WhyNotFinding

Rapport de Projet

Maximilien	Valentin	Florian	Julien
CHARMETANT	DAUZERE-PERES	LE ROUX	LUNG YUT FONG

2021

Rapport de Projet

Table des matières

1. Introduction

FIGURE 1 – Logo de notre projet WhyNotFinding

1.1. Cadre de réalisation

Dans le cadre du quatrième semestre à l'EPITA, un projet de groupe devait être réalisé. Il a donc fallu choisir un thème et un sujet correspondant au cahier des charges donné par la scolarité. Le projet devait être un logiciel dans lequel l'algorithmique avait une part très importante. Le sujet était libre. La création d'un logiciel de navigation, ayant une grande composante algorithmique, fut l'idée retenue.

1.2. Présentation du groupe

Le projet étant un projet propre à l'école, notre groupe s'est formé dans une classe à l'EPITA. Les membres du groupe se sont un réunis au hasard. Bien que le groupe se soit fait par pur hasard, le groupe entier a été emballé par le sujet qui est intéressant pour quelqu'un passionné par l'informatique. La communication a été efficace tout au long de la réalisation du projet, ce qui permet d'avoir une bonne cohésion de groupe.

1.3. État de l'art

1.3.1. Programmes existants

FIGURE 2 – Respectivement les logos de Google Map, Waze et CityMapper.

Google Maps :

Google maps est probablement le logiciel de naviguation le plus connu, il est donc une référence en terme de logiciel de naviguation. Il est très complet au niveau des fonctionnalités qu'il propose. Il permet de calculer le temps et de choisir un chemin en fonction du transport choisi, ce qui implique qu'il prend en compte les transports en commun et leurs horaires. Il permet également de faire des chemins en considérant des points par lesquels il faut impérativement passer, il considère aussi le trafic routier lors du calcul du plus court chemin. Cela est en partie possible grâce à la quantité de données qu'il possèdent.

Waze :

Waze est également très connu, mais pour des raisons differentes, il est spécialisé dans le déplacement en vehicule motorisé. Dans ce domaine, il excelle, il permet de choisir un chemin en considérant les embouteillages et le fait mieux que google maps

City Mapper :

City Mapper est un logiciel qui est quant à lui spécialisé dans les transports en commun, il permet comme Google Maps d'avoir le chemin le plus court en empruntant des transports en commun, il est cependant plus ergonomique pour cette utilisation.

1.3.2. Les Algorithmes de parcours

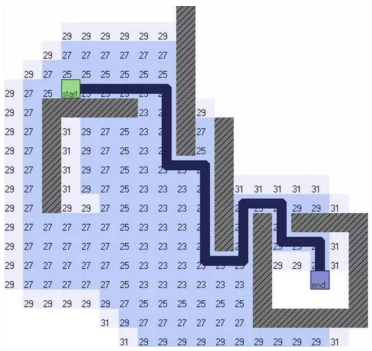


FIGURE 3 – Exemple d'application d'un algorithme de recherche du plus court chemin.

Les algorithme de parcours de graphe sont relativement récents car le plus vieux trouvé lors des recherches liées à ce projet était celui de Dijkstra qui date de 1959, le deuxième étant la première version de A* qui date de 1968. A* posséde énormément de différentes versions et la dernière en date trouvée date de 2011 et s'appelle Tree-AA* qui est lui-même une variante de Adaptive A* et de D* Lite (eux même des variantes évidemment). Tree-AA* a pour particularité, en plus qu'il soit prouvé qu'il peut être plus rapide que ses homologues, d'avoir une heuristique incrémentale, ce qui n'est pas le cas de A* de base qui utilise une heuristique pré calculé (la méthode d'évaluation heuristique étant détaillé dans la partie 2.3.2.a). Toutes ces versions de A*, sont toutes dans l'idée bien plus rapide que Dijkstra, et sont utilisé notamment dans les jeux vidéos comme algorithme de pathfindings (un algorithme qui permet trouver un chemin en temps réel pour qu'un personnage puisse se déplacer de manière cohérente).

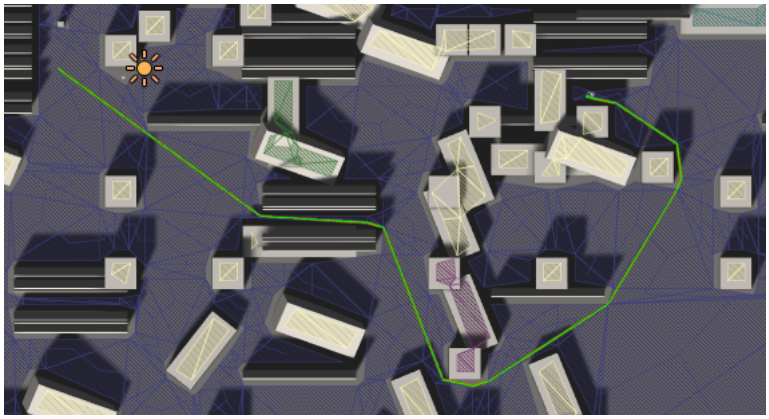


FIGURE 4 – Exemple d'application d'un algorithme de pathfinding A*.

Il n'y a donc pas un grand nombre d'algorithmes mais un grand nombre de variantes, il faut donc se repérer dans toutes ces différentes variantes plus que de juste chercher à faire la dernière en date car, même si plus rapide en théorie, les résultats ne sont pas forcément meilleurs, et il vaut mieux avoir un meilleur résultat tant que la durée de calcul n'en pâtit que raisonnablement.

Il existe aussi l'algorithme de Floyd-Warshall généralisé, qui sert à calculer tous les plus court chemins entre tous les noeuds d'un graphe et n'est donc pas adéquat pour ce projet, car stocker tous les chemins reviendrait à stocker bien trop de données pour le projet, car il doit pouvoir se lancer sur un ordinateur qui n'a pas spécialement beaucoup de capacité de calcul.

