TP n °5 : Arbres couvrants

Dans ce TP, on s'intéresse aux arbres couvrants de poids minimal dans le contexte d'un graphe géométrique (les sommets sont des points, toutes les arêtes sont définies, et leur poids est égal à la distance entre leurs extrémités).

1 Introduction

Le problème que nous allons résoudre ici est celui de concevoir le plus petit réseau possible reliant un ensemble de villes. Il se trouve que si l'on pose ce problème avec un formalisme de graphe avec arêtes pondérées, le réseau voulu correspond à l'arbre couvrant de poids minimal (Minimal Spanning Tree). Pour trouver cet arbre, nous allons tester deux algorithmes : celui de Kruskal et celui de Prim.

Le dossier de code qui vous est fourni est organisé comme suit :

- Le répertoire `common` : contient les fichiers pour la structure du graphe, le parser (lecture des fichiers de données) et le générateur de `.geojson`, utilisé pour la visualisation. Vous n'avez pas à les modifier et vous n'avez pas besoin de les lire.
- Le répertoire `eleves` : contient les fichiers que vous avez à compléter. Il y a :
  - `union-find.c` : structure d'Union-find, utilisée dans l'algorithme de Kruskal ;
  - `edge-sorting.c` : trie en place un tableau d'arêtes ;
  - `min-spanning-tree.c` : là où il faudra implémenter les algorithmes de Kruskal et de Prim ;
  - `min-heap.c` : structure de tas-min, utilisée dans l'algorithme de Prim.
- Le fichier `tp11.c` : point d'entrée du programme, il contient la fonction main. Vous n'avez pas à le modifier ni à le lire.
- Les fichiers `test_edge-sorting.c` et `test_union-find.c` : pour tester vos fonctions de bases.
  Vous êtes encouragés à les utiliser et, si besoin, à les modifier. Vous pouvez également vous en inspirer pour tester les autres structures.
- Fichiers `*.h` : Fichiers d'entêtes pour tous les codes. L'entête `city_graph.h` contient les définitions des structures pour les nœuds et les arêtes (ainsi que des fonctions pour simplifier leurs utilisations). Il est nécessaire de les comprendre pour implémenter les algorithmes de Kruskal et Prim. Il est également utile de lire `union-find.h`, puis plus tard `min-heap.h` afin de comprendre quelles implémentations de ces structures sont attendues, ainsi que les fonctions qui doivent être spécifiées (ceci n'empêche pas d'en définir d'autres, c'est même encouragé pour simplifier l'écriture des fonctions attendues).

Pour compiler le code corrigé, on utilisera :

```
gcc -Wall -I. tp11.c corr/*.c common/*.c -lm
```

Et pour compiler votre code :

```
gcc -Wall -I. tp11.c eleves/*.c common/*.c -lm
```

Similairement, pour compiler les programmes de test fournis pour Union-find et pour le tri des arêtes :

```
gcc -Wall -I. test_union-find.c eleves/*.c common/*.c -lm
```

Et

```
gcc -Wall -I. test_edge-sorting.c eleves/*.c common/*.c -lm
```

Pour décortiquer la commande : $-I.$ précise que les fichiers `.h` sont dans le dossier courant, puis on spécifie les sources (l'étoile veut dire n'importe quelle suite de caractères), et `-lm` note qu'il faut charger la librairie mathématique.

On pourra lancer l'exécutable produit avec l'option `-h` pour lire la notice d'utilisation.

Pour tester ces algorithmes, un jeu de données vous est fourni. `villes-france-100k.csv`

2 Algorithme de Kruskal

Pour implémenter l'algorithme de Kruskal, il y a besoin d'une structure d'Union-find et d'un algorithme de tri sur les arêtes.

Question 1

Lire l'entête `union-find.h` et comprendre quelle représentation est proposée et quelles sont les primitives à implémenter. La structure sera de taille fixe, décidée à la création, et implémentée via un tableau tel que, pour tout élément `a` dans la structure `u,u[a]` est le parent de `a`. Notons que la structure proposée ne permet pas d'implémenter l'union par rang.

