ULB UNIVERSITÉ LIBRE DE BRUXELLES

INFO-F203 - Projet d'algorithmique 2

Projet Mobilité

Auteurs :
Romain Liefferinckx - 000591790
Manuel Rocca - 000596086

Professeurs :
Jean Cardinal
Assistants :
Robin Petit

Année académique 2024-2025

Table des matières

1 Introduction

Dans le cadre de notre cours d'algorithmique INFO F-203, l'occasion s'est présentée à nous de créer un programme cherchant un chemin optimal entre un point A et un point B sur base d'une heure de départ. En effet, sur base d'un ensemble de données fournies sous format *General Transit Feed Specification* (**GTFS**), nous avons utilisé le *Connexion Scan Algorithm* (**CSA**) pour implémenter notre chercheur de chemin en *Java*.

Dans les sections à suivre, nous abordons l'initialisation des données, des structures formées à partir de celles-ci pour notre implémentation et certains détails techniques comme la complexité temporelle et spatiale. Nous justifierons également certains choix comme celui de l'algorithme précisé ci-dessus, à savoir le **CSA**.

2 Initialisation des données

Dans cette section nous expliquons les procédés utilisés pour charger les données en mémoire à partir des fichiers CSV fournis ainsi que les structures de données utilisées pour leur stockage et leur utilisation optimale dans l'algorithme choisi par nos soins.

Tout d'abord, décrivons brièvement la classe Parser, qui se charge de la lecture des fichiers CSV et de la création de dictionnaires (*HashMap*) contenant les données chargées, faciles d'accès. Nous avons créé, pour chaque type de fichier CSV, une classe de stockage qui contient les attributs correspondants aux champs du fichier CSV. Quelques subtilités notables au sujet de ces classes sont décrites dans les sections suivantes.

2.1 Objets de base pour le stockage

Les fichiers CSV fournis contiennent des informations sur les arrêts, les trajets, les horaires et les routes de quatre entreprises de transport en commun, à savoir la STIB en région Bruxelloise, TEC en région Wallone, DELIJN en région FLamande et, finalement, la SNCB pour le transport ferroviaire au niveau national.

Pour chaque type de fichier CSV, voici donc les classes correspondantes créées pour contenir les données de manière accessible.

2.1.1 La classe Route

Cette classe est une simple classe de stockage, chaque attribut correspondant à un champ des fichiers *routes.csv*.

Attribut	Type	Description
routeId	final String	L'identifiant de la route représentée
routeShortName	final String	Le nom de la route raccourci
routeLongName	final String	Le nom de la route complet
routeType	final String	Le type de véhicule utilisant cette route

Table 1 – Classe Route

2.1.2 La classe StopTime

La particularité de cette classe est qu'il lui manque l'identifiant de trajet donné dans les CSV concernés. Ce choix découle de la structure de la classe Trip détaillée dans la section 2.1.4.

Attribut	Type	Description
departureTime	final int	L'heure du départ à partir de l'arrêt associé sur le trajet associé en format heure ;minutes ;secondes
stopId	final String	L'identifiant de l'arrêt associé
stopSequence	final int	Le numéro de l'arrêt dans le trajet associé

TABLE 2 – Classe StopTime

2.1.3 La classe Stop

Dans cette classe, hormis le fait que chaque champ des fichiers *routes.csv* est repris, nous avons fait le choix d'ajouter une liste d'identifiants de trajet, permettant de retrouver efficacement chaque trajet partant de cet arrêt ainsi qu'une liste de *Walk*, permettant d'avoir un accès rapide à tous les arrêts accessibles à pied depuis cet arrêt.

Cette classe comprend également une méthode *getDistanceToOther* qui permet de calculer la distance entre deux arrêts en utilisant la formule de Haversine, implémentée dans une classe annexe, la classe *Calculator*.

Attribut	Type	Description
stopId	final String	L'identifiant du stop représenté
stopName	final String	Le nom du stop
stopLat	final String	La latitude du stop
stopLon	final String	La longitude du stop
tripIds	List<String>	La liste des tous les trips (leurs identifiants) partant de ce stop
walks	List<Walk>	La liste de tous les arrêts accessibles à pied depuis ce stop

TABLE 3 – Classe Stop

2.1.4 La classe Trip

Une fois de plus, en plus des champs trouvés dans les fichiers *trips.csv* retranscrits en attribut, nous avons ajouté une liste d'heures d'arrêt associées à ce trajet. Ceci nous permet de construire efficacement les connexions entre arrêts, affublées de temps de départ et d'arrivée.

Attribut	Type	Description
tripId	final String	L'identifiant du trip représenté
routeId	final String	L'identifiant de la route sur laquelle il passe
stopTimes	List<StopTime>	La liste de tous les StopTime associés à ce trip

TABLE 4 – Classe Trip

2.2 Création des structures de données

Une fois les données lues et contenues dans des dictionnaires faciles d'accès (accès en temps constant), nous procédons à la création de structures de données pertinentes à notre algorithme de recherche de chemin, le CSA à l'aide de la classe *Builder*.

La classe *Builder* est responsable de la création d'une liste de connexions entre arrêts, de la création du *BallTree* utilisé pour construire efficacement les transferts à pied entre arrêts et, finalement, la création de ces transferts à pied.

2.2.1 La classe Connexion

L'algorithme CSA se base, comme son nom l'indique, sur des connexions entre les différents arrêts. Pour traiter cet aspect, nous avons fait le choix logique d'implémenter une classe *Connexion* [Dib+17].

