Université Paris Cité

Plus court chemin

CM nº5 — Algorithmique (AL5)

Matěj Stehlík

20/10/2023

Les files de priorité

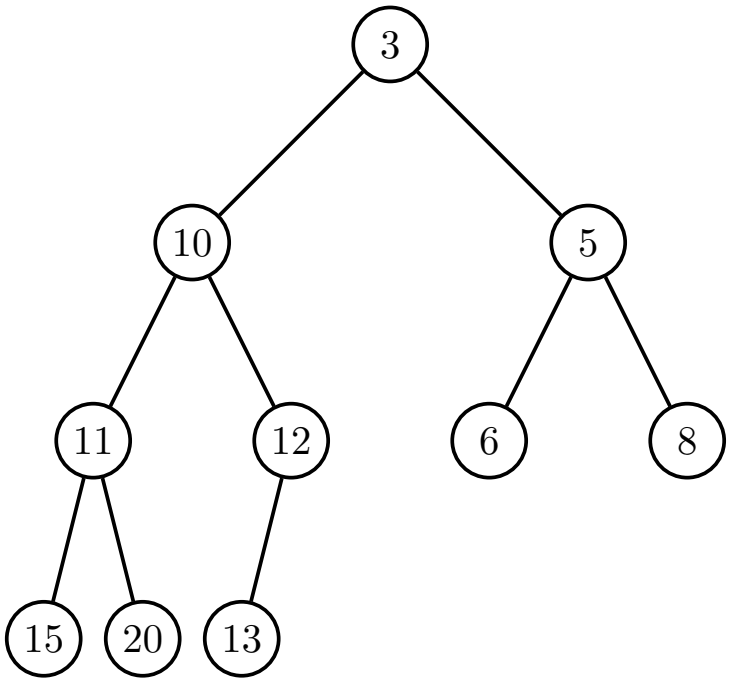
Une *file de priorité* est un type abstrait élémentaire sur laquelle on peut effectuer les opérations suivantes :

Insert	insérer un élément
Decrease-key	diminuer la valeur de la clé d'un élément particulier
Delete-min	retourner l'élément ayant la plus petite clé et supprimer-le de la file
Make-queue	créer une file de priorité à partir des éléments donnés, avec les valeurs de clés données.

Complexité des opérations dans les files de priorité

implémentation	deletemin	insert	decreasekey
liste	$O(n)$	$O(1)$	$O(1)$
tas binaire	$O(\log n)$	$O(\log n)$	$O(\log n)$
tas de Fibonacci	$O(\log n)$	$O(1)$	$O(1)$

Intuition pour les tas binaires



- Pour trouver le minimum, il n'est pas nécessaire de trier tous les éléments !
- Les éléments sont stockés dans un arbre binaire complet.
- Tout parent a une clé plus petite que celle de ses enfants.
- La hauteur de l'arbre est $O(\log(n))$.

