Brooklyn

Titouan LOCATELLI

Guillaume MERCIER

Hamza OUHMANI

2022

Deuxième rapport de soutenance

Table des matières

1 Introduction

1.1 Introduction

1.2 Reprise du cahier des charges

Lors du cahier des charges, donc avant de commencer le projet, nous avions émis la liste de tâches et la répartition suivante.

Tâches	Guillaume	Titouan	Hamza
Classes et fichiers	x		
Graph carte de base		x	
Premier algorithme	x	x	
Première méthode de transport	x		x
Améliorer l'algorithme en conséquence	x	x	
Implementer autres modes de transports	x		x
Sélection de point de départ et arrivée			x
Affichage graphique			x
Interface graphique			x
Modifier la carte		x	x
Pouvoir choisir entre plusieurs cartes		x	x
Site web	x	x	x
Rajouter des étapes de déplacement	x	x	x
Cartes aleatoires	x	x	x

Bien sûr, ce n'est pas ce qui c'est exactemet passé, il nous semble important de souligner les changements en terme de contenue et de répartition.

Tâches	Guillaume	Titouan	Hamza
Classes et fichiers	x		
Graph carte de base		x	
Premier algorithme	x		
Première méthode de transport	x	x	
Améliorer l'algorithme en conséquence	x		
Implementer autres modes de transports		x	
Sélection de point de départ et arrivée			x
Affichage graphique		x	
Interface graphique			x
Modifier la carte		x	
Pouvoir choisir entre plusieurs cartes		x	
Site web	x		
Rajouter des étapes de déplacement	x		
Cartes aleatoires		x	

Nous allons revenir dessus en détail lors de la présentation du travail, mais le gros de ce qui était prévu est déjà opérationnel, et ce n'était pas du tout attendu.

2 Présentation du travail

2.1 Guillaume

2.1.1 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.

