Filière informatique
ENSEIRB-MATMECA

— **PROJET SEMESTRE 6** —

Open Mapping

Emeric DUCHEMIN Laurent GENTY Julien MIENS Tanguy PEMEJA

Encadré par Gaëtan CHAMBRE

Mars – Mai 2019

Table des matières

1 Trier les données **3**
1.1 Récupération de la carte . . . 3
1.2 Filtrage des données . . . 3

2 Structure de graphe **4**
2.1 Graphe . . . 4
2.2 Construction du graphe à partir de données . . . 4

3 Calcul d'itinéraire **4**
3.1 Algorithme de Dijkstra . . . 4
3.2 Algorithme A* . . . 6
3.3 Calcul de distance . . . 6

4 Le voyageur de commerce **6**

5 Server HTTP **7**
5.1 Remplissage du conteneur SVG . . . 9
5.2 Affichages et visualisation . . . 11

6 Conclusion **15**

Table des figures

1 Structure du fichier .osm . . . 3
2 Complexité du filtrage suivant le type de choisi . . . 4
3 Éxécution de Dijkstra . . . 5
4 Exemple d'affichage de graphe . . . 11
5 Architecture du serveur . . . 12
6 Page principale du serveur . . . 13
7 Visualisation d'un noeud sur une page . . . 13
8 Route avec Dijkstra entre 6069094600 et 2140390965 sur map2.osm . . . 13
9 Affichage de la liste successives des noeuds pour un chemin . . . 13
10 Affichage de la distance . . . 14
11 Cycle avec le voyageur de commerce . . . 14

Introduction

Ce projet repose sur les données fournies par le site OpenStreetMap et vise à fournir un service Web de gestion de données cartographiques, calcul d'itinéraires. Le projet est implémenté en Racket et nécessite l'utilisation de graphes pour représenter les rues. Il est tout d'abord nécessaire d'extraire les données depuis un fichier récupéré sur OpenStreetMap, puis d'implémenter des algorithmes de recherche du plus court chemin : Dijkstra et A-étoile. Le problème du voyageur de commerce est ensuite abordé. Enfin, un serveur permettant de visualiser les graphes obtenus ainsi que les itinéraires.

1 Trier les données

1.1 Récupération de la carte

Depuis le site d'OpenStreetMap, il est possible d'obtenir une carte dans un fichier au format `map.osm`. Ce fichier contient différents éléments, notamment des `nodes` et des `ways`.

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="CGImap 0.6.1 (11633 thorn-03.openstreetmap.org)" copyright="C
opyright" license="http://opendatacommons.org/licenses/odbl/1-0/">
 <bounds minlat="44.7450000" minlon="-0.5370000" maxlat="44.7507000" maxlon="-0.5284000"/>
 <node id="28317528" visible="true" version="3" changeset="401485" timestamp="2008-03-26T22
  <tag k="created_by" v="almien_coastlines"/>
  <tag k="source" v="PGS"/>
 <way id="191705737" visible="true" version="4" changeset="64603517" timestamp="2018-11-17T
  <nd ref="2022917772"/>
  <nd ref="6069094600"/>
  <nd ref="2140390951"/>
  <nd ref="2022917774"/>
  <nd ref="2022917776"/>
  <tag k="highway" v="residential"/>
  <tag k="name" v="Orée du Bourg"/>
 </way>
</osm>
```

FIGURE 1 – Structure du fichier .osm

Afin traiter ces informations, nous avons besoin de les récupérer sous la forme d'une liste. Pour cela, on utilise les instructions suivantes décrites dans le sujet :

```
(require xml)