Illustration de l'opération insert dans un tas binaire

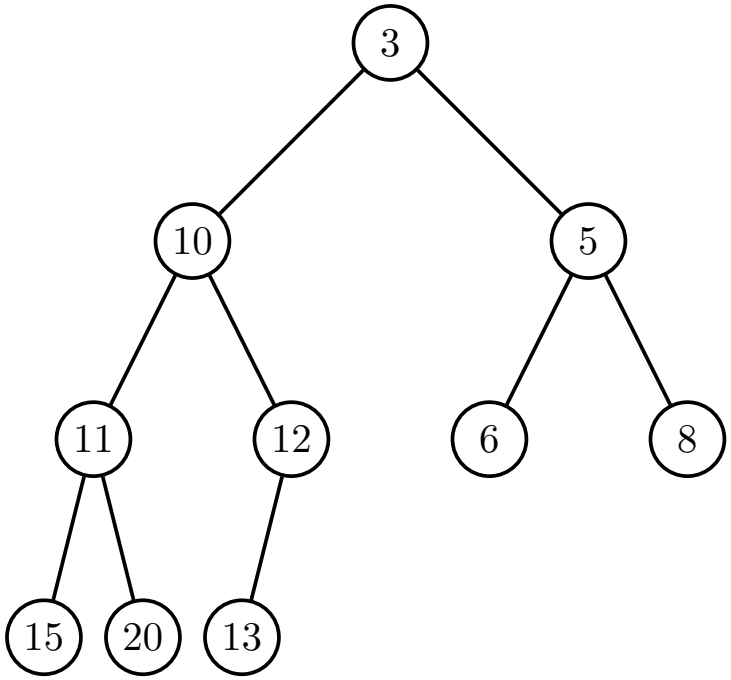


Illustration de l'opération insert dans un tas binaire

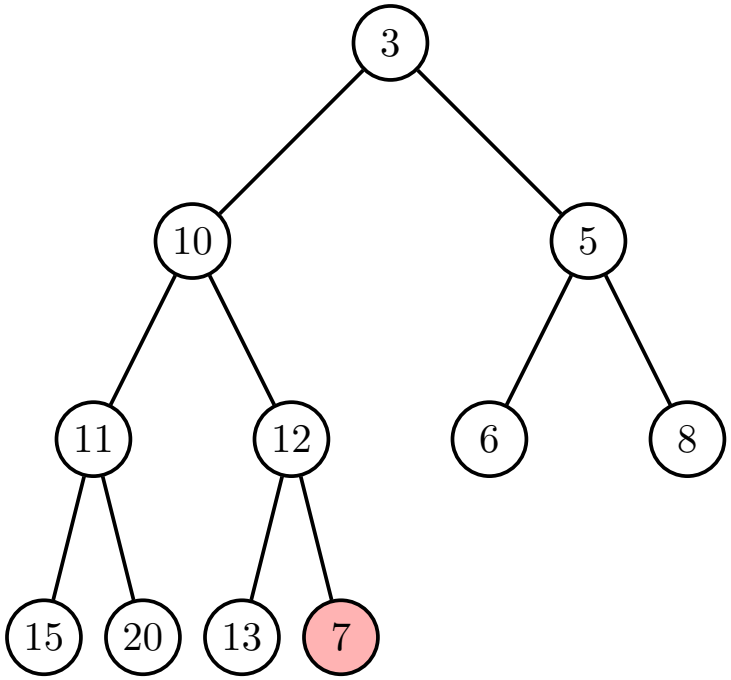


Illustration de l'opération insert dans un tas binaire

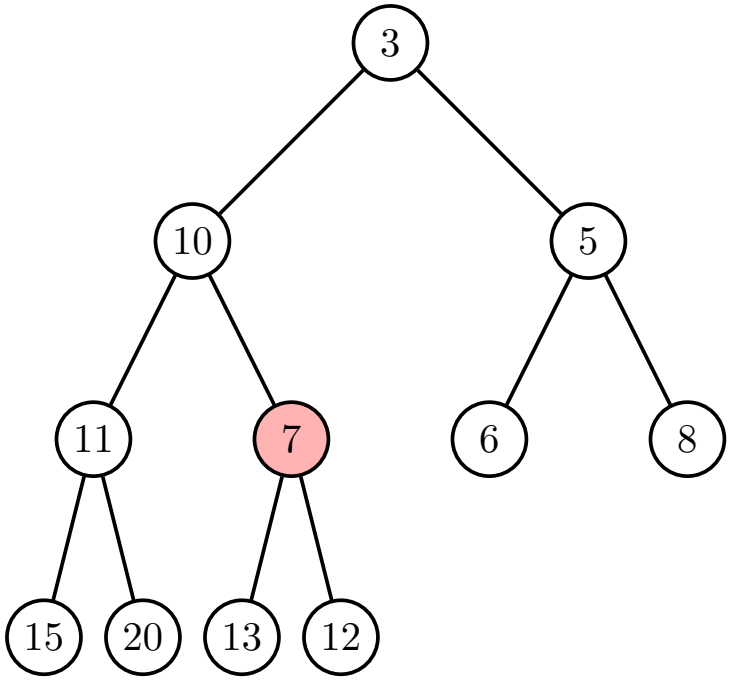