Attribut	Type	Description
tripId	final String	L'identifiant du trajet utilisé par la connexion
fromId	final String	L'identifiant de l'arrêt de de départ
toId	final String	L'identifiant de l'arrêt d'arrivée
departureTime	final int	L'heure de départ de l'arrêt *fromId*
arrivalTime	final int	L'heure d'arrivée à l'arrêt *toId*

Table 5 – Classe Trip

2.2.2 La classe BallTree

Afin de construire les transferts à pied (limités à une distance arbitraire), nous avons commencé par une implémentation brute-force en itérant, pour chaque arrêt, sur tous les autres arrêts en $O(n^2)$, n étant le nombre d'arrêts. Cependant, avec un total d'environ 65 000 arrêts, cette approche impliquait environ 4,5 milliards d'itérations avec, à chaque itération, une série d'opérations comme par exemple le calcul de la distance entre deux arrêts, en temps constant. Ceci nous a menés à un temps d'exécution d'environ 10 minutes sur nos machines rien que pour la création de ces transferts à pied, ce qui est, vous en conviendrez, absolument inacceptable.

Afin de remédier à ce problème critique, nous nous sommes penchés sur des structures de données plus adaptées à ce type de problème. Suite à une série de recherches et discussions avec nos collègues, nous avons opté pour le *BallTree*, une structure de données arborescente où chaque nœud représente une boule dans l'espace N-dimensionnel contenant un sous-ensemble d'arrêts. Cette structure permet de récupérer efficacement les voisins d'un arrêt donné dans un rayon donné à l'aide de requêtes rapides [DHMB15].

2.2.3 La classe Walk

Classe simple, représentant un transfert à pied entre deux arrêts. Ces objets sont contenus sous forme de liste dans les arrêts correspondants [Dib+17].

Attribut	Type	Description
departure	final Stop	L'arrêt de départ du transfert à pied
destination	final Stop	L'arrêt d'arrivée du transfert à pied
duration	final int	La durée en secondes du transfert à pied

TABLE 6 – Classe Walk

2.3 Complexité de l'initialisation

Bien qu'étant une étape antérieure à l'exécution de l'algorithme, il reste important de discuter la complexité de cette étape. En effet, si nous avions choisi d'absolument ignorer l'aspect de la complexité, nous aurions pu rester sur une implémentation brute-force, prenant 10 minutes à l'exécution. Bien entendu, pour éviter ces absurdités, nous avons apporté un soin particulier à la création de ces structures de données.

2.3.1 Complexité temporelle

L'initialisation des données se fait en plusieurs étapes, chacune ayant sa propre complexité. Premièrement, nous avons la lecture des fichiers CSV, qui a naturellement une complexité linéaire $O(n)$, où n est le nombre de lignes dans le fichier.

Ensuite, pour la création des connexions, nous itérons dans tous les trajets et, pour chaque trajet, nous itérons dans leurs heures d'arrêts associés. Notons n le nombre de trajets et m le nombre d'arrêts associés à chaque trajet. La complexité de cette étape est donc $O(n \cdot m)$. Une fois la liste de connexions créée, nous procédons au tri de celle-ci en se basant sur l'heure de départ de chaque connexion à l'aide de l'algorithme de tri *TimSort*, utilisé par *Collections.sort()* [Hel23]. Ceci se fait dans les pires et moyen cas avec une complexité de $O(n \log n)$, où n est le nombre de connexions avec un meilleur cas s'exécutant en $O(n)$ (si la liste est déjà majoritairement triée).

Finalement, le plus intéressant, la construction du *BallTree* et des transferts à pied. La construction de notre *BallTree* se fait de manière récursive, en divisant à chaque appel récursif l'ensemble des arrêts en deux sous-ensembles. Pour ce faire, des méthodes annexes comme *FindFarthest* ou *ComputeCenter* sont utilisées. Chacune d'entre elles s'exécute en temps linéaire, $O(n)$, où n est le nombre d'arrêts dans l'ensemble de données. Quatres fonctions linéaires sont executées à chaque appel récursif, ce qui nous amène à une complexité de $4 * O(n)$, ce que nous pouvons résumer à $O(n)$. Finalement, concernant les appels récursifs, en supposant le partitionnement parfait, nous avons une profondeur d'arbre de $\log n$ et donc $O(n \log n)$ pour la construction de l'arbre complet [DHMB15].

Suite à quelques essais et tests, nous nous sommes rendu compte que le partitionnement n'est pas toujours optimal, cela étant probablement dû à la non-uniformité dans la répartition des arrêts à travers la Belgique. Ceci nous permet de conclure, pour la complexité de la construction de notre *BallTree*, que nous avons une complexité de $O(n \log n)$ dans le cas moyen ou optimal et de $O(n^2)$ dans le pire des cas. Cependant, ce dernier cas est très peu probable dans la pratique. En effet, la construction de l'arbre se fait en 400 millisecondes en moyenne sur nos machines, ce qui est tout à fait acceptable contrairement à l'implémentation brute-force environ 1500 fois plus lente avec un temps d'exécution de 10 minutes.

Algorithm 1: BuildTree - Construction récursive d'un BallTree

```
Input: collection stops
Output: racine d'un BallTree ou null
if stops est vide then
    return null
if |stops| ≤ leafSize then
    return LeafNode(stops)
// Trouver deux pivots distants
(p1, p2) ← FindFartheststops
// Partitionner selon la distance aux pivots
left ← ∅,  right ← ∅
foreach s ∈ stops do
    if dist(p1, s) < dist(p2, s) then
        left ← left ∪ {s}
    else
        right ← right ∪ {s}
// Calcul du centre et du rayon du nœud interne
c ← ComputeCenterstops
r ← ComputeRadiusstops, c
node ← InternalNode(c, r)
// Appels récursifs
node.left ← BuildTreeleft
node.right ← BuildTreeright
return node
```