(xml->xexpr (document-element
(read-xml (open-input-file "map.osm"))))
```

1.2 Filtrage des données

Afin de réaliser le projet, de nombreuses informations sont superflus. Ainsi il est nécessaire de filtrer la liste contenant les informations de la carte pour ne garder que les informations qui nous intéressent. En plus de ne garder que les informations utiles contenus dans les `nodes` et les `ways`, ce filtrage permet de retirer toutes les `ways` ne concernant pas des routes. Il permet en autre de retirer les `ways` décrivant les contours des bâtiments. Toutefois afin d'afficher une carte plus réaliste, nous avons implémenté un filtre gardant toutes les `way`.
Ces 2 filtres sont basés sur le même algorithme, celui-ci est un parcours de gauche à droite la liste grâce à un `foldl`. Lors du parcours, on teste si l'élément est un noeud si oui on l'ajoute à une liste sinon on regarde si c'est une `way` acceptable. Ainsi l'algorithme pour sélectionner uniquement les routes est plus coûteux en complexité temporelle car il nécessite de trouver l'élément décrivant le type de `way` avant d'ajouter la `way`. Cette recherche nécessite un parcours supplémentaire de la `way` ainsi pour une carte à `n` éléments, nous obtenons les complexités suivantes :

Choix du filtre	Complexité temporelle
Avec les bâtiments	$O(n)$
Uniquement les routes	$O(n^2)$

FIGURE 2 – Complexité du filtrage suivant le type de choisi

2 Structure de graphe

2.1 Graphe

Le graphe doit contenir un ensemble de noeud ainsi qu'un ensemble d'arcs reliant les noeuds. Pour ce faire, nous utilisons une structure `node` avec les champs suivant :

— `id` : le nom du noeud
— `lat` : sa latitude
— `lon` : sa longitude
— `neighbour` : une liste des id voisins de ce noeud

Ensuite, afin de stocker l'ensemble de ces noeuds, on utilise une table de hashage fournit par racket. Cette table nous permet d'obtenir en temps constant en moyenne un noeud du graphe connaissant son `id`. De plus, cela justifie l'usage de liste contenant uniquement les `id` pour retrouver les voisins.

On utilise l'`id` des noeuds comme clé pour la table de hashage, on peut ainsi obtenir la liste des noeuds du graphe via la liste des clés de la table. Cela nous permet aussi de tester la présence d'un noeud en temps constant.

2.2 Construction du graphe à partir de données

Une fois les données récupérées et nettoyées, il faut maintenant pouvoir construire un graphe utilisable pour la suite. Les données dont nous disposons sont une liste de `node` et une liste de `way`. La première étape consiste à transformer les données en noeuds par parcours linéaire de la liste. Les noeuds obtenus n'ont pour le moment pas de voisin car les arêtes sont stockés dans les `way`. La seconde étape consiste donc à remplir les listes de voisins. Cette étape nécessite de mettre à jour les listes des voisins des deux noeuds concernés.

Il nous était demandé d'implémenter un algorithme permettant de filtrer les sommets de degré deux afin de les éliminer. L'implémentation que nous avons choisi pour réaliser un graphe est parfaitement adaptée à cette réalisation. En effet avec une liste de successeur il est très simple d'enlever tous les noeuds d'une route ayant un degré égal à deux. En effet après avoir enlevé notre noeud il suffit d'appeler récursivement la fonction réalisant ce labeur sur les voisins présent dans une liste.

Après ces opérations, on se retrouve avec un grand nombre de noeuds sans voisin, ceci est dû au fait que tous les noeuds ne participent pas forcément à des routes mais aussi à des bâtiments.

Cependant ne conserver que les sommets de degré 1 ou ceux ayant plus de 3 voisins n'est pas viable sur le long terme. En effet, si on enlève ces noeuds, l'algorithme va traverser des bâtiments ou ne pas suivre la route. Ainsi dans l'algorithme actuel nous enlevons seulement les sommets de degré 0, sommets non atteignables et donc qui ne présentent pas d'intérêt dans des algorithmes de recherche de chemin. Ainsi nous avons donc maintenant laissé seulement le filtre des sommets de degré 1 et retiré le filtre des sommets de degré 2.

Le graphe finalement obtenu représente exactement les données que nous voulions.

3 Calcul d'itinéraire

Sont abordées ici les méthodes utilisées pour calculer les itinéraires entre deux sommets du graphe.

3.1 Algorithme de Dijkstra

L'algorithme de Dijkstra est un algorithme de recherche de plus court chemin dans un graphe. Voici un bref aperçu de son fonctionnement dans le cas général : On commence avec un vecteur initialisé à

$+\infty$ de longueur le nombre de sommet dans le graphe. On commence par mettre à 0 la valeur dans la case du tableau correspondant à notre premier élément. Puis on met à jour la distance de ses voisins. On empile ses voisins dans la pile des sommets à traiter et on note le premier sommet pour ne plus avoir à le traiter. On traite ensuite tant qu'il y a des éléments dans la pile il recommence les différentes opérations.

Nous allons maintenant nous attarder sur les différents choix que nous avons réalisé dans notre implémentation.

Tout d'abord un type abstrait de données similaire à celui d'une pile a été implémenté. Il présente toutes les opérations faites sur les listes appliquées au pile pour le rendre similaire à l'implémentation de celle-ci de façon plus claire. Cependant cette implémentation n'est que similaire. En effet, pour obtenir le sommet à traiter dans la pile, c'est-à-dire celui ayant une distance minimale dans la table des distances, nous parcourons toute la "pile" puis nous sortons cet élément de sa position. Il était possible de garder la structure de pile en gardant celle-ci triée à tout moment. De cette manière il n'y aurait eu qu'à dépiler le premier élément pour avoir le sommet de distance minimale. Cependant cette implémentation de Dijkstra n'était pas viable puisqu'il aurait fallu répéter un tri après chaque insertion ce qui représente une complexité bien trop élevée. Nous avons donc plutôt choisi une complexité linéaire avec une recherche dans la pile à chaque tour.

Voici un exemple d'exécution du code :

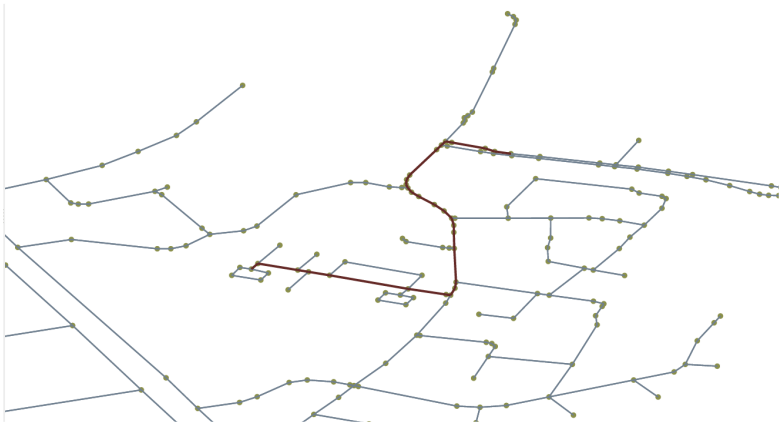

FIGURE 3 – Éxécution de Dijkstra

Nous avons aussi fait le choix de représenter les sommets visiter par un tableau de booléens. Cette implémentation est un peu lourde en terme de complexité spatiale mais présente un avantage certain pour les futurs algorithmes de voyageur de commerce : conserver la trace du passage de l'algorithme sur différents sommets. En effet l'autre solution aurait été de garder la trace du passage directement à l'intérieur du tableau des distances qui ne contiendrait non pas alors la distance mais une liste de deux éléments : la distance et un booléen indiquant si le sommet a déjà été vu. Cependant comme nous le verrons plus tard, l'algorithme du voyageur de commerce a besoin d'appel successif sur Dijkstra en partant de sommet différents. Il doit donc réinitialiser le tableau de distance à chaque nouvel appel, tout en conservant la trace des sommets déjà visité pour ne pas repasser dessus. Le choix tableau est donc la meilleure solution au vu des utilisations futures de Dijkstra.

