



MU4RBI02 C++ avancé

Rapport de projet

Jiheng WEI
Master ISI

Table des matières

1	Introduction	1
2	Implémentation	1
2.1	Stations	2
2.2	Connexions	2
2.3	Constructeur	3
2.4	Recherche de chemin	3
2.5	Affichage	4
3	Résultats	5
4	Amélioration	6
	Références	8

1 Introduction

Le projet vise à construire un programme permettant de trouver le trajet le plus court entre deux gares d'un réseau de transport. Il s'agit d'un problème typique du plus court chemin. Pour modéliser ce problème, le réseau est représenté par un graphe orienté. Dans le graphe, les stations sont des nœuds reliés par des arêtes. Les arêtes, qui représentent les liaisons de transport existantes, seront orientées car certaines gares ne sont desservies que par un seul sens de ligne. Le poids de chaque arête représente le temps nécessaire pour passer d'un nœud à un autre à travers cette arête. Comme on ne traversera pas un tunnel temporel, le poids sera partout non négatif. Ensuite, une fois le problème bien défini, on implémentera l'algorithme de Dijkstra pour trouver l'itinéraire.

Dans ce projet, la base de données du réseau RATP (métros uniquement) est fournie. Les informations sur les stations et les informations sur les connexions sont stockées dans les fichiers `.csv` respectifs. Les interfaces pour les parseurs de station et de connexion, ainsi que pour l'algorithme de recherche, sont imposées dans les fichiers d'en-tête. Par conséquent, la tâche est de mettre en œuvre la solution en suivant strictement ces impositions.

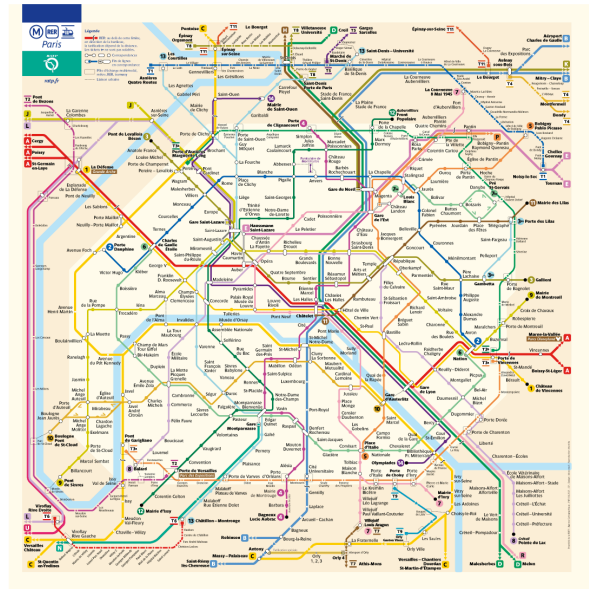
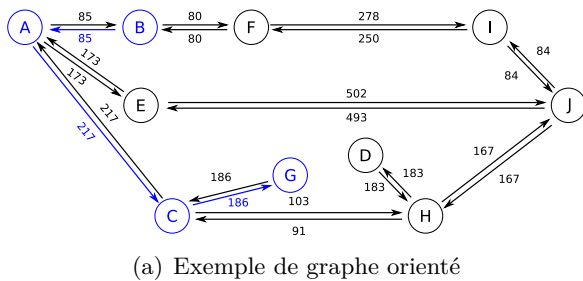


FIGURE 1 – Présentation du problème

Dans les sections suivantes, les choix de mise en œuvre pour chaque partie seront d'abord expliqués. Ensuite, les résultats seront analysés. Enfin, les améliorations possibles seront discutées.

2 Implémentation

La tâche principale est de surcharger les méthodes virtuelles des classes imposées qui sont définies dans l'espace de noms `travel`. Ces classes sont héritées par la classe `travel::Navigator` que l'on a créée.

