TP Python – Structures de données et algorithmique 4

Objectifs

- Comprendre la notion d'**arbre couvrant minimum** (*Minimum Spanning Tree*) pour un graphe non-orienté pondéré.
- Découvrir la structure de donnée **union-find** pour gérer les partitions d'un ensemble et sa complexité amortie remarquable
- Implémentation de l'algorithme de **Kruskal** et calcul de sa complexité

Arbre couvrant de poids minimal

Un **arbre couvrant** (*Spanning Tree*) d'un graphe non orienté $G = (V, E)$ est un **sous-graphe** $T = (V, E')$ (avec $E' \subseteq E$) **connexe** et **sans cycle**, i.e. un **arbre** qui connecte tous les sommets du graphe. Construire un tel arbre est un problème qui a de nombreuses applications dans la construction de réseaux électriques, hydrauliques et de communication ainsi que pour le routage dans les réseaux informatiques (pour permettre à un nœud du réseau de joindre tous les autres).

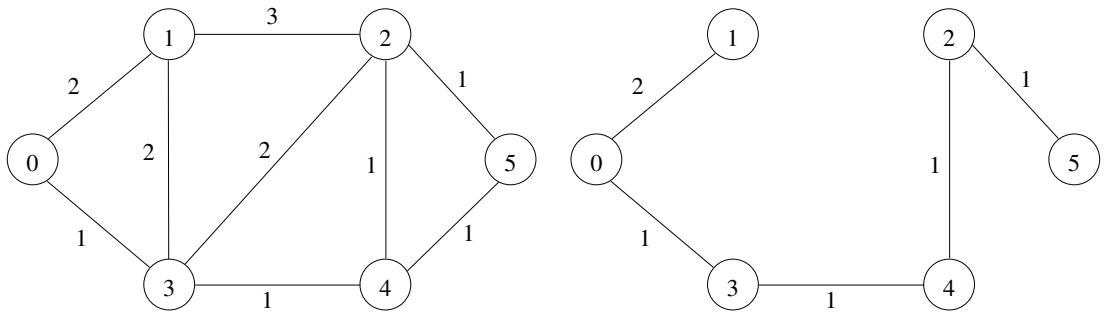


Figure 1: Un graphe non-orienté pondéré et un arbre couvrant minimum (de poids 6).

Si le graphe est pondéré (sur les arêtes), il peut alors être intéressant de calculer l'**arbre couvrant de poids minimal** (*Minimum Spanning Tree*, MST), c'est-à-dire l'arbre couvrant dont la somme des poids des arêtes est minimal, comme illustré figure 1. La résolution de ce problème permet d'optimiser la construction de réseaux en utilisant la longueur minimale de cable ou de conduite ou d'économiser les ressources d'un réseau informatique lors de la diffusion de messages.

Algorithme de Kruskal

Nous allons implémenter l'algorithme de Kruskal pour calculer le MST d'un graphe. Cet algorithme est un **algorithme glouton** qui garantit néanmoins de trouver la solution optimale en ajoutant les arêtes une à une à un arbre initialement vide selon le principe suivant :

— $S =$ les arêtes triées par poids croissant ;
— $A = \emptyset$;
— tant que A n'est pas un arbre couvrant :
  — retirer l'arête de poids minimal de S ;
  — si les extrémités de l'arête sont déjà dans la même composante connexe, on la rejette car on créerait un cycle en l'ajoutant ;
  — sinon on ajoute l'arête à A.

Si le graphe G est connexe, on peut facilement déterminer quand l'arbre est couvrant : en effet, un arbre à n sommets possède exactement $n-1$ arêtes, donc il suffit de compter le nombre d'arêtes ajoutées et de s'arrêter à $n-1$. La seule difficulté de l'algorithme reste donc de déterminer si les extrémités des arêtes sont dans la même composante connexe de l'arbre en construction.

Union-Find

Pour résoudre efficacement ce problème, on peut utiliser la structure de donnée **Union-Find** qui permet d'effectuer les opérations nécessaires pour maintenir les composantes connexes de l'arbre en construction :

— Find : déterminer à quelle composante connexe appartient un sommet.
— Union : fusionner les deux composantes connexes auxquelles appartiennent les extrémités des arêtes qu'on ajoute à l'arbre.

