شش: الگوریتم‌های گراف

ساختمان داده ها و الگوریتم

مدرس: دکتر نجمه منصوری

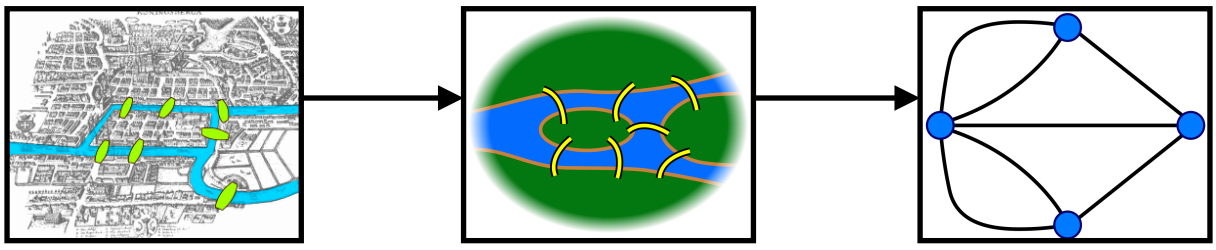
نگارنده: سجاد هاشمیان

۱. گراف چیست؟

گراف مدلی ریاضی برای یک مجموعه گسسته است که اعضایش به گونه‌ای با هم پیوند دارند. اعضای این مجموعه می‌توانند چند انسان باشند و ارتباط میان آن‌ها دست دادن با یکدیگر باشد؛ اعضا می‌توانند اتم‌ها در یک مولکول باشند و ارتباطشان پیوندهای شیمیایی باشد یا این که اعضا می‌توانند بخش‌های گوناگون یک زمین و ارتباط میانشان، پل‌هایی باشد که آن‌ها را به هم می‌پیوندند. نظریه گراف یکی از موضوع‌های مهم در ریاضیات گسسته است که به شناخت گراف‌ها و مدل‌بندی مسایل با آن‌ها می‌پردازد. لئونارد اویلر در سال ۱۷۳۶ با حل مسئله پل‌های کونیگسبرگ نظریهٔ گراف‌ها را بنیان گذاشت. اما جیمز جوزف سیلوستر نخستین کسی بود که در سال ۱۸۷۸ این مدل‌های ریاضی را گراف نامید.

مسئله پل‌های کونیگسبرگ

مسئله پل‌های کونیگسبرگ یکی از مشهورترین مسایل در نظریه گراف است که در مکان و شرایط واقعی طرح شده‌است. در اوایل سده ۱۸ ساکنین کونیگسبرگ در پروسیا (در حال حاضر کالینینگراد در روسیه) در روزهای یکشنبه به پیاده‌روی‌هایی طولانی در شهر می‌رفتند. رود پرگولیا شهر را به چهار قسمت تقسیم می‌کرد که با هفت پل به هم مرتبط بودند. ساکنان سعی می‌کردند مسیری بیابند که پیاده‌روی را از نقطه‌ای در شهر شروع کنند و از تمامی پل‌ها فقط یکبار بگذرند و دوباره به نقطه شروع بازگردند.



اویلر ابتدا نقشه شهر را با نقشه‌ای که فقط خشکی‌ها، رود و پل‌ها را نشان می‌داد، جایگزین کرد. سپس هر خشکی را با یک نقطه نشان داد که رأس نامیده می‌شود و هر پل را نیز با یک خط نشان داد که یال نامیده می‌شود. این ساختار ریاضی را گراف می‌نامند. اویلر ثابت کرد برای آنکه مسیری وجود داشته باشد که از یک رأس شروع شود و از تمامی یال‌ها یکبار بگذرد و به همان رأس بازگردد، باید گراف همبند بوده و هر یک از رأس‌های آن نیز از درجه زوج باشد. چنین مسیری، **دور اویلری** و چنین گرافی، **گراف اویلری** نامیده می‌شود.

