# Cuprins

# 1.4. Închiderea tranzitivă a unui graf

Matricea închiderii tranzitive a unui graf G este o matrice pătratică A* având n linii şi n coloane (unde n reprezintă numărul de vârfuri din graf), cu elemente din mulţimea {0, 1}, definită astfel: A*[i][j]=1 dacă şi numai dacă există un drum/lanţ de lungime >0 de la i la j.

Matricea închiderii tranzitive este cunoscută şi sub denumirea de matricea drumurilor (pentru grafuri orientate), respectiv matricea lanţurilor (pentru grafuri neorientate).

Pentru a determina matricea închiderii tranzitive vom utiliza algoritmul *Roy-Warshall*:

1. Se iniţializează matricea închiderii tranzitive cu matricea de adiacenţă a grafului.
2. Considerăm fiecare vârf intermediar k şi verificăm pentru fiecare pereche de vârfuri (i, j) pentru care A*[i][j]=0 dacă există drum/lanţ de la i la k şi drum/lanţ de la k la j; în caz afirmativ, va exista drum/lanţ de la i la j, deci A*[i][j] devine 1.

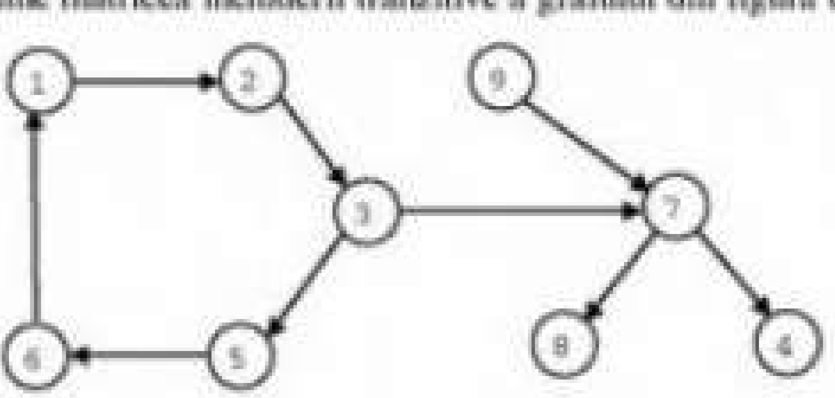
```
void RW()
{
 int k, i, j;
 for (k=1; k<=n; k++)
     for (i=1; i<=n; i++)
         for (j=1; j<=n; j++)
             AP[i][j]=AP[i][j] || AP[i][k] && AP[k][j];
}
```

*Observaţii*

1. Complexitatea algoritmului este de O($n^3$).
2. Pe diagonala principală în matricea închiderii tranzitive se obţine valoarea 1 pe linia i dacă şi numai dacă vârful i aparţine unui ciclu/circuit de lungime >0.
3. Matricea închiderii reflexive şi tranzitive a unui graf, A*, se defineşte în mod similar, cu diferenţa că A*[i][j]=1 dacă şi numai dacă există un drum/lanţ de lungime ≥0 de la i la j. Cu alte cuvinte, în matricea închiderii reflexive şi tranzitive, diagonala principală trebuie să fie iniţializată cu 1.

## *Exerciţii propuse*

1. Să se determine matricea închiderii tranzitive a grafului din figura următoare:



2. Să considerăm următoarea matrice a închiderii tranzitive a unui graf orientat. Să se identifice vârfurile care nu aparţin nici unui circuit.

```
1 1 1 1 1 1
1 1 1 1 1 1
0 0 0 0 0 1
1 1 1 1 1 1
1 1 1 1 1 1
0 0 0 0 0 0
```

3. Scrieţi un program care, utilizând matricea închiderii tranzitive, să determine toate vârfurile accesibile dintr-un vârf dat `x`. Spunem că vârful `y` este accesibil din vârful `x` dacă există un drum de la `x` la `y` (pentru graf orientat), respectiv un lanţ de la `x` la `y` (pentru graf neorientat).

## 1.5. Parcurgerea grafurilor

Parcurgerea unui *graf* presupune examinarea sistematică a vârfurilor grafului, cu scopul prelucrării informaţiilor asociate vârfurilor.

Există două metode fundamentale de parcurgere a grafurilor: parcurgerea în adâncime (*Depth First Search - DFS*) şi parcurgerea în lăţime (*Breadth First Search - BFS*).

