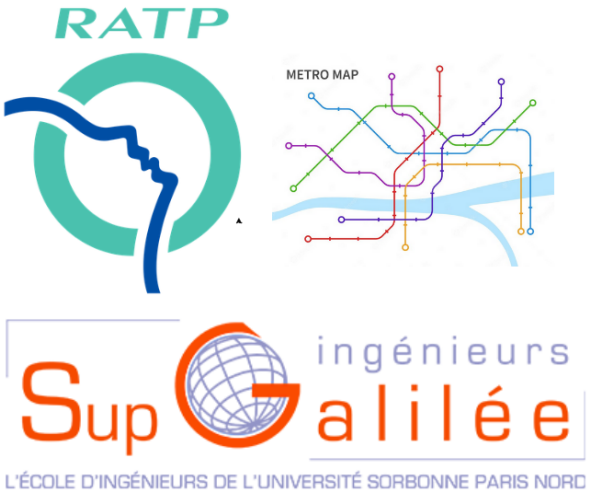

Projet - structures de données

Rapport

Calculateur d'itinéraire pour le métro

Élèves :

Vlad Alexandru Tanasov

Aïssa Pansan

Enseignant :

John Chaussard

11 février 2023

Table des matières

1 Introduction générale

Nous avons l'honneur de vous présenter le rapport de notre projet de calculateur d'itinéraire pour le métro, réalisé en tant qu'étudiants en première année d'ingénierie informatique. Ce projet s'inscrit dans le cadre de notre formation en matière de Structures de données, et a été l'occasion pour nous de mettre en pratique les concepts appris en classe.

Nous sommes également fiers de partager avec vous les résultats de notre travail acharné et espérons qu'ils démontreront notre compréhension approfondie des concepts étudiés au cours de cette matière.

Notre objectif était d'utiliser le langage C pour développer ce projet, en utilisant les pointeurs et les différents types de structures tels que les listes chainées, les piles et les files.

Ce rapport présente les détails de notre travail, expliquant les décisions prises et les difficultés rencontrées durant le développement. Il a pour objectif de synthétiser notre démarche de résolution en détaillant succinctement les étapes clé du projet.

1.1 Déroulement du projet

Nous examinons le réseau de métro de la RATP à Paris, qui comprend plus de 300 stations réparties sur 16 lignes. Pour gérer cette quantité importante de données, nous utilisons une *table de hachage* avec laquelle on classe les stations alphabétiquement.

En recourant à cette *table de hachage*, notre objectif est d'améliorer l'expérience de l'utilisateur en lui offrant la possibilité de sélectionner facilement la station de départ et d'arrivée en choisissant la première lettre de ces stations, puis en sélectionnant la station qui l'intéresse en fournissant son numéro respectif.

Pour déterminer le trajet le plus court entre les stations, nous les représentons sous la forme de sommets d'un graphe, où les arêtes représentent les lignes de métro.

Nous utilisons ensuite *l'algorithme de Dijkstra* adapté à notre structure de données pour trouver le chemin le plus court entre les stations.

2 Structure de données/Modélisation informatique du problème

2.1 Établissement de la structure de données

Nous disposons initialement de deux fichiers *csv*, l'un contenant la liste des stations de métro avec leur identifiant associé, et l'autre décrivant les relations de connexion entre les stations, comportant leur origine, leur destination et la ligne associée à cette connexion.

L'une des premières difficultés a été de ranger convenablement les données du fichier *"stations.csv"* afin d'optimiser l'accès et l'écriture : Il fallait trouver un moyen de modéliser informatiquement les relations entre chaque station.

Afin de trouver un bon compromis entre accès et écriture, l'idée a été d'adopter une *table de hachage* pour regrouper toutes ces informations en les classant alphabétiquement.

Cette structure *tab_hachage* se présente comme ceci :

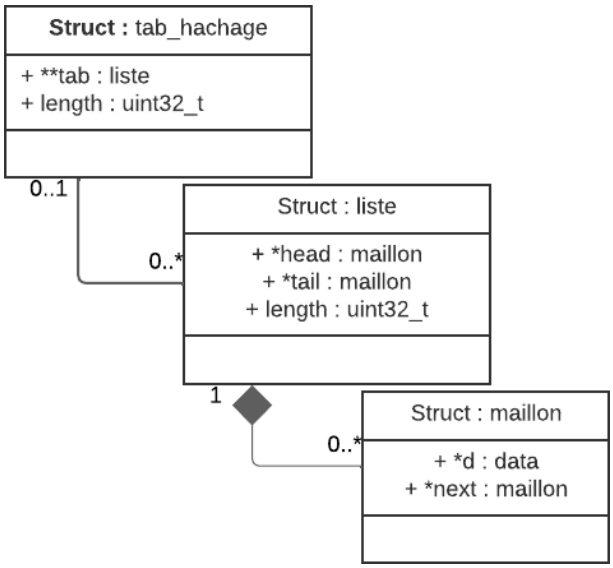

FIGURE 1 – Schéma représentant les liens entre *tab_hachage*, *liste* et *maillon*

En utilisant la structure de *liste simplement chainée*, nous stockons les données importées dans des structures appelées *maillons*. Chaque *maillon* pointe vers une structure *data* qui représente les données d'une station dans notre programme.

Cette structure *maillon* est représentée de cette façon :

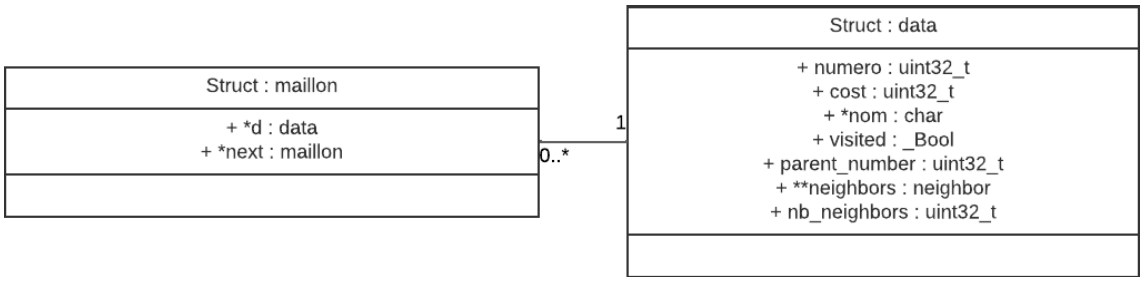

FIGURE 2 – Schéma représentant les liens entre *maillon* et *data*

Ces structures de données ci-dessus représentent les stations et leurs informations associées, il manque plus qu'à implémenter les chemins entre ces stations. Pour ce faire, on va créer une nouvelle structure nommée *neighbor*. La structure *data* quant à elle pointera vers un tableau d'adresses pointant vers la structure *neighbor*.

