Rapport Projet Algorithmique 2

Karbach Abdelmouhaïmin | Matricule: 593038

16 mai 2025

Contents

1 Introduction

Dans le cadre du Projet en algorithmique de deux année du bachelier de science informatique, il nous a été demander de créer un programme permettant de calculer le plus court chemin entre deux arrêts de transport en commun en Belgique et ce à l'aide des données fournis sous la norme GTFS, qui décrivent l'ensemble des horaires et arrêts du réseau de transport belge.

Pour ce faire, nous avons opté pour une modélisation de réseau sous la forme d'un graphe orienté pondéré, dans lequel chaque sommet représente un arrêt et chaque arrêt vaut le temps de voyage entre les deux arrêts. Pour parcourir le graphe, nous avons opté pour l'utilisation de l'algorithme de Dijkstra(Dijkstra, 1959).

Cependant, notre programme présente un défaut majeure, l'implémentation actuelle ne prends pas en compte le temps de changement de correspondance au sein d'un même arrêt. Plusieurs approches on été essayé sans succès.

Ce rapport présente dans un premier temps la modélisation choisie et les structures de données utilisées, puis l'algorithme de recherche de chemin mis en place, et enfin nous suivrons l'exécution d'une requête. Nous terminons par une analyse des résultats obtenus, une discussion sur les limites de l'approche actuelle, et des pistes d'amélioration futures, notamment concernant la gestion fine des correspondances.

2 Structure de données

2.1 Graphe

Pour la représentation du réseau, nous avons opté pour la structure d'un graphe orienté et pondéré, ce choix parait assez évident et intuitif, chaque sommet représente un arrêt, regroupant tout les stoptimes de cet arrêt, et chaque arête est pondéré à l'aide du temps que sépare deux stoptimes consécutifs. Le choix d'utiliser les arrêts comme sommet et non pas les stoptimes est justifié par le fait qu'avec les stoptime, le graphe aurait été beaucoup trop dense et donc la construction du aurait pris trop de temps et trop de place sur le heap. La taille du heap qui d'ailleurs été sujet à beaucoup de discussion, vu la taille des données manipulées. Le graphe utilisé pour construire le réseau, est le EdgeWeightedGraph proposé par la librairie algs4.jar(Sedgewick & Wayne, 2011). Etant donné l'utilisation de ce graphe, pouvant stocké dans ses sommets que des entiers, il a fallut répertorier tout les arrêts en fonction de leur entier associé sur le graphe. Nous verrons dans la section associé les problèmes lier à ce choix d'implémentation.

2.2 Implémentation en Java

Dans un premier temps, un objet Apprunner est créé, cette objet crée et initialise tout les autres objet dont nous auront besoin lors de l'execution du programme, tel que le GraphBuilder et le Parser. Le GraphBuilder est l'objet qui construira le graphe à l'aide du EdgeWeightedGraph et des données fournis. Le Parser permet d'extraire les données GTFS fournis, il utilise simplement OpenCSV(Developers, 2023) un outils open-source permettant la manipulation de fichier au format CSV. Les données extrait des fichier csv, sont stocké dans les objet StopTime et Stop, qui sont eux stockés dans les HashMap, stopTimesMap et stopsMap, pour les stoptimes, on entre le tripId et on récupére tout les stopTimes de ce tripId, facilitant la construction du graphe par la suite. Et pour les stops, on entre le nom de l'arrêt pour accéder à la liste d'arrêt de ce nom. Les stopTimes sont chargé avec une contrainte, celle d'être dans une plage horaire "logique" en fonction de l'entrée utilisateur, les heures des stopTimes ne doivent pas être avant l'heure de départ entré par l'utilisateur, ni dépassé 9h après l'entré utilisateur, car un trajet optimal en Belgique ne dépassera pas 9h de trajet.

3 Algorithmes

3.1 Algorithme de plus court chemin

L'algorithme utilisé pour parcourir le graphe est l'agorithme de Dijkstra(Dijkstra, 1959), ce choix est justifié par le fait qu'il soit adapté pour les les graphes pondérés orienté, qu'il garantit l'optimalité du chemins le plus court, parcequ'il permet une flexibilité au niveau des variantes du problème, permettant changer la manière d'on les données sont géré au court de l'implémentation du programme, il est facile de passer d'une version qui stocke des stopTimes comme sommets à une version qui stocke des stops comme sommets, ce qui permet de faire des modifications plus facilement. Son utilisation et aussi justifié par le fait que la librairie algs4(Sedgewick & Wayne, 2011), en inclue un optimisé donc pour le EdgeWeightedGraph disponible aussi dans cette même librairie. Le choix peut aussi être justifié par la complexité raisonnable, qui est de $O((E+V)logV)$, E pour le nombres d'arêtes dans le graphe et V pour le nombre de nœuds dans le graphe.

