INFO602, L3 Informatique, Algorithmique II
Lesson 3, Structures de données pour ensembles disjoints (Union-Find)

Jacques-Olivier Lachaud
LAMA, Université Savoie Mont blanc
https://www.lama.univ-savoie.fr/wiki
(suivre INFO602)

21 mars 2020

1 Structures de données pour ensembles disjoints (Union-Find)

Une structure de données d'ensembles disjoints gère une collection $S = \{S_1, \ldots, S_k\}$ d'ensembles disjoints (les ensembles sont tous deux à deux disjoints) et dynamiques (les ensembles peuvent être fusionnés ou découpés au ours du temps). Chaque ensemble S_i est identifié par un *représentant* qui appartient à l'ensemble. Savoir si deux éléments appartiennent au même ensemble revient donc à déterminer s'ils ont le même représentant.

Ces structures de données permettent de représenter les classes d'équivalence dans les relations. Une application courante est la détermination des composantes connexes d'un graphe. Une autre est de déterminer si l'ajout d'une arête à un arbre couvrant induit un cycle.

Exemples :

— On examine un ensemble de n personnes et on veut le décomposer en groupes de personnes appartenant à la même ville (une relation d'équivalence). On veut ensuite les regrouper par intercommunalité, puis par département, et enfin région. Les structures pour ensembles disjoints sont efficaces pour ces requêtes.

— L'algorithme de Kruskal de calcul de l'arbre couvrant de poids minimal utilise aussi cette structure. Elle permet de détecter si l'ajout d'une arête crée un cycle dans un graphe, simplement en regardant si ses deux extrémités sont déjà dans le même ensemble.

— Segmenter une image est le processus est de rassembler ses pixels en des régions d'intérêt. Les pixels d'une image sont vus sous forme d'un graphe, où les pixels sont ses sommets et les arêtes relient deux pixels voisins. Un découpage en régions d'une image est donc une collection d'ensemble disjoints. Souvent le processus de segmentation part de l'ensemble disjoints de tous ses pixels, puis rassemble progressivement les régions ressemblantes entre elles. Les structures pour ensembles disjoints sont donc aussi bien adaptées à ce processus.

1.1 Opérations requises sur ces structures

Les ensembles sont composées d'éléments appelés objets. Chaque ensemble est représenté par l'un de ses objets, qui le caractérise puisque tous les ensembles sont disjoints deux à deux. On note que l'on ne manipulera qu'une seule structure par ensemble disjoint à la fois, elle sera donc implicite dans les fonctions ci-dessous. On veut disposer des opérations suivantes :

— **Action** CRÉER-ENSEMBLE(x) crée un nouvel ensemble dont le seul membre et représentant est x. Attention, il faut que x ne soit pas déjà dans un autre ensemble.

— **Action** UNION(x, y) réunit les ensembles dynamiques qui contiennent x et y, mettons S_x et S_y, dans un nouvel ensemble qui est l'union des deux. Le représentant de ce nouvel ensemble est un élément quelconque de $S_x \cup S_y$ (c'est souvent l'un des représentants de S_x ou S_y). Les ensembles S_x et S_y n'existent plus dans la collection d'ensembles S.

— **Fonction** TROUVER-ENSEMBLE*(x)* retourne le représentant de l'unique ensemble contenant l'objet x.

On pourrait vouloir diviser des ensembles, mais cela n'induit pas de difficultés algorithmiques spécifiques. En gros, la division d'un ensemble en un certain nombre de sous-ensembles est proportionnel à son nombre d'éléments (car il faut les désigner). On ne se préoccupe pas de ce type d'opérations ici.

Dans toute la suite, l'analyse des algorithmes se fera en fonction du nombre n d'appels à CRÉER-ENSEMBLE et de m le nombre total d'opérations CRÉER-ENSEMBLE, UNION et TROUVER-ENSEMBLE. Ce sera donc de l'analyse *amortie*.

