10 Structures de données en action

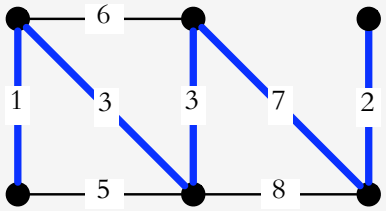
10.1 Définitions

Définition 10.1. *Un **graphe non-orienté** est un couple (V, E) où $E \subseteq \binom{V}{2}$ (paires non-ordonnées). V est l'ensemble des sommets et E est l'ensemble des arêtes.*

Définition 10.2. *Un **chemin** de longueur ℓ est une séquence $v_0, v_1, \ldots, v_\ell$ où $v_{i-1}v_i \in E$ pour tout $i = 1, \ldots, \ell$. ($\ell = 0$ est OK : c'est un chemin d'un seul sommet sans arêtes.) Si $v_0 = v_\ell$, alors le chemin forme un **cycle**. (Plus précisement, les sommets initial et final ne sont pas distingués dans le cycle.)*

Définition 10.3. *Une **coupure** (X, X') d'un graphe non-orienté $G = (V, E)$ est une partition de ses sommets : $X \in V, X' = V \setminus X$. L'arête uv* traverse la coupure *(ou «appartient à la coupure») si $u \in X$ et $v \in X'$.*

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 cycles, et $|E'| = |V| - 1$). Son poids est $w(T) = \sum_{e \in E'} w(e)$. Un *arbre couvrant minimal* (ACM) est un arbre couvrant dont le poids atteint le minimum.Notez qu'on peut avoir plus qu'un arbre couvrant minimal mais ils ont tous le même poids. W(fr)



Les arêtes bleues (grasses) forment un arbre couvrant minimal. Le poids de l'arbre est $1 + 3 + 3 + 7 + 2 = 16$.

10.2 Représentation de graphes

Matrice d'adjacence. C'est une matrice $V \times V$, $A[u, v]$ donne le poids de uv, ou une valeur booléenne pour stocker juste présence.

Listes d'adjacence. C'est un ensemble de listes $\mathsf{Adj}[u]$ pour chaque sommet u qui stocke l'ensemble $\{v\colon uv \in E\}$ ou l'ensemble des couples $\{\langle v, w(u, v)\rangle\colon uv \in E\}$. Usage de mémoire : $\Theta(|E| + |V|)$, et c'est meilleur que la matrice quand $E = o(|V|^2)$ (graphe *éparse*). W(en)

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

Déterminer si $uv \in E$ ou le poids $w(u, v)$: rapide avec la matrice mais plus lente avec les listes d'adjacence.

10.3 Arbre couvrant minimal

Il y a beaucoup de façons de 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].

> **Règle bleue.** *Choisir une copure sans aucune arête bleue. Choisir une arête non-coloriée avec le poids minimum qui traverse la coupure et la colorier par bleue.*

Les algorithmes de Kruskal et de Prim appliquent la règle bleue itérativement : ils diffèrent dans l'identification d'arêtes qui satisfont les contraintes.

10.3.1 Algorithme de Kruskal

W(fr)

L'algorithme de Kruskal peut être esquissé comme suit.

```
K1  KRUSKAL // esquissé
K2  initialiser T ← (V, ∅) // tous les sommets, aucune arête
K3  trier les arêtes par poids croissant : mettre résultat dans E[0 . . . m − 1]
K4  pour 0 ← 0 . . . m − 1
K5      soit l'arête uv ← E[i]
K6      si uv ne crée 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.

```
K1  KRUSKAL // avec UNION-FIND
K2  initialiser T ← (V, ∅) ; initialiser union-find sur les sommets
K3  trier les arêtes par poids croissant : mettre le résultat dans E[0 . . . m − 1]
K4  for i ← 0 . . . m − 1
K5      soit l'arête uv ← E[i]
K6      if find(u) ≠ find(v)
K7      then ajouter l'arête uv à T ; union(u, v)
```

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

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

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

10.3.2 Algorithme de Prim

W(fr)

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.

```
1  PRIM(s) // esquissé
2  initialiser T ← ({s}, ∅)
3  tandis que T ne contient tous les sommets
4      soit uv une arête avec u ∈ T, v ∉ T et w(uv) minimal
5      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 4. 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 4 par l'opération deleteMin. En ligne 5, 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}, ∅)                                   // arbre avec s seulement au début
P3  for u ∈ V do π[u] ← ∞                           // priorité = distance du sommet
P4  π[s] ← 0 ; initialiser file de priorité H avec les sommets
P5  v ← H.deleteMin()
P6  while v ≠ null do
P7      π[v] ← −∞                                  // π[v] = −∞ si v ∈ T
P8      for x : vx ∈ E do if w(vx) < π[x] then     // parcourir la liste d'adjacence
P9          π[x] ← w(vx) ; lien[x] ← v ; H.decreaseKey(x, π[x])
P10     v ← H.deleteMin()                          // v = null quand file devient vide
P11     if v ≠ null then u ← lien[v] ; ajouter uv à T
```

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'opération deleteMin prend $O(\log n)$. La ligne P9 performe l'opération appellée decreaseKey qui change la priorité d'un élément sur le tas : il faut juste appeler SWIM 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 P9 est executée une fois tout au plus pour chaque arête). Ça donne un temps de calcul borné par

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

Avec un **tas d-aire**, deleteMin prend $O(d \log_d n)$. L'opération decreaseKey, implantée à l'aide de SWIM prend un temps $O(\log_d n)$. On a donc un temps de calcul

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

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 $O(n \log n + m)$

W(fr)

Le plus court chemin (Dijkstra). L'algorithme de Prim s'adapte immédiatement à la recherche du plus court chemin à partir d'un sommet s : utiliser $\pi[v] + w(vx)$ comme priorité en Ligne P9.

```
P9      if π[v] + w(vx) < π[x] then π[x] ← π[v] + w(vx) ; lien[x] ← v
```