

روش های پیمایش گراف

گراف، یک ساختمان داده غیرخطی است. پس برای ترتیب دادن به عناصر آن نیاز به یک قرارداد جهت ترتیب گره ها داریم. این قرارداد ها در قالب روش های جستجو گراف تعریف می شوند.

۱. پیمایش سطحی گراف

پیمایش سطحی گراف، یکی از راحت‌ترین الگوریتم‌ها برای پیمایش یک گراف است. بسیاری از الگوریتم‌های مهم دیگر، شبیه این الگوریتم عمل می‌کنند به عنوان مثال الگوریتم Prim برای مسئله‌ی درخت پوشای مینمم و الگوریتم دایکسترا برای مسئله‌ی Single source shortest paths

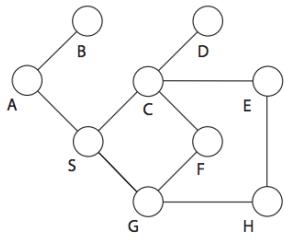
در این الگوریتم، به ازای هر گراف و یک راس دل خواه مانند S ، ابتدا تمام راس‌های قابل دسترس از S ، ملاقات می‌شوند.

اگر گراف ورودی بدون وزن باشد، این الگوریتم کوتاه‌ترین فاصله‌ها را از رأس مبدأ تا دیگر راس‌ها، محاسبه می‌کند. پیمایش سطحی گراف، یک درخت تولید می‌کند (درخت پیمایش سطحی) که شامل تمام راس‌های قابل دسترس از مبدأ است و به ازای هر راس v ، مسیر از S تا v در این درخت، برابر با کوتاه‌ترین مسیر از S تا v در گراف است. این الگوریتم هم در مورد گراف‌های جهت دار و هم غیر جهت دار به کار می‌رود.

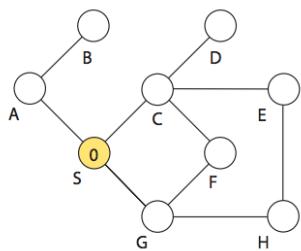
علت نام گذاری این الگوریتم به این دلیل است که در پیمایش سطحی، راس‌های ملاقات شده و ملاقات نشده به ترتیب عمق سطحی که در آن قرار دارند تعیین می‌شوند. این به آن معنی است که تمام رئوسی که در فاصله k از S قرار دارند قبل از رئوس با فاصله $k+1$ ملاقات می‌شوند.

مثال(۱)

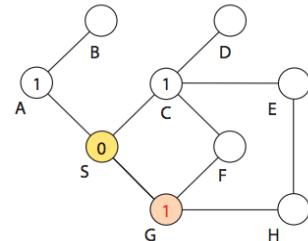
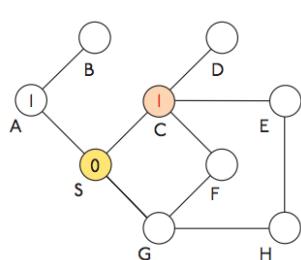
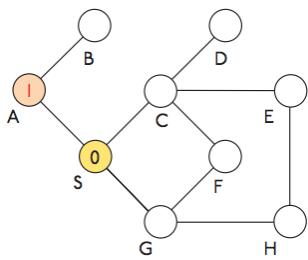
گراف زیر را با الگوریتم BFS پیمایش می‌کنیم.



۱. راس زرد را به عنوان مبداء(ریشه) در نظر می‌گیریم.

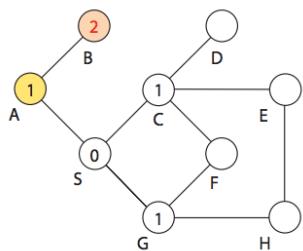


۲. راس های همسایه S را ملاقات می کنیم(با رنگ قرمز(با رنگ قرمز در تصویر مشخص شده است):



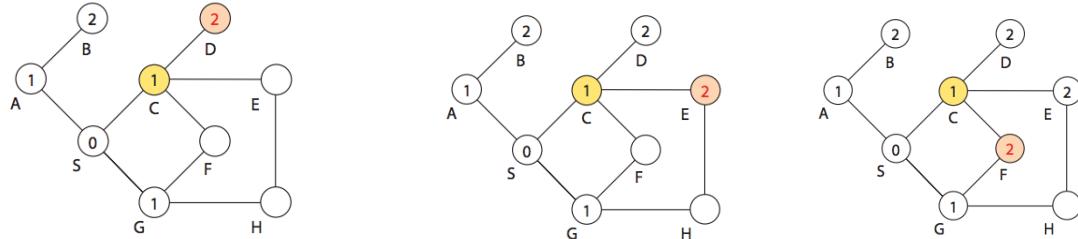
۳. اولین راس ملاقات شده در همسایه های S، راس A است. پس به راس A رفته و رئوس مجاور با آن را

مقالات می کنیم:



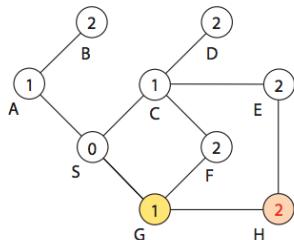
۴. برای دیگر همسایه های S نیز همین کار را انجام می دهیم (C و G):

C:



G:

راس F قبل در همسایه های C ملاقات شده است. پس دیگر ملاقات نمی شود و پدر آن در BFS-tree همان راس C است.



