LOG2810: Structures discrètes
TP1: Graphes

Felix-Antoine Bourbonnais – 1855719
Louis Popovic – 1844807
Noboru Yoshida – 1860254

Polytechnique Montréal
2 novembre 2018

Introduction

Le travail consiste à développer un algorithme qui permet de déterminer les chemins que devraient prendre des véhicules médicaux autonomes. Le logiciel conçu calcule et retourne, en tenant compte de plusieurs facteurs, le chemin le plus optimal que les véhicules médicaux autonomes devraient parcourir. Ces derniers se déplacent entre différents centres locaux de services communautaires (CLSC) de Montréal.

Le logiciel prend un fichier texte comme entrée. Ce fichier contient les informations nécessaires concernant les différents CLSC. Parmi les informations qu'on y trouve, figurent l'emplacement des bornes de recharge ainsi que la liste des CLSC voisins de chaque CLSC avec la distance qui les sépare les uns des autres.

De multiples contraintes doivent être considérées lorsque le logiciel tente de résoudre le problème. En effet, il est important de considérer, entre-autres, le type de véhicules empruntés (Li-ion ou Ni-MH), l'énergie utilisée pour chaque type de transport, le niveau de risque des patients, la présence de bornes de recharge et bien d'autres facteurs.

Le logiciel doit contenir une interface où les utilisateurs peuvent entrer les données. Il doit être en mesure de mettre à jour le fichier contenant l'information sur les CLSC, de trouver le chemin le plus court sécuritaire pour le véhicule entre deux CLSC, d'extraire un sous graphe d'un CLSC présent dans le graphe et finalement, de quitter le programme. Bref, nous devrons implémenter un ensemble de méthodes permettant au logiciel d'exécuter les différents objectifs spécifiés précédemment.

À travers ce travail, nous pensons appliquer la théorie qui a été enseignée durant les cours pour l'implémenter dans un cas concret. Nous avons aussi décidé, en équipe, de réaliser le travail en Python, un langage que nous n'avions jamais touché jusqu'à aujourd'hui. Nous avons donc comme objectif additionnel, l'apprentissage de la base de Python.

Présentation des travaux

Fichier menu.py

Pour contrôler l'interaction entre l'utilisateur et les fonctions incluses dans le logiciel, nous avons construit un menu. Ce menu permet à l'utilisateur de choisir entre les 4 options voulues. Il peut mettre à jour la carte (le graphe), déterminer le plus court chemin sécuritaire, extraire un sous-graphe qui contient le chemin qui parcourt la plus grande distance à partir d'un certain nœud et quitter. Suite à la sélection, le logiciel renvoie l'utilisateur à l'option désirée et demande alors à l'utilisateur la valeur des paramètres nécessaires à l'aide de questions. Les valeurs des réponses sont sauvegardées dans des variables. Les variables sont ensuite envoyées comme paramètres à la fonction reliée à l'option choisie par l'utilisateur qui se situe dans le fichier graphesFunctions.py. Lorsque la routine est finie, on revient à l'affichage du menu. Bien sûr, pour chaque entrée de l'utilisateur, un contrôle d'erreur est effectué. Chaque entrée de l'utilisateur est vérifiée et si elle n'est pas compatible, un message d'erreur est affiché et l'utilisateur doit alors entrer une valeur à nouveau.

Fichier dictionnaires.py

Ce fichier contient l'ensemble des constantes, classes et dictionnaires utilisés dans le programme. Les classes Vehicule et Risque sont des énumérations des véhicules et niveaux de risque possibles, respectivement. Elles sont appelées par les dictionnaires dictVehicule et dictRisque pour aider à la présentation des données. Les constantes permettent de représenter certaines valeurs qui sont utilisées dans graphesFonctions. Elles affichent avec cohérence et clarté les données et les calculs, ce qui évite la présence de "nombres magiques" dans le code. Les dictionnaires sont utilisés dans grapheFonctions et dans le menu. Le dictTauxDeDecharge permet la représentation claire des taux de décharge de chaque véhicule. Les dictionnaires dictVehicule et dictRisque relient le choix de l'utilisateur dans le menu et la valeur de retour voulue. Bref, dictionnaire.py agit comme interface pour rendre le reste du code plus clair et simple à comprendre.

