LOG2810

Structures discrètes

Polytechnique Montréal

TP2 : Automates et Machines à États

Gabriel Bourgault - 1794069

Frédéric Hallé – 1802024

6 décembre 2016

# Introduction :

Lors des dernières années, le problème du réchauffement climatique a pris beaucoup d’ampleur et de plus en plus de gens essaient de faire attention et porter des gestes plus écologiques. Ils cherchent à minimiser leur empreinte écologique. Le transport en commun fait partie de ces actions prises par les citoyens de la terre. Cependant, pour certains, parfois en régions éloignées, le transport en commun n’est pas la meilleure solution. Le transport collectif en région, n’est pas aussi développé, ni aussi flexible. La voiture reste donc bien souvent la meilleure solution. Une solution existe tout de même pour ceux qui se soucient de l’environnement et qui ont tout de même besoin de la voiture. Il s’agit de services d’autopartage. Ces services permettent de fournir un véhicule aux usagers selon leur demande. Ils peuvent donc ainsi minimiser les coûts liés à l’utilisation d’une voiture, puisqu’ils n’en sont pas propriétaires, et du même coup, porter des actions dans la bonne direction pour aider la planète.

Dans le cadre du cours de Structures discrètes, nous devons concevoir un système d’autopartage qui permettra aux citoyens d’une ville de pouvoir louer un véhicule pour effectuer leurs déplacements. Ce système, combiné aux voitures autonomes permettra aux usagers d’avoir la voiture à leur porte quand ils en auront besoin.

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 automates et machines à états 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 :

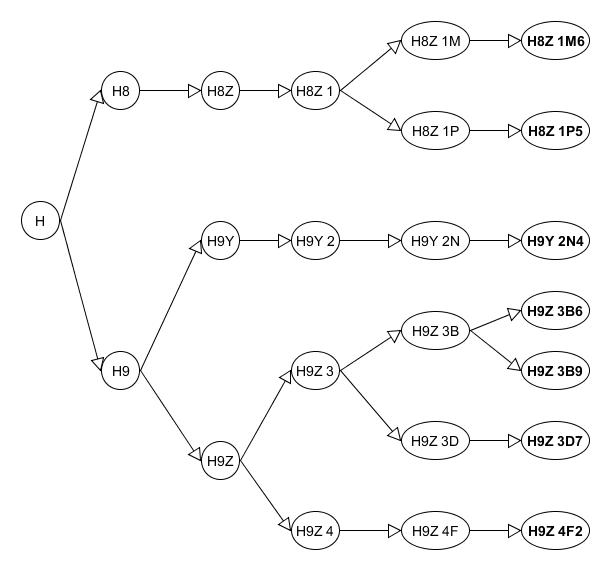
L’objectif de ce travail est de réaliser un test de viabilité dans le cadre de la mise en service d’un service d’autopartage pour la ville de Montréal. Tout d’abord, la ville est découpée en différentes zones. Chaque zone est ensuite sous-divisée en voisinages, dont chacun est défini par son code postal. Chacun de ces codes postaux sont présentés dans les fichiers zone1.txt à zone3.txt.

L’idée à la base de ce système est de toujours avoir une voiture disponible pour un usager. Bien entendu, pour minimiser le nombre de clients mécontents et maximiser le nombre de consommateurs satisfaits, il faut offrir un service de qualité. Ce service de qualité passe tout d’abord par l’accès aux véhicules. Il doit donc exister en permanence, un certain équilibre du nombre de véhicules disponibles, dans toutes les zones de la ville.

En effet, lorsqu’un utilisateur fait une demande pour un véhicule, un véhicule libre stationné dans son voisinage lui est acheminé. Advenant le cas où il n’y aurait pas de véhicule libre dans son voisinage actuel, ce sera donc un véhicule libre dans la même zone qui sera livré. Si par malheur, en grosse période d’achalandage, aucun véhicule n’est disponible dans la zone, alors la demande du client sera refusée. Le refus d’une demande peut avoir des impacts non seulement économiques, mais aussi sur la réputation de la compagnie du système. Afin d’assurer la meilleure expérience client possible, nous avons donc comme objectif d’équilibrer le plus possible le nombre de véhicules libres entre les zones.

