réf : Papadimitriou Algorithms p 132 → 137

UNION - FIND

Idée Gérer des partitions, notamment des classes d'équivalences, dans le cas où les classes ne font que grossir.

Type abstrait

PARTITION

CRÉER_PARTITION_VIDE () partition
CRÉER_CLASSE (P: partition, x: élmt) unité
TROUVER (P: partition, x: élmt) élmt
UNION (P: partition, x: élmt, y: élmt) unité

Implémenta° avec des arbres

↳ Une classe va être représentée par un arbre, d'arité quelconque, dont les nd seront les élmt, et la racine un représentant canonique

↳ Pour "trouver" un élmt cà-d pour identifier sa classe on donne ce rep. canonique. Il faut donc remonter à la racine. On a tout intérêt à minimiser la hauteur de ces arbres.

↳ Pour savoir comment faire l'union de deux classes avec une hauteur minimale, on a besoin de connaître la hauteur. Pour cela on ajoute à chaque noeud un rg qui doit dans un premier temps être compris comme la hauteur du sous-arbre engendré.

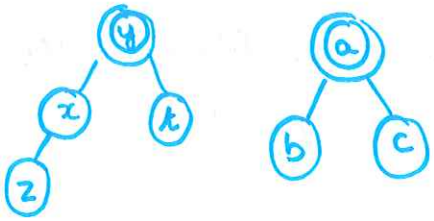
Rq une feuille est considérée de hauteur nulle.

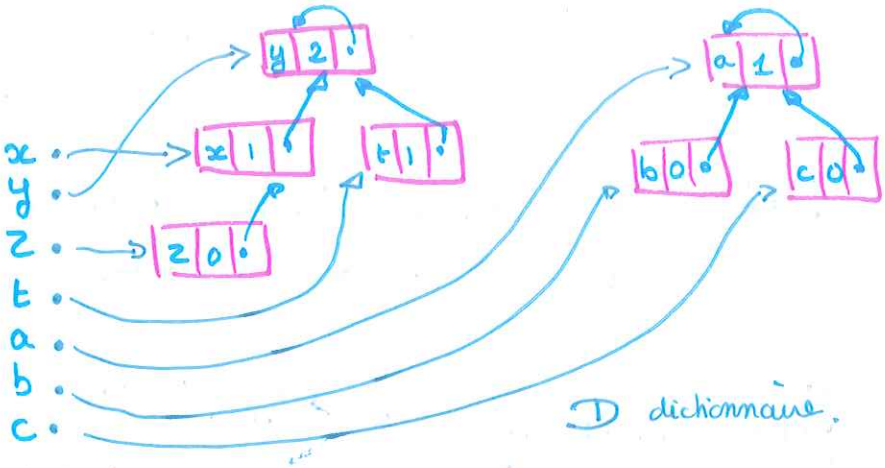
↳ Contrairement à d'habitude les nd vont ici stocker leur (unique) père, et non leurs (multiples) fils, puisqu'on va chercher à remonter l'arbre

1 noeud = 1 élmt + son rg + son père.

↳ ⚠ Cependant il faut pouvoir étant donné un élmt retrouver le nd qui correspond ce qui n'est pas évident avec cette structure ascendante. On utilise pour cela un dictionnaire.

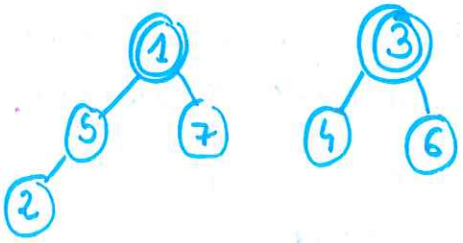
ex $P = \{\{x,y,z,t\},\{a,b,c\}\}$





↳ A partir de maintenant on se place dans le cas où les élmt sont les entiers de 1 à n, où n est fixé initialemt.
Le dictionnaire s'implémente alors simplemt comme un tableau.

ex $P = \{\{1,2,5,7\}\{3,4,6\}\}$



	1	2	3	4	5	6	7
π	1	5	3	3	1	3	1
rg	2	0	1	0	1	0	0

↳ Dans ce cadre on implémente le type abstrait par les fonctions suivantes.

