

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 کامپیوچر و دیگر شاخه‌های علوم بسیار پر استفاده‌اند. برای مثال برای مدلسازی یک شبکه کامپیوچری که از تعدادی کامپیوچر 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 مجموعه رأس‌ها<sup>۱</sup> ( $V$ ) و یال‌ها<sup>۲</sup> ( $E$ ) 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.

---

<sup>۱</sup> vertex set

<sup>۲</sup> edge set

# الگوریتم‌های گراف

- علاوه بر روش استاندارد نمایش یک گراف توسط مجموعه‌ها، می‌توانیم یک گراف را توسط مجموعه‌ای از لیست‌های مجاورت<sup>۱</sup> یا یک ماتریس مجاورت<sup>۲</sup> نشان دهیم.
- توسط لیست مجاورت می‌توان گراف‌های خلوت<sup>۳</sup> را که در آنها  $|E|$  بسیار کوچک‌تر از  $|V|^2$  است نمایش داد. وقتی گراف متراکم<sup>۴</sup> است، می‌توان از نمایش ماتریس مجاورت استفاده کرد.

---

<sup>۱</sup> adjacency lists

<sup>۲</sup> adjacency matrix

<sup>۳</sup> sparse graph

<sup>۴</sup> dense

# الگوریتم‌های گراف

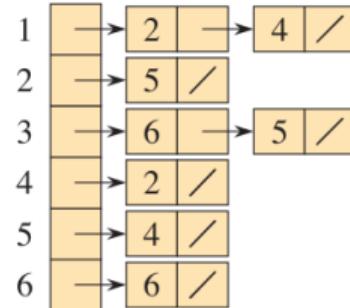
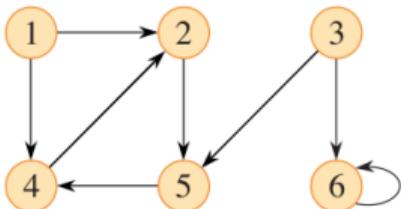
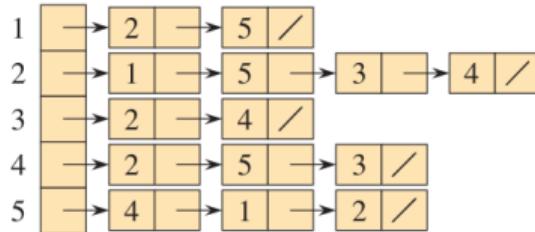
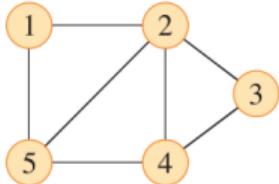
- در نمایش لیست مجاورت<sup>۱</sup> برای گراف  $G = (V, E)$  از آرایه  $\text{Adj}$  شامل  $|V|$  عنصر استفاده می‌کنیم. به ازای هر  $u \in V$ ، لیست مجاورت  $\text{Adj}[u]$  شامل رئوس  $v$  است به طوری که  $(u, v) \in E$ . پس  $[u, v] \in \text{Adj}[u]$  شامل همه رئوسی است که در گراف  $G$  مجاور  $u$  هستند. در پیاده‌سازی این روش، عناصر این آرایه می‌توانند اشاره‌گر به رئوس مجاور باشند.

---

<sup>۱</sup> adjacency-list representation

# الگوریتم‌های گراف

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



# الگوریتم‌های گراف

- اگر  $G$  یک گراف جهت دار باشد، مجموع اندازه همه لیست‌های مجاورت برابر است با  $|E|$  ، زیرا هر یک از یال‌های  $(u, v)$  توسط درایه  $\text{Adj}[u]$  نشان داده می‌شود.
- اگر  $G$  یک گراف بدون جهت باشد، آنگاه مجموع اندازه همه لیست‌های مجاورت برابر است با  $2|E|$  ، زیرا هر یک از یال‌های  $(u, v)$  هم در  $\text{Adj}[u]$  و هم در  $\text{Adj}[v]$  نمایش داده می‌شود.
- فضای حافظه‌ای که لیست مجاورت برای نگهداری گراف نیاز دارد برابر است با  $\Theta(V + E)$  .

# الگوریتم‌های گراف

- توسط لیست مجاورت می‌توانیم گراف‌های وزن دار<sup>۱</sup> را نیز نمایش دهیم. در یک گراف وزن دار، هریک از یال‌ها دارای یک وزن است که توسط تابع وزن  $w: E \rightarrow \mathbb{R}^2$  تولید می‌شود.
- برای مثال فرض کنید  $G = (V, E)$  یک گراف وزن دار با تابع وزن  $w$  باشد. آنگاه می‌توانیم وزن  $w(u, v)$  از یال  $E \in (u, v)$  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  $\text{Adj}[u]$  است.
- ماتریس مجاورت برای پیدا کردن یک یال می‌تواند از لیست مجاورت سریع‌تر عمل کند.

---

<sup>۱</sup> weighted graphs

<sup>۲</sup> weight function

# الگوریتم‌های گراف

- در نمایش ماتریس مجاورت<sup>۱</sup> برای گراف  $G = (V, E)$  فرض می‌کنیم هر رأس یک شماره از ۱ تا  $|V|$  داشته باشد. سپس گراف  $G$  را با استفاده از ماتریس  $A = (a_{ij})$  با اندازه  $|V| \times |V|$  نشان می‌دهیم به طوری که

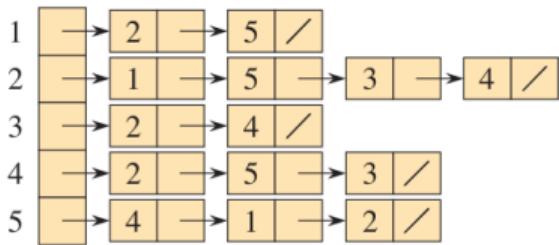
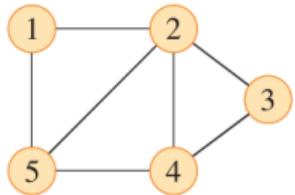
$$a_{ij} = \begin{cases} 1 & \text{اگر } (i, j) \in E \\ 0 & \text{در غیر اینصورت} \end{cases}$$

---

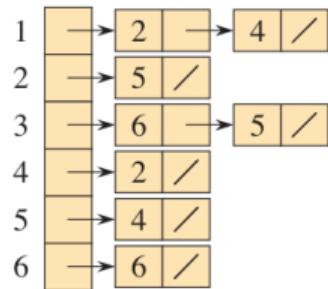
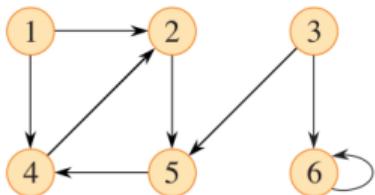
<sup>۱</sup> adjacency matrix representation

# 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 | 0 | 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 درایه  $A[u, v]$  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  $A[u, v] = A[v, u]$  برابراست با  $A$  بنابراین ماتریس مجاورت برابر با ترانهاده<sup>۱</sup> 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.

---

<sup>۱</sup> transpose

# الگوریتم‌های گراف

- با استفاده از ماتریس مجاورت می‌توانیم یک گراف وزن دار را نیز نمایش دهیم.
- برای مثال، اگر  $G = (V, E)$  یک گراف وزن دار با تابع وزن  $w$  باشد، می‌توان وزن  $w(u, v)$  برای یال  $\in E$  را به عنوان درایه 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 سطح اول

- جستجوی سطح اول<sup>۱</sup> 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 سطح اول استفاده می‌کند.
- به ازای گراف دلخواه  $(V, E) = G$  و یک رأس مبدأ<sup>۲</sup> به نام  $s$ ، الگوریتم جستجوی سطح اول همه یال‌های گراف  $G$  را با شروع از رأس  $s$  بررسی می‌کند.
- با شروع از رأس  $s$ ، الگوریتم سطح اول 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.