2.1 Stations

La gestion des stations se fait dans la classe `travel::Generic_station_parser`. Les stations sont organisées dans un attribut `stations_hashmap` de type `std::unordered_map` dans lequel les clés sont de type `uint64_t` et des valeurs de type `travel::Station`. Un `unordered_map` est un conteneur qui contient des paires clé-valeur. Il est non ordonné car ses éléments ne sont pas triés dans un ordre particulier. Avec cette instruction, l'utilisation de `stations_hashmap` est claire : à l'aide de l'index d'une station, on accède à ses informations organisées en instances de type `Station`.

La fonction `Navigator::read_stations` va lire les informations des stations dans un fichier `.csv`. Muni de son chemin `_filename`, un fichier ouvert sera d'abord stocké dans la variable `fin` de type `std::ifstream`. La fonction vérifie d'abord si le fichier est correctement lu. Sinon, il enverra un message d'erreur et quittera le programme. Si l'ouverture est réussie, la fonction `std::getline` lira une ligne du fichier à chaque fois et la stockera dans `line` de type `std::string`. La première ligne d'un fichier `.csv` correspond à l'en-tête du tableau. Pour l'ignorer, il suffit d'exécuter `getline` une fois sans aucune action supplémentaire.

Ensuite, les cellules d'une ligne seront séparées. Pour ce faire, une variable `sin` de type `std::istringstream` est créée avec `line`. `getline` lit les cellules depuis `sin` en utilisant le délimiteur `,`. Une fonction d'assistance `Navigator::trim` est définie pour supprimer les espaces avant ou après une cellule [1]. Les cellules sont d'abord stockées dans un `std::vector`. Ceux de la deuxième colonne de `.csv` sont les identifiants des stations. Puisqu'ils sont de type `string`, ils doivent être convertis en nombres entiers. La fonction de conversion la plus adéquate est `std::stoull`. Il renvoie un nombre de type `unsigned long long` qui garantit une longueur de 64 bits [2]. Pour instancier les stations, il faut bien identifier les noms en en-tête du tableau et leurs homologues dans la définition de la structure `Station`.

Champs de structure <code>travel::Station</code>	Éléments de l'en-tête du tableau
<code>name</code>	<code>string_name_station</code>
<code>line_id</code>	<code>string_short_line</code>
<code>address</code>	<code>string_adress_station</code>
<code>line_name</code>	<code>string_desc_line</code>

Pour ajouter un élément au conteneur, il suffit de spécifier une clé et sa valeur.

```
this->stations_hashmap[key] = value;
```

2.2 Connexions

La classe `travel::Generic_connection_parser` sert aux connexions entre les stations. Sa méthode `read_connections` lit un fichier `.csv` dont les colonnes sont respectivement l'ID de la station de départ, l'ID de la station d'arrivée et le temps de transfert entre les deux stations. Cela fonctionne presque de la même manière que `read_stations`, sauf que toutes les cellules doivent être converties en nombres entiers. Les connexions sont organisées en `connections_hashmap` de type `unordered_map`. Ses clés sont des entiers dont la valeur correspondante est à nouveau un `unordered_map` d'entier-entier.

```
this->connections_hashmap[start][end] = time;
```

2.3 Constructeur

À ce stade, on peut déjà définir le constructeur de la classe `Navigator`. Le principe est que l'utilisateur va interagir avec le programme via le terminal sans modifier le code. En conséquence, un constructeur avec des chemins pour les fichiers de station et de connexion comme arguments suffit. Il appellera les méthodes `read_stations` et `read_connections`.

On a décidé de ne pas autoriser de constructeur par défaut ni de setters pour les hashmaps. La raison en est que cela provoquera une perturbation importante si l'utilisateur oublie d'utiliser les setters simultanément. De plus, comme les instances de `Navigator` seront énormes, la copie est interdite en désactivant l'opérateur `=`.

2.4 Recherche de chemin

La méthode `Navigator::compute_travel` sert à calculer le chemin le plus court de `start` à `end`. Il vérifie d'abord si ces deux stations existent dans les hashmaps. Sinon, il renvoie un `vector` vide ; si c'est le cas, il trouvera le meilleur chemin entre eux en utilisant l'algorithme de Dijkstra suivant qui est adapté à partir de [3].

