L3 MIASHS
Projet Graphe/C++

Objectif général

Dans le cadre de l'initiative Open Data, la Communauté Urbaine de Toulouse Métropole rend disponible un catalogue de données ouvertes [1]. Parmi les jeux des données disponibles, vous trouverez les données sur les stations vélo en libre service [2]. L'objectif est de proposer un algorithme pour optimiser le rechargement des stations vélo. Cet algorithme doit calculer dynamiquement le plus court chemin passant pour tous les stations qui doivent être rechargées. Ce projet sera développé en C++. Le projet doit donc vous amener à utiliser et à développer vos compétences en modèles, algorithmes et applications de la théorie de graphes et en programmation C++.

Partie 1 : modélisation

La première partie du projet consiste à modéliser le scénario sous forme de graphe. Pour cela, vous allez vous appuyer sur deux sources de données :

— données statiques : fournissent des informations comme les coordonnées GPS (latitude et longitude) des 281 stations (Figure 1), leur adresse, la présence d'un terminal de paiement, etc. Ces données sont disponibles en format csv [3] ;

— données dynamiques : indiquent l'état d'une station, le nombre de vélos disponibles, le nombre d'emplacements libres, etc. Ces données sont rafraîchies régulièrement et ne sont accessibles que depuis une API.

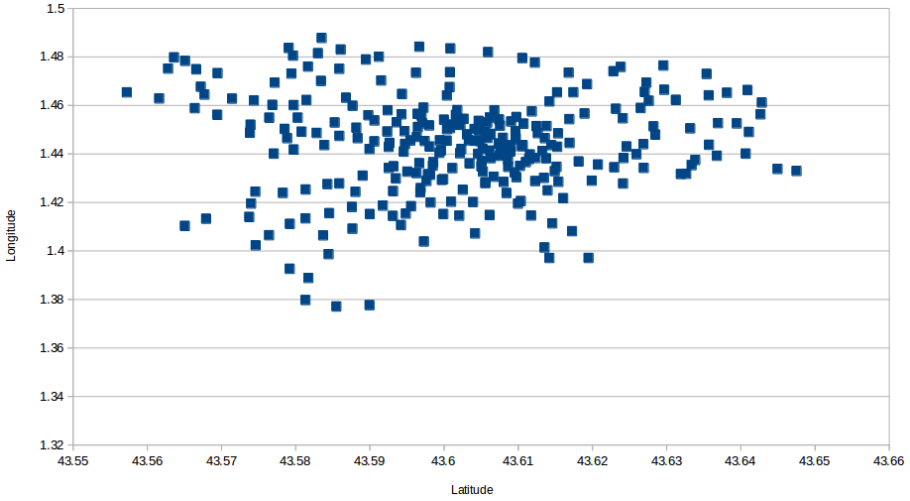

FIGURE 1 – Latitude et longitude des 281 stations vélo de la métropole de Toulouse.

Votre application doit être capable de restituer un graphe connexe à partir de ces deux sources de données. Deux solutions sont envisageables :

1. https://data.toulouse-metropole.fr/
2. https://developer.jcdecaux.com/#/opendata/vls?page=getstarted
3. https://developer.jcdecaux.com/#/opendata/vls?page=static

1. votre graphe contiendra toutes les stations du réseaux et l'algorithme de parcours ne prendra en compte que les stations à recharger (une station vélo doit être rechargée si le nombre de vélos disponibles est inférieur à un seuil donné) ;
2. votre graphe ne contiendra que les stations à recharger.

Vous devrez définir une stratégie pour relier les sommets de votre graphe (c'est à dire, pour déterminer si deux sommets sont adjacents). Ne vous basez pas sur l'hypothèse d'un graphe complet. Les poids des arrêtes (ou arcs) de votre graphe correspondent à la distance entre les deux sommets. Pour deux sommets adjacents, vous devrez représenter (toutes) les directions possibles entre eux. Pour cela, vous pouvez utiliser le *Google Directions API*[4] (plus d'informations dans la section suivante).