۵. به همین ترتیب راس های مجاور همسایه های رئوس ملاقات شده را، ملاقات می کنیم. در این پیمایش، توجه به دو نکته ضروری می باشد:

* شرط ملاقات هر راس این است که راس مذکور قبل ملاقات نشده باشد (هر راس دقیقاً یکبار دیده می شود)

** هر راسی که زودتر ملاقات شده باشد، در سطح بعدی همسایه های آن زودتر ملاقات می شوند.

۲. رنگ آمیزی رئوس:

برای در نظر گرفتن وضعیت گراف در مراحل مختلف اجرای الگوریتم، هر راس با یکی از رنگ های

سفید ، خاکستری یا سیاه رنگ می شود. تمام راس ها ابتدا سفید هستند و ممکن است در آینده خاکستری شده و سپس سیاه شوند. یک راس زمانی کشف می شود که برای اولین بار در طول پیمایش، به رنگی غیر سفید در آید .پس همه راس های خاکستری و سیاه، کشف شده (discovered) هستند. اما میان آن دو تمایز قائل می شویم تا مطمئن باشیم پیمایش براساس جستجو در سطح انجام می شود. تمام راس های مجاور با یک راس سیاه، غیر سفید هستند زیرا یک رأس زمانی سیاه می شود که تمام رأس های مجاورش کشف شده باشند .اما رأس های خاکستری ممکن است همسایگان سفید نیز داشته باشند.

۳. درخت جستجو سطحی:

جستجو سطحی، یک درخت جستجو سطحی می سازد. در ابتدا این درخت تنها با راس ریشه(S) مقدار دهی می شود که راس مبداء است. هرگاه در پیمایش گراف، هنگام ملاقات رئوس همسایه یک راس discover شده مانند u به یک راس سفید مانند v رسیدیم، راس v و یال(u, v) به درخت افزوده می شوند.

به راس u ، پدر(parent) راس v در درخت BFS میگوییم. هر راس حداکثر یک پدر دارد و در طول پیمایش، حداکثر یک بار کشف می شود. در ضمن اگر در درخت تولید شده، u در طول مسیری که s را به v وصل می کند قرار داشته باشد ، می گوییم u جد v است و v از نوادگان u است.

شبه کد زیر ، الگوریتم BFS را نشان می دهد .فرض بر این است که گراف ورودی ، به صورت لیست مجاورت ذخیره شده است .رنگ هر راس u را در $u.color$ و پدرش را در $u.parent$ ذخیره می کنیم. اگر راس u همان راس s بود و یا هنوز کشف نشده بود، پدر آن را NULL قرار می دهیم.

۴. کد پیمایش سطحی:

شبه کد زیر ، الگوریتم BFS را نشان می دهد .فرض بر این است که گراف ورودی ، به صورت لیست مجاورت ذخیره شده است .رنگ هر راس u را در $u.color$ و پدرش را در $u.parent$ ذخیره می کنیم. اگر راس u همان راس s بود و یا هنوز کشف نشده بود، پدر آن را NULL قرار می دهیم.

$U.d$ حاوی فاصله محاسبه شده در الگوریتم برای راس u تا ریشه(S) است. در این الگوریتم از یک ساختمان داده S صف ساده (FIFO Queue) برای ترتیب گره های خاکستری استفاده شده است.

```
BFS( $G, s$ )
1 for each vertex  $u \in G.V - \{s\}$ 
2    $u.color = \text{WHITE}$ 
3    $u.d = \infty$ 
4    $u.\pi = \text{NIL}$ 
5    $s.color = \text{GRAY}$ 
6    $s.d = 0$ 
7    $s.\pi = \text{NIL}$ 
8    $Q = \emptyset$ 
9   ENQUEUE( $Q, s$ )
10  while  $Q \neq \emptyset$ 
11     $u = \text{DEQUEUE}(Q)$ 
12    for each  $v \in G.Adj[u]$ 
13      if  $v.color == \text{WHITE}$ 
14         $v.color = \text{GRAY}$ 
15         $v.d = u.d + 1$ 
16         $v.\pi = u$ 
17        ENQUEUE( $Q, v$ )
18     $u.color = \text{BLACK}$ 
```

مطابق این کد، در خطوط ۱-۴ تمام رئوس به جز ریشه(S) سفید رنگ شده و پدر هر راس NULL میشود(زیرا هنوز هیچ راسی ملاقات نشده). فاصله رئوس از S را نیز ($U.d$) برابر بینهایت قرار می دهیم.

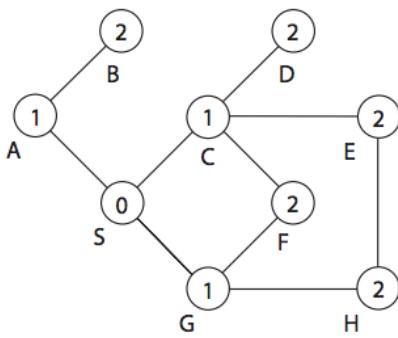
ریشه را ملاقات کرده ، رنگش را خاکستری و پدرش را NULL قرار می دهیم. $S.d$ برابر با صفر است. S را به صف اولویت(Q) اضافه می کنیم.

تا زمانی که Q خالی نشده(یا به عبارتی هنوز راس خاکستری داریم) :

عنصر سر صف را خارج میکنیم(u). در حلقه i **for** ، هر راس سفیدی که در همسایگی u قرار دارد ، با اجرای خط های ۱۴ تا ۱۷ کشف می شود و در آخر صف قرار می گیرد . زمانی که تمام راس های واقع در لیست مجاورت u بررسی شد ، u در خط ۱۸ سیاه می شود.

نتیجه هی اجرای BFS ، به ترتیب واقع شدن همسایه های یک راس در لیست مجاورت آن راس وابسته است . بنابراین برای یک گراف ممکن است بیش از یک BFS-tree ، داشته باشیم . اما در تمام این درخت ها ، فاصله ها (d) با یکدیگر برابرند .

