به نام خدا

ساختمان داده

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

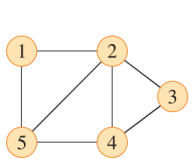
الگوریتم‌های گراف

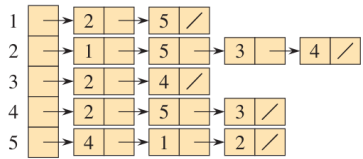
- در نمایش لیست مجاورت [1] برای گراف $G = (V, E)$ از آرایهٔ Adj شامل $|V|$ عنصر استفاده می‌کنیم. به ازای هر $u \in V$ ، لیست مجاورت $Adj[u]$ شامل رئوس v است به طوری‌که $(u, v) \in E$. پس $Adj[u]$ شامل همهٔ رئوسی است که در گراف G مجاور u هستند. در پیاده‌سازی این روش، عناصر این آرایه می‌توانند اشاره‌گر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

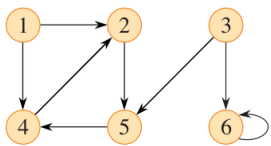
[1] adjacency-list representation

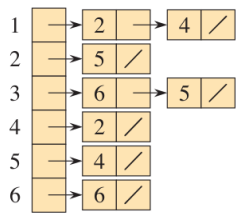
الگوریتم‌های گراف

- 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 $\Theta(V + E)$.

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

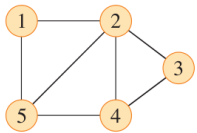
- در نمایش ماتریس مجاورت [1] برای گراف $G = (V, E)$ فرض می‌کنیم هر رأس یک شماره از 1 تا $|V|$ داشته باشد. سپس گراف G را با استفاده از ماتریس $A = (a_{ij})$ با اندازهٔ $|V| \times |V|$ نشان می‌دهیم به طوری‌که

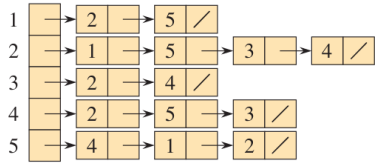
$$a_{ij} = \begin{cases} 1 & \text{اگر } (i,j) \in E \\ 0 & \text{در غیر اینصورت} \end{cases}$$

[1] adjacency matrix representation

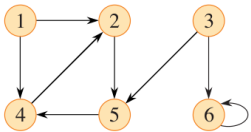
الگوریتم‌های گراف

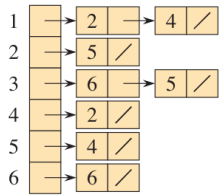
- در شکل زیر دو ماتریس مجاورت نشان داده شده‌اند.





	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 ، سَلَف [1] یا پدر رأس v و رأس v خَلَف [2] یا فرزند 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 پدر 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` پدر رأس v است.
- رأس `v.pred` پدر [1] رأس v است، و رأس v فرزند [2] رأس `v.pred` .

[1] predecessor
[2] successor

جستجوی سطح‌اول

- الگوریتم زیر جستجوی سطح‌اول را نشان می‌دهد.

Algorithm Breadth-First Search

```
   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)
```

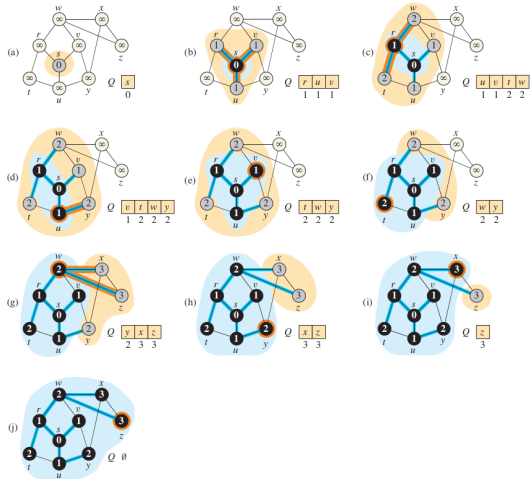
جستجوی سطح‌اول

Algorithm Breadth-First Search

```
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 آغازین 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)
```

