8. Appartenance-union

UNION-FIND est l'exemple d'une structure de données simple avec une efficacité remarquable. Il s'agit d'une abstraction applicable dans beaucoup de situations :

- ⋆ connexité et propagation d'information dans réseaux un réseau (épidemiology, réseaux informatiques, interactions moléculaires, société, éléctronique),
- ⋆ maintenance de composantes connexes dans des algorithmes sur graphes (arbre couvrant minimal, plus court chemins, ancêtre commun),
- ⋆ allocation de mémoire pour variables déclarées à compilation
- ⋆ séquençage de génomes (objets = morceau de séquence, connexion = chevauchements de séquences)
- ⋆ et d'autres

La conception suit des principles génériques de structures de données : arrangement malin d'éléments dans un tableau, maintenance de variables définissant une structure «auto-organisante». La structure est aussi notable pour son analyse superbe par la méthode de crédit-débit, élaboré par Robert Tarjan.

FIG. 1: Robert Endre Tarjan (1948–)

8.1 Connexité

Problème de connexité. Beaucoup d'objets, avec connexions entre eux. Question : est-ce qu'il existe une connexion entre deux objets x, y ?

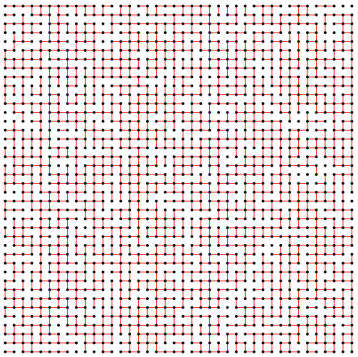

FIG. 2: Connexité : y a-t-il un chemin entre les coins opposés ?

[1] W(fr):Rélation d'équivalence

Abstraction. On veut identifier les classes d'équivalence dans l'ensemble $\{0,1,2,\ldots,n-1\}$, définies par une relation d'équivalence[1] (réfléxive, symétrique et transitive). On identifie chaque classe d'équivalence par un élément unique : son **élément canonique**, c'est à dire un entier de $\{0,1,2,\ldots,n-1\}$. Opérations :

- ⋆ $\mathsf{find}(x)$ retourne l'élément canonique de la classe de x («appartenance») : $\mathsf{find}(x) = \mathsf{find}(y)$ si et seulement si les deux appartiennent à la même classe (=connexes)
- ⋆ $\mathsf{union}(x,y)$ établit l'équivalence (connexion) entre x et y
- ⋆ intitialisation : former une classe avec le seul élément x. Au début, chaque élément forme une classe d'équivalence en soi : on initialise la structure par **for** $x \leftarrow 0,1,\ldots,n-1$ **do** $\mathsf{init}(x)$.

Exemple :

$$\mathsf{union}(3,4);\mathsf{union}(4,9);\mathsf{union}(8,0);\mathsf{union}(2,3);\mathsf{union}(7,4);\mathsf{union}(6,4);\mathsf{union}(5,0);\mathsf{find}(2);\mathsf{find}(6) \tag{8.1}$$

Solution naïve

On stocke la classe de chaque élément dans un tableau $\mathsf{id}[0..n-1]$. En conséquence, la mise à jour prend $\Theta(n)$ temps dans le code d'union.

```
INIT
    id[x] ← x
FIND(x)
    return id[x]
UNION(x, y)
    p ← FIND(x) ; q ← FIND(y)                    // déterminer la classe à travers l'interface
    if p ≠ q then                                // il faut placer tous les membres de classe p dans classe q
        for z ← 0, 1, ..., n − 1 do if id[z] = p then id[z] ← q
```

Solution avec un forêt d'arbres

Dans la deuxième solution, on représente chaque classe par un arbre enraciné à l'élément canonique. Il suffit de stocker $\mathsf{parent}[0..n-1]$; on met $\mathsf{parent}[x] = x$ à la racine. Ainsi, on peut lier les éléments canoniques de deux classes en $\Theta(1)$, et $\mathsf{find}(x)$ prend un temps proportionnel à la profondeur de x.

