Chapitre 5

Plus courts chemins (2)

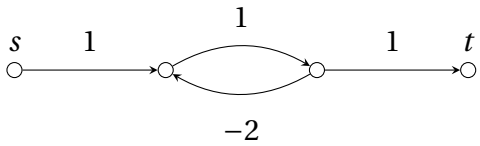
Sommaire

On a déjà vu un algorithme — celui de Dijkstra — permettant de calculer des plus courts chemins à partir d'une origine unique. Bien qu'il fonctionne à la fois pour les graphes orientés et les graphes non orientés, il ne permet pas de traiter les graphes contenant des arêtes de poids négatif (voir l'exemple 26 page 54; on verra dans l'exemple 39 que l'orientation ne change rien au problème). Ce chapitre présente l'algorithme de Bellman-Ford, qui est lui capable de traiter les graphes possédant des poids négatifs — tant que l'on n'a pas de *cycle de poids négatif* — ainsi que l'algorithme de Floyd-Warshall, qui permet de calculer les plus courts chemins entre toute paire de sommets d'un graphe bien plus efficacement qu'en lançant $|V|$ fois des algorithmes calculant les plus courts chemins à source unique.

Définition 24. Un *cycle (de poids) négatif* dans un graphe est un cycle dont la somme des poids des arêtes (ou des arcs) est négative.

Comme le montre l'exemple suivant, la présence d'un cycle négatif dans un graphe nous empêche de calculer un plus court chemin, puisque chaque nouveau passage dans le cycle nous permet d'améliorer la solution construite jusqu'ici.

Exemple 38. Voici un graphe comportant un cycle négatif; il est impossible de trouver un plus court chemin de n'importe quel sommet vers t sauf au départ de t.

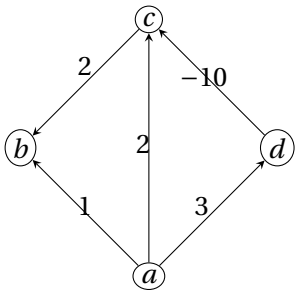


Remarquons que si le graphe est non orienté, la simple présence d'une arête de poids négatif entre deux sommets u et v suffit à nous empêcher de trouver une solution puisque cette arête nous donne un cycle négatif entre u et v (il suffit de faire des aller-retours entre u et v).

5.1 Cas des poids négatifs

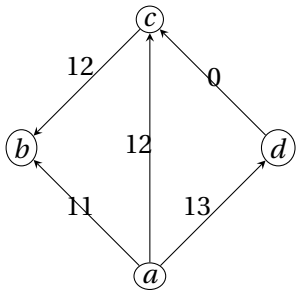
Si le graphe ne contient pas de cycle négatif mais qu'il contient des arcs de poids négatif, on ne peut plus garantir que l'algorithme de Dijkstra fonctionnera.

Exemple 39. (*a*) Un exemple de cas où l'algorithme de Dijkstra ne trouve pas le plus court chemin vers b en partant du sommet a : pour découvrir le chemin optimal $a \rightsquigarrow d \rightsquigarrow c \rightsquigarrow b$, on doit passer par c, mais comme le dernier sommet que l'on traite est d, on ne retournera plus sur c ensuite pour redécouvrir l'arc (c,b). (*b*) Le même graphe où l'on a tenté de contourner le problème en augmentant de $\min_{arc \in A(G)} w(arc)$ les poids de tous les arcs, en vue de les retirer après pour trouver une solution ; on peut vérifier que cette idée échoue.



