Algorithmique et Programmation

# Arbre couvrant minimal par l'algorithme de Kruskal

Yoann Bourse

2009-2010 : Semestre 1

# Plan de la présentation

1. **Principe de l'algorithme**
   - Algorithme de Kruskal
   - Ensembles représentatifs
   - Complexité
2. **Réalisation**
   - Listes d'arêtes et graphes
   - Forêts "union-find"
3. **Résultats**
   - Correction
   - Complexité
4. **Problème du voyageur de commerce**

# Algorithme de Kruskal

Il permet de déterminer l'arbre couvrant de poids minimal dans un graphe pondéré positivement.

**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 un arbre couvrant.

# Ensembles représentatifs

La complexité dépend des ensembles représentatifs. Il faut :

- Trouver facilement l'ensemble auquel appartient $g$.
- Unir aisément deux ensembles.

**Solution : une structure de type Union-Find**

Utiliser une foret d'arbres disjoints relativement plats.

# Ensembles représentatifs

La complexité dépend des ensembles représentatifs. Il faut :

- Trouver facilement l'ensemble auquel appartient $g$.
- Unir aisément deux ensembles.

**Solution : une structure de type Union-Find**

Utiliser une foret d'arbres disjoints relativement plats.

Nous avons deux propriétés à maintenir :

- Tout noeud est proche de la racine qui repère l'arbre.

**Compression de chemin**

Chaque fois que l'on repère dans quel arbre est un élément $g$, on attache $g$ directement à la racine.

- Une union d'arbre se fait rapidement en augmentant au minimum la profondeur.

**Union par rang**

On déclare la racine de l'arbre le moins haut comme fille de la racine de l'autre arbre.

Nous avons deux propriétés à maintenir :

- Tout noeud est proche de la racine qui repère l'arbre.

**Compression de chemin**

Chaque fois que l'on repère dans quel arbre est un élément $g$, on attache $g$ directement à la racine.

- Une union d'arbre se fait rapidement en augmentant au minimum la profondeur.

**Union par rang**

On déclare la racine de l'arbre le moins haut comme fille de la racine de l'autre arbre.

Nous avons deux propriétés à maintenir :

- Tout noeud est proche de la racine qui repère l'arbre.

**Compression de chemin**

Chaque fois que l'on repère dans quel arbre est un élément $g$, on attache $g$ directement à la racine.

- Une union d'arbre se fait rapidement en augmentant au minimum la profondeur.

**Union par rang**

On déclare la racine de l'arbre le moins haut comme fille de la racine de l'autre arbre.

Nous avons deux propriétés à maintenir :

- Tout noeud est proche de la racine qui repère l'arbre.

**Compression de chemin**

Chaque fois que l'on repère dans quel arbre est un élément $g$, on attache $g$ directement à la racine.

- Une union d'arbre se fait rapidement en augmentant au minimum la profondeur.

**Union par rang**

On déclare la racine de l'arbre le moins haut comme fille de la racine de l'autre arbre.

Nous avons deux propriétés à maintenir :

- Tout noeud est proche de la racine qui repère l'arbre.

**Compression de chemin**

Chaque fois que l'on repère dans quel arbre est un élément $g$, on attache $g$ directement à la racine.

- Une union d'arbre se fait rapidement en augmentant au minimum la profondeur.

**Union par rang**

On déclare la racine de l'arbre le moins haut comme fille de la racine de l'autre arbre.

# Complexité

La complexité totale est largement dominée par $a * log(a)$.

### Tri des arêtes

Graphe par liste d'adjacence $\Rightarrow$ Tri fusion :
- Découpage en $a$ cases (fusion des doublons). - fusion des listes triées : $log(2)$ récursions 2 aretes.

### Algorithme Kruskal

- Initialisation linéaire en n
- Dans le pire des cas, une boucle sur les $a$ aretes :
Au plus $n$ fusions, et $a$ recherches quasi-instantanées (*)

(*) La complexité exacte est de $O(n + 2a * (1 + log_{2+2d}(n)))$

# Complexité

La complexité totale est largement dominée par $a * log(a)$.

