INFO-F203 - Algorithmique 2 - Rapport

Prénom	Nom	Matricule
Lucas	Verbeiren	000591223
Ethan	Van Ruyskensvelde	000589640

Table des matières

1 Introduction

Les transports en commun jouent un rôle central dans la mobilité en Belgique, où de nombreux citoyens dépendent quotidiennement des bus, trams, métros et trains pour se déplacer. Déterminer l'itinéraire le plus rapide entre deux arrêts constitue un enjeu crucial. Ce rapport présente nos implémentations de deux algorithmes de plus court chemin prenant en compte les trajets multimodaux, i.e. pouvant combiner différents modes de transport en commun. Ceux-ci prennent également en compte des segments de marche à pied entre les arrêts, mais ne tiennent pas compte des temps de transfert (d'un véhicule à l'autre, ou transition entre véhicule et marche). Les deux algorithmes sont regroupés sous le nom de *Connection Scan Algorithm (CSA)* et sont présentés dans [8]. Dans un premier temps, nous expliquons les notions générales associées aux problèmes de plus court chemin dans le contexte des transports en commun. Dans un deuxième temps, nous détaillons le fonctionnement et notre implémentation du CSA de base qui optimise seulement le temps d'arrivée (*earliest arrival CSA*). Dans un troisième temps, nous présentons une version fortement inspirée du *Pareto Connection Scan profile algorithm*. Ce dernier peut prendre en considération d'autres critères, en plus du temps d'arrivée, et optimise ceux-ci au sens Pareto.

2 Définitions générales

- **Arrêt (Stop)** : Arrêt pour n'importe quel mode de transport en commun, e.g. arrêt de Bus.
- **Connexion** : Un trajet direct entre deux arrêts consécutifs effectué par un véhicule, avec une heure de départ et d'arrivée ainsi qu'un point de départ et d'arrivée.
- **Chemin (à pied)** : Un chemin direct entre deux arrêts pouvant être effectué à pied, avec un point de départ et d'arrivée.
- **Mouvement** : Regroupe connexion et chemin à pied.
- **Trip** : Une instance d'un véhicule parcourant un trajet, et couvrant généralement plusieurs stop.
- **Horaire (Timetable)** [8] : L'horaire fait partie des paramètres des deux algorithmes. « Un horaire est un quadruplet $(\mathcal{S}, \mathcal{C}, \mathcal{T}, \mathcal{F})$ d'arrêts (stops) $\mathcal{S}$, de connexions (connections) $\mathcal{C}$, de trips $\mathcal{T}$, et de chemins à pied (footpaths) $\mathcal{F}$. »

3 Principales classes et méthodes utilisées dans le programme java

- **Coord** : représente des coordonnées géographiques. Permet de calculer la distance du grand cercle séparant deux instances de `Coord` via la méthode `distanceTo()`. Pour ce faire, cette dernière utilise la formule de `Haversine` [3].
- **Stop** : représente un arrêt. Cette classe contient essentiellement l'identifiant, le nom et les coordonnées géographiques de l'arrêt.
- **Connection** : classe représentant une connexion.
- **Footpath** : représente un chemin à pied. Celui-ci possède une distance ainsi qu'une durée de trajet. La durée de trajet est calculée en supposant que l'utilisateur marche à une vitesse de 5 km/h.
- **Movement** : représente un mouvement, une interface pour `Footpath` et `Connection`.
- **BallTree** : notre implémentation de `BallTree`, utilisé pour accélérer la création des chemins.
- **TransportType** : énumérée contenant les différents moyens de déplacement.
- **Data** : regroupe les donnés chargées depuis les sources GTFS, les mettant à disposition des algorithmes.
- **Solver** : classe regroupant la logique nécessaire pour le CSA de base. Les méthodes essentielles sont :
  - `solve()` : implémentation de l'algorithme CSA de base.
  - `reconstructSolution()` et `printInstructions()` : reconstruisent le chemin optimal et affichent les instructions de trajet.
  - `genFootpaths()` génère les chemins à pied en ne gardant que ceux dont la longueur est inférieur ou égale celle précisée, à l'aide de la classe `BallTree`.
- **MultiCritSolver** : classe regroupant la logique nécessaire pour le CSA multicritère. Les méthodes essentielles sont :
  - `solve()` : implémentation de l'algorithme CSA multicritère.
  - `displayJourney()` : affiche les instructions de trajet.
  - `genFootpaths()` même but et principe que pour la classe `Solver`.
- **AbstractSolver** : regroupe les fonctions communes à `Solver` et `MultiCritSolver`.
- **Main** : point d'entrée du programme. Cette classe construit les objets `data`, `Solver`, `MultiCritSolver`. Elle gère également l'input de l'utilisateur pour déterminer les arrêts de départ et d'arrivée, ainsi que

de l'heure de départ. Elle passe ensuite ces informations à `Solver` ou `MultiCritSolver`.

4 Chargement des données GTFS

Avant que nous puissions effectuer des requêtes de plus court chemin à l'aide de nos algorithmes, nous devons d'abord charger les données dans un objet `Data` pour ensuite passer celui-ci à l'algorithme choisi. Ces données, conformes au format `GTFS` (General Transit Feed Specification), contiennent toutes les informations liées aux arrêts, trajets, horaires et lignes de transport.

Les données GTFS utilisées dans notre programme sont les données publiques fournies par `DE LIJN`, la `STIB`, le `TEC` et la `SNCB`.

Le processus de chargement s'effectue en trois grandes étapes :

1. Création, pour chaque opérateur, d'un objet `CsvSet` contenant les chemins vers les fichiers `csv` correspondants.
2. Création de l'objet `Data`, via la méthode statique `Data.loadFromCSVs(CsvSet... csvSets)` qui appelle la méthode `loadOneCsvSet`, qui pour chaque `CsvSet`, traite les fichiers suivants :
   — **`stops.csv`** : contient les identifiants, noms et coordonnées géographiques des arrêts. Chaque arrêt est stocké dans une structure `Stop`, identifiée par son `stop_id`.
   — **`trips.csv`** : liste les trips et leur associe une route (`route_id`). Chaque trip est identifié par son `trip_id` et représente un parcours complet effectué par un véhicule.
   — **`routes.csv`** : fournit le nom de chaque ligne, ainsi que le type de transport, et est identifiable grâce au `route_id`. Ces données sont enregistrées dans une structure `RouteInfo` contenant toutes ces informations.
   — **`stop_times.csv`** : liste, pour chaque `trip`, les arrêts desservis dans l'ordre (`stop_sequence`), avec l'heure de départ à chaque arrêt. Nous convertissons les horaires en secondes depuis minuit pour simplifier les comparaisons.

