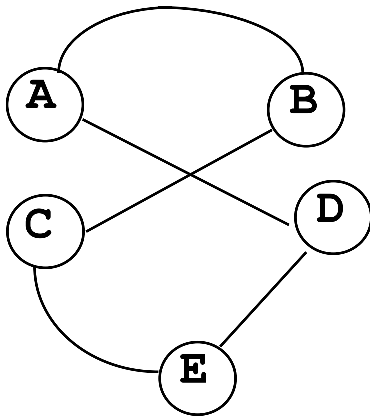
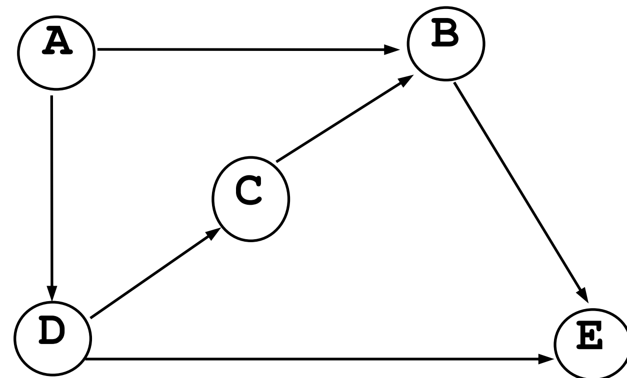
ГРАФ (МРЕЖОВИ СТРУКТУРИ)

Структурата граф подобно на дървото се състои от непразно множество върхове (възли) **V** и множество дъги (ребра) **E**.
Графът е подмножество от декартовото произведение на двете множества - **V** и **E**: **G = {V, E}**
За разлика от дървото един в граф възможно е да има обратни връзки.
Всяка дъга се дефинира чрез двойка възли, които съединява, например **(A, F)** или **(A, B)**. Ако двойката е подредена, т.е. дъгата има посока, графът се нарича ***насочен*** или ***ориентиран***, в противен случай графът е ***неориентиран*** **:**





Една дъга **X**, която се описва като **X = (A,B)**, се нарича ***инцидентна*** с върховете **A** и **B**, т.е. върховете, които съединява или от които влиза и излиза. Върховете **A** се **B**, свързани с една дъга, наричат ***съседни***. Две и повече дъги, излизащи от един връх, например **(A,B)** и **(A,D)**, също така се наричат ***инцидентни***. ***Степен на връх*** на граф **G** е броят на ребрата, инцидентни с него. Означава се с $d(V_i)$. Например $d(A)=2$. Ако $d(V_i)=0$, върхът V_i се нарича ***изолиран***. За ориентиран граф, в който всеки възел V_i може да има влизащи и излизащи дъги, $d(V_i) = d'(V_i) + d''(V_i)$, където $d'(V_i)$ е ***полустепен на входа*** на V_i, $d''(V_i)$ е ***полустепен на изхода*** V_i.

Път между върховете V_i и V_j е последователността от дъгите, които ги свързват. ***Дължината*** на пътя е броят на тези дъги. Пътят се нарича ***прост***, ако върховете в дъгите му не се повтарят. ***Цикълът*** е прост път, в който първия и последния върхове се повтарят.

Цикличен граф е граф **G**, който съдържа поне един цикъл. Графът **G** се нарича ***тегловен***, ако дъгите му имат стойности. Графът **G** се нарича свързан, ако за произволна двойка върхове V_i и V_j съществува път, който ги свързва. В противен случай графът е ***несвързан***. В един несвързан граф може да има ***свързани компоненти***, т.е. подграфи, които са свързани.

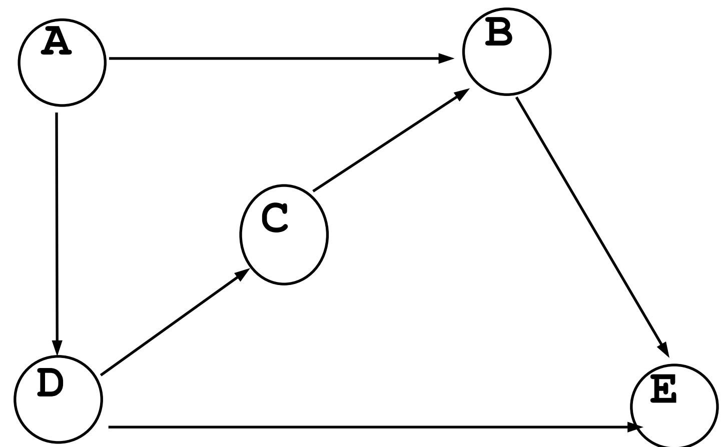
Съществуват различни начини за представянето на структурата граф в паметта.

Статични методи

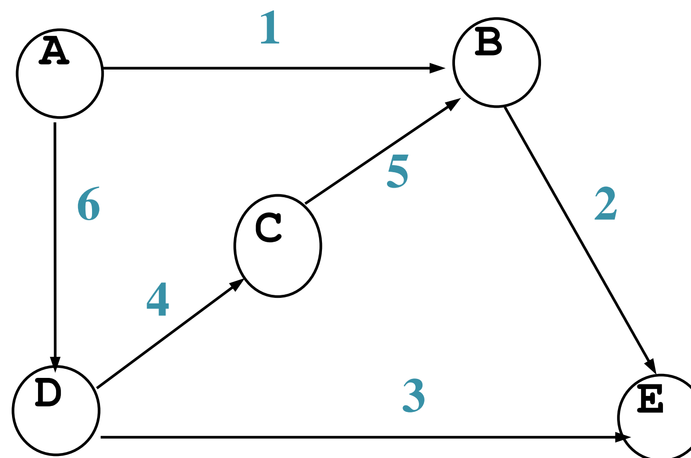
Начините са ефективни, когато представяният граф има голям брой дъги и резултантните матрици не са разредени.

- чрез матрица на съседство

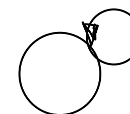
	A	B	C	D	E
A	-	1	0	1	0
B	0	-	0	0	1
C	0	1	-	0	0
D	0	0	1	-	1
E	0	0	0	0	-



- чрез матрица на инцидентност:



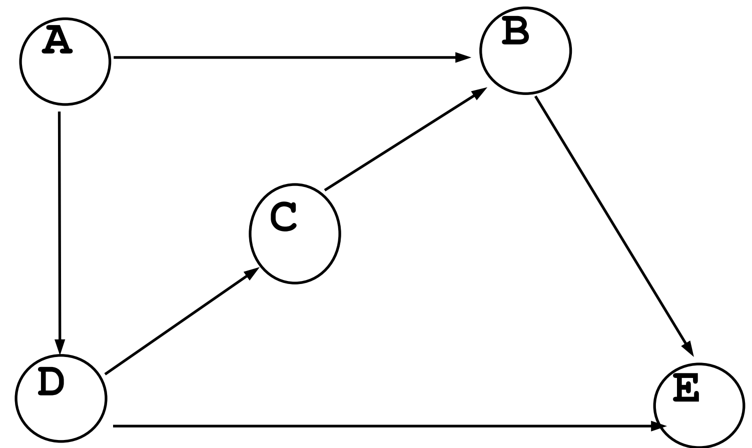
	1	*2*	*3*	*4*	*5*	*6*
A	1	0	0	0	0	1
B	2	1	0	0	2	0
C	0	0	0	2	1	0
D	0	0	1	1	0	2
E	0	2	2	0	0	0



дъга-примка

- чрез вектори на съседство:

A	B	D
B	E	0
C	B	0
D	C	E
E	0	0

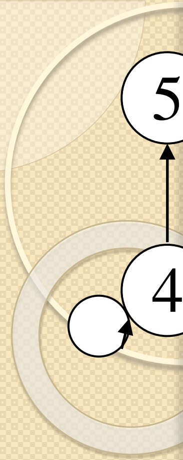


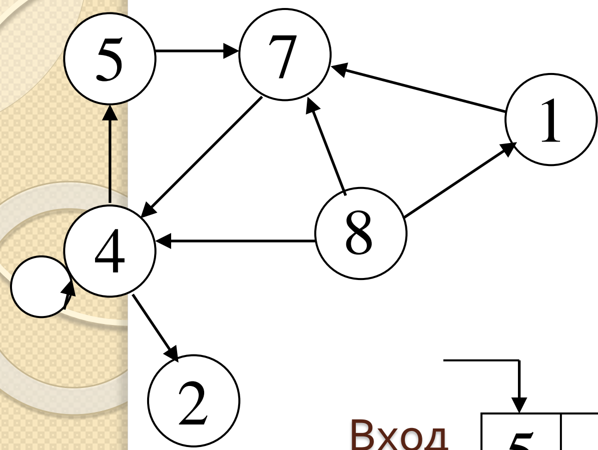
Динамично представяне на граф

Графът може да се представи динамично. Примерната декларация на структурата е следната:

```
struct nd               //структура за описание на връх
{int key; nd*p1; arc*p2;}
                        //p1 - указател към следващия връх
      //p2 - указател към списъка от наследници -
      //всички възли, до които има дъга от даден възел

struct arc              //структура за описване на дъга
{arc*p3; nd*p4;}
//p3 - указател към следващата дъга за даден възел
//p4 - указател към възел, крайния за дъгата
```





