Plus courts chemins dans les réseaux

Problème de plus courts chemins

Considérons un réseau $R = (V, E, c)$, composé d'un graphe orienté $G = (V, E)$, généralement simple et connexe, et d'une pondération $c : E \to \mathbb{R}$.

Dans un tel réseau la **longueur** d'un chemin ou d'un circuit correspond à la somme des poids de ses arcs et non plus à leur nombre !

Il existe principalement quatre variantes de problèmes de plus courts chemins :

- **déterminer des plus courts chemins d'un sommet particulier s (la source) à tous les autres sommets du graphe** : problème de base ;
- **déterminer un plus court chemin d'un sommet source s (origine) à un sommet puits t (destination)** : pas plus facile que le problème de base ;
- **déterminer des plus courts chemins de tous les sommets du graphe jusqu'à un sommet puits t** : problème de base sur le graphe transposé ;
- **déterminer les plus courts chemins entre tous les couples de sommets du graphe** : il existe des algorithmes spécialisés, souvent plus efficaces que la répétition du problème de base depuis chaque sommet.

Poids négatifs et circuits à coût négatif

- Très souvent, le poids d'un arc correspond à une distance ou un temps et est naturellement supérieur (ou égal) à zéro. Dans de tels cas, dès qu'il existe un chemin entre deux sommets, il en existe un plus court.
- Dans d'autres situations, le poids d'un arc peut aussi bien être négatif que positif (typiquement lorsqu'il modélise un coût). Si le réseau possède un circuit dont la somme des poids des arcs est négative, le calcul des plus courts chemins n'est généralement plus possible. Un tel circuit est dit **à coût négatif** ou **de longueur négative** ou encore **absorbant** et, s'il est accessible depuis la source s, chaque tour supplémentaire diminue la longueur des chemins de s aux sommets du circuit (ainsi qu'aux sommets accessibles depuis le circuit).

  Imposer que les chemins cherchés soient élémentaires rend le problème difficile (problème NP-dur, sans algorithme polynomial connu).
- En l'absence de circuits à coût négatif le problème est bien défini et, dès qu'il existe un chemin, il en existe un plus court qui est simple et élémentaire (et comporte donc au plus $n - 1$ arcs si le réseau compte n sommets).

Le principe d'optimalité de Bellman

- Également valable pour de nombreux problèmes de décisions séquentielles, le principe d'optimalité de Bellman est à la base des techniques d'optimisation de la *programmation dynamique* ainsi que de nombreux algorithmes de calcul de plus courts chemins.
- Spécialisé aux problèmes de plus courts chemins, ce principe s'énonce très simplement :

Un plus court chemin est formé de plus courts chemins.

En effet, si C est un plus court chemin de s à t et si u est un sommet intermédiaire de ce plus court chemin, alors les sous-chemins de s à u et de u à t sont également des plus courts chemins (entre leurs extrémités respectives).

Équations de Bellman

- Notons λ_j la **distance** séparant s de j, c'est-à-dire la longueur d'un plus court chemin de s à j. Une conséquence directe du principe d'optimalité de Bellman est que, pour tout sommet j du réseau R, on doit avoir

$$\lambda_j \leq \lambda_i + c_{ij} \qquad \forall\, i \in Pred[j]\,.$$

  De plus, *il n'y aura égalité dans l'inéquation précédente que s'il existe un plus court chemin de s à j se terminant par l'arc (i, j).*

- Posant $\lambda_s = 0$, les longueurs des plus courts chemins de s aux autres sommets de R, si elles existent, sont alors solution des **équations de Bellman** :

$$\lambda_j = \min_{i \in Pred[j]} (\lambda_i + c_{ij}) \qquad \forall\, j \in V \setminus \{s\}\,.$$

Algorithme de Bellman (1)

L'algorithme de Bellman est un algorithme de programmation dynamique qui résout itérativement les équations du même nom.

- Définissons

$$\lambda_j^{(k)} = \text{la longueur d'un plus court chemin de } s \text{ à } j \text{ utilisant au plus } k \text{ arcs},$$

ces longueurs vérifient les équations de récurrence

$$\lambda_j^{(k)} = \min\left(\lambda_j^{(k-1)}, \min_{i \in Pred[j]}\left(\lambda_i^{(k-1)} + c_{ij}\right)\right), \qquad j \in V, k \geq 1,$$

car le plus court chemin de s à j utilisant au plus k arcs est soit le même que celui qui en utilise au plus $k-1$, soit formé d'un plus court chemin, d'au plus $k-1$ arcs, de s à un prédécesseur i de j auquel on ajoute l'arc (i, j).

Algorithme de Bellman (2)

- Partant des valeurs initiales

$$\lambda_j^{(0)} = \infty \text{ pour } j \neq s \quad \text{et} \quad \lambda_s^{(0)} = 0,$$

la méthode calcule, pour chaque sommet j, les valeurs $\lambda_j^{(1)}$ puis les valeurs $\lambda_j^{(2)}$ et ainsi de suite.

- Plus précisément, à l'itération $k \geq 1$
  - les valeurs $\lambda_j^{(k)}$ sont initialisées à partir des valeurs $\lambda_j^{(k-1)}$ obtenue à la fin de l'itération précédente ;
  - on passe ensuite en revue chaque arc (i, j) du graphe et on teste si

$$\lambda_j^{(k)} > \lambda_i^{(k-1)} + c_{ij}.$$

Si c'est le cas on met à jour $\lambda_j^{(k)}$ qui prend la valeur $\lambda_i^{(k-1)} + c_{ij}$.

Algorithme de Bellman (3)

- L'algorithme s'arrête
  - à la fin d'une itération k si aucune amélioration n'y a été trouvée (on a alors $\lambda_j^{(k)} = \lambda_j^{(k-1)}$ pour tous les sommets j),
  - mais au plus tard après n itérations complètes.

  En effet, si le réseau ne contient aucun circuit à coût négatif accessible depuis s les longueurs $\lambda_j^{(k)}$ doivent se stabiliser au plus tard après n itérations car il existe des plus courts chemins élémentaires formés d'au plus $n-1$ arcs.

  Si une amélioration a lieu pendant l'itération n, le réseau possède un circuit à coût négatif (accessible depuis le sommet s) et les plus courts chemins n'existent pas.

- La complexité de l'algorithme est en $O(m\,n)$ pour un réseau avec n sommets et m arcs. En pratique le nombre d'itérations nécessaires dépend fortement de l'ordre de parcours des arcs (si le réseau ne contient pas de circuits à coût négatif).

Algorithme de Bellman-Ford (1)

- L'algorithme de Bellman-Ford est une simplification de celui de Bellman, partageant la même complexité dans le pire des cas mais toujours au moins aussi performant en pratique.
- L'amélioration consiste à ne pas mémoriser les valeurs $\lambda_j^{(k)}$ pour chaque k mais a n'utiliser qu'une seule variable λ_j par sommet, chaque amélioration écrasant l'ancienne valeur. Le reste de l'algorithme ne change pas :
  - Partir des valeurs initiales

$$\lambda_s = 0 \qquad \text{et} \qquad \lambda_j = \infty \quad j \in V \setminus \{s\}.$$

  - Parcourir cycliquement tous les arcs du graphe et tester pour chaque arc (i, j) si la condition

$$\lambda_j > \lambda_i + c_{ij}$$

    est vérifiée. Si c'est le cas mettre à jour λ_j en posant

$$\lambda_j = \lambda_i + c_{ij}.$$

Algorithme de Bellman-Ford (2)

- Afin de pouvoir reconstruire les plus courts chemins (depuis la fin), on associe à chaque sommet j une seconde marque $p[j]$ égale au prédécesseur immédiat de j dans le meilleur chemin connu. Ces marques sont initialisées à la valeur `NULL` et si l'arc (i, j) permet de diminuer la valeur de λ_j, $p[j]$ est mis à jour en posant $p[j] = i$.
- Les conditions d'arrêt ne sont pas différentes de la méthode précédente :
  - si aucune diminution des marques λ_j n'a lieu pendant une itération complète ;
  - au plus tard après n itérations.