Fichier graphesFunctions.py

On retrouve dans le fichier graphesFonction.py toutes les fonctions qui servent à calculer les chemins les plus courts et les plus longs. On retrouve aussi des fonctions très courtes qui servent uniquement à vérifier si un fichier existe (fichierExiste), si le graphe a été créé (grapheExiste) et si un nœud est présent dans le graphe (noeudExiste).

- creerGraphe():

  La fonction creerGraphe prend en paramètre un nom de fichier. Comme on sait que le fichier.txt est composé de deux sections (Bornes et Graphe) séparés par une ligne vide, on crée deux listes, grâce à la méthode .split(). Par la suite, on parcourt la première liste, à partir de laquelle on crée un dictionnaire ayant comme clé le nœud, et comme valeur un bool qui précise si une borne de recharge est présente. Par la suite, avec la deuxième liste, on crée notre graphe à l'aide d'un dictionnaire ayant la structure suivante :

{ nœud1 : dict{ voisin1_1 : temps1_1, voisin1_2 : temps1_2, … },
  noeud2 : […]
}

Nous avons utilisé un dictionnaire pour représenter notre graphe et nos bornes de recharge puisque toutes les informations sont situées au même endroit, et sont toutes accessibles en O(1), ce qui rend notre programme très efficace.

- lireGraphe() :
  La fonction lireGraphe() est très simple. On parcourt tous les items de notre dictionnaire, et on les imprime à l'écran. En effet, python est capable d'imprimer des structures de données sans qu'on ait à la décomposer nous-même en string.
- algoDijkstra() :
  La fonction algoDijkstra() prend en paramètre un point d'origine et une destination. Elle retourne un tuple qui contient le chemin à parcourir, et le temps total. Nous nous sommes inspirés de l'algorithme vu en classe pour implémenter la fonction. En effet, on crée un dictionnaire vide, dans lequel on place comme clé initiale notre point d'origine, et comme valeur un tuple qui contient le nœud précèdent (qui est None pour l'origine), et le temps à partir de l'origine (0 pour l'origine). On crée aussi un set() des nœuds visités (qui est vide au début). Par la suite, tant et aussi longtemps que nous ne sommes pas arrivés à notre destination, on ajoute le nœud courant aux nœuds visités, et on trouve tous les voisins du nœud courant. Pour tous ses voisins, on vérifie s'il a déjà été visité. S'il n'a pas encore été visité, on ajoute au dictionnaire le voisin (comme clé), et le tuple contenant le nœud courant, et le temps à partir de l'origine. Si le nœud a déjà été visité, on compare la valeur du temps présent dans le tuple et notre nouveau temps, et si nécessaire, on met à jour ce tuple. Après avoir ajouté, ou mis à jour, tous les voisins, on se rend au nœud où le temps à partir de l'origine est minimal, et on retourne au début de notre boucle while(). Finalement, on parcourt notre dictionnaire dans le sens inverse en partant de la destination, et on remonte jusqu'à l'origine en ajoutant les nœuds un par un. Juste avant de retourner le chemin, on inverse l'ordre du chemin, pour que l'origine soit au début.
- creerListeChemin() :
  La fonction creerListeChemin() prend en paramètre une liste de tuples (nœud, temps à partir de l'origine), et retourne une liste contenant uniquement les nœuds par lesquels on passe.
- trouverPlusCourtCheminAvecBorneRecharge() :
  La fonction trouverPlusCourtCheminAvecBorneRecharge() prend en paramètre le temps que prend le véhicule pour perdre 80% de sa batterie, le nœud d'origine, la destination et la vitesse de déchargement (%/min). Cette fonction utilise la fonction algoDijkstra() pour trouver tous les chemins optimaux entre le point d'origine et chaque borne de recharge. Ensuite, pour chaque borne de recharge, on trouve le chemin optimal jusqu'à la destination. Finalement, on vérifie, à l'aide du paramètre temps de déchargement, si le véhicule peut effectuer le parcours origine -> borne et borne -> destination sans perdre plus de 80% de sa batterie. On retourne, dans un dictionnaire, le chemin le plus court entre tous ceux calculés et faisables, le temps total et le niveau de batterie final.

