Institut Galilée - Université Paris 13

Structures de données
Projet - Calculateur d'itinéraire pour le métro
Rapport

Calcul du plus court chemin

Élève :
Sidane Alp

Enseignant :
John Chaussard

11 février 2023

Table des matières

1 Introduction

Dans le cadre de la matière structure de données et programmation avancée, ce rapport présente une solution a un problème présenté par l'encadrant de la matière, Monsieur John Chaussard pour les étudiants en première année d'ingénierie informatique de Sup Galilée.

1.1 Présentation du problème

Ce problème consiste à trouver le plus court chemin d'un point A un point B dans le réseau métropolitain de Paris, en prenant en compte les changements de station.

Chaque connexion directe entre deux stations met une minute. Chaque changement de lignes de métro met 5 minutes. Nous allons résoudre ce problème avec une structure de graphe.

1.2 Plan

Nous allons dans un premier temps, modéliser notre problème à l'aide d'un graphe. On montrera 2 modélisations possibles, une qui sera un échec, une autre qui fonctionne. Dans un deuxième temps, nous allons présenter rapidement Dijkstra et sa compatibalité avec notre graphe. Enfin nous allons implémenter notre programme en C et expliquer point par point notre programme.

2 Modélisation sous forme de Graphe du problème

On décide de modéliser le problème à l'aide d'un graphe, pour pouvoir ensuite appliquer un algorithme du plus court chemin entre une station A et une station B.

On suppose qu'il y a n stations de métro et m connexions entre chaque ligne.

On note $S = \{s_1, s_2, ..., s_n\}$, l'ensemble des sommets représentant une station de métro. On note $V = \{v_1, v_2, ..., v_m\}$, l'ensemble des arcs représantant une liaison directe entre deux stations de métros. On décide alors de modéliser le réseau de métro par un graphe orienté $G = (S, V)$.

Chaque arc de V aura un poids de 1, qui représente la minute du trajet entre les deux stations de métro. De plus, chaque arc sera coloré, et chaque coloration d'arc modélisera une ligne de métro qui lui est unique. Ainsi, on pose la fonction C, la fonction qui pour tout $v \in V$ renvoie la coloration de l'arête v.

Pour chaque schéma de Graphe, chaque sommet représente une station, chaque arc coloré représente une liaison entre deux stations. et sa couleur représente la ligne de métro auquel appartient la liaison entre les deux stations. Par exemple, tous les arcs bleus correspondent au "métro 7b", les arcs rouge le "métro 4" etc....

Les coûts de chaque arc sont représentés par un nombre au-dessus de l'arc (ici "+1"). Cela représente le nombre de minutes nécessaires à l'usager pour aller à la station pointée par l'arc.

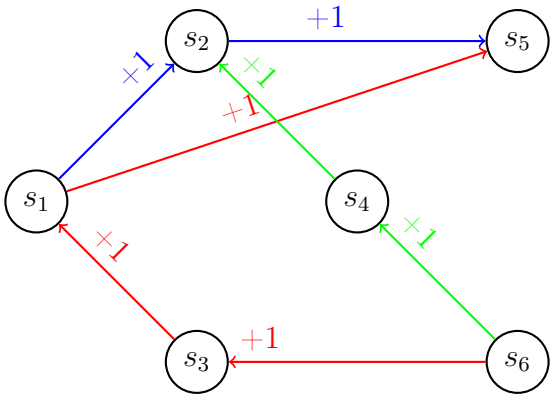

FIGURE 1 – Exemple d'illustration du modèle

2.1 Modélisation à l'aide d'arc virtuels

Cependant, ce modèle n'est pas complet puisqu'il ne prend en compte le temps d'attente de 5 minutes en gare pour changer de ligne de métro.

Pour modéliser ce temps d'attente de 5 minutes, on va poser l'ensemble des arêtes virtuelles $F = \{f_1, f_2, .., f_q\}$, avec $q \in \mathbb{N}$ et $f_1 = (s_i s_j)$ tel que pour tout $i, j \in \mathbb{N}$, il existe un arc allant du sommet s_i à un sommet s_k (avec $k \in \mathbb{N}$) et qu'il existe un autre arc allant de s_k à s_j, avec $(c_i c_k)$, une arête de coloration différente que $(c_k c_j)$. Plus formellement, on a :

$$F = \{v = (s_i s_j) | \forall (i,j) \in \mathbb{N}^2 \; \exists k \in \mathbb{N} \; ((s_i s_k) \in V) \; et \; ((s_k s_j) \in V) \; et \; (C(s_i s_k) \neq C(s_k s_j))\}$$

Ces arêtes virtuelles auront alors pour propriété d'avoir un poids de 7 qui modélise 5 minutes d'attente et 2 minutes de parcours de graphe.