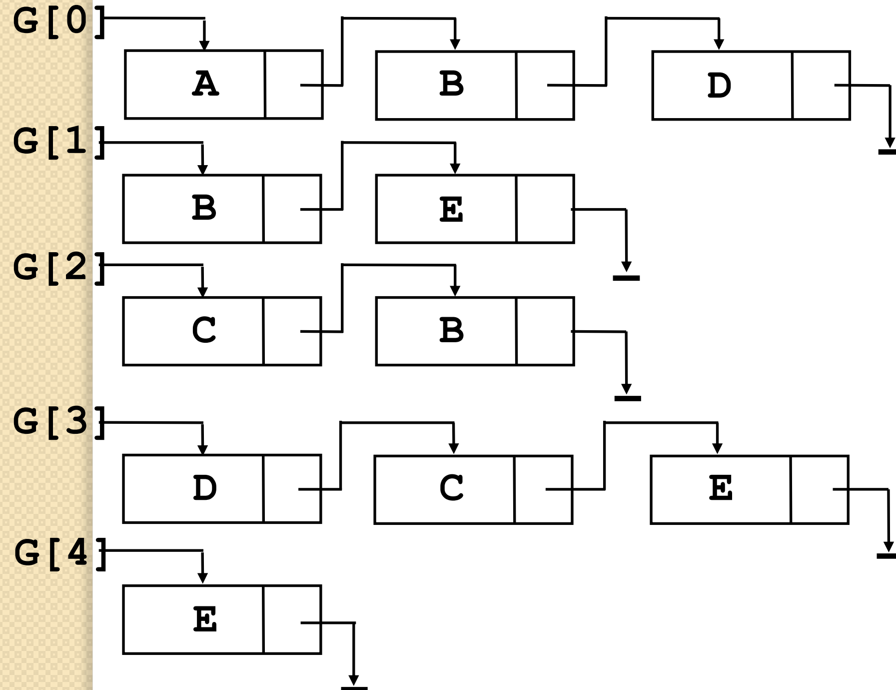
Вход

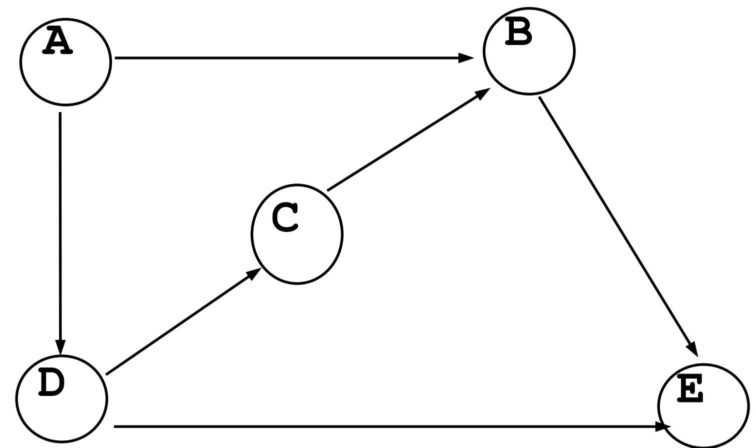
5
7
8
1
4
2

Комбинирано представяне на структурата граф чрез списъци на съседство

Структурата се представя като масив от указатели към списъци, описващи върховете и инцидентните им дъги в графа:

```
const n=10;
struct link
{char key; link *next;} *G[n];
```





Основни операции със структурата граф

- Инициализиране на граф
- Търсене на връх
- Търсене на дъга
- Включване на връх
- Включване на дъга
- Премахване на връх
- Премахване на дъга
- Обхождане на граф

Изброените операции са дефинирани при следните аксиоми:

1. Всеки нов добавян връх е изолиран връх
2. Ако се включва дъга и върховете, свързани с нея, не принадлежат на графа, те се добавят
3. Ако се премахва дъга, то върховете, свързани с нея, не се премахват
4. Ако се премахва връх, то се премахват всички дъги, инцидентни с него
5. Включването на нов връх V_i в граф G е възможно, ако V_i не принадлежи на G.
6. Включването на нова дъга (V_i, V_j) в граф G е възможно, ако тя не принадлежи на G.
7. Премахването на връх V_i от графа G е възможно, ако V_i принадлежи на G.
8. Премахването на дъга (V_i, V_j) от графа G е възможно, ако (V_i, V_j) принадлежи на G.

Ако графът G е неориентиран, трябва да се добави още една аксиома:

9. Ако дъга (V_i, V_j) принадлежи на графа G, то дъгата (V_j, V_i) също така е дъга на графа G.

- Инициализация на граф:

```
void init (link *gr[n])
{
  for (int i=0; i<n;i++)
    gr[i]=NULL;
}
```

- Търсене на връх в граф:

```
int search_node(link *gr[n], char c)
{
  int flag=0;
  for (int i=0;i<n;i++)
   if (gr[i])   //проверка, дали даден връх съществува
       if (gr[i]->key==c)
           flag=1;
  return flag;
}
```

- Търсене на дъга в граф

```
int search_arc(link *gr[5], char c1, char c2)
//c1 и c2 - ключовите стойности на възлите, които
//свързва търсената дъга
{
   int flag=0;
   if (search_node(gr, c1)&&search_node(gr, c2))
    { int i=0;
      link *p;
      do
      {
       if ((gr[i]==NULL)||(gr[i] && gr[i]->key!=c1))
                        i++;
      }
      while (gr[i]->key!=c1);
      p=gr[i];
      while (p->key!=c2&&p->next!=NULL)  p=p->next;
      if (p->key==c2) flag=1;
    }

    return flag;
}
```

- Включване на връх в граф.

```
void add_node(link *gr[n], char c)//c е добавяната
стойност
{
  if (search_node(gr,c))
   {cout<<"\nВърхът вече съществува!\n";  }
   else
   {
    int j=0;
    while (gr[j]&&(j<n))  j++;
    if (gr[j]==NULL)
     {gr[j]=new link;       //създаване на нов връх
       gr[j]->key=c;//установяване на ключовата стойност
         gr[j]->next=NULL;  //и указателя
     }
       else {cout<<"\nПрепълване на структурата!\n"; }
   }
}
```

- Включване на дъга в граф

```
void add_arc(link *gr[n], char c1, char c2)
{
   int i=0;
   link *p;
   if (search_arc(gr, c1, c2))
        {cout<<"\nДъгата вече съществува!";  }
      else
    {
        if (!(search_node(gr, c1)))  add_node(gr, c1);
        if (!(search_node(gr, c2)))  add_node(gr, c2);
        while (gr[i]->key!=c1)  i++;
        p=new link;      //създаване на нов елемент
        p->key=c2;       //в списъка на съседство
        p->next=gr[i]->next;
        gr[i]->next=p;
    }
}
```

- Премахване на връх от граф

```
void del_node(link *gr[n], char c)
{
  if (search_node(gr, c))
    {
      int i=0;
      do
         if ((gr[i]==NULL)||(gr[i] && gr[i]->key!=c)) i++;
      while (gr[i]->key!=c);
      link *p=gr[i], *q;
      while (gr[i]!=NULL)
          {p=gr[i]; gr[i]=p->next; delete p;}
     //изтриване на върха и на дъгите, излизащи от него
      for (i=0;i<n;i++)
       if (gr[i])
          {
           p=gr[i];
           while ((p->key!=c)&&(p->next!=NULL))
            {q=p; p=p->next;}
           if (p->key==c)  //изтриване на дъгите, влизащи
            {q->next=p->next; delete p;}      //във върха
          }
    }
       else {cout<<"В графа няма такъв връх!";}
}
```

- Премахване на дъга от граф

```
void del_arc(link *gr[n], char c1, char c2)
{
  if (search_arc(gr, c1, c2))
    {
      int i=0;
      while (gr[i]->key!=c1)
                    i++;
      link *p=gr[i], *q;
      while (p->key!=c2)
              {q=p; p=p->next;}
      q->next=p->next;
      delete p;            //премахване на върха
                           //от списъка на съседство
    }
      else {cout<<"\nВ графа няма такава дъга!";}
}
```

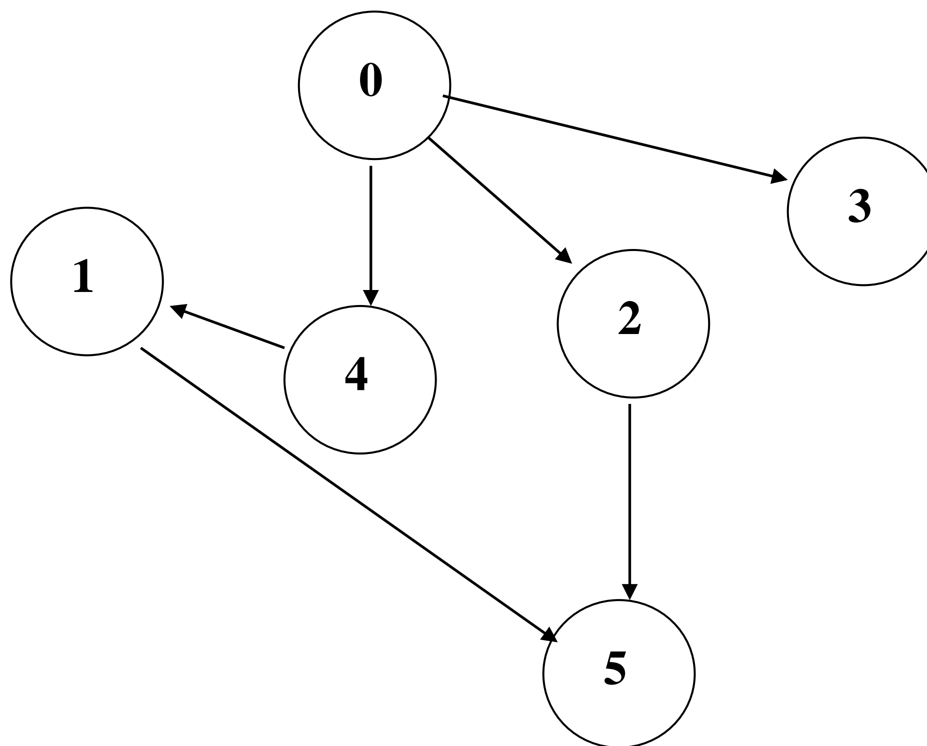
- Обхождане на граф

Обхождането на граф може да става по два начина:

- *в дълбочина* (*Depth-First Search* или *DFS*), при което се следва възможно най-дългия път, като се обработват всички срещнати по пътя върхове до достигане до връх без наследник или до връх, през който вече сме минали. С други думи, при обхождане в дълбочина се посещава един съсед $\mathbf{V_i}$ на даден връх $\mathbf{V_j}$, а останалите съседи на $\mathbf{V_j}$ ще бъдат посетени едва след като са обходени всички съседи на $\mathbf{V_i}$.

- *в ширина* (*Breadth-First Search* и *BFS*), при което върховете се обхождат по реда на нарастване на разстоянието от началния връх. С други думи, първо се посещават всички непосредствени съседи на даден връх, преди да се посетят техните съседи.

Примерно обхождане:



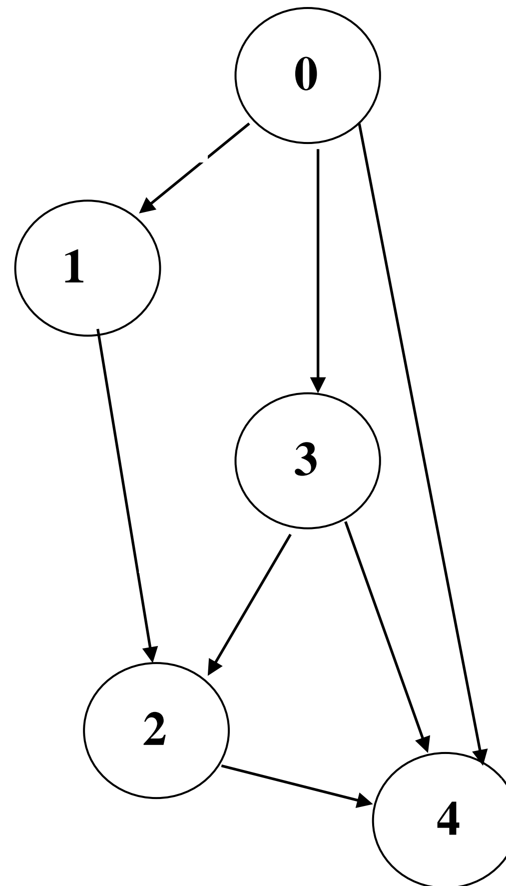
DFS (в дълбочина): 0 4 1 5 3 2

BFS (в ширина): 0 4 2 3 1 5

Друг пример за обхождане на граф:

DFS: 0 1 2 4 3
0 3 2 4 1
0 3 4 1 2
. . .

BFS: 0 1 3 4 2



Описание на рекурсивен алгоритъм за DFS:

```
const int n=20;
int m[n];       //масив за маркиране на обходените върхове
int prof (i)    //рекурсивна част
{ m[i]=1;       //възелът се маркира като посетен
   cout<<i<<"  "; //извеждане на номера на обходения връх
   for (int j=0; j<k; j++)//k - е броят на съседите на върха i
      {  v=j-ия наследник на i;
         if (m[v]==0)  prof(v);
      }
}
void dfs()
{ for (int i=0; i<n; i++)
    if (m[i]==0)
       prof(i);
}
```

В описанието на алгоритъма е използвано статичното представяне на графа **G** - чрез матрицата на съседство. Възлите на графа са номерирани последователно от **1** до **N**. Сложността на алгоритъма е **O(N²)**.

Ако графът **G** е представен чрез списъците на съседство, сложността на алгоритъма е **O(N*max (N, M))**, където **M** е броя на дъгите в графа. Възможен е също така и итеративен вариант на алгоритъма, реализиран с използване на помощен стек за запомняне на обходените върхове.

Реализация на DFS за комбинирано представяне на графа:

```
void dfs(int k)        //k - ключовата стойност
                       //на посетения възел
{
   cout<<k;
   int j=convert[k];  //convert[k] - е функция, която
                      //връща индекса на елемента на
                      //масива от списъците на
                      //съседство, чиято стойност е k
   m[j]=1;            //възелът се маркира като посетен
   for (link*t=gr[j]->next; t->next!=NULL; t=t->next)
       if (!m[convert(t->key)])
            dfs(t->key);
}
```

Описание на итеративен алгоритъм за BFS за статично представяне на граф:

```
void bfs(int v) //v - начален възел
{ инициализация на помощен масив m, използван за
  регистрация  на обходените възли;
  инициализация на помощна опашка;
  m[v]=1;     //маркиране на v като посетен
  cout<<v;
  push(v); //поместване на v опашката
  while (опашката не е празна)
   {pop(x);//извличане на поредния елемент от
           //опашката
       for (всеки връх y, съседен на x)
        {
         if(m[y]==0)
           {cout<<y; m[y]=1; push(y);}
        }
   }
}
```

Реализация на BFS за комбинирано представяне на графа:

```
void bfs (int k)    //k - ключовата стойност на началния възел
{ int m[n];         //масив за регистриране на обходените върхове
  init_queue();     //инициализация на помощната опашка
  push_queue(k);    //поместване в опашка на първия елемент
  while (empty_queue())   //докато опашката не е празна
  { k=pop_queue();    //извличане на поредния елемент от опашката
    int j=convert(k);//функция, която връща индекса
                     //на елемента на масива от списъците на
                       // съседство, чиято стойност е k.
    if (m[j]==0)         //Възелът не е посетен
   {m[j]=1;cout<<k<<" ";}//регистриране и визуализация на възела
    for (link *t=gr[j]->next; t->next!=NULL; t=t->next)
       if(m[convert(t->key)]==0) //възелът не е посетен
         push_queue(t->key);  //включване на възела в опашката
  }
}
```

Сложността на алгоритъма е същата, както и на DFS.

Описаните алгоритми позволяват да се извърши обхождане на свързан граф **G**. Ако **G** е несвързан, обхождането се извършва на части поотделно за всяка от свързаните компоненти.

Търсене на път в граф

Една от най-често решавани задачи при работа с тази структура е *търсене на най-кратки пътища*. Следва разглеждане на няколко класически алгоритъма за търсене на път в граф.

1. *Алгоритъм на DIJKSTRA*

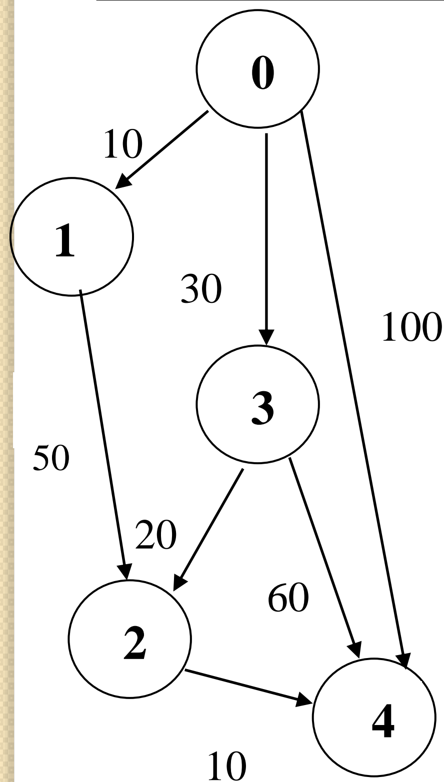
Алгоритъмът на DIJKSTRA при зададен начален връх решава задачата за намиране на най-късия път до произволен връх в графа.

Имаме тегловен ориентиран граф **G={V,E}**, където всяка дъга **E** има <u>неотрицателно</u> тегло. Върховете са последователно номерирани - **V={0, 1, 2, ..., N-1}**. Един от върховете, например нулевия, е избран за начален. Целта е да се намери най-късия път от началния връх до всеки друг връх от **V**.

Алгоритъм на DIJKSTRA работи чрез поддържане на комплект от върховете, да го наречем множество **S**, най-късите разстояния до които от началния връх са известни. Първоначално множеството **S** съдържа само началния връх. На всяка стъпка към **S** се добавя нов връх **W**, пътя до който от началния връх е с най-низка цена. Този път минава само чрез върховете от **S** и се нарича "специален". Масивът **D** се използва за запомняне на специалните пътища. Когато **S** включи всички върхове от **G, D** ще съдържа най-късите пътища от началния връх до всеки друг връх на **G**.

