به نام خدا

طراحی الگوریتم‌ها

آرش شفیعی

الگوریتم‌های گراف

الگوریتم‌های گراف

- گراف یک ساختار گسسته است که با استفاده از آن می‌توان تعدادی مفهوم که با یکدیگر در ارتباط هستند را مدلسازی کرد.
- برای مدلسازی مفاهیم از رئوس گراف و برای مدلسازی ارتباط بین مفاهیم از یال‌های گراف استفاده می‌کنیم.
- گراف‌ها در علوم کامپیوتر و دیگر شاخه‌های علوم بسیار پر استفاده‌اند. برای مثال برای مدلسازی یک شبکهٔ کامپیوتری که از تعدادی کامپیوتر و تعدادی مسیر ارتباطی تشکیل شده است، می‌توانیم از یک گراف استفاده کنیم و از الگوریتم‌های گراف برای یافتن مسیر بهینه برای انتقال یک بسته در شبکه بهره بگیریم. همچنین در شبکه‌های اجتماعی می‌توان افراد و سازمان‌ها را به عنوان رئوس یک گراف در نظر گرفت و ارتباط بین افراد و سازمان‌ها را با استفاده از یال‌های گراف مدلسازی کرد. از الگوریتم‌های گراف جهت تحلیل این شبکه اجتماعی برای یافتن اطلاعات در گراف می‌توان استفاده کرد.

الگوریتم‌های گراف

- در زبان‌شناسی می‌توان از گراف‌ها جهت نمایش ارتباط بین کلمات در یک زبان استفاده کرد و از گراف به دست آمده در پردازش زبان طبیعی، و ترجمه‌های ماشینی استفاده کرد.
- در علوم فیزیک و شیمی می‌توان از گراف‌ها جهت مدلسازی مولکول‌ها استفاده کرد و ساختار مولکول‌ها و روابط آنها و نیروهای بین اتم‌ها و مولکول‌ها را شبیه سازی کرد.
- در علوم اجتماعی می‌توان از گراف‌ها جهت مدلسازی ارتباط انسان‌ها و نحوه منتشر شدن اطلاعات و افکار بین انسان‌ها و جوامع استفاده کرد.
- در علوم زیست‌شناسی می‌توان از گراف‌ها جهت بررسی روابط بین گونه‌های جانوری و گیاهی و همچنین بررسی ساختار ژن‌ها استفاده کرد.

الگوریتم‌های گراف

- در این قسمت با روش‌های نمایش گراف و جستجوی گراف‌ها آشنا می‌شویم. با استفاده از روش‌های جستجوی گراف‌ها می‌توانیم ساختار یک گراف و ویژگی‌های آن را بشناسیم.

الگوریتم‌های گراف

- یک گراف را می‌توان با استفاده از دو مجموعهٔ رأس‌ها [1] (V) و یال‌ها [2] (E) نمایش داد. بدین ترتیب دوتایی $G = (V, E)$ گرافی را نمایش می‌دهد که در آن V مجموعه‌ای است از رئوس و E مجموعه‌ای است از یال‌ها. یک یال دو رأس را به یکدیگر متصل می‌کند.

[1] vertex set
[2] edge set

الگوریتم‌های گراف

- علاوه بر روش استاندارد نمایش یک گراف توسط مجموعه‌ها، می‌توانیم یک گراف را توسط مجموعه‌ای از لیست‌های مجاورت [1] یا یک ماتریس مجاورت [2] نشان دهیم.
- توسط لیست مجاورت می‌توان گراف‌های خلوت [3] را که در آنها $|E|$ بسیار کوچک‌تر از $|V|^2$ است نمایش داد. وقتی گراف متراکم [4] است، می‌توان از نمایش ماتریس مجاورت استفاده کرد.

[1] adjacency lists
[2] adjacency matrix
[3] sparse graph
[4] dense

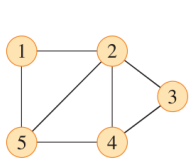
الگوریتم‌های گراف

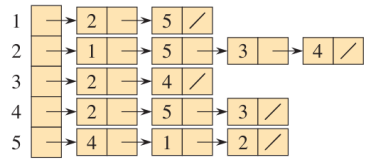
- در نمایش لیست مجاورت [1] برای گراف $G = (V, E)$ از آرایهٔ Adj شامل $|V|$ عنصر استفاده می‌کنیم. به ازای هر $u \in V$ ، لیست مجاورت $Adj[u]$ شامل رئوس v است به طوری‌که $(u, v) \in E$. پس $Adj[u]$ شامل همهٔ رئوسی است که در گراف G مجاور u هستند. در پیاده‌سازی این روش، عناصر این آرایه می‌توانند اشاره‌گر به رئوس مجاور باشند.

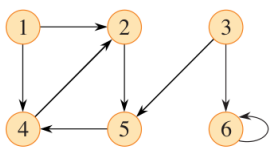
[1] adjacency-list representation

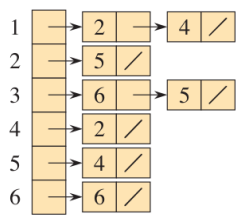
الگوریتم‌های گراف

- در شکل زیر دو گراف توسط لیست مجاورت نشان داده شده‌اند.









الگوریتم‌های گراف

- اگر G یک گراف جهت‌دار باشد، مجموع اندازه همهٔ لیست‌های مجاورت برابراست با $|E|$ ، زیرا هریک از یال‌های (u, v) توسط درایهٔ $Adj[u]$ نشان داده می‌شود.
- اگر G یک گراف بدون جهت باشد، آنگاه مجموع اندازهٔ همهٔ لیست‌های مجاورت برابراست با $2|E|$ ، زیرا هر یک از یال‌های (u, v) هم در $Adj[u]$ و هم در $Adj[v]$ نمایش داده می‌شود.
- فضای حافظه‌ای که لیست مجاورت برای نگهداری گراف نیاز دارد برابراست با $\Theta(V + E)$. یافتن یک یال در گراف نیز به زمان $\Theta(V + E)$ نیاز دارد، زیرا برای یافتن یک یال همهٔ $|V|$ درایهٔ لیست مجاورت باید بررسی شوند.

الگوریتم‌های گراف

- توسط لیست مجاورت می‌توانیم گراف‌های وزن‌دار [1] را نیز نمایش دهیم. در یک گراف وزن‌دار، هریک از یال‌ها دارای یک وزن است که توسط تابع وزن $w : E \to \mathbb{R}$ [2] تولید می‌شود.
- برای مثال فرض کنید $G = (V, E)$ یک گراف وزن‌دار با تابع وزن w باشد. آنگاه می‌توانیم وزن $w(u, v)$ از یال $(u, v) \in E$ را در کنار رأس v در لیست مجاورت u ذخیره کنیم و نمایش دهیم.
- یکی از معایب لیست مجاورت این است که برای پیدا کردن یال (u, v) سریع‌ترین روش ممکن جستجوی v در لیست مجاورت $Adj[u]$ است.
- ماتریس مجاورت برای پیدا کردن یک یال می‌تواند از لیست مجاورت سریع‌تر عمل کند.

[1] weighted graphs
[2] weight function

الگوریتم‌های گراف

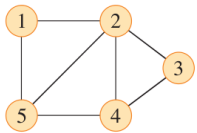
- در نمایش ماتریس مجاورت [1] برای گراف $G = (V, E)$ فرض می‌کنیم هر رأس یک شماره از 1 تا $|V|$ داشته باشد. سپس گراف G را با استفاده از ماتریس $A = (a_{ij})$ با اندازهٔ $|V| \times |V|$ نشان می‌دهیم به طوری‌که

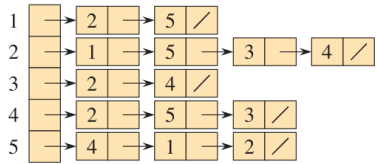
$$a_{ij} = \begin{cases} 1 & \text{اگر } (i,j) \in E \\ 0 & \text{در غیر اینصورت} \end{cases}$$

[1] adjacency matrix representation

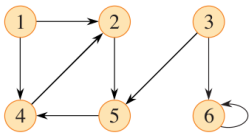
الگوریتم‌های گراف

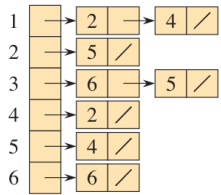
- در شکل زیر دو ماتریس مجاورت نشان داده شده‌اند.





	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0





	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

الگوریتم‌های گراف

- ماتریس مجاورت به حافظه $\Theta(V^2)$ برای نگهداری گراف نیاز دارد. برای یافتن یال (u, v) در گراف می‌توانیم درایهٔ $A[u, v]$ را بررسی کنیم و بنابراین یافتن یک یال در گراف در زمان $\Theta(1)$ انجام می‌شود.
- برای یک گراف بدون جهت، گراف نسبت به قطر اصلی‌اش متقارن است، زیرا $A[u, v]$ برابراست با $A[v, u]$ بنابراین ماتریس مجاورت برابر با ترانهادهٔ [1] آن است و داریم $A = A^T$.
- اما در یک گراف جهت‌دار می‌توانیم یالی از u به v داشته باشیم بدون اینکه یال از v به u وجود داشته باشد.

