Polytechnique Montréal

Département de génie informatique et génie logiciel

Cours LOG2810:

Stuctures Discrètes

Travail pratique 1

**Graphes**

Par l'équipe

No 2

Noms:

Alexander Ciaciek 1957252

Nour Abiad 1955147

Dana Louka 1943144

Date:

22 Mars 2020

Introduction:

Ce laboratoire demande la création d’un programme fonctionnelle qui ressemble à l’application Uber. Les clients doivent pouvoir utiliser ce service en ayant des contraintes temporelles. Le conducteur doit utiliser le chemin le plus rapide sans que la batterie de son voiture électrique baisse en-dessous de 15%. Afin d’arriver à ce but, les notions de graphes (chemins, sommets, arrêtes), les structures de données (les listes chainées en particuliers) et l’algorithme Dijkstra ont été utilisés. Premièrement, les demandes du laboratoire ont été analysés et ont permis la conception des différentes classes. Deuxièmement, les méthodes ont été codés afin de pouvoir lire un document texte passé en argument récursivement, de trouver le chemin le plus court dans un graphe et de profiter de et de parcourir les sommets/les arcs d’un graphe. Enfin, une interface qui affiche un menu a été mise en place.

Solution :

Figure 1 : Diagramme de classes.

A screenshot of a computer

Description automatically generated

La classe Main, contient la méthode choisirMenu(). Celle-ci est responsable d’afficher une interface simple. La première option permet de créer, d’afficher et de mettre à jour un graphe. La deuxième se charge de trouver le plus court chemin entre un arrondissement de départ et un arrondissement de destination. La troisième analyse les requêtes des clients. Une boucle « while » et un « switch » permettent de choisir entre ces trois options.

C’est possible de remarquer que la classe « Graphe » est composée de sommets (classe « Sommet »). Chaque sommet contient les informations d’un arrondissement : un identifiant, l’existence d’une station de recharge et une liste d’arcs adjacents. Un arc (classe « Arc ») est composé d’un sommet et d’un poids (ou un temps), ce qui fait que le graphe est pondéré.

Pour créer un graphe (méthode créerGraphe()), le fichier arrondissemnts.txt est lu récursivement jusqu’à que la fin du fichier soit atteinte. Les sommets du graphe sont sauvegardés dans une liste chaînée « listeSommets ». En plus, pour chaque sommet, les arcs ont été créés et sont sauvegardés dans une liste chaînée d’arcs « listeArcs ». En créant les arcs, il fallait associer une relation arc-sommet pour la deuxième colonne du fichier à la première et pour la première à la deuxième.

Pour la méthode afficherGraphe(), il fallait parcourir la liste des sommets, et pour chacun de ces sommets, il fallait parcourir ses arcs. Il faut noter que chaque arc d’un sommet contient le sommet adjacent et la valeur du temps qui les relie. La figure qui suit montre l’affichage du graphe après sa création.

Figure 2 : affichage de l’option a).

A picture containing window, sitting, table, black

Description automatically generated

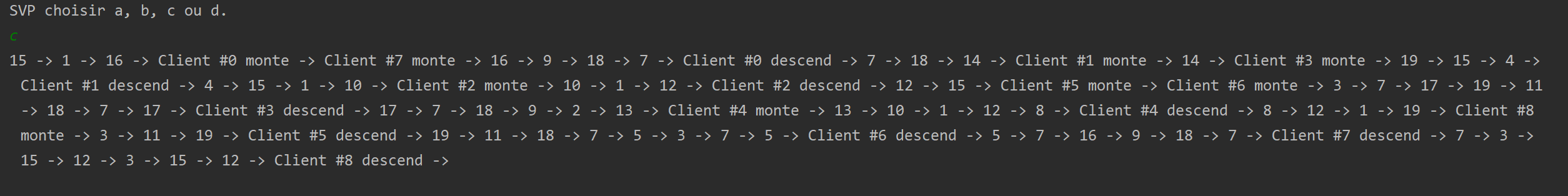
Pour la méthode plusCourtChemin(), elle prend en paramètre le sommet de départ et le sommet de destination. En utilisant Dijkstra, il a été possible de retrouver le chemin le plus court et sa longueur. Afin de pouvoir implémenter l’algorithme, deux attributs ont été ajoutés à la classe sommet : la distance (en minutes) avec le sommet de départ (« poidsAvecDepart ») et le plus court chemin (« plusCourtChemin »). Ce dernier est une liste chaînée de sommets. Il contient les sommets visités afin d’aller du départ au sommet en question. À la fin de l’algorithme, le sommet « Destination » aura la longueur du chemin final et le plus court chemin sauvegardé dans ses attributs. L’algorithme Dijkstra se compose de deux parties. La première est l’initialisation et la deuxième est la boucle « while » qui répète les étapes jusqu’à l’inclusion du sommet de destination dans un sous-graphe. L’initialisation correspond à mettre « poidsAvecDepart » à l’infini pour tous les sommets du graphe sauf le sommet de départ et de créer le sous-graphe à vide. À chaque itération, le sommet avec le poids minimal est cherché et ses sommets voisins sont mises à jour (leurs attributs changent) sous certaines conditions. Voici l’affichage final :

Figure 3 : affichage de l’option b).