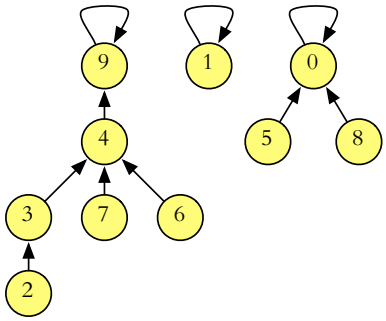

FIG. 3: Forêt d'arbres d'équivalence, après les opérations de (8.1).

```
INIT(x)
    parent[x] ← x
FIND(x)                          // chercher l'élément canonique de la classe de x
    while x ≠ parent[x] do x ← parent[x]
    return x
JOIN(p, q)   // fusionner deux classes distinctes avec éléments canoniques p ≠ q
    parent[p] ← q

UNION(x, y)
U1  p ← FIND(x) ; q ← FIND(y)
U2  if p ≠ q then JOIN(p, q)
```

Union par rang

Afin de contrôler la hauteur des arbres et ainsi assurer l'exécution rapide de find, on introduit un autre tableau $\mathsf{rang}[0..n-1]$. À tout temps, $\mathsf{rang}[x]$ est une borne supérieure sur la hauteur du sous-arbre de x.

```
INIT(x)
M1  parent[x] ← x ; rang[x] ← 0

JOIN(p, q)
J1  if rang[q] < rang[p] then échanger p ↔ q                    // assurer
    rang[p] ≤ rang[q]
J2  if rang[p] = rang[q] then rang[q] ← rang[q] + 1
J3  parent[p] ← q
```

Lemma 8.1. *Pour tout x avec $\mathsf{parent}[x] \neq x$, $\mathsf{rang}[x] < \mathsf{rang}[\mathsf{parent}[x]]$.*

Lemma 8.2. *Le nombre d'éléments dans un arbre enraciné à x est supérieur ou égal à $2^{\mathsf{rang}[x]}$, après toute séquence d'opérations* union *et* find.

Démonstration. Soit $n(x)$ le nombre d'éléments de l'arbre enraciné à x. La démonstration se fait par induction dans le nombre d'opérations d'union $m = 0, 1, \ldots$.

Cas de base. À $m = 0$, on a $\mathsf{rang}[x] = 0$ et $n(x) = 1$ pour tout x.

Hypothèse d'induction. On suppose que $n(x) \geq 2^{\mathsf{rang}[x]}$ vaut pour chaque x après $m \geq 0$ opérations d'union.

Cas inductif. Soit UNION(x, y) le $(m+1)$-ème appel à UNION. Si $p = q$ dans Ligne U2, rien ne change, donc la propriété reste vraie par l'hypothèse d'induction. Autrement, on appelle JOIN(p, q). Si $\mathsf{rang}[p] < \mathsf{rang}[q]$ dans Ligne J2, $\mathsf{rang}[q]$ ne change pas mais son sous-arbre grandit, donc après on a $n(q) > 2^{\mathsf{rang}[q]}$. Si $\mathsf{rang}[p] = \mathsf{rang}[q]$, on a (par l'hypothèse d'induction appliqué à p, q) que $n(p) + n(q) \geq 2^{\mathsf{rang}[p]} + 2^{\mathsf{rang}[q]} = 2^{1+\mathsf{rang}[q]}$.
En conclusion, le lemme est valide pour toute séquence avec $m = 0, 1, 2, \ldots$ opérations d'union. ■

Théorème 8.3. *Une opération de* $\mathsf{find}(x)$ *prend* $\Theta(\log n)$ *temps avec la heuristique d'union par rang.*

Démonstration. Par Lemme 8.1, le rang croît à chaque itération, et $0 \leq \mathsf{rang}[p] \leq \lg n$ pour tout p par Lemme 8.2. ■

Compression de chemin

On peut réduire la profondeur aussi lors d'un find par la technique de **compression de chemin**.