[1] transpose

الگوریتم‌های گراف

- با استفاده از ماتریس مجاورت می‌توانیم یک گراف وزن‌دار را نیز نمایش دهیم.
- برای مثال، اگر $G = (V, E)$ یک گراف وزن‌دار با تابع وزن w باشد، می‌توان وزن $w(u, v)$ برای یال $(u, v) \in E$ را به عنوان درایهٔ ماتریس مجاورت در سطر u و ستون v ذخیره کرد.
- استفاده از ماتریس مجاورت در عمل راحت‌تر است اما به ازای گراف‌های بزرگ خلوت ممکن است فضای حافظه مورد نیاز آنها بسیار زیاد شود. برای ذخیره‌سازی گراف‌های خلوت جهت‌دار، لیست مجاورت نسبت به ماتریس مجاورت فضای بسیار کمتری اشغال می‌کند.

جستجوی سطح‌اول

- جستجوی سطح‌اول [1] یکی از ساده‌ترین الگوریتم‌های جستجوی گراف است که در بسیاری از الگوریتم‌های گراف استفاده می‌شود. برای مثال الگوریتم دایکسترا برای یافتن کوتاهترین مسیر بین دو رأس از جستجوی سطح‌اول استفاده می‌کند.
- به ازای گراف دلخواه $G = (V, E)$ و یک رأس مبدأ [2] به نام s ، الگوریتم جستجوی سطح‌اول همهٔ یال‌های گراف G را با شروع از رأس s بررسی می‌کند.
- با شروع از رأس s ، الگوریتم سطح‌اول ابتدا رئوسی را بررسی می‌کند که به s نزدیک‌ترند، بدین معنی که برای رسیدن به آن رئوس از s باید از تعداد یال‌های کمتری عبور کرد.

[1] breadth-first search
[2] source vertic

جستجوی سطح‌اول

- یال‌هایی که در جستجوی سطح‌اول به ترتیب بررسی می‌شوند، یک درخت سطح‌اول می‌سازد که ریشهٔ آن s است و به ازای هر یک از رئوس v ، یک مسیر ساده از s به v کوتاهترین مسیر از s به v در گراف را نشان می‌دهد. در اینجا کوتاهترین مسیر درواقع مسیری است که دارای کمترین تعداد یال باشد.

- جستجوی سطح‌اول، بدین دلیل سطح‌اول نامیده می‌شود که به ازای هر رأس v ابتدا رئوس مجاور آن بررسی می‌شوند، قبل از اینکه رئوس مجاور مجاور آن بررسی شوند. بنابراین اگر نزدیکترین رئوس به یک رأس را در سطح در نظر بگیریم و دورترین رئوس را در عمق، جستجوی سطح‌اول، قبل از بررسی رئوس در عمق، همهٔ رئوس در سطح را بررسی می‌کند. بنابراین برخلاف جستجوی عمق اول که به ازای هر رأس v یک رأس مجاور مجاور v ممکن است قبل از یک رأس مجاور v پیمایش شود، در جستجوی سطح‌اول همهٔ رئوس مجاور v قبل از پیمایش رئوس مجاور مجاور v پیمایش می‌شوند.

جستجوی سطح‌اول

- در جستجوی سطح‌اول با شروع از رأس s ابتدا رئوس مجاور [1] یا همسایه‌هایی [2] بررسی می‌شوند که فاصلهٔ آنها از s برابر ۱ است، سپس همسایه‌ها با فاصلهٔ ۲ بررسی می‌شوند، پس از آن همسایه‌ها با فاصلهٔ ۳ و به همین ترتیب الی آخر، تا وقتی که همه رئوس بررسی شده باشند.
- در جستجوی سطح‌اول از یک صف استفاده می‌شود که در آن ابتدا همسایه‌ها با فاصله ۱، سپس همسایه‌ها با فاصلهٔ ۲ و به همین ترتیب الی آخر در صف قرار می‌گیرند. بنابراین با خارج کردن همسایه‌ها از صف به ترتیب فاصله، گراف به صورت سطح‌اول بررسی می‌شود.

[1] adjacent
[2] neighbour

جستجوی سطح‌اول

- در الگوریتم جستجوی سطح‌اول می‌توانیم برای هر رأس ۳ رنگ در نظر بگیریم : سفید، خاکستری و سیاه. همهٔ رئوس در ابتدا به رنگ سفید هستند و رئوسی که هیچ مسیری از s به آنها وجود ندارد تا انتها به رنگ سفید باقی می‌مانند. وقتی یک رأس برای اولین بار با شروع از s پیمایش می‌شود، آن رأس به رنگ خاکستری تبدیل می‌شود. رنگ خاکستری بدین معنی است که آن رأس در مرز جستجو قرار گرفته است. مرز جستجو در واقع مرز میان رئوس پیمایش نشده و رئوس پیمایش شده است. صفی که در جستجوی سطح‌اول استفاده می‌شود، شامل همهٔ رئوس خاکستری است.
- رئوس خاکستری به ترتیب از صف خارج می‌شوند و به رنگ سیاه تبدیل می‌شوند و رئوس سفید همسایهٔ آنها که تاکنون پیمایش نشده‌اند به رنگ خاکستری تبدیل می‌شوند و وارد صف می‌شوند.

جستجوی سطح‌اول

- یک الگوریتم جستجوی سطح‌اول، یک درخت سطح‌اول می‌سازد که ریشهٔ آن رأس s است. هرگاه در فرایند جستجو، یک رأس سفید v که در لیست همسایه‌های رأس خاکستری u قرار دارد پیدا می‌شود، رأس v و یال (u, v) به درخت اضافه می‌شوند. می‌گوییم رأس u ، سلف [1] یا پدر رأس v و رأس v خلف [2] رأس u در درخت سطح‌اول است. از آنجایی که هر رأس قابل دسترس از طریق s تنها یک بار بررسی می‌شود، هر رأس تنها یک سلف دارد.
- تنها رأس ریشه، یعنی رأس s دارای پدر نیست.
- اگر رأس u بر روی یک مسیر ساده درخت از ریشه s به رأس v قرار بگیرد، آنگاه رأس u جد [3] رأس v است و رأس v نوادهٔ [4] رأس u است.

---

[1] predecessor
[2] successor
[3] ancestor
[4] descendant

جستجوی سطح‌اول

- در الگوریتم جستجوی سطح‌اول که بررسی خواهیم کرد، `v.color` رنگ رأس v است که می‌تواند سفید، خاکستری یا سیاه باشد، `v.d` فاصلهٔ رأس v از رأس s است و `v.pred` سلف رأس v است.
- رأس `v.pred` سلف [1] رأس v است، و رأس v خلف [2] رأس `v.pred` .

[1] predecessor
[2] successor

جستجوی سطح‌اول

- الگوریتم زیر جستجوی سطح‌اول را نشان می‌دهد.

Algorithm Bfs

```
   function BFS(G,s)
1: for each vertex u ∈ G.V - {s}  do
2:    u.color = White
3:    u.d = ∞
4:    u.pred = Nil
5: s.color = Gray
6: s.d = 0
7: s.pred = Nil
8: Q = ∅
9: Enqueue(Q,s)
```

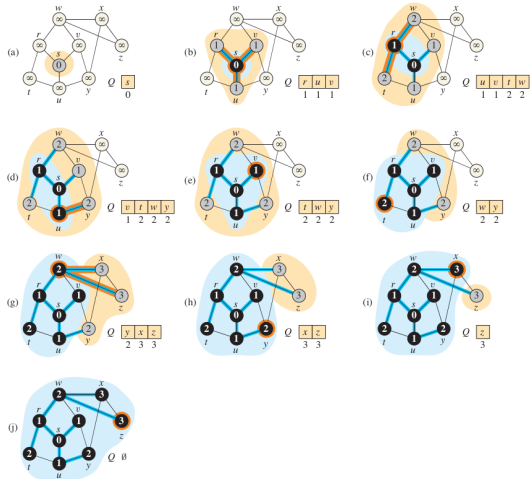
جستجوی سطح‌اول

Algorithm Bfs

```
10: while !empty(Q) do
11:    u = Dequeue(Q)
12:    for each vertex v in G.Adj[u] do      ▷ search the neighbors of u
13:       if v.color == White then          ▷ is v being discovered now?
14:          v.color = Gray
15:          v.d = u.d + 1
16:          v.pred = u
17:          Enqueue(Q,v)                   ▷ v is now on the frontier
18:    u.color = Black      ▷ u is now behind the frontier
```

جستجوی سطح‌اول

- در شکل زیر یک گراف به صورت سطح‌اول پیمایش شده است.



جستجوی سطح‌اول

- تحلیل الگوریتم جستجوی سطح‌اول : برای تحلیل الگوریتم جستجوی سطح‌اول می‌توانیم از تحلیل تجمعی استفاده کنیم. در این الگوریتم هیچ‌گاه یک رأس از رنگ خاکستری یا سیاه به رنگ سفید در نمی‌آید. هر رأس حداکثر یک بار وارد صف می‌شود و بنابراین هر رأس حداکثر یک بار از صف خارج می‌شود. عملیات اضافه کردن و برداشتن از صف در زمان $O(1)$ انجام می‌شود. خارج کردن رئوس از صف در زمان $O(V)$ انجام می‌گیرد. همچنین بررسی لیست مجاورت هر رأس حداکثر یک‌بار انجام می‌شود و مجموع طول همهٔ لیست‌های مجاورت برابر است با $\Theta(E)$ ، بنابراین زمان لازم برای اجرای الگوریتم برابراست با $O(V+E)$. نتیجه می‌گیریم جستجوی سطح‌اول در زمان خطی نسبت به اندازه لیست مجاورت اجرا می‌شود.
- می‌توان ثابت کرد الگوریتم جستجوی سطح‌اول کوتاهترین مسیر از s به هریک از رئوس را محاسبه می‌کند.

