

LOG2810

STRUCTURES DISCRÈTES

**TP1 : GRAPHES**

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

VINCENT FORTIN

DOMINIQUE PICHÉ

IBRAHIMA SÉGA JR SANGARÉ

REMIS LE VENDREDI 28 OCTOBRE 2016

**Introduction :**

L’avènement des véhicules autonomes et électriques a soulevé plusieurs problématiques lorsqu’il a fallu prendre en compte les trajets à longue distance. Ces véhicules doivent être dotés de système pour reconnaitre leur mode fonctionnement et de ravitaillement. De plus, il faudra assurer la distance la plus courte entre le point de départ et la destination. C’est pourquoi, une application a été prévue par le département de génie informatique et de génie logiciel pour régler la situation. En fait, il nous a été demandé d’appliquer nos connaissances en structures discrètes dans le contexte d’un voyage à bord d’un véhicule autonome et de sources d’énergie multiples. Le circuit de l’engin est fourni par une carte pourvue de stations de rechargement évitant une panne éventuelle lorsqu’il y manque de carburant. Dans le but de faciliter le développement de l’application, la carte a été pensée comme un graphe orienté et pondéré où les arcs (orientés) correspondent aux chemins potentiels à emprunter. La représentation du graphe indiquerait la distance en kilomètres entre chaque sommet. Les sommets possédant des indiquent toutes les points intermédiaires et entre le départ et l’arrivée. Grâce leurs degrés et leurs propriétés, ils constitueront le principal outil de décision du programme. Un menu interactif sera à disposition de l’utilisateur pour choisir les caractéristiques du véhicule ou générer le trajet par exemple. En bref, notre travail aura été d’écrire le code source de l’application à l’aide de Microsoft Visual Studio en C++ et satisfaire les exigences.

**2. Présentation des travaux :**

Lors de l’élaboration de l’application, nous avons procédé au codage en utilisant une approche orientée objet. Cela a rendu la tâche plus simple pour mettre en évidence les fonctionnalités à implémenter. La solution contient donc les quatre classes suivantes : Vehicule, Graphe, Arc et Sommet.

**2.1 Vehicule :**

Tout d’abord, la classe Vehicule représente symboliquement le véhicule qui sera utilisé lors du voyage. Y sont inclus les attributs privés qui permettent de connaitre la source d’énergie utilisée et les autonomies maximale et restante du véhicule. Mis à part les méthodes d’accès et de modification pour les attributs, rien d’autre d’important n’est signalé pour cette classe.

**2.2 Graphe :**

Ensuite, la classe Graphe constitue la partie la plus importante du projet car elle permet de contrôler quasiment tout ce qui se passe comme événement avec le véhicule, les arcs et les sommets. Les fonctions les plus importantes (*creerGraphe()*, *lireGraphe()*, *extractionGraphe()* et *plusCourtChemin()*) ont été liées à cette classe, car nous avons jugé qu’il serait judicieux de donner un nombre conséquent de responsabilités toutefois pertinentes. Étant donné que le graphe est formé d’arcs et de sommets, deux attributs de type vecteur contenant des pointeurs ont été respectivement mis à disposition de la classe. Il est à noter qu’il y a en effet, une relation de composition entre le graphe et ses deux ensembles (arcs et sommets). D’autres méthodes ont été rajoutées à la classe afin d’améliorer la qualité générale du code.

**2.2.1 Fonctions :**

- **creerGraphe** : cette fonction s’occupe en réalité de réunir toute l’information concernant chaque sommet et arc existant dans un fichier texte. Également, elle établit les liens entre les sommets pour enfin définir la carte au complet.

- **lireGraphe** : cette fonction sert à représenter comme un couple les connexions à l’intérieur de la carte (arcs et sommets) en parcourant le vecteur de sommets, puis en affichant tous les sommets voisins. Le type de sommets est aussi pris en compte.

- **extractionGraphe**: il s’agit de construire et retourner un nouveau graphe formé en recherchant un type de sommets et d’arcs en particulier. Ce graphe est alors un sous-graphe extrait du graphe départ. Il faut ajouter les arcs et sommets qui peuvent être parcourus et veiller à ce qu’ils ne soient pas inclus plus d’une fois.

- **plusCourtChemin** : cette fonction est l’une des utiles pour vérifier le bon fonctionnement du programme. Grâce à l’algorithme d’Edsger Dijkstra, il est question de trouver le chemin le plus court pour arriver à destination tout en tenant compte de l’autonomie et de la source d’énergie du véhicule. Dans notre cas, nous avons utilisé un vecteur de pour conserver les chemins empruntés et plusieurs autres chemins pour effectuer la comparaison entre les arcs et les sommets. Un vecteur temporaire contenant la carte au complet est utilisée pour effectuer les comparaisons de distance. Les sommets comparés qui sont retenus sont rajoutées au trajet minimalau fur et à mesure et il y a une réduction du nombre de sommets dans le vecteur temporaire par conséquent. À chaque sommet, il faut vérifier si les extrémités conduisent au chemin le plus court. On garde en mémoire la longueur des chemins empruntés pour donner la longueur totale parcourue. Des cas exceptions ont été définis en plus, par exemple, pour éviter des sommets inexistants. Finalement, s’affiche le parcours optimal.

En outre, des fonctions utiles à savoir *findNext()*, *findSommet()* et *viderGraphe()* ont été introduites pour ne pas surcharger inutilement les méthodes. Elles servent respectivement à retourner l’index d’un sommet voisin à vérifier, trouver un sommet à l’aide de son identifiant, et vider les vecteurs du graphe.

