Chapitre 3

Graphes pondérés

Sommaire

Comme on l'a déjà évoqué précédemment, certaines applications exigent de rajouter des propriétés sur les arêtes ou les sommets des graphes manipulés. Nous examinerons dans ce chapitre des algorithmes résolvant des problèmes sur des graphes *pondérés*, ce qui signifie que leurs *arêtes* possèdent un poids.

Définition 12. Un *graphe pondéré* est un graphe $G = (V, E, w)$, où $w : E \to \mathbb{R} : \{u, v\} \mapsto w(\{u, v\})$ est une fonction affectant à chaque arête un poids réel.

Les graphes non pondérés qu'on a vus jusqu'ici peuvent être vus comme des graphes pondérés dans lesquels on affecte le même coût (par exemple 1) à chaque arête. On pourrait aussi décider d'affecter des poids aux sommets en plus (ou à la place) des arêtes à l'aide d'une autre fonction.

Définition 13. Le *poids* d'un (sous-)graphe $G = (V, E, w)$ est la quantité $w(G) = \sum_{e \in E} w(e)$.

Dans ce chapitre, tous les graphes que nous considérerons seront pondérés et connexes afin de simplifier la discussion, mais rien ne nous empêcherait de les exécuter sur des graphes comportant plusieurs composantes.

3.1 Implémentation des graphes pondérés

Il n'y a pas grand-chose à modifier aux classes déjà implémentées pour prendre en compte les poids : si on utilise une matrice d'adjacence, n'importe quelle valeur de la matrice peut servir de poids et il n'y a donc rien à modifier; si l'on utilise une liste ou un dictionnaire d'adjacence, on utilise un couple `(v, poids)` pour décrire l'arête $\{u, v\}$ d'un poids donné.

On supposera donc l'existence d'une classe GraphePondéré, disposant des méthodes déjà couvertes dans le cas des graphes non pondérés et qui devront être légèrement modifiées :

- ajouter_arête(u, v, poids) prend maintenant en paramètre le poids de l'arête $\{u, v\}$;
- ajouter_arêtes(séquence) suppose maintenant qu'on lui fournit des triplets plutôt que des couples ;
- arêtes() renvoie également des triplets ;
- boucles() renvoie maintenant des couples (sommet, poids) ;
- sous_graphe_induit(sequence) renvoie maintenant un graphe pondéré.

On supposera également l'existence des nouvelles méthodes suivantes :

- arêtes_incidentes(sommet), qui renvoie l'ensemble des arêtes incidentes à un sommet donné sous la forme de triplets ;
- poids_arête(u, v), qui renvoie le poids de l'arête $\{u, v\}$ si elle existe, NIL sinon.

Exercice 6. Modifiez vos implémentations de la matrice d'adjacence et de la liste d'adjacence pour prendre en compte les poids sur les arêtes.

3.2 Arbres couvrants de poids minimum

On a déjà parlé d'arbre couvrant dans le contexte des arbres de parcours (section 2.3). Plus généralement :

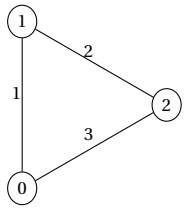
Définition 14. Un sous-graphe *couvrant* d'un graphe connexe donné $G = (V, E)$ est un graphe connexe de la forme $H = (V, F)$ où $F \subseteq E$.

Si le graphe n'est pas pondéré, il est facile de construire un arbre couvrant en réalisant un simple parcours du graphe. Si le graphe est pondéré, chaque arête a par définition un coût, et cela a du sens de préférer un arbre particulier parmi tous les arbres couvrants disponibles :

Définition 15. Un *arbre couvrant de poids minimum* (ou *ACPM*) pour un graphe pondéré G est un arbre couvrant T pour G tel que pour tout arbre couvrant T' pour G, on a $w(T) \leq w(T')$.

Dans ce cas, on ne peut plus se contenter d'utiliser un parcours.

Exemple 17. Voici un graphe pour lequel un parcours ne donne pas un arbre couvrant minimum : explorer le graphe en largeur à partir du sommet 2 donne un arbre de poids 5, alors que les arêtes $\{0, 1\}$ et $\{1, 2\}$ donnent une solution de poids 3.



Deux algorithmes vont nous permettre d'obtenir des arbres couvrants minimaux : celui de Prim et celui de Kruskal. Pour bien visualiser les différences entre ces algorithmes, et notamment constater qu'ils vont bien tous deux trouver un arbre couvrant de même poids

minimum mais pas nécessairement le même arbre, on les exécutera sur le même graphe, qui sera celui montré à la Figure 3.1 et tiré de Skiena [5].

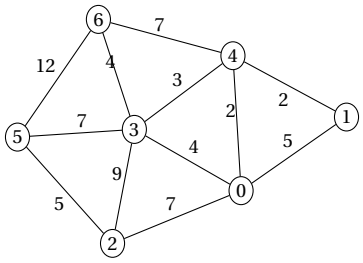


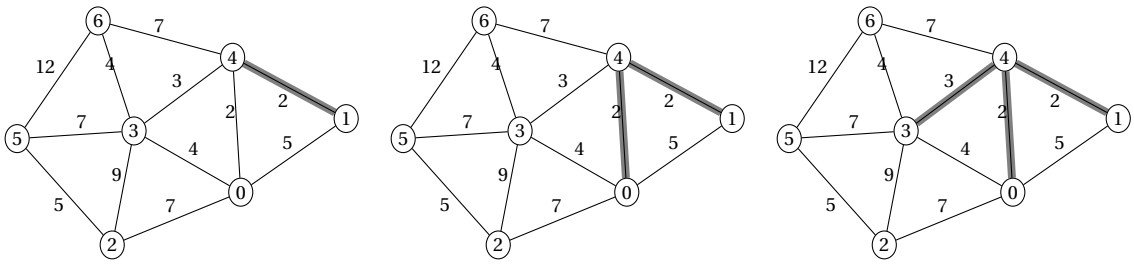
FIGURE 3.1 – Un graphe pondéré sur lequel on illustrera le fonctionnement des algorithmes de Prim et de Kruskal [5].