جستجوی عمق‌اول

- جستجوی عمق‌اول به جای اینکه جستجو را در سطح شروع کند و همهٔ رئوس مجاور را در ابتدا پیمایش کند، به ازای هر رأس پیمایش شده، مجاور رأس را پیمایش می‌کند و به عبارت دیگر جستجو در عمق انجام می‌دهد.

- برای روشن‌تر شدن جستجوی سطحی و عمقی مثال زیر را در نظر بگیرید. می‌خواهیم مطلبی را درون چندین کتاب جستجو کنیم. در یک جستجوی سطحی ابتدا به سراغ صفحه اول همهٔ کتاب‌ها می‌رویم تا این که در نهایت همهٔ کتاب‌ها را بررسی کنیم. در یک جستجوی عمقی ابتدا کتاب اول را تا انتها مطالعه می‌کنیم و در صورتی که مطلب مورد نظر را پیدا نکردیم به سراغ کتاب دوم می‌رویم تا این که در نهایت همهٔ کتاب‌ها را جستجو کنیم. در این مثال هیچ یک از جستجوها مزیتی بر دیگری ندارد چرا که مطلب مورد نظر ممکن است در صفحهٔ آخر کتاب اول باشد که در این صورت جستجوی عمقی زودتر به جواب می‌رسد و یا ممکن است مطلب مورد نظر در صفحه اول کتاب آخر باشد که در این صورت جستجوی سطح اول زودتر به جواب می‌رسد.

جستجوی عمق‌اول

- جستجوی عمق‌اول یال‌های بررسی نشدهٔ رئوس تازه پیدا شده را زودتر از یال‌های بررسی نشدهٔ رئوس قبلاً پیدا شده بررسی می‌کند. وقتی فرایند جستجو به نقطه‌ای رسید که یال‌های یک رأس همگی بررسی شده بودند، الگوریتم پسگرد می‌کند تا به رئوسی برسد که یال‌های آنها هنوز بررسی نشده‌اند.
- در صورتی که یک رأس با شروع از رأس آغازی قابل دسترس نباشد، گراف همبند نیست و برای جستجوی کامل گراف، الگوریتم یکی از رئوس را به عنوان مبدأ جدید انتخاب کرده و جستجو را از رأس جدید آغاز می‌کند.

جستجوی عمق‌اول

- همانند جستجوی سطح اول، در جستجوی عمق‌اول توسط رنگ رأس‌ها وضعیت آنها مشخص می‌شود. هر رأس در ابتدا سفید است، هنگامی که برای اولین بار پیدا می‌شود به رنگ خاکستری تبدیل می‌شود و در پایان هنگامی که بررسی شد (بدین معنی که همهٔ رئوس در لیست مجاورت آن پیدا شدند) به رنگ سیاه در می‌آید.

جستجوی عمق‌اول

- در جستجوی عمق‌اول هر رأس دارای دو برچسب زمان [1] است. برچسب زمان اول $v.s$ زمانی نشان می‌دهد که رأس برای بار اول پیدا شده است و به رنگ خاکستری درآمده است و برچسب دوم $v.f$ مشخص می‌کند که رأس v به طور کامل بررسی شده است بدین معنی که همهٔ رئوس لیست مجاورت آن پیدا شده‌اند و رأس v به رنگ مشکی درآمده است.

[1] time stamp

جستجوی عمق‌اول

- الگوریتم زیر جستجوی عمق‌اول را نشان می‌دهد.

Algorithm Depth-First Search

```
   function DFS(G)
1: for each vertex u ∈ G.V do
2:    u.color = White
3:    u.pred = Nil
4: time = 0
5: for each vertex u ∈ G.V do
6:    if u.color == White then
7:       DFS-Visit(G,u)
```

Algorithm DFS-Visit

```
   function DFS-Visit(G,u)
 1: time = time + 1  ▷ white vertex u has just been discovered
 2: u.s = time
 3: u.color = Gray
 4: for each vertex v in G.Adj[u] do  ▷ explore each edge(u,v)
 5:    if v.color == White then
 6:       v.pred = u
 7:       DFS-Visit(G,v)
 8: time = time + 1
 9: u.f = time
10: u.color = Black  ▷ blacken u; it is finished
```

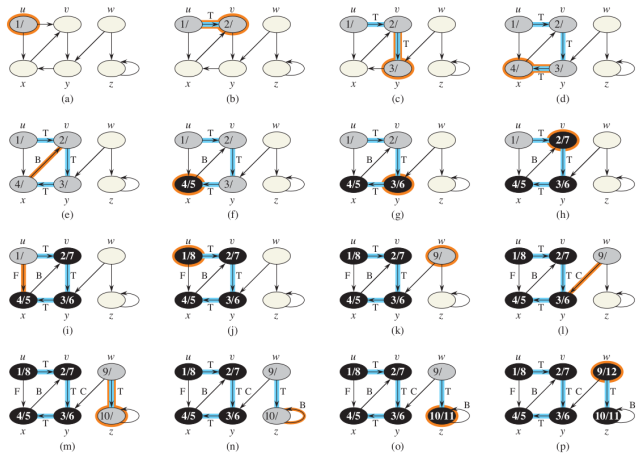
جستجوی عمق‌اول

- وقتی الگوریتم جستجوی عمق‌اول به اتمام می‌رسد هر رأس دارای دو برچسب زمان است که یکی زمان پیدا شدن [1] و دیگری زمان به پایان رسیدن [2] را نشان می‌دهد.

[1] discovery time
[2] finish time

جستجوی عمق‌اول

- در مثال زیر، گراف توسط الگوریتم عمق‌اول بررسی شده است.



جستجوی عمق‌اول

- در اینجا نیز برای تحلیل الگوریتم از تحلیل تجمعی استفاده می‌کنیم.
- الگوریتم `DFS-Visit` برای هر رأس $v \in V$ تنها یک بار فراخوانی می‌شود، چرا که این الگوریتم برای رئوس سفید فراخوانی می‌شود و آنها را به رنگ خاکستری تبدیل می‌کند. الگوریتم `DFS-Visit` به ازای هر رأس v در یک حلقه تکرار $|Adj[v]|$ بار تکرار می‌شود. بنابراین برای همهٔ رئوس، این حلقه $\sum_{v \in V} |Adj[v]| = \Theta(E)$ بار تکرار می‌شود.
- پس زمان اجرای الگوریتم جستجوی عمق‌اول $\Theta(V + E)$ است.

جستجوی عمق‌اول

- می‌توان اثبات کرد که در جستجوی عمق‌اول زمان پیدا شدن و زمان به اتمام رسیدن رئوس گراف یک ساختار پرانتزگذاری کامل دارند. اگر به ازای یافته شدن رأس u یک پرانتز به صورت "$(u$" باز کنیم و به ازای به اتمام رسیدن بررسی رأس u پرانتز را به صورت "$u)$" ببندیم، یک عبارت با پرانتزگذاری کامل به دست می‌آید بدین معنی که پرانتزها تودرتو هستند.

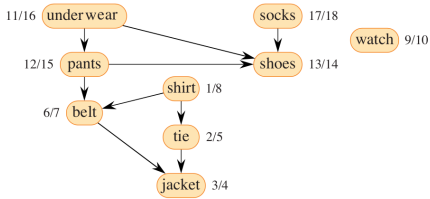
جستجوی عمق‌اول

- از جستجوی عمق‌اول در مرتب‌سازی توپولوژیکی [1] یا مرتب‌سازی موضعی یک گراف بدون دور [2] استفاده می‌شود.
- یک مرتب‌سازی توپولوژیکی در یک گراف بدون دور $G = (V, E)$ رئوس گراف را به گونه‌ای مرتب می‌کند که اگر G شامل یال (u, v) باشد، آنگاه u قبل از v در آرایهٔ مرتب شده قرار می‌گیرد.
- مرتب‌سازی توپولوژیکی تنها برای گراف‌های جهت‌دار بدون دور [3] تعریف می‌شود.
- مرتب‌سازی توپولوژیکی به گونه‌ای است که اگر رئوس مرتب شده برروی یک خط افقی قرار بگیرند، جهت همهٔ یال‌های از چپ به راست است.

[1] topological sort
[2] acyclic graph
[3] directed acyclic graph

جستجوی عمق‌اول

- از یک گراف جهت‌دار بدون دور می‌توان برای نمایش دادن رویدادها [1] استفاده کرد.
- برای مثال در شکل زیر برای یک گراف شامل تعدادی رویداد مرتب‌سازی توپولوژیکی انجام شده است.



[1] event

درخت پوشای کمینه

- در طراحی مدارهای الکتریکی، معمولاً طراحان نیاز دارند که قسمت‌هایی از مدار را که در یک سطح ولتاژ قرار دارند، به یکدیگر متصل کنند. برای متصل کردن n نقطه به یکدیگر، به $n-1$ سیم نیاز داریم که به شکل‌های متفاوت می‌توانند هر یک دو نقطه را به یکدیگر متصل کنند. در چنین مسئله‌ای هدف یافتن روشی برای اتصال است که در آن از کمترین مقدار سیم لازم استفاده می‌کنیم. برای مدلسازی این مسئله به صورت زیر عمل می‌کنیم.
- گراف $G = (V, E)$ را با وزن‌های $w : E \to \mathbb{R}$ در نظر بگیرید. بنابراین وزن یال $(u, v) \in E$ برابر است با $w(u, v)$.

