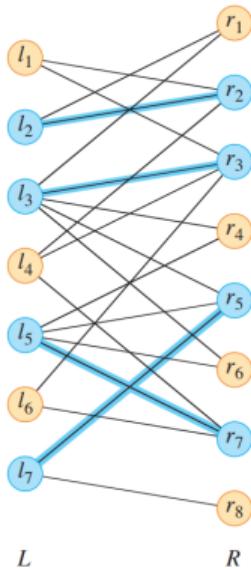
- مرحله‌ی اول، یک نسخه‌ی جهت‌دار از گراف دوبخشی بدون جهت G با نام G_M ایجاد می‌کند.
- مرحله‌ی دوم، از طریق گونه‌ای از جست‌وجوی اول عرض، یک گراف جهت‌دار بی‌دور با نام H از G_M می‌سازد.
- مرحله‌ی سوم، با اجرای گونه‌ای از جست‌وجوی اول عمق بر روی گراف معکوس‌شده‌ی H ، که آن را H^T می‌نامیم، یک مجموعه‌ی بیشینه از کوتاهترین مسیرهای افزایشی مجزا می‌یابد.
- در گراف معکوس‌شده‌ی یک گراف جهت‌دار، جهت هر یال وارونه می‌شود. از آنجا که H بی‌دور است، H^T نیز بی‌دور خواهد بود.

معرفی الگوریتم هاپکرفت-کارپ

- فرض کنید یک مسیر افزایشی نسبت به تطابقی مانند M از یک رأس تطابق نیافته در L آغاز می‌شود.
 - در این صورت یال‌های این مسیر که از L به R می‌روند، متعلق به مجموعه M نیستند، و یال‌های مسیر که از R به L می‌روند، در تطابق M قرار دارند. (به دلیل ویژگی تناوب یال‌های مسیر افزایشی می‌توان چنین نتیجه گرفت)
 - بنابراین، مرحله اول گراف جهت‌داری به نام $G_M = (V, E_M)$ را ایجاد می‌کند که یال‌های آن مطابق با توضیحات فوق جهت‌دهی شده‌اند:
- $$E_M = \{(l, r) \mid l \in L, r \in R, (l, r) \in E - M\} \cup \{(r, l) \mid r \in R, l \in L, (l, r) \in M\}$$
- به زبان ساده، در گراف جدید یال‌های از L به R در تطابق نیستند و یال‌های از R به L در تطابق قرار دارند.

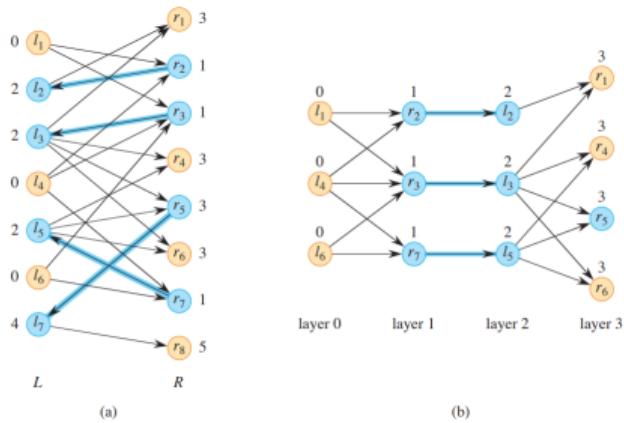
معرفی الگوریتم هاپکرفت-کارپ

- شکل سمت چپ یک گراف دو بخشی بدون جهت G و یک تطابق M بر روی آن را نشان می‌دهد و شکل سمت راست گراف جهت دار G_M را نشان می‌دهد.



معرفی الگوریتم هاپکرفت-کارپ

- در مرحله اول گراف G_M ایجاد شد. در مرحله دوم یک جهت‌دار بدون دور (DAG) ایجاد می‌شود که، دارای چند لایه رأس است.



- شکل b گراف H را نشان می‌دهد که متناظر با گراف جهت‌دار G_M در شکل b است. هر لایه فقط شامل رأس‌هایی از مجموعه L یا فقط از مجموعه R است و لایه‌ها به صورت متناوب از L و R تشکیل می‌شوند.

معرفی الگوریتم هاپکرفت-کارپ

- رأسی مثل v را در نظر بگیرید. فرض کنید در گراف G_M کمترین فاصله‌ایی که یک رأس تطابق نیافته در مجموعه L با v دارد ۳ باشد. آنگاه رأس v در لایه سوم از H قرار می‌گیرد. این فاصله را با d نشان می‌دهیم و فرض می‌کنیم هر رأس یک مقدار d دارد.
- بنابراین با کمی بررسی در می‌یابید که رأس‌های متعلق به L در لایه‌های زوج ظاهر، و رأس‌های متعلق به R در لایه‌های فرد ضاهر خواهند شد.
- اما همه رأس‌ها در H ضاهر نمی‌شوند. درواقع تنها رأس‌هایی که مقدار d آنها کمتر یا مساوی مقدار q باشد وارد H می‌شوند. پس آخرین لایه در H شامل رأس‌هایی از R با مقدار d برابر q خواهد بود.
- مقدار q که در گراف G_M تعریف می‌شود برابر است با: کمترین مقدار d در بین رئوس تطابق نیافته مجموعه R .

معرفی الگوریتم هاپکرفت-کارپ

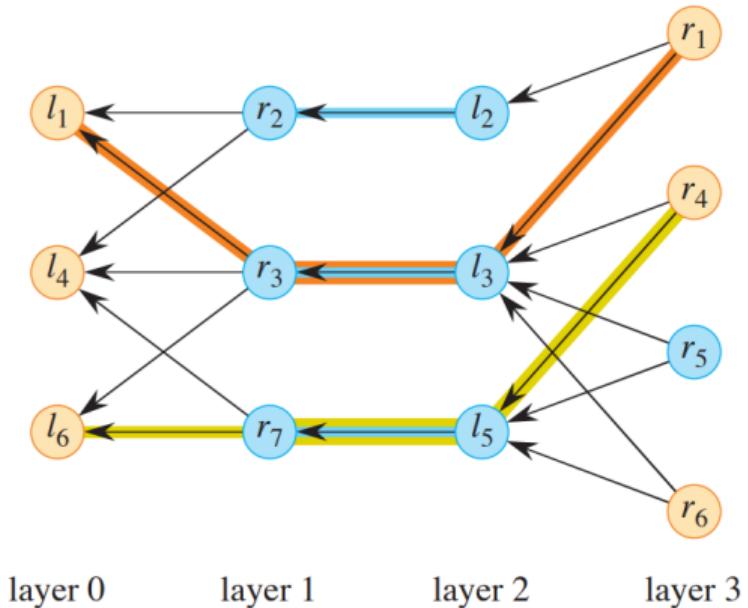
- یال‌های موجود در E_H زیرمجموعه‌ای از E_M هستند و به صورت زیر تعریف می‌شوند:

$$E_H = \{(l, r) \in E_M \mid r.d \leq q \text{ and } r.d = l.d + 1\} \cup \{(r, l) \in E_M \mid l.d \leq q\}$$

- برای تعیین مقدار d همه رأس‌ها، باید نوعی پیمایش اول سطحی را روی G_M اجرا کرد، با این تفاوت که شروع BFS نه از یک رأس، بلکه از تمام رأس‌های بدون تطابق در L انجام می‌گیرد. (توجه کنید که این کار با چند بار اجرای BFS متفاوت است).
- هر مسیر در H که از یک رأس در لایه 0 به یک رأس تطابق نیافته در لایه q منتهی شود، متناظر با یک مسیر افزایشی در گراف اصلی است و اندازه همه این مسیرها برابر با کوتاهترین مسیر افزایشی موجود در گراف است. کافی است نسخه بدون جهت از یال‌های جهت‌دار در H را استفاده کرد.
- علاوه بر این، همه کوتاهترین مسیرهای افزایشی نسبت به M در H وجود دارند.

معرفی الگوریتم هاپکرفت-کارپ

- مرحله سوم یک مجموعه حداکثری از کوتاهترین مسیرهای افزایشی مجزا نسبت به M را می‌یابد. این مرحله با ایجاد ترانهاده گراف H آغاز می‌شود. در شکل زیر H^T نمایش داده شده است.

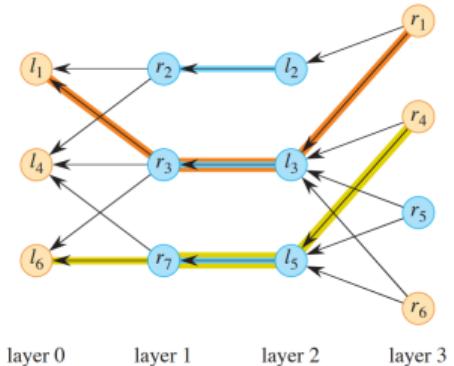


معرفی الگوریتم هاپکرفت-کارپ

- پس از تشکیل H^T از هر رأس تطابق نیافته در لایه q نوعی از جستجوی اول عمق اجرا می‌شود و با هر بار انجام جستجو حداکثر یک مسیر یافت می‌شود.
- در این جستجو باید از اشاره‌گر پدر استفاده کرد. یه این صورت که پس از رسیدن به یک رأس در لایه 0 ، با دنبال‌کردن اشاره‌گرهای پدر، یک مسیر افزایشی یافت می‌شود.
- این جستجو در دو حالت متوقف می‌شود: (الف) تمام مسیرهای ممکن را بدون رسیدن به لایه 0 بررسی کرده باشد. (ب) به یک رأس کاوش نشده در لایه 0 برسد. (اگر این جمله نامفهوم است به ادامه توضیحات توجه کنید)،
- در جستجوی اول عمق استاندارد تا زمانی که الگوریتم به بن‌بست نخورده جستجو ادامه می‌یابد. اما توجه کنید در اینجا پس از رسیدن به یک رأس در لایه 0 جستجو متوقف می‌شود. بنابراین معمولاً تعداد زیادی از رئوس کاوش نشده باقی می‌ماند.
- در جستجوی اول عمق استاندارد رئوس تنها می‌توانند یکبار کاوش شوند. در اینجا یک تفاوت اساسی وجود دارد: هر رأس در تمامی جستجوهای انجام شده تنها یک بار می‌تواند کاوش شود.

معرفی الگوریتم هاپکرفت-کارپ

- برای درک بهتر این نسخه از جستوی اول عمق بار دیگر به شکل زیر دقت کنید:



- اولین جستجو از رأس r_1 آغاز می‌شود و مسیر

$$(r_1, l_3), (l_3, r_3), (r_3, l_1)$$

را می‌یابد که با رنگ نارنجی مشخص شده. و رئوس r_1, l_3, r_3, l_1 را در حالت کاوش شده قرار می‌دهد.

معرفی الگوریتم هاپکرفت-کارپ

- جستجو بعدی از رأس r_4 شروع می‌شود که ابتدا یال (r_4, l_3) را بررسی می‌کند. اما l_4 در جستجوی قبلی کاوش شده. این جستجو در نهایت مسیر

$$(r_4, l_5), (l_5, r_7), (r_7, l_6)$$

را می‌یابد که با زرد مشخص شده است. و رئوس r_4, l_5, r_7, l_6 را کاوش می‌کند.

- سپس جستجو از رأس r_6 آغاز می‌شود که به رئوس l_3, l_5 برمی‌خورد که از قبل کاوش شده اند و بنابراین این جستجو همینجا متوقف می‌شود.

معرفی الگوریتم هاپکرفت-کارپ

- نتیجه اجرای این الگوریتم یک مجموعه حداکثری شامل دو کوتاهترین مسیر افزایشی مجزا می‌باشد:

$$\{(r_1, l_3), (l_3, r_3), (r_3, l_1) \text{ و } (r_4, l_5), (l_5, r_7), (r_7, l_6)\}$$

توجه کنید که طول همه این مسیرها و یکسان و برابر با طول کوتاهترین مسیر افزایشی گراف اصلی هستند.

- در اینجا می‌توانیم حداکثری بودن این مجموعه را بهتر درک کنیم. به مجموعه بالا هیچ مسیر افزایشی دیگری نمی‌توان افزود به گونه‌ایی که با مسیرهای فعلی رأس مشترک نداشته باشد بنابراین این مجموعه حداکثری است. اما بیشینه نیست زیرا مجموعه زیر وجود دارد که دارای سه کوتاهترین مسیر افزایشی مجزا است:

$$(r_1, l_2), (l_2, r_2), (r_2, l_1), (r_4, l_3), (l_3, r_3), (r_3, l_4) \text{ و } (r_6, l_5), (l_5, r_7), (r_7, l_6)$$

- اما الگوریتم هاپکرفت-کارپ الزامی به یافتن مجموعه بیشینه ندارد و یافتن مجموعه حداکثری کفایت می‌کند.

پیچیدگی زمانی هاپکرفت-کارپ

- بررسی نحوه کار الگوریتم هاپکرفت-کارپ به اتمام رسید. در ادامه به بررسی پیچیدگی زمانی این الگوریتم می پردازیم.
- ابتدا می خواهیم ثابت کنیم هر سه مرحله یافتن مسیرهای افزایشی در زمان $O(E)$ انجام می شود. پیش از این فرض کردیم که در گراف دو بخشی هر رأس حداقل به یک یال متصل است پس $|V| = O(V)$ و درنتیجه

$$|V| + |E| = O(E)$$

- مرحله اول گراف G_M را صرفاً با جهت دادن به یالهای گراف اصلی می سازد. تعداد یالها و رأسهای آن با گراف اصلی برابر است. هر رأس حداقل یک بار پیمایش می شود پس این مرحله از مرتبه $O(E)$ است.

پیچیدگی زمانی هاپکرفت-کارپ

- مرحله دوم یک جستجوی اول سطح روی G_M انجام می‌دهد. مرتبه جستجوی اول سطح $O(V_M + E_M)$ است که برابر است با:

$$O(V_M + E_M) = O(E_M) = O(E)$$

- ساختن گراف H از مرتبه $O(V_H)$ زمان می‌برد. رئوس و یال‌ها در این گراف زیر مجموعه‌ایی از گراف G_M هستند بنابراین

$$O(V_H + E_H) = O(E)$$

- بنابراین دیدم که اجرای این سه مرحله از مرتبه زمانی $O(E)$ است.

پیچیدگی زمانی هاپکرفت-کارپ

- بار دیگر به شبه کد الگوریتم هاپکرفت-کارپ دقت کنید؛ دیدیم که خط ۳ در زمان $O(E)$ اجرا می‌شود. بروزرسانی تطابق در خط ۴ هم در زمان $O(E)$ انجام می‌شود. کافیست یک بار روی همه یال‌های مسیرهای افزایشی پیمایش انجام داد و هر کدام را از تطابق حذف کرد یا به آن اضافه کرد.
- بنابراین هر دور از حلقه در زمان $O(E)$ انجام می‌شود اما این حلقه چند بار اجرا می‌شود؟ خواهیم دید که دفعات اجرای این حلقه از مرتبه $O(\sqrt{|V|})$ است.
- برای اثبات پیچیدگی زمانی مطرح شده باید ابتدا لمی را ثابت کنیم که بیان می‌کند پس از هر تکرار حلقه، طول کوتاهترین مسیر افزایشی زیاد می‌شود.

پیچیدگی زمانی هاپکرفت-کارپ

- فرض S یک مجموعهٔ ماکسیمال از کوتاهترین مسیرهای افزایشی با طول q باشد که توسط الگوریتم یافت شده. اعمال مسیرهای افزایشی تطابق M' را به این صورت تشکیل می‌دهد:

$$M' = M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)$$

- فرض کنید P یک کوتاهترین مسیر افزایشی نسبت به M' باشد. در این صورت، می‌خواهیم ثابت کنیم طول P بیشتر از q است.
- اثبات: دو حالت ممکن را جداگانه بررسی می‌کنیم: ۱) حالتی که P از مسیرهای موجود در S مجزا باشد، ۲) حالتی که P حداقل با یکی از آن‌ها رأس مشترک داشته باشد.

پیچیدگی زمانی هاپکرفت-کارپ

- در حالت اول P شامل یال‌هایی از M است که در هیچ‌یک از مسیرهای P_1, P_2, \dots, P_k وجود ندارند؛ پس یال‌های این مسیر پس از اعمال تغییرات دست نخورده باقی می‌مانند. بنابراین، P یک مسیر افزایشی نسبت به M نیز هست.
- مسیر P از مسیرهای P_1, \dots, P_k مجزا است ولی خود یک مسیر افزایشی نسبت به M است. از آنجا که یک مجموعهٔ حداکثری است، نتیجه می‌گیریم طول P بیش از q است و به همین دلیل نتوانسته عضو S شود.

پیچیدگی زمانی هاپکرفت-کارپ

- در حالت دوم فرض می‌کنیم P حداقل با یکی از مسیرهای افزایشی در S رأس مشترک داشته باشد.
- در روند اثبات از یک متغیر کمکی A استفاده می‌کنیم که به این صورت تعریف می‌شود:

$$A = M \oplus M' \oplus P$$

A مجموعه‌ایی از یال‌ها می‌باشد.

- همچنین به دلیل طولانی بودن اثبات حالت دوم، آن را به سه بخش تقسیم می‌کنیم تا فهم آن ساده‌تر شود.

پیچیدگی زمانی هاپکرفت-کارپ

- در بخش اول می‌خواهیم یک حد پایین برای اندازه مجموعه A بیابیم.
- می‌دانیم که $M' \oplus P$ خود یک تطابق است که آن را M^* می‌نامیم. حال بار دیگر به تعریف A دقت کنید؛

$$A = M \oplus M' \oplus P = M \oplus M^*$$

- پیش‌تر ثابت کردیم تفاضل متقارن دو تطابق، مجموعه‌ایی از مسیرهای افزایشی مجزا را نتیجه می‌دهد که تعداد آنها برابر با تفاضل انداز تطابق‌هاست. بنابراین A شامل حداقل $|M^*| - |M|$ مسیر افزایشی است.
- حال بیاید اندازه M^* را محاسبه کنیم:

$$|M^*| = |M' \oplus P| = |M'| + 1 = |M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)| + 1 = |M| + k + 1$$

پیچیدگی زمانی هاپکرفت-کارپ

- بنابراین تعداد مسیرهای مجازی موجود در A برابر است با:

$$|M^*| - |M| = (|M| + k + 1) - (|M|) = k + 1$$

- مجموعه A شامل یال‌های $k + 1$ مسیر مجاز است که هر کدام حداقل q یال دارد، نتیجه می‌گیریم:

$$(k + 1)q \leq |A|$$

بنابراین:

$$kq + q \leq |A|$$

پیچیدگی زمانی هاپکرفت-کارپ

- در بخش دوم می‌خواهیم ثابت کنیم که P با حداقل یکی از مسیرهای S یال مشترک دارد.
- می‌دانیم در هر مسیر افزایشی نسبت به M همه رئوس به غیر از رأس اول و آخر، تحت M تطابق یافته اند ولی با اعمال $M \oplus P_i$ تمام رأس‌های P_i از جمله رأس اول و آخر تطابق یافته می‌شوند.
- از آنجا که مسیرها مجزا هستند مسیرهای دیگری که بر تطابق اعمال می‌شوند نمی‌توانند وضعیت تطابق یافته بودن رئوس P_i را تحت تأثیر قرار دهند.
- بنابراین تحت تطابق M' ، تمام رأس‌های موجود در همه مسیرهای مجموعه S تطابق یافته هستند.

پیچیدگی زمانی هاپکرفت-کارپ

- حال فرض کنید که مسیر P با مسیر S یک رأس مشترک v داشته باشد. دیدم که همه رئوس مسیر P_i در نسبت به M' تطابق یافته هستند. پس v هم تطابق یافته است.
- بنابراین v نمی‌تواند یکی از دو سر انتهایی مسیر P باشد، زیرا این دو رأس در M' تطابق نیافته هستند. پس رأس v یکی از رئوس میانی P است و به دلیل ویژگی تناوب یال‌ها در مسیر افزایشی، v باید به یالی مانند e_1 عضو M' متصل باشد به طور دقیق‌تر:

$$e_1 \in M' \text{ and } e_1 \in P$$

پیچیدگی زمانی هاپکرفت-کارپ

- رأس v می‌تواند در انتهای یا در میانه مسیر P_i باشد.
 - الف اگر v در میانه مسیر باشد: یالی که در P_i عضو M باشد عضو M' نیست و بلعکس. پس ویژگی تناوب این مسیر نسبت به M' هم برقرار است یعنی یال‌های P_i به طور متناوب عضو M' هستند. پس یکی از یال‌های متصل به v عضو M' است.
 - ب اگر v در یکی از دو انتهای مسیر باشد: P_i مسیر افزایشی نسبت به M است. پس دو یال انتهایی آن که عضو M نبودند حال عضو M' هستند. پس یکی از یال‌های متصل به v عضو M' است.
- نتیجه می‌گیریم v باید به یالی مانند e_2 متصل باشد به طوری که:
$$e_2 \in M' \text{ and } e_2 \in P_i$$
- هر رأس حداکثر به یک یال از یک تطابق متصل است، بنابراین e_1 و e_2 نمی‌توانند دو یال متفاوت باشند. پس ثابت می‌شود P با حداقل یکی از مسیرهای S یال مشترک دارد.

پیچیدگی زمانی هاپکرفت-کارپ

- در بخش سوم می‌خواهیم نشان دهیم:

$$A = (P_1 \cup P_2 \cup \dots \cup P_k) \oplus P$$

- داریم:

$$A = M \oplus M' \oplus P$$

$$= M \oplus (M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)) \oplus P$$

$$= (M \oplus M) \oplus (P_1 \cup \dots \cup P_k) \oplus P \quad (\text{با توجه به شرکت پذیری } \oplus)$$

$$= \emptyset \oplus (P_1 \cup \dots \cup P_k) \oplus P \quad (X \oplus X = \emptyset \text{ برای هر } X)$$

$$= (P_1 \cup \dots \cup P_k) \oplus P \quad (\emptyset \oplus X = X)$$

پیچیدگی زمانی هاپکرفت-کارپ

- دیدم که A برابر است با $(P_1 \cup P_2 \cup \dots \cup P_k) \oplus P$ و چون مسیر P با حداقل یکی از مسیرهای $P_i \in P$ یا مشترک دارد، بنابراین خواهیم داشت:

$$|A| < |P_1 \cup P_2 \cup \dots \cup P_k| + |P|$$

از طرف دیگر:

$$\begin{aligned} kq + q &\leq |A| \\ &< |P_1 \cup P_2 \cup \dots \cup P_k| + |P| \\ &= kq + |P| \end{aligned}$$

پیچیدگی زمانی هاپکرفت-کارپ

- دریافتیم که رابطه زیر برقرار است:

$$kq + q < kq + |P|$$

- حال به خط زدن kq از طرفین نتیجه می‌شود $|P| < q$ و اثبات کامل می‌شود. پس ثابت کردیم اندازه کوتاه‌ترین مسیرهای افزایشی پس از هر تکرار از هاپکرفت-کارپ افزایش می‌یابد.
- برای اثبات پیچیدگی زمانی هاپکرفت-کارپ باید یک لم دیگر را نیز بررسی کنیم. این لم یک حد بالا برای اندازه تطابق مشخص می‌کند.

پیچیدگی زمانی هاپکرفت-کارپ

- فرض کنید طول کوتاهترین مسیر افزایشی نسبت به تطابق M در گراف $G = (V, E)$ برابر با q باشد. می خواهیم ثابت کنیم اندازه یک تطابق بیشینه در G حداقل برابر است با:

$$|M| + \frac{|V|}{q+1}$$

- اثبات: فرض کنید M^* یک تطابق بیشینه باشد. گراف G شامل حداقل $|M^*| - |M|$ مسیر افزایشی مجزا نسبت به M است. هر یک از این مسیرها حداقل شامل q یال است، و در نتیجه حداقل $1 + q$ رأس دارد.
- از آنجا که این مسیرها مجزا هستند، داریم:

$$(|M^*| - |M|)(q + 1) \leq |V|$$

و بنابراین:

$$|M^*| \leq |M| + \frac{|V|}{q+1}$$

پیچیدگی زمانی هاپکرفت-کارپ

- حال می‌توانیم نشان دهیم شبیه کد الگوریتم HOPCROFT-KARP حلقة خطوط ۲ تا ۵ از مرتبه $O(\sqrt{|V|})$ بار تکرار می‌شود.
- اثبات: طول کوتاهترین مسیرهای افزایشی که در خط ۳ پیدا می‌شوند، در هر تکرار افزایش می‌یابد. بنابراین، پس از $\sqrt{\lceil |V| \rceil}$ تکرار، خواهیم داشت:

$$\sqrt{\lceil |V| \rceil} \leq q$$

پیچیدگی زمانی هاپکرفت-کارپ

- حال، شرایط را پس از اولین باری بررسی می‌کنیم که خط ۴ اجرا می‌شود و مسیرهای افزایشی با طول حداقل $\sqrt{|V|}$ را اعمال می‌کند.
- در هر تکرار، اندازه تطابق حداقل به اندازه یک یال افزایش می‌یابد. در بدترین حالت هر تکرار تنها یک واحد به اندازه تطابق اضافه می‌کند. پس حداقل تعداد تکرارهای باقی‌مانده برای رسیدن به یک تطابق بیشینه برابر است با:

$$|M^*| - |M|$$

- دیدیم که این مقدار برابر است با:

$$\frac{|V|}{q+1} = \frac{|V|}{\sqrt{|V|} + 1} < \frac{|V|}{\sqrt{|V|}} = \sqrt{|V|}$$

پیچیدگی زمانی هاپکرفت-کارپ

- بنابراین، تعداد کل تکرارهای حلقه کمتر از $2\sqrt{|V|}$ است. پس مرتبه اجرای الگوریتم برابر است با:

$$O(2\sqrt{|V|}E) = O(\sqrt{|V|}E)$$

مسئله ازدواج پایدار

- در بخش قبلی، هدف یافتن یک تطابق بیشینه در یک گراف دوبخشی بدون جهت بود. اگر بدانیم که گرافی دوبخشی کامل است (یعنی از هر رأس در L به همه رئوس در R یال وجود دارد) آنگاه می‌توان با یک الگوریتم حریصانه ساده یک تطابق بیشینه یافت.
- زمانی که یک گراف می‌تواند چندین تطابق مختلف داشته باشد، ممکن است بخواهیم تعیین کنیم که کدام تطابق‌ها مطلوب‌تر هستند. در بخش آینده، به یال‌ها وزن اضافه خواهیم کرد و تطابقی با بیشینه وزن را می‌یابیم.
- اما در این بخش، به جای آن، به هر رأس در یک گراف دوبخشی کامل اطلاعاتی اضافه می‌کنیم: رتبه‌بندی رأس‌های سمت مقابل. به عبارت دیگر، هر رأس در L یک فهرست مرتب از تمام رأس‌های R دارد، و برعکس. برای ساده نگه داشتن بحث، فرض می‌کنیم که هر یک از مجموعه‌های L و R شامل n رأس هستند. هدف در اینجا یافتن یک تطابق بین رأس‌های L و R به‌گونه‌ای «پایدار» است.

مسئله ازدواج پایدار

- این مسئله به نام مسئله ازدواج پایدار^۱ شناخته می‌شود به طوری که L مجموعه‌ای از زنان و R مجموعه‌ای از مردان است. هر زن، تمام مردان را رتبه‌بندی می‌کند و هر مرد نیز تمام زنان را رتبه‌بندی می‌کند.
- هدف، یافتن تطابق به گونه‌ای است که اگر زن و مردی با یکدیگر زوج نشده‌اند، آنگاه حداقل یکی از آن‌ها شریک تخصیص یافته‌اش را ترجیح دهد.
- اگر زنی و مردی با یکدیگر زوج نشده باشند اما هر یک، دیگری را به شریک تخصیص یافته‌اش ترجیح دهد، آن‌ها یک «زوج مسدودکننده»^۲ را تشکیل می‌دهند. یک زوج مسدودکننده انگیزه دارد که از ازدواج تعیین شده خارج شده و با یکدیگر جفت شوند. در این صورت، این زوج پایداری تطابق را «مسدود» می‌کنند.
- یک تطابق پایدار، تطابقی است که هیچ زوج مسدودکننده‌ای ندارد.

¹ stable-marriage problem

² blocking pair

مسئله ازدواج پایدار

- تعداد پاسخ‌ها به یک مسئله ازدواج پایدار الزاماً یکتا نیست. اما آیا همیشه حداقل یک تطابق پایدار وجود دارد؟ پاسخ این پرسش، مثبت است. بنابراین همیشه یک یا چند تطابق پایدار وجود دارد.
- یک الگوریتم ساده به نام الگوریتم گیل-شپلی^۱ همواره یک تطابق پایدار پیدا می‌کند. این الگوریتم دو نسخه «زن محور»^۲ و «مرد محور»^۳ دارد. در اینجا نسخه زن محور را بررسی می‌کنیم.
- هر شرکت‌کننده یا مزدوج است یا آزاد و همه افراد در ابتدا آزاد هستند. وضعیت مزدوج زمانی اتفاق می‌افتد که زنی آزاد به مردی پیشنهاد دهد.

^۱ Gale-Shapley

^۲ woman-oriented

^۳ man-oriented

مسئله ازدواج پایدار

- هنگامی که مردی برای اولین بار پیشنهادی دریافت می‌کند، وضعیتش از آزاد به مزدوچ تغییر می‌یابد و از آن پس همیشه مزدوچ باقی می‌ماند—هرچند نه لزوماً با همان زن.
- اگر مردی مزدوچ پیشنهادی از زنی دریافت کند که او را به زنی که در حال حاضر با او زوج است ترجیح دهد، آن زوج شکسته می‌شود، زنی که قبلاً با او مزدوچ بوده آزاد می‌شود، و مرد با زنی که بیشتر ترجیح داد زوج می‌شود.
- هر زن به ترتیب به مردان موجود در فهرستش پیشنهاد می‌دهد، تا زمانی که مزدوچ شود. زمانی که زنی مزدوچ است، موقتاً از پیشنهاد دادن دست می‌کشد؛ اما اگر دوباره آزاد شود، فرآیند را از همان جایی که متوقف شده بود ادامه می‌دهد.
- زمانی که همه افراد مزدوچ شوند، الگوریتم پایان می‌یابد.

