ALGO1 – Union-find

François Schwarzentruber

8 décembre 2020

But : type abstrait pour représenter une relation d'équivalence avec les opérations créer, union et find.

1 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.

Exemple 1 On représente une relation d'équivalence avec les classes {a, c}, {b, d, e, h} et {f, g} par :

a	b	c	d	e	f	g	h
a	b	a	b	b	g	g	b

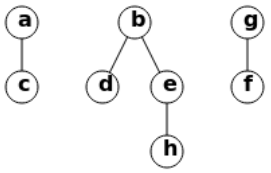
Savoir si deux éléments appartiennent à la même classe est en $O(1)$: on vérifie qu'ils ont le même représentant. Par contre, l'union requiert de parcourir tout le tableau et d'affecter le même représentant aux éléments concernés.

2 Implémentation avec des arbres

Le but est de *réaliser le calcul de l'union de manière paresseuse*. Pour cela, au lieu de stocker directement un représentant dans un tableau, on s'offre une forêt d'arbres d'arité quelconque.

Définition 2 Une structure de données *union-find* est une forêt d'arbres d'arité quelconque. Chaque classe est représentée par un arbre, où les flèches sont orientés vers le haut. De plus, la racine est reliée à elle-même. La racine est le représentant de la classe.

Exemple 3



pre : $S = \{s_1, \cdots, s_n\}$
post : construit la structure union-find où chaque élément s_i est seul dans sa classe
procédure créer_union_find(S)
    construire les arbres à un seul sommet : $[s_1, \cdots, s_n]$

pre : x
post : La racine de l'arbre qui contient x
fonction find(x)
    si $x = x.parent$ **alors**
        retourner x
    sinon
        retourner find($x.parent$)

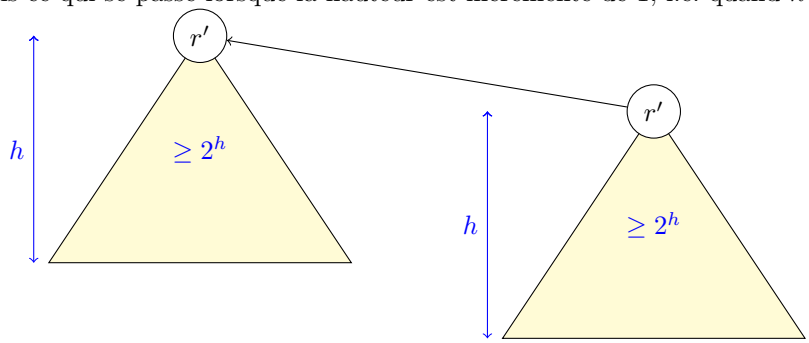
```
pre : x, x'
post : Si x, x' sont dans des arbres différents, fusionne les deux arbres
procédure union(x, x')
    r := find(x)
    r' := find(x')
    si r ≠ r' alors
        si r.h > r'.h alors
            r'.parent := r ;
        sinon
            r.parent := r' ;
            si r.h = r'.halors r'.h := r'.h + 1 ;
```

Proposition 4 Il y a au moins 2^h nœuds dans un arbre de hauteur h.

DÉMONSTRATION. C'est un invariant au cours de l'utilisation de la structure de données union-find.

— **Initialisation.** 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.

— **Conservation.** Supposons qu'à une certaine étape l'invariant est vérifiée. Maintenant regardons un appel de union. Si la hauteur de l'arbre ne change pas, la propriété reste vraie car on ne fait que rajouter des noeuds. Regardons ce qui se passe lorsque la hauteur est incrémenté de 1, i.e. quand $h = r.h = r'.h$.



Chaque arbre est de hauteur h et donc contient au moins 2^h nœuds, par l'invariant. En connectant r' à r, on obtient un arbre de racine r, de hauteur $h+1$ et contenant au moins $2^h + 2^h = 2^{h+1}$ nœuds. L'invariant est donc toujours vérifiée.

