Année 2022-2023

RAPPORT DE PROJET
-
Structures de Données

PROJET
-
INFORMATIQUE

« Calculateur d'iténéraire pour le métro »

Projet réalisé par

Mingjie ZHANG
Massyl CHAKER

Projet encadré par

John CHAUSSARD

Sup Galilée ingénieurs
L'ÉCOLE D'INGÉNIEURS DE L'UNIVERSITÉ SORBONNE PARIS NORD

SOMMAIRE

Introduction :

Dans le cadre du cours de structures de données et programmation avancée, nous avons réalisé notre projet, un calculateur d'itinéraire pour le métro. Le but de ce projet est d'utiliser les structures apprises en cours, afin de stocker les données que notre programme va utiliser et y accéder rapidement. Nous aurions pu choisir des méthodes plus basiques, mais utiliser des structures de données plus ou moins complexes rend le programme plus rapide et plus efficace, nous allons en reparler un peu plus loin.

Ce projet nous demande de créer un algorithme qui enregistre des données (stations du métro parisien et correspondance pour chacune des stations) afin de les stocker dans des structures de données qui facilitent leurs manipulation. Une fois cette étape finie, l'algorithme devra calculer le plus court chemin entre deux stations afficher le bon trajet ainsi que le temps que cela va nous prendre.

Dans ce rapport nous vous expliquons notre algorithme de calcul du plus court chemin ainsi que les structures de données utilisées pour stocker les données initiales du programme, les données nécessaires au calculs de l'algorithme ainsi que le résultat final qui est le plus court chemin et le temps mis entre une station de départ et une station d'arrivée.

Dans un premier temps nous avons choisi de représenter toutes nos stations sous forme d'un graphe, où chaque nœud représente une station et les arêtes représentent le lien entre deux station. Le coût de chaque arête est de 1 ou 5, 1 si les deux stations reliées par l'arête sont sur la même ligne et 5 si non.

Une fois que l'utilisateur ait entré une station de départ et une station d'arrivée, nous appliquons l'algorithme de Dijkstra, qui est utilisé en pour résoudre le problème du plus court chemin entre deux villes en connaissant le réseau routier mais que nous avons implémenté de façon à ce qu'il puisse trouver le plus court chemin entre deux stations de métro en connaissant le réseau du métro ainsi que le coût de chaque déplacement.

Raisonnement et réflexion :

Dans le but de plannifier l'organisation des taches que nous devons effectuer afin de résoudre le problème. Nous avons, dans un premier temps, commencé par une séance de discussion afin de fixer les objectifs que notre future programme doit accomplir, et de s'accorder une première solution du problème. Pour cela, nous avons définit et prit en compte plusieurs possibles facteurs limitants, et de mettre en commun nos idées.

Tout d'abord, nous avons décidé que le choix de l'algorithme est primordial, la structure de donnée doit s'adapter au fonctionnement de l'algorithme et non l'inverse. Notre choix s'est reposé sur l'algorithme de Dijkstra vu dans la matière « Algorithmique des graphes », par le fait qu'il nous soit familier et de son fonctionnement relativement simple et efficace dont nous allons expliquer dans la suite.

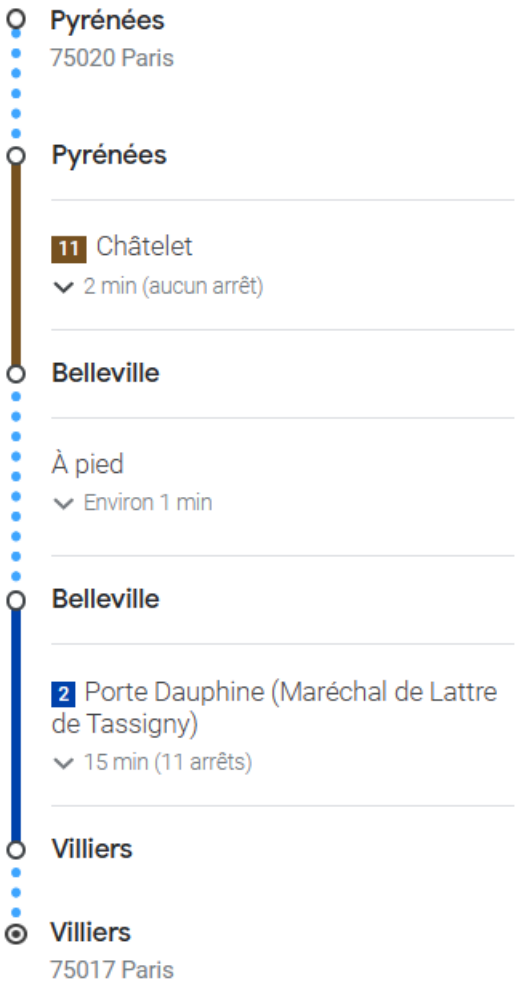
L'algorithme de Dijkstra nécessite une représentation des données sous forme d'un graphe connexe, c'est-à-dire que pour n'importe quelle pair de sommet du graphe, il existe au moins un chemin composé de sommets et d'arêtes qui les relie . Nous devons donc conserver cette notion dans la structure de donnée qu'on recherche.

