Dans tout ce cours, $G = (S, A)$ désigne un graphe non-orienté pondéré par $p : A \to \mathbb{R}$.

# I Arbre couvrant de poids minimum

**Définition : Arbre couvrant**

On dit que $T = (S', A')$ est un arbre couvrant de $G$ si :
- $T$ est un sous-graphe de $G$, c'est-à-dire : $S' \subset S$ et $A' \subset A$.
- $T$ est un arbre.
- $T$ contient tous les sommets de $G$ : $S' = S$.

On définit le poids $p(T)$ de $T$ comme la somme des poids des arêtes de $T$.

**Définition : Arbre couvrant de poids minimum**

Un arbre couvrant dont le poids est le plus petit possible est appelé un arbre couvrant de poids minimum.

**Exercice 1.**
Donner un graphe qui possède plusieurs arbres couvrants de poids minimum.

**Lemme**

Tout graphe connexe possède un arbre couvrant de poids minimum.

Preuve :

# II Algorithme de Kruskal

Algorithme de Kruskal

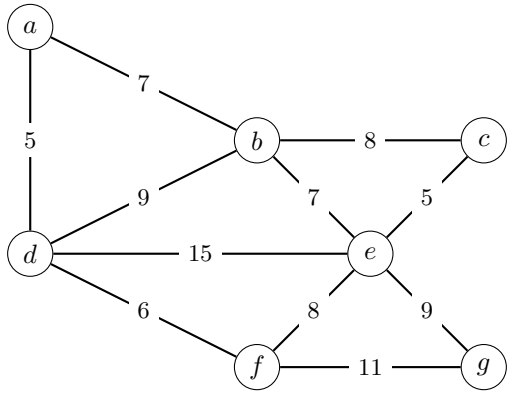
**Entrée :** Un graphe connexe $G = (S, A)$
**Sortie :** Un arbre couvrant de poids minimum $T$

Trier les arêtes de A par poids croissant.
$T \leftarrow$ arbre vide (aucune arête).
**Pour** chaque arête $e$ par poids croissant **:**
  **Si** $T + e$ est acyclique **:**
    $T \leftarrow T + e$
**Renvoyer** $T$

Remarques :

- S'il y a plusieurs arêtes de même poids, l'algorithme de Kruskal les choisit dans un ordre quelconque.
- L'algorithme de Kruskal est un algorithme glouton.
- « $T$ est acyclique » est un invariant de boucle. Une variante (« Kruskal inversé ») consiste à partir de $T = G$ et à retirer les arêtes par ordre décroissant de poids, tout en conservant la connexité comme invariant.

Exemple : les arêtes ajoutées à $T$ pour le graphe ci-dessous sont, dans l'ordre : _____




Exemple d'application de l'algorithme de Kruskal

**Théorème**

L'algorithme de Kruskal renvoie un arbre couvrant de poids minimum.

Preuve ♡:

**Exercice 2.**

1. Peut-on adapter l'algorithme de Kruskal pour trouver un arbre couvrant de poids maximum ?
2. De façon similaire, peut-on adapter un algorithme de plus courts chemins (par exemple Dijkstra) pour trouver des chemins de poids maximum ?
3. Soit $e \in A$. Peut-on adapter l'algorithme de Kruskal pour trouver un arbre couvrant de poids minimum contenant $e$ ?

# III Implémentation naïve

On suppose $G$ implémenté par une liste d'adjacence `g : (int*float) list array` telle que `g.(i)` est la liste des couples $(j, p)$ tels que $\{i, j\}$ est une arête de poids $p$.

La fonction suivante renvoie la liste des arêtes du graphe $g$, où une arête $\{u, v\}$ de poids $p$ est représentée par le couple $(u, v, p)$ :

```
let aretes (g : (int*float) list array) : (int*int*float) list =
    let rec aux i =
        if i = Array.length g then []
        else
            let rec aux_i = function (* liste des arêtes partant de i *)
                | [] -> []
                | (j, p)::q -> (i, j, p)::aux_i q in
            aux_i g.(i) @ aux (i + 1) in
    aux 0
```

Pour chaque arête $\{u, v\}$, déterminer si l'ajout de cette arête créé un cycle revient à savoir si $u$ et $v$ sont dans la même composante connexe de l'arbre $T$ en construction.

**Exercice 3.**
Écrire une fonction `chemin g u v` qui détermine s'il y a un chemin de `u` à `v` dans `g`.

On suppose l'existence d'une fonction `tri : ('a*'a*float) list -> ('a*'a*float) list` qui trie une liste d'arêtes par ordre croissant de poids, en complexité $O(p \log(p)) = O(p \log(n))$ (par exemple par tri fusion).

```
let ajout_arete g u v p =
    g.(u) <- (v, p)::g.(u);
    g.(v) <- (u, p)::g.(v)

let kruskal g =
    let n = Array.length g in
    let t = Array.make n [] in
    let rec aux l = match l with
        | [] -> t
        | (u, v, p)::q ->
            if not (chemin t u v) then ajout_arete t u v p;
            aux q in
    g |> aretes |> tri |> aux
```

Remarque : `g |> aretes |> tri |> aux` est équivalent à `aux (tri (aretes g))`

Complexité :

# IV Union-Find

La structure Union-Find (unir et trouver) permet de représenter une partition d'un ensemble $E = [\![0, n-1]\!]$ comme réunion disjointe de sous-ensembles (classes).
À chaque élément de $E$ est associé un représentant, qui est l'élément de sa classe.
Opérations sur une structure d'Union-Find :

- Création : créer une structure Union-Find avec $n$ éléments, chaque élément étant seul dans sa classe.
- Find : trouver le représentant de la classe d'un élément.

