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 zone4.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. De plus, si aucun véhicule n'est disponible dans la zone, on fera venir un véhicule libre dans une autre zone. Si par malheur, en grosse période d'achalandage, aucun véhicule n'est disponible

dans toute la ville, alors la demande du client sera refusée. Il est à noter qu'il aurait été possible de faire en sorte que le client soit simplement relégué au prochain groupe d'utilisateurs, mais que ce n'est pas le comportement choisi. 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 :

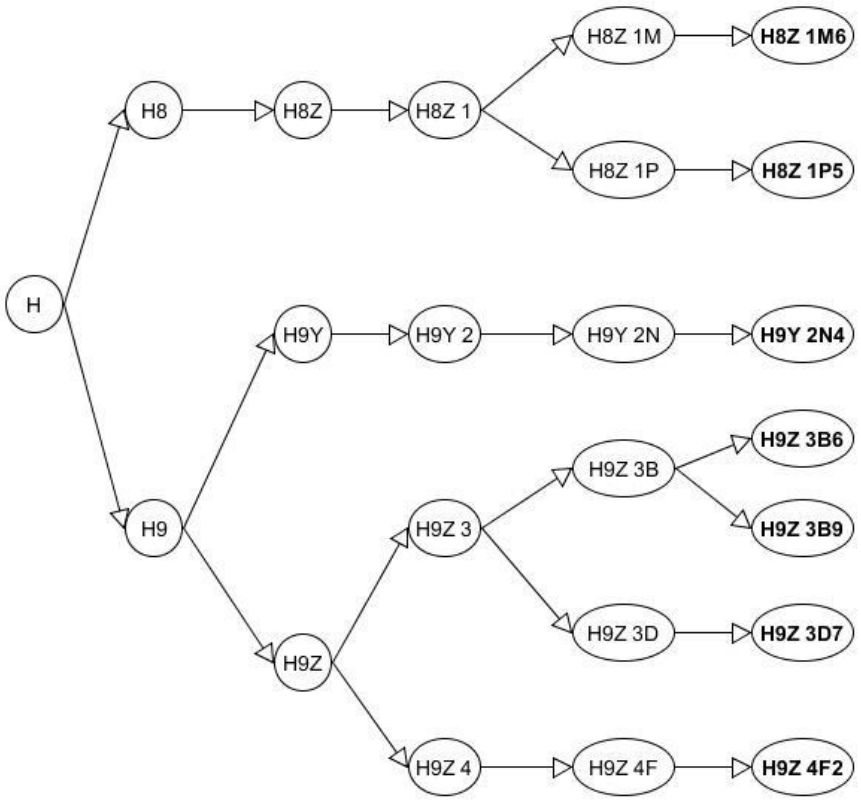

*Figure 1 - Automate qui représente les codes postaux suivants : {H8Z 1M6, H8Z 1P5, H9Y 2N4, H9Z 3B6, H9Z 3B9, H9Z 3D7, H9Z 4F2}. Veuillez prendre note que les états en **gras** sont les états finaux*

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. On s'assure donc de déplacer un véhicule d'une autre zone. Par contre, et afin d'éviter ces situations, nous ferons appel à un algorithme d'équilibrage.

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 des vélos disponibles à 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 (à la fin des déplacements d'un groupe), un véhicule provenant d'une zone où il y a plusieurs véhicules disponibles est acheminé vers cette zone en déficit.

Il est important de noter que nous ne fixons pas de restrictions par rapport au temps. En effet, lorsque nous exécutons la simulation, nous prenons pour acquis que tous les clients font leurs requêtes de véhicules en même temps. Pour faciliter la gestion de l'exécution de la simulation, nous séparons les clients en différents groupes. Chaque groupe fait donc ses requêtes en même temps, ce qui crée forcément un déséquilibre au niveau de la flotte de véhicules. Il faut donc équilibrer la flotte après que chaque groupe ait fait ses requêtes. Nous avons aussi la possibilité de considérer chaque utilisateur de façon indépendante en les mettant tous dans un groupe différent.

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 avions utilisé le langage C++ lors du premier travail pratique. Toutefois, pour celui-ci, nous avons apporté une différence majeure. Nous avons codé le système de simulation, ainsi que tout ce qui l'accompagne en Java, pour plusieurs raisons. Tout d'abord, en Java, nous n'avons pas de pointeurs à gérer, ce qui simplifie beaucoup la tâche, tout en rendant la programmation plus agréable. Nous avions aussi envie d'expérimenter un peu plus en Java, puisque tous nos projets des deux premières sessions étaient en C++.

La première fonction que nous devons avoir est la fonction « creerLexiques() », qui permet de lire les fichiers qui nous sont fournis et qui contiennent les codes postaux de chaque zones. Cette fonction prend en paramètre un chemin absolu vers le dossier contenant les fichiers texte. À partir de ce chemin, il est en mesure d'extraire tous les fichiers qu'il contient et de les parcourir afin de les charger dans la carte du programme.