  S'il existe un plus court chemin de s à j, il en existe un qui est élémentaire et qui compte donc au plus $n - 1$ arcs. Toute amélioration pendant l'itération n est l'indication de l'existence d'un circuit à coût négatif (accessible depuis le sommet source s) dans R.
- Dans le pire des cas on fait n itérations de la boucle principale, chacune en temps $O(m)$ d'où une complexité en $O(m\,n)$ pour un réseau comptant n sommets et m arcs.

Algorithme de Bellman-Ford (1958)

Données : Un réseau orienté $R = (V, E, c)$, simple et connexe, et un sommet particulier s.

Résultat : Les longueurs λ_j des plus courts chemins de s à j ainsi que les prédécesseurs immédiats $p[j]$ dans ces chemins ou l'indication de l'existence d'un circuit de longueur négative accessible depuis s.

Début

(1) **Pour tout** $j \in V$ **poser** $\lambda_j := \infty$ et $p[j] := \texttt{NULL}$

(2) **Poser** $k := 0$, $Continuer := \text{vrai}$ et $\lambda_s := 0$

(3) **Tant que** $k < n$ **et** $Continuer = \text{vrai}$ **faire** // $n = |V|$

(4) $Continuer := \text{faux}$; $k := k + 1$

(5) **Pour chaque** arc $(i, j) \in E$ **faire**

(6) **Si** $\lambda_j > \lambda_i + c_{ij}$ **poser**

(7) $\lambda_j := \lambda_i + c_{ij}$; $p[j] := i$ et $Continuer := \text{vrai}$

(8) **Si** $Continuer = \text{faux}$

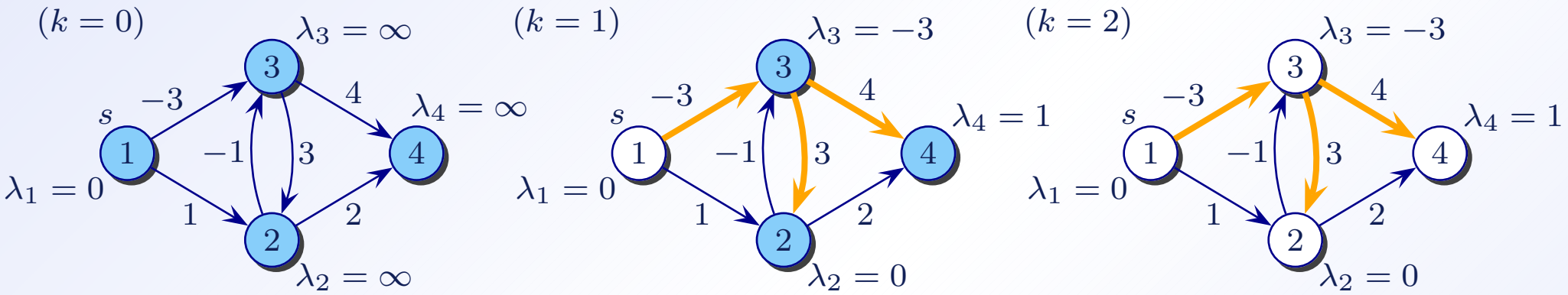
(9) Retourner les longueurs λ_j et les prédécesseurs $p[j]$

(10) **Sinon** // $Continuer = \text{vrai}$ et $k = n$

(11) Retourner « Le réseau contient un circuit à coût négatif accessible depuis s »

Fin

Exemple



Itér. k	Arc testé	Couples $(\lambda_j, p[j])$ pour chaque sommet 1	2	3	4	Valeur de *Continuer*
Valeurs de départ		(0,—)	(∞,—)	(∞,—)	(∞,—)	faux
1	(1,2)		$(1,1)$			vrai
	(1,3)			$(-3,1)$		
	(2,3)					
	(2,4)				$(3,2)$	
	(3,2)		$(0,3)$			
	(3,4)				$(1,3)$	
Valeurs de départ		(0,—)	$(0,3)$	$(-3,1)$	$(1,3)$	faux
2	(1,2)					
	(1,3)					
	(2,3)					
	(2,4)					
	(3,2)					
	(3,4)					

Algorithme de Dijkstra (1)

Considérons maintenant un réseau $R = (V, E, c)$ où $c : E \to \mathbb{R}_+$ est une pondération ***non négative*** des arcs du graphe $G = (V, E)$.

- Dans un tel réseau il n'y a évidemment pas de circuit de longueur négative et, dès qu'il existe un chemin de s à t, il existe un plus court chemin de s à t.
- On peut tirer partie de la non-négativité de la pondération c et déterminer les plus courts chemins de s aux autres sommets du graphe de proche en proche.
- L'initialisation des marques λ_j et $p[j]$ est la même que dans l'algorithme de Bellman-Ford.
- On tient à jour une liste L contenant tous les sommets pour lesquels le plus court chemin n'est pas encore connu (initialement L contient tous les sommets du graphe).
- À chaque itération, on choisit dans la liste L le sommet j possédant la marque λ_j la plus petite (L est en fait une queue de priorité). On le retire de la liste et on cherche à diminuer la marque de ses successeurs (avec le même test que dans Bellman-Ford).

Algorithme de Dijkstra (2)

On peut aussi rapprocher l'algorithme de Dijkstra de l'algorithme de Prim (on considère ici les versions orientées de ces deux algorithmes).

Tous deux sont des algorithmes gloutons et des parcours particuliers du graphe qui ne diffèrent que par la définition de la « priorité » λ_j associée au sommet j.

- Dans l'algorithme de Prim, la priorité λ_j du sommet j est égale au coût de connexion de ce sommet à l'arborescence actuelle (des chemins de section minimale depuis s).
- Dans l'algorithme de Dijkstra, la priorité λ_j du sommet j est égale à la longueur d'un plus court chemin de s à j dont tous les sommets intermédiaires sont déjà dans l'arborescence actuelle (des plus courts chemins depuis s).

Cette similitude se retrouve également dans la justification des deux algorithmes (il suffit de remplacer la notion de section minimale par celle de plus court chemin pour passer de l'une à l'autre).

Algorithme de Dijkstra (1959)

Données : Un réseau orienté $R = (V, E, c)$, simple et connexe, muni d'une pondération ***non négative*** $c : E \to \mathbb{R}_+$ des arcs et un sommet particulier s.

Résultat : Les longueurs λ_j des plus courts chemins de s à j ainsi que les prédécesseurs immédiats $p[j]$ dans ces chemins ($\lambda_j = \infty$ et $p[j] = \texttt{NULL}$ s'il n'existe pas de chemin de s à j dans R).