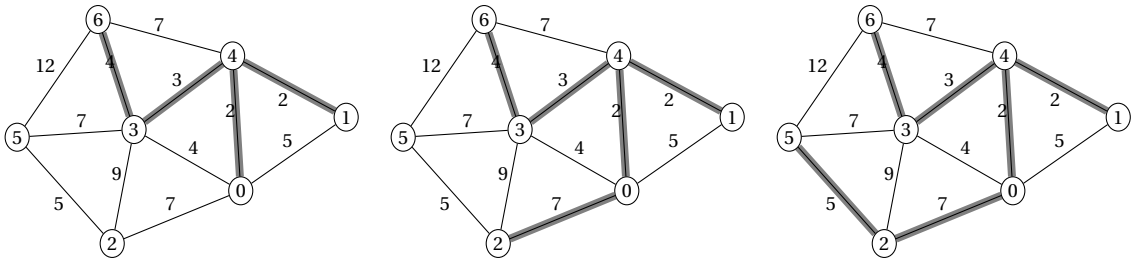
L'exemple typique de motivation pour l'étude de ces ACPM est la construction d'un sous-réseau permettant d'atteindre tous les sommets d'un réseau donné, dont on veut minimiser le coût global qui est donné par le poids du sous-graphe construit ; le plus petit graphe que l'on peut construire doit au moins être un arbre couvrant, et trouver l'arbre le moins coûteux revient à calculer un ACPM si les poids ne sont pas négatifs. Mais ces arbres trouvent également des applications dans la résolution d'autres problèmes, comme par exemple celui du voyageur de commerce, où ils servent d'approximations. Beaucoup plus d'informations et d'applications des ACPM ainsi que plusieurs sujets qui y sont liés ont été couverts par Graham et Hell [3] et Mareš [4].

3.2.1 L'algorithme de Prim

L'algorithme de Prim démarre une exploration du graphe à partir d'un sommet arbitraire, et construit un arbre à partir de ce sommet en rajoutant à chaque étape l'arête de poids minimum connectant l'arbre en cours de construction à un sommet non visité, jusqu'à ce qu'on ait exploré tous les sommets du graphe. Remarquons que l'on peut résoudre les ambiguïtés de manière arbitraire : si on a le choix entre plusieurs arêtes de même poids minimum, on peut sélectionner n'importe laquelle d'entre elles.

Exemple 18. Voici le déroulement de l'algorithme de Prim sur le graphe de la Figure 3.1 au départ du sommet 1 ; on finit par trouver un ACPM de poids 23.





(4) (5) (6)

(fin exemple 18)

Pour implémenter cet algorithme, on peut suivre sa description sans trop de problèmes : on stockera dans une structure prévue à cet effet les arêtes de notre graphe, et on passera notre temps à extraire de cette structure l'arête de poids minimum nous permettant d'augmenter notre arbre sans créer de cycles pour l'ajouter à celui-ci. On fera la distinction entre plusieurs catégories d'arêtes lors de la construction de l'arbre :

Définition 16. Soit $G = (V, E, w)$ un graphe pondéré non orienté, et T un sous-graphe acyclique de G. On dit qu'une arête $e = \{u, v\}$ de G est :
- *candidate* si elle possède au moins une extrémité dans T ;
- *valide* si elle est candidate et que $T \cup e$ est acyclique ;
- *sûre* si elle est valide et de poids minimum parmi toutes les arêtes valides.

Remarquons que dans le cas de l'algorithme de Prim, l'ajout d'une arête donnée à notre arbre créera un cycle si et seulement si les *deux* extrémités de cette arête appartiennent déjà à l'arbre. Pour pouvoir vérifier rapidement si une arête peut nous créer un cycle dans l'arbre que l'on est en train de construire, on stocke donc pour chaque sommet un booléen nous indiquant si le sommet correspondant est ou non hors de l'arbre. Ainsi, lors de la sélection d'une arête candidate, on ne la retiendra que si exactement une de ses extrémités est hors de l'arbre (par construction, l'autre appartiendra forcément à l'arbre puisqu'on n'examine que les arêtes ayant au moins une extrémité dans l'arbre), ce que le tableau nous permettra donc de vérifier en $O(1)$.

Utilisation d'un tas

Si l'on stocke les arêtes candidates dans une structure ordinaire, on doit parcourir en entier cette structure à chaque fois que l'on veut en extraire une arête, ce que l'on fait à chaque itération de notre boucle principale. Une façon d'accélérer cet algorithme serait d'utiliser une structure de données plus performante pour extraire les arêtes de poids minimum : on va donc utiliser un *heap*[1] (ou *tas*, ou *file à priorité*) pour pouvoir extraire le minimum en temps $O(\log n)$ plutôt qu'en temps $O(n)$ (n étant ici la taille de la structure stockant les arêtes).

Définition 17. Un *tas* est un arbre binaire enraciné dont les sommets sont comparables et vérifiant la propriété suivante : pour chaque sommet s du tas ayant pour fils gauche

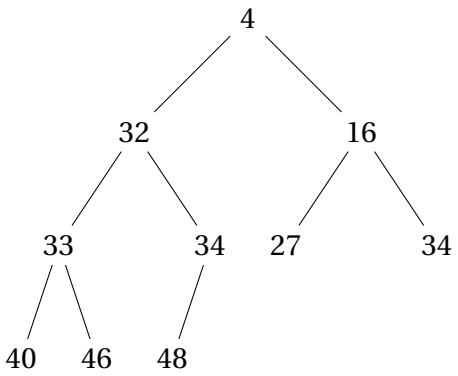
1. Voir le cours d'algorithmique des arbres pour les détails.

g et pour fils droit d, on a

$$s.\text{valeur} = \min(s.\text{valeur}, f.\text{valeur}, g.\text{valeur}).$$

Un tas à n sommets peut être représenté par un simple tableau, dans lequel les fils gauche et droit d'un sommet d'indice i se trouvent en positions respectives $2i+1$ et $2i+2$.

Exemple 19. Le tableau [16, 40, 4, 32, 34, 27, 34, 33, 46, 48] ne représente pas un tas valide, car la racine contient 16 qui est plus grand que le minimum de ses deux fils (40 et 4). On peut le réorganiser en [4, 32, 16, 33, 34, 27, 34, 40, 46, 48], qui est cette fois valide et que l'on peut représenter par un arbre binaire comme suit :



On supposera qu'une classe Tas() est disponible avec les méthodes suivantes :

- un constructeur Tas(S), qui organise les données de S sous la forme d'un tas en $O(|S|)$;
- une méthode insérer(élément), qui ajoute un élément au tas en $O(\log|T|)$ **et** garantit que le résultat après insertion est toujours un tas ;
- une méthode extraire_minimum(), qui extrait et renvoie le minimum du tas T en $O(\log|T|)$ **et** garantit que le résultat après extraction est toujours un tas.

