



دانشکده مهندسی کامپیوتر

جزوه درس

طراحی و تحلیل الگوریتم

استاد درس: سید صالح اعتمادی*

نیم‌سال دوم

سال تحصیلی ۹۸-۹۹

* مطالب این جزوه توسط دانشجویان جمع‌آوری شده است. استاد درس درستی مطالب را بررسی نکرده است.

فهرست مطالب

۷	Paths in Graphs	۲
	نگارزین العابدین - ۱۳۹۸/۱۱/۲۰	
۷	دید کلی	۱.۲
۸	Shortest path	۲.۲
۸	دور منفی چیست؟	۳.۲
۹	Dijkstra / BFS	۴.۲
۱۰	Bidirectional dijkstr / Bidirectional BFS	۵.۲
۱۲	Dijkstra And Bellman-Ford Algorithms	۴
	پریسا علائی - ۱۳۹۸/۱۱/۲۷	
۱۲	نکته :	۱.۴
۱۲	Dijkstra's Algorithm	۲.۴
۱۳	Order of Dijkstra Algorithm	۳.۴
۱۳	مزایای Dijkstra Algorithm :	۴.۴
۱۴	معایب Dijkstra Algorithm :	۵.۴
۱۴	مثال :	۶.۴
۱۷	شبه کد Dijkstra Algorithm :	۷.۴
۱۸	اثبات درست بودن الگوریتم Dijkstra	۸.۴
۱۹	Bellman-Ford algorithm	۹.۴
۱۹	Order of Bellman-Ford Algorithm	۱۰.۴
۱۹	مزایای Bellman-Ford Algorithm :	۱۱.۴

۲۰	معایب Bellman-Ford Algorithm :	۱۲.۴
۲۰	مثال :	۱۳.۴
۲۳	شبه کد :	۱۴.۴
۲۳	اثبات درست بودن الگوریتم Bellman-Ford	۱۵.۴
۲۴	دور منفی در Bellman-Ford :	۱۶.۴
۲۴	اثبات درست بودن الگوریتم دور منفی در Bellman-Ford	۱۷.۴
۲۵	Infinite Arbitrage :	۱۸.۴
۲۵	اثبات درست بودن الگوریتم Infinite Arbitrage	۱۹.۴
۲۶	سایت‌های دیگر برای مراجعه :	۲۰.۴

۲۷ **Bellman Ford و Dijkstra** ۵

یاسمن لطف‌اللهی - ۱۳۹۸/۱۱/۲۹

۲۷	شبه‌کد Dijkstra	۱.۵
۲۸	شبه‌کد پیدا کردن کوتاه‌ترین راه بین دو راس به کمک آرایه prev	۲.۵
۲۸	اثبات درستی الگوریتم Dijkstra	۳.۵
۲۹	پیچیدگی زمانی الگوریتم Dijkstra	۴.۵
۳۰	چرا Dijkstra برای گراف‌هایی که یال منفی دارند، کار نمی‌کند؟	۵.۵
۳۰	چرا اضافه کردن به یال‌ها و استفاده از Dijkstra جواب نمی‌دهد؟	۶.۵
۳۰	یک مثال از کاربرد الگوریتم Bellman Ford : تبدیل ارز	۷.۵
۳۳	شبه‌کد Bellman Ford	۸.۵
۳۳	اثبات درستی الگوریتم Bellman Ford	۹.۵

۳۴ **دور منفی در گراف و دایجسترا دوطرفه** ۶

ملیکا نوبختیان - ۱۳۹۸/۱۲/۴

۳۴	قضیه	۱.۶
۳۴	اثبات	۲.۶
۳۵	Finding Negative Cycle	۳.۶
۳۷	Infinite Arbitrage	۴.۶
۳۹	Bidirectional Search	۵.۶
۴۲	Bidirectional Dijkstra	۶.۶

۴۷ **درخت‌های پوشای کمینه** ۸

فهرست مطالب

۳

سهراب نمازی - ۱۳۹۸/۱۲/۱۱

۴۷	تعریف درخت پوشای کمینه	۱.۸
۴۸	کاربرد درخت های پوشای کمینه	۲.۸
۴۹	بدست آوردن درخت پوشای کمینه	۳.۸
۵۴	خلاصه	۴.۸

۹ الگوریتم جست وجو A*

محمدعلی فراحت - ۱۳۹۸/۱۲/۱۳

۵۵	ایده کلی	۱.۹
۵۵	Potential-function	۲.۹
۵۶	محاسبه کوتاهترین مسیر	۳.۹
۵۶	Bidirectional-A*	۴.۹

۱۱ SuffixTree

احمد بهمنی - ۱۳۹۹/۱/۳

۵۸	مقدمه	۱.۱۱
۵۹	پیدا کردن الگو در رشته	۲.۱۱

۱۲ BWT

متین مرجانی - ۱۳۹۹/۱/۱۷

۶۳	مقدمه	۱.۱۲
۶۳	BWT	۲.۱۲
۶۴	Constructing BWT	۳.۱۲
۶۶	Inverting BWT	۴.۱۲

۱۳ تطبیق الگوها و تبدیل BW

شایان موسوی نیا - ۱۳۹۹/۲/۱۶

۶۸	یک مشاهده عجیب	۱.۱۳
۷۲	پیدا کردن تطبیق الگو با استفاده از BWT	۲.۱۳

۱۴ الگوریتم KMP

امید میرزاجانی - ۱۳۹۹/۲/۱۲

۷۶	Brute Force	۱.۱۴
----	-------------	------

۴	فهرست مطالب
۷۷	Function Prefix ۲.۱۴
۷۸	الگوریتم نهایی ۳.۱۴
۷۹	الگوریتم kmp و efficient suffix array ۱۵ صدرا خاموشی - ۱۳۹۸/۲/۶
۷۹	معیار ارزیابی جزوه ۱.۱۵
۸۷	آرایه پسوندی بهینه و آرایه LCP ۱۶ زهرا حسینی - ۱۳۹۹/۱/۲۶
۸۷	دوره مفاهیم آرایه و درخت پسوندی ۱.۱۶
۸۸	ساخت آرایه پسوندی ۲.۱۶
۹۷	LCP ARRAY ۳.۱۶
۱۰۱	SuffixTree ۱۷ هستی کرمدل - ۱۳۹۹/۱/۳۱
۱۰۹	جریان درگراف ۱۸ محمد مصطفی رستم خانی - ۱۳۹۹/۲/۲
۱۰۹	جریان در شبکه: (flows in network) ۱.۱۸
۱۱۰	network ۲.۱۸
۱۱۴	گراف باقیمانده: (residual graph) ۳.۱۸
۱۱۷	جریان باقیمانده: (residual flow): ۴.۱۸
۱۱۸	maxflow: and Mincut ۵.۱۸
۱۲۱	الگوریتم های پیدا کردن flow ۱۹ آرمین غلام پور - ۱۳۹۹/۲/۷
۱۲۱	Ford Fulkerson ۱.۱۹
۱۲۴	Edmonds-Karp ۲.۱۹
۱۲۵	MeasureIt + Applications Flow Network ۲۰ فاطمه احمدی - ۱۳۹۹/۲/۹
۱۲۶	گراف دو بخشی ۱.۲۰
۱۲۷	تطابق درگراف ۲.۲۰
۱۲۸	Match Making ۳.۲۰

۱۲۸	Scheduling	۴.۲۰
۱۲۹	Find Maximum Matching	۵.۲۰
۱۳۲	Maxflow-Mincut	۶.۲۰
۱۳۳	Konig's Theorem	۷.۲۰
۱۳۴	The Marriage Lemma	۸.۲۰
۱۳۴	Image Segmentation	۹.۲۰
۱۳۷	Measure It	۱۰.۲۰
۱۳۹	Linear Programming	۲۱
	فاطمه امیدی - ۱۳۹۹/۲/۱۴	
۱۴۰	a Linear programming example	۱.۲۱
۱۴۰	وضعیت های جواب	۲.۲۱
۱۴۱	Row reduction	۳.۲۱
۱۴۳	برنامه ریزی خطی - دوگان	۲۲
	غزل زمانی نژاد - ۱۳۹۹/۲/۱۶	
۱۴۳	چندوجهی های محدب	۱.۲۲
۱۴۶	دوگان یک مسئله	۲.۲۲
۱۴۹	Complementary Slackness	۳.۲۲
۱۵۱	خلاصه ی مطالب	۴.۲۲
۱۵۲	Optimization و الگوریتم Simplex	۲۳
	محمدحسین کریمیان - ۱۳۹۹/۲/۲۱	
۱۵۲	اهداف آموزشی جلسه	۱.۲۳
۱۵۲	انواع مسائل Optimization	۲.۲۳
۱۵۳	استفاده از حل یک فرم برای حل فرم های دیگر	۳.۲۳
۱۵۵	الگوریتم Simplex	۴.۲۳
۱۵۶	الگوریتم Ellipsoid	۵.۲۳
۱۵۹	مسائل NP-complete	۲۴
	مجتبی نافذ - ۱۳۹۹/۰۲/۲۳	
۱۶۰	مسائل جستجو search problems	۱.۲۴
۱۶۰	صدق پذیری دودویی(Satisfiability) SAT problem	۲.۲۴

۱۶۰	Conjunctive Normal Form	فرم نرمال ترکیب عطفی	۳.۲۴
۱۶۱	دسته بندی مسائل		۴.۲۴
۱۶۱	نمونه هایی از مسائل با دو ورژن ساده و سخت		۵.۲۴
۱۶۵	مسائل NP, P		۶.۲۴
۱۶۶	Reductions	کاهش	۷.۲۴
۱۶۸	NP-hard, NP-complete	مسائل	۸.۲۴
۱۶۹		حل کردن مسائل NP-complete (حالت های خاص)		۲۶
				هادی شیخی - ۱۳۹۹/۲/۳۰
۱۶۹	مسائل NP-Complete		۱.۲۶
۱۷۰	حل کردن مسائل NP-Complete		۲.۲۶
۱۷۱	SAT-۲ یک حالت خاص [۲-sat]		۳.۲۶
۱۷۴		حالت های مواجهه با مسائل NP-Complete		۲۸
				ملیکا احمدی رنجبر - ۱۳۹۹/۳/۶
۱۷۴	Independent Set	حالت خاص	۱.۲۸
۱۷۷	3-Satisfiability		۲.۲۸
۱۷۸	Local Search		۳.۲۸
۱۷۹	Traveling Salesman Problem (TSP)		۴.۲۸

جلسه ۲

Paths in Graphs

نگار زین العابدین - ۱۳۹۸/۱۱/۲۰

۱.۲ دید کلی

در این چند جلسه‌ای که در پیش داریم، به مباحث مربوط به گراف اشاره خواهیم کرد.

- به طور کلی مباحث ما، شامل سه دسته زیر می‌شود که در آینده‌ای نزدیک، به آنها خواهیم پرداخت:

۱. Shortest path

۲. Minimum spanning trees

۳. Advances shortest path

- در مبحث shortest path، سه الگوریتم زیر را شرح خواهیم داد:

۱. BFS (Breadth-First-Search)

۲. Dijkstra

۳. Bellmanford

Shortest path ۲.۲

همان‌طور که در بالا اشاره کردیم، قصد معرفی سه الگوریتم برای به دست آوردن کوتاه‌ترین مسیر در گراف shortest path را داریم.

- BFS :

برای پیدا کردن کوتاه‌ترین مسیر در گراف، از راسی (Node) به تمامی راس‌های دیگر گراف، از این الگوریتم استفاده می‌کنیم.

- Dijkstra :

اگر بر روی راس‌های گرافمان، وزن داشته باشیم، از این الگوریتم استفاده می‌کنیم. این الگوریتم نیز مانند BFS کوتاه‌ترین مسیر را، از راسی به تمام راس‌های دیگر گرافمان، حساب می‌کند. البته باید به این نکته نیز توجه کرد؛ اگر در گرافمان، وزن منفی داشتیم، استفاده کردن از این الگوریتم، مجاز نیست و جواب به دست آمده صحیح نمی‌باشد.

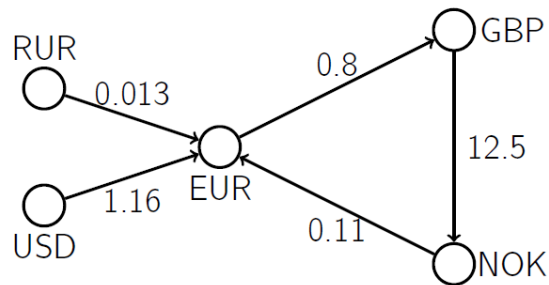
- Bellmanford :

همان‌طور که در بالا اشاره شد، اگر گراف مورد نظرمون وزن منفی داشته باشد، نمی‌توانیم از الگوریتم Dijkstra برای محاسبه کوتاه‌ترین مسیر استفاده کنیم. در این حالت، Bellmanford به کار برده می‌شود. نکته مهم در این الگوریتم این است که اگر گرافمان، دارای دور منفی باشد، الگوریتم، به خوبی عمل نکرده و با مشکل مواجه می‌شود. دلیل این رخداد در آینده با جزئیات شرح داده می‌شود.

۳.۲ دور منفی چیست ؟

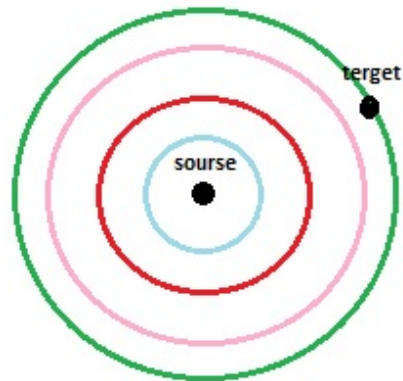
اگر در گراف وزن‌داری، دوری داشته باشیم که جمع یال‌های آن منفی شود، در این حالت گراف ما دارای دور منفی می‌باشد.

- یکی از کاربردهایی که می‌توان برای دور منفی نام برد؛ بدین شرح است که فرض کنید می‌خواهیم نرخ‌های مختلفی از پول‌ها را به گونه‌ای به یک‌دیگر تبدیل کنیم که بیشترین سود نصیبمان بشود. جزئیات این مسئله و ارتباط آن با دور منفی در جلسات آینده، شرح داده خواهد شد.



۴.۲ Dijkstra / BFS

اگر به زبان ساده‌ای به شرح این دو الگوریتم بپردازیم، می‌توانیم بگوییم که این دو الگوریتم، در ابتدا، از نقطه مبدا شروع و تمام نقاطی که فاصله‌ی یکسانی با نقطه‌ی اولیه دارند را پیدا می‌کنند. این کار را به صورت لایه‌ای تا جایی تکرار می‌کنند که به نقطه مقصد برسند. در نتیجه؛ عدد به دست آمده، همان کوتاه‌ترین مسیر ما یا shortest path می‌باشد. البته باید به این نکته توجه داشت که BFS برای گراف‌های بدون وزن (جهت‌دار و بدون جهت) و Dijkstra برای گراف‌های وزن‌دار می‌باشد.



• شبه‌کد * BFS

pseudocode*

```

Input : Graph,Source
Output : BFS on Graph
initialization;
for All  $e$  in Edges do
  | dist[e]=inf
end
dist[Source]=0;
Q <- Source : queue containing just Source
while Q is not empty do
  | u <- Deququq(Q)
  | for All  $(u,v)$  in Edges do
  | | if dist[v] = inf then
  | | | Enqueue(Q,v)
  | | | dist[v] <- dist[u] + 1
  | | end
  | end
end

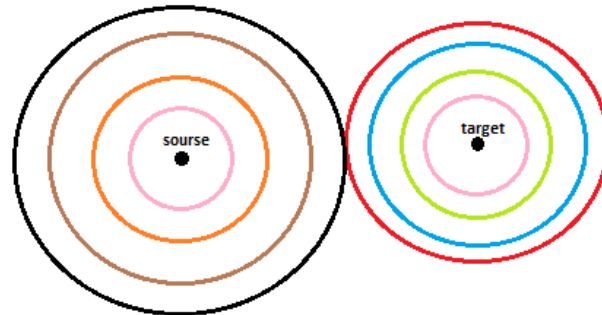
```

Algorithm 1: BFS on graph

- برای درک بیشتر الگوریتم BFS می‌توانید از این لینک و برای الگوریتم dijkstra از این لینک استفاده کنید.

۵.۲ Bidirectional dijkstr / Bidirectional BFS

در ادامه‌ی الگوریتم‌های بالا، دو الگوریتم مشابه‌ای به نام Bidirectional dijkstra و Bidirectional BFS وجود دارند. تنها تفاوت در این است که روندی که در بالا توضیح داده شده، هم از راس مبدا و هم از راس مقصد شروع می‌شود. این کار تا زمانی که به نقطه‌ای مشترک برسند، ادامه پیدا می‌کند. عدد به دست آمده، جواب مورد نظر ما می‌باشد.



- در این دو الگوریتم Bidirectional BFS / Bidirectional dijkstra ، از تعدادی محاسبات بیهوده صرف نظر می‌شود که موجب سریع‌تر شدن برنامه می‌گردد.
- از تفاوت‌هایی که می‌توان بین دو الگوریتم BFS / Dijkstra و Bidirectional BFS / Bidi-rectional dijkstra گرفت، در الگوریتم‌های دسته اول، محاسبات و عملیات‌ها بین راس مبدا و تمام راس‌های دیگر گراف صورت می‌گیرد. این در حالی است که در الگوریتم‌های دسته دوم، تنها به دو راس مبدا و مقصد می‌پردازیم.
- برای اینکه شهود بهتری از الگوریتم‌های بالا دریافت کنید، می‌توانید از [اینجا](#) بهره ببرید.

جلسه ۴

Dijkstra And Bellman-Ford Algorithms

پریسا علانی - ۱۳۹۸/۱۱/۲۷

۱.۴ نکته :

هر قسمتی از یک مسیر بهینه خودش نیز بهینه است .
اثبات: (برهان خلف) فرض کنیم مسیر بهینه ای از S به T باشد که از دو راس u و v می‌گذرد. اگر مسیر بهینه از u به v قسمتی از مسیر بهینه S به T نباشد، یعنی مسیر بهینه‌ی دیگری از u به v وجود دارد که می‌شد برای مسیر از S به T نیز آن را در گذر از u به v انتخاب کرد، پس مسیر کوتاه‌تری از S به T پیدا کردیم و مسیر قبلی ما بهینه نبوده است. تناقض \Rightarrow هر تکه از یک مسیر بهینه خود حتماً بهینه است. [book]

Dijkstra's Algorithm ۲.۴

برای پیدا کردن کوتاه‌ترین مسیر در بین نودهای یک گراف می‌توان از آن استفاده کرد .
نحوه ی عملکرد الگوریتم :

- (۱) برای همه ی گره‌ها یک متغیر فاصله در نظر می‌گیریم و مقدار اولیه ی آن را بی نهایت می‌گذاریم .
- (۲) انتخاب گره ی شروع و قرار دادن صفر برای مقدار فاصله ی آن
- (۳) محاسبه ی فاصله ، از گره‌ای که در آن قرار داریم تا همه‌ی همسایگان آن و قرار دادن فاصله‌ی همسایه‌ها با مقادیر به دست آمده؛ اگر مقدار محاسبه شده از مقداری که اکنون دارند، کمتر باشد .
- (۴) از بین تمام همسایه‌های نودهایی که بازدید کرده ایم ، نودی که کمترین فاصله را دارد انتخاب می‌کنیم و مرحله ی سوم را دوباره اجرا می‌کنیم . نکته : نودی که قبلاً بازدید شده‌است را هرگز دوباره بررسی نمی‌کنیم .
- (۵) این الگوریتم زمانی تمام می‌شود که تمام همسایه‌ها مورد بررسی قرار گیرند . اگر بعد از پایان بررسی ، نودی وجود داشت که فاصله ی آن بی نهایت بود ؛ به این معنا است که از نود شروع ، هیچ مسیری به آن نود وجود ندارد .

گرفته‌شده‌است از [DIJKSTRA۲]

۳.۴ Order of Dijkstra Algorithm

- اگر از آرایه استفاده کنیم : $O(|V|+|V|^2 +|E|) = O(|V|^2)$
- اگر از باینری هیپ استفاده کنیم : $O((|V|+|E|)\log|V|)$

گرفته‌شده‌است از [book]

۴.۴ مزایای Dijkstra Algorithm :

- (۱) کمترین زمان برای رسیدن به خانه از محل کار را به دست آورد .
- (۲) پیدا کردن سریع ترین مسیر از محل کار به خانه .
- (۳) کوتاه ترین مسیر را هم در گراف جهت دار و هم بدون جهت پیدا کرد .

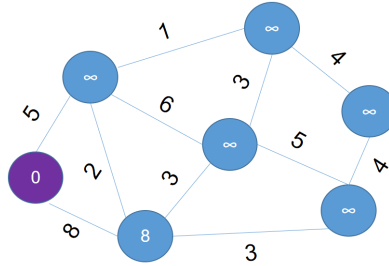
گرفته‌شده‌است از [book]

۵.۴ معایب Dijkstra Algorithm :

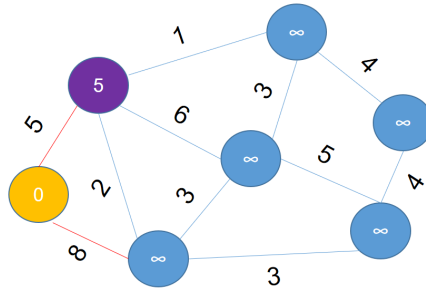
- (۱) با یک جست‌وجوی کورکورانه (blind search) روی منابع ، وقت زیادی را هدر می‌دهد .
- (۲) اگر گراف دارای یال منفی باشد ، این الگوریتم روی آن کار نمی‌کند .

گرفته‌شده‌است از [book]

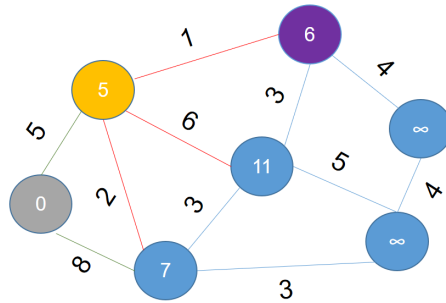
۶.۴ مثال :



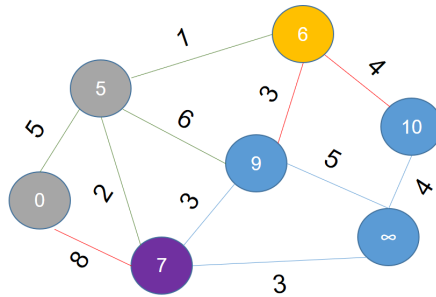
شکل ۱.۴: [DIJKSTRA] مرحله‌ی اول



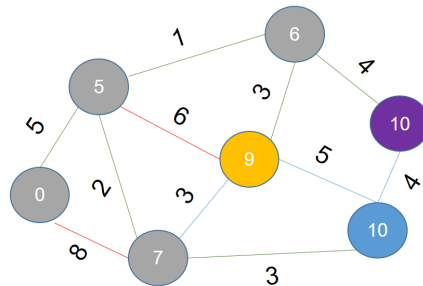
شکل ۲.۴: [DIJKSTRA] مرحله‌ی دوم



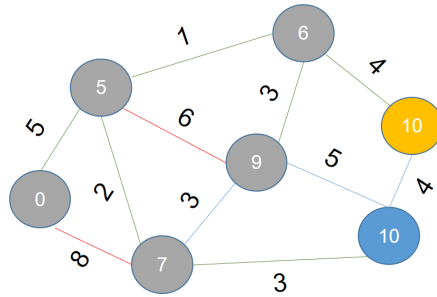
شکل ۳.۴: [DIJKSTRA] مرحله‌ی سوم



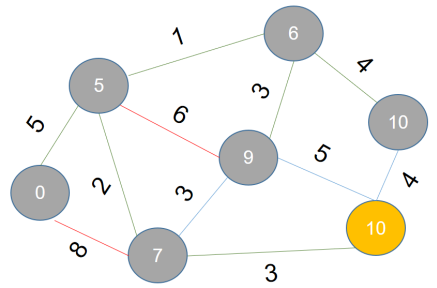
شکل ۴.۴: [DIJKSTRA] مرحله‌ی چهارم



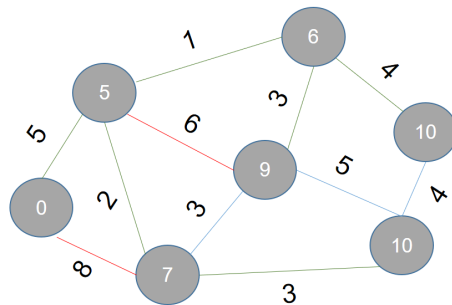
شکل ۵.۴: [DIJKSTRA] مرحله‌ی پنجم



شکل ۴.۶: [DIJKSTRA] مرحله ی ششم



شکل ۴.۷: [DIJKSTRA] مرحله ی هفتم



شکل ۴.۸: [DIJKSTRA] مرحله ی هشتم

۷.۴ شبیه‌کد Dijkstra Algorithm :

Data: $Relax((u, v) \in E)$
Result: relax the edges
if $dist[v] > dist[u] + w(u, v)$ **then**
 | $dist[v] \leftarrow dist[u] + w(u, v)$;
 | $Prev[v] \leftarrow u$;
else
end

Algorithm 2: Relax Method

Data: $Naive(G, S)$
Result: Find shortest path
for all $u \in V$ **do**
 | $dist[u] \leftarrow \infty$;
 | $prev[u] \leftarrow nil$;
end
 $dist[S] \leftarrow 0$;
while *at least one dist changes* **do**
 | relax all the edges ;
end

Algorithm 3: Naive Algorithm Of Dijkstra

$dist[v]$ فاصله ی واقعی نود شروع تا نود v است .

Relax چک می کند که آیا رفتن از نود شروع به نود v به وسیله u باعث کاهش $\text{dist}[v]$ می شود یا خیر .

```

Data: Dijkstra(G,S)
Result: Find Shortest Path
for all  $u \in V$  do
    |  $\text{dist}[u] \leftarrow \infty$  ;
    |  $\text{prev}[u] \leftarrow \text{nil}$  ;
end
 $\text{dist}[S] \leftarrow 0$  ;
 $H \leftarrow \text{MakeQueue}(V)$  dist-values as keys ;
while  $H$  is not empty do
    |  $u \leftarrow \text{ExtractMin}(H)$  ;
    | for all  $(u, v) \in E$  do
    | | if  $\text{dist}[v] > \text{dist}[u] + w(u, v)$  then
    | | |  $\text{dist}[v] \leftarrow \text{dist}[u] + w(u, v)$  ;
    | | |  $\text{Prev}[v] \leftarrow u$  ;
    | | |  $\text{ChangePriority}(H, v, \text{dist}[v])$  ;
    | | else
    | | end
    | end
end

```

Algorithm 4: Dijkstra Algorithm

گرفته شده است از [book]

۸.۴ اثبات درست بودن الگوریتم Dijkstra

وقتی که راس u را ExtractMin می کنیم، $\text{dist}[u]$ همان کمترین فاصله راس شروع از u است. اثبات: راسی که اکسترکت شود، کمترین فاصله را نسبت به راس هایی که هنوز اکسترکت نشده اند دارد. چون فاصله ی راس های اکسترکت نشده از آن بیشتر است، امکان ندارد در دفعات بعدی $\text{dist}[u]$ کمتر شود، چون اگر یکی از راسهای بعدی را v در نظر بگیریم، $\text{dist}[u]$ باید بیشتر از $\text{dist}[v] + \text{edge}(v, u)$ شود تا

آپدیت شود. در صورتی که $\text{dist}[u] \leq \text{dist}[v]$ است پس حتما $\text{dist}[v] + \text{edge}(v,u) \geq \text{dist}[u]$ است (در صورتی که یال منفی نداشته باشیم) و به همین دلیل $\text{dist}[u]$ دیگر آپدیت نمی‌شود. [book]

۹.۴ Bellman-Ford algorithm

برای پیدا کردن کوتاه‌ترین مسیر در بین نودهای یک گراف می‌توان از آن استفاده کرد. در Bellman Ford، مشابه به الگوریتم ساده‌ی Dijkstra (algorithm 3) عمل می‌کنیم. نحوه‌ی عملکرد الگوریتم:

- (۱) مثل Dijkstra (section 4.2) برای همه‌ی نودها یک متغیر فاصله در نظر می‌گیریم و مقدار اولیه‌ی آن را بی‌نهایت می‌گذاریم.
- (۲) انتخاب گره‌ی شروع و قرار دادن صفر برای مقدار فاصله‌ی آن
- (۳) یال‌هایی که به آن نود مربوط است را ویزیت می‌کنیم و آن‌ها را ریلکس می‌کنیم. (این مورد الزامی نیست و باعث سریع‌تر شدن الگوریتم می‌شود).
- (۴) یکی از همسایه‌های آن نودی که در آن قرار داریم را انتخاب می‌کنیم و مرحله‌ی دوم را برای آن اجرا می‌کنیم. لازم به ذکر است که از هیچ یالی دوبار حرکت نمی‌کنیم. (این مورد الزامی نیست و باعث سریع‌تر شدن الگوریتم می‌شود).
- (۵) باید همه‌ی یال‌ها را در هر دور ریلکس کنیم.
- (۶) این الگوریتم زمانی تمام می‌شود که مراحل یک تا پنج را به اندازه‌ی یکی کم‌تر از تعداد نودها تکرار شود.

گرفته‌شده‌است از [bell۱] [bell۲]

۱۰.۴ Order of Bellman-Ford Algorithm

اوردرد این الگوریتم برابر است با: $O(|V||E|)$ [book]

۱۱.۴ مزایای Bellman-Ford Algorithm :

- (۱) اگر گراف دارای یال منفی باشد، می‌تواند کوتاه‌ترین مسیر را بیابد.

- (۲) می‌تواند بهترین نرخ ممکن ارز را پیاده سازی کند.
- (۳) می‌توان با استفاده از آن implement infinite arbitrage (section 4.18) را عملی کرد .

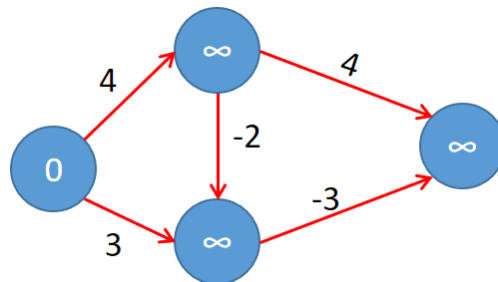
گرفته شده است از [book]

۱۲.۴ معایب Bellman-Ford Algorithm :

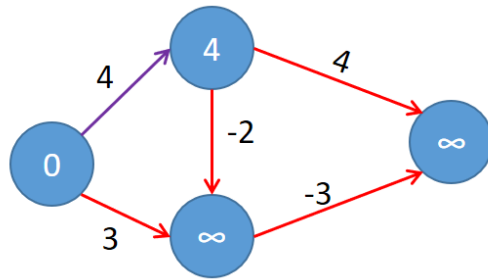
- (۱) مقیاس خوبی ندارد .
- (۲) تغییرات در توپولوژی شبکه به سرعت منعکس نمی‌شوند زیرا به روزسانی‌ها یال به یال پخش می‌شوند.
- (۳) اگر گراف دارای دور منفی باشد ، این الگوریتم کار نمی‌کند .
- (۴) الگوریتم Bellman-Ford (section 4.9) نسبت به الگوریتم Dijkstra (section 4.2) کندتر است .

گرفته شده است از [bell۱] [bell۳] [book]

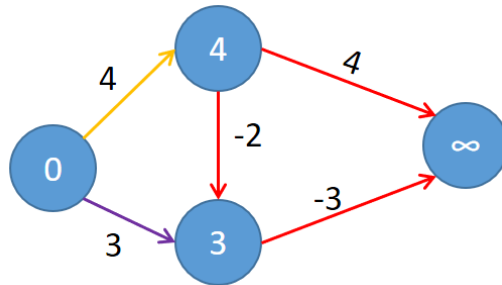
۱۳.۴ مثال :



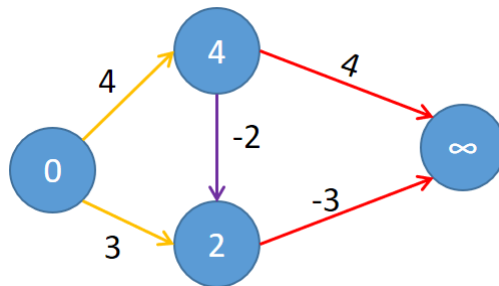
شکل ۹.۴: مرحله‌ی اول



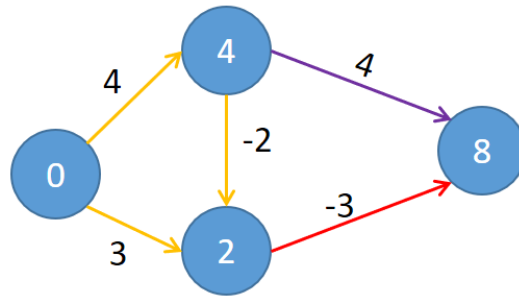
شکل ۱۰.۴: مرحله‌ی دوم



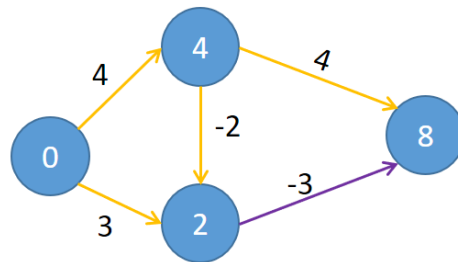
شکل ۱۱.۴: مرحله‌ی سوم



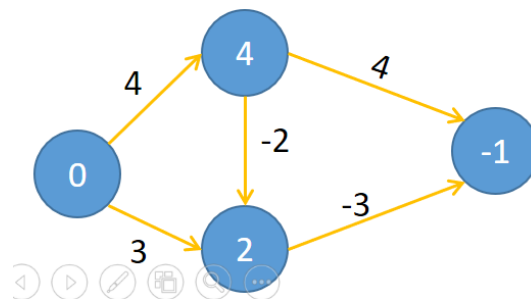
شکل ۱۲.۴: مرحله‌ی چهارم



شکل ۱۳.۴: مرحله‌ی پنجم



شکل ۱۴.۴: مرحله‌ی ششم



شکل ۱۵.۴: مرحله‌ی هفتم

مرحله‌ی هشتم به بعد این است که مراحل بالا به اندازه‌ی یکی کمتر از تعداد نودها باید تکرار شود .

