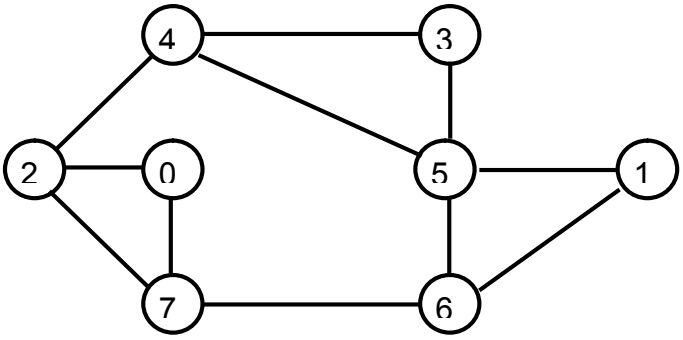
5. ГРАФИ

5.1. Определение и основни понятия

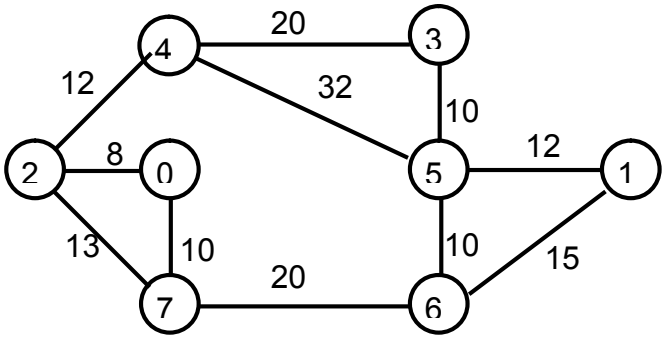
Графът е структура от данни, която се използва за решаването на редица интересни задачи от практиката. През последните десетилетия графите са обект на особен интерес както от страна на математиците, така и от страна на информатиците. Теорията на графите се обособи като обширен клон от дискретната математика с много интересни теореми и полезни приложения. Тук няма да се занимаваме с теорията на графите, а ще обърнем внимание на създаването на програми на C++, използващи графи.

Граф (фиг.27) наричаме множество от върхове (възли) и ребра. Всяко ребро свързва двойка върхове.

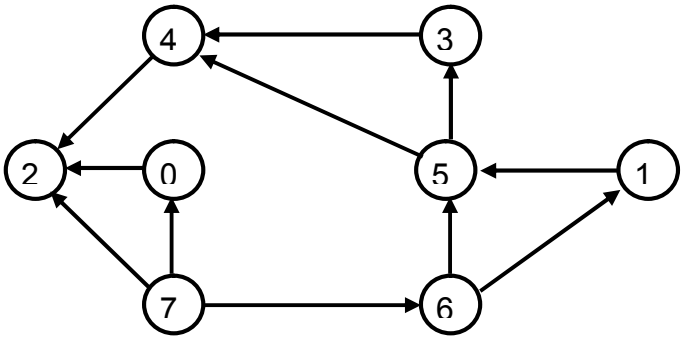
а) Неориентиран безтегловен граф



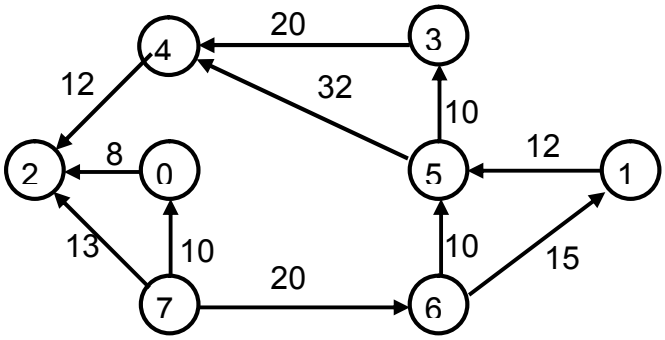
б) Неориентиран тегловен граф



в) Ориентиран безтегловен граф



г) Ориентиран тегловен граф



Фиг. 27. Графи

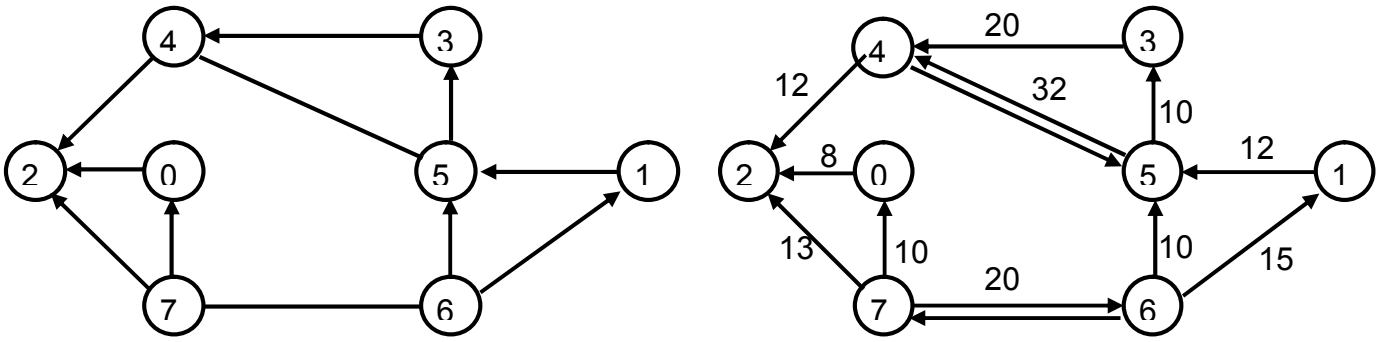
Върховете ще означаваме с числа, които можем да разглеждаме като номера на върховете. Всяко ребро ще означаваме с двойка числа, например ⟨3, 4⟩, които са номерата на върховете, свързвани от това ребро. Ребрата показват връзката между върховете, т.е. възможността да се премине от един връх в друг връх. Например двойката числа ⟨3, 4⟩ представя реброто, свързващо върховете 3 и 4 и допускащо преминаване от връх 3 във връх 4. Всяко ребро може да бъде ориентирано или неориентирано.

Ориентираното ребро е със стрелка, която указва посоката на възможното придвижване по него, а двойката, която го представя е ориентирана, например двойката ⟨3, 4⟩ показва, че е възможно придвижване от връх 3 във връх 4, а двойката ⟨4, 3⟩, че е възможно придвижване от връх 4 във връх 3.

Неориентираното ребро няма стрелка, двойката числа, която го представя не е ориентирана, и разрешава придвижване в двете посоки.

Графът е неориентиран, когато всичките му ребра са неориентирани, и обратно, графът е ориентиран, когато всичките му ребра са ориентирани. Всяко неориентирано ребро може да се представи чрез две ориентирани ребра (фиг.28).

Това позволява да превърнем в граф, който има неориентирани ребра, в ориентиран граф.



а) Граф с две неориентирани ребра б) Ориентиран граф

Фиг. 28. Превръщане на граф, съдържащ неориентирани ребра, в ориентиран граф

Даден граф може да бъде тегловен или безтегловен. Графът е тегловен граф, когато на върховете или/и ребрата са съпоставени тегла. Тук ще използваме тегловни графи с тегла, съпоставени на ребрата.

Същността на решаваната задача определя какъв граф да се използва, ориентиран или неориентиран, тегловен или безтегловен.

Бримка се нарича ребро, което има един и същи връх за начален и краен, например реброто $\langle 3, 3 \rangle$ е бримка.

Паралелни ребра са ребрата, свързващи една и съща двойка върхове.

Граф, който не съдържа бримки, паралелни ребра и ориентирани ребра, се нарича *обикновен граф* или просто *граф*.

Съседни върхове на даден връх в неориентиран граф са всички върхове, които са свързани с него чрез ребра.

Степен на връх на неориентиран граф ще наричаме броя на съседните му върхове. Връх без съседи е изолиран връх, а връх само с един съсед е висящ връх.

В ориентиран граф всеки връх има входящи и изходящи ребра. *Входящи* за върха са ребрата, по които от съседни върхове може да се стигне до него. *Изходящи* за върха са ребрата, по които от него може да се стигне до съседни върхове.

В ориентиран граф всеки връх има две степени: *входна степен* – брой на входящите ребра и *изходна степен* – брой на изходящите ребра.

Път в граф ще наричаме последователност от свързани (съседни) върхове. *Пътят е прост*, когато в него не се повтарят върхове и ребра. Пътят е *цикъл*, когато първият и последният връх от него съвпадат.

Цикълът е прост, когато не съдържа вложени цикли. Цикълът е хамилтонов, когато през всеки възел на графа се преминава еднократно. Цикълът е ойлеров, когато по всяко ребро на графа се преминава еднократно.

Пълен граф е този, в който всеки връх е свързан с всички останали.

Празен граф е този, в който няма нито едно ребро.

Допълнителен граф на даден граф е графът, който има същите върховете и само онези ребра, които не достигат на дадения, за да е пълен граф.

Еднороден граф е граф от върхове с еднаква степен. Пълният граф е винаги еднороден.

Равнинен е графът, който може да се изобрази в равнината без пресичане на ребра.

Двудялов е графът, чийто върхове са разделени на две подмножества и няма ребра между върхове, принадлежащи на различни множества.

Графите се използват за представяне на реални обекти, системи от обекти, процеси, явления и др. Например пътната мрежа в даден район може да се разглежда като тегловен граф, в който върхове са населените места и важните кръстовища, а ребра са пътищата между тях. С граф може да се представи и реализацията на някакъв процес, като всеки връх представя състоянието, до което се достига след реализиране на

определен етап, а ребро до този връх – това, което трябва да се направи, за да се реализира този етап.

5.2. Представяне на графи

5.2.1. Външно представяне

Става дума за представяне на графа на външен носител, например текстов файл. Един удобен начин е на първия ред във файла да се запише броят на върховете, а на всеки следващ ред – по едно ребро. Реброто се представя с две числа, когато графът е безтегловен, и с три числа, когато графът е тегловен. Третото число представя теглото, предписано на това ребро. Когато реброто е ориентирано, първото число е върхът, от който може да се тръгне по това ребро, а второто – върхът, в който може да се стигне по това ребро. Когато реброто не е ориентирано, редът на числата е без значение, но обикновено като първо се записва по-малкото число.

```
8
0 2
0 7
1 5
1 6
2 4
2 7
3 4
3 5
4 5
5 6
6 7
```

Външно представяне на графа от фиг. 27.а)

```
8
0 2  8
0 7 10
1 5 12
1 6 15
2 4 12
2 7 13
3 4 20
3 5 10
4 5 32
5 6 10
6 7 20
```

Външно представяне на графа от фиг. 27.б)

```
8
0 2
1 5
3 4
4 2
5 3
5 4
6 1
6 5
6 7
7 0
7 2
```

Външно представяне на графа от фиг. 27.в)

```
8
0 2  8
1 5 12
3 4 20
4 2 12
5 3 10
5 4 32
6 1 15
6 5 10
6 7 20
7 0 10
7 2 13
```

Външно представяне на графа от фиг. 27г)

Фиг. 29. Външно представяне на граф

5.2.2. Вътрешно представяне

Става дума за представяне на графа в ОП по време на изпълнение на програма, ползваща структурата граф. То се оценява по два критерия:

- необходима памет;
- сложност на реализация на операциите: определяне дали два върха са съседни и намиране на всички възли, съседни на даден възел.

5.2.2.1. Вътрешно представяне чрез матрица на съседство.

Матрицата на съседство, представяща граф, е квадратна матрица с размер, равен на броя на върховете. Номерата на редовете, както и номерата на стълбовете, са номера на върхове. Матриците на съседство на графите от фиг. 27 са следните:

за графа от фиг.1а)

	0	1	2	3	4	5	6	7
0	0	0	1	0	0	0	0	1
1	0	0	0	0	0	1	1	0
2	1	0	0	0	1	0	0	1
3	0	0	0	0	1	1	0	0
4	0	0	1	1	0	1	0	0
5	0	1	0	1	1	0	1	0
6	0	1	0	0	0	1	0	1
7	1	0	1	0	0	0	1	0

Матрица на съседство

	0	1	2	3	4	5	6	7
0	0	0	8	0	0	0	0	10
1	0	0	0	0	0	12	15	0
2	8	0	0	0	12	0	0	13
3	0	0	0	0	20	10	0	0
4	0	0	12	20	0	32	0	0
5	0	12	0	10	32	0	10	0
6	0	15	0	0	0	10	0	20

```
 7 10  0 13  0  0  0 20  0
```
Матрица на съседство за графа от фиг.1. б)

```
   0  1  2  3  4  5  6  7
0  0  0  1  0  0  0  0  0
1  0  0  0  0  0  1  0  0
2  0  0  0  0  0  0  0  0
3  0  0  0  0  1  0  0  0
4  0  0  1  0  0  0  0  0
5  0  0  0  1  1  0  0  0
6  0  1  0  0  0  1  0  1
7  1  0  1  0  0  0  0  0
```
Матрица на съседство за графа от фиг.1.в)

```
   0  1  2  3  4  5  6  7
0  0  0  8  0  0  0  0  0
1  0  0  0  0  0 12  0  0
2  0  0  0  0  0  0  0  0
3  0  0  0  0 20  0  0  0
4  0  0 12  0  0  0  0  0
5  0  0  0 10 32  0  0  0
6  0 15  0  0  0 10  0 20
7 10  0 13  0  0  0  0  0
```
Матрица на съседство за графа от фиг.1. г)

Фиг. 30. Вътрешно представяне на граф чрез матрица на съседство

Елементите на матрицата показват дали съществуват ребра между отделните върхове или не съществуват. Например елементът, който се намира на `i`-ия ред и на `j`-ия стълб, има стойност 1 или 0 в зависимост от това, дали съществува реброто ⟨`i,j`⟩ или не. При тегловен граф стойността на елемент, представящ дадено ребро, е самото тегло, а не 1. Матрицата на съседство, представяща неориентиран граф, е симетрична, защото, ако съществува реброто ⟨`i,j`⟩, съществува и реброто ⟨`j,i`⟩ . Елементите от главния диагонал са нулеви, ако графът не съдържа бримки.

Матрицата на съседство предлага определена прегледност. В `i`-ия ред, ненулеви са всички елементи, които се намират в стълбове, съответстващи на върхове, с които `i`-ият връх е свързан с изходящи ребра. В `j`-ия стълб, ненулеви са всички елементи, с които `j`-ият връх е свързан с входящи ребра Или броят на ненулевите елементи в і-ия ред дава изходящата степен на і-ия връх, а броят на ненулевите елементи в `j`-ия стълб дава входящата степен на `j`-ия връх.

Матрицата на съседство е много подходящ начин за представяне на граф от гледна точка на втория крирерий. От гледна точка на първия критерий (необходима памет), обаче, тя не е особено подходяща, защото граф с `n` върха изисква матрица с n^2 елемента, при това често преобладаващата част от тях са с нулеви стойности.

Програма 5.1. Програма за създаване и визуализация на матрицата на съседство на граф, представен външно в текстов файл

```
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <conio.h>
unsigned BrV;  //Брой върхове в графа
ifstream fin;  //Глобален входен поток за четене от текстовия файл
ofstream fout; //Глобален изходен поток за четене от текстов файл
//Клас граф
class clasGraph{
   int **A;    //Указател за създаване на матрицата на съседство
   public:
     clasGraph();         //Конструктор
     ~clasGraph();        //Деструктор
     void make_graf();    //Запълва матрицата на съседство
     void display_graf(ostream &);//Визуализира матрицата на съседство
};
//Конструктор за създаване и инициализация на матрицата на съседство
clasGraph::clasGraph()
```

```
  {
      int i,j;
      if (!(A=new int*[BrV])) exit(1);
      for (i=0; i<BrV; i++) if (!(A[i]=new int[BrV])) exit(1);
      for (i=0; i<BrV; i++)
       for (j=0; j<BrV; j++) A[i][j]=0;
  }
//Деструктор, който изтрива матрицата на съседство
clasGraph::~clasGraph()
  {
      for (int i=0; i<BrV; i++) delete []A[i];
      delete []A; cout<<"\nМатрицата на съседство е изтрита.\n";
  }
//Член-функция за запълване на матрицата на съседство
void clasGraph::make_graf()
 {
  char kod_or,kod_teg; int i,j,teg;
  cout<<"Графът ориентиран ли е /D,d,Д,д/? "; cin>>kod_or;
  cout<<"Графът тегловен ли е /D,d,Д,д/? ";   cin>>kod_teg;
  for (;;) {
     fin>>i>>j;if (fin.fail()) break;
     if (kod_teg=='D'||kod_teg=='d'||kod_teg=='Д'||kod_teg=='д')
fin>>teg;
     else teg=1;
     A[i][j]=teg;
     if (kod_or!='D'&&kod_or!='d'&&kod_or!='Д'&&kod_or!='д')
A[j][i]=teg;
  }
  fin.close();
}
//Член-функция за извеждане на матрицата на съседство на екрана или в
//текстов файл
void clasGraph::display_graf(ostream &p=cout)
 {
  if (p!=cout) fout.open("DopInf",ios::out);
  int i,j;
  p<<"\n      МАТРИЦА НА СЪСЕДСТВО\n";
  p<<"    ";
  for (i=0;i<BrV;i++) p<<setw(3)<<i;p<<endl;
  for (i=0;i<BrV;i++){
      p<<setw(4)<<i;
      for (j=0;j<BrV;j++) p<<setw(3)<<A[i][j];
      p<<endl;
  }
  if (p!=cout){
     fout.close();
     cout<<"\nОтрорете текст. файл DopInf.\n";
  }
 }
//Главна функция
 void main()
{
  char ime_vh_fl[21];
  cout<<"\nИме на входния файл: "; cin>>ime_vh_fl;
  fin.open(ime_vh_fl,ios::in);
  if (fin.bad()) {cout<<"Няма такъв файл.\n";getch();return;}
```

```
  fin>>BrV; //Въвежда от текстовия файл броя на върховете на графа
  clasGraph Graph;  //Създава обекта Graph
  Graph.make_graf();    //Запълва матрицата на съседство
  Graph.display_graf(); //Извежда матрицата на съседство на екрана
  Graph.display_graf(fout);//Извежда матрицата на съседство в текстов
//файл
  getch();
}
```

Главната функция прочита броя на върховете `BrV` от текстовия файл.