   La méthode `loadOneCsvSet` commence par regrouper les entrées par `trip_id`, puis trie chacun de ces groupes par `stop_sequence`, pour ensuite générer les **connexions**, c'est-à-dire les trajets directs entre deux arrêts consécutifs d'un même trajet avec :
   — l'identifiant du trajet (`trip_id`),
   — la ligne de transport concernée (`RouteInfo`),
   — l'arrêt de départ et d'arrivée (`Stop`),
   — le temps de départ et le temps d'arrivée (en secondes).
   Une fois toutes les connexions générées à partir des données GTFS des différents opérateurs, nous trions toutes les connexions par heure de départ croissante, les deux algorithmes nécessitent que les connections soient triées ainsi. Toutefois, ceux-ci les parcourent dans des ordres opposés.
3. Instanciation du `Solver` et/ou du `MultiCritSolver` à l'aide de l'objet `Data` (passé aux constructeurs). Le constructeur de `Solver` et de `MultiCritSolver` appellent leur méthode `genFootpaths` pour générer les chemins à pied sur base des instances de `Stop`.

4.1 Construction des chemins à pied (`Footpath`)

Les algorithmes CSA [8] requièrent que le graphe des chemins à pied
— soit transitivement fermé
— satisfasse l'inégalité triangulaire
Cela permet de supposer que nous n'allons jamais emprunter deux chemins à pied d'affilée, car s'il existe une suite de chemins passant par les arrêts $s_0, \ldots, s_i$, il existe forcément un chemin reliant s_0 directement à s_i et celui-ci est certainement le plus rapide pour relier ces deux arrêts, dû à l'inégalité triangulaire.

Une solution simple, qui respecte ces deux critères, consiste à parcourir les arrêts à l'aide de deux boucles for imbriquées, en considérant chaque paire de `Stop` appartenant à deux chemins différents (un dans chaque sens). Cette approche présente une complexité spatiale et temporelle en $\Theta(n^2)$, où n est le nombre d'arrêts. Étant donné le très grand nombre d'arrêts dans nos données de test, ceci conduirait à des durées d'éxécution particulièrement longues. Pour pallier à ce problème, nous limitons les chemins à pied à ceux ne dépassant une certaine distance donnée `maxDistKm`. Ceci se justifie d'abord par le fait qu'un grand nombre de chemins seraient extrêmement longs et qu'aucun utilisateur ne voudrait les emprunter en pratique, certains traversant

la Belgique entière. D'autre part, plus un chemin est long, moins il a de chance d'être le meilleur trajet reliant son arrêt de départ à son arrivée (dû à la vitesse de marche très lente par rapport aux autre moyens de transport).

Il faut cependant noter que par suite de cette simplification, le graphe n'est plus transitivement fermé et il n'y a donc en principe plus de garantie que les algorithmes donnent encore la/les solution(s) optimale(s). Dans la suite du rapport nous supposerons que `maxDistKm` est suffisamment grand pour ne pas (trop) impacter l'optimalité de l'algorithme. Nous pouvons ajuster cette distance via le paramètre `maxDistKm` des méthodes `genFootpaths` de `Solver` et `MultiCritSolver`, permettant de créer les instances de `Footpaths`.
Il existe deux implémentations distinctes de `genFootpaths` car `Solver` et `MultiCritSolver` stockent les chemins différemment :

- `Solver` requiert une map associant chaque `stopId` à la liste des `Footpath` partant de cet arrêt.
- `MultiCritSolver` requiert une map associant chaque `stopId` à la liste des `Footpath` arrivant à cet arrêt.

Malgré cette différence, le procédé de construction des chemins reste identique : Puisque nous limitons les chemins à un certain rayon, nous utilisons un `BallTree`, une structure permettant d'effectuer des recherches de voisinage efficaces dans un espace à plusieurs dimensions (dans notre cas, deux : latitude/longitude). L'algorithme de génération s'effectue ainsi :

- Initialiser un `BallTree` avec tous les arrêts.
- Pour chaque arrêt source, chercher tous les arrêts voisins dans un rayon `maxDistKm`. Pour chaque voisin trouvé, créer l'objet `Footpath` avec les deux arrêts et l'ajouter à la map liant les arrêts de départ ou d'arrivée aux instances de `Footpath` (en fonction de `Solver` ou `MultiCritSolver`).

4.2 Complexité du chargement

Notons n le nombre d'arrêts et m le nombre de connexions. Le nombre de chemins, si nous les générons tous (fermeture transitive) correspond au nombre suivant :

$$\text{nombre de manières de relier deux arrêts} - \text{nombre de manières de relier un sommet à lui-même}$$

Cela vaut exactement $n^2 - n$. La complexité spatiale du chargement est donc en $\Theta(n^2 + m)$ (en utilisant uniquement deux boucles `for` imbriquées pour générer les chemins).
La complexité temporelle de la génération des chemins, en employant la méthode simple permettant la fermeture transitive des chemins (i.e. la double boucle `for` imbriquée), est en en $\Theta(n^2)$. Le tri des connexions par heure de départ est en $O(n \log(n))$, car le tri de l'interface `List` de java est une version de *tim-sort* [6], dont la complexité temporelle est $O(n \log(n))$ [2]. Comme expliqué dans [5], $\log_2(n!) \sim n \log(n)$ est la borne inférieure de complexité pour les algorithmes de tri par comparaison, il n'est donc pas possible de réellement faire asymptotiquement mieux que le *tim-sort* de l'interface `List` en utilisant un tri par comparaison. La complexité du chargement est donc en $O(n \log(n) + n^2) = O(n^2)$.

5 CSA de base (arrivée la plus tôt)

Le CSA de base permet de résoudre le problème suivant :

> Étant donné l'horaire, un arrêt de départ $pDep$, un arrêt d'arrivée $pArr$, et un temps de départ $tDep$, renvoyer les différents segments du trajet arrivant le plus tôt possible en $pArr$ en partant de $pDep$ en $tDep$.

Notre version du CSA de base accepte des noms d'arrêts, qui ne sont pas forcément uniques, au lieu des identifiants uniques. Elle considère tous les arrêts correspondant au nom de départ et au nom d'arrivée, supposés proches les uns des autres, et calcule le plus court trajet parmi ceux qui partent d'un des arrêts de départ possibles et arrivent à un des arrêts d'arrivée possibles.