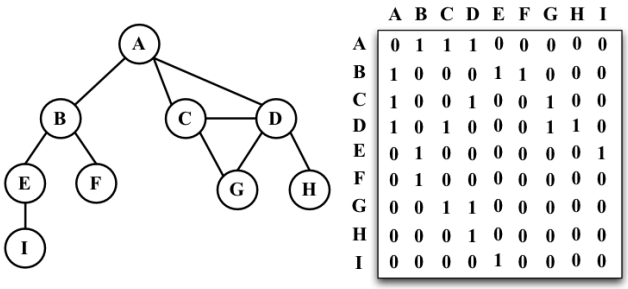
۲. نمایش گراف‌ها

نمایش گراف در واقع همان روش‌های ذخیره‌سازی گراف در کامپیوتر است، به عبارت دیگر با توجه به محدودیت‌هایی که در کامپیوتر داریم، نمی‌توانیم شکل گراف را همان‌طور که در کاغذ می‌کشیم نشان دهیم، یا با گفتن ویژگی‌های تصویری آن را به راحتی ذخیره و شبیه سازی کنیم. در این قسمت سعی می کنیم گراف را درون کامپیوتر ورودی بگیریم و ذخیره کنیم.

اولین کاری که برای ذخیره سازی گراف نیاز داریم، شماره‌گذاری رئوس است. یعنی به هر رأسی یک شماره نسبت دهیم تا بتوانیم بین آن‌ها تمایز قائل شویم. از این‌رو یک گراف، می‌تواند نمایش هایی متفاوت داشته باشد.(دقیقا چندتا؟)

۲.۱ ماتریس مجاورت

در واقع یک جدول دوبعدی از درایه‌ها است که طول سطر و ستون آن برابر تعداد راس‌های گراف است. ابتدا راس‌ها را شماره گذاری می‌کنیم. حال در درایه‌ی سطر iام و ستون jام آن اگر از راس شماره i به j یال نبود، صفر می‌گذاریم؛ اگر یال بود وزن آن را و اگر گراف وزن‌دار نبود، 1 می‌گذاریم. همچنین اگر گراف بدون‌جهت باشد، این را برای قرینه آن هم انجام می‌دهیم یعنی این‌بار همین کار را از سطر jام به ستون iام انجام می‌دهیم.



	A	B	C	D	E	F	G	H	I
A	0	1	1	1	0	0	0	0	0
B	1	0	0	0	1	1	0	0	0
C	1	0	0	1	0	0	1	0	0
D	1	0	1	0	0	0	1	1	0
E	0	1	0	0	0	0	0	0	1
F	0	1	0	0	0	0	0	0	0
G	0	0	1	1	0	0	0	0	0
H	0	0	0	1	0	0	0	0	0
I	0	0	0	0	1	0	0	0	0

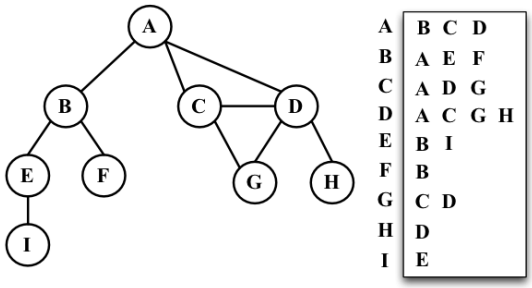
در این شیوه ذخیره سازی چون تعداد سطرها و ستون‌ها برابر تعداد رئوس است پس به فضای حافظه‌ای از $O(n^2)$ نیاز داریم. اگرچه ممکن است کمی زیاد بنظر بیاید، اما خوبی این شیوه در این است که با $O(1)$ می‌توان از وزن یال بین دو راس در صورت وجود اطلاع یافت!

```cpp
#include <iostream>
using namespace std;

const int MAXN = 1000 + 10 //  حداکثر تعداد راس ممکن
int adj[MAXN][MAXN];  //  ماتریس مجاورت

int main()
{
    int n, m;     //  به ترتیب از چپ به راست تعداد رأس‌ها و یال‌ها هستند
    cin >> n >> m;
    for(int i = 0; i < m; i++)
      {
            int v, u;
            cin >> v >> u >> w;
            adj[v][u] = w;
      }
}
```