Конструкторът на класа създава матрицата на съседство като двумерен динамичен масив с `BrV` реда и `BrV` стълба и инициализира всички елементи със стойност 0.

Член-функцията `make_graf()`, след два уточняващи въпроса (дали графът е ориентиран и дали е тегловен), прочита данните от файла външно представяне на графа и заменя част от нулите в матрицата с 1 или с теглата на съответните ребра.

Член-функцията `display_graf(ostream &p=cout)` извежда матрицата:

- на екрана, когато фиктивният аргумент p не е заместен с действителен аргумент;
- в текстов файл с име “DopInf”, когато фиктивният аргумент p е заместен с изходния поток `fout`.

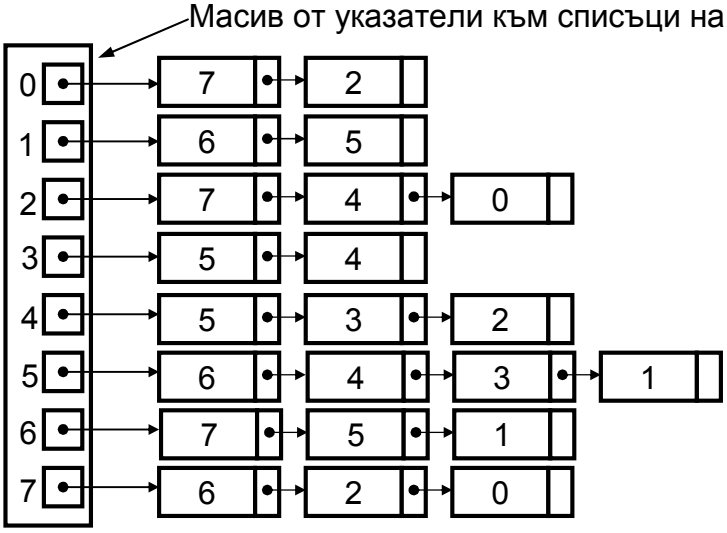
Деструкторът изтрива матрицата непосредствено преди излизане от програмата.

Самата програма не се нуждае от по-подробно описание, защото е проста и достатъчно наситена с пояснителни текстове.

5.2.2.2. Вътрешно представяне чрез масив от списъци на съседство

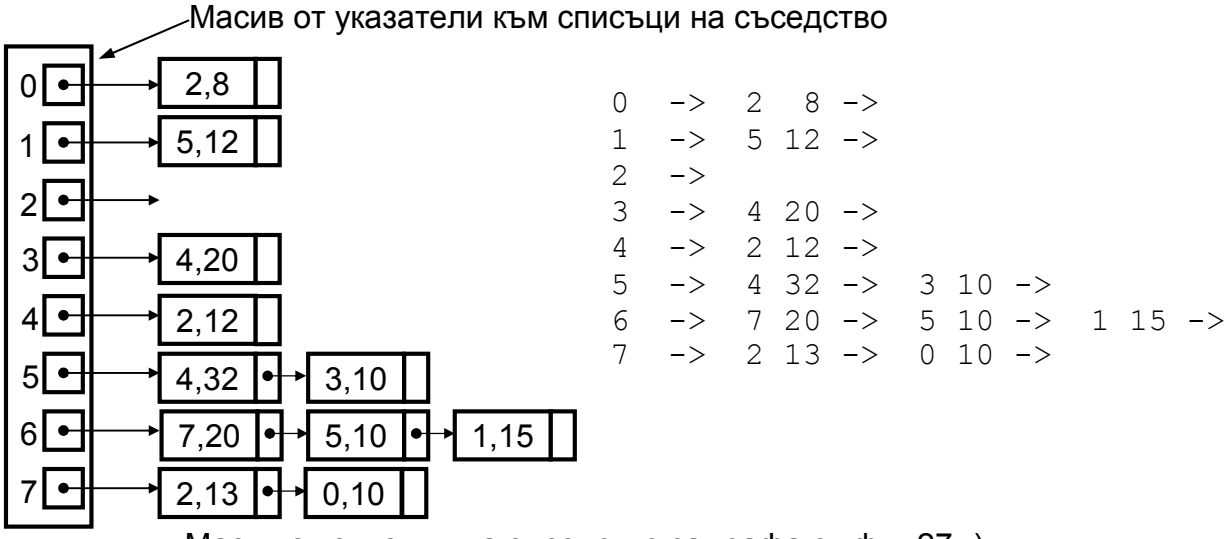
На следващата фигура се показани списъците на съседство, представящи графите от фиг.27.а) и фиг.27.г) по два начина, отляво – както вече сме го предстсвяли в в гл. 3 и отдясно – както го визуализира член-функцията `void display_graf(ostream &)` от следващата програма.

Първоначално се създава масив от указатели към списъци, които се инициализират като празни. След това започва добавянето на възли в списъците. Всеки възел представя едно ребро, например възелът 7 в 0-вия списък представя реброто ⟨0,7⟩, а възелът 2 от същия списък – реброто ⟨0,2⟩. Списъците не са подредени, поради което всеки нов възел може да се представи на произволно място в съответния списък. С цел да се постигне по-голямо бързодействие на програмата, всеки нов възел се разполага в челото на съответния списък. Всяко ребро се описва само с номера на върха, до който то води или с номера на върха и теглото на реброто, когато графът е тегловен. Например `i`-ия списък има толкова елемента, колкото са ребрата, излизащи от `i`-ия връх.




```
0  ->  7 ->  2 ->
1  ->  6 ->  5 ->
2  ->  7 ->  4 ->  0 ->
3  ->  5 ->  4 ->
4  ->  5 ->  3 ->  2 ->
5  ->  6 ->  4 ->  3 ->  1 ->
6  ->  7 ->  5 ->  1 ->
7  ->  6  ->  2  ->  0 ->
```

Масив от списъци на съседство за графа от фиг.27.а)



Масив от списъци на съседство за графа от фиг.27 г)

Фиг. 31. Вътрешно представяне на граф чрез масив от списъци на съседство.

При компютърната визуализация, числото в първата колона е индекс на елемент от масива от списъци и същевременно номер на върха, чиито съседи представя този списък-елемент на масива. Знакът -> е указател към ребро. Числото след знака -> е номер на върха-съсед.

Например редът `0 -> 7 -> 2 ->`
съдържа информацията: върхът 0 има за съседи върховете 7 и 2.

А редът `5 -> 4 32 -> 3 10 ->`
съдържа информацията: върхът 5 има за съседи върховете 4 и 3, като теглото на ⟨5, 4⟩ е 32, а на ⟨5, 3⟩ е 10.

Този начин за вътрешно представяне на граф е по-добър от гледна точка на първия критерий. Всеки списък на съседство съдържа точно толкова елемента, колкото е броят на изходящите от върха ребра.

По отношение на втория критерий има малък проблем, по-трудно се намират входящите съседи на върховете. Това обаче не е чак толкова голям проблем, защото, ако в програмата често се ползват входящите съседи, може да се създаде и масив от списъци на входящите съседи.

Програма 5.2. Програма за създаване и визуализация на масив от списъци на съседство на граф, представен външно в текстов файл

```
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <conio.h>
unsigned BrV;  //Брой на върховете в графа
//Структура ребро.
struct rebro{int nom, teg; rebro *next;};
ifstream fin; //Глобален входен поток за четене от текстовия файл
ofstream fout;  //Глобален изходен поток за четене от текст. файл
//Структура връх от графа
struct vrah { rebro *first; };
//Клас граф
class clasGraph {
     vrah *a;//Указател, необходим за съсдаване на масива от списъци
на съседство
  public:
```

```
     clasGraph(); //Конструктор за създаване и инициализация на масива
от списъци
     ~clasGraph();//Деструктор за изтриване на граф
     void make_graf();
     void display_graf(ostream &);
};
// Конструктор за създаване и инициализация на масива от списъци на
съседство
clasGraph::clasGraph()
 {
     a=new vrah[BrV];
     for (int i=0;i<BrV;i++) { a[i].first=0;}
 }
//Деструктор за изтриване на масива от списъци на съседство
clasGraph::~clasGraph()
 {
     rebro* p;
     for (int i=0;i<BrV;i++){
       while (a[i].first){
           p=a[i].first;
           a[i].first=p->next;
           delete p;
       }
     }
  delete []a; cout<<"Масивът от списъци е изтрит."<<endl;
 }
 char kod_teg;
//Член-функция за създаване списъците на съседство-елементи на масива
void clasGraph::make_graf()
 {
  rebro *p; char kod_or; unsigned i,j,teglo;
  cout<<"Графът ориентиран ли е /D,d,Д,д/? "; cin>>kod_or;
  if (kod_or=='D'||kod_or=='d'||kod_or=='Д'||kod_or=='д') kod_or=1;
  else kod_or=0;
  cout<<"Графът тегловен ли е /D,d,Д,д/? ";   cin>>kod_teg;
  if (kod_teg=='D'||kod_teg=='d'||kod_teg=='Д'||kod_teg=='д')
kod_teg=1;
  else kod_teg=0;
  for (;;) {
      fin>>i>>j; if (fin.fail()) break;
     if (kod_teg) fin>>teglo;
     else teglo=1;
     p=new rebro; p->nom=j; p->teg=teglo; p->next=a[i].first;
a[i].first=p;
     if (!kod_or) {
      p=new rebro; p->nom=i; p->teg=teglo; p->next=a[j].first;
a[j].first=p;
     }
  }
  fin.close ();
 }
//Член-функция за визуализация на масива от списъци на съседство
void clasGraph::display_graf(ostream &pp=cout)
{
 if (pp!=cout) fout.open("DopInf",ios::out);
 int i; rebro *p;
```

```
 pp<<"\nМАСИВ ОТ СПИСЪЦИ НА СЪСЕДСТВО:\n";
 for (i=0;i<BrV;i++){
  pp<<setw(2)<<i<<"     -> ";
  p=a[i].first;
  while (p!=0){
     pp<<setw(2)<<p->nom; if (kod_teg) pp<<setw(3)<<p->teg; pp<<" ->
";
     p=p->next;
  } pp<<endl;
 }
 if (pp!=cout){
     fout.close();
     cout<<"\nОтрорете текст. файл DopInf.\n";
  }
}
//Главна функция
 void main()
{
  char ime_vh_fl[21];
  cout<<"\nИме на входния файл: "; cin>>ime_vh_fl;
  fin.open(ime_vh_fl,ios::in);
  if (fin.bad()) {cout<<"Няма такъв файл.\n";getch();return;}
  fin>>BrV; //Въвеждаме от текстовия файл броя на върховете на графа
  clasGraph Graph;  //Създаваме обекта Graph
  Graph.make_graf();    //Създаваме списъците на съседство
  Graph.display_graf(fout); //Извеждаме на екрана масива от списъци на
съседство
  getch();
}
```

Главната програма прочита броя на върховете `BrV` от текстовия файл.

Конструкторът на класа създава динамичен масив от указатели с `BrV` елемента и инициализира всички указатели със стойност 0.

Член-функцията `make_graf()`, след два уточняващи въпроса дали графът е ориентиран и дали е тегловен, прочита данните от файла за външно представяне на графа и добавя към списъците на съседство елементи, съответстващи на изходящите съседни ребра.

Член-функцията `display_graf(ostream &pp=cout)` извежда масива от списъци:

- на екрана, когато фиктивният аргумент `p` не е заместен с действителен аргумент;
- в текстов файл с име `"DopInf"`, когато фиктивният аргумент `p` е заместен с изходния поток `fout`.

Деструкторът изтрива масива от списъци на съседство непосредствено преди излизане от програмата.

Самата програма не се нуждае от по-подробно описание, защото е проста и достатъчно наситена с пояснения.

5.3. Основни операции. Помощни програмни файлове

Основните операции са:

- обхождане на граф в ширина;
- обхождане на граф в дълбочина;
- топологично сортиране.

Тези операции са в основата на редица алгоритми за решаване на приложни задачи.

С цел да си улесним работата по създаването на програми, реализиращи горните операции, предварително ще подготвим два заглавни файла (по един за двата начина за вътрешно представяне на граф). Тези заглавни файлове ще ползваме както при

създаване на програмите за обхождане на граф в ширина, за обхождане на граф в дълбочина и за топологично сортиране, така и при програмите за решаване на конкретни приложни задачи. В заглавните файлове са включени клас стек и клас опашка, защото, както ще видим, при обхожданията се ползват тези познати структури.

Програма 5.3. Заглавен файл за програми, в които графът се представя чрез матрица на съседство

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <conio.h>
unsigned BrV;      //Брой върхове в графа
//Клас на стека
 class clasSt{
   public:
    unsigned *mas;   //Масив, съхраняващ елементите на стека
    unsigned Br;         //Брояч на елементите на стека
    clasSt(){Br=0; mas=new unsigned[2*BrV];}
    ~clasSt() {delete []mas; cout<<"Стекът е изтрит.\n";}
    int DobEl(unsigned &X);
    int VzeEl(unsigned &X);
  };
//Член-функция за добавяне на елемент към стек
int clasSt::DobEl(unsigned &X)
 {
   if (Br==2*BrV) return 0;//В стека няма място за нов елемент
   mas[Br++]=X; return 1;
 }
//Член-функция за отнемане на елемент от стек
int clasSt::VzeEl(unsigned &X)
 {
   if (Br==0) return 0; //Стекът е празен
   X=mas[--Br]; return 1;
 }
 //Клас на опашката
 class clasOp{
    unsigned *mas,  //Масивът, съхраняващ елементите от опашката
             firstFull, //Индекс на първия от заетите елементи в масива
             firstFree;   //Индекс на първия от свободните елементи в
масива
  public:
    int Br;          //Брояч на елементите в опашката
    clasOp() { Br=firstFull=firstFree=0;mas=new unsigned[BrV];}
    ~clasOp() {delete []mas; cout<<"Опашката е изтрита.\n";}
    int DobEl(unsigned &X);
    int VzeEl(unsigned &X);
 };

//Член-функция за добавяне на елемент към опашката.
int clasOp::DobEl(unsigned &X)
 {
   if (Br==BrV) return 0;//В опашката няма място за нов елемент
   mas[firstFree]=X;
   if (++firstFree==BrV) firstFree=0;
   Br++; return 1;
```

```
 }
////Член-функция за вземане на елемент от опашката.
int clasOp::VzeEl(unsigned &X)
 {
   if (Br==0) return 0;       //Опашката е празна
   X=mas[firstFull];
   if (++firstFull==BrV) firstFull=0;
   Br--; return 1;
 }
ifstream fin;  //Глобален входен поток за четене от текстовия файл
ofstream fout; //Глобален изходен поток за извеждане в текстов файл
class clasGraph{
   int **A;         //Матрица на съседство на графа
   clasSt Stek;     //Стек, който се използва при обхождане в
дълбочина
   clasOp Opashka;//Опашка, която се използва. при обхождане в ширина
и топологично сортиране
  public:
   clasGraph();        //Конструктор
   ~clasGraph();       //Деструктор
   void make_graf();   //Запълва матрицата на съседство
     void display_graf(ostream &);// Извежда матрицата на съседство на
екрана
   //void display_grafTF();//Извежда матрицата на съседство в
текст.файл
   void BFS();         //Обхожда в ширина. Намира се във файл
"ob_s_m.cpp"
   void kraBFS(); //Намира най-кратък път в безтегл. граф -
"najk_mat.cpp".
   void DFS();         //Обхожда в дълбочина, като използва стек-
"ob_d_m.cpp"
   void DFS(unsigned i);//Обхожда в дълб., като използва рекурсия-
"ob_d_m_r.cpp"
   void vsiDFS(unsigned i, unsigned j);//Намира всички прости пътища
от i-ия до j-ия връх, намира се във файл "allp_gr.cpp".
   void  hamDFS(unsigned  i,unsigned  nivo);  //Най-кратка  обиколка  на
всички върхове.
//Намира се във файл "ham_mat.cpp"
   void top_sort();//Извършва топологично сортиране  - "top_mas.cpp"
   void dijkstra();//Най-кратък път от здаден връх до всички други -
"dejx_mat.cpp"
   void OrgProekt();
};
//Конструктор за създаване и инициализация на матрицата на съседство
clasGraph::clasGraph()
  {
    int i,j;
      if (!(A=new int*[BrV])) exit(1);
    for (i=0; i<BrV; i++) if (!(A[i]=new int[BrV])) exit(1);
    for (i=0; i<BrV; i++)
     for (j=0; j<BrV; j++) A[i][j]=0;
  }
//Деструктор, който изтрива матрицата на съседство
clasGraph::~clasGraph()
  {
    for (int i=0; i<BrV; i++) delete []A[i];
```

