Florian CAMBRESY 21803824 TD4
Mickaël LE DENMAT 21804355 TD4
Adrien LAVALLIERE 21800547 TD3

Compte rendu du projet :

Sommaire

I) Introduction....1
II) Structures de données....1
a) Classe Matrice.java....1
b) Classe TblStation.java....2
c) Classe TblDijkstra....3
d) Classe Station.java....4
e) Classe Travel.java....4
f) Classe MetroLine.java....5
g) Classe Metro.java....5
h) Classe Fenetre.java....7
i) Classe Panneau.java....7
j) Classe Menu.java....8
III)Explication du code....8
IV) Instructions pour exécuter le programme....11

I) Introduction

Le projet a été codé en Java car c'est un langage modulaire, il n'y a ni problèmes de gestion de la mémoire grâce au "garbage collector", ni de problèmes de gestion des pointeurs. Il contient aussi une vaste bibliothèque de fonctions permettant de simplifier le code en particulier pour la partie graphique. Dans la première partie nous expliquerons le but de chaque classe et détaillerons leur contenu. Puis nous expliquerons le fonctionnement du programme dans la seconde partie. Et pour finir la manière d'interagir avec le programme. Les parties écrites en bleues sont des détailles précis du code pour vous aider à le comprendre.

II) Structures de données

a) Classe Matrice.java

Les temps entre les stations présents dans le fichier "metroL.txt" sont stockés dans un objet "Matrice" implémenté dans le fichier "Matrice.java" (80 lignes).

Variables :

- ➔ **int []directions :** notre matrice.
- ➔ **int line :** le nombre de lignes.
- ➔ **int column :** le nombre de colonnes.

Constructeur :

- ➔ **public Matrice(int[][] M)**

Les indices "line", "column" représentent les stations par leur numéro dans le fichier "metroL.txt",

ainsi la station Abbesse est la station d'indice 0, celle d'Alexandre Dumas la numéro 1, etc... Pour connaître le temps entre la station 0 et 1, on peut regarder soit aux coordonnées (0,1), soit (1,0) de la matrice. Cette objet est uniquement muni de méthodes "get" afin de récupérer les données. Cette classe a été créé dans le cas où nous aurions eu besoin d'opérations supplémentaires sur les matrices, mais cela n'a pas était le cas, c'est pourquoi il n'y a pas d'autres méthodes.

b) Classe TblStation.java

La classe "TblStation.java" permet de lire le fichier de configuration afin de créer notre matrice de temps (200 lignes).

Variables :

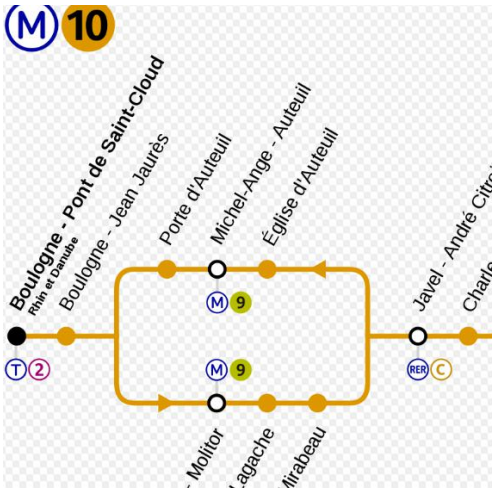
- **private final int tblSize :** la taille du tableau.
- **private final String dataFile :** le fichier de configuration.
- **private final int tbl[ ][ ] :** le tableau pour le constructeur de notre matrice.

