

Увод в задачите за графи

Съдържание

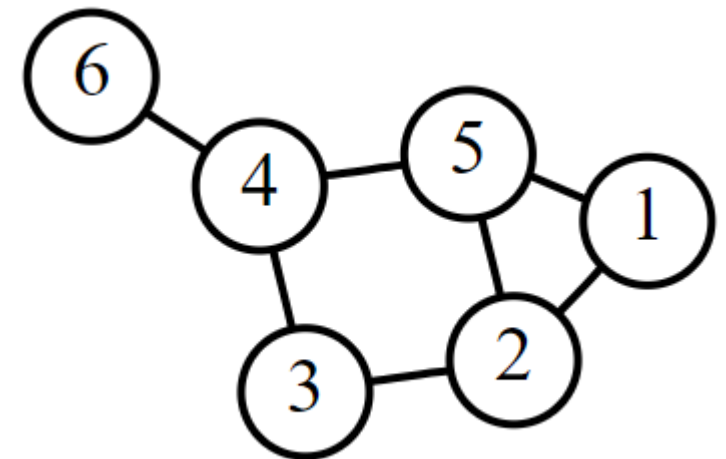
- 1. Видове графи**
- 2. Представяне на графи**
- 3. Обхождане на графи**
- 4. Приложения на обхождането**
- 5. Двусвързаност и точки на съчленяване**
- 6. Топологично сортиране**
- 7. Ойлеров цикъл**
- 8. Най-къс път**
- 9. Минимално покриващо дърво**

Що е граф и видове графи

Прост граф (понякога казваме „граф“):
Комбинаторно понятие (което може да го изобразяваме)

- множество от върхове
- множество от двойки върхове (ненаредени двойки), наречени ребра. Между двойка върхове може да има най-много едно ребро.

Примки – когато има ребро от един връх до същия връх



Още терминология:

- висящи върхове
- изолирани върхове
- степен на връх
- съседни върхове
- инцидентни ребра с връх
- път
- цикъл (от върхове или от ребра)
- подграф
- пълен граф
- клика
- премахване на върхове (заедно с инцидентните им ребра)
- свързан граф
- компоненти на свързаност
- равнинен (планарен) граф

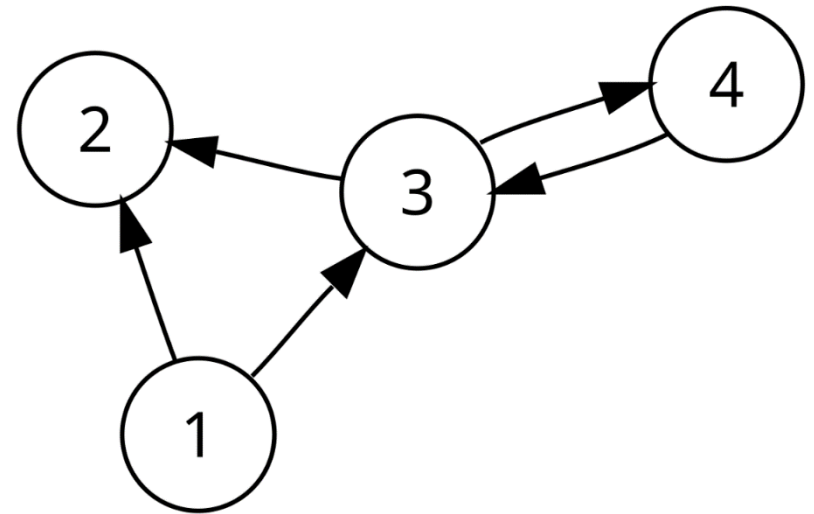
Ориентиран (насочен, directed) граф

Ребрата имат посока и се изобразяват със стрелки.

Понякога се наричат **дъги**.

Задават се като **наредени двойки** от върхове.

Понякога простият граф се нарича неориентиран.



Терминология за ориентиран граф – подобна е на терминологията за неориентиран, но има някои особености.

- ориентиран път
- входящи и изходящи ребра за връх
- силно свързан граф – между всеки два върха съществува ориентиран път.

Претеглен (натоварен, утежнен, weighted) граф (ориентиран или неориентиран)

На всяко ребро е присвоена определена числова стойност — наричаме я тегло (или разстояние).

Понякога се разглеждат графи, в които числова стойност се задава за върховете им.

Мултиграф

Когато повече от едно ребро може да свързва два върха (а при ориентиран граф – възможно е тези ребра да са ориентирани еднакво, или различно).

Дърво (като частен случай на граф)

Това е свързан граф без цикли

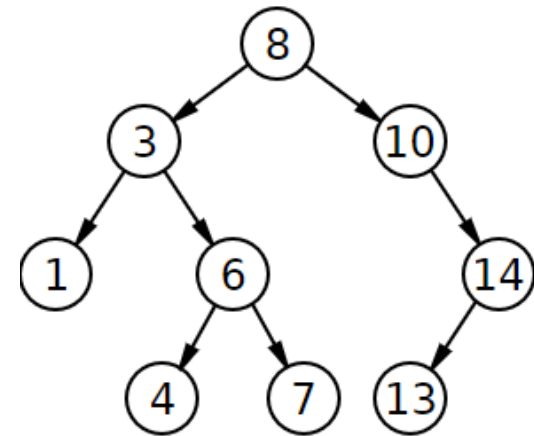
Има неориентирани и ориентирани дървета

Кореновото дърво е с естествена ориентация. Има следните свойства:

1. съществува връх, към който не сочи нито едно ребро (нарича се *корен*)
2. към всеки друг връх на дървото сочи точно едно ребро,
3. съществува единствен път от корена до всеки връх на дървото.

Терминология:

- Корен
- Листо
- Деца
- Родител
- Поддърво
- Височина на дърво
- Двоично дърво
- Подграф, който е дърво.
- Покриващо дърво



Дърво като структура от данни. Рекурсивна дефиниция:

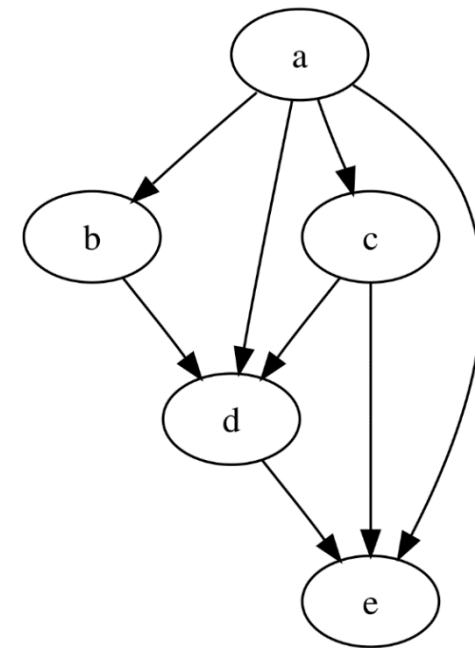
дърво е

връх без деца, ИЛИ

връх с няколко деца, всяко от които е **дърво**

И още понятия:

- гора
- ацикличен ориентиран граф (Directed Acyclic Graph – DAG), който не е дърво



Представяне на графи

- списък от ребра
- списък от съседни за всеки връх
- матрица на съседство
(по редове и стълбове стоят номера на върхове)
- матрица на инцидентност
(по редове стоят номера на върхове, а по стълбове — номера на ребра)

Програмна реализация за превръщане от списък на ребра в списък на съседни за всеки връх

с масив от вектори

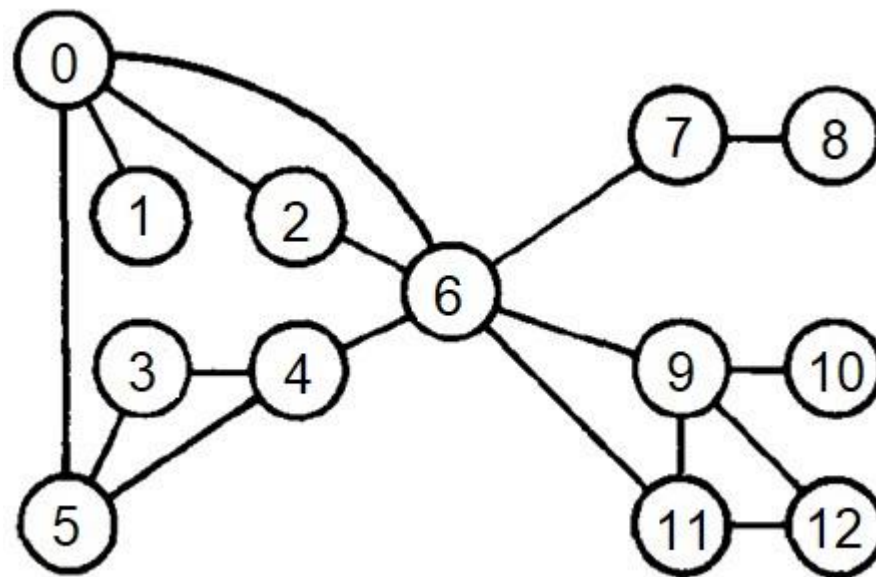
```
const int N_max=1000;
int n, m;
vector<int> r[N_max];

cin >> n >> m;
for(int i=1;i<=m;i++)
{
    int a,b; cin >> a >> b;
    r[a].push_back(b);
    r[b].push_back(a);
}
```

```

for(int j=0;j<n;j++)
{
    cout << j << " : ";
    for(auto v: r[j]) cout << " " << v;
    cout << endl;
}

```



с вектор от вектори

```
int n, m;
vector<vector<int>> r;

void input()
{
    cin >> n >> m;
    r.resize(n);
    for(int i=1; i<=m; i++)
    {
        int a, b; cin >> a >> b;
        r[a].push_back(b);
        r[b].push_back(a);
    }
}
```

```
for(int j=0;j<n;j++)  
{  
    cout << j << " : ";  
    for(auto v: r[j]) cout << " " << v;  
    cout << endl;  
}
```


Обхождане в дълбочина (DFS)

Започваме от произволен връх v и прилагаме рекурсивна функция

```
DFS (да посетим връх  $v$ )  
{  
    маркираме  $v$  като посетен;  
    посещаваме всички върхове  $w$ , които  
        са немаркирани съседи на  $v$ ;  
}
```

Времето за работа е пропорционално на броя на ребрата.