Illustration de l'opération insert dans un tas binaire

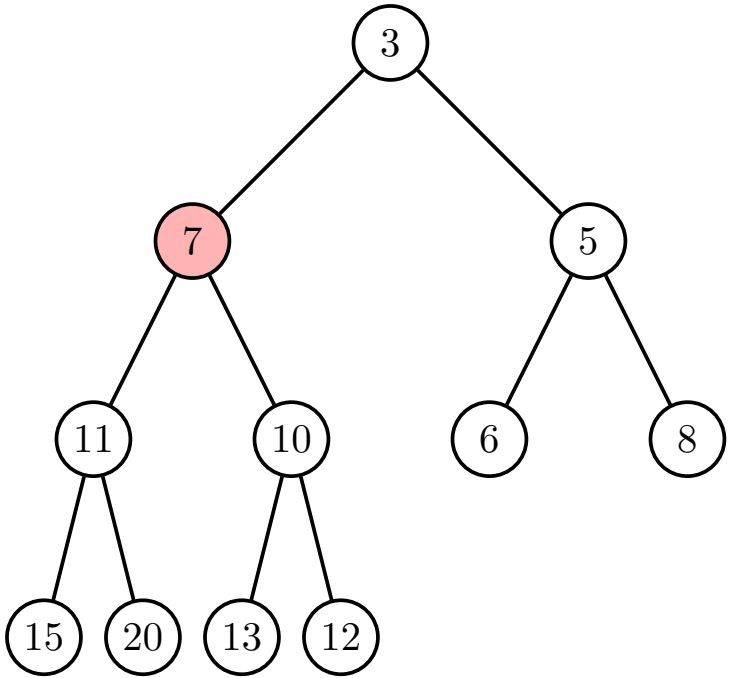


Illustration de l'opération delete-min dans un tas binaire

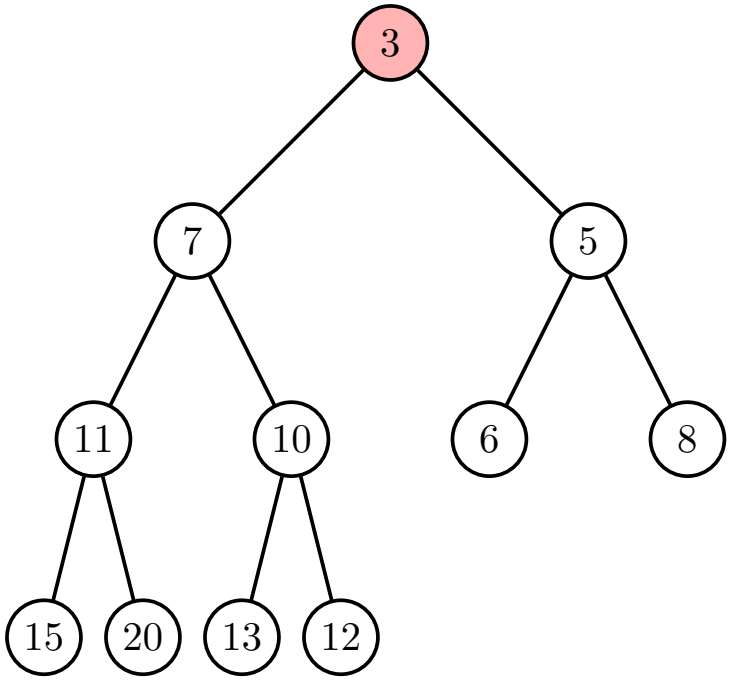


Illustration de l'opération delete-min dans un tas binaire

