IFT2015 Hiver 2009 — arbre couvrant minimal

Miklós Csűrös

9 avril 2009

1 Définitions

Soit $G = (V, E)$ un graphe dans lequel les arêtes sont ponderées par la fonction $w\colon E \mapsto (0, \infty)$. Un *arbre couvrant* est un sous-graphe $T = (V, E')$ avec $E' \subseteq E$ qui est un arbre[1] :

- ⋆ T est connexe
- ⋆ T ne contient pas de cycle
- ⋆ $|E'| = |V| - 1$

Son poids est $w(T) = \sum_{e \in E'} w(e)$. Un *arbre couvrant minimal* (ACM) est un arbre couvrant dont le poids est minimal parmi tous les arbres couvrants : voir Figure 1 pour un exemple. Notez qu'on peut avoir plus qu'un arbre couvrant minimal mais ils ont ous le même poids.

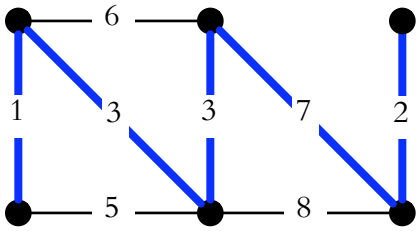

FIG. 1 – Un arbre couvrant minimal est montré par les arêtes bleues (grasses). Le poids de cet arbre est $1 + 3 + 3 + 7 + 2 = 16$.

2 Algorithmes

Il existe plusieurs façons dont on peut construire un arbre couvrant minimal. Dans les algorithmes mentionnés ici, on choisit les arêtes à inclure dans T par itération. En chaque itération, on peut appliquer la «règle bleue»[2].

[1] Notez que l'«arbre» est l'abstraction mathématique et non pas la structure de données !

[2] Par tradition, on colorie les arêtes choisies dans T par bleue, et celles qui sont excluses par rouge. Ainsi, les algorithmes ACM sont présentés comme des algorithmes de coloriage d'arêtes.

Règle bleue. *Ajouter une arête avec poids minimal qui ne produit pas de cycle avec les arêtes déjà choisies.*

Les algorithmes de Kruskal et de Prim appliquent la règle bleue itérativement : ils diffèrent dans l'identification d'arêtes qui satisfont les contraintes («poids minimal» et «pas de cycle»).

2.1 Algorithme de Kruskal

L'algorithme de Kruskal peut être esquissé comme suit.

```
K1  KRUSKAL // esquissé
K2  initialiser T ← (V, ∅) // tous les sommets, pas d'arêtes
K3  trier les arêtes par poids croissant : mettre le résultat dans le tableau E[1 . . . m]
K4  pour i ← 1 . . . m
K5      soit l'arête uv ← E[i]
K6      si uv ne forme pas de cycle dans T
K7      alors ajouter l'arête uv à T
```

Pour implanter l'algorithme, on a besoin d'une structures de données qui permet l'exécution rapide. Pour déterminer si l'arête uv forme un cycle avec des arêtes déjà choisies (ligne K6), on maintient une structure de connexité entre des sommets : on a un cycle si et seulement si u et v sont déjà liés par des arêtes choisies. La structure UNION-FIND est donc le choix parfait pour cette tâche (v. les acétates intro.pdf, page vii et après).

```
K1  KRUSKAL // avec UNION-FIND
K2  initialiser T ← (V, ∅) ; pour u ∈ V faire makeset(u)
K3  trier les arêtes par poids croissant : mettre le résultat dans le tableau E[1 . . . m]
K4  pour i ← 1 . . . m        // (on peut arrêter plus tôt dès que T contient (n − 1) arêtes)
K5      soit l'arête uv ← E[i]
K6      si find(u) ≠ find(v)
K7      alors ajouter l'arête uv à T ; union(u, v)
```

Temps de calcul. Soit n le nombre de sommets et m le nombre d'arêtes. Le tri de la ligne K2 prend un temps $O(m \log m)$ si le graphe est representé par des listes d'adjacence (avec la matrice d'adjacence, il faut extraire la liste des arêtes avant de trier qui prend $O(n^2)$). Pendant l'exécution de l'algorithme, on fait tout au plus $2m$ appels à find et $(n-1)$ appels à union. Le coût amorti de ces opérations dans une série de $2m + n - 1 \leq 3m$ appels est $O(\alpha(3m, n)) = O(\alpha(m, n))$ où $\alpha(m, n)$ est la fonction d'Ackerman inverse de croissance «pratiquement imperceptible». En total (boucle+initialisation), on a donc

$$O(m \log m) + 3m \cdot O(\alpha(m, n)) = O(m \log m) + O(m\alpha(m, n)) = O(m \log m).$$