L'algorithme 14 montre une implémentation efficace de l'algorithme de Prim, qui s'appuie sur les algorithmes auxiliaires 12 et 13, permettant respectivement l'ajout d'arêtes valides et la sélection de nouvelles arêtes sûres.

Algorithme 12 : StockerAretesValides(G, u, S, hors_arbre)

Entrées : un graphe pondéré non orienté G, un sommet u de G, un tas d'arêtes S et un tableau booléen hors_arbre.
Résultat : les arêtes valides incidentes à u sont ajoutées à S.

```
1 pour chaque v ∈ G.voisins(u) faire
2 |   si hors_arbre[v] alors S.insérer((u, v, G.poids_arête(u, v))) ;
```

Attention, l'algorithme 12 enregistre bien toutes les arêtes qui sont valides au moment où on les ajoute à S ; mais certaines de ces arêtes pourraient devenir non valides au fur et à mesure que l'on avance dans la construction de l'arbre. Ceci explique pourquoi, dans l'algorithme 13, on doit vérifier la validité des arêtes que l'on extrait de S ; et l'on en profite au passage pour se débarrasser de celles qui ne sont plus valides, puisqu'elles ne pourront pas le redevenir.

Exemple 20. Reprenons les trois premières étapes de l'exemple 18, et examinons comment l'ensemble $V(T)$ des sommets de l'arbre et l'ensemble S des arêtes valides évoluent :

— au début, on a $V(T) = \{1\}$, ce qui donne $S = \{\{0,1\},\{1,4\}\}$;
— l'extraction de $\{1,4\}$ donne $V(T) = \{1,4\}$, et force la sélection des arêtes valides incidentes à 4, donc $S = \{\{0,1\},\{0,4\},\{3,4\},\{4,6\}\}$;
— enfin, l'extraction de $\{0,4\}$ donne $V(T) = \{0,1,4\}$, et l'arête $\{0,1\} \in S$ n'est donc plus valide.

Algorithme 13 : ExtraireAreteSure(S, hors_arbre)

Entrées : un tas S d'arêtes et un tableau booléen hors_arbre.
Résultat : une arête sûre (ou factice s'il n'y en a pas) est extraite de S et renvoyée ; les arêtes invalides éventuellement rencontrées sont éliminées.

```
1 tant que S.pas_vide() faire
2     (u, v, p) ← S.extraire_minimum();
3     si hors_arbre[u] ≠ hors_arbre[v] alors renvoyer (u, v, p) ;
4 renvoyer (NIL, NIL, +∞);
```

Algorithme 14 : Prim(G, départ)

Entrées : un graphe pondéré non orienté G, un sommet de départ.
Sortie : un ACPM pour la composante connexe de G contenant départ.

```
1  arbre ← GraphePondéré();
2  arbre.ajouter_sommet(départ);
3  hors_arbre ← tableau(G.nombre_sommets(), VRAI);
4  hors_arbre[départ] ← FAUX;
5  candidates ← Tas() ;
6  StockerAretesValides(G, départ, candidates, hors_arbre) ;
7  tant que arbre.nombre_aretes() < G.nombre_sommets() - 1 faire
8      (u, v, p) ← ExtraireAreteSure(candidates, hors_arbre) ;
9      si u = NIL alors renvoyer arbre ;
10     si ¬ hors_arbre[u] alors échanger u et v;
11     arbre.ajouter_arête(u, v, p);
12     hors_arbre[u] ← FAUX;
13     StockerAretesValides(G, u, candidates, hors_arbre) ;
14 renvoyer arbre;
```

Exercice 7. Adaptez l'algorithme de Prim pour qu'il fournisse une *forêt* couvrante de poids minimum.

Correction

Prouvons maintenant que l'algorithme de Prim est correct.

Théorème 3.2.1. [2] L'algorithme 14 s'arrête et renvoie bien un ACPM pour tout graphe connexe.

Démonstration. La terminaison est claire : chaque itération de la boucle principale rajoute exactement une arête et par la même occasion un sommet à l'arbre en cours de construction, jusqu'à ce qu'on atteigne le nombre de sommets du graphe ou que l'on ne puisse plus ajouter d'arête. Passons maintenant à ce que renvoie l'algorithme ; on a trois propriétés à prouver :

1. **le résultat est un arbre** : c'est bien le cas si le graphe est connexe, car on ajoute exactement une arête à chaque étape à un arbre en cours de construction en prenant explicitement garde à ne pas créer de cycle dans cette structure.
2. **cet arbre est couvrant** : c'est bien le cas, puisqu'on ne cesse de lui rajouter des arêtes que lorsque tous les sommets ont été couverts.
3. **il est de poids minimum** : pour prouver ce dernier point, on peut procéder par induction sur la taille de l'arbre T construit par l'algorithme à chaque itération, en prouvant qu'à chaque étape, il existe un ACPM pour G qui contient T :
   (a) cas de base : si T ne contient qu'un sommet, la propriété est trivialement vérifiée puisque tout arbre couvrant pour G devra par définition contenir ce sommet.
   (b) induction : supposons la propriété vraie pour $|V(T)| = p < |V(G)|$, et montrons que cela implique qu'elle est vraie pour la valeur $p+1$. L'hypothèse d'induction nous dit qu'il existe un ACPM T_{opt} contenant T, et l'on s'apprête à construire l'arbre $T' = T \cup e$, où e est une arête sûre (voir la Figure 3.2) ;
      i. si $e \in E(T_{opt})$, alors on a prouvé la propriété pour la valeur $p+1$ puisqu'il existe un ACPM (à savoir T_{opt}) contenant T' ;
      ii. sinon, on va construire un autre ACPM T'_{opt} contenant e à partir de T_{opt}, en remplaçant une arête de T_{opt} par e.
      Comme T_{opt} est un arbre couvrant, $T_{opt} \cup e$ contient un cycle. Ce cycle contient au moins deux arêtes valides : l'une de ces deux arêtes est e, que l'on emprunte pour suivre le cycle de $V(T)$ vers $\overline{V(T)}$, et l'autre, que nous noterons f, est l'une de celles qui nous ramène dans $V(T)$ au départ de $\overline{V(T)}$.
      Le graphe $T'_{opt} = T_{opt} \cup e \setminus f$ est donc bien un arbre couvrant puisque T_{opt} en était un, et il nous reste à montrer qu'il est bien de poids minimum ; on remarque que $w(e) \leq w(f)$, car $w(f) < w(e)$ nous aurait fait choisir f à l'étape actuelle au lieu de e. On a donc $w(T'_{opt}) = w(T_{opt}) + w(e) - w(f) \leq w(T_{opt})$, et T'_{opt} est donc bien un ACPM contenant T'.