CRÉER_PARTITION_VIDE (n)
- P.π = tableau de taille n
- P.rg = ———
- retourner P

CRÉER_CLASSE (P, x)
- P.π [x] = x
- P.rg [x] = 0

TROUVER (P, x)
- t = x
- tant_que P.π [t] ≠ t
  - t ← P.π [t]
- retourner t.

UNION (P, x, y)
- r_x = TROUVER (P, x)
- r_y = ——— (P, y)
- Si $r_x = r_y$ (cas 1)
  alors fini
- Si P.rg [r_x] < P.rg [r_y] (cas 2)
  alors P.π [r_x] = r_y ; fini
- Si P.rg [r_x] > P.rg [r_y] (cas 2 bis)
  alors P.π [r_y] = r_x ; fini
- // si P.rg [r_x] = P.rg [r_y]
- P.π [r_x] = r_y
- P.rg [r_y] ← P.rg [r_y] + 1. (cas 3).

UNION-FIND (SUITE 1)

Rq
- x est une racine dans P ssi $P.\Pi[x] = x$.
- Dès qu'un nd arête d'être racine son rang est fixé, et jamais plus il ne sera racine, de plus on ne lui ajoutera pas de descendants.
- Un nd obtient son rang k par union de 2 classes de "rg" $k-1$.
- Ici le rg est la hauteur du sous-arbre engendré.

Ptés
- ★1 Si x n'est pas racine, alors $P.rg[P.\Pi[x]] > P.rg[x]$ ~~on notera~~ $rg(\Pi(x)) > rg(x)$
- ★2 Chaque nd de rang k a au moins 2^k élmt dans l'arbre qu'il engendre
- ★3 Avec n_k = nbre de nd de rg k $n_k \leq n/2^k$ et $\forall x$ $rg(x) \leq \log_2(n)$

Preuve ★1. Un noeud x arrête d'être racine dans le cas 3 de UNION, moment auquel on lui affecte un nouveau père de rang un de plus. Après cela ni son rang ni son père ne changent, et le rang de son père ne peut que croître d'où ★1.

★2 La ptié est vraie au rang 0 car le noeud lui même compte (et $2^0 = 1$). Supposons la propriété vraie au rang k. Si x est un noeud de rang $k+1$, c'est qu'il représente l'union de 2 classes de rg k (rg3) c'à-d + précisémt de classes dont la racine est de rg k. Par HR chacune d'elle a au moins 2^k élmt, donc le sous arbre engendré par x a au moins $2^k + 2^k = 2^{k+1}$ élmt. On conclut à ★2 par récurrence.

★3 D'après la rg 1, lorsqu'on remonte de père en père le rang croît strictemt. Chaque nd a donc au plus un ancêtre de rang k.
En comptant tous les descendants (au sens large) des nds de rang k, on compte alors d'après ★2, au moins $n_k \times 2^k$ élmt. Or il n'y en a que n. D'où $n_k \times 2^k \leq n$. Par suite si $k > \log_2(n)$ ie si $2^k > n$ soit $n/2^k$, $n_k < 1$ donc $n_k = 0$.
Cela traduit bien qu'au plus le rg d'un nd est $\log_2(n)$.

Cplxité (a). CRÉER_PARTITION_VIDE et CRÉER_CLASSE se font en tps constant.
(b). et donc UNION se font en $O(\log_2(n))$

(a) clair (b) Corollaire direct du fait que le rang est borné par $\log_2(n)$ (★3) et donc la hauteur des arbres (d'après rg 4) aussi.

Compression de chemins

Idée Si on calcule TROUVER pour un élém^t, pourquoi le laisser en bas dans l'arbre alors qu'on peut le remonter juste sous la racine.

```
TROUVER*(P, x)
    si P.Π[x] ≠ x
    alors P.Π[x] = TROUVER*(P.Π[x])
    retourner P.Π[x]
```

. TROUVER* est définie récursivem^t

. On notera rg^* le rang obtenu en utilisant TROUVER* plutôt que TROUVER, et Π^* le père.

Rq . Pour tout élém^t x $rg^*(x) = rg(x)$

En faisant la m̂ suite d'opérations, avec TROUVER ou TROUVER*, les élém^t finissent avec le m̂ rg. C'est leur père, et donc la forme des arbres qui change.

