**2019**

Auteurs :

Geoffrey Pouliquen

Florian Gaveau

Apolline El Baz

Projet Python - MyWay





**Introduction**

Le projet consistait à trouver l’itinéraire de métro le plus court entre deux stations pour un jour et une heure donnés. Pour ce faire nous avons implémenté un algorithme de Dijkstra, qui plus généralement, permet de trouver le plus court chemin entre deux nœuds d’un graphe. Ici, nous avons donc considéré que le plan de métro était un graphe, les stations constituant, en quelque sorte, les sommets, et les lignes, les arêtes.

Nous avons dû travailler sur les données fournies par la RATP au format GTFS. A chaque ligne de métro correspond un dossier RATP\_GTFS\_METRO\_*numéro\_de\_ligne*;dans lequel se trouvent huit fichiers. Six de ses fichiers nous ont fourni des informations indispensables pour traiter le problème - *calendar\_dates*, *stop\_times*, *transfert*, *trips*, *stops* - et améliorer la mise en forme de nos résultats - *routes*.

**1. Lecture et traitement des données**

Pour pouvoir utiliser l’algorithme de Dijsktra, il est nécessaire de déterminer les voisins de tous les quais du métro et de leur associer les informations qui permettront de calculer leur poids dans le graphe.

Chaque trajet d’un train, référencé dans les fichiers *stops\_times*, possède un *trip\_id* qui est lui-même associé à un *service\_id* dans les fichiers *trips.txt*. Les *service\_id* sont des services de train seulement actifs les jours définit dans les fichiers *calendar\_dates.txt*. En liant ces données entre elles dans des dictionnaires (*trips\_dict* et *services\_dict*), on peut trouver la date de chaque trajet. Puis grâce aux heures de départ (*departure\_time*) écrites dans *stops\_times.txt*, on peut calculer les poids de chaque voisin de tous les quais en fonction de d’une heure donnée.

De même, les fichiers *stops.txt* permettent d’associer à un nom de station la liste des quais (*stop\_id*) qu’elle contient. Ces informations sont stockées dans *platform\_dict* et permettent, à partir des noms des stations d’arrivée et de départ, de déterminer les quais de départ et d’arrivée, c’est-à-dire les nœuds initiaux et finaux de notre graphe. Ce sont donc les quais des stations et non les stations elle-mêmes qui vont constituer les nœuds de notre graphe. La lecture de l’ensemble des fichiers et la constitution des dictionnaires prend un peu plus d’une minute.

Pour rassembler les informations relatives à chaque voisin, on va se servir d’une classe *neighbour.*

Classe *neighbour*

Un objet de la classe *neighbour* est censé représenter un voisin du nœud courant dans le graphe utilisé pour l’algorithme de Dijsktra, c’est-à-dire un quai d’une station de métro qui suit immédiatement le quai où s’est arrêté l’algorithme (quai courant). Il contient donc le numéro (*stop\_id*) du quai voisin ainsi que la durée du trajet depuis le quai courant. Cette durée est un temps de transfert (*transfer*) si les deux quais sont dans la même station ou une liste de services (*services*) de train si ce n’est pas le cas. Dans le premier cas, la liste *services* sera laissée vide et dans le deuxième cas *transfer* sera considéré égal à 0.

Classe *service*

Un *service* possède deux attributs : un numéro de service (*service\_id*) et une liste d’horaires (*departures*). Ces horaires représentent l’heure de départ du train au quai courant et le temps nécessaire pour arriver au quai voisin.

Pour résumer, à un numéro de quai est associé un quai voisin et les services disponibles avec toutes les heures de passage des trains pour chacun de ses services. Chaque quai peut évidemment avoir plusieurs voisins ou n’en avoir aucun si c’est un terminus.

|  |
| --- |
| Stop\_id (quai) |
| Neighbour |

|  |  |  |
| --- | --- | --- |
| Platform\_id | [Services] | Transfer |

|  |  |
| --- | --- |
| Service\_id | [Departures] |

|  |  |
| --- | --- |
| Heure de départ | Durée de trajet |

*Path\_dict*

Le dictionnaire path\_dict contient les listes de voisins (sous la forme d’objet *neighbour*) de chaque quai. Le remplissage du dictionnairese fait après la création du *trips\_dict* et par la lecture du fichier *stop\_times* avecles fonctions *find\_platform*, *find\_service* et *add\_path* selon le schéma suivant :