De plus afin de manipuler les tableaux il faut leur donner un indice. L'implémentation naïve consiste à faire un tableau de la longueur du plus grand id et de mettre dans la case correspondant à l'id la distance du sommet à celle d'origine. Cependant cette solution n'est pas viable. En effet certains id de node dépasse 10 000 000 et réaliser un tableau de taille 10 000 000 semble beaucoup trop lourd. Cependant la structure de tableau de distance est nécessaire à la réalisation de l'algorithme de Dijkstra. Ainsi nous avons donc mis en place un liste d'association.

Son principe est très simple, il consiste à associer à chaque id une valeur entre 0 et le nombre de noeud traité. Ainsi lorsque l'on veut accéder à la distance d'un élément dans le tableau, il suffit juste de regarder l'indice du tableau correspondant à l'id que l'on veut pour après obtenir sa distance dans le tableau.

La complexité spatiale de notre algorithme est de complexité $\mathcal{O}(A * P)$ avec $|E_G|$ le nombre d'arc dans le graphe, en effet on parcourt au maximum tous les arcs. Et P est la hauteur maximale de la pile qu'il faut parcourir pour retrouver le sommet de distance minimale. Elle peut être potentiellement égale au nombre de sommet. La complexité de l'algorithme est ici gonflée par la recherche du minimum dans la pile. Cependant celle-ci reste dans la plupart des cas relativement faible et permet à notre algorithme de rester plutôt rapide.

3.2 Algorithme A*

L'algorithme A-étoile est un calcul de calcul du plus court chemin dans un graphe lorsqu'on dispose d'une heuristique pour estimer quel pourrait être le meilleur chemin à suivre depuis un noeud. C'est notre cas ici, la distance à vol d'oiseau peut nous donner une idée du chemin restant à parcourir.

Son implémentation requiert la mémorisation des listes suivantes :

— liste des noeuds en attente
— liste des noeuds traités
— liste des prédécesseurs

L'algorithme est le suivant :

On part du noeud de départ, on ajoute tous ces voisins à la liste en attente, on prend dans la liste en attente le noeud le plus proche à vol d'oiseau de la cible, on le met dans la liste des noeuds traités et on réitère sur ce noeud jusqu'à atteindre le noeud cible ou jusqu'à ce qu'il n'y ai plus de possibilité. On retrouve alors le chemin en partant de la fin et en remontant en suivant les prédécesseurs des noeuds enregistrés au fur et à mesure dans la liste des prédécesseurs.

L'implémentation proposée conserve les trois listes énoncées plus haut comme variables globales aux fichiers. Le programme commence par initialiser ces listes à vide, puis lance la recherche du chemin pour ensuite le reconstruire si on en a trouvé un. Il est nécessaire de se remémorer la distance à vol d'oiseau pour chaque noeud visité afin de ne pas avoir à la recalculer à chaque itération. C'est pourquoi la liste des noeuds en attente contient les couples `(id, distance)`, la liste des noeuds traités ne contient que les `id` et la liste des prédécesseurs contient les couples `(id, id_pere)`.

La sélection du noeud le plus susceptible de former le chemin est faite en triant la liste des noeuds en attente sur les distances puis en prenant le premier éléments, il est alors extrait de la liste.

L'algorithme fonctionne correctement lorsqu'il n'a pas à rebrousser chemin, sinon il a comportement étrange que l'on a pas su corriger.

3.3 Calcul de distance

Afin de calculer la distance `d` entre 2 sommets S_1, S_2 à partir de leurs latitudes φ_1 (respectivement φ_2), et de leurs longitudes λ_1 (respectivement λ_2), nous utilisons la `formule de Haversine` pour une sphère de rayon r correspondant à la Terre (6371 km) :

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right) \quad (1)$$

Dès lors il est facile de calculer la distance d'un chemin partant d'un noeud vers un autre en additionnant successivement la distance entre les noeuds parcourus.

4 Le voyageur de commerce

Le but de cette partie est l'implémentation d'une heuristique réalisant le voyageur de commerce. Le voyageur de commerce ne peut être réalisé qu'au travers d'une heuristique car pour trouver le véritable chemin optimale il faudrait plus qu'une vie si l'on ne considère ne serait-ce que 30 points. En effet

l'algorithme du voyageur de commerce est de complexité NP-complet. L'heuristique que nous avons implémenté pour nous approcher de la véritable valeur est l'algorithme des plus proches voisins. Cet algorithme consiste à aller de ville en ville en choisissant à chaque fois celle qui est la plus proche du point où l'on se trouve actuellement. Nous avons implémenté l'algorithme comme suit : Il choisit tout d'abord la première ville de la suite de ville proposée. Il lance ensuite Dijkstra sur ce sommet et regarde quelle ville parmi celles encore présentes dans la liste des villes à visiter est la plus proche. Il récupère et conserve le chemin pour aller du point de départ à la nouvelle ville. Il choisit ensuite la nouvelle ville comme nouveau point de départ et recommence ensuite de nouveau le même algorithme jusqu'à avoir visité toutes les villes. Sa complexité telle que nous l'avons implémentée vaut donc N fois la complexité de Dijkstra avec N le nombre de ville à visiter.

Parmi les choix que nous avons fait il y a le tableau de villes visitées. En effet, dans l'algorithme du voyageur de commerce le chemin ne doit pas repasser deux fois au même endroit. Or la modification de tableau se fait par effet de bord en racket, puisque nous avons implémenté des algorithmes de manière fonctionnelle nous avons donc rendu toutes nos opérations sur les vecteurs fonctionnelles. Ainsi lorsque nous appelons Dijkstra nous le faisons avec une copie du tableau des noeuds déjà visité. Ce tableau, après appel de Dijkstra, est d'ailleurs libéré puisqu'il est défini à l'intérieur d'un `let` dont l'environnement de définition s'arrête à l'intérieur de la fonction. Le code de $sort - Dijkstra$ l'illustre parfaitement.

Nous avons aussi implémenté $auxiliary - function - nearest$. Cette fonction permet l'économie d'un calcul de l'algorithme de Dijkstra pour chaque ville ce qui représente un gain de temps important. En effet, il faut vérifier après appel à Dijkstra si un cycle est toujours possible. Ainsi il faut tester si la valeur de retour est la liste vide (ce que l'on renvoie quand la réalisation d'un cycle n'est pas possible) ou si c'est bien une liste composée des villes auxquelles on peut accéder depuis la ville de départ. Cette fonction permet donc avec un unique appel à Dijkstra de filtrer le cas favorable et le cas défavorable.

5 Server HTTP

Maintenant que nous avons toutes les données générées par le *parsing* du fichier `OSM` et le graphe associé, il ne reste plus qu'à l'afficher. L'implémentation d'un serveur `racket` nous a permis de pouvoir visualiser.

Nous avons créer un serveur grâce à la bibliothèque `web-server/servlet` et `web-server/servlet-env`. Ce serveur aura pour missions de :

- gérer le routage
- donner accès à des formulaires de saisie
- gérer les cas d'erreurs relatives aux demandes client
- afficher les informations relatives au graphe

Le fichier `server.rkt` permet donc de gérer l'intégralité du fonctionnement du serveur. Dans un premier temps il est nécessaire de charger la source OSM et de générer le graphe associé. Il est important de noter que la communication du contenu des pages par le serveur se fait grâce à l'utilisation d'x-expressions régulières `xeprx`.

Afin de pouvoir initialiser le graphe, nous avons lier les paramètres donnés en ligne de commandes au chargement du graphe :