■

Corollaire 5 Soit n le nombre d'éléments dans la structure de données union-find. Les hauteurs des arbres de la forêt sont inférieures à $\log_2(n)$.

DÉMONSTRATION. D'après la proposition 4, on a $2^h \leq n$. On conclut en passant au log. ■

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 en $O(\log_2(n))$, où n est le nombre d'éléments.

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

Corollaire 6 Soit n le nombre d'éléments dans la structure de données union-find. Il y a au plus $\frac{n}{2^h}$ nœuds de hauteur h.

Corollaire 7 Il y a au plus $\frac{n}{2^h}$ nœuds de hauteur $\geq h+1$.

DÉMONSTRATION. Par le corollaire 6, le nombre de nœuds de hauteur $\geq h$ est majoré par

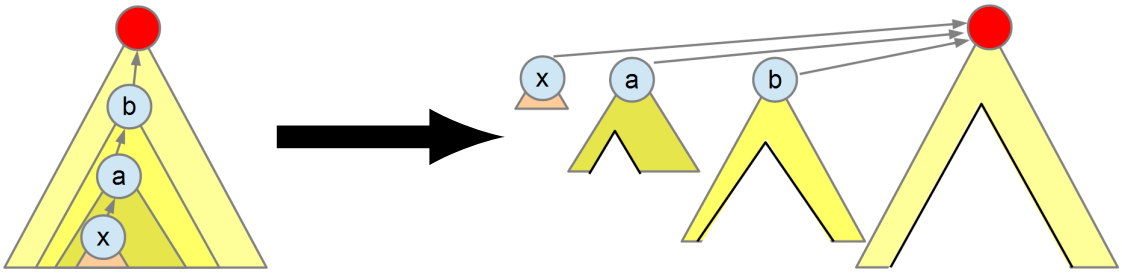
$$\frac{n}{2^{h+1}} + \frac{n}{2^{h+2}} + \cdots = \frac{n}{2^h}.$$

■

3 Amélioration : compression de chemins

En cherchant un représentant avec `find`, on en profite pour connecter les sommets visités à la racine. On parle de *compression de chemins*.

Exemple 8 L'exécution de find(x) réalise la compression de chemins comme suit :

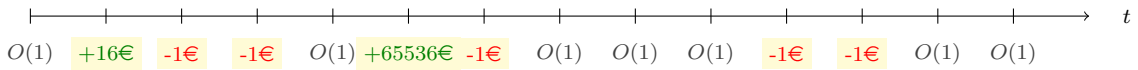


Ainsi, on remplace l'implémentation de `find` par :

pre : x
post : la racine de l'arbre qui contient x
fonction find(x)
  si $x \neq x.parent$ **alors**
    $x.parent := find(x.parent)$
  retourner $x.parent$

4 *Analyse amortie de la compression de chemin

On fait une *analyse amortie*. Certains calculs élémentaires en des nœuds ne seront pas comptabilisés dans la complexité. Mais alors le nœud paye 1 euro ; sachant qu'on donne de temps en temps de l'argent de poche aux nœuds – qui ne dépensent pas plus que ce qu'ils ont.



4.1 Rang

Définition 9 On appelle *rang* d'un nœud la hauteur qu'il aurait s'il n'y avait pas compression des chemins.

Le rang de x est stocké dans $x.h$. Dès lors qu'un nœud cesse d'être racine, son rang est fixé pour toujours.

Proposition 10 Les rangs sont entre 0 et $\log_2(n)$.

Démonstration. Par le corollaire 5. ■

Proposition 11 En remontant le long d'une branche, les rangs croient strictement.

Démonstration. Cf. comportement de l'union. ■

4.2 Puissances itérées

Définition 12 On note p_j le nombre $2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ où il y a j occurrences de "2" dans l'expression.

j	0	1	2	3	4	5	...
p_j	1	2	$2^2 = 4$	$2^{2^2} = 16$	$2^{2^{2^2}} = 65536$	$2^{2^{2^{2^2}}} = ... > 10^{19728}$	...

