PROJET METRO

2025

L'objectif de ce projet est de réaliser un plan interactif d'un réseau de transport en commun. Le but est de représenter un ensemble d'objets de nature différentes en ayant la possibilité de faire varier l'échelle d'affichage, de se déplacer dans le plan, et de sélectionner graphiquement un objet. Au delà de la représentation graphique du réseau, une interprétation algorithmique devra être faite, notamment pour déterminer le plus court chemin entre deux noeuds du réseau.

Structures de données

Coordonnées : La localisation des objets sera réalisée en utilisant leurs latitudes et longitudes exprimées en degrés sous forme décimale (les orientations nord et est correspondent aux valeurs positives) :

```
typedef struct _une_coord {

float lon; //Longitude decimale

float lat; //Latitude decimale

} Une_coord;
```

Ce type sera spécifié dans le fichier coord.h.

Stations : Une station est définie par son nom et l'ensemble des connexions qu'elle offre pour rejoindre d'autres stations :

```
typedef struct _une_station {

char *nom; //Le nom
struct _un_truc **tab_con; //Tableau des connexions

unsigned int nb_con; //Nombre de connexions
struct _un_truc *con_pcc; //Connexion du plus court chemin

} Une_station;
```

La structure Une_station sera décrite dans le fichier station.h. A une station est adjointe une connexion qui permet de l'atteindre par un plus court chemin (depuis une autre station initiale).

Connexions : Une connexion correspond à une fraction de ligne unidirectionnelle reliant deux stations. La structure correspondante contient des liens vers les stations de départ et d'arrivée ainsi qu'un lien vers le descriptif de la ligne correspondante.

```
typedef struct _une_connexion {

struct _un_truc *sta_dep; //Station de depart

struct _un_truc *sta_arr; //Station d arrivee

struct _une_ligne *ligne; //Ligne
} Une_connexion;
```

Les connexions seront décrites dans le fichier connexion.h.

Trucs : Comme les stations et les connexions vont faire l'objet de traitements analogues et contenant pour une part le même type d'informations (coordonnées, valeur), il a paru opportun au concepteur de ce projet de les regrouper sous la forme d'une structure unique nommée truc.

```
typedef enum _ttype {STA, CON} Ttype;

typedef union _data {

    Une_station sta;
    Une_connexion con;
    } Tdata;


typedef struct _un_truc {

Une_coord coord;
Ttype type;
Tdata data;
float user_val; //Distance pour plus court chemin

 } Un_truc;
```

Un truc ne pouvant à fois correspondre à une station et à une connexion, l'utilisation d'une union est bien venue. Aux données portées en propre par une station ou une connexion sont adjointes des coordonnées coord et une valeur user_val utilisée lors du calcul du plus court chemin et du tri des éléments dans les listes. Pour mémoire, une union permet d'utiliser un même emplacement mémoire pour stocker des valeurs de types différents à des instants différents. La syntaxe est identique à celle des structures, on accède à ses champs en utilisant la notation point (.) mais un seul n'est accessible à un instant donné en

fonction du type. Dans notre cas nous accéderons soit à une station soit à une connexion. L'idée est de n'avoir qu'une seule version d'une station en mémoire et de ne pas les dupliquer. Les trucs seront donc essentiellement manipulés au travers de pointeurs.

Listes : Nous définissons une bibliothèque permettant la manipulation de listes simplement chaînées de truc.

```
typedef struct _un_elem {

Un_truc *truc; //Une station ou une connexion

struct _un_elem *suiv;
} Un_elem;
```

Le fichier liste.h contiendra la définition du type et les prototypes des fonctions utiles pour gérer ces listes.

ABR : Comme nous aurons fréquemment besoin d'effectuer des recherches efficaces de stations à partir de leurs noms, nous avons décidé d'employer des arbres binaires de recherche.

```
typedef struct _un_nabr {

Un_truc *truc; //La station
struct _un_nabr *g; //Fils gauche strictement inferieur

struct _un_nabr *d; //Fils droit
} Un_nabr;
```

Le fichier abr.h contiendra les prototypes des fonctions permettant la manipulation de ces arbres de recherche. Le fichier abr_type.h contiendra la définition du type.

Lignes : Une ligne est définie par son code, la vitesse moyenne des rames qui l'utilisent, l'intervalle moyen entre deux rames et la couleur à utiliser pour la dessiner.

```
typedef struct _une_ligne {

char *code; //Le nom de la ligne A, B .., M1, M2, T1...

char *color; //La couleur de la ligne #RRGGBB
float vitesse; //Vitesse moyenne des rames en km/h

float intervalle; //Intervalle moyen entre 2 rames

struct _une_ligne *suiv;
```

} Une_ligne;

Les lignes sont chaînées entre elles et seront décrites dans le fichier ligne.h.

