TRAVAUX PRATIQUES XI

Arbres couvrants et algorithme de Kruskal

L'objectif de ce TP est d'implémenter l'algorithme de Kruskal pour la recherche d'un arbre couvrant de poids minimal dans un graphe pondéré non orienté.

A Représentation des graphes

Dans cette section et la suivante, on s'intéressera au fichier fourni `kruskal.c` (et son header `kruskal.h`) uniquement.

A.1 Structure de données

Dans ce sujet, nous allons travailler sur des graphes non orientés et pondérés : chaque arête aura donc un poids, qui sera un type numérique (`float`, `int`, `double`...).

Pour représenter une arête, on utilise la structure suivante :

C
```c
typedef float weight_t;
typedef int vertex;

struct edge {
    vertex x;
    vertex y;
    weight_t rho;
};

typedef struct edge edge;
```

Un graphe sera représenté par la structure suivante :

C
```c
struct graph {
    int n;
    int* degrees;
    edge** adj;
};

typedef struct graph graph_t;
```

- `n` indique le nombre de sommets du graphe (ces sommets sont numérotés de 0 à $n-1$).
- `degrees` est un tableau de n entiers naturels, où `degrees[i]` indique le degré du sommet i.
- `adj` est un tableau de longueur n, avec `adj[i]` un tableau de longueur `degrees[i]` qui contient toutes les arêtes incidentes au sommet i.
- On aura systématiquement `adj[i][j].x == i` : le champ `x` de la structure `edge` peut donc sembler redondant, mais il sera utile par la suite.

A.2 Sérialisation dans un fichier

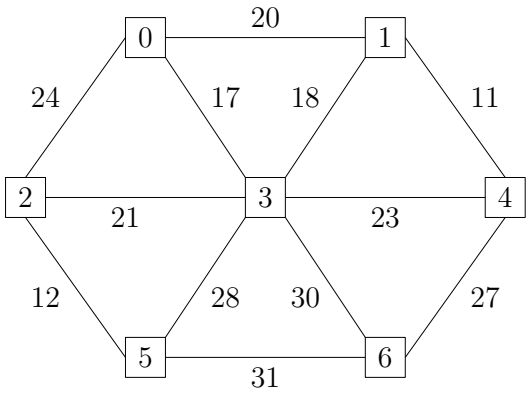
On définit le format suivant pour sérialiser un graphe dans un fichier (c'est à dire le représenter "à plat", par une simple suite de caractères pour pouvoir le stocker dans un fichier) :

- 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). Le premier entier indique le degré du sommet, les couples correspondent aux arêtes incidentes (avec le numéro du voisin en première composante et le poids en deuxième composante).

Des fonctions pour lire un graphe depuis un fichier, écrire un graphe dans un fichier et libérer la mémoire associée à un graphe sont fournies dans le fichier `kruskal.c` .

Remarque :

— La fonction `graph_free` suppose que la structure, le tableau `degrees`, le tableau `adj` et chacun des tableaux `adj[i]` ont été alloués par un appel à `malloc`. C'est bien le cas pour un graphe issu de `read_graph`.

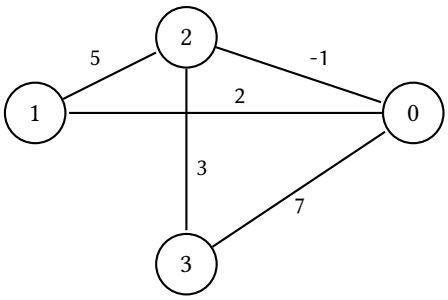



```
7
3 (1, 20) (2, 24) (3, 17)
3 (0, 20) (3, 18) (4, 11)
3 (0, 24) (3, 21) (5, 12)
6 (0, 17) (1, 18) (2, 21) (4, 23) (5, 28) (6, 30)
3 (1, 11) (3, 23) (6, 27)
3 (2, 12) (3, 28) (6, 31)
3 (3, 30) (4, 27) (5, 31)
```

FIGURE XI.1 – Le graphe pondéré G_0 et sa représentation textuelle.

