Chapitre 5 : Problème de l'arbre couvrant de poids minimum

Y. DIENG, Département Informatique
UFR ST, Université Assane Seck de Ziguinchor

5. desember 2022

1 Arbre couvrant de poids minimum

Lors de la phase de conception de circuits électroniques, on a souvent besoin de relier entre elles les broches de composants électriquement équivalents. Pour interconnecter un ensemble de n broches, on peut utiliser un arrangement de $n-1$ branchements, chacun reliant deux broches. Parmi tous les arrangements possibles, celui qui utilise une longueur de branchements minimale est souvent le plus souhaitable.

On peut modéliser ce problème de câblage à l'aide d'un graphe non orienté connexe $G = (V, E)$ où E représente l'ensemble des broches, où E l'ensemble des interconnections possibles entre paires de broches, et où pour chaque arête $(u, v) \in E$, on a un poids $w(u, v)$ qui spécifie le coût (longueur de fil nécessaire) pour connecter u et v. On souhaite alors trouver un sous-ensemble acyclique $\mathsf{T} \subseteq E$ qui connecte tous les sommets et dont le poids total

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

soit minimum. Puisque T est acyclique et connecte tous les sommets, il doit former un arbre, que l'on appelle arbre couvrant car il « couvre » le graphe G. Le problème de la détermination de l'arbre T porte le nom de problème de l'arbre couvrant minimal.

Dans ce chapitre, nous examinerons deux algorithmes permettant de résoudre le problème de l'arbre couvrant minimum : l'algorithme de `Kruskal` et l'algorithme de `Prim`.

1.1 Construction d'un arbre couvrant minimum

Supposons qu'on dispose d'un graphe $G = (V, E)$ non orienté connexe avec une fonction de pondération w, et qu'on souhaite trouver un arbre couvrant minimum pour G. Les deux algorithmes étudiés dans ce chapitre utilisent une approche gloutonne, bien qu'elle soit appliquée différemment par chacun d'eux. Cette stratégie gloutonne est explicitée par l'algorithme « générique » suivant, qui fait pousser l'arbre couvrant minimum arête par arête.

L'algorithme gère un ensemble d'arêtes D, en conservant l'invariant de boucle que voici : `avant chaque itération,` E `est un sous-ensemble d'un arbre couvrant minimum`.

A chaque étape, on détermine une arête (u, v) qui peut être ajoutée à D tout en respectant cet invariant, au sens où $D \cup \{(u, v)\}$ est également un sous-ensemble d'un arbre couvrant minimum. On appelle ce type d'arête arête sûre pour D, car on peut l'ajouter à D sans détruire l'invariant.

```
ACM-GEENEERIQUE(G, w)
1 D = Ensemble Vide
2 tant que D ne forme pas un arbre couvrant
3               faire trouver une arête (u, v) qui est sûre pour D
4                       D = D + {(u, v)}
5 retourner D
```

`Initialisation :` Après la ligne 1, l'ensemble D satisfait de manière triviale à l'invariant.

`Conservation :` La boucle des lignes 2-4 conserve l'invariant en ajoutant uniquement des arêtes sûres.

`Terminaison :` Toutes les arêtes ajoutées à D étant dans un arbre couvrant minimum, l'ensemble D retourné en ligne 5 est forcément un arbre couvrant minimum.

La partie délicate consiste, à trouver une arête sûre en ligne 3. Il doit en exister un puisque, quand on exécute la ligne 3, l'invariant impose qu'il y ait un arbre couvrant T tel que $D \subseteq T$. Dans le corps de la boucle tant que, D doit être un sous-ensemble distinct de T ; par conséquent, il doit y avoir une arête $(u, v) \in T$ tel que $(u, v) \notin D$ et tel que (u, v) est sûre pour D.

La qustion à se poser est comment reconnaitre efficement une arête sûre. Nous avons besoins de donner quelques définitions.

