Projet 1 (C) : Voyageur de commerce

Alexandre Talon & Grégoire Beaudoire

9 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 grande partie réalisée en monôme par Alexandre Talon, avec une participation tardive de Grégoire Beaudoire.

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

Commençons par définir les arbres binaires de recherche :

Définition (ABR)**.** *Un arbre binaire de recherche, ou ABR, est un arbre où chaque noeud possède une étiquette telle que, en notant (gc, x, dr) un noeud, où g est le sous-arbre gauche, x l'étiquette et dr le sous-arbre droit : pour tout noeud $N = (gc', y, dr') \in gc$, $y < x$, et pour tout noeud $N' = (gc'', z, dr'') \in dr$, $z > x$.*

Les arbres AVL sont 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. Nous allons commencer par voir les opérations basiques qu'on peut effectuer sur ces arbres, puis nous verrons comment rééquilibrer un arbre.

1.2.1 Opérations basiques

Nous allons ici voir les opérations de base, *ie* l'insertion et la suppression.

D'abord, l'insertion : pour insérer un élément x dans un arbre binaire de recherche, équilibré ou non, on va parcourir l'arbre pour trouver sa place. Ainsi l'algorithme peut s'écrire :

Si L'arbre est vide **Alors**
    Créer l'arbre réduit à l'élément que l'on veut insérer
Sinon Si $x >$ étiquette de la racine **Alors**
    Insérer x dans le sous-arbre droit
Sinon
    Insérer x dans le sous-arbre gauche
Fin Si

On peut évidemment obtenir des arbres déséquilibrés après une insertion, on expliquera plus tard sur la manière d'équilibrer un arbre. Il est en revanche clair qu'on a toujours un arbre binaire de recherche après l'insertion.

La suppression maintenant : on a plusieurs cas, suivant si le noeud à supprimer possède des fils ou pas :

Si Le noeud n'a pas de fils **Alors**
    Supprimer le noeud
Sinon Si Le noeud a un fils **Alors**
    Supprimer le noeud et le remplacer par son fils
Sinon
    Supprimer le noeud, le remplacer par son successeur, *ie* l'élément le plus à gauche dans son sous-arbre droit.
Fin Si

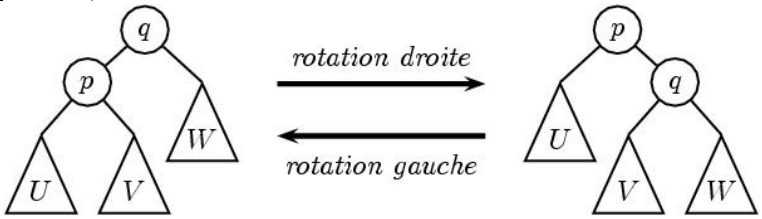
Si le noeud n'a pas de fils ou un seul fils, on garde trivialement la structure d'arbre. S'il en a deux, comme on le remplace par le plus petit élement plus grand que lui, on garde la structure d'arbre : les éléments du sous-arbre gauche sont toujours plus petits que cette nouvelle racine, et tous les éléments du sous-arbre droit étaient plus grands que leur plus petit élément, ils sont donc plus grand que cette nouvelle racine.

Ces opérations basiques sont valables pour les arbres binaires de recherche en général cependant. 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 noeud 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.

1.2.2 Rotations simples

Considérons l'exemple intuitif de déséquilibrage, qu'on pourrait représenter sous la forme (vide, x, (vide, y, (vide, z, vide))) : un noeud x possédant un fils droit y et pas de fils gauche, et y possédant un fils droit z et pas de fils gauche. Cet arbre est déséquilibré : en effet son facteur d'équilibrage vaut 2, la hauteur de son sous-arbre droit étant 2 et celle de son sous-arbre gauche 0. Pour l'équilibrer, on va effectuer une rotation :