مسئله ازدواج پایدار

- شبه کد زیر این روش را به صورت دقیق‌تر توضیح می‌دهد:

Algorithm GALE-SHAPLEY (men, women, rankings)

```
1: Assign each woman and man as free
2: while some woman  $w$  is free do
3:   Let  $m$  be the first man on  $w$ 's list to whom she hasn't proposed
4:   if  $m$  is free then
5:      $w$  and  $m$  become engaged to each other (and not free)
6:   else if  $m$  ranks  $w$  higher than his currently engaged woman then
7:      $m$  breaks the engagement to  $w'$ , who becomes free
8:      $w$  and  $m$  become engaged to each other (and not free)
9:   else
10:     $m$  rejects  $w$ , with  $w$  remaining free
11: return the stable matching consisting of the engaged pairs
```

مسئله ازدواج پایدار

- در خط ۲ می‌توان هر زن آزادی را انتخاب کرد. خواهیم دید که این روش صرف نظر از ترتیب انتخاب زنان آزاد، همواره یک تطابق پایدار تولید می‌کند.
- می‌خواهیم ثابت کنیم الگوریتم گیل-شپلی همیشه خاتمه می‌یابد و یک تطابق پایدار می‌یابد.
- اثبات: ابتدا نشان می‌دهیم که حلقه‌ی while همواره خاتمه می‌یابد. اثبات با روش تناقض انجام می‌شود. اگر این حلقه خاتمه نیابد، دلیل آن این است که برخی از زنان آزاد باقی می‌مانند. برای آنکه زنی آزاد باقی بماند، باید به تمام مردان پیشنهاد داده باشد و از سوی همه آن‌ها رد شده باشد.
- برای آنکه مردی زنی را رد کند، باید پیش‌تر زوج شده باشد. بنابراین، تمام مردان مزدوج هستند. از آنجا که تعداد زنان و مردان برابر است، نتیجه می‌گیریم که همه‌ی زنان نیز زوج هستند. این به تناقض می‌انجامد، چرا که طبق فرض اولیه، برخی زنان آزاد بودند.

مسئله ازدواج پایدار

- همچنین باید نشان دهیم این حلقه تعداد بار محدودی اجرا می‌شود. از آن‌جا که هر یک از n زن، فهرست ترجیحات خود از میان n مرد را به ترتیب بررسی می‌کند (ممکن است تا انتهای فهرست نرسد) تعداد تکرار حلقه حداقل برابر است با n^2 .
- در نتیجه، الگوریتم گیل-شپلی را می‌توان به گونه‌ای پیاده‌سازی کرد که در زمان $O(n^2)$ اجرا شود.
- اکنون باید نشان دهیم که هیچ زوج مسدودکننده‌ای وجود ندارد.
- فرض کنید در تطابق نهایی زن w با مرد m زوج شده است، اما او مردی دیگر به نام m' را ترجیح می‌دهد. می‌خواهیم ثابت کنیم زوج (w, m') مسدودکننده نیست، پس باید نشان دهیم m' فرد w را به شریک فعلی خود ترجیح نمی‌دهد.

مسئله ازدواج پایدار

- ابتدا دقت کنید وقتی مردی مانند m مزدوچ شد، مزدوچ باقی می‌ماند و هر بار که زوج خود را می‌شکند، به خاطر زنی است که او را ترجیح می‌دهد.
- از آنجا که w شخص' m' را به m ترجیح می‌دهد، پس او باید پیش از پیشنهاد به m ، به' m' پیشنهاد داده باشد؛ و' m' یا این پیشنهاد را رد کرده، یا ابتدا پذیرفته و سپس نامزدی را بر هم زده است.
- اگر' m' پیشنهاد w را رد کرده، پس در آن لحظه با زنی زوج بوده که او را به w ترجیح می‌داده است. اگر' ابتدا پذیرفته و سپس نامزدی را بر هم زده، پس در بردهای با w زوج بوده اما بعدتر پیشنهاد زنی را پذیرفته که او را به w ترجیح می‌داده است.
- بنابراین، تطابق نهایی هیچ زوج مسدودکننده‌ای ندارد.

مسئله ازدواج پایدار

- از آنجا که خط ۲ می‌تواند هر زن آزاد را انتخاب کند، ممکن است این سؤال پیش آید که آیا انتخاب‌های متفاوت می‌توانند تطابق‌های پایدار متفاوتی تولید کنند؟ پاسخ منفی است.
- در مورد الگوریتم -شپلی قضیه دیگری وجود دارد که بیان می‌کند: صرف نظر از ترتیب انتخاب زنان در خط ۲ این الگوریتم همواره یک تطابق یکسان بازمی‌گرداند.
- همچنین در الگوریتم گیل-شپلی زن محور هر زن با بهترین شریک ممکن خود در میان تمامی تطابق‌های پایدار زوج می‌شود. و هر مرد با بدترین شریک ممکن خود در میان تمامی تطابق‌های پایدار زوج می‌شود.

- بیایید باز دیگر اطلاعاتی را به یک گراف دو بخشی کامل اضافه کنیم است. این بار به هر یال یک وزن اختصاص می‌دهیم. دوباره فرض می‌کنیم مجموعه‌های L و R هر یک شامل n رأس هستند، بنابراین گراف دارای n^2 یال خواهد بود.
- برای $l \in L$ و $r \in R$ ، وزن یال (l, r) را با $w(l, r)$ نمایش می‌دهیم که نشان‌دهنده‌ی میزان سودی است که از تطبیق رأس l با رأس r به دست می‌آید.
- یک تطابق کامل^۱ تطابقی است که تحت آن همه رأس‌ها تطابق یافته باشند.

^۱ perfect matching

- در این مسئله هدف یافتن یک تطابق کامل مانند M^* است به طوری که مجموع وزن یال‌های آن، در میان همه تطابق‌های کامل، بیشینه باشد. به این مسئله، مسئله تخصیص^۱ گفته می‌شود.
- بررسی تمام تطابق‌های کامل از مرتبه $\Omega(n!)$ است. اما الگوریتمی به نام الگوریتم مجارستانی^۲ این مسئله را بسیار سریع‌تر حل می‌کند.
- این الگوریتم ا زمرتبه زمانی $O(n^4)$ است. (البته می‌توان زمان اجرای آن را به $O(n^3)$ کاهش داد.)

^۱ assignment problem

^۲ Hungarian algorithm

- الگوریتم مجارستانی به جای کار با گراف دوبخشی کامل G ، با زیرگرافی از آن به نام زیرگراف تساوی^۱ کار می‌کند. این زیرگراف به صورت پویا تغییر می‌کند و هر تطابق کامل در زیرگراف تساوی، یک راه حل بهینه برای مسئله تخصیص نیز هست.
- زیرگراف تساوی با تخصیص یک عدد به هر رأس ساخته می‌شود. به این عدد برجسب رأس گفته می‌شود و با h نشان داده می‌شود.
- می‌گوییم h یک برجسب‌گذاری مجاز^۲ است اگر برای همه $l \in L$ و $r \in R$ رابطه زیر برقرار باشد:

$$l.h + r.h \geq w(l, r)$$

^۱ equality subgraph

^۲ feasible labeling

- یک برچسب‌گذاری مجاز همواره وجود دارد. به عنوان مثال، می‌توان برچسب‌گذاری پیش‌فرض زیر را در نظر گرفت:

$$\begin{aligned} l.h &= \max\{w(l, r) \mid r \in R\} & \forall l \in L \\ r.h &= 0 & \forall r \in R \end{aligned}$$

- فرض کنید h یک برچسب‌گذاری مجاز باشد، زیرگراف تساوی $G_h = (V, E_h)$ دارای همان رأس‌های G است ولی تنها شامل یال‌هایی است که در آن‌ها تساوی زیر برقرار باشد:

$$E_h = \{(l, r) \in E : l.h + r.h = w(l, r)\}$$

- قضیه‌ی زیر، رابطه‌ی بین تطابق کامل در زیرگراف تساوی و راه حل بهینه برای مسئله تخصیص را بیان می‌کند:
- فرض کنید h یک برچسب‌گذاری مجاز برای G و G_h زیرگراف تساوی متناظر باشد. اگر M^* یک تطابق کامل روی G_h باشد، آنگاه یک راه حل بهینه برای مسئله تخصیص روی G نیز هست.
- اثبات: سودمندی M^* برابر است با:

$$w(M^*) = \sum_{(l,r) \in M^*} w(l,r)$$

- گفتیم در زیرگراف تساوی وزن یال برابر است با جمع برچسب‌های دو رأس آن پس:

$$w(M^*) = \sum_{(l,r) \in M^*} w(l.h + r.h)$$

- از آنجا که G_h و G مجموعه رأس‌های یکسانی دارند M^* یک تطابق کامل در G نیز هست. در هر تطابق کامل، هر رأس دقیقاً در یک یال قرار دارد، بنابراین:

$$w(M^*) = \sum_{l \in L} l.h + \sum_{r \in R} r.h$$

الگوریتم مجارستانی

- حال باید ثابت کنیم سودمندی M^* از هر تطابق کامل دلخواهی که روی گراف اصلی در نظر بگیریم بیشتر است. اگر M را یک تطابق کامل دلخواه در نظر بگیریم سودمندی آن برابر است با:

$$w(M) = \sum_{(l,r) \in M} w(l,r)$$

- در یک برچسب گذاری مجاز مجموع دو رأس یک یا لازم است بنا بر این:

$$w(M) = \sum_{(l,r) \in M} w(l,r) \leq \sum_{(l,r) \in M} (l.h + r.h) = \sum_{l \in L} l.h + \sum_{r \in R} r.h$$

- دیدم که سودمندی M^* برابر با

$$\sum_{l \in L} l.h + \sum_{r \in R} r.h$$

و سودمندی M کمتر از این مقدار است پس سودمندی M^* از هر تطابق کامل دلخواهی در G بیشتر است.

- پس حالا هدف یافتن یک تطابق کامل در یک زیرگراف تساوی است. اما کدام زیرگراف تساوی؟ مهم نیست! نه تنها می‌توانیم زیرگراف برابری را انتخاب کنیم، بلکه در طول اجرای الگوریتم می‌توانیم آن را تغییر دهیم. تنها چیزی که نیاز داریم، یافتن یک تطابق کامل در یک زیرگراف برابری است.

- بار دیگر یه این نامساوی که در اثبات قبل دیدیم دقت کنید:

$$w(M) \leq \sum_{l \in L} l.h + \sum_{r \in R} r.h$$

- فرض کردیم M یک تطابق کامل است. اما نامساوی با فرض اینکه M یک تطابق دلخواه باشد همچنان برقرار است.
- یعنی مجموع برچسب رأس‌ها یک حد بالا برای وزن هر تطابق است.

- حال که یک حد بالا برای مجموع وزن تطابق‌ها یافتیم، می‌خواهیم به صورت تکراری به این حد بالا نزدیک شویم.
- فرض کنید h یک برچسب‌گذاری دلخواه باشد آنگاه تطابق با اندازه بیشینه^۱ در این زیرگراف برابری، مجموع وزنی حد اکثر برابر با مجموع برچسب‌ها خواهد داشت.
- حال اگر برچسب‌گذاری h به درستی انتخاب شود، جمع برچسب‌ها برابر با $(M^*)w$ خواهد بود، و یک تطابق با اندازه بیشینه در آن زیرگراف برابری یک تطابق کامل با وزن بیشینه خواهد بود.
- همانطور که ملاحظه کردید هم تطابق را باید به سمت تطابق با اندازه بیشینه تغییر دهیم و هم برچسب‌گذاری را تغییر دهیم تا به یک برچسب‌گذاری مناسب دست یابیم. الگوریتم مجارستانی تطابق و برچسب‌گذاری رأس‌ها را به صورت مکرر تغییر می‌دهد تا به این هدف برسد.

^۱ maximum-cardinality matching

- این الگوریتم با یک برچسب‌گذاری مجاز اولیه و یک تطابق دلخواه در زیرگراف تساوی شروع می‌کند. سپس، به صورت تکراری، یک مسیر افزایشی نسبت به M در G_h پیدا می‌کند و تطابق را به $M \oplus P$ بهروزرسانی می‌کند. به این ترتیب اندازه تطابق افزایش می‌یابد.
- تا زمانی که زیرگرافی از تساوی وجود داشته باشد که شامل یک مسیر افزایشی نسبت به M باشد، می‌توان اندازه تطابق را افزایش داد. این روند ادامه می‌یابد تا یک تطابق کامل به دست آید.

الگوریتم مجارستانی

- ممکن است در این لحظه سوالات زیر برای شما پیش بیاید بباید به هر کدام کوتاه پاسخ دهیم:
 - الگوریتم با چه برچسب‌گذاری مجاز اولیه‌ای شروع می‌کند؟
برچسب‌گذاری پیش‌فرض رأس‌ها که پیشتر تعریف شد و برچسب هر رأس سمت چپ را برابر با بیشترین وزن موجود در بین یال‌ها متصل به آن قرار می‌داد.
 - الگوریتم با چه تطابق اولیه‌ای شروع می‌کند؟
هر تطابقی، حتی تطابق تهی. اما یک تطابق حداقلی حاصل از یک الگوریتم حریصانه در اینجا به خوبی عمل می‌کند.

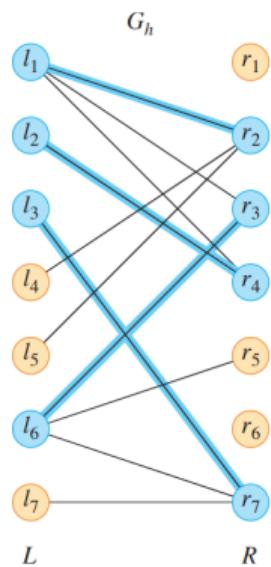
- اگر مسیر افزایشی نسبت به M در G_h وجود داشته باشد، چگونه می‌توان آن را یافت؟
با استفاده از نوعی جستجوی اول سطح؛ مشابه با آنچه در الگوریتم هاپکرافت-کارپ استفاده کردیم.
- اگر جستجو برای مسیر افزایشی شکست بخورد چه باید کرد؟
برچسب‌گذاری رأس‌ها باید بهروزرسانی شود تا حداقل یک یال جدید وارد زیرگراف تساوی شود.

الگوريتم مجازستانی

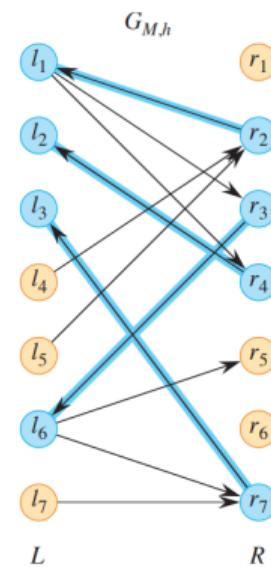
- در ادامه پاسخ سوالات بالا را با استفاده از مثال زیر نشان می‌دهیم:

	r_1	r_2	r_3	r_4	r_5	r_6	r_7	
h	0	0	0	0	0	0	0	
l_1	10	4	10	10	10	2	9	3
l_2	12	6	8	5	12	9	7	2
l_3	15	11	9	6	7	9	5	15
l_4	9	3	9	6	7	5	6	3
l_5	6	2	6	5	3	2	4	2
l_6	11	10	8	11	4	11	2	11
l_7	8	3	4	5	4	3	6	8

(a)



(b)



(c)

- وزن یال‌ها در ماتریسی که در بخش a نشان داده شده و برچسب‌ها در سمت چپ و بالای ماتریس مشخص شده‌اند.
- شکل b گراف تساوی است و شکل c گرافی است جهت‌دار که همه رأس‌های موجود در M را راست به چپ و بقیه رأس‌ها را چپ به راست جهت دهی می‌کند.
- مدخل‌های قرمز رنگ برای یال‌هایی اند که وزن آنها با جمع برچسب گذاری دو سر آن برابر است. یعنی این یال‌ها در زیرگراف تساوی در بخش b وجود دارند.

الگوریتم مجارستانی

تطابق اولیه حداکثری

- روش‌های مختلفی برای پیاده‌سازی یک الگوریتم حریصانه برای یافتن یک تطابق حداکثری دو بخشی وجود دارد. الگوریتم زیر یکی از این روش‌ها است.

Algorithm GREEDY-BIPARTITE-MATCHING(G)

```
1:  $M = \emptyset$ 
2: for each vertex  $l \in L$  do
3:   if  $l$  has an unmatched neighbor in  $R$  then
4:     choose any such unmatched neighbor  $r \in R$ 
5:      $M = M \cup (l, r)$ 
6: return  $M$ 
```

- این الگوریتم تطابقی را بازمی‌گرداند که اندازه‌ی آن حداقل نیمی از اندازه‌ی تطابق بیشینه است.

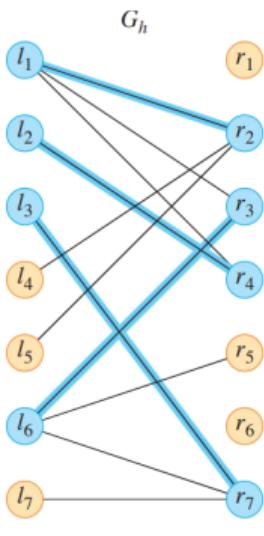
الگوریتم مجارستانی

تطابق اولیهٔ حداکثری

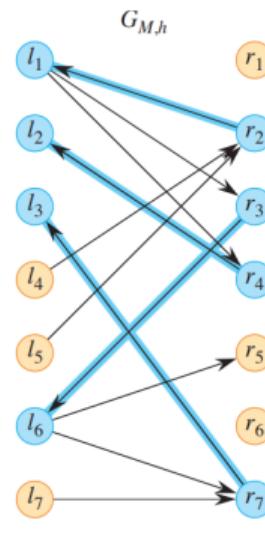
- در قسمت b این شکل، یال‌هایی که به رنگ آبی برجسته شده‌اند، تطابق اولیهٔ حریصانه در G_h را نشان می‌دهند.

	r_1	r_2	r_3	r_4	r_5	r_6	r_7
h	0	0	0	0	0	0	0
l_1	10	4	10	10	10	2	9
l_2	12	6	8	5	12	9	7
l_3	15	11	9	6	7	9	5
l_4	9	3	9	6	7	5	6
l_5	6	2	6	5	3	2	4
l_6	11	10	8	11	4	11	2
l_7	8	3	4	5	4	3	6

(a)



(b)



(c)

تطابق در گراف دو بخشی

الگوریتم مجارستانی

یافتن مسیر افزایشی

- برای یافتن یک مسیر افزایشی در زیرگراف تساوی، الگوریتم مجارستانی ابتدا زیرگراف تساوی جهت دار G_h را از G می سازد؛ دقیقاً همانگونه که الگوریتم هاپکرفت-کارپ، گراف G_M را از G می سازد.
- مانند الگوریتم هاپکرفت-کارپ، می توانید مسیر افزایشی را به گونه ای تصور کنید که از یک رأس تطابق نیافته در L آغاز می شود و به یک رأس تطابق نیافته در R ختم می شود؛ پس در این مسیر یال های تطابق نیافته از L به R و یال هایی تطابق یافته از R به L طی می شوند.
- پس یال های گراف $G_{M,h}$ را نیز به همین روال جهت دار می کنیم؛ یال های عضو تطابق به چپ و بقیه یال ها به راست.

الگوریتم مجارستانی

یافتن مسیر افزایشی

- شاید فکر کنید که ما با اینکار برخی مسیرها افزایشی را از دست می‌دهیم. اما در واقع با اینکار از مسیرهای افزایشی تکراری جلوگیری می‌کنیم.
- تصور کنید دو مسیر دقیقاً با یال‌های یکسان اما جهت برعکس داشته باشیم. در نهایت زمانی که می‌خواهیم این مسیرها را با تفاضل مقارن اعمال کنیم جهت آنها تاثیری ندارد پس عملاً این دو مسیر یکسان هستند. پس یکی از جهت‌ها را قرارداد کرده و تنها مسیرهای افزایشی که در آن جهت هستند را در نظر می‌گیریم.
- از آنجا که هر مسیر افزایشی نسبت به M در $G_{M,h}$ ، یک مسیر افزایشی در h نیز محسوب می‌شود، کافی است مسیرهای افزایشی را در $G_{M,h}$ بیابیم.

الگوریتم مجارستانی

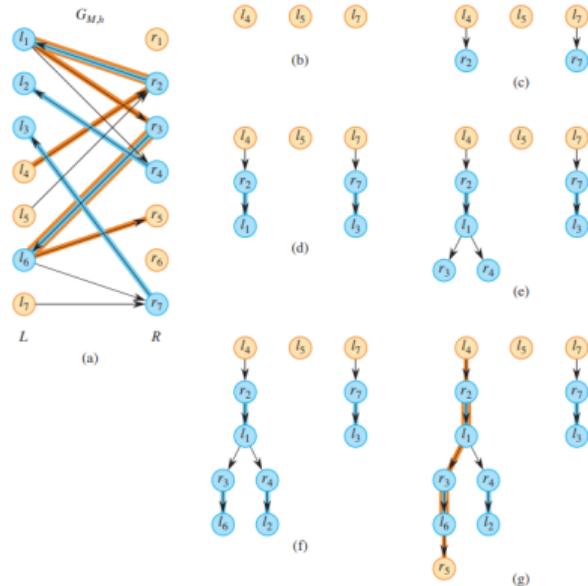
یافتن مسیر افزایشی

- با در اختیار داشتن $G_{M,h}$ الگوریتم مجارستانی به دنبال یک مسیر افزایشی نسبت به M از هر رأس تطابق نیافته در L به هر رأس تطابق نیافته در R می‌گردد.
- در اینجا از جستجوی اول سطح برای این منظور استفاده می‌کنیم، که از تمام رأس‌های تطابق نیافته در L شروع می‌شود. درست مانند الگوریتم هاپکرفت-کارپ با این تفاوت که بعد از یافتن اولین رأس تطابق نیافته در R متوقف می‌شود.

الگوریتم مجارستانی

یافتن مسیر افزایشی

- شکل زیر نحوه عملکرد این نوع جستجوی اول سطح را نشان می‌دهد.



الگوریتم مجارستانی

یافتن مسیر افزایشی

- در مثال بالا، جستجوی اول سطح مسیر زیر را یافته است:

$$\langle (l_4, r_2), (r_2, l_1), (l_1, r_3), (r_3, l_6), (l_6, r_5) \rangle$$

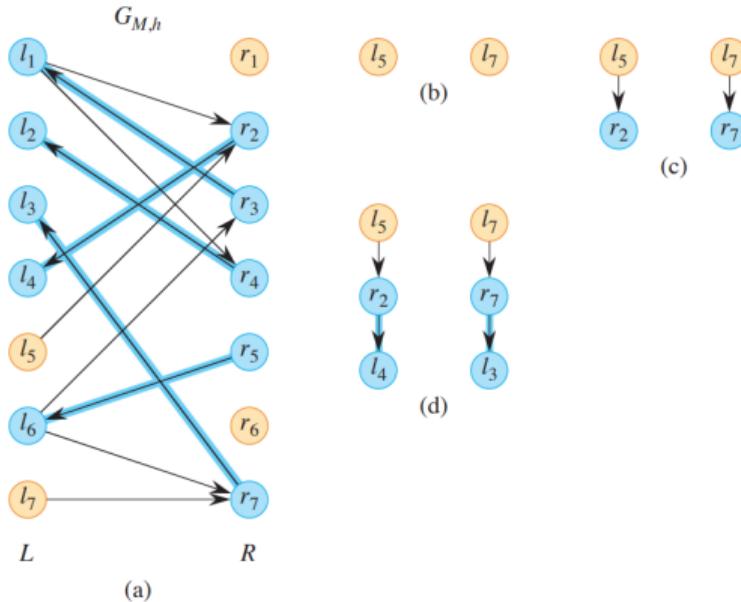
- در این جستجو ابتدا تمام رأس‌های تطابق در L را در صفت قرار می‌دهیم، نه فقط یک رأس مبدأ.
- برخلاف گراف H در الگوریتم هاپکرفت-کارپ، در اینجا هر رأس فقط به یک رأس قبلی نیاز دارد، بنابراین جستجوی اول سطح، یک جنگل اول سطح^۱ به نام $F = (V_F, E_F)$ ایجاد می‌کند، به طوری که هر رأس تطابق نیافته در L یک ریشه در F است.

^۱ breadth-first forest

الگوریتم مجارستانی

یافتن مسیر افزایشی

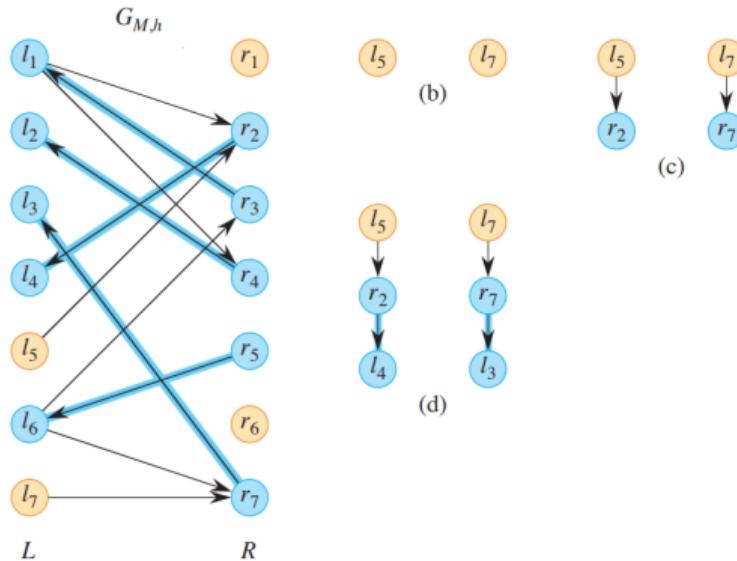
- در قسمت a از شکل زیر تطابق جدید را نشان داده شده که از طریق گرفتن تفاضل مقارن بین مسیر افزایشی به دست آمده و تطابق است.



الگوریتم مجارستانی

وقتی مسیر افزایشی یافت نشود

- پس از بهروزرسانی تطابق الگوریتم مجارستانی زیرگراف تساوی را با توجه به تطابق جدید بهروزرسانی می‌کند و سپس جستجوی اول سطح جدیدی را از تمام رأس‌های تطابق نیافته در L آغاز می‌نماید. باید با استفاده از همان مثال قبلی آغاز این فرایند را ببینیم:



تطابق در گراف دو بخشی

الگوریتم مجارستانی

وقتی مسیر افزایشی یافت نشود

- در قسمت d شکل بالا صفت شامل رأس‌های l_4 و l_3 است. با این حال، هیچ‌یک از این رأس‌ها یالی ندارند که از آن‌ها خارج شود، بنابراین زمانی که این رأس‌ها از صفت خارج می‌شوند، صفت خالی می‌گردد. در این نقطه، جست‌وجو متوقف می‌شود، بدون آن‌که رأس تطابق نیافته‌ایی در R یافته شده باشد تا یک مسیر افزایشی پیدا شود.
- هرگاه چنین وضعیتی رخ دهد، آخرین رأس‌های کشف شده^۱ الزاماً متعلق به L هستند. زیرا هرگاه که یک رأس در R شود؛
 - اگر تطابق نیافته باشد یک مسیر افزایشی پیدا شده است،
 - و اگر تطابق‌یافته باشد، همسایه کشف‌نشده‌ای در L دارد پس جست‌وجو ادامه می‌یابد.
- منظور از کشف اولین باری است که در جست‌وجو با یک رأس مواجه می‌شویم و آن را به صفت اضافه می‌کنیم.

¹ discovered

الگوریتم مجارستانی

وقتی مسیر افزایشی یافت نشود

- به یاد داشته باشید که می‌توانیم با هر زیرگراف تساوی دلخواه کار کنیم. بنابراین می‌توانیم آن را تغییر دهیم.
البته، به شرطی که تلاش‌های پیشین خراب نشوند. برای این منظور باید شرایط زیر برقرار باشند:
 - ۱ هیچ یالی از جنگل اول سطح از زیرگراف تساوی جهت‌دار خارج نشود.
 - ۲ هیچ یالی از تطابق از زیرگراف تساوی جهت‌دار خارج نشود.
- ۳ دست‌کم یک یال (l, r) ، که در آن E_h شود و بنابراین وارد $E_{M,h}$ نیز بشود. در نتیجه، دست‌کم یک رأس کشف نشده در R اضافه می‌شود.

الگوریتم مجارستانی

وقتی مسیر افزایشی یافت نشود

- بنابراین، حداقل یک یال جدید وارد زیرگراف تساوی می‌شود و هر یال می‌شود. و هر یالی که از زیرگراف تساوی حذف می‌شود، به M و F تعلق ندارد.
- برای بهروزرسانی برچسب‌گذاری، الگوریتم مجارستانی ابتدا مقدار زیر را محاسبه می‌کند:

$$\delta = \min\{l.h + r.h - w(l, r) : l \in F_L \text{ and } r \in R - F_R\}$$

الگوریتم مجارستانی

وقتی مسیر افزایشی یافت نشود

- که در آن $F_R = R \cap V_F$ و $F_L = L \cap V_F$ به ترتیب رئوس متعلق به مجموعه L و R در جنگل F هستند.
- به عبارت دیگر، دلتا نشان می‌دهد از بین یال‌هایی که امکان اضافه شدن را دارند کدام یک به تفاوت کمتری بین یال و برچسب دو طرف دارد. در واقع این یال کمترین فاصله را برای اضافه شدن به تطابق دارد.