En effet, il suffit d'initialiser la structure Union-Find avec une composante connexe par sommet de V (i.e. l'arbre $(V, \emptyset)$ n'a encore aucune arête), puis, à chaque fois qu'on retire une arête (u, v) de S, on utilise l'opération *find* pour savoir si u et v sont dans la même composante connexe. S'ils sont dans des composantes connexes différentes, on ajoute l'arête à l'arbre et on utilise l'opération *union* pour fusionner les deux composantes. Sinon on passe à l'arête suivante.

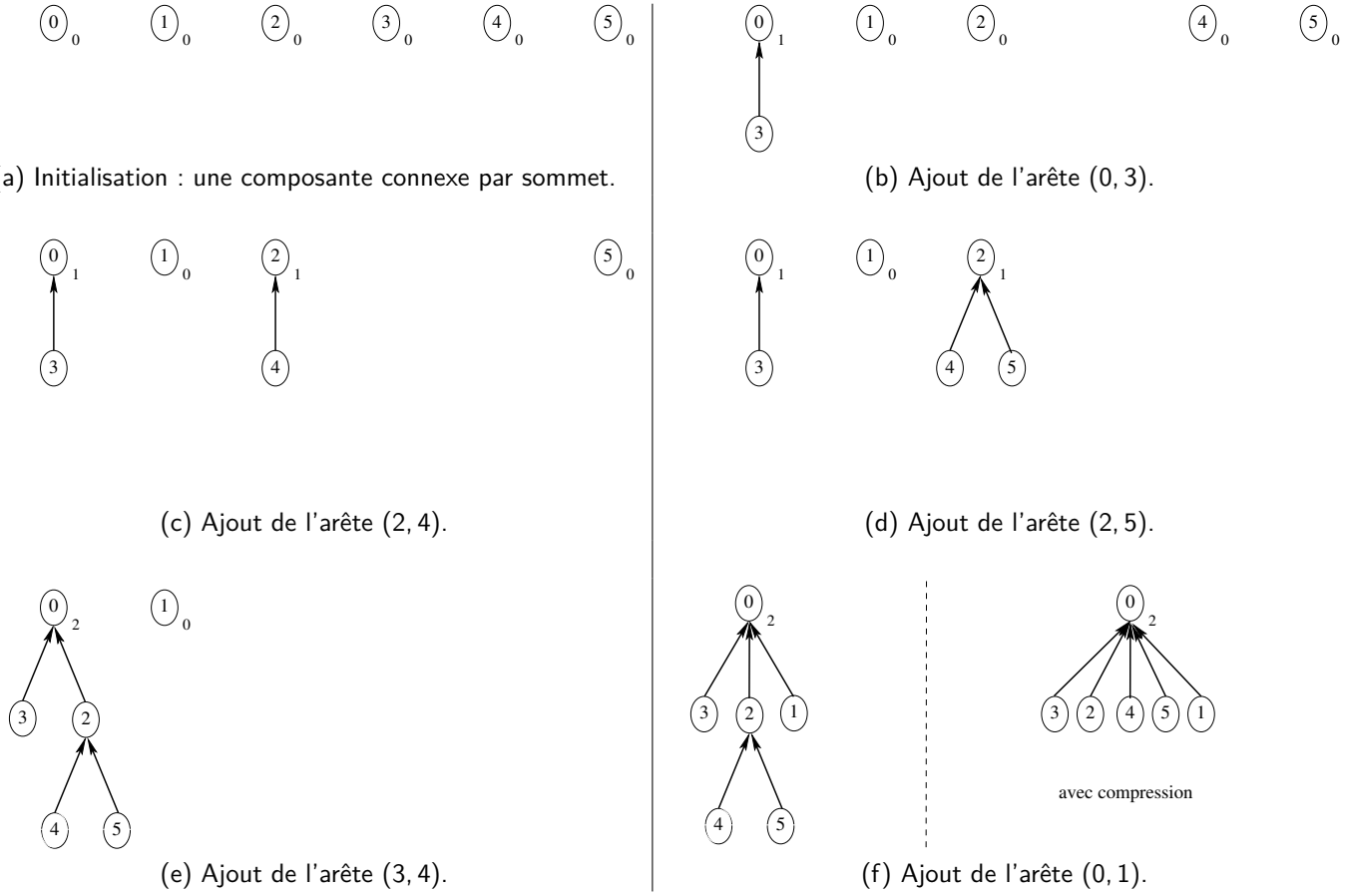
Forêt Pour maintenir cette partition des sommets, la structure Union-Find utilise une **forêt** d'arbres, c'est-à-dire un ensemble d'arbres qui représentent chacun une composante connexe et forment une partition de V. Les nœuds d'un arbre représentent donc les différents sommets de la composante, et la composante elle-même sera représentée par le sommet situé à sa racine, considéré comme l'élément caractéristique de sa classe. Ces arbres sont « n-aires », c'est-à-dire qu'un nœud peut avoir un nombre quelconque d'enfants ; cependant, on a seulement besoin de connaître le parent d'un nœud, et non ses enfants, pour réaliser les opérations sur cette structure, donc un simple tableau ou dictionnaire des parents suffit.