AQR : Nous allons utiliser des arbres quaternaires (généralisation à deux dimensions des ABR) pour représenter la topologie du réseau. Le plan à afficher correspond au rectangle englobant l'ensemble des objets (station ou connexion) présents. Ce plan est généralement découpé en quatre rectangles de dimensions égales. Si un de ces rectangle contient plus d'un objets à afficher, il sera lui même découpé en quatre. Ce processus sera répété jusqu'à ce que tous les objets soient portés par les feuilles de cet arbre. Un noeud de cet arbre peut avoir entre 0 et 4 fils suivant la répartition des objets dans les 4 partitions que ce sous-arbre contient.

typedef struct _un_noeud {

Un_truc *truc; //*Une station ou une connexion*

Une_coord limite_no; //*Limite zone*
Une_coord limite_se; //*Limite zone*
struct _un_noeud *no; //*Fils pour quart NO*

struct _un_noeud *so; //*Fils pour quart SO*

struct _un_noeud *ne; //*Fils pour quart NE*

struct _un_noeud *se; //*Fils pour quart SE*

} Un_noeud;

Cette représentation a deux objectifs : permettre un accès efficace à un objet à partir de ses coordonnées (éventuellement grossièrement approchées) et déterminer rapidement l'ensemble des objets contenus dans une portion du plan à afficher (particulièrement utile lors des changements d'échelle et déplacements). Cette structure sera décrite dans le fichier aqrtopo.h.

Exercice 1 – Fichier stations (10%)

La description de l'ensemble des stations sera fournie dans un fichier de type .csv (que vous allez créer), chaque ligne de ce fichier contient 3 champs séparés par des points-virgules : la longitude, la latitude et le nom de la station.

2.354660 ; 48.845990 ; Jussieu

Dans un premier temps, vous allez écrire l'ensemble des fonctions permettant de construire des listes de stations ordonnées suivant leur latitude. Ce choix d'ordre est guidé par la possibilité qu'il offre d'obtenir dans un second temps à partir de ces listes des ABR pas trop déséquilibrés. Il est en effet peu probable qu'il ait une corrélation entre le nom d'une station et sa latitude. Comme nous aurons plus tard besoin de listes triées suivant la valeur du champ user_val pour le calcul du plus court chemin, il suffira d'initialiser ce champ avec la latitude pour n'avoir à intégrer qu'un seul critère de tri.

1. Ces stations étant représentées par des truc, écrivez dans le fichier truc.c les fonctions suivantes :

```
Un_truc *creer_truc(Une_coord coord, Ttype type, Tdata data, double uv);

void detruire_truc(Un_truc *truc);
```

La fonction creer_truc réalise l'allocation dynamique d'un truc et l'initialise en utilisant les arguments fournis. La fonction detruire_truc désalloue la mémoire allouée avec malloc. Si le truc est une station pensez à désallouer : le nom (nom) et le tableau des connexions (tab con). Si le truc est une connexion, vous n'avez rien à désallouer (à l'exception du truc évidement).

2. L'objectif étant de constituer des listes de stations, écrivez dans le fichier liste.c les fonctions suivantes :

```
Un_elem *inserer_liste_trie(Un_elem *liste, Un_truc *truc);

void ecrire_liste( FILE *flux, Un_elem *liste);

void detruire_liste(Un_elem *liste);


void detruire_liste_et_truc(Un_elem *liste);
```

Pour mémoire la liste doit être ordonnée de façon croissante suivant la valeur du champ user_val. Pour la fonction ecrire_liste vous pouvez vous contenter de réaliser l'affichage des truc correspondant à des stations. Pour les affichages des listes de stations vous pourrez reproduire le format du fichier .csv contenant les stations. Pour la fonction detruire liste et truc il vous suffira de faire appel à detruire truc pour chaque élément.

3. Ecrivez maintenant la fonction lire_stations réalisant la construction de la liste des stations à partir du contenu du fichier .csv.

```
Un_elem *lire_stations( char *nom_fichier);
```

Attention, les noms de stations peuvent contenir des espaces et donc vous ne pouvez pas utiliser le format %s de scanf (voir manuel). Nous vous conseillons pour chaque ligne de procéder en deux temps :

- identifier les différents champs délimités par des (;) en remplaçant au passage par des marqueurs de fin de chaîne,

- interpréter les champs différemment en fonction de leurs formats.

4. Pour pouvoir afficher les stations sur une carte il faut être capables de déterminer le rectangle englobant toutes les stations. C'est à dire les longitudes et latitudes minimales et maximales des stations (ou plus généralement « truc »).

