Open Mapping

BHATI Vikram - BOUZIDI Imane
ISSAOUI Ali - LEBDAOUI SELMANE

Encadré par:

S. ARCHIPOFF
D. RENAULT

Avril 2019

Table des matières

1 Introduction

1.1 Présentation du sujet

Dans ce projet intitulé Open Mapping Service, nous étions amenés à manipuler les graphes afin d'exploiter des données cartographiques. Dans un premier temps, notre tâche consistait à construire un graphe à partir des données au format XML (fournis par ...) , dans un second temps, il s'agissait d'implémenter un ensemble d'algorithmes afin d'exploiter ces graphes : calculer les distances entre des noeuds, extraire des itinéraires et des cycles, etc... Finalement, nous avons implémenter un serveur HTTP pour afficher les résultats en utilisant le conteneur SVG. Ce rapport a pour but de décrire le déroulement du projet.

Dans un premier temps, nous allons présenter comment nous avons géré le projet. Nous avons choisi de répartir les tâches entre les différents membres du groupe. Cette première partie serait également l'occasion pour nous de justifier notre choix de certaines structures et du Hash table que nous avions jugés aptes à répondre aux consignes du sujet et adaptés à nos besoins.

Dans un second temps nous présenterons, d'une manière spécifique, les différentes parties du projet telles que nous les avons imaginées et conçues. Nous décrirons le fonctionnement de chaque algorithme implémenté. Nous saisirons l'aubaine, dans cette meme optique « technique », pour faire part de nos impressions sur le projet et les difficultés techniques rencontrées ainsi que les voies de recherches ouvertes.

Comme dans tout projet de programmation, il était nécessaire que nous nous assurions que notre code fonctionne bien et répond aux exigences du sujet, sans erreur, et produit ce pour quoi il a été construit. Pour cette raison, nous avons conçus des tests dont la principale fonction était de mettre sous la loupe chaque travail fait et de s'assurer que ce dernier répond aux consignes.

Enfin, la dernière partie serait consacrée à mettre en exergue les perspectives ouvertes, ce que nous n'avons pas pu faire, les limites des solutions que nous avons proposées.

Nous espérons que vous prendrez autant de plaisir à lire ce rapport que nous en avons pris durant le déroulement de ce projet.

2 Organisation du projet

2.1 Gestion du travail et répartition des taches

Pour un bon déroulement du projet, il était nécessaire que nous déterminions les différentes tâches et les répartir entre nous. Pour cela, et après avoir pris connaissance du sujet, nous avons défini les solutions aptes à répondre aux consignes pour, finalement, déterminé les tâches à accomplir.
Les deux principaux enjeux que nous avons cernés pour ce projet sont:

1. Manipuler les données au format XML .
2. Convertir ces données en des graphes exploitables.

Les principales tâches qui ont aiguillé notre projet sont les suivantes :

- Chargement des fichiers au format osm et leur transcription en un format de graphe adéquat.
- Choix des structures du graphe et des sommets
- Pondération du graphe
- Exploitation du graphe (1) : calcul d'un itinéraire entre deux sommets, extraction du plus court chemin
- Exploitation du graphe (2) : calcul du cycle de longueur minimale (problème du voyageur de commerce)
- Affichage des maps dans un serveur HTTP à l'aide du conteneur SVG.

Nous avons travaillé en autonomie sur ses tâches en partageant les solutions utilisées et expliquant clairement chaque modification effectuée. Ainsi, chaque vendredi après-midi a été l'occasion pour nous de mettre le point sur nos avancements et tenir compte Mr ARCHIPOFF de notre travail.