Par la suite, nous avons commencé à élaborer un début de structure sans aucun code encore en prenant en compte la contrainte précédente, et nous nous sommes inspiré de la représentation des lignes de métro, avec une structure « metro » contenant toutes les lignes, et chaque ligne est une structure contenant un ensemble de station, ses terminus et son numéro, et chaque station est une structure contenant les informations sur son nom , les changements possibles à cette station et les stations précédentes et suivantes sur la même ligne.

Nous avons remarqué que pour définir les stations précédentes et suivantes d'une station, il faudra définir une direction de la ligne, c'est-à-dire que la station suivante mène vers lequel des deux terminus.

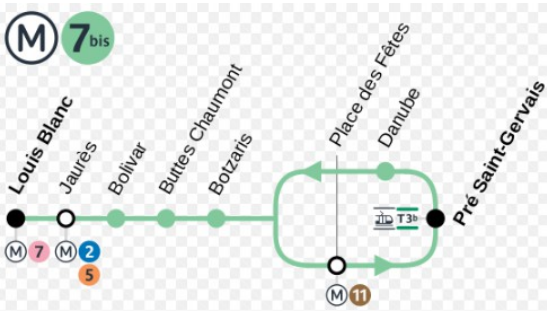
Après avoir consulté le format sous lequel les données sur le métro sont représentées, nous avons considéré cela comme aussi un facteur limitant, les fichiers « Metro Paris Data - Aretes » et « Metro Paris Data - Stations » nous donnent des informations concernant le nom des stations et leur id ainsi que le numéro des lignes qui relient chacunes des stations si elles existent. Cela signifie que nous n'avons pas d'informations explicites sur les terminus de chaque ligne, ainsi que l'ordre de leur stations, ce qui est un problème de plus à résoudre si on cherche à implémenter la structure définie ci-dessus.

Et enfin, nous voulions que notre programme affiche le trajet dans le terminal de la même manière que Google Map nous l'aurait donné.



Avec les informations concernant le nom de la station de départ, de la station d'arrivée, les changements de ligne, et pour chaque ligne utilisée, indiquer le nombre de stations à prendre sur cette ligne et vers quel terminus prendre, ainsi que les noms des stations.

Ce type d'affichage nous impose alors l'obligation de connaître les terminus de chaque ligne. Nous avions trouvé une solution qui est implémentable pour les terminus ne sont pas dans une boucle, il suffit de trouver les stations qui n'ont qu'une seule station voisine sur la même ligne. Or pour les terminus situées dans une boucle comme la station « Pré Saint-Gervais » de la ligne 7bis,



il n'est pas possible de trouver le terminus, car chaque station de la boucle possède les mêmes caractéristiques dans le graphe : elles ont toutes deux stations voisines, et le terminus peut se trouver à n'importe quelle position dans la boucle.

C'est pourquoi, nous avions abandonné l'idée de déterminer les terminus de chaque ligne, puisque nous préfèrons qu'il n'y a pas de modification du code dépendamment des exceptions comme ici.

Test, échec et progression.

Cette partie résume notre avancement jusqu'à obtenir la structure finale qui sera utilisée dans le programme de la partie « Explication du code », en passant par les essais et échecs.

Tout d'abord, pour être le plus conforme à la structure définie dans la partie « Raisonnement et Réfléxion », nous avons décidé de coder nos structures sous forme de liste doublement chainée.

