

Révisions et ajouts aux notes de cours sur les algorithmes voraces

4.1 Arbres minimum de recouvrement

Étant donné un graphe non orienté valué $G = (V, E)$ (voir Figure 4.3 du manuel, p. 139), un *arbre minimum de recouvrement* (*minimum spanning tree*) est un sous-graphe connexe de G qui contient tous les sommets de G , sous-graphe qui est aussi un arbre (graphe connexe sans cycle) et dont la somme des pondérations des arêtes est minimale.

```
PROCEDURE arbreRecouvrement( G: Graphe ): Graphe
PRECONDITION
  G = (V, E)
  G est un graphe connexe
DEBUT
  F <- {}
  solutionTrouvee <- FAUX
  TANTQUE NON solutionTrouvee FAIRE
    e <- sélectionner une arête e qui semble "intéressante"
    SI l'ajout de e à F ne crée pas de cycle ALORS
      F <- F U {e}
      SI (V, F) est un arbre de recouvrement pour G ALORS
        solutionTrouvee <- VRAI
      FIN
    FIN
  FIN
  RETOURNER (V, F)
FIN
```

Figure 1: Forme générale d'une solution vorace au problème de l'arbre de recouvrement minimum

Une solution vorace pour ce problème aurait l'allure du pseudo-code présenté à la Figure 1, où l'on cherche F tel que $T = (V, F)$ soit un arbre de recouvrement minimal pour G (donc $F \subseteq E$).

Propriété cruciale d'un arbre de recouvrement minimal

Avant d'examiner plus en détails deux algorithmes différents pour le calcul d'un arbre de recouvrement minimal, il est intéressant de présenter une propriété des arbres de recouvrement qui permet de mieux comprendre et justifier pourquoi ces algorithmes fonctionnent correctement (adapté de M.T. Goodrich et R. Tamassia, "*Data Structures and Algorithms in Java*", John Wiley & Sons, 1998). Cette propriété est illustrée à la Figure 2.

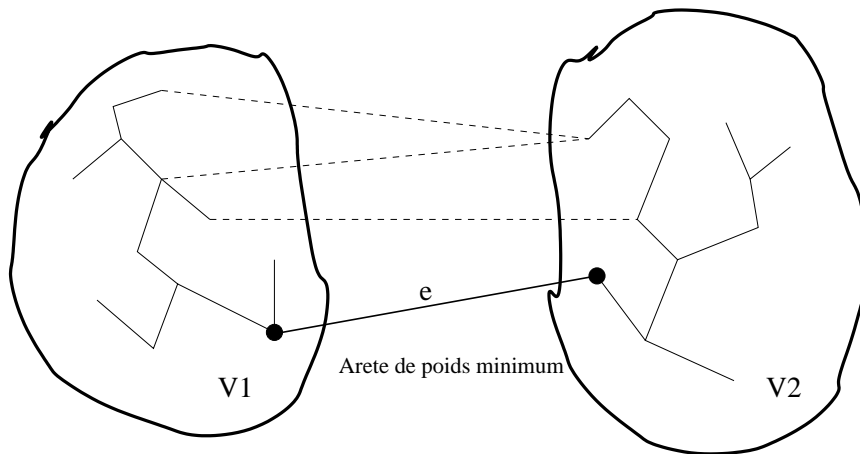


Figure 2: Une illustration de la propriété cruciale d'un arbre de recouvrement

Proposition : Soit $G = (V, E)$ un graphe non orienté, pondéré et connexe. Soit V_1 et V_2 deux ensembles disjoints non vides tels que $V = V_1 \cup V_2$. Soit e une arête dans G de poids minimum parmi toutes celles qui relient un sommet dans V_1 à un sommet dans V_2 . Alors, il existe un arbre de recouvrement minimum T dont l'une des arêtes est e .

