Chapitre 3

Algorithmique dans les graphes valués

Un *graphe valué* (resp. *digraphe valué*) est un couple (G, w) où $G = (V, E)$ est un graphe (resp. digraphe) et $w : E \to \mathbb{R}$ est une fonction de *poids* ou *longueur*. Les graphes et digraphes valués apparaissent en modélisant de nombreux problèmes où des lieux (villes, ordinateurs, serveurs, ...) sont reliés entre eux par des liens ayant des poids différents (distance kilométrique, coût, débit, ...). Notons qu'un graphe non valué peut-être vu comme un graphe dont toutes les arêtes ont poids 1.

Soit (G, w) un graphe (orienté ou non) valué. Pour tout sous-graphe H de G, le *poids* de H, noté $w(H)$, est la somme des poids des arêtes de H. En particulier, si P est un chemin le poids de P, $w(P)$ appelé communément *longueur* de P est la somme des poids (longueurs) de ses arêtes. La *distance* entre deux sommets u et v, notée $dist_G(u, v)$, est la longueur minimale d'un plus court (i.e. de longueur minimale) (u, v)-chemin. Dans le cas d'un graphe non-valué, cela correspond au plus petit nombre d'arêtes d'un (u, v)-chemin.

Dans le cas d'un graphe non-orienté G, $dist_G(u, v) = dist_G(v, u)$ alors que ce n'est en général pas le cas pour les graphes orientés. Cependant que le graphe G soit orienté ou non, $dist_G$ vérifie l'*inégalité triangulaire* : quels que soient trois sommets x, y et z, $dist_G(x, z) \leq dist_G(x, y) + dist_G(y, z)$.

3.1 Calcul de plus courts chemins

Etant donné un graphe ou digraphe valué (G, w), un des problèmes qui apparaît le plus souvent est le calcul de $dist_G(u, v)$ et la recherche d'un plus court (u, v)-chemin. Dans les cas où le graphe n'est pas valué, on peut facilement résoudre ce problème. Pour touver le plus court (u, v)-chemin, il suffit d'effectuer un parcours en largeur depuis u. Le (u, v)-chemin dans l'arbre en largeur ainsi construit est un plus court chemin. Nous allons maintenant des algorithmes pour résoudre ce problème dans le cas général. Pour cela, nous résolvons en fait un problème plus large :

Problème 3.1 (Arbre des plus courts chemins)**.** Étant donné un graphe valué (G, l) et un sommet origine r, on souhaite déterminer un arbre T de G (enraciné en r) tel que $\forall x \in V(G), dist_G(r, x) = dist_T(r, x)$

Un arbre solution est appelé *arbre des plus courts chemins.*

3.1.1 Algorithme de Dijkstra

L'algorithme de Dijkstra est basé sur le principe suivant : Soit $S \subset V(G)$ contenant r et soit $\bar{S} = V(G) \setminus S$. Si $P = (r, s_1, \ldots, s_k, \bar{s})$ est un plus court chemin de r à $\bar{S}$ alors clairement $s_k \in S$ et P est un plus court chemin de r à $\bar{s}$. Ainsi

$$dist(r, \bar{s}) = dist(r, s_k) + l([s_k, \bar{s}])$$

et la distance de r à $\bar{S}$ est donnée par la formule

$$dist(r, \bar{S}) = \min_{u \in S, v \in \bar{S}} \{dist(r, u) + l([u, v])\}$$

Afin d'éviter de faire trop de calculs, tout au long de l'algorithme, on associe à chaque sommet $v \in V(G)$ une fonction $d'(v)$ qui est une borne supérieure pour $dist(r, v)$ et un sommet $p(v)$ qui est le père potentiel de v dans l'arbre. A chaque étape i, nous avons :

$$\begin{aligned} d'(v) &= dist(r, v) \text{ si } v \in V(T_i) \\ d'(v) &= \min_{u \in V(T_{i-1})} \{dist(r, u) + l([u, v])\} \text{ si } v \in \overline{V(T_i)} \end{aligned}$$

Algorithme 3.1 (Dijkstra)**.**

1. Initialiser $d'(r) := 0$ et $d'(v) := \infty$ si $v \neq r$. T_0 est l'arbre réduit au sommet r, $u_0 := r$ et $i := 0$.
2. Pour tout $v \in \overline{V(T_i)}$, si $d'(u_i) + l([u_i, v]) \leq d'(v)$, faire $d'(v) := d'(u_i) + l([u_i, v])$ et $p(v) := u_i$.
   Calculer $\min\{d'(v) \mid v \in \overline{V(T_i)}\}$. Soit u_{i+1} un sommet pour lequel le minimum est atteint. Soit T_{i+1} l'arbre obtenu en ajoutant l'arête $[p(u_{i+1}), u_{i+1}]$.
3. Si $i = |V| - 1$, rendre T_i, sinon $i := i + 1$ et aller Etape 2.