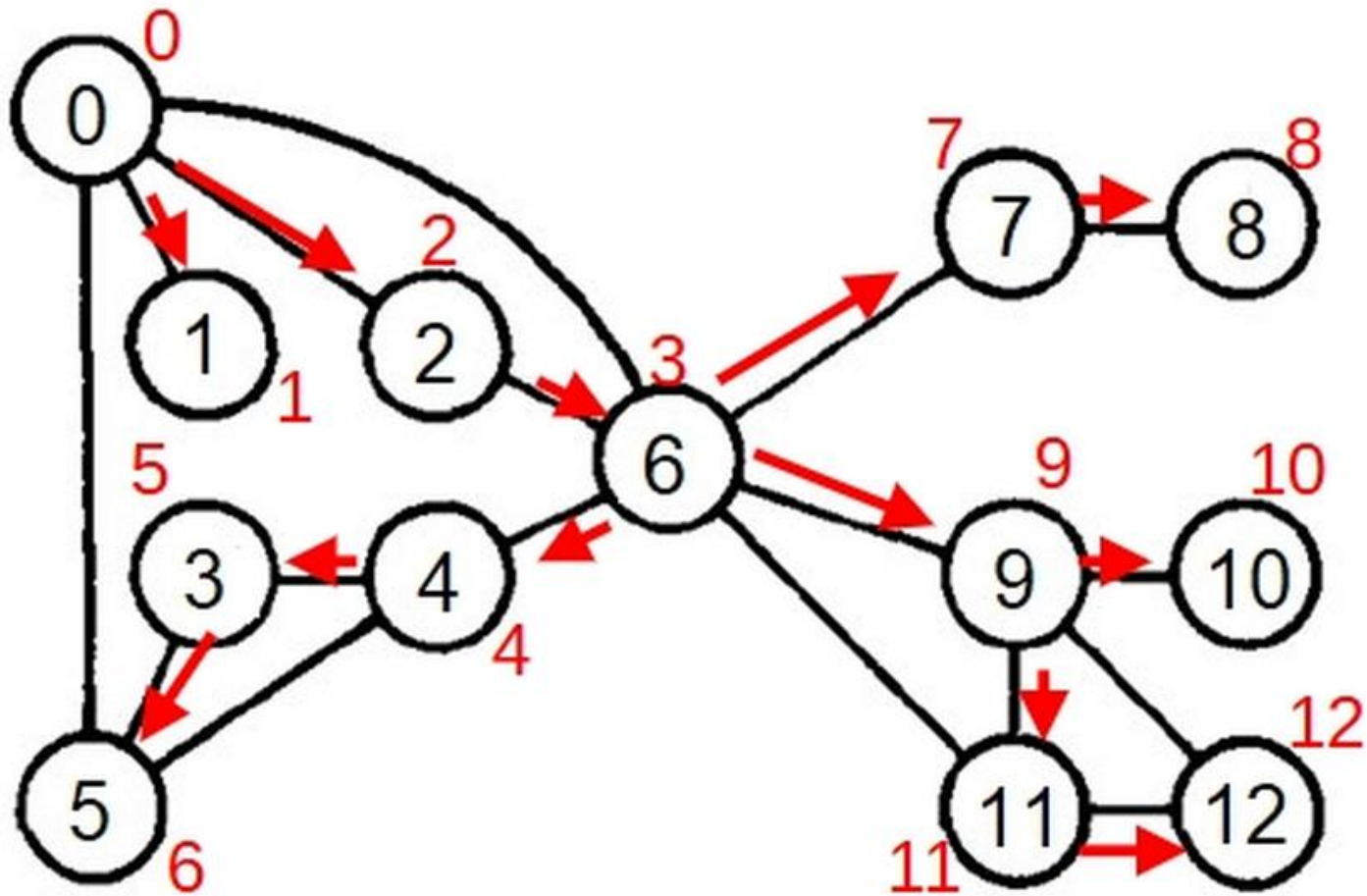
```

vector<int>u(n,-1);
int c=0;
int v=0;u[v]=c;// нови номера на върховете

void dfs(int v)
{
    for(auto w : r[v])
        if(u[w]==-1) {c++; u[w]=c; dfs(w); }
}

int main()
{
    input();
    dfs(0);
    for(int i=0;i<n;i++) cout<<u[i]<<" ";
}

```



$v =$	0	1	2	3	4	5	6	7	8	9	10	11	12
$u[v] =$	0	1	2	5	4	6	3	7	8	9	10	11	12

Програма за намиране на дървото на *правите ребра* и на *обратните ребра* при DFS

```
int n, m;
vector<vector<int>> r;
vector<int>u;
set<pair<int,int>> s,s0;
vector<int>t1, t2;
int c;

void dfs(int v)
{
    for(auto w : r[v])
        if(u[w]==-1)
            {c++; u[w]=c; t1[c]=v; t2[c]=w; dfs(w); }
}
```

```
void dodfs(int n)
{
    u.resize(n, -1);
    c=0;
    int v=0; u[v]=c;
    int cr=0;
    t1.resize(n, -1); t2.resize(n, -1);
    dfs(v);

    for(int i=0; i<n; i++) cout << u[i] << " ";
    cout << endl;
```

```

cout << "DFS tree (прави ребра):" << endl;
for(int j=1; j<t1.size(); j++)
    cout << t1[j] << " " << t2[j] << endl;
cout << "Обратни ребра:" << endl;
s0=s;
for(int j=1;j<t1.size();j++)
{s0.erase({t1[j],t2[j]});
 s0.erase({t2[j],t1[j]});}
for(auto p : s0)
    cout << p.first << " " << p.second << endl;

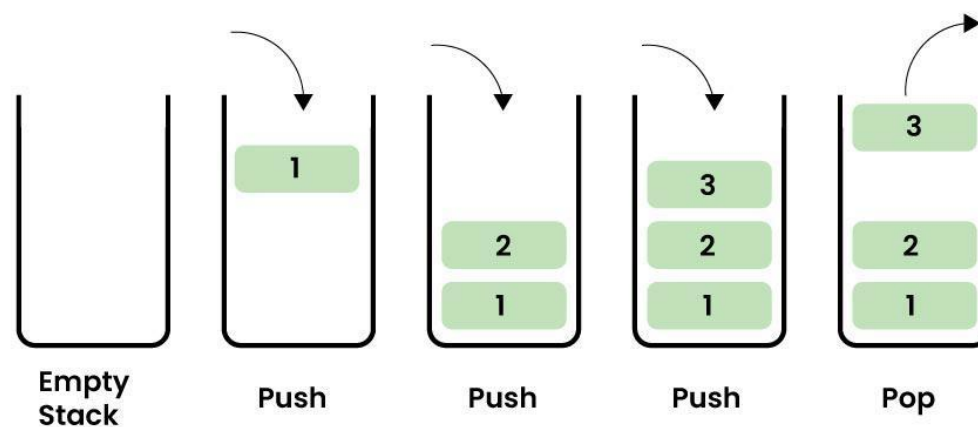
}

```

```
void input()
{
    cin >> n >> m;
    r.resize(n);
    for(int i=1;i<=m;i++)
    {
        int a,b; cin >> a >> b;
        r[a].push_back(b);
        r[b].push_back(a);
        s.insert({a,b});
    }
}
```

```
int main()
{input(); dodfs(n);}
```

При търсене в дълбочина: Непосетените върхове се поставят (неявно или явно) в стек.