درخت پوشای کمینه

- نقطه‌ها در یک مدار الکتریکی معادل رئوس گراف و سیم‌ها معادل یال‌های گراف هستند. مقدار سیمی که برای اتصال یک نقطه به نقطه دیگر نیاز است را با وزن یال مدلسازی می‌کنیم.
- هدف پیدا کردن زیر مجموعهٔ $T \subseteq E$ است که همهٔ رئوس گراف را به یکدیگر متصل می‌کند و وزن کل آن برابر با $w(T) = \sum_{(u,v)\in T} w(u,v)$ کمینه است.
- زیرمجموعهٔ T همهٔ رأس‌های گراف را به یکدیگر متصل می‌کند و دارای هیچ دوری نیست، پس یک درخت را تشکیل می‌دهد. به چنین درختی، درخت پوشا [1] گفته می‌شود. مسئله درخت پوشای کمینه [2] به دنبال درخت پوشایی می‌گردد که وزن آن از همهٔ درخت‌های پوشای دیگر کمتر باشد.

[1] spanning tree
[2] minimum spanning tree problem

درخت پوشای کمینه

- ورودی مسئله درخت پوشای کمینه گراف همبند بدون جهت $G = (V, E)$ است که تابع وزن یال‌های آن $w : E \to \mathbb{R}$ است. هدف یافتن یک درخت پوشای کمینه برای G است.
- دو الگوریتم حریصانه برای یافتن درخت پوشای کمینه معرفی خواهیم کرد که روش آنها مشابه است ولی پیاده‌سازی آنها متفاوت است.
- این استراتژی را به عنوان یک الگوریتم کلی برای یافتن درخت پوشای کمینه معرفی می‌کنیم.

درخت پوشای کمینه

- الگوریتم کلی برای یافتن درخت پوشای کمینه به صورت زیر است.

Algorithm Generic-MST

```
   function GENERIC-MST(G,w)
1: A =∅
2: while A does not form a spanning tree do
3:    find an edge (u,v) that is safe for A
4:    A = A ∪ (u,v)
5: return A
```

درخت پوشای کمینه

- قبل از هر تکرار در حلقه، A یک زیر مجموعه از یک درخت پوشای کمینه است.
- در هر گام از الگوریتم، یال (u, v) به A اضافه می‌شود بدون اینکه ویژگی A تغییر کند. به عبارت دیگر $A \cup (u, v)$ نیز زیرمجموعه‌ای از درخت پوشای کمینه است.
- یالی که به A اضافه می‌شود را یک یال مطمئن [1] می‌نامیم زیرا ویژگی درخت را حفظ می‌کند.
- پس در گام اول قبل از شروع حلقه ویژگی درخت (که زیرمجموعهٔ درخت پوشای کمینه است) برقرار است. در هرگام در حلقه تکرار ویژگی درخت حفظ می‌شود، پس در پایان یک درخت پوشای کمینه خواهیم داشت.

[1] safe edge

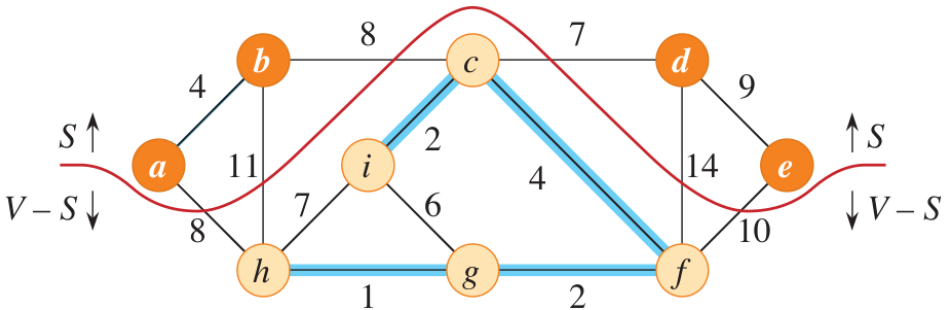
درخت پوشای کمینه

- حال روشی برای یافتن یال مطمئن ارائه می‌دهیم.
- قبل از بررسی ویژگی یال مطمئن چند تعریف ارائه می‌کنیم.
- یک برش $(S, V - S)$ [1] از یک گراف بدون جهت $G = (V, E)$ یک تقسیم‌بندی از رئوس V است که در آن مجموعهٔ رئوس به دو قسمت تقسیم می‌شوند.
- می‌گوییم یال $(u, v) \in E$ از برش $(S, V - S)$ عبور می‌کند [1] اگر یک رأس یال در مجموعهٔ S و رأس دیگر یال در مجموعهٔ $V - S$ قرار بگیرد.
- یالی را که از یک برش عبور می‌کند یال سبک [3] می‌نامیم اگر وزن آن در بین همهٔ یال‌هایی که از برش عبور می‌کنند کمینه باشد. در یک برش ممکن است چند یال سبک هم‌وزن وجود داشته باشند.

[1] cut
[1] crosses
[3] light edge

درخت پوشای کمینه

- شکل زیر یک برش را نشان می‌دهد.



درخت پوشای کمینه

- قضیه : فرض کنید $G = (V, E)$ یک گراف همبند بدون جهت با یال‌های وزن‌دار باشد و وزن‌ها با تابع w تعریف شده باشند. فرض کنید A یک زیرمجموعه از E باشد که در یک درخت پوشای کمینه برای G قرار گرفته باشد و فرض کنید $(S, V - S)$ یک برش از G باشد که هیچ یالی در A از آن عبور نمی‌کند. فرض کنید (u, v) یک یال سبک باشد که از برش $(S, V - S)$ عبور می‌کند. آنگاه یال (u, v) یک یال مطمئن برای A است.

- اثبات: از برهان خلف استفاده می‌کنیم. فرض کنیم (u, v) یک یال مطمئن برای A نیست. از آنجایی که درخت پوشای کمینه همبند است و در آن دور وجود ندارد، u و v باید از طریق یک مسیر یکتا به یکدیگر متصل شده باشند. حال یالی که S و $V - S$ را به یکدیگر متصل می‌کند از درخت پوشای کمینه حذف می‌کنیم و (u, v) را جایگزین آن می‌کنیم. درخت پوشای به دست آمده هزینه‌اش از درخت قبلی بیشتر نیست و بنابراین کمینه است. پس یال (u, v) یک یال مطمئن است.

الگوریتم کروسکال

- الگوریتم کروسکال ابتدا به ازای هر رأس یک مجموعهٔ مجزا ایجاد می‌کند. سپس برای یافتن یال مطمئن در هر مرحله از بین همهٔ یال‌هایی که دو رأس در دو مجموعهٔ مجزا را به یکدیگر متصل می‌کنند، یال (u, v) با کمترین وزن را انتخاب می‌کند. وقتی یال (u, v) به عنوان یک یال از درخت پوشای کمینه انتخاب شد، مجموعه‌ای که رأس u در آن قرار دارد به مجموعه‌ای که رأس v در آن قرار دارد متصل می‌شوند.
- الگوریتم کروسکال یک الگوریتم حریصانه است زیرا در هرگام، یالی را اضافه می‌کند که کمترین وزن را دارد و در نهایت درخت به دست آمده دارای کمترین وزن خواهد بود.

الگوریتم کروسکال

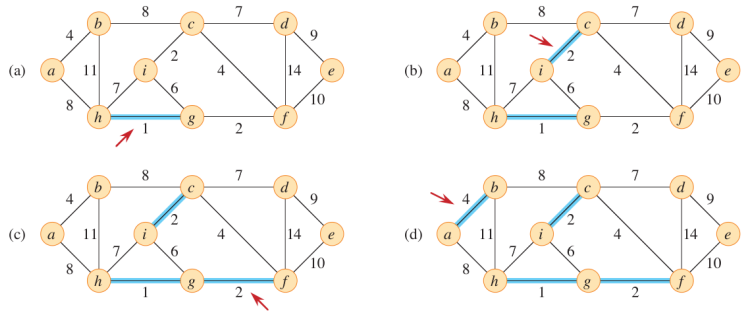
- الگوریتم کروسکال در زیر نشان داده شده است.

Algorithm Minimum Spanning Tree – Kruskal

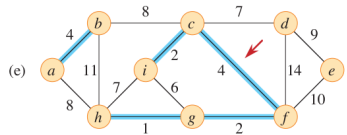
```
   function MST-KRUSKAL(G,w)
1: A = ∅
2: for each vertex v ∈ G.V do
3:    Make-Set(v)
4: creat a single list of the edges in G.E
5: sort the list of edges into monotonically increasing order by weight w
6: for each edge(u,v) taken from the sorted list in order do
7:    if Find-Set(u) ≠ Find-Set(v) then
8:       A = A ∪ (u,v)
9:       Union(Find-Set(u),Find-Set(v))
10: return A
```

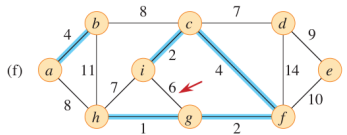
الگوریتم کروسکال

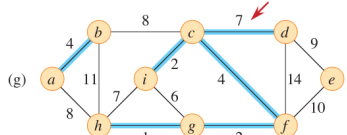
- در شکل زیر روند اجرای الگوریتم کروسکال نشان داده شده است.