Définition 1. *Une coupure d'un graphe $G = (V, E)$ est un couple partitionnant l'ensemble V de sommets, c'est-à-dire un couple de la forme $(P, V - P)$ avec $P \subseteq V$.*

- *Une arête $e \in E$ `traverse` la coupure $(P, V - P)$ si l'une des extrémités est dans P est l'autre dans $V - P$.*
- *Une coupure `respecte` un ensemble d'arêtes $D \subseteq E$ si aucune arête $e \in D$ ne traverse la coupure.*
- *Une arete est une textttarête minimale pour la traversé de la coupe si son poids est minimale parmi toutes les arêtes qui traversent la coupe. Notez qu'il peut y avoire plusieurs arêtes minimale*

Propriété 1. *Soit $G = (V, E)$ un graphe connexe. Soit D un ensemble d'arêtes contenu dans un ACM. Si $(P, V - P)$ est une coupure respectant D et si e est une arête de poids minimal traversant $(P, V - P)$, alors $D \cup \{e\}$ est contenu dans un arbre couvrant minimal.*

Preuve. 1. *Soit T un ACM contenant D. Si $e \in T$, alors la conclusion est immédiate. Considérons le cas contraire c'est à dire $e \notin T$. Puisque T est un arbre, $T \cup \{e\}$ contient un cycle c. Le cycle c contient e, ainsi le chemin ω obtenu à partir de c en supprimant e contient un sommet de P et un autre de $V \breve{} P$. L'une de ses arêtes f traverse ainsi $(P, V - P)$. L'ensemble des arêtes $U = (T \setminus f) \cup \{e\}$ est connexe car $T \cup \{e\}$ est connexe et possède un cycle*

contenant e et f , a même cardinalité que T et est donc un arbre couvrant de G.

De plus, le poids de f est au moins celui de e. On en déduit que le poids de U, égal à $p(U) = p(T) - p(f) + p(e)$ et au plus à celui de $p(T)$. Ainsi, U est une solution contenant e.

■

Corollaire 1. *Si D un ensemble d'arêtes contenu dans un ACM et si $(P, V-P)$ est une coupure respectant D. Alors sii e est une arête de poids minimal traversant $(P, V-P)$, alors e est une arête sûre.*

La Propriété **??** permet de mieux comprendre le comportement de l'algorithme `ACM-GÉNÉRIQUE` sur un graphe connexe $G = (V, E)$. Pendant l'exécution, l'ensemble D est toujours acyclique ; sinon, un arbre couvrant minimum incluant D contiendrait un cycle, ce qui amène une contradiction. A un moment quelconque de l'exécution, le graphe $G[D] = (V, D)$ est une forêt et chacune des composantes connexes de $G[D]$ est un arbre. (Certains arbres peuvent ne contenir qu'un seul sommet, comme c'est le cas par exemple au commencement de l'algorithme : D est vide et la forêt contient $|V|$ arbres, un pour chaque sommet.) Par ailleurs, n'importe quelle arête sûre (u, v) pour D relie des composantes distinctes de $G[D]$, puisque $D \cup \{(u, v)\}$ doit être acyclique.

La boucle des lignes 2–4 de `ACM-GÉNÉRIQUE` est exécutée $|S| - 1$ fois, car chacun des $|S| - 1$ arêtes d'un arbre couvrant minimum est successivement déterminé. Au départ, quand $D = \emptyset$, il existe $|V|$ arbres dans $G[D]$, et chaque itération réduit ce nombre de 1. Quand la forêt ne contient plus qu'un seul arbre, l'algorithme se termine.

2 Algorithmes de Kruskal et de Prim

Les deux algorithmes d'arbre couvrant minimum décrits dans cette section sont élaborés à partir de l'algorithme générique. Ils utilisent chacun une règle spécifique pour déterminer l'arête sûre recherchée en ligne 3 de `ACM-GÉNÉRIQUE`. Dans l'algorithme de `Kruskal`, l'ensemble D est une forêt. L'arête sûre ajoutée à D est toujours une arête de moindre poids du graphe qui relie deux composantes distinctes. Dans l'algorithme de `Prim`, l'ensemble D est un arbre. L'arête sûre ajoutée à D est toujours une arête de moindre poids qui relie l'arbre à un sommet extérieur.

2.1 Algorithme de Kruskal

L'algorithme de Kruskal s'inspire directement de l'algorithme générique de l'arbre couvrant minimum donné à la section précédente. Il trouve une arête sûre à ajouter à la forêt en cherchant, parmi toutes les arêtes reliant deux arbres quelconques de la forêt, une arête (u, v) de poids minimal. Soient C_1 et C_2 les deux arbres reliés par (u, v). Puisque (u, v) doit être une arête minimale reliant C_1àun autre arbre, le corollaire **??** implique que (u, v) est une arête sûre pour C_1. L'algorithme de Kruskal est un algorithme glouton car, à chaque étape, il ajoute à la forêt une arête de poids minimal.

