



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

۵

Paths in Graphs ۲

نگار زین العابدین - ۱۳۹۸/۱۱/۲۰

۵

دیدگلی ۱.۲

۶

Shortest path ۲.۲

۶

دور منفی چیست؟ ۳.۲

۷

Dijkstra / BFS ۴.۲

۸

Bidirectional dijkstr / Bidirectional BFS ۵.۲

۱۰

Dijkstra And Bellman-Ford Algorithms ۴

پریسا علائی - ۱۳۹۸/۱۱/۲۷

۱۰

نکته: ۱.۴

۱۰

Dijkstra's Algorithm ۲.۴

۱۱

Order of Dijkstra Algorithm ۳.۴

۱۱

: Dijkstra Algorithm ۴.۴

۱۲

: Dijkstra Algorithm ۵.۴

۱۲

: مثال ۶.۴

۱۵

: Dijkstra Algorithm ۷.۴

۱۶

ایثات درست بودن الگوریتم ۸.۴

۱۷

Bellman–Ford algorithm ۹.۴

۱۷

Order of Bellman-Ford Algorithm ۱۰.۴

۱۷

: Bellman-Ford Algorithm ۱۱.۴

۱۸	۱۲.۴	معایب Bellman-Ford Algorithm
۱۸	۱۳.۴	مثال :
۲۱	۱۴.۴	شبکه کد :
۲۱	۱۵.۴	اثبات درست بودن الگوریتم Bellman-Ford
۲۲	۱۶.۴	دور منفی در : Bellman-Ford
۲۲	۱۷.۴	اثبات درست بودن الگوریتم دور منفی در Bellman-Ford
۲۳	۱۸.۴	Infinite Arbitrage :
۲۳	۱۹.۴	اثبات درست بودن الگوریتم Infinite Arbitrage
۲۴	۲۰.۴	سایتهاي ديجير برای مراجعه :

۲۵	۱.۵	Dijkstra شبکه
۲۶ prev	۲.۵	شبکه پیدا کردن کوتاه ترین راه بین دو راس به کمک آرایه
۲۶	۳.۵	اثبات درستی الگوریتم Dijkstra
۲۷	۴.۵	پیچیدگی زمانی الگوریتم Dijkstra
۲۸	۵.۵	چرا Dijkstra برای گراف هایی که یال منفی دارند، کار نمی کند؟
۲۸	۶.۵	چرا اضافه کردن به یال ها و استفاده از Dijkstra جواب نمی دهد؟
۲۸ Bellman Ford	۷.۵	یک مثال از کاربرد الگوریتم Bellman Ford : تبدیل ارز
۳۱	۸.۵	شبکه Bellman Ford
۳۱	۹.۵	اثبات درستی الگوریتم Bellman Ford

۳۲	۱.۶	قضیه
۳۲	۲.۶	اثبات
۳۳	۳.۶	Finding Negative Cycle
۳۵	۴.۶	Infinite Arbitrage
۳۷	۵.۶	Bidirectional Search
۴۰	۶.۶	Bidirectional Dijkstra

فهرست مطالب

۳

سه راب نمازی - ۱۳۹۸/۱۲/۱۱

۴۵	۱.۸ تعریف درخت پوشای کمینه
۴۶	۲.۸ کاربرد درخت های پوشای کمینه
۴۷	۳.۸ بدست آوردن درخت پوشای کمینه
۵۲	۴.۸ خلاصه
۵۳	۹ الگوریتم جست وجو A* محمدعلی فراحت - ۱۳۹۸/۱۲/۱۳
۵۳	۱.۹ ایده کلی
۵۳	۲.۹ Potential-function
۵۴	۳.۹ محاسبه کوتاه ترین مسیر
۵۴	۴.۹ Bidirectional-A*
۵۶	SuffixTree ۱۱ احمد بهمنی - ۱۳۹۹/۱/۳
۵۶	۱.۱۱ مقدمه
۵۷	۲.۱۱ پیدا کردن الگو در رشته
۶۱	BWT ۱۲ متین مرجانی - ۱۳۹۹/۱/۱۷
۶۱	۱.۱۲ مقدمه
۶۱	۲.۱۲ BWT
۶۲	۳.۱۲ Constructing BWT
۶۴	۴.۱۲ Inverting BWT
۶۶	۱۳ تطبیق الگوها و تبدیل BW شایان موسوی نیا - ۱۳۹۹/۲/۱۶
۶۶	۱.۱۳ یک مشاهده عجیب
۷۰	۲.۱۳ پیدا کردن تطبیق الگو با استفاده از BWT
۷۴	۱۴ الگوریتم KMP امید میرزا جانی - ۱۳۹۹/۲/۱۲
۷۴	Brute Force ۱.۱۴

فهرست مطالب

٤	
٧٥ Function Prefix ٢.١٤
٧٦ الگوریتم نهایی ٣.١٤
٧٧ الگوریتم kmp و effcient suffix array ١٥ صدا خاموشی - ١٣٩٨/٢/٦
٧٧ ١.١٥ معیار ارزیابی جزو
٨٥ LCP آرایه پسوندی بهینه و آرایه LCP ١٦ زهرا حسینی - ١٣٩٩/١/٢٦
٨٥ ١.١٦ دوره مفاهیم آرایه و درخت پسوندی
٨٦ ٢.١٦ ساخت آرایه پسوندی
٩٥ LCP ARRAY ٣.١٦
٩٩ SuffixTree ١٧ هستی کرمند - ١٣٩٩/١/٣١
١٠٧ جریان در گراف ١٨ محمد مصطفی رستم خانی - ١٣٩٩/٢/٢
١٠٧ ١.١٨ جریان در شبکه: (flows in network)
١٠٨ ٢.١٨ network
١١٢ ٣.١٨ گراف باقیمانده: (residual graph)
١١٥ ٤.١٨ جریان باقیمانده: (residual flow)
١١٦ ٥.١٨ maxflow: and MinCut

جلسه ۲

Paths in Graphs

نگار زین العابدین - ۱۴۹۸/۱۱/۲۰

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:

Shortest path .۱

Minimom spanning trees .۲

Advances shortest path .۳

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

BFS (Breadth-First-Search) .۱

Dijkstra .۲

Bellmanford .۳

Shortest path ۲.۲

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

: BFS •

برای پیداکردن کوتاه‌ترین مسیر در گراف، از راسی (Node) به همه‌ی 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.

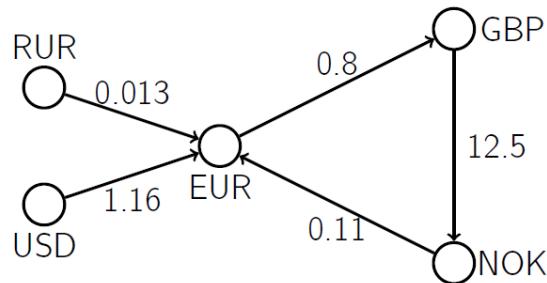
: Bellmanford •

همان طور که در بالا اشاره شد، اگر گراف مورد نظرمان وزن منفی داشته باشد، نمی‌توانیم از الگوریتم Dijkstra برای محاسبه کوتاه‌ترین مسیر استفاده کنیم. در این حالت، Bellmanford به کار بردۀ 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؟

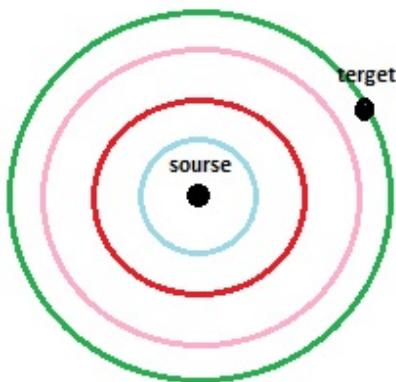
اگر در گراف وزن‌داری، دوری داشته باشیم که جمع یال‌های آن منفی شود، در این حالت گراف ما دارای دور منفی می‌باشد.

- 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 یشود. جزئیات این مسئله و ارتباط آن با دور منفی در جلسات آینده، شرح داده خواهد شد.



Dijkstra / BFS ۴.۲

اگر به زبان ساده‌ای به شرح این دو الگوریتم پیردازیم، می‌توانیم بگوییم که این دو الگوریتم، در ابتدا، از نقطه مبدأ شروع و تمام نهادنی 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.



BFS * شبکه

pseudocode*

```

Input: Graph,Source
Ouput: 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

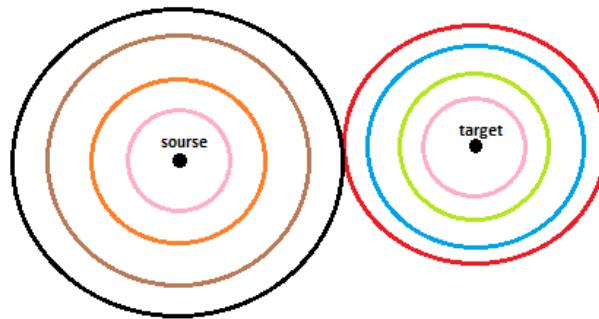
```

Algorithm 1: BFS on graph

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

Bidirectional dijkstr / Bidirectional BFS ۵.۲

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 مبدأ 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 dijkstra / Bidirectional BFS ، از تعدادی محاسبات بیهوده صرف نظر می‌شود که موجب سریعتر شدن برنامه می‌گردد.
- از تفاوت‌هایی که می‌توان بین دو الگوریتم Bidirectional BFS / Dijkstra و Bidi-directional dijkstra گرفت، در الگوریتم‌های دسته اول، محاسبات و عملیات‌ها بین راس مبدأ و تمام راس‌های دیگر گراف صورت می‌گیرد. این در حالی است که در الگوریتم‌های دسته دوم، تنها به دو راس مبدأ 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 علائی - 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 .

نحوه‌ی 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 فاصله ی 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 قبلًا بازدید شده است 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 .

گرفته شده است از [DIJKSTRA ۲]

Order of Dijkstra Algorithm ۳.۴

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

گرفته شده است 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 .

گرفته شده است از [book]

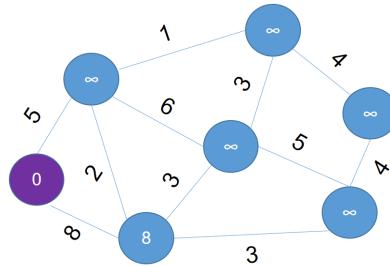
۵.۴ معایب Dijkstra Algorithm

۱) با یک جستجوی کورکورانه (blind search) روی منابع ، وقت زیادی را هدر می‌دهد .

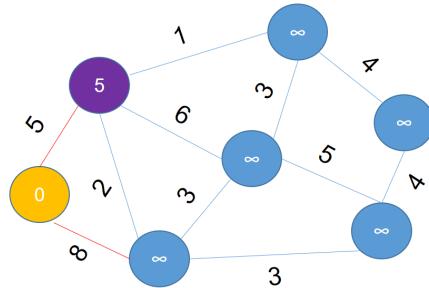
۲) اگر گراف دارای یال منفی باشد ، این الگوریتم روی آن کار نمی‌کند .

