Mission 6 : Rapport Final

Groupe 2.2

Vendredi 12 Décembre 2014

Introduction

Pour cette mission, il nous a été demandé de pouvoir établir la liste de connections aériennes à inclure dans l'offre d'une compagnie low cost. L'objectif était donc de pouvoir desservir un nombre maximum de destinations à moindre coût à partir d'une liste de coûts d'ores et déjà définis entre deux villes.

Approche générale du problème

Pour minimiser la somme des coûts de chacune des connections effectives du réseau, nous avons décidé de procéder de la manière suivante :

1. Choisir une ville de départ.
2. Regarder les coûts des différents vols depuis la ville de départ vers les destinations possibles. Stocker les vols et leur coût dans une priority Queue ayant pour priorité le coût minimale.
3. Prendre le vol se trouvant en tête de la priority Queue, le vol le moins coûteux.
4. Regarder les différents vols possibles depuis cette nouvelle destination. Stocker les nouveaux vols et leur coût dans la priority Queue.
5. Prendre le vol se trouvant en tête de la priority Queue, le vol le moins coûteux. Regarder si la ville de destination est déjà relié au réseau si c'est le cas on supprime le vol de la priority Queue. Si non, on reprend à l'étape 4.
6. Tant que toutes les villes n'ont pas été visitées, on applique les étapes 4 et 5.

Il est important de donner certaines précisions sur le graphe dans lequel nous travaillons. Le graphe est non dirigé, c'est-à-dire que le coût du vol aller est le même que celui du vol retour. Nous pouvons affirmer que cet algorithme nous fournit le minimum de la somme des coûts de chacune des connections effectives dans le réseau car nous élargissons le réseau de connections au fur et à mesure avec le vol de plus faible coût reliant le réseau déjà existant à une nouvelle destination. La solution n'est unique si les coûts d'un vol à partir d'une même ville d'origine sont identiques,car nous travaillons avec une priority queue et donc nous ne savons pas dans quel ordre elle va classer les vols de même coût.

Description de l'implémentation

Nous avons divisé le programme en deux classes : Graphe et SpanningTree. La première classe va nous servir à créer le Graphe à partir d'une liste de coûts entre deux villes présentée dans le format suivant :

0	37	398
1	39	929
	...	

Nous avons décidé de définir un constructeur pour la classe Graphe. Celui-ci va prendre en argument un String filename représentant un fichier qui contient une liste comme expliquée ci-dessus.

Après avoir calculé la taille de la liste, le programme va se poursuivre en créant une *ArrayList* de taille n contenant à chaque indice une liste d'arêtes *Edge*. Il est important de préciser que *Edge* est une sous classe de Graphe. Celle-ci va nous permettre de représenter une arête composée d'un coût, d'une origine et d'une destination. Cette sous classe est accompagnée d'une méthode toString et d'une méthode compareTo pour pouvoir comparer deux arêtes sur base de leur coût respectif.

La seconde classe, SpanningTree, ne contient qu'une méthode main qui va réaliser tous les calculs de plus courts chemins ; c'est elle qui va nous fournir la liste de connections aériennes à inclure dans l'offre de la compagnie low cost. Pour ce faire, la méthode va tout d'abord se charger de créer un Graphe par le biais d'un nom de fichier indiqué en argument. Ensuite, elle va créer une *PriorityQueue* qui contiendra toutes les arêtes du graphe classé en fonction de leurs coûts respectifs. De plus nous allons utiliser un tableau de *boolean* pour savoir quel noeud a déjà été visité lors de l'application de l'algorithme.

Dans un premier temps l'agorithme va ajouter à la *PriorityQueue* toutes les arêtes du premier noeud du graphe. Ensuite nous allons récupérer la première arêtes de la *PriorityQueue*, c'est-à-dire celle avec le plus petit coûts, et vérifier si soit l'origine ou la destination de cette arrêtes n'est pas encore été visité, si ce n'est pas le cas on ajoute l'arrêtes à la liste de solution, on note que l'origine ou la destination de cette arêtes a été visité et on rajoute dans la *PriorityQueue* toutes les arrêtes du noeud d'origine ou de destination de cette arêtes. Et ainsi de suite jusqu'à ce que la *PriorityQueue* soit vide.

Complexité de l'implémentation de la recherche du minimum de la somme des coûts de chacune des connections effectives dans le réseau

Comme expliqué précédemment pour représenter un graphe, nous utilisons une arraylist pour stocker une liste d'arête et la position dans l'arraylist représente une ville. Cela nous permet de trouver toutes les arrêtes liées à une ville en O(1). Le tableau de boolean représentant le fait qu'une ville est déjà visitée ou non fonctionne de la même manière.

Adaptation

Notre implémentation ne permet pas de gérer un tel changement de problème. Nous pouvons malgré tout proposer un algorithme permettant de résoudre de manière naïve cette question. Tout en sachant qu'avec un tel algorithme nous n'obtiendrons pas une solution optimale.

1. On prend un nœud au hasard comme racine.
2. On prend un autre nœud au hasard.
3. On calcule un Dijkstra modifié entre ces deux nœuds et on sélectionne le meilleur chemin.
4. On ajoute tous les nœuds sur ce chemin à l'ensemble des nœuds connectés.
5. On prend un autre noeud au hasard, on fait Dijkstra pour calculer sa distance à tous les nœuds.
6. On ajoute les nœuds et arête sur le chemin le plus court vers un nœud déjà relié au graphe des nœuds connectés.

Tant que tous les nœuds ne sont pas reliés, on reprend à l'étape 5.

La solution exacte du problème semble très compliquée et en tous cas ne peut sans doute pas être trouvé avec une *greedymethod*. D'après nous, pour résoudre ce problème il faudrait un algorithme de complexité exponentielle.