A screenshot of a cell phone

Description automatically generated

Pour implémenter la méthode traiterRequetes(), la première étape consistait à trouver le chemin correcte, le plus court et celui qui priorise les requêtes les plus anciens. Cette première version du code permet de ramasser et de déposer les clients aux bons endroits du graphe. À cette étape, l’algorithme ne paye pas attention aux restrictions temporelles des clients, ni au niveau de la batterie de la voiture. Pour implémenter cette première étape, le conducteur priorise le chemin indiqué par la première requête. Il enlève une requête de la liste « listeRequetes » lorsqu’il dépose le client. S’il trouve un client mentionné dans la liste des requêtes qu’il peut le ramasser ou s’il trouve un client dans sa voiture qu’il peut le déposer sur son chemin, il le fera. Les classes « Conducteur » et « Client » ont été créées. Elles contiennent le nombre des personnes dans la voiture (afin de ne pas dépasser 4), la liste des clients qui se trouvent dans la voiture et d’autres informations. Il faut noter que la méthode « lireRequetes » est responsable de lire le fichier requetes.txt et sauvegarder les requêtes dans une liste de requêtes de type « LinkedList<Client> ». L’affichage de cette première étape est le suivant:

Figure 4 : affichage de l’option c) avant d’inclure le système de recharge dans la méthode traiterRequetes. 

Pour gérer les restrictions des temps et la batterie de la voiture, nous ajoutons les attributs « tempsPasse » (temps passé) dans la voiture pour le conducteur et pour chaque client. En plus, l’attribut « pourcentageEnergie » (pourcentage d’énergie) a été ajouté dans la classe « Conducteur ». L’algorithme connait le chemin prochain à emprunter avant de l’emprunter, donc s’il y a une station de recharge sur le chemin. Il s’arrêtera pour 10 minutes (ajouté à « tempsPasse » des clients dans la voiture) afin de recharger la voiture.

Figure 5 : affichage de l’option c) avant d’inclure le système de recharge dans la méthode traiterRequetes.

A screenshot of a computer

Description automatically generated

Difficultés :

La chose la plus difficile à faire dans ce laboratoire est de faire le lien entre les structures discrètes, comme le graphe, et les structures de données dans le code. En effet, le choix d’utiliser des listes comme structure de données au lieu d’un hashMap ou d’autres structures a été difficile à prendre. Cependant, les listes sont faciles à parcourir et peuvent bien sauvegarder toutes les informations d’un sommet. Plusieurs de leurs méthodes sont de complexité O(1) et, en plus, un algorithme de Dijkstra ou d’affichage oblige de parcourir tout le graphe, donc c’est impossible de faire mieux qu’une complexité O(n).

Une deuxième difficulté a été de bien utiliser les librairies du langage Java, la langue de programmation utilisé dans ce laboratoire. Il y avait plusieurs classes et méthodes qui ont facilité le travail. Il fallait découvrir en soi-même ces méthodes pertinentes. Par exemple, la classe Comparator a simplifié énormément la méthode trouverSommetAvecPoidsMinimal() afin de trouver le poids minimal.

Plusieurs problèmes ont été rencontrés lors de l’écriture de la méthode traiterRequetes() et n’ont pas pu être régler. Premièrement, l’affichage du string « -> Recharge -> » se fait avant d’emprunter le chemin suivant et non pas après l’affichage de l’identifiant de l’arrondissement où la voiture a été rechargé. Deuxièmement, si un client est dans la voiture, le conducteur ne peut pas directement aller vers l’arrondissement de destination et est obligé de passer par l’arrondissement de départ. Troisièmement, l’algorithme ne vérifie pas les restrictions temporelles des clients même s’ils sont bien sauvegardés dans la liste des requêtes. Finalement, si sur le chemin il n’existe pas un arrondissement avec une station de recharge, l’algorithme ne s’occupe pas de charger la voiture; Il affiche simplement « pas de station de recharge ». C’est pour cela que le résultat trouvé est négatif.

Conclusion :

Ce laboratoire était utile car ils permettent d’approfondir les connaissances sur les fonctions récursives, la lecture des fichiers et l’écriture d’un programme complet à partir de rien. Ce laboratoire a pris beaucoup de temps, car il fallait se familiariser avec le langage de programmation, l’algorithme de Dijkstra et la conception des classes. Il fallait refaire plusieurs fois les diagrammes de classe et écrire beaucoup de pseudocode. En plus, il fallait fouiller l’internet pour connaitre la bonne syntaxe et les classes intéressantes en Java. Le prochain laboratoire serait probablement aussi demandant et intéressant. Il faut mettre beaucoup de temps afin d’arriver.