Quentin Gruchet,
Thomas Pittis
et Fadi Mechri
vous présentent :

&-BonVoyage®

SOMMAIRES :

INTRODUCTION :

Ce projet nous a été soumis dans le cadre de la deuxième année de Licence d'Informatique. Il consiste en la création d'un programme qui agirait comme une application de voyage à l'exception, qu'ici on ne traite que le Métropolitain Parisien.

Ce rapport contient l'ensemble des éléments du projet. Pour commencer, nous vous présenterons le cahier des charges, regroupant toutes les contraintes qui nous ont été posées.

Nous expliquerons comment lancer et manipuler le programme sans problèmes.
Par la suite nous verrons comment nous avons géré le projet du point de vue humain et avons réparti les tâches.

Nous allons vous expliquer le code pourquoi avoir choisi telle ou telle structure, ainsi que le choix de notre algorithme de recherche du plus court chemin.

Enfin, nous vous énumérons tous les problèmes rencontrés lors de la mise en place du projet et si nous avons réussi à les résoudre.

Pour conclure, nous vous présenterons le ressentis de chacun de l'équipe, ce que nous en avons appris, retenu, ainsi que toutes les possibilités d'améliorations de notre programme.

UTILISATIONS DU PROGRAMME :

Nous devions créer un programme qui trouve le plus court chemin entre deux stations du Métropolitain et ce dans le langage informatique de notre choix (C, C++, Java, Python, …). L'utilisateur choisit doit pouvoir choisir deux stations, une de départ et une d'arrivée, et le programme lui retourne le chemin le plus court possible entre elles. De plus le programme doit détailler le trajet parcouru.

Pour lancer le programme il suffit d'ouvrir un terminal dans le dossier source du code et de taper la commande make dedans. Si le programme a bien été lancé, ceci devrait apparaître :

```
superuser@quentin:~/Bureau/fac/algo/projet metro$ make
gcc -Wall -o metro metro.o lire_ecrire.o graphe.o
./metro
   Bienvenue dans l'appli Ebovoyag. Cette application permet de trouver
 son trajet (le plus rapide) à travers le metropolitain Parisien. Tapez
 'help' pour obtenir la liste des commandes disponibles.
   P.S: N'oubliez pas de quitter le programme pour libérer la mémoire.
La Direction.
```

Le programme est prêt à exécuter vos commandes. En tapant 'help', vous obtiendrez la liste des commandes disponibles :

```
help : affiche le guide des commandes disponibles
ls-l : liste les stations par ligne
ls-a : liste les stations par ordre alphabetique
reseau : affiche le reseau
nombre : affiche le nombres de stations
calcul-l : affiche le trajet suivi par l'algo (permet de débugg)
calcul-a : affiche le trajet sous forme d'un affichage
calcul-t : affiche le trajet sous forme d'affichage + debugg
quitter : quitte le programme
```

Chaque commande est associée à une fonctionnalité du programme. A chaque fois que vous tapez une commande le programme fait l'instruction correspondante et affiche son résultat.

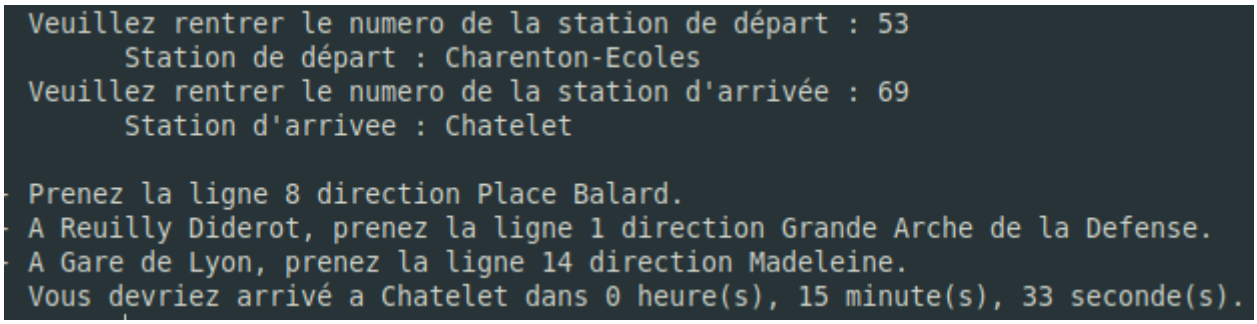
Certaines fonctionnalités doivent prendre un argument pour effectuer précisément ce que veut l'utilisateur

