ENSEIRB-MATMECA

Projet S6

Filière Informatique

Open Mapping Service

Auteurs :
Marc Duclusaud
Sonia Ouedraogo
Pierre Pavia
Lucas Trocherie

Superviseur :
M. Georges Eyrolles

24 août 2020

Table des matières

Introduction

Dans le cadre de ce projet, nous mettons en place un ensemble d'algorithmes permettant de manipuler des données cartographiques, et de fournir un service web pour afficher les résultats. Les données initiales que nous manipulons sont au format OSM (OpenStreetMap), dérivé du language *xml*. L'objectif final est d'en extraire un format de graphe permettant différents calculs notamment de distance puis de réaliser un server web affichant la map et différents services.

Ce rapport reprend en détail le travail qui a été réalisé tout en expliquant les choix d'implémentation qui ont été faits et la conception algorithmique. Nous analysons également les problèmes rencontrés et discutons de la complexité et de la correction de nos algorithmes.

Cadre de travail

La réalisation de ce projet s'est étalée sur huit semaines à raison de quatre heures par semaine sous la tutelle d'un encadrant auquel il faut ajouter le temps de pratique libre non encadrée. Pendant les séances nous avons cherché à subdiviser le travail et ainsi réaliser des fonctions auxiliaires sur lesquelles nous nous sommes appuyées pour le projet global.

Ce projet est réalisé entièrement en langage Scheme et en utilisant l'IDE drracket. Les programmes ont été implémentés en s'efforçant de respecter une des principales spécificités du langage fonctionnel qu'est la récursivité. Nous avions aussi à notre disposition une plateforme en ligne (git) où nous pouvions mettre en commun le travail de l'équipe. Par une série de tests, la forge git nous permettait aussi de voir où nous en étions et si nous validions au fur et à mesure le travail demandé.

1 Problématiques et difficultés

1.1 Problématique du projet

La réalisation de ce projet se divise en trois parties principales : il fallait d'une part traduire le fichier osm en un format de graphe et d'autre part extraire du graphe résultant des calculs de distance, de plus court chemin et de cycle. Il fallait enfin réaliser un server http afin de visualiser ces résultats.

1.2 Difficultés rencontrées

Etant donné que la grande majorité du projet consistait à manipuler le type de données abstraites que sont les *graphes*, il était important de définir au préalable cette structure de données. Une des principales difficultés a été justement de déterminer comment représenter notre graphe. Il fallait trouver la structure adaptée dans l'optique de réduire par la suite la complexité de nos calculs.

2 Solutions apportées

Les contraintes évoquées dans le paragraphe précédent nous ont donc menés à faire des choix d'implémentations que nous détaillerons dans cette partie, puis nous analyserons la conception algorithmique des différentes fonctions.

2.1 Choix du Type Abstrait de Données

2.1.1 Spécification du type graphe

Nous avons choisi une liste pour répresenter un noeud. Cette liste contient trois éléments relatifs au noeud :

— son identifiant
— ses coordonnées
— une liste contenant les identifiants de ses noeuds voisins

Cela a conduit à créer deux types intermédiaires. En effet, l'identifiant devait être un entier positif et les coordonées une liste de réels. Nous testons la spécification du noeud par le code 1 :

Listing 1 – Fonction de vérification de la nature d'un noeud

```
1 (define (vertex? v)
2   (and (positive? (car v))
3        (coord? (cadr v))
4        (neighbours? (caddr v))))
```

Une fois le noeud spécifié, le format graphe que nous avons choisi n'est autre qu'une liste de noeuds. Nous testons récursivement le type graphe par le code 2 :

Listing 2 – Fonction de vérification de la nature d'un graphe

```
1 (define (graph? g)
2   (or (null? g)
3       (and (vertex? (car g))
4            (graph? (cdr g)))))
```

2.1.2 Intérêt de ce choix

L'avantage principal de ce format de graphe est que, connaissant un nœud, nous pouvions avoir accès en temps constant à ses successeurs. En effet nous avions pensé dans un premier temps à réaliser un graphe sous la forme d'une liste contenant la liste des identifiants des nœuds et la liste des arcs. Cette spécification a ses avantages mais le principal inconvénient que nous avons rencontré est qu'elle augmentait la complexité des calculs qui suivraient, notamment la supression des nœuds de degré 2. Cette action nécessitait avant de supprimer un nœud , de modifier les successeurs de ses voisins. Ainsi, avoir en temps constant accès à chacun des voisins du noeud considéré n'est point négligeable. Par ailleurs cette structure de noeud permet d'accéder directement au degré du nœud par la longueur de la liste de ses voisins. Un deuxième avantage intervient lors de la recherche d'itinéraire puisqu'on accède directement aux différents itinéraires possibles à partir d'un nœud grâçe à la liste de ses voisins.

