7 Laborator 7: Grafuri - reprezentare, parcurgere BFS

7.1 Obiective

Scopul acestui laborator este de a prezenta câteva noțiuni legate de grafuri, modurile de reprezentare a lor în aplicații și parcurgerea lor în lățime.

7.2 Noțiuni teoretice

7.2.1 De ce grafuri?

În multe aplicații e necesar să memorăm informații referitor la conectivitate sau relații între perechi de obiecte, de ex. rețele sociale, rețele de drumuri, rețele de calculatoare, diferite jocuri pe calculator, circuite electrice etc. De exemplu, în rețelele de calculatoare există calculatoare care sunt conectate prin diverse legături (router, switch, cabluri, wireless). Informația, sub formă de pachete, trece de la un calculator la altul prin conexiunile intermediare.

În unele situații ne interesează să găsim cea mai scurtă cale între structurile care sunt conectate. De exemplu, ca pachetele să ajungă de la un calculator la altul, am prefera să treacă prin cât mai puține conexiuni intermediare. În alte aplicații ne interesează dacă există o cale între două astfel de structuri.

Grafurile reprezintă o modalitate de a reprezenta în aplicații informații despre conectivitate, cu scopul de a trata eficient astfel de informații. Pe scurt, un graf este un set de obiecte numite *vârfuri* sau *noduri* (*vertices*), împreună cu o colecție de conexiuni (*arce* sau *muchii* (*edges*)), între perechi de obiecte din acel set.

Prezentăm în continuare unele noțiuni de bază care descriu diferite componente și proprietăți ale grafurilor, precum și moduri de reprezentare a grafurilor și traversarea lor în lățime.

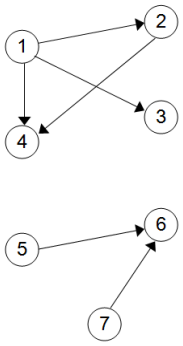
7.2.2 Noțiuni de bază

Graf orientat sau *digraf* $G = (V, E)$, este perechea formată din mulțimea V de vârfuri și mulțimea $E \subseteq V \times V$ de arce (exemplu: figura 7.1).
Arc: o pereche ordonată de vârfuri (v, w), unde v este *baza* arcului, iar w este *vârful* arcului. În alți termeni se spune că w este *adiacent* lui v sau că arcul este *incident* lui v sau w, dacă nu se precizează orientarea arcului.
Arc care iese dintr-un vârf v este un arc de forma (v, w).
Arc care intră într-un vârf v este un arc de forma (w, v).
Cale: o succesiune de vârfuri $v[1], v[2], ..., v[k]$, astfel că există arcele $(v[1], v[2]), (v[2], v[3]), ..., (v[k-1], v[k])$ în mulțimea arcelor E. *Lungimea căii* este numărul de arce din cale. Prin convenție, calea de la un nod la el însuși are lungimea 0.
Cale simplă: O cale este simplă, dacă toate vârfurile, cu excepția primului și ultimului sunt distincte între ele.
Ciclu: Un ciclu este o cale de la un vârf la el însuși.
Etichetare: Un graf orientat etichetat este un graf în care fiecare arc și/sau vârf are o etichetă asociată, care poate fi un nume, un cost sau o valoare de un tip oarecare.
Graf tare conex: Un graf orientat este tare conex, dacă oricare ar fi vârfurile v și w există o cale de la v la w și una de la w la v.
Subgraf: Un graf $G' = (V', E')$ este subgraf al lui G dacă $V' \subset V$ și $E' \subset E$. Se spune că subgraful indus de $V' \subset V$ este $G' = (V', E \subset (V' \times V'))$.
Componentă tare conexă a unui graf orientat este un subgraf maximal tare conex al grafului.
Graf neorientat (notat și $G = (N, R)$), este perechea formată din mulțimea N de noduri și mulțimea R de muchii.
Muchie este o pereche neordonată $(v, w) = (w, v)$ de noduri. Definițiile prezentate rămân valabile și în cazul grafurilor neorientate.

7.2.3 Moduri de reprezentare

Există două structuri de date utilizate adesea pentru reprezentarea grafurilor: *matrice de adiacență* și *liste de adiacență*. În ambele reprezentări, dacă trebuie memorate informații suplimentare despre vârfuri sau muchii/arce, se presupune că există o modalitate de mapare a vârfurilor și muchiilor/arcelor cu datele asociate lor.