Find L'opération *find* consiste alors, pour un sommet donné, à remonter dans l'arbre jusqu'à sa racine [1] pour renvoyer son élément représentatif. La complexité de cette opération dépend donc de la *profondeur* du nœud dans l'arbre, et on aura intérêt à « aplatir » au maximum les arbres pour limiter leur profondeur. La question 5 présente ainsi une amélioration de l'opération find appelée « compression de chemin » et qui permet de profiter des résultats d'un appel à find pour créer des « raccourcis » vers la racine sans changer la complexité.

Union L'opération d'*union* de deux composantes connexes, quant à elle, consiste simplement à ajouter l'un des arbres comme nouvel enfant de la racine de l'autre – ainsi, on n'augmentera que d'une unité la distance entre les nœuds du nouvel enfant et la racine. Pour éviter de trop faire grandir les arbres

1. En informatique, les arbres poussent vers le bas...

en profondeur et limiter ainsi l'éloignement de la racine, on maintient pour chaque arbre son **rang** (attribut `rank` dans l'implémentation), qui correspond en première approximation à sa profondeur, et on va ajouter l'arbre de plus petit rang sous la racine de l'arbre de plus grand rang, en mettant à jour son parent (attribut `pred`). Ainsi, le rang de ce dernier ne sera pas modifié. En revanche, si les arbres ont le **même rang**, on peut les fusionner dans un sens ou dans l'autre, mais il faut **incrémenter** le rang de celui qui reçoit l'autre comme nouvel enfant de sa racine. La figure 2 illustre les opérations d'union réalisée par l'algorithme de Kruskal sur la structure Union-Find associée au graphe de la figure 1 (avec le rang indiqué à côté du nœud).



(a) Initialisation : une composante connexe par sommet.

(b) Ajout de l'arête $(0, 3)$.

(c) Ajout de l'arête $(2, 4)$.

(d) Ajout de l'arête $(2, 5)$.

(e) Ajout de l'arête $(3, 4)$.

(f) Ajout de l'arête $(0, 1)$.

Figure 2: Opérations successives d'union lors de l'algorithme de Kruskal sur la structure Union-Find du graphe de la figure 1. Le rang des arbres est noté à côté de la racine. Dans la sous-figure 2f, la (vaine) tentative d'ajout de l'arête $(4, 5)$ (avant d'ajouter l'arête suivante $(0, 1)$) permet de faire remonter les nœuds 4 et 5 sous la racine 0.

Complexité On obtient ainsi une structure de donnée dont la *complexité amortie* des opérations est en $\mathcal{O}(\alpha(n))$ (la *réciproque* de la *fonction d'Ackermann*[2]), c'est-à-dire **constante en pratique** pour une

2. Wilhelm Ackermann, étudiant de Hilbert, publie en 1928 l'un des premiers exemples de fonction récursive *non récursive primitive* (en particulier, une fonction primitive récursive ne peut pas être « doublement récursive ») :

$$A(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m-1, A(m,n-1)) & \text{sinon} \end{cases}$$

séquence d'opérations sur toute taille de données réaliste – ce qui est tout à fait remarquable !

1. Pour implémenter efficacement la structure Union-Find, on va définir dans `unionfind.py` une classe de nœud `Node` avec deux attributs :
   — `pred` l'identifiant (`id`) du parent d'un nœud, initialisé à None ;
   — `r` le rang de l'arbre, initialisé à 0.
   Définir également la méthode spéciale `__repr__` pour faciliter l'affichage lors de la mise au point.
2. Définir une classe `UFind` pour la structure Union-Find avec deux attributs :
   — `nodes` le dictionnaire des nœuds qui associe l'identifiant d'un sommet à un objet de la classe `Node`, initialisé au dictionnaire vide ;
   — `size` le nombre de composantes connexes, initialisé à zéro.
3. Définir une méthode `add(id)` de la classe `UFind` qui prend l'identifiant d'un sommet en paramètre, ajoute une nouvelle association au dictionnaire `nodes` entre l'identifiant (clé) et un nouveau nœud de la classe `Node`, et incrémente la taille `size`.
4. Définir une méthode `find(id)` qui prend l'identifiant d'un sommet en paramètre et renvoie celui de sa racine (i.e. l'élément représentatif de la classe). Pour réaliser cette opération, il suffit de tester le parent du nœud : s'il est égal à `None`, on est à la racine et on peut renvoyer `id` ; sinon, on appelle `find` récursivement sur le parent.

