

IN403 - Algorithmique de graphes

Rapport de projet - Métro 2020

05/05/2020

Gabriel Dos Santos et Raphaël Lin

Sommaire

- 1. Introduction**
- 2. Utilisation de l'application**
 - 2.1. Pré-requis
 - 2.2. Exécution
 - 2.3. Utilisation
- 3. Description du travail réalisé**
 - 3.1. Modifications apportées au fichier de configuration
 - 3.2. Choix du langage
 - 3.3. Répartition du travail
 - 3.4. Résumé détaillé du fonctionnement de l'application
- 4. Description des algorithmes et classe utilisés**
 - 4.1. Représentation du graphe
 - 4.2. Implémentation de l'algorithme de Dijkstra
 - 4.3. Recherche des terminus
 - 4.4. Affichage du trajet
 - 4.5. Résolution des problèmes rencontrés
- 5. Pour aller plus loin**
 - 5.1. Idées pour un affichage graphique
 - 5.2. Comment améliorer l'application ?

1. Introduction

L'objectif de ce projet est d'écrire une application, dans le langage de programmation de notre choix, permettant de déterminer le plus court itinéraire possible pour se rendre d'une station à une autre dans le métro Parisien.

L'utilisateur doit simplement saisir une station de départ et une station d'arrivée.

L'application se charge alors de calculer l'itinéraire le plus court à travers les différentes lignes du métro de Paris et affiche le détail du parcours à l'écran.

Afin de réaliser ce projet, nous avons à disposition un fichier texte de configuration contenant l'ensemble des stations parisiennes (mis à part quelques exceptions pour les extensions les plus récentes) ainsi que la liste des transitions entre chaque station. Notre travail a donc été de représenter ce système de transport en commun en un graphe sur lequel nous pourrions appliquer un des algorithmes de recherche du plus court chemin vu en cours d'IN403.

Pour un guide rapide de l'utilisation du programme, nous vous invitons à consulter le README.md en annexe de ce rapport.

2. Utilisation de l'application

2.1. Pré-requis

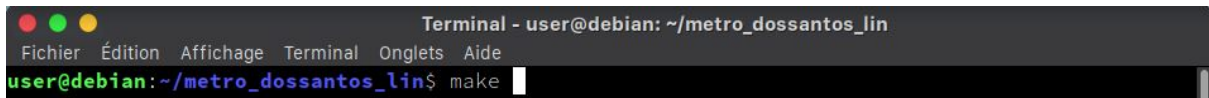
Pour utiliser cette application, il faudra avoir installé le package python3 que vous pouvez trouver ici :

<https://www.python.org/downloads/release/python-382/>

Python3 est généralement déjà installé sur les machines Linux et est présent sur la machine virtuelle de l'UVSQ.

2.2. Exécution

Comme expliqué dans le fichier README.md contenu dans l'annexe de ce rapport, pour utiliser l'application il vous suffit de taper la commande “make” dans un terminal ouvert sur le dossier “metro_dossantos_lin” :



```
Terminal - user@debian: ~/metro_dossantos_lin
Fichier  Édition  Affichage  Terminal  Onglets  Aide
user@debian:~/metro_dossantos_lin$ make
```



```
Terminal - user@debian: ~/metro_dossantos_lin
Fichier  Édition  Affichage  Terminal  Onglets  Aide
user@debian:~/metro_dossantos_lin$ python3 metro.py
```

Vous pouvez également exécuter le programme manuellement depuis le même dossier en tapant la commande “python3 metro.py” :

2.3. Utilisation

Lorsque l'application se lance, il vous sera proposé de saisir la station où vous vous trouvez. Tapez simplement son nom (prenez garde aux majuscules et n'hésitez pas à vous référer au fichier "metro.txt") puis cliquez sur la touche "Entrée" pour que le programme l'enregistre. Il vous sera ensuite demandé d'indiquer la station où vous souhaitez vous rendre. De même, saisissez le nom de votre destination et appuyez sur "Entrée".