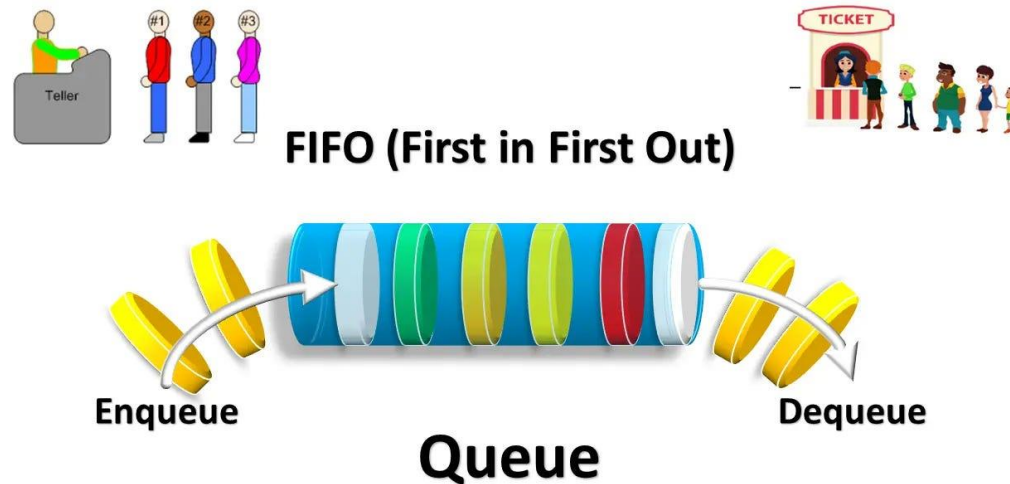


LIFO Operations in stack



Търсене в ширина (BFS)

При търсене в ширина: Непосетените върхове се поставят в опашка.



BFS всъщност намира най-късия път (в смисъл на път с най-малък брой ребра) от началния връх до всеки друг.

Идея на алгоритъма:

Поставете началния връх v_0 в **опашка** Q (Q е FIFO структура от данни).

Повторете, докато опашката Q не е празна:

- премахнете от Q най-дълго стоялия там връх v .
- добавете всеки от непосетените съседи на v към опашката, и ги маркирайте като посетени.

Свойство: BFS обработва върховете по нарастващо разстояние от началния връх v_0 .

Реализация на BFS

```
int n, m; vector<vector<int>>>r;  
vector<bool>u; vector<int>L;  
  
void bfs(int v0)  
{  
    queue<int> q; u[v0] = true; q.push(v0);  
    while (!q.empty())  
    {int v = q.front(); q.pop();  
        cout << v << " (" << L[v]<<" ) ";  
        for (int w : r[v]) if (!u[w])  
            {u[w]=true; L[w]=L[v]+1; q.push(w); }  
    }  
}
```

```
void do bfs ()  
{  
    u.resize (n, false) ;  
    L.resize (n, 0) ;  
    int v=0 ;  
    bfs (v) ;  
}
```

```
int main ()  
{ input () ; do bfs () ; }
```

Приложение на алгоритмите за обхождане

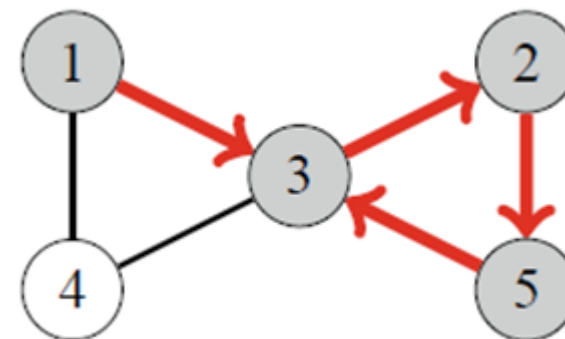
DFS и BFS могат да се ползват за много задачи, но DFS се предпочита, понеже се имплементира малко по-лесно. Обаче BFS се ползва за намиране на най-къс (по брой ребра) път.

Проверка за свързаност. Графът е свързан, ако има път между всеки два върха. Започваме от произволен връх и проверяваме с DFS дали можем да достигнем до всеки друг връх.

По подобен начин можем да намерим всички **свързани компоненти**. Започваме DFS и когато не може да се продължи, започваме ново търсене в дълбочина от връх, който не принадлежи към нито един компонент от намерените компоненти до момента.

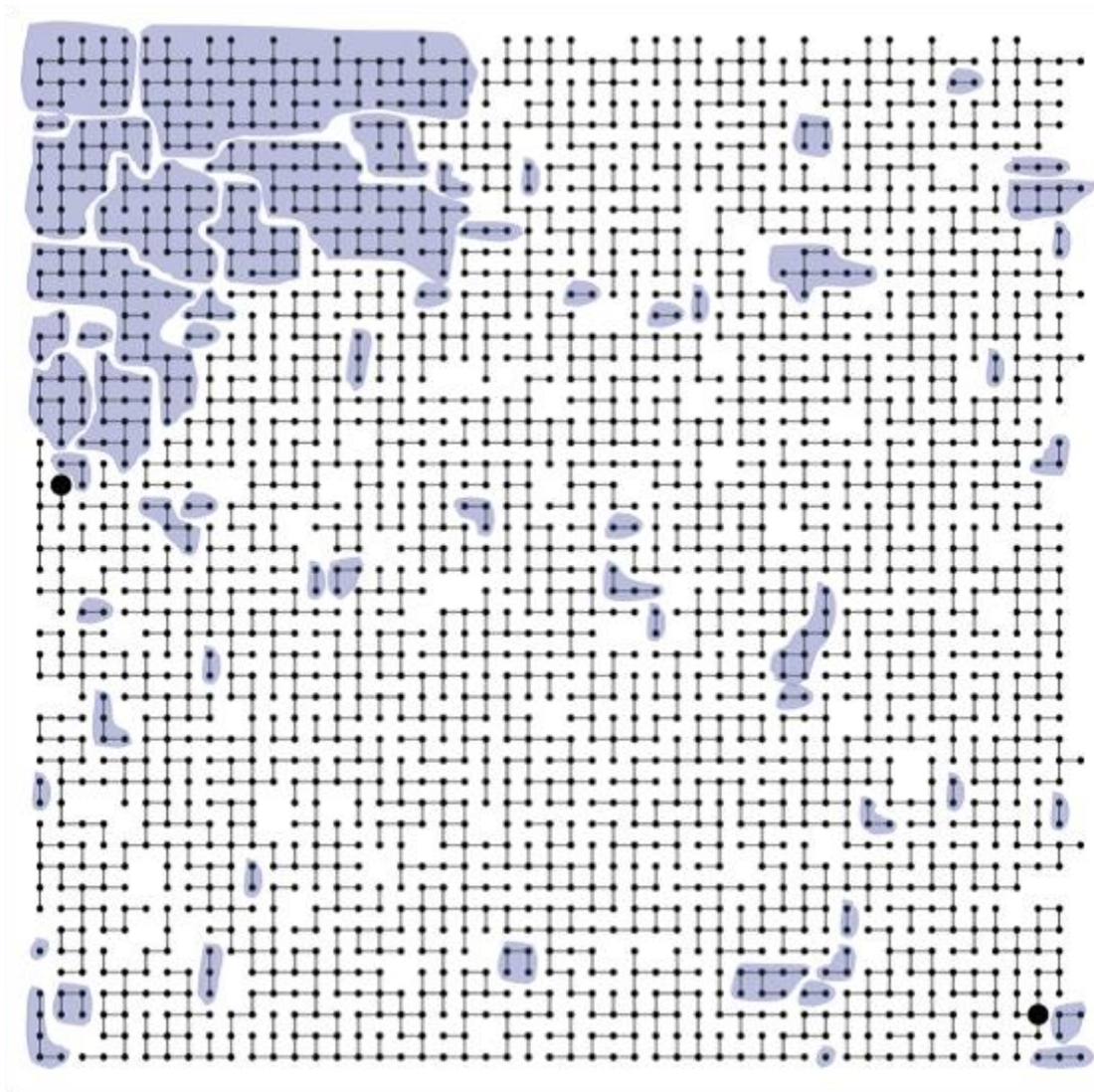
Откриване на цикъл. Графът съдържа цикъл, ако по време на обхождане на намерим връх, чийто съсед (различен от предишния връх в текущия път) вече е бил посетен.

Например, първо търсене в дълбочина от възел 1 намира, че графът съдържа цикъл. След преминаване от връх 2 към връх 5 забелязваме, че съсед 3 на връх 5 вече е бил посетен. По този начин графът съдържа цикъл, който преминава през връх 3, например $3 \rightarrow 2 \rightarrow 5 \rightarrow 3$.



Друг начин да определите дали графът съдържа цикъл е да пресметнем броя на върховете и ребрата във всеки компонент.

Ако даден компонент съдържа s възли и няма цикъл, той трябва да съдържа точно $s - 1$ ребра (така трябва да е дърво). Ако има s или повече ребра, компонентът със сигурност съдържа цикъл.



