Projet Algorithmie et programmation

Hermès Noumbogo et Becker Obie Obolo

15 janvier 2026

Table des matières

1 Introduction

Ce rapport présente le développement d'une application de gestion de réseau de transport ferroviaire en langage C. L'objectif principal est de modéliser un graphe représentant le métro parisien afin d'offrir des fonctionnalités de recherche, de tri et d'optimisation de trajet. Ce projet met en œuvre des concepts fondamentaux tels que l'allocation dynamique, la manipulation de listes chaînées et l'analyse de la complexité algorithmique.

2 Organisation du Projet et Compilation

2.1 Arborescence des fichiers

Pour garantir une maintenance aisée et une séparation claire des responsabilités, le projet est structuré selon les standards du développement en C :

- `./` : Contient l'exécutable final `metro` après compilation.
- `./include/` : Regroupe l'ensemble des fichiers en-têtes (`.h`). Ils définissent les structures (`Node`, `Station`) et les prototypes des fonctions (Dijkstra, tris).
- `./src/` : Contient les fichiers sources (`.c`). Chaque fichier correspond à un module logique :
  - `donnees.c` : Chargement et gestion du graphe.
  - `tri.c` : Algorithmes de tri et calcul des statistiques.
  - `menus.c` : Gestion de l'affichage et de l'interface utilisateur.
  - `structures.c` : Fonctions utilitaires pour les listes et les graphes.
- `./src/main.c` : Point d'entrée du programme assurant la boucle principale.

2.2 Processus de compilation (Makefile)

La gestion de la compilation est automatisée par un fichier `Makefile`. Ce choix permet d'optimiser le temps de développement en ne recompilant que les fichiers modifiés.

Principe 2.1: Avantages du Makefile

- **Compilateur** : Utilisation de `gcc` avec les drapeaux `-Wall -Wextra` pour garantir un code sans avertissements (*warnings*).
- **Modularité** : Génération de fichiers objets (`.o`) intermédiaires.
- **Nettoyage** : Commande `make clean` pour supprimer les fichiers temporaires et maintenir un environnement propre.

3 Gestion de la Mémoire

3.1 Libération dynamique

Un soin particulier a été apporté à la gestion de la mémoire vive pour éviter toute fuite (*memory leak*).

Remarque 3.1: Nettoyage du graphe

La fonction `free_graph` parcourt chaque station du tableau et, pour chacune d'elles, libère récursivement chaque maillon (`Node`) de sa liste d'adjacence. Enfin, elle libère le tableau de stations et la structure du graphe elle-même.

4 Modélisation du Réseau et Structures de Données

4.1 Choix de la structure de données

Pour représenter le réseau, nous avons opté pour une **liste d'adjacence**. Contrairement à une matrice d'adjacence, cette structure est particulièrement adaptée aux graphes "creux" (où chaque sommet possède peu de voisins), ce qui est le cas d'un réseau de métro.

Definition 4.1: Structures fondamentales

- **Node** : Structure de liste chaînée stockant l'index de destination (`dest`), le temps de trajet (`weight`) et un pointeur vers le maillon suivant.
- **Station** : Structure pivot contenant l'ID original, le nom de la station, son degré de connectivité et le pointeur vers sa liste de voisins.
- **Tuple** : Structure exigée par le cahier des charges, facilitant le tri en couplant un ID de station à une valeur spécifique (le degré).

4.2 Chargement robuste des données

Le module `donnees.c` a été conçu pour être "incassable" face à un fichier `metro.txt` potentiellement corrompu.

Principe 4.2: Mécanisme de sécurité au chargement

Pour respecter les contraintes du projet, nous avons implémenté :

- **Indexation indirecte** : Les IDs du fichier sont mappés sur des index de tableau $[0, N-1]$. Cela permet de gérer des IDs non-consécutifs (ex : passage de l'ID 5 à 100) sans erreur de segmentation.
- **Filtrage** : Le programme ignore les lignes vides, les commentaires (`#`) et les stations malformées (ex : `STATION;;Fantôme`).
- **Bidirectionnalité** : Chaque ligne `EDGE` génère deux arcs dans le graphe pour permettre la navigation dans les deux sens de circulation.

