[4AI02] Projet de C++ Avancé: Application RATP

9 Avril 2019

Trois séances de 4H sont mises à disposition pour réaliser ce projet. Les étapes à réaliser sont suggérées dans la feuille de route au chapitre 5. Le rythme n'est pas imposé étant donné que la soumission du projet est possible à tout moment lors de la période du projet. Des programmes de test sont fournis afin d'évaluer le code et d'estimer le niveau d'accomplissement du projet.

Notation :

- Instantiation des classes (+3pts => 3/20);
- Implémentation de read_stations (+3pts => 6/20);
- Implémentation de read_connections (+3pts => 9/20);
- Génération du graphe (+4pts => 13/20);
- Estimation du meilleur chemin et affichage (+7pts => 20/20);
- Amélioration (+5pts => 25/20);
- Les critères non respectés de la charte d'écriture (Voir Chapitre 2) fera perdre jusqu'a 1 point par critère (−1pt).
- Une mauvaise gestion des exceptions fera perdre 2 points (−2pts).

Les points au dessus de 20 pourront être utilisés comme bonus.

Date limite du rendu du code : **Vendredi 24 mai 2019, 20h00**. Chaque minute de retard fera perdre un point sur la note du projet. Un rendu doit contenir le code nécéssaire pour pouvoir compiler le projet en une seule commande, de même pour l'éxécution. Ces commandes peuvent être définies par exemple dans un fichier *lisez-moi*.

La soumission du projet se fera sur l'espace Moodle dédié. La soumission multiple est possible, la dernière soumission écrasant les précédentes. Il est fortement encouragé de soumettre à chaque étape de résolution du projet.

1 Introduction

Vous êtes membre d'une start-up qui veut se lancer sur le marché de la vente de données de trafic. L'équipe de développement vous demande d'implémenter un algorithme capable de planifier des temps de parcours au moyen d'une heuristique fixe, ou dynamique. Une base de données réaliste et fiable pour le tester est requise. Celle-ci permettera d'établir la validité de l'algorithme, et du cadre dans lequel va naître le projet. L'algorithme de Dijkstra et la base de données de la RATP sont des choix logiques car robustes. Après concertation, un contrat de programmation est écrit, afin de juger des fonctionnalités minimales permettant l'évolutivité de la solution. La tâche est de réaliser un programme respectant la structure imposée. Une recherche sera nécéssaire afin de maîtriser l'algorithme de Dijkstra, afin de pouvoir l'implémenter.

1.1 Une résolution du problème du plus court chemin

Le but de ce projet est de faire une application qui établit le trajet le plus court entre deux stations du réseau de métro parisien.

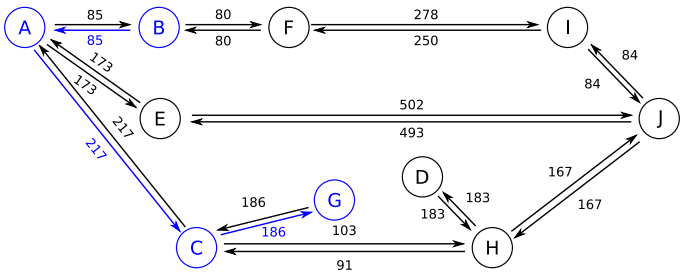

Figure 1: Exemple de graphe orienté

En théorie des graphes, un graphe orienté est un ensemble de points (ici, des entrées de stations de métro) connectés par des liens orientés (ici, les connexions entre les stations, en secondes). Dans le cadre de ce projet, chaque lien représente une possibilité de relier deux points, dans un sens précis, et avec un coût déterminé. C'est une connection possible entre ces points.

Le problème du plus court chemin, dans ce système, est d'évaluer le coût minimal pour aller d'un point A à un point B. L'algorithme de Dijkstra est une méthode de calcul afin de résoudre ce problème.

Par exemple, dans le graphe de la Figure 1, le chemin le plus court entre le noeud B et le noeud G passe par les noeuds A et C, pour un coût cumulé de 85+217+186=488. Ce chemin peut être exprimé comme une liste de noeuds et le coût cumulé du chemin, ici, $best\_path = [\{B, 0\}, \{A, 85\}, \{C, 302\}, \{D, 488\}]$;

Cet algorithme peut s'appliquer sur des graphes orientés, modèle auquel plusieurs réseaux de transports sont disponibles (RATP, SNCF, routes, etc), et est également une porte d'entrée pour le domaine de la théorie des graphes et de la théorie de la décision.

