Devoir maison n°2

À rendre le mercredi 13/10

⋆ ⋆ ⋆

Vous êtes encouragés à travailler sur ordinateur pour les questions de programmation. Pour tester vos fonctions, un fichier `graphes_exemples.ml` est disponible sur le dépôt `https://github.com/nathaniel-carre/MP-LLG`, dans le dossier `Devoirs/DM/DM2_arbres_couvrants`. Vous pourrez soit copier directement le code dans votre fichier, soit l'importer en utilisant la commande :

```
#use "graphes_exemples.ml";;
```

dans votre fichier principal, qui doit se trouver dans le même dossier (attention, ne fonctionne pas avec WinCaml, ou avec un éditeur en ligne).

1 Algorithme de Kruskal

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 Représentation des graphes

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 $f(s, t) = p$.

Question 3 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 4 Vérifier que les graphes `g1` et `g2` ont des poids respectifs de 168 et 38137. Quel est le poids du graphe `g3` ?

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, une grandeur qui caractérisera 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 5 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 6 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 7 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 8 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 9 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

On rappelle que OCaml peut comparer naturellement des n-uplets avec `<` ou `<=`, en procédant selon l'ordre lexicographique.

Question 10 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 11 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 12 Vérifier que les arbres couvrants minimaux des graphes `g1` et `g2` ont des poids respectifs de 66 et 18564. Quel est le poids de l'arbre couvrant minimal de `g3` ?

2 Algorithme de Borůvka

2.1 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 quelconque 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 $A \setminus B$, ayant une seule extrémité dans C et dont le poids est minimal parmi les arêtes ayant une seule extrémité dans C.

Soit $T^* = (S, B^*)$ un arbre couvrant de poids minimal de G. On suppose pour les trois questions suivantes que $H = (S, B)$ est un sous-graphe de T^*, c'est-à-dire tel que $B \subseteq B^*$.

Question 13 Montrer que B^* ne contient aucune arête inutile pour H.

Question 14 Montrer que si f est injective, alors B^* contient toutes les arêtes sûres pour H.

Question 15 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, où la minimalité est maintenant considérée selon la relation $\prec$.

Question 16 Écrire une fonction `composantes (h: graphe) : int * int array` qui prend en argument un graphe $H = (S, B, f)$ d'ordre n et renvoie un couple formé d'un entier m égal au nombre de composantes connexes de H et d'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 $0, 1, \ldots, m-1$.

Question 17 Écrire une fonction `aretes_sures (g: graphe) (h: graphe) (cc: int array) (m: int) : (int * int * int) array` 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 une extrémité dans cette composante connexe. Une arête sera ici représentée par un triplet (poids, sommet$_1$, sommet$_2$).

Le tableau pourra contenir deux fois une même arête si elle est minimale pour deux composantes connexes.

Question 18 Déterminer la complexité temporelle de la fonction `aretes_sures` en fonction des dimensions du graphe G.

2.2 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 19 Montrer que l'algorithme de Borůvka termine et renvoie un arbre couvrant de poids minimal de G.

Question 20 Écrire une fonction `boruvka (g: graphe) : graphe` qui applique l'algorithme de Borůvka et renvoie un arbre couvrant minimal de G.

Question 21 Vérifier que les poids des arbres couvrants minimaux de `g1`, `g2` et `g3` sont les mêmes que ceux trouvés par l'algorithme de Kruskal.

Question 22 Déterminer la complexité temporelle de la fonction `boruvka`. Cette complexité est-elle toujours atteinte en pratique ? Justifier.

⋆ ⋆ ⋆