Si vous avez fait une faute de frappe ou bien que la station entrée n'est pas dans le fichier de configuration, l'application vous affichera le message suivant dans le terminal (ce message apparaîtra aussi bien pour la saisie de la station de départ que pour celle d'arrivée) :

```
Terminal - user@debian: ~/metro_dossantos_lin
Fichier Édition Affichage Terminal Onglets Aide
user@debian:~/metro_dossantos_lin$ make
python3 metro.py

Station de départ :
>> arts et métiers
La station n'a pas été trouvée dans la base de données !
Vérifiez que vous n'avez pas fait de fautes de frappe.

Station de départ :
>> 
```

Si vous n'avez pas fait d'erreurs lors de la saisie, l'application calculera l'itinéraire le plus court pour votre trajet et l'affichera comme suit dans le terminal :

```
Terminal - user@debian: ~/metro_dossantos_lin
Fichier Édition Affichage Terminal Onglets Aide
user@debian:~/metro_dossantos_lin$ make
python3 metro.py

Station de départ :
>> Arts et Métiers
Station d'arrivée :
>> Porte de Vincennes

Un itinéraire a été trouvé !
Vous êtes à Arts et Métiers. Prenez la ligne 11, direction Châtelet
Arrivé(e) à Hôtel de Ville, prenez la ligne 01, direction Château de Vincennes
Vous arriverez à Porte de Vincennes dans : 11 min 48 s
```

3. Description du travail réalisé

3.1. Modifications apportées au fichier de configuration

Afin de pouvoir effectuer la recherche du plus court chemin, nous avons dû modifier le fichier de configuration en ajoutant pour chaque station les informations qui n'y figuraient pas. Nous avons donc ajouté pour chacune des 376 stations leur numéros de ligne respectifs et, soit un 0 pour les stations classiques, soit un 1 pour les terminus. Pour uniformiser le fichier de configuration, nous avons modifié les numéros des stations dans les transitions pour qu'elles comportent 4 chiffres.

Ci-dessous, un exemple du fichier de configuration avant et après modification :

```
< > metro.txt x
1 # stations et liens du métro parisien
2 # format : V num_sommet nom_sommet
3 #           E num_sommet1 num_sommet1 temps_en_secondes
4 V 0000 Abbesses
5 V 0001 Alexandre Dumas
6 V 0002 Alma Marceau
7 V 0003 Alésia
8 V 0004 Anatole France
9 V 0005 Anvers
10 V 0006 Argentine

380 E 0 238 41
381 E 0 159 46
382 E 1 12 36
383 E 1 235 44
384 E 2 110 69
```

Fichier de configuration avant modification

```
< > metro.txt x
1 # stations et liens du métro parisien
2 # format : V num_sommet ligne_sommet station/terminus nom_sommet
3 #           E num_sommet1 num_sommet1 temps_en_secondes
4 V 0000 12 0 Abbesses
5 V 0001 02 0 Alexandre Dumas
6 V 0002 09 0 Alma Marceau
7 V 0003 04 0 Alésia
8 V 0004 03 0 Anatole France
9 V 0005 02 0 Anvers
10 V 0006 01 0 Argentine

380 E 0000 0238 41
381 E 0000 0159 46
382 E 0001 0012 36
383 E 0001 0235 44
384 E 0002 0110 69
```

Fichier de configuration après modification

3.2. Choix du langage

Pour réaliser ce projet, le langage de programmation que nous avons choisi est le Python. Nous l'avons retenu principalement pour sa syntaxe simple et facile à lire. Le Python est un langage de haut niveau qui permet notamment de programmer en orienté objet, rendant aisée la création de classes complexes. L'utilisation des méthodes de classes nous a ainsi permis de manipuler les données contenues dans le fichier de configuration avec une grande flexibilité. Ce langage comprend également un ramasse-miettes, ce qui nous permet de ne pas avoir à nous soucier de la gestion de la mémoire et ainsi nous concentrer pleinement sur la mise en place des algorithmes. Enfin, le Python étant un langage portable, il est simple à mettre en place sur tout type de plate-formes et permettrait potentiellement à l'application d'être exportée sur smartphone.

3.3. Répartition du travail

Afin de réaliser le projet, nous nous sommes partagés le travail.

Parties réalisées par Gabriel :

- Lecture du fichier de configuration et récupération de ses données
- Initialisation de la classe
- Implémentation de l'algorithme de Dijkstra
- Récupération du trajet le plus court et de sa distance (partie commune)

Parties réalisées par Raphaël :

- Récupération des stations de départ et d'arrivée de l'utilisateur
- Récupération du trajet le plus court et de sa distance (partie commune)
- Implémentation de l'algorithme de recherche des terminus
- Gestion de l'affichage dans le terminal

Nous avons néanmoins discutés de chacune de ces sections et nous avons tous deux apporté certaines modifications à des parties écrites par l'autre lorsque c'était nécessaire. De même, ce rapport a été écrit conjointement par le binôme.