1.2 Méthode de développement, et outils de la *STL*

Un programme informatique est une suite d'instructions mathématiques. A ce titre il existe deux difficultés lorsque l'on veut construire un programme informatique pour répondre à un problème :

1. Quel jeu d'instructions choisir pour répondre au problème algorithmique?
   La solution ici a été trouvée par l'informaticien Edsger Dijkstra, répondant au problème du plus court chemin énoncée ci-dessus.

2. Comment structurer, implémenter mon problème algorithmique, qui requiert ici la construction de centaines de milliers d'instructions machine, avec la méthodologie nécéssaire pour que n'importe quel humain puisse maintenir ce code et y ajouter des fonctionnalités?
   Cette question est en partie résolue par l'outil C++. C'est une méthodologie, permettant de générer une grande compléxité dans le jeu d'instructions (comme beaucoup d'outils informatiques), mais également d'utiliser une interface compréhensible, qui vous est donnée ici, et va permettre l'ajout de fonctionnalités de manière concise et ergonomique.

Afin de résoudre ce problème, des fichiers contenant les points d'un graphe, et contenant des connexions, sont mis à votre disposition. La lecture de ces fichiers devra être réalisée avec des conteneurs spécifiques imposés. Puis, un calcul du chemin le plus court devra être réalisé.

Afin de travailler les connaissances vues en cours, nous allons utiliser la *Standard Template Library*, une bibliothèque C++ parmi tant d'autres, mais qui a l'avantage d'être très puissante dans plusieurs modalités, notamment pour les conteneurs que celle-ci propose. Ceux-ci représentent des tableaux associatifs, généralisation des tableaux où les indices peuvent ne pas être des entiers.