- ls : qui a pour but de lister les stations du programme, a deux possibilités d'argument :

- ✗ -l : liste les stations en fonction de leurs numéros de lignes.
- ✗ -a : liste les stations par ordre alphabétique.

- Calcul : qui est le cœur même du projet. Elle permet de calculer le chemin le plus court entre deux stations choisies par l'utilisateur. Elle peut prendre 3 arguments :
  - ✗ -l : affiche toutes les stations parcourues par l'utilisateur.
  - ✗ -a : affiche le résultat style 'application'.
  - ✗ -t : affiche les stations parcourues et le style 'application'.
  - ✗ -r : affiche un trajet (application + debugg) entre deux stations aléatoires

En cas de réussite du programme, une interface de ce type devrait apparaître :



RÉPARTITION DES TÂCHES :

Tâches\ Prénom	Quentin	Fadi	Thomas
Initialisation stations		X	
Initialisation du réseau			X
Réflexion structures		X	X
Mise en place liste chaînées	X	X	
Algorithme de Dijkstra	X	X	X
Affichage et interactions avec l'utilisateur	X		
Résolution des bugs	X		X

TYPE DE STRUCTURES UTILISES:

Nous avons choisi de coder ce programme en C. Cela nous paraissait le plus judicieux au vu des problèmes éventuels que pourraient nous causer cet exercice. De plus, nous commençons à avoir un peu d'expérience dans ce langage, autant s'en servir.

```
typedef struct sommet {
    int num_sommet;
    int num_ligne;
    int nombre_nom;
    char nom_station[64];
} SOMMET;
```

Pour stocker les informations nous avons décidé d'utiliser des structures. Chaque information est contenue un champ de différentes structures. Cela permet plus de lisibilité dans le code et nous a facilité le codage. Si on avait utilisé des variables globales, la gestion de ces variables devenait compliquée et l'on se perdait facilement dans le code.

```
SOMMET *station;
LIAISON **reseau;
```

Certaines structures prennent en champs un tableau de structures. Ils nous permettent d'obtenir l'intégralité des informations propres à chaque station et ainsi de pouvoir les manipuler à notre guise. Pour manipuler les liaisons entre les stations, l'utilisation des listes chaînées nous semblait évidente. Elles permettent d'obtenir facilement les différents trajets que peut utiliser l'algorithme de recherche du plus court chemin.

Concernant l'algorithme de recherche du trajet le plus court, nous avions au départ choisi l'algorithme de Kruskal, mais cela posait certains problèmes pour la durée d'exécution car nous devions connaître toutes les distances des chaque stations avant d'exécuter l'algorithme. De plus, nous devions vérifier que notre graphe n'était pas connexe. Cet algorithme parcourt plusieurs fois le tableau des distances entre stations ce qui est problématique.

Alors nous avons choisi d'utiliser l'algorithme de Dijkstra car c'est l'un des plus simple à implémenter. De plus, il ne parcourt qu'une seule fois chaque sommet, ce qui le rend plus rapide et optimal.

EXPLICATION DU CODE :

Pour plus de lisibilité, ce programme est découpé en un total de 8 fichiers dont nous vous expliquons pour chacun le fonctionnement et éventuellement les détails techniques.