---

<sup>۱</sup> breadth-first search

<sup>۲</sup> source vertic

# جستجوی سطح اول

- یال‌هایی که در جستجوی سطح اول به ترتیب بررسی می‌شوند، یک درخت سطح اول می‌سازد که ریشه 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 سطح اول، بدین دلیل سطح اول 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 سطح اول، قبل از بررسی رئوس در عمق، همه رئوس در سطح را بررسی می‌کند. بنابراین برخلاف جستجوی عمق اول که به ازای هر رأس  $v$  یک رأس مجاور  $v$  ممکن است قبل از یک رأس مجاور  $v$  پیمایش شود، در جستجوی سطح اول همه رئوس مجاور  $v$  قبل از پیمایش رئوس مجاور مجاور  $v$  پیمایش می‌شوند.

# جستجوی سطح اول

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

---

<sup>1</sup> adjacent

<sup>2</sup> neighbour

# جستجوی سطح اول

- در الگوریتم جستجوی سطح اول 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. صفحی که در جستجوی سطح اول استفاده می‌شود، شامل همهٔ رئوس خاکستری است.
- رئوس خاکستری به ترتیب از صفات خارج می‌شوند و به رنگ سیاه تبدیل می‌شوند و رئوس سفید همسایه آنها که تاکنون پیمایش نشده‌اند به رنگ خاکستری تبدیل می‌شوند و وارد صفات می‌شوند.

# جستجوی سطح اول

- یک الگوریتم جستجوی سطح اول، یک درخت سطح اول می‌سازد که ریشه 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<sup>۱</sup> یا پدر رأس  $v$  و رأس  $v$  خلف<sup>۲</sup> یا فرزند رأس  $u$  در درخت سطح اول 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<sup>۳</sup> رأس  $v$  است و رأس  $v$  نواده<sup>۴</sup> رأس  $u$  است.

---

<sup>1</sup> predecessor

<sup>2</sup> successor

<sup>3</sup> ancestor

<sup>4</sup> descendant

# جستجوی سطح اول

- در الگوریتم جستجوی سطح اول که بررسی خواهیم کرد،  $v.\text{color}$  رنگ رأس  $v$  است که می‌تواند سفید، خاکستری یا سیاه باشد،  $v.d$  فاصله رأس  $v$  از رأس  $s$  است و  $v.\text{pred}$  پدر رأس  $v$  است.
- رأس  $v.\text{pred}$ <sup>1</sup> پدر<sup>2</sup> رأس  $v$  است، و رأس  $v$  فرزند<sup>2</sup> رأس  $v.\text{pred}$  است.

---

<sup>1</sup> predecessor

<sup>2</sup> successor

# جستجوی سطح اول

الگوریتم زیر جستجوی سطح اول 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)
```

---

---

## Algorithm Breadth-First Search

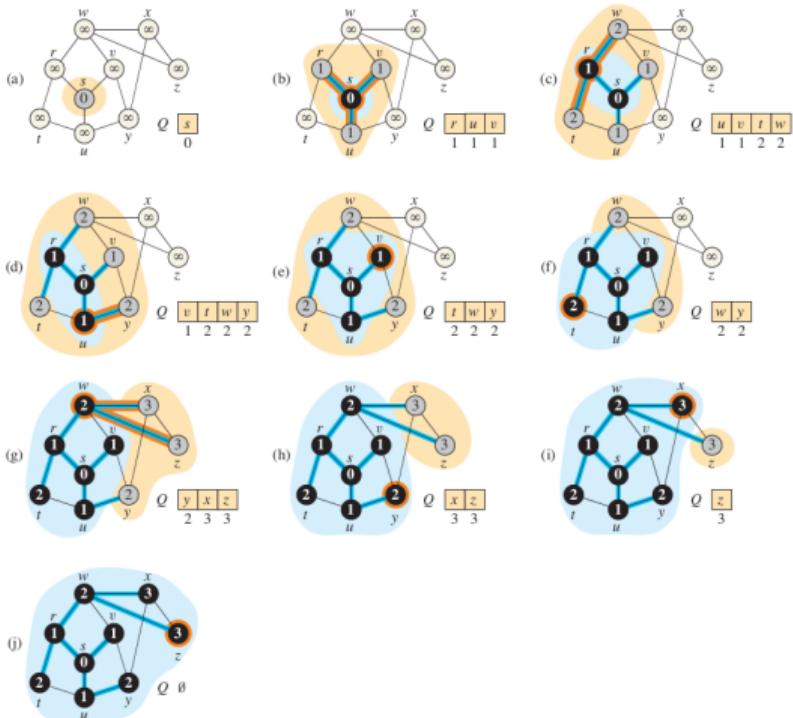
---

```
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 سطح اول

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



# جستجوی سطح اول

- تحلیل الگوریتم جستجوی سطح اول : برای تحلیل الگوریتم جستجوی سطح اول 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 یک بار انجام می‌شود و مجموع طول همه لیست‌های مجاورت برابر است با  $\Theta(E)$  ، بنابراین زمان لازم برای اجرای الگوریتم برابر است با  $O(V + E)$  . نتیجه می‌گیریم جستجوی سطح اول در زمان خطی نسبت به اندازه لیست مجاورت اجرا می‌شود.
- می‌توان ثابت کرد الگوریتم جستجوی سطح اول کوتاهترین مسیر از  $s$  به هریک از رئوس را محاسبه می‌کند.

# جستجوی عممق اول

- جستجوی عممق اول به جای اینکه جستجو را در سطح شروع کند و همه 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 روشن تر 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 همه کتاب‌ها می‌رویم تا این که در نهایت همه 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 همه کتاب‌ها را جستجو کنیم. در این مثال هیچ یک از جستجوها مزیتی بر دیگری ندارد چرا که مطلب مورد نظر ممکن است در صفحه 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 عمق اول

- جستجوی عمق اول یال‌های بررسی نشده رئوس تازه پیدا شده را زودتر از یال‌های بررسی نشده رئوس قبل‌پیدا 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 كامل گراف، الگوریتم یکی از رئوس را به عنوان مبدأ جدید انتخاب کرده و جستجو را از رأس جدید آغاز می‌کند.

# جستجوی عممق اول

- همانند جستجوی سطح اول، در جستجوی عممق اول 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 همه رئوس در لیست مجاورت آن پیدا شدند) به رنگ سیاه در می‌آید.

# جستجوی عممق اول

- در جستجوی عممق اول هر رأس دارای دو 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 همه رئوس لیست مجاورت آن پیدا شده‌اند و رأس  $v$  به رنگ مشکی درآمده است.

---

<sup>1</sup> time stamp

- الگوریتم زیر جستجوی عمق اول را نشان می دهد.