5.1 Structure de données

Notre implémentation utilise principalement des `HashMap`, des `ArrayList` et une `Stack`.

- `HashMap<String, Stop> stopIdToStop` : permet d'associer l'identifiant d'un arrêt à l'objet `Stop`.
- `HashMap<String, List<Footpath» stopIdToOutgoingFootpaths` : associe à chaque arrêt une liste contenant les différentes instances de `Footpath` qui démarrent de cet arrêt.

— `List<Connection> connections` : liste des connexions triée par heure de départ croissante.
— `Map<String, BestKnownEntry> bestKnown` : permet de garder à jour le meilleur temps d'arrivée connu pour chaque arrêt, ainsi que le denier mouvement (trajet ou marche) qui nous y mène. Pour chaque arrêt, si son identifiant unique est `stopId`, alors `bestKnown` à l'entrée `stopId` peut se lire comme "nous parvenons en `stopId` à l'instant `bestKnown.get(stopId).getTArr()` en empruntant un trajet se terminant par `bestKnown.get(stopId).getMovement()`".
— `Stack<BestKnownEntry>` : utilisée pour reconstruire le chemin optimal, afin d'afficher les instructions dans l'ordre.

5.2 Description de l'algorithme

Dans cette sous-section, nous définissons d'abord les différentes notions cruciales pour comprendre l'algorithme `CSA de base (arrivée la plus tôt)`, puis nous détaillons son fonctionnement.

— Une connexion est dite *atteignable* si nous sommes en mesures de nous trouver à son arrêt de départ au moment auquel celle-ci démarre, i.e. $bestKnown_{tArr}(c_{pDep}) \leq c_{tDep}$.
— Calcul du temps d'arrivé d'un chemin à pied : $f_{tArr} = f_{tDep} + f_{travelTime}$ où f_{tDep} correspond au meilleur temps connu pour se rendre au point de départ de f, i.e. f_{pDep}.

1. **Critère de départ** : [8] Aucune connexion partant avant le temps de départ demandé n'est *atteignable*, il est donc inutile de prendre celles-ci en compte dans la suite de l'algorithme. La méthode `getEarliestReachableConnectionIdx` détermine l'indice de la première connexion dont l'heure de départ est supérieur ou égale à l'heure de départ demandée à l'aide d'une recherche dichotomique dans `connections`.
2. **Initialisation** : Pour chaque arrêt de départ, nous initialisons `bestKnown` aux entrées correspondantes comme suit :
   — Le temps d'arrivée `tArr = tDep` : Nous pouvons nous y rendre instantanément.
   — Le mouvement emprunté `movement = null` : Aucun mouvement n'est nécessaire pour s'y rendre.
   Nous explorons également chaque chemin accessible à pied f partant de ces arrêts de départ. Si f permet d'atteindre f_{pArr} plus rapidement que le meilleur temps connu pour atteindre celui-ci, i.e. $f_{tArr} < bestKnown_{tArr}(f_{pArr})$, alors nous mettons à jour le temps d'arrivée dans `bestKnown` à l'entrée f_{pArr} comme-suit :
   — Le temps d'arrivée `tArr =` f_{tArr}
   — Le mouvement emprunté `movement =` f
3. **Scan des connections et chemins** : Nous traitons les `Connection` itérativement, en ne tenant compte que de celles partant à notre heure de départ ou après, i.e. à partir de l'indice trouvé a l'étape **Critère de départ**. Appelons la connexion en cours de traitement c. (Pour rappel, les connexions sont triées par ordre croissant d'heure de départ, nous les scannons donc dans ce même ordre) :
   — si c est *atteignable* ,
   — et si c permet d'arriver plus tôt à son arrêt d'arrivée c_{pArr} que le meilleur temps connu pour s'y rendre à cette étape de l'algorithme : $c_{tArr} < bestKnown_{tArr}(c_{pArr})$
   — Alors, nous mettons à jour l'entrée de `bestKnown` correspondant à l'arrêt c_{pArr} car nous venons de trouver un moyen de s'y rendre plus rapidement :
   — Le temps d'arrivée `tArr =` c_{tArr}
   — Le mouvement emprunté `movement =` c
   Ensuite, pour chaque chemin f sortant de c_{pArr}, nous calculons le temps d'arrivée de ce chemin $f_{tArr} = c_{tArr} + f_{travelTime}$.
   Nous vérifions si f nous permet d'atteindre son arrêt d'arrivée f_{pArr} plus rapidement que le meilleur temps connu à cette étape de l'algorithme, i.e. $f_{tArr} < bestKnown_{tArr}(f_{pArr})$ Si c'est le cas, nous mettons à jour bestKnown correspondant à f_{pArr}.
   Notons que cette dernière étape liée aux chemins est sautée si c ne s'avère pas atteignable out pas plus rapide. Cette optimisation est appelée *limited walking*, et est expliquée dans [8].
4. **Critère d'arrêt** : Selon [8], nous pouvons arrêter l'itération sur les connections dès que nous scannons une connexion c telle que $c_{tDep} >= bestKnown_{tArr}(pArr)$.
   Supposons c, la première connexion telle que $c_{tDep} >= bestKnown_{tArr}(pArr)$, comme $c_{tDep} < c_{tArr}$, si $c_{tDep} >= bestKnown_{tArr}(pArr)$, alors par transitivité, nous avons que $bestKnown_{tArr}(pArr) < c_{tArr}$. c ne peut donc pas permettre d'améliorer `bestKnown`.
   Étant donné que nous scannons les connections par ordre croissant, ces inégalité reste vraie pour toutes les connections après c.

Puisque notre algorithme supporte plusieurs arrêts d'arrivée avec le même nom, et doit choisir celui auquel nous parvenons le plus rapidement, nous devons tester cette inégalité pour chaque arrêt dont le nom correspond au nom de la destination finale. Dès que cette inégalité n'est plus vraie, nous pouvons arrêter le scan des connections.

5. **Reconstruction** : Une fois $pArr$ le meilleur arrêt d'arrivée trouvé, nous reconstruisons, à partir de $pArr$, le trajet final (chaque mouvement composant celui-ci) partant de $pDep$ en $tDep$ et arrivant en $pArr$ le plus rapidement. Il s'agit d'un parcours en post-ordre car nous le reconstruisons en partant du point d'arrivée. Pour ce faire, nous utilisons donc une stack appelée `finalPath`. Chaque 'BestKnownEntry' (le mouvement et son temps d'arrivée) est empilée sur `finalPath`. Une autre fonction dépile ensuite la stack tout en affichant les différents segments du trajet.
   Comme pour les arrêts d'arrivée, plusieurs arrêts de départ pourraient avoir un nom correspondant à celui entré par l'utilisateur. Le cas de base de notre parcours en post-ordre est donc le cas où le stop que nous regardons actuellement est un des potentiels arrêts de départ.