Пример:

Стъпка	S	w	D[1]	D[2]	D[3]	D[4]
0 (init)	0	-	10 (0,1)	-	30 (0,3)	100 (0,4)
1	0, 1	1	10 (0,1)	60 (0,1,2)	30 (0,3)	100 (0,4)
2	0, 1, 3	3	10 (0,1)	50 (0,3,2)	30 (0,3)	90 (0,3,4)
3	0, 1, 2, 3	2	10 (0,1)	50 (0,3,2)	30 (0,3)	60 (0,3,2,4)
4	0,1,2,3,4	4	10 (0,1)	50 (0,3,2)	30 (0,3)	60 (0,3,2,4)



P - помощен масив, съдържащ върховете, предхождащи текущите в специалния път:

Път от 0 към 4:

4 ->2 ->3 ->0

P[0]=*

P[1]=*

P[2]=3

P[3]=0

P[4]=2

Описание на алгоритъма на DIJKSTRA

```
dijkstra()
{ int s[n]={0};    //множеството S е реализирано като масив
                   //n - брой възли в графа
   int d[n];       // масив, в който се съхраняват
                   // най-кратките пътища от началния
                   //(0-вия) възел до всички останали
   int p[n]        // p е помощен масив, съдържащ върховете,
                   // предхождащи текущите в специалния път
   int w, i;
   s[0]=1;
   for (i=1; i<n; i++)
      { d[i]=c[0,i]; p[i]=0; } //c е ценовата матрица с
                               //разстоянията
   for (i=1; i<n; i++)
      { избиране на връх w, за който d[w] е min;
         s[w]=1;            //добавяне на w към множеството
         for (i=1; i<n; i++)
           if (s[i]==0)
               if (d[i]>d[w]+c[w,i])
                    {
                       p[i]=w;
                       d[i]=d[w]+c[w,i];
                    }
      }
}
```

При използване на статичното представяне на графа G сложността на алгоритъма на DIJKSTRA е **O(N²)**. Ако броят на дъгите в графа е малък и матрицата на съседство е разредена, по-добър резултат дава представянето на графа чрез списъците на съседство. В този случай сложността на алгоритъма на DIJKSTRA е **$O(M+N * Log_2(N))$**, където **M** е броя на дъгите в **G**.

2. *Алгоритъм на FLOYD*

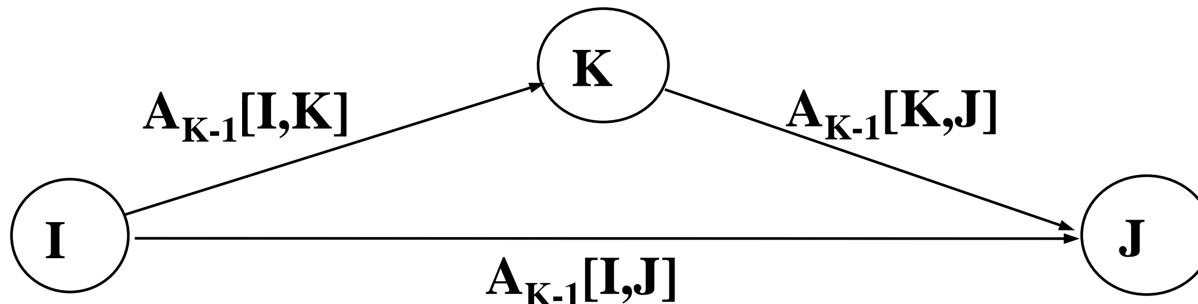
Алгоритъмът на FLOYD решава задачата за намиране на минимален път между всяка двойка върхове в даден граф.

Имаме граф **G={V, E}**, в който всяка дъга **(V_i, V_j)** има неотрицателна цена **C [V_i, V_j]**. Задачата се свежда до намирането на най-къс път от **V_i** до **V_j** за всяка двойка върхове **(V_i, V_j)**.

Нека за по-голямо удобство пак да предположим, че върховете са номерирани последователно от **0** до **N-1**. За пресмятане на дължините на най-късите пътища ще се използва помощна матрица **A** с размерност **N x N**, която се инициализира като **A[I, J]=C[I, J]** за **I** и **J** от **1** до **N**. Ако няма път от **I** до **J**, **A[I, J]** се установява в някаква голяма стойност. Диагоналните елементи на **A** се приемат за **0**.

Матрицата **A** се променя итеративно, като след **K**-та итерация **A[I, J]** ще съдържа **MIN** път от всички възможни пътища от **I** до **J**, които минават през върховете, номерирани не по-високо от **K**:

$$A_K[I, J] = MIN(A_{K-1}[I, J], A_{K-1}[I, K] + A_{K-1}[K, J])$$



Примерен граф:

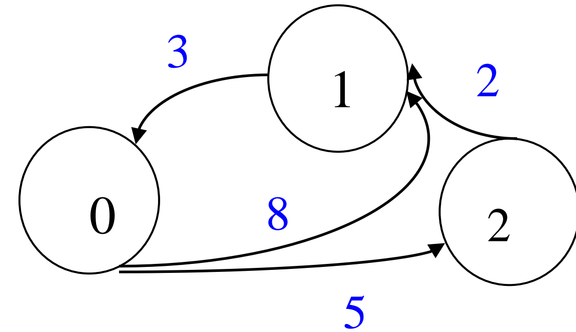
Инициализация:

	0	1	2
0	0	8	5
1	3	0	-
2	-	2	0

A[0]

	0	1	2
0	0	8	5
1	3	0	8
2	-	2	0

през 0 (3+5)



A[1]

	0	1	2
0	0	8	5
1	3	0	8
2	5	2	0

през 1 (2+3)

A[2]

през 2 (5+2)

	0	1	2
0	0	7	5
1	3	0	8
2	5	2	0

P

	0	1	2
0	0	3	0
1	0	0	1
2	2	0	0

За да се видят пътищата, използваме помощната матрица **p**, елементите на която **p[i],[j]** съдържат върховете, включвани в MIN пътища. Ако **p[i],[j]=0**, това означава, че пътят е директен, т.е. съществува дъга **(i,j).**

Описание на алгоритъма на FLOYD

```
 void floyd()
{for (int i=0; i<n; i++)
     for (int j=0; j<n; j++)
        { a[i][j]=c[i][j]; // инициализация на a
          p[i][j]=0;       // инициализация на p
        }
  for (int k=0; k<n; k++)
     for (i=0; i<n; i++)
       for (j=0; j<n; j++)
           if ((a[i][k]+a[k][j])<a[i][j])
                {  a[i][j]= a[i][k]+a[k][j];
                   p[i][j]=k;
                 }
}
```

Сложността на алгоритъма е **O(N^3)**, тъй като има три цикъла, вложени един в друг. Ако го сравним с алгоритъма на DIJKSTRA, която е **O(N^2)** за един връх, то при повторение на действието **N** пъти ще получим същия резултат - **O(N^3)**.

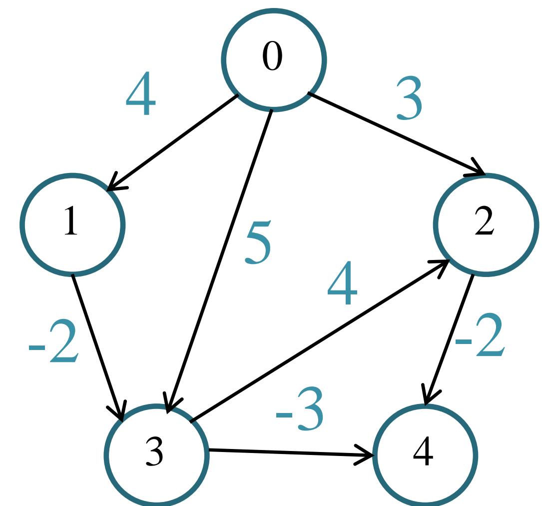
От най-известните алгоритми за търсене на пътища в граф следва също така да се спомене алгоритмът на FORD-BELLMAN, който, подобно на алгоритъма на DIJKSTRA намира най-кратки разстояния от даден връх **S** до останалите върхове в графа **G={V, E}**. За разлика от алгоритъма на DIJKSTRA, алгоритмът на FORD-BELLMAN не налага ограничения на стойностите на дъгите, които могат да бъдат и отрицателни. Сложността на алгоритъма също така се оценява като **O(N^3)**.

Алгоритъм на FORD-BELLMAN

Ценова матрица на Граф **G**

Ориентиран тегловен граф **G**

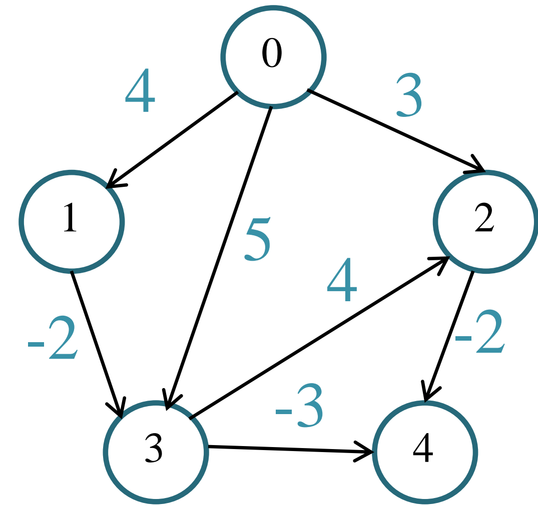
	0	1	2	3	4
0	∞	**4**	**3**	**5**	∞
1	∞	∞	∞	**-2**	∞
2	∞	∞	∞	∞	**-2**
3	∞	∞	**4**	∞	**-3**
4	∞	∞	∞	∞	∞



Алгоритмът търси най-кратките пътища от даден начален връх до всички останали върхове в графа.

