Rapport projet

2 – Creating Graph Classes

2.1 Graph Classes

```
class Graph {
protected:
    std::map<int, Vertex> vertList;
    std::vector<Edge> edgeList;

public:
    Graph(const std::string fileName);

    void neighbors(const Vertex v, std::vector<Vertex> &list);
    void bfs(uint32_t vstart, uint32_t vstop);
};
```

Notre classe **Graph** est composée de deux variables membres, la vertList qui recense toutes les intersections de routes et la edgeList qui représente tous les chemins entre chaque intersection :

- La vertList est un dictionnaire avec des indices de type int et des valeurs de type classe Vertex. La classe **Vertex** correspond à une « vertice », avec comme variables membres longitude et latitude qui représente les coordonnées de l'intersection et l'id comme identifiant.

```
class Vertex{

protected:
    uint32_t id;
    double longitude;
    double latitude;
```

Vertex possède plusieurs méthodes :

  - Les méthodes **get**, pour chaque variable.

```
uint32_t getId(){ return id; }
double getLong() { return longitude; }// méthodes pour récupérer les valeurs des différentes variables membres
double getLat() { return latitude; }
```

  - Les surcharges d'opérateurs « == » et « = ».

```
friend bool operator==(Vertex v1, Vertex v2){ // surcharge de l'opérateur ==, pour vérifier si deux Vertex sont égaux

    return (v1.getId() == v2.getId() && v1.getLat() == v2.getLat() && v1.getLong() == v2.getLong());
}

Vertex &operator=(Vertex &v){//surcharde de l'opérateur =, pour affecter de nouvelles valeurs à un vertex

    id = v.getId();
    longitude = v.longitude;
    latitude = v.latitude;

    return *this;
}
```

- La edgeList est un tableau dynamique de vecteurs de type classe **Edge**, chaque Edge représente « a road segment between two intersections ».

  La classe **Edge** comprend 3 variables membres : 2 int qui correspond aux identifiants de 2 vertex et d'un double dist étant la distance entre ces deux Vertex.

```
class Edge {

protected:
    int v1Id;
    int v2Id;
    double dist;
```

  Edge possède plusieurs méthodes :

  - Les méthodes pour chaque variable.

```
uint32_t getV1Id(){ return v1Id; }
uint32_t getV2Id(){ return v2Id; }// mêmes fonctions que pour la classe Vertex
double getDist(){ return dist; }
```

  - Une surcharge de l'opérateur « == ».

```
friend bool operator==(Edge e1, Edge e2){

    return(e1.getV1Id() == e2.getV1Id() && e1.getV2Id() == e2.getV2Id() && e1.getDist() == e2.getDist());// mêmes fonctions
}
```

  Ces deux classes possèdent évidemment un constructeur mais peu d'intérêt de le montrer.

2.2 Reading a Graph Map File

La classe **Graph** contient deux méthodes et un constructeur, seul le constructeur sera expliqué pour cette partie :

Le constructeur **Graph :**

On déclare un ifstream et on vérifie qu'il est bien ouvert.

```
ifstream fin(fileName, ios::in);
if(!fin.is_open()){
    perror("File can't be opened");
}
```

On parcourt le fichier et on stocke les 4 données dans des string en prenant en compte qu'il sont séparés par une virgule :

```
stringstream ss(line);
            string c1, c2, c3, c4;

            getline(ss, c1,',');
            getline(ss, c2,','); // on récupère les 4 données par ligne séparées par une virgule
            getline(ss, c3,',');
            getline(ss, c4,',');
```

On prend en compte les deux cas puis on convertit les coordonnées en int et double :

```
if(c1 == "V"){
                Vertex v(stoi(c2), stod(c3), stod(c4));
                vertList.insert({stoi(c2), v});
                v_counter ++;
            }                                           // on sépare les deux cas quand il s'agit d'un Vertex ou d'un Edge
            else if(c1 == "E"){
                Edge e(stoi(c2), stoi(c3), stod(c4));   // on convertit les coordonnées en int et en double
                edgeList.push_back(e);
                e_counter ++;
            }
        }
```

De plus, on ajoute chaque edge ou vertex à sa liste correspondante et on incrémente un compteur à chaque fois qu'on en ajoute un dans une liste.

