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

List of Algorithms

فهرست تصاویر

جلسه ۲

Paths in Graphs

نگار زین‌العابدین - ۱۳۹۸/۱۱/۲۰

۱.۲ دید کلی

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

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

۱. Shortest path

۲. Minimum spanning trees

۳. Advances shortest path

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

۱. BFS (Breadth-First-Search)

۲. Dijkstra

۳. Bellmanford

۲.۲ Shortest path

همان‌طور 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 دیگرِ گراف، از این الگوریتم استفاده می‌کنیم.

- Dijkstra :

اگر بر روی راس‌های گرافمان، وزن داشته‌باشیم، از این الگوریتم استفاده می‌کنیم. این الگوریتم نیز مانند BFS کوتاه‌ترین مسیر را، از راسی به تمام راس‌های دیگرِ گرافمان، حساب می‌کند. البته باید به این نکته نیز توجه کرد؛ اگر در گرافمان، وزن منفی داشتیم، استفاده‌کردن از این الگوریتم، مجاز نیست و جواب به دست‌آمده صحیح نمی‌باشد.

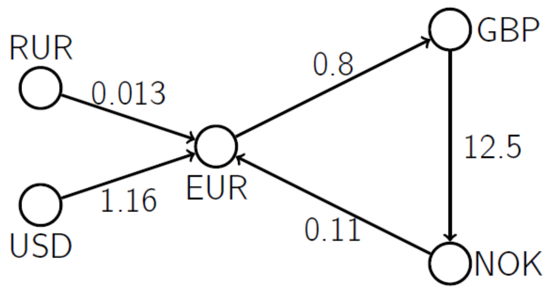
- Bellmanford :

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

۳.۲ دور منفی چسیت ؟

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

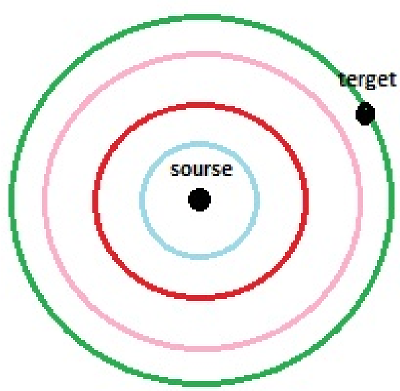
- یکی از کاربردهایی که می‌توان برای دورِ منفی نام برد، بدین شرح است که فرض کنید می‌خواهیم نرخ‌های مختلفی از پول‌ها را به گونه‌ای به یک‌دیگر تبدیل کنیم که بیش‌ترین سود نصیبمان یشود. جزئیات این مسئله و ارتباط آن با دورِ منفی در جلسات آینده، شرح داده خواهدشد.



شکل ۱.۲: currency exchange

۴.۲ Dijkstra / BFS

اگر به زبان ساده‌ای به شرح این دو الگوریتم بپردازیم، می‌توانیم بگوییم که این دو الگوریتم، در ابتدا، از نقطه مبدا شروع و تمام نقاطی که فاصه‌ی 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.



شکل ۲.۲: s-t shape

- شبه‌کد* `BFS`

pseudocode*

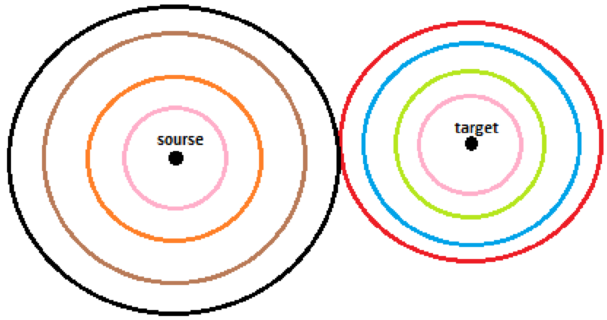
```
Input: Graph,Source
Ouput: BFS on Graph
initialization;
for All e in Edges do
|  dist[e]=inf
end
dist[Source]=0;
Q <- Source : queue containing just Source
while Q is not empty do
    u <- Dequeue(Q)
    for All (u,v) in Edges do
        if dist[v] = inf then
            Enqueue(Q,v)
           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

Algorithm 1: BFS on graph

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

۵.۲ Bidirectional dijkstr / Bidirectional BFS

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



شکل ۳.۲: bidirectional shape

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

جلسه ۴

Dijkstra And Bellman-Ford Algorithms

پریسا 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 => هر تکه از یک مسیر بهینه خود حتما بهینه است. [۲۳]

۲.۴ Dijkstra's Algorithm

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

۱) برای همه ی گره‌ها یک متغیر فاصله در نظر می‌گیریم و مقدار اولیه ی آن را بی نهایت می‌گذاریم .

۲) انتخاب گره ی شروع و قرار دادن صفر برای مقدار فاصله ی آن

۳) محاسبه ی فاصله ، از گره‌ای که در آن قرار داریم تا همه‌ی همسایگان آن و قرار دادن فاصله‌ی همسایه‌ها با مقادیر به دست آمده؛ اگر مقدار محاسبه شده از مقداری که اکنون دارند، کمتر باشد .

۴) از بین تمام همسایه‌های نودهایی که بازدید کرده ایم ، نودی که کمترین فاصله را دارد انتخاب می‌کنیم و مرحله ی سوم را دوباره اجرا می کنیم . نکته : نودی که قبلا بازدید شده‌است را هرگز دوباره بررسی نمی کنیم .

۵) این الگوریتم زمانی تمام می‌شود که تمام همسایه ها مورد بررسی قرار گیرند . اگر بعد از پایان بررسی ، نودی وجود داشت که فاصله ی آن بی نهایت بود ؛ به این معنا است که از نود شروع ، هیچ مسیری به آن نود وجود ندارد .

گرفته‌شده‌است از [۲]

۳.۴ Order of Dijkstra Algorithm

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

گرفته‌شده‌است از [۲۳]

۴.۴ مزایای Dijkstra Algorithm :

۱) کمترین زمان برای رسیدن به خانه از محل کار را به دست آورد .

۲) پیدا کردن سریع ترین مسیر از محل کار به خانه .

۳) کوتاه ترین مسیر را هم در گراف جهت دار و هم بدون جهت پیدا کرد .

گرفته‌شده‌است از [۲۳]

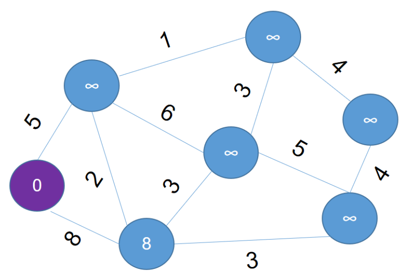
۵.۴ معایب **Dijkstra Algorithm** :

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

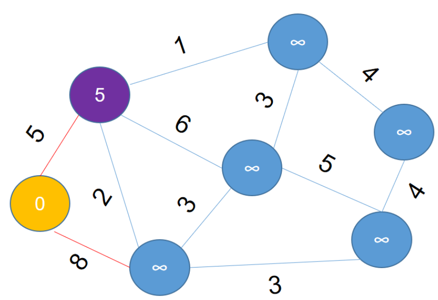
۲) اگر گراف دارای یال منفی باشد ، این الگوریتم روی آن کار نمی کند .

گرفته‌شده‌است از [۲۳]

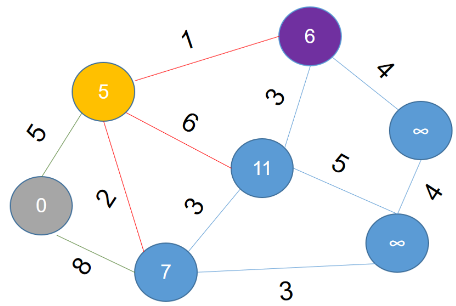
۶.۴ مثال :



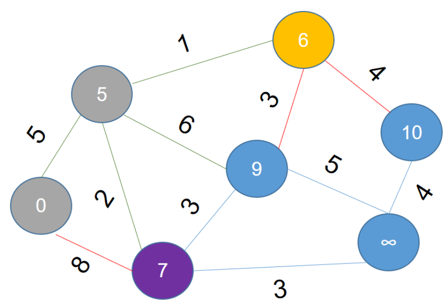
شکل ۱.۴: step first
[۱]



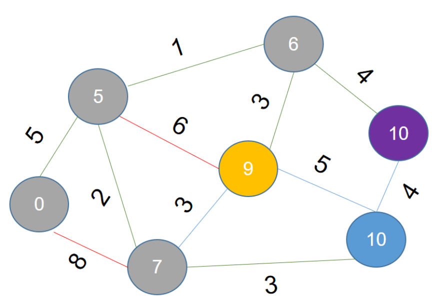
شکل ۲.۴: step second
[۱]



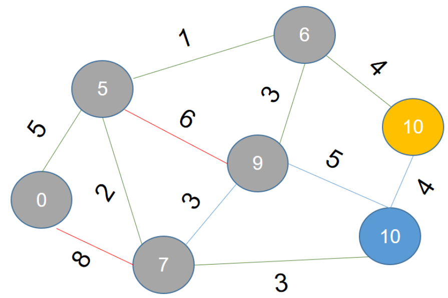
شکل ۳.۴: step third
[۱]



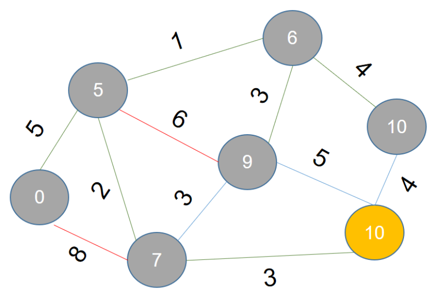
شکل ۴.۴: step fourth
[۱]



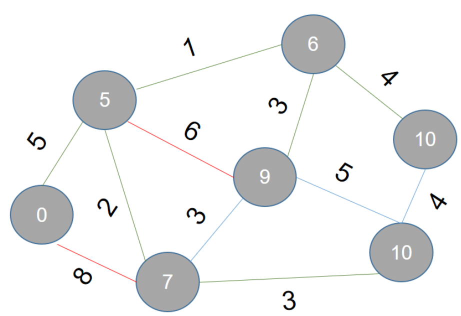
شکل ۵.۴: step fifth
[۱]



شکل ۶.۴: step sixth
[۱]



شکل ۷.۴: step seventh
[۱]



شکل ۸.۴: step eighth
[۱]

۷.۴ شبه کد Dijkstra Algorithm :

Data: $Relax((u, v) \in E)$

Result: relax the edges

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

  dist[v]←dist[u]+w(u,v) ;

  Prev[v] ← u ;

else

end

Algorithm 2: Relax Method

Data: Naive(G,S)

Result: Find shortest path

for $allu \in V$ **do**

  dist[u] ← ∞ ;

  prev[u] ← nil ;

end

dist[S] ← 0 ;

while *at least one dist changes* **do**

  relax all the edges ;

end

Algorithm 3: Naive Algorithm Of Dijkstra

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

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

Data: Dijkstra(G,S)
Result: Find Shortest Path

```
for all u ∈ V do
    dist[u] ← ∞ ;
    prev[u] ← nil ;
end
dist[S] ← 0 ;
H ← MakeQueue(V) dist-values as keys ;
while H is not empty do
    u ← ExtractMin(H) ;
    for all(u, v) ∈ 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]) ;
        else
        end
    end
end
```

Algorithm 4: Dijkstra Algorithm

گرفته‌شده‌است از [۲۳]

۸.۴ اثبات درست بودن الگوریتم Dijkstra

وقتی که راس u را ExtractMin می‌کنیم، 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 dist[u] کمتر شود، چون اگر یکی از راسهای بعدی را v درنظر بگیریم، dist[u] باید بیشتر از dist[v]+edge(v,u) شود تا آپدیت شود. درصورتی که dist[u]<=dist[v] است پس حتما dist[v]+edge(v,u)>=dist[u] است (در صورتی که یال منفی نداشته باشیم) و به همین دلیل dist[u] دیگر آپدیت نمی‌شود. [۲۳]

۹.۴ Bellman–Ford algorithm

برای پیدا کردن کوتاه ترین مسیر در بین نودهای یک گراف می‌توان از ان استفاده کرد .
در Bellman Ford ، مشابه به الگوریتم ساده‌ی Dijkstra (algorithm 3) عمل می کنیم .
نحوه ی عملکرد الگوریتم :

۱) مثل Dijkstra (section 4.2) برای همه‌ی نودها یک متغیر فاصله در نظر می‌گیریم و مقدار اولیه‌ی آن را بی‌نهایت می‌گذاریم .

۲) انتخاب گره‌ی شروع و قرار دادن صفر برای مقدار فاصله‌ی آن

۳) یال‌هایی که به آن نود مربوط است را ویزیت می‌کنیم و آن‌ها را ریلکس می‌کنیم . (این مورد الزامی نیست و باعث سریع‌تر شدن الگوریتم می‌شود.)

۴) یکی از همسایه‌های آن نودی که در آن قرار داریم را انتخاب می‌کنیم و مرحله‌ی دوم را برای آن اجرا می‌کنیم .لازم به ذکر است که از هیچ یالی دوبار حرکت نمی‌کنیم . (این مورد الزامی نیست و باعث سریع‌تر شدن الگوریتم می‌شود.)

۵) باید همه‌ی یال‌ها را در هر دور ریلکس کنیم .

۶) این الگوریتم زمانی تمام می‌شود که مراحل یک تا پنج را به اندازه‌ی یکی کم‌تر از تعداد نودها تکرار شود .

گرفته‌شده‌است از [۳] [۴]

۱۰.۴ Order of Bellman-Ford Algorithm

اوردر این الگوریتم برابر است با : O(|V||E|) [۲۳]

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 implement infinite arbitrage (section 4.18) را عملی کرد .

گرفته‌شده‌است از [۲۳]

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.

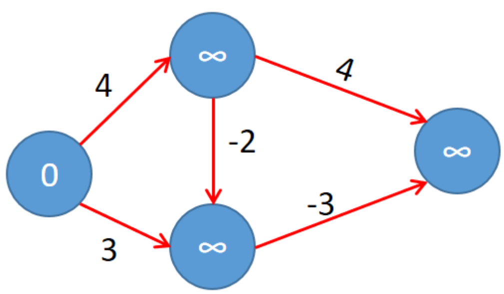
۳) اگر گراف دارای دور منفی باشد ، این الگوریتم کارنمی‌کند .

۴) الگوریتم Bellman-Ford (section 4.9) نسبت به الگوریتم Dijkstra (section 4.2) کندتر است .

گرفته‌شده‌است از [۳] [۶] [۲۳]

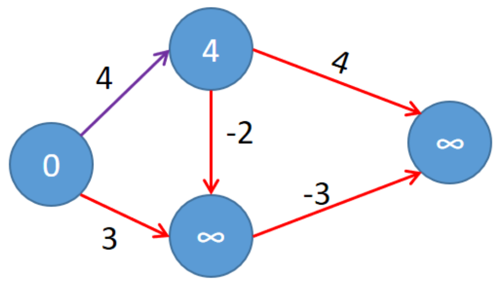
۱۳.۴ مثال :

figure reference



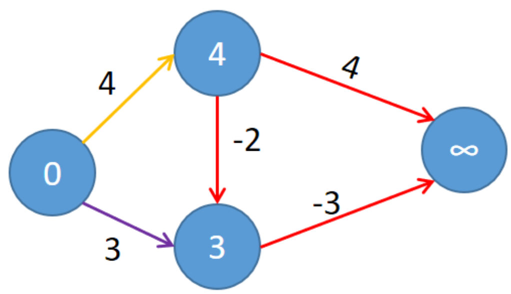
شکل ۹.۴: step ۱

figure reference



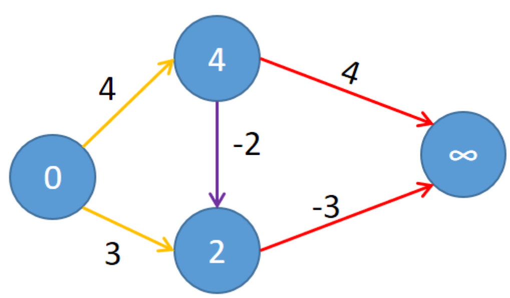
شکل ۱۰.۴: step ۲

figure reference



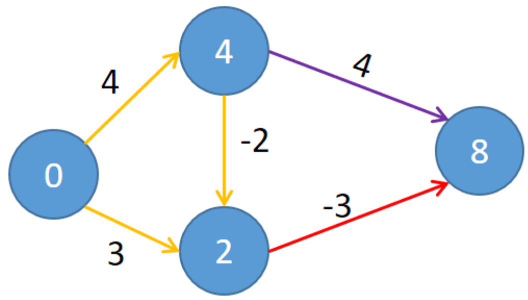
شکل ۱۱.۴: step ۳

figure reference



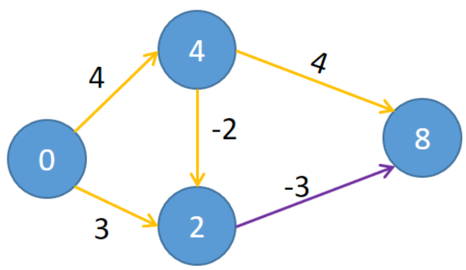
شکل ۱۲.۴: step ۴

figure reference



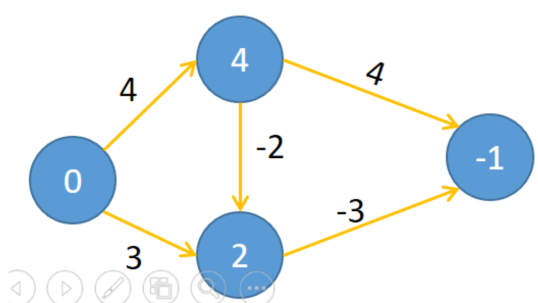
شکل ۱۳.۴: step ۵

figure reference



شکل ۱۴.۴: step ۶

figure reference



شکل ۱۵.۴: step ۷

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

۱۴.۴ شبه کد :

```
Data: BellmanFord(G,S)
Result: Find Shortest Path
no negative weight cycles in G
for allu ∈ V do
    dist[u] ← ∞ ;
    prev[u] ← nil ;
end
dist[S] ← 0;
repeat |V|-1 times :
for all(u, v) ∈ E do
    Relax(u,v) ;
end
```

Algorithm 5: Bellman-Ford Algorithm

گرفته‌شده‌است از [۲۳]

۱۵.۴ اثبات درست بودن الگوریتم Bellman-Ford

بعد از k بار relaxation (algorithm 2) ، همه ی کوتاه‌ترین فاصله‌ها از راس شروع که حداکثر شامل k یال هستند؛ مشخص شده اند.

با استفاده از استقرای ریاضی:

۱. اگر k=0 فاصله همه راس‌ها از راس شروع بی‌نهایت است، به غیر از خود راس شروع که dist[S]=0 .

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

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

گرفته‌شده‌است از [۲۳]

۱۶.۴ دور منفی در Bellman-Ford :

۱) Bellman-Ford Algorithm (section 4.9) را به اندازه ی تعداد نودها اجرا می‌کنیم .نودهایی که در دور آخر ریلکس می‌شوند را در یک کیو یا لیست ذخیره می‌کنیم.

۲) از $v \leftarrow x$ شروع کنیم، مسیر $x \leftarrow$ prev[x] را به اندازه ی نود ها تکرار می‌کنیم .به طور قطع در چرخه خواهد بود .

۳) $x \leftarrow y$ را ذخیره می کنیم و $x \leftarrow$prev[x] را ادامه دهید تا به $y = x$ برسیم .

گرفته‌شده است از [۲۳]

۱۷.۴ اثبات درست بودن الگوریتم دور منفی در Bellman-Ford

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

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

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

dist[b] <= dist[a] + w(a,b)

dist[c] <= dist[b] + w(b,c)

dist[a] <= dist[c] + w(c,a)

با جمع کردن این سه معادله: w(a,b) + w(b,c) + w(c,a) >= 0

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

یالها باید relax شود. [۲۳]

۱۸.۴ **Infinite Arbitrage :**

۱) به اندازه ی تعداد نودها Bellman-Ford Algorithm (section 4.9) را انجام می‌دهیم . نودهایی که در دور آخر ریلکس می‌شوند را در لیست یا کیو ذخیره می‌کنیم .

۲) برای نودهایی که ذخیره کردیم ، الگوریتم بی اف اس (BFS) [۵] 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) [۵] نودهایی که قبلا ویزیت شده‌اند را کاری نداریم .
- از نودی که در دور آخر ریلکس شده است، می‌توان دور منفی را یافت و از دور منفی برای به دست آوردن infinite arbitrage استفاده کرد.

گرفته‌شده است از [۲۳]

۱۹.۴ اثبات درست بودن الگوریتم Infinite Arbitrage

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

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

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

اثبات سمت => 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 ی u از s می‌تواند کمتروکم‌تر شود، پس در بعضی از دفعات بعدی اجرای بلمن-فورد مثلا k>=V فاصله‌اش تغییر می‌کند. اگر

راسی در دفعه i ام از relaxation (algorithm 2) یال‌ها فاصله‌اش تغییر نکند، همه یال‌هایی که از آن راس آغاز می‌شنود نیز در دفعه i+1 ام relax نمی‌شوند (راس ابتداییشان تغییری نکرده که راس انتهاییشان را تغییر بدهند.) پس برای اینکه راسی فاصله‌اش تغییر کند، باید راسی دیگر در مسیر بین آن و راس شروع قبلا تغییر کرده باشد. پس برای اینکه dist[u] در $k>=V$ 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. [۲۳]

۲۰.۴ سایت‌های دیگر برای مراجعه :

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

جلسه ۵

Dijkstra و Bellman Ford

یاسمن لطف‌اللهی - ۱۳۹۸/۱۱/۲۹

جزوه جلسه ۵ام مورخ ۱۳۹۸/۱۱/۲۹درس 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 جزییات این دو الگوریتم خواهیم پرداخت.

۱.۵ شبه‌کد Dijkstra

در ابتدا، مقدار dist را برای تمام رأس‌های گراف به جز رأس مبدا $+\infty$، و مقدار prev را برای تمام رأس‌ها null در نظر می‌گیریم. سپس از تمام رأس‌ها، یک `Priority Queue` می‌سازیم، به‌طوری که الویت هر رأس، مقدار dist آن باشد. رأسی را که کمترین dist را دارد، از Priority Queue خارج می‌کنیم و یال های 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 شبه‌کد 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
```

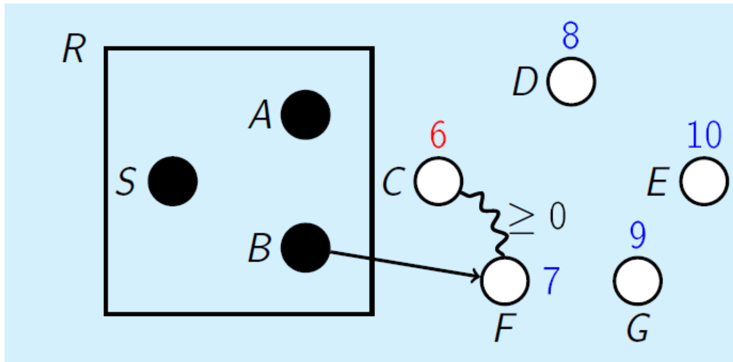
Algorithm 7: Finding Shortest Path

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

۳.۵ اثبات درستی الگوریتم Dijkstra

مثال زیر را در نظر بگیرید:

figure reference



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

طبق الگوریتم، وقتی رأسی از طریق ExtractMin انتخاب می‌شود، به این معناست که کمترین فاصله بین آن رأس و رأس مبدا برابر dist آن رأس است. پس در این مثال، رأس C انتخاب می‌شود و نتیجه گرفته می‌شود که کمترین فاصله بین C و S برابر ۶ است. فرض می‌کنیم که این عبارت درست نیست و کمترین فاصله بین این دو رأس از ۶ کمتر است. بنابراین رأسی مانند F وجود دارد که در کوتاه‌ترین مسیر بین 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

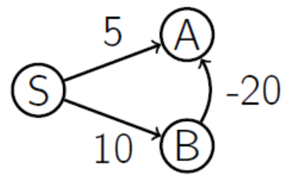
طبق شبه‌کد ۶ ، پیچیدگی زمانی این الگوریتم برابر است با مجموع پیچیدگی‌های زمانی این سه قسمت:

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

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

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

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



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

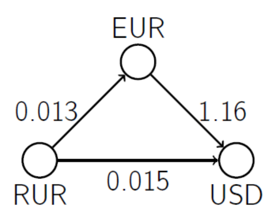
در این مثال، رأس A به رأس S نزدیک‌تر است. اما کوتاه‌ترین راه به S ، از رأس B که از S دورتر است، می‌گذرد.