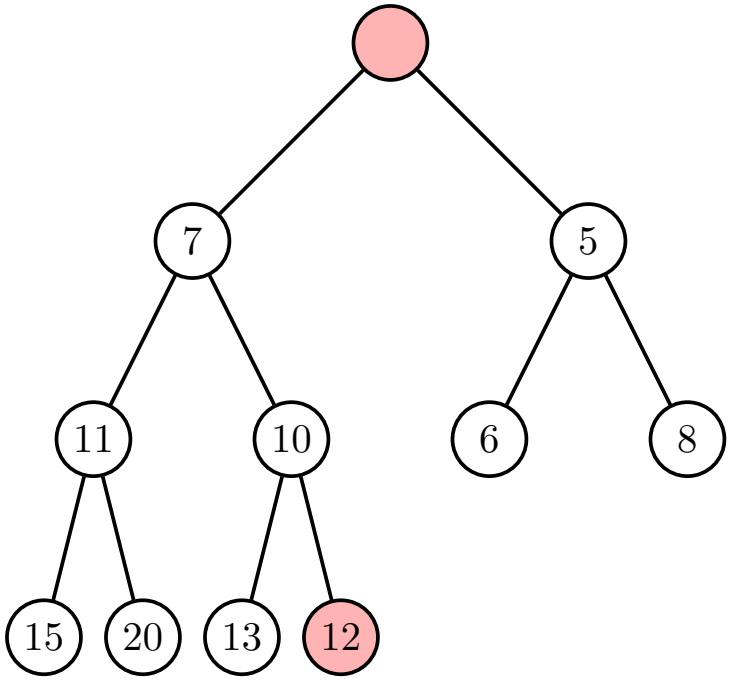


Illustration de l'opération delete-min dans un tas binaire

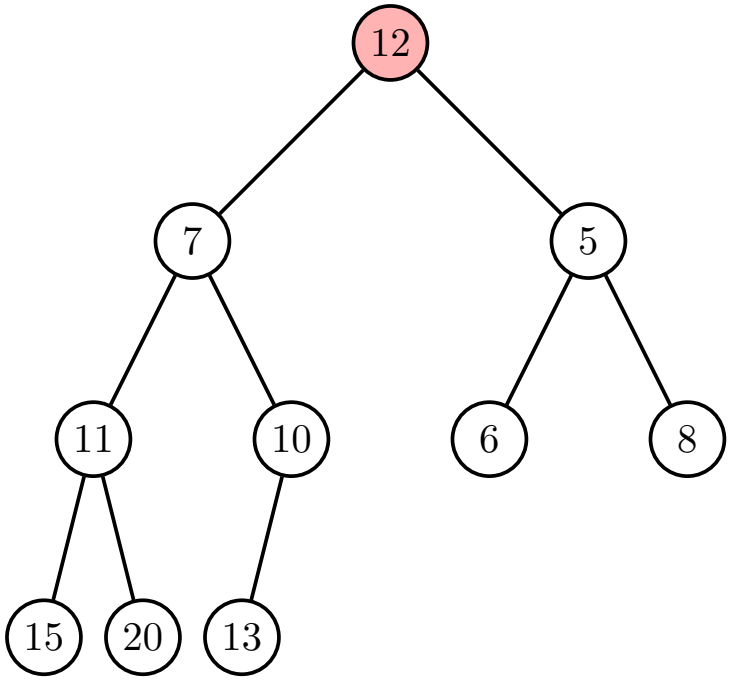


Illustration de l'opération delete-min dans un tas binaire

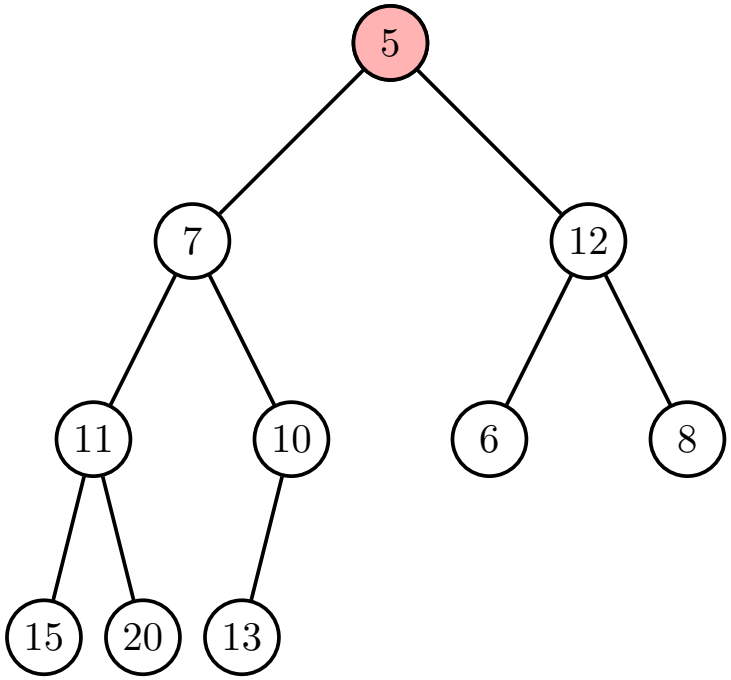
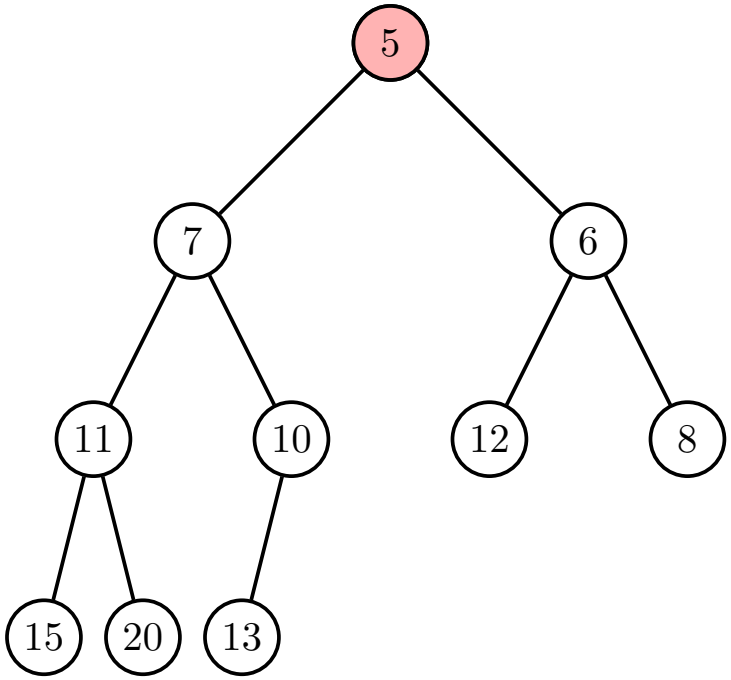


