Chapitre 3 : Structure de donnée Union Find (unir et trouver)

03 octobre

1 Structure abstraite

On va ici présenter une structure permettant de représenter une partition d'un ensemble fini.

Soit E un ensemble fini, une partition de E est une famille $\{A_i\}_{i\in I}$ de parties de E deux à deux disjointes dont la réunion est E. Ces parties sont appelées classes car l'on sait que ce sont les classes d'équivalence d'une certaine relation d'équivalence.

On va définir une représentation des partitions de l'ensemble $\{0,\dots,n-1\}$ (tout ensemble fini de cardinal n peut être mis en bijection avec cet ensemble) muni des opérations suivantes :

1. Création d'une nouvelle partition de taille n où chaque élément est dans sa propre classe : `create : int -> partition`
2. Union des classes de deux éléments de E passés en argument `union : partition, int, int -> void`
3. Recherche de la classe associée à un élément de E `find : partition, int-> int`

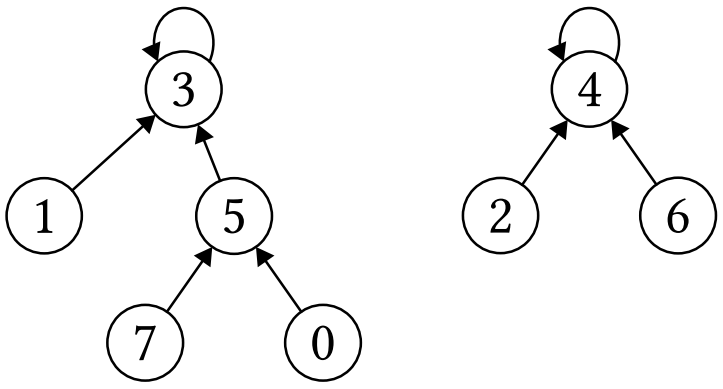
Un appel à `create` sur l'entrée `n` construit une nouvelle partition de $\{0,1,\dots,n-1\}$ où chaque élément forme une classe à lui tout seul.

L'opération `find` sur une partition et un entier `i` détermine la classe de l'élément `i`, sous la forme d'un entier considéré comme l'unique représentant de cette classe. En particulier, on détermine si deux éléments sont dans la même classe en comparant les résultats donnés par `find` pour chacun.

Enfin, l'opération `union` réunit les deux classes des éléments passés en argument, la structure de données étant modifiée en place.

2 Une première implémentation

L'idée principale est de lier entre eux les éléments d'une même classe. Dans chaque classe, ces liaisons forment des chemins qui mènent tous à un unique représentant, qui est le seul élément lié à lui-même. La figure ci-dessous montre un exemple où l'ensemble $\{0,1,\dots,7\}$ est partitionné en deux classes dont les représentants sont respectivement 3 et 4.



Il est facile de matérialiser ces relations par un simple tableau qui relie chaque entier à un autre entier de la même classe. Ces liaisons mènent toujours au représentant de la classe, qui est associé à sa propre valeur dans le tableau. Ainsi, la partition de la figure est représentée par le tableau suivant : `{5,3,4,3,4,3,4,5}`.

Réalisons alors notre structure en OCAML. On considère la variable `uf` qui est un tableau d'entiers représentant une partition comme décrit ci-dessus.

```
let create n = Array.init n (fun i->i);;

let rec find uf i =
   if uf.(i)=i then i
   else



let union uf i j =
```

On remarque que si `i` et `j` étaient dans la même classe alors l'union est sans effet.

On pouvait difficilement imaginer un code plus simple que cela. Cependant, notre structure est un peu naive. En effet, on peut se retrouver avec de très longues chaînes dans le tableau, voire impliquant les `n` éléments. C'est le cas par exemple si on fait `union i (i+1)` pour tout `i`. Dès lors la complexité de `find` et donc de `union` peuvent être aussi grandes que $O(n)$. Ce n'est pas acceptable en pratique. Fort heureusement, il est facile d'atteindre de bien meilleurs performances, en apportant deux améliorations à notre code.

