Brooklyn

Titouan LOCATELLI

Guillaume MERCIER

Hamza OUHMANI

2022

Deuxième rapport de soutenance

Table des matières

1 Introduction

1.1 Présentation du projet

Ce projet consiste à reconduire un Google maps très simplifié. Nous ne travaillons pas sur une carte réelle mais sur une grille (d'où le nom Brooklyn avec la distance Brooklyn facile à calculer), et nous ne sommes pas aussi précis sur les horaires des transports en communs et d'autre détails que nous faisons moins bien que Google... Le résultat final est de sélectionner plusieurs points sur la grille où l'on aimerait se rendre, et que notre programme renvoi l'itinéraire le plus rapide pour se rendre à tout ce point et puis l'afficher a l'utilisateur. Vous pouvez retrouver le projet en téléchargeable sur notre site, et les ressources utilisés pour le réaliser.

1.2 Reprise du cahier des charges

Dans le cahier des charges, nous voulions trouver le plus cours chemin entre deux points dans une carte fixe ou avec deux trois variante tout en ayant une interface graphique. Au final, nous pouvons choisir autant de point que voulu et la carte peut être généré aléatoirement de la taille voulu, l'interface graphique est fonctionnel donc il n'y a pas de retard sur ce qui était prévu, mais que de l'avance.

2 Première soutenance

2.1 Guillaume

2.1.1 Structures et fichiers

La première partie de son travail a été d'implémenter les différentes structures et de décider du format des fichiers. Pour ce qui est des fichiers, nous avons besoin de charger une grille comprenant plusieurs modes de transports sous la forme d'un graph. Après avoir un peu réfléchi et expérimenté une fois le projet un peu avancé, j'ai choisi de représenter notre carte sous le format suivant. Premièrement un fichier comprenant tous les arcs de bases, ainsi que leurs poids, soit les chemins piétons, ce fichier aura pour en tête le nombre de nœuds du graph du graph. Dans le même dossier il y aura un fichier par moyen de transport soit le bus, le métro, le tram et le vélo, ces fichiers ne comprendront que la liste des arcs correspondant au mode de transport ainsi que le poids dudit arc. Ensuite en ce qui concerne les structures de bases : il y en a deux, une pour les graph : graph, et une pour les nœuds : node. node est définie avec les parametres suivant :

— size_t vertex

— struct node* next

— size_t weight

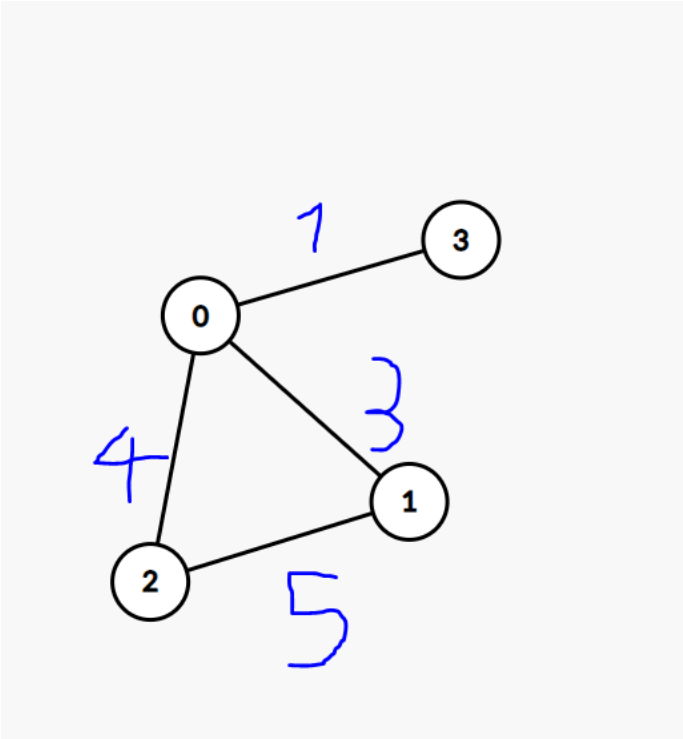
— int transport

vertex est le numéro du nœud dans le graph. next est un pointeur vers le prochain nœud de la liste d'adjacence. weight est le poid de l'arc entre le nœud source et celui ci. Le transport représente le type de transport de l'arc entre le la source et ce nœud. graph est défini avec les paramètres suivants :

— size_t order

— struct node** adjlists

order est bien entendu le nombre de nœuds dans le graph. adjlists est une liste ou l'index i pointe vers une liste chaînée de noeuds représentant les voisins du noeud i. Prenons l'exemple suivant :



Dans le graph ci dessus, la liste chaînée de adjlists[0] comprendra 3 éléments, s'ils sont dans l'ordre numérique, adjlists[0] pointrai vers un élément node dont le vertex est 1 et le poid 3, celui ci pointrai vers un élément node dont le vertex est 2 et le poid 4, ect... Il est important de noter que ces éléments nodes ne sont pas les mêmes pour tout le monde, par exemple, l'élément ayant pour vertex 2 dans les voisins 0 n'est pas le même que celui des voisins de 1, en effet le poid est différent.

2.1.2 Implémentation du premier algorithme(dijkstra)

Après quelque recherche sur les algorithmes de plus court chemin, j'en ai conclu que le plus approprié pour notre projet, au moins dans sa version initiale est l'algorithme de dijkstra, en raison du fait que nous ne cherchions que le plus court chemin d'un point à un autre. Cependant il faudra peut-être considérer changer de l'algorithme quand nous rajouterons plusieurs points de destination, notamment l'algorithme Floyd-Warshall semble plus adapté à ce problème.
La fonction dijkstra est l'implémentation de l'algorithme pour notre problème, le prototype est le suivant :
struct node* dijkstra(struct graph* graph, size_t source, size_t destination) ;
La fonction prend donc en paramètre le graph, le nœud d'origine du chemin à chercher et le nœud d'arrivée.
La première étape est d'enfiler tous les index des nœuds du graph a une file a priorité (voir plus loin pour l'implémentation), tous les nœuds auront initialement une priorité virtuellement infini, sauf la source pour qui elle sera de 0. Il faut également créer une liste "distances" tel que l'index représente le coût du chemin entre l'origine et le noeud i. Et enfin créer une liste "list_prev" tel que l'index i de la liste renvoie le noeud précédent a i dans le plus court chemin.
Ensuite l'algorithme va, tant que la file n'est pas vide, défiler le nœud n avec la priorité la plus basse et parcourir tous ses voisins. Pour chaque voisin sa distance devient le minimum entre sa distance actuelle et la somme de la distance de n plus le poid de l'arc, la priorité du voisin est changé si besoin.
Si le nœud défilé est la destination, alors l'algorithme appelle la fonction de retour puis s'arrête.

2.1.3 Première fonction de retour

La première implémentation de la fonction de retour prenait en paramètre la liste de précédent, la source et la destination. Tant que la destination a un précédent il ajoutait la destination à une liste chainée et la destination devenait son précédent. A la fin il fallait inverser la liste chaînée pour avoir le chemin dans le bon ordre. Cette version était bien entendu sous optimale à cause de l'inversion mais également du fait qu'elle ne travaillait qu'avec les index des nœuds, ainsi les informations telles que le poids ou la méthode de transport étaient perdues.

2.1.4 Tentative d'implémentation du tas de Fibonacci

Après quelques recherche sur comment améliorer l'algorithme de Dijkstra, la première implémentation que j'ai trouvé était d'utiliser une file à priorité (comme dit précédemment) et selon wikipedia l'une des plus performante est l'implémentation passant par le tas de Fibonacci, je me suis donc fixé comme but de l'implémenter. L'implémentation du tas a demandé beaucoup de temps et d'effort car la structure est assez compliquée

à installer correctement. Finalement la structure est presque fonctionnelle mais sur les graphs de grande taille elle crée une erreur mémoire, de plus après de plus ample recherche je me suis rendu compte que ce n'était surement pas la structure la plus adaptée. En premier lieu à cause du coup initial pour créer la pile, il faut initialiser une grande quantité de pointeurs et les fonctions notamment pour consolider le tas après un changement de priorité sont lourdes. Tout cela crée un coup initial très grand qui ne sera compensée que lors de l'utilisation de grand graph. J'ai donc pour l'instant laisser tomber cette implémentation des listes à priorité mais si dans le future j'en ai besoin car la taille des données augmentent, elle est presque opérationnelle.

2.1.5 Implémentation d'une file a priorité

Après l'abandon du tas de Fibonacci je me suis tourné vers une liste à priorité linéaire, il s'agit d'une liste chaînée dont chaque élément contient la priorité, elle est donc triée par ordre croissant. La structure nommée queue_elt contient les attributs

- suivant :
- size_t node
- size_t priority
- struct queue_elt* next

node est l'index du nœud associé, priority la priorité dans la liste et next est un pointer vers le prochain élément de la liste chaînée. Pour la faire fonctionner, plusieurs fonctions sont implémenté, une fonction d'insertion "insert" de prototype :
void insert(struct queue_elt** queue, size_t node, size_t priority) ;
Elle crée un élément de liste qui aura comme priorité "priority" et comme index de node "node", puis elle place cet élément à l'endroit ou le suivant a une priorité supérieure ou égale à la sienne.

Ensuite il y a la fonction extract_min qui renvoie l'index de l'élément avec la plus faible priorité et le supprime de la file.
Enfin il y a la fonction change_prio qui a pour prototype :
void change_prio(struct queue_elt** queue, size_t node, size_t new_prio) ;
Cette fonction va changer la priorité de l'élément ayant comme index de noeud "node", pour cela elle va supprimer le noeud en question puis appeler la fonction insert pour le replacer à la bonne place.

2.1.6 Amélioration fonctions de retour

Comme dit précédemment une fois que l'algorithme de Dijkstra à trouver sa destination il va appeler la fonction de retour, la version de final de celle ci a pour prototype : struct node* build_return(struct graph* graph, size_t source, size_t destination, size_t* list_prev) ;

Elle fonctionne globalement comme la précédente sauf qu'elle crée directement la liste chaînée dans le bon ordre en jouant avec les pointers et qu'elle récupère le poids et la méthode de transport des chemins. Pour cela elle appelle une fonction build_node qui va parcourir les listes d'adjacences pour récupérer les informations demandées. Cela pourrait être très coûteux si notre graph avait beaucoup d'arêtes par nœud mais heureusement ce n'est pas le cas.
Par exemple sur notre graphe principal détaillé plus tard par Titouan, le chemin le plus court du point 0 au point 299, donc du coin en haut à gauche au coin en bas à droite, donné par les fonctions est le suivant

```
Guillaume_tests $ ./main graphs/main/ 0 299

-(-1)->0-(0)->20-(0)->40-(0)->60-(0)->80-(2)->107-(2)->131-(2)->249-(0)->250-(1)->291-(1)->298-(0)->299
total time: 0.003971
```

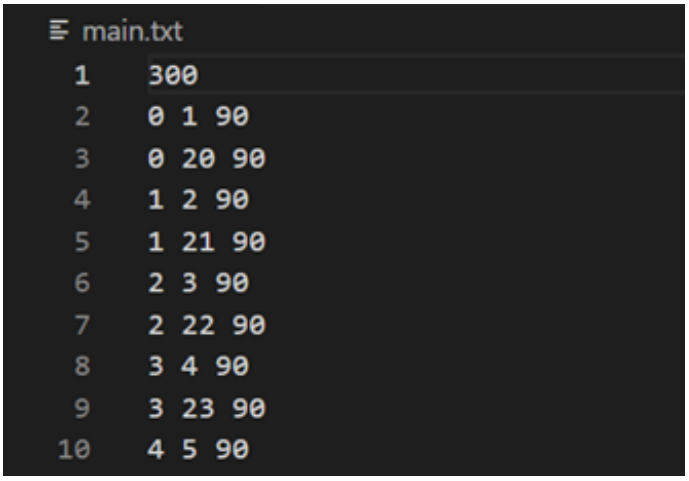
Le temps est donné grâce à la librairie time.h, on peut donc voir que sur un petit graphe comme celui ci, même chercher dans l'un des pire cas est extrêmement rapide. Le chemin rendu donne le mode de transport entre les parenthèses, 0 est pour la marche, 1 le bus et 2 le métro, cela est définie par des macros dans un fichier à part.