Une fois l'arbre construit, nous l'utilisons pour faire des requêtes pour trouver les arrêts à une distance donnée de tous les arrêts. Pour analyser la complexité d'une requête, nous allons considérer plusieurs scénarios. Premièrement, dans un cas moyen voire optimal, à savoir quand la plupart des noeuds sont prunés (c'est-à-dire que plusieurs sous-arbres ne sont pas visités, améliorant ainsi l'efficacité de la recherche) grâce à la condition *distToCenter - node.radius > maxDist* et que l'arbre est bien équilibré, la complexité est sous-linéaire en $O(\log n + k)$, où n est le nombre d'arrêts et k le nombre d'arrêts trouvés dans la distance donnée. En revanche, comme abordé plus haut, nous nous devons de considérer le cas où l'arbre est déséquilibré et/ou que la plupart de noeuds ne sont pas prunés durant la recherche (la plupart des sous-arbres sont visités car le rayon de recherche est grand). Dans ce cas, la complexité de la recherche tend vers la linéarité (parcours de l'arbre au complet) $O(n + k)$, où n est le nombre d'arrêts et k le nombre d'arrêts trouvés dans la distance donnée. Mais, une fois de plus, difficile d'arriver dans ce cas car la Belgique a une superficie de 30 528 km², une distance Nord-Sud de 225 km et une distance Ouest-Est de 282 km. Additionnellement, le rayon de recherche que nous utilisons est de 500 mètres, ce qui est relativement faible par rapport à la taille de la Belgique, nous obtenons une vitesse de requête de l'ordre de la 0,015 milliseconde (ce rayon est bien sûr changeable, mais nous avons considéré un rayon de transfert supposé acceptable pour la majorité de la population).

Algorithm 2: `knn_search` — Recherche en rayon dans un BallTree

```
Input: nœud node, requête q, distance max maxDist, collection out
Output: ajoute dans out tous les stops à distance ≤ maxDist de q
if node = null then
    return
if node est une feuille then
    foreach s ∈ node.stops do
        if dist(q, s) ≤ maxDist then
            out ← out ∪{s}
    return
distToCenter <- dist(q, node.center)
if distToCenter − node.radius > maxDist then
    return                                          // prune le sous-arbre
// Sinon, nous explorons les deux sous-arbres
knn_search(node.left, q, maxDist, out)
knn_search(node.right, q, maxDist, out)
```

Pour construire les transferts à pied, nous itérons donc sur tous les arrêts et, pour chaque arrêt, nous récupérons tous ses voisins à grâce au *BallTree* fraîchement construit. Ceci nous donne donc une complexité linéaire pour la boucle principale $O(n)$, où n est le nombre d'arrêts. Les requêtes de voisins ayant une complexité moyenne de $O(\log n + k)$, la construction des transferts à pied atteint une complexité totale de $O(n \log(n + k))$ pour n arrêts et k arrêts trouvés dans la distance donnée.

2.3.2 Complexité spatiale

Chaque donnée des fichiers CSV est chargée en mémoire, impliquant une complexité spatiale de $O(n)$, où n est le nombre de lignes total dans tous les fichiers réunis. Ensuite, les objets comme les connexions, le *BallTree* et les transferts à pied sont créés mais en utilisant les données initiales par référence. De ce fait, la complexité spatiale de ces objets se résume à leurs données utiles (*overhead*) et à la mémoire utilisée pour les références. En d'autres termes, la complexité spatiale de ces objets est très petite mais tout de même à considérer étant donné leur nombre.

Abordons maintenant un aspect de la complexité spatial qui n'est que 'temporaire'. Le tri *TimSort* utilise un espace supplémentaire de $O(n)$ pour le stockage temporaire des données pendant le tri uniquement. La construction du *BallTree* se faisant récursivement, il faut considérer les appels sur le stack. Ceux-ci dépendent en fonction de la taille des feuilles (le paramètre *leafSize*). En effet, celui-ci détermine le nombre maximal d'arrêts contenus sur une feuille. Plus ce nombre est grand, moins l'arbre sera grand et occupera d'espace en mémoire. En revanche les requêtes seront longues. D'autre part, plus ce nombre est petit, plus nous augmentons le nombre d'appels récursifs (risquant ainsi un StackOverflowError qui fait planter le programme) et plus nous augmentons la taille de l'arbre et donc l'espace en mémoire tout en affinant la vitesse de requêtes (note : une taille de feuille minimale ne signifie pas la vitesse de requête minimale). Voici un graphe pour illustrer ce propos :

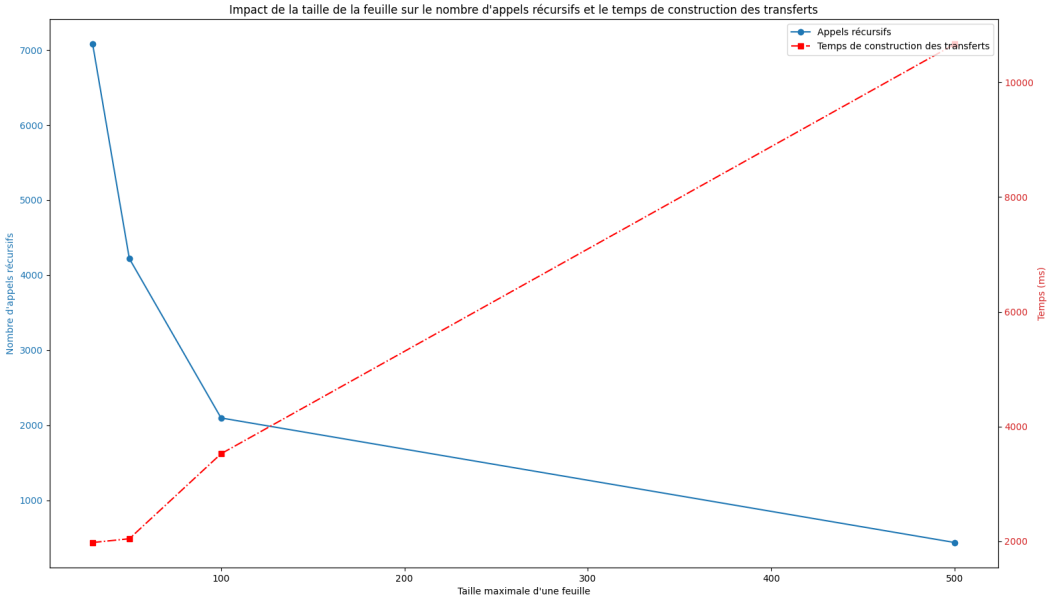

FIGURE 1 – Nombre d'appels récursifs et temps de construction des transferts à pied en fonction de la taille des feuilles