Début

(1) **Pour chaque** sommet $i \in V$ **poser** $\lambda_j := \infty$ et $p[j] := \texttt{NULL}$

(3) $\lambda_s := 0$

(4) $L := V$ // Queue de priorité contenant les sommets n'appartenant pas encore à l'arborescence des PCC

(5) **Tant que** $L \neq \varnothing$ **faire**

(6) Retirer de L le sommet i de plus petite marque λ_i

(7) **Si** $\lambda_i = \infty$ **sortir de la boucle** // Le sommet i et ceux encore dans L sont inaccessibles depuis s

(8) **Pour chaque** successeur $j \in Succ[i]$ **faire**

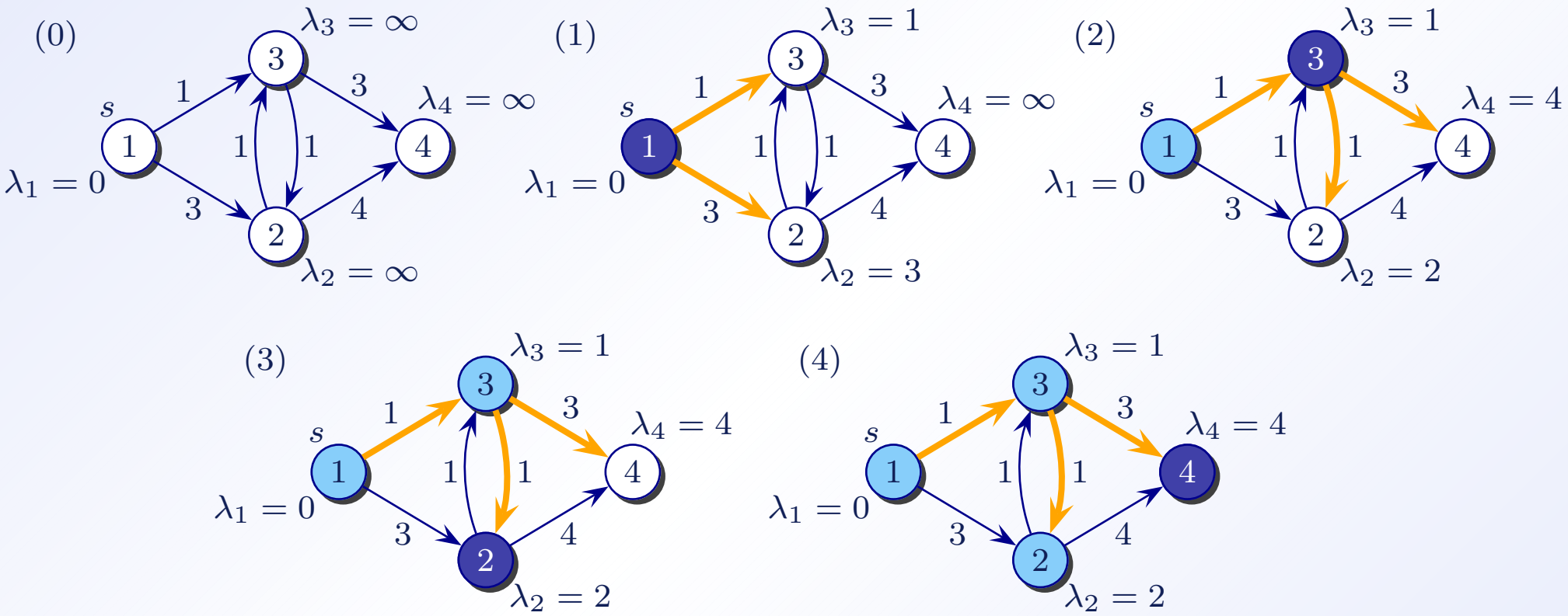
(9) **Si** $j \in L$ et $\lambda_j > \lambda_i + c_{ij}$ **poser** // Amélioration trouvée

(10) $\lambda_j := \lambda_i + c_{ij}$ et $p[j] := i$ // -> Mise à jour de la priorité

(11) Retourner les longueurs λ_j et les prédécesseurs $p[j]$

Fin

Exemple



Itér.	Sommet retiré de L	Couples $(\lambda_j, p[j])$ pour les sommets de L, en fin d'itér.			
		1	2	3	4
0		**(0,—)**	(∞,—)	(∞,—)	(∞,—)
1	1		(3,1)	**(1,1)**	(∞,—)
2	3		**(2,3)**		(4,3)
3	2				**(4,3)**
4	4				

Complexité de l'algorithme de Dijkstra

- La similitude entre les algorithmes de Prim et de Dijkstra s'étend également à la complexité des deux algorithmes qui est la même et dépend, rappelons-le, de la structure utilisée pour stocker et gérer la liste L.
  - La complexité de l'algorithme de Dijsktra est en $O(n^2)$ si L est gérée à l'aide d'un tableau contenant les priorités λ, les prédécesseurs immédiats p et une marque précisant si un sommet est encore dans L ou non.
  - Elle est en $O(m \log n)$ si L est une queue de priorité simple (un tas binaire).
  - Elle est en $O(m + n \log n)$ si L est gérée à l'aide d'un tas de Fibonacci.
- La complexité spatiale additionnelle est égale à l'espace nécessaire pour stocker la liste L et les différentes marques. Elle est donc en $O(n)$.

Plus courtes chaînes dans un réseau non orienté

- L'algorithme de Dijkstra (ainsi que celui de Bellman-Ford) peut également être appliqué à des réseaux non orientés. Il suffit pour cela de remplacer le parcours des successeurs par celui des sommets adjacents.

- L'algorithme calcule alors les longueurs des plus courtes chaînes entre la source s et les autres sommets du graphe ainsi que l'arbre des plus courtes chaînes depuis s.

- Évidemment dans le cas d'un réseau non orienté, il est impératif que le poids de chaque arête soit positif ou nul pour que le problème de la recherche de plus courtes chaînes soit bien défini. En effet toute arête de poids négatif permet de définir un cycle à coût négatif (en traversant deux fois de suite l'arête) et de construire des chaînes (non simples) de longueur arbitrairement petite.

Pondération constante

- Si tous les arcs d'un réseau ont le même poids positif c, la longueur d'un chemin n'est rien d'autre que le nombre d'arcs le composant, au facteur multiplicatif c près.
- Pour déterminer les plus courts chemins depuis une source s dans un tel réseau il suffit donc de minimiser le nombres d'arcs des chemins retenus.
- La construction de l'arborescence des plus courts chemins s'effectue alors à l'aide d'une simple *exploration en largeur* du graphe.
- L'utilisation de l'algorithme BFS, de complexité linéaire (en $O(n+m)$), est dans un telle situation clairement supérieure à l'application bête et méchante des algorithmes de Dijkstra ou de Bellman-Ford.

Tout ce qui précède s'applique également au cas d'un réseau non orienté dont toutes les arêtes ont le même poids positif c.

PCC entre tous les couples de sommets

Considérons maintenant le problème de la recherche de tous les plus courts chemins d'un réseau $R = (V, E, c)$. Sans perte de généralité nous supposerons que le graphe orienté $G = (V, E)$ est simple (et connexe). La pondération $c : E \to \mathbb{R}$ est, elle, *a priori* quelconque (et le réseau peut donc contenir des circuits à coût négatif).