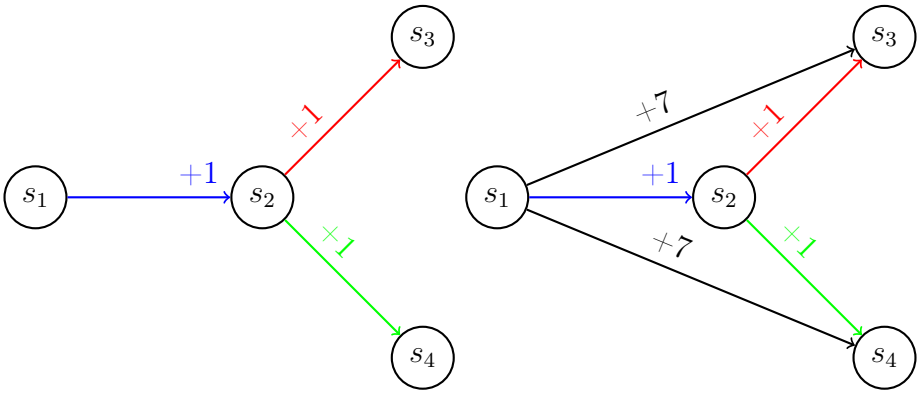

FIGURE 2 – Exemple d'ajout d'arêtes virtuelles

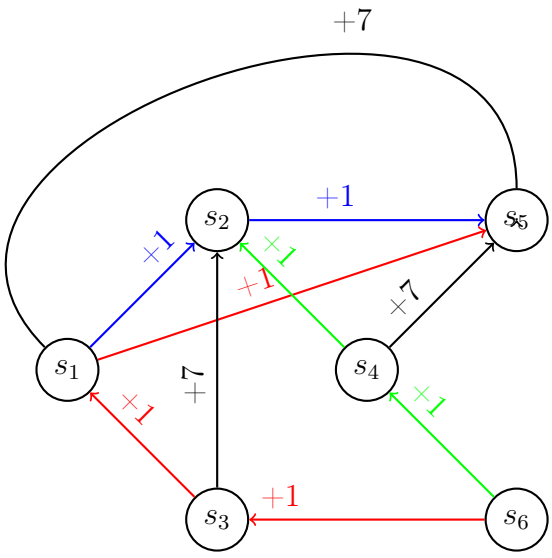

FIGURE 3 – Ajout d'arêtes virtuelles sur le précédent exemple

2.2 Échec de la modélisation précédente

L'implémentation précédente avec les arêtes virtuelles n'est pas bonne. Plusieurs raisons peuvent expliquer pourquoi c'était une mauvaise idée :

— Il existe plusieurs placements d'arcs virtuels selon notre poids de départ. Par exemple, dans le graphe de la figure 4 ci-dessous, si le point de départ se fait en s_1, il existe alors un arc virtuel s_1s_2, cependant si le point de départ se fait en s_3, alors l'arc s_1s_2 n'existe plus car il existe un arc s_3s_1.
Si à première vue, cela n'apporte pas de souci car le point de départ est fixe et ne change jamais, cela pose de gros problème lors du déroulement d'algorithme de recherche de plus court chemin tel que Dijkstra. Car lorsqu'on lance Dijkstra sur le graphe, il "change" de point départ avec son pivot au cours de ses itérations, et donc les arêtes virtuelles placées précédemment deviennent obsolètes.

— Un autre problème est que les arcs virtuels prennent énormément de place dans la mémoire car, sur des grandes données tels que les stations de métro, il peut exister des centaines de nouvelles arcs virtuels qui peuvent alourdir et rendre pénible l'exploration de plus court chemin. C'est un problème moins grave que le premier point mais qui a rendu incapable mon code de calculer le plus court chemin sur des longues distances.

En méconnaissance de ces soucis théoriques, j'ai essayé de faire exécuter l'algorithme de Dijkstra sur ce modèle.

Cette modélisation a véritablement nuit au projet, puisque pendant des semaines de travail, mon algorithme de plus court chemin Dijkstra, donnait plus de la moitié du temps des très mauvais résultats et avait des comportements inattendus.

Je pensais à tort que c'est l'algorithme de recherche Dijkstra la source du problème, c'est pourquoi je réécrivais et effaçais le code de mon algorithme pendant des semaines ; alors qu'au final l'algorithme était correct, c'était juste ma conception du graphe qui faussait le résultat.

