Chapître VII.
Arbres couvrants minimaux
(KRUSKAL contre PRIM)

1. Applications et motivations

-) Conception d'un circuit électrique

Lors de la phase de conception d'un circuit électrique, on a souvent besoin de relier entre elles les broches de composants électriquement équivalents.

-) En amerique aux environs de 1850: L'héritage

Il était une fois ...

2. Problème de l'arbre couvrant minimal

-) donnée:

graphe non-orienté connexe $G = (V, E)$

poids $w(u, v)$ pour chaque arête $\{u, v\} \in E$

-) problème:

On souhaite trouver un sous-ensemble acyclique $T \subseteq E$ qui connecte tous les sommets de G et dont le poids total

$$w(T) = \sum_{\{u,v\} \in T} w(u, v)$$

est minimal.

Puisque $G' = (V, T)$ est connexe et sans cycle le graphe G' est un arbre et aussi un sous-graphe de G. Alors on l'appelle **arbre couvrant**.

Le problème de la détermination de l'arbre $G' = (V, T)$ porte le nom de **problème de l'arbre couvrant minimal**.

Exemple:

C'est un problème classique du domaine "algorithmes de graphes". Le premier algorithme avait été obtenu en 1926 par Boruvka.

3. Deux algorithmes gloutons

Nous examinerons deux algorithmes permettant de résoudre le problème de l'arbre couvrant minimal:

-) algorithme de Kruskal

-) algorithme de Prim

Les deux algorithmes sont des **algorithmes gloutons**. Les méthodes gloutonnes présentées dans ce chapître constituent une application classique des notions théoriques introduites dans le Chapître VI.

Observation: Puisqu'une solution possible du problème est un arbre $G' = (V, T)$ alors $|T| = |V| - 1 = n - 1$.

Dans les deux algorithmes, l'ensemble A est l'ensemble courant d'arêtes déjà choisi pour T.

Algorithme de Kruskal

On trie les arêtes de E par ordre croissant de pondération w et on ajoute les arêtes dans cet ordre. On n'ajoute pas une arête $\{u, v\} \in E$ si $G' = (V, A \cup \{u, v\})$ a un cycle.

$G = (V, A)$ **est une fôret.**

C'est-à-dire que le graphe intermédiaire $G = (V, A)$ peut avoir plus qu'une composante connexe.

Algorithme de Prim

On choisit un sommet quelconque comme racine r. Dans chaque étape, on ajoute une arête de moindre poids parmi celles qui relient l'arbre à un sommet extérieur à l'arbre.

$G = (V, A)$ **est un arbre unique.**

C'est-à-dire que tous les graphes intermédiaires $G = (V, A)$ sont connexes.

4. L'algorithme générique

Les deux algorithmes étudiés utilisent une **approche gloutonne**, bien qu'elle soit appliquée différemment par chacun d'eux.

La validité des algorithmes est garantie par le même principe. Ce principe est explicité par l'**algorithme "générique"** ci-dessous.

$ACM - GENERIQUE(G, w)$

$A \leftarrow \emptyset$
tant que A ne forme pas un arbre couvrant
    faire trouver une arête $\{u, v\}$ sûre pour A
            $A \leftarrow A \cup \{\{u, v\}\}$
retourner A

5. Arête sûre

Définition. On appelle une arête $\{u, v\}$ de graphe connexe $G = (V, E)$ **arête sûre pour** A si:

1) A est un sous-ensemble d'une solution minimale $T \subseteq E$ et

2) $A' = A \cup \{\{u, v\}\}$ reste un sous-ensemble d'une solution minimale $T' \subseteq E$

C'est-à-dire ajoutant une arête sûre pour A dans l'ensemble A on ne fait pas d'erreur au sens qu'il y a encore une extension de A' dit T' qui est une solution minimale.

En conséquence l'algorithme $ACM-GENERIQUE$ respecte l'**invariant** que l'ensemble A est un sous-ensemble d'une solution minimale. Alors $ACM - GENERIQUE$ trouve toujours un arbre couvrant minimal.

Question: Comment peut-on trouver une arête sûre pour $A \subseteq E$? Est-ce vraiment toujours possible?

Définition. La **coupure** $(S, V \setminus S)$ d'un graphe non orienté $G = (V, E)$ est une partition de V. On dit qu'une arête $\{u, v\} \in E$ **traverse** la coupure $(S, V \setminus S)$ si $u \in S$ et $v \in V \setminus S$, ou $v \in S$ et $u \in V \setminus S$.

Définition. On dit qu'une coupure $(S, V \setminus S)$ **respecte** un ensemble $A \subseteq E$ si aucune arête de A ne traverse la coupure.