3.4. Résumé détaillé du fonctionnement de l'application

Dès que vous lancez l'exécution du programme, la fonction `main()` appelle le constructeur de la classe `MetroParisien`. Celui-ci ouvre et lit le fichier de configuration afin d'initialiser la classe.

Une fois ceci fait, la fonction `getStart()` est appelée et vous invite à entrer la station où vous vous trouvez. Tant que votre réponse n'est pas reconnue, le programme vous affichera un message d'erreur tel que vous avez pu le voir plus haut dans ce rapport et rappellera `getStart()` pour que vous puissiez indiquer la station de départ.

Dès que celle-ci est valide, le programme appelle la fonction `getDestination()` qui vous demande de rentrer la station où vous souhaitez vous rendre. De même que pour `getStart()`, cette fonction affiche un message d'erreur si la station entrée n'est pas reconnue et est ensuite rappelée par le programme jusqu'à avoir une réponse valide.

Après avoir récupéré vos stations de départ et d'arrivée, l'application lance la fonction `ShortestPath()`. Celle-ci appelle la méthode de classe `dijkstra()` qui est l'algorithme permettant de calculer le chemin le plus rapide à travers le métro parisien pour vos stations de départ et d'arrivée (nous entrerons dans le détail de cet algorithme dans la section 4.1 de ce rapport).

A la fin de son exécution, `dijkstra()` lance les fonctions `getTime()` et `getTravel()` qui récupèrent respectivement le temps de trajet pour les stations de départ et d'arrivée données, puis le détail de ce trajet. Enfin, la fonction `Terminus()` permet de déterminer dans quelle direction doivent être prises les lignes de métro. L'itinéraire à suivre est ensuite affiché à l'écran, dans le terminal, comme illustré dans la section 2.3 de ce rapport.

4. Description des algorithmes et classe utilisés

4.1. Représentation du graphe

Afin de pouvoir utiliser les informations du fichier de configuration et représenter le métro parisien par un graphe, nous avons décidé d'utiliser une des fonctionnalités qui rend le Python un langage orienté objet : les classes. Nous avons donc créé une classe appelée `MetroParisien` qui possède deux attributs, la liste `stations` et la liste `transitions` :

```
9 class MetroParisien:
10 # ATTRIBUTS
11     # Liste des stations
12     stations = []
13     # Liste des transitions
14     transitions = []
15
```

Cette classe est initialisée par la méthode suivante :

```
17 # CONSTRUCTEUR
18     # Initialisation d'une instance de MetroParisien
19     def __init__(self, filename):
```

Elle prend en paramètre un nom de fichier qui sera celui du fichier de configuration. Lors de la création d'une instance de la classe, cette fonction commence par ouvrir le fichier de configuration. On déclare ensuite une variable "line" qui va nous permettre de stocker le contenu du fichier ligne par ligne. Dans une boucle `while(True)`, on lit d'abord une ligne du fichier avec la méthode `readline()` (on utilise également la méthode `rstrip()` pour supprimer les "\n" en fin de ligne). On entre ensuite dans des boucles permettant le traitement de chaque ligne lue :

- Si la ligne lue commence par un "V", on sait que c'est un sommet du graphe, donc une station. On copie le contenu de la ligne dans une variable "vertex" et on supprime le "V" du début de ligne. On applique ensuite la méthode `split()` avec les paramètres suivants : `split(' ', 3)`. Ceci nous permet de diviser la ligne lue au niveau des espaces, un maximum de trois fois. On obtient alors une liste qui contient :
 - à l'indice 0 : le numéro de la station
 - à l'indice 1 : la ligne de métro sur laquelle se trouve la station
 - à l'indice 2 : le type de station (si c'est un terminus ou non)
 - à l'indice 3 : le nom complet de la station

On ajoute cette liste à la liste des stations à l'aide de la méthode `append()` et on utilise le mot-clé "continue" pour arrêter l'itération courante de la boucle `while(True)` afin de lire une nouvelle ligne.