J'ai perdu plusieurs semaines de travail à cause de ce souci, mais j'ai retenu comme leçon que la conception théorique d'un code, doit être suffisamment solide et doit bien épouser correctement pour qu'on puisse travailler dessus.

Une simple erreur de conception dans la philosophie du code peut provoquer des erreurs indétectables à la machine et ruiner plusieurs heures de travail. A cause de cela et du manque de main d'oeuvre (je suis en monôme), j'ai dû avorter certaines fonctionnalités dont j'avais prévues de relier à ce devoir, tel que l'ajout d'une interface graphique. Un exemplaire du code qui ne fonctionne pas et mise dans le dossier de ce devoir. Il est appelé "legacy-code.c".

J'ai donc du repartir de zéro, et créer une conception théorique plus robuste que le précédent face à ce problème.

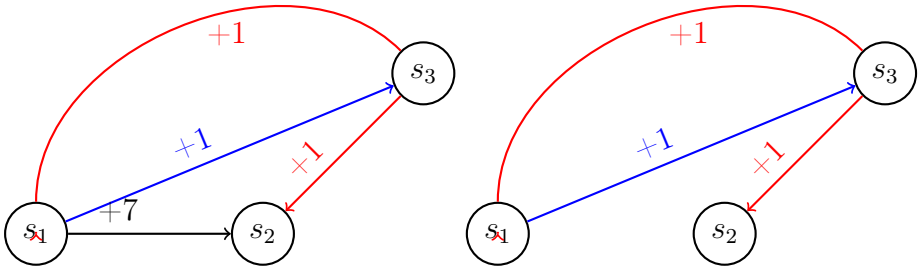

FIGURE 4 – Graphe où l'ajout d'arcs virtuels pose problème : le point de départ du premier graphe est s_1, celui du deuxième graphe est s_3

2.3 Modélisation à l'aide de poids virtuels

Nous allons donc abandonner l'idée de rajouter des arcs virtuels, et nous allons revenir à la représentation classique des sommets reliés uniquement avec arcs de poids 1. Il n'y aura donc dans notre modélisation uniquement les arcs présents dans le fichier .csv fournir par le professeur.

La grosse erreur que j'ai faite dans le précédent modèle est que je n'ai pas pris en compte la notion de parcours d'un chemin. C'est pourquoi on va utiliser cette notion pour pouvoir ajouter des poids fictifs (ici qui valent 5) lors du parcours de notre graphe, si on accède à un sommet donné avec une ligne différente.

Pour calculer le plus court chemin d'un point de départ A jusqu'à un point d'arrivée B, on définit le parcours $(s_n)_{n\in[\![1;k]\!]}$ qui est défini par le fait que $s_1 = \text{S}$ et que $p \in \mathbb{N}$ est un entier tel que $s_k = B$. Pour tout $i \in [\![1; k]\!]$, les sommets s_i sont l'ensemble des sommets tels que le parcours $(s_n)_{n\in[\![1;k]\!]}$ est le chemin minimal qui puisse exister entre A et B.

On va ajouter un poids virtuel qui n'apparaîtra que lorsqu'on parcourt le graphe. Ainsi pour un parcours de sommet dans un ordre défini $(s_n)_{n\in[\![1;k]\!]}$ donné, on définit alors la suite de poids associée suivante :

$$p_{n+1} = \begin{cases} 1 & n = 1 \\ 1 & C(s_n) = C(s_{n+1}) \\ 6 & C(s_n) \neq C(s_{n+1}) \end{cases} \tag{1}$$

Voici un exemple d'ajout de poids virtuels lors d'un parcours, avec en rouge, les stations s tel que $s \in Im(s)$

