POLYTECHNIQUE MONTRÉAL

LE GÉNIE EN PREMIÈRE CLASSE

LOG2810 : Structures discrètes

TP1 : GRAPHES

Session	Automne 2018
Pondération	10 % de la note finale
Taille des équipes	3 étudiants
Date de remise du projet	29 octobre 2018 (23h55 au plus tard)
Directives particulières	Soumission du livrable par moodle uniquement (https ://moodle.polymtl.ca).
	Toute soumission du livrable en retard est pénalisée à raison de 10% par jour de retard.
	Programmation C++, Python ou Java

Équipe	
PRENOM Nom	**Matricule**
DRATWA David	1839262
BEDDOUK Allan	1920352
CANTON-LAMOUSSE Robin	1833489

Table des matières

1. Introduction

Google Maps, Waze, NavigGo, ou encore TomTom, autant d'applications permettant l'aide à la navigation. Toutes ces applications possèdent un bagage technologique considérable. Un bagage compliqué à construire, nécessitant de relever énormément de défis. Ce sont ces défis que nous allons essayer de relever aujourd'hui.

Le programme que nous allons mettre en place à pour but d'optimiser le chemin de prise en charge et de transport de patients d'un centre médical à un autre, tout en tenant compte du type de véhicule utilisé, de sa portée (dictée par la batterie du véhicule électrique), ainsi que de l'urgence du transport. On imaginera une ville, divisée en régions (numérotées) ou sont situés les différents CLSC, avec un réseau routier reliant certaines de ces régions entre-elles et deux types de véhicules médicaux électriques.

Dans le but de relever le défi, nous allons utiliser des graphes dans lesquels les sommets représenteront les centres médicaux et les arcs, quant à eux, représenteront les distances qui les séparent (en minutes). Afin d'optimiser les trajets nous allons appliquer l'algorithme de Dijkstra qui permet de trouver le chemin le plus court entre deux sommets d'un graphe, il s'avère être particulièrement intéressant même s'il peut être compliqué à mettre en place. Le projet présenté ci-dessous tente de relever ce défi et de l'appliquer. Nous avons choisi le langage C++ car c'est celui dans lequel nous étions tous le plus à l'aise.

Dans ce rapport il vous est possible de découvrir le fruit de notre réflexion à propos de ce problème. D'abord nous présentons notre solution et la structure de notre code pour son bon fonctionnement. Ensuite nous mettrons en avant les plus grosses difficultés qui se sont posées durant la conception du projet ainsi que la résolution de celles-ci. Et nous concluront en mettant en avant nos impressions et ce que nous avons appris dans la réalisation de ce travail.

2. Solution et diagramme de classe

2.1 Introduction

Dans cette partie, nous allons vous présenter les principales étapes de notre projet ainsi que notre diagramme de classe final. Nous avons décidé, afin de vous le présenter, de diviser notre travail en quatre étapes que sont la lecture du fichier, l'adaptation de l'algorithme de Dijkstra, l'algorithme de résolution du problème et enfin l'implémentation et la gestion d'erreurs du menu.

2.2 La lecture du fichier

La lecture du fichier a représenté une tâche assez simple car nous avons déjà fait cela dans de nombreux cours. En effet avec un ***getLine()*** on pouvait attraper les chaines de caractères qui nous intéressaient et les transformer en *int* et la fonction ***atoi()*** de *cstdlib*. Nous les avons alors placés dans des vecteurs arcs et sommets.

2.3 L'adaptation de l'algorithme de Dijkstra

Pour l'algorithme de Dijkstra, nous nous sommes appuyés sur e cours et sur la méthode pour trouver le plus court chemin mais il s'avère qu'une méthode était plus rapide et efficace (en termes de complexité). Cette méthode consiste à regrouper dans un tableau les sommets (en colonnes) et le nombre d'étapes (en lignes, nb de nœuds-1). Pour commencer, on établit tous les voisins du sommet de départ puis nous déterminons le plus petit chemin parmi ces voisins. Puis cela revient à résoudre le problème précédent avec comme nouveau sommet de départ le sommet voisin courant. En répétant ces étapes (nbNoeuds-1) fois sans jamais repasser au même endroit. Ainsi les derniers éléments de chaque colonne représentent le chemin le plus court pour aller au sommet de cette même colonne.