Définition 13 (définition inductive alternative) On définit p_j par induction sur j par :
- $p_{-1} = 0$;
- $p_j = 2^{p_{j-1}}$ pour tout $j \in \mathbb{N}$.

Remarque 14 Avec les notations des flèches de Knuth que nous n'utiliserons pas, p_j se note $2 \uparrow\uparrow j$.

4.3 Logarithme itéré

Définition 15 On définit $\log^*(h)$ le plus petit nombre j tel que :

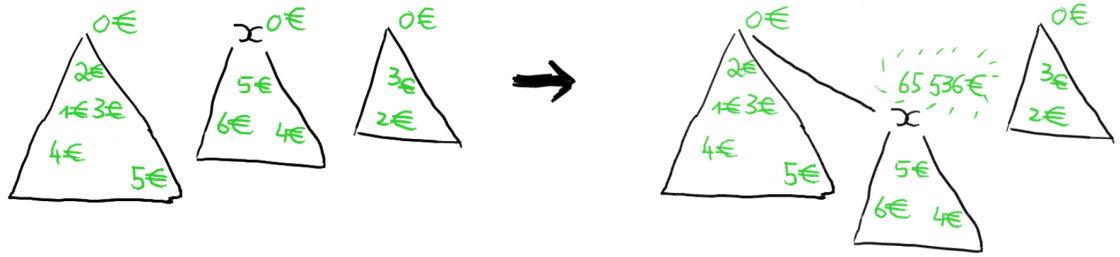
$$\underbrace{\log_2(\log_2(\cdots(\log_2}_{j \text{ fois}}(h)\underbrace{)\cdots)}_{j \text{ fois}} \leqslant 1$$

h	1	2	3	4	5	...	16	17	...	65536	65537	...	p_5	p_5+1	...
$\log^* h$	0	1	2	2	3	...	3	4	...	4	5	...	5	6	

Proposition 16 La fonction $\log^*$ donne à tout entier h le numéro $j = \log^*(h)$ tel que $h \in [p_{j-1}+1, \ldots, p_j]$.

4.4 Distribution d'argent

Au moment où un nœud x non feuille cesse d'être racine, on lui distribue $p_{\log^*(x.h)}$ euros.



Proposition 17 On distribue au plus $O(n \log^*(n))$ euros durant toute la vie de la structure de données.

DÉMONSTRATION. L'argent distribué au cours de l'utilisation de la structure de données est :

$$\text{argentDistribué} \leq \sum_{j=0}^{\ln^*(n)} p_j n_j.$$

où n_j = nb de noeuds x avec $\log^*(x.h) = j$ à la fin. Or on a :

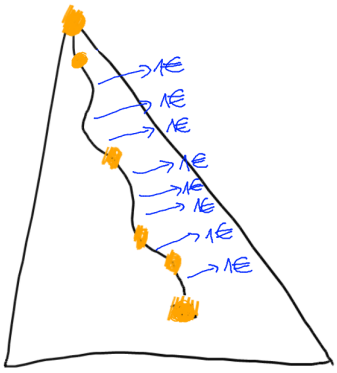
$$\begin{aligned} n_j &= \text{nombre de nœuds dont le rang est dans } [p_{j-1}+1, \ldots, p_j] \\ &\leq \text{nombre de noeuds dont le rang dans } [p_{j-1}+1, \ldots, +\infty[\\ &\leq \frac{n}{2^{p_{j-1}}} = \frac{n}{p_j} \text{par le corollaire 7} \end{aligned}$$

Ainsi :

$$\text{argentDistribué} \leq \sum_{j=0}^{\ln^*(n)} \frac{n}{p_j} p_j = n(\ln^*(n)+1).$$

■

4.5 Opérations comptabilisées VS opérations payées



Lors d'un appel à `find`, on ne comptabilise que la complexité que pour les opérations élémentaires réalisés au niveau des appel `find`(x) avec :

- $x = x.parent$ i.e. lorsque x est une racine
- ou $x.parent$ est une racine
- ou $\log^*(x.h) \neq \log^*(x.parent.h)$, i.e. les rangs ne sont pas dans les même intervalles.
- ou $x.h = 0$, i.e. x est une feuille

Pour les autres x, le noeud x paie un euro.