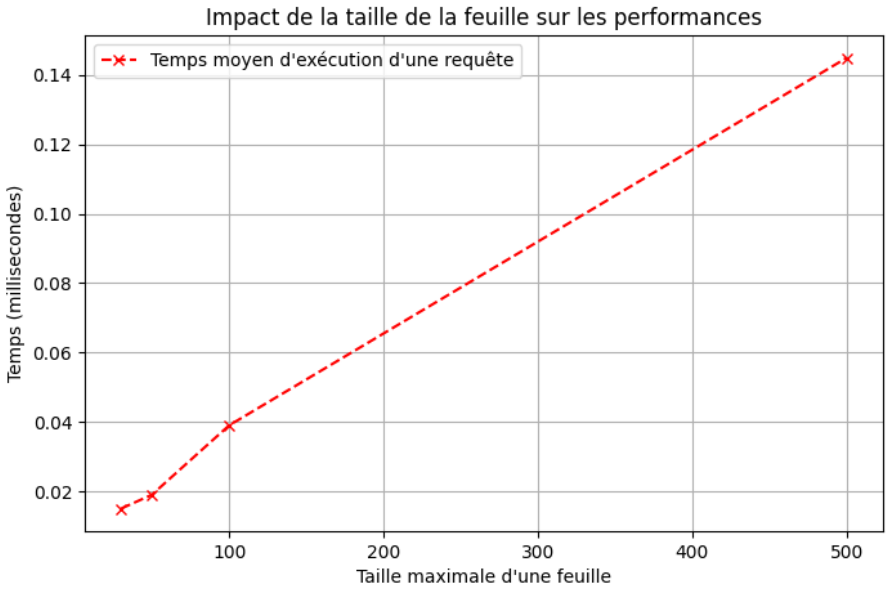

FIGURE 2 – Temps de requête moyen en fonction de la taille des feuilles

3 Le Connexion Scan Algorithm

3.1 Choix de l'algorithme

Comme expliqué plus haut, pour résoudre le problème de recherche du plus court chemin, nous avons choisi d'implémenter le *Connexion Scan Algorithm* (**CSA**). Cet algorithme est particulièrement adapté pour les réseaux de transport en commun, car il permet de trouver rapidement le meilleur chemin entre deux arrêts en tenant compte des horaires de départ et d'arrivée. Le CSA a été conçu spécifiquement pour traiter les problèmes de planification d'itinéraires dans les systèmes de transport en commun. Cependant, au travers de nos recherches, nous avons découvert des alternatives algorithmiques comme *Dijkstra* ou *A**. Nous jugeons donc important de justifier notre choix.

3.1.1 Fonctionnement

CSA : Basé sur une approche de recherche de chemin utilisant une liste triée de connexions entre arrêts, et parcourant ces connexions linéairement. Plus précisément, l'algorithme parcourt les connexions dans un ordre croissant suivant les temps de départ, en mettant à jour les meilleurs temps d'arrivée à chaque arrêt [Dib+17].

Dijkstra : Cherche le plus court chemin en partant du noeud de départ et en explorant tous les voisins, en choisissant toujours le chemin avec le coût total le plus faible. Il utilise une file de priorité pour gérer les noeuds à explorer [DE59].

A* : Une amélioration de Dijkstra ayant la particularité d'utiliser une heuristique afin de guider la recherche vers la destination. Nécessitant une heuristique optimale pour garantir la complétude et l'optimalité de l'algorithme. Celui-ci est donc plus rapide que Dijkstra à condition d'avoir une heuristique correcte [Foe+21].

3.1.2 Facilité d'implémentation avec GTFS

CSA : Essentiellement une simple itération sur les connexions.

Dijkstra : Nécessite de transformer les données de GTFS en graphe temporel (chaque arrêt devient un noeud). Cette implémentation est moins naturelle que celle du CSA mais marche tout aussi bien [DE59].

A* : En plus de la construction du graphe temporel, il est primordial de définir une heuristique efficace.

3.1.3 Complexité

CSA : $O(n)$, où n est le nombre de connexions (de manière naïve). Très rapide en pratique pour les requêtes *earliest arrival* grâce au tri temporel. Cependant, il demande un tri réalisé au préalable, durant l'initialisation des données, déjà expliqué en détail à la section 2.3.1. Pour le cas du CSA, nous verrons après qu'il y a encore des optimisations possibles pour diminuer la complexité temporelle.

Dijkstra : $O((V + E) \log V)$, où V est le nombre de noeuds et E le nombre d'arêtes du graphe temporel.

A* : $O((V + E) \log V)$, mais dépend de la qualité de l'heuristique.

3.1.4 Choix final

Les algorithmes Dijkstra et A* utilisent donc des structures de données plus complexes comme des graphes temporels et des files de priorité. Ceci implique donc un certain niveau de complexité supplémentaire dans l'implémentation.

Nous avons donc décidé de miser sur la simplicité d'implémentation et la rapidité de l'algorithme CSA, celui-ci se basant essentiellement sur une liste de connexions triées. De plus, celui-ci est par défaut adapté aux fichiers GTFS et les horaires de transports publics [Lin].