2.2 Titouan

J'ai décider de commencer le travail un peu plus tôt pour ce projet, pour éviter le travail dernière minute. Le début était évident, avec Guillaume on s'est divisé le début, il codait la classe graph et je me chargeais de coder le graph initial, nous avons décidé pour une taille initiale de travailler sur un 20*15, soit 300 nœuds.

2.2.1 Graph.txt

On a décider de s'y prendre un peu comme dans les DM d'algo. J'ai donc codé une fonction assez simple qui met dans un fichier texte un graph. Donc nos graphs sont stockés en fichiers texte, manière simple de stockage et rapide à récupérer le graphe de celui-ci.



Première ligne est un int, celui-ci est la quantité de nœuds que notre graphe a. A partir de la deuxième ligne : nous avons les liens entre les nœuds. Le premier chiffre (ex de la deuxième ligne : 0) est l'origine de l'arc et le deuxième chiffre (1) est la destination de l'arc. Nous travaillons avec des graphs pondérés, donc le troisième nombre (90) est le poids de l'arc. En parlant de graphs pondérés : Nous cherchons à représenter plusieurs moyens de transports sur notre carte, car le but et de prendre en compte plusieurs moyens de transports, ou tout simplement marcher, pour ce rendre d'un point A à B. Nous avons donc associé un poids a chaque moyen de transport. Nous les avons choisis en fonction des vitesses de ces transports dans la vie réelle. Ceux cis sont donc :

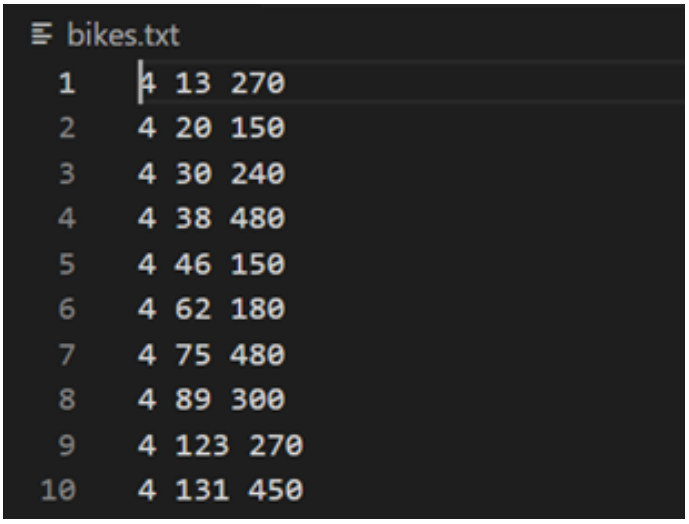
- Walking : 90
- Bike : 30
- Tram : 18
- Bus : 24
- Metro : est calculé différemment.

Tous les autres transports se déplacent sur les routes (c'est-à-dire les nœuds de base de notre grille) de notre grille de 20*15. Donc leurs poids représentent le temps en secondes pour parcourir cent mètres. Donc pour une station de bus a 500 mètres de distance (5 nœuds de distance) de l'autre, le poids de l'arc reliant ces deux stations aura un poids de 5*24 = 120. Le métro, contrairement, ce déplace sous terre. Donc nous avons calculé la distance (pas la distance Brooklyn cette fois) sur notre carte et associé un poids indique a chaque arc reliant deux stations.

J'ai donc procédé à dessiner une carte sur papier de lignes de métro, bus, tram, j'ai placé un paquet de stations de vélos etc... Tout cela pour que la carte est un peu un sens, qu'ils ne soient pas générés au hasard. Nous avons décidé avec Guillaume de faire des fichiers séparés pour les transports. Il y avait donc le fichier main qui était la grille

avec chaque arc avec le poids à pied (=90). Donc tous les fichiers textes des transports étaient séparés du fichier texte main.

En fonction de la carte que j'ai dessiné sur papier. J'ai implémenté à la main dans chaque fichier texte respective, les arcs reliant les moyens de transports. Sauf pour les vélos. Sur ma carte j'avais à peine représenté les stations de vélo. J'ai donc codé une fonction qui prenais le numéro du nœuds et créait un nouveau fichier texte avec tous les arcs entre ce nœuds stations. Nous pensons que c'est la manière la plus simple d'implémenter les vélos pour notre projet. Nous avons médité quelque temps avec Guillaume sur le sujet, nous avions notamment pensé à réutiliser la grille main (celle de marche), mais à rajouter un paramètre d'activation qui diviserais le temps de transports (pour atteindre celle du vélo) jusqu'à atteindre une nouvelle station vélo, mais bon bref, c'était trop compliqué, et probablement beaucoup plus couteux.



Voici en exemple le fichier texte de vélo avec (par ligne) en premier le nœud d'origine, puis le nœud d'arrivé et le poids (proportionnel a la distance entre les deux stations de vélo). 4 est ici le départ sur les dix premières lignes car nous avons liés toutes les stations entre elles, donc il y a énormément d'arcs vélos. Nous pourrions rajouter une contrainte, un peu comme celle de velov, où on a le droit qu'a 30 minutes de déplacement (mais bon, notre carte est trop petite encore pour cela).

2.2.2 Imprimer les resultats sur le terminal.

J'ai pris la tache de l'affichage de notre parcours de graphe avec l'algorithme Dijkstra. Pour la première soutenance je vais afficher ces résultats sur un terminal, puis Hamza prendra le relais et son programme gtk affichera les résultats. Nos graphes ont été implémenter en C, après de la recherche en ligne, les voisins d'un nœud seraient donc mis dans une liste chainée. Et donc, nous avons mis le résultat de notre parcours de Dijkstra sous forme de liste chainé.

Ex : pour aller de nœud 1 a nœud 46 : 1–>2–>6–>5–>45–>46

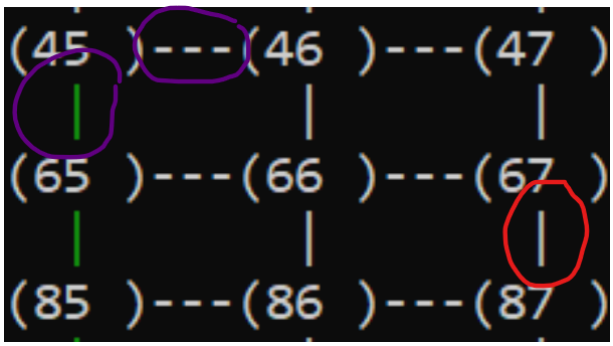
Il va parcourir 5 nœuds avant d'arriver à 45. Je vais d'abord analyser les modes de transport que l'algorithme de Dijkstra a empreinté. Disons que :

— Walking : 0
— Bus = 1
— Metro = 2
— Bike = 3
— Tram = 4

Et je vais voir comment ce déplace donc notre utilisateur du programme avec entre guillemets le mode de transport empreinté durant le chemin :

1—(walk)->2—(bus)->6—(walk)->5—(bike)->45—(walk)->46

Je vais donc chercher à compléter ce chemin (car j'imprime une grille et je n'imprime que les arcs de la grille – pas d'arcs qui vont d'un coin à l'autre de la grille en passant par le milieu ou par l'extérieur. . . peut être sur gtk après) en simulant un peu comme si le bonhomme qui parcourt la grille marchais plus vite ou plus lentement par moments. En effet en dessous par exemple, je n'imprime que les arcs à droite et en dessous de chaque noeud (sauf évidemment pour ceux sur les extrémités du graphe, 67, qui lui a un seul arc (en rouge) qui l'a pour origine).



La liste complétée donne donc :

1—(walk)->2—(bus)->3—(bus)->4—(bus)->5—(bus)->6—(walk)->5—(velo)->25—(velo
>45—(walk)->46

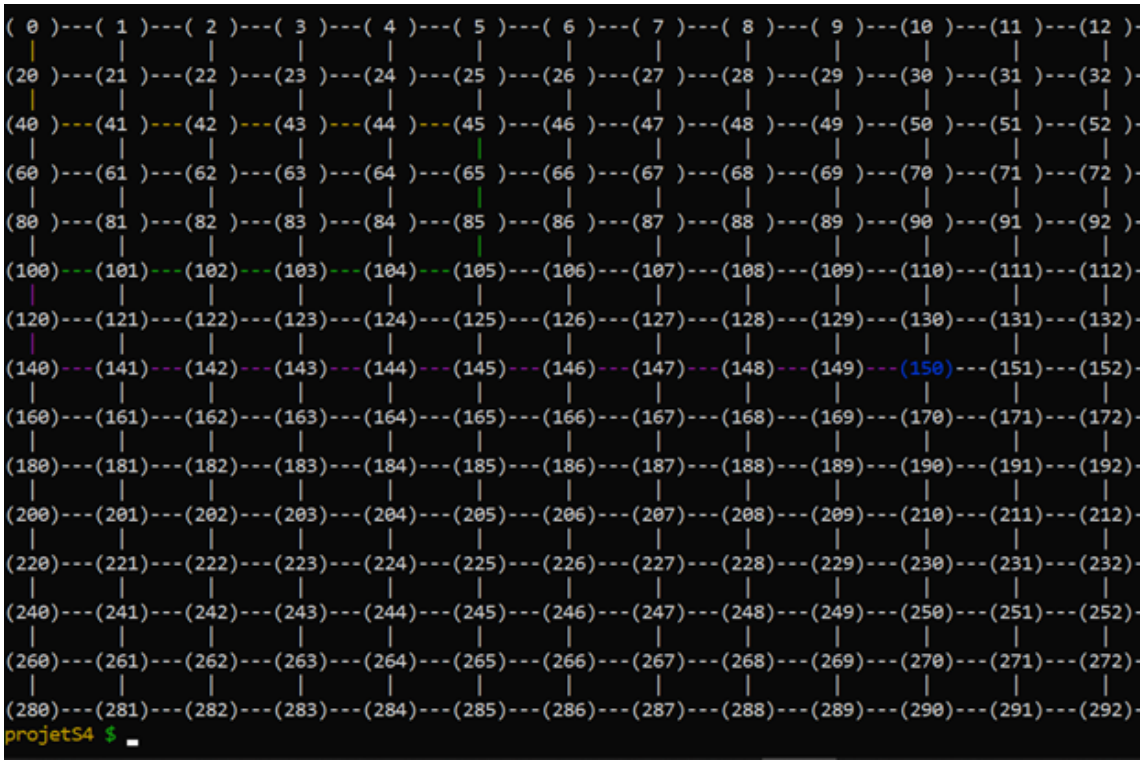
(Je n'ai pas mis l'exemple du métro car celui-ci passe sous terre et je ne vais chercher à compléter son chemin -> j'explique dans la suite)

J'imprime donc la grille dans le terminal en prenant en compte le chemin complété en imprimant les chemins d'une couleur particulière au mode de transport empreinté durant le passage sur ce chemin.

```
 7   #define CWALK   "\x1B[31m" // red
 8   #define CMETRO   "\x1B[34m" // blue
 9   #define CBIKE   "\x1B[33m" // yellow
10   #define CBUS   "\x1B[32m" // green
11   #define CTRAM   "\x1B[35m" // magenta
12   #define CNORMAL   "\x1B[0m" // default
```

Le métro est en bleu (voir ci-dessus) mais je n'imprime pas les chemins souterrains qu'empreinte de métro, je mets plutôt la station de départ et d'arrivé du métro en bleu, c'est-à-dire les nœuds sont imprimés en bleu sur le terminal.