---

## Algorithm Depth-First Search

---

```
function DFS(G)
1: for each vertex  $u \in G.V$  do
2:    $u.color = \text{White}$ 
3:    $u.pred = \text{Nil}$ 
4: time = 0
5: for each vertex  $u \in G.V$  do
6:   if  $u.color == \text{White}$  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 عمق اول

- وقتی الگوریتم جستجوی عمق اول به اتمام می‌رسد هر رأس دارای دو برچسب زمان است که یکی زمان پیدا شدن<sup>۱</sup> و دیگری زمان به پایان رسیدن<sup>۲</sup> را نشان می‌دهد.

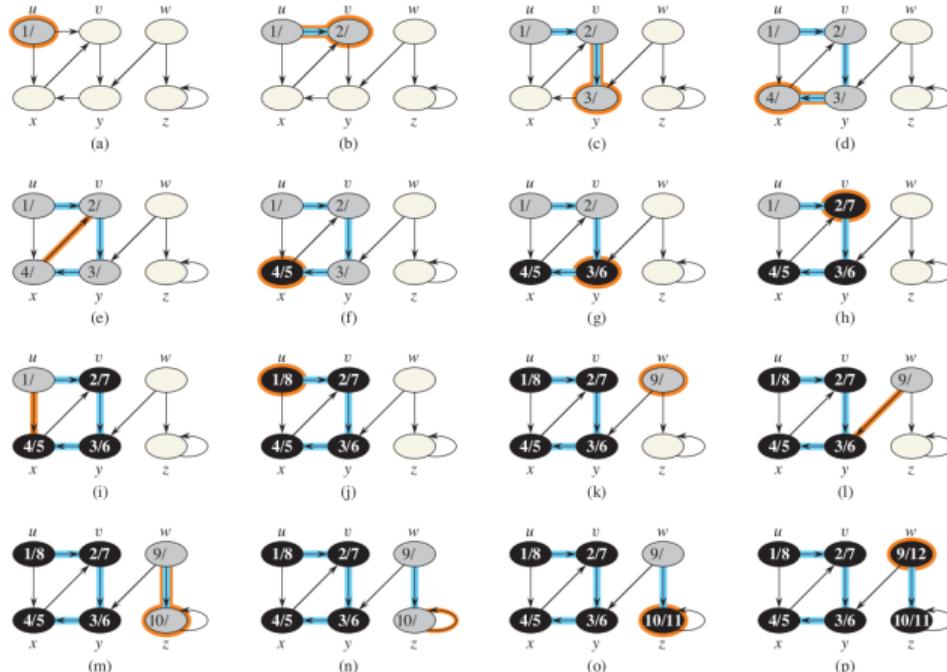
---

<sup>۱</sup> discovery time

<sup>۲</sup> finish time

# جستجوی عمق اول

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



# جستجوی عممق اول

- 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 همه رئوس، این حلقه  $\sum_{v \in V} |Adj[v]| = \Theta(E)$ .
- پس زمان اجرای الگوریتم جستجوی عممق اول  $\Theta(V + E)$  است.

# جستجوی عمقداول

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

# جستجوی عمق اول

- از جستجوی عمق اول در مرتب‌سازی توپولوژیکی<sup>۱</sup> یا مرتب‌سازی موضعی یک گراف بدون دور<sup>۲</sup> 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 آرایه مرتب شده قرار می‌گیرد.
- مرتب‌سازی توپولوژیکی تنها برای گراف‌های جهت‌دار بدون دور<sup>۳</sup> تعریف می‌شود.
- مرتب‌سازی توپولوژیکی به گونه‌ای است که اگر رئوس مرتب شده ببروی یک خط افقی قرار بگیرند، جهت همه یال‌های از چپ به راست است.

---

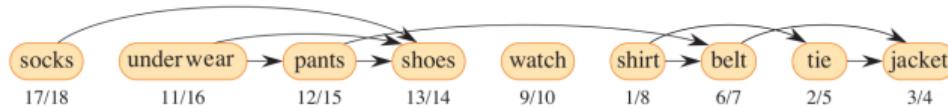
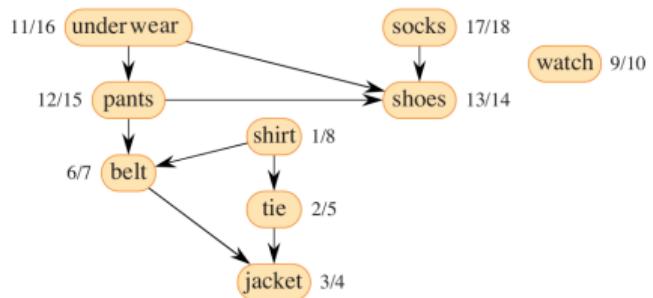
<sup>۱</sup> topological sort

<sup>۲</sup> acyclic graph

<sup>۳</sup> directed acyclic graph

# جستجوی عمق اول

- از یک گراف جهت دار بدون دور می توان برای نمایش دادن رویدادها<sup>۱</sup> استفاده کرد.
- برای مثال در شکل زیر برای یک گراف شامل تعدادی رویداد مرتب سازی توپولوژیکی انجام شده است.



<sup>1</sup> 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  $1 - n$  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  $w : E \rightarrow \mathbb{R}$  در نظر بگیرید. وزن یال  $e \in E$  برابر است با  $w(e)$ .

## 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 مجموعه  $T \subseteq E$  است که همه 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.
- زیرمجموعه  $T$  همه رأس‌های گراف را به یکدیگر متصل می‌کند و دارای هیچ دوری نیست، پس یک درخت را تشکیل می‌دهد. به چنین درختی، درخت پوشای<sup>1</sup> گفته می‌شود. مسئله درخت پوشای کمینه<sup>2</sup> به دنبال درخت پوشایی می‌گردد که وزن آن از همه درخت‌های پوشای دیگر کمتر باشد.

---

<sup>1</sup> spanning tree

<sup>2</sup> 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  $w : E \rightarrow \mathbb{R}$  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.

