TP15 : Graphes en C

31 mai 2022

L'objet de ce TP est d'implémenter les algorithmes vus en classe sur les graphes.

1 RERÉSENTATION :

Nous utiliserons ici le modèle suivant pour représenter un graphe par ses listes d'adjacence.

Les sommets seront des entiers indexés de manière consécutive. Ainsi un graphe à n sommets aura ses sommets qui seront $\{0, ..., n-1\}$.

```
#define MAXV 100 /* nombre maximum de sommets */

struct edgenode {
    int y; // le premier voisin
    struct edgenode* next; // la suite de la liste
};
typedef struct edgenode edgenode;

struct graph {
    edgenode* edges[MAXV]; // tableau de listes d'adjacence
    int degree[MAXV]; // le degré de chaque sommet
    int nedges;
    int nbvertices;
    bool directed; // indique si le graphe est orienté
};

typedef struct graph graph;
```

▷ **Question 1.** Ecrire une fonction `void initialize_graph(graph* g, bool directed)` qui initialise le graphe g passé par pointeur comme étant le graphe vide, orienté ou non selon la valeur de `directed`. ◁

▷ **Question 2.** Ecrire une fonction `void insert_edge(graph* g, int x, int y, bool directed)` qui insère une arête de `x` à `y` en considérant `g` comme orienté ou non selon `directed` et donc en ignorant `g->directed`. La fonction se contentera de mettre à jour les listes d'adjacence mais on ne modifiera pas les champs `nbvertices` ni `nedeges` considérés comme déjà à jour.

◁

Pour travailler sur des graphes, on va ecrire une fonction permettant de lire un fichier contenant le graphe sous le format suivant :

Une première ligne contenant trois entiers, le nombre de sommets `n`, le nombre d'arêtes `p` et 0 ou 1 selon que le graphe soit non orienté ou orienté. Ensuite `p` lignes contenant deux entiers `i` et `j` et indiquant qu'il y a une arête de `i` vers `j`.

▷ **Question 3.** Ecrire une fonction `void read_graph(FILE* f, graph* g)` qui lit un graphe sur le flux `f` et le place dans `g` après l'avoir initialisé. ◁

▷ **Question 4.** Ecrire une fonction `void free_edges(graph* g)` qui libère les listes d'adjacence. ◁

2 Parcours :

On va modifier la structure de graphe et rajouter trois nouveaux champs :

```
bool discovered[MAXV]; // Quels sommets sont connus
bool processed[MAXV]; // Quels sommets sont traités
int parent[MAXV]; // parent[x] est le père de x dans le parcours
                  // s'il n'y en a pas, c'est -1
```

▷ **Question 5.** Ecrire une fonction `void initialize_search(graph *g)` qui initialise ces tableaux. ◁

▷ **Question 6.** Implémenter un parcours en profondeur récursif qui affiche les sommets du graphe dans l'ordre parcouru à partir d'un sommet de départ `s`. ◁

▷ **Question 7.** Adapter l'algorithme précédent pour détecter un cycle dans un graphe (orienté ou non). ◁

▷ **Question 8.** Implémenter un parcours en largeur qui affiche les sommets du graphe dans l'ordre parcouru. On pourra utiliser la librairie file fournie sur le dossier public. ◁

▷ **Question 9.** Adapter la fonction précédente pour remplir un tableau contenant les distances de chaque sommet du graphe à l'origine du parcours. ◁

3 Grands entiers :

Pour représenter de grands entiers, on exploite l'observation suivante : tout entier naturel s'écrit de manière unique

1. soit comme l'entier 0 ;
2. soit comme l'entier 1 ;
3. soit comme $h \times 2^{2^p} + l$ avec $0 < h < 2^{2^p}$ et $0 \leq l < 2^{2^p}$.

Dans ce dernier cas, on note `<h,p,l>` ce triplet. Ainsi, l'entier 42 s'écrit `<2,2,10>` car $42 = 2 \times 2^{2^2} + 10 = 2 \times 16 + 10$. De même, l'entier 10 s'écrit `<2,1,2>` car $10 = 2 \times 2^{2^1} + 2 = 2 \times 4 + 2$ et l'entier 2 s'écrit `<1,0,0>` car $2 = 1 \times 2^{2^0} + 0 = 1 \times 2 + 0$.

A cette décomposition, on rajoute l'idée qu'un même triplet peut être construit de manière unique en mémoire. On a alors une représentation sous forme d'un graphe où les sommets sont des triplets `<h,p,l>`, avec h, p et l étant 0, 1 ou une référence à un autre sommet. Une telle représentation des entiers est baptisée IDD (pour Integer Dichotomy Diagrams). La figure suivante illustre l'IDD représentant l'entier 42.

