

UE ALGO5 — TD2 — Séance 11 : Union-Find

Objectifs

À la fin de cette séance, vous devriez être capable de :

- choisir des structures de données adaptées aux spécifications ;
- calculer des complexités en moyenne, pondérées, du meilleur et pire cas.

Exercice 1.

Soit un ensemble $E = \{0, \dots, n - 1\}$. On souhaite implémenter une structure permettant de gérer des *partitions* de E .

Rappel : L'ensemble d'ensembles $\{E_1, \dots, E_k\}$ est une partition de E si :

- aucun sous-ensemble n'est vide : $E_i \neq \emptyset$
- E est recouvert par les sous-ensembles : $\bigcup_i E_i = E$
- les sous-ensembles sont deux à deux disjoints : $i \neq j \Rightarrow E_i \cap E_j = \emptyset$.

On spécifie un type `id` permettant d'identifier les sous-ensembles, ainsi que les primitives suivantes :

`id` : type abstrait

Initialiser

{Après l'appel à Initialiser, la structure contient la partition P où chaque élément de E est l'élément unique d'un ensemble de P : $P = \{\{0\}, \dots, \{n - 1\}\}$ }

Find : entier \rightarrow id

{Find(x) renvoie l'identifiant du sous-ensemble auquel appartient x . Find(x)=Find(y) ssi x et y appartiennent au même sous-ensemble. }

Union(données x, y : entiers)

{ Union(x, y) réalise l'union des deux sous-ensembles auxquels appartiennent x et y . Après cet appel, Find(x)=Find(y) }

On souhaite implémenter de manière efficace les primitives Union et Find. Pour cela, on utilise un tableau `Parent` de n entiers représentant une *forêt*, dans laquelle chaque *arbre* contient les éléments d'un même sous-ensemble. `Parent(x)` est défini comme étant le parent de x ; un sous-ensemble est représenté par l'élément à la racine de l'arbre.

`Parent` : tableau sur $[0..n-1]$ d'entiers sur $0..n-1$

Q1. Dessiner la structure `Parent` et les arbres correspondants à l'initialisation, puis après plusieurs appels à Union.

Corrigé —

Après initialisation, `Parent` est représenté par une forêt de nœuds (sans enfants) :

$$Parent = \begin{array}{|c|c|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline \end{array}$$

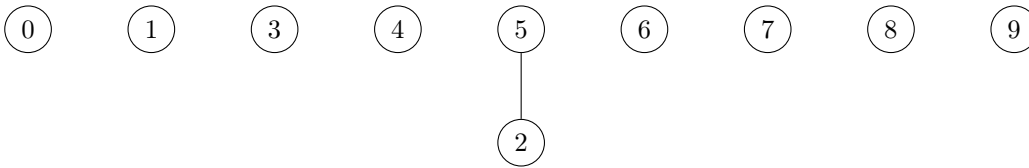


Après appel à Union, la structure dépend de l'implémentation choisie. Un exemple :

1 Union(2,5)

Parent =

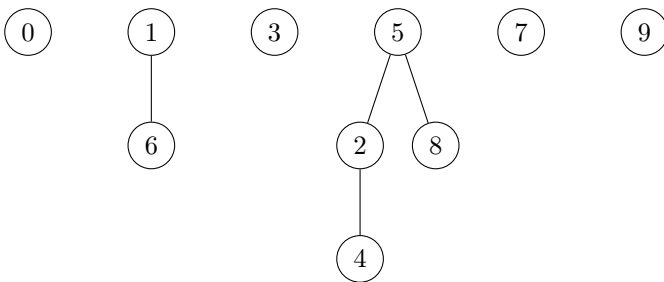
0	1	2	3	4	5	6	7	8	9
0	1	5	3	4	5	6	7	8	9



- 1 Union(4,2)
- Union(6,1)
- 3 Union(8,5)

Parent =

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



Q2. Réaliser l'opération d'initialisation, puis les primitives Union et Find. Quel est le coût de ces deux opérations ?

Corrigé —

L'identifiant d'un ensemble de la partition correspond à l'élément à la racine de l'arbre représentant cet ensemble.

Initialiser crée n nœuds pointant sur eux-mêmes (donc n arbres comportant un seul nœud). On identifie par la suite la racine r d'un arbre au fait que $\text{Parent}[r]=r$.

Initialiser:

```
2  pour i de 0 à n-1
   Parent[i] := i
```

```
4  Find(x):
```

```
6  si Parent[x] = x alors
   retourner x
```

```
8  sinon
   retourner Find(Parent[x])
```

```
10 Union(x,y):
```

```
12 { on relie la racine de l'arbre contenant x à y }
```

```
px := Find(x)
```

```
14 Parent[px] := y
```

Dans le pire cas, on va créer un arbre réduit à une liste : les deux opérations sont en $O(n)$ (coût de recherche de la racine).

Q3. Proposer puis implémenter une méthode permettant d'améliorer le coût des deux opérations.

Corrigé —

L'idée est d'essayer de minimiser la hauteur des arbres construits. On peut identifier deux heuristiques :
— «*union by rank*» : on conserve, pour chaque arbre, la hauteur de l'arbre. Lorsqu'on réalise une union, on relie la racine de l'arbre le moins haut à la racine de l'arbre le plus haut.

{ $h[x]$ contient la hauteur de l'arbre de racine x }

2 Initialiser:

pour i **de** 0 à $n-1$

4 Parent[i] := i

$h[i]$:= 0

6

Union(x,y):

8 { on relie la racine de l'arbre contenant x à y }

px := Find(x)

10 py := Find(y)

si $h[px] > h[py]$ **alors**

12 Parent[py] := px

sinon

14 Parent[px] := py

 { cas particulier : arbres de hauteurs égales }

16 **si** $h[px] = h[py]$ **alors**

$h[py]$:= $h[py] + 1$

— «*path compression*» : lors de chaque opération Find(x), on relie le nœud x à la racine de l'arbre

Find(x):

2 **si** Parent[x] = x **alors**

 retourner x

4 **sinon**

 Parent[x] := Find(Parent[x])

6 retourner Parent[x]