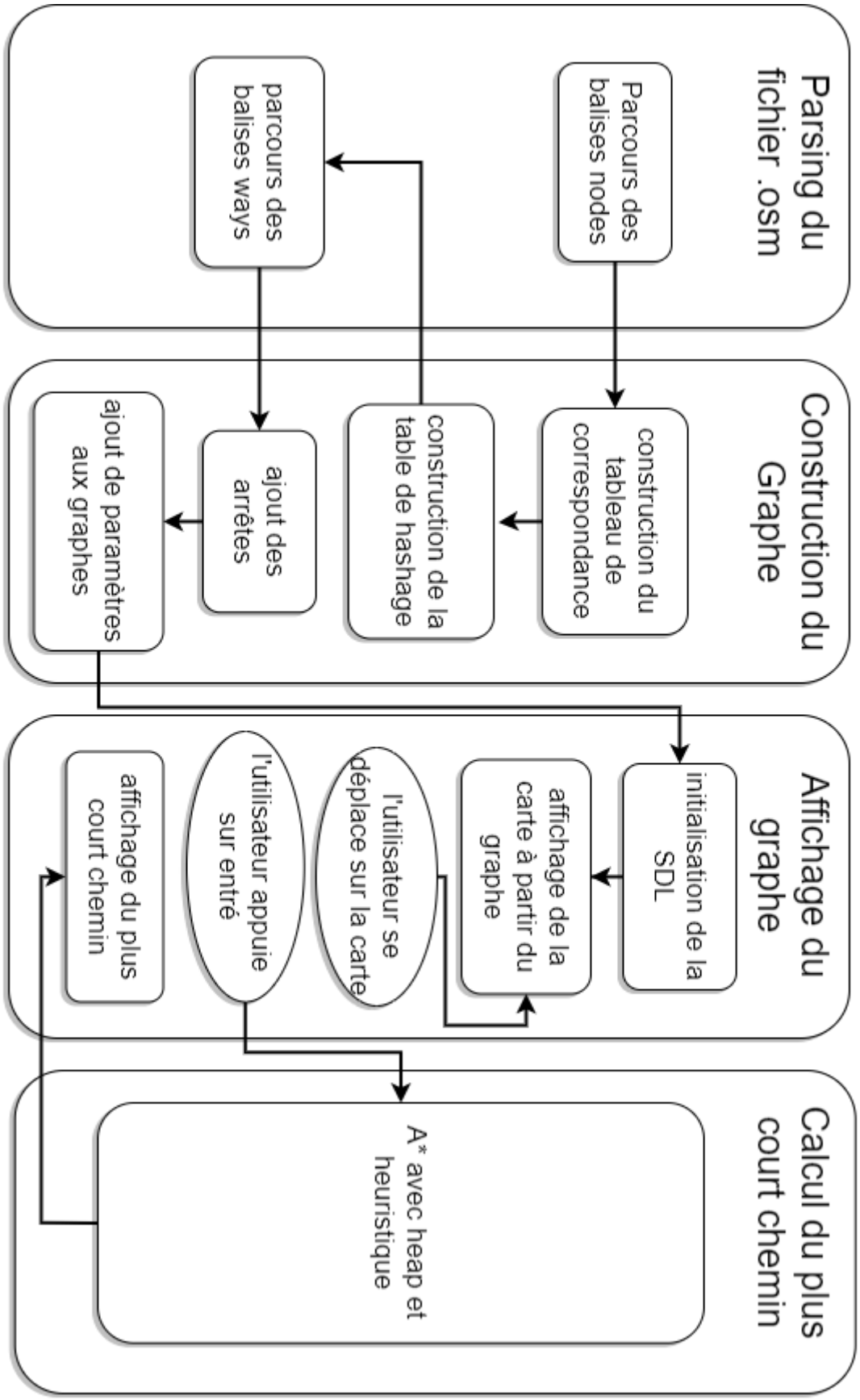
1.3.3. Comparaison au projet

Le projet se différencie de ce qui existe déjà car sa spécificité est qu'il essaie de trouver en chemin en prenant des contraintes differentes de celles que pallient les logiciels déjà existants. Le projet a pour but de trouver un chemin en passant uniquement par des rues éclairées, pour permettre d'être rassuré la nuit et de mieux voir ce qu'il se passe en général. Notre projet est moins précis que d'autres programmes déjà existants, qui sont parfois plus complets mais aucun n'a la spécificité des chemins éclairés. En ce qui concerne les algorithmes déjà existants, nous n'utilisons qu'un simple A* utilisant une heap.

1.4. Présentation du projet

Le programme fonctionne en trois parties, d'abord il récupére une carte sur internet, à partir de cette carte il construit un graphe, puis dessine ce graphe dans une interface graphique, qui appelle des algorithmes de plus court chemins lorsque l'utilisateur veut connaître le plus court chemin entre deux points. Le but et l'essence du projet et de pouvoir donner un chemin qui est éclairé pour aller du point A au point B

1.4.1. Schéma explicatif



2. Fonctionnement du programme

2.1. Récuperation de la carte

Dans cette première partie, le but est de récupérer une carte de Lyon sur internet et de construire un graphe à partir de ce fichier. Le graphe représentera notre carte, les noeuds du graphe sont par exemple des bouts de bâtiments, des bouts de rues, ou même des arbres. Les liaisons de notre graphe vont représenter les rues, les routes et les chemins (mais aussi les bâtiments par exemple en liant ples differents noeuds représentants le batiment.)

2.1.1. OpenStreetMap

FIGURE 5 – Logo de OpenStreetMap.

OpenstreetMap est un site internet qui met à disposition une carte du monde libre d'accès pour n'importe quel utilisateur. A partir de ce site, il est possible de récuperer un fichier en ".osm" qui represente la carte du monde ou de simplement une ville par exemple. Les données sont émises par des utilisateurs du site. Ce site permet d'avoir des données, peut-être moins précises mais gratuites car contrairement à google qui se peut se permettre d'engager des entreprises pour récupérer ces informations, le groupe n'a pas autant de moyens.

2.1.2. Parsing d'un fichier .osm

2.1.2.1. Parcours du fichier

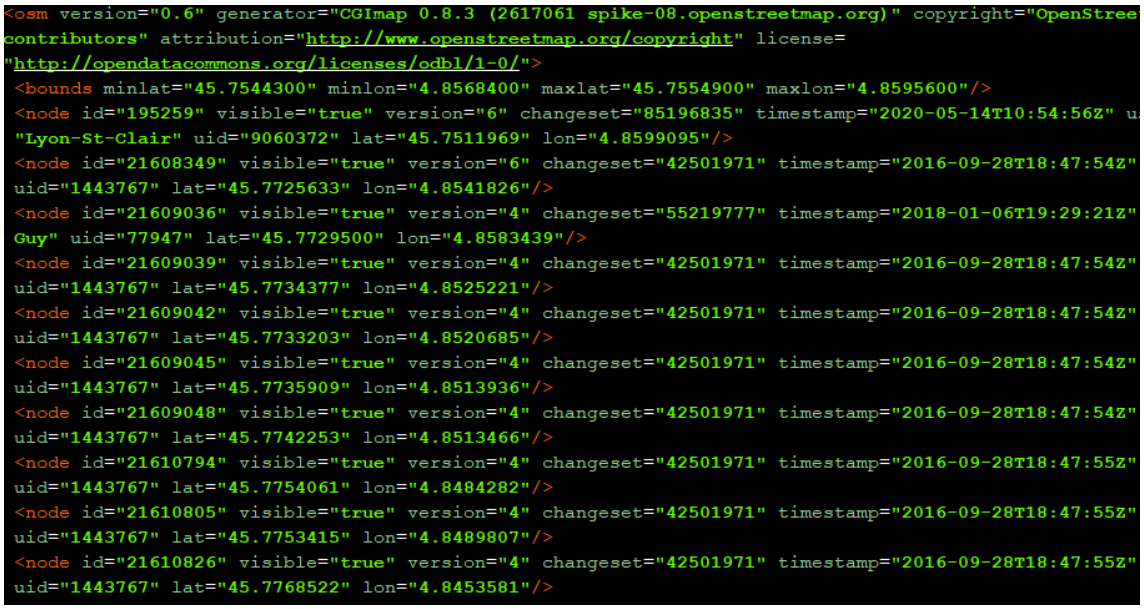


FIGURE 6 – Image de la structure d'un fichier .osm

Comme expliqué précédement, le fichier en ".osm" represente une carte, de Lyon par exemple, il est écrit dans un langage de balisage. Le programme parcours ce fichier caractère par caractère et en extrait des informations pour construire un graphe.

