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 et 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 important de l'algorithme sont l'optimalité et la validité de celui-ci.

Définition du problème

Soient les entrées suivantes :

- Des données GTFS représentant des réseaux de transports publiques.
- Le nom d'une gare source que nous appellerons s tout au long de ce rapport.
- Le nom d'une gare cible que nous appellerons t tout au long de ce rapport.
- Une heure de départ que nous appellerons h tout au long de ce rapport.

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. Ce choix est expliqué plus bas dans ce rapport.

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

Définition 1 (Chemin)**.** Un **chemin** est une suite de connexions et/ou de transferts.

Définition 2 (Connexion)**.** Une **connexion** est le mouvement d'un transport d'une gare à une autre caractérisée par une heure de départ et une durée.

Définition 3 (Transfert)**.** Un **transfert** est le mouvement 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 4 (Temps d'attente)**.** Le **temps d'attente** $W_{(v,e)}$ d'un arc e après un nœud v (donc e.from $= v$) 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 représente l'heure de départ de 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 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 5 (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 allegée 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-même 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 valide

routes.csv

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

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.

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
...	...	...	...

FIGURE 2 – Exemple d'un fichier `routes.csv`

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

FIGURE 3 – 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 représentent la latitude et longitude de la gare. Comme dit précedement, `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
...	...	...	...

FIGURE 4 – 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
...	...	...	...

FIGURE 5 – 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 6). Les données GTFS de cet exemple se trouve dans le dossier `src/test/resources/GTFS`. Nous posons donc :

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

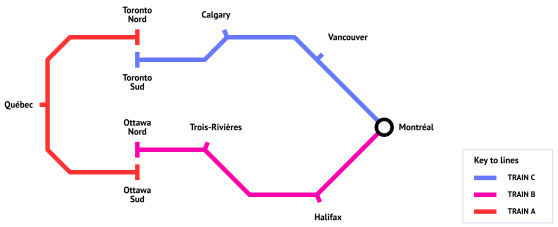

FIGURE 6 – Réseau ferroviaire de la TOC

2 Solution

Le problème que nous souhaitons résoudre est connus sous le nom de Time-Dependent Shortest Path, donc la recherche d'un chemin qui minimise un ou des critères dans un graphe aux arcs dont le poids varie en fonction du temps. L'algorithme que nous utilisons pour le résoudre est une variante de l'algorithme de Dijkstra [6, 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

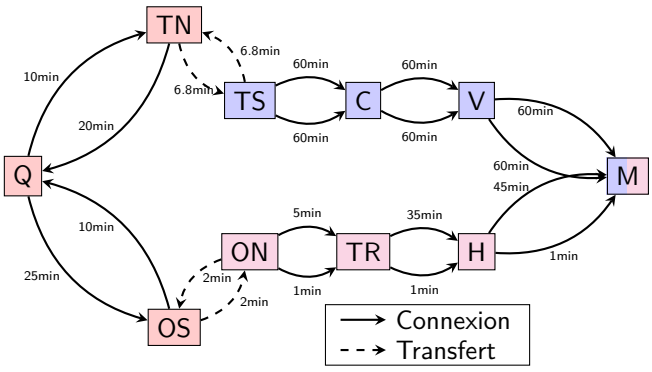

FIGURE 7 – Graphe de la TOC

- T l'ensemble des arcs de transfert

Nous implémentons ce graphe à l'aide d'une liste d'adjacence. Donc chaque nœud gardera une liste des connexions et transferts sortant de celui-ci. Ce graphe compte comme propriétés :

- Le graphe G est un multigraphe non-FIFO [4].
- 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 7 représente le graphe dépendant du temps de l'exemple de la TOC (les noms des gares ont été remplacées par leur 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 transferts. Les arcs de connexion ont été déterminés à partir du contenu de `src/test/resources/GTFS/TOC/stop_times.csv`. L'évaluation des arcs de transfert est expliquée dans la sous-section ci-dessous.