```
(define src-file (vector-ref (current-command-line-arguments) 0))
(define maps (xml->xexpr (document-element
                          (read-xml (open-input-file src-file)))))
(define g (osm-to-full-sorted maps))
```

Nous allons stocker le chemin vers le fichier dans la variable `src-file` afin de pouvoir ouvrir en format `XML` ce fichier et le stocker dans le fichier `maps`. Par la suite, nous allons convertir ce fichier depuis le format `OSM` en graphe pleinement construit.

Par exemple, `racket src/server.rkt maps/bergerac.osm` récupérera la carte de Bergerac et en créera un graphe complètement trié. À noter qu'il s'agit de cette étape qui prend le plus du temps

dans l'intégralité de nos algorithmes. En effet, si le fichier est lourd, on pourra considérer plus de 50000 lignes, alors la construction et le filtrage prendra plus de 2 fois la taille du fichier.

Afin de pouvoir afficher le graphe à l'écran, il était nécessaire de trouver un moyen afin de construire une balise dans le code html. Pour ce faire, l'idée était de remplir un conteneur `SVG`. Le fichier `svg.rkt` lui est d'ailleurs dédié afin de construire, remplir et vérifier tous les noeuds dans le conteneur `SVG`.

Ce conteneur va avoir :

- l'entête `SVG`
- la liste des noeuds
- la liste des chemins entre les noeuds

En théorie, il suffirait donc de lister les noeuds du graphe d'une quelconque manière en créant pour chaque noeud une balise `circle` avec comme x la latitude du noeud et comme y la longitude. Ensuite, rajouter les chemins en utilisant la liste des voisins du noeud décrite dans le graphe section 2. Au final, après avoir parcouru tous les noeuds nous allons nous retrouver avec le conteneur rempli avec l'entête `SVG`, les noeuds puis les chemins.

Le format d'utilisation de `SVG` en *racket* est spécial : il faut lui créer des listes et les ajouter afin de créer une grande liste avec comme balise englobante, la balise `SVG`. Le format de communication du corps des pages HTML en *racket* se fait grâce à l'utilisation d'expression régulières `xeprx`. La suite des données concernant le `SVG` seront donc contenues dans des listes que nous allons concaténer afin d'obtenir la grande liste finale HTML que nous donnerons au serveur qui se chargera de la traiter et de l'afficher.

Nous avons donc le code 1 suivant contenant l'algorithme de génération du conteneur `SVG` final :

Algorithm 1 graphToSVG(E G : graphe) : liste de balises