گرفته شده است از [book]

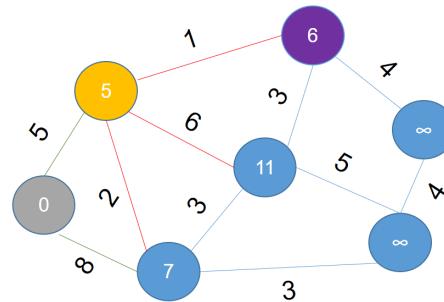
۶.۴ مثال :



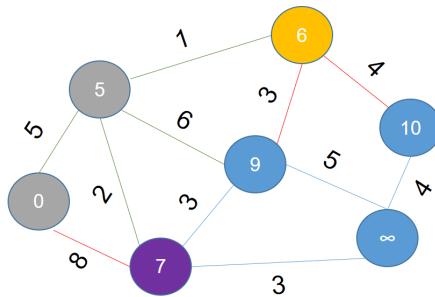
شکل ۱.۴ : [DIJKSTRA] مرحله‌ی اول



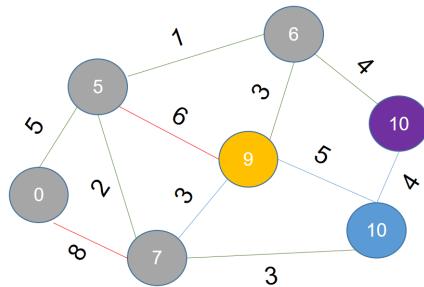
شکل ۲.۴ : [DIJKSTRA] مرحله‌ی دوم



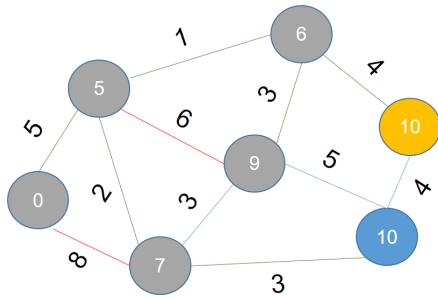
شکل ۳.۴: [Dijkstra] مرحله‌ی سوم



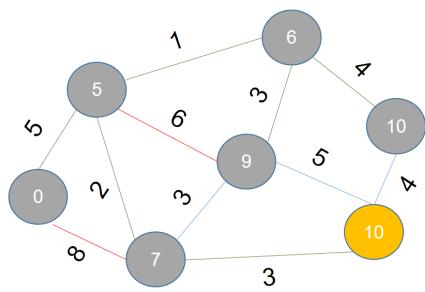
شکل ۴.۴: [Dijkstra] مرحله‌ی چهارم



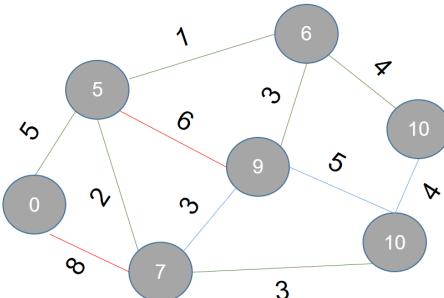
شکل ۵.۴: [Dijkstra] مرحله‌ی پنجم



شکل ۴.۴: [Dijkstra] مرحله‌ی ششم



شکل ۴.۵: [Dijkstra] مرحله‌ی هفتم



شکل ۴.۶: [Dijkstra] مرحله‌ی هشتم

شبہ کد Dijkstra Algorithm ۷.۴

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
     $dist[u] \leftarrow \infty$  ;
     $prev[u] \leftarrow \text{nil}$  ;
end
 $dist[S] \leftarrow 0$  ;
while at least one  $dist$  changes do
     $\mid$  relax all the edges ;
end

```

Algorithm 3: Naive Algorithm Of Dijkstra

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

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

[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، $\text{dist}[u] + \text{edge}(v, u)$ باید بیشتر از $\text{dist}[v] + \text{edge}(v, u)$ شود تا

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

Bellman–Ford algorithm ۹.۴

برای پیدا کردن کوتاه ترین 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 (section 4.2) Dijkstra 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 .

گرفته شده است از [bell2] [bell1]

Order of Bellman-Ford Algorithm ۱۰.۴

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

۱۱.۴ مزایای Bellman-Ford Algorithm

- ۱) اگر گراف دارای یال منفی باشد ، می‌تواند کوتاه ترین مسیر را بیابد .

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

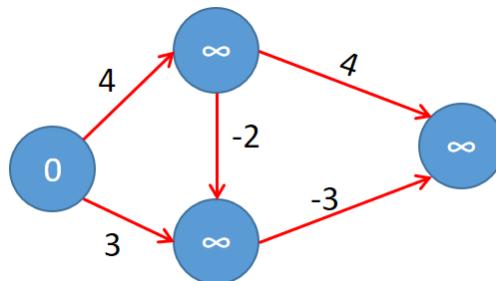
[book] گرفته شده است از

۱۲.۴ معایب : Bellman-Ford Algorithm

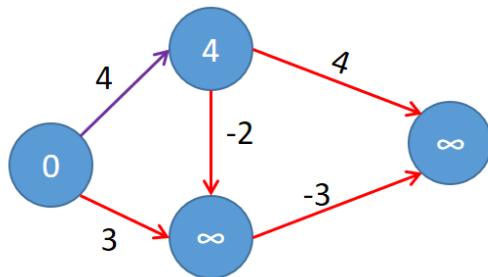
- ۱) مقیاس خوبی ندارد .
- ۲) تغییرات در توپولوژی شبکه به سرعت منعکس نمی‌شوند زیرا به روزرسانی‌ها یال به یال پخش می‌شوند.
- ۳) اگر گراف دارای دور منفی باشد ، این الگوریتم کارنمی‌کند .
- ۴) الگوریتم ([section 4.2](#)) Dijkstra نسبت به الگوریتم ([section 4.9](#)) Bellman-Ford کندتر است .

[book] [bell³] [bell¹] گرفته شده است از

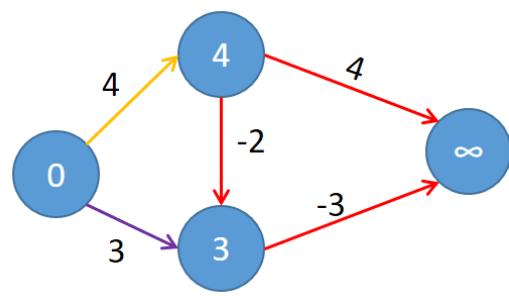
۱۳.۴ مثال :



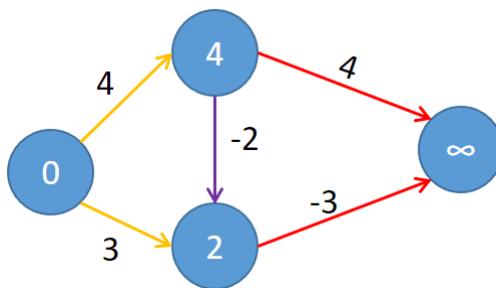
شکل ۹.۴: مرحله‌ی اول



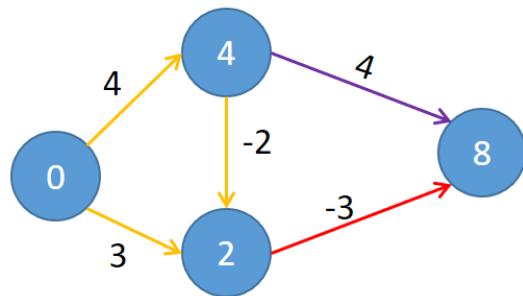
شکل ۱۰.۴: مرحله‌ی دوم



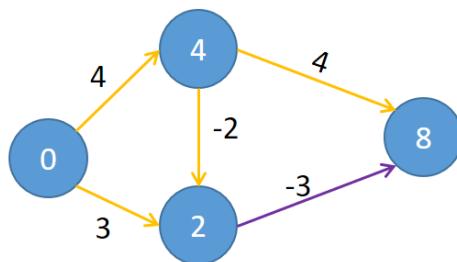
شکل ۱۱.۴: مرحله‌ی سوم



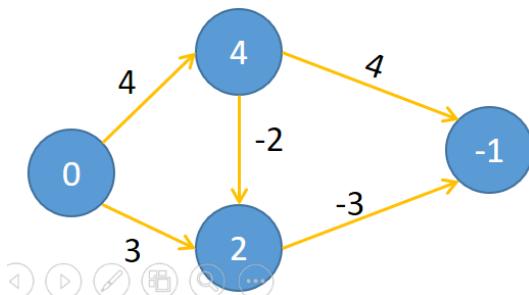
شکل ۱۲.۴: مرحله‌ی چهارم



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 شبکه 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) ، همهی کوتاهترین فاصله‌ها از راس شروع که حداقل شامل k یال هستند؛ مشخص شده اند.
با استفاده از استقرای ریاضی:

۱. اگر $k=0$ فاصله همه راس‌ها از راس شروع بینهاست، به غیر از خود راس شروع که.

$$\text{dist}[S]=0$$

۲. فرض استقرای: بعد از k بار relaxation همهی کوتاهترین مسیرها با طول حداقل k مشخص شده‌اند.

۳. حکم استقرای: قبل از $k+1$ بار $\text{dist}[u]$ ، relaxation با کمترین فاصله به طول حداقل k مشخص شده است. اگر از u یال‌هایی به راس‌های دیگر باشد، در دفعه‌ی $k+1$ همهی آنها relax می‌شوند، پس کوتاهترین مسیرها با حداقل طول $k+1$ (کوتاهترین مسیرها با حداقل طول k بعلاوه یک یال که آنها را به راس دیگری وصل کند) مشخص می‌شوند.

گرفته شده است از [book]

۱۶.۴ دور منفی در Bellman-Ford :

- ۰ • (section 4.9) Bellman-Ford Algorithm را به اندازه‌ی تعداد نودها اجرا می‌کنیم. نودهایی که در دور آخر ریلکس می‌شوند را در یک کیو یا لیست ذخیره می‌کنیم.
- ۰ • (۲) از $v \leftarrow x \leftarrow \text{prev}[x] \leftarrow \dots \leftarrow x$ را به اندازه‌ی نود‌ها تکرار می‌کنیم. به طور قطع در چرخه خواهد بود.
- ۰ • (۳) $y \leftarrow x \leftarrow \text{prev}[x] \leftarrow \dots \leftarrow x$ 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 دفعه‌ی $|V|$ 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 حتماً دوری با وزن منفی دارد)

اگر گرافی دارای دور با وزن منفی نباشد، کوتاهترین مسیرها از راس شروع حداقل 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 فاصله‌ای 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 حتماً در بار $|V|$ ام یکی از فاصله‌ها آپدیت می‌شوند) فرض کنید گرافی با دور با وزن منفی داریم مثل $a \rightarrow b \rightarrow c \rightarrow a$ اما در دفعه‌ی $|V|$ ام یال relaxation ها، هیچ یالی relax نشود. برای اینکه هیچ‌کدام relax نشوند، باید فاصله‌ای که هر راس دارد از فاصله‌ی راس مجاور بعلاوه‌ی یال بین آن دو بزرگ‌تر باشد، حتی برای سه راس a, b, c ، پس داریم:

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

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

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

$$w(a,b) + w(b,c) + w(c,a) >= 0$$

در حالی که دور بین این سه راس مجموع وزنش باید منفی باشد، پس به تناقض خوردیم، و حتماً حداقل یکی از

[book] relax شود. بالا باید

۱۸.۴ : Infinite Arbitrage

- ۱) به اندازه‌ی تعداد نودها (section 4.9) Bellman-Ford Algorithm 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 قبلاً 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.

[book] گرفته شده است از

۱۹.۴ اثبات درست بودن الگوریتم

(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 کرده کرد)

اثبات سمت \Rightarrow 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 دفعه‌ی $|V|$ 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، $dist[u]=L$ باشد. از آنجا که فاصله u از s می‌تواند کمتر کمتر شود، پس در بعضی از دفعات بعدی اجرای بلمن-فورد مثلا $V=k$ فاصله اش تغییر می‌کند. اگر

راسی در دفعه i از $i+1$ (algorithm 2) relaxtion یالها فاصله‌اش تغییر نکند، همه یال‌هایی که از آن راس آغاز می‌شوند نیز در دفعه i 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 قبل از تغییر کرده باشد. پس برای اینکه $\text{dist}[u] \leq V$ 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 نحوه 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 شبکه و کد الگوریتم بلمن-فورد را مشاهده کنید : [لینک سوم](#)

- در این سایت می‌توانید مثال و توضیح و کد و ویدیوی مرتبه 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 شبکه و مرحله، مرحله انجام‌شدن شبکه 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 شبکه 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، به‌طوری که الیت 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[ ] ← an array with size of |V| filled with +∞;
prev[ ] ← an array with size of |V| filled with null;
dist[S] ← 0;
H ← MakeQueue(V);
while H is not empty do
    u ← ExtractMin(H);
    for all (u, v) in E do
        if dist[v] > dist[u] + w(u, v) then
            dist[v] ← dist[u] + w(u, v);
            prev [v] ← u;
            ChangePriority(H, v, dist[v]);
        end
    end
end

```

Algorithm 6: Dijkstra Algorithm

۲.۵ شبکه کد پیدا کردن کوتاه ترین 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 ← x;

while n != S **do**

 path.add(n);

 n ← prev[n];

end

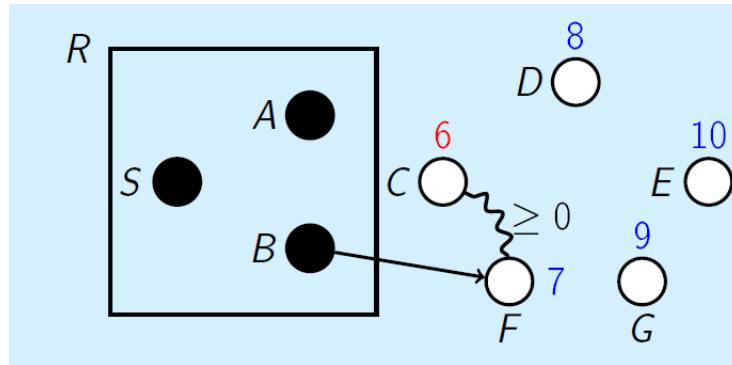
Algorithm 7: Finding Shortest Path

• پیچیدگی زمانی: در گرافی که دور منفی وجود نداشته باشد، کوتاهترین راه از تمام رأس‌ها حداقل یک

بار عبور می‌کند. پس پیچیدگی زمانی این شبکه کد 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 مبدأ 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 C و S قرار دارد. اما مقدار dist این رأس بیشتر از ۶ است و یال بین F و C 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

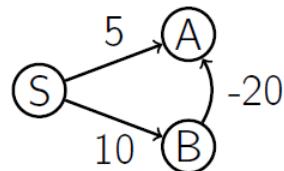
طبق شبهکد ۹، پیچیدگی زمانی این الگوریتم برابر است با مجموع پیچیدگی‌های زمانی این سه قسمت:

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

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

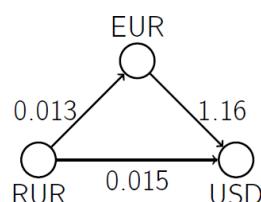
در این مثال، رأس A به رأس S نزدیک‌تر است. اما کوتاهترین راه به S ، از رأس B که از S دورتر است، می‌گذرد.

