# TP4 - Union-Find

## Partition d'un ensemble

Soit $E$ un ensemble. Une partition $(X_i)_{i\in[1,p]}$, $p \in \mathbb{N}^*$, de $E$ est un ensemble de parties non vides de $E$ deux à deux disjointes dont l'union est $E$.

$$\bigcup_{i=1}^{p} X_i = E \qquad \forall (i,j) \in [1,p]^2, i \neq j, X_i \cap X_j = \varnothing$$

Par exemple, si $E = \{0,1,2,3,4,5,6,7\}$, une partition de $E$ est :

$$X_1 = \{3,1,4\} \qquad X_2 = \{0\} \qquad X_3 = \{6,5,2,7\}$$

En informatique, étant donné un ensemble d'objets, partitionner cet ensemble consiste à construire des sous-ensembles de ces derniers appelés *classes* puis à définir des fonctions qui permettent de manipuler *efficacement* ces classes. Avec l'ensemble précédent, une représentation initiale possible de $E$ est un tableau d'entiers dont chaque élément est son indice dans le tableau. Avec l'exemple précédent, on aurait :

$$t = [0,1,2,3,4,5,6,7]$$

Dans cette première représentation, tout se passe comme si chaque entier définissait sa propre partition.
Définir une partition de $E$ consiste alors à attribuer un entier correspondant à la classe d'appartenance de chaque élément de $E$. Cet entier peut être le plus petit de la classe. Avec l'exemple précédent, la partition :

$$0 \to \{0\} \qquad 1 \to \{1,3,4\} \qquad 2 \to \{2,5,6,7\}$$

ménerait à la définition du tableau suivant.

$$t = [0,1,2,1,1,2,2,2]$$

Deux opérations fondamentales doivent permettre de manipuler les partitions.

- *Unir* deux partitions pour n'en faire qu'une seule. Par exemple, unir les classes 0 et 1 précédentes ferait évoluer le contenu du tableau de la façon suivante :

  $$t = [0,0,2,0,0,2,2,2]$$

  associé aux deux classes :

  $$0 \to \{0,1,3,4\} \qquad 2 \to \{2,5,6,7\}$$

  Cette opération remplace l'entier associé à l'une des partitions par l'entier associé à l'autre partition[1]. Cette opération n'est pas de coût de constant mais a un coût au pire en $O(n)$.
- *Trouver* la partition à laquelle appartient un entier $i$. Ici, cette opération est élémentaire : il suffit de lire l'entier $t[i]$ associé, opération de coût constant avec un tableau.

En pratique, la question se pose de comment définir une structure de données alternative de meilleure complexité. La structure *union-find*[2] répond à cette question en apportant deux améliorations à cette approche.

## Implémentation naïve

Avant de présenter cette solution, l'implémentation OCaml suivante est une approche naïve. L'interface définit les fonctions pour *créer*, pour *trouver* et pour *unir* des classes, le type `uf` étant un alias pour un tableau d'entiers.

```
type uf = int array
val create: int -> uf
val find: int array -> int -> int
val union: uf -> int -> int -> unit
```

Les codes suivants définissent très simplement les fonctions.

1. Par convention, on peut conserver le plus petit numéro de partition.
2. En français, on parle de structure *unir et trouver*, expression peu usitée.

```
type uf = int array

let create n = Array.init n (fun i -> i)
let size (t:uf) = Array.length t
let find (t:uf) i = t.(i)

let union (t:uf) i j =
  let n = size t in
  let c_i = find t i in
  let c_j = find t j in
  let mini = min c_i c_j in
  let maxi = max c_i c_j in
  for k = 0 to n-1 do
    if t.(k) = maxi then t.(k) <- mini
  done
```

L'appel à `create n` construit une partition de $\{0, 1, \dots n-1\}$ où chaque entier forme une classe à lui tout seul.

**Question 1.** Quelles sont les complexités des fonctions `find` et `union` ?

## Union-Find

L'idée principale est de lier entre eux les éléments d'une même classe. Une représentation sous la forme d'un arbre illustre cette approche. Considérons les étapes suivantes de construction d'une partition.

```
t <- create 8
union t 1 3
union t 3 4
union t 2 5
union t 2 6
union t 6 7
```

**Question 2.** Représenter les arbres associés aux différentes étapes de la construction précédente.

Ces relations peuvent se représenter par un simple tableau qui lie 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 notre exemple est représentée par le même tableau que celui déjà défini plus haut.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 1 | 1 | 2 | 2 | 2 |

Il est encore immédiat de réaliser les deux opérations `find` et `union` sur la base de cette idée. L'opération `find` se contente de suivre les liaisons jusqu'à trouver le représentant.

```
let rec find t i =
  if t.(i) = i then i else find t t.(i)
```

Enfin, l'opération `union` commence par trouver les représentants des deux éléments, puis lie l'un des deux représentants à l'autre, en choisissant arbitrairement.

```
let union t i j =
  t.(find t i) <- find t j
```

En particulier, si `i` et `j` sont déjà dans la même classe, alors `union` est sans effet.

