Chapitre 5

Union-find

Ce chapitre s'intéresse au type abstrait **partition d'un ensemble** ou **relation d'équivalence** sur un ensemble. Il s'agit de trouver une structure de données efficace pour gérer trois opérations :

— Créer une relation d'équivalence triviale où chaque élément est seul dans sa classe d'équivalence ;
— Tester si deux éléments sont dans la même classe d'équivalence
— Faire l'union de deux classes d'équivalences pour n'en faire qu'une seule.

5.1 Motivation : génération d'un labyrinthe

On souhaite créer un labyrinthe comme la figure 5.1. Les contraintes sont les suivantes :

— On peut se déplacer partout et en particulier il y a un chemin de l'entrée vers la sortie ;
— On ne peut pas tourner en rond (ie. on a assez de murs)

La figure 5.2 présente un labyrinthe où l'on peut tourner en rond. Ce n'est pas un labyrinthe que l'on souhaite générer.

Remarque. Cet algorithme est un cas particulier de l'algorithme de Kruskal que l'on présente dans le chapitre 8.

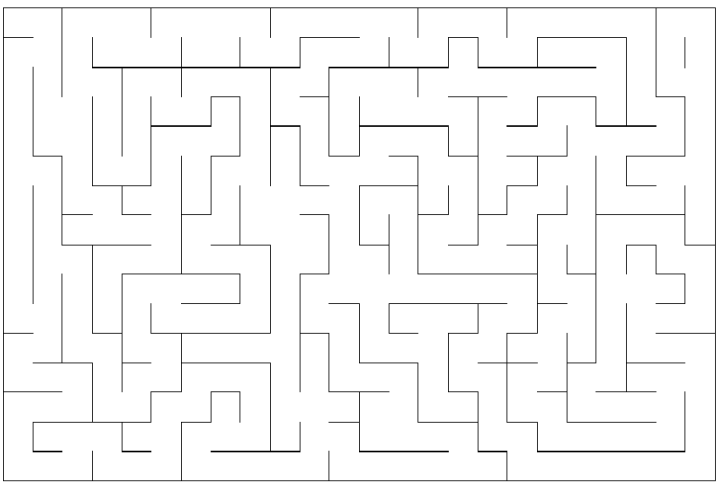


FIGURE 5.1 – Un labyrinthe

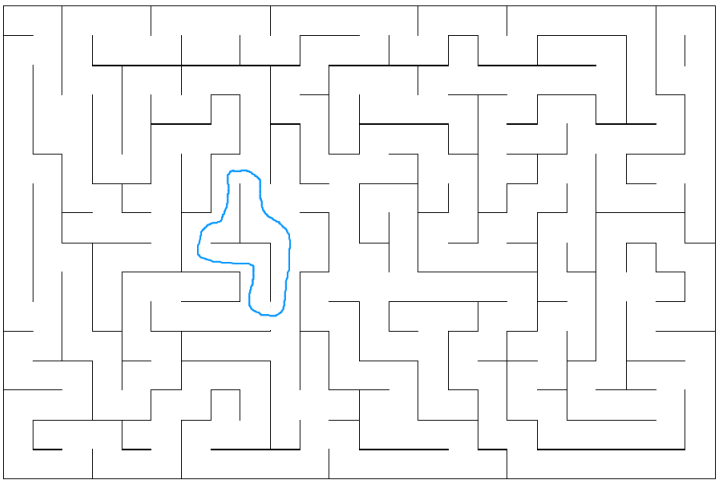


FIGURE 5.2 – Un mauvais labyrinthe car on peut tourner en rond

Algorithme 14 : créer_labyrinthe

Sorties : les murs d'un labyrinthe correct

S = la relation d'équivalence 'égalité' sur les cases
E = les murs possibles
$L := E$
while S *contient plusieurs classes* **do**
    $\{x, y\}$:= un mur au hasard dans E
    si x *et* y *ne sont pas dans la même classe dans* S **alors**
        on supprime $\{x, y\}$ de E
        on fusionne les classes de x et de y dans S
retourner L

5.2 Implémentation naïve

On peut implémenter une relation d'équivalence à l'aide d'un tableau associatif qui à tout élément associe un entier qui désigne le numéro de la classe d'équivalence.

Par exemple la table :

Mohammed	Martha	Srdjan	Wang	Brenda	François
1	2	1	2	2	3

représente le fait que les classes de la relation d'équivalence sont {Mohammed, Srdjan}, {Martha, Wang, Brenda} et {François}.

Savoir si deux éléments appartiennent à la même classe est facile : c'est le cas lorsqu'ils ont le même numéro. Par contre, l'union requiert de parcourir tout le tableau.

5.3 Implémentation avec des arbres

Les arbres ici ne sont pas binaires mais d'arité quelconque. On peut utiliser une forêt d'arbres : chaque partie de la partition est représentée par un arbre. On oriente les flèches : elles pointent vers le père. De plus, la racine est reliée à elle-même.