Dans un tel réseau on cherche pour chaque sommet i

- la longueur d'un plus court chemin de i à chacun des autres sommets du réseau (autrement dit la **distance** entre i et chacun des autres sommets de R) ;
- l'arborescence des plus courts chemins depuis le sommet i.

Si le poids de certains arcs peut être négatif, il s'agira également de détecter l'existence de circuits à coût négatif (auquel cas les distances entre sommets et les arborescences des plus courts chemins n'existent pas).

Résolution par répétitions du problème de base

- Il est possible de calculer les longueurs des plus courts chemins entre tous les couples de sommets (et les arborescence associées) en appliquant n fois l'algorithme de Bellman-Ford et en variant à chaque fois la source s.

  Une telle approche permet de détecter la présence de circuits à coût négatif. Sa complexité dans le pire des cas est en $O(m\,n^2)$ qui peut donc varier entre $O(n^3)$ pour des réseaux connexes mais peu denses ($m \in \Theta(n)$) et $O(n^4)$ pour des réseaux très denses ($m \in \Theta(n^2)$).

  En pratique le temps de résolution dépend fortement de la structure du graphe, de l'ordre de parcours des arcs et des améliorations apportées à l'algorithme de Bellman-Ford.

- Si la pondération est non négative le réseau ne contient aucun circuit absorbant et on peut remplacer l'algorithme de Bellman-Ford par celui de Dijkstra.

  La complexité dans le pire des cas varie entre $O(n^2 \log n)$ pour des réseaux peu denses (en utilisant des tas binaires ou des tas de Fibonacci) et $O(n^3)$ pour des réseaux très denses (en utilisant des tas de Fibonacci voire des tableaux).

Représentation et résolution matricielles

Soit $R = (V, E, c)$ un réseau simple. On peut représenter R par une **matrice de poids** $C = (c_{ij})$ où

$$c_{ij} = \begin{cases} 0 & \text{si } i = j, \\ c_{ij} & \text{si } i \neq j \text{ et } (i,j) \in E, \\ \infty & \text{si } i \neq j \text{ et } (i,j) \notin E, \end{cases} \qquad i, j = 1, \ldots, n.$$

Les méthodes de résolution matricielles calculent, à partir de C, une **matrice de distances** $D = (d_{ij})$ où

$$d_{ij} = \text{ longueur d'un plus court chemin de } i \text{ à } j \text{ dans } R$$
$$(d_{ij} = \infty \text{ si } j \text{ n'est pas accessible depuis } i),$$

et une **matrice de prédécesseurs** $P = (p_{ij})$ où

$$p_{ij} = \text{ prédécesseur immédiat de } j \text{ dans un plus court chemin de } i \text{ à } j$$
$$(p_{ij} = \texttt{NULL} \text{ si } j \text{ n'est pas accessible depuis } i \text{ ou si } j = i).$$

Algorithme de Floyd-Warshall (1)

Proposé par Floyd (1962), mais basé sur un principe déjà utilisé par Roy (1959) et Warshall (1962) dans le cadre du calcul de la clôture (fermeture) transitive d'une relation, cette méthode repose sur la notion de *sommets intermédiaires* pour décomposer le calcul de tous les plus courts chemins d'un réseau R.

Plus précisément, à l'itération k, l'algorithme de Floyd-Warshall calcule les plus courts chemins (entre tous les couples de sommets) **n'utilisant que les sommets de l'ensemble $\{1, \ldots, k\}$ comme sommets intermédiaires**.

Notons

- $w_{ij}^{(k)}$ la longueur d'un plus court chemin de i à j dont tous les sommets intermédiaires sont compris entre 1 et k et $\boldsymbol{W}^{(k)}$ la matrice associée.
- $p_{ij}^{(k)}$ le prédécesseur immédiat de j dans le chemin de longueur $w_{ij}^{(k)}$.

L'initialisation de l'algorithme est très simple : $\boldsymbol{W}^{(0)} = \boldsymbol{C}$ et $p_{ij} = i$ si l'arc (i, j) existe ($p_{ij} = \texttt{NULL}$ sinon).

Algorithme de Floyd-Warshall (2)

À l'itération k on ne doit considérer que deux cas :

- le plus court chemin de i à j utilisant uniquement les sommets 1 à k comme sommets intermédiaires est le même que celui de l'itération précédente (utilisant uniquement les sommets 1 à $k-1$), on a alors

$$w_{ij}^{(k)} = w_{ij}^{(k-1)} \qquad \text{et} \qquad p_{ij}^{(k)} = p_{ij}^{(k-1)}$$

- le plus court chemin de i à j utilisant uniquement les sommets 1 à k comme sommets intermédiaires passe par le sommet k. Il se décompose donc en un plus court chemin de i à k suivi d'un plus court chemin de k à j, tous deux utilisant uniquement les sommets 1 à $k-1$ comme sommets intermédiaires. On a alors

$$w_{ij}^{(k)} = w_{ik}^{(k-1)} + w_{kj}^{(k-1)} \qquad \text{et} \qquad p_{ij}^{(k)} = p_{kj}^{(k-1)}$$

Algorithme de Floyd-Warshall (3)

- **Simplification des équations de récurrence** : tous les calculs peuvent être effectués en place (ceci simplifie également la gestion de l'espace mémoire). Une itération de l'algorithme consiste donc en une boucle de la forme

Pour chaque couple (i, j) de sommets **faire**
Si $w_{ij} > w_{ik} + w_{kj}$ **faire**
$w_{ij} := w_{ik} + w_{kj}$
$p_{ij} := p_{kj}$

- La complexité spatiale de l'algorithme est en $O(n^2)$ (mémoire nécessaire au stockage des deux matrices $\boldsymbol{W}$ et $\boldsymbol{P}$).
- **Détection des circuits à coût négatif** : si le réseau R contient un circuit absorbant certains éléments diagonaux de $\boldsymbol{W}$ vont prendre une valeur négative. Il suffit donc de tester l'apparition d'un tel événement (en cours d'algorithme ou à la fin des n itérations).
- La complexité de l'algorithme est en $O(n^3)$ (3 boucles « de 1 à n » imbriquées).

Algorithme de Floyd-Warshall (1962)

Données : Un réseau simple $R = (V, E, c)$ donné par sa matrice de poids $\boldsymbol{C}$.

Résultat : La matrice $\boldsymbol{W}$ des distances entre sommets ainsi que la matrice $\boldsymbol{P}$ des prédécesseurs immédiats dans les plus courts chemins ou l'indication que le réseau contient un circuit à coût négatif.

