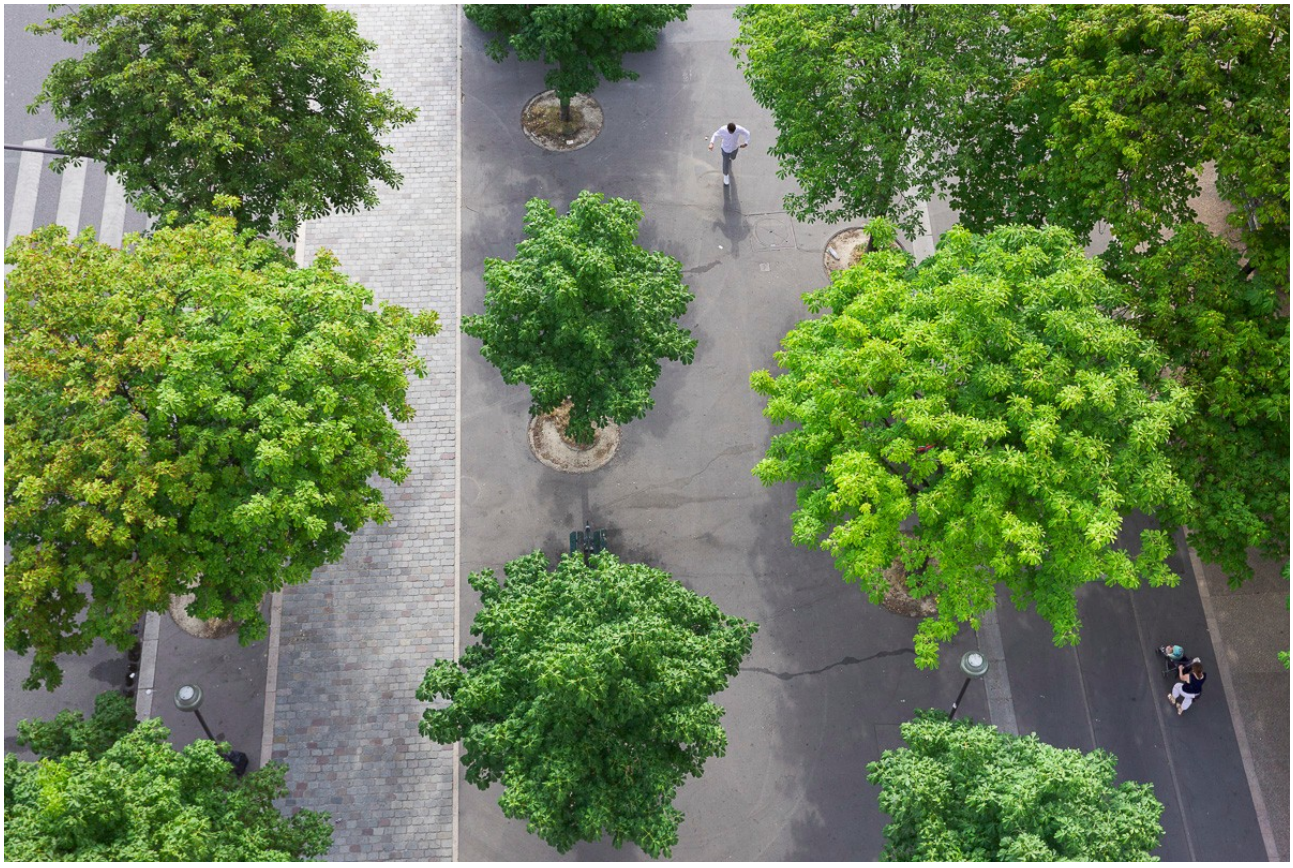
BAUMANN
Valentine
1A IR

École d'ingénieurs
Télécom Physique
Université de Strasbourg

Projet Programmation - 2020

Rapport



Jeu de données

Le jeu de données utilisé ici est fourni en OpenDatabase par la mairie de Paris et recense l'ensemble des arbres présents sur le territoire parisien. Il est au format **CSV** (un seul fichier, les données sont séparées par des points-virgules).
Les champs mis à disposition sont les suivants :
`IDBASE ; TYPE EMPLACEMENT ; DOMANIALITE ; ARRONDISSEMENT ; COMPLEMENT ADRESSE ; NUMERO ; LIEU / ADRESSE ; IDEMPLACEMENT ; LIBELLE FRANCAIS ; GENRE ; ESPECE ; VARIETE OUCULTIVAR ; CIRCONFERENCE (cm) ; HAUTEUR (m) ; STADE DE DEVELOPPEMENT ; REMARQUABLE ; geo_point_2d .`

J'ai choisi de garder les champs suivants pour remplir les structures :

`IDBASE`	Identifiant unique pour chaque arbre, type `int`	`162823`
`geo_point_2d`	Couple de coordonnées géographiques (système géodésique réduit à la latitude et à la longitude, en degrés décimaux), type `geo_point_2d`	`48.7575482555, 2.38328567067`

Construction du graphe

À partir du graphe formé par tous les arbres de Paris reliés par des cordes, on veut pouvoir extraire le plus petit sous-graphe reliant tous les arbres. Ce graphe est connexe, non-orienté et pondéré.

On utilise l'un des algorithmes de minimisation vu en cours de Théorie des Graphes pour déterminer le sous-graphe connexe de poids minimum (arbre couvrant de poids minimum).

J'ai choisi d'utiliser l'algorithme de Kruskal.
Celui-ci repose sur le principe suivant : on trie les arêtes par poids croissant puis on les ajoute au graphe tant qu'elles ne créent pas de cycle. On stoppe l'algorithme dès que $|A_i| = n-1$, où A_i ensemble des arêtes du sous-graphe et n le nombre de sommets du graphe.