2.1.2 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	$\Theta(1)$	Θ(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.

```
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..

2.1.3 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.

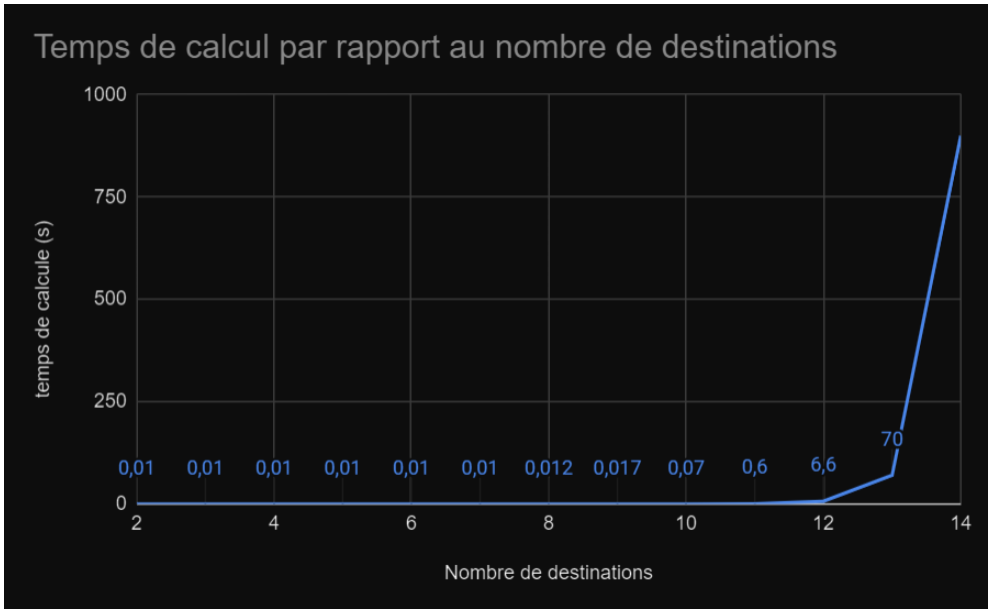
La première approche est évidemment le brut force, il faudrait donc tester toutes les possibilités ce qui revient à tester n ! chemin, cela est impossible. C'est un problème NP-difficile, il existe donc un algorithme ayant une complexité polynomial qui le résou, mais aucun n'a était trouvé. Pour choisir 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.

2.1.4 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.

2.1.5 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.

2.1.6 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.

2.1.7 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 !

2.2 Titouan

2.2.1 Recapitulatif de la dernière soutenance

Nous avons présenté un projet fonctionnel pour la première soutenance, nous pouvions utiliser l'algorithme de Dijsktra sur un premier graphe assez petit : de taille 20*15, et puis finalement afficher celui-ci sur un terminal. Nous avions pris pas mal d'avance et fait la plus grande partie du travail indiqué sur notre charte de travail. Il nous restait plus qu'a incorporé le problème du voyageur de commerce et d'avoir plusieurs graphes sur lequel le tester. Donc, pour la deuxième soutenance, j'ai attaqué la création de graphes en abusant de la fonction rand().

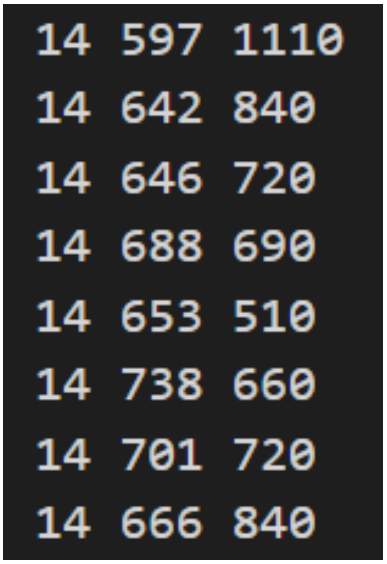
2.2.2 Génération de carte

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

2.2.3 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