Ecrivez dans le fichier liste.c la fonction limites_zone.

```
void limites_zone(Un_elem *liste, Une_coord *limite_no, Une_coord *limite_se);
```

limite_no correspond à la longitude minimale et la latitude maximale, limite_se correspond à la longitude maximale et à la latitude minimale.

Exercice 2 – ABR stations (10%)

Notre objectif pour cette partie est de construire un arbre binaire de recherche accélérant l'accès aux stations déjà présentes dans une liste de truc correspondant à des stations. L'idée est de ne pas dupliquer les stations. La destruction de l'arbre ne devra donc pas libérer la mémoire allouée pour les stations.

1. Dans un premier temps écrivez la fonction creer_nabr qui réalise l'allocation d'un noeud et son initialisation :

```
Un_nabr *creer_nabr(Un_truc *truc)
```

2. Il vous faut maintenant écrire la fonction inserer_abr qui insère un noeud à sa place dans un ABR :

```
Un_nabr *inserer_abr(Un_nabr *abr, Un_nabr *n)
```

3. Ecrivez les fonctions:

```
Un_nabr *construire_abr(Un_elem *liste_sta);
void detruire_abr(Un_nabr *abr);
Un_truc *chercher_station(Un_nabr *abr, char *nom);
```

L'objectif de cet ABR étant de rechercher une station par son nom, c'est la fonction strcmp qui permettra d'ordonner l'arbre. La fonction construire abr devra laisser la liste source inchangée et retourner un pointeur sur la racine de l'ABR. Vous écrirez vous fonctions dans le fichier abr.c.

Exercice 3 – Listes de lignes de metro (10%)

Un fichier .csv (que vous allez créer) contient l'ensemble des lignes du réseau, chaque ligne de ce fichier correspond à la description d'une ligne :

```
A ; 55.0 ; 15.0 ; #808080
```

Elle contient dans l'ordre le code de la ligne, la vitesse moyenne (en km/h), l'intervalle moyen entre rames(en minutes) et la couleur.

1. Ecrivez les fonctions:

```
Une_ligne *lire_lignes(char *nom_fichier);
void afficher_lignes(Une_ligne *lligne);
void detruire_lignes(Une_ligne *lligne);
Une_ligne *chercher_ligne(Une_ligne *lligne, char *code);
```

La fonction lire_lignes parcours le fichier .csv fournis en argument et construit la liste de Une_ligne correspondante. La fonction afficher lignes permet d'afficher sur le flux de sortie standard (stdout) les lignes contenues dans une liste en respectant le format du fichier .csv d'origine. detruire_lignes réalise la désallocation complète d'une liste de lignes et chercher ligne permet de rechercher une ligne par son code.

Exercice 4 – Connexions (10%)

Comme les stations, les connexions seront stockées dans des truc, elles seront toutes regroupées dans des listes de truc mais devront aussi être accessibles à partir de leur station de départ.

Lors de la lecture du fichier décrivant les connexions au format .csv, vous devrez créer les truc correspondants, les insérer dans la liste des stations et mettre à jour le tableau de connexions accessibles des stations concernées. Vous devrez vérifier que la ligne existe bien et si la valeur fournie pour une connexion est égale à 0.0 calculer à partir de la vitesse de la ligne et de la distance qui sépare les stations le temps de parcours (en minutes).

Chaque ligne du fichier contient soit un commentaire si elle commence par le caractère #, soit la description d'une connexion comme suit :

B ; Chatelet Les Halles ; Gare du Nord ; 0.0

1. La liste des connexions n'ayant pas de raison d'être triée ajoutez la fonction d'insertion en debut de liste au fichier liste.c :

```
Un_elem *inserer_deb_liste(Un_elem *liste, Un_truc *truc);
```

2.Ecrivez maintenant la fonction de lecture des connexions:

```
Un_elem *lire_connexions(char *nom_fichier, Une_ligne *liste_ligne, Un_nabr *abr_sta);
```

Exercice 5 – AQR (20%)

Comme indiqué au début du sujet nous allons constituer un arbre quaternaire correspondant à la répétition de quadri- partitions en 4 rectangles de tailles égales. Vous aurez à écrire 5 fonctions :

1. Ecrivez la fonction d'insertion dans l'AQR.

```
Un_noeud *inserer_aqr(Un_noeud *aqr, Une_coord limite_no, Une_coord limite_se, Un_truc *truc);
```

Les coordonnées limite_no et limite_se ne sont nécessaires que lors du premier appel lorsque aqr est égal à NULL, elles seront nécessaires pour établir les lignes de partition et pourront facilement être obtenues en utilisant la fonction limites zone. Attention il est impossible d'insérer deux trucs aux mêmes coordonnées, cela correspondrait à un arbre de profondeur infinie. Vous devrez donc évaluer la distance séparant le truc à insérer d'un truc déjà présent dans l'arbre. Si cette distance est inférieure à un seuil que vous aurez pris le soin

d'évaluer de façon pertinente par rapport au problème que vous traitez, nous vous suggérons de modifier les coordonnées du truc à insérer de ce seuil.

