

Cuprins

1. Teoria grafurilor.....	7
1.1. Noțiuni introductive.....	7
1.2. Reprezentarea grafurilor în memorie.....	24
1.3. Grafuri ponderate. Reprezentare.....	36
1.4. Închiderea tranzitivă a unui graf.....	37
1.5. Parcurgerea grafurilor.....	38
1.6. Conexitate.....	46
1.7. Tare-conexitate.....	51
1.8. Arbori.....	54
1.9. Arbori parțiali.....	58
1.10. Arbori parțiali de cost minim.....	62
1.11. Biconexitate.....	74
1.12. Descompunere pe niveluri a unui graf fără circuite.....	78
1.13. Sortare topologică.....	81
1.14. Grafuri hamiltoniene.....	83
1.15. Grafuri euleriene.....	84
1.16. Drumuri minime în graf.....	87
1.17. Rețele de transport.....	95
1.18. Cuplaj maximal în graf bipartit.....	101
1.19. Aplicații rezolvate.....	103
1.20. Aplicații propuse.....	135
2. Liste înlanțuite.....	149
2.1. Liste simplu înlanțuite.....	149
2.2. Liste simplu înlanțuite circulare.....	160
2.3. Liste dublu înlanțuite.....	163
3. Structuri de date arborescente.....	167
3.1. Terminologie.....	167
3.2. Arbori binari.....	168
3.3. Reprezentarea arborilor cu rădăcină.....	172
3.4. Crearea unui arbore binar.....	175
3.5. Parcurgerea arborilor binari.....	177

3.6. Determinarea înălțimii unui arbore	179
3.7. Crearea unui arbore binar pe baza parcurgerilor în preordine și înordine.....	179
3.8. <i>Heap</i> -uri.....	182
3.9. Arbori binari de căutare	198
3.10. Reprezentarea mulțimilor disjuncte	208
3.11. Arbori de compresie Huffman	212
3.12. Arbori asociați expresiilor aritmetice	216
3.13. Aplicații rezolvate.....	219
3.14. Aplicații propuse	226
4. <i>Pattern-matching</i>	233
4.1. Introducere.....	233
4.2. Algoritmul elementar (naïv)	233
4.3. Algoritmul Knuth-Morris-Pratt (KMP)	234
4.4. Aplicație. Deșert	236
4.5. Probleme propuse.....	238
5. <i>Hashing</i>	240
5.1. Introducere.....	240
5.2. Funcții <i>hash</i>	240
5.3. Rezolvarea coliziunilor	242
5.4. <i>Hashing</i> perfect	244
5.5. Algoritmul Rabin Karp	245
5.6. Probleme propuse.....	247
6. Geometrie computațională	251
6.1. Introducere.....	251
6.2. Puncte și drepte	251
6.3. Poligoane.....	255
6.4. Geometria dreptunghiului	266
6.5. Aplicații	269
6.6. Probleme propuse.....	278
7. Soluții și indicații	284
Bibliografie	293

1.4. Închiderea tranzitivă a unui graf

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

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

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

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

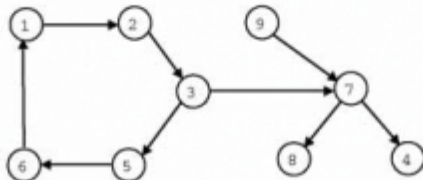
```
void RW()
{
    int k, i, j;
    for (k=1; k<=n; k++)
        for (i=1; i<=n; i++)
            for (j=1; j<=n; j++)
                AP[i][j]=AP[i][j] || AP[i][k] && AP[k][j];
}
```

Observații

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

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

1.5. Parcurgerea grafurilor

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

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

Parcurgerea în adâncime

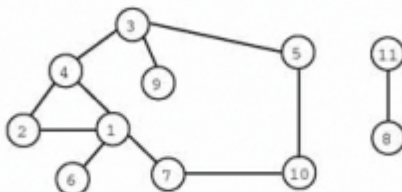
Parcurgerea începe cu un vârf inițial, denumit vârf de start. Se vizitează mai întâi vârfurile de start. La vizitarea unui vârf se efectuează asupra informațiilor asociate vârfurilor o serie de operații specifice problemei.

Se vizitează apoi primul vecin nevizitat al vârfului de start. Vârfurile y este considerat vecin al vârfurilor 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, și așa mai departe, mergând în adâncime până când ajungem într-un vârf care nu mai are vecini nevizitați. Când ajungem într-un astfel de vârf, revenim la vârfurile său părinte (vârfurile din care acest nod a fost vizitat). Dacă acest vârf mai are vecini nevizitați, alegem primul vecin nevizitat al său și continuăm parcurgerea în același mod. Dacă nici acest vârf nu mai are vecini nevizitați, revenim în vârfurile său părinte și continuăm în același mod, până când toate vârfurile accesibile din vârfurile 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. Vârfurile 1 mai are vecini nevizitați, îl vizităm pe primul dintre aceștia, vârfurile 6. Vârfurile 6 nu are vecini nevizitați, deci vom reveni în vârfurile 1, părintele său. Vârfurile 1 mai are un vecin nevizitat, vârfurile 7. Vizităm vârfurile 7, apoi primul vecin nevizitat al lui 7, vârfurile 10, apoi primul vecin nevizitat al lui 10, vârfurile 5. Vârfurile 5 nu mai are vecini nevizitați, deci revenim în 10. Nici vârfurile 10 nu mai are vecini nevizitați, deci revenim în 7. Nici vârfurile 7 nu mai are vecini nevizitați, revenim în 1, apoi revenim în 4, apoi în 3. Vârfurile 3 mai are un vecin nevizitat - vârfurile 9. Vizităm vârfurile 9, apoi, deoarece vârfurile 9 nu are vecini nevizitați, revenim în vârfurile 3. Cum vârfurile 3 nu mai are vecini nevizitați și nici părinte (fiind vârfurile de start), parcurgerea s-a încheiat.

Concluzionând, ordinea în care sunt vizitate vârfurile grafului la parcurgerea DFS cu vârfurile 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 vârfurile 3.

Analizând parcurgerea în adâncime, deducem că vârfurile sunt explorate în ordinea inversă a „atingerilor” 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 vârfurile 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]);
}

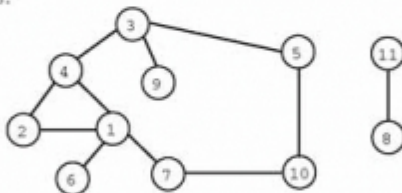
```

Parcurea în lățime

Parcurea 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 parcurea 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 parcurea BFS cu vârful de start 3 este: 3, 4, 5, 9, 1, 2, 10, 6, 7.

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

Analizând parcurea 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.