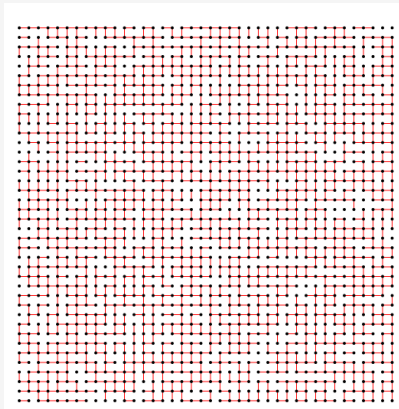
6 Appartenance-union

W(fr)

6.1 Connexité



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

- ⋆ connexité dans réseaux (ordinateurs, éléctronique, interaction moléculaire, société)
- ⋆ séquençage de génomes (objets = morceau de séquence, connexion = chevauchements de séquences)
- ⋆ équivalence de variables dans un programme FORTRAN
- ⋆ algorithmes sur graphes (algorithme de Kruskal pour l'arbre couvrant)

Abstraction. On veut identifier les classes d'équivalence dans l'ensemble $\{0, 1, 2, \ldots, n-1\}$, définies par une relation d'équivalence (réfléxive, symétrique et transitive). Opérations :

- ⋆ $\boxed{\mathsf{find}(x)}$ retourne un identificateur (unique) de la classe de x («appartenance») : $\mathsf{find}(x) = \mathsf{find}(y)$ si et seulement si les deux appartiennent dans la même classe (ils sont connexes)
- ⋆ $\boxed{\mathsf{union}(x, y)}$ établit l'équivalence (connexion) entre x et y

Exemple :

$$\mathsf{union}(3,4); \mathsf{union}(4,9); \mathsf{union}(8,0); \mathsf{union}(2,3); \mathsf{union}(7,4); \mathsf{union}(6,4); \mathsf{union}(5,0); \mathsf{find}(2); \mathsf{find}(6) \tag{6.1}$$

Une solution simple. On utilise un tableau $\mathsf{id}[0..n-1]$, On identifie chaque classe d'équivalence par un élément **canonique**. Au début, chaque élément forme une classe d'équivalence en soi :

```
INITIALISATION
for x ← 0, 1, ..., n − 1 do id[x] ← x
```

6.2 QuickF : appartenance rapide

Dans une première solution, $\mathsf{id}[x]$ donne l'élément canonique directement.

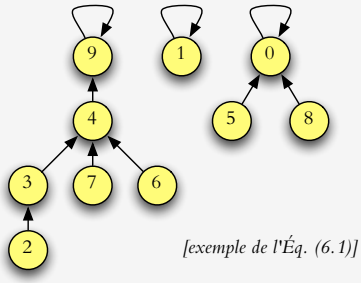
```
FIND(x)
1 return id[x]

UNION(x, y)
1 p ← FIND(x) ; q ← FIND(y)
2 if p ≠ q then
3       for z ← 0, 1, ..., n − 1 do if id[z] = p then id[z] ← q
```

L'implantation de find est aussi efficace que possible, mais l'opération union est très lente (nécessite n itérations).

6.3 QuickU : union rapide

Comment peut-on rendre union plus rapide ? Dans la deuxième solution, on utilise id différemment : au lieu de montrer directement l'élélement canonique, il dirige *vers* l'élément canonique.

1. si $\mathsf{id}[x] = x$ alors x est l'élément canonique de l'ensemble
2. sinon, faire $x \leftarrow \mathsf{id}[x]$ et retourner à 1 (ainsi, chaque classe forme un arbre)



[exemple de l'Éq. (6.1)]

```
    FIND(x)
1   while x ≠ id[x] do x ← id[x]
2   return x

    UNION(x, y)
1   p ← FIND(x) ; q ← FIND(y)
2   if p ≠ q then id[p] ← q                    // classes différentes
```

Si on est chanceux, la boucle de FIND termine rapidement, mais ce n'est pas toujours le cas (la série $\textsc{union}(1,2), \textsc{union}(2,3), \textsc{union}(3,4), \ldots$ mène à une longue liste de n éléments).

6.4 QuickUW : union rapide équilibrée

On veut garder les arbres mieux équilibrés : en Ligne 2, on devrait attacher le «plus petit» arbre au plus grand. Pour cela, on introduit un autre tableau $\mathsf{sz}[0..n-1]$ qui maintient la taille de l'arbre enraciné à chaque x. Au début, on initialise avec $\mathsf{sz}[x] \leftarrow 1$.

```
    FIND(x) comme en QuickU

    UNION(x, y)
1   p ← FIND(x) ; q ← FIND(y)
2   if p ≠ q then
3       if sz[p] < sz[q] then id[p] ← q; sz[q] ← sz[p] + sz[q]
4       else id[q] ← p; sz[p] ← sz[p] + sz[q]
```

La performance dépend de la **hauteur** de l'arbre : nombre de liens à suivre jusqu'à ce qu'on arrive à la racine. En maintenant la taille, on contrôle la hauteur des arbres ... Principe esquissé : dans le pire cas, on doit joindre deux arbres de la même taille, et la hauteur augmente par un. Est-ce qu'on peut démontrer que hauteur $\leq \lg(\mathsf{sz})$? Si oui, le nombre d'itérations en FIND est logarithmique, même dans le pire cas ! (notation : $\lg n = \log_2 n$)

Théorème 6.1. *Dans la structure QuickUW, on a*

$$\text{hauteur} \leq \lg(\mathsf{sz}) \tag{6.2}$$

pour chaque arbre, après n'importe quelle séquence d'opérations union *et* find.

