Laborator 10. Grafuri neorientate - reprezentări, parcurgeri

Un graf este o structură de date formată din noduri și muchii – legături intre noduri. Grafurile sunt folosite pentru a modela relațiile dintre perechi de obiecte, de exemplu drumurile dintr-o țară între diferite orașe (nodurile reprezintă orașe, iar muchiile reprezintă drumuri). Grafurile pot avea costuri asociate muchiilor sau nu, în funcție de ceea ce se dorește să se reprezinte.

Grafurile pot fi neorientate (muchie intre x și y înseamnă și muchie între y și x) sau orientate (precum în cazul drumurilor sun sens unic într-un oraș).

Grafurile pot fi implementate în 2 moduri:

1. Matrice de adiacență – dacă graful are n noduri, se formeaza o matrice nxn, în care m[i][j] = 1 dacă există muchie de la nodul i la nodul j și 0, altfel. Pentru parcurgerea vecinilor unui nod trebuie parcursă toată linia din matrice (complexitate ridicată).

Graful va fi reprezentat în felul următor:

```
typedef struct
{
     int nn;
     int **Ma;
}    TGraphM;
```

2. Liste de adiacență – fiecare nod are o listă cu vecini (nodurile către care exista muchii). Aceasta este metoda optimă de reprezentare.

Graful va fi reprezentat în felul următor:

```
typedef struct node
{
     int v;
     TCost c;
     struct node *next;
} TNode, *ATNode;

typedef struct
{
     int nn;
     ATNode *adl;
} TgraphL;
```

Există două parcurgeri uzuale ale unui graf:

1. Parcurgerea în adâncime – DFS (Depth-First Search). Se pornește de la un nod, de exemplu nodul 1. Se vizitează toți vecinii acestuia, și, în momentul în care am găsit un vecin, continuăm parcurgerea de la acesta. Se poate implementa recursiv (când găsim un vecin care nu a fost marcat apelam recursiv DFS din vecin) sau nerecursiv, folosind o stiva.

2. Parcurgerea în lățime – BFS (Breadth-First Search). Se pornește de la un nod de start, care este adaugat într-o coadă. Se vizitează toți vecinii nodului, apoi nodul este șters din coadă. Vecinii sunt adăugați în coadă dacă nu au fost vizitati deja, apoi se reia parcurgerea cât timp încă există noduri în coadă.

Aplicații laborator

Se va rula comanda ***make test*** pentru o verificare completă.

În cadrul laboratorului, vom lucra cu un **graf neorientat** (muchiile sunt bidirecționale, când se adauga muchia x->y trebuie adaugată și muchia y->x, iar implementarea grafului este cu liste de adiacență (vezi structurile de mai sus).
Nodurile sunt reprezentate prin numere **indexate de la 0 la n-1**.

Aveți de implementat următoarele funcții, în fișierul *Graph.c*:

1. Crearea grafului, cu liste de adiacență – 1.5p

`TGraphL* createGraphAdjList(int numberOfNodes)`

- Alocă structura de tip TGraphL, precum și vectorul care va ține capetele listelor de adiacență (setate la NULL, inițial).

`void addEdgeList(TGraphL* graph, int v1, int v2)`

- Adaugă nodul v1 **la începutul** listei de adiacență a lui v2 si viceversa (atenție, noile celule trebuie alocate).
  - Adăugăm la începutul listei din motive de eficiență: cum nu reținem finalul listei nicăieri, aceasta ar trebui parcursă de fiecare dată când dorim să adaugăm încă un nod.
- Costul poate fi setat la 1 pentru toate muchiile, dar nu este utilizat în acest laborator.

2. DFS – varianta iterativă – 2.5p

`List* dfsIterative(TGraphL* graph, int s)`

- Implementează parcurgerea DFS iterativ, cu stivă, plecând de la nodul sursă s. Vă puteți folosi de implementările existente ale funcțiilor corespunzătoare stivei, cu antetele în *util.h*.
  - Nu uitați să distrugeți stiva la final, precum și să dezalocați vectorul *visited* pe care il veți folosi pentru a marca nodurile descoperite.
- Returnează o referință de tipul List* (disponibilă tot prin intermediul *util.h*) ce conține nodurile accesate conform parcurgerii la începutul explorării.
  - Puteți folosi `path = createList()` pentru a crea lista și `enqueue(path, s)`, pentru a adăuga nodul s.

3. DFS – varianta recursivă – 2.5p

`List* dfsRecursive(TGraphL* graph, int s)`

- Implementează parcurgerea DFS recursiv, plecând de la nodul sursă s.
- Returnează o listă cu aceeași semnificație precum mai sus.
- Se folosește de funcția ajutătoare

void `dfsRecHelper(TGraphL* graph, int* visited, List* path, int s)`, unde
    - visited este vectorul alocat dinamic care ne spune dacă un nod a fost descoperit sau nu.
    - path – lista ce reține ordinea parcurgerii.

4. BFS – varianta iterativă – 2.5p

List* bfs(TGraphL* graph, int s)

- Implementează parcurgerea BFS, cu o coadă, plecând de la nodul sursă s. Vă puteți folosi de implementările existente ale funcțiilor corespunzătoare cozii, cu antetele în *util.h*.
    - Nu uitați sa distrugeți coada la final.
- Și aici vom avea nevoie sa alocăm un vector *visited*, pe care îl eliberăm la final.
- Returnează o listă ce conține nodurile accesate conform parcurgerii, la începutul explorării (cu aceeași semnificație ca mai sus).

5. Eliberarea memoriei – 1p

void destroyGraphAdjList(TGraphL* graph)

- Eliberează întreaga memorie alocată grafului.
- Va fi testată cu Valgrind.