Remarque 3.2. L'algorithme ne fonctionne pas s'il y a des arêtes de longueur négative.

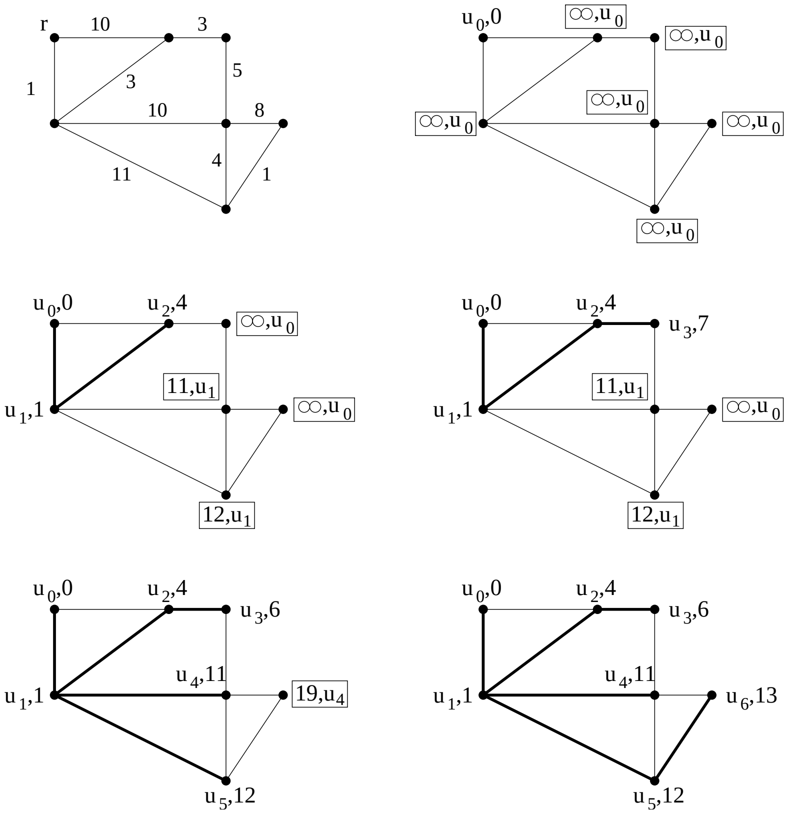


FIG. 3.1 – Déroulement de l'algorithme de Dijkstra

Complexité de l'Algorithme de Dijkstra :

A chaque sommet, on associe une étiquette temporaire qui correspond à $(d'(v), p(v))$. Celles-ci sont encadrées Figure 3.1. On effectue :

- au plus $|E|$ mises à jour d'étiquettes ;
- $|V|$ recherches du sommet pour lequel $d'(v)$ est minimum et suppressions de son étiquette.

La complexité de l'algorithme va dépendre du choix de la structure de donnée stockant les étiquettes : si c'est une liste, elle est de $|E||V| + |V|^2$; si c'est un arbre binaire équilibré $|E| \log |V| + |V| \log |V|$. On peut aussi utiliser une pile de Fibbonacci pour laquelle l'extraction du minimum s'effectue en temps log, et l'insertion en temps amorti $O(1)$, on obtient alors une complexité en $O(|V| \log |V| + |E|)$.

3.1.2 Algorithme de Bellmann et Ford

L'algorithme effectue n itérations, et attribue une étiquette $h(v)$ à tout sommet ; à l'itération i, $h(v)$ est la longueur du plus court chemin utilisant au plus i arêtes i entre r et v .

Notons qu'il existe toujours une plus courte marche utilisant moins de V arêtes entre r et v (sinon le chemin contiendrait un cycle de poids négatif).

Algorithme 3.2 (Bellmann et Ford)**.**
1. Initialisation : $h(r) := 0$, $h(v) := +\infty, \forall v \neq r$.
2. Pour $i = 0, i < |V|$ faire : $\forall v \in V(G), h(v) := \min(h(v), \min\{h(u) + l([u, v]), [u, v] \in E\})$.
3. Retourner $d(r, v) = h(r, v)$.