Voici la structure « ligne» qui est donc une liste doublement chainée dont chacun des maillons est une station, cette structure a été élaboré avant qu'on abandonne l'idée de mettre les terminus dans la structure.

```
typedef struct station_s{
    char nom[50];
    char ligne[5];
    struct station_s ** voisins;
    struct station_s * suivant;
    struct station_s * precedent;
}station;

typedef struct{
    station terminus1;
    station terminus2;
    int nb_station;
}ligne;
```

La structure « station » contient un tableau de pointeur sur des stations voisines, ce tableau sera alloué dynamiquement selon le nombre de voisins que la station possède.

Nous avons abandonné cette structure aussi pour une autre raison, l'initialisation des structures de données n'est pas optimisée. En effet, pour créer les stations, il suffit d'allouer de la mémoire pour une structure « station ».

```
station * creer_station(char * nom){
     station * new = malloc(sizeof(station));
     strcpy(new->nom, nom);
     new->suivant = NULL;
     new->precedent = NULL;
     new->voisins = NULL;
     return new;
}
```

Mais cette station nouvellement crée doit être stockée quelque part le temps de trouver le terminus d'une station car ce dernier est le début/fin de notre ligne. Donc il nous faut une structure pour stocker temporairement les stations créees, supposons que c'est une pile qui servira de stockage, une fois que toutes les stations seront créees, on doit alors parcourir l'entiereté de la pile pour trouver à chaque parcours une station pour la rattacher à une de nos ligne. Ce qui n'est pas du tout optimisé en terme de temps.

Après cette première tentative, nous avons remarqué qu'au lieu de stocker les données sous forme d'un graphe, nous pouvons les stocker sous forme d'une matrice d'adjacence, une matrice d'adjacence est une autre forme de représentation d'un graphe équivalente à la représentation visuelle. Pour des sommets i et j donnés, elle donne une valeur qui correspond aux nombres d'arêtes reliant ces deux sommets.
Nous avions donc décidé de représenter cette matrice sous forme d'un tableau à deux dimensions dont chacune des cases est une chaîne de caractère indiquant la ligne accompagné d'un tableau dont pour une case i, nous avons le nom de la station associée.

```
#define STAT 304

typedef struct{
    char nom[5];
}ligne ;

typedef struct{
    char nom[50];
}station;

typedef struct{
    ligne tab[STAT][STAT];
    station tab[STAT] ;
}metro;
```

mais assez rapidement, nous avons constaté que ce n'est pas non plus une solution correcte. En effet, même si ce tableau à deux dimensions est simple à initialiser et à utiliser. Il dispose de deux problèmes majeurs.
Le premier étant : il existe des stations voisines qui sont reliées par plusieurs lignes, comme dans le cas des stations « Grands Boulevards » et « Bonne Nouvelle », toutes deux reliées par la ligne 9 et 8, par conséquent, au lieu que chaque case ne puisse stocker qu'une seule ligne, il faut qu'elle puisse en stocker plusieurs, ce qui augmente la charge mémoire.
Cela nous amène au deuxième problème, un tableau à deux dimensions de 304 cases de côté, dont chaque case contient une liste de chaîne de caractère, occupe un espace mémoire non négligeable. Surtout la majorité des cases ne sont pas utilisées, le tableau comporte 92 416 cases, or nous avons qu'un peu plus de 700 liaisons entre stations, ce qui fait un taux d'utilisation que de seulement (700/92 416)* 100 ≈ 0,76% !
Nous pourrions allouer dynamique de la mémoire que pour les cases que nous utilisons, ainsi nous économiserons beaucoup de mémoire, mais la simplicité d'utilisation de ce tableau nous vient de son parcourt facile, pour trouver tous les voisins d'une station d'identifiant « id », il suffit de parcourir la ligne du tableau d'indice « id ».

```
En variant j de 0 à 403, on peux obtenir tous les voisins
metro->tab[id][j]
```

Et c'est ce trait qui pose problème, en effet, cela signifie que le programme doit parcourir 304 cases du tableau pour chaque station. C'est pourquoi nous délaissons aussi cette structure.