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

---

## Algorithm Generic-MST

---

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

---

- 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  $(u, v) \cup A$  نیز زیرمجموعه‌ای از درخت پوشای کمینه است.
- یالی که به  $A$  اضافه می‌شود را یک یال مطمئن<sup>1</sup> می‌نامیم زیرا ویژگی درخت را حفظ می‌کند.
- پس در گام اول قبل از شروع حلقه ویژگی درخت (که زیرمجموعه درخت پوشای کمینه است) برقرار است. در هر گام در حلقه تکرار ویژگی درخت حفظ می‌شود، پس در پایان یک درخت پوشای کمینه خواهیم داشت.

---

<sup>1</sup> 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<sup>۱</sup>  $(S, V - S)$  از یک گراف بدون جهت  $(V, E) = G$  یک تقسیم‌بندی از رئوس  $V$  است که در آن مجموعهٔ رئوس به دو قسمت تقسیم می‌شوند.
- می‌گوییم یال  $E \in \{(u, v) \mid u, v \in S, V - S\}$  از برش  $(S, V - S)$  عبور می‌کند<sup>۲</sup> اگر یک رأس یال در مجموعهٔ  $S$  و رأس دیگر یال در مجموعهٔ  $S - V$  قرار بگیرد.
- یالی را که از یک برش عبور می‌کند یال سبک<sup>۳</sup> می‌نامیم اگر وزن آن در بین همهٔ یال‌هایی که از برش عبور می‌کنند کمینه باشد. در یک برش ممکن است چند یال سبک هم وزن وجود داشته باشند.

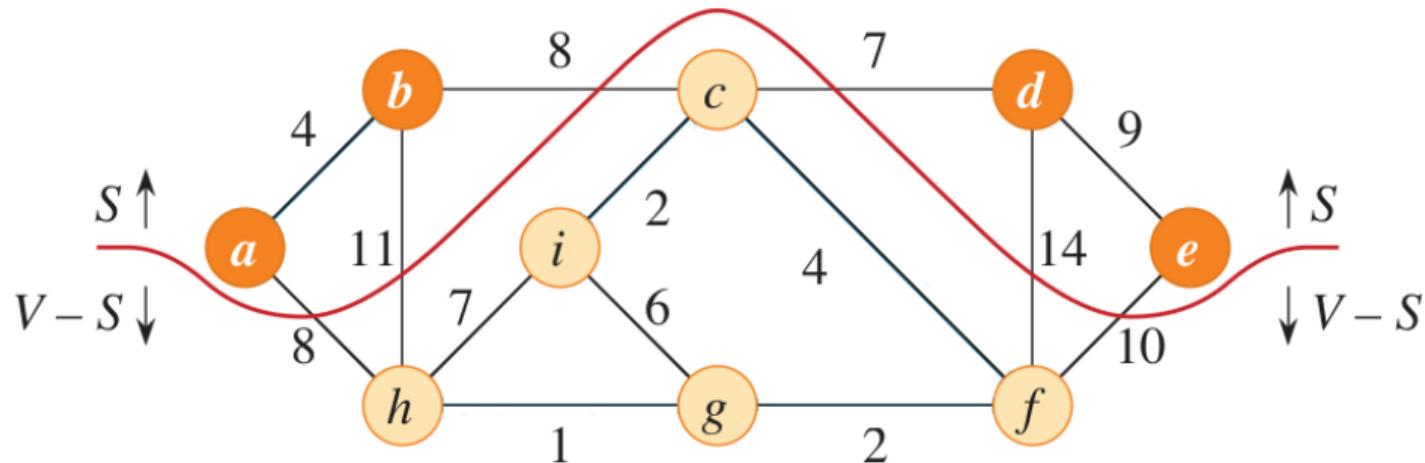
---

<sup>1</sup> cut

<sup>2</sup> crosses

<sup>3</sup> light edge

- شکل زیر یک برش را نشان می‌دهد. یال سبک در این برش یال  $(c, d)$  با وزن ۷ است.



## درخت پوشای کمینه

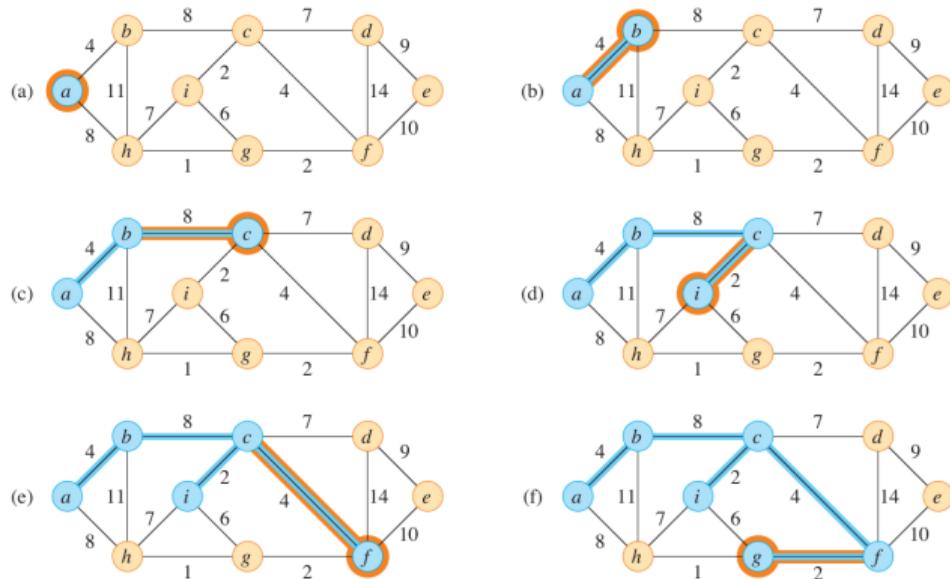
- قضیه: فرض کنید  $G = (V, E)$  یک گراف همبند بدون جهت با یال‌های وزن‌دار باشد و وزن‌ها با تابع  $w$  تعريف شده باشند. فرض کنید  $A$  یک زیرمجموعه از  $E$  باشد که در یک درخت پوشای کمینه برای  $G$  قرار گرفته باشد و فرض کنید  $(S, V - S)$  یک برش از  $G$  باشد که هیچ یالی در  $A$  از آن عبور نمی‌کند. 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 پریم مطابق با قضیه‌ای که بیان شد طراحی شده است.
- در الگوریتم پریم<sup>۱</sup> یال‌های مجموعه A (زیرمجموعهٔ یال‌های درخت پوشای کمینه) همیشه یک درخت را تشکیل می‌دهند.
- درخت پوشای کمینه با یک رأس ریشه آغاز می‌شود تا در نهایت همه یال‌ها را پوشش دهد. در هرگام یک یال سبک به درخت A افزوده می‌شود که A را به یک رأس متصل می‌کند.
- این الگوریتم یک الگوریتم حریصانه است، زیرا در هر مرحله یک یال با وزن کمینه به درخت افزوده می‌شود.

---

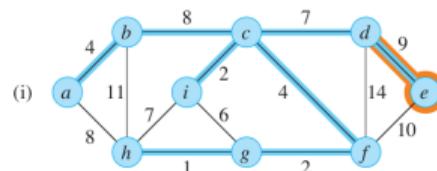
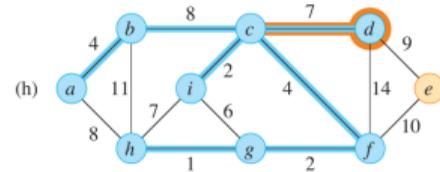
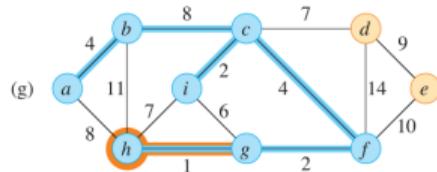
<sup>۱</sup> Prim's algorithm

- شکل زیر روند اجرای الگوریتم پریم را نشان می‌دهد.



# الگوریتم پریم

- شکل زیر روند اجرای الگوریتم پریم را نشان می‌دهد.



– 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)
```