جدول زیر، مراحل کد را برای مثال ۱ نشان می دهد:



v	w	Action	Queue
—	—	Start	{S}
S	A	set $d(A) = 1$	{A}
S	C	set $d(C) = 1$	{A, C}
S	G	set $d(G) = 1$	{A, C, G}
A	B	set $d(B) = 2$	{C, G, B}
A	S	none, as $d(S) = 0$	{C, G, B}
C	D	set $d(D) = 2$	{G, B, D}
C	E	set $d(E) = 2$	{G, B, D, E}
C	F	set $d(F) = 2$	{G, B, D, E, F}
C	S	none	{G, B, D, E, F}
G	F	none	{B, D, E, F}
G	H	set $d(H) = 2$	{B, D, E, F, H}
G	S	none	{B, D, E, F, H}
B	A	none	{D, E, F, H}
D	C	none	{E, F, H}
E	C	none	{F, H}
E	H	none	{F, H}
F	C	none	{H}
F	G	none	{H}
H	E	none	{}
H	G	none	{}

آنالیز زمان اجرای الگوریتم:

قبل از اثبات ویژگی های مختلف BFS ، به بررسی زمان اجرای آن روی گرافی چون (V, E) ، می پردازیم برای این کار از aggregate analysis استفاده شده است.

بعد از initialization ، هیچ راسی دوباره سفید نمی شود .پس با توجه به تست موجود در خط 13 ، هر راس حد اکثر یک بار وارد صف می شود .بنابراین هر راس حد اکثر یک بار از صف خارج می شود .پس زمان اختصاص داده شده به عملیات صف ، $O(V)$ خواهد بود .

به دلیل اینکه در این الگوریتم ، لیست مجاورت هر راس تنها زمانی بررسی می شود که آن راس از صف خارج شده باشد ، لیست مجاورت هر راس حد اکثر یک بار بررسی می شود و از آنجایی که مجموع طول های لیست های مجاورت راس ها $O(E)$ است ، زمان اختصاص یافته به بررسی لیست های مجاورت نیز $O(E)$ است .زمان لازم برای initialization هم $O(V)$ است .پس زمان اجرای الگوریتم $O(V+E)$ است.

۵. یافتن کوتاهترین مسیر:

همان طور که گفته شد ، به ازای هر راس v ، کوتاه ترین فاصله از S تا v را حساب کرده و در $v.d$ ذخیره می کند. طول کوتاه ترین مسیر از S تا v با $d(v,S)$ نشان داده می شود. اگر هیچ مسیری از S به v وجود نداشته باشد ، $d(v,S)$ برابر با بی نهایت در نظر گرفته می شود. پیش از آنکه اثبات کنیم الگوریتم طول کوتاهترین مسیر را محاسبه میکند، یکی از ویژگی های مهم را درباره کوتاه ترین مسیرها بیان می کنیم.

لم ۱:

فرض کنید $G(V,E)$ ، یک گراف جهت دار یا غیر جهت دار باشد و S نیز یک راس دل خواه باشد. آنگاه به ازای هر یال (v,u) خواهیم داشت:

$$1 + d(s,u) \Rightarrow d(s,v)$$

اثبات: اگر u از s قابل دسترسی باشد v نیز هست. در این صورت کوتاه ترین مسیر از s به v نمی تواند بلند تر از کوتاه ترین مسیر از s به u به علاوه ی v باشد (v,u) .

اگر u از s غیر قابل دسترسی باشد نیز نامساوی برقرار است. $(\infty = d(s,u))$.

برای اثبات این که پس از اجرای BFS روی یک گراف ، به ازای هر راس v ، $d(v,s)$ برابر با $d(v,s)$ خوهد بود ، ابتدا نشان می دهیم که در طول اجرا همواره $d(v,s) \leq d(v,u)$ است.

لم ۲:

فرض کنید $G(V,E)$ ، یک گراف جهت دار یا غیر جهت دار باشد و از راس دل خواهی مانند s روی گراف ، BFS اجرا شده است. آنگاه پس از اتمام اجرای الگوریتم به ازای هر v خواهیم داشت $d(v,s) \leq d(v,u)$.

اثبات: میتوان از استقرا روی تعداد عملیات Enqueue (وارد کردن به صف) استفاده کرد. پایه ای استقرا مربوط به زمانی است که s را در خط ۹ وارد Q کرده ایم. فرض استقرا در این مورد درست است زیرا $d(s,s) = 0$ و به ازای هر v ، $d(v,s) \leq d(v,u)$ بیشتر است.

اکنون راس سفیدی چون v را در نظر بگیرید که در زمان بررسی لیست مجاورت u ، ملاقات شده است. با توجه به فرض استقرا داریم:

با توجه به خط ۱۵ کد و با استفاده از لم ۱ میتوان نوشت:

$$v.d = u.d + 1$$

$$>= d(s, u) + 1$$

$$>= d(s, v)$$

سپس v وارد صف میشود و هیچگاه دوباره وارد صف نمیشود. زیرا خاکستری شده است. پس مقدار $v.d$ دیگر تغییر نمی‌کند و استقرا اثبات می‌شود.

برای اثبات $v.d = d(s, v)$ ، باید دقیقاً نشان دهیم صف Q در حین BFS چگونه عمل میکند. لم بعدی نشان میدهد همواره Q حداقل دو مقدار d را در خود نگه میدارد.

لم ۳:

فرض کنید در طول اجرای BFS بر روی $G(V, E)$ ، صف Q شامل راس‌های v_1, v_2, \dots, v_r باشد به طوری که v_1 سر Q و v_r انتهای صف است. برای $i = 1, 2, \dots, r-1$ خواهیم داشت:

$$v_r.d \leq v_i.d + 1, \quad v_i.d \leq v_{i+1}.d$$

اثبات: با استقرا روی تعداد عملیات صف (`enqueue` و `dequeue`) اثبات می‌شود. در ابتدا که Q تنها شامل s است، لم برقرار است.

برای استدلال استقرایی باید ثابت کنیم بعد از ورود و خروج یک راس به صف، لم همچنان برقرار است. اگر سر صف یعنی v_1 از صف خارج شود، v_2 سر صف جدید می‌شود. (اگر صف خالی شود، لم به وضوح برقرار است). از روی فرض استقرا داریم $v_r.d \leq v_1.d + 1$ و ضمناً $v_r.d \leq v_2.d$. پس $v_1.d \leq v_2.d + 1$. پس تمام ویژگی‌های ذکر شده پس از خروج سر صف، برای v_2 به عنوان سرصف جدید برقرار است. اکنون ثابت می‌کنیم این ویژگی‌ها پس از ورود یک راس جدید به آخر صف نیز برقرار خواهد بود.