۲.۲ لیست مجاورت

لیست مجاورت، در واقع لیستی است که به ازای هر رأسی لیستی از مجموعه یال‌هایش (یال‌های خروجی در گراف‌های جهت‌دار) را نگه می‌داریم.پس فضای حافظه‌ی ما به تعداد یال‌ها وابسته است؛ با کمی تفکر میتوان دریافت که فضای مصرفی در گراف‌های بی‌جهت و جهت‌دار به ترتیب دوبرابر و برابر تعداد یال‌هاست.

از آنجایی که برای هر رأسی تعداد یال‌های متفاوتی را نگه می‌داریم، می‌توانیم به ازای هر رأسی یک لیست پیوندی بگیریم، اما از طرفی هم میتوان از ابزارهای دیگری نیز استفاده کرد که به صورت سرشکن فضای اضافه‌ای نمی‌گیرند و یا حتی در عمل کار را ساده‌تر می‌کنند.



با توجه به مطالب گفته شده، مرتبه حافظه‌ای مورد نظر برای اینکار از $O(n+e)$ است که برای گراف های تنک، فضای بهینه و کمی است. اما برای گراف های شلوغ، ماتریس مجاورت بهتر است، چراکه در این حالت با وجود فضای مصرفی‌ای به اندازه ماتریس مجاورت، همچنان برای فهمیدن وجود یال بین دو راس v و u باید لیست را بگردیم که خود مشکل اساسی لیست‌ها در مقابل آرایه ها بود و این یعنی باید در بدترین حالت(یا حتی حالت متوسط) از $O(n)$ زمان مصرف کنیم.

```cpp
#include <iostream>
#include <vector>
using namespace std;

const int MAXN = 1000 + 10 //  حداکثر تعداد راس ممکن
vector <pair<int, int> > adj[MAXN];

int main()
{
    int n, m;
    cin >> n >> m;
    for(int i = 0; i < m; i++)
      {
            int v, u, w;
            cin >> v >> u >> w;
            adj[v].push_back({u, w});
      }
}
```

البته به جز دو روش گفته شده روش های دیگری مثل ماتریس وقوع گراف نیز برای ذخیره سازی انواع گراف مناسب است، همچنین روش هایی نیز برای پردازش موازی گراف ها نیز توسعه داده شده اند!

به طور کلی باید با توجه به عملیات های مورد نظر (پویایی یا ایستایی گراف، حذف و درج راس، حذف و درج یال، بررسی و دسترسی به یک یا چند یال و یا حتی نگه داری تاریخچه تغییرات یا ...) مدل مورد نظر خود را برای نگه‌داری گراف انتخاب کنید.

۳. پیمایش گراف‌ها

پیمایش گراف در حالت کلی به معنی گذشتن و دیدن تمام راس‌های گراف است و معمولا پیمایش از طریق یال‌های موجود در گراف انجام می‌شود.

عموما پیمایش گراف به تنهایی ارزش خاصی ندارد و هدف از پیمایش، محاسبه یا پیدا کردن چیز دیگری است. برای مثال شاید با نگاه کردن به یک گراف بتوانید خواص آن را پیدا کنید و مسئله را حل کنید؛ اما در کامپیوتر به دلیل محدودیت‌هابه همین راحتی نمی‌توان این کار را کرد. الگوریتم‌های پیمایش در حرکت روی گراف و پیدا کردن خواص آن به ما کمک می‌کنند.