Justification : (Preuve par contradiction) Supposons que la proposition soit fautive, donc qu'il n'existe aucun arbre de recouvrement minimum qui contient l'arête e (l'arête non pointillée reliant V_1 et V_2 dans la Figure 2). Soit alors T un arbre de recouvrement minimum. Si on ajoute l'arête e aux arêtes de T , un cycle sera nécessairement créé. Puisque l'ajout de e créerait un cycle, il existe donc une arête f de l'arbre T qui relie un sommet de V_1 à un sommet de V_2 (l'une des arêtes pointillées dans la Figure 2). De plus, on a que le poids de e est l'arête de poids minimum parmi toutes celles reliant V_1 et V_2 , c'est-à-dire, $\text{poids}(e) \leq \text{poids}(f)$. Si on supprime f de T en ajoutant e , alors on aura quand même un arbre de recouvrement, et son poids *ne sera pas supérieur* à celui de T . Puisque T était un arbre de recouvrement minimum, ce nouvel arbre obtenu par le remplacement de f par e sera lui aussi un arbre de recouvrement minimum. Or, nous avons supposé qu'il n'existait pas de tel arbre contenant e . Notre hypothèse de départ était donc fautive. On peut donc conclure qu'il existe bien un arbre de recouvrement minimum qui contient e .

4.1.1 Algorithme de Prim

– Soit Y un ensemble de sommets provenant de $G = (V, E)$, G étant un graphe connexe. Le sommet le plus près de Y est *un* sommet $v' \in V - Y$ qui est relié à un sommet de Y par une arête de poids minimum.

Note : S'il n'existe aucune arête entre s et s' , on note alors le poids comme étant $+\infty$.

– Description de haut niveau de l'algorithme de Prim : Figure 3.

```

PROCEDURE arbreRecouvrement( G: Graphe ): Graphe
PRECONDITION
  G = (V, E)
  G est un graphe connexe
DEBUT
  F <- {}
  Y <- {v1} // Le choix du sommet de départ est arbitraire
  TANTQUE Y ≠ V FAIRE
    v' <- sélectionner un sommet dans V-Y qui est le plus près de Y
    Y <- Y ∪ {v'}
    F <- F ∪ {arête (de poids minimum) qui relie v' à Y}
  FIN
  RETOURNER (Y, F)
FIN

```

Figure 3: Description de haut niveau de l'algorithme de Prim

– Supposons que les poids des arêtes de G sont donnés par la matrice suivante :

$$W[i][j] = \begin{cases} \text{poids}(\{v_i, v_j\}) & \text{s'il existe une arête entre } v_i \text{ et } v_j \\ +\infty & \text{s'il n'existe aucune arête entre } v_i \text{ et } v_j \\ 0 & \text{si } i = j \end{cases}$$

L'algorithme 4.1 du manuel (présenté aux pages 144–145) permet alors de déterminer un arbre de recouvrement minimum (le choix du sommet de départ v_1 est arbitraire). Les structures de données utilisées par l'algorithme jouent les rôles suivants :

- **nearest**[i] = indice du sommet dans Y le plus près de v_i .

Puisque Y varie en cours d'exécution, au fur et à mesure où on y ajoute des sommets, ce tableau sera donc lui aussi mis à jour, à chaque fois en fonction du nouvel élément ajouté à Y.

Notons aussi que, initialement, **nearest**[i] = 1 pour tous les $i \geq 2$, puisque Y ne contient initialement que le sommet v_1 . Soulignons que le choix du sommet initial n'a pas d'importance : ultimement, pour que l'on ait un arbre de recouvrement, tous les sommets devront nécessairement être inclus.

- **distance**[i] = poids sur l'arête qui relie v_i à l'arête indexée par **nearest**[i] (le sommet de Y le plus près du sommet v_i).

Cette distance va évidemment être mise à jour en fonction de **nearest**[i], c'est-à-dire, en fonction de l'ajout d'un nouvel élément dans Y.

Donc, au fur et à mesure que de nouvelles arêtes sont ajoutées à Y, les tableaux **nearest** et **distance** sont mis à jour en fonction du nouveau sommet ajouté dans Y. À chaque itération de l'algorithme, il faudra donc déterminer l'index pour lequel **distance**[i] est

minimum, ce qui nous permettra de déterminer le sommet le plus près de Y . C'est l'index de ce sommet qui est conservé dans la variable \mathbf{vnear} .