۱۴.۴ شبه کد :

```

Data: BellmanFord(G,S)
Result: Find Shortest Path
no negative weight cycles in G
for all  $u \in V$  do
    | dist[u]  $\leftarrow \infty$  ;
    | prev[u]  $\leftarrow \text{nil}$  ;
end
dist[S]  $\leftarrow 0$ ;
repeat  $|V|-1$  times :
for all  $(u, v) \in E$  do
    | Relax(u,v) ;
end

```

Algorithm 5: Bellman-Ford Algorithm

گرفته شده است از [book]

۱۵.۴ اثبات درست بودن الگوریتم Bellman-Ford

بعد از k بار relaxation (algorithm 2) ، همه ی کوتاه ترین فاصله ها از راس شروع که حداکثر شامل k یال هستند؛ مشخص شده اند.
با استفاده از استقرای ریاضی:

- ۱. اگر $k=0$ فاصله همه راس ها از راس شروع بی نهایت است، به غیر از خود راس شروع که .
 $\text{dist}[S]=0$
- ۲. فرض استقرا: بعد از k بار relaxation همه ی کوتاه ترین مسیرها با طول حداکثر k مشخص شده اند.
- ۳. حکم استقرا: قبل از $k+1$ بار relaxation ، $\text{dist}[u]$ با کم ترین فاصله به طول حداکثر k مشخص شده است. اگر از u یال هایی به راس های دیگر باشد، در دفعه ی $k+1$ همه ی آنها relax می شوند، پس کوتاه ترین مسیرها با حداکثر طول $k+1$ (کوتاه ترین مسیرها با حداکثر طول k بعلاوه یک یال که آنها را به راس دیگری وصل کند) مشخص می شوند.

گرفته شده است از [book]

۱۶.۴ دور منفی در Bellman-Ford :

- (۱) Bellman-Ford Algorithm (section 4.9) را به اندازه ی تعداد نودها اجرا می‌کنیم. نودهایی که در دور آخر ریلکس می‌شوند را در یک کیو یا لیست ذخیره می‌کنیم.
- (۲) از $x \leftarrow v$ شروع کنیم، مسیر $x \leftarrow \text{prev}[x]$ را به اندازه ی نودها تکرار می‌کنیم. به طور قطع در چرخه خواهد بود.
- (۳) $y \leftarrow x$ را ذخیره می‌کنیم و $x \leftarrow \text{prev}[x]$ را ادامه دهید تا به $y = x$ برسیم.

گرفته شده است از [book]

۱۷.۴ اثبات درست بودن الگوریتم دور منفی در Bellman-Ford

یک گراف دارای دور با وزن منفی است اگر و تنها اگر در دفعه ی $|V|$ ام (تعداد نودها) از relaxation (algorithm 2) همه یال‌ها، حداقل یکی از dist ها آپدیت شوند.

اثبات : (اثبات اینکه اگر در بار $|V|$ ام یکی از فاصله‌ها آپدیت شد، آن گراف حتما دوری با وزن منفی دارد) اگر گرافی دارای دور با وزن منفی نباشد، کوتاه‌ترین مسیرها از راس شروع حداکثر طول $|V|-1$ را دارند. زیرا اگر مسیری طولش بیش‌تر یا مساوی $|V|$ باشد، آن مسیر دارای یک دور است، و اگر وزن کلی دورش منفی نباشد، وجودش تنها طول آن مسیر و فاصله را بیش‌تر می‌کند و برای داشتن کوتاه‌ترین مسیر باید از آن حذف شود. پس هیچ فاصله ای در دفعه $|V|$ ام نباید آپدیت شود (طبق اثبات قبلی مسیری که در دفعه $|V|$ آپدیت شود یعنی طول آن مسیر $|V|$ است).

اثبات : (اثبات اینکه اگر دارای دور با وزن منفی باشد حتما در بار $|V|$ ام یکی از فاصله ها آپدیت می‌شوند) فرض کنید گرافی با دور با وزن منفی داریم مثلا $a \rightarrow b \rightarrow c \rightarrow a$ اما در دفعه ی $|V|$ ام relaxation یال ها، هیچ یالی relax نشود. برای اینکه هیچکدام relax نشوند، باید فاصله‌ای که هر راس دارد از فاصله ی راس مجاور بعلاوه ی یال بین آن دو بزرگ‌تر باشد، حتی برای سه راس a, b, c ، پس داریم:

$$\text{dist}[b] \leq \text{dist}[a] + w(a,b)$$

$$\text{dist}[c] \leq \text{dist}[b] + w(b,c)$$

$$\text{dist}[a] \leq \text{dist}[c] + w(c,a)$$

با جمع کردن این سه معادله: $w(a,b) + w(b,c) + w(c,a) \geq 0$

درحالی که دور بین این سه راس مجموع وزنش باید منفی باشد، پس به تناقض خوردیم، وحتما حداقل یکی از

یالها باید relax شود. [book]

۱۸.۴ : Infinite Arbitrage

- (۱) به اندازه ی تعداد نودها Bellman-Ford Algorithm (section 4.9) را انجام می‌دهیم .
نودهایی که در دور آخر ریلکس می‌شوند را در لیست یا کیو ذخیره می‌کنیم .
- (۲) برای نودهایی که ذخیره کردیم ، الگوریتم بی اف اس (BFS) [bfs] را انجام می‌دهیم .
- (۳) نودهایی که از سری ذخیره شده‌ی اولیه قابل دسترس اند دارای infinite arbitrage هستند .

چند نکته:

- در بی اف اس (BFS) [bfs] نودهایی که قبلا ویزیت شده‌اند را کاری نداریم .
- از نودی که در دور آخر ریلکس شده است، می‌توان دور منفی را یافت و از دور منفی برای به دست آوردن infinite arbitrage استفاده کرد.

گرفته‌شده است از [book]

۱۹.۴ اثبات درست بودن الگوریتم Infinite Arbitrage

(Infinite Arbitrage) فاصله u از s منفی بینهایت است اگر و تنها اگر از راسی که در بار $|V|$ ام بلمن-فورد فاصله اش تغییر کرده به آن مسیری وجود داشته باشد. (فاصله منفی بی‌نهایت یعنی می‌توان فاصله آن را کم‌تر و کم‌تر کرد)

اثبات سمت \Leftarrow از اگر و تنها اگر (اثبات این که اگر از آن راس به u مسیری باشد پس فاصله از s منفی بی‌نهایت است).

اگر راسی که فاصله اش در دفعه ی $|V|$ ام تغییر کرده را w به وسیله یک دور با وزن منفی به راس شروع متصل شده است. از آنجا که مسیر از s به w از یک دور با وزن منفی می‌گذرد، و از w نیز مسیری به u وجود دارد. پس با استفاده از دور با وزن منفی، می‌توانیم فاصله u از s کاهش دهیم.

اثبات سمت \Rightarrow از اگر و تنها اگر (اثبات این که اگر فاصله u از s منفی بی‌نهایت باشد، از راس w که در بار $|V|$ ام بلمن-فورد فاصله اش تغییر کرده، به u مسیری وجود دارد)

فرض کنیم بعد از $V-1$ بار اجرای بلمن-فورد، $\text{dist}[u]=L$ باشد. از آنجا که فاصله ی u از s می‌تواند کم‌تر و کم‌تر شود، پس در بعضی از دفعات بعدی اجرای بلمن-فورد مثلا $k > V$ فاصله اش تغییر می‌کند. اگر

راسی در دفعه i ام از relaxation (algorithm 2) یال‌ها فاصله‌اش تغییر نکند، همه یال‌هایی که از آن راس آغاز می‌شوند نیز در دفعه $i+1$ ام relax نمی‌شوند (راس ابتدایی‌شان تغییری نکرده که راس انتهاییشان را تغییر بدهند). پس برای اینکه راسی فاصله‌اش تغییر کند، باید راسی دیگر در مسیر بین آن و راس شروع قبلا تغییر کرده باشد. پس برای اینکه $\text{dist}[u] \geq V$ در $k \geq V$ تغییر کند، راسی پیش از آن تغییر کرده و از آن راس به u مسیری وجود داشته باشد. پس از راسی که در دفعه V ام تغییر کرده است، به u مسیر وجود داشته است. [book]

۲۰.۴ سایت‌های دیگر برای مراجعه :

- در این سایت می‌توانید نحوه‌ی عملکرد الگوریتم دایجسترا را ببینید . همراه با مثال‌های متعدد :
لینک اول
- در این سایت می‌توانید مثال و کد الگوریتم بلمن-فورد را مشاهده کنید : **لینک دوم**
- در این سایت می‌توانید مثال و شبه‌کد و کد الگوریتم بلمن-فورد را مشاهده کنید : **لینک سوم**
- در این سایت می‌توانید مثال و توضیح و کد و ویدیوی مرتبط با الگوریتم دایجسترا را بررسی کنید :
لینک چهارم
- در این سایت می‌توانید توضیح همراه با مثال برای الگوریتم دایجسترا را مشاهده کنید : **لینک پنجم**
- در این سایت می‌توانید شبه‌کد و مرحله، مرحله انجام‌شدن شبه‌کد برای مثال در الگوریتم دایجسترا را مشاهده کنید : **لینک ششم**
- در این سایت می‌توانید مثال و کد و توضیح برای دور منفی در بلمن-فورد را مشاهده کنید : **لینک هفتم**
- پی‌دی‌اف‌ی برای دور منفی در بلمن-فورد همراه با مثال و توضیح کامل: **لینک هشتم**
- سایت کورسرا برای Infinite Arbitrage : **لینک نهم**

جلسه ۵

Bellman Ford و Dijkstra

یاسمن لطف‌اللهی - ۱۳۹۸/۱۱/۲۹

جزوه جلسه ۵ ام مورخ ۱۳۹۸/۱۱/۲۹ درس طراحی و تحلیل الگوریتم تهیه شده توسط یاسمن لطف‌اللهی. در جلسات گذشته، با دو الگوریتم Dijkstra و Bellman Ford برای پیدا کردن کوتاه‌ترین راه در یک گراف، آشنا شدیم. در این جلسه به جزئیات این دو الگوریتم خواهیم پرداخت.

۱.۵ شبه‌کد Dijkstra

در ابتدا، مقدار $dist$ را برای تمام رأس‌های گراف به جز رأس مبدا $+$ ∞ ، و مقدار $prev$ را برای تمام رأس‌ها $null$ در نظر می‌گیریم. سپس از تمام رأس‌ها، یک **Priority Queue** می‌سازیم، به طوری که الویت هر رأس، مقدار $dist$ آن باشد. رأسی را که کمترین $dist$ را دارد، از **Priority Queue** خارج می‌کنیم و یال‌های مجاورش را $relax$ می‌کنیم. این کار را تا زمانی که **Priority Queue** خالی نشده، تکرار می‌کنیم. در آخر، آرایه $dist$ کمترین فاصله بین هر رأس و رأس مبدا را مشخص می‌کند و به کمک آرایه $prev$ ، می‌توانیم کوتاه‌ترین راه به هر رأس را پیدا کنیم.

Data: Graph G, Node Source

Result: Finding the shortest paths between Source and all other nodes in Graph G

dist[] \leftarrow an array with size of |V| filled with $+\infty$;

prev[] \leftarrow an array with size of |V| filled with null;

dist[S] \leftarrow 0;

H \leftarrow MakeQueue(V);

while H is not empty **do**

 u \leftarrow ExtractMin(H);

for all (u, v) in E **do**

if dist[v] > dist[u] + w(u, v) **then**

 dist[v] \leftarrow dist[u] + w(u, v);

 prev[v] \leftarrow u;

 ChangePriority(H, v, dist[v]);

end

end

end

Algorithm 6: Dijkstra Algorithm

۲.۵ شبکه‌کد پیدا کردن کوتاه‌ترین راه بین دو راس به کمک آرایه prev

Data: Graph G, Node Source, Node x, Node[] prev

Result: Finding the shortest path between Source and x in Graph G

Node[] path;

Node n \leftarrow x;

while n \neq S **do**

 path.add(n);

 n \leftarrow prev[n];

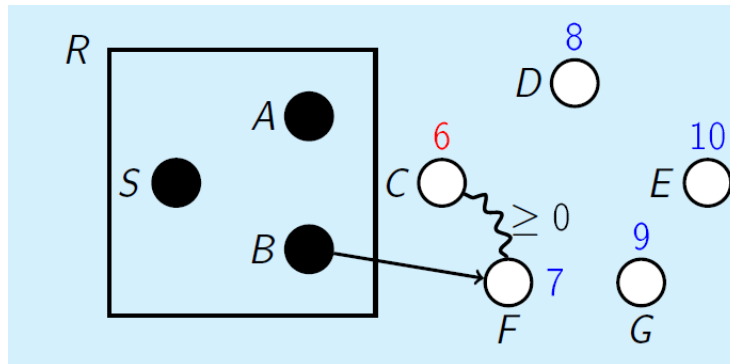
end

Algorithm 7: Finding Shortest Path

- پیچیدگی زمانی: در گرافی که دور منفی وجود نداشته باشد، کوتاه‌ترین راه از تمام رأس‌ها حداکثر یک بار عبور می‌کند. پس پیچیدگی زمانی این شبکه‌کد برابر $O(V)$ است.

۳.۵ اثبات درستی الگوریتم Dijkstra

مثال زیر را در نظر بگیرید:



شکل ۱.۵: رأس‌هایی که در مربع قرار دارند، از قبل انتخاب و بررسی شده‌اند.

طبق الگوریتم، وقتی رأسی از طریق ExtractMin انتخاب می‌شود، به این معناست که کمترین فاصله بین آن رأس و رأس مبدا برابر dist آن رأس است. پس در این مثال، رأس C انتخاب می‌شود و نتیجه گرفته می‌شود که کمترین فاصله بین C و S برابر ۶ است. فرض می‌کنیم که این عبارت درست نیست و کمترین فاصله بین این دو رأس از ۶ کمتر است. بنابراین رأسی مانند F وجود دارد که در کوتاه‌ترین مسیر بین S و C قرار دارد. اما مقدار dist این رأس بیشتر از ۶ است و یال بین C و F نامنفی است. پس فاصله این مسیری که از F می‌گذرد، نمی‌تواند کمتر از ۶ باشد که با فرض اولیه تناقض دارد.

۴.۵ پیچیدگی زمانی الگوریتم Dijkstra

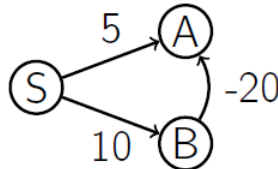
طبق شبکه‌کد ۶، پیچیدگی زمانی این الگوریتم برابر است با مجموع پیچیدگی‌های زمانی این سه قسمت:

- ساختن Priority Queue : $T(\text{MakeQueue})$
- خارج کردن تمام رأس‌ها از Priority Queue : $|V| \cdot T(\text{ExtractMin})$
- relax کردن یال‌ها (هر یال، حداکثر یک بار relax می‌شود) : $|E| \cdot T(\text{ChangePriority})$

اگر Priority Queue با آرایه پیاده‌سازی شده باشد، پیچیدگی زمانی برابر می‌شود با $O(V^2)$ ، و اگر با Heap پیاده‌سازی شده باشد، پیچیدگی زمانی برابر می‌شود با $O((V + E) \log V)$.

۵.۵ چرا Dijkstra برای گراف‌هایی که یال منفی دارند، کار نمی‌کند؟

در الگوریتم Dijkstra فرض می‌شود که کوتاه‌ترین مسیر بین دو رأس S و T، از رأس‌هایی می‌گذرد که به S نزدیک‌ترند. اما این فرض در صورت وجود یال منفی، صادق نیست. به مثال زیر توجه کنید:



شکل ۲.۵: محاسبه کمترین فاصله بین S و A با الگوریتم Dijkstra

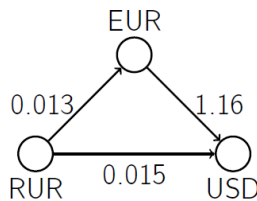
در این مثال، رأس A به رأس S نزدیک‌تر است. اما کوتاه‌ترین راه به S، از رأس B که از S دورتر است، می‌گذرد.

۶.۵ چرا اضافه کردن به یال‌ها و استفاده از Dijkstra جواب نمی‌دهد؟

وقتی به تمام یال‌ها، یک مقدار ثابتی اضافه می‌کنیم، به مسیری که از تعداد یال بیشتری تشکیل شده‌اند، مقدار بیشتری اضافه می‌شود.

۷.۵ یک مثال از کاربرد الگوریتم Bellman Ford : تبدیل ارز

مسئله: گراف زیر، نرخ تبدیل ارزهای مختلف به یکدیگر را نشان می‌دهد. فرض کنید می‌خواهیم مقداری دلار را به یورو تبدیل کنیم. چگونه این کار را انجام بدهیم تا بیشترین مقدار یورو را به دست بیاوریم؟



شکل ۳.۵: گراف تبدیل ارز

ابتدا به جای هر یال، لگاریتم آن یال را قرار می‌دهیم. در این صورت، به جای محاسبه حداکثر حاصل ضرب تبدیل‌ها، حداکثر مجموع لگاریتم آن‌ها را حساب می‌کنیم.

$$\prod_{j=1}^k r_{e_j} \rightarrow \max \Leftrightarrow \sum_{j=1}^k \log(r_{e_j}) \rightarrow \max$$

شکل ۴.۵: تبدیل به لگاریتم

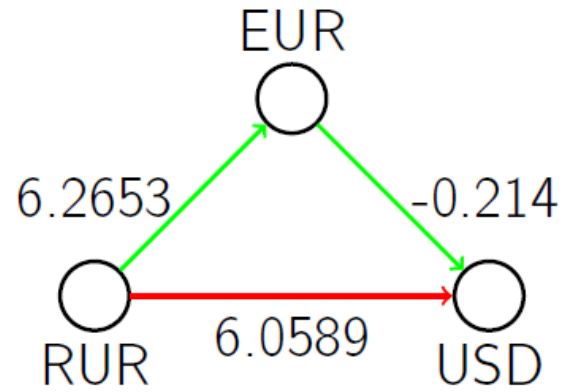
سپس هر یال را قرینه می‌کنیم. در این صورت، کافی است حداقل مجموع قرینه لگاریتم یال‌ها، محاسبه شود.

$$\sum_{j=1}^k \log(r_{e_j}) \rightarrow \max \Leftrightarrow - \sum_{j=1}^k \log(r_{e_j}) \rightarrow \min$$

$$\sum_{j=1}^k \log(r_{e_j}) \rightarrow \max \Leftrightarrow \sum_{j=1}^k (-\log(r_{e_j})) \rightarrow \min$$

شکل ۵.۵: قرینه کردن لگاریتم‌ها

برای محاسبه این مقدار، باید کمترین فاصله بین دو ارز را در گراف متناظر پیدا کنیم.



شکل ۶.۵: گراف تبدیل ارز

در این مسئله، چون احتمال داشتن یال منفی وجود دارد، از الگوریتم Bellman Ford استفاده می‌شود، نه Dijkstra.

۸.۵ شبه‌کد Bellman Ford

Data: Graph G, Node Source

Result: Finding the shortest paths between Source and all other nodes
in Graph G

dist[] \leftarrow an array with size of $|V|$ filled with $+\infty$;

prev[] \leftarrow an array with size of $|V|$ filled with null;

dist[S] \leftarrow 0;

H \leftarrow MakeQueue(V);

for $i = 0$; $i < |V| - 1$; $i = i + 1$ **do**

for all (u, v) in E **do**

 Relax(u, v);

end

end

Algorithm 8: Bellman Ford Algorithm

• پیچیدگی زمانی Bellman Ford: $O(|V||E|)$

۹.۵ اثبات درستی الگوریتم Bellman Ford

طبق این الگوریتم، بعد از k بار اُمی که تمام یال‌ها را relax کردیم، $\text{dist}[u]$ نشان‌دهنده اندازه کوتاه‌ترین مسیر بین u و S با حداکثر k یال است. برای اثبات این عبارت با استفاده از استقرا، باید ثابت کنیم که این عبارت برای $k + 1$ نیز صادق است. طبق فرض استقرا، کوتاه‌ترین مسیر بین u و S و بین v و S ، حداکثر از k یال تشکیل شده. در relax کردن بار $k + 1$ اُم، یال (u, v) relax می‌شود. در این صورت، کوتاه‌ترین مسیر بین u و S ، یا همان مسیر قبلی با حداکثر k یال باقی می‌ماند، یا به مسیر بین v و S با حداکثر k یال، به علاوه یال (u, v) ، تغییر پیدا می‌کند. در هر دو مورد، مسیر بین u و S بعد از relax کردن بار $k + 1$ اُم، حداکثر از $k + 1$ یال تشکیل شده و این الگوریتم اثبات می‌شود.

جلسه ۶

دور منفی در گراف و دایجسترا دو طرفه

ملیکا نوبختیان - ۱۳۹۸/۱۲/۴

۱.۶ قضیه

گراف G دارای یک دور با وزن منفی است اگر و فقط اگر در V امین تکرار از الگوریتم بلمن فورد روی گراف G و شروع از گره S تعدادی از فاصله ها تغییر کنند.

۲.۶ اثبات

اگر در گراف هیچ دور منفی وجود نداشته باشد، همه کوتاه ترین مسیرها از S حداکثر دارای $|V|-1$ یال خواهند بود (هر مسیری که دارای تعداد بیشتر یا مساوی V یال باشند، منفی نیستند و می توانند از کوتاه ترین مسیرها حذف شوند) بنابراین هیچ فاصله ای در V امین تکرار تغییر نخواهد کرد. در حالت دیگر می دانیم در گراف دور منفی وجود دارد و این دور به این صورت است:

$$a \rightarrow b \rightarrow c \rightarrow a$$

اما در V امین تکرار هیچ تغییری ایجاد نمی شود.

$$\text{dist}[b] \leq \text{dist}[a] + w(a, b)$$

$$\text{dist}[c] \leq \text{dist}[b] + w(b, c)$$

$$\text{dist}[a] \leq \text{dist}[c] + w(c, a)$$

می دانیم که دور شامل این سه گره منفی است پس جمع وزن های این سه یال منفی می شود در حالی با توجه به سه عبارتی که در بالا نوشته شده است به تناقض می رسیم پس حتما در V امین تکرار در یک گراف با دور منفی حتما تعدادی از فاصله ها تغییر می کنند.

$$w(a, b) + w(b, c) + w(c, a) \geq 0$$

۳.۶ Finding Negative Cycle

برای پیدا کردن دور منفی در یک گراف به مراتب زیر عمل می کنیم:

- $|V|$ بار الگوریتم بلمن فورد را روی گراف اجرا می کنیم و گره v را که در بار $|V|$ ام ریلکس شده است را ذخیره می کنیم.
- از دور منفی قابل دسترسی است
- از $x \leftarrow v$ شروع می کنیم، $x \leftarrow \text{prev}[x]$ را $|V|$ بار انجام می دهیم، در نهایت به داخل حلقه خواهیم رسید.
- $y \leftarrow x$ را ذخیره می کنیم و $x \leftarrow \text{prev}[x]$ را تا جایی انجام می دهیم که به $y = x$ برسیم.
- با ذخیره کردن گره های در این مسیر دور منفی بدست آمده است.

با استفاده از قطعه کدی که در ادامه آمده است می توانیم پی ببریم که آیا گراف ما دارای دور منفی است یا خیر:

```

۱ public bool HasNegativeCycle(Graph G, int Start, long N)
۲ {
۳     long[] dist = new long[N];
۴     dist[Start] = 0;
۵     for (int i = 0; i < N - 1; i++)
۶     {
۷         foreach (edge e in Graph.edges)
۸             relax(e);
۹     }
۱۰    foreach (edge e in Graph.edges)
۱۱    {
۱۲        if (relax(e))
۱۳            return true;
۱۴    }
۱۵    return false;
۱۶ }

```

نمونه کد ۱: تابع تشخیص وجود دور منفی در گراف

با کمی تغییر در ساختار متد قبلی می توانیم گره هایی را که در بار $|V|$ ام انجام بلمن فاصله شان تغییر پیدا می کند را ذخیره کنیم و با استفاده از متد زیر دورهای منفی گراف را بیابیم:

```

۱ List<long> FindNegCycle(int v, long[] Prev, int N)
۲ {
۳     long x = v;
۴     for (int i = 0; i < N; i++)
۵         x = Prev[x];
۶     List<long> cycle = new List<long>();
۷     long y = Prev[x];
۸     while (y != x)
۹     {
۱۰        cycle.Add(y);
۱۱        y = Prev[y];
۱۲    }
۱۳    return cycle;
۱۴ }

```

نمونه کد ۲: تابع پیدا کردن دور منفی در گراف

هر چند وجود دور منفی در گراف تبدیل ارز می تواند ما را به هر مقدار پول که می خواهیم برساند اما اگر از دور منفی موجود در گراف مسیری به ارز مبدا وجود نداشته باشد این کار ممکن نخواهد بود.

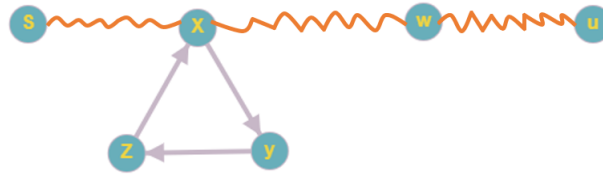
۴.۶ Infinite Arbitrage

۱.۴.۶ قضیه

این امکان وجود دارد که هر مقدار پول از ارز u با شروع از ارز S بدست آورید اگر و فقط اگر گره u قابل رسیدن از گره w که در $|V|$ امین تکرار بلمن فورد فاصله آن تغییر کرده است باشد.

۲.۴.۶ اثبات

اگر $\text{dist}[w]$ در $|V|$ امین تکرار از بلمن فورد کاهش یافته باشد، از دور منفی داخل گراف قابل دسترسی است. [createon]



شکل ۱.۶: Infinite Arbitrage

چون w از دور منفی قابل دسترسی است پس u هم قابل دسترسی است. فرض می‌کنیم L طول کوتاه ترین مسیر به u با حداکثر $V-1$ یال باشد. بعد از $V-1$ تکرار $\text{dist}[u]$ برابر L خواهد بود. برای داشتن Infinite Arbitrage به u یک مسیر کوچک تر از L وجود خواهد داشت. بنابراین $\text{dist}[u]$ در تکراری که $k \geq V$ باشد کاهش خواهد یافت. اگر یال (x, y) ریلکس نشود و $\text{dist}[x]$ در i امین تکرار کاهش نیابد، پس یال (x, y) در $i+1$ امین تکرار هم ریلکس نخواهد شد. تنها گره هایی که گره های ریلکس شده در تکرار قبلی قابل دسترسی هستند می‌توانند ریلکس شوند. اگر $\text{dist}[u]$ اگر $\text{dist}[u]$ در یک تکرار $k \geq V$ کاهش یابد، u از گره هایی که V امین تکرار ریلکس شده اند قابل دسترسی است.

۳.۴.۶ Detect Infinite Arbitrage

برای پیدا کردن گره هایی که در Arbitrage Infinite هستند به صورت زیر عمل می‌کنیم:

- V بار الگوریتم بلمن فورد را اجرا می‌کنیم و گره هایی که در بار V ام ریلکس می‌شوند را در مجموعه A ذخیره می‌کنیم.

- همه گره هایی که در مجموعه A قرار دارند را در صف Q قرار می دهیم.
- BFS را روی اعضای Q انجام می دهیم تا همه گره هایی که از A قابل دسترسی هستند را بدست آوریم.
- فقط و فقط این گره ها دارای Infinite Arbitrage هستند.

با استفاده از متد زیر گره هایی که دارای Infinite Arbitrage هستند را پیدا می کنیم:

```

۱ public List<long> DetectInfiniteArbitrage(Graph G, int Start, long N)
۲ {
۳     long[] dist = new long[N];
۴     dist[Start] = 0;
۵     Queue<long> relaxed = new Queue<long>();
۶     List<long> Arbitrage = new List<long>();
۷     for (int i = 0; i < N - 1; i++)
۸         foreach (edge e in Graph.edges)
۹             relax(e);
۱0    foreach (edge e in Graph.edges)
۱1        if (relax(e))
۱2            relaxed.Enqueue(e.target);
۱3    foreach (var v in relaxed)
۱4    {
۱5        List<long> nodes = BFS(G, v);
۱6        foreach (var u in nodes)
۱7            if (!Arbitrage.Contains(u))
۱8                Arbitrage.Add(u);
۱9    }
۲0    return Arbitrage;
۲۱ }

```

نمونه کد ۳: Detect Infinite Arbitrage

۴.۴.۶ Reconstruct Infinite Arbitrage

- در طول BFS ، parent هر گره را به خاطر می سپاریم.
- مسیر به گره u را از گره ای مانند w که در تکرار V ام ریلکس شده است را دوباره می سازیم.
- از w برمی گردیم تا دوری منفی که w از آن قابل دسترسی است را پیدا کنیم.
- از دور منفی استفاده می کنیم تا به Infinite Arbitrage که از S به u دست پیدا کنیم.

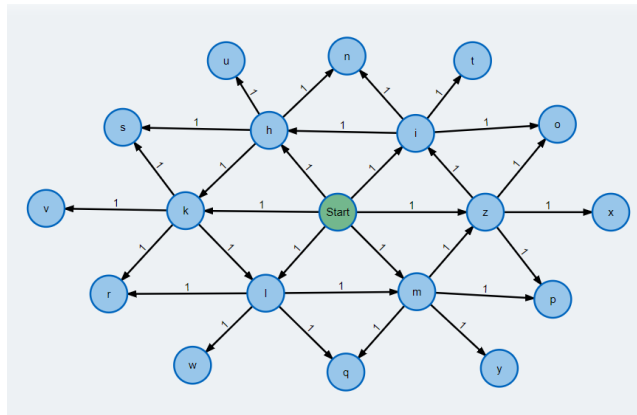
Bidirectional Search ۵.۶

Why not just Dijkstra ۱.۵.۶

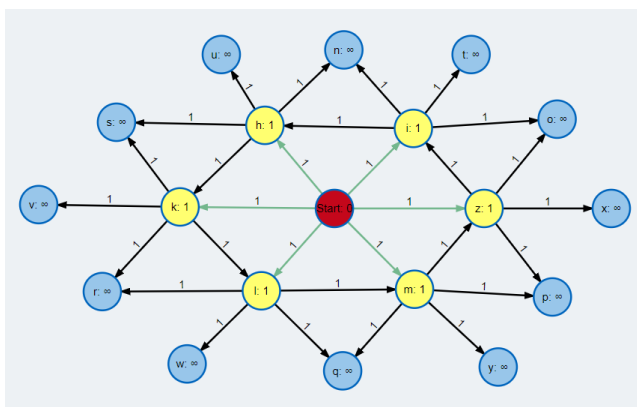
برای بدست آوردن کوتاه ترین مسیر از گره ای به گره دیگر گاهی اوقت دایجسترا الگوریتم مناسبی نیست ولی چرا؟ پیچیدگی زمانی دایجسترا $O((|V|+|E|)\log|V|)$ است که نسبتا سریع است پس چرا ممکن است گاهی به اندازه کافی خوب نباشد؟

- برای گراف آمریکا با بیست میلیون گره و پنجاه میلیون یال، دایجسترا به طور متوسط چند ثانیه طول خواهد کشید تا اجرا شود.
- در حالی که میلیون ها کاربر Google Maps می خواهند در یک چشم به هم زدن به نتیجه برسند.
- پس نیاز به الگوریتمی سریع تر خواهیم داشت.

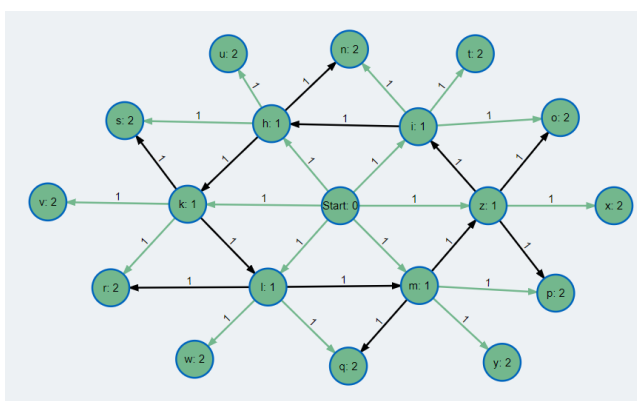
در شکل های زیر نحوه پیشرفت الگوریتم دایجسترا روی یک گراف را می بینیم و خواهیم فهمید که چرا ممکن است سریع عمل نکند: [dijder]



شکل ۲.۶: دایجسترا (۱)



شکل ۳.۶: دایجسترا (۲)



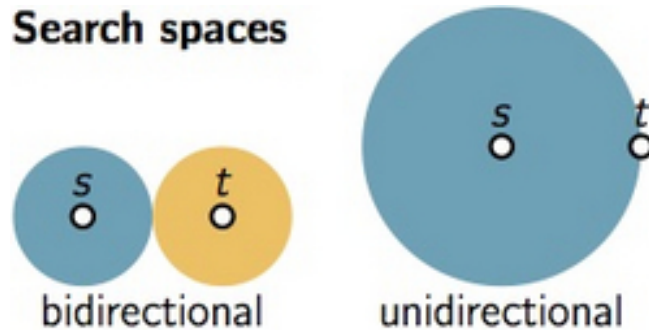
شکل ۴.۶: دایجسترا (۳)

۲.۵.۶ Growing Circle

قضیه: وقتی که راس v از طریق Extract Min انتخاب می شود، فاصله u برابر قطر s تا u است.

$$(\text{dist}[u]=d(s,u))$$

اثبات: وقتی که یک راس از priority queue برای پردازش کردن خارج می شود، همه راس هایی که فاصله کمتری داشته اند قبلاً پردازش شده اند. دایره راس های پردازش شده هر بار بزرگ تر می شود.