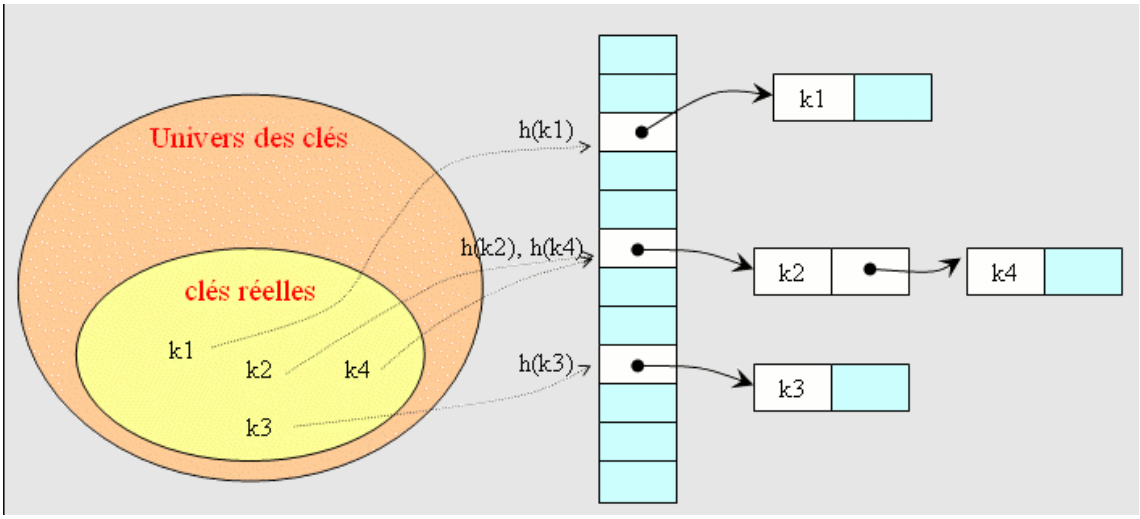
Tout d'abord, le programme execute un premier parcours caractère par caractère du fichier et analyse les balises "nodes" du fichier, chaque node contient des informations comme la latitude et la longitude du noeud, et un id.

Ensuite, le programme reparcours le fichier et analyse les balises "ways" qui représentent les liaisons entre les noeuds du graph. Grâce à ce deuxième parcours, le programme remplie la liste d'adjacence de notre graphe et stocke également d'autres informations comme par exemple : si la liaison représente une rue ou non, si c'est une rue éclairée ou non et le nom de la rue dans le cas où c'est une rue.

Pour faire le parsing, plusieurs méthodes sont utilisées, pour le premier parcours, le programme lit chaque caractère du fichier une seule fois et l'analyse ainsi, cette méthode est très optimisé puisque chaque caractère n'est lu

qu'une seule fois, mais est compliqué à coder et à relire. Pour le deuxième parcours, une certaine partie du fichier est stocké dans une chaine de caractères et des regex sont utilisés dessus, cette méthode est moins optimisée car chaque caractère peut être lu plus d'une fois, mais cela permet d'analyser le texte beaucoup plus simplement. En effet l'utilisation d'expression regulières rend le parsing beaucoup plus intuitif.

2.1.2.2. Table de hashage



Un gros problème s'est révélé lors du parsing. Si par exemple le graphe est de taille order, on considère que chaque noeud à un numéro compris entre 0 et order, le premier noeud aura comme numéro 0, le deuxième 1, jusqu'au dernier qui aura comme numéro order-1 . Cependant, dans le fichier .osm, chaque noeud a un id qui est potentielement très supérieur a order.

Cependant, pendant le parcours des ways dans le fichier, il est nécessaire de savoir à quel numéro de sommet du graphe correspond l'id du noeud (resp. node) dans la way.

Il faut donc un tableau de correspondance entre les numéros de sommet du graphe et l'id du noeud(resp. node) du fichier.

Prenons un exemple avec un graphe de taille 3, les ids des nodes dans le fichier vont être par exemple : 4885, 991058498 et 9974878 Il faut donc simplement faire un tableau de correspondance qui associe a chaque sommet du graphe l'id du node du fichier, comme cela :

Tableau de Correspondance	
0	4885
1	991058498
2	9974878

Cette méthode marche bien sur des petits graphes, mais sur un graphe de plus de quelques centaines de milliers de noeuds cela s'avère être lent, car pour récupérer le numero de sommet d'un id d'un node du fichier il faut parcourir tout le tableau de noeuds.

Le coût pour récupérer le sommet d'un noeud du graphe associé a l'id du node du fichier est de l'ordre de n (la taille du graphe). Avec cette methode, constuire un graphe contenant environt 5 millions de noeuds et environs 7 millions d'edges (comme la ville de Lyon) , prendrait au moins plusieurs dizaines d'heures voire plusieurs jours.

Pour résoudre ce problème, et ainsi pouvoir construire des graphes très grands (de la taille de la ville de Lyon par exemple) en un temps très court (moins de 30 secondes), il a fallu utiliser une table de hachage.

La fonction de hachage utilisée doit convertir des potentiellement grands nombres de manière la plus uniforme possible sur l'intervalle.

$$[0; order[\tag{1}$$

pour ce faire, l'utilisation d'un simple modulo fût suffisant.

$$hash(x) = x\%G.order \tag{2}$$

En reprenant l'exemple suivant, on a :

- hash(4885) = 1
- hash(991058498) = 2
- hash(9974878) = 1

Voici la table de hachage correspondante :

Table de hachage	
0	NULL
1, hash(9974878) ou hash(4885)	2 => 0 => NULL
2, hash(991058498)	1 => NULL

Une fois la table de hachage construite, pour connaître le sommet du graphe correspondant a l'id, on doit parcourir la liste chaînée de tableDeHachage[hash(i)] Si la liste chainée a plusieurs éléments, on determine lequel est le bon car il vérifie tableauDeCorrespondance[listeChainéeX] == i
dans notre exemple, par exemple, pour récupérer le numéro de sommet de l'id du node 4885, on parcourt les éléments de la liste chaînée à l'adresse tableDeHashage[hash(4885)] ;
le premier élément est 2, or tableDeCorrespondance[2] != 4885, le deuxième élément est 0 et tableDeCorresondance[0] == 4885, donc 0 est le numéro de sommet correspondant à l'id du node 4885

2.2. Interface graphique

2.2.1. Utilisation de SDL

FIGURE 7 – Logo de la bibliothèque SDL.