---

---

## Algorithm Minimum Spanning Tree - Prim

---

```
8: while Q ≠ ∅ do
9:   u = Extract-Min(Q)      ▷ add u to the tree
10:  for each vertex v 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 صفت اولویت  $Q$  استفاده می کند که در آن رئوسی که به درخت افزوده نشده اند نگهداری می شود.
- برای هر رأس  $v$  ، ویژگی  $v.key$  وزن کمینه یالی است که  $v$  را به یک رأس دیگر در درخت متصل می کند.
- مقدار  $v.pred$  پدر رأس  $v$  را در درخت مشخص می کند.

# الگوریتم پریم

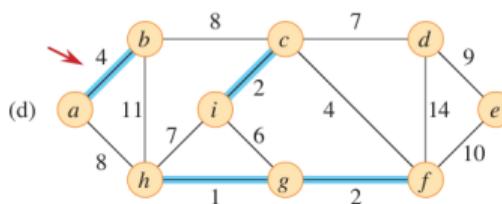
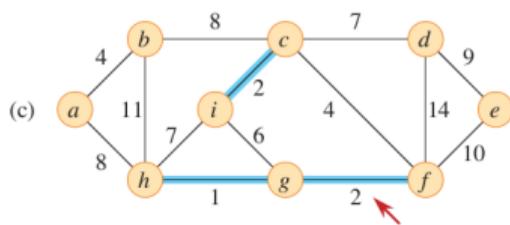
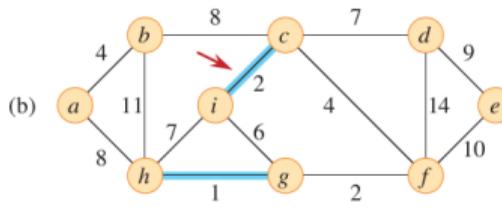
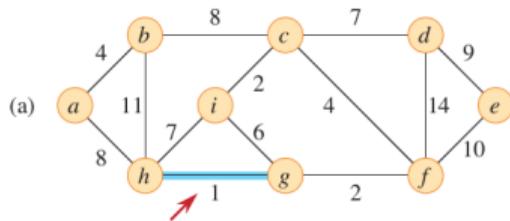
- زمان اجرای الگوریتم پریم را به صورت زیر تحلیل می‌کنیم:
- حلقة while به تعداد  $|V|$  بار تکرار می‌شود و از آنجایی که عملیات Extract-Min در زمان  $(|V| \lg |V|)$  O است، زمان لازم برای انجام این عملیات  $(|V| \lg |V|) O$  است.
- حلقة for در خطوط ۱۰ تا ۱۴ جمعاً  $O(|E|)$  بار تکرار می‌شود. هر فرآخوانی Decrease-Key در زمان  $(|V| \lg |V|)$  O اجرا می‌شود، بنابراین الگوریتم پریم در زمان  $(|V| \lg |V| + |E| \lg |V|) O$  اجرا می‌شود. چون تعداد یال‌های گرافی که دارای درخت پوشای کمینه است، حداقل برابر با  $|V|$  است، پس زمان اجرای الگوریتم پریم برابر با  $(|E| \lg |V|) O$  است.

# الگوریتم کروسکال

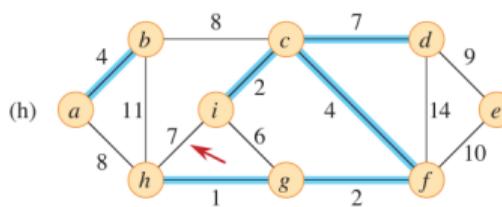
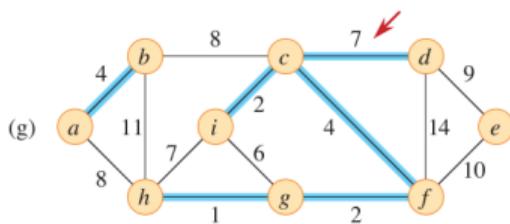
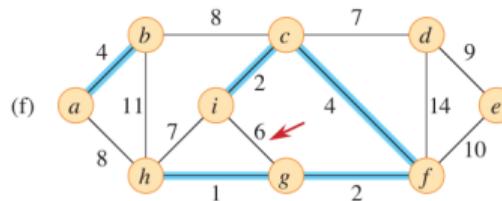
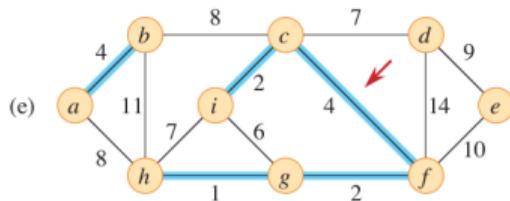
- الگوریتم کروسکال ابتدا به ازای هر رأس یک مجموعهٔ مجزا ایجاد می‌کند. سپس برای یافتن یال مطمئن در هر مرحله از بین همهٔ یال‌هایی که دو رأس در دو مجموعهٔ مجزا را به یکدیگر متصل می‌کنند، یال  $(u, v)$  با کمترین وزن را انتخاب می‌کند. وقتی یال  $(u, v)$  به عنوان یک یال از درخت پوشای کمینه انتخاب شد، مجموعه‌ای 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

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

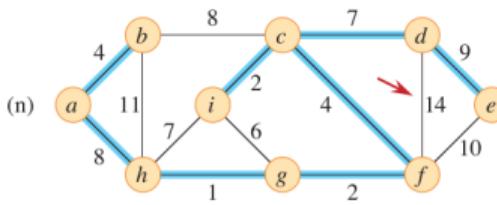
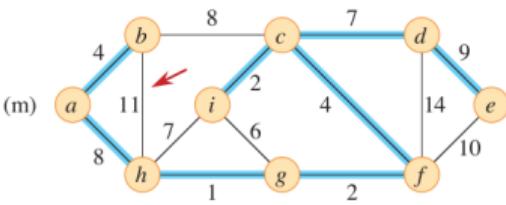
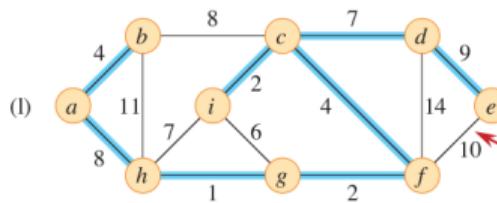
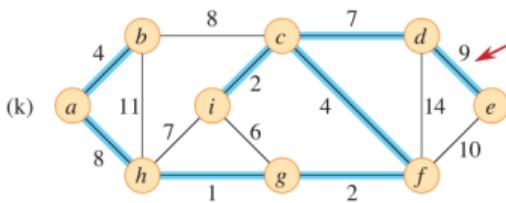
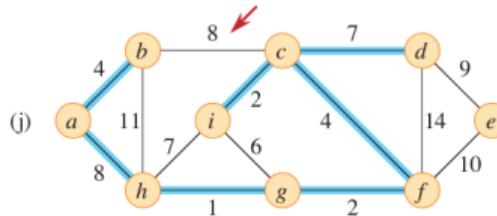
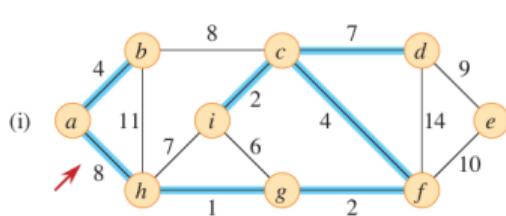


# الگوریتم کروسکال



-

# الگوریتم کروسکال



# 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
```

---

# الگوريتم کروسکال

- يافتن اجتماع از كتاب کلاینبرگ

# الگوریتم کروسکال

- در الگوریتم کروسکال، زمان لازم برای مرتب‌سازی یال‌ها  $O(|E|\lg|E|)$  است.
- زمان لازم برای بررسی همه یال‌ها  $O(|E|\lg|V|)$  است، اگر از داده ساختار مجموعه‌های مجزا<sup>1</sup> با نمایش درختی استفاده شود.
- 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 برابر است با  $O(|E|\lg|V|)$  .

---

<sup>1</sup> disjoint-set data structure

# کوتاهترین مسیر از یک مبدأ

- فرض کنید می خواهیم از شهری به شهر دیگر برویم و برای کاهش هزینه می خواهیم کوتاهترین مسیر را انتخاب کنیم. اطلاعات همه راهها و شهرها و فاصله بین شهرها را در اختیار داریم. چگونه می توانیم با این اطلاعات کوتاهترین مسیر را انتخاب کنیم؟
- یک راه ساده این است که همه 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 محاسبه جواب این مسئله به طور کارامد ارائه می کنیم.

## کوتاهترین مسیر از یک مبدأ

- ورودی مسئله کوتاهترین مسیر<sup>1</sup>، گراف جهت دار وزن دار  $G = (V, E)$  با تابع وزن  $w : E \rightarrow \mathbb{R}$  است که به ازای هر یال وزن آن را باز می گرداند. وزن مسیر  $p = \langle v_0, v_1, \dots, v_k \rangle$  که به صورت  $w(p)$  نشان داده می شود برابراست با مجموع وزن همه یال های 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 می کنیم.

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

---

<sup>1</sup> 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 سطح اول 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<sup>۱</sup> استفاده می‌کنیم.
- به ازای هر رأس  $v \in V$  الگوریتم کوتاهترین مسیر از یک رأس<sup>۲</sup> یک متغیر به نام  $v.d$  نگه‌دارد که یک کران بالا برای کوتاهترین مسیر از  $s$  به  $v$  است.
- مقدار  $v.d$  را تخمین کوتاهترین مسیر<sup>۳</sup> می‌نامیم.

---

<sup>1</sup> relaxation

<sup>2</sup> single-source shortest path

<sup>3</sup> 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 یا  $(u, v)$  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

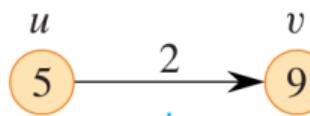
---

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

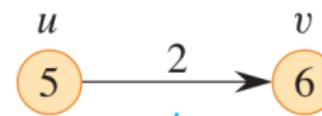
---

# 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.



RELAX( $u, v, w$ )



RELAX( $u, v, w$ )



# کوتاهترین مسیر از یک مبدأ

- دو الگوریتم مهم برای محاسبه کوتاهترین مسیر عبارتند از الگوریتم بلمن فورد و الگوریتم دایکسترا.
- در الگوریتم بلمن فورد هر یال ۱ - | $V$ | بار آزادسازی می‌شود، اما در الگوریتم دایکسترا هر یال فقط یک بار آزادسازی می‌شود.
- در الگوریتم بلمن فورد وزن یال‌ها می‌توانند منفی نیز باشد، اما الگوریتم دایکسترا تنها گراف‌هایی با وزن یال مثبت را می‌پذیرد. وزن یال منفی می‌تواند کاربردهای متنوعی داشته باشد. برای مثال، اگر یک خودروی برقی را در نظر بگیریم که در جاده‌هایی با شیب منفی شارژ می‌شود و در جاده‌هایی با شیب مثبت انرژی مصرف می‌کند، می‌توانیم شیب جاده را به عنوان وزن یال‌های گراف در نظر بگیریم.