Constructeur :

- **public TblStations(String dataFile)**
  this.dataFile = dataFile
  this.tblSize = stationCount() + 1
  this.tbl = new int[this.tblSize][this.tblSize]

Méthodes :

- **public int[][] getTbl() :** Retourne le tableau de temps.
- **private int stationCount() :** Compte le nombre de station.
- **public void fillTravelTime() :** Permet de remplir le tableau "tbl". 0 signifie la distance entre deux stations identiques, -1 signifie qu'il n'y a pas de temps entre les deux stations. Enfin, si la distance entre la station i et j est définie, alors contenu case [i][j] = contenu case [j][i]. Excepté pour une partie de la ligne 7b et de la ligne 10, où nous avons un trajet différent en fonction du sens de circulation (un seule sens possible, exemple en photo).



c) Classe TblDijkstra

L'algorithme de Dijkstra est contenu dans l'objet "TblDijkstra". Il contient tout ce qui est nécessaire à l'algorithme et les résultats de celui-ci. (346 lignes)

Variables :

- **Matrice M :** la matrice de temps entre les stations.
- **int departure, arrived :** le numéro de la station de départ et celui d'arrivée.
- **int []verif :** un tableau de vérification de chaque station pour savoir si le plus court temps pour l'atteindre a été trouvé, permet de ne plus vérifier une station une fois que le plus court temps a été trouvé. -1 signifie non-traité.
- **int []dist :** Résultat de l'algorithme, le tableau des plus courts temps entre la station de départ et les autres. -1 signifie la distance infinie.
- **int []father :** le tableau des pères (numéro de station) de chaque station obtenu par l'algorithme de Dijkstra. Une station A est père d'une station B si l'on doit passer par A pour obtenir le plus court temps de B. -1 signifie que la station n'a pas de père.
- **int []way :** notre chemin sous forme de tableau contenant le numéro des stations, les indices correspondent à l'ordre dans lequel on doit prendre les stations (0→1→2...)
- **NTREATED (-1) :** correspond soit à la distance infinie, soit signifie que l'on peut encore traiter la station dans l'algorithme, soit père inconnu.

Dans chacun des tableaux l'indice correspond au numéro de la station.

Constructeur :

- **public TblDijkstra(Matrice A) :** A est la matrice de temps entre toutes les stations, lors de la création toutes les stations sont déclarées non-traitées (NTREATED) dans chacun des tableaux. Un non-traité dans le tableau de vérification signifie que le plus court temps entre la station de départ et celle d'arrivée n'a pas été trouvé, dans le tableau de distance cela signifie que la station n'a pas encore été rencontrée dans l'algorithme (équivalent de distance infinie), dans celui des pères cela signifie que la station n'a pas de père.

Méthodes :

- **public int getTmpTotal() :** renvoie le temps de notre trajet recherché.
- **public int[] getWay() :** renvoie notre tableau de chemin.
- **public void calcul(int at, int to) :** fait appel aux autres fonctions pour remplir notre tableaux des plus courts temps, notre tableau des pères et notre chemin de "at" à "to".
- **private void treatment(int vertex) :** actualise les temps du tableau "dist". Vérifie entre notre station (vertex) et les stations voisines si le temps trouvé (dist[origine] [vertex] + M[vertex][voisin]) est plus court que celui dans le tableau "dist" ou si la station n'a pas était rencontrée auparavant (-1). Si c'est le cas, on remplace la valeur.
- **private int min() :** renvoie le numéro de station ayant le temps de trajet le plus court depuis le départ.
- **private boolean finished() :** renvoie vrai si tous les sommets ont été traités.
- **private int[] CalculWay() :** renvoie et calcul le chemin en remontant successivement pas les stations pères depuis la station d'arrivée jusqu'à celle de départ.

d) Classe Station.java

La classe 'Station.java' implémente les structures de données pour stocker les stations, les lignes de métros et le métro parisien dans sa globalité (108 lignes).

Variables :

- ➔ **private final String name :** nom de la station.
- ➔ **private final int number :** numéro de station.
- ➔ **private final boolean isTerminus :** Indique si la station est un terminus.

Constructeurs :

- ➔ **public Station() :** Constructeur par défaut.
- ➔ **public Station(String name, int number, boolean isTerminus)**

Méthodes :

- ➔ **public String getName() :** Retourne le nom de la station.
- ➔ **public int getNumber() :** Retourne le numéro de la station.
- ➔ **public boolean getIsTerminus() :** Retourne si oui ou non la station est un terminus.
- ➔ **public String toString() :** Retourne une String avec le nom, le numéro, et le booléen 'isTerminus' de la station.
- ➔ **public boolean equals(Station other) :** Retourne un booléen indiquant si les deux stations sont égales.
- ➔ **public Station copy() :** Retourne une copie de la station.

e) Classe Travel.java

La classe "Travel.java", représente un voyage entre deux stations. Cette classe est utilisé dans les méthodes pour connaître les directions dans les lignes (162 lignes).

