I Algorithme de Prim

I.1 File de priorité implémenté par un tas (rappels de MP2I)

Une file de priorité (min) est une structure de données possédant les opérations suivantes :

- Extraire minimum : supprime et renvoie le minimum
- Ajouter élément
- Tester si la file de priorité est vide

On utilise souvent un tas pour implémenter une file de priorité. Un tas est un arbre binaire presque complet (tous les niveaux, sauf le dernier, sont complets) où chaque noeud est plus petit que ses fils. On stocke les noeuds du tas dans un tableau `t` tel que `t.(0)` est la racine, `t.(i)` a pour fils `t.(2*i + 1)` et `t.(2*i + 2)` si ceux-ci sont définis, et le père de `t.(j)` est `t.((j - 1)/2)` si $j \neq 0$.
On utilisera le type suivant pour représenter un tas : `type 'a tas = {t : 'a array; mutable n : int}`, où `n` est le nombre d'éléments du tas.

1. Écrire une fonction `swap : 'a tas -> int -> int -> unit` qui échange les éléments `tas.t.(i)` et `tas.t.(j)`.

On utilise deux fonctions auxiliaires pour implémenter les opérations sur un tas :

2. Écrire une fonction `up tas i` qui suppose que `tas` est un tas sauf `tas.t.(i)` qui peut être inférieur à son père et fait monter `tas.t.(i)` (en l'échangeant avec son père) de façon à obtenir un tas.
3. En déduire une fonction `add tas x` qui ajoute l'élément `x` à `tas` (en tant que feuille la plus à droite) et la fait remonter pour conserver la propriété de tas.
4. Écrire une fonction `down tas i` qui suppose que `tas` est un tas sauf `tas.t.(i)` qui peut être supérieur à un fils et fait descendre `tas.t.(i)` de façon à obtenir un tas.
5. En déduire une fonction `extract` qui extrait et renvoie le minimum d'un tas en remplaçant la racine par la dernière feuille et en la faisant descendre.
6. Montrer que les fonctions `add` et `extract` sont en $O(\log(n))$ où n est le nombre d'éléments du tas.

I.2 Algorithme de Prim

Soit $G = (S, A)$ un graphe pondéré.

7. Soit $X \subsetneq S$ et $e \in A$ une arête de poids minimum ayant exactement une extrémité dans X. Montrer qu'il existe un arbre couvrant de poids minimum contenant e.
8. En déduire un algorithme en pseudo-code pour trouver un arbre couvrant de poids minimum de G.
9. Écrire une fonction `prim : (int * float) list array -> int array` qui implémente cet algorithme, où :
   - G est représenté par une liste d'adjacence pondérée `g` : `g.(i)` contient une liste de couples `(j, w)` où j est un sommet adjacent à i et w est le poids de l'arête entre i et j.
   - `prim g` renvoie un tableau `p` tel que `p.(i)` contient le prédécesseur de i dans un arbre couvrant de poids minimum.
10. Donner la complexité de votre algorithme. Comparer avec l'algorithme de Kruskal.
11. Comment modifier l'algorithme de Prim pour obtenir l'algorithme de Dijkstra permettant de calculer les distances dans un graphe pondéré par des poids positifs ?

II Autour de la complexité de l'algorithme de Kruskal

Soit $G = (S, A)$ un graphe pondéré, $n = |S|$ et $p = |A|$.

1. Écrire une fonction `void tri(int* t, int n)` qui trie un tableau `t` contenant n entiers entre 0 et K, en $O(n + K)$.
2. On suppose que les poids des arêtes de G sont compris entre 0 et p et que les opération de Union-Find en $O(1)$. Expliquer comment implémenter l'algorithme de Kruskal en $O(n + p)$.
3. Montrer que la fonction suivante (réutilisant le code de l'exercice précédent) transforme un tableau en un tas en $O(n)$:

```
let tab_en_tas t =
    let n = Array.length t in
    let tas = { t=t; n=n } in
    for i = n/2 - 1 downto 0 do
        down tas i;
    done;
    tas
```

4. Expliquer comment modifier l'algorithme de Kruskal en utilisant la question précédente pour améliorer sa complexité dans certains cas.

III Union par taille

Dans le cours, on a étudié l'optimisation de la structure d'Union-Find en utilisant l'union par rang, en attachant l'arbre de hauteur la plus petite à celui de hauteur la plus grande. On peut aussi utiliser l'union par taille, en attachant l'arbre de plus petit nombre de sommets à celui de plus grand nombre de sommets.
On choisit alors d'utiliser un tableau `uf` tel que `uf.(i)` contient :

- $-k$ si i est une racine, où k est le nombre de sommets dans l'arbre de racine i.
- le prédécesseur de i dans l'arbre sinon.

1. Écrire les fonctions `create`, `find` et `union` en utilisant l'union par taille (et pas d'autre optimisation).
2. Montrer que l'opération `find` et `union` sont en $O(\log(n))$ pour une partition d'un ensemble de n éléments.

IV Questions sur les arbres couvrants de poids minimum

Soit $G = (S, A)$ un graphe pondéré.

1. Un dominant de G est un ensemble $X \subseteq S$ tel que $\forall v \in S$, $v \in X$ ou v est adjacent à un sommet de X.
   On note $d(G)$ la taille minimum d'un dominant de G.
   Montrer que si G est connexe alors $d(G) \leq \dfrac{|S|}{2}$.
2. Soit C un cycle de G et $e = \{u, v\}$ une arête de C dont le poids est strictement supérieur au poids des autres arêtes de C. Montrer que e ne peut pas appartenir à un arbre couvrant de poids minimum de G.
3. (Propriété d'échange) Soient T_1, T_2 deux arbres couvrants de G et e_1 une arête de $T_1 - T_2$. Montrer qu'il existe une arête e_2 de T_2 telle que $T_1 - e_1 + e_2$ (le graphe obtenu en remplaçant e_1 par e_2 dans T_1) est un arbre couvrant de G.
4. Soit T_1 un arbre couvrant de poids minimum de G et T_2 le 2ème plus petit arbre couvrant, c'est-à-dire l'arbre couvrant de poids minimum en excluant T_1. Montrer que T_1 et T_2 diffèrent d'une arête et en déduire un algorithme pour trouver T_2.