Réseaux de transport : Algorithmes et Structures de données

Rapport

Bilal Vandenberge

Mai 2025

1 Introduction

Ce rapport sert de point d'entrée pour le projet du cours Algorithmique 2 de l'Université libre de Bruxelles pour l'année académique 2024-2025. Il a été demandé de livrer un programme Java dont le but est de convertir des données stockées dans des fichiers sous le format GTFS en une ou plusieurs structures de données permettant la recherche du chemin optimal selon certains critères. Les aspects importants de l'algorithme sont la validité, l'efficacité et la généricité.

Définition du problème

À partir des entrées suivantes :

- Des données GTFS représentant des réseaux de transport public.
- Le nom d'une gare source que nous appellerons s.
- Le nom d'une gare cible que nous appellerons t.
- Une heure de départ que nous appellerons h.

Sous les conditions suivantes :

- Les données GTFS sont stockées dans un ensemble de fichiers sous contraintes détaillées dans la sous-section ci-dessous.
- s correspond au nom d'au moins une gare.
- t correspond au nom d'au moins une gare.
- h est compris entre 00:00:00 et 99:99:99.

L'algorithme doit retourner un chemin valide partant à `h` de `s` à `t` qui est considéré le moins coûteux selon certains critères. Le critère par défaut est la durée totale du chemin (que nous cherchons à minimiser).

Définition 1 (Chemin)**.** Un **chemin** est une suite d'arcs.

Définition 2 (Arc)**.** Un **arc** est le déplacement d'une gare à une autre.

Définition 3 (Connexion)**.** Une **connexion** est un type d'arc représentant le déplacement d'un transport d'une gare à une autre caractérisé par une heure de départ et une durée.

Définition 4 (Transfert)**.** Un **transfert** est un type d'arc représentant le déplacement à pied entre deux gares proches caractérisé par une durée et sans contrainte de départ (la personne peut partir à l'heure qu'elle veut).

Définition 5 (Temps d'attente)**.** Le **temps d'attente** $W_{(v,e)}$ d'un arc e après une gare v (donc $v =$ gare de départ de e) est la durée qui sépare l'heure d'arrivée à v et l'heure de départ avec le transport de e depuis v. Si arr représente l'heure d'arrivée à v et dep l'heure à laquelle nous quittons v via e :

$$W_{(v,e)} = \begin{cases} (dep - arr)\%\texttt{24:00:00} & \text{si e est une connexion} \\ 0 & \text{si e est un transfert} \end{cases}$$

où $\%$ est le modulo euclidien [2].

Cette définition est délicate : pourquoi effectuons-nous un modulo sur la différence entre le départ et l'arrivée ? Cela nous permet de nous débarrasser de deux problèmes. Sachant que les données GTFS sont périodiques (c'est-à-dire elles se répètent jour après jour) :

1. Un temps d'attente ne peut être négatif : l'utilisateur peut attendre le jour suivant pour prendre le transport de e. Par exemple, si l'utilisateur manque le dernier métro à 23:00:00, et que le premier métro le lendemain est à 05:30:00, le temps d'attente n'est pas de -17 heures et 30 minutes (illégal) mais de 6 heures et 30 minutes.
2. Un temps d'attente supérieur à 24:00:00 peut et doit être réduit. Pour rappel nous autorisons des heures dans les fichiers GTFS et pour h jusque 99:99:99. Par exemple, 28:00:00 est 02:00:00 du lendemain. Ainsi, vu que les données GTFS sont périodiques, l'attente jusque 28:00:00 peut être réduit, avec un modulo, à l'attente jusque 02:00:00.

Définition 6 (Validité d'un chemin)**.** Formellement, un chemin P est dit **valide** si pour deux arcs consécutifs de P, e de la gare u à la gare v et j de la gare k à la gare l, v la gare d'arrivée de e est identique à k la gare de départ de j et le temps d'attente $W_{(v,j)}$ est positif. La définition du temps d'attente ci-dessus indique que celui-ci sera toujours positif, nous pouvons résumer la définition à :
Un chemin P est dit **valide**, si pour toute paire d'arcs consécutifs dans P, la gare d'arrivée du premier arc est identique à la gare de départ du second.

Format des fichiers GTFS

Le format utilisé par le programme est une version allégée de la General Transit Feed Specification [12].

Les fichiers GTFS doivent être stockés dans un dossier composé d'un sous-dossier pour chaque agence. Eux-mêmes doivent être composés de 5 fichiers : `routes.csv`, `trips.csv`, `stops.csv`, `stop_times.csv`.

- `routes.csv` contient les informations de chaque ligne (nom et type de transport).
- `trips.csv` liste tous les trajets pour chaque ligne.
- `stops.csv` contient les informations de chaque gare (nom et position géographique).
- `stop_times.csv` range le contenu des trajets, c'est-à-dire la séquence de gares visitées par le transport pour chaque trajet.

La figure 1 représente un exemple avec l'arbre du dossier contenant les données des grandes agences de transport public belges.

```
GTFS
├── DELIJN
│   ├── routes.csv
│   ├── stops.csv
│   ├── stop_times.csv
│   └── trips.csv
├── SNCB
│   ├── routes.csv
│   ├── stops.csv
│   ├── stop_times.csv
│   └── trips.csv
├── STIB
│   ├── routes.csv
│   ├── stops.csv
│   ├── stop_times.csv
│   └── trips.csv
└── TEC
    ├── routes.csv
    ├── stops.csv
    ├── stop_times.csv
    └── trips.csv
```

FIGURE 1 – Exemple de structure d'un dossier GTFS

routes.csv

Un fichier `routes.csv` est composé des colonnes (route_id, route_short_name, route_long_name, route_type).

route_id	route_short_name	route_long_name	route_type
SNCF-1	IC	LYON - PARIS NORD	TRAIN
SNCF-2	L	MODANE - TORINO	TRAIN
STIB-12	19	SIMONIS - DE WAND	TRAM
...	...	...	...

TABLE 1 – Exemple d'un fichier `routes.csv`

trips.csv

Un fichier `trips.csv` est composé des colonnes (trip_id, route_id). Chaque ligne peut avoir plusieurs trajets prenant place à des heures différentes dans la journée.

trip_id	route_id
SNCF-2500_2–250025F_0	SNCF-128
SNCF-2500_2–250025F_1	SNCF-128
STIB-125400536291553002	STIB-12
...	...

TABLE 2 – Exemple d'un fichier `trips.csv`

stops.csv

Un fichier `stops.csv` est composé des colonnes (stop_id, stop_name, stop_lat, stop_lon). Les deux dernières colonnes représentent la latitude et longitude de la gare. Comme dit précédemment, `s` et `t` doivent référer à une gare et donc correspondre à au moins une entrée `stop_name`.

stop_id	stop_name	stop_lat	stop_lon
SNCF-S05	Paris Nord	48.880619	2.355638
SNCF-S08	Phare des baleines	46.2431	-1.558861
STIB-8711	AUMALE	50.839462	4.312081
...	...	...	...

TABLE 3 – Exemple d'un fichier `stops.csv`

stop_times.csv

Un fichier `stop_times.csv` est composé des colonnes (trip_id, departure_time, stop_id, stop_sequence). L'heure de départ (`departure_time`) est représentée sous le format HH:MM:SS. Une heure n'est pas bornée à 24:00:00 car dans le cas où un trajet débute tard dans la soirée (23h50 par exemple), il doit pouvoir continuer jusqu'au petit matin. Nous placerons une limite à 99:99:99 pour éviter des erreurs d'overflow dans l'implémentation et garder un affichage propre dans les réponses du programme.

La position (`stop_sequence`) précise la position de la gare dans le trajet tel que le transport passe par chaque gare dans un ordre croissant des positions. Il est important de noter que le format n'oblige pas le fichier à être ordonné par positions et les trajets à être contigu (que chaque position de 1 à n, le nombre de gares du trajet, apparaisse une et une seule fois) mais il doit exister au plus une gare dans le trajet pour chaque position.

trip_id	departure_time	stop_id	stop_sequence
SNCF-2500_2–250025F_0	08:00:00	SNCF-S13	1
SNCF-2500_2–250025F_0	08:34:00	SNCF-S88	2
SNCF-2500_2–250025F_0	09:10:00	SNCF-S08	4
...	...	...	...