Le graphe G_0 est donné dans le fichier texte `exemple.txt` .

1. Donner la représentation textuelle du graphe G_1 suivant et la stocker dans un fichier `g1.txt` :



2. Dans la fonction `main.c` du fichier `kruskal.c` , utilisez la fonction `read_graph` pour obtenir le graphe G_0.

Rappel : on ouvre un fichier avec la fonction `fopen` et on n'oubliera pas de le fermer après utilisation, avec la fonction `fclose` [1].

B Fonctions élémentaires sur les graphes

Commençons maintenant à écrire quelques fonctions utiles pour l'algorithme de Kruskal. Rappelons cet algorithme :

Algorithme 3 Algorithme de Kruskal

Entrées : Un graphe pondéré $G = (S, A, \rho)$, et une liste $A = (a_1, \dots, a_p)$ de ses arêtes.
Sorties : Un arbre couvrant minimal de G

```
. T ← (S, ∅)
pour chaque xy ∈ A, dans l'ordre croissant de poids faire
    si x et y sont dans deux composantes connexes distinctes de T alors
        T ← T + xy
renvoyer T
```

1. Si vous en avez besoin, vous pouvez retourner voir le TP-cours 6 pour la manipulation des fichiers en C : « Autour des expressions régulières »

On souhaite donc obtenir toutes les arêtes du graphe, dans l'ordre croissant de poids, pour pouvoir les parcourir aisément.

Les fonctions à implémenter à ce stade pour le fichier `kruskal.c` sont écrites dans le header `kruskal.h`. Lisez le header pour avoir leurs prototypes.

3. Écrire la fonction `number_of_edges`. Elle prend en entrée un pointeur vers un graphe et renvoie son nombre total d'arêtes. Attention à ne compter chaque arête qu'une seule fois, le graphe est non orienté.
4. Écrire la fonction `get_edges`, qui construit un tableau de toutes les arêtes du graphe. Elle prend en entrée un pointeur vers un graphe et un pointeur `nb_edges` vers un entier, et renvoie un pointeur vers un bloc alloué contenant toutes les arêtes du graphe.
   La fonction aura comme effet secondaire de fixer la valeur pointée par `nb_edges` au nombre total d'arêtes du graphe (et donc à la longueur du bloc renvoyé).

Pour une version non-étoilée du TP, la fonction suivante (et ses auxiliaires) sont fournies dans le fichier `sort-edges.c`. Vous pouvez la copier-coller dans votre fichier `kruskal.c` et passer directement à la question 6. Pour une version étoilée du TP, nous allons maintenant apprendre à utiliser la fonction `qsort` en C et à manipuler du `void*`.

Si vous pensez que vous n'aurez pas le temps de finir d'implémenter l'algorithme de Kruskal, je vous demande de faire la version non-étoilée du TP et de passer directement à la suite pour ne pas perdre du temps sur cette notion, quitte à y revenir par la suite.

5. Écrire la fonction `sort_edges` qui prend en entrée un tableau d'arêtes et le trie par poids croissant.

   On pourra au choix utiliser la fonction `qsort` expliquée ci-dessous, ou ré-écrire soi-même un tri rapide (ce qui n'est jamais inutile pour s'entraîner…).

Fonction qsort

Pour la réutilisation du code, il est très souvent souhaitable d'avoir du code *générique* : par exemple, une structure de pile capable de contenir des éléments de type quelconque (mais uniforme), ou une fonction de tri capable de trier un tableau d'objets de type quelconque étant donnée une fonction de comparaison. Il n'y a pas de moyen élégant et pratique de faire cela en C (l'absence de tel mécanisme est clairement une des raisons pour la création du C++), mais c'est cependant possible :

— soit en abandonnant partiellement le système de type et en utilisant des `void*` ;

— soit en utilisant des *macros*.