Ci-dessous un exemple d'un extrait de parcours du graph imprimé sur le terminal, le graphe n'est pas affiché en entiers, ça serais un peu dur a voir, mais on visualise bien les chemin avec des couleurs et une station de métro empreinté en bleu.



2.3 Hamza

En ce qui me concerne je me charge principalement de la partie interface graphique et du site web, ce qui implique toutes les intéractions de l'utilisateur avec les différents algorithmes mis en place. Comme il a été expliqué dans le cahier des charges précédemment validé le but ici est de faire l'interface la plus intuitive et élégante possible pour l'utilisateur, pour cela j'ai opté pour la version 3.0 de GTK et Glade, un outil d'interface graphique.

2.3.1 Pourqoi GTK et Glade ?

Gérer la partie interface graphique d'un projet est une absolue nouveauté pour ma part je ne m'étais jamais occupé de cette partie et c'est une chance que j'ai de découvrir cet aspect important dans un projet.

Lors des mes premières recherches sur les différentes possibilités d'une mise en place d'interface graphique en C j'ai eu plusieurs résultats qui me renvoyaient vers GTK et étant donné que c'était une bibliothèque que nous avions déjà utilisé lors du précédent semestre mon choix s'est naturellement porté sur celle-ci.

Glade quand à lui est un outil qui permet de prendre en charge toute la partie de génération et de gestion de l'interface pour permettre de se concentrer uniquements

sur les notions essentielles.

2.3.2 Prise en main et lancement

C'est ici que vont commencer pour moi les déboires de l'interface graphique, lors de l'installation et de l'utilisation de la bibliothèque GTK et de Glade. En ce qui concernait le TP du dernier semestre je l'avais fait sur les ordinateurs de l'école ce qui n'avait pas posé de problèmes alors j'ai tenté d'installer GTK ainsi que Glade sur la machine virtuelle que j'utilise régulièrement (Kali), ce qui s'est soldé par un premier échec.

N'arrivant pas à installer mes outils sous Linux je me suis redirigé vers Windows en pensant que la tâche serait plus simple, avec du recul j'ai été naïf. Pour opérer sur Windows j'ai d'abord installé Mysys2 qui permet de fournir un environnement similaire à Unix pour Windows.

Arrivé à cette étape, 2 choix m'étaient présentés, soit passer par CodeBlocks qui est un envrionnement de développement intégré libre et multi-plateforme qui se caractérise par les multiples possibilités de configuration qui auraient pu me permettre d'utiliser efficacement la librairie GTK ou de choisir Glade. Dans un premier temps j'ai d'abord tenté d'installer CodeBlocks mais ça n'a pas été fructueux. Je me suis alors tourné vers Glade qui était ma solution de départ mais malheureusement malgré le bon lancement de l'outil tout le travail effectué est à chaque fois irrécupérable ou corrompu voir même subi des crash pendant le développement.

J'ai tenté d'installer une version d'Ubuntu sur Windows qui me permettrait de pouvoir profiter de l'environnement Linux (pas sous format machine virtuelle) sur Windows afin de directement pouvoir travailler dessus mais cette tentative aura été sans succès aussi. J'aurai compris que mon entêtement à vouloir essayer de trouver la solution tout seul n'aura servi à rien et n'aura fait que retarder un aspect du projet pour notre groupe, il faudra que j'apprenne à demander plus souvent de l'aide notamment aux ASM qui sont à notre disposition.

2.3.3 Site Web

Comme je l'ai dit précédemment ce projet est rempli de nouveautés pour moi car c'est aussi la première fois que je m'occupe de créer un site web qui permettra de représenter l'état d'avancement de notre et l'évolution de notre travail. Mon choix s'est logiquement

porté sur l'HTML afin de gérer cette partie-là étant donné la quantité importante de documentation disponible sur internet.

Contrairement à l'interface graphique où l'installation et le lancement des différents outils choisis a été un cauchemar, cette étape en ce qui concerne le site web aura été un rêve étant donné qu'il était possible de gérer directement sur Bloc-Notes pour le début puis de passer à l'éditeur SublimText afin d'avoir plus de couleurs et d'options disponibles lors du codage.

Etant donné que j'ai malheureusement accordé trop de temps sur mon temps de projet à tenter d'installer les outils de l'interface graphique je n'ai pas pu pour l'instant pleinement prendre conscience de toutes les capacités qu'offrent l'HTML c'est pour cela que je me suis dans un premier temps concentré sur l'édition de texte dans un premier temps dans l'HTML. C'est pour cela que pour l'instant notre site web correspond actuellement à une tentative de reprise dans le même format d'édition de notre rapport de soutenance.

3 Deuxième soutenance

3.1 Guillaume

3.1.1 Reprise du projet

J'ai personnellement commencé à retravailler sur le projet un mois après la première soutenance, ce n'est pas assez long pour tout oublier. Cependant, il y a quand même eu une période de relecture pour re-comprendre ce que j'avais codé, et ce n'était pas vraiment agréable. Certaines parties du code étaient hasardeuses et pas très organisé, mais j'ai vite recommencer a coder.

3.1.2 Remplacer la file à priorité

Lors de mes tests sur un graphe de 1000*1000, qui semble être un objectif raisonnable, je me suis rendu compte que mon implémentation de l'algorithme de Dijkstra était vraiment lente. J'ai bien relu le code est l'algorithme me semble raisonnablement bien implémenté, le problème venait donc des structures de données. J'ai donc relancé mes recherches de structures de files de priorités, la complexité de ces structures ce calcul sur 5 opérations : trouver le minimum, supprimer le minimum de la file, réduire la priorité d'un élément, insérer un nœud dans la file et enfin fusionner deux files. J'ai donc réuni la complexité annoncée sur wikipédia des plus importants dans ce tableau.

Structures	Trouver mini	Supprimer mini	Inserer noued	Réduire prio	Fusionner
File ordoné	$O(1)$	$O(1)$	$O(n)$	$O(n^2)$	$O(n)$
Tas binaire	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Tas Fibonacci	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Bien sûr, dans la pratique, toutes ses informations ne sont pas importantes, par exemple, je n'utiliserai jamais l'opération fusion donc sa complexité n'est pas importante. De même, la différence entre " O " et " Θ " et assez peu importante et sont souvent confondu, l'idée ici n'est pas d'être précis, mais de déterminer la structure la plus adaptée. La complexité utilisant la file ordonnée n'était pas explicité, et c'est qu'après y avoir réfléchis que je me suis rendu compte d'à quel point elle était désastreuse.
Finalement, j'ai choisi le tas binaire pour deux raisons : premièrement, il est simple à implémenter donc beaucoup plus facile à optimiser, deuxièmement, la complexité du tas de Fibonacci contient une importante constante donc la différence n'est pas très significative pour les données utilisées.

```c
struct bin_elt
{
    size_t key;
    size_t value;
};

struct bin_heap
{
    struct bin_elt* arr;
    size_t capacity; //max nb elt
    size_t heap_size; //current nb elt
    size_t* map; //at map[v] is the index of v in arr
};
```

— L'implémentation est assez directe, la structure contient 4 valeurs :

— arr : une liste de couples (priorité, valeur)

— capacity : nombre maximum d'éléments aloués

— heap_size : nombre actuel d'éléments

— map : une liste pour passer la valeur a son index dans arr : map[v] contient l'index de v dans arr

les règles de base sont simple, pour un noeud à l'index i dans arr : le fils gauche est à l'indice 2*i+1 et le fils droit 2*1+2. Pour ce qui est des opérations :
Le minimum est à l'indice 0 donc facile à obtenir.
Pour supprimer le minimum, je diminum heap_size de 1 puis j'échange l'élément 0 avec celui d'indice heap_size, ensuite il suffit d'inverser pour le faire redescendre à sa

place.
Pour insérer, je mets l'élément à la fin et je le fais remonter jusqu'à ce qu'il soit à sa place, en réalité cela est instantané car le seul moment où un élément est inséré est l'initialisation où tous les éléments ont la même valeur.
Enfin pour diminuer la priorité d'une valeur je la mets à jour et je la fais remonter au bon endroit, cela est possible grâce à la liste map qui permet d'avoir rapidement la priorité d'une valeur donnée.

Cette implémentation n'était pas instantané, bien qu'il existe beaucoup d'aide en ligne, choisir le plus adapté n'est pas évident et a demandé beaucoup de temps, je suis particulièrement fière de la liste map qui est une de mes idées. Je pense que je suis proche de la limite pour ce qui de cette structure, si une plus grande vitesse est requise, il faudra changer de structure, mais je ne pense pas que cela soit nécessaire..

3.1.3 Problème du voyageur de commerce : choix de l'algorithme

La plus grosse fonctionnalité que nous avions prévue pour le projet est de pouvoir choisir plusieurs destinations et de laisser le programme décider de l'ordre le plus rapide, et il se trouve que cela correspond presque exactement au problème du voyageur de commerce.
Titouan était occupé à générer la carte, j'ai ainsi décidé de m'attaquer au problème. J'ai commencé par faire des recherches, la documentation est gigantesque, car le problème est célèbre et les recherches ont donc pris beaucoup de temps. Le problème du voyageur de commerce est le suivant : on part d'un graphe connexe, l'idée est de décider le plus court chemin parcourant tous les nœuds du graphe et de revenir au point de départ. On voit qu'il s'agit d'un problème très proche du notre, mais il y aura quand même un petit ajustement à faire une fois qu'une implémentation sera proposé.

La première approche est évidemment le brut force, il faudrait donc tester toutes les possibilités ce qui revient à tester n ! chemin, cela est impossible. C'est un problème NP-difficile, il existe donc un algorithme ayant une complexité polynomial qui le résou, mais aucun n'a était trouvé. Pour choisir un algorithme, il faut connaître la quantité de points dans notre graphe, vu notre projet, je me suis dit que 20 sommets/destinations étaient un nombre acceptable, c'est un nombre relativement faible et donc la rapidité n'était pas ma priorité.