برای بررسی اینکه پس از ورود یک راس جدید به صف، کد BFS را دقیق‌تر بررسی میکنیم. زمانی که راسی چون v را وارد صف می‌کنیم (خط ۱۷)، تبدیل به v_{r+1} می‌شود. در این زمان راس u از صف خارج شده و

اکنون در لیست مجاورت U هستیم. (راسی که پس از حذف U سرف جدید شده است را v_1 می نامیم . از روی فرض استقرا . $v_1.d \geq u.d$ بنا بر این :

$$v_{r+1}.d = v.d = u.d + 1 \leq v_1.d + 1$$

از روی فرض استقرا همچنین داریم $v_r.d \leq u.d + 1 = v.d = v_{r+1}.d \leq u.d + 1$ پس $v_r.d \leq u.d + 1$ و بقیه نامساوی ها نیز بدون تغییر خواهند بود . بنابراین زمانی که راسی جدید وارد صفحه میشود نیز لم برقرار است.

نتیجه زیر نشان می دهد مقدار d در زمان ورود رئوس به صفحه به طور یکنواخت افزایش می یابد.

نتیجه ۴:

فرض کنید که v_i و v_j ، در طول اجرای الگوریتم وارد صفحه شده اند . در ضمن v_j قبل از v_i وارد شده است . آنگاه در زمانی که v_j وارد شده ، $v_i.d \leq v_j.d$ است .

اثبات: به راحتی از لم ۳ و این ویژگی که هر راس حداقل یک بار مقدار متناهی d می گیرد ، ثابت می شود .

اکنون می توانیم ثابت کنیم که الگوریتم BFS به درستی کوتاهترین مسیر را پیدا می کند .

BFS - درستی

فرض کنید (V,E) یک گراف جهت دار یا غیر جهت دار باشد که BFS روی یک راس دل خواه از آن مانند S اجرا شده است . آنگاه در طول اجرای BFS ، تمام راس های قابل دسترس از S ، کشف می شوند و پس از اتمام اجرا ، به ازای هر v ، $v.d = d(s,v)$ است . ضمناً به ازای هر v که از S قابل دسترسی است ، یکی از کوتاه ترین مسیر ها از S به v ، کوتاهترین مسیر از S به پدر v ($V.\Pi, v$) به همراه یال ($V.\Pi, v$) است .

اثبات:

فرض کنید راس هایی وجود دارند که در آن ها مقدار d برابر با طول کوتاه ترین مسیر از S نباشد . فرض کنید v یکی از این راس ها باشد که در آن $d(s,v)$ از بقیه ای راس ها کمتر است . به ضوح v نمی تواند S باشد . از لم ۲ نتیجه می گیریم که $v.d \geq d(s,v)$ است . در ضمن v باید قابل دسترسی از S باشد . زیرا در غیر این صورت $d(s,v)$ برابر با بینهایت است و بزرگتر از $v.d$ میشود .

فرض کنید که u راسی باشد که در یکی از کوتاه ترین مسیرها از s به v ، پدر v است. پس $d(s,v) = d(s,u) + 1$. با در کنار هم قرار دادن تمام این اطلاعات داریم:

$$v.d > d(s,v) = d(s,u) + 1 = u.d + 1 \quad (1)$$

اکنون زمانی را در نظر بگیرید که **BFS** انتخاب می‌کند که u را از Q خارج کند (خط 11). در این لحظه v میتواند سفید، خاکستری یا سیاه باشد. نشان خواهیم داد که در هر صورت به تنافض خواهیم رسید. اگر v سفید باشد آنگاه با استفاده از نتیجه $v.d \leq u.d$ است که با (1) در تنافض است. اگر v خاکستری باشد، آنگاه v زمانی خاکستری شده که راسی چون w از صفحه خارج می‌شده است. و w زودتر از u از صفحه خارج شده و $v.d = w.d + 1$ است. با نتیجه $w.d \leq u.d$ داریم. بنابراین $v.d = d(s,v) = u.d + 1$ که با (1) در تنافض است. پس نتیجه می‌گیریم که به ازای هر v تمام راس‌هایی چون v که از s قابل دسترس‌اند، در پایان الگوریتم کشف می‌شوند. زیرا در غیر این صورت $v.d$ بی‌نهایت بوده و از $d(s,v)$ بیشتر می‌شود.

توجه کنید که اگر $u = v$ ، آنگاه $v.d = u.d + 1$. پس میتوانیم یک کوتاه‌ترین مسیر از s تا v با استفاده از کوتاه‌ترین مسیر از s تا v به همراه یال (v, u) داشته باشیم.

:BFS درخت ۷

برای گراف $G(V, E)$ و راس مبدأ s ، زیر گراف G_π ، اینگونه تعریف می‌شود.

$$G_\pi = (V_\pi, E_\pi) \text{ where}$$

$$V_\pi = \{v \in V \mid v.\pi \neq \text{NULL}\} + \{s\}$$

$$E_\pi = \{(v.\pi, v) \mid v \in V_\pi - \{s\}\}$$

یک G_π است اگر V_π شامل تمام راس‌های قابل دسترس از s باشد و به ازای هر $v \in V_\pi$ یک مسیر یکتا از s به v باشد به طوری که این مسیر یک کوتاه‌ترین مسیر از s به v در G باشد. G_π شامل یک درخت است زیرا همبند است و $|E_\pi| = |V_\pi| - 1$. یال‌های tree edges را E_π مینامیم.

لم زیر نشان می‌دهد که در طول اجرای الگوریتم به دست آمده، یک **BFS-tree** است.