Un exemple de fonction utilisant des `void*` est la fonction `malloc`, car son type de retour. C'est le seul cas où ce type `void*` est au programme. Le transtypage vers un « vrai » type se fait alors immédiatement, soit de manière explicite, soit de manière implicite (équivalentes) :

```c
int* t = (int*)malloc(n * sizeof(int));
//ou
int* t = malloc(n * sizeof(int));
```

Voyons maintenant la fonction `qsort`, qui permet de trier un tableau. Cette fonction a pour prototype :

```c
void qsort(void* ptr, size_t count, size_t size, int comp(const void*, const void*));
```

- `ptr` est un pointeur vers le début de la zone à trier (autrement dit, c'est un « tableau »), et son type est `void*`. Ce type est celui d'un pointeur *générique* : on ne sait pas quel est le type des objets que contient la zone à trier.
- `count` indique le nombre d'éléments qu'il faut trier. Son type est `size_t`, qui est simplement un type entier non signé suffisamment grand pour contenir la taille de n'importe quel tableau (en pratique, c'est un alias pour `uint64_t` sur toutes les machines modernes).
- `size` indique la taille (en octets) d'*un* objet à trier. La taille totale (en octets) de la zone à trier est donc `count*size`. Dans les fonctions que nous écrivons usuellement et qui prennent en entrée un tableau, ce paramètre est absent car inutile : comme le type du tableau est connu (disons `double*`), la taille d'un élément l'est également (`sizeof(double)`).
- Le dernier paramètre, `comp` est une fonction (un *function pointer*, techniquement, mais cela n'a pas d'importance), qui sera utilisée pour comparer entre eux les éléments du tableau à trier. La fonction passée en paramètre doit avoir le prototype suivant :

```c
int comp(const void* x, const void* y);
```

- Le mot clé `const` indique qu'on ne modifiera pas les valeurs pointées par `x` et `y`.
- `x` et `y` sont donc fondamentalement des `void*`, c'est-à-dire qu'ils pointent chacun vers un objet de type quelconque.
- La fonction doit renvoyer un entier strictement négatif si $x \prec y$, nul si $x \approx y$, strictement positif si $y \prec x$ (pour l'ordre utilisé pour le tri). C'est la même convention que pour la fonction de comparaison à passer à `List.sort` ou `Array.sort` en OCaml.

Exemple :

Pour trier un tableau d'entiers par ordre croissant [2], on pourrait donc procéder ainsi :

```c
//requiert #include <stdlib>
int compare_ints(const void* px, const void* py){
    // On transtype px en int*, puis l'on déréférence le pointeur.
    int x = *(int*)px;
    int y = *(int*)py;
    return x - y;
}

int main(void){
    int t[4] = {12, 1, 7, -3};
    qsort(t, 4, sizeof(int), compare_ints);
    ...
}
```

À vous d'adapter ça à notre tableau d'arêtes ! Notez que vous n'aurez jamais à écrire de fonctions manipulant des `void*` , ou alors de manière extrêmement guidée (par exemple dans le cas d'une fonction de comparaison pour `qsort`).

Retour aux fonctions élémentaires

6. Écrire la fonction `print_edge_array` qui affiche le contenu d'un tableau d'arêtes. Utiliser cette fonction pour vérifier sur le graphe fourni en exemple le bon fonctionnement des fonctions `get_edges` et `sort_edges`.

C Structure Union-Find

Dans cette section, on s'intéressera au fichier fourni `union_find.c` (et son header `union_find.h`) uniquement.

Rappelons que pour coder efficacement l'algorithme de Kruskal, et notamment les tests successifs d'appartenance à la même composante connexe, on utilise une structure *union-find*. On va donc en implémenter une, cette fois en C.

