TP 8 - Structures de données

9 mars 2011

Pour toutes questions, suggestions, remarques ou autres, n'hésitez pas à m'envoyer un mail à edbonnet@hotmail.com en mettant en objet [TP Caml].

1 Introduction

Aujourd'hui, nous allons voir que l'arbre et ses divers procédés d'équilibrage sont souvent à l'origine de structures de données astucieuses et performantes.

2 Algorithme Union-Find

Union-Find est un algorithme qui vise à manipuler des relations d'équivalence (dont on a besoin par exemple pour la minimisation d'automates, cf un précédent TP). "Union" consiste à fusionner deux classes d'équivalence en une seule et "Find" consiste à déterminer la classe d'équivalence d'un élément. En outre, on aura besoin de deux autres opérations : une initialisation où chaque classe d'équivalence est réduit à un singleton et un compte du nombre de classes d'équivalences. La notion de complexité la plus pertinente dans ce contexte est la complexité amortie, c'est-à-dire sur une suite de N opérations que vaut $\frac{T(n,N)}{N}$ où $T(n, N)$ est le temps d'exécution des N opérations sur une instance de taille n. On va commencer par utiliser un tableau où chaque élément se voit attribuer sa classe d'équivalence.

Question 1 *Implémenter cette version naïve d'Union-Find.*

Question 2 *Quelle est la complexité de chaque opération ?*

Nous allons maintenant utiliser la structure de données de forêt et on retiendra également le nombre d'arbres. Chaque arbre correspond aux membres d'une même classe d'équivalence, faire l'union de deux classes consiste à rattacher un arbre à la racine de l'autre et trouver la classe d'équivalence d'un élément consiste à partir de cet élément et à remonter vers sa racine. La forêt sera représenté par un tableau d'entiers encodant la relation père. Une racine sera son propre père.

Question 3 *Implémenter cette deuxième version d'Union-Find.*

Question 4 *Quelle est la complexité de chaque opération ? Pourquoi cette version n'est-elle pas meilleure que la première ?*

Pour que l'idée d'utiliser des forêts soit vraiment efficace, il faut s'arranger pour contrôler l'équilibre des arbres. On va donc retenir la hauteur des arbres et greffer le plus petit arbre à la racine du plus grand, dans le procédé d'union de deux classes.

Question 5 *Proposer une deuxième amélioration très pragmatique pour que le temps passé à trouver la classe d'équivalence d'un élément ne soit pas perdu si on la cherche à nouveau par la suite.*

Question 6 *Implémenter ces deux améliorations.*

On peut montrer (cf Introduction to algorithmics de Cormen, Leiserson, Rivest et Stein) que la complexité amortie est maintenant $O(\alpha(n))$ où α est l'inverse de la fonction d'Ackermann définie par $n \mapsto A(n,n)$ avec $A(0,n) = n+1$, $A(m,0) = A(m-1,1)$ et $A(m,n) = A(m-1, A(m,n-1))$. Comme $A(4,4) = 2^{2^{2^{65536}}} - 3$, $\alpha < 4$ pour des entrées raisonnables, et le coût amorti est donc considéré comme constant.

Question 7 *Écrire une fonction qui prend une liste de congruences (sous la forme par exemple de couples) et qui renvoie la plus fine relation d'équivalence correspondante.*

3 Arbres d'Adelson-Velsky et Landis (AVL)

Les arbres AVL comme les arbres rouge-noir sont des arbres binaires de recherche qui possèdent des propriétés supplémentaires d'équilibre. Un arbre AVL vérifie que tous ses nœuds internes ont comme fils gauche et fils droit des arbres de hauteur variant de 1 au maximum.

Question 8 *Montrer qu'un AVL de taille n a une hauteur en $O(log(n))$.*

Question 9 *Écrire des fonctions naïves de recherche, d'insertion et de suppression dans un arbre binaire de recherche quelconque. Quelle est la complexité amortie de ces opérations ?*

Si on arrive à préserver la structure d'AVL en insérant et en supprimant des clés dans l'arbre, alors sa hauteur reste en $O(log(n))$ et donc la recherche d'une clé, ainsi que l'insertion et la suppression, se font en $O(log(n))$ en complexité amortie.

Question 10 *Écrire des fonctions d'insertion et de suppression qui conservent la structure d'AVL au moyen de rotations entre un nœud interne et ses fils.*