Initialement, $Y = \{v_1\}$, donc ces structures de données sont initialisées comme suit :

- $\mathbf{nearest}[i] = 1$.
- $\mathbf{distance}[i] =$ poids de l'arête qui relie v_1 à v_i (peut être $+\infty$ s'il n'y a pas une telle arête).

– Explications plus détaillées de l'algorithme de Prim (pp. 144–145 du manuel) :

- La boucle **for** au début de l'algorithme (p. 144) initialise **nearest** et **distance** pour Y ne contenant que v_1 .
- La première boucle **for** à l'intérieur de la boucle **repeat** (p. 145) détermine l'arête qui est la plus près de Y . La condition $0 \leq \mathbf{distance}[i]$ est utilisée pour ne pas examiner les sommets qui sont déjà présents dans Y (la **distance** d'un tel sommet par rapport à Y est définie, juste après l'ajout de l'arête e à F , comme étant -1).
- La deuxième boucle **for** met à jour les tableaux **nearest** et **distance**. On examine chacun des sommets qui n'est pas déjà dans Y (si le sommet est dans Y , alors $\mathbf{distance}[i] = -1$, donc l'instruction **if** n'est pas exécutée) : si la distance de ce sommet à Y est inférieure à ce qu'elle était avant l'ajout du sommet sélectionné (\mathbf{vnear}), alors on modifie **distance**, en indiquant que le sommet de Y qui est maintenant le plus près est bien le nouveau sommet ($\mathbf{nearest}[i] = \mathbf{vnear}$);

– Complexité de l'algorithme pour $G = (V, E)$: $\Theta(n^2)$, où $n = |V|$ (nombre de sommets).

– Preuve que l'algorithme de Prim produit un arbre optimal : Voir Lemme 4.1, Figure 4.6 (p. 146) et Théorème 4.1 (p. 147).

La preuve repose sur la notion de sous-ensemble *prometteur* (*promising*) : un sous-ensemble d'arêtes est prometteur si on peut lui ajouter des arêtes de façon à obtenir un arbre minimum de recouvrement. Tout d'abord, le lemme nous dit qu'un ensemble prometteur d'arêtes peut être étendu avec une autre arête de façon à obtenir un nouvel ensemble prometteur si on choisit une arête de poids minimum qui relie un sommet de Y à un sommet de $V - Y$, Y étant défini comme l'ensemble des sommets référés dans l'ensemble prometteur. Ensuite, il suffit de montrer par induction (Théorème 4.1) que l'ensemble F de l'algorithme est toujours prometteur.

Toutefois, sans entrer dans les détails formels de la preuve, la propriété présentée à la page 4 nous donne une bonne intuition sur pourquoi l'algorithme produit bien un arbre de recouvrement minimum.

4.1.2 Algorithme de Kruskal

L'approche utilisée par l'algorithme de Kruskal est différente de celle utilisée par l'algorithme de Prim. Dans l'algorithme de Prim, on étend petit à petit l'arbre à partir d'un sommet arbitraire. À une étape donnée de l'algorithme, on a donc un ensemble de sommets reliés entre eux qui font partie de l'arbre final, et un ensemble de sommets encore isolés. Par contre, au cours de l'exécution de l'algorithme de Kruskal, on aura plutôt une *forêt* (une collection d'arbres) et, à chaque étape de l'exécution de l'algorithme, on réduira la taille de cette forêt en reliant ensemble deux des sous-arbres déjà identifiés.