3.2 Algorithme d'ajout de marche entre les arrêts

Pour pouvoir prendre en compte la marche entre les différents arrêts, nous avons opté pour la structure de donnée KDtree(Bentley, 1975). Ce choix est justifié par sa capacité de recherche spatiale rapide permettant de trouver tout les arrêts dans un rayon donné autour d'un point, sa complexité de construction est de $O(n\ log(n))$, n étant le nombre d'arrêts, et sa complexité de recherche est de $O(log(n))$ en moyenne. Son utilisation est surtout justifié par le fait qu'il soit adapté pour l'utilisation avec des données géographique, comme celles fournis dans le fichier stops.csv. Après avoir trouvé deux arrêts assez proches l'un de l'autre, il faut déterminé la distance entre ces deux arrêts, pour se faire, nous avons utilisé Haversine(Sinnot, 1984).

4 Exécution d'une requête

4.1 Standardisation de l'entrée utilisateur

Dans un premier temps l'entrée utilisateur est standardisée pour permettre à l'utilisateur de pouvoir passer outre les majuscules.

4.2 Parsing

Après la standardisation de l'entrée utilisateur, on passe directement au parsing des fichier .csv, effectué par l'objet ParcerCSV, on parse les stopTimes et les stops respectivement, à l'aide des methode ***parceStoptimes()*** et ***parceStops()*** du Parcer, comme ceci nous obtenons deux HashMap, stopTimesMap et stopsMap.

4.3 Constuction du Graphe

Après le parsing, le graphe est directement construit à l'aide de la methode ***buildGraph()*** de l'objet GraphBuilder, lors de la l'appel à ***buildGraph()*** le graphe est initialisé a l'aide de la taille de stopMap, étant donné qu'un sommet est un arrêt. Ensuite le graphe est implémenté de façon "réel", c'est à dire que l'on parcours tout les stopTimes et on crée un chemin à l'aide de l'ordre de séquence de ces stopTimes. Une bonne façon de le visualisé est de simplement de regarder une carte de société de transport comme celle de la STIB (Figure 1) et pour chaque arrêts nous avons une liste de stopTimes correspondant à l'arrêt (Figure 2). Étant donné que les sommet du graphe ne peuvent être assigné qu'à des entiers, un dictionnaire, nom d'arrêts vers entier, est nécessaire pour voir si l'arrêt a déjà été placé sur le graphe et un dictionnaire, entier vers nom d'arrêts, pour pouvoir retracer le chemin par la suite. Les arêtes correspondes simplement à la différence entre le temps de départ d'un stopTime et celui du prochain dans la séquence du tripId correspondant dans le prochain arrêt. Ensuite les index des arrêts de départ et d'arrivé son retenu lors de la création

du graphe. Et pour finir les arêtes de marche sont ajouté entre les arrêts qui sont à une distance maximal de 500m.

Figure 1: Visualisation du graphe (STIB)

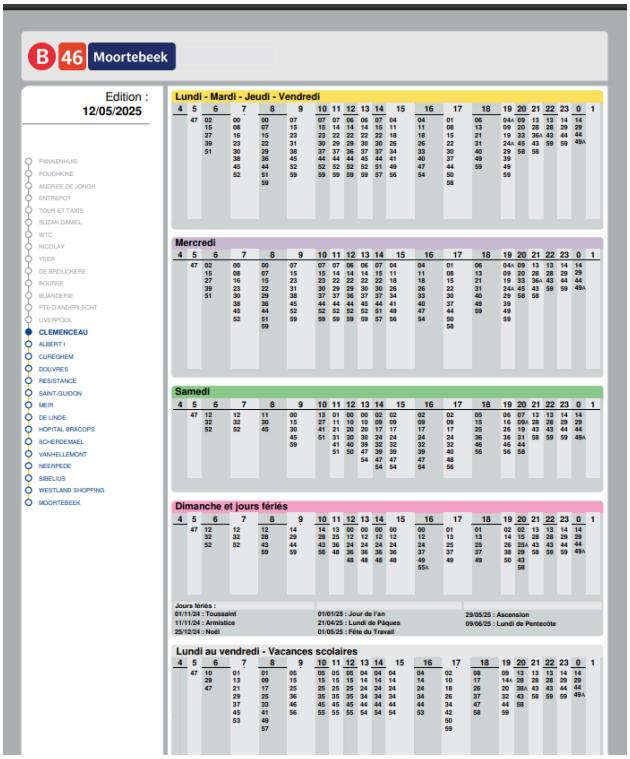

Figure 2: Visualisation liste de StopTimes (bus 46 Clemenceau)

4.4 Recherche du plus court chemin

Après la construction du graphe, il faut trouver le chemin le plus court, et pour se faire, nous avons utiliser Dijkstra(Dijkstra, 1959) implémenté dans la librairie algs4(Sedgewick & Wayne, 2011).