# الگوریتم بلمن-فورد

- الگوریتم بلمن-فورد<sup>۱</sup> 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  $w : E \rightarrow \mathbb{R}$  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 همه رئوس را باز می‌گرداند.

---

<sup>۱</sup> Bellman-Ford algorithm

- الگوریتم بلمن فورد در زیر توصیف شده است.

---

## Algorithm Bellman-Ford

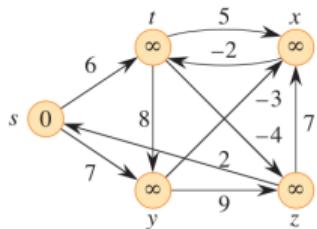
---

```
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
```

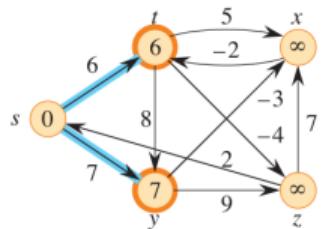
---

# الگوریتم بلمن-فورد

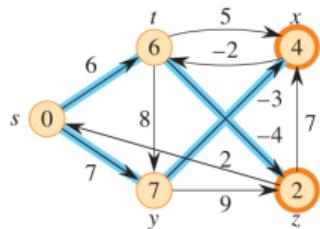
- یک مثال از اجرای الگوریتم بلمن فورد در زیر نشان داده شده است.



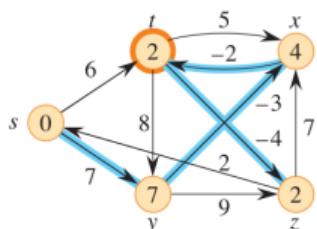
(a)



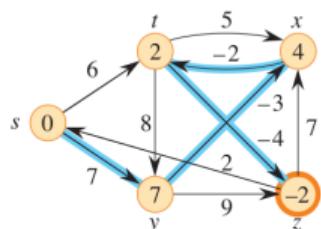
(b)



(c)



(d)



(e)

# الگوریتم بلمن-فورد

- مقداردهی اولیه در خط ۱ در زمان  $\Theta(|V|)$  اجرا می‌شود.
- بررسی همهٔ یال‌ها به زمان  $\Theta(|E|)$  نیاز خواهد داشت.
- در حلقه خطوط ۲ تا ۴ هر یک از  $|V|$  تکرارهای حلقه، در زمان  $\Theta(|E|)$  اجرا می‌شود.
- حلقه خطوط ۵ تا ۷ در زمان  $\Theta(|E|)$  اجرا می‌شود.
- بنابراین الگوریتم بلمن فورد در زمان  $O(|V||E|)$  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 همه رئوس از یک رأس مبدأ محاسبه می‌کند.
- قضیه: فرض کنید  $G = (V, E)$  یک گراف وزن دار جهت دار با رأس مبدأ  $s$  و تابع وزن  $w : E \rightarrow \mathbb{R}$  باشد و فرض کنید  $G$  هیچ دوری با وزن منفی نداشته باشد که از  $s$  قابل دسترسی باشد. آنگاه بعد از  $1 - |V|$  تکرار در حلقة خطوط ۲ تا ۴ الگوریتم بلمن فورد به دست می‌آوریم  $\delta(s, v) = v.d$  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  $\langle v_0, v_1, \dots, v_k \rangle$  به  $p = \langle v_0, v_1, \dots, v_k \rangle$  طوری که  $s = v_0$  و  $v_k = v$  و  $p$  کوتاهترین مسیر از  $s$  به  $v$  باشد.
- از آنجایی که کوتاهترین مسیر باید یک مسیر ساده باشد،  $p$  حداقل  $1 - |V|$  یال دارد و بنابراین  $. k \leqslant |V| - 1$ .
- هر یک از  $1 - |V|$  تکرار در حلقه خطوط ۲ تا ۴ همه  $|E|$  یال را آزادسازی می‌کند.

## الگوریتم بلمن-فورد

- بعد از یک بار تکرار حلقه، یال  $(s, v_1)$  آزاد سازی می شود و بنابراین  $v_1.d = w(s, v_1) = \delta(s, v_1)$  وزن  $v_1.d$  کوتاهترین مسیر از  $s$  به  $v_1$  خواهد بود.
- بعد از دو بار تکرار حلقه، یال  $(v_1, v_2)$  برای بار دوم آزادسازی می شود بنابراین  $v_2.d = v_1.d + w(v_1, v_2) = \delta(s, v_2)$  وزن کوتاهترین مسیر از  $s$  به  $v_2$  خواهد بود.
- بنابراین در تکرار  $i$  ام، به ازای  $k = 1, 2, \dots, i$  یال  $(v_{i-1}, v_i)$  آزادسازی می شود و  $v_i.d = v_{i-1}.d + w(v_{i-1}, v_i) = \delta(s, v_i)$  وزن کوتاهترین مسیر از  $s$  به  $v_i$  خواهد بود.
- پس از  $|V| - 1$  بار آزادسازی یالها به دست می آوریم:

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

# الگوریتم دایکسترا

- الگوریتم دایکسترا مسئله کوتاهترین مسیر برای گراف وزن دار جهت دار  $(V, E) = G$  را وقتی وزن ها منفی نباشند حل می کند. به عبارت دیگر به ازای هر یال  $E \in \{(u, v)\}$  در الگوریتم دایکسترا لازم است داشته باشیم  $w(u, v) \geq 0$ .
- با یک پیاده سازی بھینه، الگوریتم دایکسترا می تواند در زمان کمتری نسبت به الگوریتم بلمن- فورد مسئله را حل کند.

---

## Algorithm Dijkstra

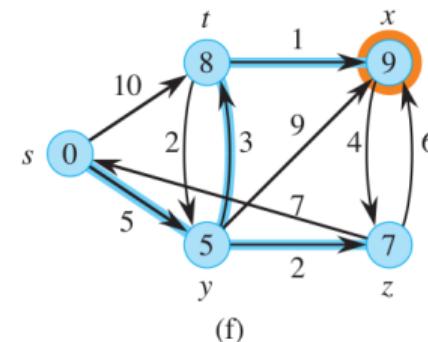
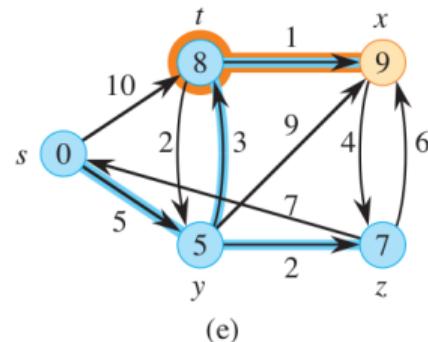
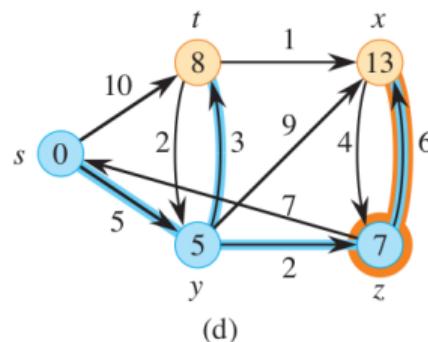
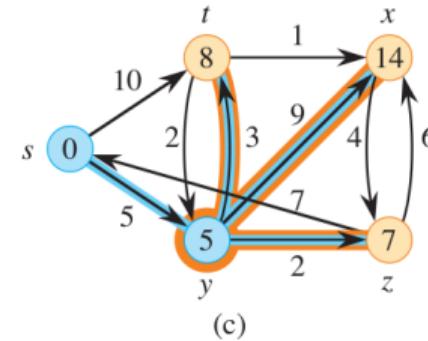
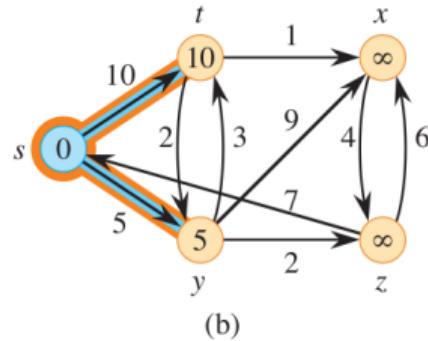
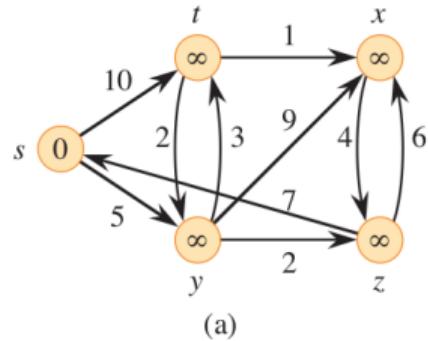
---

```
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)
```