Remarque : d'un point de vue pratique, en C un objet est un pointeur vers la structure modélisant un élément, en C++ c'est un pointeur ou une référence vers cette structure, en JAVA, ce serait une variable objet (donc un pointeur vers l'instance). On verra que chaque élément stocke au moins sa valeur et un lien/pointeur vers un autre objet.

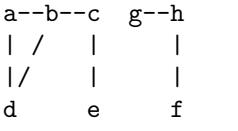
1.2 Le calcul des composantes connexes

Il s'agit de l'application la plus connue des structures pour ensembles disjoints. Etant donné un graphe, on veut savoir le nombre de composantes connexes, ou effectuer des réquêtes pour savoir si deux sommets sont dans la même composante connexe. Une approche serait de faire un parcours en largeur à partir d'un sommet, mais cette approche devient très vite coûteuse si on a plusieurs couples de sommets à tester. Il est alors plus intéressant de construire les ensembles disjoints qui correspondent aux composantes connexes du graphe. De plus, si on rajoute des arêtes au graphe, ces structures sont mises à jour grâce à des appels à UNION. L'algorithme basé sur les ensembles disjoints s'écrit ainsi. On fait d'abord le précalcul avec l'appel de COMPOSANTES-CONNEXES, puis on appelle autant de fois que l'on souhaite MÊME-COMPOSANTE.

```
// Calcule les ensembles disjoints correspondant aux composantes connexes d'un
   graphe
Action COMPOSANTES-CONNEXES( E G : Graphe );
Var : u,v : Sommet
début
    Pour chaque sommet v de G Faire
        CRÉER-ENSEMBLE(v);
    Pour chaque arête (u, v) de G Faire
        si TROUVER-ENSEMBLE(u) ≠ TROUVER-ENSEMBLE(v) alors
            UNION(u, v);
```

```
// Retourne vrai si u et v sont dans la même composante
Fonction MÊME-COMPOSANTE( E u, v : Sommet ) : booléen ;
début
    Retourner TROUVER-ENSEMBLE(u) = TROUVER-ENSEMBLE(v) ;
```

L'exécution de COMPOSANTES-CONNEXES sur le graphe ci-dessous est faite pas à pas dans la Table 1.



Arêtes testées	Ensembles disjoints							
au début	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$
(b,a)	$\{a,b\}$		$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$
(d,a)	$\{a,b,d\}$		$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$
(a,b)	$\{a,b,d\}$		$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$
(c,b)	$\{a,b,c,d\}$				$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$
(d,b)	$\{a,b,c,d\}$				$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$
(b,c)	$\{a,b,c,d\}$				$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$
(e,c)	$\{a,b,c,d,e\}$					$\{f\}$	$\{g\}$	$\{h\}$
(a,d)	$\{a,b,c,d,e\}$					$\{f\}$	$\{g\}$	$\{h\}$
(b,d)	$\{a,b,c,d,e\}$					$\{f\}$	$\{g\}$	$\{h\}$
(c,e)	$\{a,b,c,d,e\}$					$\{f\}$	$\{g\}$	$\{h\}$
(h,f)	$\{a,b,c,d,e\}$					$\{f,h\}$	$\{g\}$	
(h,g)	$\{a,b,c,d,e\}$					$\{f,g,h\}$		
(f,h)	$\{a,b,c,d,e\}$					$\{f,g,h\}$		
(g,h)	$\{a,b,c,d,e\}$					$\{f,g,h\}$		

TABLE 1 – Exécution de COMPOSANTES-CONNEXES sur le graphe non orienté $G = \{\{a,b\},\{a,d\},\{b,c\},\{c,e\},\{g,h\},\{h,f\}\}$.

Dans une implémentation réelle, il faudrait que l'objet représentant un sommet ait un pointeur vers l'objet correspondant dans la structure pour ensembles disjoints et vice-versa. Nous ignorons ces détails ici.

Exercices

1. Adaptez l'action COMPOSANTES-CONNEXES pour que ce soit une fonction qui retourne aussi le nombre de composantes connexes du graphe G.