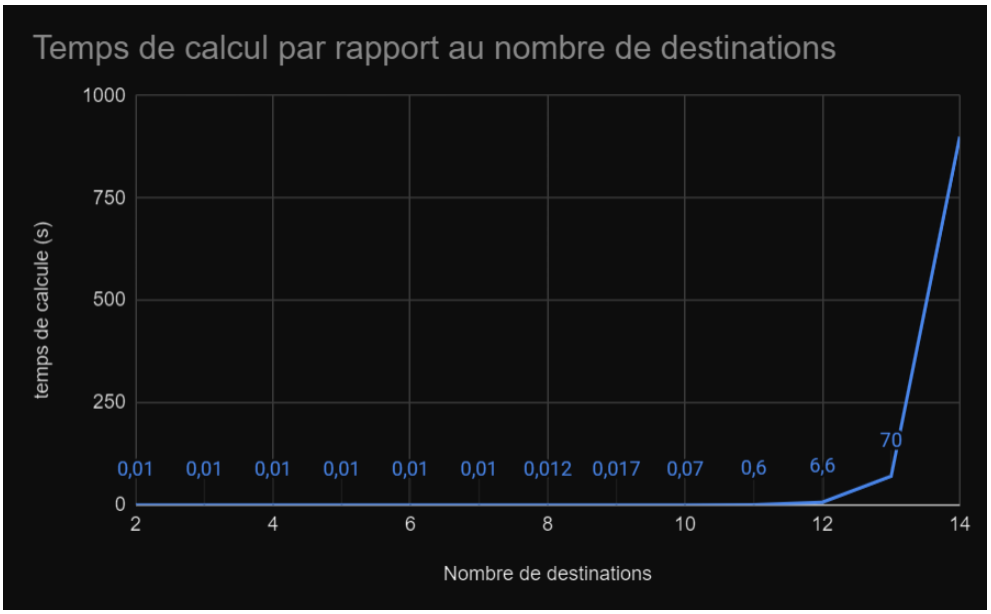
3.1.4 Essaie de l'implémentation de Held-Karp

J'ai donc choisi l'algorithme de Held-Karp car c'est un algorithme exacte qui marche dans tous les cas mais dont la complexité est $O(n^2 2^n)$. L'idée derriere est assez simple :

si le chemin 1->2->3->4 est plus cours que le chemin 1->3->2->4 alors 1->2->3->4->5 est plus cours que 1->3->2->4->5. De la même facon, si le chemin de 1 a 5 traverssant {2, 3, 4} est 1->4->3->2->5 et que, le chemin de 1 a 6 traverssant {2, 3, 4, 5} se termine par 5->6 alors le chemin de 1 à 6 est 1->4->3->2->5->6.

La première étape est de générer les données nécessaire à l'algorithme pour fonctionner, il a besoin d'un graphe complet ! Il a donc fallu le créer, comme il est complet, j'ai choisi de le représenter sous forme de matrice pour simplifier le code. Seul les villes désignés comme destinations m'interraissent, elles seront donc les sommets de mon graphe, et pour le poids des arc il s'agira de la distance renvoyer par l'algorithme de Dijkstra. Si n est le nombre de destinations à parcourir il faudra faire n Dijkstra. Dans un même temps, je sauvegarde les vecteurs pères renvoyés pour reconstruire le chemin plus tard.

Pour l'algorithme j'ai fait beaucoup de recherche puis j'ai essayé de l'implémenter et j'ai obtenu une version fonctionnelle. Cependant, après relecture et plusieurs tests d'efficacité, je pense que ma version ne profite pas du tout des particularités de l'algorithme, car elle est beaucoup trop longue. Ci-dessous le temps en fonction du nombre de nœuds.



La courbe montre en effet le manque de performance de l'algorithme, malheureusement, le temps me manque et cette version devra suffir pour cette soutenance.

3.1.5 Adaptation au problème

Bien que l'algorithme soit très lent, il fonctionne, mais il fallait l'adapter à notre problème et pour cela, il a fallait supprimer le caractère cyclique de la valeur de retour. La solution que j'ai trouvée est de créer, un faut point qui est à une distance de 0 de

tous les autres, il suffira ensuite d'enlever le premier et le dernier élément du chemin de retour. Le seul désavantage de cette méthode, c'est que cela augmente le temps de calcul de manière significative.
Cependant cette technique à créer un nouveau problème, je ne pouvais plus choisir le point de départ, cette fois-ci l'astuce a été de changer la distance de ce faux point. Pour sortir de ce point le coup est très grand sauf pour le point de départ voulu, de cette façon l'algorithme choisira lui-même un chemin commençant par le point voulu.

3.1.6 Multithreading

Afin de rentre le programme plus rapide et grâce aux tps j'ai eu l'idée d'implémenter du multithreading. En théorie, le processus est simple, mais il crée beaucoup de petits problèmes au niveau des données. Globalment c'était assez direct, au lieu d'appeler Dijkstra n fois de suite, je crée n threads qui vont chacun calculer leur Dijkstra. J'ai toutefois eu pas mal de problème de mémoire, car il enregistrait au même endroit, problème classique en C, mais il m'a tout de même pris un certain temps à résoudre. J'ai considéré multithreader l'algorithme pour le voyageur de commerce, mais cet algorithme est temporaire donc cela ne vaut pas la peine.

3.1.7 Valgrind

J'ai eu la joie de découvrir valgrind, en effet, des fois mon programme se fait tuer par l'ordinateur, après avoir regardé ce qu'il se passait avec "htop" j'ai observé avec surprise que ma RAM se remplissait lentement, mais sûrement lorsque le programme tournait. Je me suis alors souvenu des conseils avisés du Grand Vizir Suprême : utiliser valgrind pour les problèmes de mémoires. Après 2 bonnes heures et 46 problèmes résolus, le code est maintenant 100% leak free !

3.2 Titouan

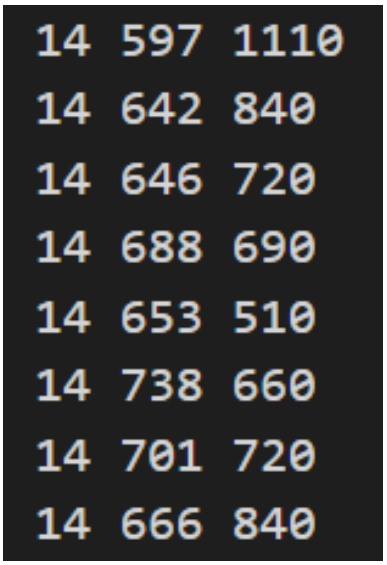
3.2.1 Génération de carte

Pour cette soutenance, je me suis donc concentré sur la génération de graphes et donc surtout de la génération aléatoire des modes de transports.

3.2.2 Vélos

Après une pause de 2 bonnes semaines sans toucher au projet, je commence par la génération des vélos. J'avais déjà une fonction qui liait toutes stations de vélos entre elles et qui stockaient ces liaisons dans un fichier avec le format ci-dessous a chaque ligne :

paragraphStation de départ | Station d'arrivé | Cout de la liaison

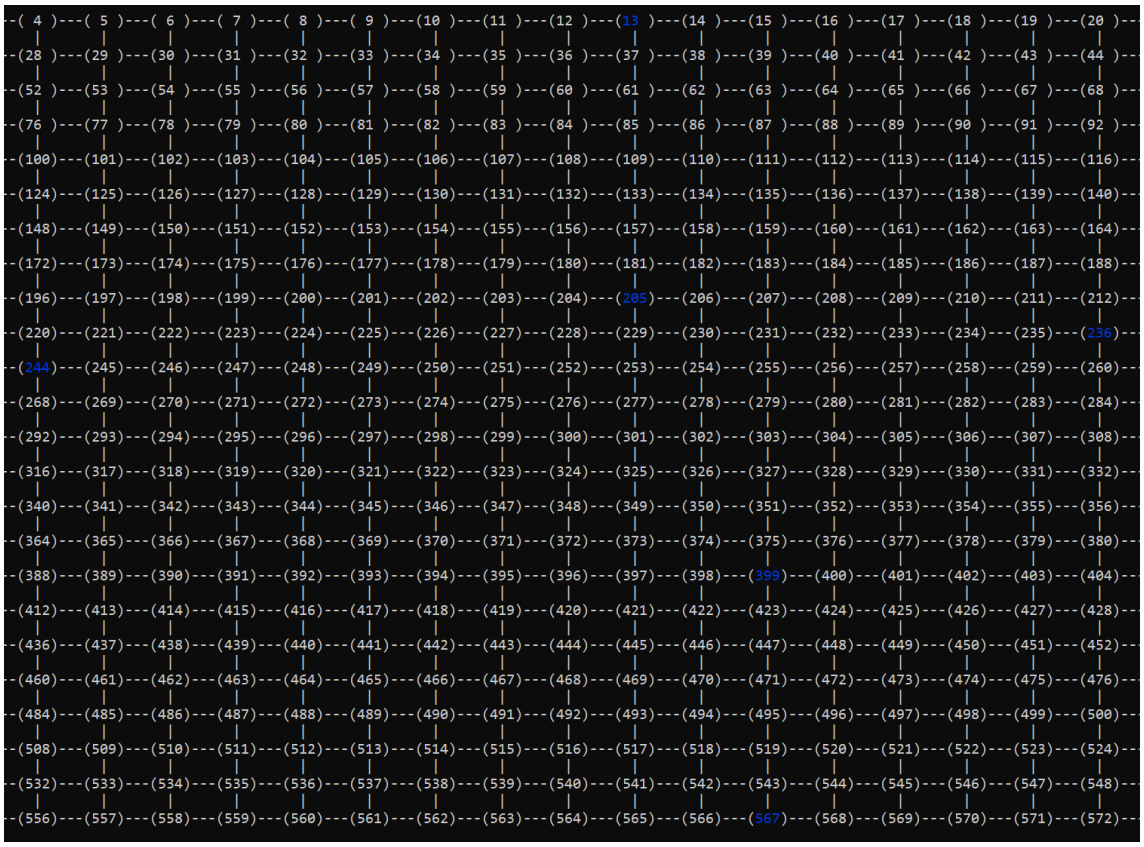


J'en ai parlé avec Guillaume et nous avons décidé de lier tous les vélos entre eux, ça paraissait plus simple. Guillaume n'a donc qu'à récupérer ce fichier avec créer les arcs dans le graphe avec le coût indiqué. L'idée était donc à peu près de diviser le graphe en plein de blocs de taille 5*5, voir 4*4 pour graphes plus petits, et de répartir un peu aléatoirement un vélo par case. Je fais aussi attention à ne pas avoir 2 stations de vélos l'une a coté de l'autre, bien que cela soit possible en vraie vie, dans des lieux clés d'un centre-ville par exemple.

3.2.3 Métro

Générer les vélos étaient assez faciles, j'attaque les métros une semaine plus tard. L'idée était de faire des lignes horizontales et verticales. Je détermine la quantité de lignes par rapport à la taille du graphe et une pincée de rand(). Mes lignes verticales partent du haut, d'un point qui est choisi au hasard dans un carré 5*5 a peu dans le meme style que quand je choisis un vélo au hasard. Il descend de 5 plus ou moins jusqu'à arriver au bord bas du graphe. Il zigzag un peu, grâce a des variations dans le paramètre horizontal (comme par hasard, par hasard), tout en prenant en compte les bords du graphe. Les lignes horizontales fonctionnent de la même manière, mais elles vont de gauche à droite. J'ai ensuite codé une fonction pour réunir certaines stations. Je code les lignes de métro verticales en premier, et le but sera donc que si une ligne de métro horizontale croise une ligne verticale et que leur station sont assez proches

l'une de l'autre (ici, si l'une est dans le carrée de 3*3 qui entoure l'autre), je réunis les 2 stations en une. Cela permet de ne pas avoir plein de stations éparpillés de partout et ça ressemble beaucoup plus à la vraie vie. C'est très peu possible que ça arrive sur deux lignes verticales, mais si avec les variations horizontales, elles se rapprochent, j'ai donc rajouté la fonction pour réunir les stations, on voit beaucoup d'occurrences de cela à Paris, donc pourquoi pas l'implémenter.