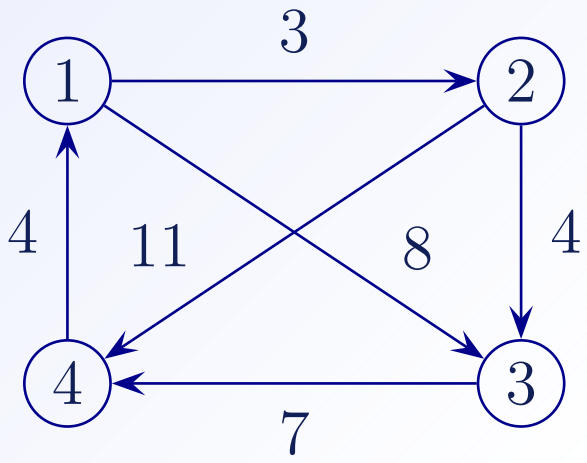
Début

(1) **Pour** i de 1 à n **faire**
(2) **Pour** j de 1 à n **faire**
(3) **Poser** $w_{ij} := c_{ij}$ et $p_{ij} := i$
(4) **Si** $i = j$ **ou si** $w_{ij} = \infty$ **poser** $p_{ij} :=$ `NULL`
(5) **Pour** k de 1 à n **faire**
(6) **Pour** i de 1 à n **faire** // et $i \neq k$
(7) **Pour** j de 1 à n **faire** // et $j \neq k$
(8) **Si** $w_{ij} > w_{ik} + w_{kj}$ **faire**
(9) **Poser** $w_{ij} := w_{ik} + w_{kj}$ et $p_{ij} := p_{kj}$
(10) **Si** $w_{ii} < 0$ **retourner** R contient un circuit à coût négatif passant par i
(11) **Retourner** la matrice de distances $\boldsymbol{W}$ et la matrice de prédécesseurs $\boldsymbol{P}$

Fin

Exemple (1)

Réseau $R = (V, E, c)$



Matrice de poids $\boldsymbol{C}$

$$\boldsymbol{C} = \begin{bmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & 4 & 11 \\ \infty & \infty & 0 & 7 \\ 4 & \infty & \infty & 0 \end{bmatrix}$$

Initialisation (Itération 0) :

$$\boldsymbol{W}^{(0)} = \begin{bmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & 4 & 11 \\ \infty & \infty & 0 & 7 \\ 4 & \infty & \infty & 0 \end{bmatrix}$$

$$\boldsymbol{P}^{(0)} = \begin{bmatrix} — & 1 & 1 & — \\ — & — & 2 & 2 \\ — & — & — & 3 \\ 4 & — & — & — \end{bmatrix}$$

Exemple (2)

$$\boldsymbol{W}^{(0)} = \begin{bmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & 4 & 11 \\ \infty & \infty & 0 & 7 \\ 4 & \infty & \infty & 0 \end{bmatrix} \qquad \boldsymbol{P}^{(0)} = \begin{bmatrix} — & 1 & 1 & — \\ — & — & 2 & 2 \\ — & — & — & 3 \\ 4 & — & — & — \end{bmatrix}$$

Itération 1 : Seul le sommet 1 est autorisé comme sommet intermédiaire.

Pendant l'itération k, la ligne k et la colonne k des matrices $\boldsymbol{W}$ et $\boldsymbol{P}$ ne subissent aucune modification.

$$\boldsymbol{W}^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & 4 & 11 \\ \infty & \infty & 0 & 7 \\ 4 & \mathbf{7} & \mathbf{12} & 0 \end{bmatrix} \qquad \boldsymbol{P}^{(1)} = \begin{bmatrix} — & 1 & 1 & — \\ — & — & 2 & 2 \\ — & — & — & 3 \\ 4 & \mathbf{1} & \mathbf{1} & — \end{bmatrix}$$

Pour les autres éléments de la matrice $\boldsymbol{W}$ on compare la valeur actuelle à la somme de l'élément en même ligne mais colonne k et de celui en ligne k et même colonne. Si la somme est plus petite que la valeur actuelle, on met à jour.

Exemple (2)

$$\boldsymbol{W}^{(0)} = \begin{bmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & 4 & 11 \\ \infty & \infty & 0 & 7 \\ 4 & \infty & \infty & 0 \end{bmatrix} \qquad \boldsymbol{P}^{(0)} = \begin{bmatrix} — & 1 & 1 & — \\ — & — & 2 & 2 \\ — & — & — & 3 \\ 4 & — & — & — \end{bmatrix}$$

Itération 1 : Seul le sommet 1 est autorisé comme sommet intermédiaire.

Pendant l'itération k, la ligne k et la colonne k des matrices $\boldsymbol{W}$ et $\boldsymbol{P}$ ne subissent aucune modification.

$$\boldsymbol{W}^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & 4 & 11 \\ \infty + \infty & 0 & 7 & \\ 4 & \mathbf{7} & \mathbf{12} & 0 \end{bmatrix} \qquad \boldsymbol{P}^{(1)} = \begin{bmatrix} — & 1 & 1 & — \\ — & — & 2 & 2 \\ — & — & — & 3 \\ 4 & \mathbf{1} & \mathbf{1} & — \end{bmatrix}$$

Pour les autres éléments de la matrice $\boldsymbol{W}$ on compare la valeur actuelle à la somme de l'élément en même ligne mais colonne k et de celui en ligne k et même colonne. Si la somme est plus petite que la valeur actuelle, on met à jour.

Exemple (2)

$$\boldsymbol{W}^{(0)} = \begin{bmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & 4 & 11 \\ \infty & \infty & 0 & 7 \\ 4 & \infty & \infty & 0 \end{bmatrix} \qquad \boldsymbol{P}^{(0)} = \begin{bmatrix} — & 1 & 1 & — \\ — & — & 2 & 2 \\ — & — & — & 3 \\ 4 & — & — & — \end{bmatrix}$$

Itération 1 : Seul le sommet 1 est autorisé comme sommet intermédiaire.

Pendant l'itération k, la ligne k et la colonne k des matrices $\boldsymbol{W}$ et $\boldsymbol{P}$ ne subissent aucune modification.

$$\boldsymbol{W}^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & 4 & 11 \\ \infty & \infty & 0 & 7 \\ 4 & 7 & 12 & 0 \end{bmatrix} \qquad \boldsymbol{P}^{(1)} = \begin{bmatrix} — & 1 & 1 & — \\ — & — & 2 & 2 \\ — & — & — & 3 \\ 4 & 1 & 1 & — \end{bmatrix}$$

Pour les autres éléments de la matrice $\boldsymbol{W}$ on compare la valeur actuelle à la somme de l'élément en même ligne mais colonne k et de celui en ligne k et même colonne. Si la somme est plus petite que la valeur actuelle, on met à jour.

Exemple (3)

$$\boldsymbol{W}^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & 4 & 11 \\ \infty & \infty & 0 & 7 \\ 4 & 7 & 12 & 0 \end{bmatrix} \qquad \boldsymbol{P}^{(1)} = \begin{bmatrix} — & 1 & 1 & — \\ — & — & 2 & 2 \\ — & — & — & 3 \\ 4 & 1 & 1 & — \end{bmatrix}$$

$$\boldsymbol{W}^{(2)} = \begin{bmatrix} 0 & 3 & 7 & 14 \\ \infty & 0 & 4 & 11 \\ \infty & \infty & 0 & 7 \\ 4 & 7 & 11 & 0 \end{bmatrix} \qquad \boldsymbol{P}^{(2)} = \begin{bmatrix} — & 1 & 2 & 2 \\ — & — & 2 & 2 \\ — & — & — & 3 \\ 4 & 1 & 2 & — \end{bmatrix}$$

$$\boldsymbol{W}^{(3)} = \begin{bmatrix} 0 & 3 & 7 & 14 \\ \infty & 0 & 4 & 11 \\ \infty & \infty & 0 & 7 \\ 4 & 7 & 11 & 0 \end{bmatrix} \qquad \boldsymbol{P}^{(3)} = \begin{bmatrix} — & 1 & 2 & 2 \\ — & — & 2 & 2 \\ — & — & — & 3 \\ 4 & 1 & 2 & — \end{bmatrix}$$

