TP n° 5 – Union-find

etienne.jl.simon@gmail.com thomas.h@ens.fr
thomash.fr/teaching/

1 Graphes

Un graphe $G = (V, E)$ est défini par l'ensemble $V = \{0, ..., n-1\}$ de ses sommets et la relation d'adjacence $E \subset V^2$. On parle de graphe non orienté si la relation E est symétrique. Un graphe pondéré est un graphe $G = (V, E)$ muni d'une fonction de pondération $p : E \to R$ qui donne un poids à chaque arête.

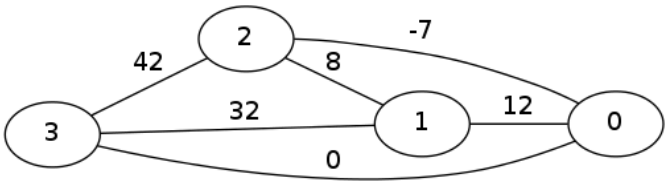

FIGURE 1 – Exemple de graphe pondéré

Un chemin de a à b est une suite de sommets $s_0, ..., s_r$ telle que $a = s_0$, $b = s_r$ et pour tout $0 \leq i < r$, $(s_i, s_{i+1}) \in E$. Un graphe est connexe si pour tout couple de sommets, il existe un chemin allant de l'un à l'autre. C'est un arbre s'il est connexe et ne contient pas de cycle (chemin d'un sommet à lui-même emprutant des arêtes distinctes).

Il existe différentes façon de représenter des graphes en mémoire. La façon la plus proche de la définition mathématique est d'utiliser une liste d'arêtes (couples de sommets ou triplets dans le cas de graphes pondérés).

```
let example_edges = 4, [1, 2, 8; 1, 3, 32; 1, 0, 12; 2, 3, 42; 2, 0, -7; 3, 0, 0]
```

Cependant cette représentation est peu efficace ($O(|E|)$ pour accéder à une arête arbitraire). Une représentation plus efficace est la matrice d'adjacence, une matrice carré A de taille $n = |V|$ telle que $A_{i,j} = p(i, j)$.

```
let example_adj_mat = [|[|None ; Some 12; Some (-7); Some 0|];
                       [|Some 12; None ; Some 8; Some 32|];
                       [|Some (-7); Some 8; None ; Some 42|];
                       [|Some 0; Some 32; Some 42; None|]|]
```

Cette représentation est beaucoup plus performante puisqu'elle permet d'accéder à une arête arbitraire en $O(1)$. Mais la matrice d'adjacence a un inconvénient : sa taille. En effet si le graphe est peu dense, seule une faible partie de la matrice sera utile. Pour résoudre ce problème, on utilise des listes d'adjacences : pour chaque sommet, on stocke l'ensemble des arêtes sortantes dans une liste (on a donc un tableau de listes d'arêtes).

```
let example_adj_list = [|[1, 12; 2, -7; 3, 0];
                         [0, 12; 2, 8; 3, 32];
                         [0, -7; 1, 8; 3, 42];
                         [0, 0; 1, 32; 2, 42]|]
```

Question 1. Écrire les fonctions suivantes :

```
edges_to_adj_mat  : int * (int * int * int) list -> int option array array
adj_mat_to_adj_list  : int option array array -> (int * int) list array
adj_list_to_edges  : (int * int) list array -> int * (int * int * int) list
```

2 Union-find

La structure communément appelée union-find permet de représenter une partition de $\{0, ..., n-1\}$ en supportant l'opération *find* qui donne pour un élément donné le représentant de sa classe et l'opération *union* qui fusionne les classes de deux éléments.

Une partition est représentée par une forêt où chaque arbre est une classe de la partition et la racine est le représentant de la classe. On utilise pour cela un tableau de taille n où la case i contient le parent de i (ou i si c'est la racine). Ainsi pour trouver le représentant de la classe d'un élément, il suffit de remonter jusqu'à la racine de l'arbre et pour fusionner deux classes il suffit de brancher l'arbre d'une classe à la racine de l'autre arbre.

Afin d'obtenir une bonne complexité, on veut que la hauteur des arbres soit la plus faible possible. Pour cela on utilise deux heuristiques :

- La compression de chemin : Lorsqu'on trouve le représentant de la classe d'un élément, on branche cet élément directement à la racine.
- Le rang : À chaque élément on associe un rang qui est un majorant de la hauteur de l'arbre enraciné en cet élément et lorsqu'on fusionne deux arbres, on branche la racine de rang inférieur sur celle de rang supérieur. Le rang d'un sommet seul est 0 et ce rang est augmenté de 1 lorsqu'on fusionne deux arbres de même rang.

Une partition sera donc représentée par le type `(int * int) array` où le premier élément de la case i du tableau est le parent de i et le second élément son rang.

Question 2. Écrire la fonction `init : int -> (int * int) array` qui étant donné un entier n construit la partition de $\{0, ..., n-1\}$ en singletons.

Question 3. Écrire la fonction `find : (int * int) array -> int -> int`.

Question 4. * Écrire la fonction `find_tr : (int * int) array -> int -> int` qui fait la même chose que `find` mais est récursive terminale.

Question 5. Écrire la fonction `union : (int * int) array -> int -> int -> bool` qui renvoie `true` si les deux éléments sont déjà dans la même classe et `false` sinon.

Question 6. Utiliser union-find pour écrire une fonction `cc : int * (int * int * int) list -> (int * int) array` (on utilise la représentation par liste d'arêtes) qui calcule les composantes connexes d'un graphe.

3 Algorithme de Kruskal

Un arbre couvrant minimal d'un graphe $G = (V, E)$ connexe pondéré à poids positifs est un arbre $A = (V, E')$ avec $E' \subset E$ de poids minimal.

L'algorithme de Kruskal calcule un arbre couvrant minimal de la façon suivante :

- On trie les arêtes par poids croissants.
- On initialise l'arbre couvrant avec aucune arête.
- Pour chaque arête du graphe, si l'ajouter à l'arbre couvrant ne crée pas de cycle (*i.e.* si les deux sommets de l'arête sont dans des somposantes connexes distinctes) alors on l'ajoute.

Question 7. Montrer que l'algorithme de Kruskal est correct.

Question 8. Écrire la fonction `kruskal : int * (int * int * int) list -> int * (int * int * int) list` qui implémente l'algorithme de Kruskal.