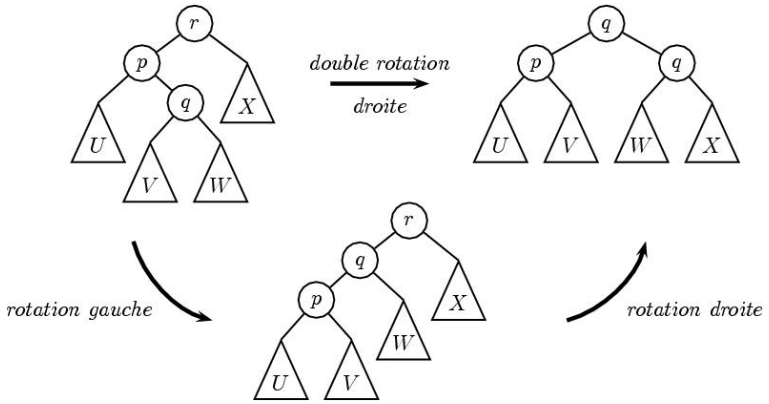


Ainsi on peut résoudre notre problème d'arbre déséquilibré en faisant une simple rotation gauche. Cependant, les simples rotations ne permettent pas de conclure dans tous les cas. Des problèmes vont apparaître lorsque un arbre et son sous-arbre dominant ne «penchent » pas du même côté : quand on a un arbre right heavy, avec un sous-arbre droit right heavy, ou un arbre left heavy, avec un sous-arbre gauche left-heavy, on peut appliquer une simple rotation. Quand on a un arbre right heavy avec un sous-arbre droit left heavy, et inversement, les rotations simples ne donnent rien.

Ainsi les rotations simples ne suffisent pas pour rééquilibrer tous les abres : il est nécessaire d'utiliser ce qu'on appelle des rotations doubles pour pouvoir avoir un algorithme général de rééquilibrage.

1.2.3 Rotations doubles

L'idée des rotations doubles est de d'abord travailler sur le sous-arbre, pour se ramener à un cas simple «droit» qu'on peut résoudre avec une rotation simple. Les rotations doubles fonctionnent comme suit [1] :



Ainsi, avec les simples rotations et les doubles rotations, on peut maintenant équilibrer tous les arbres déséquilibrés et ainsi garantir cette propriété d'équilibrage.

1.2.4 Algorithme d'équilibrage

Lors d'insertions ou délétions, il peut arriver que l'on obtienne un arbre déséquilibré. Dès qu'on rencontre un nœud 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

2 Algorithme de Prim

Le but de l'algorithme de Prim est de renvoyer un arbre couvrant minimal d'un graphe connexe donné en entrée.

D'abord, il convient de se demander ce que fait réellement cet algorithme. Pour comprendre ce qu'il se passe, nous allons avoir besoin de quelques définitions :

Définition. *(Arbre couvrant minimal) Un arbre couvrant est un ensemble d'arêtes tel qu'entre toute paire de sommets du graphe il existe un chemin les reliant et n'empreintant que des arêtes de cet ensemble. On dit de plus qu'un arbre couvrant est minimal si la somme des poids de ses arêtes est inférieure à la somme des poids des arêtes de tout autre arbre couvrant.*

2.1 Algorithme

L'alogrithme prend en entrée un graphe $G = (V, E)$ et une fonction de poids w. L'algorithme consiste en :

▷ Choisir $r \in V$ comme racine.

▷ On initialise notre arbre couvrant, qui ne contient que r pour l'instant : $V(T) = \{r\}; E(T) = \{\}$;

▷ $E' = \{e = (v_i, v_j) \in E \| v_i \in V(T), v_j \in V(G)/V(T)\}$

▷ Trier E' par ordre croissant de poids selon w.

Tant que $V(T) \neq V(G)$ **Faire**

Sélectionner l'arête $e = (v_1, v_2) \in E'$ de poids minimal, avec $v_1 \in V(T)$ et $v_2 \in V(G) \neq V(T)$

Ajouter e à $E(T)$ et v_2 à $V(T)$

$E' = \{e = (v_i, v_j) | v_i \in V(T)$ et $v_j \in V(G) \neq V(T)\}$

Fin Tant que

Retourner $T = (V(T), E(T))$

Ainsi, tant que l'arbre ne recouvre pas le graphe, on prend une arête dont une extrémité est dans l'arbre et l'autre non, on l'ajoute à l'arbre, ainsi que le sommet qui n'était pas dans l'arbre, et on recommence.