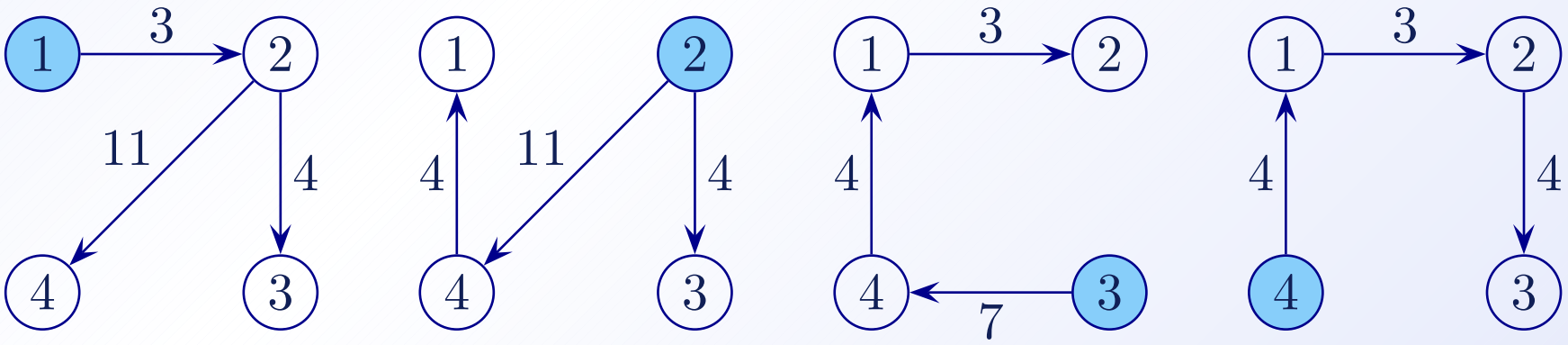
$$\boldsymbol{W}^{(4)} = \begin{bmatrix} 0 & 3 & 7 & 14 \\ 15 & 0 & 4 & 11 \\ 11 & 14 & 0 & 7 \\ 4 & 7 & 11 & 0 \end{bmatrix} \qquad \boldsymbol{P}^{(4)} = \begin{bmatrix} — & 1 & 2 & 2 \\ 4 & — & 2 & 2 \\ 4 & 1 & — & 3 \\ 4 & 1 & 2 & — \end{bmatrix}$$

Exemple (4)

Matrices finales :

$$\boldsymbol{W} = \begin{bmatrix} 0 & 3 & 7 & 14 \\ 15 & 0 & 4 & 11 \\ 11 & 14 & 0 & 7 \\ 4 & 7 & 11 & 0 \end{bmatrix} \qquad \boldsymbol{P} = \begin{bmatrix} — & 1 & 2 & 2 \\ 4 & — & 2 & 2 \\ 4 & 1 & — & 3 \\ 4 & 1 & 2 & — \end{bmatrix}$$

Arborescences des plus courts chemins depuis chacun des sommets :



Algorithme de Dantzig (1966)

L'algorithme de Dantzig :

- résout le même problème que l'algorithme de Floyd-Warshall (calcul des plus courts chemins pour tous les couples de sommets et détection de circuits absorbants) ;
- partage la même complexité asymptotique en $O(n^3)$ (et un espace mémoire en $O(n^2)$) ;
- mais repose sur une décomposition différente du problème :
  - à l'itération k, la méthode calcule les plus courts chemins entre tous les couples de sommets du **sous-graphe G_k engendré par les k premiers sommets** ;
  - la résolution du problème de l'étape k se base évidemment sur les résultats obtenus à l'itération précédente ;
  - après n itérations, le sous-graphe G_n n'est rien d'autre que le graphe de départ et les dernières valeurs calculées correspondent aux longueurs des plus courts chemins entre tous les couples de sommets du réseau.

Algorithme de Johnson (1977)

- Si les poids des arcs d'un réseau $R = (V, E, c)$ sont tous non négatifs, n applications de l'algorithme de Dijkstra permettent de déterminer tous les plus courts chemins de R en un temps $O(m\,n\log n)$ (ou même $O(m\,n + n^2\log n)$).

  Si le graphe est peu dense ($m \in O(n)$) le gain par rapport aux deux méthodes précédentes peut être non négligeable.
- L'algorithme de Johnson permet le calcul de tous les plus courts chemins en un temps $O(m\,n\log n)$ (resp. $O(m\,n + n^2\log n)$) même lorsque le réseau contient des arcs de poids négatif :
  1) Une seule application (sur un réseau à peine modifié) de Bellman-Ford permet
     - de détecter l'existence d'un circuit à coût négatif dans R, auquel cas l'algorithme s'arrête ;
     - de calculer, sinon, une **fonction potentiel** $\delta : V \to \mathbb{R}$.
  2) La fonction potentiel est ensuite utilisée par calculer une nouvelle pondération *non négative des arcs de R qui conserve les plus courts chemins*.

Algorithme de Johnson : calcul des potentiels

- La première étape de l'algorithme de Johnson consiste à modifier le réseau R
  - en lui ajoutant un sommet auxiliaire 0 (on suppose les sommets de R numérotés de 1 à n) ;
  - en reliant ce sommet à tous les autres sommets du graphe par des arcs $(0, i)$ de poids nul.

- Sur le réseau auxiliaire obtenu on applique l'algorithme de Bellman-Ford (ou une variante) avec comme source le sommet auxiliaire 0.
  - Si l'algorithme détecte un circuit à coût négatif le calcul s'arrête car les plus courts chemins n'existent pas (il est alors possible d'identifier un circuit absorbant en utilisant les résultats de l'exercice 5.4) ;
  - sinon on associe à chaque sommet i du réseau initial R un **potentiel δ_i** égal à la longueur d'un plus court chemin du sommet auxiliaire 0 au sommet i.

Algorithme de Johnson : calcul des coûts réduits

À partir de la fonction potentiel δ on calcule une nouvelle pondération

$$c' : E \to \mathbb{R}_+$$

des arcs du réseau (technique de *reweighting*). Ces nouveaux poids, souvent appelés **coûts réduits** (*reduced costs*), sont donnés par

$$\boxed{c'_{ij} = c_{ij} + \delta_i - \delta_j \qquad \forall\,(i,j) \in E}.$$

Théorème. Pour tout arc $(i,j) \in E$ on a $c'_{ij} \geq 0$. De plus $c'_{ij} = 0$ si et seulement si l'arc (i,j) appartient à un plus court chemin depuis la source 0 du réseau auxiliaire.

DÉMONSTRATION. Les distances δ vérifient les équations de Bellman :

$$\delta_j \leq \delta_i + c_{ij} \qquad \Longleftrightarrow \qquad c_{ij} + \delta_i - \delta_j \geq 0$$

pour chaque arc (i,j) et il n'y a égalité dans les inéquations précédentes que si l'arc (i,j) appartient à un PCC de 0 à j.

Algorithme de Johnson : propriétés des coûts réduits

Théorème. Tout chemin du sommet i au sommet j voit sa longueur augmenter de $\delta_i - \delta_j$ lorsque l'on remplace la pondération c par la pondération c'.