2.4 L'algorithme de résolution du problème

Afin de résoudre le problème posé, nous devions prendre en compte notre graphe (carte), le type de patients et l'autonomie des véhicules. Dans ce développement à chaque fois que l'on dira que l'on vérifie si un véhicule peut aller à un endroit, cela signifie que l'on vérifie qu'il a la batterie nécessaire pour ne pas passer en dessous de 20% de batterie sur son trajet. Pour cela nous avons décidé de faire un algorithme qui se décompose en plusieurs étapes. Premièrement il vérifie si le trajet est possible en une étape avec le véhicule NI-NH, si c'est le cas il le fait, sinon il cherche s'il peut le faire avec des étapes de rechargement. Pour cela, on fait un test avec un véhicule NI-NH fictif. Ce véhicule fictif vérifie s'il peut aller au prochain sommet ayant une borne de rechargement qui se trouve sur son trajet, s'il peut y aller il y va, sinon il dit que ce n'est pas possible et on test avec le second véhicule. S'il il pouvait y aller il se trouve alors on rentre dans une boucle, celle-ci vérifie s'il peut aller au sommet final, si oui il y va sinon on vérifie s'il peut aller au prochain sommet qui a une borne sur son chemine, s'il peut y aller il y va et il recommence la boucle depuis là, sinon il recharge puis recommence la boucle. Si le véhicule fictif arrive à se rendre jusqu'au sommet final on envoie alors un vrai véhicule NI-NH. Si le véhicule fictif ne se rend jamais au sommet final alors on effectue exactement le même algorithme avec un véhicule

LI-ion cette fois ci. Soit le véhicule fictif LI-ion peut se rendre jusqu'au bout et alors on envoie un vrai véhicule LI-ion, soit il ne peut pas et à ce moment-là on ne peut pas réaliser le trajet.

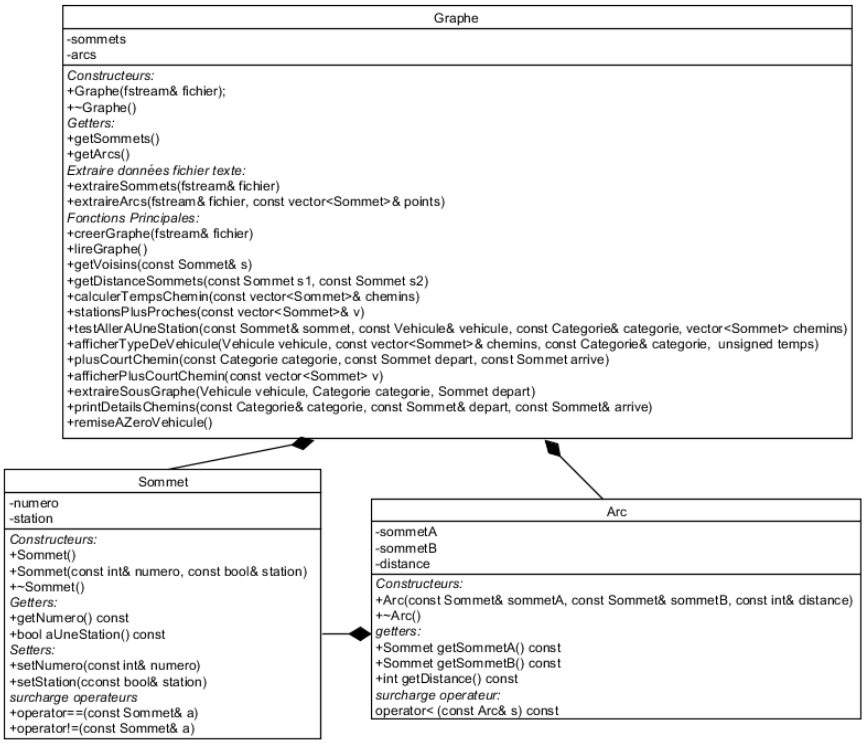
A chaque fois en fonction des cas, on envoie un message disant ce qu'il s'est passé. Si aucun véhicule ne peut effectuer le trajet on indique que le trajet est impossible, sinon on retourne le trajet effectué (les sommets parcourus), le véhicule utilisé et la batterie à l'arrivée.