2.2 Prise en main du fichier OSM

Nous avons dû nous adapter pour lire les fichiers OSM et leur structure, qui n'étaient pas toujours facile à faire lire efficacement par un algorithme à cause du nombre de cas différents de structures de fichier faisant parfois apparaître des cas non considérés.
Dans un fichier OSM, les nœuds sont d'abord listés en début de fichier, avec chacun deux coordonnées, latitude et longitude, leur identifiant, et d'autres informations éventuelles qui ne nous intéressent pas dans le cadre de ce projet. Ensuite, les arcs reliant ces points sont listés en tant que "ways", avec une liste de noeuds et une série de tags. Les arcs qui nous intéressent sont ceux représentant des routes, avec le tag "highway".
Il nous a donc fallu d'abord importer les nœuds avec leur identifiant et leurs coordonnées en les extrayant du fichier OSM, puis importer chaque way ayant le tag "highway", c'est-à-dire ajouter en successeur de chaque nœud les nœuds qui lui sont adjacents dans la way.
Pour réussir à contruire un graphe sous notre format de données, nous avons implémenté des fonctions de recherche de tag pour les ways, les noeuds et les frontières d'une map.

2.3 Réduction du graphe

2.3.1 Suppression des sommets isolés

Dans notre simplification du graphe complexe obtenu via OpenMapping, nous enlevons tous les tracés ne relevant pas du réseau routier, comme par exemple les bâtiments, les frontières ...Les arètes correspondant à ces tracés ne sont pas enregistrées, et certains de nos nœuds se retrouvent donc complètement isolés, présents dans la base de données, mais reliés à aucun autre nœud. Nous associons donc un entier à chaque nœud, qui correspond au nombre de nœuds auxquels il est connecté : son degré.

Pour résoudre ce problème des noeuds isolés, nous supprimons donc les nœuds de degré zéro. La suppression d'un nœud de degré zéro n'a aucun impact sur le reste du graphe car la liste de leurs voisins est vide. Nous estimons alors que supprimer un nœud de degré zéro se fait en temps constant.

2.3.2 Supression des sommets de dégré 2

Principe

Contrairement à la suppression d'un nœud de degré zéro, celle d'un nœud de degré 2 est un peu plus complexe car elle impacte le reste du graphe. Illustrons cela par l'exemple suivant :

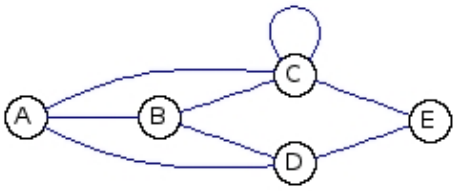

FIGURE 1 – Exemple de graphe illustratif

Supposons que sur la figure 1, nous voulions supprimer le noeud E qui est de dégré 2. Avant la suppression on constate qu'il est possible d'aller de C à D en passant par E. Si l'on supprime naïvement le noeud E, on supprimerait l'accès de C à D. Il fallait donc faire attention à bien relier les noeuds C et D avant la suppression, comme illustré sur la figure 2.

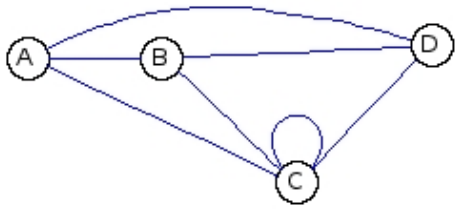

FIGURE 2 – Après suppression correcte du noeud E

Implémentation

Ainsi pour supprimer un nœud ayant des successeurs, il faut d'abord relier certains voisins entre eux. Dans l'implémentation algorithmique, cela se traduit par la fonction *manage_ node*. Cette fonction prend en paramètre deux nœuds voisins n_1 et n_2 avec au moins n_1 de degré 2. On va alors supprimer n_1 des voisins de n_2 et le remplacer par l'autre voisin de n_1. Il suffit alors de refaire cette action avec n_1 et le second voisin de n_1 (différent de n_2). Cette opération de *concaténation* se fait en temps constant avec ces deux nœuds en paramètre.