الگوریتم مجارستانی

وقتی مسیر افزایشی یافت نشود

- سپس، الگوریتم مجارستانی یک برچسب‌گذاری مجاز جدید، مثلاً h' ، ایجاد می‌کند، که در آن به ازای هر رأس $v \in F_L$ مقدار دلتا از $l.h$ کم می‌شود، و به ازای هر رأس $v \in F_R$ مقدار دلتا به $r.h$ افزوده می‌شود. این برچسب‌گذاری جدید به صورت زیر تعریف می‌شود:

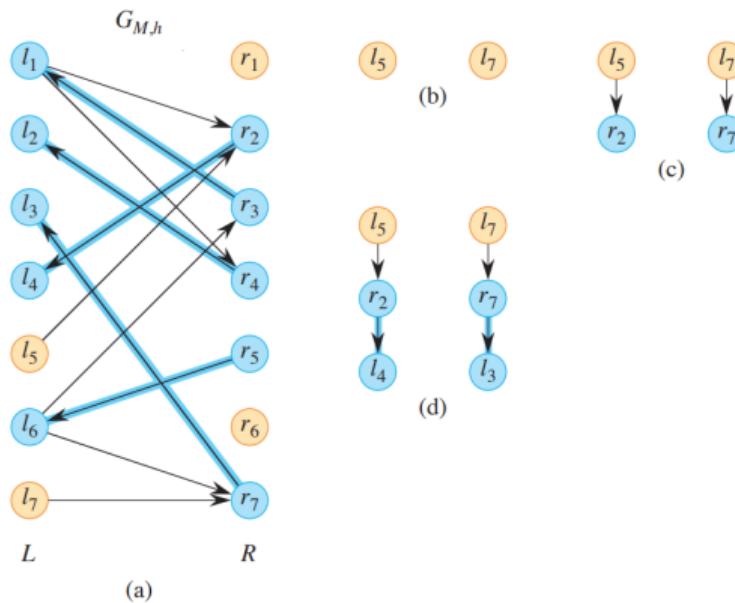
$$v.h' = \begin{cases} v.h - \delta & \text{if } v \in F_L \\ v.h + \delta & \text{if } v \in F_R \\ v.h & \text{otherwise } (v \in V - V_F) \end{cases}$$

- نشان داده می‌شود که این تغییرات به سه معیاری که پیشتر ذکر شد را رعایت می‌کنند.

الگوریتم مجارستانی

مثال اجرای الگوریتم مجارستانی

- بیایید مثال قبلی خودمان را ادامه دهیم؛ تا آنجا رسیدیم که صف پیش از یافتن یک مسیر افزایشی خالی شد.



تطابق در گراف دو بخشی

۴۸۶ / ۲۶۲

الگوریتم‌های پیشرفته

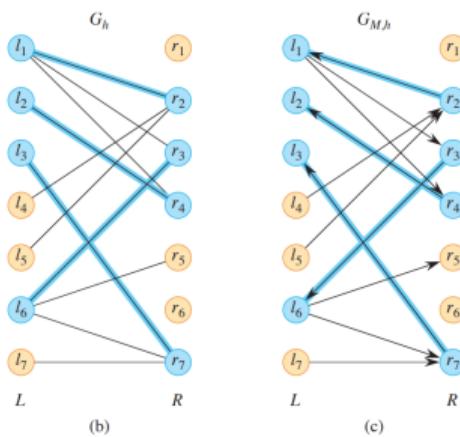
الگوریتم مجارستانی

مثال اجرای الگوریتم مجارستانی

- مقدار $1 = \delta = 1$ توسط یال (l_5, r_3) حاصل می‌شود. به شکل زیر دقت کنید:

	r_1	r_2	r_3	r_4	r_5	r_6	r_7
h	0	0	0	0	0	0	0
l_1	10	4	10	10	2	9	3
l_2	12	6	8	5	12	9	7
l_3	15	11	9	6	7	9	5
l_4	9	3	9	6	7	5	6
l_5	6	2	6	5	3	2	4
l_6	11	10	8	11	4	11	2
l_7	8	3	4	5	4	3	6

(a)



$$l_5.h + r_3.h - w(l_5, r_3) = 6 + 0 - 5 = 1$$

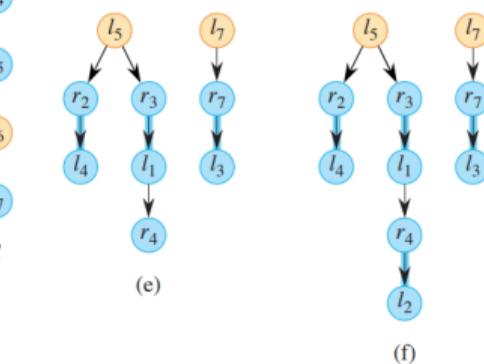
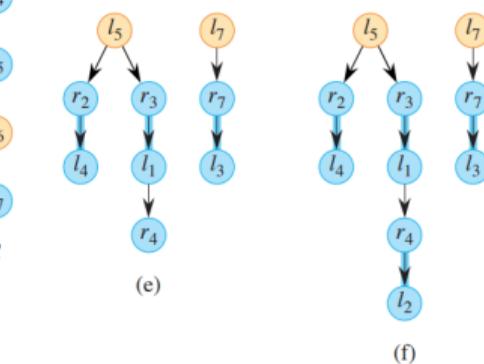
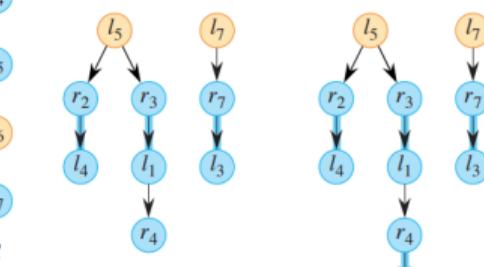
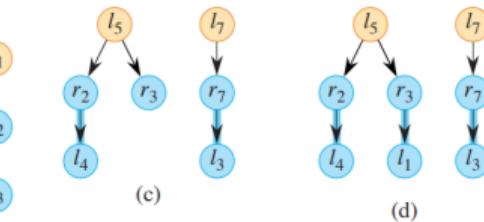
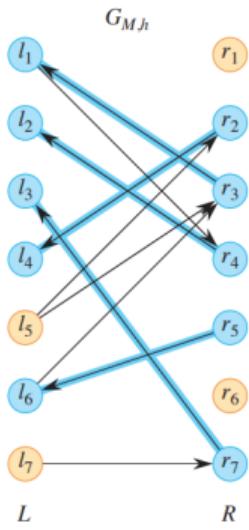
الگوریتم مجارستانی

مثال اجرای الگوریتم مجارستانی

- شکل زیر گام‌های بعدی الگوریتم را نشان می‌دهد:

	r_1	r_2	r_3	r_4	r_5	r_6	r_7
h	0	1	0	0	0	0	1
l_1	10	4	10	10	2	9	3
l_2	12	6	8	5	12	9	7
l_3	14	11	9	6	7	9	5
l_4	8	3	9	6	7	5	6
l_5	5	2	6	5	3	2	4
l_6	11	10	8	11	4	11	2
l_7	7	3	4	5	4	3	6

(a)



تطابق در گراف دو بخشی

الگوریتم مجارستانی

مثال اجرای الگوریتم مجارستانی

- همانطور که مشاهده می‌کنید، مقادیر $l_1.h$, $l_2.h$, $l_3.h$, $l_4.h$, $l_5.h$, و $l_7.h$ واحد کاوش و مقادیر $r_2.h$ و $r_7.h$ واحد افزایش یافته‌اند، زیرا این رأس‌ها در F قرار دارند.
- در نتیجه، یال‌های (l_5, r_3) و (l_6, r_7) از GM, h خارج شده و یال (l_1, r_2) وارد آن شده است.

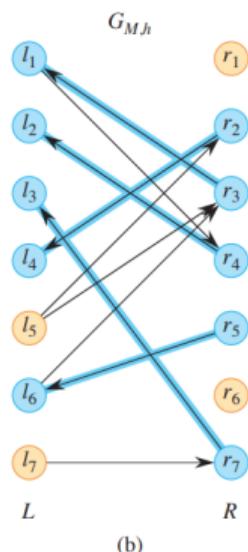
الگوریتم مجارستانی

مثال اجرای الگوریتم مجارستانی

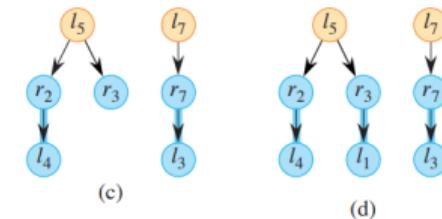
- شکل زیر قسمت b زیرگراف تساوی جدید را نشان می‌دهد. با ورود یال (l_5, r_3) در قسمت c مشاهده می‌کنیم که این یال به F افزوده شده و رأس r_3 به صفت افزوده شده است.

	r_1	r_2	r_3	r_4	r_5	r_6	r_7	
h	0	1	0	0	0	0	1	
l_1	10	4	10	10	10	2	9	3
l_2	12	6	8	5	12	9	7	2
l_3	14	11	9	6	7	9	5	15
l_4	8	3	9	6	7	5	6	3
l_5	5	2	6	5	3	2	4	2
l_6	11	10	8	11	4	11	2	11
l_7	7	3	4	5	4	3	6	8

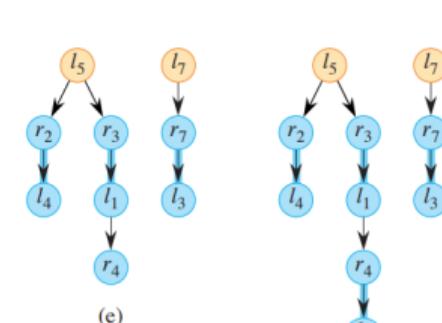
(a)



(b)



(c)



(d)



(e)

تطابق در گراف دو بخشی

الگوریتم مجارستانی

مثال اجرای الگوریتم مجارستانی

- بخش‌های c تا f شکل قبل ادامه ساخت جنگل اول سطح را نشان می‌دهند تا اینکه در بخش f صف پس از حذف رأس l_2 ، بار دیگر خالی می‌شود.
- بنابراین، الگوریتم باید مجدداً برچسب‌گذاری‌ها را بهروزرسانی کند. این‌بار مقدار $1 = \delta$ توسط سه یال حاصل می‌شود:
 $(l_1, r_6), (l_5, r_6), (l_7, r_6)$

الگوریتم مجارستانی

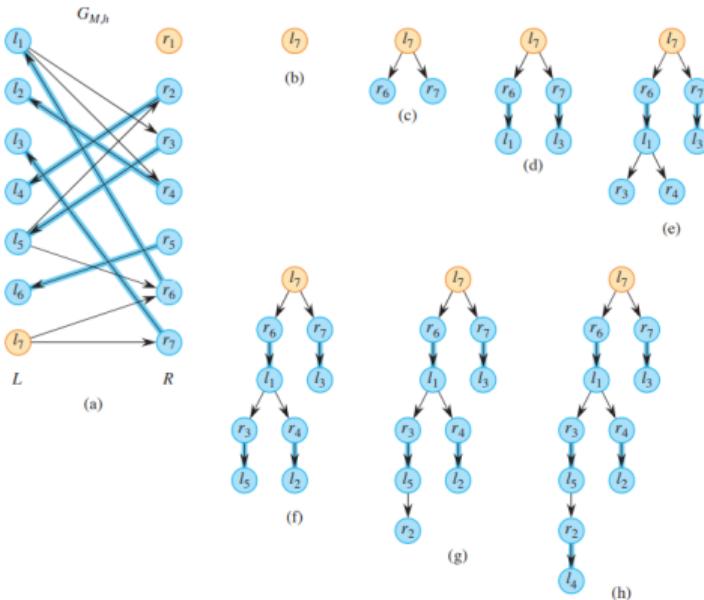
مثال اجرای الگوریتم مجارستانی

- همان‌طور که شکل زیر در قسمت‌های a و b نشان داده شده، این یال‌ها وارد شده‌اند و یال (l_6, r_3) خارج شده است.
بخش c نشان می‌دهد که یال (l_1, r_6) به جنگل افزوده شده است. (یال‌های (l_5, r_6) یا (l_7, r_6) نیز می‌توانستند به جای آن افزوده شوند.)
- از آنجا که r_6 تطابق نیافته است، جست‌وجو مسیر افزایشی زیر را یافته است:
 $\langle(l_5, r_3), (r_3, l_1), (l_1, r_6)\rangle$
این مسیر در شکل بالا با رنگ نارنجی مشخص شده است.

الگوریتم مجارستانی

مثال اجرای الگوریتم مجارستانی

- شکل زیر $G_{M,h}$ را پس از بهروزرسانی تطابق توسط گرفتن تفاصل متقارن آن با مسیر افزایشی نشان می‌دهد.



الگوریتم مجارستانی

مثال اجرای الگوریتم مجارستانی

- الگوریتم مجارستانی جستجوی اول سطح نهایی خود را آغاز می‌کند. (تنها با رأس l_7 به عنوان تنها ریشه)
جستجو طبق بخش‌های b تا h شکل ادامه می‌یابد، تا زمانی که صف پس از حذف رأس l_4 خالی شود.
- این بار، داریم $2 = \delta$ ، که توسط پنج یال زیر حاصل می‌شود:

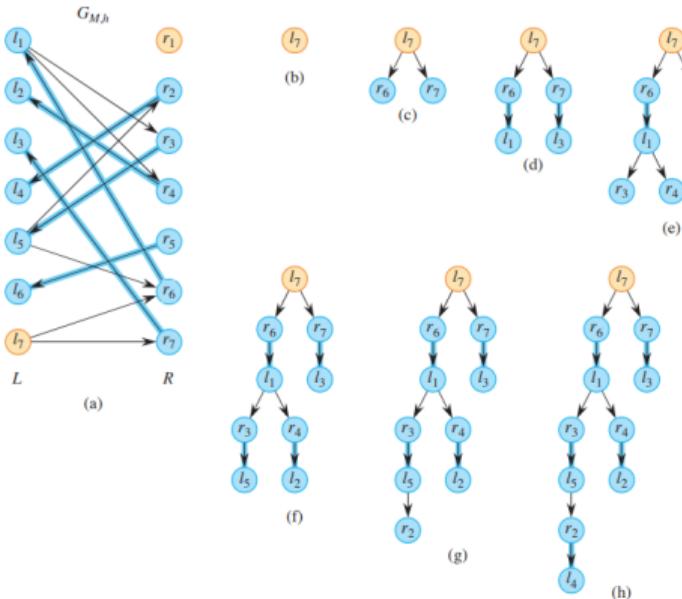
$$(l_2, r_5), (l_3, r_1), (l_4, r_5), (l_5, r_1), (l_5, r_5)$$

هر یک از این یال‌ها وارد $G_{M,h}$ می‌شوند.

الگوریتم مجارستانی

مثال اجرای الگوریتم مجارستانی

- قسمت a از شکل زیر نتیجه کاهش مقدار برچسب رأس‌های F_L به اندازه ۲ و افزایش مقدار برچسب رأس‌های F_R به اندازه ۲ را نشان می‌دهد.



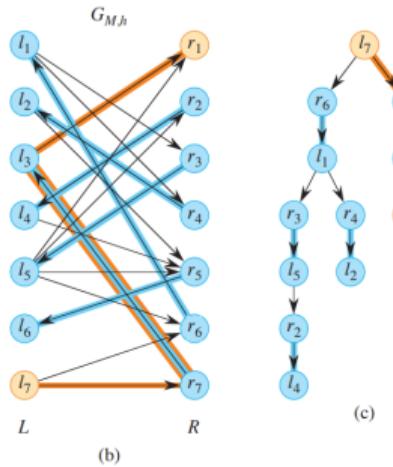
الگوریتم مجارستانی

مثال اجرای الگوریتم مجارستانی

- و قسمت b این شکل زیرگراف برابری جهت دار حاصل را نمایش می دهد. همچنین قسمت c نشان می دهد که یال (l_3, r_1) به جنگل اول سطح افزوده شده است.

	r_1	r_2	r_3	r_4	r_5	r_6	r_7
h	0	4	3	3	0	2	4
l_1	7	4	10	10	2	9	3
l_2	9	6	8	5	12	9	7
l_3	11	11	9	6	7	9	5
l_4	5	3	9	6	7	5	6
l_5	2	2	6	5	3	2	4
l_6	11	10	8	11	4	11	2
l_7	4	3	4	5	4	3	6

(a)



(b)

(c)

الگوریتم مجارستانی

مثال اجرای الگوریتم مجارستانی

- از آنجا که r_1 یک رأس تطابق نیافته است، جستجو خاتمه یافته و مسیر افزایشی

$$\langle (l_7, r_7), (r_7, l_3), (l_3, r_1) \rangle$$

که با رنگ نارنجی برجسته شده، یافته می‌شود. اگر r_1 تطابق یافته بود، رأس r_5 نیز به جنگل اول سطح اضافه می‌شد، که هر یک از l_2 , l_4 , یا l_5 والد آن بودند.

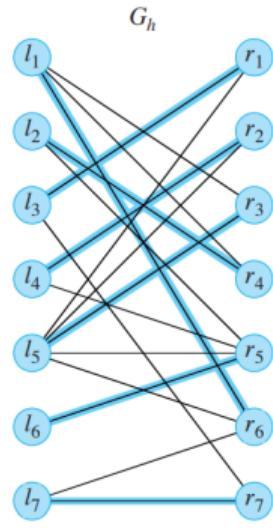
الگوریتم مجارستانی

مثال اجرای الگوریتم مجارستانی

- پس از بهروزرسانی M , الگوریتم به یک تطابق کامل برای زیرگراف تساوی می‌رسد. می‌دانیم این تطابق کامل برای زیرگراف برابری، یک جواب بهینه برای مسئله تخصیص اصلی است.

	r_1	r_2	r_3	r_4	r_5	r_6	r_7
h	0	4	3	3	0	2	4
l_1	7	4	10	10	2	9	3
l_2	9	6	8	5	12	9	7
l_3	11	11	9	6	7	9	5
l_4	5	3	9	6	7	5	6
l_5	2	2	6	5	3	2	4
l_6	11	10	8	11	4	11	2
l_7	4	3	4	5	4	3	6

(a)



(b)

الگوریتم‌های تقریبی

- بسیاری از مسائل محاسباتی کاربردی انپی کامل هستند و با این حال با توجه به اهمیت زیادی که دارند نیاز داریم جوابی برای آنها پیدا کنیم گرچه پیدا کردن جواب دقیق برای اینگونه مسائل در زمان چندجمله‌ای امکان‌پذیر نیست.
- وقتی یک مسئله انپی کامل است، برای حل آن سه راه پیش رو داریم : ۱- اگر ورودی نسبتاً کوچک باشد، می‌توان یک جواب بهینه در زمان نمایی به سرعت برای آن پیدا کرد. ۲- می‌توان یک حالت خاص از مسئله را در زمان چند جمله‌ای حل کرد. ۳- می‌توان یک جواب نزدیک به جواب بهینه در زمان چند جمله‌ای برای آن پیدا کرد.
- در بسیاری از کاربردها جواب نزدیک به جواب بهینه^۱ نیز کافی است. به چنین الگوریتم‌هایی که جواب نزدیک به بهینه تولید می‌کنند، الگوریتم‌های تقریبی^۲ می‌گوییم. برای بسیاری از مسائل انپی کامل می‌توان یک الگوریتم تقریبی در زمان چندجمله‌ای پیدا کرد.

¹ near-optimal solution

² approximation algorithm

- فرض کنید بر روی مسئله بهینه‌سازی کار می‌کنید که در آن هر یک از جواب‌های بالقوه^۱ دارای یک هزینه است و می‌خواهید یک جواب نزدیک به بهینه پیدا کنید. بسته به نوع مسئله، ممکن است مسئله بیشینه سازی^۲ یا کمینه سازی^۳ باشد. می‌توانید یک جواب بهینه با هزینه حداقل پیدا کنید.

¹ potential solution

² maximization

³ minimization

- می‌گوییم یک الگوریتم دارای «ضرب تقریب»^۱ $\rho(n)$ است اگر به ازای هر ورودی با اندازه n ، هزینه جواب تولید شده توسط الگوریتم نسبت به هزینه جواب بهینه از مقدار $\rho(n)$ کمتر باشد. اگر هزینه جواب تولید شده را با C و هزینه جواب بهینه را با C^* نشان دهیم داریم:

$$\max\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} \leq \rho(n)$$

- اگر یک الگوریتم دارای ضرب تقریب $\rho(n)$ باشد، به آن الگوریتم تقریبی $\rho(n)$ می‌گوییم.

^۱ approximation ratio

- از الگوریتم‌های تقریبی $(n)^{\rho}$ هم برای مسائل کمینه سازی و هم برای مسائل بیشینه سازی استفاده می‌شود.
- در یک مسئله بیشینه سازی، داریم $C^* \leq C < 0$ و بنابراین مقدار C/C^* مقدار بزرگ‌تری است که در آن هزینه جواب بهینه از هزینه جواب تقریبی بزرگ‌تر است.
- در یک مسئله کمینه سازی، داریم $C < C^* \leq 0$ و بنابراین مقدار C/C^* مقدار بزرگ‌تری است که در آن هزینه جواب تقریبی از هزینه جواب بهینه بزرگ‌تر است.
- با فرض اینکه همه هزینه‌ها مقادیر مثبت هستند، ضریب تقریب در یک الگوریتم تقریبی هیچ‌گاه کمتر از ۱ نیست.
- بنابراین یک الگوریتم تقریبی با ضریب ۱ جوابی بهینه تولید می‌کند و هر چه ضریب تقریب الگوریتم تقریبی بیشتر باشد، جواب به دست آمده از جواب بهینه دورتر است.

- برای بسیاری از مسائل، الگوریتم‌های تقریبی چند جمله‌ای با ضریب تقریب کوچک وجود دارند و برای برخی دیگر از مسائل، الگوریتم‌های تقریبی دارای ضریب تقریبی هستند که با افزایش اندازه ورودی افزایش پیدا می‌کنند.
- در برخی از الگوریتم‌های تقریبی چندجمله‌ای، هرچه الگوریتم در زمان بیشتری اجرا شود، ضریب تقریب بهتری به دست می‌آید. در چنین مسائلی می‌توان با افزایش زمان محاسبات ضریب تقریب را بهبود داد.
- این وضعیت حائز اهمیت است و یک ناگذاری برای آن وجود دارد که در ادامه معرفی می‌کنیم.

- طرح تقریب^۱ برای یک مسئله بهینه‌سازی، الگوریتمی تقریبی است که یک نمونه از مسئله را همراه با مقدار $\epsilon > 0$ دریافت می‌کند، به طوری که این الگوریتم یک الگوریتم تقریبی $(\epsilon + 1)$ باشد.
- اگر این طرح تقریب برای هر مقدار $\epsilon > 0$ در زمانی چندجمله‌ای نسبت به اندازه ورودی n اجرا شود، آن را طرح تقریب چندجمله‌ای^۲ می‌نامیم.
- زمان اجرای یک طرح تقریب چند جمله‌ایی ممکن است با کاهش ϵ به شدت افزایش یابد. برای مثال زمان اجرای آن می‌تواند چنین چیزی باشد:

$$O(n^{2/\epsilon})$$

^۱ Approximation Scheme

^۲ Polynomial-Time Approximation Scheme

- اگر یک طرح تقریب، زمان اجرای چندجمله‌ای نسبت به هر دو پارامتر $\epsilon/1$ و n داشته باشد، آن را «طرح تقریب چندجمله‌ای کامل»^۱ می‌نامیم.

^۱ می‌تواند چنین باشد:

$$O((1/\epsilon)^2 n^3)$$

- در چنین الگوریتمی، هر کاهش با ضریب ثابت در ϵ ، باعث افزایش زمان اجرا به اندازه یک ضریب ثابت می‌شود.

^۱ Fully Polynomial-Time Approximation Scheme

مسئله پوشش رأسی

- مسئله پوشش رأسی^۱ یک مسئله انپی کامل است.
- پوشش رأسی یک گراف بدون جهت $G = (V, E)$ زیر مجموعه‌ای از رئوس گراف $V \subseteq V'$ است به طوری که اگر (u, v) یک یال از گراف G باشد، آنگاه $u \in V'$ یا $v \in V'$ یا هردو. اندازه پوشش رأسی تعداد رأس‌های V' است.
- در مسئله پوشش رأسی می‌خواهیم کمترین تعداد رئوس در یک گراف را پیدا کنیم که یک پوشش رأسی تشکیل می‌دهند یا به عبارت دیگر همهٔ یال‌ها را پوشش می‌دهند. به چنین پوشش رأسی یک پوشش رأسی بهینه^۲ می‌گوییم.
- اگرچه هیچ الگوریتم چند جمله‌ای برای مسئله پوشش رأسی یافته نشده است، اما یک الگوریتم تقریبی چند جمله‌ای برای پیدا کردن جواب نزدیک به بهینه وجود دارد.

¹ vertex cover problem

² optimal vertex cover

مسئله پوشش رأسی

- الگوریتم تقریبی زیر یک گراف بدون جهت را دریافت می‌کند و یک پوشش رأسی باز می‌گرداند که اندازه آن کمتر از دو برابر پوشش رأسی بهینه است.

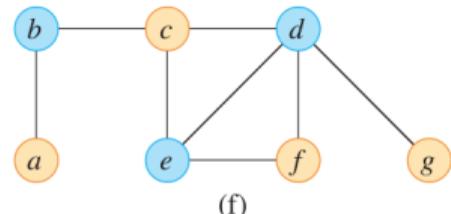
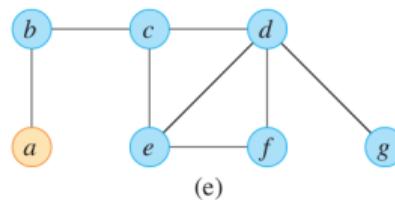
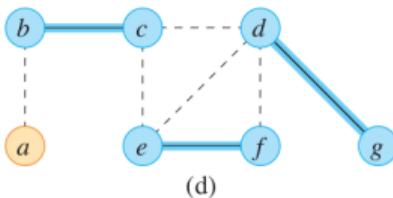
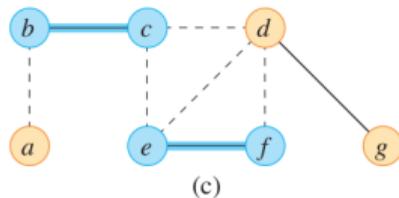
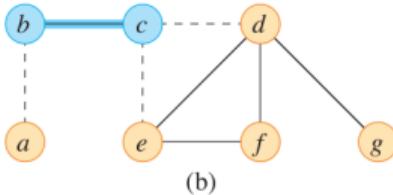
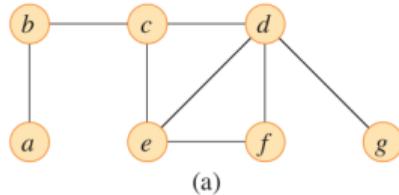
Algorithm Approx-Vertex-Cover

```
function APPROX-VERTEX-COVER(G)
1: C =  $\emptyset$ 
2: E' = G.E
3: while E'  $\neq \emptyset$  do
4:   let (u,v) be an arbitrary edge of E'
5:   C = C  $\cup$  {(u,v)}
6:   remove from E' edge (u,v) and every edge incident on either u or v
7: return C
```

- متغیر C پوشش رأسی در حال ساخت را نگه می‌دارد. این الگوریتم با لیست مجاورت در زمان $O(|V| + |E|)$ اجرا می‌شود.

مسئله پوشش رأسی

- شکل زیر نشان می‌دهد این الگوریتم چگونه بر روی یک گراف عمل می‌کند. در پایان این الگوریتم تقریبی ۶ رأس را برای پوشش رأسی پیدا می‌کند در صورتی که جواب بهینه برای این نمونه مسئله ۳ است.



مسئله پوشش رأسی

- قضیه: الگوریتم تقریبی پوشش رأسی یک الگوریتم چند جمله‌ای تقریبی با ضریب ۲ است.
- اثبات: مجموعه C یک پوشش رأسی است زیرا الگوریتم در حلقه تا وقتی تکرار می‌شود که همه یال‌های $G.E$ با یکی از رئوس C پوشش داده شده‌اند.
- حال می‌خواهیم ثابت کنیم این الگوریتم یک الگوریتم تقریبی با ضریب ۲ است.
- فرض کنید A مجموعه‌ای از یال‌ها باشد که در خط ۴ الگوریتم انتخاب شده‌اند. برای اینکه یال‌های مجموعه A پوشش داده شوند، هر پوشش رأسی باید حداقل یکی از دو رأس هریال در A را شامل شود. هیچ دو یالی در A رأس مشترک ندارند، زیرا وقتی یک یال در خط ۴ انتخاب شد، همه یال‌هایی که با آن یال رأس مشترک دارند از مجموعه E' در خط ۶ حذف می‌شوند.

مسئله پوشش رأسی

- بنابراین هیچ دویالی در A با یک رأس از C^* پوشش داده نشده‌اند و این بدین معنی است که به ازای هر رأس در C^* ، حداقل یک یال در A وجود دارد و بنابراین داریم :

$$|C^*| \geq |A|$$

- هر اجرای خط ۴ یک یال را انتخاب می‌کند که هیچ‌کدام از دو رأس مجاور آن در C نیستند و بنابراین داریم :

$$|C| = 2|A|$$

- با استفاده از دو رابطه به دست آمده خواهیم داشت :

$$|C| = 2|A| \leq 2|C^*|$$

و قضیه بدین ترتیب اثبات می‌شود.

مسئله فروشنده دوره گرد

- ورودی مسئله فروشنده دوره‌گرد، یک گراف کامل و بدون جهت است که هر یال دارای یک هزینه‌ی صحیح نامنفی $c(u, v)$ می‌باشد. هدف، یافتن یک دور همیلتونی با کمترین هزینه است.
- باید یک نمادگذاری دیگر به این مسئله اضافه کنیم؛ اگر A یک مجموعه از یال‌ها باشد آنگاه $(A)c$ مجموع کل وزن یال‌های این مجموعه است.

مسئله فروشنده دوره گرد

- ما به طور شهودی در بسیاری از موقعیت‌های عملی می‌دانیم کوتاهترین مسیر بین دو نقطه مسیر مستقیم است. در علم ریاضیات به این مسئله نابرابری مثلثی می‌گوییم.
- می‌گوییم تابع هزینه یک گراف مانند نابرابری مثلثی را ارضا می‌کند اگر

$$c(u, w) \leq c(u, v) + c(v, w)$$

- به عنوان مثال، اگر رأس‌های گراف نقاطی در صفحه باشند و هزینه‌ی سفر بین دو رأس همان فاصله‌ی اقلیدسی معمولی باشد، آنگاه نابرابری مثلثی برقرار است. افزون بر این، بسیاری از توابع هزینه‌ی دیگر به جز فاصله‌ی اقلیدسی نیز نابرابری مثلثی را برآورده می‌سازند.