```
enteteSVG ← "<svg width=1280 height=720 viewBox="0 0 1280 720">"
circles ← get-nodes(G)
lines ← get-ways(G)
svg ← concatListes(enteteSVG,circles,lines)
retourner svg
```

À partir du graphe donné en paramètre, nous allons ajouter les noeuds dans une liste, faire de même pour les chemins, et ensuite créer le conteneur `SVG` final.

Cependant, un problème intervient, si l'on rempli bêtement les noeuds dans le conteneur `SVG`, nous allons nous retrouver avec un amas de points sur les pages. En effet, si l'on prend une carte avec des points plus ou moins proches, nous devons effectuer une conversion sur les positions du noeuds : la position en pixels sur la page est différente de celle dans la réalité.

Pour ce faire, nous avons utilisé la formule 3 permettant de mettre à l'échelle, selon les positions maximales en latitude et longitude :

$$\frac{nLatitude - minLatitude}{maxLatitude - minLatitude} * height \tag{2}$$

$$\frac{nLongitude - minLongitude}{maxLongitude - minLongitude} * width \tag{3}$$

Cette conversion nous permet donc d'adapter la position d'un noeud selon les bornes maximales et minimales de l'ensemble du graphe. Nous allons replacer le point en centrant et réduisant selon les proportions de la page.

Cependant, cette formule n'est pas complète. Selon que l'on prenne une coordonnée géographique dans l'hémisphère sud ou nord, ou bien que l'on soit à l'ouest du méridien de Greenwich ou à l'est, il est possible que les positions converties soient inversées. En effet, lorsque l'on passe en dessous de l'équateur, la longitude devient négative et pareil pour la latitude lorsqu'un point se trouve à l'ouest du méridien de Greenwich.

Afin d'avoir des graphes représentatifs de la réalité, nous avons donc changer notre formule afin de prendre en compte la potentielle rotation de la carte.

$$height - \frac{nLatitude - minLatitude}{maxLatitude - minLatitude} * height \tag{4}$$

$$width - \left(1 - \frac{nLongitude - minLongitude}{maxLongitude - minLongitude}\right) * width \tag{5}$$

Concernant le calcul des bornes maximales et minimales du graphes nous avons fait le choix de ne pas les prendre dans la balise :

