LOG2810

Structures discrètes

Polytechnique Montréal

TP1 : Graphes

Gabriel Bourgault - 1794069

Frédéric Hallé – 1802024

28 octobre 2016

# Introduction :

Tous ont déjà rêvé un jour ou l’autre à une voiture qui serait en mesure de nous mener du point A au point B, sans que nous ayons à la conduire. Nous pourrions donc étudier, regarder une série sur Netflix, ou même prendre une sieste tout en se rendant, en toute sécurité à notre destination. Ces voitures sont maintenant choses de la vie actuelle. Elles sont toutefois coûteuses, mais accessible au grand public.

Dans le cadre du cours de Structures discrètes, nous devons concevoir un système de navigation permettant d’assurer un ravitaillement en énergie qui soit optimal pour le véhicule selon les trajets qu’il doit parcourir. Par exemple, comme certaines voitures ont un mode de fonctionnement un peu plus archaïque, il faut prévoir des arrêts à la station de service pour le ravitaillement. Une voiture électrique, pour sa part devra s’assurer de la présence de points de charge sur le trajet.

Nous verrons tout d’abord un aperçu de la tâche que nous avons à réaliser, en prenant soin de noter les particularités et éléments requis de notre système. Nous parcourrons certaines conditions de fonctionnement de ce système, qui sera suivi par l’explication du fonctionnement des graphes dans notre système et comment nous avons exploité certaines propriétés. Nous verrons aussi par après les fonctions que nous devions obligatoirement implémenter dans notre système. Leur fonctionnement sera démontré avec un diagramme de classe, qui nous permettra aussi du même coup de comprendre la structure et le fonctionnement de tout ce système. Finalement, nous présenterons les principales difficultés que nous avons rencontré ainsi que les méthodes employées pour les surmonter.

# Présentation des travaux

## Énoncé de la tâche à réaliser :

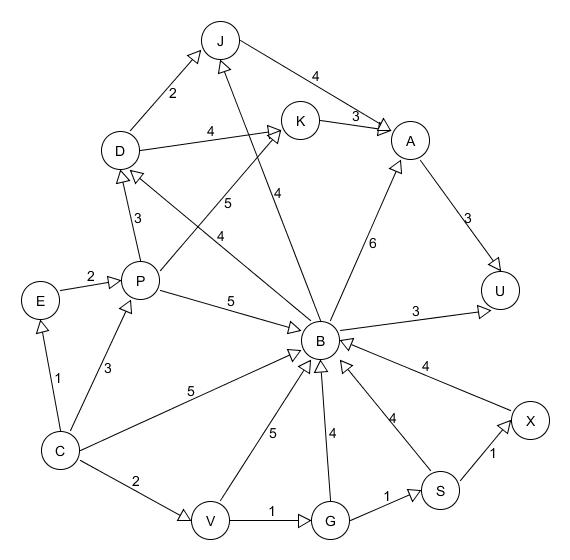
Un des éléments les plus importants de ce système est de s’assurer qu’il soit modulaire autant que possible. Cet aspect du logiciel commence avec les caractéristiques du véhicule. Le système doit fonctionner autant pour une voiture 100% électrique, que pour une voiture hybride ou à essence. L’usager doit spécifier plusieurs informations concernant son véhicule avant de démarrer un trajet. Il doit tout d’abord spécifier le type d’énergie que requiert son véhicule pour fonctionner. Il doit aussi spécifier son autonomie maximale, qui est la distance maximale en kilomètres qu’il peut parcourir avec un plein, ou une charge. Afin de réduire la complexité du problème, nous ne prenons pas en compte les différents facteurs physiques qui influencent la consommation d’énergie, tel que le vent, les côtes sur le chemin ou le trafic. Ces données sont d’une importance capitale pour ce système, car c’est ce qui nous permettra de prévoir les points de ravitaillement sur le chemin, et ainsi, éviter d’avoir recours à la dépanneuse. Cet aspect constitue la première option du menu du système.

Le second aspect d’importance de ce système, est de s’assurer que nous ne sommes pas restreints à une seule zone. Nous voulons être en mesure d’utiliser le système dans différentes, villes, provinces et même différents pays. Il doit donc être possible de modifier la carte que le logiciel aura à gérer. La carte sera spécifiée par l’usager en donnant le nom du fichier qui la contient dans l’option (b) du menu principal.