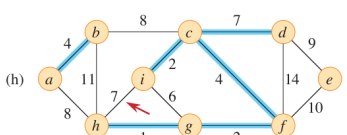


الگوریتم کروسکال

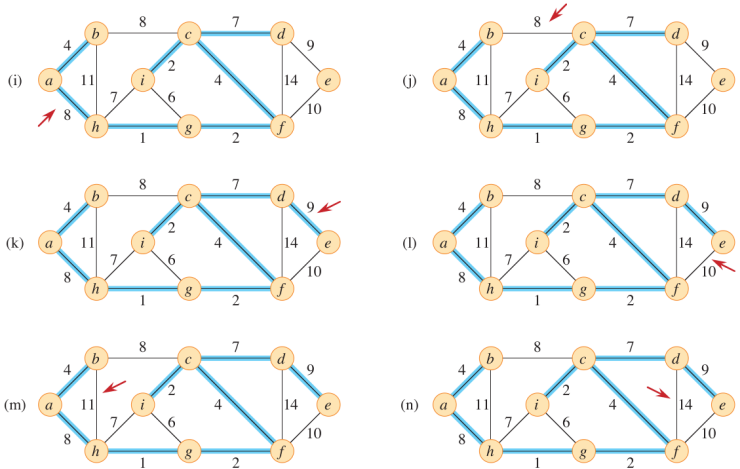








الگوریتم کروسکال



الگوریتم کروسکال

- در الگوریتم کروسکال، زمان لازم برای مرتب‌سازی یال‌ها $O(|E|\lg|E|)$ است.
- زمان لازم برای بررسی همهٔ یال‌ها $O(|E|)$ است.
- بنابراین، زمان اجرای الگوریتم کروسکال در مجموع $O(|E|\lg|E|)$ است.
- همچنین با توجه به اینکه $|E| < |V|^2$ ، داریم $\lg|E| < 2\lg|V|$ و بنابراین $\lg|E| = O(\lg|V|)$. پس می‌توانیم بگوییم زمان اجرای الگوریتم کروسکال برابراست با $O(|E|\lg|V|)$.

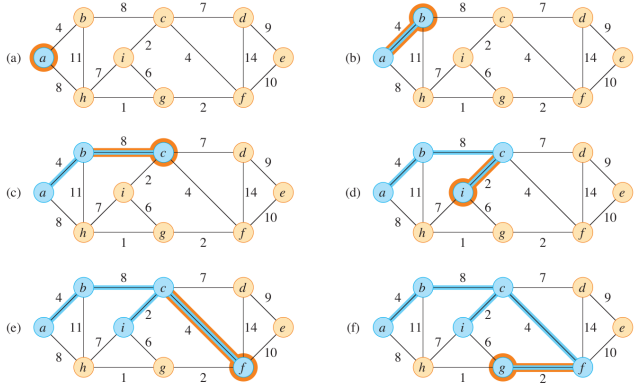
الگوریتم پریم

- ویژگی الگوریتم پریم [1] این است که یال‌های مجموعهٔ A (زیرمجموعهٔ یال‌های درخت پوشای کمینه) همیشه یک درخت را تشکیل می‌دهند.
- درخت پوشای کمینه با رأس ریشهٔ r آغاز می‌شود تا در نهایت همهٔ یال‌های V را پوشش دهد. در هرگام یک یال سبک به درخت A افزوده می‌شود که A را به یک رأس متصل می‌کند.
- این الگوریتم نیز یک الگوریتم حریصانه است، زیرا در هر مرحله یک یال با وزن کمینه به درخت افزوده می‌شود.

[1] Prim's algorithm

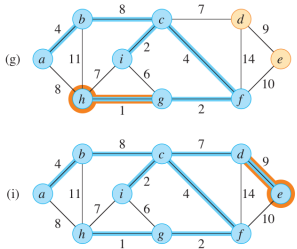
الگوریتم پریم

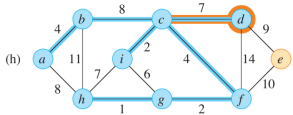
- شکل زیر روند اجرای الگوریتم پریم را نشان می‌دهد.



الگوریتم پریم

- شکل زیر روند اجرای الگوریتم پریم را نشان می‌دهد.





الگوریتم پریم

- الگوریتم پریم در زیر نشان داده شده است.

Algorithm Minimum Spanning Tree – Prim

```
   function MST-PRIM(G,w,r)
1: for each vertex u ∈ G.V do
2:     u.key = ∞
3:     u.pred = Nil
4: r.key = 0
5: Q = ∅
6: for each vertex u ∈ G.V do
7:     Insert(Q,u)
```

الگوریتم پریم

Algorithm Minimum Spanning Tree – Prim

```
 8: while Q ≠ ∅ do
 9:     u = Extract-Min(Q)      ▷ add u to the tree
10:     for each vertex u in G.Adj[u] do      ▷ update keys of u's non-tree
   neighbors
11:         if v ∈ Q and w(u,v) < v.key then
12:             v.pred = u
13:             v.key = w(u,v)
14:             Decrease-Key(Q,v,w(u,v))
```

الگوریتم پریم

- برای اضافه کردن یال جدید به درخت A ، الگوریتم از صف اولویت Q استفاده می‌کند که در آن رئوسی که به درخت افزوده نشده‌اند نگهداری می‌شود.
- برای هر رأس v ، ویژگی $v.key$ وزن کمینه یالی است که v را به یک رأس دیگر در درخت متصل می‌کند.
- مقدار $v.pred$ پدر رأس v را در درخت مشخص می‌کند.
- به طور ضمنی در این الگوریتم مجموعه A حاوی $\{(v, v.pred) : v \in V - \{r\} - Q\}$ است.
- وقتی الگوریتم به پایان می‌رسد، صف اولویت خالی می‌شود و در نتیجه داریم :

$$A = \{(v, v.pred) : v \in V - \{r\}\}$$

الگوریتم پریم

- زمان اجرای الگوریتم پریم را به صورت زیر تحلیل می‌کنیم.
- حلقهٔ `while` به تعداد $|V|$ بار تکرار می‌شود و از آنجایی که عملیات `Extract-Min` در زمان $O(\lg|V|)$ اجرا می‌شود، زمان لازم برای انجام این عملیات $O(|V|\lg|V|)$ است.
- حلقهٔ `for` در خطوط ۱۰ تا ۱۴ جمعاً $O(|E|)$ بار تکرار می‌شود. هر فراخوانی `Decrease-Key` در زمان $O(\lg|V|)$ اجرا می‌شود، بنابراین الگوریتم پریم در زمان $O(|V|\lg|V| + |E|\lg|V|)$ اجرا می‌شود. پس زمان اجرای الگوریتم پریم برابر با $O(|E|\lg|V|)$ است.

کوتاهترین مسیر از یک مبدأ

- فرض کنید می‌خواهید از شهری به شهر دیگر بروید و برای کاهش هزینه می‌خواهید کوتاهترین مسیر را انتخاب کنید. اطلاعات همهٔ راه‌ها و شهرها و فاصلهٔ بین شهرها را در اختیار دارید. چگونه می‌توانیم با این اطلاعات کوتاهترین مسیر را انتخاب کنیم؟
- یک راه ساده این است که همهٔ مسیرها را به دست آورده و طول آنها را با یکدیگر مقایسه کنیم، اما زمان لازم برای انجام چنین الگوریتم آنقدر زیاد است که در عمل مورد استفاده نیست.
- در اینجا الگوریتمی برای محاسبهٔ جواب این مسئله به طور کارامد ارائه می‌کنیم.

کوتاهترین مسیر از یک مبدأ

- ورودی مسئلهٔ کوتاهترین مسیر [1] گراف جهت‌دار وزن‌دار $G = (V, E)$ با تابع وزن $w : E \to \mathbb{R}$ است که به ازای هر یال وزن آن را باز می‌گرداند. وزن مسیر $p = \langle v_0, v_1, \cdots, v_k \rangle$ که به صورت $w(p)$ نشان داده می‌شود برابراست با مجموع وزن همهٔ یال‌های مسیر:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- وزن کوتاهترین مسیر از رأس u به v را به صورت زیر تعریف می‌کنیم.

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{اگر مسیری از } u \text{ به } v \text{ وجود داشته باشد} \\ \infty & \text{در غیر اینصورت} \end{cases}$$

[1] shortest path problem

کوتاهترین مسیر از یک مبدأ

- کوتاهترین مسیر از رأس u به v مسیر p است که وزن آن برابر با وزن کوتاهترین مسیر از u به v باشد : $w(p) = \delta(u, v)$
- در مثال پیدا کردن مسیر بین دو شهر، شهرها رأس‌های گراف، و جاده‌های بین دو شهر یال‌های گراف و فاصله جاده‌های بین دو شهر وزن یال‌ها هستند.
- الگوریتم جستجوی اول سطح در واقع یک الگوریتم کوتاهترین مسیر برای یک گراف بدون وزن است یعنی گرافی که در آن وزن یال‌ها برابر با مقدار واحد است.

