Algorithmique et programmation

# Arbre couvrant minimal par la méthode de Kruskal

Yoann Bourse

2009-2010 : Semestre 1

**Résumé**

Nous décrirons l'algorithme de Kruskal et son implémentation pour trouver l'arbre couvrant de poids minimal d'un graphe connexe. Nous analyserons théoriquement et par l'expérience sa complexité. Nous utiliserons ensuite ces résultats pour créer une heuristique de résolution du problème du voyageur de commerce.

## Table des matières

# 1 Principe de l'algorithme

## 1.1 Algorithme de Kruskal

L'algorithme de Kruskal permet de déterminer l'arbre couvrant de poids minimal dans un graphe pondéré positivement. Son principe de base est de représenter les classes de connexité par des ensembles. Au départ, on supprime toutes les arêtes, puis on ajoute celles de poids le plus petit possible nécessaire à unir tous les sommets dans une unique classe de connexité. Ceci se traduit par l'algorithme suivant :

**Kruskal (G)**

$\forall g \in G$ on définit un ensemble représentatif $\mathcal{E}_i(g) = g$.
Pour chaque arête $(u, v)$ de G considérée par poids croissant :
Si $\mathcal{E}_i(u) \neq \mathcal{E}_i(v)$, on les fusionne et on garde (u,v).

L'ensemble des arêtes gardées forme par construction un arbre couvrant. De plus on a gardé les plus petites arêtes nécessaires à la couverture du graphe : c'est l'arbre couvrant minimal.

## 1.2 Ensembles représentatifs

L'efficacité et la complexité de cet algorithme dépend de l'implémentation de ce que nous avons désigné sous le terme "ensemble représentatif". Pour être optimaux, ces derniers doivent répondre à deux principales contraintes.

- Il faut pouvoir trouver facilement l'ensemble auquel appartient un noeud $g$.
- Unir aisément deux ensembles en allourdissant au minimum la structure de données..

Pour répondre à ces besoins, on utilise une structure dite de type **Union-Find** : une foret d'arbres disjoints relativement plats. Les arbres étant plats, on peut facilement remonter à leur racine qui servira d'identification à l'arbre. D'autre part, une union se traduit simplement par le fait de déclarer la racine d'un arbre fils de la racine d'un autre.
Nous cherchons donc à maintenir les propriétés qui rendent cette structure de données pratique dans le cas présent. Ceci est possible grâce aux mécanismes suivants :

**Compression de chemin**

Chaque fois que l'on cherche la racine de l'arbre où est un noeud $g$, on attache $g$ et tous ses parents directement comme fils de la racine : l'arbre est ainsi radicalement aplati, et tout noeud reste proche de la racine.

**Union par rang**

Pour unir deux arbres, on déclare la racine de l'arbre le moins haut comme fille de la racine de l'autre arbre, ce qui augmente la hauteur totale d'au plus 1.

Ces mécanismes garantissent des arbres quasi-plats. La fusion de deux arbres est alors instantanée, et la recherche d'un terme n'est non-instantanée que très rarement (et a alors une complexité très limitée).

### 1.3 Complexité

L'utilisation de cette structure de données optimale permet à l'algorithme Kruskal d'être très performant. Sur un graphe à $n$ noeuds et $a$ arêtes,

- Tri des arêtes :
  Il faut recopier l'ensemble des arêtes (complexité linéraire) pour pouvoir ensuite les trier par un tri efficace. Si la copie se charge d'effacer les doublons présents dans les listes d'adjacence, cette phase a pour complexité totale $O(2a + a * log(a)) = O(a * log(a))$. Nous utiliserons ici un tri fusion, dans la mesure où il est facile à utiliser, adéquat à la structure de listes d'adjacence (le découpage est déjà commencé) et que nous ne sommes pas limités en espace.
- Algorithme Kruskal :
  L'initialisation (construction de la forêt) est linéaire en $n$. On effectue ensuite une boucle sur, dans le pire des cas, les $a$ arêtes. Toutefois, au total, ces boucles n'effectueront que $n$ fusions avant que toutes les classes de connexité ne soient unies, et donc autant d'agrandissement de la profondeur. Nous avons au plus $a$ recherches quasi-instantanées (une étude complexe montre une complexité exacte de l'ordre de $n + 2a * (1 + log_{2+2a/n}(n))$ qui a une croissance extrêmement lente, proche de l'inverse de la fonction d'Ackerman). La complexité totale de cette phase est donc proche de $O(n + a)$.