Pour mettre en place un tel système, il faut tout d’abord s’assurer de compléter certaines tâches. Par exemple, la première consiste à implémenter les zones. Il faut s’assurer de créer un modèle qui soit facilement interprétable par l’humain, tout en étant facile à implémenter dans un contexte de programmation.

Comme mentionné plus tôt, la ville est séparée en zones. Chaque zone regroupe plusieurs quartiers, qui chacun, peuvent être associés à un code postal. C’est ce code postal qui nous permettra de déterminer dans quelle zone le client se trouve actuellement Nous fonctionnerons avec un système d’automates et de machines à états pour déterminer dans quelle zone le client se trouve actuellement. À l’aide de ce code postal, nous pouvons créer un automate, ou les états finis correspondent à un code postal complet. Les arcs correspondent aussi à chaque caractère du code postal. On pourrait donc représenter le lexique suivant : {H8Z 1M6, H8Z 1P5, H9Y 2N4, H9Z 3B6, H9Z 3B9, H9Z 3D7, H9Z 4F2} avec l’automate suivant :

Ainsi, nous générons un automate de ce genre par zone. Cependant, chaque zone contient bien plus que 7 codes postaux. On parle plus d’environ 25 par zone. On peut s’imaginer la complexité d’un automate pour 25 éléments, sachant que celui-ci n’en contient que 7.

Une fois l’automate généré avec le lexique qui nous a été fourni, et qu’un nouveau client fait une demande pour avoir un véhicule, il faut déterminer quel véhicule lui acheminer selon sa localisation qu’il connait maintenant. Si un véhicule est libre et dans le même voisinage que le client, cette voiture lui est directement acheminée. Si aucune voiture n’est disponible dans le voisinage actuel, l’usager recevra une voiture qui provient d’un autre voisinage, mais tout de même de la même zone. Dans cette situation, un cas peut sembler négligé. Le cas où il n’y aurait pas de voiture disponible dans la zone. Ce cas est couvert par l’algorithme d’équilibrage.

Figure 1 - Automate qui représente les codes postaux suivants : {H8Z 1M6, H8Z 1P5, H9Y 2N4, H9Z 3B6, H9Z 3B9, H9Z 3D7, H9Z 4F2}

Comme mentionné, l’algorithme d’équilibrage permet d’éliminer le cas où il n’y aurait pas de voiture disponible dans la zone. Ce cas pourrait se produire, car les utilisateurs ne sont pas limités à leur zone une fois qu’ils sont au volant du véhicule. Ils peuvent sortir de la zone, ce qui pourrait débalancer les zones, laissant ainsi des zones avec aucun véhicule disponible. On peut comparer cet algorithme à celui des Bixi de Montréal. Le système doit s’assurer qu’il y ait toujours de vélos disponibles pour emprunter aux diverses stations, mais aussi qu’il y ait des places disponibles pour que les utilisateurs qui en ont terminé puissent le remettre. La façon de procéder est relativement simple. Dès qu’un déséquilibre est remarqué dans une zone, un véhicule provenant d’une zone où il y a plusieurs véhicules disponibles est acheminé vers cette zone en déficit.

## 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.

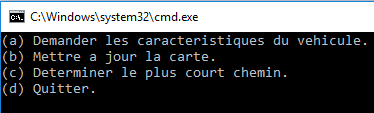
Finalement, le système devra toujours revenir au menu principal, qui lui permettra de modifier les caractéristiques du véhicule, de mettre à jour la carte, de déterminer le plus court chemin ou de quitter. Il nous a été demandé de bâtir le logiciel autour d’une petite interface simple et facile d’utilisation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

