

Cuprins

I. Teoria grafurilor	7
1.1. Noțiuni introductive	7
1.2. Reprezentarea grafurilor în memorie	24
1.3. Grafuri ponderate. Reprezentare	36
1.4. Închiderea transițivă 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 maximul în graf bipartit	101
1.19. Aplicații rezolvate	103
1.20. Aplicații propuse	135
2. Liste înălungite	149
2.1. Liste simplu înălungite	149
2.2. Liste simplu înălungite circulare	160
2.3. Liste dublu înălungite	163
3. Structuri de date arborescente	167
3.1. Terminologie	167
3.2. Arbori binari	168
3.3. Reprezentarea arborilor cu rădăcini	172
3.4. Crearea unui arbore binar	175
3.5. Parcurgerea arborilor binari	177

3.6. Desemnarea înălțimii unui arbore	179
3.7. Crearea unui arbore binar pe baza parcurgerilor în preordine și în ordine.....	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. Pattern-matching	233
4.1. Introducere	233
4.2. Algoritmul elementar (naiv)	233
4.3. Algoritmul Knuth-Morris-Pratt (KMP)	234
4.4. Aplicație. Deser	236
4.5. Probleme propuse	238
5. Hashing	240
5.1. Introducere	240
5.2. Funcții hash	240
5.3. Rezolvarea coliziunilor	242
5.4. Hashing 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 patratică A^* având n linii și n coloane (unde n reprezintă numărul de vîrfuri 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 vîrf intermediar k și verificăm pentru fiecare perche de vîrfuri (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.

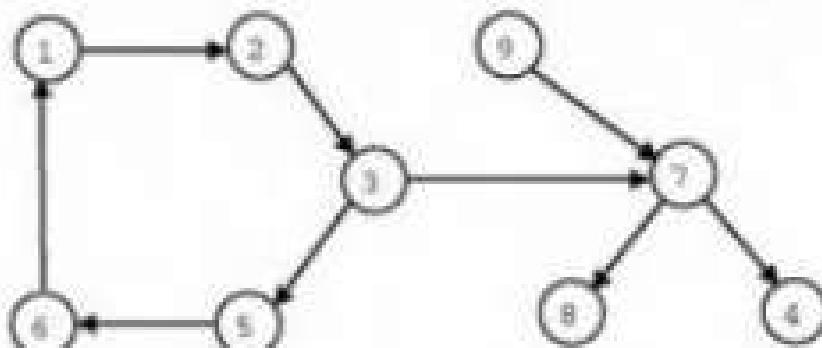
```
void Roy()
{
    int k, i, j;
    for (k=1; k<n; k++)
        for (i=1; i<n; i++)
            for (j=1; j<n; j++)
                AP[i][j] = AP[i][j] || AP[i][k] && AP[k][j];
}
```

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 vîrful 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 matricei 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 vîrfurile care nu aparțin nici unui circuit.

1	1	1	1	1	1
1	1	1	1	1	1
0	0	0	0	1	
1	1	1	1	1	
1	1	1	1	1	
0	0	0	0	0	

3. Scrieți un program care, utilizând matricea închiderii tranzitive, să determine toate vîrfurile accesibile dintr-un vîrf dat x . Spunem că vîrful y este accesibil din vîrful 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 vîrfurilor grafului, cu scopul preluării informațiilor asociate vîrfurilor.

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

Parcurgerea în adâncime

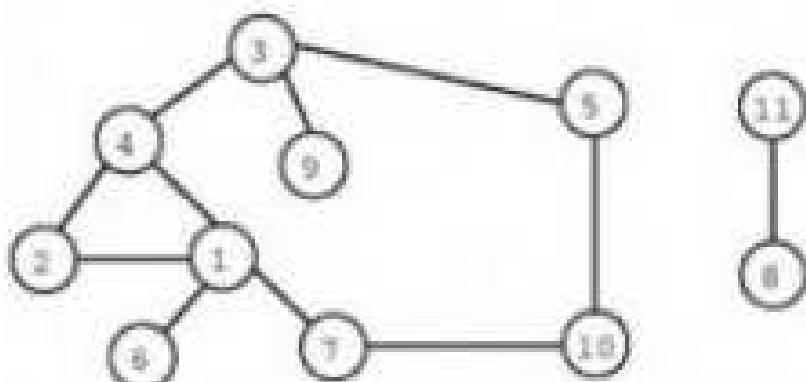
Parcugerea începe cu un vîrf inițial, denumit vîrf de start. Se viziteză mai întâi vîrful de start. La vizitarea unui vîrf se efectuează asupra informațiilor asociate vîrfului o serie de operații specifice problemei.

Se viziteză apoi primul vecin nevizitat al vîrfului de start. Vîrful y este considerat vecin al vîrfului x dacă există muchia (x, y) (pentru graf neorientat), respectiv arcul (x, y) (pentru graf orientat).

Se viziteză în continuare primul vecin nevizitat al primului vecin al vîrfului de start, și așa mai departe, mergând în adâncime până când ajungem într-un vîrf care nu mai are vecini nevizitați. Când ajungem într-un astfel de vîrf, revenim la vîrful său părinte (vîrful din care acest nod a fost vizitat). Dacă acest vîrf mai are vecini nevizitați, alegem primul vecin nevizitat al său și continuăm parcugerea în același mod. Dacă nici acest vîrf nu mai are vecini nevizitați, revenim în vîrful său părinte și continuăm în același mod, pînă când toate vîrfurile accesibile din vîrful de start sunt vizitate.