شکل ۵.۶: dijkstra vs bidirectional dijkstra

با توجه به شکل بالا می‌توانیم مقدار فضا و سطحی که هر دو نوع دایجسترا پوشش می‌دهند را ببینیم. اگر فاصله s تا t را $2r$ در نظر بگیریم، با محاسبه مساحت هر دو شکل در خواهیم یافت که فضای اشغال شده توسط **bidirectional** نصف نظیر آن برای دایجسترا معمولی خواهد بود.

این الگوریتم برای نقشه‌های جاده‌ای نسبتاً خوب عمل می‌کند و سرعت پیدا کردن را تقریباً دو برابر می‌کند. اما باید برای گراف شبکه‌های اجتماعی هم بررسی شود.

۳.۵.۶ Six Handshakes

در سال ۱۹۲۹، فریگیس کاریتنی، ریاضیدان مجارستانی، فرضیه‌ای به نام دنیای کوچک را مطرح کرد. بر اساس این فرضیه شما حداکثر با ۶ بار دست به دست کردن یک پیام می‌توانید آن را به هر کسی در سراسر دنیا برسانید. این فرضیه بر اساس آزمایش‌های مختلف نزدیک به حقیقت است.

۴.۵.۶ Facebook

اگر فرض کنیم که هر نفر در فیسبوک به طور متوسط ۱۰۰ دوست در اطراف خود دارد پس دوستان او ۱۰۰۰۰ دوست دیگر خواهند داشت. اگر فرضیه small world قدم به قدم ادامه دهیم در ششمین واسطه به یک تریلیون انسان خواهیم رسید در حالی که در نهایت ۷ میلیارد انسان در کره زمین وجود دارد پس این روش برای شبکه اجتماعی ممکن نیست.

می‌خواهیم کوتاه‌ترین مسیر بین باب و مایکل از طریق ارتباطات دوستی پیدا کنیم. برای دورترین مردم دایجسترا حداکثر بین ۲ میلیون نفر جستجو می‌کند. اگر ما در دوستان دوستان دوستان باب و مایکل جستجو کنیم یک راه ارتباطی پیدا خواهیم کرد. حداکثر بین یک میلیون دوست هر کدام جستجو خواهیم کرد که حداکثر

جستجو به طور کلی بین دو میلیون نفر خواهد بود که ۱۰۰۰ بار از روش قبلی سریع تر است.

این روش را می توانیم نه تنها در گراف بلکه در هر مورد دیگر هم به کار بگیریم. با توجه به مثال قبلی پیچیدگی زمانی از $O(n)$ به $O(\sqrt{n})$ تغییر خواهد کرد.

۶.۶ Bidirectional Dijkstra

ابتدا بهتر است دایجسترا را یادآوری کنید. می توانید از این لینک کمک بگیرید [onlinedij]

۱.۶.۶ Reversed Graph

گراف معکوس G^R برای گراف G گرافی است با همان مجموعه از راس ها و مجموعه ای از یال های معکوس E^R به طوری که برای هر یال $(u, v) \in E$ یک یال $(v, u) \in E^R$ وجود خواهد داشت و بالعکس.



شکل ۶.۶: Reversed Graph

۲.۶.۶ Algorithm

- گراف G^R را بسازید.
- دایجسترا را از s در گراف G و از t در گراف G^R شروع کنید.
- به نوبت مراحل دایجسترا برای G و G^R انجام دهید.
- وقتی که راسی مانند v در دو گراف G و G^R پردازش شد این عملیات را متوقف می کنیم.
- کوتاه ترین مسیر بین s و t را حساب می کنیم.

۳.۶.۶ Computing Distance

فرض می‌کنیم v اولین راسی باشد که هم در G و هم در G^R پردازش شده باشد. آیا از آن پیروی می‌کند که کوتاه‌ترین مسیری که از s به t وجود دارد از v می‌گذرد؟



شکل ۷.۶: Compute Distance

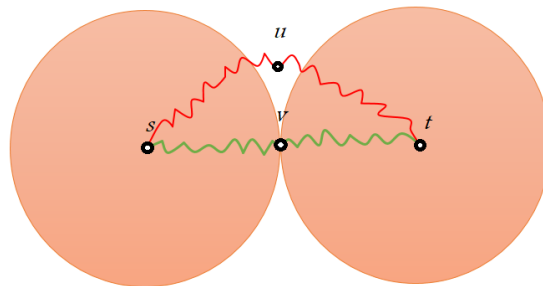
۴.۶.۶ قضیه

فرض کنید $dist[u]$ فاصله‌ای است که در دایجسترا از s در گراف G تخمین زده می‌شود و $dist^R[u]$ به طور یکسان در دایجسترا از t در گراف G^R باشد. کوتاه‌ترین مسیر از s تا t از گره‌ای مانند u می‌گذرد که یا در G یا در G^R و یا در هر دو آن‌ها پردازش شده است و خواهیم داشت:

$$d(s, t) = dist[u] + dist^R[u]$$

۵.۶.۶ اثبات

فرض می‌کنیم راسی مانند u که در خارج از دایره دایجسترا چه در G و چه در G^R باشد و کوتاه‌ترین مسیر ما از این راس می‌گذرد در حالی که هنوز پردازش نشده است. در اینجا به تناقض می‌رسیم چون اگر راس u در فاصله کمتری از s یا t قرار داشت باید زودتر پردازش می‌شد.



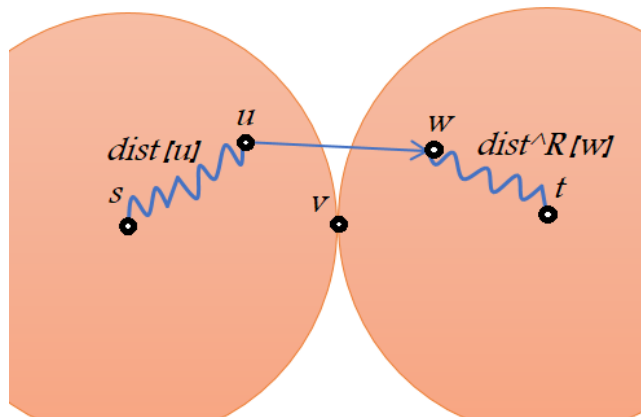
شکل ۸.۶: Proof

$$\begin{aligned}
 d(s, u) &= \text{dist}[u] + l(u, w) \text{dist}^R[w] = \\
 &= \text{dist}[u] + \text{dist}^R[w]
 \end{aligned}$$

با توجه به نکته ای در اثبات قبلی گفتیم راسی که از آن کوتاه ترین مسیر می گذرد نمی تواند خارج از دایره دایجسترا G یا G^R باشد. هم چنین با توجه به آنچه گفته شد لزومی ندارد که کوتاه ترین فاصله از راسی که آخرین بار پردازش شده است بگذرد و ممکن است راس هایی با مسیر کوتاه تر هم وجود داشته باشند. بنابراین این دو قضیه گفته شده دایجسترا دو طرفه اثبات می شود.

در بدترین حالت زمان اجرای دایجسترا دوطرفه برابر با دایجسترا عادی خواهد بود. در عمل افزایش سرعت در دایجسترا دوطرفه به گراف بستگی دارد

مصرف حافظه به دلیل نگهداری G و G^R دو برابر خواهد شد.



شکل ۹.۶: Proof(۲)

Function *Process*($u, G, dist, prev, proc$):

```

for  $(u, v) \in E(G)$  do
  | Relax( $u, v, dist, prev$ )
end
proc.Append( $u$ )

```

Input: G, s, t

Output: shortest path s to t

Function *Bidirectional Dijkstra*(G, s, t):

```

 $G^R \leftarrow ReverseGraph(G)$ 
Fill  $dist, dist^R$  with inf for each node
 $dist[s] \leftarrow 0, dist^R[t] \leftarrow 0$ 
Fill  $prev, prev^R$  with none for each node
 $proc \leftarrow empty, proc^R \leftarrow empty$ 
while true do
  |  $v \leftarrow ExtractMin(dist)$ 
  | Process( $v, G, dist, prev, proc$ )
  | if  $v$  in  $proc^R$  then
  |   | return ShortestPath( $s, dist, prev, proc, \dots$ )
  | end
  |  $v^R \leftarrow ExtractMin(dist^R)$ 
  | repeat symmetrically for  $v^R$  as for  $v$ 
end

```

حالا که دایجسترا دوطرفه را انجام دادیم باید کوتاه ترین مسیر بین s و t را با استفاده از اطلاعات

بدست آمده از متدهای قبلی و قضیه ذکرشده بدست آوریم. شبه کد این عملیات به این صورت است:

Input: $s, dist, prev, proc, t, dist^R, prev^R, proc^R$

Function *ShortestPath*($s, dist, prev, proc, t, dist^R, prev^R, proc^R$):

```

distance  $\leftarrow inf, u_{best} \leftarrow None$ 
for  $u$  in  $proc + proc^R$  do
    if  $dist[u] + dist^R[u] < distance$  then
         $u_{best} \leftarrow u$ 
        distance  $\leftarrow dist[u] + dist^R[u]$ 
    end
end
path  $\leftarrow$  empty
last  $\leftarrow u_{best}$ 
while  $last \neq s$  do
    path.Append(last)
    last  $\leftarrow prev[last]$ 
end
path  $\leftarrow$  Reverse(path)
last  $\leftarrow u_{best}$ 
while  $last \neq t$  do
    last  $\leftarrow prev^R[last]$ 
    path.Append(last)
    return (distance, path)
end

```

جلسه ۸

درخت های پوشای کمینه

سهراب نمازی - ۱۳۹۸/۱۲/۱۱

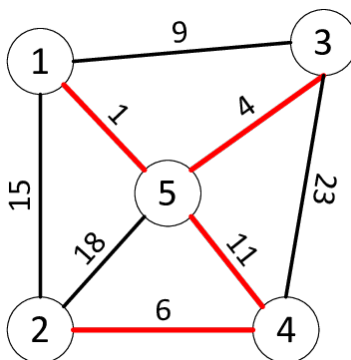
۱.۸ تعریف درخت پوشای کمینه

تعدادی راس از یک گراف کامل را در نظر بگیرید. می‌خواهیم تعدادی از یال های این گراف را به گونه ای انتخاب کنیم که تمام راس های گراف، دو به دو به همدیگر قابل دسترسی باشند. بدیهی است که برای این کار حداقل باید به تعداد یکی کمتر از تعداد راس های گراف یال انتخاب کنیم. اگر صرفاً همین تعداد یال را انتخاب کنیم گراف حاصل درختی خواهد شد که تمام رئوس آن دو به دو قابل دسترسی به همدیگر هستند. به این درخت، درخت پوشا می‌گوییم.

حال بار دیگر همین مسئله را در نظر بگیرید، با این تفاوت که یال های گراف اولیه وزن داشته باشند، بنابراین درخت کمینه حاصل هم مجموعه ای از یال های وزن دار خواهد بود. اما با توجه به این که ما کدام یال ها از گراف اولیه را برای درخت پوشای خود انتخاب کرده ایم، مجموع وزن یال های درخت پوشا میتواند متفاوت باشد. اگر یال ها را به گونه ای انتخاب کنیم که مجموع وزن یال های درخت حاصل کمترین مقدار ممکن شود، به درخت پوشای حاصل، درخت پوشای کمینه می‌گوییم.

بنابراین شروط لازم برای آنکه یک درخت برای یک گراف اولیه، درخت پوشای کمینه باشد به شرح زیر است:

- گراف اولیه یک گراف همبند، بدون جهت و وزن دار با وزن های مثبت باشد
- طبیعتاً از آنجا که حاصل درخت است، باید تعداد یال ها دقیقاً یکی کمتر از تعداد رئوس باشد
- درخت حاصل حداقل مجموع وزن یال های ممکن را داشته باشد



شکل ۱.۸: درخت پوشای کمینه گراف بالا، با یال های قرمز نشان داده شده است

۲.۸ کاربرد درخت های پوشای کمینه

درخت های پوشای کمینه کاربرد های بسیار زیادی در زمینه های مختلف دارند. به عنوان مثال در یک مثال ساده اگر بخواهیم با تعدادی کامپیوتر یک شبکه تشکیل دهیم، به گونه ای که کامپیوتر ها را با سیم به هم مرتبط سازیم، لازم است که از هر کامپیوتر به هر کامپیوتر دیگر به صورت دو به دو مسیر وجود داشته باشد. طبیعتاً اتصال سیم بین دو کامپیوتر میتواند هزینه متفاوتی با انجام همین کار بین دو کامپیوتر دیگر داشته باشد. در این مثال اگر کامپیوترها را رئوس گراف در نظر بگیریم و هزینه اتصال هر دو کامپیوتر را مانند یک یال وزن دار بین آن دو کامپیوتر در نظر بگیریم، حاصل گراف اولیه ما خواهد بود. حال با پیدا کردن درخت پوشای کمینه این گراف، ما میتوانیم به سادگی و با کمترین هزینه ممکن شبکه مورد نظر خود را تشکیل دهیم.

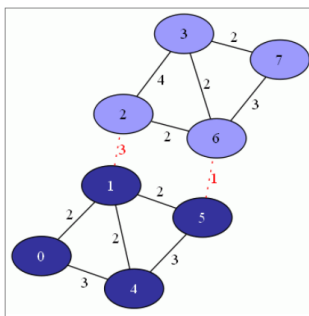
اما مسئله مهم چگونگی بدست آوردن درخت پوشای کمینه ی یک گراف اولیه است که در ادامه به این موضوع خواهیم پرداخت.

۳.۸ بدست آوردن درخت پوشای کمینه

برای بدست آوردن درخت پوشای کمینه یک گراف بدون جهت وزن دار، دو الگوریتم حریصانه مورد استفاده قرار میگیرد که در ادامه هر دو به تفصیل توضیح داده خواهند شد. اما ابتدا میخواهیم به یک خاصیت مهم که ما را در رسیدن به الگوریتم لازم برای حل این سوال یاری میکند، پی ببریم. این ویژگی، به خاصیت قطع (کات پراپرتی) معروف است.

کات پراپرتی

فرض کنید تا میانه الگوریتم لازم برای بدست آوردن درخت پوشای کمینه رفته ایم. به این معنای که تعدادی از یال هایی که قطعاً در جواب نهایی ظاهر خواهند شد را بدست آورده ایم. مطابق شکل پایین این یال ها با رنگ آبی و بنفش مشخص شده اند. حال فرض کنید این یال ها را به دو گروه مطابق شکل طوری تقسیم بندی کرده ایم که هیچ یالی که گذرا از گروهی به گروه دیگر است، تا اکنون جزو جواب نباشد. حال از میان یال های گذرا از یک گروه به گروه دیگر، خاصیت قطع ادعا میکند که قطعاً باید یال با کمترین وزن ممکن را انتخاب کرد.



شکل ۳.۸: خاصیت قطع که در شکل بالا نشان داده شده است

اثبات خاصیت قطع

برای اثبات خاصیت قطع از اثبات به روش برهان خلف کمک میگیریم. مطابق شکل بالا فرض میکنیم انتخاب یال با وزن کمینه یعنی وزن یک، انتخاب اشتباهی است. پس یال دیگر که وزن سه را دارد، انتخاب میکنیم. درخت حاصل نسبت به درخت قبل مجموعاً دو واحد وزن بیشتری دارد. پس نتیجه میگیریم انتخاب ما اشتباه بوده و درخت حاصل کمینه نیست. پس فرض ما که انتخاب نکردن یال با کمترین وزن بوده، غلط است، پس کات پراپرتی برقرار است.

۱.۳.۸ الگوریتم کروسکال

برای بدست آوردن درخت پوشای کمینه به روش الگوریتم کروسکال مراحل زیر را انجام میدهیم

- تمام یال های گراف را برحسب وزن آن ها به صورت صعودی مرتب میکنیم.
- هر بار کم وزن ترین یال را انتخاب و آن را به درخت خود اضافه میکنیم. توجه کنید که این کار در صورتی انجام میشود که اضافه کردن یال مورد نظر ایجاد دور نکند. زیرا در درخت دور وجود ندارد.
- مرحله دو را آنقدر ادامه میدهیم تا به تعداد یکی کمتر از تعداد رئوس یال اضافه کرده باشیم، درخت حاصل درخت پوشای کمینه است.

چک کردن دور در گراف

برای چک کردن دور، از ساختار داده ای مجموعه های جدا یا همان دیسجیونت ست ها استفاده میکنیم. به این صورت که اگر آیدی مربوط به دو راس یکی است، دیگر نمیتوانیم بین آن دو یال اضافه کنیم. اما اگر یکی نیست، پس از اضافه کردن یال ، مجموعه های آن دو را با هم ادغام میکنیم. جهت یادآوری مباحث مربوط به دیسجیونت ست ها میتوانید به جلسات هجدهم و نوزدهم از همین کورسی که لینک آن در انتهای این درس آمده

است مراجعه کنید [Disjoint]

شبه کد

شبه کد الگوریتم کروسکال در زیر آمده است:

```

Data: Graph G
Result: MST of G
V <- Set of vertices of G
E <- Set of Edges of G
for all u in V:
  MakeSet(u);
X <- empty set
Sort the edges in E by weight
i <- 0;
while i != |V| - 1 do
  (u, v) <- the least wheight edge
  if find(u) != find(v) then
    Union(u, v);
    Add (u, v) to X;
    i++;
  else
    continue;
  end
end

```

Algorithm 9: Kruskal Algorithm

پیچیدگی زمانی

پیچیدگی زمانی این لگوریتم را میتوان به دو بخش تقسیم کرد:

مرتب کردن یال ها

$$O(|E|\log|E|) = O(|E|\log|V^2|) = O(2|E|\log|V|) = O(|E|\log|V|)$$

و همچنین بررسی کردن یال ها:

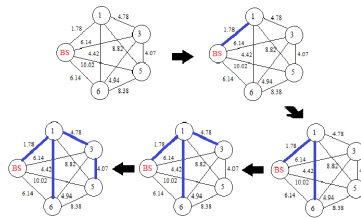
$$2|E| \cdot T(\text{Find}) + |V| \cdot T(\text{Union}) = O((|E|+|V|)\log|V|) = O(|E|\log|V|)$$

بنابراین پیچیدگی زمانی کلی الگوریتم کروسکال برابر است با :

$$O(|E|\log|V|)$$

۲.۳.۸ الگوریتم پریم

در این الگوریتم ما از ابتدا درخت کمینه خود را شروع به گسترش دادن میکنیم. در حالی که در الگوریتم کروسکال این گونه نبود و ممکن بود در میانه الگوریتم، ما یک جنگل به جای درخت داشته باشیم. در این الگوریتم مشابه الگوریتم دایکسترا عمل میشود و هر بار از هر راس، کم وزن ترین یال ممکن را انتخاب میکنیم به شرطی که در سمت دیگر یال راسی باشد که هنوز بررسی نشده باشد.



شکل ۳.۸: در الگوریتم پریم، حاصل در هر مرحله یک درخت میماند.

برای بررسی دقیقتر جزئیات و تفاوت های میان دو الگوریتم پریم و کروسکال میتوانید به لینکی که در انتهای این جزوه آورده شده است مراجعه کنید. [PrimVSKruskal]

شبه کد

شبه کد مربوط به الگوریتم پریم در زیر آمده است:

```

Data: Graph G
Result: MST of G
V <- Set of vertices of G
for all u in V:
  Cost[u] <- inf;
  Parent[u] <- nil;
Pick any initial vertex u*
Cost[u*] <- 0;
PrioQ <- MakeQueue(V); (Priority is Cost)
while PrioQ is not Empty do
  v <- ExtractMin(PrioQ)
  while There is a new adjacent for v that is called "z" do
    if z is in PrioQ and cost(z) > w(v, z) then
      cost[z] = w(v, z);
      parent[z] = v;
      changePriority(PrioQ, z, cost[z]);
    else
      continue;
    end
  end
end

```

Algorithm 10: Prim Algorithm

پیچیدگی زمانی

پیچیدگی زمانی الگوریتم پریم بستگی به نحوه پیاده سازی آن دارد، اما در حالت کلی به این گونه است:

$$|V| \cdot T(\text{ExtractMin}) + |E| \cdot T(\text{ChangePriority})$$

که این مقدار با توجه به نحوه پیاده سازی الگوریتم میتواند متفاوت باشد.

در صورت پیاده سازی با آرایه:

$$O(V^2)$$

در صورت پیاده سازی با باینری هیپ:

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

۴.۸ خلاصه

در این جلسه تعریف کاملی از درخت پوشای کمینه برای یک گراف بدون جهت وزن دار ارائه شد، سپس خاصیت قطع توضیح داده شد و نهایتاً دو الگوریتم حریصانه پریم و کروسکال برای بدست آوردن درخت پوشای کمینه ارائه شد.

جلسه ۹

الگوریتم جست وجو A^*

محمدعلی فراحت - ۱۳۹۸/۱۲/۱۳

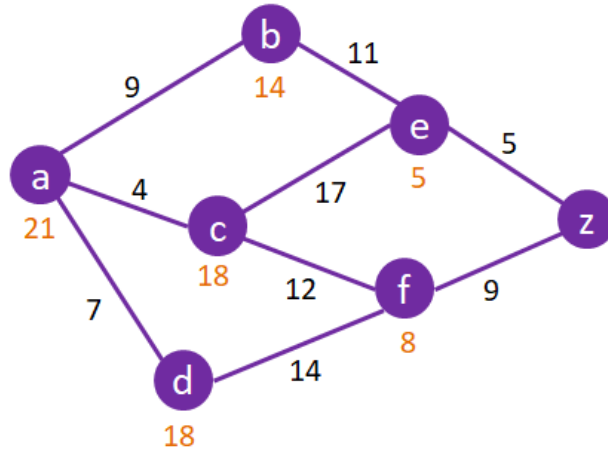
جزوه جلسه ۱۹ مورخ ۱۳۹۸/۱۲/۱۳ درس طراحی و تحلیل الگوریتم تهیه شده توسط محمدعلی فراحت. در جهت مستند کردن مطالب درس طراحی و تحلیل الگوریتم.

۱.۹ ایده کلی

- تعریف تابع پتانسیل potential-function
- از همان الگوریتم Dijkstra استفاده می‌کنیم
- Bidirectional- A^*

Potential-function ۲.۹

برای اجرای این الگوریتم ما نیاز داریم تا یک تابع تعریف کنیم این تابع در واقع یک حدس و مقدار تقریبی فاصله هر راس از راس مقصد است. برای مثال می‌توان فاصله چند شهر را در نظر گرفت. ما می‌خواهیم در کوتاه‌ترین فاصله را بین دو شهر پیدا کنیم. در اینجا تابع پتانسیل همان فاصله خطی بین دو شهر می‌باشد.



شکل ۱.۹: چند شهر همراه با فاصله آن‌ها و تابع پتانسیل هر شهر

۳.۹ محاسبه کوتاه‌ترین مسیر

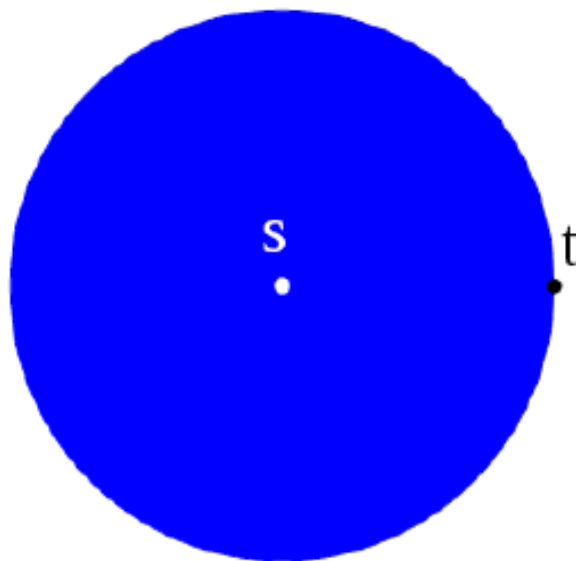
ایده کلی این الگوریتم مانند الگوریتم Dijkstra است. با این تفاوت که وزن هر یال را با فرمول زیر محاسبه می‌کنیم:

$$l_{\pi}(u, v) = l(u, v) - \pi(u) + \pi(v)$$

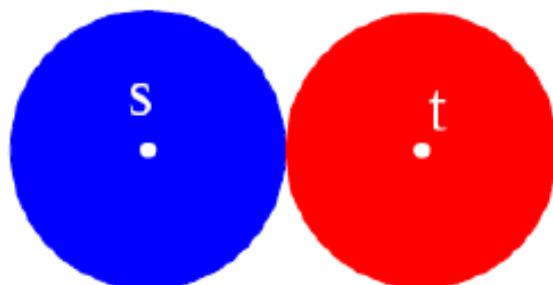
در این فرمول l همان وزن اولیه یال است و π هم همان تابع پتانسیل است.

۴.۹ Bidirectional-A*

تفاوت این روش با روش قبلی این است که ما هم از طرف راس مبدا و هم از طرف راس مقصد الگوریتم سرچ را شروع می‌کنیم و محلی که به هم می‌رسند را پیدا کرده و کوتاه‌ترین مسیر پیدا می‌شود. در شکل‌های زیر می‌توان تفاوت محسوس این دو روش را به راحتی متوجه شد:



شکل ۲.۹: روش معمولی



شکل ۳.۹: روش bidirectional

می بینیم که مساحت روش دوم بسیار کمتر از روش اول است .

جلسه ۱۱

SuffixTree

احمد بهمنی - ۱۳۹۹/۱/۳

۱.۱۱ مقدمه

پیدا کردن الگوهای خاص در ژنوم انسان و جانداران از مهمترین و کاربردی‌ترین مسائل در بحث ژنتیک* و بیوانفورماتیک† است. داشتن ژنوم موجودات مختلف می‌تواند کاربردهای زیادی داشته باشد. از جمله:

- داروسازی
- کشاورزی
- بیوتکنولوژی

طول ژنوم انسان در حدود 3×10^9 می‌باشد و بین ژنوم انسان‌های مختلف، تفاوت‌های اندکی وجود دارد که موجب تفاوت‌های فردی مانند قد، بیماری‌های ژنتیکی و ... می‌شود. پیدا کردن این تفاوت‌ها با داشتن ژنوم می‌تواند کمک بسیاری در درمان بیماری‌ها کند.

genetics*
bioinformatics†

۲.۱۱ پیدا کردن الگو در رشته

۱.۲.۱۱ راه حل ساده †

ساده ترین راه این است که الگو را روی هر کدام از کاراکترهای متن بررسی کنیم ۴.

```

۱ class PatternInText {
۲
۳     void search(String txt, String pat)
۴     {
۵         int M = pat.Length;
۶         int N = txt.Length;
۷
۸         for (int i = 0; i <= N - M; i++) {
۹             int j;
۱0
۱1             for (j = 0; j < M; j++)
۱2                 if (txt[i + j] != pat[j])
۱3                     break;
۱4
۱5             if (j == M)
۱6                 Console.WriteLine(" index at found "Pattern + i);
۱7         }
۱8     }
۱9 }

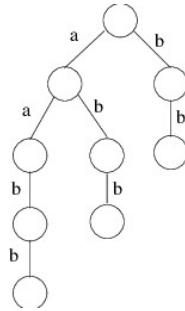
```

نمونه کد ۴: راه ساده در سی شارپ

این روش از $O(|Text| * |pattern|)$ می باشد. که برای رشته های طولانی مانند ژنوم انسان مناسب نیست!

۲.۲.۱۱ ساخت Trie از الگوها

Trie ساختار داده‌ای است که از هر رشته درختی می‌سازد که عمل پیدا کردن زیررشته را سریعتر می‌کند. در این درخت هر Node یک کاراکتر از رشته می‌باشد. ۱.۱۱



شکل ۱.۱۱: Trie for aabb, abb, bb

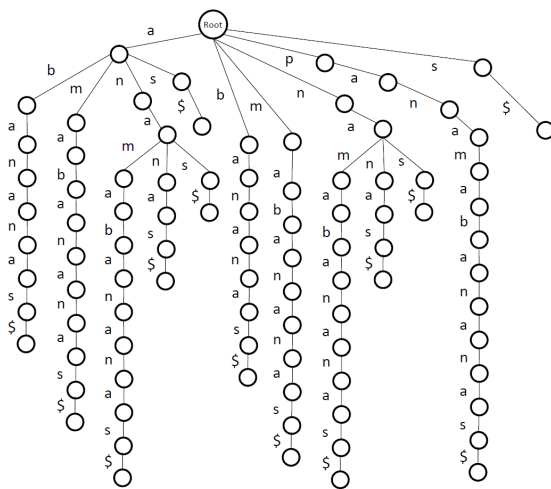
با بررسی Trie ساخته شده از الگوها، به ازای همهی کاراکترهای متن، می‌توان الگوها و مکانشان در متن را پیدا کرد. این روش از $O(|Text| * |LongestPattern|)$ می‌باشد (ساخت Trie از $O(|Pattern|)$ است). همچنین از نظر حافظه این روش از $O(|Patterns|)$ می‌باشد. §

§(در ژنوم انسان طول الگوها حدود 10^{12} است)

۳.۲.۱۱ ساخت Suffix Tree از متن

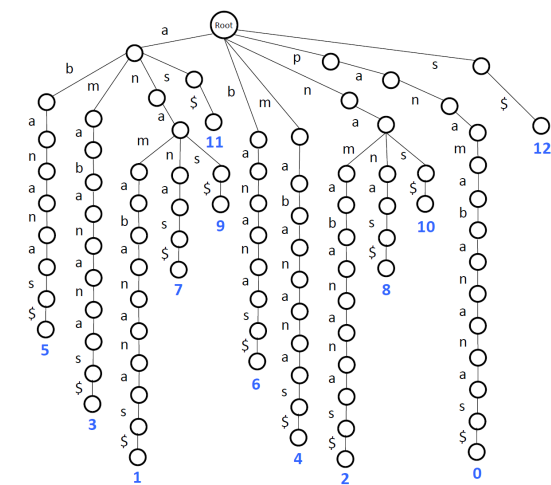
اگر به جای ساختن Trie از الگوها، همه‌ی Suffix های متن را پیدا کنیم و از آن ها Trie بسازیم، به

Suffix Tree متن رسیده‌ایم. ۲.۱۱



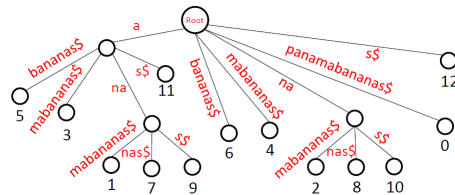
شکل ۲.۱۱: Suffix Tree for "panamabanana"

با جایگزین کردن \$ با مکان شروع هر شاخه از درخت در متن، می‌توانیم مکان الگو را پیدا کنیم. ۳.۱۱



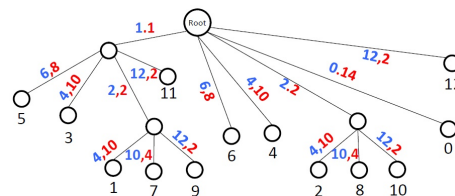
شکل ۳.۱۱: Suffix Tree with indexes for "panamabanana"

Suffix Tree از نظر حافظه به اندازهی $O(Text^2)$ جا می‌گیرد که برای ژنوم زیاد است. می‌توان به جای قرار دادن هر کاراکتر روی edge ها، راس هایی که فقط یک فرزند دارند را با راس بعدی روی یک edge ذخیره کرد. ۴.۱۱



شکل ۴.۱۱: Compressed Suffix Tree with indexes for "panamabanana"

اما با اینکار باز هم باید به اندازهی $Text^2$ زیر مجموعه‌ی هر رشته را ذخیره کنیم. برای حل این مشکل می‌توان به جای ذخیره کردن زیررشته متن، اندیس شروع و طول زیررشته را روی هر edge ذخیره کنیم که از نظر حافظه از $O(|Text|)$ می‌باشد. ۵.۱۱



شکل ۵.۱۱: Compressed Suffix Tree with indexes for "panamabanana"

جلسه ۱۲

BWT

متین مرجانی - ۱۳۹۹/۱/۱۷

۱.۱۲ مقدمه

در جلسه ی گذشته با مفهوم Suffix Trie آشنا شدیم ولی با مشکل حافظه از $O(|Text| * |Text|)$ نیز روبرو شدیم برای همین آن را به Suffix Tree ارتقا دادیم که در آن تعداد Node های درخت و در نتیجه حافظه ای اشغال می شد را کاهش دادیم. در Suffix Tree حجم حافظه ای از $O(|Text|)$ می باشد که پیشرفت قابل توجهی نسبت به Suffix Trie است.

ولی خاصیت Big O Notation این است که ضریب ثابت را در خود مشخص نمی کند. طبق محاسبات انجام شده این ضریب در $O(|Text|)$ برای ژنوم انسان چیزی حدود $20 * |Text|$ می باشد که عددی تاثیر گذار است.

برای حل این مشکل از الگوریتم BWT (Burrows-wheeler transform) استفاده می کنیم.

BWT ۲.۱۲

همانطور که از اسمش پیداست BWT یک تبدیل برای رشته ی مورد نظر است. خاصیت این تبدیل برگشت پذیر بودن آن است.