2.2 Le choix des structures

Pour représenter les sommets, on a choisi la structure #**vertex** où on a l'identifiant du sommet noté *id*, la latitude du sommet notée *lat* , la longitude du sommet notée *lon* et la liste des voisins notée *way*

$$\{id_1 : lon : lat : way\}$$

Pour la représentation du graph , on a définit une structure #**graph** qui contient une table de hachage qui à chaque clé : *id* associe un #**vertex** . L'id contenu dans ce vertex doit etre le même que l'id se trouvant dans la table de hachage,

$$(\{id_1 : V_1\},\{id_2 : V_2\},...,\{id_n : V_n\}) \text{ avec } (vertex - id\ V_n) = id_n$$

2.3 Le choix du Hash table

Dans le projet, le choix de la structure de graph se prêtait naturellement à l'implémentation par liste d'adjacence. Toutefois, cela a un cout en temps assez

élevé: au pire l'on accède à une donnée en $O(n)$ (n étant de nombre de noeuds du graphe), au mieux en $O(\log_2(n))$ (par dichotomie). Dans tous les cas, cette approche devient très handicapante pour un nombre important de noeuds. Et c'est exactement le cas pour les fichiers osm que l'on traite. C'est pourquoi, il nous a semblé plus judicieux d'utiliser une table de hachage dont l'accès et la mise à jour des noeuds se fait en $O(\log(N))$ (où N est une constante) [1].

3 Explication des algorithmes et des implémentations choisies

3.1 Le graphe

La fonction *create-vertex()* sert à créer à partir d'un identifiant *id*, des coordonnées *lat* et *lon* et la liste des voisins *way* un sommet. La fonction **add-vertex()** sert à ajouter un sommet à un graph. A noter que chaque sommet dans le graph dispose d'un id unique, et qu'il reste beaucoup de fonctions basiques de traitement du graphe spécifiques à chaque fichier source.

3.2 Passage de l'Osm au graph

Cette partie explique le processus permettant de passer d'un fichier osm à un graphe. Tout d'abord, l'importation du fichier osm sous Racket conserve certaines normes de paranthésages qui nous ont semblé obsolètes pour les raisons suivantes: notre but étant d'extraire les noeuds sous forme de vecteur **(identifiant longitude lattitude liste:(voisin 1 voisin 2 ...))**, on a remarqué que les trois premières informations étaient toujours précédées du mot-clé "node" quand à la dernière information, elle était précédée de "way". Ainsi, il suffisait d'abord d'effecteur un $flatten$ sur le fichier osm importé, suivi d'un *member* pour trouver les mot-clés et ensuite récupérer les informations voulues.

Dans notre implementation, on ne parcourt le fichier osm que 3 fois : la première pour $flatten$, la seconde pour créer la liste des noeuds avec les caractéristiques (identifiant longitude lattitude) et une troisième fois pour créer la liste des voisins comme liste de couple **(a b)** où a et b sont les identifiants des noeuds qui se suivent dans la section "way" du fichier osm. Bien que l'on aurait aimé regroupé ces deux derniers parcours (la liste des noeuds et la liste des voisins) en 1 seul, cela aurait impacté la simplicité et la lisibilité du code.
Enfin, l'étape finale consiste à croiser la liste des noeuds avec celle des voisins et à les insérer dans la table de hachage.

Une fois le graphe sous forme de table de hachage obtenu, il faut désormais le simplifier. Pour cela, on doit supprimer les noeuds de degrés exactement 2. En effet, si on considère trois noeuds **A, B** et **C**, tels que **B** est voisin avec

1. lien vers la documentation Racket du Hash Table mentionnant la complexité: *https* : $//docs.racket - lang.org/reference/hashtables.html$

uniquement **A** et **C** (**B** est alors de degré 2), et si l'on va de **A** vers **C**, alors il est inutil de spécifier que l'on passe par **B**. C'est la simplification qui a été choisie (imposée).

Il y a deux façon de traiter le problème: la première est de parcourir le graphe pour supprimer les noeuds dans une copie du graphe, et la seconde consiste à supprimer directement dans le graphe au fur et à mesure. Nous avons choisi la seconde méthode par considération du coût en espace et en temps qu'impliquerait une copie d'un fichier osm imposant. De plus, peu importe la méthode choisie, le problème reste fondamentalement le même, ainsi, l'on pourrait toujours apporter des modification mineures pour passer d'une implémentation à une autre.