- Si la ligne lue commence par un "E", on sait que c'est une transition du graphe, et donc qu'elle nous indique le temps de trajet entre deux stations. On copie le contenu de la ligne dans une variable "edge" et on supprime le "E" en début de ligne. On applique ensuite la méthode `split()` avec les paramètres : `split(' ', 2)`. On découpe ainsi la ligne de transition à chaque espace rencontré, 2 fois au maximum. On obtient alors une liste qui contient :

- à l'indice 0 : le numéro de la première station
- à l'indice 1 : le numéro de la deuxième station
- à l'indice 2 : le temps en secondes entre ces deux stations

On ajoute cette liste à la liste des transitions à l'aide de la méthode `append()` et on utilise le mot-clé "continue" pour stopper l'itération courante de la boucle `while(True)` afin de lire une nouvelle ligne dans le fichier de configuration.

- Si la variable "line" est vide, c'est que l'on a atteint la fin du fichier. On le ferme et on utilise le mot-clé "break" pour stopper la boucle `while(True)`.

Après cela, on parcourt la liste des stations et on transforme en entier le numéro de chaque station de la liste (cast en int). Ceci nous facilitera la tâche lors de la recherche du plus court chemin.

On parcourt ensuite la liste des transitions, que l'on transforme intégralement en entiers. De même, cela rendra la recherche du plus court chemin plus simple et nous évitera de devoir réaliser des casts au milieu de l'algorithme.

A la fin de ce traitement, notre classe est alors initialisée avec deux listes de listes, l'une contenant les stations, l'autre les transitions entre chacune de ces stations. Le graphe est ainsi représenté par ces deux listes de listes. Ceci nous permet de facilement parcourir les stations car leur indice dans la liste des stations correspond à leur numéro. On peut ainsi très simplement accéder aux informations de la station qui nous intéresse seulement avec son numéro, comme savoir sur quelle ligne de métro elle se trouve, si c'est un terminus, ou quel est son nom (utile pour l'affichage).

4.2. Implémentation de l'algorithme de Dijkstra

Pour résoudre le problème de recherche du plus court chemin, nous avons décidé d'employer l'algorithme de Dijkstra. Bien que n'étant pas l'algorithme le plus performant pour la recherche du plus court chemin, il est fiable et infallible lorsque bien implémenté.

Dans notre cas, l'algorithme de Dijkstra est une méthode de la classe `MetroParisien` qui prend en paramètre le numéro de la station de départ et celui de la station d'arrivée.

On commence par créer une liste vide de la même longueur que la liste "stations", appelée "visited". On crée ensuite une liste des sommets non visités de la même longueur que "stations", initialisée de la façon suivante :

```
164         unvisited = [(i) for i in range(len(self.stations))]
```

Ainsi, on a :

```
unvisited[i] = i
```

Ceci nous permet d'identifier chaque élément de la liste au numéro d'une station. Dès lors, on peut supprimer l'élément de la liste correspondant au numéro d'une station une fois que celle-ci a été vue par l'algorithme.

On crée ensuite une liste "distance" de la même longueur que "stations" et dont on initialise chaque valeur à l'infini. Dans cette liste, on initialise l'indice qui a le même numéro que la station de départ à 0. Cette liste nous permettra de connaître la distance entre la station de départ et la station recherchée. Par exemple, si l'on recherche la distance pour la station n° 275, il nous suffira de récupérer la valeur contenue dans `distance[275]` pour connaître le temps de trajet (en secondes) entre la station de départ et la station n° 275.

On crée une quatrième liste appelée "prev" de la même longueur que la liste des stations et dont on initialise toutes les valeurs à -1. Cette liste contiendra la liste des pères pour chaque sommet. C'est en traversant cette liste jusqu'à revenir à la station de départ que l'on pourra récupérer le trajet entre la station de départ et d'arrivée.

Par exemple, si la station n° 275 a pour père la station n° 301, on aura :

```
prev[275] = 301
```

```
prev[301] = ...
```

Et ainsi de suite jusqu'à revenir à la station de départ.

Une fois toutes nos variables initialisées, nous allons pouvoir appliquer l'algorithme de Dijkstra en lui-même.

On démarre une boucle while dont la condition d'arrêt est la suivante :

```
177 while len(visited) < len(self.stations):
```

Cela revient à dire : tant que l'on a pas visité tous les sommets du graphe, on répète la boucle.

