Union-Find

Feuille n°7

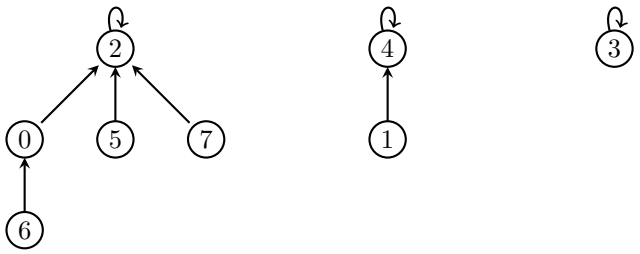
Dans ce TP, on va étudier l'algorithme Union-Find, qui est l'algorithme le plus efficace à ce jour pour manipuler les partitions d'ensembles.
On écrira d'abord une version naïve de l'algorithme Union-Find. Ensuite, on écrira une deuxième version plus avancée, qui met en place deux améliorations : la *compression des chemins* et la *fusion par rang*. Enfin, on comparera la différence d'efficacité entre les deux versions en comptant le nombre d'accès à la structure de données.

Soit N un entier. Dans la suite, on va manipuler des partitions de l'ensemble $\{0, \ldots, N-1\}$. Une partition est une famille de sous-ensembles $\mathbf{E}_0, \ldots, \mathbf{E}_k$ de l'ensemble $\{0, \ldots, N-1\}$, telle que tout entier compris entre 0 et $N-1$ appartient à un et un seul ensemble $\mathbf{E}_i$. On appellera ces sous-ensembles des *classes* pour aider à la compréhension.

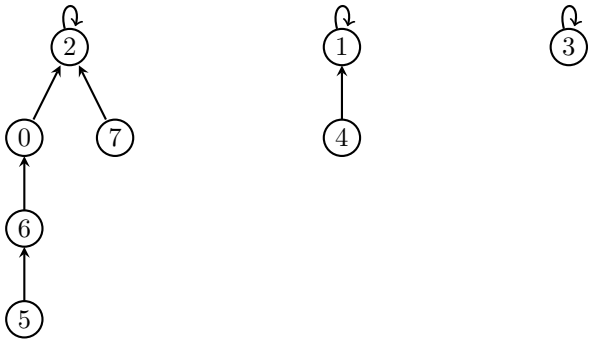
Exemple 1 : Les trois classes $E_0 = \{0, 2, 5, 6, 7\}$, $E_1 = \{1, 4\}$ et $E_2 = \{3\}$ forment une partition de l'ensemble $\{0, 1, 2, 3, 4, 5, 6, 7\}$; dans ce cas, $N = 8$.

Pour représenter la partition, on fait correspondre à chaque classe un arbre qui utilise la relation *avoir pour parent* (au contraire des TPs précédents, où on utilise plutôt des relations *avoir pour fils*). Deux entiers appartiennent à la même classe si et seulement si ils sont dans le même arbre, c'est-à-dire s'ils ont un ancêtre commun. Pour des raisons de commodité, on fera en sorte que la racine de chaque arbre soit son propre père. On appelle la racine de l'arbre, le *représentant* de la classe correspondante.

Exemple 1 (suite) : Ci dessous se trouvent des arbres correspondant à la partition E_0, E_1, E_2. L'arbre de gauche correspond à la classe $E_0 = \{0, 2, 5, 6, 7\}$, dont le représentant est donc 2 ; l'arbre du milieu correspond à $E_1 = \{1, 4\}$, dont le représentant est donc 4 ; l'arbre de droite correspond à $E_2 = \{3\}$.



Exemple 1 (fin) : D'autres arbres pourraient aussi représenter la même partition, comme par exemple ceux ci-dessous. On remarquera que c'est maintenant 1 qui est le représentant de la classe $\{1, 4\}$.



En termes d'implémentation pratique, et contrairement aux TP précédents, le nombre total de nœuds de nos différents arbres est constant. Au lieu d'utiliser les structures de `Noeud` utilisées jusqu'alors, on utilisera directement la structure de données suivante :

```
typedef struct {
  int taille;        // Taille des tableaux
  int *pere;         // Tableau des peres
  int *rang;         // Tableau des rangs
  int nb_appels;
} UnionFind;
```

Notre partition sera donc représentée en mémoire par deux tableaux de la même longueur, regroupé dans la structure `UnionFind` ci-dessus. Dans l'exemple qui suit, on utilise une variable `partition` de type `UnionFind*`. Dans ce cas, `partition->taille` est la taille des tableaux `partition->pere` et `partition->rang`. De plus, l'entier `partition->pere[i]` est l'indice du parent de l'élément `i`. Enfin :

— si l'entier `i` est le représentant de l'ensemble auquel il appartient, alors `partition->pere[i]` vaut `i` et `partition->rang[i]` est le rang de l'arbre enraciné en `i` ;

— sinon, l'entier `partition->rang[i]` n'a aucun sens : on ne mettra pas à jour et on ne devra jamais y accéder.

Par ailleurs, et afin de mieux contrôler le nombre d'appels de fonction effectués, on augmentera la valeur de `partition->nb_appels` d'une unité à chaque fois que l'on accède à un entier `partition->pere[i]`.