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

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

۷.۵ یک مثال از کاربرد الگوریتم Bellman Ford : تبدیل ارز

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



شکل ۳.۵: گراف تبدیل ارز

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

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

شکل ۴.۵: تبدیل به لگاریتم

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

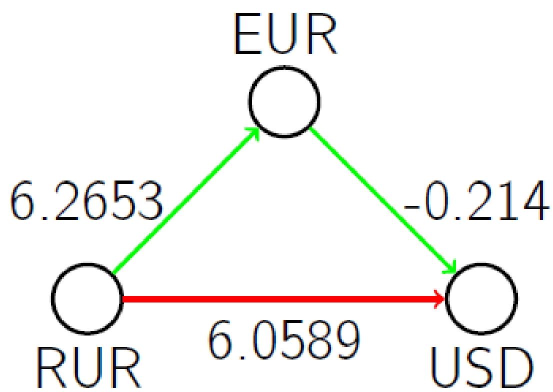
$$\sum_{j=1}^{k} \log(r_{e_j}) \to \max \Leftrightarrow -\sum_{j=1}^{k} \log(r_{e_j}) \to \min$$

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

شکل ۵.۵: قرینه کردن لگاریتم‌ها

figure reference

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



شکل ۶.۵: گراف تبدیل ارز

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

۸.۵ شبه‌کد Bellman Ford

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

```
dist[ ] ← 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 Bellman Ford: O(|V||E|)

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

جلسه ۶

دور منفی در گراف و دایجسترا دوطرفه

ملیکا نوبختیان - ۱۳۹۸/۱۲/۴

۱.۶ قضیه

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

۲.۶ اثبات

اگر در گراف هیچ دور منفی وجود نداشته باشد،همه کوتاه ترین مسیرها از S حداکثر دارای |V|-1 یال خواهند بود (هر مسیری که دارای تعداد بیشتر یا مساوی V یال باشند،منفی نیستند و می توانند از کوتاه‌ترین 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:

$$a \to b \to c \to a$$

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

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

۳.۶ Finding Negative Cycle

برای پیداکردن 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 `x<-v` 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:

```
public bool HasNegativeCycle(Graph G, int Start, long N)
{
    long[] dist = new long[N];
    dist[Start] = 0;
    for (int i = 0; i < N - 1; i++)
    {
        foreach (edge e in Graph.edges)
            relax(e);
    }
    foreach (edge e in Graph.edges)
    {
        if (relax(e))
            return true;
    }
    return false;
}
```

نمونه کد ۱: تابع تشخیص وجود دور منفی درگراف

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:

```
List<long> FindNegCycle(int v,long[] Prev,int N)
{
    long x = v;
    for (int i = 0; i < N; i++)
        x = Prev[x];
    List<long> cycle = new List<long>();
    long y = Prev[x];
    while (y != x)
    {
        cycle.Add(y);
        y = Prev[y];
    }
    return cycle;
}
```

نمونه کد ۲: تابع پیداکردن دور منفی درگراف

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

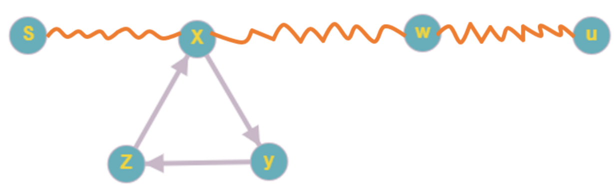
۴.۶ Infinite Arbitrage

۱.۴.۶ قضیه

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

۲.۴.۶ اثبات

اگر `dist[w]` در `|V|` امین تکرار از بلمن فورد کاهش یافته باشد،از دور منفی داخل گراف قابل دسترسی است.[۲۰]



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 حاکثر `V-1` یال 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]` در `i` امین تکرار کاهش نیابد،پس یال `(x,y)` در `i+1` امین تکرار هم ریلکس نخواهد شد.تنها گره هایی که گره های ریلکس شده در تکرار قبلی قابل دسترسی هستند می توانند ریلکس شوند. اگر `dist[u]` اگر `dist[u]` در یک تکرار `k>=V` کاهش یابد، `u` از گره هایی که `V` امین تکرار ریلکس شده اند قابل دسترسی است.

۳.۴.۶ Detect Infinite Arbitrage

برای پیداکردن 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 متد زیر گره هایی که دارای Infinite Arbitrage هستند را پیدا می کنیم:

```
public List<long> DetectInfiniteArbitrage(Graph G, int Start, long N)
{
    long[] dist = new long[N];
    dist[Start] = 0;
    Queue<long> relaxed = new Queue<long>();
    List<long> Arbitrage = new List<long>();
    for (int i = 0; i < N - 1; i++)
        foreach (edge e in Graph.edges)
            relax(e);
    foreach (edge e in Graph.edges)
        if (relax(e))
            relaxed.Enqueue(e.target);
    foreach(var v in relaxed)
    {
        List<long> nodes = BFS(G, v);
        foreach (var u in nodes)
            if (!Arbitrage.Contains(u))
                Arbitrage.Add(u);
    }
    return Arbitrage;
}
```

نمونه کد ۳: Detect Infinite Arbitrage

۴.۴.۶ Reconstruct Infinite Arbitrage

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

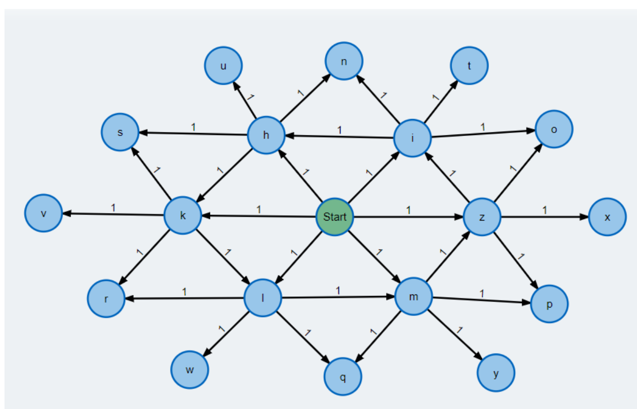
۵.۶ Bidirectional Search

۱.۵.۶ Why not just Dijkstra

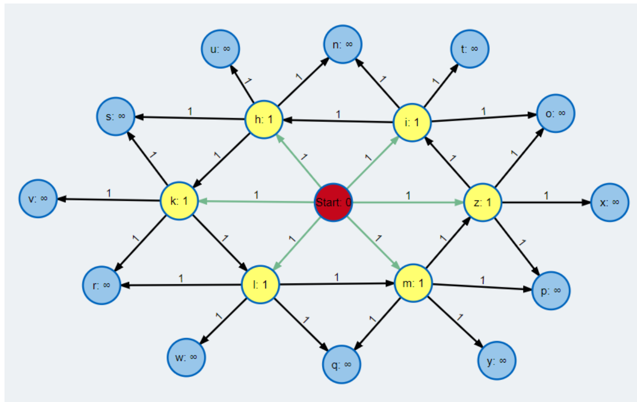
برای بدست آوردن کوتاه ترین مسیر از گره ای به گره دیگر گاهی اوقت دایجسترا الگوریتم مناسبی نیست ولی چرا؟ پیچیدگی زمانی دایجسترا `O((|V|+|E|)log|V|)` است که نسبتا سریع است پس چرا ممکن است گاهی به اندازه کافی خوب نباشد؟

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

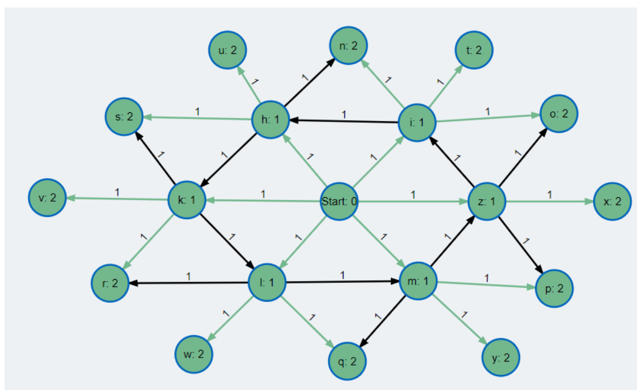
در شکل های زیر نحوه پیشرفت الگوریتم دایجسترا روی یک گراف را می بینیم و خواهیم فهمید که چرا ممکن است سریع عمل نکند:[۱۱]



شکل ۲.۶: دایجسترا(۱)



شکل ۳.۶: دایجسترا(۲)



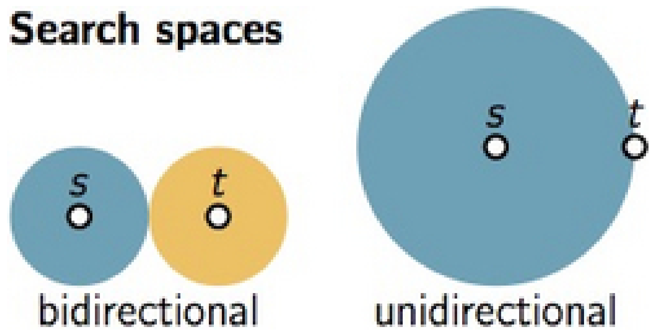
شکل ۴.۶: دایجسترا(۳)

۶.۵.۲ Growing Circle

قضیه:وقتی که راس `v` از طریق `Extract Min` انتخاب می شود،فاصله `u` برابر قطر `s` تا `u` است. `(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 قبلا پردازش شده اند.دایره راس های پردازش شده هر بار بزرگ تر می شود.

figure reference



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 نسبتا خوب عمل می کند و سرعت پیداکردن را تقریبا دو برابر می کند.اما باید برای گراف شبکه های اجتماعی هم بررسی شود.

۳.۵.۶ Six Handshakes

در سال ۱۹۲۹،فریگیس کاریتنی،ریاضیدان مجارستانی،فرضیه ای به نام دنیای کوچک را مطرح کرد.بر اساس این فرضیه شما حداکثر 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.

۴.۵.۶ Facebook

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

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

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

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

۶.۶ Bidirectional Dijkstra

ابتدا بهتر است دایجسترا را یادآوری کنید.می توانید از این لینک کمک بگیرید[۱۴]

۱.۶.۶ Reversed Graph

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

شکل ۶.۶: Reversed Graph

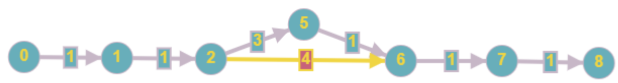
۲.۶.۶ Algorithm

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

۳.۶.۶ Computing Distance

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

figure reference



شکل ۷.۶: Compute Distance

۴.۶.۶ قضیه

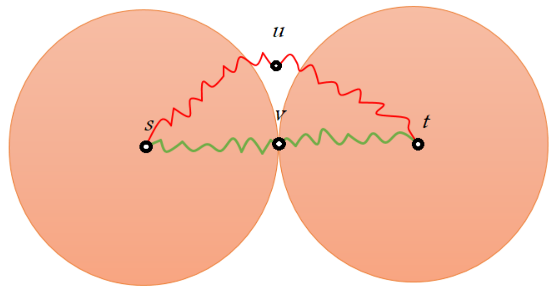
فرض کنید `dist[u]` فاصله ای است که در دایجسترا از `s` در گراف G تخمین زده می شود و $dist^R$`[u]` به طور یکسان در دایجسترا از `t` در گراف G^R باشد.کوتاه ترین مسیر از `s` تا `t` از گره ای مانند `u` می گذرد که یا در G یا در G^R و یا در هر دو آن ها پردازش شده است و خواهیم داشت:

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

۵.۶.۶ اثبات

فرض می کنیم راسی مانند `u` که در خارج از دایره دایجسترا چه در G و چه در G^R 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.

figure reference



شکل ۶.۸: Proof

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

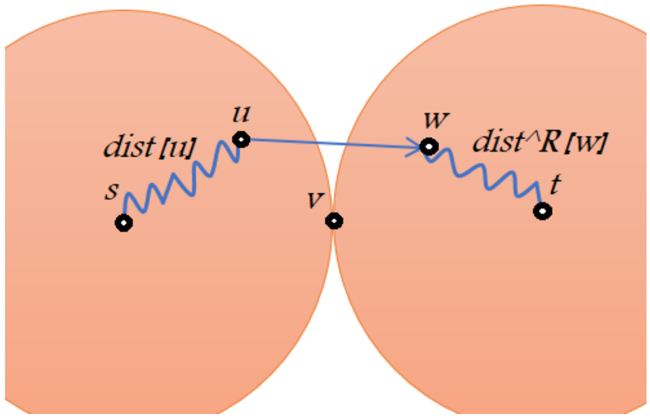
$$= dist[u] + dist^R[u]$$

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

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

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

figure reference



شکل ۹.۶: Proof(۲)

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

```
for (u,v) ∈ E(G) do
    Relax(u,v,dist,prev)
end
proc.Append(u)
```

```
Input: G,s,t
Output: shortest path s to t
Function Bidirectional Dijkstra(G,s,t):
    G^R ← ReverseGraph(G)
    Fill dist,dist^R with inf for each node
    dist[s] ← 0, dist^R[t] ← 0
    Fill prev,prev^R with none for each node
    proc ← empty, proc^R ← empty
    while true do
        v ← ExtractMin(dist)
        Process(v,G,dist,prev,proc)
        if v in proc^R then
            return ShortestPath(s,dist,prev,proc,...)
        end
        v^R ← ExtractMin(dist^R)
        repeat symmetrically for v^R as for v
    end
```

حالا که دایجسترا دوطرفه را انجام دادیم باید کوتاه ترین مسیر بین `s` و `t` را با استفاده از اطلاعات بدست آمده از متدهای قبلی و قضیه ذکرشده بدست آوریم.شبه کد این عملیات به این صورت است:

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

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

```
distance ← inf, u_best ← None
for u in proc+proc^R do
    if dist[u]+dist^R[u]<distance then
        u_best ← u
        distance←dist[u]+dist^R[u]
    end
end
path←empty
last← u_best
while last≠s do
    path.Append(last)
    last←prev[last]
end
path←Reverse(path)
last← u_best
while last≠t do
    last← prev^R[last]
    path.Append(last)
    return (distance,path)
end
```

جلسه ۸

درخت های پوشای کمینه

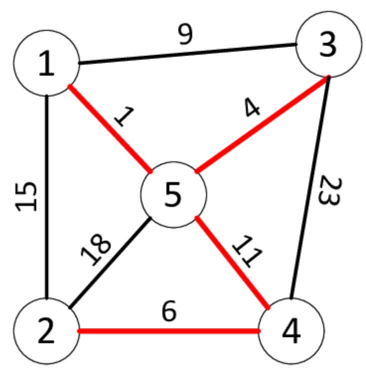
سهراب نمازی - ۱۳۹۸/۱۲/۱۱

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 پوشا 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 مجوعه ای از یال های وزن دار خواهد بود. اما با توجه به این که ما کدام یال ها از گراف اولیه را برای درخت پوشای خود انتخاب کرده ایم، مجموع وزن یال های درخت پوشا میتواند متفاوت باشد. اگر یال ها را به گونه ای انتخاب کنیم که مجموع وزن یال های درخت حاصل کمترین مقدار ممکن شود، به درخت پوشای حاصل، درخت پوشای کمینه میگوییم.

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

- گراف اولیه یک گراف همبند، بدون جهت و وزن دار با وزن های مثبت باشد
- 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. طبیعتا اتصال سیم بین دو کامپیوتر میتواند هزینه متفاوتی با انجام همین کار بین دو کامپیوتر دیگر داشته باشد. در این مثال اگر کامپیوترها را رئوس گراف در نظر بگیریم و هزینه اتصال هر دو کامپیوتر را مانند یک یال وزن دار بین آن دو کامپیوتر در نظر بگیریم، حاصل گراف اولیه ما خواهد بود. حال با پیدا کردن درخت پوشای کمینه این گراف، ما میتوانیم به سادگی و با کمترین هزینه ممکن شبکه مورد نظر خود را تشکیل دهیم.

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

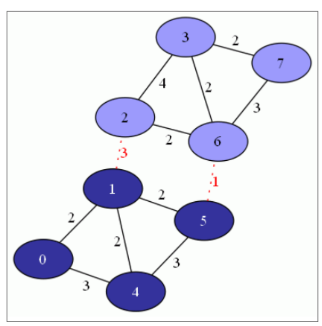
۳.۸ بدست آوردن درخت پوشای کمینه

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

کات پراپرتی

فرض کنید تا میانه الگوریتم لازم برای بدست آوردن درخت پوشای کمینه رفته ایم. به این معنای که تعدادی از یال هایی که قطعا 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 قطعا باید یال با کمترین وزن ممکن را انتخاب کرد.

figure reference



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 مجموعا دو واحد وزن بیشتری دارد. پس نتیجه میگیریم انتخاب ما اشتباه

بوده و درخت حاصل کمینه نیست. پس فرض ما که انتخاب نکردن یال با کمترین وزن بوده، غلط است، پس کات پراپرتی برقرار است.

۱.۳.۸ الگوریتم کروسکال

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

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

چک کردن دور در گراف

برای چک کردن دور، از ساختار داده ای محموعه های جدا یا همان دیسجوینت 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 ، مجوعه های آن دو را با هم ادغام میکنیم. جهت یادآوری مباحث مربوط به دیسجوینت ست ها میتوانید به جلسات هجدهم و نوزدهم از همین کورسی که لینک آن در انتهای این درس آمده است مراجعه کنید [۱۵]

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:

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

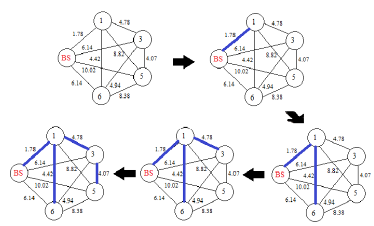
بنابراین پیچیدگی زمانی کلی الگوریتم کروسکال برابر است با :

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

۲.۳.۸ الگوریتم پریم

در ابن 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.

figure reference



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

برای بررسی دقیقتر جزئیات و تفاوت های میان دو الگوریتم پریم و کروسکال میتوانید به لینکی که در انتهای این جزوه آورده شده است مراجعه کنید. [۲۷]

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:

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

که این مقدار با توجه به نحوه پیاده سازی الگوریتم میتواند متفاوت باشد.
در صورت پیاده سازی با آرایه:

$$\mathrm{O}(V^2)$$

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 نهایتا دو الگوریتم حریصانه پریم و کروسکال برای بدست آوردن درخت پوشای کمینه ارائه شد.

جلسه ۹

الگوریتم جست وجو *A

محمدعلی فراهت - ۱۳۹۸/۱۲/۱۳

جزوه جلسه ۹ام مورخ ۱۳۹۸/۱۲/۱۳درس طراحی و تحلیل الگوریتم تهیه شده توسط محمدعلی فراهت. در جهت مستند کردن مطالب درس طراحی و تحلیل الگوریتم .

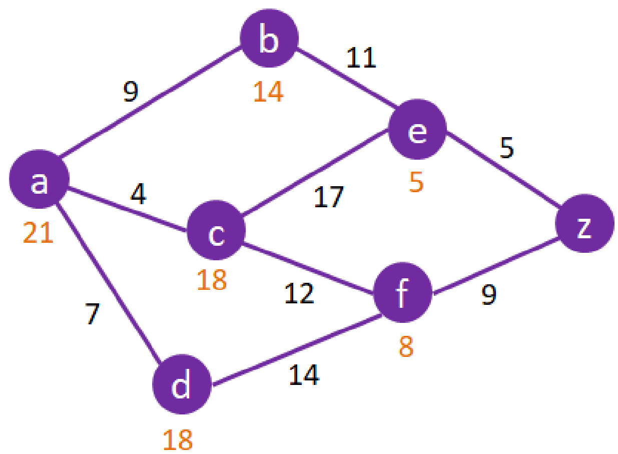
۱.۹ ایده کلی

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

۲.۹ 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 .

figure reference



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

۳.۹ محاسبه کوتاه‌ترین مسیر

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

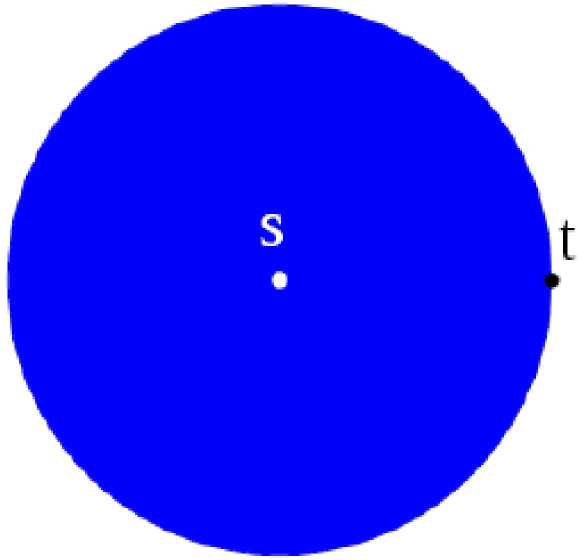
$$\ell_\pi(u, v) = \ell(u, v) - \pi(u) + \pi(v)$$

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

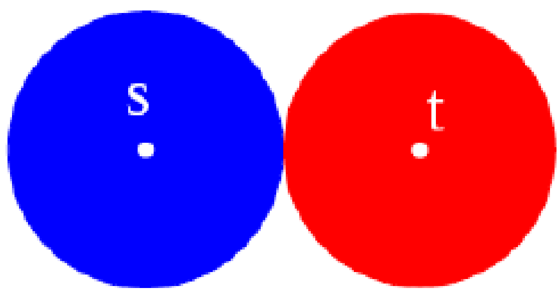
۴.۹ Bidirectional-A*

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

figure reference



شکل ۲.۹: روش معمولی



شکل 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 می تواند کاربردهای زیادی داشته باشد. از جمله:

- داروسازی
- کشاورزی
- بیوتکنولوژی

طول ژنوم انسان در حدود $3 * 10^9$ می‌باشد و بین ژنوم انسان های مختلف، تفاوت های اندکی وجود دارد که موجب تفاوت های فردی مانند قد، بیماری های ژنتیکی و ... می‌شود. پیدا کردن این تفاوت ها 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.

```
class PatternInText {

    void search(String txt, String pat)
    {
        int M = pat.Length;
        int N = txt.Length;

        for (int i = 0; i <= N - M; i++) {
            int j;

            for (j = 0; j < M; j++)
                if (txt[i + j] != pat[j])
                    break;

            if (j == M)
                Console.WriteLine(" index at found "Pattern + i);
        }
    }
}
```

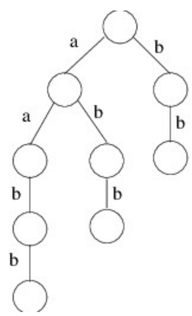
نمونه کد ۴: راه ساده در سی‌شارپ

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

[‡] BruteForce

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



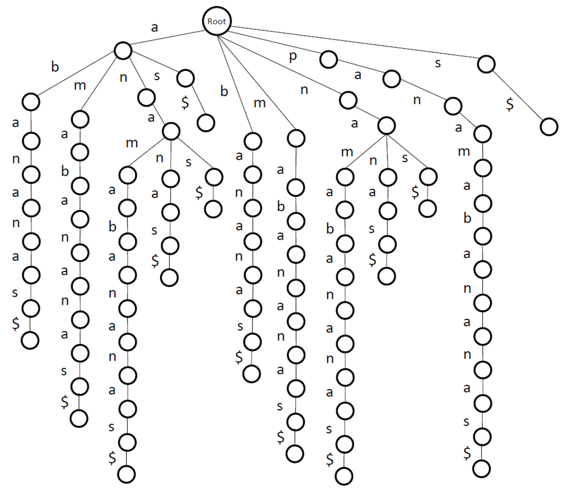
شکل ۱.۱۱: Trie for aabb, abb, bb

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

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

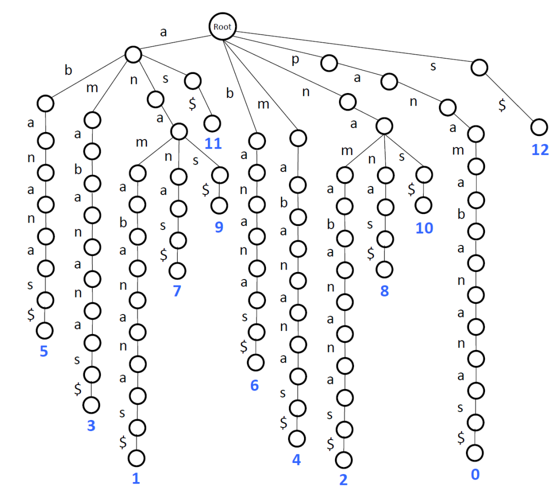
۳.۲.۱۱ ساخت Suffix Tree از متن

اگر به جای ساختن Trie از الگوها، همه‌ی Suffix های متن را پیدا کنیم و از آن ها Trie بسازیم، به `Suffix Tree` متن رسیده‌ایم. ۲.۱۱



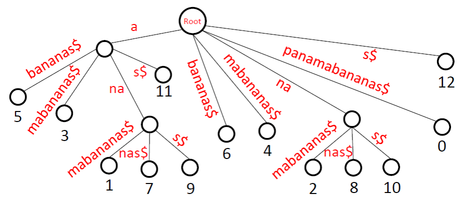
شکل ۲.۱۱: Suffix Tree for "panamabanana"

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



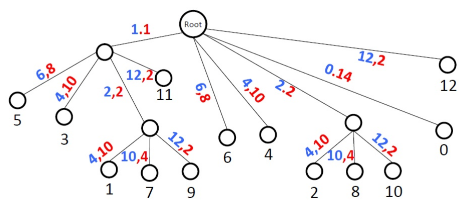
شکل ۳.۱۱: Suffix Tree with indexes for "panamabanana"

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



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

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



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

جلسه ۱۲

BWT

متین مرجانی - ۱۳۹۹/۱/۱۷

۱.۱۲ مقدمه

در جلسه ی گذشته با مفهموم Suffix Trie آشنا شدیم ولی با مشکل حافظه از O(|Text|*|Text|) نیز روبرو شدیم برای همین آن را به Suffix Tree ارتقا دادیم که در آن تعداد Node های درخت و در نتیجه حافظه ای اشغال می شد را کاهش دادیم. در Suffix Tree حجم حافظه ای از O(|Text|) 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 |Text|*20 می باشد که عددی تاثیر گذار است.

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

۲.۱۲ BWT

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

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

Run-length encoding :

Text

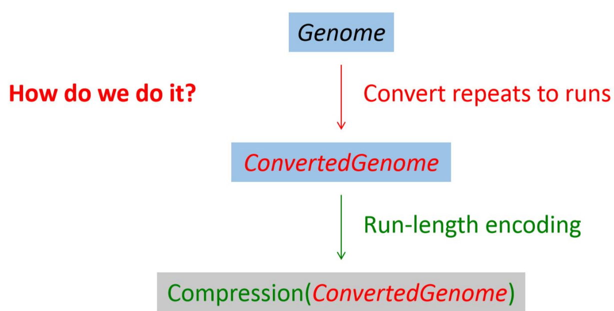
GGGGGGGGGGCCCCCCCCCCCAAAAAAATTTTTTTTTTTTTTTCCCCCG

=

۱۰G۱۱C۷A۱۵T۵C۱G

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

figure reference



۳.۱۲ Constructing BWT

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

figure reference

Cyclic Rotations

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

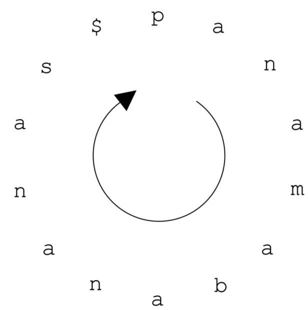


figure
reference

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

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

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.

figure
reference

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

رشته ی بدست آمده همان تبدیل BWT مورد نظر میباشد:

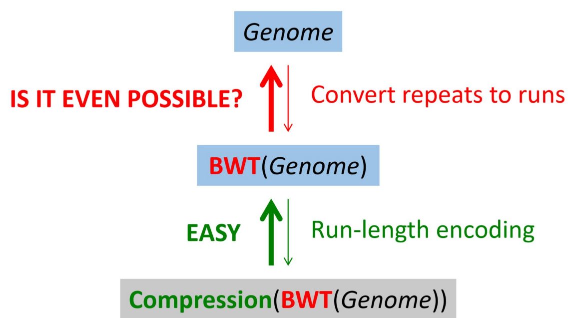
BWT(panamabananas$)=smnpbnnaaaaa$a

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

BWT(panamabananas$)=smnpbnnaaaaa$a

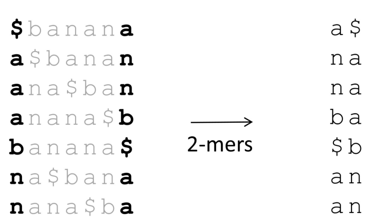
۴.۱۲ Inverting BWT

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



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

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



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 ی اصلی را تا الان دو به دو مرتب کردیم. این یعنی دو حرف اول ماتریس را بدست آورده ایم.
با Sort کردن ماتریس این دو حرف به صورت زیر، و جایگذاری آن در ماتریس اصلی؛ به کامل تر شدن ماتریس تبدیل نزدیک تر می شویم.

figure reference

$banan**a**		a$		**$b**
a$bana**n**		na		**a$**
ana$ba**n**		na		**an**
anana$**b**	$\xrightarrow{\text{2-mers}}$	ba	$\xrightarrow{\text{Sort}}$	**an**
banana**$**		$b		**ba**
na$ban**a**		an		**na**
nana$b**a**		an		**na**

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

figure reference

$banan**a**		a$b
a$bana**n**		na$
ana$ba**n**		nan
anana$**b**	$\xrightarrow{\text{3-mers}}$	ban
banana**$**		$ba
na$ban**a**		ana
nana$b**a**		ana

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

figure reference

$banana		a$bana		**$banan**
a$banan		na$ban		**a$bana**
ana$ban		nana$b		**ana$ba**
anana$b	$\xrightarrow{\text{6-mers}}$	banana	$\xrightarrow{\text{Sort}}$	**anana$**
banana$		$banan		**banana**
na$bana		ana$ba		**na$ban**
nana$ba		anana$		**nana$b**

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

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

figure reference

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

شکل ۱.۱۳: ماتریس حالات

اگر به شکل ۱.۱۳ توجه کنیم،میتوان به یک مشاهده جالب رسید. به طور مثال بیایید به هر a موجود در متن یک شماره به خصوص دهیم. به اولین a در سطر دوم ماتریس عدد ۱ ،به اولین a در سطح سوم عدد ۲ و تا اولین a در سطر هفتم عدد ۶ را بدهیم. با این کار ما به هر a موجود در متن مان یک عدد منحصر به فرد دادیم.

حال بیاید a ها را از اول سطر های ۲ تا ۷ حذف کنیم و انهارا 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.

- امینn وقوع حرف در اولین سطر
- و امینn وقوع حرف در اخرین سطر
- مطابق با حرف در همان نقطه در متن است

figure reference

$p_1a_3n_1a_2m_1a_1b_1a_4n_2a_5n_3a_6s_1\$_1$

شکل ۲.۱۳: متن شماره گذاری شده

$\$_1$panamabananas$s_1$
$a_1$bananas\$panamm_1
$a_2$mabananas\$pann_1
$a_3$namabananas\$$p_1$
$a_4$nanas\$panamab_1
$a_5$nas\$panamabann_2
a_6s\$panamabanann_3
$b_1$ananas\$panama_1
$m_1$abananas\$pana_2
$n_1$amabananas\$pa_3
$n_2$anas\$panamaba_4
$n_3$as\$panamabana_5
$p_1$anamabananas$\$_1$
s_1\$panamabanana_6

شکل ۳.۱۳: ماتریس حالات شماره گذاری شده

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

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

حال از علامت دلار شروع میکنیم و به خرف قبلی مبرویم و این کار را تا زمانی انحام میدهیم که دوباره به علامت دلار برسیم.

برای نشان دادن روند کار از علامت دلار استفاده میکنیم و به اولین s میرسیم،سپس بعد ان از اولین s به ۶ امین a 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 * |Test|
- Time : O(|Text|)

به تطبیق الگو ها برگردیم.

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

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

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

حال سوال این است که میتوان از BWT(Text) برای طراحی یک الگوریتم زمان - خطی با حافظه مفید بیشتری برای تطبیق الگو های 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.