De exemplu, se poate utiliza o structură de căutare (tabelă de dispersie, arbore binar de căutare etc.) pentru a memora informațiile, utilizând ca și chei numele nodurilor, respectiv ale muchiilor/arcelor. Sau putem reprezenta chiar vârfurile,



Vârfuri :

```
{1, 2, 3, 4, 5, 6, 7}
```

Arce:

```
{1,2},{1,3},{1,4},{2,4},{5,6},{7,6}
```

Figure 7.1: Un graf format din două componente conexe, cu vârfurile și arcele date

respectiv muchiile ca obiecte (structuri) având mai multe câmpuri, în care memorăm atât numele lor cât și informațiile asociate.

Cele două reprezentări, prin matrice de adiacență și liste de adiacență, se utilizează atât pentru grafurile orientate, cât și pentru cele neorientate. Principala diferență dintre reprezentări este că algoritmii pentru diverse operații pe grafuri pot avea performanțe diferite în funcție de reprezentarea aleasă. De asemenea, pentru un graf cu *n* vârfuri și *m* muchii/arce, reprezentarea cu liste de adiacență necesită un spațiu $O(n+m)$, în timp ce reprezentarea cu matrice de adiacență necesită un spațiu $O(n^2)$.

Matrice de adiacență

Pentru un graf orientat $G = (V, E)$ unde V este mulțimea vârfurilor numerotate $\{1, 2, \ldots, n\}$, putem privi arcele ca fiind perechi de astfel de valori. Matricea de adiacență, A, este o matrice $n \times n$, definită astfel:

$$A[i][j] = \begin{cases} 1, & \text{daca } (i,j) \in E \\ 0, & \text{daca } (i,j) \notin E \end{cases}$$

(vezi un exemplu în figura 7.2). Pentru un graf neorientat, matricea A va fi simetrică.

Dacă vârfurile au și informații asociate (nume și/sau alte caracteristici), ele se vor putea memora așa cum s-a amintit mai sus, și e necesară o modalitate de mapare a numerotării vârfurilor la informațiile memorate.

În cazul în care fiecare arc/muchie are asociată o valoare (etichetă), matricea de adiacență devine matrice etichetată (sau matricea costurilor), definită astfel:

$$A[i][j] = \begin{cases} \text{eticheta arcului } (i,j), & \text{daca } (i,j) \in E \\ \text{un simbol special}, & \text{daca } (i,j) \notin E \end{cases}$$

Dacă arcele/muchiile au mai multe informații asociate (nu doar un cost sau o etichetă), acele informații se pot memora cum s-a amintit, în obiecte (structuri) alocate dinamic, iar în matricea de adiacență elementul $A[i,j]$ va memora adresa unei astfel structuri e, dacă există arcul/muchia $e = (i,j)$ în graf, și `null` dacă nu există.

Matricea de costuri, ca și cea de adiacență, este simetrică pentru grafuri neorientate și în general este nesimetrică pentru cele orientate.

Reprezentarea prin matrice de adiacență/de costuri permite verificarea într-un timp constant a faptului că o pereche de vârfuri sunt adiacente, iar obținerea tuturor arcelor/muchiilor incidente unui vârf durează un timp $O(n)$. În schimb, indiferent de numărul de muchii, spațiul necesar pentru reprezentarea unui graf cu n vârfuri este $O(n^2)$, așa cum s-a văzut.

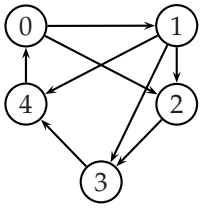
Grafurile pot fi reprezentate prin matrici prin următoarea structură:

```
typedef struct {
        int n; // numarul de noduri
        int ** m; // matricea de adiacenta (daca are valori intregi)
} Graf;
```

Observație: Trebuie ținut cont de faptul că, în limbajul C, indicii tablourilor pornesc de la 0. Astfel, în implementare vom presupune că vârfurile sunt numerotate de la 0 la $n-1$, și nu de la 1 la n.