Complexité de l'Algorithme de Bellmann et Ford :
Chaque itération va coûter $|E|$ (L'algorithme examine toutes les arêtes) ; la complexité totale est donc de $|E||V|$.

L'algorithme fonctionne même si certaines longueurs d'arêtes sont négatives. Il détecte aussi l'existence d'un cycle de longueur négative : il existe un circuit de poids négatif si et seulement si après la $|V|^{\text{eme}}$ itération les étiquettes h peuvent diminuer. Enfin si lors d'une itération aucun $h(v)$ ne diminue alors $h(v) = d(r, v)$. On peut donc améliorer quelque peu l'algorithme en ne poursuivant l'itération que si $h(v)$ devient $\min\{h(u) + l([u, v], [u, v] \in E\}$ pour au moins un sommet. L'algorithme se déroule alors en $L|E|$ où L est le nombre maximum d'arêtes dans un plus court chemin.

3.2 Arbre couvrant de poids minimum

Un autre problème récurrent est le suivant : étant donné un graphe valué connexe, quel est le sous-graphe connexe de plus petit poids. Si tous les poids sont positifs, comme tout graphe connexe possède un arbre couvrant (Corollaire 1.5), le problème revient à trouver un arbre couvrant de poids minimum.

Problème 3.3 (Arbre de poids min)**.** Étant donnés un graphe valué (G, w), déterminer un arbre couvrant T de G de poids minimum, i.e. pour lequel $\sum_{e \in T} w(e)$ soit minimum.

Remarque 3.4. *Le problème analogue pour les digraphes, à savoir rechercher un sous-digraphe connexe de poids minimum dans un digraphe connexe, est beaucoup plus complexe. En effet, la notion d'arbre couvrant n'existe pas. En fait, ce problème est NP-difficile, même lorsque les poids sont identiques sur toutes les arêtes. Il existe cependant une 2-approximation.*

Les algorithmes pour trouver un arbre couvrant de poids minimum sont basés sur le fait qu'une arête transverse de poids minimum est contenue dans un arbre de poids minimum.

Définition 3.5. Soit $S \subset V(G)$. Une arête $e = xy$ est *S-transverse*, si $x \in S$ et $y \notin S$.

Lemme 3.6. *Soit $G = (V, E)$ un graphe, w une fonction de poids positive sur les arêtes et $S \subset V$. Si $e = s\bar{s}$ est une arête S-transverse de poids minimum alors il existe un arbre couvrant de poids minimum contenant e. En particulier, si e est une arête de poids minimum i.e. pour laquelle $w(e) = \min_{f \in E(G)} w(f) = w_{min}$ alors il existe un arbre couvrant de poids minimum contenant e.*

Démonstration. Considérons un arbre T qui ne contient pas l'arête e. Un chemin P relie s et $\bar{s}$ dans cet arbre. Au moins une arête de P, disons e', est S-transverse. Ainsi l'arbre $T' = (T - e') + e$ est de poids $w(T') = w(T) + w(e) - w(e') \leq w(T)$ car $w(e) \leq w(e')$. Donc si T était couvrant de poids minimum alors T' l'est aussi et $w(e) = w(e')$. □

Remarque 3.7. Si $w(e) = w_{min}$ alors on peut remplacer e par n'importe quelle arête de P et obtenir un arbre de poids au plus égal. Ainsi si T était aussi optimum alors toutes les arêtes de P étaient de poids w_{min}.

3.2.1 Algorithme de Prim

L'idée de l'algorithme de Prim est de faire croître l'arbre T de poids minimum sommet par sommet en ajoutant à chaque étape une arête $V(T)$-transverse de poids minimum. A chaque étape, l'ensemble E_T est l'ensemble des arêtes $V(T)$-transverses.

> **Algorithme 3.3** (Prim).
> 1. Initialiser l'arbre T à un sommet quelconque x et E_T est l'ensemble des arêtes incidentes à x.
> 2. Tant que $V(T) \neq V(G)$:
>   Trouver une arête $e \in E_T$ de poids minimum. Ajouter à T l'arête e et son extrémité y qui n'est pas dans T. Soit E_y les arêtes incidentes à y. Remplacer E_T par $(E_T \triangle E_y)$.