Cette structure se présente comme ceci :

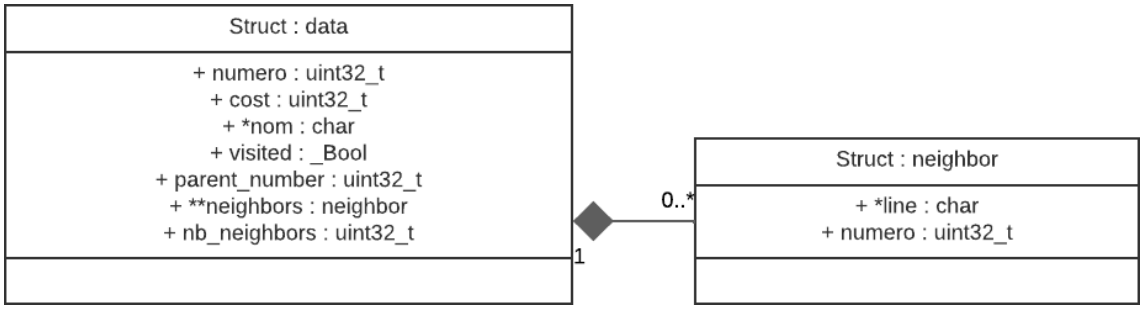

FIGURE 3 – Schéma représentant les liens entre *data* et *neighbor*

2.2 Implémentation des fonctions de récupération des données

Maintenant qu'on a défini à quoi ressemble notre modèle, nous pouvons alors implémenter les fonctions d'importations :

get_stations :

Entrées : tab_hachage* th, char* link

Sortie : tab_hachage*

Fonctionnement : Permet la récupération des données du fichier *"stations.csv"* et d'organiser ces données dans la structure *data* citée ci-dessus.

get_paths :

Entrées : tab_hachage* th, char* link

Sortie : tab_hachage*

Fonctionnement : Permet la récupération des données du fichier *"paths.csv"* et d'organiser ces données dans la structure *neighbor* citée ci-dessus.

get_data :

Entrées : tab_hachage* th, char* link_stations, char* link_paths

Sortie : tab_hachage*

Fonctionnement : Permet d'encapsuler les données récupérées des fichiers *"stations.csv"* et *"paths.csv"* dans une même *table de hachage*.

2.3 Implémentation des fonctions usuelles

Une fois que l'on a importé les données que l'on veut traiter, il ne reste plus qu'à créer des fonctions pour interagir avec ces données :

new_data :

Entrées : uint32_t* numero, char* nom

Sortie : data*

Fonctionnement : Permet de déclarer et d'initialiser un pointeur vers une structure *data*.

new_maillon :

Entrées : data* d

Sortie : maillon*

Fonctionnement : Permet de déclarer et d'initialiser un pointeur vers une structure *maillon*.

add_head :

Entrées : liste* l, data* d

Sortie : ∅

Fonctionnement : Permet d'ajouter un *maillon* en tête d'une *liste*.

rem_head :

Entrées : liste* l

Sortie : data*

Fonctionnement : Permet de libérer un *maillon* en tête d'une *liste*.

add_tail :

Entrées : liste* l, data* d

Sortie : ∅

Fonctionnement : Permet d'ajouter un *maillon* en queue d'une *liste*.

rem_tail :

Entrées : liste* l

Sortie : data*

Fonctionnement : Permet de libérer un *maillon* en queue d'une *liste*.

free_liste :

Entrées : liste* l

Sortie : ∅

Fonctionnement : Permet de libérer tous les *maillons* d'une *liste* et la *liste* elle-même.

new_tab_hachage :

Entrées : uint32_t length

Sortie : tab_hachage*

Fonctionnement : Permet de déclarer et d'initialiser un pointeur vers une structure *tab_hachage*.

hachage :

Entrées : tab_hachage* th, char c

Sortie : uint32_t

Fonctionnement : Permet de renvoyer un indice de classement pour *tab_hachage* et dans notre cas à nous : Renvoyer un indice unique en fonction de la lettre d'alphabet.

add_head_th :

Entrées : tab_hachage* th, data* d

Sortie : ∅

Fonctionnement : Permet d'ajouter un *maillon* en tête d'une *liste i* en fonction du paramètre *d* et de la fonction *hachage*.

free_th :

Entrées : tab_hachage* th

Sortie : ∅

Fonctionnement : Appelle la fonction *free_liste* pour toutes les *listes* contenues par *th* et libère la *table de hachage* elle-même.

trim :

Entrées : char* str

Sortie : ∅

Fonctionnement : Permet de supprimer les espaces dans une chaîne de caractères.

purge :

Entrées : ∅

Sortie : ∅

Fonctionnement : Permet de vider le *buffer* d'entrée, pour l'utilisation des *scanf*.

2.4 Analyse détaillée de la fonction de hachage

Comme énoncé précédemment, notre *table de hachage* va trier ses *listes* par le premier caractère des noms des stations qu'elles contiennent, analysons maintenant comment cette fonction de *hachage* fonctionne :

```
uint32_t hachage(tab_hachage *th, char c) {
 uint32_t r = c - 'A';
 if ((int32_t)c == -61) // Les É seront rangés avec les E.
   r = 'E' - 'A';
 return r % th->length;
}
```

Tout d'abord, pour tous les caractères *c*, qui correspondent au premier caractère du nom des stations, on leur retranche leur valeur *ASCII* par *"A"*, de sorte que *r* se situe entre 0 et 26. Cependant, nous avons dû gérer les stations qui commencent par le caractère *"É"*, car certaines stations comme *"École militaire"* peuvent causer des conflits avec la fonction de recherche de stations. Pour résoudre ce problème, nous avons décidé de ranger toutes les stations commençant par le caractère *"É"* dans la *liste* des caractères *"E"*. Enfin, la fonction de *hachage* retourne *r* modulo le nombre de *listes* dans la *table de hachage*, afin de garantir un accès au tableau de la table de hachage, allant de l'indice 0 à l'indice 25 ici.