On utilise la structure suivante pour représenter une partition de $[0 \dots n-1]$:

```c
struct partition {
    int nb_elements;
    int nb_sets;
    int* arr;
};
```

- `nb_elements` est l'entier n (autrement dit, l'ensemble partitionné est $[0, \dots,$ `nb_elements - 1`$]$).
- `nb_sets` est le nombre d'ensembles dans la partition.
- `arr` est un tableau de longueur `nb_elements`.
  - Si i n'est pas la racine de son arbre, alors `arr[i]` est le père de `i`.
  - Si i est le représentant d'un ensemble X (et donc une racine), alors plutôt que de stocker simplement `i`, on en profite pour **stocker les informations nécessaires à la fusion** :
    `arr[i]`$= -|X|$ (où $|X|$ est le cardinal de X).

2. en ignorant les problèmes de dépassement de capacité

Par rapport à ce que nous avons vu en cours, nous allons choisir une stratégie légèrement différente [3] pour la fusion : au lieu d'effectuer une *union par rang*, on effectuera une **union par taille**. Autrement dit, lorsque l'on fusionne deux arbres, le plus gros de deux (en terme de nombre de nœuds) deviendra le père de l'autre.

7. (Bonus) Montrer que si l'on part d'une forêt de n arbres de taille 1 et que l'on effectue une suite quelconque d'unions par taille, chaque arbre t vérifiera bien toujours $|t| \geq 2^{h(t)}$ (où $|t|$ est la taille de l'arbre et $h(t)$ sa hauteur, qui vaut zéro dans le cas d'un arbre réduit à sa racine).

Remarque :

— Cette propriété a été prouvée dans le cours pour l'union par hauteur : on voit donc qu'elle reste valable pour l'union par taille. Il en va de même des complexités amorties mentionnées ultérieurement, y compris le $\log^*$ quand on ajoute la compression de chemin.

Implémentez maintenant les opérations de la structure *union-find* dans le fichier `union_find.c` . Les prototypes de ces opérations se trouvent dans le header associé. Vous pouvez ajouter une fonction `main` dans le fichier pour les tests, mais il faudra penser à la retirer à la fin de la partie, car ce fichier sera utilisé comme une bibliothèque pour l'algorithme de Kruskal.

8. Écrire la fonction `partition_new` qui crée une partition de $[0, \dots, n-1]$ en singletons.

9. Écrire la fonction `partition_free` qui libère la mémoire associée à une partition.

10. Écrire la fonction `find` qui renvoie le représentant de l'entier x passé en argument. Cette fonction procédera à une compression de chemin.

11. Écrire la fonction `merge` qui fusionne les ensembles auxquels appartiennent les deux entiers passés en argument. Cette fonction effectuera une union par taille.

En cas d'égalité, `merge(p, x, y)` fera de `y` le nouveau représentant de la classe de `x`.

D Algorithme de Kruskal

Dans cette section, on s'intéressera de nouveau au fichier `kruskal.c` . Ce fichier devra désormais faire appel à la bibliothèque `union_find.h` codée précédemment (**à inclure**).

Quelques considérations théoriques utiles pour la fonction `kruskal` :

12. Appliqué à un graphe connexe contenant n sommets, combien d'arêtes l'algorithme de Kruskal va-t-il choisir ?

13. Quand l'algorithme de Kruskal est utilisé sur un graphe non connexe, il renvoie (un ensemble d'arêtes qui définit) une **forêt couvrante minimale** : un arbre couvrant minimal de chaque composante connexe. Quel est le nombre d'arêtes choisies dans ce cas, en fonction du nombre n de sommets, du nombre p d'arêtes et du nombre c de composantes connexes (toutes ces quantités ne sont pas nécessairement utiles) ?