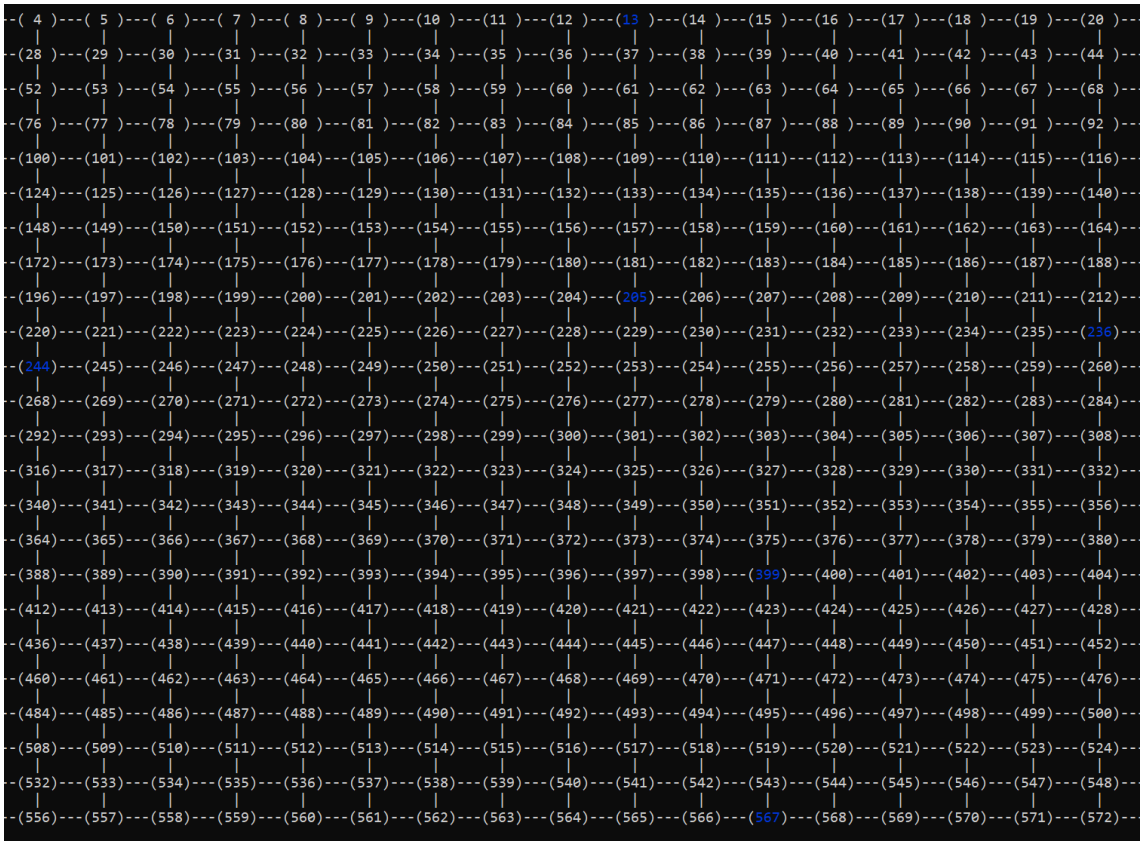


J'en ai parlé avec Guillaume et nous avons décidé de lier tous les vélos entre eux, ça paraissait plus simple. Guillaume n'a donc qu'à récupérer ce fichier avec créer les arcs dans le graphe avec le coût indiqué.
L'idée était donc à peu près de diviser le graphe en plein de blocs de taille 5*5, voir 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.

2.2.4 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.



2.2.5 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.

2.2.6 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 :

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 :

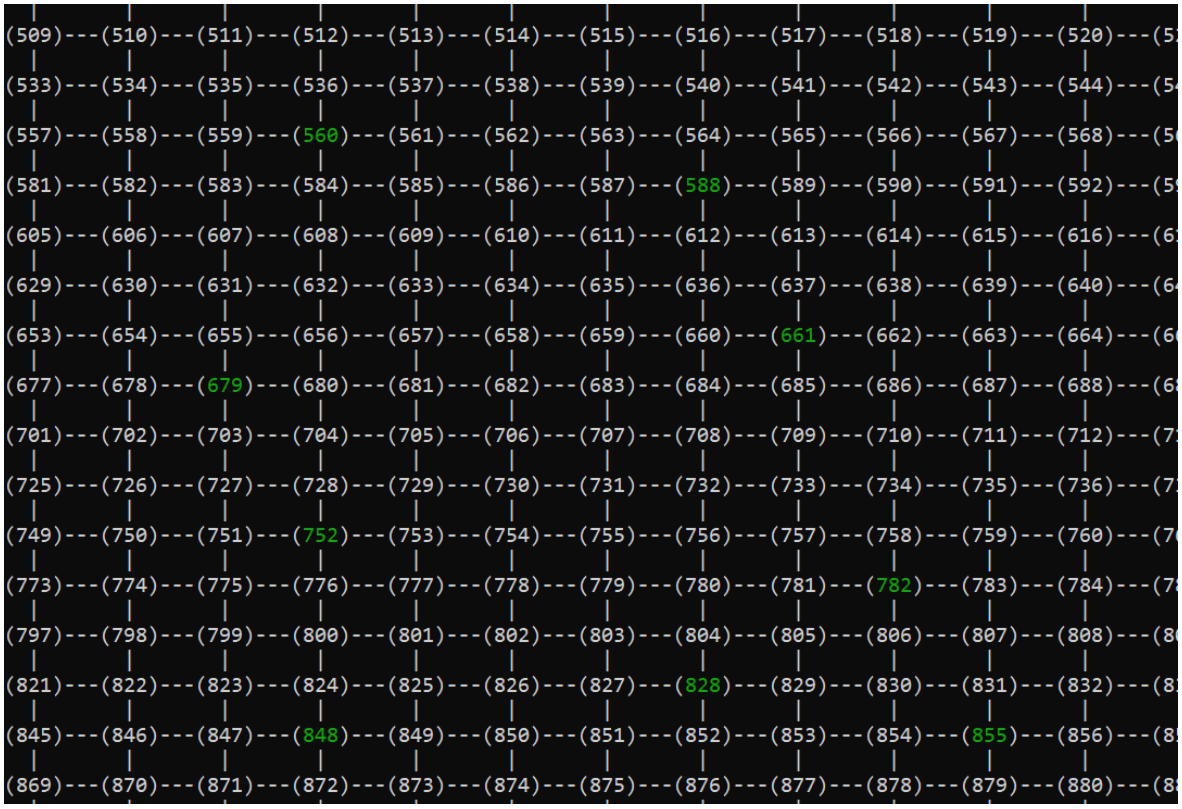
if (rand()%(distance jusqu'au bord / 5) == 0)

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.

2.2.7 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.



2.3 Hamza

2.3.1 Récapitulatif

Pour la première phase de ce projet j'ai rencontré toutes les difficultés du monde à installer les outils nécessaires au lancement de ma contribution au projet. Les principales difficultés venant du fait que je n'étais pas assez à l'aise sur Linux pour pouvoir installer GTK et Glade sans provoquer d'erreurs. Mais depuis grâce à des camarades ayant la même distribution Linux que moi j'ai pu mener à bien l'installation et commencer à pouvoir fournir une interface graphique.

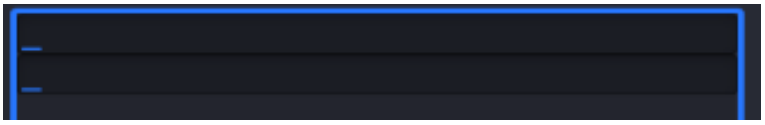
2.3.2 Début de l'interface graphique

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.

2.3.3 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

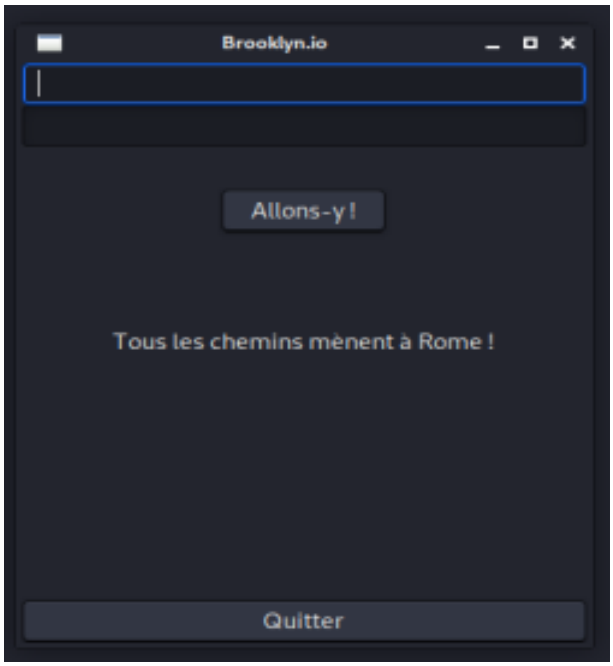