2.5 Problèmes rencontrés et leurs résolutions

Avant de se fixer sur l'utilisation d'une seule *table de hachage*, on avait pensé à utiliser deux *tables de hachages* qui sont triés différemment : L'une servirait à afficher à l'utilisateur les stations triés par ordre alphabétique, et l'autre regrouperait les voisins de chaque station. Mais on s'est vite rendu compte de l'infaisabilité de cette perspective :

On avait à chaque fois des maillons qui pointaient vers deux structures différentes et les erreurs de segmentations étaient incontrôlables. Il nous est venu alors l'idée de créer la fonction *get_maillon_by_numero* pour interagir facilement avec la *table de hachage*, qui nous a beaucoup servi pour *l'algorithme de Dijkstra* et pour l'assignation des chemins aux différentes stations. De plus, l'outil **valgrind** nous a été indispensable, il s'est présenté à nous comme un outil incontournable au débogage et nous a fait économiser énormément de temps. Ainsi, par le biais de ce projet, on a compris l'indispensabilité de cet outil dans des projets de grande ampleur comme celui-ci.

3 Interface utilisateur

Au cours de la phase de développement du programme, nous avons identifié la nécessité de configurer une interface intuitive entre l'utilisateur et le programme. Cette interface est conçue pour permettre à l'utilisateur de trouver les stations souhaitées en saisissant uniquement les initiales qui les composent. Elle est également programmée pour gérer les différentes saisies de l'utilisateur, les erreurs de saisie et les situations dans lesquelles certaines données pourraient être manquantes.

3.1 Problèmes rencontrés et leurs résolutions

Cette partie fut un véritable challenge à maitriser, nous avons rencontré énormément d'erreurs de segmentation. La plupart de ces erreurs venaient tout d'abord de la récupération des lignes des fichiers *csv* : Notamment par le fait que l'on utilisait énormément de fonctions préexistantes, comme l'utilisation de la fonction *strtok*, qui nous a permis de découper les lignes en morceaux, mais celle-ci nous écrasait les précédentes valeur, il fallait ainsi une fois utilisé faire un malloc pour stocker la valeur dans une autre adresse. Bien que l'on faisait attention de libérer la mémoire une fois créée, cette pratique nous a amenés à des problèmes de segmentation qui survenait au fur et à mesure du code. Cela, nous l'avons compris grâce à l'utilisation de **valgrind**. Nous avons constaté que plusieurs personnes se plaignaient également de cette fonction et même certains la consi-

dèrent comme obsolète sur *stackoverflow*. Il nous a rapidement paru essentiel de trouver une solution à cela. Nous avons alors décidé de cloner les chaines de caractères importées avec la fonction *strcpy* qui alloue d'elle-même de la mémoire (on est plus obligé de faire de *malloc*, puisque cela est fait automatiquement). Mais le problème n'est pas réglé pour autant. Nous avons alors changé totalement de manière de faire, et on s'est mis à utiliser la fonction *sscanf* qui permet de garder une simplicité d'écriture dans notre code, étant plus intuitive que la fonction *strtok* et son token de parcours grâce à son paramètre *format* qui rend le code beaucoup plus clair.

3.2 Fonctionnement et gestion des erreurs de l'interface utilisateur

Comme un schéma vaut mieux que mille mots et pour comprendre simplement comment fonctionne notre interface utilisateur, nous avons opté pour un diagramme d'activités. Mais d'abord pour simplifier l'affichage et éviter la répétition pour que le diagramme soit le plus clair possible, nous avons créé deux blocs qui regroupent un ensemble d'instructions, que voici :

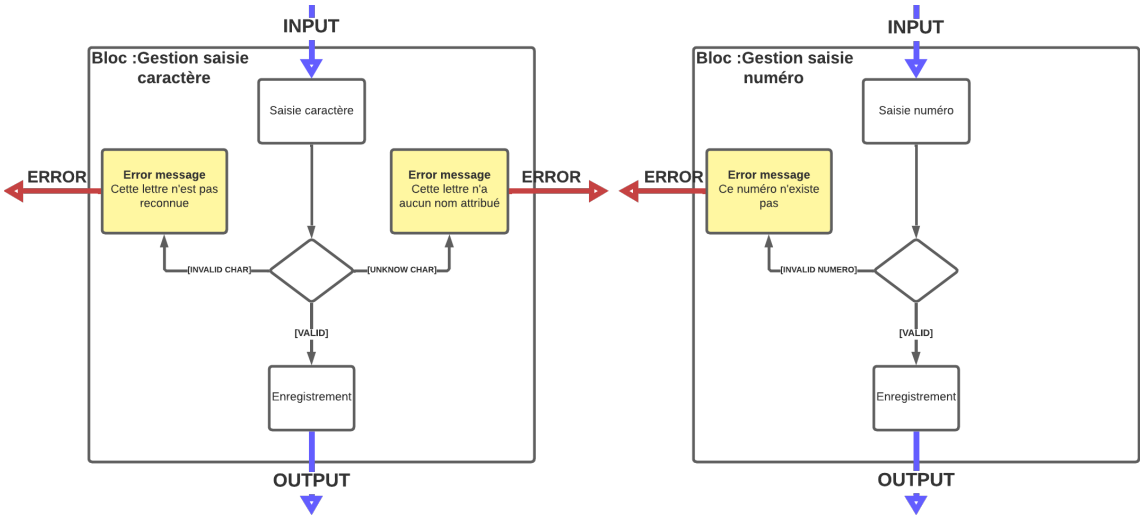

FIGURE 4 – Blocs diagramme d'activités

Maintenant voici le diagramme d'activités avec ces blocs implémentés :

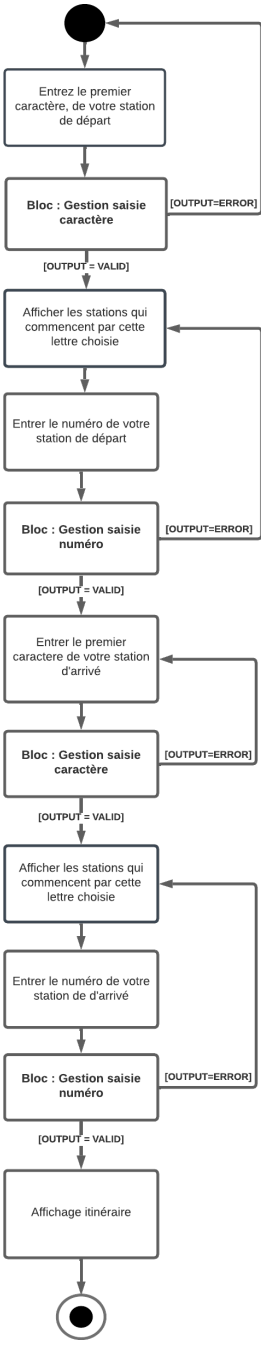

FIGURE 5 – Diagramme d'activités du programme

3.3 Détails des fonctions spécifiques à l'interface utilisateur

disp_elements :

Entrées : tab_hachage* th

Sortie : ∅

Fonctionnement : Permet d'afficher toutes les stations dans la *table de hachage* (a servi au débogage).

disp_by_letter :

Entrées : tab_hachage* th, char c

Sortie : ∅

Fonctionnement : Permet d'afficher toutes les stations d'une *liste* dans la *table de hachage* en fonction de la lettre choisie.