2.2 Algorithme de Prim

Dans l'algorithme de Kruskal, T est un forêt (ensemble d'arbres) dans un état intermédiaire. Dans l'algorithme de Prim, T est toujours un arbre, on ajoute un sommet à la fois. On construit l'ACM à partir d'un *sommet de départ* s : il n'est pas important où on commence. À chaque itération, on ajoute une arête en appliquant la règle bleue.

```
P1  PRIM(s) // esquissé
P2  initialiser T ← ({s}, ∅)
P3  tandis que T ne contient tous les sommets
P4      soit uv une arête avec u ∈ T, v ∉ T et w(uv) minimal
P5      ajouter uv à T
```

Comme implantation naïve, on peut juste parcourir toute les arêtes pour choisir uv, mais cela prend $\Theta(m)$ à chaque itération qui mène à un temps de calcul $\Theta(nm)$ ce qui est beaucoup pire que celui de l'algorithme de Kruskal. On a donc besoin d'une structure de données qui permet la détermination rapide de uv en ligne P4. La solution est d'utiliser une file à priorités : la file contient les sommets $v \notin T$ à chaque itération. La priorité de v est le coût minimal $\min_{u \in T} w(uv)$ (si $uv \notin E$, on suppose $w(uv) = \infty$) de l'ajouter à T. Ainsi, on identifie uv en ligne P4 par l'opération deleteMin. En ligne P5, il faut vérifier si pour un $x \notin T$, vx donne maintenant un lien à coût inferieur qu'avant. Si oui, on doit ajuster la priorité de x. Dans l'algorithme ci-dessous, $\pi(v)$ dénote la priorité de $v \notin T$. Pour chaque tel v, il faut aussi stocker le sommet $\mathsf{lien}(v) = u \in T$ avec lequel $\pi(v) = w(uv)$.

```
P1  PRIM(s) // avec une file à priorités
P2  T ← ({s}, ∅) ; pour tout u ≠ s faire { π(u) ← w(su); lien(u) ← s }
P2a initialiser la file
P3  tandis que T ne contient tous les sommets
P4      v ← deleteMin() ; u ← f(v)
P5      ajouter uv à T
P6      pour tout x ∉ T  // parcourir la liste d'adjacence x ∈ Adj[v]
P7          si w(vx) < π(x) alors π(v) ← w(vx) ; lien(x) ← v
```

Temps de calcul. Le temps de calcul dépend de la structure de données qu'on choisit pour implanter la file à priorités dans l'algorithme. Avec un **tas binaire**, l'initialisation en ligne P2a est l'opération heapify qui prend un temps $O(n)$. L'opération deleteMin en ligne P5 prend $O(\log n)$. La ligne P7 performe l'opération appellée decreaseKey ($\pi(v) \leftarrow w(vx)$) qui change la priorité d'un élément sur le tas : il faut juste appeler NAGER après l'affectation de priorité pour assurer que le tas reste correct. Donc ceci prend $O(\log n)$ temps aussi. En total, on a l'initialisation du tas, $(n-1)$ opérations deleteMin et $\leq m$ opérations decreaseKey (car ligne P7 est executée une fois tout au plus pour chaque arête). Ça donne un temps de calcul borné par

$$\underbrace{O(n)}_{\text{heapify}} + (n-1)\underbrace{O(\log n)}_{\text{deleteMin()}} + m\underbrace{O(\log n)}_{\text{decreaseKey}} = O(m \log n).$$

Avec un **tas d-aire**, deleteMin prend $O(d \log_d n)$. L'opération decreaseKey, implantée à l'aide de NAGER prend un temps $O(\log_d n)$. Dans ce cas-ci, on a donc un temps de calcul

$$\underbrace{O(n)}_{\text{heapify}} + \underbrace{O(nd \log_d n)}_{\text{deleteMin}()} + \underbrace{O(m \log_d n)}_{\text{decreaseKey}} = O\big((nd+m)\log_d n\big) = O\left(\frac{nd+m}{\log d} \log n\right).$$

L'avantage d'un tas d-aire est qu'on peut ajuster d selon les dimensions du graphe (n, m). Un bon choix est $d = \lceil 2 + m/n \rceil$.

Il existe d'autre structure de données pour files à priorités, où decreaseKey est plus rapide. Avec un **tas Fibonacci** par exemple, le coût amorti de decreaseKey est $O(1)$, donc on peut exécuter l'algorithme de Prim en un temps de

$$O(n \log n + m).$$