Union-Find

En informatique, **union-find** est une structure de données qui représente une partition d'un ensemble (ou de manière équivalente relation d'équivalence). Elle a essentiellement deux opérations *trouver* et *unir*, on l'appelle *union-find*, suivant en cela la terminologie anglaise :

- **Find** (trouver) : détermine la classe d'équivalence d'un élément ; elle sert aussi à déterminer si deux éléments appartiennent à la même classe d'équivalence.
- **Union** (unir) : réunit deux classes d'équivalence en une seule.

Une autre opération importante, **MakeSet**, construit une classe d'équivalence contenant un seul élément, autrement dit un singleton.

Partition avec 8 classes (qui sont des singletons) obtenue avec **MakeSet(1), ..., MakeSet(8)**

Partition avec 3 classes disjointes obtenue après **Union(1, 2), Union(3, 4), Union(2, 5), Union(1, 6)** et **Union(2, 8)**.

Afin de définir ces opérations plus précisément, il faut choisir un moyen de représenter les classes. L'approche traditionnelle consiste à sélectionner un élément particulier de chaque classe, appelé le *représentant*, pour identifier la classe entière. Lors d'un appel, **Find**(x) retourne le représentant de la classe de *x*.

Sommaire

Implémentation utilisant des listes chaînées

La solution la plus immédiate pour le problème des classes disjointes consiste à construire une liste chaînée pour chaque classe. On choisit alors l'élément en tête de liste comme représentant.

MakeSet crée une liste contenant un élément. **Union** concatène les deux listes, une opération en temps constant. Malheureusement, l'opération **Find** nécessite alors un temps $\Omega(n)$ avec cette approche.

On peut y remédier en ajoutant à chaque élément d'une liste chaînée un pointeur vers la tête de la liste ; **Find** est alors réalisée en temps constant (le représentant étant la tête de la liste). Cependant, on a détérioré l'efficacité de **Union**, qui doit maintenant parcourir tous les éléments d'une des deux listes pour les faire pointer vers la tête de l'autre liste, ce qui nécessite un temps $\Omega(n)$.

On peut améliorer ceci en ajoutant toujours la plus petite des deux listes en queue de la plus longue, ce qui porte le nom d'*heuristique de l'union pondérée*. Ceci nécessite de maintenir la longueur des listes au fur et à

mesure des opérations. Avec cette solution, une séquence de *m* opérations **MakeSet**, **Union** et **Find** sur *n* éléments nécessite un temps O(*m* + *n*log *n*). Pour obtenir de meilleurs résultats, nous devons considérer une structure de données différente.

Implémentation utilisant des forêts

On considère maintenant une structure de données dans laquelle chaque classe est représentée par un arbre dans lequel chaque nœud contient une référence vers son nœud père. Une telle structure de forêt a été introduite par Bernard A. Galler et Michael J. Fisher en 1964[1], bien que son analyse détaillée ait attendu plusieurs années.

Dans une telle forêt, le représentant de chaque classe est la racine de l'arbre correspondant. **Find** se contente de suivre les liens vers les nœuds pères jusqu'à atteindre la racine. **Union** réunit deux arbres en attachant la racine de l'un à la racine de l'autre. Une manière d'écrire ces opérations est la suivante :

```
fonction MakeSet(x)
    x.parent := null

fonction Find(x)
    si x.parent == null
       retourner x
    retourner Find(x.parent)

fonction Union(x, y)
    xRacine := Find(x)
    yRacine := Find(y)
    si xRacine ≠ yRacine
           xRacine.parent := yRacine
```

Sous cette forme naïve, cette approche n'est pas meilleure que celle des listes chaînées, car les arbres créés peuvent être très déséquilibrés. Mais elle peut être améliorée de deux façons.

La première consiste à toujours attacher l'arbre le plus petit à la racine de l'arbre le plus grand, plutôt que l'inverse. Pour évaluer quel arbre est le plus grand, on utilise une heuristique appelée le *rang*[2] : les arbres contenant un élément sont de rang zéro, et lorsque deux arbres de même rang sont réunis, le résultat a un rang plus grand d'une unité. Avec cette seule amélioration, la complexité amortie des opérations **MakeSet**, **Union** et **Find** devient $O(\log n)$. Voici les nouveaux codes de `MakeSet` et `Union` :

```
fonction MakeSet(x)
    x.parent := x
    x.rang   := 0

fonction Union(x, y)
    xRacine := Find(x)
    yRacine := Find(y)
    si xRacine ≠ yRacine
          si xRacine.rang < yRacine.rang
                xRacine.parent := yRacine
          sinon
                yRacine.parent := xRacine
                if xRacine.rang == yRacine.rang
                        xRacine.rang := xRacine.rang + 1
```