étape	a	b	c	d
0	0	$+\infty$	$+\infty$	$+\infty$
1	0	1	2	3
2	0	1	2	3
3	0	1	2	3
4	0	1	-7	3

(*a*)

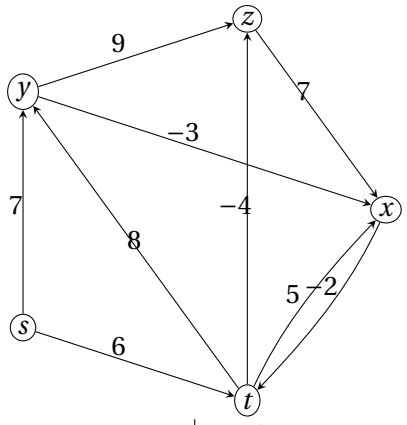


(*b*)

On utilise donc un autre algorithme, celui de Bellman-Ford, qui utilise comme celui de Dijkstra un tableau de distances estimées de la source à tous les autres sommets du graphe qu'on améliore progressivement. Ces estimations sont initialement infinies, sauf pour la source qui est à distance 0.

L'algorithme de Bellman-Ford recherche également des raccourcis entre les sommets ; mais contrairement à l'algorithme de Dijkstra, qui cherchait des chemins moins coûteux entre une source s et un sommet u en passant par un autre sommet v, l'algorithme de Bellman-Ford vérifie pour chaque arc (ou arête) s'il existe un chemin moins coûteux entre les sommets reliés. Si c'est le cas, on effectue la modification, et on examine l'ensemble des arêtes $|V|$ fois.

Exemple 40. Voici un graphe sur lequel on illustre le déroulement des étapes de l'algorithme de Bellman-Ford (en entier pour la première étape, et juste la version finale pour les suivantes) pour trouver les plus courts chemins issus de s. On examine ici chaque arc dans l'ordre lexicographique : $(s,t),(s,y),(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x)$. Pour chaque arc (u,v) de poids p, on vérifie si cet arc nous permet d'atteindre v avec un meilleur coût en "payant" p au départ de u ; si oui, on met à jour la distance correspondante entre s et v.



étape	s	t	x	y	z	après traitement des arcs issus de ...
0	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	(étape finale)
1	0	6	$+\infty$	7	$+\infty$	$s : st, sy$
	0	6	11	7	2	$t : tx, ty, tz$
	0	6	11	7	2	$x : xt$
	0	6	4	7	2	$y : yx, yz$
	0	6	4	7	2	$z : zx$
2	0	2	4	7	2	(étape finale)
3	0	2	4	7	-2	(étape finale)
4	0	2	4	7	-2	(étape finale)
5	0	2	4	7	-2	(étape finale)

Quand l'algorithme se termine, on obtient les chemins optimaux suivants :

chemins optimaux	
$s \to t$	$s \to y \to x \to t$
$s \to x$	$s \to y \to x$
$s \to y$	$s \to y$
$s \to z$	$s \to y \to x \to t \to z$

(fin exemple 40)

L'algorithme 26 présente une implémentation destinée au traitement d'un graphe orienté, qui renvoie le tableau des distances estimées. On peut facilement l'adapter au cas non orienté, mais cela aurait peu d'intérêt puisque si le graphe non orienté contient une arête de poids négatif, l'algorithme échoue. De plus, comme on le verra plus loin, la complexité de l'algorithme de Bellman-Ford est moins bonne que celle de l'algorithme de Dijkstra : on ne l'utilisera donc que quand on y sera obligé.

5.1.1 Correction

Avant d'examiner la correction de l'algorithme, demandons-nous comment et pourquoi il pourrait fonctionner.

Pourquoi parcourir tous les arcs exactement $|V|$ fois ? La raison est que la longueur maximale d'un chemin — en termes de nombres d'arcs ou d'arêtes — entre deux sommets est $|V| - 1$; toute itération supplémentaire permettrait donc de créer un cycle dans le chemin. Ceci explique aussi pourquoi la vérification de la présence d'un cycle négatif avant le renvoi des distances fonctionne : le seul moyen d'améliorer encore les distances serait d'utiliser un arc de poids négatif à insérer dans un cycle.