```
<bounds minlat="44.8297000" minlon="-0.5794000"
maxlat="44.8361000" maxlon="-0.5691000"/>
```

En effet, lors du téléchargement du fichier `OSM`, cette balise nous permet d'obtenir les extremum de **tous** les noeuds contenus dans le fichier. Cependant, ces bornes prennent en compte les positions des noeuds dans le fichier, mais aussi les positions des noeuds qui ne sont pas stockés dans le fichier directement, mais par l'intermédiaire des chemins qu'ils ont en commun avec les noeuds présents. Ce qui veut dire, que si l'on prend ces informations dans cette balise, il est possible que nous ayant des points en dehors du conteneur `SVG`.

Nous avons fait le choix de parcourir le graphe afin de récupérer les bornes ce qui augmente notre complexité de $\mathcal{O}(|V_g|)$ avec V_g le nombre de noeuds dans le graphe. C'est un compromis que nous avons fait, cependant, nous partons du principe que nous utilisons un conteneur `SVG` afin de pouvoir voir **tous** les noeuds. Il est donc pertinent de connaître les "vraies" bornes du graphe afin de pouvoir l'afficher en entier. Si nous avions fait la visualisation des noeuds avec notamment une boîte dans laquelle nous pouvions naviguer (à la manière d'un *slider*) alors il aurait été préférable de garder ces bornes extérieures.

Afin d'éviter de recalculer les bornes entre chaque page lorsque nous affichons des informations relatives au graphe, nous avons aussi fait le choix d'utiliser des effets de bords sur des variables définies en *top-level* dans le fichier `server.rkt` afin de pouvoir toujours y avoir accès. Il faut au maximum éviter les fonctions d'affectation lorsque l'on fait de la programmation fonctionnelle (car cela serait ne pas suivre le paradigme utilisé), il est parfois pertinent de le faire.

De plus, nous avons aussi fait le choix d'utiliser le `html-svg` en tant que variable top-level. Tout comme le graphe `g` lui aussi déclaré en top-level, c'est une information que nous souhaitons avoir dans chacune des pages sans avoir à recalculer. Concernant la création des routes nous pouvons potentiellement utiliser le `SVG` déjà créé le modifier.

5.1 Remplissage du conteneur SVG

Nous avons donc un conteneur qui doit être sous la forme :

```
<svg width=1280 height=720 viewBox="0 0 1280 720">
    <circle cx="485.78533123664386" cy="235.86806743185622"
    r="3" fill="#000000"></circle>
    <line x1="720.1093665578014" y1="329.4838993403599"
    x2="485.78533123664386" y2="235.86806743185622"
    r="3" fill="#000000"></line>
    ...
</svg>
```

Afin de pouvoir remplir le conteneur `SVG`, nous avons utiliser une fonction interne à *racket* : `hash-map`. En effet, notre structure de données concernant le graphe étant une table de hashage, il est efficace d'utiliser la fonction `hash-map`.

Comme nous l'avons dit précédemment, nous voulons avoir la liste des noeuds pour le conteneur `SVG`. Nous allons donc lister les noeuds de la table de hashage et appliquer une fonction qui va créer le cercle pour le conteneur `SVG` relatif à chaque noeud. Cette procédure va s'appliquer pour chaque noeud du graphe et renvoyer une liste formatée pour l'utilisation des x-expressions régulières comme précédemment énoncé.

```
circles (hash-map graph create-circle)
...
(define (create-circle id n)
    `(circle ((cx ,(lon-scaled n))
              (cy ,(lat-scaled n))
              (r "3")
              (fill "#8b904b"))))
```

Cette procédure va donc prendre en paramètre le noeud de la table de hashage et créer une liste associée à ce noeud en particulier. Dans la variable `circles` nous allons donc avoir la liste des noeuds, à savoir une liste de liste sous la forme :

```
'((circle ((cx "898.7064553848617")
           (cy "260.01596221171116")
           (r "3")
           (fill "#8b904b")))
 (circle ((cx "939.7267045683672")
           (cy "222.44782148381438")
           (r "3")
           (fill "#8b904b")))
...)
```

La construction d'une telle liste se fait en $\mathcal{O}(|V_g|)$ avec $|V_g|$ le nombre de noeuds dans le graphe.

Concernant la construction de la liste des chemins des voisins, la version actuelle du rendu du projet se fait à la suite de la création de la liste des noeuds, ce qui n'est bien entendu pas optimal en terme de complexité.

Le principe est le même, nous allons appliquer la fonction `hash-map` sur notre graphe afin que pour chaque noeud, il crée l'ensemble des chemins, et cela pour tous les noeuds du graphe :