La complexité totale de l'algorithme sera donc de l'ordre de $O(n + a * log(a)) = O(a * log(a))$.

## 2 Réalisation

Nous allons brièvement décrire les principes généraux de l'implantation de l'algorithme en C. Vous pouvez consulter le code source réorganisé ci-joint pour plus de détails.

### 2.1 Listes d'arêtes

Cette partie correspond aux fichiers list.h et list.c. J'ai créé une structure de liste chainée pour recevoir la liste triée des arêtes du graphe et pour

lire l'entrée en tant que graphe donné par listes d'adjacence. Elle se fond avec la structure d'arête classique, et contient un champ *origin*, non rempli aléatoirement dans les listes d'adjacence, qui sera déduit par la fonction transformant les listes d'adjacence en une grande liste ordonnée de toutes les arêtes du graphe.

**Structure d'arête**

```
typedef struct edge {
int weight ;
int dest ; // attributs réels de l'arête

int origin ; // générée par le programme

struct edge *next ; // structure de liste chainée
} edge ;
```

Une liste est donc un pointeur $*edge$, et un graphe n'est qu'un tableau de listes à $n$ cases. Les fonctions suivantes sont cruciales à l'usage de cette structure de données :

**list* copy (list graph[], int n, int a) ;**

Retourne un tableau de taille $a$ dont chaque case contient une liste formée d'une seule arête (orientée arbitrairement).

**list fusion (list l1, list l2) ;**

Retourne une liste qui est le résultat de la fusion des listes triées l1 et l2, formée en comparant leurs têtes.

Combinées, elles forment le tri fusion.

## 2.2 Graphes

Cette partie correspond aux fichiers graph.h et graph.c, et décrit les fonctions nécessaires à la manipulation de graphes :

**list *random_graph (int n, int a, int maxweight) ;**

Tant qu'il n'a pas ajouté $a$ arêtes, ce programme tire au hasard deux entiers (en vérifiant qu'ils sont distincts) inférieurs à $n$ (qui correspondront aux extrémités de l'arête) et un poids inférieur à *maxweight*. A chaque ajout d'arête, il complète un tableau $done[n][n]$ pour éviter les doublons. Une structure de type Union-Find est utilisée comme dans Kruskal pour garantir la connexité (si le graphe n'est pas connexe, on retire les dernières arêtes jusqu'à ce qu'il le soit).

Rien n'est fait pour forcer la connexité du graphe généré, le cas non-connexe étant relativement peu probable pour un nombre conséquent d'arêtes.

**list list_graph (list graph[],int n,int a) ;**

Cette fonction donne la liste triée des arêtes du graphe grâces aux fonctions annexes :

**void set_origin (list l, int i) ;**
*Remplit le champ "origin" des "edges".*

**list sort_graph (list igraph[],int n, int a) ;**
*Applique le tri fusion.*

### 2.3 Forêts "union-find"

Cette partie correspond aux fichiers forest.h et forest.c. Nous définissons la structure de données décrite en (1.2) comme :

**Structure de forêt**

```
typedef struct node {
struct node *parent ;
int rank ; // Profondeur
int label ; // Etiquette numérique pour l'affichage

// relation fils-ainé/frère droit générée par et pour l'affichage
struct node *brother ;
struct node *son ;
} edge ;
```

Ainsi que les fonctions qui les manipulent :

**list kruskal(list *graph, int n,int a) ;**

Retourne la liste des arêtes formant un arbre couvrant, à l'aide des fonctions annexes :