- Pseudo-code de l'algorithme de Kruskal (version additive) :

```
Kruskal(G) :
1   A := ∅ //initialisation de l'ensemble des arrêtes
2   pour chaque sommet v de G :
3      créerEnsemble(v)
4   trier les arêtes de G par poids croissant
5   pour chaque arête (u, v) de G prise par poids croissant :
6      si find(u)!= find(v) :
7         ajouter l'arête (u, v) à l'ensemble A
8         union(u, v)
9   renvoyer A
```

G est le graphe d'origine (connexe, non-orienté). La fonction **créerEnsemble** crée une classe d'équivalence pour le sommet placé en paramètre.
La fonction **find** renvoie la classe d'équivalence de l'élément placé en paramètre et la fonction **union** fusionne deux classes d'équivalence.

Pour former le graphe, il faut créer des arêtes reliant les arbres de façon à former un graphe connexe.

- Pour chaque couple d'arbres à relier, il faut donc calculer à partir des coordonnées fournies la distance qui les sépare. Elle constituera la pondération de l'arête reliant deux nœuds du graphe.

On applique ici la formule de **haversine** (permet de calculer le périmètre du grand cercle entre deux points d'une sphère) :

$$a = \sin^2\left(\frac{\varphi_B - \varphi_A}{2}\right) + \cos(\varphi_A)\cdot\cos(\varphi_B)\cdot\sin^2\left(\frac{\lambda_B - \lambda_A}{2}\right) \quad ,$$

où φ_A, φ_B latitudes des points A, B et λ_A, λ_B longitudes des points A, B en radians.

La distance entre deux arbres est donc : $d = 6371 \cdot 2\arctan\left(\frac{\sqrt{a}}{\sqrt{1-a}}\right) \cdot 10^3$ mètres.

Les fonctions `calcul_distance` et `convert_rad` (prototypes dans `distance.h`) mettent en place ces calculs.

