PROJET RATP 1A HUGO ET MAXIME (Algo. et Prog. & Th. des Graphes)

L'objectif de ce projet est d'implémenter un algorithme de calcul de plus court chemin pour un réseau de transport public. Pour cela on a un accès libre à des fichiers dit GTFS sur le site de la RATP. Nous allons implémenter nos algorithmes en langages C (et Bash).

Nous avons découpé notre gitlab de la manière suivante : Un dossier "Gtfstxt" contenant les fichiers gtfs, le jeu de données créé par nous même avec les données des métros.Un dossier "Headers" dans lequel on met tous nos prototypes (les .h). Un dossier "Sources" dans lequel on met nos .c liés aux .h de "Headers", et le fichier main.c à exécuter. Un dossier "Saves" dans lequel on retrouve tous les retours de fonctions: Graph, rendus des structures des fichiers Gtfs. Ainsi qu'un dossier "Obj" ou on retrouve tous les .o .

Il y a de plus un fichier "Define_Nbr.sh" dans le répertoire courant du git, qui est un fichier en bash, qui permet de créer "Define_Nbr.h" dans "Headers", il est exécuté avec le makefile.

Un **makefile** accompagne des différents fichiers:
La commande "make clean" permet de supprimer les fichier.o du répertoir "Obj", la commande "make Define_Nbr" permet de creer le fichier "Define_Nbr.h" dans le répertoire "Headers". Et enfin la commande "make" permet de compiler le projet.

Ainsi la suite d'instructions suivant permet d'utiliser le projet :
make clean
make Define_Nbr
make
Puis il faut aller dans le fichier "Sources" et exécuter le main avec les options désirées .

(Attention : des erreurs de types proviennent de la Hashmap lors de la compilation, mais cela ne gène pas l'avancé des choses)

- **Bloc “extraction des fichiers gtfs” et “structure”**

  Les fichiers de ce bloc sont les suivants.

  1. Struct.c et son fichier prototype struct.h, Ils permettent de mettre en place différentes structures.

  Une structure “station_t” contenant tous les types de données qu’on peut extraire de tous les types de fichiers gtfs. Ainsi on a une seule structure assez complexe mais générale. Cela nous permet de globaliser l’approche d’extraction.

  Une structure “listes” dans laquelle on range des tableaux de “station_t”, il y a un tableau par fichier gtfs, comme cela on sépare bien les données malgré tout. En effet l'accès aux données qu’on stock est alors assez général: par exemple pour accéder aux données de routes.txt on va dans list->routes[i]->... .

  Une structure “main_args” qui répertorie les option d'exécution de main.c et donc contient à la fois, les données : station de départ, station d’arrivée, et heure de départ. Mais aussi les options telles que l'écriture ou non des fichiers des structures extraites des fichiers gtfs dans le dossier Saves. Cette écriture rend les extractions plus lisibles, mais elles n’apportent rien de plus que les fichiers gtfs, elles sont le reflet de la manière dont on stocke ces données.

  2. L'ensemble des fichiers calendar_dates.h(.c), calendar.h(.c), routes.h(.c), stops.h(.c), stops_times.h(.c), transfers.h(.c), trips.h(.c).
  Chaque fichier.h(.c) possède la même structure:

  1. Fonction "write" qui permet d'écrire les données extraites et la manière dont on les stocke dans un fichiertest.txt qu’on choisit ou non de créer lors de l'exécution du main avec les options d’arguments.

  2. Fonction "fill" qui permet de remplir les structures pour chaque fichier gtfs avec les données qu’il contient. Cette fonction lit ligne par ligne un fichier gtfs et remplis dans le même temps la structure associée. Cela a l’aide majoritairement de la fonction strtok.

  3. Fonction "ffichier" qui implémente les deux précédentes. Elle permet de creer la structure associée et de la remplir et si on le veut de la mettre dans un fichier lisible.

  /!\ Sauf route.h(.c) , ces fichiers ont beaucoup plus de fonctions associées a cause de la syntaxe du fichier gtfs routes.txt . (voir dans difficultés rencontrées).

3. Un fichier Define_Nbr.h permet de définir le nombre de ligne de chaque fichier Gtfs, ainsi on connaît la taille des structures définies précédemment qui permettent de stocker ces données.