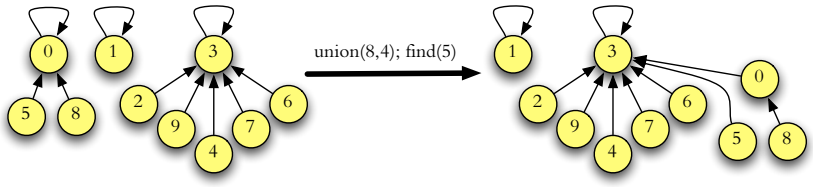

FIG. 4: Union-find avec compression de chemin.

```
FIND(x)                                          // compression par récursion
FC1  y ← parent[x]
FC2  if x ≠ y
FC3  then y ← parent[x] ← FIND(y)
FC4  return y
```

On peut éviter la récursivité avec l'astuce de **compression par réduction à moitié** (*path halving*). Cela nécessite un seul passage.

```
FIND(x)                                          // compression par réduction à moitié
FH1  y ← parent[x]
FH2  z ← parent[y]
FH3  while y ≠ z do x ← parent[x] ← z ; y ← parent[x] ; z ← parent[y]
FH4  return y
```

Temps de calcul

Si on implante union par rang, le temps de calcul de find est $O(\log n)$ au pire sans ou avec compression de chemin. Mais la compression mène à un coût amorti presque constant : $O(\log^* n)$ par opération, où $\log^*$ dénote le logarithme itéré[2] :

$$\lg^* n = \begin{cases} 0 & \text{si } n \leq 1 \\ 1 + \lg^*(\lg n) & \text{si } n > 1 \end{cases}$$

[2] W(en):*iterated logarithm*

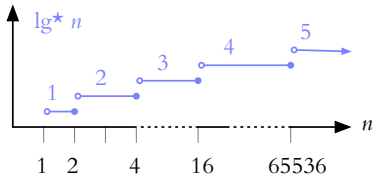

FIG. 5: Logarithme itéré : une fonction à croissance très lente (mais monotone). Noter que $\lg^* n \leq 5$ pour tout $n \leq 2^{65536}$.

Théorème 8.4. *En utilisant union-par-rang et compression de chemin (ou compression par réduction à moitié), une séquence de m opérations sur n éléments prend $O(m \lg^* n)$ temps, où $\lg^*$ dénote le logarithme itéré.*

On peut trouver une borne plus serrée que celle du Théorème 8.4 ; avec la réciproque de la fonction d'Ackermann[3].

[3] W(fr):Ackermann

Définition 8.1 (définition de Tarjan)**.** *La **fonction Ackermann** $A(i,j)$ avec $i,j \geq 1$ est définie par*

$$A(i,j) = \begin{cases} 2^j & \text{si } i = 1; \\ A(i-1,2) & \text{si } j = 1 \text{ et } i \geq 2 \\ A(i-1,A(i,j-1)) & \text{si } i,j \geq 2 \end{cases}$$

On définit la fonction Ackermann inverse ($m \geq n \geq 1$) par

$$\alpha(m,n) = \min\{i : A(i,\lfloor m/n \rfloor) \geq \lg n\}.$$

Ackermann inverse est une fonction à croissance même plus lente que $\lg^*$: $\alpha(m,n) \leq 3$ pour tout $n < 65536$ et $\alpha(m,n) \leq 4$ pour tout $n < 2^{2^{\cdot^{\cdot^{2}}}}$ (exponentiation 16 fois).

Théorème 8.5. *En utilisant union-par-rang et compression de chemin (ou compression par réduction à moitié), une séquence de m opérations sur n éléments prend $O(m\alpha(m,n))$ temps.*