2. Ecrivez la fonction construire_aqr qui crée un AQR à partir des trucs contenus dans une liste.

```
Un_noeud *construire_aqr(Un_elem *liste);
```

3. Ecrivez la fonction de destruction d'un AQR.

void detruire_aqr(Un_noeud *aqr);

4. Ecrivez la fonction réalisant la recherche d'un truc à partir d'une coordonnée.

```
Un_truc *chercher_aqr(Un_noeud *aqr, Une_coord coord);
```

La fonction retournera le pointeur sur le truc contenu dans la feuille de l'arbre correspondante.

5. Ecrivez la fonction permettant d'obtenir une liste des truc contenus dans une zone rectangulaire :

```
Un_elem *chercher_zone(Un_noeud *aqr, Un_elem *liste, Une_coord limite_no, Une_coord limite_se);
```

La fonction devra construire une liste de truc et devra retourner un pointeur sur le premier élément de cette liste.

Exercice 6 – Plus court chemin (20%)

Maintenant que nous avons construit le graphe représentant le réseau nous disposons de toutes les structures nous permettant d'appliquer un algorithme de recherche du plus court chemin. Nous vous proposons pour ce travail d'implanter l'algorithme de Dijkstra. La mise oeuvre de cet algorithme nécessite l'utilisation d'ensembles ordonnés de noeuds qui dans notre cas vont correspondre aux stations de notre réseau. Nous allons devoir ajouter deux fonctions d'extraction à celles déjà écrites dans le fichier liste.h :

1.

```
Un_truc *extraire_deb_liste(Un_elem **liste);
Un_truc *extraire_liste(Un_elem **liste, Un_truc *truc);
```

Ecrivez ces fonctions en prenant soin de libérer la mémoire correspondant aux éléments extraits.

L'objectif de l'algorithme de Dijkstra appliqué à notre problème est de calculer pour toutes les stations du réseau le temps minimal de parcours qui la sépare d'une station origine, le point de départ de notre recherche. Pour chaque station cet algorithme permet de mémoriser la connexion y menant et faisant partie du plus court chemin depuis la station origine. Il est alors aisé en réalisant un parcours de l'arrivée au départ de déterminer le plus court chemin entre ces deux stations.

Nous pouvons énoncer l'algorithme de Dijkstra de la façon suivante :

1. Initialiser la valeur de la station d'origine (user val) à 0.0.
2. Initialiser la valeur de toutes les autres stations à $+\infty$.
3. Constituer Q un ensemble de toutes les stations ordonnées suivant leurs valeurs croissantes.
4. Tant que Q est non vide en extraire la première station.
5. Pour toutes les connexions partant de cette station, mettre à jour les valeurs des stations destinations de ces connexions si nécessaire (nouvelle valeur calculée du temps de parcours inférieure) tout en préservant l'ordre de l'ensemble Q. Si la valeur d'une station est mise à jour mémoriser la connexion. Répéter 4.

2. Ecrivez la fonction correspondant à cet algorithme:

```
void dijkstra(Un_elem *liste_sta, Un_truc *sta_dep);
```

Si vous voulez obtenir un temps réaliste il faudra considérer les temps nécessaires aux correspondances et aux arrêts en station.

3. Il ne vous reste plus qu'à reconstituer la liste des connexions correspondant au plus court chemin jusqu'à une station destination, écrivez la fonction :

```
Un_elem *cherche_chemin(Un_truc *sta_arr);
```

4. Ecrivez un programme permettent de tester votre recherche de plus court chemin, vérifiez que le temps calculé n'est pas aberrant.

Exercice 7 – Interface graphique (20%)

Faites le design d'une interface graphique pour votre application et écrire le programme correspondant. Le programme principal de votre application lancera l'interface graphique permettant à l'utilisateur de votre application de choisir une fonctionnalité.

Le projet sera rendu sous la forme d'une archive .zip ou .tar. L'archive devra contenir :

- les sources de votre application : tous les fichiers .c et .h
- les fichiers .csv permettant de tester votre programme
- un Makefile permettant de compiler votre application (à tenir compte de toutes les indications données lors du cours « Compilation séparée)
- un fichier Readme.txt permettant à un utilisateur de comprendre le fonctionnement de l'application. Le fichier Readme doit contenir une explication générale de l'application et de ses fonctionnalités, la manière dont on lance l'exécution de l'application (le client n'est pas sensé lire le Makefile).

Le non respect des consignes ci-dessus sera pénalisé 5 points sur la note finale.