4 L'algorithme du plus court chemin

Dijkstra's Shortest Path algorithm est un algorithme qui permet de trouver le chemin le plus court entre deux sommets dans un graphe pondéré. Il s'agit d'un algorithme idéal pour résoudre les problèmes de recherche de chemin les plus courts. Dans notre cas, nous avons utilisé cette méthode pour trouver le trajet le plus rapide entre deux stations de métro choisies par l'utilisateur. Ce choix algorithmique nous garantit ainsi la détermination d'un unique trajet optimal, sans avoir à explorer l'ensemble des possibilités.

4.1 Fonctionnement de l'algorithme de Dijkstra

L'algorithme de Dijkstra comporte les étapes suivantes :

— **Initialisation :** On choisit un sommet de départ et on assigne un coût infini à tous les autres sommets. Le coût du sommet de départ est défini à zéro et il est établi comme pivot. Ensuite, on initialise une file.

Une fois que cette étape est faite, jusqu'à ce que tous les sommets du graphe soient explorés, on répète les étapes suivantes :

— **Relâchement :** On parcourt tous les voisins sortants du sommet pivot et on ajoute ceux dont le coût est plus faible que leur coût antérieur dans la *file*.

— **Tri :** On détermine le sommet le plus ancien avec le coût le plus faible dans la file en utilisant le principe *FIFO (first-in first-out)*. Une fois ce sommet déterminé, on le marque comme *"visited"* et on le définit comme pivot.

L'algorithme s'arrête lorsque tous les sommets ont été explorés.

4.2 Rentrons plus en détails

Tout d'abord, listons les constantes utilisés dans l'algorithme :

— **MAX_NUMBER :** correspond à l'infini dans notre cas, car il est inutile ici de stocker une valeur gigantesque pour le représenter, il est évident qu'on ne s'attendra jamais à un résultat de 9999 minutes.

— **CODE_END :** est retournée lorsqu'on ne trouve plus de voisins à cout minimal, elle permet seulement de garder une simplicité d'écriture en évitant de changer le comportement d'une fonction.

— **NIL :** vaut -1 et permet simplement de spécifier qu'un sommet n'a pas encore de numéro parent associé.

Partie initialisation :

Dans cette partie, on déclare et initialise une *liste* que l'on nomme *file* grâce à la fonction *new_liste*. Ensuite, à l'aide de la fonction *"get_maillon_by_numero"*, on récupère le maillon pivot qui correspond à la gare de départ. On initialise le coût de ce pivot à zéro. Enfin, on initialise une variable index_min à **CODE_END** pour éviter d'avoir un message d'erreur lorsque l'utilisateur spécifie un point de départ et une destination identiques. Ensuite, on initialise une chaîne de caractère qui nous permet de sauvegarder le nom de la ligne du sommet parent pour nous permettre d'adapter les coûts des chemins

des sommets suivants. Cette chaîne de caractères est vide au départ puisque le sommet de départ n'a pas encore de sommet parent.

Partie Relâchement :

Cette partie correspond à ce qui se trouve dans la boucle *for* d'indice *j*. Cette boucle parcourt tous les voisins sortants du sommet pivot. Pour chaque voisin, on regarde si son coût partant de ce pivot est inférieur à son coût précédent (si existant). Si ce coût est inférieur ou si ce coût n'existe pas encore (on le sait en regardant si son numéro parent vaut **NIL** ou non) alors, on ajoute ce voisin dans la *file*.

Partie Tri :

Dans cette partie, on initialise la propriété *"visited"* du pivot à 1. Puis, on fait appel à la fonction *get_minimum_cost_shortest_path* pour trouver le sommet le plus ancien dans la *file* avec le coût le moins élevé. On initialise ensuite ce sommet comme pivot.

4.3 Détails des fonctions spécifiques à l'utilisation de Dijsktra

get_liste_by_numero :

Entrées : tab_hachage* th, uint32_t numero

Sortie : liste*

Fonctionnement : Permet de renvoyer la *liste* correspondant à la lettre associée au numéro demandé. Puisque les *listes* sont ordonnées, il suffit de trouver une tête avec un numéro plus grand que le numéro demandé. Cette fonction ne joue qu'un rôle intermédiaire pour la fonction *get_maillon_by_numero*.

get_maillon_by_numero :

Entrées : tab_hachage* th, uint32_t numero

Sortie : maillon*

Fonctionnement : Fait appel à la fonction *get_liste_by_numero* pour récupérer

la *liste* dans laquelle le *maillon* demandé se situe puis renvoie ce *maillon* en parcourant cette *liste* qui est ordonnée.

get_number_of_neighbors :

Entrées : char* link

Sortie : uint32_t*

Fonctionnement : Parcoure le fichier *"paths.csv"* et pour chaque numéro ajoute 1 à un tableau d'entiers à l'index *(numero-1)*, pour à la fin avoir le nombre de voisins de chaque station.

find_maillon :

Entrées : liste* l, uint32_t position

Sortie : maillon*

Fonctionnement : Permet, grâce à une position, de trouver le *maillon* correspondant dans une *liste* et de l'envoyer. Cette fonction n'est qu'un intermédiaire pour la fonction *get_minimum_cost_shortest_path*.

rem_maillon :

Entrées : liste* l, uint32_t position

Sortie : data*

Fonctionnement : Permet de libérer le *maillon* correspondant grâce à une position dans une *liste*. Cette fonction n'est qu'un intermédiaire pour la fonction *get_minimum_cost_shortest_path*.

get_minimum_cost_shortest_path :

Entrées : liste* file

Sortie : int32_t

Fonctionnement : Parcoure tous les *maillons* de la *file* grâce à la fonction *find_maillon* puis sélectionne celui dont le coût et l'indice dans la *file* est minimum. Enfin, ce *maillon*

est supprimé de la *file* à l'aide de la fonction *rem_ maillon* et est renvoyé son numéro. Dans le cas où il ne trouve aucun *maillon*, il retourne **CODE_END** pour mettre fin au programme.

4.4 Explication de la gestion des coûts dans Dijkstra

Soit un graphe $G = (V, E)$ avec un ensemble de nœuds V et un ensemble d'arêtes E. Pour chaque arête (u, v) appartenant à E, son coût peut être soit de 1 ou de 6. Ce coût représente le temps que l'arête va ajouter au trajet total.