هدف این تبدیل این است که کاراکترهای یکسان حداالامکان بهم نزدیک تر شوند. چون رشته هایی که حروف تکرار شونده در آن ها پشت سر هم هستند راحت تر فشرده میشوند و در نتیجه حافظه ی کمتری میگیرند.

Run-length encoding :

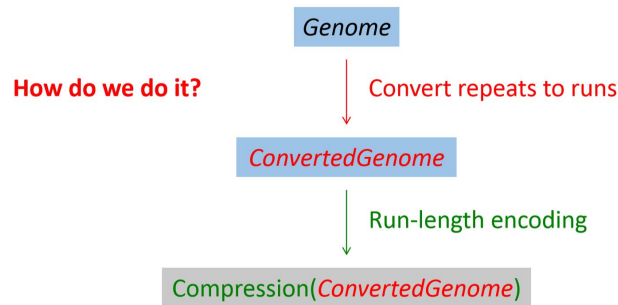
Text

GGGGGGGGGGCCCCCCCCCCAAAAAATTTTTTTTTTTTTTTTCCCCCG

=

10G11CYA15T5C1G

حروف داخل ژنوم انسان طول تکرار زیادی ندارند اما چون از تعداد حروف محدود (۴) تشکیل شده اند پس بااعمال کردن این تبدیل میتوان رشته ی آن را بسیار فشرده کرد.



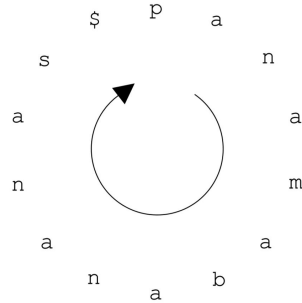
۳.۱۲ Constructing BWT

در مرحله ی اول نیاز داریم که همه ی چرخش های رشته ی مورد نظر را بدست بیاوریم. برای مثال کلمه ی panamabananas\$ را به عنوان رشته ی اصلی در نظر بگیرید. Cyclic Rotations های آن به صورت زیر ساخته میشوند:

Cyclic Rotations

```

panamabananas$
$panamabananas
s$panamabananas
as$panamabanan
nas$panamabana
anas$panamaban
nanas$panamaba
ananas$panamab
bananas$panama
abananas$panam
mabananas$pana
amabananas$pan
namabananas$pa
anamabananas$pa
anamabananas$p
    
```



سپس همه ی این رشته ها را بر اساس حروف الفبا Sort می کنیم :

panamabananas\$	→	\$panamabananas
\$panamabananas		abananas\$panam
s\$panamabananas		amabananas\$pan
as\$panamabanan		anamabananas\$p
nas\$panamabana		ananas\$panamab
anas\$panamaban		anas\$panamaban
nanas\$panamaba		as\$panamabanan
ananas\$panamab		bananas\$panama
bananas\$panama		mabananas\$pana
abananas\$panam		namabananas\$pa
mabananas\$pana		nanas\$panamaba
amabananas\$pan		nas\$panamabana
namabananas\$pa		panamabananas\$
anamabananas\$p		s\$panamabanan

حال از ماتریس Sort شده، حرف ستون آخر هر ردیف را انتخاب میکنیم. به ترتیب از بالا به پایین، این حروف را کنار هم میچینیم.

```

$panamabananass
abananas$panamm
amabananas$pann
anamabananas$p
ananas$panamabb
anas$panamabann
as$panamabanann
bananas$panamaa
mabananas$panaa
namabananas$paa
nanas$panamabaa
nas$panamabanaa
panamabananas$$
s$panamabanana
    
```

رشته ی بدست آمده همان تبدیل BWT مورد نظر میباشد:

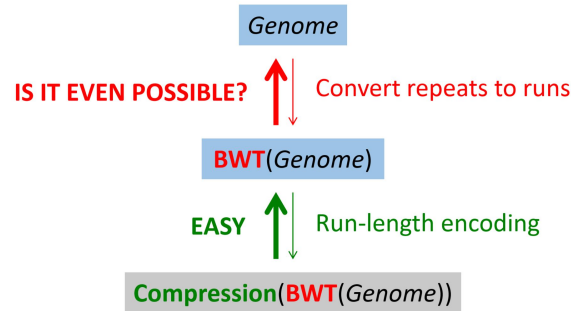
$BWT(\text{panamabananas}) = \text{smnbpbnnaaaaa}\a

همانطور که می بینید حروف مشابه در این تبدیل بهم نزدیک تر شدند:

$BWT(\text{panamabananas}) = \text{smnbpbnnaaaaa}\a

۴.۱۲ Inverting BWT

حال باید نشان دهیم چگونه می توان از یک رشته ی تبدیل شده به خود اصل آن رشته برگشت.



فرض کنیم رشته ی تبدیل شده $\text{annb}\$aa$ است و رشته ی اصلی که میخوایم به آن برگردیم $\text{banana}\$$ می باشد.

با Sort کردن حروف رشته ای که داریم، حروف ستون اول ماتریس تبدیل BWT بدست می آید. ستون آخر هم همان حروف رشته ی تبدیل می باشد.

$\$banana$		$a\$$
$a\$banan$		na
$ana\$ban$		na
$anana\$b$		ba
$banana\$$	→	$\$b$
$na\$bana$	2-mers	an
$nana\$ba$		an

چون با منطق Cyclic Rotations این رشته ها را ساخته ایم پس حرف اول هر ردیف در اصل کاراکتر بعدی از حرف آخر همان ردیف می باشد. با این فرض ما کاراکترهای رشته ی اصلی را تا الان دو به دو مرتب کردیم. این یعنی دو حرف اول ماتریس را بدست آورده ایم. با Sort کردن ماتریس این دو حرف به صورت زیر، و جایگذاری آن در ماتریس اصلی؛ به کامل تر شدن ماتریس تبدیل نزدیک تر می شویم.

\$banana		a\$		\$b
a\$banan		na		a\$
ana\$ban		na		an
anana\$b	→	ba	→	an
banana\$	2-mers	\$b	Sort	ba
na\$bana		an		na
nana\$ba		an		na

حال دوباره با منطق Cyclinc Rotations حرف آخر هر رشته در اصل، کاراکتر قبلی دو حرف بدست آمده است. پس ترتیب سه کاراکتر از رشته ی اصلی را بدست آوردیم.

\$banana		a\$b
a\$banan		na\$
ana\$ban		nan
anana\$b	→	ban
banana\$	3-mers	\$ba
na\$bana		ana
nana\$ba		ana

با تکرار همین عملیات به تعداد حروف رشته به Invert BWT یا همان اصل رشته (ژنوم) میرسیم.

\$banana		a\$bana		\$banan
a\$banan		na\$ban		a\$bana
ana\$ban		nana\$b		ana\$ba
anana\$b	→	banana	→	anana\$
banana\$	6-mers	\$banan	Sort	banana
na\$bana		ana\$ba		na\$ban
nana\$ba		anana\$		nana\$b

$$\text{Invert-BWT}(\text{anmb$aa}) = \text{banana\$}$$

جلسه ۱۳

تطبيق الگوها و تبدیل BW

شایان موسوی نیا - ۱۳۹۹/۲/۱۶

جزوه جلسه ۱۳ ام مورخ ۱۳۹۹/۲/۱۶ درس طراحی و تحلیل الگوریتم تهیه شده توسط شایان موسوی نیا.

۱.۱۳ یک مشاهده عجیب

در جلسه قبل با تبدیل BW آشنا شدیم و مزیت های آن را نسبت به درخت پسوندی بیان کردیم. با این حال سعی در بهبود این راه حل داریم. یکی از مشکلات حال حاضر این روش، استفاده زیاد از حافظه است. به طور مثال اگر ما میخواستیم ماتریس تمام حالت ها یک الگو را درست کنیم باید به اندازه طول متن * طول متن حافظه در اختیار داشته باشیم. برای بهبود این روش باید به یک خاصیت در تمامی این ماترس های پی ببریم.

\$panamabananas
 abananas\$panam
 amabananas\$pan
 anamabananas\$p
 ananas\$panamab
 anas\$panamaban
 as\$panamabanan
 bananas\$panama
 mabananas\$pana
 namabananas\$pa
 nanas\$panamaba
 nas\$panamabana
 panamabananas\$
 s\$panamabanana

شکل ۱۰.۱۳: ماتریس حالات

اگر به شکل ۱۰.۱۳ توجه کنیم، میتوان به یک مشاهده جالب رسید. به طور مثال بیایید به هر a موجود در متن یک شماره به خصوص دهیم. به اولین a در سطر دوم ماتریس عدد ۱، به اولین a در سطح سوم عدد ۲ و تا اولین a در سطر هفتم عدد ۶ را بدهیم. با این کار ما به هر a موجود در متن مان یک عدد منحصر به فرد دادیم.

حال بیایید a ها را از اول سطرهای ۲ تا ۷ حذف کنیم و آنها را به انتهای سطرهای خودشان اضافه کنیم تا همان سطرهای پایین درست شود. حال میتوان به این موضوع پی برد که ترتیب شماره گذاری a های سطرهای پایین همان ترتیب شماره گذاری a های بالا است. این موضوع برای باقی حروف نیز درست است.

- امین n وقوع حرف در اولین سطر
- و امین n وقوع حرف در آخرین سطر
- مطابق با حرف در همان نقطه در متن است

$$p_1 a_3 n_1 a_2 m_1 a_1 b_1 a_4 n_2 a_5 n_3 a_6 s_1 \$_1$$

شکل ۲۰.۱۳: متن شماره گذاری شده

s_1 panamabanana
 a_1 bananas\$panam
 a_2 mabananas\$pan
 a_3 namabananas\$p
 a_4 nanas\$panama**b**₁
 a_5 nas\$panamaba**n**₂
 a_6 s\$panamabana**n**₃
b₁ananas\$panama**a**₁
m₁abananas\$pana**a**₂
n₁amabananas\$pa**a**₃
n₂anas\$panamaba**a**₄
n₃as\$panamabana**a**₅
p₁anamabananas\$**s**₁
s₁\$panamabanana**a**₆

شکل ۳.۱۳: ماتریس حالات شماره گذاری شده

با توجه به شکل های ۲.۱۳ و ۳.۱۳ میتوان به روشی دست یافت که لازم نباشد تمامی ماتریس حالات را، رسم کنیم که در ادامه توضیح آن را میدهیم.

حال بیاید با توجه به نکات بالا و بدون تشکیل کامل ماتریس معکوس BWT را حساب کنیم. طبق شکل ۲.۱۳ میدانیم که به طور مثال حرف قبل از ۵ امین a ، ۲ امین n است.

حال از علامت دلار شروع میکنیم و به حرف قبلی میرویم و این کار را تا زمانی انجام میدهیم که دوباره به علامت دلار برسیم.

برای نشان دادن روند کار از علامت دلار استفاده میکنیم و به اولین s میرسیم، سپس بعد از اولین s به ۶ امین a میرویم و این کار را تا آخر ادامه میدهیم طوری که در مرحله آخر از اولین p به علامت دلار دوباره میرسیم. مطابق با شکل ۴.۱۳



شکل ۴.۱۳: معکوس BWT

حال توانستیم با فضای حافظه کمتری نسبت به قبل معکوس BWT را حساب بکنیم، به طوری که نیازی نیست تمام ماتریس حالت را ذخیره کنیم.

- Memory : $2 * |\text{Test}|$
- Time : $O(|\text{Text}|)$

به تطبیق الگوها برگردیم.

در درخت پسوند ها مقدار زمان اجرا برنامه و حافظه مورد نیاز ضیق زیر بود :

- Memory : $20 * |\text{Test}|$
- Time : $O(|\text{Text}| + |\text{Patterns}|)$

میدانیم که طول ژنوم های انسان ۳ ضربدر ۱۰ به توان ۹ است.

حال سوال این است که میتوان از BWT(Text) برای طراحی یک الگوریتم زمان - خطی با حافظه مفید بیشتری برای تطبیق الگوهای چندگانه استفاده کرد؟

بیااید با یک مثال بیشتر به این سوال بپردازیم.

۲.۱۳ پیدا کردن تطبیق الگو با استفاده از BWT

مثال ۱: الگوی ana را در عبارت panamabanans پیدا کنید. در ابتدا باید بدانیم در این راه حل از انتهای الگو باید شروع به حرکت کنیم و میدانیم که فقط ستون ابتدا و انتهای ماتریس حالات رو داریم.

```

$1panamabananas1
a1bananas$panam1
a2mabananas$pan1
a3namabananas$p1
a4nanas$panamab1
a5nas$panamaban2
a6s$panamabana3
b1ananas$panama1
m1abananas$pana2
n1amabananas$pa3
n2anas$panamaba4
n3as$panamabana5
p1anamabananas$1
s1$panamabanana6

```

شکل ۵.۱۳: مثال ۱

آخرین حرف عبارت ana حرف a است، پس می‌ایم تمامی a های موجود را در ستون اول پیدا می‌کنیم. حال باید ببینیم از این a ۵ ای که پیدا کردیم کدامشان حرف قبلیشان n بوده. (شکل ۵.۱۳) سپس دوباره باید چک کنیم کدام یک از n های حرف قبلیشان a است. با این روش به شکل نهایی که در شکل ۶.۱۳ می‌ریسیم.

```

$1panamabananas1
a1bananas$panam1
a2mabananas$pan1
a3namabananas$p1
a4nanas$panamab1
a5nas$panamaban2
a6s$panamabana3
b1ananas$panama1
m1abananas$pana2
n1amabananas$pa3
n2anas$panamaba4
n3as$panamabana5
p1anamabananas$1
s1$panamabanana6

```

شکل ۶.۱۳: مرحله دوم

حالا به شکل الگوریتمی بیایید سوال بالا را حل کنیم.

ابتدا با دو مفهوم اندیس بالا و اندیس پایین در این بخش باید آشنا بشیم.

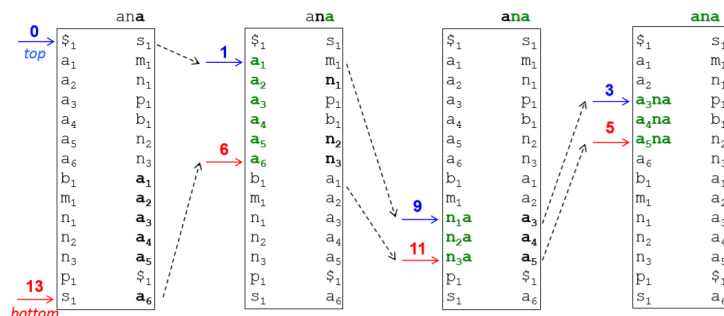
اندیس بالا: اولین مکان حرف مورد نظر بین جایگاه های بالا به پایین در آخرین ستون.
اندیس پایین: آخرین مکان حرف مورد نظر بین جایگاه های بالا به پایین در اولین ستون.

```
BWMatching(FirstColumn,LastColumn,Pattern,LastToFirst)
top <- 0
bottom <- |LastColumn| - 1
while top <= bottom do
  if Pattern is nonempty then
    symbol <- last letter in Pattern
    remove last letter from pattern
    if positions from top to bottom in LastColumn contain symbol
      then
        topIndex <- first position of symbol among positions from
          top to bottom in LastColumn
        bottomIndex <- last position of symbol among positions
          from top to bottom in LastColumn
        top <- LastToFirst(topIndex)
        bottom <- LastToFirst(bottomIndex)
      else
        | return 0
    end
  else
    | return bottom - top + 1
  end
end
```

Algorithm 11: BWMatching

با این حال BWMatching سرعت زیادی طبق الگوریتم بالا ندارد. دلیل آن هم این است که تمامی حروف را از بالا تا پایین در هر مرحله بررسی میکند. به طور مثال در شکل ۷.۱۳ میتوان به این موضوع پی برد که در مرحله اول تمامی a ها را بررسی میکند و سپس تمامی n ها و سپس دوباره تمامی a ها را بررسی میکند که همین باعث کندی برنامه میشود.

پس باید الگوریتم بالا را بهبود بخشید.



شکل ۷.۱۳: نحوه عمل کرد الگوریتم بالا

در ابتدا باید ارایه ای به اسم شمارش را تعریف کنیم که مطابق شکل ۸.۱۳ عمل میکند.

نحوه عملکرد این ارایه به فرم زیر است :

$\text{Count.symbol}(i, \text{LastColumn}) =$ رخ داد نماد در اولین i امین مکان در آخرین ستون

i	FirstColumn	LastColumn	LASTTOFIRST(i)	COUNT
				\$ a b m n p s
0	\$1	s1	13	0 0 0 0 0 0 0
1	a1	m1	8	0 0 0 0 0 0 1
2	a2	n1	9	0 0 0 1 0 0 1
3	a3	p1	12	0 0 0 1 1 0 1
4	a4	b1	7	0 0 0 1 1 1 1
5	a5	n2	10	0 0 1 1 1 1 1
6	a6	n3	11	0 0 1 1 2 1 1
7	b1	a1	1	0 0 1 1 3 1 1
8	m1	a2	2	0 1 1 1 3 1 1
9	n1	a3	3	0 2 1 1 3 1 1
10	n2	a4	4	0 3 1 1 3 1 1
11	n3	a5	5	0 4 1 1 3 1 1
12	p1	\$1	0	0 5 1 1 3 1 1
13	s1	a6	6	1 5 1 1 3 1 1
				1 6 1 1 3 1 1

شکل ۸.۱۳: ارایه شمارش

```

BetterBWMatching(FirstOccurrence,LastColumn,Pattern,Count)
  top <- 0
  bottom <- |LastColumn| - 1 while top<=bottom do
    if Pattern is nonepmty then
      symbol <- last letter in Pattern
      remove last letter from Pattern

      top <- FirstOccurrence(symbol) +
      Count.symbol(top,LastColumn)
      bottom <- FirstOccurrence(symbol) + Count.symbol(bottom
      + 1,LastColumn) - 1
    else
      | return bottom - top + 1
    end
  end
end

```

Algorithm 12: BetterBWMatching

در مثال پیدا کردن الگوهای ana ما فهمیدیم که ۳ بار این الگو تکرار شده، ولی حال سوال این است که این ۳ بار در کجای متن آماده اند.

برای اینکار از ارایه پسوند ها استفاده میکنیم که در این ارایه موقعیت شروع هر پسوند را با شروع یک ردیف نگه میدارد. این موضوع در شکل ۹.۱۳ نشان داده شده است.

13	\$ ₁ panamabananas ₁
5	a ₁ bananas\$panam ₁
3	a ₂ mabananas\$pan ₁
1	a ₃ namabananas\$pa ₁
7	a ₄ nanas\$panamab ₁
9	a ₅ nas\$panamaban ₂
11	a ₆ s\$panamabanan ₃
6	b ₁ ananas\$panama ₁
4	m ₁ abananas\$pana ₂
2	n ₁ amabananas\$pa ₃
8	n ₂ anas\$panamaba ₄
10	n ₃ as\$panamabana ₅
0	p ₁ anamabananas\$ ₁
12	s ₁ \$panamabanan _a ₆

شکل ۹.۱۳: ارایه پسوند

جلسه ۱۴

الگوریتم KMP

امید میرزاجانی - ۱۳۹۹/۲/۱۲

جزوه جلسه ۱۴م مورخ ۱۳۹۹/۲/۱۲ درس طراحی و تحلیل الگوریتم تهیه شده توسط امید میرزاجانی. در جهت مستند کردن مطالب درس طراحی و تحلیل الگوریتم این الگوریتم که یکی از معروف ترین الگوریتم ها برای String Pattern Matching است، در سال ۱۹۷۰ توسط سه نفر به نام های Knuth و Morris ، Pratt پیدا شد. این الگوریتم از این قرار است که دو رشته، یکی متن اصلی و دیگری الگو، را به عنوان ورودی میگیرد و خروجی آن تمام نمایه * های است که آن الگو در متن اصلی پیدا می شود.

Brute Force ۱.۱۴

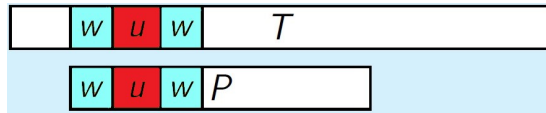
وقتی از Matching Pattern blue حرف میزنیم، اولین و ساده ترین ایده که به ذهن می رسد این است که به ترتیب از اول تا آخر متن اصلی را پیمایش کنیم و ببینیم که آیا در جایی با الگو، برابر میشود یا خیر. اما ایده ایده برای متن ها و یا الگوهای طولانی بسیار زمان گیر است و اصلاً به صرفه نیست؛ زیرا از اردر `|pattern|*|text|` است.

Index*

اما کمی که به همین الگوریتم ساده فکر کنیم، میتوانیم بعضی از نمایه ها را صرف نظر کنیم. به طور مثال در شکل زیر، پس از چک کردن نمایه اول، دیگر نیازی به چک کردن نمایه های دوم و سوم نیست؛ زیرا مشخصاً آن ها با A شروع نمیشوند و قابل صرف نظر هستند.

A	T	T	A	C	G	G	T	A	A	C	G
A	C	G	G								
	A	C	G	G							

Border : بردر یک رشته به نام S پیشوندی از آن رشته است، که در انتها نیز آمده باشد. به عبارتی هم پیشوند است و هم پسوند. البته باید توجه داشت که بردر یک رشته، نمیتواند خود آن رشته باشد. برای مثال، رشته های **AG** و **AGAG** بردر های رشته **AGAGAG** هستند. برای حل مسأله از لم زیر استفاده میکنیم؛ اگر بزرگ ترین بردر الگو X باشد، و الگو و متن اصلی در یک نمایه ای با هم مرتبط[†] شدند، دیگر امکان مرتبط شدن در هیچ یک از نمایه های قسمت قرمز رنگ وجود ندارد.



۲.۱۴ Function Prefix

یک آرایه ای تعریف میکنم که نمایه ام[†] آن مقدار طولانی ترین بردر آن رشته را برمیگرداند.

A	T	A	T	A	C	A	T	C	A	T	A
0	0	1	2	3	0	1	2	0	1	2	3

Prefix Function

همچنین قضیه ای که در رابطه با Prefix Function مطرح است، این است که این آرایه صعودی است و در هر مرحله، حداکثر یک واحد افزایش میابد زیرا با اضافه شدن یک کاراکتر به انتها، نهایتاً همان کاراکتر نیز مرتبط شود و مقدار Prefix Function یک واحد زیاد شود.

[†]Match

این سودو کد نیز برای محاسبه Prefix Function به کار میرود. $O(|P|)$

```

۱ ComputePrefixFunction(P)
۲ {
۳     s = array of integers of length |P|
۴     s[0] = 0, border = 0
۵     for i from 1 to |P| - 1:
۶         while (border > 0) and (P[i] != P[border]):
۷             border = s[border - 1]
۸         if P[i] == P[border]:
۹             border = border + 1
۱0        else:
۱1            border = 0
۱2        s[i] = border
۱3    return s
۱4 }

```

نمونه کد ۵: محاسبه Prefix Function

۳.۱۴ الگوریتم نهایی

برای محاسبه Pattern Matching ایده این است که رشته الگو و متن اصلی را به هم بچسبانیم و Function Prefix را بر روی آن حساب کنیم.

S	A	T	C	A	\$	A	T	C	A	T	C	C	A	T	C	A
	0	0	0	1	0	1	2	3	4	2	3	0	1	2	3	4

همانطور که به نظر میرسد، همه نمایه هایی که مقدار Prefix Function در آن به اندازه طول الگو است، به عنوان جواب در خروجی باید نمایش داده شود.

```

۱ FindAllOccurrences(P, T)
۲     S = P + '$' + T
۳     s = ComputePrefixFunction(S)
۴     result = empty list
۵     for i from |P| + 1 to |S| - 1:
۶         if s[i] == |P|:
۷             result.Append(i - |P|)
۸     return result

```

نمونه کد ۶: محاسبه Prefix Function

پس سرانجام ما به کمک این الگوریتم، اردر را از $|T| * |P|$ به $|T| + |P|$ کاهش دادیم.

جلسه ۱۵

الگوریتم kmp و suffix array efficient

صدرا خاموشی - ۱۳۹۸/۲/۶

جزوه جلسه ۱۵ ام مورخ ۱۳۹۸/۲/۶ درس طراحی و تحلیل الگوریتم تهیه شده توسط صدرا خاموشی . در جهت مستند کردن مطالب درس طراحی و تحلیل الگوریتم ، بر آن شدیم که از دانشجویان جهت مکتوب کردن مطالب کمک بگیریم. هر دانشجو می‌تواند برای مکتوب کردن یک جلسه داوطلب شده و با توجه به کیفیت جزوه از لحاظ کامل بودن مطالب، کیفیت نوشتار و استفاده از اشکال و منابع کمک آموزشی، حداکثر یک نمره مثبت از بیست نمره دریافت کند. خواهش‌مند است نام و نام خانوادگی خود، عنوان درس، شماره و تاریخ جلسه در ابتدای این فایل را با دقت پر کنید. مطالبی که در ادامه آمده فقط جنبه راهنمایی شیوه استفاده از لاتک می‌باشد. خواهشمند است این پاراگراف و مطالب بعدی را از نسخه جزوه‌ای که تحویل می‌دهید، حذف کنید.

۱.۱۵ معیار ارزیابی جزوه

معیارهای مورد استفاده برای ارزیابی کیفیت جزوه به شرح زیر است:

- پوشش کامل مطالب

- رعایت قواعد نگارشی دستور زبان فارسی
- استفاده از اشکال مناسب
- اشاره به منابع کمک آموزشی

مقدمه :

در ابتدا جلسه راجه الگوریتم *kmp* صحبت میشود . و همچنین نحوه ی محاسبه ی *prefix function* میپردازیم و سپس سراغ *suffix array* رفته و نحوه ی محاسبه ی آن را با پیچیدگی زمانی $O(n+m)$ میبینیم. برای الگوریتم *kmp* ابتدا باید *prefix array* را محاسبه کنیم

دلیل محاسبه *prefix array* این است که ، ما برای تطبیق دادن *pattern* با متن لزومی نداره که تکی تکی *character* ها رو چک کنیم میتونیم بعضی از *character* ها رو *skip* کنیم.

در ابتدا باید با مفهوم *border* در یک *string* آشنا شویم.

Border : *border* یک *string* قسمتی از *string* هست که *prefix* و *suffix* با هم برابر باشند. برای اطلاعات بیشتر درباره *prefix* و *suffix* به این سایت مراجعه کنید. [[prefixVsSuffix](#)].

تعریف *prefix function*: یک آرایه ای هست که برای خانه ی *i* ام از آرایه برابر است با طول بزرگترین *border* تا خانه ی *i* ام. *

الگوریتم *kmp* : پس از محاسبه کردن *prefix function* حال میبینیم که چگونه از *prefix function* استفاده کنیم. یک *string* جدید میسازیم به فرم : `pattern + "$" + text` ، و برای آن *function* *prefix* را محاسبه میکنیم. سپس برای خانه ی *i* ام از آرایه ، اگر `prefixFunction[i]` برابر با طول *pattern* شد، یعنی که *match* ، *pattern* شده است.

برای اطلاعات بیشتر درباره ی *kmp* به این لینک مراجعه شود. [[KMP](#)]

*برای توضیحات بیشتر و کامل تر به صفحه ۸۶ ، [string۳.pdf](#) مراجعه کنید

برای مشاهده شبه کد *kmp*[†] می‌توانید از مثال زیر در الگوریتم ۲۹ استفاده کنید :

```

Data: text , pattern
Result: all Occurrences
s=pattern + "$"+ text;
result = empty list;
prefix = computePrefixFunc(s);
initialization;
index = |pattern| + 1;
while index < |s| do
    if prefix[index]== |pattern| then
        result.Append(index -2|P|);
    end
end

```

Algorithm 13: Knuth-Morris-Pratt Algorithm

کوئیز حل شده واسه ی prefix function :

```

text:ABABCABABC
pattern:ABC
solve:
ABC$ABABCABABC
prefixFunc : 00001212312123

```

پیدا کردن suffix tree از روی suffix array :

پیچیدگی زمانی برای پیدا کردن suffix tree در حالت معمولی برابر $O(\text{text}*\text{text})$ بود.

حال می‌خواهیم الگوریتمی با پیچیدگی زمانی $n \log(n)$ ارائه دهیم. برای پیدا کردن suffix tree باید دو مرحله انجام بدهیم. ابتدا پیدا کردن suffix array و سپس تبدیل suffix array به suffix tree .

: Suffix array

Suffix array برای یک string اینگونه بدست می آید. ابتدا همه ی suffix ها رو پیدا کرده و آنها را بر اساس حروف اول شان مرتب کرده. و در آرایه، index، شروع suffix متناظر را میزاریم. در حالت عادی این کار $O(n*n)$ طول میکشد.

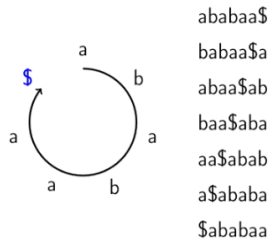
اگر به آخر string، \$ اضافه کنیم و جایگشت دوری اش را بنویسیم. به cyclic shift میرسیم. حال اگر این cyclic shift های بدست آمده را sort کنیم براساس حروف اولشان، به sorting cyclic shift میرسیم. سپس اگر برای هر جایگشت، character های بعد از \$ را پاک کنیم به همان suffix array میرسیم.

```

aba$
sorting cyclic shift:
$aba
a$ab
aba$
ba$a

```

مثال:



شکل ۱۰.۱۵ : cyclic shift length of ۶

‡ برای توضیحات تکمیل تر به صفحه ی ۴۰ اسلاید ۴.۱ string مراجعه شود

cyclic shift	sorting cyclic shift	suffix array
ababaa\$	\$ababaa	\$
babaa\$a	a\$ababa	a\$
abaa\$ab	aa\$abab	aa\$
baa\$aba	abaa\$ab	abaa\$
aa\$abab	ababaa\$	ababaa\$
a\$ababa	baa\$aba	baa\$
\$ababaa	babaa\$a	babaa\$

شکل ۲.۱۵ : Suffix array

حال برای اینکه این sorting cyclic shift را بدست بیاریم ، باید از روش زیر استفاده کنیم.

ابتدا character ها text داده شده را sort میکنیم.

حال cyclic shift به طول $L=1$ sort شده.

تازمانی که $L < \text{length}(\text{text})$ ، cyclic shift های به طول $2L$ را با استفاده از قبلی ها sort میکنیم.

۱ - ۲ - ۴ - ۸ . . .

از آنجا که مثلا cyclic shift طول ۲ از ۲ تا cyclic shift با طول ۱ تشکیل شده پس میتوان آن ها را

با cyclic shift به طول ۱ ، sort کرد و الی آخر....

وقتی $L > \text{length}(\text{text})$ شد cyclic shift سورت شده ی بدست آمده همان sorting cyclic shift

هست.

Count Sort : برای sort کردن single character ها از count sort استفاده میکنیم.

abbcaaabcc

count Sort :

aaaabbbccc

مثال :

در واقع تعداد alphabet ها را در آورده و با استفاده از آن ها single character ها را sort میکنیم.

پیچیدگی زمانی *count sort* ، $O(\text{length}(\text{text}) + \text{alphabets})$ ، میشود. هر بار که طول *cyclic shift* را ۲ برابر میکنیم. برای *sort* کردن آن ها با پیچیدگی زمانی $O(n)$ انجام میدهیم و این کار را هم $\log(n)$ بار را انجام میدهیم. پس ساختن *suffix array* ، $O(n \log n)$ طول میکشد.

نکته: میدانیم که *alphabet* ما *sort* شده است. یعنی مثلا میدونیم که a, b, c, d, \dots به این ترتیب هستند و زمانی برای *sort* کردن آنها در نظر نمیگیریم. §

برای مطالعه در مورد *sort stable* به لینک زیر مراجعه شود: [*stableSort*]

: *sort single charecter*

§ (برای توضیحات بیشتر به مثال صفحه ی ۵۴ اسلاید ۴۰۱ *string* مراجعه شود)

□ its our alphabets

```

Data: S
Result: sorted charecters
order = array of size |S|;
count = array of size |Σ| index=0;
while index < |S|-1 do
    | count[S[index]] = count[S[index]] + 1;
    | index = index + 1;
end
index = 1;
while index < |Σ|-1 do
    | count[index] = count[index] + count[index-1];
    | index = index + 1;
end
index = |S|-1;
while index >= 0 do
    | c = S[index];
    | count[c] = count[c] - 1;
    | order[count[c]] = index;
end
return order;

```

Algorithm 14: sorting Charecters

: Equivalnce Class

C_i : این نماد یعنی cyclic shift به طول L که از i index، ام شروع میشود. اگر $C_i = C_j$ در نتیجه این دو تا cyclic shift از یک نوع کلاس هستند. حال برای محاسبه equivalence class به آرایه به طول L تکست در نظر گرفته. و تعداد نوع shift cyclic ها به طول L را محاسبه کرده و داخل آرایه قرار میدهم. و برای i و j اگر $C_i = C_j$ آنگاه $class[i] = class[j]$

Example		
$S = ababaa\$$		
6	\$	$order = [6, 0, 2, 4, 5, 1, 3]$
0	a	$class = [1, 2, 1, 2, 1, 1, 0]$
2	a	
4	a	
5	a	
1	b	
3	b	

شکل ۳.۱۵ : class

برای توضیحات تکمیلی به صفحه ی ۷۶ اسلاید ۴.۱ string مراجعه شود

۹

جلسه ۱۶

آرایه پسوندی بهینه و آرایه LCP

زهرا حسینی - ۱۳۹۹/۱/۲۶

۱.۱۶ دوره مفاهیم آرایه و درخت پسوندی

آرایهٔ پسوندی آرایه‌ای از همهٔ پسوندهای یک رشته است. این آرایه بر اساس نویسه‌های * هر یک از پسوند ها مرتب شده است درخت پسوندی ساختمان داده‌ای است که هر یک از یال‌های این درخت با یک پسوند برجسب شده است. به عنوان مثال آرایه پسوندی رشته $S = \text{"ababaa\$"}$ به صورت زیر است:

```
$
a$
aa$
abaa$
ababaa$
baa$
babaa$
Suffix array: order=[6,5,4,2,0,3,1]
```

character*

مجموع طول همه ی پسوندهای یک رشته به طول S برابر است با:

$$1+2+\dots+|S|=\theta(|S|^2)$$

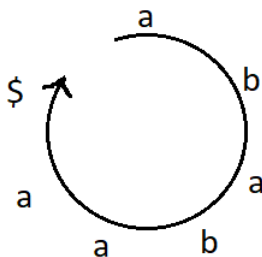
مرتب کردن همه ی این موارد حافظه زیادی را مصرف میکند. ذخیره کردن آنگاه نیز از لحاظ پیچیدگی زمانی برابر است با: $O(|S|)$

حال به بررسی نحوه ساخت آرایه پسوندی میپردازیم

۲.۱۶ ساخت آرایه پسوندی

ایده کلی به این صورت است که تغییر مکان چرخه ایی با طول یک شروع میکنیم و پسوند های حاصل را مرتب کنیم، حال طول چرخش را دو برابر میکنیم به این معنی که طول پسوندهای حاصل در این مرحله دو برابر مرحله قبل است. این عمل را تا جایی ادامه میدهیم که طول چرخش بزرگتر از طول رشته مورد بررسی باشد. سپس نویسه های بعد از علامت $\$$ را حذف میکنیم و مقادیر حاصل در واقع همان آرایه پسوندی است.