Variables :

- ➔ **private Station stationStart, stationStop :** La station de départ et la station d'arrivée, sans prendre en compte la direction.
- ➔ **private int time :** Le temps de voyage entre ces deux stations.

Constructeurs:

- ➔ **public Travel() :** Le constructeur par défaut.
- ➔ **public Travel(Station stationStart, Station stationStop, int time).**

Méthodes:

- ➔ **public int getNumStationStart() :** Retourne le numéro de la station de départ.
- ➔ **public Station getStationStart() :** Retourne la station de départ.
- ➔ **public void setStationStart(Station stationStart) :** Modifie la station de départ.

- ➔ **public int getNumStationStop() :** Retourne le numéro de la station d'arrivée.
- ➔ **public Station getStationStop() :** Retourne la station d'arrivée.
- ➔ **public void setStationStop(Station stationStop) :** Modifie la station d'arrivée.

- ➔ **public int getTime() :** Retourne le temps de trajet.

- ➔ **public void setTime(int time) :** Modifie le temps de trajet.

- ➔ **public Travel switchStation() :** Inverse la station de départ avec la station d'arrivée.
- ➔ **public boolean equals(Travel other) :** Retourne un booléen indiquant si les deux trajets sont égaux.
- ➔ **public String toString() :** Retourne une String avec les informations sur les deux stations et le temps du trajet.

f) Classe MetroLine.java

La classe "MetroLine.java" représente une ligne de métro avec ses stations et les trajets (132 lignes):

Variables :

- ➔ **private String name :** Le nom de la ligne de métro.
- ➔ **private ArrayList < Station > listStation :** La liste des stations présentes dans la ligne.
- ➔ **private ArrayList < Travel > listTravel :** La liste des trajets triés de manière successive excepté les lignes 7, 7b, 10 et 13.

Constructeur :

- ➔ **public MetroLine()**

Méthodes :

- ➔ **public String getName() :** Retourne le nom de la ligne.
- ➔ **public void setName(String newName) :** Change le nom de la ligne.
- ➔ **public ArrayList < Station > getListStation() :** Retourne la liste des stations.
- ➔ **public ArrayList < Travel > getListTravel() :** Retourne la liste des trajets.
- ➔ **public void setListTravel(ArrayList < Travel > _new) :** Modifie la liste des trajets.

- ➔ **public void addStation(Station _new) :** Ajoute une nouvelle station à la liste de station.
- ➔ **public void addTravel(Travel _new) :** Ajoute un nouveau trajet à la liste des trajet.

- ➔ **public void printStation() :** Affiche toutes les stations de la ligne.
- ➔ **public void printTravel() :** Affiche tous les trajets de la ligne.

g) Classe Metro.java

La classe "Metro.java" manipule les classes précédentes et représente la totalité de notre métro :

Variables :

- ➔ **private final MetroLine[] metro :** Le tableau de ligne de métro.
- ➔ **private final String dataFile :** Le nom du fichier de configuration.

Constructeur :

- **public Metro(String dataFile) :** Initialise le tableau métro avec une taille de 16 (14 lignes + 3b et 7b).

Méthodes :

- **private Station whatStation(int numStation) :** Retourne la Station correspondant au numéro.
- **public int convertNameToNumStation(String nameStation) :** Retourne le numéro d'une station correspondant à son nom.
- **private String whatMetroLine(int numStation) :** Retourne le nom de la ligne de métro sur laquelle est la station caractérisée par son numéro.

- **private int strLineToInt(String line) :** Convertit le nom de la ligne en indice pour avoir ça position dans le tableau.
- **private String intToStrLine(int line) :** Convertit l'indice de la ligne en son nom.