. TROUVER* n'agit ni sur les racines, ni sur leurs fils.

. UNION à l'envers n'agit que sur les racines.

Ptés

$\star_1'$ Si x n'est pas racine, $rg^*(\Pi^*(x)) > rg^*(x)$

$\star_2'$ Chaque racine de rg^* k a au moins 2^k élém^t ds la ss arbre enraciné

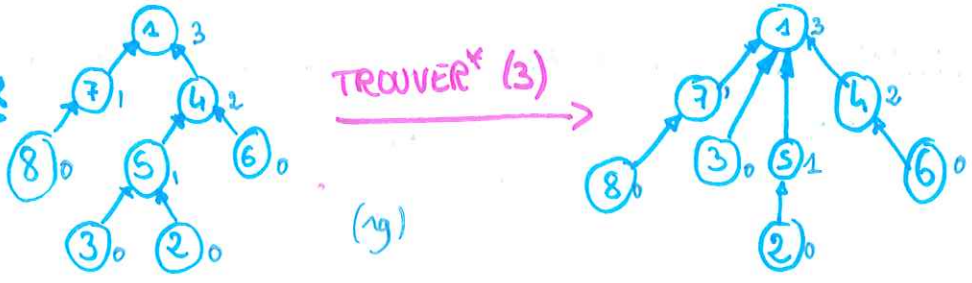
$\star_3'$ Avec n_k^* = nbre de nd de rg^* k on a $n_k^* \leq n/2^k$ et $\forall x\ rg^*(x) \leq \log_2(n)$

Preuve $\star_1'$ Le $rg^*(x)$ est le m̂ que $rg(x)$. $\Pi^*(x)$ peut être différent de $\Pi(x)$ si TROUVER*(x) a été appelé, auquel cas $\Pi(x)$ a été remplacé par la racine qui est née de rang plus grand d'après $\star_1$. Donc $rg^*(\Pi^*(x)) = rg(\Pi^*(x)) \geq rg(\Pi(x)) > rg(x) = rg^*(x)$.

$\star_2$ Ce n'est plus vrai si on n'est pas racine car l'action de TROUVER* a pu faire perdre des enfants à un nd, au profit de la racine. Mais comme TROUVER* n'affecte pas les racines, $\star_2$ reste vrai pour elles.

$\star_3$. Simplem^t parce que le $rg = rg^*$ donc $n_k = n_k^*$.

Ex



UNION-FIND (SUITE 2)

Analyse amortie

L'avantage de TROUVER* c'est qu'il profite de son exécution pour améliorer la structure, ce qui réduit la complexité des suivants. On ne peut donc, pour prendre en compte cette amélioration, se restreindre à étudier la complexité d'un appel, on doit au contraire faire une analyse de la complexité amortie.

Pté Considérons une suite d'opérations qui appelle m fois TROUVER* sans compter les appels récursifs internes.
Sa complexité est en $O(m \log^*(n)) + O(n \log^*(n))$

On comprend cette complexité comme un coût de TROUVER* en $O(\log^*(n))$, à ccl près qu'on a un surcoût en $O(n \log^*(n))$, relatif à la taille de la structure.

Pour montrer cela on utilise un modèle budgétaire

→ Chaque étape de calcul coûte 1 $.

→ TROUVER* propose un forfait de $\log^*(n)+2$ étapes par appel, qu'on paye d'avance pour les m étapes avec un budget $B_1 = m \times (\log^*(n)+2)$

→ les opérations supplémentaires éventuellement nécessaires seront payées par les nœuds concernés, auquels on aura globalement attribué un budget B_2.

. Comment répartir le budget sur les nœuds ?

Lorsqu'un nœud arête d'être racine, il a atteint son rg définitif.
Si ce rang est dans I_k où $k > 0$, on attribue $2^{\boxed{k}}$ $ à ce nœud.

. Calculer B_2

Notons a_i l'ensemble des nœuds de rg i à la fin ; $n_i = \#a_i \leq \frac{n}{2^i}$
A_k ——— de I_k ——— ; $N_k = \#A_k$.