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 بدھیم 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

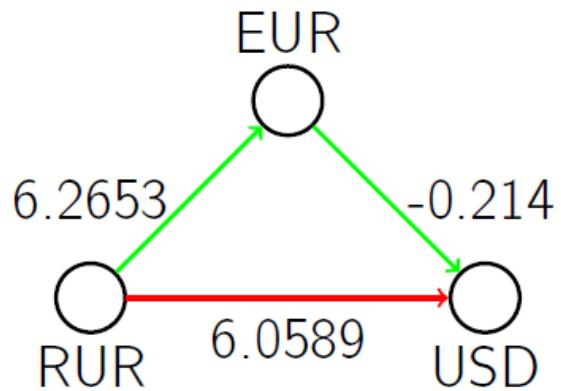
سپس هر یال را قرینه می‌کنیم. در این صورت، کافی است حداقل مجموع قرینه لگاریتم یال‌ها، محاسبه شود.

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

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

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 یا ل منفی وجود دارد، از الگوریتم Bellman Ford استفاده می‌شود،
نه Dijkstra.

Bellman Ford شبکه ۸.۵

Data: Graph G, Node Source

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

in Graph G

```

dist[ ] ← an array with size of |V| filled with +∞;
prev[ ] ← an array with size of |V| filled with null;
dist[S] ← 0;
H ← 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 $O(|V||E|)$:Bellman Ford

۹.۵ اثبات درستی الگوریتم Bellman Ford

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

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

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

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

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 تنافق می رسیم پس حتما در V امین تکرار در یک گراف با دور منفی حتما تعدادی از فاصله ها تغییر می کنند.

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

Finding Negative Cycle ۳.۶

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.
- v از دور منفی قابل دسترسی است
- از v -> x شروع می کنیم، $[x]-prev[x]$ را $|V|$ بار انجام می دهیم، در نهایت به داخل حلقه خواهیم رسید.
- $y=x$ را ذخیره می کنیم و $[x]-prev[x]$ 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:

```

1  public bool HasNegativeCycle(Graph G, int Start, long N)
2  {
3      long[] dist = new long[N];
4      dist[Start] = 0;
5      for (int i = 0; i < N - 1; i++)
6      {
7          foreach (edge e in Graph.edges)
8              relax(e);
9      }
10     foreach (edge e in Graph.edges)
11     {
12         if (relax(e))
13             return true;
14     }
15     return false;
16 }
```

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 فورد فاصله شان تغییر پیدا می کند را ذخیره کنیم و با استفاده از متد زیر دورهای منفی گراف را بیابیم:

```

1  List<long> FindNegCycle(int v, long[] Prev, int N)
2  {
3      long x = v;
4      for (int i = 0; i < N; i++)
5          x = Prev[x];
6      List<long> cycle = new List<long>();
7      long y = Prev[x];
8      while (y != x)
9      {
10          cycle.Add(y);
11          y = Prev[y];
12      }
13      return cycle;
14 }
```

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.

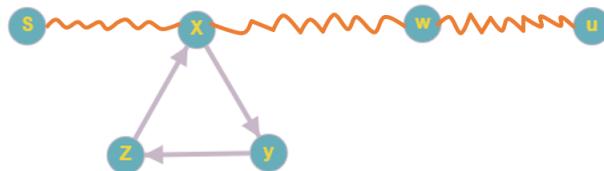
Infinite Arbitrage ۴.۶

۱.۶ 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.

۲.۶ اثبات

اگر $dist[w]$ 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 می کنیم L طول کوتاه ترین مسیر به u با حداقل $V-1$ بال باشد. بعد از $V-1$ تکرار $dist[u]$ برابر L خواهد بود. برای داشتن Infinite Arbitrage به u یک مسیر کوچک تر از L وجود خواهد داشت. بنابراین $dist[u]$ در تکراری که $k >= V$ باشد کاهش خواهد یافت. اگر یال (x, y) ریلکس نشود و $dist[x] \neq \infty$ در i امین تکرار کاهش نیابد، پس یال (y, x) در $i+1$ امین تکرار هم ریلکس نخواهد شد. تنها گره هایی که گره های ریلکس شده در تکرار قبلی قابل دسترسی هستند می توانند ریلکس شوند. اگر $dist[u] < dist[u]$ اگر $k >= V$ در یک تکرار $k >= V$ کاهش نیابد، u از گره هایی که V امین تکرار ریلکس شده اند قابل دسترسی است.

Detect Infinite Arbitrage ۳.۶

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

- V بار الگوریتم بلمن فورد را اجرا می کنیم و گره هایی که هایی که در بار V ام ریلکس می شوند را در مجموعه A ذخیره می کنیم.

- 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 مت زیر گره هایی که دارای Infinite Arbitrage هستند را پیدا می کنیم:

```

1 public List<long> DetectInfiniteArbitrage(Graph G, int Start, long N)
2 {
3     long[] dist = new long[N];
4     dist[Start] = 0;
5     Queue<long> relaxed = new Queue<long>();
6     List<long> Arbitrage = new List<long>();
7     for (int i = 0; i < N - 1; i++)
8         foreach (edge e in Graph.edges)
9             relax(e);
10    foreach (edge e in Graph.edges)
11        if (relax(e))
12            relaxed.Enqueue(e.target);
13    foreach(var v in relaxed)
14    {
15        List<long> nodes = BFS(G, v);
16        foreach (var u in nodes)
17            if (!Arbitrage.Contains(u))
18                Arbitrage.Add(u);
19    }
20    return Arbitrage;
21 }
```

نمونه کد ۳ : Detect Infinite Arbitrage

Reconstruct Infinite Arbitrage ۴.۴.۶

- در طول parent ، BFS 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 u دست پیدا کنیم.

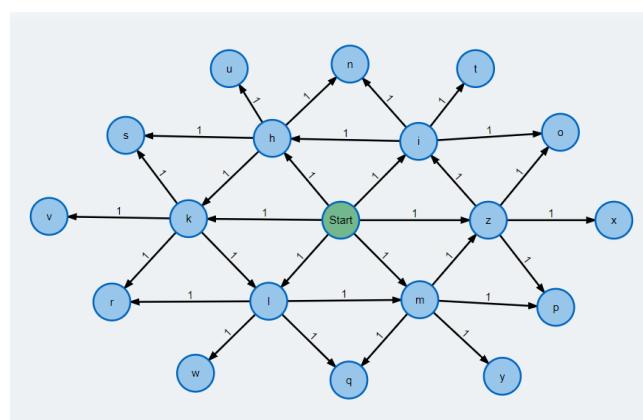
Bidirectional Search 5.9

Why not just Dijkstra 1.0.6

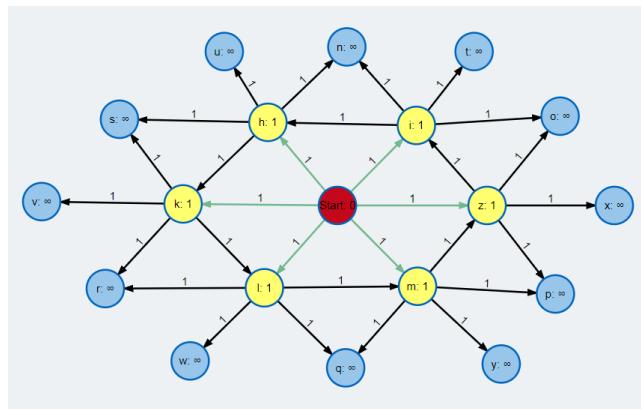
برای بدست آوردن کوتاه ترین مسیر از گره ای به گره دیگر گاهی وقت دایجسترا الگوریتم مناسبی نیست ولی چرا؟ پیچیدگی زمانی دایجسترا $(|V| \log(|E| + |V|))$ است که نسبتاً سریع است پس چرا ممکن است گاهی ب اندازه کافی خوب نباشد؟

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

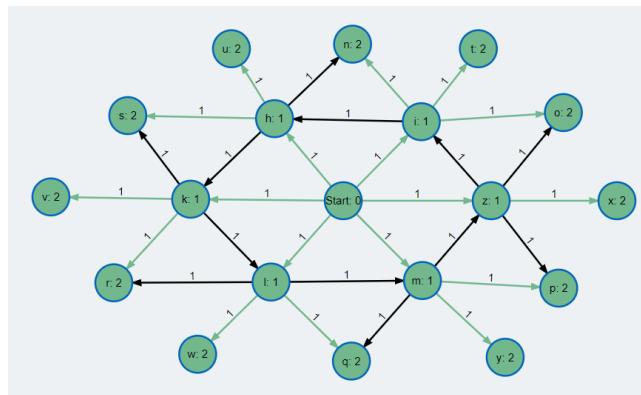
در شکل های زیر نحوه پیشرفت الگوریتم دایجسترا روی یک گراف را می بینیم و خواهیم فهمید که چرا ممکن است سریع عمل نکند: [dijder]



شكل ٢.٦: دایجسترا(۱)



شکل ۶: دایجسترا(۲)



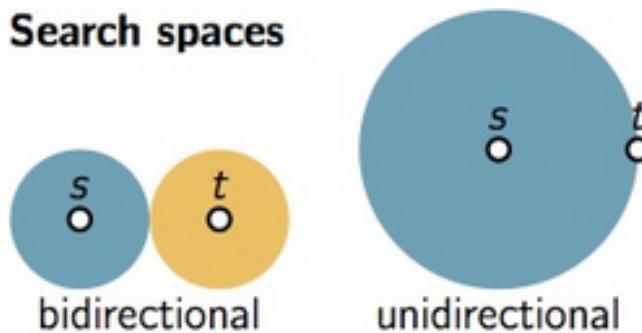
شکل ۶: دایجسترا(۳)

Growing Circle ۲.۵.۶

قضیه: وقتی که راس v از طریق Extract Min انتخاب می شود، فاصله s برابر قطر u 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 قبل از 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 می توانیم مقدار فضا و سطحی که هر دو نوع دایجسترا پوشش می دهند 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 نقشه های جاده ای نسبتاً خوب عمل می کند و سرعت پیدا کردن را تقریباً دو برابر می کند. اما باید برای گراف شبکه های اجتماعی هم بررسی شود.

Six Handshakes ۳.۵.۶

در سال ۱۹۲۹، فریگیس کاریتنی، ریاضیدان مجارستانی، فرضیه ای به نام دنیای کوچک را مطرح کرد. بر اساس این فرضیه شما حداقل با ۶ بار دست به دست کردن یک پیام می توانید آن را به هر کسی در سراسر دنیا برسانید. این فرضیه بر اساس آزمایش های مختلف نزدیک به حقیقت است.

Facebook ۴.۵.۶

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 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.

می خواهیم کوتاه ترین مسیر بین باب و مایکل از طریق ارتباطات دوستی پیدا کنیم. برای دورترین مردم دایجسترا حداقل بین ۲ میلیون نفر جستجو می کند. اگر ما در دوستان دوستان باب و مایکل جستجو کنیم یک راه ارتباطی پیدا خواهیم کرد. حداقل بین یک میلیون دوست هر کدام جستجو خواهیم کرد که حداقل

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

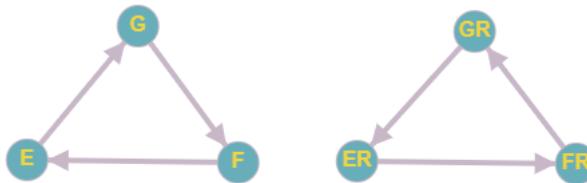
این روش را می توانیم نه تنها در گراف بلکه در هر مورد دیگر هم به کار بگیریم. با توجه به مثال قبلی پیچیدگی زمانی از $O(\sqrt{n})$ به $O(n)$ تغییر خواهد کرد.

Bidirectional Dijkstra ۶.۶

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

Reversed Graph ۱.۶.۶

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

Algorithm ۲.۶.۶

- 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 می کنیم v اولین راسی باشد که هم در G^R و هم در G پردازش شده باشد. آیا از آن پیروی می کند که کوتاه ترین مسیری که از s به t وجود دارد از v می گذرد؟



شکل ۷.۶ Compute Distance :

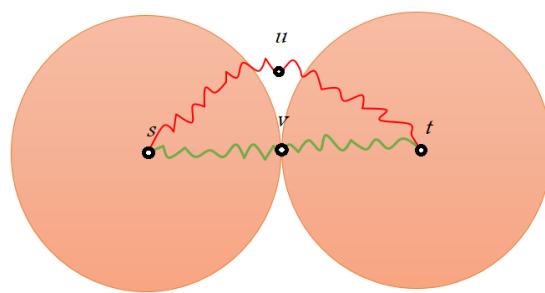
۴.۶.۶ قضیه

فرض کنید $dist[u]$ فاصله ای است که در دایجسترا از s در گراف G تخمین زده می شود و $dist^R[u]$ به طور یکسان در دایجسترا از t در گراف G^R باشد. کوتاه ترین مسیر از s تا t از گره ای مانند u می گذرد که یا در G^R یا در G یا در هر دو آن ها پردازش شده است و خواهیم داشت:

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

۵.۶.۶ اثبات

فرض می کنیم راسی مانند u که در خارج از دایرہ دایجسترا چه در G^R و چه در G باشد و کوتاه ترین مسیر ما از این راس می گذرد در حالی که هنوز پردازش نشده است. در اینجا به تناقض می رسیم چون اگر راس u در فاصله کمتری از s یا t قرار داشت باید زودتر پردازش می شد.



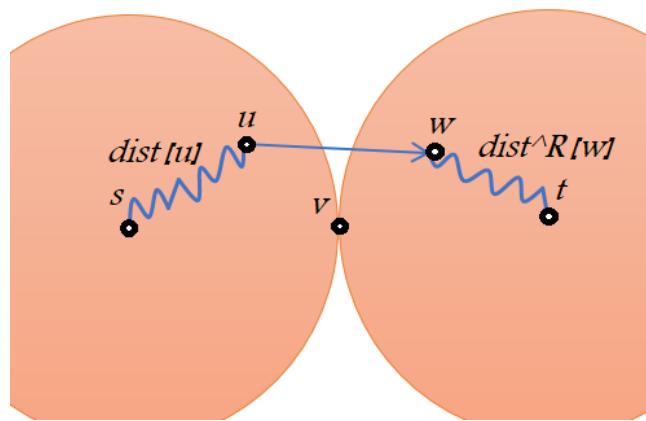
شکل ۸.۶ Proof :

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

با توجه به نکته ای در اثبات قبلی گفتیم راسی که از آن کوتاه ترین مسیر می‌گذرد نمی‌تواند 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 بگردد و ممکن است راس‌هایی با مسیر کوتاه‌تر هم وجود داشته باشند. بنابر این دو قضیه گفته شده دایجسترا دو طرفه اثبات می‌شود.

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

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



شکل (۲) : ۹.۶ Proof(۲)

Function $\text{Process}(u, G, \text{dist}, \text{prev}, \text{proc})$:

```

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 $\text{Bidirectional Dijkstra}(G, s, t)$:

```

 $G^R \leftarrow \text{ReverseGraph}(G)$ 
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,...)
    | end
    |  $v^R \leftarrow ExtractMin(dist^R)$ 
    | repeat symmetrically for  $v^R$  as for  $v$ 
end

```

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

بدست آمده از متد های قبلی و قضیه ذکر شده بدست آوریم. شبیه 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  $\leftarrow inf$ , ubest  $\leftarrow None$ 
    for u in proc+procR do
        if dist[u]+distR[u]<distance then
            ubest  $\leftarrow u$ 
            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

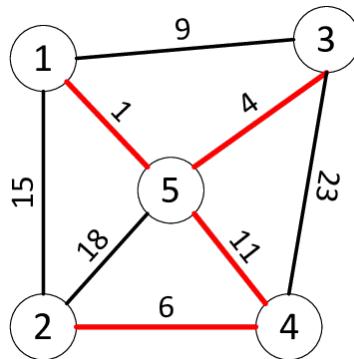
۱.۸ تعریف درخت پوشای کمینه

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 صرفا 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 پوشای میگوییم.

حال بار دیگر همین مسئله را درنظر 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 پوشای 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
- طبیعتا از آنجا که حاصل درخت است، باید تعداد یال ها دقیقا 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.

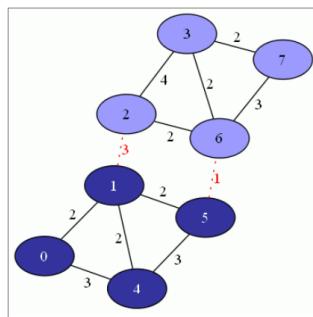
اما مسئله مهم چگونگی بدست آوردن درخت پوشای کمینه ی یک گراف اولیه است که در ادامه به این موضوع خواهیم پرداخت.

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 خواهد 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 ۲.۸: 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 بر حسب 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 دیسجوبینت 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 مباحثت مربوط به دیسجوبینت 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:

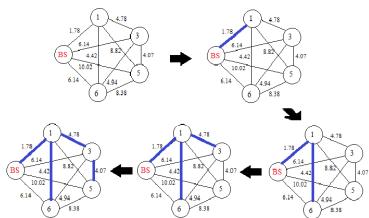
$$2|E| \cdot T(\text{Find}) + |V| \cdot T(\text{Union}) = O((|E|+|V|)\log|V|) = O(|E|\log|V|)$$

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 جزو آورده شده است مراجعه کنید. [Prim VS Kruskal]

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.

جلسه ۸. درخت های پوشای کمینه

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

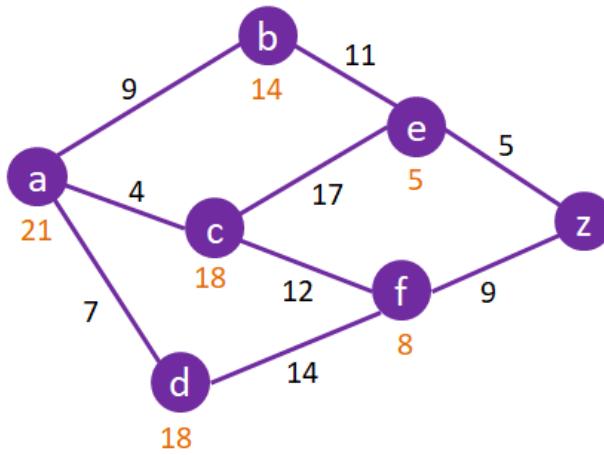
جزوه جلسه ۱۹ مورخ ۱۳۹۸/۱۲/۱۳ درس طراحی و تحلیل الگوریتم تهیه شده توسط محمدعلی فراحت. در جهت مستند کردن مطالب درس طراحی و تحلیل الگوریتم.

۱.۹ ایده کلی

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

2.9 Potential-function

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 کوتاه ترین 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

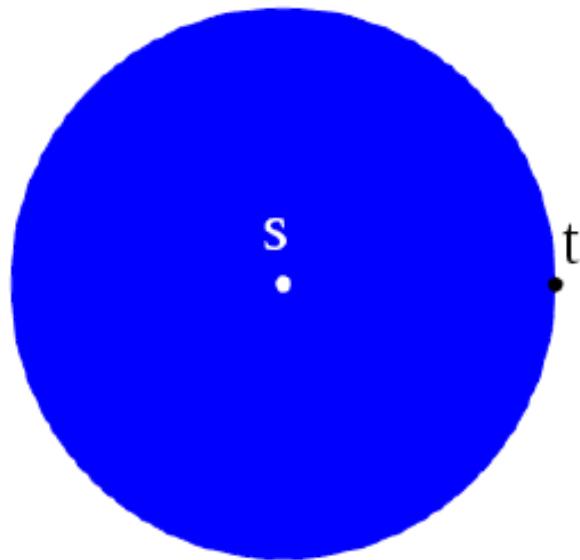
ایده کلی این الگوریتم مانند الگوریتم Dijkstra است . با این تفاوت که وزن هر یال را با فرمول زیر محاسبه می کنیم :

$$\ell_{\pi}(u, v) = \ell(u, v) - \pi(u) + \pi(v)$$

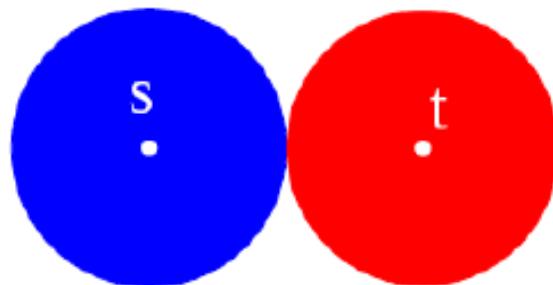
در این فرمول ℓ همان وزن اولیه یال است و π_i همان تابع پتانسیل است .

Bidirectional-A* ۴.۹

تفاوت این روش با روش قبلی این است که ما هم از طرف راس مبدأ و هم از طرف راس مقصد الگوریتم سرچ را شروع می کنیم و محلی که به هم مرسند را پیدا کرده و کوتاهترین مسیر پیدا می شود . در شکل های زیر می توان تفاوت محسوس این دو روش را به راحتی متوجه شد :



شكل ۲.۹: روش معمولی



شكل ۳.۹: روش bidirectional

می‌بینیم 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 بایوانفورماتیک[†] 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

```

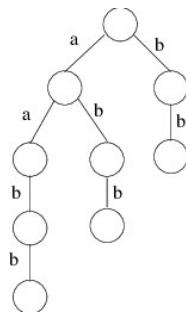
1  class PatternInText {
2
3      void search(String txt, String pat)
4      {
5          int M = pat.Length;
6          int N = txt.Length;
7
8          for (int i = 0; i <= N - M; i++) {
9              int j;
10
11             for (j = 0; j < M; j++)
12                 if (txt[i + j] != pat[j])
13                     break;
14
15             if (j == M)
16                 Console.WriteLine(" index at found "Pattern + i);
17         }
18     }
19 }
```

نمونه کد ۴: راه ساده در سی شارپ

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

۲.۲.۱۱ ساخت Suffix 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 : ۱.۱۱

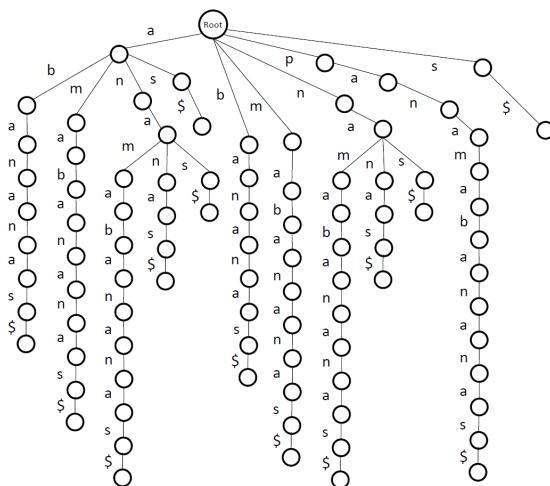
با بررسی Trie ساخته شده از الگوها، به ازای همه کاراکترهای متن، می‌توان الگوها و مکانشان در متن را پیدا کرد. این روش از `(|Text|*|LongestPattern| 0)` می‌باشد (ساخت Trie از `(|Pattern| 0)` است). همچنین از نظر حافظه این روش از `(|Patterns| 0)` می‌باشد.[§]

[§](در ذهن انسان طول الگوها حدود 10^{12} است)

۳.۲.۱۱ ساخت Suffix Tree از متن

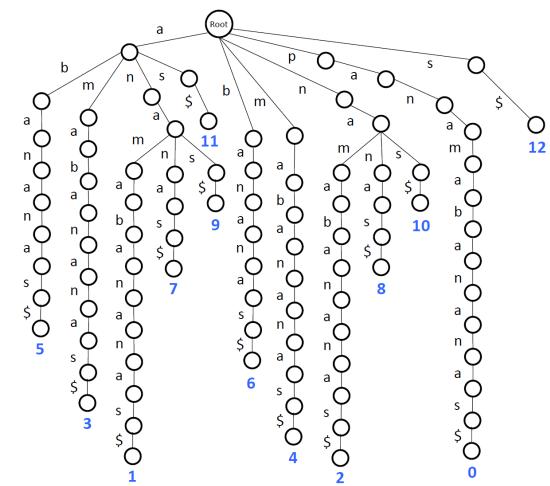
اگر به جای ساختن Trie از الگوها، همهی Suffix های متن را پیدا کنیم و از آن ها Trie بسازیم، به

۲.۱۱ متن رسیده ایم. Suffix Tree



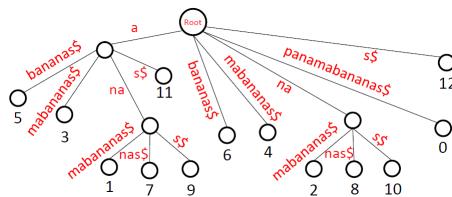
شکل ۲.۱۱: Suffix Tree for "panamabanana"

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



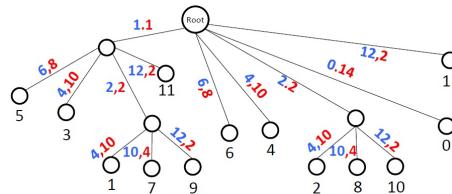
شکل ۳.۱۱: Suffix Tree with indexes for "panamabanana"

از نظر حافظه به اندازه $Text^2$ 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.