тук има 63 компонента на
свързаност

Проверка, дали графът е двуделен (двустраниен) - (Bipartiteness Check)

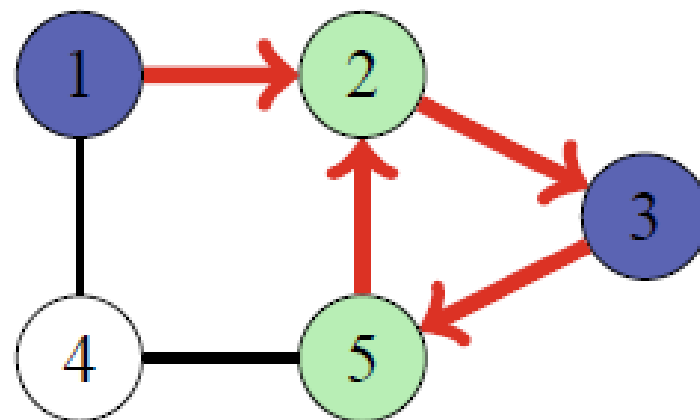
Графът е двустраниен, ако неговите върхове могат да бъдат оцветени с два цвята, така че да няма съседни върхове с един и същ цвят.

Това се проверява чрез обхождане DFS.

Нека цветовете са X и Y . Идеята е да оцветим началния връх с цвят X , всичките му съседни с цвят Y , всичките техни съседни отново с цвят X и т.н. Ако в даден момент от обхождането забележим, че два съседни върха имат еднакъв цвят, това означава, че графът не е двустраниен.

В противен случай графът е двустраниен и е намерено едно оцветяване.

Например при търсене в дълбочина, започвайки от връх 1 се получава, че е графът не е двустранен, защото и двата върха 2 и 5 трябва да имат един, а те са съседни.



В общия случай за $k > 2$ цвята е трудно да се установи дали един граф може правилно да се оцвети – няма бърз полиномиален алгоритми и трябва да се прави изчерпващо търсене.

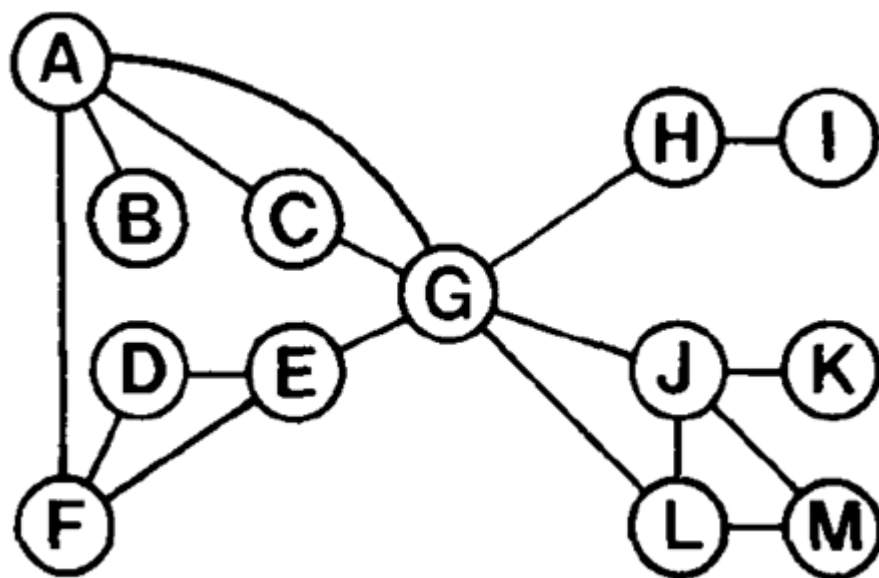
Двусвързаност (**Biconnectivity**)

Критичен връх (артикуляционна точка — точка на съчленяване) е такъв, че премахването му води до увеличаване броя на свързаните компоненти. Ако графът е бил свързан преди премахването на върха, той ще бъде несвързан след това.

Мостът е ребро, аналогично на критичния връх, тоест премахването на мост увеличава броя на свързаните компоненти на графа. В едно дърво всеки връх със степен по-голяма от 1 е критичен връх.

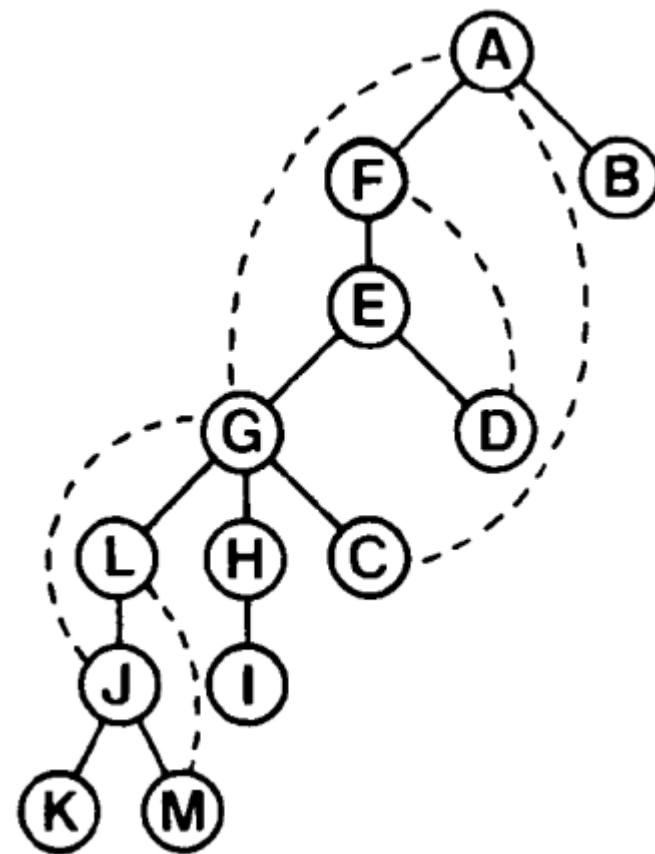
Двусвързаният граф е свързан граф без критични върхове. С други думи, двусвързаният граф е неразделим, което означава, че ако някое ребро бъде премахнато, графът ще остане свързан. Двусвързаността е важна от практическа гледна точка, за да бъде графът надежден, ако например с него се моделира транспортна система. *Това дава двукратен запас* за предотвратяване на прекъсване на функционирането на системата при премахване на ребра.

Пример: Критичните върхове в този граф са: А (защото свързва В с останалата част от графа), Н (защото свързва І с останалата част от графа), J (защото свързва К с останалата част от графа) и G (защото графът ще се раздели на три части, ако G бъде изтрито). Има шест двусвързани компонента: ACGDEF, GJLM и отделните възли В, Н, І и К



Намирането на критичните точки може да стане чрез DFS и използване на правите и обратните ребра

Изтриването на възел E няма да прекъсне връзката с графа, защото G и D имат пунктирани връзки (обратни ребра при DSF), които сочат над E, давайки алтернативни пътища от тях до F (бащата на E в дървото). Но, изтриването на G ще прекъсне връзката с графиката, защото няма такива алтернативни пътища от L или H до E (бащата на L).



Връх x не е критичен, ако за всеки негов син y има някакъв връх по-надолу в поддървото му, свързан (чрез пунктирана връзка, т.е. чрез обратно ребро) с връх, по-нагоре от x в дървото (така се осигурява допълнителна връзка от x към y).

Обаче, гореописаното не работи, когато x е корена на дървото, получено от DFS, тъй като няма възли „по-високо в дървото“. В този случай трябва да се направи отделна проверка за броя на децата на x в дървото, получено от DFS, защото, както лесно се вижда, коренът е критичен връх, тогава и само тогава, когато той има два или повече синове, тъй като единственият път, свързващ синовете на корена, минава през корена.

Програмна реализация. Намиране на критична точка – наивен подход: Премахваме поотделно всеки връх (заедно с инцидентните му ребра) и проверяваме дали полученият граф е свързан.

```
int n,m;
vector<int> a,b;
vector<vector<int>> r;
void input()
{ cin >> n >> m;
  for(int i=1;i<=m;i++)
  { int v1,v2; cin >> v1 >> v2;
    a.push_back(v1); b.push_back(v2); }
}
```



```
vector<int> u;  
int c;  
void dfs(int v)  
{  
    if(u[v] != -1) return;  
    c++;  
    u[v]=c;  
    for(auto w : r[v]) dfs(w);  
}
```

```

set<int>s;
void dodfs ()
{
    for(int j=0;j<n;j++)
    { r.clear(); r.resize(n);
      for(int i=0;i<m;i++)
      {
          int v1=a[i]; int v2=b[i];
          if(v1!=j && v2!=j)
              {r[v1].push_back(v2);r[v2].push_back(v1);}
      }
    c=-1; u.clear(); u.resize(n,-1); int k0=0;
    for(int k=0; k<n; k++)
        if(k!=j)if(r[k].size()>0) {k0=k; break;}

```

```

        dfs(k0);
        if(c < n-2) s.insert(j);
    }
}
int main()
{ input(); dodfs(); cout << s.size() << endl;
    if(s.size()>0)
        {bool b=false;
            for(auto v : s)
                {if(b) cout << " "; b=true; cout << v;}
            cout << endl;
        }
}