3.2.4 Tram

J'ai fait les lignes de trams un peu avec le même style que les lignes de métro, lignes horizontales et verticales avec regroupement. Mais les lignes sont légèrement plus courtes et il y en plus. Je fais le regroupement sur un carré 2*2 plutôt que 3*3 et je fais avancer le tram de 4 plus ou moins plutôt que 5 entre chaque station.

3.2.5 Bus

Le mode de transport le plus compliqué a implémenté, j'ai voulu m'appliquer. L'idée était celle-ci : je voulais que mon bus commence à un point totalement au hasard sur le graphe (plus tard, je pense que je vais faire une fonction pour que les débuts de ces stations ne spawnent pas trop proche l'une de l'autre.), puis que celui-ci se déplace dans toutes les directions (nord, est, ouest et sud) avant de finalement se ramener au point de départ. La ligne de bus boucle donc.

En détail : à partir du point choisi au hasard, j'avance dans une direction de plus ou moins 3 nœuds dans cette direction et possiblement de 1 dans les directions normale a la direction initiale (comme par hasard, encore une fois, mais donc pas par hasard, par hasard). Le bus va continuer à avancer jusqu'à soit arriver vers le bord, ou si une condition que je donne par :

$$\text{if (rand()\%(distance jusqu'au bord / 5) == 0)}$$

Je divise par 5 pour être sûr que le bus ne dépassera jamais les limites du graphe, car chaque mouvement dans une direction ne dépasse jamais 5 de longueur. Cela fait donc en sorte que plus il s'approche du bord, plus il aura tendance à changer de direction. Evidement, je prends en compte les bords du graphe, s'il va vers le nord, mais est trop proche du bord gauche, il ne pourra pas aller vers l'ouest, et donc ira vers l'est s'il ne l'a pas déjà fait avant. Le but étant que le bus aille dans les 4 directions qu'une seule fois. S'il a déjà visité 3 directions, mais les bords du graphe l'empêchent d'aller dans la quatrième direction, peu importe, je marque comme s'il avait fait les 4 directions.

Une fois que j'ai parcouru les 4 directions, je me déplace dans la dernière direction soit jusqu'à arriver vers le bord du graphe, soit si la condition suivante est vraie :

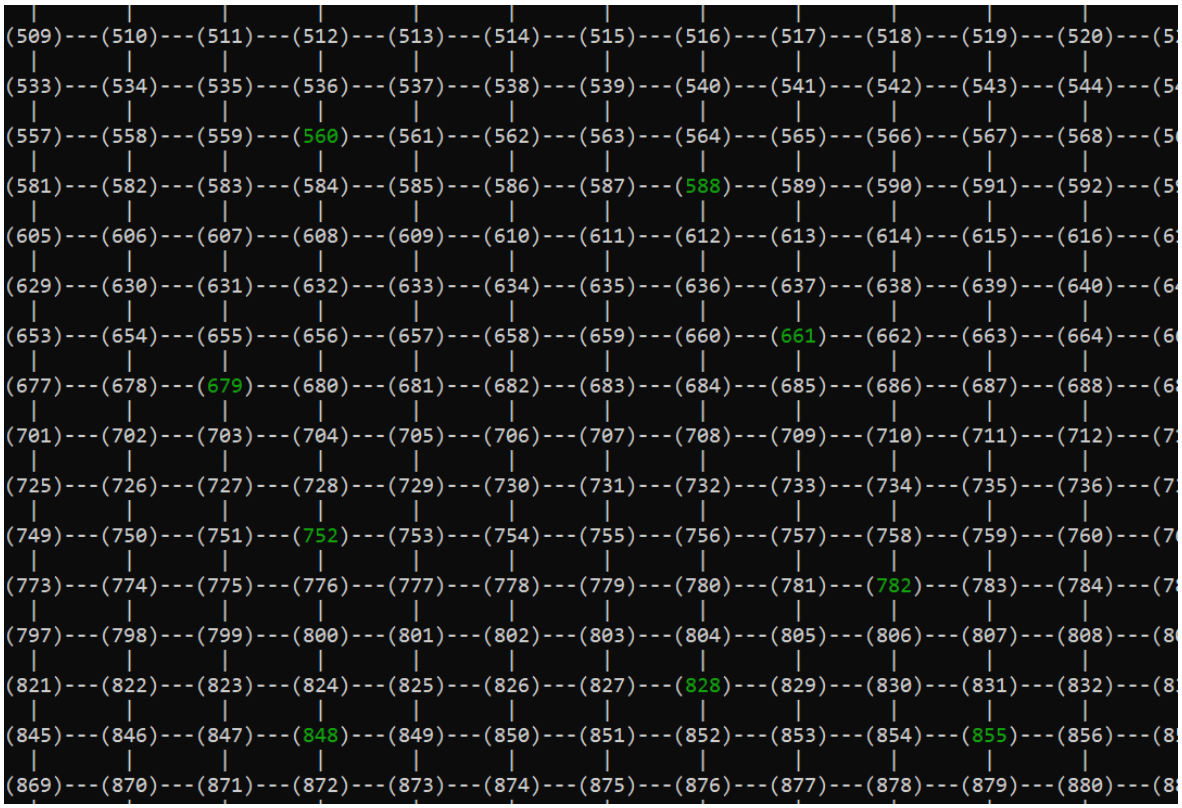
$$\text{if (rand()\%(distance jusqu'au bord / 5) == 0)}$$

La dernière partie consiste à revenir vers le nœud de départ pour avoir une ligne de bus qui boucle, car nos bus sont directed, donc on peut les prendre que dans un sens (Eh oui fallait rajouter de la complexité dans notre projet ; grosses complexités). Je reviens donc peu à peu vers le début de la ligne en : (distance à la source/4) étapes. Et donc je fais une boucle sur cette quantité d'étapes et je reviens, 4 par 4 a la source de la ligne. Tout comme pour le métro et les trams, je regroupe les stations de bus qui sont trop proches l'une de l'autre. Mais cette fois que si elles sont à 1 de distances

l'une de l'autre. J'ai encore besoin de nettoyer la fonction et rendre les résultats un peu plus cohérent avec la réalité mais comme premier prototype, c'est déjà pas mal

3.2.6 Affichage de cartes

Pour tester, et voir si ma génération de graffe étaient satisfaisante, j'ai codé une fonction qui affiche sur le terminal mes graphes générés au hasard, nous n'avons pas encore d'interface graphique pour afficher cela donc je suis assez limité sur le terminal, avec une largeur maximale de 24 nœuds. Pour cela, dans chacun de mes algos j'ai fait une liste de nœuds que je récupère pour print le tout. D'ailleurs j'utilise cette liste aussi pour regrouper les stations trop proches l'une de l'autre.



3.3 Hamza

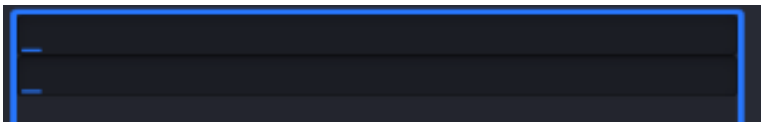
3.3.1 Début de l'interface graphique

Comme évoqué pour la première soutenance le but ici est de créer une interface graphique simple et intuitive qui ne nécessitera pas d'importantes explications. Pour

cela j'ai décidé qu'il serait idéal de proposer deux choix à l'utilisateur, un premier où il pourra simplement rentrer manuellement les points de départ et d'arrivée puis lui afficher le trajet, ou alors séléctionner directement sur le carte deux noeuds dont le chemin s'affichera d'une couleur différente. J'ai donc deux axes sur lesquels travailler.

3.3.2 1ère partie

Pour cette 1ère partie je me suis occupé de créer la première zone dans laquelle l'utilisateur pourra rentrer directement les coordonnées de départ et d'arrivée et obtenir le chemin sous forme de retour. Pour cela j'ai d'abord crée une fenêtre à l'aide de Glade dans laquelle j'ai rajouté toutes les zones de travail nécessaires au bon fonctionnement de notre application. Il fallait donc premièrement créer deux zones d'entrée dans lesquelles l'utilisateur pourra envoyer ses coordonnées aux programmes codées par Titouan et Guillaume

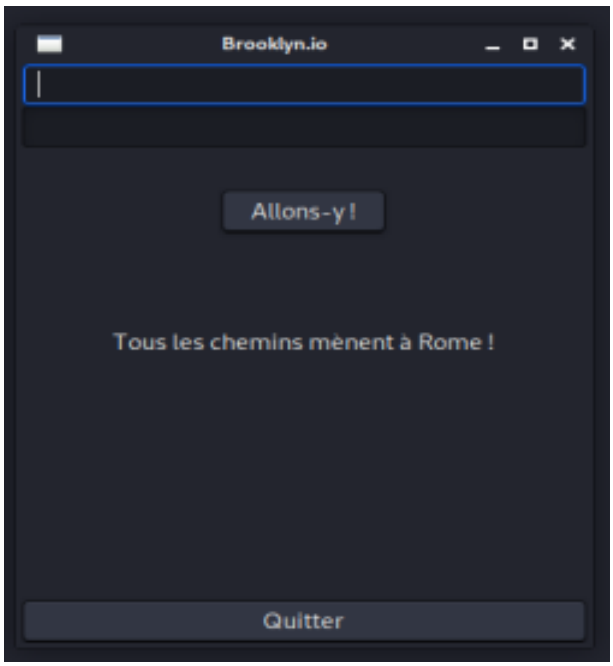


Ces zones de textes permettentront à l'utilisateur de pouvoir rentrer une position de départ ainsi qu'une position d'arrivée qui sera ensuite directement envoyé aux programmes déjà codés. La petite spécificité ici étant que la zone d'entrée récupère une chaîne de caractères il fallait faire attention à ce que le retour au programme soit bien deux entiers qui correspondent aux noeuds de départ et d'arrivée. Les coordonnées étant maitenant tapées il faut pouvoir les renvoyer au programme et cela passe par un bouton.

J'ai choisi d'utiliser Glade car c'est un très bon outil de conception d'interface graphique et qui permet aussi de pouvoir s'intéresser directement aux parties utiles du code, la preuve ici. Pour la création du bouton j'ai pu le "concevoir" directement sur Glade ce qui m'a pris peu de temps mais m'a permis de pouvoir passer plus de temps à me concentrer sur la gestion du signal que renvoit ce bouton. Lorsque l'utilisateur

clique sur le bouton, le signal est lancé afin d'utiliser la fonction calculate que j'ai codé et qui va permettre à partir des deux zones d'entrées plus haut de récupérer les informations de l'utilisateur, passer de la chaîne de caractères à un entier avant de renvoyer directement ces informations pour les programmes de Guillaume et Titouan. Maintenant que le chemin est calculé, il s'agirait de l'afficher. Pour cela j'ai pu créer avec Glade une zone "label" qui permettra d'afficher le résultat du calcul. Le chemin étant calculé par la fonction calculate, c'est dans cette même fonction que je crée un buffer qui permettra de stocker le résultat renvoyé par le programme avant de l'afficher dans la zone "label".