Ces zones de textes permettentront à l'utilisateur de pouvoir rentrer une position de départ ainsi qu'une position d'arrivée qui sera ensuite directement envoyé aux programmes déjà codés. La petite spécificité ici étant que la zone d'entrée récupère une chaîne de caractères il fallait faire attention 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.



2.3.4 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.

3 Conclusion

3.1 Objectifs pour la prochaine soutenance

- Nouvel algorithme pour répondre au problème du voyageur de commerce : soit on cherche un nouvel algorithme, ou on implémente celui de Bellman Held-Karp.
- Tweaks a la génération de cartes pour la rendre plus réaliste.
- Possiblement rajouté quelques contraintes supplémentaires si nous avons de l'avance.
- Une interface graphique affichant les graphes de taille quelconque et les chemins.

3.2 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	
Début interface graphique			x
Tas binaire	x		
Graphe pour TSP	x		
Correction bug chargement graphe	x		
Algo de résoltion du TSP	x		
Adaptation de l'algorithme	x		
Multithreading	x		
Valgrind	x		

3.3 Conclusion

Nous avons bien avancé, plus que prévu, la preuve, nous avons fait tout ce que nous avions pensé faire dans le cahier des charges, il est désormais que le grand

vizir suprême rajoute de la complexité à notre projet.
Nous avons commencé le et implémenté une solution au problème du voyageur de commerce, avec un algorithme a compléxité similaire a factorielle, nous avons généré des graphes aléatoires sur lesquels travailler et nous avons un début d'interface graphique.
Guillaume et Titouan vont ensemble unis leurs pouvoirs pour réduire la complexité du problème du voyageur de commerce avec un nouvel algorithme. Nous pourrons peut-être rajouter quelques contraintes en plus, seul le grand vizir sera la juge de cela.