### *Parcurgerea în adâncime*

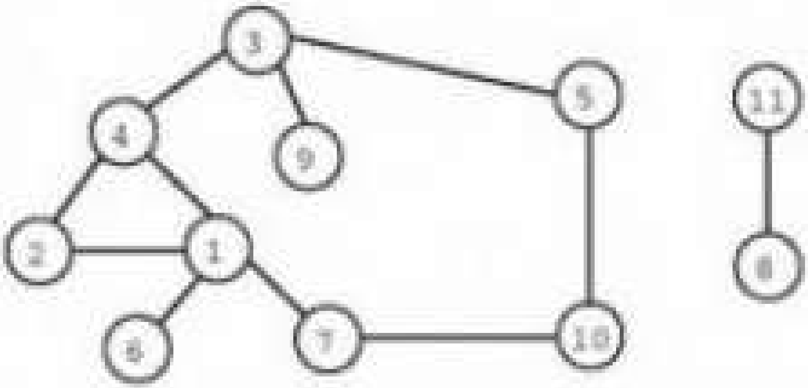
Parcurgerea începe cu un vârf iniţial, denumit vârf de start. Se vizitează mai întâi vârful de start. La vizitarea unui vârf se efectuează asupra informaţiilor asociate vârfului o serie de operaţii specifice problemei.

Se vizitează apoi primul vecin nevizitat al vârfului de start. Vârful `y` este considerat vecin al vârfului `x` dacă există muchia `[x,y]` (pentru graf neorientat), respectiv arcul `(x,y)` (pentru graf orientat).

Se vizitează în continuare primul vecin nevizitat al primului vecin al vârfului de start, şi aşa mai departe, mergând în adâncime până când ajungem într-un vârf care nu mai are vecini nevizitaţi. Când ajungem într-un astfel de vârf, revenim la vârful său părinte (vârful din care acest nod a fost vizitat). Dacă acest vârf mai are vecini nevizitaţi, alegem primul vecin nevizitat al său şi continuăm parcurgerea în acelaşi mod. Dacă nici acest vârf nu mai are vecini nevizitaţi, revenim în vârful său părinte şi continuăm în acelaşi mod, până când toate vârfurile accesibile din vârful de start sunt vizitate.

*Exemplu*

Să parcurgem în adâncime graful din figura următoare, considerând drept vârf de start vârful 3.




Se vizitează mai întâi vârful de start 3. Apoi se vizitează primul vecin nevizitat al lui 3 (ca ordine a vecinilor vom considera ordinea crescătoare a numerelor lor), deci 4. Vizităm apoi primul vecin nevizitat al lui 4, adică pe 1. Apoi vizităm primul vecin nevizitat al lui 1, adică pe 2. În acest moment suntem într-un nod care nu mai are vecini nevizitați, revenim în nodul său părinte, adică în 1. Vârful 1 mai are vecini nevizitați, îl vizităm pe primul dintre aceștia, vârful 6. Vârful 6 nu are vecini nevizitați, deci vom reveni în vârful 1, părintele său. Vârful 1 mai are un vecin nevizitat, vârful 7. Vizităm vârful 7, apoi primul vecin nevizitat al lui 7, vârful 10, apoi primul vecin nevizitat al lui 10, vârful 5. Vârful 5 nu mai are vecini nevizitați, deci revenim în 10. Nici vârful 10 nu mai are vecini nevizitați, deci revenim în 7. Nici vârful 7 nu mai are vecini nevizitați, revenim în 1, apoi revenim în 4, apoi în 3. Vârful 3 mai are un vecin nevizitat - vârful 9. Vizităm vârful 9, apoi, deoarece vârful 9 nu are vecini nevizitați, revenim în vârful 3. Cum vârful 3 nu mai are vecini nevizitați și nici părinte (fiind vârful de start), parcurgerea s-a încheiat.

Concluzionând, ordinea în care sunt vizitate vârfurile grafului la parcurgerea DFS cu vârful de start 3 este: 3, 4, 1, 2, 6, 7, 10, 5, 9.

Vârfurile 8 și 11 nu au fost vizitate, deoarece nu sunt accesibile din vârful 3.

Analizând parcurgerea în adâncime, deducem că vârfurile sunt explorate în ordinea inversă a „atingerii" lor, mecanism care poate fi implementat utilizând o stivă. Prin urmare, pentru concizie și claritate se impune o abordare recursivă a parcurgerii DFS.