Le coût de chaque arête est déterminé en fonction de l'arête précédente. Si l'arête (u, v) représente la même ligne que l'arête suivante (v, w), son coût sera de 1, sinon son coût sera de 6.

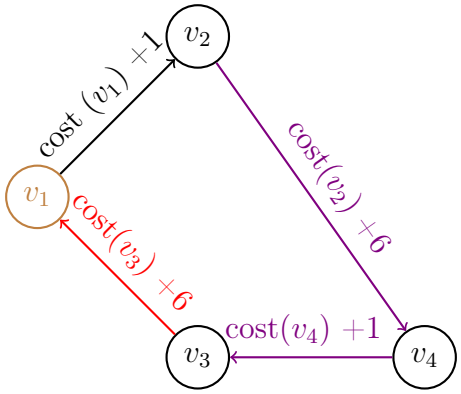

FIGURE 6 – Modèle simplifié illustrant le fonctionnement des coûts avec v_1 comme pivot

Dans ce modèle simplifié, la couleur représente la ligne. On part du sommet v_1 pour arriver au sommet v_2 et on ajoute 1 au coût de v_1, cela s'explique, car le sommet v_1 n'a pas de parent, il a alors une ligne précédente *"joker"*. Puis, on ajoute 6 au coût, car la couleur change du sommet v_2 au sommet v_4, et on ajoute ensuite 1 au coût en allant vers v_3. Enfin, l'algorithme s'arrête puisqu'il a parcouru tous les sommets. Ici l'arête (v_1, v_3) n'est pas parcouru.

4.5 Problèmes rencontrés et leurs résolutions

Avant d'arriver à notre *algorithme de Dijkstra*, nous avons eu l'idée d'interagir directement avec les sommets dans la *table de hachage* : c'est-à-dire qu'à chaque itération les seules opérations que l'on faisait était de modifier directement les paramètres des stations dans la *table de hachage*. On s'est rendu compte que cela n'était pas suffisant et qu'il fallait garder la notion d'ordre dans nos actions, en effet pour que *Dijkstra* marche, il faut garder l'ordre de parcours des stations, car lorsque l'on a une nouvelle station avec un coût minime égal au coût d'une plus vieille station, il faut sélectionner la plus vieille pour que *Dijkstra* fonctionne. Nous avons alors eu l'idée d'implémenter une *file* qui garde cette notion d'ordre. Cela nous a paru avantageux de choisir cette option, puisque nous avons déjà une structure *liste* implémentée qui partage les mêmes propriétés.

5 Résultats et améliorations possibles

5.1 Résultats

En s'exécutant, et en renseignant les 1ers caractères des stations de départ et d'arrivée, notre programme affiche dans la console le chemin à emprunter station par station et renseigne à la fin le temps de parcours. Voici ci-contre un exemple d'exécution où l'on part de *Porte Dauphine* et l'on arrive à *Nation* :

```
Récap trajet :
 (départ : Porte Dauphine(217) => destination : Nation(179))
Itinéraire :
(Reuilly - Diderot(248) => Nation(179)) ligne 1
(Gare de Lyon(105) => Reuilly - Diderot(248)) ligne 1
(Châtelet(55) => Gare de Lyon(105)) ligne 14
(Louvre - Rivoli(148) => Châtelet(55)) ligne 1
(Palais-Royal - Musée du Louvre(188) => Louvre - Rivoli(148)) ligne 1
(Tuileries(292) => Palais-Royal - Musée du Louvre(188)) ligne 1
(Concorde(64) => Tuileries(292)) ligne 1
(Champs-Élysées - Clemenceau(45) => Concorde(64)) ligne 1
(Franklin D. Roosevelt(97) => Champs-Élysées - Clemenceau(45)) ligne 1
(George V(108) => Franklin D. Roosevelt(97)) ligne 1
(Charles de Gaulle - Étoile(48) => George V(108)) ligne 1
(Victor Hugo(297) => Charles de Gaulle - Étoile(48)) ligne 2
(Porte Dauphine(217) => Victor Hugo(297)) ligne 2

Il vous faudra 28 minutes pour arriver à destination.
```

FIGURE 7 – Itinéraire de *Porte Dauphine* à *Nation*

Malgré la complexité de *l'algorithme de Dijkstra*, le programme se montre performant en déterminant rapidement et avec précision le trajet optimal pour arriver à la destination spécifiée.

Cependant, il est important de noter que l'algorithme n'est pas parfait. Dans certains cas, notamment lorsqu'un trajet implique une correspondance entre deux lignes, le programme peut recommander un changement de ligne alors qu'il aurait été plus efficace de rester dans la même ligne. Cela peut conduire à des changements de ligne inutiles. Le montre le schéma ci-dessous :

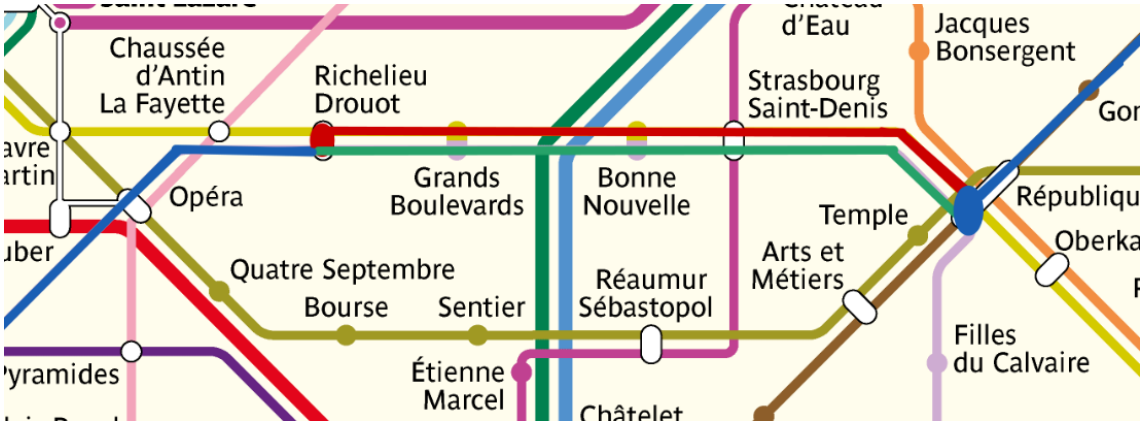

FIGURE 8 – Portion du trajet problématique

Pour un trajet au départ de *Boulogne Pont de St-Cloud* jusqu'à *Mairie des Lilas*, l'algorithme propose de changer de la ligne 8 à ligne 3 pour effectuer une nouvelle correspondance à *République* pour la ligne 11, ce trajet est représenté en rouge. Ce changement est inutile, car la ligne 8 a déjà une correspondance avec la ligne 11 à *République*, ce qui est représenté en vert.