Plus précisément, dans l'algorithme de Kruskal, on commence par définir une collection de sous-ensembles disjoints de sommets (d'éléments de V). On inspecte ensuite chacune des arêtes, en ordre non décroissant de poids. Si l'arête sélectionnée relie deux sommets dans des sous-ensembles encore disjoints, alors on l'ajoute aux arêtes de l'arbre de recouvrement et on fusionne les deux sous-ensembles, puisque tous les sommets de ces deux sous-ensembles sont maintenant reliés entre eux. En d'autres mots, un sous-ensemble de sommets dénote donc un groupe de sommets qui sont actuellement reliés entre eux dans la forêt qui formera l'arbre final (une classe d'équivalence, donc). C'est pour cela que si l'arête de poids minimum relie deux sommets qui font déjà partie du même sous-ensemble, alors cette arête doit être rejetée (ne fera pas partie de l'arbre final) car son ajout créerait un cycle et on n'obtiendrait pas un arbre.

```
PROCEDURE arbreRecouvrement( G: Graphe ): Graphe
PRECONDITION
  G = (V, E)
  G est un graphe connexe
DEBUT
  F <- {}
  // On notera S = {S1, ..., Sk}
  S <- { {vi} | 1 ≤ i ≤ |V| } // On crée un sous-ensemble distinct pour chaque sommet
  E' <- séquence ordonnée (non décroissante) des arêtes provenant de E
  TANTQUE |S| ≠ 1 FAIRE
    e <- head E' // L'arête de poids minimum
    E' <- tail E'
    SI l'arête e relie deux sommets entre Si et Sj avec i ≠ j ALORS
      // On fusionne les sous-ensembles
      S <- S - {Si, Sj} U {Si U Sj}
      F <- F U {e}
  FIN
  FIN
  Soit S = {S0}
  RETOURNER (S0, F)
FIN
```

Figure 4: Description de haut niveau de l'algorithme de Kruskal

– Description de haut niveau de l'algorithme de Kruskal : Figure 4.

Une partie clé de cet algorithme est la suivante :

```
SI l'arête  $e$  relie deux sommets entre  $S_i$  et  $S_j$  avec  $i \neq j$  ALORS
  // On fusionne les sous-ensembles
   $S \leftarrow S - \{S_i, S_j\} \cup \{S_i \cup S_j\}$ 
   $F \leftarrow F \cup \{e\}$ 
FIN
```

La tâche effectuée par cette partie de l'algorithme consiste à identifier les sous-ensembles S_i et S_j qui contiennent, respectivement, les sommets v_i et v_j associés à l'arête e de poids minimum sélectionnée à l'étape qui précède de l'algorithme. Cette partie de l'algorithme pourrait donc s'exprimer de la façon suivante, en sachant que e relie les sommets d'index i et j :

```
Trouver l'ensemble  $S_i$  qui contient le sommet  $i$ 
Trouver l'ensemble  $S_j$  qui contient le sommet  $j$ 
SI  $S_i \neq S_j$  ALORS
  Fusionner  $S_i$  et  $S_j$  en un seul ensemble
   $F \leftarrow F \cup \{e\}$ 
FIN
```

Note : Dans l'algorithme de la Figure 4, `head s` retourne l'élément à la tête de s , c'est-à-dire le premier élément de s (donc `head s = s[1]`), alors que `tail s` retourne la sous-séquence qui contient les mêmes éléments que s sauf le premier élément (donc `tail s = s[2..length(s)]`). On a donc toujours que $s = [\text{head } s] ++ \text{tail } s$, ce qui peut aussi s'exprimer par l'équation suivante : $s = \text{add}(\text{head } s, \text{tail } s)$.

– L'algorithme 4.2 du manuel (présenté aux pages 149–150) permet de déterminer un arbre de recouvrement minimum utilisant l'algorithme de Kruskal. L'algorithme utilise une structure de données permettant de manipuler efficacement des sous-ensembles disjoints d'indices. Ces opérations sont décrites un peu plus bas (Section 4.5).

– Complexité de l'algorithme pour $G = (V, E)$, où $|V| = n$ et $|E| = m$:

1. Temps pour effectuer le tri des arêtes : $\Theta(m \lg m)$.

Note : Comme dans l'exemple du sac à dos, on pourrait aussi utiliser une file de priorité plutôt qu'un tri complet. La complexité asymptotique résultante serait la même, mais le facteur de proportionalité serait inférieur.