Algorithm 1: Reconstruction du chemin optimal à partir de `bestKnown`

```
initialiser stack finalPath
currentStopId ← pArrIdEarliest
while currentStopId ∉ pDepIds do
    currentEntry ← bestKnown[currentStopId]
    empiler currentEntry dans finalPath
    currentStopId ← identifiant de l'arrêt de départ du mouvement de currentEntry
end
```

6. **Affichage de la solution** : Il suffit de dépiler la pile construite à l'étape de *reconstruction* et afficher chacune des étapes dans l'ordre, jusqu'à ce que cette pile soit vide.

5.3 Preuve d'exactitude

Dans cette sous-section, nous montrons que l'algorithme `CSA de base (arrivée la plus tôt)` fournit une solution optimale (pour rappel, en supposant que `maxDistKm` soit suffisamment grand).
Nous basons notre preuve sur [4], selon lequel, pour résoudre le problème du plus court chemin (*chemin* au sens théorie des graphes, correspondant donc à un *trajet* dans notre cas, à ne pas confondre avec les chemins au sens instances de `Footpath`) dans un graphe dirigé acyclique, notre algorithme doit :

— Examiner les sommets dans un ordre topologique,
— Pour chaque sommet v, on assigne la valeur $d(v) \leftarrow \min_{e=(u,v)\in\mathcal{E}}\{d(u) + w(e)\}$

D'abord, il est évident que notre graphe est dirigé. En effet, sans même tenir compte des horaires, l'existence d'une connexion n'implique pas qu'il existe une autre connexion effectuant le même trajet dans l'autre sens.

Ensuite, montrons que ce graphe est acyclique, comme cela est mentionné dans [7]. Par l'absurde, supposons qu'il existe un cycle formé de mouvements (formé de connexions et/ou chemins à pied) $m_1, m_2, \ldots, m_n$ tels que :

— $m_{n_{pArr}} = m_{1_{pArr}}$: m_n revient au point de départ du cycle.
— L'ensemble de ces mouvements forme un chemin valide dans le réseau (chaque connexion est atteignable).
— Le départ de m_1 se fait à l'instant t et l'arrivée de m_n se produit à l'instant t'.

Puisque chaque connection peut être empruntée seulement si celle-ci est atteignable, et que chaque movement prend un temps strictement positif, cela implique que chaque transition de mouvement augmente strictement le temps. Donc $t' > t$, ce qui contredit le fait que $m_1, \ldots, m_n$ forme un cycle, car certes, m_n revient à l'arrêt de départ de m_1, mais du temps s'est écoulé entre le moment auquel nous démarrons le cycle et celui auquel nous arrivons du cycle.

Étant donné que nous avons un graphe dirigé acyclique, il nous suffit de vérifier que le CSA respecte bien les deux critères pour les algorithmes de plus court chemin dans un DAG :

— **Ordre topologique** : [1] Puisque notre graphe est un DAG, il existe forcément au moins un ordre topologique. Montrons maintenant que parcourir les connexions par ordre de départ croissant constitue un ordre topologique. Par l'absurde, supposons qu'il ne s'agisse pas d'un ordre topologique. Cela signifie qu'il existe un trajet $m_1, \ldots, m_i$, et un autre trajet $m_i, \ldots, m_n$ nous permettant de reprendre m_1 au même moment que la fois précédente. Clairement, cela entre directement en contradiction avec le fait que les durées des mouvements sont strictement positives.

— **Relaxation des arêtes** : Le second point est trivial, cela correspond exactement à l'étape de relaxation des mouvements et de mise à jour de `bestKnown`.

5.4 Complexité

Notons n le nombre d'arrêts et m le nombre de connexions.
La complexité spatiale est en $O(n)$, car les seules structures de données utilisées sont `bestKnown` et `finalPath`.

— La taille de `bestKnown` est bornée par le nombre d'arrêts n, car elle contient au plus une entrée par arrêt.
— La taille de `finalPath` correspond au nombre de mouvements constituant le plus long des plus courts trajets considérés à un instant donné. Dans un graphe connexe à n sommets, la longueur d'un plus court chemin entre deux sommets est au plus $n-1$, ce qui correspond à un chemin simple visitant chaque sommet au plus une fois. En effet, tout chemin de longueur strictement supérieure à $n-1$ contient nécessairement un cycle, que l'on peut éliminer pour obtenir un chemin plus court. Ainsi, finalPath est également borné par $O(n)$.

La complexité temporelle de la recherche dichotomique à l'étape *critère de départ* est en $O(\log(m))$.

La complexité temporelle du scan des connexions (la boucle `for` principale, itérant sur les connexions) est en $O(m \cdot n)$ dans le pire des cas. En effet, si ni le *critère de départ*, ni le *critère d'arrêt* ne permettent d'éliminer des connexions, alors toutes les m connexions sont explorées. Pour chacune d'elles, l'algorithme parcourt les chemins partant de l'arrêt d'arrivée de la connexion. Or, si le graphe des chemins est transitivement fermé, un arrêt peut être relié à exactement $n-1$ autres, ce qui donne au total une complexité de $O(m \cdot n)$

La complexité temporelle d'une requête comprenant la recherche dichotomique est donc de l'ordre de $O(\log(m) + n \cdot m)$ dans le pire cas. Notons que comme $O(\log(m)) \in O(m)$, nous pouvons conclure que $O(\log(m) + n \cdot m) = O(n \cdot m)$.

6 CSA multicritère

Dans cette section, nous présentons puis réfutons une idée intuitive selon laquelle une simple modification du CSA basique permettrait d'en faire un algorithme multicritère. Nous présentons ensuite notre implémentation modifiée de l'algorithme mcpCSA qui peut prendre en compte plusieurs critères et optimise ceux-ci au sens Pareto.

Dans le CSA basique, la fonction de coût, utilisée pour comparer deux trajets lors de la mise à jour de `bestKnown`, est définie par :

$$\text{cost} : \mathbb{T} \to \mathbb{T}, \quad t \mapsto t$$

C'est-à-dire que le coût est directement le temps d'arrivée.

