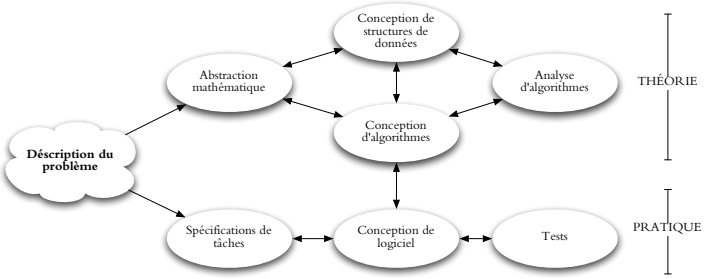
IFT2015 hiver 2010 — Notes de cours

Miklós Csűrös

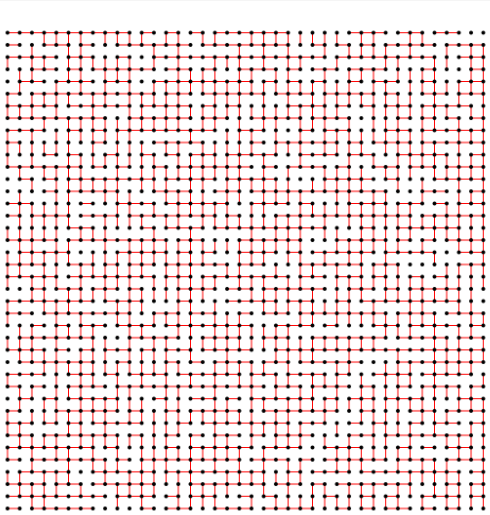
6 janvier 2010

1 Union-Find

Informatique. On va illustrer la branche théorique par l'exemple de connexité.



Connexité



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

Opérations abstraites.

⋆ FIND(x) trouve l'ensemble contenant x («appartenance»)

⋆ UNION(x, y) remplace les ensembles de x et y par leur union

⋆ MAKESET(x) crée un ensemble avec le seul élément x

Exemple :

$$\begin{aligned}&\text{UNION}(3,4);\ \text{UNION}(4,9);\ \text{UNION}(8,0);\ \text{UNION}(2,3);\\&\text{UNION}(7,4);\ \text{UNION}(6,4);\ \text{UNION}(5,0);\\&\text{FIND}(2);\ \text{FIND}(6)\end{aligned} \tag{1}$$

Connexité : on sait que FIND(p) = FIND(q) si et seulement si les deux sont connexes.

Une solution simple. on réprésente un ensemble par un élément **canonique**.

Structure de données : stocker $\mathsf{id}[x]$ — c'est l'ensemble auquel x appartient

```
    MAKESET(x)
1   id[x] ← x
```

Pour simplifier la discussion d'une structure de données, on traite souvent les éléments comme des entiers $0, 1, \ldots, N-1$. Néanmoins, des implantations équivalentes mais plus générales sont faciles à imaginer.

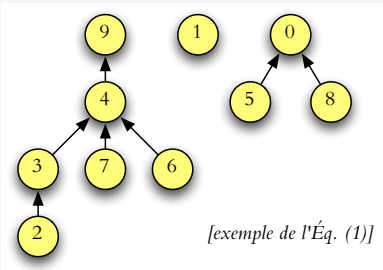
```
public class MonUF
{
    public static class Element
    {
        private Element id;
        private Object contenu;
    }

    public Element makeset(Object contenu)
    {
        Element emt = new Element();
        emt.id = emt;
        emt.contenu = contenu;
        return emt;
    }
    ...
}
```

QuickF — appartenance rapide . . . union lente

```
    UNION(x, y)
1   si id[x] ≠ id[y] alors
2       pour tout z
3           si id[z] = id[x] alors id[z] = id[y]
    FIND(x)
1   retourner id[x]
```

QuickU — union rapide. Idée : on utilise id différemment.