Le fonctionnement de la carte pour déterminer le meilleur itinéraire est l’un des éléments les plus complexes à traiter pour ce système. Tout d’abord, la carte sera vue par le logiciel comme un gros graphe orienté.

Le premier élément qui sera traité dans ce système sera les sommets du graphe. Ils peuvent représenter plusieurs choses, tel que des extrémités de transport. En gros, ils représentent les différents points de départ et d’arrivée possibles. C’est aussi sur ces sommets que sont les points de ravitaillement. Chaque sommet possède l’information sur les sources d’énergie qu’il peut distribuer, soit, *null*, si le sommet n’a aucune machine de ravitaillement, *essence*, s’il y a une station de service, ou *elec*, s’il y a une station de charge électrique. Un sommet peut distribuer aucune énergie, tout comme il peut distribuer les deux.

Un autre élément qui sera traité dans ce graphe sont les routes. Comme il s’agit d’un graphe orienté, on peut établir une comparaison avec les routes à sens unique. Cela veut donc dire qu’un itinéraire est possible pour se rendre du point A au point B, mais que ce ne sera pas nécessairement le même itinéraire pour faire du point B au point A. Pour pouvoir faire un aller-retour entre deux points, il faut deux arcs orientés de sens opposés.

Chaque sommet du graphe possède un degré entrant et un degré sortant. Ce degré indique le nombre d’arcs entrants et sortants de ce nœud. Ces arcs nous permettent de créer différents chemins, qui correspondent au parcours pour se rendre du point A au point B pour chaque utilisateur. Ces chemins sont composés d’un ou plusieurs arcs, qui passent par différents sommets.

Ces cartes/graphes peuvent parfois être très complexes et très difficiles à analyser. Une des façons à employer pour en réduire la complexité est d’avoir recours aux sous-graphes. Comme il est possible de le deviner, un sous-graphes est contenu dans un graphe. Par exemple, on peut définir B peut être un sous-graphe de A, mais il doit répondre à quelques conditions. Par exemple, B doit lui-même être un graphe, l’ensemble des sommets de A doit contenir l’ensemble des sommets de B. La même condition s’applique pour les arcs des graphes A et B.

Figure 1 Voici un exemple de graphe qu'il est possible de créer à partir d'une carte. On remarque principalement valeurs associées à chaque arc pour en spécifier sa longueur.

Nous aurons aussi recours aux graphes valués C’est cette donnée qui nous permet de calculer l’autonomie restante des véhicules selon leur autonomie initiale. Les graphes valués contiennent une donnée sur chaque arc. Cette valeur représente la longueur de cet arc en km. On peut donc calculer la longueur totale de ce chemin en faisant la somme des distances de chaque arc parcouru. Avec cette donnée, nous pouvons tout d’abord nous assurer que l’autonomie du véhicule est plus grande que la distance à parcourir et nous assurer que l’autonomie restante nous permette de se rendre à une station de ravitaillement. Nous voulons éviter qu’une voiture tombe en panne en allant faire le plein ou en allant se recharger.

## Notre solution :

Tout d’abord, pour élaborer du mieux possible ce système, nous avons quelques conditions à respecter et quelques classes et fonctions que nous devons obligatoirement utiliser.

Nous devons premièrement créer une fonction nommée « *creerGraphe()* », qui permettra de créer le graphe et représenter les routes en arcs, ainsi que les stations de chargement en nœuds, à partir du nom de fichier qui contient la carte.

Nous devrons aussi créer une fonction « l*ireGraphe* », qui nous permettra d’afficher le graphe créé. L’affichage de ce graphe constitue un élément complexe de ce système, car nous devons indiquer pour chaque sommet, de quel type de station il s’agit, ainsi que la liste de ses sommets voisins. Voici un exemple d’affichage : (station1, typestation1, (station\_voisine1.1, station\_voisine1.2, ..., station\_voisine1.n)) (station2, typestation2, (station\_voisine2.1, station voisine2.2, ..., station voisine2.m)) …

La fonction *« extractionGraphe() »*, permet de déterminer l’ensemble des possibilités qui s’offrent à la voiture selon ses restrictions. Les résultats de cette fonction sont représentés sous forme de sous-graphe. Seules les routes empruntables selon l’autonomie restante et les points de ravitaillement y sont présentés.