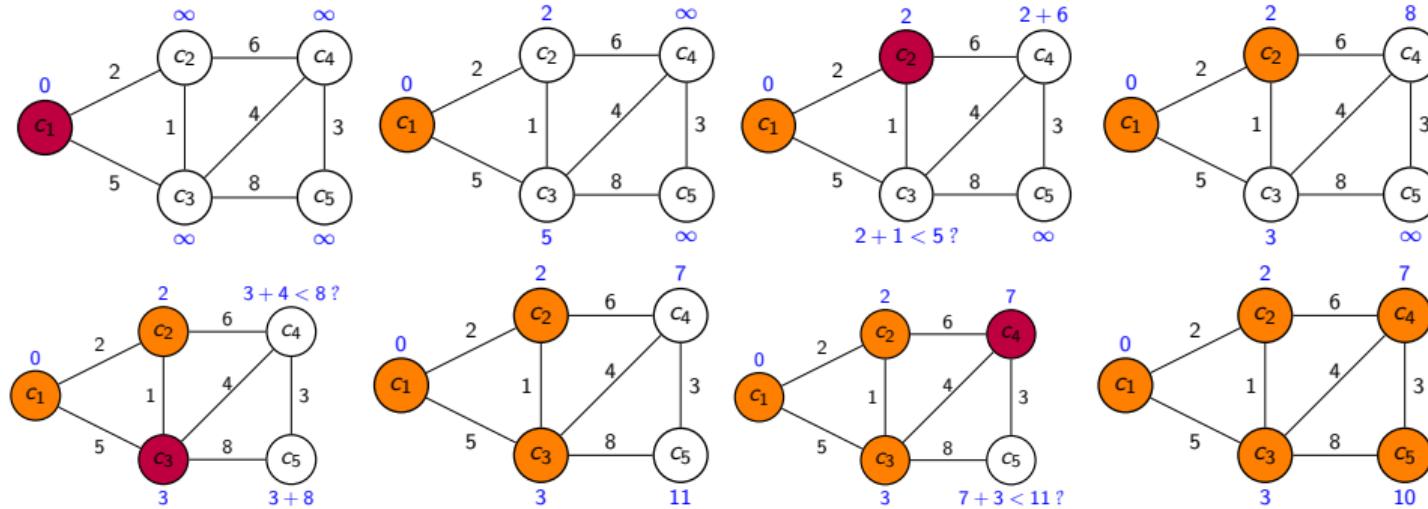
---

# الگوریتم دایکسترا

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



# الگوریتم دایکسترا



# الگوریتم دایکسترا

- مجموعه  $S$  شامل رئوسی است که کوتاهترین مسیر از مبدأ برای آنها تعیین شده است.
- از آنجایی که الگوریتم دایکسترا همیشه نزدیکترین رأس به مبدأ در  $S - V$  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  $V \in U$  به مجموعه  $S$  فقط یک بار اضافه می‌شود، هر یال در لیست مجاورت  $[u]$  در حلقه `for` خطوط ۹ تا ۱۲ دقیقاً یک بار در طول اجرای الگوریتم بررسی می‌شود. بنابراین این حلقه در مجموع به تعداد یال‌های گراف تکرار می‌شود و `Decrease-key` در مجموع حداقل به تعداد یال‌ها تکرار می‌شود. هزینه بررسی همه یال‌ها  $O(|E|)$  است.
- حلقه `while` نیز به تعداد رئوس گراف تکرار می‌شود.
- زمان اجرای الگوریتم دایکسترا به پیاده‌سازی صفت اولویت بستگی پیدا می‌کند.
- در یک پیاده‌سازی ساده صفت 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 Decrease-key و Extract-Min در زمان  $O(|V| \lg |V|)$  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.

# کوتاهترین مسیر بین همه جفت‌ها

- اکنون مسئله کوتاهترین مسیر بین همه جفت رأس‌ها در گراف را بررسی می‌کنیم.
- یکی از کاربرهای این الگوریتم، پیدا کردن کوتاهترین مسیر بین هر دو شهر در یک اطلس جغرافیایی است.
- یکی از کاربردهای دیگر این الگوریتم پیدا کردن فاصله بین دو نقطه در یک شبکه کامپیوتری برای ارسال بسته‌ها به طور بهینه است.
- خروجی الگوریتم یک جدول به اندازه  $|V| \times |V|$  است که در سطر ۱۱ و ستون ۷ فاصله بین شهر ۱۱ و شهر ۷ را باز می‌گرداند.

## کوتاهترین مسیر بین همه 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 همه 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 صفت اولویت در الگوریتم دایکسترا توسط هرم فیبوناچی پیاده‌سازی شود، زمان اجرا به  $(|V||E| + |V|^2 \lg |V|)$  کاهش پیدا می‌کند. البته در بدترین حالت  $O(|V|^2|E|)$ ، بنابراین زمان اجرای الگوریتم یافتن کوتاهترین مسیر بین همه جفت رئوس با استفاده از الگوریتم دایکسترا برابر است با  $O(|V|^3)$ .
- اگر گراف یال‌هایی با وزن منفی داشته باشد، نمی‌توان از الگوریتم دایکسترا استفاده کرد. می‌توانیم در این صورت با استفاده از الگوریتم بلمن-فورد، کوتاهترین مسیر بین همه جفت‌ها را در زمان  $O(|V|^2|E|)$  محاسبه کنیم که در صورتی که گراف متراکم باشد، زمان اجرا برابر خواهد بود با  $O(|V|^4)$ .

## کوتاهترین مسیر بین همه جفت‌ها

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

## کوتاهترین مسیر بین همه جفت‌ها

- اگر رأس‌ها را با شماره‌های  $1, 2, \dots, |V|$  شماره‌گذاری 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

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

- خروجی الگوریتم یک جدول  $n \times n$  خواهد بود به طوری‌که در سطر  $i$  و ستون  $j$  مقدار  $(j, i)$  قرار گیرد.
- خروجی الگوریتم کوتاهترین مسیر می‌تواند شامل یک ماتریس رئوس ماقبل<sup>1</sup> به نام  $(\pi_{ij}) = \Pi$  نیز باشد که در آن  $\pi_{iz}$  رأس پدر (ماقبل) رأس  $z$  را بر روی مسیری که از رأس  $i$  آغاز شده نشان می‌دهد. در صورتی که  $j = i$  باشد یا هیچ مسیری از  $i$  به  $j$  وجود نداشته باشد، آنگاه  $\pi_{iz} = \text{NIL}$  برابر با  $\text{NIL}$  خواهد بود.

