Algorithmique 2 (INFO-F203)
Étude de cas : la structure Union-Find

Jean Cardinal

Février 2021

Questions

Élément majoritaire
: Il est possible de résoudre le problème en un temps linéaire et avec uniquement deux variables supplémentaires.

Bipartition
: On ne connaît pas d'algorithme en temps polynomial ! Le problème est *NP-complet*.

Élément majoritaire

```
Comparable candidat;
int compteur = 0;
int i = 0;
int n = a.length;
while (i < n) {
   if (compteur==0){
      candidat = a[i];
   }
   if (a[i]==candidat){
      compteur++;
   } else {
      compteur--;
   }
   i++;
}
```

Si un élément majoritaire existe, c'est `candidat` !

Union-Find

- n éléments distincts
- Partition en classes
- Identifiant de la classe d'équivalence contenant un élément `p` donné :
  `int find (int p)`
- Union des classes contenant `p` et `q` :
  `void union (int p, int q)`

Relation d'équivalence

Rappel Mathématique 1

Une **relation d'équivalence** est une relation binaire $\equiv$ satisfaisant les propriétés suivantes :

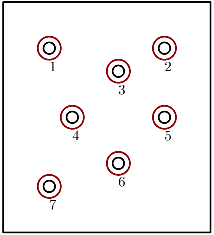
Réflexivité. $x \equiv x$ pour tout x.

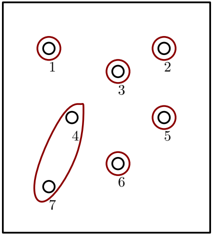
symétrie. Si $x \equiv y$, alors $y \equiv x$.

Transitivité. Si $x \equiv y$ et $y \equiv z$, alors $x \equiv z$.

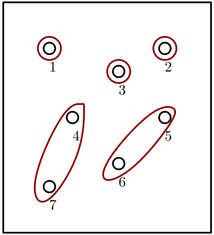
La **classe d'équivalence** d'un élément x est l'ensemble des éléments y tels que $x \equiv y$.

Exemple

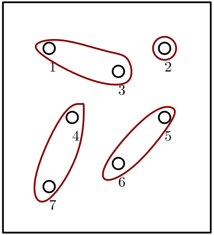




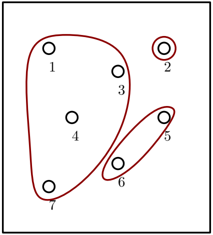
union (4, 7)



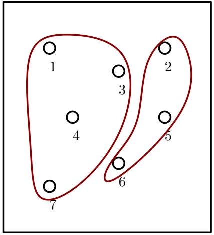
union (6, 5)



union (1, 3)



union (7, 1)



union (2, 6)

Méthode naïve

Dans la première méthode, on stocke à l'indice `i` du tableau `id[]` l'identifiant de la composante à laquelle appartient l'élément `i`. Au départ, tous les éléments appartiennent à des classes distinctes, et la valeur `id[i]` est initialisée à `i`.

```java
public UF(int n) {
  count = n;
  id = new int[n];
  for (int i = 0; i < n; i++)
    id[i] = i;
}
```

Méthode naïve

```
public void union(int p, int q) {
 int pID = id[p];
 int qID = id[q];

 // p and q are already in the same component
 if (pID == qID) return;

 for (int i = 0; i < id.length; i++)
  if (id[i] == pID) id[i] = qID;
 count--;
}
```

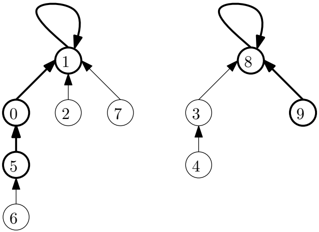
Il est ici facile de voir que le nombre d'accès est toujours au moins égal au nombre n d'éléments dans la structure.

Union rapide

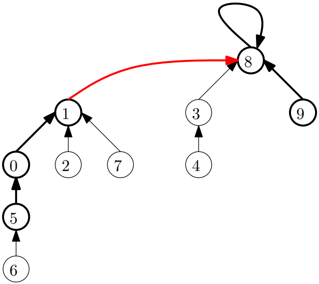
La valeur `parent[p]` contenue à l'indice `p` du tableau `parent` est celle d'un élément appartenant à la même classe d'équivalence que `p`, éventuellement `p` lui-même. On peut interpréter `parent[p]` comme un pointeur. Pour mettre en œuvre la procédure `int find (int p)`, on suit ces pointeurs jusqu'à trouver une valeur `q` telle que `parent[q]` est égal à `q`. C'est cette valeur qui identifiera la classe d'équivalence. On l'appelle la *racine*, puisqu'elle correspond à la racine de l'arbre représentant la classe.

```
public int find(int p) {
 while (p != parent[p])
  p = parent[p];
 return p;
}
```