Complexité de l'algorithme de Prim :
Lors de l'algorithme, on doit effectuer au plus $|E|$ ajouts d'arête dans E_T et au plus $|E|$ suppressions d'arête. En effet, une arête e est supprimée lorsque ses deux extrémités sont dans $V(T)$. Comme $V(T)$ croit, e ne sera plus jamais ajoutée à E_T. Il faut également effectuer $|V|$ recherches de minimum dans E_T. Si on utilise une structure permettant l'insertion, la suppression et la recherche du minimum en temps log (par exemple des arbres binaires équilibrés) la complexité totale sera $O(|E| \log |E|)$.

3.2.2 Algorithme de Kruskal

L'algorithme de Kruskal est voisin de celui de Prim et sa validité est également assurée par le Lemme 3.6. L'idée de l'algorithme de Kruskal est de partir d'une forêt couvrante et de faire diminuer son nombre de composantes jusqu'à obtenir un arbre. Au départ, la forêt n'a pas d'arêtes et à chaque étape on rajoute une arête de poids minimum qui relie deux composantes.

Pour cela, nous avons besoin d'un mécanisme rapide permettant de tester si oui ou non u et v sont dans la même composante. Une façon de le faire est d'associer à chaque composante connexe la liste des sommets qu'elle contient, et à chaque sommet un pointeur $p(u)$ vers la composante connexe à laquelle il appartient.

> **Algorithme 3.4** (Kruskal).
> 1. Initaliser T : $V(T) := V(G)$, $E(T) := \emptyset$. Trier les arêtes dans l'ordre croissant et les placer dans une pile L ; Associer à chaque sommet u un pointeur $p(u)$ sur la liste $Comp(u) := \{u\}$.
> 2. Si L est vide terminer. Sinon dépiler l'arête de poids minimum $e = uv$;
> 3. Si $p(u) = p(v)$ (les sommets sont dans la même composante) aller en 2. Sinon $p(u) \neq p(v)$, ajouter e dans T.
> 4. Si $|Comp(u)| \geq |Comp(v)|$, effectuer $Comp(u) := Comp(u).Comp(v)$, et pour tout $w \in Comp(v)$, $p(w) := p(u)$.
>   Sinon $|Comp(u)| < |Comp(v)|$, effectuer $Comp(v) := Comp(v).Comp(u)$ et pour tout $w \in Comp(u)$, $p(w) := p(v)$.
> 5. Aller en 2.

Complexité de l'algorithme de Kruskal : La phase de tri coûte $|E| \log |E|$. Ensuite, il est clair que chaque arête est examinée au plus 1 fois et tester si l'arête doit ou non être ajoutée dans T coûte un nombre constant d'opérations.

Maintenant considérons les opérations effectuées afin de mettre à jour la structure de donnée, lorsqu'une arête est insérée dans l'arbre. On effectue $Comp(u) := Comp(u).Comp(v)$, cette opération prend un temps constant (à condition que la liste $L(u)$ dispose d'un pointeur sur son dernier élément). Notons aussi qu'il y a $|V| - 1$ concaténations de listes.

La seule phase non constante est la mise à jour des pointeurs qui associent à chaque sommet une composante. Considérons un sommet $x \in V(G)$ et estimons combien de fois le pointeur $p(x)$ est mis à jour au cours de l'algorithme. Quand $p(x)$ est mis à jour, la taille de la composante de x est au moins doublée. En effet, si l'on fusionne deux composantes, le pointeur $p(x)$ n'est mis à jour que si x appartient à la plus petite des deux composantes. Ainsi, après fusion, la taille de la composante de x a doublé. Comme au final x est dans une composante de taille $|V|$, nous concluons que le pointeur $p(x)$ est mis à jour au plus $\log_2(|V|)$ fois. Au total, il y a donc au plus $|V|\log_2|V|$ mises à jour de pointeurs.

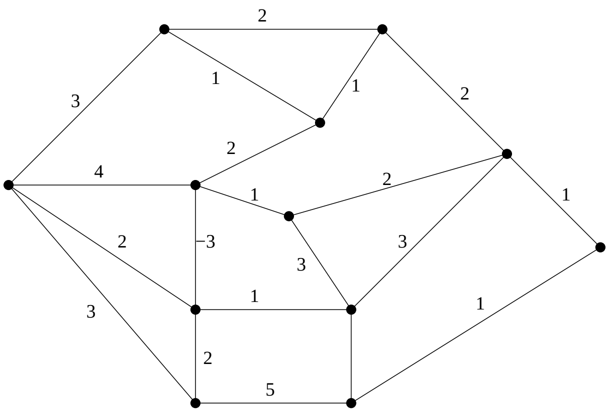
Comme $|V| \leq |E|$ la complexité globale est de $O(|E|\log|E|)$.