Notre implémentation de l'algorithme de Kruskal ressemble à celle de l'algorithme de calcul des composantes connexes. Elle utilise une structure de données d'ensembles disjoints pour gérer plusieurs ensembles disjoints. Chaque ensemble contient les sommets d'un arbre de la forêt courante. L'opération `TROUVER- ENSEMBLE(u)` retourne un élément représentatif de l'ensemble qui contient u. Ainsi, on peut déterminer si deux sommets u et v appartiennent au même arbre, en testant si `TROUVER- ENSEMBLE(u)` est égal à `TROUVER-ENSEMBLE(v)`. La combinaison des arbres est effectuée par la procédure `UNION`.

```
ACM-KRUSKAL(G = (V, E), w)
1 D ← Ensemble Vide
2 pour chaque sommet v de V
3            faire CREEER-ENSEMBLE(v)
4 trier les arêtes de E par ordre croissant de poids w
5 pour chaque arête e = (u, v) de E pris par ordre de poids croissant
6            faire si TROUVER-ENSEMBLE(u) <> TROUVER-ENSEMBLE(v)
7                           alors D←D+ {(u,v)}
8                                    UNION(u, v)
9 retourner D
```

Le temps d'exécution de l'algorithme de Kruskal sur un graphe $G = (V, E)$ dépend de l'implémentation de la structure d'ensembles disjoints. Pour la suite, nous retiendrons l'implémentation par forêt d'ensembles disjoints. L'initialisation de l'ensemble D en ligne 1 requiert un temps $O(1)$, et le temps pris par le tri des arcs en ligne 4 est $O(E \log E)$. (Nous tiendrons compte du coût des $|S|$ opérations CREER-ENSEMBLE de la boucle pour des lignes 1–3 dans un moment.) La boucle pour des lignes 5–8 fait $O(E)$ opérations TROUVER-ENSEMBLE et UNION sur la forêt d'ensembles disjoints. Avec les $|V|$ opérations CRÉER-ENSEMBLE, cela fait un total de $O((V + E)\alpha(S))$ temps, où α est la fonction à très lente croissance. Comme G est censé être connexe, on a $|E| \geq |V| - 1$, et donc les opérations d'ensembles disjoints prennent un temps $O(E\alpha(V))$. En outre, comme $\alpha(|V|) = O(\log V) = O(\log E)$, le temps d'exécution total de l'algorithme de Kruskal est $O(E \log E)$. On observant que $|A| < |S|^2$, on a $\log |E| = O(\log V)$, et l'on peut donc reformuler le temps d'exécution de l'algorithme de Kruskal comme étant $O(E \log V)$.

2.2 Algorithme de Prim

Comme l'algorithme de Kruskal, l'algorithme de Prim est un cas particulier de l'algorithme générique étudié à la section précédente. L'algorithme de Prim ressemble à l'algorithme de Dijkstra de recherche des plus courts chemins d'un graphe. L'algorithme de Prim a pour propriété que les arêtes de l'ensemble D constituent toujours un arbre. L'arbre démarre d'un sommet racine r arbitraire puis croît jusqu'à couvrir tous les sommets de V. A chaque étape, une arête minimale est ajoutée à l'arbre D pour relier celui-ci à un sommet isolé de $G[D] = (V, D)$.

D'après le corollaire X, cette règle permet de n'ajouter que des arêtes qui sont sûres pour D ; ainsi, quand l'algorithme se termine, les arêtes de D forment un arbre couvrant minimum. Cette stratégie est gloutonne puisque, à chaque

étape, l'arbre est augmenté d'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 à l'arbre constitué des arêtes de D. Dans le pseudo code qui suit, le graphe connexe G et la racine r de l'arbre couvrant minimum à construire sont les entrées de l'algorithme. Pendant l'exécution, tous les sommets qui n'appartiennent pas à l'arbre se trouvent dans une file de priorités min F basée sur un champ clé. Pour chaque sommet v, $cle[v]$ est le poids minimal d'une arête reliant v à un sommet de l'arbre ; par convention, $cle[v] = \infty$ si une telle arête n'existe pas. Le champ $parent[v]$ désigne le parent de v dans l'arbre.