جستجوی عمق‌اول

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
```

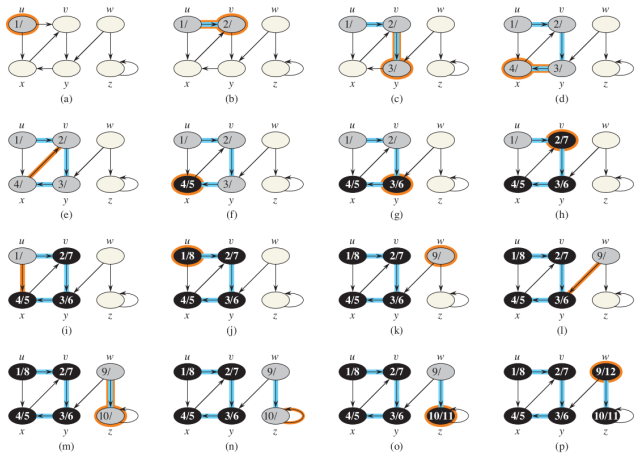
جستجوی عمق‌اول

- 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.

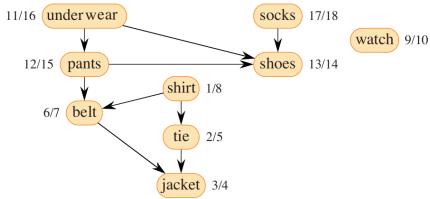
جستجوی عمق‌اول

- 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

کوتاهترین مسیر از یک مبدأ

- فرض کنید می‌خواهیم از شهری به شهر دیگر برویم و برای کاهش هزینه می‌خواهیم کوتاهترین مسیر را انتخاب کنیم. اطلاعات همهٔ راه‌ها و شهرها و فاصلهٔ بین شهرها را در اختیار داریم. 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 سطح‌اول 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 تخمین فاصله و رئوس پدر هر رأس 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

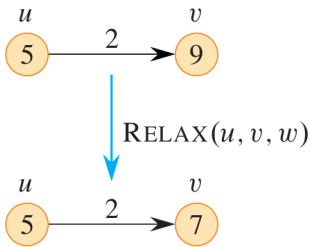
- با فرض اینکه کوتاهترین مسیر از مبدأ s به رأس u محاسبه شده است و $u.d$ به دست آمده است، روند آزادسازی یال (u, v) بدین صورت است که بررسی می‌کنیم آیا با عبور از u کوتاهترین مسیر از s به v بهبود پیدا می‌کند یا خیر. اگر مقدار کوتاهترین مسیر بهبود پیدا می‌کند $v.d$ و $v.pred$ را به روز رسانی می‌کنیم.
- الگوریتم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

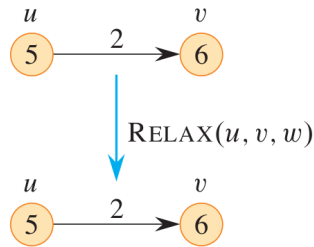
Algorithm Relax

```
   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.





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

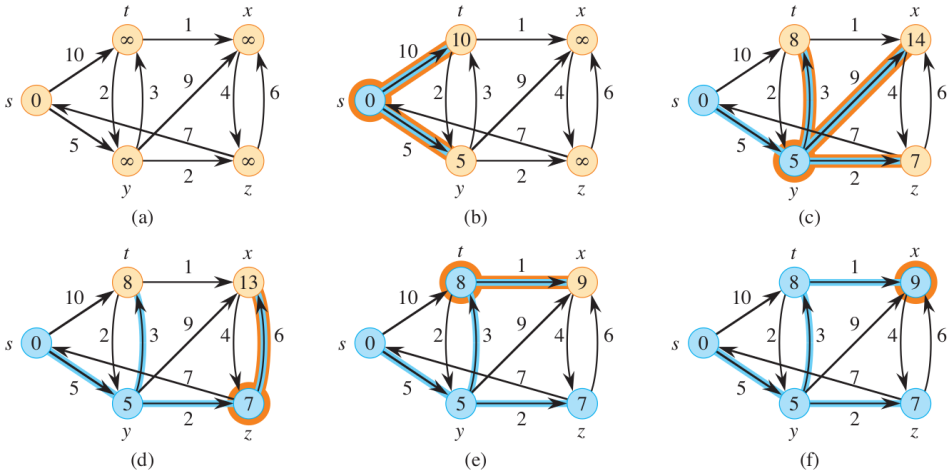
- الگوریتم دایکسترا به صورت زیر است.

Algorithm Dijkstra

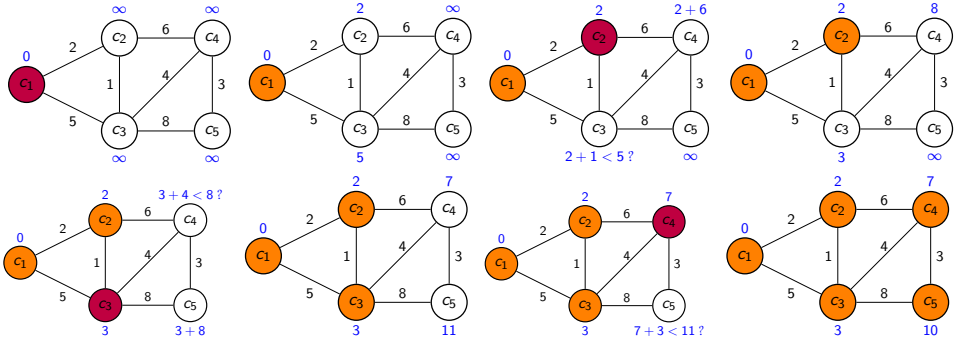
```
   function DIJKSTRA(G, w, s)
 1: Initialize-Single-Source(G,s)
 2: S = ∅
 3: Q = ∅
 4: for each vertex u ∈ G.V do
 5:    Insert(Q,u)
 6: while Q ≠ ∅ do
 7:    u = Extract-Min(Q)
 8:    S = S ∪ {u}
 9:    for each vertex v in G.Adj[u] do
10:       Relax(u,v,w)
11:       if the call of Relax decreased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



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 $O(|E|)$ است.
- حلقه `while` نیز به تعداد رئوس گراف تکرار می‌شود.
- 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 $O(|V|^2) = O(|V|^2 + |E|)$ 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 $O((|V|+|E|)\lg|V|)$ خواهد بود.
- در یک گراف همبند که تعداد یال‌ها بزرگتر یا مساوی تعداد رئوس است، الگوریتم دایکسترا در زمان $O(|E|\lg|V|)$ اجرا می‌شود.
- یک پیاده‌سازی بهینه‌تر نیز با استفاده از هرم فیبوناچی برای الگوریتم دایکسترا وجود دارد که زمان اجرا را به $O(|E|+|V|\lg|V|)$ کاهش می‌دهد که در اینجا به آن نمی‌پردازیم.