```
    delete []A; cout<<"\nМатрицата на съседство е изтрита.\n";
  }
//Член-функция за запълване на матрицата на съседство
void clasGraph::make_graf()
 {
  char kod_or,kod_teg; int i,j,teg;
  cout<<"Графът ориентиран ли е /D,d,Д,д/? "; cin>>kod_or;
  cout<<"Графът тегловен ли е /D,d,Д,д/? ";   cin>>kod_teg;
  for (;;) {
   fin>>i>>j;if (fin.fail()) break;
   if (kod_teg=='D'||kod_teg=='d'||kod_teg=='Д'||kod_teg=='д')
fin>>teg;
     else teg=1;
   A[i][j]=teg;
   if (kod_or!='D'&&kod_or!='d'&&kod_or!='Д'&&kod_or!='д')
A[j][i]=teg;
  }
  fin.close();
}
//Член-функция за извеждане на матрицата на съседство на екрана
//или в текстов файл
void clasGraph::display_graf(ostream &p=cout)
 {
  if (p!=cout) fout.open("DopInf",ios::out);
  int i,j;
  p<<"\n      МАТРИЦА НА СЪСЕДСТВО\n";
  p<<"    ";
  for (i=0;i<BrV;i++) p<<setw(3)<<i;p<<endl;
  for (i=0;i<BrV;i++){
      p<<setw(4)<<i;
      for (j=0;j<BrV;j++) p<<setw(3)<<A[i][j];
      p<<endl;
  }
  if (p!=cout){
     fout.close();
     cout<<"\nОтрорете текст. файл DopInf.\n";
  }
 }
```

Програмата включва само познати неща: класовете стек, опашка и граф, както и член-функциите на тези класове.

Програма 5.4. Заглавен файл за програми, в които графът се представя чрез масив от списъци на съседство

```
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <conio.h>
unsigned BrV;  //Брой на върховете в графа
//Структура rebro. Всеки елемент от списък на съседство, стек или
опашка е
//обект от тази структура.
struct rebro{int nom, teg; rebro *next;};
// Клас стек
 class clasSt{
   rebro *first;    //Указател към върха на стека
  public:
```

```
   int Br;          //Брояч на елементите в стека
   clasSt() { Br=0; first=0; }
   int DobEl(unsigned X);
   int VzeEl(unsigned &X);
  };

//Член-функция за добавяне на елемент в стек

int clasSt::DobEl(unsigned X)
 {
   rebro *p;
   if (!(p=new rebro)) return 0;
   p->nom=X; p->next=first;
   first=p;Br++;return 1;
 }

//Член-функция за вземане на елемент от стека
int clasSt::VzeEl(unsigned &X)
 {
   rebro *p;
   if (!first) return 0;
   p=first;
   X=p->nom;first=p->next;
   delete p; Br--; return 1;
 }

// Клас опашка
 class clasOp{
   rebro *first,*last;
  public:
   unsigned Br;
   clasOp::clasOp() { Br=0; first=last=0;}
   int DobEl(unsigned X);
   int VzeEl(unsigned &X);
  };

//Член-функция за добавяне на компонента в опашка
int clasOp::DobEl(unsigned X)
 {
   rebro *p;
   if (!(p=new rebro)) return 0;
   p->nom=X;
   if (!last) first=p;
   else last->next=p;
   last=p;
   Br++;
   return 1;
 }

//Член-функция за вземане на компонента от опашката
int clasOp::VzeEl(unsigned &X)
 {
   rebro *p;
   if (!first) return 0;
   p=first;
   X=p->nom;
```

```
   if (first==last) last=0;
   first=first->next;
   delete p; Br--; return 1;
 }
ifstream fin; //Глобален входен поток за четене от текстовия файл
ofstream fout;
//Структура връх от граф
struct vrah { rebro *first; };
//Клас граф
class clasGraph {
  private:
   vrah *a;//Указател, необходим за съсдаване на масива от списъци на
съседство
   clasSt Stek;
   clasOp Opashka;
  public:
   clasGraph(); //Конструктор за създаване и инициализация на граф
   ~clasGraph();//Деструктор за изтриване на граф
   void make_graf();
     void display_graf(ostream &);
void BFS();       //Обхожда в ширина. Намира се във файл "ob_s_s.cpp"
   void kraBFS(); //Намира най-кратък път в безтегл. граф -
"najkr_sp.cpp".
   void DFS();    //Обхожда в дълбочина, като използва стек -
"ob_d_s.cpp"
   void DFS(unsigned i); //Обхожда в дълбочина, като използва рекурсия
- "ob_d_s_r.cpp"
   void vsiDFS(unsigned, unsigned);     //Намира всички прости пътища
от i-ия до j-ия връх -файл "allp_sp.cpp".
   void hamDFS(unsigned, unsigned); //Най-кратка обиколка на всички
върхове – "ham_sp.cpp"
   void top_sort();//Извършва топологично сортиране  - "top_sp.cpp"
   void dijkstra();//Най-кратък път от здаден връх до всички други -
"dejx_sp.cpp"
};
// Конструктор за създаване и инициализация на граф
clasGraph::clasGraph()
 {
   a=new vrah[BrV];
   for (int i=0;i<BrV;i++) { a[i].first=0;}
 }
//Деструктор за изтриване на графа
clasGraph::~clasGraph()
 {
   rebro* p;
   for (int i=0;i<BrV;i++){
     while (a[i].first){
       p=a[i].first;
       a[i].first=p->next;
       delete p;
     }
   }
  delete []a; cout<<"\nМасивът от списъци е изтрит."<<endl;
 }
 char kod_teg;
//Член-функция за изграждане на масива от списъци на съседство
```

```
void clasGraph::make_graf()
 {
  rebro *p; char kod_or; unsigned i,j,teglo;
  cout<<"Графът ориентиран ли е /D,d,Д,д/? "; cin>>kod_or;
  if (kod_or=='D'||kod_or=='d'||kod_or=='Д'||kod_or=='д') kod_or=1;
  else kod_or=0;
  cout<<"Графът тегловен ли е /D,d,Д,д/? ";   cin>>kod_teg;
  if (kod_teg=='D'||kod_teg=='d'||kod_teg=='Д'||kod_teg=='д')
kod_teg=1;
  else kod_teg=0;
  for (;;) {
    fin>>i>>j; if (fin.fail()) break;
   if (kod_teg) fin>>teglo;
   else teglo=1;
   p=new rebro; p->nom=j; p->teg=teglo; p->next=a[i].first;
a[i].first=p;
   if (!kod_or) {
    p=new rebro; p->nom=i; p->teg=teglo; p->next=a[j].first;
a[j].first=p;
   }
  }
  fin.close ();
 }
//Член-функция за визуализация на масива от списъци на съседство
void clasGraph::display_graf(ostream &pp=cout)
{
 if (pp!=cout) fout.open("DopInf",ios::out);
 int i; rebro *p;
 pp<<"\n МАСИВ ОТ СПИСЪЦИ НА СЪСЕДСТВО\n";
 for (i=0;i<BrV;i++){
  pp<<setw(2)<<i<<"     -> ";
  p=a[i].first;
  while (p!=0){
     pp<<setw(2)<<p->nom; if (kod_teg) pp<<setw(3)<<p->teg; pp<<" ->
";
     p=p->next;
  } pp<<endl;
 }
 if (pp!=cout){
     fout.close();
     cout<<"\nОтрорете текст. файл DopInf.\n";
 }
}
```

Програмата включва само познати неща: класовете стек, опашка и граф, както и член-функциите на тези класове. Новото е, че към член-данните на класовете стек и опашка е добавен брояч на елементите. Той е необходим за повечето от програмите, които ползват заглавния файл.

5.4. Обхождане на граф в ширина

Да обходим един граф означава да посетим последователно всичките му върхове. Обикновено обхождането се прави с цел да се ползва информацията, която се съдържа във всеки връх. Съществуват два вида охождания: обхождане в ширина и обхождане в дълбочина. Обхождането може да започне от всеки връх. Върхът, от който започва обхождането, ще наричаме стартов връх.

Обхождане в ширина става по нива. Стартовият връх е на нулево ниво. На първо ниво се намират всички съседи на стартовия връх. На второ ниво се намират всички необходени съседи на върховете от първо ниво и т.н. Нека да си поставим задачата да обходим в ширина графа от фиг. 1а, приемайки за стартов връх върха 2. Обхождането ще става по нива както следва:

0-во ниво - връх 2;

1-во ниво - върхове 0, 4, 7;

2-ро ниво - върхове 3, 5, 6;

3-то ниво - връх 1.

Или при обхождане в ширина на графа от фиг. 1а от стартов връх 2 върховете ще бъдат посетени в следната последователност: 2, 0, 4, 7, 3, 5, 6, 1, а при стартов връх 0: 0, 2, 7, 4, 6, 3, 5, 1. Препоръчваме на читателя да намери реда на обхождане, като избере за стартов всеки от върховете 1, 3, 4, 5, 6 и 7.

Върховете от едно ниво могат да се обходят в произволен ред, например коректен е и следният ред 2, 7, 4, 0, 6, 5, 3, 1. Обикновено върховете, съседи на даден връх, се обхождат в нарастващ ред на номерата им.

Видяхме, че не е трудно да обходим граф, представен графично. Нашата цел обаче е да напишем програма за обхождане на граф. Очевидно тази програма ще използва вътрешното представяне на графа, т.е. в случая матрицата на съседство. Само тя, обаче, не е достатъчна. Ще използваме още:

- опашка, в която ще чакат реда си върховете, за да бъдат посетени;
- масив, в който ще отбелязваме посетените върхове, ще го кръстим visit[].

Обхождането става по следния алгоритъм:

1. Инициализираме елементите на масива `visit[]` с нули, т.е. обявяваме върховете на графа за непосетени.

2. Ползваме (извеждаме в списъка на обходените), маркираме като ползван (обходен) и изпращаме в опашката стартовия връх.

3. Докато в опашката има върхове, вземаме ги един по един. След всяко вземане на връх от опашката, посещаваме един по един съседите му и с всеки непосетен преди това съсед правим следното:

- ползваме го (извеждаме го в списъка на обходените);
- маркираме го като посетен;
- изпращаме го в опашката.

Очевидно, операция 3 от алгоритъма е цикъл. Ползването, маркирането като посетен и изпращането в опашката на всеки непосетен преди това връх е вложен цикъл.

Нека да изпробваме този алгоритъм, за да обходим графа от фиг. 27 а), като изберем за начален връх 2.

1. Създаваме масив `visit[8]` и инициализираме елементите му с нули.

2. Ползваме (извеждаме в списъка на обходените) старъовия връх, маркираме го като посетен (`visit[2]`=1) и го изпращаме в опашката.

3. Вземаме от опашката стартовия връх 2 и от 2-ия ред на матрицата на съседство намираме съседите му (0, 4 и 7). От масива `visit[]` виждаме, че те не са посещавани и затова ги ползваме (извеждаме ги в списъка на обходените), маркираме ги като посетени (`visit[0]`=1, `visit[4]`=1 и `visit[7]`=1) и ги изпращаме в опашката.

4. Вземаме от опашката 0-ия връх и от 0-ия ред на матрицата на съседство намираме съседите му (2 и 7). От масива `visit[]` виждаме, че те вече са посещавани и затова ги отминаваме.

5. Вземаме от опашката 4-ия връх и от 4-ия ред на матрицата на съседство намираме съседите му (2, 3 и 5). От масива `visit[]` виждаме, че 3 и 5 не са посещавани и затова ги ползваме (извеждаме ги в списъка на обходените), маркираме ги като посетени (`visit[3]`=1 и `visit[5]`=1) и ги изпращаме в опашката.

6. Вземаме от опашката 7-ия връх и от 7-ия ред на матрицата на съседство намираме съседите му (0, 2 и 6). От масива `visit[]` виждаме, че само 6 не е посещаван и затова

го ползваме (извеждаме го в списъка на обходените), маркираме го като посетен (visit[6]=1) и го изпращаме в опашката.

7. Вземаме от опашката 3-ия връх и от 3-ия ред на матрицата на съседство намираме съседите му (4 и 5). От масива visit[] виждаме, че те посещаван и затова ги отминаваме.

8. Вземаме от опашката 5-ия връх и от 5-ия ред на матрицата на съседство намираме съседите му (1, 3, 4 и 6). От масива visit[] виждаме, че само връх 1 не е посещаван и затова го ползваме (извеждаме го в списъка на обходените), маркираме го като посетен (visit[1]=1) и го изпращаме в опашката.

9. Вземаме от опашката 6-ия връх и от 6-ия ред на матрицата на съседство намираме съседите му (1, 5 и 7). От масива visit[] виждаме, че те са посещавани и затова ги отминаваме.

10. Вземаме от опашката 1-ия връх и от 1-ия ред на матрицата на съседство намираме съседите му (5 и 7). От масива visit[] виждаме, че те са посещавани и затова ги отминаваме.

Тук обхождането приключва, защото опашката е празна.

По този алгоритъм е създадена следващата програма. Както се вижда, тя ползва заглавния файл "graf_mat.h" и се състои от член-функцията BFS(Breadth-First-Search) на класа clasGraph и главна функция. Член-функцията реализира описания по-горе алгоритъм.

Програма 5.5. Обхождане в ширина на граф, представен чрез матрица на съседство

```
# include <iostream.h>
# include <conio.h>
#include "graf_mat.h"
//Член-функция за обхождане на граф в ширина
void clasGraph::BFS()
{
  unsigned i,j,k,sv,*visit;
  if(!(visit=new unsigned[BrV])) exit(1);
  for (i=0; i<BrV; i++) visit[i]=0;
  cout<<"Начален връх: "; cin>>sv;
  cout<<"Обхождане в ширина от стартов връх "<<sv<<" -> ";
  cout<<sv<<' '; //Извеждаме (ползваме) стартовия връх
  visit[sv]=1;   //Маркираме стартовия връх като посетен
  Opashka.DobEl(sv);     //Добавяме стартовия елемент към опашката
  while (Opashka.Br){ //Докато в опашката има чакащи върхове ...
      Opashka.VzeEl(k);//... вземаме поредния чакащ връх.
      for (j=0;j<BrV;j++)//Всеки съсед на k-ия връх,...
       if (A[k][j]&&!visit[j]){//... ако не е посетен (ползван):...
          cout<<j<<' ';// - извеждаме го (ползваме го),
          visit[j]=1;  // - маркираме го и го изпращаме в опашката
          if (!Opashka.DobEl(j)){cout<<"\nОпашката е
пълна!\n";getch();exit(1);}
     }
  } delete []visit;
}

void main()
{
  char ime_vh_fl[21];
  cout<<"\nИме на входния файл: "; cin>>ime_vh_fl;
  fin.open(ime_vh_fl,ios::in);
  if (fin.bad()) {cout<<"Няма такъв файл.\n";getch();return;}
  fin>>BrV; //Въвежда от текстовия файл броя на върховете на графа
```

```
  clasGraph Graph;            //Създава обекта Graph
  Graph.make_graf();          //Запълва матрицата на съседство
  Graph.display_graf();       //Извежда матр. на съседство на екрана
  Graph.display_graf(fout);//Извежда матр. на съседство в текст. файл
  Graph.BFS();              //Обхожда графа в ширина
  getch();
}
```

Следващата програма реализира обхождане в ширина на граф, представен чрез масив от списъци на съседство. Тя се различава минимално от предходната и се надяваме, че читателят ще я разбере без допълнителен коментар. Ако се обходи същият граф (фиг.1а) с новата програма ще се получи следната последователност на обхождане: 2, 7, 4, 0, 5, 3, 6, 1. Тя са различава от последователността, която намира предходната програма, но също е вярна. Предоставяме на читателя да изясни причината.

Програма 5.6. Обхождане в ширина на граф, представен чрез масив от списъци на съседство

```
# include <iostream.h>
# include <conio.h>
# include "graf_sp.h"
//Член-функция за обхождане в ширина
void clasGraph::BFS()
{
  unsigned i,j,k,sv,*visit;
  if (!(visit=new unsigned[BrV])) exit(0);
  for (i=0;i<BrV;i++) visit[i]=0;
  cout<<"Стартов връх: "; cin>>sv;
  cout<<"Обхождане в ширина от стартов връх "<<sv<<" -> ";
  cout<<sv<<' '; //Извеждаме (ползваме) стартовия връх
  visit[sv]=1;   //Маркираме стартовия връх като посетен
  Opashka.DobEl(sv);  //Добавяме стартовия елемент към опашката
  while (Opashka.Br) {//Докато в опашката има чакащи върхове ...
      Opashka.VzeEl(k);//... вземаме поредния връх от опашката, ...
      rebro *p=a[k].first;
      //... и всеки негов съсед,...
      while (p){
          j=p->nom;
          if (!visit[j]){//...ако не е посетен (ползван)
            cout<<j<<' ';      // - извеждаме го (ползваме го),
            visit[j]=1;        // - маркираме го и го изпращаме в
опашката
            Opashka.DobEl(j);// - изпращаме го в опашката
          }
          p=p->next;
      }
  }
}

void main()
{
  char ime_vh_fl[21];
  cout<<"\nИме на входния файл: "; cin>>ime_vh_fl;
  fin.open(ime_vh_fl,ios::in);
  if (fin.bad()) {cout<<"НЯма такъв файл.\n";getch();return;}
  fin>>BrV;              //Въвежда броя на върховете от текстовия файл
  clasGraph Graph;           //Създава обекта граф
```