Pendant le déroulement de l'algorithme, l'ensemble D de `ACM-GENERIQUE` est conservé implicitement sous la forme

$D = \{(v, pqrent[v]) : v \in V - \{r\} - F\}$.

Quand l'algorithme se termine la file de priorite min F est vide et l'arbre couvrant minimum de G est donc

$D = \{(v, parent[v]) : v \in V - \{r\}\}$.

```
ACM-PRIM(G, w, r)
1 pour chaque sommet u de V
2    faire cle[u] ← infini
3         p[u] ← NIL
4 cle[r] ← 0
5 F ← V
6 tantque F est non vide
7.    faire u ← EXTRAIRE-MIN(F)
8        pour chaque sommet v de Adj[u]
9.              faire si v appartient a F et w(u,v) < cle[v]
10                     alors p[v] ← u
11                     cle[v] ← w(u, v)
```

Les lignes 1-5 : On initialisent la clé de chaque sommet à ∞ (sauf pour la racine r, dont la clé est initialisée à 0 pour qu'elle soit le premier sommet traité). De même, ces lignes initialisent le parent $p[v]$ de chaque sommet v à `NIL`. A la ligne 5, on initialise la file de priorités min F de telle sorte qu'elle contienne tous les sommets.

L'algorithme conserve un invariant de boucle composé de trois parties. Cet invariant de boucle est : avant chaque itération de la boucle tant que des lignes 6–11, on a :

- $D = \{(v, p[v]) : v \in V - \{r\} - F\}$.
- Les sommets déjà placés dans l'arbre couvrant minimum sont ceux de $V - F$.
- Pour tous les sommets $v \in F$, si $p[v] \neq NIL$, alors $cle[v] < \infty$ et $cle[v]$ est le poids d'une arête minimale $(v, p[v])$ qui relie v à un certain sommet déjà placé dans l'arbre couvrant minimum.

La ligne 7 identifie un sommet u qui est incident pour une arête minimale traversant la coupe $(V - F, F)$ (sauf dans la première itération, où $u = r$ à cause de la ligne 4). Supprimer u de l'ensemble F a pour effet de l'ajouter à l'ensemble $V - F$ des sommets de l'arbre, et donc d'ajouter $(u, p[u])$ à D.

La boucle pour des lignes 8 - 11 met à jour les champs clé et p de chaque sommet v adjacent à u mais pas dans l'arbre. Cette mise à jour conserve la troisième partie de l'invariant.

Les performances de l'algorithme de `Prim` dépendent de la manière dont la file de priorités min F est implémentée. Si F est implémentée comme un tas min binaire, on peut se servir de la procédure `CONSTRUIRE-TAS-MIN` pour effectuer l'initialisation des lignes 1–5 en temps $O(V)$. Le corps de la boucle tant que est exécuté $|V|$ fois et comme chaque opération `EXTRAIRE-MIN` prend un temps $O(\log V)$, le temps total requis par tous les appels à `EXTRAIRE-MIN` est $O(V \log V)$.

La boucle pour des lignes 8–11 est exécutée en tout $O(E)$ fois, puisque la somme des longueurs de toutes les listes d'adjacences vaut 2 $|E|$. A l'intérieur de la boucle pour, le test d'appartenance à F de la ligne 9 peut être implémenté en temps constant, via attribution à chaque sommet d'un bit indiquant s'il appartient ou non à F et mise à jour de ce bit quand le sommet est supprimé de F.

L'affectation en ligne 11 implique une opération `DIMINUER-CLE` implicite sur le tas min, qu'on peut mplémenter dans un tas min binaire en $O(\log V)$. Donc, le temps total de l'algorithme de `Prim` est $O(V \log V + E \log V) = O(E \log V)$, qui est asymptotiquement le même que pour notre implémentation de l'algorithme de `Kruskal`.

Cependant, le temps d'exécution asymptotique de l'algorithme de `Prim` peut être amélioré en utilisant des tas de `Fibonacci`. Le chapitre 20 montre que si $|V|$ éléments sont organisés en tas de Fibonacci, on peut effectuer une opération EXTRAIRE-MIN dans un temps amorti $O(\log V)$ et une opération DIMINUER-CLÉ (pour implémenter la ligne 11) dans un temps amorti $O(1)$. En conséquence, si l'on se sert d'un tas de Fibonacci pour implémenter la file de priorités min F, le temps d'exécution de l'algorithme de Prim est amélioré pour devenir $O(E + V \log V)$.