La seconde amélioration, appelée *compression de chemin*, consiste à profiter de chaque **Find** pour aplatir la structure d'arbre, en effet, chaque nœud rencontré sur le chemin menant à la racine peut être directement relié à celle-ci; car tous ces nœuds ont le même ancêtre. Pour réaliser cela, on fait un parcours vers la racine, afin de la déterminer, puis un autre parcours pour faire de cette racine la mère de tous les nœuds rencontrés en chemin. L'arbre résultant ainsi aplati améliore les performances des futures recherches d'ancêtre, mais profite aussi aux autres nœuds pointant vers ceux-ci, que ce soit directement ou indirectement. Voici l'opération `Find` améliorée :

```
fonction Find(x)
    si x.parent ≠ x
       x.parent := Find(x.parent)
    retourner x.parent
```

Ces deux améliorations (fusion optimisée des racines et compression de chemin) sont complémentaires. Ensemble, elles permettent d'atteindre une complexité amortie en temps $O(\alpha(n))$, où $\alpha(n)$ est la réciproque de la fonction $f(n) = A(n, n)$ et A la fonction d'Ackermann dont la croissance est extrêmement rapide. $\alpha(n)$ vaut moins de 5 pour toute valeur n en pratique[3]. En conséquence, la complexité amortie par opération est de fait une constante.

Il n'est pas possible d'obtenir un meilleur résultat : Fredman et Saks ont montré en 1989 que $\Omega(\alpha(n))$ mots en moyenne doivent être lus par opération sur *toute* structure de données pour le problème des classes disjointes.

Applications

Cette structure est souvent utilisée pour identifier les composantes connexes d'un graphe (voir l'article sur les algorithmes de connexité basés sur des pointeurs). Dans le même esprit, elle est utilisée pour étudier la percolation, avec l'algorithme de Hoshen-Kopelman.

Union-Find est également utilisée dans l'algorithme de Kruskal, pour trouver un arbre couvrant de poids minimal d'un graphe. Elle est enfin utilisée dans les algorithmes d'unification[4,5].

Voir aussi

Articles connexes

- L'opération Union en théorie des ensembles
- find, une commande UNIX

Lien externe

- Implémentation de union-find, graphique et en ligne (http://www.irisa.fr/prive/fschwarz/mit1_algo2_2013/union_find/)

Notes et références

- **(en)** Cet article est partiellement ou en totalité issu de l'article de Wikipédia en anglais intitulé « Disjoint-set_data_structure (https://en.wikipedia.org/wiki/Disjoint-set_data_structure?oldid=104581158) » (voir la liste des auteurs (https://en.wikipedia.org/wiki/Disjoint-set_data_structure?action=history)).

1. Bernard A. Galler et Michael J. Fisher, « An improved equivalence algorithm, *[[Communications of the ACM]]* » (http://portal.acm.org/citation.cfm?doid=364099.364331), ACM Digital Library, mai 1964 (consulté le 27 octobre 2007), p. Volume 7, Issue 5, pages 301-303
2. On pourrait l'appeler « profondeur », mais en présence de compression de chemin, ce concept perd son sens.
3. **(en)** Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein, *Introduction à l'algorithmique*, MIT Press et McGraw-Hill, 2001, 2e éd. [détail de l'édition]. Chapter 21: Data structures for Disjoint Sets, pp.498–524.
4. Sylvain Conchon, Jean-Christophe Filliâtre: A persistent union-find data structure. ML 2007: 37-46 (https://www.lri.fr/~filliatr/ftp/publis/puf-wml07.pdf)
5. Kevin Knight, « Unification: A Multidisciplinary Survey », *ACM Comput. Surv.*, vol. 21, 1er mars 1989, p. 93–124 (ISSN 0360-0300 (http://worldcat.org/issn/0360-0300&lang=fr), DOI 10.1145/62029.62030 (http://dx.doi.org/10.1145%2F62029.62030), lire en ligne (http://doi.acm.org/10.1145

/62029.62030))

Sources

- Zvi Galil et Giuseppe F. Italiano, « Data structures and algorithms for disjoint set union problems » (http://portal.acm.org/citation.cfm?id=116878), ACM Digital Library, septembre 1991 (consulté le 27 octobre 2007), p. Volume 23, Issue 3, pages 319-344
- J. A. Spirko and A. P. Hickman, « Molecular-dynamics simulations of collisions of Ne with La@C_{82} » (http://prola.aps.org/abstract/PRA/v57/i5/p3674_1), mai 1998 (consulté le 27 octobre 2007), *Phys. Rev. A* 57, 3674–3682