```
  Graph.make_graf();           //Запълва матр. на съседство
  Graph.display_graf();        //Извежда матр. на съседство на екрана
  Graph.display_graf(fout);    //Извежда матр. на съседство в текст.
файл
  Graph.BFS();                 //Обхожда графа в ширина
  getch();
}
```

5.5. Обхождане на граф в дълбочина

Това обхождане обаче не е обхождане по нива. Напротив, всеки следващ преход е към един от непосетените съседите на върха, а той е от друго ниво. Преходът напред към следващ връх продължава до достигане до ситуация, когато върхът, в който се намираме, няма непосетени съседи, т.е. не можем да продължим напред. Това обаче в общия случай не означава, че са обходени всички върхове. Затова правим стъпка назад към предходния връх и, ако той няма непосетен съсед, пак правим стъпка назад към неговия предходен връх. Това връщане назад продължава, докато попаднем във връх с непосетен съсед. Тогава преминаваме в непосетения съсед и продължаваме напред по този нов клон, докато отново достигнем до невъзможност да продължим напред. Отново започваме връщане назад и т.н. Обхождането е завършило, когато се окаже, че, отстъпвайки назад, сме се върнали в началния връх и той няма нито един непосетен съсед.

Има два алгоритъма за обхождане в дълбочина.

5.5.1. Обхождане на граф в дълбочина с използване на стек

Алгоритъмът за обхождане в дълбочина с използване на стек се различава от този за обхождане в ширина по следното:

- съседите на поредния връх се изпращат в стек (при обхождане в ширина се изпращат в опашка);
- с стека се изпращат всички съседи (както обходени, така и необходени). Проверка дали са обходени се прави при вземане на върховете от стека.

Обхождането в дълбочина с използване на стек става по следния алгоритъм:

1. Инициализираме елементите масива `visit[]` с нули, т.е. обявяваме върховете на графа за непосетени.
2. Изпращаме стартовия връх в стека.
3. Докато в стека има върхове, изпълняваме следното:
4. Вземаме от стека върховия елемент (върхът, изпратен в стека последен) и, ако той се окаже непосетен:
   - ползваме го (извеждаме го в списъка на обходените);
   - маркираме го като посетен;
   - изпращаме всички негови съседи (посетени и непосетени) в стека.

Очевидно, операция 4 от алгоритъма е цикъл с вложен цикъл. Добавянето към стека на съседите на взетия от стека връх е вложения.

Нека да изпробваме този алгоритъм, за да обходим графа от фиг. 1а, като изберем за начален връх 2.

1. Създаваме масив `visit[8]` и инициализираме елементите му с нули.
2. Изпращаме началния връх 2 в стека.
3. Вземаме от стека стартовия връх. Той не е посетен и затова: извеждаме **2** в списъка на обходените, регистрираме го като посетен (`visit[2]`=1), намираме от 2-ия ред на матрицата на съседство съседите му 0, 4 и 7 и ги добавяме към стека.
4. Вземаме от стека 7-ия връх. Той не е посетен и затова: извеждаме **7** в списъка на обходените, регистрираме го като посетен (`visit[7]`=1), намираме от 7-ия ред на матрицата на съседство съседите му 0, 2 и 6 и ги добавяме към стека.
5. Вземаме от стека 6-ия връх. Той не е посетен и затова: извеждаме **6** в списъка на обходените, регистрираме го като посетен (`visit[6]`=1), намираме от 6-ия ред на матрицата на съседство съседите му 1, 5 и 7 и ги добавяме към стека.

6. Вземаме от стека 7-ия връх. Той е посетен и затова го отминаваме.

7. Вземаме от стека 5-ия връх. Той не е посетен и затова: извеждаме **5** в списъка на обходените, регистрираме го като посетен (`visit[5]`=1), намираме от 5-ия ред на матрицата на съседство съседите му 1, 3, 4 и 6 и ги добавяме към стека.

8. Вземаме от стека 6-ия връх. Той е посетен и затова го отминаваме.

9. Вземаме от стека 4-ия връх. Той не е посетен и затова: извеждаме 4 в списъка на обходените, регистрираме го като посетен (`visit[4]`=1), намираме от 4-ия ред на матрицата на съседство съседите му 2, 3, и 5 и ги добавяме към стека.

10. Вземаме от стека 5-ия връх. Той е посетен и затова го отминаваме.

11. Вземаме от стека 3-ия връх. Той не е посетен и затова: извеждаме 3 в списъка на обходените, регистрираме го като посетен (`visit[3]`=1), намираме от 3-ия ред на матрицата на съседство съседите му 4 и 5 и ги добавяме към стека.

12. Вземаме от стека 5-ия връх. Той е посетен и затова го отминаваме.

13. Вземаме от стека 4-ия връх. Той е посетен и затова го отминаваме.

14. Вземаме от стека 3-ия връх. Той е посетен и затова го отминаваме.

15. Вземаме от стека 2-ия връх. Той е посетен и затова го отминаваме.

16. Вземаме от стека 3-ия връх. Той е посетен и затова го отминаваме.

17. Вземаме от стека 1-ия връх. Той не е посетен и затова: извеждаме 1 в списъка на обходените, регистрираме го като посетен (`visit[1]`=1), намираме от 3-ия ред на матрицата на съседство съседите му 5 и 6 и ги добавяме към стека.

18. Вземаме от стека 5-ия връх. Той е посетен и затова го отминаваме.

19. Вземаме от стека 4-ия връх. Той е посетен и затова го отминаваме.

20. Вземаме от стека 1-ия връх. Той е посетен и затова го отминаваме.

21. Вземаме от стека 2-ия връх. Той е посетен и затова го отминаваме.

22. Вземаме от стека 0-ия връх. Той не е посетен и затова: извеждаме 0 в списъка на обходените, регистрираме го като посетен (`visit[0]`=1), намираме от 0-ия ред на матрицата на съседство съседите му 2 и 7 и ги добавяме към стека.

23. Вземаме от стека 7-ия връх. Той е посетен и затова го отминаваме.

24. Вземаме от стека 2-ия връх. Той е посетен и затова го отминаваме.

25. Вземаме от стека 4-ия връх. Той е посетен и затова го отминаваме.

26. Вземаме от стека 0-ия връх. Той е посетен и затова го отминаваме.

Стекът вече е празен. Обхождането в дълбочина е завършило.

Следват две програми за обхождане на граф в дълбочина с използване на стек: програма за обхождане граф, представен чрез матрица на съседство и програма за обхождане граф, представен чрез масив от списъци на съседство. Обърнете внимание на това, че в отделни моменти броят на върховете в стека е по-голям от общия им брой. Затова броят на елементите на масива, ползван от стека, е два пъти по-голям от броя на върховете.

Програма 5.7. Обхождане в дълбочина на граф, представен чрез матрица на съседство

```
# include <iostream.h>
# include <conio.h>
#include "graf_mat.h"
//Член-функция за обхождане в дълбочина
void clasGraph::DFS()
{
  unsigned i,j,k,n,sv,*visit;
  if(!(visit=new unsigned[BrV])) exit(1);
  for (i=0; i<BrV; i++) visit[i]=0;
  cout<<"Стартов връх: "; cin>>sv;
  cout<<"Обхождане в дълбочина от стартов връх "<<sv<<" -> ";
  Stek.DobEl(sv);  //Добавяме към стека стартовия връх
  while (Stek.Br){ //Докато в стекът има чакащи върхове, ...
   Stek.VzeEl(k);  //... вземаме от стека върховия елемент, ...
```

```
   if (visit[k]) continue;// и ако е обходен, отминаваме го, ...
   visit[k]=1;  // ... маркираме го като обходен и ...
   cout<<k<<' ';
   // добавяме към стека всички негови съседи.
   for (j=0;j<BrV;j++)
    if (A[k][j]&&!Stek.DobEl(j)){cout<<"\nСтекът е
пълен!\n";getch();exit(1);}
  }
  delete []visit;
}
void main()
{
 char ime_vh_fl[21];
  cout<<"\nИме на входния файл: "; cin>>ime_vh_fl;
  fin.open(ime_vh_fl,ios::in);
  if (fin.bad()) {cout<<"Няма такъв файл.\n";getch();return;}
  fin>>BrV; //Въвеждаме от текстовия файл броя на върховете на графа
  clasGraph Graph;       //Създава обекта Graph
  Graph.make_graf();     //Запълва матрицата на съседство
  Graph.display_graf();  //Извежда матрицата на съседство на екрана
  Graph.display_graf(fout);//Извежда матрицата на съседство в текст.
файл
  Graph.DFS();           //Обхожда графа в дълбочина
  getch();
}
```

Ако обходим в дълбочина графа от фиг. 1а с горната програма, тръгвайки за начален връх върха 2, ще получим следния ред на обхождане : 2, 7, 6, 5, 4, 3, 1, 0.

Програма 5.8. Обхождане в дълбочина на граф, представен чрез масив от списъци на съседство

```
# include <iostream.h>
# include <conio.h>
# include "graf_sp.h"
//Функция за обхождане в дълбочина
void clasGraph::DFS()
{
  unsigned i,j,k,n,sv,*visit;
  if (!(visit=new unsigned[BrV])) exit(0);
  for (i=0;i<BrV;i++) visit[i]=0;
  cout<<"Стартов връх: "; cin>>sv;
  cout<<"Обхождане в дълбочина от стартов връх "<<sv<<" -> ";
  Stek.DobEl(sv);
  while (Stek.Br>0) {//Докато в стека има чакащи върхове ...
      Stek.VzeEl(k);//... вземаме върховия елемент от стека, ...
      if (visit[k]==1) continue; // ако е обходен, го отминаваме,...
      cout<<k<<' ';           //... ако не обходен, извеждаме го,...
      visit[k]=1;         //... маркираме го като обходен и ...
      rebro *p=a[k].first;
      while (p){ //... изпращаме в стека съседите му.
       j=p->nom;
       if (visit[j]==0) Stek.DobEl(j);
       p=p->next;
      }
  }
 }
```

```
void main()
{
  char ime_vh_fl[21];
  cout<<"\nИме на входния файл: "; cin>>ime_vh_fl;
  fin.open(ime_vh_fl,ios::in);
  if (fin.bad()) {cout<<"Няма такъв файл.\n";getch();return;}
  fin>>BrV; //Въвеждаме броя на върховете от текстовия файл
  clasGraph Graph;
  Graph.make_graf();
  Graph.display_graf();
  Graph.DFS();
  cout<<endl;
  getch();
}
```

Ако обходим в дълбочина графа от фиг. 1а с тази програма, вземайки за стартов връх върха 2, ще получим следния ред на обхождане : 2, 0, 7, 6, 1, 5, 3, 4.

Както се вижда двете програми предлагат различни редове на обхождане. Защо е така?

5.5.2. Обхождане на граф в дълбочина с използване на рекурсия

Този алгоритъм е по-кратък и по-елегантен. Той също ползва стек, но неявно, т.е. той използва не собствен стек, а програмния стек, който се ползва при всяко обръщение към функция.

Ще покажем два варианта на програма за обхождане в дълбочина.

Програма 5.9. Съдържа рекурсивна член-функция за обхождане в дълбочина на граф, представен чрез матрица на съседство

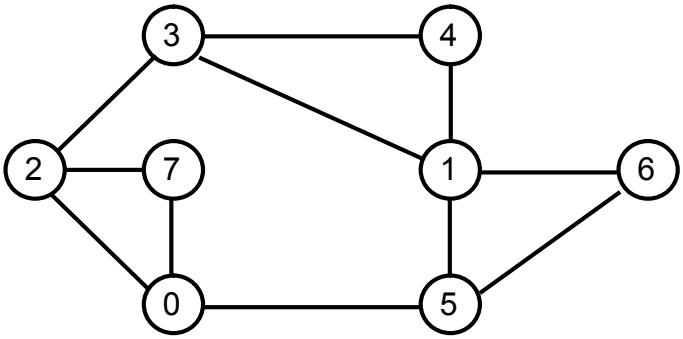
```
#include <fstream.h>
#include <conio.h>
#include "graf_mat.h"
unsigned *visit;
//Рекурсивна член-функция за обхождане на граф в дълбочина
void clasGraph::DFS(unsigned i)
 {
  unsigned j;
  visit[i]=1;  fout<<endl;
  cout<<i<<' ';fout<<setw(2)<<i<<"     ";
  for (j=0;j<BrV;j++) {
     fout<<j<<' ';
     if (A[i][j] && !visit[j]) { DFS(j); fout<<'\n'<<setw(2)<<i<<"
";}
  }
 }
void main()
{
  char ime_vh_fl[21]; unsigned k,sv;
  cout<<"\nИме на входния файл: "; cin>>ime_vh_fl;
  fin.open(ime_vh_fl,ios::in);
  if (fin.bad()) {cout<<"Няма такъв файл.\n";getch();return;}
  fin>>BrV; //Въвеждаме от текстовия файл броя на върховете на графа
  clasGraph Graph;         //Създава обекта Graph
  Graph.make_graf();       //Запълва матрицата на съседство
  Graph.display_graf();    //Извежда матр. на съседство на екрана
  Graph.display_graf(fout);//Извежда матр. на съседство в т. файл
  if(!(visit=new unsigned[BrV])) exit(1);
  for (k=0; k<BrV; k++) visit[k]=0;
```

```
  fout.open("DopInf",ios::ate);
  cout<<"Стартов връх: "; cin>>sv;//Въвеждаме стартовия връх
  cout<<"Обхождане в дълбочина от стартов връх "<<sv<<" -> ";
  Graph.DFS(sv);
  fout.close();
  cout<<"\nДопълнителна информация можете да намерите във файл
\"DopInf.\"\n";
}
```

Както се вижда, рекурсивна член-функция DFS, която реализира обхождането в дълбочина е много кратка. Тя ще стане още по-кратка, ако отстраним операторите, които извеждат допълнителна информация в текстовия файл "DopInf". Тази информация позволява по-лесно да се проследи изпълнението на член-функцията DFS и е предназначена за онези читатели, които се затрудняват, когато четат програми, с рекурсивни функции.

Програмата използва матрицата на съседство и масива visit, показващ във всеки момент от изпълнението на програмата кои върхове са посетени и кои не са.

Главната функция се обръща към член-функцията DFS, като замества фиктивния аргумент i с номера на стартовия връх. Започва първото изпълнение на член-функцията DFS. Тя отбелязва стартовия връх, като посетен, извежда го на екрана (в списъка на обходените върхове) и започва да търси в i-ия ред на матрицата на съседство непосетен съсед на i-ия връх. Щом се намери такъв връх, функцията се обръща към себе си, като замества фиктивния аргумент i с номера j на намерения непосетен съседен връх.



Фиг. 32. Неориентиран безтегловен граф за илюстрация на обхождане в дълбочина

С това временно се спира първото изпълнението на DFS и започва новото изпълнение на рекурсивната функция. Отбелязва се i-ия връх (това i няма нищо общо с i-то от първото изпълнение на рекурсивната функция) като посетен, извежда се номерът му на екрана и започва търсене на негов непосетен съсед. Ако се намери такъв, следва ново обръщение към рекурсивната функция, но ако не се намери, следва излизане от изпълняваното в момента обръщение към DFS.

Тук са дадени матрицата на съседство и допълнителната информация, която е изведена в текстов файл при изпълнение на програмата за графа, даден на фиг. 32. В във всеки ред в първата колона на допълнителната информация се намира номерът на върха, за които се търси непосетен съсед при съответното рекурсивно обръщение. От 6-ия ред става ясно, че 4-ия връх няма непосетени съседи и затова не следва ново обръщение, а напротив, излизане от текущата рекурсия. От 7-ия ред става ясно, че продължава изпълнението на предходното обръщение, т.е. продължава търсенето на непосетен съсед на 3-ия връх. Такъв не се намира и следва излизане и от това обръщение. От 8-ия ред се вижда, че се търси непосетен съсед на 1-ия връх. Такъв се оказва 6-ия връх и следва обръщение към DFS, за да

	0	1	2	3	4	5	6	7
0	0	0	1	0	0	1	0	1
1	0	0	0	1	1	1	1	0
2	1	0	0	1	0	0	0	1
3	0	1	1	0	1	0	0	0
4	0	1	0	1	0	0	0	0
5	1	1	0	0	0	0	1	0
6	0	1	0	0	0	1	0	0
7	1	0	1	0	0	0	0	0

Матрица на съседство

се търси непосетен съсед на 6-ия връх и т.н.

Следващата информация проследява реализираните обръщения към рекурсивната функция:

2 0 Търсене непосетен съсед на 2-ия. Такъв е 0-ия. Ново обръщение към рекурсивната функция.

0 0 1 2 3 4 5 Търсене непосетен съсед на 0-ия. Такъв е 5-ия. Ново обръщение към рекурсивната функция.

5 0 1 Търсене непосетен съсед на 5-ия. Такъв е 1-ия. Ново обръщение към рекурсивната функция.

1 0 1 2 3 Търсене непосетен съсед на 1-ия. Такъв е 3-ия. Ново обръщение към рекурсивната функция.

3 0 1 2 3 4 Търсене непосетен съсед на 3-ия. Такъв е 4-ия. Ново обръщение към рекурсивната функция.

4 0 1 2 3 4 5 6 7 Търсене непосетен съсед на 4-ия. Няма. Връщане от рекурсивната функция.

3 5 6 7 Търсене непосетен съсед на 3-ия. Няма. Връщане от рекурсивната функция.

1 4 5 6 Търсене непосетен съсед на 1-ия. Такъв е 6-ия. Ново обръщение към рекурсивната функция.

6 0 1 2 3 4 5 6 7 Търсене непосетен съсед на 6-ия. Няма. Връщане от рекурсивната функция.

1 7 Търсене непосетен съсед на 1-ия. Няма. Връщане от рекурсивната функция.

5 2 3 4 5 6 7 Търсене непосетен съсед на 5-ия. Няма. Връщане от рекурсивната функция.

0 6 7 Търсене непосетен съсед на 0-ия. Такъв е 7-ия. Ново обръщение към рекурсивната функция.

7 0 1 2 3 4 5 6 7 Търсене непосетен съсед на 7-ия. Няма. Връщане от рекурсивната функция.

0 Търсене непосетен съсед на 0-ия. Няма. Връщане от рекурсивната функция.

2 1 2 3 4 5 6 7 Търсене непосетен съсед на 2-ия. Няма. Връщане от рекурсивната функция.

Намерен ред на обхождане: 2, 0, 5, 1, 3, 4, 6, 7.

Програма 5.10. Съдържа рекурсивна член-функция за обхождане в дълбочина на граф, представен чрез масив от списъци на съседство

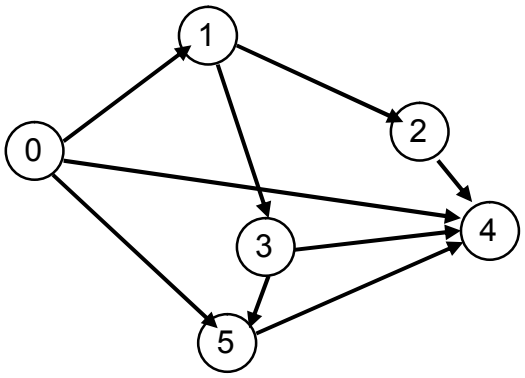
```
# include <iostream.h>
# include <conio.h>
# include "graf_sp.h"
unsigned *visit;
//Функция за обхождане в дълбочина от даден врах
void clasGraph::DFS(unsigned i)
 {
  unsigned j;
  visit[i]=1;
  cout<<i<<' ';
  rebro *p=a[i].first;
  while (p){
     j=p->nom;
     if (visit[j]) p=p->next;
     else DFS(j);
  }
 }
void main()
```