Algorithm 1 : Dijkstra's algorithm

Data : departure station *start*, arrival station *end*, connections_hashmap *C*

Result : *cost* : time to the stations, *prev* : the last station before reaching the current one

```
1 begin
2   cost  $\leftarrow \emptyset$ , prev  $\leftarrow \emptyset$ , Q  $\leftarrow \emptyset$ 
3   foreach v : vertice of the graph do
4     cost[v]  $\leftarrow \infty$ 
5     Add v to Q
6   cost[start]  $\leftarrow 0$ 
7   foreach v : neighbour of start do
8     cost[v]  $\leftarrow C[start, v]$ 
9
10  init  $\leftarrow \text{true}$ 
11  while Q  $\neq \emptyset$  do
12    if init then
13      u  $\leftarrow start$ 
14      init  $\leftarrow \text{false}$ 
15    else
16      u  $\leftarrow \arg \min_{i \in Q} cost[i]$ 
17
18    Remove u from Q
19    if u = end then // Best path from start to end is found
20      break
21
22    foreach v  $\in Q$  : neighbour of u do
23      alt  $\leftarrow cost[u] + C[u, v]$ 
24      if alt  $\leq cost[v]$  then // Update the path: from start to v via u
25        cost[v]  $\leftarrow alt$ 
26        prev[v]  $\leftarrow u$ 
```

La valeur de ∞ est ici définie comme la constante `INF` qui est la valeur maximale autorisée pour le type de données `uint64_t`.

Le sommet `start` est d'abord exploré dans la boucle `while`. En conséquence, `alt` aura la même valeur que ce que l'on a initialisé dans `cost` à la ligne 8. Si la comparaison à la ligne 24 est `<`, la connexion de `start` à ses voisins ne sera pas enregistrée dans `prev`.

Le résultat du chemin le plus court doit être dans le vecteur `best_path` de paires de `uint64_t`. Les deux éléments de la paire sont respectivement l'identifiant de la station et le temps nécessaire de `start` à cette station. Afin de remplir `best_path`, on part de `end` et en utilisant l'algorithme d'itération inverse suivant.

Algorithm 2 : Reverse iteration

```

1 begin
2   best_path  $\leftarrow \emptyset$ , child  $\leftarrow$  end
3
4   Add {end, cost[end]} to best_path
5   while child  $\neq$  start do
6     parent  $\leftarrow$  prev[child]
7     Add {parent, cost[parent]} to the beginning of best_path
8     child  $\leftarrow$  parent
9
10  return best_path

```

2.5 Affichage

La méthode `Navigator::compute_and_display_travel` appellera d'abord `compute_travel` pour obtenir le chemin. Selon le contenu de `best_path`, quatre cas d'affichage sont possibles :

- Le vecteur est vide, un message indiquant que la ou les stations n'existent pas sera affiché.
- `_start` et `_end` partagent le même `name` et `line_id`, le programme indique que l'on est déjà arrivé.
- `_start` et `_end` partagent le même `name` mais avec des `line_id` différent, l'utilisateur est invité à marcher au quai d'une autre ligne.
- Tous les autres scénarios où un planificateur d'itinéraire est utile.

Dans les trois premiers cas, le programme affichera simplement un message. On s'intéresse au dernier cas. Le style d'affichage pour ce cas est :

```

Departure station
  -> Metro line (time)
Transfer station
  -> Metro line (time)
...
  -> Metro line (time)
Arrival station

Total time

```

Le style est beaucoup plus léger que celui de `grade`. Les numéros de lignes ne sont pas indiqués dans les adresses et la marche d'un quai à l'autre n'est pas explicitée. Le temps affiché après une

ligne est le temps écoulé depuis que la personne se met en ligne jusqu'à ce qu'elle arrive au quai de la ligne suivante. La simplification a du sens en réalité : si l'on est dans une gare à une ligne, on n'a pas le choix ; si l'on est dans une station de transfert, le planificateur lui indiquera quelle ligne il doit prendre. Ainsi, lorsqu'on est dans une gare, l'information que le quai appartient à quelle ligne est moins intéressante.