SDL étant une bibliotèque bien documentée sur internet et assez rapide, elle fut utilisée pour ce projet. Cependant, il n'existe pas de fonctions par defaut pour créer des boutons ou des menus, l'affichage du texte à l'écran aurait donc peut être était plus simple avec GTK.

2.2.2. Transcription des coordonnées gps en coordonnées cartésiennes

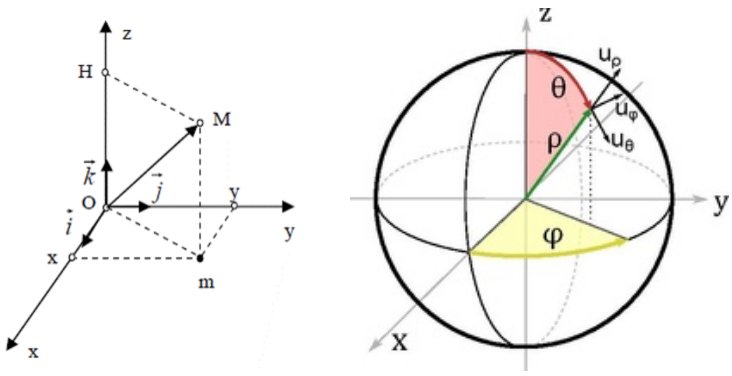


FIGURE 8 – Représentation de coordonnées cartésiennes et sphériques.

Les coordonnées récupérées par le parsing, sont des coordonnées gps, c'est a dire que pour chaque noeud, il y un paramètre latitude et un paramètre longitude. Or, il a fallu transformer ces coordonnées en coordonnées cartésiennes, ce qui correspond aux coordonnées des pixels sur l'écran. La ville de Lyon étant très petite par rapport à la terre, les coordonnées gps furent considéré comme des coordonnées cartésiennes, ce qui est une toute petit approximation a l'echelle d'une ville, cependant cela permet un grand gain en rapidité, car il n'y a pas de calcul d'angle à faire, il faut simplement soustraire la longitude et la latitude un point x a la longitude et la latitude d'un point de reference situé au au milieu de l'ecran, à cela est ajouté la moitié de la largeur de la fenêtre pour la longitude et la moitié de la hauteur de la fenêtre pour la latitude, à cette équation est rajouté un niveau de zoom. Lorsque l'utilisateur se déplace, le programme considère que le point de référence situé au milieu du graphe se déplace, et il recalcule les positions des autres points à partir de ce dernier.

2.2.3. Affichage de la carte

L'affichage de la carte, consiste à afficher toutes les arêtes du graphe aux bonnes coordonnées et de la bonne couleur en fonction de si l'arête représente une rue éclairée ou non (jaune ou gris). Pour ce faire, un parcours des listes d'adjacence de chaque sommet est suffisant, avec un vecteur de marque pour ne pas dessiner plusieurs fois les mêmes arêtes.

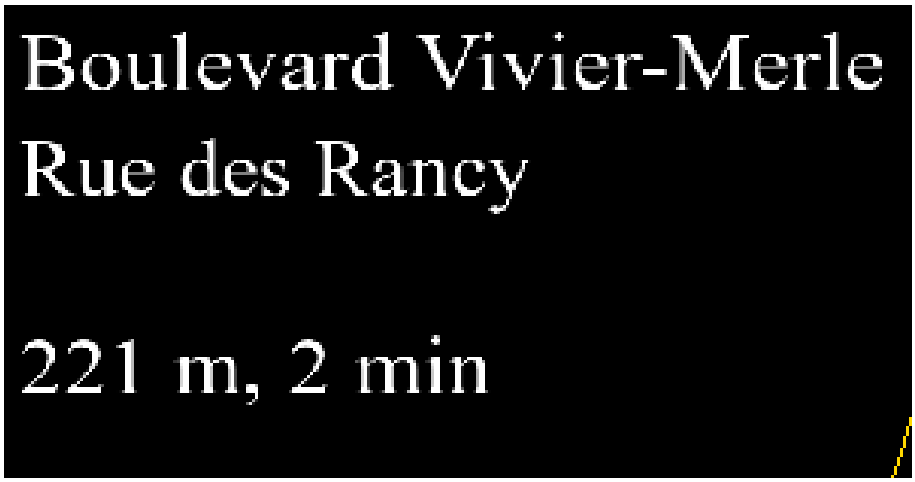
Une grosse optimisation fut de ne dessiner que les arêtes dont au moins l'une des deux extremitées apparaît à l'écran, car bien qu'ils ne se voit pas sur l'interface graphique, essayer de dessiner des traits à des coordonnées indisponibles, par exemple du point (-4, -10) au point (-59 -82), est coûteux.

2.2.4. Récupération d'événements

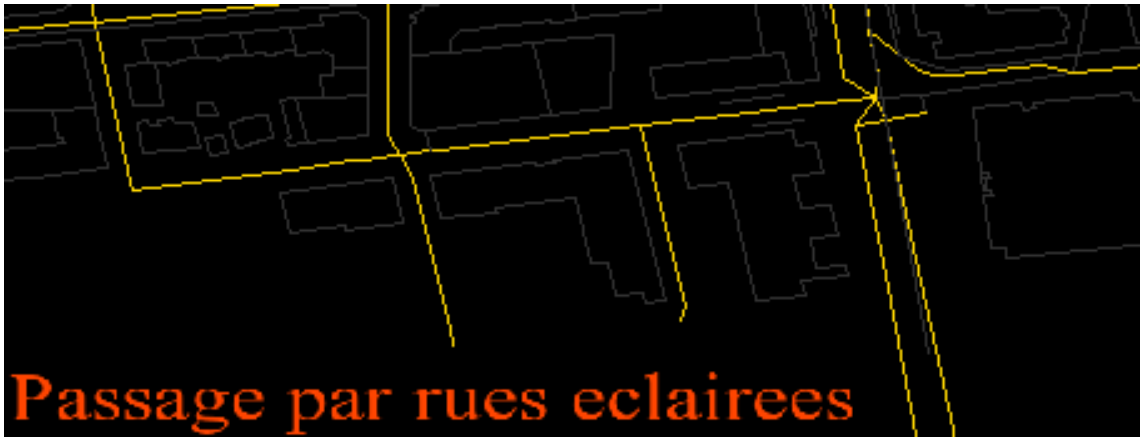
L'interface graphique détecte des événements de la part de l'utilisateur, par exemple, lorsque l'utilisateurs appuie sur les fleches du clavier, la carte est decalée et redessiné. Cela est aussi vrai lorsque l'utilisateur clique sur la carte, le sommet le plus proche est selectionné (ceci est fait en reconvertissant les coordonnées cartésiennes en coordonnées gps).Lorsque l'utilisateur appuie sur la touche entrée, le chemin le plus court est calculé entre les deux noeuds précédement selectionnés

2.2.5. Affichage du texte

Le programme affiche en haut à droite de l'ecran les rues auquelles correspondent les noeuds séléctionnés, Cela est fait grâce à une police d'écriture qui est enregistrée dans le fichier times.ttf et qui est chargée, puis la fonction SDLCreateTextureFromSurface est utilisée pour créer une texture contenant le texte à afficher. Le programme affiche également la distance et le temps qu'il faut pour aller du premier point selectionné au deuxième point séléctionné, comme sur l'exemple ci-dessous



Le programme affiche de la même manière en bas a gauche si oui ou non l'utilisateur veut passer uniquement par les rues éclairées (ce paramètre est modifé quand l'utilisateur appuie sur L).