Implémenter la structure d'Union-find dans le fichier `union-find.c`. Pour simplifier, on implémentera la compression des chemins mais pas l'union par rang.

On rappelle que pour l'algorithme de Kruskal, le plus simple est de commencer par trier l'intégralité des arêtes par ordre de poids croissant. Comme on considère un graphe géométrique, il contient toutes les arêtes (non-orientées) possibles.

Question 2

Implémenter un algorithme de tri en place (c'est-à-dire sans allouer de mémoire supplémentaire mais en procédant par simples échanges) dans le fichier `edge-sorting.c`. On pourra proposer un tri quadratique simple puis revenir sur cette question une fois que l'on aura traité la partie sur les Tas-min pour implémenter un tri par tas. Les plus à l'aise pourront implémenter soigneusement un tri rapide.

Question 3

Si n est le nombre de sommets du graphe G géométrique considéré, combien G possède-t-il d'arêtes ? Quel est le nombre d'arêtes d'un arbre couvrant de G ?

De ce fait, les listes d'arêtes renvoyées par les fonctions Kruskal et Prim, correspondant aux arêtes d'un arbre couvrant de poids minimal, peuvent être implémentées par des tableaux de taille connue à l'appel. On prendra donc soin de renvoyer des tableaux alloués sur le tas avec la bonne taille.

Question 4

Implémenter l'algorithme de Kruskal dans `min-spanning-tree.c`.
Indications : commencer par générer toutes les arêtes dans un tableau de la bonne taille, puis trier ce tableau d'arêtes avec la fonction écrite précedemment (que l'on peut bien utiliser puisque son prototype est connu dans l'entête `min-spanning-tree.h`). On peut ensuite parcourir toutes les arêtes et ajouter les arêtes admissibles dans un tableau d'arêtes de la bonne taille que l'on renverra à la fin. N'oubliez pas d'arrêter le parcours des arêtes si on sait déjà que l'on a terminé.

Pour la première étape, on aura besoin d'une indexation explicite de l'ensemble des arêts c'est-à-dire des couples $\{(i, j) : i < j\}$.

3 Visualisation

Question 5

Aller sur http://umap.openstreetmap.fr/fr/map/new et importer le fichier de sortie `. geojson` produit par votre programme. Recommandation : dans les paramètres aller dans Propriétés de forme par défaut → Forme de l'icône choisir Cercle, pour avoir des points plus petits et un rendu plus lisible.

Comparer le graphe obtenu en France avec une carte du réseau de TGV : https://fr. wikipedia.org/wiki/Liste_des_lignes_à_grar /media/Fichier :Carte_TGV-fr.svg

4 Un algorithme plus approprié : Prim

L'algorithme de Kruskal n'est pas très approprié à ce contexte étant donné la quantité importe d'arêtes à considérer (et donc de mémoire à utiliser).

Nous allons introduire l'algorithme de Prim, mieux adapté car il utilisant autant de mémoire que de sommets. Cet algorithme, très similaire à celui de Dijkstra pour les plus courtes distances, requiert une structure de file de priorité. Cette dernière sera implémenté via un Tas-min.

4.1 Tas-min

Question 6

Rappeler la définition d'un Tas-min.

Comme dans l'algorithme de Prim, la file de priorité requise est de taille bornée connue à l'avance, nous allons utiliser une représentation efficace d'un Tas-min à capacité maximale fixée. Cette représentation se fera avec un tableau (indexé en partant de 0) dont la taille réelle est une puissance de deux. Nous aurons, pour le noeud d'indice i :

— Le fils gauche à l'indice $2i + 1$;
— Le fils droit à l'indice $2 * i + 2$;
— Le parent à l'indice $(i - 1)/2$;

Évidemment, ces valeurs d'indices sont à prendre dans la limite des indices valides, à savoir $[\![0, 2^n - 1]\!]$, où 2^n est la capacité de notre Tas-min.