Dans un premier temps, vous pouvez vous concentrer sur un sous-ensemble des 281 stations (par exemple, en choisissant les n stations autour d'une coordonnée GPS donnée, où n >20).

Partie 2 : développement

Vous devrez développer un algorithme qui prend en entrée un graphe représentant le réseaux des stations vélo et qui retourne le plus court chemin passant par tous les sommets représentant les stations qui doivent être rechargées :

— votre application doit être capable de construire un graphe à partir d'un ou plusieurs fichiers d'entrée (format csv, JSON, ou autre) ;
— choisissez une façon appropriée pour représenter votre graphe (matrice d'adjacence, liste d'adjacence, ou matrice d'incidence sommets-arcs) ;
— identifiez le problème parmi les problèmes connus de la théorie de graphes et implémentez l'algorithme correspondant (vous êtes libres de proposer une adaptation pour l'algorithme en question) ;
— votre application doit renvoyer le plus court chemin sous forme d'une liste de sommets à parcourir (avec la distance entre chaque pair de sommets). Le développement de toute interface graphique (par exemple, l'affichage d'un plan à la Google à partir d'un fichier généré par l'application) sera considéré comme un bonus ;
— vous n'avez pas besoin de prendre en compte les possibles changements, en temps réel, du nombre de vélos disponibles dans une station, au cours de l'exécution de l'algorithme ;
— exploitez au mieux les ressources du langage C++ (classes, modèles, conteneurs de la bibliothèque standard). L'évaluation prendra en compte également l'organisation et la clarté de votre code (fichiers d'en-tête, fichiers .cpp, documentation, etc).

Vous trouverez ci-dessous des fragments de code qui vous seront utiles pour le développement de votre application.

Lecture/écriture d'un fichier

Pour la lecture et l'écriture de fichiers en C++ vous utiliserez les classes `ifstream` (flux de fichier d'entrée) et `ofstream` (flux de fichier de sortie), disponibles dans l'en-tête `fstream`. Un fichier est ouvert pour une entrée en créant un objet de la classe `ifstream` avec le nom du fichier comme argument. De la même façon, un fichier est ouvert pour la sortie en créant un objet de la classe `ofstream` avec le nom de fichier comme argument.

4. `https://developers.google.com/maps/documentation/directions/`

```cpp
#include <iostream>
#include <fstream>
#include <string>

int main() {
    ifstream fileIn ("projet/Toulouse.csv"); // read
    ofstream fileOut ("projet/Toulouse1.csv"); // write
    string line;
    if (fileIn.is_open()) {
        while (getline(fileIn,line)) { // read a line from the file
             cout << line << endl;
             fileOut << line << '\n'; // write a line in the file
        }
        fileIn.close();
        fileOut.close();
    } else {
        cout << "Erreur traitement fichier ! ";
    }
    return 0;
}
```

La bibliothèque standard ne fournit pas une façon simple de 'tokenizer' une chaîne de caractères (utile pour le traitement des fichiers csv). Pour cela, vous pouvez utiliser le code fourni ci-dessous [5] :

```cpp
void tokenize(const string& str,vector<string>& tokens, const string& delim = ",") {
    // Skip delimiters at beginning.
    string::size_type lastPos = str.find_first_not_of(delim, 0);
    // Find first "non-delimiter".
    string::size_type pos = str.find_first_of(delim, lastPos);
    while (string::npos != pos || string::npos != lastPos) {
        // Found a token, add it to the vector.
        tokens.push_back(str.substr(lastPos, pos - lastPos));
        // Skip delimiters.
        lastPos = str.find_first_not_of(delim, pos);
        // Find next "non-delimiter".
        pos = str.find_first_of(delim, lastPos);
    }
}
```

Requêtes HTTP et réponses JSON

Afin de pouvoir calculer (toutes) les directions possibles entre deux coordonnées données, des requêtes HTTP doivent être envoyées sur le service Google Directions API. Une réponse en format JSON est renvoyée. Voici un exemple de requête (notez les paramètres `origin`, `destination` et `alternatives`) : `https://maps.googleapis.com/maps/api/directions/json?origin=43.604134687,1.4454207807&destination=43.6048526522,1.4452858767&alternatives=true&key=AIzaSyAG3bhdapKXj9TpHlic9DgluyQ0Be_Hw5A`.