```

По-добър начин: за всеки връх проверка чрез обратните ребра

```
int V; vector<vector<int>> r, rf, rr;
set<pair<int,int>>srf;
vector<int>p; int c,c0;
set<int>s;

void input()
{ cin >> V; int m; cin >> m;
  r.resize(V); rf.resize(V); rr.resize(V);
  for(int i=0; i<m; i++)
  { int a,b; cin >> a >> b;
    r[a].push_back(b);r[b].push_back(a);
  }
}
```

```

// намиране на правите ребра (rf) и
// намира колко деца има коренът (c0)

void dfs0(int v)
{
    p[v]=c; c++;
    for(int j = 0; j < r[v].size(); j++)
    { int w=r[v][j];
        if(p[w]==-1)
        {if(v==0) c0++; rf[v].push_back(w);
            srf.insert({v,w}); dfs0(w);
        }
    }
}

```

```

void find_rr() // намира обратните ребра (rr)
{
    for(int v=0;v<V;v++)
    {
        for(int j = 0; j < r[v].size(); j++)
        { int w=r[v][j];
            if(srf.count({v,w})==0 &&
                srf.count({w,v})==0)
                if(p[v]>p[w]) rr[v].push_back(w);
        }
    }
}

```

```

bool con, AP;
int v0;
void dfs(int v) // записва con=true, ако намери
//обратно ребро, сочецо по-нагоре от v0
{
    for(int k=0;k<rr[v].size();k++)
    {
        int ww=rr[v][k];
        if(p[ww]<p[v0]) {con=true;}
    }
    for(int j = 0; j < rf[v].size(); j++)
    {
        int w=rf[v][j];
        dfs(w);
    }
}

```

```

void find_AP()
{ for(int v=0;v<V;v++) s.insert(v);
  for(int v=0;v<V;v++) // проверява всеки връх
  {
    bool AP=true; v0=v;
    for(int j=0;j<rf[v].size();j++)
      // проверява децата
      { con=false; dfs(rf[v][j]);
        if(!con) AP=false; // когато поддървото на
          //всяко дете няма нужното обратно ребро
      }
    if(AP) s.erase(v0); //остават само крит.
      // върхове
  }
}

```



```

int main()
{ input();
  p.resize(V,-1); c=0;
  dfs0(0); // намира правите ребра
  find_rr(); намира обратните ребра
  find_AP(); намира крит. върхове
  if(c0<2) s.erase(0); // коренът има едно дете
  cout << s.size() << endl;
  if(s.size()>0)
  {bool b=false;
   for(auto v : s)
    {if(b) cout << " "; b=true; cout << v;}
   cout << endl;
  }
}

```

Програмата за намиране на критични върхове може да се направи още по-добра, като се извършва само едно търсене в дълбочина:

```
int V; // брой върхове
vector<vector<int>> r;

void input()
{
    cin >> V; int m; cin >> m;
    r.resize(V);
    for(int i=0; i<m; i++)
    { int a,b; cin >> a >> b;
      r[a].push_back(b); r[b].push_back(a);
    }
}
```

```

set<int>s; // за намерените критични върхове
vector<int>p; // за посетените върхове
int c;

int mdfs(int v)// модифициран dfs
{ int min=c; p[v]=c; c++;
  for(int j = 0; j < r[v].size(); j++)
  {int w=r[v][j];
   if(p[w]==-1)
   {int m=mdfs(w);
    if(m<min)min=m;
    if(m>=p[v]) {s.insert(v); // v е критичен връх
   }
   else if(p[w]<min) min=p[w];
  }
  return min;
}

```

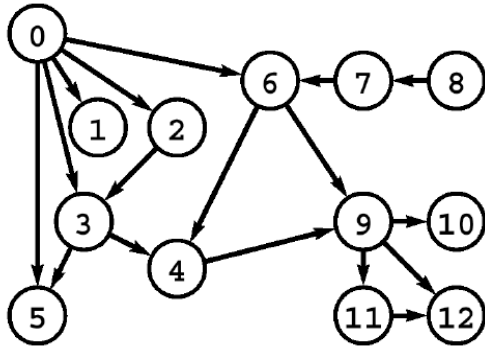
```

int c0;
void dfs0(int v) // за изследване на корена
{
    p[v]=c; c++;
    for(int j = 0; j < r[v].size(); j++)
        { int w=r[v][j];
            if(p[w]==-1) {if(v==0) c0++; dfs0(w);}
        }
}
int main()
{input();
  p.resize(V,-1); c=0; mdfs(0);
  p.clear(); p.resize(V,-1); c=0; c0=0; dfs0(0);
  if(c0<2) s.erase(0);cout << s.size() << endl;
  if(s.size()>0)
    for(auto v : s) cout << v << endl;
}

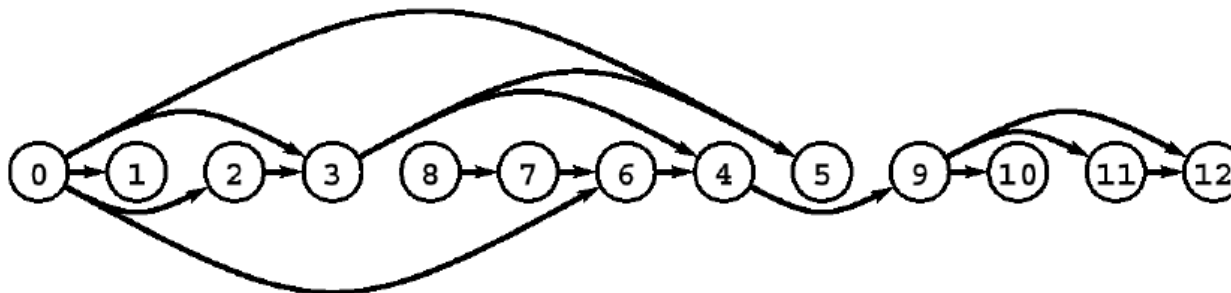
```

Топологично сортиране за ориентиран граф

Даден е DAG – ориентиран ацикличен граф



Да се прерисува така, че всички ребра да сочат **от ляво надясно** (не е възможно ако има цикъл)



Наивен подход: Избираме връх, в който не влизат ребра и го записваме като първи. Изтрива ме този връх от графа (заедно с ребрата му. Повтаряме: избираме връх в новия граф, в който не влизат ребра и правим същото, като този връх го записваме на следващо място и т.н.

Недостатък е, че трябва многократно да променяме графа, което води до бавен алгоритъм.

Добър подход чрез DFS. Преномерирането на върховете при връщането от рекурсията дава търсеното топологично сортиране – когато DFS се връща от връх v , това означава че, всички върхове надолу от v са вече посетени.

```
vector<vector<int>>r; // adjacency list of graph  
vector<bool> u; vector<int> ans;
```

```
void dfs(int v)  
{  
    u[v] = true;  
    for (int w : r[v]) if (!u[w]) dfs(w);  
    ans.push_back(v);  
}
```

```
void top_sort()
{ ans.clear(); u.resize(r.size(), false);
  for (int i=0; i<r.size();i++) if(!u[i])dfs(i);
  reverse(ans.begin(), ans.end());
  for(auto i : ans) cout << i << endl;
}
```

```
void input()
{
  int n,m; cin >> n >> m; r.resize(n);
  for(int i=1;i<=m;i++)
  { int a,b; cin >> a >> b; r[a].push_back(b);
  }
}
```

```
int main()
{ input();top_sort();}
```


Ойлеров цикъл

Разглеждаме при ориентиран граф

Ойлеров цикъл в подграф: Тръгвайки от произволен връх в подграфа и движейки се по стрелките да посетим всяка дъга точно веднъж и да се върнем с началния връх

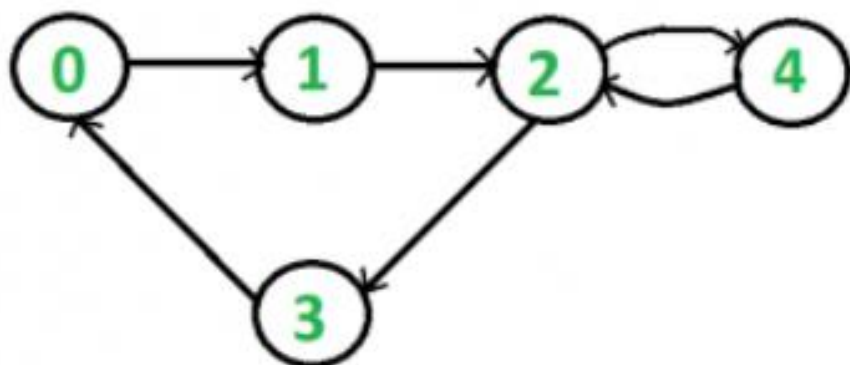
Необходимо и достатъчно условие един граф да е ойлеров е да са изпълнени следните две свойства:

1. За всеки връх броят на влизащите ребра да е равен на броя на излизащите
2. Графът да е силно свързан.