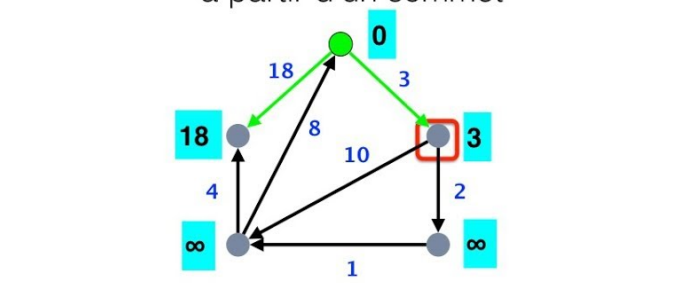
2.3. Calcul du chemin

Pour calculer le chemin le plus court entre 2 points choisis sur une carte, le plan a été de d'abord de faire en premier un algorithme en python avant de l'implémenter en c grâce à l'implémentation en graphe énoncé précédemment. L'implémentation de l'algorithme Dijkstra fût le premier puis l'algorithme A* a été implémenté avant de se concentrer sur l'optimisation du calcul pour avoir notre réponse le plus rapidement possible. Ensuite, il fallait connecter nos algorithmes avec l'interface graphique pour pouvoir faire des tests plus rapidement et vérifier directement sur une carte nos chemins calculés ce qui est bien plus pratique pour tout débugger, et évidemment pour la fin du projet.

Cela a permit de pouvoir grâce à l'interface graphique de vérifier la vitesse de réponse sur différentes cartes même très grandes comme celle de Lyon de manière efficace car nous n'avions plus besoin d'aller dans le code pour changer le départ et la destination mais juste plus qu'à faire 2 cliques et appuyer sur entrer pour voir s'afficher le chemin. C'était bien plus clair qu'une suite de noeuds puis devoir vérifier soi-même sur le graphe à part.

2.3.1. L'algorithme de Dijkstra

C'est la première méthode pour calculer la distance entre 2 points trouvés durant les premières recherches au tout début du projet, après avoir choisi l'implémentation en graphe pour ce projet, le choix s'est donc porté sur cet algorithme. En plus de cela, l'algorithme de Dijkstra, est en fait assez simple a comprendre, car en effet, celui-ci prend en compte les graphes orientés ou non, et les graphes ne possédants pas de coût négatif entre chaque sommet du graphe. En premier lieu, il y a eu l'implémentation Dijkstra en python pour être sûrs de son fonctionnement et de son efficacité en se servant de l'implémentation des graphes fournie dans la bibliothèque algopy en attendant l'implémentation des graphes complètes et fonctionnelles en C.



C'est un algorithme de compléxité logarithmique est a été pendant longtemps l'un des seuls algorithmes cherchant le plus court chemin au sein d'un graph. C'est aussi pour cette raison que qu'il a été implémenté pour le projet, cela nous a semblé être une évidence.

2.3.2. L'algorithme A*

L'algorithme A* ou encore appelé "A étoile" ou "A star", est l'algorithme qui sera utilisé par notre programme afin de trouver le plus court chemin. Cependant, avant de s'accorder sur le fait d'implémenter cet algorithme dans notre projet, le programme disposait déjà de l'algorithme de Dijkstra. Mais pour trouver le plus court chemin entre deux points sur un graphe contenant plusieurs milliers voire millions de noeuds, notre Dijkstra prenait de longues minutes avant de renvoyer un résultat au programme principal. En effet il faut attendre d'avoir relaché la destination.

Malgré, le fait que l'algorithme A* implémenté dans notre programme est bien plus efficace que le Dijkstra ajouté plus tôt durant le projet, il reste très similaire ca celui-ci. En effet A* est a la base inspiré de l'algorithme de Dijkstra. Ce qui fait la principale différence entre ces deux algorithme se fait dans l'ordre de traitement des differents sommets. En effet, A* va avoir besoin d'un vecteur qui sera le résultat d'une évaluation heuristique. Ce vecteur va donc avoir une certaine influence sur l'ordre des noeuds traités.

En plus d'une heuristique, notre A* va stopper les itérations à partir du moment où il rencontre le noeud objectif de notre chemin. En effet grâce, au parcours de notre algorithme, si un noeud est traité dans notre algorithme, cela veut dire que nous ne pourrons pas troueré de chemin plus court que celui deja obtenu.

2.3.2.a. Méthode d'évalutation heuristique

Comme nous l'avons vu, L'algorithme A* nécessite de créer une évalutation heuristique. Cette dernière va aider l'algorithme à choisir les noeuds dans la bonne direction en fonction de l'emplacement du debut et la fin du chemin.

Donc pour creer cette évaluation, nous avons pour chaque point du graphe calculer l'angle entre le vecteur du point en question, et le départ, et le vecteur allant du départ à l'arrivé. En fonction de l'angle, le sommet sera plus ou moin traité rapidement. Par exemple, si notre vecteur (point,départ) possède un angle a 180 degrès avec le vecteur (départ, arrivé), alors notre évaluation heuristique, va faire en sorte que notre algorithme traite notre point le plus tard possible.

La création d'une heuristique est très importante dans l'algorithme, car étant donné que si A* rencontre notre destination, alors le programme de recherche s'arrête. Donc si l'évaluation heuristique est mal faite, alors nous risquons de pas trouver le chemin le plus court, ou même de trouver un chemin completement absurde.

2.3.2.b. Implémentation de l'heuristique

Étant donné que l'algorithme de notre programme traite le noeud ayant la plus petite distance par rapport a lui et qui n'a pas été encore traité. Alors, la méthode d'implémentation a été la suivante :

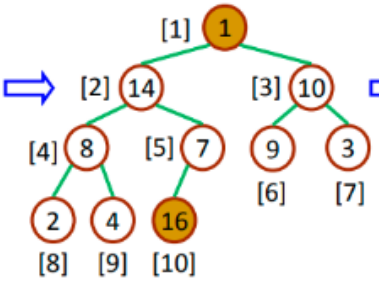
Dans un premier temps le programme créer un vecteur avec n éléments, ce qui correcpond a l'ordre du graphe. Puis pour chaque nous créons le vecteurs avec le début du trajet avec le vecteur de (début et fin). Ainsi, nous sommes maintenant capable d'obtenir un angle pour chaqu'un de ces vecteur par rapport a l'axe des abscisse. Ainsi en soustrayant les angles des vecteur nous avons l'angle de différence de ces derniers.