Illustrer l'évolution de la partition avec 4 éléments initialement isolés si on effectue `union i (i+1)` pour tout `i` :

Illustrer l'évolution de la partition avec 4 éléments initialement isolés si on effectue `union (i+1) i` pour tout `i` :

3 Optimisation 1 : union pondérée

On remarque que lors de l'union, on fait pointer le représentant d'une des deux classes vers le représentant de l'autre. Le choix des rôles n'est pas arbitraire puisqu'une classe peut contenir des chemins beaucoup plus longs que l'autre. Ainsi, on va maintenir, pour chaque représentant, une valeur appelée rang qui représente la longueur maximale que pourrait avoir un chemin dans cette classe. Cette information est stockée dans un second tableau, à côté du tableau qui contient les liaisons.

```
type uf = {
link: int array; rank: int array;
}
```

L'information contenue dans `rank` n'est significative que pour des éléments `i` qui sont des représentants, c'est-à-dire pour lesquels `link.(i)= i`. Initialement, le rang de chaque classe vaut 0.

```
let create n =
{ link = Array.init n (fun i -> i);
      rank = Array.make n 0; }
```

Le rang est ensuite utilisé par la fonction `union` pour choisir entre les deux représentants possibles d'une union.

```
let union uf i j =
   let ri = find uf i in
   let rj = find uf j in
   if (ri<>rj) then
      begin
      if (uf.rank.(ri) < uf.rank.(rj)) then


      else if (uf.rank.(ri) > uf.rank.(rj)) then


      else



      end;;
```

Cette optimisation est déjà intéressante :

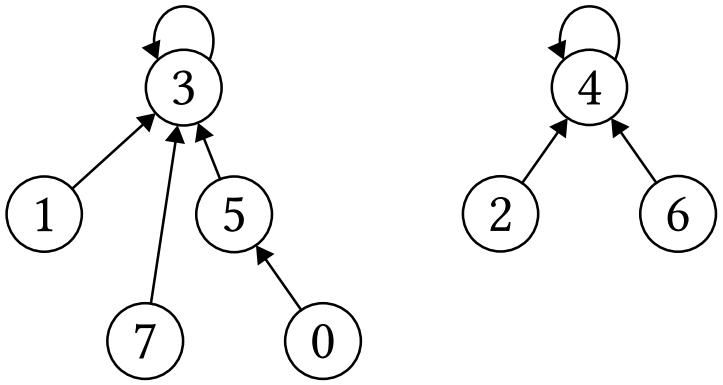
Théorème 1. *Une classe de rang k possède des chemins de longueur maximale k et au moins 2^k éléments.*

On fait la preuve par récurrence sur le nombre d'appels à union.

On en conclut que grâce à cette première optimisation, on obtient des opérations `find` et `union` en $O(\log(n))$.

4 Optimisation 2 : compression des chemins

On peut cependant améliorer encore l'efficacité de notre structure. L'idée consiste à compresser les chemins pendant la recherche effectuée par `find` : on tire profit du parcours de la branche pour compresser le chemin et ainsi obtenir plus tard de meilleures complexités. Ainsi, on lie directement au représentant de la classe tous les éléments rencontrés sur le chemin parcouru pour l'atteindre lors de l'appel à `find`. Par exemple, un appel à `find 7` dans la situation de la figure précédente va renvoyer `3` et modifier la structure pour que `7` (rencontré sur le chemin) pointe désormais directement sur `3` :



On obtient alors la nouvelle fonction `find` suivante :

```
let rec find uf i =
   let p = uf.link.(i) in
   if (p=i) then i
   else
      let r = find uf p in
```

En particulier, après le calcul de `r = find uf i`, on a `uf.link.(i) = r` directement. Mais on a également `link.(j) = r` pour tous les éléments `j` qui se trouvaient initialement sur le chemin entre `i` et `r` grâce aux appels récursifs.

Il est important de noter que la fonction `union` utilise la fonction `find` et réalise donc des compressions de chemin, même dans le cas où il s'avère que `i` et `j` sont déjà dans la même classe. Le théorème 1 reste valable si on entend " de longueur maximale `k` " comme " de longueur au plus `k` " . En effet, une classe peut avoir le rang `k` mais des chemins de fait tous strictement plus petits que `k` car la compression de chemin les a tous raccourcis. En particulier, on a donc toujours une complexité au pire logarithmique pour les opérations `find` et `union`.

La complexité est en réalité bien meilleure. On peut montrer que la complexité amortie de chaque opération est $O(\alpha(n))$, où α est l'inverse de la fonction d'Ackermann. Cette fonction croît si lentement qu'on peut la considérer comme constante pour toute application pratique — vues les valeurs de `n` que les limites de mémoire nous autorisent à admettre — ce qui nous permet de supposer un temps amorti constant pour chaque opération. Cette analyse de complexité est subtile et dépasse largement le cadre du programme.

Exercice : Implémenter la structure union-find en C avec la structure suivante :

```
struct UnionFind {
  int size;  int *link; int *rank;
};
typedef struct UnionFind uf;
```

On écrira les fonctions `uf* uf_create(int size)` ; `int uf_find(uf* uf, int i)` et `void uf_union(uf* uf, int i, int j)`.