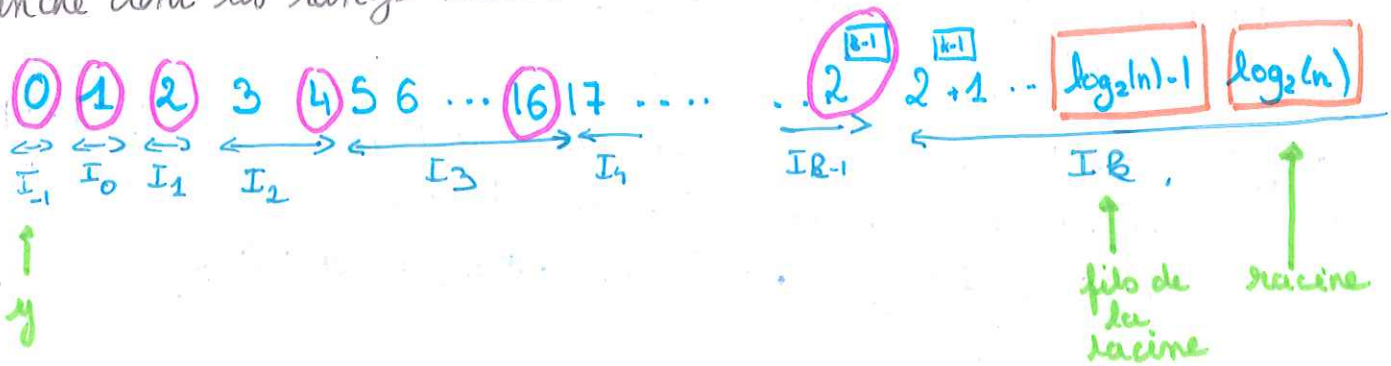
$$B_2 = \sum_{x \text{ nœud}} \text{budget}(x) = \sum_{k=-1}^{\log^*(\log_2(n))} \sum_{x \in A_k} \text{budget}(x) = \sum_{x \in A_{-1}} 0 + \sum_{x \in A_0} 0 + \sum_{k=1}^{\log^*(n)-1} \sum_{x \in A_k} \underbrace{\text{budget}(x)}_{\leq 2^{\boxed{k}}}$$

$$\leq \sum_{k=1}^{\log^*(n)-1} N_k \times 2^{\boxed{k}} \quad \text{or} \quad N_k = \sum_{i=2^{\boxed{k-1}}+1}^{2^{\boxed{k}}} n_i \leq \sum_{i=2^{\boxed{k-1}}+1}^{+\infty} \frac{n}{2^i} = \frac{n}{2^{2^{\boxed{k-1}}}} \sum_{j=1}^{+\infty} \frac{1}{2^j} = \frac{n}{2^{\boxed{k}}} \times 1.$$

D'où finalement $B_2 \leq \sum_{k=1}^{\log^*(n)-1} \frac{n}{2^{2^k}} \times 2^{2^k} \leq n \log^*(n)$.

- Vérifier qu'on pourra bien payé toutes les étapes

Considérons l'appel de TROUVER*(y).
Il demande autant d'étapes de calcul qu'il y a de nœud sur la branche remontant de y à la racine, branche le long de laquelle le rang croît strictement.
Le pire cas, celui de la branche la plus longue possible, est alors une branche dont les rangs sont :



On fait rentrer dans le forfait les calculs des nœuds dont le père a son rang dans un intervalle plus grand 0, et ceux pour le fils de la racine et la racine □.
Il y a au plus - ie dans ce pire cas - $k+1$ nœuds du 1er type où $k = \log^*(\log_2(n)) = \log^*(n) - 1$. On rentre bien dans le forfait des $\log^*(n) + 2$ calculs pour cet appel.
Il reste à s'assurer que les autres nœuds ont de quoi payer.
Ils ne sont pas racine et leur nœud est de rang dans un I_k avec $k > 0$, ils ont donc bien reçu 2^{2^k} $.
Comme à chaque fois qu'un nœud paye il change de père pour un père de rang strictement plus grand, quand un nœud n'a plus de budget son père est nécessairement de rang dans un intervalle plus grand, il sera alors dispensé de payer ie rentrera dans le forfait.

La complexité est de même que le budget

$$\underbrace{O(m \log^*(n))}_{B_1} + \underbrace{O(n \log^*(n))}_{B_2}.$$