```
$_1 panamabanana s_1
a_1 bananas$pana m_1
a_2 mabananas$pa n_1
a_3 namabananas$ p_1
a_4 nanas$panama b_1
a_5 nas$panamaba n_2
a_6 s$panamabana n_3
b_1 ananas$panam a_1
m_1 abananas$pan a_2
n_1 amabananas$p a_3
n_2 anas$panamab a_4
n_3 as$panamaban a_5
p_1 anamabananas $_1
s_1 $panamabanan a_6
```

شکل ۵.۱۳: مثال ۱

اخرین 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 ای که پیدا کردیم کدامشان 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.

```
$_1 panamabanana s_1
a_1 bananas$pana m_1
a_2 mabananas$pa n_1
a_3 namabananas$ p_1
a_4 nanas$panama b_1
a_5 nas$panamaba n_2
a_6 s$panamabana n_3
b_1 ananas$panam a_1
m_1 abananas$pan a_2
n_1 amabananas$p a_3
n_2 anas$panamab a_4
n_3 as$panamaban a_5
p_1 anamabananas $_1
s_1 $panamabanan a_6
```

شکل ۶.۱۳: مرحله دوم

حالا به شکل الگوریتمی بیایید سوال بالا را حل کنیم.

ابتدا با دو مفهوم اندیس بالا و اندیس پایین دراین 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
```

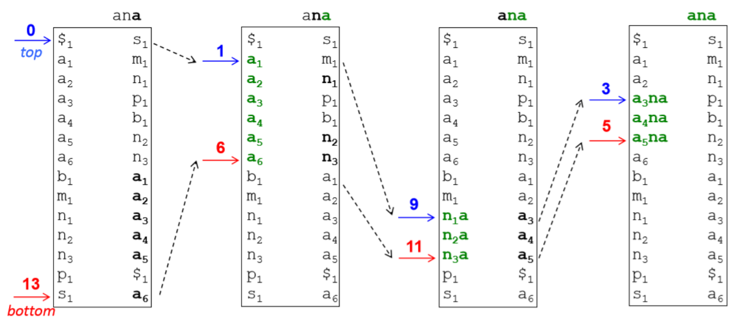
Algorithm 11: BWMatching

با این حال BWMatching سرعت زیادی طبق الگوریتم بالا ندارد. دلیل ان 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 بخشید.

figure reference



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 امین مکان در اخرین ستون = Count،symbol(i،LastColumn)

figure reference

i	*FirstColumn*	*LastColumn*	LastToFirst(i)	Count						
				$	a	b	m	n	p	s
0	$\$_1$	s_1	13	0	0	0	0	0	0	0
1	a_1	m_1	8	0	0	0	0	0	0	1
2	a_2	n_1	9	0	0	0	1	0	0	1
3	a_3	p_1	12	0	0	0	1	1	0	1
4	a_4	b_1	7	0	0	0	1	1	1	1
5	a_5	n_2	10	0	0	1	1	1	1	1
6	a_6	n_3	11	0	0	1	1	2	1	1
7	b_1	a_1	1	0	0	1	1	3	1	1
8	m_1	a_2	2	0	1	1	1	3	1	1
9	n_1	a_3	3	0	2	1	1	3	1	1
10	n_2	a_4	4	0	3	1	1	3	1	1
11	n_3	a_5	5	0	4	1	1	3	1	1
12	p_1	$\$_1$	0	0	5	1	1	3	1	1
13	s_1	a_6	6	1	5	1	1	3	1	1
				1	6	1	1	3	1	1

شکل ۸.۱۳: ارایه شمارش

BetterBWMatching(FirstOccurrence,LastColumn,Pattern,Count)

top <- 0

bottom <- |LastColumn| - 1 **while** *top<=bottom* **do**

  if *Pattern is nonepmty* **then**

    symbol <- last letter in Pattern

    remove last letter from Pattern

    top <- FirstOccurrence(symbol) + Count.symbol(top,LastColumn)

    bottom <- FirstOccurrence(symbol) + Count.symbol(bottom + 1,LastColumn) - 1

  else

    return bottom - top + 1

  end

end

Algorithm 12: BetterBWMatching

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

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

figure reference

13	$\$_1$panamabanana$s_1$
5	$a_1$bananas\$panam_1
3	$a_2$mabananas\$pan_1
1	$a_3$namabananas\$$p_1$
7	$a_4$nanas\$panamab_1
9	$a_5$nas\$panamaban_2
11	a_6s\$panamabanan_3
6	$b_1$ananas\$panama_1
4	$m_1$abananas\$pana_2
2	$n_1$amabananas\$pa_3
8	$n_2$anas\$panamaba_4
10	$n_3$as\$panamabana_5
0	$p_1$anamabananas$\$_1$
12	s_1\$panamabanana_6

شکل ۹.۱۳: ارایه پسوند

جلسه ۱۴

الگوریتم KMP

امید میرزاجانی - ۱۳۹۹/۲/۱۲

جزوه جلسه ۱۴ام مورخ ۱۳۹۹/۲/۱۲ درس طراحی و تحلیل الگوریتم تهیه شده توسط امید میرزاجانی. در جهت مستند کردن مطالب درس طراحی و تحلیل الگوریتم
این الگوریتم که یکی از معروف ترین الگوریتم ها برای String Pattern Matching است، در سال ۱۹۷۰ توسط سه نفر به نام های 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 خروجیِ آن تمام نمایه * های است که آن الگو در متن اصلی پیدا می شود.

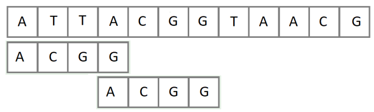
۱.۱۴ Brute Force

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

*Index

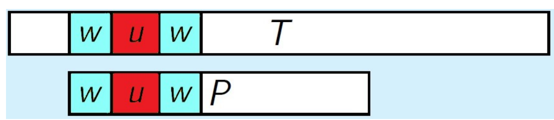
اما کمی که به همین الگوریتم ساده فکر کنیم، میتوانیم بعضی از نمایه ها را صرف نظر کنیم. به طور مثال در شکل زیر، پس از چک کردن نمایه اول، دیگر نیازی به چک کردن نمایه های دوم و سوم نیست؛ زیرا مشخصاً آن ها با A شروع نمیشوند و قابل صرف نظر هستند.

figure reference



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

figure reference



۲.۱۴ Function Prefix

یک آرایه ای تعریف میکنم که نمایه امِ i آن مقدار طولانی ترین بردر آن رشته را برمیگرداند.

figure reference

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

Prefix Function

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

---

† Match

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

```
ComputePrefixFunction(P)
{
        s = array of integers of length |P|
        s[0] = 0, border = 0
        for i from 1 to |P| - 1:
                while (border > 0) and (P[i] != P[border]):
                        border = s[border - 1]
                if P[i ] == P[border ]:
                        border = border + 1
                else:
                        border = 0
                s[i ] = border
        return s
}
```

نمونه کد ۵: محاسبه Prefix Function

۳.۱۴ الگوریتم نهایی

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

figure reference

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

همانطورکه 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.

```
FindAllOccurrences(P, T)
        S = P + '$' + T
        s = ComputePrefixFunction(S)
        result = empty list
        for i from |P| + 1 to |S| - 1:
                if s[i ] == |P|:
                        result.Append(i - 2|P|)
        return result
```

نمونه کد ۶: محاسبه Prefix Function

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

جلسه ۱۵

الگوریتم kmp و suffix array efficent

صدرا خاموشی - ۱۳۹۸/۲/۶

جزوه جلسه ۱۵ام مورخ ۱۳۹۸/۲/۶ درس طراحی و تحلیل الگوریتم تهیه شده توسط صدرا خاموشی .

۱.۱۵ مقدمه :

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

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

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

Border : border یک string قسمتی از string هست که prefix و suffix با هم برابر باشند. برای اطلاعات بیشتر درباره suffix وprefix به این سایت مراجعه کنید. [۲۶].

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

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

برای اطلاعت بیشتر درباره ی kmp به این لینک مراجعه شود. [۲۱]

برای مشاهده شبه کدkmp † می‌توانید از مثال زیر در الگوریتم ۴۲ 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

کوییز حل شده واسه ی prefix function :

*برای توضیحات بیشتر و کامل تر به صفحه ۸۶ ، string۳.pdf مراجعه کنید
†kmp

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

پیدا کردن suffix tree از روی suffix array :

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

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

Suffix array :

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

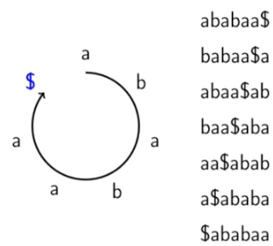
اگه به آخر string ، $ اضافه کنیم و جایگشت دوری اش را بنویسیم. به cyclic shift میرسیم .حال اگر این cyclic shift های بدست آمده را sort کنیم براساس حروف اولشان، به sorting cyclic shift میرسیم. سپس اگر برای هر جایگشت ، character های بعد از $ را پاک کنیم به همان suffix array میرسیم.

مثال :

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

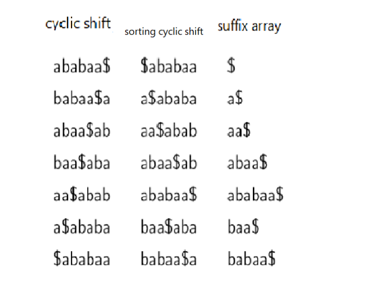
‡

figure reference



شکل ۱.۱۵: cyclic shift length of ۶

figure reference



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

شکل ۲.۱۵: Suffix array

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

ابتدا character ها text داده شده را sort میکنیم.

حال cyclic shift به طول L=۱ sort، شده.

تازمانی که L<length(text) ، cyclic shift های به طول ۲L را با استفاده از قبلی ها sort میکنیم.

۱ ->۲->۴->۸ . . .

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

از آنجا که مثلا cyclic shift طول ۲ از ۲ تا cyclic shift با طول ۱ تشکیل شده پس میتوان آن ها را با cyclic shift به طول ۱ ، sort کرد و الی آخر....

وقتی L >length(text) شد cyclic shift سورت شده ی بدست آمده همان sorting cyclic shift هست.

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

مثال :

```
abbcaaabcc
count Sort :
aaaabbbccc
```

در واقع تعداد alphabet ها را در آورده و با استفاده از آن ها single character ها را sort میکنیم. پیچیدگی زمانی count sort ، `O(length(text) + alphabets)` میشود. هر بار که طول cyclic shift را ۲ برابر میکنیم.برای sort کردن آن ها با پیچیدگی زمانی O(n) انجام میدهیم و این کار را هم log(n) بار را انجام میدهیم.پس ساختن suffix array ، `O(nlogn)` طول میکشد.

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

برای مطالعه در مورد sort stable به لینک زیر مراجعه شود: [۲۸]

: sort single charecter

its our alphabets

§(برای توضیحات بیشتر به مثال صفحه ی ۵۴ اسلاید string۴.۱ مراجعه شود)

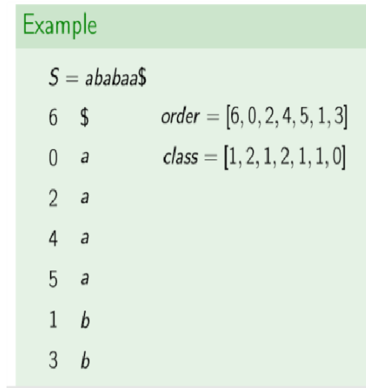
```
Data: S
Result: sorted charecters
order = array of size |S|;
count = array of size |Σ| index=0;
while index< |S|-1 do
    count[S[index]]=count[S[index]]+1;
    index=index+1;
end
index=1;
while index< |Σ|-1 do
    count[index]=count[index]+count[index-1];
    index=index+1;
end
index = |S|-1;
while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

Algorithm 14: sorting Charecters

Equivalnce Class :

Ci : این نماد یعنی cyclic shift به طول L که از i index ام شروع میشود. اگر Ci=Cj 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 cyclic shift ها به طول L را محاسبه کرده و داخل آرایه قرار میدهیم.و برای i و j اگر Ci=Cj آنگاه class[i]=class[j].

figure reference



شکل ۳.۱۵: class

برای توضیحات تکمیلی به صفحه ی ۷۶ اسلاید string۴.۱ مراجعه شود

[۹]

[۹] page ۷۶ of string۴.۱

جلسه ۱۶

آرایه پسوندی بهینه و آرایه LCP

زهرا حسینی - ۱۳۹۹/۱/۲۶

۱.۱۶ دوره مفاهیم آرایه و درخت پسوندی

آرایهٔ پسوندی آرایه‌ای از همهٔ پسوندهای یک رشته است.این آرایه بر اساس نویسه های * هر یک از پسوند ها مرتب شده است درخت پسوندی ساختمان داده ایی است که هر یک از یال های این درخت با یک پسوند برچسب 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 ی پسوندهای یک رشته به طول S برابر است با:

```
1+2+...+|S|=Θ(|S|^2)
```

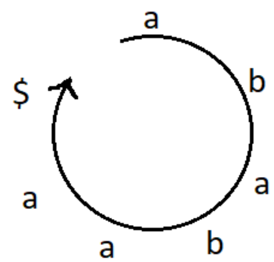
مرتب کردن همه ی این موارد حافظه زیادی را مصرف میکند. ذخیره کردن آنگاه نیز از لحاظ پیچیدگی زمانی برابر است با: $O(|S|)$

حال به بررسی نحوه ساخت آرایه پسوندی میپردازیم

۲.۱۶ ساخت آرایه پسوندی

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

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

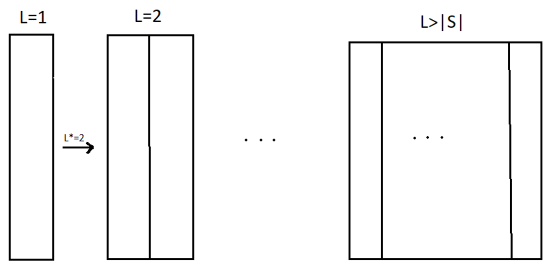


شکل ۱.۱۶: جا به جایی جزئی چرخه ای†

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

figure reference

† partial cyclic shift



شکل ۲.۱۶: تغییر طول چرخش

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

۱.۲.۱۶ مرتب کردن نویسه های منفرد

[‡] در این مورد از counting sort . که یک روش پایدار [§] است استفاده میکنیم برای این منظور میتوان از شبه کد زیر استفاده کرد .

[‡] sort single characters
[§] stable

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

Algorithm 15: SortCharacters(S)

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

۲.۲.۱۶ کلاس های هم ارزی

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

```
s=abcdefghi
c_{2-3}=cdef
```

[۹] Equivalence classes

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

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

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

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

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

Data: S,order

Result: class

class ← array of size |S|

class[order[0]] ← 0

for *i from 0 to |S|-1* **do**

  if *S[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 شیفت های جرخشی دو برابر شده

همانطور که درابتدای این بخش گفتیم در هر مرحله طول چرخش را دو برابر میکنیم و رشته های حاصل را مرتب میکنیم،باید به خاطر داشت که فقط قسمت اضافه شده به رشته مرتب میشود. ۳.۱۶ [۴]

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

c_i

c_i'

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

به مثال زیر دقت کنید:

[۴] Sort Doubled cyclic Shifts

```
S = ababaa$
L = 2
i = 2
```
c_i = c_2 = ab

c_{i+L} = c_{2+2} =c_4 = aa

c'_i = c'_2 = abaa =c_2c_4

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

Data: S

Result: order

count ← zero array of size |S|

newOrder ← array of size |S|

for *i from 0 to |S|-1* **do**

  count[class[i]]← count[class[i]]+1

end

for *j from 1 to* |Σ|*-1* **do**

  count[j]←count[j]+count[j-1]

end

for *i from |S|-1 to 0* **do**

  start← (order[i]-L+|S|) mod |S|

  cl←class[start] count[cl]← count[cl]-1

  newOrder[count[cl]]← start

end

return newOrder

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

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

۴.۲.۱۶ محاسبه مجدد کلاس هم ارزی

همانطور که دیدیم با دو برابر شدن طول چرخش آرایه order را نیز بر اساس رشته های جدید تغییر دادیم و بر اساس آن پیش رفتیم در این قسمت نیز باید آرایه مربوط به کلاس های هم ارزی را تغییر دهیم: در این قسمت دیگر نویسه ها را بررسی نمیکنیم در واقع باید چند نویسه که حاصل از شیفت 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

$(P_1 == Q_1)$ و $(P_2 == Q_2)$

به مثال زیر توجه کنید:

S = ababaa$

class = [1, 2, 1, 2, 1, 1, 0]

c'_6 $a (0, 1)

c'_5 a$ (1, 0)

c'_4 aa (1, 1)

c'_0 ab (1, 2)

c'_2 ab(1, 2)

c'_1 ba (2, 1)

c'_3 ba (2, 1)

newClass = [3, 4, 3, 4, 2, 1, 0]

برای ساخت این آرایه از شبه کد زیر استفاده میکنیم:

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

Algorithm 18: UpdateClasses(newOrder, class, L)

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

۵.۲.۱۶ ساخت آرایه پسوندی مرحله نهایی

در آخر باید از مراحلی که توضیح دادیم در کنار یکدیگر استفاده کنیم تا آرایه مورد نظر حاصل شود. برای این منطور از شبه کد زیر استفاده میکنیم. حلقه آمده شده در شبه 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 با: $O(|S| \log |S| + |\Sigma|)$

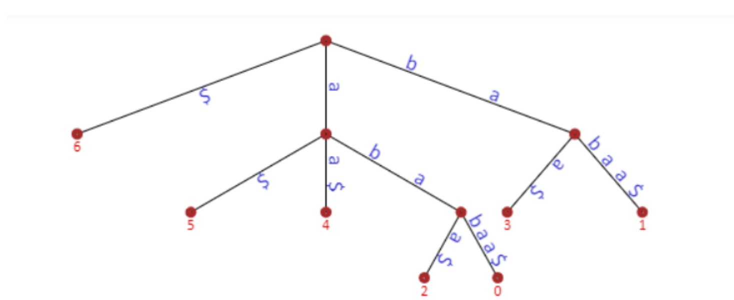
۳.۱۶ LCP ARRAY

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

۱.۳.۱۶ ایده کلی

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

figure reference



شکل ۳.۱۶: suffix tree

** suffix tree
†† Longest common prefix
‡‡ suffix array

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

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

```
∀ i < j
LCP(A[i], A[j])≤ lcp[i]
and
LCP(A[i],A[j]) )≤ lcp[j - 1]
```

۲.۳.۱۶ محاسبه ی آرایه LCP

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

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

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

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

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

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

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

Algorithm 21: InvertSuffixArray(order)

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

Algorithm 22: ComputeLCPArray(S,order)

مثال ها و شبه کد های ذکر شده در این بخش از دوره ی مربوط به رشته،هفته ی چهارم برداشت شده است.برای مطالعه بیشتر به سایت زیر مراجعه کنید [۲۲]. همچنین ابزار های محازی استفاده شده برای رسم شکل را میتوانید در این قسما مشاهده کنید. [۲۹].

جلسه ۱۷

Suffix Tree

هستی کرمدل - ۱۳۹۹/۱/۳۱

۱.۱۷ خلاصه ای از مطالب جلسه ی قبل

در آخر مباحث رشته در مورد ساختن بهینه suffixtree صحبت کردیم که در مرحله اولیه(o(n۲ داشت و می خواهیم آن را بهترکنیم. از ساختن suffixarray شروع می شود که همه ی suffix های آن sort شده است. که در مرحله اول همه تک کاراکترها را sort می کردیم بعد دو برابر آن ها را تا به آخر برسیم . بعد برای استفاده درست از suffixarray یک LCParray ساختیم که آرایه ای برای حساب کردن تعداد اشتراکات بین دو suffix پشت سر هم است. ایده کلی این کار این است که بعد از مقایسه ی دو suffix ، دو suffix بعدی که مقایسه می شود حداقل تعداد اشتراکاتشان LCP قبلی منهای یک است.

۲.۱۷ Prerequisites of LCP Array :

در ادامه مبحث LCParray به الگوریتم آن می پردازیم. دو suffix داریم و می خواهیم اشتراکاتشان را حساب کنیم، ورودی های تابع ما: LCP قبلی منهای یک به عنوان ،equal شماره suffix مورد نظر و بعدی آن بعنوان i و j و string اصلی هستند. اول LCP را برابر(max(۰،equal می گذارد تا منفی نشود بعد می گوییم تا موقعی که کاراکتر بعدی jوi از طول string بیشتر نشده ، اگر برابر نبودند LCP=LCP+۱ . این روند تا وقتی

ادامه دارد که به یکی برسند که برابر نیستند و return کنند .

۱.۱۷

algorithm

```
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: پیشنیاز LCPArray

```
Data: LCPOfSuffixes(S,i,j,equal)
lcp ← max(0,equal) ;
while i+lcp<|S| and j+lcp<|S| do
    if S[i+lcp] = S[j+lcp] then
        lcp ← lcp+1
    else
        break ;
    end
end
return lcp ;
```