- **public void initLine() :** Initialise la liste de stations et la liste de trajets.
- **public void initLineStation() :** Initialise la liste de stations.
- **public void initLineTravel() :** Initialise la liste de trajets

- **public void sortTravel() :** Trie toutes les listes de trajets.
- **public void sortIndexTravel(int index) :** Trie la liste de trajets de la ligne "index". On part du premier trajet que l'on voit et on l'ajoute dans la liste chaînée. Puis on regarde le trajet suivant et on essaie de l'aligner avec le précédent trajet. Exemple :
  premier trajet : A→B, deuxième : B→C, troisième : X→A quatrième : D→C
  alignement du premier et du deuxième : A→B→C
  alignement avec le troisième : X→A→B→C.
  alignement avec le dernier en inversant D→C : X→A→B→C→D
  remarque on ne peut savoir si la ligne triée sera : X→A→B→C→D
  ou D→C→B→A→X

- **public String knowDirection(int[] way, int pos, int numLine, int wayLength)** : Retourne la direction à prendre pour la suite du trajet, en fonction du tableau de station à emprunter "way", de la position où l'on se situe "pos" et du numéro de la ligne "numLine".
- **public String knowDirectionRight(int numStationStart, int numStationStop, int numLine) :** Retourne la direction (terminus) de numStationStart à numStationStop.
  Imaginons la ligne contenant les stations : A,B,C,D,E.
  numStationStart = C et numStationStop = D.
  En comparant la liste de "travel", si on trouve le "travel" (C→D), alors la liste est triée comme ceci : A→B→C→D→E. La méthode renvoie la dernière station.
  Si on ne l'a pas trouvée, alors la liste est triée comme cela : E→D→C→B→A . On renvoie la première station.
  Dans tous les cas, on trouve bien la direction E.

- ➔ **public String knowDirectionFork(int[] way, int pos, int wayLength) :** Retourne la direction si la ligne n'est pas droite (fourche et boucle), en fonction du tableau de station à emprunter "way", de la position où l'on se situe "pos". On lit dans le fichier chaque transition en partant de là où nous sommes et en bloquant la transition pour revenir en arrière. On s'arrête lorsque l'on tombe sur un terminus. Elle fait appel à la méthode fillTbl() ce qui nous assure de ne pas avoir un terminus d'une autre ligne.

- ➔ **private String[] fillTbl(String metroLine) :** Retourne un tableau de String composé de tous les voyages possibles sur la ligne "metroLine".

- ➔ **public void printTravelDetail(int way[], int wayLength, String time) :** Prépare le contenu à écrire dans le fichier "UserTravel.txt", pour la méthode writeDataFile(). A
- ➔ **public void writeDataFile(String UserTravelFile, String[] data) :** Écrit un fichier avec les informations du voyage utilisé. Sert pour l'affichage de la partie graphique.
- ➔ **public void printStation() :** Affiche le trajet que doit prendre l'utilisateur et fait appel à writeDataFile() pour écrire les informations'.
- ➔ **public void printIndexStation(String index) :** Affiche toutes les stations dans la ligne qui est située dans le tableau à la position 'index'.
- ➔ **public void printTravel() :** Affiche tous les trajets.
- ➔ **public void printIndexTravel(String index) :** Affiche tous les trajets possibles sur la ligne qui est située dans le tableau à la position 'index'.

Comment s'articule le tout ?

La méthode **printTravelDetail()** récupère les informations produites par l'algorithme de dijkstra, affiche les stations en fonction de leur numéro ainsi que les lignes de métros via le même procédé. Elle est aussi en charge de trouver et d'afficher les terminus. Les structures "Station" et "Travel" sont encapsulées dans "MetroLine", qui est encapsulée dans la structure "Metro".

h) Classe Fenetre.java

La classe "Fenetre" permet de créer une fenêtre graphique de la taille souhaitée dans laquelle on appelle la classe" Panneau" permettant de faire notre affichage graphique.

i) Classe Panneau.java

La classe "Panneau.java" contient tous les éléments permettant d'afficher le contenu de notre fenêtre. On y trouve donc la taille des caractères ainsi que la police et les images de font.