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 prendre en compte pour chaque gare la distance avec le reste des gares du graphe (nous voulons éviter des comportements quadratiques comme celui-ci). Nous définissons donc :

Définition 6 (Critère de proximité)**.** Le **critère de proximité** est la distance x (en mètres) tel que pour tout transfert représenté dans le graphe, la distance à vol d'oiseau entre les deux gares n'excède pas x :

$$\forall (u, v) \in T, \quad \texttt{dist}(u, v) \leq x$$

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

Définition 7 (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/test/resources/GTFS/TOC/stops.csv`).

Le choix de x est cruciale : x est directement proportionnel à la taille de T (complexités temporelles et spatiales impliquées) comme le montre la figure 8 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 qui ont des chemins à pied plus grand que x. Nous avons choisi le compromis $x = 500m$ comme valeur par défaut du programme mais l'utilisateur est libre de la redéfinir (voir README).

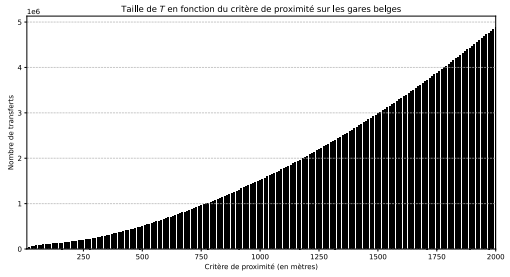

FIGURE 8 – Analyse de la taille de T en fonction de x sur les gares belges

Maintenant 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.

Algorithm 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 dist(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 $O(|V|^2)$. Nous éviterons donc de l'emprunter.

Approche avancée

L'approche avancée utilise un un arbre STR qui permet la recherche de gares à proximité en temps logarithmique amortie. Un arbre STR (ou STR tree) est un arbre T [10] qui indexe les gares spatialement en utilisant l'algorithme Sort-Tile-Recursive [11] pour packer 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).

Algorithm 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 beaucoup 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 conçu pour traversé les murs. Certains obstacles sur le chemin doivent être contourné. 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 cerlce autour de celui-ci :

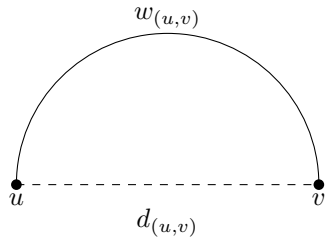

FIGURE 9 – Représentation d'un contournement d'obstacle à pied

Cette distance semble donc être une bonne norme 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 comptons également 90 secondes de marche de sécurité ($safety\_padding$) pour descendre et monter des transports. Pour obtenir le poid 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 noeud u à v où u.name $= s$ et v.name $= t$.
Dans les pseudocodes 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 fonction de coût sont expliquées dans la sous-section "Variantes" plus bas.

Dans l'algorithme TD-Dijkstra 3 (qui n'est qu'une pâle copie de l'algorithme de Dijkstra), nous initialisons un tas binaire avec (∞, ∞) pour tous les nœuds sauf les nœuds sources qui sont initialisés à $(0, 0)$ (le poids du chemin d'un nœud à lui-même est par définition nul). Le but est de piocher dans le tas (heap.delMin) chaque nœud qui a son chemin de coût minimal et