زمانی که BFS روی یک گراف جهت دار یا غیر جهت دار چون (V, E) اجرا می شود ، $G_{\pi} (V_{\pi}, E_{\pi})$ یک BFS-tree است.

اثبات: خط ۱۶ در BFS را برابر U قرار میدهد اگر و تنها اگر (u, v) یکی از یال های G باشد و $d(s, v) = d(s, u) + 1$. یعنی v از s قابل دسترسی باشد .بنا بر این V_{π} شامل تمام راس هایی است که از s قابل دسترسی اند .چون G_{π} یک درخت است ، به ازای هر $v \in V_{\pi}$ ، یک مسیر ساده ای یکتا از s به تمام رئوس V_{π} وجود دارد .با به کار گیری قضیه ۵ به طور متوالی ، نتیجه می شود که هر کدام از این مسیر ها در G_{π} یک کوتاهترین مسیر در G هستند.

الگوریتم زیر، رئوس در کوتاهترین مسیر از s به v را چاپ میکند. با فرض اینکه با استفاده از BFS ، یک bfs-tree تشکیل شده است.

```

PRINT-PATH( $G, s, v$ )
1 if  $v == s$ 
2   print  $s$ 
3 elseif  $v.\pi == \text{NIL}$ 
4   print "no path from"  $s$  "to"  $v$  "exists"
5 else PRINT-PATH( $G, s, v.\pi$ )
6   print  $v$ 
```

زمان اجرای این الگوریتم نسبت به V خطی میباشد.

۲. پیمایش عمقی گراف

در پیمایش عمقی یا DFS، جستجو در عمیق ترین سطح ممکن انجام می‌شود.

در DFS یال خارج شونده از آخرین راسی که کشف شده و دارای راس‌های مجاوری است که هنوز کشف نشده‌اند، پیمایش می‌شود. این روند به طور بازگشتی تا جایی ادامه می‌ابد که به راسی برسیم که دیگر همسایه‌ی کشف نشده‌ای نداشته باشد. آنگاه back-track کرده و همین روند را ادامه میدهیم تا جایی که هیچ راس ملاقات نشده‌ای وجود نداشته باشد.

مانند BFS، هرگاه در DFS راس v هنگام جستجو در لیست مجاورت راس u ، ملاقات شود:

پدر راس v را $v.\pi$ برابر u قرار می‌دهیم.

برخلاف BFS که حاصل پیمایش آن یک درخت است، در DFS ممکن است حاصل یک جنگل باشد. زیرا ممکن است الگوریتم از چندین مبداء تکرار شود.

در DFS کمی متفاوت از BFS تعریف می‌شود:

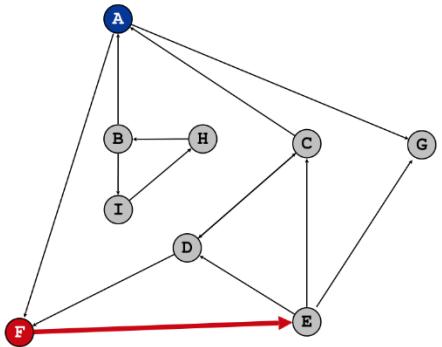
$$G_\pi = (V, E_\pi)$$

$$E_\pi = \{(v.\pi, v) \mid v \in V \text{ and } v.\pi \neq \text{NULL}\}$$

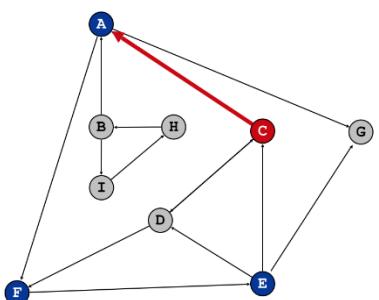
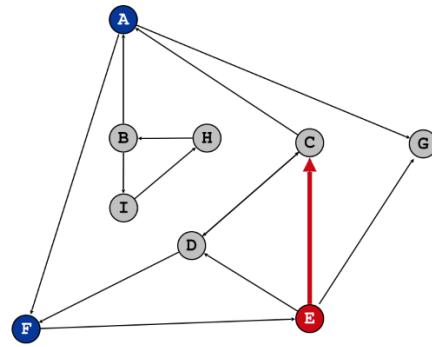
جنگل DFS از تعدادی درخت tree edges E_π تشکیل شده است یال‌های E_π هستند.

در گراف‌های جهت دار نیز به همین ترتیب تعریف می‌شود. به مثال زیر توجه کنید:

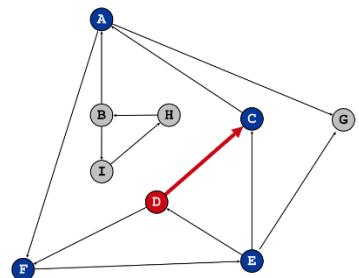
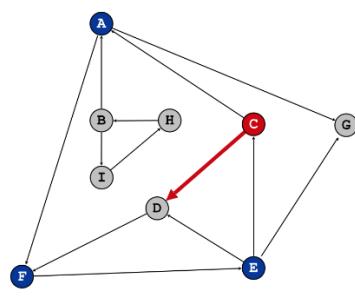