3.3 Le serveur SVG

3.3.1 Manipulation basique

Pour une manipulation basique du graphe, on utilise une première famille de fonctions qui génèrent une liste des données du graph telles que **keys()** qui retourne la liste de tous les identifiants et **list-lat()** qui génère la liste des latitudes.
Le deuxième type de fonctions est celui pour lequel on extrait des sommets particuliers, par exemple, le sommet dont la latitude est maximale ou encore celui dont la latitude est minimale. A noter que dans l'entete du fichier donné OSM figurent les bornes inférieure et supérieure de la latitude et de la longitude, nous récupérons alors ces bornes à l'aide de la fonction **extremums()** .

3.3.2 Manipulation avancée (1) - Représentation d'un graph

On a d'abord songé à convertir les longitudes et les latitudes avant de procéder au dessin du graph pour une mise à l'échelle. On convertit par exemple la latitude *lat* de la manière suivante:

$$coefficient\_lat * \frac{minlat - lat}{lat - maxlat}$$

- La fonction **circle-ray()** place un cercle (un point) pour un sommet. Pour placer tous les sommets d'un graph on utilise la fonction **graph-circles()** .
- Pour relier deux vertex, on utilise **create-line()**. Pour relier tous les sommets d'un graph on utilise la fonction **graph-lines()**.
- Pour insérer du texte dans un point de coordonnées {x;y} on utilise la fonction **display-distance()**.
- On trace l'itinéraire entre un vertex et ses voisins à l'aide de la fonction **vertex-lines()**. Les sommets et les aretes sont tracés avec **graph-map()**.

3.3.3 Manipulation avancée (2) - Représentation d'un itinéraire

Etant donné un itinéraire entre une liste de sommets.

1. La fonction **itinerary-circles()** place chaque sommet de l'itinéraire.
2. La fonction **itinerary-lines()** trace les aretes entre chaque paire de sommets.
3. **dijkstra-map()** sert à tracer le chemin le plus court entre deux sommets. Enfin, la fonction **itinerary-map()** est la combinaison des deux fonctions précédentes c'est-à-dire elle place les sommets et les aretes à la fois.

3.3.4 Le serveur racket

Dans le fichier *server.rkt*, nous avons implémenté les différents fonctions qui servent à traiter les *requests* du language Racket, afin de pouvoir récupérer l'url éxecutée. En effet, nous avons fait quatres **rules** comme il est précisé dans l'énoncé, *display*, *route*, *distance* et *cycle*. La fonction *server-dispatch* permet de récupérer l'url à exécuter et vérifie si le mot directement après $http://localhost:9000/$ et avant ? correspond à l'un des *définis* précedemment, si c'est le cas, elle appelle la fonction correspondante.

3.4 Parcours du graph et itinéraire

Une première façon de chercher l'itinéraire entre deux noeuds est le parcours en profondeur. En effet, nous avons implémenté une fonction de parcours en profondeur, qui, étant donné un graphe et un noeud de départ parcourt la totalité des noeuds appartenant à la même composante connexe que le noeud de départ, suivant le trajet du parcours en pronfondeur. Commençant par une racine, on parcourt un premier fils puis tous ses fils récursivement, ensuite, on passe aux autres fils de la racine. Deux variables nous ont été utiles, une première variable nommée *Marked* qui marque chaque élement par lequel on est déjà passé pour ne pas le parcourir une deuxième fois. Et la variable *result* qui joue le rôle d'une pile enregistrant à chaque fois l'itinéraire entre la racine est le noeud en cours de traitement par la fonction. En effet, à chaque **appel récursif**, la liste empile le noeud en cours de traitement, mais une fois cet appel fini, la liste dépile ce noeud. De cette façon, dès que le noeud traité est celui qu'on recherche, la liste *result* contient le trajet voulu à partir de la racine, et est donc retournée. Par contre, cette fonction a besoin d'un parcours en profondeur déjà effectué sur le noeud de départ pour décider si le noeud d'arrivé est joignable ou pas.