Algorithm 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) ;
```

pour celui-ci, nous vérifions s'il n'est pas la cible ou de coût égale à (∞, ∞). 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, nous ne trouverons plus de meilleurs chemins dans les prochaine pioche et pouvons donc arrêter l'algorithme. Pour chaque nœud pioché, nous améliorons le coût de tout les voisins (Relaxation).

Relaxation

Algorithm 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 u évalue le coût avec la fonction de coût sur chaque voisin (par transfert et par connexion). Pour rappel, vu qu'on considère le temps d'attente positif dans sa définition, tous les voisins doivent être traité, même ceux dont l'heure de départ de la connexion part avant l'heure d'arrivée du chemin jusque u. Si, pour un voisin v, nous trouvons un meilleur chemin passant par u, nous mettons à jour v.best_cost, v.best_time et v.best_edge et la position de v dans le tas (heap.decreaseKey) grâce au meilleur chemin menant jusque u.

Reconstruction de la solution

Algorithm 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, nous devons reconstruire le chemin de s à t. Nous effectuons un parcour de la liste chainée formée par les prédécesseurs stockés dans v.best_edge depuis la cible jusqu'à atteindre la première source rencontrée sur le chemin.

Pourquoi l'algorithme fonctionne

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

Lemme 1. *L'algorithme de TD-Dijkstra de s à t dès h dans le graphe temporel $G(V, A, T)$ retourne le chemin P qui minimise le coût (défini par la fonction de coût) parmi tous les chemins valides de s à t.*

Démonstration par l'absurde.

Si P est le chemin minimisant le coût, t.best_cost, il minimise également v.best_cost pour tout v le composant.

Vu que TD-Dijkstra retourne le chemin obtenu dès que t est pioché du tas, nous devons prouvé que pour tout u pioché après un nœud v du tas, u.best_cost $\geq v$.best_cost.

Supposons que ce n'est pas le cas : un nœud u pioché après un nœud v a un chemin moins coûteux que v.

Cela indique qu'il y a eu une relaxation avant que u soit pioché qui améliore son coût à u.best_cost $< v$.best_cost.

Deux cas possibles :

1. Cette relaxation à lieu **avant** que v soit pioché : Cela voudrait dire qu'au moment de pioché v, u.best_cost $< v$.best_cost et que donc ce serait u qui devrait être pioché avant v. contradiction
2. Cette relaxation à lieu **après** que v soit pioché : Par le lemme 2, la fonction de coût utilisée par notre algorithme de TD-Dijkstra lors de la relaxation d'un nœud w ne peut être inférieur à w.best_cost (et donc pour v et tous les nœuds piochés après, il est impossible d'améliorer u.best_cost en dessous de v.best_cost. contradiction

□

Variantes du problème

Algorithm 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 8 (Durée ponderée d'un arc)**.** La durée ponderée d'un arc est la durée de celui-ci multiplié 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 9 (Durée ponderée d'un chemin)**.** La durée ponderée d'un chemin et la somme des durées ponderées des arcs constitutifs (avec v la gare de départ d'un arc e) :

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

Définition 10 (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 vaut :

- au tuple (d_P, n_P) si l'utilisateur veut minimiser en priorité la durée pondérée totale.
- au 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 coût revient à d'abord comparer le premier élément puis le second. Donc le premier critère du couple sera minimisé en priorité. Additioner deux coût revient à additioner les éléments entre eux.

Pour rajouter un arc e au chemin finissant par la gare v, nous avons d_{P+e} et n_{P+e} :

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

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

Lemme 2. *La fonction de coût cost_function(v, e) qui calcule le coût du meilleur chemin jusque $v + e$ ne peut être inférieur à v.best_cost.*

Démonstration.

Selon les formules pour trouver n_{v+e} et d_{v+e} ci-dessus

1. n_{v+e} ne peut être inférieur à n_v.
2. d_{v+e} ne peut être inférieur à d_v car une durée et le temps d'attente ne peuvent être négatif.

□

Exemple de la TOC

Revenons à notre exemple sur le graphe de la TOC 7.

Premierement, nous initialisons le tas binaire sachant que $s =$ "Québec" et que le coût est (d_P, n_P) (donc on prioritise la durée totale du chemin) :
heap = [Q : (0, 0), * : (∞, ∞)]
(Notez que * : (∞, ∞) signifie que le reste des gares du tas ont un poids infini)

#1 Nœud pioché : Q (Québec)
relaxation de ses voisins : TN et OS
heap = [TN : (45min, 1), OS : (60min, 1), * : (∞, ∞)]

#2 Nœud pioché : TN (Toronto Nord)
relaxation de ses voisins : TS
heap = [TS : (51min, 2), OS : (60min, 1), * : (∞, ∞)]

#3 Nœud pioché : TS (Toronto Sud)
relaxation de ses voisins : TN et C
heap = [OS : (60min, 1), C : (180min, 3), * : (∞, ∞)]