مسئله فروشنده دوره گرد

- مسئله فروشنده دوره گرد حتی در حالتی که تابع هزینه نابرابری مثلثی را ارضا کند نیز انپی-کامل است. بنابراین نباید انتظار داشته باشید الگوریتمی چندجمله‌ای برای حل دقیق این مسئله بیابید. بهتر است به دنبال یک الگوریتم‌های تقریبی باشیم.

مسئله فروشنده دوره گرد

حل با نابرابری مثلثی

- وزن یک درخت پوشای کمینه برای مسئله فروشنده دوره‌گرد، یک کران پایین است. بنابراین ابتدا یک درخت پوشای کمینه محاسبه می‌کنیم و این کران پایین را به دست می‌آوریم.
- سپس از همین درخت پوشای کمینه برای ساخت یک دور همیلتونی استفاده می‌کنیم و هزینه‌ی آن حداقل دو برابر وزن درخت پوشای کمینه خواهد بود. البته به شرطی که تابع هزینه، نابرابری مثلثی را ارضا کند.

مسئله فروشنده دوره گرد

حل با نابرابری مثلثی

- الگوریتم زیر این روش را پیاده‌سازی می‌کند.تابع MST-PRIM درخت پوشای کمینه را محاسبه می‌کند.

Algorithm APPROX-TSP-TOUR

```
function APPROX-TSP-TOUR( $G, c$ )
1: select a vertex  $r \in V(G)$  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 a list of vertices, ordered according to when they are first
   visited in a preorder tree walk of  $T$ 
4: return the Hamiltonian cycle  $H$ 
```

مسئله فروشنده دوره گرد

حل با نابرابری مثلثی

- در شبکه که بالا از پیمایش پیش ترتیب^۱ استفاده شده، پیش‌مایش پیش ترتیب به‌طور بازگشتی هر رأس را درخت پیمایش کرده و رأس را زمانی که برای نخستین بار ملاقات می‌شود (پیش از بازدید از فرزندانش) لیست می‌کند.
- زمان اجرای APPROX-TSP-TOUR برابر $\Theta(V^2)$ است.

^۱ preorder tree

مسئله فروشنده دوره گرد حل با نابرابری مثلثی

- اکنون نشان می‌دهیم که اگر تابع هزینه‌ی یک نمونه از مسئله‌ی فروشنده‌ی دوره‌گرد نابرابری مثلثی را ارضا کند، آنگاه APPROX-TSP-TOUR دور همیلتونی‌ایی برمی‌گرداند که هزینه‌ی آن حداقل دو برابر هزینه‌ی یک دور همیلتونی بهینه است. یعنی ضریب تقریب آن ۲ است.
- اثبات فرض کنید H^* یک دور همیلتونی بهینه باشد. حذف هر یال از یک دور همیلتونی، یک درخت پوشای ایجاد می‌کند و هزینه‌ی یال‌ها نامنفی است و بنابراین وزن درخت پوشای کمینه از هر دور همیلتونی‌ایی کمتر است. بنابراین اگر یک درخت پوشای کمینه را T بنامیم:

$$c(T) \leq c(H^*)$$

مسئله فروشنده دوره گرد

حل با نابرابری مثلثی

- یک پیمایش کامل^۱ از T رأس‌ها را در هر بار اولین ملاقات و همچنین هنگام بازگشت از هر فرزند لیست می‌کند. پیمایش کامل این درخت را W می‌نامیم. از آنجا که هر یال درخت دقیقاً دو بار طی می‌شود، داریم:

$$c(W) = 2c(T)$$

همچنین می‌دانیم که $c(T) \leq c(H^*)$ پس:

$$c(W) \leq 2c(H^*)$$

^۱ full walk

مسئله فروشنده دوره گرد

حل با نابرابری مثلثی

- حال می‌خواهیم از روی W یک دور همیلتونی بسازیم. به دلیل نابرابری مثلثی، حذف یک رأس هزینه را افزایش نمی‌دهد. بنابراین بازدیدهای تکراری را حذف می‌کنیم، به طوری که در نهایت هر رأس تنها یک بار در لیست باشد. این همان پیمایش پیش ترتیب درخت است.
- دور همیلتونی‌ای که بدین ترتیب ساخته می‌شود را H می‌نامیم. اما چرا و H دور همیلتونی است؟ چون هر رأس را دقیقاً یکبار دارد و توجه کنید که گراف کامل است بنابراین هر ترتیب دلخواه از رأس‌های دور است.
- همچنین دیدم که:

$$c(H) \leq c(W)$$

مسئله فروشنده دوره گرد

حل با نابرابری مثلثی

- ترکیب ($c(W) \leq 2c(H^*)$ و $c(H) \leq c(W)$ می‌دهد:

$$c(H) \leq 2c(H^*)$$

که برهان را کامل می‌کند.

- ضریب تقریب این الگوریتم قابل قبول است اما معمولاً بهترین انتخاب برای این مسئله نیست. الگوریتم‌های تقریبی دیگری هم وجود دارند که در عمل بسیار بهتراند.
- همچنین ثابت می‌شود اگر $P \neq NP$ باشد، آنگاه هیچ الگوریتم تقریبی چندجمله‌ای برای این مسئله‌ی وجود ندارد. (فارغ از اینکه چه ضریب تقریبی مدنظر داشته باشد.)

طراحی الگوریتم‌های تقریبی

- در این بخش دو تکنیک قدرتمند برای طراحی الگوریتم‌های تقریبی را بررسی می‌کنیم: تصادفی‌سازی و برنامه‌ریزی خطی.
- ابتدا با یک الگوریتم تصادفی برای نسخه‌ی بهینه‌سازی مسئله‌ی ارضاه پذیری 3-CNF آغاز می‌کنیم و سپس نشان می‌دهیم چگونه می‌توان یک الگوریتم تقریبی برای نسخه‌ی وزن‌دار مسئله‌ی پوشش رأسی طراحی کرد که مبتنی بر برنامه‌ریزی خطی باشد.

طراحی الگوریتم‌های تقریبی

طراحی به وسیله الگوریتم‌های تصادفی

- همان‌طور که برخی الگوریتم‌های تصادفی پاسخ دقیق محاسبه می‌کنند، برخی دیگر پاسخ‌های تقریبی تولید می‌کنند.
- می‌گوییم یک الگوریتم تصادفی برای مسئله‌ای دارای ضریب تقریبی برابر با $\alpha(n)$ است، اگر:

$$\max \left\{ \frac{C}{C^*}, \frac{C^*}{C} \right\} \leq \alpha(n).$$

n اندازه‌ی ورودی و C هزینه‌ی مورد انتظار است.

- به بیان دیگر، الگوریتم تقریب تصادفی مشابه الگوریتم تقریب قطعی است، با این تفاوت که ضریب تقریب بر اساس هزینه‌ی مورد انتظار بیان می‌شود.

طراحی الگوریتم‌های تقریبی

طراحی به وسیله الگوریتم‌های تصادفی

- پیش‌تر با مسئله‌ی ارضا‌پذیری 3-CNF آشنا شدیم. برای ارضا‌پذیری عبارت، باید انتسابی از متغیرها وجود داشته باشد که هر بند^۱ به مقدار ۱ ارزیابی شود.
- اگر نمونه‌ای ارضا‌پذیر نباشد، شاید بخواهد بدانید که تا چه حد به ارضا‌پذیر بودن نزدیک است؛ یعنی انتسابی از متغیرها بیابیم که بیشترین تعداد بندها را ارضاء کند. این مسئله‌ی بیشینه‌سازی را ارضا‌پذیری $MAX\text{-}3\text{-CNF}$ می‌نامیم.

^۱ clause

طراحی الگوریتم‌های تقریبی

طراحی به وسیله الگوریتم‌های تصادفی

- شاید شگفتزده شوید که اگر هر متغیر با احتمال $1/2$ به ۱ و با احتمال $1/2$ به ۰ مقداردهی شود، الگوریتمی با ضریب تقریب $\frac{8}{7}$ به دست می‌آید. در ادامه دلیل آن را خواهیم دید.
- در اینجا برخلاف تعریف ارضاءپذیری ۳-CNF فرض می‌کنیم که هیچ بندی شامل یک متغیر و نقیض آن نباشد.
- فرض کنید n متغیر با نام‌های x_1, x_2, \dots, x_n و m بند داشته باشیم و هر متغیر به‌طور مستقل با احتمال $\frac{1}{2}$ برابر ۱ و با احتمال $\frac{1}{2}$ برابر ۰ قرار گیرد. حال برای هر $i = 1, 2, \dots, m$ ، متغیر تصادفی نشانگر زیر را تعریف کنید:

$$Y_i = I\{\text{بند } i \text{ ارضاء شود}\}$$

طراحی الگوریتم‌های تقریبی

طراحی به وسیله الگوریتم‌های تصادفی

- پس $Y_i = 1$ است زمانی که حداقل یکی از گزاره‌نماهای^۱ بند i برابر ۱ باشد. از آن‌جا که هیچ گزاره‌نمایی بیش از یک بار در یک بند ظاهر نمی‌شود، و همچنین فرض کردہ‌ایم که هیچ متغیر و نقیض آن در یک بند نیامده باشند، مقادیر سه گزاره‌نما در هر بند مستقل‌اند.
- یک بند تنها زمانی ارضا نمی‌شود که هر سه گزاره‌نمای آن برابر صفر باشند. بنابراین:

$$\Pr\{\text{بند } i \text{ ارضا نشود}\} = (1/2)^3 = 1/8$$

در نتیجه:

$$\Pr\{\text{بند } i \text{ ارضا شود}\} = 1 - 1/8 = 7/8$$

^۱ literal

طراحی الگوریتم‌های تقریبی

طراحی به وسیله الگوریتم‌های تصادفی

- بر اساس ویژگی‌های متغیرهای تصادفی نشانگر داریم:

$$E[Y_i] = 7/8$$

حال اگر Y تعداد کل بندهای ارضاشده باشد داریم:

$$Y = Y_1 + Y_2 + \dots + Y_m$$

طراحی الگوریتم‌های تقریبی

طراحی به وسیله الگوریتم‌های تصادفی

- پس باید امید ریاضی متغیر Y را محاسبه کنیم:

$$\begin{aligned} E[Y] &= E \left[\sum_{i=1}^m Y_i \right] \\ &= \sum_{i=1}^m E[Y_i] \\ &= \sum_{i=1}^m 7/8 \\ &= 7m/8. \end{aligned}$$

طراحی الگوریتم‌های تقریبی

طراحی به وسیله الگوریتم‌های تصادفی

- از آن‌جا که m یک کران بالا برای تعداد بندهای ارضاء‌پذیر است، ضریب تقریب در بیشترین حالت برابر خواهد بود با:

$$\frac{m}{7m/8} = 8/7$$

طراحی الگوریتم‌های تقریبی

طراحی به وسیله برنامه ریزی خطی

- در مسئله‌ی «پوشش رأسی با وزن کمینه»^۱ هر رأس مانند v وزنی مثبت به نام $w(v)$ دارد. وزن یک پوشش رأسی مانند V' به صورت زیر تعریف می‌شود:

$$w(V') = \sum_{v \in V'} w(v)$$

هدف، یافتن یک پوشش رأسی با وزن کمینه است.

- الگوریتم تقریبی که برای حالت بدون وزن دیدیم، در اینجا کارآمد نیست. زیرا برای حالت وزن‌دار جواب حاصل می‌تواند فاصله‌ی زیادی از جواب بهینه داشته باشد.

^۱ minimum-weight vertex-cover problem

طراحی الگوریتم‌های تقریبی

طراحی به وسیله برنامه ریزی خطی

- در عوض با استفاده از یک برنامه‌ریزی خطی، یک کران پایین برای وزن پوشش رأسی محاسبه می‌کنیم.
سپس این جواب را گرد کرده و از آن برای به دست آوردن یک پوشش رأسی استفاده می‌کنیم.
- ابتدا برای هر رأس $V \in v$ یک متغیر $x(v)$ تعریف می‌کنیم و الزام می‌کنیم که $x(v)$ برابر با ۰ یا ۱ باشد.
رأس v تنها در صورتی در پوشش رأسی وجود دارد که

$$x(v) = 1$$

باشد. بنابراین، قید مربوط به هر یال (u, v) این است که دستکم یکی از u و v در پوشش رأسی قرار داشته باشند، که می‌توان آن را به صورت زیر نوشت:

$$x(u) + x(v) \geq 1$$

طراحی الگوریتم‌های تقریبی

طراحی به وسیله برنامه ریزی خطی

- این دیدگاه منجر به یک «برنامه‌ریزی صحیح^۱» برای یافتن پوشش رأسی کمینه می‌شود:

$$\text{minimize} \sum_{v \in V} w(v)x(v)$$

subject to

$$x(u) + x(v) \geq 1 \quad \text{for each } (u, v) \in E$$

$$x(v) \in \{0, 1\} \quad \text{for each } v \in V$$

^۱ 0-1 integer program

طراحی الگوریتم‌های تقریبی

طراحی به وسیله برنامه ریزی خطی

- یک حالت خاص از صورت بندی بالا در نظر بگیرید که تمام وزن‌ها $w(v) = 1$ باشند. این صورت بندی معادل نسخه‌ی بهینه سازی مسئله‌ی پوشش رأسی است. می‌دانیم که پوشش رأسی انپی کامل و نسخه بهینه سازی آن انپی سخت است.
- مشخص است که این مسئله از نسخه بهینه‌سازی مسئله پوشش رأسی سخت‌تر است پس حداقل انپی سخت است.

طراحی الگوریتم‌های تقریبی

طراحی به وسیله برنامه ریزی خطی

- اما ما به دنبال حل دقیق این مسئله نیستیم پس باید قید $x(v) \in \{0, 1\}$ را با

$$0 \leq x(v) \leq 1$$

جایگزین کنیم. در نتیجه، برنامه‌ریزی خطی زیر به دست می‌آید:

$$\text{minimize} \quad \sum_{v \in V} w(v)x(v)$$

subject to

$$x(u) + x(v) \geq 1 \quad \text{for each } (u, v) \in E$$

$$x(v) \leq 1 \quad \text{for each } v \in V$$

$$x(v) \geq 0 \quad \text{for each } v \in V$$

طراحی الگوریتم‌های تقریبی

طراحی به وسیله برنامه ریزی خطی

- به این کار «ساده‌سازی»^۱ برنامه‌ریزی خطی گفته می‌شود. هر جواب مجاز برای برنامه‌ی اصلی، یک جوابی مجاز برای نسخه ساده شده است.
- بنابراین، مقدار یک جواب بهینه در برنامه‌ریزی خطی ساده شده، یک کران پایین برای مقدار جواب بهینه در برنامه‌ی اصلی است.
- در نتیجه مقدار یک جواب بهینه در برنامه‌ریزی خطی ساده شده، یک کران پایین برای وزن بهینه در مسئله‌ی «پوشش رأسی با وزن کمینه» است.

^۱ relaxation

طراحی الگوریتم‌های تقریبی

طراحی به وسیله برنامه ریزی خطی

- روش APPROX-MIN-WEIGHT-VC با استفاده از یک جواب برای نسخه ساده شده برنامه ریزی خطی آغاز می‌کند و از آن برای ساختن یک جواب تقریبی برای مسئله پوشش رأسی کمینه استفاده می‌کند:

Algorithm APPROX-MIN-WEIGHT-VC

```
function APPROX-MIN-WEIGHT-VC( $G, w$ )
1:  $C \leftarrow \emptyset$ 
2: compute  $\bar{x}$ , an optimal solution to the linear-programming relaxation
3: for each vertex  $v \in V$  do
4:   if  $\bar{x}(v) \geq \frac{1}{2}$  then
5:      $C \leftarrow C \cup \{v\}$ 
6: return  $C$ 
```

طراحی الگوریتم‌های تقریبی

طراحی به وسیله برنامه ریزی خطی

- این روش به شکل زیر عمل می‌کند: خط ۲ ساده‌شده برنامه‌ریزی خطی را صورت‌بندی کرده و آن را حل می‌کند. یک جواب بهینه که حاصل حل برنامه‌ریزی خطی است، به هر رأس مانند v یک مقدار $\bar{x}(v)$ بین ۰ و ۱ نسبت می‌دهد.

- سپس این مقدار برای انتخاب رئوسی که باید به پوشش رأسی افزوده شوند به کار می‌رود: رأس v تنها در صورتی در C قرار می‌گیرد که

$$\bar{x}(v) \geq 1/2$$

باشد.

در واقع، روش هر متغیر را به ۰ یا ۱ گرد می‌کند تا جوابی برای برنامه‌ی خطی اصلی به دست آید که جوابی برای مسئله پوشش رأسی است.

طراحی الگوریتم‌های تقریبی

طراحی به وسیله برنامه ریزی خطی

- الگوریتم APPROX-MIN-WEIGHT-VC یک الگوریتم تقریبی با ضریب ۲ و در زمان چندجمله‌ای برای مسئله پوشش رأسی با وزن کمینه است.
- از آنجا که الگوریتمی در زمان چندجمله‌ای برای حل برنامه ریزی خطی (در خط ۲) وجود دارد و حلقه‌ی for نیز در زمان چندجمله‌ای اجرا می‌شود، این الگوریتم چندجمله‌ای است.

طراحی الگوریتم‌های تقریبی

طراحی به وسیله برنامه ریزی خطی

- اکنون باید نشان دهیم که این الگوریتم یک الگوریتم تقریبی با ضریب ۲ است. فرض کنید C^* یک جواب بهینه برای مسئله‌ی پوشش رأسی کمینه باشد، و \hat{z}^* مقدار یک جواب بهینه برای نسخه ساده شده آن باشد.
- از آنجا که یک پوشش رأسی بهینه جوابی مجاز برای برنامه ساده شده است، داریم:

$$\hat{z}^* \leq w(C^*)$$

طراحی الگوریتم‌های تقریبی

طراحی به وسیله برنامه ریزی خطی

- اما چگونه گرد کردن مقادیر کسری (v, \bar{x}) , پوشش رأسی است؟
- برای دیدن این موضوع، یالی مانند (v, u) را در نظر بگیرید. در برنامه خطی ساده شده قید کردیم:

$$x(u) + x(v) \geq 1$$

- پس حداقل یکی از (v, \bar{x}) یا (u, \bar{x}) باید بزرگ‌تر یا مساوی با $1/2$ باشد. بنابراین، حداقل یکی از u یا v در پوشش رأسی قرار می‌گیرد و همه‌ی یال‌ها پوشش داده می‌شوند.

طراحی الگوریتم‌های تقریبی

طراحی به وسیله برنامه ریزی خطی

- حال می‌خواهیم ثابت کنیم:

$$w(C) \leq 2w(C^*)$$

برای اینکار ابتدا باید ثابت کنیم:

$$w(C) \leq 2z^*$$

طراحی الگوریتم‌های تقریبی

طراحی به وسیله برنامه ریزی خطی

- باید از وزن پوشش شروع کنیم:

$$\begin{aligned} z^* &= \sum_{v \in V} w(v) \bar{x}(v) \\ &\geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v) \bar{x}(v) \\ &\geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v) \cdot \frac{1}{2} \\ &= \frac{1}{2} \sum_{v \in C} w(v) \\ &= \frac{1}{2} w(C) \end{aligned}$$

طراحی الگوریتم‌های تقریبی

طراحی به وسیله برنامه ریزی خطی

- پیش‌تر دیدیم که

$$z^* \leq 2w(C^*)$$

پس داریم:

$$w(C) \leq 2z^* \leq 2w(C^*)$$

- بنابراین

$$w(C) \leq 2w(C^*)$$

- در نتیجه، الگوریتم APPROX-MIN-WEIGHT-VC دارای ضریب تقریب ۲ است.

برنامه ریزی خطی

- بسیاری از مسائل به شکل بهینه‌سازی یک تابع هدف تحت تعدادی قید مطرح می‌شوند.
- اگر بتوانید تابع هدف را به صورت یک تابع خطی از متغیرها و قیود را به صورت معادلات یا نامعادلات خطی بر این متغیرها بیان کنید، آنگاه با یک «مسئله برنامه‌ریزی خطی»^۱ روبرو هستید.

^۱ linear programming problem

- برای مثال بباید یکی از کابردهای برنامه ریزی خطی را بررسی کنیم؛ فرض کنید شما یک سیاستمدار هستید که می‌خواهید در یک انتخابات پیروز شوید.
- ناحیه انتخاباتی شما شامل سه نوع منطقه است: شهری، حومه‌ای، و روستایی. این مناطق، به ترتیب شامل ۱۰۰، ۲۰۰، و ۵۰ هزار رأی دهنده هستند. هر چند همه رأی دهنگان به پای صندوق رأی نمی‌روند، شما می‌خواهید دستکم نیمی از رأی دهنگان در هر منطقه به شما رأی دهند.

- برخی موضوعات در مناطق خاصی در جلب رأی مؤثرتر اند. موضوعات موجود عبارتند از: آمادگی برای آخرالزمان زامبی‌ها، تجهیز کوسه‌ها با لیزر، ساخت بزرگراه برای خودروهای پرنده، حق رأی برای دلفین‌ها.
- در جدول زیر نشان داده شده که با صرف هر هزار تومان تبلیغات برای هر موضوع، چه تعداد رأی از هر قشر جمعیتی به دست می‌آورید یا از دست می‌دهید. مقادیر منفی بیانگر از دست دادن رأی هستند.

policy	urban	suburban	rural
zombie apocalypse	-2	5	3
sharks with lasers	8	2	-5
highways for flying cars	0	0	10
dolphins voting	10	0	-2

- حال مسئله این است: یافتن حداقل مقدار پول لازم برای جذب ۵۰ هزار رأی شهری، ۱۰۰ هزار رأی حومه‌ای، و ۲۵ هزار رأی روستایی.
- بیایید این مسئله را مدل سازی ریاضی کنیم. در گام نخست باید تصمیم‌هایی که شما باید در طول حل مسئله بگیرید بیابید و آنها را در غالب متغیرها معرفی کنید. از آنجا که در این مسئله در مورد چهار مقدار باید تصمیم گرفته شود، چهار «متغیر تصمیم»^۱ معرفی می‌کنیم:

$$x_1, x_2, x_3, x_4$$

این متغیرها به ترتیب تعداد هزاران تومان صرف شده برای موضوعات اول تا چهارم هستند.

¹ decision variables

- حال باید قیود^۱ را مدل کنیم. به محدودیت‌های مقادیر متغیرها قید گفته می‌شود. شرط کسب دست‌کم ۵۰،۰۰۰ رأی شهری به صورت زیر بیان می‌شود:

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50$$

- به طور مشابه، قیود برای حومه و روستا چنین هستند:

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100$$

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25$$

^۱ constraints

- هر مقداردهی به متغیرهای x_1 تا x_4 که نامعادلات بالا را ارضا کند، تعداد رأی مورد نظر را کسب می‌کند.
- در نهایت، به تابع هدف فکر کنید؛ کمیتی که می‌خواهید آن را بهینه کنید. در اینجا می‌خواهیم هزینه تبلیغات کمینه شود، پس تابع هدف به این صورت است:

$$x_1 + x_2 + x_3 + x_4$$

- همچنین تبلیغات با هزینه منفی وجود ندارد. بنابراین باید داشته باشیم:

$$x_1 \geq 0, \quad x_2 \geq 0, \quad x_3 \geq 0, \quad x_4 \geq 0$$

- ترکیب نامعادلات قید با تابع هدف، منجر به چیزی می‌شود که آن را یک «برنامه خطی»^۱ می‌نامند:

$$\text{minimize} \quad x_1 + x_2 + x_3 + x_4$$

subject to

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50$$

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100$$

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25$$

$$x_1, x_2, x_3, x_4 \geq 0$$

^۱ linear program

الگوریتم‌های برنامه ریزی خطی

- الگوریتم‌های متعددی برای حل برنامه‌های خطی توسعه یافته‌اند که برخی از آن‌ها در زمان چندجمله‌ای اجرا می‌شوند و برخی دیگر نه، اما همگی آن‌ها بسیار پیچیده‌اند.
- بنابراین، در اینجا این الگوریتم‌ها را نشان نمی‌دهیم. در عوض مثالی خواهیم دید که برخی از ایده‌های پشت الگوریتم سیمپلکس^۱ را نشان می‌دهد؛ الگوریتمی که در حال حاضر رایج‌ترین روش حل است.

¹ simplex algorithm

الگوریتم‌های برنامه ریزی خطی

برنامه‌های خطی در حالت کلی

- در مسئله برنامه‌ریزی خطی، هدف بهینه‌سازی یک تابع خطی با رعایت مجموعه‌ای از نامعادلات خطی است.
- فرض کنید اعداد حقیقی a_1, a_2, \dots, a_n و متغیرهای x_1, x_2, \dots, x_n داده شده باشند. آنگاه تابع خطی^۱ f روی این متغیرها به صورت زیر تعریف می‌شود:

$$f(x_1, x_2, \dots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n = \sum_{j=1}^n a_jx_j$$

^۱ linear function

الگوریتم‌های برنامه ریزی خطی

برنامه‌های خطی در حالت کلی

- حال اگر b عددی حقیقی و f تابعی خطی باشد، آنگاه معادله

$$f(x_1, x_2, \dots, x_n) = b$$

یک تساوی خطی است و نامعادلات

$$f(x_1, x_2, \dots, x_n) \geq b \quad \text{و} \quad f(x_1, x_2, \dots, x_n) \leq b$$

نیز نامعادلات خطی هستند. ما اصطلاح کلی «محدودیت خطی»^۱ را برای اشاره به هر دو نوع به کار می‌بریم.

- توجه داشته باشید در برنامه‌ریزی خطی نامساوی اکید^۲ را مجاز نیست. ($<$, $>$)

^۱ linear constraints

^۲ strict inequalities

الگوریتم‌های برنامه ریزی خطی

برنامه‌های خطی در حالت کلی

- به طور رسمی، مسئله برنامه ریزی خطی به معنای کمینه‌سازی یا بیشینه‌سازی یک تابع خطی تحت مجموعه‌ای متناهی از محدودیت‌های خطی است. اگر هدف کمینه‌سازی باشد، آن را «برنامه خطی کمینه‌سازی»^۱ و در غیر این صورت «برنامه خطی بیشینه‌سازی»^۲ می‌نامیم.
- برای بحث درباره الگوریتم‌ها و ویژگی‌های برنامه ریزی خطی، بهتر است که از نمادگذاری استاندارد برای ورودی‌ها استفاده کنیم.
- به طور قراردادی، یک برنامه خطی بیشینه‌سازی شامل ورودی‌هایی به صورت n عدد حقیقی c_1, c_2, \dots, c_n ، m عدد حقیقی b_1, b_2, \dots, b_m و mn عدد حقیقی a_{ij} به ازای $i = 1, 2, \dots, m$ و $j = 1, 2, \dots, n$ است.

¹ minimization linear program

² maximization linear program

الگوریتم‌های برنامه ریزی خطی

برنامه‌های خطی در حالت کلی

- هدف یافتن n عدد حقیقی x_1, x_2, \dots, x_n است که عبارت زیر (تابع هدف^۱) را بیشینه کند:

$$\sum_{j=1}^n c_j x_j$$

با رعایت محدودیت‌های:

$$\sum_{j=1}^n a_{ij} x_j \geq b_i \quad \text{for } i = 1, 2, \dots, m$$

$$x_j \geq 0 \quad \text{for } j = 1, 2, \dots, n$$

^۱ objective function

الگوریتم‌های برنامه ریزی خطی

برنامه‌های خطی در حالت کلی

- برای فشرده‌تر کردن این بیان، می‌توانیم ماتریس $A = (a_{ij})$ از ابعاد $n \times m$ ، بردار $(b_i) = b$ از اندازه m ،
بردار $(c_j) = c$ از اندازه n ، و بردار $(x_j) = x$ از اندازه n را تعریف کنیم. آنگاه برنامه خطی به صورت زیر بازنویسی می‌شود:

$$\text{maximize } c^T x$$

subject to $Ax \geq b$

$$x \geq 0$$

الگوریتم‌های برنامه ریزی خطی

برنامه‌های خطی در حالت کلی

- درک معادلات بالا نیاز به کمی دانش جبر خطی دارد؛ $c^T x$ ضرب داخلی دو بردار n تایی است. حاصل ضرب ماتریس $n \times m$ در بردار n تایی، و $x \geq 0$ بدین معنی است که تمامی مؤلفه‌های بردار x باید نامنفی باشند.
- این نوع نمایش را «فرم استاندارد»^۱ یک برنامه خطی می‌نامیم، و در سراسر این فصل فرض می‌کنیم که A ، b ، و c دارای ابعاد فوق هستند.
- فرم استاندارد لزوماً با موقعیت‌های واقعی که می‌خواهید مدل‌سازی کنید مطابقت ندارد. برای مثال، ممکن است محدودیت‌های تساوی داشته باشد یا متغیرهایی که می‌توانند مقادیر منفی بگیرند. اما می‌توان هر برنامه خطی را به این فرم استاندارد تبدیل کرد.

^۱ standard form

الگوریتم‌های برنامه ریزی خطی

برنامه‌های خطی در حالت کلی

- در ادامه اصطلاحاتی را برای توصیف راه حل‌های برنامه‌های خطی معرفی می‌کنیم. یک مقداردهی خاص به متغیر x را با نماد \bar{x} نشان می‌دهیم. اگر \bar{x} تمام محدودیت‌ها را ارضاء کند، آنگاه یک راه حل مجاز^۱ است. در غیر این صورت، یک راه حل غیرمجاز^۲ است.
- یک راه حل مانند \bar{x} دارای مقدار هدف^۳ $c^T \bar{x}$ است. اگر \bar{x} یک راه حل «مجاز» باشد و مقدار هدف آن در میان همه راه حل‌های مجاز بیشینه باشد، آنگاه \bar{x} یک «راه حل بهینه»^۴ است و $c^T \bar{x}$ مقدار هدف بهینه^۵ نامیده می‌شود.

¹ feasible solution

² infeasible solution

³ objective value

⁴ optimal solution

⁵ optimal objective value

الگوریتم‌های برنامه ریزی خطی

برنامه‌های خطی در حالت کلی

- اگر یک برنامه خطی هیچ راه حل مجازی نداشته باشد، آن را یک برنامه خطی ناممکن^۱ و در غیر این صورت آن را یک برنامه خطی ممکن^۲ می‌نامیم.
- اگر یک دستگاه مختصات با n محور فرض کنیم، آنگاه هر نقطه روی این دستگاه یک مقدار دهی به بردار x است. ناحیه‌ایی روی این نمودار را که همه نقاط آن پاسخ‌های مجاز هستند «ناحیه مجاز»^۳ می‌نامیم.
- اگر یک برنامه خطی راه حل مجاز داشته باشد، اما «مقدار هدف بهینه» متناهی نداشته باشد، آنگاه ناحیه مجاز و در نتیجه برنامه خطی «نامحدود»^۴ است. عکس این قضیه الزاماً صادق نیست.