□

Complexité

Les appels répétés à StockerAretesValides(G, u, S, hors_arbre) nous font examiner chaque arête deux fois, et leur insertion dans le tas S coûte un temps logarithmique, ce qui nous donne donc une complexité de $O(|E|\log|E|)$. Le nombre d'itérations de la boucle de l'algorithme 14 à la ligne 7 est de l'ordre de $O(|V|)$, puisqu'on s'arrêtera après avoir examiné V sommets. Enfin, ExtraireAreteSure(S, hors_arbre) s'exécute en $O(\log|S|)$, et S contient dans le pire des cas toutes les arêtes du graphe. On a donc du $O(|V|\log|E| + |E|\log|E|) = O(|E|\log|E|) = O(|E|\log|V|)$ pour cette implémentation, si l'on utilise une liste d'adjacence pour implémenter le graphe.

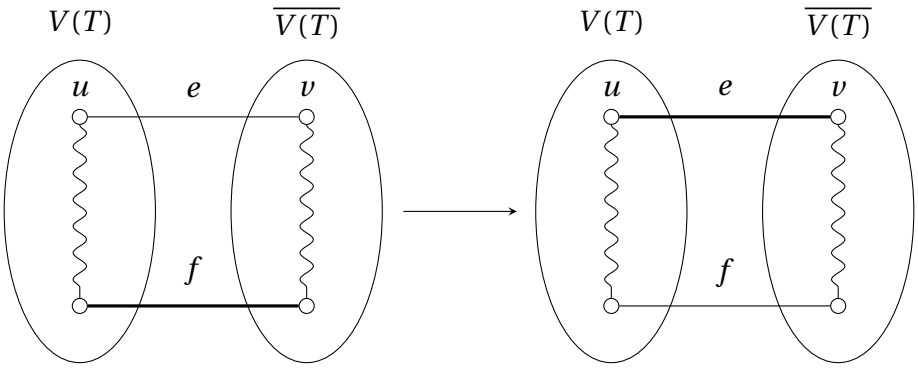
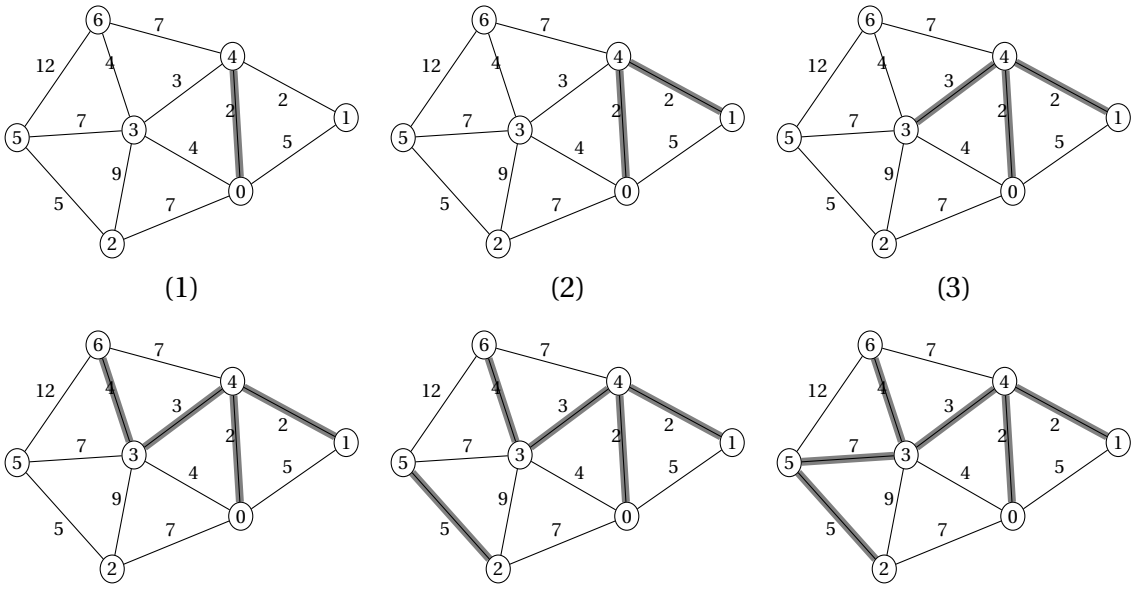


FIGURE 3.2 – Preuve du Théorème 3.2.1 ; les traits en zig-zag sont des chemins contenus dans T_{opt}, et l'on remplace f par e dans T_{opt}.

3.2.2 L'algorithme de Kruskal

L'idée de l'algorithme de Kruskal est de faire croître une forêt de poids minimal initialement vide, en lui rajoutant à chaque étape une arête sûre. Ceci est donc très proche de la stratégie de Prim : la différence étant que dans l'algorithme de Prim, on doit spécifier un sommet de départ, ce qui limite notre choix des arêtes candidates, alors que dans l'algorithme de Kruskal tous les sommets du graphe appartiennent à la forêt en cours de construction et donc toutes les arêtes du graphe sont candidates. Remarquons que l'on peut résoudre les ambiguïtés de manière arbitraire, comme pour l'algorithme de Prim.

Exemple 21. Voici le déroulement de l'algorithme de Kruskal sur le graphe de la Figure 3.1 au départ du sommet 1 ; on finit par trouver un ACPM de poids 23.



Pour implémenter cet algorithme, on peut suivre sa description sans trop de problèmes en stockant les arêtes triées par poids croissant. Il nous faudra également pouvoir vérifier rapidement si une arête risque de nous créer un cycle dans la forêt que l'on est en train de

construire ; on ne peut plus utiliser l'astuce de l'algorithme de Prim, car il est tout à fait possible que deux sommets appartiennent à la forêt mais que l'ajout d'une arête entre ces deux sommets ne crée pas de cycle. Au lieu de cela, il faut que pour chaque sommet, on enregistre à quelle **composante** de la forêt il appartient. Ainsi, lorsqu'on examine une arête candidate $\{u, v\}$ à rajouter à notre forêt, on distingue les cas suivants :

1. si u et v n'appartiennent pas à la même composante, on peut utiliser l'arête sans crainte puisqu'elle ne nous créera pas de cycle ;
2. si u et v appartiennent à la même composante, on ignore l'arête puisqu'elle créera un cycle.

