

Rapport de projet de Structures de données

Ryan Yala & Mohamed Aziz Taieb

2 février 2023

Encadrant : John Chaussard



Table des matières

INTRODUCTION	2
Approche de l'Algorithme de résolution	3
RESOLUTION	
Implémentation	6
Tableau synthétique des fonctions	8
Avantages et Inconvénients	10
Conclusion	11

Introduction Général :

Ce document est un rapport expliquant le projet réalisé par Ryan YALA et Mohamed Aziz TAIEB , étudiants en première année du cycle ingénieur informatique de SUP Galilée . En fin d'enseignement de la matière *Structures de donnée et algorithmique*, nous sommes menés à la réalisation d'un projet (dans notre cas en binôme) qui complétera notre note, testera nos compétences pratiques et qui nous fera même un rappel sur *Algorithme des graphes* ! Ce rapport servira de guide à quiconque voulant comprendre notre code du projet et les étapes abouties afin de trouver la solution de tout problème rencontré.

Le projet consiste en un programme qui trouve le plus court chemin entre deux stations du métro parisien. Ce programme a accès à une base de données contenant les noms et les plans de correspondance entre les stations. Théoriquement, entre deux stations reliées par la même ligne, le trajet durera 1 minute, tandis qu'un changement de ligne durera 5 minutes. Dans cette version du code, un changement de direction ne prendra que 1 minute (information non fournie dans le sujet).

Approches :

1- Dijkstra classique (échec)

Notre première approche consistait à présenter le plan de métro tel qu'il est et appliquer l'algorithme de Dijkstra classique tout en conservant la ligne par laquelle on est arrivé. Dans ce cas, un changement de ligne va incrémenter la distance de 5 et un avancement sur la même ligne de 1. On a grâce à cela un arbre couvrant du plan avec la distance entre chaque station et le nœud de départ.

Après la fin du codage nous nous sommes rendu compte en effectuant plusieurs tests que cette approche aboutie à l'échec !

Malheureusement, en arrivant à une station, pour marquer un sommet, Dijkstra n'a aucun moyen de savoir si un changement de ligne aura lieu ou non après le marquage du sommet et donc ne choisira pas toujours le minimum .

Voici un exemple d'échec :



Figure 1 : Ligne 10

Supposons qu'on voudra aller d'Eglise d'Auteuil (86) à Porte de Saint-Cloud (228)

En arrivant à Michel Ange Molitor, le programme doit faire un choix entre 4 (venant de Boulogne Jean Jaurès) et 7 (venant de Michel Ange Auteuil). Il choisira le min et donc 4. Il sera cependant obligé de changer de ligne s'il choisit de passer par Boulogne Jean Jaurès. La distance totale sera donc de 11 alors que le chemin optimal est de 9 :

Eglise d'Auteuil → Michel Ange Auteuil → Michel Ange Molitor → Exelmans → Porte de Saint Cloud
Contrairement à ce que l'on pense le problème ne vient pas de la forme de la ligne 10 mais plutôt de l'algorithme ! Rajouter le fait qu'un changement de direction coûte 5, corrigera cela, MAIS UNIQUEMENT DANS CE CAS.

Voici un autre contre-exemple où l'algorithme classique ne marche pas et l'ajout de 5 minute pour un changement de direction n'aboutit à rien :

Départ : Abesses (1)

Arrivée : Carrefour Pleyel(43)

Algorithme classique : distance = 20

Le chemin optimal est 18 !

Comment peut-on faire alors ?

2-Nouvelle représentation du plan (Succès)

Si adapter Dijkstra aux contraintes n'a pas marché alors on adaptera les contraintes à Dijkstra !

Comment peut-on faire pour utiliser l'algorithme classique de Dijkstra sur un plan de métro, sachant que les changements de ligne sont à prendre en compte ?

On peut représenter le plan de manière différente :

Pour chaque station, on compte le nombre de ligne que cette station a. Pour chaque ligne on crée un sommet différent ! Par exemple : Trocadéro, dans cette station on peut emprunter la ligne 6 ou 9 . On aura donc deux sommets Trocadéro L6 et Trocadéro L9 .

Comment faire pour les arêtes ?

-un arrêt entre deux sommets de la même ligne a un cout de 1

-un arrêt entre deux sommets de la même station et de ligne différente a un cout de 5

Ainsi en total, pour 304 stations , il y'aura 386 sommet + un sommet **Joker** !

Pourquoi un sommet Joker ? Réponse plus loin.