2.5 L'implémentation du menu et de la gestion d'erreurs dans celui-ci

Une fois l'algorithme développé, il était nécessaire de créer une interface facile d'utilisation pour permettre à tout utilisateur, même sans connaissance précise en technologie informatique, d'utiliser notre programme. Au départ, implémenter le menu était assez simple car il faut juste créer l'algorithme de choix et appeler la bonne méthode dans chaque choix. Cependant, la construction de cette fenêtre sur notre code est compliquée car elle nécessite de gérer les erreurs pouvant être commises par l'utilisateur lors de son utilisation. Afin de d'éviter toutes erreurs causées par les entrées de l'utilisateur, nous avons élaboré un tout petit algorithme correspondant à chaque entrée afin de vérifier si elles étaient conformes à nos attentes, permettant alors d'annoncer l'erreur ou alors de continuer l'exécution normale du programme.

2.6 Diagramme de classes de la solution

Image n°1 : Diagramme de Classes:



n.b : Nos fichiers Main et menu ne sont pas dans le diagramme de classe car ce ne sont pas des classes et donc ils ne possèdent pas de méthodes.

3. Difficultés

3.1 Introduction

Lors de la réalisation de ce travail, nous avons bien sur rencontré certaines difficultés. Certaines d'entre elles ont pu être surmontées alors que pour d'autres, nous avons dû faire avec et nous adapter. C'est pourquoi nous allons vous décrire ces difficultés qui nous ont ralenti tout au long de ce projet et les solutions que nous avons y apporter.

3.2 Le langage de programmation

Une des premières étapes fut de déterminer le langage de programmation dans lequel nous allions coder durant le projet parmi C++, Python et Java. Rapidement nous avons choisi C++ car c'était le langage dans lequel nous étions le plus à l'aise, grâce au cours de programmation orienté objet que nous avons tous les trois passés. Ce fut peut-être une erreur car Python est plus productif et aurait plus été adapté à la situation. Par exemple Python peut détecter automatiquement les types des variables (C++ le peut aussi avec « auto » mais est beaucoup moins effectif et fiable que Python). Dans le cadre du premier problème que nous avons décrit ci-dessus, cette fonctionnalité aurait surement permis d'éviter cette erreur d'exécution et nous aurais certainement sauvé beaucoup de temps.

Nous n'avons pas trouvé de solution à ce problème, après avoir avancé jusque-là, changer de langage de programmation nous aurais fait perdre beaucoup de temps. De plus nous ne maîtrisons pas très bien Python.

3.3 Les erreurs de compilation et d'exécution

Comme toujours, lorsque l'on code, que l'on tente d'implémenter un algorithme dans ce cas, il y a des erreurs qui interviennent à la compilation et à l'exécution. Le code de ce TP était tout de même conséquent et donc nous avons rencontré des dizaines d'erreurs tout au long de notre travail. L'une d'entre elles nous à fait perdre plus de temps, en effet, en compilant la méthode ***calculerIndexDernierElement***, il s'est avéré que nous avions une boucle infinie. Nous décidons donc de déboguer étape par étape pour trouver où était l'erreur et on se rend compte que notre variable prend une valeur extrêmement grande et hors de l'index. Il était à ce moment là difficile de déterminer l'erreur car notre algorithme semblait correct et nous n'avions pas d'idées pour la résoudre malgré les indications du débogueur de Visual Studio.