2.2 Implémentation

On utilise une structure de tas pour stocker les arêtes choisies qui formeront l'arbre couvrant renvoyé. On renvoie cet arbre sous la forme de liste d'adjacence : un tableau adj tel que $adj[i]$ contienne la liste des voisins de i dans l'arbre.

A chaque étape, on prend l'arête de poids minimum, et si elle ne connecte pas deux sommets déjà présents dans l'arbre couvrant, on ajoute cette arête à l'arbre couvrant, ainsi que le sommet non encore dans l'arbre couvrant. On mémorise les pères des sommets ajoutés dans un tableau des pères, pour savoir quelles villes sont connectées dans l'arbre. On utilise également un tableau mémorisant le nombre de voisins de chaque sommet dans l'arbre.

Enfin, on construit la liste d'adjacence à partir du tableau des pères et de celui du nombre de voisins.

3 Algorithme TSP

On utilise au cœur de ce projet l'algorithme TSP qui, comme précisé dans l'introduction, donne une réponse approchée au problème du voyageur de commerce.

3.1 Algorithme

L'algorithme TSP est en fait assez simple. Il s'agit tout d'abord d'obtenir un arbre couvrant minimal du graphe considéré.

On construit, dans ce projet, cet arbre couvrant à l'aide de l'algorithme de Prim décrit ci-dessus.

Une fois que l'on a un arbre couvrant minimal, l'algorithme TSP consiste à partir d'un sommet et parcourir l'arbre en conservant les sommets rencontrés dans l'ordre où on les a rencontrés dans l'arbre. On ne garde alors que la première occurence de chaque sommet et on renote le premier sommet à la fin (on doit revenir au point de départ). La réponse de l'algorithme est alors cette liste de villes.

3.2 Implémentation

L'algorithme de Prim que l'on a codé nous renvoie un arbre couvrant minimal sous la forme d'une liste d'adjacence : il s'agit d'un tableau adj tel que $adj[i]$ contienne la liste des voisins de i dans l'arbre. On effectue alors un parcours infixe des sommets de cet arbre. Afin de ne pas voir des sommets plusieurs fois, on déclare un tableau *seen* dont la i-ème case vaut *true* si et seulement si on a déjà visité le sommet i au cours du parcours. On note donc dans un tableau le sommet en cours de visite, on le marque comme déjà vu, puis on visite récursivement ses fils non encore visités. On leur indique à partir de quelle case du tableau écrire les villes que cette visite parcourera. Puis quand on passe au voisin suivant, on a récupéré l'adresse de la case du tableau à laquelle l'écrire, et ainsi de suite.

3.3 Interface utilisateur

Une fois l'algorithme compilé et l'éxécutable lancé, l'utilisateur est invité à choisir ce qu'il veut faire. Il a plusieurs options : il peut choisir d'ajouter une ville, d'en supprimer une, d'afficher les villes déjà choisies ou bien de lancer l'algorithme TSP.

S'il choisit d'ajouter une ville, il est alors invité à rentrer les premières lettres de la ville qu'il veut ajouter. S'il y a moins de 50 villes qui commencent par ces lettres, elles sont alors affichées à l'écran et numérotées. L'utilisateur peut ensuite taper les numéros des villes qu'il souhaite ajouter.

Les villes sont en fait stockées dans un tableau, préalablement trié. Elles sont ensuite ajoutées dans un arbre AVL au fur et à mesure que l'utilisateur le demande (ceci permet un ajout et une suppresion rapide).

Il n'y a pas d'interface graphique, il s'agit simplement d'une interface textuelle.

Conclusion

Nous avons réalisé un voyageur de commerce en C, en utilisant les algorithmes de TSP et de Prim. L'algorithme marche mais pourrait sans doute être

amélioré, notamment sur le plan du stockage et de l'accès aux villes et aux distances. L'idée des arbres AVL permet une certaine efficacité adns l'ajout et suppression de villes par l'utilisateur cependant. Enfin l'interface utilisateur n'ext pas du tout graphique, et seulement textuelle, bien que claire et facile d'accès.

Références

[1] Luc Maranget. Arbres binaires. `http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/421/poly/arbre-bin.html`.