**node* table (int n) ;**
*Génère un tableau de pointeurs vers n nodes et alloue la mémoire nécessaire.*

**node* find (node* n) ;**
*Retourne un pointeur vers la racine de l'arbre contenant n, qui devient le parent immédiat de n et de ses ancêtres.*

**void merge (node* a, node* b) ;**
*Fusionne les arbres contenant a et b selon le procédé d'union par rang.*

En pratique, nous utiliserons *merge* directement avec la racine des arbres, ce qui le dispensera d'appeler $find$.

### 2.4 Affichage

J'ai également créé de nombreuses fonctions pour afficher les résultats. J'en ai par la suite modifié certaines pour que le résultat du programme soit directement dans le langage de Graphviz, mais cette fonctionnalité n'est pas présentée ici.

**Affichage**

**void print_list (list l) ;**
*Sortie de la forme :* $[>\ adresse]origin \leftarrow dest(weight)[>\ next]$

**void print_graph (list graph[],int n) ;**
*Suite de print_list séparés par une ligne de symboles.*

**void print_forest(node *table, int n) ;**
*Parcours en profondeur par la relation fils ainé / frère droit.*

Un flag *-verbose* permet d'activer des commentaires détaillés du déroulement du programme, et un flag *-time* permet d'afficher des informations sur le temps d'execution (en cycles machine).

# 3 Utilisation

## 3.1 Utilisation du programme

Vous pouvez compiler le projet en utilisant le logiciel Code Blocks ou le makefile fourni. Il créera un fichier "kruskal" que vous pourrez utiliser de la façon suivante :

**Exécution**

```
kruskal [-flags] ? n a maxweight
```

Les trois derniers paramètres sont obligatoires. A noter que le programme vérifie que la valeur de $a$ soit cohérente avec elle de $n$.

**Important :** Par défaut, le module de génération aléatoire d'un graphe est toujours lancé. Pour travailler sur un graphe spécifique, il est nécessaire de l'insérer directement dans le code source du programme. Je n'ai pas mis au point de parseur simplement pour ne pas imposer de syntaxe d'entrée à l'utilisateur, mais son implantation ne pose aucun problème.