- plusCourtChemin() :
  La fonction plusCourtChemin() est la fonction appelée par le menu lorsque l'utilisateur sélectionne l'option c. L'utilisateur donne le niveau de risque, l'origine et la destination. Pour trouver le plus court chemin entre les deux points, on utilise d'abord la fonction algoDijkstra() de l'origine jusqu'à la destination. Par la suite, on vérifie, en fonction du type de transport (risque faible, moyen ou haut), si un véhicule Ni-MH peut s'y rendre avant de perdre 80% de sa batterie. Si c'est possible, on retourne un dictionnaire contenant le chemin, le temps total, le niveau de batterie final et le type de véhicule choisi. Par contre, si un véhicule de type Ni-MH ne peut pas se rendre à destination sans perdre plus de 80% de sa batterie, on vérifie si une borne de recharge est présente dans le chemin, et s'il est accessible depuis le point de départ. Si oui, on recalcule le temps total et le niveau de batterie. Par contre, s'il n'y a pas de borne sur le chemin, on fait appel à la fonction trouverPlusCourtCheminAvecBorneRecharge(), qui retournera un dictionnaire s'il existe un chemin, et None sinon. Finalement, si aucun chemin n'a encore été trouvé, on refait appel à la fonction plusCourtChemin(), avec les mêmes paramètres, sauf que cette fois ci, on change le type de véhicule (pour Li-ion), qui par défaut était Ni-MH. Encore une fois, si aucun chemin n'est possible, on retournera None. Grâce aux valeurs de retour possible, la fonction menu() est capable de facilement afficher toutes les informations, et de savoir si aucun chemin n'est possible.
- extraireSousGraphe() :
  Cette fonction prend en paramètre le risque de transport, l'origine et le type de véhicule utilisé et retourne les nœuds qui constituent le plus long chemin possible qui peut être parcourue de l'origine. Elle calcule la durée pour laquelle le véhicule peut se déplacer et appelle la fonction trouverPlusLongChemin() avec la durée calculée.
- trouverPlusLongChemin() :
  Cette fonction récursive prend en paramètre l'origine, la durée pour laquelle un véhicule peut se déplacer, une liste de nœuds visités, une variable qui contient le temps total parcouru et la liste qui décrit le chemin parcouru jusqu'au moment où la fonction est appelée. La fonction procède à un depth-first-search des chemins possibles en vérifiant que le temps total ne dépasse pas la durée entrée en paramètre. Elle retourne le plus long chemin qui peut être parcouru de l'origine ainsi que le temps qu'il faut pour le parcourir.

Difficultés rencontrées

Une des plus grandes difficultés rencontrées a été imposée par nous-mêmes. Il s'agit d'écrire le programme en Python, qui n'est pas un langage de programmation connu par les membres de l'équipe. Nous avons dû apprendre la base du langage et sa syntaxe au fur et à mesure que nous écrivions le code du programme. Nous avons donc consacré beaucoup de temps à faire de la recherche sur internet afin de régler ces problèmes. Une situation particulière est survenue lors de l'étape de l'écriture du code de l'interface qui affiche le menu. Nous voulions initialement implémenter une structure « switch-case » afin de faciliter la réception des valeurs entrées par l'utilisateur. Cependant, on a assez rapidement compris qu'une telle structure n'était pas très efficace d'usage en Python. Plusieurs problèmes de ce type sont survenus tout au long du travail, mais à chaque fois, nous pouvions trouver la réponse aux ambiguïtés d'ordre technique sur le web.