On affiche les deux compteurs et on ferme le fichier.

```
    cout << "Graph created from " << fileName << " : " << v_counter << " vertex and " << e_counter << " edges added." << endl;
}
fin.close();
```

3 – Breadth First Search

3.1 Introduction

Présentation des méthodes utilisées :

- **Neighboors** : cette méthode prend en paramètre un Vertex et un tableau de Vertex (qui va être remplis de tous les Vertex proches de celui mis en paramètre) et elle identifie les voisins atteignables en parcourant la edgeList.
  Si l'id du Vertex et celui de l'edge sont les mêmes alors on ajoute le deuxième identifiant de l'edge dans la liste en paramètre, puisque cela veut dire qu'il existe une route entre ces deux intersections.

```
void Graph::neighbors(Vertex v, vector<Vertex> &list){

    for(Edge e : edgeList){

        if(e.getV1Id() == v.getId()){
            auto it = vertList.find(e.getV2Id()); //Remplit la liste avec les voisins ATTEIGNABLES depuis un certain Vertex
            list.push_back(it->second);
        }
    }
}
```

- **Bfs** : cette méthode reprend donc l'algorithme, tout d'abord on déclare les conteneurs et les variables que l'on va utiliser.

```
//Conteneurs
deque<Vertex> active_queue;
vector<Vertex> closed_set;
vector<Vertex> visited;
map<uint32_t, pair<uint32_t, double>> next_before; //Map dans laquelle sera stockee les vertex, leur precedent direct et la distance entre eux

//Variables
Vertex vcurrent;
int visited_vertices = 0;
bool stop_flag = false;
double dist;
```

On commence par récupérer les vertex grâce à leurs identifiants.

```
//On recupere les deux vecteurs a partir de leur identifiant
auto it_va = vertList.find(vstart);
auto it_vb = vertList.find(vstop);
Vertex v1 = it_va->second;
Vertex v2 = it_vb->second;

active_queue.push_back(v1);
```

On incrémente le compteur de vertices visitées tant que active_queue n'est pas encore vide. Dès que le drapeau est up on stoppe la boucle.

```
while(active_queue.size() != 0){

    visited_vertices ++;

    vcurrent = active_queue.front();
    active_queue.pop_front();
    closed_set.push_back(vcurrent);

    if(stop_flag == true){ //On arrete la boucle si on trouve vstop
        cout << "A path does exist." << endl;
        break;
    }
```

On calcule les vertices atteignables par rapport àla vertice actuelle.

```
vector<Vertex> adjacency_list; //Liste des voisins directs de vcurrent
neighbors(vcurrent, adjacency_list);
```

Pour chaque vertex atteignable, on calcule les distances en récupérant l'Edge correspondant puis on ajoute le vertice à la liste de ceux visitées.

```
for(Vertex vnext : adjacency_list){

    //On recupere la distance en fonction des deux vertex
    for(Edge e : edgeList){
        if(e.getV1Id() == vcurrent.getId() && e.getV2Id() == vnext.getId()) dist = e.getDist();
    }

    next_before.insert({vnext.getId(), {vcurrent.getId(), dist}});
    visited.push_back(vnext);
```

On recherche le vertice dans le closed_set et dans l' active_queue, si la vertice n'est pas présent dans active_queue on l'ajoute et on change le statut du flag.

```
    //Pour debugger -->
    //cout << "next : " << vnext.getId() << " prev : " << vcurrent.getId() << endl;

    auto it_cs = find(closed_set.begin(), closed_set.end(), vnext); //Recherche de vnext dans closed_set
    auto it_aq = find(active_queue.begin(), active_queue.end(), vnext); //Recherche de vnext dans active_queue

    if(it_cs != closed_set.end()); //Si vnext est présent dans closed_set on continue
    else if(it_aq == active_queue.end()){ //Si vnext n'est pas présent dans active_queue on l'ajoute
        if(vnext == v2){ //Condition d'arret
            stop_flag = true;
            visited.push_back(vnext);
            break;
        }
        active_queue.push_back(vnext);
    }
```

S'il n'y a aucun chemin, on l'affiche sur le terminal.

```
//Si le chemin entre les deux vertex n'existe pas
if(active_queue.size() == 0){
    cout << "There is no path between these two vertices." << endl;
}
```

Remplissage des vectors path et distances par rapport aux résultats trouvés.

```
//Remplissage liste du chemin et des distances
double total_dist = 0;
vector<uint32_t> path = {visited.back().getId()};
vector<double> distances = {0};
auto it_nb = next_before.find(visited.back().getId());
uint32_t previous = it_nb->second.first;

while(previous != v1.getId()){ //On remplit le chemin en partant de la fin
    total_dist += it_nb->second.second;
    path.push_back(previous);
    distances.push_back(total_dist);

    it_nb = next_before.find(previous);
    previous = it_nb->second.first;
}

total_dist += it_nb->second.second;
path.push_back(v1.getId());
distances.push_back(total_dist);
```

On affiche les résultats :

```
//Affichage :
cout << "Total visited vertex : " << visited_vertices << endl;//nb de vertex parcourues
cout << "Total vertex on path from start to end : " << path.size() << endl;//nb de vertex sur le chemin

for(uint32_t i = 0; i<path.size(); i++){

    cout << "Vertex[";
    printf("%3d", i+1);
    cout << "] = ";
    printf("%6d", path[path.size()-i-1]);
    cout << ", length = ";
    printf("%9.2f", distances[i]);
    cout << endl;
}
```