METRO.TXT

Ce fichier est un fichier de configuration. Il est composé de deux parties dont voici des exemples et leurs explications :

```
V 0000 12 Abbesses
V 0001 2 Alexandre Dumas
V 0002 9 Alma Marceau
V 0003 4 Alésia
```

V : signifie que c'est une ligne de déclaration de station.
0001 : c'est le numéro de la station (relatif au fichier).
12 : ligne sur laquelle se trouve la station.
Abbesses : Nom de la station.

```
E 75 142 93 0256 0213
E 76 156 59 0089 0240
E 76 111 30 0240 0089
E 77 356 57 0066 0130
```

E : signifie que c'est une ligne de liaisons entre stations.
75 & 142 : sont les numéros de la première et de la dernière station.
0256 & 0213 : représente les deux terminus des lignes du trajet entre les deux stations

---

METRO.C

Ce fichier permet de gérer l'interaction avec l'utilisateur et lier tous les fichiers du programme entre eux. C'est dans ce fichier que seront traitées les informations rentrées par l'utilisateur. Il contient deux fonctions :

```
int choix(const char* cmd, GRAPHE graphe){
```

Les informations rentrées par l'utilisateur sont traitées dans cette fonction. Elle prend deux arguments : un tableau dynamique, chaque case contient une lettre tapée par

l'utilisateur ; une structure GRAPHE : elle contient toutes les informations nécessaires du programme, nous y reviendrons un peu plus tard. La fonction compare chaque lettre entrée puis vérifie la correspondance avec la condition if (cf page 4 pour la liste des commandes). Si la commande entrée ne correspond à aucune condition if le programme ne fais rien et attend que l'utilisateur entre une autre commande.
Seuls les champs « primaires » sont initialisés (nom et nombre de stations, liaisons entre stations). Le calcul ne se fait qu'après que l'utilisateur ait tapé la bonne commande.

```
int main(int argc, char* argv[])
```

Cette fonction affiche un message de bienvenue puis initialise les champs de la structure GRAPHE et rentre dans une boucle.

```
do{
    printf("  >>");

    fgets(buffer, 64, stdin);
}while(choix(buffer, graphe));
```

Cette boucle prend dans le while la fonction choix(). Cela veut dire que tant que choix() ne renvoie rien, la boucle ne s'arrête jamais . Bien entendu, choix() ne renvoie quelque chose que si l'utilisateur tape la commande 'quitter'.
Cette fonction exécute aussi libere_graphe() qui permet de libérer toutes la mémoire utilisée par le programme.

DEF.H

Ce fichier (probablement le plus simple de tous) est simplement ici pour plus de lisibilité du code. Il remplace définit par un nom certaines valeurs qui seront appelées dans le code.

```
#define NON_TRAITE 0
```

Par exemple ici 0 sera remplacé dans le code par NON_TRAITE.

LIRE_ECRIRE.H

Ce fichier contient les prototypes de fonctions créées dans lire_ecrire.c.
Grâce à ce fichier nous pouvons utiliser les fonctions de lire_ecrir.c dans le fichier metro.c, il permet de faire le lien entre les deux.
Les noms des fonctions sont assez parlants pour comprendre leurs utilités. Leurs fonctionnements seront expliqués dans l'explication de lire_ecrire.c.

LIRE_ECRIRE.C

Ce fichier est un des plus importants, il permet de lire le fichier metro.txt, ainsi que l'affichage du plus court chemin.
A chaque lecture, il remplit certains champs de chaque structure. Pour plus de clarté, nous l'avons décomposé en plusieurs fonctions.

```
GRAPHE initialise_stations(char *nomFichier, GRAPHE graphe)
```

Cette fonction a pour but de lire le début du fichier .txt (lignes commençantes par 'V'). Elle prend en argument le nom du fichier à lire ainsi qu'une structure de type GRAPHE.
A chaque nouveau tour de la boucle, la fonction remplit toute la structure SOMMET.
Pour être plus clair, la fonction ouvre le fichier de configuration, saute les trois premières lignes d'explications, et remplit les champs avec l'information correspondante dès qu'elle voit une ligne commençant par 'V'. Afin de faire une allocation dynamique des tableaux, cette fonction fait un appel de la fonction compte_nb_sommets() qui permet de compter le nombre de stations.
Elle retourne la structure GRAPHE dont son champs SOMMET a été rempli.

```
GRAPHE initialise_reseau(char *nomFichier, GRAPHE graphe)
```