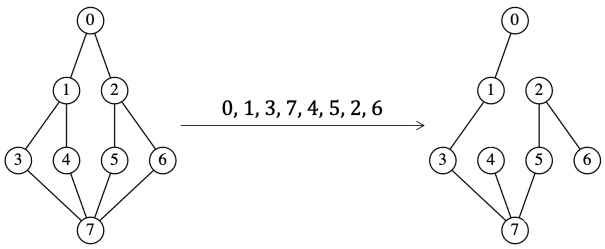
۳.۱ جستجو عمق اول

جست‌وجوی عمق‌اول (Depth First Search) که به DFS معروف است، الگوریتمی برای پیمایش گراف است. شاید با کمی شک بتوان گفت که پرکاربردترین الگوریتم در گراف همین الگوریتم است چراکه هم پیاده‌سازی آن ساده است، هم هزینه زمانی و حافظه‌ی مصرفی آن کم است.

۱. راس شروع را انتخاب کرده و به **پشته** ملاقات اضافه می‌کنیم.

۲. تا زمانی که **پشته** ملاقات خالی نشده:

۲.۱. برای هر راس ملاقات شده، تمام رئوس همسایه آن را که تا به حال ملاقات نشده‌اند را، به **پشته** ملاقات اضافه می‌کنیم.



برای پیاده سازی الگوریتم، از آنجا که در حافظه کامپیوترمان یک پشته برای فراخوانی توابع داریم، نیازی به پیاده سازی یا استفاده از یک پشته به صورت دستی نیست و تنها با فراخوانی بازگشتی می‌توانیم این عملیات حذف و درج را انجام دهیم. (همین هم باعث ساده شدن پیاده‌سازی می‌شود!)

```
void dfs(int v)
{
   mark[v] = 1;
   for(int i = 0; i < adj[v].size(); i++)
      {
            int u = adj[v][i];
            if(mark[u] != 1)
                  dfs(u);
      }
}
```

جست‌وجوی اول‌عمق به تنهایی کاربرد خاصی ندارد و در نتیجه محاسباتی که در کنار آن انجام میشود باعث اهمیت آن میشود. به طور کلی این محاسبات را میتوان به دو دسته پس‌ترتیب و پیش‌ترتیب تقسیم کرد. محاسبات پیش‌ترتیب برای هر رأسی هنگام اولین ورود به آن و محاسبات پس‌ترتیب هنگام آخرین خروج از آن انجام میشود؛ همچنین زمان ورود و خروج از یک راس نسبت به دیگر راس ها یا حتی دنباله بازدید شده از راس ها می تواند به ما اطلاعات جالبی در مورد گراف مورد پیمایش بدهد.

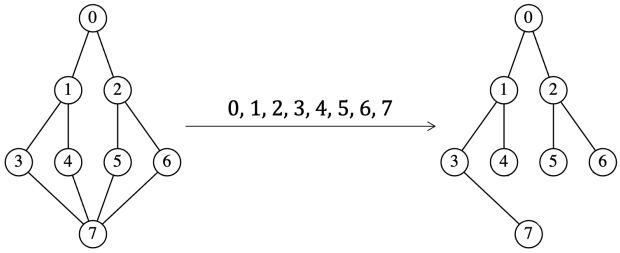
۳.۲ جستجو سطح اول

جستوجوی اول‌سطح (Breadth First Search) که به BFS مشهور است، روشی برای پیمایش گراف است که بعد از جست‌وجوی عمق‌اول مهم‌ترین و پرکاربردترین الگوریتم پیمایش گراف محسوب می‌شود.

۱. راس شروع را انتخاب کرده و به **صف** ملاقات اضافه می‌کنیم.

۲. تا زمانی که **صف** ملاقات خالی نشده:

۲.۱. برای هر راس ملاقات شده، تمام رئوس همسایه آن را که تا به حال ملاقات نشده‌اند را، به **صف** ملاقات اضافه می‌کنیم.