- Représentation du graphe en mémoire :

J'ai choisi de représenter le graphe en mémoire à l'aide d'une structure de données allouée dynamiquement :

```
graph G ;

if ((G.pond = malloc(nb_lig*sizeof(int*))) == NULL) {
    fprintf(stderr, "Malloc graphe a échoué.\n") ;
    exit(1) ;
}

for(int i=0 ; i<nb_lig ; i++)
{
    if (((G.pond)[i] = malloc(nb_lig*sizeof(int))) == NULL) {
        fprintf(stderr, "Malloc graphe a échoué.\n") ;
        exit(1) ;
    }
}

if (((G.id_arbre) = malloc(nb_lig*sizeof(int))) == NULL) {
    fprintf(stderr, "Malloc graphe a échoué.\n") ;
    exit(1) ;
}
```

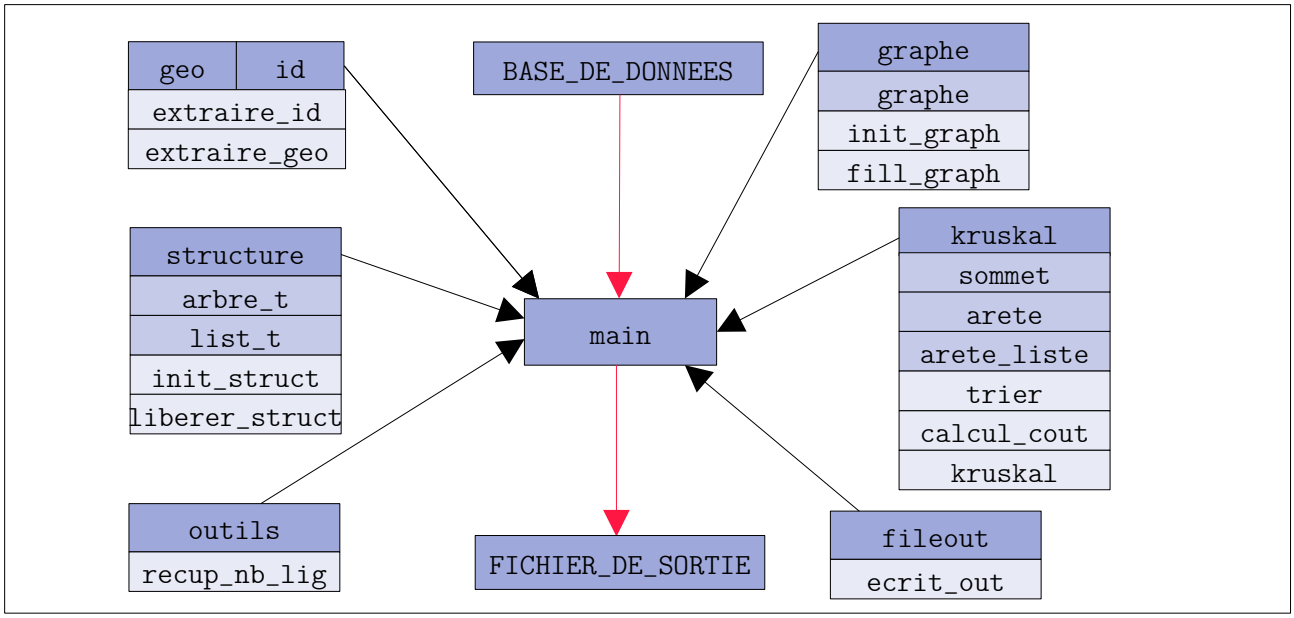
Ainsi chaque pondération d'arête du graphe est accessible de la façon suivante :

`Graphe.pond[sommet_1][sommet_2] ;`

Et chaque identifiant d'arbre de la façon suivante : `Graphe.id_arbre[sommet]` .

Structure du programme

- Schéma de l'architecture logicielle du programme :



Le programme est organisé de façon à découper les fonctions d'extraction-récupération des données (`geo`, `id`, `outils` et `structure`) ; d'initialisation-remplissage du graphe (avec `graphe`) d'application de l'algorithme (fonctions et struct de `kruskal`) et enfin d'écriture des résultats (`fileout`).