Notre troisième essai peut se résumer à une fusion de certaines caractèristiques des deux tentatives précédentes. En effet, pour la facilité d'accès et initialisation des structures de données, nous avons choisi un tableau à une seule dimension dont chaque case correspond à une station, qui est une structure, et chaque station aura une liste de champs indiquant son nom, un pointeur sur une liste de voisin et son nombre de voisins. De cette façon, nous facilitons l'accès aux données par l'utilisation d'un tableau, et en même temps nous limitons l'utilisation de l'espace mémoire à ce qui est nécessaire. Nous verrons cela en détail dans la partie « Explication du code ».

Description de l'algorithme et choix de la structure pour stocker le parcours.

Dans cette partie, nous allons vous expliquer le fonctionnement de l'algorithme de Dijsktra, et quel type de structure de donnée nous avons choisi pour stocker le trajet trouvé.

L'algorithme de Dijkstra se résume à trouver un chemin entre deux sommets d'un graphe, si ce chemin existe. Il fonctionne de la manière suivante, pour un graphe donné, un sommet de départ A et un sommet d'arrivé B donnés. A l'initialisation, on met le sommet A dans l'ensemble S, initialement vide et qui servira à stocker tous les sommets qu'on parcourra, et ensuite à chaque itération, nous ferons dans l'ordre :

-La recherche d'un sommet x qui n'est pas dans S, et qui est le plus proche des sommets de notre ensemble, c'est-à-dire l'arête qui a comme extrémités, un sommet dans S et le sommet x dans \S, et dont le coût qui est minimal.

-Nous ajoutons ce sommet x dans l'ensemble S, et le sommet est considéré comme marqué.

-Et tant que le somme d'arrivé B n'est pas dans l'ensemble S, on continue les itérations.

Dans la matière « Algorithmique des graphes », nous utilisons cet algorithme avec ce qu'on appelle un tableau de prédécesseur, un tableau dont chaque case contient un couple d'information, l'indice de la case prédécésseur, et le coût de la case.
Nous avons décidé de reprendre cette structure pour stocker le parcours, et nous initialiserons les cases au besoin, c'est pourquoi nous considérons cette structure comme adapté.

```
typedef struct {
  char ligne[5];
  int indice_pred ;
  int cout;
  int marque;
}predecesseur;
```

Ce tableau de prédécesseur se comprend de cette manière, pour une case donnée non vide, qui correspond à une station, on retrouve le nom de la ligne qui a permit d'arriver à cette station, l'indice de la station précédente dans le trajet, le coût qui est le temps à mettre pour arriver à cette station, et un champ marque qui indique si cette station a été mise ou non dans l'ensemble S.

Explication du code :

Notre fonction main contient un seul appel à une fonction qui est « initialisation algorithme »

```
void initialisation_algorithme(){

   metro * _metro = creer_metro();
    int st1 = 0, st2 = 0;
    extraire_donnee("Metro Paris Data - Stations.csv", "Metro Paris Data -
Aretes.csv", _metro);
    for(int i = 0; i <STAT ; i++){
        printf("\"%s\" - %d\n",_metro->tab[i].nom, i + 1);
    }

    printf("Veuillez saisir l'id des stations entre 1 et 304 inclus :\n");
    while((st1 < 1 || st1 > 304) || (st2 < 1 || st2 > 304)){
        printf("Entrez la station de départ : ");
        scanf("%d", &st1);
        printf("Entrez la station d'arrivée :");
        scanf("%d", &st2);
        if((st1 < 1 || st1 > 304) || (st2 < 1 || st2 > 304)
            printf("Les id entrés sont invalides, veuillez resaisir\n");
    }

    algorithme_plusCourtChemin(_metro, st1 - 1, st2 - 1);
}
```

Cette fonction créée une nouvelle structure métro afin d'y stocker les données contenus dans les fichiers « Metro Paris Data - Stations.csv » et « Metro Paris Data - Aretes.csv ». Le nom des stations ainsi que leur indice est affiché à l'aide d'une boucle for. On demande ensuite à l'utilisateur d'entrer le numéro de la station de départ ainsi que le numéro de la station d'arrivée. Si l'un des deux numéros n'est pas compris entre 1 et 304, le programme le lui redemandera, car il y a 304 stations dans notre structure métro.