3.5 La formule de Haversine

Puisque dans ce projet, nous recourons à la manipulation des graphes, il est évident que l'on ait besoin d'un outil de calcul des distances à partir des coordonnées des sommets. Conformément aux instructions de l'énoncé, nous avons

utilisé la formule de Haversine :

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

- λ_1, λ_2 : longitudes respectives des ommets 1 et 2
- φ_1, φ_2 : latitudes respectves des sommets 1 et 2
- d la distance de haversine
- r est pris, dans notre code , égal au rayon de la terre

Les résultats du calcul des distances avec cette formule ont été comparés aux valeurs données par l'outil qui nous a été livré en guise de ressource supplémentaire dans une page d'explication [2].

3.6 Dijkstra

Dans cette partie, notre objectif était d'implémenter un algorithme permettant de trouver le plus court chemin entre deux sommets. Notre choix s'est naturellement porté sur l'algorithme de Dijkstra de par la simplicité de son principe mais aussi parce qu'il a été l'objet d'un enseignement dispensé dans le module Algorithmique des graphes.

Quoique le pseudo-code soit connu, l'implémentation de Dijkstra en fonctionnelle reste une tache ardue. Néanmoins, une façon facile de procéder consiste à décomposer le code en des fonctions auxiliaires qui interviendront toutes, à la fin, dans une boucle nommée **dijkstra_loop()** dans notre code.

Le pseudo-code considéré est le suivant:

```
dijkstra(g,dis,sdep)
Initialisation(g,sdep)
Q:= ensemble de tous les noeuds
Tant que Q n'est pas un ensemble vide
Faire
      s1:= min(Q)
      Q := Q priv   de s1
      pour chaque noeud s2 voisin de s1 Faire
        maj_distances(s1,s2)
      fin pour
```

2. lien vers la page d'explication livrant un outil de calcul de distance à partir de la latitude et de la longitude: *https : //www.movable − type.co.uk/scripts/latlong.html*

```
fin tant que
```

Pour l'initialisation, la fonction **initial** prend en argument l'identifiant du sommet de départ, une liste des identifiants de tous les sommets du graph (y compris le sommet de départ) et une liste l *(cette liste l serait toujours la liste vide au 1er appel).*
A l'issue de cette fonction, nous avons une liste de paires où chaque paire est sous la forme **(id_sommet distance)** le champ distance est nul pour le sommet de départ et est égal à une valeur très grande (supérieure à toutes les distances considérées dans le problème) pour les autres sommets.

3.6.1 Une première version de Dijkstra

Dans cette première version nommée **old-dijkstra** , nous avons choisi de manipuler surtout les identifiants des sommets et non pas la structure vertex (toute entière) puisque la structure du graph en Hash table nous permettait de récupérer, facilement, un vertex à partir de l'identifiant uniquement. Les deux enjeux principaux dans cette implémentation étaient de :

1. Avoir une structure qui nous permet de sauvegarder les sommets différents du sommet de départ et la distance séparant chacun de ces sommets du sommet de départ. Nous avons choisi d'utiliser une liste de paires, par exemple : **((1 2.4) (2 3.01))** où le premier élément de la paire désigne l'identifiant du sommet et le deuxième représente la distance séparant ce sommet du sommet de départ (sdep) .
2. Mettre à jour à chaque itération cette liste de paires. La procedure majtabdis, prenant en argument une listedouble , une distance d et un sommet s met à jour la liste de telle sorte que la distance séparant le sommet de départ du sommet s devienne d.

La liste des paires est également la structure que nous avions privilégié pour sauvegarder les prédecesseurs/les voisins de chaque sommet.

3.6.2 La version fonctionnelle de Dijkstra

Dans cette seconde version fonctionnelle nommée dijkstra, nous nous sommes basés une deuxième fois sur les tables de hachages, en effet, notre fonction *initialize* génère à partir d'un graphe et un noeud de départ passés en paramètre une table de hachage dont l'élèment (id-v : (distance , predecessor)) où distance est la distance séparant le vertex dont l'id est id-v du vertex de départ, cette distance étant inialisée à 10000, et predecessor représente l'identifiant du noeud prédecesseur du noeud d'id-v dans la liste de Dijkstra. Ensuite, on a eu besoin de la fonction *find-min* prenant en paramètre "distance" qui est une table de hachage comme spécifié précedamment et Q qui est une liste de sommet. Cette fonction retrouve le noeud le plus proche du noeud de départ. Puis, la fonction *maj_ distances* met à jour les distances stockées dans la table de hachage à