TABLE 4 – Exemple d'un fichier `stop_times.csv`

Exemple du problème

Afin d'éclaircir les différents algorithmes et structures de données abordés dans ce rapport, nous utiliserons tout au long un exemple simple et bref afin d'y appliquer les concepts. Dans celui-ci, nous tentons de nous déplacer de Québec à Montréal dès 8h45 du matin dans le réseau ferroviaire fictif du Canada, la TOC (`Transport of Canada`) (voir figure **??**). Les données GTFS de cet exemple se trouvent dans le dossier `src/main/resources/TOC_GTFS`. Nous posons donc :

- $s =$ "Québec"
- $t =$ "Montréal"
- $h =$ 08:45:00

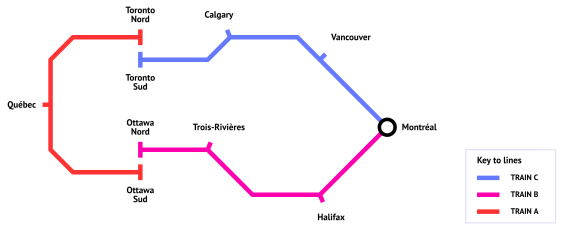

FIGURE 2 – Réseau ferroviaire de la TOC

2 Solution

Le problème que nous souhaitons résoudre est connu sous le nom de Time-Dependent Shortest Path, la recherche d'un chemin qui minimise un ou des critères dans un graphe aux arcs dont le poids (positif) varie en fonction du temps. L'algorithme que nous utilisons pour le résoudre est une variante de l'algorithme de Dijkstra [5, 3] connu sous le nom de TD Dijkstra (Dijkstra dépendant du temps ou Time-Dependent Dijkstra).

Structures de données utilisées

Représentation du graphe

Nous travaillons avec une adaptation simplifiée d'un graphe dépendant du temps (ou Time-Dependent graph en anglais) [15]. Le graphe $G(V, A, T)$ est composé de trois ensembles :

- V l'ensemble des nœuds, des gares
- A l'ensemble des arcs de connexion
- T l'ensemble des arcs de transfert

Nous implémentons ce graphe à l'aide d'une liste d'adjacence. Chaque nœud garde une liste des connexions et transferts sortants de celui-ci. Ce graphe compte comme propriétés :

- G est un multigraphe.
- Les arcs de connexions ont des poids positifs dépendants du temps (= un temps d'attente + une durée).
- Les arcs de transferts ont des poids positifs constants dans le temps (= une durée).

La figure 3 représente le graphe dépendant du temps de l'exemple de la TOC (les noms ont été remplacés par les initiales : Québec → Q etc.). Elle montre que pour cet exemple, il y a en tout 10 nœuds, 16 arcs de connexion et 4 arcs de transfert. Les arcs de connexion ont été déterminés à partir du contenu de `src/main/resources/TOC_GTFS/TOC/stop_times.csv`. L'évaluation des arcs de transfert est expliquée dans la sous-section ci-dessous.

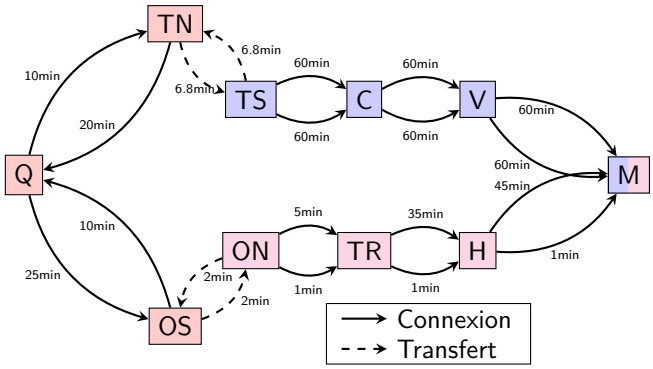

FIGURE 3 – Graphe de la TOC

Transferts à pied

Il doit être possible pour un utilisateur de changer de gares (Notons que sans cela, l'utilisateur ne pourrait pas changer de réseau : Dans l'exemple belge, un utilisateur commençant à une gare SNCB ne pourra jamais accéder à une gare STIB ou `De Lijn` ou TEC). Premièrement, il est fondamental de définir un critère de proximité entre deux gares. Si aucune limite n'était fixée, nous devrions stocker et considérer pour chaque gare un chemin à pied avec toutes les autres gares du graphe (nous voulons éviter des comportements quadratiques comme celui-ci). Nous définissons donc :

Définition 7 (Critère de proximité)**.** Le **critère de proximité** est la distance x (en mètres) tel que pour tout transfert t représenté dans le graphe, la distance à vol d'oiseau entre les deux gares n'excède pas x :

$$\forall t : (u, v) \in T, \quad d_{(u,v)} \leq x$$

où $d_{(u,v)}$ est la distance géographique entre u et v.

Définition 8 (Distance géographique)**.** La **distance** $d_{(u,v)}$ entre deux points géographiques u et v sur le globe terrestre :

$$d_{(u,v)} = 2R \cdot \arcsin\left(\sqrt{\sin^2\left(\frac{v_{lat}-u_{lat}}{2}\right) + \cos(u_{lat}) \cdot \cos(v_{lat}) \cdot \sin^2\left(\frac{v_{long}-u_{long}}{2}\right)}\right)$$

où R est le rayon de la Terre (environ 6371 km), u_{lat}, v_{lat} sont les latitudes (en radians) et u_{lon}, v_{lon} sont les longitudes (en radians). Cette définition découle de la formule de haversine [9, 14].

Pour l'exemple de la TOC avec un critère de proximité $x = 500$, T est composé de 4 arcs :

$T = \{$
`("Toronto Nord", "Toronto Sud"),`
`("Toronto Sud", "Toronto Nord"),`
`("Ottawa Nord", "Ottawa Sud"),`
`("Ottawa Sud", "Ottawa Nord")`
$\}$

car les gares Toronto Nord et Toronto Sud sont séparés par une distance de 400 mètres et Ottawa Nord et Ottawa Sud par 50 mètres (données géographiques de `src/main/resources/TOC_GTFS/TOC/stops.csv`).

Le choix de x est crucial : x est directement proportionnel à la taille de T (complexités temporelles et spatiales impliquées) comme le montre la figure 4 analysant le nombre de transferts par rapport au critère de proximité sur les 60.000 gares des données GTFS belges, un très grand graphe. Un x trop grand tuera la mémoire et le temps du programme tandis qu'un x trop petit éliminera des meilleurs chemins avec une trop grande distance à pied. Nous avons choisi le compromis $x = 500m$ comme valeur par défaut du programme mais l'utilisateur est libre de la redéfinir (voir README).

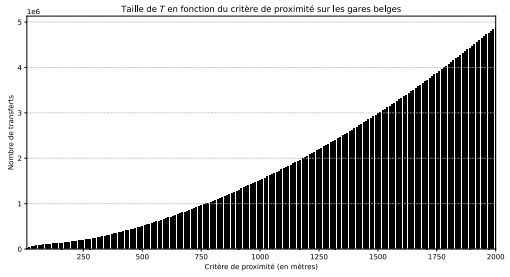

FIGURE 4 – Analyse de la taille de T en fonction de x sur les gares belges

Pour construire T, plusieurs choix de modélisation s'offrent à nous.

Approche naïve

La première approche qui vient à l'esprit est d'itérer sur toutes les paires possibles de gares et d'ajouter un arc de transfert dont le poids correspond au temps de marche, en prenant en compte le critère de proximité.

Algorithme 1 : Évaluation des transferts à pied par force brute

Input : V : ensemble de gares, x : critère de proximité
Output : T : ensemble des transferts respectant le critère de proximité

```
T ← ∅ ;
foreach u ∈ V do
    foreach v ∈ V do
        if u = v or d_(u,v) > x then
            continue
        weight ← walktime(u, v) ;
        T.add(Edge(u, v, weight)) ;
return T ;
```

L'algorithme 1 présente cette approche. La complexité temporelle de celle-ci est en $O(|V|^2)$ (ce qui est peu souhaitable).

Approche avancée

L'approche avancée utilise un arbre STR qui permet la recherche de gares à proximité en temps logarithmique amortie. Un arbre STR (STR tree) est un arbre R [10] qui partitionne les gares dans le plan en utilisant l'algorithme Sort-Tile-Recursive [11] pour emballer l'arbre (voir annexe A). Si nous fixons le nombre de gares par nœud de l'arbre à 10, cette structure de donnée offre les opérations :

- Construction en $O(|V|log_{10}|V|)$
- Recherche des voisins dans un rayon de x mètres en moyenne en $O(log_{10}|V| + \bar{t}_x)$ et dans le pire des cas en $O(|V|)$

avec $\bar{t}_x$ le nombre moyen de gares dans un rayon de x mètres autour d'une gare donnée (ce paramètre est directement proportionnel à x et à la densité géographique de la zone).

