TP 9 : Arbres couvrants de poids minimal

Exercice 1 : Mise en place

On travaille bien évidemment sur des graphes non orientés pondérés. Chaque arête aura un poids de type `double`.

On représente les arêtes à l'aide du type suivant :

```
typedef int sommet;

struct arete {
  sommet s;
  sommet t;
  double p;
};
typedef struct arete arete;
```

Un graphe sera représenté à l'aide du type suivant :

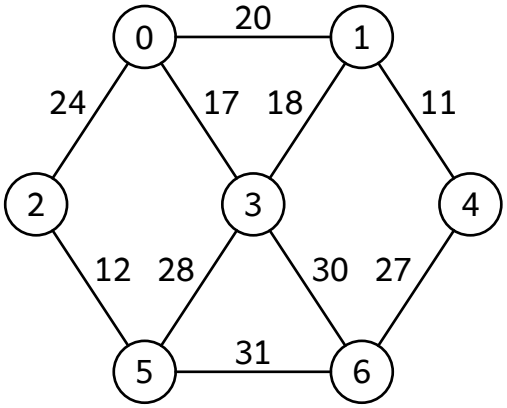
```
struct graph {
  int n;
  int* degres;
  arete** liste_adjacence;
}
```

- `n` indique le nombre de sommets du graphe (les sommets sont numérotés de 0 à $n-1$) ;
- `degres` est un tableau de longueur `n` où `degres[i]` contient le degré du sommet i ;
- `liste_adjacence` est un tableau de longueur avec `liste_adjacence[i]` un tableau de longueur `degres[i]` qui contient toutes les arêtes incidentes au sommet i ;
- on aura systématiquement `liste_adjacence[i][j].s == i` (le champ `s` de la structure `arete` peut donc sembler redondant, mais il sera utile par la suite).

On définit le format suivant pour sérialiser un graphe dans un fichier texte :

- la première ligne contient un entier (strictement positif), le nombre n de sommets du graphe ;
- les n lignes suivantes contiennent chacune un entier `d` suivis de `d` couples (j, p) séparés par des espaces, `d` correspond au degré du sommet, les `j` à ses voisins et `p` aux poids des arêtes.

En exemple, voici un graphe et sa représentation sous forme d'un texte :




```
7
3 (1, 20) (2, 24) (3, 17)
3 (1, 20) (2, 18) (4, 11)
3 (0, 24) (3, 21) (5, 12)
6 (0, 17) (1, 18) (2, 21) (4, 23) (5, 28) (6, 30)
3 (1, 11) (3, 23) (6, 27)
3 (2, 12) (3, 28) (6, 31)
3 (3, 30) (4, 27) (5, 31)
```

1) Écrire une fonction `graphe* lire_graphe(char* nom_fichier)` qui prend en paramètre le nom d'un fichier contenant la représentation textuelle d'un graphe et qui renvoie ce graphe.

2) Écrire une fonction `void ecrire_graphe(graphe* g, char* nom_fichier)` qui prend en paramètre un graphe et un nom de fichier et qui crée un fichier portant ce nom et contenant la représentation textuelle du graphe.

3) Écrire une fonction `void liberer_graphe(graphe* g)` qui prend en paramètre un graphe et qui libére la mémoire qui a été allouée pour le graphe.

4) Écrire une fonction `int taille_graphe(graphe* g)` qui prend en paramètre un graphe et qui renvoie la taille de ce graphe.

On rappelle que la taille d'un graphe est son nombre d'arêtes. Si vous aviez besoin de ce rappel, arrêtez tout de suite ce TP pour créer une carte Anki.

5) Écrire une fonction `arete* obtenir_aretes(graphe* g, int* nb_aretes)` qui prend en paramètre un graphe et qui renvoie un tableau contenant toutes les arêtes du graphe.

La fonction à pour effet de bord de modifier la valeur de `*nb_aretes` pour qu'elle devienne la taille de `g`.

6) Écrire une fonction `void afficher_tableau_aretes(arete* aretes, int nb_aretes)` qui affiche le contenu d'un tableau d'arêtes.

7) Écrire une fonction `double sommes_poids(arete* aretes, int nb_aretes)` qui prend en paramètre un tableau d'arêtes et qui calcule la somme des poids des arêtes du tableau.

8) Écrire une fonction `void trier_aretes(arete* aretes, int nb_aretes)` qui trie un tableau de `nb_aretes` arêtes par poids croissant à l'aide d'un tri fusion.

9) Implémenter la structure de données unir-trouver (sans oublier une fonction permettant de libérer la mémoire).

Exercice 2 : Algorithme de Kruskal

10) Appliqué à un graphe connexe d'ordre n, combien d'arêtes l'algorithme de Kruskal va-t-il choisir ?

11) Si on applique l'algorithme de Kruskal à un graphe non connexe, il renvoie une forêt couvrante de poids minimale et non pas un arbre.

Exprimer le nombre d'arêtes choisies en fonction de l'ordre n du graphe et de son nombre de composantes connexes c.

12) Écrire une fonction `arete* kruskal(graphe* g, int* nb_choisis)` qui prend en paramètre un graphe et qui renvoie un tableau d'arêtes qui constitue une forêt couvrante de poids minimale de `g`.

La fonction a pour effet de bord de changer la valeur de `*nb_choisis` pour qu'elle contienne le nombre d'arêtes de la forêt couvrante.

Exercice 3 : Algorithme de Boruvka (bonus)

13) Implémenter l'algorithme de Boruvka.

Vous trouverez une description de cet algorithme dans le livre « Algorithms » de Jeff Erickson (http://jeffe.cs.illinois.edu/teaching/algorithms/#book).