---

<sup>1</sup> predecessor matrix

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

---

## Algorithm Print-All-Pairs-Shortest-Path

---

```
function PRINT-ALL-PAIRS-SHORTEST-PATH( $\Pi$ ,  $i$ ,  $j$ )
1: if  $i == j$  then
2:   print  $i$ 
3: else if  $\pi_{ij} == \text{NIL}$  then
4:   print "no path from"  $i$  "to"  $j$  "exists"
5: else
6:   Print-All-Pairs-Shortest-Path( $\Pi$ ,  $i$ ,  $\pi_{ij}$ )
7:   print  $j$ 
```

---

# الگوریتم فلوید-وارشال

- الگوریتم فلوید-وارشال<sup>۱</sup> مسئله کوتاهترین مسیر بین همه 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.

---

<sup>۱</sup> Floyd-Warshall algorithm

# الگوریتم فلوید-وارشال

- رئوس گراف  $G$  را با اعداد صحیح شماره گذاری می‌کنیم. بنابراین خواهیم داشت  $\{1, 2, \dots, n\}$ .  $V = \{1, 2, \dots, n\}$ .
- 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  $(j, i)$  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 همه جفت رئوس را با فرض اینکه رئوس میانی مسیر در مجموعه  $\{1, 2\}$  باشند محاسبه می‌کنیم.
- به همین ترتیب به ازای  $n \leq k \leq 1$  زیر مجموعه  $\{1, 2, \dots, k\}$  از رئوس را در نظر می‌گیریم و به ازای هر جفت از رئوس  $V \in j, i$ ، همه مسیرها از  $i$  به  $j$  را که از رئوس  $\{1, 2, \dots, k\}$  می‌گذرند را در نظر می‌گیریم و فرض می‌کنیم  $p$  کوتاهترین مسیر بین همه این مسیرها باشد.

# الگوریتم فلوید-وارشال

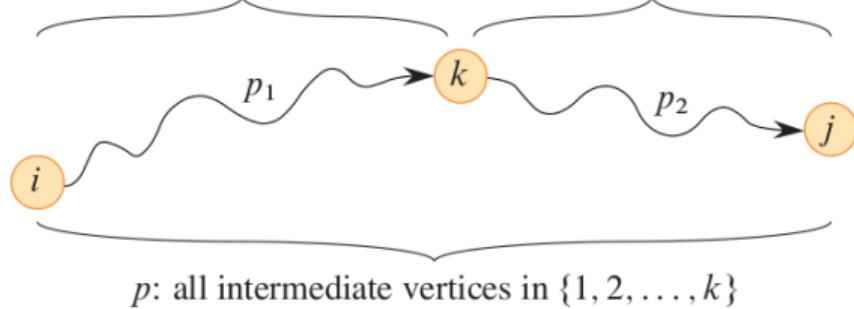
- حال حالت‌های زیر را در نظر می‌گیریم.

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

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

- شکل زیر این تقسیم بندی مسیر را نشان می‌دهد.

$p_1$ : all intermediate vertices in  $\{1, 2, \dots, k - 1\}$      $p_2$ : all intermediate vertices in  $\{1, 2, \dots, k - 1\}$



# الگوریتم فلوید-وارشال

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

## الگوریتم فلوید-وارشال

- وقتی  $k = 0$  است، یک مسیر از رأس  $i$  به رأس  $j$  که هیچ رأس میانی با شماره‌ای بزرگ‌تر از ۰ نداشته باشد، در واقع هیچ رأس میانی ندارد. چنین مسیری حداکثر یک یال دارد، بنابراین  $w_{ij} = d_{ij}^{(0)}$
- می‌توانیم مقدار  $d_{ij}^{(k)}$  را به صورت بازگشتی تعریف کنیم.

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

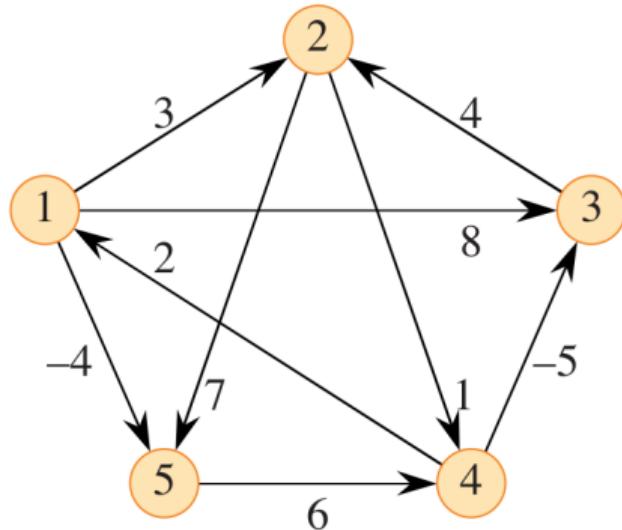
- چون برای هر مسیر، همه رئوس میانی متعلق به مجموعه  $\{1, 2, \dots, k\}$  هستند، بنابراین ماتریس  $D^n = (d_{ij}^{(n)})$  شامل جواب پایانی است. به ازای هر  $i, j \in V$  داریم  $d_{ij}^{(n)} = \delta(i, j)$ .

## 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.

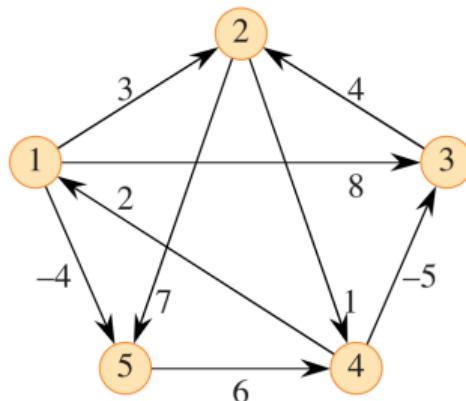
# الگوریتم فلوید-وارشال

- 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.



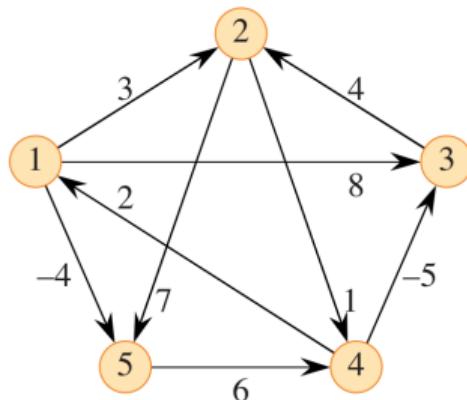
$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL}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

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

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

# الگوریتم فلوبید-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 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) = (dij(k)) be a new n × n matrix
4:   for i = 1 to n do
5:     for j = 1 to n do
6:       dij(k) = min{dij(k-1), dik(k-1) + dkj(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 رئوس ماقبل  $\Pi$  نیز می‌تواند در زمان محاسبه  $D^{(0)}, D^{(1)}, \dots, D^{(n)}$  محاسبه شود.
- به عبارت دیگر می‌توانیم  $\Pi^0, \Pi^1, \dots, \Pi^n$  را محاسبه کنیم به طوری که  $\Pi = \Pi^{(n)}$  و  $\pi_{ij}^{(k)}$  رأس ماقبل رأس  $j$  در کوتاهترین مسیری است که از رأس  $i$  شروع می‌شود و همه رئوس میانی در مجموعه  $\{1, 2, \dots, k\}$  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 :

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

## الگوریتم فلوید-وارشال

- به ازای  $1 \leq k$  ، 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  $\{1, 2, \dots, k\}$  از  $i$  آغاز شده است برابر است با رأس ماقبل  $j$  در مسیری که با رئوس میانی  $\{1, 2, \dots, k-1, \dots, 1\}$  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  $j \rightarrow k \rightarrow i$  و  $j \neq k$  ، آنگاه رأس ماقبل رأس  $j$  در این مسیر همان رأس ماقبل  $j$  در مسیری است که از  $k$  آغاز می شود و شامل همه رئوس در مجموعه  $\{1, 2, \dots, k-1, \dots, 1\}$  می شود.
- پس به ازای  $1 \leq k \leq n$  داریم :

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