Algorithme 15 : créer_union_find

Entrées : $S = \{s_1, \cdots, s_n\}$
Sorties : Union Find

retourner *Les arbres à un seul sommet :* $[s_1, \cdots, s_n]$

Algorithme 16 : find

Entrées : x
Sorties : La racine de l'arbre qui contient x

si $x = x.parent$ **alors**
  retourner x
sinon
  retourner *find(x.parent)*

Algorithme 17 : union

Entrées : x, y
Sorties : Si x, y sont dans des arbres, fusionne les deux arbres

$r_x := find(x)$
$r_y := find(y)$
si $r_x = r_y$ **alors**
  Ne rien faire
si $r_x.h > r_y.h$ **alors**
  $r_y.parent := r_x$;
sinon
  $r_x.parent := r_y$;
  si $r_x.h = r_y.h$ **alors**
    $r_y.h := r_y.h + 1$;

La proposition qui suit peut paraître surprenante mais elle provient du fait que l'on assure de bien faire que la hauteur reste la plus petite possible dans union.

Proposition.
Il y a au moins 2^h nœuds dans un arbre de hauteur h.

Démonstration.
On montre que c'est un invariant au cours de l'utilisation de la structure de données union-find.

- Au début, les arbres sont de hauteur 0 car ce sont des racines toutes seules. Et il y a au moins 1 noeud dans un arbre-racine de hauteur 0.
- On suppose qu'à une certaine étape l'invariant est vérifiée. Maintenant regardons un appel de union. On construit un nouvel arbre à partir de l'arbre de racine r_x de hauteur $h(r_x)$ et de l'arbre de racine r_y de hauteur $h(r_x)$. Entamons une étude de cas.
  - Si $h(r_x) > h(r_y)$, l'arbre résultant est l'arbre de racine r_x auquel on a acroché l'arbre r_y. Il est toujours de la même hauteur et contient plus de noeud. A fortiori, la propriété va rester vérifiée.
  - Si $h(r_x) < h(r_y)$, raisonnement symétrique.
  - Si $h(r_x) = h(r_y)$, l'arbre résultant aura une hauteur de $h = h(r_x) + 1$. Il contient les noeuds de r_x et de r_y soit au moins $2^{r_x} + 2^{r_x} = 2^h$ noeuds. Et la prioriété reste vérifiée.

■

Corollaire. *Les hauteurs des arbres de la forêt sont inférieures à* $\log_2(n)$.

Démonstration.
Par l'absurde. S'il existe un arbres de hauteur $h > \log_2(n)$, alors il contient au moins $2^h > n$. Or la structure contient n éléments. Contradiction. ■

Le coût des algorithmes find et union est au pire des cas en la plus grande hauteur apparaissant dans la structure. Autrement dit, les coûts sont majorés par $O(\log_2(n))$, où n est le nombre d'éléments dans la structure.

Implémentation	Tableau	Arbres
|creer_ union_ find|	$O(n)$	$O(n)$
|find|	$O(1)$	$O(\log_2(n))$
|union|	$O(n)$	$O(\log_2(n))$

5.4 Amélioration : compression de chemins

Quand on cherche un représentant avec `find`, on est idiots. On peut en profiter pour réduire le chemin d'un nœud à sa racine. On parle de compression de chemins.

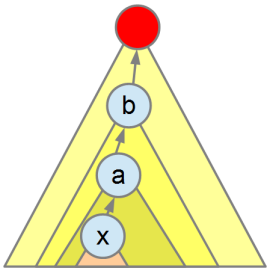
Ainsi, on remplace l'implémentation de `find` par :

Algorithme 18 : find

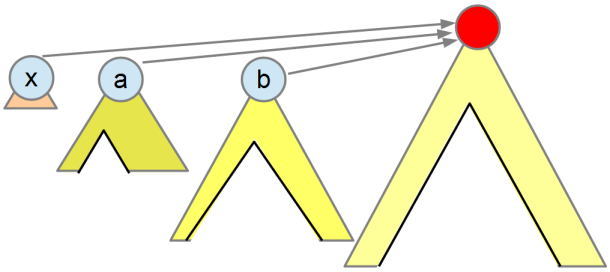
Entrées : x
Sorties : La racine de l'arbre qui contient x

si $x \neq x.parent$ **alors**
  $x.parent := find(x.parent)$
retourner $x.parent$

Exemple 20. *Par exemple, si l'on exécute find(x) et que l'arbre qui contient x ressemble à ceci :*



alors à la fin de l'appel find(x), l'arbre est transformé de cette manière :



5.5 *Analyse amortie de la compression de chemin

Remarque 6. *On peut laisser tomber l'analyse amortie qui suit pour une première lecture.*

5.5.1 Rangs

On appelle *rang* d'un nœud la hauteur qu'il aurait s'il n'y avait pas compression des chemins. Ainsi, les rangs sont entre 0 et $\log_2(n)$. Dès lors qu'un nœud cesse d'être racine, son rang est fixé pour toujours. On stocke toujours le rang de y dans $y.h$.