Algorithm 23: Prerequisites of LCP Array

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

algorithm

```
InvertSuffixArray(order)

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

شکل ۲.۱۷: پیشنیاز LCPArray

```
Data: InvertSuffixArray(order)
pos ← array of size |order| ;
for i from 0 to |pos|-1 do
  | pos[order[i]] ← i ;
end
return pos ;
```

Algorithm 24: Prerequisites of LCP Array

۳.۱۷ LCP Array :

برای حساب کردن LCParray ابتدا آرایه ای به طول ۱-|S| می سازیم که S همان string اصلی است. سپس pos را حساب می کنیم. بعد suffix ما در order[۰] است یعنی order[۰] یک مکان است برای شروع suffix. بعد pos آن می شود همان مکانش در حال.suffixarray nextsuffix را حساب می کنیم که suffix بعدی آن است و LCP این دو را حساب می کنیم و در LCParray[orderindex] قرار می دهیم همچنین

suffix=suffix+۱ می کنیم. حال دوباره به اول for برمی گردیم. اگر orderindex به آخر رسید LCP=۰ و suffix=suffix+۱ می کنیم. زمان اجرای این الگوریتم O|S| است. در LCP یک while و یک for داریم که این while در آن محاسبه می شود که O|S| است. چرا O|S|۲ نمی شود؟علت آن این است که LCP در هر مرحله می تواند هر چقدر می خواهد زیاد شود ولی یکی کم می شود تعداد کل دفعاتی که می تواند زیاد شود بیشتر ۲|S| نیست.

۳.۱۷

algorithm

ComputeLCPArray(S, order)

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

شکل ۳.۱۷: ComputeLCPArray

```
Data: ComputeLCPArray(S,order)
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 ;
    else

    end
    nextSuffix ← order[orderIndex +1];
    lcp ← LCPOfSuffixes(S,suffix,nextSuffix,lcp-1);
    lcpArray[orderIndex] ← lcp ;
    suffix ← (suffix+1) mode |S| ;
end
return lcpArray ;
```

Algorithm 25: Compute LCP Array

۴.۱۷ Building suffix tree :

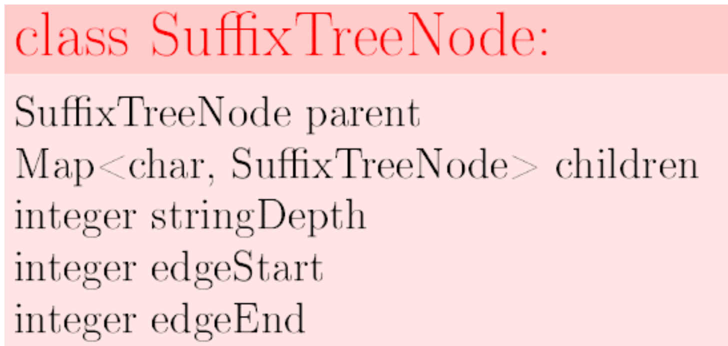
تا اینجا suffixarray و LCParray را حساب کردیم. حال اول به root، dollarsignرا اضافه می کنیم و LCP=۰ است. پس به ارتفاع ۰(که همان root است) می رویم و suffix بعدی را add می کنیم. و عدد suffix array را هم به انتهای آن اضافه می کنیم. مرحله بعدی آنقدر می آییم بالا که عمق کمتر از LCP شود.

۱.۴.۱۷ Implementation of Suffix Tree :

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

۴.۱۷

algorithm



شکل ۴.۱۷: Buildingsuffixtree

Data: ClassSuffixTreeNode

SuffixTreeNode parent;

Map<char,Suffix Tree Node> children ;

integer stringDepth ;

integer edgeStart ;

integer edgeEnd ;

Algorithm 26: Building Suffic Tree

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

۵.۱۷

algorithm

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

```
Data: STFromSA(S,order,lcpArray)
root ← newSuffixTreeNode(children =, parent=nil , stringDepth=0,
 edgeStart=-1, edgeEnd=-1);
lcpPrev ←0 ;
curNode ← root ;
for i from 0 to |S|-1 do
    suffix ← order[i] ;
    while curNode.stringDepth>lcpPrev do
        curNode ← curNode.parent ;
    end
    if curNode.stringDepth = lcpPrev then
        curNode ← CreatNewLeaf(curNode,s,suffix) ;
    else
        edgeStart ← order[i-1]+curNode.stringDepth ;
        offset ← lcpPrev- curNode.stringDepth ;
        midNode ← BreakEdge(curNode,d,edgeStart,offset) ;
        curNode ← CreatNewLeaf(midNode,s,suffix) ;
    end
    if i<|S|-1 then
        lcpPrev ← lcpArray[i] ;
    else