Algorithme 2 : Évaluation des transferts à pied à l'aide d'un arbre STR

Input : V : ensemble de gares, x : critère de proximité
Output : T : ensemble des transferts respectant le critère de proximité

```
T ← ∅ ;
tree ← STRtree(V) ;
foreach u ∈ V do
    neighbours ← tree.query(u, x) ;
    foreach v ∈ neighbours do
        weight ← walktime(u, v) ;
        T.add(Edge(u, v, weight)) ;
return T ;
```

L'algorithme 2 construit T à l'aide d'un arbre STR. La complexité de celui-ci est en moyenne en $O(|V|\bar{t}_x + |V|log|V|)$ et dans le pire cas en $O(|V|^2)$. Cette alternative est bien plus efficace que l'approche naïve.

Estimation du temps de marche

Nous avons donc la distance à vol d'oiseau entre deux gares. Mais comment pouvons-nous déterminer la distance à pied ? Les humains n'ont pas été conçus pour traverser les murs. Certains obstacles sur le chemin doivent être contournés. Dans les entrées du problème nous avons seulement accès aux coordonnées géographiques des gares. Nous estimons donc la distance réelle $w_{(u,v)}$ entre deux gares u et v à partir de la distance à vol d'oiseau $d_{(u,v)}$:

$$w_{(u,v)} = \frac{\pi}{2} \times d_{(u,v)}$$

Cette formule vient du mouvement naturel de contournement d'un obstacle : un arc de cercle autour de celui-ci :

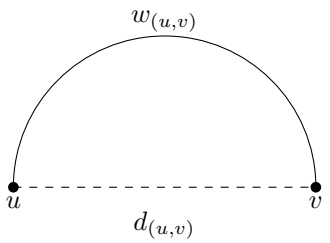

FIGURE 5 – Représentation d'un contournement d'obstacle à pied

Cette distance semble donc être une borne maximale qui estime l'impact qu'a un obstacle sur le trajet à pied d'un utilisateur.
La vitesse moyenne à pied ($human\_speed$) est de 5km/h [13]. Nous ajoutons également 90 secondes de marge de sécurité ($safety\_padding$) pour descendre et monter des transports. Pour obtenir le poids des arcs de transfert, nous appliquons la formule :

$$t_{(u,v)} = w_{(u,v)} \times human\_speed + safety\_padding$$

Comment l'algorithme fonctionne

L'algorithme qui cherche le meilleur chemin est, comme dit plus haut, une variante de TD-Dijkstra. Il peut exister plusieurs nœuds sources et plusieurs nœuds cibles. Nous retournerons le meilleur chemin du nœud u à v où u.name $= s$ et v.name $= t$.
Dans les pseudo-codes de cette section, nous stockons le coût, l'heure d'arrivée et le dernier arc du meilleur chemin jusque v dans v.best_cost, v.best_time et v.best_edge.
La raison pour laquelle les coûts sont des couples d'entier et la présentation de la fonction de coût se trouve dans la sous-section "Variantes".

Dans l'algorithme TD-Dijkstra 3 (qui n'est qu'une pâle copie de l'algorithme de Dijkstra), nous initialisons un tas binaire avec un coût infini pour tous les nœuds sauf les nœuds sources qui ont un coût nul. Le but est de piocher dans le tas (heap.delMin) chaque nœud qui a son chemin de coût minimal et pour celui-ci, nous

Algorithme 3 : TD-Dijkstra

```
Input : G : un graphe, s : nom de la source, t :
        nom de la cible, h : heure de départ
Output : le chemin le moins coûteux de s à t
         dès h
heap ← ∅ ;
foreach u ∈ V do
    if u.name = s then
        u.best_cost ← (0, 0) ;
        heap.insert(u, (0, 0)) ;
    else
        u.best_cost ← (∞, ∞) ;
        heap.insert(u, (∞, ∞)) ;
while heap.size > 0 do
    u ← heap.delMin() ;
    if u.name = t then
        return reconstruct_solution(u) ;
    else if u.cost = ∞ then
        return ∅ ;
    relaxe(u, heap) ;
```

vérifions s'il n'est pas la cible ou de coût infini. Si un nœud pioché est la cible : nous venons de trouver le chemin au moindre coût de la source jusqu'à la cible (principe de Dijkstra one-to-one). Si le chemin d'un nœud a un coût infini, tous les plus courts chemins dans le graphe ont été déterminés, nous pouvons donc arrêter l'algorithme (aucune solution trouvée). Pour chaque nœud pioché dans le tas, nous améliorons le coût de tous les voisins (Relaxation).

Relaxation

Algorithme 4 : Relaxation

```
Routine relaxe(v, heap)
    foreach e arc sortant de v do
        candidate ← function_cost(v, e) ;
        if candidate < e.to.best_cost then
            e.to.best_cost ← candidate ;
            e.to.best_edge ← e ;
            e.to.best_time
              ← v.best_time+W_{v,e}+e.duration ;
            heap.decreaseKey(e.to, candidate) ;
```

La relaxation d'un nœud v évalue le coût passant par v pour chaque voisin u (par transfert et par connexion). Si ce coût s'avère être meilleur que celui de l'actuel meilleur chemin jusque u, nous mettons à jour u.best_cost, u.best_time, u.best_edge et la position de u dans le tas (heap.decreaseKey) avec les données du chemin menant jusque v et de l'arc $e : (v, u)$. Pour rappel, vu qu'on considère le temps d'attente positif dans sa définition, tous les voisins doivent être traités, même ceux dont l'heure de départ de la connexion part avant l'heure d'arrivée du chemin jusque u.

Reconstruction de la solution

Algorithme 5 : Reconstruction du chemin

```
Routine reconstruct_solution(s, v)
    path ← ∅ ;
    current ← t.best_predecessor ;
    while ∃current do
        path.add(current) ;
        if (current.from = s) break ;
        current ←
         current.from.best_predecessor ;
    return path
```

Une fois qu'un nœud cible a été pioché du tas, l'algorithme reconstruit le chemin de s à t. Il effectue un parcours de la liste chaînée formée par les prédécesseurs stockés dans v.best_edge depuis la cible jusqu'à atteindre la première source rencontrée sur le chemin.

Variantes du problème

Algorithme 6 : Fonction de coût

```
Routine cost_function(v, e)
    d_{v+e} ← v.best_time + W_{(v,e)} + e.duration ×
     weight_{e.type} ;
    if e.trip != v.best_edge.trip then
        n_{v+e} ← n_P + 1 ;
    else
        n_{v+e} ← n_P ;
    if priority = TIME then
        return v.best_cost + (d_{v+e}, n_{v+e}) ;
    else if priority = TRIPS then
        return v.best_cost + (n_{v+e}, d_{v+e}) ;
```

Nous avons travaillé sur la généricité de l'algorithme afin d'offrir à l'utilisateur certains paramètres de préférence. La fonction de coût de chemin évalue la durée totale pondérée par mode de transport du chemin et le nombre des changements de trajet.

Définition 9 (Durée pondérée d'un arc)**.** La durée pondérée d'un arc est la durée de celui-ci multipliée par $weight_{e.type}$, une valeur strictement positive choisie par l'utilisateur en fonction de ses préférences de mode de transport :

$$d_e = e.duration \times weight_{e.type}$$

Définition 10 (Durée pondérée d'un nœud)**.** La durée pondérée d'un chemin P jusqu'à un nœud v est la somme des durées pondérées des arcs constitutifs (avec e_{from} la gare de départ d'un arc e) :

$$d_v = d_P = \sum_{e \in P} (d_e + W_{(e_{from}, e)})$$

Définition 11 (Nombre de changements de trajet)**.** Le nombre de changements de trajet d'un chemin n_P est le nombre de fois où l'utilisateur doit entrer dans un nouveau transport ou entamer une nouvelle marche à pied.

Dès lors, le coût d'un chemin P que nous cherchons à minimiser est :

- le tuple (d_P, n_P) si l'utilisateur veut minimiser en priorité la durée pondérée totale.
- le tuple (n_P, d_P) si l'utilisateur veut minimiser en priorité le nombre de changements de trajet.