DÉMONSTRATION. Soit $(i_0 = i, i_1, \ldots, i_p = j)$ la suite des sommets d'un chemin de i à j. La longueur de ce chemin dans le réseau initial (avec la pondération c) est

$$L = \sum_{k=0}^{p-1} c_{i_k, i_{k+1}}$$

alors que pour la pondération c', sa longueur est

$$L' = \sum_{k=0}^{p-1} c'_{i_k, i_{k+1}} = \sum_{k=0}^{p-1} \left(c_{i_k, i_{k+1}} + \delta_{i_k} - \delta_{i_{k+1}} \right) = \sum_{k=0}^{p-1} c_{i_k, i_{k+1}} + \delta_{i_0} - \delta_{i_p} = L + \delta_i - \delta_j.$$

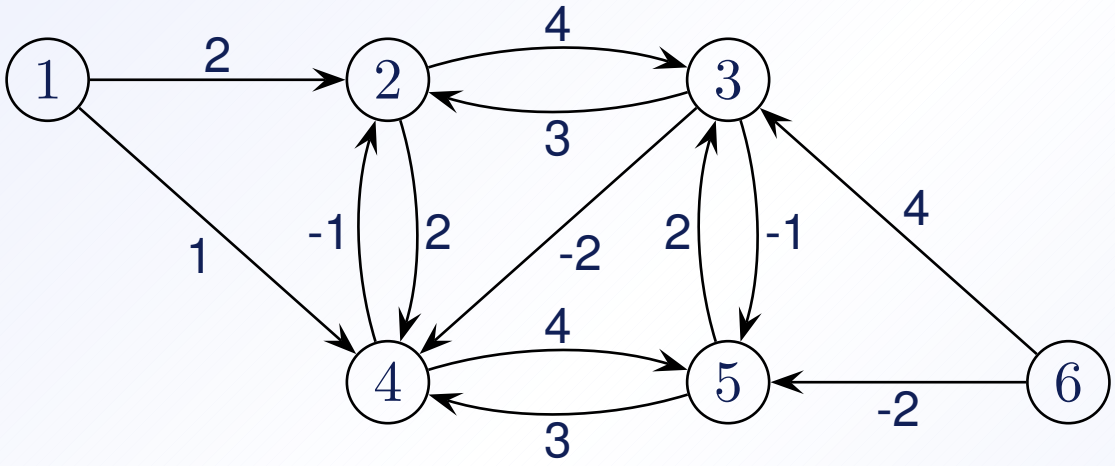
Corollaire. Un chemin de i à j est un plus court chemin de i à j dans $R = (V, E, c)$ si et seulement s'il s'agit d'un plus court chemin de i à j dans $R' = (V, E, c')$.

Algorithme de Johnson : calcul des PCC

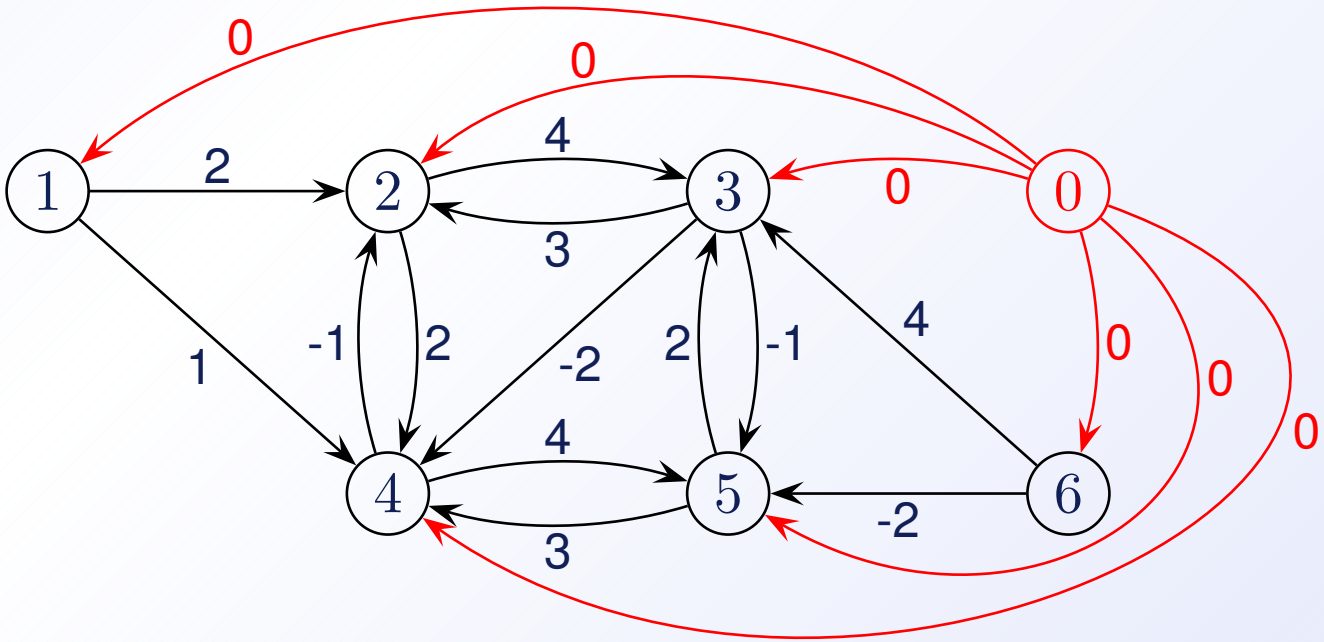
- Le réseau $R' = (V, E, c')$ possédant un pondération non négative, on peut lui appliquer autant de fois que voulu l'algorithme de Dijkstra en variant la source à chaque application.
  - Les arborescences obtenues sont des arborescences de plus courts chemins aussi bien dans R que dans R'.
  - Les distances calculées sur le réseau R' ne sont pas correctes pour le réseau R. En effet, lors de la « repondération » du graphe, chaque chemin de i à j a vu sa longueur augmenter de $\delta_i - \delta_j$. Pour obtenir les distances dans R il faut donc soustraire $\delta_i - \delta_j$ à celles calculées dans R'.
- Complexité globale pour le calcul des PCC entre tous les couples de sommets en $O(m\,n + n^2 \log n)$ ($= O(n^2 \log n)$ si R est peu dense) :
  - Construction du réseau auxiliaire : $O(n)$
  - Calcul des potentiels à l'aide de Bellman-Ford : $O(m\,n)$
  - Calcul des nouveaux poids : $O(m)$
  - Calcul des plus courts chemins dans R' : $O(m\,n + n^2 \log n)$
  - Correction des distances : $O(n^2)$

Exemple (1)

Réseau $R = (V, E, c)$ (réseau de l'exercice 5.2) :

