I Algorithme de Borůvka

Soit $G = (S, A)$ un graphe non-orienté connexe et pondéré par $w : A \longrightarrow \mathbb{R}$. On note $n = |S|$ et $p = |A|$.

I.1 Théorie

1. Montrer que G possède un arbre couvrant de poids minimum.

Dans toute la suite, on suppose que tous les poids de G sont distincts (c'est-à-dire : w injective) et que $S = \{0, ..., n-1\}$.

2. Montrer que G possède un unique arbre couvrant de poids minimum.

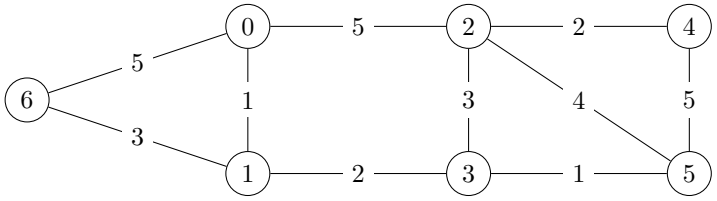
On appelle T^* l'unique arbre couvrant de poids minimum de G.
Soit $X \subset S$. On dit qu'une arête est sûre pour X si elle est de poids minimum parmi les arêtes ayant exactement une extrémité dans X. Autrement dit, une arête e est sûre pour X si $w(e) = \min\{w(e') \mid \{u, v\} \in A, u \in X, v \notin X\}$.
L'objectif de l'algorithme de Borůvka est de construire un arbre couvrant de poids minimum T en conservant une partition F de S, correspondant aux composantes connexes de T.
À chaque étape, on ajoute une arête sûre pour chaque composante connexe de F :

Algorithme de Borůvka

$F \longleftarrow \{\{x\} \mid x \in S\}$
$T \longleftarrow \emptyset$
Tant que $|F| > 1$ **:**
  $E \longleftarrow \emptyset$
  Pour $C \in F$ **:**
    $e \longleftarrow$ arête sûre pour C
    $E \longleftarrow E \cup \{e\}$
  $F \longleftarrow$ partition de S obtenue en fusionnant les composantes connexes de F avec les arêtes de E
  $T \longleftarrow T \cup E$
Renvoyer T

L'étape de fusion des composantes connexes consiste, pour chaque arête $e = \{u, v\}$ de E, à remplacer dans F les composantes connexes C_1 et C_2 contenant u et v par leur union $C_1 \cup C_2$.

3. Appliquer l'algorithme de Borůvka sur le graphe suivant, en donnant à chaque l'ensemble des arêtes de T à la fin de chaque passage dans la boucle **Tant que** :



4. Montrer que l'algorithme de Borůvka termine, en utilisant un variant de boucle.
5. Soit $X \subset S$ et e une arête sûre pour X. Montrer que T^* contient e.
6. Montrer que l'algorithme de Borůvka renvoie bien T^*.

I.2 Implémentation

On va utiliser une structure d'Union-Find pour représenter les composantes connexes de F, sous la forme d'un tableau `uf` de taille n tel que `uf.(x)` soit le père de `x` dans l'arbre contenant `x`. Si `x` est une racine, `uf.(x)` contiendra `x`.
On n'utilisera pas d'optimisation de type union par rang ou compression de chemin.

7. Écrire une fonction `create : int -> int array` telle que `create n` renvoie un tableau de taille `n` initialisé avec les entiers de 0 à `n - 1`.
8. Écrire une fonction `find : int array -> int -> int` telle que `find uf x` renvoie la racine de l'arbre contenant `x` dans la structure d'Union-Find représentée par le tableau `uf`.

9. Écrire une fonction `union : int array -> int -> int -> unit` telle que `union uf x y` fusionne les composantes connexes de `x` et `y` dans la structure d'Union-Find représentée par le tableau `uf`.
10. Écrire une fonction `meme_cc : int array -> int -> int -> bool` telle que `meme_cc uf x y` détermine si `x` et `y` sont dans la même composante connexe.
11. Écrire une fonction `n_cc : int array -> int` telle que `n_cc uf` renvoie le nombre de composantes connexes dans `uf`.

Le graphe G est représenté par une liste d'adjacence `g` telle que `g.(i)` contient une liste des arêtes partant de `i`, où chaque arête est un couple (w, j) où w est le poids de l'arête et i et j les extrémités de l'arête.

12. Écrire une fonction `aretes_sures : (float * int) list array -> int array -> (float * int * int) array` telle que
`aretes_sures g uf` renvoie un tableau `ans` de taille n où, si `i` est une racine dans `uf`, `ans.(i)` contient l'arête sûre pour la composante connexe de `i`.
Si `i` n'est pas une racine, `ans.(i)` contiendra `(max_float, -1, -1)`.
13. Écrire une fonction `boruvka : (float * int) list array -> (float * int) list array` renvoyant l'arbre couvrant de poids minimum de G par l'algorithme de Borůvka.
14. Quitte à utiliser l'optimisation par compression de chemin et union par rang, on suppose que `union` et `find` sont en $\mathrm{O}(1)$. Montrer que la complexité de l'algorithme de Borůvka est en $O(p \log n)$. Comparer avec l'algorithme de Kruskal.

II Algorithme de Johnson

Soit $G = (S, A)$ un graphe orienté pondéré par $w : A \longrightarrow \mathbb{R}$ (des poids peuvent être négatifs). On note $n = |S|$ et $p = |A|$.
Comme les poids de G peuvent être négatifs, l'algorithme de Dijkstra ne peut pas être utilisé pour trouver tous les plus courts chemins.
L'algorithme de Johnson consiste à modifier les poids de G pour les rendre positifs, sans modifier les plus courts chemins.

1. Rappeler la complexité $C(n, p)$ de l'algorithme de Dijkstra.
2. Soit $h : S \longrightarrow \mathbb{R}$. On définit $w_h : (u, v) \longmapsto w(u, v) + h(u) - h(v)$. Montrer que, dans G, les plus courts chemins pour w et w_h sont les mêmes et qu'il existe un cycle de poids négatif pour w si et seulement s'il en existe un pour w_h.

Dans la suite, on suppose que G n'a pas de cycle de poids négatif.

3. Trouver $h : S \longrightarrow \mathbb{R}$ telle que $\forall e \in A, w_h(e) \geq 0$. On pourra supposer dans un premier temps que tous les sommets de G sont atteignables depuis un sommet r.
4. On admet qu'il est possible de trouver les distances depuis un sommet fixé dans G en $\mathrm{O}(np)$. En déduire un algorithme en pseudo-code permettant de trouver toutes les distances entre les sommets de G en $\mathrm{O}(np \log(n))$. Comparer avec l'algorithme de Floyd-Warshall.