L'idée intuitive consiste alors à remplacer la fonction de coût par une fonction déterminée par l'utilisateur, prenant en paramètre les valeurs pour les critères supportés par le programme, e.g. le nombre de mouvements effectués en trams :

$$\text{cost} : \mathbb{X}_1 \times \cdots \times \mathbb{X}_n \to \mathbb{C}, \quad (x_1, \ldots, x_n) \mapsto c$$

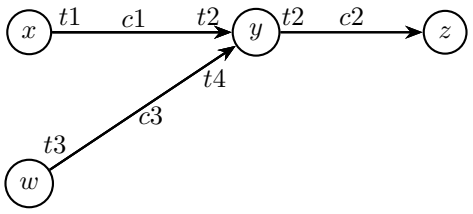
où

— $x_1, \ldots, x_n$ représentent des valeurs pour les critères choisis par l'utilisateur, e.g. temps d'arrivée, nombre de connections en tram, nombre de chemins à pied.
— $c \in \mathbb{C}$ est le coût global calculé selon la fonction définie par l'utilisateur.

Nous conservons alors dans `bestKnown`, pour chaque arrêt, les $x_1, \ldots, x_n$ correspondants au trajet menant à cet arrêt et minimisant le coût.
Afin d'effectuer la comparaison entre deux trajets X, Y pour déterminer si X a un meilleur coût que Y, il suffit de vérifier :

$$\text{cost}(x_1, \ldots, x_n) < \text{cost}(y_1, \ldots, y_n)$$

Malheureusement, cette approche intuitive ne fonctionne pas, comme le contre-exemple suivant le montre :



Notons pour toute connexion c :

$$\text{cost}(c) = \text{cost}(pDep \rightarrow c_{pDep} + c)$$

Autrement dit, le coût d'une connexion c correspond au coût du trajet composé du meilleur itinéraire partant d'un point de départ spécifié par l'utilisateur, atteignant c_{pDep}, puis empruntant la connexion c.

Étant donné que $c1_{tDep} < c2_{tDep} < c3_{tdep}$, nous scannons d'abord $c1$. Une fois le scan de $c1$ terminé :

$$\texttt{bestKnown}_{cost}(y) \leq \text{cost}(c1)$$

Il en va de même pour le scan de $c2$:

$$\texttt{bestKnown}_{cost}(z) \leq \text{cost}(c2)$$

Et enfin, nous scannons la connexion $c3$. Supposons que :

$$\texttt{bestKnown}_{cost}(z) > \text{cost}(c3)$$

Nous remplaçons alors les $x_1, \ldots, x_n$ (ainsi que l'instance de `Movement`) dans `bestKnown` pour l'arrêt y. Cette décision rend notre solution partielle invalide, car en prenant $c3$ qui arrive en y à l'instant $t4$, nous ne pouvons plus atteindre $c2$ qui part de y en $t2$.

Formellement, cela est dû au fait que seul le critère de temps d'arrivée le plus tôt respecte *la condition d'échangeabilité préfixe* : [9] Soit $s_1, s_2, \ldots, s_n$ un trajet optimal $A@tDep \rightarrow B$, i.e. partant de A en $tDep$ et arrivant en B. Considérons un préfixe quelconque $s_1, \ldots, s_i$, un trajet complet arrivant en C. Remplacer le préfixe par un autre trajet optimal $A@tDep \rightarrow C$ doit donner un autre trajet optimal $A@tDep \rightarrow B$. Cependant, seul le critère d'arrivée le plus tôt satisfait cette propriété, les autres critères ne garantissent pas que la connexion que nous avions prévu de prendre en C soit toujours atteignable avec le nouveau préfixe.

Si cette approche intuitive ne fonctionne pas, nous pouvons cependant utiliser une autre condition d'échangeabilité : La *condition d'échangeabilité suffixe* [9] : Soit $s_1, \ldots, s_n$ un trajet optimal $A@\tau \rightarrow B$. Considérons n'importe quel suffixe $s_i, \ldots, s_n$ qui est un trajet complet partant de C en τ'. Remplacer ce suffixe par un autre trajet optimal $C@\tau' \rightarrow B$ doit donner un autre trajet optimal.

Étant donné que cette seconde condition est maintenue par les fonctions de coût multicritères, nous en tirons avantage pour notre algorithme en construisant les trajets à partir du point d'arrivée [9].

Définissons le concept de *domination* [8] : soient deux tuples $X = (x_1, \ldots, x_n)$ et $Y = (y_1, \ldots, y_n)$. Nous dirons que X domine Y si et seulement si

$$\forall i \in \{1, \ldots, n\},\ x_i \leq y_i \quad \wedge \quad \exists i \in \{1, \ldots, n\},\ x_i < y_i.$$

Sur base de la définition de la domination, nous définissons le concept de *Pareto-optimal* [8] : Soit P un ensemble de tuples à composantes scalaires, un tuple x est Pareto-optimal par rapport à P s'il n'existe aucun autre tuple $y \in P$ tel que y domine x.

Nous pouvons appliquer la définition de la domination pour deux trajets arrivant au même arrêt $j1, j2$ en remplaçant $x_1, \ldots, x_n$ et $y_1, \ldots, y_n$ par les valeurs pour chaque critère, e.g. x_1 représente le nombre de connexions en tram du trajet $j1$, de même pour x_2 mais pour le trajet $j2$.

Le mcCSA permet de résoudre le problème suivant :

Étant donné l'horaire, un arrêt de départ $pDep$, un arrêt d'arrivée $pArr$, et un temps de départ $tDep$, et des critères de domination, renvoyer le sous-ensemble de trajets *Pareto-optimaux* par rapport aux critères de domination imposés par l'utilisateur et à l'ensemble de tous les trajets partant de $pDep$ à $tDep$ ou plus tard, et arrivant en $pArr$.

Puisque nous construisons notre trajet en partant de l'arrivée, nous scannons les connexions par ordre de départ décroissant. À chaque étape de l'exécution de l'algorithme, pour chaque arrêt p, l'algorithme doit maintenir un ensemble de trajets partiels $\mathcal{S}_p$ tel que :

$$\forall j \in S_p, \quad \nexists j' \in S_p \text{ tel que } j'_{tDep} > j_{tDep} \text{ et } j' \text{ domine } j$$

où :

— j_{tDep} désigne l'heure de départ du trajet j,

Nous appelons cet ensemble S_p la *fonction de profil* de l'arrêt p. L'algorithme ajoute et supprime des trajets partiels (jusqu'à destination) dans les fonctions de profile au fur et à mesure que celui-ci scanne les connexions dans l'ordre de départ décroissant. À chaque connexion scannée c nous tentons d'ajouter tous les trajets partiels qui peuvent être effectués à partir de la connexion c dans la fonction de profile de l'arrêt c_{pDep}. Ceux dominés par un trajet partiel de $S_{c_{pDep}}$ partant en $tDep$ ou plus tard, ne sont pas gardés.