Algorithme 26 : BELLMANFORD(G, source)

Entrées : un graphe pondéré 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), ou NIL si le graphe contient un cycle négatif.

```
1 distances ← tableau(G.nombre_sommets(), +∞);
2 distances[source] ← 0;
  // parcourir chaque arc |V| fois
3 pour i allant de 1 à G.nombre_sommets()-1 faire
4 |   pour chaque (u, v, p) ∈ G.arcs() faire
5 |   |   distances[v] ← min(distances[v], distances[u] + p);
  // vérifier la présence d'un cycle négatif
6 pour chaque (u, v, p) ∈ G.arcs() faire
7 |   si distances[v] > distances[u] + p alors renvoyer NIL;
8 renvoyer distances;
```

5.1.2 Complexité

La complexité est assez simple à calculer : on examine intégralement l'ensemble des arcs $|V|$ fois, et les calculs effectués se font en temps constant. On a donc une complexité en $O(|V||A|)$ si on utilise une liste d'adjacence, et $O(|V|^3)$ si on utilise une matrice d'adjacence.

5.1.3 Améliorations

On peut décider de s'arrêter au moment où l'algorithme se stabilise : si les distances estimées n'ont subi aucun changement après l'itération i, alors elles n'en subiront plus non plus dans les suivantes et l'on peut donc s'arrêter.

5.2 Plus courts chemins entre toute paire de sommets

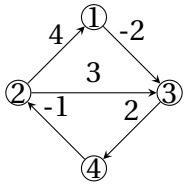
Dans certaines situations, il arrivera que l'on doive calculer des plus courts chemins entre plusieurs paires de sommets, et s'il faut répéter cette opération fréquemment, il vaudra mieux pré-calculer ces chemins. Par exemple, si un serveur stocke des cartes routières afin d'offrir un service GPS, on sait que ces cartes changeront très peu et très lentement, et que chaque utilisateur risque de lancer une requête avec des points de départ et d'arrivée différents. Il est donc utile de disposer d'algorithmes efficaces permettant de calculer les plus courts chemins entre toute paire de sommets. On peut bien sûr y arriver avec les algorithmes qu'on connaît déjà, en les lançant $|V|$ fois sur le graphe donné en entrée : les complexités résultantes seraient alors au mieux de $O(|V||E|\log|V|)$ pour la version optimisée de l'algorithme de Dijkstra, et de $O(|V|^2|E|)$ pour l'algorithme de Bellman-Ford. L'algorithme de Floyd-Warshall, présenté plus bas, résoud ce problème en temps $O(|V|^3)$.

L'algorithme de Floyd-Warshall procède de la façon suivante : à chaque itération de l'algorithme, on cherche à améliorer les distances entre chaque paire de sommets du graphe, mais en ne s'autorisant que les sommets intermédiaires d'indice 0, 1, 2, …, k pour un certain k fixé. Ainsi :

— à la première étape, les arcs du graphe nous donnent les paires de sommets accessibles, et toutes les autres sont à distance infinie ;
— à la deuxième étape, on cherche à améliorer la distance entre les sommets u et v en vérifiant si le chemin $u \to 0 \to v$ donne une distance plus petite ;
— à la troisième étape, on cherche à améliorer la distance entre les sommets u et v, donnée par un chemin de longueur ≤ 2, en vérifiant si l'utilisation du sommet intermédiaire 1 donne une distance plus petite ;
— et ainsi de suite jusqu'à ce que tous les candidats intermédiaires aient été examinés.

Les améliorations utilisent bien tous les sommets d'indice $\leq k$, mais la seule modification à l'étape k consiste à vérifier si le chemin $u \rightsquigarrow k \rightsquigarrow v$ est plus court que le chemin $u \rightsquigarrow v$, puisque les sommets d'indice inférieur ont déjà été utilisés.