Exemplu

Să parcurgem în adâncime graful din figura următoare, considerând drept vîrf de start vîrful 3.



Se vizitează mai întâi vîrful 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. Vîrful 1 mai are vecini nevizitați, îl vizităm pe primul dintr-oarece acesta, vîrful 6. Vîrful 6 nu are vecini nevizitați, deci vom reveni în vîrful 1, părintele său. Vîrful 1 mai are un vecin nevizitat, vîrful 7. Vizităm vîrful 7, apoi primul vecin nevizitat al lui 7, vîrful 10, apoi primul vecin nevizitat al lui 10, vîrful 5. Vîrful 5 nu mai are vecini nevizitați, deci revenim în 10. Nici vîrful 10 nu mai are vecini nevizitați, deci revenim în 7. Nici vîrful 7 nu mai are vecini nevizitați, revenim în 1, apoi revenim în 4, apoi în 3. Vîrful 3 mai are un vecin nevizitat - vîrful 2. Vizităm vîrful 2, apoi, deoarece vîrful 9 nu are vecini nevizitați, revenim în vîrful 3. Cum vîrful 3 nu mai are vecini nevizitați și nici părinte (fiind vîrful de start), parcurgerea s-a încheiat.

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

Vîrfurile 8 și 11 nu au fost vizitate, deoarece nu sunt accesibile din vîrful 3.

Analizând parcurgerea în adâncime, deducem că vîrfurile sunt explorate în ordinea inversă a „stingerii” 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 tabelul A; pe poziția i a fiecărei liste de adiacență se află numărul de vîrfuri din listă.
2. Pentru a reține care vîrfuri 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 vîrful i a fost deja vizitat, respectiv 0, în caz contrar.

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

```

void DFSlist (x)
{
    int i;
    //vizitam varful x
    printf("vd ", x);
    vizit(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]);
}

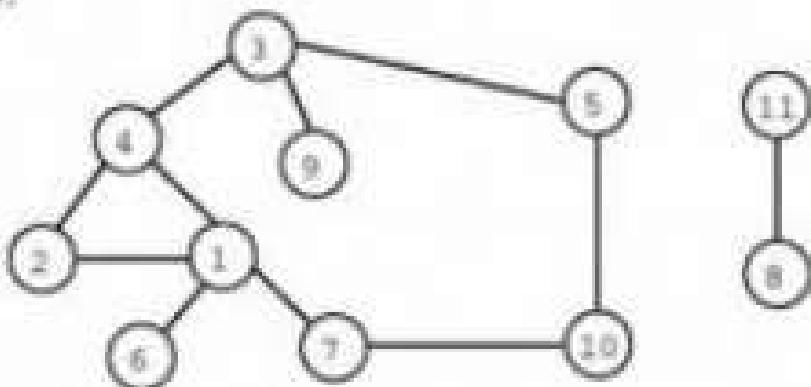
```

Parcurgerea înălțime

Parcurgerea înălțime 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 înălțimea 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 7. Se vizitează apoi, în ordine, vecinii nevizitați ai lui 4 (vârfurile 2 și 9). 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 în parcurgerea BFS cu vârful de start 3 este: 3, 4, 5, 9, 1, 2, 10, 6, 7.

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

Analizând parcurgerea înălțime deducem că vârfurile sunt explorate exact în ordinea „stingerii” 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;
 - parcugem 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 tabelul A ; pe poziția i 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 $\text{viz}[i]=1$ dacă vârful i a fost deja vizitat, respectiv 0 în caz contrar.
3. Vom utiliza o coadă implementată static intr-un vector C cu n elemente, în care reținem vârfurile în ordinea vizitării lor. Variabilele prim și ult im rezin 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 l, prim, ultim;
    //vizitam vârful de start
    printf("%d ", x);
    viz[x]=1;
    //initializam coada cu vârful de start
    C[0]=x; prim=ultim=0;
    while (prim<=ultim) //cât timp coada nu este vidă
        //extragem un element din coada
        x=C[prim++];
        //parcugem lista de adiacență a vârfului x
        for (l=1; l<=A[x][0]; l++)
            if ((viz[A[x][l]]==0))
                //A[x][l] este un vecin nevizitat al lui x
                //l vizitam
                printf("%d ", A[x][l]);
                viz[A[x][l]]=1;
                C[++ultim]=A[x][l]; //l plasat în coada
}
```

Observații

1. Parcurgerea înălțime 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 înălțime 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/lant între două vârfuri

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

Soluție

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

Pentru a reconstituia drumul, vom modifica semnificația vectorului viz . Mai exact, $viz[i] = j$ dacă j este vîrful parinte al lui i , adică dacă vîrful i a fost vizitat prin parcurgerea arcului (j, i) (sau a muchiei (j, i)), respectiv \circ dacă vîrful i nu a fost vizitat. Singurul vîrf vizitat care nu are vîrf parinte este vîrful de start. Prin convenție, stabilim $viz[x] = -1$.

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

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

Reconstituirea drumului/lantului se face în sens invers, pornind de la vîrful y , determinând apoi părțile său (menținut în $viz(y)$), apoi părțile părțiselui său (menținut în $viz(viz(y))$) și aşa mai departe, până când întâlnim un vîrf care nu are părinte (acesta este vîrful 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.