9 Algorithmes et structures de données sur graphes

9.1 Graphes et leur représentation

Définition 9.1. *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 9.2. *Un **graphe orienté** est un couple (V, E) où $E \subseteq V \times V$ (paires ordonnées). V est l'ensemble des **nœuds** ou sommets, et E est l'ensemble des **arcs**.*

Matrice d'adjacence. C'est une matrice $V \times V$, où la cellule $A[u, v]$ contient information sur l'arête uv.

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

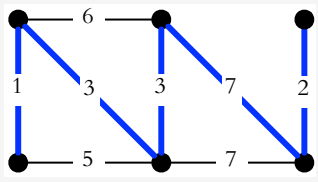
9.2 Graphe pondéré

Soit $w : E \mapsto [0, \infty]$ la *pondération des arêtes* du graphe $G = (V, E)$. Le poids d'une arête représente le coût de lier deux sommets dans beaucoup de problèmes d'optimisation ; on peut même considérer l'extension à toutes les paires avec $w(uv) = \infty$ pour tout $uv \notin E$. Le poids d'un ensemble d'arêtes est simplement la somme des poids : ici, on considère des problèmes de choisir un sous-ensemble d'arêtes formant un *chemin* ou un *arbre couvrant* avec poids minimal.

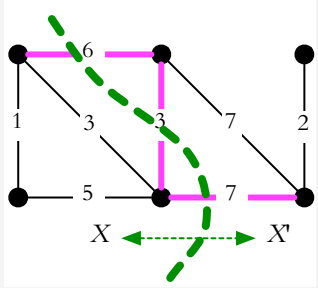
Définition 9.3. *Un **chemin** de longueur ℓ est une séquence de $\ell + 1$ sommets distincts $v_0, v_1, \ldots, v_\ell$ où $v_{i-1}v_i \in E$ pour tout $i = 1, \ldots, \ell$. ($\ell = 0$ est OK : c'est un chemin d'un seul sommet sans arêtes.) Le **plus court chemin** de sommet s à sommet t est le chemin de s à t avec poids minimal.*

Définition 9.4. *Un **arbre couvrant** est un sous-graphe $T = (V, E')$ avec $E' \subseteq E$ qui est connexe et ne contient pas de cycles. Un **arbre couvrant minimal** (ACM) est un arbre couvrant dont le poids atteint le minimum.* W(fr)

9.3 Arbre couvrant minimal



Les arêtes bleues (grasses) forment un arbre couvrant minimal. Le poids de l'arbre est $1 + 3 + 3 + 7 + 2 = 16$. Notez qu'on peut avoir plus qu'un arbre couvrant minimal mais ils ont tous le même poids.



Définition 9.5. *Une **coupure** (X, X') est une partition de sommets : $X \in V, X' = V \setminus X$. L'arête uv* traverse la coupure *(ou «appartient à la coupure») si $u \in X$ et $v \in X'$.*

Clairement, si (X, X') est une coupure quelconque, l'ACM T doit avoir au moins une de ses arêtes. En plus, une arête avec le plus petit poids dans la coupure doit être dans T (sinon, on peut remplacer une arête de T avec l'arête minimale). Les algorithmes voraces discutés ici «colorient» les arêtes de G un-à-un par bleue et rouge ; les arêtes bleues correspondent à un ACM à la fin. Pour choisir les arêtes, on utilise la «règle bleue».

> **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.*

Théorème 9.1. *Les arêtes bleues forment un ACM.*

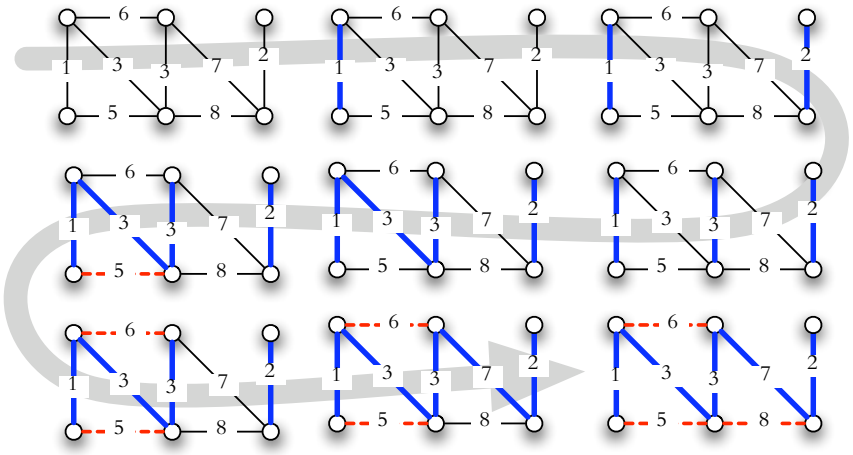
Démonstration. On démontre par induction que la proriété suivante vaut à toute application de la règle bleue : il existe un ACM T qui contient toutes les arêtes bleues. Au début, on n'a aucune arête bleue. Supposons (hypothèse d'induction) que la propriété vaut avant l'application de la règle avec un ACM T et la règle choisit une arête $e^* \notin T$ dans coupure C. Soit e^* l'arête choisie dans une coupure C, et soit $e' \in T$ l'arête de T qui traverse C. Considérons le remplacement de e' par e^* correspondant au sous-graphe $T^* = T - \{e'\} \cup \{e^*\}$. Maintenant, T^* est un arbre couvrant (reste connexe et acyclic) et $w(T^*) = w(T) - w(e') + w(e^*)$, d'où $w(e') \leq w(e^*)$ par la minimalité de T. Comme $w(e^*) = \min_{e \in C}\{w(e)\} \leq w(e')$, on a $w(e) = w(e^*)$ et $w(T^*) = w(T)$. En conséquence, la propriété vaut avec l'ACM T^*. ■