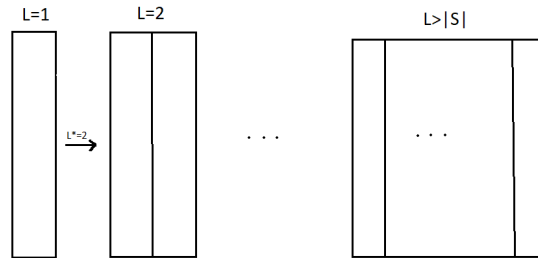
منظور از تغییر مکان چرخه ایی شکل زیر است که برای رشته ی $S = "ababaa\$"$ رسم شده است:



شکل ۱.۱۶: جا به جایی جزئی چرخه ایی[†]

برای درک بهتر دو برابر کردن طول رشته ها به شکل زیر دقت کنید، در هر مرحله اطلاعات جزئی مراحل قبل به مرتب کردن در آن مرحله کمک میکند:

[†]partial cyclic shift



شکل ۲.۱۶: تغییر طول چرخش

ساخت آرایه پسوندی چند مرحله دارد که به صورت مجزا به شرح هریک میپردازیم:

۱.۲.۱۶ مرتب کردن نویسه های منفرد

‡ در این مورد از counting sort . که یک روش پایدار § است استفاده میکنیم برای این منظور میتوان از

شبه کد زیر استفاده کرد .

sort single characters‡

stable§

```

Data: S
Result: order
order ← array of size |S|
count ← zero array of size |Σ|
for i from 0 to |S|-1 do
  | count[S[i]] ← count[S[i]]+1
end
for j from 1 to |Σ|-1 do
  | count[j] ← count[j]+count[j-1]
end
for i from |S|-1 to 0 do
  | c ← S[i]
  | count[c] ← count[c]-1
  | order[count[c]] ← i
end
return order

```

Algorithm 15: SortCharacters(S)

همانطور که مشخص است مدت زمان اجرای این الگوریتم $O(|S| + |\Sigma|)$ است.

۲.۲.۱۶ کلاس های هم ارزی

در این قسمت مقدار c_i را تعریف میکنیم. در واقع c_i یعنی partial cyclic shift ای که از خانه شماره i در رشته مورد نظر شروع شده و به طول L جلو میرود. به مثال زیر دقت کنید

```

s=abcdefghi
c2-3=cdef

```

حال اگر c_i و c_j با هم برابر بودند باید مقدار کلاس هم ارزی آنها نیز یکسان باشد یعنی $class[i]==class[j]$ که عکس این حالت هم برقرار است به مثال زیر توجه کنید:

```
s=ababbbba
L=2
c0-2=ab
c2-2=ab
c0=c2
class[0]=class[2]
```

حال به محاسبه ی آرایه هم ارزی برای رشته ی زیر میپردازیم: $s=ababaa\$$

```
6 $
0 a
2 a
4 a
5 a
1 b
3 b
class=[1,2,1,2,1,1,0]
```

برای محاسبه ی این آرایه از شبه کد زیر استفاده میشود:

```

Data: S,order
Result: class
class ← array of size |S|
class[order[0]] ← 0
for i from 0 to |S|-1 do
    if  $S[\text{order}[i]] \neq S[\text{order}[i-1]]$  then
        | class[order[i]]=class[order[i-1]]+1
    else
        | class[order[i]]=class[order[i-1]]
    end
end
return class

```

Algorithm 16: ComputeCharClasses(S, order)

مدت زمان اجرای این الگوریتم $O(|S|)$ است.

۳.۲.۱۶ مرتب کردن شیفت های چرخشی دو برابر شده

همانطور که در ابتدای این بخش گفتیم در هر مرحله طول چرخش را دو برابر میکنیم و رشته های حاصل را مرتب میکنیم، باید به خاطر داشت که فقط قسمت اضافه شده به رشته مرتب میشود. ۳.۱۶

این به این دلیل است که قسما های قبل مرتب شده هستند و نیازی نیست که دوباره زمان صرف شود برای مرتب کردن آن ها. مقدار c'_i را اینگونه تعریف میکنیم که شیفت چرخشی که از مکان i شروع شده و به اندازه ۲ برابر طول L مرحله قبل ادامه پیدا میکند.

$$c_i$$

$$c'_i$$

$$c'_i = c_i c_{i+L}$$

به مثال زیر دقت کنید:

Sort Doubled cyclic Shifts^{۱۱}

```

S = ababaa$
L = 2
i = 2
ci = c2 = ab
ci+L = c2+2 = c4 = aa
c'i = c'2 = abaa = c2c4

```

برای محاسبه ی مجدد آرایه order از شبه کد زیر استفاده میکنیم و آرایه ی newOrder را برای این مقادیر تعریف میکنیم:

Data: S

Result: order

count \leftarrow zero array of size |S|

newOrder \leftarrow array of size |S|

```

for i from 0 to |S|-1 do
  | count[class[i]]  $\leftarrow$  count[class[i]]+1
end

for j from 1 to |Σ|-1 do
  | count[j]  $\leftarrow$  count[j]+count[j-1]
end

for i from |S|-1 to 0 do
  | start  $\leftarrow$  (order[i]-L+|S|) mod |S|
  | cl  $\leftarrow$  class[start] count[cl]  $\leftarrow$  count[cl]-1
  | newOrder[count[cl]]  $\leftarrow$  start
end

return newOrder

```

Algorithm 17: SortDoub led(S, L, order, class)

مدت زمان اجرای این الگوریتم $O(|S|)$ است.

۴.۲.۱۶ محاسبه مجدد کلاس هم ارزی

همانطور که دیدیم با دو برابر شدن طول چرخش آرایه order را نیز بر اساس رشته های جدید تغییر دادیم و بر اساس آن پیش رفتیم در این قسمت نیز باید آرایه مربوط به کلاس های هم ارزی را تغییر دهیم: در این قسمت دیگر نویسه ها را بررسی نمیکنیم در واقع باید چند نویسه که حاصل از شیفت چرھشی هستند را مقایسه کنیم برای اینکه از لحاظ زمانی و حافظه ایی بهینه باشد از تعریف کلاس هم ارزی مرحله قبل استفاده میکنیم و جفت هایی برای هر رشته تعریف میکنیم که عضو های این جفت ها کلاس های هم ارزی نویسه های تشکیل دهنده رشته هستند اگر جفت ها را به صورت (P_1, P_2) و (Q_1, Q_2) اگر شرط زیر برقرار باشد کلاس های جدید حاصل آنها باهم برابر است

$$(P_1 == Q_1) \text{ و } (P_2 == Q_2)$$

به مثال زیر توجه کنید:

```
S = ababaa$
class = [1, 2, 1, 2, 1, 1, 0]
c'_6 $a (0, 1)
c'_5 a$ (1, 0)
c'_4 aa (1, 1)
c'_0 ab (1, 2)
c'_2 ab(1, 2)
c'_1 ba (2, 1)
c'_3 ba (2, 1)
newClass = [3, 4, 3, 4, 2, 1, 0]
```

برای ساخت این آرایه از شبه کد زیر استفاده میکنیم:

```

Data: S
Result: order
n ← |newOrder|
newClass ← array of size n
newClass[newOrder[0]] ← 0
for i from 1 to n-1 do
  cur ← newOrder[i]
  prev ← newOrder[i-1]
  mid ← (cur+L)
  midPrev ← (prev+L)(mod n)
  if class[cur] ≠ class[prev] or class[mid] ≠ class[midPrev] then
    | newClass[cur] ← newClass[prev]+1
  else
    | newClass[cur] ← newClass[prev]
  end
end
return newClass

```

Algorithm 18: UpdateClasses(newOrder, class, L)

مدت زمان اجرای این الگوریتم $O(|S|)$ است.

۵.۲.۱۶ ساخت آرایه پسوندی مرحله نهایی

در آخر باید از مراحل که توضیح دادیم در کنار یکدیگر استفاده کنیم تا آرایه مورد نظر حاصل شود. برای این منظور از شبه کد زیر استفاده میکنیم. حلقه آمده شده در شبه کد را تا جایی ادامه میدهم که L از طول رشته

بیشتر شود

Data: S
Result: order

```

order ← SortedCharacters(S)
class ← ComputeCharClasses(S,order)
L ← 1
while L < |S| do
  | order ← SortDoubled(S,L,order,class)
  | class ← UpdateClass(order,class,L) L ← 2L
end
return newOrder

```

Algorithm 19: BuildSuffixArray(S)

مدت زمان اجرای این الگوریتم با توجه به مراحل قبل که مشخص شده میتوان نتیجه گرفت که برابر است

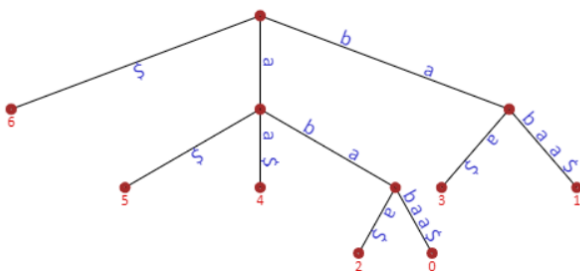
با: $O(|S| \log |S| + |\Sigma|)$

LCP ARRAY ۳.۱۶

قدم دیگری که برای ساخت درخت پسوندی** بهینه باید برداریم ساخت آرایه ی طولانی ترین پیشوند مشترک†† است. برای این منظور آرایه ای به اندازه ی $|S|-1$ میسازیم. خانه های آرایه را میزان اشتراک هر پسوند با پسوند بعدی خود پر میکند. این پسوند ها به همان ترتیبی هستند که در آرایه پسوندی ‡‡ ترتیبشان ذکر شده است.

۱.۳.۱۶ ایده کلی

با ساخت آرایه پسوندی در قسمت قبل دسترسی راحت تر و سریع تری به زیر رشته ها داریم. در درخت پسوندی نیز از اولین زیر رشته شروع میکنیم و به ریشه ی درخت اضافه میکنیم برای اضافه کردن زیر رشته های بعدی باید ابتدا با یال های موجود در درخت مقایسه شوند که در صورت اشتراک به آن یال وارد شوند. اگر طول زیر رشته ها طولانی باشد زمان زیادی برای مقایسه صرف میشود. در این قسمت میتوان از آرایه ی LCP استفاده کرد، به این صورت که در این آرایه تعداد اشتراک زیر رشته ها وجود دارد و نیاز به مقایسه نیست، برای اضافه شدن به درخت کافی است در یالی که با آن اشتراک دارد به میزان اشتراکشان از عمق یال کاسته و در آن نقطه اضافه شود. به مثال زیر توجه کنید:



شکل ۳.۱۶: suffix tree

suffix tree**

Longest common prefix††

suffix array‡‡

```

S = ababaa$
0 $
1 a$
2 aa$
3 abaa$
4 ababaa$
5 baa$
6 babaa$
lcp = [ 0,1,1,3,0,2]

```

یکی از خواصی که آرایه LCP دارد به صورت زیر تعریف میشود:

```

 $\forall i < j$ 
 $LCP(A[i], A[j]) \leq lcp[i]$ 
and
 $LCP(A[i], A[j]) \leq lcp[j - 1]$ 

```

۲.۳.۱۶ محاسبه ی آرایه LCP

برای محاسبه ی این آرایه میتوان از الگوریتم های ساده تری استفاده کردی با پیچیدگی زمانی $O(|S|^2)$ که بهینه نیست. از این رو برای افزایش سرعت از این ایده استفاده میکنیم: فرض کنید که h طولانی ترین پیشوند مشترک بین S_i و پسوند بعدی آن در آرایه ی پسوندی مربوط به رشته باشد. در نتیجه طولانی ترین پیشوند مشترک S_i و پسوند بعدی آن حداقل برابر $h-1$ است.

برای محاسبه از شبه کد های زیر استفاده میکنیم که در جلسات بعدی به بررسی جزئی تر میپردازیم.

```

Data: S,i,j,equal
Result: lcp
lcp ← max(0,equal)

while  $i+lcp < |S|$  and  $j+lcp < |S|$  do
  if  $S[i+lcp] == S[j+lcp]$  then
    | lcp ← lcp+1
  else
    | break
  end
end
return lcp

```

Algorithm 20: LCPOfSuffixes(S,i,j,equal)

```

Data: S,order
Result: class
pos ← array of size |order|

for  $i$  from 0 to  $|pos|-1$  do
  | pos[order[i]] ← i
end
return pos

```

Algorithm 21: InvertSuffixArray(order)

```

Data: S,order
Result: class
lcpArray ← array of size |S|-1
lcp ← 0
posInOrder ← InvertSuffixArray(order)
suffix ← order[0]
for i from 0 to |S|-1 do
  orderIndex ← posInOrder[suffix]
  if orderIndex == |S|-1 then
    lcp ← 0
    suffix ← (suffix+1) mod |S|
    continue
  nextSuffix ← order[orderIndex+1]
  lcp ← LCPOfSuffixes(S,suffix,nextSuffix,lcp-1)
  lcpArray[orderIndex] ← lcp
  suffix ← (suffix+1) mod |s|
end
return lcpArray

```

Algorithm 22: ComputeLCPArray(S,order)

مثال‌ها و شبه‌کدهای ذکر شده در این بخش از دوره‌ی مربوط به رشته، هفته‌ی چهارم برداشت شده است. برای مطالعه بیشتر به سایت زیر مراجعه کنید [algorithmsOnStrings]. همچنین ابزارهای محازی استفاده شده برای رسم شکل را می‌توانید در این قسمت مشاهده کنید. [ST].

جلسه ۱۷

SuffixTree

هستی کرمدل - ۱۳۹۹/۱/۳۱

جلسه قبل :

در آخر مباحث رشته در مورد ساختن بهینه suffixtree صحبت کردیم که در مرحله اولیه $O(n^2)$ داشت و می خواهیم آن را بهتر کنیم. از ساختن suffixarray شروع می شود که همه ی suffix های آن sort شده است. که در مرحله اول همه تک کاراکترها را sort می کردیم بعد دو برابر آن ها را تا به آخر برسیم . بعد برای استفاده درست از suffixarray یک LCParray ساختیم که آرایه ای برای حساب کردن تعداد اشتراکات بین دو suffix پشت سر هم است. ایده کلی این کار این است که بعد از مقایسه ی دو suffix ، دو suffix بعدی که مقایسه می شود حداقل تعداد اشتراکاتشان LCP قبلی منهای یک است.

پیشنیاز: LCParray

در ادامه مبحث LCParray به الگوریتم آن می پردازیم. دو suffix داریم و می خواهیم اشتراکاتشان را حساب کنیم، ورودی های تابع ما: LCP قبلی منهای یک به عنوان equal، شماره suffix مورد نظر و بعدی آن بعنوان i و j و string اصلی هستند. اول LCP را برابر(0،equal) می گذارد تا منفی نشود بعد می گوییم تا موقعی که کاراکتر بعدی j و i از طول string بیشتر نشده ، اگر برابر نبودند $LCP=LCP+1$. این روند تا وقتی ادامه دارد که به یکی برسند که برابر نیستند و return کنند .

۱.۱۷

LCPOfSuffixes(S, i, j, equal)

```

lcp ← max(0, equal)
while i + lcp < |S| and j + lcp < |S|:
  if S[i + lcp] == S[j + lcp]:
    lcp ← lcp + 1
  else:
    break
return lcp

```

شکل ۱.۱۷: پیشنهاد LCPArray

همچنین به یک چیز دیگر هم نیاز داریم چون وقتی index شروع suffix را داریم نخواهیم یکی یکی در text یا آرایه بگردیم، پس به یک `invertsuffixarray` نیاز داریم که یک آرایه برای position ها درست می کنیم. مثلاً اگر suffix اول ۶ باشد در خانه ۶ ام مقدار ۰ می گذارد. الان اگر بخواهیم ببینیم خانه ۶ ام در suffixarray کجاست در pos نگاه می کنیم خانه ۶ ام چه عددی دارد. الگوریتم pos : (ورودی آن همان suffixarray است).

۲.۱۷

InvertSuffixArray(order)

```

pos ← array of size |order|
for i from 0 to |pos| - 1:
    pos[order[i]] ← i
return pos

```

شکل ۲.۱۷: پیشنهاد LCPArray

:LCPArray

برای حساب کردن LCPArray ابتدا آرایه ای به طول $|S|-۱$ می سازیم که S همان string اصلی است. سپس pos را حساب می کنیم. بعد suffix ما در $order[۰]$ یعنی $order[۰]$ یک مکان است برای شروع. suffix بعد pos آن می شود همان مکانش در حال. nextsuffix suffixarray را حساب می کنیم که suffix بعدی آن است و LCP این دو را حساب می کنیم و در $LCPArray[orderindex]$ قرار می دهیم همچنین $suffix=suffix+۱$ می کنیم. حال دوباره به اول for برمی گردیم. اگر $orderindex$ به آخر رسید $LCP=۰$ و $suffix=suffix+۱$ می کنیم. زمان اجرای این الگوریتم OISI است. در LCP یک while و یک for داریم که این while در آن محاسبه می شود که OISI است. چرا OISI نمی شود؟ علت آن این است که LCP در هر مرحله می تواند هر چقدر می خواهد زیاد شود ولی یکی کم می شود تعداد کل دفعاتی که می تواند زیاد شود بیشتر OISI نیست.

ComputeLCPArray(S, order)

```

lcpArray ← array of size |S| - 1
lcp ← 0
posInOrder ← InvertSuffixArray(order)
suffix ← order[0]
for i from 0 to |S| - 1:
    orderIndex ← posInOrder[suffix]
    if orderIndex == |S| - 1:
        lcp ← 0
        suffix ← (suffix + 1) mod |S|
        continue
    nextSuffix ← order[orderIndex + 1]
    lcp ← LCPOfSuffixes(S, suffix, nextSuffix, lcp - 1)
    lcpArray[orderIndex] ← lcp
    suffix ← (suffix + 1) mod |S|
return lcpArray

```

شکل ۳.۱۷: ComputeLCPArray

: Buildingsuffixtree

تا اینجا suffixarray و LCParray را حساب کردیم. حال اول به root را اضافه می کنیم و $LCP=0$ است. پس به ارتفاع ۰ (که همان root است) می رویم و suffix بعدی را add می کنیم. و عدد suffix array را هم به انتهای آن اضافه می کنیم. مرحله بعدی آنقدر می آییم بالا که عمق کمتر از LCP شود.

پیاده سازی: suffixtree

برای بالا رفتن در درخت اولین شاخه ای که به آن نیاز داریم linkparent است که در node ذخیره می کنیم سپس بچه ها را در یک dictionary براساس کارکتر اولشان نگه می داریم همچنین عمق را هم ذخیره می کنیم. edgestart یعنی از کجا شروع می شود و از edgeend یعنی کجا تمام می شود.

class SuffixTreeNode:

```
SuffixTreeNode parent
Map<char, SuffixTreeNode> children
integer stringDepth
integer edgeStart
integer edgeEnd
```

شکل ۴.۱۷: Buildingsuffixtree

root را در ابتدا شکل می دهیم و در root قرار داریم. حال از اول suffixarray شروع می کنیم تا موقعی که عمق درخت بزرگتر از LCP است به بالا می رویم اگر عمق برابر LCP قبلی بود لازم نیست node جدید بسازیم وگرنه باید از همان جا بشکنیم و node جدید بسازیم.

۵.۱۷

STFromSA(S, order, lcpArray)

```
root ← new SuffixTreeNode(
  children = {}, parent = nil, stringDepth = 0,
  edgeStart = -1, edgeEnd = -1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| - 1:
  suffix ← order[i]
  while curNode.stringDepth > lcpPrev:
    curNode ← curNode.parent
  if curNode.stringDepth == lcpPrev:
    curNode ← CreateNewLeaf(curNode, S, suffix)
  else:
    edgeStart ← order[i - 1] + curNode.stringDepth
    offset ← lcpPrev - curNode.stringDepth
    midNode ← BreakEdge(curNode, S, edgeStart, offset)
    curNode ← CreateNewLeaf(midNode, S, suffix)
  if i < |S| - 1:
    lcpPrev ← lcpArray[i]
return root
```

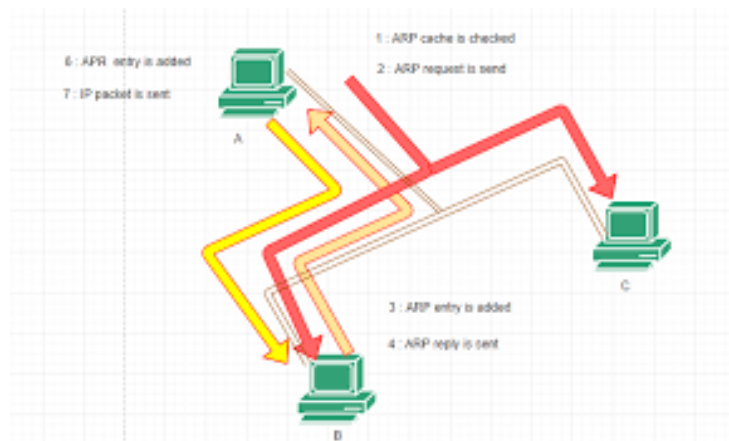
شکل ۵.۱۷: Buildingsuffixtree

:Suffixtreeorder

ساختن suffixtree از روی suffixarray در زمان خطی یعنی $O(S)$ انجام می شود و ساختن suffixtree از ابتدا $O(S \log S)$ است زیرا زمان آن برابر با $O(S \log S + S \log S)$ است.

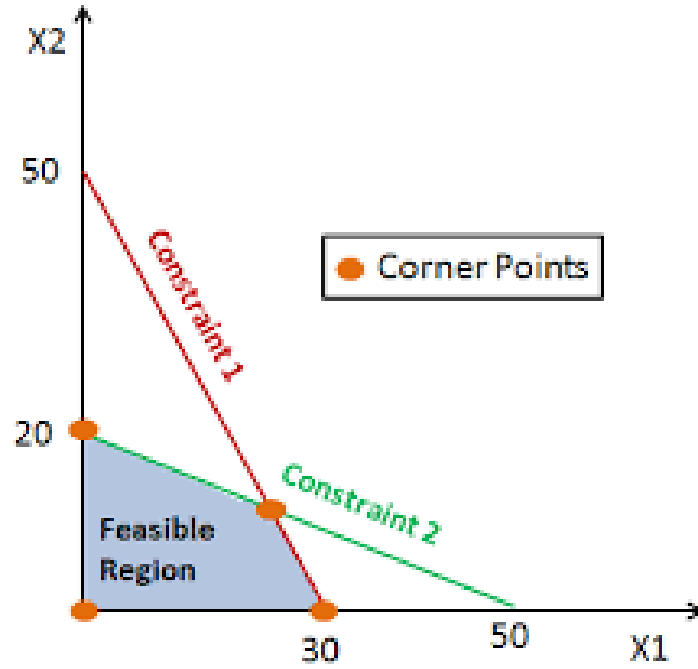
کلیت جلسه بعد:

شروع الگوریتم های پیشرفته با مباحث زیر: (۱) Flowinnetworks ۶.۱۷

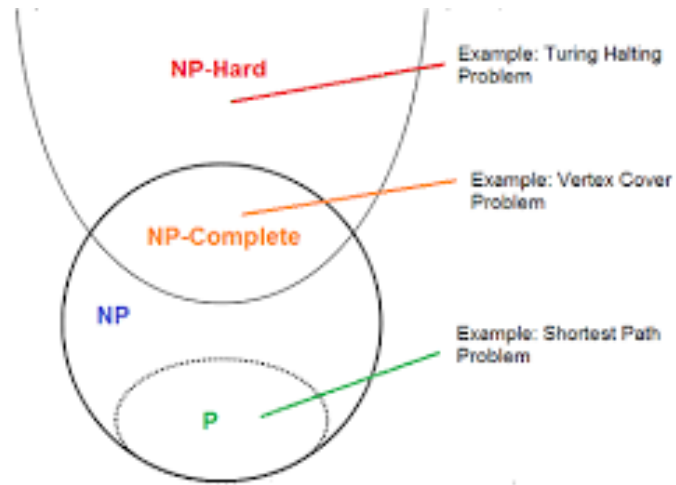


شکل ۶.۱۷: Flowinnetworks

(۲) Linearprogramming ۷.۱۷



شکل ٧.١٧ : Linearprogramming



This diagram assumes that $P \neq NP$

شکل ١٧.٨: NPcompleteproblems

CopingwithNPcompleteness (٤)

Streamingalgorithms(optional) (٥)

جلسه ۱۸

جریان در گراف

محمد مصطفی رستم خانی - ۱۳۹۹/۲/۲

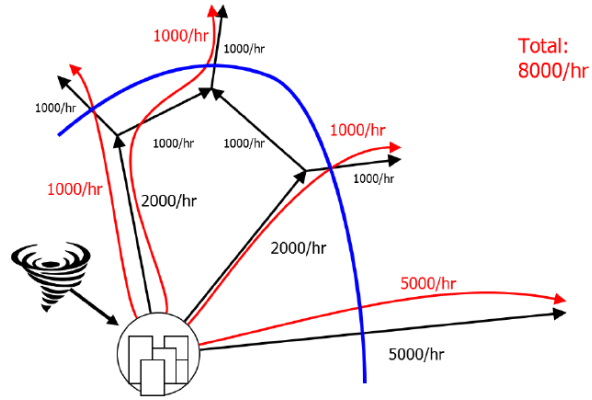
جزوه جلسه ۱۸ مورخ ۱۳۹۹/۲/۲ درس طراحی و تحلیل الگوریتم تهیه شده توسط محمد مصطفی رستم خانی. در جهت مستند کردن مطالب درس طراحی و تحلیل الگوریتم، بر آن شدیم که از دانشجویان جهت مکتوب کردن مطالب کمک بگیریم. هر دانشجو می‌تواند برای مکتوب کردن یک جلسه داوطلب شده و با توجه به کیفیت جزوه از لحاظ کامل بودن مطالب، کیفیت نوشتار و استفاده از اشکال و منابع کمک آموزشی، حداکثر یک نمره مثبت از بیست نمره دریافت کند. خواهشمند است نام و نام خانوادگی خود، عنوان درس، شماره و تاریخ جلسه در ابتدای این فایل را با دقت پر کنید. مطالبی که در ادامه آمده فقط جنبه راهنمایی شیوه استفاده از لاتک می‌باشد. خواهشمند است این پاراگراف و مطالب بعدی را از نسخه جزوه‌ای که تحویل می‌دهید، حذف کنید.

۱.۱۸ جریان در شبکه: (flows in network)

مثال:

طوفانی در راه است و قصد داریم شهر را خالی از سکنه کنیم. ولی محدودیت هایی داریم از جمله اینکه مسیر هایی که داریم هرکدام میتوانند جریانی را از خود عبور دهند و قادر به عبور جریانی بیشتر از خود نیستند. می خواهیم بیشترین تعداد افرادی را که می توانند شهر را تخلیه کرده و به جای امن بروند را بیابیم. اینگونه مسائل کاربرد جریان در شبکه را نشان می دهند. جریان در شبکه به شما این اجازه را می دهد که بتوانید بفهمید که

می توانید این کار را بهتر انجام دهید یا خیر.



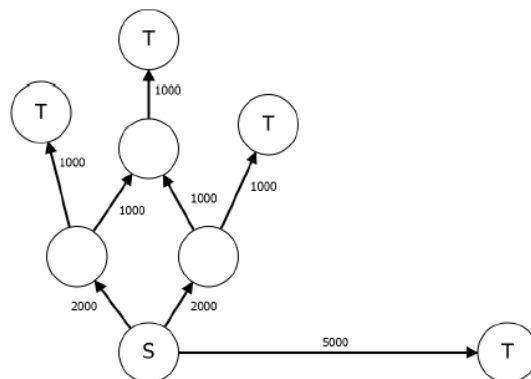
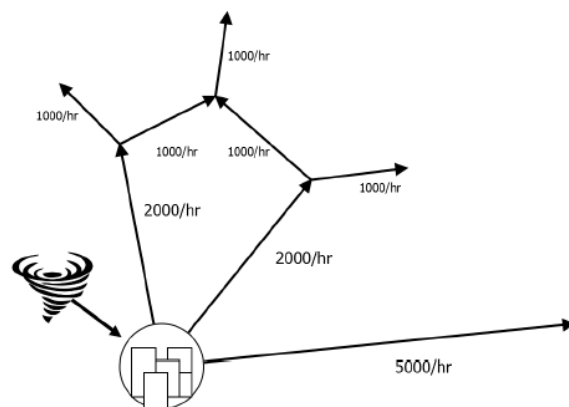
شکل ۱۰۱۸ : network

network ۲.۱۸

• تعریف:

شبکه (network): یک شبکه یک گراف جهت دار G (directed graph) است به گونه ای که:

۱. به هر یال آن به مانند e مقداری حقیقی و مثبت به عنوان ظرفیت (c_e) اختصاص داده می شود.
۲. تعداد یک یا بیشتر راس مبدا (source) دارد.
۳. یک یا بیشتر راس مقصد (sink) دارد.



شکل ۲.۱۸ : example

جریان-ترافیک (flows-traffic): یک جریان در گراف اختصاص یک عدد حقیقی به هر یال e است به گونه ای که در شروط زیر صدق کند:

۱. برای هر یال e مانند e داریم: (rate limitation)

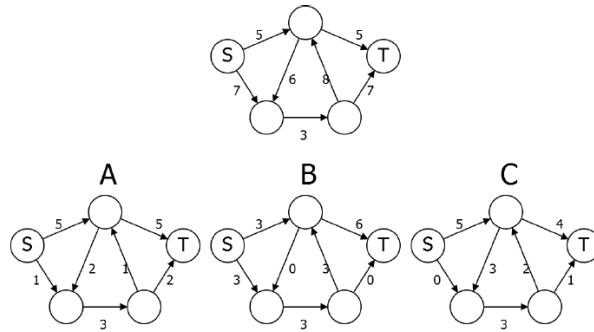
$$0 \leq f_e \leq c_e \quad (۱.۱۸)$$

۲. برای هر یال به مانند برای هر راس v که مبدا و مقصد نیستند داریم:

$$\sum_{v \text{ into } (e)} f_e = \sum_{v \text{ of out } (e)} f_e \quad (۲.۱۸)$$

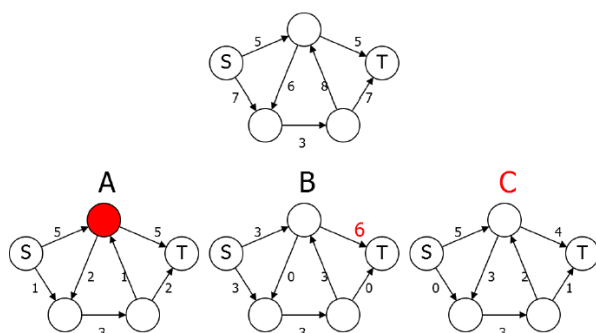
(conservation of flow): یعنی باید مجموع جریان های ورودی و خروجی به هر راس با هم برابر باشند .

مثال: کدام یک از flow های زیر برای گراف داده شده معتبر هستند؟



شکل ۳.۱۸: flow example

توضیح: در گراف A مقدار جریان ورودی و خروجی برای راس بالایی گراف برابر نیستند و از آنجایی که این راس جزو source و sink نیست پس این جریان معتبر نیست. در گراف B مقدار جریان برای یال بالایی که به target وارد می شود بیشتر از ظرفیت آن است. گراف C همه ی شروط لازم برای یک جریان معتبر را رعایت کرده است.



شکل ۱۸.۴ : example flow

• کاربرد هایی از Flow Max :

- ترافیک در شبکه ی حمل و نقل شهری
- جریان ورودی و خروجی در خطوط نیرو
- جریان در لوله های آب
- شبکه ی اینترنت

اندازه ی جریان (flow size):

تعریف: برای یک جریان f اندازه ی جریان به صورت زیر تعریف می شود:

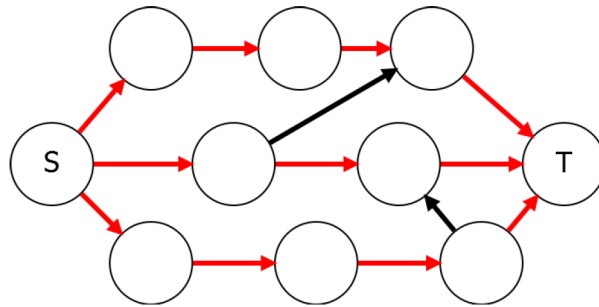
$$|f| = \sum_{\text{source } a \text{ of out } (e)} f_e - \sum_{\text{source } a \text{ into } (e)} f_e \quad (۳.۱۸)$$

$$|f| = \sum_{\text{sink } a \text{ into } (e)} f_e - \sum_{\text{sink } a \text{ of out } (e)} f_e \quad (۴.۱۸)$$

۳.۱۸ گراف باقیمانده (residual graph):

برای به دست آوردن flow max ابتدا باید پیدا کنیم که آیا راهی از source به target وجود دارد یا خیر. برای این کار می توان از BFS یا DFS استفاده کرد.