کوتاهترین مسیر از یک مبدأ

- در الگوریتم‌های کوتاهترین مسیر از روشی به نام آزادسازی [1] استفاده می‌کنیم.
- به ازای هر رأس $v \in V$ الگوریتم کوتاهترین مسیر از یک رأس [2] یک متغیر به نام $v.\mathrm{d}$ نگه‌می‌دارد که یک کران بالا برای کوتاهترین مسیر از s به v است.
- مقدار $v.\mathrm{d}$ را تخمین کوتاهترین مسیر [3] می‌نامیم.

[1] relaxation
[2] single-source shortest path
[3] shortest-path estimate

کوتاهترین مسیر از یک مبدأ

- برای مقداردهی اولیه تخمین‌ها و رئوس سَلَف (ماقبل) در مسئله کوتاهترین مسیر به صورت زیر عمل می‌کنیم.

Algorithm Initialize-Single-Source

```
   function INITIALIZE-SINGLE-SOURCE(G,s)
1: for each vertex v ∈ G.V do
2:     v.d = ∞
3:     v.pred = Nil
4: s.d = 0
```

کوتاهترین مسیر از یک مبدأ

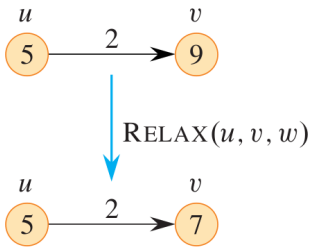
- روند آزادسازی یال (u, v) بدین صورت است که بررسی می‌کنیم آیا با عبور از u کوتاهترین مسیر از s به v بهبود پیدا می‌کند یا خیر. اگر مقدار کوتاهترین مسیر بهبود پیدا می‌کند $v.d$ و $v.pred$ را به روز رسانی می‌کنیم.
- الگوریتم آزادسازی در زیر نشان داده شده است.

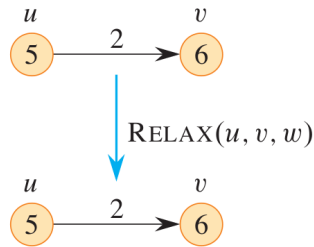
Algorithm Relax

```
   function RELAX(u,v,w)
1: if v.d > u.d + w(u,v) then
2:    v.d = u.d + w(u,v)
3:    v.pred = u
```

کوتاهترین مسیر از یک مبدأ

- در شکل زیر دو مثال از آزادسازی یک یال نشان داده شده است. در یکی از مثال‌ها تخمین کوتاهترین مسیر کاهش پیدا می‌کند و در مثال دیگر تغییری پیدا نمی‌کند.





کوتاهترین مسیر از یک مبدأ

- در الگوریتم بلمن فورد هر یال $|V| - 1$ بار آزادسازی می‌شود، اما در الگوریتم دایکسترا هر یال فقط یک بار آزادسازی می‌شود.

الگوریتم بلمن–فورد

- الگوریتم بلمن–فورد [1] مسئله کوتاهترین مسیر را در حالت کلی حل می‌کند وقتی وزن یال‌ها می‌توانند منفی نیز باشند.
- به ازای یک گراف دلخواه $G = (V, E)$ با یال‌های وزن‌دار و رأس مبدأ s و تابع وزن $w : E \to \mathbb{R}$ الگوریتم بلمن فورد در صورتی که یک دور با وزن منفی وجود داشته باشد که از مبدأ قابل دسترسی باشد، مقدار نادرست را باز می‌گرداند، بدین معنی که کوتاهترین مسیر وجود ندارد. اما اگر چنین دوری وجود نداشته باشد، الگوریتم بلمن فورد کوتاهترین مسیر را از مبدأ به همهٔ رئوس را باز می‌گرداند.

[1] Bellman–Ford algorithm

الگوریتم بلمن–فورد

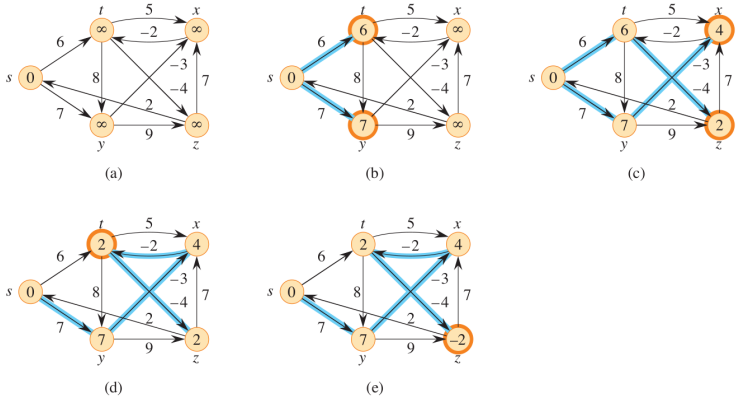
- الگوریتم بلمن فورد در زیر توصیف شده است.

Algorithm Bellman-Ford

```
   function BELLMAN-FORD(G,w,s)
1: Initialize-Single-Source(G, s)
2: for i = 1 to |G.V| - 1 do
3:    for each edge (u,v) ∈ G.E do
4:       Relax(u,v,w)
5: for each edge (u,v) ∈ G.E do
6:    if v.d > u.d + w(u,v) then
7:       return False
8: return True
```

الگوریتم بلمن–فورد

- یک مثال از اجرای الگوریتم بلمن فورد در زیر نشان داده شده است.



الگوریتم بلمن–فورد

- مقداردهی اولیه در خط ۱ در زمان $\Theta(V)$ اجرا می‌شود.
- اگر گراف با لیست مجاورت نمایش داده شود، بررسی همهٔ یال‌ها به زمان $\Theta(V + E)$ نیاز خواهد داشت.
- در حلقه خطوط ۲ تا ۴ هر یک از $|V| - 1$ تکرارهای حلقه، در زمان $\Theta(V + E)$ اجرا می‌شود.
- حلقه خطوط ۵ تا ۷ در زمان $O(V + E)$ اجرا می‌شود.
- بنابراین الگوریتم بلمن فورد در زمان $O(V^2 + VE)$ اجرا می‌شود.

الگوریتم بلمن–فورد

- برای اثبات درستی الگوریتم بلمن فورد نشان می‌دهیم اگر هیچ دوری با وزن منفی وجود نداشته باشد، این الگوریتم به درستی کوتاهترین مسیر را برای همهٔ رئوس از یک رأس مبدأ محاسبه می‌کند.
- فرض کنید $G = (V, E)$ یک گراف وزن‌دار جهت‌دار با رأس مبدأ s و تابع وزن $w : E \to \mathbb{R}$ باشد و فرض کنید G هیچ دوری با وزن منفی نداشته باشد که از s قابل دسترسی باشد. آنگاه بعد از $|V| - 1$ تکرار در حلقهٔ خطوط ۲ تا ۴ الگوریتم بلمن فورد به دست می‌آوریم $v.d = \delta(s, v)$ به ازای همه رئوس v که از s قابل دسترس هستند.

الگوریتم بلمن–فورد

- اثبات : یک رأس v را در نظر بگیرید که از s قابل دسترس است و فرض کنید $p = \langle v_0, v_1, \cdots, v_k \rangle$ به طوری‌که $v_0 = s$ و $v_k = v$ و p کوتاهترین مسیر از s به v باشد.
- از آنجایی که کوتاهترین مسیر باید یک مسیر ساده باشد، p حداکثر $|V| - 1$ یال دارد و بنابراین $k \leqslant |V| - 1$
- هر یک از $|V| - 1$ تکرار در حلقه خطوط ۲ تا ۴ همه $|E|$ یال را آزادسازی می‌کند.

الگوریتم بلمن–فورد

- بعد از یک بار تکرار حلقه یال (v_0, v_1) آزاد سازی می‌شود و بنابراین مسیر $p = \langle v_0, v_1 \rangle$ کوتاهترین مسیر از v_0 به v_1 خواهد بود.
- بعد از دو بار تکرار حلقه، یال (v_1, v_2) برای بار دوم آزادسازی می‌شود بنابراین مسیر $p = \langle v_0, v_1, v_2 \rangle$ کوتاهترین مسیر از v_0 به v_2 خواهد بود.
- بنابراین در تکرار i ام، به ازای $i = 1, 2, \cdots, k$ یال (v_{i-1}, v_i) آزادسازی می‌شود و مسیر $p = \langle v_0, ..., v_i \rangle$ کوتاهترین مسیر از v_0 به v_i خواهد بود.
- پس از $|V| - 1$ بار آزادسازی یال‌ها به دست می‌آوریم:

$$v_k.d = \delta(s, v_k)$$

الگوریتم دایکسترا

- الگوریتم دایکسترا مسئله کوتاهترین مسیر برای گراف وزن‌دار جهت‌دار $G = (V, E)$ را وقتی وزن‌ها منفی نباشند حل می‌کند. به عبارت دیگر به ازای هر یال $(u, v) \in E$ در الگوریتم دایکسترا لازم است داشته باشیم $w(u, v) \geqslant 0$.
- با یک پیاده‌سازی بهینه، الگوریتم دایکسترا می‌تواند در زمان کمتری نسبت به الگوریتم بلمن–فورد مسئله را حل کند.