La fonction utilisée afin d'extraire les données des deux fichier csv pour les stocker dans la structure metro est `void extraire_donnee(char* _stationFileName, char * _edgeFileName, metro * _metro)`

```
/* extraire les données des fichiers */
void extraire_donnee(char* _stationFileName, char * _edgeFileName, metro *
_metro) {
    //Ouverture du fichier _stationFileName, et extraction de ses données vers
le tableau contenu dans _metro
    FILE * f_station = fopen(_stationFileName,"r");
    if(f_station == NULL){
        printf("extraire_donnee : Erreur lecture fichier f_station\n");
        assert(0);
    }

    char *name;
    char * id;
    char e;
    e = fscanf(f_station, "%*[^\n]\n");
    while(!feof(f_station)){
        char data[50];
        fgets(data,50,f_station);
        name = strtok(data, ",");
        id = strtok(NULL, ",");
```

```
        //printf("%s----%s\n", data, id);
        strcpy(_metro->tab[atoi(id)- 1].nom, data);
    }

    //Ouverture du fichier _edgeFileName et extraction de ses données vers des
maillons voisions
    //puis ajoute les voisins aux stations
    FILE * f_edge = fopen(_edgeFileName, "r");
    if(f_edge == NULL){
        printf("extraire_donnee : Erreur lecture fichier f_edge\n");
        assert(0);
    }
    int start, end;
    char line[5];
    e = fscanf(f_edge, "%*[^\n]\n");
    while(!feof(f_edge)){
        e = fscanf(f_edge, "%d,%d,%s\n", &start, &end, line);
        //printf("start : %d, end : %d, ligne : %s\n", start, end, line);
        ajouter_voisin(_metro, start - 1, end - 1, line);
    }


    fclose(f_edge);
    fclose(f_station);
}
```

Cette fonction commence par ouvrir le fichier en lecture _stationFileName puis lit le fichier jusqu'à trouver le caractère de fin de fichier (eof). Afin de séparer le nom des stations de leur numéro, nous avons utilisé la fonction « strtok() », ainsi après avoir lu la première station et son numéro, nous faisons appel à cette fonction deux fois name = strtok(data, ","); et id = strtok(NULL, ",");
Pour le premier appel de fonction nous avons passé en paramètre la chaîne de caractère à séparer ainsi que son séparateur, la fonction nous renvoie le premier token (la partie avant le séparateur), nous faisons ensuite un deuxième appel mais cette fois-ci en passant un pointeur NULL comme paramètre, cela nous permet de récupérer le deuxième token qui est le numéro de la station. Pour chaque case du tableau de stations stocké dans la structure metro, nous faisons une copie du nom de la station avec la ligne de code : strcpy(_metro->tab[atoi(id)- 1].nom, data);

Ensuite nous ouvrons en lecture le fichier _edgeFileName, qui contient les numéros de stations voisines ainsi que la ligne qu'elles ont en commun. Nous récupérons le numéro de la première station, le numéro de la seconde station ainsi que la ligne qui les relient avec : e = fscanf(f_edge, "%d,%d,%s\n", &start, &end, line);
Après cela on met à jour la liste des voisins de la première station en ajoutant la seconde station dans sa liste de voisin, le coût de cette nouvelle station sera calculé par la fonction ajouter_voisin().

```
void ajouter_voisin(metro *m, int dep, int arv, char *ligne) {
    voisin *vs = creer_voisin(arv, ligne);
    vs->suivant = m->tab[dep].premier;
    m->tab[dep].premier = vs;
    m->tab[dep].nb_voisins++;
}
```

Chaque station de tab contient un pointeur sur un de ses voisins(station voisine), nous insérons donc le nouveau voisin en tête de liste et on incrément de nombre de voisins.

Enfin nous appliquons l'algorithme de recherche du plus court chemin avec cette fonction :

```
void algorithme_plusCourtChemin(metro *m, int depart, int arrivee)
```

Nous commençons par déclarer un tableau de « prédécesseur » qui nous sera très utile pour retrouver le plus court chemin(la solution optimale) d'une station A à une station B.

```
    predecesseur *tab[STAT];
    for(int i = 0; i < STAT; i++)
        tab[i] = NULL;
    tab[depart] = malloc(sizeof(predecesseur));
```

Le tableau de prédécesseurs nous sert à retrouver le plus court chemin, mais aussi à mettre à jour les prédécesseur de certaines stations si au cours des itérations, un chemin plus court que celui qui y était est trouvé.

```
    char ligne_depart[4] = "null";
    tab[depart]->indice_pred = depart;
    tab[depart]->cout = 0;
    tab[depart]->marque = 1; // sommet de départ marqué
    strcpy(tab[depart]->ligne, ligne_depart);
    mettre_a_jour_voisins(m->tab[depart], tab, ligne_depart, depart);
```