Grâce a l'angle obtenue précedamment, le programme va executer le calculs suivant sur ce dernier

$$\frac{Angle}{180} <= 1 \tag{3}$$

Cette dernière étape nous permet donc de savoir si un vecteur est donc le bon sens on non par rapport au vecteur (début et arrivé). En effet si la valeur de ce calcul se rapproche de 0, alors nous ajoutons à la heap le numero du sommet en question, avec sont couts multiplier par la valeur obtenue précèdemment. En revanche même procédé lorsque notre valeur se rapproche de 0, cependant ayant un plus grand coût dans la heap, alors que deux noeuds sont à distance équivalente, notre heuristique va leur ajouter du coût afin que notre heap puisse savoir quel élément est le plus important à tester ensuite.

2.3.3. Utilisation d'une Heap



Pour optimiser notre recherche du plus court chemin, après quelques recherches, la création d'une heap à la place de la liste qui était précédemment utilisée a été la solution qui paraissait être la plus logique.

La structure liste est composée de 3 éléments. Le premier est un int, qui permet de stocker la valeur, le deuxième élément, un double, qui permet de stocker le coût du chemin, car il faut trier la liste comme on doit le faire pour A* car les éléments mis dans la liste dans ce contexte doivent ressortir dans l'ordre croissant, et un pointeur qui pointe donc vers le prochain élément de la liste. Cette structure permet aussi de stocker le chemin entre le départ et l'arrivée pour la renvoyer, mais ce n'est pas cette utilisation que va avoir l'heap.

La structure heap va donc servir à stocker et ressortir les éléments dans l'ordre croissant plus rapidement. C'est une structure qui a été créée sous la forme d'un arbre binaire. Elle est composée de 5 éléments. La valeur en int et le coût en double comme la liste, mais avec 2 pointeurs, un pour le fils droit et l'autre pour le fils gauche, et un autre int qui permet de stocker le nombre d'enfants et donc d'équilibrer l'arbre lors de l'ajout d'un élément.

Cette structure utilise 4 fonctions principales, la première qui permet de l'initialiser, la deuxième qui permet d'ajouter un élément (heap update), la troisième qui permet de récupérer la racine de l'arbre qui est donc l'élément qui possède le plus petit coût (heap pop), et la dernière qui renvoie simplement si la heap est vide. Il y a aussi une fonction interne qui permet lors du heap pop de garder l'équilibre de l'arbre binaire car quand on supprime l'élément racine de l'arbre, si on veut faire en sorte que se soit toujours l'élément avec le plus petit coût qui soit racine, il faut donc choisir entre le fils droit et le fils gauche pour être la nouvelle racine mais l'autre fils ne de doit pas disparaître.

Cette structure permet donc lors de l'ajout d'un élément, de ne parcourir que très peu d'éléments et donc de perdre moins de temps si le graph possède des milliers voire des millions de noeuds.

2.3.4. Les rues éclairées

Pour prendre en compte les rues éclairées dans l'algorithme A*, il faut qu'à chaque fois que notre algorithme regarde un nouveau noeud qui pourrait être parcouru lors d'une itération, il regarde dans la liste de la structure graph, si l'élément est éclairé ou non. A* prend en paramètre un int qui, s'il est égal à 0, ne prend en compte que les rues éclairées et s'il est égal à 1 de rechercher dans tous les noeuds de manière équilibrée, et plus ce nombre sera grand, plus la recherche de chemin de plus court prendra en compte les rues non éclairées.

Nous avons donc 3 principaux états de cette variable. Le cas où notre variable vaut 0, ici, qu'importe le coût des arcs des rues non eclairés, ces routes ne seront pas prises en compte. Le cas où nous avons 1, ici, nous chercherons le chemin le plus court, et même si nous passons par des routes non éclairées. Et le dernier cas, où notre variable est différente de 0 et de 1. Dans ce dernier cas, notre programme va faire en sorte de traiter le noeud plus ou moins rapidement en fonction s'il est éclairé ou non. Prenons un exemple, si un noeud se trouve à l'opposé de la fin de notre trajet par rapport a notre point de départ, alors il sera traité bien plus tard que ses camarades étants dans le bon sens.

3. Site

Le site est composé de 5 section :

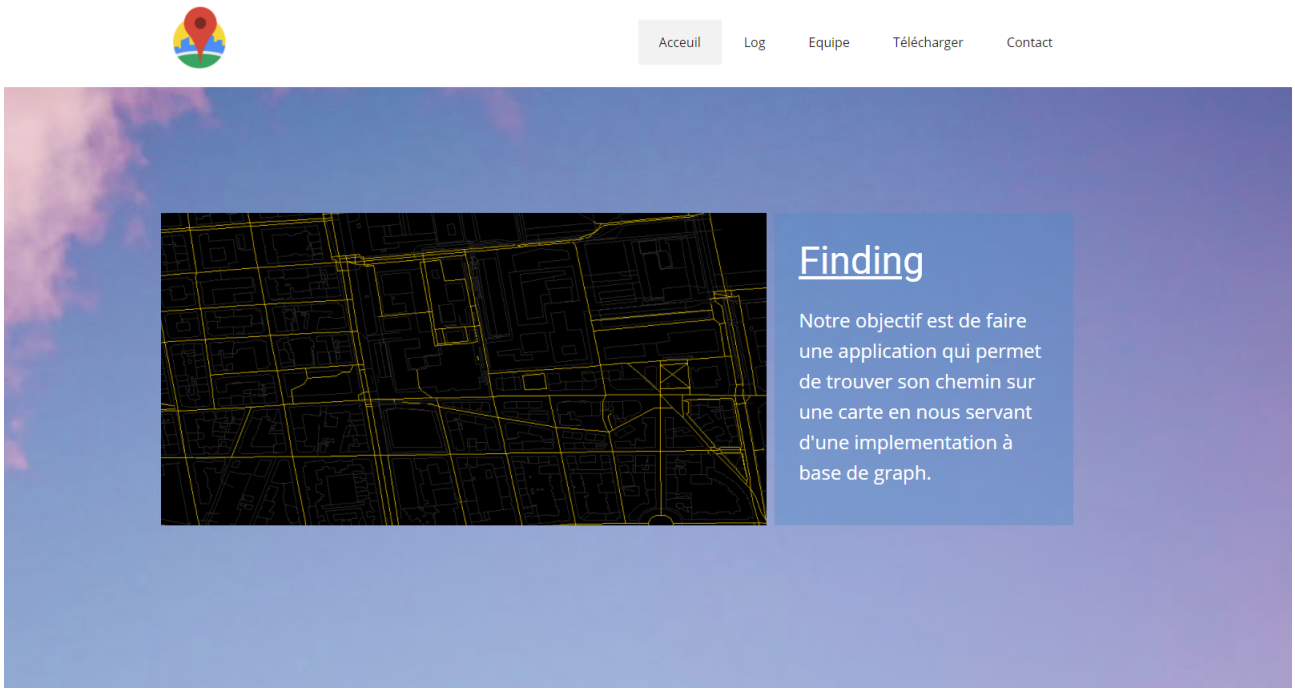
— L'accueil, qui est comme son nom l'indique l'accueil du site et qui fait une description globale du projet.