Utilisation de la structure *Union-Find*

La difficulté pour obtenir une implémentation efficace réside dans le fait que l'on va devoir mettre à jour les informations d'appartenance de chaque sommet aux composantes. Deux idées immédiates nous permettent d'implémenter cette structure :

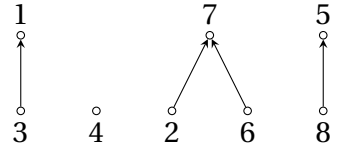
1. on pourrait stocker explicitement les classes de notre partition sous la forme d'une collection d'ensembles. Réaliser l'union de deux classes A et B nécessite une complexité en $O(|A| + |B|)$, et identifier à quelle classe appartient un élément requiert de toutes les parcourir en entier, ce qui nous donne du $O(|V|)$ et n'est pas satisfaisant.
2. on peut utiliser une liste `marqueurs`, dans laquelle `marqueurs[i]` contient la classe à laquelle appartient `i` ; ceci nous permet de répondre à la requête de calculer la classe d'un élément en temps $O(1)$, mais cela nous oblige à propager les informations d'appartenance lorsqu'on réalise une fusion et coûte du $O(|V|)$.

Au lieu de cela, on aura recours à une structure de données bien plus efficace : la structure de données *Union-Find*. Elle a pour but de représenter une partition d'un ensemble S de manière à ce que les deux opérations suivantes qui nous intéressent, dont la structure tire son nom, s'exécutent rapidement :

— *union(A, B)* : fusionne les classes A et B ;
— *find(x)* : renvoie l'identifiant de la classe à laquelle appartient l'élément x.

L'idée de la structure *Union-Find* consiste à représenter les morceaux de notre partition par des arbres enracinés dont les arêtes sont orientées des fils vers leur père.

Exemple 22. Voici une représentation possible de la partition $\{\{1,3\},\{2,6,7\},\{4\},\{5,8\}\}$ par une forêt.



Cette représentation n'est pas unique : choisir 3 comme parent de 1 au lieu du contraire serait tout aussi correct.

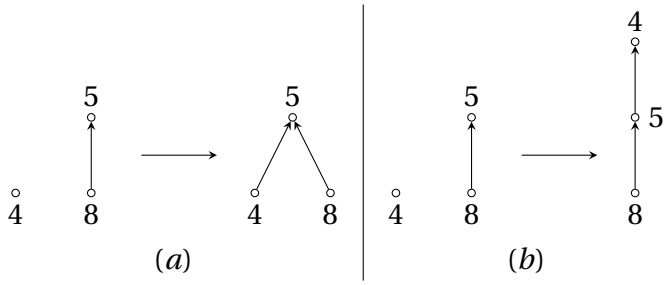
Chaque classe de notre partition peut donc être identifiée par sa racine, et cette vision nous permet de déduire les algorithmes suivants pour les deux opérations qui nous intéressent :

1. *union(A, B)* : pour fusionner deux classes, il nous suffit de fusionner les arbres correspondants, ce qui implique de faire de la racine de l'un un descendant de la racine de l'autre.
2. *find(x)* : pour savoir à quelle classe l'élément x appartient, il nous suffit de remonter l'arbre de père en père jusqu'à ce que l'on tombe sur la racine que l'on renvoie. Si x est le seul élément de sa classe, on renvoie x.

Fusions

On remarque que s'il n'y a pas d'ambiguïté pour l'opération *find*, l'union peut se réaliser de deux façons différentes : soit on accroche la racine de A sous celle de B, soit on accroche la racine de B sous celle de A. L'option que l'on choisit est importante, car si l'on n'y prend pas garde, on se retrouvera avec un arbre très déséquilibré[2]. C'est pourquoi on décidera d'accrocher l'arbre le moins haut sous la racine de l'arbre le plus haut.

Exemple 23. Deux exemples de fusions : (*a*) une bonne fusion, qui limite la croissance des arbres, et (*b*) une mauvaise fusion.



Calculer explicitement la hauteur des arbres à fusionner serait trop coûteux en temps ; on va donc plutôt se contenter de stocker un champ `rang`, qui sera au départ égal à la hauteur de l'arbre (à savoir 0). Ainsi, lorsqu'on aura une fusion à effectuer, on consultera les valeurs `rang[r]` et `rang[s]` associées aux racines `r` et `s` des arbres à fusionner, et l'on se trouvera dans l'un de ces trois cas de figure :

1. si `rang[r] < rang[s]`, `s` deviendra le parent de `r` ;
2. si `rang[s] < rang[r]`, `r` deviendra le parent de `s` ;
3. sinon, `rang[r] == rang[s]` ; on prendra alors une décision arbitraire parmi les deux précédentes, en incrémentant ensuite le rang de l'arbre qui se sera retrouvé sous la racine de l'autre.

Cette stratégie garantit que la hauteur de chaque arbre sera logarithmique en son nombre de sommets, et donc que l'opération *find* s'exécutera en $O(\log|S|)$ — ce qui sera aussi le cas de l'union, puisque *union(a, b)* fera appel à *find* pour trouver les racines de a et de b.

Compression de chemins

Une optimisation simple et efficace consiste, lors de l'exécution d'une opération *find(x)*, à réduire la taille de l'arbre sur lequel la requête a été effectuée en plaçant tous les sommets

2. Rappelons que c'est ce qui avait poussé les créateurs des arbres AVL à proposer leur structure comme alternative aux arbres binaires de recherche.

rencontrés dans le chemin de x vers la racine directement sous la racine. En d'autres termes : plus on utilise la structure, plus elle devient performante. Une analyse poussée permet de déduire que le temps d'exécution des opérations *union* et *find* est en $O(\alpha(n))$, où $\alpha(n)$ est l'inverse de la fonction d'Ackermann [1]. Cette fonction croît tellement lentement qu'on peut considérer que les opérations qui nous intéressent s'effectueront toujours en $O(1)$, ce qui nous permet d'affirmer que la complexité de l'algorithme de Kruskal implémenté à l'aide de cette structure est en $O(|E|\log|V|)$: seule l'étape de tri des arêtes est coûteuse.