    end
end
return root ;
```

Algorithm 27: Building Suffix Tree

۵.۱۷ Suffix Tree Order :

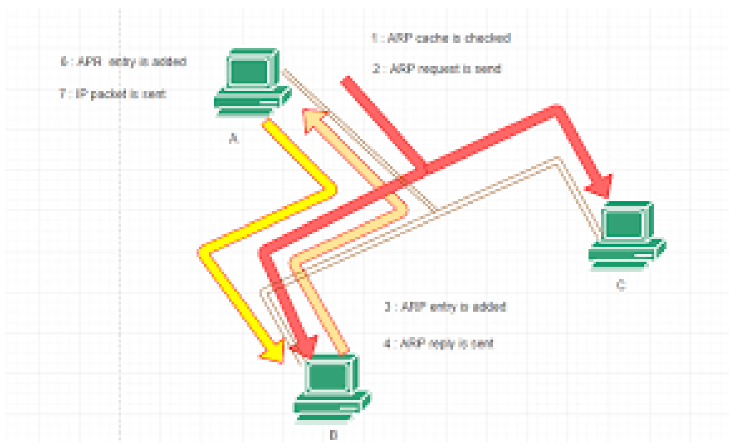
ساختن suffixtree از روی suffixarray در زمان خطی یعنی O|S| انجام می شود و ساختن suffixtree از ابتدا O(|S|log|S|) است زیرا زمان آن برابر با |S|log|S|+|S|+|S| است.

۶.۱۷ کلیت مطالب جلسه ی بعد :

شروع الگوریتم های پیشرفته با مباحث زیر:

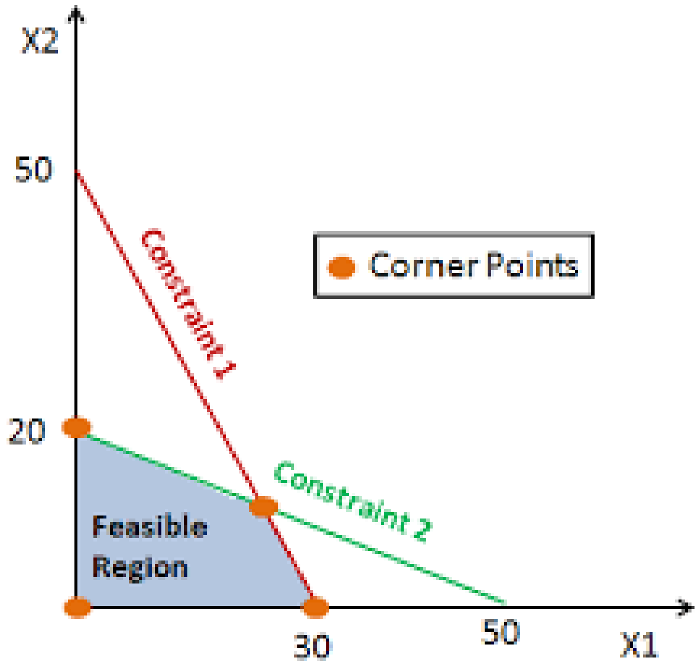
۱) Flowin networks

۶.۱۷



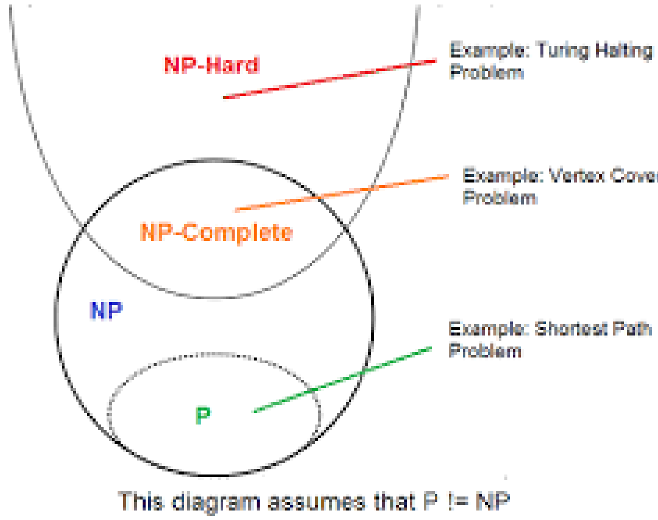
شکل ۶.۱۷: Flowin networks

۲) Linear Programming ۷.۱۷



شکل ۷.۱۷: Linear programming

۳) NP complete problems ۸.۱۷



شکل ۸.۱۷: NP Complete problems

۴) Coping with NP completeness

۵) Streaming Algorithms(optional)

جلسه ۱۸

جریان در گراف

محمد مصطفی رستم خانی - ۱۳۹۹/۲/۲

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

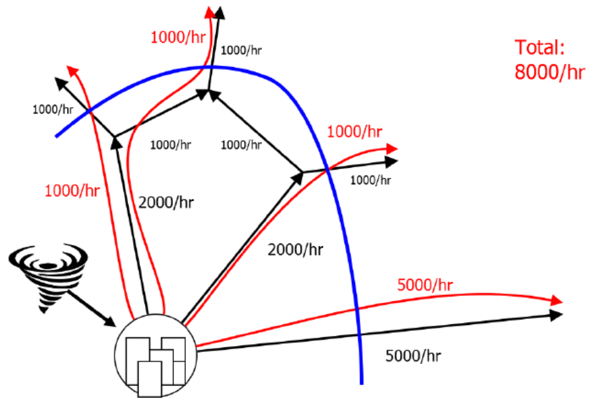
۱.۱۸ جریان در شبکه :(flows in network)

مثال:

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

figure reference

می توانید این کار را بهتر انجام دهید یا خیر.



شکل ۱.۱۸: network

۲.۱۸ network

- تعریف:

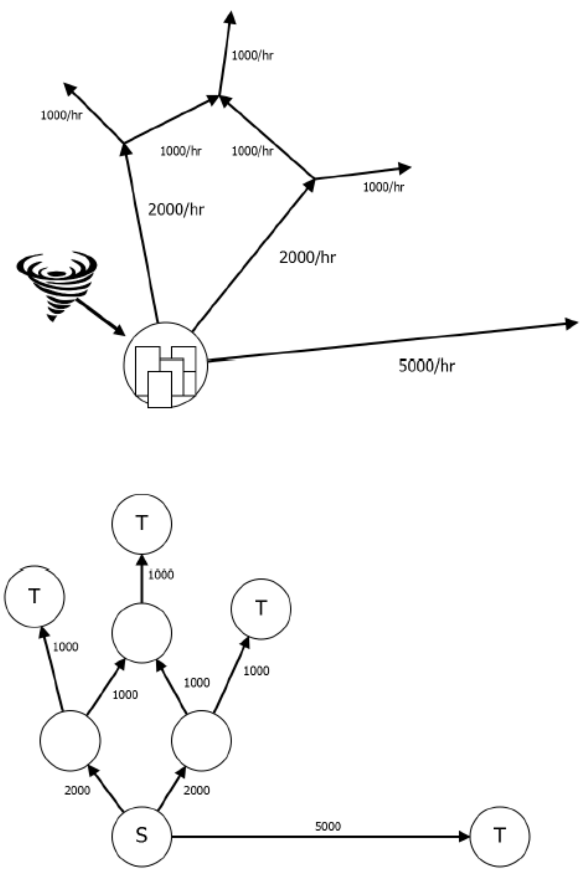
شبکه(network): یک شبکه یک گراف جهت دار (directed graph) G است به گونه ای که:

۱. به هر یال آن به مانند e مقداری حقیقی و مثبت به عنوان ظرفیت (c_e) ختصاص داده می شود.

۲. تعداد یک یا بیشتر راس مبدا(source) دارد.

۳. یک یا بیشتر راس مقصد(sink) دارد.

figure reference



شکل ۲.۱۸: example

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

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

$$0 \leq f_e \leq c_e \quad (۱.۱۸)$$

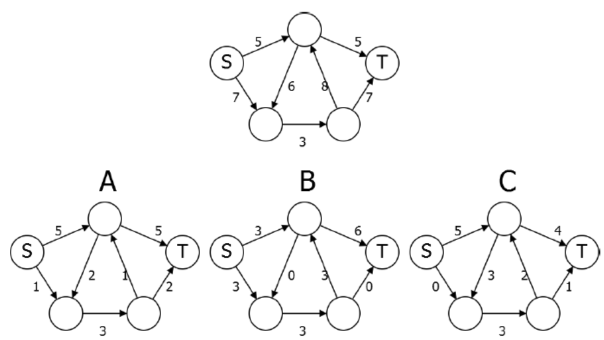
۲. برای هر یال به مانند برای هر راس v که مبدا و مقصد نیستند داریم:

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

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

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

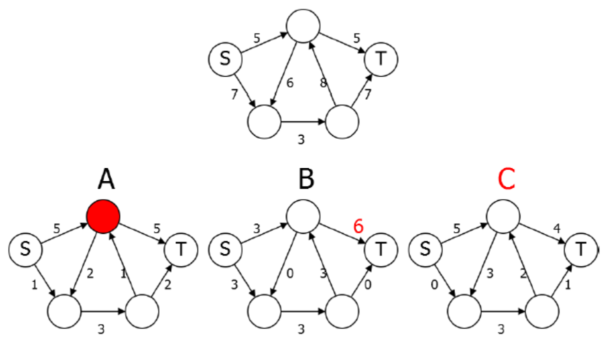
figure reference



شکل ۳.۱۸: flow example

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

figure reference



شکل ۴.۱۸: example flow

- کاربرد هایی از Flow Max :
  - ترافیک در شبکه ی حمل و نقل شهری
  - جریان ورودی و خروجی در خطوط نیرو
  - جریان در لوله های آب
  - شبکه ی اینترنت

اندازه ی جریان (flow size):

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

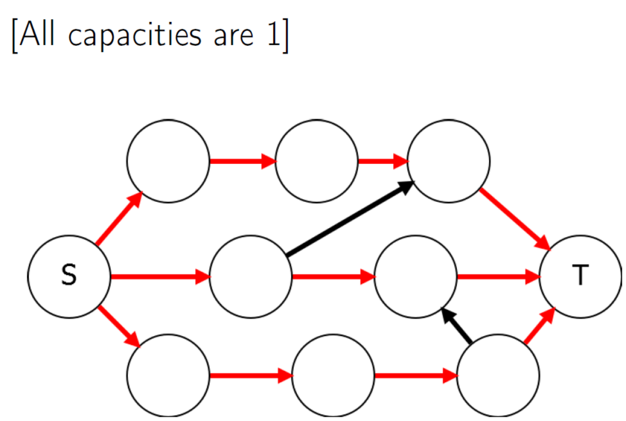
$$|f| = \sum_{\text{source a of out (e)}} f_e - \sum_{\text{source) a into (e}} f_e \qquad (۳.۱۸)$$

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

۳.۱۸ گراف باقیمانده (residual graph):

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

figure reference

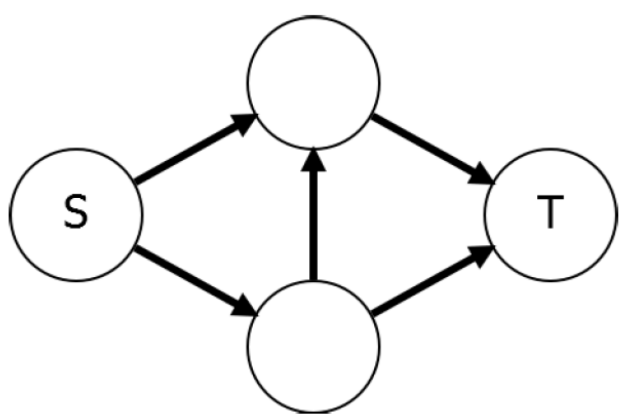


شکل ۵.۱۸: graph

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

برای مثالی دیگر گراف زیر را در نظر بگیرید.

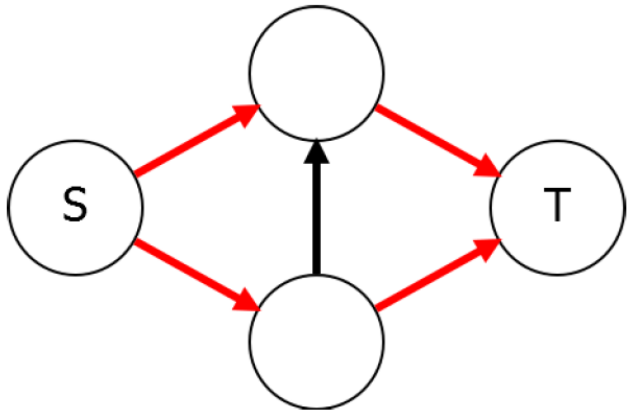
figure reference



شکل ۶.۱۸: graph

در این گراف بیشینه ی جریان برابر است با ۲.

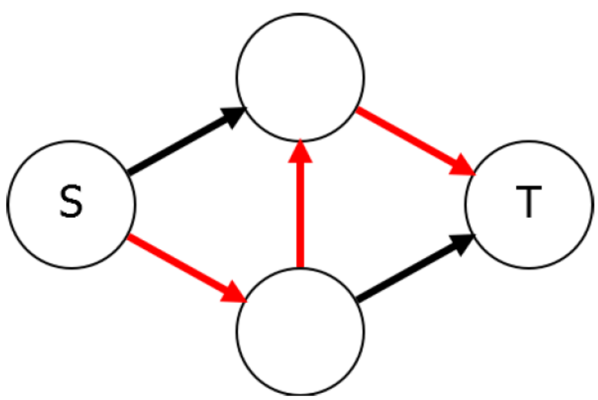
figure
reference



شکل ۱۸.۷: graph

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

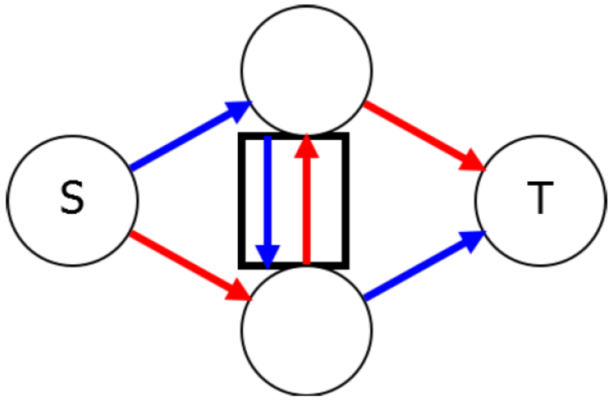
figure
reference



شکل ۱۸.۸: graph

اگر این جریان را به گراف اضافه کنیم بیشینه ی جریان ۱ خواهد شد. برای رفع این مشکل از گراف

باقیمانده استفاده می‌کنیم. به صورت زیر:



شکل ۱۸.۹: graph

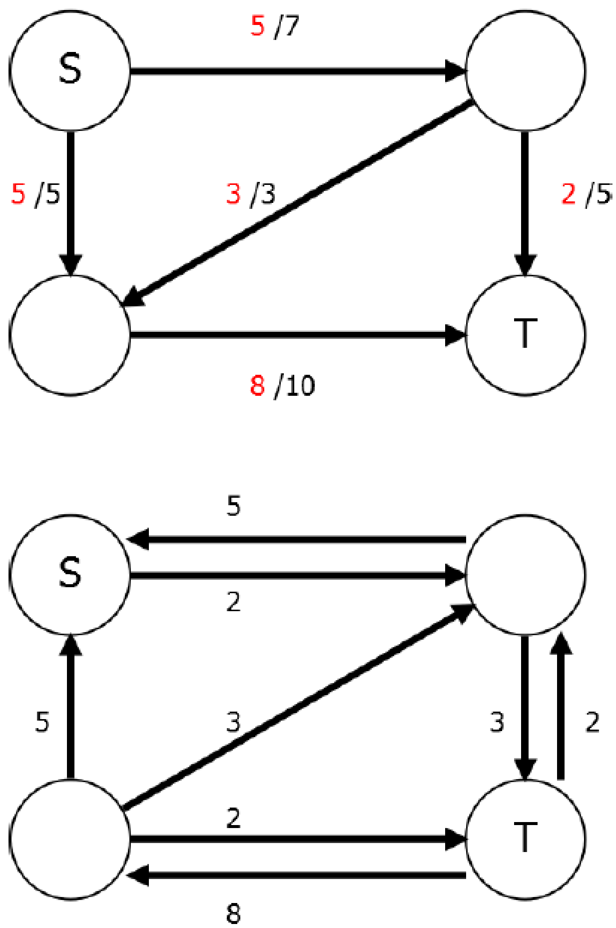
با استفاده از این گراف می توان جریان وسط را خنثی کرد.

برای یک گرافG و یک جریانf 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 می‌کشیم 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 می‌کنیم. به عبارتی دیگر: برای هر یال $e(u,v)$ روی گراف G، G_f یال های زیر را دارد:

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

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

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



شکل ۱۰.۱۸: residual network

۴.۱۸ جریان باقیمانده (residual flow):

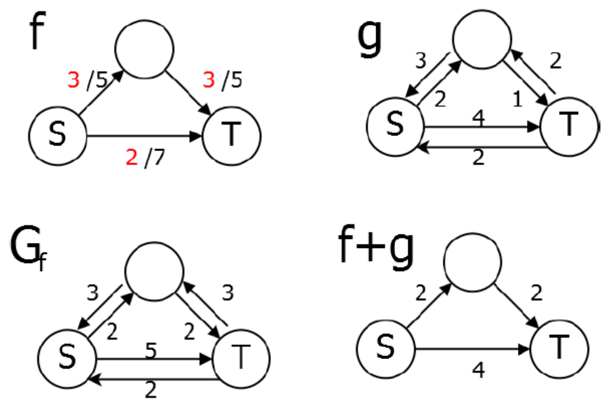
اگر گراف G و جریان f را داشته باشیم ، آنگاه هر جریان g روی G_f (گراف باقیمانده)می تواند به جریان f اضافه شود و جریان جدید نیز یک جریان معتبر روی G است. به گونه ای که:

۱. g_e به f_e اضافه می شود.

۲. $g_{e'}$ از f_e کم می شود.

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

figure reference



شکل ۱۱.۱۸: residual network

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

۱. f+g یک جریان روی گراف اصلی است.

۲. $|f+g|=|f|+|g|$

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

اثبات:

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

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

=>So f + g is a flow.

۵.۱۸ maxflow: and Mincut

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

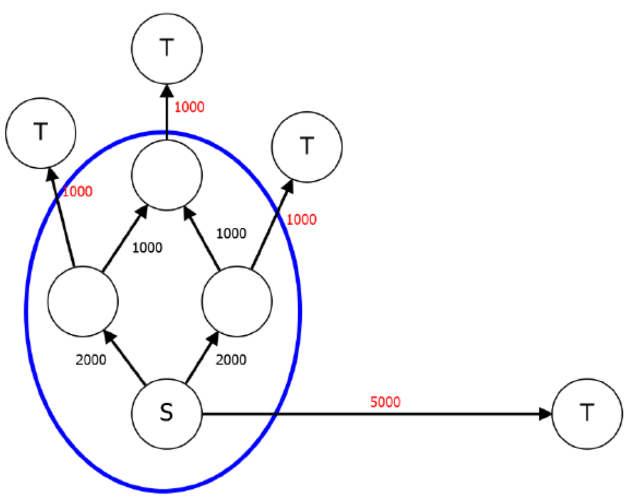
برش:(cut)

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

$$|C| = \sum_{\text{C of out e}} C_e \tag{۵.۱۸}$$

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

figure reference

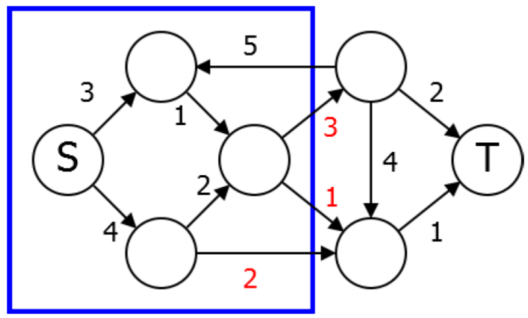


شکل ۱۲.۱۸: residual network

مثال:

figure reference

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



شکل ۱۳.۱۸: residual network

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

$$|f| \leq Maxflow Maxflow \leq |C| \tag{۶.۱۸}$$

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

قضیه: برای هر گراف G داریم:

$$maxflow|f| = mincut|C| \tag{۷.۱۸}$$

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

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

جلسه ۱۹

الگوریتم های پیدا کردن flow

آرمین غلام پور - ۱۳۹۹/۲/۷

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

۱.۱۹ Ford Fulkerson

ایده ی کلی الگوریتم به این صورت است:

۱. flow اولیه را صفر قرار دهید

۲. مکررا flow اضافه کنید

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

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

شبه کد الگوریتم * فورد فولکرسون ۴۲ :

*pseudocode

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

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

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

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

  Find c_f (p) = min (c_f (u , v) : (u , v) in p)

  ;

  for *each edge (u , v) in p* **do**

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

    ;

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

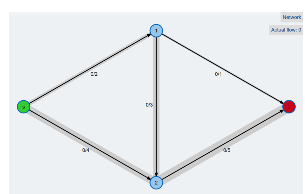
    ;

  end

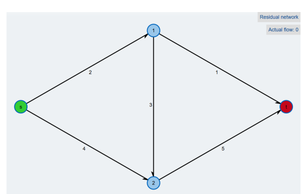
end

Algorithm 28: Ford Fulkerson Algorithm [17]

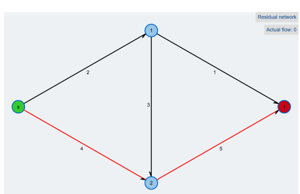
مثال زیر برای یک مساله به صورت مرحله به مرحله با الگوریتم فورد حل شده است:



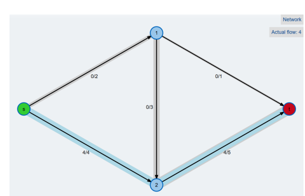
شکل ۱.۱۹: step ۱



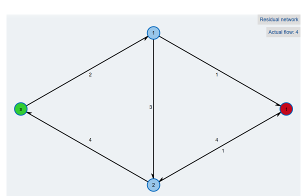
شکل ۲.۱۹: step ۲



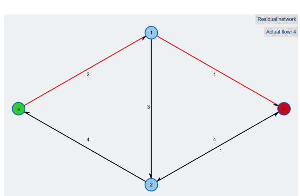
شکل ۳.۱۹: step ۳



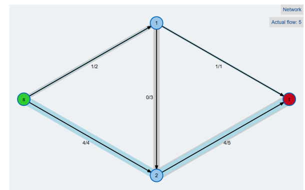
شکل ۴.۱۹: step ۴



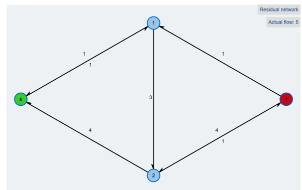
شکل ۵.۱۹: step ۵



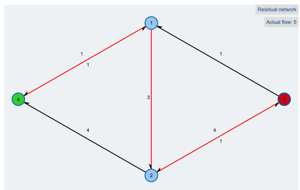
شکل ۶.۱۹: step ۶



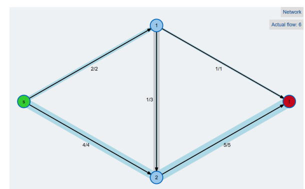
شکل ۷.۱۹: step ۷



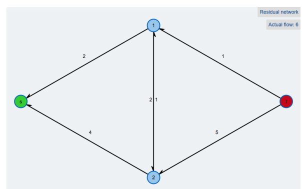
شکل ۸.۱۹: step ۸



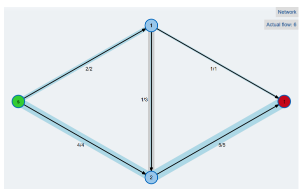
شکل ۹.۱۹: step ۹



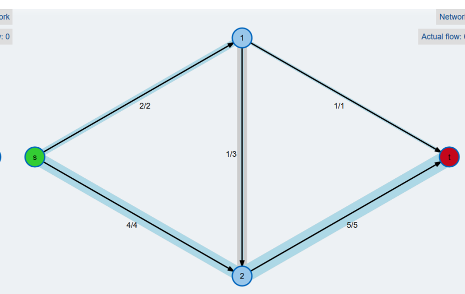
شکل ۱۰.۱۹: step ۱۰



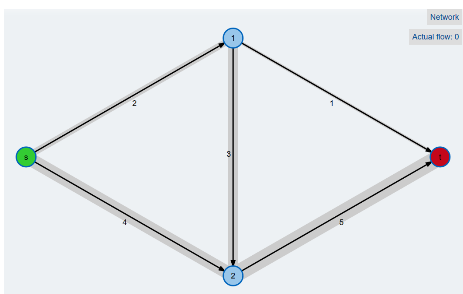
شکل ۱۱.۱۹: step ۱۱



شکل ۱۲.۱۹: step ۱۲



شکل ۱۳.۱۹: step ۱۳



شکل ۱۴.۱۹: step ۱۴

[۱۸]

برخی از ویژگی های الگوریتم فورد فولکرسون:

۱. فقط جواب های integer را پیدا میکند

۲. اردر زمانی اش O(|E||F|) هست

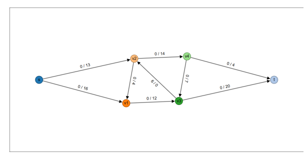
۳. اگر جریان ها مقادیر بزرگی داشته باشند، به علت استفاده از الگوریتم dfs ممکن است طول بکشد و بهینه نباشد

راه حل بهبود الگوریتم برای مقادیر بزرگ جریان:

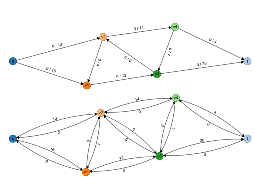
استفاده از الگوریتم Edmonds-Karp

۲.۱۹ Edmonds-Karp

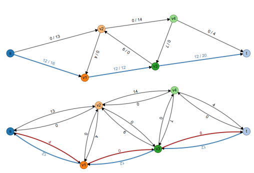
تنها فرق این الگوریتم با الگوریتم فورد، استفاده ی bfs به جای dfs است. با استفاده از این الگوریتم بهینه سازی برای جریان های بزرگ صورت میگیرد.



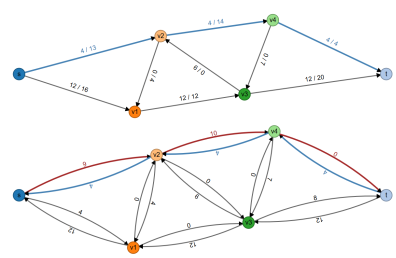
شکل ۱۵.۱۹: step ۱



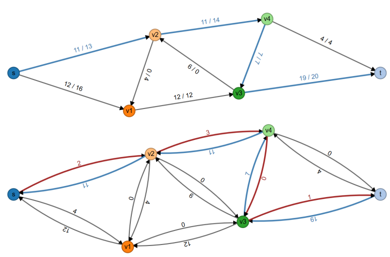
شکل ۱۶.۱۹: step ۲



شکل ۱۷.۱۹: step ۳



شکل ۱۸.۱۹: step ۴



شکل ۱۹.۱۹: step ۵

[۱۶]

جلسه ۲۰

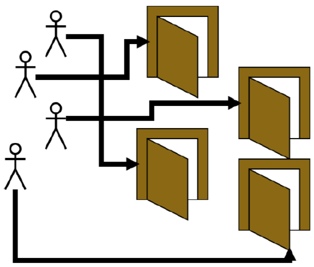
+ Applications Flow Network MeasureIt

فاطمه احمدی - ۱۳۹۹/۲/۹

جزوه جلسه ۲۰ام مورخ ۱۳۹۹/۲/۹ درس طراحی و تحلیل الگوریتم تهیه شده توسط فاطمه احمدی.

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

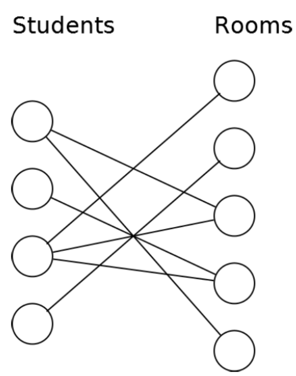
figure reference



شکل ۱.۲۰: matching dormitory

figure reference

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

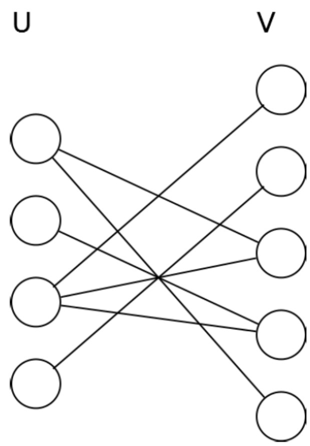


ابتدا تعریفی از گراف دو بخشی ارائه می دهیم:

۱.۲۰ گراف دو بخشی

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

figure reference

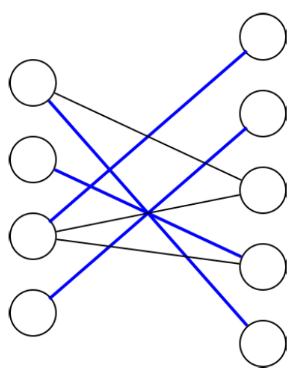


شکل ۲.۲۰: Bipartite Graph

۲.۲۰ تطابق در گراف

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

figure reference



شکل ۳.۲۰: matching

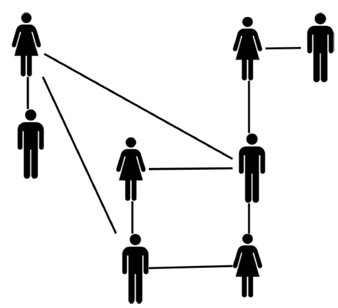
در مثال خوابگاه رابطه بین اتاق و دانشجو مانند یک گراف دو بخشی است و برای حل مسئله باید تطابق یا همان Matching گراف را بیابیم، زیرا هر دانشجویی فقط باید یک اتاق داشته باشد و هر اتاق نیز فقط متعلق به یک دانشجو است و حداکثر رضایت را نیز با پیدا کردن Matching با بیشترین یال بدست می آید به طوری که هر راس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

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

۳.۲۰ Match Making

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

figure reference

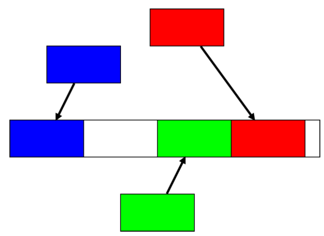


شکل ۴.۲۰: Match Making

۴.۲۰ Scheduling

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

figure reference



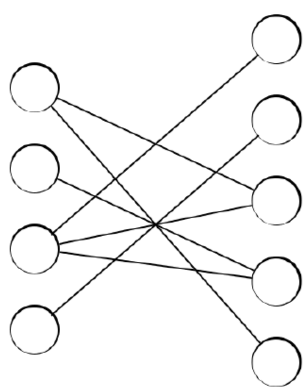
شکل ۵.۲۰: Scheduling

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

۵.۲۰ Find Maximum Matching

step۱ : مطابق شکل گراف دو بخشی مسئله را رسم می کنیم:

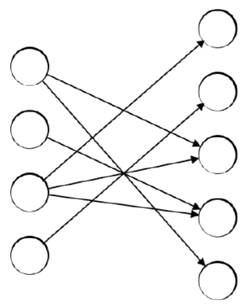
figure reference



شکل ۶.۲۰: Step۱

step۲ : مطابق شکل برای یال ها جهت تعیین می کنیم به طوری که جهت همه یال ها از U به V باشد:

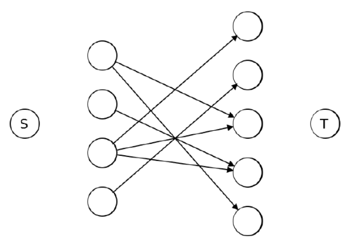
figure reference



شکل ۲۰.۷: Step۲

step۳ : مطابق شکل دو راس به عنوان مبدا یا Source و مقصد یا Target اضافه می کنیم:

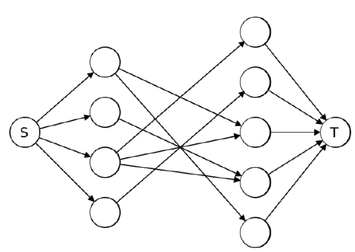
figure reference



شکل ۲۰.۸: Step۳

step۴ : مطابق شکل از راس مبدا به همه راس های U و از همه راس های V به راس مقصد یال رسم می کنیم:

figure reference



شکل ۲۰.۹: Step۴

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

الگوریتم پیدا کردن matching نیز به صورت زیر است:

Data: Bipartite Graph

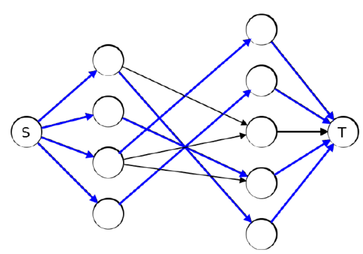
Result: Maximum Matching

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

Algorithm 29: How to find maximum matching

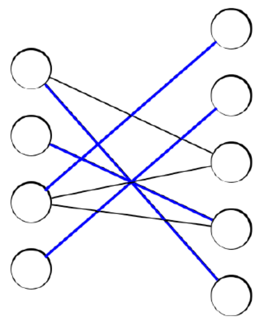
بر اساس maxflow که به دست می آید آن یال هایی که جریان آن ها یک است در matching هستند و نتیجه به صورت زیر خواهد بود:

figure reference



شکل ۲۰.۱۰: Flow

figure reference

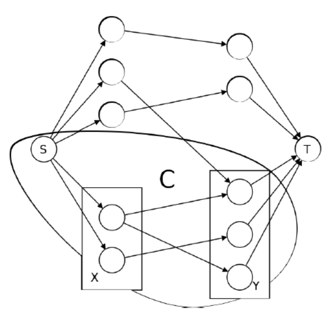


شکل ۱۱.۲۰: Matching

۶.۲۰ Maxflow-Mincut

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

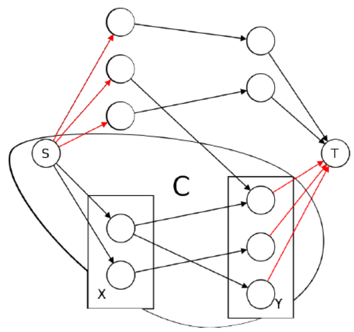
figure reference



شکل ۱۲.۲۰: Maxflow-Mincut

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

figure reference

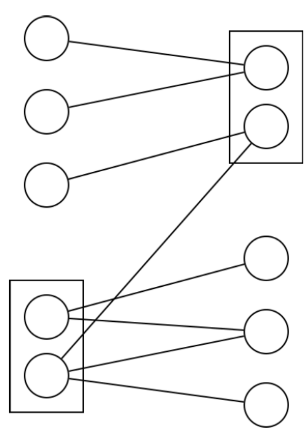


شکل ۱۳.۲۰: Maxflow-Mincut

۷.۲۰ Konig's Theorem

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

figure reference



شکل ۱۴.۲۰: Konig's Theorem

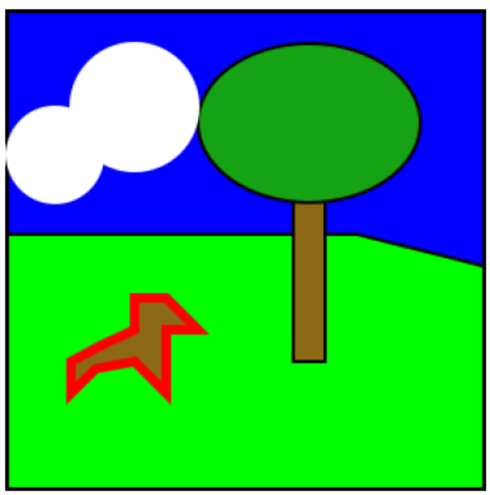
۸.۲۰ The Marriage Lemma

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

۹.۲۰ Image Segmentation

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

figure reference



شکل ۱۵.۲۰: Image

برای هر پیکسل دو مشخصه تعیین می کنیم:

a: احتمال اینکه foreground باشه.

b: احتمال اینکه background باشه.

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

را به دو دسته background و foreground

تقسیم کنیم که عبارت زیر بیشینه باشد.

figure reference

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

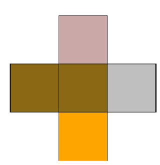
برای مثال جدول زیر را داریم:

figure reference

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

شکل ۲۰.۱۶: Example

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



شکل ۲۰.۱۷: Pixels

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

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

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

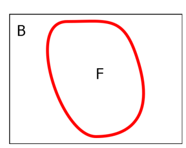
$$-\left(\sum_{v\in\mathcal{F}} b_v + \sum_{v\in\mathcal{B}} a_v + \sum_{v\in\mathcal{F},w\in\mathcal{B}} p_{vw}\right)$$

حال مسئله تغییر می‌کند و به عبارت زیر تبدیل می‌شود که این بار باید آن را مینیمم کنیم:

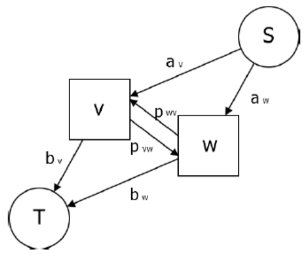
$$\sum_{v\in\mathcal{F}} b_v + \sum_{v\in\mathcal{B}} a_v + \sum_{v\in\mathcal{F},w\in\mathcal{B}} p_{vw}$$

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

figure reference

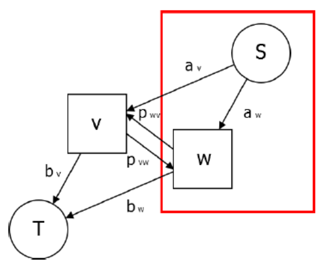


برای حل از طریق mincut باید پیکسل ها را به یک شبکه به طور مثال مطابق شکل زیر تقسیم کنیم:



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

figure reference



اندازه cut از فرمول زیر محاسبه می شود:

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

و در این صورت به این نتیجه می رسیم که هر آنچه در cut است همان مجموعه F است و هر آنچه خارج از cut است همان مجموعه B می باشد.

در ادامه الگوریتم حل این مسئله را می بینید:

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

Algorithm 30: Image Segmentation

۱۰.۲۰ Measure It

در ادامه به توضیحات مختصری از measure it می پردازیم.
developer ها معمولا در تلاشند تا کد خود را بهینه کنند ، در این راستا دانستن اینکه هر دستور یا قطعه کد چه مدت زمانی طول می کشد بسیار به آن ها کمک می کند.
MeasureIt ابزاری است که توسط آقای Vance Morrison طراحی شده است تا در این زمینه به developer ها کمک کند. به عنوان مثال دستیابی به `ReaderWriterLock` زمان بیشتری را می طلبد تا از قفل / مانیتور ساده استفاده کنید.با استفاده از MeasureIt می توان به اطلاعات دیگری از قبیل آنچه در شکل زیر آمده نیز دسترسی پیدا کرد:

figure reference

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

StatsCollection

PerformanceMeasurement.StatsCollection

Measurements

Dictionary<String,Stats> (2 items)

Key	Value
new guid [count=25]	**Stats** mean=0.009 median=0.016 min=0.000 max=0.384 sdtdev=0.014 samples=1000 **Minimum**: -4.768372E-10 **Maximum**: 0.384 **Median**: 0.016 **Mean**: 0.008735999 **StandardDeviation**: 0.01432768 **Count**: 1000
TypeOf(Guid).ToString() [count=25]	**Stats** mean=0.117 median=0.120 min=0.104 max=1.468 sdtdev=0.045 samples=1000 **Minimum**: 0.104 **Maximum**: 1.468 **Median**: 0.12 **Mean**: 0.117496 **StandardDeviation**: 0.04494942 **Count**: 1000

جلسه ۲۱

Linear Programming

فاطمه امیدی - ۱۳۹۹/۲/۱۴

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

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

شکل ۱.۲۱: مثال هایی از کاربرد Linear Programming

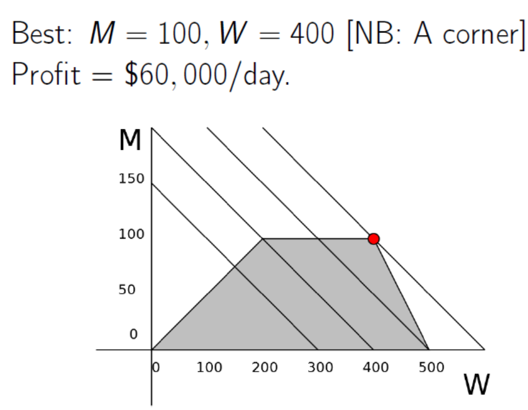
۱.۲۱ a Linear programming example

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

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

$\Leftarrow (200(W - 2M) + 600M - 100W = 100W + 200M)max$

figure reference



شکل ۲.۲۱: نمودار معادلات

۲.۲۱ وضعیت های جواب

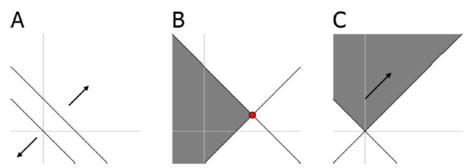
جواب معادلات میتواند به سه صورت باشد:

A. No Solution

B. One Optimum

C. No Optimum

figure reference



شکل ۳.۲۱: وضعیت های جواب

۳.۲۱ Row reduction

Basic row operations:

۱. Adding

۲. Scaling

۳. Swapping

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

شکل ۴.۲۱: adding

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

شکل ۵.۲۱: scaling

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

شکل ۶.۲۱: swapping

با استفاده از row operations ماتریس معادلات را به شکل ساده ی استاندارد شده تبدیل میکنیم تا بتوانیم به راحتی ان را حل کنیم.

Data: RowReduce(A)

Left none-zero in none-pivot row

Swap row to top of non-pivot rows

Make entry *pivot*

Rescale to make pivot 1

Subtract row from others to make other entries in column 0

Repeat until no more non-zero entries outside of pivot rows

Algorithm 31: pseudocode of row reduction

جلسه ۲۲

برنامه ریزی خطی - دوگان

غزل زمانی نژاد – ۱۳۹۹/۲/۱۶

جزوه جلسه ۲۲ام مورخ ۱۳۹۹/۲/۱۶ درس طراحی و تحلیل الگوریتم تهیه شده توسط غزل زمانی نژاد. در جهت مستند کردن مطالب درس طراحی و تحلیل الگوریتم

۱.۲۲ چندوجهی‌های محدب *

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

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

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

---

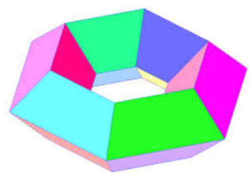
*Convex Polytopes

†hyperplane

‡facets

figure reference

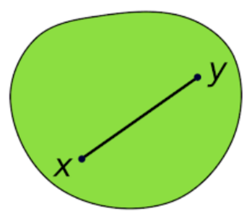
داخل آن عبور دهیم، تمامی نقاط در یک سمت واقع نشده اند (در نتیجه یک چندوجهی محدب نیست).



شکل ۱.۲۲: shape convex

figure reference

یک ناحیه در صورتی محدب است که مانند شکل ۲.۲۲ به ازای هر دو نقطه ی x و y عضو ناحیه، خطی که این دو نقطه را به یکدیگر متصل می‌کند تماما در داخل ناحیه قرار گیرد.



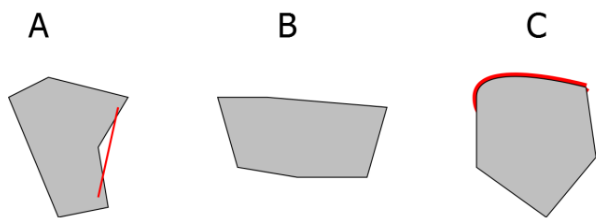
شکل ۲.۲۲: ناحیه محدب

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

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

در شکل ۳.۲۲ ناحیه A محدب نیست چون خطی که دو نقطه دلخواه داخل ناحیه را به هم متصل کرده تماما در داخل ناحیه قرار نگرفته است. ناحیه C نیز محدب نیست چون از قوس تشکیل شده است. تنها ناحیه B محدب است.

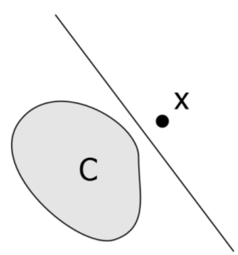
figure reference



شکل ۳.۲۲: ناحیه محدب

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

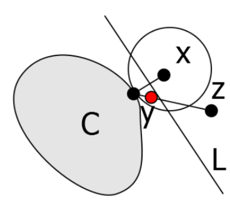
figure reference



شکل ۴.۲۲: صفحه ی جداکننده

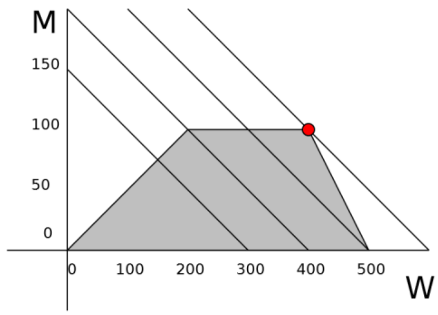
اثبات: مطابق شکل ۵.۲۲ نزدیکترین نقطه به x روی ناحیه را y می نامیم. عمود منصف خط واصل دو نقطه x و y را L می نامیم. فرض می‌کنیم نقطه z در داخل ناحیه و در سمت اشتباه خط L قرار دارد. در آن صورت yz شامل نقطه ای می شود که به x نزدیکتر است. به تناقض رسیدیم. پس حتما صفحه ی جداکننده ی چندوجهی و نقطه x یافت می شود.

figure reference



شکل ۵.۲۲: اثبات وجود صفحه ی جداکننده چندوجهی و نقطه ی خارج از آن

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



شکل ۶.۲۲: بیشترین مقدار تابع هدف روی گره واقع شده است

۲.۲۲ دوگان یک مسئله

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

$$
\begin{array}{rcl}
c_1 \cdot [a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n & \geq & b_1] \\
\ldots & & \\
+c_m \cdot [a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n & \geq & b_m] \\
\hline
w_1x_1 + w_2x_2 + \ldots + w_nx_n & \geq & t,
\end{array}
$$

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

شکل ۷.۲۲: ترکیب کردن نامعادلات با شرط $c_i \geq 0$

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

$$v_i = \sum_{j=1}^{m} c_j a_{ji} \tag{۱.۲۲}$$

برقرار باشد و

$$t = \sum_{j=1}^{m} c_j b_j \tag{۲.۲۲}$$

بیشترین مقدار ممکن را اختیار کند.

تعریف دوگان:

برنامه خطی زیر را داریم:

Minimize $v.x$

Subject to $Ax \geq b$

دوگان آن، برنامه خطی دیگری به شکل زیر است:

Maximize $y.b$

Subject to $y^T A = v$, and $y \geq b$

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

مثال: مسئله ی شاره [§]

اندازه شاره به شکل زیر است:

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

دوگان آن به صورت زیر است:

§ Flows

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

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

دوگان آن معادل پیدا کردن برش کمینه [۹] است. مثال: مسئله ی رژیم غذایی [۱۳]
فردی می خواهد از شیرینی فروشی مقداری چیزکیک و برانی تهیه کند. قیمت هر برانی ۵۰ سنت و هر چیزکیک ۸۰ سنت است. او می خواهد از مواد مغذی حداقل به میزان ۶ واحد شکلات، ۱۰ واحد شکر و ۸ واحد پنیرخامه ای مصرف کند به گونه ای که کمترین هزینه را داشته باشد. اطلاعات این مسئله به طور خلاصه به صورت زیر است:

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

شکل ۸.۲۲: اطلاعات مسئله ی رژیم غذایی

در این مسئله، دستگاه نامعادلات بدین صورت است:

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

شکل ۹.۲۲: نامعادلات مسئله ی رژیم غذایی

[۹] minimum cut

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

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

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

شکل ۱۰.۲۲: نامعادلات دوگان مسئله ی رژیم غذایی

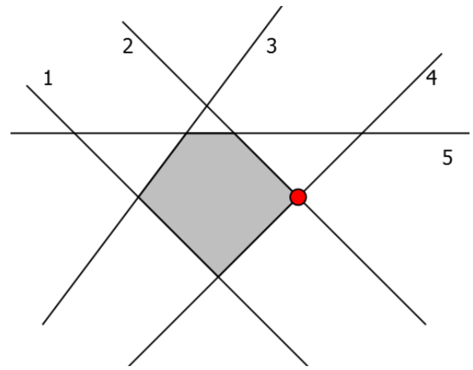
u_1 نشان دهندی قیمت هر واحد شکلات، u_2 نشان دهنده ی قیمت هر واحد شکر و u_3 نشان دهندی قیمت هر واحد پنیرخامه ای است.

۳.۲۲ Complementary Slackness

اگر مسئله LP به شکل Minimize vx subject to $Ax \geq b$ باشد و دوگان آن به صورت Maximize yb subject to $y^T A = v, y \geq 0$ باشد، آنگاه در جواب ها تنها زمانی $y_i > 0$ است که i^{th} معادله در x ، tight باشد.

سوال: اگر نقطه ی مشخص شده در شکل ۱۱.۲۲ جواب یک مسئله ی LP باشد، کدام معادلات می توانند در مسئله ی دوگان ضریب غیر صفر داشته باشند؟

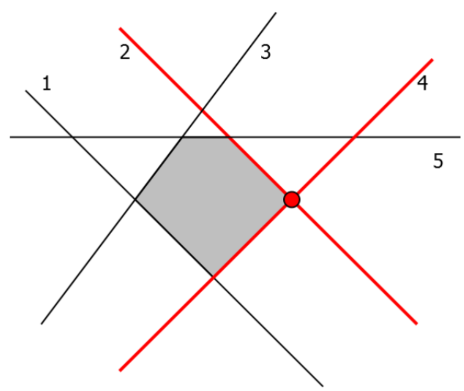
figure reference



شکل ۱۱.۲۲: Complementary Slackness example

پاسخ: مطابق شکل ۱۲.۲۲ جواب خطوط ۲ و ۴ هستند (جواب مسئله ی اصلی روی تقاطع این دو خط واقع شده است).

figure reference



شکل ۱۲.۲۲: Complementary Slackness answer

summary

۴.۲۲ خلاصه ی مطالب

Summary

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

جلسه ۲۳

Optimization و الگوریتم Simplex

محمدحسین کریمیان – ۱۳۹۹/۲/۲۱

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

۱.۲۳ اهداف آموزشی جلسه

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

۲.۲۳ انواع مسائل Optimization

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

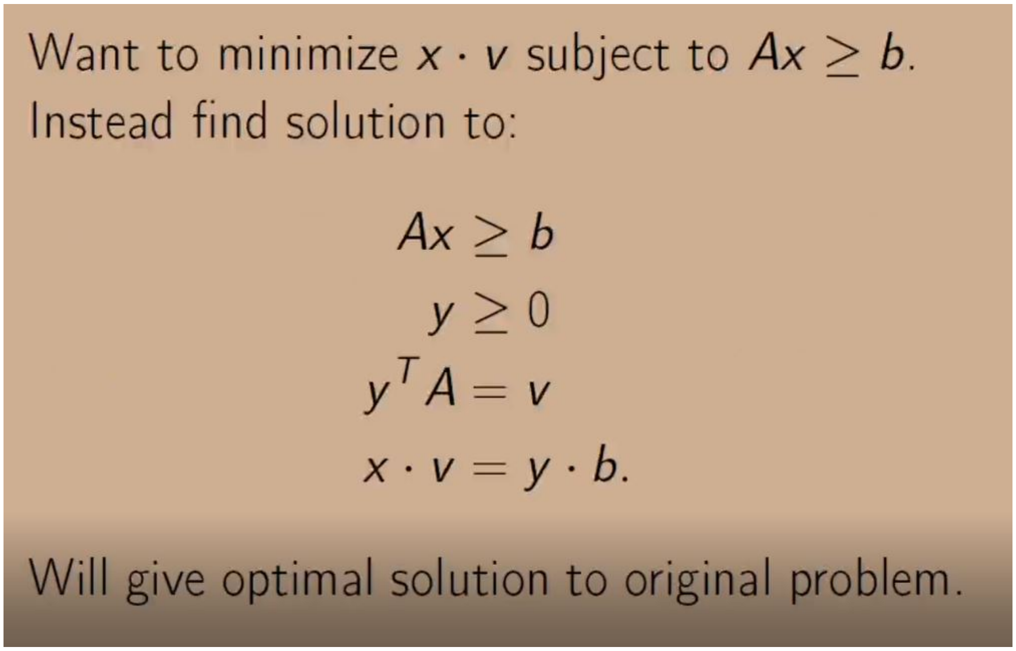
- OptimizationFromStartingPoint : راساس یک دستگاه نامعادلات و یک نقطه یا راس به عنوان نقطه ابتدایی، تابع مورد نظر را Optimize کنیم، یعنی مقدار حداکثر یا حداقل آن را بدست آوریم
-
- SoloutionFinding : در این نسخه براساس دستگاه نامعادلات جواب دلخواه که با شرایط صدق کند پیدا می کنیم.
- Satisfiability : فقط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

۳.۲۳ استفاده از حل یک فرم برای حل فرم های دیگر

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

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

figure reference



شکل ۱.۲۳: یک مثال برای به دست آوردن بهترین جواب به کمک dual

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

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

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

۴.۲۳ الگوریتم Simplex

از یک نقطه شروع می کنیم و یکی یکی یال ها را رد می کنیم و در هر مرحله به بهترین جواب اون مرحله می رسیم.با داشتن m نامعادله و n متغیر، به تعداد متغیر ها نا معادله انتخاب می کنیم و یک نقطه مشترک پیدا می کنیم.سپس یکی از معادله هایی که آن نقطه در آن صدق می کند را انتخاب می کنیم و آن را relax می کنیمتا یک خط به دست بیائریم که نقطه در آن قرار دارد.بعد از این کار از نقطه ی اولیه روی خط حرکت می کنیم و هر جا از محدوده نامعادله دیگری خارج شدیم آن نقطه را بزابز نقطه شروع جدید در نظر می گیریم و دوباره این عملیات را با نا معادلات باقی مانده انجام می دهیم.

algorithm

Data: Simplex

Start at vertex p

repeat

for *each equation through p* **do**
  relax equation to get edge
  if *edge improves objective :* **then**
    replace p by other end
    break
  else
  end
end

if *no improvement :* **then**
  return p
else
end

Algorithm 32: Algorithm Of Simplex

algorithm

Data: OtherEndOfEdge

Vertex p defined by n equations

Relax one, write general solution as p+tw (Gaussian elimination)

Relax inequality requires t ≥ 0

for *each other inequality in systen :* **do**
 | $Largest\ t\ so\ p + tw\ satisfies$

end

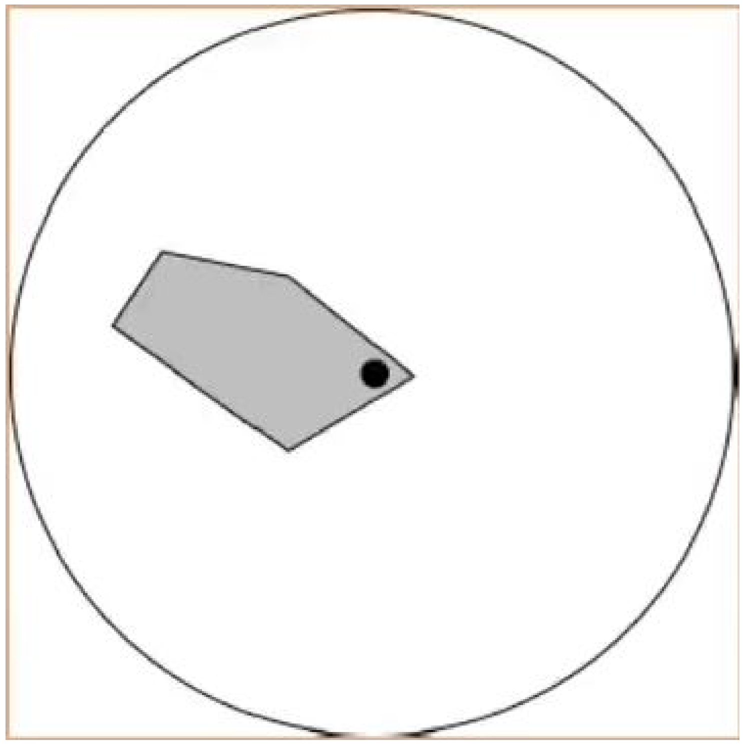
$Let\ t_0\ be\ smallest\ such\ t$

$return\ p + t_0 w$

Algorithm 33: To get a new starting point code

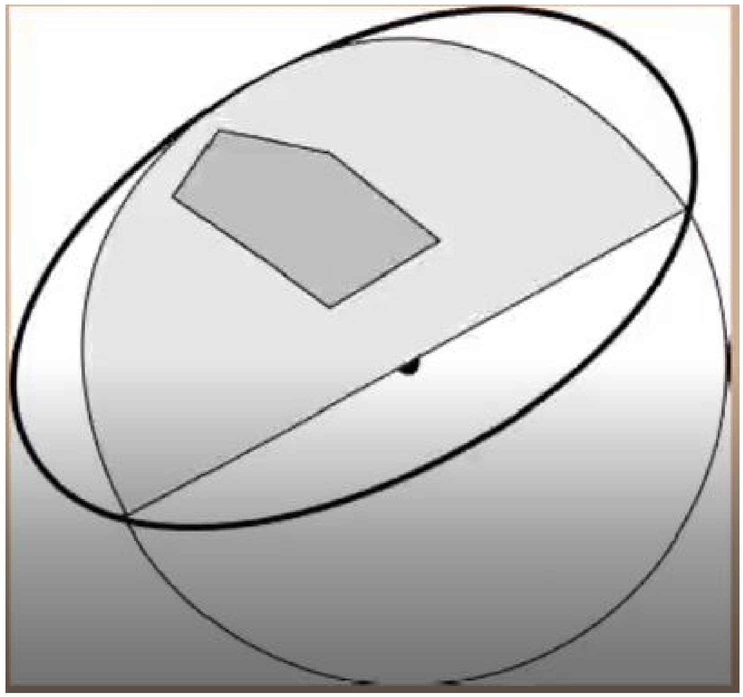
۵.۲۳ الگوریتم Ellipsoid

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



شکل ۲.۲۳: dp۱

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



شکل ۳.۲۳: dp۲

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

.

جلسه ۲۴

مسائل NP-complete

مجتبی نافذ - ۱۳۹۹/۰۲/۲۳

در این جلسه میخواهیم مسائل را بر اساس مرتبه ی زمانی دسته بندی کنیم و بیشتر روی مسائل ای از مرتبه زماانی توانی بحث کنیم.

معمولا ما مسائلی را که مرتبه زمانی کمتر از ۱۰۹ عمل دارند را میتوانیم حل کنیم.

figure reference

Polynomial vs Exponential

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

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

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

۱.۲۴ مسائل جستجو search problems

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