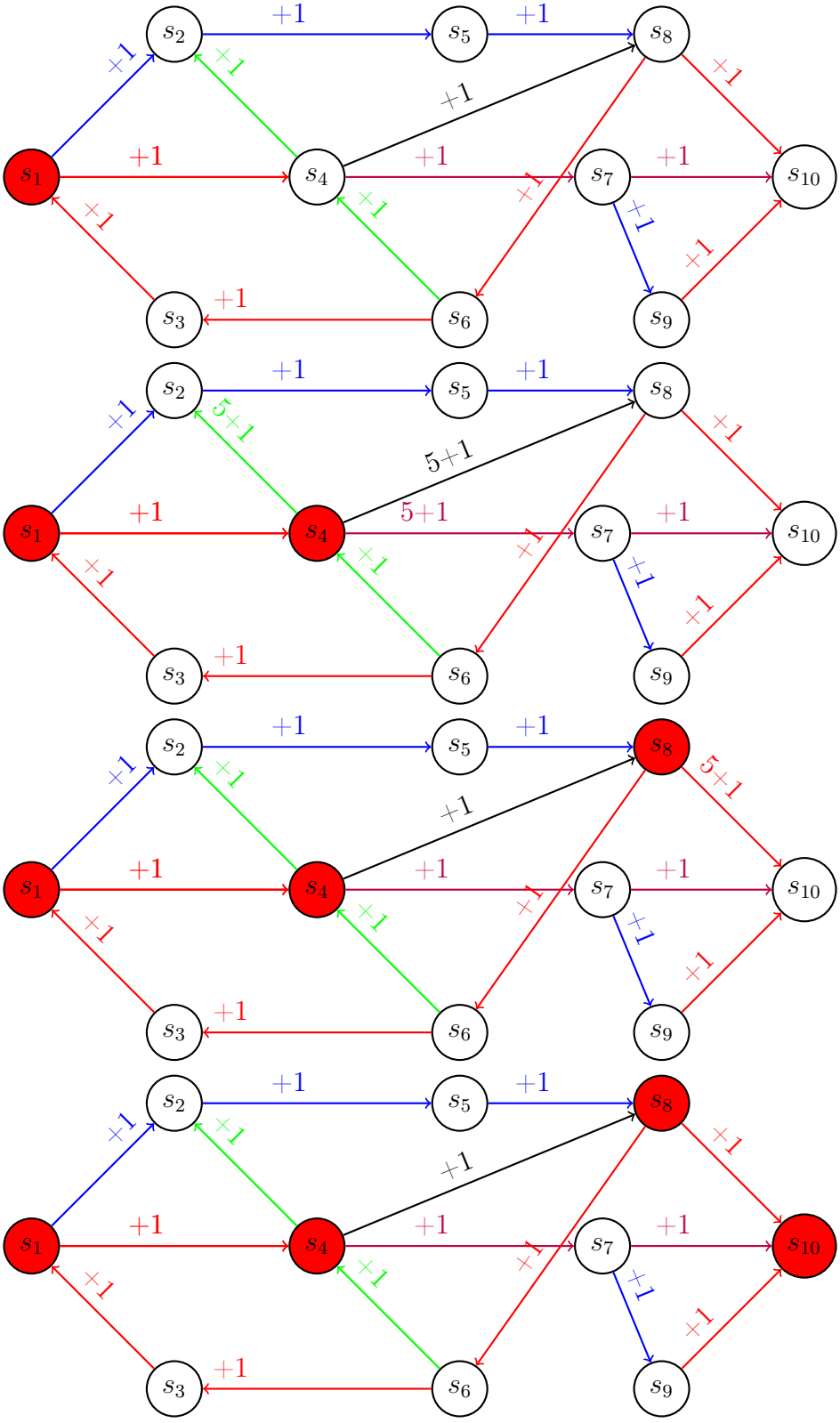

FIGURE 5 – Exemple d'illustration du modèle

3 Algorithme du plus court chemin

Nous allons dans cette partie, essayer d'appliquer l'algorithme de Dijkstra pour calculer le plus court chemin entre A et B. Cependant, pour l'instant notre modèle de Graphe demande quelques ajustements pour qu'une application de Dijkstra soit possible sur ce modèle de graphe. C'est pourquoi dans les deux premières sous-parties, nous allons nous ateler à ajuster notre graphe puis nous allons en troisième sous-partie, faire un rappel sur l'algorithme de Dijkstra. On note alors s_0, le sommet de départ et s_m, le sommet d'arrivée.

3.1 Quelques ajustement de notre modèle

3.1.1 Le parallélisme des arcs

On veut que notre Dijkstra soit compatible avec notre modèle de Graphe peu importe ce qu'il passe. Par exemple, notre graphe peut posséder des arêtes parallèles, c'est-à-dire que deux sommets peuvent être reliés par plusieurs arcs en même temps. Pour remédier à cela, on prend tout simplement l'arc avec le coût le plus bas (si les deux arcs sont de coûts égaux, on en supprime un au hasard).