Pour chaque itération dans cette boucle, on doit d'abord déterminer quel sommet a la distance la plus courte par rapport au sommet de départ dans la liste de départ. Pour cela, on utilise la méthode de classe minDistance() qui parcourt la liste "distance" et retourne le numéro du sommet dont la distance est la plus faible. Pour ce moment, minDistance() l'ajoute également à la liste "visited" et le retire de la liste "unvisited". La valeur de retour de minDistance() est alors stockée dans la variable "currentStation".

Connaissant le numéro de la station la plus proche de la station de départ, il nous faut maintenant déterminer quelles sont ses stations voisines en parcourant la liste des transitions. Pour cela, on utilise la méthode de classe getNeighbours() qui parcourt la liste des transitions à la recherche des transitions possibles pour "currentStation" et retourne une liste contenant ces transitions. La méthode getNeighbours() gère également des exceptions dont nous parlerons plus avant dans la section 4.5 du rapport.

Une fois le sommet courant et ses voisins connus, il nous faut parcourir la liste de ses stations voisines. Si la distance de la station voisine est actuellement plus grande que la somme de la distance de la station courante et de sa voisine, alors on met à jour la distance de la station voisine (la nouvelle distance est la somme de la distance de la station courante ainsi que le temps de trajet entre la station courante et la station voisine). De plus, le père de la station voisine devient la station courante. Cette vérification est faite grâce aux boucles if suivantes dans le code :

```
186 if (neighbours[nextStation][0] == currentStation):
187     if (distance[neighbours[nextStation][1]] > distance[currentStation] + neighbours[nextStation][2]):
188         distance[neighbours[nextStation][1]] = distance[currentStation] + neighbours[nextStation][2]
189         prev[neighbours[nextStation][1]] = currentStation
190 if (neighbours[nextStation][1] == currentStation):
191     if (distance[neighbours[nextStation][0]] > distance[currentStation] + neighbours[nextStation][2]):
192         distance[neighbours[nextStation][0]] = distance[currentStation] + neighbours[nextStation][2]
193         prev[neighbours[nextStation][0]] = currentStation
194
```

Lorsque la boucle while() s'arrête, cela signifie que tous les sommets du graphe ont été vus et on peut désormais connaître la distance de n'importe quelle station par rapport à celle de départ.

Afin de vérifier que la station d'arrivée entrée par l'utilisateur n'est pas le noeud de plusieurs lignes de métro, on utilise la méthode de classe `getFinalDest()` qui nous retourne une liste des stations portant le même nom que celle entrée par l'utilisateur.

La méthode `dijkstra()` lance ensuite la fonction `getTime()` afin de traduire la durée de trajet pour la station d'arrivée (stockée en secondes dans la liste "distance") en minutes-secondes.

Enfin, `dijkstra()` retourne le résultat de `getTravel()` qui lit simplement la liste "prev" (listes des pères de chaque station), depuis la station d'arrivée jusqu'à revenir à la station de départ (comme cela a été expliqué plus haut dans cette section).

4.3. Recherche des terminus

Toutes les lignes de métro sont à double sens : l'intérêt ici est d'informer l'utilisateur du sens dans lequel il doit prendre la ligne. Pour cela, on a une fonction `Terminus()`, qui prend en paramètre deux stations de métro d'une même ligne et qui se suivent. En fonction de l'ordre dans lequel elles sont données en paramètre, l'algorithme repère cet ordre et recherche les stations suivantes jusqu'à ce qu'il tombe sur une station terminus. Il renvoie finalement son nom.

Pour ce faire, les stations ont été préalablement annotées dans le fichier de configuration, d'un 0 si la station n'est pas un terminus, ou bien d'un 1 dans le cas contraire. Sur l'image ci-dessous, on peut remarquer par exemple que Château de Vincennes est un des deux terminus de la ligne 01.

