

# Algorithme de Kruskal

## Rappels :

- Les TPs de tout le semestre seront réalisés en C++. Le but n'est pas d'utiliser toute la puissance du C++, mais simplement quelques outils qui nous faciliteront la tâche, en particulier la fonction `cout` à la place de `printf`, et la structure de données `vector` pour faire des tableaux dynamiques.
- Les fichiers à télécharger sont disponibles à <http://pagesperso.g-scop.fr/~pastor1/> (ou bien, tapez *Lucas Pastor* dans Google, trouvez sa page web, et descendez jusqu'à *Teaching*), et également sur Moodle → INF303.
- Les questions marquées d'une ou plusieurs étoiles sont plus compliquées. **Elles ne sont à traiter que lorsque le reste est fait.**
- **Toutes vos fonctions doivent être testées au fur et à mesure.**

Le but du TP est d'implémenter l'algorithme de Kruskal, qui permet de calculer un arbre couvrant de poids minimum dans un graphe pondéré. On rappelle cet algorithme :

---

### Algorithm 1 Kruskal( $G$ )

---

**Entrée :** Un graphe  $G$  pondéré

**Sortie :** Un arbre  $T$  couvrant de poids minimum de  $G$

$T \leftarrow \emptyset$

Trier les arêtes de  $G$  par ordre croissant de poids

**Tant que**  $T$  contient strictement moins de  $n - 1$  arêtes **faire**

$e \leftarrow$  prochaine arête pas encore vue dans cet ordre

**Si**  $e$  ne génère pas de cycle dans  $T$  **alors**

ajouter  $e$  à  $T$

**Fin si**

**Fin tant que**

**Renvoyer**  $T$

---

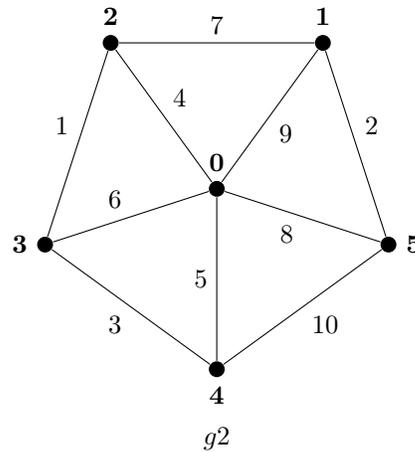
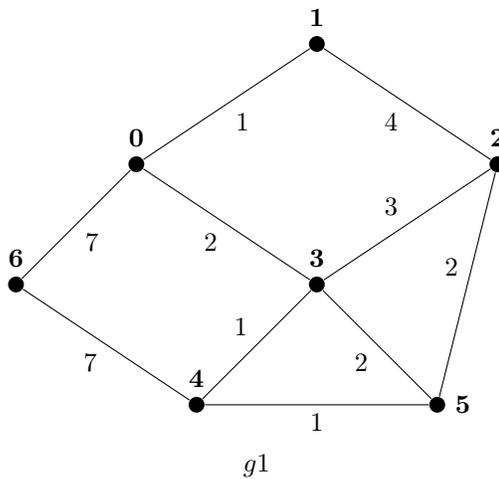
Dans ce TP, un graphe pondéré sera représenté par une matrice d'adjacence  $g$  comme (vue dans les TPs précédents) mais prenant en compte le poids entre deux sommets adjacents. Pour toute paire  $uv$  de sommet, on a  $g[u][v] = 0$  si  $uv$  ne sont pas adjacents, et  $g[u][v] = g[v][u] = p_{u,v}$  avec  $p_{u,v}$  le poids de l'arête  $uv$ .

## 1 Manipulation des graphes

*Question 1.* Télécharger les fichiers `kruskal.cpp`, `graphe.h` et `graphe.cpp`. Compiler à l'aide de la commande `g++ graphe.cpp kruskal.cpp -o kruskal` puis tester avec `./kruskal`.

*Question 2.* Écrire une fonction `void ajoute_arete(graphe &g, int u, int v, int poids)` qui modifie la matrice d'adjacence `g` en ajoutant l'arête  $uv$  dont le poids est donné en paramètre. *NB : on notera l'importance du `&` qui permet de **modifier** l'argument `g`. Pour vous en convaincre, enlever le `&` et testez votre fonction : vous verrez que `g` n'est jamais modifié.*

*Question 3.* Décommenter les deux fonctions `graphe init_g1()` et `graphe init_g2()` qui renvoient les matrices d'adjacences des deux graphes suivants. Vous décommenterez également dans votre `main` la déclaration des des deux variables `g1` et `g2` de type `graphe` initialisés grâce à ces fonctions, ainsi que leur affichage avec la fonction `affiche` (définie dans `graphe.cpp`). Testez le programme.



*Question 4.* Écrire une fonction `bool est_connexe(graphe &g)` qui renvoie vraie si  $g$  est connexe, sinon faux. Pour cela, implémenter un parcours en largeur ou profondeur du graphe et vérifier que tous les sommets sont visités. *Attention : pour tester si  $u$  et  $v$  sont voisins, il faut tester si  $g[u][v] \neq 0$  et non pas  $g[u][v] == 1$ , car le poids des arêtes n'est pas forcément 1!*

*Question 5.* Écrire une fonction `bool sont_connectes(graphe &g, int u, int v)` qui indique si il existe un chemin entre  $u$  et  $v$  dans le graphe  $g$ . Quelle est la complexité de cette fonction ?