Illustration de l'opération delete-min dans un tas binaire



Implémentation de Dijkstra avec une file de priorité

pour tous les $u \in V$ **faire**
    $D[u] \leftarrow +\infty$
    $\texttt{prev}[u] \leftarrow \emptyset$
$D[s] \leftarrow 0$
$H \leftarrow \text{makequeue}(V)$ n fois (n fois insert)
tant que $H \neq \emptyset$ **faire**
    $u \leftarrow \text{deletemin}(H)$ ✓ n fois
    pour tous les $(u, v) \in E$ **faire**
        si $D[v] > D[u] + w(u, v)$ **alors**
            $D[v] = D[u] + w(u, v)$
            $\texttt{prev}[v] \leftarrow u$
            $\text{decreasekey}(H, v)$ n fois

m fois

Complexité de l'algorithme de Dijkstra

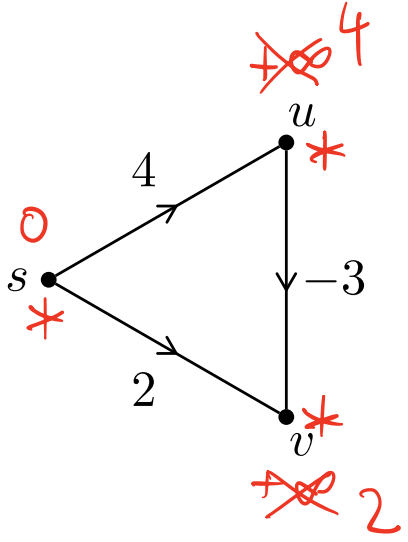
- L'algorithme de Dijkstra est très similaire au parcours en largeur.
- Cependant, il est plus lent car les files de priorité sont plus exigeantes que les files simples de BFS.
- Puisque makequeue prend au plus autant de temps que n opérations d'insertion, nous obtenons un total de n deletemin, n insert et m decreasekey.
- On obtient ainsi les complexités suivantes de Dijkstra selon l'implémentation de la file de priorité :

liste	$O(n^2)$
tas binaire	$O((n+m)\log n)$
tas de Fibonacci	$O(m + n\log n)$

Si G est connexe, $n = O(m)$

Arcs négatifs

- L'algorithme de Dijkstra fonctionne en partie parce que le plus court chemin entre le point de départ s et un sommet v doit passer exclusivement par des sommets plus proches que v.
- Ce n'est plus le cas lorsque l'on permet des arcs de poids négatif.
- Dans cet exemple, le plus court chemin de s à v passe par u, un sommet plus éloigné