Choix d'implémentation

- Choix des structures de sauvegarde des données en mémoire (après la lecture du fichier CSV) :

```
typedef struct arbre_t {
    int id_base ;
    long double geo_lat ;
    long double geo_long ;
} arbre_t ;

typedef struct list_t {
    arbre_t *tab ;
} list_t ;
```

La structure `arbre_t` comprend les 3 éléments essentiels pour la mise en graphe : l'identifiant propre à chaque arbre, sa latitude et sa longitude.

La structure `list_t` comprend un pointeur vers un élément `tab` de types `arbre_t`. Elle constitue un ensemble de pointeurs vers `arbre_t`.

- Choix des structures du graphe pour leur utilisation dans `kruskal` :

Pour l'implémentation de l'algorithme de Kruskal, on utilise une liste d'arêtes.
On définit pour cela trois structures : sommet, arete et arete_liste.

```
typedef struct sommet {
    int num_sommet ;
    int classe ;

} sommet ;
```

```
typedef struct arete {
    sommet u,v ;
    int pond ;

} arete ;
```

```
typedef struct arete_liste {
    arete *a ;
    int nb_arete ;
} arete_liste ;
```

Un sommet est caractérisé par son numéro de sommet et la classe à laquelle il appartient.
Une arête comporte deux sommets et une pondération.
La liste d'arêtes est donc une structure donnant accès à un pointeur vers `arete` et au nombre d'arêtes qu'elle comporte.

L'algorithme que j'ai implémenté dans `kruskal` commence par remplir le graphe initial (à savoir la liste `A_liste`) à l'aide des données contenues dans le `graph` G placé en paramètre puis trie les arêtes par ordre croissant avec la fonction `trier` : itération après itération, les arêtes de poids plus faible sont avancées vers le début de la liste.
Ensuite, l'algorithme parcourt les sommets du graphe d'origine : si ces sommets ne sont pas dans la même classe (ils ne sont donc pas encore reliés), on les enregistre comme sommets du nouveau graphe et on fusionne leurs classes. Cette fonction renvoie la nouvelle liste d'arêtes `new_liste`.

Difficultés rencontrées

- Stockage des arêtes du graphe en mémoire :

Pendant les premiers tests que j'ai mené (en réduisant un peu la base de donnée des arbres de Paris) j'ai remarqué qu'à partir d'un certain nombre d'arbres (environ 36), le stockage est défaillant. L'erreur à lieu au moment de l'accès par la fonction `kruskal` au graphe G passé en paramètre.
L'allocation mémoire se fait bien (`malloc` n'échoue pas, aucun message d'erreur), cependant durant la session de debug, en essayant de 'print' des valeurs, GDB renvoie l'errur « `Cannot access memory at adress Ox(...)` ».

Le programme n'est donc fonctionnel que pour un jeu de données assez réduit.
Si le nombre d'abres n'est pas trop important, il renvoie dans le fichier de sortie la liste des arbres (identifiant `id_base`) et la longueur minimale de corde nécessaire pour les relier.

- Représenter le graphe pour appliquer Kruskal :

Une autre difficulté à laquelle j'ai été confrontée a été la façon de représenter le graphe en mémoire, et notamment l'attribution d'une classe d'équivalence pour chacun des sommets du graphe (celui-ci étant stocké avec des listes d'arêtes et non de sommets). Finalement, j'ai trouvé une solution au problème en ne parcourant plus dans la boucle les arêtes du graphe d'origine mais ses sommets (taille connue puisqu'ils sont aussi nombreux que ceux du nouveau graphe). J'ai dû pour cela faire des modifications sur la façon d'indicer les arêtes (un graphe à n sommets que l'on relie tous comportera $\frac{n(n-1)}{2}$ arêtes).

La réalisation de ce projet m'a permis de me rendre compte des progrès effectués depuis le début de cette année. J'ai apprécié réfléchir aux structures du programme, organiser son fonctionnement et débuger.