- Union : fusionner les classes de deux éléments.

Chaque classe est représentée par un arbre, enraciné en le représentant.

Par exemple, la forêt suivante est une représentation possible de la partition $\{\{1,3,4\},\{2\},\{0,5\}\}$ :


On la représente par un tableau `uf` tel que `uf.(i)` est le père de $i$ dans l'arbre (`uf.(i) = i` si `i` est le représentant de sa classe).
Sur l'exemple ci-dessus, `uf = [|0; 3; 2; 3; 3; 0|]`.
Pour réaliser l'union des classes de $x$ et $y$, on cherche leurs représentants $r_x$ et $r_y$ et choisit, par exemple, $r_x$ comme père de $r_y$.

**Exercice 4.**

1. Écrire une fonction `create n` qui crée une structure Union-Find avec $n$ éléments.
2. Écrire une fonction `find uf i` qui renvoie le représentant de la classe de `i`.
3. Écrire une fonction `union uf i j` qui fusionne les classes de `i` et `j`.
4. Quelles sont les complexités des fonctions précédentes ?

Application d'Union-Find à Kruskal :

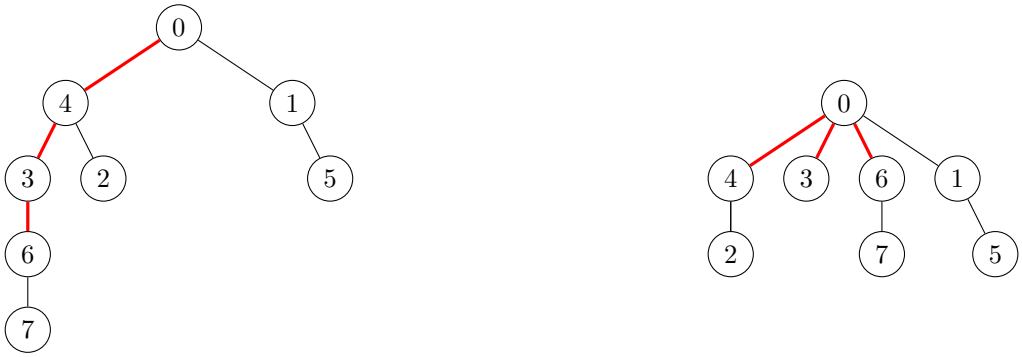
- Chaque classe correspond à une composante connexe dans $T$.
- Si $u$ et $v$ sont dans la même classe (`find t u = find t v`) alors l'ajout de l'arête $\{u,v\}$ à $T$ créerait un cycle.
- Sinon, ajouter l'arête à $T$ et fusionner les classes de $u$ et $v$ : `union t u v`).

```
let kruskal g =
    let n = Array.length g in
    let t = Array.make n [] in
    let uf = create n in
    let rec aux = function
    | [] -> t
    | (u, v, p)::q ->
        if find uf u <> find uf v then (
            union uf u v;
            ajout_arete t u v p
        );
        aux q in
    g |> aretes |> tri |> aux
```

Complexité :

# V Améliorations d'Union-Find

On peut améliorer la structure d'Union-Find en utilisant l'heuristique de l'union par rang et la compression de chemin :




Exemple : appel de `find uf 6` avec compression de chemin

- Union par rang : dans `union`, on attache l'arbre de hauteur la plus petite à celui de hauteur la plus grande.
- Compression de chemin : dans `find`, on attache directement chaque nœud rencontré à la racine.

On ajoute un tableau `h` tel que `h.(i)` est la hauteur de l'arbre enraciné en $i$.

```
type uf = {t : int array; h : int array}

let create n = (* O(n) *)
    {t = Array.init n (fun i -> i); h = Array.make n 0}

let union uf x y = (* O(1) *)
    let rx = find uf x in
    let ry = find uf y in
    if rx <> ry then (
        if uf.h.(rx) < uf.h.(ry) then uf.t.(rx) <- ry
        else if uf.h.(rx) > uf.h.(ry) then uf.t.(ry) <- rx
        else (
            uf.t.(ry) <- rx;
            uf.h.(rx) <- uf.h.(rx) + 1
        )
    )
```

**Théorème**

Dans une structure d'Union-Find avec union par rang, la hauteur $h$ d'un arbre à $k$ nœuds vérifie $h \leq \log_2(k)$ (dit autrement : $k \geq 2^h$).

Preuve :

```
let rec find uf i =
    if uf.t.(i) = i then i
    else (
        let r = find uf uf.t.(i) in
        uf.t.(i) <- r;
        r
    )
```

**Théorème : (admis)**

Avec union par rang et compression de chemin, la complexité amortie de `union` et `find` est en $O(\alpha(n))$, où $\alpha$ est une fonction à croissance tellement lente qu'on peut la considérer comme constante.

Ne pas confondre :

- Complexité en moyenne : on moyenne la complexité sur toutes les entrées possibles.
- Complexité amortie : complexité dans le pire cas d'une suite de $n$ opérations, divisé par $n$.

**Théorème**

Avec union par rang et compression de chemin, la complexité amortie de l'algorithme de Kruskal est en $O(p \log(n))$.