TP n° 5 – Arbres couvrants minimaux aussi

vie@jill-jenn.net/tp
thomas.h@ens.fr

1 Oups !

Yorel Reivax ne trouve plus son chat, et décide d'exécuter le code suivant pour en savoir plus :

```
if false then
  if true then print_string "Le chat est vivant."
else
  print_string "Le chat est mort."
;;
```

Alors ? Bref, n'oubliez pas : chaque fois que vous oubliez une parenthèse, Dieu ne tue pas un chat.

2 Union-find

2.1 Préliminaires

On cherche à partitionner $\{0, \ldots, n-1\}$. Pour ce faire, nous requérons une structure qui nous permette efficacement de :

- **trouver** la partition à laquelle appartient un élément ;
- **unir** deux partitions.

Nous allons considérer une forêt d'arbres enracinés [1]. Chaque arbre représentera une partition [2], et il est naturel d'unir deux partitions en accrochant la racine d'un arbre à la racine de l'autre.

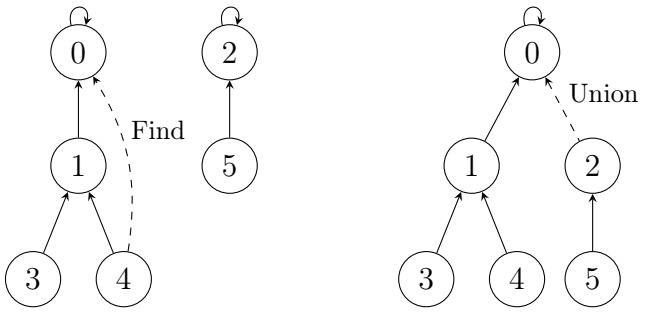

FIGURE 1 – À gauche, une représentation des partitions {0, 1, 3, 4} et {2, 5}. Le père de 4 est 1, et 0 et 2 sont les *représentants* de chaque partition (ils sont en particulier leurs propres pères). À droite, leur union.

Il convient de représenter cette structure en mémoire par un tableau `père` de n éléments, où `père.(i)` est le numéro du père de `i`.

Question 1. Quelle est la complexité des opérations **trouver** et **unir** dans le pire cas ?

On souhaite donc améliorer l'efficacité de notre structure en s'aidant de deux astuces :

- la *compression de chemin*, le concept d'aplatir la structure d'arbre au fur et à mesure qu'on cherche le représentant d'un sommet, en reliant tous les sommets parcourus à la racine ;
- le *rang* d'un sommet : une estimation de la hauteur de l'arbre enraciné en ce sommet.

```
type unionfind = {père : int vect, rang : int vect};;
```

1. Un arbre enraciné est un arbre simplement connexe tel que tous les nœuds ont un unique parent.
2. Deux éléments seront dans le même arbre si et seulement s'ils sont dans la même partition.

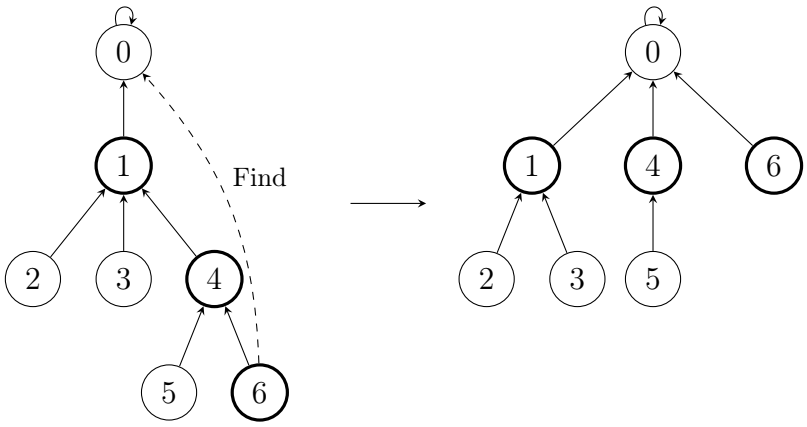

FIGURE 2 – La compression de chemin en images.

Question 2. Écrire une fonction `initialise n` qui retourne le tableau initial `père` où chaque élément est son propre père.

Question 3. Écrire une fonction `trouver` telle que `trouver uf i` retourne le représentant de la classe de `i` dans la structure `uf`, en effectuant la compression de chemin et en mettant à jour les rangs des sommets parcourus.

Question 4. Écrire une fonction `unir` telle que `unir père rang i j` réunit [3] les partitions de `i` et de `j`. Distinguer les cas selon si le rang du représentant de `i` est plus petit, plus grand ou égal au rang du représentant de `j`, pour que votre fonction soit la plus efficace possible.

Question 5. Quelle est la complexité des fonctions `unir` et `trouver` ?

Question 6. Écrire une fonction qui prend en argument un graphe sous forme de listes d'adjacence et retourne son nombre de composantes connexes. Allez, c'est bientôt Noël, on vous autorise à utiliser une référence pour le compteur.

2.2 Algorithme de Kruskal

On considère un graphe non orienté G connexe et pondéré à poids positifs.

Un *arbre couvrant* du graphe $G = (V, \to)$ est un arbre $\mathcal{A} = (V, \to')$ connexe tel que $\to' \subset \to$. L'objectif de ce TP est aussi de trouver un arbre couvrant minimal (soit, de poids minimum) de G.

L'algorithme de Kruskal construit un arbre couvrant minimal en procédant de la façon suivante :
- On trie les arêtes par poids croissant.
- On considère chaque sommet comme un arbre à un élément.
- Si une arête relie deux arbres distincts de la forêt, on ajoute cette arête à $\mathcal{A}$ et on fusionne les deux arbres, sinon on la jette.

Question 7. Montrer que l'algorithme de Kruskal termine.

Question 8. Écrire une fonction qui détermine le nombre de sommets n d'un graphe connexe à partir de la liste de ses arêtes, de type `int * int * int list`.

Question 9. Implémenter l'algorithme de Kruskal à l'aide d'une structure union-find. Quelle est sa complexité ?

3. Sauf si c'est la même, ne soyez pas stupide.