Enfin j'ai rajouté un dernier bouton sait-on jamais si l'utilisateur n'a pas de souris, notre application est utilisable sans souris, afin de quitter l'application. Ce bouton lorsqu'il est activé renvoit directement vers le signal de la fonction quitClicked que j'ai codée et qui permet de fermer la fenêtre dans laquelle l'application est présentée. Tous ces élèments imbriqués entre eux m'ont permis de fournir la 1ère partie de l'interface graphique.



3.3.3 2ème partie

En ce qui concerne la deuxième partie de l'intérface étant donné les spécifications des programmes des calculs du plus court chemin sous forme de noeuds j'ai dû écarter l'utilisation des API Google Maps qui auraient pu être utile ici pour afficher une carte ainsi que le travail nécessaire pour afficher le chemin entre deux points de cette carte. Dans cette optique là je travaille et doit continuer à travailler afin de trouver une solution pour afficher les graphs,car ce ne sont pas vraiment des cartes, sur l'application et permettre à l'utilisateur de séléctionner deux noeuds de ce graph avant d'afficher le chemin correspondant entre les deux points. Une piste intéressante de travail est de créer au fur et à messure des boutons selon la taille des graphs ainsi que des liens entre eux, puis selon la sélection par l'utilisateur et le retour des programmes, changer la couleur des liens entre les boutons de départ et d'arrivée afin d'afficher le chemin. À terme, mon objectif pour la prochaine soutenance est de terminer cette deuxième partie et améliorer potentiellement la première.

4 Fin du projet, dernière soutenance

4.1 Guillaume

4.1.1 Après la soutenance

Après la deuxième soutenance j'ai directement repris le projet pour réctifier ce qui n'allait pas. Comme dit pendant la soutenance j'ai commencé par enlever tous les fichier qui n'avaient à faire ici comme les anciens rapports et le site web. J'ai mis en ligne le site en utilisant les fonctionalités de git avec un nouveau projet dédié. J'ai mis en place les branches pour clarifier le git, créer un .gitignore pour ne pas push les fichiers de test. Enfin j'ai passé un temps concidérable à tout commenter d'une manière qui me semblait propre pour ne pas avoir de difficulté à reprendre le projet après les partiels.

4.1.2 Nouvel algorithme

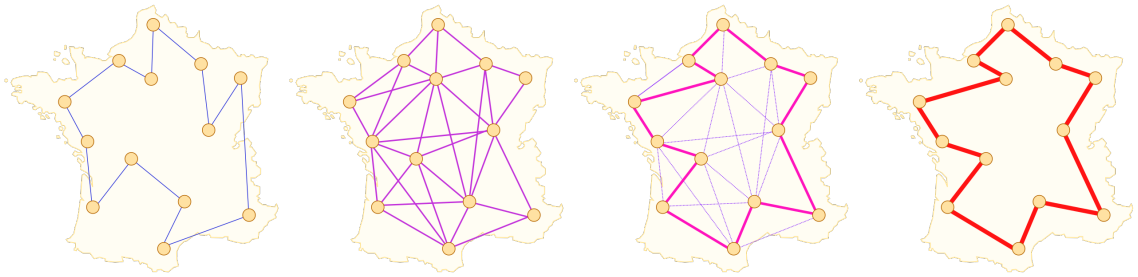
Mon objectif pour la dernière soutenance a été de pouvoir résoudre le problème du voyageur de commerce avec plus de points et pour cela j'ai commencé à chercher un algorithme moins complexe. Les premiers résultats intéressant ont été les algorithmes partant du principe que l'inégalité triangulaire est respecté, cela permet des heuristics très éfficaces. Cependant une fois que j'ai compris ce qu'était l'inégalité triangulaire j'ai aussi compris que ces algorithmes ne pourraient pas nous etre utile, car notre graph ne la respect pas. En effet elle suppose que le chemin le plus court pour aller d'un point A à un point B est une ligne direct, sauf que à cause des transports en commun il

peut etre plus intéressant de passer par un point intermédiaire. J'ai donc continué mes recherches et je suis arrivé sur un algorithme d'un autre type : celui des algorithme de fourmi.

4.1.3 Algorithme de colonies de fourmies

C'est un algorithme d'optimisation qui s'inspire du comportement des fourmies cherchant à aller de la fourmilière jusqu'a la source de nouriture. C'est un genre d'algorithme vaste, dans mon cas il est nommé "système fourmique", le déroulement et le suivant :

- Chaque fourmis ne peut visiter les noeuds qu'un par un
- Plus le noeud est loin de l'actuel, moins il a de chance d'etre choisi
- Plus les phéromone sont important en un noeud plus il a de chance d'etre choisi
- Une fois le trajet terminé la fourmi dépose des pheromones en fonction du temps qu'elle a mis, plus c'est court plus il y en a
- les pheomones s'évaporent partielement à chaque itération



1 **2** **3** **4**

On voit bien dans cette visualisation le set de noeud initial, puis tous les chemins possible, la quantité de pheromone déposé et enfin le chemin en ayant le plus retenu

Le plus important dans ce type d'algorithme est les parametres choisi, quelle importance donner aux pheromones ou à la distance, quelle est la quantité de pheromone qui s'évapore ect..
Heureusement il y a beaucoup de ressource et j'ai facilement trouvé des données pour ne pas partir de rien. Pour ce qui est de la probalité de choisir le prochain noeud elle est calculé comme cela : $((Pij)^A * (Dij)^B) / \sum((Fij)^A * (Dij)^B)$
On a donc Pij qui représente la quantité de phéromone pour aller à J depuis I, Dij repésente la distance et A, B sont des parametres constant. Une fois qu'on a tous ces éléments ont peut implémenter l'algorithme.

La fonction principal : main_ant

- Création tableau à deux dimensions pour les pheromones
- Lancement de la colonie n fois avec n le nombre d'iteration constant
- création de du chemin en choisissant les noeuds ou il y a plus de féromones

La fonction de la colonie : run_colony

- Evaporation des pheromone précépendant par un facteur x constant
- Sauvegarde des précédent pheromone pour que toutes les fourmies de la colonie ai les mêmes
- Ensuite pour toute les fourmies de la colonie
- — Création du chemin aléatoire avec la fonction make_ant_path
  - ajoue des pheromones en divisant une constante par ratio par rapport au meilleur chemin
- normalisation des pheromones pour avoir des nombres plus propres

La fonction pour le chemin : make_ant_path

- Creation d'une liste des prochains noeud possible si la fourmie n'y ai pas deja passé
- Calule des probabilité pour chacun de ces noeuds
- Création d'un nombre flotant aléatoir inferieur au max des probabilités
- ajoue du noeud choisi au chemin de la fourmie
- retourne la longueur du chemin

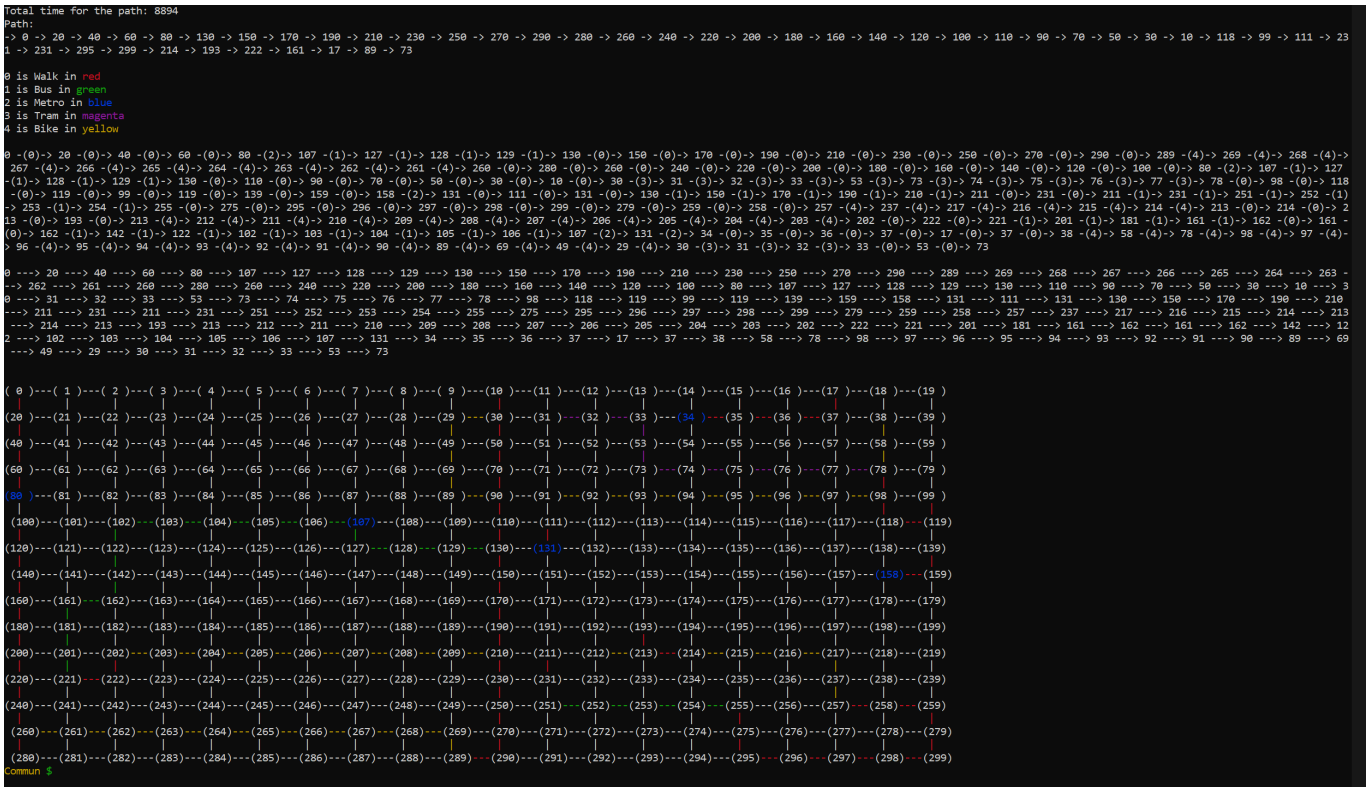
Après beaucoup d'essaie facilité par des fonctions de test pour pouvoir tester beaucoup de cas d'un coup, les paramtres j'ai choisi sont : A=3, B=5, x=0.99, n=100 et un colonie de 20 fourmie. Pour tester je me suis aidé de l'algorithme de brutforce écrit précédement car je suis sûr qu'il donne le bon résultat, j'ai donc pu calculer la marge d'erreur sur des petites donnés pour la réduire rapidement

4.1.4 Efficacité

Au niveau de la vitesse il n'y rien à redire c'est extremement rapide, instantané même pour 50 noeud, cependant ce qu'on ne perd pas en temps on le perd en précision, pour 10 noeuds la marge d'erreur est d'environ 15%, et pour 50 environ 30%.
J'ai donc choisi de concerver l'ancien algorithme pour les cas ou les nombre de noeud est inferieur à 10, qui était sa limite pratique, de cette façon le nombre de cas possible s'étend sans compromettre la précision. Utiliser les fonctions deja en place s'est révélé assez ardu car elle n'était pas prévu pour, il a donc fallu faire un peu de bidouillage, globalement le code est beaucoup moins beau mais je n'ai pas le temps de le rendre propre.