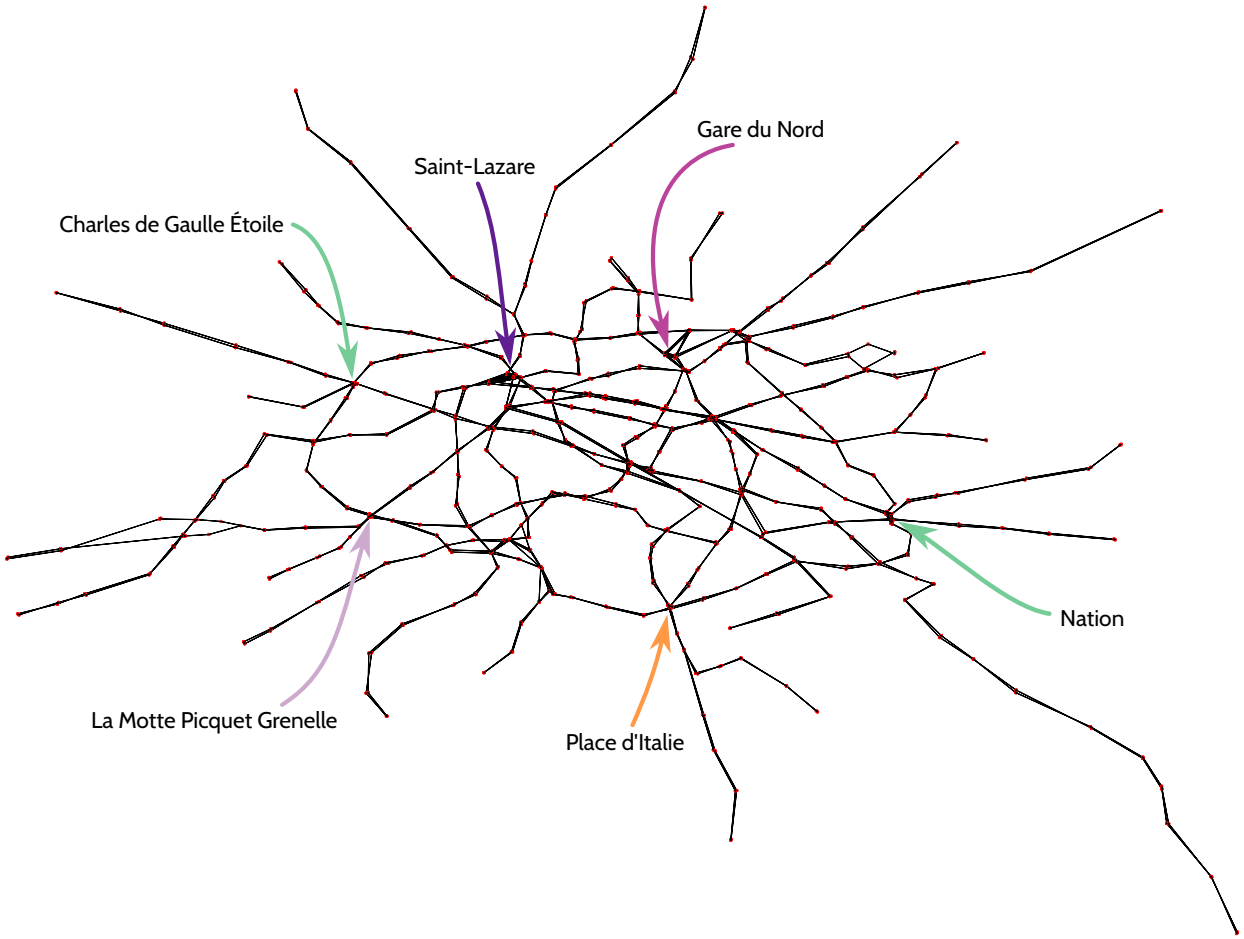

Figure 2: Graphe orienté généré à partir des fichiers `c.csv` et `s.csv`, avec les données GPS des stations.

Une table de hachage est une structure de données efficace pour représenter un tableau associatif, avec une réprésentation mémoire compacte, via le hachage des données qui, dans ce cas, associe une information (la station) à une clé (l'identifiant de station). Les éléments à insérer dans le conteneur ne sont pas triés. Ce sont les raisons pour lesquelles nous utiliseront le conteneur `std::unordered_map`, similaire d'un point de vue utilisateur au conteneur `std::map` vu en cours, qui implémente lui un arbre binaire de recherche pour garantir l'ordre des données.

Ces conteneurs vont garder les données du graphe fourni via des fichiers qu'il faudra lire et dont les données doivent être extraites. Dans ce cas précis, l'utilisation de la bibliothèque standard est fortement conseillé, via la classe `std::ifstream`, prenant en argument le nom du fichier à lire. Une des différentes méthodes pour lire une ligne est d'utiliser la fonction `std::getline` définie dans `<string>`.

Une auto-évaluation vous est proposée au chapitre 4, afin de construire progressivement les fonctions nécéssaires à l'accomplissement du projet, et d'ajuster le temps nécéssaire pour le terminer.

2 Charte d'écriture

Ce projet vous impose une charte d'écriture afin de pouvoir maintenir le code et faciliter sa transmission. Cette charte est énoncée dans un ordre quelconque:

- **Utilisation des drapeaux de compilation -Wall -Wextra -Werror -pedantic -pedantic-errors -O3 de g++** afin de garantir le respect de l'implémentation dans les normes C++, et d'optimiser la compilation;
- **Utilisation des fonctionnalités du C++ 11 maximum** afin d'utiliser les ressources du cours. Le drapeau de compilation *std=c++11* pourra être utilisé;
- **Minimisation de l'utilisation des outils d'allocation dynamique** : *new*, *new[]*, *delete* et *delete[]*, afin d'optimiser la sécurité, la lisibilité, et de tirer parti du concept *RAII* : l'acquisition d'une ressource est une initialisation. Ce projet est faisable sans allocation dynamique, cela sera apprécié et simplifiera votre projet;
- **Mémoire dynamique via les conteneurs de la *STL***;
- **Variables nommées par leur utilité**, ayant un sens, reflétant leur fonction dans le programme, et permettant d'éviter les commentaires redondants;
- **Indentation parfaite du code source du programme** afin de garantir la lisibilité et de prévenir les erreurs d'implémentation;
- **Minimisation de duplication du code source du programme** afin d'éviter le dangereux copier-coller et les graves problèmes de maintenance induits par ce comportement;
- **Utilisation de la convention *snake_case***, avec des noms de fonctions en *verbe_complément*, de variables en *complément_nom*, afin de garantir la lisibilité du code;
- **Tout type créé commence par une majuscule**, par convention;
- **Les constantes déclarées via *define* sont écrites intégralement en majuscules**; on parlera ici de *SCREAMING_SNAKE_CASE*;
- **Membres de classes appelés via *this*** afin de répèrer facilement l'utilisation et la modification d'outils interne;
- **Utilisation du mot clef *const* lorsque l'entrée ne doit rester la même**, afin de garantir ses droits d'accès en lecture/écriture;
- **Utilisation systématique de référence ou de pointeur lors du passage d'un objet**, les objets seront passés par copie;