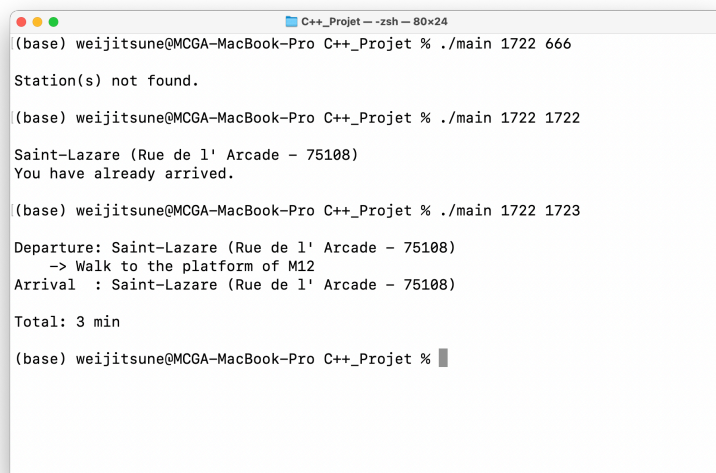
Il y a un problème dans le chemin comme suit :

```
station a, time 1 // End of a line
station b, time 1
-----
station c, time 2
station d, time 2 // Beginning of a line
```

Lorsque deux paires partagent le même temps, cela signifie que les deux stations sont en fait identiques mais de sens de ligne opposé. Lorsque cela se produit, le premier pour la fin et le dernier pour le début d'une ligne est dans le bon sens. Les mauvais en rouge seront ignorés dans le programme.

3 Résultats

Les résultats obtenus correspondent à l'attente. Celui du quatrième cas concorde avec l'un des itinéraires proposés par l'application mobile officielle de la RATP.



```
(base) weijitsune@MCGA-MacBook-Pro C++_Projet % ./main 1722 666
Station(s) not found.

(base) weijitsune@MCGA-MacBook-Pro C++_Projet % ./main 1722 1722
Saint-Lazare (Rue de l' Arcade - 75108)
You have already arrived.

(base) weijitsune@MCGA-MacBook-Pro C++_Projet % ./main 1722 1723
Departure: Saint-Lazare (Rue de l' Arcade - 75108)
-> Walk to the platform of M12
Arrival : Saint-Lazare (Rue de l' Arcade - 75108)
Total: 3 min

(base) weijitsune@MCGA-MacBook-Pro C++_Projet %
```

FIGURE 2 – Résultats des trois premiers cas

```

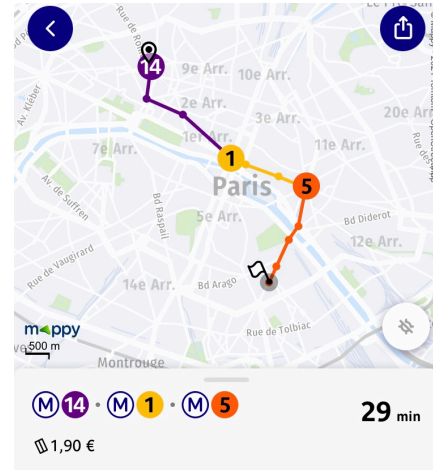
(base) weijitsune@MCGA-MacBook-Pro C++_Projet % ./main 1722 2017

Departure: Saint-Lazare (Rue de l' Arcade - 75108)
-> M14 (SAINT-LAZARE <-> OLYMPIADES) - Aller (12 min)
Châtelet
-> M1 (CHATEAU DE VINCENNES <-> LA DEFENSE) - Retour (7 min)
Bastille
-> M5 (BOBIGNY - PABLO PICASSO <-> PLACE D'ITALIE) - Aller (8 min)
Arrival : Campo-Formio (106-112 boulevard de l' Hôpital - 75113)

Total: 28 min

(base) weijitsune@MCGA-MacBook-Pro C++_Projet %