**2.3 Arc :**

Cette classe contient comme attendu des attributs privés comme la longueur de l’arc ou la distance séparant ses deux sommets. Ensuite, un identifiant pour décrire l’arc et un tableau à deux entrées représentants les extrémités. À cela s’ajoutent comme on peut l’attendre, des méthodes d’accès et de modification des attributs. Pour finir, trois autres fonctions *contains()*, *getTerminal()* et *hasType()* servent vérifier l’existence d’un sommet, obtenir l’extrémité de l’arc et de vérifier le type de sommet associé à l’arc. On notera une association entre l’arc et le sommet dans cette configuration.

**2.4 Sommet :**

Pour finir, la classe sommet est munie d’un identifiant, un type, l’ensemble de ses arcs et son degré, à titre d’attributs. Excepté les accesseurs et les modificateurs, une méthode *ajouterVoisin()* qui permet d’ajouter un arc à l’ensemble d’arcs existant et augmenter le degré du sommet. Deux autres méthodes *isConnectedTo()* et *hasNeighborType()* servent à vérifier les sommets voisins d’un certain sommet puis de s’assurer de l’existence d’un type de sommets en particulier en parcourant un tableau de voisins.

La figure suivante représente le diagramme de classes de la solution et les relations entre celles-ci.

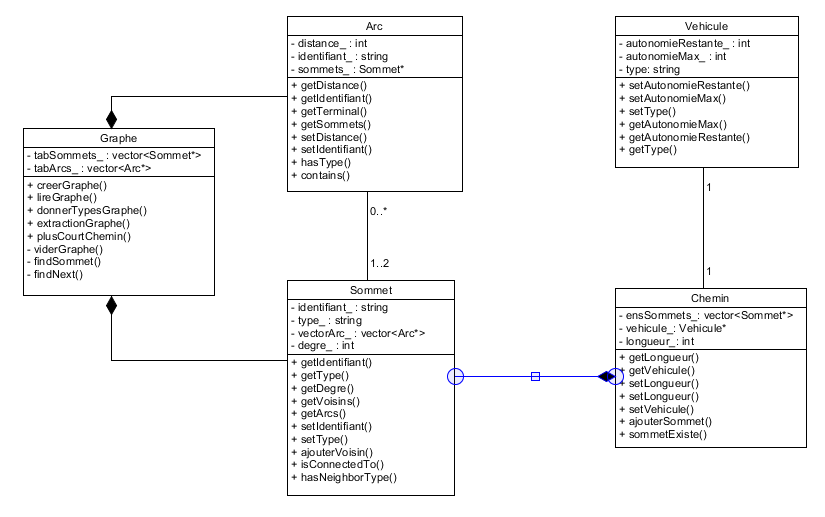


Figure 2 : Diagramme de classes de la solution

**2.6 Chemin :**

Cette classe permet collectionner les sommets qui ont été parcourus lors d’une recherche de trajet par exemple. Elle contient un vecteur de sommets qui ont été parcourus, un véhicule qui la parcourt et sa longueur ou distance totale. La méthode *ajouterSommet()* permet de faire grandir le chemin tout en affectant le carburant du véhicule qui lui est associé, ainsi que la distance. Une autre méthode, *sommetExiste()* permet de faire des comparaisons entre un sommet quelconque et un autre présent dans le chemin. À part cela il y a comme d’habitude des méthodes d’accès et de modification.

**2.7 Fonction principale :**

La fonction *main()* comprend le menu interactif censé représenter une ou plusieurs utilisations de l’application. L’utilisateur doit être capable, à travers les options fournies, d’indiquer d’abord les caractéristiques du véhicule qui sera utilisé pour effectuer le trajet (a). Si l’utilisateur ne rentre pas les bonnes données, il en est averti. Il peut ordonner la création de la carte ou graphe (b) en fournissant le nom du fichier texte contenant les informations d’arcs et sommets. Par la suite, il peut commencer son trajet avec l’option du court chemin possible (c). Il y a bien un ordre dans lequel la première et la troisième option s’effectuent pour éviter toute confusion. Une fonction *afficherMenu()* doit s’occuper d’informer l’utilisateur s’il veut continuer ou quitter. L’autonomie du véhicule est conservée si le véhicule devait faire plusieurs trajets. En tout, il y a trois cas de figures dans le programme. Le premier représente l’option a, le second, l’option b, et le dernier simule le trajet.

**3. Difficultés rencontrées :**

Parmi les différentes méthodes à implémenter, la méthode *plusCourtChemin* fut définitivement la plus compliquée à implémenter, malgré le fait qu’il soit relativement simple de comprendre l’algorithme de Djikstra. Cela étant dû la récursivité de la méthode qui rend rapidement l’algorithme à trouver plus difficile. Bien sûr, il y eut aussi les multiples bugs tout au long du TP qui ont augmenté la durée du travail, malgré qu’un grand nombre de ces bugs aient été des erreurs d’inattention, réglées à l’aide de l’outil de débogage intégré à Visual Studio.

**Conclusion :**

À l’aide de l’algorithme de Djikstra, ainsi que du langage C++ et du logiciel Microsoft Visual Studio, nous avons réussi à créer un programme qui prend en compte la réalité du futur, soit un monde où il y a de plus en plus de véhicules électriques et de stations de rechargement électriques. Le conducteur devant alors adapter son trajet, afin d’emprunter le trajet le plus court, tout en ne tombant pas en panne, ce que notre programme permet d’effectuer.

Ce TP nous a donc permis, non-seulement d’en apprendre davantage sur les graphes et l’algorithme du plus court chemin de Djikstra et nous a permi de rafraichir nos connaissances en orienté-objet en C++ ce que nous avons peu fait cette session jusqu’à présent.

Nous espérons que le prochain laboratoire sera tout aussi éducatif, mais un peu plus court si possible.