- **Bloc “Graph”.**

Les fichiers associés au BLOC “graph” sont :

1 .create_graph.c. et son fichier prototype create_graph.h. Les fonctions associées sont des fonctions de création du graph à partir des données extraites des fichiers gtfs et donc des structures créées précédemment. Le jeu de fonctions fonctionne en deux temps. On construit ligne par ligne les lignes de métro (par exemple) avec les structures créées à partir de routes.txt et stop_times.txt tel qu’on a alors plusieurs graphs linéaires.
Puis on rejoint les graphs avec les transferts de la structure créées à partir de transfers.txt, tel qu' il existe des stations “pont” entre deux ou plusieurs lignes. C'est-à-dire que plusieurs stations du même nom et d’id différentes sont joignables à pied et permettent de lier plusieurs stations de lignes différentes.
La Fonction “write_graph” : écrit le graph dans un fichier “Resultats graph” du dossier Saves tel que pour chaque ligne de métro on la trace verticalement tandis qu’on trace horizontalement les transferts entre même stations d’id différentes qui mènent vers des lignes différentes. On choisit visuellement de noter les arêtes avec des :

/\
||
[poid]
||
\/

pour les stations verticales. Et pour les stations horizontalement (les transferts) station 1 < = > { [poid de l'arête entre station 1 et station 2] station 2 [poid de l'arête entre station 1 et station 3] station 3 }

Une fonction permet de free les structures contenant les données extraites des fichiers gtfs, c'est-à- dire les structures , maintenant qu’on en a plus besoins, puisqu’on a notre graph.

2.struct_nodes.h(.c) comprennent la structure de "node" = sommet de notre graph, avec les différentes fonctions : fonction création de node, fonction de jointure entre deux nodes existants, fonction de création d'un node et jointure à un node existant, fonction de remplissage d'un node avec stop_id, nombre de voisins et nom, fonction de suppression (free) de nodes et du nom (str) associé et enfin fonction de recherche d'un node en fonction d'un stop_id ou d'un nom.

La structure “node” contient bien sûr un nom, un stop_id, mais surtout un nombre de voisins, un poid d'arête et une liste de nodes voisins de types “node”. On fait

correspondre le poid Poid[i] à l'arête entre le node courant et son voisins nextnode[i]. De plus on ajoute a cette structure un int BOOL, qui permet le parcours du graph la lignes: En fait chaque ligne est délimité par un node “DEBUT” et un node “FIN”, de BOOL nuls alors que les nodes propres au graph sont a BOOL = 1. Ce booléen devient ensuite un entier (pas juste 0 ou 1) lorsqu’on applique dijkstra au graph, car il permet de marquer un sommet par exemple.

- **<u>Les Difficultés</u>**

  Une première difficulté s’est trouvée dans la syntaxe du fichier gtfs routes.txt. En effet, d'une ligne à l'autre, elle peut différer lorsqu’il existe plus de deux extrémités sur une ligne de métro (par exemple). Alors pour extraire les nom des stations concernées il faut un jeu de fonctions: rechercheSlash, rechercheParentheses, fillShortName, fillNoneStr, fillCoef0 ,fillSlashAGauche,fillSlashADroite, fillSansSeparateur, enleverParenthesesDepart et enfin fill_struct_route. Qui permettent de distinguer :

  “station 1 / station 2 ⇔ station 3”
  “station 1 ⇔ station 2 / station 3”
  “station 1 ⇔ station 2”
  “station 1”

  Ainsi les fichiers routes.c(.h) sont bien fournis, mais permettent une lecture complète de tous les cas.

  Une deuxième difficulté résidait dans l’utilisation des données des fichiers gtfs calendar_date.txt et de calendar.txt . On les extrait bien comme les autres, dans les structures standards. Puis on ne les utilise pas dans l’implémentation du graph.

  Une troisième difficulté a été de travailler avec un grand nombre de données, ainsi l'exécution de l'ensemble des fichiers est longue. On a alors décidé d’utiliser un jeu de données plus restreint, qu’on a créé nous même à partir des fichiers gtfs full. On a créé un jeu de données : gtfs Métros, qui contient des données de toutes les lignes de métros. Ainsi on peut implémenter plus rapidement nos codes sur ce jeu de données restreint et cela fonctionnera aussi pour le jeu de données : gtfs full.

- **Bloc hashmap**

  Lorsque nous avons réalisé que le traitement du graphe pourrait s'avérer très long, nous avons entrepris de changer notre structure de graphe. Après avoir fait des recherches et compte tenu de la faible connexité du graphe (le réseau des métros est faiblement connexe mais en ajoutant les bus et trams, cela peut changer) nous avons déduit qu'il fallait représenter ce dernier avec une liste d'adjacence où seraient stockées les "paires" (sommets,voisins).

  Nous avons décidé de l'implémenter avec une table de hachage puisque grâce à cette structure, le temps d'accès à une case est moindre. En effet ,la fonction de hachage, pour une clé donnée, permet de réduire drastiquement les cases à parcourir si on choisit judicieusement la taille de la table.

  Dans le fichier graph_lib.c se trouvent les définitions des fonctions utiles à l'exploitation de la table de hachage. Dans le fichier graph_lib.h se trouvent les déclarations des structures et des fonctions ainsi que pour chacune d'entre elles, la description détaillée des arguments et valeurs de retour.