## Tri des arêtes

Graphe par liste d'adjacence $\Rightarrow$ Tri fusion :
- Découpage en $a$ cases (fusion des doublons). - fusion des listes triées : $log(2)$ récursions 2 aretes.

## Algorithme Kruskal

- Initialisation linéaire en n
- Dans le pire des cas, une boucle sur les $a$ aretes :
Au plus $n$ fusions, et $a$ recherches quasi-instantanées (*)

(*) La complexité exacte est de $O(n + 2a * (1 + log_{2+2d}(n)))$

# Complexité

La complexité totale est largement dominée par $a * log(a)$.

### Tri des arêtes

Graphe par liste d'adjacence $\Rightarrow$ Tri fusion :
- Découpage en $a$ cases (fusion des doublons). - fusion des listes triées : $log(2)$ récursions 2 aretes.

### Algorithme Kruskal

- Initialisation linéaire en n
- Dans le pire des cas, une boucle sur les $a$ aretes :
Au plus $n$ fusions, et $a$ recherches quasi-instantanées (*)

(*) La complexité exacte est de $O(n + 2a * (1 + log_{2+2d}(n)))$

# Liste et arêtes

Voir list.h et list.c.

**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 un pointeur $*edge$, un graphe est un tableau de listes.

# Tri fusion

**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).

Puis en divisant a par deux a chaque itération,

**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.

# Tri fusion

**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).

Puis en divisant a par deux a chaque itération,

**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.

# Graphes

Voir graph.h et graph.c.

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

- Tire au hasard *a* fois deux entiers distincts inférieurs à *n* (extrémités de l'arête) et un poids inférieur à maxweight.
- Tableau *done*[*n*][*n*] pour éviter les doublons.
- Structure union-find pour garantir la connexité.

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

**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.

# Graphes

Voir graph.h et graph.c.

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

- Tire au hasard *a* fois deux entiers distincts inférieurs à *n* (extrémités de l'arête) et un poids inférieur à maxweight.
- Tableau *done*[*n*][*n*] pour éviter les doublons.
- Structure union-find pour garantir la connexité.

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

**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.

# Forêts de type "union-find"

Voir forest.h et forest.c.

