UE Programmation objet C++
Compte-rendu projet

TOUZARI Lisa
Master Automatique,Robotique, Parcours: ISI

Introduction

Lors de ce projet, on a été confronté au problème du plus court chemin. Le but était de déterminer le meilleur trajet à prendre (celui qui prend le minimum de temps) entre une station de départ et une station d'arrivée. Pour cela, la méthode proposé pour ce calcul et l'application de l'algorithme de dijkstra.

Différentes méthodes de classe implémentées :

1. **Read-Stations** :

   à partir d'un fichier de données de l'ensemble des stations du métro, on initialise l'attribut stations-hashmap (container de type unordered-map) de notre nouvelle classe.
   -Chaque station est considérée comme une structure qui a plusieurs champs, étant donné que chaque station a une id unique, on considère cette valeur comme une clé de notre container, les reste des données consistera la structure de la station et la valeur du container.
   -On fait attention à l'exception ou on ne peut pas ouvrir le fichier des données, dans ce cas on revoie en sortie un message d'erreur.

2. **Read-Connections** :

   Similaire à la méthode read.stations, à partir d'un fichier de données de l'ensemble des connections entre les stations du métro, on initialise l'attribut connections-hashmap (container de type unordered-map) de notre nouvelle classe.
   -Chaque connexion nous informe sur la station de départ, d'arrivée, et le temps parcouru entre les deux. Etant donné que chaque station a une id unique, on considère alors l'id des deux stations comme des clés de notre graphe et la valeur le temps parcouru.
   -On fait attention à l'exception ou on ne peut pas ouvrir le fichier des données, dans ce cas on revoie en sortie un message d'erreur.

3. **Compute-travel** :

   - Avec cette méthode, on implémente l'algorithme de dijkstra, en sortie on aura alors le meilleur chemin qui sera donné sous forme de vecteur représentant les stations par les quelles on passe, et le temps parcouru depuis le départ jusqu'à une station donnée.
   Le principe : en partant de la station de départ, à chaque itération on déterminera la station la plus proche à une station considérée jusqu'à l'arrivée à la station finale. Pour cela, on a besoin d'un graphe des connections entre les stations, on choisit d'utiliser un container de type unordered map, ce graphe c'est l'attribut de la classe connections-hashmap.
   -On a besoin de connaitre et de mettre à jour la distance de toutes les stations par rapport à la source, on sauvegarde alors ces données-là dans un vecteur (Q, distance) de type pair (id de la station, distance par rapport à la source).
   -A chaque itération, lorsqu'on se trouve dans une station précise, on doit déterminer l'ensemble de ses voisins, ainsi que la station qui la précède. On sauvegarde les voisin dans un container de type unordered-map (id de la station voisine, distance par rapport à la station voisine)), et dans un container de type unordered-map, on sauvegarde chaque station et sa station précédente (id de la station, id de la station qui la précède).
   -Donc, on commence par la station de départ, de là à chaque itération on détermine la station la plus proche à la station de départ tout en mettant à jour les distances ainsi que les prédécesseurs des stations jusqu'à ce qu'on arrive à la station finale. De là, de manière récurrente, on pourra récupérer les prédécesseurs en commençant par la station finale

(sauvegardés dans predec) ainsi que les distances correspondantes (sauvegardés dans distance).
-Pour notre cas, on considère l'exception où la station de départ est la même que celle d'arrivée, et le cas où une station n'existe pas dans la base de données, on renvoie alors un message d'erreur.

4. **Compute-and-display-travel** :

   Cette fonction permet d'afficher le chemin calculé par la fonction précédente, de manière à ce que la personne puisse suivre le chemin facilement.