Laborator 9 – Algoritmi elementari pe grafuri

Marti 16-18

Un graf este o structura de date foarte utila, ce poate fi cel mai usor vizualizata daca ne gandim la niste orase conectate intre ele prin drumuri. Orasele sunt noduri in graf iar drumurile reprezinta muchii. Daca toate drumurile sunt bidirectionale atunci graful se numeste neorientat. Daca exista si drumuri cu sens unic, atunci trebuie specificat pentru toate drumurile si sensul lor, iar daca avem un drum bidirectional intre 2 noduri A si B trebuie specificat ca avem drum si de la A la B,si de la B la A. Acest tip de graf se numeste graf orientat.
Grafurile se pot implementa in mai multe moduri:
1. Matrice de adiacenta. Pentru un graf de N noduri, tinem o matrice NxN, in care A[i][j] = 1 daca exista drum de la i la j. Pentru un graf neorientat, A[i][j] = 1 <=> A[j][i] = 1.
2. Liste de adiacenta. Solutia precedenta este consumatoare de spatiu si timp, cand cautam vecinii unui nod trebuie sa iteram prin toate nodurile, chit ca el poate avea doar un singur vecin, asa ca ideea intuitiva este sa tinem pentru fiecare nod o lista cu vecini.
3. Vectori de adiacenta. Metoda preferata in limbaje precum C++ care au biblioteci cu vectori care se modifica dinamic este sa folosim vectori in loc de liste. Se poate implementa si in C – avem vectori alocati dinamic de dimensiune mica, de exemplu 10, pentru fiecare nod tinem un vector de dimensiune 10 pentru vecini. Daca se depaseste acest numar, se dubleaza lungimea vectorului. In acest fel, memoria folosita va fi cel mult de 2 ori mai mare decat cea optima, iar timpul de acces la noduri va fi mai bun.
Pentru laborator se cere implementarea cu liste de adiacenta.

```
typedef struct graph {
	int nr_nodes;
	struct node** neighbours;
} graph;
typedef struct node {
	int val;
	struct node* next;
} node;
```

Pentru un graf, ne dorim sa exploram nodurile astfel incat sa accesam toate nodurile grafului.
Avem 2 variante posibile:
1. Parcurgere in latime – BFS
E utila pentru a afla distanta de la un nod la toate celelalte. Se introduce nodul sursa intr-o coada, apoi cat timp coada are elemente se scoate primul nod din coada, se parcurg vecinii si daca n-au fost vizitati se adauga in coada. Cand sunt adaugati in coada, li se modifica si distanta fata de nodul sursa. Datorita faptului ca nodurile sunt parcurse pe niveluri, distanta va fi minima.

Pseudocod:

```
Q.push(source);
dist [source]  = 0;
while (!Q.empty())
{
	x = Q.pop();
	visited[x] = true;
	for (y in neighbours(x))
		if (!visited[y])
		{
			Q.push(y);
			dist [ y] = dist[x]+1;
}
```

Daca in urma rularii algoritmului toate nodurile sunt vizitate inseamna ca graful este conex – putem ajunge din orice nod in orice nod. Toate nodurile care sunt vizitate din sursa formeaza o componenta conexa. Pentru a afla toate componentele conexe, putem rula algoritmul BFS cu o noua sursa care nu a fost vizitata.

2. Parcurgere in adancime – DFS
Parcurgerea DFS este utila pentru alte aplicatii, cum ar fi sortare topologica sau componente tare conexe. Se poate face cu algoritmul de mai sus, folosind o stiva in loc de coada, sau se poate simula stiva recursiv. In timpul parcurgerii DFS, exista 2 tipuri de muchii: muchii de arbore, cele pe care se merge in parcurgere, si muchii de intoarcere, cele care nu se folosesc, de la un nod la un alt nod care a fost deja vizitat. Aceasta impartire e utila pentru alte aplicatii, cum ar fi componente tare conexe sau biconexe. Pentru a vizita toate nodurile, apelam algoritmul dfs de mai multe ori daca e cazul, cu alt nod initial, care nu a fost vizitat dupa prima parcurgere.

```
dfs(nod):
        visited[nod] = true
        discover_time[nod] = time++;
        for (y in neighbours(nod))
                if (visited[y] == false)
                        dfs(y)
        finish_time[nod] = time++;
        S.push(nod)

apel_dfs():
        visited[] = false
        for x in nodes:
                if (visited[x] == false)
                        dfs(x)

tare_conex():
        apel_dfs();
        while (!S.empty())
        {
                int x = S.pop();
                if (visited2[x] == false)
                {
                        dfs2(x); // dfs pe graful transpus
                        ctc++;
                }
        }
```

In algoritmul de mai sus, am adaugat si timpul de descoperire, respectiv cel de finalizare. Timpul de finalizare este util pentru aplicatii- sortare topologica si componente tare conexe.
O sortare topologica reprezinta o sortare a nodurilor astfel incat daca avem muchie $a \rightarrow b$, a trebuie sa apara inaintea lui b in sortare. Sortarea topologica se poate face pentru grafuri orientate care nu au cicluri (pentru ca altfel am avea o dependenta circulara care nu poate fi satisfacuta).