¹ infeasible

² feasible

³ feasible region

⁴ unbounded

الگوریتم‌های برنامه ریزی خطی

برنامه‌های خطی در حالت کلی

- به طور کلی، برنامه‌های خطی را می‌توان به شکل کارآمد حل کرد. دو دسته الگوریتم، به نام‌های الگوریتم‌های بیضوی^۱ و الگوریتم‌های نقطه-داخلی^۲، وجود دارند که برنامه‌های خطی را در زمان چندجمله‌ای حل می‌کنند.
- افزون بر این‌ها، الگوریتم سیمپلکس نیز وجود دارد که به طور گسترده استفاده می‌شود. اگرچه این الگوریتم در بدترین حالت، زمان چندجمله‌ای ندارد، اما در عمل عملکرد خوبی دارد.
- اینجا وارد جزئیات هیچ کدام از الگوریتم‌ها نمی‌شویم بلکه تنها به بحث چند ایده‌ی مهم می‌پردازیم.

¹ ellipsoid algorithms

² interior-point algorithms

الگوریتم‌های برنامه ریزی خطی

مثالی از یک برنامه خطی دومتغیره

- نخست، مثالی از به کارگیری یک روش هندسی برای حل یک برنامه خطی دو متغیره ارائه می‌کنیم. هرچند این مثال بلافاصله به یک الگوریتم کارآمد تعمیم نمی‌یابد، اما شما را با مفاهیم مهمی در این زمینه آشنا می‌کند.
- ابتدا برنامه خطی متغیره زیر را در نظر بگیرید:

$$\text{maximize } x_1 + x_2$$

$$\text{subject to } 4x_1 - x_2 \geq 8$$

$$2x_1 + x_2 \geq 10$$

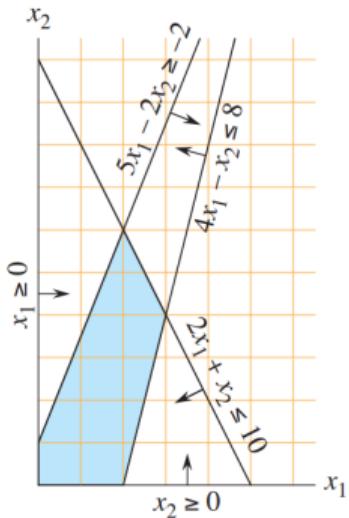
$$5x_1 - 2x_2 \leq -2$$

$$x_1, x_2 \geq 0$$

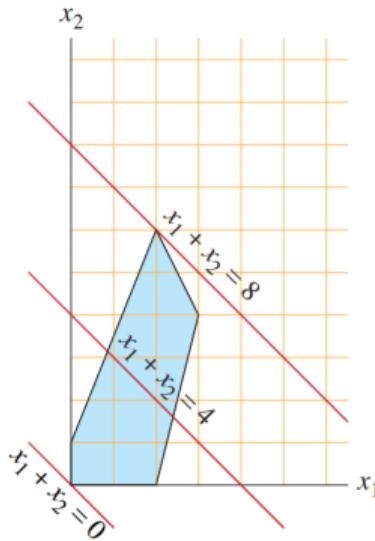
الگوریتم‌های برنامه ریزی خطی

مثالی از یک برنامه خطی دو متغیره

- اشتراک قیود یک ناحیه محدب (ناحیه آبی رنگ) را تشکیل می‌دهند که همان ناحیه مجاز است:



(a)



(b)

الگوریتم‌های برنامه ریزی خطی

مثالی از یک برنامه خطی دو متغیره

- در دو بعد، می‌توان به کمک یک روش بصری بهینه‌سازی را انجام داد؛ به ازای یک عدد حقیقی مانند z ، مجموعه نقاطی که در آنها $z = x_1 + x_2$ برقرار است، یک خط با شیب ۱– تشکیل می‌دهند. برای مثال، $x_1 + x_2 = 0$ خطی با شیب ۱– از مبدأ است.
- اشتراک این خط با ناحیه‌ی مجاز، مجموعه‌ای از نقاط مجاز با مقدار تابع هدف صفر است که در این حالت تنها شامل نقطه $(0, 0)$ می‌شود.
- به طور کلی، برای هر مقدار z ، اشتراک خط $z = x_1 + x_2$ با ناحیه‌ی مجاز، مجموعه‌ای از نقاط مجاز با مقدار تابع هدف z خواهد بود.
- از آن‌جا که ناحیه‌ی مجاز در این شکل کراندار است، باید یک مقدار بیشینه z وجود داشته باشد به طوری که اشتراک خط $z = x_1 + x_2$ با ناحیه‌ی مجاز تهی نباشد. همانطور که در شکل نشان داده شده، رأس ناحیه‌ی مجاز در $(2, 6)$ مقدار بهینه را با ارزش تابع هدف ۸ به دست می‌دهد.

الگوریتم‌های برنامه ریزی خطی

مثالی از یک برنامه خطی دو متغیره

- اتفاقی نیست که راه حل بهینه در یک رأس ناحیه مجاز ظاهر شد. بیشترین مقدار Z برای خط $x_1 + x_2 = z$ الزاماً بر مرز ناحیه مجاز رخ می‌دهد.
- بنابراین اشتراک این خط با مرز ناحیه مجاز، یک رأس منفرد یا یک پاره‌خط است. اگر اشتراک یک رأس باشد، همان رأس تنها راه حل بهینه است.
- اگر اشتراک یک پاره‌خط باشد، تمام نقاط آن پاره‌خط مقدار تابع هدف یکسانی دارند، و دو انتهای آن نیز راه حل بهینه خواهند بود. چون هر انتهای پاره‌خط یک رأس است، در این حالت هم یک راه حل بهینه در رأس وجود دارد.

الگوریتم‌های برنامه ریزی خطی

مثالی از یک برنامه خطی دو متغیره

- اگرچه ترسیم گرافیکی برای مسائل با بیش از دو متغیر آسان نیست، اما همان شهود برقرار می‌ماند. برای سه متغیر، هر قید یک نیم‌فضا در فضای سه‌بعدی تعریف می‌کند.
- اشتراک این نیم‌فضاهای ناحیه‌ی مجاز را تشکیل می‌دهد. مجموعه نقاطی که مقدار تابع هدف برابری دارند، اکنون یک صفحه است.
- اگر همه ضرایب تابع هدف نامنفی باشند و مبدأ یک راه حل مجاز باشد، با دور کردن این صفحه از مبدأ در جهت عمود بر تابع هدف، نقاطی با مقادیر بزرگ‌تر تابع هدف به دست می‌آید. (در غیر این صورت، تصویر شهودی کمی پیچیده‌تر می‌شود.)
- همانند دو بعد، چون ناحیه‌ی مجاز محدب است، مجموعه نقاطی که مقدار بهینه را می‌سازند، باید شامل یک رأس باشند.

الگوریتم‌های برنامه ریزی خطی

مثالی از یک برنامه خطی دو متغیره

- به طور کلی، اگر n متغیر وجود داشته باشد، هر قید یک نیم فضا در فضای n بعدی تعریف می‌کند.
- ناحیه‌ی مجاز حاصل از اشتراک این نیم فضاهای یک سیمپلکس¹ نامیده می‌شود.
- تابع هدف اکنون یک ابرصفحه است و به دلیل محدب بودن، همچنان راه حل بهینه در یکی از رأس‌های سیمپلکس رخ می‌دهد.

¹ simplex

الگوریتم‌های برنامه ریزی خطی

مثالی از یک برنامه خطی دو متغیره

- الگوریتم سیمپلکس از یک رأس سیمپلکس آغاز کرده و در هر تکرار از یک رأس به رأس مجاور حرکت می‌کند که مقدار تابع هدف در آن بزرگ‌تر یا مساوی با مقدار کنونی باشد.
- الگوریتم زمانی متوقف می‌شود که به یک بیشینه محلی برسد، یعنی رأسی که همه رأس‌های مجاور آن مقدار تابع هدف کوچک‌تری دارند. ثابت می‌شود، این بهینه‌ی موضعی همان بهینه‌ی سراسری است.
- الگوریتم سیمپلکس اغلب مسائل برنامه‌ریزی خطی را سریع حل می‌کند. اما برای برخی ورودی‌های خاص که با دقت طراحی شده باشند، زمان اجرای نمایی دارد.

صورت بندی برنامه‌ریزی خطی

- برنامه‌ریزی خطی کاربردهای بسیاری دارد. در این بخش، بررسی می‌کنیم که چگونه چند مسائل را می‌توان به صورت برنامه‌ریزی خطی فرمول بندی کرد.
- در بحث برنامه ریزی خطی، مهم‌ترین مهارت تشخیص این است که چه زمانی می‌توان مسئله‌ای را به صورت یک برنامه‌ی خطی نوشت. اگر مسئله‌ای را به برنامه‌ای خطی با اندازه‌ی چندجمله‌ای تبدیل کنید، می‌توان آن را در زمان چندجمله‌ای حل کرد. بسته‌های نرم‌افزاری متعددی برای حل برنامه‌های خطی به صورت کارآمد وجود دارند.

صورت بندی برنامه‌ریزی خطی

- پیش‌تر هنگام حل مسائل گراف از نشانه گذاری‌هایی مانند $d : v$ و $f : (u, v)$ استفاده می‌کردیم.
با این حال در برنامه‌های خطی، هنگام بیان متغیرها، رأس‌ها و یال‌ها را با زیرنویس مشخص می‌کنیم. مثلاً وزن کوتاه‌ترین مسیر برای رأس v را به صورت d_v می‌نویسیم.
- مسائلی مانند کوتاه‌ترین مسیر تک مبدأ و شار بیشینه، با استفاده از برنامه‌ریزی خطی برای قابل حل اند. اما برای این مسائل قبلاً نیز الگوریتم‌های کارآمدی می‌شناختیم.
- در واقع، یک الگوریتم که برای مسئله‌ی خاصی طراحی شده باشد، اغلب کاراتر از برنامه‌ریزی خطی است.
(مانند الگوریتم دیکسترا برای مسئله‌ی کوتاه‌ترین مسیر تک‌مبدأ)

صورت بندی برنامه‌ریزی خطی

- اما قدرت واقعی برنامه‌ریزی خطی از توانایی آن در حل مسائل جدید ناشی می‌شود.
- برای مثال مسئله انتخابات را که در ابتدای این فصل با آن رو به رو شدیم در نظر بگیرید. این مسئله توسط هیچ‌یک از الگوریتم‌هایی که تا کنون مطالعه کردہ‌ایم حل نمی‌شود، اما با برنامه‌ریزی خطی می‌توان آن را حل کرد.
- برنامه‌ریزی خطی به ویژه زمانی مفید است که بخواهیم گونه‌های تعمیم‌یافته‌ی مسائلی را حل کنیم که هنوز الگوریتم کارآمدی برای آن‌ها نمی‌شناسیم. به عنوان نمونه، تعمیم‌هایی از مسئله‌ی شار بیشینه را بررسی خواهیم کرد.

صورت بندی برنامه‌ریزی خطی

شار با کمترین هزینه

- فرض کنید در مسئله شار بیشینه، علاوه بر یک ظرفیت $c(u, v)$ برای هر یال (u, v) ، یک هزینه‌ی حقیقی $a(u, v)$ نیز داده شده است.
- به طوری که اگر f_{uv} واحد شار از یال (u, v) ارسال کنید، هزینه‌ای برابر $a(u, v) \cdot f_{uv}$ خواهد داشت.
- هدف آن است که دقیقاً d واحد شار از s به t ارسال کنیم، به‌گونه‌ای که مجموع هزینه

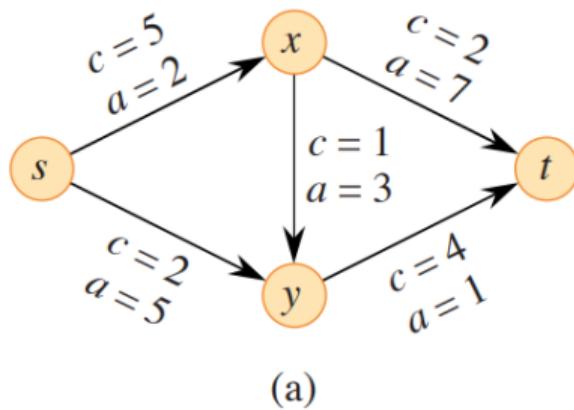
$$\sum_{(u,v) \in E} a(u, v) \cdot f_{uv}$$

کمینه شود. این مسئله به نام مسئله‌ی شار با کمترین هزینه^۱ شناخته می‌شود.

^۱ minimum-cost fow problem

صورت بندی برنامه ریزی خطی شار با کمترین هزینه

- شکل زیر نمونه‌ای از این مسئله را نشان می‌دهد. هدف ارسال ۴ واحد شار از s به t با کمترین هزینه است.



صورت بندی برنامه‌ریزی خطی

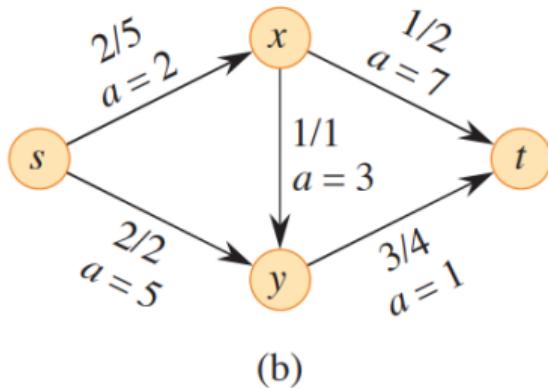
شار با کمترین هزینه

- هر شار مجاز، هزینه‌ای برابر $\sum_{(u,v) \in E} a(u,v) \cdot f_{uv}$ تحمیل می‌کند.
- پرسش این است: کدام شار ۴ واحدی این هزینه را کمینه می‌کند؟

صورت بندی برنامه‌ریزی خطی

شار با کمترین هزینه

- شکل زیر یک راه حل بهینه را نشان می‌دهد که هزینه‌ی کل آن نوشته شده است:



$$\sum_{(u,v) \in E} a(u,v) \cdot f_{uv} = (2 \cdot 2) + (5 \cdot 2) + (3 \cdot 1) + (7 \cdot 1) + (1 \cdot 3) = 27$$

صورت بندی برنامه‌ریزی خطی

شار با کمترین هزینه

- الگوریتم‌هایی با زمان چندجمله‌ای که به‌طور خاص برای حل این مسئله طراحی شده‌اند وجود دارند. با این حال، مسئله‌ی شار با کمترین هزینه را می‌توان به‌صورت یک برنامه‌ی خطی بیان کرد:

$$\text{minimize} \quad \sum_{(u,v) \in E} a(u,v) \cdot f_{uv}$$

subject to

$$f_{uv} \leq c(u,v) \quad \text{for each } u, v \in V$$

$$\sum_{v \in V} f_{vu} - \sum_{v \in V} f_{uv} = 0 \quad \text{for each } u \in V - \{s, t\}$$

$$\sum_{v \in V} f_{sv} - \sum_{v \in V} f_{vs} = d$$

$$f_{uv} \geq 0 \quad \text{for each } u, v \in V$$

صورت بندی برنامه‌ریزی خطی

شار چندکالایی

- فرض کنید یک کارخانه توب‌های هاکی تولید می‌کند و باید آن‌ها را به از انبار برساند. این وضعیت به صورت یک شبکه شار مدل می‌شود: گره مبدأ نمایانگر محل تولید، گره‌های میانی مسیرهای حمل و نقل، و گره مقصد نمایانگر انبار است. در مسئله شار بیشینه هدف این بود که مشخص کنیم حداقل چه تعداد توب می‌تواند از کارخانه به انبار برسد.
- فرض کنید در بخش جدید، اقلام دیگری مثل چوب و کلاه هاکی هم تولید می‌شود. هر قطعه در کارخانه مخصوص به خود ساخته می‌شود، انبار جداگانه‌ای دارد، و هر روز باید از کارخانه به انبار منتقل شود. این نمونه‌ای از مسئله شار چندکالایی^۱ است.

¹ multi-commodity flow

صورت بندی برنامه‌ریزی خطی

شار چندکالایی

- فرض کنید، k کالای مختلف داریم:

$$K_1, K_2, \dots, K_k$$

که هر کالای i با سه تایی

$$K_i = (s_i, t_i, d_i)$$

تعريف می‌شود. که در آن، رأس s_i مبدأ کالای i ، رأس t_i مقصد آن، و d_i شار مورد نیاز آن کالا است.

صورت بندی برنامه‌ریزی خطی

شار چندکالایی

- برای هر کالا i شاری به صورت تابع حقیقی f_i تعریف می‌شود، به گونه‌ای که f_{iuv} شار کالای i از رأس u به رأس v باشد. این تابع باید قیود ظرفیت و پایستگی شار را رعایت کند. شار کل^۱ روی یال (u, v) برابر است با مجموع شارهای کالاهای مختلف:

$$f_{uv} = \sum_{i=1}^k f_{iuv}$$

و باید قید

$$f_{uv} \leq c(u, v)$$

را رعایت کند.

- این مسئله تابع هدفی ندارد؛ پرسش فقط این است که آیا چنین شاری وجود دارد یا نه. بنابراین برنامه‌خطی این مسئله دارای تابع هدف تهی^۲ است:

¹ aggregate flow

² null objective function

صورت بندی برنامه‌ریزی خطی

شار چندکالایی

- صورت بندی مسئله شار چندکالایی به این شکل است:

minimize 0

subject to

$$\sum_{i=1}^k f_{iuv} \leq c(u, v) \quad \text{for each } u, v \in V$$
$$\sum_{v \in V} f_{iuv} - \sum_{v \in V} f_{ivu} = 0 \quad \text{for each } i = 1, 2, \dots, k \text{ and each } u \in V - \{s_i, t_i\}$$
$$\sum_{v \in V} f_{is_i v} - \sum_{v \in V} f_{iv s_i} = d_i \quad \text{for each } i = 1, 2, \dots, k$$
$$f_{iuv} \geq 0 \quad \text{for each } u, v \in V \text{ and each } i = 1, 2, \dots, k$$

صورت بندی برنامه‌ریزی خطی

شار چندکالایی

- جالب است بدانید تنها الگوریتم چندجمله‌ای شناخته شده برای این مسئله، بر مبنای برنامه ریزی خطی است.

الگوریتم‌های موازی

- الگوریتم‌های سریالی^۱ برای اجرا روی یک پردازنده طراحی شده‌اند. این فصل، مدل الگوریتمی ما را گسترش می‌دهد تا الگوریتم‌های موازی^۲ را نیز در بر بگیرد، که در آن‌ها چندین دستور می‌توانند به طور همزمان اجرا شوند.
- به طور خاص، ما مدل الگوریتم‌های وظیفه-موازی^۳ را بررسی می‌کنیم. ساده‌ترین نوع الگوریتم‌های وظیفه-موازی، الگوریتم‌های انشعاب-الحاق^۴ نام دارد که ما در این فصل این نوع را مطالعه می‌کنیم.

¹ serial algorithms

² parallel algorithms

³ task-parallel

⁴ fork-join

- الگوریتم‌های انشعاب-الحاق را می‌توان با استفاده از افزودن امکانات ساده‌ایی به همان کد سریالی عادی، به صورت واضح بیان کرد. این الگوریتم‌ها در عمل نیز می‌توانند به صورت کارآمد پیاده‌سازی شوند.
- به کامپیوترهایی که بیش از یک پردازنده داشته باشند چندهسته‌ایی^۱ به هر پردازنده یک هسته گفته می‌شود و هر هسته می‌تواند به حافظه مشترک^۲ کامپیوتر دسترسی داشته باشد.
- یکی از روش‌های برنامه‌نویسی کامپیوترهای چندهسته‌ای، موازی‌سازی نخ‌محور^۳ است. این مدل از یک انتزاع نرم‌افزاری به نام «پردازنده‌ی مجازی» یا نخ^۴ استفاده می‌کند. نخ‌ها همگی حافظه‌ی مشترکی را به اشتراک می‌گذارند. هر نخ شمارنده‌ی^۵ اختصاصی خود را دارد و می‌تواند به طور مستقل از نخ‌های دیگر اجرا شود.

¹ multicore

² shared memory

³ thread parallelism

⁴ thread

⁵ program counter

- اما برنامهنویسی نخ محور دشوار و مستعد خطاست. برای مثال تقسیمکار میان نخها به گونه‌ای که هر نخ بار کاری تقریباً برابری دریافت کند، ممکن است پیچیده باشد. بنابراین به سراغ مدل وظیفه-موازی می‌رویم.
- در این مدل یک لایه‌ی نرم‌افزاری روی نخها قرار می‌دهیم تا پردازنده‌های یک کامپیوتر چندهسته‌ای را مدیریت کند. به این لایه نرم افزاری سکوی وظیفه-موازی^۱ گفته می‌شود که می‌تواند یک کتابخانه یا یک زبان برنامه‌نویسی باشد.
- در برنامه‌نویسی وظیفه-موازی، برنامه‌نویس مشخص می‌کند که چه وظایف محاسباتی‌ای ممکن است به صورت موازی اجرا شوند، اما مشخص نمی‌کند که کدام نخ یا پردازنده، آن وظیفه را اجرا کند.

^۱ task-parallel platform

- سکوی وظیفه-موازی دارای یک زمانبند است که به طور متوازن وظایف را میان پردازنده‌ها تقسیم می‌کند.
- پس برنامهنویس درگیر پروتکل‌های ارتباطی، توازن بار کاری، و سایر پیچیدگی‌های برنامهنویسی با نخ نمی‌شود.
- الگوریتم‌هایی وظیفه موازی^۱ نیز تعمیم یافته همان الگوریتم‌های سریالی هستند.

^۱ Task-parallel algorithms

- تقریباً همهی محیط‌های وظیفه-موازی، از موازی‌سازی انشعاب-الحاق پشتیبانی می‌کنند که معمولاً در دو ویژگی زبانی ظاهر می‌شود: انشعاب^۱ و حلقه موازی^۲.
- انشعاب یعنی بعد از فراخوانی یک تابع، فراخوانی کننده به اجرای خود ادامه دهند. در حالی که تابع فراخوانی شده نیز مشغول محاسبه‌ی نتیجه‌اش است.
- حلقه‌ی موازی مانند یک حلقه‌ی for معمولی است، با این تفاوت که چندین تکرار از حلقه می‌توانند به‌طور همزمان اجرا شوند.

¹ spawning

² parallel loop

- برخلاف مدل چند نخی برنامه نویس در اینجا مشخص نمی‌کند که کدام وظایف «باید» حتماً به صورت موازی اجرا شوند، بلکه تنها مشخص می‌کند که کدام وظایف «می‌توانند» به صورت موازی اجرا شوند. این ویژگی از مدل وظیفه-موازی به ارث برده شده.
- موازی‌سازی انشعباب-الحاق چندین مزیت دارد:
 - این مدل برنامه‌نویسی تعمیم یافته مدل برنامه‌نویسی سریالی است. که با آن آشنایی کامل داریم. برای توصیف یک الگوریتم موازی انشعباب-همگرا، فقط سه واژه‌ی کلیدی به شبه کدها اضافه می‌شود:

parallel, spawn, sync

- حذف این واژه‌های کلیدی از شبهکد موازی منجر به ساخت شبهکد سریالی برای همان مسئله می‌شود که آن را «تصویر سریالی»^۱ الگوریتم موازی می‌نامیم.
- ۲ مدل وظیفه-موازی راهی دقیق برای سنجش میزان موازی‌سازی فراهم می‌آورد.
- ۳ با استفاده از قابلیت انشعاب بسیاری از الگوریتم‌های تقسیم‌وحل به سادگی تبدیل به الگوریتم‌های موازی می‌شوند. همچنین، همانند الگوریتم‌های سریالی تقسیم‌وحل، الگوریتم‌های موازی در مدل انشعاب-الحاق نیز توسط رابطه‌های بازگشتی قابل تحلیل اند.
- ۴ انشعاب-الحاق تنها یک مدل تئوری نیست و تقریباً همهٔ محیط‌های چندهسته‌ای از آن پشتیبانی می‌کنند.

^۱ serial projection

مبانی موازی سازی انشعاب-الحاق

- بحث برنامه نویسی موازی را با مسئله‌ی محاسبه‌ی اعداد فیبوناچی به صورت بازگشته آغاز می‌کنیم.
- ابتدا به یک محاسبه‌ی ساده‌ی دنباله فیبوناچی نگاه می‌کنیم که هرچند ناکارآمد است، اما نحوه‌ی بیان موازی سازی در شبکه کد را نشان می‌دهد.

- الگوریتم FIB عدد فیبوناچی n را به صورت بازگشتی محاسبه می‌کند.

Algorithm FIB

```
function FIB(n)
1: if  $n \leq 1$  then
2:   return  $n$ 
3: else
4:    $x \leftarrow \text{FIB}(n - 1)$ 
5:    $y \leftarrow \text{FIB}(n - 2)$ 
6:   return  $x + y$ 
```

مبانی موازی انشعاب-الحاق

- برای تحلیل این الگوریتم، فرض کنید $T(n)$ زمان اجرای $\text{FIB}(n)$ را نشان دهد. $\text{FIB}(n)$ شامل دو فراخوانی بازگشتی به علاوه‌ی مقداری کار ثابت است:

$$T(n) = T(n - 1) + T(n - 2) + \Theta(1)$$

- جواب این رابطه‌ی بازگشتی با استفاده از روش جایگزینی¹ به دست می‌آید که نتیجه می‌دهد:

$$T(n) = \Theta(\phi^n)$$

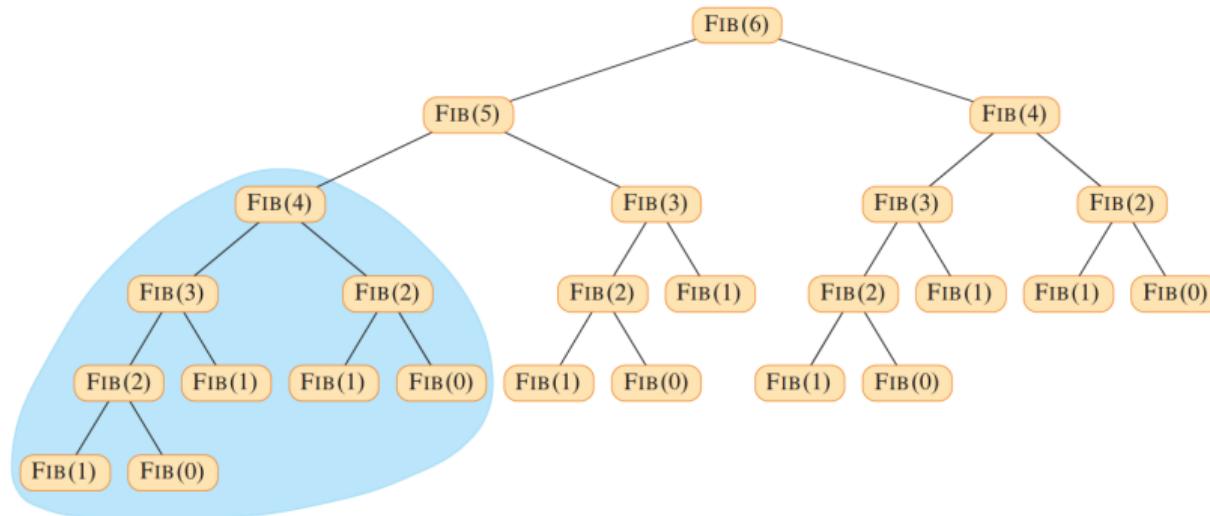
که در آن

$$\phi = \frac{1 + \sqrt{5}}{2}$$

¹ substitution method

مبانی موازی انشعاب-الحاق

- دیدم که زمان اجرای این روش نمایی است، بنابراین بسیار کند است. باید ببینیم چرا این الگوریتم ناکارآمد است. شکل زیر درخت فراخوانی‌های بازگشتی را نشان می‌دهد که توسط الگوریتم FIB ایجاد می‌شوند.



مبانی موازی سازی انشعاب-الحاق

- گرچه رویه‌ی FIB روش ناکارآمدی است، می‌تواند به ما کمک کند تا با مفاهیم موازی سازی آشنا شویم.
- شاید ابتدایی‌ترین مفهوم این است که اگر دو وظیفه‌ی موازی روی داده‌های کاملاً مستقل عمل کنند آنگاه تفاوتی نمی‌کند که این دو همزمان اجرا شوند یا به صورت سریالی و یکی پس از دیگری. در هر دو صورت نتیجه یکسان است.
- به عنوان مثال، در الگوریتم FIB دو فراخوانی بازگشتی $FIB(n - 1)$ و $FIB(n - 2)$ می‌توانند با این‌مانی کامل به صورت موازی اجرا شوند، زیرا هیچ‌یک اثری بر محاسبات دیگری ندارد.

مبانی موازی سازی انشعاب-الحاق

کلیدواژه‌های موازی سازی

- رویه‌ی P-FIB، اعداد فیبوناچی را محاسبه می‌کند، اما در شبکه ک خود از کلیدواژه‌های spawn و sync برای نمایش موازی بودن استفاده می‌کند.

Algorithm P-FIB

```
function P-FIB( $n$ )
1: if  $n \leq 1$  then
2:   return  $n$ 
3: else
4:    $x \leftarrow$  spawn P-FIB( $n - 1$ )           // don't wait for subroutine to return
5:    $y \leftarrow$  P-FIB( $n - 2$ )                 // in parallel with spawned subroutine
6:   sync                                     // wait for spawned subroutine to finish
7:   return  $x + y$ 
```

مبانی موازی سازی انشعاب-الحاق

کلیدواژه‌های موازی سازی

- اگر کلیدواژه‌های `spawn` و `sync` از رویه‌ی P-FIB حذف شوند، متن شبکه‌کد حاصل دقیقاً مشابه رویه‌ی FIB خواهد بود.
- «تصویر سریالی»^۱ الگوریتم موازی، یک الگوریتم سریالی است که با حذف دستورات موازی سازی به دست می‌آید.
- اگر یک الگوریتم موازی داشته باشیم آنگاه، تصویر سریالی آن همواره یک شبکه‌کد سریالی برای حل همان مسئله است.