#4 Nœud pioché : OS (Ottawa Sud)
relaxation de ses voisins : ON
heap = [ON : (62min, 2), C : (180min, 3), * : (∞, ∞)]

#5 Nœud pioché : ON (Ottawa Nord)
relaxation de ses voisins : OS et TR
heap = [TR : (85min, 3), C : (180min, 3), * : (∞, ∞)]

#6 Nœud pioché : TR (Trois-Rivières)
relaxation de ses voisins : H
heap = [H : (120min, 3), C : (180min, 3), * : (∞, ∞)]

#7 Nœud pioché : H (Halifax)
relaxation de ses voisins : M
heap = [M : (165min, 3), C : (180min, 3), * : (∞, ∞)]

#8 Nœud pioché : M (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 donnée 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 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

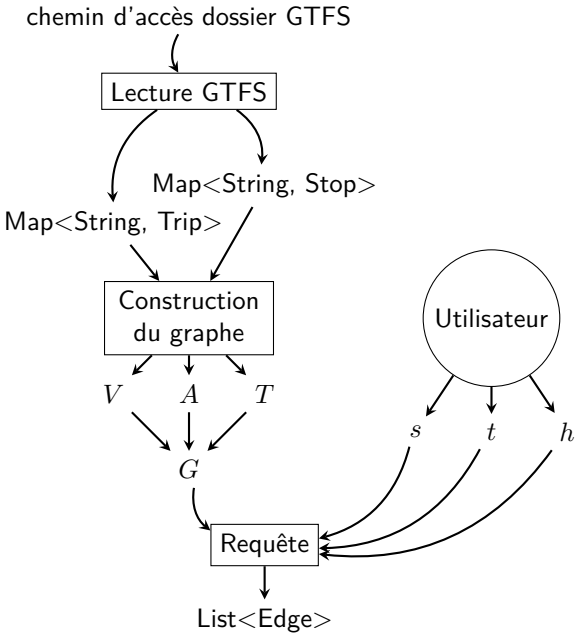

FIGURE 10 – Cheminement de l'information dans le programme Java

Convertir le contenu de stop_times.csv en trajets

L'une des façons les plus optimales de construire la table des trajets reviens 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 connexions 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 stocke le premier entier du couple et les 4 suivants le second :

n_P				d_P			
0	1	2	3	4	5	6	7

FIGURE 11 – Stockage d'un coût (n_P, d_P) d'un chemin en Java

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 que de trier le contenus 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) : $O(\bar{t}_x |V| log |V|)$.

NOTE TEMPORAIRE : O((V + t)logV) = O(VlogV)
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 consciense 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'entiereté 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 donc en :

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

dans notre cas car nous utilisons un tas binaire. Nous aurions également pu utiliser un tas de Fibonacci qui réduit la complexité à :

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

Nous n'avons pas implémenté de tas de Fibonacci dans notre programme Java du à la complexité et au manque de praticité de la structure.

Totale

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

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

Complexité spatiale

Lecture GTFS

La lecture qui converti des données GTFS en deux table de hashage : 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 (qui n'est qu'un arbre R) 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

Dans cette section, nous répondons à la grande question : "Dijkstra est-il le bon algorithme à utiliser dans ce contexte ?" (même s'il n'y a pas réellement de bon ou de mauvais algorithme) en comparant celui-ci avec d'autres algorithmes de *Time Depandent Shortest Path* connus.

Connection Scan Algorithm

Le *Connection scan algorithm* (CSA) [5] est un algorithme de recherche du plus cours 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 n'avons pas décidé de travailler avec 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 auquelle 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 peut 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 ici le poids des arcs sont positifs (donc pas besoin de gérer les cycles absorbant) 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 dévloppement vers le nœud cible afin de visiter le moins de nœuds possible. 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 heurisitque 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 portabilités. Nous comptons comme librairies externes :

- **univocity-parsers** offrant des outils de parcour de fichiers CSV rapides.
- **algs4** offrant l'implémentation du tas binaire (`IndexMinPQ`) utilisée pour l'algorithme de TD-Dijkstra.

6 Conclusion

Plusieurs défis ont été relever 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 requête de plus court chemin. Nous avons également réfléchi sur la façon la plus optimale d'évaluer les chemins à pieds 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 à pied. La théorie des graphes et de recherche du plus court chemin reste un domaine en évolution constante, d'autre algorithme récent (comme CSA, 2017) ou moins ont été consideré dans l'étape de pensée du projet, mais c'est l'algorithme de Dijkstra qui est resorti 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] Daniel Delling and Dorothea Wagner. *Time-Dependent Route Planning*, page 210. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

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

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

[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 T

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();

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

  /**
   * Obtient la cardinalité de
   * l'ensemble de n\oe uds
   */
  int V_card();

  /**
   * Obtient la liste d'adjacence du graphe
   */
  List<Node> vertices();

  /**
   * Obtient les paramètres utilisés
   * pour la construction du graphe
   */
  GraphSettings settings();
}
```