```

(a) `travel::Navigator`

(b) Application RATP

FIGURE 3 – Résultat du dernier cas

En outre, la mise en œuvre a réussi le test complet de `travel::Grade`.

```

(base) weijitsune@MCGA-MacBook-Pro C++_Projet % ./main
===== Grade 1 <=====
Stations: seems ok
===== Grade 2 <=====
Connections: seems ok
===== Grade 3 <=====
First tests
Tests: seems ok.
Would you try complete test (wait ~ 39 mins and 53 secs) ? (N/y):

```

```

97% 37 mins and 54 secs elapsed, wait 1 mins and 1 secs
97% 37 mins and 57 secs elapsed, wait 58 secs
97% 38 mins and 0 secs elapsed, wait 55 secs
97% 38 mins and 3 secs elapsed, wait 52 secs
97% 38 mins and 5 secs elapsed, wait 49 secs
98% 38 mins and 8 secs elapsed, wait 46 secs
98% 38 mins and 12 secs elapsed, wait 43 secs
98% 38 mins and 15 secs elapsed, wait 39 secs
98% 38 mins and 18 secs elapsed, wait 36 secs
98% 38 mins and 20 secs elapsed, wait 33 secs
98% 38 mins and 23 secs elapsed, wait 30 secs
98% 38 mins and 26 secs elapsed, wait 27 secs
98% 38 mins and 29 secs elapsed, wait 24 secs
99% 38 mins and 32 secs elapsed, wait 21 secs
99% 38 mins and 35 secs elapsed, wait 18 secs
99% 38 mins and 38 secs elapsed, wait 15 secs
99% 38 mins and 41 secs elapsed, wait 12 secs
99% 38 mins and 44 secs elapsed, wait 9 secs
99% 38 mins and 47 secs elapsed, wait 6 secs
99% 38 mins and 50 secs elapsed, wait 3 secs
100% 38 mins and 53 secs elapsed, wait 0 secs

Tests: seems ok
(base) weijitsune@MCGA-MacBook-Pro C++_Projet %

```

FIGURE 4 – Résultats des tests de `travel::Grade`

4 Amélioration

D'après les exemples précédents, il est facile de se sentir contre-intuitif que l'on ait besoin d'utiliser les identifiants des stations pour la navigation. Par conséquent, on souhaite avoir une méthode qui permette de planifier des itinéraires en utilisant les noms de stations. En conséquence, les méthodes `compute_travel` et `compute_and_display_travel` seront surchargées en remplaçant le type de ses arguments par `std::string`.

La tâche principale ici est de trouver l'ID correspondant en fonction du nom qu'un utilisateur a saisi. Par conséquent, la méthode

```
std::pair<uint64_t, uint64_t> compute_id(const std::string&, const std::string&);
```

est faite pour cela. Il sera trop simple de trouver uniquement la station dont le nom correspond exactement à celui donné par l'utilisateur. Au lieu de cela, pour être à l'épreuve des erreurs, on effectuera une mesure de similitude.

La mesure de similarité se fait en calculant la distance de Hamming dans la fonction `hamming`, qui est le nombre de caractères différents, entre deux chaînes de longueur égale. Ainsi, la contrainte pour trouver la bonne station est que le nom saisi doit avoir la même longueur que celui de la station souhaitée. Le type `std::string` ici est gênante car les caractères accentués occupent plus d'une position dedans. Pour résoudre ce problème, les noms de stations seront d'abord convertis en type `std::wstring` composé de caractères larges [4]. Ensuite, les noms sont convertis en minuscules pour le calcul. Ainsi, le résultat n'est pas sensible à la casse.

La méthode `Navigator::compute_id` retourne la paire `{start_id, end_id}`, dans lequel les identifiants appartiennent aux stations dont les noms ont les plus petites distances de Hamming respectivement avec `_start` et `_end`. La méthode tolère jusqu'à 3 caractères erronés, car le nom de station le plus court comporte quatre caractères (Cit  /I  na). Si l'erreur d  passe cette limite, la station saisie est consid  r  e comme introuvable.