De plus, puisque le travail a été effectué en équipe de 3, apporter une uniformité au code a été un défi pour nous. Plusieurs variables et fonctions étant réutilisées dans des parties différentes, il fallait que nous nous entendions sur une manière d'appeler chaque élément afin de garantir une certaine qualité de code. La lisibilité du code est surtout affectée par ce point. Nous nous sommes rendus à la conclusion qu'on devrait créer un élément qui se servirait de « librairie ». Le fichier dictionnaires.py tient ce rôle et contient la valeur de certaines variables qui sont implémentées dans un dictionnaire. Aussi, afin de garder des fichiers de code structurés, nous avions ajouté des fonctions dans graphesFonctions.py qui ont accès aux nœuds existants et au graphe. Ces dernières, qui sont utilisées pour tester les entrées de l'utilisateur, renvoient si les nœuds ou le graphe existent ou non. En plaçant ces fonctions sous le fichier graphesFonctions.py, on peut s'assurer qu'uniquement les fonctions directement liées à l'interface se trouvent dans le fichier menu.py.

Une autre grande difficulté que nous avons éprouvée concerne une caractéristique propre à Python. En effet, dans la fonction récursive trouverPlusLongChemin() du fichier graphesFonctions.py, on avait besoin de créer une copie de la liste chemins[] afin de toujours garder la liste originale intacte à travers les multiples itérations. Cependant, nous n'arrivions pas à réaliser ceci malgré la création d'une nouvelle liste liste_copie[]. Après de longues heures de débogage et de recherche sur internet, nous avions enfin pu trouver la solution qui était d'utiliser la méthode .copy() qui nous permet de créer un nouvel objet et de passer les valeurs par copie, plutôt que de le faire par référence. Ce concept diffère énormément de celui de Java, un langage qui nous est le plus familier en ce moment.

Dans la même fonction, c'est-à-dire celle de trouverPlusLongChemin(), on a également éprouvé de la difficulté à toujours mettre à jour le contenu de la liste voisinsAVisiter[]. Cette liste ne contenait pas les bonnes valeurs lors de chaque appel de la fonction récursive, ce qui produisait un décalage de valeurs que contenait la liste. Une erreur s'affichait à la fin à cause de

ceci. On a passé beaucoup de temps pour régler ce problème et au final, on s'est rendu compte que, malgré le fait qu'habituellement il est mieux de mettre des éléments de calcul dans des variables afin de réduire le nombre de calcul qui sera exécuté par le programme au total (cela mène à une diminution de la complexité Big-O), dans notre cas, on devait absolument recalculer les valeurs à chaque itération de la fonction récursive.

Conclusion

Ce laboratoire nous a permis d'acquérir beaucoup de compétences. D'abord, il s'agit du premier travail d'envergure réalisé en utilisant git afin de gérer les versions de nos fichiers pour l'ensemble des coéquipiers. Nous avons donc appris grandement au niveau de l'écriture d'un code en équipe et de la séparation des tâches. L'utilisation de git dans ce travail s'est prouvé essentiel et nous a permis d'apprendre à quel point il est important de bien gérer le répertoire git et d'écrire des commentaires significatifs lors des "commits".

Nous avons appris énormément quant à la programmation avec python également. En effet, le passage par référence dans python s'avère différent que dans d'autres langage comme Java. Ceci nous a causé beaucoup de problèmes lors de la programmation. Nous avons aussi remarqué plusieurs avantages et inconvénients dans l'utilisation avec python. Par exemple, python n'a pas besoin de déclaration de type, ce qui est très intéressant. Ceci rend le code plus facile à écrire, mais peut toutefois porter à confusion lors de la relecture. En effet, il peut être difficile de trouver le type d'une variable si le code devient relativement complexe.

Enfin, nous avons acquis de nouvelles connaissances par rapport aux fonctions récursives. Effectivement, avant ce travail, nous n'avions pas vraiment eu la chance d'implémenter nous-mêmes des fonctions récursives. Maintenant que nous avions fini le travail, nous réalisons l'utilité de ces fonctions potentiellement compliquées, considérant qu'elles sont essentielles aux fonctions principales de notre programme.

Pour le prochain laboratoire, on s'attend à être plus efficace au niveau de l'écriture du code et de la séparation des tâches. Ceci va nous permettre de compléter l'écriture du programme avec moins de temps, vu que nous avons passé plus de 100 heures sur le projet. On pourra ainsi passer plus de temps pour tester notre code pour s'assurer que tout fonctionne comme prévu. On s'attend finalement à avoir moins de problèmes avec git, considérant l'expérience acquise durant ce travail pratique.