En général, l'efficacité des itinéraires recommandés peut être remise en question, surtout dans le cas de trajets longs impliquant plusieurs correspondances.

5.2 Améliorations possibles

L'une des améliorations potentielles de ce projet aurait été d'afficher l'itinéraire du haut vers le bas pour l'utilisateur. Cela aurait pu être accompli en créant un tableau pour stocker les informations relatives à l'itinéraire, puis en inversant ce tableau avant

d'afficher ces informations. Une interface graphique aurait également pu être mise en place pour améliorer l'expérience utilisateur, mais étant donné les contraintes de temps et les objectifs du cours, l'accent a été mis sur l'optimisation de l'algorithme plutôt que sur l'amélioration de l'interface utilisateur.

6 Conclusion

Ce projet fut enrichissant à bien des égards, il nous a permis d'approfondir nos connaissances sur les structures de données telles que *la table de hachage*, mais également notre compréhension sur l'utilisation des outils de débogages tels que *valgrind*. Grâce à ce tout, nous avons véritablement pu améliorer le programme au fur et à mesure des échecs pour qu'il affiche des trajets de plus en plus pertinents jusqu'à nous approprier le fonctionnement même de l'algorithme de Dijkstra. Bien que des incohérences restent encore à résoudre, nous sommes très fiers de ce que nous avons accompli. Cette expérience a été très formative pour nous, et nous sommes reconnaissants de ces opportunités de développement.

7 Annexe

tab_hachage.h

```
#include "limits.h"
#include <assert.h>
#include <ctype.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BIG_LENGTH 64
#define SMALL_LENGTH 10

#define NB_ALPHABET 26
#define NB_STATIONS 304
#define NB_ARETES 734
```

```
#define MAX_NUMBER 9999
#define NIL -1

typedef struct _neighbor {
  char *line;
  int32_t numero;
} neighbor;

typedef struct _data {
  uint32_t numero;
  uint32_t cost;
  char *nom;
  _Bool visited;
  int32_t parent_numero;
  neighbor **neighbors;
  uint32_t nb_neighbors;
} data;

typedef struct _maillon {
  data *d;
  struct _maillon *next;
} maillon;

typedef struct _liste {
  maillon *head;
  maillon *tail;
  uint32_t length;
} liste;

typedef struct _tab_hachage {
  liste **tab;
  uint32_t length;
} tab_hachage;

data *new_data(uint32_t numero, char *nom);
maillon *new_maillon(data *d);

liste *new_liste();

void add_head(liste *l, data *d);
data *rem_head(liste *l);

void *add_tail(liste *l, data *d);
data *rem_tail(liste *l);

void free_liste(liste *l);

void disp_elements(tab_hachage *th);
liste *disp_by_letter(tab_hachage *th, char c);
```

```
tab_hachage *new_tab_hachage(uint32_t length);
uint32_t hachage(tab_hachage *th, char c);
void add_head_th(tab_hachage *th, data *d);
void free_th(tab_hachage *th);

void trim(char *str);

maillon *get_maillon_by_numero(tab_hachage *th, uint32_t numero);
liste *get_liste_by_numero(tab_hachage *th, uint32_t numero);

uint32_t *get_number_of_neighbors(char *link);

maillon *find_maillon(liste *l, uint32_t position);
data *rem_maillon(liste *l, uint32_t position);
```