Question 7

Implémenter, dans le fichier `min-heap.c` la fonction `mh_create` qui permet de créer un Tas-min de capacité au moins supérieure à celle donnée en paramètre. On prendra bien soin à ce que ce soit une puissance de deux. Attention, cette fonction renvoie une structure et pas un pointeur. On pensera à bien initialiser le tableau.

Question 8

Implémenter la fonction `mh_free` permettant de liberer l'espace alloué pour la structure (mais pas la structure elle-même !).

Question 9

Implémenter les fonctions `mh_empty`,`mh_size` et `mh_capacity` qui permettent d'inspecter la structure de données. Remarquez que la taille du tas est donnée par le nœud racine.

On rappelle qu'il existe deux opérations internes spéciales à un Tas-min : `percolate-up` et `percolate-down`. Ces opérations sont chargées de corriger la propriété de minimalité au sein du Tas-min pour un noeud qui la violerait (en le faisant monter dans l'arbre s'il a un poids trop petit, en le faisant descendre sinon), tout en maintenant la propriété de tas.

Question 10

Implémenter les deux fonctions `percolate_up` et `percolate_down`.

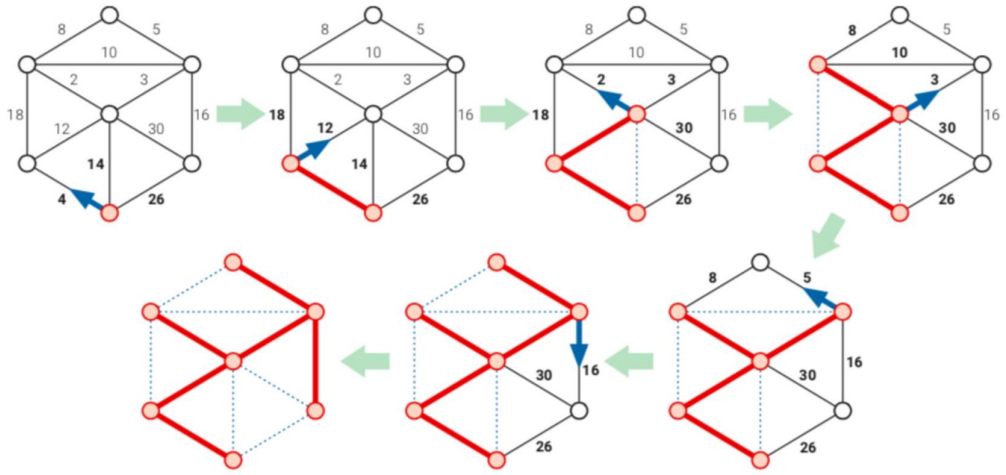
Avec ces deux fonctions, il devient aisé d'implémenter :
— l'insertion : on ajoute un nouveau nœud au seul endroit possible, puis on percole vers le haut ;
— l'extraction du minimum : on met le dernier nœud à la place de la racine que l'on renverra et on percole vers le bas ;
— la modification de poids : on percole vers le haut ou vers le bas en fonction du nouveau poids.

Question 11

Implémenter les fonctions `mh_insert, mh_pop` et `mh_modify_weight`.

4.2 Algorithm de Prim

L'algorithme de Kruskal maintient comme invariant une forêt couvrante (un sous-graphe acyclique composé de tous les sommets) et ajoute à chaque étape la plus petite arête qui permet de regrouper deux arbres de la forêt couvrante en cours de construction, jusqu'à avoir une forêt couvrante constituée d'un seul arbre. L'algorithme de Prim va construire progressivement un arbre (sous-graphe connexe acyclique) et ajoute à chaque étape la plus petite arête entre cet arbre en cours et un autre sommet qui n'est pas encore dans l'arbre, et ce jusqu'à avoir ajouté tous les sommets à l'arbre.
La figure ci-dessous illustre ce procédé.