Nous stockons le coût composé de ces deux critères dans un couple. Comparer deux coûts revient à d'abord comparer le premier élément puis le second : le premier critère est plus important que le deuxième. Additionner deux coûts revient à additionner les éléments entre eux. Pour rajouter un arc e au chemin P jusqu'à la gare v, nous avons d_{P+e} et n_{P+e} :

$$d_{P+e} = d_P + W_{(v,e)} + d_e$$

$$n_{P+e} = \begin{cases} n_P & \text{si } e.\text{trip} = v.\text{predecessor.trip} \\ n_P + 1 & \text{sinon} \end{cases}$$

Lemme 1. *La fonction de coût cost_function(v, e) qui calcule le coût du chemin $P + e$ est supérieure ou égale au coût de P (v.best_cost) : $n_{P+e} \geq n_P$ et $d_{P+e} \geq d_P$.*

Pourquoi l'algorithme fonctionne

La démonstration de correction ci-dessous est issue du syllabus INFOF203 24-25 de l'Université libre de Bruxelles [3].

Lemme 2. *L'algorithme de TD-Dijkstra de s à t dès h dans le graphe temporel $G(V, A, T)$ pondéré positif retourne le chemin P qui minimise le coût (défini par n_P et d_P) parmi tous les chemins valides de s à t.*

Prouver que quand TD-Dijkstra pioche t, t.best_cost est le coût du meilleur chemin de s à t revient à prouver que lorsqu'il pioche un nœud v du tas, pour tout nœud u pioché jusque v (compris), u.best_cost est le coût du meilleur chemin de s à u.

Démonstration par induction.

Le cas de base est vérifié : nous initialisons s.best_cost à (0, 0). Quand s est pioché du tas : la distance de s à s est 0 et le nombre de changements de trajet dans un chemin vide est nul. Pour l'induction : supposons qu'il existe un autre chemin de s à v moins coûteux passant par au moins un nœud w non-pioché.

Si l'on pose w comme le premier nœud qui diffère entre les deux chemins, cela implique que tous les nœuds avant w ont été piochés et que le coût du chemin de s à w est inférieur à celui du chemin de s à v.

Par le lemme 2, il en découle que w est pioché avant v. Ce qui n'est pas sensé être possible. □

Exemple de la TOC

Revenons à notre exemple sur le graphe de la TOC 3.

Premièrement, nous initialisons le tas binaire sachant que $s =$ "Québec" et que le coût est (d_P, n_P) (donc on priorise la durée totale du chemin) :

heap =

Québec	*
(0, 0)	(∞, ∞)

(Notez que * : (∞, ∞) signifie que le reste des gares du tas ont un poids infini)
#1 Nœud pioché : Québec
relaxation de ses voisins : Toronto Nord et Ottawa Sud

Toronto Nord	Ottawa Sud	*
(45min, 1)	(60min, 1)	(∞, ∞)

#2 Nœud pioché : Toronto Nord
relaxation de ses voisins : Toronto Sud

Toronto Sud	Ottawa Sud	*
(51min, 2)	(60min, 1)	(∞, ∞)

#3 Nœud pioché : Toronto Sud
relaxation de ses voisins : Toronto Nord et Calgary

Ottawa Sud	Calgary	*
(60min, 1)	(180min, 3)	(∞, ∞)

#4 Nœud pioché : Ottawa Sud
relaxation de ses voisins : Ottawa Nord

Ottawa Nord	Calgary	*
(62min, 2)	(180min, 3)	(∞, ∞)

#5 Nœud pioché : Ottawa Nord
relaxation de ses voisins : Ottawa Sud et Trois-Rivières

Trois-Rivières	Calgary	*
(85min, 3)	(180min, 3)	(∞, ∞)

#6 Nœud pioché : TR (Trois-Rivières)
relaxation de ses voisins : Halifax

Halifax	Calgary	*
(120min, 3)	(180min, 3)	(∞, ∞)

#7 Nœud pioché : Halifax
relaxation de ses voisins : Montréal

Montréal	Calgary	*
(165min, 3)	(180min, 3)	(∞, ∞)

#8 Nœud pioché : Montréal
Cible trouvée !

- Durée totale du chemin : 2 heures et 45 minutes (arrivée à 11:30:00)
- Nombre de trajets : 3 (TRAIN A → chemin à pied → TRAIN B)

3 Implémentation

Nous avons donné le nom `algochoochoo` au projet Java.

Packets Java

- `algochoochoo.cli` lance le CLI.
- `algochoochoo.gui` lance la GUI.
- `algochoochoo.graph` construit le graphe.
- `algochoochoo.query` exécute l'algorithme.
- `algochoochoo.parsing` charge et lit les données GTFS.

Classes principales

Les classes les plus importantes du programme sont listées ci-dessous. Un descriptif complet est disponible dans la javadoc du projet (voir README) et l'annexe B.

- `Parser` : Module se chargant de lire les fichiers CSV et de les transformer en tables de trajets (Parser.trips) et de gares (Parser.stops)
- `Node` : Un nœud du graphe qui garde les informations d'une gare lors de l'algorithme de Dijkstra (coût du chemin, etc.)
- `Edge` : Un arc (transfert ou connexion) du graphe avec ses informations (route, trajet, durée, etc.)
- `Graph` : Le graphe $G(V, A, T)$. Construction de V à partir de Parser.stops, de A à partir de Parser.trips et de T à l'aide d'un arbre STR temporaire.
- `Algorithm` : Module d'exécution d'une requête du plus court chemin. Algorithm.dijkstra lance l'algorithme de TD-Dijkstra à l'aide des méthodes Algorithm.relaxe et Algorithm.build_solution. Le lancement du CLI est contenu dans la méthode `main` de la classe.
- `cli.View et gui.View` : Modules qui gèrent l'affichage (sur cli ou sur gui) du programme.
- `AlgoSettings` : Stockage des préférences utilisateur sur l'exécution de l'algorithme pour les variances (poids, priorité, etc.). Cette classe est responsable du calcul du coût des chemins (AlgoSettings.cost_function).

- `GraphSettings` : Stockage des préférences utilisateur sur la construction du graphe (critère de proximité, etc.)

Chemin de l'information

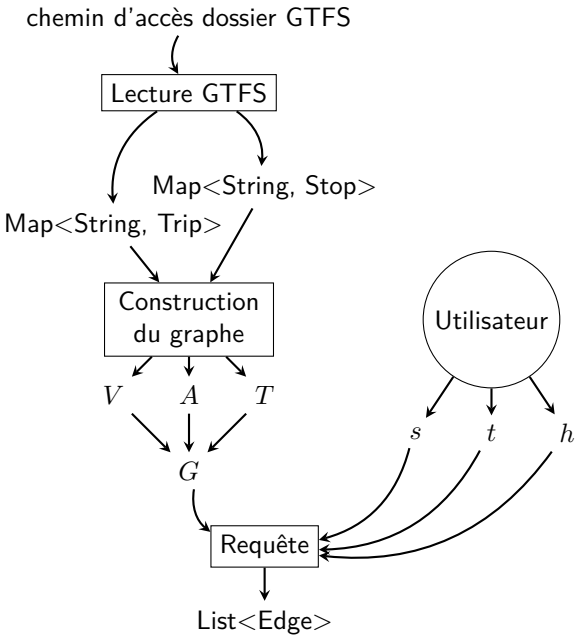

FIGURE 6 – Cheminement de l'information dans le programme Java

Convertir le contenu de stop_times.csv en trajets

L'une des façons les plus optimales de construire la table des trajets revient à rajouter chaque ligne de stop_times.csv dans son trajet (sans se préoccuper de la séquence) et ensuite de trier chaque trajet de la table (par `stop_sequence`). Ainsi, on obtient les gares consécutives dans les trajets, qui formeront des arcs de connexion dans le graphe G.

Stockage du coût

Dans ce rapport, nous définissons le coût comme un couple d'entiers. En java, nous représentons cela par une variable de type `long` où les 4 premiers octets stockent le premier entier du couple et les 4 suivants le second :

n_P				d_P			
0	1	2	3	4	5	6	7

FIGURE 7 – Stockage d'un coût (n_P, d_P) d'un chemin en Java

d_P et toutes les durées et heures du programme sont des entiers : la durée/heure en secondes.