## 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;
```

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

Retourne la liste des arêtes formant un arbre couvrant.

**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* et en fait son parent immédiat.

**void merge (node* a, node* b) ;**

Fusionne les arbres contenant *a* et *b* selon le procédé d'*union par rang*.

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

Retourne la liste des arêtes formant un arbre couvrant.

**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* et en fait son parent immédiat.

**void merge (node* a, node* b) ;**

Fusionne les arbres contenant *a* et *b* selon le procédé d'*union par rang*.

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

Retourne la liste des arêtes formant un arbre couvrant.

**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* et en fait son parent immédiat.

**void merge (node* a, node* b) ;**

Fusionne les arbres contenant *a* et *b* selon le procédé d'*union par rang*.

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

Retourne la liste des arêtes formant un arbre couvrant.

**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* et en fait son parent immédiat.

**void merge (node* a, node* b) ;**

Fusionne les arbres contenant *a* et *b* selon le procédé d'*union par rang*.

# Affichage

**void print_list (list l) ;**

Sortie de la forme : $[> \textit{adresse}]\textit{origin} \leftarrow \textit{dest}(\textit{weight})[> \textit{next}]$

**void print_graph (list graph[],int n) ;**

Suite de print_list séparés.

**void print_forest(node *table, int n) ;**

Parcours en profondeur par la relation fils ainé / frère droit.

Flag *-verbose* active les commentaires détaillés du déroulement du programme.

# 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.

**void print_forest(node *table, int n) ;**

Parcours en profondeur par la relation fils ainé / frère droit.

Flag -*verbose* active les commentaires détaillés du déroulement du programme.

# Utilisation

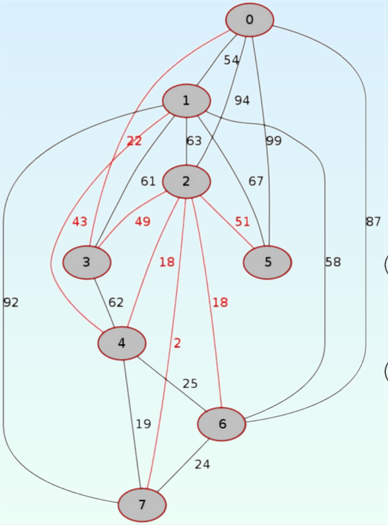
### Compilation

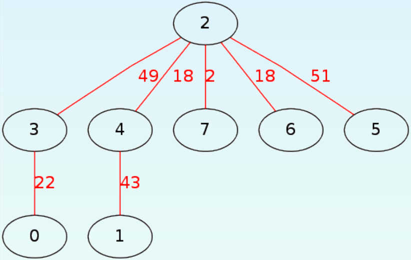
Compilation assistée par Code Blocks ou makefile.

### Exécution

bin/Debug/Kruskal [-flags] ? n a maxweight

# Correction de l'algorithme



# Complexité réelle

Flag *-time*.
Fonction *rdtsc()*.

| $n$ | $a$ | $a * log(a)$ | 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 |
| **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 |

# Voyageur de commerce

## Principe de l'heuristique

Parcourir l'arbre couvrant minimal.

## Nouvelle génération

Les sommets sont des points dans un plan, les poids sont leurs distances euclidiennes.
**Exécution :** On travaille dans le quart superieur droit du plan, maxweight (float) $\leftarrow$ maximum des coordonnées.

## Parcours en profondeur de l'arbre couvrant

- Numérotation des noeuds dans $done[n]$.
- Départ d'un retour en arrière dans *far*, négatif sinon.

# Voyageur de commerce

## Principe de l'heuristique

Parcourir l'arbre couvrant minimal.

## Nouvelle génération

Les sommets sont des points dans un plan, les poids sont leurs distances euclidiennes.
**Exécution :** On travaille dans le quart superieur droit du plan, maxweight (float) $\leftarrow$ maximum des coordonnées.

## Parcours en profondeur de l'arbre couvrant

- Numérotation des noeuds dans *done*[*n*].
- Départ d'un retour en arrière dans *far*, négatif sinon.

# Voyageur de commerce

### Principe de l'heuristique

Parcourir l'arbre couvrant minimal.

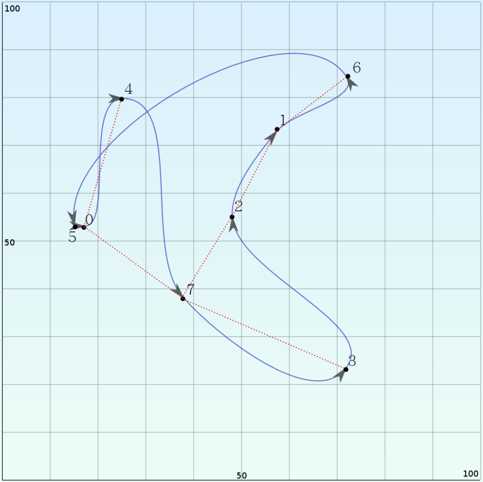
### Nouvelle génération

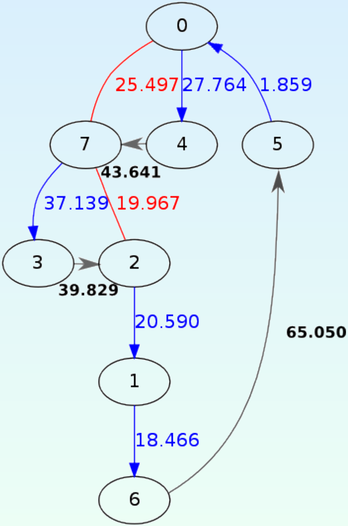
Les sommets sont des points dans un plan, les poids sont leurs distances euclidiennes.
**Exécution :** On travaille dans le quart superieur droit du plan, maxweight (float) $\longleftarrow$ maximum des coordonnées.

### Parcours en profondeur de l'arbre couvrant

- Numérotation des noeuds dans $done[n]$.
- Départ d'un retour en arrière dans *far*, négatif sinon.

# Résultats




Poids total : 254.337