Une autre fonction que nous devrons écrire est la fonction « *plusCourtChemin()* ». Cette fonction constitue l’essence même de tout le système. C’est cette fonction qui lui permet de réaliser la tâche pour laquelle il a été imaginé et conçu, soit de déterminer le chemin le plus court tout en répondant aux diverses conditions fixées par la distance à parcourir. Cette fonction sera inspirée de l’algorithme de Dijkstra[[1]](#footnote-1), qui en effet, est utilisé pour calculer la plus courte distance entre deux points dans un graphe pondéré. Cet algorithme construit un sous-graphe classé par ordre croissant selon la distance minimale du sommet de départ. Le sous-graphe initial n’est composé que du point de départ. À chaque itération, on choisit en dehors du sous-graphe, le sommet qui est à une distance minimale du point de départ. On ajoute ce somment au sous-graphe. Lorsque ce sommet est ajouté, on répète le processus, mais en s’assurant de minimiser la distance en prenant en compte les sommets qui ont été ajoutés précédemment au chemin.

Comme mentionné plus haut, il est très important de nous assurer que même une fois arrivé à destination, la voiture a encore assez d’énergie pour au moins se rendre à la station de ravitaillement la plus proche en cas de besoin. Pour ce faire, nous avons imaginé trois façons de valider le tout. La première façon à laquelle nous avons pensé, est de calculer la plus courte distance dans le graphe, du point de départ à la station de service la plus proche de notre point d’arrivée. Cette distance doit être plus petite que l’autonomie actuelle du véhicule avant de démarrer le chemin. Si par malheur, la distance de ce chemin est plus grande que l’autonomie du véhicule, le système choisira un point de service pour arrêter sur le chemin et faire le plein et/ou recharger les piles. C’est l’algorithme de Dijkstra qui sera employé pour faire cela. La seconde option qui s’offrait à nous était de calculer le parcours en deux étapes. La première étant de de calculer la distance à parcourir pour se rendre au terminus. Si la distance à parcourir jusqu’à ce point est plus grande que l’autonomie, un arrêt sera forcé. Sinon, le transport peut démarrer. La seconde étape du calcul est de déterminer la station de service/ charge la plus proche du point d’arrivée. Si la somme de ces deux distances est plus grande que l’autonomie du véhicule, un arrêt sera nécessaire. Finalement, la troisième possibilité qui s’offrait à nous, est celle que nous avons choisi. Cette option se résume à faire faire le plein d’énergie au véhicule à chaque nœud qui lui permet de le faire. De cette façon, on s’assure que la voiture est presque toujours à son autonomie maximale. De cette façon, nous avons diminué de beaucoup la complexité de notre programme, tout en s’assurant qu’il n’avait aucune faille.

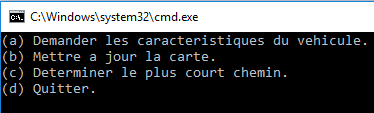
Finalement, le système devra toujours revenir au menu principal, qui lui permettra de modifier les caractéristiques du véhicule, de mettre à jour la carte, de déterminer le plus court chemin ou de quitter. Il nous a été demandé de bâtir le logiciel autour d’une petite interface simple et facile d’utilisation pour l’usager. Notre menu aura cette apparence, comme vous pouvez voir sur l’image ci-contre.