4 Analyse de la complexité

Complexité temporelle

Pour rappel nous avons déjà mis un nom sur les données suivantes :

- V L'ensemble de gares.
- A L'ensemble de connexions.
- T L'ensemble de transferts.
- x Le critère de proximité.
- $\bar{t}_x$ Le nombre moyen de gares dans un rayon de x mètres autour d'une gare.

Et nous définissons par la même occasion :

- R L'ensemble de routes.
- F L'ensemble de trajets.

Lecture GTFS

Cette opération revient à parcourir et stocker les gares, routes, trajets et connexions, ainsi qu'à trier le contenu des trajets :
Parcourir les gares : $O(|V|)$.
Parcourir les routes : $O(|R|)$.
Parcourir les trajets : $O(|F|)$.
Parcourir les connexions : $O(|A|)$.
Trier le contenu de $|F|$ trajets qui ont en moyenne une taille de $\frac{|A|}{|F|}$: $O(|F| \times \frac{|A|}{|F|} log \frac{|A|}{|F|}) = O(|A| log \frac{|A|}{|F|})$.

En sommant le tout, nous avons :

$$O(|V| + |R| + |F| + |A| log \frac{|A|}{|F|})$$

Construction du graphe

Cette opération revient à créer V, T et A :
Créer les nœuds du graphe : $O(|V|)$.
Convertir les trajets en connexions : $O(|F| \times \frac{|A|}{|F|}) = O(|A|)$.
Évaluer les transferts à l'aide d'un arbre STR (voir 2) : $O(|V|\bar{t}_x + |V| log |V|)$.

Ce qui nous donne :

$$O(\bar{t}_x |V| + |V| log |V| + |A|)$$

Requête

TD-Dijsktra utilise un tas binaire de taille $|V|$ qui offre les opérations :

- `heap.insert` en O($log|V|$)
- `heap.delMin` en O($log|V|$)
- `heap.decreaseKey` en O($log|V|$)

L'initialisation du tas se fait en $O(|V|log|V|)$ (nous avons conscience qu'il est possible de construire un tas binaire en $O(|V|)$ mais l'implémentation du tas binaire d'algs4 que nous utilisons ne l'offre pas).
Dans le pire des cas, nous vidons l'entièreté du tas binaire (heap.delMin) en $O(|V|log|V|)$ et effectuons une relaxation sur chaque nœud. Une relaxation (avec heap.decreaseKey) sur chaque nœud du graphe implique qu'on visite tous les arcs de celui-ci, donc, en $O((|T|+|A|)log|V|)$.
L'algorithme s'exécute en :

$$O((|A|+|T|+|V|)log|V|)$$

dans notre cas car nous utilisons un tas binaire. Il est également possible d'utiliser un tas de Fibonacci [6] qui réduit la complexité à :

$$O(|V|log|V|+|A|+|T|)$$

Nous n'avons pas implémenté de tas de Fibonacci dans notre programme Java dû au manque de praticité de la structure.

Totale

L'exécution des 3 étapes ci-dessus se fera en :

$$O(|R|+|F|+|A|log\frac{|A|}{|F|}+|V|\bar{t}_x+(|A|+|T|+|V|)log|V|)$$

Complexité spatiale

Lecture GTFS

La lecture qui convertit des données GTFS en deux tables de hachage : les trajets remplis ($|F|$ trajets stockant les arcs sous forme de tableau) et les gares.

Un trajet de n gares sera converti en $n-1$ arcs dans le graphe, donc $|F|$ trajets stockant en moyenne n gares (tel que $|F|\times(n-1)=|A|$) a une complexité spatiale de $O(|A|-|F|)$.

Ce module utilise un arbre STR contenant $|V|$ éléments.

La complexité est donc de $O(|V|+|A|-|F|)$.

Graphe

Comme dit dans la solution, le graphe est représenté par une liste d'adjacence. La classe Java `Graph` stocke un tableau de nœuds (classe `Node`) qui eux-mêmes stockent des tableaux pour les arcs de transfert et de connexion sortant. La complexité du graphe est donc :

$$O(|V|+|A|+|T|)$$

Requête

La requête se sert d'un tas binaire d'une taille maximale de $|V|$ éléments.

Totale

$$O(|V|+|A|+|T|-|F|)$$

Comparaison avec d'autres algorithmes

Connection Scan Algorithm

Le *Connection scan algorithm* (CSA) [4] est un algorithme de recherche du plus court chemin dans un graphe dépendant du temps qui se base sur une représentation du graphe par une liste triée des connexions triées par heure de départ et qui prône une recherche en temps linéaire. Nous avons décidé de ne pas travailler avec celui-ci par manque de généricité de l'algorithme (très difficile d'y inclure d'autres fonctions de coût que le temps total du chemin) et à cause des chemins à pied. En effet, implémenter les transferts dans le CSA correctement demande de nombreuses connaissances et algorithmes qui dépassent le cadre de ce projet. Une implémentation naïve du CSA à laquelle nous avons réfléchi viendrait à relaxer les transferts lors d'une étape de l'itération des connexions si celle-ci améliore le coût pour atteindre un nœud. Or, les graphes issus des données GTFS sont des multigraphes (donc le nombre de connexions peut s'avérer bien plus grand que le nombre de nœuds), et avec une relaxation des transferts pour chaque étape de l'itération, la complexité temporelle d'une recherche s'estimerait en $O(|A|\bar{t}_x)$, ce qui semble peu enviable pour ce type de graphe et/ou un critère de proximité élevé.

Bellman-Ford

L'algorithme de *Bellman-Ford* [1, 8] n'est pas adéquat pour ce contexte car les arcs sont de poids positifs (donc pas besoin de gérer les cycles absorbants) et la complexité de la recherche dans le pire des cas est en $O(|V|(|A|+|T|))$.

A*

L'algorithme *A** [7] est une amélioration directe de l'algorithme de Dijkstra one-to-one à l'aide d'une heuristique pour diriger le développement vers le nœud cible afin de visiter le moins de nœuds possibles. Il est essentiel pour l'admissibilité de A* que l'heuristique soit elle-même admissible (c'est-à-dire qu'elle surestime toujours le véritable coût pour se rendre au nœud cible). Trouver et calculer une heuristique admissible assez intéressante pour gagner en temps d'exécution sur Dijkstra n'est pas une tâche facile, mais nous gardons en tête cet algorithme s'il nous vient un jour l'occasion d'améliorer le contenu de ce projet.

5 Librairies externes utilisées

Un guide d'installation est présent dans le fichier README. Ce projet Java utilise le gestionnaire de projet Maven afin de faciliter l'utilisation des librairies externes

et d'éviter les problèmes de portabilité. Nous comptons comme librairies externes :

- **univocity-parsers** offrant des outils de parcours de fichiers CSV rapides.
- **algs4** offrant l'implémentation du tas binaire (`IndexMinPQ`) utilisé pour l'algorithme de TD-Dijkstra.
- **JTS** offrant l'implémentation de l'arbre STR (`STRtree`) utilisé pour l'algorithme d'évaluation des transferts.
- **JavaFx, Swing et JXMapViewer2** pour la GUI.

6 Conclusion

Plusieurs défis ont été relevés dans ce projet. Il fallait correctement analyser les données GTFS et stocker celles-ci dans un graphe G qui offre une bonne complexité spatiale et temporelle pour les requêtes de plus court chemin. Nous avons également réfléchi sur la façon optimale d'évaluer les chemins à pied entre les gares, comment les considérer dans l'algorithme de Dijkstra. La généricité de l'algorithme était aussi un autre point important, laissant à l'utilisateur des options pour définir ses préférences de mode de transport et de distance de marche. La théorie des graphes et de recherche du plus court chemin reste un domaine en évolution constante. D'autres algorithmes récents (CSA, 2017) et moins récents (A*, 1968) ont été considérés dans l'étape de pensée du projet, mais c'est l'algorithme de Dijkstra qui est ressorti comme l'un des meilleurs pour le contexte du problème.

Références

[1] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16 :87–90, 1958.

[2] Raymond T. Boute. The euclidean definition of the functions div and mod. *ACM Trans. Program. Lang. Syst.*, 14(2) :127144, April 1992.

[3] Jean Cardinal. *Algorithmique 2*. Université libre de Bruxelles (ULB), Bruxelles, Belgique, 2024-2025. Syllabus de cours, pages 75–77.

[4] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Connection scan algorithm. *CoRR*, abs/1703.05997, 2017.

[5] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1 :269–271, 1959.

[6] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3) :596615, July 1987.

[7] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2) :100–107, 1968.