Liste de adiacențe

Reprezentarea prin liste de adiacențe folosește mai bine memoria în cazul în care numărul de arce/muchii este mult mai mic decât $O(n^2)$, dar în general căutarea arcelor este mai greoaie. Reprezentarea prin liste de adiacență a unui graf $G = (V, E)$ presupune existența următoarelor:



(a) Graf orientat.

	0	1	2	3	4
0	0	1	1	0	0
1	0	0	1	1	1
2	0	0	0	1	0
3	0	0	0	0	1
4	1	0	0	0	0

(b) Matrice de adiacență pentru graful din figura Figure 7.2(a)

Figure 7.2: Un graf orientat și matricea sa de adiacență

- colecția V a celor n vârfuri, numerotate de la 1 la n. Ca și în cazul reprezentării anterioare, această colecție poate însemna simplu numerele de la 1 la n, sau poate fi reprezentată prin diverse structuri de date, caz în care se presupune existența unei mapări între valorile de la 1 la n și informațiile din structura asociată.

- colecția E de m arce/muchii, adică perechi de vârfuri. Această colecție poate fi definită fie doar prin perechile de vârfuri, fie să aibă și informații asociate fiecărui arc/muchie, caz în care din nou e necesară o mapare.

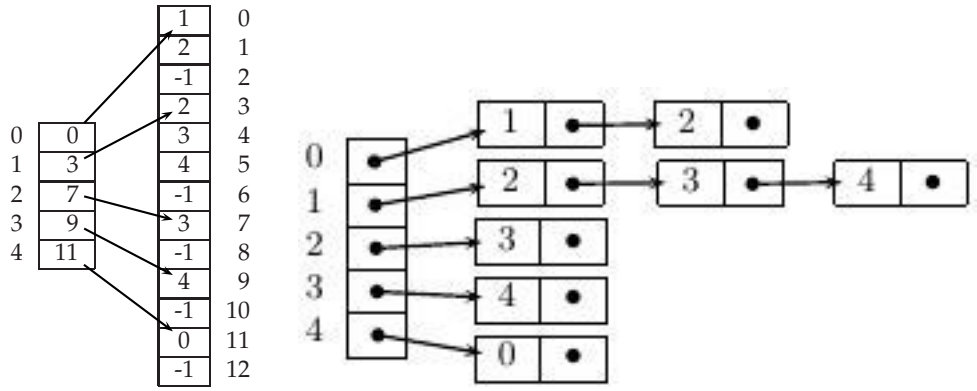
- pentru fiecare vârf v din V, se memorează o listă, numită *lista de adiacență pentru* v, care reprezintă arcele/muchiile incidente cu v. Concret, aceasta se poate implementa fie ca lista tuturor vârfurilor w astfel încât $(v, w) \in E$, fie ca o listă de arce/muchii incidente cu v. În acest ultim caz, dacă G este un graf orientat, lista de adiacență pentru v trebuie împărțită în două părți: una care reprezintă arcele care ies din v, cealaltă reprezentând arcele care intră în v.

Se poate vedea că spațiul necesar pentru această reprezentare este $O(n + m)$. În schimb, dacă notăm cu $\texttt{deg}(v)$ gradul unui vârf v, atunci:

- obținerea arcelor/muchiilor incidente unui vârf durează un timp $O(deg(v))$.

- determinarea adiacenței a două vârfuri u și v se poate face în timp $O(min\{deg(u), deg(v)\})$.

Deci, în această reprezentare, întregul graf poate fi reprezentat printr-un tablou indexat după noduri, fiecare intrare în tablou conținând adresa listei nodurilor adiacente. Lista nodurilor adiacente poate fi *dinamică* sau *statică*. În cazul listei dinamice (fig. 7.3(b)), lista nodurilor adiacente pentru fiecare nod este o listă simplu înlănțuită, cu nodurile alocate dinamic, iar în tabloul indexat după noduri, intrarea i în tablou conține adresa primului nod al listei de noduri adiacente cu nodul i. În schimb, pentru lista de adiacență statică (fig. 7.3(a)) se folosesc două tablouri. Primul (cel din dreapta în figură) memorează lista nodurilor adiacente cu fiecare nod (fiecare astfel de listă se încheie cu o valoare -1 care semnifică sfârșit de listă). Al doilea tablou (cel din stânga) are dimensiunea n (numărul de noduri) și memorează, pentru fiecare nod i, indicele (din tabloul din dreapta) de unde începe lista nodurilor adiacente cu nodul i.