Une fois ce travail effectué, il est alors possible de supprimer le nœud de degré 2 de la manière suivante : étant donné le graphe, nous recupérons dans une liste tous les nœuds de degré 2, puis nous les traitons un a un jusqu'à ce que cette liste soit vide. Une des principales difficultés avec cette méthode est qu'après la concaténation un autre noeud de degré 2 contenu dans le graphe pourrait être modifié. Il fallait donc veiller à bien modifier ce nœud dans la liste des nœuds de degré 2 pour garder la cohérence.

Etude de complexité

Finalement la suppression des nœuds de degré 2 suit le principe suivant : prendre un nœud de degré 2, récupérer ces voisins en temps linéaire, effectuer la concaténation en temps constant, et supprimer le nœud en temps constant. Si **n** est le nombre de nœuds du graphe, l'algorithme est donc en complexité $\mathcal{O}(n^2)$ dans le pire des cas. On pourrait simplifier cette complexité en $\mathcal{O}(nlog(n))$ si l'on effectuait la recherche en dichotomie. Mais cela aurait impliqué de faire de la programmation impérative ce qui n'est pas le propre du langage Scheme.

2.4 Utilisation du graphe

2.4.1 Calcul d'itinéraire

Au début du projet, pour trouver naïvement un chemin entre deux points donnés nous avons effectué une recherche en profondeur : nous listions l'ensemble des chemins possibles reliant nos nœuds de départ et d'arrivée grâçe à `all_ways`.

L'intérêt d'un tel algorithme était qu'il permettait d'obtenir assez rapidement un itinéraire valide quelconque en retournant le premier qu'il déterminait et en ne calculant pas les suivants (heuristique). Cependant assez vite cette implémentation s'est retrouvée confronté à des limites : dans le cas de la recherche d'un itinéraire particulier (du meilleur itinéraire ou de celui passant par le moins de noeuds par exemple), il devait d'abord lister l'ensemble des chemins possibles avant d'y rechercher celui souhaité. Cette méthode est donc certes réalisable sur de petits graphes ne contenant que peu de noeuds, mais devient inutilisable dans le cas de graphes plus conséquents (comme dans le cas d'une carte) en raison de sa complexité élevé.

Nous avons ensuite abandonné cet algorithme au profit de Dijkstra.

2.4.2 Algorithme de Dijkstra : calcul du plus court chemin

Nous nous sommes ensuite intéressés à la détermination du plus court chemin. Pour cela l'utilisation de l'algorithme de Dijkstra nous a semblé être la meilleure alternative, même si l'implémentation de `find_way_djk` nous a posé plusieurs problèmes.

Principe

L'algorithme consiste en la détermination récursive des itinéraires possibles partant de notre noeud de départ par ordre croissant de longueur. Ainsi, le premier itinéraire obtenu arrivant jusqu'à notre noeud d'arrivé est nécessairement le plus court possible.

Implémentation

Le principe de base de `find_way_djk` est de stocker dans une liste `previous` des triplets de la forme `(noeud_suivant distance_cumulée noeud_précédent)` ainsi que dans une liste `bad_nodes` l'ensemble des noeuds vers lesquels un itinéraire à déjà été calculé. Ces triplets sont fournis par la fonction `next` qui est chargée à partir du graphe de déterminer le prochain noeud permettant de former l'itinéraire de plus petite distance ne menant pas à un noeud de `bad_nodes`, et qui a donc nécessairement déjà un itinéraire plus court menant à lui. La fonction est ensuite capable dès que le noeud d'arrivé est rencontré de reconstituer l'itinéraire à partir de `previous`.

