Devoir maison n°3

À rendre le lundi 14/10

⋆ ⋆ ⋆

Pour chacune des deux parties, vous utiliserez les fichiers texte disponible sur le dépôt `https://github.com/nathaniel-carre/MPI-LLG`, dans le dossier `Devoirs/DM/DM3`. Chacun de ces fichiers contient la description d'un graphe non orienté pondéré. Le format d'un fichier est le suivant :

- la première ligne contient un entier n, correspondant au nombre de sommets du graphe ;
- les lignes suivantes correspondent chacune à une arête. Une ligne contient trois entiers s, t et p séparés par des espaces. L'arête correspondante est alors l'arête $\{s, t\}$, de poids p. On impose que $s < t$ pour garantir que chaque arête n'apparaît qu'une seule fois.

Par exemple, les 5 premières lignes du fichier `g_10_ad.txt` sont :

```
10
0 2 2
0 3 11
1 7 13
1 3 14
```

ce qui signifie que le graphe est d'ordre 10, possède une arête entre 0 et 2, de poids 2, une arête entre 0 et 3, de poids 11, …

Le nom de chaque fichier contient un nombre, correspondant au nombre de sommets du graphe, ainsi qu'un suffixe `sd` (sans doublons) ou `ad` (avec doublons) qui indique si les poids des arêtes peuvent apparaître en plusieurs exemplaires ou non.

Pour ce devoir, vous devrez rendre une copie papier, contenant la rédaction des questions théoriques ou des valeurs numériques à écrire, indiquées dans l'énoncé par un emoji ✍, ainsi qu'un fichier de code source, contenant les fonctions et les tests que vous aurez écrit. Les questions de programmation sont indiquées par un emoji 💻. Il est demandé que le fichier source puisse se compiler (ou s'interpréter) correctement. Tout code erroné encore présent dans le fichier devra être commenté. Vous nommerez vos fichiers au format `DM3_NOM_Prenom.ml` et `DM3_NOM_Prenom.c`, par exemple `DM3_CARRÉ_Nathaniel.ml`.

La copie papier devra contenir au moins une page complète blanche, à destination du correcteur, pour la rédaction des commentaires lors de la correction du code. Le code source devra inclure des commentaires de code expliquant le fonctionnement des différentes parties du code. Il devra également contenir des tests des différentes fonctions écrites.

1 Algorithme de Kruskal

Cette partie est à traiter en langage OCaml.

1.1 Un algorithme de tri

Question 1 💻 Écrire une fonction `partition (piv: 'a) (lst: 'a list) : 'a list * 'a list` qui renvoie un couple `(l1, l2)` tel que `l1` contient les éléments de `lst` inférieurs ou égaux à `piv` et `l2` contient les éléments de `lst` strictement supérieurs à `piv`.

Question 2 💻 En déduire une fonction `tri_rapide (lst: 'a list) : 'a list` qui trie une liste selon le principe du tri rapide.

1.2 Création des graphes

On cherche à écrire une fonction qui prend en argument une chaîne de caractère correspondant à un nom de fichier texte et renvoie le graphe associé. On représente en OCaml un graphe non orienté pondéré $G = (S, A, f)$, avec $S = [\![0, n-1]\!]$, comme un tableau de taille n contenant des listes d'adjacence de couples (sommet, poids).

```
type graphe = (int * int) list array
```

Ainsi, si `g` est un objet de type `graphe` représentant G, et $s \in S$, alors `g.(s)` est une liste contenant tous les couples `(t, p)` où $\{s, t\} \in A$ et $f(s, t) = p$.