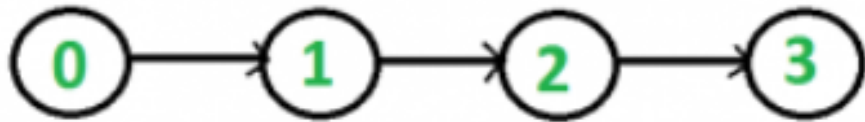
Когато само първото условие е изпълнено може да казваме, че графът има **ойлерово свойство**

Свързаност и силна свързаност

Силно свързан граф:



Граф, който не е силно свързан:



При **неориентиран граф** проверката за **свързаност** става с едно прилагане на DFS (или BFS) от произволен връх

При **ориентиран граф** проверката за **силна свързаност** не може да стане с едно прилагане на DFS (или BFS) от произволен връх. В примера тръгвайки от връх 0, ще посетим всички върхове, но графът не е силно свързан.

Когато обаче графът има ойлерово свойство, тогава проверката за **силна свързаност** може да стане с **едно прилагане** на DFS (или BFS) от произволен връх.

Алгоритъм за проверка на силна свързаност в общия случай на ориентиран граф (Алгоритъм на **Kosaraju** (**Косараджу**)) – чрез две прилагания на DFS

Тръгваме с DFS от произволен връх V и ако не може да стигнем до всички върхове, графът не е силно свързан.

Ако обаче можем да стигнем до всички върхове правим още една проверка с DFS от същия връх V , обаче като обърнем посоките на всички стрелки. Тогава, ако може да стигнем до всички върхове, това означава, че всъщност от всеки връх можем да стигнем до V , понеже дъгите са обърнати и следва графът е силно свързан.

Има по-бърз алгоритъм на Tarjan (Тарджън)

Алгоритъм за намиране на Ойлеров цикъл в ориентиран граф (Hierholzer)

Идея:

Тръгваме от произволен връх v и следваме последователно дъгите, докато се върнем във v . Не е възможно да не върнем. Когато пътят влезе в друг връх w , трябва да има неизползвана дъга, напускаща w . Така получения цикъл обаче може да не покрива всички върхове и ръбове на графа.

Тогава гледаме дали съществува връх u , който принадлежи на текущия цикъл, но u има дъги, които не са част от текущия цикъл. Така може да направим друг цикъл, тръгвайки от u и

движеййки се по непосетени дъги да се върнем в u . Слѣпваме този цикъл с предишния. Това може да се повтори няколко пъти, докато посетим всички дъги.

Алгоритъм за намиране на Ойлеров цикъл в неориентиран граф

Рекурсивна функция `findEulerPath(int j)`

Стартираме DFS от произволен връх j и разглеждаме всички съседни върхове i .

Маркираме свързания връх i като посетен, като премахваме ръба между него и текущия връх j .

След посещаването на всички съседи, поставяме текущия връх j в масива `path[]`.

```
vector<vector<int>> graph =  
{  
    {0, 1, 0, 0, 1},  
    {1, 0, 1, 1, 1},  
    {0, 1, 0, 1, 0},  
    {0, 1, 1, 0, 0},  
    {1, 1, 0, 0, 0}  
};  
int n=5;  
vector<int> path;
```



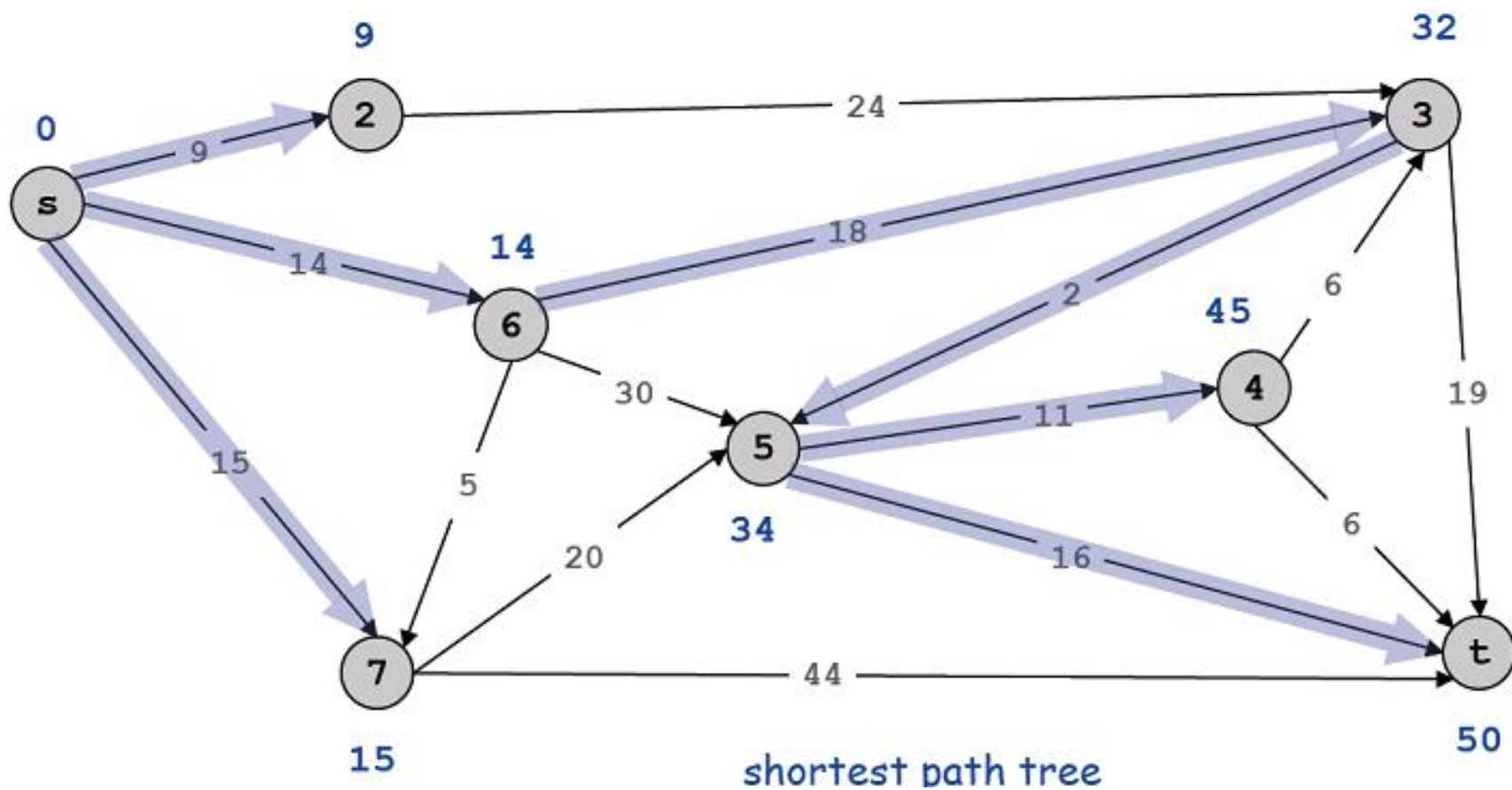
```

void findEulerPath(int j)
{
    for (int i=0; i<n; i++)
        if (graph[j][i] == 1)
        {
            // Remove the edge
            graph[j][i] = graph[i][j] = 0;
            findEulerPath(i);
        }
    path.push_back(j);
}

int main()
{
    findEulerPath(0);
    for (int v:path) cout << v << " ";
}

```

Най-къс път – алгоритъм на Дийкстра



Неориентиран граф с *неотрицателни* тегла

Намират се **всички най-къси пътища** от даден връх s

Итеративен процес, като на всяка стъпка се взема по един необработен връх, който да е най-близък (по-пътя му) до s

Върховете, които подлежат на обработване се намират в приоритетна опашка.

Накрая се получава дърво от най-къси пътища

Времева сложност $O(E \log V)$ за V върха и E ребра

```
const int INF=1e9;
int V; // Брой върхове
vector<vector<pair<int,int>>>r;
void input()
{
    int m;
    cin >> V >> m;
    r.resize(V);
    for(int i=1;i<=m;i++)
    {
        int u,v,w; cin >> u >> v >> w;
        r[u].push_back({v, w});
        r[v].push_back({u, w});
    }
}
```

```

cout << "V=" << V << endl;
for(int j=0;j<V;j++)
{
    cout << j << " : ";
    for(auto p : r[j])
        cout << "(" << p.first << "," << p.second << ") ";
    cout << endl;
}
}

```

```

dijkstra(int src)
{
    priority_queue<pair<int,int>> pq;
    vector<int> d(V, INF);
    pq.push({0, src});
    d[src] = 0; // разстояние до src по
               //намерения до момент път
}

```

```

while (!pq.empty())
{
    int u = pq.top().second;
    pq.pop();
    for (auto p : r[u])
    {
        int v = p.first;
        int w = p.second;
        // При намиране на текущ най-къс път:
        if (d[v] > d[u] + w)
        {
            // Ъпдейтване на разстоянието до v
            // и поставяне в pq:
            d[v] = d[u] + w;
            pq.push({d[v], v}); // първо: разстоянието
                               // второ: върха
        }
    }
}