الگوریتم دایکسترا

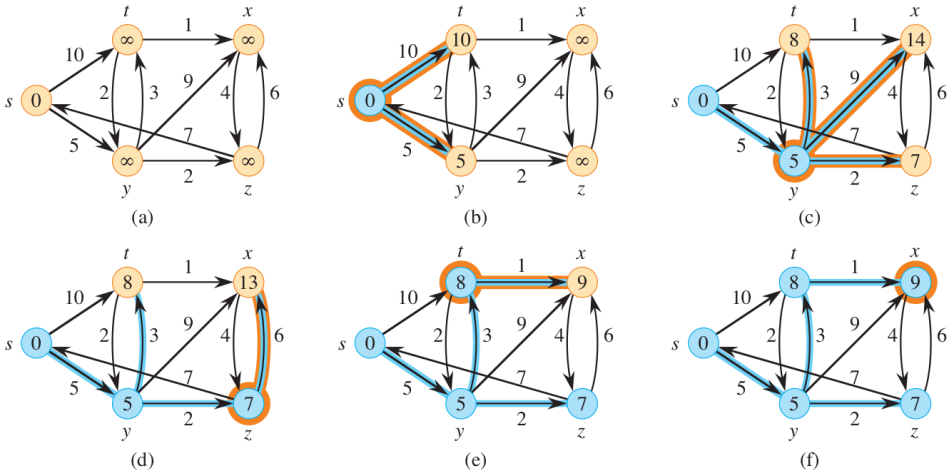
- الگوریتم دایکسترا به صورت زیر است.

Algorithm Dijkstra(G،w،s)

```
   function DIJKSTRA(G, w, s)
 1: Initialize-Single-Source(G,s)
 2: S = ∅
 3: Q = ∅
 4: for each vertex u ∈ G.V do
 5:    Insert(Q,u)
 6: while Q ≠ ∅ do
 7:    u = Extract-Min(Q)
 8:    S = S ∪ {u}
 9:    for each vertex v in G.Adj[u] do
10:       Relax(u,v,w)
11:       if the call of Relax decreased v.d then
12:          Decrease-Key(Q,v,v.d)
```

الگوریتم دایکسترا

- یک مثال از الگوریتم دایکستر در شکل زیر نشان داده شده است.



الگوریتم دایکسترا

- مجموعهٔ S شامل رئوسی است که کوتاهترین مسیر از مبدأ برای آنها تعیین شده است.
- از آنجایی که الگوریتم دایکسترا همیشه نزدیک‌ترین رأس به مبدأ در $V - S$ را به S اضافه می‌کند، این الگوریتم یک الگوریتم حریصانه است.

الگوریتم دایکسترا

- برای تحلیل زمان اجرای الگوریتم دایکسترا از تحلیل تجمعی استفاده می‌کنیم.
- از آنجایی که هر رأس $u \in V$ به مجموعهٔ S فقط یک‌بار اضافه می‌شود، هر یال در لیست مجاورت $Adj[u]$ در حلقه `for` خطوط ۹ تا ۱۲ دقیقا یک‌بار در طول اجرای الگوریتم بررسی می‌شود.
- بنابراین این حلقه در مجموع به تعداد یال‌های گراف تکرار می‌شود و `Decrease-key` در مجموع حداکثر به تعداد یال‌ها تکرار می‌شود. هزینه بررسی همهٔ یال‌ها $O(V + E)$ است.
- زمان اجرای الگوریتم دایکسترا به پیاده‌سازی صف اولویت بستگی پیدا می‌کند.
- در یک پیاده‌سازی ساده صف اولویت توابع `Insert` و `Decrease-key` در زمان $O(1)$ و تابع `Extract-Min` در زمان $O(V)$ اجرا می‌شود.
- بنابراین زمان اجرای الگوریتم $O(V^2 + E) = O(V^2)$ است.

الگوریتم دایکسترا

- اگر صف اولویت با استفاده از هیپ پیاده‌سازی شود، توابع `Extract-Min` و `Decrease-key` در زمان $O(\lg V)$ اجرا می‌شوند.
- بنابراین زمان اجرای الگوریتم $O(V \lg V + E \lg V)$ خواهد بود. در یک گراف معمولی که تعداد یال‌ها از تعداد رئوس بیشتر است، الگوریتم دایکسترا در زمان $O(E \lg V)$ اجرا می‌شود.

کوتاهترین مسیر بین همهٔ جفت‌ها

- اکنون مسئله کوتاهترین مسیر بین همهٔ جفت رأس‌ها در گراف را بررسی می‌کنیم.
- یکی از کاربردهای این الگوریتم، پیدا کردن کوتاهترین مسیر بین هر دو شهر در یک اطلس جغرافیایی است. یکی از کاربردهای دیگر این الگوریتم پیدا کردن فاصلهٔ بین دو نقطه در یک شبکه کامپیوتری برای ارسال بسته‌ها به طور بهینه است.
- خروجی الگوریتم یک جدول به اندازهٔ $|V| \times |V|$ است که در سطر u و ستون v فاصله بین شهر u و شهر v را باز می‌گرداند.

کوتاهترین مسیر بین همهٔ جفت‌ها

- یک راه حل ساده این است که به ازای هر یک از رئوس گراف، آن رأس را مبدأ فرض کرده و الگوریتم کوتاهترین مسیر از رأس مبدأ را از هریک از رئوس گراف اجرا کنیم تا فاصله بین همهٔ رئوس به دست بیاید برای مثال اگر وزن یال‌ها مثبت باشند، می‌توان از الگوریتم دایکسترا به تعداد $|V|$ بار استفاده کرد که در مجموع کل محاسبات در زمان $O(V^3)$ اجرا می‌شود.
- اگر گراف یال‌هایی با وزن منفی داشته باشد، نمی‌توان از الگوریتم دایکسترا استفاده کرد. می‌توانیم در این‌صورت با استفاده از الگوریتم بلمن–فورد، کوتاهترین مسیر بین همهٔ جفت‌ها را در زمان $O(V^2E)$ محاسبه کنیم که در صورتی که گراف متراکم باشد، زمان اجرا برابر خواهد بود با $O(V^4)$.

کوتاهترین مسیر بین همهٔ جفت‌ها

- در این قسمت الگوریتمی ارائه می‌کنیم که کوتاهترین مسیر بین همهٔ رئوس را در زمان کمتری محاسبه کند.
- برای استفاده از این الگوریتم، گراف را با استفاده از ماتریس مجاورت نشان می‌دهیم.

کوتاهترین مسیر بین همهٔ جفت‌ها

- اگر رأس‌ها را با شماره‌های ۱، ۲، ... ، $|V|$ شماره گذاری کنیم، به طوری‌که تعداد رئوس برابر با n باشد، آنگاه یک ماتریس $n \times n$ که با $W = (w_{ij})$ نشان می‌دهیم، وزن یال‌ها را در گراف $G = (V, E)$ نشان می‌دهد، به طوری‌که

$$w_{ij} = \begin{cases} 0 & \text{اگر } i = j \\ w(i,j) & \text{اگر } (i,j) \in E \text{ و } i \neq j \\ \infty & \text{اگر } (i,j) \notin E \end{cases}$$

- خروجی الگوریتم یک جدول $n \times n$ خواهد بود به طوری‌که در سطر i و ستون j مقدار $\delta(i,j)$ قرار گیرد.

- خروجی الگوریتم کوتاهترین مسیر می‌تواند شامل یک ماتریس سلف ها [1] یا ماتریس رئوس ماقبل به نام $\Pi = (\pi_{ij})$ نیز باشد که در آن π_{ij} سلف (رأس ماقبل) رأس j را بر روی مسیری که از رأس i آغاز شده نشان می‌دهد. در صورتی که $i = j$ باشد یا هیچ مسیری از i به j وجود نداشته باشد، آنگاه π_{ij} برابر با NIL خواهد بود.

[1] predecessor matrix

کوتاهترین مسیر بین همهٔ جفت‌ها

- برای چاپ کوتاهترین مسیر از i به j با استفاده از ماتریس سلف‌ها می‌توانیم از الگوریتم زیر استفاده کنیم.

Algorithm Print-All-Pairs-Shortest-Path

```
   function PRINT-ALL-PAIRS-SHORTEST-PATH(Π,i,j)
1: if i == j then
2:    print i
3: else if π_ij == NIL then
4:    print "no path from" i "to" j "exists"
5: else
6:    Print-All-Pairs-Shortest-Path(Π,i,π_ij)
7:    print j
```

الگوریتم فلوید–وارشال

- الگوریتم فلوید–وارشال [1] مسئله کوتاهترین مسیر بین همهٔ جفت‌ها را در زمان $O(V^3)$ حل می‌کند. در این الگوریتم یال‌ها با وزن منفی می‌توانند وجود داشته باشند، اما دورها با وزن منفی در گراف ورودی مسئله مجاز نیستند. این الگوریتم یک الگوریتم از نوع برنامه‌ریزی پویا است.