1.3 Ensembles disjoints par listes chaînées

Chaque ensemble de S est une liste chaînée de ses objets. Le premier élément de la liste est le représentant de l'ensemble. Chaque élément a un pointeur vers son successeur et un pointeur vers le représentant de la liste (donc le premier élément de la liste). De plus chaque liste a un pointeur vers le premier élément plus un pointeur vers le dernier élément (pour concaténer les listes lors d'un UNION).

On note immédiatement que :

— Tout appel à CRÉER-ENSEMBLE coûte $\Theta(1)$ car il suffit de créer une liste à un élément.

— Tout appel à TROUVER-ENSEMBLE coûte $\Theta(1)$ car chaque élément a un pointeur vers son représentant. Il suffit donc de le consulter.

— Si UNION(x,y) est implémentée en déplaçant la liste de x à la fin de la liste de y, il faut mettre à jour les représentants de la liste de x. Cela prend donc un temps $\Theta(k)$ où k est la taille de la liste de x.

Il est facile de voir que $n = m/2$ CRÉER-ENSEMBLE suivis de $n-1$ UNION donne un coût total en $\Theta(n^2)$, donc un coût amorti en $\Theta(n)$.

Actions	Ensembles disjoints	Coût
$\vdots$	$(x_1)(x_2)(x_3)(x_4)\dots(x_{n-1})$	1
CRÉER-ENSEMBLE(x_n)	$(x_1)(x_2)(x_3)(x_4)\dots(x_{n-1})(x_n)$	1
UNION(x_1,x_2)	$(x_2,x_1)(x_3)(x_4)\dots(x_{n-1})(x_n)$	1
UNION(x_1,x_3)	$(x_3,x_2,x_1)(x_4)\dots(x_{n-1})(x_n)$	2
UNION(x_1,x_4)	$(x_4,x_3,x_2,x_1)\dots(x_{n-1})(x_n)$	3
$\vdots$	$\vdots$	$\vdots$
UNION(x_1,x_{n-1})	$(x_{n-1},\dots,x_4,x_3,x_2,x_1)(x_n)$	$n-2$
UNION(x_1,x_n)	$(x_n,x_{n-1},\dots,x_4,x_3,x_2,x_1)$	$n-1$

On peut baisser cette borne en utilisant l'*heuristique de l'union pondérée*. Le principe est le suivant. Chaque liste stocke aussi son nombre d'éléments. Lors d'un appel UNION(x, y) c'est toujours la liste la plus courte qui est concaténée à la plus longue. Grâce à ce petit test tout simple, on montre que :

Théorème 1 *Si les ensembles disjoints sont représentés par listes chaînées et qu'on utilise l'heuristique de l'union pondérée, une séquence arbitraire de m opérations* CRÉER-ENSEMBLE, TROUVER-ENSEMBLE *et* UNION, *dont n* CRÉER-ENSEMBLE *prend un temps $\mathcal{O}(m + n \log n)$.*

Preuve. On utilise la méthode de l'agrégat.

$$\begin{aligned} T_{total}(m) = & \underbrace{\text{CRÉER-ENSEMBLE} + \cdots + \text{CRÉER-ENSEMBLE}}_{\Theta(n)} \\ & + \underbrace{\text{TROUVER-ENSEMBLE} + \cdots + \text{TROUVER-ENSEMBLE}}_{\leq (m-n)\Theta(1)} \\ & + \underbrace{\text{UNION} + \cdots + \text{UNION}}_{\leq (m-n)\times \mathcal{O}(k)} \end{aligned}$$

Il suffit de calculer un majorant du nombre de fois où le pointeur vers le représentant d'un objet x est changé. Or chaque fois qu'il est mis à jour, l'objet x était dans le plus petit des ensembles. Donc la première fois, au plus petit, il était tout seul et a rejoint un ensemble de taille au moins 2. La deuxième fois, il rejoint un ensemble de taille au moins 4, etc, jusqu'à au maximum $\lceil \log_2 n \rceil$ fois (car l'ensemble le plus grand a une taille n maximum). □