On rappelle que les fonctions suivantes permettent d'effectuer de la lecture de fichier et manipulation de chaînes de caractères :

- `open_in : string -> in_channel` prend en argument un nom de fichier texte (contenant le chemin, par exemple `"~/MPI/DM3/g_10_sd.txt"`) et renvoie un canal de lecture de ce fichier ;
- `input_line : in_channel -> string` prend en argument un canal de lecture et renvoie une chaîne de caractère correspondant à la prochaine ligne lue. Cette fonction lève l'exception `End_of_file` si le fichier a été lu dans son intégralité ;
- `close_in : in_channel -> unit` ferme un canal de lecture ;
- `String.split_on_char : char -> string -> string list` prend en argument un caractère `c` et une chaîne `str` et renvoie la liste des chaînes de caractères correspondant aux parties de `str` entourées par le caractère `c`. Par exemple, `String.split_on_char ' ' "0 5 12"` renverra `["0"; "5"; "12"]` ;
- `int_of_string : string -> int` prend en argument une chaîne de caractère et renvoie l'entier correspondant (ou une erreur si la chaîne ne correspond pas à un entier).

Question 3 Écrire une fonction `construire_graphe (nom_fichier: string) : graphe` qui prend en argument un nom de fichier correspondant à un graphe et renvoie le graphe correspondant.

Question 4 Écrire une fonction `poids_graphe (g: graphe) : int` qui calcule et renvoie le poids d'un graphe, c'est-à-dire la somme des poids de ses arêtes.

Question 5 Vérifier que les graphes des fichiers `g_10_ad` et `g_10_sd` ont des poids respectifs de 168 et 343. Quels sont les poids des graphes `g_1000_ad` et `g_1000_sd` ?

1.3 Union-Find

On cherche à implémenter une structure Union-Find pour gérer une partition d'un ensemble $S = [\![0, n-1]\!]$. On représente une telle partition par un tableau `partition` de taille n tel que :

- si x est le représentant de sa classe d'équivalence X, c'est-à-dire la racine de son arbre, alors `partition.(x)` est égal à $-rg(X) - 1$, où $rg(X)$ est le **rang** de X, qui est un **majorant** de la hauteur de l'arbre correspondant à la classe de X ;
- sinon, `partition.(x)` est le parent de x dans l'arbre correspondant à la classe X.

Question 6 Écrire une fonction `creer_partition (n: int) : int array` qui crée une partition où chaque élément est son propre représentant. On initialisera chaque classe avec un rang nul.

Question 7 Écrire une fonction `trouver (partition: int array) (x: int) : int` qui détermine et renvoie le représentant de x dans la partition correspondant à `t`. La fonction `trouver` devra appliquer la compression des arbres, telle que vue en cours, pendant la recherche du représentant.

Pour unir les classes d'équivalence de deux éléments x et y, on détermine leurs représentants respectifs r_x et r_y et, s'il sont distincts, le représentant de l'arbre ayant le rang le plus élevé devient le parent de l'autre. Lors d'une union, le rang de la classe ainsi obtenue est alors modifié selon le principe suivant :

- si les rangs des deux classes étaient distincts, le rang de la nouvelle classe est égal au maximum des deux rangs ;
- si les rangs des deux classes étaient égaux à r, le rang de la nouvelle classe devient $r + 1$.

Question 8 Écrire une fonction `unir (partition: int array) (x: int) (y: int) : unit` qui effectue l'union de deux classes selon ce principe. La fonction ne devra pas modifier le tableau si les deux éléments sont déjà dans la même classe.

Question 9 Justifier que le rang d'une classe n'est pas nécessairement égal à la hauteur de la classe si on effectue plusieurs opérations `trouver` et `unir` depuis une partition créée par `creer_partition`.

Question 10 Montrer que si X est une classe d'équivalence obtenue par des unions successives de classes à partir d'une partition créée par `creer_partition`, alors $|X| \geqslant 2^{h(X)}$ (où $h(X)$ est la hauteur de la classe X).

1.4 Implémentation finale

Question 11 Écrire une fonction `aretes (g: graphe) : (int * int * int) list` qui prend en argument un graphe et renvoie la liste des triplets $(f(s,t), s, t)$ tels que $\{s,t\} \in A$ et $s < t$, triée par ordre croissant des $f(s,t)$.

Question 12 En déduire une fonction `kruskal (g: graphe) : graphe` qui prend en argument un graphe et renvoie un arbre couvrant minimal calculé avec l'algorithme de Kruskal.