Масив D с разстоянията между възел 0 и останалите възли

	D[1]	D[2]	D[3]	D[4]
0	**4**	**3**	**5**	∞
1	**4**	**3**	**2**	∞
2	**4**	**3**	**2**	**1**
3	**4**	**3**	**2**	**-1**
4	**4**	**3**	**2**	**-1**



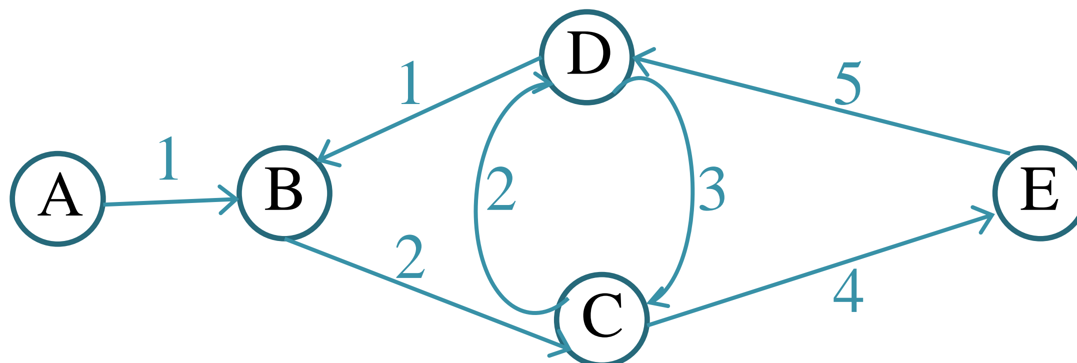
За всеки възел от V:
ако D[i]>D[j]+A[j][i]
то
D[i]=D[j]+A[j][i]

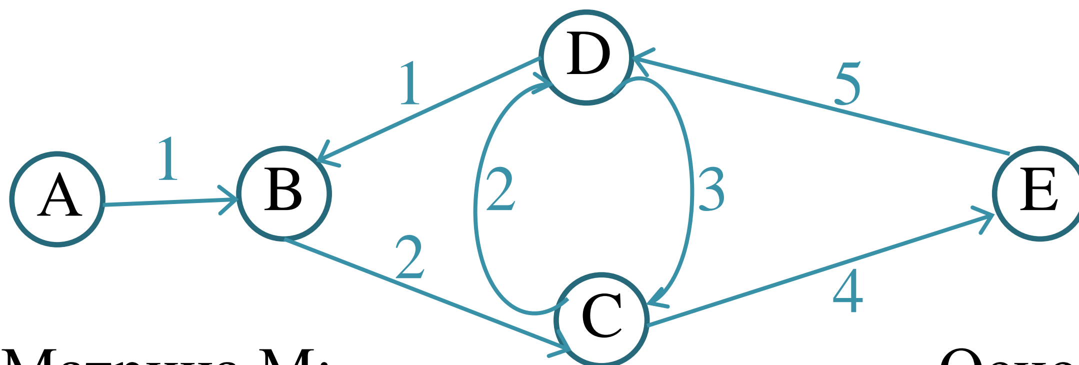
```
for (k=1;  k<=n-2;  k++)
    for(i=0;  i<n;  i++)
        for(j=0;  j<n;  j++)
            if (D[i]>D[j]+A[j][i])
                D[i]=D[j]+A[j][i])
```

Център на графа

Център (централен връх) на графа **G (V, E)** е върхът **w** $\in$ **V** с минимален *ексцентриситет* (максимално отдалечаване), т.е. максималното разстояние до останалите върхове е минимално:

min(максимална дължина на пътя от връх w до връх $v_i \in V$*)*
за $w \in V$





Матрица M:

	A	**B**	**C**	**D**	**E**
A	0	1	3	5	7
B	∞	0	2	4	6
C	∞	3	0	2	4
D	∞	1	3	0	7
E	∞	6	8	5	0
EXT (max)	∞	**6**	**8**	**5**	**7**

Основни стъпки:

1. Прилагане на алгоритма на Floyd за пресмятане на матрицата M с най-кратките пътища в G ($O(n^3)$).
2. Търсене на ***max*** във всеки стълб на A ($O(n^2)$).
3. Търсене на ***min*** ексцентриситет ($O(n)$).

Резултат: Центърът на граф G е връх **D.**

Сложност: $O(n^3)$

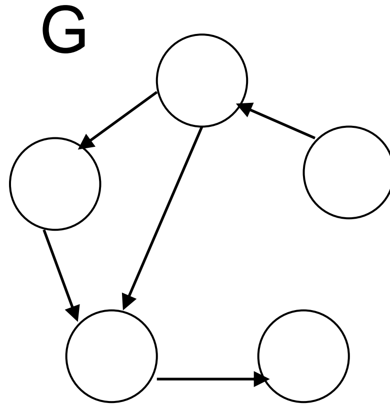
Транзитивно затваряне на граф. Алгоритъм на Уоршал

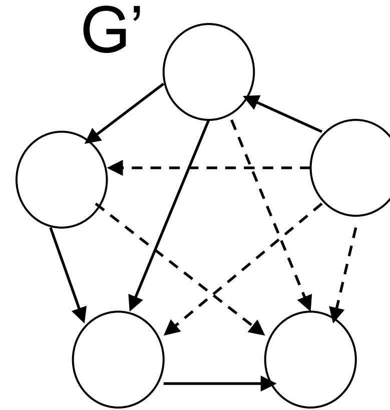
Даден е ориентиран граф G(V,E), представен чрез матрица на съседство A[ ][ ]. Търсим матрица A'[ ][ ], такава, че:

- A'[i][j]==1, тогава и само тогава, когато съществува път (с каквато и да е дължина) между върховете i и j.
- A'[i][j]==0, когато път между двата върха не съществува.

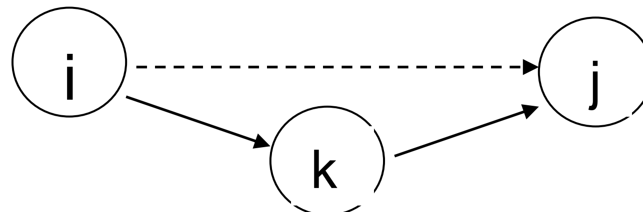
Алгоритъмът за решаване на тази задача носи името на Уоршал и резултатът от прилагането му – матрицата A'[ ][ ], се нарича *матрица на достижимост* на графа. Графът G'(V, E'), съответен на матрицата A'[ ][ ] (ако я разглеждаме като матрица на съседство), се нарича *транзитивно затворен граф* на G.

Например:





Алгоритъмът на Уоршал има сложност $O(n^3)$ и процедира по следния начин: за всеки три върха k, i, j принадлежащи на V, ако (i,k) принадлежи на E и (k,j) принадлежи на E, то добавяме в множеството от дъги на графа G’ и дъгата (i,j):



Разглеждането на тройки върхове се извършва чрез три вложени цикъла по следния начин:

```
…
// Описание на алгоритъма на Уоршал:
for (k=0; k<n; k++)
    for (i=0; i<n; i++)
        if (a[i][k])
            for (j=0; j<n; j++)
                if (a[k][j])    a[i][j]=1;
…
```

След изпълнението на програмния фрагмент матрицата A[ ] [ ] ще бъде модифицирана до търсената матрица на достижимост A'[ ] [ ], отговаряща на новия граф G'.

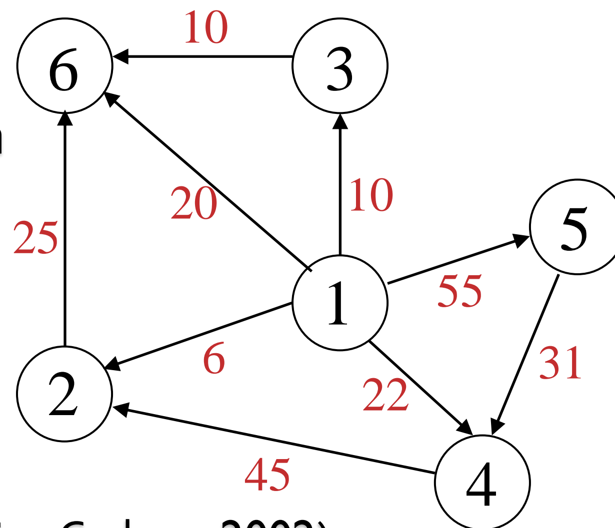
Контрол на компании

Задача: Даден е тегловен граф *G(V,E)* с тегла на ребрата естествени числа между 0 и 100. Казваме, че върхът i контролира j, ако съществува реброто *(i, j)* със стойност >50 или съществува множество от върхове $v_1, v_2, \ldots, v_k$, контролирани от i, ако съществуват ребрата (v_1, j) $(v_2, j), \ldots, (v_k, j)$, чиято сума е >50. Задачата е по зададен връх *m* да се намерят всички контролирани от *m* върхове.

