Grafuri neorientate.Parcurgerea in latime

Parcurgerea grafurilor presupune examinarea în vederea prelucrării tuturor vârfurilor acelui graf într-o anumită ordine, ordine care să permită prelucrarea optimă a informaţiilor ataşate grafului. În acest scop s-au dezvoltat două tehnici fundamentale de traversare a grafurilor, una bazată pe căutarea în adâncime, cealaltă bazată pe căutarea prin cuprindere. Ambele tehnici constituie nuclee de bază pornind de la care se pot dezvolta numeroşi algoritmi eficienţi de prelucrare a grafurilor.

Parcurgerea în lăţime a fost descoperită de către Moore în contextul căutării de drumuri în labirinturi. Lee a descoperit, în mod independent, acelaşi algoritm în contextul stabilirii firelor de pe plăcile de circuite. Hopcroft şi Tarjan au argumentat folosirea reprezentării prin liste de adiacenţă în defavoarea reprezentării prin matrice de adiacenţă, pentru grafurile rare, şi au fost primii care au recunoscut importanţa algoritmică a parcurgerii în adâncime. Parcurgerea în adâncime a fost folosită pe scară largă începând cu anul sfârşitul anului 1950, în special în programele din domeniul inteligenţei artificiale. Tarjan este cel care a elaborat un algoritm liniar pentru determinarea componentelor tare conexe, iar Knuth a fost primul care a dat un algoritm liniar pentru sortarea topologică.

Căutarea prin cuprindere sau traversarea grafurilor în lăţime este unul dintre cei mai simpli algoritmi de căutare într-un graf şi arhetipul pentru mulţi algoritmi de grafuri importanţi. Algoritmul lui Dijkstra pentru determinarea drumurilor minime de la un nod sursă la toate celelalte şi algoritmul lui Prim pentru determinarea arborelui parţial de cost minim folosesc idei similare din algoritmul de căutare în lăţime

Această metodă se bazează pe următoarea tehnică:

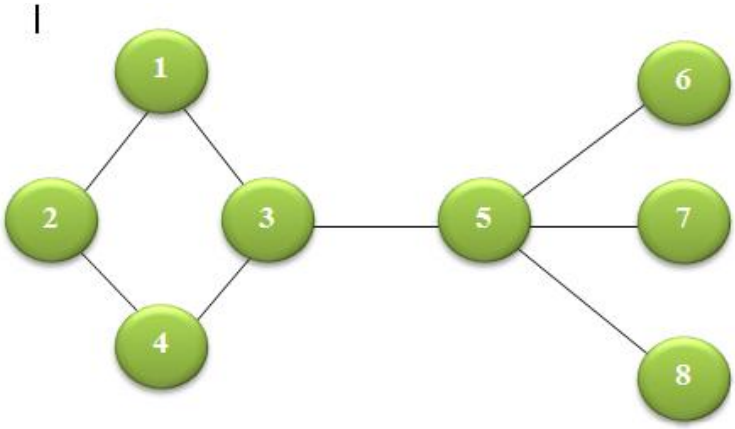
- fie un graf **G = (X,U)** cu **n** noduri şi un nod de plecare **ns** numit şi nod sursă

- căutarea în lăţime explorează sistematic muchiile grafului G pentru a "**descoperi**" fiecare nod accesibil din **ns**. Algoritmul calculează distanţa (cel mai mic număr de muchii) de la **ns** la toate vârfurile accesibile lui. El produce un "arbore de lăţime" cu rădăcina în **ns**, care conţine toate nodurile accesibile. Pentru fiecare nod **v** accesibil din **ns**, calea din arborele de lăţime de la **ns** la **v** corespunde "**celui mai scurt drum**" de la **ns** la **v**, adică conţine un număr minim de muchii.

Traversarea grafurilor în lăţime sau **Breadth-First** este numită astfel pentru că lărgeşte, uniform, frontiera dintre nodurile descoperite şi cele nedescoperite, pe lăţimea frontierei. Aceasta înseamnă că algoritmul descoperă toate vârfurile aflate la distanţa **k** faţă de **ns** înainte de a descoperi vreun vârf la distanţa **k+1**. Cu alte cuvinte traversarea în lăţime a grafurilor presupune faptul că după vizitarea unui anumit nod **v**, sunt parcurşi toţi vecinii nevizitaţi ai acestuia, apoi toţi vecinii nevizitaţi ai acestora din urmă până la vizitarea tuturor nodurilor grafului(spunem că două noduri sunt vecine dacă sunt adiacente).

