

## 14 Algorithmes et structures de données sur graphes

### 14.1 Graphes et leur représentation

**Définition 14.1.** 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 14.2.** 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  $\text{Adj}[u]$  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 épars avec  $E = o(|V|^2)$ .

W<sub>(en)</sub>

### 14.2 Graphe pondéré

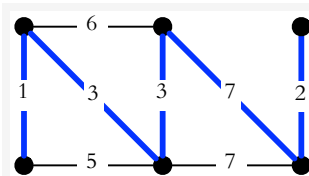
Soit  $w: E \mapsto [0, \infty]$  la *pondération des arêtes* du graphe  $G = (V, E)$ . Le poids d'une arête représente le coût de lier deux sommets dans beaucoup de problèmes d'optimisation ; on peut même considérer l'extension à toutes les paires avec  $w(uv) = \infty$  pour tout  $uv \notin E$ . Le 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 14.3.** Un *chemin* de longueur  $\ell$  est une séquence de  $\ell + 1$  sommets distincts  $v_0, v_1, \dots, v_\ell$  où  $v_{i-1}v_i \in E$  pour tout  $i = 1, \dots, \ell$ . ( $\ell = 0$  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.

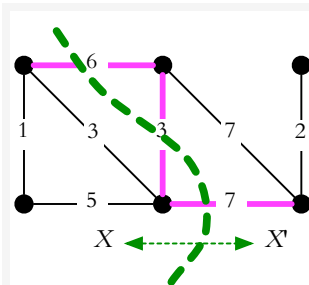
**Définition 14.4.** Un *arbre couvrant* est un sous-graphe  $T = (V, E')$  avec  $E' \subseteq E$  qui est connexe et ne contient pas de cycles. Un *arbre couvrant minimal* (ACM) est un arbre couvrant dont le poids atteint le minimum.

W<sub>(fr)</sub>

### 14.3 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 14.5.** 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 coupure sans aucune arête bleue. Choisir une arête non-coloriée avec le poids minimum qui traverse la coupure et la colorier par bleue.

**Théorème 14.1.** Les arêtes bleues forment un ACM.

*Démonstration.* On démontre par induction que la propriété 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 la 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 acyclique) 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  $T^*$ . ■

### 14.3.1 Algorithme de Kruskal

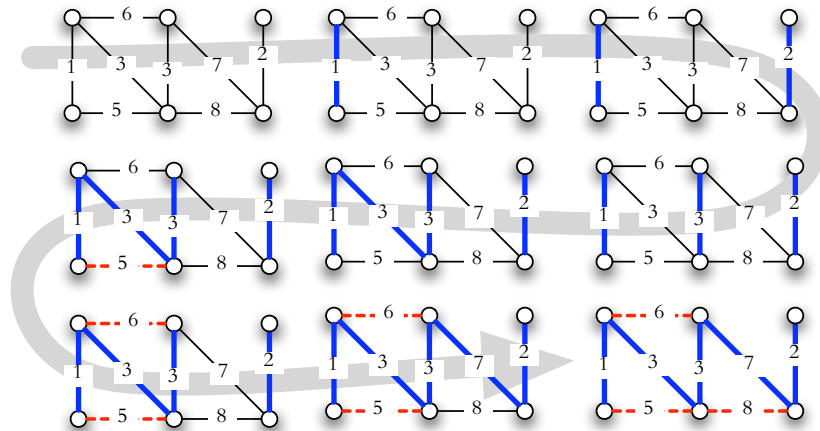
Dans l'algorithme de Kruskal, les arêtes bleues forment un forêt. À chaque itération, on colorie l'arête de poids minimal qui ne crée pas de cycle. On a un cycle si et seulement si  $u$  et  $v$  sont déjà liés par des arêtes choisies. En conséquence, il suffit de maintenir la connexité par les arêtes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