```
{
  char ime_vh_fl[21]; int sv;
  cout<<"\nИме на входния файл: "; cin>>ime_vh_fl;
  fin.open(ime_vh_fl,ios::in);
  if (fin.bad()) {cout<<"Няма такъв файл.\n";getch();return;}
  fin>>BrV; //Въвеждаме броя на върховете от текстовия файл
  clasGraph Graph;
  Graph.make_graf();
  Graph.display_graf();
  Graph.display_graf(fout);
  cout<<"Стартов връх: "; cin>>sv;
  cout<<"Обхождане в дълбочина от стартов връх "<<sv<<" -> ";
  if (!(visit=new unsigned[BrV])) exit(0);
  for (int i=0;i<BrV;i++) visit[i]=0;
  Graph.DFS(sv); cout<<endl;
}
```

Новата програма се различава от предходната само по това, че съседите на даден връх се търсят в масива от списъци на съседство, а не в матрицата на съседство.

5.6. Топологично сортиране на граф

Топологичното сортиране на граф е операция, която се изпълнява само за ориентирани ациклични графи. В резултат на изпълнението на тази операция се получава редица от номера на върхове, наречена топологичен ред или топологична подредба. В топологичния ред номерата на всички върхове, преки предшественици на даден връх, са преди номера на този връх. На един граф често съответстват няколко топологични реда, но обикновено е достатъчно да се намери един от тях.



Фиг. 33. Граф за илюстрация на топологично сортиране

Топологичен ред се получава по следния алгоритъм:

1. Ако графът няма върхове без предшественици, той е цикличен и не подлежи на топологично сортиране.

2. Един по един, в произволен ред записваме в топологичния ред върховете без предшественици и ги изтриваме.

3. Ако още има неизтрити върхове, преминаваме към операция 1, ако са изтрити всички, топологичното сортиране е завършило.

Нека да приложим този алгоритъм за графа от фиг. 33.

1. Без предшественици е връх 0. Записваме го в топологичния ред и го изтриваме.

2. Без предшественици е връх 1. Записваме го топологичния ред и го изтриваме.

3. Сега без предшественици са върхове 2 и 3. Записваме в топологичния ред и изтриваме първо връх 2 и след това връх 3, но можем да го направим и в обратен ред.

4. След изтриването на върхове 2 и 3, без предшественици остава връх 5. Записваме го в топологичния ред и го изтриваме.

5. Сега без предшественици е връх 4. Записваме го в топологичния ред и го изтриваме.

Всички върхове са изтрити. Един топологичен ред за този граф е: 0, 1, 2, 3, 5, 4. Графът е ацикличен.

Сега сменете посоката на ребро ⟨3, 4⟩ и се опитайте да намерите топологичен ред на променения граф.

Разгледаният алгоритъм е прост и ясен, но той не е компютърен алгоритъм.

Компютърният алгоритъм трябва да ползва вътрешното представяне на графа, например матрицата на съседство. Ако искаме да следваме описания алгоритъм точно, ще трябва след всяко изтриване на връх да променяме матрицата на съседство. По този начин създаването на топологичен ред е съпроводено от унищожаване на матрицата на съседство, което често не е желателно.

Има друг по-добър алгоритъм. Той ползва масивите: `visit[]` за регистрация на посетените върхове и `br_pred[]` за текущия брой предшественици на всеки връх и брояча на посетените върхове `br_visit`.

Алгоритъмът се състои в следното:

1. Регистрират се всички върхове като непосетени.
2. Намира се входната степен (преброяват се предшествениците) на всеки връх.
3. Всички върхове без предшественици се изпращат в опашката на чакащите върхове.
4. Един по един се вземат чакащите върхове от опашката, извеждат се в топологичния ред, регистрират се като посетени, преброяват се като посетени, намалява се входната степен на всеки техен наследник, всички наследници, останали с входна степен 0, се изпращат в опашката. Това продължава докато в опашката има чакащи върхове. Когато опашката остане празна има две възможности:
   - посетени са всички върхове и топологичният ред е създаден;
   - посетени са само част от върховете – графът е цикличен и не се поддава на топологично сортиране.

Следват две програми за топологично сортиране.

Програма 5.11. Топологично сортиране на граф, представен чрез матрица на съседство

```
# include <iostream.h>
# include <conio.h>
#include "graf_mat.h"

void clasGraph::top_sort()
{
 unsigned i,j,k,n,
 br_visit=0, // Брой на посетените върхове
 *visit,     // За масива, отчитащ посетеността на върховете
 *br_pred;   // За масива, отчитащ броя на предшествениците на всеки
връх
 if (!(visit=new unsigned[BrV])) exit(0);
 if (!(br_pred=new unsigned[BrV])) exit(0);
 for (i=0;i<BrV;i++) {visit[i]=0;br_pred[i]=0;}
 //Намираме броя на предшествениците на всеки връх
 //и изпращаме елементите без предшественици в опашката
 for (j=0;j<BrV;j++){
  for (i=0;i<BrV;i++) if (A[i][j]) br_pred[j]++;
  if (br_pred[j]==0) Opashka.DobEl(j);
 }
  //Генериране на топологичен ред
 cout<<"Топологичен ред: ";
 while (Opashka.Br) { //Докато опашката не е празна,...
  n=Opashka.Br;
  for (i=0;i<n;i++){
     Opashka.VzeEl(k);//... вземаме връх от опашката ...
     cout<<k<<' ';    //... извеждаме го,...
```

```
     visit[k]=1;      //... маркираме го като посетен и...
     br_visit++;      //... го преброяваме, като посетен.
     for (j=0;j<BrV;j++)
      if (A[k][j]!=0 && visit[j]==0){ //На непосетените наследници на
к-ия връх ...
          br_pred[j]--;     //... намаляваме входната степен и...
          if (br_pred[j]==0) Opashka.DobEl(j); //ако стане 0, пращаме
го в опашката
      }
  }
 }
 if (br_visit<BrV) cout<<"Графът е цикличен.\n";
}
void main()
{
  char ime_vh_fl[21];
  cout<<"\nИме на входния файл: "; cin>>ime_vh_fl;
  fin.open(ime_vh_fl,ios::in);
  if (fin.bad()) {cout<<"Няма такъв файл.\n";getch();return;}
  fin>>BrV; //Въвежда броя на върховете от текстовия файл
  clasGraph Graph;
  Graph.make_graf();
  Graph.display_graf();
  Graph.top_sort();
}
```

Програма 5.12. Топологично сортиране на граф, представен чрез масив от списъци на съседство

```
#include <fstream.h>
#include <conio.h>
#include "graf_sp.h"
//Член-функция за топологично сортиране.
void clasGraph::top_sort()
{
  unsigned i,j,k,n,br_visit=0,*br_pred, *visit; rebro* p;//br_visit -
брой посетени
  if (!(br_pred=new unsigned[BrV])) exit(0);
  if (!(visit=new unsigned[BrV])) exit(0);
  for (i=0;i<BrV;i++) { br_pred[i]=0;visit[i]=0;}
// Преброяване на предшествениците
  for (i=0;i<BrV;i++) {
      p=a[i].first;
      while (p) {
       j=p->nom;
       br_pred[j]++;//br_pred[j] е брой на предшествениците на j-ия
връх.
       p=p->next;
      }
  }
//Изпращаме в опашката всички елементи без предшественици
  for (i=0;i<BrV;i++) if (br_pred[i]==0) Opashka.DobEl(i);
  cout<<"Топологичен ред: ";
//    Докато опашката не е празна, вземаме връх, извеждаме го и
намаляваме
```

```
//с 1 входната степен на всички негови непосредствени наследници. Щом
обновената
//входна степен на някой наследник стане нула, изпращаме го в опашката
  while (Opashka.Br) { //Докато опашката не е празна,...
      n=Opashka.Br;
      for (i=0;i<n;i++) {
          Opashka.VzeEl(j); //...вземаме връх от опашката,...
          cout<<j<<' '; visit[j]=1;
          br_visit++;
          p=a[j].first;
          while (p) {
     k=p->nom;       // k - номер на поредния наследник
     if (visit[k]==0){
      br_pred[k]--;//... намаляваме входната степен на този
наследник,...
      if (br_pred[k]==0)   //...и ако тя стане 0,...
       Opashka.DobEl(k);//...изпрщаме го в опашката от върхове без
предшественици...
     }
     p=p->next;
          }
      }
  }
  if (br_visit<BrV) cout<<"\nВ графа има цикли."<<endl;
}
void main()
{
  char ime_vh_fl[21];
  cout<<"\nИме на входния файл: "; cin>>ime_vh_fl;
  fin.open(ime_vh_fl,ios::in);
  if (fin.bad()) {cout<<"Няма такъв файл.\n";getch();return;}
  fin>>BrV; //Въвежда броя на върховете от текстовия файл
  clasGraph Graph;
  Graph.make_graf();
  Graph.display_graf();
  Graph.top_sort(); cout<<endl;
}
```

След като се запознахме с основните операции с графи, ще ги изпозване при решаването на някои приложни задачи. Ще започнем с една популярна група задачи, свързана с намиране на пътища в граф.

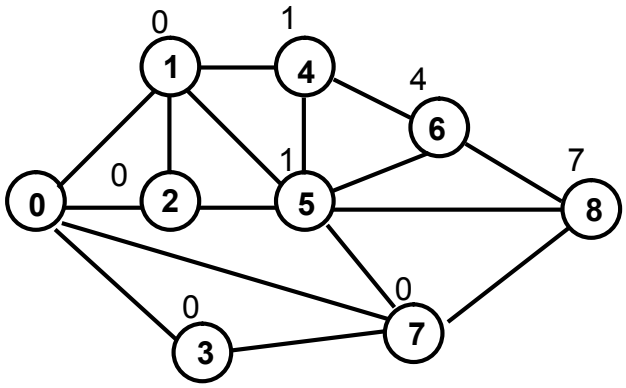
5.7. Намиране най-кратък път от зададен връх до всички други върхове в безтегловен граф

Път между два върха (начален и краен) в граф наричаме последователността от свързаните върхове, които се намират между двата върха. Дължината на пътя между два върха се измерва с броя на върховете, през които минава пътят, включително началния и крайния връх.

Между два върха може да има много пътища, само един път или изобщо да няма път. Когато има много пътища, най-кратък е този, който включва най-малко върхове.

Алгоритъм за намиране най-кратък път между два върха в безтегловен граф можем да създадем като използваме алгоритъма за обхождане в ширина, тъй като то е обхождане по нива.

Нека да си поставим задачата да намерим най-краткия път от 0-ия връх до някой друг връх от графа, показан на фиг. 34. За целта ще обходим графа в ширина, като започнем от 0-ия връх, който е на 0-во ниво.



Фиг. 34. Намиране на най-кратък път в безтегловен граф

На първо ниво са всички непосетени върхове (1, 2, 3 и 7), съседи на 0-ия връх. На тях 0-ия връх е предшественик и на фиг. 34 това е показано с нулите, изписани извън кръгчетата на тези върхове.

На второ ниво са върховете 4 и 5, като непосетени съседи на върха 1, и връх 8, като непосетен съсед на 7.

На трето ниво е само връх 6 и негов предшественик е върхът 4.

Сега вече можем да намерим най-краткия път от 0-ия връх до всеки от останалите. Например най-краткия път от 0-ия 6-ия връх ще намерим по следния начин:

- изпращаме в стек: крайния връх, т.е. връх 6, неговия предшественик, т.е. връх 4, предшественика на връх 4, т.е. връх 1, предшественика на връх 1, т.е. връх 0. Тъй като 0-ия връх нама предшественик, той е началния връх. В стека, от върха към дъното, се намират върховете: 0, 1, 4 и 6
- вземаме върховете от стека и ги записваме в редицата, образуваща пътя от 0-ия 6-ия връх. Ще получим, че търсеният път е редицата 0, 1, 4 и 6.

Програма за намиране на най-кратките пътища от зададен начален връх до всички останали върхове можем да направим, като модифицираме програмата за **обхождане в ширина** така, че в края на обхождането да е известен предшественикът на всеки връх. Това можем да постигнем като въведем масив на предшествениците `pred[]`. Елементите му ще инициализираме с –1, а по време на изпълнение на програмата щом даден връх се сдобие с предшественик, отбелязваме това в масива. Ако в края на обхождането `pred[5]`=3, то това ще означава, че връх 3 е предшественик на връх 5. Ако обаче `pred[5]`=-1, то това ще означава, че връх 5 е без предшественик, т.е. той е началният връх. От този масив можем да намерим най-краткия път от който и да е краен връх до началния връх, т.е. да намерим обратния път. Това става по следния начин. Записваме крайния връх, от масива `pred[]` намираме неговия предшественик и го записваме, пак от масива `pred[]` намираме предшественика на току що записания връх и т. н. докато стигнем до началния връх. За да получим пътя от началния до крайния връх е целесъобразно да използваме стек. Тръгвайки от крайния връх, изпращаме в стека всеки връх от пътя. Като стигнем до началния връх, всички върхове от пътя са в стека. Изваждайки ги от стека ще получим пътя в прав ред

Следват две програми за намиране на най-кратък път в безтегловен граф.

Програма 5.13. Намиране на най-кратък път между два върха в безтегловен граф, представен с матрица на съседство

```
# include <iostream.h>
# include <conio.h>
#include "graf_mat.h"
void clasGraph::kraBFS()
{
```

```
  unsigned i,j,
           nv,       //Начален връх
           kv,       //Краен връх
           v,        //Текущ връх
           *visit;   //Масив, отчитащ посетеността на върховете
  int* pred;         //Масив с предшествениците на върховете при
обхождането
  if (!(visit=new unsigned[BrV])) exit(0);
  if (!(pred=new int[BrV])) exit(0);
  for (i=0;i<BrV;i++) { visit[i]=0; pred[i]=-1; }
  fout.open("DopInf",ios::ate);
  fout<<"\nСъдържание на масива на предшествениците след поредна
стъпка:\n";
  cout<<"Начален връх: "; cin>>nv;
  cout<<"Краен връх: "; cin>>kv;
  Opashka.DobEl(nv);//Добавяме към опашката стартовия елемент
  visit[nv]=1;
  while (Opashka.Br>0) {//Докато опашката не е празна ...
      Opashka.VzeEl(v);  //... вземаме поредния връх от опашката,...
      //...добавяме към опашката необходените му наследници,...
      //...отбелязваме ги, като обходени, и записваме предшественика
им.
      for (j=0;j<BrV;j++)
       if (A[v][j]==1 && visit[j]==0)
{pred[j]=v;visit[j]=1;Opashka.DobEl(j);}
      for (j=0;j<BrV;j++) fout<<setw(3)<<j;fout<<endl;
      for (j=0;j<BrV;j++) fout<<setw(3)<<pred[j];fout<<endl<<endl;
  }
  if (pred[kv]==-1) {cout<<"Не е намерен път!\n"; getch();return;}
//Съществува път между двата върха. Получаваме го от масива на
предшествениците
//по следния начин: изпращаме в стека крайния връх, изпращаме там
неговия
//предшественик и т.н. докато изпратим там и началния връх. След това
вземаме
//върховете от стека и ги извеждаме на екрана. Изведената редица е
пътят.
  Stek.DobEl(kv);
  v=kv;
  while (pred[v] != -1){v=pred[v]; Stek.DobEl(v);}
  unsigned L=Stek.Br;  //Дължина на пътя
//Извеждане на получения път на екрана
  cout<<"Най-кратъкиЯт път от връх "<<nv<<" до връх "<<kv<<" е ";
  while (Stek.Br) {Stek.VzeEl(v); cout<<v<<' ';}
  cout<<"\b, дължината му е "<<L<<".\n";
  fout.close();
}

void main()
{
  char ime_vh_fl[21];
  cout<<"\nИме на входния файл: "; cin>>ime_vh_fl;
  fin.open(ime_vh_fl,ios::in);
  if (fin.bad()) {cout<<"Няма такъв файл.\n";getch();return;}
  fin>>BrV; //Въвеждаме от текстовия файл броя на върховете на графа
  clasGraph Graph;       //Създава обекта Graph
```

```
  Graph.make_graf();          //Запълва матрицата на съседство
  Graph.display_graf();       //Извежда матрицата на съседство на
екрана
  Graph.display_graf(fout);   //Извежда матр. на съседство в
текст.файл
  Graph.kraBFS();             //Намира и извежда най-краткия път
  cout<<"\nДопълнителна информация можете да намерите във файл
DopInf.\n";
}
```

Програма 5.14. Намиране на най-кратък път между два върха в безтегловен граф, представен с масив от списъци на съседство