trip\_id,arrival\_time,departure\_time,stop\_id,stop\_sequence

10027735930912409, 14:18:00, 14:18:00, 1875, 1

10027735930912409, 14:20:00, 14:20:00, 1753, 2

10027735930912409, 14:21:00, 14:21:00, 1682, 3

10027735930912409, 14:22:00, 14:22:00, 1798, 4

10027735930912409, 14:23:00, 14:23:00, 1905, 5

10027735930912409, 14:24:00, 14:24:00, 1758, 6

* Dans la liste des voisins de 1875, on ajoute 1753 s’il n’y est pas déjà.
* Grâce à *trips\_dict*, on trouve que le *trip\_id* fait partie du service de *service\_id* 2773593
* Dans la liste des services de 1753, on ajoute un service 2773593 s’il n’existe pas encore.
* Dans la liste *departures* du service 2773593, on ajoute l’horaire de départ en 1875 et la durée de trajet vers 1753, ici 2 min.

On parcourt tout le fichier

Certains voisins ont ensuite été ajoutés à la main, car ils constituent des exceptions. Comme il n’y a pas de temps de transfert définie entre deux quais appartenant à la même ligne, on ne peut pas changer de direction lorsque l’on arrive à une fourche pour repartir de l’autre côté de la fourche. Il est donc nécessaire, pour les quais qui précèdent une fourche, de leur ajouter le quai opposé (même ligne mais dans l’autre sens) comme voisin. Par exemple sur la *ligne 7*, à la station *« Maison Blanche »*, on pourrait vouloir changer de direction, pour aller de *« Mairie d’Ivry »* à *« Villejuif »*. Par défaut, nous avons fixé le temps de transfert à trente seconde pour ceux-là.

**2. Déterminer l’itinéraire : l’algorithme de Dijsktra**

Calcul des poids

Le calcul des poids des nœuds voisins est réalisé par la fonction *Calcul\_weight*. Elle prend en argument un voisin, une heure de départ, et la liste des services disponibles le jour du départ ainsi que le jour précédant ou suivant. L’objectif de cette fonction est de calculer le poids d’un voisin dans le graphe.

Si nœud courant est dans le même station que le nœud voisin (*transfer* > 0), alors la fonction renverra directement la valeur de *transfer* du voisin.

Si ce n’est pas le cas, l’objectif de la fonction sera, dans un premier temps, de trouver le plus petit poids, c’est-à-dire le temps d’attente pour le prochain train ajouté au temps de trajet vers le voisin pour chaque service inclus dans *day\_services* et *other\_day\_services*. Puis dans un second temps de renvoyer le plus petit poids tous services confondus ce qui permettra de déterminer le prochain train à passer en gare. L’heure de départ de ce train est également enregistrée pour des raisons d’affichage.

L’algorithme de Dijkstra

*Calcul\_itinerary* prend en argument le nom de la station de départ, le nom de la station d’arrivée, la date et l’heure de départ du trajet défini par l’utilisateur. Ces données initiales sont ensuite converties pour être utilisables par l’algorithme avec les fonctions *define\_time* (heure convertie en secondes par rapport au début de la journée), *define\_*services (revoie les services du jour de départ) et *define\_platform* (renvoie un quai de la station de départ comme quai initial).

Si l’heure définie est après 22h ou avant 2h, alors les services du jour suivant (resp. précédent) est enregistré dans *other\_day\_services*, liste contenant également un 1 (resp. -1) qui sera utilisé pour la calcul des poids. Le quai opposé (même ligne mais en sens inverse) est ajouté en tant que voisin du quai initial avec un temps de transfert de 30 secondes.

On initialise ensuite une liste *Dijsktra\_tab* vide. A chaque étape de l’algorithme, pour chaque quai voisin du quai courant, on ajoutera à *Dijsktra\_tab* une liste contenant : le temps écoulé depuis le début du voyage + le poids du voisin (calculé avec *Calcul\_weight*), le quai voisin, le quai courant et l’heure de départ pour le trajet entre les deux (pour l’affichage des résultats). L’élément de *Dijsktra\_tab* possédant le premier élément le plus petit sera ajouté à une autre liste *Path*. Le voisin choisi sera alors ajouté au set *Already\_visited*. Par exemple, en partant de **A**, à la première étape, supposons que nous soyons dans la situation suivante :

1min

1min40

2min

1min30

*Dijsktra\_tab* pourrait se représenter ainsi :