این الگوریتم ابتدا یک راس ریشه مانند v را به عنوان شروع میگیرد. سپس راس‌ها را سطح بندی میکند. سطح بندی به این صورت است که تمام راس‌های مجاور v را در سطح اول قرار میدهیم، حال برای سطح $i+1$، راس u که مجاور یکی از رئوس سطح i است را در این سطح قرار میدهیم به شرطی که u در هیچ سطح دیگری نیامده باشد. به عبارت دیگر راس هارا بر اساس کوتاه‌ترین فاصله از v سطح بندی میکنیم. حال برای پیمایش به ترتیب سطح، و در هر سطح به ترتیب دلخواه وارد راس‌ها میشویم. پس اولین زمانی که به هر راس می‌رسیم، با کمترین فاصله ممکن نسبت به v رسیده ایم.

```
void bfs(int v) {
    queue<int> q;

    mark[v] = 1;
    q.push(v);

    while(q.size()) {
        v = q.front(); //     راس ابتدایی را از صف بر میداریم
        q.pop();
        // کارهای پیشترتیب را اینجا مینویسیم
        for(int i = 0; i < adj[v].size(); i++) {
            int u = adj[v][i];

            if (mark[u] == 1) //قبلا این راس را به صف اضافه نکرده باشیم
                continue;

            mark[u] = 1;
            q.push(u);
        }

    }
}
```

۳.۳ مرتبه اجرایی:

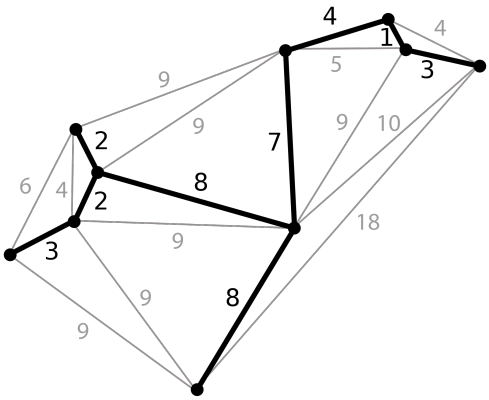
- اگر گراف G توسط ماتریس مجاورت ارائه شود، آن‌گاه زمان لازم برای هر دو پیمایش $O(n^2)$ است.
- اگر گراف G توسط لیست مجاورت ارائه شود، آن‌گاه زمان لازم برای هر دو پیمایش $O(n+e)$ است.

از آنجا که e حداکثر از $O(n^2)$ است(چرا؟) پس در بدترین حالت زمان اجرایی برابر با $O(n^2)$ است.

۴. درخت پوشا کمینه

فرض کنید گراف یک گراف همبند باشد؛ منظور از یک درخت پوشا از این گراف درختی است که شامل همه رئوس این گراف باشد ولی فقط بعضی از یال‌های آن را دربر گیرد.

منظور از درخت پوشای کمینه (برای گراف همبند وزن دار) درختی است که بین درخت‌های پوشای آن گراف، مجموع وزن یال‌های آن، کمترین مقدار ممکن باشد. برای به دست آوردن درخت پوشای بهینه یک گراف جهت دار متصل می‌توان از الگوریتم‌های متفاوتی استفاده نمود. چند الگوریتم معروف پیدا کردن درخت پوشای کمینه عبارتند از: الگوریتم کروسکال، الگوریتم پریم، الگوریتم بروکا (سولین) و الگوریتم حذف معکوس. البته در اینجا صرفا الگوریتم های کروسکال و پریم را با هم مرور می‌کنیم.



۴.۱. ویژگی های درخت پوشا کمینه

لطفاً سعی کنید این ها را اثبات کنید.

۴.۱.۱. تعداد حالات ممکن

امکان وجود بیش از یک درخت پوشای کمینه وجود دارد، برای مثال اگر تمام یال‌ها وزن برابری داشته باشند، تمام زیر درخت‌ها درخت پوشای کمینه هستند. در واقع تنها در حالتی که وزن هر دو یال با هم متفاوت باشد، یک درخت پوشای کمینه خواهیم داشت.