[8] Lester R. Ford Jr. Network flow theory. Technical report, RAND Corporation, Santa Monica, California, aug 1956.

[9] Charles F. F. Karney. Algorithms for geodesics. *Journal of Geodesy*, 87(1) :43–55, 2013.

[10] Tobin J. Lehman and Michael J. Carey. A study of index structures for main memory database management systems. In *Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 294–303, 1986.

[11] Scott Leutenegger, Mario Lopez, and Jeffrey Edgington. Str : A simple and efficient algorithm for r-tree packing. pages 497–506, 05 1997.

[12] MobilityData. Plate-forme centrale de documentation pour la general transit feed specification. `https://gtfs.org`, 2025. Consulté le 5 mai 2025.

[13] H.J. Ralston. Energy-speed relation and optimal speed during level walking. *International Zeitschrift für Angewandte Physiologie Einschließlich Arbeitsphysiologie*, 17 :277–283, 1958.

[14] Chris Veness. Great-circle distance between two points. Movable Type Scripts, 2011. Consulté le 15 mai 2025.

[15] Y. Wang, Y. Yuan, Y. Ma, and et al. Time-dependent graphs : Definitions, applications, and algorithms. *Data Sci. Eng.*, 4 :352–366, 2019.

A Algorithme STR et arbre R

Nous utilisons l'implémentation de l'arbre STR de la `Java Suite Topology`. Sachant que nous ne l'avons pas écrit nous-même en Java, il est important d'en expliquer l'implémentation interne. Nous nous référençons tout au long de cette annexe sur les travaux effectués par Scott T. Leutenegger, Jeffrey M. Edgington et Mario A. Lopez en 1997 sur les arbres R et l'algorithme STR [11].

A.1 Définition de l'arbre R

Un arbre R est une amélioration de l'arbre B en 2D (ou plus si besoin) qui accélère les requêtes d'intersection. Il collectionne un ensemble de r rectangles dans le plan qui englobe chacun un maximum de m entrées. Une entrée est caractérisée par un rectangle R et P, une gare si c'est une feuille, sinon un nœud fils. Le rectangle R d'un nœud de l'arbre est le MBR (Minimum Bounding Rectangle) du sous-arbre du nœud (c'est-à-dire le plus petit rectangle englobant la gare si c'est une feuille ou toutes les gares de toutes les feuilles du sous-arbre sinon).

A.2 Algorithme STR

L'algorithme `Sort-Tile-Recursive` est une routine d'emballage (packing) de r rectangles dans un arbre R. Dans notre utilisation, les r rectangles sont des rectangles longueur et largeur nulle sur la position de chaque gare. Le défi lors de l'emballage est d'offrir un arbre R balancé. Un arbre balancé offre un parcours récursif en $\Theta(log(r))$, tandis qu'un arbre non balancé risque dans le pire des cas un parcours en $O(r)$. L'algorithme passe par les étapes :

- Calculer le nombre de feuilles : $P = \lceil \frac{r}{m} \rceil$.
- Définir le nombre de tranches verticales $S = \sqrt{P}$.
- Trier les rectangles par rapport à leur position x (ou longitude si gare) et partitionner le plan en S tranches verticales composées de S éléments.
- Pour chaque tranche : Trier les rectangles par rapport à leur position y (ou latitude si gare) et regrouper les rectangles consécutifs par groupe de m entrées. Chaque regroupement représente un nœud de l'arbre R à une hauteur donnée.

Si le nombre de nœud résultant de cet algorithme excède m, nous réeffectuons l'algorithme mais cette fois-ci avec les MBR de chaque nœud. C'est ainsi qu'on construit un arbre R en commençant avec les feuilles et en remontant jusqu'à la racine.
La routine stops_to_entries utilisée dans le pseudo-code 8 convertit une liste de gares en une liste d'entrée à rectangle de taille nulle sur la position de chaque gare. La routine MBR dans le pseudo-code 7 détermine le minimum bounding rectangle d'un ensemble de nœuds de l'arbre.

Algorithme 7 : Sort-Tile-Recursive

Input : R : un ensemble de r nœuds de l'arbre R à même hauteur, m : nombre d'entrées max. par nœud
Output : $result$: les nœuds parents regroupant les nœuds dans R
Routine *STR(R, m)*

```
result ← ∅ ;
P = ⌈r/m⌉ ;
S = √P ;
sort(R, key : MBR.X) ;
foreach R_x vertical slice of R of max. S × m rectangles do
    sort(R_x, key : MBR.Y) ;
    foreach R_y horizontal slice of R_x of max. m rectangles do
        node ← Node(MBR(R_y), R_y) ;
        result.add(node) ;
return result ;
```

Algorithme 8 : Construction d'un arbre R avec STR

Input : V : ensemble non-nulle de gares
Output : $root$: racine de l'arbre R
Routine *STRtree.build(V)*

```
m ← 10 ;
entries ←stops_to_entries(V) ;
while entries.size > 1 do
    entries ←STR(entries, m) ;
return entries[0] ;
```

A.3 Requête sur l'arbre R

Une requête brute sur l'arbre R demande toutes les gares dans un rectangle Q. Lors de celle-ci, nous devons parcourir tous les rectangles qui chevauchent Q. Cela revient simplement à effectuer une recherche récursive dans l'arbre en commençant à la racine. Pour chaque nœud que l'on visite :

- Le nœud est une feuille : on ajoute la gare P au résultat.
- Le nœud est interne : on effectue un appel récursif sur tous les sous-nœuds P dont R chevauche la requête Q.

Mais, lors de notre algorithme, nous ne recherchons pas les gares dans un rectangles Q mais dans le cercle de rayon x (critère de proximité) autour d'une gare. Pour régler ceci, nous effectuons premièrement une requête rectangulaire et ensuite un filtrage des voisins trouvés qui se trouvent bel et bien dans le cercle.

Algorithme 9 : Recherche rectangulaire dans l'arbre R

```
Routine STRtree.rect_query(node, Q)
    result ← ∅ ;
    foreach (R, P) ∈ node do
        if R ∩ Q then
            if node.is_leaf then
                result.add(P) ;
            else
                result.add(STRtree.query(P,
                  Q)) ;
    return result ;
```

Algorithme 10 : Recherche de rayon dans l'arbre R

```
Routine STRtree.query(u, x)
    result ← ∅ ;
    rect_result ← STRtree.rect_query(root,
      [u_lon − x, u_lon + x], [u_lat − x, u_lat + x]) ;
    foreach v ∈ result do
        if dist(v, u) < x then
            results.add(v) ;
    return result ;
```

Notez que le pseudo-code 10 est simplifié au niveau des distances, qui doivent être adaptées normalement selon la courbure de la terre (avec la formule d'haversine).

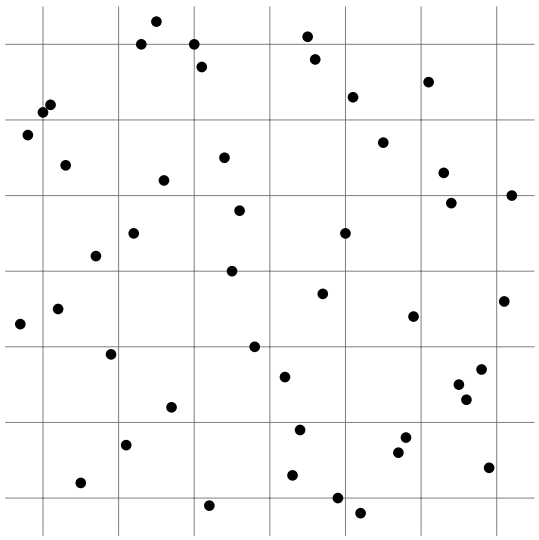

FIGURE 8 – Plan d'exemple contenant 45 gares

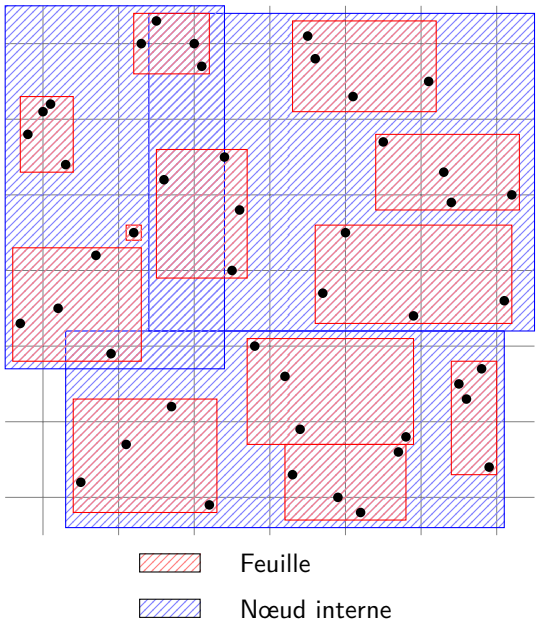

FIGURE 9 – Partitionnement balancé des 45 gares dans un arbre R ($m = 4$)