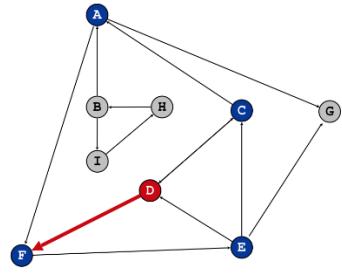
>>>

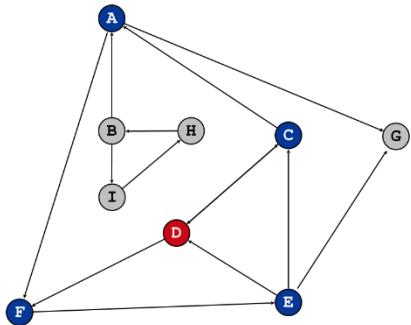


>>>

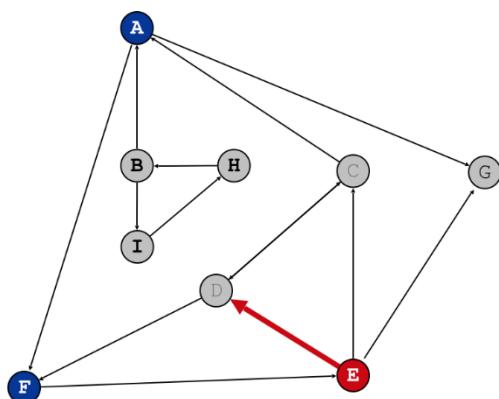
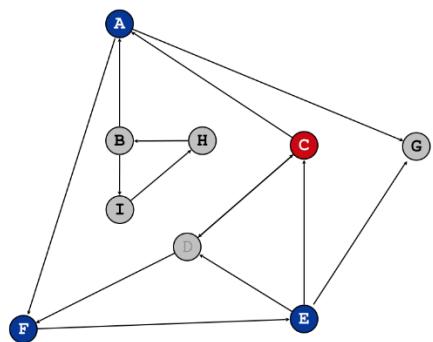


>>>

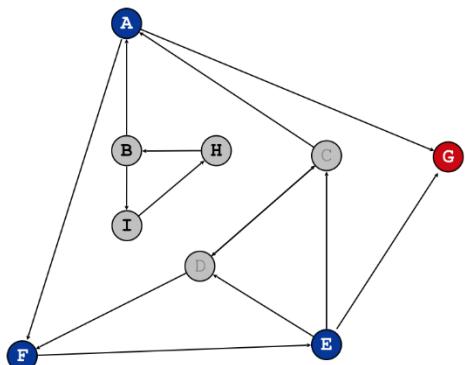
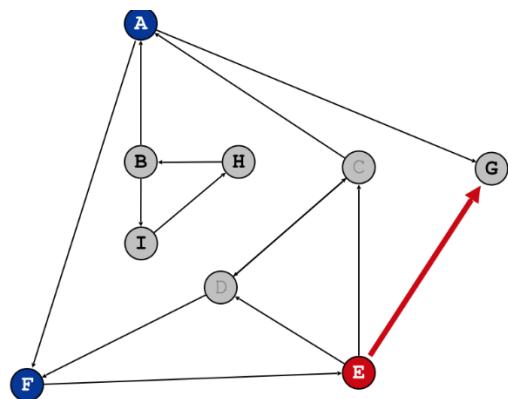




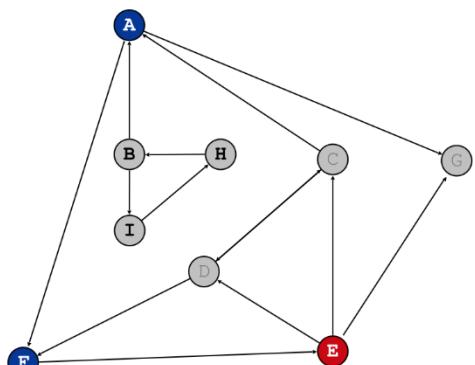
>>>

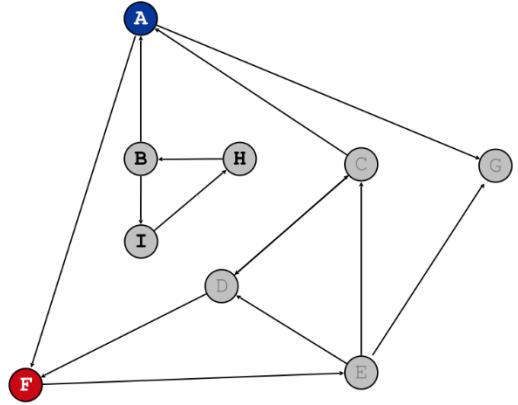


>>>

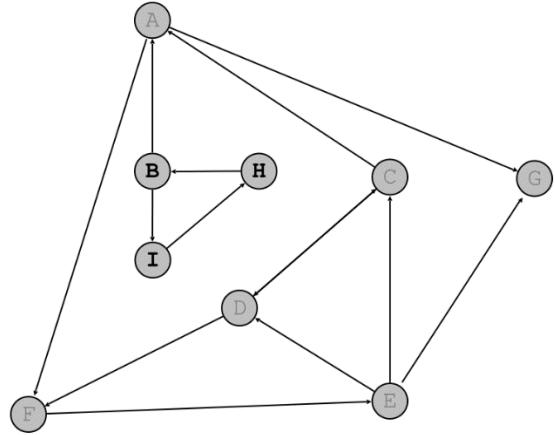
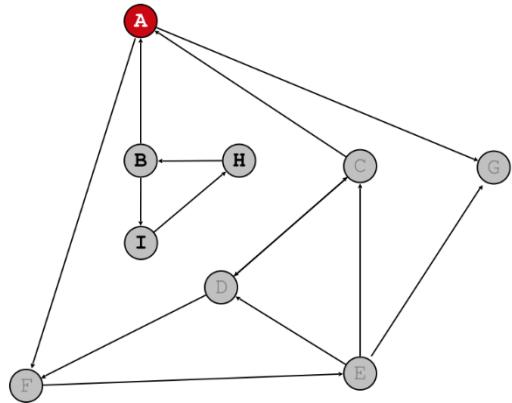
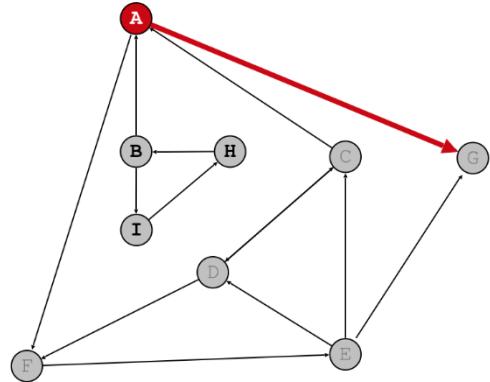


>>>





>>>



>>>

۱. رنگ آمیزی:

همانند DFS، در BFS نیز به هر گره یکی از رنگ های سفید، خاکستری و سیاه را نسبت می دهیم. تمام رئوس در ابتدا سفید هستند. هر راسی پس از ملاقات شدن، خاکستری می شود. هنگامی که تمام رئوس در قلمرو راس خاکستری v را ملاقات کردیم، قبل از خارج شدن از قلمرو v ، رنگ آن را سیاه می کنیم.