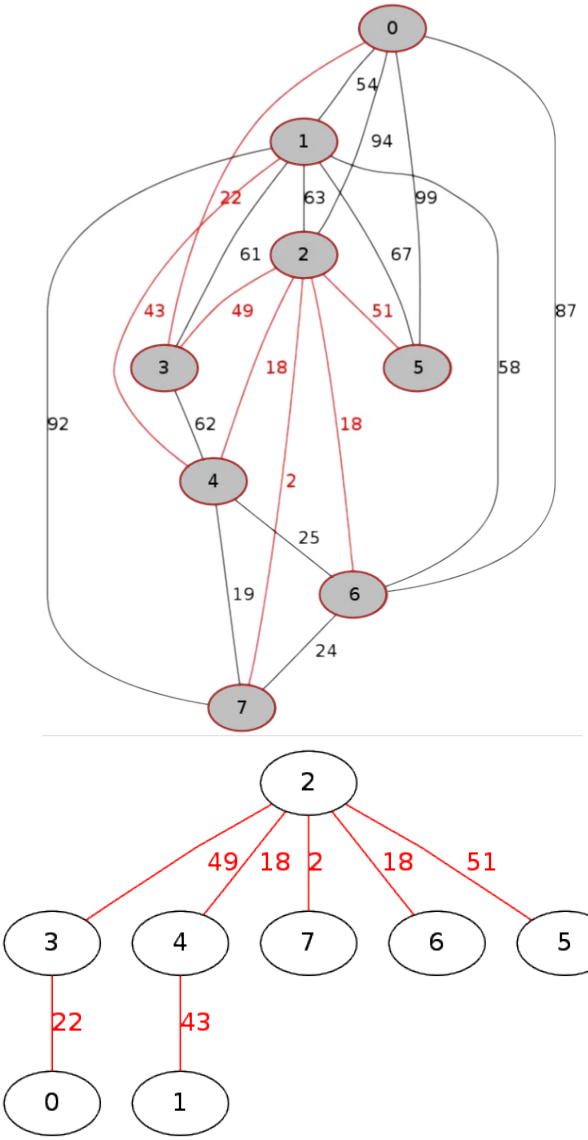
## 3.2 Correction de l'algorithme

Afin d'illustrer le fonctionnement de l'algorithme, nous vous proposons un exemple sur un graphe à 8 noeuds.

**Sortie du programme**

```
[> *]2 → 5(51)[> *]
[> *]2 → 3(49)[> *]
[> *]0 → 3(43)[> *]
[> *]1 → 4(22)[> *]
[> *]2 → 4(18)[> *]
[> *]2 → 6(18)[> *]
[> *]2 → 7(2) [> 0]
```

Les étoiles représentent des valeurs de pointeurs non affichées ici, qui permettent de surveiller la correction de la liste chainée. Ceci se traduit par les dessins suivants :




### 3.3 Complexité réelle

Nous mesurons la complexité à l'aide de la fonction *rdtsc()*, donc en nombre de cycles machine, ce qui permet une mesure précise. Celle-ci est affichable par le flag *-time*.

| $n$ | $a$ | $a * log(a)$ | Génération | Tri | Kruskal | Total |
|---|---|---|---|---|---|---|
| 100 | 150 | 325 | 36k | 22k | 28k | 86k |
| **Variations de** $n$ | | | | | | |
| 500 | 1500 | 4750 | 5M | 4M | 0.2M | 9M |
| 750 | 1500 | 4750 | 11M | 7M | 0.3M | 18M |
| 1000 | 1500 | 4750 | 18M | 12M | 0.3M | 30M |

En laissant $a$ invariant, on observe des variations de la complexité, principalement dans la génération du graphe, ce qui est justifié par le fait que l'on opère sur un tableau de plus grande taille avec des plus grands nombres. Cette justification est aussi valable pour le tri, qui parcours toutes les cases du graphe. La complexité de l'algorithme Kruskal à proprement parler, qui travaille sur les arêtes, n'en souffre que très peu.

| $n$ | $a$ | $a * log(a)$ | Génération | Tri | Kruskal | Total |
|---|---|---|---|---|---|---|
| 100 | 150 | 325 | 36k | 22k | 28k | 86k |
| **Variations de** $a$ | | | | | | |
| 1000 | 1500 | 4750 | 17M | 12M | 0.3M | 29M |
| 1000 | 3000 | 10k | 23M | 17M | 0.5M | 40.5M |
| 1000 | 6000 | 23k | 27M | 18M | 0.7M | 46M |

Comme attendu, la complexité varie avec $a$. Mais les variations de la complexité sont très largement inférieures à celles de $a*log(a)$ : quand ce dernier double, la complexité réelle passe de 30 à 40 millions, puis de 40 à 46. La majoration par $a*log(a)$ est donc très large, pour toutes les étapes de calcul.

Remarquons que l'algorithme Kruskal est globalement extrêmement peu couteux par rapport à la génération et au tri, différence qui se creuse asymptotiquement..

# 4 Problème du voyageur de commerce

Etant donné un ensemble de points, il s'agit de trouver un cycle de distance minimale passant une fois et une seule par chaque point. Nous allons mettre au point une heuristique de résolution de ce problème par l'algorithme de Kruskal.

## 4.1 Principe de l'heuristique

Il s'agit de visualiser le problème sous la forme d'un graphe dont les sommets sont les points à visiter. On aurait un graphe complet où les arêtes seront pondérées par la distance euclidienne entre les deux points qu'elles relient. Il s'agit alors d'utiliser le maximum possible d'arrêtes présentes dans l'arbre couvrant minimal, en le parcourant par un parcours en profondeur.