Implementarea acestei metode se face folosind o structură de date de tip **coadă**. Cozile sunt structuri de date în care elementele sunt **inserate** la un capăt (**sfârşitul cozii**) şi sunt **suprimate** de la celălalt capăt (**începutul cozii**). Ele implementează politica "**primul venit - primul servit**". Asupra unei cozi acţionează operatori specifici cum ar fi: iniţializare coadă, test de coadă vidă, adăugă un element la sfârşitul cozii, scoate un element de la începutul cozii. Cozile pot fi implementate static(cu variabile de tip tablou unidimensional) sau dinamic.

În acest caz coada este iniţializată cu un nod oarecare al grafului. La fiecare pas, pentru nodul aflat în vârful cozii, se adaugă la coadă toţi vecinii nevizitaţi ai nodului respectiv după care se şterge din coadă primul nod.Fie graful din figura următoare care are n = 8 noduri



Vom utiliza un vector v, cu un număr de elemente egal cu numărul de noduri din graf, iar fiecare element al său poate lua valoarea 1, dacă şi numai dacă nodul a fost "vizitat", sau valoarea dacă nodul nu a fost vizitat.

Algoritm de **parcurgere in latime a unei singure componente conexe**:

1. **citirea datelor de intrare**(număr de noduri si muchiile grafului) şi construirea **matricei de adiacenţă**
2. afisarea pe ecran a matricei de adiacenta
3. **citirea/determinarea** unui **nod de start**
4. marcarea nodului de start ca **fiind vizitat:** v[i]=0
5. **afisarea** nodului de stat
6. **adaug la coadă** în **prima** pozitie nodul de start:
   - prim=1; //**poziţia primului nod** din coadă
   - ultim=1; // **poziţia ultimul nod** aşezat la coada
   - c[ultim]=ns; //**adăugarea nodului de start la coada**
7. **cât timp** coada nu este vidă **execută**
   - determină **TOATE** nodurile **adiacente cu primul nod din coadă şi nevizitate,** iar pentru fiecare nod astfel găsit efectuează următoarele operaţii:
     - marchează-l vizitat
     - afişează-l
     - adaugă-l la coada
   - **elimină** primul nod din coada

Aplicatii rezolvate ale algoritmului de parcurgere in latime

1.Determinarea componentelor conexe

În fişerul text **graf.in** este memorat un **graf neorientat neconex** astfel:

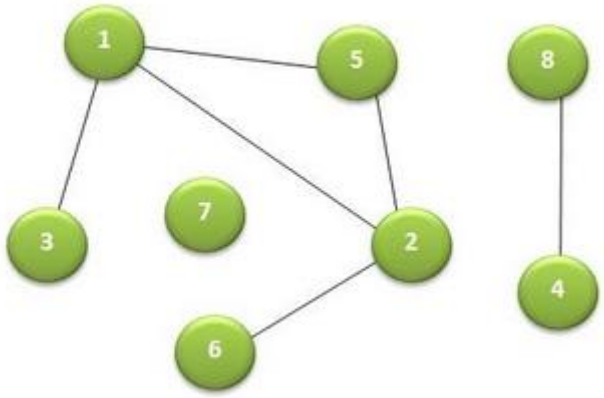
- pe prima linie un număr natural **n**, care reprezintă numărul de noduri ale unui graf neorientat.
- pe următoarele linii sunt memorate câte două numere naturale care reprezintă muchiile grafului.

Scrieţi un program care să **parcurgă în lăţime fiecare componetă conexă** a grafului dat. Componentele conexe vor fi numerotate, iar pentru fiecare componentă conexă vor fi afişate nodurile care o alcătuiesc.