Méthodes :

- ➔ **drawTravel(Graphics g, int police, int imgHeight, Font font) :** permet d'afficher la station de départ, celle d'arrivée, les changements de ligne, les directions ainsi que le temps de trajet. Ces données sont lues dans le fichier "UserTravel.txt" grâce à la méthode readUserData().
- ➔ **readUserData(String userDataFile) :** lit les données dans "UserTravel.txt"

j) Classe Menu.java

La classe Menu.java permet d'appeler toutes les classes pour pouvoir exécuter un trajet. Grâce au menu l'utilisateur peut utiliser quatre commandes:

- ✔ Help: pour afficher toute les commandes disponibles.
- ✔ Random Travel: effectue un trajet entre deux stations aléatoires.
- ✔ Classic Travel: effectue un trajet choisit par l'utilisateur.
- ✔ Exit: pour quitter le programme.

Si la commande n'est pas reconnue alors le programme affichera "commande inconnue"

Variables :

- ➔ **Metro metro :** structure contenant toutes les lignes de métro.
- ➔ **TblStations ts :** tableaux des stations.
- ➔ **TblDijkstra td :** Résultats de l'algorithme de Dijkstra.

Constructeur :

- ➔ **Menu(String dataFile) :** initialise métro et les deux tableaux, en lisant le fichier metroL.txt.

Méthodes :

- ➔ **run() :** Analyse la saisie de l'utilisateur, convertie les noms des stations en numéro.

La classe Main.java est la classe principale où la classe Menu est appelée pour effectuer le programme. Cette classe inclue le fichier metroL.txt où sont répertoriées toutes les stations de métro ainsi que les temps de trajet entre elles.

III)Explication du code

Le début du code commence dans la partie Main.java, la structure Menu est crée à partir du nom du fichier contenant les stations avec : leur numéro, leur ligne et le temps entre chaque station. Le constructeur de Menu fait appel aux constructeur de Metro, tblStations, tblDijkstra. L'objet tblStations lit notre fichier et créer la matrice des temps entre les stations, celle-ci est pour le moment vide mais comme un tableau est un pointeur, on peut déjà le donner à tblDijkstra. On retourne dans le fichier "Main.java" qui lance la méthode **initMenu()**, faisant appel aux méthodes :

- **initline()** : Cette méthode fait appel à trois sous méthodes permettant de remplir le contenu de notre objet "Metro". L'objet "Metro" contient un tableau de "metroLine" et chaque "metroLine" contient une liste de "Travel" et une liste de "Station".
  - la première méthode est **initLineStation()** lit dans le fichier toutes les stations, récupère leur nom (name), leur numéro de station (strNumber), leur numéro de ligne (strLine) et détermine si ce sont des terminus. Pour finir elle les ajoute à notre liste de station se trouvant dans la variable "metro".
  - La seconde méthode est **initLineTravel()** qui lit les temps entre les stations et créer nos listes de "travel". La ligne lue est stockée dans "line", on la sépare en 4 en fonction du caractère "espace". Chaque partie est stockée dans le tableau "parts" contenant : E, station de départ, station d'arrivé et le temps entre les deux stations. On complète ensuite notre objet de type "travel" grâce au contenu de "parts" et on l'ajoute à la liste si et seulement si les deux stations sont sur la même ligne de métro.
  - Enfin la dernière méthode, **sortTravel()**, trie les trajets de sorte à avoir les stations dans l'ordre (identique à celui sur la carte de la ligne de métro dans la vie réelle) avec un

terminus au début et à la fin (uniquement pour les lignes droites, et la lignes 7b). Voir explication dans la première partie.

- **fillTravelTime()** : Cette méthode remplie la matrice contenue dans notre objet tblDijkstra avec les temps de trajet (le détail étant plus haut). Cette méthode prend en compte les lignes qui n'ont que un seule sens de circulation (7b, 10...) à ce moment là on remplit uniquement la case [i][j] de la direction, dans les autres cas contenu [i][j] = contenu [j][i].