علاوه بر رنگ آمیزی، در DFS یک discovery time و یک finishing time برای هر راس v در نظر می گیریم. $v.d$ نشان دهنده می زمانیست که v کشف شده. $v.f$ نشان دهنده می زمانیست که v ، شده است. پس v قبل از $v.d$ سفید بوده و بین $v.d$ و $v.f$ خاکستری می شود و پس از $v.f$ سیاه خواهد بود.

همچنین داریم:

$$1 < v.d, v.f < 2|V|$$

$$v.d < v.f$$

۲. کد پیمایش عمقی:

شبیه کد زیر، نحوه عملکرد DFS را نشان می دهد. گراف G جهت دار یا غیر جهت دار است. متغیر time یک متغیر global است که برای timestamping استفاده شده است.

```

DFS( $G$ )
1 for each vertex  $u \in G.V$ 
2    $u.color = \text{WHITE}$ 
3    $u.\pi = \text{NIL}$ 
4    $time = 0$ 
5   for each vertex  $u \in G.V$ 
6     if  $u.color == \text{WHITE}$ 
7       DFS-VISIT( $G, u$ )

```

DFS-VISIT(G, u)

```

1    $time = time + 1$            // white vertex  $u$  has just been discovered
2    $u.d = time$ 
3    $u.color = \text{GRAY}$ 
4   for each  $v \in G.Adj[u]$     // explore edge  $(u, v)$ 
5     if  $v.color == \text{WHITE}$ 
6        $v.\pi = u$ 
7       DFS-VISIT( $G, v$ )
8    $u.color = \text{BLACK}$         // blacken  $u$ ; it is finished
9    $time = time + 1$ 
10   $u.f = time$ 

```

در خط ۳-۱ تمام رئوس سفید و پدرشان NULL میشود.

خط ۵-۷ به ترتیب تمام رئوس ۷ را چک میکند. هرگاه به راس سفیدی برسوردیم آن را با DFS-Visit ملاقات میکنیم. هربار DFS-Visit(G, u) فراخوانده میشود، راس u ریشه‌ی یک درخت جدید از جنگل DFS میشود.

در DFS-Visit(G, u)

رئوس موجود در لیست مجاورت u بررسی شده و اگر سفید بود به صورت بازگشتی ملاقات میشود.

حلقه‌هایی که در خطوط ۱ تا ۳ و ۵ تا ۷ از DFS هستند $O(V)$ زمان میبرند، البته بدون محاسبه زمان لازم برای DFS-Visit

روی هر راس، حداقل یکبار DFS-Visit فراخوانی میشود زیرا اگر DFS-Visit روی راسی فراخوانی شود، آن راس حتما سفید بوده و در اجرای DFS-Visit بلافاصله خاکستری شده و دیگر سفید نمیگردد. به ازای راسی چون ۷ حلقه‌ی موجود در خطوط ۴ تا ۷ از DFS-Visit، دقیقا به اندازه $|Adj[v]|$ بار تکرار میشود. پس هزینه‌ی اجرای این حلقه $O(E)$ است. بنابراین هزینه‌ی اجرای الگوریتم $O(V + E)$ است.

۳. ویژگی های DFS

یکی از ویژگی های DFS این است که اگر زمان کشف راسی چون v را با " v " و زمان اتمام آن را با " v " روی محور زمان علامت گذاری کنیم، یک عبارت خوش فرم پرانتری ایجاد می شود.

قضیه ۷ - قضیه‌ی پرانتری

در هر پیمایش DFS روی هر گراف جهتدار یا غیر جهتدار ، به ازای هر دو راس u و v ، دقیقاً یکی از حالت‌های زیر اتفاق می افتد.

- بازه های $[v.d, v.f]$ و $[u.d, u.f]$ جدا از هم اند و هیچکدام از دو راس ، از نوادگان دیگری نیست.

در $[v.d, v.f]$ قرار دارد و u از نوادگان v است.

در $[u.d, u.f]$ قرار دارد و v از نوادگان u است.

اثبات:

فرض کنید $u.d < v.d$ باشد . اکنون بر این اساس که آیا $v.d < u.f$ است یا نه دو حالت را در نظر میگیریم .

در حالت اول $v.d < u.f$ است . بنابراین v زمانی که u هنوز خاکستری بوده کشف شده پس v از نوادگان u است . به علاوه چون v بعد از u کشف شده است ، تمام يالهای خارج شونده از آن پیمایش شده و v قبل از u به پایان رسیده است . بنابراین $[v.d, v.f]$ در $[u.d, u.f]$ واقع است

اکنون فرض کنیم $u.f < v.d$ باشد . در این حالت بنابر نامساوی $u.d < u.f < v.d < v.f$ (2) پس دو بازه جدا از هم اند . هیچکدام از دو راس زمانی که دیگری خاکستری بوده کشف نشده پس هیچکدام از نوادگان دیگری نیست .

برای حالتی که $v.d < u.d$ نیز میتوان با استدلالی مشابه قضیه را ثابت کرد.