L'algorithme 15 est basé sur cette structure. Remarquons que contrairement à l'algorithme de Prim, l'algorithme de Kruskal ne requiert pas que le graphe soit connexe puisqu'on prend directement le parti de construire une forêt. Il est important d'affecter à v la classe de u **après** avoir traité les autres sommets ; si on le fait avant, on devient incapable d'identifier les sommets appartenant à la même classe que v.

Algorithme 15 : KRUSKAL(G)

Entrées : un graphe pondéré non orienté G.
Sortie : une forêt couvrante de poids minimum pour G consistant en un arbre couvrant de poids minimum pour chaque composante connexe de G.

```
1 forêt ← GraphePondéré(G.sommets());
2 classes ← UnionFind(G.sommets()) ;
3 pour chaque (u, v, p) ∈ tri_par_poids_croissant(G.arêtes()) faire
4     si classes.find(u) ≠ classes.find(v) alors
5         forêt.ajouter_arête(u, v, p);
6         classes.union(classes.find(u), classes.find(v));
7 renvoyer forêt;
```

Complexité

La correction de l'algorithme de Kruskal peut se prouver d'une manière similaire à celle de l'algorithme de Prim. La complexité de l'algorithme 15 s'obtient comme suit :

— le tri nous coûte $O(|E|\log|E|)$;
— on passe $O(|E|)$ fois dans la boucle, et chaque itération induit potentiellement deux recherches avec *find* (en $O(\log|V|)$) ainsi qu'une fusion en $O(\log|V|)$.

La complexité de cette implémentation de l'algorithme de Kruskal est donc dominée par l'étape de tri, et l'on obtient $O(|E|\log|E|) = O(|E|\log|V|)$ pour une liste d'adjacence.

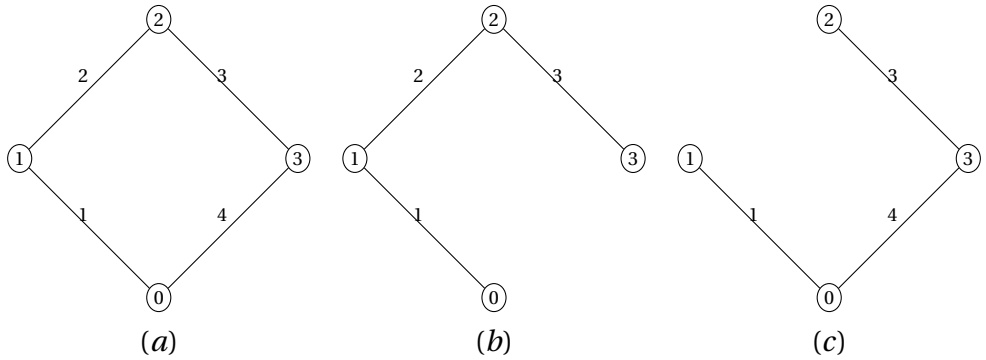
3.3 Plus courts chemins dans un graphe

Un usage extrêmement répandu des graphes pondérés, que l'on retrouve notamment dans les systèmes GPS, est de rechercher le plus court chemin entre une origine et une destination fixées.

Exercice 8. Si le graphe n'était pas pondéré, comment calculerait-on un plus court chemin entre deux sommets ?

Comme le montre l'exemple suivant, les algorithmes de calcul d'un arbre couvrant de poids minimum ne fonctionneront pas pour atteindre cet objectif en général.

Exemple 24. Voici (*a*) un graphe pondéré G , (*b*) un arbre couvrant de poids minimum (6) pour G, et (*c*) un arbre des plus courts chemins du sommet 3 vers tous les autres, de poids 8. L'ACPM ne contient pas l'arête $\{0,3\}$, et nous force donc à emprunter un chemin de longueur 6 pour passer de 0 à 3, au lieu du chemin optimal de longueur 4.



Les algorithmes permettant de trouver le plus court chemin entre une source et une destination construisent souvent un arbre de parcours, dont la racine est la source et dont la construction se termine quand l'arbre contient également le chemin le plus court vers la destination. Remarquons que même si l'arbre obtenu est couvrant, il n'est pas nécessairement de poids minimum, comme on l'a vu dans l'exemple 24.

Les objectifs dans les deux cas sont donc bien distincts : on construit un arbre couvrant de poids minimum quand on veut relier tous les sommets d'une composante connexe de la manière la moins coûteuse possible, et on construit les plus courts chemins quand on veut qu'un *trajet particulier* nous coûte le moins possible.