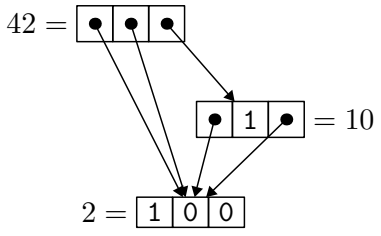

Figure 1 – Représentation de l'entier 42 par un IDD.

▷ **Question 10.** Dessiner l'IDD correspondant à l'entier 773. ◁

Nous utiliserons le type suivant pour représenter un IDD :

```
struct grand {
    bool est_zero;
    bool est_one;
    struct grand* p1;
    struct grand* p2;
    struct grand* p3;
    bool vu;};

typedef struct grand idd;
```

Le champs `vu` ne servira que pour la gestion d'un parcours du graphe représentant un IDD.

Si on reprend l'exemple de 42, son IDD peut être initialisé par la commande suivante :

```
idd* zero = (idd*)malloc(sizeof(idd));
idd* one = (idd*)malloc(sizeof(idd));
zero->p1=NULL;
zero->p2=NULL;
zero->p3=NULL;
one->p1=NULL;
one->p2=NULL;
one->p3=NULL;
zero->vu=false;
one->vu=false;
zero->est_one=false;
zero->est_zero=true;
one->est_one=true;
one->est_zero=false;

idd* deux = (idd*)malloc(sizeof(idd));
deux->est_one=false;
deux->est_zero=false;
deux->vu=false;
deux->p1 = one;
deux->p2=zero;
deux->p3=zero;

idd* dix = (idd*)malloc(sizeof(idd));
dix->est_one=false;
dix->est_zero=false;
dix->vu=false;
dix->p1 = deux;
dix->p2=one;
dix->p3=deux;


idd* quarante_deux = (idd*)malloc(sizeof(idd));
quarante_deux->est_one=false;
quarante_deux->est_zero=false;
quarante_deux->vu=false;
quarante_deux->p1=deux;
quarante_deux->p2=deux;
quarante_deux->p3=dix;
```

▷ **Question 11.** Initialiser l'entier 773. ◁

▷ **Question 12.** On définit la taille d'un entier n, notée $s(n)$, comme le nombre de sommets distincts dans l'IDD qui le représente, les entiers 0 et 1 n'étant pas comptés comme des sommets. Ainsi, $s(42) = 3$ au regard de la figure 1. En particulier, $s(0) = s(1) = 0$.

Ecrire une fonction `int size(idd* i)` qui prend en entrée un IDD et renvoie sa taille. ◁

▷ **Question 13.** Ecrire une fonction `idd* of_int(int n)` qui convertit un entier en IDD.
◁

▷ **Question 14.** Ecrire une fonction `int to_int(idd* i, idd* memo[10000])` qui convertit un IDD en un entier. Il conviendra de garantir que chaque sommet utilse n'est créé qu'une seule fois. Pour cela, on pourra utiliser un tableau qui stocke les adresses des sommets déjà créés. Le tableau sera déclaré dans le `main` et passé en argument de la fonction. Idéalement ce tableau devrait être remplacé par une table de hachage pour éviter une utilisation mémoire abusive. ◁

Pour écrire un IDD $n \geq 2$ dans un fichier, on se propose d'utiliser le format texte suivant. Le fichier contient exactement $s(n)$ lignes et chaque ligne est de la forme `i j k l` où i, j, k et l sont quatre entiers, avec $2 \leq i \leq s(n)+1$ et $0 \leq j, k, l < i$. L'entier i numérote la ligne, à partir de 2. Cette ligne définit un nouvel IDD, numéroté i, comme valant `<j, k, l>` où j, k et l sont les trois IDD respectivement num'erotés j, k et l. Les numéros 0 et 1 font référence aux IDD 0 et 1. Le fichier représente l'IDD défini par la dernière ligne. Ainsi, l'IDD représentant l'entier 42 peut être sérialisé par les trois lignes suivantes :

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

Ces trois lignes correspondent aux trois IDD de la figure 1.

▷ **Question 15.** Réaliser cette sérialisation, c'est-à-dire écire un programme permettant d'imprimer successivement les lignes du fichier qui représente un IDD n donné. ◁

▷ **Question 16.** Ecrire une fonction de désérialisation. ◁

4 FLOYD WARSHALL

▷ **Question 17.** Dans cette partie, on représentera un graphe orienté pondéré par sa matrice d'adjacence pondérée. Un graphe sera donc représenté par un tableau à deux dimensions statique (on fixera le nombre de lignes et de colonnes). Ecrire l'algorithme de Floyd Warshall qui permet de calculer la plus courte distance entre tous couples de chemins dans un graphe sans cycle de poids strictement négatif. ◁