UE MU4RBI02 - Programmation avancée C++
Rapport de projet : Application de calcul d'itinéraire.

HOCINE Karim
Master ISI

Introduction

Dans le cadre de ce projet nous avons développer un programme de calcul d'itinéraire basé sur l'algorithme Dijkstra. Cet algorithme permet de résoudre des problèmes de plus court chemin entre deux points A et B en se basant sur un réseau routier, ou comme dans notre cas en se basant sur le réseau féroviaire de la RATP.
A partir de deux fichiers CSV, l'un contenant les stations du réseau féroviaire de la RATP et l'autre les différentes connexions de chaque station avec le temps correpondant à chaque connexion, le programme devra déterminer le meilleur trajet possible entre deux stations, une station de départ (**Start**) et une station d'arrivée (**End**) qui pourra s'effectuer en un temps minimum.
Le programme a été implémenté sous *Visual Studio 2019.*

1 Récupération des stations :

A partir d'un fichier CSV contenant toutes les stations de Métro parisienne, nous allones récupérer chaque station et la stocker dans une structure du mème nom `travel :: Station`. Chaque ligne du fichier correspond aux informations relatives à une station particulière. Ces informations seront récupérés grâce à la fonction `std::getline()`.
Après avoir récupérer les informations relatives à une station, on stockera cette dernière dans le `stations_hashmap`. Pour cela on utilisera comme clé le champ `station_id` présent dans les fichier CSV qu'il faudra convertir en `uint64_t` au préalable grâce à la fonction `std::stoi()`.

Problèmes rencontrés : Lors de la conversion de la clé `station_id` en entier en utilisant la fonction `std :: stoi()` deux exceptions peuvent être levées :

- `std::invalid_argument:` Cette exception est levée lorsque la variable `string` qu'on souhaite convertir contient en premier une lettre ou ne contient pas de d'entiers. Pour y remédier, le seul moyen et de donnée en argument à `std::stoi()` les bons arguments.
- `std::out_of_range:` Cette exception est levée lorsque l'argument de la fonction `std::stoi()` est à une valeur trop important pour être stocker dans entier de type `"int"`. Pour y remédier il suffit de remplacer la fonction `std::stoi()` par `std::stol()` où `std::stoll()` (conversion en `long` ou `long long`) dans un bloc catch.

2 Récupération des connexions :

Le principe est le mème que celui de la récupération des stations : A partir d'un fichier CSV contenant les connexions de chaque stations du réseau RATP, nous allonrs récupérer trois champs par ligne : Start, End et time_transferet stocker le tout dans le `connection_hashmap`. Comme précedemment il faudra convertir les trois champs en entier gràce à la fonction `std::stoi()`.

Problèmes rencontrés : Les problèmes sont les mèmes que l"étape précedente et peuvent être traitées de la mème manière.

3 Algorithme Dijkstra :

L'algorithme Dijkstra est utilisé en théorie des graphes pour résoudre des problématiques liées à l'optimisation des distances entre deux points. A partir d'un graphe de départ qui sera dans notre cas d'application le `stations_hashmap` associé au `connection_hashmap`. Le premier consituera les nœuds du graphe et le deuxième le poids des arcs entre chaque paires de nœuds. Il s'agit de construire progressivement un sous-graphe dans lequel sont classés les différents sommets par ordre croissant de leur distance minimale au sommet de départ. La distance correspond à la somme des poids des arcs empruntés.

L'algortihme se déroule en 4 parties :

1. **Initialisation :** Cette partie consiste à initialiser les distances correspondant à chaque nœud à une valeur infinie sauf celle du nœud de départ `Start` qui sera initialiser à zéro.

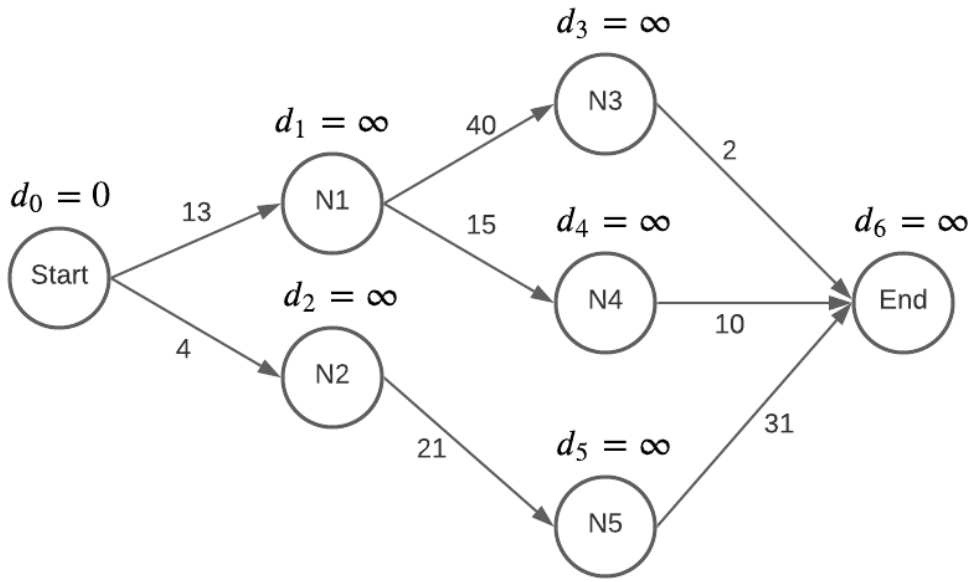

FIGURE 1 – Initialisation des distances.