9.3.1 Algorithme de Kruskal

W(fr)

Dans l'algorithme de Kruskal, les arêtes bleues forment un forêt. À chaque itération, on colorie l'arête de poids minimal qui ne crée pas de cycle. On a un cycle si et seulement si u et v sont déjà liés par des arêtes choisies. En conséquence, il suffit de maintenir la connexité par les arêtes bleues dans une structure UNION-FIND.

```
K1  KRUSKAL(V, E)                          // ACM pour graphe avec |V| = n sommets et E| = m arêtes
K2  initialiser F ← ∅ ;                    // forêt d'arêtes bleues
K3  initialiser union-find pour n sommets
K4  trier les arêtes E[0 . . . m − 1] dans l'ordre croissant de poids w
K5  for i ← 0 . . . m − 1
K6      uv ← E[i]                          // prochaine arête
K7      if find(u) ≠ find(v)               // uv ne peut créer de cycle
K8      then F ← F ∪ {uv} ; union(u, v)    // colorier uv par bleue et stocker la connexion
```



Temps de calcul. Le tri de la ligne K4 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 ce qui prend $O(n^2)$). Pendant l'exécution de l'algorithme, on fait tout au plus $2m$ appels à find (Ligne K7) et $(n-1)$ appels à union (Ligne K8). En total (boucle+initialisation), on a donc

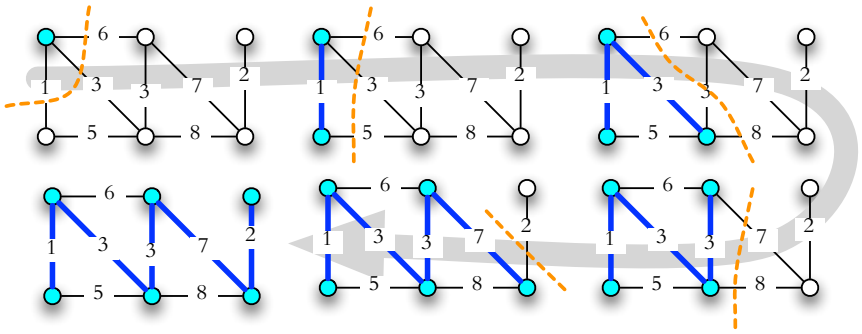
$$O(m \log m) + (2m + n - 1) \cdot O(\alpha(2m + n - 1, n)) = O(m \log m) + O(m\alpha(m, n)) = O(m \log m),$$

où $\alpha(m, n)$ est la fonction d'Ackerman inverse avec croissance «pratiquement imperceptible».

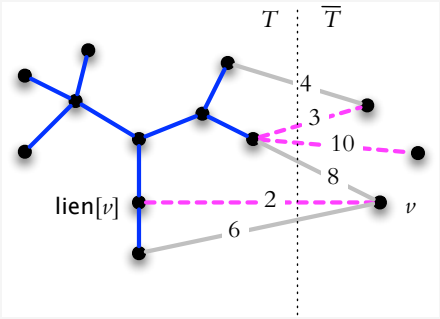
9.3.2 Algorithme de Prim

W(fr)

Dans l'algorithme de Kruskal, F est un forêt (ensemble d'arbres) dans un état intermédiaire. Dans l'algorithme de Prim, les arêtes bleues forment toujours un atbre. 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 à la coupure entre T et le reste des sommets.




```
1  PRIM(s) // esquissé
2  initialiser T ← ∅
3  tandis que T ne contient tous les sommets
4      soit uv une arête avec u ∈ T, v ∉ T et w(uv) minimal
5      T ← T ∪ {uv}
```



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 $\Theta(nm)$. 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 de priorité : 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 lien$[v] = u \in T$ avec lequel $\pi[v] = w(uv)$.

```
P1  PRIM(s) // avec une file à priorités
P2  T ← ∅                                              // T contient les arêtes bleues
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  do
P7      π[v] ← −∞                                      // π[v] = −∞ si v ∈ T
P8      for x : vx ∈ E do                              // parcourir la liste d'adjacence
P9          if w(vx) < π[x] then 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] ; T ← T ∪ {uv}
P12 while v ≠ null
```

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

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

Implantation du tas. En Ligne P9, on doit accéder à sommet x dans le tas. Pour cela, il faut maintenir un tableau d'indices sur le tas heapidx$[0..n-1]$ à l'addition des tableaux lien$[0..n-1]$ et $\pi[0..n-1]$ pour les n sommets. On initialise heapidx$[i] \leftarrow i$, et on échange les valeurs des cellules lors des itérations de swim et sink comme approprié. Ainsi, $j =$ heapidx$[x]$ identifie la position sur le tas : $H[j] = x$ à tout temps. Pour implanter decreaseKey, on appelle swim$\big(H[\text{heapidx}[x], \pi[x]\big)$.

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 ; H.decreaseKey(x, π[x])
```

Choix de file de priorité. Il existe d'autre structure de données pour files de priorité, avec lesquelles 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 $O(n \log n + m)$

	liste triée	**liste non-triée**	**binaire**	***d*-aire**	**binomial**	**skew** (amorti)	**Fibonacci** (amorti)
deleteMin	O(1)	O(n)	O(log n)	O(d log n/ log d)	O(log n)	O(log n)	O(log n)
insert	O(n)	O(1)	O(log n)	O(log n/ log d)	O(log n)	O(1)	O(1)
merge	O(n)	O(1)	O(n)	O(n)	O(log n)	O(1)	O(1)
decreaseKey	O(n)	O(1)	O(log n)	O(log n)	O(log n)	O(log n)	O(1)