6. La règle de reconnaissance des arêtes sûres

Théorème. *Soit $G = (V, E)$ un graphe non orienté connexe, avec une fonction de pondération w à valeurs réelles définie sur E.*

Soit $A \subseteq E$ et $A \subseteq T$ pour un arbre couvrant minimal $G' = (V, T)$ de G.

Soit $(S, V \setminus S)$ une coupure quelconque de G qui respecte A, et soit $\{u, v\} \in E$ une arête de poids minimal parmi toutes les arêtes qui traversent la coupure $(S, V \setminus S)$.

Alors $\{u, v\}$ est une arête sûre pour A.

C'est-à-dire qu'il y a un arbre couvrant minimal $G'' = (V, T')$ tel que $A \cup \{\{u, v\}\} \subseteq T'$.

7. La validité des algorithmes

Corollaire. *Soit $G = (V, E)$ un graphe non orienté connexe, avec une fonction de pondération w à valeurs réelles définie sur E.*

Soit $A \subseteq E$ et $A \subseteq T$ pour un arbre couvrant minimal $G' = (V, T)$ de G.

Soit C une composante connexe de la fôret $G_A = (V, A)$.

Si $\{u, v\}$ est une arête de poids minimal parmi toutes les arêtes reliant C à une autre composante quelconque de G_A, alors $\{u, v\}$ est une arête sûre pour A.

Démonstration: La coupure $(C, V \setminus C)$ respecte A, est $\{u, v\}$ est donc une arête de poids minimal parmi toutes les arêtes qui traversent la coupure $(C, V \setminus C)$. $\square$

A chaque étape, l'algorithme de Kruskal et l'algorithme de Prim choisissent une arête, satisfaisant les hypothèses du corollaire, qui est donc toujours une arête sûre pour A.

Alors les deux algorithmes sont valides bien que le choix glouton des algorithmes soit différents.

Remarque. Bien que les deux algorithmes soient différents ils donnent naturellement le même arbre couvrant minimal si l'arbre couvrant minimal de l'entrée est unique.

Points très interessants:

1) Les deux algorithmes sont élaborés à partir de l'algorithme générique. Cependant il faut des structures de données complètement différentes pour une implementation efficace.

2) On peut dire que chaque algorithme a besoin d'une structure de données particulière (et donc il faudrait la construire si elle n'existait pas encore).

8. L'algorithme de PRIM

L'algorithme de Prim ressemble à l'algorithme de Dijkstra.

L'algorithme de Prim a pour propriété que les arêtes de l'ensemble A constituent toujours un arbre unique.

L'arbre démarre d'un sommet initial r arbitraire, et croît jusqu'à couvrir tous les sommets de G.

A chaque étape, une arête de poids minimal parmi toutes les arêtes qui traversent la coupure $(A, V \setminus A)$ est ajoutée à l'ensemble A.

Choix glouton: Cette stratégie est gloutonne puisqu'à chaque étape, l'arbre est augmenté avec une arête qui accroît le moins possible le poids total de l'arbre.

Pour implémenter efficacement l'algorithme de Prim, l'important est de faciliter la sélection de la nouvelle arête à ajouter dans l'arbre constitué des arêtes de A.

Pour ce faire on utilise une **file de priorité** F contenant tous les sommets du graphe.

Pour chaque sommet $v \in F$, $cle[v]$ est le poids minimal d'une arête reliant v à un sommet de l'arbre, qui est la composante connexe de $G_A = (V, A)$ contenant r. Par convention, $cle[v] = \infty$ si une telle arête n'existe pas.

Le champ $\pi[v]$ désigne le "père" de v dans l'arbre.

Quand l'algorithme se termine, la file de priorité F est vide. L'arbre couvrant minimal de G est $G' = (V, A)$ tel que

$$A = \Big\{\{v, \pi[v]\} : v \in V \setminus \{r\}\Big\}.$$

$ACM - PRIM(G, w, r)$

$F \leftarrow V$
pour chaque $u \in F$
    faire $cle[u] \leftarrow \infty$
$cle[r] \leftarrow 0$
$\pi[r] \leftarrow NIL$
tant que $F \neq \emptyset$
   faire $u \leftarrow EXTRAIRE - MIN(F)$
            pour chaque $v \in Adj[u]$
               faire si $v \in F$ et $w(\{u, v\}) < cle[v]$
                           alors $\pi[v] \leftarrow u$
                                   $cle[v] \leftarrow w(\{u, v\})$

9. Files de priorité

Une **file de priorité** est une structure de données permettant de gérer un ensemble dynamique S d'éléments, chacun ayant une valeur associée appelée **clé**.

Une file de priorité supporte les opérations suivantes.

-) $INSERER(S, x)$

-) $MINIMUM(S)$

-) $EXTRAIRE{-}MIN(S)$ supprime et retourne l'élément de S ayant le plus petite clé.