Figure 2 Interface simple du système

## Diagrammes de classes :

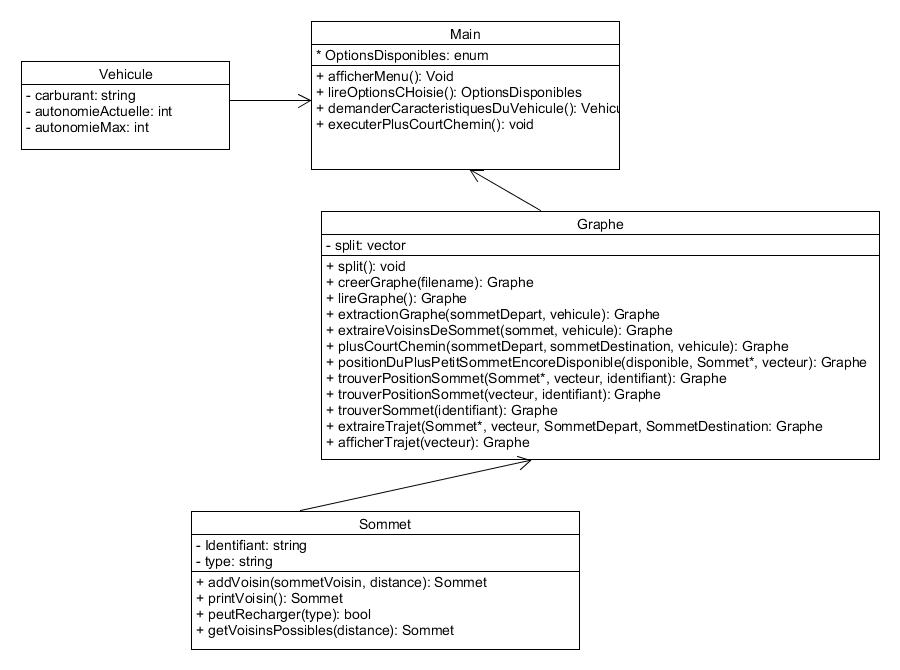
 Pour faciliter notre compréhension du problème et pour structurer le plus possible notre système, un diagramme de classe est nécessaire. Ce diagramme nous permet de s’assurer que notre code reste compréhensible et structuré autant que possible. Il nous permettait aussi d’assigner des rôles spécifiques aux classes. De cette façon, lorsqu’un bogue faisait surface, nous savions exactement dans quelle classe chercher pour le régler. Une des bonnes façons de penser en programmation se relate au « diviser pour régner », qui consiste à diviser le programme en classes liées entre elles, pour les raisons spécifiées plus haut. Voici donc le diagramme de classe qui représente l’architecture de notre système. Le fait d’avoir une structure comme cela, avec plusieurs subdivisions rend le logiciel beaucoup plus modulaire. Cela le rend ainsi plus facile à porter vers d’autres plateformes et systèmes d’exploitation.

Figure 3 Diagramme de classe représentant la structure de notre système

# Difficultés rencontrées et solutions apportées :

## Dijkstra :

La difficulté principale que nous avons rencontré était d’implémenter l’algorithme de Dijkstra à notre structure. Nous comprenions tout à fait comment fonctionnait cet algorithme, mais ce n’était pas la cause de nos soucis. Le problème venait plus de la structure de notre projet et de l’interaction entre les classes.

Ce même algorithme nous a causé beaucoup de troubles à implémenter, car nous n’étions pas certains de la bonne façon de le faire pour conserver sa complexité au minimum. C’est surtout au moment de s’assurer que la voiture ait assez d’autonomie pour se rendre à sa destination que l’algorithme nous causait des troubles. Comme mentionné plus haut, nous avions trois options pour réaliser cette option. Les deux premières, très semblables, nous permettaient de calculer la distance totale, ainsi que l’autonomie requise pour s’y rendre. Cependant, advenant le cas ou l’autonomie ne permettait pas de s’y rendre, tout devenait beaucoup plus compliqué, car nous étions forcés de sortir de la boucle de récursivité, chose qui n’est pas souhaitable. La solution que nous avons trouvé pour palier à ce problème est la troisième option que nous avons énoncé plus haut. Cette solution nous permet de s’assurer que la voiture ne soit jamais à court de carburant, ce qui est le but du système.

## extraireGraphe() :

Une des autres fonctions qui nous a causé beaucoup de soucis à coder était la fonction *extraireGraphe()*, car elle nous forçait aussi a implémenter de la récursivité. Cette fonction a comme rôle d’extraire le sous-graphes de chemins empruntables. La difficulté venait du fait que nous devions parcourir tous les sommets à partir du point de départ, en plus de s’assurer que l’autonomie du véhicule nous permettait de se rendre à chacun de ces points. Nous avons surmonté cette difficulté en créant une fonction récursive qui extrait les voisins disponibles pour chaque sommet, puis qui s’appelle elle-même sur chacun de ces sommets disponibles.