Задачата описва реална ситуация, в която върховете на графа представят компании, а всяко ребро *(i, j)* показва процента от акциите, които компанията *i* притежава от компанията *j*.

Например: компания 1 контролира компании 2, 4 и 5.

Задачата може да се реши чрез адаптация на алгоритъма на Уоршал за транзитивно затваряне.

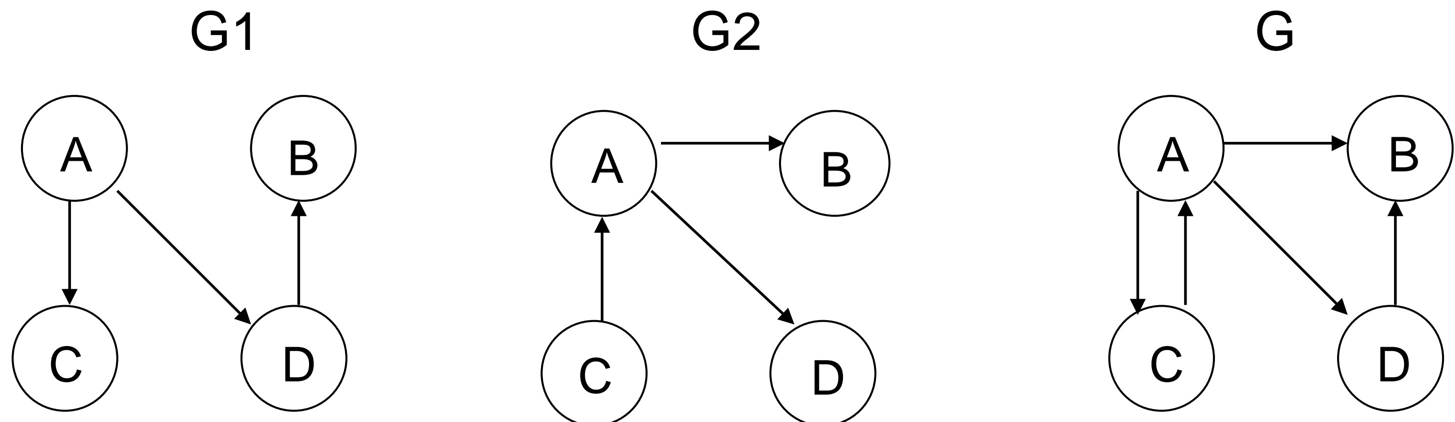


(Преслав Наков, Панайот Добриков, *Програмиране = ++Алгоритми*, TopTeam Co., София, 2002)

Суперпозиция на два графа

Ако два графа G1(V1,E1), G2(V2,E2) са такива, че {V1} = {V2}, то суперпозицията им представлява нов граф G(V,E), в който множеството върхове {V}={V1}={V2},
а множеството дъги {E}={E1}+{E2}

Например:



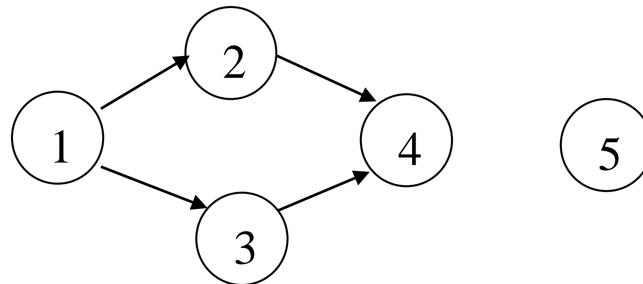
Идеята на алгоритъма се състои в следното: новият граф G се създава чрез копиране на един от двата графа, след което към G се добавят всички дъги от втория граф.

Топологично сортиране на ацикличен граф

Топологично сортиране на ацикличен граф G(V,E) се нарича линейно нареден списък Z от върховете му, за който е изпълнено следното: за всеки два върха i, j принадлежащи на V, ако съществува път от i до j, върхът i трябва да предшества j в Z.

На практика списъкът Z рядко може да се построи по единствен начин.

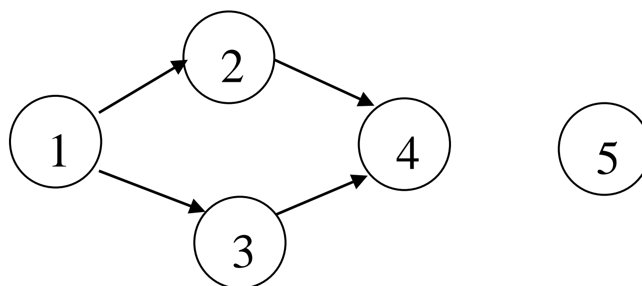
Например:



Възможни са следните варианти на подредбата:
Z=(1,2,3,4,5)
Z=(1,3,2,4,5)
Z=(5,1,2,3,4) и т. н.

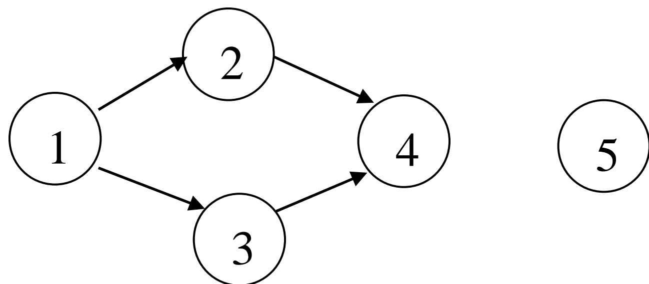
Т.к. сортираният граф е ацикличен, то в него съществува поне един връх без предшественици. Именно такъв връх е първи в наредбата. Алгоритъмът може да се опише така:

1. Инициализираме ***Z*** като празен списък.
2. Избираме връх ***i*** без предшественици и го добавяме в списъка ***Z***. Изключваме ***i*** от графа, както и всички дъги, инцидентни с него. Ако има повече от един връх без предшественици, избираме произволен.
3. Повтаряме стъпка 2, докато не остане нито един връх.



Алгоритъмът може да бъде реализиран и по обратния начин: на всяка стъпка да се търси връх без наследник. Така ще се получи топологичното сортиране на графа в обратен ред.

Когато съществува повече от едно нареждане на върховете в ***Z***, множеството от всички възможни списъци ***Z*** се нарича пълно топологично сортиране.



Реализациите на описаните алгоритми могат да се намерят в Преслав Наков, Панайот Добриков. *Програмиране = ++Алгоритми*, TopTeam Co., София, 2002

Най-дълъг път в ацикличен граф

Много практически задачи се решават, като се сведат до задача за намиране на най-дълъг път в ацикличен граф. Условието за ацикличност е съществено, т.к. единственият подход за решаване на задачата в общия случай е чрез пълно изчерпване (когато в графа няма цикли, са изпълнени някои принципи за оптималност, необходими за задачи, които се решават с динамично оптимиране, което гарантира по-ефективно решение).

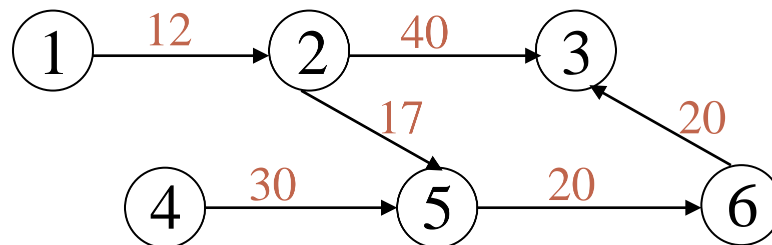
Например: Екип от програмисти разработва програмен продукт, който се състои от отделни задачи. Всяка задача има определена продължителност и свързва два етапа от разработването на продукта: начален и краен. Една задача не може да бъде започната, ако не е завършен началния й етап. За да бъде завършен изцяло един етап, трябва да бъдат завършени всички задачи, за които той се явява краен.

Задача: да се определи минималното време (при неограничен брой програмисти), достатъчно за завършването на целия проект (т.е. всичките му етапи).

Ако за модел на задачата използваме тегловен ориентиран граф, в който върховете са етапите , а ребрата - задачите, то търсенето на минималното време ще бъде равно на дължината на максималния по брой върховете път в графа. В литературата този път се нарича *критичен път*.

Алгоритъм:

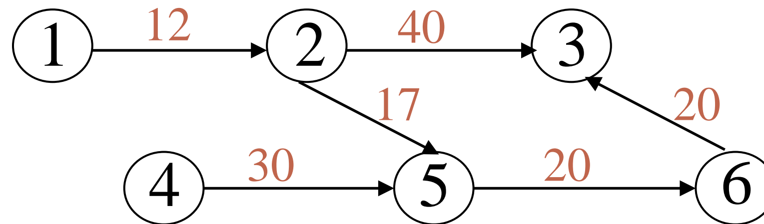
Нека е даден ацикличен граф *G (V, E)* с *n* върха и тегловна матрица *A[n][n]* на дъгите в *G*:



1. Въвеждаме масив *maxDist[]*, в който ще пазим *max* разстояния от всеки връх в графа, т.е. *maxDist[i]* ще бъде равно на дължината на *max* път с начало върха *i*. В началото инициализираме всички елементи на *maxDist[]* с нули. Във втори масив *savePath[]* ще пазим намерения най-дълъг път, като го инициализираме с -1.
2. Нека да започнем с връх i от V, от който не излизат ребра (такъв връх задължително съществува, т.к. графът е ацикличен). Премахваме този връх от графа и на всеки негов предшественик *k* присвояваме: *maxDist[k]* = **max**{*maxDist[k]*, *maxDist[i]*+*A[k][i]*}

3. Изпълняваме стъпка 2, докато не остане нито един връх в графа. Тогава максималната стойност на *maxDist[i]* за $i = 1, 2, \ldots, n$ ще бъде дължината на търсения максимален път.