5 Résultats

Comme dit plus haut dans ce document, le programme ne prends pas en compte les temps entres les changements de correspondance, mais il est tout de même fonctionnel pour les trajets où les arrêts sont bien desservis, donc ou les temps de changement de correspondance est moindre. Avec une entrée Clemenceau Delta 18:30:00, nous obtenons un résultat optimal (Figure 3).

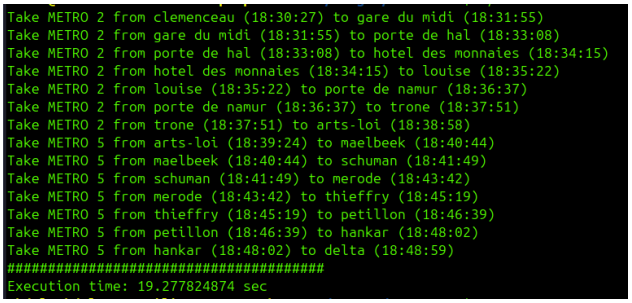

Figure 3: Résultat Clemenceau Delta 18:30:00

Tandis qu'avec une entrée "Alveringem Nieuwe Herberg" Aubange 10:30:00, nous obtenons un résultat ou les temps de changements de correspondance ne sont pas pris en compte et où ne devons marcher à la fin car les arrêts ne sont pas chargés car les stopTimes concernés dépace l'heure de départ + 9h (Figure 4).

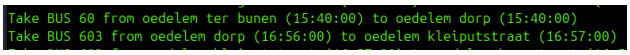

Figure 4: Résultat "Alveringem Nieuwe Herberg" Aubange 10:30:00

6 Améliorations/corrections possibles

Nous allons voir les différentes améliorations/corrections possibles pour que le programme puisse prendre en compte les changements de correspondance, ou encore pour optimiser le code.

6.1 Modification de Dijkstra

Une des façons de prendre en compte les changements de correspondance est de changer l'algorithme de Dijkstra utilisé, pour qu'à chaque fois que l'on veuille parcourir une arête, on vérifie si le tripId utilisé lors de la dernière arête est le même que celui qui va être utilisé pour traverser la prochaine. Si c'est le cas, aucune mesure n'est prise, mais si ce n'est pas le cas, le temps d'attente entre le stopTime étant le plus proche du dernier stopTime et celui est ajouté au poids de l'arête.

6.2 Parsing des fichier CSV

La librairie utilisé pour le parsing des fichier .csv est OpenCSV(Developers, 2023) or il existe bien d'autre librairie plus performante. En optimisant le parsing nous pouvons gagné énormément de temps de calcule,

étant donné que le parsing est la partie prenant le plus de temps lors de l'exécution (Figure 5). On peut voir grâce à quelques benchmarks que OpenCSV(Developers, 2023) est 173% plus lent que le parser le plus rapide(Univocity, 2023).

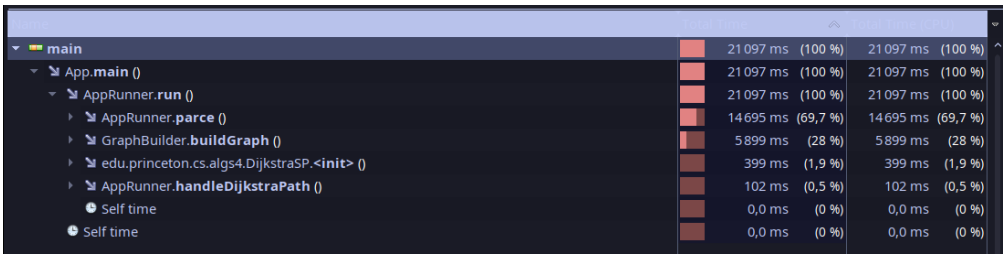

Figure 5: Temps d'exécution de l'entrée Clemenceau Delta 10:30:00 décomposé à l'aide du profiler VisualVM

7 Bibliographie

References

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, *18*(9), 509–517. https://doi.org/10.1145/361002.361007

Developers, O. (2023). Opencsv: A simple csv parser for java. http://opencsv.sourceforge.net/

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*, 269–271. https://doi.org/10.1007/BF01386390

Sedgewick, R., & Wayne, K. (2011). Algorithms, 4th edition — code library (algs4.jar). https://algs4.cs.princeton.edu/code/

Sinnot, R. W. (1984). Virtues of the haversine. *Sky and Telescope*, *68*, 159.

Univocity. (2023). Univocity parsers: High-speed csv parsing for java. https://www.univocity.com/pages/univocity_parsers_tutorial.html