شکل ۱۱ Compressed Suffix Tree with indexes for "panamabananana" :۴.۱۱

اما با اینکار باز هم باید به اندازه $Text^2$ 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 $|Text|$ می‌باشد.



شکل ۱۱ Compressed Suffix Tree with indexes for "panamabananana" :۵.۱۱

جلسه ۱۲

BWT

متین مرجانی - ۱۳۹۹/۱/۱۷

۱.۱۲ مقدمه

در جلسه‌ی 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 Node‌های درخت و در نتیجه حافظه‌ای اشغال می‌شد را کاهش دادیم. در Suffix Tree حجم حافظه ای از $O(|Text|)$ می‌باشد که پیشرفت قابل توجهی نسبت به Suffix Trie است.

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

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

۲.۱۲ BWT

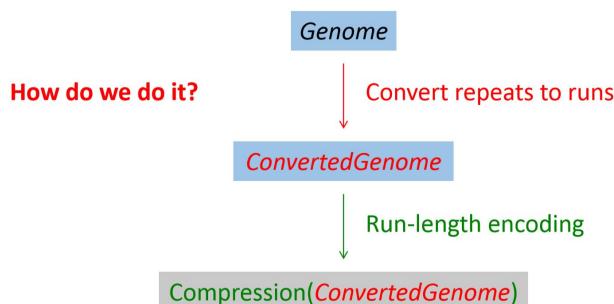
همانطور که از اسمش پیداست BWT یک تبدیل برای رشته‌ی مورد نظر است. خاصیت این تبدیل برگشت پذیر بودن آن است.

هدف این تبدیل این است که کاراکتر های یکسان حداکثر 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 ای کمتری میگیرند.

Run-length encoding :

Text
GGGGGGGGGGCCCCCCCCCCAAAAAAATTTTTTTTTTCCCCG
=
10G11CVA15T5C1G

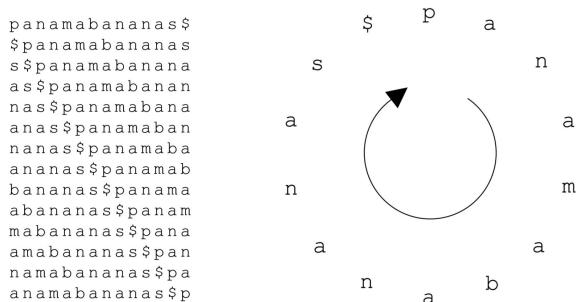
حروف داخل ژنوم انسان طول تکرار زیادی ندارند اما چون از تعداد حروف محدود (۴) تشکیل شده اند پس بااعمال کردن این تبدیل میتوان رشته ای آن را بسیار فشرده کرد.



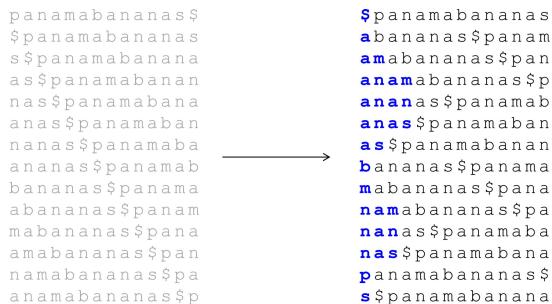
Constructing BWT ۳.۱۲

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

Cyclic Rotations



سپس همه‌ی این رشته‌ها را بر اساس حروف الفبا Sort می‌کنیم :



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.

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

رشته‌ی بdst آمده همان تبدیل BWT مورد نظر می‌باشد:

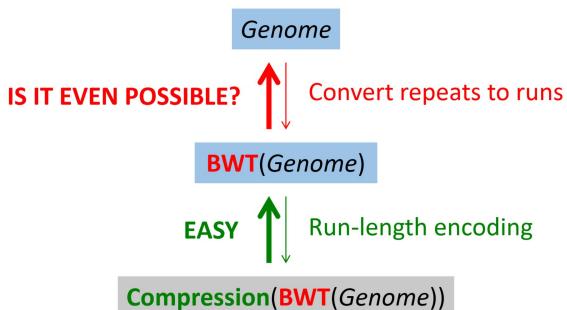
$BWT(\text{panamabananas\$}) = \text{smnpbnnaaaa\$a}$

همانطور که می‌بینید حروف مشابه در این تبدیل بهم نزدیک‌تر شدند:

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

Inverting BWT ۴.۱۲

حال باید نشان دهیم چگونه می‌توان از یک رشته‌ی تبدیل شده به خود اصل آن رشته برگشت.



فرض کنیم رشته‌ی تبدیل شده $\text{annb\$aa}$ است و رشته‌ی اصلی که می‌خوایم به آن برگردیم می‌باشد.

با Sort کردن حروف رشته‌ای که داریم، حروف ستون اول ماتریس تبدیل BWT بdst می‌آید. ستون آخر هم همان حروف رشته‌ی تبدیل می‌باشد.

$\$$	b a n a n a	$a \$$
a	$\$ b a n a n$	$n a$
n	$a n a \$ b a n$	$n a$
b	$a n a n a \$ b$	$b a$
$\$$	$b a n a n a \$$	$\$ b$
a	$n a \$ b a n a$	$a n$
n	$a n a \$ b a$	$a n$

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 کاراکتر های رشته ای اصلی را تا الان دو به دو مرتب کردیم. این یعنی دو حرف اول ماتریس را بدست آورده ایم. 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.

\$b a n a n a		\$b
a\$ ba n a		a\$
a na \$ ba n		a n
a na n a \$ b	→	a n
b a n a n a \$ b	2-mers	b a
n a \$ ba n a		n a
n a n a \$ ba		n a

Sort

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 ای اصلی را بدست اوردهیم.

\$b a n a n a		a \$ b
a\$ ba n a		n a \$
a na \$ ba n		n a n
a na n a \$ b	→	b a n
b a n a n a \$ b	3-mers	\$ b a
n a \$ ba n a		a n a
n a n a \$ ba		a n a

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

\$b a n a n a	a \$ b a n a	\$b a n a n a
a\$ ba n a	n a \$ b a n	a\$ ba n a
a na \$ ba n	n a n a \$ b a	a na \$ ba n
a na n a \$ b	b a n a n a \$ b a	a na n a \$ b
b a n a n a \$ b	6-mers	b a n a n a \$ b
n a \$ ba n a	Sort	n a \$ ba n a
n a n a \$ ba		n a n a \$ b

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

جلسه ۱۳

تطبیق الگوها و تبدیل BW

شایان موسوی نیا - ۱۳۹۹/۲/۱۶

جزوه جلسه ۱۳ام 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 اشنا 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 ماتریس های پی ببریم.

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

شکل ۱.۱۳: ماتریس حالات

اگر به شکل ۱.۱۳ توجه کنیم، میتوان به یک مشاهده جالب رسید. به ظور مثال بباید 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 بباید a را از اول سطر های ۲ تا ۷ حذف کنیم و انها را به انتهای سطر های خودشان اضافه کنیم تا همان سطر های پایین درست شود. حال میتوان به این موضوع پی برد که ترتیب شماره گذاری a های سطر های پایین همان ترتیب شماره گذاری a های بالا است. این موضوع برای باقی حروف نیز درست است.

- امین n و قوع حرف در اولین سطر
- و امین n و قوع حرف در اخرین 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 ۲.۱۳: متن شماره گذاری شده

$\$_1panamabanana s_1$
 $a_1bananas\$panam_1$
 $a_2mabananas\$pana_1$
 $a_3namabanas\$p_1$
 $a_4nanas\$panamab_1$
 $a_5nas\$panamaba_2$
 $a_6s\$panamabana_3$
 $b_1ananas\$panama_1$
 $m_1abanas\$pana_2$
 $n_1amabanas\$pa_3$
 $n_2anas\$panamab_4$
 $n_3as\$panamaban_5$
 $p_1anamabanas\$_1$
 $s_1\$panamabanan_6$

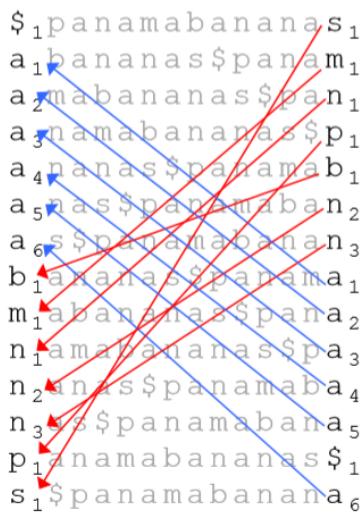
شکل ۳.۱۳: ماتریس حالت شماره گذاری شده

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

حال بباید 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 خرف قبلی مبرویم و این کار را تا زمانی انعام 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 a به ۶ امین n میرویم و این کار را تا اخر ادامه میدهیم طوری که در مرحله اخر 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 : $2 * |Text|$
- Time : $O(|Text|)$

به تطبیق الگوهای برگردیم.

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

- Memory : $20 * |Text|$
- Time : $O(|Text| + |Patterns|)$

میدانیم که طول ژنوم های انسان 3^{10} ضریبدر 10^9 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؟

بیایید 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.

\$	₁	panamabananas	₁
a	₁	bananas	\$panam ₁
a ₂	mabananas	\$pa n ₁	
a	₃	namabananas	\$p ₁
a	₄	anas	\$panamab ₁
a ₅	nas	\$panamaba n ₂	
a ₆	s	\$panamaba n ₃	
b	₁	ananas	\$panama ₁
m	₁	abananas	\$pana ₂
n	₁	amabananas	\$pa ₃
n	₂	anas	\$panamaba ₄
n	₃	as	\$panamaban ₅
p	₁	anamabananas	\$ ₁
s	₁	\$panamabanana ₆	

شکل ۵.۱۳: مثال ۱

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

\$	₁	panamabananas	₁
a	₁	bananas	\$panam ₁
a ₂	mabananas	\$pan ₁	
a ₃	na	mabananas\$p ₁	
a ₄	na	nas\$panamab ₁	
a ₅	na	s\$panamaban ₂	
a	₆	s\$panamaban ₃	
b	₁	ananas	\$panama ₁
m	₁	abananas	\$pana ₂
n	₁	amabananas	\$pa ₃
n	₂	anas	\$panamaba ₄
n	₃	as	\$panamaban ₅
p	₁	anamabananas	\$ ₁
s	₁	\$panamabanana ₆	

شکل ۶.۱۳: مرحله دوم

حالا به شکل الگوریتمی بباید سوال بالا حل کنیم.

ابتدا با دو مفهوم اندیس بالا و اندیس پایین بخش باید اشنا بشیم.

