TP: Arbre couvrant minimal, structure d'union-set

1 Structure efficace d'Union-Set

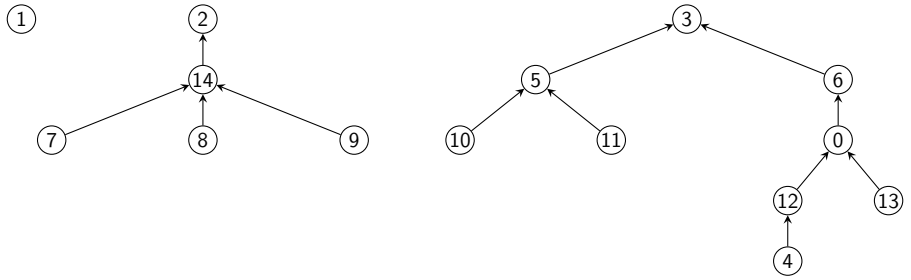
On s'intéresse ici à une structure spéciale sur les éléments de $[\![0, n-1]\!]$: on veut gérer une *partition* de cet ensemble. Les opérations à écrire sont :

— `creer n` : renvoie une structure `u` où chaque élément de $[\![0, n-1]\!]$ est seul dans sa partie.

— `rep u x` : avec `u` la structure, renvoie le représentant de $x \in [\![0, n-1]\!]$ (deux éléments ont même représentant si et seulement si ils sont dans la même partie. À la création, le représentant de x est x lui-même)

— `fusion u x y` : *fusionne* les deux parties de $[\![0, n-1]\!]$ contenant x et y, supposés appartenir à deux parties différentes.

Il est très facile d'écrire une structure basique d'union-set où chaque opération a un coût $O(n)$, un tableau stockant pour chaque élément son représentant suffit. Néanmoins, on peut faire beaucoup mieux en utilisant une structure arborescente. L'ensemble $[\![0, n-1]\!]$ est vu comme une forêt, chaque arbre étant associé à une partie. La racine d'un arbre est le représentant de la partie.



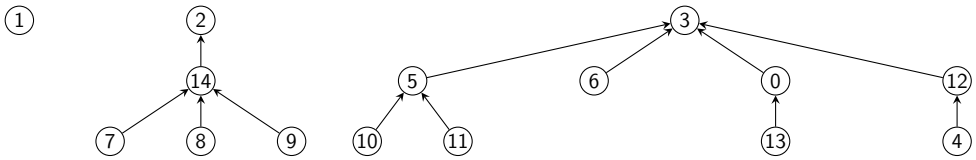
Une telle forêt se représente facilement : un tableau `p` de taille n convient, avec `p.(i)` le parent de `i`, sauf si `i` est la racine de son arbre et dans ce cas `p.(i)` = `i`. Avec une telle structure, les trois opérations sont faciles à écrire :

— pour trouver le représentant d'un élément, on remonte en « suivant les flêches » jusqu'à trouver la racine de l'arbre.

— pour fusionner les parties (disjointes) contenant x et y, il suffit de calculer les représentants r_x et r_y de x et y, et de modifier le tableau pour que le parent de r_x soit r_y, ou l'inverse.

Pour optimiser le temps d'exécution des futures opérations, on utilise en général deux heuristiques simples :

— *la compression de chemin* consiste, lorsqu'on calcule le représentant r_x de x, à raccorder tous les éléments du chemin entre r_x et x directement à r_x. Voici par exemple la forêt obtenue après calcul du représentant de 12 (qui est 3) sur la forêt précédente :



Cette optimisation ne requiert pas de donnée supplémentaire.

— *l'union par rang* consiste, lorsqu'on fusionne deux parties de représentants r_x et r_y, à essayer de déclarer r_x comme fils de r_y si l'arbre issu de r_x a une hauteur plus petite que l'arbre issu de r_y plutôt que l'inverse. Par exemple, pour fusionner les parties de représentants 1 et 3 dans l'exemple précédent, mieux vaut déclarer 1 comme fils de 3 que l'inverse. Pour cela, on utilise un tableau de taille n en plus de `p` : le tableau `r` des *rangs*. Initialement, tous les rangs sont nuls. Lorsqu'on veut fusionner deux parties de représentants r_x et r_y, trois cas peuvent se produire (on note $R(a)$ le rang de a) :

  — $R(r_x) < R(r_y)$: on déclare r_x comme fils de r_y, le rang de r_y est inchangé ;

  — $R(r_y) < R(r_x)$: c'est l'inverse, r_y est déclaré comme fils de r_x, et le rang de r_x est inchangé.