Bien que simple à définir, cette structure est naïve car 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 $0 \leqslant i < n-1$. Les complexités de `find` et de `union` peuvent être aussi grandes que $O(n)$. Ce n'est pas acceptable en pratique mais l'approche sous forme d'arbre permet les deux améliorations envisagées plus haut.

### Union pondérée

La première amélioration consiste à 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. On commence par calculer les deux représentants `ri` et `rj` des éléments `i` et `j` dont on cherche à réunir les classes. On les compare pour savoir s'il y a quelque chose à faire.

```
let union uf i j =
  let ri = find uf i in
  let rj = find uf j in
  if ri <> rj then
```

Le cas échéant, on compare les rangs des deux classes. Si celui de `ri` est strictement plus petit que celui de `rj`, on fait de `rj` le représentant de l'union, c'est-à-dire qu'on lie `ri` à `rj`.

```
    if uf.rank.(ri) < uf.rank.(rj) then
      uf.link.(ri) <- rj
```

Le rang n'a pas besoin d'être mis à jour pour cette nouvelle classe. En effet, seuls les chemins de l'ancienne classe de `ri` ont vu leur longueur augmentée d'une unité et cette nouvelle longueur n'excède pas le rang de `rj`. Si en revanche c'est le rang de `rj` qui est le plus petit, on procède symétriquement.

```
    else (
      uf.link.(rj) <- ri;
```

Dans le cas où les deux classes ont le même rang, l'information de rang doit alors être mise à jour, car la longueur du plus long chemin est susceptible d'augmenter d'une unité.

```
      if uf.rank.(ri) = uf.rank.(rj) then
        uf.rank.(ri) <- uf.rank.(ri) + 1
  )
```

Illustrons le déroulement de ce code sur en reprenant l'exemple précédent rappelé ci-dessous.

```
t <- create 8
union t 1 3
union t 3 4
union t 2 5
union t 2 6
union t 6 7
```

**Propriété 1**

Une classe de rang $k$ a des chemins de longueur maximale $k$ et possède au moins $2^k$ éléments.

*Démonstration.* On fait la preuve par récurrence sur le nombre d'appels à `union`. C'est vrai initialement car chaque classe a le rang $k = 0$ et un unique élément.

La dernière classe construite, de rang $k$, provient

- soit d'une classe de rang $k$ et d'une classe de rang $k' < k$ :
  - les nouveaux chemins ont une longueur maximale $k' + 1 \leqslant k$,
  - on a au moins $2^k$ éléments provenant de la première classe ;
- soit de deux classes de rang $k - 1$ :
  - les nouveaux chemins ont une longueur maximale $k - 1 + 1 = k$,
  - on a au moins $2^{k-1} + 2^{k-1} = 2^k$ éléments.

Par ailleurs, l'opération `find` ne fait que consulter la structure. □

**Question 3.** Des ces résultats, en déduire la complexité des opérations `find` et `union`.

## Compression de chemin

On peut encore améliorer l'efficacité de cette structure. L'idée consiste à *compresser les chemins* pendant la recherche effectuée par `find` : on lie directement au représentant tous les éléments trouvés sur le chemin parcouru pour l'atteindre. Si on reprend la partition définie notre dernier exemple, illustrons l'effet de la recherche des classes associées à 4, 6 et 7. Une très légère modification du code de la fonction `find` suffit pour réaliser une telle compression de chemins :

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

En particulier, après le calcul de $r =$ `find` $i$, on a `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 $r$. On a donc raccourci le chemin qui mène de $i$ à $r$, mais également *tous* les chemins qui passaient par n'importe lequel des éléments initialement situés entre $i$ et $r$.

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. La propriété 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 $\alpha$ 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 de ce cours. Il a été prouvé que cette complexité est optimale.

### Fonction d'Ackermann et logarithme itéré

La fonction d'Ackermann est définie récursivement par :

$$A(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m-1, A(m,n-1)) & \text{si } m > 0 \text{ et } n > 0 \end{cases}$$

Cette fonction croît extrêmement vite lorsque ses paramètres augmentent. Par exemple, $A(4,2)$ est un nombre qui comporte 19 729 chiffres !

