LI215 - Plan Interactif
semaine 8

Objectif(s)

- ✯ Listes simplement chaînées.
- ✯ Arbres binaires de recherche.
- ✯ Utilisation des unions.
- ✯ Retour sur les fichiers textes.
- ✯ Utilisation d'arbres quaternaires.
- ✯ Introduction à la cartographie numérique.
- ✯ Introduction aux graphes.

Présentation

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

Le réseau de transport RER, Métro et Tramway d'île de France servira d'illustration à ce problème et les fichiers le décrivant sont fournis. Libre à vous d'enrichir ce réseau en complétant les fichiers fournis (`Stations_MRT.csv`, `Connexions_MRT.csv` et `Lignes_MRT.csv`).

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 est spécifié dans le fichier `coord.h`.

Stations : Une station est définie par son nom, 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;
```

À une station décrite dans le fichier `station.h` 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 sont décrites dans le fichier `connexion.h`.

Trucs : Comme les stations et les connexions vont faire 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 utilisée lors du calcul du plus court chemin.

L'idée est de ne pas dupliquer les stations d'où l'utilisation de pointeurs sur des trucs.

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` contient 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` contient les prototypes des fonctions permettant la manipulation de ces arbres de recherche. Le fichier `abr_type.h` contient la définition du type et a du être séparé du fichier `abr.h` pour éviter les inclusions circulaires.

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 décrites dans le fichier `ligne.h`.

AQR : Nous allons utiliser des arbres quaternaires 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 est décrite dans le fichier `aqrtopo.h`.

Interface graphique

L'objectif est évidement d'offrir une interface graphique à ce mini-projet. Elle sera fournie sous une forme minimale comportant :

- Une zone de dessin pour le plan,
- quatre boutons commandant le déplacement par demi-écran suivant quatre directions (Nord, Sud, Est et Ouest),
- deux boutons permettant les zoom avant et arrière d'un facteur 2,
- un bouton permettant la sélection de deux stations et restreignant l'affichage au plus court chemin les reliant.

Vous pourrez utiliser cette interface et la compléter mais dans un premier temps nous vous invitons à tester vos fonctions séparément dans un terminal, des petits programmes de test sont fournis pour cela.

1 Listes de stations

La description de l'ensemble des stations est fournie dans un fichier de type `.csv`, chaque ligne de ce fichier contient 3 champs séparés par de 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_al` 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.

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);
```

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);

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

Vous prendrez bien soin de libérer la mémoire allouée pour les chaînes de caractères et tableaux de connexions. Pour les affichages des listes de stations et connexions vous pourrez reproduire celui des fichier `.csv`, voir plus loin pour les connexions.

Testez vos fonctions, vous pourrez pour cela utiliser partie du fichier `test_sta.c` fourni.

2 Arbre binaire de recherche de stations

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 pas libérer la mémoire allouée pour les stations.

Écrivez 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);
```

Testez vos fonctions.

3 Listes de lignes

Un fichier `.csv` 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.

Écrivez 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);
```

Testez vos fonctions, vous pourrez pour cela utiliser le fichier `test_ligne.c` fourni.

4 Connexions

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
```

La liste des connexion n'ayant pas de raison d'être triée ajoutez la fonction d'insertion en début de liste au fichier `liste.c` :

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

Écrivez maintenant la fonction de lecture des connexions :

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

Testez vos fonctions, vous pourrez pour cela utiliser le fichier `test_connexion.c` fourni.

5 Plus court chemin

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` :

```
Un_truc *extraire_deb_liste(Un_elem **liste);

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

Écrivez 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 cours 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 jours mémoriser la connexion. Répéter 4.

Écrivez la fonction correspondant à cet algorithme :

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

Si vous voulez obtenir un temps calculé correct vous aurez intérêt à considérer les temps nécessaires aux correspondances et aux arrêts en station.

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

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

Écrivez un petit programme permettent de tester votre recherche de plus cours chemin, vérifiez que le temps calculé n'est pas aberrant.

6 Arbre quaternaire pour affichage

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 :

– Construction de l'arbre :

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

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

– Destruction de l'arbre :

```
void detruire_aqr(Un_noeud *aqr);
```

– Une fonction de recherche d'un `truc` à partir d'une coordonnée.

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

– Une fonction permettant d'obtenir une liste des `truc` contenus dans une zone rectangulaire :

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