Exemple

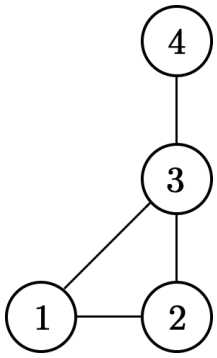

FIGURE 3 – Graphe d'exemple

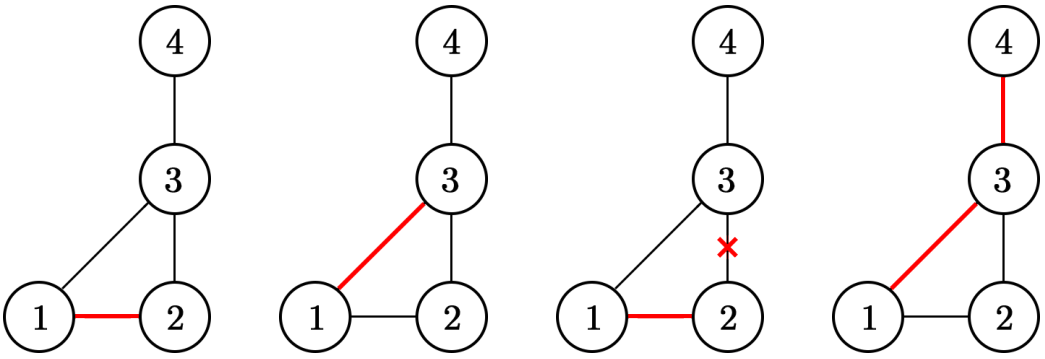

FIGURE 4 – Etapes algorithmiques de Dijkstra

Dans l'exemple ci-dessus on représente les différentes étapes par lesquelles notre algorithme passe lors de l'élaboration du plus court trajet reliant le noeud 1 et 4 :

— Tout d'abord, il regarde parmis les successeurs du noeud 1 lequel est le plus proche, ajoute le trajet 1-2 à `previous` et garde en mémoire que tout autre chemin menant à 2 n'est plus utile (car plus long).
— Il compare ensuite les distances des trajets 1-2-3 et 1-3 (le fait que les deux noeuds d'arrivée soient les mêmes étant une coïncidence). 1-3 étant le plus court, il est ajouté à `previous` et le fait que tout chemin menant à 3 n'est plus utile est gardé en mémoire. Ainsi, on peut remarquer que le trajet 2-3 ne sera jamais utilisé.
— Enfin, le trajet 1-3-4 étant le dernier possible, il est ajouté à `previous` et l'algorithme, remarquant que le noeud d'arrivée est atteint, renvoie l'itinéraire et sa distance. S'il n'avait pas été le dernier trajet possible, il aurait été d'abord comparé aux autres chemins possibles constructibles à partir de `previous`

Note

En premier lieu, les distances inter-noeuds ont été calculées en appliquant le théorème de Pythagore et non la formule de Haversine à leurs coordonées afin de simplifier l'élaboration de `find_way_djk`. En effet, cela permet de se placer dans un graphe plan, et donc d'élaborer des tests et des exemples plus aisément. Ce n'est qu'une fois la fonction achevée que la transition entre les deux méthodes d'élaboration des distances a été effectuée.

Etude de complexité

Soit n le nombre de noeuds du graphe. La fonction `next` parcourt `previous` une fois à chaque appel, et la fonction `find_way_fct` quant à elle appelle `next` autant de fois qu'il y a un noeud plus proche de celui de départ que celui d'arrivé. Sachant que `previous` est plus longue d'un élément à chaque nouvel appel de `next`, on a donc une complexité en $\mathcal{O}(n(n+1)/2)$, *ie* $\mathcal{O}(n^2)$ pour `find_way_fct`. La fonction `extract_way` utilisée pour passer de `find_way_fct` à `find_way_djk` étant de complexité $\mathcal{O}(1)$, on a donc une fonction `find_way_djk` de complexité $\mathcal{O}(n^2)$. Cette complexité est plus élevée que celle de l'algorithme théorique de Dijkstra qui est en $\mathcal{O}(nlog(n))$, il doit donc être possible d'améliorer notre algorithme.

2.4.3 Problème du voyageur de commerce

Le problème du voyageur de commerce pose la question suivante. Etant donné un graphe, quel est le chemin le plus court passant par tous les noeuds ? Ce problème a commencé à être abordé avec la stratégie du *nearest*, qui consiste à partir d'un noeud et ensuite de se déplacer dans le graphe en visitant le noeud le plus proche du noeud sur lequel on se trouve et qui n'a pas encore été visité. On itère de cette manière, jusqu'à avoir visité tout le graphe.

Le problème du voyageur de commerce est réellement plus compliqué que le routage simple entre un point et un autre. La complexité et la longueur du parcours fait que le trajet optimal n'est pas du tout celui qu'on aurait eu en allant simplement d'un point à un autre en prenant chaque point au hasard avec Dijkstra.

2.5 Affichage Web

2.5.1 Réalisation du dessin SVG

Une fois le graphe traité, l'objectif final, rappellons le, était d'assurer un affichage web du graphe et des itinéraires. Pour permettre cet affichage, le graphe devait être transformé en dessin vectoriel svg. Une balise svg est de ce type :

'(svg ((width w) (heigth h)) (path ((d "M x_1 y_1 L x_2 y_2") (stroke color))))

Ainsi pour construire l'arc entre deux nœuds, il fallait accéder à leur coordonnées afin de construire la chaîne de caractères du *path*. Nous avons donc modifié temporairement notre type noeud, ce changement n'a aucun impact sur le travail dans les autres parties puisqu'il ne concerne que le svg.