chaque fois qu'elle trouve des noeuds plus proches. La fonction *maj_ vertex* est une fonction intermediaire qui met à jours les distances de tous ses voisins.
Enfin, la fonction *Dijkstra_ loop* représente la boucle du programme, elle prend en paramètres g un graphe, v un noeud de départ, *distances* une table de hachage contenant les distances initiales et Q contenant la liste des ids des noeuds appartenant à la même composante connexe que le noeud de départ [3]. Dans la boucle principale, on cherche à chaque fois le noeud plus proche, mis à jour les distances de ses voisins et l'enlève de la liste Q. Le résultat de ce traitement est une table de hachage contenant les ids des noeuds accessibles à partir du noeud de départ, Ensuite, pour trouver l'itinéraire il suffit de remonter les prédécesseurs du noeuds recherché dans la liste des distances jusqu'à ce qu'on trouve le noeuds de départ. Le résultat étant une liste d'élements (id_n, $distance_n$) avec distance étant la distance cummulée du noeud d'id id_n] au noeud de départ.

3.7 Voyageur de commerce

Le problème du voyageur de commerce est un problème classique d'optimisation, qui se résout efficamement à l'aide d'heuristiques comme celles des algorithmes génétiques ou des algorithmes de colonies de fourmis. Mais ces algorithmes sont plus adaptés à une implémentation itérative que fonctionelle.

Le but véritable de cette partie était de se familiariser au parcours de graphe en implémentant plusieurs stratégies de résolution du TSP (problème du voyageur de commerce). À commencer par la méthode des plus proches voisins, qui est une méthode naïve qui part d'un noeud arbitraire et construit un cycle en passant à chaque fois par le noeud le plus proche. La suivante est l'insertion minimale que l'on apellera greedy: on commence par une liste L de deux noeuds, et on ajoute les noeuds restants à L aux position où la distance est minimale. Une autre façon (peu commune) de résoudre le problème est de parcourir aléatoirement les sommets.

3. fournie par un parcours en profondeur

4 Complexité des algorithmes, temps d'éxecution et mémoire

4.1 Le passage de l'Osm au graph et la réduction des sommets de degré 2

Pour obtenir le graph, on doit passer par une série d'opérations sur le fichier osm: il faut d'abord le charger en mémoire, puis appliquer la procédure $flatten$. En notant K la longueur du fichier obtenu après avoir $flatten$ le fichier osm, obtenir le graph se fait en temps $O(K)$. En effet, on parcourt le fichier 3 fois au total: la première pour le flatten, la seconde pour obtenir la liste des sommets, et une troisème pour obtenir la liste des chemins (l'opération de stockage se faisant dans un hash table en $O(1)$).
Notons g le graph obtenu. L'opération de réduction supprime un noeud de degré deux, notons le **A** dans la liste de ses voisins de **B** et **C** ce qui est au pire, en temps $O(|S|)$ (dans le cas où B ou C est voisin avec tous les sommets dans S). Ensuite l'on doit supprimer A dans g, ce qui se fait en $O(1)$. Donc supprimmer un noeud est réalisé en $O(|S|)$. On peut facilement imaginer le cas où B et C sont voisins de tous les sommet, et où tous les autres sommets sont d'arité 2. Alors il y aurait $|S| - 2$ sommets à supprimer. Ainsi, la compléxité temporelle au pire est en $O(|S|^2)$.

4.2 Parcours du graph et les itinéraires

4.3 La formule de Haversine

Il s'agit ici de calculer une formule mathématique dont les paramètres sont donnés en temps constant grace au hash table. D'où une compléxité en $O(1)$.

4.4 Dijkstra

Pour obtenir la distance d'un sommet de départ à un sommet d'arrivée, on doit d'abord savoir s'ils sont dans un même sous-graphe connexe. C'est la misssion de la fonction depth-first de complexité temporelle $O(|V| + |F|)$.En somme, la complexité est quadratique. L'implémentation d'un structure de donnée comme une file de priorité par tas aurait pu permettre d'obtenir une complexité en $O((|F| + |V|) \times \log(|V|))$.