## 2 Manipulation des arêtes et des vector d'arêtes

*Question 6.* Afin de faciliter la manipulation des arêtes (et en particulier le tri), on souhaite créer une structure de données représentant une arête pondérée. Pour cela, on utilise la structure suivante (voir `Partie 2` dans `kruskal.cpp`) :

```

1 struct arete{
2     /* 3 champs: u et v sont les extremités de l'arete, l'autre est le poids */
3     int u;
4     int v;
5     int poids;
6
7     /* Constructeur numero 1: initialise tous les champs à zero */
8     arete() {u=0; v=0; poids=0;}
9     /* Constructeur numero 2: initialise les champs selon les paramètres */
10    arete(int uu, int vv, int p) {u=uu; v=vv; poids=p;}
11 };

```

On accède alors à un élément d'une instance  $e$  d'une arête via le caractère `point`. Voici deux exemples d'utilisation de la nouvelle structure `arete`, donnés dans le `main` : décommentez-les et testez-les.

```

1     /* Partie 2 */
2     arete e=arete();
3     e.u=0;
4     e.v=1;
5     e.poids=100;
6     cout << "L'arete " << e.u << e.v << " a un poids de " << e.poids << endl;
7
8     arete e2=arete(2,3,10);
9     cout << "L'arete " << e2.u <<"-"<< e2.v << " a un poids de " << e2.poids << endl;

```

*Question 7.* Écrire une fonction `void affiche_arete(arete &a)` qui affiche l'arête  $uv$  de poids  $p$  sous la forme " $(u, v) : p$ " puis qui saute une ligne.

*Question 8.* Écrire une fonction `void affiche_aretés(vector<arete> &t)` qui affiche toutes les arêtes contenues dans le tableau  $t$ .

*Question 9.* Écrire une fonction `vector<arete> tableau_aretés(graphe& g)` qui renvoie un tableau contenant toutes les arêtes de  $g$ . Testez-la dans votre `main` avec `g1` et `g2`.

### 3 Algorithme de Kruskal

*Question 10.* Exécuter à la main l'algorithme de Kruskal sur  $g_1$  et  $g_2$ . Indiquer le poids de l'arbre couvrant trouvé ainsi que leurs arêtes, dans l'ordre dans lequel elles sont ajoutées par l'algorithme. *Se convaincre qu'ajouter l'arête  $e = uv$  à  $T$  génère un cycle si et seulement si il existe un chemin de  $u$  à  $v$  dans  $T$ .*

*Question 11.* Écrire une fonction `void tri(vector<arete> &t)` qui trie un tableau  $t$  d'arêtes par ordre croissant de poids. Pour ce faire, utiliser votre algorithme de tri préféré. Si vous n'avez pas encore de tri préféré, essayer le tri par sélection : pour chaque case  $i$  du tableau, appelez une fonction auxiliaire qui sélectionne l'indice dont l'arête est de plus petit poids, parmi les indices  $i$  à  $t.size()-1$  ; ensuite, faites un échange pour mettre cette arête de plus petit poids en  $i$ . *Bonus : Quelle est la complexité de votre algorithme ?*

*Question 12.* Implémenter l'algorithme de Kruskal dans une fonction `int kruskal(graphe &g)`. Cette fonction devra afficher les arêtes choisies et renvoyer le poids d'un arbre couvrant de poids minimum de  $g$ . *Attention : cette fonction ne doit pas modifier  $g$ . Il est donc fortement conseillé d'utiliser une variable `sous_graphe_intermediaire` de type `graphe` correspondant au  $T$  dans le pseudo-code.*

*Question 13.* Quelle est la complexité de votre algorithme ?

### 4 Algorithme de Kruskal inversé

On s'intéresse maintenant à une version alternative de Kruskal. Le but est toujours de trouver un arbre couvrant de poids minimum, mais cette fois-ci en triant les arêtes par ordre décroissant et tentant de les retirer si possible. Voici l'algorithme de Kruskal inversé.

---

**Algorithm 2** `Kruskal2( $G$ )`

---

**Entrée :** Un graphe  $G$  pondéré

**Sortie :** Un arbre  $T$  couvrant de poids minimum de  $G$

$T \leftarrow E(G)$

Trier les arêtes de  $G$  par ordre décroissant de poids

**Tant que**  $T$  a strictement plus de  $n - 1$  arêtes **faire**

$e \leftarrow$  prochaine arête pas encore vue dans cet ordre

**Si** retirer  $e$  ne déconnecte pas  $T$  **alors**

        retirer  $e$  de  $T$

**Fin si**

**Fin tant que**

**Renvoyer**  $T$

---

*Question 14.* Implémenter l'algorithme de Kruskal inversé. On pourra réutiliser la fonction de la Question 3, et introduire toutes les fonctions intermédiaires qui semblent utiles (en expliquant leur rôle dans les commentaires).

### 5 Optimisations

*Question 15.* (\*\*) Notre algorithme passe beaucoup de temps à vérifier si une arête n'ajoute pas un cycle à l'arbre  $T$ . Cette partie de l'algorithme peut être améliorée, par exemple en stockant l'arbre sous une forme rendant plus efficace son parcours. Proposer et implémenter une telle méthode dans une fonction `int meilleur_kruskal(graphe &g)`. Quelle est sa complexité ? *Prendre connaissance de la structure de donnée nommée Union-Find, <https://fr.wikipedia.org/wiki/Union-Find>.*

*Question 16.* (\*\*) Améliorez votre fonction de tri pour avoir une complexité optimale  $\mathcal{O}(n \log n)$  : pour cela, implémentez un tri fusion, ou un tri par tas (ou éventuellement un quicksort, dont la complexité moyenne mais pas en pire cas est  $\mathcal{O}(n \log n)$ ). Vous pouvez vous renseigner sur Wikipédia pour savoir comment fonctionnent ces tris.