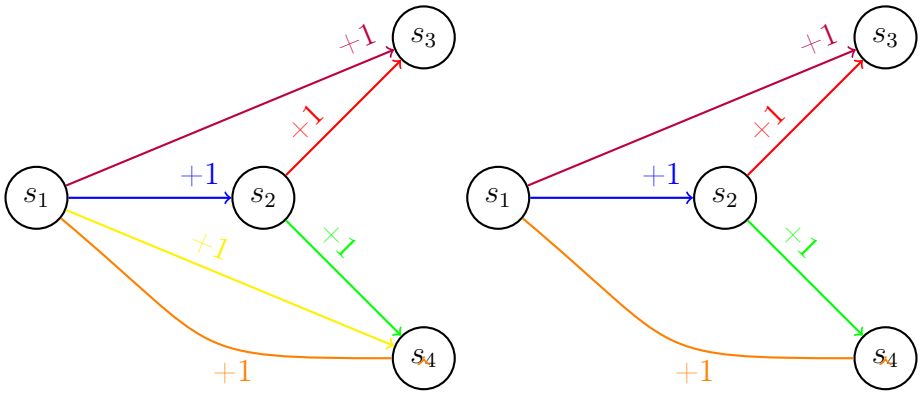

FIGURE 6 – Exemple de simplification d'arête

3.2 Algorithme de Dijkstra

L'algorithme de Dijkstra est un algorithme de recherche de plus court chemin entre un point A et un point B, qui fonctionne dans les graphes où il n'y a pas de poids négatifs. Condition que remplit notre graphe. Je ne vais pas détailler la démonstration de la terminaison, complexité et le pseudo-code de Dijkstra car cela a déjà été fait dans la littérature scientifique [1].

Cependant, j'irais détailler le fonctionnement de mon implémentation de Dijkstra en C plus en détail dans la partie 4.2.

1. Algorithmique de graphes, Sylvie BORNE

4 Implémentation de l'algorithme en C

Maintenant qu'on a défini à quoi ressemble notre modèle d'algorithme, nous pouvons alors implémenter tout cela en C.

4.1 Les structures

Pour pouvoir importer notre modèle en C. Il nous faut d'abord créer les structures associées.

Premièrement, on va créer une structure Station qui va représenter une station, un sommet de notre graphe, dans notre programme.

Listing 1 – Structure d'une station

```
typedef struct station
{
    uint32_t id;
    char nom[L_MAX];
    uint32_t visite;//Booleen
    uint32_t pi;

    uint32_t pds;
    uint32_t precedent;
    char ligne_precedent[L_MAX];

}station;
```

- L'attribut id, est l'id de la station contenu dans le fichier "Metro Paris Data - Stations.csv". C'est un attribut constante, une fois déclarée, elle ne peut plus être changé.
- L'attribut nom est le nom de la station associé à l'id, chargé en même temps qu'id. C'est un attribut constant.
- L'attribut visite est un booléen qui vaut 1 si la station a déjà été visité lors du parcours en largeur de Dijkstra. Elle vaut 0 à l'initialisation, et change au cours de Dijkstra.
- L'attribut pi, désigne le poids total nécessaire pour atteindre la station en question. Elle vaut infini à l'initialisation et change au cours de Dijkstra.
- L'attribut pds, enregistre le poids requis pour aller jusqu'à station (elle vaut 1 ou 6 en cas de changement de station). Elle vaut 1 à l'initialisation et change au cours de l'algorithme de Dijkstra .
- L'attribut précédent, enregistre l'id de la station précédente pour arriver à la station. Non déclaré, à l'initialisation, elle change au cours de Dijkstra.

On crée une structure arc, pour pouvoir y stocker les données csv du fichier "Metro Paris Data - Aretes.csv"

Listing 2 – Structure d'un arc

```
typedef struct arc
{
    uint32_t id_depart;
```

```
    uint32_t id_arrivee;
    char ligne[L_MAX];

}arc;
```

— L'attribut id_depart, charge les id de départ des arcs selon "Metro Paris Data - Aretes.csv"
— L'attribut id_arrivee, charge les id d'arivée des arcs selon "Metro Paris Data - Aretes.csv"
— L'attribut ligne,charge les lignes de métro des arcs selon "Metro Paris Data - Aretes.csv"

Enfin on va créer une liste chaînée pour pouvoir stocker les différents voisins des différentes stations.

Listing 3 – Structure d'un maillon

```
typedef struct maillon
{
    arc *arc_courant; //
    uint32_t poids;
    struct maillon* suivant; //
}maillon;
```

— L'attribut arc_courant, est un pointeur renvoyant un arc
— L'attribut poids, est un maillon renvoyant le poids
— L'attribut suivant, est un pointeur renvoyant sur le pointeur suivant sur le maillon de la liste.