4.5 Voyageur de commerce

La fonction nearest_diistance est quadratique en temps d'exécution, mais linéaire en espace de stockage: en effet pour chaque sommet, elle parcourt la liste des sommets restants à ajouter dans la liste des sommets déjà parcourus. La fonction farthest_distance est de complexité spatiale et temporelle identiques.
La fonction rand_tsp, a la même complexité que la fonction $shuffle$ dans la documentation Racket.

Enfin, la fonction **greedy** a une complexité quadratique. En insérer chaque sommet non visité dans la liste des sommets visités requiert un temps $O(|S|)$ (S représente l'ensemble des sommets) comme il y a $|S|$ sommets, on a bien la complexité quadratique. A savoir qu'un appel de heversine est lancé pour insérer le sommet, mais haversine a une complexité en $O(1)$.

5 Le passage de l'OSM au graph

Ici, il ne s'agit pas vraiment de problèmes à proprement parler. Pour contextualiser: l'on a d'abord commencé par implémenter une liste d'adjacence, puis on a dû réimplémenté avec l'idée d'un Hash Table, abandonnant 70% du code précédent, mais avec l'avantage d'une meilleure conceptualisation et maîtrise du fichier osm, ce qui a permis de gagner du temps, que ce soit à l'écriture du code, ou du débuggage. De plus, ce qui a été vraiment difficile, ce sont les différents choix à faire, par exemple celui où l'on devait choisir entre copier le graphe, ou le modifier directement (vu dans la partie 3.2) Ainsi, une des difficultés est que l'on devait soigneusement prendre les décisions concernant le graphe car cela impacterait toute l'équipe.

5.1 Le parcours du graph et les itinéraires

Dans le cas de la résolution du TSP, les difficultés étaient le plus souvent de l'ordre de la programmation que conceptuelles. Généralement, un découpage en fonctions auxiliaires permet de s'en sortir car ces dernières sont plus basiques, donc plus simples à débugger, et ainsi, grace à la validité des fonctions auxiliaire, on peut prouver la validité de l'algorithme se servant de ces fonctions.

5.2 Dijkstra

Dans la première version **old-dijkstra** un des problèmes importants rencontrés est celui de la manipulation des listes des paires c'est la raison essentielle pour laquelle nous avons opté pour une version plus élaborée qui ,elle, utilisait des hash-tables. Ce choix nous a permis ,par exemple, d'accéder plus facilement à la distance séparant un sommet s du sommet de départ sdep en ayant seulement l'identifiant du sommet s.

6 Nos attentes

6.1 Un affichage interprojets

Comme bonus de notre part, nous avons essayé de trouver un point commun entre les deux projets du semèstre S6, finalement, on a trouvé une possiblité de faire un affichage du bitboard du jeu Gomoku, meilleur que celui sur les terminales linux. En effet, on a fait de sorte que le serveur du gomoku genère des fichiers au format .osm acceptés par les fonctions de traitement de graphes implémentées dans le projet de programmation fonctionnelle. Nous avons ainsi créé un graphe dans la forme d'un bitboard et des noeuds spéciaux dans le graphe représentant les différents pions joués. Pour que le serveur Racket puisse savoir à qui appartiennent les pions, nous avons fait des ids avec la formule "id = couleur10" . "row + col bitboard_size" [4] , où couleur est 0 si le pion apparatient au premier jouer et 0 sinon, ainsi le serveur peut juste vérifier si l'id en numérique est suppérieur à 500 par exemple ou pas, pour savoir à qui il appartient. Notre résultat final était de la forme suivante. Remarque: On a pas commité le source