Listing 2 – Classe graph.Graph

graph.Node

```java
package algochoochoo.graph;

/**
 * Un n\oe ud du graphe
 */
public class Node {
  /**
   * Construit un n\oe ud vide depuis
   * une gare s
   */
  Node(Stop s);

  /** Convertit un n\oe ud en String */
  @Override
  String toString();

  /**
   * (TD-Dijkstra) Initialise le n\oe ud
   * pour une nouvelle requete
   * meilleur coût: INF
   * meilleur temps: INF
   * meilleur arc: null
   */
  void init();

  /**
   * Obtient la gare du n\oe ud
   */
  Stop stop();

  /**
   * (TD-Dijkstra) Obtient l'arc menant
   * à ce n\oe ud du meilleur chemin à l'étape
```

```
     * actuelle de l'algorithme
     */
    Edge best_edge();

    /**
     * (TD-Dijkstra) Obtient le meilleur coût
     * pour atteindre ce n\oe ud à l'étape
         actuelle de l'algorithme
     */
    long best_cost();

    /**
     * (TD-Dijkstra) Obtient la meilleure
     * heure d'arrivée pour atteindre ce n\oe ud
     * à l'étape actuelle de l'algorithme
     */
    int best_time();

    /**
     * Obtient les arcs de transferts sortants
     * du n\oe uds
     */
    List<Edge> transfers();

    /**
     * Obtient les arcs de connexions sortants
     * du n\oe uds
     */
    List<Edge> connections();

    /**
     * Obtient l'index du n\oe ud dans
     * la liste des n\oe uds
     * représentant le graphe
     */
    int index();

    /**
     * (TD-Dijkstra) Retourne si le n\oe ud est
     * source de l'algorithme
     */
    boolean is_source();

    /**
     * (TD-Dijkstra) Retourne si le n\oe ud est
     * cible de l'algorithme
     */
    boolean is_target();

    /**
     * (TD-Dijkstra) Améliore les variables
     * coût, heure d'arrivée et arc du chemin
     * après relaxation dans l'algorithme
     */
    void set_best(long cost, int time, Edge edge)
        ;

    /**
     * Défini la liste des transferts sortants
     * du n\oe ud
     */
    void set_transfers(List<Edge> t);

    /**
     * Ajoute une connexion du n\oe ud
     * dans sa liste de connexion sortantes
     */
    void add_connection(Edge c);

    /**
     * Défini l'index du n\oe ud dans la liste
     * de n\oe ud représentant le graphe
     * (Nécessaire pour algs4.IndexMinPQ)
     */
    void set_index(int i);

    /**
     * (TD-Dijkstra) Défini ce n\oe ud comme
     * source de l'algorithme
     */
    void declare_source();

    /**
     * (TD-Dijkstra) Défini ce n\oe ud comme
         cible
     * de l'algorithme
     */
    void declare_target();
}
```