2. **Recherche de la distance minimale :** A partir du graphe et des distances initialiser l'algorithme va chercher la distance minimale en parcourant le conteneur des distances. A la première ittérations le minimum correspondra forcémment au nœud `Start`. Le minimum sera alors retiré de liste des nœud car il sera considéré comme étant parcouru.

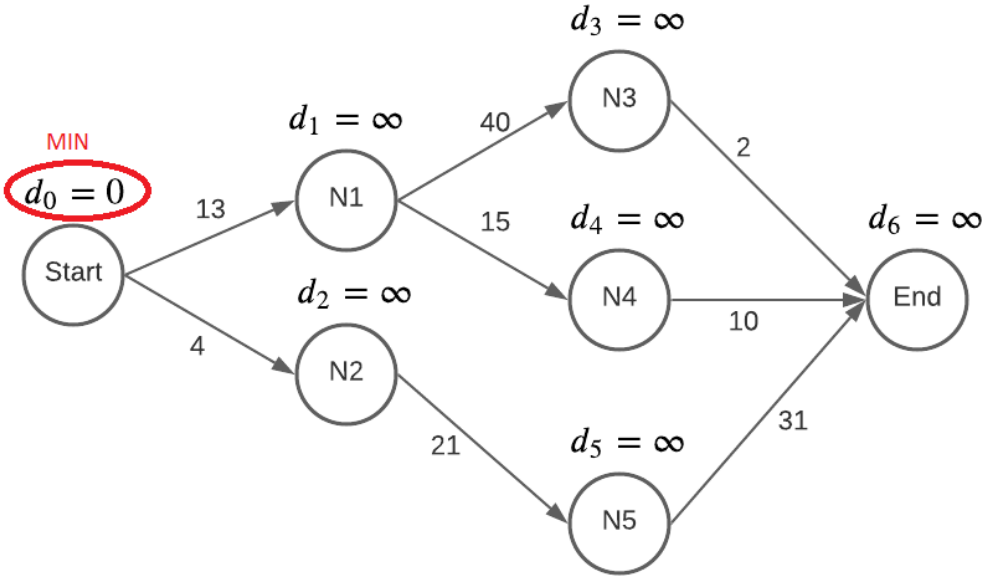

FIGURE 2 – Recherche de la distance minimale.

3. **Mise à jour des distances :** Cette étape consiste à parcourir les connexions du nœud correspondant à la distance minimale et à mettre à jour le conteneur des distances et à attribuer à chaque noeud un prédecesseur dans le but d'enregistrer en mémoire les nœeuds parcourus jusqu'au nœud courant. Les étapes de recherche de distance minimale et de mise à jour des distances s'enchainent jusqu'à ce que tous les nœuds du graphe soient parcourus.

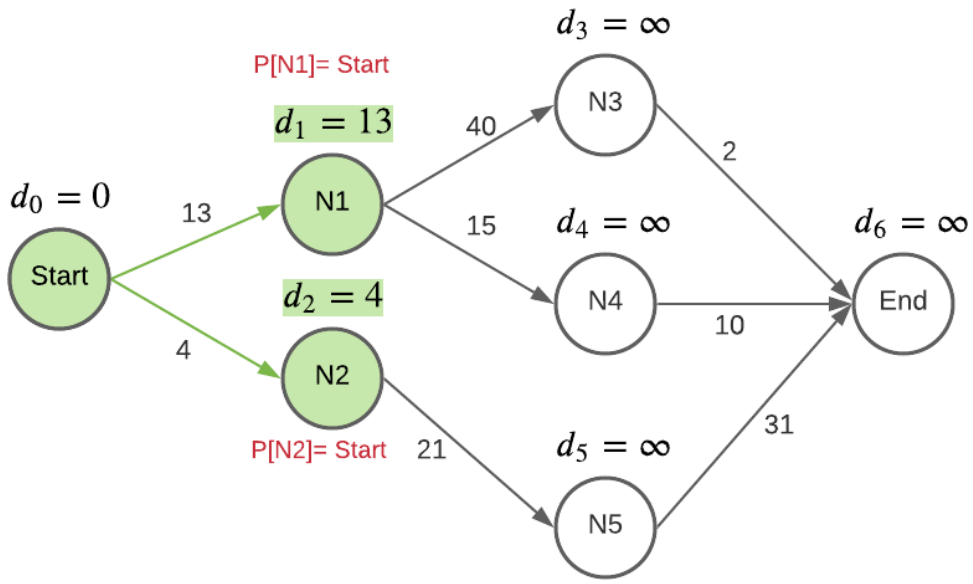

FIGURE 3 – Mise à jour des distances.