*Reprezentarea informațiilor*

1. Graful va fi reprezentat prin liste de adiacență, memorate în tabloul A; pe poziția 0 a fiecărei liste de adiacență se află numărul de vârfuri din listă.
2. Pentru a reține care vârfuri au fost deja vizitate în timpul parcurgerii vom utiliza un vector viz, cu n componente din mulțimea {0, 1}, cu semnificația viz[i]=1 dacă vârful i a fost deja vizitat, respectiv 0, în caz contrar.

Considerăm că variabilele n (numărul de vârfuri din graf), A (listele de adiacență) și viz sunt globale. De asemenea, considerăm că la vizitarea unui vârf va fi afișat pe ecran numărul acestuia.

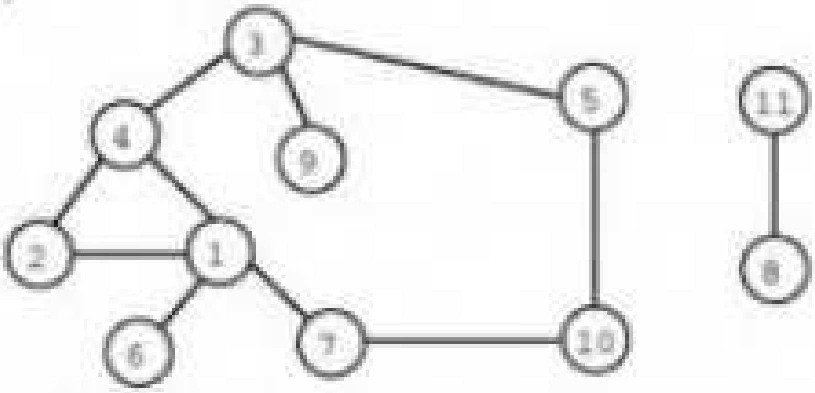
```
void DFS(int x)
{int i;
 //vizitam varful x
 printf("%d ", x);
 viz[x]=1;
 //parcurgem lista de adiacenta a varfului x
 for (i=1; i<=A[x][0]; i++)
     if (!viz[A[x][i]])
        //A[x][i] este un vecin nevizitat al lui x
        DFS(A[x][i]);
}
```

## *Parcurgerea în lățime*

Parcurgerea în lățime începe, de asemenea, cu un vârf inițial, denumit vârf de start. Se vizitează mai întâi vârful de start. Se vizitează în ordine toți vecinii nevizitați ai vârfului de start. Apoi se vizitează în ordine toți vecinii nevizitați ai vecinilor vârfului de start și așa mai departe, până la epuizarea tuturor vârfurilor accesibile din vârful de start.

*Exemplu*

Să parcurgem în lățime graful parcurs deja în adâncime, considerând drept vârf de start vârful 3.




Se vizitează mai întâi vârful de start 3. Apoi se vizitează, în ordine, vecinii nevizitați ai lui 3, deci 4, 5 și 9. Se vizitează apoi, în ordine, vecinii nevizitați ai lui 4 (vârfurile 1 și 2), apoi ai lui 5 (vârful 10) și apoi ai lui 9 (care nu are vecini nevizitați). Se vizitează apoi vecinii vârfului 1 (vârfurile 6 și 7) și parcurgerea s-a încheiat (deoarece vârful 2 nu mai are vecini nevizitați, nici vârful 10 și nici vârfurile 6 și 7).

Concluzionând, ordinea în care sunt vizitate vârfurile grafului la parcurgerea BFS cu vârful de start 3 este: 3, 4, 5, 9, 1, 2, 10, 6, 7.

Observați că și în cazul parcurgerii în lățime vârfurile 8 și 11 nu au fost vizitate, deoarece nu sunt accesibile din vârful 3.

Analizând parcurgerea în lățime deducem că vârfurile sunt explorate exact în ordinea „atingerii" lor, mecanism care poate fi implementat utilizând o coadă.

*Descrierea algoritmului*

1. Inițializăm coada cu vârful de start și vizităm vârful de start.
2. Cât timp există elemente în coadă executăm:
   - extragem din coadă primul element;
   - parcurgem toți vecinii elementului extras, identificându-i pe cei nevizitați; aceștia vor fi vizitați și vor fi plasați în coadă.