Nous avons maintenant tout ce qu'il nous faut pour implémenter l'algorithme de Kruskal ! Rappelons son pseudo-code un peu plus détaillé, prenant en compte la structure *union-find* :

Algorithme 3 Algorithme de Kruskal avec structure UnionFind

Entrées : Un graphe pondéré $G = (S, A, \rho)$, et une liste $A = (a_1, \dots, a_p)$ de ses arêtes.
Sorties : Un arbre couvrant minimal de G.

$T \leftarrow \{\}$
$X \leftarrow$ CreatePartition(n)
pour chaque $xy \in A$, *dans l'ordre croissant de poids* **faire**
  si Find$(X, x) \neq$ Find(X, y) **alors**
    Union(X, x, y)
    $T \leftarrow T \cup \{xy\}$
renvoyer T //on confond un arbre couvrant et ses arêtes

14. Écrire une fonction `kruskal` ayant la spécification suivante :

Prototype

```c
edge* kruskal(graph_t* g, int* nb_chosen);
```

3. Pourquoi ? Parce que vous avez déjà fait l'union par rang deux fois, et que les sujets de concours pourront vous demander de vous adapter à d'autres méthodes proches du cours !

Entrées `g` est un pointeur vers un graphe pondéré non orienté, `nb_chosen` est un argument de sortie.

Sortie Un pointeur vers un bloc alloué d'arêtes qui constituent une forêt couvrante minimale de `g`.

Effet secondaire Après l'appel, la valeur pointée par `nb_chosen` doit être égale au nombre d'arêtes choisies.

Remarques :

- On allouera un bloc suffisamment grand, que l'on n'utilisera que partiellement si jamais le graphe s'avère être non connexe.
- On s'arrêtera dès que possible (on ne continuera pas à examiner les arêtes si l'on sait que plus aucune ne peut être choisie).

Maintenant, testons et manipulons !

15. Écrire un programme qui lit prend en entrée un nom de fichier décrivant un graphe en ligne de commande, et affiche les arêtes choisies par l'algorithme de Kruskal, en précisant si ces arêtes définissent un arbre couvrant ou une forêt couvrante (non réduite à un arbre), ainsi que le poids total de ces arêtes.

16. Vérifier que vous obtenez bien cet arbre couvrant pour l'exemple fourni :

Terminal

```
Total weight 106.000
(1, 4, 11.00)
(2, 5, 12.00)
(0, 3, 17.00)
(1, 3, 18.00)
(2, 3, 21.00)
(4, 6, 27.00)
```

E Pour aller plus loin : Algorithme de Boruvka

L'algorithme de Boruvka est un autre algorithme permettant de calculer un arbre couvrant minimal (d'un graphe pondéré, non orienté et connexe $G = (V, E, \rho)$). Son principe est le suivant :

Algorithme 2 : Algorithme de Boruvka

Entrées : Un graphe pondéré $G = (S, A, \rho)$ **connexe**, et une liste $A = (a_1, \dots, a_p)$ de ses arêtes.
Sorties : Un arbre couvrant minimal de G.

$T \leftarrow (V, \emptyset)$
tant que T *n'est pas connexe* **faire**
  Calculer $C_1, \dots, C_k$ les composantes connexes de T
  pour chaque $i = 1$ *à* k **faire**
    Trouver $e \in E$ de poids minimal ayant une extrémité dans C_i et une dans $V \backslash C_i$
    $T \leftarrow T + e$
renvoyer T

E.1 Étude de l'algorithme de Boruvka (bonus théorique)

On rappelle que G est supposé connexe, et l'on suppose de plus pour simplifier (légèrement) que les poids des arêtes sont deux à deux distincts.

17. Justifier que l'étape « trouver e de poids minimal telle que… » ne peut pas échouer.

18. Montrer que l'algorithme termine.

19. Montrer que l'algorithme renvoie un arbre couvrant.

20. Montrer que l'algorithme renvoie un arbre couvrant minimal.

21. On suppose que G est donné par un tableau de listes d'adjacence, on note n son nombre de sommets et p son nombre d'arêtes.

a. Quelle est la complexité du calcul des composantes connexes de T?

b. Montrer que le nombre de passages dans la boucle principale est en $\mathcal{O}(\log n)$.

c. En supposant qu'il est possible d'exécuter la boucle interne en temps $\mathcal{O}(p)$, calculer la complexité de l'algorithme de Boruvka.