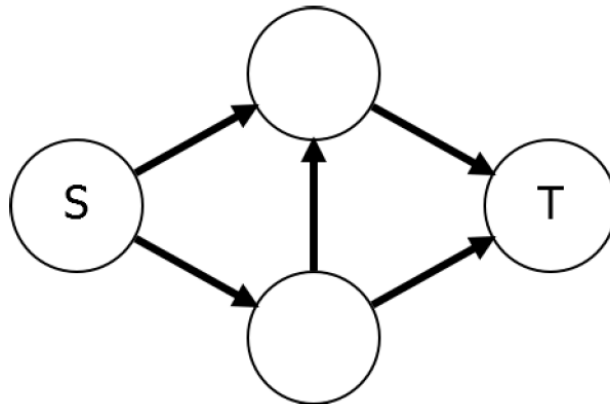
[All capacities are 1]



شکل ۵.۱۸: graph

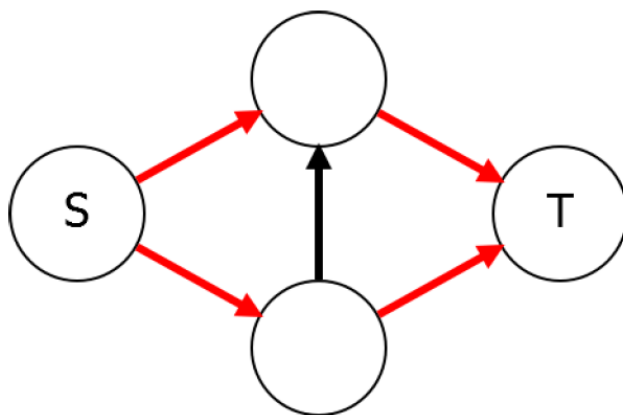
برای مثال در گراف بالا flow max برابر است با ۳.

برای مثالی دیگر گراف زیر را در نظر بگیرید.



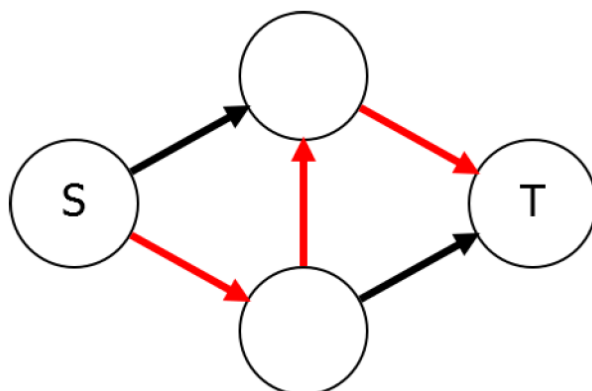
شکل ۶.۱۸: graph

در این گراف بیشینه ی جریان برابر است با ۲.



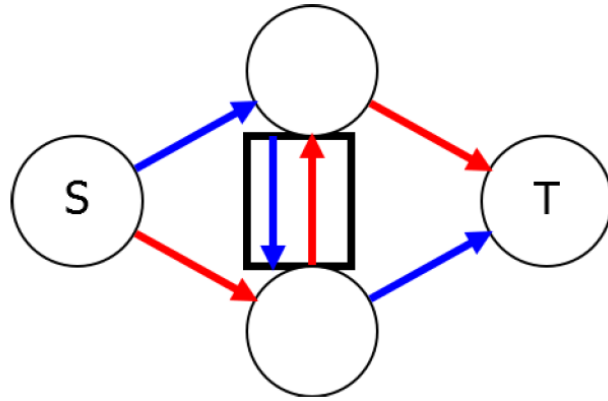
شکل ۷.۱۸: graph

ولی اگر یک راه دیگر پیدا کنیم این مقدار را متفاوت پیدا خواهیم کرد.



شکل ۸.۱۸: graph

اگر این جریان را به گراف اضافه کنیم بیشینه ی جریان ۱ خواهد شد. برای رفع این مشکل از گراف باقیمانده استفاده می کنیم. به صورت زیر:



شکل ۹.۱۸ : graph

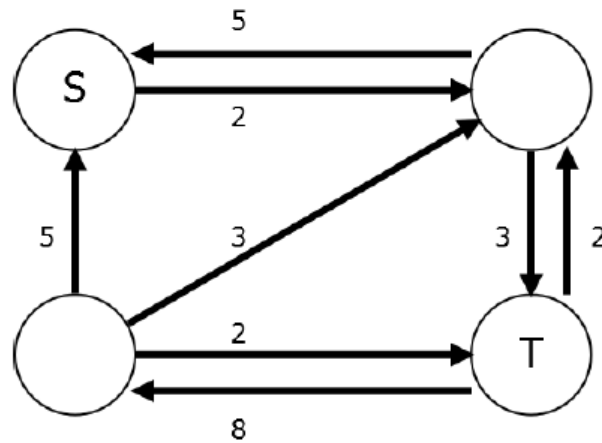
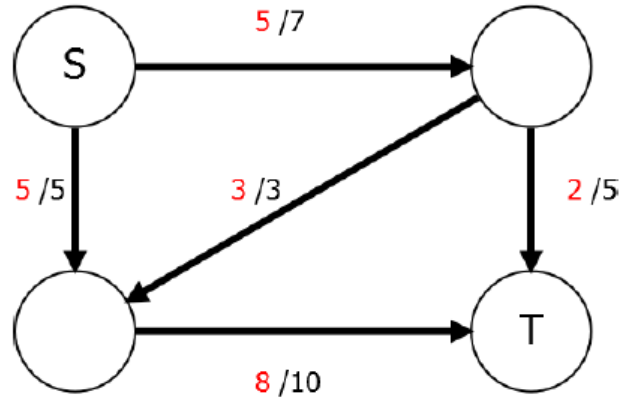
با استفاده از این گراف می توان جریان وسط را خنثی کرد.

برای یک گراف G و یک جریان f داده شده می توان گراف باقیمانده G_f را به این صورت به دست آورد:
 روی هر یال اگر می توانستیم به جریان روی آن یال اضافه کنیم یک یال با آن ظرفیتی که می توانیم اضافه کنیم می کشیم و به ازای هر یالی روی گراف اصلی یالی را با همان ظرفیت ولی در جهت برعکس برای کنسل کردن آن یال به گراف باقیمانده اضافه می کنیم. به عبارتی دیگر: برای هر یال $e(u,v)$ روی گراف G_f یال های زیر را دارد:

۱. یک یال از u به v با ظرفیت $f_e - C_e$ مگر در حالتی که $f_e = C_e$

۲. یک یال از v به u با ظرفیت f_e مگر در حالتی که $f_e = 0$

برای مثال گراف بالایی مربوط به یک جریان و گراف پایینی مربوط به باقیمانده ی آن است.



شکل ۱۰.۱۸ : residual network

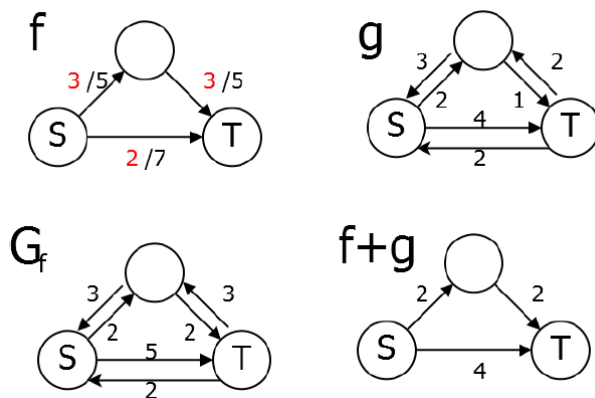
۴.۱۸ جریان باقیمانده (residual flow):

اگر گراف G و جریان f را داشته باشیم، آنگاه هر جریان g روی G_f (گراف باقیمانده) می تواند به جریان f اضافه شود و جریان جدید نیز یک جریان معتبر روی G است. به گونه ای که:

۱. f_e به g_e اضافه می شود.

۲. g_e از f_e کم می شود.

مثال: در شکل زیر جمع دو جریان نشان داده شده است و همان طور که می توان مشاهده کرد جریان جدید نیز یک جریان معتبر است.



شکل ۱۱.۱۸ : residual network

قضیه: اگر گراف G و جریان f روی آن و جریان g روی G_f داده شده باشند آنگاه:

۱. $f+g$ یک جریان روی گراف اصلی است.

$$|f+g|=|f|+|g| \quad ۲.$$

۳. تمام جریانات روی گراف اصلی به این شکل هستند.

اثبات:

$$f_e + g_e \leq f_e + (C_e - f_e) = C_e$$

$$f_e - g_e \geq f_e - f_e = 0$$

\Rightarrow So $f + g$ is a flow.

۵.۱۸ maxflow: and Mincut

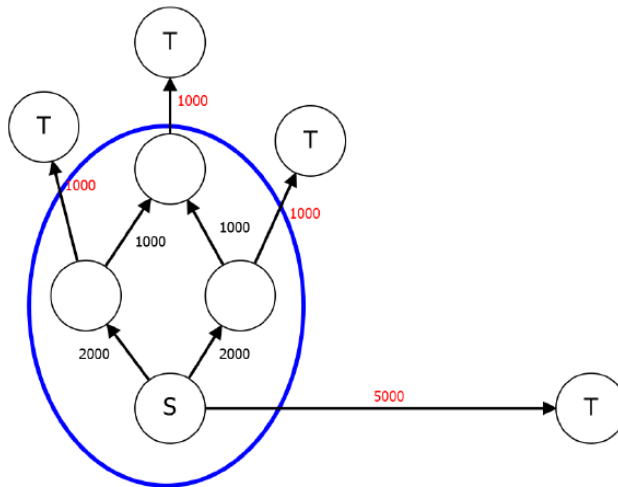
برای پیدا کردن maxflow ما به راهی برای تشخیص اینکه جریان به دست آمده بیشینه است داریم. برای این منظور از تکنیک هایی برای محدود کردن اندازه ی maxflow استفاده می کنیم.

برش: (cut)

تعریف: روی گراف G یک برش (cut) C به مجموعه ای از راس ها گفته می شود به گونه ای که شامل همه ی source ها باشند و شامل هیچکدام از sink ها نباشند. اندازه ی یک برش به صورت زیر به دست می آید:

$$|C| = \sum_{C \text{ of out } e} C_e \quad (۵.۱۸)$$

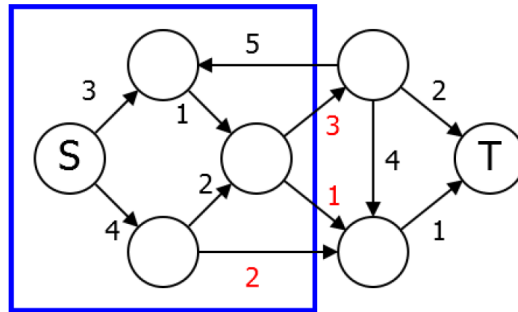
برای مثال اندازه ی cut در شکل زیر برابر با ۸۰۰۰ است.



شکل ۱۲.۱۸ : residual network

مثال:

$$1 + 2 + 3 = 6.$$



شکل ۱۳.۱۸ : residual network

قضیه: فرض کنید G یک گراف باشد. آنگاه برای هر جریان f و هر برش C داریم:

$$|f| \leq \text{Maxflow} \leq |C| \quad (۶.۱۸)$$

به عبارت دیگر با این روش می توانیم یک حد بالا برای maxflow به دست آوریم.

قضیه: برای هر گراف G داریم:

$$\text{maxflow} |f| = \text{mincut} |C| \quad (۷.۱۸)$$

یعنی اندازه ی maxflow با اندازه ی mincut برابر است. حالت خاص:
اگر maxflow برابر با صفر باشد آنگاه $(\text{maxflow}=0)$:

۱. هیچ مسیری از source به sink وجود ندارد.
۲. اگر C مجموعه ی رئوس قابل دسترس از source باشد آنگاه $|C|=0$.

جلسه ۱۹

الگوریتم های پیدا کردن flow

آرمین غلام پور - ۱۳۹۹/۲/۷

جزوه جلسه ۱۹ ام مورخ ۱۳۹۹/۲/۷ درس طراحی و تحلیل الگوریتم تهیه شده توسط آرمین غلام پور.

۱.۱۹ Ford Fulkerson

ایده ی کلی الگوریتم به این صورت است:

۱. flow اولیه را صفر قرار دهید

۲. مکرراً flow اضافه کنید

۳. مرحله ی ۲ را تا جایی که دیگر مسیری از نود شروع به نود پایان نباشد ادامه دهید

در این الگوریتم برای پیدا کردن مسیر در هر مرحله از الگوریتم dfs استفاده میکنیم. همچنین در هر مرحله پس از پیدا کردن جریان یک مسیر مقادیر جریان های گراف را به روز رسانی میکنیم.

شبه کد الگوریتم * فورد فولکرسون ۲۹ :

Data: Given a Network $G = (V, E)$ with flow capacity c , a source node s , and a sink node t

Result: Compute a flow f from s to t of maximum value

$f(u, v) \leftarrow 0$ for all edges (u, v)

; **while** there is a path p from s to t in G_f **do**

 Find $c_f(p) = \min(c_f(u, v) : (u, v)$

 in p)

 ;

for each edge (u, v)

 in p **do**

$f(u, v) \leftarrow f(u, v) + c_f(p)$

 ;

$f(v, u) \leftarrow f(v, u) - c_f(p)$

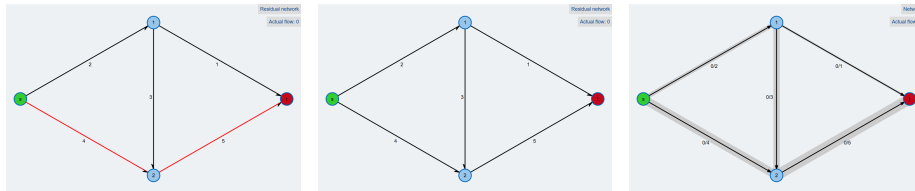
 ;

end

end

Algorithm 23: Ford Fulkerson Algorithm [pseudocodewebsite]

مثال زیر برای یک مساله به صورت مرحله به مرحله با الگوریتم فورد حل شده است:

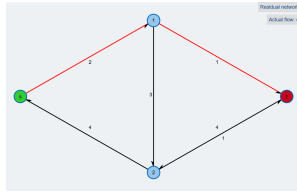


شکل ۱۹:۳ step ۳

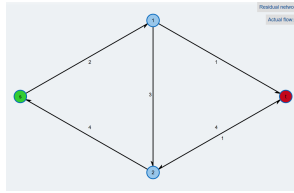
شکل ۱۹:۲ step ۲

شکل ۱۹:۱ step ۱

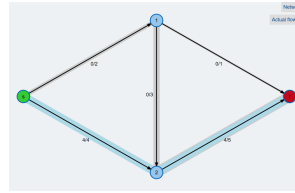
pseudocode*



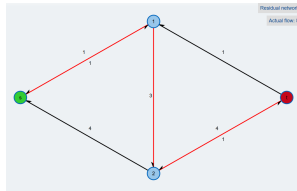
شکل ۱۹.۶: step ۶



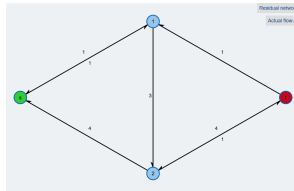
شکل ۱۹.۵: step ۵



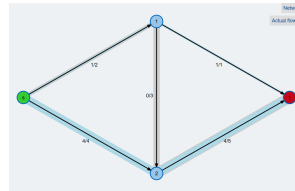
شکل ۱۹.۴: step ۴



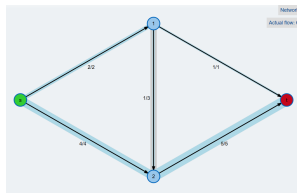
شکل ۱۹.۹: step ۹



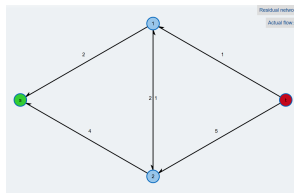
شکل ۱۹.۸: step ۸



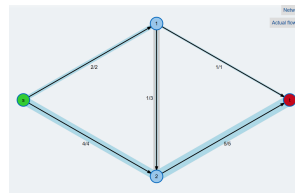
شکل ۱۹.۷: step ۷



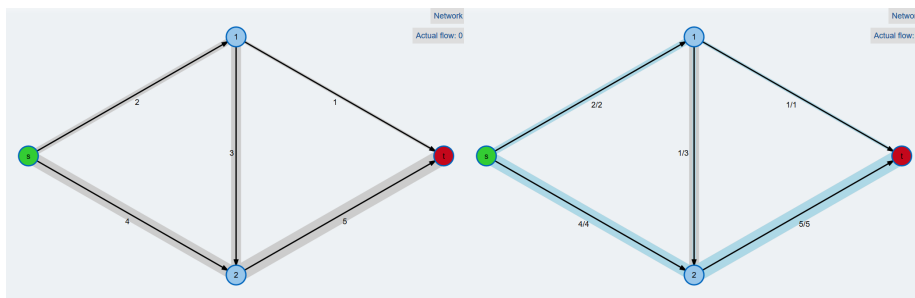
شکل ۱۹.۱۲: step ۱۲



شکل ۱۹.۱۱: step ۱۱



شکل ۱۹.۱۰: step ۱۰



شکل ۱۹.۱۴: step ۱۴

شکل ۱۹.۱۳: step ۱۳
[visualizationwebsite]

برخی از ویژگی های الگوریتم فورد فولکرسون:

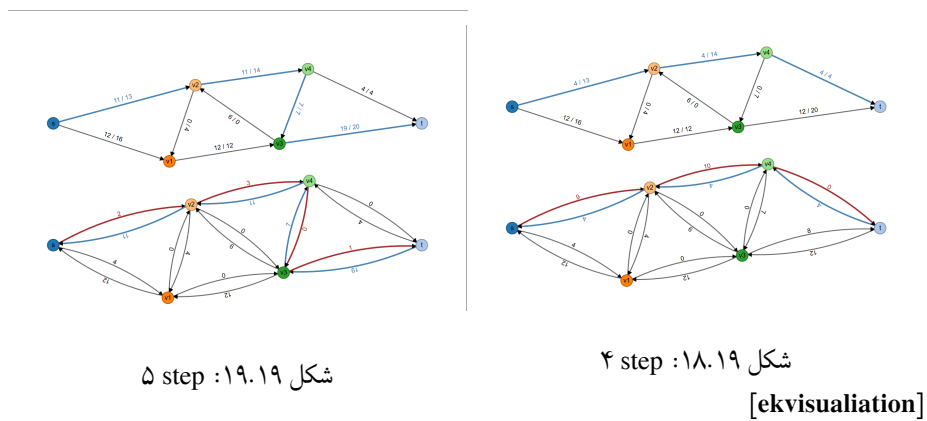
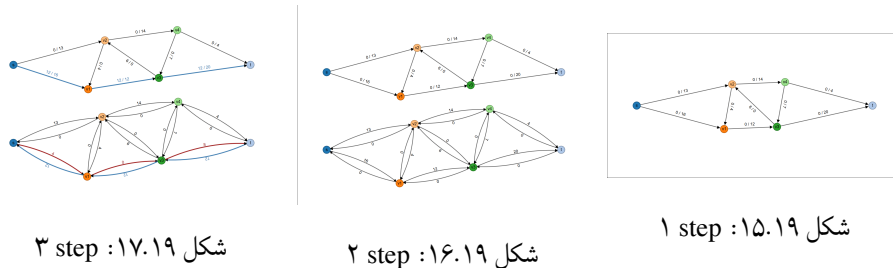
۱. فقط جواب های integer را پیدا میکند
۲. اردر زمانی اش $O(|E||F|)$ هست
۳. اگر جریان ها مقادیر بزرگی داشته باشند، به علت استفاده از الگوریتم dfs ممکن است طول بکشد و بهینه نباشد

راه حل بهبود الگوریتم برای مقادیر بزرگ جریان:

استفاده از الگوریتم Edmonds-Karp

۲.۱۹ Edmonds-Karp

تنها فرق این الگوریتم با الگوریتم فورد، استفاده ی bfs به جای dfs است. با استفاده از این الگوریتم بهینه سازی برای جریان های بزرگ صورت میگیرد.



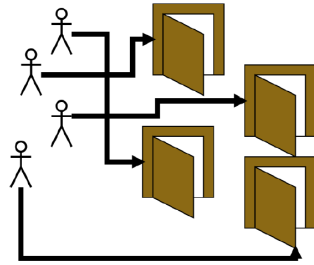
جلسه ۲۰

+ Applications Flow Network MeasureIt

فاطمه احمدی - ۱۳۹۹/۲/۹

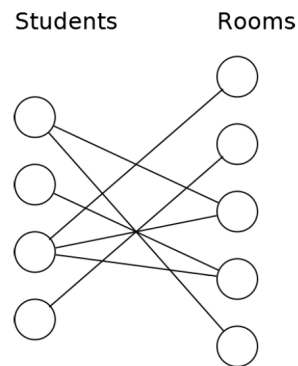
جزوه جلسه ۲۰ مورخ ۱۳۹۹/۲/۹ درس طراحی و تحلیل الگوریتم تهیه شده توسط فاطمه احمدی. در جهت مستند کردن مطالب درس طراحی و تحلیل الگوریتم، بر آن شدیم که از دانشجویان جهت مکتوب کردن مطالب کمک بگیریم. هر دانشجو می‌تواند برای مکتوب کردن یک جلسه داوطلب شده و با توجه به کیفیت جزوه از لحاظ کامل بودن مطالب، کیفیت نوشتار و استفاده از اشکال و منابع کمک آموزشی، حداکثر یک نمره مثبت از بیست نمره دریافت کند. خواهشمند است نام و نام خانوادگی خود، عنوان درس، شماره و تاریخ جلسه در ابتدای این فایل را با دقت پر کنید. مطالبی که در ادامه آمده فقط جنبه راهنمایی شیوه استفاده از لاتک می‌باشد. خواهشمند است این پاراگراف و مطالب بعدی را از نسخه جزوه‌ای که تحویل می‌دهید، حذف کنید.

در این جلسه به بررسی برخی از کاربرد های جریان شبکه ای می پردازیم. برای شروع به مثال زیر توجه کنید: فرض کنید می‌خواهیم دانشجویان را در خوابگاه در اتاق ها قرار دهیم. در این مثال بررسی می‌کنیم به هر دانشجویی چه اتاقی بدهیم که در آخر حداکثر رضایت را داشته باشند



شکل ۱.۲۰: matching dormitory

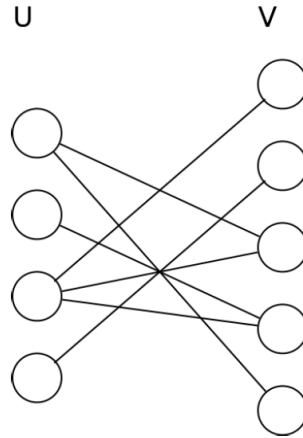
برای حل این مسئله به گرافی دو بخشی به شکل زیر نیاز داریم:



ابتدا تعریفی از گراف دو بخشی ارائه می دهیم:

۱.۲۰ گراف دو بخشی

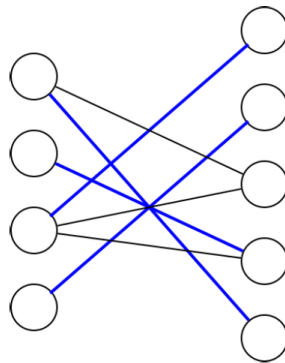
گراف دو بخشی گرافی است که می توان راس های آن را به دو دسته U و V تقسیم کرد به طوری که همه یال های بین راس های U و V باشد و هر کدام از دسته ها بین راس های خودشان یالی نداشته باشند.



شکل ۲.۲۰: Bipartite Graph

۲.۲۰ تطابق در گراف

تطابق در گراف ، مجموعه ای از یال هاست که هیچ دو یالی سر مشترک ندارند. به طور مثال شکل زیر یک تطابق را از گراف داده شده نشان می دهد.



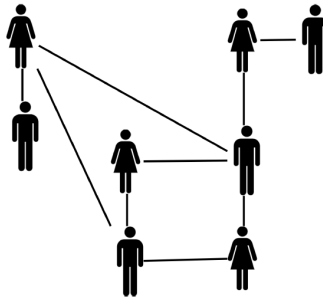
شکل ۲.۲۰: matching

در مثال خوابگاه رابطه بین اتاق و دانشجو مانند یک گراف دو بخشی است و برای حل مسئله باید تطابق یا همان Matching گراف را بیابیم، زیرا هر دانشجویی فقط باید یک اتاق داشته باشد و هر اتاق نیز فقط متعلق به یک دانشجو است و حداکثر رضایت را نیز با پیدا کردن Matching با بیشترین یال بدست می آید به طوری که هر راس از U به یک راس از V متصل باشد.

دو مورد زیر نیز از نمونه مسائلی است که با گراف دو بخشی و تطابق حل می شود:

۳.۲۰ Match Making

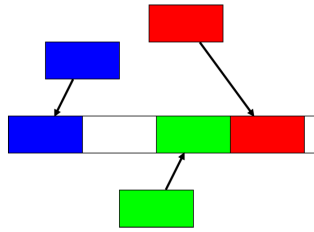
این مسئله به این صورت است که بین تعدادی زن و مرد باید طوری اتصال برقرار کنیم که هر مرد فقط با یک زن و هر زن فقط با یک مرد تطابق (match) یابد.



شکل ۳.۲۰: Match Making

۴.۲۰ Scheduling

این مسئله، مسئله برنامه ریزی درس هایی است که در یک کلاس مشخص برگزار می شود. طبیعتاً برنامه باید به صورتی باشد که در هر زمان فقط یک درس در آن کلاس تشکیل شود و برای هر درس هم فقط یک زمان تعیین گردد.

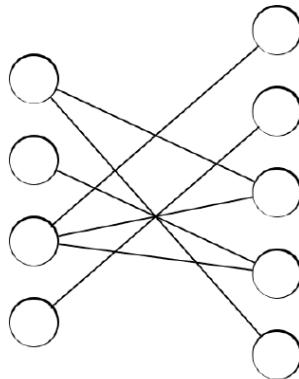


شکل ۵.۲۰: Scheduling

حال به بررسی راه حل می پردازیم. چگونه باید matching با حداکثر تعداد یال را از گراف دو بخشی مسئله پیدا کنیم؟؟

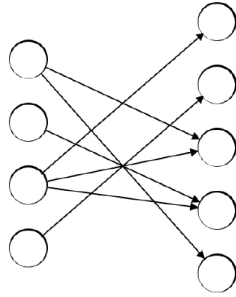
۵.۲۰ Find Maximum Matching

step ۱ : مطابق شکل گراف دو بخشی مسئله را رسم می کنیم:



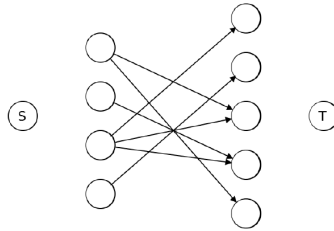
شکل ۶.۲۰: Step ۱

step ۲ : مطابق شکل برای یال ها جهت تعیین می کنیم به طوری که جهت همه یال ها از U به V باشد:



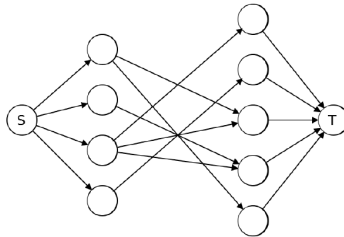
شکل ۷.۲۰: Step ۲

step ۳ : مطابق شکل دو راس به عنوان مبدا یا Source و مقصد یا Target اضافه می کنیم:



شکل ۸.۲۰: Step ۳

step ۴ : مطابق شکل از راس مبدا به همه راس های U و از همه راس های V به راس مقصد یال رسم می کنیم:



شکل ۹.۲۰: Step ۴

حال مطابق آنچه در جریان شبکه ای گفته شد بیشترین جریان یا همان Maxflow را بدست می آوریم که این مقدار با اندازه matching برابر است. به این دلیل که به همه راس های U فقط یک یال وارد شده و فقط هم یک یال می تواند خارج شود و این محدودیت برای راس های V وجود دارد، بنابراین به ازای هر $flow$ یک معادل $matching$ داریم و بالعکس، به همین دلیل است که $MaxFlow$ با $Maximum Matching$ برابر است.

الگوریتم پیدا کردن $matching$ نیز به صورت زیر است:

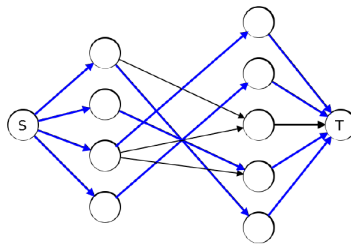
Data: Bipartite Graph

Result: Maximum Matching

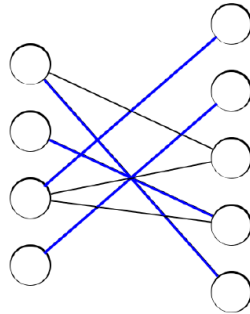
```
BipartiteMatching(G)
Construct corresponding network G
Compute Maxflow(G)
Find corresponding matching M
return M
```

Algorithm 24: How to find maximum matching

بر اساس maxflow که به دست می آید آن یال هایی که جریان آن ها یک است در $matching$ هستند و نتیجه به صورت زیر خواهد بود:



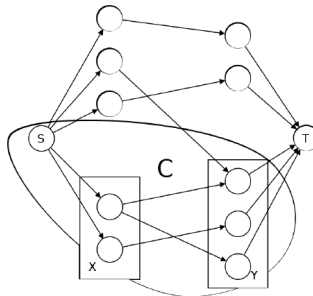
شکل ۱۰.۲۰ : Flow



شکل ۱۱.۲۰: Matching

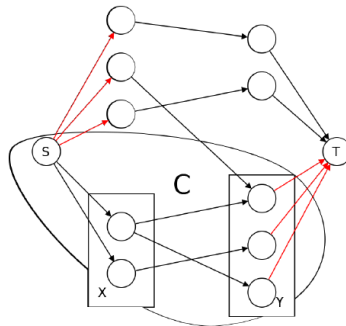
۶.۲۰ Maxflow-Mincut

یک راه دیگر یافتن maximum matching از طریق mincut است. برای این کار مطابق شکل دو مجموعه راس X و Y تعریف می‌کنیم:



شکل ۱۲.۲۰: Maxflow-Mincut

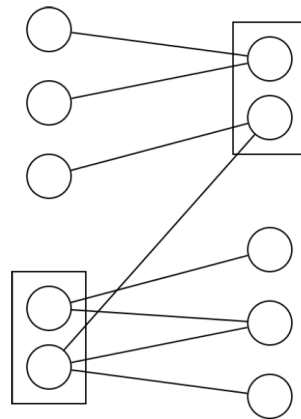
وقتی maxflow پیدا می‌کنیم، هر maxflow یک cut معادل دارد و هر cut در گراف دوبخشی دو قسمت دارد، یک قسمت در U و یک قسمت در V . هر موقع mincut پیدا کنیم یا به راس‌های خارج cut در U متصل است یا به راس‌های داخل cut در V متصل است، به همین دلیل و مطابق شکل زیر اندازه cut برابر است با مجموع تعداد راس‌های U که عضو X نیستند با تعداد راس‌های Y که همان maxflow نیز هست.



Maxflow-Mincut : شکل ۱۳.۲۰

Konig's Theorem ۷.۲۰

طبق قضیه König در هر گراف دو بخشی اگر k اندازه maximal matching باشد، آنگاه وجود دارد مجموعه ای از k راس که همه یال های گراف حداقل یک سر آن ها به یکی از راس های این مجموعه متصل باشد. (به مجموعه این راس ها Cover Set نیز گفته می شود)



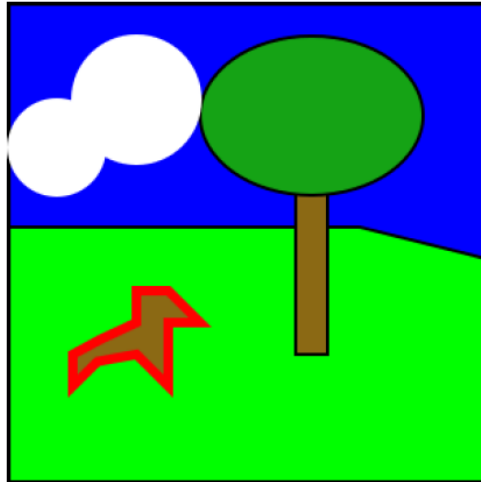
Konig's Theorem : شکل ۱۴.۲۰

۸.۲۰ The Marriage Lemma

اگر یک گراف دو بخشی داشته باشیم به طوری هر دو قسمت U و V در گراف n راس داشته باشند، این گراف یک perfect matching دارد مگر اینکه یک مجموعه S وجود داشته باشد به طوری که a راس از U و b راس از V را شامل باشد و a کمتر از b یا b کمتر از a باشد و نتوانیم یک به یک نظیر کنیم.

۹.۲۰ Image Segmentation

در این مسئله یک تصویر به ما می دهند و می خواهیم background و foreground را از هم تشخیص دهیم.