۴.۱.۲. کم وزن‌ترین یال در برش گراف

اگر گراف را به دو مجموعه V و V' از راس‌ها افراز کنیم، کم‌وزن‌ترین یال بین یال‌هایی که یک طرفشان در مجموعه V و طرف دیگرشان در مجموعه V' است باید جزء درخت پوشای کمینه باشد. همچنین اگر چند یال با کم‌ترین وزن وجود داشت، باید دقیقاً یکی از آن‌ها جزء درخت پوشای کمینه باشد.

۴.۱.۳. کم وزن‌ترین یال گراف

اگر کم‌وزن‌ترین یال گراف یکتا باشد، در هر درخت پوشای کمینه‌ای وجود خواهد داشت.

۴.۱.۴. پر وزن‌ترین یال هر دور

اگر یالی در یک دور از تمام یال‌های موجود در آن دور وزنش بیشتر باشد، نمی‌تواند در درخت پوشای کمینه قرار بگیرد.

۴.۲. کاربرد های درخت پوشا کمینه

این درختان دارای کاربردهای مستقیم در طراحی شبکه ها هستند، از جمله شبکه های رایانه ای، شبکه های ارتباط از راه دور، شبکه های حمل و نقل، شبکه های آب‌رسانی و شبکه‌های الکتریکی (که اولین بار برای آنها اختراع شدند).

برای الگوریتمهای دیگر، از جمله الگوریتم کریستوفید که برای تقریب جواب مسئله فروشنده دوره‌گرد استفاده می‌شود؛ به عنوان زیر برنامه در الگوریتم فراخوانی می شوند و البته جواب نهایی نیز براساس آنها بدست می‌آید.

۴.۳. الگوریتم کراسکال

همانطور که گفتیم این الگوریتم برای پیدا کردن کمینه زیردرخت فراگیر T از گراف G، استفاده می‌شود؛ این الگوریتم بر خلاف الگوریتم پریم لزوما اجزایی که در حین اجرا جزء درخت پوشای کمینه تشخیص می‌دهد همبند نیستند و تنها تضمین می‌کند که در پایان این شرط برقرار است.

۱. یال e_1 را طوری انتخاب کن که وزن آن کوچکترین مقدار موجود باشد.

۲. اگر یال‌های $\{e_1, e_2, \ldots, e_i\}$ انتخاب شده‌اند، یال e_{i+1} را به گونه‌ای انتخاب کن که:

۲.۱. عضو e_{i+1} از میان $E(G) - \{e_1, e_2, \ldots, e_i\}$ باشد

۲.۲. زیرگراف با یال‌های $\{e_1, e_2, \ldots, e_i, e_{i+1}\}$ بدون دور باشد.

۲.۳. از میان تمام یال‌های مشمول شرط های **۲.۲** و **۲.۱** وزن e_{i+1} دارای کمترین مقدار ممکن باشد.

۳. در صورتی که مرحله II دیگر قابل اجرا نیست توقف کن.