Les champs de tab[depart] doivent être mis à jour, avant qu'on fasse appel à la fonction mettre_a_jour_voisin, qui ajoutera les voisins de la station de départ dans sa liste de voisins.

```
    int min = 100000;
    int ind_min = depart;
    int fini = 0;
    while(!fini) {
        min = 100000;
        for(int i = 0; i < STAT; i++) {
            if(tab[i] != NULL && tab[i]->marque == 0) {
                if(tab[i]->cout < min) {
                    min = tab[i]->cout;
                    ind_min = i;
                }
            }
        }

        tab[ind_min]->marque = 1;
        if(tab[arrivee] != NULL && tab[arrivee]->marque == 1)
            fini = 1;
        mettre_a_jour_voisins(m->tab[ind_min], tab, tab[ind_min]->ligne,
ind_min);
    }
```

Maintenant que la station de départ est marquée nous devons trouver notre nouveau pivot (station non marquée ayant le coût le plus petit du tableau), une fois qu'on a trouvé le pivot, il sera marqué et ses voisins seront mis à jour. Plusieurs itérations seront faites mais la dernière itération sera faite quand la condition « tab[arrivee] != NULL && tab[arrivee]->marque == 1 » sera satisfaite. Cela veut dire que l'on va sortir de la boucle while quand la station d'arrivée sera marquée.

```
    /* affichage du trajet */
```

```
    int i = arrivee;
    while(i != depart) {
        printf("minute %d : arrivé à la station \"%s\" avec la ligne
%s\n",tab[i]->cout,m->tab[i].nom, tab[i]->ligne);
        printf("%d\n",i + 1);
        i = tab[i]->indice_pred;
    }
    printf("minute 0 : Station de départ : %s\n", m->tab[i].nom);
```

Maintenant que le tableau de prédécesseur contient tous les indices des prédécesseurs de la stations d'arrivée, nous n'avons plus qu'à parcourir le tableau en commençant par l'indice de la station d'arrivée, tout en affichant le coût pour chaque station, qui lui aussi est stocké dans le tableau.

La fonction mettre_a_jour voisin qui permet de mettre à jour le tableau de prédécesseur afin de trouver le pivot traite deux cas, le premier cas est celui où nous devons mettre à jour les voisins de la station de départ

```
    voisin * vs = st.premier;
    char ligne_depart[4] = "null";
    // si nous sommes dans le premier cas (station de départ)
    if(!strcmp(ligne_depart, ligne_actuelle)){
        while(vs != NULL){
            int id = vs->id_voisin;
            tab[id] = malloc(sizeof(predecesseur));

            strcpy(tab[id]->ligne,vs->ligne);
            tab[id]->indice_pred = ind_station_actuelle;
            tab[id]->cout = 1;
            tab[id]->marque = 0;
            vs = vs->suivant;
        }

      }
```

Nous avons choisi de mettre à 1 tous les coûts des voisins de la station de départ. Ensuite nous devons traiter le cas des autres stations

```
  else {
        while(vs != NULL){
            int id = vs->id_voisin;
            if(tab[id] == NULL) { // si la case n'a jamais été initialisé
                tab[id] = malloc(sizeof(predecesseur));
                strcpy(tab[id]->ligne,vs->ligne);
                tab[id]->indice_pred = ind_station_actuelle;
                if(!strcmp(vs->ligne, ligne_actuelle))
                    tab[id]->cout = tab[ind_station_actuelle]->cout + tps_dplmt;
                else
                    tab[id]->cout = tab[ind_station_actuelle]->cout + tps_dplmt
                                    + tps_chgmt;
                tab[id]->marque = 0;
            }
            else { // si la case a déjà été initialisé
                int cout1;
                // on stocke le cout
                if(!strcmp(vs->ligne, ligne_actuelle))
                    cout1 = tab[ind_station_actuelle]->cout + 1;
                else
                    cout1 = tab[ind_station_actuelle]->cout + 6;
```

```
                // si le nouveau cout est inférieur a celui déjà stocké => on le
met a jour
                // il ne doit pas être mis a jour si déjà marqué
                if(cout1 < tab[id]->cout && tab[id]->marque == 0){
                    strcpy(tab[id]->ligne,vs->ligne);
                    tab[id]->indice_pred = ind_station_actuelle;
                    if(!strcmp(vs->ligne, ligne_actuelle))
                        tab[id]->cout = tab[ind_station_actuelle]->cout + 1;
                    else
                        tab[id]->cout = tab[ind_station_actuelle]->cout + 6;
                    tab[id]->marque = 0;
                }
            }
            vs = vs->suivant;
        }

    }
```

