Rapport du Projet C++

Peilin LI

Introduction :

Le projet nous a lancé un défi montrant l'utilité de l'informatique dans les applications quotidiennes et un exercice rigoureux en algorithmie et conception logicielle. En tant que membres d'une startup fictive spécialisée dans la vente de données de trafic, nous devions créer un algorithme pour planifier les temps de trajet en utilisant une heuristique fixe ou dynamique. La fiabilité de notre solution reposait sur une base de données réaliste, ce qui nous a conduit à utiliser l'algorithme de Dijkstra et la base de données RATP.

L'algorithme de Dijkstra, un pilier de la théorie des graphes, a été la base de notre travail. Il calcule le chemin le plus court entre deux points, ce qui est crucial pour les réseaux de transport. En modélisant les stations de métro comme des nœuds et les connexions entre elles comme des arêtes, nous avons construit un graphe dirigé représentant le réseau complexe du métro parisien.

Nous avons suivi une approche méthodique, avec une feuille de route détaillée pour l'instanciation des classes, la lecture de fichiers, la génération de graphes et la détermination des meilleurs chemins. Nous avons utilisé la Standard Template Library (STL) pour gérer efficacement les structures de données, notamment std::unordered_map pour les tableaux associatifs. Les fonctionnalités de la STL ont simplifié notre implémentation et montré l'importance des cadres bien établis pour gérer la complexité et garantir la maintenabilité.

Étapes de mise en œuvre pour chaque partie :

Generic_my_station_parser.hpp

Il hérite de la classe **Generic_station_parser** et implémente une fonction virtuelle nommée **read_stations** utilisé pour lire les informations de la station à partir d'un fichier et les stocker dans une table de hachage.

1. Créez un objet flux de fichier en entrée **‘infile’** et ouvrez le fichier spécifié. Si l'ouverture du fichier échoue, affichez un message d'erreur et retournez.
2. Définissez une variable de type chaîne de caractères **‘line’** pour stocker le contenu de chaque ligne, et **‘line_num’** pour enregistrer le numéro de ligne actuel.
3. Lisez chaque ligne du fichier et utilisez la fonction **‘split_string_by_string (line, ",")’** pour diviser le contenu de la ligne par des virgules en plusieurs chaînes de caractères, stockées dans un vecteur **tokens**.
4. Créez un objet ‘**Station’** et affectez les informations analysées aux propriétés correspondantes.
5. Insérer l'objet Station dans la table de hachage **‘stations_hashmap’**, avec la clé correspondant à l'ID de la ligne **‘line_id’**.
6. Générez la table de hachage **stations_id_hashmap_by_name** pour permettre une recherche par nom de station.
7. Si une exception se produit lors de la lecture, vider la table de hachage et affichez un message d'erreur.

Generic_my_connection_parser.hpp

Il hérite de la classe **Generic_connection_parser** et implémente deux fonctions virtuelles nommées **read_connections** et read_stations utilisées pour lire les informations de la station et de la connexion à partir de fichiers et les stocker dans des tables de hachage.

1. Créez un objet flux de fichier en entrée **‘infile’** et ouvrez le fichier spécifié. Si l'ouverture du fichier échoue,

affichez un message d'erreur et retournez.

2. Définissez une variable de type chaîne de caractères **‘line’** pour stocker le contenu de chaque ligne, et **‘line_num’** pour enregistrer le numéro de ligne actuel.
3. Lisez chaque ligne du fichier et utilisez la fonction **‘split_string_by_string (line, ",")’** pour diviser le contenu de la ligne par des virgules en plusieurs chaînes de caractères, stockées dans un vecteur ‘tokens’.
4. Utilisez **‘std::stoull’** pour convertir les chaînes de caractères en type **‘unsigned long long’** et insérez les valeurs dans la table de hachage **‘connections_hashmap’**.
5. Si une exception se produit lors de la lecture, vider la table de hachage et affichez un message d'erreur.

Generic_my_connection_parser.hpp

Il hérite de la classe **Generic_mapper** et implémente plusieurs fonctions virtuelles nommées **compute_travel**, **compute_and_display_travel**, read_connections, et read_stations utilisées pour lire les informations des stations et des connexions à partir de fichiers et les stocker dans des tables de hachage, ainsi que pour utiliser algorithme djsk pour calculer et afficher les itinéraires de voyage.

Fonction ‘compute_travel (uint64_t _start, uint64_t _end’ :

Cette fonction compute_travel a pour but de calculer le chemin le plus court entre le point de départ _start et le point d'arrivée _end. Elle utilise l'algorithme de Dijkstra, en se servant d'une file de priorité et de tables de hachage pour enregistrer la distance la plus courte vers chaque nœud ainsi que son nœud prédécesseur. Si le point de départ ou le point d'arrivée n'existe pas, la fonction renvoie un vecteur vide. Sinon, elle renvoie un vecteur contenant chaque nœud du chemin ainsi que la distance cumulée jusqu'à ce nœud.