Listing 4 – Structure d'une liste

```
typedef struct liste
{
    maillon* tete;
    maillon* queue;
    uint32_t taille;
}liste;
```

— L'attribut tete, désigne la tête de la liste, le premier arc courant
— L'attribut queue, désigne le queue de la liste, inutilisé ici
— L'attribut taille, désigne la taille de la liste ie le nombre d'arc pour chaque stations

Au final, il y a deux structures principales pour notre algorithme : un tableau de stations station ** stations de taille N_STATIONS, qui comporte sur chaque case i un attribut d'id i+1. Un tableau de liste arcs**, qui comporte sur chaque case i, une liste contenant des maillons des arcs, dont l'id_depart vaut i.

Et donc, avec un même indice i, il est possible d'accéder à toutes les informations de la stations de la stations d'id i+1, en faisant stations[i] et il est aussi possible d'accéder à tous les voisins de la stations en faisant arcs[i]->arc_courant. Voir la figure 7, pour avoir un diagramme de fonctionnement.

4.2 Algorithme de Dijkstra

Analysons point par point notre algorithme de Dijkstra.

Listing 5 – Analyse de l'en-tête

```
void dijkstra(station** stations, liste** arcs, uint32_t id_depart, uint32_t
    id_stop)
```

Dijkstra demande le tableau de stations, et le tableau de liste, décrit précédemment. Il demande aussi id_depart qui correspond à s_0 et id_stop qui correspond à s_m.

Listing 6 – Analyse des initialisations

```
{
    liste* s = new_liste(); //liste pivot
    int v = 0;
    stations[id_depart]->visite = 1;
    uint32_t pivot_precedent = id_depart;
    uint32_t pivot = id_depart;
    stations[pivot]->pi = 0;
    strcpy(stations[pivot]->ligne_precedent,"DEFAULT");
    char derniereLigne[L_MAX] = "DEFAULT";
```

On initialise divers variables qui serviront à stocker temporairement les différentes de la boucle qui va suivre.

Listing 7 – Analyse de la boucle while

```
    while((v<N_STATIONS-1)||stations[id_stop]->visite==0)
```

On parcours l'intégralité des stations parcourus avec un l'itérateur v, qu'on va incrémenter au cours de la boucle. La deuxième condition ne sert qu'à raccourcir la boucle, si jamais on visite le sommet de départ on a fini puis on sort de la boucle.

Listing 8 – Analyse du parcours des arêtes partant de i

```
while((v<N_STATIONS-1)||stations[id_stop]->visite==0)
{
    maillon* e = arcs[pivot]->tete;
    while (e!=NULL)
    {
        if (stations[e->arc_courant->id_arrivee]->visite ==0)
        {
        uint32_t poids = POIDS_DIRECT;

        if (strcmp(e->arc_courant->ligne,
            stations[pivot]->ligne_precedent)!= 0 &&
            strcmp(stations[pivot]->ligne_precedent, "DEFAULT")!=0)
        {
            poids += POIDS_TRANSITION;
        }
        if (stations[e->arc_courant->id_arrivee]->pi >
            stations[pivot]->pi+poids)
        {
            stations[e->arc_courant->id_arrivee]->pi = stations[pivot]->pi +
                poids;
            stations[e->arc_courant->id_arrivee]->pds = poids;
            strcpy(stations[e->arc_courant->id_arrivee]->ligne_precedent,
                e->arc_courant->ligne);
            stations[e->arc_courant->id_arrivee]->precedent =
                stations[pivot]->id;
        }
        }
        e = e->suivant;

    }
```

Ici, on prend pour chaque pivot parcouru, la tête de la liste de ses arcs.

On fait un parcours de liste avec while (e!=NULL) et e = e->suivant, tel que si e deviens la fin de la liste, elle devient nulle et termine la boucle.

On vérifie si on n'a pas déjà visité, le sommet dont l'arc pointe avec la condition if (stations[e->arc_courant->id_arrivee]->visite ==0)

On initialise une variable poids à 1, puis on compare la ligne de métro utilisé par l'arc avec celle du pivot. Si c'est différent on rajoute le temps d'attente. La deuxième condition strcmp(stations[pivot]->ligne_precedent, "DEFAULT") !=0 sert à permettre às_0 d'emprunter n'importe quelle ligne sans pénalité.

Pour chaque station, on calcule un poids total minimal pour arriver à cette station. Ce poids total est enregistré dans la variable pi de la structure station. Ainsi pour chaque station parcouru, on regarde si l'un de ses voisins à un poids total supérieur à celui du pivot avec le poids.

