

Rapport Projet Algo

Samy Boumali , 21801950 , projet fait seul

26 Mars 2020

1 Introduction

Le projet a été fait en langage de programmation Java pour des raisons de gestion de mémoire évidentes .

Le projet se divise en deux répertoires : le répertoire src qui contient le code des classes ainsi que le fichier "metro.txt" modifié afin que le programme puisse y lire les informations dont il a besoin , et le répertoire target qui contient les fichiers ".class" obtenus à la compilation de la classe principale : Itineraire.java. (plus le metro.txt pour pouvoir exécuter le programme de n'importe quel directory).

2 Les Classes

Les classes me semble assez évidentes , nous avons :

La classe Ligne : avec un nom , une liste de station , le nom d'un terminus , et une liste de deux terminus au maximum correspondant aux deux autres directions possibles . Cette classe prend en paramètre un nom (String) lors de son instanciation , de plus elle est dotée de la méthode setTerminus qui prend trois chaînes de caractères correspondant aux terminus possibles de la ligne , et bien-sûr si le troisième paramètre est "null" il ne sera pas ajouté et il n'y aura pas d'erreurs (et aussi la méthode toString qui n'est pas utilisée dans notre programme).

La classe Station : avec un nom , une ligne et un numéro de sommet . Elle est donc créée avec trois variables (String , Ligne , int) et représente les sommets du graphe . Elle est aussi dotée de deux méthodes pour comparer deux chaînes de caractères : l'une qui se contente de comparer les tailles , puis si elles sont égales caractère par caractère , et l'autre qui compare deux String en ignorant les espaces (utile pour la comparaison des terminus ajoutés dans les fichiers).

La classe Lien : qui est constituée de deux stations (il les relie) ainsi qu'une valeur int qui symbolise le temps en secondes qu'il faut pour arriver de la première station à la deuxième . Mis à part toString elle n'a pas de méthodes.

La classe Graphe , l'une des plus riches en méthodes : elle a un constructeur par défaut qui crée une liste de Station et de Lien . Elle possède donc na-

turellement une méthode `ajoutSommet` qui prend une variable `Station` qui crée un noeud (S'il est déjà présent il n'est évidemment pas ajouté) , la méthode `createLink` qui quant à elle , ne prend que deux numéros de sommets (qui cherche les sommets correspondants aux numéros entrées dans la liste de sommets) et un int "temps" qui correspond au poids de l'arc , le lien crée est donc ajouté à la liste de liens.

Vient maintenant la méthode principale , une forme de l'algorithme de Djisktra . J'ai fait le choix d'en faire deux versions et voici pourquoi: la première prend deux stations qui peuvent être récupérées de la liste de sommets avec la méthode `list.get(int indice)` (l'indice étant le numéro de la station -1 , puisque une liste débute à l'indice 0) et une variable int appelée "Affichage" que je détaillerai plus tard , et pour la deuxième version elle prend en paramètre deux chaînes de caractères ainsi que cette même variable affichage . La première version s'appelle : " `ItineraireFromProgram(Station depart, Station arrivee, int Affichage)` " , comme son nom l'indique elle est utilisable une fois qu'on est dans la classe main : `Itineraire` . Et la deuxième quant à elle est directement utilisée : à l'exécution du programme (`java Itineraire`) il y a trois "Scanner(System.in)" qui vous demanderont votre station de départ , votre station d'arrivée , ainsi que si vous voulez afficher ou non les stations intermédiaires du trajet obtenu , cette dernière affectera une valeur de 1 à la variable Affichage si vous répondez 'y' et 0 si vous répondez autre chose (pas forcément 'n'). Son nom est " `ItineraireFromExec(String stat1, String stat2,int Affichage)` "

Les deux String correspondantes aux deux premières demandes seront enregistrées et il y aura une recherche dans la liste de sommets selon ce critère : c'est la méthode : `Station getNumStation(String Name)` . Et la dernière méthode à présenter est "String `getDirection(Station s1, Ligne line)` " : elle prend une station et une ligne et renvoie une chaîne de caractère correspondant au terminus dans lequel vous vous dirigez , c'est la direction .

3 La classe principale : Itineraire

Elle est différente des autres de par le fait que c'est elle qui lit le fichier de configuration "metro.txt" , et par conséquent il y a des erreurs à gérer . Elle débute par une tentative d'ouverture du fichier (différentes recherches sur internet m'ont amené à choisir la méthode `getResourceAsStream` qui arrive à localiser le fichier de configuration directement dans le jar) .

Note : si le fichier n'existe pas le programme s'arrête avec un `nullPointerException` . Pour cette raison je l'inclus dans le jar.

On demande à l'utilisateur les trois informations citées précédemment (départ , arrivée , affichage) et à partir de là on parcourt le fichier en construisant et en complétant le graphe en même temps , et cette lecture est en grande partie gérée par l'analyse du premier caractère et de la division de la ligne à laquelle on est selon ce dernier (voir en détail le code commenté) . Après avoir construit le graphe , il suffit de lancer la recherche de l'itinéraire .

4 Conclusion

Le programme est entièrement commenté , et il sera soumis sous le format d'une archive .tar.gz qui contient :

- Une archive .jar qui contient le code compilé ainsi que le fichier configuration (obtenu en faisant : " cd target/classes jar cvfe Itineraire.jar uvsq.algo.Itineraire . ")

- Ce rapport

- Le code source (dans src)

Il est donc inutile de compiler le projet , et à fin de l'exécuter il suffira d'écrire en ligne de commande (après avoir extrait le contenu de l'archive tar.gz) :

java -jar Itineraire.jar .