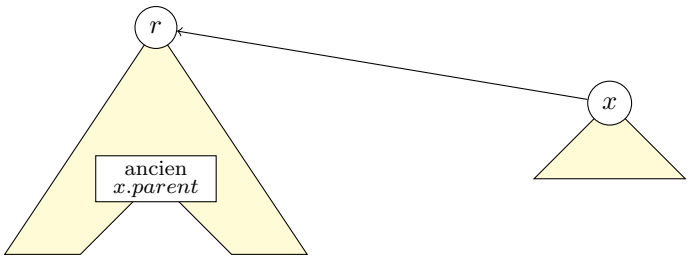
Proposition 18 Un appel à find coûte $O(\log^* n)$ opérations comptabilisées.

DÉMONSTRATION. Un appel à `find` visite une branche d'un arbre. Le nombre de nœud x le long de cette branche avec $\log^*(x.h) \neq \log^*(x.parent.h)$ est majoré par $O(\log^* n)$. D'où $O(\log^* n)$ opérations comptabilisées. ■

Lemme 19 Aucun nœud n'est "à découvert".

DÉMONSTRATION. Considérons un nœud x non feuille et non racine. Lorsqu'il a cessé d'être racine, x a reçu $p_{\log^*(x.h)}$ euros.

À chaque fois que le nœud x paie, on va le connecter à la racine courante[1] r avec $r.h > x.parent.h$. Le résultat est alors :



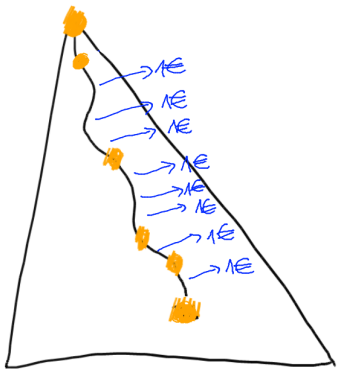
En d'autres termes, $x.parent.h$ croît strictement lorsque x paie. On a $x.h \in [p_{\log^*(x.h)-1}+1, p_{\log^*(x.h)}]$, ainsi, au bout de au plus $p_{\log^*(x.h)}$ paiements, on aura $\log^*(x.h) < \log^*(x.parent.h)$ à tout jamais, et x ne paiera plus. Donc x ne dépense pas plus que la somme qu'il a obtenu. ■

4.6 Bilan

Considérons m à appels à `find`.

Proposition 20 m appels à `find` réalisent $O(m \log^* n)$ opérations comptabilisées.

DÉMONSTRATION. Chaque appel est une remontée le long d'une des branches dans l'un des arbres :



Pour une remontée, il y a $O(\log^* n)$ opérations élémentaires comptabilisées ; les autres opérations élémentaires sont payées. Comme on réalise m appels, il y a $O(m \log^* n)$ opérations comptabilisées. ■

Proposition 21 m appels à `find` réalisent $O(n \log^*(n))$ opérations élémentaires non comptabilisées.

DÉMONSTRATION. Chaque opération élémentaire non comptabilisée est payée un euro. L'argent utilisée pour payer est bien l'argent que l'on a distribué à la structure, et pas plus (Lemme 19). Et on donne $O(n \log^*(n))$ euros à la structure (cf. Proposition 17). ■

Ainsi :

Théorème 22 m appels à `find` coûte $O((n+m) \log^*(n))$.

DÉMONSTRATION. On fait la somme des opérations élémentaires comptabilisées et celles qui ne le sont pas. ■

1. Bien sûr, cette racine r peut devenir un jour un nœud interne via `union`.