(a) Liste de adiacență statice.

(b) Liste de adiacență dinamice.

Figure 7.3: Listele de adiacență pentru graful din figura 7.2(a)

7.2.4 Explorarea (căutarea) în lățime

Explorarea unui graf se referă la o operație asupra unui graf, necesară în diverse aplicații, și anume traversarea vârfurilor și muchiilor/arcelor grafului. Mai precis, o **traversare** este un algoritm de explorare a unui graf constând în examinarea tuturor vârfurilor și muchiilor/arcelor sale. O traversare este eficientă dacă vizitează toate vârfurile și arcele/muchiile în timp liniar (proporțional cu numărul lor).

De exemplu, un *web spider* sau *crawler*, care este partea de colectare a datelor dintr-un motor de căutare, trebuie să exploreze un graf de documente hipertext prin examinarea vârfurilor, care sunt documentele, și a arcelor, care sunt hiperlegăturile dintre documente.

Explorarea (căutarea) în lățime (BFS, *Breadth-First Search*) poate fi utilizată pentru rezolvarea mai multor probleme care implică grafuri, de exemplu găsirea celei mai scurte căi (drum) dintre două vârfuri (unde lungimea căii se măsoară prin numărul de arce/muchii), metoda de calcul a fluxului maxim (Ford-Fulkerson) într-o rețea de transport, verificarea faptului că un graf este bipartit etc.

Explorarea în lățime pornește de la un vârf al grafului și explorează mai întâi vârfurile vecine, înainte de a trece la vecinii de la nivelul următor (vecinii vecinilor) etc. Nivelul unui vârf, în acest caz, reprezintă numărul minim de muchii de la vârful de start până la acel vârf. Explorarea în lățime constă din următoarele acțiuni:

1. Se trece într-o coadă vidă nodul (vârful) de pornire;
2. Se trece extrage din coadă câte un nod, care este prelucrat (vizitat) și apoi se adaugă toate nodurile adiacente lui care încă nu au fost prelucrate;
3. Se repetă pasul 2 până când coada devine vidă.

Algoritmul (pseudocod) este următorul:

```
enum { NEVIZITAT, VIZITAT };
void bfs( int nrNoduri, int nodSursa )
{
        int vizitate[ nrNoduri ]; /* marcarea nodurilor vizitate */
        Coada Q; /* coada nodurilor - intregi */
        int i, v, w; /* noduri */

        initializeaza( Q );  /* la inceput, coada trebuie sa fie vida */
        for ( i = 0; i < nrNoduri; i++ ) /* marcam toate nodurile ca nevizitate */
                vizitate[ i ] = NEVIZITAT;
        vizitate[ nodSursa ] = VIZITAT; /* marcam nodul sursa ca vizitat */
        procesam informatia pt nodSursa;
        enqueue( nodSursa, Q );
        /* nodSursa va fi primul nod scos din coada */
        while( ! goala( Q ))
        {
                v = dequeue( Q );
                for ( fiecare nod w adiacent cu v )
                        if ( vizitate[ w ] == NEVIZITAT )
                        {
                                vizitate[ w ] = VIZITAT;
                                procesam informatia pt w;
                                enqueue( w, Q );
                        }
        }
}
```