Алгоритъмът има сложност O(n+m) в случая на динамично представяне на графа и $O(n^2)$ при представяне на графа с матрица на съседство.



1. Инициализация: *maxDist[i]*=0 за всяко *i*;
2. Изтриваме връх 3: *maxDist[6]=**max**{maxDist[6], maxDist[3]+20}= =max{0,20}=20;* Аналогично *maxDist[2]=40;*
3. Изтриваме връх 6: *maxDist[5]=40;*
4. Изтриваме връх 5: *maxDist[4]=70; maxDist[2]= **max**{40, 40+17}=57*
5. Изтриваме връх 4.
6. Изтриваме връх 2: *maxDist[1]=69;*
7. Изтриваме връх 1.

Най-дълъг път: 70 (*maxDist[4]=70)*

Алгоритъмът може да се реализира рекурсивно, като се използва модификация на обхождане в дълбочина (DFS).
При всяко връщане от рекурсивното извикване на *DFS(i)* се озоваваме в някой предшественик *j* на върха *i* . Там ще запазваме максималното *maxDist[j]+A[j][i]* за всеки *j* наследник на *i* и в края, преди изхода от функцията *DFS*, ще го присвояваме на *maxDist[i]*.
Функцията *DFS(i)* се изпълнява само, ако *maxDist[i]* още не е пресметнато, т. е. при *maxDist[i]* ==0 (като в началото инициализираме целия масив с нули). Функцията *DFS* се стартира последователно за всеки от върховете на графа.

За да запазим и отпечатаме всички върхове от търсения път, а не само дължината му, в масива *savePath[]* на позиция *i* ще записваме този наследник *j*, за който е намерено максималното *maxDist[i]+A[i][j]*.

Изходния код на програмата може да се намери в книгата на Преслав Наков, Панайот Добриков. *Програмиране = ++Алгоритми*, TopTeam Co., София, 2002
Там също така може да се намери описание на алгоритъма за търсене на най-дълъг прост път между два върха в произволен граф.

Цикли в граф (намиране и изследване)

Как можем да проверим, дали даден граф G е цикличен? Например, прилагаме вече известния алгоритъм за обхождане на граф в дълбочина (*DFS*). Ако на някоя стъпка от обхождането се окаже, че разглежданият връх *i* има съсед, който вече сме обходили и различен от предшественика на *i*, то следва, че G съдържа цикъл.

За целта функцията *DFS()* се модифицира - като параметър се добавя *int parent* - родителя на текущия връх. Ако от текущия връх *i* достигнем до връх, в който вече сме били, различен от *parent* , значи се е затворил цикъл.

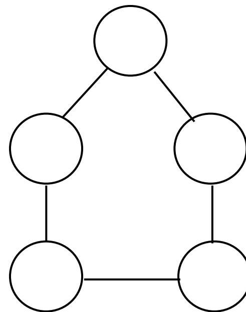
Реализацията на алгоритъма е представена в книгата на Преслав Наков, Панайот Добриков. *Програмиране = ++Алгоритми*, TopTeam Co., София, 2002.

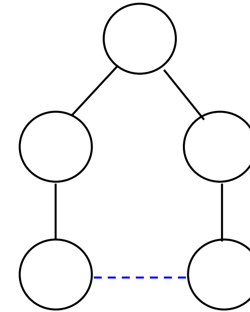
Намиране на фундаментално множество цикли

Неориентиран свързан граф без цикли се нарича *дърво*. *Покриващо (обхващащо) дърво* в свързан неориентиран граф *G(V,E) се нарича всеки свързан ацикличен подграф G'(V,E')* на *G.*

Нека е даден неориентиран свързан граф *G(V,E)* с *n* върха и *m* ребра. Нека *D(V,T)* е произволно покриващо дърво на *G:*

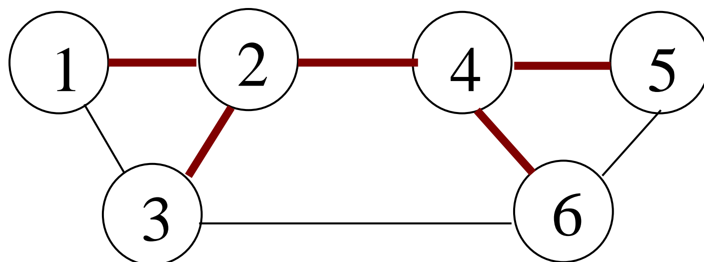
G(V,E) *D(V,T) (или G'(V,E'))*





Добавянето към *D* на ребро, непринадлежащо на *T,* ще доведе до затваряне на *прост цикъл.* Този цикъл принадлежи на *фундаменталното множество цикли* на графа *G* относно покриващото му дърво *D.*

Всяко покриващо дърво *D* има точно $q=n-1$ ребра. Ребрата на *G*, непринадлежащи на *D*, са общо $m-n+1$. Т. к. добавянето на всяко едно към *T* води до получаването на точно един цикъл, то броят на циклите от фундаменталното множество е $m-n+1$.



Определяйки фундаменталното множество цикли, ние определяме еднозначно цикличната структура на графа, т.к. всеки друг цикъл в *G* може да се представи чрез "слепване" на цикли от фундаменталното множество. В случая простите цикли са: A=(1,2,3); B=(4,5,6) и C=(2,3,4,6). Всеки друг прост цикъл може да се построи чрез съединяване на два от (A,B,C), които имат поне едно общо ребро (общото ребро/ребра се премахват, както и получените след тази процедура изолирани възли). Например, съединявайки A и C получаваме (1,2,4,6,3).

Намирането на фундаментално множество цикли пак може да бъде извършено чрез използване на модификация на функцията за обхождане на граф в дълбочина (*DFS*). На първата стъпка ще намерим едно произволно покриващо дърво на графа, а по-нататък всяко ребро, което не участва в построеното покриващо дърво, ще затваря цикъл, и този цикъл ще намерим с още едно обхождане на графа, реализирано също така с *DFS*. Общата сложност на алгоритъма е O(m*(m+n)).

Реализацията на алгоритъма е представена в книгата на Преслав Наков, Панайот Добриков. *Програмиране = ++Алгоритми*, TopTeam Co., София, 2002.

Хамилтонови цикли

Хамилтонов цикъл в граф се нарича цикъл, съдържащ всеки връх от графа точно веднъж. Граф, съдържащ такъв цикъл, се нарича *Хамилтонов*.

Известни са две класически задачи:

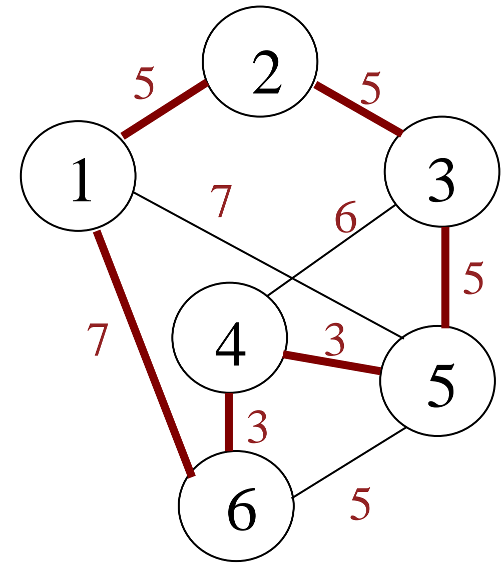
- проверка, дали даден граф е Хамилтонов;
- задача за търговския пътник (търсене на Хамилтонов цикъл с минимална дължина в тегловен граф).

И двете задачи се определят като NP-пълни, което означава, че сложността им в най-лошия случай е експоненциална.

Първата задача може да бъде решена чрез проверка на циклите в графа (пълно изчерпване на вариантите).

Втората вече е разглеждана. Тя също може да се реализира чрез пълно изчерпване (реализацията е представена в книгата на Преслав Наков, Панайот Добриков. *Програмиране = ++Алгоритми*, TopTeam Co., София, 2002 и практически е неприложима при големи ***n***).

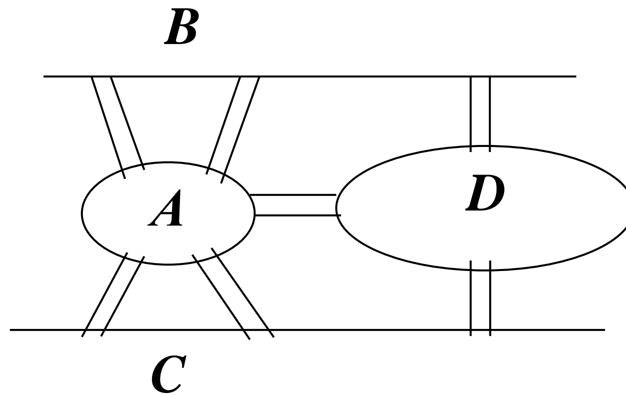
Например, за следния граф е Хамилтонов. С червено е означен минималния Хамилтонов цикъл, който е 1 2 3 5 4 6 1 и има цена 28. Получен е чрез преглеждане на всички маршрути, започващи от даден базов възел (в случая - 1). Ако на дадена стъпка получаваме междинна цена на изграждания маршрут, по-голяма от тази на минималния, изграждането на този маршрут се прекратява.