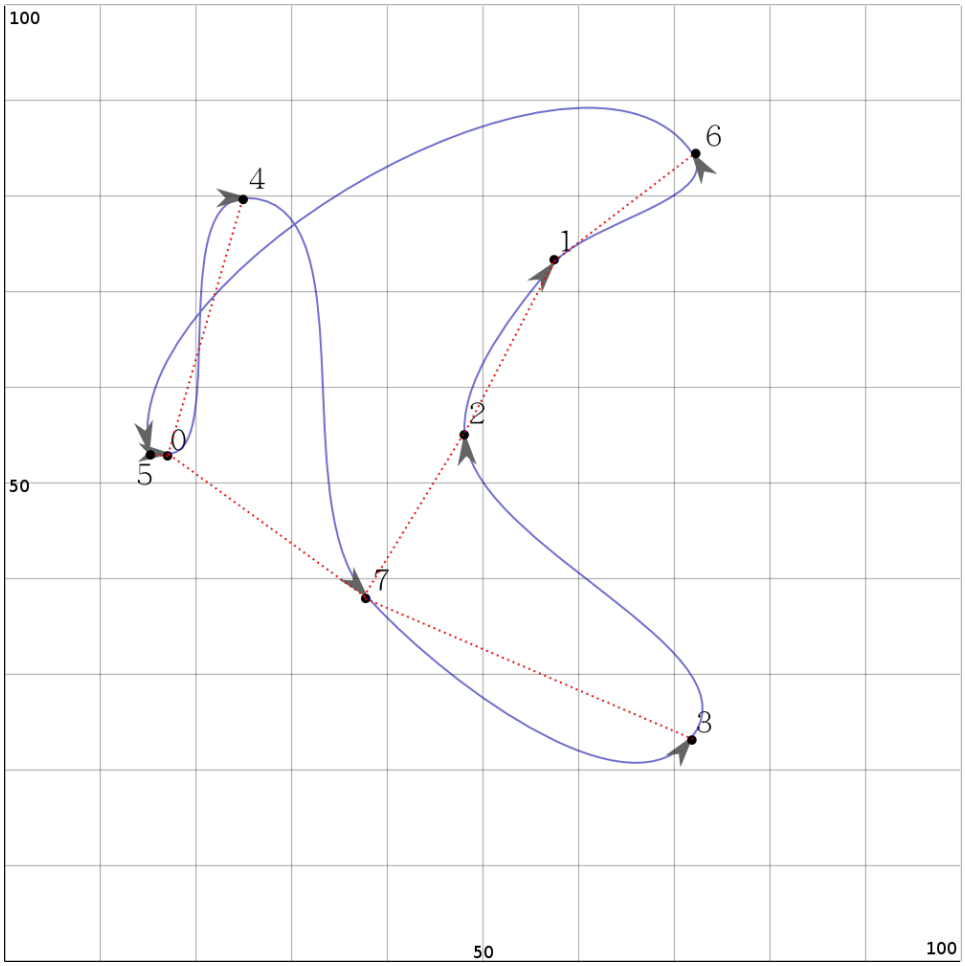
## 4.2 Modifications du programme

- Nouvelle génération du graphe :
  Cette fois ci, il s'agit de tirer aléatoirement les coordonnées des points. On se placera dans le quart supérieur droit du plan, en tirant aléatoirement des valeurs de coordonnées entre 0 et *maxweight*. Il faudra donc modifier entièrement le programme pour transformer tous les poids en poids flotants, ainsi que la nature de *maxweight*.
- Parcours en profondeur de l'arbre couvrant :
  On parcours l'arbre en profondeur en numérotant au passage les noeuds dans l'ordre chronologique d'étude (dans le tableau $done[n]$). Ceci nous permet de revenir sur nos pas en arrivant dans un "cul de sac". Dans ce cas, nous mémorisons le noeud où nous étions arrivé dans la variable $far$ (qui est négative le reste du temps) et remontons chronologiquement notre parcours jusqu'à trouver un noeud frère non étudié.

