1 Mise en place

Créer un répertoire dédié à ce sujet de TP, télécharger les fichiers correspondant sur l'ENT. Vous devriez voir les fichiers suivants :

`graph_mat-2.h`	nouvelle version de l'interface de la bibliothèque graphe
`graph_mat-2.c`	fichier source correspondant
`acm.h`	déclaration des fonctions de calcul d'arbres couvrants minimaux
`acm.c`	fichier source correspondant
`test-acm.c`	fonction `main` de test
`Makefile`	pour compiler tout ce petit monde sans effort

Taper la commande `make`, pour vérifier qu'il n'y a pas d'erreur de compilation (bien que le programme produit ne soit pas intéressant).

La bibliothèque `graph_mat` a subit de très modestes modifications pour prendre en compte les coûts (de type `double`) des arêtes dans le graphe :

— un champ `value` a été ajouté à la structure `graph_mat`, mais vous ne devez pas y accéder directement ;

— si au moins une arête $\{v, w\}$ est présente dans le graphe, la fonction

```
double gm_val_edge(const graph_mat *g, int v, int w);
```

retourne le plus petit coût d'une arête entre v et w dans le graphe ;

— La fonction d'ajout d'une arête :

```
void gm_add_edge(const graph_mat *g, int v, int w, double val);
```

ajoute une arête de coût `val` entre v et w dans le graphe g. Si plusieurs arêtes entre v et w sont présentes, cette fonction se charge de n'enregistrer que le plus petit coût.

— La fonction `gm_init` initialise tous les coûts à la constante `DBL_MAX`, égale au plus grand `double` représentable.

Algorithme de Prim-Dijkstra

On rappelle l'algorithme de Prim-Dijkstra dans le cas où le graphe est supposé *connexe*.

Pour chaque sommet v dans l'ensemble $V = \{0, 1, ..., n-1\}$ des sommets du graphe, on maintient les données suivantes :

Une marque : un entier égal à 0 si le sommet a été traité, à 1 sinon ;

Un poids : le plus petit coût d'une arête menant d'un voisin marqué à ce sommet ;

Un parent : un voisin marqué dont est issue une arête menant à v de coût égal au poids de v.

Attention : le poids et le parent d'un sommet ne changent plus une fois qu'il a été marqué.

On se donne un sommet de départ.

Initialisation : le parent du sommet de départ est lui-même, les autres sont indéfinis (non initialisés ou initialisés à autre chose qu'un sommet du graphe). Les poids sont tous initialisés à $+\infty$ (pour l'implémentation : à `DBL_MAX`) sauf celui du sommet de départ qui vaut 0. Les sommets sont tous non marqués.

Répéter n **fois** : choisir un sommet non marqué de poids minimal, dit « pivot » ; marquer ce pivot, puis mettre à jour les poids et les parents de ses voisins non marqués.

Retourner l'arbre couvrant minimal dont les arêtes sont celles reliant les sommets à leurs parents (sauf, bien sûr, pour le point de départ).

Question 1: Implémenter dans le fichier `acm.c` la fonction

```
graph_mat *gm_acm_prim(const graph_mat *g, unsigned depart);
```

qui retourne l'adresse `acm` d'un arbre couvrant minimal du graphe d'adresse `g` obtenu par l'algorithme de Prim-Dijkstra avec pour sommet de départ `depart`. La valeur de retour de cette fonction est `NULL` en cas d'échec d'allocation mémoire.

On suppose dans cette question que le graphe est connexe et qu'aucun coût d'une arête présente n'est égal à `DBL_MAX`.

Pour tester votre fonction, il suffit, après compilation, de lancer le programme `./test-acm`. Un graphe et son arbre couvrant minimal (tracé en rouge) devraient apparaître à l'écran.

--- * ---

Question 2: (Bonus) Améliorer la fonction précédente, de manière à ce qu'elle puisse traiter le cas de graphes non connexes en retournant une *forêt couvrante minimale. Indication : lorsque l'on n'arrive pas à trouver de nouveau pivot, c'est que le graphe n'est pas connexe ; il faut alors choisir (arbitrairement) un nouveau sommet de départ.*

Pour tester la nouvelle fonction, on pourra modifier le fichier `test-acm.c` en enlevant des arêtes au graphes, ou bien en appelant la fonction `gm_random` avec des paramètres bien choisis (valeur de `p` petite).

--- * ---

Algorithme de Kruskal

On rappelle l'algorithme de Kruskal dans le cas où le graphe est supposé *connexe*.

Pour chaque sommet $v \in \{0, 1, ..., n-1\}$ du graphe, on maintient un représentant $R(v)$ de sa composante connexe dans la forêt en cours de construction. Le représentant de chaque composante connexe de la forêt doit être unique.

Les arêtes du graphe sont triées par coût croissant.

Initialisation : la forêt est au départ sans arête, donc pour chaque sommet v, $R(v) = v$. Le compteur i vaut 0.

Tant que la forêt couvrante n'a pas $n-1$ **arêtes**

Considérer l'arête numéro i dans la suite triée d'arêtes.

Si ses extrémités sont dans deux composantes connexes différentes de la forêt, ajouter l'arête i à la forêt couvrante et fusionner les composantes connexes de ses extrémités en mettant à jour les valeurs $R(v)$.

Sinon, ne rien faire.

Incrémenter i.

Pour vous aider à implémenter cet algorithme, nous avons déjà écrit :

Le type strucuré `struct edge`**,** contenant les champs `v` (extrémité de l'arête de plus petite étiquette), `w` (autre extrémité) et `value` (coût de l'arête). On accèdera directement à ces champs.

La fonction statique `cmp_edges` dont vous n'avez pas à vous soucier : elle n'est appelée que par la fonction de tri. [1]

1. Une fonction déclarée `static` n'est visible que dans le fichier dans lequel elle est implémentée. C'est une façon de rendre « privée » une fonction auxiliaire.

La fonction statique `struct edge *sort edges(const graphe *g)` que vous utiliserez dans votre implémentation de l'algorithme de Kruskal : cette fonction retourne un pointeur vers un tableau contenant les arêtes de `g`, triées par coût croissant. Les boucles et arêtes multiples de `g` sont ignorées. Le tableau est dynamiquement alloué, et doit être libéré après utilisation.

Question 3: 16 points. Implémenter dans le fichier `acm.c` la fonction

```
graph_mat *gm_acm_kruskal(const graph_mat *g);
```

qui retourne l'adresse d'un arbre couvrant minimal (représenté sous forme de graphe) du graphe d'adresse `g` obtenu par l'algorithme de Kruskal (`NULL` en cas d'échec d'allocation mémoire).

On suppose dans cette question que le graphe est connexe.

Pour tester votre fonction, il suffit, après compilation, de lancer le programme `./test-acm`. Un graphe et son arbre couvrant minimal (tracé en rouge) devrait apparaître à l'écran.

--- * ---

Question 4: (Bonus) Améliorer la fonction précédente, de manière à ce qu'elle puisse traiter le cas de graphes non connexes en retournant une *forêt couvrante minimale. Indication : il pourra être utile de modifier la fonction* `sort_edges` *de manière à ce qu'elle renseigne également le nombre d'arêtes du graphe lorsqu'on ignore les boucles et les arêtes multiples.*

Pour tester la nouvelle fonction, on pourra modifier le fichier `test-acm.c` en enlevant des arêtes au graphes, ou bien en appelant la fonction `graphe_aleatoire` avec des paramètres bien choisis (valeur de `p` petite).

--- * ---