On utilise ensuite dans "Menu.java", la fonction **run()** stockant la commande utilisateur dans la variable "choice". Dans le cas où il demande de l'aide, la liste des saisies possibles est affichée. Dans le cas où il demande un trajet aléatoire, on détermine un trajet entre deux stations quelconque. Dans le dernier cas on invite d'abord l'utilisateur à saisir sa station de départ (contenue dans start) puis sa station d'arrivée (contenue dans stop). Grâce aux fonctions **convertNameToNumStation()** on récupère les numéros de stations correspondantes. Les deux boucles for-loope (ligne 124 et 128 de Menu.java) sont là pour gérer les cas avec plusieurs stations qui ont le même nom (ligne différente, ex : bastille). La première boucle pour celle de départ la seconde pour celle d'arrivée. Ces boucles font appel à **WhatStationEquals()** qui renvoie un tableau des numéros de station avec le même nom. A chaque fois on utilise la méthode **Calcul()** (Dijkstra) pour tester tous les cas possibles et on stocke le temps du chemin le plus court et ce chemin respectivement dans "minTime" et "way". Par exemple pour la station "Opéra", il y a quatre ligne de métro possible, on cherche à partir de laquelle le chemin est le plus court.

La méthode **Calcul()** dans la classe "TblDijkstra" permet de trouver notre plus court chemin. Elle vérifie en premier que "departure" vaut -1, si c'est le cas alors il n'y a pas eu de demande de trajet avant et toutes les variables sont déjà initialisées. Dans le cas contraire on les réinitialise avec la méthode **reinitialized()**. Ensuite seul le temps entre les stations voisines et celle de départ est ajouté au tableau dist, les autres valeurs sont à -1 (distance infini) grâce à l'initialisation dans le constructeur. Si il existe un temps entre les deux stations alors la deuxième station aura comme père la station de départ. Je rappel que la station de départ n'a pas de père (vaut -1). Tout ceci est fait dans la première boucle for. Puis la station de départ est déclarée traitée dans le tableau de vérification (dès que une station est traitée on ne touche plus à la valeur dans le tableau dist car c'est la plus petite valeur possible).

Tant que toutes les stations n'ont pas été traitées (boucle while) et qu'il existe au moins une station atteignable ont continu de calculer les plus courts temps. Nous cherchons à chaque itération la station ayant le temps le plus court avec l'origine (appelé index_min, renvoyé par **min()**) parmi les stations non-traitées. A partir de cette station on utilise la méthode **treatment()** qui actualise les temps entre notre station et les stations voisines si dist[origine][station] + M[station][voisin] (M notre matrice de temps initiale) est plus court que celui de dist[origine][voisin], ou si la station n'a pas été rencontrée auparavant. Si le temps d'une station est actualisé alors la station trouvée aura comme père le numéro de index_min. Une fois tous ses voisins vérifié, index_min est déclarée TREATED dans le tableau "verif".

Dès que tous les sommets sont traités, la méthode **CalculWay()** renvoie notre plus court chemin. Pour se faire, elle remonte les pères depuis la station d'arrivée jusqu'à la station qui n'a pas de père (notre station de départ), ceux-ci sont stockés dans le tableaux "way" que l'on inverse pour obtenir notre chemin.

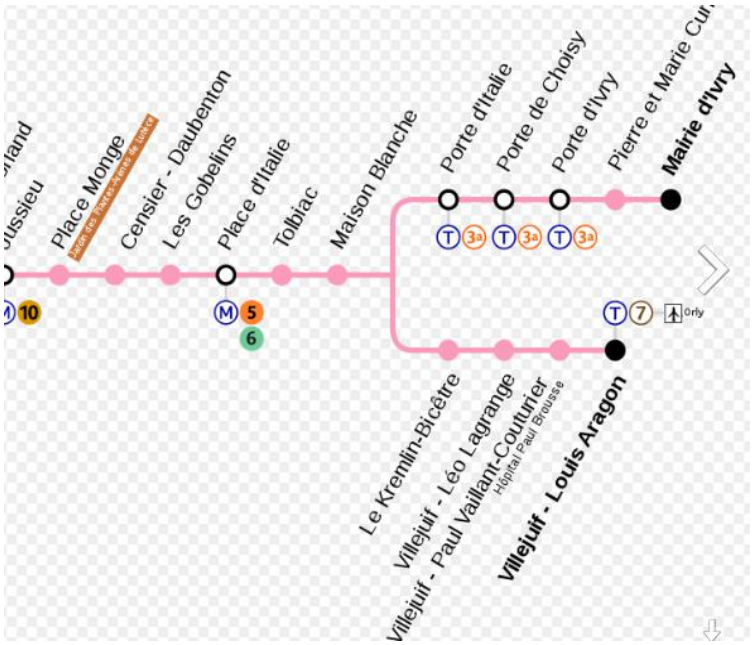
Pour finir, on fait appel à la méthode **printTravelDetail()** qui affiche le trajet dans le terminal et créer le fichier "UserTravel.txt" comprenant les directions à suivre, ce fichier sera lu lors de l'affichage de la fenêtre graphique. L'écriture suit la nomenclature :