Souvent on souhaite de supporter d'autres opérations en particulier:

-) $DIMINUER - CLE(S, x, k)$ affecte a l'élément x de S la nouvelle valeur de clé k, dont on suppose qu'elle est inférieure à sa valeur de clé courante.

Théorème. *L'algorithme de Prim s'exécute en temps* $O(m + n \log n)$ *et il donne toujours un arbre couvrant minimal de l'entrée.*

Démonstration: La validité est une consequence immediate de corollaire parce que l'algorithme choisit à chaque étape une aréte de poids minimal parmi toutes les arêtes qui traversent la coupure $(A, V \setminus A)$.

La performance de l'algorithme de Prim dépend de la manière dont la file de priorité F est implementée. En effet on peut regarder l'exécution de l'algorithme comme une séquence des opérations d'une file de priorité:

-) $CONSTRUIRE$ (ou n fois $INSERER$)

-) n fois $EXTRAIRE - MIN$

-) au plus $2m$ fois $DIMINUER - CLE$

Si on utilise un **tas** (ordinaire) ou un **tas binomial** on obtient $O(m \log n)$.

Si on utilise un **tas de Fibonacci** on obtient $O(m + n \log n)$. $\square$

10. L'algorithme de KRUSKAL

L'algorithme de Kruskal s'inspire directement de l'algorithme générique.

Il trouve une arête sûre à ajouter à la fôret en cherchant, parmi toutes les arêtes reliant deux arbres quelconques de la fôret, une arête de poids minimal.

Choix glouton: Cette stratégie est gloutonne puisqu'à chaque étape, l'algorithme ajoute à la fôret une arête de poids minimal.

Pour implémenter efficacement l'algorithme de Kruskal, l'important est de faciliter le test si deux sommets appartiennent au même arbre. La nouvelle arête à ajouter dans l'arbre constitué une arête de A. (C'est-à-dire il faut trouver tres rapide si deux sommets appartiennent au même composante connexe de la fôret $G = (V, A)$.)

Pour ce faire on utilise une **structure de données d'ensembles disjoints (UNION-FIND structure)** pour gérer plusieurs ensembles disjoints.

Chaque ensemble contient les sommets d'un arbre de la fôret courante $G_A = (V, A)$.

$ACM - KRUSKAL(G, w)$

$A \leftarrow \infty$
pour chaque sommet $v \in V$
    faire CREER-ENSEMBLE(v)
trier les arêtes de E par ordre croissant de pondération w
pour chaque arête $\{u, v\} \in E$, par ordre de pondération croissante
    faire si $TROUVER{-}ENSEMBLE(u) \neq TROUVER{-}ENSEMBLE(v)$
        alors $A \leftarrow A \cup \{\{u, v\}\}$
            $UNION(u, v)$
retourner A

11. Structure de données d'ensembles disjoints

Une **structure de données d'ensembles disjoints (UNION-FIND structure)** est une structure de données qui maintient à jour une collection $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ d'ensemble dynamique disjoints. Chaque ensemble est identifié par un **représentant**, qui est un certain membre de l'ensemble.

Une structure de données d'ensembles disjoints supporte les opérations suivantes.

-) $CREER - ENSEMBLE(x)$

-) $UNION(x, y)$

-) $TROUVER - ENSEMBLE(x)$

$CREER - ENSEMBLE(x)$ est une opération qui crée un nouvel ensemble dont le seul membre est donc le représentant est x. (Il faut que x ne soit pas déjà membre d'un autre ensemble.)

$UNION(x, y)$ réunit les ensemble dynamiques qui contiennent x et y dans un nouvel ensemble qui est l'union de ces deux ensemble. Le représentant de l'ensemble est un membre quelconque.

$TROUVER{-}ENSEMBLE(x)$ retourne un pointeur sur le représentant de l'ensemble contenant x.

Théorème. *L'algorithme de Kruskal s'exécute en temps $O(m + n \log n)$ et donne toujours un arbre couvrant minimal de l'entrée.*

Démonstration: La validité est une consequence immediate de corollaire parce que l'algorithme choisit à chaque étape une aréte de poids minimal et il ajoute cette arête à l'ensemble A si les extremities d'arête appartiennent à deux composante connexe différents de la fôret courante.

La performance de l'algorithme de Kruskal dépend de l'implementation de la structure de données d'ensembles disjoints.

En effet on peut regarder l'exécution de l'algorithme comme une séquence des opérations d'une structure de données d'ensembles disjoints.

-) n fois $CREER-ENSEMBLE$

-) au plus m fois $TROUVER-ENSEMBLE$

-) $n-1$ fois $UNION$

Utilisant une implementation célèbrée de la structure de données d'ensembles disjoints, qui était obtenu par R. Tarjan, on obtient une implementation qui s'exécute en temps $O(m+n\log n)$.

$\square$