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 $p = |A|$. On suppose que tous les poids de G sont distincts (c'est-à-dire : w injective) et que $S = \{0, ..., n-1\}$.

I.1 Théorie

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

> Solution : Étant connexe, G possède bien un arbre couvrant, obtenu par exemple avec un arbre de parcours en profondeur (constitué de toutes les arêtes parcourues dans le DFS).
> Ainsi l'ensemble $E = \{w(T) \mid T \text{ arbre couvrant de } G\}$ est non vide et fini donc il possède un minimum.

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

> Solution : Supposons par l'absurde que G possède deux arbres couvrants T et T' de poids minimum. Soit e l'arête de poids minimum appartenant à exactement un de ces deux arbres. Supposons par exemple que e appartient à T.
> Comme il contient n sommets et n arêtes, $T' + e$ contient un cycle C. T n'a pas de cycle donc C contient une arête e' qui n'appartient pas à T.
> Soit $T'' = T' + e - e'$. Alors :
>
> - T'' est connexe. En effet, si $u, v \in S$ alors il existe un chemin P de u à v dans T'. Si P passe par e', on remplace e' par le reste du chemin dans C. Ainsi, il existe un chemin de u à v dans T''.
> - T'' est un arbre couvrant car il contient $n-1$ arêtes et est connexe.
> - $w(T'') = w(T') + w(e) - w(e') < w(T')$ car $w(e) < w(e')$, tous les poids étant distincts.
>
> T'' contredit l'hypothèse de minimalité de T'. Ainsi, G possède bien un unique arbre couvrant de poids minimum.

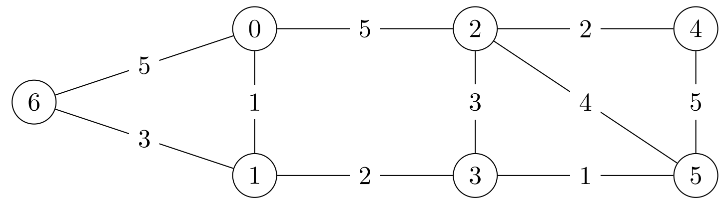
On appelle T^* l'unique arbre couvrant de poids minimum de G.
Soit $X \subset S$. On dit qu'une arête est sûre pour X si elle est de poids minimum parmi les arêtes ayant exactement une extrémité dans X. Autrement dit, une arête e est sûre pour X si $w(e) = \min\{w(e') \mid \{u, v\} \in A, u \in X, v \notin X\}$.
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 :

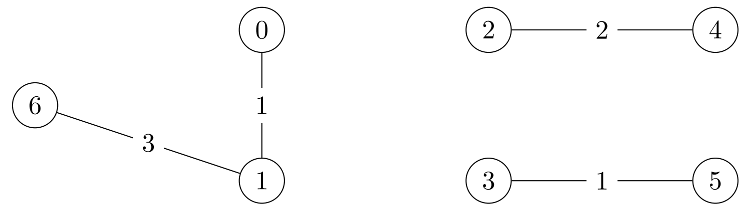
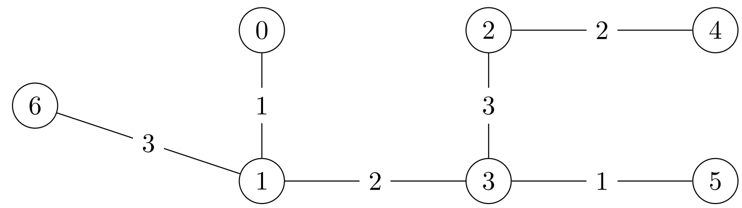
Algorithme de Borůvka

```
F ⟵ {{x} | x ∈ S}
T ⟵ ∅
Tant que |F| > 1 :
    E ⟵ ∅
    Pour C ∈ F :
        e ⟵ arête sûre pour C
        E ⟵ E ∪ {e}
    F ⟵ partition de S obtenue en fusionnant les composantes
        connexes de F avec les arêtes de E
    T ⟵ T ∪ 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** :



> Solution :
>
> 
>
> 

4. Montrer que l'algorithme de Borůvka termine, en utilisant un variant de boucle.

> Solution : $|F|$ est strictement décroissant à chaque itération de la boucle **Tant que** et $|F| \geq 0$ donc l'algorithme termine.

5. Soit $X \subset S$ et e une arête sûre pour X. Montrer que T^* contient e.

> Solution : Notons $e = \{u, v\}$ et supposons que T^* ne contienne pas e.
> $T^* + e$ contient un cycle C car il a n arêtes pour n sommets.
> Soit $e^* \neq e$ une arête de C ayant exactement une extrémité dans X. $w(e) < w(e^*)$ car e est sûre pour X.
> $T^* + e - e^*$ contient $n - 1$ arêtes et est acyclique (s'il possédait un cycle C', on pourrait y remplacer e par le reste du cycle dans C pour obtenir un cycle dans T^*).
> Ainsi, $T^* + e - e^*$ est un arbre couvrant de poids strictement inférieur à T^*, ce qui est absurde.

6. Montrer que l'algorithme de Borůvka renvoie bien T^*.

> Solution : L'algorithme ne rajoute dans T que des arêtes appartenant à T^*, donc T est toujours un sous-ensemble de T^*. De plus, T ne contient qu'une composante connexe à la fin donc $T = T^*$.

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`.

> Solution :
>
> ```
> let create n =
>   let uf = Array.make n 0 in
>   for i = 0 to n - 1 do
>     uf.(i) <- i
>   done;
>   uf
> ```

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`.

> Solution :
>
> ```
> let rec find uf i =
>     if uf.(i) = i then i
>     else find uf uf.(i)
> ```

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`.

> Solution :
>
> ```
> let union uf x y =
>     let rx = find uf x in
>     let ry = find uf y in
>     uf.(rx) <- ry
> ```

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.

> Solution :
>
> ```
> let meme_cc uf i j =
>     find uf i = find uf j
> ```

11. Écrire une fonction `n_cc : int array -> int` telle que `n_cc uf` renvoie le nombre de composantes connexes dans `uf`.

> Solution :
>
> ```
> let n_cc uf =
>   let n = Array.length uf in
>   let ans = ref 0 in
>   for i = 0 to n - 1 do
>     if uf.(i) = i then incr ans
>   done;
>   !ans
> ```

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)`.

> Solution : On parcourt chaque arête et on met à jour l'arête sûre de la composantes connexe si elle est plus petite.