Union rapide



0	1	2	3	4	5	6	7	8	9
1	1	1	8	3	0	5	1	8	8



0	1	2	3	4	5	6	7	8	9
1	**8**	1	8	3	0	5	1	8	8

Union rapide

Pour procéder à l'union de deux classes, on trouve les racines `rootP` et `rootQ` des deux éléments `p` et `q` en entrée, et on change la valeur de `parent[rootP]` en `rootQ`.

```
public void union(int p, int q) {
 int rootP = find(p);
 int rootQ = find(q);
 if (rootP == rootQ) return;
 parent[rootP] = rootQ;
 count--;
}
```

Nous observons que le coût d'un appel à la procédure `find (p)` est proportionnel à la *hauteur de l'arbre*.

Amélioration ?

Comment contrôler la hauteurs des arbres ?
Lors d'une union, un choix est possible entre

```
parent[rootP] = rootQ;
```

et

```
parent[rootQ] = rootP;
```

On peut faire le choix qui minimise la hauteur...

Union rapide pondérée

```java
public void union(int p, int q) {
  int i = find(p);
  int j = find(q);
  if (i == j)
    return;

  // make shorter root point to taller one
  if     (height[i] < height[j]) parent[i] = j;
  else if (height[i] > height[j]) parent[j] = i;
  else {
        parent[j] = i;
        height[i]++;
      }
  count--;
}
```

Analyse

Proposition 1

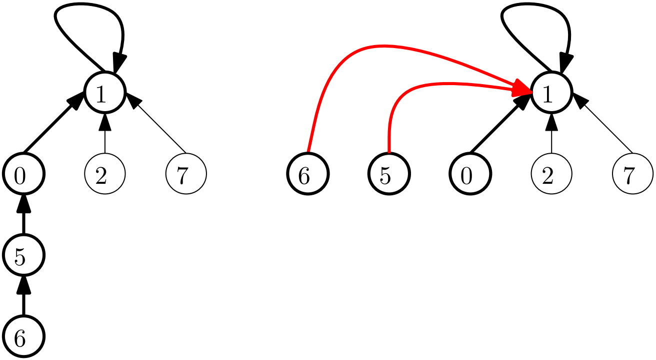
La hauteur d'un arbre dans la forêt construite par la procédure d'union rapide pondérée pour n éléments est au plus $\log n$.

Démonstration

Par induction	Un arbre de hauteur h contient au moins 2^h éléments.
Cas de base	La proposition est vraie à l'initialisation de la structure.
Induction	Montrons que la proposition reste vraie après une union.
HI	Les deux arbres de $\mathtt{p}$ et $\mathtt{q}$ de hauteur respective h_1 et h_2 contiennent $n_1 \geq 2^{h_1}$ et $n_2 \geq 2^{h_2}$ éléments.
Hypothèse	On suppose **sans perte de généralité** que $h_1 \leq h_2$ et qu'on attache donc l'arbre de $\mathtt{p}$ à la racine de celui contenant $\mathtt{q}$.
Étude de cas	Si la hauteur du nouvel arbre est au plus h_2, la proposition est certainement vraie. Supposons donc que la hauteur du nouvel arbre est $h_2 + 1$.
Conclusion	Cela signifie que $h_1 = h_2$. Le nombre d'éléments dans le nouvel arbre est $n_1 + n_2 \geq 2^{h_1} + 2^{h_2} = 2 \cdot 2^{h_2} = 2^{h_2+1}$. La proposition reste donc vraie pour le nouvel arbre.

Enfin si $n \geq 2^h$ éléments, on a bien $h \leq \log_2 n$.

Compression de chemin



Compression de chemin

```
public int find(int p) {
  int root = p;
  while (root != parent[root])
    root = parent[root];
  while (p != root) {
    int newp = parent[p];
    parent[p] = root;
    p = newp;
  }
  return root;
}
```

Compression de chemin

Proposition 2

La complexité de m opérations sur n éléments dans la méthode d'union rapide pondérée combinée à la compression de chemin est $O(m\alpha(m, n))$.

Dans cet énoncé, la fonction $\alpha(m, n)$ est appelée *fonction d'Ackermann inverse*. Cette fonction est croissante, mais croît extrêmement lentement. En pratique, il est raisonnable de considérer que la structure de données obtenue effectue toutes les opérations voulues en temps constant.

Pour la semaine prochaine: Algorithmes de Tri

Tri fusion (Merge Sort)

Tri rapide (Quicksort)

Borne inférieure Pourquoi a-t-on besoin de $\Omega(n \log n)$ comparaisons?