**Structure de données**

[Parler de la du choix de la structure/ praticité/ liberté…]

**Construction du Graphe**

Pour construire le graphe nous voulions dans un premier temps considérer que celui-ci était non-orienté. Cependant en travaillant dessus, en particulier en voulant afficher le texte indiquant à l’utilisateur le parcours qu’il doit faire pour le plus court chemin nous avons constaté une anomalie sur la ligne 10.



*Zone de la ligne 10 représentant un problème.*

En effet sur l’ensemble du graphe les arrêtes entre les stations peuvent être parcourus dans les deux sens. Alors par soucis de cohérence nous avons décidé d’orienter cette partie du graphe.

[Parler des fonctions de constructions]

**Plus court chemin entre deux stations**

Dans cette partie nous cherchons le plus court chemin pour aller d’une station de métro à n’importe quelle autre.

**Choix de l’algorithme**

Parmi les différents algorithmes étudiés nous avons choisi d’utiliser celui de Dijkstra pour résoudre ce problème.

Nous pouvons nous demander pour quelle raison ne pas utiliser l’algorithme de Floyd-Warshall ou celui de [INSERER NOM ALGO]. L’algorithme de Floyd-Warshall permet de trouver les plus courts chemins entre tous les sommets d’un graphe ce qui n’est pas nécessaire dans notre cas, utiliser cet algorithme augmenterait donc le temps de traitements de notre programme inutilement. L’algorithme de [INSERER NOM ALGO] quant à lui est structurellement très proche de Dijkstra mais d’une complexité plus importante afin de pouvoir calculer les plus courts chemins dans le cas où le graphe possède des arrêtes de cout négatif. Or notre graphe représente les temps de trajet entre les différentes stations du métro parisien, il ne contient évidemment pas d’arrête de type. Ces deux algorithmes n’étaient donc pas nécessaires pour réaliser notre objectif.

**Fonctionnement du programme**

On construit dans un premier temps la fonction autour de laquelle tout notre programme va fonctionner.

La fonction « dijkstra (graph, summit) » prend en entré « graph », le graphe représentant le réseau de métro et « summit » le sommet de départ, la fonction rend les dictionnaires « dist » et « pred ». Le dictionnaire « dist » à pour clé les sommets du graphe et leur associe le temps de parcours du plus court chemin entre ce sommet et le sommet de départ calculé par notre fonction. Le dictionnaire « pred » a lui aussi pour clé les différents sommets du graphe et leur associe le sommet qui doit les précéder afin de pouvoir réaliser le plus court chemin.

Pour reconstituer le chemin nous utilisons la fonction « shortest\_path (summit, target\_vertex, pred) » où « summit » est le sommet de départ, « target\_vertex » le sommet d’arrivé et pred le dictionnaire énoncé précédemment. Le principe de la fonction est de partir du sommet d’arrivé et de regarder les sommets qui doit le précéder (dans « pred ») jusqu’à arriver au sommet de départ. Une fois les sommets placés dans cet ordre dans une liste, il suffit de d’inverser l’ordre de celle-ci pour connaitre notre plus court chemin.

Dans un second temps nous pouvons coder la partie du programme permettant à l’utilisateur d’obtenir un résultat correct, clair et lisible.

[PARLER DE LA PARTIE GRAPHIQUE / DU SYSTEME QUI RECUPERE LES CLIQUES ET TROUVE LES STATIONS ASSOCIES]

Une fois que nous connaissons les stations entre lesquelles veut voyager l’utilisateur, il faut s’assurer d’avoir un résultat cohérent. Les données auxquelles nous avons accès représente les différentes lignes arrivant à une même station de métro comme des sommets différents du graphe. Nous avons donc créé la fonction « utilisation\_dijkstra (nom\_depart, nom\_arrive, graph) » qui sélectionne dans le graphe tous les sommets nommés « nom\_depart » et « nom\_arrive », c’est-à-dire tous les sommets correspondant aux stations de départ et d’arrivé. La fonction calcul en utilisant « dijkstra (graph, summit) » le temps de parcours de tous les plus courts chemins entre ses sommets et renvoi le plus faible d’entre eux ainsi que le chemin correspondant. De cette manière on indique à l’utilisateur le meilleur chemin entre deux stations toutes lignes confondues.

On peut alors afficher ce chemin [PARLER DE LA FONCTION D’AFFICHAGE]

La partie délicate est maintenant de décrire le trajet à effectuer comme l’aurait fait un GPS. Pour cela on construit 2 fonctions principales. La première est la fonction « trouver\_terminus ( station1, station2, graph) » qui permet de donner le terminus de la ligne dans le sens de « station1 » vers « station2 ». Cette fonction correspond en quelque sorte à un parcours en profondeur en répondant à certaines conditions, le principe général est le suivant.

*trouver\_terminus (station1, station2, graph) :*

*vertex = station2*

*liste\_pred = [station1]*

*Tant que vertex n’est pas un terminus :*

*Pour tout u successeur de vertex :*

*Si u est sur la même ligne que vertex et u n’est pas dans liste\_pred :*

*vertex = u*

*liste\_pred += [vertex]*

*Fin si*

*Fin Pour tout*

*Fin Tant que*

*renvoyer vertex*

Le programme parcourt donc la ligne jusqu’à tomber sur le terminus. La fonction est dans les faits un peu plus complexe que ça et gère aussi différents cas particuliers comme lorsqu’on arrive sur un embranchement où le cas spécial liée à la ligne 10.

Cette fonction n’est qu’un outil créer pour participer à l’affichage du texte servant à décrire à l’utilisateur le parcourt qu’il doit effectuer. La fonction « afficher\_texte\_parcours (pcc, graph) » permet à l’aide du plus court chemin préalablement trouver d’afficher les étapes du parcours de l’utilisateur. Le principe de la fonction est en lui-même simple, elle observe les différents sommets du plus court chemin, sélectionne à l’aide du numéro des lignes ou des embranchements les stations auxquelles l’utilisateur doit effectuer un changement et les indique en précisant la direction que doit prendre l’utilisateur en appelant la fonction terminus.

La fonction gère les différents cas où l’utilisateur doit descendre à un embranchement pour prendre l’autre direction et le cas particulier de la ligne 10.