Listing 3 – Classe graph.Node

graph.Edge

```
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();
```

```
54
55    /**
56     * Obtient le type de route de l'arc
57     */
58    RouteType type();
59
60    /**
61     * Obtient l'heure de départ de l'arc
62     * si c'est un arc de connexion, sinon -1
63     */
64    int departure_time();
65
66    /**
67     * Obtient la durée de l'arc
68     */
69    int duration();
70
71    /**
72     * Retourne si l'arc représente
73     * un transfert à pied
74     */
75    boolean is_transfer();
76    /**
77     * Retourne si l'arc représente une
78     * connexion
79     */
80    boolean is_connection();
81
82    /**
83     * Obtient la couleur de l'arc (fonctionnalit
          é GUI)
84     */
85    Color color();
86  }
```

Listing 4 – Classe graph.Edge

query.AlgoPriority

```
1   package algochoochoo.query;
2
3   /**
4    * La priorité de l'algorithme
5    * (= critère à minimiser)
6    */
7   public enum AlgoPriority {
8     /** La durée totale (pondérée) du chemin */
9     TOTAL_DURATION,
10
11    /** Le nombre de changements de trajet */
12    TRIP_CHANGES,
13  }
```

Listing 5 – Classe query.AlgoPriority

query.AlgoSettings

```
1   package algochoochoo.query;
2
3   /**
4    * Stocke les préférences utilisateurs
5    * liées à l'exécution de l'algorithme
6    */
7   public class AlgoSettings {
8     /**
9      * Construit des paramètres par défaut
10     */
11    AlgoSettings();
12
13    /**
14     * Construit les paramètres de l'algorithme à
15     * partir des arguments de l'exécutable
16     */
17    AlgoSettings(String[] args);
18
19    /**
20     * Si le nom des gares sources et cibles doit
            être identique au
21     * caractère près
22     */
23    boolean strict_search = true;
24
25    /** Les poids de mode de transport de l'
          algorithme */
26    Map<RouteType, Double> weights = new
          LinkedHashMap<>();
27
28    /** La priorité de l'algorithme */
29    AlgoPriority priority = AlgoPriority.
          TOTAL_DURATION;
30
31    /**
32     *
33     */
34    long cost_function(Node v, Edge e);
35
36    /**
37     * Affiche les paramètres sur STDOUT
38     */
39    void print();
40  }
```

Listing 6 – Classe query.AlgoSettings

query.Algorithm

```
1   package algochoochoo.query;
2
3   /**
4    * Exécute l'algorithme du plus court chemin
5    */
6   public class Algorithm {
7     /**
8      * Lance le CLI
9      */
10    public static void main(String[] args);
11
12    /**
13     * Construit un Algorithm avec un graph et
           des
14     * préférences utilisateur
15     */
16    public Algorithm(Graph G, AlgoSettings set);
17
18    // #### Private helpers ####
19
20    /**
21     * Retourne le meilleur chemin de s à t à
           partir de h
22     * en exécutant TD-Dijkstra
23     */
24    public AlgoResult dijkstra(String s, String t
          , int h);
25  }
```

Listing 7 – Classe query.Algorithm

query.Algorithm

```
1   package algochoochoo.query;
2
3   /**
4    * Stocke le résultat d'un algorithme de
         recherche
5    * du plus court chemin
6    */
```

```java
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.Algorithm

cli.View

```java
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

```java
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

```java
package algochoochoo.gui;


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

C Formule d'haversine