شکل ۱۵.۲۰: Image

برای هر پیکسل دو مشخصه تعیین می کنیم:

a: احتمال اینکه foreground باشد.

b: احتمال اینکه background باشد.

در این مسئله تصویر با مشخصه های هر پیکسل به ما داده می شود و ما می خواهیم طوری پیکسل ها

را به دو دسته background و foreground تقسیم کنیم که عبارت زیر بیشینه باشد.

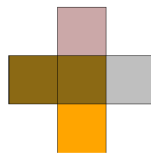
$$\sum_{v \in \mathcal{F}} a_v + \sum_{v \in \mathcal{B}} b_v$$

برای مثال جدول زیر را داریم:

v	1	2	3
a_v	3	5	6
b_v	4	3	5

شکل ۱۶.۲۰: Example

در این مثال ساده به راحتی قابل محاسبه است هر پیکسلی که a آن بزرگتر از b آن باشد foreground است و در غیر این صورت background است. مسئله دیگر پیکسل های اطراف هر پیکسل است که باید با یک مشخصه احتمالی دیگر به نام p به این پیکسل مربوط باشد.



شکل ۱۷.۲۰: Pixels

در این حالت ، با وجود p مسئله این می شود که عبارت زیر را بیشینه کنیم:

$$\sum_{v \in \mathcal{F}} a_v + \sum_{v \in \mathcal{B}} b_v - \sum_{v \in \mathcal{F}, w \in \mathcal{B}} p_{vw}$$

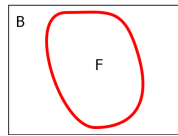
برای پیشرفت در حل مسئله عبارت زیر را از عبارت بالا کم می کنیم:

$$- \left(\sum_{v \in \mathcal{F}} b_v + \sum_{v \in \mathcal{B}} a_v + \sum_{v \in \mathcal{F}, w \in \mathcal{B}} p_{vw} \right)$$

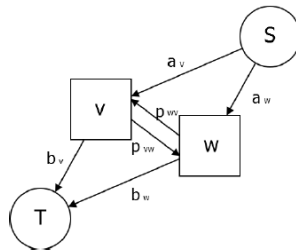
حال مسئله تغییر می کند و به عبارت زیر تبدیل می شود که این بار باید آن را مینیمم کنیم:

$$\sum_{v \in \mathcal{F}} b_v + \sum_{v \in \mathcal{B}} a_v + \sum_{v \in \mathcal{F}, w \in \mathcal{B}} p_{vw}$$

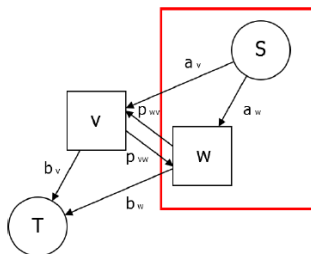
حال مسئله برای ما قابل حل می شود زیرا می خواهیم پیکسل ها را دو بخش کنیم و در عین حال عبارت را مینیمم کنیم ، این مسئله را می توانیم از طریق mincut انجام دهیم.



برای حل از طریق mincut باید پیکسل ها را به یک شبکه به طور مثال مطابق شکل زیر تقسیم کنیم:



در مرحله بعد به طور دلخواه یک cut به نام c انتخاب می کنیم که در شکل نیز می بینید:



اندازه cut از فرمول زیر محاسبه می شود:

$$\sum_{v \in C} b_v + \sum_{v \notin C} a_v + \sum_{v \in C, w \notin C} p_{vw}$$

و در این صورت به این نتیجه می رسیم که هر آنچه در cut است همان مجموعه F است و هر آنچه خارج از cut است همان مجموعه B می باشد.
در ادامه الگوریتم حل این مسئله را می بینید:

```

ImageSegmentation(av , bv , pvw )
Construct corresponding network G
Compute a maxflow f for G
Compute residual Gf
Let C be the collection of vertices
reachable from s in Gf
return F = C & B = not C

```

Algorithm 25: Image Segmentation

Measure It ۱۰.۲۰

در ادامه به توضیحات مختصری از measure it می پردازیم.
developer ها معمولا در تلاشند تا کد خود را بهینه کنند ، در این راستا دانستن اینکه هر دستور یا قطعه کد

چه مدت زمانی طول می کشد بسیار به آن ها کمک می کند.

MeasureIt ابزاری است که توسط آقای Vance Morrison طراحی شده است تا در این زمینه به developer ها کمک کند. به عنوان مثال دستیابی به ReaderWriterLock زمان بیشتری را می طلبد تا از قفل / مانیتور ساده استفاده کنید. با استفاده از MeasureIt می توان به اطلاعات دیگری از قبیل آنچه در شکل زیر آمده نیز دسترسی پیدا کرد:

ComputerSpecs	
Name: IGORDM2 Manufacturer: LENOVO Model: 4291CB5 Operating System: Microsoft Windows 7 Enterprise Version: 6.1.7601 ServicePack: 1 ...	
Name	IGORDM2
Manufacturer	LENOVO
Model	4291CB5
OperatingSystem	Microsoft Windows 7 Enterprise
OperatingSystemVersion	6.1.7601
OperatingSystemServicePack	1
NumberOfDisks	1
SystemDiskModel	INTEL SSDSA2BW160G3L
NumberOfProcessors	1
ProcessorName	Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz
ProcessorDescription	Intel64 Family 6 Model 42 Stepping 7
ProcessorClockSpeedMhz	2801
MemoryMBytes	16267
L1KBytes	64
L2KBytes	256

StatsCollection																	
PerformanceMeasurement.StatsCollection																	
Measurements																	
Dictionary<String,Stats> (2 items)																	
Key	Value																
new guid [count=25]	<table border="1"> <thead> <tr> <th colspan="2">Stats</th> </tr> </thead> <tbody> <tr> <td>mean=0.009 median=0.016 min=0.000 max=0.384 sdtdev=0.014 samples=1000</td> <td></td> </tr> <tr> <td>Minimum</td> <td>-4.768372E-10</td> </tr> <tr> <td>Maximum</td> <td>0.384</td> </tr> <tr> <td>Median</td> <td>0.016</td> </tr> <tr> <td>Mean</td> <td>0.008735999</td> </tr> <tr> <td>StandardDeviation</td> <td>0.01432768</td> </tr> <tr> <td>Count</td> <td>1000</td> </tr> </tbody> </table>	Stats		mean=0.009 median=0.016 min=0.000 max=0.384 sdtdev=0.014 samples=1000		Minimum	-4.768372E-10	Maximum	0.384	Median	0.016	Mean	0.008735999	StandardDeviation	0.01432768	Count	1000
Stats																	
mean=0.009 median=0.016 min=0.000 max=0.384 sdtdev=0.014 samples=1000																	
Minimum	-4.768372E-10																
Maximum	0.384																
Median	0.016																
Mean	0.008735999																
StandardDeviation	0.01432768																
Count	1000																
TypeOf(Guid).ToString() [count=25]	<table border="1"> <thead> <tr> <th colspan="2">Stats</th> </tr> </thead> <tbody> <tr> <td>mean=0.117 median=0.120 min=0.104 max=1.468 sdtdev=0.045 samples=1000</td> <td></td> </tr> <tr> <td>Minimum</td> <td>0.104</td> </tr> <tr> <td>Maximum</td> <td>1.468</td> </tr> <tr> <td>Median</td> <td>0.12</td> </tr> <tr> <td>Mean</td> <td>0.117496</td> </tr> <tr> <td>StandardDeviation</td> <td>0.04494942</td> </tr> <tr> <td>Count</td> <td>1000</td> </tr> </tbody> </table>	Stats		mean=0.117 median=0.120 min=0.104 max=1.468 sdtdev=0.045 samples=1000		Minimum	0.104	Maximum	1.468	Median	0.12	Mean	0.117496	StandardDeviation	0.04494942	Count	1000
Stats																	
mean=0.117 median=0.120 min=0.104 max=1.468 sdtdev=0.045 samples=1000																	
Minimum	0.104																
Maximum	1.468																
Median	0.12																
Mean	0.117496																
StandardDeviation	0.04494942																
Count	1000																

جلسه ۲۱

Linear Programming

فاطمه امیدی - ۱۳۹۹/۲/۱۴

در Linear Programming مسئله کلی بصورت تعدادی مجهول و تعدادی معادله محدود کننده مجهولات داده شده است و دنبال بیشترین یا کمترین مقدار ممکن برای یک معادله خطی خاص هستیم.

Agriculture. Diet problem.
Computer science. Compiler register allocation, data mining.
Electrical engineering. VLSI design, optimal clocking.
Energy. Blending petroleum products.
Economics. Equilibrium theory, two-person zero-sum games.
Environment. Water quality management.
Finance. Portfolio optimization.
Logistics. Supply-chain management.
Management. Hotel yield management.
Marketing. Direct mail advertising.
Manufacturing. Production line balancing, cutting stock.
Medicine. Radioactive seed placement in cancer treatment.
Operations research. Airline crew assignment, vehicle routing.
Physics. Ground states of 3-D Ising spin glasses.
Telecommunication. Network design, Internet routing.
Sports. Scheduling ACC basketball, handicapping horse races.

شکل ۱.۲۱: مثال هایی از کاربرد Linear Programming

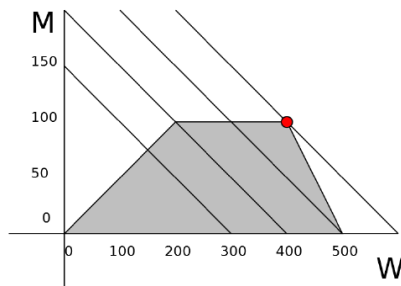
۱.۲۱ a Linear programming example

به عنوان مثال می‌خواهیم در کارخانه‌ای با استفاده از کمترین امکانات ۱۰۰۰۰۰ عدد محصول را تولید کنیم و بیشترین سود را ببریم و در این کارخانه:

- تنها ۱۰۰ عدد ماشین داریم $\Leftrightarrow 100 > M > 0$
 - به تعداد دلخواه کارگر داریم $\Leftrightarrow W > 0$
 - هر ماشین برای کار کردن به دو کارگر نیاز دارد $\Leftrightarrow W > 2M$
 - هر ماشین ۶۰۰ محصول در روز تولید میکند و
 - هر کارگر ۲۰۰ محصول در روز تولید میکند $\Leftrightarrow 100000 > 200(W - 2M) + 600M$
 - هر محصول یک دلار سود دارد و
 - هر کارگر روزانه صد دلار دستمزد میگیرد
- $$(200(W - 2M) + 600M - 100W = 100W + 200M)_{max} \Leftarrow$$

Best: $M = 100, W = 400$ [NB: A corner]

Profit = \$60,000/day.



شکل ۲.۲۱: نمودار معادلات

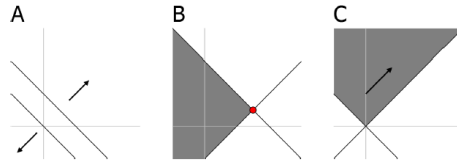
۲.۲۱ وضعیت های جواب

جواب معادلات میتواند به سه صورت باشد:

No Solution .A

One Optimum .B

No Optimum .C



شکل ۳.۲۱: وضعیت های جواب

Row reduction ۳.۲۱

Basic row operations:

.۱ Adding

.۲ Scaling

.۳ Swapping

$$\left[\begin{array}{cc|c} 1 & 1 & 5 \\ 2 & 4 & 12 \end{array} \right] \rightarrow \left[\begin{array}{cc|c} 1 & 1 & 5 \\ 0 & 2 & 2 \end{array} \right]$$

شکل ۴.۲۱: adding

$$\left[\begin{array}{cc|c} 1 & 1 & 5 \\ 0 & 2 & 2 \end{array} \right] \rightarrow \left[\begin{array}{cc|c} 1 & 1 & 5 \\ 0 & 1 & 1 \end{array} \right]$$

شکل ۵.۲۱: scaling

$$\left[\begin{array}{cc|c} 1 & 1 & 5 \\ 0 & 1 & 1 \end{array} \right] \rightarrow \left[\begin{array}{cc|c} 0 & 1 & 1 \\ 1 & 1 & 5 \end{array} \right]$$

شکل ۶.۲۱: swapping

با استفاده از row operations ماتریس معادلات را به شکل ساده ی استاندارد شده تبدیل میکنیم تا

بتوانیم به راحتی ان را حل کنیم.

RowReduce(A)

```
Leftmost non-zero in non-pivot row
Swap row to top of non-pivot rows
Make entry pivot
Rescale to make pivot 1
Subtract row from others to make
other entries in column 0
Repeat until no more non-zero
entries outside of pivot rows
```

شکل ۷.۲۱: pseudocode of row reduction

جلسه ۲۲

برنامه ریزی خطی - دوگان

غزل زمانی نژاد - ۱۳۹۹/۲/۱۶

جزوه جلسه ۱۲۲ مورخ ۱۳۹۹/۲/۱۶ درس طراحی و تحلیل الگوریتم تهیه شده توسط غزل زمانی نژاد. در جهت مستند کردن مطالب درس طراحی و تحلیل الگوریتم

۱.۲۲ چندوجهی‌های محدب *

یک معادله ی خطی نشان دهنده ی یک ابرصفحه [†] است و یک نامساوی نشان دهنده ی یک نیم فضا است. (هر محدودیتی فضا را به دو قسمت تقسیم می کند.) در نتیجه یک دستگاه از نامعادلات نشان دهنده ی یک ناحیه است که با تعدادی ابرصفحه محصور شده است.

یک چندوجهی ناحیه ای است که توسط تعداد متناهی سطح صاف محصور شده است. این سطوح می توانند در ابعاد کمتر [‡] (مثل یال ها) و یا در بعد صفر (گره ها) تقاطع داشته باشند.

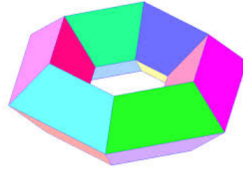
اما هر چندوجهی ممکن نیست، بلکه چندوجهی مورد قبول است که هر صفحه ای از آن را در نظر بگیریم، تمامی نقاط آن در یک سمت واقع شده باشند (آن را چندوجهی محدب می نامیم). در شکل زیر اگر صفحه ای از

Convex Polytopes*

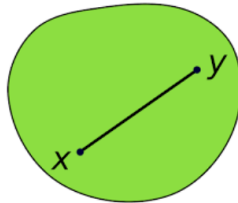
hyperplane[†]

facets[‡]

داخل آن عبور دهیم، تمامی نقاط در یک سمت واقع نشده اند (در نتیجه یک چندوجهی محدب نیست).



یک ناحیه در صورتی محدب است که مانند شکل ۱.۲۲ به ازای هر دو نقطه x و y عضو ناحیه، خطی که این دو نقطه را به یکدیگر متصل می کند تماما در داخل ناحیه قرار گیرد.

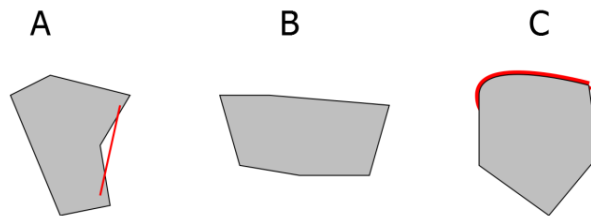


شکل ۱.۲۲: ناحیه محدب

تقاطع چند نیم فضا حتما محدب است و نمی تواند مقعر باشد.

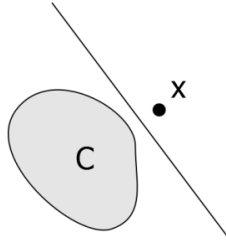
ناحیه ی تشکیل شده توسط دستگاه نامعادلات حتما چندوجهی محدب است.

در شکل ۲.۲۲ ناحیه A محدب نیست چون خطی که دو نقطه دلخواه داخل ناحیه را به هم متصل کرده تماما در داخل ناحیه قرار نگرفته است. ناحیه C نیز محدب نیست چون از قوس تشکیل شده است. تنها ناحیه B محدب است.

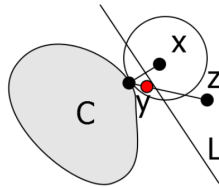


شکل ۲.۲۲: ناحیه محدب

اگر ناحیه C یک ناحیه ی محدب باشد، به ازای هر نقطه x که عضو ناحیه نیست، صفحه ای وجود دارد که تمام نقاط چندوجهی در یک سمت آن و x در سمت دیگر آن قرار می گیرد.

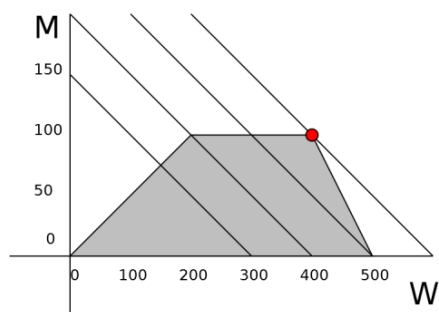


اثبات: مطابق شکل ۳.۲۲ نزدیکترین نقطه به x روی ناحیه را y می نامیم. عمود منصف خط واصل دو نقطه x و y را L می نامیم. فرض می کنیم نقطه z در داخل ناحیه و در سمت اشتباه خط L قرار دارد. در آن صورت yz شامل نقطه ای می شود که به x نزدیکتر است. به تناقض رسیدیم. پس حتما صفحه ی جداکننده ی چندوجهی و نقطه x یافت می شود.



شکل ۳.۲۲: اثبات وجود صفحه ی جداکننده چندوجهی و نقطه ی خارج از آن

یک تابع خطی بیشترین / کمترین مقادیرش را روی گره های چندوجهی اختیار می کند.



شکل ۴.۲۲: بیشترین مقدار تابع هدف روی گره واقع شده است

۲.۲۲ دوگان یک مسئله

[linear_programming] اگر تابع هدف به شکل $v_1 x_1 + v_2 x_2 + \dots + v_n x_n$ باشد و بخواهیم کمترین مقدار آن را محاسبه کنیم، می توانیم از ترکیب کردن ضرایب خطی نامعادلات به جهت ساختن تابع هدف بهره بگیریم.

$$c_1 \cdot [a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \geq b_1]$$

...

$$+ c_m \cdot [a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \geq b_m]$$

$$w_1x_1 + w_2x_2 + \dots + w_nx_n \geq t,$$

$$w_i = \sum c_j a_{ji}, \quad t = \sum c_j b_j.$$

شکل ۵.۲۲: ترکیب کردن نامعادلات با شرط $c_i \geq 0$

برای پیدا کردن مقادیر c_i ها باید به گونه ای عمل کنیم که

$$v_i = \sum_{j=1}^m c_j a_{ji} \quad (۱.۲۲)$$

برقرار باشد و

$$t = \sum_{j=1}^m c_j b_j \quad (۲.۲۲)$$

بیشترین مقدار ممکن را اختیار کند.

تعریف دوگان:

برنامه خطی زیر را داریم:

$$\begin{array}{l} \text{Minimize } v.x \\ \text{Subject to } Ax \geq b \end{array}$$

دوگان آن، برنامه خطی دیگری به شکل زیر است:

$$\begin{array}{l} \text{Maximize } y.b \\ \text{Subject to } y^T A = v, \text{ and } y \geq b \end{array}$$

یک برنامه خطی و دوگان آن همیشه جواب یکسانی دارند. در مواردی که حل برنامه خطی سخت است،

می توانیم دوگان آن را حل کنیم.

مثال: مسئله ی شماره §

اندازه شماره به شکل زیر است:

$$\sum_{e \text{ out of a source}} f_e - \sum_{e \text{ into a source}} f_e.$$

دوگان آن به صورت زیر است:

$$\sum_{e=(v,w)} C_e \max(c_v - c_w, 0).$$

$$\sum_{e=(v,w), v \in C, w \notin C} C_e = |C|.$$

دوگان آن معادل پیدا کردن برش کمینه^۹ است. مثال: مسئله ی رژیم غذایی [diet_problem] فردی می خواهد از شیرینی فروشی مقداری چیزکیک و برانی تهیه کند. قیمت هر برانی ۵۰ سنت و هر چیزکیک ۸۰ سنت است. او می خواهد از مواد مغذی حداقل به میزان ۶ واحد شکلات، ۱۰ واحد شکر و ۸ واحد پنیرخامه ای مصرف کند به گونه ای که کمترین هزینه را داشته باشد. اطلاعات این مسئله به طور خلاصه به صورت زیر است:

	Chocolate	Sugar	Cream Cheese	Cost
Brownie	3	2	2	50
Cheesecake	0	4	5	80
Requirements	6	10	8	

شکل ۶.۲۲: اطلاعات مسئله ی رژیم غذایی

در این مسئله، دستگاه نامعادلات بدین صورت است:

$$\begin{array}{ll} \min_{x_1, x_2} & 50x_1 + 80x_2 \\ & 3x_1 \geq 6, \\ \text{subject to} & 2x_1 + 4x_2 \geq 10, \\ & 2x_1 + 5x_2 \geq 8, \\ & x_1, x_2 \geq 0, \end{array}$$

شکل ۷.۲۲: نامعادلات مسئله ی رژیم غذایی

^۹minimum cut

x_1 نشان دهنده ی مقدار برانی و x_2 نشان دهنده ی مقدار چیزکیک است.

اکنون می توانیم به مسئله از دیدگاه قناد نگاه کنیم (دوگان دستگاه نامعادلات بالا) قناد می خواهد حداقل ۶ واحد شکلات، ۱۰ واحد شکر و ۸ واحد پنیرخامه ای بفروشد تا مواد مغذی مورد نیاز خریدار را تامین کند. هم چنین می خواهد قیمت هر واحد شکلات، شکر و پنیرخامه ای را به گونه ای تعیین کند که بیشترین درآمد را داشته باشد و قیمت برانی کمتر از ۵۰ سنت و قیمت چیزکیک کمتر از ۸۰ سنت شود. دستگاه نامعادلات به صورت زیر است:

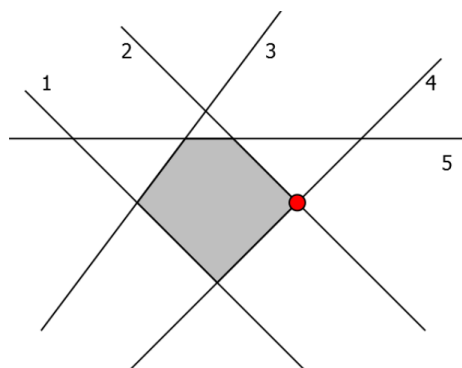
$$\begin{aligned} \max_{u_1, u_2, u_3} \quad & 6u_1 + 10u_2 + 8u_3 \\ \text{subject to} \quad & 3u_1 + 2u_2 + 2u_3 \leq 50, \\ & 4u_2 + 5u_3 \leq 80, \\ & u_1, u_2, u_3 \geq 0. \end{aligned}$$

شکل ۸.۲۲: نامعادلات دوگان مسئله ی رژیم غذایی

u_1 نشان دهنده ی قیمت هر واحد شکلات، u_2 نشان دهنده ی قیمت هر واحد شکر و u_3 نشان دهنده ی قیمت هر واحد پنیرخامه ای است.

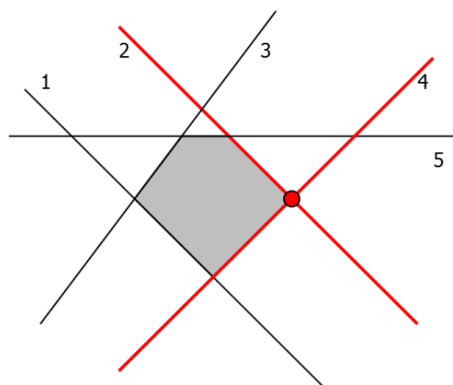
۳.۲۲ Complementary Slackness

اگر مسئله LP به شکل $\text{Minimize } vx \text{ subject to } Ax \geq b$ باشد و دوگان آن به صورت $\text{Maximize } yb \text{ subject to } y^T A = v, y \geq 0$ باشد، آنگاه در جواب ها تنها زمانی $y_i > 0$ است که i^{th} معادله در x ، tight باشد. سوال: اگر نقطه ی مشخص شده در شکل ۹.۲۲ جواب یک مسئله ی LP باشد، کدام معادلات می توانند در مسئله ی دوگان ضریب غیر صفر داشته باشند؟



شکل ۹.۲۲: Complementary Slackness example

پاسخ: مطابق شکل ۱۰.۲۲ جواب خطوط ۲ و ۴ هستند (جواب مسئله ی اصلی روی تقاطع این دو خط واقع شده است).



شکل ۱۰.۲۲: Complementary Slackness answer

Summary

- Every LP has dual LP.
- Solutions to dual bound solutions to primal.
- LP and dual have same answer!
- Complementary slackness.

جلسه ۲۳

Optimization و الگوریتم Simplex

محمدحسین کریمیان - ۱۳۹۹/۲/۲۱

جزوه جلسه ۲۳ ام مورخ ۱۳۹۹/۲/۲۱ درس طراحی و تحلیل الگوریتم تهیه شده توسط محمدحسین کریمیان. در جهت مستند کردن مطالب درس طراحی و تحلیل الگوریتم.

۱.۲۳ اهداف آموزشی جلسه

- تشخیص دادن و تمایز قایل شدن بین مسائل مختلف Optimization .
- استفاده از الگوریتم حل یکی از فرم ها برای حل کردن فرم های دیگر
- الگوریتم Simplex
- الگوریتم Ellipsoid

۲.۲۳ انواع مسائل Optimization

- FullOptimization : در این مسئله یک دستگاه از نامعادلات در اختیار داریم و یک تابع که می خواهیم با صدق کردن تمام نامعادلات مقدار حداقل یا حداکثر این تابع را به دست بیاوریم.

- OptimizationFromStartingPoint : راساس یک دستگاه نامعادلات و یک نقطه یا راس به عنوان نقطه ابتدایی، تابع مورد نظر را Optimize کنیم، یعنی مقدار حداکثر یا حداقل آن را بدست آوریم.
- SoloutionFinding : در این نسخه براساس دستگاه نامعادلات جواب دلخواه که با شرایط صدق کند پیدا می کنیم.
- Satisfiability : فقط به بررسی این که جوابی وجود دارد یا نه می پردازیم.

۳.۲۳ استفاده از حل یک فرم برای حل فرم های دیگر

هر ۴ نوع سوالات Optimization مانند هم هستند و در صورت داشتن حل یکی از آن ها می توان بقیه موارد را هم حل کرد. مثلا برای حل کردن FullOptimization از طریق نقطه ی شروع به این صورت عمل می کنیم که نامعادلات را دونه دونه اضافه می کنیم یعنی در هر مرحله نامعادله جدید را به صورت تابع نوشته و با توجه به شرایط داده شده از نامعادله های دیگر برای آن حداقل یا حداکثر پیدا می کنیم ومحدودیت ها را هر دفعه کم می کنیم و به نقطه جدیدی در هر مسئله می رسمیم تا وقتی که دیگر معادله ای حل نشده باقی نماند آنگاه نقطه ی نهایی را در جهت یالها(همان نامعدله های)مربوط به آن ادامه می دهیم و هر کدام که مدار Optimal بود جواب مسئله است.

همچنین اگر یک جواب را به دست آوریم می توانیم بهترین جواب را به دست آوریم یعنی با استفاده از SoloutionFinding می توانیم بهترین جواب را با به دست آوردن یک جواب از نامعادله و جواب متناظر dual آن پیدا کنیم.

Want to minimize $x \cdot v$ subject to $Ax \geq b$.
Instead find solution to:

$$\begin{aligned} Ax &\geq b \\ y &\geq 0 \\ y^T A &= v \\ x \cdot v &= y \cdot b. \end{aligned}$$

Will give optimal solution to original problem.

شکل ۱۰.۲۳: یک مثال برای به دست آوردن بهترین جواب به کمک dual

با استفاده از satisfiability و فرض ناحیه که جواب در آنجاست، هر معادله ای که با آن ناحیه اشتراک داشته باشد را بررسی می کنیم و جواب ها را به دست می آوریم که در نهایت می شود بهترین جواب را پیدا کرد.

با در نظر گرفتن FullOptimization می توان سه نوع دیگر هم حل کرد:

- OptimizationFromStartingPoint : نقطه اولیه داده شده را نادیده می گیریم و حداقل یا حداکثر را بدست می آوریم.
- SoloutionFinding : در FullOptimization مقدار حداکثر یا حداقل به دست می آید پس یک جواب پیدا کرده ایم.
- Satisfiability : بررسی این که جوابی به دست می آید یا نه در FullOptimization پاسخ داده می شود و اگر جوابی داشته باشد آنگاه Satisfiable است.

۴.۲۳ الگوریتم Simplex

از یک نقطه شروع می‌کنیم و یکی یکی یال‌ها را رد می‌کنیم و در هر مرحله به بهترین جواب اون مرحله می‌رسیم. با داشتن m نامعادله و n متغیر، به تعداد متغیرها n معادله انتخاب می‌کنیم و یک نقطه مشترک پیدا می‌کنیم. سپس یکی از معادله‌هایی که آن نقطه در آن صدق می‌کند را انتخاب می‌کنیم و آن را relax می‌کنیم تا یک خط به دست بیائیم که نقطه در آن قرار دارد. بعد از این کار از نقطه اولیه روی خط حرکت می‌کنیم و هر جا از محدوده نامعادله دیگری خارج شدیم آن نقطه را بزای نقطه شروع جدید در نظر می‌گیریم و دوباره این عملیات را با نامعادلات باقی مانده انجام می‌دهیم.

Pseudocode

Simplex

```

Start at vertex  $p$ 
repeat
  for each equation through  $p$ 
    relax equation to get edge
    if edge improves objective:
      replace  $p$  by other end
    break
if no improvement: return  $p$ 

```

شکل ۲.۲۳: کد الگوریتم Simplex

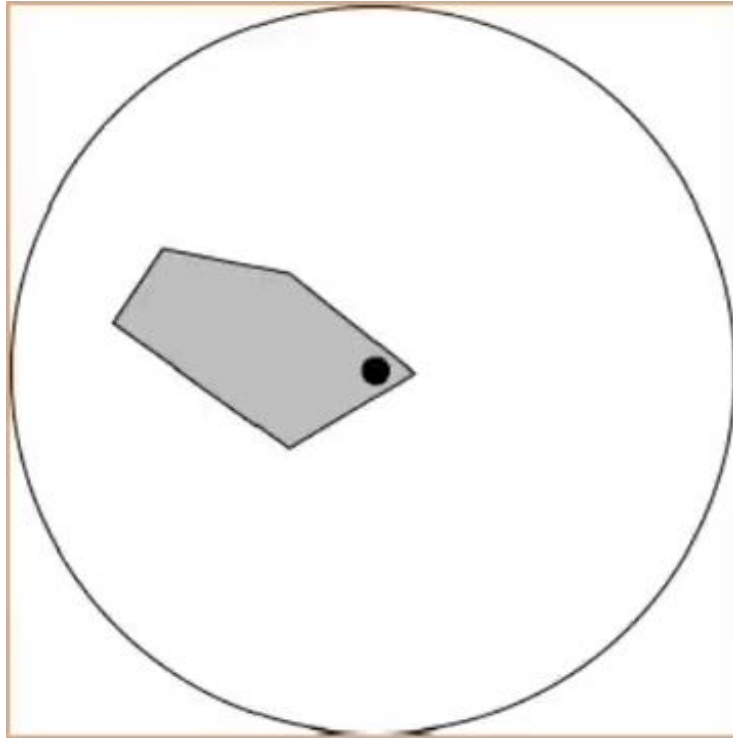
OtherEndOfEdge

Vertex p defined by n equations
 Relax one, write general solution
 as $p + tw$ (Gaussian elimination)
 Relaxed inequality requires $t \geq 0$
 For each other inequality in
 system:
 Largest t so $p + tw$ satisfies
 Let t_0 be smallest such t
 return $p + t_0w$.

شکل ۳.۲۳: کد به دست آوردن نقطه شروع جدید

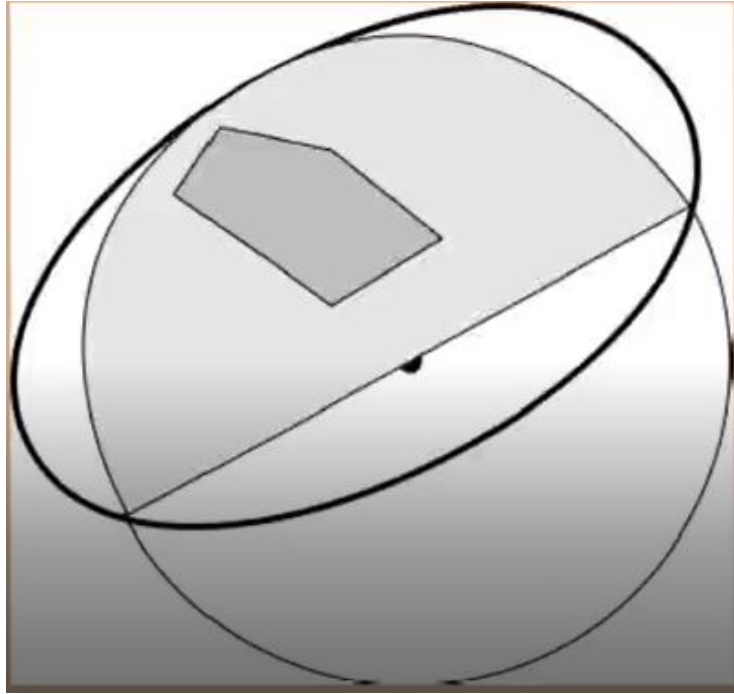
۵.۲۳ الگوریتم Ellipsoid

این الگوریتم نسخه ی Satisfiability را حل می کند و به این دلیل به معادلات نیازی ندارد. در این الگوریتم، همه ی نا معادلات را تا حدی relax می کنیم. اگر جوابی وجود داشته باشد آن جواب را می توان یک چند وجهی در فضا فرض کرد که مساحت آن مثبت می باشد. سپس یک دایره ی بزرگ که این چند وجهی را شامل شود در نظر می گیریم. اگر وسط این دایره ی فرضی درون چندوجهی قرار داشت، دستگاه Satisfiable می باشد



شکل ۴.۲۳:

و معادله دارای جواب است. در غیر این صورت نصفه ای از دایره که شکل ما در آن است را ذریک بیضی جدید که حجم کمتری دارد قرار می دهیم اگر وسط بیضی درون چندوجهی بود Satisfiable است



شکل ۵.۲۳:

و در غیراین صورت باز همین عملیات را ادامه می دهیم و اگر حجم بیضی از مقلدر مفروضی کمتر شد، Satisfiable نمی باشد و جواب ندارد.