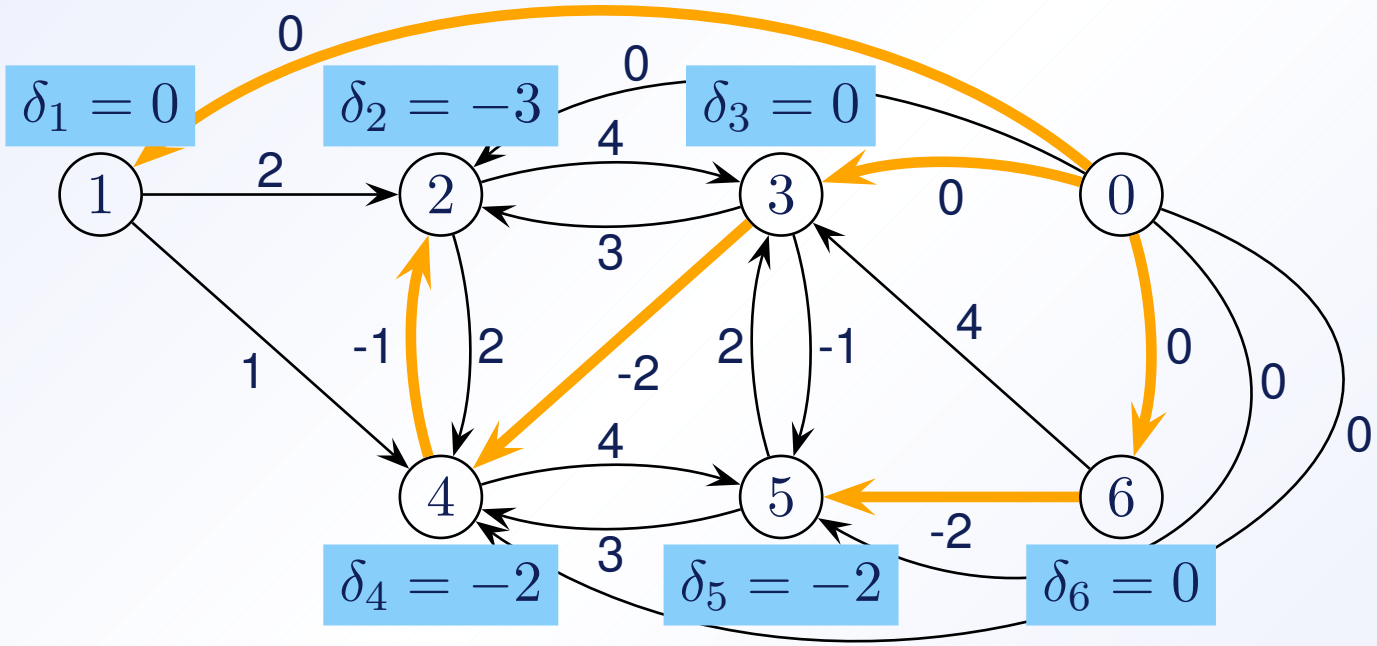


Réseau auxiliaire :

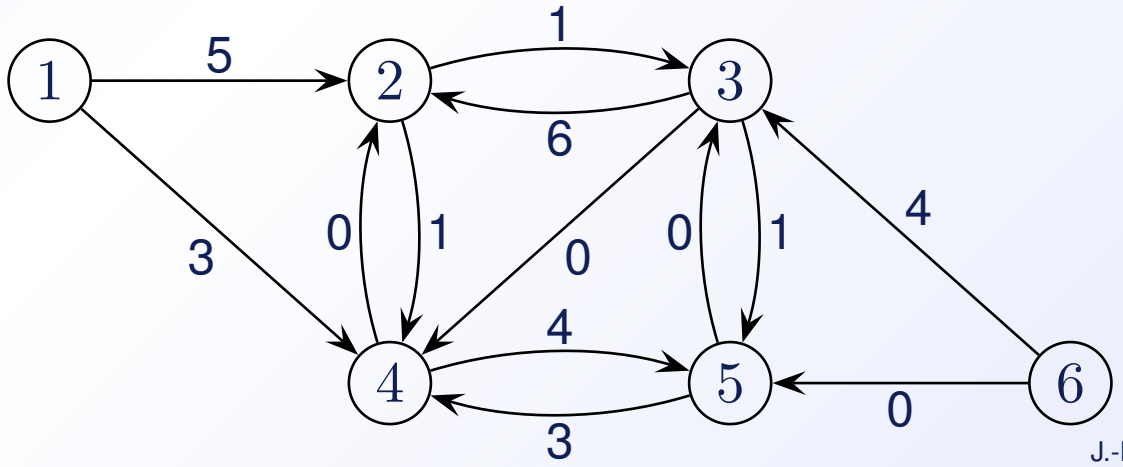


Exemple (2)

Résultat de l'application de Bellman-ford sur le réseau auxiliaire depuis le sommet 0 :

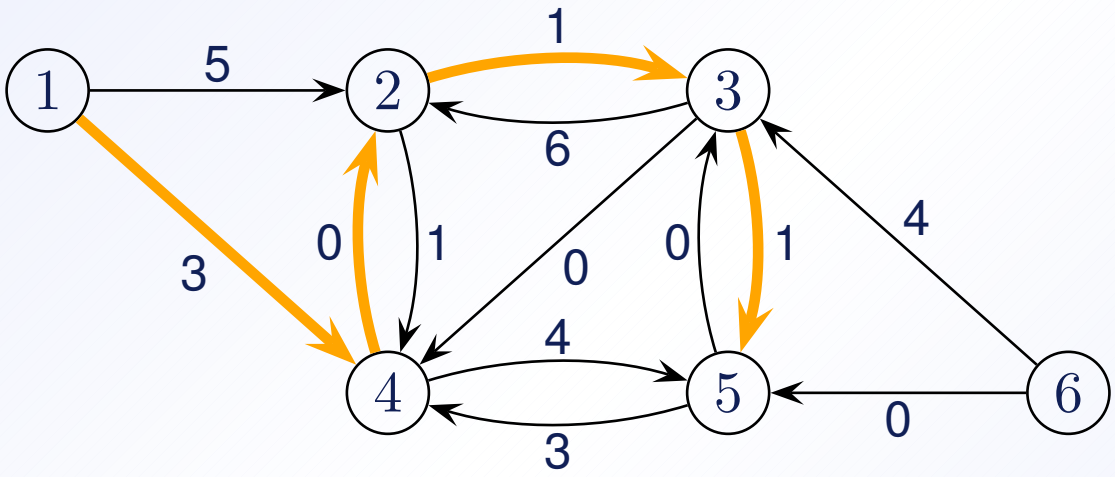


Réseau $R' = (V, E, c')$ avec la nouvelle pondération non négative des arcs :



Exemple (3)

Résultat de l'application de Dijkstra sur le réseau R' depuis le sommet 1 :



Itér.	Sommet retiré de L	Couples $(\lambda'_j, p[j])$ pour les sommets de L, en fin d'itér.					
		1	2	3	4	5	6
0		**(0,—)**	$(\infty$,—)	$(\infty$,—)	$(\infty$,—)	$(\infty$,—)	$(\infty$,—)
1	1		(5,1)	||	**(3,1)**	||	||
2	4		**(3,4)**	||		(7,4)	||
3	2			**(4,2)**		(7,4)	||
4	3					**(5,3)**	||
5	5						||

Exemple (4)

Calcul des longueurs des plus courts chemins dans le réseau initial R :

- Tout chemin de i à j voit sa longueur augmenter de $\delta_i - \delta_j$ lorsque l'on remplace la pondération c par la pondération c'.
- Si λ'_j dénote la longueur d'un plus court chemin de 1 à j dans R', la longueur λ_j de ce chemin dans R est $\lambda_j = \lambda'_j - \delta_1 + \delta_j = \lambda'_j + \delta_j$ (car $\delta_1 = 0$ ici).

Sommet j	1	2	3	4	5	6
Distance λ'_j dans R'	0	3	4	3	5	∞
Potentiel δ_j	0	-3	0	-2	-2	0
Distance λ_j dans R	0	0	4	1	3	∞