Afin de pouvoir utiliser ce service, vous devrez récupérer une clé d'autorisation. Vous êtes, cependant, libre de choisir une autre façon de calculer les distances entre deux coordonnées données.

La bibliothèque standard de C++ ne fournit pas un ensemble de classes pour la manipulation de requêtes HTTP. Pour cela, vous devez utiliser une bibliothèque externe. Ici, nous utiliserons la bibliothèque multi-plateforme C++ REST SDK [6]. Cependant, vous pouvez utiliser une autre bibliothèque de votre choix. Vous pouvez suivre la documentation [7] pour

5. Source : `http://www.oopweb.com/CPP/Documents/CPPHOWTO/Volume/C++Programming-HOWTO-7.html`
6. `https://casablanca.codeplex.com/`
7. `https://casablanca.codeplex.com/documentation`

télécharger, générer les binaires, et configurer cette bibliothèque sur votre système d'exploitation. Un exemple [8] détaillant l'utilisation de la bibliothèque est également disponible.

Ci-dessous vous trouverez un fragment de code qui peut être utilisé pour lancer une requête HTTP sur Google API Directions, en utilisant C++ REST SDK :

```
#include <cpprest/http_client.h>
#include <cpprest/json.h>

using namespace std;
using namespace web; // Common features like URIs.
using namespace web::http; // Common HTTP functionality
using namespace web::http::client; // HTTP client features

pplx::task<json::value> get_JSON_google_API() {

      // Set up the client
      http_client client(U("https://maps.googleapis.com/maps/api/directions"));
      http_request request(methods::GET);

      // Build request URI
      uri_builder builder(U("/json")); // U = UTF-32 encoded string literal
      builder.append_query(U("origin"), U("43.604134687,1.4454207807"));
      builder.append_query(U("destination"), U("43.6048526522,1.4452858767"));
      builder.append_query(U("alternatives"), U("true"));
      builder.append_query(U("key"), U("AIzaSyAG3bhdapKXj9TpHlic9DgluyQOBe_Hw5A"));

      request.set_request_uri(builder.to_string());
      request.headers().add(header_names::accept, L"application/json");

      // Start the request
      return client.request(request)
      .then([](http_response response) {
            return response.extract_json();
      });
}

int main() {
    pplx::task<json::value> task;
    task = get_JSON_google_API();
    json::value json_value = task.get();
    cout << json_value << endl;
    return 0;
}
```

Notez que vous devez indiquer au compilateur le répertoire où les fichiers d'en-têtes et les fichiers binaires d'une bibliothèque externe sont stockés (voir `g++ man`, pour les options `-I`, `-L`, et `-l`). Dans le cas de la bibliothèque C++ REST SDK, pour la compilation et l'édition des liens pour le fichier ci-dessus, vous devez utiliser la commande suivante :

```
g++ http-json.cpp -std=c++11 -I [repertoire Release/include/] -L [repertoire Release/build.release/
     Binaries] -lcpprest
```

Vous devez également configurer la variable d'environnement `LD_LIBRARY_PATH` :

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:[repertoire Release/build.release/Binaries]
```

La documentation sur cette bibliothèque (y compris l'interface des classes pour la manipulation d'une réponse en format JSON) est également disponible [9].

8. https://casablanca.codeplex.com/wikipage?title=Http%20Client%20Tutorial
9. http://microsoft.github.io/cpprestsdk/