La réciproque de $A(n,n)$ croît alors très lentement. Souvent notée $\alpha(n)$, elle intervient pour exprimer des complexités liées à l'usage de certaines structures de données comme *union-find* ou la mise en œuvre d'un algorithme de calcul d'*arbre couvrant de poids minimal*.

Pour décrire la complexité de certains algorithmes, on rencontre une autre fonction qui croît également très lentement : le *logarithme itéré* d'un nombre $n$, noté $\log^* x$. C'est l'entier égal au nombre de fois que le logarithme doit être appliqué pour que le résultat soit inférieur à 1. Ses premières valeurs sont les suivantes.

| $x$ | $]0,1]$ | $]1,2]$ | $]2,4]$ | $]4,16]$ | $]16,65536]$ | $]65536,2^{65536}]$ |
|---|---|---|---|---|---|---|
| $\log^* x$ | 0 | 1 | 2 | 3 | 4 | 5 |

La fonction $\alpha$ précédente croît encore moins vite que le logarithme itéré !

# Arbre couvrant

On considère uniquement à des graphes non orientés, connexes et *sans boucles*, c'est-à-dire sans arc de la forme $a - a$.

**Définition 1 – arbre couvrant**

Étant donné un graphe $G$, un *arbre couvrant* de $G$ est un sous-ensemble $T$ d'arcs de $G$ tel que $T$ est un arbre et chaque sommet de $G$ est l'extrémité d'au moins un arc de $T$ (on dit que $T$ *couvre* tous les sommets de $G$).

Voici par exemple un graphe de six sommets à gauche et l'un de ses arbres couvrants à droite.


Il y a bien sûr d'autres arbres couvrants de ce même graphe.

**Définition 2 – arbre couvrant minimal**

Soit $G$ un graphe non orienté pondéré. Un *arbre couvrant minimal* (ACM) de $G$ est un arbre couvrant de $G$ dont la somme des poids des arcs est minimale.

Voici un exemple de graphe pondéré, à gauche, et un arbre couvrant minimal à droite.


Le poids total est ici 12 et il n'y a pas d'arbre couvrant de poids inférieur.

# Algorithme de Kruskal

Étant donné un graphe $G$, l'*algorithme de Kruskal* construit un ACM pour $G$. En supposant que les sommets de $G$ sont les entiers $0, 1, \dots, V-1$, le fonctionnement de l'algorithme de Kruskal est le suivant.

1. Soit $U$ une structure *union-find* pour les sommets $0, 1, \dots, V-1$ de $G$.
2. Soit $Q$ une file de priorité contenant tous les arcs de $G$,ordonnés par leur poids.
3. Soit $T$ une liste d'arcs, initialement vide.
4. Tant que $T$ contient moins que $V-1$ arcs :
   - **(a)** retirer un arc $x - y$ de poids minimal de la file de priorité $Q$,
   - **(b)** si $x$ et $y$ ne sont pas dans la même classe pour $U$, alors
     - **i.** ajouter l'arc $x - y$ à $T$,
     - **ii.** fusionner dans $U$ les classes de $x$ et $y$.

À la fin de l'algorithme, la liste $T$ contient un arbre couvrant minimal pour $G$. Illustrons le fonctionnement de cet algorithme sur l'exemple donné plus haut.

| arc $x - y$ | poids | action | $U$ |
|---|---|---|---|
| | | | $\{0\},\{1\},\{2\},\{3\},\{4\},\{5\}$ |
| $0-1$ | 1 | ajouté | $\{0,1\},\{2\},\{3\},\{4\},\{5\}$ |
| $0-2$ | 2 | ajouté | $\{0,1,2\},\{3\},\{4\},\{5\}$ |
| $4-5$ | 2 | ajouté | $\{0,1,2\},\{3\},\{4,5\}$ |
| $1-2$ | 3 | ignoré | — |
| $1-3$ | 3 | ajouté | $\{0,1,2,3\},\{4,5\}$ |
| $2-3$ | 4 | ignoré | — |
| $3-4$ | 4 | ajouté | $\{0,1,2,3,4,5\}$ |

On s'arrête là car $T$ contient alors 5 arcs. En particulier, les arcs $3-5$ (de poids 5) et $2-4$ (de poids 6) ne sont pas retirés de la file. Ce n'est pas la seule exécution possible, car il y a plusieurs arcs de même poids. Ainsi, si on considère l'arc $3-4$ avant l'arc $2-3$, on fait une étape de moins. Mais dans tous les cas, on obtiendra bien un arbre couvrant de poids total 12.