4.1.5 Conclusion

En récapitulatif ma contribution au projet a été la recherche de plus court chemin et la résoltution du problème de voyageur de commerce. Le projet peut calculer même avec des graph de 1000x1000 dans un temps résonable et le nombre de destination peut aller jusqu'a 50 sans poser trop de problème grâce aux optimisations et algorithme adapté.



Exemple de rendu terminal lorsque l'on utilise enfiron 50 noeud dans un graph de 15x20, la première serie de nombre est l'ordre des destionations, les deux suivantes représentent le trajet total et le tableau tout en bas est la visualisation du trajet, même si il deviens peu lisible avec autant de noeud.

4.2 Titouan

4.2.1 Reprise du travail

Après la deuxième soutenance, j'ai continué le travail, j'ai procédé au nettoyage du projet, séparant quelques fonctions et commentant les fonctions. J'ai créé ma branche git pour être plus organisé avec le groupe. Je ne savais toujours pas à quoi cela servait, mais voilà qu'on apprend des choses de partout. Puis les partiels... et abandon du projet pour bosser le partiels

4.2.2 Nettoyer le code

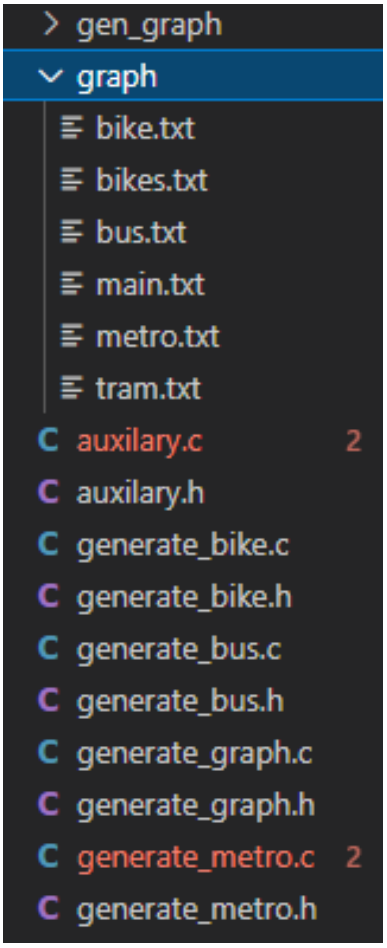
Je voyais bien qu'etre organisé et propre est bien mieux que l'inverse, mais je ne me rendais pas a quel point c'est important. Le fait d'avoir laissé de côté le projet durant les partiels m'a fait prendre du recul sur celui-ci, je n'avais plus la tête dedans, et je ne me rappelais plus exactement comment marchais mes algorithmes.

Ceci n'a fait que d'un : se remettre au travail était extrêmement difficile, la motivation n'y était pas, je ne comprenais plus comment était organisé mon projet ni mes fonctions. Et de deux reprendre des fonctions qui n'étaient pas 100% débuggé impliquait que la remise au travail ne pouvait pas etre progréssif, mais qu'il fallais y mettre toute ca tete d'un seul coup.

J'ai commencé donc par commenter toutes les fonctions, cela me force a être un peu plus propre et organisé, mais aussi de me remettre doucement la tête dans le bain. Je relis mes fonctions pour mettre des commentaires sur certaines conditions un peu complexes.

Une fois cela fait, je cherche à diviser les fonctions. J'avais certaines fonctions qui faisaient à peu près 800 lignes. Le but est donc de trouver des choses que je fais à plusieurs endroits de la même manière pour en faire une fonction auxiliaire que j'appellerais plusieurs fois dans ma fonction principale. J'en ai fait plusieurs comme cela.

Après cela, j'ai cherché à organiser mes fichiers, pas juste mes fonctions. J'ai fait des nouveaux fichiers, 1 pour chaque transport que je cherche à générer sur ma carte et 1 fichier avec toutes les fonctions auxiliaires. Ce fichier auxiliaires contient tous les includes et librairies dont j'ai besoin, j'appelle donc ce fichier.c dans tous les autres fichiers.c de génération de transports qui sont eux même included et appellés dans un fichier : graph_generation.c. De plus tous les fichier .txt sont désormais stockés dans un dossier ce qui rend bien plus compréhensible l'organisation du projet.

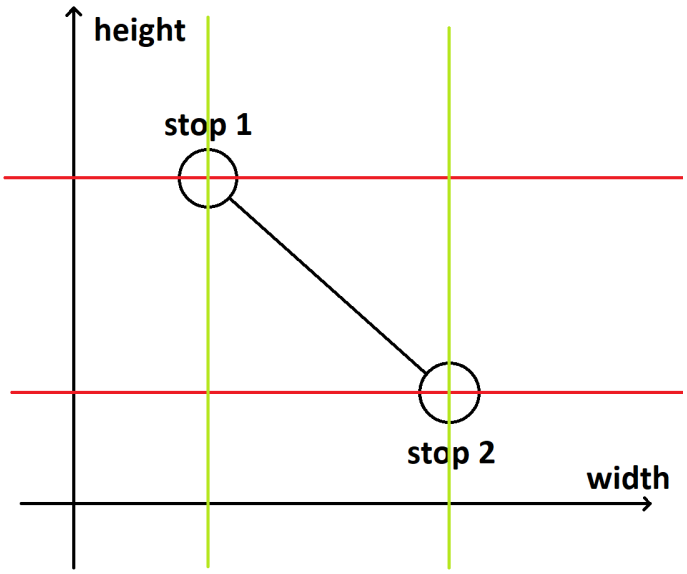


4.2.3 Amélioration génération métro

Durant la deuxième soutenance, j'ai présenté des méthodes pour générer tous les transports, toutes différents sauf pour le métro et le tram. Les deux étaient générés similairement, ils formaient une sorte de quadrillages de la carte, évidemment bien plus complexes que cela, avec beaucoup de random dans la fonction. Cette méthode collait assez bien avec l'organisation de métros que nous connaissons de nos jours aux Etats Unis par exemple, mais bon, on reconnaît tout de même que ce ne sont pas les champions de l'organisation de villes (avec leurs millions de parkings et autoroutes, tout tourne uniquement autour de voitures). Mais cette méthode reste assez simple comparé a celle de génération du bus par exemple. Loïc a donc suggéré que je trouve une nouvelle manière de générer le métro. Il m'a suggéré de parler à un autre groupe qui générait une ville, leur méthode était bien différente, mais je m'en suis inspiré un peu.

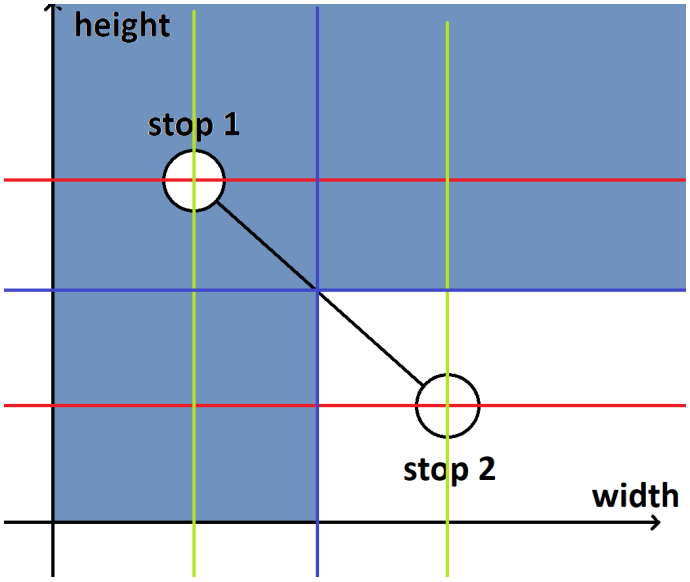
Grace au fait de commenter et diviser les fonction, coder cette fonction a été beaucoup plus agréable.

Voila comment je génere mon métro : Je décide déjà combien de lignes de métro je vais avoir, cette quantité dépend bien évidemment de la longueur et largeur de la carte, mais augmente de manière logarithmique. Je commence pour chaque ligne par placer un point, totalement au hasard (rand()) dans les limites de ma carte. Je vérifie que ce point est assez éloigné de tous les autres, pour éviter deux lignes qui commencent au même endroit ou trop proches l'une de l'autre. Je trouve un deuxième point, donc dans les limites de la carte j'avance d'a peu pres 7 cases pour trouver dans n'importe quelle direction un deuxième arrêt de métro.



Je commence donc une pseudo-récursivité. Je garde en mémoire le premier arrêt. A partir du deuxième arrêt je cherche a en placer un troisième, celui-ci sera donc limité par le bord de la carte, mais aussi par le premier arrêt.

Ce troisième arrêt pourra aller dans toutes les directions par rapport au deuxième, mais ne pourra que possiblement (selon random()) remonté que de la moitié de la distance qu'il y a entre l'arret 1 et arrêt 2. C'est assez compliqué a expliqué. Mais dans le graphique ci-dessous, à partir de stop 2, le nouvel arrêt 3 ne pourra pas être dans l'espace bleu en plus d'être contraint pas les limites de la map.



Ainsi donc, je remplacerais dans ma boucle le premier arrêt par le deuxième, et le deuxième par le nouvel arrêt, le troisième et je répète le procédé jusqu'à ne plus pouvoir avancer, c'est-à-dire se retrouver dans un coin, ou je pourrais rajouter un autre paramètre qui coupe certaines lignes avant le coin.

4.3 Hamza

4.3.1 Récapitulatif

Jusqu'ici j'avais une première partie d'interface qui étaient à peu près fonctionnelle et toute la deuxième partie à réaliser encore. J'avais rencontré quelques problèmes au moment d'utiliser les fonctions codées par mes camarades mais cette fois si en communicant le travail a pu être réalisé. J'avais essayé de réaliser la deuxième partie de l'interface uniquement avec GTK mais ça aura été une perte de temps étant donné le résultat non concluant de cette démarche. J'ai donc décidé de repasser à la SDL afin d'afficher graphiquement les graph.

4.3.2 1ère partie de l'interface

En ce qui concerne la 1ère partie de l'interface, il manquait peu de travail encore. J'avais quelques soucis lors de la récupération du chemin et de son affichage sur l'interface graphique. Mais grâce aux explications de Guillaume et de Titouan j'ai pu mieux comprendre les résultats renvoyés par leurs fonctions et donc traiter l'information. J'ai crée un string qui récupère noeud à noeud le chemin que doit parcourir l'utilisateur avant de rajouter des flèches entre chaque noeud afin qu'il soit affiché correctement dans la zone label correspondante.

4.3.3 2ème partie de l'interface

C'est ici que les choses ont commencé à se compléxifier pour moi. J'avais essayé avant notre passage en soutenance puis même après de réaliser l'interface graphique uniquement avec GTK ce qui a été un échec. J'ai essayé de générér des boutons que l'utilisateur pourrait actionner ce qui afficherait le chemin le plus rapide entre ces deux boutons (qui correspondent aux noeuds du graph). Cette tentative n'a pas porté ses fruits mais j'ai pu me familiariser avec toutes les possibilités qu'offre GTK.