3.3 Exercices

Exercice 3.1.

On considère le graphe de la figure 3.1

1) Appliquer les algorithmes de recherche de l'arbre des plus courts chemins (Dijkstra et Bellmann Ford).
2) Appliquer les algorithmes de recherche de l'arbre de poids minimum.



Exercice 3.2. Soit (G, l) un graphe valué.
1) Montrer que si $l(e)$ est constante alors tout arbre des plus courts chemins est un arbre de poids minimum.
2) Donner un graphe et une fonction l pour laquelle il existe un arbre des plus courts chemins qui n'est pas un arbre de poids minimum.

Exercice 3.3. Quatre randonneurs imprudents sont surpris par la nuit et par l'orage. Pour rejoindre le refuge, ils doivent traverser un ravin par une frêle passerelle de cordes sur laquelle on ne peut passer qu'à deux personnes. De plus, traverser la passerelle requiert de porter une lampe de poche, afin de ne pas poser ses pieds dans le vide. Malheureusement les randonneurs n'ont qu'une seule lampe de poche et le ravin est trop large pour la jeter au-dessus de lui. Compte-tenu du vertige et de l'état de fatigue, les 4 randonneurs traversent la passerelle en 1, 2, 5 et 10 minutes. Lorsque deux randonneurs traverse la passerelle à deux, ils doivet tous deux être à côté de la lampe et avancent donc à la vitesse du plus lent. A l'aide d'un graphe, trouver le temps minimum qu'il faut aux randonneurs pour traverser le pont.

Exercice 3.4. Le *diamètre* d'un graphe valué G, est la distance maximale entre deux sommets : $diam(G) := \max\{dist_G(u, v) \mid u \in V(G), v \in V(G)\}$.
Montrer que l'algorithme suivant trouve le diamètre d'un arbre T.

1. Prendre un sommet quelconque u de T.
2. Trouver un sommet v à distance maximale de u (*par l'algorithme de Dijkstra par exemple*).
3. Trouver un sommet w à distance maximale de v.
4. Rendre $dist_T(v, w)$

Exercice 3.5. Soit G un graphe non-orienté, non-valué.

1) Montrer que si G a diamètre supérieur à 3 alors son complémentaire $\overline{G}$ a diamètre inférieur à 3.
2) En déduire que tout graphe auto-complémentaire (i. e. tel que $G = \overline{G}$) à diamètre au plus 3.

3) Pour $k = 0, 1, 2, 3$, donner un exemple de graphe auto-complémentaire de diamètre k.

Exercice 3.6. Soit T un arbre couvrant de poids minimum du graphe valué (G, w) et T' un autre arbre couvrant (pas nécessairement de poids minimum) de G. Montrer que T' peut être transformer en T par une suite déchange d'une arête de T' et d'une arête de T de telle manière qu'à chaque étape T' est un arbre et que le poids total n'augmente jamais.

Exercice 3.7. Le petit Attila a proposé l'algorithme suivant pour le problème de l'arbre couvrant de poids minimum : il considère les arêtes successivement par poids décroissant (non strictement croissant) et supprime celles qui se trouvent sur un cycle du graphe restant. Cet algorithme fournit-il la solution optimale au problème ? Justifiez votre réponse.

Exercice 3.8.
On souhaite déterminer un arbre couvrant de poids minimum approché. Pour cela, on étudie la question suivante : Que devient la complexité de l'algorithme quand on suppose que les arêtes ont un poids appartenant à un ensemble fixé de taille P ? (On pourra considérer pour commencer le cas où les arêtes ont toutes le même poids, ou des poids 1 et 2).

On suppose que les arêtes ont un poids entier appartenant à $[1, M]$ et l'on remplace une arête de poids $p \in [2^i, 2^{i+1}]$ par une arête de poids 2^i (on échantillonne les poids). Montrer que si on calcule un arbre couvrant de poids minimum avec les poids simplifiés l'arbre obtenu coûte au plus deux fois le minimum.

Qu'advient-il si on augmente le nombre d'échantillons ?

Exercice 3.9. Soit G un graphe valué 2-connexe. Montrer que tous les arbres couvrants sont de poids minimum si et seulement si toutes les arêtes ont le même poids. Donner un exemple de graphe valué connexe, pour lequel tous les arbres couvrants sont de même poids mais dont les poids des arêtes ne sont pas tous égaux.