  La clé de la table est l'id des stations puisqu'on ne pouvait pas choisir le nom d'une station, car plusieurs stations de même nom ont des id différents. A fortiori, la valeur de la table est donc le nom de la station.

  Structure graph_station: cette structure représente une station du réseau. C'est aussi un élément de la table de hachage. En plus des champs clé,valeur et "next" essentiels à la structure, nous avons ajouté les champs *voisins* et *nb_voisins* afin de représenter la structure du graphe, c'est-à-dire la liste d'adjacence.

  Structure hashmap_graph: La structure comprend les données essentielles à la création d'une table de hachage: sa taille (i.e le nombre de sous-listes qu'elle contient), la taille mémoire de la clé et celle de la valeur, les fonctions de comparaison et de hachage (nous avons défini des fonctions par défaut mais on laisse à l'utilisateur la possibilité de les spécifier), le pointeur sur la liste des noeuds.

  Nous avons essayé de rendre cette implémentation la plus générale possible dans le cas où nous aurions besoin de plus d'une table, et dans ce cas éviter de définir une nouvelle structure qui serait similaire à la table précédente. C'est pour cela que nous ne spécifions ni le type de la clé ou de la valeur au cas ou nous aurions besoin d'une table pour stocker d'autres types de valeurs.

  En fait, nous espérions terminer l'implémentation générale avec la structure précédente et ensuite l'adapter à la table de hachage, cependant nous n'avons pas eu le temps d'utiliser cette dernière. Cette partie du projet reste donc inutilisée par notre programme.

- ## Bloc dijkstra

  Notre graphe étant pondéré par le temps de trajet pour chaque arête, le calcul du trajet le plus court est en fait un calcul de plus court chemin. Nous avons choisi l'algorithme de DIjkstra pour deux raisons: D'abord, n'ayant que des poids positifs, cet algorithme peut-être appliqué pour un calcul de plus court chemin dans ce graphe. Ensuite, nous voulions par la suite réaliser un Dijkstra bi-directionnel afin d'optimiser le temps de calcul.

  Les fonctions pour réaliser cet algorithme sont définies dans le fichier dijkstra.c et l'explication des paramètres et valeurs de retour se trouvent dans le fichier dijkstra.h.

  Récupérer la distance min: Le premier problème dans l'implémentation de l'algorithme était de récupérer le sommet non traité de distance minimale. En effet, il était nécessaire de créer une liste de sommets puisque parcourir le graphe à chaque itération est bien trop coûteux. Nous avons donc trié chaque sommet déjà vu dans une liste, par distance croissante. Cette liste est représentée sous forme de liste chaînée, les éléments de la liste étant des structures dist_list_node comprenant l'id, la distance et un pointeur vers le prochain nœud. Le pointeur sur la liste est en attribut de la structure dist_list,son deuxième attribut étant le nombre de sommets de la liste pas encore visités (variable nécessaire à la condition d'arrêt de DIjkstra). Ensuite, l'attribut BOOL de la structure node servait de marqueur pour savoir si l'algorithme avait déjà visité cette station.

  Récupérer le chemin: Même si nous l'avons supprimé de notre programme, notre algorithme comprenait une fonction pour récupérer le chemin du départ à l'arrivée. En effet, pour ce faire il fallait partir du sommet d'arrivée puis d'ajouter le voisin de ce dernier ayant la plus petite distance à l'origine. En répétant ce processus on obtient la liste des stations parcourue qu'il ne reste qu'à inverser pour avoir le chemin.

- ## Conclusion :

  Nous avons rencontré plusieurs problèmes dans la réalisation de ce projet, même si nous avons surmonté la plupart d'entre eux, l'algorithme reste cependant incomplet.

  En effet, la partie dédiée au traitement des données GTFS est fonctionnelle, et la structure de graphe créée en amont de la table de hachage l'est aussi, et même si le temps de création d'un graphe peut s'avérer plutôt long avec cette structure, nous n'avons pas eu le temps de mettre en pratique la table de hachage et de l'intégrer à

notre code. De plus, il nous reste des erreurs lorsqu'on lance l'algorithme de DIjkstra, ce dernier n'est donc pas fonctionnel.

Quant aux améliorations possibles, nous aurions pu dès le départ utiliser des tables de hachage pour les structures dont nous avions besoin afin de diminuer le temps d'accès, ainsi que le temps passé à coder.En effet, devoir trouver une structure et implémenter les fonctions pour la manipuler à chaque fois nous a retardé dans l'exécution du projet.

Nous aurions aussi pu recourir à l'algorithme bidirectionnel de Dijkstra cependant nous ne savions pas si le gain de temps d'exécution était suffisant. De plus, une implémentation bidirectionnelle de Dijkstra nécessite la création de deux files d'attente ce qui peut augmenter encore le coût du programme en mémoire.