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. De asemenea, grafurile pot fi neorientate (muchie între 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** – dându-se un graf cu *N* noduri, se formează o matrice de dimensiune *N*x*N*, 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 aferentă acelui nod (numărul de operații necesare în acest caz fiind ridicat).

Graful va fi reprezentat în felul următor:

```
typedef struct
{
     int nn; // numărul de noduri
     int **Ma; // matricea de adiacență
}    TGraphM;
```

2. **Liste de adiacență** – dându-se un graf cu *N* noduri, se formează un vector de dimensiune *N* unde intrarea de pe poziția *x* constituie o listă simplu înlănțuită reprezentând celelalte noduri din graf cu care nodul *x* are muchie directă.
Pentru parcurgerea vecinilor unui nod, această metodă este mai eficientă decât cea cu matrice de adiacență pe cazul general deoarece se pot accesa direct vecinii, fără să se mai verifice toată mulțimea de noduri.

Graful va fi reprezentat în felul următor:

```
typedef struct node
{
     int v; // nodul vecin
     struct node *next; // pointer la următorul nod din listă
} TNode, *ATNode;

typedef struct
{
     int nn; // numărul de noduri
     ATNode *adl; // vectorul de liste de adicență
} 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 nevizitat, continuăm parcurgerea de la acesta. Se poate implementa recursiv (când găsim un vecin care nu a fost marcat apelam recursiv DFS din vecin) sau iterativ, folosind o stiva.

Pseudocod DFS recursiv:

```
funcție DFS_Recursiv(Graf, nod):
        marchez nod ca vizitat
        pentru fiecare vecin v al lui nod din Graf:
                dacă v nu este vizitat:
                        apelez recursiv DFS_Recursiv(Graf, v)
```

2. **Parcurgerea în lățime** – **BFS (Breadth-First Search)**. Se pornește de la un nod de start, de exemplu nodul 1. El este adăugat într-o coadă. Se vizitează toți vecinii nodului, apoi nodul este șters din coadă. Vecinii sunt adăugați în coadă dacă nu au fost vizitați deja, apoi se reia parcurgerea cât timp încă există noduri în coadă.

Pseudocod BFS:

```
funcție BFS(Graf, nod):
        fie Q o coadă
        marchez nod ca vizitat
        Q.enqueue(nod)
        cât timp Q nu e goală:
                v = Q.dequeue()
                pentru fiecare vecin w al lui v din Graf:
                        dacă w nu este vizitat:
                                marchez w ca vizitat
                                Q.enqueue(w)
```

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.

2. DFS – varianta recursivă – 4p

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

- Implementează parcurgerea DFS recursiv, plecând de la nodul sursă s.
- Returnează o referință de tipul List* (disponibilă tot prin intermediul *util.h*) ce conține nodurile accesate conform parcurgerii de la începutul explorării.
  - Puteți folosi `path = createList()` pentru a crea lista și `enqueue(path, s)`, pentru a adăuga nodul s.
- Se folosește 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.

3. BFS – varianta iterativă – 4p

`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, de la începutul explorării (cu aceeași semnificație ca mai sus).

4. Eliberarea memoriei – 0.5p

`void destroyGraphAdjList(TGraphL* graph)`

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