```

```

    }

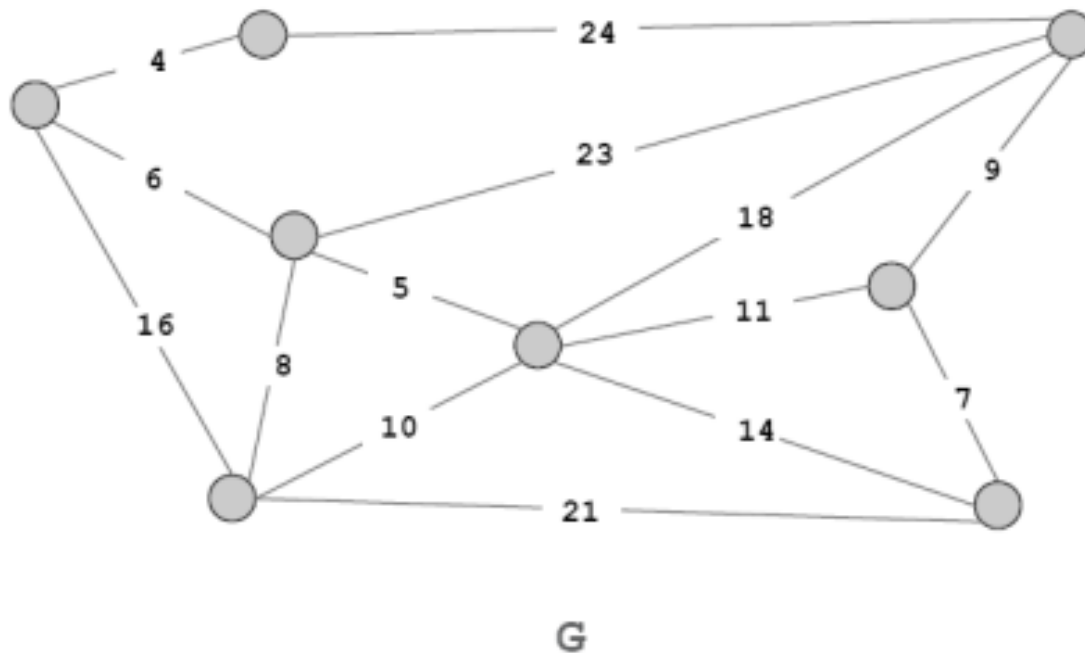
    cout << "Мин. разстояния на върховете до src
           << endl;
    for (int i = 0; i < V; ++i)
        cout << i << " " << d[i] << endl;
}

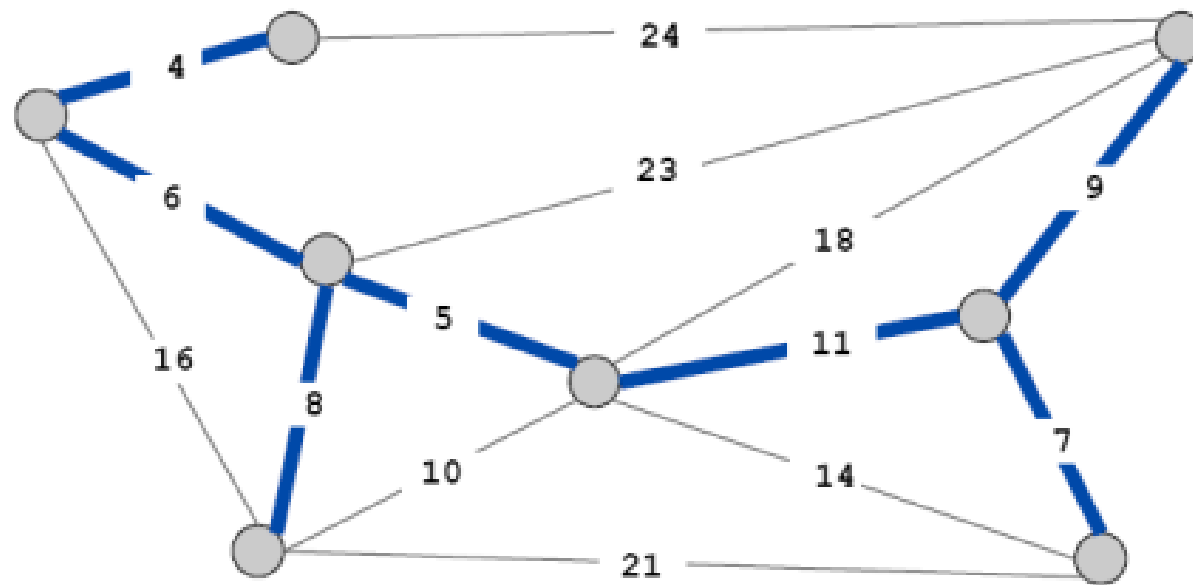
int main()
{ input(); dijkstra(0); }

```

Минимално покриващо дърво - MST

За свързан неориентиран граф с положителни тегла по ребрата, да се намери множество от ребра, което има минимална сума от тегла, така че да покрива всички върхове.





$$\text{cost}(T) = 50$$

Два алчни (грийди) алгоритъма за построяване на минималното покриващо дърво T:

Алгоритъм на **Крускал**. Подреждаме ребрата във възходящ ред на теглата им. Последователно добавяме към T следващо ребро, само ако това ребро не образува цикъл с текущо построеното T.

Сложност $O(E \log E + E \log V)$ при добра реализация

Алгоритъм на **Прим**. Започваме с произволен връх s да построяваме дърво T. На всяка стъпка добавяме реброто с най-малкото тегло измежду неизползваните до момента, което ребро има точно единия си край в T.

Сложност $O(E \log E + V \log V)$ при добра реализация

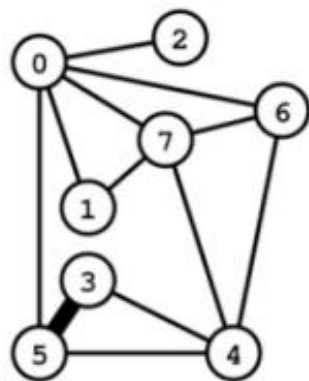
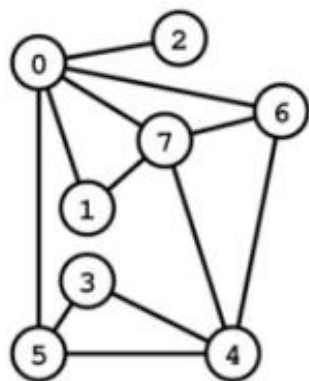
Крускал е по-лесен за кодиране.

Прим изисква имплементиране на Неар (Фибоначиев) за оптимална реализация.

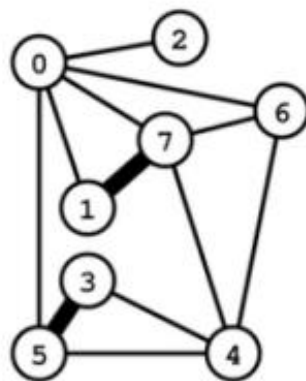
Крускал е за предпочитане, когато графът има „малък“ брой ребра, т.е. $E=O(V)$ и когато ребрата са дадени сортирани по дължина.

Прим е за предпочитане, когато има „много“ ребра, т.е. $E=O(V^2)$.

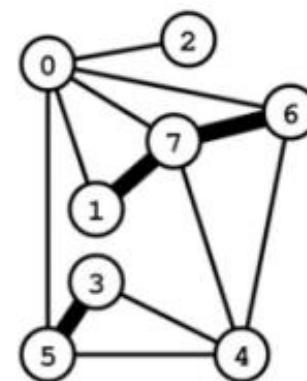
Крускал идея: (теглата на ребрата са геометрич. им дължини)



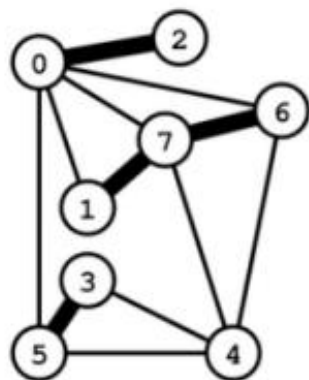
3-5



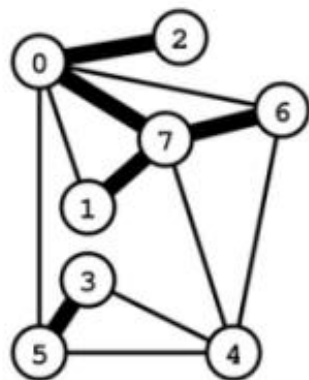
1-7



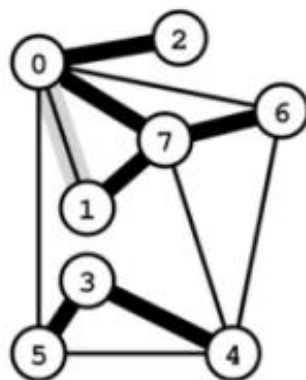
6-7



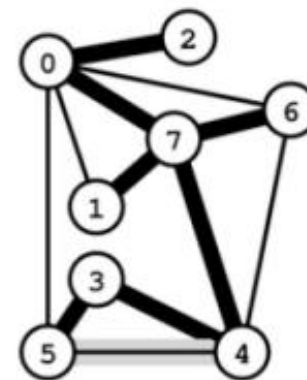
0-2



0-7

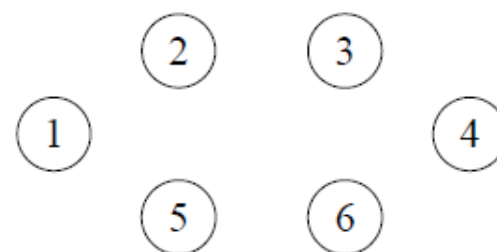
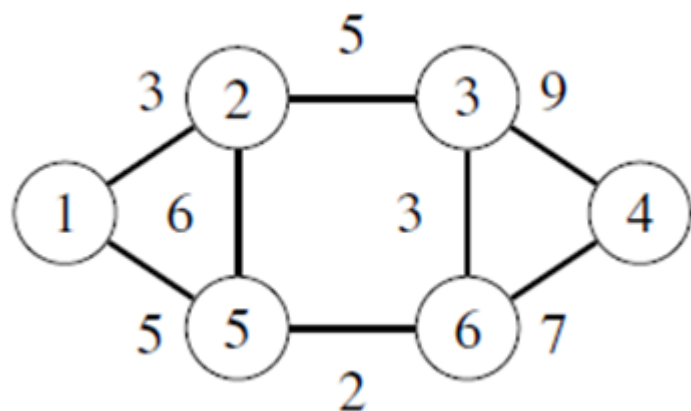