Voici la représentation de Trocadéro dans notre graphe :

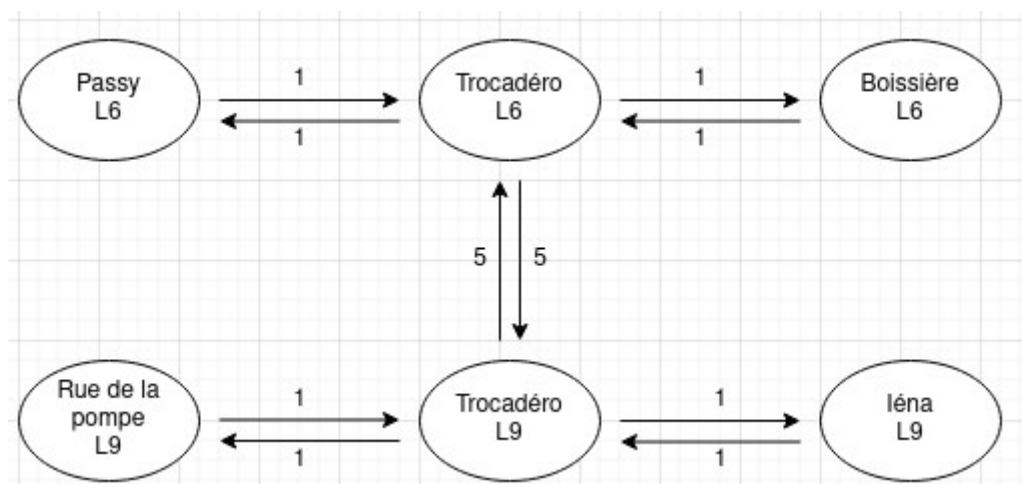


Figure 2 : schéma du graphe implémenté

Problème :

Imaginons que la station de départ a plusieurs lignes, sur quel sommet va-t-on lancer Dijkstra ? Si on lance Dijkstra le nombre de ligne fois et comparer le min, le coût est énorme donc non.

Solution :

Le **Joker** entre en jeu, il est tout simplement un sommet qui a un arrêt vers tous les sommets de départ avec un cout 0. On lancera Dijkstra sur sommet toujours.

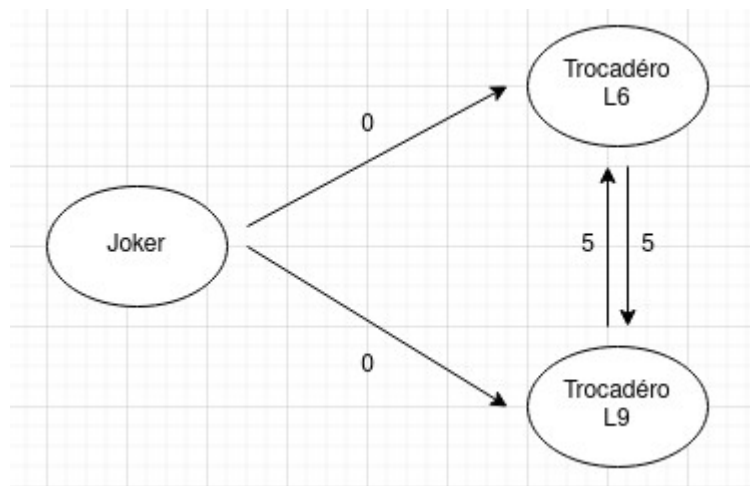


Figure 3 : Utilité du sommet joker

Si la station d'arrivée a plusieurs lignes, on prendra le minimum entre la distance de tous les sommets représentant la station. On utilisera le fait qu'on connaît le nombre de ligne par station.

Implémentation :

Structures utilisées :

Structure pour représenter une station :

```
typedef struct sommet{
    uint32_t id;
    char*ligne;
    uint32_t nbligne;
}sommet;
```

Cette structure va nous servir, lors de la lecture du fichier contenant les arrêts ,à stocker les informations de chaque station .Elle contient un champ **id** un entier non signé qui est l'id de la station et une chaîne de caractère **ligne** qui est le numéro de la ligne et enfin un entier non signé **nbligne** qui est le nombre de correspondance dans cette station .

NB : Chaque station aura nbligne sommets.

Structure nécessaire pour Dijkstra :

```
typedef struct dj{
    uint32_t**ad;
    uint32_t**cout;
}dj;
```

Grace à cette structure Dijkstra s'exécute en ayant accès à toutes les données.

Le champ **ad** est une matrice d'adjacente qui sera de taille 387x387.

Le champ **cout** est une matrice de cout qui sera de taille 387x387.

Exemple : $ad[0][1]=1$ et $cout[0][1]=5$ alors on peut aller du sommet 0 à 1 avec un cout de 5 .

