M1 ALMA
Université de Nantes
2010-2011

Projet de TP n°2
Structures complexes et algorithmes

MARGUERITE Alain
RINCE Romain

Université de Nantes
2 rue de la Houssinière, BP92208, F-44322 Nantes cedex 03, FRANCE

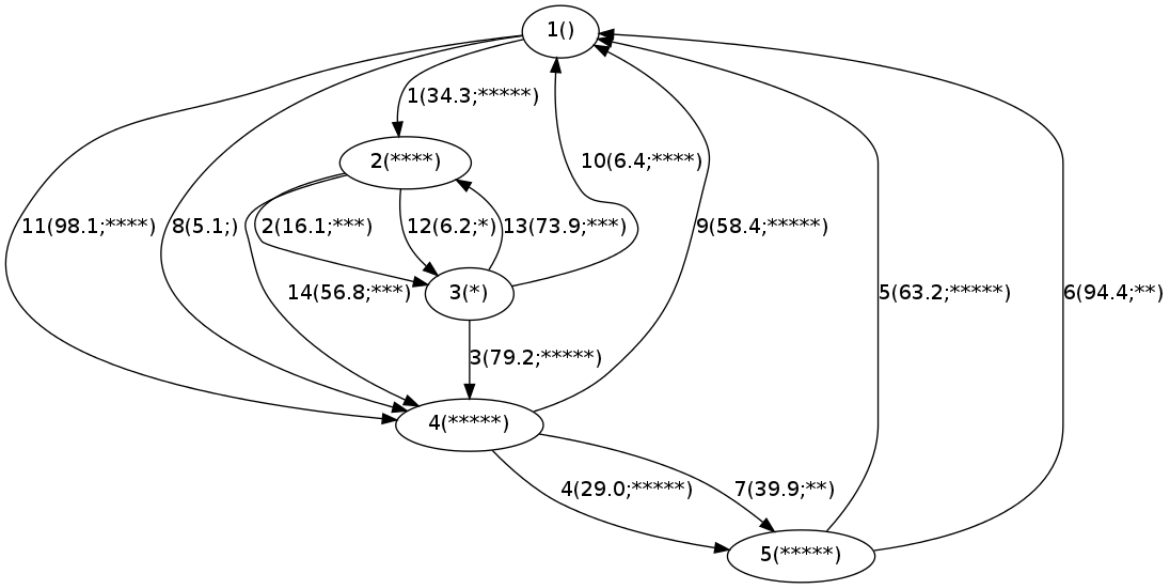
Encadrant : Christophe JERMANN

Table des matières

1 Introduction

1.1 Problème à résoudre

L'objectif de ce projet est de réaliser un Global Positioning System «amélioré». Sans entrer dans les détails, un GPS est réalisé à partir d'une modelisation sous forme de graphe orienté d'une carte où les posistions sont des noeuds et les routes sont des arcs. L'application d'algorithmes de plus courts chemins sur ces graphes permettent alors de répondre au problème concrét du calcul d'iténéraires. Cependant il est rare, dans la vie réelle, que ce type de problème admette une unique contrainte. Ici nous allons travailler sur des graphes plus complexes (voir figure ci-après) par leurs nombres de paramètres



L'idée de réaliser un GPS prenant aussi en compte l'aspect touristique d'un lieu ou d'une route donne les opportunités suivantes :

- Une étude approfondie des algorithmes sur les graphes étudiés en cours pour choisir le plus adapté en fonction du problème et de la structure de donnée concrète.

- Des manipulations plus complexes de ces algorithmes puisqu'ils prennent plusieurs (et non un seul) paramètres en compte.
- Ces différents choix de combinaisons d'implémentations et d'algorithmes entraînes des calculs variés.

Tous ces aspects sont au coeur même du module Structures complexes et algorithmique. La première partie du projet se contentait d'étudier et d'implémenter des structures et des algorithmes complexes. Au terme de l'ultime partie, nous réalisons à partir de ces outils théoriques une application concrète.