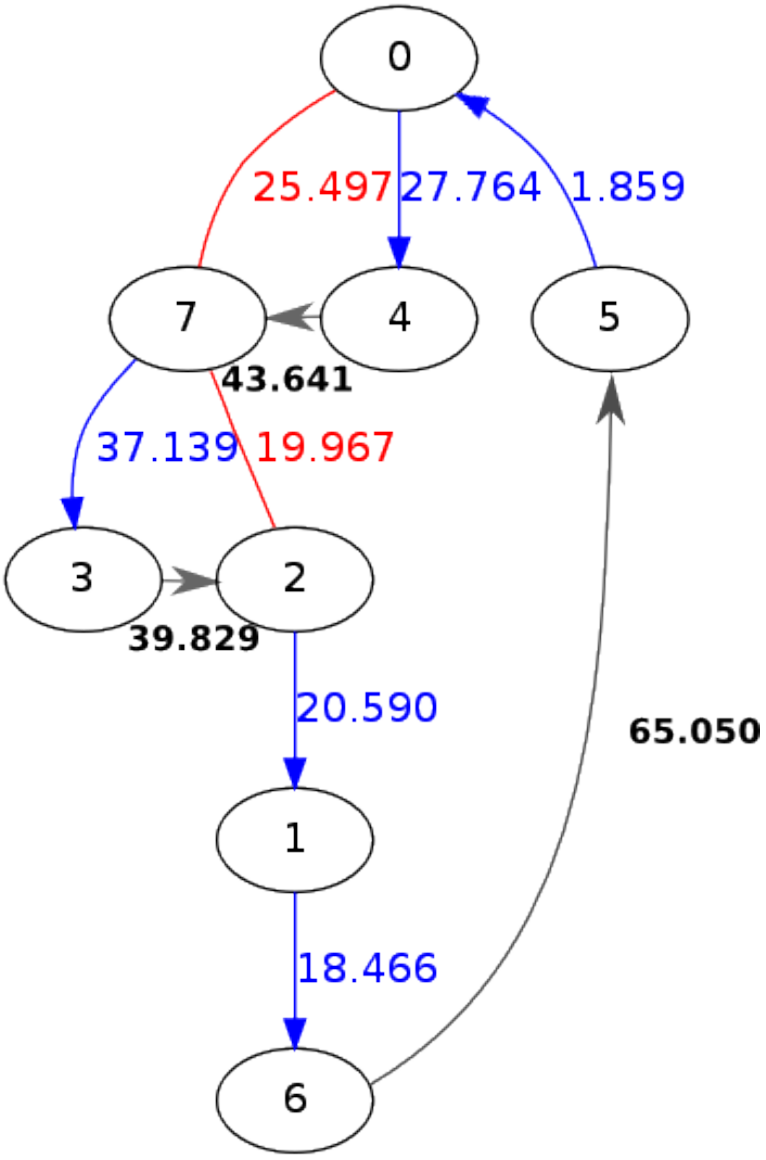
Les modifications étant à la fois nombreuses et simples, j'ai décidé de créer une copie du programme et de transformer tout simplement les poids en flotants. Vous la trouverez dans le dossier "KruskalVDC", et vous pourrez l'utiliser de la même façon que le programme précédent, à l'exception près que *maxweight* doit être un flotant. J'ai choisi de laisser le paramètre $a$ présent dans la syntaxe d'appel au programme, mais il est cependant inutilisé. Nous pourrions le supprimer.

## 4.3 Résultats

Voici les résultats sur 8 sommets générés aléatoirement. Le cycle trouvé est de poids total 254.337 unités de longueur. Les traits sur le schémas sont arrondis par pur esthétisme (les arêtes réelles sont droites).



Les pointillés rouges correspondent à l'arbre couvrant minimal.




Les arêtes en grises ne sont pas dans l'arbre couvrant et sont ajoutés par des "retours en arrière".