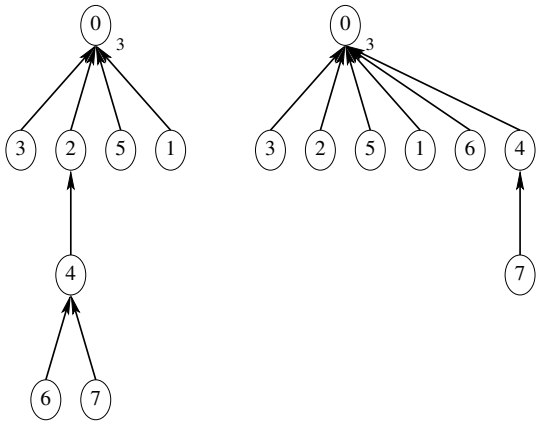


Figure 3: Compression de chemin après un appel à `find(6)` : les nœuds 6 et le sous-arbre contenant 4 et 7 remontent ainsi directement sous la racine.

5. Pour améliorer la complexité amortie de la structure Union-Find, on peut réaliser une **compression de chemin** durant la recherche de la racine par la méthode `find` : à chaque étape, au lieu de renvoyer directement la racine (`return self.find(...)`), on la retient dans une variable et on remplace le parent du nœud courant par cette racine, tel qu'illustré figure 3. Ainsi, après une première recherche `find`, toutes les suivantes sur l'un des sommets de la branche qui mène au nœud prendront un temps constant puisque les sommets se trouveront directement sous la racine.

La croissance de A est extrêmement rapide : notamment $f(n) = A(n, n)$ croît plus vite que les fonctions exponentielles, $A(4, 2)$ est un nombre de 19729 chiffres et $A(4, n) = 2 \uparrow\uparrow (n + 3) - 3 = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+3} - 3$.

Ce type de fonction a permis de prouver que l'un des premiers modèles de calcul supposé formaliser la notion de *fonction calculable*, les fonctions *primitives récursives*, n'était pas aussi puissant que celui de la *machine de Turing*. Cette fonction est donc fondamentale en *calculabilité*, permet d'exprimer certains résultats de complexité et est utilisée pour tester les performances des langages de programmation.

6. Définir une méthode `union(id1, id2)` qui prend deux identifiants de racines (i.e. deux éléments représentatifs), fusionne les composantes associées et décrémente la taille `size`.
7. On va utiliser les classes de sommet `Vertex` et de graphe non orienté pondéré `WGraph` définies dans le fichier `tp4.py` fourni. Le graphe sera représenté par des listes d'adjacence : à chaque sommet `Vertex` est associé un attribut `adj` qui est le dictionnaire dont les clés sont les sommets adjacents et les valeurs la pondération associée. Pour construire le graphe, la méthode `add_edge(u,v,w)` permet d'ajouter une arête pondérée même si les sommets n'ont pas encore été ajoutés. Pour obtenir la liste des arêtes pondérées, on peut utiliser la méthode `edges`.

   Définir une nouvelle méthode `kruskal` de la classe `WGraph` qui renvoie les arêtes d'un arbre couvrant minimal en utilisant l'algorithme de Kruskal. On commencera par initialiser la structure Union-Find avec une composante pour chaque nœud de V, à trier les arêtes par poids croissant, puis à initialiser les arêtes de l'arbre avec la liste vide. Ensuite, lors du parcours des arêtes, si les composantes connexes associées aux extrémités de l'arête considérée sont différentes, il faut les fusionner et ajouter l'arête à l'arbre. On terminera par renvoyer l'arbre lorsqu'on aura ajouté $|V| - 1$ arêtes ou qu'il ne reste plus qu'une seule composante connexe dans la structure Union-Find.
8. Quelle est la complexité de l'algorithme de Kruskal ? Mesurer le temps de calcul pour quelques graphes à n sommets et m arêtes pondérées dans l'intervalle $[1, w]$ générés aléatoirement grâce à la méthode `rand(n, m, w)` de la classe `WGraph`.