^۱ serial projection

مبانی موازی سازی انشعاب-الحاق

گراف توازی

- زمانی که کلیدواژه‌ی `spawn` قبل از یک فراخوانی رویه باید، می‌گوییم «فراخوانی موازی»^۱ رخ می‌دهد؛ مانند خط ۳ در P-FIB
- رویه‌ای که دستور `spawn` را اجرا می‌کند (والد)، می‌تواند به‌طور موازی با زیررویه‌ی ایجادشده (فرزنده) ادامه یابد. در حالی که در حالت سریالی والد باید منتظر بماند تا اجرای فرزند تمام شود.
- در این مثال، در حالی که فرزند ایجادشده در حال محاسبه‌ی $P - FIB(n - 1)$ است، والد می‌تواند در خط ۴ محاسبه‌ی $P - FIB(n - 2)$ را موازی با فرزند اجرا کند.

^۱ Spawning

مبانی موازی سازی انشعاب-الحاق

گراف توازی

- از آنجا که رویه‌ی P-FIB بازگشتی است، این دو فراخوانی خودشان نیز فرزندهای موازی ایجاد می‌کنند، و فرزندانشان نیز همین کار را انجام می‌دهند؛ بنابراین یک درخت بسیار بزرگ شکل می‌گیرد که همگی به طور موازی اجرا می‌شوند.
- البته کلیدواژه‌ی spawn به این معنی نیست که یک رویه الزاماً باید موازی با فرزندانش اجرا شود، تنها می‌گوید یک فراخوانی می‌تواند موازی اجرا شود.
- به طور کلی کلیدواژه‌های موازی سازی، موازی بودن محاسبات را به طور «منطقی»^۱ بیان می‌کنند. یعنی نشان می‌دهند که کدام بخش‌ها می‌توانند موازی اجرا شوند.

¹ logical parallelism

مبانی موازی سازی انشعاب-الحاق

گراف توازی

- در زمان اجرا، واحدی به نام زمان‌بند^۱ تعیین می‌کند کدام زیر محاسبات^۲ واقعاً موازی اجرا شوند.
- همچنین یک رویه نمی‌تواند از مقادیر بازگشته فرزندان خود استفاده کند، مگر پس از اجرای یک دستور sync (مانند خط ۵).
- کلیدواژه sync بیان می‌کند که رویه باید منتظر بماند تا همهٔ فرزندان آن پایان یابند، سپس به اجرا ادامه دهد.
- علاوه بر همگام‌سازی صریح با استفاده از دستور sync این فرض ساده‌کننده را نیز می‌پذیریم که هر رویه پیش از دستور return به طور ضمنی یک sync اجرا می‌کند؛ این امر تضمین می‌کند که همهٔ فرزندان پیش از پایان والد به اتمام برسند.

^۱ scheduler

^۲ subcomputations

مبانی موازی سازی انشعاب-الحاق

گراف توازی

- برای درک اجرای یک محاسبهٔ موازی می‌توان آن را به صورت یک گراف جهت‌دار بدون دور نشان داد. این گراف یک که یک گراف توازی^۱ نامیده می‌شود.
- به طوری که رأس‌های این گراف نشان‌دهندهٔ دستورالعمل‌های اجرا شده، و یال‌های آن وابستگی بین دستورالعمل‌ها را نمایش می‌دهند، به طوری که اگر $E \in E(u, v)$ باشد، به این معناست که دستور u باید پیش از دستور v اجرا شود.
- اما نمایش هر دستور به عنوان یک رأس در گراف توازی، گراف را بی دلیل شلوغ می‌کند. بنابراین اگر دنباله‌ایی از دستورها شامل هیچ کلیدواژهٔ موازی‌سازی یا دستور کنترل توابع نباشد (به طور دقیق، spawn، sync، فراخوانی و بازگشت توابع)، کل این زنجیره را در قالب یک بلوک^۲ گروه‌بندی می‌کنیم.

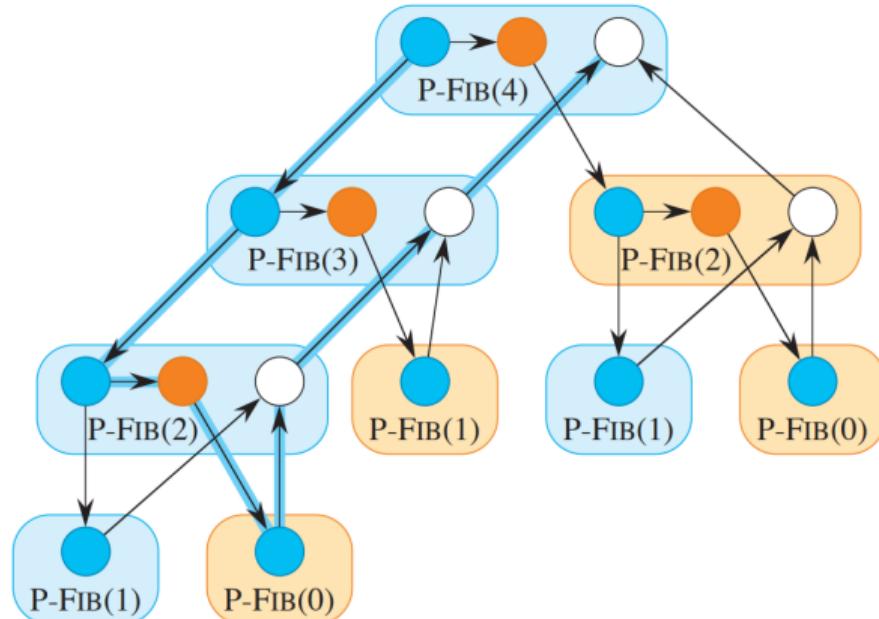
^۱ parallel trace

^۲ strand

مبانی موازی سازی انشعاب-الحاق

گراف توازی

- به عنوان مثال، شکل زیر یک گراف توازی که از اجرای $P - FIB(4)$ حاصل می‌شود را نشان می‌دهد.



مبانی موازی سازی انشعاب-الحاق

گراف توازی

- در شکل، هر دایره یک بلوکی محاسباتی را نشان می‌دهد؛ دایره‌های آبی شامل دستورات تابع FIB از اول تا فراخوانی (۱) P-FIB($n-1$) هستند.
- دایره‌های نارنجی از فراخوانی (۲) P-FIB($n-2$) شروع و با دستور sync پایان می‌یابند. و دایره‌های سفید شامل دستورات بعد از پس از sync هستند. بلوک‌هایی که متعلق به یک تابع هستند گروه‌بندی می‌شوند؛ رنگ آبی برای توابع فراخوانی شده با spawn و رنگ قهوه‌ای برای توابعی که فراخوانی معمولی شده‌اند.
- یال‌های نشان‌داده شده با رنگ آبی مسیر بحرانی را نشان می‌دهند که در آینده بیشتر با آن آشنا خواهیم شد.

مبانی موازی سازی انشعاب-الحاق

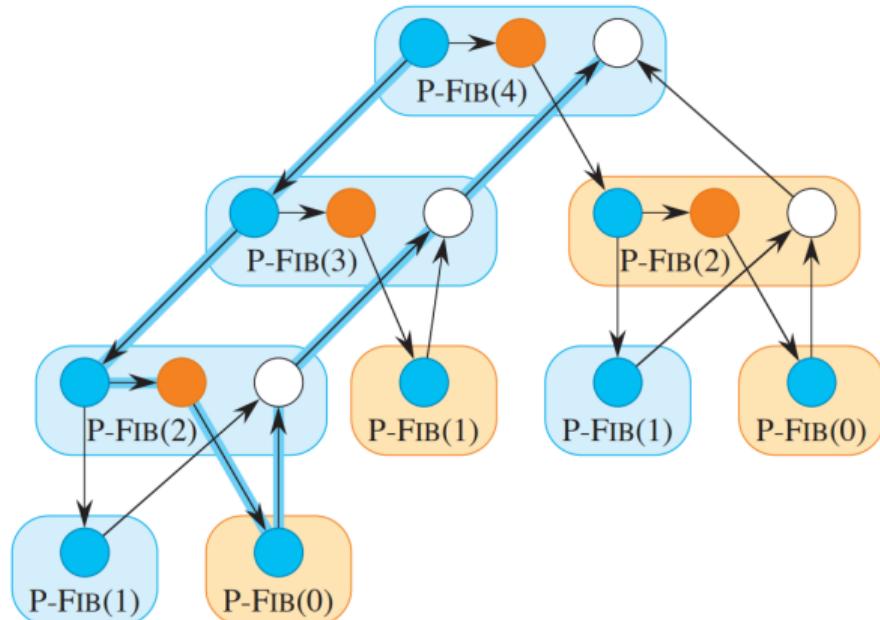
گراف توازی

- همانطور که دیدیم بلوک‌ها شامل دستورهای کنترل توازی یا کنترل تابع نمی‌باشند؛ بلکه این وابستگی‌های به صورت یال نمایش داده شوند.
- هنگامی که یک والد یک فرزند را فراخوانی می‌کند، گراف توازی شامل یالی از بلوکی که فراخوانی را اجرا می‌کند در والد به نخستین بلوکی فرزند ایجاد شده می‌کنیم.
- این حالت در شکل، توسط یک یال از بلوکی نارنجی در $P - FIB(4) - P$ به بلوکی آبی در $(P - FIB(2)) - P$ نشان داده شده است.
- زمانی که آخرین بلوک در فرزند بازمی‌گردد، یالی از آخرین بلوک تابع فرزند به بلوک بعدی در والد (بلوک بعد از فراخوانی کننده) اضافه می‌کنیم. نمونه‌ی آن یال از بلوکی سفید در $(P - FIB(2)) - P$ به بلوکی سفید در $(P - FIB(4)) - P$ است.

مبانی موازی سازی انشعاب-الحاق

گراف توازی

- می‌توانید به یال‌هایی که در بالا توصیف شد در شکل دقت کنید:



مبانی موازی سازی انشعباب-الحاق

گراف توازی

- چیزی که در بالا توصیف شد فراخوانی عادی بود. گفتیم زمانی که قبل از فراخوانی کلمه `spawn` باید تبدیل به فراخوانی موازی می‌شود. باید فراخوانی موازی را نیز در شکل بررسی کنیم.
- مانند فراخوانی عادی، در اینجا هم از والد به فرزند یال وجود دارد، اما یک یال دیگر نیز وجود دارد که نشان می‌دهد در فرزند، اجرا می‌تواند به بلوک بعدی انتقال یابد در حالی که بلوک فراخوانی کننده در حال اجراست.
- به عنوان مثال، یال از بلوکی آبی در $P - FIB(4) - P$ به بلوکی نارنجی در $(P - FIB(4) - P)$ نمونه‌ای از چنین یالی است.
- همچنین، مانند فراخوانی عادی، از آخرین بلوک فرزند یک یال خارج می‌شود؛ که در فراخوانی عادی به بلوک بعد از فراخوانی کننده می‌رود. اما در فراخوانی موازی به بلوک بعد از دستور `sync` در والد می‌رود.

مبانی موازی‌سازی انشعاب-الحاق

گراف توازی

- اگر در گراف توازی مسیری جهت‌دار از u به v داشته باشد، می‌گوییم این دو بلوک «سری منطقی» هستند.^۱
اگر هیچ مسیری نه از u به v و نه از v به u وجود نداشته باشد، این دو بلوک «موازی منطقی»^۲ هستند.
- در تحلیل‌ها فرض می‌کنیم که کامپیوتر ما شامل مجموعه‌ای از پردازنده‌ها و یک حافظه‌ی مشترک «سازگار ترتیبی»^۳ است.
- برای اینکه سازگاری ترتیبی را بهتر درک کنیم باید کمی در مورد دستورات سطح پایین (asmbl) کار با حافظه بیشتر بدانیم.

¹ logically in series

² logically in parallel

³ sequentially consistent

مبانی موازی سازی انشعاب-الحاق

گراف توازی

- دستورهای دسترسی به حافظه به دو صورت‌اند:
load : کپی داده از حافظه به ثبات پردازنده
store : کپی داده از ثبات پردازنده به حافظه

- برای نمونه، دستور زیر شامل چندین load و store است:

$$x = y + z$$

- در یک رایانهٔ موازی، ممکن است چند پردازنده همزمان به حافظه دسترسی داشته باشند. سازگاری ترتیبی سلامت داده‌ها را تضمین می‌کند.

مبانی موازی سازی انشعاب-الحاق

گراف توازی

- در واقع رفتار حافظه مشترک طوری است که گویی در هر لحظه تنها یک دستور از یکی از پردازنده‌ها اجرا شده است. در حالی که ممکن است چندین انتقال داده به‌طور فیزیکی همزمان رخ دهند.
- محاسبات وظیفه-موازی، توسط سیستم زمان روی پردازنده‌ها مدیریت می‌شوند. در این حالت ترتیب اجرای دستورات روی حافظه مطابق یک مرتب‌سازی توپولوژیکی از گراف توازی است.
- این بدان معناست که می‌توان اجرای واقعی را به صورت یک توالی خطی از دستورها تصور کرد. اگرچه زمان‌بند، می‌تواند ترتیب اجرا را بین دفعات مختلف اجرا تغییر دهد، نتیجه محاسبه همواره معادل اجرای سریالی است.

مبانی موازی سازی انشعاب-الحاق

گراف توازی

- علاوه بر فرض‌های قبلی در مورد رایانه مورد استفاده، دو فرض دیگر هم اضافه می‌کنم:
 - ۱ همه‌ی پردازنده‌ها توان محاسباتی برابر دارند.
 - ۲ هزینه‌ی زمان‌بندی نادیده گرفته می‌شود.
- هرچند این فرض دوم خوش‌بینانه به نظر می‌رسد، در عمل با یک پیاده‌سازی مناسب، سربار زمان‌بندی ناچیز است.

مبانی موازی سازی انشعاب-الحاق

معیارهای کارایی

- می‌توانیم کارایی یک الگوریتم وظیفه-موازی را با استفاده از تحلیل کار-گستره^۱ بسنجیم؛ تحلیلی که بر اساس دو معیار کار^۲ و گستره^۳ تعریف می‌شود.
- «کار» کل زمانی است که طول می‌کشد تا کل محاسبه بر روی تنها یک پردازنده اجرا شود. به بیان دیگر، کار برابر است با مجموع زمان صرف شده توسط هر بلوک. اگر هر بلوک یک واحد زمان طول بکشد، آنگاه کار برابر با تعداد رأس‌های گراف خواهد بود.
- اما «گستره»، سریع‌ترین زمان ممکن برای اجرای محاسبه بر روی تعداد نامحدودی پردازنده است، که متناظر است با مجموع زمان بلوک‌های طولانی‌ترین مسیر در گراف توازنی است. در محاسبه «طولانی‌ترین» مسیر هر بلوک به اندازه‌ی زمان اجرای خودش وزن دهی می‌شود. به این مسیر، «مسیر بحرانی» گفته می‌شود و بنابراین گستره برابر است با وزن مسیر بحرانی^۴ در گراف توازنی.

¹ work/span analysis

² work

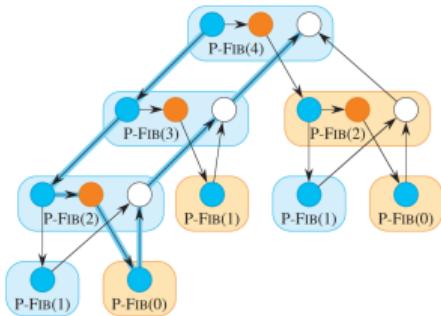
³ span

⁴ critical path

مبانی موازی انشعاب-الحاق

معیارهای کارایی

- برای یک گراف توازی که هر بلوک آن یک واحد زمان طول می‌کشد، گستره برابر است با تعداد بلوک‌ها در مسیر بحرانی.
- برای مثال در شکل زیر، گراف توازی دارای ۱۷ رأس در کل و ۸ رأس در مسیر بحرانی است؛ بنابراین اگر هر بلوک یک واحد زمان طول بکشد، کار برابر ۱۷ واحد و گستره برابر ۸ واحد زمان است.



مبانی موازی سازی انشعاب-الحاق

معیارهای کارایی

- زمان اجرای یک محاسبه‌ی به تعداد پردازنده‌های موجود و همچنین به نحوی تخصیص بلوک‌ها توسط زمان‌بند نیز وابسته است.
- برای نمایش زمان اجرای یک محاسبه‌ی وظیفه-موازی بر روی P پردازنده، از اندیس زیرنویس استفاده می‌کنیم. برای مثال، زمان اجرای یک الگوریتم روی P پردازنده را T_P می‌نویسیم.
- دیدم که «کار» همان زمان اجرا روی یک پردازنده است پس برابر است با T_1 . همچنین گستره همان زمان اجرا روی تعداد نامحدودی پردازنده است؛ یعنی اگر هر بلوک روی پردازنده‌ی خودش اجرا شود. بنابراین گستره را با T_∞ نشان می‌دهیم.

مبانی موازی سازی انشعاب-الحاق

معیارهای کارایی

- کار و گستره، کران‌های پایین زمان اجرای یک محاسبه را تعیین می‌کنند:

۱ «قانون کار»^۱: یک کامپیوتر موازی با P پردازنده در یک واحد زمان، حداکثر می‌تواند P واحد کار انجام دهد. بنابراین در زمان T_P کل کاری که می‌تواند انجام شود حداکثر PT_P است. از آنجا که کل کار برابر T_1 است، خواهیم داشت:

$$PT_P \geq T_1$$

با تقسیم بر P می‌رسیم به:

$$T_P \geq T_1/P$$

که به آن قانون کار گفته می‌شود.

^۱ Work Law

مبانی موازی سازی انشعاب-الحاق

معیارهای کارایی

۲ «قانون گستره»^۱: یک کامپیوتر موازی با P پردازنده نمی‌تواند سریع‌تر از ماشینی با تعداد نامحدود پردازنده باشد. بنابراین:

$$T_P \geq T_{\infty}$$

به نامساوی بالا قانون گستره می‌گوییم.

^۱ span Law

مبانی موازی سازی انشعاب-الحاق

معیارهای کارایی

- شتاب^۱ یک محاسبه روی P پردازنده را به این صورت تعریف می‌کنیم:

$$T_1/T_P$$

که بیان می‌کند محاسبه چند برابر سریع‌تر از حالت تک‌پردازنده اجرا می‌شود.

- با استفاده از قانون کار، چون

$$T_P \geq T_1/P$$

داریم:

$$T_1/T_P \leq P$$

بنابراین، شتاب روی یک کامپیوتر موازی با P پردازنده، حداقل برابر با P خواهد بود.

^۱ speedup

مبانی موازی سازی انشعاب-الحاق

معیارهای کارایی

- می‌گوییم محاسبه «شتاب خطی»^۱ دارد. اگر

$$T_1/T_P = \Theta(P)$$

باشد، «شتاب خطی کامل»^۲ زمانی رخ می‌دهد که:

$$T_1/T_P = P$$

^۱ linear speedup

^۲ perfect linear speedup

مبانی موازی‌سازی انشعاب-الحاق

معیارهای کارایی

- نسبت کار به گستره، «نرخ موازی‌سازی»^۱ محاسبه‌ی موازی را نشان می‌دهد:

$$T_1/T_\infty$$

این کمیت را می‌توان از سه دیدگاه بررسی کرد:

- ۱ به عنوان یک نسبت: مقدار متوسط کاری که در هر گام مسیر بحرانی می‌تواند به طور موازی انجام شود.
- ۲ به عنوان یک کران بالا: بیشینه‌ی شتابی که روی هر تعداد پردازنده می‌تواند به دست آید.
- ۳ به عنوان یک محدودیت بنیادی: اگر تعداد پردازنده‌ها بیشتر از «نرخ موازی‌سازی» شود، رسیدن به شتاب خطی کامل غیرممکن خواهد بود.

^۱ parallelism

مبانی موازی سازی انشعاب-الحاق

معیارهای کارایی

- دیدگاه سوم از بقیه مهمتر است؛ برای روشن شدن آن، فرض کنید:

$$P > T_1/T_\infty$$

در این صورت، قانون گستره نشان می‌دهد:

$$T_1/T_P \leq T_1/T_\infty < P$$

بنابراین:

$$T_1/T_P < P$$

بنابراین شتاب نمی‌تواند برابر با تعداد پردازنده‌ها باشد.

مبانی موازی سازی انشعاب-الحاق

معیارهای کارایی

- به علاوه، اگر تعداد پردازنده‌ها خیلی بیشتر از نرخ موازی سازی باشد؛ یعنی

$$P \gg T_1/T_\infty$$

آنگاه:

$$T_1/T_P \ll P$$

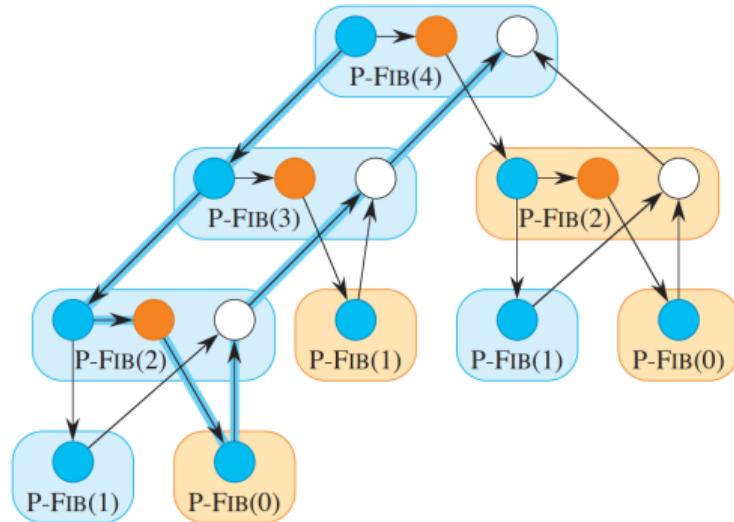
یعنی شتاب بسیار کمتر از تعداد پردازنده‌ها خواهد بود.

- به بیان دیگر، اگر تعداد پردازنده‌ها بیشتر از نرخ موازی سازی باشد، افزودن پردازنده‌ها شتاب ما را از شتاب کامل دورتر می‌کند.

مبانی موازی سازی انشعاب-الحاق

معیارهای کارایی

- برای مثال بار دیگر شکلی که پیشتر دیدیم را در نظر بگیرید و فرض کنید هر بلوک یک واحد زمان طول می‌کشد.



مبانی موازی‌سازی انشعاب-الحاق

معیارهای کارایی

- دیدم که $T_1 = 17$ و $T_\infty = 8$ بنابراین نرخ موازی‌سازی برابر است با:

$$T_1/T_\infty = 17/8 = 2.125$$

- در نتیجه، فارغ از اینکه چند پردازنده استفاده شود، دستیابی به بیش از دو برابر کارایی (شتاپ) غیرممکن است.
- دقت کنید که این تنها برای $P - FIB(4)$ بود. اما برای ورودی‌های بزرگ‌تر، نرخ موازی‌سازی بالایی خواهیم داشت.

مبانی موازی سازی انشعاب-الحاق

معیارهای کارایی

- رابطه‌ی میان موازی سازی یک محاسبه و تعداد پردازنده‌های موجود، مفهوم مهمی به نام انعطاف‌پذیری^۱ را معرفی می‌کند.
- انعطاف‌پذیری یک محاسبه روی تعداد P پردازنده، برابر است با:

$$\frac{T_1/T_\infty}{P} = \frac{T_1}{PT_\infty}$$

که نشان می‌دهد موازی سازی محاسبه تا چه اندازه از تعداد پردازنده‌های موجود در ماشین بیشتر است.

¹ slackness

مبانی موازی سازی انشعاب-الحاق

معیارهای کارایی

- اگر مقدار انعطاف‌پذیری کمتر از ۱ باشد، شتاب خطی کامل غیرممکن است، زیرا

$$\frac{T_1}{PT_\infty} < 1$$

و طبق قانون گستره:

$$\frac{T_1}{T_P} \leq \frac{T_1}{T_\infty} < P.$$

- در واقع، هرچه مقدار انعطاف‌پذیری از ۱ کمتر شود و به \circ نزدیک شود، شتاب بیشتر و بیشتر از شتاب خطی کامل فاصله می‌گیرد. در این حالت، افزودن موازی سازی به الگوریتم می‌تواند تأثیر زیادی بر کارایی آن داشته باشد.

مبانی موازی‌سازی انشعاب-الحاق

معیارهای کارایی

- اگر مقدار انعطاف‌پذیری بیشتر از ۱ باشد، در این صورت «کار بهازای هر پردازنده» به محدودکننده‌ی اصلی تبدیل می‌شود. در این حالت، یک زمان‌بند خوب می‌تواند شتابی نزدیک‌تر و نزدیک‌تر به شتاب خطی کامل نتیجه دهد.
- وقتی مقدار انعطاف‌پذیری خیلی بیشتر از ۱ شود، اضافه کردن پردازنده‌های بیشتر، دیگر فایده‌ی چندانی ندارد و سود ناشی از موازی‌سازی به تدریج کمتر و کمتر می‌شود.

مبانی موازی سازی انشعباب-الحاق

زمان بندی

- عملکرد خوب تنها به کمینه سازی کار و گستره وابسته نیست. باید بلوک ها نیز به طور کارآمد روی پردازنده های ماشین موازی زمان بندی شوند.
- در مدل برنامه نویسی انشعباب-الحاق نمی توان مشخص کرد کدام بلوک روی کدام پردازنده اجرا شود. در عوض، ما به زمان بند تکیه می کنیم تا محاسبات را به صورت پویا را به پردازنده ها نگاشت کند.
- در عمل، زمان بند بلوک ها را به نخ ها نگاشت می کند و سپس سیستم عامل این نخ ها را روی پردازنده ها زمان بندی می کند. اما ما می توانیم به سادگی تصور کنیم که زمان بند بلوک ها را مستقیماً به پردازنده ها نگاشت می کند.
- یک زمان بند باید «در لحظه»^۱ عمل کند. یعنی بدون اینکه از قبل بداند چه زمانی فرآخونی های موازی ایجاد می شوند یا چه زمانی پایان می یابند، محاسبه را زمان بندی کند.

¹ online

مبانی موازی سازی انشعاب-الحاق

زمان بندی

- به طور خاص، ما زمان بندهای حریصانه^۱ را تحلیل می کنیم که در هر گام بیشترین تعداد بلوک را به پردازندهها اختصاص می دهند و سعی می کنند هیچ پردازنده ای را بیکار نگذارند. در یک زمان بند حریصانه برای هر گام دو حالت وجود دارد:

۱ گام کامل^۲ : دست کم P بلوک، آماده ای اجرا هستند. یک زمان بند حریصانه هر P بلوک آماده را به پردازندهها تخصیص می دهد.

۲ گام ناقص^۳ : کمتر از P بلوک آماده ای اجرا هستند. زمان بند حریصانه هر بلوک آماده را به پردازندهی خودش تخصیص می دهد، تعدادی پردازنده بیکار باقی می مانند، اما همهی بلوک های آماده اجرا می شوند.

- منظور از بلوک آماده بلوکی است که همه بلوک هایی که به آنها وابسته است اجرا شده باشند.

¹ greedy schedulers

² Complete step

³ Incomplete step

مبانی موازی سازی انشعاب-الحاق

زمان بندی

- دیدیم که طبق قانون کار، سریع‌ترین زمان اجرای یک محاسبه روی P پردازنده دست‌کم برابر است با:

$$T_P \geq T_1/P$$

از طرف دیگر، قانون گستره بیان می‌کند که سریع‌ترین زمان ممکن دست‌کم برابر است با:

$$T_P \geq T_\infty$$

- در ادامه قضیه‌ایی را بررسی می‌کنیم که نشان می‌دهد زمان اجرای یک محاسبه با زمان بند حریصانه کران بالایی برابر با مجموع این دو کران پایین دارد. این کران بالا خوب است زیرا تا حد قابل قبولی به کران‌های پایین نزدیک است.

مبانی موازی انشعاب-الحاق

زمانبندی

- می خواهیم ثابت کنیم زمان اجرای یک محاسبه با زمانبند حریصانه حداکثر به این مقدار زمان نیاز دارد:

$$T_P \leq T_1/P + T_\infty$$

- اثبات: فرض کنید هر بلوک یک واحد زمان طول می کشد. (هر بلوک طولانی تر را می توان با زنجیرهای از بلوک های تک واحدی جایگزین کرد.) حال گام های کامل و ناقص را جداگانه بررسی می کنیم:

الف در هر گام کامل، P پردازنده روی هم واحد کار انجام می دهند. بنابراین، اگر تعداد گام های کامل k باشد، کل کاری که در این گام ها انجام می شود kP است. این مقدار از تعداد کل کارها کمتر است پس $T_1 \leq kP$ از اینجا نتیجه می گیریم که تعداد گام های کامل حداکثر

$$T_1/P$$

است.

مبانی موازی سازی انشعاب-الحاق

زمانبندی

ب) حال گام ناقص را در نظر بگیریم. فرض کنید G گراف توازی کل محاسبه باشد. در ابتدای گام ناقص، زیرگراف توازی G' هنوز اجرا نشده است و پس از این گام زیرگراف توازی G'' باقی می‌ماند.

- همچنین تعریف می‌کنیم مجموعه R هم بلوک‌هایی هستند که در آغاز این گام ناقص آماده‌اند.
- یک طولانی‌ترین مسیر در G' باید با بلوک‌ای از R شروع شود. فرض کنید یک طولانی‌ترین مسیر در G' از بلوکی خارج از R شروع شود، می‌دانیم این بلوک آماده نیست و گراف بدون دور است پس می‌توانیم پیش‌نیاز این بلوک را به مسیر اضافه کنیم و در آینه به تناقض می‌رسیم.