L'indicateur "n'existe pas" DNE est ici d  fini comme un alias de INF. Il est utilis   dans `hamming` lorsque les deux cha  nes ont des longueurs diff  rentes, de sorte que la distance est ind  finie par d  finition. Comme on trouve des r  sultats avec le moins de distance, un ∞    la fin de la recherche indique qu'aucune cha  ne ne correspond    celle en question. DNE est   galement utilis   pour indiquer qu'une station recherch  e n'est pas trouv  e, car aucun pays ne construit de r  seau de transport avec $2^{64} - 1$ stations.

Les r  sultats sont satisfaisants avec des cas de test dans lesquels des erreurs de saisie sont d  lib  r  ment ajout  es.

```

(base) weijitsune@MCGA-MacBook-Pro C++_Projet % ./main Heiligenstadt Karlsplatz
Station(s) not found.

(base) weijitsune@MCGA-MacBook-Pro C++_Projet % ./main Jussieu jUsSIeU
Jussieu (39 rue Lin   - 75105)
You have already arrived.

(base) weijitsune@MCGA-MacBook-Pro C++_Projet % ./main chAtelet citY
Departure: Ch  telet (Th   tre du Ch  telet - 75101)
-> M4 (PORTE DE CLIGNANCOURT <-> MAIRIE DE MONTROUGE) - Aller (5 min)
Arrival : Cit   (2 Place Louis Lepine - 75104)

Total: 5 min

(base) weijitsune@MCGA-MacBook-Pro C++_Projet % ./main saint-BIZARE Pastille
Departure: Saint-Lazare (Rue de l' Arcade - 75108)
-> M14 (SAINT-LAZARE <-> OLYMPIADES) - Aller (12 min)
Ch  telet
-> M1 (CHATEAU DE VINCENNES <-> LA DEFENSE) - Retour (9 min)
Arrival : Bastille (Rue de la Roquette - 75104)

```

FIGURE 5 – R  sultats utilisant les noms de stations

Malheureusement, l'algorithme n'a pas r  ussi le test de `Grade`. D'apr  s le message d'erreur et les heures dans les exemples de `Grade`, on sait que les identifiants de d  but et de fin doivent correspondre au niveau de num  ro de ligne. Cependant, ce n'est pas ce que l'on fait dans une application de transport en r  alit  . Si l'on sait au pr  alable quelle ligne prendre, il est inutile de d  verrouiller son portable et de demander de l'aide    un programme.

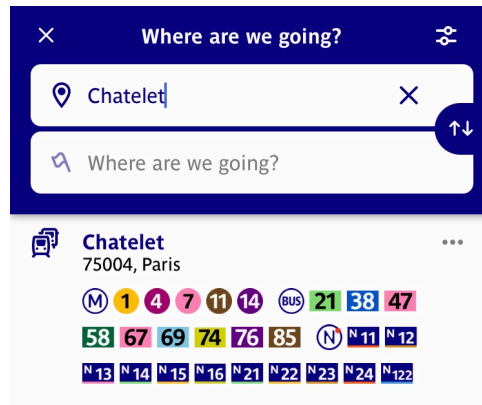


FIGURE 6 – Un planificateur d’itinéraire réel ne demande pas de spécifier une ligne au préalable

Références

- [1] T. Zhang, “Read data in a CSV file in C++.” [Online]. Available : <https://blog.csdn.net/u013232740/article/details/50828062>
- [2] K. Thompson, “c++ - unsigned long long conflict with uint64_t?” [Online]. Available : <https://stackoverflow.com/a/32198541>
- [3] “Dijkstra’s algorithm,” May 2021, page Version ID : 1025240537. [Online]. Available : https://en.wikipedia.org/w/index.php?title=Dijkstra%27s_algorithm&oldid=1025240537
- [4] AJKepler, “c++ - How do I convert a string to a wstring using the value of the string?” [Online]. Available : <https://stackoverflow.com/a/49741944>