```
# include <iostream.h>
# include <conio.h>
# include "graf_sp.h"
//Функция за обхождане в ширина от даден връх, модифицирана да запомня
на предшественика на всеки връх
void clasGraph::kraBFS()
{
  unsigned i,j,
          nv, //Начален връх
          kv, //Краен връх
          v,  //Текущ връх
          *visit;// Масив, отчитащ посетеността на върховете
  int* pred;// Масив с предшествениците на върховете при обхождането
  rebro *p; // Указател към обект ребро
  if (!(visit=new unsigned[BrV])) exit(0);
  if (!(pred=new int[BrV])) exit(0);
  for (i=0;i<BrV;i++) { visit[i]=0; pred[i]=-1; }
  fout.open("DopInf",ios::ate);
  fout<<"\n//Масивът на предшествениците след поредна стъпка:\n";
  cout<<"Начален връх: "; cin>>nv;
  cout<<"Краен връх: ";   cin>>kv;
  Opashka.DobEl(nv);//Добавяме към опашката началния елемент
  visit[nv]=1;
  while (Opashka.Br>0) {//Докато опашката не е празна ...
      Opashka.VzeEl(v);//... вземаме поредния връх от опашката ...
      p=a[v].first;
      while (p){
      //...и добавяме към опашката необходените му наследници,...
      //...отбелязваме ги, като обходени, и записваме предшественика
им.
       j=p->nom;
       if (!visit[j]) {Opashka.DobEl(j); visit[j]=1; pred[j]=v;}
       p=p->next;
  }
     for (j=0;j<BrV;j++) fout<<setw(3)<<j;fout<<endl;
     for (j=0;j<BrV;j++) fout<<setw(3)<<pred[j];fout<<endl<<endl;
  }
  if (pred[kv]==-1) {cout<<"Не е намерен път!\n"; getch();return;}
//Съществува път между двата върха. Получаваме го от масива на
предшествениците
//по следния начин: изпращаме в стека крайния връх, изпращаме там
неговия
```

```
                    //предшественик и т.н. докато изпратим там и
    0 1 2 5 4 6 8   началния връх. След това вземаме
    0 1 2 5 6 8     //върховете от стека и ги извеждаме на екрана.
    0 1 2 5 7 8     Изведената редица е пътят.
    0 1 2 5 8             Stek.DobEl(kv);
    0 1 4 5 6 8           v=kv;
    0 1 4 5 7 8           while (pred[v] != -1){v=pred[v];
    0 1 4 5 8       Stek.DobEl(v);}
                          unsigned L=Stek.Br;
                          //Извеждане на получения път на екрана
      cout<<"Най-кратъкият път от връх "<<nv<<" до връх "<<kv<<" е ";
      while (Stek.Br) {
          Stek.VzeEl(v); cout<<v<<' ';
      }
      cout<<"\b, дължината му е "<<L<<".\n";
 fout.close();
}

void main()
{
  char ime_vh_fl[21];
  cout<<"\nИме на входния файл: "; cin>>ime_vh_fl;
  fin.open(ime_vh_fl,ios::in);
  if (fin.bad()) {cout<<"Няма такъв файл.\n";getch();return;}
  fin>>BrV; //Въвеждаме от текстовия файл броя на върховете на графа
  clasGraph Graph;     //Създава обекта Graph
  Graph.make_graf();       //Запълва матрицата на съседство
  Graph.display_graf();     //Извежда матрицата на съседство на екрана
  Graph.display_graf(fout);//Извежда матрицата на съседство в текст.
файл
  Graph.kraBFS();          //Намира и извежда най-краткия път
  cout<<"\nДопълнителна информация можете да намерите във файл
DopInf.\n";
}
```

5.8. Намиране на всички прости пътища между два върха в безтегловен граф

Ще припомним, че прост път между два върха е всеки път между тях, който не съдържа цикли. В графа, даден на фиг. 34, могат да се намерят 41 прости пътя между връх 0 и връх 8. Ето една част от тях:

Нека да се опитаме да потърсим някои от тези пътища.

Тръгваме от началния връх 0. Можем да преминем във всеки от върховете 1, 2, 3 и 7. Избираме да преминем във връх 1, защото е с най-малък номер. От връх 1 можем да преминем във всеки от върховете 2, 4 и 5. Избираме да преминем във връх 2, защото е с най-малък номер. От връх 2 можем да преминем само във връх 5. От връх 5 можем да преминем във всеки от върховете 4, 6, 7 и 8. Избираме да преминем във връх 4, защото е с най-малък номер. От връх 4 преминаваме в 6, защото нямаме друга възможност. От връх 6 преминаваме в 8, защото отново нямаме друга възможност и с това е намерен първият прост път: 0, 1, 2, 5, 4, 6, 8.

За да намерим втори път, ще се върнем с една стъпка назад, т.е. във връх 6. Тъй като връх от 6 не може да се продължи по нов път към 8, ще се върнем с още една стъпка назад, т.е. във връх 4. И от връх 4 не можем да продължим по нов път към 8, затова ще се върнем с още една стъпка назад, т.е. във връх 5. От връх 5 можем да преминем във всеки от върховете 6, 7 и 8. Избираме да преминем във връх 6, защото е с най-малък

номер. От връх 6 преминаваме във връх 8, защото нямаме друга възможност и с това е намерен вторият прост път: 0, 1, 2, 5, 6, 8.

За да намерим трети път, ще се върнем с една стъпка назад, т.е. във връх 6. Тъй като връх от 6 не може да се продължи по нов път към 8, ще се върнем с още една стъпка назад, т.е. във връх 5. От връх 5 можем да преминем само във връх 7 и 8. Избираме да преминем във връх 7, защото е с най-малък номер. От връх 7 преминаваме във връх 8, защото нямаме друга възможност, и с това е намерен третият прост път: 0, 1, 2, 5, 7, 8.

За да намерим четвърти път, ще се върнем с една стъпка назад, т.е. във връх 7. Тъй като връх от 7 не може да се продължи по нов път към 8, ще се върнем с още една стъпка назад, т.е. във връх 5. От връх 7 преминаваме във връх 8, защото нямаме друга възможност, и с това е намерен четвъртият прост път: 0, 1, 2, 5, 8.

От изложеното се разбира, че, за да създадем алгоритъм за намиране на всички прости пътища, трябва да променим алгоритъма за обхождане в дълбочина, като направим следните промени:

1. Обхождането се прекратява, когато се достигне крайният връх, а не когато се обходят всички върхове, както беше при обхождането в дълбочина. Това налага рекурсивната функция да има и втори аргумент, крайния връх.

2. Когато се налага да се отстъпва назад по графа с цел откриване на необходени клонове, ще отменяме регистрацията за посетеност на върха, от които отстъпваме. Това се налага, защото върхът, от който отстъпваме, може да участва в следващ път.

3. Ще регистрираме в масива `path[]` всеки връх, включен в поредния генериран път. За индекс на елементите на масива `path[]` ще ползваме нивото `(nivo)`, на което се намира всеки от тях.

4. След всяко достигане до крайния връх, т.е. намиране на нов път, ще го извеждаме.

Програма 5.15. Намиране на всички прости пътища между два върха от граф, представен с матрица на съседство

```
#include <fstream.h>
#include <iomanip.h>
#include <conio.h>
#include "graf_mat.h"
unsigned *visit,
         *path;      //Масив път, в който се регистрира намереният път
unsigned nivo;       //Ниво, на което се намира посетеният връх,
използва се за индекс
                     //на елементите от масива път
//Член-функция за намиране на всички прости пътища между върховете  i
и j
void clasGraph::vsiDFS(unsigned i, unsigned j)
 {
  unsigned k;
  if (i!=j){         //Крайният връх още не е достигнат.
     visit[i]=1;     //Маркиране на i-ия връх като посетен.
     path[nivo++]=i;//Включване i-ия връх в масива път
     for (k=0;k<BrV;k++) //Рекурсия за всички непосетени съседи на i
      if (A[i][k] && !visit[k]) vsiDFS(k,j); //След връщане от vsiDFS:
      visit[i]=0;    // отмяна маркировката на върха, като посетен
и...
      nivo--;        //... изключване на върха от масива път
      return;
  }
     path[nivo]=i;  //Записване на връх i като краен в масива път
     for (k=0;k<=nivo;k++) cout<<path[k]<<' ';//Извеждане на намерения
път
     cout<<endl;
```

```
 }

 void main()
{
  char ime_vh_fl[21];unsigned i,k;
  unsigned  nv; //Стартов връх при обхождането
  unsigned  kv; //Краен връх при обхождането
  cout<<"\nИме на входния файл: "; cin>>ime_vh_fl;
  fin.open(ime_vh_fl,ios::in);
  if (fin.bad()) {cout<<"Няма такъв файл.\n";getch();return;}
  fin>>BrV; //Въвеждаме от текстовия файл броя на върховете на графа
  clasGraph Graph;  //Създаваме обекта Graph
  Graph.make_graf();    //Запълваме матрицата на съседство
  Graph.display_graf(); //Извеждаме на екрана матрицата на съседство
  cout<<endl;
  if (!(visit=new unsigned[BrV])) exit(0);
  for (k=0;k<BrV;k++) visit[k]=0;
  if (!(path=new unsigned[BrV])) exit(0);
  nivo=0;
  cout<<"Стартов връх: "; cin>>nv;
  cout<<"Краен връх:"; cin>>kv;
  cout<<"Простите пътища между върховете "<<nv<<" и "<<kv<<"
са:"<<endl;
  Graph.vsiDFS(nv,kv);
  cout<<endl;
}
```

Програма 5.16. Намиране на всички прости пътища между два върха от граф, представен с масив от списъци на съседство

```
//Програма за намиране на всички прости пътища между два върха
#include <fstream.h>
#include <iomanip.h>
#include <conio.h>
#include "graf_sp.h"
unsigned *visit,
      *path;        //Масив път, в който се регистрира намерения път
unsigned nivo; //Ниво, на което се намира посетения връх. Използва се
//за индекс на елементите от масива път
//Член-функция за намиране на всички прости пътишта между върховете  i
и j
void clasGraph::vsiDFS(unsigned i, unsigned j)
 {
  unsigned k,m;
  if (i!=j)  {           //Крайният връх още не е достигнат.
      visit[i]=1;        //Маркиране на i-ия връх като посетен.
      path[nivo++]=i;    //Включване на i-ия връх в масива път
      rebro *p=a[i].first;
      while (p){
       k=p->nom;
       if (!visit[k]) vsiDFS(k,j);//След връщане от vsiDFS:
       p=p->next;
      }
      visit[i]=0;   // отмяна маркировката на врърха, като посетен
и...
      nivo--;       //...изключване на върха от масива път
      return;
```

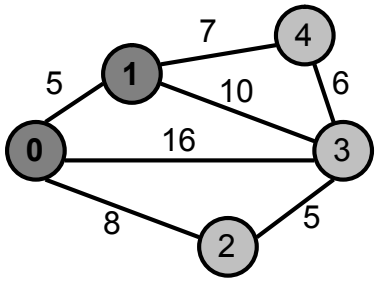
```
  }
  path[nivo]=i;     //Записване на връх i като краен в масива път
  for (k=0;k<=nivo;k++) cout<<path[k]<<' ';//Извеждане на намерен път
      cout<<endl;
  }

 void main()
{
  char ime_vh_fl[21];unsigned i,k;
  unsigned  nv; //Стартов връх при обхождането
  unsigned  kv; //Краен връх при обхождането
  cout<<"\nИме на входния файл: "; cin>>ime_vh_fl;
  fin.open(ime_vh_fl,ios::in);
  if (fin.bad()) {cout<<"Няма такъв файл.\n";getch();return;}
  fin>>BrV; //Въвеждаме от текстовия файл броя на върховете на графа
  clasGraph Graph;        //Създава обекта Graph
  Graph.make_graf();    //Запълва матрицата на съседство
  Graph.display_graf(); //Извежда матрицата на съседство на екрана
  cout<<endl;
  if (!(visit=new unsigned[BrV])) exit(0); for (k=0;k<BrV;k++)
visit[k]=0;
  if (!(path=new unsigned[BrV])) exit(0);
  nivo=0;
  cout<<"Начален връх: "; cin>>nv;
  cout<<"Краен връх: "; cin>>kv;
  cout<<"Простите пътища между върховете "<<nv<<" и "<<kv<<"
са:"<<endl;
  Graph.vsiDFS(nv,kv);
  cout<<endl;
}
```

5.9. Намиране най-кратък път от даден връх до всички други върхове в тегловен граф

В гл. 7 разгледахме задачата за намиране най-кратък път от даден връх до всички други върхове в безтегловен граф. Решението на тази задача беше едно приложение на обхождането на граф в ширина. Новата задача е по сложна, но тя има по голямо приложение. Едно от приложенията й е намирането на най-кратките пътища от едно населено място до останалите от дадена селищна система.

Сред алгоритмите за решаване на тази задача за най-ефективен се счита алгоритъмът на **Дейкстра**, създаден през 1959 г. Тук ще се въздържим от неговото строго математическо представяне и ще се опитаме да го обясним с помощта на един пример, графа от фиг. 9. Нека да приемем, че искаме да намерим най-късите пътища от връх 0 до всички останали. Алгоритъмът се състои в следното:

1. Установяваме се в началния връх и:
   - регистрираме го като посетен;
   - намираме дължината на пътя от него до всеки от непосетените му съседи. Такива са върховете 1, 2 и 3, а дължината на пътя до всеки от тях са съответно 5, 8 и 16;
   - регистрираме връх 0, като предшественик на върховете 1, 2 и 3;
   - намираме най-близкия връх до началния, в случая такъв е връх 1 и разстоянието до него е 5.



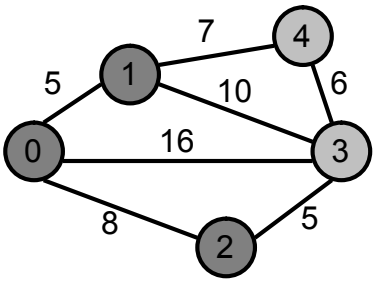
2. Преместваме се във връх 1, защото от непосетените съседи на връх 0, той е най-близко до началния връх и:
   - регистрираме връх 1, като посетен;

- намираме дължината на пътя от началния връх до непосетените съседи на връх 1, но по пътя до тях през връх 1 и:

· за връх 3: дължината на пътя до него през връх 1 е 5+10=15. Този път е по-кратък от вече известния път 0-3 с дължина 16. Затова записваме, че дължината на най-краткият път до връх 3 е 15, и регистрираме връх 1, като предшественик на връх 3. С това връх 0 отпадна като предшественик на връх 3;

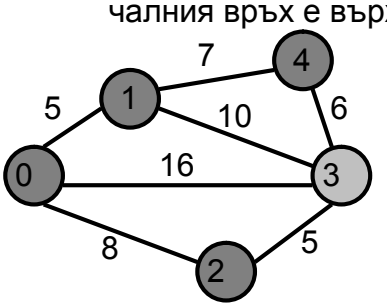
· за връх 4: дължината на пътя до него през връх 1 е 5+7=12, за предшественик на връх 4 ще запишем връх 1.

- намираме, че от непосетените върхове 2, 3 и 4 на най-кратко разстояние от началния връх се намира върхът 2.



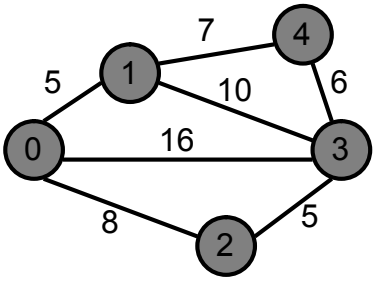
3. Преместваме се във връх 2, защото, от непосетените, той е на най-кратко разстояние от чалния връх 0 и:

- регистрираме връх 2 като посетен;
- намираме дължината на пътя от началния връх до непосетените съседи на връх 2, но по пътя през връх 2. Непосетен съсед на 2 е само връх 3, а дължината на пътя до връх 3 през връх 2 е 8+5=13. Пътят 0-2-3 с дължина 13 е по-кратък пътя 0-1-3 с дължина 15 и затова връх 2 ще измести връх 1 като предшественик на връх 3.
- намираме, че от непосетените върхове 3 и 4 на по-кратко разстояние от началния връх е върхът 4. Минималният път на пътя до връх 3 е с дължината 13.



4. Преместваме се във връх 4, защото от непосетените той е най-близко до началния връх и:

- регистрираме връх 4, като посетен;
- намираме дължината на пътя от началния връх до непосетения съсед на връх 4, но по пътя през връх 4. Непосетен съсед на 4 е само връх 3, а дължината на пътя от началния връх до връх 3 през връх 4, т.е.пътя 0-1-4-3, е 12+6=18. Този път не е по-кратък от вече намерения път 0-2-3 с дължина 13. Новият път 0-1-4-3 няма да елиминира стария път 0-2-3, а връх 4 няма да измести връх 2 като предшественик на връх 3.



5. Преместваме се във връх 3, защото само той не е посетен. Регистрираме го като посетен и приключваме, защото той няма непосетени съседи.

Фиг. 35. Реализация на алготитъма на Дейкстра

Препоръчваме на читателя да намери по описания алгоритъм най-кратките пътища от връх 1, 2, 3 и 4 до всички останали.

Разгледаният алгоритъм е прост и ясен.

Програмата по този алгоритъм ползва вътрешното представяне на графа (матрица на съседство или масив от списъци на съседство), за търсене съседите на поредния връх. За регистрацията на посетените върхове, предшествениците на върховете в намерените пътища и дължините на намерените пътища, ще използваме три масива: `visit[]`, `pred[]` и `dist[]`. Тук новост е само масивът `dist[]`. Масивите `visit[]` и `pred[]` вече сме ползвали.

Масивът `visit[]` инициализираме в началото с нули, т.е. обявяваме всички върхове за непосетени, а когато посетим например `i`-ия връх, на `visit[i]` присвояваме 1.

Масивът `dist[]` инициализираме в началото с достатъчно голямо число, например 9999, т.е. в началото допускаме, че дължината на пътя до всеки връх е 9999 и веднага

щом изчислим дължината на намерен път, например до i-ия връх, заменяме 9999 с изчислената дължина.

Масивът pred[] инициализираме с числото –1. С това допускаме, че не е известен предшественика на нито един връх. В процеса на търсенето на пътища, ако се окаже, че j-ия връх е предшественик на i-ия връх, на pred[i] присвояване стойността на j.

Следват матрицата на съседство и състоянието на трите масива след всеки преход в следващ връх за графа от фиг. 35. Те дават възможност да се проследи работата на компютърния вариант на описания по-горе алгоритъм.

Матрица на съседство:

```
      0  1  2  3  4
   0  0  5  8 16  0
   1  5  0  0 10  7
   2  8  0  0  5  0
   3 16 10  5  0  6
   4  0  7  0  6  0
```

Промените в трите масива:

```
        i ->     0     1     2     3     4
```

Масивите след инициализацията:

```
     visit->     0     0     0     0     0
     dist ->  9999  9999  9999  9999  9999
     pred ->    -1    -1    -1    -1    -1
```

Масивите след престоя в 0-ия връх:

```
j=0  visit->     1     0     0     0     0
     dist ->  9999     5     8    16  9999
     pred ->    -1     0     0     0    -1
```

Масивите след престоя в 1-ия връх:

```
j=1  visit->     1     1     0     0     0
     dist ->  9999     5     8    15    12
     pred ->    -1     0     0     1     1
```

Масивите след престоя в 2-ия връх:

```
j=2  visit->     1     1     1     0     0
     dist ->  9999     5     8    13    12
     pred ->    -1     0     0     2     1
```

Масивите след престоя в 4-ия връх:

```
j=4  visit->     1     1     1     0     1
     dist ->  9999     5     8    13    12
     pred ->    -1     0     0     2     1
```

Масивите след престоя в 3-ия връх:

```
j= 3 visit->     1     1     1     1     1
     dist ->  9999     5     8    13    12
     pred ->    -1     0     0     2     1
```

От масива visit[] се вижда, че накрая всички върхове са посетени.

От масива dist[] се вижда, че дължината на пътя от 0-ия връх до 1-ия е 5, до 2-ия е 8, до 3-ия е 13 и до 4-ия връх е 12.

От масива pred [] се получават най-кратките пътища по следния начин:

Връх 1 има за предшественик връх 0. Следователно пътят от 0 до 1 е 0 – 1.

Връх 2 има за предшественик връх 0. Следователно пътят от 0 до 2 е 0 – 2.

Връх 3 има за предшественик връх 2, а връх 2 има за предшественик връх 0. Следователно пътят от 0 до 3 е 0 – 2 – 3.

Връх 4 има за предшественик връх 1, а връх 1 има за предшественик връх 0. Следователно пътят от 0 до 4 е 0 – 1 – 4.

Програма 5.17. Намиране най-кратките пътища в граф, представен чрез матрица на съседство, по алгоритъма на Дейкстра

```
#include <fstream.h>
#include <iomanip.h>
#include <conio.h>
#include "graf_mat.h"
unsigned *visit, *dist,v; int *pred;
void clasGraph::dijkstra()
 {
  if (!(visit=new unsigned[BrV])) exit(1);
  if (!(dist=new unsigned[BrV])) exit(1);
  if (!(pred=new int[BrV])) exit(1);
  unsigned i,
          t_min_dist;//Текуща минимална дистанция от стартовия връх
  int j;
  for (i=0;i<BrV;i++) {visit[i]=0; dist[i]=9999; pred[i]=-1;}
  cout<<"Начален връх: "; cin>>v;
  visit[v]=1;
  for (i=0;i<BrV;i++)
      if (A[v][i]!=0) { dist[i]=A[v][i]; pred[i]=v; }
  for (;;) {
  j=-1;
  t_min_dist=9999;
  //Избираме като връх j този, за който visit[j]=0 и dist[j] е
минимално
  for (i=0;i<BrV;i++)
     if (visit[i]==0 && dist[i]<t_min_dist){ t_min_dist=dist[i]; j=i;
}
  if (j==-1) break;//Не е намерен непосетен на разстояние по-малко от
9999.
  visit[j]=1; //Намерен е и го маркираме като посетен
  //За всеки непосетен връх i изпълняваме d[i]=min(d[i], d[j]+A[j][i])
  for (i=0;i<BrV;i++)
     if (visit[i]==0 && A[j][i]!=0)
       if ( dist[j] + A[j][i] < dist[i] ) {
           dist[i]=dist[j] + A[j][i];
           pred[i]=j;//Регистрираме j-ия връх като предшественик на i-
ия.
       }
     }
 }
void printPath(unsigned s, unsigned j)
 {
  if (pred[j]!=s) printPath(s, pred[j]);
  cout<<j<<' ';
 }

void printResult(unsigned s)
 {
     unsigned i;
      for (i=0;i<BrV;i++) {
```

```
          if (i!=s) {
      if (dist[i]==9999)
        cout<<"Няма път между върховете "<<s<<" и "<<i<<endl;
      else {
        cout<<"Минимален път от връх "<<s<<" до връх "<<i<<": ";
        cout<<s<<' ';printPath(s,i);
        cout<<"\b, дължина на пътя "<<dist[i]<<endl;
      }
          }
       }
 }

void main()
{
  char ime_vh_fl[21];
  cout<<"\nИме на входния файл: "; cin>>ime_vh_fl;
  fin.open(ime_vh_fl,ios::in);
  if (fin.bad()) {cout<<"Няма такъв файл.\n"; getch(); return;}
  fin>>BrV; //Въвежда от текстовия файл броя на върховете на графа
  clasGraph Graph;       //Създава обекта Graph
  Graph.make_graf();          //Запълва матрицата на съседство
  Graph.display_graf();       //Извежда матрицата на съседство на
екрана
  Graph.display_graf(fout);   //Извежда матрицата на съседство в
текст. файл
  cout<<"\n"; Graph.dijkstra();
  printResult(v);
}
```

Програма 5.18. Намиране най-кратките пътища в граф, представен чрез масив от списъци на съседство, по алгоритъма на Дейкстра

```
#include <fstream.h>
#include <iomanip.h>
# include <conio.h>
#include "graf_sp.h"
unsigned *used,*dist, v; int *pred;
//
void clasGraph::dijkstra()
{
 if (!(used=new unsigned[BrV])) exit(0);
 if (!(dist=new unsigned[BrV])) exit(0);
 if (!(pred=new int[BrV])) exit(0);
 for (int i=0;i<BrV;i++) { used[i]=0; dist[i]=9999; pred[i]=-1; }
 unsigned t_min_dist;//Текуща дължина на най-късия път
 int j;
 cout<<"Начален връх: "; cin>>v;
 rebro *p=a[v].first;
 while (p){ i=p->nom; dist[i]=p->teg; pred[i]=v; p=p->next; }
 used[v]=1;
 for (;;) {
  j=-1;
  t_min_dist=9999;
  //Избираме като връх j този, за който used[j]=0 и dist[j] е
минимално
  for (i=0;i<BrV;i++)
   if (used[i]==0 && dist[i]<t_min_dist){ t_min_dist=dist[i]; j=i; }
```

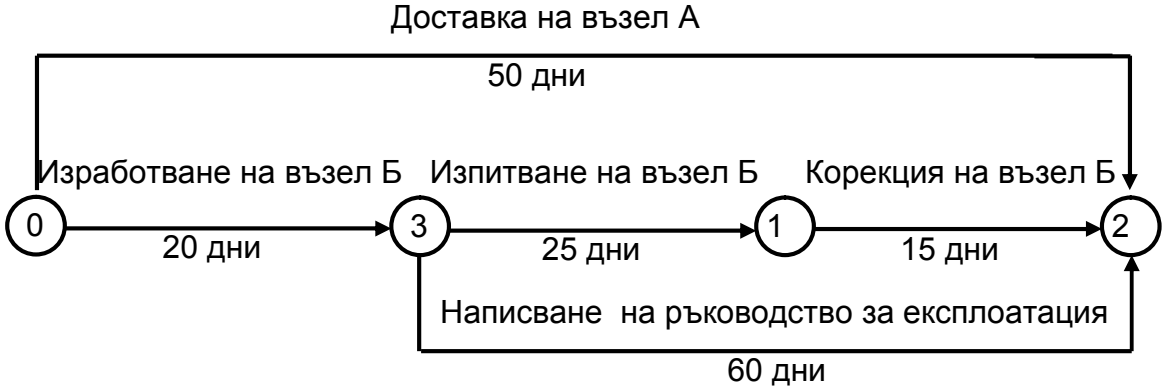
```
  if (j==-1) break;//Няма непосетен, до който няма път.
  used[j]=1; // j-ия връх е непосетен и има път до него.
  //За всеки непосетен връх i изпълняваме d[i]=min(d[i], d[j]+A[j][i])
  p=a[j].first;
  while (p){
   i=p->nom;
   if (!used[i])
    if (dist[i] > dist[j] + p->teg) {
       dist[i]=dist[j] + p->teg;
       pred[i]=j;//Регистрираме j-ия връх като предшественик на i-ия.
    }
    p=p->next;
  }
 }
}
void printPath(unsigned s, unsigned j)
 { if (pred[j]!=s) printPath(s, pred[j]); cout<<j<<' '; }

void printResult(unsigned s)
{
 unsigned i;
 for (i=0;i<BrV;i++) {
  if (i!=s) {
     if (dist[i]==9999)cout<<"Няма път между върховете "<<s<<" и
"<<i<<endl;
     else {
       cout<<"Минимален път от връх "<<s<<" до връх "<<i<<": ";
       cout<<s<<' ';printPath(s,i);
       cout<<"\b, дължина на пътя "<<dist[i]<<endl;
     }
  }
 }
}

void main()
{
  char ime_vh_fl[21];
  cout<<"\nИме на входния файл: "; cin>>ime_vh_fl;
  fin.open(ime_vh_fl,ios::in);
  if (fin.bad()) {cout<<"Няма такъв файл.\n";getch();return;}
  fin>>BrV; //Въвеждаме от текстовия файл броя на върховете на графа
  clasGraph Graph;  //Създаваме обекта Graph
  Graph.make_graf();    //Запълваме матрицата на съседство
  Graph.display_graf(); //Извеждаме на екрана матрицата на съседство
  cout<<"\n"; Graph.dijkstra();//Намиране на най-кратките пътища
  printResult(v);  //Извеждане на резултата
}
```

5.10. Мрежа на дейностите. Метод на критичния път

Едно важно приложение на графите е планирането изпълнението на проект. Някои добре известни техники в тази област са `CPM (Critical Path Method` – метод на критичния път) и `PERT (Perfonmance Evaluation and Review Technique` – оценка на изпълнението и рецензия). Един проект е съвкупност от дейности, някои от които са свързани с останалите. Например, когато строим къща, изграждането на покрива не мож да започне, преди да са завършени стените. Има два начина за представяне на дейности

чрез граф: всяка дейност е връх или всяка дейност е ребро. Ще изберем да представяме дейностите чрез ребра. Завършването на някаква дейност ще разглеждаме като събитие и в графа то ще присъства като връх. Накратко графът, представящ изпълнението на проект, ще се състои от ребра (дейности) и върхове (събития) и ще бъде ориентиран и тегловен. Тегло на дадено ребро ще бъде времетраенето за реализацията на дейността, която то представя. На фиг. 36 е представен графът на дейности по създаването на някакво устройство, което се състои от два възела: възел А и възел Б.

Възел А се доставя от друг производител и доставката отнема 50 дни. Възел Б се изработва на място и това отнема 20 дни. След това възел Б се изпитва, което отнема 25 дни. Накрая, въз основа на резултатите от изпитването, се правят корекции, които отнемат още 15 дни. Докато възелът се изпитва и коригира, трябва да се напише и ръководство за експлоатация. Писането може да започне едва след като възел Б е изработен и отнема 60 дни.



Фиг. 36. Граф, представящ изпълнението на проект (мрежа на дейностите)

Ще си поставим задачата да определим кога най-рано и кога най-късно можем да започнем изпълнението на всяка дейност и съответно кога най-рано и кога най-късно ще завършим с изпълнението на всяка дейност.

Тази задача се решава чрез топологично сортиране. Затова и програмата за нейното решаване ще получим чрез целесъобразни изменения в програмата за топологично сортиране. Тук ще се ограничим само с програма за граф, представен с матрица на съседство.

Програмата ще се състои от три части.

I-ва част. Задачата на тази част е намирането на най-ранните моменти на настъпване на събитията в графа. Тази част получаваме от програмата за топологично сортиране, като направим следните промени:

- добавяме нов масив за регистриране най-ранно настъпване на събитието nrns[]. Елемент `nrns[i]` съответства на `i`-ия връх и стойността му трябва да показва момента, в който най-рано може да настъпи събитие `i`, например `nrns[1]=45` означава, че събитие 1 ще настъпи най-рано 45 дни след началото;
- след като вземем връх (събитие) от опашката, не го извеждаме в топологичния ред, а изчисляване най-дългия път до него;
- въведена е допълнителна променлива `maxT` за максималното време за реализация на всички дейности.

В края на I-та част от програмата елементите на масива `nrns[]` и променливата `maxT` вече имат стойности. За разглеждания пример те са:

```
nrns[0]=0,
nrns[1]=45,
nrns[2]=80,
nrns[3]=20,
maxT=80.
```

II-ра част. Задачата на тази част е намирането на най-късните моменти на настъпване на събитията в графа. Това става пак чрез топологично сортиране, но в обратен ред, т.е. със следните особености:

- вместо масива с броя на предшествениците br_pred ще ползваме масив с броя на наследниците br_nasl. Броят на наследниците на всеки връх намираме като преброим ненулевите елементи в съответния ред в матрицата на съседство;
- вместо насива nrns[] сега ще ползваме масив за регистрация на най-късното настъпване на събитията nkns[]. Елемент nkns[i] съответства на i-ия връх и стойността трябва да показва момента, в който най-късно настъпва събитие i, например nkns[1]=65 означава, че събитие 1 ще настъпи най-късно 65 дни след началото;
- най-напред в опашката ще изпратим събитията (върховете) без наследници. След това ще ги вземем от опашката, ще отбележим, че моментът на тяхното настъпване е maxT, ще намалим броя на наследниците на техните предшественици с 1 и ще изчислим времето до тяхното настъпване и т.н.
- за тази втора част можем да копираме първата част, но трябва да отчетем, че сега наследниците и предшествениците са си разменили местата, а това налага да разменим i-тата и j-тата.

III-та част. В тази част от масивите nrns[] и nkns[] получаваме за всяка дейност (ребро) от графа следните времена:

nrzap – най-ранно започване на дадена дейност
nrzav – най-ранно завършване на дадена дейност
nkzap – най-късно започване на дадена дейност
nkzav – най-късно завършване на дадена дейност

Резултатите се извеждат във вид на таблица в текстов файл. За нашия пример програмата извежда следната таблица:

i	j	d	nrzap	nrzav	nkzap	nkzav
0	2	50	0	50	30	80
0	3	20	0	20	0	20
1	2	15	45	60	65	80
3	1	25	20	45	40	65
3	2	60	20	80	20	80

В таблицата i е началото на реброто-дейност, j е краят на реброто-дейност, а d е теглото на реброто или броя дни за извършване на тази дейност. Останалите означения обяснихме по-горе.

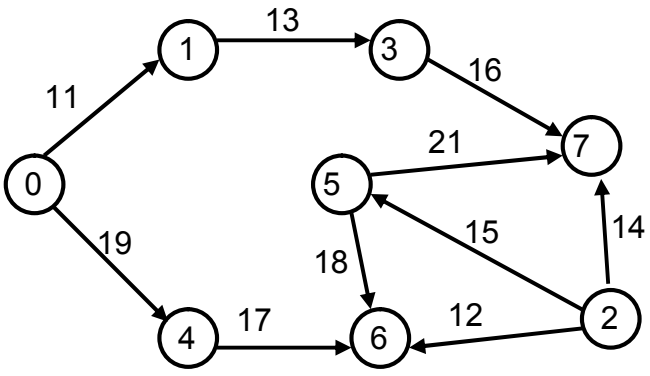
Програма 5.19. Организация на изпълнението на проект

```
# include <iostream.h>
# include <conio.h>
#include "graf_mat.h"

void clasGraph::OrgProekt()
{
 unsigned i,j,k,n,t,maxT=0,
 br_visit=0, // Брой на посетените върхове
 *visit,     // За масива, отчитащ посетеността на върховете
 *br_pred,   // За масива, отчитащ броя на предшествениците на всеки
връх
 *br_nasl,   // За масива, отчитащ броя на наследниците на всеки връх
 *nrns,      // За масива най-ранно настъпване на събитие
 *nkns;      // За масива най-късно настъпване на събитие
 if (!(visit=new unsigned[BrV])) exit(0);
 if (!(br_pred=new unsigned[BrV])) exit(0);
 if (!(br_nasl=new unsigned[BrV])) exit(0);
 if (!(nrns=new unsigned[BrV])) exit(0);
```