E.2 Implémentation de l'algorithme de Boruvka

22. Écrire une fonction `get_components` ayant la spécification suivante :

Prototype

```c
int* get_components(graph_t* g, int* nb_components);
```

Entrées Un pointeur `g` vers un graphe non orienté (et pondéré) $G = (V, E)$; `nb_components` est un argument de sortie.

Effets secondaires Après l'appel, la valeur pointée par `nb_components` vaudra k, le nombre de composantes connexes de G.

Sortie Un pointeur vers un bloc alloué `arr` de taille $|V|$, indiquant pour chaque sommet le numéro de sa composante connexe. Plus précisément, on exige que :

- les valeurs contenues dans `arr` soient dans l'intervalle $[0 \dots k-1]$;
- `arr[i]` soit égal à `arr[j]` si et seulement si i et j sont dans la même composante connexe.

Remarques :

— On exige une complexité en $\mathcal{O}(n+p)$.

— Suivant la méthode choisie, il peut être nécessaire de créer une fonction auxiliaire.

Pour implémenter l'algorithme de Boruvka, il va falloir ajouter des arêtes à un graphe. Vu les structures de données choisies, la manière la plus simple de faire cela est de pré-allouer des listes d'adjacence suffisamment grandes et de faire varier le degré des nœuds. On sait que les arêtes choisies formeront un sous-ensemble des arêtes du graphe initial, on peut donc allouer des listes d'adjacence de même taille que celles du graphe initial.

23. Écrire une fonction `without_edges` qui renvoie un pointeur `t` vers un `graph_t` ayant le même nombre de sommets que le graphe `g` passé en argument, des blocs `t->adj[i]` de même taille que les `g->adj[i]` mais tous les degrés égaux à zéro (et donc aucune arête).

```c
graph_t* without_edges(graph_t* g);
```

24. Écrire une fonction `get_minimal_edges` ayant le prototype suivant :

```c
edge* get_minimal_edges(graph_t* g, int* components, int nb_components);
```

Le tableau `components` et l'entier `nb_components` indiquent les composantes connexes du graphe T en cours de construction (qui n'est pas passé en argument), avec les mêmes conventions que `get_components`. Cette fonction renvoie un tableau `edges` de longueur `nb_components` tel que `edges[c]` soit l'arête `e` de G la plus légère telle que `e.x` soit dans la composante numéro c de T et `e.y` n'y soit pas.

Remarque :

— On supposera que G est connexe et que T ne l'est pas, ce qui garantit qu'une telle arête existe pour chaque composante.

25. Écrire une fonction `add_edges` ayant la spécification suivante :

Prototype

```c
bool add_edges(graph_t* g, graph_t* t);
```

Entrées Le graphe G pour lequel on cherche à construire un arbre couvrant minimal et le graphe T correspondant à l'arbre en cours de construction, sous forme de pointeurs vers des `graph_t`.

Sortie `true` si T est connexe, `false` sinon.

Effets secondaires Si T n'était pas connexe, alors pour chaque composante connexe C de T, l'arête de poids minimal reliant C à $V \setminus C$ a été ajoutée à T. Attention, une arête donnée ne doit être ajoutée qu'une seule fois (et le graphe est non orienté).

On exige une complexité en $\mathcal{O}(n+p)$, où n et p sont le nombre de sommets et d'arêtes de G.

26. Écrire une fonction `boruvka` calculant un arbre couvrant minimal du graphe passé en argument en utilisant l'algorithme de Boruvka.

```c
graph_t* boruvka(graph_t* g);
```