En effet nous avions une structure du type :

$$(\text{id } (\text{coord}_{id}) \text{ (v1 v2 v3 ..)})$$

Nous avons alors choisi de la transformer en :

$$((coord_{id}) \text{ } ((coord_{v1}) \text{ } (coord_{v2}) \text{ } (coord_{v3})..) \tag{1}$$

Par un *map* nous formions alors tous les *path* partant du nœud id. Finalement la fonction *paths-from-all-nodes* fourni de manière récursive la liste de tous les paths.

```
1 (define (paths-from-all-nodes graph color)
2   (if (null? graph)
3       '()
4       (append (paths-from-a-node (car graph) color)
5               (paths-from-all-nodes (cdr graph)
6                                     color))))
```

Etude de complexité

Un inconvénient que nous avons rencontré est que comme le graphe n'est pas forcément orienté, la fonction de path est appelée inutilement pour certains nœuds. Il aurait été peut-être intéressant de supprimer au préalable les doublons pour génerer temporairement un graphe orienté avant de réaliser le dessin vectoriel. Mais après réflexion nous avons renoncé à le faire compte tenu de la complexité que cela aurait générée. En effet pour un sommet n_1

donné, rechercher ses voisins un à un puis supprimer n_1 de ce voisin aurait été trop complexe temporellement. Si l'on suppose que l'arbre est complet c'est à dire qu'il existe un arc entre tous les sommets deux à deux et que le nombre de sommets initiaux est n, cette opération aurait nécessité $\sum_{i=1}^{n-1} i^2$ opérations élémentaires soit une complexité en $\mathcal{O}(n^3)$ (ce qui n'aurait pas changé la complexité globale comme on le voit par la suite)

En effet la complexité principale de la fonction finale pour le svg vient de l'opération de modification de nœud évoquée en (1). Cette opération pour un nœud est en $\mathcal{O}(n^2)$ dans le pire des cas. Sachant qu'il faut le faire pour tous les nœuds du graphe on note une complexité final en $\mathcal{O}(n^3)$. Voilà pourquoi il n'aurait pas été si intéressant de former au préalable un graphe orienté.

2.5.2 Implémentation du server et interprétation des URL

Nous avions déjà un morceau de code qui lisait une URL, et qui nous renvoyait le reste de la ligne (après le nom de domaine) sous la forme d'une chaîne de caractères.
Nous avons du segmenter les arguments donnés en fonction de la page demandée :

- pour "route" et "distance", deux identifiants correspondants aux tags "start" et "end",
- pour "cycle", une liste ordonnée d'identifiants.

Après avoir récupéré les identifiants sous forme de chaîne de caractères, il a fallu les convertir en entier, puis les envoyer comme argument à la fonction concernée (respectivement ‘route‘, ‘distance‘ et ‘cycle‘, situées dans interprete.rkt).

2.6 Résultats

Nous avons pu tester notre travail sur différentes tailles de données d'entrée. Les résultats obtenus sont les suivants.

— Affichage d'un fichier OSM (**abeille.osm**) contenant 7984 noeuds :

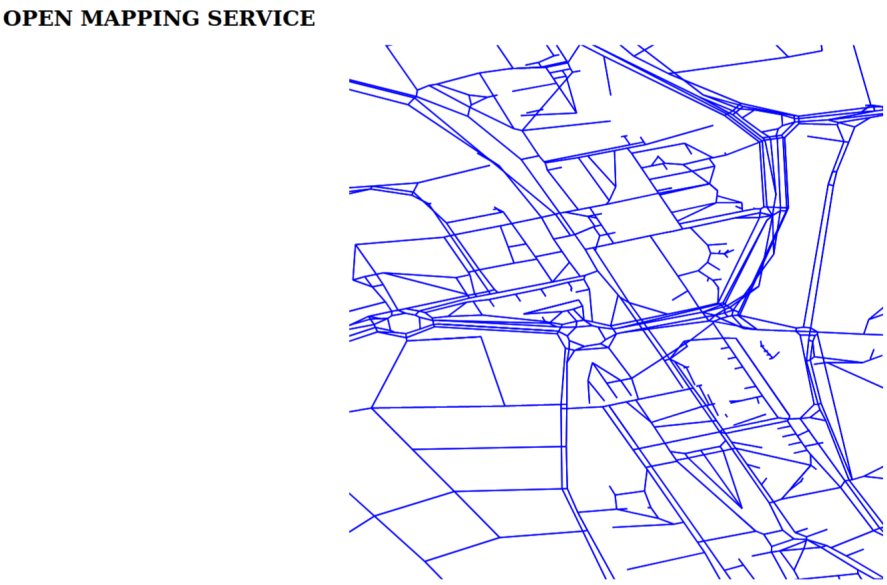

FIGURE 5 – Carte de l'ENSEIRB, ficher **abeille.osm**

— En demandant l'itinéraire entre deux points (affichage pour le fichier **1-lm.osm**) :