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | 2min | B | A | 13:02 | | |  |  |  |  | | --- | --- | --- | --- | | 1min30 | C | A | 13:01:30 | | |  |  |  |  | | --- | --- | --- | --- | | 1min40 | D | A | 13:01:40 | | |  |  |  |  | | --- | --- | --- | --- | | 1min | E | A | 13:01 | |

|  |  |  |  |
| --- | --- | --- | --- |
| 1min | E | A | 13:01 |

*Path :*

|  |  |
| --- | --- |
| A | E |

*Already\_visited*:

Le quai choisi (**E** dans notre exemple) devient le quai courant et on répète l’opération décrite ci-dessus jusqu’à atteindre un quai qui soit dans le liste des quais de la station d’arrivée. Pour les quais inclus dans *Already\_visited*, les poids ne sont plus calculés et ils ne peuvent plus être ajoutés dans *Disjktra\_tab*.

Une fois arrivé à la bonne station, il faut rebrousser le *Path* à partir du numéro du quai d’arrivée. On cherche alors dans *Path* un quai qui soit un voisin du quai d’arrivée et que le temps de trajet entre les deux soit le plus petit. Puis on considère que ce voisin est le point d’arrivée. En répétant cette opération, on remplit la liste *Itinerary* qui contiendra le bon chemin. *Calcul\_itinerary* affiche et renvoie *Itinerary*.

3. Affichage du résultat

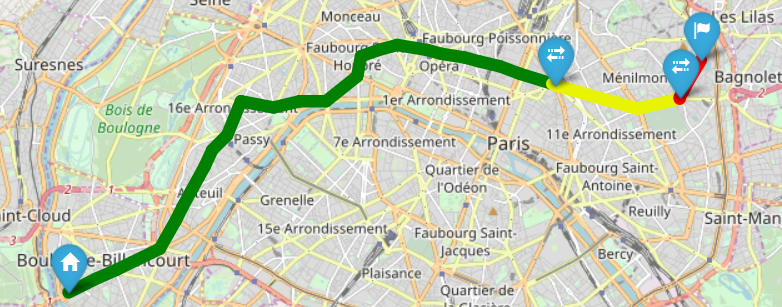
Pour des questions de présentation, il était nécessaire de récupérer et de reconvertir les horaires traités en secondes au format « hh:mm:ss », avec *convert\_hour*. Il fallait également récupérer le numéro de ligne associé à un quai, et la direction dans laquelle il fallait que l’usager prenne la ligne.

Nous avons donc construit trois dictionnaires, *stop\_trip\_dict*, *routes\_dict* et enfin *line\_direction\_dict* pour remonter à l’information qui nous intéressait, en lisant respectivement les fichiers *stop\_times*, *trips* et *routes*.

De plus, la fonction *Display\_itinerary* permet de reconstituer le trajet sur une carte *folium*. Nous avons enrichi la carte en récupérant les stations d’arrivée et de départ et celles où l’on effectuait un changement.

Les noms des stations et les coordonnées des quais sont donnés par les dictionnaires *display\_dict* et *coordinates\_dict* et sont construits à partir de *stops*.

Le lien *html* de la carte est sauvegardé dans le dossier courant sous le nom de « *Itinerary.html »*. Pour le cas-test "Starting in a terminus", on obtient la sortie suivante :



**Conclusion**

Nous avons présenté en fin du fichier « *FindWay.py »* douze cas-tests qui nous ont permis de vérifier le bon fonctionnement du code. Par exemple, pour le cas-test « Multiple changes » nous avions remarqué que l’algorithme de Dijkstra nous faisait faire un détour en forme de boucle, puisque, dans un premier temps, nous n’avions considéré que le premier élément de *platform\_dict ["Ecole Militaire"]*. Si le quai choisit comme quai initial n’était pas dans le ‘‘bon sens’’ par rapport à la destination finale alors l’algorithme renvoyait un itinéraire alambiqué avec de multiples changements ou échouait à trouver un itinéraire. Il a donc fallu ajouter le quai opposé au quai initial avec un temps de transfert de 30 secondes (choix arbitraire).

On peut aussi noter que le cas des fourches est problématique puisqu’il n’est possible de traiter ces configurations qu’en ajoutant manuellement des voisins. Il serait sans doute nécessaire de référencer ce type de quai et de ligne différemment dans les fichiers.

Par ailleurs, nous avons remarqué que certains temps de transfert des fichiers de la RATP avaient, sans doute, été sous-évalués. L’itinéraire du dernier test « Walking time underestimated » comprend en effet la station « *Châtelet* » qui comporte un grand nombre de correspondances.