3 Objectifs

Un des buts de ce projet est de permettre l'assemblage de code de plus en plus complet, avec un écosystème compréhensible alors qu'il grossit. La méthodologie va montrer cette difficulté, et c'est pourquoi les objets englobent ces fonctionnalités. Le point le plus important dans le cadre de ce projet est de comprendre l'interface, la structure imposée. Le travail demandé, qui est d'implémenter les fonctionnalités clés, nécéssite la compréhension de cette structure. Il est aussi judicieux de se renseigner sur les outils possibles et nécéssaires pour réussir le projet.

Les classes doivent hériter d'une classe mère virtuelle pure sans implémentation. Ce principe permet de rajouter des fonctionnalités à une classe sans l'altérer et en garantissant son appel via un pointeur. Si la classe respecte le contrat, elle garantit d'avoir implémenté les fonctionnalités nécéssaires à la validité du programme. Cela permet de pouvoir améliorer et ajouter diverses fonctionnalités supplémentaires, notamment celle proposée en amélioration dans le chapitre 5.

4 Auto-évaluation

Parmi les fichiers de l'archive du projet se trouve la classe `Grade` au travers du fichier d'entête `Grade.hpp` et du fichier objet `Grade.o`. C'est une aide, l'utilisation de `Grade` est optionelle et ne fait pas partie de l'évaluation du projet.

Ce fichier objet, compilé via le compilateur g++ avec un ordinateur sous Unix, permet une auto-évaluation.
En incluant le fichier binaire à la compilation, et l'entête au programme principal du projet, deux objets statiques seront disponibles. Ceux-ci permettent l'évaluation des différentes parties du projet. Pour rappel, un objet statique est un objet alloué au lancement du programme et détruit à sa toute fin (par exemple : `std::cout`, `std::cin`). Il est non rédéfinissable, pour l'ensemble du code à sa portée.

Ces deux objets se nomment ici `travel::evaluate_small` et `travel::evaluate`. Le premier permet d'évaluer le réseau simple, le second évaluant le réseau RATP.

Trois méthodes sont disponibles :

- `stations(const travel::Generic_station_parser&)`, qui évalue de manière exhaustive la construction de la table de stations.
- `connections(const travel::Generic_connection_parser&)`, qui évalue de manière exhaustive la construction de la table de stations et de connections.
- `dijkstra(travel::Generic_mapper&,bool)`, qui évalue dans un premier temps quelques trajets tests, puis propose une exploration exhaustive des trajets via une estimation grossière du temps de calcul.
  - Si l'argument booléen est vrai : L'évaluation du projet est faite dans sa version de fin de séances de TP. Le temps de calcul pour le réseau RATP est d'environ 0.3ms/trajet, pour 6 minutes de test.
  - Si l'argument booléen est faux : L'évaluation du projet est faite dans sa version améliorée, via l'utilisation de noms de stations. Le temps de calcul pour le réseau RATP est cinq fois plus long.

Attention : La dernière méthode `dijkstra(travel::Generic_mapper&,bool)` va vérifier la validité des trajets, et ceci dans un certain format. Ce format est d'avoir une liste de noeuds, le premier noeud et le dernier noeud du trajet étant dans cette liste, et avec un coût cumulé associé à chaque noeud, comme dans l'exemple de la section 1.1.

Lorsque une méthode est appelée, elle appelle automatiquement les méthodes précédentes pour garantir le maintien des acquis au fur et à mesure de l'avancement du projet ; Il est donc facultatif d'appeler les anciens tests.
De plus, il est à noter que la classe de test est totalement indépendante des classes abstraites et apporte toujours une résolution via les fichiers de références du projet. En définitive, les appels à cette classe seront appréciés, mais sont facultatifs dans le cadre du projet.
L'interface est imposée, et que vous ne devez pas modifier ces fichiers. La correction sera faite avec seulement l'implémentation de votre classe. Le code doit donc être fonctionnel avec les fichiers récupérés, au risque d'être pénalisé sinon.