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

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

`BWMatching(FirstColumn,LastColumn,Pattern,LastToFirst)`

```

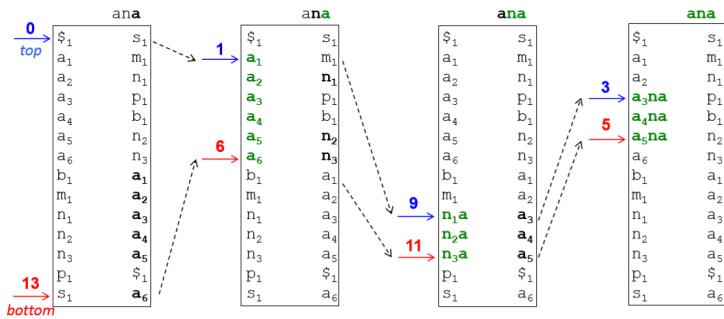
top <- 0
bottom <- |LastColmun| - 1
while top <= bottom do
    if Pattern is nonempty then
        symbol <- last letter in Pattern
        remove last letter from pattern
        if positions from top to bottom in LastColumn contain symbol
        then
            topIndex <- first position of symbol among positions form
            top to bottom in LastColumn
            bottomIndex <- last position of symbol among positions
            form 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 هارو بررسی میکند که همین باعث کندی برنامه میشود.

پس باید الگوریتم بالا را بهبود پختشید.



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 :

رخ داد نماد در اولین i مکان در اخرین ستون = Count.symbol(i .LastColumn)

i	$FirstColumn$	$LastColumn$	$LASTTOFIRST(i)$	COUNT
0	\$ ₁	s ₁	13	\$ 0 0 0 0 0 0 0
1	a ₁	m ₁	8	a 0 0 0 0 0 0 1
2	a ₂	n ₁	9	a 0 0 0 1 0 0 1
3	a ₃	p ₁	12	a 0 0 0 1 1 0 1
4	a ₄	b ₁	7	a 0 0 0 1 1 1 1
5	a ₅	n ₂	10	a 0 0 1 1 1 1 1
6	a ₆	n ₃	11	a 0 0 1 1 2 1 1
7	b ₁	a ₁	1	b 0 0 1 1 3 1 1
8	m ₁	a ₂	2	m 0 1 1 1 3 1 1
9	n ₁	a ₃	3	n 0 2 1 1 3 1 1
10	n ₂	a ₄	4	n 0 3 1 1 3 1 1
11	n ₃	a ₅	5	n 0 4 1 1 3 1 1
12	p ₁	\$ ₁	0	p 0 5 1 1 3 1 1
13	s ₁	a ₆	6	s 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 nonempty 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

```

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 کجا متن اماده آند.
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 namabanananas\$p ₁
7	a ₄ nanas\$panamab ₁
9	a ₅ nas\$panamaban ₂
11	a ₆ s\$panamaban ₃
6	b ₁ ananas\$panama ₁
4	m ₁ abanananas\$pana ₂
2	n ₁ amabanananas\$pa ₃
8	n ₂ anas\$panamaba ₄
10	n ₃ as\$panamabana ₅
0	p ₁ anamabananas\$ ₁
12	s ₁ \$panamabanan ₆

شکل ۹.۱۳: ارایه پسوند

جلسه ۱۴

الگوریتم KMP

امید میرزا جانی - ۱۳۹۹/۲/۱۲

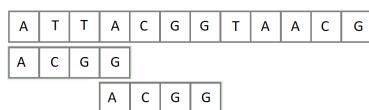
جزوه جلسه ۱۴ ام مورخ ۱۳۹۹/۲/۱۲ درس طراحی و تحلیل الگوریتم تهیه شده توسط امید میرزا جانی. در جهت مستند کردن مطالب درس طراحی و تحلیل الگوریتم این الگوریتم که یکی از معروف ترین الگوریتم ها برای String Pattern Matching است، در سال ۱۹۷۰ توسط سه نفر به نام های Pratt ، Morris و Knuth پیدا شد. این الگوریتم از 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

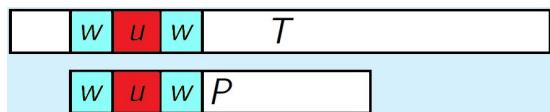
وقتی از Matching Pattern blue حرف میزنیم، اولین و ساده ترین ایده که به ذهن می رسد این است که به ترتیب از اول تا آخر متن اصلی را پیمایش کنیم و ببینیم که آیا در جایی با الگو، برابر میشود یا خیر. اما ایده ایده برای متن ها و یا الگوهای طولانی بسیار زمان گیر است و اصلاً به صرفه نیست؛ زیرا از اردر است. |pattern|*|text|

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.



Border : بردر یک رشته به نام S پیشوندی از آن رشته است، که در انتهای 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 هستند. برای حل مسئله 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.



Function Prefix ۲.۱۴

یک آرایه ای تعریف میکنم که نمایه امّا 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 انتهای، 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.

```

1 ComputePrefixFunction(P)
2 {
3     s = array of integers of length |P|
4     s[0] = 0, border = 0
5     for i from 1 to |P| - 1:
6         while (border > 0) and (P[i] != P[border]):
7             border = s[border - 1]
8             if P[i] == P[border]:
9                 border = border + 1
10            else:
11                border = 0
12            s[i] = border
13    return s
14 }
```

نمونه کد ۵: محاسبه Prefix Function

۳.۱۴ الگوریتم نهای

برای محاسبه Pattern Matching ایده این است که رشته الگو و متن اصلی را به هم بجستنیم و Preffix 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 Funcition در آن به اندازه طول الگو است، به عنوان جواب در خروجی باید نمایش داده شود.

```

1 FindAllOccurrences(P, T)
2     S = P + '$' + T
3     s = ComputePrefixFunction(S)
4     result = empty list
5     for i from |P| + 1 to |S| - 1:
6         if s[i] == |P|:
7             result.append(i - 2|P|)
8     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 suffix array و kmp efficent

صدراء 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 ، برآن 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 جزوء 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 جزوء‌ای که تحويل می‌دهید، حذف کنید.

۱.۱۵ معیار ارزیابی جزوء

معیارهای مورد استفاده برای ارزشیابی کیفیت جزوء 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 راجبه الگوریتم kmp صحبت میشود . و همچنین نحوه محاسبه prefix function میپردازیم و سپس سراغ suffix array رفته و نحوه محاسبه 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 میتوانیم بعضی از character ها رو skip کنیم.

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

اطلاعات بیشتر درباره suffix و prefix به این سایت مراجعه کنید . [prefixVsSuffix]

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

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

برای اطلاعات بیشتر درباره kmp به این لینک مراجعه شود . [KMP]

* برای توضیحات بیشتر و کامل تر به صفحه ۸۶ ، string3.pdf مراجعه کنید

برای مشاهده شبہ کد kmp^{\dagger} می‌توانید از مثال زیر در الگوریتم ۲۲ 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

: prefix function کوییز حل شده واسه ی

```

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

پیدا کردن suffix array از روی suffix tree :

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

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

kmp^{\dagger}

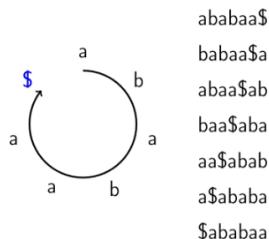
: Suffix array

برای یک string اینگونه بدست می آید. ابتدا همه suffix ها را پیدا کرده و آنها را بر اساس حروف اول شان مرتب کرده. و در آرایه، index، شروع suffix متناظر را را میزاریم. در حالت عادی این کار $O(n^2)$ طول میکشد.

اگه به آخر string ، \$ اضافه کنیم و جایگشت دوری اش را بنویسیم. به cyclic shift میرسیم. حال اگر این 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:



شكل ۱.۱۵ : cyclic shift length of ۶

[‡] برای توضیحات تکمیل تر به صفحه ۴۰ اسلاید ۱ string مراجعه شود

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

شکل ۲.۱۵ : Suffix array

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

ابتدا character ها sort داده شده را میکنیم.

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

تازمانی که ترتیب cyclic shift های به طول $2L$ را با استفاده از قبلي 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 sort به طول ۱ ، sort کرد و الى آخر....

وقتی $L > \text{length}(\text{text})$ شد cyclic shift سوت شده بود آمده همان Count Sort هست.

برای sort کردن count ها از single character استفاده میکنیم.

abbcaaaabcc count Sort : aaaabbbccc	مثال :
---	--------

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

پیچیدگی زمانی $O(n \log n)$ میشود. هر بار که طول cyclic shift را ۲ برابر میکنیم. برای sort کردن آن ها با پیچیدگی زمانی $O(n^2)$ انجام میدهیم و این کار را هم باز را انجام میدهیم. پس ساختن suffix array ، طول میکشد.

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

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

: sort single character

[§](برای توضیحات بیشتر به مثال صفحه ۵۴ اسلاید ۱ string مراجعه شود)

its our alphabets

Data: S

Result: sorted charecters

order = array of size $|S|$;

count = array of size $|\Sigma|$ index=0;

while $index < |S|-1$ **do**

| count[S[index]] = count[S[index]] + 1;

| index = index + 1;

end

index = 1;

while $index < |\Sigma|-1$ **do**

| count[index] = count[index] + count[index - 1];

| index = index + 1;

end

index = $|S|-1$;

while $index \geq 0$ **do**

| c = S[index];

| count[c] = count[c] - 1;

| order[count[c]] = index;

end

return order;

Algorithm 14: sorting Charecters

: Equivalnce Class

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

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	

شكل ۳.۱۵ class

برای توضیحات تکمیلی به صفحه ۷۶ اسلاید ۴.۱ string مراجعه شود

جلسه ۱۶

آرایه پسوندی بهینه و آرایه LCP

زهرا حسینی - ۱۳۹۹/۱/۲۶

۱.۱۶ دوره مفاهیم آرایه و درخت پسوندی

آرایه پسوندی آرایه‌ای از همه پسوندهای یک رشته است. این آرایه بر اساس نویسه‌های * هر یک از پسوند ها مرتب شده است درخت پسوندی ساختمان داده ایی است که هر یک از یال‌های این درخت با یک پسوند بروچسب شده است. به عنوان مثال آرایه پسوندی رشته / $S = "ababaa\$"$ به صورت زیر است:

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

character*

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

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

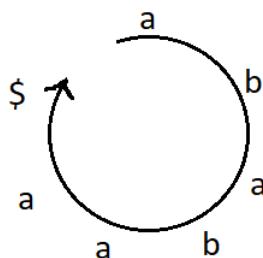
مرتب کردن همه این موارد حافظه زیادی را مصرف میکند. ذخیره کردن آنگاه نیز از لحاظ پیجیدگی زمانی برابر است با: $O(|S|)$

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 i "ababaa\$" رسم شده است:

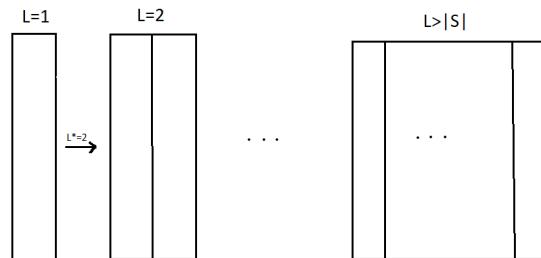


شکل ۱.۱۶: جا به جایی جزئی چرخه ایی [†]

برای درک بهتر دو برابر کردن طول رشته ها به شکل زیر دقت کنید، در هر مرحله اطلاعات جرئی 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 شبکه 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  $|\Sigma|-1$  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=abcdefghijklm
c2-3=cdef
```

Equivalence classes

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

$s=ababbbba$
 $L=2$
 $c_{0-2}=ab$
 $c_{2-2}=ab$
 $c_0=c_2$
 $\text{class}[0]=\text{class}[2]$

حال به محاسبه‌ی آرایه هم ارزی برای رشته‌ی زیر میپردازیم: $s=ababaa\$$

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

برای محاسبه‌ی این آرایه از شبکه 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[order[i]] ≠ S[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 شیفت‌های چرخشی دو برابر شده

مانطور که در ابتدای این بخش گفتیم در هر مرحله طول چرخش را دو برابر می‌کنیم و رشته‌های 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 قسماتی که قبل مرتب شده هستند و نیازی نیستند که دوباره زمان صرف شود برای مرتب کردن آنها. مقدار c'_i را اینگونه تعریف می‌کنیم که شیفت چرخشی که از مکان i شروع شده و به اندازه L برابر طول L مرحله قبل ادامه پیدا می‌کند.

$$\begin{aligned} c_i \\ c'_i \\ c'_i = c_i c_{i+L} \end{aligned}$$

به مثال زیر دقت کنید:

Sort Doubled cyclic Shifts^[۷]