Nous devons aussi avoir une fonction « equilibrerFlotte() ». C'est cette fonction qui sera exécutée après qu'un groupe de clients ait utilisé des véhicules. Comme son nom l'indique, cette fonction fera équilibrer le nombre de véhicules dans chaque zone de la ville. L'idée générale est d'avoir le même nombre de véhicules dans chaque zone. Par contre, certains cas limites font en sorte qu'il n'est pas toujours possible d'avoir un nombre exact de véhicules qui se répartissent bien. Ainsi, l'alternative est de considérer qu'une répartition est légitime lorsque la différence entre

le nombre minimum et le nombre maximum de véhicules par zone est de 1 ou de 0. Afin d'implémenter cet algorithme, nous commençons par parcourir toutes le zones afin de déterminer le nombre de véhicules qu'il contient. Puis, nous calculons combien de véhicules il faut ajouter ou retrancher dans chacune des zones afin d'atteindre un équilibre acceptable, et finalement nous effectuons les transferts.

Il faudra aussi une fonction « lancerSimulation() » qui exécutera la simulation. L'usager doit entrer toutes les informations nécessaires au préalable. En effet, il doit sélectionner les options (a) et (b) avant de pouvoir lancer une simulation. Pour ce faire, il devra passer par les différents menus. Ces menus permettront tout d'abord de créer les zones dans la ville. Cette option (a) appellera donc la fonction qui fait la lecture des fichiers et qui lit les codes postaux. Il sera ensuite nécessaire d'entrer (b) les clients, ainsi que les véhicules. Dans ce menu, nous demandons tout d'abord à l'utilisateur quel est son point de départ, sa destination, ainsi que son numéro de groupe. Les voitures ont ensuite aussi une zone de départ. Un détail important ici, est de s'assurer d'exécuter la première partie pour créer les zones avant la seconde, car ces zones sont nécessaires pour valider que l'utilisateur entre des données valides. Une fois toutes ces informations complétées, il est possible d'exécuter la simulation. Ces options seront donc représentées dans un menu semblable à celui ci-contre :

```
Sélectionnez une option parmi les suivantes :
(a) Créer les zones.
(b) Entrer les clients et les véhicules.
(c) Démarrer la simulation.
(d) Quitter
Votre choix :
```

Figure 2 - Menu principal de sélection d'options de notre application

Automates et machines à états :

Comme nous l'avons montré ci-haut, nous utilisons des automates et machines à états pour déterminer la position des clients. Ces automates sont utilisés principalement lors de la lecture des 4 fichiers, lorsque l'option (a) est appelée dans le menu. À ce point, nous lisons les fichiers et créons l'automate à partir de ces derniers, de la même façon qui a été expliquée plus tôt. Une fois les automates créés, nous les utilisons pour repérer les données entrées par les utilisateurs et les voitures. Ce sont les données contenues dans ces automates qui sont donc utilisées.

Diagrammes de classes :

Pour faciliter notre compréhension du problème, nous avons tout d'abord eu recours à un diagramme de classe. Ce diagramme consiste en une structure que nous avons eu à élaborer et conserver tout au long du développement de l'application. Avoir un tel diagramme nous force à créer du code de qualité en utilisant les bonnes fonctions pour faire les bonnes choses. On s'assure ainsi de la qualité du travail tout au long du processus. Cette structure nous permet d'implémenter très facilement des tests unitaires, ce qui contribue grandement à la solidité de notre système. Une autre utilité à ce diagramme est de pouvoir représenter de façon visuelle les fonctions de l'application, ainsi que tout ce qu'elle sera en mesure de faire. Une telle séparation des éléments d'un logiciel est aussi une bonne pratique de programmation. Avoir une telle structure permet aussi une meilleure encapsulation, afin de mieux contrôler quelles fonctions ont accès à quelles données. On limite donc l'accès aux données.