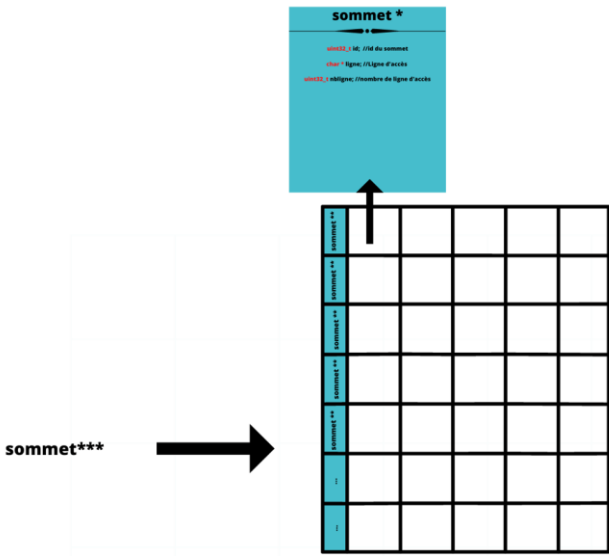
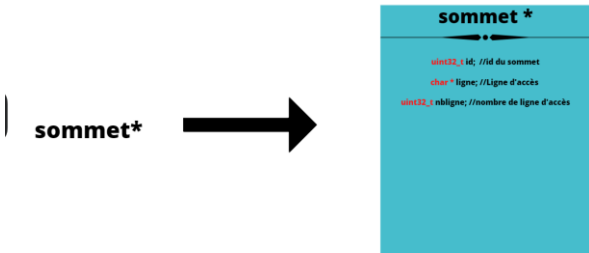
Comment passe-t-on de 304 stations à 386 sommets ?

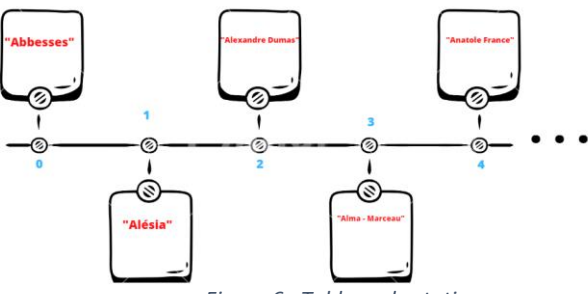
Pour ce faire, nous avons commencé par regarder pour chaque station de métro, combien de ligne permettais l'accès à celle-ci. Par exemple, Trocadéro est accessible par la ligne 9 et la ligne 6. Ainsi dans notre implémentation, chaque station du métro parisien est donc représentée par plusieurs sommets. Chaque sommet représente une ligne permettant l'accès à cette même station puis nous avons généraliser cette approche pour toutes les stations ce qui nous a donc mené à implémenter un graphe à 386 sommets. En effet, comme nous l'avons mentionné ci-dessus une station peut être associé à un ensemble de plusieurs sommets du graphe. Notons $F = \{S_0, S_1, \dots, S_k\}$, l'ensemble de sommet associé à la station A, ainsi naturellement nous avons liées ensemble dans notre graphe tous les élément (sommets) de F ensemble. Chacune de ces liaisons à donc un coût de 5 car celle-ci caractérisent un changement de ligne.

Voici la fonction permettant d'effectuer la tâche correspondante :

```
uint32_t trouversommet(sommet***tab,uint32_t id,uint32_t nbttotal){  
    uint32_t i,j=0;  
    for( i = 0 ; i < id-1 ; i ++){  
        j=j+tab[i][0]->nbligne;  
    }  
    return j;  
}
```


Résumé des fonctions : Tableau explicatif et synthétique :

<pre>sommet ***ouvrir_fichier_station()</pre>	<p>Fonction qui ouvre le fichier Aretes.csv contenant toutes les liaisons possibles entre les stations, initialise une matrice de structure sommets et la remplit.</p> <p>La matrice initialisée nous permettra de savoir pour chaque sommet du graphe combien de ligne lui seront associé. Le nombre de ligne de la matrice sera égale au nombre de station et le nombre de colonne de la matrice sera égale au nombre maximum de ligne.</p>  <p>Figure 4 : Matrice de sommets</p>
<pre>Sommet * init_sommet()</pre>	<p>Fonction qui alloue et renvoie un pointeur sur un sommet. Initialement l'id et nbligne valent 0.</p>  <p>Figure 5 : Structure sommet</p>
<pre>Uint32_t has_line(sommet ***tab,int x,char * s ,uint32_t nbligne)</pre>	<p>Renvoie 1 si s est une ligne de tab[x-1], 0 sinon. Le paramètre nbligne correspond au nombre de colonne de la matrice tab. Il est donc utilisé pour parcourir et comparer toutes les lignes de tab[x-1].</p>
<pre>Void ajouter(sommet ***tab,int num,char*s, uint32_t nbligne)</pre>	<p>Fonction qui ajoute une ligne de métro permettant d'accéder à une station. Plus formellement on ajoute une structure sommet aux tableaux de structure tab[num-1].</p>
<pre>Uint32_t trouversommet(sommet***tab,uint32_t id,uint32_t nbtotal)</pre>	<p>Fonction qui prend en entrée l'identifiant d'une station et renvoie son nouvelle identifiant dans la matrice d'incidence.</p>
<pre>Dj* consdji(sommet***tab)</pre>	<p>Fonction qui initialise la matrice d'incidence ainsi que la matrice de coût pour l'algorithme de Dijkstra.</p>
<pre>uint32_t trouverid(sommet***tab,uint32_t x,uint32_t nbtotal)</pre>	<p>Fonction qui réalise la tâche inverse de la fonction trouversommet. Plus particulièrement, cette fonction prend</p>