Si c'est le cas, on met à jour la variable avec "stations[e->arc_courant->id_arrivee]->pi = stations[pivot]->pi + poids ;". Enfin, à chaque qu'on fait cette mise à jour, on stocke le poids courant, la variable précédente, et la ligne précédente pour l'afficher ensuite quand on listera les prédecesseurs.

Listing 9 – Appel de minpi

```
        pivot = minpi(stations);

        add_tete(s, pivot,
            stations[pivot_precedent]->id,stations[pivot]->ligne_precedent,
            stations[pivot]->pds);

        stations[pivot]->visite = 1;
        v++;
```

Ici, on met à jour le pivot à l'aide de la fonction minpi, qu'on décrit ici :

Listing 10 – Appel de minpi

```
uint32_t minpi(station** stations)
{
    uint32_t min = INFINI;
    uint32_t id;
    for (int i = 0; i< N_STATIONS;i++)
    {
        if (stations[i]->pi<min&&stations[i]->visite != 1)
        {
            min = stations[i]->pi;
            id = i;
        }
    }
    return id;
}
```

minpi est une fonction qui renvoie l'idée de la station avec le poids minimal. On récupère l'id et on le met dans la variable pivot. On ajoute aussi le pivot actuel à la liste s, qui répertorie les éléments de la suite $(s_n)_{n\in[\![1;k]\!]}$.

Listing 11 – Algorithme complet de Dijkstra

```
void dijkstra(station** stations, liste** arcs, uint32_t id_depart, uint32_t
    id_stop)
{
    liste* s = new_liste(); //liste pivot
    int v = 0;
    stations[id_depart]->visite = 1;
    uint32_t pivot_precedent = id_depart;
    uint32_t pivot = id_depart;
    stations[pivot]->pi = 0;
    strcpy(stations[pivot]->ligne_precedent,"DEFAULT");
    char derniereLigne[L_MAX] = "DEFAULT";
    //stations pivot = new_maillon(id_depart, )


    while((v<N_STATIONS-1)||stations[id_stop]->visite==0)
    {
        maillon* e = arcs[pivot]->tete;
        while (e!=NULL)
        {
            if (stations[e->arc_courant->id_arrivee]->visite ==0)
            {
            uint32_t poids = POIDS_DIRECT;

            if (strcmp(e->arc_courant->ligne,
                stations[pivot]->ligne_precedent)!= 0 &&
                strcmp(stations[pivot]->ligne_precedent, "DEFAULT")!=0)
            {
                poids += POIDS_TRANSITION;
            }
            if (stations[e->arc_courant->id_arrivee]->pi >
                stations[pivot]->pi+poids)
            {
                stations[e->arc_courant->id_arrivee]->pi = stations[pivot]->pi +
                    poids;
                stations[e->arc_courant->id_arrivee]->pds = poids;
                strcpy(stations[e->arc_courant->id_arrivee]->ligne_precedent,
                    e->arc_courant->ligne);
                stations[e->arc_courant->id_arrivee]->precedent =
                    stations[pivot]->id;
            }
            }
            e = e->suivant;

        }
        pivot = minpi(stations);

        add_tete(s, pivot,
            stations[pivot_precedent]->id,stations[pivot]->ligne_precedent);
```

```
        stations[pivot]->visite = 1;
        v++;

    }
    //maillon* e = id_stop;
    printf(" %d (%s) -> %d (%s) : %d \n", id_depart+1,stations[id_depart]->nom,
        id_stop+1,stations[id_stop]->nom, stations[id_stop]->pi);

    printf("\n------------------------------\n");
    uint32_t p = id_stop;
    uint32_t poids_total = 0;
    while (p != id_depart)
    {
        printf("%s(%d)<-%s(%d) (ligne %s) poids +%d\n",
            stations[p]->nom,p+1,stations[stations[p]->precedent]->nom,
            stations[p]->precedent+1, stations[p]->ligne_precedent,
            stations[p]->pds);
        poids_total += stations[p]->pds;
        p = stations[p]->precedent;
    }
    printf("\n");
    printf("poids %d", poids_total);
}
```

