Projet 1 (C) : Voyageur de commerce

Alexandre Talon & Grégoire Beaudoire

4 novembre 2012

Table des matières

Introduction

On a ici réalisé un projet de programmation dans le langage C. Ce projet, qui a duré sept semaines, a été en très grande partie réalisée en monôme par Alexandre Talon.

On étudie le problème problème d'optimisation classique du voyageur de commerce : il s'agit de parcourir une liste de ville donnée et de revenir à la ville de départ en un chemin de longueur minimale. D'un point de vue plus formel,

on peut voir les villes comme étant les sommets d'un graphe, la distance entre deux villes le poids de l'arête reliant les sommets correspondant, le problème étant de trouver un cycle hamiltonien de longueur minimale.

Il s'agit en fait d'un problème NP-complet, c'est-à-dire qu'on ne dispose pas d'algorithme le résolvant de manière exacte en un tems polynomial en fonction de la taille de l'entrée (villes et distances entre chaque paire de villes). On s'intéresse alors à un algorithme résolvant le problème du voyageur de commerce de manière approchée : on cherche à programmer un algorithme permettant de trouver une tournée passant par toutes les villes imposées par l'utilisateur, le tout en un temps raisonnable et en fournissant une réponse pas trop éloignée de la réponse optimale.

Pour y parvenir, il est nécessaire de choisir des structures de données adaptées aux données que l'on va stocker et à la façon dont on les utilise. On cherche ici à optimiser le temps mis à l'exécution des divers algorithmes, tout en limitant raisonnablement la place occupée par les données en mémoire. On commencera par décrire ces structures et la façon dont elles sont implémentées. Dans un second temps il sera question des algorithmes à proprement parler utilisés pour résoudre le problème. On terminera par le choix de l'interface utilisateur.

1 Structures de données

On utilise dans ce projet deux structures non triviales : d'une part des tas, d'autre part des arbres arbres binaires de recherche équilibrés de type arbres AVL.

1.1 Tas

Un tas est un arbre binaire complet vérifiant la propriété suivante : l'étiquette de tout noeud différent de la racine est supérieure à l'étiquette de son père. De plus, l'arbre est rempli « dans l'ordre » : si h est la profondeur de l'arbre, alors tous les étages de profondeur plus petite que h sont remplis.

D'un point de vue pratique, on peut stocker cet arbre binaire dans un tableau t. Pour des raisons pratiques, on place la racine en $t[1]$. Alors si un noeud est en position i, donc dans $t[i]$ alors son fils gauche est dans $t[2*i]$ et son fils droit dans $t[2*i+1]$. De plus le père de $t[i]$ est en $t[E(i/2)]$, où $E(x)$ désigne la partie entière de x. On peut ainsi calculer très facilement les positions des fils et du père d'un noeud.

Aussi, dans ce projet on a besoin de tas dont on connaît la taille maximale, on ne change donc pas la taille du tableau utilisé pour stocker le tas au fur et à mesure de son utilisation.

Cette structure supporte plusieurs opérations, décrites ci-dessous.

1.1.1 Accès au plus petit élément

Il est trivial d'accéder au plus petit élément d'un ensemble stocké dans un tas : il s'agit de l'étiquette de la racine. Ceci découle immédiatement de la

propriété d'ordonnancement des noeuds.

1.1.2 Insertion

Lorsqu'on ajoute un élément, on ne peut se contenter de l'ajouter à la première case libre car alors la propriété sur les étiquettes des arbres ne serait plus respectée. Pour insérer un élément, on procède donc comme suit. On commence par ajouter l'élément à la première place libre. Puis tant que l'étiquette de ce noeud est plus petite que celle de son père on échange ces deux noeuds de place, et on continue avec le même noeud. Dans notre cas, il faut vérifier que l'on s'arrête une fois que le noeud est devenu la racine. On gère ce problème en posant $t[0] = -1$. Comme toutes les étiquettes sont positives, ceci agit comme une barrière.

1.1.3 Suppression de la racine

Dans un tas on ne s'autorise la suppression que de la racine, donc du minimum du tas. Il faut alors trouver une nouvelle racine. On commence par choisir le noeud le dernier noeud de l'arbre comme racine. Ceci permet de conserver la propriété de complétude du tas. Ensuite, on fait descendre ce noeud dans l'arbre : tant que ce noeud est plus grand que le plus petit de ses fils, on échange ces deux noeuds et on continue avec le même noeud. La complétude de l'arbre est donc bien conservée.

1.1.4 Performances

Notons tout d'abord h la hauteur de l'arbre binaire, et n le nombre de noeuds qu'il contient. Comme cet arbre est binaire et complet, on a $h = E(log(n))$. La lecture du plus petit élément du tas se fait en $O(1)$. L'insertion et la délétion, correspondant à un parcours dans l'arbre, sont donc en $O(h) = O(log(n))$. Enfin, un tas prend $O(n)$ en mémoire.

Ces propriétés font du tas une structure utile pour maintenir le minimum d'un ensemble auquel on ajoute des éléments et retire le minimum. Il est utile, comme nous le verrons ci-dessous, pour l'algorithme de Prim.

1.2 Arbres AVL

Les arbres AVL appartiennent à la classe des arbres binaires de recherche. Ils présentent en outre la particularité d'être automatiquement équilibrés, c'est-à-dire :

Définition (Arbre équilibré)**.** *Un arbre équilibré est un arbre dans lequel la hauteur des deux sous-arbres de tout noeud diffère d'au plus un.*

Pour pouvoir définir plus aisément la notion de "différence de hauteur de sous-arbre", on va définir le facteur d'équilibrage :

Définition (Facteur d'équilibrage d'un noeud)**.** *Le facteur d'équilibrage d'un noeud est la différence entre la hauteur de son sous-arbre droit et celle de son sous-arbre gauche.*

Ainsi un arbre équilibré est simplement un arbre dont le facteur d'équilibrage de tout noeud est compris entre -1 et 1.

Les arbres AVL s'utilisent comme les arbres binaires de recherche, mais on peut également effectuer sur eux les opérations de rééquilibrage. Pour rééquilibrer un arbre déséquilibré, on effectue ce qu'on appelle des rotations. Il convient tout d'abord de remarquer qu'un neoud peut être déséquilibré de deux façons : il peut être "left heavy", c'est-à-dire qu'il possède un facteur d'équilibrage inférieur ou égal à -2, ou "right heavy", c'est-à-dire qu'il possède un facteur d'équilibrage supérieur ou égal à 2. Nous allons commencer par voir comment résoudre ces problèmes par des rotations simples, puis remarquer qu'elles ne suffisent pas dans certains cas où il est obligatoire d'utiliser des rotations doubles.

1.2.1 Rotations simples

1.2.2 Rotations doubles

1.2.3 Algorithme d'équilibrage

Lors d'insertions ou délétions, il peut arriver que l'on obtienne un arbre déséquilibré. Dès qu'on rencontre un noeud ayant un "<mauvais"> facteur d'équilibrage, on applique un rééquilibrage, dont le pseudo-code est ci-dessous.

Si L'arbre est right heavy **Alors**
  Si Le sous-arbre droit de la racine est left heavy **Alors**
    Effectuer une double rotation gauche.
  Sinon
    Effectuer une simple rotation gauche.
  Fin Si
Sinon Si L'arbre est left heavy **Alors**
  Si Le sous-arbre gauche de la racine est right heavy **Alors**
    Effectuer une double rotation droite.
  Sinon
    Effectuer une simple rotation droite.
  Fin Si
Sinon
    Ne rien faire : l'arbre est équilibré.
Fin Si