Dijkstra vu comme une séquence de mises à jour

- Pour tout sommet v, la valeur de $D[v]$ est toujours supérieure ou égale à $\mathrm{dist}(s, v)$.
- Les valeurs de $D[v]$ changent uniquement grâce à la “mise à jour” le long d’un arc :

  Procédure $\texttt{maj}(u, v)$**:**
  $D[v] \leftarrow \min\{D[v], D[u] + \ell((u, v))\}$

- Cette opération, basée sur le principe de sous-optimalité, satisfait les propriétés suivantes :
  - Elle donne la distance correcte de s à v dans le cas particulier où u est l’avant-dernier noeud du plus court chemin vers v, et $D[u] = \mathrm{dist}(s, u)$.
  - Elle ne rendra jamais $D[v]$ trop petit.

Plus courts chemins dans les graphes avec arcs négatifs

- Soit t un sommet quelconque dans G, et soit $P = (s, u_1, \ldots, u_k, t)$ un plus court chemin de s à t.
- P comprend au plus $n - 1$ arcs (sinon, P n'est pas élémentaire).
- Si la séquence de `maj` comprend $(s, u_1), (u_1, u_2), (u_2, u_3), \ldots, (u_k, t)$, *dans cet ordre* (mais pas nécessairement de manière consécutive), alors, grâce à la première propriété de la diapo précédente, $D[t] = \mathrm{dist}(s, t)$, *même s'il y a des arcs négatifs* !
- Il suffit donc de mettre à jour tous les arcs $n - 1$ fois.

Algorithme de Bellman–Ford

Entrées : Graphe orienté $G = (V, E)$, pondération $\ell \in \mathbb{R}^m$, source $s \in V$

Sorties : D, distances de s aux autres sommets ; `prev`, arbre du plus court chemin