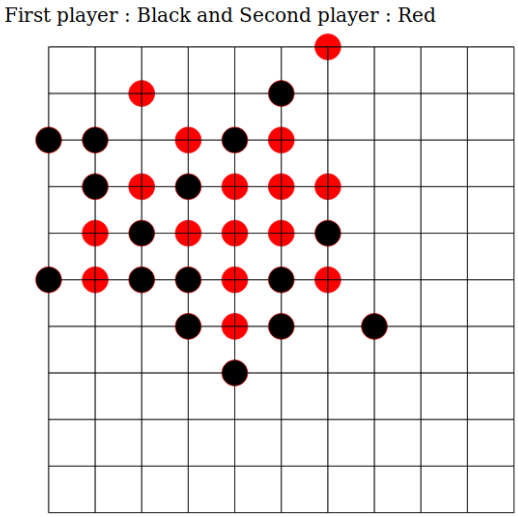


FIG. 1

correspondant à cet affichage vu des raisons de simpliification, et qu'il n'était pas directement demandé par l'énoncé des sujets.

6.2 Un serveur fonctionnant avec plusieurs joueurs

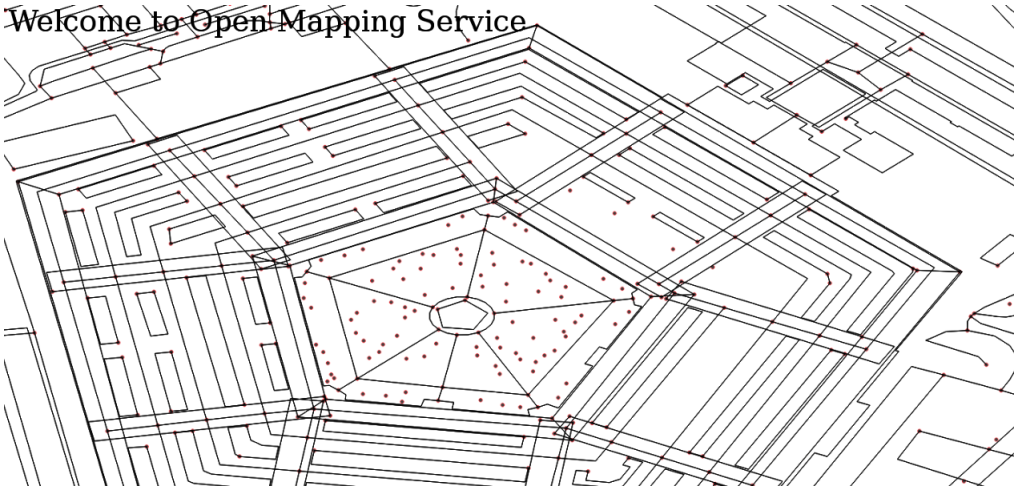
Une autre chose à laquelle on a pensé est de créer un serveur qui marche avec plusieurs joueurs, mais vu la contrainte du temps, nous n'avons pas réussi

4. . repésente la concaténation des chaines de caractères

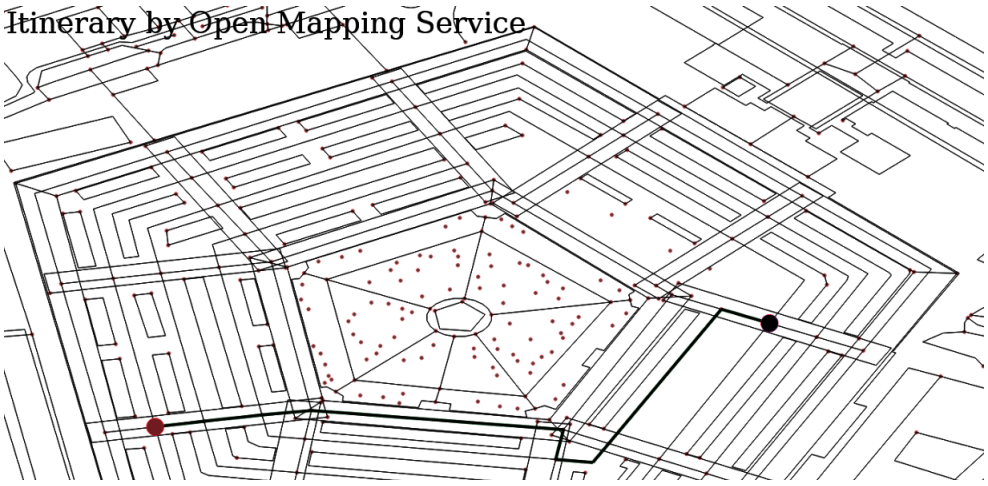
à la faire mais notre serveur peut plus ou moins facilement être adaptable à plusieurs joueurs.