Après une longue période de réflexion nous avons enfin trouvé l'erreur. Dans la fonction lambda de la méthode ***calculerIndexDernierElement*** nous parcourons un vecteur de la fin vers le début (en utilisant *i*--) et nous demandons à la méthode de break lorsque la valeur de i devient négative. À cet endroit nous avions déclaré *i* comme un *unsigned int*, par habitude, afin d'économiser de l'espace mémoire, donc notre *i* ne devenait jamais négatif et donc la méthode ne breakait jamais, ce qui provoquait une boucle infinie. De plus quand *i* devait devenir négatif il prenait une très grande valeur et c'est pourquoi *i* était hors de l'index. Nous avons donc remplacé le type de *i* par un *int* et notre méthode s'est instantanément mise à fonctionner. Cette simple erreur de type nous a fait perdre beaucoup de temps et d'énergie car elle était difficile à trouver mais c'est grâce à de la persévérance et de la concentration que nous l'avons résolue.

3.4 L'adaptation de l'algorithme de Dijkstra

L'algorithme de Dijkstra comme nous l'avons expliqué permet de trouver le plus court chemin entre deux sommets d'un graphe, il est assez complexe et demande déjà un certain temps à comprendre sur papier. Durant ce travail nous avons dû, dans la méthode ***plusCourtChemin()***, le coder. C'est là qu'était la partie la plus complexe du devoir car nous avons eu du mal à adapter avec des tableaux et vecteurs ce qui nous semblait plus simple à la main. En effet, à la main, avec un peu d'entrainement on arrive à l'exécuter rapidement mais cela devient difficile quand il faut l'adapter avec des objets et des classes.

La solution que nous avons trouvée, après longue réflexion, ressemble fortement à la manière écrite que nous avions pour exécuter l'algorithme en remplissant un tableau dont les colonnes sont des sommets et les lignes des étapes dans un chemin vers un deuxième sommet. Il fallait tout de même s'adapter à toutes les exceptions et gérer les cas particuliers tels différents *cheminLePLusCourt* de même longueur à gérer dans certains cas. On a donc un tableau à deux dimensions qui à autant de lignes que d'étapes pour la résolution du problème.

3.5 La gestion des erreurs dans le menu

À ce moment-là, on doit gérer tous les cas possibles de bogues dans notre menu. Le menu étant composé de nombreux sous-menus il y avait donc un nombre très élevé d'exceptions à gérer pour que le programme fonctionne en toute situation.

Pour être sûrs de cela nous avons établi une liste de tous les bogues possibles en utilisant un tableau à double entrée. Ensuite il suffisait d'établir un petit algorithme de gestion d'exception pour chaque cas mis en avant. Nous avons essayé de faire crasher notre menu par tous les moyens, puis résolu chacun des problèmes rencontrés jusqu'à-ce que notre menu fonctionne en toute circonstance.

4. Conclusion

Pour conclure, ce laboratoire nous a permis de réviser certaines notions que nous avions acquises précédemment mais aussi et surtout d'en apprendre et d'appliquer certaines autres. En effet, nous avons pu revoir et réutiliser la programmation orientée objet, comment lire un fichier texte et en extraire les informations, mais aussi la création d'un menu et enfin la gestion des exceptions (manipulation des flux d'entrées). Nous avons aussi appris à appliquer ce que nous avions appris des graphes dans la partie théorique du cours, à le coder, et surtout à l'adapter à la situation du réseau d'ambulance qui prenait beaucoup de facteurs tels que la batterie, le type de véhicule et la gravité de l'état du patient. Durant ce TP, nous avons aussi dû nous organiser au sein du groupe pour la répartition des tâches et nous organiser pour gérer le temps avant la remise. Heureusement, nous avons eu du temps supplémentaire qui nous a été précieux car ce travail représentait une charge de travail conséquente dans cette période où nous avons des examens et de nombreuses remises dans différents cours. Durant le temps supplémentaire nous avons pu améliorer notre gestion des erreurs, dans l'interface mais aussi dans l'algorithme, il nous a aussi permis de réaliser un rapport plus complet.