— $R(r_x) = R(r_y)$: on déclare arbitairement r_y comme fils de r_x, et dans ce cas **le rang de r_x est augmenté de 1**.

Notons que le rang $R(i)$ d'un élément i n'est réellement utile que lorsque i est le représentant de sa partie.

On peut montrer qu'avec ces optimisations, la structure d'Union-Set possède, pour chaque opération de fusion ou de calcul de représentant, une complexité *amortie* $O(\alpha(n))$, où α est une fonction de croissance extrêmement lente [1], tellement faible qu'on la considère comme quasi-constante : $\alpha(n) \leq 4$ pour toute valeur de n sensée !

Travail demandé : implémenter la structure d'Union-Set avec les optimisations. La structure sera représentée par un couple `(p,r)`, comme décrit ci-dessus.

Test. Exécuter le code suivant (la fonction est présente dans l'annexe sur mon site). Vérifier que vous obtenez comme moi !

```
# petit_test () ;;
026226215226291022221521591915152222252615289152151522 2- : unit = ()
```

2 Algorithme de Kruskall et Project Euler 107

On rappelle l'algorithme de Kruskall pour calculer un arbre couvrant minimal d'un graphe non orienté connexe : trier les arêtes du graphe par poids croissant, les considérer une à une, et rajouter à l'arbre couvrant minimal en construction toutes celles qui ne créent pas de cycle (de manière équivalente, ce sont celles qui réduisent le nombre de composantes connexes). On travaille ici avec des graphes dont toutes les arêtes ont un poids strictement positif, représentés par leur matrice d'adjacence $M = (m_{i,j})$. On convient que $m_{i,j} = 0$ s'il n'y a pas d'arête entre i et j.

Travail demandé. Écrire une fonction `kruskall m` prenant en entrée une telle matrice d'adjacence, et renvoyant la liste des arêtes formant un ACM (de la forme $(i, j, \omega(i, j))$, avec $i < j$). Indications :

— extraire les arêtes du graphe : attention à ne considérer que les entrées non nulles de la matrice, et que celles pour lesquelles $i < j$.

— pour le tri des arêtes dans l'ordre croissant, utiliser `List.sort : ('a -> 'a -> int) -> 'a list -> 'a list`. L'appel `List.sort f q` renvoie une liste contenant les mêmes éléments que `q`, telle que `f x y` est négatif si `x` est placé avant `y`.

— utiliser la structure d'union set pour l'algorithme de Kruskall : on rajoute une arête dans l'ACM si et seulement si les sommets associés sont situés dans des parties différentes. Ne pas oublier de fusionner les parties !

Tester avec la petite matrice fournie :

```
# kruskall m7 ;;
- : (int * int * int) list =
[(4, 6, 11); (0, 2, 12); (0, 1, 16); (1, 3, 17); (3, 4, 18); (3, 5, 19)]
```

Application au Project Euler 107. Dans ce problème, on se demande quelles arêtes on peut supprimer d'un graphe G connexe, pour obtenir un graphe G' sans perdre la connexité. On considère que le coût économisé est la somme des poids des arêtes **qui sont dans G mais pas dans G'**. Déduire de la fonction précédente une fonction `pb_107 m` renvoyant le coût maximal que l'on peut économiser. L'appliquer à la matrice `m40`.

```
# pb_107 m7 ;;
- : int = 150
# pb_107 m40 ;;
- : int = ... (* reponse au problème 107 ! *)
```

3 Utilisation de la structure d'union set et Project Euler 186

Travail demandé : résoudre le Project Euler 186. Remarque : il faut moins de trois millions d'appels, donc six millions de termes de la suite S suffisent, ce qui tient largement en mémoire (même si on aurait pu faire plus économique en mémoire : stocker seulement 55 termes simultanément dans un tableau circulaire est possible).

1. La réciproque généralisée de la fonction $n \mapsto A(n, n)$, où A est la fonction d'Ackermann `https://fr.wikipedia.org/wiki/Fonction_d%27Ackermann`.