Question 13 Vérifier que les arbres couvrants minimaux des graphes des fichiers `g_10_ad` et `g_10_sd` ont des poids respectifs de 66 et 117. Quels sont les poids des arbres couvrants minimaux des autres graphes ?

2 Algorithme de Borůvka

Cette partie est à traiter en langage C.

On suppose que tous les graphes manipulés ont des sommets dont les degrés sont inférieurs ou égaux à 10.

2.1 Création des graphes

On cherche à écrire une fonction qui prend en argument une chaîne de caractère correspondant à un nom de fichier texte et renvoie le graphe associé. On représente en C un graphe non orienté pondéré $G = (S, A, f)$, avec $S = [\![0, n-1]\!]$ par le type suivant :

```c
struct Graphe{
    int n;
    int** adj;
};

typedef struct Graphe graphe;
```

Si `G` est un objet de type `graphe` correspondant à G, alors `G.n` est égal à n, et si $s \in S$ est un sommet de degré $\deg(s) = k$, alors `G.adj[s]` est un tableau de taille 21 tel que si on note $(t_0, , t_{k-1})$ les voisins de s, alors :

- pour $i \in [\![0, k-1]\!]$, `G.adj[s][2 * i]` est égal à t_i et `G.adj[s][2 * i + 1]` est égal à $f(s, t_i)$;
- `G.adj[s][2 * k]` est une valeur sentinelle égale à -1 ;
- pour $i \in [\![2k+1, 20]\!]$, `G.adj[s][i]` a une valeur quelconque.

Question 14 Écrire une fonction `int degre(graphe G, int s)` qui détermine le degré d'un sommet s dans un graphe G.

On encode une arête par le type suivant :

```c
struct Arete{
    int s;
    int t;
    int p;
};

typedef struct Arete arete;
```

tel qu'une arête $a = \{s, t\}$, $s < t$, encodée par `a` de type `arete` vérifie `a.s` $= s$, `a.t` $= t$ et `a.p` $= f(s, t)$.

Question 15 💻 Écrire une fonction `void ajouter_arete(graphe G, arete a)` qui ajoute une arête a dans un graphe G. Cette fonction modifie directement le graphe, et n'effectue aucune modification si l'arête existe déjà, ou si l'une de ses extrémités est de degré égal à 10.

On rappelle que les fonctions suivantes permettent d'effectuer de la lecture de fichier et sont utilisables avec l'inclusion de l'entête `stdio.h`. On pourra se référer à une documentation pour plus de détails :

- `fopen` ouvre un fichier texte et renvoie un pointeur vers le canal de lecture, de type `FILE*` ; par exemple, l'appel `fopen("7MPI/DM3/g_10_sd.txt", "r")` permet d'ouvrir le fichier `g_10_sd` en mode lecture ;
- `fscanf` permet de lire la ligne suivante du fichier selon un certain formattage ; par exemple, si `canal` est un canal de lecture et si `a`, `b` et `c` sont des variables entières, alors `fscanf(canal, "%d %d %d", &a, &b, &c)` permet de lire une ligne contenant deux entiers et de stocker leurs valeurs dans les variables `a`, `b` et `c`. La fonction renvoie le nombre de valeurs affectées si la lecture s'est déroulée normalement (donc 3 dans l'exemple précédent), et renvoie `EOF` si le canal était à la fin du fichier ;
- `fclose` ferme un canal de lecture. Par exemple, `fclose(canal)` permet de fermer `canal`.

Question 16 💻 Écrire une fonction `graphe construire_graphe(const char* nom_fichier)` qui prend en argument un nom de fichier correspondant à un graphe et renvoie le graphe correspondant.

Question 17 💻 Écrire une fonction `int poids_graphe(graphe G)` qui calcule et renvoie le poids d'un graphe, c'est-à-dire la somme des poids de ses arêtes.

Question 18 ✍ Vérifier que les graphes des fichiers `g_10_sd` et `g_10_ad` ont des poids respectifs de 168 et 343. Quels sont les poids des graphes `g_1000_sd` et `g_1000_ad` ?