```

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

```

برای محاسبه‌ی مجدد آرایه order از شبه کد زیر استفاده میکنیم و آرایه‌ی 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 |\Sigma|-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 رفته 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 چرهشی 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 حدید حاصل آنها باهم برابر است

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

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 شبکه 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 مراحلی که توضیخ دادیم در کتاب یکدیگر استفاده کنیم تا آرایه مورد نظر حاصل شود. برای این منظور از شبکه کد زیر استفاده میکنیم. حلقه آمده شده در شبکه کد را تا جایی ادامه میدهیم 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

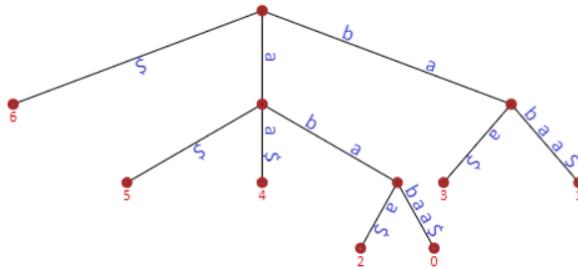
$$O(|S| \log |S| + |\Sigma|)$$

LCP ARRAY ۳.۱۶

قدم دیگری که برای ساخت درخت پسوندی * بهینه باید برداریم ساخت آرایه‌ی طولانی ترین پیشوند مشترک است. برای این منظور آرایه‌ایی به اندازه $|S|-1$ می‌سازیم. خانه‌های آرایه را میزان اشتراک هر پسوند با پسوند بعدی خود پر می‌کنند. این پسوند‌ها به همان ترتیبی هستند که در آرایه پسوندی ** ترتیبیشان ذکر شده است.

۱.۳.۱۶ ایده کلی

با ساخت آرایه پسوندی در قسمت قبل دسترسی راحت‌تر و سریع‌تری به زیررشته‌ها داریم. در درخت پسوندی نیز از اولین زیررشته شروع می‌کنیم و به ریشه‌ی درخت اضافه می‌کنیم برای اضافه کردن زیررشته‌های بعدی باید ابتدا با یال‌های موجود در درخت مقایسه شوند که در صورت اشتراک به آن یال وارد شوند. اگر طول زیررشته‌ها طولانی باشد زمان زیادی برای مقایسه صرف می‌شود. در این قسمت می‌توان از آرایه‌ی LCP استفاده کرد، به این صورت که در این آرایه تعداد اشتراک زیررشته‌ها 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:



شكل 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 baba$  

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 محاسبه‌ی آرایه LCP

برای محاسبه‌ی این آرایه میتوان از الگوریتم‌های ساده‌تری 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 طولانی‌ترین پیشوند مشترک بین S_i و پسوند بعدی آن در آرایه‌ی پسوندی مربوط به رشته باشد. در نتیجه طولانی‌ترین پیشوند مشترک S_i و پسوند بعدی آن حداقل برابر $h-1$ است.

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

Data: S,i,j,equal

Result: lcp

$lcp \leftarrow \max(0,equal)$

```

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

```

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

Data: S,order

Result: class

$pos \leftarrow \text{array of size } |\text{order}|$

```

for  $i$  from 0 to  $|pos|-1$  do
    |  $pos[\text{order}[i]] \leftarrow i$ 
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)

مثال ها و شبه کد های 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 محاذی استفاده شده برای رسم شکل را میتوانید در این قسم مشاهده کنید. [ST].

۱۷ جلسه

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 کیم. از ساختن suffixarray شروع می شود که همه می 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 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 دو suffix ، دو بعدی که مقایسه می شود حداقل تعداد اشتراکاتشان LCP قبلی منهای یک است.

LCPArray: پیشنباز

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 به عنوان i و j و string اصلی هستند. اول LCP را برابر(max(0, equal(0, string بیشتر نشده ، اگر برابر نبودند ۱ . این روند گوییم تا موقعی که کاراکتر بعدی زو از طول string بیشتر نشده ، اگر برابر نبودند ۱ . 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 : پیشناز 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. مثلا 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: پیشناز LCPArray

:LCParay

برای حساب کردن LCParay ابتدا آرایه ای به طول $|S| - 1$ می سازیم که S همان string اصلی است. سپس pos را حساب می کنیم. بعد suffix ما در $[0..|order| - 1]$ است یعنی $[0..|order| - 1]$ یک مکان است برای شروع suffix. بعد pos آن می شود همان مکانش در حال nextsuffix suffixarray. را حساب می کنیم که بعدی آن است و LCP این دو را حساب می کنیم و در $[LCParay[orderindex..LCP[orderindex + 1..|S| - 1]]]$ قرار می دهیم همچنین $suffix = suffix + 1$ می کنیم. حال دوباره به اول for برمی گردیم. اگر $suffix = 0$ و یک while داریم که این در آن محاسبه می شود که $OISI_1$ است. چرا $OISI_2$ نمی شود؟ علت آن این است که 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 $2|S|$ نیست.

۲.۱۷

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 ۱۷ : ComputeLCPArray

: Building suffixtree

تا اینجا suffixarray و LCParray را حساب کردیم. حال اول به root را dollarsign اضافه می کنیم و LCP=۰ است. پس به ارتفاع ۰ (که همان root است) می رویم و suffix بعدی را add می کنیم. و عدد array suffix 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 شاخصه ای که به آن نیاز داریم linkparent است که در node ذخیره می کنیم سپس بجه 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