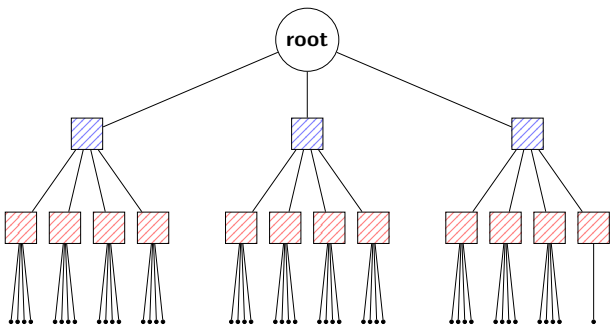

FIGURE 10 – Arbre R de la hiérarchie des rectangles de la figure 9

La figure 11 montre que la recherche dans un carré au-

tour d'une gare G dans notre plan d'exemple compte 7 résultats, tandis que la figure 12 indique que la recherche dans un rayon n'en garde que 6.

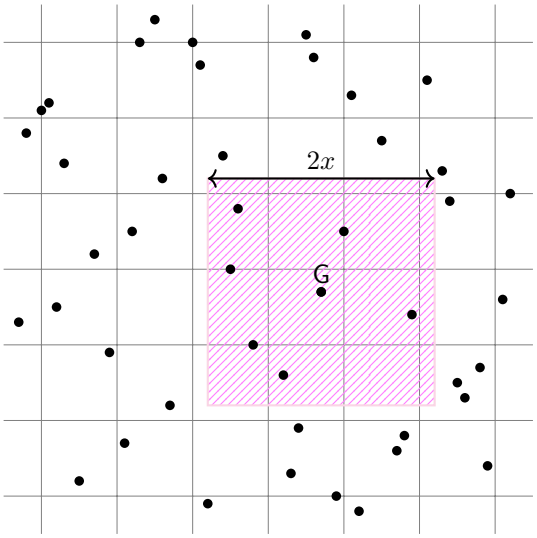

FIGURE 11 – Recherche rectangulaire autour de G

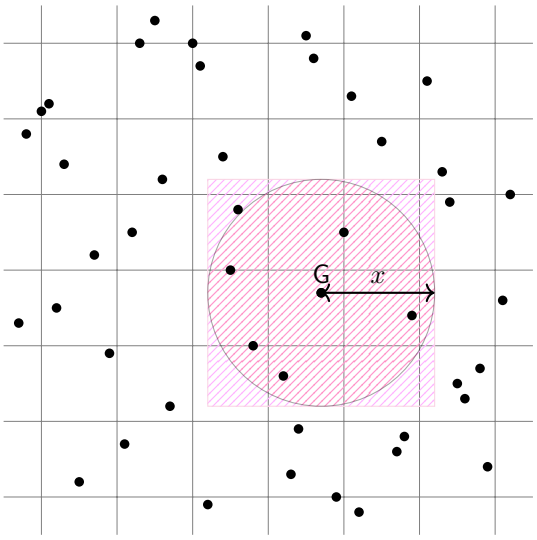

FIGURE 12 – Recherche de rayon x autour de G

A.4 Complexité spatiale d'un arbre R

Pour stocker $|V|$ gares, l'arbre R compte $\frac{|V|}{m}$ feuilles, $\frac{|V|}{m^2}$ nœuds parents de celle-ci et ainsi de suite jusqu'à la racine. avec $m = 10$, la complexité spatiale de l'arbre s'estime en $O(|V|)$.

B Description complète des classes Java

Cette annexe contient la description complète des classes, méthodes et attributs publiques du code Java.

graph.GraphSettings

```java
package algochoochoo.graph;

/**
 * Stocke les préférences utilisateur pour
 * la construction du graphe
 */
public class GraphSettings {
  /**
   * Construit un GraphSettings par défaut
   */
  GraphSettings();

  /**
   * Construit un GraphSettings depuis le
       chemin
   * d'accès du dossier GTFS
   */
  GraphSettings(String gtfs_path);

  /**
   * Construit un GraphSettings depuis les
   * arguments de l'exécutable (CLI)
   */
  GraphSettings(String[] args);

  /**
   * Affiche les paramètres sur STDOUT
   */
  void print();

  /** Le chemin d'accès des données GTFS */
  String GTFS_path = "src/main/resources/GTFS";

  /** Le critère de proximité */
  int foot_radius = 500;
}
```

Listing 1 – Classe graph.GraphSettings

graph.Graph

```java
package algochoochoo.graph;

/**
 * Un graphe
 */
public class Graph {
  /**
   * Construit le graphe à partir de
   * paramètres utilisateur
   */
  Graph(GraphSettings set);

  /** Convertit un graphe en String */
  @Override
  String toString();

  /**
   * Reconstruit le graphe à partir de
   * nouveaux paramètres
   */
  void reload(GraphSettings new_settings);

  /**
   * Obtient la cardinalité de
   * l'ensemble d'arcs
   */
  int E_card();

  /**
   * Obtient la cardinalité de
   * l'ensemble d'arcs de transfert
   */
  int T_card();
```

```
34
35    /**
36     * Obtient la cardinalité de
37     * l'ensemble d'arcs de connexion
38     */
39    int A_card();
40
41    /**
42     * Obtient la cardinalité de
43     * l'ensemble de n\oe uds
44     */
45    int V_card();
46
47    /**
48     * Obtient la liste d'adjacence du graphe
49     */
50    List<Node> vertices();
51
52    /**
53     * Obtient les paramètres utilisés
54     * pour la construction du graphe
55     */
56    GraphSettings settings();
57  }
```

Listing 2 – Classe graph.Graph

graph.Node

```
1   package algochoochoo.graph;
2
3   /**
4    * Un n\oe ud du graphe
5    */
6   public class Node {
7     /**
8      * Construit un n\oe ud vide depuis
9      * une gare s
10     */
11    Node(Stop s);
12
13    /** Convertit un n\oe ud en String */
14    @Override
15    String toString();
16
17    /**
18     * (TD-Dijkstra) Initialise le n\oe ud
19     * pour une nouvelle requete
20     * meilleur coût: INF
21     * meilleur temps: INF
22     * meilleur arc: null
23     */
24    void init();
25
26    /**
27     * Obtient la gare du n\oe ud
28     */
29    Stop stop();
30
31    /**
32     * (TD-Dijkstra) Obtient l'arc menant
33     * à ce n\oe ud du meilleur chemin à l'étape
34     * actuelle de l'algorithme
35     */
36    Edge best_edge();
37
38    /**
39     * (TD-Dijkstra) Obtient le meilleur coût
40     * pour atteindre ce n\oe ud à l'étape
       actuelle de l'algorithme
41     */
42    long best_cost();
43
44    /**
45     * (TD-Dijkstra) Obtient la meilleure
46     * heure d'arrivée pour atteindre ce n\oe ud
47     * à l'étape actuelle de l'algorithme
48     */
49    int best_time();
50
51    /**
52     * Obtient les arcs de transferts sortants
53     * du n\oe uds
54     */
55    List<Edge> transfers();
56
57    /**
58     * Obtient les arcs de connexions sortants
59     * du n\oe uds
60     */
61    List<Edge> connections();
62
63    /**
64     * Obtient l'index du n\oe ud dans
65     * la liste des n\oe uds
66     * représentant le graphe
67     */
68    int index();
69
70    /**
71     * (TD-Dijkstra) Retourne si le n\oe ud est
72     * source de l'algorithme
73     */
74    boolean is_source();
75
76    /**
77     * (TD-Dijkstra) Retourne si le n\oe ud est
78     * cible de l'algorithme
79     */
80    boolean is_target();
81
82    /**
83     * (TD-Dijkstra) Améliore les variables
84     * coût, heure d'arrivée et arc du chemin
85     * après relaxation dans l'algorithme
86     */
87    void set_best(long cost, int time, Edge edge)
          ;
88
89    /**
90     * Défini la liste des transferts sortants
91     * du n\oe ud
92     */
93    void set_transfers(List<Edge> t);
94
95    /**
96     * Ajoute une connexion du n\oe ud
97     * dans sa liste de connexion sortantes
98     */
99    void add_connection(Edge c);
100
101   /**
102    * Défini l'index du n\oe ud dans la liste
103    * de n\oe ud représentant le graphe
104    * (Nécessaire pour algs4.IndexMinPQ)
105    */
106   void set_index(int i);
107
108   /**
109    * (TD-Dijkstra) Défini ce n\oe ud comme
110    * source de l'algorithme
111    */
112   void declare_source();
113
114   /**
115    * (TD-Dijkstra) Défini ce n\oe ud comme
         cible
116    * de l'algorithme
117    */
118   void declare_target();
119 }
```