```
 if (!(nkns=new unsigned[BrV])) exit(0);
 for (i=0;i<BrV;i++) {visit[i]=0;br_pred[i]=0; nrns[i]=0;}
// 1. Търсене най-ранните моменти на настъпване на събитията от графа
 for (j=0;j<BrV;j++){
  for (i=0;i<BrV;i++) if (A[i][j]) br_pred[j]++;//Преброяване на
предшествениците
  if (br_pred[j]==0) Opashka.DobEl(j);//Връх без предшественици отива
в опашката
 }
 while (Opashka.Br) { //Докато опашката не е празна,...
  n=Opashka.Br;
  for (i=0;i<n;i++){
   Opashka.VzeEl(k);//... вземаме връх от опашката ...
   visit[k]=1;      //... маркираме го като посетен и...
   br_visit++;      //... го преброяваме, като посетен.
   for (j=0;j<BrV;j++)
    if (A[k][j]!=0 && visit[j]==0){ //На непосетените наследници на к-
ия връх ...
     br_pred[j]--;               //... намаляваме входната степен и...
     t=nrns[k]+A[k][j];
     if (t>nrns[j]) nrns[j]=t;
     if (t>maxT) maxT=t;
     if (br_pred[j]==0) Opashka.DobEl(j); //ако стане 0, пращаме го в
опашката
    }
  }
 }
 if (br_visit<BrV) {cout<<"Графът е цикличен.\n"; getch();return;}
// 2. Търсене най-късните моменти на настъпване на събитията от графа
for (i=0;i<BrV;i++) {visit[i]=0;br_nasl[i]=0; nkns[i]=maxT;}
 for (j=0;j<BrV;j++){
  for (i=0;i<BrV;i++) if (A[j][i]) br_nasl[j]++;//Преброяване на
наследниците
  if (br_nasl[j]==0) Opashka.DobEl(j);//Връх без наследници отива в
опашката
 }
 while (Opashka.Br) { //Докато опашката не е празна,...
  n=Opashka.Br;
  for (i=0;i<n;i++){
   Opashka.VzeEl(k);//... вземаме връх от опашката ...
   visit[k]=1;      //... маркираме го като посетен и...
   br_visit++;      //... го преброяваме, като посетен.
   for (j=0;j<BrV;j++)
    if (A[j][k]!=0 && visit[j]==0){ //На непосетените предшественици
на j-ия връх ...
     br_nasl[j]--;         //... намаляваме входната степен и...
     t=nkns[k]-A[j][k];
     if (t<nkns[j]) nkns[j]=t;
     if (br_nasl[j]==0) Opashka.DobEl(j); //ако стане 0, пращаме го в
опашката
    }
  }
 }
// 3. Отпечатване на графика на операциите
 int nrzap,nkzap,nrzav, nkzav,d;
 fout.open("c:\\sdp\\graphs\\DopInf",ios::out);
```

```
 fout<<"\n   i    j    d   nrzap  nrzav  nkzap  nkzav\n";
 for (i=0;i<BrV;i++)
  for (j=0;j<BrV;j++)
     if (A[i][j]!=0) {
      d=A[i][j];
      nrzap=nrns[i];
      nrzav=nrzap+d;
      nkzav=nkns[j];
      nkzap=nkzav-d;
      fout<<setw(4)<<i<<"  "
     <<setw(3)<<j<<"  "
     <<setw(3)<<d<<"  "
     <<setw(5)<<nrzap<<"  "
     <<setw(5)<<nrzav<<"  "
     <<setw(5)<<nkzap<<"  "
     <<setw(5)<<nkzav<<"  \n";
  } fout.close();
}

void main()
{
  char ime_vh_fl[21];
  cout<<"\nИме на входния файл: "; cin>>ime_vh_fl;
  fin.open(ime_vh_fl,ios::in);
  if (fin.bad()) {cout<<"Няма такъв файл.\n";getch();return;}
  fin>>BrV; //Въвеждаме броя на върховете от текстовия файл
  clasGraph Graph;
  Graph.make_graf();
  Graph.display_graf();
  //Graph.display_grafTF();
  Graph.OrgProekt();
  cout<<"\nДопълнителна информация можете да намерите във файл
\"DopInf.\"\n";
}
```

Интересен е например следният граф. Той има два върха (0 и 2) без предшественици т.е. възможно е изпълнението на дейностите от графа да започне едновременно от връх 0 и от връх 2. Той има и два върха без наследници.



Фиг. 37. Граф на дейности

Резултатите от изпълнението на програмата са следните:

```
 i    j    d   nrzap  nrzav  nkzap  nkzav
 0    1   11      0     11      0     11
 0    4   19      0     19      4     23
 1    3   13     11     24     11     24
```

```
2     5    15       0      15       4      19
2     6    12       0      12      28      40
2     7    14       0      14      26      40
3     7    16      24      40      24      40
4     6    17      19      36      23      40
5     6    18      15      33      22      40
```

5.11. Цикли

Нека да си припомним, че *цикъл* в граф е пътят, който започва от един връх и завършва в същия връх. Цикълът е прост, когато не съдържа вложени цикли. Представляват интерес хамилтоновият цикъл и ойлеровият цикъл.

5.11.1. Хамилтонов цикъл. Задача за търговския пътник

Хамилтонов цикъл в граф се нарича цикъл, който съдържа всички върхове от графа еднократно. Или хамилтоновият цикъл в граф е път, който минава през всички върхове от графа по един път. Графът, чийто върхове образуват хамилтонов цикъл, се нарича хамилтонов граф. Хамилтоновият граф може да бъде тегловен или безтегловен.

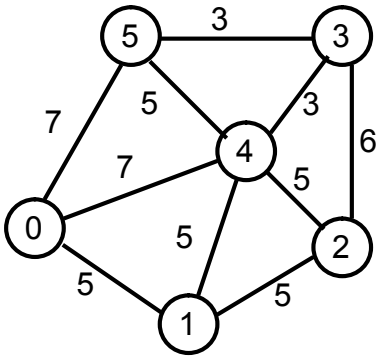
Съществуват две класически задачи, свързани с хамилтонови графи:

- проверката дали даден граф е хамилтонов;
- намиране хамилтоновия цикъл с най-малка дължина в тегловен хамилтонов граф (задача за търговския пътник).

Тук ще се спрем само на втората задача, т.е. на задачата за търговския пътник, която ще формулираме като задача за намиране на *хамилтонов цикъл с най-малка дължина в тегловен неориентиран хамилтонов граф.*

Тази задача има много общо със задачата за намиране на всички пътища между два зададени върха в безтегловен граф. Всъщност разликите са следните три:

- графът е тегловен;
- началният и крайният връх на търсените пътища в графа е един и същ;
- трябва да се запомня не само текущия (търсения в момента) път, но и най-краткия от всички намерени до момента пътища;
- трябва да се изчислява дължината на всеки намерен път - хамилтонов цикъл и да се помни дължината на най-краткия от тях;
- за решение на задачата се приема най-краткият път от всички намерени пътища-хамилтонови цикли;
- тъй като търсеният път трябва да бъде хамилтонов цикъл, т.е. трябва да преминава през всички върхове, за начален връх може да се избере всеки връх от графа. За такъв ще изберем 0-ия връх.



Фиг. 38. Граф за илюстрация на задачата за търговския пътник

Ето защо, лесно можем да създадем алгоритъм и да напишем програма за намиране на хамилтонов цикъл с най-малка дължина, като модифицираме съответно алгоритъма и програмата за намиране на всички пътища между два зададени върха в безтегловен граф. Полезно е да си припомним, че алгоритъма и програмата за намиране на всички пътища между два зададени върха в безтегловен граф получихме като модифицирахме алгоритъма за обхождане на граф в дълбочина.

За да изясним алгоритъма ще използваме графа от фиг. 38.

1. По алгоритъма за обхождане на граф в дълбочина, ще получим първия хамилтонов цикъл 0-1-2-3-4-5-0 с дължина 31 и ще го приемем за минимален (най-кратък);

2. За да намерим друг път, ще отстъпим от 0-ия връх в 5-ия и ще обявим 0-ия за непосещаван. Ще отстъпим и от 5-ия връх в 4-ия и ще обявим 5-ия за непосещаван. Ще отстъпим и от 4-ия връх в 3-ия и ще обявим 4-ия за непосещаван. От 3-ия връх можем да продължим напред и да получим втория хамилтонов цикъл 0-1-2-3-5-4-0 със същата дължина.

3. За да намерим друг път, ще отстъпим последователно от 0-ия връх в 4-ия и ще обявим 0-ия за непосещаван. Ще отстъпим и от 4-ия връх в 5-ия и ще обявим 4-ия за непосещаван. Ще отстъпим и от 5-ия връх в 3-ия и ще обявим 5-ия за непосещаван. Ще отстъпим и от 3-ия връх в 2-ия и ще обявим 3-ия за непосещаван. От 3-ия връх можем да продължим напред и да получим третия хамилтонов цикъл 0-1-2-4-3-5-0 със същата дължина 28. Този път (хамилтонов цикъл) е по кратък, затова го приемаме за минимален.

4. По описания начин ще намерим четвъртия хамилтонов цикъл 0-1-4-2-3-5-0 с дължина 31, т.е. той не е по кратък от третия.

5. Пак по описания начин ще намерим петия хамилтонов цикъл 0-4-1-2-3-5-0 с дължина 33, т.е. и той не е по кратък от третия.

Минимален се оказа третият хамилтонов цикъл.

Програма 5.20. Намиране на хамилтънов цикъл с минимална дължина в тегловен граф, предстаен чрез матрица на съседство. Задача за търговския пътник

```
#include <iostream.h>
# include <conio.h>
#include "graf_mat.h"
 int      tekSum,   //Дължина на текущия цикъл
     minSum;   //Дължина на минималния цикъл
 unsigned
  *tekCycle,   //За масива, съдържащ върховете на текущия цикъл
  *minCycle,        //За масива, съдържащ върховете на минималния
цикъл
  *visit;      //За масива, регистриращ посетените върхове
//Рекурсивна член-функция за намиране минималния хамилтонов цикъл
void clasGraph::hamDFS(unsigned i, unsigned nivo)
{
  unsigned k;
  if (i==0 && nivo>0) { //Край на търсенето на пореден по-кратък цикъл
      if (nivo==BrV) {    //Намерен е по-кратък цикъл и ...
       minSum=tekSum;     // ...регистрираме него, като минимален
       for (k=0;k<BrV;k++) minCycle[k]=tekCycle[k];
      }
      return;
  }
  if (visit[i]) return;
  visit[i]=1;//Регистрираме i-ия връх като посетен и ...
  for (k=0;k<BrV;k++) //... обхождаме неговите съседи
     if (A[i][k] && k!=i) {
       tekCycle[nivo]=k;
       tekSum+=A[i][k];
       if (tekSum<minSum) hamDFS(k,nivo+1);
       tekSum-=A[i][k];
     }
     visit[i]=0;
}
```

```
void main()
{
  char ime_vh_fl[21]; unsigned k;
  cout<<"\nИме на входния файл: "; cin>>ime_vh_fl;
  fin.open(ime_vh_fl,ios::in);
  if (fin.bad()) {cout<<"Няма такъв файл.\n";getch();return;}
  fin>>BrV; //Въвеждаме от текстовия файл броя на върховете на графа
  clasGraph Graph;  //Създава обекта Graph
  Graph.make_graf();    //Запълва матрицата на съседство
  Graph.display_graf(); //Извежда на екрана матрицата на съседство
  if (!(tekCycle=new unsigned [BrV])) exit(0);
  if (!(minCycle=new unsigned [BrV])) exit(0);
  if (!(visit=new unsigned [BrV])) exit(0);
  for (k=0;k<BrV;k++) visit[k]=0;
  minSum=9999; tekSum=0; tekCycle[0]=1;
  Graph.hamDFS(0,0);
  cout<<"Минималният хамилтонов цикъл е 0 ";
  for (int i=0; i<BrV-1; i++) cout<<minCycle[i]<<' ';
  cout<<"0 с дължина на пътя " <<minSum<<endl;
}
```

Програма 5.21. Намиране на хамилтонов цикъл с минимална дължина в тегловен граф, представен чрез масив от списъци на съседство. Задача за търговския пътник

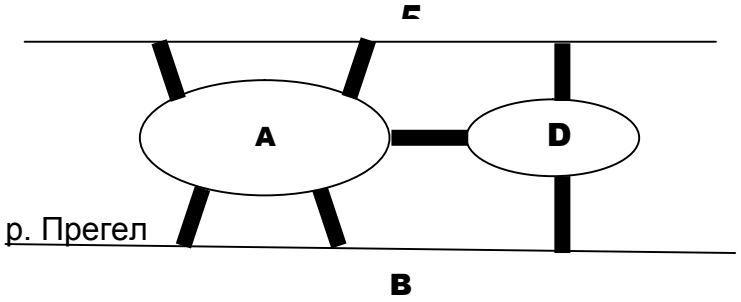
```
//Хамилтънов цикъл. Задача за търговския пътник.
# include <iostream.h>
# include <conio.h>
#include "graf_sp.h"
 int      tekSum,//Дължина на текущия цикъл
          minSum;//Дължина на минималния цикъл
 unsigned
  *tekCycle,   //За масива, съдържащ върховете на текущия цикъл
  *minCycle,   //За масива, съдържащ върховете на минималния цикъл
  *visit;      //За масива, регистриращ посетените върхове
//Рекурсивна член-функция за намиране минималния хамилтонов цикъл
void clasGraph::hamDFS(unsigned i, unsigned nivo)
{
  unsigned k;
  if (i==0 && nivo>0) { //Край на поредото търсене на по-кратък цикъл
    if (nivo==BrV) {    //Намерен е по-кратък цикъл
     minSum=tekSum;
     for (k=0;k<BrV;k++) minCycle[k]=tekCycle[k];
    }
    return;
  }
  if (visit[i]) return;
  visit[i]=1;
  rebro *p=a[i].first;
  while (p){
    k=p->nom;
    if (k!=i) {
       tekCycle[nivo]=k;
       tekSum+=p->teg;
        if (tekSum<minSum) hamDFS(k,nivo+1);
       tekSum-=p->teg;
    }p=p->next;
```

```
  }visit[i]=0;
 }

void main()
{
  char ime_vh_fl[21]; unsigned k;
  cout<<"\nИме на входния файл: "; cin>>ime_vh_fl;
  fin.open(ime_vh_fl,ios::in);
  if (fin.bad()) {cout<<"Няма такъв файл.\n";getch();return;}
  fin>>BrV; //Въвеждаме от текстовия файл броя на върховете на графа
  clasGraph Graph;  //Създава обекта Graph
  Graph.make_graf();    //Запълва матрицата на съседство
  Graph.display_graf(); //Извежда на екрана матрицата на съседство
  if (!(tekCycle=new unsigned [BrV])) exit(0);
  if (!(minCycle=new unsigned [BrV])) exit(0);
  if (!(visit=new unsigned [BrV])) exit(0);
  for (k=0;k<BrV;k++) visit[k]=0;
  minSum=9999; tekSum=0; tekCycle[0]=1;
  Graph.hamDFS(0,0);
  cout<<"\nМинималният хамилтънов цикъл е 0 ";
  for (int i=0; i<BrV-1; i++) cout<<minCycle[i]<<' ';
  cout<<"0 с дължина на пътя " <<minSum<<endl;
}
```

5.11.2. Ойлеров цикъл

Цикълът е ойлеров, когато по всяко ребро се преминава еднократно. Той носи името на един от най-големите математици на всички времена Леонард Ойлер, който е дал много на инженерните науки, използвайки математичните си знания за решаване на задачи от реалния живот. Много често, когато познатата математика не достатъчна, за да реши възникналия проблем, той създава нови математични методи и по този начин развива и самата математика.

Ойлер преподавал в университета в Калининград, наричан тогава Кьонигсберг. Градът е разположен на двата бряга и на два острова на р. Прегел. Четирите части на града се



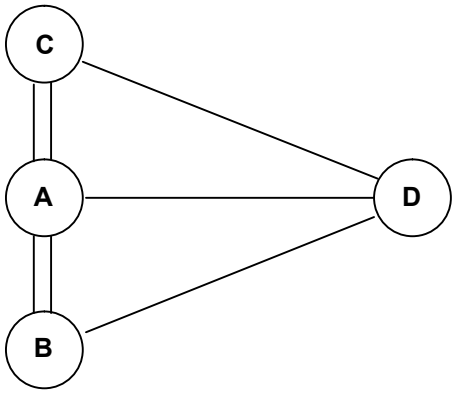
свързвали помежду си чрез седем моста, както е показано на фиг. 39.

Фиг. 39. Скица на град Калининград

Гражданите на този град много обичали разходките из града и докато се разхождали се запитали, дали е възможно гражданин на този град да излезе от дома си, да мине по един път през всеки мост и да се прибере отново у дома си. Тази, на пръв поглед проста задача, се оказала такава главоблъсканица, че привлякла вниманието дори на именития Ойлер.

Ойлер представил четирите части от града с кръгчета, а мостовете - чрез линии (фиг.40) и веднага заключил, че описаната разходка не е възможна. За да е възможна, всека част на града (A, B, C и D) трябва да е свързана с останалите с четен брой мостове. Това е така, защото, когато разхождащият се навлезе в терирорията на някоя

част по един мост, трябва да има още един мост, по който да излезе от нея. Има само две изключения от това правило: когато започва и когато завършва разходката.



Фиг. 40. Скица на Ойлер на град Калининград

Представяйки града и неговите мостове по този начин, Ойлер създал първия граф и дефинирал самото понятие граф. Наред с това, той веднага осъзнал, че с това поставя началото на нов дял от математиката – теорията на графите.

Ойлер имал славата на човек, който може да реши всяка задача, формулирана като математическа. Когато не съществуват математическите знания, нужни за решаването на дадаена задача, той ги създавал, какъвто е случаят с теорията на графите.