Exemple 2 : La partition $\{\{0,2\},\{3\},\{1,4,5\}\}$ peut être représentée par le `UnionFind` suivant. Les `?` représentent des entiers dont la valeur n'a aucune importance.

`taille: 6` `pere:`

0	1	0	3	5	1

`rang:`

2	3	?	1	?	?

Les tableaux ci-dessus se lisent comme suit :

— Le père de 0 est la valeur de `pere[0]`, c'est-à-dire 0 ; le noeud 0 est donc une racine.
— Le père de 1 est la valeur de `pere[1]`, c'est-à-dire 1 ; le noeud 1 est donc une racine.
— Le père de 2 est la valeur de `pere[2]`, c'est-à-dire 0.
— Le père de 3 est la valeur de `pere[3]`, c'est-à-dire 3 ; le noeud 3 est donc une racine
— Le père de 4 est la valeur de `pere[4]`, c'est-à-dire 5.
— Le père de 5 est la valeur de `pere[5]`, c'est-à-dire 1.

— Le rang[1] de l'arbre enraciné en 0 est la valeur de `rang[0]`, c'est-à-dire 2 ;
— Le rang[1] de l'arbre enraciné en 1 est la valeur de `rang[1]`, c'est-à-dire 3 ;
— Le rang[1] de l'arbre enraciné en 3 est la valeur de `rang[3]`, c'est-à-dire 1 ;
— Les valeurs `rang[2]`, `rang[4]` et `rang[5]` n'ont aucun sens car 2, 4 et 5 ne sont pas des racines.

Exemple 3 : La même partition peut-être aussi représentée par d'autres `UnionFind`, par exemple, par l'`UnionFind` suivant, dans lequel le représentant de l'ensemble $\{1,4,5\}$ est 4 plutôt que 1.

`taille: 6` `pere:`

0	4	0	3	4	4

`rang:`

2	?	?	1	2	?

0. **Sortir un papier et un crayon** — Si vous ne réussissez pas cet exercice, votre enseignant ne viendra pas vous aider en cas de problème lors d'une des questions qui suivent.
1. Dessiner les arbres représentés par chacun des deux `UnionFind` donnés dans les exemples ci-dessus. Il doit y avoir un arbre par `UnionFind` et par classe dans la partition, donc 6 arbres au total.

1. Dans ces exemples, le rang est égal à la hauteur. A partir du moment où l'on compresse les chemins, le rang pourra être plus grand que la hauteur.

2. Écrire une fonction `UnionFind *initialisation(int N)` qui alloue et initialise un `UnionFind` de façon à ce qu'il représente la partition triviale des entiers de 0 à `N` − 1., c'est-à-dire qui comprend `N` ensembles réduits à un éléments : $\{0\}, \{1\}, ...\{N-1\}$.
3. Écrire une fonction `int trouve_naif(UnionFind *partition, int x)` qui renvoie la valeur du représentant de l'ensemble contenant l'entier `x`. On n'oubliera pas d'augmenter la valeur de `nb_appels` dès que nécessaire.
4. Écrire une fonction `void fusion_naive(UnionFind *partition, int x, int y)` qui effectue la fusion des ensembles contenant `x` et `y`. Si `x` et `y` ne sont pas dans le même ensemble, le représentant de l'ensemble contenant `x` deviendra le nouveau représentant de l'ensemble fusionné.

   Il est conseillé de tester les fonctions `trouve_naif` et `fusion_naive` dans une fonction `void test_naive()`. Il pourra être utile d'écrire une fonction qui affiche un `UnionFind*`.
5. Écrire une fonction `int trouve_compresse(UnionFind *partition, int x)` qui renvoie la valeur du représentant de l'ensemble contenant l'entier `x`. Dans cette fonction, on n'oubliera pas d'augmenter la valeur de `nb_appels` dès que nécessaire, et on prendra garde de **toujours** compresser les chemins ainsi identifiés.
6. Écrire une fonction `void fusion_rang(UnionFind *partition, int x, int y)` qui effectue la fusion des ensembles contenant `x` et `y`. Si `x` et `y` ne sont pas dans le même ensemble, le représentant de l'ensemble de plus grand rang deviendra celui de l'ensemble fusionné (en cas d'égalité, le représentant de l'ensemble contenant `x` deviendra le représentant de l'ensemble fusionné). On n'oubliera pas de mettre à jour, si nécessaire, le rang de l'ensemble fusionné.

   Il est conseillé de tester les fonctions `trouve_compresse` et `fusion_rang` dans une fonction `void test_avance()`.
7. Écrire la fonction `int main(void)` permettant de tester l'efficacité de l'amélioration apportée par `trouve_compresse` et `fusion_rang`. On pourra choisir un entier `M` puis effectuer `M` appels aléatoires en utilisant

   a. les fonctions `trouve_naif` et `fusion_naif` ;

   b. les fonctions `trouve_compresse` et `fusion_rang`.

   On prendra garde à effectuer les mêmes appels dans les deux cas, mais à bien réinitialiser la structure de donnée entre les deux.

   Les valeurs des entiers manipulés seront obtenues à l'aide des fonctions `void srandom(unsigned int seed)` et `long int random()`. On pourra tracer des graphiques à l'aide de `gnuplot`.