1.4 Ensembles disjoints par forêts

Chaque ensemble sera maintenant représenté par un arbre, dont les nœuds sont les objets appartenant à l'ensemble et la racine est le représentant de l'ensemble. Chaque nœud d'un arbre a donc un pointeur vers son parent. La racine pointe vers elle-même. Les fonctions sur les ensembles disjoints sont mis en œuvre ainsi :

- Un appel de CRÉER-ENSEMBLE crée simplement un arbre a un seul nœud (coûte $\Theta(1)$).
- La fonction TROUVER-ENSEMBLE remonte d'un nœud jusqu'à sa racine, ce qui coûte $\Theta(k)$ où k est la profondeur du nœud.
- La fonction UNION(x, y) relie la racine de x à la racine de y. Pour ce faire, il suffit donc de remonter à la racine de chaque arbre et de modifier un pointeur.

```
// Ensembles disjoints par forêts sans optimisation
Action CRÉER-ENSEMBLE( E x : Objet ) ;
début
    x.pere ← x;
Action UNION( E x, y : Objet ) ;
début
    y.pere ← x
Fonction TROUVER-ENSEMBLE( E x : Objet ) : Objet ;
début
    si x ≠ x.pere alors
        Retourner TROUVER-ENSEMBLE(x.pere)
    else Retourner x;
```

Tel quel, une telle implémentation n'est pas plus rapide que la précédente. On applique deux heuristiques à cette structure de façon à la rendre la plus efficace possible.

L'union par rang. On stocke dans chaque nœud un majorant de la hauteur de son sous-arbre appelé *rang* (qui vaut 1 lorsque l'arbre est réduit à un élément). Lorsqu'on réalise l'union, c'est

la racine de moindre rang qui pointe vers la racine de rang supérieur. C'est seulement lorsque les deux racines ont même rang qu'on augmente le rang. Il est clair que cette opération ne rajoute pas de surcoût en temps.

La compression de chemin. Dès que l'on utilise TROUVER-ENSEMBLE, tous les nœuds traversés pour trouver la racine, sont modifiés de façon à ce que leur parent soit directement la racine. Il est clair que cette opération ne rajoute pas de surcoût en temps.

On stocke le rang et le père d'un objet x dans des champs de x. Cela donne les pseudo-codes suivants :

```
Action CRÉER-ENSEMBLE( E x : Objet ) ;
début
    x.pere ← x;
    x.rang ← 1;
Action UNION( E x, y : Objet ) ;
début
    LIER(TROUVER-ENSEMBLE(x),TROUVER-ENSEMBLE(y));
Action LIER( E x, y : Objet ) ;                    // x et y sont des racines.
début
    si x.rang > y.rang alors y.pere ← x;
    else
        x.pere ← y ;
        si x.rang = y.rang alors y.rang ← y.rang + 1;
Fonction TROUVER-ENSEMBLE( E x : Objet ) : Objet ;
début
    si x ≠ x.pere alors
        x.pere ← TROUVER-ENSEMBLE(x.pere)
    Retourner x.pere
```

Ces heuristiques ont un effet très important sur la complexité amortie des opérations sur les ensembles disjoints. On a [Tarjan 1975] :

Théorème 2 *Si on utilise les forêts d'ensembles disjoints avec les heuristiques d'union par rang et de compression de chemin, alors une séquence arbitraire de m opérations* CRÉER-ENSEMBLE, TROUVER-ENSEMBLE *et* UNION, *dont n* CRÉER-ENSEMBLE *prend un temps $\mathcal{O}(m\alpha(n))$, où $\alpha(n)$ est une fonction qui croît extrêmement lentement ($n \leq 4$ dans tous les cas concevables).*

Preuve. Nous ne démontrerons pas cette borne. La preuve se fait en utilisant la méthode du potentiel. □

Qu'est-ce que $\alpha(n)$ et que vaut-il ? On définit d'abord la fonction à croissance très rapide ci-dessous :

$$\forall k \geq 0, j \geq 1, \quad A_k(j) = \begin{cases} j+1 & \text{si } k = 0, \\ A_{k-1}^{(j+1)}(j) & \text{si } k \geq 1. \end{cases} \tag{1}$$

On note qu'on utilise la notation exponentielle (j) pour exprimer qu'on affectue j la composée de la fonction. Le paramètre k est appelé *niveau* de la fonction A.

