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 Non graphiques

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 évidement 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: ItineraireFromProgram , une forme de l'algorithme de Djisktra . Elle prend en paramètre deux objets de type Station et le principe est simple , on calcule les distances minimales pour chaque station à partir d'une station et on renvoie la distance et le trajet de la station de départ à la station d'arrivée . Et dans un soucis d'optimisation il y a aussi une autre méthode (getItineraireGraphe) qui fait appel à cet algortihme de façon à traiter toutes les lignes de départ possibles (donc de ne pas imposer une ligne de départ) , et de même si la station d'arrivée a plusieurs correspondances . Ainsi on obtient un trajet de durée optimale , et à partir de là on convertit l'ensemble des stations sous forme de chaîne de String (itineraireToString) de façon à pouvoir l'afficher .

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) qui renvoie un ensemble de stations qui ont le même nom mais qui sont sur des lignes différentes . 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 ($Nongraphique$) : Itinéraire

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 deux informations citées précédemment (départ , arrivée) 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 avec "buildGraphe", il suffit de lancer la recherche de l'itinéraire et de l'afficher

4 Classes graphiques

Pour montrer ce à quoi ressemble l'interface je joindrai des captures d'écrans au cas ou vous ne voulez ou ne notez pas utiliser celle ci , et voici comment je l'ai programmée ,en classes graphiques nous avons: MetroGraphe : classe dont le constructeur appelle la fonction "buidGraphe" de la classe Itinéraire afin de construire le graphe , et fait aussi appel à une méthode nommée "addButtons" qui prends en paramètre deux lignes de métro , qui affiche ses stations et définit le comportement de chaque bouton (programmé dans mouseClicked , quand on appuie sur un bouton on récupère la station dont il porte le nom et on le stocke dans une chaîne de caractères . Une fois que la station de départ ait été définie ainsi on fait de même pour la station d'arrivée) puis on fait appel à la fonction de recherche d'itinéraire sur les deux chaînes de caractères qu'on a obtenu . (Les deux lignes dont on veut afficher les stations sont paramétrées dans le constructeur de cette classe)

MapInterface : classe principale de la partie graphique , elle crée une fenêtre , créer un objet de classe MetroGraphe (ainsi on ajoute tous les boutons) .

5 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 de 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 .