```
C(S,l) = true
```

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

۲.۲۴ صدق پذیری دودویی(SAT problem (Satisfiability

ورودی: یک فرمول CNF

خروجی: یک مجموعه جواب بولین به تمام متغییر های CNF برای true کردن تمامی clausesها (اگر جوابی وجود داشته باشد)

۳.۲۴ فرم نرمال ترکیب عطفی Conjunctive Normal Form

گزاره‌ای که به فرم ضرب حاصل جمع‌ها نوشته شود را، فرم نرمال CNF گوییم. به عنوان مثال عبارت زیر یک CNF می‌باشد:

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

مثالی از مسئله SAT:

مثال ۱:

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

مجموعه ی satisfiable : ۱ = x ، ۰ = y .

مثال ۲:

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

مجموعه ی satisfiable : x =۱، y = ۱، z = ۱ . or x = ۱ ، y = ۰ ، z = ۰

مثال ۳:

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

مجموعه ی satisfiable : is unsatisfiable

۴.۲۴ دسته بندی مسائل

مسائل decision: مسائلی که جواب بله و خیر دارند این پاسخ های بله و خیر در جواب مقادیر ورودی ای هستند که ادعا بر آن است که این ورودی ها پاسخ مسئله هستند
به عبارتی درستی جواب را چک می کنند.
مسائل Optimization: مسائل که بهترین داه حل را در بین تمام راه حل های feasible پیدا می کند.
راه حل استانداردی وجود دارد که مسائل oprimizationرا به decisionتبدیل کرد.که مورد بحث ما نیست.

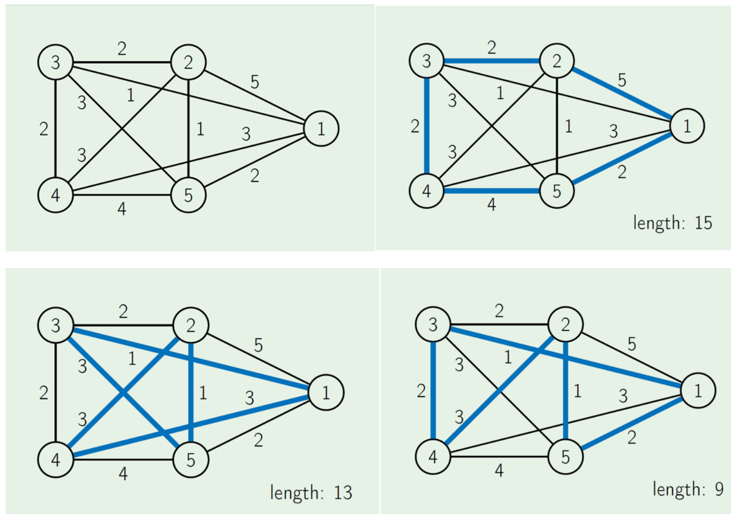
۵.۲۴ نمونه هایی از مسائل با دو ورژن ساده و سخت

مسئله: فروشنده ی دوره گرد traveling salesman problem

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

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

figure reference



شکل ۲.۲۴: مثال

این مسئله یک مسئله جستجو است. یک مجموعه راس می گیریم باید چک شود آیا همه یال ها فقط یکبار با حداکثر طول b دیده شده اند یا خیر.
مسئله TSP معمولا به صورت یک مسئله optimiziation بیان میشود اما ما به صورت decisonبیان میکنیم برای این مسئله تاکنون راه حل چند جمله پیدا نشده و راه $O(2^n n^2)$ dynamic programming میانگین زمانی را به دنبال دارد.
چک کردن تمام حالت ها از $O(n!)$ است مسائل مشابهی مانند درخت پوشای کمینه در زمان چند جمله ای قابل حل هستند.
چند نمونه را مقایسه میکنیم:

figure reference

Comparing to MST

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

شکل ۳.۲۴: مسئله فروشنده ی دوره گرد

figure reference

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

شکل ۴.۲۴: مسئله دور همیلتونی

figure reference

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

شکل ۵.۲۴: بلند ترین مسیر

figure reference

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

شکل ۶.۲۴: برنامه ریزی خطی برای عدد صحیح

figure reference

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

ب

شکل ۷.۲۴: مسئله مجموعه مستقل

۶.۲۴ مسائل NP, P

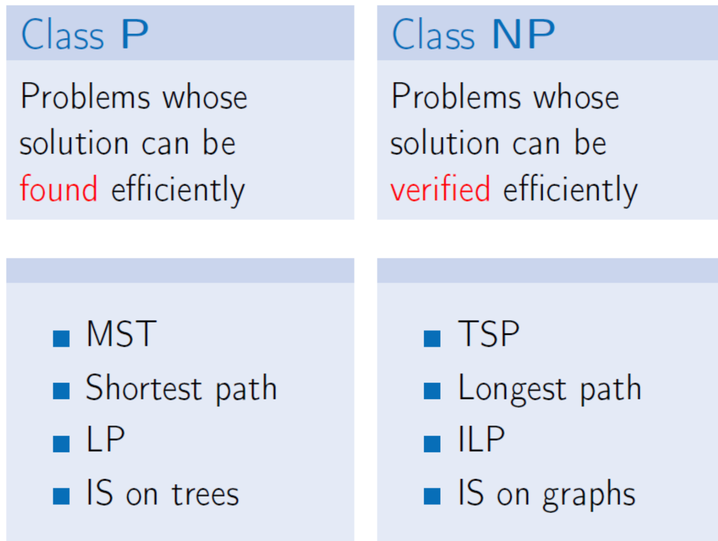
مسائل کلاس P: تمامی مسائل جستجویی که (search problem)در زمان چند جمله ای قابل حل باشند.

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

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

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

figure reference



شکل ۸.۲۴: P vs NP

مسئله باز بزرگ علم کامپیوتر با جایزه یک میلیون دلاری:

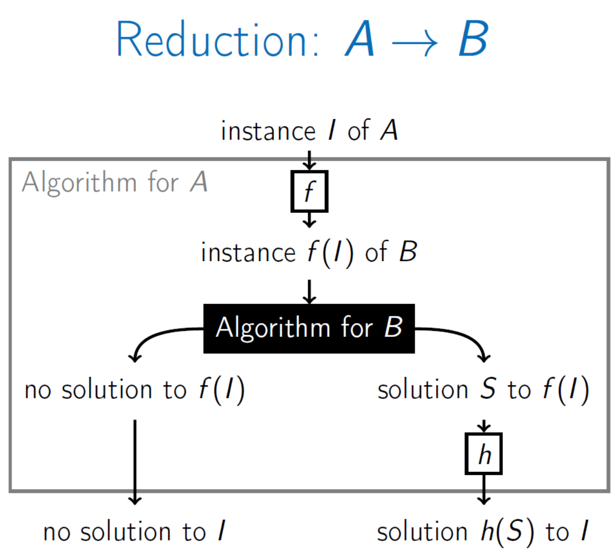
ایا مسائل Pبا مسائل NPبرابر هستند

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

۷.۲۴ کاهش Reductions

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

figure reference

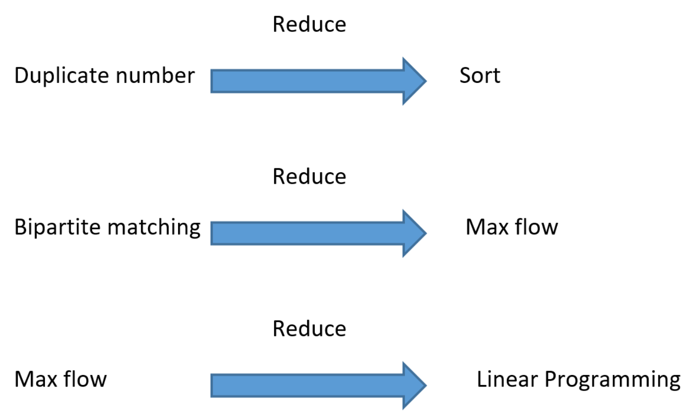


شکل ۹.۲۴: کاهش Reduction

مثال ۱ : برای حل مسئله پیدا کردن عدد تکراری در آرایه میتوان آن را به مسئله مرتب سازی کاهش داد و بعد از مرتب سازی اعداد متوالی را چک کنیم.

مثال ۲ : در اسلاید های قبل ما برای حل مسئله ی bipartitmatching آن را به maxflow کاهش دادیم

مثال ۳ : در اسلاید های قبل ما مسئله ی maxflow را به Linear Programming کاهش دادیم



شکل ۱۰.۲۴: کاهش

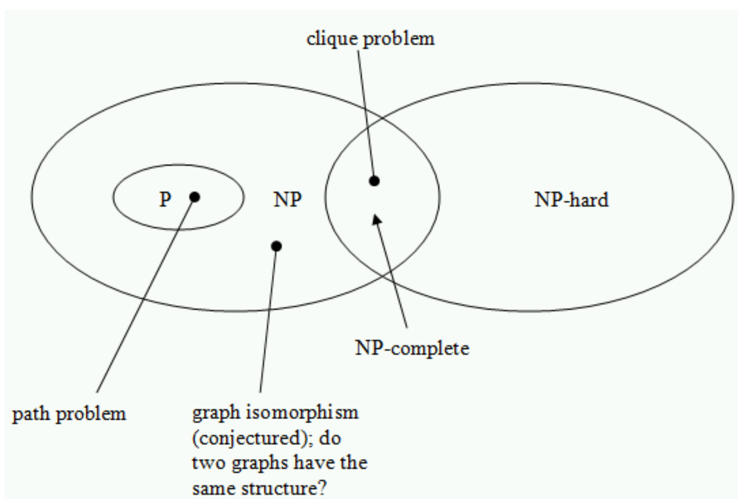
نکته: کاهش مسئله به مسئله پیچده تر امکان پذیز هست اما راه گشا نیست مثلا تبدیل مسئله میانگین آرایه به مسئله فروشنده ی دوره گرد حتی اگر امکان پذیر هم بود فقط حل مسئله را پیچیده تر می کند.

۸.۲۴ مسائل NP-hard, NP-complete

مسائل np-complete: مسائل NP ای که همه ی مسائل NP را بتوان به آن کاهش داد را NPC می نامیم
برای مثال مسئله صدق پذیری دودویی (SAT) نمونه ای از مسائل np-complete است

مسائل NP-hard مسئله H NP-hard, است زمانی که هر مسئله ی L در NP را بتوان به در زمان چند جمله ای به H کاهش داد و جواب H را زمان چند جمله ای به جواب L تبدیل کرد. اما تا به حال راه حلی بهینه ای برای حل خود H پیدا نشده است
برای مثال فروشنده ی دوره گرد (TSP) یک مسئله NP-hard میباشد
نکته: تفاوت این دو در این است که np-complete باید np باشد یعنی در زمان چندجمله ای درستی جواب را چک کند اما np-hard چه np باشد و چه نباشد فقط مسائل np به ان قابل کاهش باشند .

figure reference



شکل ۱۱.۲۴: جمع بندی

[۹] [۲۵] [۲۴] [۱۲]

جلسه ۲۵

مسائل ان پی کامل و مسئله ی صدق پذیری

سهند نظرزاده - ۱۳۹۹/۲/۲۸

۱.۲۵ مقدمه

در این جلسه کلاس مسائل ان پی کامل * ویژگی ها و ارتباط بین آن ها و مسئله ی صدق پذیری † و روش استفاده از حل کننده ی این مسئله ‡ مورد بررسی قرار گرفته است.

NP-complete*
SAT problem†
SAT solver‡

۲.۲۵ کلاس های پیچیدگی محاسباتی مسائل

۱.۲.۲۵ کلاس پی

در نظریه پیچیدگی محاسباتی، کلاس P یکی از پایه‌ترین کلاس‌های پیچیدگی است. این کلاس، شامل مسائلی است که می توان برای آن ها الگوریتمی با پیچیدگی زمانی چند جمله ای ارائه کرد. به بیانی دقیق تر می توان این کلاس را به این شکل نیز تعریف کرد: این کلاس، شامل همهٔ مسئله‌های تصمیمی است که می‌توانند با استفاده از پیچیدگی زمانی چندجمله‌ای، با کمک ماشین تورینگ پایستار حل شوند.

۲.۲.۲۵ کلاس ان پی

در نظریه پیچیدگی محاسباتی، کلاس NP یکی از بنیادی‌ترین کلاس‌ها است. NP مخفف عبارت «Non-Deterministic Polynomial» است که به زمان اجرای آن اشاره دارد. این کلاس شامل مسائلی است که بررسی درستی یا نادرستی، پاسخی به این مسائل در زمان چند جمله ای صورت می پذیرد.

۳.۲.۲۵ کلاس ان پی کامل

برای تعریف این کلاس از کلاس های نظریه پیچیدگی محاسباتی باید از مفهوم کاهش مسئله ای به مسئله ای [§] دیگر استفاده کرد. پس ابتدا این مفهوم را تعریف می کنیم.

کاهش مسئله ای به مسئله ای دیگر

اگر بتوان نمونه ای از مسئله ی A را در پیچیدگی زمانی چند جمله ای به نمونه ای برای مسئله ی B تبدیل کرد و راه حل بدست آمده از مسئله ی B را در پیچیدگی زمانی چند جمله ای به راه حل مسئله ی A تبدیل کرد. در واقع با حل مسئله ی B مسئله ی A حل شده است. به این روند کاهش مسئله ی A به مسئله ی B گفته می شود. به عنوان مثال مسئله ی محاسبه ی میانه برای تعدادی داده را می توان به مسئله ی مرتب سازی [¶] کاهش داد و سپس مقدار میانه را بدست اورد.

[§] Reduction

[¶] Sort

ویژگی های کاهش یک مسئله به مسئله ای دیگر :

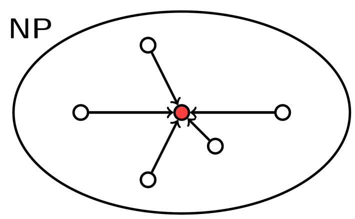
۱. اگر بتوان مسئله ی A را به مسئله ی B کاهش داد می توان نتیجه گرفت :

- اگر مسئله ی A مسئله ی سختی باشد آنگاه مسئله ی B نیز مسئله ی سختی است.
- اگر مسئله ی B مسئله ی ساده ای باشد آنگاه مسئله ی A نیز مسئله ی ساده ای است.

۲. اگر بتوان مسئله ی A را به مسئله ی B و مسئله ی B را به مسئله ی C کاهش داد می توان نتیجه گرفت که مسئله ی A نیز به مسئله ی C کاهش پیدا میکند.

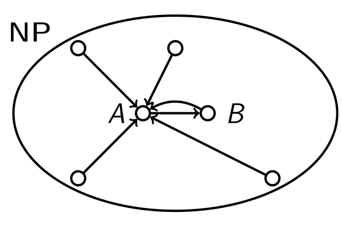
مسائل ان پی کامل

به مسائلی گفته می شود بتوان که تمام مسائل کلاس ان پی را به آن ها کاهش داد.



شکل ۱.۲۵: شکال اول مسئله ی ان پی کامل

اگر بتوان مسئله ی ان پی کامل A را به مسئله ی B کاهش داد می توان به کمک ویژگی های کاهش یک مسئله به مسئله ای دیگر این نتیجه را گرفت که مسئله ی B نیز ان پی کامل است.

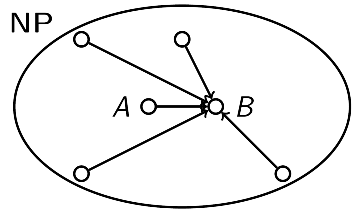


شکل ۲.۲۵: شکل دوم مسئله ی ان پی کامل

با توجه به شکل بالا همه ی مسائل به مسئله ی A کاهش پیدا کرده اند و مسئله ی A به مسئله ی B

کاهش پیدا کرده است. به کمک ویژگی دوم (تعدی، تراگذری) [*] که در بخش قبل امده است می توان اثبات کرد که تمامی مسائلی که به مسئله ی A کاهش پیدا کرده بودند به مسئله ی B نیز کاهش پیدا میکنند.

figure reference



شکل ۳.۲۵: شکل سوم مسئله ی ان پی کامل

۴.۲.۲۵ کاهش مسئله ی SAT به سایر مسائل

ابتدا صورت مسائل را بررسی می کنیم. در ادامه می خواهیم نشان دهیم که همه ی مسائل به مسئله ی SAT کاهش پیدا میکنند، پس این مسئله، یک مسئله ی ان پی کامل است و سپس نشان خواهیم داد که مسئله ی SAT به مسئله ی 3-SAT کاهش پیدا می کند و سپس مسئله ی 3-SAT را به مسئله ی Independent Set کاهش می دهیم و در آخر مسئله ی Independent Set را به مسئله ی Vertex Cover کاهش می دهیم. به دلیل این که مسئله ی SAT ان پی کامل است و به سایر مسائل کاهش پیدا کرده است پس می توان نتیجه گرفت که سایر مسائل نیز ان پی کامل هستند و می توان همه ی مسائل کلاس ان پی را به آن ها کاهش داد.

مسئله ی 3-SAT

در این مسئله یک فرمول منطقی به فرم 3-CNF [**] به عنوان ورودی داده میشود و مقدار خروجی باید تعیین کند آیا حالتی وجود دارد که با مقدار دهی به متغییر های هر عبارت منطقی حاصل این فرمول 1 بشود.

به عنوان مثال در فرمول $(x \mid y \mid z)(x \mid \overline{y} \mid z)(y \mid \overline{z})$ که یک 3-CNF است که در صورتی که با مقدار دهی متغییر های این فرمول، حاصل این فرمول 1 بشود پاسخ مسئله قابل حل بودن این فرمول است. پس باید به دنبال مقادیری برای متغییر ها باشیم که این عبارت را ارضا [††] کند در این مثال مقادیر متغییر ها $x = 1, y = 1, z = 0$ می تواند باشد که حاصل فرمول را 1 میکند. ممکن است چندین مقدار دهی مختلف بتوانند این فرمول را ارضا کنند اما در این مسئله تنها ارضا پذیری اهمیت دارد.

---

[*] Transitive property

[**] Conjunctive Normal Form (AND of ORs)

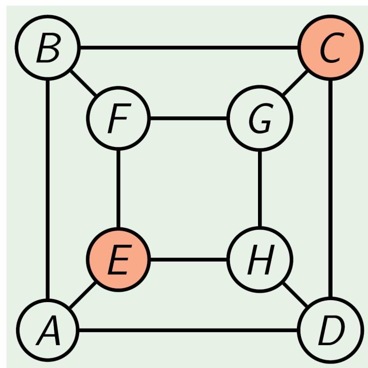
[††] Satisfy

مسئله ی Independent Set

در این مسئله یک گراف به عنوان ورودی داده می شود و مقدار خروجی باید با حداکثر تعداد گره ها به صورتی که هیچ دو گره ی انتخاب شده مجاور نباشند، برابر باشد.

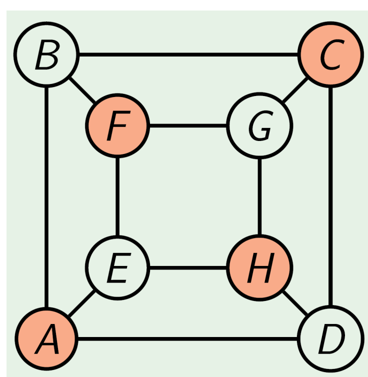
به عنوان مثال در شکل زیر مجموعه رئوس (E, C) مجاور نیستند اما حداکثر تعداد رئوس ممکن را شامل نمی شوند.

figure reference



در شکل زیر مجموعه رئوس (A, C, F, H) دو به دو مجاور نیستند و حداکثر تعداد رئوس ممکن را شامل می شود پس این مجموعه یک Independent set است.

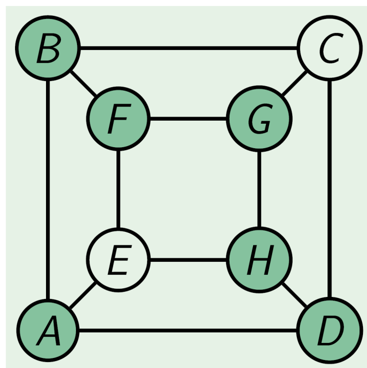
figure reference



مسئله ی Vertex Cover

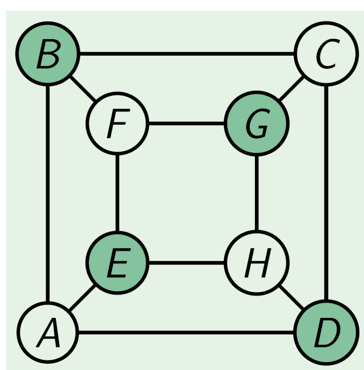
در این مسئله ی یک گراف به عنوان ورودی داده می شود و مقدار خروجی باید با حداقل تعداد گره به صورتی که حداقل یک سر از هر یال گراف در گره های انتخاب شده باشد، برابر باشد.
به عنوان مثال در شکل زیر مجموعه رئوس (A, B, D, F, G, H) حداقل یک راس از یال های گراف را می پوشانند اما این مجموعه کمترین تعداد رئوس ممکن را ندارد.

figure reference



اما در شکل زیر مجموعه ی (B, D, E, G) رئوس این مجموعه حداقل یک راس از یال های گراف را پوشش میدهد و کمترین تعداد رئوس ممکن را دارد.

figure reference



کاهش همه ی مسائل کلاس ان پی به مسئله ی SAT

به دلیل این که تمام مسائل کلاس ان پی باید توسط ماشین تورینگ غیرقطعی قابل بررسی باشند [‡‡]، مداری منطقی برای این حل هر مسئله وجود دارد که روندی از محاسبات منطقی را انجام می دهد که می توان این روند را به فرم CNF نشان داد. پس به کمک مسئله ی Circuit SAT که همه ی مسائل به ان کاهش پیدا می کنند و با کاهش آن به مسئله SAT نشان دادیم همه ی مسائل به مسئله ی SAT کاهش پیدا می کنند.

کاهش مسئله ی SAT به مسئله ی 3-SAT

به کمک جبر بول می توان هر یک از عبارات که بیش از ۳ متغییر دارد را به چندین عبارت با تعداد متغییر کمتر تبدیل کرد. برای این کار از متغییر های جدید استفاده می کنیم به عنوان مثال اگر عبارت $(x \mid y \mid z \mid w)$ به خواهیم به عبارت کوچک تری تبدیل کنیم به متغییر جدید A نیاز داریم و به کمک آن عبارت را به دو عبارت جدید $(x \mid y \mid A)$ $(\overline{A} \mid z \mid w)$. تبدیل می کنیم . به کمک جدول درستی و سایر روش های ساده سازی می توان نشان داد که حاصل این دو عبارت جدید با حاصل عبارت اولیه برابر است. به کمک این روش می توان مسئله ی SAT را به مسئله ی 3-SAT کاهش داد.

کاهش مسئله ی 3-SAT به مسئله ی Independent Set

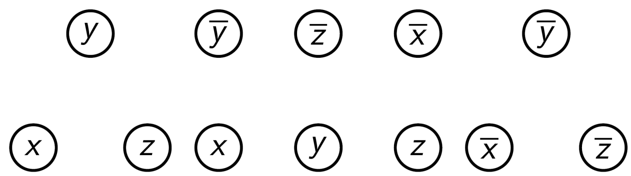
برای کاهش مسئله ی 3-SAT به مسئله ی Independent Set به ازای متغییر های داخل هر عبارت یک گره جدید در نظر می گیریم و بین این گره های یال جدیدی رسم می کنیم و بین گره هایی که متغییر آن ها نقیض هم است یال جدید دیگری رسم می کنیم. در مسئله ی 3-SAT حداقل باید یکی از متغییر های هر عبارت یک بشود به همین دلیل بین این گره ها یالی قرار دادیم تا مسئله ی Independent Set یک گره از این گره های مجاور را انتخاب کند. و بین گره های که متغییر های آن های نقیض یک دیگر است به این دلیل یال جدیدی اضافه کردیم که همزمان نمی تواند یک متغییر مقادیر $0, 1$ را بپذیرد.

اگر و تنها اگر این گراف را به عنوان ورودی به مسئله ی Independent Set بدهیم و تعداد گره های انتخاب شده توسط این مسئله برابر با تعداد عبارت های فرمول مسئله ی 3-SAT بشود، مسئله ی 3-SAT ارضا پذیر است.

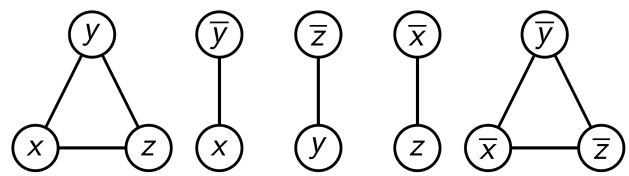
به عنوان مثال برای عبارت $(x \mid y \mid z)$ $(x \mid \overline{y})$ $(y \mid \overline{z})$ $(z \mid \overline{x})$ $(\overline{x} \mid \overline{y} \mid \overline{z})$ برای متغییر های هر یک از عبارات گره هایی مانند شکل زیر در نظر میگیریم.

figure reference

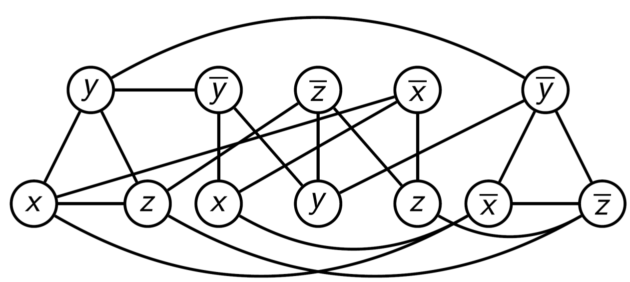
‡‡تعریف کلاس ان پی کامل



سپس مانند شکل زیر بین گره های هر یک از عبارات یال های جدیدی رسم می کنیم.



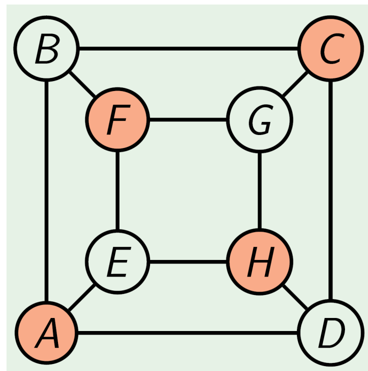
سپس مانند شکل زیر بین گره هایی که متغییر آن ها نقیض یک دیگر است یال جدید دیگری رسم می کنیم.



کاهش مسئله ی Independent Set به مسئله ی Vertex Cover

اگر و تنها اگر I یک مجموعه ی ناوابسته [§§] برای گراف $G(V,E)$ باشد، آنگاه $V-I$ یک مجموعه پوشاننده ی گره [¶¶] است.
به عنوان مثال در شکل زیر اگر مجموعه ی $I=\{A,C,F,H\}$ یک مجحوعه ی ناوابسته باشد، آنگاه مجموعه ی $V-I=\{B,D,E,G\}$ یک مجموعه ی پوشاننده ی گره است.

[§§] Independent Set
[¶¶] Vertex Cover



به کمک این قضیه می توان به راحتی مسئله ی Independent Set را به مسئله ی Vertex Cover کاهش داد. و برعکس.

۳.۲۵ حل کننده ی مسئله ی SAT

یکی از راه های حل کردن مسائل کلاس NP کاهش دادن این مسائل به مسئله ی SAT است و سپس استفاده از SAT Solver های موجود است.

۱.۳.۲۵ پازل سودوکو

به کمک SAT Solver می توانیم با توجه به قوانین پازل سودوکو و اعداد درون آن این نتیجه را بگیریم که آیا این پازل قابل حل است یا نه. برای این که این مسئله را به مسئله ی SAT کاهش دهیم کافیست که قوانین آن را به صورت فرمولی منطقی و به فرم CNF تبدیل کنیم.
پازل سودوکو ۳ قانون زیر را دارد:

۱. هر یک از اعداد ۱ تا ۹ در سطر i ام که $1 \leq i < 10$ است باید یک بار ظاهر بشوند.

۲. هر یک از اعداد ۱ تا ۹ در ستون i ام که $1 \leq i < 10$ است باید یک بار ظاهر بشوند.

۳. هر یک از اعداد ۱ تا ۹ در یک بلوک ۳ در ۳ باید یک بار ظاهر بشوند.

با توجه به این قوانین برای هر خانه از جدول ۹ متغییر با اندیس های ۱ تا ۹ به شکل زیر تعریف می کنیم:
اگر و تنها اگر $x_{ijk} = 1$ که $1 \leq i, j, k \leq 9$ باشد آنگاه، عدد خانه ی سطر i ام و ستون j ام برابر با k است.
به همین دلیل باید یکی از این ۹ متغییر می تواند یک باشد زیر در یک خانه از جدول نمی توان هم زمان چند عدد ظاهر بشود. پس باید این شرط را نیز در فرمول CNF اضافه کنیم.
برای ساخت فرمول CNF ابتدا تابع $ExactlyOneOf(l_0,\ l_1,\ l_2,\ ...,\ l_n)$ را تعریف میکنیم.

تابع ExactlyOneOf

این تابع از بین مقادیر داده شده شرایطی را به فرم CNF ایجاد میکند که یک و فقط یک متغییر از بین متغییر های پاس داد شده به تابع بتواند یک به شود. این تابع با اضافه کردن عبارت $(l_0 \mid l_1 \mid l_2 \mid ... \mid l_n)$ به فرمول باعث می شود که حداقل یکی از این متغییر ها یک شود و با اضافه کردن عبارات $(\overline{l_i} \mid \overline{l_j})$ که $0 \leq i, j \leq n; i \neq j$ است به فرمول باعث می شود که هیچ دو متغییری هم زمان یک نشود یا به نحوی دیگر می توان گفت با اضافه کردن این عبارات به فرمول حداکثر یک متغییر می تواند یک باشد.
به عنوان مثال اگر تابع $ExactlyOneOf(l_0,\ l_1,\ l_2)$ صدا زده بشود عبارت
$(l_0 \mid l_1 \mid l_2)(\overline{l_0} \mid \overline{l_1})(\overline{l_1} \mid \overline{l_2})(\overline{l_2} \mid \overline{l_1})$ برگردانده میشود.

اکنون به کمک این تابع فرمول پازل سودوکو را می سازیم.

طبق متغییر هایی که تعریف کردیم به این نتیجه رسیدیم که باید تنها یکی از متغییر های هر خانه یک باشد پس تابع $ExactlyOneOf$ را به شکل زیر و برای متغییر هایی هر خانه سطر i ام و ستون j ام صدا می زنیم:

$$ExactlyOneOf(x_{ij1},\ x_{ij2},\ x_{ij3},\ ...,\ x_{ij9})$$

سپس طبق قانون اول باید هر عدد یک بار در هر سطر ظاهر بشود پس تابع $ExactlyOneOf$ را به شکل زیر برای متغییر های عدد k و سطر i ام صدا میزنیم:

$$ExactlyOneOf(x_{i1k},\ x_{i2k},\ x_{i3k},\ ...,\ x_{i9k})$$

سپس طبق قانون دوم باید هر عدد یک بار در هر ستون ظاهر بشود پس تابع $ExactlyOneOf$ را به شکل زیر و برای متغییر های ستون j ام و عدد k صدا می زنیم:

$$ExactlyOneOf(x_{1jk},\ x_{2jk},\ x_{3jk},\ ...,\ x_{9jk})$$

سپس طبق قانون سوم هم برای هر یک از متغییرهای بلوک ها و هر عدد یک بار تابع را صدا می زنیم. و برای خانه های که دارای مقدار اولیه هستند متغییر مربوط باید برابر با یک باشد به همین دلیل یک عبارت با همان متغییر به فرمول اضافه می کنیم. به کمک این فرمول بدست آمده و SAT Solver می توان امکان پر شدن یا نشدن این پازل را با توجه به اعدادی که از ابتدا در آن بوده اند تشخیص داد.

جلسه ۲۶

حل کردن مسائل NP-complete (حالت های خاص)

هادی شیخی - ۱۳۹۹/۲/۳۰

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

۱.۲۶ مسائل NP-Complete

همانطورکه در جلسات پیش توضیح داده‌شد مسائل NP معادل یک Search Problem هستند. حال برای این مسائل یا راه حل چندجمله‌ای وجود دارد یا اینکه تا به حال موفق به یافتن راه‌حل با پیچیدگی چندجمله‌ای

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

۲.۲۶ حل کردن مسائل NP-Complete

در حال حاضر الگوریتم های ارائه شده برای حل مسائل NP-Complete نیاز به پیچیدگی محاسباتی Super Polynomial دارد و همچنان اثبات نشده‌است که راهی با پیچیدگی کمتری وجود دارد یا خیر. به طور کلی برای مسائل محاسباتی از روش هایی می‌توان استفاده کرد که منجر به ایجاد الگوریتم‌هایی با پیچیدگی محاسباتی کمتر می‌شود [۳۱].

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

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

۳.۲۶ ۲-SAT یک حالت خاص [۱۹]

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

۱.۳.۲۶ ساخت Implication Graph

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

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

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

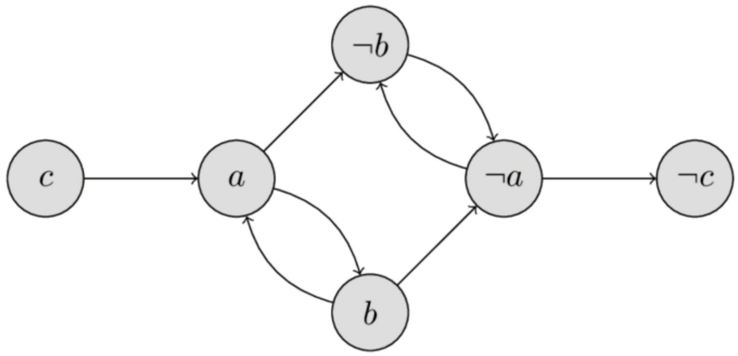
$$(X + Y) ::= \bar{X} \to Y, \bar{Y} \to X$$

و با توجه به تعریف Implication Graph ، اگر گرافی داشته باشیم که برای هر متغیر X تعریف شده در SAT دو رأس $\bar{X}$ و X را شامل شود برای هر جمله (X+Y) باید بالی از $\bar{X}$ به Y و یالی از $\bar{Y}$ به X باشد. حال با انجام مراحل فوق برای هر جمله از ۲-SAT گراف مربوط به آن را می‌سازیم.
برای مثال [۷] ۲-SAT زیر را در نظر بگیرید.

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

گراف مربوط به این مسأله به صورت زیر است.

figure reference



شکل ۱.۲۶: گراف مربوط به 2-SAT

۲.۳.۲۶ پیدا کردن مولفه‌های قویاً همبند (Strongly Connected Components)

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

$$l_1 \rightarrow l_3 \rightarrow ... \rightarrow l_2$$

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

$$l_i \rightarrow l_j$$

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

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

۳.۳.۲۶ مقدار دهی متغیرها

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

۱. مرتب کردن مولفه‌ها بر اساس Topological Sort

۲. شروع از آخر و اگر هیچ یک از متغیرها مقدار دهی نشده‌باشد ، به همه متغیرها مقدار ۱ را می‌دهیم و مقدار متغیر متضاد هر کدارم را ۰ قرار می‌دهیم.

پس از انجام مراحل فوق مقدار هر یک از متغیر ها معلوم است.

۴.۳.۲۶ اثبات درستی

هرگاه متغیری ۱ شود آنگاه همه متغیرهایی که از آن قابل دسترسی هستند هم برابر ۱ می‌شود که بدان معنی است که همه Implication ها satisfy می‌شوند.
به همین ترتیب اگر متغیری ۰ باشد آنگاه همه متغیرهایی که قابل دسترسی ار آن هستند نیز برابر ۰ می‌شوند. (Skew-Symmetry) [۳۰]

جلسه ۲۸

حالت‌های مواجهه با مسائل NP-Complete

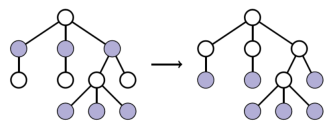
ملیکا احمدی‌رنجبر - ۱۳۹۹/۳/۶

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

۱.۲۸ حالت خاص Independent Set

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

برگ‌هایش را انتخاب نمود.



شکل ۱.۲۸: Move Safe

۴۲ *

O(|T|);

initialization;

while *T is not empty* **do**

  take all the leaves to the solution ;

  remove them and their parents from T;

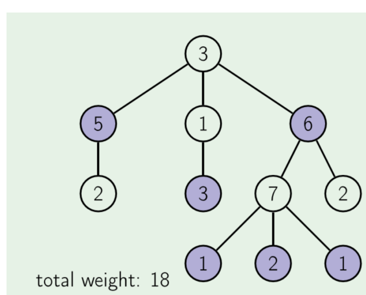
end

return the constructed solution;

Algorithm 34: PartyGreedy

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

figure reference



شکل ۲.۲۸: نمونه‌ای از محاسبات این سوال روی درخت

کد این قسمت را می‌توانید در قسمت زیر مشاهده کنید.

*pseudocode

$$\max \begin{cases} w(v) + \sum_{\substack{\text{grandchildren} \\ w \text{ of } v}} D(w), \sum_{\substack{\text{children} \\ w \text{ of } v}} D(w) \end{cases}$$