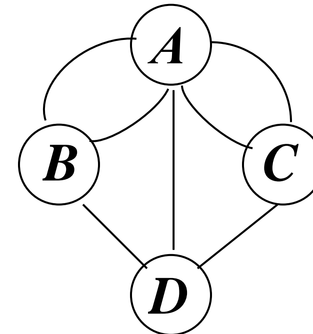


Поради широкото си приложение, задачата е изследвана много подробно в литературата. За решението й съществуват десетки алгоритми, някои от които гарантират точно решение с полиномиална сложност (например, метод на клоните и границите). Известни са и евристични подходи за решаване на задачата.

Ойлерови цикли

В теорията на графите се разглежда специален вид цикли, наречен на името на техния пръв изследовател Леонард Ойлер. Градът Кьонингсберг, където през XVIII век е живял Ойлер, е имал седем моста на река Прегел, построени по следния начин:





Хората там често си задавали въпроса: дали е възможно да се премине точно веднъж по всеки от мостовете и обиколката да приключи в изходната позиция. Ойлер е представил мостовете с *мултиграф* и показал, че това е невъзможно.

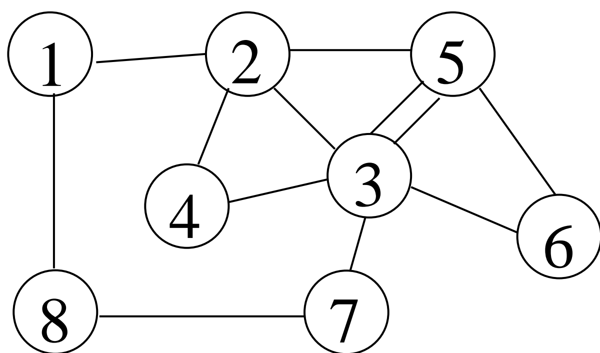
Граф *G(V,E)* се нарича *мултиграф*, ако множеството на ребрата му допуска повторение (т.е. *E* е *мултимножество*).

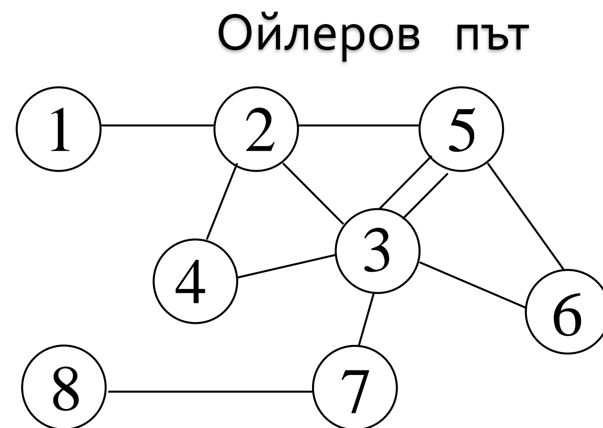
Нека е даден свързан мултиграф *G(V,E)*. Цикъл, в който всяко ребро участва точно по веднъж, се нарича *Ойлеров цикъл*. Един мултиграф се нарича *Ойлеров*, ако в него съществува *Ойлеров цикъл*.

Ойлеров път в граф се нарича път, в който всяко ребро участва точно по веднъж.

Теорема 1. (Изказана от Ойлер без доказателство). Свързан неориентиран мултиграф съдържа Ойлеров цикъл тогава и само тогава, когато всички върхове на графа са от четна степен.

Следствие 1. Свързан неориентиран мултиграф съдържа Ойлеров път тогава и само тогава, когато има точно два върха от нечетна степен.

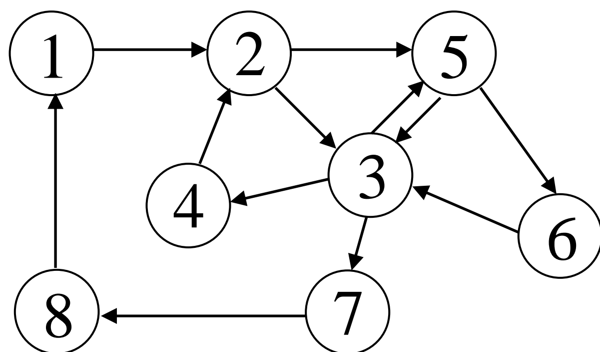


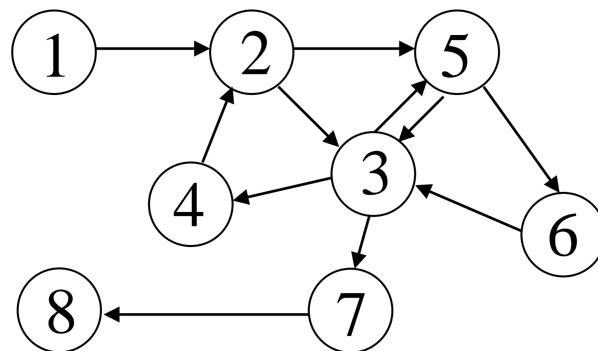


Ойлеров цикъл: (1,2),(2,3),(3,4),(4,2),(2,5),(5,3),(3,5),(5,6),(6,3),(3,7),(7,8),(8,1)

Теорема 2. Свързан ориентиран мултиграф съдържа Ойлеров цикъл тогава и само тогава, когато полустепента на входа *d'(i)* на всеки връх *i* е равна на полустепента на изхода *d(i),* т.е. *d'(i)=d(i)* за всеки възел *i* в множеството *V*.

Следствие 2. Свързан ориентиран мултиграф съдържа Ойлеров път тогава и само тогава, когато има точно два върха *i* и *j*, такива че *d'(i)=d(i)+1*, *d'(j)=d(j)-1* и *d'(k)=d(k)* за всяко *k* от множеството *V*, когато *k<>i, k<>j.*
(На дясната фигура: i=8, j=1, k=2, 3, 4, 5, 6, 7)

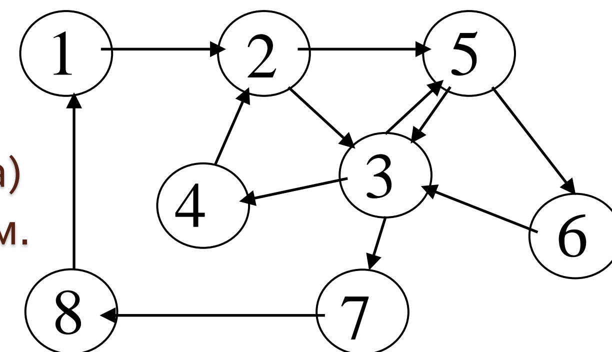




Как да се намери Ойлеров цикъл?

Започваме обхождане на графа от произволен връх *i*. Намираме инцидентно с него ребро *(i, j)* и го маркираме като посетено. Продължаваме с върха *j*: за него намираме непосетено ребро *(j,k)*, маркираме го като посетено и преминаваме в *k*. Продължавайки по този начин, в даден момент ще се озовем в началния връх *i* и ще затворим цикъла. Ако всички ребра на графа вече са маркирани, то този цикъл е Ойлеров. Ако са останали непосетени ребра, то намираме връх *x*, принадлежащ на току-що намерения цикъл и инцидентен с поне едно непосетено ребро. От това, че всеки връх трябва да бъде от четна степен, следва, че *x* е инцидентен с четен брой (т.е. поне две) непосетени ребра . От *x* започваме да изграждаме цикъл по вече описания начин, докато отново се върнем в него. Получаваме втори цикъл, който обединяваме с първия (общата им точка ще бъде върхът *x*). Така, след краен брой стъпки, всички ребра ще се включат в един общ цикъл - търсеният Ойлеров цикъл.

В случая, че търсим Ойлеров път, можем да приложим следната проста модификация: съединяваме върховете от нечетна степен с ребро (от. Следств.1: съществуват точно два такива върха) и намираме Ойлеров цикъл по описания алгоритъм. След отстраняване на добавеното ребро получаваме търсения Ойлеров път.



Алгоритъм за намиране на Ойлеров цикъл

Нека графът е представен с матрица на съседство *A[][]*. За реализацията ще се използват два стека: *stack[]* - за текущо построявания цикъл, и *cstack[]* - за обединението на всички построени до момента цикли. И двата стека първоначално ще бъдат празни.

```
Добавяме произволен връх i в stack;
While (stack не е празен)
  { Вземаме върха i от върха на stack без да го изключваме;
    if (i има наследници)
       { Вземаме произволен наследник j на i;
         Слагаме j в stack;
         A[j][i]=0;A[j][i]=0; //изключваме реброто от графа
       }
       else { Изключваме i от stack;
              Включваме i в cstack;
            }
  }
```

Реализацията е представена в книгата на Преслав Наков, Панайот Добриков. *Програмиране = ++Алгоритми*, TopTeam Co., София, 2002