La mise en œuvre de l'algorithme de Kruskal nécessite l'utilisation d'une structure de données de type *union-find* et d'une structure de données de type *file de priorité*.

**Question 4.** Le graphe de la figure 1 sert d'exemple. Présenter et détailler les étapes de la construction d'un *arbre couvrant minimal*.

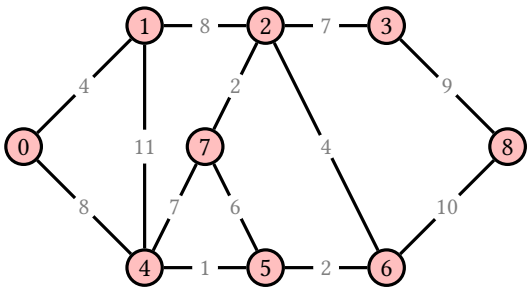


**Figure 1** – Graphe exemple.

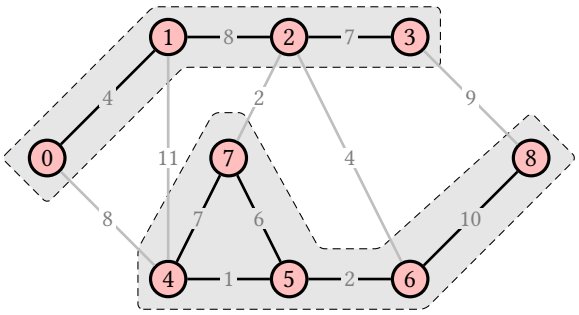


**Figure 2** – Coupure matérialisée par les arêtes grises.

**Question 5.**
- ☐ **5.1.** Proposer une ADT pour la structure *union-find*.
- ☐ **5.2.** Donner une implémentation OCaml ce cet ADT.
- ☐ **5.3.** Étudier la complexité de vos fonctions.

**Question 6.**
- ☐ **6.1.** Proposer une ADT pour la structure *file de priorité*.
- ☐ **6.2.** Donner une implémentation OCaml ce cet ADT.
- ☐ **6.3.** Étudier la complexité de vos fonctions.

**Question 7.**
- ☐ **7.1.** Donner une implémentation OCaml de l'algorithme de Kruskal.
- ☐ **7.2.** Quelle est la complexité de votre solution ?
- ☐ **7.3.** Établir sa correction.

**Question 8.** Justifier qu'un tel algorithme appartienne à la catégorie des *algorithmes gloutons*.

## Algorithme de Prim

**Définition 3 – coupure d'un graphe**

Une *coupure* d'un graphe $G = (V, E)$ est un ensemble d'arêtes dont une extrémité est dans un sous-ensemble $U \subseteq V$ et l'autre extrémité est dans $V \setminus U$. La coupure est notée $(U, V \setminus U)$.

La figure 2 illustre cette notion.

L'*algorithme de Prim* détermine un *arbre couvrant minimal* en mettant en œuvre une *stratégie gloutonne*. À partir d'un sommet quelconque, l'algorithme construit un ACM par *croissance progressive*. Si $G$ est de taille $n$, un ACM de $G$ comporte $n-1$ arêtes. L'ACM est unique si et seulement les poids du graphe sont tous distincts. Sinon, plusieurs ACM peuvent être associés à un graphe. Soit $G = (V, E)$ un graphe et $n = |V|$.

- Choisir un sommet $v_{i_0} \in V$ et poser $U_0 = \{v_{i_0}\}$.
- Dans la coupure $(U_0, V \setminus U_0)$, trouver une arête $\{v_{i_0}, v_{i_1}\}$ de poids minimal puis poser $U_1 = \{v_{i_0}, v_{i_1}\}$.
- Dans la coupure $(U_1, V \setminus U_1)$, trouver une arête $\{v, v_{i_2}\}$ de poids minimal où $v \in U_1$ puis poser $U_2 = \{v_{i_0}, v_{i_1}, v_{i_2}\}$.
- Répéter cette procédure jusqu'à obtenir l'ensemble $U_{n-1}$.

$U_{n-1}$ est un *arbre couvrant minimal* de $G$.

**Question 9.** Proposer des structures de données pour mettre en œuvre le plus efficacement possible cet algorithme.

**Question 10.** Donner une implémentation OCaml de votre proposition.