3.2 Explication et description de l'algorithme

Le *Connexion Scan Algorithm* est une méthode efficace pour résoudre les problèmes de planification d'itinéraires dans les réseaux de transports en commun basés sur des horaires, comme les trains et les bus. Contrairement aux algorithmes classiques qui utilisent une file de priorité (comme Dijkstra), le CSA se base sur une liste triée de tous les transports en commun, classés par leur heure de départ.

Il parcourt cette liste de manière séquentielle pour déterminer rapidement les meilleures connexions possibles pour atteindre une destination dans un délai optimal ou avec le moins de changements de véhicule.

3.3 Structures de l'algorithme et implémentation

L'algorithme est stocké dans une classe *PathFinder* contenant la méthode findPath, qui implémente l'algorithme CSA.

3.4 Pseudo-code sans la marche à pied et autres optimisations

Algorithm 3: CSA — Recherche de chemin entre deux arrêts sans marche à pied

Input: start, destination, time
Output: Affiche le chemin optimal ou un message d'erreur

```
// La liste de connexions est triée par heure de départ préalablement
   durant le parsing
foreach stopId dans tous les arrêts do
    shortestPath[stopId] ← ∞
foreach startingStop dans startingStops do
    shortestPath[startingStop] ← userStartTime
foreach connexion c dans connexions do
    if shortestPath[dep] ≤ depTime et shortestPath[arrive] > arrTime then
        Mettre à jour shortestPath[arrive] et previousConnection[arrive]
    foreach walk depuis depart do
        if shortestPath[walkArr] > walkTime then
            Mettre à jour shortestPath[walkArr] et previousConnection[walkArr]
currentStop ← end_stop
while currentStop dans previousConnection do
    Ajouter la connexion à path au début
    currentStop ← origine de la connexion
Afficher le chemin trouvé
```

3.5 Complexité sans la marche à pied et autres optimisations

Une fois l'initialisation des données terminée (cf. section 2), l'algorithme CSA est donc relativement direct à implémenter. Il débute par le parcours de toutes les connexions une par une. Cette opération est naturellement linéaire, $O(n)$, où n est le nombre de connexions. Cette complexité est particulièrement avantageuse pour les grands ensembles de données, car elle permet un traitement efficace des requêtes après le prétraitement initial.

Tout ceci nous amène donc à une complexité totale de $O(n)$ pour la recherche de chemin, où n est le nombre de connexions.

Notons tout de même le fait que cette complexité est valable uniquement dans le cas où nous ne considérons pas les transferts à pied, chose que nous explicitons dans la section suivante.

3.6 Pseudo-code avec la marche à pied et algorithme optimal

Algorithm 4: CSA — Recherche de chemin avec critères d'arrêt et marches à pied

```
Input: startName, destinationName, userTime
Output: Affiche le chemin optimal ou un message d'erreur
foreach stopId dans tous les arrêts do
    shortestPath[stopId] ← ∞
foreach startingStop dans startingStops do
    shortestPath[startingStop] ← userStartTime
bestArrivalTime ← ∞ // Initialisation du meilleur temps d'arrivée

// Starting criterion
startIndex ← index de la première connexion avec depTime >= userStartTime (recherche dichotomique)
for i ← startIndex to connexions.size() do
    c ← connexions[i]
    if c.depTime > bestArrivalTime then
        break                    // Stopping criterion — plus de chemin possible
    if shortestPath[c.dep] ≤ c.depTime et shortestPath[c.arr] > c.arrTime then
        shortestPath[c.arr] ← c.arrTime
        previousConnection[c.arr] ← c
        if c.arr ∈ endStops et c.arrTime < bestArrivalTime then
            bestArrivalTime ← c.arrTime
        foreach walk ∈ walks depuis c.arr do
            walkArr ← walk.destination
            walkTime ← c.arrTime + walk.duration
            if shortestPath[walkArr] > walkTime then
                shortestPath[walkArr] ← walkTime
                previousConnection[walkArr] ← Connexion de type marche
while currentStop dans previousConnection do
    Ajouter la connexion à path au début
    currentStop ← origine de la connexion
Afficher le chemin trouvé
```

3.7 Complexité avec la marche à pied et algorithme optimal

Tel que mentionné précédemment, la complexité de l'algorithme principal est de l'ordre de $O(n)$, où n représente le nombre de connexions. Mais, il n'est clairement pas logique de considérer un voyage où nous ne marchons pas du tout. C'est pourquoi nous devons aussi traiter les transferts à pied. Lors de l'exécution de l'algorithme, à chaque itération, autrement dit à chaque connexion, nous devons vérifier si l'arrêt de départ concerné possède des arrêts voisins à une distance donnée (500 mètres dans notre cas, cf. 2.3.1.). Un transfert entre deux arrêts voisins est représenté comme un *Walk*, une classe décrite dans la section 2.2.3.. Donc, pour chaque arrêt nous allons parcourons l'ensemble des transferts à pied disponibles. En prenant $\bar{m}$ comme nombre moyen de transferts à pied moyen, nous avons une complexité totale de $O(n * \bar{m})$, avec n étant toujours le nombre de connexions.

Nous pourrions croire que la complexité de l'algorithme avec les transferts à pied se rapproche de $O(n^2)$, mais ce n'est pas le cas. Pour répondre à cette incertitude, nous avons tout simplement calculé la moyenne du nombre de transferts à pied par arrêt à l'aide d'un script Python. Ceci nous a menés à une valeur de 7 voisins dans un rayon de 500 mètres par arrêt. Nous concluons donc que, avec un nombre de connexions étant de l'ordre de dizaines de millions, $\bar{m}$ est négligeable par rapport à n et nous arrivons à une complexité totale de $O(n)$.

3.7.1 Optimisations

Comme montré ci-dessus, l'algorithme n'est pas exactement linéaire. Nous avons donc décidé d'implémenter deux optimisations afin d'améliorer la vitesse de l'algorithme.