```
let aretes_sures g uf =
  let n = Array.length g in
  let ans = Array.make n (max_float, -1, -1) in
  for u = 0 to n - 1 do
    let rec aux = function
      | [] -> ()
      | (w, v) :: q -> if not (meme_cc uf u v) then (
        let e = (w, u , v) in
        let e_ = ans.(find uf u) in
        if e_ > e then ans.(find uf u) <- e;
        let e_ = ans.(find uf v) in
        if e_ > e then ans.(find uf v) <- e
      );
      aux q in
    aux g.(u)
  done;
  ans
```

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.

Solution :

```
let boruvka g =
  let n = Array.length g in
  let uf = create n in
  let t = Array.make n [] in
  while n_cc uf > 1 do
    let ans = aretes_sures g uf in
    for i = 0 to n - 1 do
      let (w, u, v) = ans.(i) in
      if u <> -1 then (
        t.(u) <- (w, v) :: t.(u);
        t.(v) <- (w, u) :: t.(v);
        union uf u v
      )
    done
  done;
  t;;
```

14. Quitte à utiliser l'optimisation par compression de chemin et union par rang, on suppose que `union` et `find` sont en O(1). Montrer que la complexité de l'algorithme de Borůvka est en $O(p \log n)$. Comparer avec l'algorithme de Kruskal.

Solution : À chaque étape, $|F|$ est divisé au moins par deux donc l'algorithme s'arrête en $O(\log n)$ étapes.
À chaque étape, on parcourt en O(p) toutes les arêtes.
La complexité est donc en $O(p \log n)$, comme 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.

Solution : $O(p \log(n) + n)$ si on utilise une file de priorité.

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.

Solution : Soit C un chemin de u à v. Notons $u = v_1, v_2, ..., v_k = v$ les sommets utilisés par C. Alors $w_h(C) = \sum_{(v_i, v_{i+1})} w(v_i, v_{i+1}) + h(v_i) - h(v_{i+1}) = w(C) + h(u) - h(v)$ (somme télescopique). $h(u) - h(v)$ ne dépend pas du chemin C, seulement des extrémités. Donc un chemin C de poids minimum de u à v est obtenu quand $w(C)$ est minimum, ce qui montre le résultat.

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.

Solution : Soit $h(v) = d(r, v)$. Si $(u, v) \in A$, il existe un chemin de r à u de poids $d(r, u)$ auquel on peut ajouter (u, v) pour obtenir un chemin de r à v. Comme $d(r, v)$ est le poids d'un plus court chemin de r à v, $d(r, v) \leq d(r, u) + w(u, v)$, d'où $w_h \geq 0$.
S'il n'existe pas de tel sommet r, on peut ajouter un sommet r relié à tous les autres sommets par des arêtes de poids nul.

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.

Solution : On peut calculer h en $\mathrm{O}(np)$ avec l'algorithme de Bellman-Ford. Ensuite, on peut appliquer Dijkstra n fois en $\mathrm{O}(np \log(n))$ pour obtenir les distances entre tous les sommets.
S'il reste du temps on peut faire retrouver l'algorithme de Bellman-Ford (sujet 122).