On montre facilement que :

— $\forall j \geq 1, A_1(j) = A_0^{(j+1)}(j) = A_0(A_0(\dots A_0(j+1)\dots)) = 2j+1$.

— $\forall j \geq 2, A_2(j) = A_1^{(j+1)}(j) = 2^{j+1}(j+1) - 1$.

— $A_3(1) = A_2^{(2)}(1) = A_2(A_2(1)) = A_2(7) = 2^8 \cdot 8 - 1 = 2047$

— $A_4(1) = A_3^{(2)}(1) = A_3(A_3(1)) = A_3(2047) = A_2^{(2048)}(2047) \gg A_2(2047) = 2^{2048} \cdot 2048 - 1$.

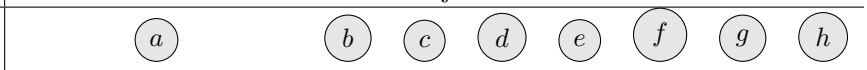
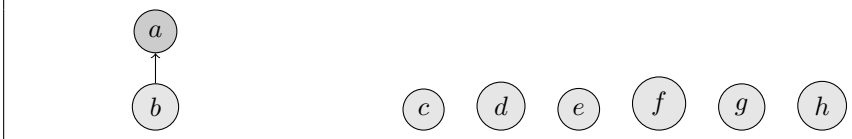
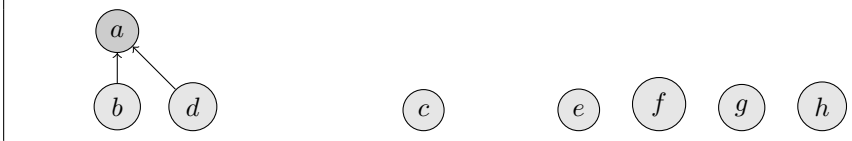
Arêtes testées	Ensembles disjoints
au début	
(b, a)	
(d, a)	
(a, b)	
(c, b)	
(d, b)	
(b, c)	
(e, c)	

TABLE 2 – Une partie de l'exécution de COMPOSANTES-CONNEXES sur le graphe non orienté $G = \{\{a, b\}, \{a, d\}, \{b, c\}, \{c, e\}, \{g, h\}, \{h, f\}\}$, avec les forêts d'ensembles disjoints. Le rang est donné par le niveau de gris.

j	1	2	3	4	5
$A_0(j)$	2	3	4	5	6
$A_1(j)$	3	5	7	9	11
$A_2(j)$	7	23	63	159	383
$A_3(j)$	2047	$\gg 2^{2^{27}}$			
$A_4(j)$	$\gg 2^{1000000}$				

La fonction $\alpha(n)$ est définie comme $\min\{k, A_k(1) \geq n\}$. Autrement dit :

$$\alpha(n) = \begin{cases} 0 & \text{pour } 0 \leq n \leq 2, \\ 1 & \text{pour } n = 3, \\ 2 & \text{pour } 4 \leq n \leq 7, \\ 3 & \text{pour } 8 \leq n \leq 2047, \\ 4 & \text{pour } 2048 \leq n \leq A_4(1). \end{cases}$$

En pratique, les opérations ont donc un coût amorti constant (même si mathématiquement, $\alpha(n)$ tend vers l'infini).

La Table 2 donne une partie de l'exécution de l'algorithme COMPOSANTES-CONNEXES avec des forêts d'ensembles disjoints. On voit que la profondeur des arbres n'augmente vraiment pas.