3.3.1 L'algorithme de Dijkstra

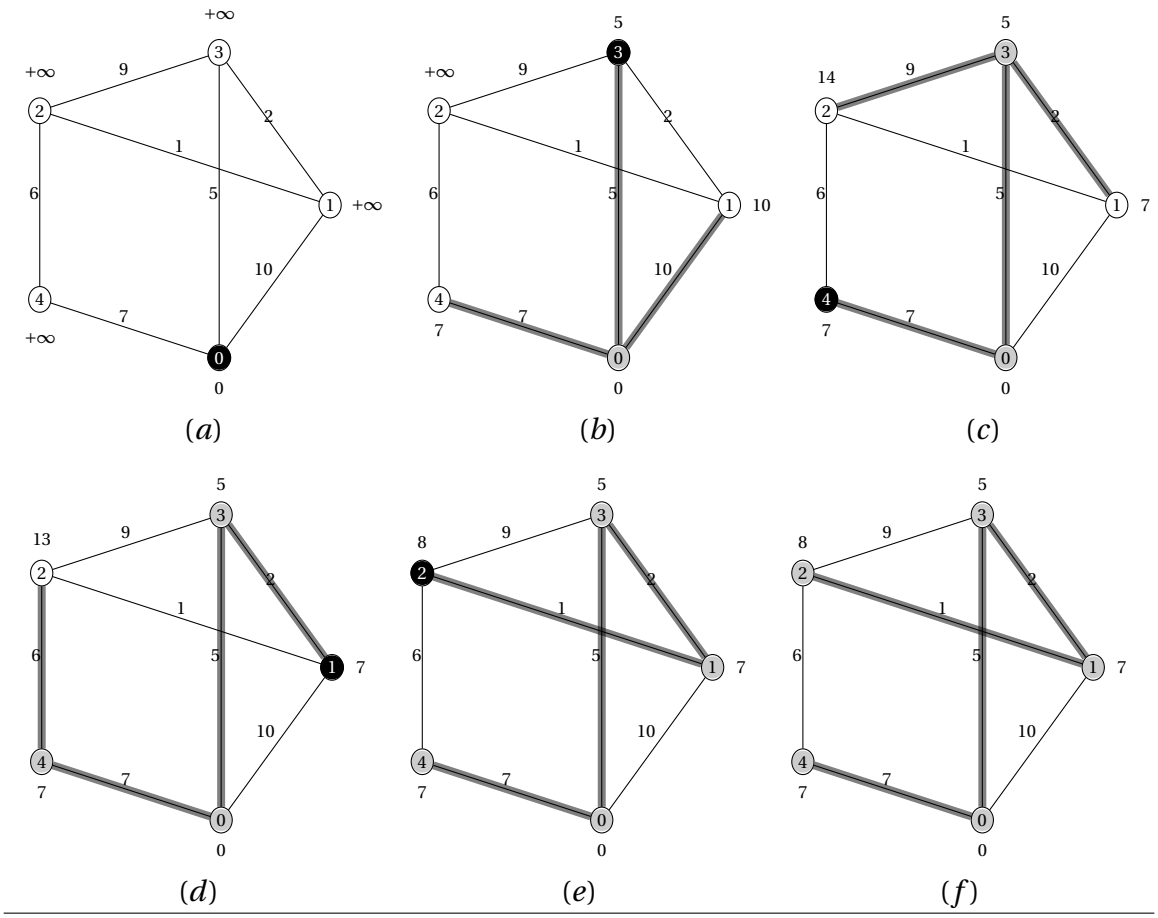
L'algorithme de Dijkstra[3] construit ce qu'on appelle souvent un *arbre des plus courts chemins* : la racine de cet arbre est le sommet de départ, et l'arbre ne contient qu'un chemin de poids minimal entre la source et chaque sommet du graphe. On obtient donc plus d'informations que ce qu'on a demandé : le plus court chemin entre une source et une destination, mais également de cette source vers un grand ensemble d'autres sommets du graphe par construction. Pour des raisons qu'on expliquera plus loin, **on suppose notre graphe simple[4] et sans arête de poids négatif**.

Cet algorithme explore le graphe à partir du sommet de départ s en traitant les sommets du graphe par distance croissante à un ensemble de sommets déjà traités. Pour connaître ces distances, on maintient dans une structure les distances connues du sommet de départ vers tous les sommets du graphe, en initialisant la distance au sommet de départ à 0 et les autres à $+\infty$ puisqu'on n'a pas encore découvert de chemin vers ces sommets. À chaque itération de l'algorithme, on extrait le sommet u dont la distance à s est la plus petite, et pour chaque voisin v de u, on met à jour la distance de s à v en conservant le minimum entre ce qu'on connaît déjà et le poids de la concaténation du plus court chemin de s à u avec l'arête $\{u,v\}$.

3. Le "Dij" se prononce comme "Day" en anglais.
4. Donc pour rappel : au plus une arête entre deux sommets et pas de boucles.

Ceci ressemble donc très fort à ce que réalise l'algorithme de Prim, à ceci près que l'arbre sous-jacent peut changer de structure, alors que dans l'algorithme de Prim, les arêtes de l'arbre en cours de construction ne sont plus remplacées une fois sélectionnées.

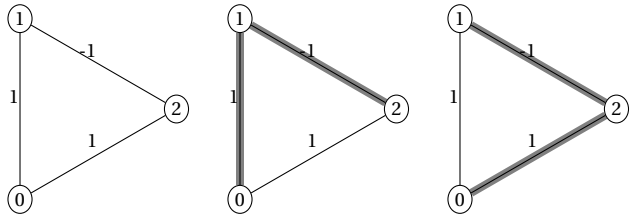
Exemple 25. Les étapes de l'algorithme de Dijkstra sur un graphe tiré de Skiena [5]. Les sommets déjà traités sont en gris, celui qu'on va traiter est en noir. En bas, l'évolution du tableau de distances à chaque étape de l'algorithme. L'encadrement montre le minimum et le sommet correspondant sera sélectionné ; les sommets dont les distances sont en **gras** ont été traités et ne seront plus examinés.



étape	0	1	2	3	4
(*a*)	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$
(*b*)	**0**	10	$+\infty$	5	7
(*c*)	**0**	7	14	**5**	7
(*d*)	**0**	7	13	**5**	**7**
(*e*)	**0**	**7**	8	**5**	**7**
(*f*)	**0**	**7**	**8**	**5**	**7**

L'algorithme de Dijkstra suppose que le graphe ne contient pas d'arête de poids négatif. Si ce n'est pas le cas, on ne peut pas en général s'attendre à ce que l'algorithme fonctionne.

Exemple 26. Dans le graphe suivant (a), le poids négatif sur l'arête $\{1,2\}$ pose problème : (b, c) sélectionner le plus court chemin de 0 à 1 (resp. 2) nous force à prendre le chemin le plus long de 0 à 2 (resp. 1). Le plus court chemin de 0 à 2 passe par 1, mais la structure d'arbre que l'on cherche nous force alors à sélectionner l'arête $\{0,1\}$ pour atteindre 1 alors que le chemin le plus court vers 1 passe par 2 ; et décider de passer par 2 pour atteindre 1 nous empêche de sélectionner le chemin optimal de 0 à 2. Remarquons par contre que si l'on veut obtenir les plus courts chemins de 1 vers le reste du graphe, sélectionner le chemin $(1,2,0)$ est une solution optimale.



(a) (b) (c)

Notons que la présence d'arêtes de poids négatifs n'était pas un problème pour les algorithmes de Prim et Kruskal cherchant des arbres ou des forêts couvrantes de poids minimum.

L'algorithme 17 implémente l'approche de Dijkstra : on enregistre dans un tableau les distances connues du sommet de départ à tous les autres sommets du graphe, et l'on traite chaque sommet du graphe par rapport à sa proximité à l'ensemble des sommets déjà traités. Lorsqu'un sommet est extrait, on examine tous ses voisins, et l'on met à jour le tableau de distances à chaque fois que l'on découvre un nouveau raccourci via ces voisins.

L'algorithme 16 utilisé comme routine auxiliaire implémente la recherche naïve de la recherche du minimum (et son extraction). Une approche plus efficace utiliserait un tas comme pour l'algorithme de Prim, mais requiert un peu plus de prudence car les distances estimées changent au fur et à mesure que l'algorithme 17 progresse dans ses calculs, au contraire de l'algorithme de Prim où les éléments stockés étaient des arêtes dont le poids ne changeait pas.