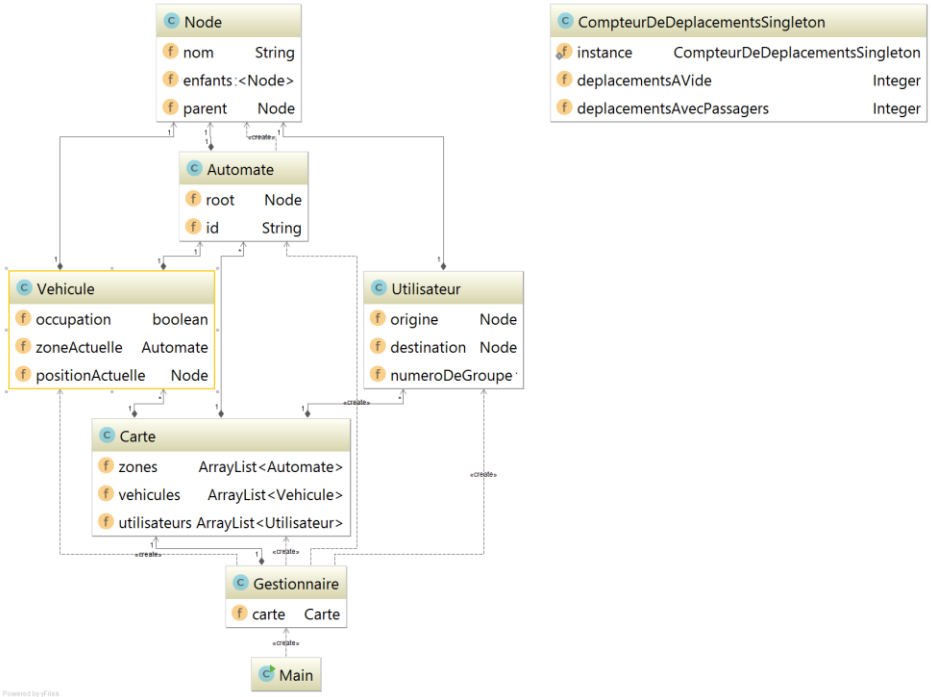

Figure 3 - Diagramme de classes de notre solution.
NB. Il nous était impossible de générer le diagramme de classes contenant toutes les méthodes, car notre solution comporte un nombre trop grand de méthodes. Il s'agit là de l'application des bonnes pratiques de développement logiciel.

Difficultés rencontrées et solutions apportées :

L'une des premières difficultés rencontrées a été de faire la lecture des fichiers contenus dans le dossier. En effet, il n'existe pas de fonction simple qui permet de découvrir les fichiers contenus dans un dossier. Par contre, avec un peu d'utilisation d'Internet, nous avons été en mesure de trouver un algorithme qui permettait de faire exactement ce dont nous avions besoin. La référence se trouve dans le code à l'endroit concerné.

Suite à cette difficulté, nous avons pu commencer à implémenter l'algorithme de répartition des véhicules. Cela a immédiatement posé d'autres problèmes, car dès que nous croyions avoir un bon algorithme, nous trouvions des cas limites qui ne fonctionnaient pas. Afin de régler ce problème, nous avons passé en revue tout notre algorithme en ayant en tête certains scénarios limites. Puis, afin de consolider le tout et dans le but de rendre les changements ultérieurs plus simples à valider, nous avons développé une série de tests unitaires. Ces tests permettent de vérifier plusieurs cas de répartition de véhicules et vient donc consolider notre algorithme en s'assurant que les véhicules sont toujours répartis selon les règles définies plus haut.

Finalement, la dernière difficulté a été d'effectuer les déplacements des utilisateurs de la façon la plus efficace possible. En effet, nous avons constaté que certaines situations ne sont pas nécessairement les plus optimales. Par exemple, si nous avons le cas suivant :

	Zone 1	**Zone 2**	**Zone 3**	**Zone 4**
Véhicules	1	0	1	1
Utilisateurs (départ)	2	0	1	0

Notre algorithme verrait donc les 2 utilisateurs dans la zone 1 et devrait dépêcher un véhicule supplémentaire afin de ne pas perdre un client. Selon l'implémentation actuelle, on ne fait aucune vérification quant à savoir quel véhicule il serait plus efficace de prendre. Aléatoirement, il choisirait peut-être celui de la zone 3, mais on sait qu'on aura ensuite besoin d'un véhicule dans la zone 3. On ferait donc mieux de prendre le véhicule libre de la zone 4 afin de déplacer le client de la zone 1. Il s'agit donc d'une difficulté que nous n'avons pas pu corriger.

Conclusion :

Pour conclure, notre système est fonctionnel et permet d'acheminer un véhicule à chaque groupe d'utilisateur qui en demande un. Une fois les véhicules attribués, il redistribue les véhicules inutilisés, équitablement entre toutes les zones de la ville. Nous avons aussi été en mesure de créer les fonctions « creerLexiques() », qui créait l'automate à partir des fichiers qui contiennent les codes postaux de chaque zones. La fonction « equilibrerFlotte() » uniformise le nombre de véhicules entre les zones. La fonction « lancerSimulation() », pour sa part lance la simulation, une fois que l'utilisateur a créé les zones et rempli les informations concernant le client et le véhicule. Les zones doivent être créer en premier lieu, suivi de la saisie des informations du client.

Ce système permettra donc de simuler un système d'autopartage pour la ville de montréal. Il permettra d'équilibrer le nombre de véhicules entre les zones de la ville, pour s'assurer que tous les utilisateurs puissent avoir accès à une voiture quand ils en ont besoin.

Nous avons aussi discuté des difficultés rencontrées et des améliorations qui pourraient être apportées au système, par exemple, si une zone manque de véhicules, compenser par ceux de zones voisines, plutôt que de zones éloignées qui contiendraient plus de véhicules.

Il serait intéressant d'évaluer comment ce système se comporterait dans la réalité dans le cadre d'une entreprise de taxi, telle que Téo Taxi, pour rester dans le même contexte. Il pourrait aussi être intéressant de comparer avec des systèmes de taxi existants et déterminer les ressemblances et différences.