— Log, qui est une description du projet plus précise et qui présente l'interface graphique.

— Equipe, qui est donc une présentation du groupe.

— Télécharger, qui sert à télécharger le projet et qui sert aussi à télécharger le rapport.

— Contact, qui donne des moyens de nous contacter.



4. Organisation du travail

Pour travailler sur ce projet le groupe a été divisé en deux équipes. Il a été décidé de faire ainsi car c'était le plus simple pour avancer tout en minimisant le temps de debug. Le fait de travailler en duo permet d'augmenter la communication, il suffit d'avoir un représentant de chaque partie pour discuter. Et la discussion de problèmes Algorithmique est bien plus efficace car la discussion se fait entre deux personnes qui ont travaillé sur le même problème. Cela permet également d'avoir plusieurs points de vue sur certains problèmes. C'est egalement plus motivant de pouvoir travailler à deux sur le même code ou à deux en général. Travailler sur le même code fait que l'on est obligé d'arriver ce que l'on a fait sur le code, ce qui est une compétence primordiale de l'ingénieur.

4.1. Environnement de développement

Le fait de travailler en groupe force à utiliser un environnement de développement spécifique. Pour le projet, les differents fichiers qui dépendaient d'autres fichiers qui sont fournis par d'autres personnes ont forcé l'utilisation d'un environnement nous permettant de tous avoir la même chose. Il a donc été décidé de tous télécharger une machine virtuelle. Cela a permis que tout le monde ait les mêmes librairies et que cela ne pose pas de problème de compatibilité.

4.1.1. Git

L'utilisation de GitHub était indispensable car il faillait souvent mettre en commun le code des deux équipes, ce qui est rendu très simple grace à Github. Github nous permet aussi de garder d'anciennes versions di code, lorsqu'il y a un problème, et de garder un historique de ce que l'on a fait.

4.1.2. Gcc

En ce qui concerne le compilateur qu'il a été décidé que nous utiliserions tous pour éviter tout problème de compatibilité, un MakeFile utilisant GCC et permettant de tout compiler a été crée.

4.2. Répartition des tâches

	Florian	Valentin	Maximilien	Julien
Parsing du fichier			Resp.	Resp.
Creation de la table de hachage			Resp.	Resp.
Recupération des propriétés des noeuds			Resp.	Resp.
Creation des structures liées au Graphe			Resp.	Resp.
Creation des liaisons			Resp.	Resp.
Transcription des coordonnées gps en cartesienne			Resp.	Resp.
Interface utilisateur			Resp.	Resp.
Affichage du graphe			Resp.	Resp.
Affichage des noms des rues et des dist			Resp.	Resp.
Implémentation de Dijsktra	Resp.	Resp.		
Implémentation de A*	Resp.	Resp.		
Creation du tas	Resp.	Resp.		
Implémentation de l'heuristique	Resp.	Resp.		
Site	Resp.	Resp.		
Compilation générale			Resp.	Resp.

4.3. Récit de la réalisation

4.3.1. Historique de l'avancement du projet

4.3.1.a. 1 ère Soutenance

Pour cette soutenance beaucoup de recherches ont étés faites, le parsing du texte était bien avancé, le site été commencé et l'algorithme de Dijsktra avait été implémenté en python.

4.3.1.b. 2 ème Soutenance

Pour cette soutenance le parsing du texte et la création du graphe étaient bien avancés, notamment avec l'ajout un vecteur de positions qui contient pour chaque sommet du graphe, sa latitude et sa longitude, le site était bien avancé et l'algorithme de Dijsktra avait été implémenté en C, sur la structure graphe créée précédemment

4.3.1.c. Soutenance finale

Pour cette soutenance le parsing du texte et la création du graphe sont finis, avec beaucoup d'ajout à la structure Graphe, notamment le noms des rues, si les rues sont éclairées où non. L'interface graphique fut entierement créée, avec donc la transcription des coordonnées gps en coordonnées cartesienne, l'affichage du graphe, la rzcuperation d'événement de la part de l'utilisateur, comme la dectection de la touches entré pour calculer le plus court chemin, la detection des fleches du clavier pour deplacer la carte et la redessiner. la récuperation des cliques de l'utilisateur et la détection du noeud le plus proche du clique, l'écriture du nom des rues sur lesquelles l'utilisateur clique. L'ajout de l'algorithme A* est complet avec son optimisation grâce à la heap. Le projet répond donc aux attentes qui étaient fixées au début avec l'ajout du calcul d'itinéraire en ne prenant en compte seulement les rues éclairé.

4.3.2. Peines et joies

4.3.2.a. Problèmes rencontrés

Nous avons rencontré de nombreux problèmes, liés pour certains aux structures qu'on utilisait pour la première fois, dans la SDL par exemple, avec les textes, où on a eu des problèmes à cause des librairies. On a également rencontré beaucoup de problèmes liés à l'allocation de mémoire et les memory leaks en C, qui est sont un vrai casse-tête à corriger. On a également eu beaucoup de problèmes algorithmiques, par exemple pour pouvoir optimiser l'affichage du graphe sur sur la fenêtre, au lieu de dessiner même les rues qui sont en dehors de la fenête on ne dessine que ceux qu'on peut voir. Un autre problème algorithmique que l'on a eu est lors de la création des noeuds via le parsing du .osm, on stockait les id des noeuds dans un tableau de correspondance, cependant pour avoir le noeud du graphe à partir de son id dans le fichier on était obligés de parcourir tout le tableau. On a donc dû réflechir à une méthode pour que ce soit plus optimisé car sur les grands graphes, le programme était beaucoup trop lent. On a donc réalisé une table de hachage pour pouvoir avoir accès au noeud dans le graphe beaucoup plus rapidement. Un autre soucis d'optimisation se présentait avec nos structures de liste chainées, l'ajout à une liste se faisait en parcourant toute la liste, pour l'optimiser on a donc dû modifier la structure de liste pour rajouter un pointeur vers le dernier element de la liste pour avoir un ajout instantané.

4.3.2.b. Joies

Nous sommes allés plus loins que prévu dans ce projet, nous avons implementé toutes les fonctionnalités que nous voulions et même plus, notamment le fait de passer uniquement par les routes éclairées, ou le fait de faire d'énormes optimisations qui nous permettent d'utiliser notre programme sur de lourds fichiers, comme sur le fichier contenant la ville de Lyon entière. La réalisation de ce projet fût difficile, mais nous sommes maintenant très satisfaits et fièrs de notre projet.

5. Conclusion

5.1. Conlusions personnelles

5.1.1. Maximilien CHARMETANT

J'ai passé énormement de temps sur ce projet, et j'ai appris énormement de choses, j'ai le sentiment d'avoir beaucoup progressé en C notamment. J'ai aussi appris a utiliser la SDL, et je me suis amélioré en logique de programmation. J'ai beaucoup aimé travailler en groupe, et j'ai notamment découvert le travail en binôme avec julien que j'ai trouvé super. En plus d'être plus motivant, le fait de travailler à deux nous permet d'aller plus vite car on comprends et trouve souvent les bugs de l'autre. Je me suis rendu compte de l'importance de l'algorithmie dès qu'on a commencé à faire des calculs un grand nombre de fois, par exemple, l'utilisation d'une table de hachage nous a permis de faire aller notre programme des milliers de fois plus vite lors du parsing sur de très grand graphes. Je suis aussi très satisfait de notre résultat final. Je suis fier de notre projet.