7 Résultats finaux

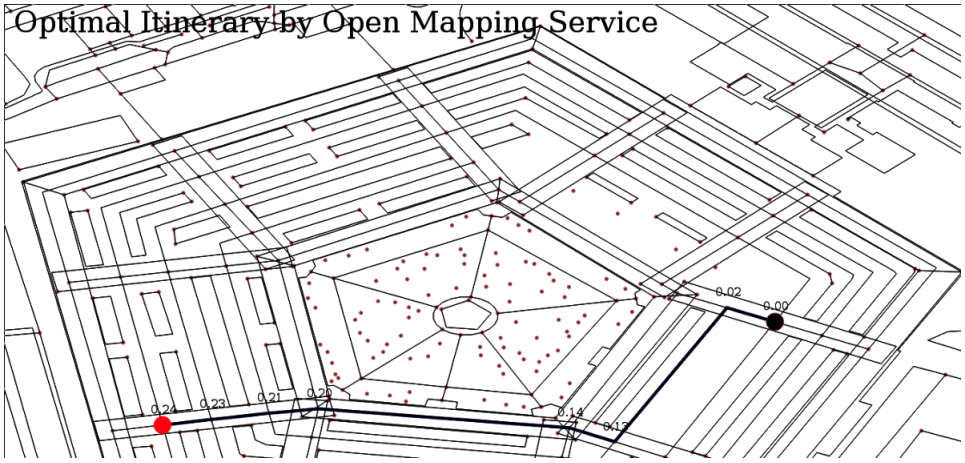
L'image ci-dessous est la représentation d'un graph. Les cercles en rouge désignent les sommets, les lignes en noir représentent les aretes reliant les sommets. cette map est obtenue à l'aide du lien : $http : //localhost : 9000/$



Dans cette seconde image on représente les sommets et les aretes en plus du chemin reliant deux sommets. Le choix de couleurs différentes est tout à fait possible. cette map est obtenue à l'aide du lien : $http : //localhost : 9000/route?start = 392014874\&end = 392015296$



Dans cette troisième image, on représente les sommets et les arretes et le chemin le plus court reliant deux sommets choisis aléatoirement. Le cercle en bleu désigne le sommet de départ et le cercle en rouge désigne le sommet d'arrivée. Vous pouvez constater qu'au dessus de chaque *sommet/cercle* figure la distance (en km) séparant ce sommet du sommet de départ. cette map est obtenue à l'aide du lien : $http : //localhost : 9000/distance?start = 392014874\&end = 392015296$



8 Conclusion

Dans ce projet notre premier défi était de passer des fichiers au format osm au format graph. Ensuite, s'est posée la question de quelles structures choisir pour le graph et pour les sommets. En plus de ces contraintes techniques, il a fallu songé à comment est-ce qu'on peut pondérer notre graph d'où l'usage de *la formule de haversine*, comment trouver l'itinéraire entre deux sommets d'où *le parcours en profondeur* et ,enfin, comment trouver le plus court chemin entre deux sommets d'où l'implémentation de *l'algorithme de dijkstra*. Une première implémentation de dijkstra a nécessité l'usage de liste des paires. L'idée de la mise à jour des distances et des prédecesseurs étant centrale dans dijkstra , il a fallu que l'on opte pour une seconde version de dijkstra qui , elle, utilise les hash-table. Une fois la manipulation des graphes finie, il a fallu que l'on travaille sur comment représenter les graphes, pour cela on a utilisé des fonctions permettant de placer des sommets, de tracer des aretes, de marquer les distances correspondant à chaque sommet mais aussi qui permettent de tracer à partir de l'algorithme de dijkstra le plus court chemin séparant deux sommets. Enfin, l'outil principal qui nous a permis de manipuler le graph est les tables de hachage.