5 Feuille de route

L'équipe de développement vous propose une feuille de route, permettant de représenter les diverses tâches et l'avancement du projet. Dans le cadre des TP, cette feuille de route est facultative, mais est conseillé afin de vous guider :

1. Instanciez une classe dérivée : faites un nouveau fichier, une inclusion de la classe parente et créez une classe qui hérite de la classe mère `Generic_station_parser`.
   Compilez votre fichier de classe. Celui-ci doit générer aucune erreur, même avec les restrictions de compilation imposées dans la charte d'écriture.
2. Créez un fichier avec une fonction principale, et instanciez un objet de votre classe. Le code compile-t-il? Pourquoi?
3. Surchargez la fonction `Generic_station_parser::read_stations` : Dans la déclaration de la classe, écrivez le prototype de cette fonction `protected` en ajoutant à la fin le mot clé `override`. Implémentez cette fonction.
4. Dans votre fonction principale, instanciez votre classe et appelez la méthode `stations` de la classe `Grade`. Vous utiliserez l'objet statique mis a disposition dans cette classe, selon la base de données utilisée.

   A ce point, vous devriez avoir soumis une version fonctionnelle de votre projet sur Moodle, avant la deuxième séance de TP.

5. Changez votre classe pour qu'elle hérite de la classe mère `Generic_connection_parser`.
   Le code compile-t-il, pourquoi?
6. Implémentez la fonction `read_connections`.
7. Instanciez votre classe et appelez la méthode `connections` de la classe `Grade`.

   A ce point, vous devriez avoir soumis une nouvelle version fonctionnelle de votre projet sur Moodle, avant la troisième séance de TP.

8. Changez votre classe pour qu'elle hérite de la classe mère `Generic_mapper`.
   Le code compile-t-il, pourquoi?
9. Implémentez la fonction `compute_travel`, ayant comme arguments d'entrée des nombres correspondant aux identifiants de station. Cette fonction rend un vecteur de `std::pair`, contenant l'identifiant de la station et le coût associé de la connexion.
10. Implémentez la fonction `compute_and_display_travel`, permettant d'afficher le chemin calculé par la fonction précédent pour relier les deux stations que vous voulez relier. La notation est dépendante de la clarté de l'affichage, étant donné que le client doit suivre facilement des instructions claires pour se déplacer.
11. Instanciez votre classe et appelez la méthode `dijkstra` de la classe `Grade`, avec l'argument booléen à `false`.

    A ce point, vous avez débloqué l'accès aux 20 premiers points de la notation sur le code du projet.

12. Implémentez une surcharge des fonctions qui estiment le trajet minimal pour relier les stations via leurs noms, et non plus leurs identifiants. Il serait convenable d'être résistant aux erreurs (casse, mauvaise orthographe, etc) lors de la saisie des noms par l'utilisateur.
13. Instanciez votre classe et appelez la méthode `dijkstra` de la classe `Grade`, avec l'argument booléen à `true`.

A Interface : Generic_station_parser.hpp

```
#include <string>
#include <iostream>
#include <unordered_map>

namespace travel{
  struct Station{
    std::string name;
    std::string line_id;
    std::string address;
    std::string line_name;
  };

  class Generic_station_parser{
  protected:
    virtual void read_stations(const std::string& _filename) = 0;

  protected:
    std::unordered_map<uint64_t, Station> stations_hashmap;
  };
}
```

B Interface : Generic_connection_parser.hpp

```
#include "Generic_station_parser.hpp"

namespace travel{
  class Generic_connection_parser: public Generic_station_parser{

  protected:
    virtual void read_connections(const std::string& _filename) = 0;

  protected:
    std::unordered_map<uint64_t,std::unordered_map<uint64_t,uint64_t> > connections_hashmap;
  };
}
```

C Interface : Generic_mapper.hpp

```
#include <vector>
#include <utility>

#include "Generic_connection_parser.hpp"

namespace travel{
  class Generic_mapper: public Generic_connection_parser{
  public:
    virtual std::vector<std::pair<uint64_t,uint64_t> > compute_travel(uint64_t _start,
        uint64_t _end) = 0;
    virtual std::vector<std::pair<uint64_t,uint64_t> > compute_and_display_travel(uint64_t
        _start, uint64_t _end) = 0;
  };
}
```