Si vous voyez mal

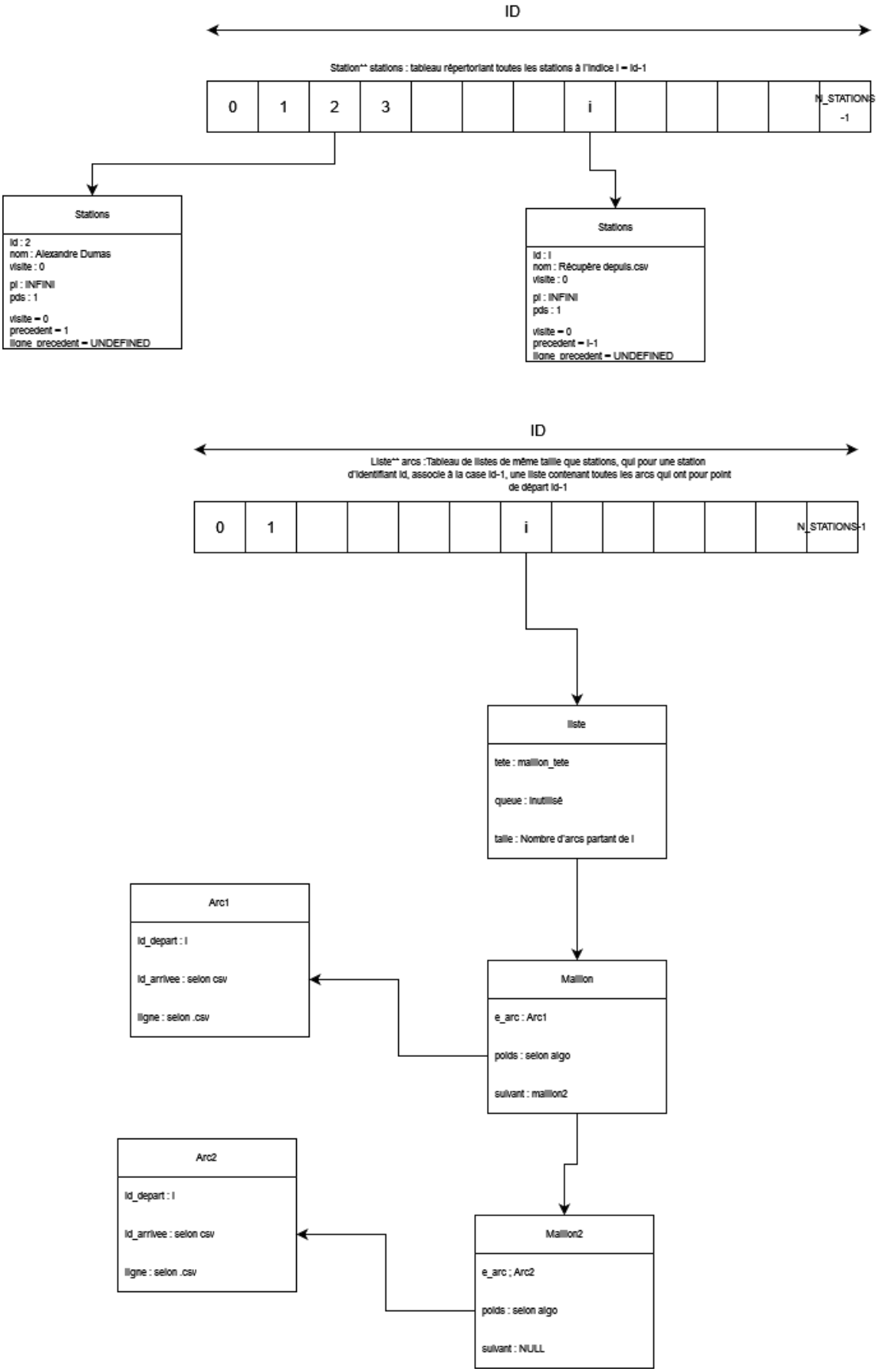

FIGURE 7 – Diagramme de notre structure de donnée

5 Conclusion

Après avoir programmé notre nouvelle version du programme, et puis tester sur différents exemples de lignes de métro notre programme sur différentes stations de métro, nous en concluons que notre programme fonctionne. Il donne un résultat correct sur les quelques exemples que j'ai pu tester, et arrive à sortir correctement les stations de métro qu'on visite.

5.1 Améliorations possibles

En ce qui concerne l'affichage du programme, il a été prévu au départ, de faire un affichage avec une fenêtre graphique. Cependant à cause de mon modélisation qui marchait pas, je n'ai pas pu commencer à étudier cela.

De plus on peut améliorer ce programme, en utilisant un non pas l'algorithme de Dijkstra classique mais un algorithme de recherche A*, qui est une version avancée de Dijkstra dans le domaine de l'intelligence artificielle.

FIN