1.2 Application réalisée

1.2.1 Structure du programme

Dans la continuité de la première partie du projet, nous avons conçu de nouvelles classes en nous appuyant sur celles de notre graphe orienté. Voici la liste des différents fichiers du package GPS :

- Ville.java : Composée d'un nom et d'une entier qualilté dénombrant le nombre d'étoiles
- Route.java : Héritant de la classe Arc.java, posède un plus une longueur et d'une qualité tout comme Ville.java
- GPS.java : Comporte les différentes méthodes demandées dans le sujet. Le constructeur prend en compte les données fournies par le parseur ainsi que celle saisie sur l'entrée standards par l'utilisateur
- Parser.java : ROMAIN ROMAIN ROMAIN ROMAIN ROMAIN ROMAIN ROMAIN ROMAIN ROMAIN ROMAIN ROMAIN ROMAIN

1.2.2 Déroulement de l'application

Le programme se lance sans paramètres. Il demande à l'utilisateur d'entrer successivement :

- Le nom du fichier d'entrée (devant être présent dans le dossier courant).
- Le choix d'implémentation (l la liste d'adjacence et m pour la matrice d'adjacence).
- Le nom de la ville de départ.
- Le nom de la ville d'arrivée.
- La valeur (K ou A) du paramètre en fonction de la méthode choisie.

L'itinéraire calculé sera alors affiché selon les consignes du sujet. Le programme se termine par la suite. Il faut relancer le programme un autre calcul ?

2 Algorithmes mis en oeuvre

2.1 Algorithme de plus court chemin

Les deux methodes demandées pour le calul d'itinéaire (par agrégation et à détour borné), il est necessaire d'employer un algorithme de plus court chemin. Notre choix de d'algorithme s'est orienté vers celui Bellman-Ford. Son avantage par rapport à celui de Dijkstra est qu'il est capable de détecter un circuit absorbant. Il peut de la sorte informer l'utilisateur l'echec de a recherche d'un itinéraire compromis selon ces paramètres.

2.2 Methode Agrégation

L'objectif de cette methode est d'obtenir un plus court chemin avec une pondération particulière pour routes calculée au préalable. Il suffit donc d'appliquer un algorithme standard de parcours de graphe en fournissant toutes les pondérations de chaques routes et villes que l'on aura calculé au préalable. N Nous avons procédé de cette manière. Ainsi la fonction $get_agregat(Routeroute, doubleA)$ permet de donner l'agrégat d'une route. Elle est utilisée par $publicArrayList < Double > agregation(doubleA)$ qui retourne un tableau de ces pondération indicé par l'id des routes. Ce tableau est ensuite fourni en paramètre à l'Algorithme de Bellman Ford. Celui ci pourra alors procéder au relâchement des arcs en fonctions de ces pondérations. Dans le cas ou une route améliorante est trouvé, celle ci est stockée dans une hash map à la clef de valeur id de la ville destination de cette route. A la fin de l'algorithme de Bellman-Ford nous avons notre plus court chemin par rapport aux agrégations pré calculées dans cette hash map. Les opérations qui suivent servent uniquement à stocker ce chemin dans le bon sens dans une ArrayList. L'opération d'affichage a en effet du resultat sous cette forme pour le traiter.

2.3 Detour Borné

ROMAIN ROMAIN

3 Analyse théorique

3.1 Etudes de cas

3.1.1 Cas de la méthode d'agrégation

Cette methode repose principalement sur l'algorithme de Bellman-Ford de compléxité : O(|N|*|A|) où |N| est le nombre de ville et |A| le nombre de routes. Il faut aussi fournir le tableau des agrégations. Cette opération consite à opérer pour chaque route la formule et l'ajouter dans un tableau. Sa compléxité est donc en O(|A|). La methode se termine par quelques manipulations pour fournir le plus court chemin dans une ArrayList (cf 2.2). Les deux parcours de ces opérations sont en O(|R|) taille maximun du plus court chemin.
Nous avons donc au total une compléxité de O(|A|*|N|).

Le choix de l'implémentation du graph est choisie selon la performance de *listeArcs*. Sa compléxité est de O(|N|+|A|) pour la liste d'adjacence et en $(O|N|^2)$ pour la matrice d'adjacence. Le choix de liste d'adjacence est donc à privilégier.

3.1.2 Cas de la méthode par detour borné

ROMAIN ROMAIN

3.2 Conclusion

methode la plus efficacemethode la plus efficacemethode la plus efficacemethode la plus efficacemethode la plus efficacemethode la plus efficacemethode la plus efficace

4 Analyse experimentale

4.1 BLA

BLABLABLABLA

4.2 Conclusion

BLABLABLABLA