- **Starting criterion :** Nous utilisons l'algorithme élémentaire qu'est la recherche dichotomique (en $O(\log n)$), pour trouver le premier arrêt qui est supérieur ou égal à l'heure de départ. Ceci nous évite le parcours toutes les connexions en nous permettant de commencer à partir de l'indice trouvé. À travers ce procédé, nous réduisons le nombre d'itérations nécessaire pour trouver le chemin optimal. En effet, pour une requête de trajet débutant en fin de journée, pourquoi itérer dans les premières connexions ? Ceci est tout naturellement inutile. Grâce à cela, l'algorithme passe de $O(n)$ à $O(n-k)$ avec k le nombre de connexions avant l'heure de départ. La variable k pouvant prendre des valeurs de 0 à n, elle n'est pas négligeable dans le calcul de complexité totale. En conclusion, cela nous permet de grandement réduire le temps d'exécution de l'algorithme, et encore plus si l'heure de départ est élevée. Toutefois, si l'heure entrée par l'utilisateur est la première dans la liste de connexion, la complexité ne change pas et reste $O(n)$ car nous devons parcourir toutes les connexions.
- **Stopping criterion :** Cette seconde optimisation nous permet d'arrêter l'algorithme soit lorsque la destination atteinte, soit lorsque toutes les connexions possibles ont été explorées. Cela nous éviter le parcours inutile des connexions qui ne contribuent pas à améliorer le chemin trouvé. En pratique, dans le cas où l'heure d'arrivée à un arrêt est déjà optimale (c'est-à-dire qu'aucune connexion ultérieure ne peut améliorer cette heure), cela signifie que nous pouvons ignorer les connexions restantes. Cette optimisation réduit potentiellement une fois de plus le nombre d'itérations nécessaires afin de trouver le chemin optimal. En ce qui concerne la complexité, en prenant la complexité de base de l'algorithme, nous passons de $O(n)$ à $O(n-j)$ où j est le nombre de connexions que nous avons pu soustraire. Ceci nous permet une fois de plus d'avoir de meilleures performances globales.
- **Limited walking :** Une troisième optimisation que nous avons implémentée est la limitation du nombre et des distances des transferts à pied. Nous parlons de la distance choisie et ce que cela apporte dans la section 2.3.1. donc nous n'en reparlerons plus ici. Cela dit, nous aurions également pu implémenter une limite sur le nombre de transferts à pied. Sauf qu'en y ayant bien réfléchi, nous avons décidé de ne pas le faire car cela aurait pu amener à des chemins non optimaux et même impossibles pour des chemins demandant beaucoup de correspondances. En effet, si nous avions limité le nombre de transferts à pied, l'algorithme aurait pu trouver un chemin avec beaucoup de correspondances mais être dans l'impossibilité car il serait limité à un nombre de transferts à pied trop faible.

En somme, en assemblant nos deux optimisations, nous obtenons une complexité totale de $O(n-k-j)$ pour le coeur de l'algorithme [Dib+17].

3.8 Chiffres concrets

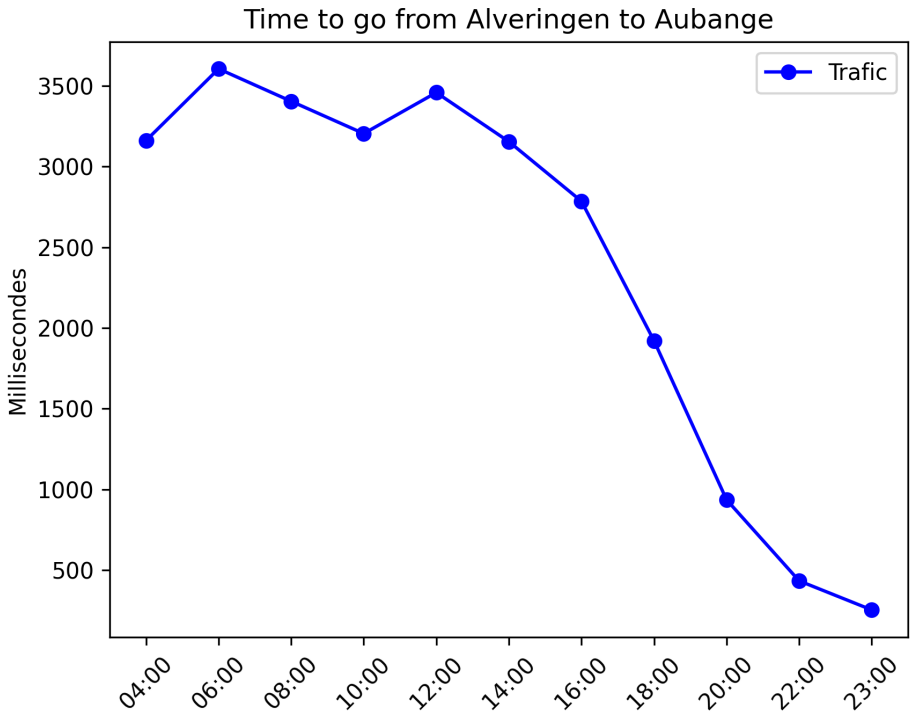

FIGURE 3 – Graphique du temps d'exécution en fonction de l'heure de départ.

De part ce graphe, nous pouvons voir que le temps d'exécution de l'algorithme est relativement constant jusqu'à 16 heures, puis grâce au *Starting criterion*, l'algorithme devient de plus en plus rapide. En effet, à partir de 16 heures, il y a moins de connexions et il arrive a chaque fois à trouver le chemin optimal en mois d'une seconde. Nous pouvons aussi parler du *Stopping criterion*, qui lui permet de ne pas parcourir toutes les connexions et donc de ne pas perdre de temps à chercher des connexions inutiles. Bien que nous ne puissions pas le voir sur le graphe, il est tout de même présent et permet de réduire le temps d'exécution de l'algorithme.