67	V	0063	07	0	Château Landon
68	V	0064	04	0	Château Rouge
69	V	0065	04	0	Château d'Eau
70	V	0066	01	1	Château de Vincennes
71	V	0067	01	0	Châtelet
72	V	0068	11	1	Châtelet
73	V	0069	14	0	Châtelet
74	V	0070	04	0	Châtelet
75	V	0071	07	0	Châtelet
76	V	0072	13	1	Châtillon-Montrouge
77	V	0073	04	0	Cité

L'algorithme est une boucle qui s'effectue de la manière suivante :

- Deux stations sont analysées, **previous** étant une station déjà parcourue précédemment et **current** une station en cours de trajet.
- La valeur de **term** vaut 0 si **current** n'est pas un terminus, 1 dans le cas contraire.
- Tant que **term** vaut 0, on effectue la démarche suivante:

- Toutes les transitions du fichier de configuration avec la station **current** sont stockées dans une variable **neighbours**.
- De **neighbours** sont retirées les transitions entre **current** et une station d'une ligne de métro différente, et les transitions avec **previous**.
- Il ne reste plus qu'une seule transition dans **neighbours**, celle entre **current** et la prochaine station à parcourir.
- **previous** prend alors la valeur de **current**, **current** prend la valeur de la prochaine station, et **term** prend la valeur de terminus de cette dernière.

Une exception se trouve sur la ligne 13 : il y a un sens ou deux terminus existent. Il faut donc que l'algorithme détermine lequel de ces deux terminus est le bon. Ci-dessous une image de la portion de la ligne 13 où l'on peut distinguer la séparation de la ligne en deux :



La fonction prend donc un troisième paramètre qui est la liste de toutes les stations du trajet parcouru par l'utilisateur. Si dans ce trajet la station La Fourche est suivie de Guy Môquet, l'algorithme de terminus empêche l'analyse de la transition entre La Fourche et Brochant. Inversement, si la station La Fourche est suivie de Brochant, l'algorithme empêche l'analyse de la transition entre La Fourche et Guy Môquet.

4.4. Affichage du trajet

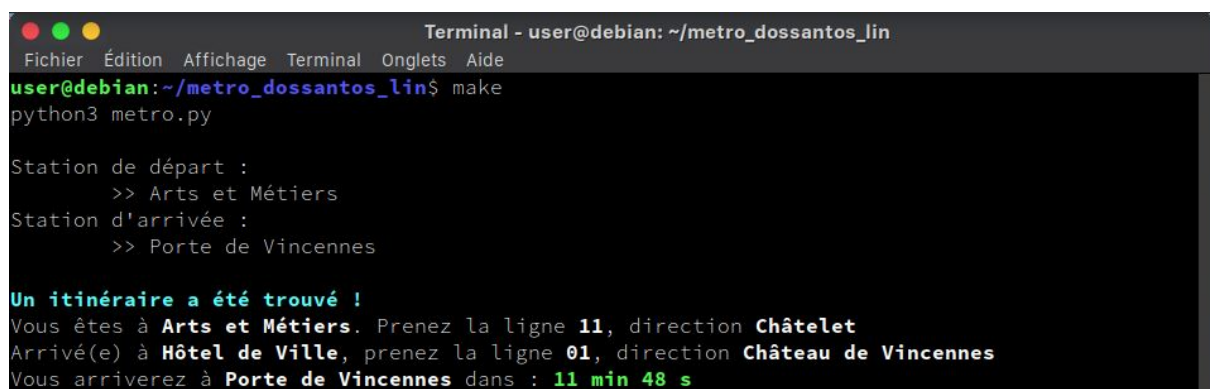
L'affichage du trajet s'effectue avec la fonction ShortestPath(). Celle-ci permet de stocker plusieurs trajets, si plusieurs lignes de métro de départ sont possibles pour la même station.

Cette fonction appelle donc dijkstra() autant que nécessaire. Pour chaque appel de dijkstra(), le chemin est retourné par getTravel() et la durée du trajet par getTime().

getTravel() retourne une liste de String qui est créée dans StockTravel(). Stocktravel() parcourt les stations du chemin précédemment calculé, et retourne une liste de phrases qui décrivent celui-ci.

Tout d'abord, une phrase décrit la station de départ, la ligne et la direction. Cette direction est déterminée avec Terminus(). Ensuite, à chaque fois qu'un changement de ligne est effectué, une phrase est ajoutée : celle-ci décrit la station à laquelle le changement s'effectue, la nouvelle ligne de métro à prendre et sa direction. A la fin, la dernière phrase contient la station d'arrivée et la durée du trajet.