	en entrée l'identifiant d'une station dans la nouvelle matrice d'incidence issu de notre algorithme et renvoie l'identifiant initiale de la station tel que dans le fichier Metro Paris Data - Stations.csv.
char*trouverligne(sommet***tab,uint32_t id,uint32_t sommet)	Fonction qui renvoie la ligne correspondant a un sommet du nouveau graphe issu de notre implémentation.
int32_t* dijkstra(uint32_t**G,uint32_t**cost,uint32_t n,uint32_t startnode,uint32_t u,uint32_t nbligne)	Cette fonction implémente l'algorithme de plus court chemin de Dijkstra. Elle prend donc en entrée la matrice d'incidence du graphe, une matrice de coût, leur taille n, le nombre de ligne et enfin un nœud de départ pour pouvoir obtenir l'arborescence des plus court chemin ayant pour racine startnode.
void detruiresommet(sommet*s)	Fonction qui libère la mémoire allouer par un sommet
void detruirestsommet(sommet***s)	Fonction qui désalloue l'espace mémoire de la matrice de sommet
void detruiredj(dj*d,uint32_t nbsommet)	Fonction qui désalloue l'espace mémoire occupé par la matrice d'incidence et la matrice de cout.
char**nomstations()	<p>Fonction qui ouvre le fichier de station et renvoie un tableau de string qui permet de renseigner l'information suivante : tab[i]= string correspondant à la station ayant pour numéro d'index dans le fichier i.</p>  <p>Figure 6 : Tableau de station</p>
void detruirestation(char**station)	Fonction de désallocation du tableau de string ci-dessus.
void destructiontotale(dj*d,sommet***s,char**station)	Fonction qui désalloue tout l'espace mémoire alloué par notre programme. Elle reprend en grande partie les fonctions de désallocation définit ci-dessus.
void affichersolution(int32_t*tab,char**station,somme t***s)	Fonction qui affiche le trajet le plus court grâce à l'arborescence des plus courts chemins issu de l'algorithme de Dijkstra.

Avantages et inconvénients :

Avantage du programme :

- Fonctionnel, donne le plus court chemin dans tous les cas.
- Affiche le trajet en entier
- Flexible, utilise Dijkstra classique il suffit juste de choisir le bon graphe.

Inconvénients :

- N'affiche pas la direction du trajet (par manque de temps).
- Couteux en terme d'espace (construction du graphe).
- La variable nbsommet=386 est calculée à chaque fois alors qu'elle est la meme dans tout le projet (problème de conception).
- Un problème avec la fonction **ajouter** et le champ **sommet→ nbligne** qui reste un mystère :

Si on incrémente **sommet→ nbligne** dans la fonction **ajouter** on reçoit une Erreur de Segmentation. On avait essayé à plusieurs reprises de trouver une solution mais on était bloqué.

Comme la fonction **ouvrir_fichier_station** fait appel à **ajouter** on a décidé d'incrémenter en stockant les valeurs de nbligne donc un tableau et ensuite lui associer la structure.

Code:

```
for(uint32_t i=0 ; i<304 ; i++){  
    nbligne[i]=0;  
}  
  
if(!has_ligne(tab,num1,num3,n)){  
    ajouter(tab,num1,num3,n);  
    nbligne[num1-1]++;  
}  
  
for(uint32_t i=0;i<304;i++) tab[i][0]->nbligne=nbligne[i];
```

Conclusion

En conclusion, ce projet nous a permis de mettre en pratique nos connaissances en algorithmique et en structures de données. Nous avons cherché à trouver le plus court chemin entre deux stations du métro parisien en utilisant l'algorithme de Dijkstra. Après avoir échoué à adapter l'algorithme aux contraintes du métro, nous avons décidé de représenter le plan de métro différemment en créant un sommet pour chaque ligne à chaque station. Cette nouvelle représentation nous a permis de faire correspondre les coûts des déplacements entre les stations aux contraintes du métro. Finalement, notre solution aboutit à un total de 386 sommets pour 304 stations, ce qui nous permet de trouver le plus court chemin entre n'importe quelles deux stations du métro parisien.