نتیجه 8

راس v یکی از نوادگان سره u است اگر و تنها اگر $v.f < u.f < v.d < u.d$ نتیجه میشود . اثبات: به راحتی از قضیه 7

قضیه 9 - قضیه ی مسیر سفید

در یک جنگل حاصل شده از اجرای DFS روی $G = (V, E)$ ، v یکی از نوادگان u است اگر و تنها اگر در زمان $u.d$ یک مسیر از u به v وجود داشته باشد که تماماً شامل راس های سفید باشد .

اثبات: اگر $v = u$ باشد، مسیر موجود از u به v تنها شامل v است که هنوز سفید است . اکنون فرض میکنیم v یکی از نوادگان سره u باشد . از نتیجه 8 در میابیم که $u.d < v.d$ است پس v در زمان $u.d$ سفید بوده است . چون v میتواند هر کدام از نوادگان u باشد، تمام راس ها در مسیری از u به v در زمان $u.d$ سفیدند .

فرض کنیم که در زمان $u.d$ یک مسیر از راس های سفید از u به v وجود دارد اما v در پایان از نوادگان u نمیشود . بدون از بین رفتن عمومیت (without loss of generality) ، فرض کنید که تمام راس های غیر از v در مسیر مذکور ، از نوادگان u باشند) . در غیر این صورت فرض کنید که v در مسیر ، نزدیکترین راس به u است که از نوادگان u نیست (فرض کنید w پدر v در این مسیر کاملاً سفید باشد . با استفاده از نتیجه 8 میدانیم $w.f < u.f$ است . از آنجایی که v باید بعد از اینکه u کشف شده و قبل از اینکه تمام شده باشد ، کشف شده باشد ، داریم $u.d < v.d < w.f < u.f$. قضیه ی 7 ایجاب میکند که بازه w

[درون بازه‌ی $[u.d, u.f]$ قرار گرفته باشد . با استفاده از نتیجه 8 در میابیم که 7 باید از نوادگان u باشد.

۴. دسته‌بندی یال‌ها:

میتوان 4 نوع یال را تعریف کرد:

Tree edges : یال‌ها بی هستند که در G_p وجود دارند.

Back edges : یال‌ها بی چون (u,v) هستند که u را به یکی از اجدادش وصل میکند . (طوقه self-loop) نیز یک back edge است.

Forward edges : یال‌هایی که اولا tree edge نباشند ثانیا راسی را به یکی از نوادگانش متصل کنند.

Cross edges : بقیه‌ی یال‌ها را تشکیل میدهند . این یال‌ها ممکن است بین دو درختDFS باشند و یا ممکن است دو راس موجود در یک درخت را به هم وصل کنند.

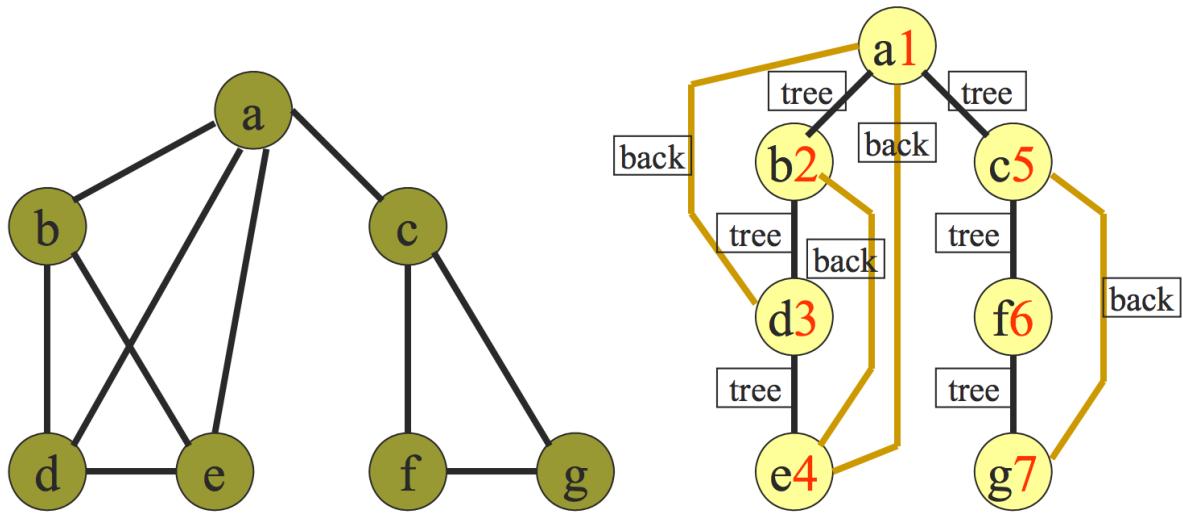
این الگوریتم اطلاعات لازم را برای شناسایی نوع یک یال در زمانی که آن یال را می‌پیماید ، دارد . به این صورت که وقتی برای اولین بار یال (u,v) پیموده میشود ، رنگ 7 اطلاعاتی در باره‌ی نوع یال به دست میدهد.

اگر سفید باشد (u,v) tree edge است.

اگر خاکستری باشد (u,v) back edge است.

اگر سیاه باشد در صورتی که forward edge باشد ، $u.d < v.d$ و در غیر این صورت cross edge است.

در مورد یک گراف غیر جهتدار ، این دسته‌بندی ممکن است مبهم باشد زیرا دو یال (u,v) و (v,u) در واقع یکی هستند . در این حالت نوع یال از دسته‌ای خواهد بود که برای اولین بار شناسایی شده .



قضیه 10

در یک گراف غیر جهتدار ، cross edge یا forward edge نخواهیم داشت.

اثبات : فرض کنید (u,v) یک یال دلخواه از G باشند . بدون از بین رفتن عمومیت ، فرض کنید $u.d < v.d$ است . بنا بر این راس v قبل از اینکه u تمام شود ، کشف و تمام شده است . اگر اولین بار یال از u به v پیموده شده باشد ، (u,v) tree edge خواهد بود . اگر در اولین بار از طرف v به u پیموده شده باشد ، یک back edge خواهد بود .