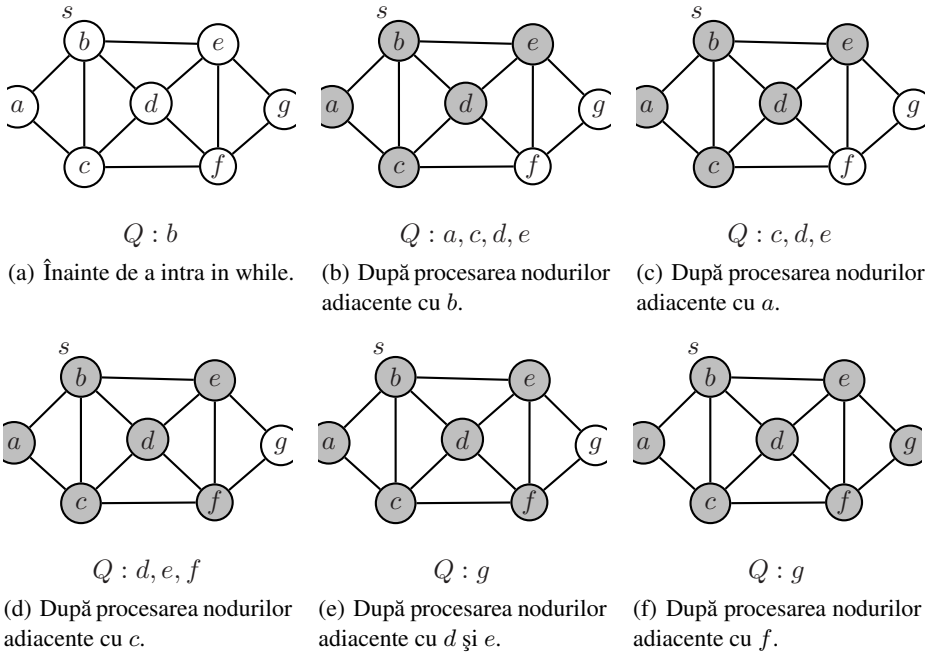
O trasare a pașilor relevanți pentru căutarea în lățime este prezentată în figura 7.4.

Algoritmul BFS se aplică atât pentru grafuri orientate cât și pentru cele neorientate. Pentru grafuri neorientate, BFS vizitează toate nodurile din aceeași componentă conexă cu nodul de start. Pentru grafuri orientate, nodurile vizitate pornind de la nodul de start sunt cele la care se poate ajunge din nodul de start, dar ele nu formează neapărat o componentă tare conexă.

Pentru a vizita toate nodurile grafului, se aplică algoritmul BFS repetat pentru noduri sursă care nu au fost încă vizitate, până când nu mai rămân noduri nevizitate. Arcele/muchiile parcurse la vizitarea nodurilor unei componente conexe formează un *arbore de acoperire* (*spanning tree*) pentru acea componentă. Arborii de acoperire pentru toate componentele conexe formează o *pădure de acoperire* (*spanning forest*).

Pentru un graf G cu n vârfuri și m muchii, traversarea BFS durează $O(n+m)$. Ca urmare, există algoritmi $O(n+m)$ bazați pe BFS pentru următoarele probleme:

- a verifica dacă G este conex
- a determina o pădure de acoperire pentru G
- a determina componentele conexe ale lui G (pentru un graf neorientat)



$Q : b$

(a) Înainte de a intra in while.

$Q : a, c, d, e$

(b) După procesarea nodurilor adiacente cu b.

$Q : c, d, e$

(c) După procesarea nodurilor adiacente cu a.

$Q : d, e, f$

(d) După procesarea nodurilor adiacente cu c.

$Q : g$

(e) După procesarea nodurilor adiacente cu d şi e.

$Q : g$

(f) După procesarea nodurilor adiacente cu f.

Figure 7.4: O trasare a căutarii în lățime pe un graf.

- pentru un vârf de start s al lui G, a determina pentru fiecare vârf v din G a căii celei mai scurte dintre s și v sau a preciza că nu există cale între ele
- a determina un ciclu în G sau a preciza că nu există cicluri

7.3 Mersul lucrării

7.3.1 Probleme obligatorii

1. Folosind reprezentarea prin *matrice de adiacență*, implementați algoritmul de explorare în lățime a unui graf citit dintr-un fișier (se dă numărul de noduri și perechile de vârfuri care formează arce/muchii). Nodul de start se citește de la tastatură. Scrieți în alt fișier nodurile în ordinea parcurgerii lor. Testați programul considerând că în fișier e reprezentat un graf neorientat, respectiv orientat.
2. Aceeași problemă, dar folosind reprezentarea cu *liste de adiacență dinamice*.
3. Aceeași problemă, dar folosind reprezentarea cu *liste de adiacență statice*.

7.3.2 Probleme opționale

1. Pentru un graf citit dintr-un fișier și două noduri citite de la tastatură, să se verifice și afișeze dacă cele două noduri sunt conectate.
2. Să se implementeze o aplicație care, citind două noduri de la tastatură, să afișeze distanța minimă dintre ele.
3. * Să se implementeze o aplicație care să detecteze dacă un graf este bipartit. În cazul în care nu este bipartit, să se determine un ciclu de lungime impară în graf.