می‌توان گفت که این الگوریتم تنها از خاصیت ۴.۱.۳ که بیان کردیم استفاده میکند(به کفایت تمام!) ، به طور کلی و بیانی ساده تر این الگوریتم در هر مرحله کمینه یال ممکن را انتخاب می‌کند و از میان یال های قابل انتخاب برای مرحله بعد حذف می‌کند، در مرحله بعد همین کار را تکرار میکند و در واقع ادعا می‌کند که این انتخاب حریصانه بهترین انتخاب ممکن است(طبق همان ۴.۱.۳).

```
const int N=1000; // حداکثر تعداد راس ها
int n,m;
vector <pair <int, pair <int, int> > > e; // لیست شامل دو سر یال و وزن آن
vector <pair <int, int> > MST; // لیست شامل یال های جواب نهایی
int tree_ID [N]; // شماره مولفه‌همبندی هر راس

void kruskal(){
      sort (e.begin (), e.end ());
      MST.clear();
//تنظیم کردن شماره مولفه همبندی ها
      for(int i=0; i<n; i++)
          tree_ID[i]=i;

      for(int i=0; i<m; i++){
            int f=e[i].u;
            int t=g[i].v;
            if(tree_ID[f]!=tree_ID[t]){
                  MST.push_back(e[i]);
                  int old_ID = tree_ID [t], new_ID = tree_ID [f];
//یکی کردن شماره مولفه همبندی ها
                  for(int j=0; j<n; j++)
                        if (tree_ID [j] == old_ID)
                              tree_ID [j] = new_ID;
            }
      }
}
```

پیچیدگی زمانی

مرتب کردن یال ها و بررسی یال‌ها از $O(m + m \log n)$ است که برابر $O(m \log n)$ است و هربار اتصال دو مولفه از $O(n)$ است که چون اتصال $n - 1$ بار انجام می‌شود، پیچیدگی الگوریتم $O(m \log n + n^2)$ می‌شود.

البته می‌توان از داده‌ساختار مجموعه‌های مجزا (Disjoint Set Union) برای ادغام مولفه‌های همبندی استفده کرد، در این صورت پیچیدگی الگوریتم به $O(n + m \log n)$ که برابر $O(m \log n)$ است کاهش پیدا می‌کند.

۴.۴. الگوریتم پریم

همانند الگوریتم کروسکال این الگوریتم نیز برای پیدا کردن کمینه زیردرخت فراگیر T از گراف G، استفاده می‌شود؛ این الگوریتم مرتب‌سازی درخت را که از یک یال شروع شده‌است، افزایش می‌دهد تا جایی که که همه رئوس وارد درخت شوند.

این الگوریتم را به‌طور خلاصه می‌توان چنین شرح داد:

۱. ورودی: گراف همبند وزن دار $G(V, E)$.

۲. مقدار دهی اولیه: $V_T = \{u\}$ که V_T مجموعه رئوس درخت پوشای کمینه در حالت آغازین را نشان می‌دهد و u یک راس دلخواه است(نقطه شروع) و $E_T = \{\}$ که E_T بیانگر مجموعه یال‌های این درخت است.

۳. حلقه زیر را تا وقتی که $V_T \neq V$ تکرار کن:

۳.۱. یال (u, v) را با وزن کمینه انتخاب کن به‌طوری‌که v در V_T قرار داشته باشد ولی u عضوی از این مجموعه نباشد.

پ.ن: اگر چند یال با وزن یکسان وجود دارند یکی را به دلخواه انتخاب کن(طبق ۴.۱.۴ واقعا این یال ها تفاوتی ایجاد نمی‌کنند.)

۳.۲. راس u را به V_T و یال (u, v)را به E_T اضافه کن.

۴. خروجی : V_T و E_T درخت پوشا کمینه را توصیف می‌کنند.

الگوریتم پریم را به این صورت نیز می‌توان بیان کرد، ابتدا گره‌ای به دلخواه انتخاب شود و سپس از بین یال‌های متصل به آن یالی که با کمترین وزن انتخاب می‌شود متصل شود به گونه‌ای که دور ایجاد نشود.