Exemplu:

Continutul fişierului text **graf.in**	Rezultate asteptate
8 **1 3** **1 2** **1 5** **2 5** **2 6** **4 8**	Dacă nodul de start este 1 atunci se vor afişa următoarele rezultate: **Componenta conexă 1 conţine nodurile: 1 2 3 5 6** **Componenta conexă 2 conţine nodurile: 4 8** **Componenta conexă 3 conţine nodurile:7** **Graful este alcătuit din 3 componente conexe.**

Graful memorat în fişerul text de mai sus are următorul aspect grafic:



Program C++:

```
#include<iostream>
#include<fstream>
using namespace std;
int a[20][20],c[20],v[20],ns,n,comp;
int prim;
int ultim;
```

```
// citirea grafului din fisier text si construirea matricei de adiacenta
```

```
void citire(int a[20][20], int &n)
{  ifstream f("graf.in");
   int x,y;
    f>>n;
    while(f>>x>>y)
       a[x][y]=a[y][x]=1;
    f.close();
}
```

```
// afisarea pe ecran a matricei de adiacenta
```

```
void afisare(int a[20][20],int n)
{   cout<<"Matricea de adiacenta este : "<<endl;
```

```
    for( int i=1;i<=n;i++)
      { for(int j=1;j<=n;j++)
           cout<<a[i][j]<<" ";
        cout<<endl;
      }
}
```

// returnează primului nod nevizitat

```
int exista_nod_nevizitat(int v[20], int n)
{  for(int i=1;i<=n;i++)
     if(v[i]==0)
       return i; // primul nod nevizitat
     return 0;   // nu mai exista noduri nevizitate
}
```

// parcurgerea în latime a unei componente conexe, plecând din nodul de start ns

---

```
void parcurgere_latime(int a[20][20], int n,int ns)
   { comp++;
     v[ns]=1;
     cout<<"Componenta conexa : "<<comp<<" este formata din nodurile :";
     cout<<ns<<" ";
     prim=ultim=1;
     c[ultim]=ns;
     while(prim<=ultim)
        {for(int i=1;i<=n;i++)
           if(a[c[prim]][i]==1)
             if(v[i]==0)
               { ultim++;
                 c[ultim]=i;
                 cout<<i<<" ";
                 v[i]=1;
               }
           prim++;
        }
      cout<<endl;
   }
```

// functia principala main()

```
int main()
{ citire(a,n);
  afisare(a,n);
  cout<<"Dati nodul de start : "; cin>>ns;

  parcurgere_latime(a,n,ns);
  while(exista_nod_nevizitat(v,n)!=0)
    {ns=exista_nod_nevizitat(v,n);
     parcurgere_latime(a,n,ns);  //parcurg o alta componenta conexa
     }

  cout<<"Graful este alcătuit din "<<comp <<" componente conexe. ";
```

```
return 0;
}
```

2. Transformarea unui graf neconex intr-un graf conex

Enunt

În fişerul text **graf.in** este memorat un **graf neorientat neconex** astfel:

- pe prima linie un număr natural **n**, care reprezintă numărul de noduri ale unui graf neorientat.
- pe următoarele linii sunt memorate câte două numere naturale care reprezintă muchiile grafului.

Scrieţi un program care să determine **numărul minim de muchii care trebuiesc adăugate la graf astfel încât graful să devină conex**. Afişaţi şi o posibilă soluţie.

Exemplu:

Continutul fişierului text **graf.in**	Rezultate asteptate
8 **1 3** **1 2** **1 5** **2 5** **2 6** **4 8**	**Muchiile adăugate sunt: (1,4) (1,7)** **Numărul minim de muchii adăugate este 2.**

Graful memorat în fişerul text de mai sus are aspectul grafic ca in exemplul anterior.

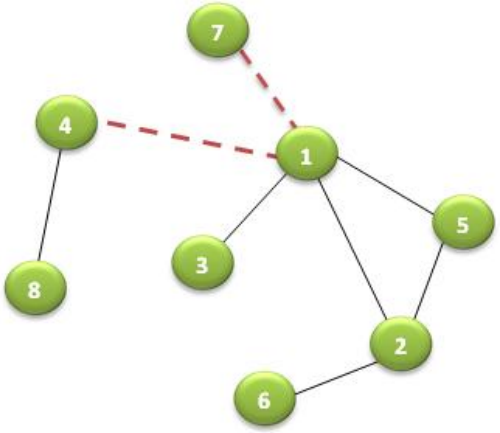
Rezolvare: Graful ales ca exemplu este alcătuit din trei componente conexe după cum urmează::

- **Componenta conexă 1 conţine nodurile: 1 2 3 5 6**
- **Componenta conexă 2 conţine nodurile: 4 8**
- **Componenta conexă 3 conţine nodurile:7**

Folosind algoritmul de parcurgere în lăţime se determină numărul de componente conexe. Numărul minim de muchii care trebuiesc adăugate pentru a transforma un graf neconex într-un graf conex este dat de următoarea relaţie:

numărul componentelor conexe-1

Muchiile care trebuiesc adăugate vor fi formate din nodul de start şi primul nod din fiecare componenta conexă iar graful conex va arata astfel:



Programul C++ este:

```
#include<iostream>
#include<fstream>
using namespace std;
int a[20][20],c[20],v[20],ns,n,comp;
int prim;
int ultim;
```

```
// citirea grafului din fisier text si construirea matricei de adiacenta
```

```
void citire(int a[20][20], int &n)
{   ifstream f("graf.in");
    int x,y;
    f>>n;
    while(f>>x>>y)
        a[x][y]=a[y][x]=1;
     f.close();
}
```

```
// afisarea pe ecran a matricei de adiacenta
```

```
void afisare(int a[20][20],int n)
{    cout<<"Matricea de adiacenta este : "<<endl;
     for( int i=1;i<=n;i++)
        { for(int j=1;j<=n;j++)
              cout<<a[i][j]<<" ";
          cout<<endl;
        }
}
// returnează primului nod nevizitat
int exista_nod_nevizitat(int v[20], int n)
{   for(int i=1;i<=n;i++)
      if(v[i]==0)
         return i; // primul nod nevizitat
      return 0;   // nu mai exista noduri nevizitate
}
```

```
// parcurgerea în latime a unei componente conexe, plecând din nodul de start ns
```

```
void parcurgere_latime(int a[20][20], int n,int ns)
   { comp++;
     v[ns]=1;
     prim=ultim=1;
     c[ultim]=ns;
     while(prim<=ultim)
         {for(int i=1;i<=n;i++)
            if(a[c[prim]][i]==1)
              if(v[i]==0)
                { ultim++;
                  c[ultim]=i;
                  v[i]=1;
                }
            prim++;
```

```
        }
    }
```

// functia principala main()

```
int main()
{ citire(a,n);
  afisare(a,n);
  ns=1;
  cout<<"Muchiile adaugate sunt:";
  parcurgere_latime(a,n,ns);
  while(exista_nod_nevizitat(v,n)!=0)
     {ns=exista_nod_nevizitat(v,n);
      cout<<"(1,"<<ns<<") ";
      parcurgere_latime(a,n,ns);  //parcurg o alta componenta conexa
      }
  cout<<endl<<"Numarul minim de muchii adaugate este "<<comp-1<<".";
return 0;
}
```

3. Algoritmul lui LEE

```
#include<iostream>
#include<fstream>
using namespace std;
int a[20][20],c[20],v[20],ns,n;
int prim;
int ultim;
void citire(int a[20][20],int &n)
{
    ifstream f("graf.in");
    int x,y;
    f>>n;
    while(f>>x>>y)
      a[x][y]=a[y][x]=1;
    f.close();
}
void afisare(int a[20][20],int n)
{
    cout<<"Matricea de adiacenta este : "<<endl;
    for(int i=1;i<=n;i++)
     {
        for(int j=1;j<=n;j++)
        cout<<a[i][j]<<" ";
        cout<<endl;
     }
}
int exista_nod_nevizitat(int v[20],int n)
{
    for(int i=1;i<=n;i++)
     if(v[i]==-1)
```

```
    return i;
    return 0;
}
int main()
{ citire(a,n);
  afisare(a,n);
  for(int i=1;i<=n;i++)
    v[i]=-1;
  cout<<"Dati nodul de start : "; cin>>ns;
  v[ns]=0;
  cout<<"Parcurgerea in latime este : ";
  cout<<ns<<" ";
  prim=ultim=1;
  c[ultim]=ns;
  while(exista_nod_nevizitat(v,n)!=0)
    {
       if(prim<=ultim)
        {//adaug la coada toate nodurile adiacente si nevizitate
        for(int i=1;i<=n;i++)
         if(a[c[prim]][i]==1)
          if(v[i]==-1)
           {
              ultim++;
              c[ultim]=i;
              cout<<i<<" ";
              v[i]=v[c[prim]]+1;
           }
           prim++;
        }
           else
            {
               ns=exista_nod_nevizitat(v,n);
               v[ns]=0;
               prim=ultim=1;
               c[prim]=ns;
               cout<<ns<<" ";
            }
    }

 for(int i=1;i<=n;i++)
    cout<<"Distanta minima dintre nodul "<<i<<" si "<<ns<<" este "<<v[i]<<endl;
 return 0;
}
```