Afin de rendre le choix des critères de domination souple, nous introduisons la notion de `CriteriaTracker`, une classe permettant de maintenir les informations nécessaires à l'évaluation de tous les critères d'un trajet partiel donné. Les fonctions de profile gardent pour chaque temps de départ de trajets partiels, l'ensemble des trajets partiels concernés, chacun avec son `CriteriaTracker` associé. Lors de l'ajout des trajets partiels dans la fonction de profile de c_{pDep}, seuls sont ajoutés les trajets partiels non-dominés par les trajets-partiels partant en c_{tDep} ou plus tard.

Notons que la définition du contenu d'une fonction de profil nous autorise à conserver un trajet j malgré le fait que celui-ci soit dominé par un trajet j' si l'heure de départ de j' est antérieure à celle de j, c'est-à-dire : $j'_{tDep} < j_{tDep}$

Nous devons conserver ces trajets car nous construisons les trajets en partant du point d'arrivée : Lorsque nous ajoutons un trajet j dans la fonction de profile de l'arrêt p, nous n'avons aucune garantie qu'une autre connexion arrivant en p et rendant la première connexion de j atteignable existe. Nous devons donc conserver des alternatives à j, partant plus tard, celles-ci peuvent être dominées par j.

De plus, nous retirons les trajets partant avant c_{tDep} qui sont dominés par les trajets partiels que nous venons d'ajouter, car il est préférable d'attendre et prendre ceux qui les dominent plutôt que de prendre ces trajets partiels dominés.

Les trajets partiels que nous tentons d'ajouter à chaque connexion c scannée peuvent résulter de trois cas évalués successivement [8]. En effet, l'utilisateur n'a que trois possibilités pour continuer son trajet à partir de la connexion c :

1. L'utilisateur peut sortir du véhicule et marcher jusqu'à la destination (s'il n'y est pas déjà).
2. Il peut rester dans le même véhicule et prendre la prochaine connexion (si celle-ci existe).
3. Il peut sortir du véhicule et rentrer dans un autre véhicule.

Pour le cas 1, nous considérons un nouveau trajet partiel démarrant en $tDep$ dans S_{cPdep}, celui-ci contient un chemin (le chemin final), et une connexion. De plus, puisque ce chemin final fait partie d'un candidat, nous ajoutons le trajet composé d'uniquement celui-ci dans la fonction de profil de $S_{c_{pArr}}$.

Pour le cas 2, nous devons considérer les meilleurs trajets partiels pour chaque `CriteriaTracker` (chaque combinaison de valeurs des différents critères) empruntant la prochaine connexion de ce trip. Pour cela, nous stockons dans T, pour chaque trip, ces meilleurs trajets. Celui-ci est mis à jour après avoir considéré les trois cas.

Pour le cas 3, il faut envisager les meilleurs trajets que nous pouvons effectuer à partir de c_{pArr}, à nouveau, pour chaque `CriteriaTracker` (chaque combinaison de valeurs des différents critères). C'est ce que nous appelons *évaluer* $S_{c_{pArr}}$ au temps c_{tArr}.

Nous regroupons les résultats de l'évaluation de ces trois cas. Pour ce faire, si deux `CriteriaTracker` contiennent les mêmes valeurs des différents critères, nous prenons le meilleur temps d'arrivée des deux trajets. Ce groupe représente donc les meilleurs trajets possibles à partir de c.

Nous utilisons ce groupe pour mettre à jour l'entrée correspondant à c_{tripId} dans T, et nous insérons ces différents trajets partiels dans la fonction de profil $S_{c_{pDep}}$, avec le temps de départ c_{tDep}.

Pour chaque cas, nous calculons les nouveaux `CriteriaTracker` (e.g. ajouter un tram au compteur de trams).

Notre gestion du cas 1 ne permet que de gérer les chemins finaux, il nous reste à gérer les autres chemins. Nous devons donc considérer le cas où l'utilisateur est arrivé au départ de c en prenant un chemin.
Pour ce faire, pour chaque chemin qui arrive en c_{pDep}, nous réutilisons le groupe des meilleurs trajets partiels et nous modifions chacun de ses `CriteriaTracker` pour que ceux-ci tiennent compte du chemin arrivant en c_{pDep}. Nous insérons chacun de ces groupes de meilleurs trajets partiels modifiés dans la fonction de profile de c_{pDep}, avec le temps de départ $c_{tDep} - f_{travelTime}$. Lorsqu'aucun trajet partiel n'a été gardé à l'étape précédente (l'insertion dans des meilleurs trajets partiels dans la fonction de profile $S_{c_{pDep}}$), cette étape-ci n'est pas nécessaire, car tous ces trajets seront inévitablement aussi dominés. Cette optimisation est appelée *limited walking* [8].

Avec ceci et le cas 1, nous tenons compte de tous les chemins, excepté celui qui relie directement $pDep$ à $pArr$.

Il n'est nécessaire de distinguer le cas 2 du cas 3 que si des critères sont liés aux trips, e.g. le nombre de transferts. Notre implémentation ne supporte pas ce type de critères, celle-ci supporte seulement les critères liés aux mouvements individuels (e.g. nombre de connexions en bus). Nous laissons donc le code pour le cas 2 commenté, pour ne pas impacter les performances, et nous ne tenons pas compte de celui-ci (et de T) pour la suite.

Une fois l'ensemble des connexions scannées, S_{pDep} doit contenir l'ensemble de tous les trajets partiels partant de $pDep$ en $tDep$ ou plus tard, arrivant en $pArr$ et n'étant pas dominé par un autre trajet partant plus tard. Connaissant notre heure de départ $tDep$, il suffit d'extraire de S_{pDep} tous les trajets non-dominés partant en $tDep$ ou plus tard. Nous ne conservons donc plus les trajets qui sont dominés par d'autres trajets partant plus tôt. Nous pouvons les retirer car nous avons maintenant la certitude que les meilleures alternatives sont atteignables puisque celles-ci démarrent en $tDep$ ou après. (Nous n'avions pas encore cette certitude lorsque nous construisions nos trajets partiels). Cet ensemble est le sous-ensemble de trajets *Pareto-optimaux* par rapport aux critères de domination imposés par l'utilisateur et à l'ensemble de tous les trajets partant de $pDep$ à $tDep$ ou plus tard, et arrivant en $pArr$.