- S + nom de la station de départ + numéro de ligne
- D + nom de la station pour la direction + numéro de ligne

- C + nom de la station de changement + numéro de ligne
- E + nom de la station d'arrivé + numéro de ligne
- T + temps du trajet.

Pour connaître les directions de notre trajet, on utilise la méthode **KnowDirection()** dans "Metro.java" qui en fonction de la ligne fait soit appel à :

- **knowDirectionRight()** : cas des lignes sans fourches. On met dans tmp la liste des stations de la ligne triée. Dans la boucle for, on essaye de chercher notre couple de station donné par le chemin de l'algorithme de Dijkstra (numStationStart et numStationStop). Si on le trouve cela veut dire que la liste est triée dans la bonne direction, le terminus est la dernière station de la liste. Sinon (direction vaut 0, rien trouvé) alors la liste est triée dans le sens inverse, le terminus est la première station de la liste. Dans les deux cas ce terminus est la direction recherchée. Voir dans la partie 1 si besoin de plus d'explications.
- **knowDirectionFork()** dans le cas où la ligne se sépare, voir photo :

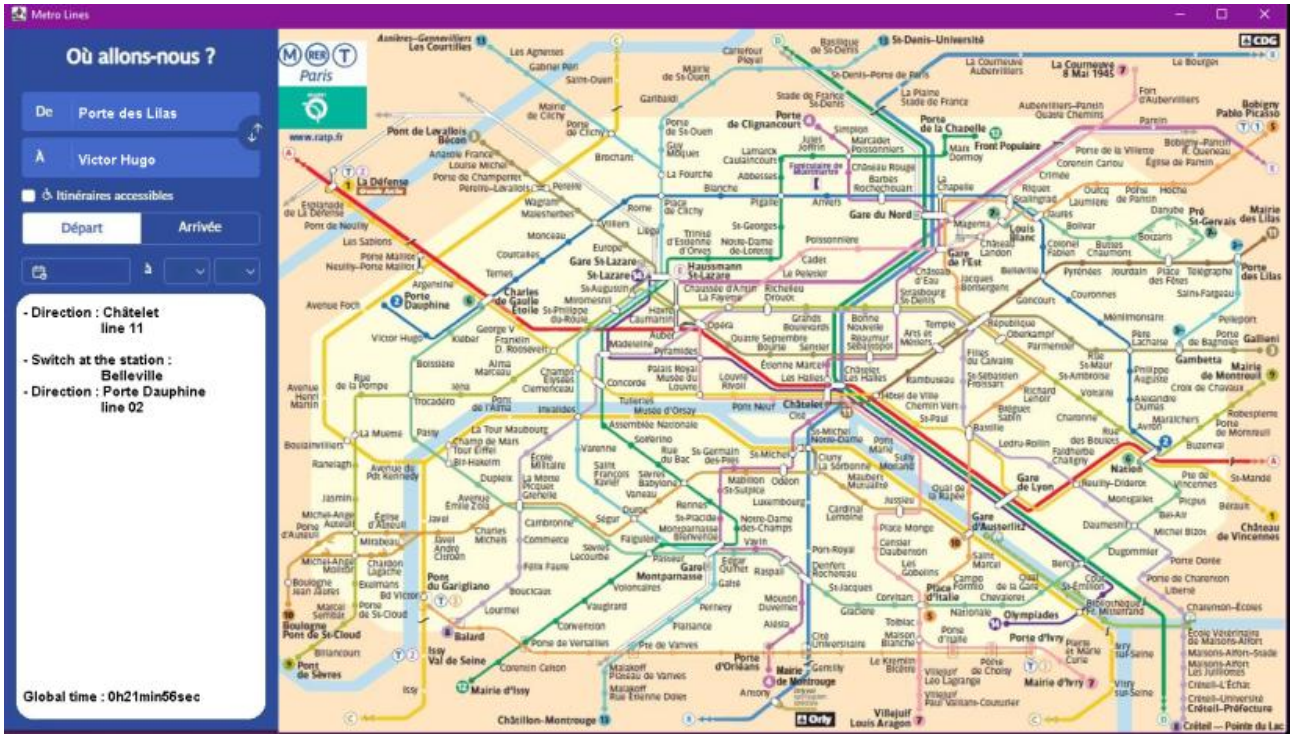


La première partie la boucle 'while' permet de parcourir la liste de stations dans "way" (notre chemin) afin de se rapprocher au maximum du terminus tous en restant sur la même ligne. Cela permet de se placer sur le bon côté de la fourche (si way ne quitte pas la ligne), à contrario si on s'arrête avant cette dîtes fourche ou si il y a un changement de ligne, la direction importe peut, on peut se diriger soit vers le terminus d'une ou de l'autre partie de la fourche. (Sur notre exemple si on part de Place Monge et que l'on change de ligne à Place d'italie, on peut prendre la direction mairie d'Ivry ou Villejuif - Louis Aragon). On récupère tous les trajets possibles de la ligne via **fillTbl()**. Chaque ligne de tbl est sous la forme "E stationDépart stationArrivée temps". On définit 'stop' comme la station d'arrêt et "lastSeen" comme la dernière station vue. Puis on boucle sur ce tableau tant que l'on a pas trouvé de terminus. A chaque itération on regarde si "stop'"ou "lastSeen" est un terminus. Si ce n'est pas le cas on vérifie l'existence d'une