جلسه ۲۴

مسائل NP-complete

مجتبی نافذ - ۱۳۹۹/۰۲/۲۳

در این جلسه می‌خواهیم مسائل را بر اساس مرتبه‌ی زمانی دسته‌بندی کنیم و بیشتر روی مسائل ای از مرتبه زمانی توانی بحث کنیم.
معمولا ما مسائلی را که مرتبه زمانی کمتر از 10^9 عمل دارند را میتوانیم حل کنیم.

Polynomial vs Exponential

running time:	n	n^2	n^3	2^n
less than 10^9 :	10^9	$10^{4.5}$	10^3	29

شکل ۱.۲۴: مرتبه زمانی بر اساس مقدار n (تعداد اعمال) قابل حل

معمولا الگوریتم بهینه در فضای حالت n^{n-2} ، 2^n ، $n!$ در جستجوی جواب است به طور مثال n^{n-2} درخت پوشای کمینه در یک گراف کامل وجود دارد هزاران مسئله مهم وجود دارد که تاکنون الگوریتم بهینه ای برای آن ثبت نشده است که حل یکی از آن مسایل منجر به حل همه ی آن ها خواهد شد و حل یکی از آن ها جایزه یک میلیون دلاری دارد.

۱.۲۴ مسائل جستجو search problems

مسائلی که به وسیله ی الگوریتم C که دو ورودی l به عنوان یک نمونه مسئله و S به عنوان یک جواب از مسئله از ما میگیرد در زمان چند جمله ای چک میکند که آیا S در l صدق میکند یا خیر اگر صدق کند:

$$C(S,l) = \text{true}$$

نمونه ای از مسئله جستجو مسئله SAT problem است که شرح خواهیم داد.

۲.۲۴ صدق پذیری دودویی (Satisfiability) SAT problem

ورودی: یک فرمول CNF
خروجی: یک مجموعه جواب بولین به تمام متغیرهای CNF برای true کردن تمامی clauses (اگر جوابی وجود داشته باشد)

۳.۲۴ فرم نرمال ترکیب عطفی Conjunctive Normal Form

گزاره ای که به فرم ضرب حاصل جمعها نوشته شود را، فرم نرمال CNF گوئیم. به عنوان مثال عبارت زیر یک CNF می باشد:

$$(A \vee B) \wedge (\neg B \vee C \vee \neg D) \wedge (D \vee \neg E)$$

مثالی از مسئله SAT:

مثال ۱:

$$\text{فرمول سی ان اف: } (x \vee \neg y) \wedge (\neg x \vee \neg y) \wedge (x \vee y)$$

مجموعه ی satisfiable : $x = 1$ ، $y = 0$.

مثال ۲:

فرمول سی ان اف: $(x \vee y \vee z) \wedge (x \vee \neg y) \wedge (y \vee \neg z)$ مجموعه ی satisfiable : $z = 0, y = 0, x = 1$ or $z = 1, y = 1, x = 1$

مثال ۳:

فرمول سی ان اف: $(x \vee y \vee z) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg x) \wedge (\neg z \vee \neg y \vee \neg z)$

مجموعه ی satisfiable : is unsatisfiable

۴.۲۴ دسته بندی مسائل

مسائل decision: مسائلی که جواب بله و خیر دارند این پاسخ های بله و خیر در جواب مقادیر ورودی ای هستند که ادعا بر آن است که این ورودی ها پاسخ مسئله هستند به عبارتی درستی جواب را چک می کنند.

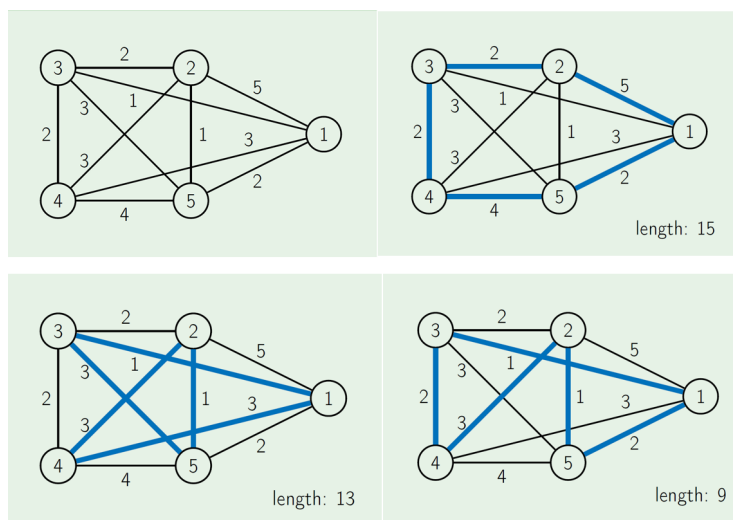
مسائل Optimization: مسائل که بهترین راه حل را در بین تمام راه حل های feasible پیدا می کند. راه حل استاندارد وجود دارد که مسائل optimization را به decision تبدیل کرد. که مورد بحث ما نیست.

۵.۲۴ نمونه هایی از مسائل با دو ورژن ساده و سخت

مسئله: فروشنده ی دوره گرد traveling salesman problem

ورودی: فاصله ی دو به دو شهر ها به هم و مقدار بودجه a برای طول مسیر حرکتی

خروجی: یک دور یا حلقه که هر شهر را دقیقاً یکبار طی و مجموع طول مسیر حداکثر b باشد.



شکل ۲۰۲۴: مثال

این مسئله یک مسئله جستجو است. یک مجموعه راس می‌گیریم باید چک شود آیا همه یال‌ها فقط یکبار با حداکثر طول b دیده شده‌اند یا خیر.

مسئله TSP معمولاً به صورت یک مسئله optimization بیان میشود اما ما به صورت decision بیان میکنیم برای این مسئله تاکنون راه حل چند جمله پیدا نشده و راه $O(2^n n^2)$ dynamic programming میانگین زمانی را به دنبال دارد.

چک کردن تمام حالت‌ها از $O(n!)$ است مسائل مشابهی مانند درخت پوشای کمینه در زمان چند جمله‌ای قابل حل هستند.

چند نمونه را مقایسه میکنیم:

Comparing to MST

MST	TSP
Decision version: given n cities, connect them by $(n - 1)$ roads of minimal total length	Decision version: given n cities, connect them in a path by $(n - 1)$ roads of minimal total length
Can be solved efficiently	No polynomial algorithm known!

شکل ۳.۲۴: مسئله فروشنده ی دوره گرد

Eulerian cycle	Hamiltonian cycle
Find a cycle visiting each edge exactly once	Find a cycle visiting each vertex exactly once
Can be solved efficiently	No polynomial algorithm known!

شکل ۴.۲۴: مسئله دور همیلتونی

Shortest path	Longest path
Find a simple path from s to t of total length at most b	Find a simple path from s to t of total length at least b
Can be solved efficiently	No polynomial algorithm known!

شکل ۵.۲۴: بلند ترین مسیر

LP (decision version)	ILP
Find a real solution of a system of linear inequalities	Find an integer solution of a system of linear inequalities
Can be solved efficiently	No polynomial algorithm known!

شکل ۶.۲۴: برنامه ریزی خطی برای عدد صحیح

Independent set in a tree	Independent set in a graph
Find an independent set of size at least b in a given tree	Find an independent set of size at least b in a given graph
Can be solved efficiently	No polynomial algorithm known!

ب

شکل ۷.۲۴: مسئله مجموعه مستقل

۶.۲۴ مسائل P, NP

مسائل کلاس P: تمامی مسائل جستجویی که (search problem) در زمان چند جمله ای قابل حل باشند.

مسائل کلاس NP: تمامی مسائل جستجویی (search problem) عضو NP هستند

مسائلی که به وسیله ی الگوریتم C که دو ورودی 1 به عنوان یک نمونه مسئله و S به عنوان یک جواب از مسئله از ما میگیرد در زمان چند جمله ای چک میکند که آیا S در 1 صدق میکند یا خیر اگر صدق کند:

$$C(S,1) = \text{true}$$

Class P	Class NP
Problems whose solution can be found efficiently	Problems whose solution can be verified efficiently
<ul style="list-style-type: none"> ■ MST ■ Shortest path ■ LP ■ IS on trees 	<ul style="list-style-type: none"> ■ TSP ■ Longest path ■ ILP ■ IS on graphs

شکل ۸.۲۴: P vs NP

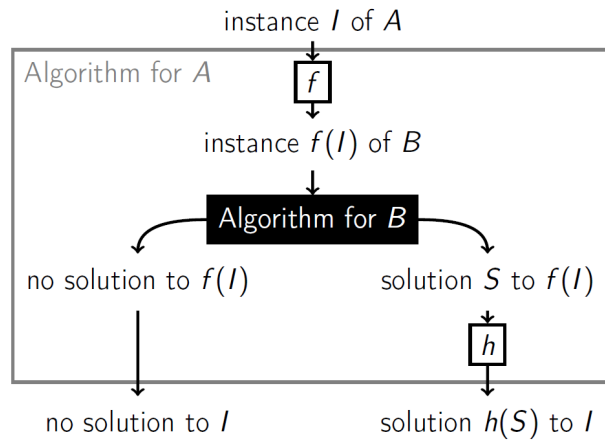
مسئله باز بزرگ علم کامپیوتر با جایزه یک میلیون دلاری:

ایا مسائل P با مسائل NP برابر هستند

اگر برابر باشند مسائل search problem در زمان چند جمله ای قابل حل خواهند بود وگرنه مسئله NP وجود دارد که در زمان چند جمله ای قابل حل نیست

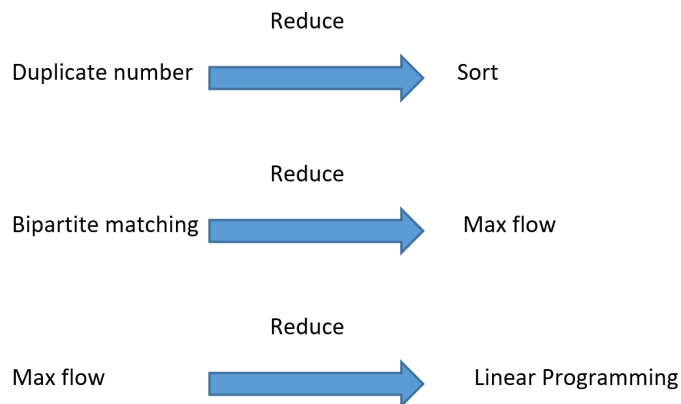
۷.۲۴ کاهش Reductions

مسئله A search problem به مسئله B search problem قابل کاهش است اگر الگوریتم زمانی چند جمله ای برای تبدیل نمونه ای مانند l از مسئله A به یک نمونه f(1) از مسئله B وجود داشته باشد و همچنین جواب نمونه S ی برای f(1) در زمان چند جمله ای قابل تبدیل به h(S) برای A باشد. اگر f(1) ای وجود نداشته باشد راه حلی برای l وجود ندارد

Reduction: $A \rightarrow B$ 

شکل ۹.۲۴: کاهش Reduction

- مثال ۱: برای حل مسئله پیدا کردن عدد تکراری در آرایه میتوان آن را به مسئله مرتب سازی کاهش داد و بعد از مرتب سازی اعداد متوالی را چک کنیم.
- مثال ۲: در اسلاید های قبل ما برای حل مسئله ی bipartitmatching آن را به maxflow کاهش دادیم
- مثال ۳: در اسلاید های قبل ما مسئله ی maxflow را به Linear Programming کاهش دادیم



شکل ۱۰.۲۴: کاهش

نکته: کاهش مسئله به مسئله پیچده تر امکان پذیر هست اما راه گشا نیست مثلا تبدیل مسئله میانگین آرایه به مسئله فروشنده ی دوره گرد حتی اگر امکان پذیر هم بود فقط حل مسئله را پیچیده تر می کند.

۸.۲۴ مسائل NP-hard, NP-complete

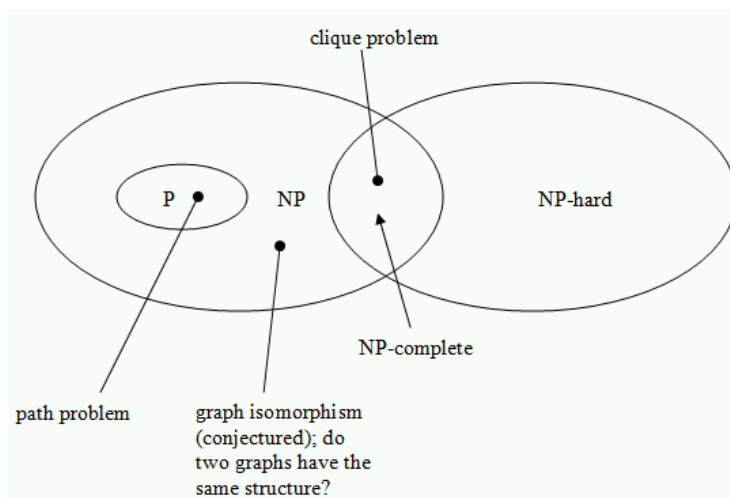
مسائل np-complete: مسائل NP ای که همه ی مسائل NP را بتوان به آن کاهش داد را NPC می نامیم

برای مثال مسئله صدق پذیری دودویی (SAT) نمونه ای از مسائل np-complete است

مسائل NP-hard مسئله H است زمانی که هر مسئله ی L در NP را بتوان به در زمان چند جمله ای به H کاهش داد و جواب H را زمان چند جمله ای به جواب L تبدیل کرد. اما تا به حال راه حلی بهینه ای برای حل خود H پیدا نشده است

برای مثال فروشنده ی دوره گرد (TSP) یک مسئله NP-hard میباشد

نکته: تفاوت این دو در این است که np-complete باید np باشد یعنی در زمان چندجمله ای درستی جواب را چک کند اما np-hard چه np باشد و چه نباشد فقط مسائل np به ان قابل کاهش باشند .



شکل ۱۱.۲۴: جمع بندی

[coursera] [wikipedia۲] [wikipedia۱] [CLRS]

جلسه ۲۶

حل کردن مسائل NP-complete (حالت های خاص)

هادی شیخی - ۱۳۹۹/۲/۳۰

جزوه جلسه ۲۶م مورخ ۱۳۹۹/۲/۳۰ درس طراحی و تحلیل الگوریتم تهیه شده توسط هادی شیخی. در جهت مستند کردن مطالب درس طراحی و تحلیل الگوریتم، بر آن شدیم که از دانشجویان جهت مکتوب کردن مطالب کمک بگیریم. هر دانشجو می تواند برای مکتوب کردن یک جلسه داوطلب شده و با توجه به کیفیت جزوه از لحاظ کامل بودن مطالب، کیفیت نوشتار و استفاده از اشکال و منابع کمک آموزشی، حداکثر یک نمره مثبت از بیست نمره دریافت کند. خواهشمند است نام و نام خانوادگی خود، عنوان درس، شماره و تاریخ جلسه در ابتدای این فایل را با دقت پر کنید. مطالبی که در ادامه آمده فقط جنبه راهنمایی شیوه استفاده از لاتک می باشد. خواهشمند است این پاراگراف و مطالب بعدی را از نسخه جزوه ای که تحویل می دهید، حذف کنید.

۱.۲۶ مسائل NP-Complete

همانطور که در جلسات پیش توضیح داده شد مسائل NP معادل یک Search Problem هستند. حال برای این مسائل یا راه حل چندجمله ای وجود دارد یا اینکه تا به حال موفق به یافتن راه حل با پیچیدگی چندجمله ای

برای آن‌ها نشده‌ایم. زیر مجموعه‌ای از مسائل NP هستند که همه مسائل موجود در NP را میتوان به آن‌ها کاهش داد. این مجموعه را مسائل NP-Complete می‌نامیم.

۲.۲۶ حل کردن مسائل NP-Complete

در حال حاضر الگوریتم های ارائه شده برای حل مسائل NP-Complete نیاز به پیچیدگی محاسباتی Super Polynomial دارد و همچنان اثبات نشده‌است که راهی با پیچیدگی کمتری وجود دارد یا خیر. به طور کلی برای مسائل محاسباتی از روش هایی می‌توان استفاده کرد که منجر به ایجاد الگوریتم‌هایی با پیچیدگی محاسباتی کمتر می‌شود [Solve-NP].

- Approximation : به جای پیدا کردن جواب بهینه به دنبال جواب‌هایی باشیم که به جواب بهینه نزدیک هستند.
- Randomization : استفاده از انتخاب های تصادفی برای کاهش پیچیدگی محاسباتی میانگین با اینکه الگوریتم در برخی موارد به درستی جواب نمیدهد.
- Parameterization : اگر برخی پارامتر های ورودی الگوریتم ثابت باشند، پیچیدگی محاسباتی کاهش می‌یابد.
- Heuristic : الگوریتم برای بسیاری از حالات جواب می‌دهد ولی اثبات دقیقی ندارد که برای همه حالات جوابگو هست یا خیر.

هنگام حل کردن مسائل به جای اینکه ثابت کنیم مسأله‌ای الگوریتمی بهینه ندارد، می‌توانیم ثابت کنیم مسأله یکی از Search Problem های سخت است. که در این صورت نشان داده‌ایم که مسأله ما NP-Complete است.

۳.۲۶ ۲-SAT یک حالت خاص [۲-sat]

از آنجا که در جلسات پیش بحث شده می‌دانیم SAT یک مسأله NP-Complete است و در حال حاضر الگوریتمی با پیچیدگی چند جمله‌ای برای آن ارائه نشده است. با این حال ۲-SAT یک حالت خاص از SAT است که میتوان آن را با پیچیدگی چند جمله‌ای حل کرد که به بررسی آن می‌پردازیم.

۱.۳.۲۶ ساخت Implication Graph

مرحله اول در حل ۲-SAT تبدیل فرم نرمال به گرافی است که وجود هر یال جهت دار از رأس x به y به معنی $x \rightarrow y$ است. بر اساس جبر بولی می‌توان نوشت:

$$X \rightarrow Y ::= (\bar{X} + Y)$$

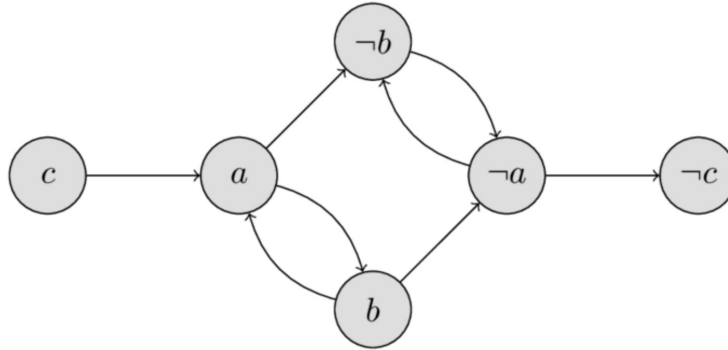
طبق آنچه گفته شد، می‌توان هر جمله از SAT را به صورت دو Implication نوشت یعنی:

$$(X + Y) ::= \bar{X} \rightarrow Y, \bar{Y} \rightarrow X$$

و با توجه به تعریف Implication Graph، اگر گرافی داشته باشیم که برای هر متغیر X تعریف شده در SAT دو رأس X و \bar{X} را شامل شود برای هر جمله $(X+Y)$ باید بالای \bar{X} به Y و یالی از \bar{Y} به X باشد. حال با انجام مراحل فوق برای هر جمله از ۲-SAT گراف مربوط به آن را می‌سازیم. برای مثال [Implication-graph] ۲-SAT زیر را در نظر بگیرید.

$$(a + \bar{b})(\bar{a} + b)(\bar{a} + \bar{b})(a + \bar{c})$$

گراف مربوط به این مسأله به صورت زیر است.



شکل ۱۰۲۶: گراف مربوط به ۲-SAT

۲.۳.۲۶ پیدا کردن مولفه‌های قویاً همبند (Strongly Connected Components)

قبل از اینکه به پیدا کردن مولفه‌ها بپردازیم، لم زیر را تعریف می‌کنیم.
 لم: اگر SAT جواب ۱ داشته‌باشد و مسیری از l_1 به l_2 وجود داشته‌باشد نمی‌تواند $l_1 = 1$ و $l_2 = 0$ باشد.
 اثبات:

$$l_1 \rightarrow l_3 \rightarrow \dots \rightarrow l_2$$

حال اگر $l_1 = 1$ و $l_2 = 0$ باشد آنگاه وجود دارد l_i به طوری که

$$l_i \rightarrow l_j$$

و $l_i = 1$ ، $l_j = 0$. وجود یال بین l_i و l_j بدین معنی است که جمله $(l_j + \bar{l}_i)$ در SAT موجود است و با توجه به مقادیر داده شده این جمله برابر ۰ خواهد شد که جواب SAT را هم ۰ می‌کند، و این تناقض است با فرض سوال. پس لم درست است.
 حال با توجه به لم فوق و اینکه همه رأس‌ها در یک مولفه‌های قویاً همبند به هم دیگر مسیری دارند میتوان نتیجه‌گیری کرد:

- همه متغیرهایی که در یک مولفه قویاً همبند هستند باید مقدار یکسانی داشته‌باشند.
- برای هر متغیر X اگر X و \bar{X} در یک مولفه قویاً همبند باشند جواب unsatisfiable است.

۳.۳.۲۶ مقدار دهی متغیرها

پس از پیدا کردن مولفه‌های قویاً همبند و بررسی شرایط پذیرش (Satisfiability)، اگر مورد پذیرش واقع شود برای بدست آوردن مقادیر متغیرها به صورت زیر عمل میکنیم:

۱. مرتب کردن مولفه‌ها بر اساس Topological Sort
 ۲. شروع از آخر و اگر هیچ یک از متغیرها مقدار دهی نشده باشد، به همه متغیرها مقدار ۱ را می‌دهیم و مقدار متغیر متضاد هر کدام را ۰ قرار می‌دهیم.
- پس از انجام مراحل فوق مقدار هر یک از متغیرها معلوم است.

۴.۳.۲۶ اثبات درستی

هرگاه متغیری ۱ شود آنگاه همه متغیرهایی که از آن قابل دسترسی هستند هم برابر ۱ می‌شود که بدان معنی است که همه Implication ها satisfy می‌شوند.

به همین ترتیب اگر متغیری ۰ باشد آنگاه همه متغیرهایی که قابل دسترسی از آن هستند نیز برابر ۰ می‌شوند.

[skew-symmetric] (Skew-Symmetry)

جلسه ۲۸

حالت‌های مواجهه با مسائل NP-Complete

ملیکا احمدی رنجبر - ۱۳۹۹/۳/۶

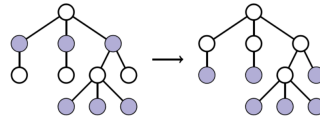
جزوه جلسه ۲۸ مورخ ۱۳۹۹/۳/۶ درس طراحی و تحلیل الگوریتم تهیه شده توسط ملیکا احمدی رنجبر. در جهت مستند کردن مطالب درس طراحی و تحلیل الگوریتم

۱.۲۸ حالت خاص Independent Set

در ادامه بحث حالت‌های خاص برای حل مسائل NP-Complete مسئله‌ی Independent Set مطرح می‌شود. این مسئله در حالت خاص برای درخت در زمان چندجمله‌ای قابل حل است اما در حالت کلی برای گراف به این شکل نیست.

سوالی که برای این نمونه حالت خاص مسائل NP-Complete می‌توان مطرح کرد مسئله‌ی Planning A Company Party است، به این گونه که باید بیشترین تعداد افراد در این مهمانی حضور داشته باشند با این شرط که هیچ مهمانی همراه با رئیسیش نیاید. این سوال را می‌توان توسط ورودی درخت با Independent Set در زمان چندجمله‌ای حل نمود. Safe Move برای حل مطرح می‌کند که به ازای هر Node می‌توان

برگ‌هایش را انتخاب نمود.



شکل ۱.۲۸: Move Safe

* ۲۹

```

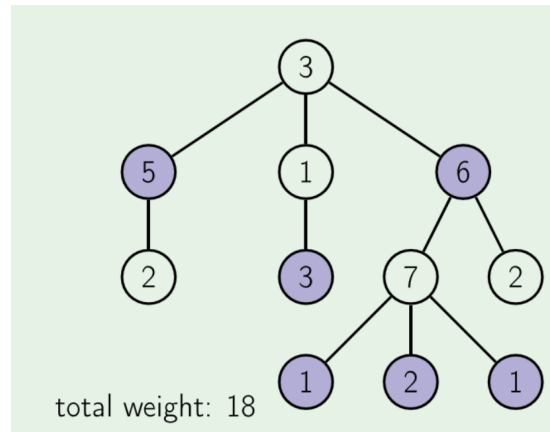
O(|T|);
initialization;
while T is not empty do
    take all the leaves to the solution ;
    remove them and their parents from T;
end
return the constructed solution;

```

Algorithm 26: PartyGreedy

همین مسئله می‌تواند به شکل Maximum weighted independent set in trees مطرح شود که در این صورت باید بیشترین وزن Node ها را برگردانیم.

pseudocode*



شکل ۲۰۲۸: نمونه‌ای از محاسبات این سوال روی درخت

کد این قسمت را می‌توانید در قسمت زیر مشاهده کنید.

$$\max \left\{ w(v) + \sum_{\substack{\text{grandchildren} \\ w \text{ of } v}} D(w), \sum_{\substack{\text{children} \\ w \text{ of } v}} D(w) \right.$$

Function FunParty(v)

```

if  $D(v) = \infty$ :
  if  $v$  has no children:
     $D(v) \leftarrow w(v)$ 
  else:
     $m_1 \leftarrow w(v)$ 
    for all children  $u$  of  $v$ :
      for all children  $w$  of  $u$ :
         $m_1 \leftarrow m_1 + \text{FunParty}(w)$ 
     $m_0 \leftarrow 0$ 
    for all children  $u$  of  $v$ :
       $m_0 \leftarrow m_0 + \text{FunParty}(u)$ 
     $D(v) \leftarrow \max(m_1, m_0)$ 
return  $D(v)$ 

```

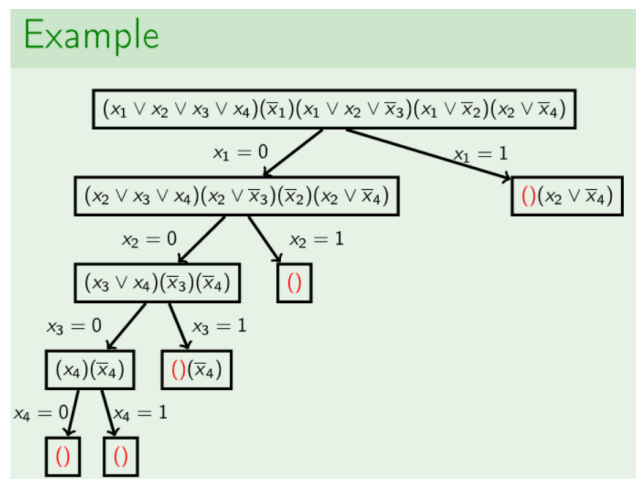
شکل ۳۰۲۸: Main Code

3-Satisfiability ۲.۲۸

Backtracking ۱.۲.۲۸

یکی از روش‌های حل به این شکل است. در این روش برای مثال اگر 3-sat داشته باشیم با توجه به شرایط مثال داده شده به دلخواه یکی از مقادیر ۰ یا ۱ را به یکی از Variable ها می‌دهیم و نسبت به آن مقدار داده شده به ترتیب عملیات‌ها و پرانتزهای ساده‌تری را خواهیم داشت و به همین ترتیب به تمام Variable ها مقدار می‌دهیم. برخی قابل قبول نیستند برای همین یک شاخه به عقب بازمی‌گردیم و مقدار مخالف آن مقداری که قابل قبول نبود می‌دهیم و جلو می‌رویم. هرگاه به جایی رسیدیم که مقادیر قابل قبول نبودند تنها کافی است به شاخه‌های قبل برگردیم و مقادیر مخالف مقدار قبل را امتحان کنیم و تا زمانی ادامه می‌دهیم (چه به سمت پایین چه بازگشت در صورت قابل قبول نبودن) که به جواب برسیم (منظور از بالا و پایین رفتن روی درختی است که ایجاد می‌شود).

Goal : Avoid going through all 2 assignments



شکل ۴.۲۸: Sample Of The Tree Which is Made

سرعت این روش به دلیل آن است که زمانی که متوجه می‌شویم مقداری درست نیست و به بالا برمی‌گردیم یعنی دیگر یک شاخه را ادامه نمی‌دهیم و آن را قطع کردیم بنابراین همه حالت‌ها نیازی به بررسی ندارند. و در هر قسمت از مقدار یک متغیر مطمئن می‌شویم.

Local Search ۳.۲۸

تعريفات ۱.۳.۲۸

Hamming Distance: تعداد بیت‌هایی که دو عدد باینری در آن تفاوت دارند.
 Hamming Ball: با مرکز a و شعاع r مجموعه اعداد باینری است که حداکثر به اندازه r تفاوت دارند.
 منظور از اعداد باینری تعدادی متغیر است که مقادیر 0 و 1 می‌پذیرند. در واقع منظور Assignments است.

Example

- $\mathcal{H}(1011, 0) = \{1011\}$
- $\mathcal{H}(1011, 1) = \{1011, 0011, 1111, 1001, 1010\}$
- $\mathcal{H}(1011, 2) = \{1011, 0011, 1111, 1001, 1010, 0111, 0001, 0010, 1101, 1110, 1000\}$

شکل ۵.۲۸: Hamming Ball And Hamming Distance

- If α satisfies F , return α
- Otherwise, take an unsatisfied clause — say, $(x_i \vee \bar{x}_j \vee x_k)$
- α assigns $x_i = 0, x_j = 1, x_k = 0$
- Let $\alpha^i, \alpha^j, \alpha^k$ be assignments resulting from α by flipping the i -th, j -th, k -th bit, respectively
- **Crucial observation:** at least one of them is closer to β than α
- Hence there are at most 3^r recursive calls □

شکل ۶.۲۸: Proof

اگر برای متغیرهایی تعداد 0 ها بیشتر باشد بنابراین Hamming Ball را در فاصله $N/2$ برای N

متغیر با مقدار ۰ پیدا می‌کنیم و می‌دانیم پاسخ در آن است و اگر تعداد ۱‌ها بیشتر بود تمام متغیرها را ۱ در نظر می‌گیریم و همان کار را انجام می‌دهیم.

CheckBall(F, α, r)

```

if  $\alpha$  satisfies  $F$ :
    return  $\alpha$ 
if  $r = 0$ :
    return "not found"
 $x_i, x_j, x_k \leftarrow$  variables of unsatisfied clause
 $\alpha^i, \alpha^j, \alpha^k \leftarrow \alpha$  with bits  $i, j, k$  flipped
CheckBall( $F, \alpha^i, r - 1$ )
CheckBall( $F, \alpha^j, r - 1$ )
CheckBall( $F, \alpha^k, r - 1$ )
if a satisfying assignment is found:
    return it
else:
    return "not found"

```

شکل ۷.۲۸: Hamming Ball And Hamming Distance

۴.۲۸ Traveling Salesman Problem (TSP)

گراف کامل وزن‌داری است که ما در آن به دنبال دوری هستیم که از یک Node شروع کند و از هر کدام از Node ها یک بار عبور کند و به جای اول بازگردد اما حداکثر وزن طی شده مقدار خاصی باشد.

Dynamic Programming

الگوریتم بدیهی: تمام حالت‌ها را چک کنیم که برابر است با Cycles $(n - 1)!$. اما می‌توان از Dynamic Programming استفاده کرد که دارای زمان $O(n^2 \cdot 2^n)$ است.

در واقع ما مسئله را به زیر مسئله‌هایی تقسیم می‌کنیم که با هم Overlapping دارند.

TSP(G)

```

C({1},1) ← 0
for s from 2 to n:
  for all  $1 \in S \subseteq \{1, \dots, n\}$  of size s:
    C(S,1) ← +∞
    for all  $i \in S, i \neq 1$ :
      for all  $j \in S, j \neq i$ :
        C(S,i) ← min{C(S,i), C(S - {i},j) + dji}
return mini{C({1, ..., n}, i) + di,1}

```

شکل ۸.۲۸: Hamming Ball And Hamming Distance

ترتیب چک کردن تمام زیرمجموعه‌ها (2^n) یک نگاهت به صورت زیر دارد.

$k \leftrightarrow i$: i-th bit of k is 1

Branch-and-bound

این روش در واقع همان روش Backtracking است اما Optimize شده است . یعنی در مرحله برای این مسئله جدید چک می‌شود که آیا مسیر بهتری وجود دارد اگر احتمالش کم بود چک نمی‌شود. در واقع مقداری را برای چک کردن ذخیره می‌کند اگر بیشتر می‌شد آن شاخه را ادامه نمی‌دهد و مسیر جدید برای چک کردن انتخاب می‌کند. کمترین مقدار در این مسئله مد نظر است پس کوچکترین مقداری که در الگوریتم می‌یابد را ذخیره می‌کند اگر هر مسیری از آن بیشتر شد ادامه آن را نمی‌رود و مسیر جدید را شروع می‌کند