Mais l'échec de cette tentative ne signifie pas l'abondon. J'ai cherché ensuite quel serait le moyen le plus simple pour pouvoir générer selon la taille demandée différents graphs. Car le principal soucis était là, s'il s'agissait de graph unique j'aurai pu les réaliser à la main mais étant donnée que les graphs étaient généré au fur et à mesure, la complexité des algorithmes augmentait. C'est après cette réflexion que je me suis dit que le meilleur moyen de résoudre ce problème était d'utiliser la SDL.

Pour ce qui est de la SDL j'ai eu à l'utiliser une première fois durant notre projet d'OCR donc j'avais déjà quelques connaissances dessus mais avec le temps il fallait se remettre dessus avant d'être totalement fluide (je ne pense pas encore l'être entièrement à l'heure d'aujoud'hui). J'avais connu quelques déboires durant les dernières soutenances par rapport à l'installation de certains packages sur Linux, c'est quelque chose que je ne maîtrisais pas vraiment mais la pratique m'aura aidé cette fois-ci malgré quelques soucis je n'ai pas eu vraiment de mal à bien installer tous les packages. J'ai donc pu me lancer directement après dans le code.

En ce qui concerne la deuxième partie de l'interface graphique le but était assez simple, pouvoir générer un graph aux mesures demandées et afficher le chemin le plus rapide entre les deux points séléctionnés selon les différents types de transports en commun disponible. Pour cela j'ai d'abord crée une surface entièrement noire qui me servirait de planche de travail. Cette surface est selon les dimensions rentrées par l'utilisateur multipliées par un ratio (qui m'a bien pris des centaines de test afin de trouver le bon ratio qui s'applique à n'importe quelle taille de graph) afin de laisser assez d'espace entre chaque noeud pour tracer un chemin entre ces derniers.

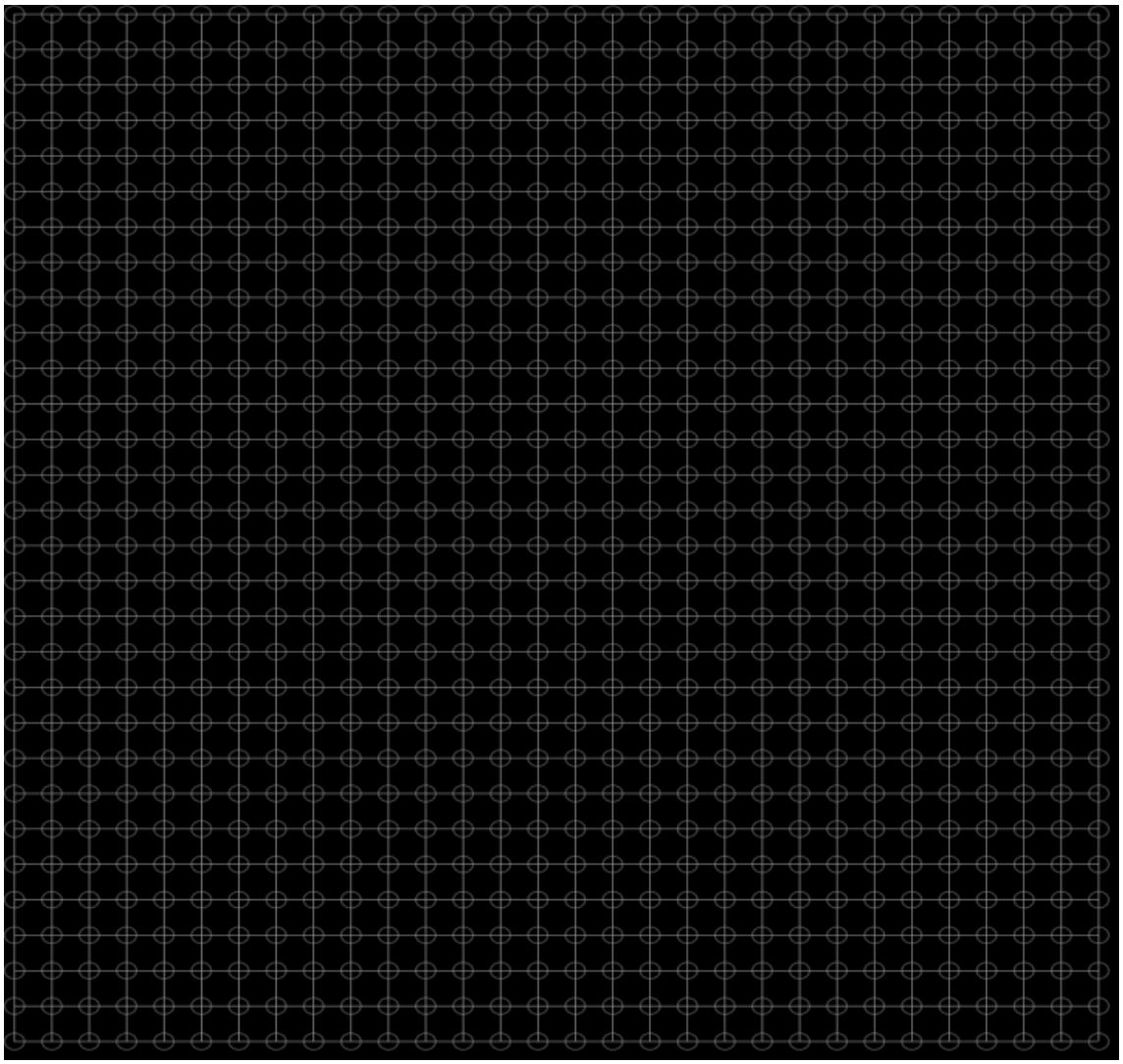
Maintenant que la planche de travail est prête il faut s'atteler à la tâche. Pour ce qui est des noeuds en eux-même j'ai décidé de récupérer une image de cercle simple et de la redimensionner en fonction de la taille des graphs. Après avoir chargé cette image, j'avais deux surfaces SDL que je devais superposer dont l'une à répétition sur l'autre. J'ai donc utilisé deux tours de boucles selon la longueur et la largeur en nombre de noeuds afin de positionner les points. Pour incruster l'image du point dans la surface j'ai dû créer une surface rectangulaire dont les dimensions correspondent aux ratios précédemment calculés et qui permet d'avoir juste le bon nombre de points en long et

en large. A chaque itération de boucle les positions à modifier sont actualisées selon le ratio.

Grille 30x30

Les différents points du graph crées il reste une étape avant d'attaquer le chemin, les relier entre eux. Pour cela j'ai crée deux fonctions auxiliaires Hline et Vline qui permettent de respectivement tracer des lignes horizontales et verticales selon les coordonées x,y et la taille de la ligne souhaitée. Pour ce qui est de la génération pure du graph j'ai décidé de tracer les grandes lignes qui passeraient par tous les points sur la ligne.
Pour cela j'ai quelque peu modifié la fonction qui permettait de rajouter les points sur la grille en ajoutant des grands lignes directement à chaque itération de la boucle, en effet j'ai utilisé des variables temporaires pour ne pas chambouler les placements des points afin de travailler sur les lignes. J'ai réalisé le même travail sur les lignes qui consistait à chaque fois à rajouter le ratio précédemment calculé à chaque coordonnée

afin de bien avoir des lignes horizontales et verticales pour chaque point. La différence ici étant que je ne partais non plus de l'origine du graph c'est-à-dire le point 0,0 mais le premier centre de cercle que je rencontrais afin que le croisement des verticales et horizontales se fasse au centre du cercle.



Grille 30x30 reliée

Il ne me restait plus qu'à modifier le chemin selon le trajet renvoyé par les fonctions codées par mes camarades sur le graph afin de compléter le traitement de l'image. Pour se faire j'ai crée une nouvelle structure dans notre projet qui correspond aux points de graph. Chaque point est constitué de ses coordonnées de son numéro ainsi que du suivant. J'ai donc pu récupérer à chaque fois le point dans le chemin à parcourir directement dans cette liste de noeuds afin de récupérer ses coordonées. Une fois les coordonnées récupérées je peux utiliser soit la fonction Hline ou Vline selon la position du point suivant afin de tracer le chemin entre ses deux points selon la couleur correspondante au moyen de transport utilisé. Cette information se trouve dans le chemin

entre les deux points que je récupère lors du traitement du graph.

4.3.4 Conclusion

En ce qui me concerne personnellement je reste un peu sur ma faim concernant ce projet étant donné que je n'ai pas vraiment été passioné par l'affichage graphique et le traitement d'images mais mon travail au cours de ce projet m'aura permis de plus me familiariser avec ces aspects d'un projet informatique. Il faut dire que j'avais pas vraiment le choix non plus étant donnée mon parachutage assez tardif dans le groupe ce qui aura à mon sens posé quelques bases de problèmes que je n'ai pas su rapidement régler ce qui a eu pour conséquence une prise de retard dans mon travail. Malgré les difficultés rencontrées au début, la fin du projet s'est bien mieux passée car nous avions une meilleure communication ce qui m'a fait réaliser l'importance de partager avec ses collègues. Ca n'aura peut-être pas été simple du début à la fin mais je suis fier du résulat et des leçons tirées.

5 Conclusion

5.1 Récapitulatif de qui a fait quoi

Tâches	Guillaume	Titouan	Hamza
Génération vélos		x	
Génération métro		x	
Génération tram		x	
Génération bus		x	
Print transports		x	
Interface graphique			x
Génération des graph		x	
Affichage des graph		x	x
Affichage du chemin		x	x
Tas binaire	x		
Adaptation des données pour le TSP	x		
Bruteforce du TSP	x		
Multithreading de Dijkstra	x		
Valgrind du code	x		
Système fourmique	x		
Création du chemin rendu	x	x	

5.2 Conclusion

Nous avions un projet plus léger du fait que nous étions moins nombreux dans le groupe. Vite fait, peu après la première soutenance, nous avions tout ce dont nous voulions faire, c'est-à-dire créer un graph, et puis le parcourir pour trouver le chemin le plus court entre 2 points grâce a Dijkstra. La seule chose à ce moment qu'il manquait par rapport au cahier des charges était la fenêtre graphique. Hamza a réussi à faire marcher une fenêtre graphique et a afficher un graph sur celle-ci. Nous avons donc augmenté la difficulté du projet. Guillaume entame la mis en place du problème du voyageur de commerce, puis entreprend un algorithme de brut force puis un système fourmique. Titouan entreprend de la génération de graphe avec nombreuses techniques différentes. Ces deux taches peuvent vite fait devenir très complexes : le voyageur de commerce est un problème très complexe et de nombreux chercheurs passent encore des années pour améliorer leurs algorithmes. La génération de cartes, quant à elle, est très utile dans des domaines comme les jeux vidéo et s'agit de représenter la réalité grâce à des conditions précises. En conclusion, Guillaume et Titouan se sont plongé dans deux problèmes bien intéressants, mais on a peine frollé la surface de ce cis. Le projet et surtout les recherches ont été très intéressantes. Nous aurions dû organiser plus notre projet depuis le début, commenter notre code, et mettre plus en commun nos parties, il manquait un peu de travail d'équipe.