Le pseudo-code algorithm 2 renvoie les fonctions de profile de tous les arrêts, S, en considérant tous les trajets arrivant en $pArr$ et empruntant uniquement des connexions partant en $tDep$ ou plus tard. (Les trajets partant d'autres arrêts différents de $pDep$ sont également pris en compte, car cet algorithme est un algorithme *all to one*, signifiant que celui-ci résout pour tous les arrêts de départ).

Algorithm 2: Calcul de toutes les fonctions de profile pour une requête

```
1  Function solve(pArr, tDep) return S :
2      initialiser Map S
3      initialiser Map D
4      for connexions c par c_{tDep} décroissant dans connexions partant après tDep do
5          initialiser Map τ_c
6          if c arrive en p_{Arr} then
7              initaliser newTracker et lui faire tenir compte de c
8              mettre à jour τ_c avec (newTracker, (c_{tArr}, c))
9          end
10         else
11             finalFootpath ← D[c_{pArr}]
12             tArrWithFootpath ← c_{tArr} + finalFootpath_{travelTime}
13             initaliser newTracker et lui faire tenir compte de c et finalFootpath
14             mettre à jour τ_c avec (newTracker, (tArrWithFootpath, c))
15             foopathTDep ← c_{tArr}
16             initaliser finalFootpathNewTracker et lui faire tenir compte de finalFootpath
17             insérer (tArrWithFootpath, finalFootpath) au temps de départ foopathTDep, au tracker
                finalFootpathNewTracker, dans S[c_{pArr}]
18         end
19         SCpArrEvaluatedAtCtArr ← S[c_{pArr}] évalué en c_{tArr}
20         for trajetPartiel dans SCpArrEvaluatedAtCtArr do
21             newTracker ← trajetPartiel_{tracker} qui tient compte de c
22             mettre à jour τ_c avec (newTracker, (trajetPartiel_{tArr}, c))
23         end
24         insérer tout τ_c au temps de départ c_{tDep}, dans S[c_{pDep}]
25         auMoinsUnNonDominé ← true si au moins un trajet partiel de τ_c a été gardé dans S[c_{pDep}]
26         if auMoinsUnNonDominé then
27             SCpDepEvaluatedAtCtDep ← S[c_{pDep}] évalué en c_{tDep}
28             for f dans chemins qui arrive en c_{pDep} do
29                 fTDep ← c_{tDep} − f_{travelTime}
30                 for trajetPartiel dans SCpDepEvaluatedAtCtDep do
31                     newTracker ← trajetPartiel_{CriteriaTracker} tenant compte de f
32                     insérer (trajetPartiel_{tArr}, f) au temps de départ fTDep, au tracker newTracker
33                 end
34             end
35         end
36     end
37     return S
```

Notons que, comme pour le CSA de base, nous pouvons à nouveau appliquer l'optimisation *critère de départ*, c'est pourquoi nous n'itérons pas sur toutes les connexions. Notons également que τ_c correspond à la structure dans laquelle nous procédons au groupement des trajets partiels empruntant c.

6.1 Reconstruction de la solution

Nous reconstruisons les différents trajets Pareto-optimaux à l'aide des `CriteriaTracker`, des fonctions de profil et des temps de départ.
Considérons le trajet j démarrant en j_{tDep} de l'arrêt j_{pDep} et arrivant en j_{pArr}, composé des mouvements $m_1, \ldots, m_n$. Notons $j_1, \ldots, j_n$, les trajets partiels suffixes de j : j_i correspond au sous-trajet partiel composé de $m_i, \ldots, m_n$.

Pour tout sous-trajet j_i, connaissant
- l'arrêt de départ $j_{i_{pDep}}$,
- le temps de départ $j_{i_{tDep}}$,
- $j_{i_{\text{CriteriaTracker}}}$, le CriteriaTracker en $j_{i_{pDep}}$

nous pouvons retrouver le prochain mouvement à emprunter m_i : Il s'agit du mouvement se trouvant dans $S_{j_{i_{pDep}}}$, partant de $j_{i_{pDep}}$ en $j_{i_{tDep}}$ ou le plus tôt après, et dont le `CriteriaTracker` est égal à $j_{i_{\text{CriteriaTracker}}}$.

Cette étape est effectuée par la procédure `getFirstMatch`.
Connaissant également le mouvement à emprunter m_i, nous pouvons déterminer :

— $j_{i+1_{pDep}} = m_{i_{pArr}}$

— $$j_{i+1_{tDep}} = \begin{cases} c_{tArr} \text{ si } m_i \text{ est une connexion } c, \\ j_{i-1_{tDep}} + f_{travelTime} \text{ si } m_i \text{ est un chemin f.} \end{cases}$$

— $j_{i+1_{\text{CriteriaTracker}}}$, en décrémentant le compteur de bus/tram/train/metro/chemin de $j_{i_{\text{CriteriaTracker}}}$, en fonction du type de mouvement de m_i.

Le cas de départ utilise le `CriteriaTracker` et le temps de départ du trajet complet (partant de $pDep$) que nous voulons reconstruire. L'algorithme s'arrête à l'étape à laquelle l'arrêt de départ du trajet partiel considéré est l'arrêt de destination, c'est-à-dire quand $j_{i_{pDep}} = pArr$.

Le pseudo-code algorithm 3 illustre comment nous tirons parti de cette induction pour reconstruire la solution.

Algorithm 3: Reconstruction d'un trajet solution

```
1  Function displayJourney(S, pDepId, pArrId, tDep, criteriaTracker) :
2      stopId ← pDepId
3      while stopId ≠ pArrId do
4          movement ← S[stopId].getFirstMatch(tDep, criteriaTracker)
5          print "taking" movement
6          stopId ← movement.pArr.id
7          criteriaTracker ← criteriaTracker ajusté en fonction du type de transport de movement
8          if movement est un Footpath then
9              tDep ← tDep + movement_travelTime
10         else
11             tDep ← movement_tArr
12         end
13     end
```

6.2 Structure de données

Notre implémentation utilise principalement des `HashMap`, des `ArrayList` et des `Pair`.

