Projet - Rapport Algo SD

Réseau de transport et parcours de graphes

Auteurs : Adam Zekari - Alexis Miras
1A IR
2025-2026

Introduction :

Le projet consiste à modéliser un réseau de transport (métro, bus, …) sous la forme d'un graphe orienté. Les stations sont représentées par des sommets et les liaisons par des arcs pondérés par un temps de trajet. Le programme est écrit en langage C et repose sur des structures de données classiques : tableaux dynamiques, listes chaînées, tables de hachage et algorithmes de graphes (Dijkstra). Un menu interactif permet de choisir les différentes fonctionnalités proposées.

1. Chargement et affichage des stations (menu)

1.1 Lecture du fichier

Le chargement du réseau transport se fait à partir d'un fichier texte passé en argument du programme. La fonction **init_station()** lit le fichier ligne par ligne. Chaque station est alors stockée dans une structure contenant son identifiant et son nom. Une allocation dynamique est utilisée pour adapter la mémoire au nombre réel de stations présentes dans le fichier.

1.2 Affichage des informations d'une station

Le menu principal, implémenté dans **menu.c,** propose plusieurs choix accessibles par un numéro. L'utilisation de la boucle while permet de répéter le mune jusqu'au choix de sortie (0).

- Case 1 : Afficher les informations d'une station
- Case 2 : Lister les voisins d'une station
- Case 3 : Calculer un chemin minimal (Dijkstra) entre deux stations
- Case 4 : Afficher les stations triées par degré sortant
- Case 0 : Quitter le programme

1.3 Menu interactif

Le programme via **main.c** permet d'afficher les informations d'une station à partir de son identifiant ou de son nom. Les informations affichées incluent l'identifiant, le nom et le nombre de voisins sortant (degré sortant).

2. Arcs, sommets et structure du graphe

2.1 Représentation du graphe et parcours des arcs

Le réseau de transport est représenté par un graphe orienté :

- Chaque station est un sommet, représenté par une structure **SStation** contenant son identifiant et son nom.
- Chaque arc sortant est représenté par une structure **SArc** contenant la destination, le temps, et un pointeur vers l'arc suivant pour former une liste chaînée (**next_destination**).

- L'ensemble du graphe est encapsulé dans un graphe **SGraphe** qui contient un tableau de stations, un tableau de listes d'adjacence pour chaque station et le nombre total de stations.

Ce choix permet de mieux parcourir les voisins en étant efficace et d'avoir une meilleure mémoire, ce qui correspond bien au réseau de transport.

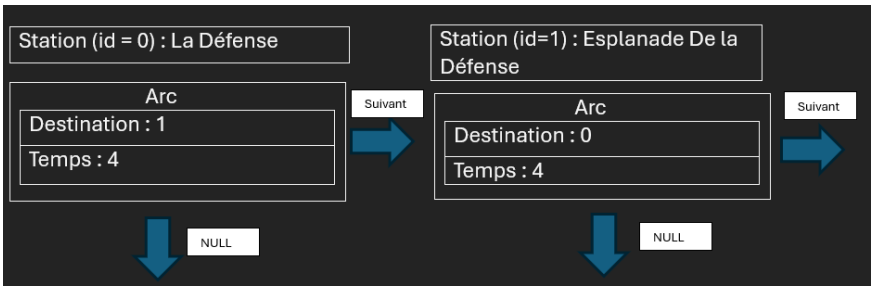

Figure 1. Représentation du graphe avec liste d'adjacence

3. Table de hachage (Hashmap)

3.1 Rôle de la Hashmap

Une table de hachage dans ce projet est utilisée pour associer rapidement le nom d'une station à son identifiant. Cela permet d'avoir une recherche rapide des stations par nom afin d'éviter de parcourir tout le tableau des stations lors d'une recherche par nom.

3.2 L'implémentation de la Hashmap

La Hashmap est implémentée à l'aide d'un tableau de listes chaînées **hash_table[TABLE_SIZE]** afin de gérer les collisions. Elle est représentée par une structure **HashNode** contenant le nom de la station, son ID et un pointeur vers le nœud suivant pour gérer les collisions. Ici, la fonction de hachage transforme le nom de la station en indice de tableau. Ainsi, des fonctions présentes dans **hash.c** permettent d'initialiser la table, d'insérer des éléments et de rechercher un identifiant à partir d'un nom.

4. Liste des voisins d'une station

Pour chaque station, les voisins accessibles directement sont obtenus en parcourant la liste d'adjacence correspondante. Chaque voisin est affiché avec :

- Son identifiant
- Son nom
- Le temps de trajet

Cette fonctionnalité correspond à la case 2 du **menu.c** et permet à l'utilisateur de connaître toutes les stations connectées à une station concernée. Le parcours devient efficace grâce à l'utilisation de la liste d'adjacence.

5. Chemin minimal – Algorithme de Dijkstra

Cette partie est représentée par une structure **SSDijkstra** qui permet de stocker les résultats et de déterminer le chemin minimal. Chaque station conserve la distance minimale depuis la station de départ et le prédécesseur **id_station_precedente** permettant de reconstruire le chemin. La fonction **dijkstra()** parcourt les listes d'adjacence du graphe afin de mettre à jour les distances et les prédécesseurs. Une fois que les résultats sont calculés et stockés, le chemin minimal est affiché dans l'ordre avec un temps minimal (temps du trajet) : station de départ -> … -> station d'arrivée.

Cette fonctionnalité correspond à la case 3 du **menu.c** et permet à l'utilisateur de connaitre le trajet optimal entre deux stations.

6. Degré des stations et tri

Les informations sur le nombre de voisins pour chaque station sont stockés dans un tableau de structures **SDegreDesStations** contenant le couple (**id_station**, **degre**). La fonction **calcul_du_degre()** remplit le tableau en parcourant la liste d'adjacence de chaque station. Ce tableau est ensuite trié pour afficher la liste des stations par degré croissant. Trois algorithmes sont implémentés dans tri.c afin de visualiser les stations triées par nombre de voisins (tri par insertion, tri par sélection et tri rapide).

Ce programme permet de comparer l'efficacité entre ces trois tris en comptant le nombre de comparaisons et le nombre de permutations. Cette fonctionnalité correspond à la case 4 et utilise le tri par sélection pour afficher les stations par degré croissant. Ainsi, le programme affiche aussi les statistiques de comparaisons et de permutations.

Nous avons opté pour le tri rapide en raison de ses meilleures performances, notamment un nombre nettement inférieur de comparaisons et de permutations par rapport aux autres algorithmes de tri.

Conclusion :

Ce projet a permis de manipuler et représenter un réseau de transport sous la forme de graphe avec la liste d'adjacence.