Figure 3 Interface simple du système

## Diagrammes de classes :

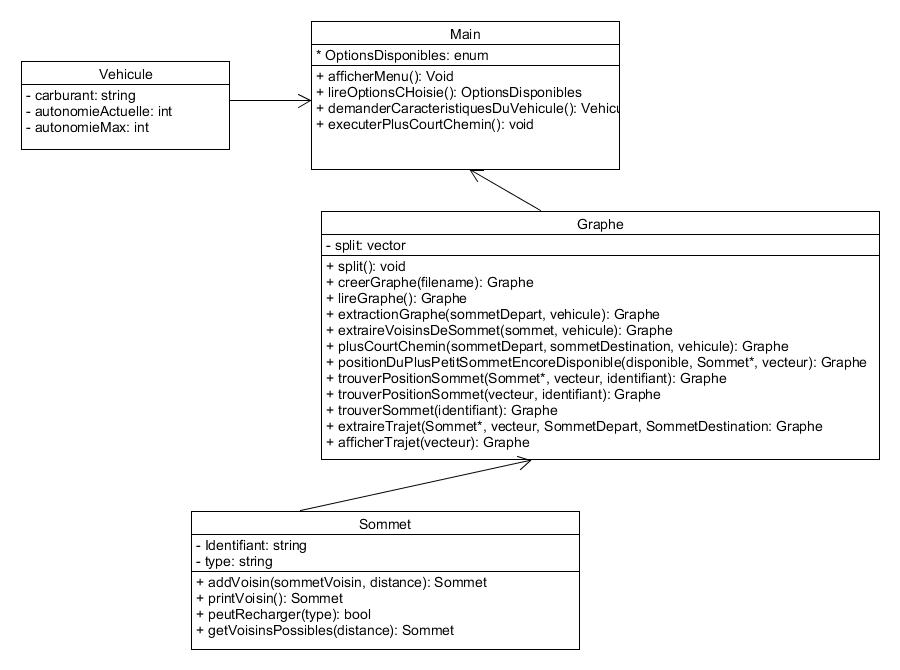
 Pour faciliter notre compréhension du problème et pour structurer le plus possible notre système, un diagramme de classe est nécessaire. Ce diagramme nous permet de s’assurer que notre code reste compréhensible et structuré autant que possible. Il nous permettait aussi d’assigner des rôles spécifiques 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 4 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 faisait tomber le programme dans une boucle infinie. Quelques jours de plus nous auraient probablement permis de trouver une solution complète et adapter la structure de notre projet. Pour les fins du TP, nous avons simplement bloqué l’utilisation de graphes contenants des arcs bidirectionnels en amont. Le système en tout de même entièrement fonctionnel pour des graphes contenant uniquement des liens unidirectionnels.

# 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 :

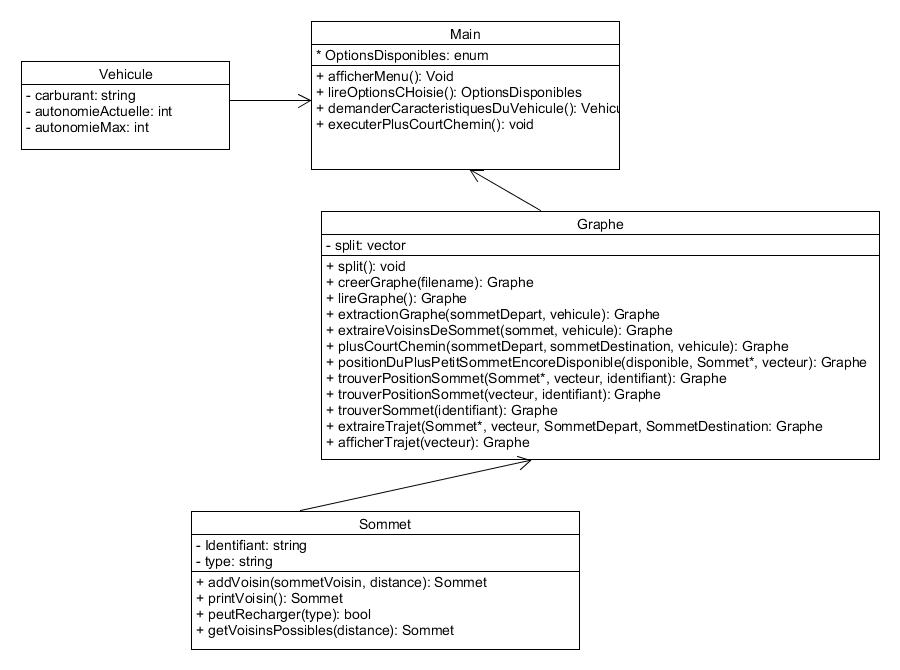


Figure 5 Diagramme de classes

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)