مبانی موازی انشعاب-الحاق

زمانبندی

- چون زمانبند حریصانه همهی بلوک‌های آماده را در این گام اجرا می‌کند، G'' برابر است با G' منهای R .
- بنابراین طولانی‌ترین مسیر در G'' دقیقاً یک واحد کوتاه‌تر از طولانی‌ترین مسیر در G' است. در نتیجه، هر گام ناقص طول مسیر بحرانی زیر گراف باقی‌مانده را یک واحد کاهش می‌دهد.
- می‌دانیم که طول مسیر بحرانی یا همان گستره برابر است با T_∞ بنابراین تعداد گام‌های ناقص حداقل T_∞ است.

مبانی موازی انشعاب-الحاق

زمانبندی

- دیدیم که حداکثر تعداد گام‌های کامل T_1/P است و حداکثر تعداد گام‌های ناقص T_∞ است بنابراین حداکثر تعداد مجموع گام‌ها

$$T_\infty + T_1/P$$

است. و بدین ترتیب قضیه ثابت می‌شود.

- با استفاده از این قضیه ثابت می‌شود که اگر

$$P \gg T_1/T_\infty$$

باشد، (یعنی انعطاف‌پذیری خیلی بیشتر از ۱ باشد) آنگاه، شتاب تقریباً برابر با P است (شتاب نزدیک به کامل).

مبانی موازی‌سازی انشعاب-الحاقد

تحلیل الگوریتم‌های موازی

- اکنون همه‌ی ابزارهای لازم را برای تحلیل الگوریتم‌های موازی با استفاده از تحلیل کار-گستره را در اختیار داریم.
- تحلیل «کار» ساده است. کافیست تصویر سریال الگوریتم موازی را به دست آورده و با همان روش تحلیل زمانی الگوریتم‌های سری آن را تحلیل کنیم.
- آنچه جدید است، تحلیل گستره است. اما خواهید دید که چندان پیچیدگی بیشتری ندارد.

مبانی موازی سازی انشعاب-الحاق

تحلیل الگوریتم‌های موازی

- بیایید دوباره به مثال P-FIB بازگردیم و با این مثال با بعضی از ایده‌های ابتدایی آشنا شویم.
- دیدم که برای تحلیل کار باید تصویر سری الگوریتم P-FIB را به دست آورده و آن را تحلیل زمانی عادی کنیم. تصویر سری این الگوریتم همان الگوریتم FIB است که پیشتر آن را تحلیلی کرده‌ایم بنابراین می‌دانیم:

$$T_1(n) = T(n) = \Theta(\phi^n)$$

مبانی موازی سازی انشعاب-الحاق

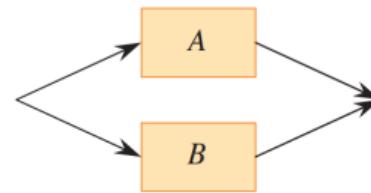
تحلیل الگوریتم‌های موازی

- حال با استفاده از شکل زیر نشان خواهیم داد که چگونه باید گستره را تحلیل کنیم.



Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
Span: $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$

(a)



Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
Span: $T_\infty(A \cup B) = \max(T_\infty(A), T_\infty(B))$

(b)

مبانی موازی سازی انشعاب-الحاق

تحلیل الگوریتم‌های موازی

- ۱ اگر دو محاسبه به صورت سری ترکیب شوند، گستره‌ی کل برابر است با جمع گستره‌ی دو محاسبه.
- ۲ اما اگر به صورت موازی ترکیب شوند، گستره‌ی کل برابر است با بیشینه‌ی گستره‌های دو محاسبه.
 - در واقع، گراف توافقی هر محاسبه‌ای را می‌توان با ترکیب موازی یا سری تعدادی بلوک ساخت.
 - حال که ترکیب سری و موازی را آموختیم، می‌توانیم گستره‌ی $FIB(n) - P$ را تحلیل کنیم.

مبانی موازی سازی انشعاب-الحاق

تحلیل الگوریتم‌های موازی

- فراخوان $P - FIB(n - 1)$ به صورت موازی با فراخوانی $P - FIB(n - 2)$ اجرا می‌شود. بنابراین، گستره‌ی $P - FIB(n)$ را می‌توان به صورت رابطه‌ی بازگشتی زیر نوشت:

$$\begin{aligned}T_{\infty}(n) &= \max\{T_{\infty}(n - 1), T_{\infty}(n - 2)\} + \Theta(1) \\&= T_{\infty}(n - 1) + \Theta(1),\end{aligned}$$

- که جواب آن برابر است با:

$$T_{\infty}(n) = \Theta(n)$$

مبانی موازی‌سازی انشعاب-الحاق

تحلیل الگوریتم‌های موازی

- پس نرخ موازی‌سازی در $P\text{-FIB}(n)$ برابر است با:

$$T_1(n)/T_\infty(n) = \Omega(\phi^n/n)$$

که با افزایش n به سرعت رشد می‌کند.

مبانی موازی‌سازی انشعاب-الحاق

حلقه‌های موازی

- بسیاری از الگوریتم‌ها شامل حلقه‌هایی هستند که تمام تکرارهای آن‌ها می‌توانند به صورت موازی اجرا شوند.
- هرچند می‌توان از کلیدواژه‌های `spawn` و `sync` برای موازی‌سازی چنین حلقه‌هایی استفاده کرد، ولی راحت‌تر است که کلید واژه‌ی جداگانه‌ایی داشته باشیم که مشخص کند یک حلقه می‌تواند موازی اجرا شود. این قابلیت از طریق کلیدواژه‌ی `parallel` فراهم می‌شود که قبل از کلیدواژه‌ی `for` قرار می‌گیرد.

مبانی موازی سازی انشعاب-الحاق

حلقه‌های موازی

- به عنوان مثال، رویه‌ی P-MAT-VEC ضرب یک ماتریس مربعی در بردار را به صورت موازی انجام می‌دهد. کلیدواژه‌های parallel قبل از for نشان می‌دهند که هر تکرار از بدنه‌ی حلقه، می‌توانند به صورت موازی اجرا شوند.

Algorithm P-MAT-VEC

```
function P-MAT-VEC( $A, x, y, n$ )
1: parallel for  $i = 1$  to  $n$  do                                // parallel loop
2:   for  $j = 1$  to  $n$  do                                    // serial loop
3:      $y_i \leftarrow y_i + a_{ij} x_j$ 
```

مبانی موازی سازی انشعاب-الحاق

حلقه‌های موازی

- کامپایلرها برای برنامه‌های موازی می‌توانند حلقه‌های `fork-join` را با استفاده از `spawn` و `sync` پیاده‌سازی کنند. اینکار به شکل بازگشته انجام می‌شود.
- به عنوان مثال، برای حلقه‌ی موازی بالا، یک کامپایلر می‌تواند تابع کمکی `P-MAT-VEC-RECURSIVE` را تولید کرده و سپس حلقه `for` را با فراخوانی

`P-MAT-VEC-RECURSIVE($A, x, y; n, , 1, n$)`

جایگزین کند.

- این روال بازگشتی نصف اول تکرارهای حلقه را با نصف دوم آن، موازی اجرا کرده و سپس یک sync اجرا می‌کند. بدین ترتیب یک درخت دودویی از اجرای موازی ساخته می‌شود. خطوط ۲-۳ نیز حالت پایه می‌باشد.

Algorithm P-MAT-VEC-RECURSIVE

```

function P-MAT-VEC-RECURSIVE( $A, x, y, n, i, i'$ )
1: if  $i = i'$  then                                // just one iteration to do
2:   for  $j = 1$  to  $n$  do                         // mimic P-MAT-VEC serial loop
3:      $y_i \leftarrow y_i + a_{ij} x_j$ 
4: else
5:    $mid \leftarrow \lfloor (i + i')/2 \rfloor$            // parallel divide-and-conquer
6:   spawn P-MAT-VEC-RECURSIVE( $A, x, y, n, i, mid$ )
7:   P-MAT-VEC-RECURSIVE( $A, x, y, n, mid + 1, i_0$ )
8:   sync

```

مبانی موازی‌سازی انشعاب-الحاق

حلقه‌های موازی

- برای محاسبه‌ی کار P-MAT-VEC کافی است زمان اجرای نسخه‌ی ترتیبی آن را حساب کنیم؛ یعنی حالتی که حلقه‌ی for parallel در خط ۱ با یک حلقه‌ی معمولی for جایگزین شود. بنابراین داریم:

$$T_1(n) = \Theta(n^2)$$

- این تحلیل به ظاهر سربار ناشی از فراخوانی بازگشتی برای پیاده‌سازی حلقه‌های موازی را نادیده می‌گیرد. درست است که این فراخوانی‌ها سربار دارد، اما از نظر مجانبی تاثیری روی زمان اجرا نمی‌گذارد.

مبانی موازی سازی انشعاب-الحاق

حلقه‌های موازی

- در هنگام تحلیل کار دیدم که می‌توانیم از سربار فراخوانی‌های بازگشتی صرف نظر کنیم. ولی هنگام تحلیل «گستره» باید آن را لحاظ کرد.
- به طور کلی اگر یک حلقه موازی با n تکرار داشته باشیم و گستره تکرار امّا آن را $(i)_{\infty}^{\infty}$ iter بنامیم، گستره حلقه برابر خواهد بود با:

$$T_{\infty}(n) = \Theta(\lg n) + \max\{iter_{\infty}(i) \mid 1 \leq i \leq n\}$$

- برای فهم بهتر تساوی بالا، توجه کنید که گه عمق درخت بازگشت لگاریتمی است. حال باید تاثیر عملیاتی که درون حلقه انجام می‌شود را لحاظ کنیم که همان عبارت دوم است. در حالت سری ما تاثیر حلقه دوم را در تکرار حلقه اول ضرب می‌کردیم. در اینجا دقت کنید که به خاطر موازی بودن تکرارهای حلقه، جمع صورت گرفته.

مبانی موازی سازی انشعاب-الحاق

حلقه‌های موازی

- حال که تحلیل حلقه‌های موازی را در حالت کلی آموختیم بباید به تحلیل الگوریتم P-MAT-VEC بازگردیم.
- گستره حلقه‌ی بیرونی بدون در نظر گرفتن بدنه حلقه برابر است با:

$$\Theta(\lg n)$$

- در هر تکرار از حلقه‌ی بیرونی، حلقه‌ی درونی شامل n تکرار است و در همه تکرارهای حلقه‌ی بیرونی، گستره حلقه درونی برابر است. پس نتیجه گرفتن بیشینه گستره حلقه درونی، روی تمام تکرارهای حلقه‌ی بیرونی برابر است با:

$$\Theta(n)$$

مبانی موازی‌سازی انشعاب-الحاق

حلقه‌های موازی

- بنابراین، گستره‌ی کلی این تابع برابر است با:

$$T_\infty(n) = \Theta(n) + \Theta(\lg n) = \Theta(n)$$

- دیدم که کار برابر $\Theta(n^2)$ است پس نرخ موازی‌سازی برابر خواهد بود با:

$$\Theta(n^2)/\Theta(n) = \Theta(n).$$

مبانی موازی سازی انشعاب-الحاق

وضعیت رقابت

- یک الگوریتم موازی را قطعی^۱ می‌نامیم اگر همواره روی یک ورودی مشخص، یک خروجی یکسان تولید کند.
اگر خروجی الگوریتم بتواند بین اجراهای مختلف روی یک ورودی تغییر کند، الگوریتم غیرقطعی^۲ است.
- زمانی که یک الگوریتم را به صورت موازی پیاده‌سازی می‌کنیم ممکن است خطایی به نام «عدم قطعیت رقابتی»^۳ رخ دهد که یکی از انواع «وضعیت رقابت»^۴ است.
- خطای عدم قطعیت رقابتی می‌تواند باعث شود یک الگوریتم موازی که هدف آن قطعی بودن است، به طور غیرقطعی عمل کند. تشخیص این خطا می‌تواند بسیار سخت باشد.

^۱ deterministic

^۲ nondeterministic

^۳ determinacy race

^۴ Race condition

مبانی موازی سازی انشعاب-الحاق

وضعیت رقابت

- خطاهای مشهور زیر در طول تاریخ به از نوع «عدم قطعیت رقابتی» بوده اند:
 - ١ خطای دستگاه پرتو درمانی Therac-۲۵ که سه نفر را کشت و چندین نفر دیگر را مجروح کرد.
 - ٢ خاموشی بزرگ شمال شرق آمریکا در سال ۲۰۰۳ که باعث قطعی برق بیش از ۵۰ میلیون نفر شد.
- ممکن است نرم افزار شما روزها در آزمایشگاه بدون خطا اجرا شود، ولی سپس به طور پراکنده در محیط واقعی دچار کرش های جدی گردد.

مبانی موازی سازی انشعاب-الحاق

وضعیت رقابت

- عدم قطعیت رقابتی هنگامی رخ می‌دهد که دو دستور موازی به یک مکان حافظه مشترک دسترسی یابند و دست کم یکی از آن‌ها مقدار ذخیره شده را تغییر دهد. تابع ساده‌ی زیر این وضعیت را نشان می‌دهد.

Algorithm RACE-EXAMPLE

```
function RACE-EXAMPLE()
1:  $x \leftarrow 0$ 
2: parallel for  $i = 1$  to  $2$  do
3:    $x \leftarrow x + 1$                                 // determinacy race
4: print( $x$ )
```

مبانی موازی انشعاب-الحاق

وضعیت رقابت

- تابع RACE-EXAMPLE دو بلوک موازی ایجاد می‌کند که هر یک مقدار x را یک واحد افزایش می‌دهند.
هرچند انتظار داریم این روال مانند نسخه ترتیبی، همواره مقدار ۲ را چاپ کند ولی ممکن است مقدار ۱ چاپ شود. زیرا عمل افزایش مقدار x در کد سطح پایین‌تر (asmبلی) در واقع از چند دستور تشکیل شده است:
 - بارگذاری x از حافظه به یکی از ثبات‌های پردازنده.
 - افزایش مقدار در ثبات.
 - ذخیره‌سازی مقدار جدید ثبات در حافظه.
- اگر دستورات دو پردازنده به ترتیب خاصی «درهم‌تنیده»^۱ شوند، تاثیر یکی از بهروزرسانی‌ها ختنشی می‌شود و در نهایت مقدار ۱ چاپ خواهد شد.

¹ interleaved

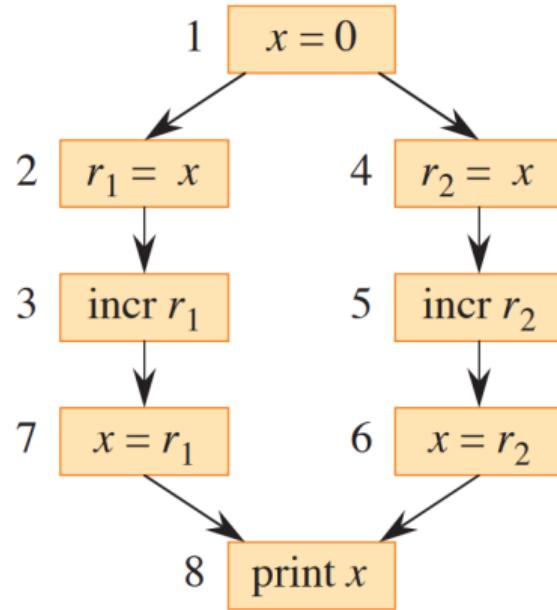
مبانی موازی سازی انشعاب-الحاق وضعیت رقابت

- توجه کنید که حتی با وجود خطای عدم قطعیت رقابتی، باز هم بسیاری از اجراهای منجر به نتیجه‌ی درست می‌شوند. اما برخی اجراهای به‌طور غیرقابل پیش‌بینی نتیجه‌ی غلط تولید می‌کنند. همین غیرقابل تکرار بودن، یافتن خطاهای رقابتی را به‌شدت دشوار می‌سازد.

مبانی موازی سازی انشعاب-الحاق

وضعیت رقابت

- شکل زیر گراف توازی الگوریتم RACE-EXAMPLE را نمایش می‌دهد، با این تفاوت که بلوک‌ها تا سطح دستورات سطح پایین شکسته شده‌اند.



مبانی موازی سازی انشعاب-الحاق

وضعیت رقابت

- گفتیم که یک طبق فرض این بحث، کامپیوتر ما از سازگاری ترتیبی^۱ پشتیبانی می‌کند. بنابراین می‌توانید اجرای موازی دو بلوک را مانند یک توالی خطی از دستورات هر دو بلوک درنظر بگیرید.
- به این توالی خطی درهم‌تنیدگی^۲ گفته می‌شود زیرا چند دستور موازی می‌توانند در هم تنیده شوند بدین معنی که دستورات سطح پایین می‌توانند در میان یکدیگر اجرا شوند. البته بهگونه‌ای که وابستگی‌های موجود در گراف موازی حفظ شوند.

^۱ sequential consistency

^۲ interleaving

مبانی موازی سازی انشعاب-الحاق

وضعیت رقابت

- شکل زیر، یک ترتیب خاص از اجرای دستورات را نشان می‌دهد که باعث ایجاد این مشکل می‌شود.

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

مبانی موازی انشعاب-الحاق

وضعیت رقابت

- هر چند با روش‌هایی مانند انحصار متقابل^۱، می‌توان از این خطا جلوگیری کرد. اما ما در این بحث به طور کلی وجود وضعیت رقابت را مجاز نمی‌دانیم و می‌خواهیم ببینم چطور می‌توان از بروز این وضعیت جلوگیری کرد.
- برای اطمینان از قطعی بودن الگوریتم‌ها، هر دو رشته‌ای که به صورت موازی عمل می‌کنند باید بدون تداخل متقابل^۲ باشند. یعنی هر دو تنها بخوانند و هیچ مقدار مشترکی را تغییر ندهند.
- بنابراین در یک ساختار parallel for باید تمام تکرارهای بدنه حلقه، بدون تداخل متقابل باشند. همچنین بین یک sync و spawn متناظر، نباید تداخل متقابل وجود داشته باشد.

^۱ mutual-exclusion locks

^۲ mutually noninterfering

- باید یک مثال ساده از پیاده‌سازی اشتباه تابع P-MAT-VEC ببینیم. تابع زیر را در نظر بگیرید. این روال حلقه‌ی درونی را نیز موازی کرده و گستره¹ (lgn) Θ را به دست می‌آورد، اما به دلیل وضعیت رقابت در خط ۳، خروجی نادرست تولید می‌کند:

Algorithm P-MAT-VEC-WRONG

```

function P-MAT-VEC-WRONG( $A, x, y, n$ )
1: parallel for  $i = 1$  to  $n$  do
2:   parallel for  $j = 1$  to  $n$  do
3:      $y_i \leftarrow y_i + a_{ij} x_j$                                 // determinacy race

```

- توجه کنید در اینجا متغیرهای اندیس¹ i و j مشکلی ایجاد نمی‌کنند. زیرا هر تکرار عملأً نسخه‌ی مستقل خودش از متغیر اندیس را ایجاد می‌کند.

¹ index

مبانی موازی‌سازی انشعاب-الحاقد

مثالی از تحلیل کار-گستره

- رای نشان دادن قدرت تحلیل کار-گستره، یک داستان واقعی را بررسی خواهیم کرد که چندین سال پیش در جریان توسعه یکی از اولین برنامه‌های شطرنج موازی سطح جهانی رخ داده است.
- این برنامه شطرنج روی یک رایانه 32×32 هسته‌ایی توسعه و آزمایش شد، اما قرار بود روی یک ابررایانه با 512×512 هسته اجرا شود. از آنجا که دسترسی به ابررایانه محدود و پرهزینه بود، توسعه‌دهندگان آزمایش‌ها را روی رایانه کوچک اجرا کرده و عملکرد را برای رایانه بزرگ برآورد می‌کردند.
- در مقطعی، توسعه‌دهندگان یک بهینه‌سازی به برنامه اضافه کردند که زمان اجرای برنامه را روی رایانه کوچک از 65 ثانیه به $T'_{32} = 40$ کاهش داد. با این حال، با استفاده از تحلیل کار و گستره، نتیجه گرفتند که نسخه بهینه‌شده که روی 32×32 پردازنده سریع‌تر بود، روی 512×512 پردازنده از نسخه اصلی کنتر خواهد بود. بنابراین، آنها از اجرای این بهینه‌سازی صرف نظر کردند.

مبانی موازی‌سازی انشعاب-الحاقد

مثالی از تحلیلی کار-گستره

- تحلیل کار-گستره آنها به این صورت بود: نسخه اصلی برنامه دارای $T_1 = 2048$ و $T_\infty = 1$ ثانیه بود.
- نابرابری زیر را قبلًا در هنگام معرفی زمانبند دیده‌ایم:

$$T_P \leq \frac{T_1}{P} + T_\infty$$

- اگر این نابرابری را به صورت برابری در نظر بگیریم می‌توانیم با جایگذاری کار و گستره، زمان اجرا روی P پردازنده را تخمین بزنیم (در واقع یک کران برای آن می‌یابیم)

مبانی موازی‌سازی انشعاب-الحاق

مثالی از تحلیلی کار-گستره

- پس باید زمان حالت اولیه روی کامپیوتر ۳۲ هسته‌ایی را تخمین بزنیم:

$$T_{32} = \frac{2048}{32} + 1 = 65$$

- حال به انجام بهینه‌سازی، کار به $T'_\infty = 8$ ثانیه تغییر می‌یابد. پس داریم:

$$T'_{32} = \frac{1024}{32} + 8 = 40$$

مبانی موازی‌سازی انشعاب-الحاق

مثالی از تحلیلی کار-گستره

- بنابراین روی کامپیوتر ۳۲ هسته‌ایی بهبود حاصل شد. اما نسبت سرعت دو نسخه هنگامی که زمان اجرای آنها روی ۵۱۲ پردازنده برآورده می‌شود، معکوس می‌شود. نسخه اول دارای زمان اجرای

$$T_{512} = \frac{2048}{512} + 1 = 5$$

و نسخه دوم دارای زمان اجرای

$$T'_{512} = \frac{1024}{512} + 8 = 10$$

ثانیه است.

مبانی موازی سازی انشعاب-الحاقد

مثالی از تحلیلی کار-گستره

- بهینه‌سازی ایی که برنامه را روی ۳۲ پردازنده سریع‌تر می‌کرد، باعث شد برنامه روی ۵۱۲ پردازنده دو برابر طولانی‌تر اجرا شود!
- گستره نسخه بهینه‌شده برابر با ۸ بود که در زمان اجرای روی ۳۲ پردازنده نقش غالب نداشت، اما روی ۵۱۲ پردازنده نقش غالب پیدا می‌کند و مزیت استفاده از پردازنده‌های بیشتر را از بین می‌برد. اصطلاحاً می‌گوییم این بهینه‌سازی مقیاس‌پذیری نیست.
- بنابراین در هنگام برآوردن مقیاس‌پذیری، تحلیل کار-گستره نسبت به اندازه‌گیری زمان اجرا برتری دارد.

ضرب موازی ماتریس‌ها

- در این بخش، بررسی می‌کنیم که چگونه می‌توان الگوریتم‌های ضرب ماتریس را موازی‌سازی کرد. خواهیم دید که هر الگوریتم را می‌توان به سادگی با استفاده از کلیدواژه‌هایی که آموختیم موازی‌سازی نمود.
- این الگوریتم‌ها را با استفاده از کار-گستره تحلیل خواهیم کرد و مشاهده می‌کنیم که الگوریتم‌های موازی همان کارایی الگوریتم سری متناظر خود را روی یک پردازنده به دست می‌آورند، در حالی که روی تعداد زیادی پردازنده نیز مقیاس‌پذیر هستند.

ضرب موازی ماتریس‌ها

موازی‌سازی روش حلقه‌های تودرتو

- اولین الگوریتمی که بررسی می‌کنیم P-MATRIX-MULTIPLY است که به‌سادگی دو حلقة بیرونی در ضرب ماتریسی عادی را موازی‌سازی می‌کند.

Algorithm P-MATRIX-MULTIPLY

```
function P-MATRIX-MULTIPLY( $A, B, C, n$ )
1: parallel for  $i = 1$  to  $n$  do           // compute entries in each of  $n$  rows
2:   parallel for  $j = 1$  to  $n$  do           // compute  $n$  entries in row  $i$ 
3:     for  $k = 1$  to  $n$  do
4:        $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$ 
```

ضرب موازی ماتریس‌ها

موازی‌سازی روش حلقه‌های تودرتو

- اکنون این الگوریتم را تحلیل می‌کنیم. از تحلیل تصویر سری این الگوریتم در می‌یابیم کار برابر با است با:

$$T_1(n) = \Theta(n^3)$$

- زیرا مسیر بحرانی شامل حرکت بازگشته ناشی از حلقةٌ موازی خط ۱، سپس درخت بازگشته ناشی از حلقةٌ موازی خط ۲، و سپس اجرای تمام n تکرار حلقةٌ معمولی خط ۳ است. در نتیجه گستره برابر می‌شود با:

$$T_\infty(n) = \Theta(\lg n) + \Theta(\lg n) + \Theta(n) = \Theta(n)$$

بنابراین، نرخ موازی‌سازی برابر است با:

$$\Theta(n^3)/\Theta(n) = \Theta(n^2).$$

ضرب موازی ماتریس‌ها

موازی‌سازی روش تقسیم و حل

- احتمالاً با انجام ضرب ماتریسی با استفاده از تقسیم و حل آشنایی داشته باشید. در این قسمت می‌خواهیم همان روش را موازی‌سازی کنیم.
- این روش دو ماتریس $n \times n$ را در زمان $\Theta(n^3)$ ضرب می‌کند.
- بباید ببینیم چگونه می‌توان این الگوریتم را با استفاده از فراخوانی‌ها موازی بازگشتی پیاده سازی کرد.

ضرب موازی ماتریس‌ها

موازی‌سازی روش تقسیم و حل

- می‌دانیم که تابع سریالی ضرب ماتریس به روش تقسیم و حل، سه ماتریس $n \times n$ دریافت می‌کند و سپس محاسبه ماتریسی زیر را انجام می‌دهد: (فرض کنید سه ماتریسی را $C B A$ بنامیم)

$$C = C + A \cdot B$$

این کار با انجام هشت ضرب از زیر ماتریس‌های $\frac{n}{2} \times \frac{n}{2}$ از A و B به صورت بازگشته انجام می‌شود.

- تابع P-MATRIX-MULTIPLY-RECURSIVE در اسلاید بعدی همین راهبرد تقسیم و حل را پیاده‌سازی می‌کند، اما برای انجام این هشت ضرب از فراخوانی موازی استفاده می‌کند. برای اجتناب از بروز عدم قطعیت روابطی هنگام به روزرسانی عناصر، یک ماتریس موقتی D ساخته می‌شود و سپس C و D جمع می‌شوند تا نتیجهٔ نهایی به دست آید.

ضرب موازی ماتریس‌ها

موازی‌سازی روش تقسیم و حل

- الگوریتم ضرب ماتریسی موازی به روش تقسیم و حل به این صورت است:

Algorithm P-MATRIX-MULTIPLY-RECURSIVE

```
function P-MATRIX-MULTIPLY-RECURSIVE( $A, B, C, n$ )
1: if  $n = 1$  then // base case
2:    $c_{11} \leftarrow c_{11} + a_{11} \cdot b_{11}$ 
3:   return
4: let  $D$  be a new  $n \times n$  matrix // temporary matrix
5: parallel for  $i = 1$  to  $n$  do
6:   parallel for  $j = 1$  to  $n$  do
7:      $d_{ij} \leftarrow 0$ 
8: partition  $A, B, C, D$  into  $\frac{n}{2} \times \frac{n}{2}$  submatrices:
9:    $A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22},$ 
10:   $C_{11}, C_{12}, C_{21}, C_{22}$ , and  $D_{11}, D_{12}, D_{21}, D_{22}$ 
```

ضرب موازی ماتریس‌ها

موازی‌سازی روش تقسیم و حل

- ادامه تابع ضرب ماتریسی موازی به روش تقسیم و حل...

```
11: spawn P-MATRIX-MULTIPLY-RECURSIVE( $A_{11}, B_{11}, C_{11}, n/2$ )
12: spawn P-MATRIX-MULTIPLY-RECURSIVE( $A_{11}, B_{12}, C_{12}, n/2$ )
13: spawn P-MATRIX-MULTIPLY-RECURSIVE( $A_{21}, B_{11}, C_{21}, n/2$ )
14: spawn P-MATRIX-MULTIPLY-RECURSIVE( $A_{21}, B_{12}, C_{22}, n/2$ )
15: spawn P-MATRIX-MULTIPLY-RECURSIVE( $A_{12}, B_{21}, D_{11}, n/2$ )
16: spawn P-MATRIX-MULTIPLY-RECURSIVE( $A_{12}, B_{22}, D_{12}, n/2$ )
17: spawn P-MATRIX-MULTIPLY-RECURSIVE( $A_{22}, B_{21}, D_{21}, n/2$ )
18: spawn P-MATRIX-MULTIPLY-RECURSIVE( $A_{22}, B_{22}, D_{22}, n/2$ )
19: sync                                // wait for spawned submatrix products
20: for parallel i = 1 to n do           // update  $C = C + D$ 
21:   for parallel j = 1 to n do
22:      $c_{ij} \leftarrow c_{ij} + d_{ij}$ 
```

ضرب موازی ماتریس‌ها

موازی‌سازی روش تقسیم و حل

- خطوط ۱ تا ۳ حالت پایه را که مربوط به ضرب ماتریس‌های 1×1 است، مدیریت می‌کنند و بخش باقی‌مانده با حالت بازگشته سروکار دارد. خط ۸ هر یک از چهار ماتریس A, B, C, D را به زیرماتریس‌های $\frac{n}{2} \times \frac{n}{2}$ تقسیم می‌کند.
- در خطوط ۱۱ تا ۱۸ هشت ضرب ماترسی مذکور به شکل موازی انجام شده و در نتیجه در ماتریس‌ها C و D قرار می‌گیرد.
- در خط ۱۹ دستور `sync` تضمین می‌کند که همه ضرب‌ها محاسبه شده باشند، و پس از آن در خطوط ۲۰ تا ۲۲ ماتریس C و D جمع می‌شوند.

ضرب موازی ماتریس‌ها

موازی‌سازی روش تقسیم و حل

- باید رویه P-MATRIX-MULTIPLY-RECURSIVE تحلیل می‌کنیم.
- می‌دانیم باید قسمتی که هشت ضرب بازگشته را انجام می‌دهد از بقیه قسمت‌ها پیچیدگی زمانی بالاتری دارد و پیچیدگی زمانی آن برابر است با:

$$M_1(n) = 8M_1(n/2) + \Theta(n^2) = \Theta(n^3)$$

که با حالت ۱ قضیه اصلی^۱ به دست می‌آید.