Cette fonction est assez similaire à la précédente, excepté qu'elle remplit le champ RESEAU de la structure GRAPHE. Elle lit le fichier de configuration et ne commence à remplir la structure, avec les données correspondantes, que lorsqu'elle lit une ligne commençant pas 'E '.

```
void ecrit_chemin(GRAPHE graphe, DIJKSTRA d)
```

Cette fonction permet d'écrire le chemin parcouru dans le terminal. Quand cette fonction est appelée, toutes les structures sont remplies et les calculs déjà faits. C'est la dernière fonction à se lancer. Elle récupère la structure GRAPHE avec les informations de toutes les stations et la structure DIJKSTRA qui contient le chemin le plus court pour le trajet demandé.

GRAPHE.H

Ce fichier est le .h le plus important du programme.
Il contient les prototypes des fonctions de graphe.c et toutes les structures de données énoncées plus tôt.

Dans un premier temps, ce fichier contient toutes les fonctions relatives aux listes chaînées dont voici un exemple :

```c
void affiche_liste(struct elem *l)
{

    if (teste_liste_vide(l)) {
        printf("La liste du chemin est vide \n");
    }
    while (l) {
        printf("%d: %s -> ligne ", l->s.num_sommet, l->s.nom_station);
        if (l->s.num_ligne == 30)
            printf("3bis\n");
        else if (l->s.num_ligne == 70)
            printf("7bis\n");
        else
            printf("%d\n", l->s.num_ligne);
        l = l->suiv;
    }
    printf("\n");
}
```

Ces fonctions permettent de créer une liste, d'y ajouter des éléments, de tester si la liste est vide et de libérer la mémoire utilisée.

Il contient également les fonctions relatives à l'initialisation des variables et tableaux nécessaires à l'utilisation de l'algorithme de Dijkstra.

```c
DIJKSTRA calcul_dijkstra(GRAPHE graphe, int rang_depart, int rang_fin)
```

Enfin voici le corps de ce programme, il ne serait pas grand-chose sans cet algorithme. Cette fonction permet de calculer le trajet le plus court entre la station de départ et la station d'arrivée. Nous commençons par initialisé le tableau contenant tous les sommets traités par une valeur non atteignable par le programme (*NON_TRAITE)*. Nous faisons de même avec le tableau des pères, ; le parcours n'a pas commencé donc aucun sommet n'a de père.
Pour commencer le parcours nous allons indiquer que le sommet de départ a été traité en changeant sa valeur dans le tableau des stations traitées.
Ensuite nous allons faire un parcours de tous les sommets du graphe et vérifier s'il existe une liaison entre la station de départ et tous ces voisins. Si une liaison existe alors nous stockons, dans le tableau *min_Dijkstra*, le temps de trajet entre ces deux stations. On rajoute aussi le rang de départ en tant que père de la *i-ème* station.

Maintenant nous lançons le parcours principal de recherche de plus cours chemin qui nous intéresse.

Tant que tous les sommets n'ont pas été traités nous allons rechercher quel sera le prochain sommet à traiter. Donc si le sommet n'a pas été vu et que la distance minimale entre lui et un de ses voisins est inférieure à *INFINI* (valeur jugée impossible à atteindre), alors nous pouvons prendre le *i-ème* sommet comme sommet à traiter. Nous allons changer la distance minimum par la distance minimale utilisée au-dessus pour le sommet *i*.