## Graphes bidirectionnels :

Nous avons rencontré une troisième difficulté de dernière minute. Cette difficulté est en lien avec la vérification dans le cas d’un arc bidirectionnel. Nous avons tenté de trouver une solution pour remédier à ce problème à l’aide d’une vérification et d’un booléen, mais cette vérification provoquait une autre erreur. Le système pouvait nous retourner une distance parcourue négative. Nous jugeons donc qu’il est mieux de ne pas supporter les arcs bidirectionnels que d’avoir des distances parcourues négatives. Cette erreur fait donc le programme tomber dans une boucle infinie. Quelques jours de plus nous auraient probablement permis de trouver une solution complète et adapter la structure de notre projet.

# Conclusion :

Pour conclure, notre système est fonctionnel et permet de déterminer les différents parcours possibles, tout en s’assurant que la voiture ne tombe jamais en panne. Nous avons été en mesure d’implémenter les fonctions obligatoires, *creerGraphe()*, *lireGraphe()*, *extractionGraphe()* et *plusCourtChemin()*. Nous sommes aussi en mesure de modifier les paramètres de la voiture et d’utiliser cette dernière dans plusieurs régions en modifiant la carte de l’environnement. La structure du système a aussi été pensée et visualisée avec un diagramme de clases qui est aussi fourni en annexes. Nous avons discuté des principales difficultés que nous avons rencontré et trouvé une façon de régler le blocage pour arriver à une solution. Nous croyons rencontrer toutes les exigences de ce système. Une amélioration qui serait bien à apporter à ce projet pour le pousser davantage, serait de l’intégrer aux systèmes de covoiturage tels que Uber au moment d’assigner une course à un conducteur, pour s’assurer que le véhicule aura une assez grande autonomie pour porter le client à sa destination sans avoir à arrêter faire le plein.

# Annexe :

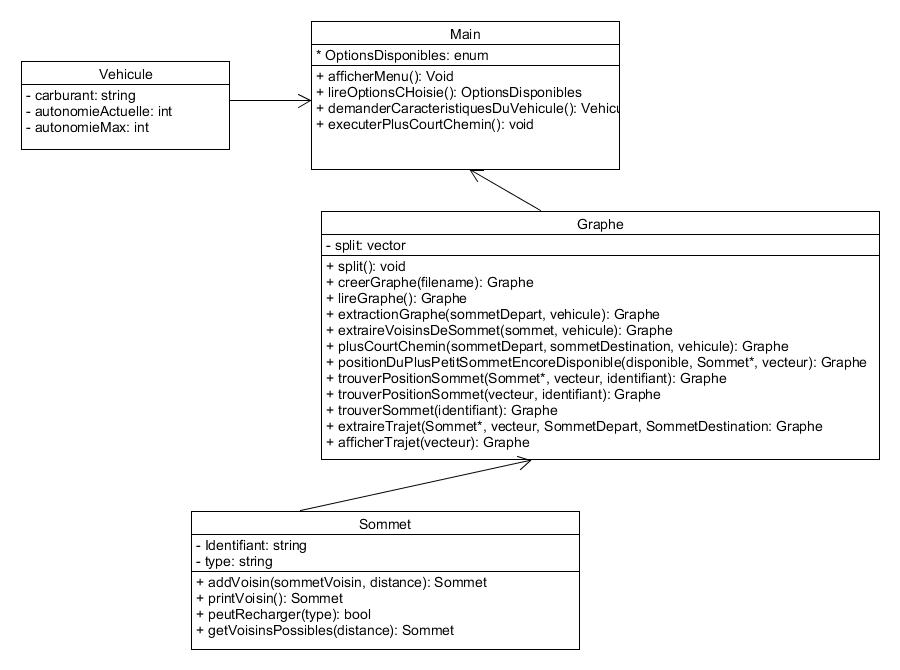


Figure 4 Diagramme de classes

1. Probablement pas la meilleure source possible, mais tout de même relativement complète dans le contexte de de rapport. <https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra?oldformat=true> [↑](#footnote-ref-1)