— `HashMap<String, Stop> stopIdToStop` : permet d'associer l'identifiant d'un arrêt à l'objet `Stop`.

— `HashMap<String, List<Footpath» stopIdToIncomingFootpaths` : associe à chaque arrêt une liste contenant les différentes instances de `Footpath` qui arrivent de cet arrêt.

— `List<Connection> connections` : liste des connexions triée par heure de départ croissante.

— `Map<String, ProfileFunction> S` : associe chaque arrêt à sa fonction de profile.

— `Map<String, Footpath> D` : associe chaque arrêt au chemin partant de cet arrêt et arrivant à destination. (Celui-ci est utilisé pour le cas 1, les chemins finaux).

6.2.1 Fonction de profile

La fonction de profil est composée d'un unique attribut : `ArrayList<Pair<Integer, HashMap<CriteriaTracker, Pair<Integer, Movement»» entries;`
Les paires se trouvant dans le `ArrayList` contiennent :

1. Un temps de départ,
2. Une map contenant les trajets partiels partant à ce temps de départ. Cette map stocke les trajets partiels en associant à chaque `CriteriaTracker`, une paire contenant :
   (a) le temps d'arrivée de ce trajet partiel,
   (b) le prochain mouvement à emprunter pour suivre ce trajet,

Celles-ci sont ordonnées par temps de départ décroissant afin de maximiser le nombre d'insertions en fin de liste (moins coûteuses). Puisque nous scannons les connexions par ordre de départ décroissant, nous allons généralement insérer dans la fonction de profile des groupes de trajets partiels débutant de plus en plus tôt. Nous ne pouvons pas garantir que cela est toujours de plus en plus tôt à cause des chemins entre les arrêts : Pour rappel, pour chaque chemin non-final, nous devons insérer nos trajets partiels empruntant f suivi d'une connexion c, au temps de départ $c_{tDep} - f_{travelTime}$. Ce temps de départ pourrait être avant le départ de la prochaine connexion scannée partant de f_{pDep}, appelons-la c'. Dans ce cas, l'insertion dans le profile de

$S_{c'_{pDep}}$ ne sera pas en fin de liste car il existe déjà les trajets partiels partant plus tôt dans le profile : ceux partant en $c_{tDep} - f_{travelTime}$.

6.3 Complexité du CSA profile multicritère Pareto

Nous ne sommes pas parvenus à trouver une expression de la complexité de l'algorithme dans son ensemble, dans le pire cas. Toutefois, nous donnons des bornes pour les différentes fonctions.

- **insertion dans une fonction de profile** : Supposons que la fonction de profile contienne déjà n trajets partiels, et que nous tentons d'insérer m nouveaux trajets partiels dans celle-ci. La complexité temporelle de l'insertion sera alors en $O(m^2 + m \cdot n)$, car pour chaque trajet partiel dans m, nous devons :
  - ignorer celui-ci s'il est dominé par un autre autre trajet partiel au sein des m nouveaux trajets partiels. Cette vérification a une complexité temporelle en $O(m)$.
  - vérifier si celui-ci est dominé par un autre des n trajets partiels déjà présents et partant plus tard, auquel cas nous n'ajoutons pas ce nouveau trajet partiel. Cette opération prend un temps $O(n)$.
  - retirer les trajets partiels partant plus tôt et étant dominé par ce nouveau trajet partiel. À nouveau, la complexité temporelle de cette opération est $O(n)$.
- **boucle principale** En considérant les itérations sur les trajet partiels en temps constant (ce qui en pratique n'est évidemment pas le cas), et qu'il y a n arrêts et m connexions, nous pouvons dire que la complexité de l'algorithme est en $O(m \cdot n)$.
  La complexité temporelle du scan des connexions (la boucle `for` principale, itérant sur les connexions) est en $O(m \cdot n)$ dans le pire des cas. En effet, si ni le *critère de départ* ne permet pas d'éliminer des connexions, alors toutes les m connexions sont explorées. Pour chacune d'elles, l'algorithme parcourt les chemins arrivant à l'arrêt d'arrivée de la connexion. Or, si le graphe des chemins est transitivement fermé, un arrêt peut être relié à exactement $n - 1$ autres, ce qui donne au total une complexité de $O(m \cdot n)$
  (Comme pour le CSA de base, la recherche dichotomique pour le *critère de départ* est en $O(\log(m)) \in O(m)$).

Ces mauvaises complexités temporelles nous obligent à réduire le set de données de test à un seul opérateur pour que l'algorithme se termine en un temps raisonnable.

7 Conclusion

En conclusion, notre implémentation du CSA de base se distingue par sa simplicité, car elle ne nécessite pas de file de priorité comme Dijkstra. Cependant, la nécessité de la fermeture transitive du graphe des chemins constitue une limitation importante. Notre version multi-critère offre une grande flexibilité grâce à l'optimisation de Pareto (plusieurs voyages proposés), mais cela se traduit par un temps d'exécution nettement plus élevé.

Références

[1] Dags and topological ordering. `https://ocw.tudelft.nl/wp-content/uploads/Algoritmiek_DAGs_and_Topological_Ordering.pdf`. TU Delft OpenCourseWare, voir Lemma 3.20, pages 20–25.

[2] Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. On the worst-case complexity of timsort. `https://arxiv.org/pdf/1805.08612`, 2019.

[3] Rezania Agramanisti Azdy and Febriyanti Darnis. Use of haversine formula in finding distance between temporary shelter and waste end processing sites, 2019. voir page 3.

[4] Jean Cardinal. Plus courts chemins dans les graphes acycliques, 2021. Slides du cours INFO-F203 : Algorithmique 2, Université Libre de Bruxelles, voir page 16.

[5] Jean Cardinal. Tri fusion (merge sort) et tri rapide (quicksort), 2021. Slides du cours INFO-F203 : Algorithmique 2, Université Libre de Bruxelles, voir page 17.

[6] Oracle Corporation. List.sort(comparator). Visitté le : 2025-05-14. implémentation de la méthode sort de l'interface List (TimSort).

[7] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly simple and fast transit routing. `https://i11www.iti.kit.edu/extra/publications/dpsw-isftr-13.pdf`, 2013.

[8] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Connection scan algorithm, 2017.

[9] Ben Strasser and Dorothea Wagner. Connection scan accelerated. https://dl.acm.org/doi/pdf/10.5555/2790174.2790186, 2014. Workshop on Algorithm Engineering and Experiments (ALENEX), SIAM.