4. **Recherche du chemin le plus court :** A partir du nœud d'arrivée `End`, nous allons parcourir les nœuds en utilisant le conteneur des predecesseurs jusqu'à ce que le nœud courant soit égal au nœud de départ `Start`.

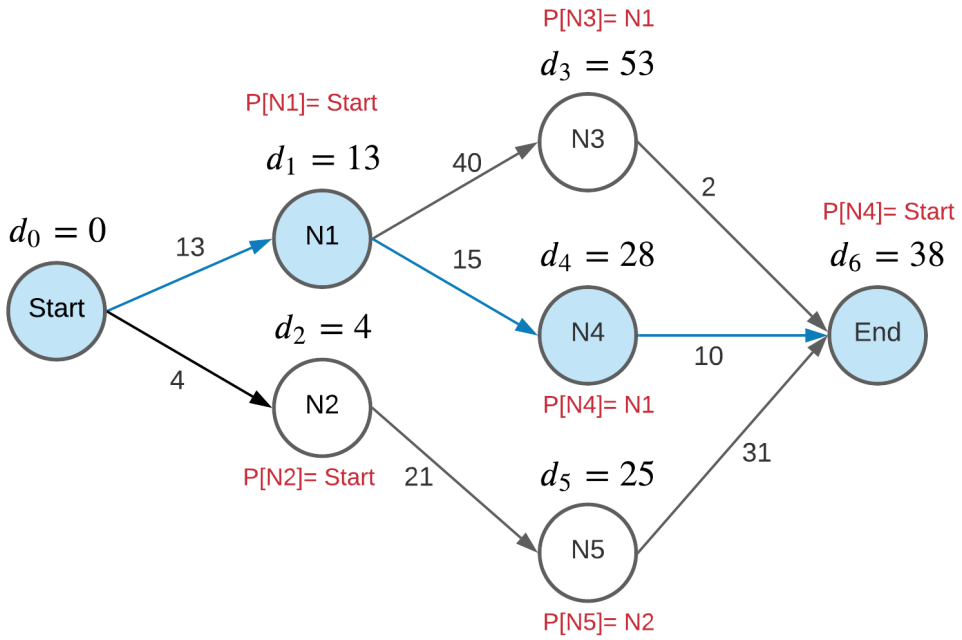

FIGURE 4 – Meilleur chemin.

Algorithme :
distance[*i*] ← ∞ ;
distance[*start*] ← 0 ;
tant que *Q n'est pas vide* **faire**
  mini := ∞
  pour *chaque station_id i présent dans Q* **faire**
    si *distance [i] < mini* **alors**
      mini = distance [i]
      S = i
    fin
  fin
  Retirer S de Q.
  pour *chaque sommet S2 voisin de S* **faire**
    si *distance [S2] > distance[S] + connection [S][S2]* **alors**
      distance [S2] = distance[S] + connection [S][S2]
      predecesseur[S2] = S
    fin
  fin
fin
sommet := End
tant que *sommet est different de Start* **faire**
  Ajouter le sommet au chemin.
  sommet = predecesseur[sommet]
fin

***NB : Sommet = Nœud.**

Problèmes rencontés : Lors de l'implémentation de l'algorithme je n'ai pas rencontré de problème majeur. Le seul problème rencontré etait une boucle infinie causée par une mauvaise initialisation de la variable `mini` à chaque début de boucle. Ce qui ne permettait pas de changer de nœud courant car le minimum était toujours de 0, minimum qui correspondait à la distance du point de départ `Start`.

4 Choix des conteneurs :

Dans ce projet nous avons utilisé trois type de conteneurs :

1. `std::unordered_map:` Ce conteneur a été utilisé pour stocker le `stations_hashmap` et le `connection_hashmap`. Etant donnée que les élements de ces deux structure disposent d'index qui ne se suivent pas forcémment. Les `std::unordered_map` sont très utiles pour ce type de logique de stockage.
2. `std::vector:` Ce type de container a été utilisé pour stocker les différents nœuds du chemin le plus court car l'indice de stockage ne nous interesse pas. Le séquencement des nœuds étant plus inmportant est assuré par gràce par la fonction `push_back()`.
3. `std::pair:` Ce type de container est très utile pour associer deux objets. Dans notre cas, il a été utilisé pour associé chaque nœud du chemin et le temps nécessaire pour arriver à ce dernier.

Conclusion

En plus d'être simple à implémenter l'algorithme Dijkstra est très puissant et a prouvé sont efficacité dans plusieurs applications. Une application courante de cet algorithme apparait dans les protocoles de routage sur les réseaux ATM, qui permettent le routage internet.