[1] Floyed–Warshall algorithm

الگوریتم فلوید–وارشال

- رئوس گراف G را با اعداد صحیح شماره‌گذاری می‌کنیم. بنابراین خواهیم داشت $V = \{1, 2, \cdots, n\}$.
- برای حل این مسئله با استفاده از برنامه‌ریزی پویا ابتدا کوتاهترین مسیر بین رئوس i و j را با فرض اینکه هیچ رأسی در مسیر وجود نداشته باشد محاسبه می‌کنیم. در صورتی که یال (i, j) وجود داشته باشد، طول چنین مسیری برابر با وزن این یال است. سپس کوتاهترین مسیر بین رئوس i و j را با فرض بر اینکه فقط رأس 1 در مسیر وجود داشته باشد به دست می‌آوریم. سپس فرض می‌کنیم رأس 2 نیز وجود داشته باشد و کوتاهترین مسیر بین همهٔ جفت رئوس را با فرض اینکه رئوس میانی مسیر در مجموعهٔ $\{1, 2\}$ باشند محاسبه می‌کنیم.
- به همین ترتیب به ازای $1 \leqslant k \leqslant n$ زیر مجموعهٔ $\{1, 2, \cdots, k\}$ از رئوس را در نظر می‌گیریم و به ازای هر جفت از رئوس $i, j \in V$ ، همه مسیرها از i به j را که از رئوس $\{1, 2, \cdots, k\}$ می‌گذرند را در نظر می‌گیریم و فرض می‌کنیم p کوتاهترین مسیر بین همهٔ این مسیرها باشد.

الگوریتم فلوید–وارشال

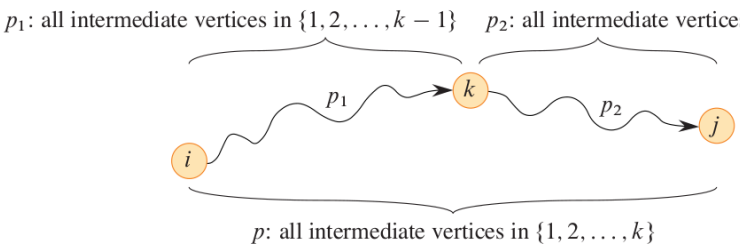
- حال حالت‌های زیر را در نظر می‌گیریم.

- اگر k یک رأس میانی در مسیر p نباشد، آنگاه همهٔ رئوس میانی مسیر p متعلق به مجموعهٔ $\{1, 2, \cdots, k-1\}$ هستند. بنابراین کوتاهترین مسیر از رأس i به رأس j با رئوس میانی $\{1, 2, \cdots, k\}$، همان کوتاهترین مسیر از i به j با رئوس میانی در مجموعهٔ $\{1, 2, \cdots, k-1\}$ است.

- اگر k یک رأس میانی در مسیر p باشد، آنگاه مسیر p را به دو قسمت $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$ تقسیم می‌کنیم. چون رأس k یک رأس میانی در مسیر p_1 نیست، همهٔ رئوس میانی در مسیر p_1 به مجموعهٔ $\{1, 2, \cdots, k-1\}$ تعلق دارند. بنابراین p_1 کوتاهترین مسیر از i به k با همهٔ رئوس میانی $\{1, 2, \cdots, k-1\}$ است. همچنین p_2 کوتاهترین مسیر از رأس k به رأس j با همهٔ رئوس میانی در مجموعهٔ $\{1, 2, \cdots, k-1\}$ است.

الگوریتم فلوید–وارشال

- شکل زیر این تقسیم بندی مسیر را نشان می‌دهد.



الگوریتم فلوید–وارشال

- بر اساس مشاهدهٔ قبلی می‌توانیم جواب مسئله کوتاهترین مسیر بین جفت‌ها را به صورت یک رابطهٔ بازگشتی بیان کنیم.
- فرض کنید $d_{ij}^{(k)}$ طول کوتاهترین مسیر از i به j باشد به طوری‌که رئوس میانی متعلق به مجموعه $\{1, 2, \cdots, k\}$ باشند.

الگوریتم فلوید–وارشال

- وقتی $k = 0$ است، یک مسیر از رأس i به رأس j که هیچ رأس میانی با شماره‌ای بزرگ‌تر از 0 نداشته باشد، درواقع هیچ رأس میانی ندارد. چنین مسیری حداکثر یک یال دارد، بنابراین $d_{ij}^{(0)} = w_{ij}$
- می‌توانیم مقدار $d_{ij}^{(k)}$ را به صورت بازگشتی تعریف کنیم.

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{اگر } k = 0 \\ min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} & \text{اگر } k \geqslant 1 \end{cases}$$

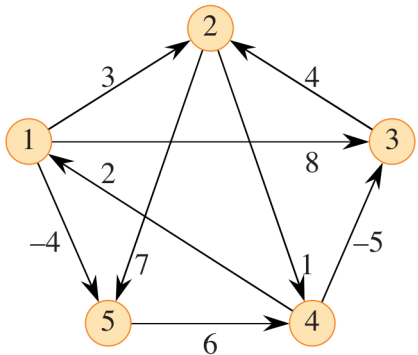
- چون برای هر مسیر، همهٔ رئوس میانی متعلق به مجموعهٔ $\{1, 2, \cdots, k\}$ هستند، بنابراین ماتریس $D^n = (d_{ij}^{(n)})$ شامل جواب پایانی است. به ازای هر $i, j \in V$ داریم $d_{ij}^{(n)} = \delta(i, j)$

الگوریتم فلوید–وارشال

- الگوریتم فلوید–وارشال از رابطه بازگشتی محاسبه شده استفاده می‌کند و توسط یک روند پایین به بالا مقدار بالا $d_{ij}^{(k)}$ را به ازای k های متفاوت از کوچک به بزرگ محاسبه می‌کند.

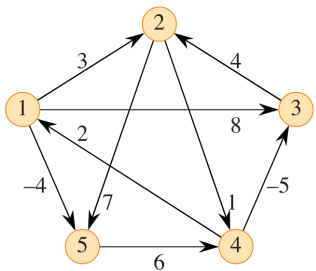
الگوریتم فلوید–وارشال

- گراف زیر را در نظر بگیرید.



الگوریتم فلوید–وارشال

- شکل زیر فرایند محاسبه ماتریس‌های $D^{(k)}$ و $\Pi^{(k)}$ را برای این گراف نشان می‌دهد.



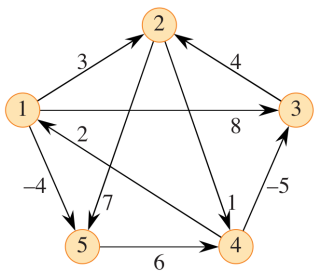
$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

الگوریتم فلوید–وارشال

- شکل زیر فرایند محاسبه ماتریس‌های $D^{(k)}$ و $\Pi^{(k)}$ را برای این گراف نشان می‌دهد.



$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

الگوریتم فلوید–وارشال

- الگوریتم فلوید–وارشال به صورت زیر است.

Algorithm Floyd-Warshall

```
   function FLOYD-WARSHALL(W,n)
1: D^(0) = W
2: for k = 1 to n do
3:    let D^(k) = (d_ij^(k)) be a new n × n matrix
4:    for i = 1 to n do
5:       for j = 1 to n do
6:          d_ij^(k) = min{d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1)}
7: return D^(n)
```

الگوریتم فلوید–وارشال

- در الگوریتم فلوید–وارشال سه حلقه `for` تودرتو وجود دارد. چون محاسبه خط ۶ برنامه در زمان $O(1)$ انجام می‌شود، بنابراین این الگوریتم در زمان $\Theta(n^3)$ محاسبه می‌شود.

الگوریتم فلوید–وارشال

- ماتریس سلف‌ها یا رئوس ماقبل Π نیز می‌تواند در زمان محاسبهٔ $D^{(0)}$ ، $D^{(1)}$ ، $\cdots$ ، $D^{(n)}$ محاسبه شود.
- به عبارت دیگر می‌توانیم Π^0 ، Π^1 ، $\cdots$ ، Π^n را محاسبه کنیم به طوری‌که $\Pi = \Pi^{(n)}$ و $\pi_{ij}^{(k)}$ سلف رأس j در کوتاهترین مسیری است که از رأس i شروع می‌شود و همهٔ رئوس میانی در مجموعه $\{1, 2, \cdots, k\}$ را شامل می‌شود.
- مقدار $\pi_{ij}^{(k)}$ را می‌توانیم به صورت بازگشتی تعریف کنیم.
- وقتی $k = 0$ است، کوتاهترین مسیر از i به j هیچ رأس میانی ندارد، بنابراین داریم :

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{اگر } i = j \text{ یا } w_{ij} = \infty \\ i & \text{اگر } i \neq j \text{ و } w_{ij} < \infty \end{cases}$$

الگوریتم فلوید–وارشال

- به ازای $k \geqslant 1$ ، اگر کوتاهترین مسیر از رأس i به j ، رأس k را به عنوان رأس میانی شامل نشود، آنگاه رأس ماقبل j در مسیری که با رئوس میانی $\{1, 2, \cdots, k\}$ از i آغاز شده است برابر است با رأس ماقبل j در مسیری که با رئوس میانی $\{1, 2, \cdots, k-1\}$ از i آغاز شده است.
- اگر کوتاهترین مسیر از رأس i به j ، رأس k را به عنوان رأس میانی شامل شود، به طوری که $i \rightsquigarrow k \rightsquigarrow j$ و $k \neq j$ ، آنگاه سَلَف رأس j در این مسیر همان سلف رأس j در مسیری است که از k آغاز می‌شود و شامل همهٔ رئوس در مجموعهٔ $\{1, 2, \cdots, k-1\}$ می‌شود.
- پس به ازای $k \geqslant 1$ داریم :

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{kj}^{(k-1)} & \text{اگر } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{ij}^{(k-1)} & \text{اگر } d_{ij}^{(k-1)} \leqslant d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$