transition, partant de "stop" mais n'arrivant pas à "lastSeen" ou inversement. Dans ce cas on décale "stop" et "lastSeen" sur les deux stations de la transition (on a avancé d'une station). "lastSeen" nous évite de prendre le mauvais sens et de tomber sur le mauvais terminus. La fonction retourne une chaîne de caractère indiquant le nom de la direction et sa ligne.

A la fin de **printTravelDetail(),** on créer la fenêtre graphique qui créer notre panneau (variable setContentPane de la classe fenetre). Celui-ci lance toutes les méthodes, en particulier readUserData(), qui lit les informations du trajet stockés dans "UserTravel.txt" pour les afficher.

IV) Instructions pour exécuter le programme

Dans cette partie nous expliquons comment utiliser le programme. Assurez-vous de disposer de la dernière version de Java sinon des erreurs peuvent se produire lors de la compilation au niveau de la syntaxe. Entrez dans le dossier projet et tapez make dans le terminal, attendre que les fichiers compilent. Tapez Help pour voir la liste des commandes possibles. Attention aux majuscules. Tapez "Classic Travel", puis le nom de la station de départ et enfin la station d'arrivée. Voici une image de ce que l'on obtient après avoir entré le nom de nos deux stations.
Remarque : il y a un script "test_voyages.sh" qui prend en argument un nombre de répétition pour générer des itinéraires aléatoires. Pour l'utiliser : Tapez make compil sur le terminal puis ./test_voyages.sh (nombre de votre choix). Celui-ci n'affiche pas la fenêtre graphique, les réponses sont écrites dans un fichier save.txt.



Il y a cependant un problème d'affichage parmi les membres du groupe, certains étant sur Windows et d'autre sur Linux pour coder. La description sort du cadre, voir seconde photo :

Partie Adrien LAVALLIERE :

- Gestion des structures de type matrices
- Implémentation de l'algorithme de Dijkstra

Partie Mickaël LE DENMAT :

- Lire le fichier de configuration afin de créer une matrice avec toutes les stations et les distances pour pouvoir appliquer Dijkstra.
- Création des structures de données pour stocker les stations, les voyages, les lignes de métros et le métro dans sa globalité.

Partie de Florian CAMBRESY :

- Partie graphique