5.1.2. Valentin DAUZERE-PERES

Je suis très content d'avoir participé à ce projet, celui-ci m'ayant notamment appris à utiliser le HTML et le CSS en plus de toujours apprendre du travail en groupe qui est l'une de mes raisons personnelles pour lesquelles j'ai choisi Epita au lieu d'une école plus classique en plus de l'omniprésence de l'informatique évidement. Je suis aussi heureux d'avoir participé à ce projet car il m'a paru concret et pas seulement théorique, calculer le plus court chemin dans la ville dans laquelle j'habite m'a paru très gratifiant. Je me sens aussi bien plus à l'aise dans la création et manipulation de structure en C grâce à ce projet car il a demandé la création de différente structure, leurs manipulation et de les debugger nous même évidemment.

5.1.3. Florian LE ROUX

Travailler dans ce groupe et sur ce projet aura été très intéressant pour ma part. Que ce soit au niveau de la programmation en passant par la création du site web, ou simplement par l'implémentation et l'application d'algorithmes. En effet, en plus de nous apprendre comment entretenir un projet, ce dernier nous a plongé dans de nouveaux problemes auxquels nous n'avions pas encore fait face. Je peux donner l'exemple de l'optimisation d'un algorithme ou simplement d'un code. Étant donné que nous avions à traiter le plus de données possible, nous devions implémenter les differents algorithmes où le nombre de données était assez conséquent. J'en conclus que ce projet aura été une expérience très enrichissante, autant dans l'aspect humain que technique.

5.1.4. Julien LUNG YUT FONG

Ce projet a été très enrichissant pour moi, grâce à toutes les heures passées à réorganiser le projet et à travailler dessus. Ce projet m'as permis de m'améliorer en terme de rigueur de travail, de travail en équipe, et de développement personnel en général. Il m'a permis d'en apprendre bien plus sur le C et sur l'algorithmie en général. Le projet m'as permis de me rendre réelement compte de toutes les structures et tout ce qu'on apprend en cours d'algorithmie est très utile et efficace, et que la façon de réfléchir qu'on utilise en algorithmie permet de trouver des solutions à des problèmes réels. La partie du travail en groupe m'as permis de me rendre compte à quel point la lisibilité et la clarté du code était important dans l'explication et la relecture de code en groupe. Pour finir je dirais que je suis vraiment fier de ce qu'on a réussi à faire avec nos petites mains. En terme de code et de réflexion, on a réussi à résoudre des problèmes dont je ne rendais même pas compte de l'existence. Je pense que c'est le projet qui m'as le plus apporté, autant sur le plan des compétences que sur le plan personnel. Je suis donc très heureux d'avoir travaillé sur ce projet.

5.2. Conlusion générale

Pour conclure, Ce projet fut très prenant, et nous en avons tiré beaucoup de choses, nous avons progressé en C. Nous nous sommes aussi rendus compte de la quantité de travail énorme que nous pouvions éxecuter en peu de temps lorsque nous y passions toute la journée dessus, comme pour cette derniere semaine avant la soutenance finale, où nous avons ajouté énormement d'améliorations, non prévues initialement, à notre projet. C'est une certitude que ce projet nous a entrainé pour la suite de nos études et pour notre travail plus tard, car nous nous sommes beaucoup amélioré en programmation et nous avons appris a travailler et à communiquer efficacement en groupe. Globalement, nous sommes très satisfaits et très fièrs de notre projet.

6. Annexes

6.1. Le git

Lien vers le git : https ://github.com/Maximilien22/The_answer_finding

6.2. Bibliographie

— -https ://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra

— -https ://fr.wikipedia.org/wiki/Algorithme_A*

— -https ://en.wikipedia.org/wiki/Incremental_heuristic_search

— -https ://cstheory.stackexchange.com/questions/11855/how-do-the-state-of-the-art-pathfinding-algorithms-for-changing-graphs-d-d-l

— -http ://www.ifaamas.org/Proceedings/aamas2011/papers/A1_B44.pdf

— -https ://www.libsdl.org

— -https ://www.openstreetmap.fr

— -www.google.fr/maps/

— -https ://www.waze.com/fr

— -https ://citymapper.com/

6.3. Autres

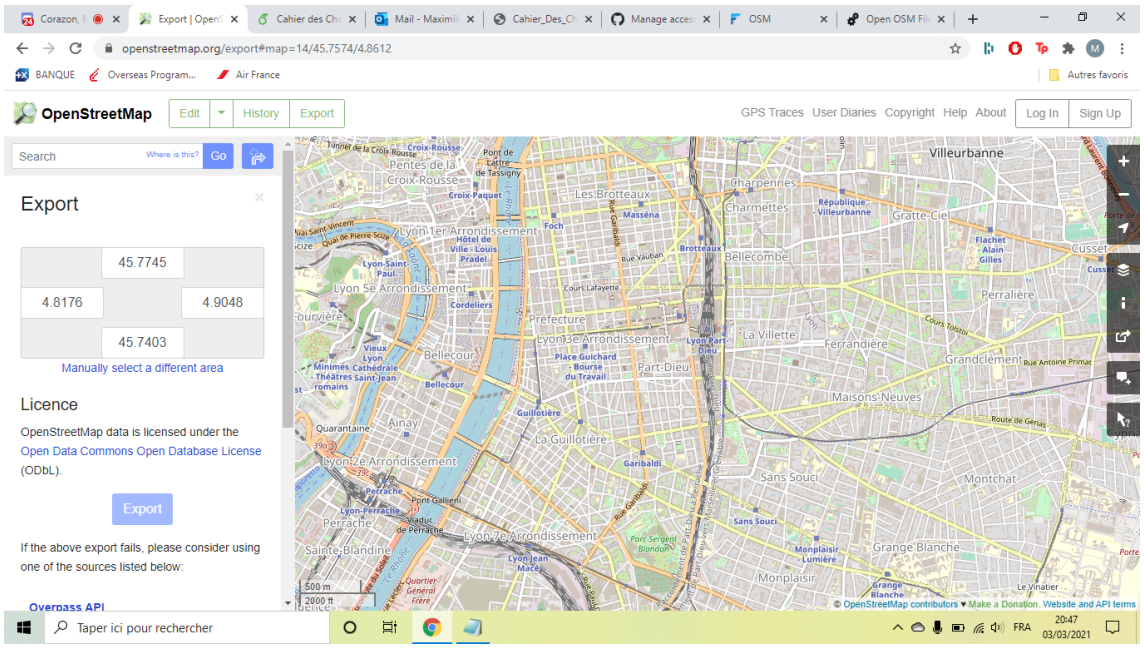


FIGURE 9 – Image du site OpenstreetMap