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




```
    UNION(x, y)
1   p ← FIND(x) ; q ← FIND(y)
2   si p ≠ q alors
3       id[p] ← q
    FIND(x)
1   tandis que x ≠ id[x] faire x ← id[x]
2   retourner x
```

. . . appartenance lente (appels consécutifs $\textsc{union}(1,2), \textsc{union}(2,3), \textsc{union}(3,4), \ldots$ mènent essentiellement à une liste chaînée)

QuickUW — union rapide équilibrée. Idée : au lieu de toujours faire $\mathsf{id}[x] \leftarrow y$, lors d'un appel $\textsc{union}(x, y)$, on examine d'abord la taille des deux arbres. Taille stockée dans $\mathsf{taille}[x]$

```
    MAKESET(x)
1   id[x] ← x
2   taille[x] ← 1
```

```
UNION(x, y)
1 p ← FIND(x) ; q ← FIND(y)
2 si p ≠ q alors
3     si taille[p] < taille[q]
4     alors id[p] ← q; taille[q] ← taille[p] + taille[q]
5     sinon id[q] ← p; taille[p] ← taille[p] + taille[q]

FIND(x)
1 tandis que x ≠ id[x] faire x ← id[x]
2 retourner x
```

avantage : arbres équilibrés (on arrive à la racine en suivant $\leq \lg k$ liens dans un ensemble de k éléments)

Analyse de QuickUW. Performance de FIND 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 $\leq \lg(\mathsf{taille})$? (notation : $\lg n = \log_2 n$)

Théorème 1. *Dans la structure QuickUW, on a*

$$\text{hauteur} \leq \lg(\textit{taille}) \tag{2}$$

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 (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 $\mathsf{taille} = 1$.

Hypothèse d'induction. On suppose que (2) est vraie pour chaque ensemble disjoint après $m \geq 0$ opérations d'UNION.

Cas inductif. Soit UNION(x, y) la $(m+1)$-ème opération. Si FIND$(x) =$ FIND(y) jusqu'ici, alors rien ne change. Si FIND$(x) \neq$ FIND(y), on considère les arbres associés avec leurs ensembles. 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. ■

Compression de chemin. Si on a n éléments (n appels de MAKESET) et une série de m appels UNION/FIND

⋆ QuickF utilise nm opérations

⋆ QuickU utilise $nm/2$ opérations

⋆ QuickUW utilise $m \lg n$ opérations

Est-ce qu'on peut faire mieux ? Idée : **compression de chemin**. Quand on monte jusqu'à la racine, faire $\mathsf{id}[x] \leftarrow \mathsf{racine}$ pour tous les membres sur le chemin. (On modifie QuickUW pour équilibrer les arbres toujours.)

```
    FIND(x)
1   si x ≠ id[x] alors id[x] ← FIND(id[x])
2   retourner id[x]
```

On a besoin de deux passages . . . Autre idée : **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   tandis que id[id[x]] ≠ id[x] faire id[x] ← id[id[x]] ; x ← id[x]
2   retourner id[x]
```

Logarithme itéré. Composition : $\lg\lg n = \lg(\lg(n))$.

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 2. *En utilisant la structure QuickUW avec compression de chemin, une séquence de m opérations sur n éléments prend un nombre d'instructions élémentaires de* l'ordre de $m \lg^* n$.

- ⋆ Ici, «instruction élémentaire» veut dire des opérations simples comme addition, affectation, etc.
- ⋆ On verra la définition précise de l'«ordre de» bientôt. (Ce qu'on dénotera par $O\big(m \lg^* n\big)$.) Cela veut dire que le nombre d'instructions executées est $\leq c \cdot m \lg^* n$ avec une constante quelconque c, indépendante de m, n.
- ⋆ On a donc un théorème sur le temps d'exécution sur n'importe quel CPU.

Fonction d'Ackermann.

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

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

Définition 2. *Ackermann inverse ($m \geq n \geq 1$)*

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

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 3. *En utilisant la structure QuickUW avec compression de chemin, une séquence de m opérations sur n éléments prend un nombre d'instructions élémentaires de* l'ordre de $m\alpha(m,n)$.