Pentru sortarea topologica, e suficient sa sortam nodurile descrescator in functie de timpul de finalizare din dfs. Acest lucru se poate realiza si cu o stiva – in loc sa actualizam timpul de finalizare, punem nodul pe o stiva, iar la final le vom scoate in ordine descrescatoare a timpului de finalizare.

O componenta tare conexa e similara cu o componenta conexa (putem ajunge din orice nod in orice nod), dar cu mentiunea ca este pentru un graf orientat. Pentru componente tare conexe, avem 2 algoritmi folositi – Kosaraju si Tarjan, ambele pe baza DFS.

Algoritmul Kosaraju foloseste 2 parcurgeri DFS. Se face o parcurgere DFS a nodurilor, pe graful initial, apoi se iau nodurile in functie de timpul de finalizare (sau se scot de pe stiva). Pentru un nod scos de pe stiva, se face o noua parcurgere DFS, dar pe graful transpus (Daca aveam inainte muchie $a \rightarrow b$ acum vom avea muchie $b \rightarrow a$). Toate nodurile vizitate vor face parte din componenta tare conexa a nodului respectiv.

Algoritmul lui Tarjan este util pentru componente tare conexe dar si pentru componente biconexe, muchii critice si puncte de articulatie. Un graf neorientat e biconex daca oricum am elimina un nod, ramane conex. Un nod e punct de articulatie daca eliminand nodul, graful nu mai ramane conex. O muchie se numeste critica daca eliminand muchia graful nu mai este conex. Algoritmul lui Tarjan e folosit pentru determinarea tuturor acestor elemente si este o parcurgere DFS in care se mai adauga o informatie – pe langa timpul de descoperire, pe care il vom nota cu index[nod], se tine inca un vector, low[nod], reprezentand nivelul cel mai de sus din arbore unde se poate ajunge folosind muchiile de arbore din nod (copiii nodului in parcurgerea DFS). In functie de relatia dintre low[nod] si index[nod], se poate stabili daca un nod e critic sau daca o muchie e critica, iar daca un nod e critic atunci formeaza o noua componenta tare conexa sau biconexa, in functie de caz.

Pseudocod Tarjan pentru componente tare conexe (se foloseste o stiva in care se adauga nodurile intalnite pe parcurs, daca vecinul a fost vizitat actualizarea se face doar daca este in stiva):

```
void tarjan_ctc(int n)
{
   idx[n] = lowlink[n] = indecs;
   indecs ++;
   S.push(n), in_stack[n] = 1;

   for (v in vec(n)) {
      if (idx[v] == -1)
         tarjan_ctc(v),
         lowlink[n] = Min(lowlink[n], lowlink[v]);
      else if (in_stack[v])
         lowlink[n] = Min(lowlink[n], idx[v]);
   }
   if (idx[n] == lowlink[n])
         //new CTC, pop from stack until we have found node n
}
```

Pseudocod Tarjan pentru componente biconexe (se foloseste o stiva in care se adauga muchii, de data aceasta, actualizarea facandu-se pentru toate nodurile, mai putin parintele):

```
void tarjan_bc(int n, int parent)
{
   idx[n] = lowlink[n] = indecs;
   indecs ++;
   for (v in vec(n)) {
      if (v == parent)   continue ;
      if (idx[v] == -1) {
         S.push(n,v);
         tarjan_bc(v, n);
         lowlink[n] = Min(lowlink[n], lowlink[v]);
         if (lowlink[v] >= dfn[n])
            //new biconex component, pop from stack until we have found edge n->v
      }
      else
         lowlink[n] = Min(lowlink[n], idx[v]);
   }
```

Cerinte:
0. Scrieti functiile de citire si construire a unui graf cu N noduri si M muchii, citite din fisierul “graf.in”. Afisati pentru fiecare nod lista cu vecini si eliberati memoria la final.
1. Scrieti algoritmul BFS pornind din nodul 1. Afisati la final distanta de la fiecare nod la nodul 1.
2. Scrieti algoritmul DFS pentru un graf orientat, afisand la final nodurile in functie de timpul de finalizare (o sortare topologica).
3. Pentru un graf dat, sa se afiseze componentele tare conexe folosind algoritmul lui Kosaraju cu 2 parcurgeri DFS sau algoritmul lui Tarjan (la alegere).
Bonus.: Afisati componentele biconexe folosind algoritmul lui Tarjan.

Resurse utile:
https://infoarena.ro/problema/biconex
https://infoarena.ro/problema/ctc
https://ocw.cs.pub.ro/courses/pa/laboratoare/laborator-07
https://ocw.cs.pub.ro/courses/pa/laboratoare/laborator-08

A[i][j] = 1 <=> ai muchie de la i la j
A[j][i] = 1 <=> am muchie de la j la i