Si un voisin de la station n'est pas dans le tableau de prédécesseur on l'initialise et le coût est égal au coût de la station actuelle + 1 si elles sont sur la même ligne ou bien c'est égal au coût de la station actuelle + 6 si elles ne sont pas sur la même ligne.
Deuxième cas, si le voisin est déjà présent dans le tableau de prédécesseur, il sera mis à jour uniquement si son coût calculé à partie du coût de la station actuelle est inférieur au coût déjà stocké dans le tableau.

Conclusion :

Ce projet nous a permis de mettre en pratique des connaissances que nous avons acquis en cours, nous avons manipulé des structures de données dans le but d'optimiser notre programme afin d'utiliser le moins de mémoire possible, c'est pour cela que nous sommes passé par différentes méthodes pour au final trouver la méthode qui nous convenait et qui trouvait le résultat optimal dans la majorité des cas. L'amélioration que nous avons apporté à notre code en choisissant de stocké nos données dans un tableau de structure et de représenter le lien entre deux stations grâce à une liste chaînée n'est venu qu'après être passé par des méthodes qui sont moins performante et qui utilisent beaucoup plus de mémoire inutilement. Nous avons donc choisi l'efficacité d'un tableau a une dimension contenant des structures à la facilité d'implémentation d'un tableau à deux dimensions.

Nous avons réussi à créer des structures de données qui sont facilement utilisable pour l'algorithme de Dijkstra, en effet notre tableau de prédécesseur permet de marquer un sommet(une station avec un simple accès à son indice) et la mise à jour des voisins d'un sommet se fait aussi en parcourant une liste chaînée. La structure station nous permet d'accéder à la liste des voisins de chaque station et la structure prédécesseur contient tous les champs indispensable au bon fonctionnement de l'algorithme. Néanmoins notre programme n'est pas parfait, en effet l'algorithme de Dijkstra s'applique parfaitement à la recherche du plus court chemin entre deux villes mais pour des stations de métro c'est un peu différent. Dans le cas d'un réseau routier, une solution optimale d'un problème trouvée avec l'algorithme de Dijkstra fait apparaitre une sous solution optimale d'un sous-problème, car le plus court chemin d'un point A à un point B est égal au plus court chemin du point A au point C(prédécesseur de B) plus le coût de l'arête C-B. Le chemin A-C est forcément optimal pour le problème du plus court chemin de A à C. Dans le cas des stations de métro nous n'avons pas forcément cette propriété de sous structures optimale. En effet dans certains cas le plus court chemin d'une station A à une station B ne dépend pas forcément du plus court chemin de la station A à la station C (prédécesseur de B), car le plus court chemin dépend également de la ligne qui va être prise de la station C à la station B. Par exemple le plus court chemin de la station A à la station C nous donne un coût de 14 en prenant la ligne 1 pour arrivé à la station C, hors nous devons maintenant prendre la ligne 2 de C à B ce qui nous donne un coût total de 20. Un autre chemin de A à C nous donne un coût de 17, en prenant la ligne 2 pour arriver à la station C, ce n'est pas le plus court chemin de A à C mais le fait de rester sur la ligne 2 de la station C à B nous donne un coût total de 18 , qui est la vrai solution optimale.

Faut de temps nous n'avons pas pu traiter ce cas particulier, mais nous aurions aimé changer notre mode de marquage, nous ne marquerons pas uniquement les stations mais aussi les lignes qui mènent vers ces stations, ainsi dans notre exemple, nous aurions marqué la station C avec la ligne 1 mais aussi la station C avec la ligne 2 et le plus court chemin de A vers B sera égal au minimum de tous les chemins de A à C plus le coût minimal de C à B, ainsi la propriété de sous-structure optimale aurait été appliqué correctement dans notre algorithme.