Trace de l'algorithme de Prim en partant du sommet le plus bas. On ajoute la plus petite arête incidente à ce sommet pour former un arbre à deux nœuds, puis la plus petite arête entre cet arbre à deux nœuds et un autre sommet, et ainsi de suite.

Pendant le déroulé de l'algorithme on représente l'arbre en cours de construction, de manière orientée, par un tableau `pred` où un nœud de l'arbre pointe vers son unique parent. La racine étant le tout premier nœud choisi. On maintient également un tableau `cout` qui pour chaque sommet $s \in S$ qui n'est pas déjà dans l'arbre en cours de construction donne la valeur de la plus petite arête entre ce sommet et un sommet de l'arbre, s'il en existe une et $+\infty$ sinon. On maintiendra alors dans `pred` l'arête qui réalise ce minimum, pour les sommets adjacents à l'arbre en cours de construction. Une file de priorité permet de stocker exactement les sommets qui ne sont pas encore ajoutés à l'arbre. On s'arrête donc lorsque la file de priorité est vide. Le sommet de priorité minimale de la file, correspond donc bien au sommet incident à l'arbre en construction par une arête de poids minimal et on peut donc l'ajouter à l'arbre en construction. Cette arête étant celle indiquée dans `pred` il n'y a rien de plus à faire que de supprimer ce sommet de la file de priorité. En revanche, ajouter ce sommet à l'arbre en cours impose de mettre à jour, si nécessaire, la priorité de tous ses voisins.

Soit $G = (V, E)$ un graphe pondéré par $w : E \to \mathbb{R}$, c'est-à-dire que pour toute arête $e \in E$, son poids est $w(e)$, l'algorithme 1 décrit l'algorithme de Prim pour ce graphe. On commence par initialiser les structures de données puis on entre dans la boucle principale, tant que la file de priorité n'est pas vide. On extrait le sommet de priorité minimale et on met à jour tous ses voisins (qui ne sont pas déjà dans l'arbre).

```
Algorithme 1 : Algorithme de Prim.
    Données : Un graphe G=(V, E) pondéré par w et un sommet s
    Résultat : Un arbre couvrant de poids minimal avec s pour racine
    cout[s]:=0
    pred[s]:=s
    Soit F une file de priorité sur V
    Ajouter s à F avec priorité cout[s]=0
    pour v dans V-{s} faire
        cout[v]:=infini
        pred[v]:= une valeur différente de tous les autres sommets
        Ajouter v à F avec priorité cout[v]
    fin
    tant que F n'est pas vide faire
        t=F.defile /* Extraire l'élément de priorité minimale */
        pour u voisin de t tel que u dans F faire
            si cout[u]>w(t,u) alors
                cout[u]=w(t,u)
                pred[u]=t
                F.modifierPriorite (u,w(t,u))
            fin
        fin
    fin
    retourner pred
```

Remarquons ici que comme le graphe est géométrique, les voisins d'un sommet qui ne sont pas dans l'arbre en construction, sont exactement tous les sommets qui sont dans la file de priorité. On peut donc directement parcourir la file de priorité pour mettre à jour, si nécessaire, la priorité de tous les sommets restant.

Remarquons enfin que cet algorithme renvoie l'arbre couvrant sous la forme d'un tableau de prédécesseurs alors que le programme attend une liste d'arêtes. Ce n'est pas vraiment un problème, il suffira de faire une transformation à la toute fin.

Question 12

Quelle est la complexité spatiale occupée par l'algorithme de Kruskal (sachant qu'en entrée, la liste des arêtes est implicite) ? Quelle est la complexité spatiale occupée par l'algorithme de Prim. La comparer à celle de notre implémentation de l'algorithme de Kruskal.

Question 13

Implémenter l'algorithme de Prim dans le fichier `min-spanning-tree.c`. Indications : en C, il existe pour le type double la valeur spéciale `INFINITY`, définie dans le fichier d'entête standard `<math.h>`.