```
'(  ;; premier noeud
    ((line ((x1 "720.1093665578014") (y1 "329.4838993403599")
            (x2 "485.7853312366438") (y2 "235.86806743185622")
            (r "3") (fill "#000000")))
    (line ((x1 "870.659592626620") (y1 "300.4838954548699")
            (x2 "109.7867434466438") (y2 "306.86805454185622")
            (r "3") (fill "#000000"))))
    ;; deuxieme noeud
   (line ...))
```

Donc grâce à `lines (reformat-lines (sort-empty-lines (hash-map graph create-line)))` nous obtenons la liste des chemins `line` sous forme de liste de liste de liste (pour chaque noeud, nous avons une liste associée de voisins).

Cette construction se fait en $\mathcal{O}(|E_g| * 2)$ avec $|E_g|$ le nombre d'arcs présents dans le graphe. Le faire n'est pas si lourd dans la mesure où il s'agit d'une complexité linéaire en nombre d'arcs. Il y a seulement un problème : la construction du graphe actuel ne prend pas en compte les sens des routes, et donc l'arc est traité deux fois dans la mesure où il apparaît dans la liste des voisins de deux noeuds. Si nous avions pris en compte un graphe orienté, cette complexité serait descendue en $\mathcal{O}(|E_g|)$ (en faisant les changements nécessaires lors de la construction du graphe bien évidemment).

Nous retrouvons donc avec une complexité de $\mathcal{O}(|V_g| + |E_g| * 2)$ afin de construire l'ensemble du conteneur `SVG`. Le fait est que cette complexité est largement plus faible que lors du filtrage des données partie 1. Si nous prenons un fichier par exemple de 60000 lignes `OSM`, nous pouvons nous retrouver qu'avec "seulement" 1200 noeuds. Cependant, elle n'en reste pas moins élevée.

Cette complexité aurait pu être largement supérieure si nous avions eu une architecture de données mal implementée, par exemple en utilisant une liste de liste pour la représentation du graphe.

Une fois les chemins et noeuds obtenus, il nous suffit seulement de créer un `SVG` final et de le renvoyer, on concatène la syntaxe `SVG` au reste :

```
final-svg (append `(svg ((width ,width_box) (height ,height_box)
                         (viewBox ,box) (xmlns "http://www.w3.org/2000/svg")))
                  circles-lines)
```

Encore une fois, il est important de préciser que nous avons stocké cette liste `SVG` au top-level en l'affectant dans la mesure où dans les pages que nous allons présenter juste après, nous en aurons besoin et tout cela sans recalculer.

5.2 Affichages et visualisation

Nous obtenons donc un graphe final ressemblant à la Figure 4. Sur le graphe nous avons donc les noeuds qui sont affichés en vert clair. Ces noeuds sont positionnés relativement par rapport aux bornes maximales du graphe dans lequel ils appartiennent. Les chemins quant à eux, sont indiqués en bleu et relient les noeuds du graphe.

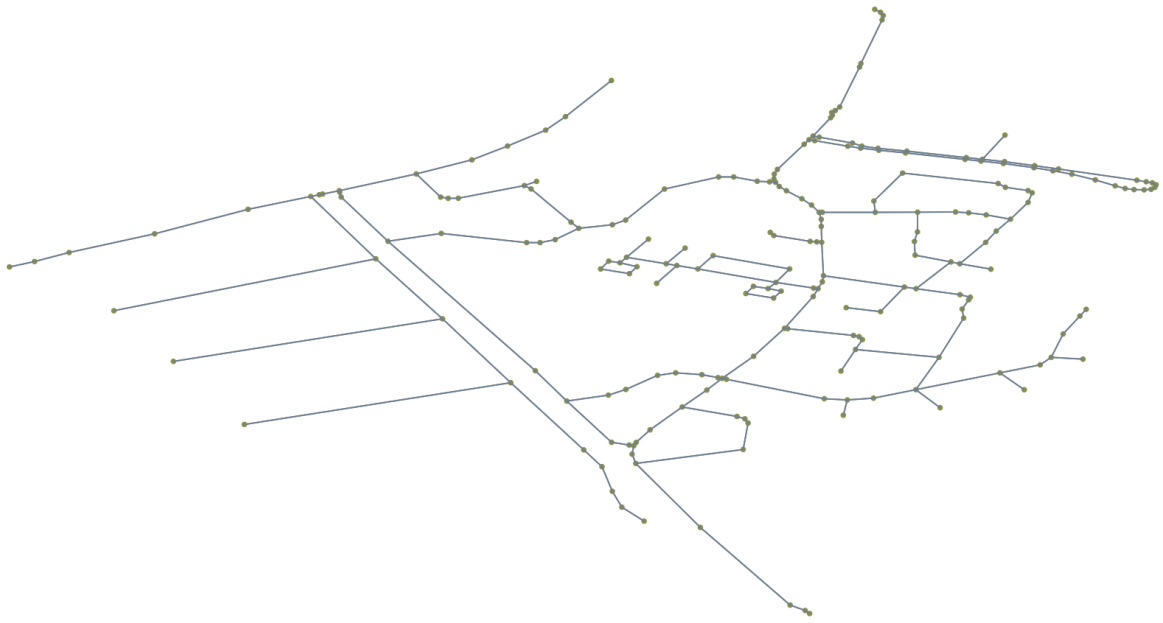

FIGURE 4 – Exemple d'affichage de graphe

Dans la mesure où nous avons fait le choix de placer tous les noeuds du graphe dans la boite de visualisation, il est possible que selon le graphe que nous ayons choisi lors de l'exécution du serveur *racket* la carte paraisse étirée. Si un noeud se retrouve très éloigné des autres, alors cela aura pour effet d'étirer les positions de tous les noeuds du graphe.

De plus, comme précédemment énoncé dans la partie 1, nous avons fait le choix de ne pas prendre en compte tous les noeuds autres que ceux appartenant à des routes. Le graphe peut donc paraître un peu vide, cependant il contient seulement les noeuds dans lesquels nous voulons effectuer les algorithmes de parcours et de recherche de chemin. Dans les pages `html` nous affichons la liste des noeuds du graphe, ce qui nous permet d'en sélectionner un et que l'on soit sûr qu'il s'agisse d'un noeud d'une route. Si nous avions garder les bâtiments, cela aurait été encore de la surcharge d'informations pour rien. Cependant, comme énoncé précédemment, il est possible d'utiliser la fonction de filtrage `filt2-all-way` permettant de prendre en compte tous les noeuds, mais cela a bien évidemment un coût en terme de complexité.

Le serveur va donc gérer la gestion des routes lors de l'utilisation. La fonction `server-dispatch` nous permet donc de naviguer selon les pages. Si l'on met `"node"` dans l'URL de la page html, alors nous allons être redirigé vers cette page.