5 Algorithmes de Tri et Statistiques

L'option 4 de l'application permet de classer les stations par degré de connectivité décroissant. Nous avons implémenté trois types de tris pour comparer leurs performances.

5.1 Analyse Comparative

Conformément aux consignes, nous comptabilisons le nombre de comparaisons (C) et de permutations (P) :

Algorithme	Complexité théorique	Observations
Tri par Sélection	$\mathcal{O}(n^2)$	C élevé et constant, P minimal.
Tri par Insertion	$\mathcal{O}(n^2)$	Très performant sur un réseau presque trié.
Tri Rapide (QuickSort)	$\mathcal{O}(n \log n)$	Le plus efficace en nombre de comparaisons.

TABLE 1 – Performances des tris sur le réseau chargé

5.2 Tri Rapide Itératif

Pour éviter les risques de *stack overflow* sur de très grands réseaux, nous avons implémenté une version itérative du QuickSort utilisant une pile manuelle pour simuler la récursion.

6 Optimisation d'Itinéraire (Algorithme de Dijkstra)

Le calcul du plus court chemin entre deux stations repose sur l'algorithme de Dijkstra.

Remarque 6.1: Fonctionnement

L'algorithme utilise un tableau de distances initialisé à l'infini (`INT_MAX`) et un tableau de prédécesseurs pour reconstruire le chemin. Grâce à notre système d'indexation, l'algorithme est capable de trouver un trajet vers n'importe quelle station valide, y compris les dernières du fichier comme *Gymnastique*.

7 Guide Utilisateur et Navigation

7.1 Lancement et Menu Principal

L'exécution s'effectue par la commande : `./metro metro.txt`. Le programme propose ensuite un menu sécurisé :

- **1 - Recherche** : Localiser une station par ID ou nom.
- **2 - Voisins** : Lister les stations adjacentes.
- **3 - Itinéraire** : Calculer le trajet le plus rapide.
- **4 - Classement** : Analyser la connectivité du réseau.

7.2 Rôle des variables globales et gestion des entrées

Pour fluidifier la communication entre le `main` et les différents modules de menus, nous avons utilisé des variables globales stratégiques :

- `int* choix` : Pointeur vers l'entier stockant la navigation de l'utilisateur. Son allocation dynamique permet une persistance des choix à travers les appels de fonctions.
- `char* name` : Zone tampon (buffer) dédiée au stockage des chaînes de caractères saisies. Elle est utilisée dès que l'utilisateur recherche une station par son nom (Menu 1 ou 3).
- `int id` : Variable de transfert utilisée pour stocker temporairement un identifiant de station saisi au clavier avant sa conversion en index interne.
- `int nb` : Stocke le nombre total de stations validées, servant de borne de sécurité pour toutes les itérations.

7.3 Définition et calcul du degré

Un point clé de notre modélisation concerne la nature des liaisons du réseau :

Principe 7.1: Symétrie et Degré

Bien que le fichier source puisse présenter les arêtes de manière directionnelle (`EDGE;A;B`), nous avons considéré le réseau de métro comme un **graphe non-orienté (symétrique)**.
À chaque lecture d'une liaison entre deux stations u et v, nous créons deux arcs : $(u \to v)$ et $(v \to u)$. En conséquence :

- Le degré d'une station correspond à la **somme de ses liaisons directes** (degré symétrique).
- Cette approche garantit que l'algorithme de Dijkstra peut calculer un itinéraire de retour et que le classement du Menu 4 reflète la connectivité réelle (interchanges) de chaque station.

8 Conclusion

Ce projet a permis de valider la maîtrise des pointeurs et des structures de données dynamiques en C. La séparation du code en modules (`tri.c`, `donnees.c`, `menus.c`) a facilité le débogage et garantit une maintenance aisée. La robustesse face aux erreurs de saisie et de fichier fait de cette application un outil fiable pour la gestion d'un réseau de transport.