```
Data: Function FunParty(v)
if D(v)= ∞ then
    if v has no children then
        D(v) ← w(v)
    else
        m1 ← w(v)
        for all children u of v do
            for all children w of u do
                m1 ← m1 + FunParty(w)
            end
        end
        m0 ← 0
        for all children u of v do
            m0 ← m0 + FunParty(u)
        end
    end
    D(v) ← max(m1, m0)
else

end
return D(v)
```

Algorithm 35: Main code

۲.۲۸ 3-Satisfiability

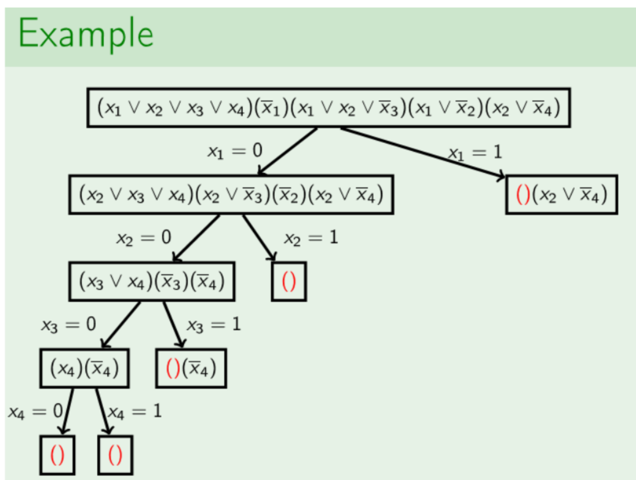
۱.۲.۲۸ Backtracking

یکی از روش‌های حل به این شکل است. در این روش برای مثال اگر 3-satداشته باشیم با توجه به شرایط مثال داده شده به دلخواه یکی از مقادیر ۰ یا ۱ را به یکی از Variable ها می‌دهیم و نسبت به آن مقدار داده

شده به ترتیب عملیات‌ها و پرانتزهای ساده‌تری را خواهیم داشت و به همین ترتیب به تمام Variableها مقدار می‌دهیم.برخی قابل قبول نیستند برای همین یک شاخه به عقب بازمی‌گردیم و مقدار مخالف آن مقداری که قابل قبول نبود می‌دهیم و جلو می‌رویم . هرگاه به جایی رسیدیم که مقادیر قابل قبول نبودند تنها کافی است به شاخه‌های قبل برگردیم و مقادیر مخالف مقدار قبل را امتحان کنیم و تا زمانی ادامه می‌دهیم(چه به سمت پایین چه بازگشت در صورت قابل قبول نبودن) که به جواب برسیم(منظور از بالا و پایین رفتن روی درختی است که ایجاد می‌شود.)