On change l'état traité du sommet vu précédemment.
On refait un autre parcours de tous les sommets du graphe à partir du sommet à traiter trouvé au-dessus. S'il existe une liaisons entre le sommet à traiter du dessus et le sommet *i*, et si la distance entre le sommet *i* et le départ est supérieur ou égale à la distance du sommet depuis le sommet de départ + la durée du trajet, alors on a trouvé le minimum pour le moment. On rajoute aussi le sommet que l'on vient de traiter comme père du sommet i.

Tous les sommets devraient avoir été traités, l'algorithme est donc terminé et le chemin le plus cours connu.

```
GRAPHE calcul plus court chemin(GRAPHE graphe, int a, int b, int choix)
```

Ici, cette fonction s'occupe d'appeler la fonction précédente et en fonction de ce que l'utilisateur a entré comme commande, elle affiche la liste des stations parcourues, le type 'affichage' ou les deux.

```
void libere_graphe(GRAPHE graphe)
```

Voici la dernière fonction à être appelée. Elle libère la mémoire allouée pour le programme grâce à la fonction free() .

PROBLÈMES RENCONTRES ET RÉSOLUTION

Lors de la réalisation beaucoup de problèmes sont apparus. Certains mineurs et d'autres un peu plus compliqués :

① Gestion de la mémoire : Comme nous avons choisi le c, pendant le codage du programme nous avons dû faire face à d'énormes fuites de mémoire (parfois près de 40 000 000 bytes perdus). Certaines de ces fuites était dûes à une mauvaise gestion de notre part et d'autre à la conception même de notre code. Nous avons dû réorganiser notre programme pour obtenir un résultat correct.

② Erreurs d'affichage : afficher le résultat du plus court trajet n'a pas été si évident que ça n'y paraît. Nous avons dû réfléchir à quel moment afficher certains éléments et sous quelles conditions. Cela nous a demandé pas mal de réflexion et beaucoup de tests.

③ Gestion des liaisons entre stations : pour gérer ça, nous avions, à la base, utilisé des tableaux qui était modifiés dans beaucoup de fonction. Ces fonctions faisaient beaucoup de manipulation intra-tableaux. A la fin cela devenait fastidieux à suivre qui faisait quoi. De cela découlait de nombreux problèmes qui rendait le programme inexacte. Nous avons donc choisi la solution des listes chaînées.

④ Hiérarchie des fichiers : nous n'étions pas vraiment à l'aise avec la compilation séparée, ce qui nous a donné du fil à retordre pour l'imbrication des fichiers. Nous avons commencé par tout réaliser dans un même .c pour plus de compréhension. Finalement, ce fichier était complètement incompréhensible.

⑤ Remplissage des structures : lire dans les fichiers a été plutôt fastidieux. Nous voulions utiliser la bibliothèque SE_FICHIER.c (que nous avons codé en système d'exploitation au début du semestre). Cependant beaucoup d'erreurs liées à la bibliothèque sont apparues, nous avons donc décider d'abandonner cette bibliothèque.

BILAN ET PERSPECTIVES DU PROJET

Ce projet nous a permis de progresser individuellement et collectivement de bien des façons. En plus d’être un projet pédagogique, il est aussi ludique et nous a donné beaucoup de liberté dans le code et dans la conception. Les améliorations du projet sont nombreuses et demanderaient plus de temps maintenant que les techniques sont acquises :

- Extension avec les transports proposés : Train, Transilien, Tram, RER, Bus, etc..
- Une interface graphique : nous pourrions mettre en place une interface en affichant , par exemple, le plan du métro et le trajet dessus.
- Une sauvegarde des trajets favoris.
- Proposer à l'utilisateur dans quel wagon entrer pour faciliter la sortie.
- Afficher l’horaire de passage des différents moyens de transport.
- Permettre à l'utilisateur de choisir le moyen de transport en en excluant certains et/ou de choisir les lignes à emprunter.
- Optimisation du code : notre code est loin d’être le plus efficace, ni le plus sécurisé. Nous sommes persuadés qu’une version plus efficace est possible.