1. Vérifiez si le point de début ou de fin existe dans **connections_hashmap**.
2. Utilisez **std::priority_queue** pour créer une file d'attente prioritaire **‘q’** Le type de données est **‘vector’** et la méthode de tri est **‘greater’** (c'est-à-dire que la plus petite valeur a la priorité la plus élevée).
3. Créez un **unordered_set** pour détecter si un nœud a été visité et deux **unordered_map** pour enregistrer la plus courte distance et le prédécesseur de chaque nœud.
    - **vis** : Un **unordered_set** pour garder une trace des nœuds qui ont été visités, évitant ainsi des cycles et des recalculs inutiles.
    - **dis** : Un **unordered_map** qui enregistre la plus courte distance trouvée pour atteindre chaque nœud à partir du nœud de départ.
    - **pre** : Un **unordered_map** qui enregistre le prédécesseur de chaque nœud sur le chemin le plus court, ce qui est essentiel pour reconstruire le chemin final une fois le nœud de fin atteint.

4. Insérez le point de départ dans la file d'attente prioritaire **q** et définissez la distance initiale à 0.
5. Mettez à jour q via une boucle.
    - La boucle principale traite les nœuds de la file d'attente q jusqu'à ce que celle-ci soit vide ou que le nœud de fin soit atteint. À chaque itération :
        1. Le nœud avec la plus petite distance cumulée est extrait de **q**.
        2. Si ce nœud a déjà été visité, l'itération est sautée.
        3. Si le nœud extrait est le nœud de fin, la boucle se termine car le plus court chemin a été trouvé.
        4. Pour chaque voisin du nœud actuel, la distance cumulée pour atteindre ce voisin est calculée. Si cette nouvelle distance est plus courte que celle précédemment enregistrée, les informations de distance et de prédécesseur sont mises à jour, et le voisin est ajouté à la file d'attente **q**.
6. Initialisez **res** pour sauvegarder le chemin. **res** est un vecteur de std::pair<uint64_t, uint64_t> qui stockera le chemin du nœud de fin au nœud de départ, avec chaque nœud et sa distance cumulée.

7. Confirmez si le nœud de fin est atteignable. Si le nœud de fin n'a pas été visité après la boucle principale, cela signifie qu'il est inatteignable à partir du nœud de départ, et un vecteur vide est retourné.
8. Reconstituez le chemin du nœud de fin au nœud de départ. Si le nœud de fin est atteignable, le chemin est reconstitué en suivant les prédécesseurs enregistrés dans **pre**, du nœud de fin au nœud de départ. Le vecteur **res** est inversé pour passer de l'ordre du nœud de fin au nœud de départ à l'ordre du nœud de départ au nœud de fin, correspondant ainsi au chemin réel parcouru.

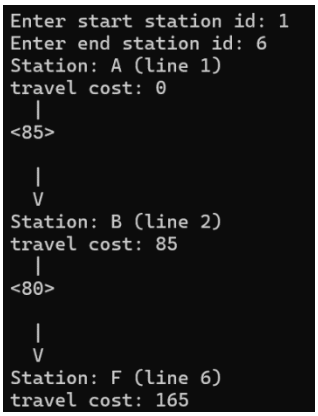
Fonction 'compute_travel(const std::string& _start, const std::string& _end)' :

Cette fonction nous permet d'interroger la distance la plus proche entre deux sites en utilisant les noms en entrée, Il appelle la fonction **'compute_travel'** ci-dessus en convertissant le nom de notre station en ID.

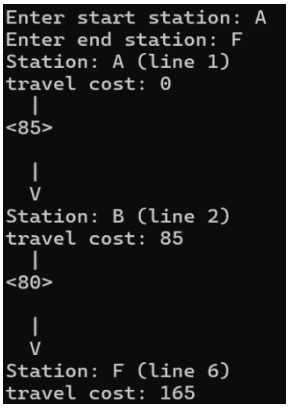
1. Vérifiez si le nom de la station existe dans la table de hachage **stations_id_hashmap_by_name**.
2. Si le nom n'existe pas, retournez un pointeur nul (nullptr).
3. Si le nom existe, récupérez l'ID de la station à partir de la table de hachage.
4. Utilisez l'ID pour obtenir l'objet Station correspondant à partir de la table de hachage **stations_hashmap**.
5. Utilisez les ID convertis des stations de départ et d'arrivée pour détecter le chemin le plus court en utilisant la fonction **compute_travel(uint64_t _start, uint64_t _end)** mentionnée ci-dessus.

Présentation des résultats :

1. Détection avec de petites données

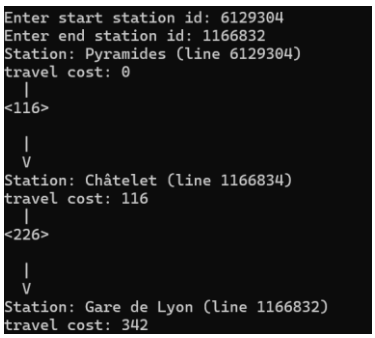


(Par ID)



(Par nom)

2. Détection avec de grandes données



(Par ID)