$W_{(fr)}$

```

K1 KRUSKAL( $V, E$ )                                // ACM pour graphe avec  $|V| = n$  sommets et  $|E| = m$  arêtes
K2 initialiser  $F \leftarrow \emptyset$ ;                // forêt d'arêtes bleues
K3 initialiser union-find pour  $n$  sommets
K4 trier les arêtes  $E[0 \dots m - 1]$  dans l'ordre croissant de poids  $w$ 
K5 for  $i \leftarrow 0 \dots m - 1$ 
K6    $uv \leftarrow E[i]$                                // prochaine arête
K7   if  $\text{find}(u) \neq \text{find}(v)$                        //  $uv$  ne peut créer de cycle
K8   then  $F \leftarrow F \cup \{uv\}$ ; 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 représenté 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

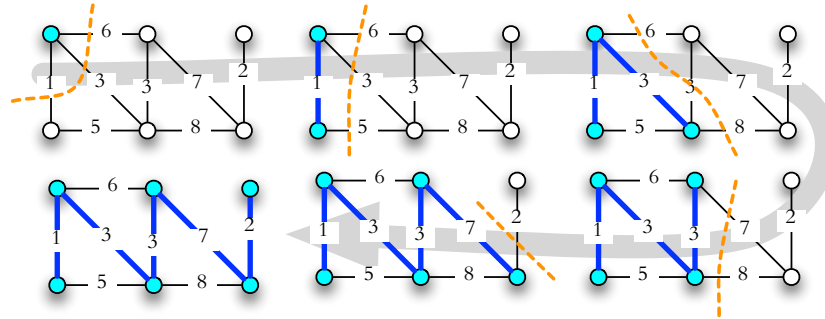
$$O(m \log m) + (2m + n - 1) \cdot O(\alpha(2m + n - 1, n)) = O(m \log m) + O(m\alpha(m, n)) = O(m \log m),$$

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

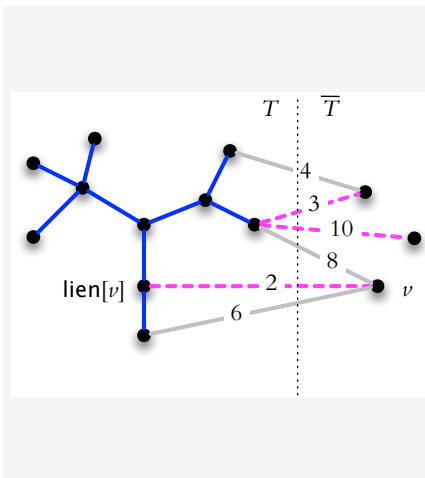
### 14.3.2 Algorithme de Prim

$W_{(fr)}$

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 arbre. 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  $T \leftarrow \emptyset$
- 3 **tandis que**  $T$  ne contient tous les sommets
- 4     soit  $uv$  une arête avec  $u \in T, v \notin T$  et  $w(uv)$  minimal
- 5      $T \leftarrow T \cup \{uv\}$



Comme implémentation naïve, on peut juste parcourir toutes 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 inférieur 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  $\text{lien}[v] = u \in T$  avec lequel  $\pi[v] = w(uv)$ .

```

P1 PRIM(s) // avec une file à priorités
P2  $T \leftarrow \emptyset$  //  $T$  contient les arêtes bleues
P3 for  $u \in V$  do  $\pi[u] \leftarrow \infty$  // priorité = distance du sommet
P4  $\pi[s] \leftarrow 0$ ; initialiser file de priorité  $H$  avec les sommets
P5  $v \leftarrow H.deleteMin()$ 
P6 do
P7  $\pi[v] \leftarrow -\infty$  //  $\pi[v] = -\infty$  si  $v \in T$ 
P8 for  $x: vx \in E$  do // parcourir la liste d'adjacence
P9 if  $w(vx) < \pi[x]$  then  $\pi[x] \leftarrow w(vx)$ ;  $lien[x] \leftarrow v$ ;  $H.decreaseKey(x, \pi[x])$ 
P10  $v \leftarrow H.deleteMin()$  //  $v = null$  quand file devient vide
P11 if  $v \neq null$  then  $u \leftarrow lien[v]$ ;  $T \leftarrow T \cup \{uv\}$ 
P12 while  $v \neq null$ 

```

**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 appelée `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 exécutée 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. Avec un tas binaire  $H[]$ , 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. `heapidx[x]` identifie la position sur le tas :  $H[\text{heapidx}[x]] = x$  à tout temps. Lors des itérations de `swim` et `sink`, on échange les valeurs des cellules `heapidx` comme appropriée. Pour implanter `decreaseKey`, on appelle `swim(heapidx[x], ...)`.

**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.

$W_{(n)}$

```

P9 if  $\pi[v] + w(vx) < \pi[x]$  then  $\pi[x] \leftarrow \pi[v] + w(vx)$ ;  $lien[x] \leftarrow v$ ;  $H.decreaseKey(x, \pi[x])$ 

```

**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	$O(n)$	$O(1)$	$O(\log n)$	$O(\log n / \log d)$	$O(\log n)$	$O(\log n)$	$O(1)$