tab_hachage.c

```
#include "tab_hachage.h"

data *new_data(uint32_t numero, char *nom) {
  data *d = malloc(sizeof(data));
  assert(d != NULL);
  d->numero = numero;
  d->nom = nom;
  d->visited = 0;
  d->cost = MAX_NUMBER;
  d->neighbors = NULL;
  d->parent_numero = NIL;
  return d;
}
maillon *new_maillon(data *d) {
  maillon *m = malloc(sizeof(maillon));
  assert(m != NULL);
  m->d = d;
  m->next = NULL;
  return m;
}

liste *new_liste() {
  liste *r = malloc(sizeof(liste));
  assert(r != NULL);
  r->length = 0;
  r->head = NULL;
  r->tail = NULL;
  return r;
}
```

```
void add_head(liste *l, data *d) {
  maillon *m = new_maillon(d);
  m->next = l->head;
  l->head = m;

  if (l->length == 0)
    l->tail = m;
  l->length++;
}

data *rem_head(liste *l) {
  maillon *t = l->head;
  data *r = t->d;

  l->head = l->head->next;
  free(t);
  l->length -= 1;
  if (l->length == 0)
    l->tail = NULL;
  return r;
}

void *add_tail(liste *l, data *d) {
  maillon *m = new_maillon(d);
  m->next = NULL;
  if (l->length > 0)
    l->tail->next = m;
  else
    l->head = m;
  l->tail = m;

  l->length++;
}

data *rem_tail(liste *l) {
  maillon *avant_dernier = NULL;
  maillon *dernier = l->head;

  while (dernier != l->tail) {
    avant_dernier = dernier;
    dernier = dernier->next;
  }
  data *d = dernier->d;
  l->tail = avant_dernier;
  l->tail->next = NULL;
  free(dernier);
  l->length -= 1;

  if (l->length == 0)
    l->head = NULL;
```

```
  return d;
}

void free_liste(liste *l) {
  uint32_t i;
  while (l->length > 0) {
    data *d = rem_head(l);
    for (i = 0; i < d->nb_neighbors; i++) {
      free(d->neighbors[i]->line);
      free(d->neighbors[i]);
    }
    free(d->nom);
    free(d->neighbors);
    free(d);
  }
  free(l);
}

tab_hachage *new_tab_hachage(uint32_t length) {
  uint32_t i;
  tab_hachage *th = malloc((sizeof(tab_hachage)));
  assert(th != NULL);
  th->tab = malloc(length * sizeof(liste *));
  assert(th->tab != NULL);
  for (int i = 0; i < length; i++)
    th->tab[i] = new_liste();
  th->length = length;
  return th;
}

void free_th(tab_hachage *th) {
  uint32_t i;
  for (i = 0; i < th->length; i++)
    free_liste(th->tab[i]);
  free(th->tab);
  free(th);
}

uint32_t hachage(tab_hachage *th, char c) {
  uint32_t r = c - 'A';
  if ((int32_t)c == -61) // Les É seront rangés avec les E.
    r = 'E' - 'A';
  return r % th->length;
}
void add_head_th(tab_hachage *th, data *d) {
  uint32_t p = hachage(th, d->nom[0]);
  add_head(th->tab[p], d);
}

void disp_elements(tab_hachage *th) {
```

```
  uint32_t i;
  for (i = 0; i < th->length; i++) {
    maillon *m = th->tab[i]->head;
    while (m != NULL) {
      printf("\n%s, %d", m->d->nom, m->d->numero);
      m = m->next;
    }
  }
}

liste *disp_by_letter(tab_hachage *th, char c) {
  liste *l = th->tab[hachage(th, c)];

  maillon *m = l->head;
  while (m != NULL) {
    printf("\n%s, %d", m->d->nom, m->d->numero);
    m = m->next;
  }
  return l;
}

liste *get_liste_by_numero(tab_hachage *th, uint32_t numero) {
  uint32_t i = 0;

  while (i < th->length) {
    if (th->tab[i] != NULL && th->tab[i]->head != NULL) {
      uint32_t n = th->tab[i]->head->d->numero;
      if (numero <= n)
        break;
    }
    i++;
  }

  if (i == th->length)
    return NULL;
  else
    return th->tab[i];
}

maillon *get_maillon_by_numero(tab_hachage *th, uint32_t numero) {
  liste *l = get_liste_by_numero(th, numero);
  if (l == NULL || l->head == NULL)
    return NULL;
  int i = 0;
  maillon *m = l->head;
  while (m != NULL && numero < m->d->numero) {
    m = m->next;
  }

  return m;
```

```
}

void trim(char *str) {
  char *end;
  while (isspace((unsigned char)*str))
    str++;
  if (*str == 0)
    return;
  end = str + strlen(str) - 1;
  while (end > str && isspace((unsigned char)*end))
    end--;
  end[1] = '\0';
}

uint32_t *get_number_of_neighbors(char *link) {

  FILE *aretes_f;
  char c[BIG_LENGTH];
  char source[SMALL_LENGTH];
  char destination[SMALL_LENGTH];
  char ligne[SMALL_LENGTH];
  uint32_t *stations_n = malloc(sizeof(uint32_t) * NB_STATIONS);
  uint32_t i = 0;
  for (i = 0; i < NB_STATIONS; i++)
    stations_n[i] = 0;
  aretes_f = fopen(link, "r");

  fseek(aretes_f, 0, SEEK_SET);

  if (aretes_f == NULL) {
    printf("%s", "Le fichier n'existe pas !");
    return NULL;
  }

  while (fgets(c, sizeof(c), aretes_f)) {
    sscanf(c, "%[^,],%[^,],%[^\n]", source, destination, ligne);

    trim(ligne);
    if (strcmp(ligne, "Ligne") != 0) // Pour ne pas prendre la 1ere ligne
    {
      trim(destination);
      trim(source);
      uint32_t destination_format = strtol(destination, NULL, 0);
      uint32_t source_format = strtol(source, NULL, 0);

      stations_n[source_format - 1]++;
    }
  }

  return stations_n;
```

```
}

maillon *find_maillon(liste *l, uint32_t position) {
  maillon *r = l->head;
  uint32_t i;

  if (position >= l->length) {
    return NULL;
  }
  for (i = 0; i < position; i++) {
    r = r->next;
  }
  return r;
}

data *rem_maillon(liste *l, uint32_t position) {
  maillon *m, *r;
  data *d;
  if (position >= l->length) {
    return NULL;
  } else if (position == 0) {
    return rem_head(l);
  } else if (position == l->length - 1) {
    return rem_tail(l);
  } else {
    r = find_maillon(l, position - 1);
    m = r->next;
    d = m->d;
    r->next = m->next;
    free(m);
    l->length -= 1;
    return d;
  }
}
```

dijkstra.h

```
#include "tab_hachage.h"

#define COST_PER_LINE 5
#define COST_PER_STATION 1
#define CODE_END -99

int32_t get_minimum_cost_shortest_path(liste *file);
void dijkstra(tab_hachage *th, uint32_t source);
```

dijkstra.c

```
#include "dijkstra.h"

int32_t get_minimum_cost_shortest_path(liste *file) {
  uint32_t min_cost = MAX_NUMBER, min_index = MAX_NUMBER, i, source;
  maillon *r = NULL, *m = NULL, *prev = NULL;
  data *d;

  for (i = 0; i < file->length; i++) {
    r = find_maillon(file, i);
    if (r->d->cost < min_cost && i < min_index) {
      min_cost = r->d->cost;
      min_index = i;
    }
  }
  if (min_index == 0)
    d = rem_maillon(file, min_index);
  else
    d = rem_maillon(file, min_index + 1);

  if (d == NULL)
    return CODE_END;
  source = d->numero;

  free(d);
  return source;
}

void dijkstra(tab_hachage *th, uint32_t source) {

  uint32_t cost_min, index_min, i, j, n;

  liste *file = new_liste();
  maillon *p = get_maillon_by_numero(th, source);

  p->d->cost = 0;
  index_min = CODE_END;
  char *last_line = "";

  while (p != NULL) {
    for (j = 0; j < p->d->nb_neighbors; j++) {
      maillon *neighbor = get_maillon_by_numero(th, p->d->neighbors[j]->numero);
      uint32_t new_cost = p->d->cost + COST_PER_STATION;
      if (strlen(last_line) > 0) {
        if (strcmp(last_line, p->d->neighbors[j]->line) != 0) {
          new_cost += COST_PER_LINE;
        }
      }
      if (neighbor->d->cost >= new_cost) {
```

```
        neighbor->d->cost = new_cost;
        neighbor->d->parent_numero = p->d->numero;
        if (neighbor->d->parent_numero != NIL) {
          data *d2 = new_data(neighbor->d->numero, neighbor->d->nom);
          d2->cost = neighbor->d->cost;
          add_tail(file, d2);
        }
      }
      if (neighbor->d->parent_numero == NIL) {
        neighbor->d->parent_numero = p->d->numero;
        data *d1 = new_data(neighbor->d->numero, neighbor->d->nom);
        d1->cost = neighbor->d->cost;
        add_tail(file, d1);
      }
    }
    p->d->visited = 1;

    index_min = get_minimum_cost_shortest_path(file);
    if (index_min == CODE_END)
      break;

    maillon *temp = get_maillon_by_numero(th, index_min);
    maillon *parent = get_maillon_by_numero(th, temp->d->parent_numero);

    for (j = 0; j < parent->d->nb_neighbors; j++) {
      if (parent->d->neighbors[j]->numero == index_min)
        last_line = parent->d->neighbors[j]->line;
    }
    p = temp;
  }
  free_liste(file);
}
```