2. Initialisation des n ensembles disjoints à l'aide des opérations de manipulation de sous-ensembles disjoints : $\Theta(n)$.
3. Temps à l'intérieur de la boucle **TANTQUE** : on a m itérations, et à chaque itération on utilise les opérations `find`, `equal` ou `merge` de manipulation de sous-ensembles disjoints. La complexité de cette partie sera donc $\Theta(m \lg^* m)$ (voir plus bas, Section 4.5, p. 15 : la complexité est indépendante du nombre d'items dans l'ensemble, mais dépend uniquement du nombre d'opérations effectuées).

Parce que $m \geq n-1$, c'est donc le tri qui domine le temps d'exécution (notons qu'on aurait le même résultat si on utilisait une file de priorité). Donc, $W(m, n) = \Theta(m \lg m)$. Toutefois, dans le pire cas, le nombre d'arêtes $m \in \Theta(n^2)$ (graphe complet = chaque sommet est relié à chacun des autres sommets). Le pire cas peut donc aussi être exprimé par $W(m, n) \in \Theta(n^2 \lg n^2) = \Theta(n^2 \lg n)$.

– Preuve que l'algorithme de Prim produit un arbre optimal : Voir Lemme 4.2 (p. 151) et Théorème 4.2 (p. 152) : encore basé sur la notion d'ensemble prometteur, avec preuve par induction sur les diverses itérations.

4.1.3 Comparaison entre l'algorithme de Prim et celui de Kruskal

Algorithme de Prim : $T(n) \in \Theta(n^2)$

Algorithme de Kruskal : $W(m, n) \in \Theta(m \lg m)$ et $W(m, n) \in \Theta(n^2 \lg n)$

Or, dans un graphe connexe, on a toujours la propriété suivante :

$$n-1 \leq m \leq \frac{n(n-1)}{2}$$

Donc :

- Si m est “petit”, alors l'algorithme de Kruskal sera préférable.
- Si m est “grand”, alors l'algorithme de Prim sera préférable.

4.5 Manipulation de sous-ensembles disjoints (relations d'équivalence)

Certains algorithmes, par exemple, l'algorithme de Kruskal, demandent de pouvoir manipuler les sous-ensembles disjoints d'un ensemble de base. En d'autres mots, étant donné un ensemble de départ $U = \{u_1, \dots, u_n\}$, on veut pouvoir manipuler efficacement un ensemble de sous-ensemble U_1, \dots, U_k tels que les propriétés suivantes sont satisfaites (propriétés qui définissent, rappelons-le, une relation d'équivalence entre les éléments) :

$$\bigcup_{i=1}^k U_i = U$$
$$\bigcap_{i=1}^k U_i = \{\}$$

De plus, on veut aussi être capable d'effectuer les opérations suivantes de façon efficace :

- Recherche : Étant donné un élément u_j , on veut trouver l'ensemble U_i auquel il appartient (identification de la classe d'équivalence d'un élément).
- Fusion : Étant donné deux sous-ensembles U_i et U_j , on veut les fusionner en un seul sous-ensemble (formation d'une nouvelle classe d'équivalence).
- Comparaison : Étant donné deux sous-ensembles U_i et U_j , on veut pouvoir déterminer de façon efficace (temps constant) si ces deux sous-ensembles sont égaux (s'ils dénotent la même classe d'équivalence).
- Initialisation : Au départ, on veut faire en sorte que $U_i = \{u_i\}$ (chaque élément est dans sa propre classe d'équivalence).

Dans un tel type de données, les opérations clés ont donc l'allure suivante (version du manuel, décrite plus en détail à l'annexe C) :

- *initial*(n) : crée n sous-ensembles disjoints, chacun contenant une valeur comprise entre 1 et n (représentant donc les divers $U_i = \{u_i\}$).
- $p = \textit{find}(i)$: p réfère à l'ensemble contenant l'élément d'index i (c'est-à-dire, la classe d'équivalence pour l'élément u_i).
- *merge*(p, q) : fusionne les deux ensembles en un seul sous-ensemble. Après l'exécution, p et q réfèrent alors au même sous-ensemble.
- *equal*(p, q) : retourne **VRAI** si et seulement si p et q réfère au même sous-ensemble (dénotent la même classe d'équivalence).