*Reprezentarea informațiilor*

1. Graful va fi reprezentat prin liste de adiacență, memorate în tabloul `A`; pe poziția `0` a fiecărei liste de adiacență se află numărul de vârfuri din listă.
2. Pentru a reține care vârfuri au fost deja vizitate în timpul parcurgerii, vom utiliza un vector `viz`, cu `n` componente din mulțimea `{0, 1}`, cu semnificația `viz[i]=1` dacă vârful `i` a fost deja vizitat, respectiv `0` în caz contrar.
3. Vom utiliza o coadă implementată static într-un vector `C` cu `n` elemente, în care reținem vârfurile în ordinea vizitării lor. Variabile `prim` și `ultim` rețin poziția de început, respectiv poziția de sfârșit în coadă.

Considerăm că variabilele `n` (numărul de vârfuri din graf), `A` (listele de adiacență), `C` (coada) și `viz` sunt globale. De asemenea, considerăm că la vizitarea unui vârf va fi afișat pe ecran numărul acestuia.

```
void BFS(int x)
{int i, prim, ultim;
 //vizitam varful de start
 printf("%d ", x);
 viz[x]=1;
 //initializam coada cu varful de start
 C[0]=x; prim=ultim=0;
 while (prim<=ultim) //cat timp coada nu este vida
       {//extragem un element din coada
        x=C[prim++];
        //parcurgem lista de adiacenta a varfului x
        for (i=1; i<=A[x][0]; i++)
             if (!viz[A[x][i]])
                 //A[x][i] este un vecin nevizitat al lui x
                 {//il vizitam
                  printf("%d ", A[x][i]);
                  viz[A[x][i]]=1;
                  C[++ultim]=A[x][i]; //il plasam in coada
                 }
       }
}
```

*Observații*

1. Parcurgerea în lățime are o proprietate remarcabilă: fiecare vârf este vizitat pe cel mai scurt drum/lanț începând din vârful de start.
2. Complexitatea timp a algoritmilor de parcurgere în adâncime și în lățime a unui graf depinde de modalitatea de reprezentare a acestuia. În cazul reprezentării prin liste de adiacență, complexitatea este O($n+m$). În cazul reprezentării prin matrice de adiacență, complexitatea este O($n^2$).

## *Aplicații*

*Determinarea celui mai scurt drum/lanț între două vârfuri*

Fie G un graf (orientat sau neorientat) și x, y două vârfuri din graf. Să se determine cel mai scurt drum (respectiv lanț, pentru cazul în care graful este neorientat) de la x la y.

*Soluție*

Vom utiliza proprietatea parcurgerii în lățime de a vizita fiecare vârf pe cel mai scurt drum/lanț care pleacă din vârful de start. Prin urmare, pentru a determina cel mai scurt drum de la x la y, vom efectua o parcurgere în lățime începând din x, până când atingem vârful y, sau până când vizităm toate vârfurile accesibile din x.

Pentru a reconstitui drumul, vom modifica semnificația vectorului viz. Mai exact, viz[i]=j dacă j este vârful părinte al lui i, adică dacă vârful i a fost vizitat prin parcurgerea arcului (j,i) (sau a muchiei [j,i]), respectiv 0 dacă vârful i nu a fost vizitat. Singurul vârf vizitat care nu are vârf părinte este vârful de start. Prin convenție, stabilim viz[x]=-1.

Funcția de parcurgere în lățime, modificată astfel încât să permită reconstituirea celui mai scurt drum/lanț de la x la y este:

```
void BFS(int x)
{int i, prim, ultim;
 viz[x]=-1; //vizitam varful de start
 C[0]=x; prim=ultim=0; //initializam coada
 while (prim<=ultim && !viz[y])
     //cat timp coada nu este vida si nu am vizitat varful y
     {x=C[prim++]; //extragem un element din coada
      //parcurgem lista de adiacenta a varfului x
      for (i=1; i<=A[x][0]; i++)
          if (!viz[A[x][i]])
              //A[x][i] este un vecin nevizitat al lui x
              {viz[A[x][i]]=x;      //il vizitam
               C[++ultim]=A[x][i]; } //il plasam in coada
      }
}
```

Reconstituirea drumului/lanțului se face în sens invers, pornind de la vârful y, determinând apoi părintele său (memorat în viz[y]), apoi părintele părintelui său (memorat în viz[viz[y]]) și așa mai departe, până când întâlnim un vârf care nu are părinte (acesta este vârful de start x).

Pentru a afișa drumul în ordinea firească, îl vom memora într-un vector auxiliar, denumit drum, apoi vom afișa vectorul de la sfârșit către început.