پس در الگوریتم پریم دو محدودیت در هر مرحله داریم یکی آن که جنگل ایجاد نشود و دوم آنکه دور شکل نگیرد.

```
int total_weight = 0;
vector<bool> selected(n, false);
vector<Edge> min_e(n);
void prim(){
    min_e[0].w = 0;
    for (int i=0; i<n; ++i) {
        int v = -1;
        for (int j = 0; j < n; ++j) {
            if (!selected[j] && (v == -1 || min_e[j].w < min_e[v].w))
                v = j;
        }
        selected[v] = true;
        total_weight += min_e[v].w;
        for (int to = 0; to < n; ++to) {
            if (adj[v][u] < min_e[u].w)
                min_e[u] = {adj[v][u], v};
        }
    }
}
```

پیچیدگی زمانی

در یک پیاده‌سازی ساده، که در آن به دنبال آرایه‌ای از وزن‌ها هستیم و در هر مرحله یال‌های با وزن کمینه را به مجموعه خود اضافه می‌کنیم، از مرتبه $O(n^2)$ زمان می‌برد.

اما الگوریتم پریم را با استفاده از داده ساختار هیپ می‌تواند در زمان $O(m \log n)$اجرا شود. استفاده از مدل پیچیده تری به نام هیپ فیبوناچی نیز باعث می‌شود این زمان تا$O(m + n \log n)$ کاهش یاب؛ سرعت این روش به خصوص زمانی آشکار می‌شود که گراف به اندازه کافی شلوغ باشد یا به عبارتی در گراف رابطه $m = \omega(n)$ (این رابطه یعنی چه؟) بین رئوس و یالها برقرار باشد.

۵. تمارین مروری

۱. (تست دو بخشی بودن) الگوریتم را طراحی کنید که راس های گراف دلخواه G را با استفاده از دو رنگ های به گونه‌ای رنگ‌آمیزی کند که هیچ دو راس مجاوری هم رنگ نباشند؛ آیا چنین رنگ‌آمیزی همیشه ممکن است؟

۲. الگوریتمی را توسعه‌دهید که برای گراف دلخواه G بررسی کند که آیا شامل هیچ دوری هست، یا خیر؟ آیا می‌توانید تعداد دورها را نیز در الگوریتم خود بیابید؟ مرتبه زمانی الگوریتم خود را بهبود ببخشید یا که بهینه بودن آن را ثابت کنید.

۳. (دایکسترا) در یک گراف ساده G به شما دو راس u و v داده می‌شود، فاصله بین این دو را بیابید. (آیا می‌توانید الگوریتم خود را به گراف های وزن دار نیز توسعه دهید؟)

۴. از گراف ساده G به شما دو مجموعه راس U و V داده شده است، برای هر راس $u \in U$ کمینه فاصله آن را تا مجموعه V بیابید؛ کمینه فاصله را طول کوچکترین مسیر از u به هرکدام از راس های $v \in V$ تعریف می‌کنیم. (آیا می‌توانید الگوریتم خود را به گراف های وزن دار نیز توسعه دهید؟)

۵. (فلوید-وارشال) آیا می‌توانید کمینه مسیر بین دو به دو رئوس G را از مرتبه $O(n^3)$ بیابید؟

۶. (بلمن-فورد) آیا می‌توانید کوتاه ترین مسیر بین دو به دو رئوس G را از مرتبه $O(n + e)$ بیابید؟

۷. (یافتن قطر) برای درخت T طولانی ترین مسیر موجود در آن درخت را بیابید. (الگوریتم خود را به گراف ها نیز توسعه دهید)

۸. برای گراف ساده G مولفه های همبندی آن را بیابید.

۹. (Kosaraju) برای گراف جهت‌دار D مولفه های قویا همبند آن را پیدا کنید.

۶. سوالات برنامه‌نویسی

۱. کوئرا، هیچ وقت مغرور نشو!

۲. کوئرا، علی خلافه

۳. کوئرا، خسته تر از امید خودشه

۴. کوئرا، دزد و تمرکز

۵. کوئرا، مرید تنبل

۶. کوئرا، زندان

۷. برای جستجو

1. Monte Carlo Tree Search
2. best-first search
3. Disjoint Set Union
4. Flood-Fill
5. Max Flow- Min Cut
6. Boolean satisfiability problem
7. Karp's 21 NP-complete problems
8. TREE-REGULAR: Regular Tree-Valued Languages