Listing 3 – Classe graph.Node

graph.Edge

```java
package algochoochoo.graph;

/**
 * Un arc du graphe
 */
public class Edge {
  /**
   * Construit un arc de connexion à
   * partir de deux elements de trajet (TripElement)
   * Une map {stop_id:Node} et un trajet (Trip)
   */
  Edge(
      TripElement first, TripElement second,
          Map<String, Node> V, Trip t);

  /**
   * Construit un arc de transfert à partir de deux
   * gares et d'une distance à vol d'oiseau
   */
  Edge(Node source, Node target, int distance);

  /**
   * Construit un arc de dest purement à but
   * de comparaison
   */
  Edge(int departure);

  /** Convertit l'arc en String */
  @Override
  String toString();

  /**
   * Obtient une directive en anglais de l'arc
   * Ex: Taking BUS 71
   * (fonctionnalité CLI)
   */
  String directive();

  /**
   * Obtient la gare d'arrivée de l'arc
   */
  Node to();

  /**
   * Obtient la gare de départ de l'arc
   */
  Node from();

  /**
   * Get the trip of the edge or null if its a transfer
   *
   * @return Edge trip or null if is transfer
   */
  Trip trip();

  /**
   * Obtient le type de route de l'arc
   */
  RouteType type();

  /**
   * Obtient l'heure de départ de l'arc
   * si c'est un arc de connexion, sinon -1
   */
  int departure_time();

  /**
   * Obtient la durée de l'arc
   */
  int duration();

  /**
   * Retourne si l'arc représente
   * un transfert à pied
   */
  boolean is_transfer();
  /**
   * Retourne si l'arc représente une
   * connexion
   */
  boolean is_connection();

  /**
   * Obtient la couleur de l'arc (fonctionnalité GUI)
   */
  Color color();
}
```

Listing 4 – Classe graph.Edge

query.AlgoPriority

```java
package algochoochoo.query;

/**
 * La priorité de l'algorithme
 * (= critère à minimiser)
 */
public enum AlgoPriority {
  /** La durée totale (pondérée) du chemin */
  TOTAL_DURATION,

  /** Le nombre de changements de trajet */
  TRIP_CHANGES,
}
```

Listing 5 – Classe query.AlgoPriority

query.AlgoSettings

```java
package algochoochoo.query;

/**
 * Stocke les préférences utilisateurs
 * liées à l'exécution de l'algorithme
 */
public class AlgoSettings {
  /**
   * Construit des paramètres par défaut
   */
  AlgoSettings();

  /**
   * Construit les paramètres de l'algorithme à
   * partir des arguments de l'exécutable
   */
  AlgoSettings(String[] args);

  /**
   * Si le nom des gares sources et cibles doit être identique au
   * caractère près
   */
  boolean strict_search = true;

  /** Les poids de mode de transport de l'algorithme */
```

```
  Map<RouteType, Double> weights = new
      LinkedHashMap<>();

  /** La priorité de l'algorithme */
  AlgoPriority priority = AlgoPriority.
      TOTAL_DURATION;

  /**
   *
   */
  long cost_function(Node v, Edge e);

  /**
   * Affiche les paramètres sur STDOUT
   */
  void print();
}
```

Listing 6 – Classe query.AlgoSettings

query.Algorithm

```
package algochoochoo.query;

/**
 * Exécute l'algorithme du plus court chemin
 */
public class Algorithm {
  /**
   * Lance le CLI
   */
  public static void main(String[] args);

  /**
   * Construit un Algorithm avec un graph et
       des
   * préférences utilisateur
   */
  public Algorithm(Graph G, AlgoSettings set);

  /**
   * Obtient les préférences utilisateur
   * de l'algorithme
   */
  public AlgoSettings settings();

  /**
   * Retourne le meilleur chemin de s à t à
       partir de h
   * en exécutant TD-Dijkstra
   */
  public AlgoResult dijkstra(String s, String t
      , int h);
}
```

Listing 7 – Classe query.Algorithm

query.AlgoResult

```
package algochoochoo.query;

/**
 * Stocke le résultat d'un algorithme de
     recherche
 * du plus court chemin
 */
public class AlgoResult {

  /** Plus court chemin trouvé */
  List<Edge> path = new ArrayList<>();

  /** Nombre de noeuds visités par Dijkstra */
  int visited_vertices = 0;

  /** Temps de la requête */
  double runtime = 0;

  /** Temps de lecture GTFS et construction du
      graphe */
  double preruntime = 0;
}
```

Listing 8 – Classe query.AlgoResult

cli.View

```
package algochoochoo.cli;

/**
 * Gére l'affichage du CLI (command line
 * interface)
 */
public class View {
  /**
   * Affiche dans le terminal un résultat de
   * l'algorithme (chemin, heure de départ)
   */
  static void print(AlgoResult, int h);
}
```

Listing 9 – Classe cli.View

gui.View

```
package algochoochoo.gui;

/**
 * Gére l'affichage du GUI (graphical
 * user interface)
 */
public class View {
  /**
   * Initialise et lance le GUI
   */
  static void main(String[] args);

  /**
   * Construit un GUI
   */
  View();

  /**
   * Initialise le GUI
   */
  void set_up();

  /**
   * Lance le GUI
   */
  void run();
}
```

Listing 10 – Classe gui.View

gui.PathPainter

```
package algochoochoo.gui;


/**
 * Enfant de la classe Painter<JXMapViewer>
 * de la dépendance JXMapViewer2,
 * Gére le dessin d'un chemin sur la GUI
 */
```

```java
public class PathPainter implements Painter<
    JXMapViewer> {
  /**
   * Construit un PathPainter vide
   */
  PathPainter();

  /**
   * Construit un PathPainter avec
   * t les positions des gares
   * c Les couleurs des arcs
   * w Si les arcs sont des transferts
   */
  PathPainter(List<GeoPosition> t, List<Color>
      c, List<Boolean> w);

  @Override
  /**
   * Dessine le chemin sur le widget de
   * JXMapViewer
   */
  void paint(Graphics2D g, JXMapViewer map, int
       w, int h);
}
```

Listing 11 – Classe gui.PathPainter

gui.StopWaypoint

```java
package algochoochoo.gui;

/**
 * Enfant de la class Waypoint
 * de la dépendance JXMapViewer2,
 * Représente une gare sur la carte du GUI
 */
public class StopWaypoint implements Waypoint {
  /**
   * Construit un StopWaypoint avec une
       position,
   * couleur et si c'est une intersection
   */
  StopWaypoint(GeoPosition pos, Color c,
      boolean intersec);

  @Override
  /**
   * Obtient la position du Waypoint
   */
  GeoPosition getPosition();

  /**
   * Obtient la couleur du Waypoint
   */
  Color color();

  /**
   * Retourne vrai si le Waypoint est une
       intersection entre deux trajets
   */
  boolean is_intersection();
}
```

Listing 12 – Classe gui.StopWaypoint

gui.StopWaypointRenderer

```java
package algochoochoo.gui;

/**
 * Enfant de la classe
 * WaypointRenderer<Waypoint>
 * de la dépendance JXMapViewer2,
 * Gére l'affichage des gares sur
 * la GUI
 */
public class StopWaypointRenderer
    implements WaypointRenderer<StopWaypoint> {
  /**
   * Construit un StopWaypointRenderer
   */
  StopWaypointRenderer();

  @Override
  /**
   * Dessine une gare sur la carte
   */
  void paintWaypoint(Graphics2D g, JXMapViewer
      map, StopWaypoint waypoint);
}
```

Listing 13 – Classe gui.StopWaypointRenderer