TD Info 7 : Union-Find & Tables de hachage

Michael Monerau

27 janvier 2011

1 Introduction

On commence par reprendre le problème Union-Find de la dernière fois, et on l'utilise pour résoudre trois problèmes particuliers. Ensuite on va voir comment fonctionnent les tables de hachage.

2 Applications de Union-Find

Beaucoup d'algorithmes utilisent Union-Find, souvent même sans le dire comme brique de base classique. La linéarité en pratique est évidemment très attirante. Voyons trois tels algorithmes.

2.1 Connexité dans un graphe

Question 2.1. Connexité

Supposons qu'on donne un graphe comme la liste de ses arêtes. Donner un algorithme utilisant Union-Find qui permet de dire si deux sommets sont ou non dans la même composante connexe.

Donner un avantage de cette approche sur l'algorithme déjà vu au TD 2 de parcours DFS ou BFS du graphe.

Implémenter l'algorithme (on utilisera le code pour Union-Find disponible sur http ://www.michael.monerau.com).

2.2 Plus petit ancêtre commun dans un arbre

Étant donné un ensemble de paires de nœuds P, on veut trouver leur ancêtre commun qui leur est le plus proche (le plus bas dans l'arbre). [Voir explications au tableau]

Implémenter l'algorithme.

2.3 Équivalence de deux automates finis

L'algorithme de Hopcroft Karp permet de déterminer si deux automates finis sur le même alphabet reconnaissent ou non le même langage. [Voir explications au tableau]

Implémenter l'algorithme.

3 Tables de hachage

3.1 Principe

Nous avons déjà vu dans des TDs précédents des structures pour stocker efficacement de grands nombres d'objets, comme les arbres binaires de recherche, les arbres rouge-noir et autres.

Ici on veut faire un telle structure de données mais qui a une complexité en temps encore meilleure, comme souvent au détriment de l'espace mémoire nécessaire.

L'idée tient en peu de mots : les objets sont stockés dans un tableau, et on dispose d'une *fonction de hachage* qui transforme un objet en un indice dans le tableau. Ensuite lorsqu'on cherche un objet, on regarde s'il est présent à la case où il est supposé être (grossièrement).

3.2 Listes de collision

On suppose qu'on veut stocker des éléments de type `'a` et qu'on dispose d'une fonction de hachage `f : 'a -> int`. Nécessairement, f n'est pas injective et il faut donc gérer les *collisions* : deux éléments qui veulent aller dans la même case.

On choisit ici de gérer ce problème en stockant non pas les éléments eux-mêmes mais des listes d'éléments dans chaque case du tableau.

Question 3.1. Implémentation

Implémenter la structure de données avec des listes de collision : création (avec la taille en argument),

insertion, recherche dans la table de hachage. Attention, il faut toujours prendre le modulo par la taille de la table pour ne pas dépasser des bornes du vecteur.

Regarder la complexité.

3.3 Linear probing

Une autre stratégie consiste à cette fois-ci bien stocker les éléments dans les cases du tableau, mais la gestion des collisions diffère. Si on veut insérer un élément dans une case déjà occupée, on l'insère dans la première case suivante qui est vide.

Question 3.2. Réalisation

Quel problème se pose pour la recherche ? Proposer une méthode pour contrecarrer ce souci.

Indication trop gentille : on a le droit de changer la taille de la table quand on veut tant qu'on maintient les invariants de la structure.

Question 3.3. Implémentation

Implémenter la structure de données avec des listes de collision : création (avec la taille en argument), insertion, recherche dans la table de hachage.

Regarder la complexité.