Projet Algorithmique & Structures de Données

Réseau de transport et parcours de graphes

I. Description globale

Voici les détails de certains choix que nous avons faits durant la réalisation de notre projet.

Nous avons choisi de mettre la quasi-totalité de l'interaction avec l'utilisateur dans le fichier *menu.c* dans l'optique d'un futur ajout d'interface graphique. En effet, à ce moment-là, il suffira de toucher à ce fichier et il ne sera pas nécessaire de toucher aux autres fichiers.

Nous avons utilisé *Valgrind* afin d'analyser la gestion de la mémoire du programme. Cet outil nous a permis de détecter et corriger des fuites de mémoire et des accès mémoire invalides, garantissant ainsi la fiabilité et la robustesse des algorithmes implémentés.

Afin de créer un fichier de test de réseau énorme, nous avons sollicité une IA, en lui demandant d'ajouter des erreurs similaires au fichier fourni, mais aussi d'autres erreurs de son choix. Ne pouvant pas nous envoyer un fichier aussi long, elle nous a donné un script *Python*. Ces nouvelles erreurs nous ont contraints à ajouter le traitement de pas mal de cas dont nous n'avions pas pensé, comme certains champs vides, des temps de trajets nuls ou négatifs, des boucles dans les trajets, ou des mauvais délimiteurs. Il reste certaines erreurs non gérées, les stations de nom *TEST_DO_NOT_USE* sont ajoutées comme des stations normales et si deux stations sont sur une même ligne *STATION;88;Château ClichySTATION;89;Saint Jaurès* cela ajoute la station 88 nommée *Château ClichySTATION*.

II. Choix des structures

A. Représentation du graphe

Le réseau de métro est modélisé sous la forme d'un graphe représenté par des listes d'adjacence. Le graphe est construit dynamiquement à partir des données du fichier d'entrée décrivant les stations et les liaisons entre celles-ci.
La structure *Graph* contient :

- *numVertices :* nombre total de sommets, de stations
- *adjLists* : tableau de pointeurs vers la liste d'adjacence
- *isDirected* : booléen qui indique si le graphe est orienté ou non

Lors de la création du graphe, la fonction *createGraph* alloue la structure principale ainsi qu'un tableau de taille égale au nombre de stations. Chaque case de ce tableau est initialisée à *NULL*, ce qui signifie qu'aucune connexion n'est encore définie.

Les arêtes sont ensuite ajoutées à l'aide de la fonction *addEdge*. Pour chaque arête, un nouveau nœud est créé et inséré en tête de la liste d'adjacence de la station source. Avant l'insertion, une vérification est effectuée afin d'éviter l'ajout de doublons.

Dans le fichier *metro.txt* les arêtes sont représentées de façon bidirectionnelle, c'est-à-dire que pour un *edge* qui va de la station 70->71, on trouve toujours un *edge* inverse. En effet, le

métro peut toujours circuler dans les deux sens entre deux stations. Le paramètre *isDirected* est donc égal à 1.

La complexité de la création de la structure du graphe est de O(V), avec V le nombre de sommets (de stations). L'ajout des arêtes est de O(E), avec E le nombre d'*edges*.

B. Listes d'adjacence

La structure de liste d'adjacence appelée *Node* contient :

- *vertex* : l'identifiant de la station de destination
- *weight* : la durée de trajet
- *next* : un pointeur vers le prochain voisin dans la liste d'adjacence

```
Station 71 : 78(2mn) -> 18(2mn) -> 72(3mn) -> 70(3mn) -> NULL
Station 72 : 73(2mn) -> 71(3mn) -> NULL
Station 73 : 74(2mn) -> 72(2mn) -> NULL
Station 74 : 20(3mn) -> 73(2mn) -> NULL
Station 75 : 110(2mn) -> 109(2mn) -> 76(2mn) -> NULL
Station 76 : 77(1mn) -> 75(2mn) -> NULL
Station 77 : 113(2mn) -> 112(2mn) -> 14(2mn) -> 76(1mn) -> NULL
```

Dans cette représentation, chaque station est associée à une liste chaînée contenant ses stations voisines ainsi que le temps de trajet correspondant. Par exemple, l'affichage *Station 71 : 78(2mn) -> 18(2mn) -> 72(3mn) -> 70(3mn) -> NULL* indique que la station 71 est directement reliée à quatre autres stations, chacune avec un poids représentant la durée du trajet.

Le fichier de réseau initial comportait des temps en nombres entiers, mais notre second fichier de réseau lui a des temps avec des nombres flottants. Par simplicité, la fonction *addEdge* transforme le poids reçu en entier.

Le choix de la liste d'adjacence permet d'éviter de stocker des connexions inexistantes. Cela rend la structure plus économe en mémoire. Chaque élément de la liste correspond à une arête sortante depuis la station considérée.

C. Annuaire des stations, tableau dynamique

Pour faire le lien entre les identifiants numériques et les noms réels des stations, nous utilisons un tableau de sommets.

La structure appelée *Station* contient l'identité d'un sommet, d'une station :

- *id* : entier unique
- *nom* : Chaîne de caractères (char*) allouée dynamiquement

```
Indice (ID) | Station
------------+-----------
0           | Gare_Centre
1           | Université
2           | Hôpital
3           | Aéroport
```

L'utilisation d'un tableau alloué dynamiquement après une première lecture du fichier permet un accès en temps constant O(1) à n'importe quelle station à partir de son ID.

D. Recherche par nom, table de hachage

Pour répondre à l'exigence de recherche efficace par nom, une table de hachage a été implémentée.

La structure utilisée appelée *HashNode* contient :

- *station* : pointeur vers la structure *Station* correspondante dans le tableau principal
- *next* : gestion des collisions par chaînage linéaire

Sans table de hachage, chercher une station par son nom nécessiterait de parcourir tout le tableau (complexité O(N)). La table de hachage permet de ramener cette complexité à un temps moyen constant O(1), ce qui est crucial pour la fluidité de l'interface utilisateur lors de la saisie des itinéraires.

III. Comparaison algorithmes de tri

Nous avons utilisé la fonction *clock* de la bibliothèque *time.h* afin d'avoir une donnée supplémentaire pour différencier les algorithmes. Lors de nos premiers tests sur le petit jeu de données, les temps de traitement des différents algorithmes de tri (tri par sélection, tri par insertion et tri rapide) variaient beaucoup d'une exécution à l'autre. Avec un nombre limité de stations, il était difficile d'observer une différence nette de performance, et il aurait fallu exécuter le code des centaines de fois ou augmenter la taille des données pour obtenir une statistique fiable.

En revanche, sur le jeu de données réel, beaucoup plus volumineux, la différence de performances entre les algorithmes devient clairement visible :

Algorithme	Comparaisons	Permutations	Temps (s)
Tri par sélection	155 628	154 735	0,000763
Tri par insertion	400 960	871	0,001758
Tri rapide	93 272	2 425	0,000579

Le tri par sélection et le tri par insertion effectuent soit un nombre de comparaisons soit de permutations très élevé, ce qui se traduit par un temps d'exécution significatif. Ces deux algorithmes ont une complexité moyenne en O(n^2), ce qui les rend inefficaces dès que le nombre d'éléments à trier devient important.

Le tri rapide récursif, avec une complexité moyenne de O(n\log n), s'en sort beaucoup mieux sur de grands ensembles de données, avec un nombre de comparaisons et de permutations beaucoup plus faible, et donc un temps d'exécution nettement inférieur. Ainsi, sur un jeu de données volumineux, la supériorité du tri rapide en termes de temps de calcul est clairement perceptible.