```
(define-values (server-dispatch server-url)
(dispatch-rules
 [("node") node-page]
 [("route") route-page]
 [("distance") distance-page]
 [("cycle") cycle-page]
 [else main-page]))
```

La Figure 5 nous montre donc l'architecture globale du serveur `HTML`.

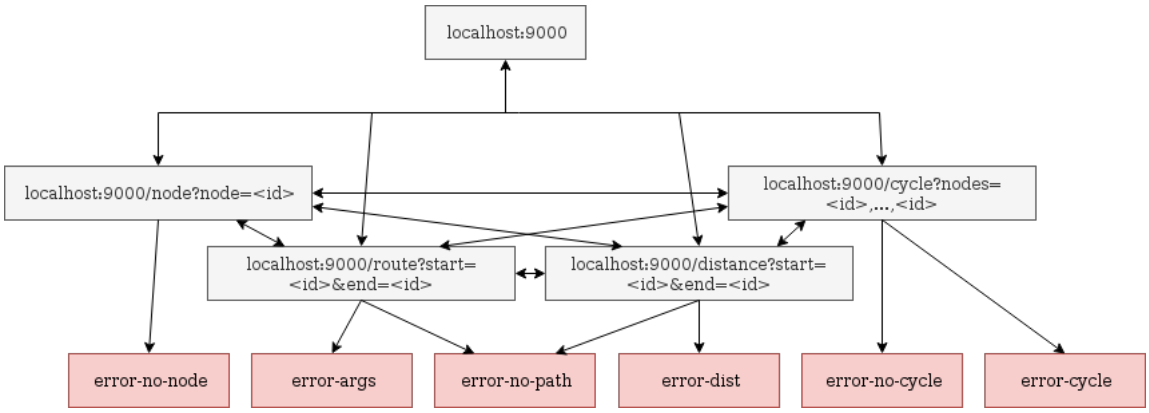

FIGURE 5 – Architecture du serveur

Afin de pouvoir lancer le serveur, il faut donc lancer la commande suivante : `racket src/server.rkt maps/<map` Cela lancera un serveur sur le hôte local sur le port 9000. Ouvrez votre navigateur internet et allez sur la page `localhost:9000` comme sur la Figure 6. Le lancement du serveur peut varier selon la taille du fichier OSM donné en entrée : cela peut varier de quelques secondes, à plusieurs dizaines de secondes.

Sur cette dernière nous avons les informations générales concernant le graphe :

— les boutons de navigation
— le nombre de noeuds
— le nombre de chemins
— la liste des noeuds
— le graphe

Il est possible de naviguer sur la page `localhost:9000/node?node=<id>`. Cette page sert notamment à pouvoir visualiser un noeud en particulier, faute d'avoir implémenté un script dynamique nous permettant d'obtenir l'`id` des noeuds en passant sa souris par dessus par exemple.

Il est possible de remplir un formulaire dans lequel le noeud est indiqué (l'intérêt d'avoir lister les noeuds sur la page est donc pertinent et utile ici). Une fois que vous avez appuyé sur le bouton `Submit`, le noeud va s'afficher en rouge comme sur la Figure 7. Cependant, si le noeud n'existe pas, alors une page nous préviendra.

De plus, il est possible de trouver les chemins d'un point A à un point B. Pour cela il suffit d'accéder à la page `localhost:9000/route?start=<id>&end=<id`. Sur cette page nous avons un formulaire de saisie permettant de rentrer les noeuds sur lesquels nous voulons appliquer l'algorithme de recherche du plus court chemin. L'algorithme utilisé sur la page est celui de Dijkstra.

La Figure 8 nous montre la sortie sur la page avec les noeuds sélectionnés.

Le chemin s'affiche comme sur la Figure 9 à savoir comme une liste de noeuds successifs avant d'atteindre l'arrivée.

A noté que comme nous l'avons énoncé au début de cette section, le serveur gère les cas d'erreurs. La Figure 5 nous montre les différentes erreurs qui peuvent arriver. Sur cette page plusieurs erreurs peuvent arriver :

FIGURE 6 – Page principale du serveur

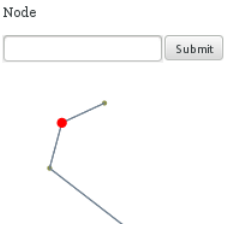

FIGURE 7 – Visualisation d'un noeud sur une page

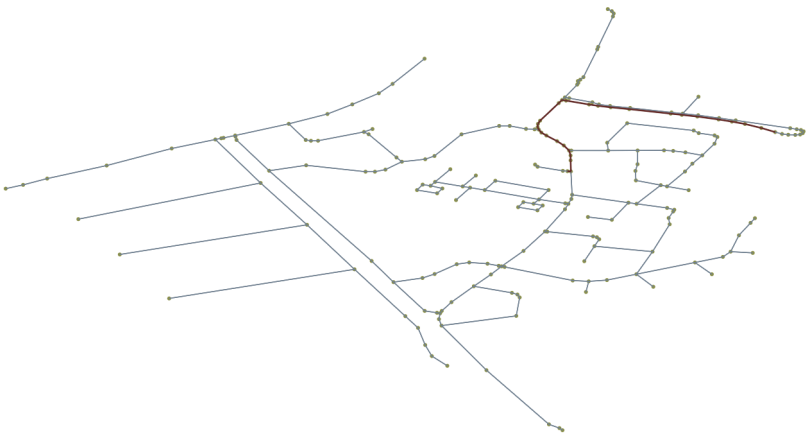

FIGURE 8 – Route avec Dijkstra entre 6069094600 et 2140390965 sur map2.osm

Path : (6069094600 2022917772 283653051 2140390954 283653052 2140390956 283653053 2140390957 2140390964 283652957 2140390972 2140390974 283653055 6069094595 2140390998 6069094597 2140390995 2140390994 2140390991 2140390989 2140390986 2140390984 2140390980 2140390977 2140390975 2140390973 2140390971 2140390965)

FIGURE 9 – Affichage de la liste successives des noeuds pour un chemin

— mauvais nombre d'arguments
— noeuds qui n'existent pas dans le graphe
— noeuds égaux
— n'existe pas de chemin entre les noeuds : `"Disconnected Universe Error"`

Dans ces cas là, une page d'erreur sera affichée avec l'usage de la page en question.

Concernant la page `localhost:9000/distance?start=<id>&end=<id>` il y a seulement une différence par rapport à la précédente page, à savoir que la distance en mètres est affichée en plus sur la page grâce à la formule de Haversine décrite dans la section 3.3.

Route between node : 2549065239 and node : 2549065224

Path : (2549065239 2549065224)

Length of the path : 2

Distance : 33.27101148937886 meters

FIGURE 10 – Affichage de la distance

Encore une fois, si l'utilisation de la page n'est pas bonne, pour les mêmes raisons que celle de la route, alors les pages d'erreurs seront levées.

Enfin, pour finir sur les cycles, la page `localhost:9000/cycle?nodes=<id>,...,<id>` permet de lancer l'algorithme du voyageur de commerce afin de trouver un chemin passant par tous les noeuds donnés en paramètres.

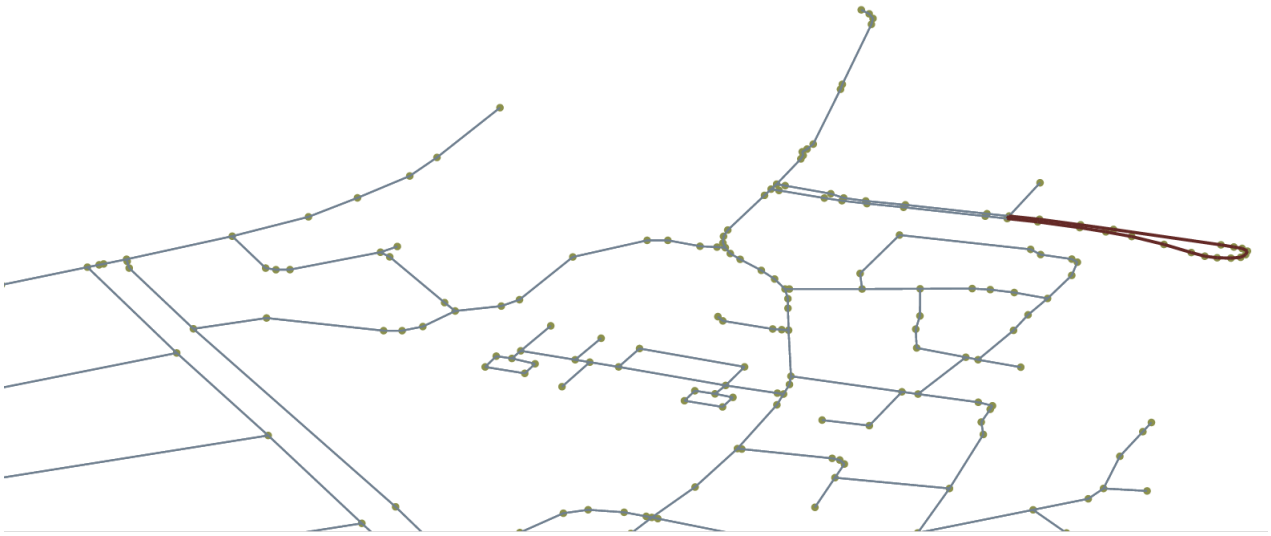

FIGURE 11 – Cycle avec le voyageur de commerce

Attention, dans le formulaire de saisie il est important de noter qu'il faut séparer les noeuds par des virgules, auquel cas cela aura aussi pour effet de lancer la page d'erreur. Tout comme le fait de donner des noeuds n'existant pas ou bien qu'un cycle entre les noeuds donnés n'existe pas (`"Disconnected Universe Error"`).

La version actuelle du site pourrait cependant être améliorée. En effet, nous aurions pu utilisé la librairie *JQuery* afin notamment d'obtenir une meilleure ergonomie durant la navigation de ce dernier. En effet, malgré le fait que nous ayons affiché la liste des noeuds sur la page et ajouté une page nous permettant de visualiser un noeud en particulier, nous aurions pu faire en sorte que si nous passons par exemple notre souris sur un point, alors l'`id` s'affiche. De plus, en ce qui concerne les algorithmes de recherche de chemin, lorsque nous les affichons à l'écran, il nous est difficile de voir le début et la fin concernant Dijkstra ou bien tous les noeuds par lesquels doit passer le voyageur de commerce. Cependant, par manque de gestion du temps, nous n'avons pas eu le temps de l'implémenter.

6 Conclusion

Nous avons proposé une implémentation d'un serveur capable de lancer différents algorithmes de recherche de chemin. Ces algorithmes s'exécutent aujourd'hui rapidement, moins de 5 secondes sur une grande map (1210 noeuds) et sont fonctionnels. Nous avons implémenté la structure de graphe à l'aide d'une table de hashage qui nous permet d'être beaucoup plus rapide dans la recherche de voisins.

Cependant, le réel problème ne vient pas de là. Comme énoncé précédemment, il vient surtout de la complexité lors du traitement du fichier OSM. En effet, la lourdeur des algorithmes fait que les temps de génération d'un graphe à partir d'un fichier OSM sont (très) longs. Nous pouvons passer de plus de 60000 lignes dans le fichier, à un graphe avec moins de 1000 noeuds (dans la mesure où nous avons fait le choix d'enlever les noeuds des bâtiments). Donc malgré le fait que nous ayons essayé d'être le plus efficace possible dans le filtrage en limitant au maximum la quantité de données lues, ce problème reste handicapant. Nous avons aussi essayé de réduire les coups de recherche dans les graphes afin de ne pas augmenter la complexité d'algorithmes gourmands utilisant les fonctions de recherche du plus court chemin, comme par exemple les algorithmes du voyageur de commerce.