3.9 Preuve du fonctionnement de l'algorithme

Finalement, dans cette section concernant l'algorithme CSA, nous allons prouver sa véracité, pour chaque sommet v accessible depuis une source s, une valeur $d(v)$ correspondant à la longueur d'un plus court chemin de s à v. Pour cela, nous vérifions qu'il satisfait les trois conditions énoncées à la *Proposition 26* trouvée dans le Syllabus d'algorithme 2, qui sont nécessaires et suffisantes [Car24].

Proposition 26 : Soit $G = (V, \mathcal{E})$ un graphe orienté, pondéré positivement sur les arcs, $s \in V$ un sommet source, et une fonction d : V -> R. On a :

1. $d(s) = 0$;
2. Pour tout sommet $v \in V$, $d(v)$ est la longueur d'un chemin de s vers v ;
3. Pour tout arc $e = (u, v) \in \mathcal{E}$, $d(v) \leq d(u) + w(e)$.

Démontrons maintenant que l'algorithme CSA vérifie chacune de ces conditions.

Condition 1 : $d(s) = 0$. L'algorithme initialise, pour chaque arrêt de source s, l'heure d'arrivée à la valeur de départ τ_0. On note :

$d(s) := \tau_0 - \tau_0 = 0$. Cela revient à poser $d(s) = 0$ en considérant que le départ est à l'instant zéro. Cette condition est donc bien satisfaite.

Condition 2 : Pour tout sommet $v \in V$, $d(v)$ est la longueur d'un chemin de s vers v.

Preuve : Par récurrence sur l'ordre topologique des sommets, soit $v_1, v_2, \ldots, v_n$ un ordre topologique de G. On note $P(v)$ un plus court chemin de s vers v découvert par l'algorithme.

Cas de base : $v_1 = s$. Par définition, $d(s) = 0$ et il existe un chemin trivial de s à s de longueur 0. La propriété est vraie.

Démonstration : Supposons que pour tout v_i avec $i < k$, il existe un chemin $P(v_i)$ de s à v_i tel que $d(v_i)$ est la longueur de ce chemin. Considérons maintenant v_k. Par l'ordre topologique, tous les prédécesseurs u de v_k ont déjà été traités. Pour chaque arc $(u, v_k) \in \mathcal{E}$, l'algorithme met à jour :

$d(v_k) := \min_{(u,v_k)\in\mathcal{E}} \{d(u) + w(u, v_k)\}$.

Par hypothèse de récurrence, pour chaque u, tel qu'il existe un chemin $P(u)$ de s à u de longueur $d(u)$. En ajoutant l'arc (u, v_k), on obtient un chemin $P(u) \cdot (u, v_k)$ de s à v_k de longueur $d(u) + w(u, v_k)$. Comme l'algorithme prend le minimum parmi tous ces chemins, il sélectionne l'un d'eux comme étant $P(v_k)$, un chemin effectif de s à v_k, de longueur :

$d(v_k) = \min_{(u,v_k)\in\mathcal{E}} \{d(u) + w(u, v_k)\}$.

Donc $d(v_k)$ est bien la longueur d'un chemin de s vers v_k. Par conséquent, tout sommet v possède un chemin de s vers v de longueur $d(v)$. La condition 2 est donc satisfaite.

Condition 3 : Inégalité triangulaire $d(v) \leq d(u) + w(e)$. À chaque traitement d'un arc $e = (u, v)$ (d'une connexion (u, v) avec un poids $w(e)$ égal à *connexion.getArrivalTime−connexion.getDepartureTime*), l'algorithme vérifie si :
$d(v) > d(u) + w(e)$ et effectue la mise à jour dans ce cas. Une fois toutes les connexions traitées, aucune mise à jour n'est plus possible, ce qui implique : $d(v) \leq d(u) + w(e), \quad$ pour tout $(u, v) \in \mathcal{E}$. Cela garantit que la fonction d satisfait l'inégalité triangulaire pour tous les arcs du graphe.

Fin de la preuve : Les trois conditions de la Proposition 26 sont remplies par l'algorithme CSA. On en déduit que les valeurs $d(v)$ calculées représentent exactement les longueurs des plus courts chemins en temps du sommet s vers tous les sommets atteignables dans le graphe.

4 Classes utilisées

Afin de clôturer ce rapport, revenons sur les classes utilisées dans notre code source afin de tout clarifier. À la section 2, nous expliquons en détail les classes *Route*, *StopTime*, *Stop*, *Trip*, *Connexion*, *BallTree* et, finalement, *Walk*. Nous ne reviendrons donc pas sur celles-ci.

Certaines classes ne sont que mentionnées dans les sections précédentes, à savoir les classes trouvées dans le dossier *functional*.

- **BinarySearch :** Une classe utilitaire possédant une unique méthode statique *findStartIndex* permettant d'appliquer la première optimisation, c'est-à-dire le *Starting criterion*. Cette méthode est une implémentation de la recherche dichotomique.
- **Calculator :** Une classe utilitaire possédant plusieurs méthodes statiques. Nous y trouvons d'abord *haversine_distance* permettant de calculer la distance entre deux arrêts en utilisant la formule de Haversine. Ensuite, nous avons une série de méthodes de conversion temporelles,

plus précisément des moyens de convertir un temps en format *HH :mm :ss* en secondes et inversement.
- **Initializer :** Une classe s'occupant de toute la partie initialisation des données. Elle possède une instance de la classe *Parser* qui lit les fichiers CSV et les convertit en objets Java et une instance de la classe *Builder* qui, sur base des données chargées, construit les structures utiles à l'exécution du CSA.
- **Parser :** Une classe s'occupant de la lecture des fichiers CSV et de la conversion de chaque ligne en objets Java. Elle est utilisée par la classe *Initializer*.
- **Builder :** Une classe s'occupant de la construction des objets Java à partir des données lues par la classe *Parser*. Elle est utilisée par la classe *Initializer*.
- **PathFinder :** La pièce maîtresse du programme, elle contient la méthode *findPath* implémentant l'algorithme CSA. D'autres méthodes annexes sont également trouvables comme une méthode pour afficher un transfert à pied, ou une connexion. Nous soulignons la présence de la méthode *findStopsByName* qui permet d'ajouter la propriété multisource à notre algorithme, à savoir, pour un nom d'arrêts, nous récupérons tous les autres arrêts possédant ce nom.