Toute cette liste est donc renvoyée à chaque appel de dijkstra() dans ShortestPath(). ShortestPath() compare ensuite les différentes durées de trajet des différentes listes, et la liste ayant la durée la plus courte est affichée à l'écran.



```
Terminal - user@debian: ~/metro_dossantos_lin
Fichier Édition Affichage Terminal Onglets Aide
user@debian:~/metro_dossantos_lin$ make
python3 metro.py

Station de départ :
    >> Arts et Métiers
Station d'arrivée :
    >> Porte de Vincennes

Un itinéraire a été trouvé !
Vous êtes à Arts et Métiers. Prenez la ligne 11, direction Châtelet
Arrivé(e) à Hôtel de Ville, prenez la ligne 01, direction Château de Vincennes
Vous arriverez à Porte de Vincennes dans : 11 min 48 s
```

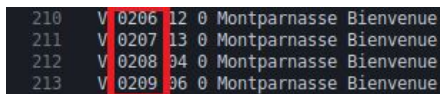
Sur l'image ci-dessus, Arts et Métiers est la station de départ et Porte de Vincennes celle d'arrivée. On voit qu'il y a un changement de ligne entre la 11 et la 01 à Hôtel de Ville.

4.5. Résolution des problèmes rencontrés

L'une des premières difficultés que nous avons rencontrées au cours de ce projet est l'importance de la fiabilité du fichier de configuration. En effet, il est capital que les lignes de métro et les terminus aient été correctement ajoutés lors de la modification de ce fichier afin de ne pas fausser les calculs, voire créer des erreurs. Pour chaque station, il nous a fallu regarder dans la liste des transitions quelles étaient ses voisines en faisant attention à ce qu'elles soient bien sur la même ligne.

Un autre des problèmes que nous avons eus a été le cas où l'utilisateur entre une station qui se trouve être au croisement de différentes lignes de métro. Ils nous a fallu optimiser l'algorithme de recherche pour proposer le trajet le plus rapide. Bien

qu'elle puisse sembler peu performante, notre méthode pour gérer ces cas a été de lancer l'algorithme de Dijkstra pour chacune de ces stations de départ. Prenons l'exemple concret de la station "Montparnasse Bienvenue" qui est le noeud de 4 lignes de métro (la M4, la M6, la M12 et la M13). Chacune de ces stations possède un numéro particulier comme vous pouvez le voir dans l'image ci-dessous :



```
210 V 0206 12 0 Montparnasse Bienvenue
211 V 0207 13 0 Montparnasse Bienvenue
212 V 0208 34 0 Montparnasse Bienvenue
213 V 0209 36 0 Montparnasse Bienvenue
```

Ainsi, on lance quatre fois la méthode de classe `dijkstra()`, une pour chaque station Montparnasse Bienvenue (comme cela a été expliqué dans la section 4.3 du rapport). Après calcul, on affiche simplement le trajet le plus court parmi les quatre ayant été calculés. Malgré l'aspect quelque peu "*brute force*" de gérer ce problème, nous n'avons pas constaté d'augmentation significative du temps de calcul. Celui-ci s'effectue en moins d'une seconde, quelle que soit la station en entrée.

De même, dans certains cas, la station d'arrivée de l'utilisateur est, elle aussi, le noeud de plusieurs lignes de métro. Dans ce cas, l'algorithme choisissait la première station correspondant à la demande de l'utilisateur dans la liste "stations". Cela occasionnait un dernier changement au sein de celle-ci, ce qui augmentait inutilement le temps de trajet. Il nous a donc fallu trouver une parade. Prenons l'exemple du trajet Montparnasse Bienvenue - Pasteur : Avant implémentation de la méthode de classe `getFinalDest()`, le trajet le plus court donné par `dijkstra()` était :

***Vous êtes à Montparnasse Bienvenue, prenez la ligne 12 direction Mairie d'Issy
Vous arriverez à Pasteur dans : 1 min 5 s***