5.5.2 Intervalles

On va couvrir l'intervalle $[1, \log_2(n)]$ par des intervalles de la forme $[k+1, 2^k]$ où k est une tour de puissance de deux. Voici précisément ces intervalles que l'on a numérotés :

$\{1\}$	$\{2\}$	$\{3,4\}$	$\{5,\dots,16\}$	$\{17,\dots,2^{16}=65536\}$	$\{65537,\dots,2^{65536}\}$	$\dots$
n° 0	n° 1	n° 2	n° 3	n° 4	n° 5	$\dots$

Le j^e intervalle est de la forme $[2{\uparrow\uparrow}(j-1)+1,\dots,2{\uparrow\uparrow}j]$
où
— $2{\uparrow\uparrow}(-1)=0$;
— $2{\uparrow\uparrow}j = 2^{2{\uparrow\uparrow}(j-1)}$ pour tout $j\in\mathbb{N}$.
C'est à dire $2{\uparrow\uparrow}0=1$, $2{\uparrow\uparrow}1=2$, $2{\uparrow\uparrow}2=4$, $2{\uparrow\uparrow}3=16$, etc.
On appelle $\mathcal{I}$ l'ensemble de ces intervalles.

5.5.3 Repérer ces intervalles

La fonction suivante donne à tout entier le numéro de l'intervalle dans lequel il est.
Définition.
On définit $\log^*(n)$ le plus petit nombre k tel que :

$$\underbrace{\log_2(\log_2(\cdots(\log_2(n)}_{k\text{ fois}}\underbrace{)\cdots)}_{k\text{ fois}} \leqslant 1$$

Exemple 21.

— $\log^*(1) = 0$;
— $\log^*(2) = 1$;
— $\log^*(3) = 2$;
— $\log^*(4) = 2$;
— $\log^*(5) = 3$;
— ... ;
— $\log^*(2^{2^2}) = 3$;
— $\log^*\left(2^{2^{2^2}}\right) = 5$

5.5.4 Distribution d'argent

On va distribuer de l'argent aux nœuds au cours de l'exécution de l'algorithme selon la règle suivante :

— Un nœud x reçoit $2\uparrow\uparrow j$ euros quand il cesse d'être racine (et son rang $x.h$ ne sera alors jamais plus modifié) et que $x.h \in [2\uparrow\uparrow(j-1)+1, 2\uparrow\uparrow j]$.

Lemma 2. *On distribue au plus $n \log^*(n)$ euros durant toute la vie de la structure de données.*

Démonstration.

Fait : Il y en a au plus $\frac{n}{2\uparrow\uparrow j}$ nœuds de rang $r \in [2\uparrow\uparrow(j-1)+1, 2\uparrow\uparrow j]$.

En effet, comme le nombre de nœuds de rang h est au plus $\frac{n}{2^h}$, alors le nombre de nœuds de rang $h \in [2\uparrow\uparrow(j-1)+1, 2\uparrow\uparrow j]$ est majoré par le nombre de nœud dont le rang est supérieur ou égal à $2\uparrow\uparrow(j-1)+1$ est majoré par

$$\frac{n}{2^{2\uparrow\uparrow(j-1)+1}} + \frac{n}{2^{2\uparrow\uparrow(j-1)+2}} + \cdots = \frac{n}{2^{2\uparrow\uparrow(j-1)}} = \frac{n}{2\uparrow\uparrow j}.$$

Ainsi, l'argent distribué durant toute l'utilisation de la structure de données est majoré par :

$$\sum_{j=1}^{\ln^*(n)} \frac{n}{2\uparrow\uparrow j} 2\uparrow\uparrow j = n \ln^*(n)$$

■

5.5.5 Paiement

Regardons un appel de `find` où $x \neq y.parent$:

- Si $\log^*(x.h) \neq \log^*(x.parent.h)$, on comptabilise une opération.
- Si $\log^*(x.h) = \log^*(x.parent.h)$, on ne comptabilise pas l'opération mais x paie 1 euro.

Lemma 3. *Aucun nœud n'est "à découvert".*

Démonstration.

À chaque fois qu'un nœud paie, on le connecte à la racine dont le rang est stristement plus grand que le rang $x.parent$. (bien sûr, cette racine peut devenir un jour un noeud interne via `union`). Ainsi, si le rang de $x \in [2\uparrow\uparrow(j-1)+1, 2\uparrow\uparrow j]$, alors, au bout de au plus $2\uparrow\uparrow j$ paiements, x et $x.parent$ auront des rangs dans des intervalles différents. Donc x ne paiera plus jamais car il sera toujours connecté à un élément de rang dans un intervalle différent. ■

5.5.6 Bilan

Il y a au plus $n \log^*(n)$ opérations non comptabilisées durant toute l'utilisation de la structure. Chaque appel coûte $O(\log^* n)$ opérations comptabilisées.

Ainsi, si un algorithme (Kruskal par exemple) appelle m fois `find` cela coûte $O((n+m)\log^* n)$.