main.c

```
#include "tab_hachage.h"

void purge() {
  int c = 0;
  while ((c = getchar()) != '\n' && c != EOF)
    ;
}

tab_hachage *get_stations(tab_hachage *th, char *link) {
  FILE *stations_f;
  char c[BIG_LENGTH];
  char nom[BIG_LENGTH];
  char numero[BIG_LENGTH];
```

```
  stations_f = fopen(link, "r");

  if (stations_f == NULL) {
    printf("%s", "Le fichier n'existe pas !");
    return NULL;
  }

  while (fgets(c, sizeof(c), stations_f)) {
    sscanf(c, "%[^,],%[^\n]", nom, numero);

    char *nom_format = strdup(nom);
    uint32_t numero_format = strtol(numero, NULL, 0);

    data *d = new_data(numero_format, nom_format);
    if (strcmp(nom_format, "Nom")) // Pour passer la 1ere ligne
    {
      add_head_th(th, d);
    }
  }

  fclose(stations_f);
  return th;
}
tab_hachage *get_paths(tab_hachage *th, char *link) {
  FILE *paths_f;
  char c[BIG_LENGTH];
  char source[SMALL_LENGTH];
  char destination[SMALL_LENGTH];
  char ligne[SMALL_LENGTH];

  uint32_t *neighbors_n = get_number_of_neighbors(link);
  uint32_t i, j, k;
  for (i = 1; i <= NB_STATIONS; i++) {
    uint32_t length = neighbors_n[i - 1];

    if (length > 0) {
      maillon *m = get_maillon_by_numero(th, i);
      m->d->neighbors = malloc(sizeof(neighbor *) * length);
      assert(m->d->neighbors != NULL);

      for (j = 0; j < length; j++) {
        neighbor *v = malloc(sizeof(neighbor));
        assert(v != NULL);
        v->line = "";
        v->numero = 0;
        m->d->neighbors[j] = v;
      }
      m->d->nb_neighbors = 0;
    }
```

```
  }
  free(neighbors_n);

  paths_f = fopen(link, "r");

  if (paths_f == NULL) {
    printf("%s", "Le fichier n'existe pas !");
    return NULL;
  }

  fseek(paths_f, 0, SEEK_SET);

  while (fgets(c, sizeof(c), paths_f)) {

    sscanf(c, "%[^,],%[^,],%[^\n]", source, destination, ligne);

    trim(ligne);
    if (strcmp(ligne, "Ligne") != 0) // Pour ne pas prendre la 1ere ligne
    {

      char *ligne_format = strdup(ligne);
      trim(destination);
      trim(source);
      uint32_t destination_format = strtol(destination, NULL, 0);
      uint32_t source_format = strtol(source, NULL, 0);

      maillon *temp = get_maillon_by_numero(th, source_format);

      temp->d->neighbors[temp->d->nb_neighbors]->line = ligne_format;
      temp->d->neighbors[temp->d->nb_neighbors]->numero = destination_format;
      temp->d->nb_neighbors++;
    }
  }

  fclose(paths_f);
  return th;
}

tab_hachage *get_data(char *link_stations, char *link_paths) {

  tab_hachage *th = new_tab_hachage(NB_ALPHABET);
  th = get_stations(th, link_stations);
  th = get_paths(th, link_paths);
  return th;
}

int32_t get_numero(tab_hachage *th, _Bool start) {
  char c;
  uint32_t k;
```

```
if (start)
  printf("Entrez s'il vous plat la premire lettre de votre station de "
         " dpart : ");
else
  printf("Entrez s'il vous plat la premire lettre de votre station "
         "d' arrive : ");
scanf(" %c", &c);
c = toupper(c);

if (!isalpha(c)) {
  purge();
  printf("\nCette lettre n'est pas reconnue ...\n\n");
  return -1;
}
liste *l = disp_by_letter(th, c);

if (l->head == NULL || l->length == 0) {
  printf("\nCette lettre n'a aucun nom attribu ...\n\n");
  return -1;
}

_Bool isPresent = 0;

while (!isPresent) {
  if (start)
    printf("\n\nChoisissez parmi la liste prsente le numro de votre "
           "station de "
           " dpart : ");
  else
    printf(
        "\n\nChoisissez parmi la liste prsente le numro de votre station "
        "d' arrive : ");
  purge();
  scanf("%d", &k);
  isPresent = 0;
  maillon *m = l->head;
  while (m != NULL) {
    if (m->d->numero == k) {
      isPresent = 1;
      break;
    }
    if (m->next == NULL) {
      printf("\nCe numro n'existe pas, ressayez s'il vous "
             " plat , voici la liste  nouveau :\n");

      disp_by_letter(th, c);
    }
    m = m->next;
  }
}
```

```
  if (isPresent)
    printf("\nMerci, c'est not !\n\n");

  return k;
}

int main() {
  tab_hachage *th = get_data("stations.csv", "paths.csv");

  uint32_t num_depart = -1;
  uint32_t num_arrivee = -1;

  while (num_depart == -1)
    num_depart = get_numero(th, 1);
  while (num_arrivee == -1)
    num_arrivee = get_numero(th, 0);

  dijkstra(th, num_depart);

  maillon *m_depart = get_maillon_by_numero(th, num_depart);
  maillon *m_arrivee = get_maillon_by_numero(th, num_arrivee);
  printf("\nRcap trajet : \n (dpart : %s(%d) => destination : %s(%d))",
         m_depart->d->nom, num_depart, m_arrivee->d->nom, num_arrivee);

  uint32_t max_cost = 0;

  printf("%s", "\nItinraire : \n");
  uint32_t i, j;
  uint32_t minutes;
  char *line = malloc(sizeof(char) * SMALL_LENGTH);

  data *d = m_arrivee->d;
  line[0] = '*';

  while (d->numero != num_depart) {
    maillon *parent = get_maillon_by_numero(th, d->parent_numero);

    if (parent == NULL)
      break;

    for (i = 0; i < parent->d->nb_neighbors; i++) {
      if (parent->d->neighbors[i]->numero == d->numero) {
        max_cost++;
        if (line[0] != '*') {
          if (strcmp(line, parent->d->neighbors[i]->line) != 0) {
            max_cost += 5;
          }
        }
        strcpy(line, parent->d->neighbors[i]->line);
      }
```

```
    }
    printf("(%s(%d) => %s(%d)) ligne %s\n", parent->d->nom, parent->d->numero,
           d->nom, d->numero, line);
    d = parent->d;
  }
  printf("\nIl vous faudra %d minutes pour arriver  destination.", max_cost);

  free(line);
  free_th(th);
  return EXIT_SUCCESS;
}
```