2.2 Arêtes inutiles et arêtes sûres

Soit $G = (S, A, f)$ un graphe non orienté pondéré. On suppose que $H = (S, B)$ est un sous-graphe de G (c'est-à-dire que $B \subseteq A$), supposé sans cycle. On considère $C \subseteq S$ une composante connexe de H. On appelle :

- arête **inutile** pour H une arête de $A \setminus B$ dont les deux extrémités sont dans C ;
- arête **sûre** pour H une arête de $A \setminus B$, non inutile, dont le poids est minimal parmi les arêtes qui ont une extrémité dans C.

Soit $T^* = (S, B^*)$ un arbre couvrant de poids minimal de G. On suppose pour les trois questions suivantes que $B \subseteq B^*$.

Question 19 ✍ Montrer que B^* ne contient aucune arête inutile pour H.

Question 20 ✍ Montrer que si f est injective, alors B^* contient toutes les arêtes sûres pour H.

Question 21 ✍ Donner un exemple simple pour G, T et H tel que f n'est pas injective et B^* ne contient pas toutes les arêtes sûres pour H.

Pour toute la suite du sujet, on considère un ordre total $\prec$ sur les arêtes de G défini de la manière suivante : si $a, a' \in A$ sont telles que $a = \{s, t\}$ et $a' = \{u, v\}$, avec $s < t$ et $u < v$, alors $a \prec a'$ si et seulement si $(f(a), s, t) <_{\text{lex}} (f(a'), u, v)$, où $<_{\text{lex}}$ désigne l'ordre lexicographique.

On considère une nouvelle définition d'arête sûre, comme une arête de $A \setminus B$, non inutile, qui est minimale pour $\prec$ parmi les arêtes qui ont une extrémité dans C.

Question 22 Écrire une fonction `int* composantes(graphe H, int* m)` qui prend en argument un graphe $H = (S, B, f)$ d'ordre n et renvoie un tableau `cc` de taille n tel que pour tout sommet $s \in S$, `cc[s]` est le numéro de la composante connexe de s. On supposera que les composantes connexes sont numérotées $0, 1, , m - 1$. La fonction devra également modifier l'entier pointé par `m` pour y stocker la valeur de m correspondant au nombre de composantes connexes.

Question 23 Écrire une fonction `arete* aretes_sures(graphe G, graphe H, int* cc, int m)` qui prend en argument un graphe H, le tableau contenant les numéros des composantes connexes et le nombre de composantes et renvoie un tableau d'arêtes `sures` pour H de taille m contenant, pour chaque composante connexe, l'arête sûre minimale pour $\prec$ ayant un sommet dans cette composante connexe.

Question 24 Déterminer la complexité temporelle de la fonction `aretes_sures` en fonction des dimensions du graphe G.

2.3 Algorithme de Borůvka

L'algorithme de Borůvka utilise les propriétés sur les arêtes sûres pour construire un arbre couvrant minimal. Il se résume de la manière suivante :

Entrée : Graphe pondéré non orienté connexe $G = (S, A, f)$.
Début algorithme
- Poser $B = \emptyset$
- **Tant que** il reste des arêtes sûres pour $H = (S, B)$ **Faire**
  - Ajouter à B toutes les arêtes sûres pour H.
- **Renvoyer** (S, B)

Question 25 Montrer que l'algorithme de Borůvka termine et renvoie un arbre couvrant de poids minimal de G.

Question 26 Écrire une fonction `graphe boruvka(graphe G)` qui applique l'algorithme de Borůvka et renvoie un arbre couvrant minimal de G.

Question 27 Vérifier que les arbres couvrants minimaux des graphes des fichiers `g_10_sd` et `g_10_ad` ont des poids respectifs de 66 et 117. Quels sont les poids des arbres couvrants minimaux des autres graphes ?

Question 28 Déterminer la complexité temporelle de la fonction `boruvka`. Cette complexité est-elle toujours atteinte en pratique ? Justifier.

⋆ ⋆ ⋆