^۱ master theorem

ضرب موازی ماتریس‌ها

موازی‌سازی روش تقسیم و حل

- اکنون باید طول مسیر بحرانی یا همان $M_\infty(n)$ را تعیین کنیم. از آنجا که هشت فراخوانی بازگشتهای موازی همگی روی ماتریس‌هایی با اندازه یکسان اجرا می‌شوند، گستره همه فراخوانی‌های بازگشتهای با هم برابر است و بنابراین بیشینه گستره آنها برابر است با گستره یکی از آنها و برابر است با:

$$M_\infty(n/2)$$

- گستره برای حلقه‌های تو در توی موازی در خطوط ۵ تا ۷ برابر با $\Theta(\lg n)$ است، به‌طور مشابه، حلقه‌های تو در توی موازی در خطوط ۲۰ تا ۲۲ نیز $\Theta(\lg n)$ دیگر به گستره اضافه می‌کنند. پیچیدگی زمانی تقسیم ماتریس هم ثابت است و از آن صرف نظر می‌کنیم. بنابراین گستره این تابع به صورت زیر است:

$$M_\infty(n) = M_\infty(n/2) + \Theta(\lg n)$$

ضرب موازی ماتریس‌ها

موازی‌سازی روش تقسیم و حل

- این رابطه بازگشته تحت حالت دوم قضیه اصلی با قرار می‌گیرد و جواب آن چنین است:

$$M_\infty(n) = \Theta(\lg^2 n)$$

- پس نرخ موازی‌سازی الگوریتم P-MATRIX-MULTIPLY-RECURSIVE برابر است با:

$$M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n)$$

که بسیار بزرگ است و نشان دهنده تأثیر بسزای موازی سازی در افزایش سرعت این الگوریتم است.

تحلیل احتمالاتی و الگوریتم‌های تصادفی

- در این فصل به معرفی تحلیل احتمالاتی و الگوریتم های تصادفی می پردازیم.
- در این فصل به مبانی نظریه احتمال نیاز خواهیم داشت، برای مرور این مبحث می توانید به بخش X از فصل ضمیمه مراجعه کنید.

مسئله استخدام

- فرض کنید شما می‌خواهید به کمک یک آژانس کاریابی دستیار جدیدی استخدام کنید. آژانس کاریابی هر روز یک متقاضی را برای شما می‌فرستد و شما تصمیم می‌گیرید که آیا او را استخدام کنید یا خیر. برای مصاحبه با هر متقاضی باید مبلغی به آژانس پرداخت کنید، اما استخدام یک متقاضی هزینه‌ی بیشتری دارد، چرا که باید دستیار فعلی را اخراج کرده و هزینه‌ی قابل توجهی به آژانس بپردازید.
- فرض کنید برای اینکه در هر زمان، بهترین فرد ممکن را داشته باشید، این روش را اتخاذ می‌کنید؛ گر هر متقاضی از دستیار فعلی بهتر بود، جایگزین دستیار فعلی شود.

- الگوریتم این رویکرد در شبه کد زیر قابل مشاهده است. متقاضیان از ۱ تا n شماره‌گذاری شده‌اند.

Algorithm HIRE-ASSISTANT

```
1: procedure HIRE-ASSISTANT( $n$ )
2:   best  $\leftarrow 0$            // candidate 0 is a least-qualified dummy candidate
3:   for  $i \leftarrow 1$  to  $n$  do
4:     interview candidate  $i$ 
5:     if candidate  $i$  is better than candidate  $best$  then
6:        $best \leftarrow i$ 
7:     hire candidate  $i$ 
```

- متقاضی شماره ۰ ساختگی است و از تمام متقاضیان دیگر صلاحیت کمتری دارد.

مسئله استخدام

- مشخصاً این روش پر هزینه است. فرض کنید شما حاضر به پرداخت این هزینه هستند اما می خواهید تخمینی از این هزینه به دست آورید.
- برخلاف تحلیل پیچیدگی زمانی ما تمرکز خود را نه بر زمان اجرای الگوریتم، بلکه بر هزینه هایی که برای مصاحبه و استخدام پرداخت می شود، قرار می دهیم.
- در نگاه اول، تحلیل هزینه این الگوریتم ممکن است با تحلیل زمان اجرا متفاوت به نظر برسد. با این حال، تکنیک های تحلیل یکسان هستند، چه در تحلیل هزینه و چه در تحلیل زمان اجرا. در هر دو حالت، ما تعداد دفعاتی که عملیات های پایه ای خاصی انجام می شوند را می شماریم.

مسئله استخدام

- فرض کنید هزینه‌ی مصاحبہ، c_i و هزینه‌ی استخدام، c_h باشد. اگر m تعداد افرادی باشد که استخدام می‌شوند، آنگاه هزینه‌ی کلی برابر است با:

$$O(c_i n + c_h m)$$

- ما همیشه n مقاضی را مصاحبہ می‌کنیم پس همواره هزینه‌ی $c_i n$ را متحمل می‌شویم. بنابراین، تمرکز خود را بر تحلیل هزینه‌ی استخدام یعنی $c_h m$ می‌گذاریم که به ترتیب مقاضیان بستگی دارد.

مسئله استخدام

تحلیل بدترین حالت

- در بدترین حالت، شما هر متقاضی ای که مصاحبه می‌کنید را استخدام می‌دهد که متقاضیان به ترتیب افزایشی از نظر صلاحیت ظاهر شوند؛ در این صورت، شما n بار استخدام می‌کنید و هزینه‌ی کلی استخدام برابر است با:

$$O(c_h n)$$

- البته، متقاضیان همیشه به ترتیب افزایشی نمی‌آیند. در واقع، شما هیچ اطلاعاتی درباره‌ی ترتیب آن‌ها ندارید و کنترلی نیز بر آن ندارید. بنابراین، طبیعی است که بپرسید در حالت معمول یا متوسط چه اتفاقی می‌افتد.

مسئله استخدام

تحلیل احتمالاتی

- تحلیل احتمالاتی^۱، استفاده از احتمالات برای تحلیل مسائل است. از تحلیل احتمالاتی برای تحلیل زمان اجرای الگوریتم و گاهی نیز برای تحلیل کمیت‌های دیگر، مانند هزینه‌ی استخدام در مسئله فعلى استفاده می‌شود.
- برای انجام یک تحلیل احتمالاتی، باید دانشی درباره‌ی توزیع ورودی‌ها داشته باشیم یا فرضیاتی درباره‌ی آن انجام دهیم. سپس الگوریتم‌مان را تحلیل می‌کنیم و زمان اجرای میانگین آن را محاسبه می‌کنیم؛ یعنی به جای بررسی بدترین حالت یا بهترین حالت، میانگین زمان اجرا را روی همه ورودی‌های ممکن در نظر می‌گیریم. این میانگین با استفاده از محاسبه امید ریاضی به دست می‌آید.
- به زمان اجرایی که از این طریق به دست می‌آید، زمان اجرای حالت میانگین^۲ می‌گوییم.

¹ Probabilistic analysis

² average-case running time

مسئله استخدام

تحلیل احتمالاتی

- برای برخی مسائل، می‌توان فرض‌های معقولی دربارهٔ توزیع تمام ورودی‌های ممکن در نظر گرفت. اما برای دیگر مسائل، نمی‌توان توزیع معقولی برای ورودی‌ها در نظر گرفت؛ در این موارد، تحلیل احتمالاتی ممکن نیست.
- برای مسئلهٔ استخدام، می‌توان فرض کرد که متقارضیان به ترتیب تصادفی وارد می‌شوند. با استفاده از این دانش چطور می‌توانیم توزیع ورودی‌ها را مشخص کنیم؟
- بایید به هر متقارضی یک رتبهٔ یکتا از ۱ تا n بر اساس صلاحیت اختصاص دهیم. رتبهٔ متقارضی i را با $rank(i)$ نمایش می‌دهیم و فرض می‌کنیم که رتبه بالاتر به معنای صلاحیت بیشتر است. (مثلاً اگر $rank(3) = 1$ باشد، یعنی متقارضی شماره‌ی ۳ بهترین است.)

مسئله استخدام

تحلیل احتمالاتی

- تعداد $n!$ حالت برای ترتیب رتبه‌ها وجود دارد. این‌که متقاضیان به ترتیب تصادفی وارد می‌شوند، معادل آن است که بگوییم هر یک از این $n!$ حالت، با احتمال مساوی ظاهر می‌شوند. به عبارتی، رتبه‌ها یک جایگشت تصادفی یکنواخت^۱ تشکیل می‌دهند.
- به طور خلاصه؛ وقتی افراد تصادفی وارد می‌شوند، یعنی ترتیب رتبه‌ها تصادفی و یکنواخت است. یعنی همهٔ حالت‌های ممکن برای چیدن رتبه‌ها، با احتمال مساوی رخ می‌دهند.

^۱ uniform random permutation

مسئله استخدام الگوریتم‌های تصادفی

- اما در بسیاری از موارد، اطلاعات اندکی دربارهٔ توزیع ورودی‌ها داریم. حتی اگر دانشی هم داشته باشیم، ممکن است مدل‌سازی محاسباتی آن امکان‌پذیر نباشد. با این حال افزودن رفتار تصادفی به بخشی از الگوریتم می‌تواند کمک کننده باشد.
- در مسئله استخدام، ممکن است به نظر برسد که مقاضیان به ترتیب تصادفی ارائه می‌شوند، اما هیچ راهی برای اطمینان از آن ندارید. بنابراین، برای توسعهٔ یک الگوریتم تصادفی برای این مسئله، باید کنترل بیشتری بر ترتیب مقاضیان داشته باشد.
- پس مدل را اندکی تغییر می‌دهیم: آژانس کاریابی فهرست تمام n مقاضی را از قبل به شما می‌دهد. در هر روز، شما به صورت تصادفی انتخاب می‌کنید که با کدام مقاضی مصاحبه کنید. اگرچه دربارهٔ مقاضیان اطلاعاتی ندارید، اکنون به جای پذیرش ترتیبی که آژانس به شما می‌دهد و امید داشتن به تصادفی بودن آن، شما کنترل فرایند را در دست دارید و ترتیبی تصادفی را اعمال می‌کنید.

مسئله استخدام الگوریتم‌های تصادفی

- به طور کلی، یک الگوریتم را تصادفی^۱ می‌نامیم اگر رفتار آن تنها به ورودی وابسته نباشد، بلکه به مقادیری که یک تولیدکنندهٔ اعداد تصادفی^۲ نیز تولید می‌کند، بستگی داشته باشد.
- فرض می‌کنیم که یکتابع تولیدکنندهٔ اعداد تصادفی به نام RANDOM در اختیار داریم. یک فراخوانی به صورت $\text{RANDOM}(a, b)$ عددی صحیح بین a و b (شامل هر دو) بازمی‌گرداند، به‌طوری که هر عدد با احتمال مساوی ظاهر می‌شود.

^۱ randomized

^۲ random-number generator

مسئله استخدام الگوریتم‌های تصادفی

- برای تحلیل زمانی الگوریتم‌های تصادفی، امید ریاضی زمان اجرا را بر روی توزیع مقادیر بازگردانده شده توسط تولیدکننده محاسبه می‌کنیم.
- این الگوریتم‌ها را از الگوریتم‌هایی که ورودی آن‌ها تصادفی است، متمایز می‌کنیم؛ و زمان اجرای آن‌ها را «زمان اجرای مورد انتظار»^۱ می‌نامیم.
- به طور کلی، وقتی توزیع احتمالاتی در ورودی وجود داشته باشد، در مورد «زمان اجرای حالت میانگین»، و وقتی که خود الگوریتم تصمیمات تصادفی می‌گیرد، در مورد «زمان اجرای مورد انتظار» بحث می‌کنیم.

¹ expected running time

متغیرهای تصادفی نشانگر

- برای تحلیل بسیاری از الگوریتم‌ها، از جمله مسئله‌ی استخدام، از متغیرهای تصادفی نشانگر^۱ استفاده می‌کنیم که نوعی از متغیرهای تصادفی هستند. این متغیرها ابزار مناسبی برای تبدیل بین احتمال و امید ریاضی فراهم می‌کنند.
- فرض کنید یک فضای نمونه S و یک رویداد A داریم. آنگاه متغیر تصادفی نشانگر نسبت به رویداد A ، به صورت زیر تعریف می‌گردد:

$$I\{A\} = \begin{cases} 1 & \text{اگر } A \text{ رخ دهد} \\ 0 & \text{در غیر این صورت} \end{cases}$$

^۱ Indicator random variables

متغیرهای تصادفی نشانگر

- به عنوان مثالی ساده، بباید امید ریاضی تعداد دفعاتی که در پرتاپ یک سکه، «شیر» ظاهر می‌شود را محاسبه کنیم. فضای نمونه برای یک بار پرتاپ سکه برابر است با $S = \{H, T\}$. متغیر تصادفی نشانگر X_H را برای رویداد H (آمدن شیر) تعریف می‌کنیم:

$$X_H = I\{H\} = \begin{cases} 1 & \text{اگر } H \text{ رخ دهد} \\ 0 & \text{اگر } T \text{ رخ دهد} \end{cases}$$

- در نحوه نوشتار بالا X_H یک متغیر تصادفی است که برابر با $I\{H\}$ قرار داده شده. $I\{H\}$ به طور خاص به تعریف متغیر تصادفی نشانگر اشاره می‌کند.

متغیرهای تصادفی نشانگر

- امید ریاضی تعداد دفعات آمدن شیر در یک پرتاپ، برابر با امید ریاضی X_H است:

$$E[X_H] = 1 \cdot \Pr\{H\} + 0 \cdot \Pr\{T\} = \frac{1}{2}$$

- پس امید ریاضی متغیر تصادفی X_H برابر است با احتمال رویداد H .
- به طور کلی اگر X_A یک متغیر تصادفی نشانگر برای رویداد A باشد، آنگاه داریم:

$$E[X_A] = \Pr\{A\}$$

متغیرهای تصادفی نشانگر

- شاید متغیرهای نشانگر برای چنین مثال ساده‌ای بیش از حد پیچیده به نظر برسند، ولی در تحلیل موقعیت‌هایی که شامل تکرار آزمایش‌های تصادفی هستند، بسیار سودمندند.
- برای مثال، فرض کنید می‌خواهیم امید ریاضی تعداد شیر در n پرتاپ سکه را محاسبه کنیم. یک روش محاسبه احتمال وقوع ۰، ۱، ۲، و ... شیر است، اما راه ساده‌تری با استفاده از متغیرهای نشانگر وجود دارد.
- فرض کنید X_i متغیر نشانگر برای رخداد رویداد «آمدن شیر در پرتاپ i -ام» باشد. آنگاه متغیر تصادفی X که تعداد کل دفعات شیر در n پرتاپ را نشان می‌دهد، به صورت زیر تعریف می‌شود:

$$X = \sum_{i=1}^n X_i$$

متغیرهای تصادفی نشانگر

- برای محاسبه امید ریاضی X ، از خاصیت خطی بودن امید ریاضی استفاده می‌کنیم (امید ریاضی مجموع برابر است با مجموع امید ریاضی):

$$E[X] = E\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n E[X_i]$$

- احتمال شیر آمدن در هر پرتاپ برابر $\frac{1}{2}$ است پس:

$$E[X_i] = \frac{1}{2} \quad \Rightarrow \quad \sum_{i=1}^n E[X_i] = \frac{n}{2}$$

در نتیجه:

$$E[X] = \frac{n}{2}$$

متغیرهای تصادفی نشانگر

تحلیل مسئله‌ی استخدام

- حال دوباره به مسئله‌ی استخدام بازمی‌گردیم. می‌خواهیم امید ریاضی تعداد دفعاتی که شما یک دستیار اداری جدید استخدام می‌کنید را محاسبه کنیم. همان‌طور که پیش‌تر بحث شد، فرض می‌کنیم که متقارضیان به صورت تصادفی وارد می‌شوند.
- فرض کنید X متغیر تصادفی‌ای باشد که برابر است با تعداد دفعاتی که یک متقارضی جدید استخدام می‌شود.
اگر بخواهیم امید ریاضی X را مستقیماً از محاسبه کنیم، داریم:

$$E[X] = \sum_{x=1}^n x \cdot \Pr\{X = x\}$$

متغیرهای تصادفی نشانگر

تحلیل مسئله‌ی استخدام

- محاسبه مستقیم پیچیده خواهد بود. به جای آن، از متغیرهای نشانگر استفاده می‌کنیم. ابتدا n متغیر نشانگر تعریف می‌کنیم که مشخص می‌کند آیا متقارضی i ام استخدام شده است یا نه.

$$X_i = I_{\{\text{متقارضی } i \text{ استخدام شود}\}} = \begin{cases} 1 & \text{اگر متقارضی } i \text{ استخدام شود} \\ 0 & \text{در غیر این صورت} \end{cases}$$

$$X = X_1 + X_2 + \cdots + X_n$$

- در نتیجه:
- طبق خاصیت متغیرهای نشانگر می‌دانیم:

$$E[X_i] = \Pr\{\text{متقارضی } i \text{ استخدام شود}\}$$

متغیرهای تصادفی نشانگر

تحلیل مسئله‌ی استخدام

- متقاضی i -ام زمانی استخدام می‌شود که از همهٔ متقاضیان ۱ تا $1 - i$ بهتر باشد. از آنجا که فرض کردہ‌ایم ترتیب ورود متقاضیان تصادفی است، i متقاضی اول به صورت تصادفی ترتیب یافته‌اند، و هر یک از آن‌ها به طور مساوی می‌تواند بهترین باشد. بنابراین، احتمال اینکه متقاضی i بهترین در میان این i نفر باشد برابر با $\frac{1}{i}$ است.
- اکنون می‌توانیم امید ریاضی X را محاسبه کنیم:

$$E[X] = E\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n E[X_i] = \sum_{i=1}^n \frac{1}{i} = \ln n + O(1)$$

- پاسخ این جمع با استفاده از ویژگی‌های سری هارمونیک داده شده است.

متغیرهای تصادفی نشانگر

تحلیل مسئله‌ی استخدام

- بنابراین در الگوریتم HIRE-ASSISTANT در حالت میانگین حدود $\ln n$ نفر را استخدام می‌کنید.
- پس اگر مقاضیان به صورت تصادفی وارد شوند، هزینه‌ی میانگین استخدام برابر است با:

$$O(c_h \ln n)$$

بنابراین، هزینه‌ی میانگین استخدام به طور قابل توجهی کمتر از هزینه بدترین حالت ($O(c_h n)$) است.

الگوریتم‌های تصادفی

- دیدیم که چگونه دانستن توزیع روی ورودی‌ها می‌تواند به ما در تحلیل رفتار یک الگوریتم در حالت میانگین کمک کند. اما اگر توزیع ورودی را ندانیم چه؟
- آنگاه نمی‌توانیم الگوریتم را در حالت میانگین انجام تحلیل کنیم. پیش‌تر دیدم که در این حالت استفاده از یک الگوریتم تصادفی^۱ می‌تواند مفید باشد.
- برای برخی مسائل، مانند مسئله‌ی استخدام، فرض برابر بودن احتمال همهی جایگشت‌های ورودی مفید است. در این حالت به جای آنکه یک توزیع برای ورودی فرض کنیم، یک توزیع به آن تحمیل می‌کنیم.

¹ randomized algorithms

الگوریتم‌های تصادفی

- دیدم که، پیش از اجرای الگوریتم استخدام، فهرست نامزدها را به صورت تصادفی جایگشت می‌دهیم تا تضمین کنیم که هر جایگشت به یک اندازه محتمل است. نشان دادیم با اینکار حدود $\ln n$ بار یک دستیار جدید استخدام می‌کنیم.
- با انجام این جایگشت تصادفی، اکنون این میانگین برای هر ورودی برقرار است، نه فقط برای ورودی‌هایی که از یک توزیع خاص گرفته شده‌اند.

الگوریتم‌های تصادفی

- باید تفاوت میان تحلیل احتمالاتی و الگوریتم‌های تصادفی را بیشتر بررسی کنیم.
- اگر الگوریتم استخدام به صورت قطعی پیاده شود، برای هر ورودی مشخص، تعداد دفعاتی که دستیار جدیدی استخدام می‌شود همیشه یکسان است. چون این تعداد تنها به رتبه‌ها وابسته است، برای نمایش یک ورودی خاص کافی است رتبه‌ها را فهرست کنیم.
- به طور مثال برای فهرست:
 $\langle 5, 2, 1, 8, 4, 7, 10, 9, 3, 6 \rangle$
- سه بار (رتبه‌های ۵، ۸، و ۱۰) دستیار جدید استخدام می‌شود.

الگوریتم‌های تصادفی

- بنابراین هزینه الگوریتم، به ورودی وابسته است. در مقابل، الگوریتم تصادفی‌ای را در نظر بگیرید که ابتدا فهرست نامزدها را به صورت تصادفی جایگشت می‌دهد. در این حالت برای یک ورودی خاص، نمی‌توانیم بگوییم چند بار مقدار بیشینه به روزرسانی می‌شود، چون این مقدار در هر بار اجرای الگوریتم فرق می‌کند.
- برای بسیاری از الگوریتم‌های تصادفی، هیچ ورودی خاصی منجر به بدترین حالت نمی‌شود. حتی دشمن شما هم نمی‌تواند یک آرایه‌ی ورودی بد تولید کند، چون جایگشت تصادفی ترتیب ورودی را بی‌اثر می‌کند.
- شبه کد مسئله استخدام که پیش‌تر بررسی کردیم را در نظر بگیرید. اگر در ابتدای آن، آرایه‌ی نامزدها را تصادفی جایگشت دهیم، الگوریتم RANDOMIZED-HIRE-ASSISTANT به دست می‌آید.

الگوریتم‌های تصادفی

- همانطور که قبلًا دیدیم، اگر ترتیب ورودی‌ها تصادفی باشد هزینه‌ی استخدام در «حالت میانگین» $HIRE\text{-ASSISTANT}$ برابر است با:

$$O(clnn)$$

- در الگوریتم $RANDOMIZED\text{-}HIRE\text{-}ASSISTANT$ بدون فرض توزیع خاصی بر روی ورودی هزینه استخدام «مورد انتظار» برابر است با

$$O(clnn)$$

- اصطلاح، هزینه‌ی «حالت میانگین» را برای تحلیل احتمالاتی و هزینه‌ی «مورد انتظار» را برای الگوریتم‌های تصادفی به کار می‌بریم. وقتی می‌گوییم حالت میانگین، یعنی تحلیل احتمالاتی انجام داده‌ایم و توزیعی برای ورودی فرض کرده‌ایم اما در مورد الگوریتم‌های تصادفی حدس توزیع ورودی وجود ندارد پس بهتر است این دو کار را با اسمی متفاوتی نام‌گذاری کنیم.

الگوریتم‌های تصادفی

جایگشت تصادفی آرایه‌ها

- بسیاری از الگوریتم‌های تصادفی، ورودی را به صورت تصادفی جایگشت می‌دهند. هدف تولید یک «جایگشت تصادفی یکنواخت»^۱ است، یعنی هر جایگشت به یک اندازه محتمل باشد.
- چون تعداد جایگشت‌ها برابر با $n!$ است، می‌خواهیم احتمال تولید هر جایگشت برابر با $\frac{1}{n!}$ باشد.

^۱ uniform random permutation

الگوریتم‌های تصادفی

جایگشت تصادفی آرایه‌ها

- الگوریتم RANDOMLY-PERMUTE آرایه $A[1 \dots n]$ را در زمان $\Theta(n)$ جایگشت می‌دهد:

Algorithm RANDOMLY-PERMUTE

```
1: procedure RANDOMLY-PERMUTE( $A, n$ )
2:   for  $i \leftarrow 1$  to  $n$  do
3:     swap  $A[i]$  with  $A[\text{RANDOM}(i, n)]$ 
```

- این الگوریتم یک جایگشت تصادفی یکنواخت تولید می‌کند.

پیوست شماره یک: مروری بر نظریه زبان‌ها صوری

مروری بر نظریه زبان‌ها صوری

- یک الفبا Σ یک مجموعه متناهی از نمادهای است. یک زبان L روی Σ ، هر مجموعه‌ای از رشته‌هایی است که از نمادهای موجود در Σ ساخته شده باشند.
- برای مثال، اگر $\Sigma = \{0, 1\}$ ، آنگاه مجموعه

$$L = \{10, 11, 101, 111, 1011, 1101, 10001, \dots\}$$

یک زبان است که حاوی نمایش دودویی اعداد اول است.

- رشته تهی را با ϵ نمایش می‌دهیم و زبان تهی را با \emptyset و ${}^*\Sigma$ یک زبان شامل همه رشته‌ها روی الفبای Σ است.

مروری بر نظریه زبان‌ها صوری

- هر زبان L روی Σ زیرمجموعه‌ای از ${}^*\Sigma$ است.
- زبان‌ها از عملیات مختلفی پشتیبانی می‌کنند. از انجا که زبان‌ها مجموعه هستند، عملیات‌های نظریه مجموعه‌ها مانند اجتماع و اشتراک روی آنها تعریف می‌شود که مستقیماً از تعریف‌های مجموعه‌ای پیروی می‌کنند.
- همچنین متمم^۱ یک زبان L به صورت زیر تعریف می‌شود:

$$\overline{L} = \Sigma^* - L$$

¹ complement

- الحاق^۱ دو زبان L_1 و L_2 زبانی است به صورت:

$$L = \{x_1 x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2\}$$

- بستار^۲ یا ستاره کلینی^۳ یک زبان L ، زبانی است به صورت:

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \dots$$

همچنین L^k زبانی است که از الحاق k مرتبه‌ای زبان L با خودش به دست می‌آید.

^۱ concatenation

^۲ closure

^۳ Kleene star

مروری بر نظریه زبان‌ها صوری

- در دیدگاه نظریه زبان‌ها مجموعه Σ^* است که در آن $\Sigma = \{0, 1\}$ ، مجموعه همه نمونه‌های یک مسئله تصمیم‌گیری است.
- از آنجا که Q به‌طور کامل با آن دسته از نمونه‌های مسئله که خروجی آن‌ها برابر ۱ (یعنی پاسخ «بله») است مشخص می‌شود، می‌توان Q را به صورت زبان L روی $\Sigma = \{0, 1\}$ در نظر گرفت، به طوری که:

$$L = \{x \in \Sigma^* : Q(x) = 1\}$$

مروری بر نظریه زبان‌ها صوری

- برای مثال، مسئله‌ی تصمیم‌گیری PATH دارای زبان متناظر زیر است:

$\text{PATH} = \{\langle G, u, v, k \rangle : G = (V, E) \text{ graph undirected an is}$

$u, v \in V$

$k \geq 0$ and integer, an is

G edges k most at with v to u from path a contains $G\}$

- گاهی از نام یکسان برای اشاره به هر دو مفهوم «مسئله‌ی تصمیم‌گیری» و «زبان متناظر آن» استفاده می‌کنیم.
برای مثال PATH می‌تواند به مسئله تصمیم‌گیری کوتاه‌ترین مسیر یا زبان بالا اشاره داشته باشد.

مروری بر نظریه زبان‌ها صوری

- چارچوب زبان‌های صوری به ما امکان می‌دهد که رابطه‌ی بین مسائل تصمیم‌گیری و الگوریتم‌هایی که آن‌ها را حل می‌کنند را به‌طور فشرده بیان کنیم.
- می‌گوییم که یک الگوریتم A رشته‌ای $\{0, 1\}^*$ را «می‌پذیرد»¹ اگر خروجی الگوریتم با ورودی x برابر با ۱ باشد، یعنی $1 = A(x)$ زبان پذیرفته شده توسط الگوریتم A مجموعه‌ای از رشته‌های است که الگوریتم آن‌ها را می‌پذیرد:

$$L = \{x \in \{0, 1\}^* : A(x) = 1\}$$

¹ accepts

- اگر $A(x) = 0$ باشد، می‌گوییم الگوریتم A رشته‌ی x را «رد می‌کند».^۱.
- حتی اگر الگوریتمی مثل A زبانی مانند L را بپذیرد، این الگوریتم لزوماً رشته‌ی $L \notin x$ را که به آن داده شده است رد نمی‌کند. به جای آن، ممکن است الگوریتم برای همیشه در حلقه بماند.

¹ rejects

مروری بر نظریه زبان‌ها صوری

- زبان L توسط الگوریتم A «تصمیم‌گیری»^۱ می‌شود اگر:
 - ۱ هر رشته‌ی دودویی در L توسط A پذیرفته شود، و
 - ۲ هر رشته‌ی دودویی که در L نیست، توسط A رد شود.
- به یک زبان «قابل پذیرش در زمان چندجمله‌ای»^۲ گفته می‌شود اگر:
 - ۱ توسط الگوریتم A پذیرفته شود، و
 - ۲ یک ثابت k وجود داشته باشد به‌طوری‌که برای هر رشته‌ی A رشته‌ی x را در زمان (n^k) پذیرد.

^۱ decided

^۲ accepted in polynomial time

مروری بر نظریه زبان‌ها صوری

- به طور مشابه، به یک زبان «قابل تصمیم‌گیری در زمان چندجمله‌ای»^۱ گفته می‌شود اگر یک ثابت k وجود داشته باشد به‌طوری‌که برای هر رشته‌ی $x \in 0, 1^*$ با طول n الگوریتم در زمان $O(n^k)$ به درستی تشخیص دهد که آیا $x \in L$ است یا نه.
- در نتیجه، برای «پذیرفتن» یک زبان، یک الگوریتم کافیست تنها برای رشته‌هایی که در زبان موجود اند پاسخ تولید کند. اما برای «تصمیم‌گیری»، باید برای تمام رشته‌های دودویی در $0, 1^*$ به درستی تشخیص دهد که آیا عضو L هستند یا نه.
- برای مثال، زبان PATH قابل پذیرش و قابل تصمیم‌گیری در زمان چند جمله‌ایی است. اما برای برخی مسائل دیگر، مانند مسئله توقف تورینگ، الگوریتم پذیرنده وجود دارد، اما هیچ الگوریتم تصمیم‌گیرنده‌ای وجود ندارد.

^۱ decided in polynomial