0-1 3-4

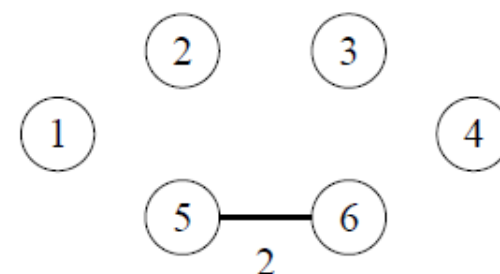


4-5 4-7

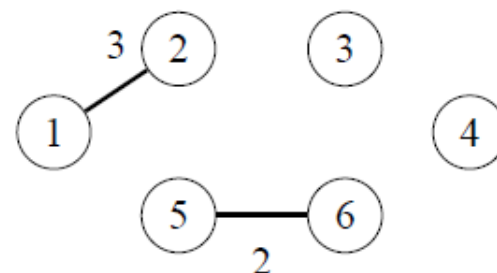
Друг пример,
започвайки от граф,
съставен само от
изолираните върхове на
оригиналния граф:



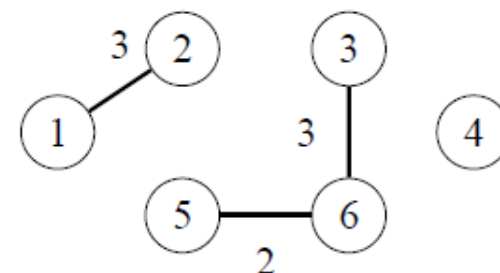
step 1



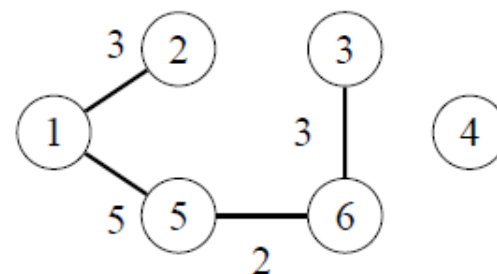
step 2



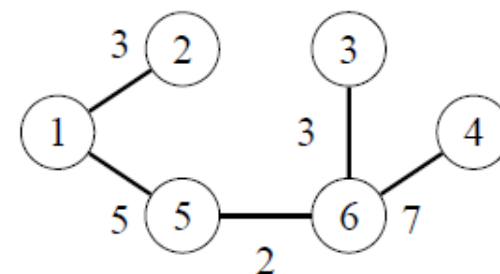
step 3



step 4



step 5



step 6

Ще използваме алгоритъм от структурата данни наречена
„Система от непресичащи се множества“ – Disjoint Set
(Union-Find Algorithm)

Пример: дадени са обекти, например върхове на граф

0 1 2 3 4 5 6 7 8 9

По време на работата се образуват непресичащи се множества от
върхове на свързани подграфи, например:

0 1 2-3-9 5-6 4-8

Използваме две операции: Find и Union

Операция Find – дава отговор на въпроса дали два обекта са в едно и също множество (например Find(2,9): дали 2 и 9 са такива)

0 1 2-3-9 5-6 4-8

Операция Union – променя системата от множества, като обединява (merge) две множества, всяко определено с по един свой елемент. Например Union(3,8):

0 1 2-3-9-4-8 5-6

Използваме два масива $p[]$ и $h[]$. Поддържа се гора и за всеки елемент i стойността $p[i]$ е номер на предшественика му в дървото. За върха i на дървото (лидерът), $p[i] = -1$. Стойността на $h[i]$ е дълбочината на дървото за лидера i .

Реализация Крускал

```
int V; vector<vector<int>> e;
vector<int> p, h;

int find(int i) // light implementation
{ if (p[i] == -1) return i;
  return p[i] = find(p[i]);}

void unite(int x, int y)
{
  int s1 = find(x); int s2 = find(y);
  if (s1 != s2)
  { if (h[s1] < h[s2]) p[s1] = s2;
    else if (h[s1] > h[s2]) p[s2] = s1;
    else {p[s2] = s1; h[s1] += 1;}
  }
}
```



```

void show()
{
    cout << "r.size()=" << e.size() << endl;
    for(auto a : e)
        cout<<a[0]<<" "<<a[1]<<" "<<a[2]<<endl;
    cout << endl;
}

void kruskal()
{
    sort(e.begin(), e.end());
    // Initialize UF:
    p.resize(V,-1); h.resize(V,1);
    int ans = 0;
    //cout << "Edges in the constructed MST:"<< endl;

```

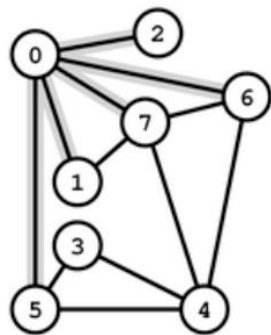
```

for (auto a : e)
{int w = a[0]; int x = a[1]; int y = a[2];
  // Take the edge if it does not form a cycle
  if (find(x) != find(y))
  {
    unite(x, y);
    ans += w;
    cout << x << " -- " << y << " : " << w << endl;
  }
}
cout << "Cost of the MST " << ans;
}

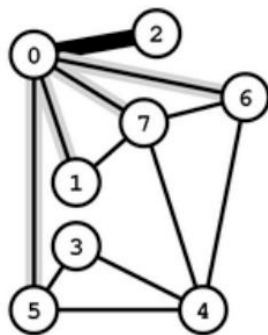
```

```
int main()
{
    cin >> V;
    int m; cin >> m;
    for(int i=0;i<m;i++)
    {
        int x,y,w; cin >> x >> y >> w;
        e.push_back({w,x,y});
    }
    show();
    kruskal();
}
```

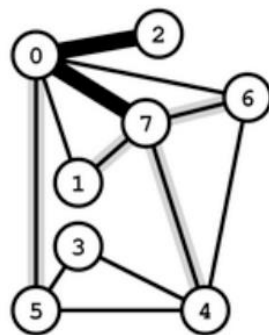
Алгоритъм на Прим - идея: започваме от връх 0 и строим дърво (теглата на ребрата са геометричните им дължини).



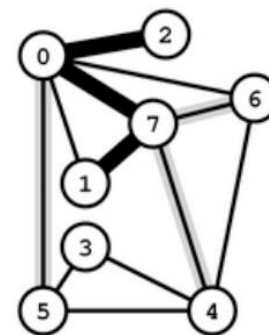
0-2 0-7 0-1 0-6 0-5



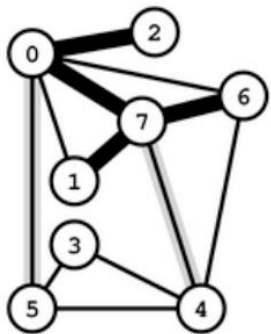
0-7 0-1 0-6 0-5



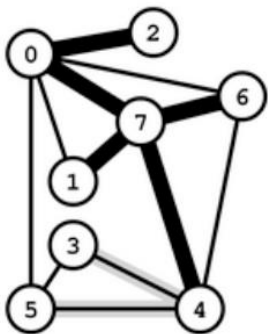
7-1 7-6 7-4 0-5



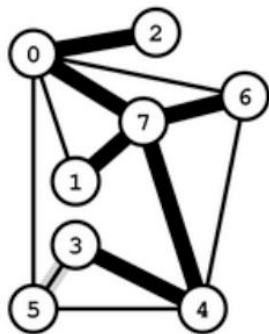
7-6 7-4 0-5



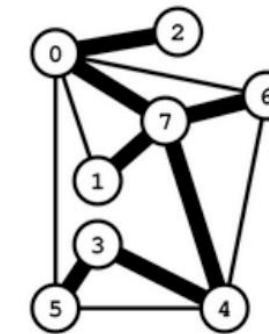
7-4 0-5



4-3 4-5



3-5



Благодаря за вниманието!