Algorithme 16 : EXTRAIRESOMMETLEPLUSPROCHE(S, distances)

Entrées : un ensemble S de sommets, et la distance de chaque sommet de S.
Résultat : le sommet de S le plus proche est extrait et renvoyé.

```
1 sommet ← NIL;
2 distance_min ← +∞;
3 pour chaque candidat ∈ S faire
4 |   si distances[candidat] < distance_min alors
5 |   |   sommet ← candidat;
6 |   |   distance_min ← distances[candidat];
7 si sommet ≠ NIL alors S ← S\ sommet ;
8 renvoyer sommet;
```

Algorithme 17 : DIJKSTRA(*G*, source)

Entrées : un graphe pondéré non orienté *G*, un sommet source.
Sortie : la longueur d'un plus court chemin de la source à chacun des sommets du graphe ($+\infty$ pour les sommets non accessibles).

```
1 a_traiter ← G.sommets();
2 distances ← tableau(G.nombre_sommets(), +∞);
3 distances[source] ← 0;
4 tant que a_traiter.pas_vide() faire
5 |   u ← ExtraireSommetLePlusProche(a_traiter, distances);
6 |   si u = NIL alors renvoyer distances ;
7 |   pour chaque v ∈ G.voisins(u) faire
8 |   |   distances[v] ← min(distances[v], distances[u] + G.poids_arête(u, v));
9 renvoyer distances;
```

3.3.2 Correction

Prouvons maintenant que l'algorithme de Dijkstra est correct.

Théorème 3.3.1. [2] Pour tout sommet source s d'un graphe pondéré G, l'algorithme 17 s'arrête et renvoie bien la liste des longueurs des plus courts chemins de s à tous les sommets de G.

Démonstration. La terminaison est évidente, puisqu'à chaque itération, on élimine un sommet de la structure des sommets à traiter et qu'on s'arrête quand cette structure est vide.

Pour montrer que l'algorithme de Dijkstra est correct, notons :

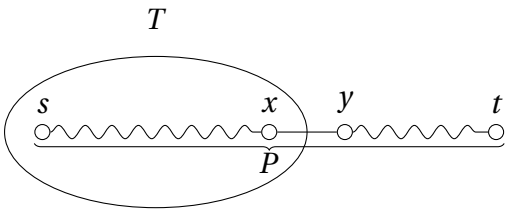
— T l'ensemble des sommets déjà traités, c'est-à-dire ceux que l'on a déjà extraits de la structure `a_traiter` ;
— $\delta(a, b)$ la distance réelle entre deux sommets a et b, c'est-à-dire la longueur d'un plus court chemin entre ces sommets ; et
— $\varepsilon(a, b)$ la distance estimée entre a et b, c'est-à-dire celle qu'on a calculée dans notre algorithme.

On va prouver par induction sur la taille de T que pour chaque sommet v que l'on est en train d'extraire lors de l'itération actuelle, la distance calculée est correcte, c'est-à-dire qu'on a $\varepsilon(s, v) = \delta(s, v)$.

1. cas de base : si $|T| = 0$, le sommet qu'on va extraire est la source s puisque $\varepsilon(s, s)$ est minimal et vaut 0, qui est également la valeur de $\delta(s, s)$.

2. induction : soit t le sommet que l'on est en train d'extraire ; on a $\varepsilon(s, t) \geq \delta(s, t)$, et pour prouver que notre estimation est correcte, il nous reste à montrer que $\varepsilon(s, t) \leq \delta(s, t)$.

   Soit P un plus court chemin reliant s à t ; ce chemin contient forcément une arête reliant un sommet x de T à un sommet y hors de T, puisque l'on part de $s \in T$ pour arriver à $t \notin T$:



On en conclut bien que $\varepsilon(s,t) \leq \delta(s,t)$, car

$$\begin{aligned}
\delta(s,t) &\geq \delta(s,y) && (t \text{ est plus loin et } G \text{ n'a pas de poids négatif})\\
&= \delta(s,x) + w(\{x,y\}) && (\{x,y\} \text{ appartient au plus court chemin } P)\\
&\geq \varepsilon(s,x) + w(\{x,y\}) && (\text{par induction } (x \text{ a déjà été traité}))\\
&\geq \varepsilon(s,y) && (\text{mise à jour de } \varepsilon(s,y) \text{ après avoir extrait } x)\\
&\geq \varepsilon(s,t). && (\text{si } \varepsilon(s,y) < \varepsilon(s,t), \text{ on aurait extrait } y, \text{ pas } t)
\end{aligned}$$

□

3.3.3 Complexité

On passe $O(|V|)$ fois dans la boucle principale, et ExtraireSommetLePlusProche(S, distances) coûte $O(|S|) = O(|V|)$; comme on passe également sur chaque arête deux fois (dans le sens (u,v) et dans le sens (v,u)), on obtient une implémentation en $O(|E|+|V|^2) = O(|V|^2)$.

3.4 Principe des algorithmes gloutons

Les algorithmes vus dans ce chapitre appartiennent à la catégorie des algorithmes dits *gloutons*, qui sont caractérisés par une stratégie consistant à effectuer à chaque étape le meilleur choix possible. Contrairement à ce que cette formulation pourrait laisser penser, et malgré le fait que les trois algorithmes vus dans ce chapitre fournissent un résultat correct, il est relativement rare que cette stratégie donne une solution optimale, mais il est utile de l'avoir à l'esprit car elle peut souvent fournir de bonnes approximations dans les cas où l'on ne peut espérer obtenir un algorithme exact efficace. Nous en reparlerons en section 7.2.

Bibliographie

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, et C. Stein, *Introduction to Algorithms*, MIT Press, 3ème édition, 2009.

[2] S. Even, *Graph Algorithms*, Cambridge University Press, 2ème édition, 2012.

[3] R. L. Graham et P. Hell, *On the history of the minimum spanning tree problem*, IEEE Annals of the History of Computing, 7 (1985), pages 43–57. http://www.math.ucsd.edu/~ronspubs/85_07_minimum_spanning_tree.pdf.

[4] M. Mareš, *The saga of minimum spanning trees*, Computer Science Review, 2 (2008), pages 165–221.

[5] S. S. Skiena, *The Algorithm Design Manual*, Springer, 2ème édition, 2008.