Démonstration. La hauteur et la taille ne sont pas affectées par les opérations find. On démontre que l'Équation (6.2) est vraie pour chaque ensemble disjoint après m opérations union. La démonstration se fait par induction en $m = 0, 1, \ldots$.

Cas de base. À $m = 0$, on a des éléments disjoints, avec hauteur $= 0$ et $\mathsf{sz} = 1$.

Hypothèse d'induction. On suppose que (6.2) est vraie pour chaque ensemble disjoint après $m \geq 0$ opérations d'union.

Cas inductif. Soit UNION(x, y) la $(m+1)$-ème appel. Si FIND$(x) =$ FIND(y) jusqu'ici, alors rien ne change. Si FIND$(x) \neq$ FIND(y), on considère les arbres associés avec leurs classes. Par l'hypothèse d'induction, on a que $h_x \leq \lg t_x$ et $h_y \leq \lg t_y$, où h et t dénotent les hauteurs et les tailles de deux arbres. Supposons que $t_x \leq t_y$ (sans perdre la généralité). La hauteur du nouvel arbre est alors $h = \max\{h_y, 1 + h_x\}$. Par l'hypothèse d'induction,

$$h \leq \max\{\lg t_y, 1 + \lg t_x\} = \lg \max\{t_y, 2t_x\}.$$

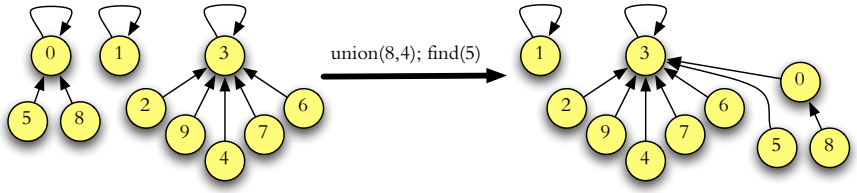
Or, la taille du nouvel arbre est $t_x + t_y \geq \max\{t_y, 2t_x\}$, Donc $h \leq \lg(t_x + t_y)$. En conséquence, le théorème reste valide après $(m+1)$ opérations. ■

6.5 Compression de chemin

Est-ce qu'on peut faire même mieux ? Oui, avec **compression de chemin**. Quand on monte jusqu'à l'élément canonique p, faire $\mathsf{id}[x] \leftarrow p$ pour tous les membres sur le chemin.

```
FIND(x)                                   // compression par itération
1  p ← id[x] ; while p ≠ id[p] do p ← id[p]
2  while x ≠ p do y ← id[x]; id[x] ← p; x ← y
3  return p

UNION(x, y) comme en QuickUW
```

```
FIND(x)                                   // compression par récursion
1  if x ≠ id[x] then id[x] ← FIND(id[x])
2  return id[x]

UNION(x, y) comme en QuickUW
```

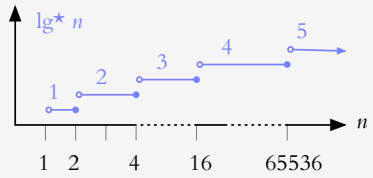


On peut même éviter la récursivité avec l'astuce de **compression de chemin par réduction à moitié** (*path halving*). Cela nécessite un seul passage mais les chemins restent deux fois plus longs.

```
FIND(x)
1  while id[id[x]] ≠ id[x] do id[x] ← id[id[x]] ; x ← id[x]
2  retourner id[x]

UNION(x, y) comme en QuickUW
```

6.6 Temps de calcul

W(en)



Logarithme itéré

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

C'est une fonction à croissance très lente (mais monotone) : $\lg^* n \leq 5$ pour tout $n \leq 2^{65536}$.

Théorème 6.2. *En utilisant la structure QuickUW avec compression de chemin, une séquence de m opérations sur n éléments prend $O(m \lg^* n)$ temps, où $\lg^*$ dénote le logarithme itéré.*

Théorème 6.3. *Avec la structure de QuickUW avec compression de chemin, les opérations s'exécutent individuellement en $O(\log n)$ au pire cas, et en $O(\log^* n)$ temps amorti.*

W(fr)

REMARQUE. On peut trouver une borne plus serrée que celle du Théorème 6.2 ; avec la réciproque de la fonction d'Ackermann.

Définition 6.1 (définition de R. E. Tarjan)**.** *La **fonction Ackermann** $A(i, j)$ avec $i, j \geq 1$ est définie par*

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

Définition 6.2. *Ackermann inverse ($m \geq n \geq 1$)*

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

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 $n < 2^{2^{\cdots^2}}$ (exponentiation 16 fois).

Théorème 6.4. *En utilisant la structure QuickUW avec compression de chemin, une séquence de m opérations sur n éléments prend $O(m\alpha(m, n))$ temps.*