On réalise deux essais :

⇒ 19791 à 50179

```
Total visited vertex : 14349
Total vertex on path from start to end : 66
```

```
Vertex[ 66] = 272851, length =  10346.38
```

⇒ 73964 à 272851

```
Total visited vertex : 19562
Total vertex on path from start to end : 69
```

```
Vertex[ 69] =  50179, length =  13080.25
```

4 – Dijkstra Shortest Path

4.1 Introduction

On définit dans cette partie le poids d'un sommet qui n'est rien d'autre que la distance séparant deux vertex. On retrouve cette donnée en 3ième paramètre des Edges dans le fichier data_area

L'algorithme précédent nous permet de trouver un chemin d'un vertex A à un vertex B mais ce chemin n'est pas optimal. Dans cette partie, nous allons utiliser l'algorithme de Dijkstra qui va nous permettre d'obtenir un chemin plus court. Cet algorithme repose sur le classement et le calcul de la distance minimale entre les sommets et le sommet initial.

4.2 Programmation

On réutilise en grande partie la méthode BFS qu'on a écrite et expliqué précédemment.

```
    if(it_aq == active_queue.end()){ //Si vnext n'est pas présent dans active_queue on l'ajoute
        vnext.setW(weight);
        active_queue.push_back(vnext);
        next_before.insert({vnext.getId(), {vcurrent.getId(), dist}});
    }
    else if(weight < vnext.getW()){
        vnext.setW(weight);
    }
}
sort(active_queue.begin(), active_queue.end(), isLighter);
```

La différence réside dans le fait que les vecteurs prennent un poids comme expliqué précédemment. On prend à chaque fois le poids le plus court.

4.3 Essai

On test notre programme avec les vertex 73964 et 272851 et on obtient le résultat suivant :

```
Total visited vertex : 14732
Total vertex on path from start to end : 81
Vertex[ 81] = 272851, length =   7793.30
```

Conformément à nos attentes, on obtient un meilleur résultat que pour la fonction BFS.

5 – A-Star

4.1 Introduction

Dans cette partie, nous allons encore une fois améliorer le déplacement d'un vertex à un autre. Pour cela nous allons utiliser l'algorithme A qui est un algorithme de recherche de chemin plus performant que celui de Dijkstra (moins de vertex à parcourir).

Cet algorithme est plus simple dans sa programmation et permet l'obtention d'un résultat très rapidement. A chaque itération), on va tenter de se rapprocher de la destination, on va donc privilégier les possibilités directement plus proches de la destination, en mettant de côté toutes les autres.

4.2 Programmation

Ici on utilise la fonction dijkstra sur tous les vertices proches pour déduire le plus proche.

```
double hdEstimator(Vertex v1, Vertex v2){
    double l = v1.getLong()*pi/180;
    double lc = v2.getLong()*pi/180;
    double p = v1.getLat()*pi/180;
    double pc = v2.getLat()*pi/180;
    double r = 6378137;

    double x = r*cos(pc)*(l - lc);
    double y = r*log(tan(((p-pc)/2)+pi/4));

    return sqrt(pow(x,2) + pow(y,2));
}
```

On programme une fonction hdEstimator (heuristic distance estimator) qui permet de convertir les longitudes et latitudes de chaque vecteur en coordonnées cartésiennes et on retourne la distance qui les sépare.

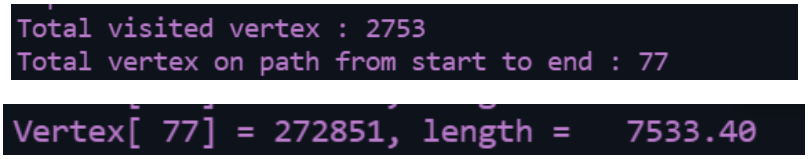
```
    g = vcurrent.getW() + dist;
    f = g + hdEstimator(vnext, v2);

    if(it_aq == active_queue.end()){ //Si vnext n'est pas présent dans active_queue on l'ajoute
        vnext.setW(g);
        vnext.setE(f);
        active_queue.push_back(vnext);
        next_before.insert({vnext.getId(), {vcurrent.getId(), dist}});
    }
    else if(f < vnext.getE()){
        vnext.setW(g);
        vnext.setE(f);
    }
}
sort(active_queue.begin(), active_queue.end(), estimateSort);
```

Dans l'algorithme astar, on reprend la aussi le code précédent. On se sert de la fonction hdEstimator pour trouver notre fonction F.

4.3 Essai

On test notre programme avec les vertex 73964 et 272851 et on obtient le résultat suivant :



Là aussi, on observe une longueur moindre comparé à l'algorithme précédent et une diminution du nombre de vertex visités.