Exemple 41. Voici les étapes de l'algorithme de Floyd-Warshall sur le graphe suivant (tiré de Wikipédia) :



À chaque étape, on examine comment raccourcir le plus court chemin connu jusqu'à présent entre chaque paire (u, v) de sommets en s'autorisant à passer par les sommets intermédiaires 0, 1, 2, …. La toute première étape consiste donc à passer en revue tous les arcs, puisque le sommet 0 n'existe pas. Dans les étapes suivantes, on ne montre que les chemins qu'on a réussi à améliorer, ainsi que l'évolution de la matrice de distances.

$k = 0$	$k = 1$	$k = 2$
① −2→ ③ ② 4→ ① ② 3→ ③ ③ 2→ ④ ④ −1→ ②	② 4→ ① −2→ ③	④ −1→ ② 4→ ① ④ −1→ ② 4→ ① −2→ ③
$0 \quad +\infty \quad -2 \quad +\infty$ $4 \quad 0 \quad 3 \quad +\infty$ $+\infty \quad +\infty \quad 0 \quad 2$ $+\infty \quad -1 \quad +\infty \quad 0$	$0 \quad +\infty \quad -2 \quad +\infty$ $4 \quad 0 \quad 2 \quad +\infty$ $+\infty \quad +\infty \quad 0 \quad 2$ $+\infty \quad -1 \quad +\infty \quad 0$	$0 \quad +\infty \quad -2 \quad +\infty$ $4 \quad 0 \quad 2 \quad +\infty$ $+\infty \quad +\infty \quad 0 \quad 2$ $3 \quad -1 \quad 1 \quad 0$
$k = 3$	$k = 4$	
① -2→ ③ 2→ ④ ② 4→ ① -2→ ③ 2→ ④	③ 2→ ④ -1→ ② ③ 2→ ④ -1→ ② 4→ ① ① -2→ ③ 2→ ④ -1→ ②	
$0 \quad +\infty \quad -2 \quad 0$ $4 \quad 0 \quad 2 \quad 4$ $+\infty \quad +\infty \quad 0 \quad 2$ $3 \quad -1 \quad 1 \quad 0$	$0 \quad -1 \quad -2 \quad 0$ $4 \quad 0 \quad 2 \quad 4$ $5 \quad 1 \quad 0 \quad 2$ $3 \quad -1 \quad 1 \quad 0$	

L'algorithme 27 présente une implémentation dans le cas orienté.

Algorithme 27 : FLOYDWARSHALL(G)

Entrées : un graphe orienté pondéré G.
Sortie : la matrice des distances entre toute paire de sommets du graphe.

```
1  n ← G.nombre_sommets();
2  distances ← matrice(n, n, +∞);
3  pour i allant de 0 à n − 1 faire
4  |  distances[i][i] ← 0;
5  pour chaque (u, v, p) ∈ G.arcs() faire
6  |  distances[u][v] ← p;
   // chercher les améliorations en passant par k = 0,1,2,...
7  pour k allant de 0 à n − 1 faire
8  |  pour i allant de 0 à n − 1 faire
9  |  |  pour j allant de 0 à n − 1 faire
10 |  |  |  distances[i][j] ← min(distances[i][j], distances[i][k]+distances[k][j]);
11 renvoyer distances;
```

5.2.1 Complexité

On voit assez rapidement que l'algorithme de Floyd-Warshall s'exécute en $O(|V|^3)$. Ceci est indépendant de la représentation choisie, car la seule étape où l'on accède aux arcs du graphe est lors de l'initialisation, ce qui nous coûte au pire $O(|V|^2)$ si l'on utilise une matrice d'adjacence, mais cette complexité est de toute façon dominée par les trois boucles imbriquées qui suivent.

5.2.2 Applications

L'algorithme de Floyd-Warshall nous permet de construire la fermeture transitive du graphe donné en entrée : lorsque son exécution se termine, les entrées non infinies de la matrice de distances nous disent quelles paires de sommets du graphe sont accessibles dans l'ordre examiné, et il nous suffit donc d'ajouter les arcs correspondants au graphe.