5 Commentaires

Au travers de ce rapport, nous avons justifé nos choix d'implémentation, de structures de données et autres d'un point de vue théorique et pratique. Nous avons développé et démontré la complexité des étapes de notre programme et expliqué pourquoi celles-ci sont optimales.

Suite à des conversations avec nos collègues étudiants suivies de recherches supplémentaires, nous nous sommes rendu compte que notre implémentation n'est pas la plus optimale. Nous dédions donc cette section à souligner les défauts de notre implémentation et à expliquer comment nous aurions pu l'améliorer.

5.1 Les requêtes de voisins

L'utilisation du *BallTree*, bien qu'amplement justifiée, n'est pas la plus optimale. En effet, le *BallTree* est en fait une structure de données pour la recherche des plus proches voisins adaptée pour des données de hautes dimensions. Dans notre cas, nous utilisons que deux dimensions, à savoir la latitude et la longitude. Certaines alternatives plus intéressantes comme par exemple le *R-Tree*, un arbre équilibré de recherche spatiale permettant une recherche plus rapide [Gut].

5.2 Le choix de l'algorithme

Bien que le *Connexion Scan Algorithm* soit rapide et simple à implémenter, nous avons constaté qu'il est peu adapté à la prise en compte de critères personnalisés autres que l'heure (comme le type de transport préféré). L'algorithme montre donc des limites en termes de flexibilité et de généricité.

Nous avons envisagé une piste consistant à ne construire que les connexions correspondant aux contraintes de l'utilisateur. Par exemple, si un utilisateur souhaite éviter les bus, il serait possible d'exclure toutes les connexions de ce type lors de l'initialisation. De même, les transferts à pied pourraient être générés avec un rayon réduit pour des utilisateurs ne souhaitant pas marcher longtemps.

Nous n'avons pas poursuivi cette piste car celle-ci impliquait une modification au niveau de l'initialisation des données avec le point focal du projet étant l'aspect algorithmique. Nous avons donc jugé cela non pertinent.

Afin de répondre à cette problématique, nous avons fait usage de *flags* contenus en tant qu'attributs dans la classe *PathFinder*. Ces *flags* sont des booléens permettant d'exclure certains types

de connexions directement durant l'exécution de l'algorithme. L'utilisateur peut donc choisir s'il ne souhaite pas prendre le bus, le tram ou autres par exemple.

Malheureusement, cette solution ne répond pas à toutes les situations. Illustrons ceci par un exemple :

Un utilisateur souhaite se rendre d'un arrêt dense en services de transports à un arrêt éloigné peu desservi. En supposant que l'utilisateur ne souhaite pas prendre le train et que ces deux arrêts uniquement reliés par une ligne de train, l'algorithme ne trouvera pas de chemin.

Notre algorithme ne prend pas en compte le concept de 'poids' et n'est donc pas complètement générique, mais nous avons tout de même cherché une solution comme décrit ci-dessus.

6 Conclusion

Ce projet nous a menés vers la découverte de nombreux concepts intéressants comme la recherche de voisins les plus proches à l'aide de différentes structures de données ainsi que la recherche du plus court chemin au moyen de différents algorithmes. En particulier, l'algorithme *Connexion Scan Algorithm*, choisi par nos soins afin de répondre à la problématique proposée par ce projet est donc une solution rapide et efficace pour la recherche du plus court chemin sur base d'un ensemble de connexions. Cependant, au travers, d'échanges et de recherches annexes, nous nous sommes rendu compte de certains défauts. Comme expliqué dans la section précédente, malgré sa facilité d'implémentation, il ne répond pas à l'ensemble de la problématique imposée. Nous tenions donc à souligner les défauts de l'implémentation et les pistes d'amélioration possibles par souci de complétude et de transparence.

7 Sources-bibliographie

Références

[Car24] Jean Cardinal. *Algorithmique 2*. 2024-2025. url : https://uv.ulb.ac.be/pluginfile.php/4036956/mod_resource/content/2/Algorithmique.pdf.

[DE59] Dijkstra et E.W. "A note on two problems in connexion with graphs". In : *Numerische Mathematik* 1 (1959), p. 269-271. doi : 10.1007/BF01386390. url : https://doi.org/10.1007/BF01386390.

[DHMB15] Mohamad Dolatshah, Ali Hadian et Behrouz Minaei-Bidgol. "Ball*-tree : Efficient spatial indexing for constrained nearest-neighbor search in metric spaces". In : (2015). url : https://arxiv.org/pdf/1511.00628.

[Dib+17] Julian Dibbelt et al. "Connection Scan Algorithm". In : (mars 2017), p. 1-49. url : https://arxiv.org/pdf/1703.05997.

[Foe+21] Daniel Foeada et al. "A Systematic Literature Review of A* Pathfinding". In : (2021), p. 1-3. url : https://doi.org/10.1016/j.procs.2021.01.034.

[Gut] Antonin Guttman. *R-TREES - A DYNAMIC INDEX STRUCTURE FOR SPATIAL SEARCHING*. url : http://www-db.deis.unibo.it/courses/SI-LS/papers/Gut84.pdf.

[Hel23] Mohamed Helmy. "Time Complexity of Java - Collections Sort in Java". In : (2023). url : https://www.baeldung.com/java-time-complexity-collections-sort.

[Lin] Github repositories. url : https://linkedconnections.org/tools.