La mise en oeuvre décrite dans l'annexe C du manuel utilise une structure de données appelée une *forêt d'arbres inversées* (un enfant pointe vers son parent, la racine pointant simplement vers elle-même). Une telle structure peut être réalisée à l'aide d'un tableau d'items, donc pas nécessairement avec allocation dynamique et pointeurs (comme pour un monceau, sauf que la structure, l'organisation des items est différente).

Les détails de la mise en oeuvre d'une telle structure de données dépassent le cadre du présent cours (voir l'exemple présenté en classe ; une présentation plus détaillée serait

plutôt le sujet du cours INF7341 Structures de données). On peut toutefois mentionner qu'une mise en oeuvre naïve peut conduire à des résultats tels qu'une opération de recherche soit de temps linéaire. Par contre, en s'assurant par diverses techniques (fusion qui tient compte de la taille des ensembles à fusionner et compression des chemins lors d'une recherche) de minimiser le plus possible la hauteur des arbres inversés manipulés, on peut en arriver à une mise en oeuvre plus efficace. Plus précisément, pour une série de m opérations *equal*, *find* ou *merge*, la complexité sera $O(m \lg^* m)$, où la fonction $\lg^* m$ est définie comme suit :

$$\begin{aligned} \lg^* m &= \min\{i: t(i) \geq m\} \\ t(i) &= \begin{cases} 1 & \text{si } i = 0 \\ 2^{t(i-1)} & \text{si } i \geq 1 \end{cases} \end{aligned}$$

L'analyse détaillée de la complexité de cette structure de données et des opérations associées requiert l'utilisation d'une technique appelée *analyse amortie*. Dans une telle forme d'analyse, on s'intéresse non pas strictement à la complexité d'une opération donnée, mais plutôt à la complexité d'une série d'opérations. Intuitivement, ceci signifie qu'on peut permettre qu'une opération donnée soit un peu plus coûteuse si on est assuré que ce coût additionnel pourra être *amorti* sur l'ensemble des opérations. Dans le cas présent, la borne asymptotique ne porte donc pas sur un appel spécifique à une opération donnée, mais porte plutôt sur le coût pour exécuter (après l'appel à *initial*) une série de m opérations *equal*, *find* ou *merge*, la borne étant alors de $O(m \lg^* m)$.

Note : Dans le livre de Cormen, Leiserson et Rivest, la borne mentionnée est plutôt $O(m \lg^* n)$. Toutefois, la construction de l'ensemble initial se fait plutôt à l'aide de n appels à une fonction d'initialisation pour créer un ensemble singleton (plutôt qu'avec un unique appel à *initial*) et le nombre m d'opérations inclut aussi ces n appels d'initialisation de l'ensemble. Le résultat est donc équivalent. Notons aussi que l'analyse de la version plus efficace demande plusieurs pages (8) d'analyse mathématique détaillée, et ce *après* que les notions de base de l'analyse amortie aient déjà été présentées dans un chapitre précédent . . .

Note : Dans le manuel (appendice C), la borne mentionnée est plutôt $O(m \lg m)$. Le dernier paragraphe de l'annexe C mentionne toutefois l'existence de la technique de *path compression*, qui permet d'obtenir une mise en oeuvre qui soit presque de temps linéaire en m (le nombre d'opérations). Cette mise en oeuvre avec *path compression* est bien celle mentionnée dans les paragraphes qui précèdent. Pour obtenir la borne moins stricte $O(m \lg m)$, il suffit simplement de fusionner les arbres inversés de façon à minimiser la hauteur de l'arbre résultant, ce qui se fait simplement en faisant pointer la racine du plus petit arbre vers la racine du plus grand.