Autour des parcours

21 et 28 septembre

Considérons un graphe $G = (V, E)$ orienté connexe. On peut pondérer les arêtes du graphe avec une fonction $p : E \to \mathbb{R}$, dite de pondération.

1 Construction d'un graphe :

On considère maintenant un graphe orienté G connexe et pondéré à poids positifs.
Nous allons travailler avec le type de graphe suivant :

```
struct sommetpond{int val; int poids;};
struct arete{int u; int v; int p;};

typedef struct arete arete;
typedef struct sommetpond sommetpond;


struct edgenode{
    sommetpond y;
    struct edgenode* next;
};

typedef struct edgenode edgenode;

struct graph {
    edgenode* edges[MAXV];//tableau de listes d'adjacence
    int degree[MAXV];
    int nedges;
    int nbvertices;
    bool discovered[MAXV];
};

typedef struct graph graph;
```

Cette partie va permettre de construire efficacement des graphes à partir de fichiers textes. Elle doit être préparée en amont. Il conviendra de bien tester que vos fonctions fonctionnent.

▷ **Question 1.** Ecrire une fonction `void initialize_graph(graph* g)` qui initialise le graphe `g` passé par pointeur comme étant le graphe vide non orienté (on ne connait pas le nombre de sommets). ◁

▷ **Question 2.** Ecrire une fonction `void insert_edge(graph* g, int x, int y,int p)` qui insère une arête de $\{x, y, p\}$ orientée. ◁

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

Une première ligne contenant deux entiers, le nombre de sommets `n` et le nombre d'arêtes `k`. Ensuite `k` lignes contenant trois entiers `i`, `j` et `p` indiquant qu'il y a une arête de `i` vers `j` de poids `p`.

▷ **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. ◁

▷ **Question 5.** Ecrire une fonction `void initialize_search(graph* g)` qui initialise les tableaux utilisés pour les fonctions de parcours. ◁

▷ **Question 6.** Ecrire une fonction `void pp_affiche(graph* g)` qui effectue un parcours en profondeur du graphe et affiche les sommets du graphe dans l'ordre du parcours. ◁

A ce stade, vous devez être en mesure de bien tester toutes vos fonctions : dessiner un graphe, le représenter dans un fichier, le construire en lisant ce fichier et faire un parcours en profondeur pour vérifier que l'affichage correspond.

2 Dijkstra

Nous avons vu l'algorithme de Dijkstra qui permet de calculer les distances à un sommet dans un graphe pondéré dont les poids sont positifs ou nuls. Il existe plusieurs manières de formuler, de prouver et d'implémenter cet algorithme.

La version la plus simple à écrire consiste à adapter le parcours en largeur.

▷ **Question 7.** Ecrire une fonction `void parcours_largeur(graph* g, int s)` qui parcourt `g` en largeur à partir du sommet `s` et affiche ses sommets. On pourra utiliser le fichier `file.h` fourni. ◁

Pour adapter ce parcours, il suffit de remplacer la file par une file de priorité en utilisant comme valeur de priorité pour un sommet `x` la distance actuellement estimée par l'algorithme entre `x` et l'origine du parcours `s`.

▷ **Question 8.** En utilisant le fichier `file_prio.h`, écrire une fonction `int* dijkstra(graph* g, int s)` qui renvoie un tableau indexé par les sommets de `g` et contenant la distance à `s` pour chacun d'entre eux. ◁

3 2-SAT en OCaml

L'objectif de cette partie est d'implémenter le test de satisfiabilité d'une formule du calcul propositionnel sous forme 2SAT.

1. Rappeler la définition d'une formule sous forme 2SAT.

   Pour représenter une telle formule, on utilisera le type suivant :

```
type litteral =
| P of int (* positive occurrence *)
| N of int (* negative occurrence *)

type clause = litteral * litteral
type twocnf = clause list
```

   On supposera que les variables présentes dans une formule sont numérotées consécutivement à partir de zéro.

Soit $\phi = \bigwedge_{i=1}^{r} F_i$ une formule en 2-CNF (chaque F_i est donc de la forme $l \vee l'$, où l et l' ne contiennent pas la même variable) et $x_0, \ldots, x_{n-1}$ les variables présentes dans ϕ. On définit le graphe $G_\phi = (V_\phi, E_\phi)$ comme suit :

- il y a un sommet pour chaque littéral possible (autrement dit, $2n$ sommets $x_0, \neg x_0, \ldots, x_{n-1}, \neg x_{n-1}$) ;
- pour chaque clause $l \vee l'$, il y a deux arcs $\neg l \to l'$ et $\neg l' \to l$.

2. On considère la formule suivante :

$$\phi = (x_0 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (x_3 \vee \neg x_0)$$

   Représenter son graphe G_ϕ.

3. Ecrire une fonction `max_var : twocnf->int` qui renvoie l'indice maximal d'une variable apparaissant dans la formule passée en entrée.

4. Écrire une fonction `graph_of_cnf: twocnf->graph` prenant en entrée une formule ϕ en 2-CNF et renvoyant le graphe G_ϕ associé. On pourra supposer qu'aucune clause ne contient deux fois le même littéral, ni un littéral et son complémenté. Pour le type graphe, on utilisera `type graph = int list array`. Tester votre fonction sur le graphe `ex` fourni.

5. Rappeler la condition nécessaire et suffisante sur le graphe G_ϕ vue en cours qui caractérise le fait qu'une formule ϕ sous forme 2SAT est satisfiable.

6. Afin d'utiliser cette caractérisation, on va implémenter l'algorithme de Kosaraju.

   (a) Écrire une fonction `transpose` qui prend en entrée un graphe G et renvoie son graphe transposé (ou miroir).

   (b) Écrire une fonction `post_order` qui effectue un parcours en profondeur complet d'un graphe et renvoie la liste de ses sommets par instant de fin de traitement décroissant.

   (c) Compléter le code fourni pour obtenir une fonction `kosaraju : graph-> int list list` qui renvoie une liste contenant chaque composante fortement connexe du graphe passé en entrée. Chaque CFC est elle même une liste.

   (d) Tester votre fonction en vérifiant que le fichier `arxiv.txt` représente un graphe à 21608 composantes fortement connexes (une fonction de lecture du fichier vous est fournie).

7. Écrire une fonction `satisfiable` déterminant si une formule ϕ en 2-CNF est satisfiable. On exige une complexité linéaire en la taille de la formule (nombre d'occurrences de variables).