موقعی که عمق درخت بزرگتر از LCP است به بالا می رویم اگر عمق برابر LCP قبلى بود لازم نیست node جدید بسازیم و گرنه 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

```

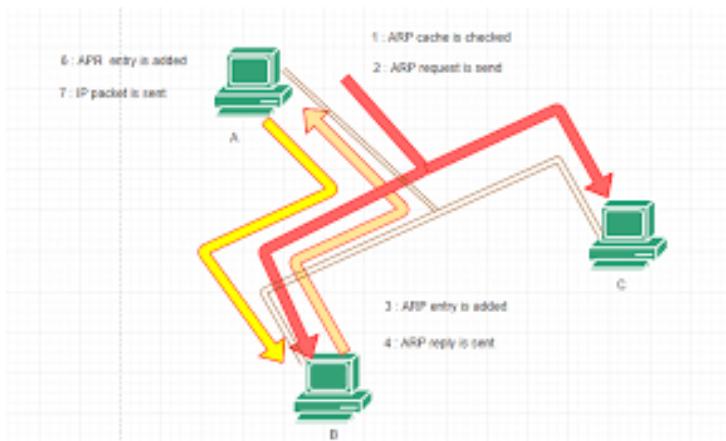
شکل ۵.۱۷ Buildingsuffixtree

:Suffixtreeorder

ساختن suffixtree از روی suffixarray در زمان خطی یعنی $O(|S|)$ انجام می شود و ساختن suffixtree از ابتدا $O(|S|^2 \log |S|)$ است زیرا زمان آن برابر با $|S| + |S| \log |S|$ است.

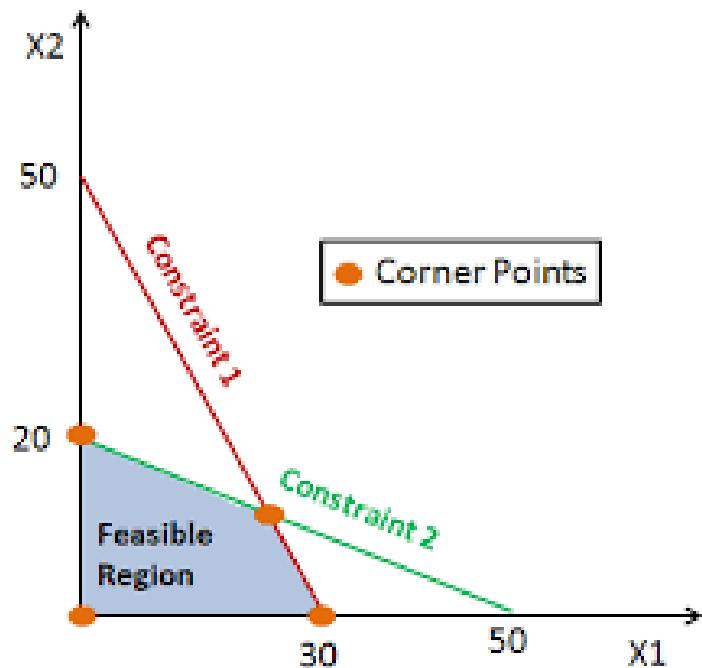
کلیت جلسه بعد:

۶.۱۷ Flowinnetworks : شروع الگوریتم های پیشرفته با مباحث زیر:



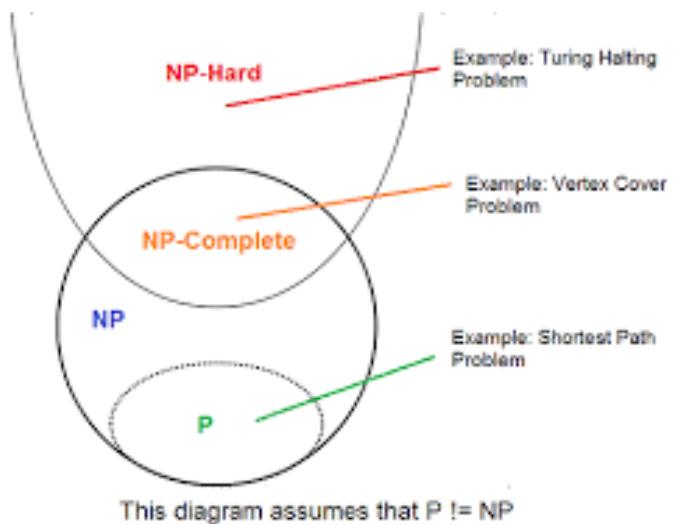
شکل ۶.۱۷ Flowinnetworks :

۷.۱۷ Linearprogramming (۲)



Linearprogramming : ۷.۱۷ شکل

۸.۱۴ NPcompleteproblems (۳



شکل ۱۷. NPcompleteproblems

CopingwithNPcompleteness (۴)

Streamingalgorithms(optional) (۵)

جلسه ۱۸

جريان در گراف

محمد مصطفی رستم خانی - ۱۳۹۹/۲/۲

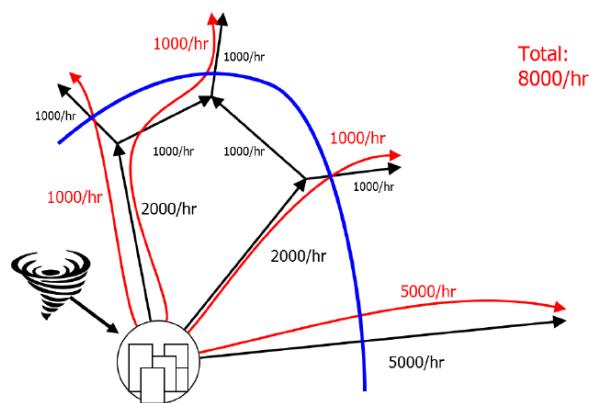
جزوه جلسه ام ۱۱۸ مورخ ۱۳۹۹/۲/۲ درس طراحی و تحلیل الگوریتم تهیه شده توسط محمد مصطفی رستم خانی. در جهت مستند کردن مطالب درس طراحی و تحلیل الگوریتم، برآن شدیم که از دانشجویان جهت مکتوب کردن مطالب کمک بگیریم. هر دانشجو می‌تواند برای مکتوب کردن یک جلسه داطلب شده و با توجه به کیفیت جزو از لحاظ کامل بودن مطالب، 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. مطالibi که در ادامه آمده فقط جنبه راهنمایی شیوه استفاده از لاتک می‌باشد. خواهشمند است این پاراگراف و مطالب بعدی را از نسخه جزوی که تحويل می‌دهید، حذف کنید.

۱.۱۸ جريان 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 هر کدام 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. جريان در شبکه به شما اين اجازه را می دهد که بتوانید بفهمید که

می توانید این کار را بهتر انجام دهید یا خیر.



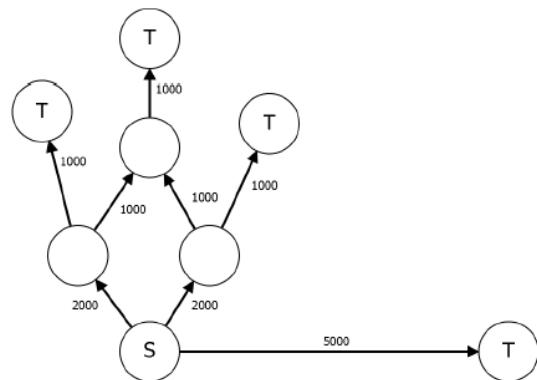
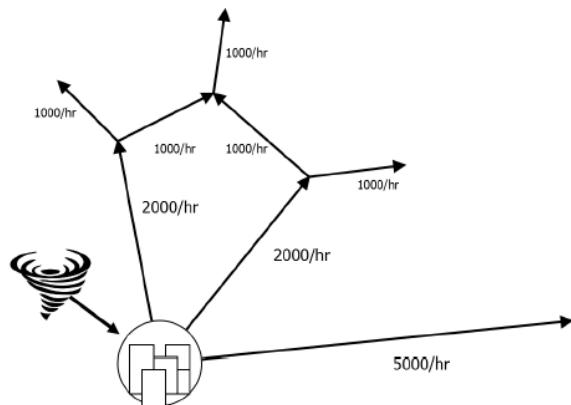
شکل ۱.۱۸ : network

network ۲.۱۸

• تعریف:

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

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



شکل ۲.۱۸

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 به مانند e داریم: (rate limitation)

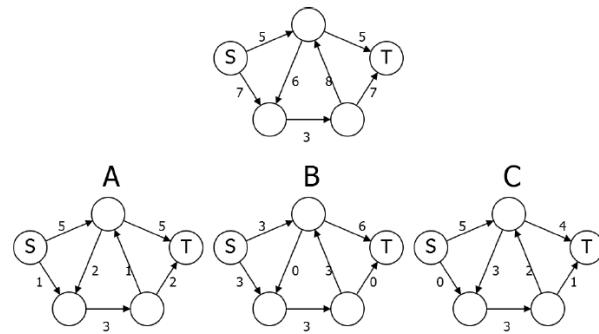
$$0 \leq f_e \leq c_e \quad (1.18)$$

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

$$\sum_{v \text{ into } (e)} f_e = \sum_{v \text{ of out } (e)} f_e \quad (2.18)$$

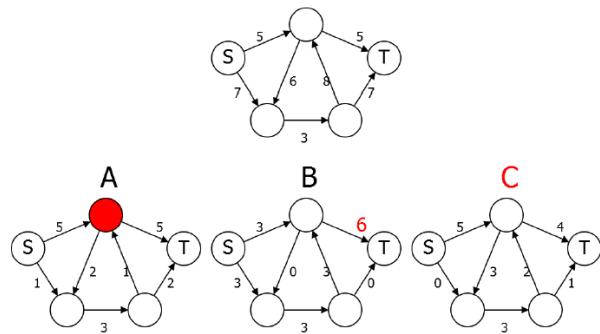
(conservation of flow) : یعنی باید مجموع جریان های ورودی و خروجی به هر راس با هم برابر باشند.

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 همه‌ی شروط لازم برای یک جریان معتبر را رعایت کرده است.



شکل ۴.۱۸ example flow :

- کاربرد هایی از Flow Max :

- ترافیک در شبکه‌ی حمل و نقل شهری
- جریان ورودی و خروجی در خطوط نیرو
- جریان در لوله‌های آب
- شبکه‌ی اینترنت

: (flow size)

تعریف: برای یک جریان f اندازه جریان به صورت زیر تعریف می‌شود:

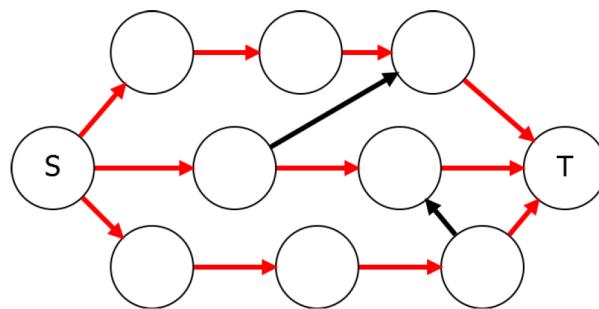
$$|f| = \sum_{\text{source } a \text{ of out } (e)} f_e - \sum_{\text{source } a \text{ into } (e)} f_e \quad (3.18)$$

$$|f| = \sum_{\text{sink } a \text{ into } (e)} f_e - \sum_{\text{sink } a \text{ of out } (e)} f_e \quad (4.18)$$

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 DFS یا BFS استفاده کرد.

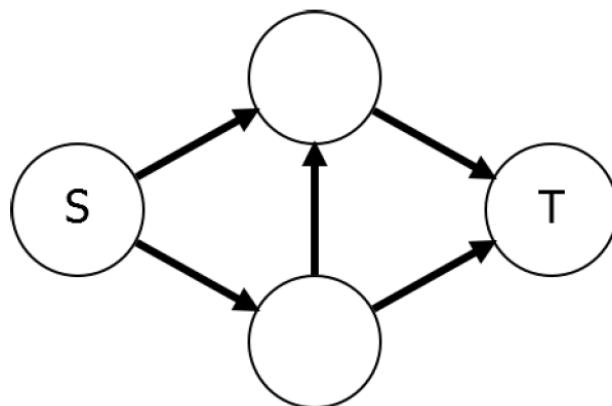
[All capacities are 1]



شکل ۵.۱۸ graph

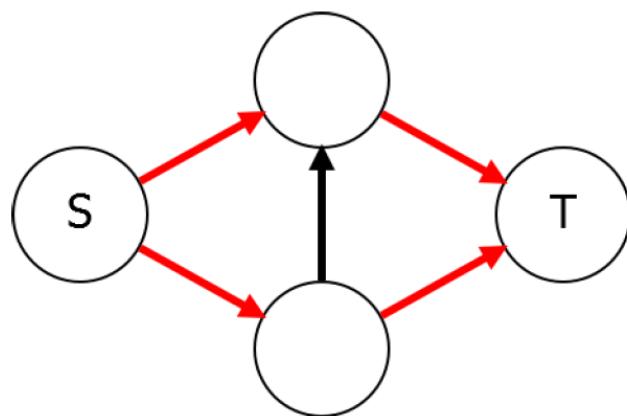
برای مثال در گراف بالا flow max برابر است با ۳.

برای مثالی دیگر گراف زیر را در نظر بگیرید.



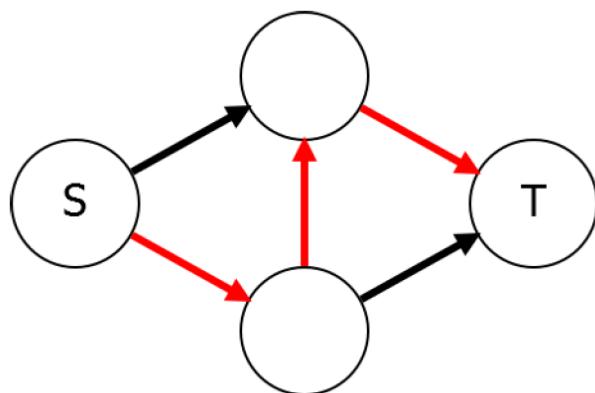
شکل ۶.۱۸ graph

در این گراف بیشینه‌ی جریان برابر است با ۲.



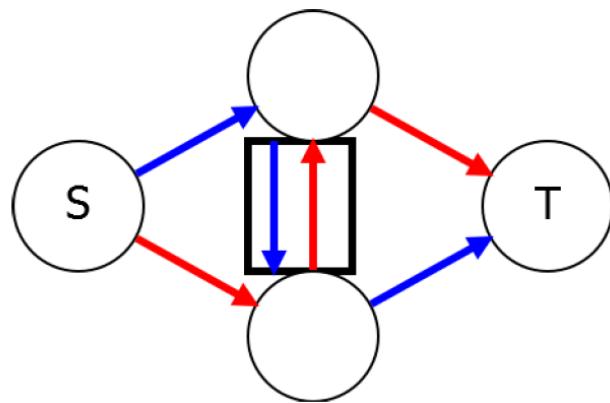
شکل ۷.۱۸

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



شکل ۸.۱۸

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



شکل ۹.۱۸

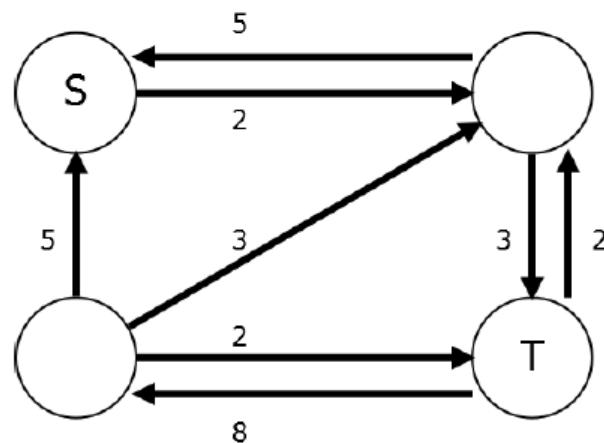
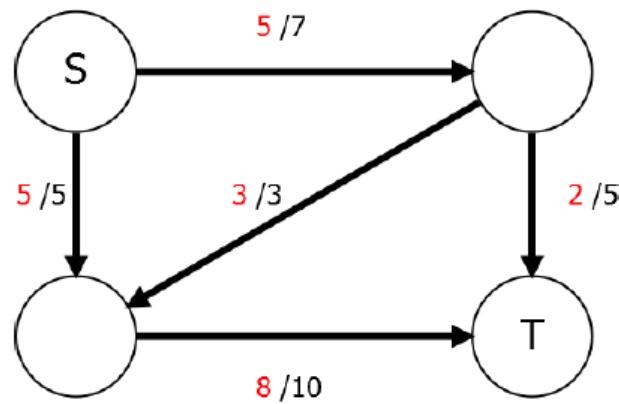
با استفاده از این گراف می توان جریان وسط را خنثی کرد.

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 $C_e - f_e$ مگر در حالتی که $f_e = C_e$

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

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



شکل ۱۰.۱۸ residual network : ۱۰.۱۸

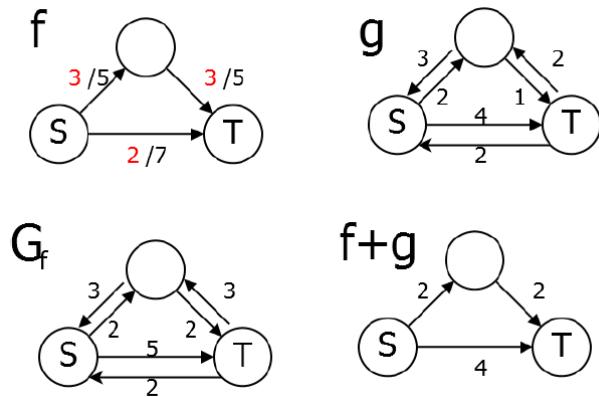
۴.۱۸ جریان باقیمانده (residual flow)

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

۱. f_e به g_e اضافه می‌شود.

۲. g_e از f_e کم می‌شود.

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.



شكل ۱۱.۱۸ residual network : ۱۱.۱۸

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

۱. یک جریان روی گراف اصلی است.

$$|f+g|=|f|+|g| \quad .2$$

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.

maxflow: and Mincut ۵.۱۸

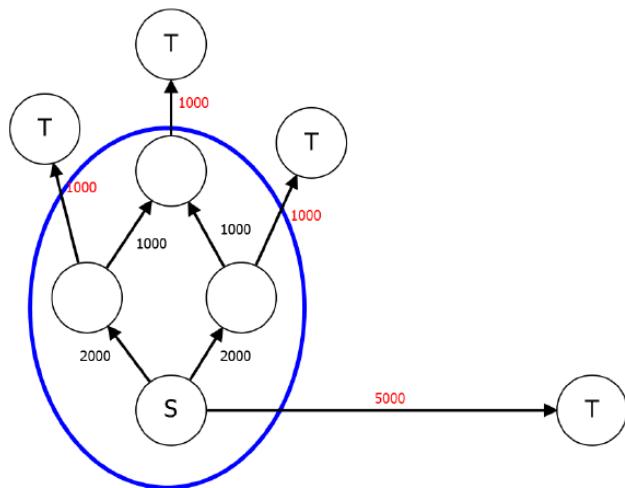
برای پیدا کردن maxflow ما به راهی برای تشخیص اینکه جریان به دست آمده بیشینه است داریم. برای این منظور از تکنیک هایی برای محدود کردن اندازه ی maxflow استفاده می کنیم.

(cut): برش:

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

$$|C| = \sum_{C \text{ of out } e} C_e \quad (5.18)$$

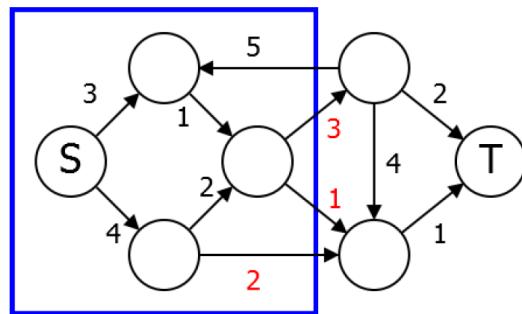
برای مثال اندازه ای cut در شکل زیر برابر با ۸۰۰۰ است.



شکل ۱۲.۱۸ residual network :

مثال:

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



شکل ۱۳.۱۸ residual network : ۱۳.۱۸

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 $|f|$ با اندازه $|C|$ برابر است. حالت خاص:

اگر $\text{maxflow} = 0$ باشد آنگاه $(\text{maxflow}=0)$

۱. هیچ مسیری از source به sink وجود ندارد.

۲. اگر $|C|=0$ باشد آنگاه source قابل دسترس باشد.