```
Goal : Avoid going through all 2 assignments
```

figure reference



شکل ۳.۲۸: Sapmle Of The Tree Which is Made

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

Local Search ۳.۲۸

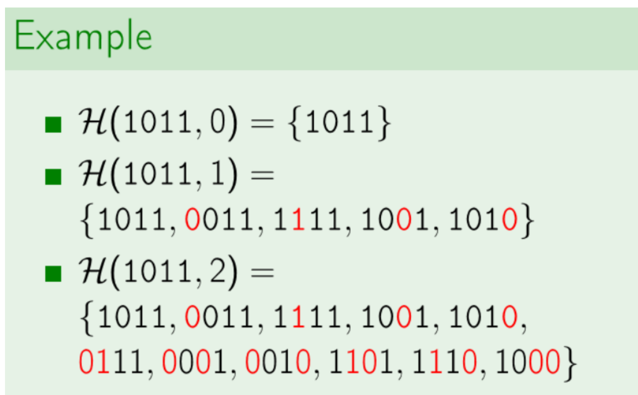
۱.۳.۲۸ تعریفات

Hamming Distance: تعداد بیت‌هایی که دو عدد باینری در آن تفاوت دارند.

Hamming Ball: با مرکز a و شعاع r مجموعهٔ اعداد باینری است که حداکثر به اندازهٔ r تفاوت دارند.

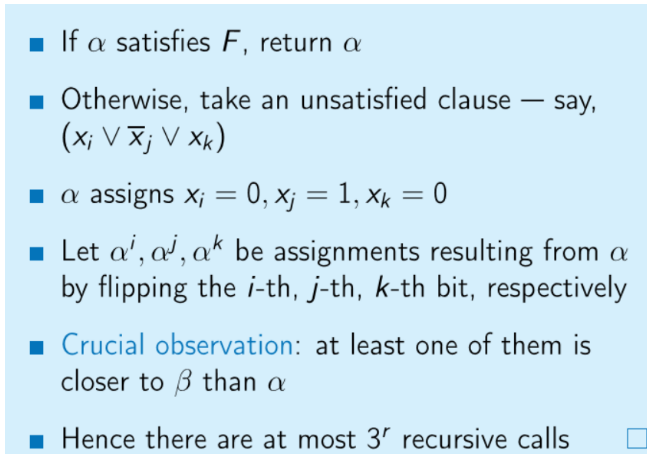
منظور از اعداد باینری تعدادی متغیر است که مقادیر ۰ و ۱ می‌پذیرند . در واقع منظور Assignments است.

figure reference



شکل ۴.۲۸: Hamming Ball And Hamming Distance

figure reference



شکل ۵.۲۸: Proof

اگر برای متغیرهایی تعداد ۰ ها بیشتر باشد بنابراین Hamming Ball را در فاصلهٔ N/۲ برای N متغیر با مقدار ۰ پیدا می‌کنیم و می‌دانیم پاسخ در آن است و اگر تعداد ۱ ها بیشتر بود تمام متغیرها را ۱ در نظر می‌گیریم و همان کار را انجام می‌دهیم.

Data: CheckBall(F,α,r)

if α *satisfies* F **then**
| *return* α

else

end

if *r=0* **then**
| *return "not found"*

else

end

$x_i, x_j, x_k \leftarrow variables\ \ of\ \ unsatisfied\ \ clause$

$\alpha^i, \alpha^j, \alpha^k \leftarrow\ \ \alpha\ \ with\ \ bits\ \ i, j, k\ \ flipped$

CheckBall(F,α^i,r-1)

CheckBall(F,α^j,r-1)

CheckBall(F,α^k,r-1)

if *a satisfying assignment is found* **then**
| return it

else
| *return "not found"*

end

Algorithm 36: Hamming Ball And Hamming Distance

۴.۲۸ Traveling Salesman Problem (TSP)

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

Dynamic Programming

الگوریتم بدیهی : تمام حالت‌ها را چک کنیم که برابر است با Cycles !(n - 1). اما می‌توان از Dynamic Programming استفاده کرد که دارای زمان $O(n^2.2^n)$است.

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

Data: TSP(G)

$C(\{1\},1) \leftarrow 0$

for s $from$ 2 to n **do**

  for all $1 \in S \subseteq \{1,..,n\} of$ $size$ s **do**

    C(S,1) ← ∞ ;

    for all $i \in S, i \neq 1$ **do**

      for all $j \in S, j \neq i$ **do**

        $C(S,i) \leftarrow min\{C(S,i), C(S-\{i\},j) + d_{ji}\}$

      end

    end

  end

end

$return$ $min_i\{C(\{1,...,n\},i) + d_{i,1})\}$

Algorithm 37: Hamming Ball And Hamming Distance

ترتیب چک کردن تمام زیرمجموعه ها(2^n) یک نگاشت به صورت زیر دارد.

k <-> i : i-th bit of k is 1

Branch-and-bound

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

جلسه ۲۹

Coping with NP-completeness

احمد بهمنی - ۱۳۹۹/۵/۱۳

۱.۲۹ مقدمه

برای حل کردن برخی مسائل NP-complete مانند 2-SAT یا Independent set در درخت، می توان راه حل چند جمله ای پیدا کرد که حالت های خاصی از این مسائل هستند. برای بقیه ی مسائل، که راه حل توانی* دارند، می توان order محاسبه جواب دقیق را تا حد خوبی کاهش داد. در این جلسه به پیدا کردن این راه حل ها می پردازیم.

Exponential*

۲.۲۹ TSP[†]

۱.۲.۲۹ تعریف مسئله

در مسئله traveling sales person هدف پیدا کردن دوری است که از هر رأس دقیقا یکبار رد شده باشیم، به صورتی که حداکثر مسیر طی شده از مقدار ثابتی بیشتر نباشد.

۲.۲.۲۹ راه حل بدیهی[‡]

ساده ترین راه برای حل این مسئله این است که تک تکِ جایگشت های هر رأس گراف را محاسبه کنیم وکوتاهترین مسیر را به عنوان جواب مسئله در نظر بگیریم. این راه از `O((n - 1)!)` می باشد. مشکل این روش این است که برخی از زیرمسئله ها را چندبار به صورت تکراری محاسبه می کنیم که با استفاده از dynamic programming می توان به الگوریتمی بهینه تر رسید.

۳.۲.۲۹ dynamic programming

همان طور که گفته شد در روش Brute Force برخی از زیرمسئله ها را چندبار تکراری محاسبه می کنیم که با حل کردن این زیرمسئله ها و استفاده از آن ها برای سرعت بخشیدن به الگوریتم حل، می توان مسئله TSP را با $O(n^2.2^n)$ حل کرد.

اگر S را زیر مجموعه ای از راس های ۱ تا n در نظر بگیریم، و i یکی از راس های موجود در S باشد، تابع C(S,i) را اینگونه تعریف می کنیم که برابر است با طول کوتاهترین مسیری که از ۱ شروع شده و به i ختم شود و از تمام رأس های موجود در S دقیقا یک بار رد شود. [§]

برای پیدا کردن رابطه ی بازگشتی، رأس یکی مانده به آخر j که در مسیر ۱ تا i که کوتاهترین مسیر است و از همه ی رأس های موجود در S رد شده را در نظر بگیرید. مسیری که از ۱ به j می رود، کوتاهترین مسیری است که همه ی رأس های عضو S به جز i را دقیقا یک بار رد می کند. در نتیجه:

$$C(S,i) = min(C(S - \{i\}, j) + d_{ij})$$

که min شامل تمام j هایی است که عضو s باشد، به جز حالتی که $j = i$ است

† Traveling sales person

‡ Brute Force

§ C({1}, 1) = 0 and C(S, 1) = +∞ when |S| > 1

پس باید همه ی زیر مجموعه های $\{1, ..., n\}$ را محاسبه کنیم، به صورتی که وقتی به $C(S, i)$ رسیدیم، مقدار $C(S\{i\}, j)$ را محاسبه کرده باشیم.

C({1}, 1) ← 0;
for *s from 2 to n* **do**
  for *all 1   S   {1, . . . , n} of size s* **do**
    C(S, 1) ← +∞
    for *all i   S, i = 1* **do**
      for *all j   S, j = i* **do**
        $C(S, i) \leftarrow minC(S, i), C(S\{i\}, j) + d_{ji}$
      end
    end
  end
end
return $min_i\{C(\{1, ..., n\}, i) + d_i, 1\}$

Algorithm 38: Dynamic programming

۲۹.۲.۴ Branch and bound

با استفاده از این روش می توان روش dynamic programming که گفته شد را بهینه کرد. به این صورت که در هر مرحله، بهترین جواب را نگه می داریم. وقتی به دنبال مسیر کوتاه تری هستیم، با پیشرفت هر رأس، طول کلی مسیر فعلی را با کوتاه ترین مسیر پیدا شده مقایسه می کنیم. اگر طول مسیر ناقص پیدا شده کمتر از کوتاه ترین مسیر بود، ادامه می دهیم. در غیر این صورت با توجه به اینکه یال منفی نداریم، میفهمیم که مسیر انتخاب شده در انتها طولانی تر از مسیر بهینه است. پس به دنبال مسیر دیگری می گردیم. روش گفته شده، ساده ترین راه در استفاده از lower bound می باشد.

۵.۲.۲۹ Metric TSP

این حالت خاص از TSP به این صورت بیان می شود:
گرافی غیر جهتدار داریم که یال منفی ندارد و نامساوی مثلثی در آن صدق می کند. یعنی با در نظر گرفتن u, v, w که عضو V باشند داریم:

$$d(u,v) + d(v,w) >= d(u,w)$$

هدف پیدا کردن TSP با الگوریتم ۲-approximate در این گراف است.

می دانیم که با داشتن گراف G با شرایط ذکر شده، $MST(G) <= TSP(G)$ است (با حذف کردن بزرگترین یال موجود در TSP ، دور نداریم و درخت حاصل همان MST است).

به این صورت می توانیم از MST به TSP برسیم که اگر بین هر دو رأسی که در MST بین آنها یالی وجود دارد، یال دیگری اضافه کنیم، با توجه به رابطه ی نامساوی ذکر شده ای که در گراف وجود دارد، می توان یال دیگری پیدا کرد که درخت MST را به TSP تبدیل می کند. که چون یال سوم اضافه شده فقط وزن کل را کاهش می دهد (رابطه ی نامساوی مثلثی) و اینکه با دو برابر کردن تعداد یال ها در MST وزن کل را ۲ برابر کرده ایم، پس جواب بدست آمده حداکثر ۲ برابر جواب بهینه است.

۶.۲.۲۹ local search

ایده ی local search مانند hamming ball می باشد. به این صورت که وقتی به جواب s می رسیم (در این قسمت همان TSP)، همه ی مسیرهایی که به اندازه ی یک یال از مسیر s فاصله دارند را بررسی می کنیم. با بزرگتر کردن شعاع و چک کردن یال های بیشتر، شانس پیدا کردن جواب بهینه را بیشتر می کنیم. فضای جستجو در این راه حل می تواند exponential باشد. اما با گذاشتن وقت بیشتر، شانس رسیدن به جواب بهینه را بیشتر می کنیم.

۷.۲.۲۹ vertex cover

۸.۲.۲۹ تعریف مسئله

با داشتن گراف، به دنبال کوچکترین زیر مجموعه از رأس ها هستیم که حداقل یک طرف از هر کدام از یال ها را انتخاب کرده باشیم.

۹.۲.۲۹ Approximate vertex cover

برای حل مسئله Vertex cover می توان در هر مرحله، یکی از یال ها را به صورت رندوم انتخاب کرد. سپس خود آن یال به همراه یال های متصل به رأس های دو طرف آن را از گراف حذف کنیم. در ادامه ثابت می کنیم که این الگوریتم، در زمان چندجمله ای، جوابی می دهد که حداکثر ۲ برابر جواب بهینه است.[۹]

```
C ← empty set;
while E is not empty do
    {u, v} ← any edge from E add u, v to C remove from E all edges
    incident to u, v
end
return C
```

Algorithm 39: Approximate vertex cover

اثبات: اگر M را مجموعه یال های انتخاب شده در نظر بگیریم، بین آن ها matching وجود دارد. یعنی با انتخاب هر یال، از آنجایی که یال های متصل به رأس های ۲ طرفش را حذف می کنیم، می دانیم که به هیچکدام از رأس های اطراف یال قبلی انتخاب شده وصل نیست. پس vertex cover گراف، حداقل به اندازه ی M می باشد. با توجه به اینکه در الگوریتم بیان شده، ما هردو رأس اطراف یال را انتخاب می کنیم، نتیجه می گیریم که جواب بدست آمده، حداکثر مساوی ۲ برابر جواب بهینه است.

[۹] به اصطلاح این الگوریتم approximate-۲ است

جلسه ۳۰

الگوریتم کوانتوم

امیرحسین احمدی – ۱۳۹۹/۵/۱۵

جزوه جلسه ۳۰ام مورخ ۱۳۹۹/۵/۱۵ درس طراحی و تحلیل الگوریتم تهیه شده توسط امیرحسین احمدی. در جهت مستند کردن مطالب درس طراحی و تحلیل الگوریتم

علاوه بر مسایل P ، NP مسایلی داریم به نام BQP. این نام مخفف عبارتِ Bounded-error Quantum Polynomial time است و به این معنی است که حل آن مساله به وسیله کامپیوتر کوانتومی با خطای محدود قابل انجام است. برای مثال میتوان مسایل Integer Factorization را نام برد که برای به دست آوردن عوامل اول یک عدد نسبتاً بزرگ به کار میرود و در الگوریتم های بانکی مانند RSA کاربرد فراوانی دارد. که در این الگوریتم RSA دو عدد بزرگ اول را در هم ضرب میکنند که با رمزگشایی آن به اطلاعات دسترسی خواهیم داشت. همچنین یک اشاره کوچکی به کامپیوتر های کوانتومی خواهیم زد که برای به دست آوردن عوامل اول یک عدد N بیتی، به N تا Qbit نیاز خواهیم داشت که در ادامه بیشتر توضیح خواهیم داد. ابتدا به تعریف چندین اصطلاح میپردازیم:

Quantum Advantage: به معنای این است که آن کامپیوتر کوانتومی آن مساله را بهتر از یک کامپیوتر عادی حل کرده است.

Quantum Supremacy: به معنای این است که مساله ای که در کامپیوتر عادی قابل حل نیست، روی کوانتوم میتوان انجام داد.

شاید به نظر برسد که با افزایش تعداد Qbit قادر به انجام محاسبات بیشتر خواهیم بود، اما واقعیت به اینگونه است که هر چه تعداد Qbit افزایش پیدا کند، خطا نیز زیاد میشود که اتفاق خوبی نیست! بنابراین بیشترین تعداد Qbit که تاکنون استفاده شده، ۵۳ تاست و هنوز بیشتر از آن استفاده نشده است.

۱.۳۰ رابطه بین محاسبات و آزمایش

تا الآن احتمالاً هرگونه مساله ای که به گوشمان خورده است، این بوده که میخواستیم نتیجه یک آزمایش را محاسبه کنیم. برای مثال در آزمایش پرتاب توپ، میتوان برای محاسبه، از قوانین فیزیکی بهره گرفت. اما نکته این است که اگر این محاسبات بیش از حد پیچیده باشد و زمان زیادی ببرد، شاید ایده بهتر این باشد که واقعاً آن آزمایش را انجام دهیم و نتیجه را اعلام کنیم؛ زیرا آزمایش ها به مراتب سریع تر از محاسبات پیچیده انجام خواهند شد.

۲.۳۰ Dirac Vector Notation

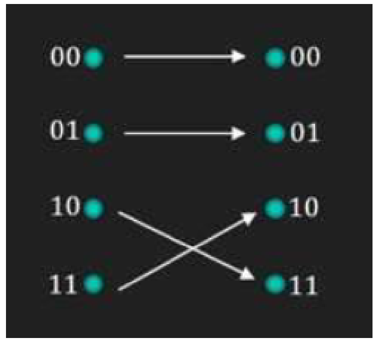
یک بیت با مقدار صفر به صورتِ `|0>` نشان داده خواهد شد که به صورت وکتوری به شکل (۰و۱) است و اگر مقدارش ۱ باشد، به صورتِ `|1>` نمایش داده خواهد شد که به صورتِ وکتوری به شکل (۱و۰) خواهد بود.

۳.۳۰ اعمال مجاز روی کامپیوتر عادی

برای یک بیت، ۴ عملِ Identity، Negation، Constant−۰، Constant−۱ مجاز خواهند بود که از بین این ۴ مورد، دو مورد عمل بازگشت پذیر خواهد بود که جواب را از پاسخ سوال برگردانیم.

به جز اینها، یک عمل داریم به نام CNOT و به این معنی است که یک بیت کنترل و یک بیت هدف داریم، که به این صورت عمل میکند: بیت کنترل عیناً منتقل میشود و بیت هدف نیز وابسته به بیت کنترل است که اگر صفر بود، خودش منتقل میشود وگرنه not میشود و انتقال می یابد که تصویر نمونه ای از این عملگر را مشاهده میکنید.

figure reference



۴.۳۰ Qbits

این بیت ها به صورت (a،b) نمایش داده میشوند که یکی از خواصش این است که a*a+b*b=۱ و همچنین احتمال اینکه این بیت ۱ باشد، a*a و احتمال آنکه صفر باشد، b*b است.
بدیهتاً ۴ عملی که در cnot ارایه کردیم، برای qnot نیز کاربرد دارد و همان شرایط را اعمال میکند.

۵.۳۰ Hadamard Gate

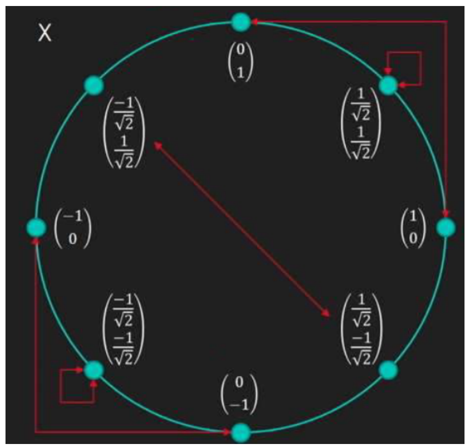
این عمل از ضرب ماتریس زیر در ماتریس اصلی به دست می آید که همچنان برگشت پذیر هست.

figure reference

$$= \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{pmatrix}$$

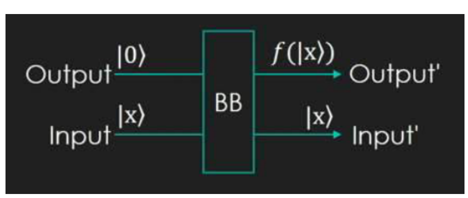
۶.۳۰ Circle State Machine

برای qbit ها، یک استیت ماشین تعریف میشود که به ما به صورت گرافیکی نشان میدهد که با اعمال آن عملگر، از هر استیت به چه استیتی خواهیم رفت که برای نمونه، عملگر not را در زیر مشاهده میکنید. همچنین دو خاصیتِ برگشت پذیری و اعمال دوگانه از این شکل به خوبی قابل مشاهده است که چرا با دو بار برای مثال not کردن، به استیت اولیه خواهیم رسید.



۷.۳۰ Deutsch Oracle

یکی از مثال هایی که برای قدرت و برتری کامپیوترهای کوانتومی میتوان زد این مثال است که فرض کنید در یک Black Box یکی از ۴ عمل اصلی برای کامپیوتر عادی را داشته باشیم. یک ورودی به این جعبه میدهیم و با استفاده از خروجیش میخواهیم بدانیم که در Black Box چه عملگری وجود داشت. این مساله با استفاده از یک کویری انجام پذیر نخواهد بود. اما با استفاده از ورودی دادن به صورت زیر به یک کامپیوتر کوانتومی، میتوانیم برتری آن را متوجه شویم. و نهایتاً اگر خروجی نهایی به صورت ۱۱> بود، یعنی آن عملگر constant بوده و اگر ۰۱> باشد، یعنی آن عملگر وابسته به ورودی بوده است.



Entanglement ۸.۳۰

فرض کنید یک product state داشته باشیم. برای اینکه آن را به عواملش تجزیه کنیم، ممکن است از لحاظ منطقی این کار امکان‌پذیر نباشد. در اصطلاح میگویند که آنها Entangled شده‌اند.

Teleportation ۹.۳۰

به این معنی است که میتوانیم یک استیت را به جای دیگر منتقل کنیم؛ البته باید توجه کنید که این عمل به صورت cut ، paste است نه به صورت copy ، paste.

Bibliography

[1] http://pi.math.cornell.edu/~mec/Winter2009/Thompson/search.html.

[2] https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.

[3] https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm.

[4] https://www.programiz.com/dsa/bellman-ford-algorithm.

[5] https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/.

[6] https://www.quora.com/What-are-the-advantages-of-Bellman-Ford-algorithm-when-to-use-it-over-Djisktra.

[7] Competitive-Programming. *2-SAT*. https://cp-algorithms.com/graph/2SAT.html.

[8] *Computer Science Tutorials*. https://www.geeksforgeeks.org/.

[9] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[10] C.H.Papadimitriou Dasgupta and U.V.Vazirani. *Algorithms*. http://algorithmics.lsi.upc.edu/docs/Dasgupta-Papadimitriou-Vazirani.pdf.

[11] *Der Dijkstra Algorithmus*. https://www-m9.ma.tum.de/graph-algorithms/spp-dijkstra/index_en.html.

[12] university of california san diego. *P and NP*. https://www.coursera.org/lecture/advanced-algorithms-and-complexity/p-and-np-MJFwb.

[13] *diet problem*. http://pages.cs.wisc.edu/~swright/525/handouts/dualexample.pdf.

[14] *Dijkstra's Algorithm Learning Tool*. http://syllabus.cs.manchester.ac.uk/ugt/2019/COMP26120/DijkstraLearningTool/DijkstraLearningTool/index.html.

[15] *Disjoint Sets*. https://sauleh.github.io/ds98/lectures/.

[16] *Edmonds Karp pseudocode*. https://jamieheller.github.io/editor.html?view=step.

[17] *Ford Fulkerson pseudocode*. https://en.wikipedia.org/wiki/Ford-Fulkerson_algorithm#Algorithm.

[18] David Galles. *Data Structure Visualizations*. https://www.cs.usfca.edu/~galles/visualization/StackArray.html. Accessed: 2019-12-10.

[19] Geek-For-Geeks. *2-SAT-Problem*. geeksforgeeks.org/2-satisfiability-2-sat-problem/.

[20] *Graph Online*. https://graphonline.ru/en/.

[21] *KMP algorithm*. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching. Accessed: 2019-12-10.

[22] Michael Levin. *algorithms-on-strings*. https://www.coursera.org/learn/algorithms-on-strings/home/week/4.

[23] Michael Levin. *Data Structures and Algorithms Specialization on Coursera: Algorithms on Graphs: Dijkstra's Algorithm ,Bellman–Ford algorithm*. Michael Levin, Lecturer at the Computer Science Department of Higher School of Economics.

[24] *NP-completeness*. https://en.wikipedia.org/wiki/NP-completene.

[25] *NP-hardness*. https://en.wikipedia.org/wiki/NP-hardness.

[26] *prefix and suffix*. https://https://en.wikipedia.org/wiki/Substring. Accessed: 2019-12-10.

[27] *Prim algorithm VS Kruskal algorithm*. https://medium.com/@pp7954296/minimum-spanning-tree-940b80568ecb.

[28] *stable sorting*. https://www.geeksforgeeks.org/stability-in-sorting-algorithms. Accessed: 2019-12-10.

[29] *suffix tree constructor*. https://hwv.dk/st.html.

[30] Wikipedia. *Skew-Symmetric Matrix*. https://en.wikipedia.org/wiki/Skew-symmetric_matrix.

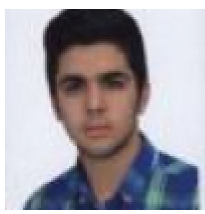
[31] Wikipedia. *Solve NP Complete Problems*. https://en.wikipedia.org/wiki/NP-completeness#Solving_NP-complete_problems.

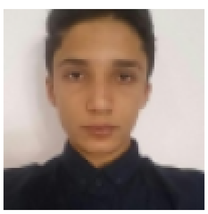
۳۱

فهرست دانشجویان


احمد بهمني


ارمین غلام پور


امین یعقوبي ثاني

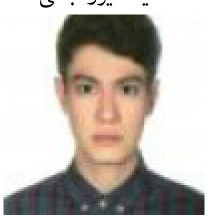

امید میرزاجانی


امیر حسین احمدی

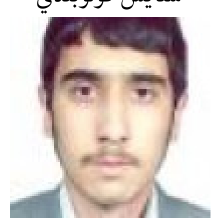

ستایش کولوبندي


سها جعفري مذهب حقیقي


سهراب نمازی نیا

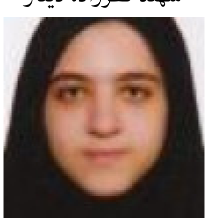

سهند نظرزاده دینار


سپهر باباپور


سیدمحمدمهدی رضوي


شایان موسوی نیا

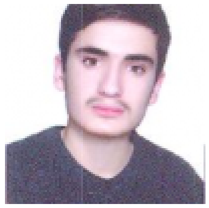

شقایق مبشر


غزل زماني نژاد


فاطمه امیدي

فاطمه احمدي

متین مرجانی



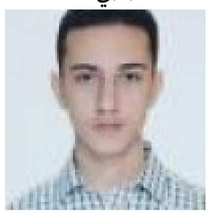
مجتبي نافذ

محمدحسین کریمیان

محمدرضا امین رعیا

محمدسجاد نقی زاده

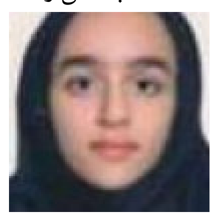
محمدصدرا خاموشی فر



محمدعلی فراهت

محمدمصطفی رستم خاني

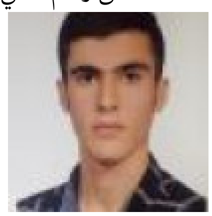
ملیکا احمدی رنجبر



ملیکا نوبختیان

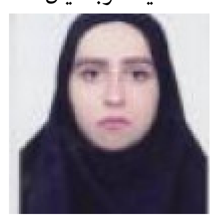
نگار زین العابدین

نگین درخشان



هادی شیخی محمدابادی

هستی کرمدل



پریسا علائي

یاسمن لطف اللهی میراشرف