L'algorithme ne proposait pas le trajet pourtant plus court de Montparnasse Bienvenue - Pasteur sur la ligne 6. Celui-ci effectuait un dernier changement à Pasteur (M6) pour se rendre à Pasteur (M12), ce dernier étant le premier à être lu dans la liste des stations. Comme expliqué dans la section 4.2, nous avons réglé ce problème en récupérant une liste contenant l'ensemble des stations d'arrivée portant le même nom. Ainsi, pour chaque `dijkstra()` effectué au départ, on regarde également laquelle des destinations offre le trajet le plus court et c'est celui-ci qui sera proposé à l'utilisateur. Pour l'exemple de Montparnasse Bienvenue - Pasteur, on aura ainsi le résultat suivant :

```
Station de départ :  
    >> Montparnasse Bienvenue  
Station d'arrivée :  
    >> Pasteur  
  
Un itinéraire a été trouvé !  
Vous êtes à Montparnasse Bienvenue. Prenez la ligne 06, direction Charles de Gaulle, Etoile  
Vous arriverez à Pasteur dans : 0 min 54 s
```

Enfin, la dernière difficulté que nous avons eue à été de gérer les lignes qui avaient des portions à sens unique telle que la ligne 10 ou la ligne 7B. Afin d'éviter que l'algorithme de Dijkstra ne prenne ces portions en sens inverse pour réduire le temps de trajet, nous avons ajouté dans la méthode de classe `getNeighbours()` des conditions pour supprimer de la liste des voisins les stations vers lesquelles la station courante ne pouvait pas aller. Par exemple, depuis la station Javel (n° 145), il est impossible de se rendre à la station Mirabeau (n° 201) directement car cette portion est à sens unique (depuis Mirabeau vers Javel). Ainsi, on supprime la transition "0145 0201" de la liste "neighbours".

5. Pour aller plus loin

5.1. Idées pour l’affichage graphique

Notre programme possède une interface écrite. Une amélioration de celui-ci serait de proposer une interface graphique, avec une carte du métro parisien et la possibilité de visualiser le trajet. Plusieurs options pourraient être étudiées : cliquer sur les stations pour effectuer le calcul du plus court chemin entre celles-ci, ainsi qu’afficher le chemin en surbrillance sur la carte avec, par exemple différentes couleurs représentant différentes lignes de métro ce qui permet de bien voir où se situent les changements de ligne à effectuer. Enfin, si l’application possède des données GPS, il serait très utile d’afficher un point sur la carte représentant la position de l'utilisateur, afin qu'il se repère par rapport aux stations et faciliter ainsi son choix de trajet.

5.2. Comment améliorer l’application ?

Malgré l’efficacité d’un algorithme de recherche du plus court chemin comme Dijkstra, il est toujours possible de l’optimiser, notamment pour un système de transport en commun comme le métro parisien. En ajoutant la position GPS de chaque station dans le fichier de configuration, on pourrait calculer la distance réelle qui sépare deux stations. Dès lors, un algorithme de recherche tel que A* (prononcé A Star en anglais) serait bien plus performant que Dijkstra car il implémente une heuristique qui nous indique dans quelle direction chercher en priorité. A* utilise lui aussi un calcul de distance comme le fait Dijkstra, mais il ajoute la distance physique qu’il reste à parcourir entre le lieu où l’on se trouve et celui où l'on veut aller.

La condition pour mettre à jour la distance et le père d’un sommet deviendrait alors:

```
1  time = transitionTime(currentStation, nextStation)
2  realDist = realDistance(nextStation, dest)
3
4  if (distance[nextStation] + realDist > distance[currentStation] + time + realDist):
5      distance[nextStation] = distance[currentStation] + time + realDist
6      prev[nextStation] = currentStation
```

Une autre amélioration que l’on pourrait proposer pour cette application serait d’ajouter un temps d’arrêt entre chaque station. En effet, en l’état actuel, le programme ne prend pas en compte qu’à chaque station le métro doit s’arrêter pour

laisser monter et descendre des passagers. Ainsi, le temps de trajet est significativement réduit par rapport à la réalité (même lorsque les conditions de circulation réelles sont optimales).

Par exemple, un trajet Montparnasse Bienvenue - Porte de Clignancourt sur la ligne 4 prend 14 min et 16 s d'après l'application alors qu'en réalité, il faut environ 22 minutes.

Une autre amélioration à laquelle nous avons pensé serait d'ajouter un algorithme de Levenshtein qui permettrait de suggérer un nom de station en cas d'erreur de frappe de l'utilisateur ou pour réaliser une auto-complétion du nom des stations si assez de caractères correspondent avec une station de la liste.

Enfin, une dernière amélioration possible serait de proposer à l'utilisateur le choix d'une langue (français, anglais, espagnol, allemand, etc...) pour l'affichage du trajet et des éventuelles erreurs.