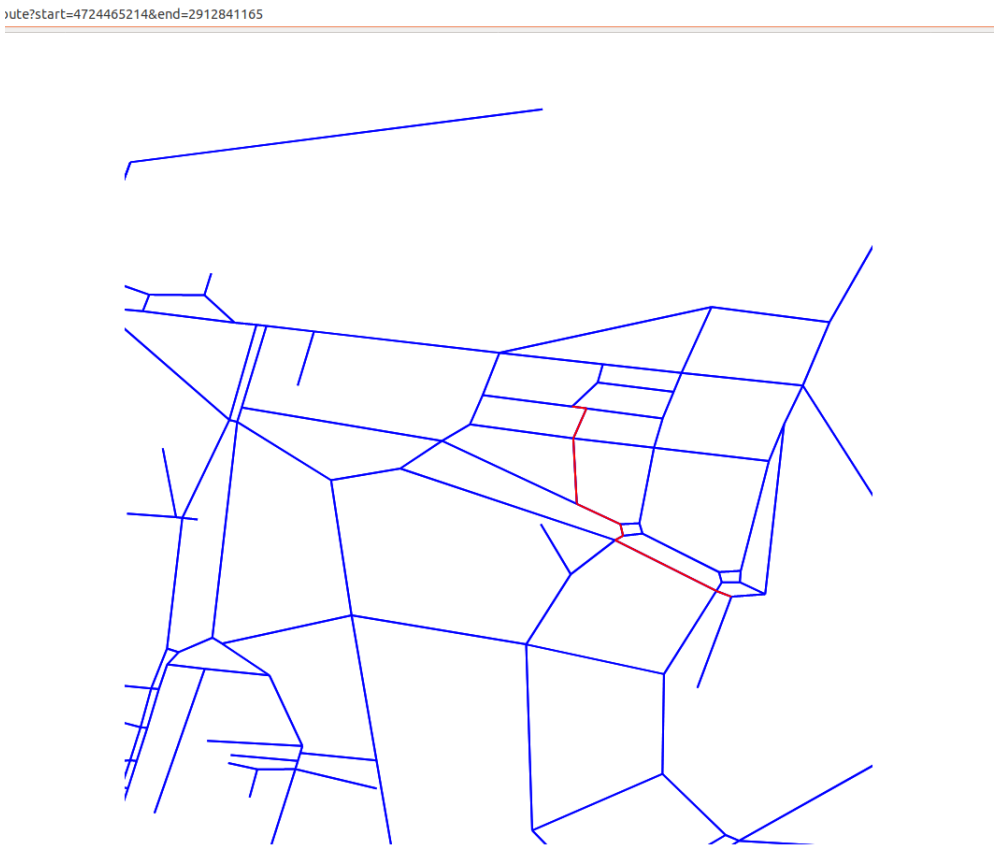

FIGURE 6 – Affichage de l'itinéraire entre deux points

— En demandant la distance entre ces deux points (affichage pour le fichier **1-lm.osm**) :

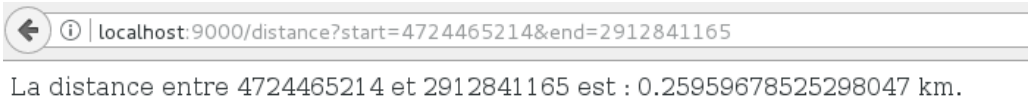

FIGURE 7 – Affichage de la distance entre deux points

3 Améliorations possibles

Il y a plusieurs sources d'améliorations possibles, comme :

— Une précision plus importante sur les distances en gardant les nœuds de degré 2 dans le graphe.

— Une optimisation des algorithmes de recherche de chemin.

4 Conclusion

Ce projet nous a permit de développer et consolider nos compétences en Scheme et de manipuler les objets mathématiques théoriques que sont les graphes, dans le cadre de cartes géographiques. Il nous a également permit d'afficher ces cartes dans un format proche du format xml.