$D[s] \leftarrow 0$

$\texttt{prev}[s] \leftarrow s$

pour tous les $u \in V \setminus \{s\}$ **faire**

  $D[u] \leftarrow +\infty$

  $\texttt{prev}[u] \leftarrow \emptyset$

repéter $|V| - 1$ **fois**

  pour tous les $e \in E$ **faire**

    $\texttt{maj}(e)$

$O(nm)$

retourner $D, \texttt{prev}$

Illustration de l'algorithme de Bellman–Ford

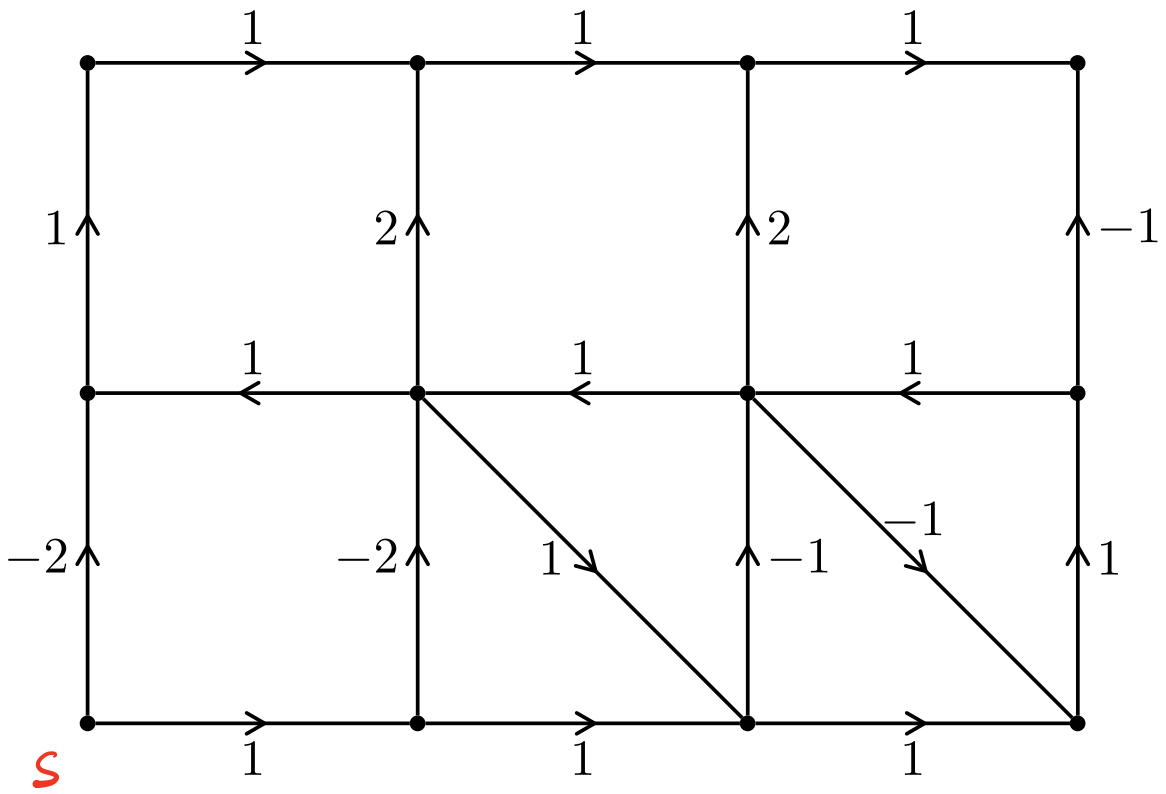


Illustration de l'algorithme de Bellman–Ford

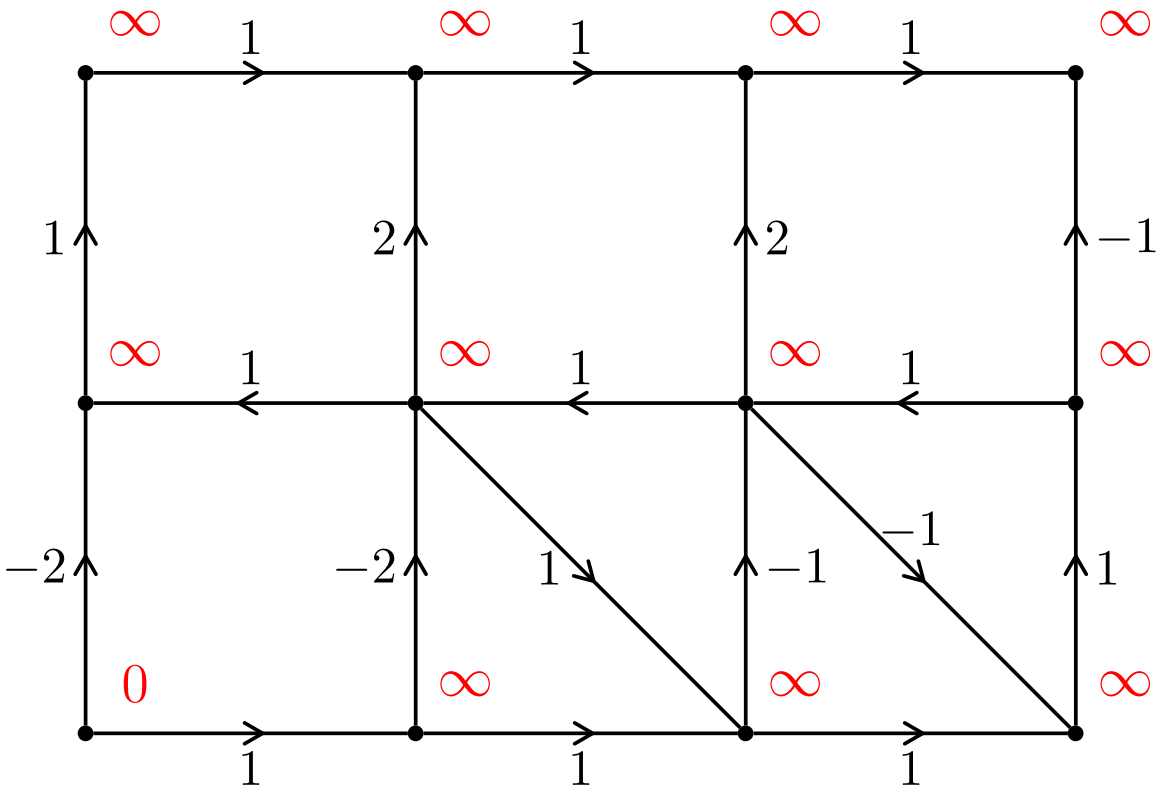


Illustration de l'algorithme de Bellman–Ford

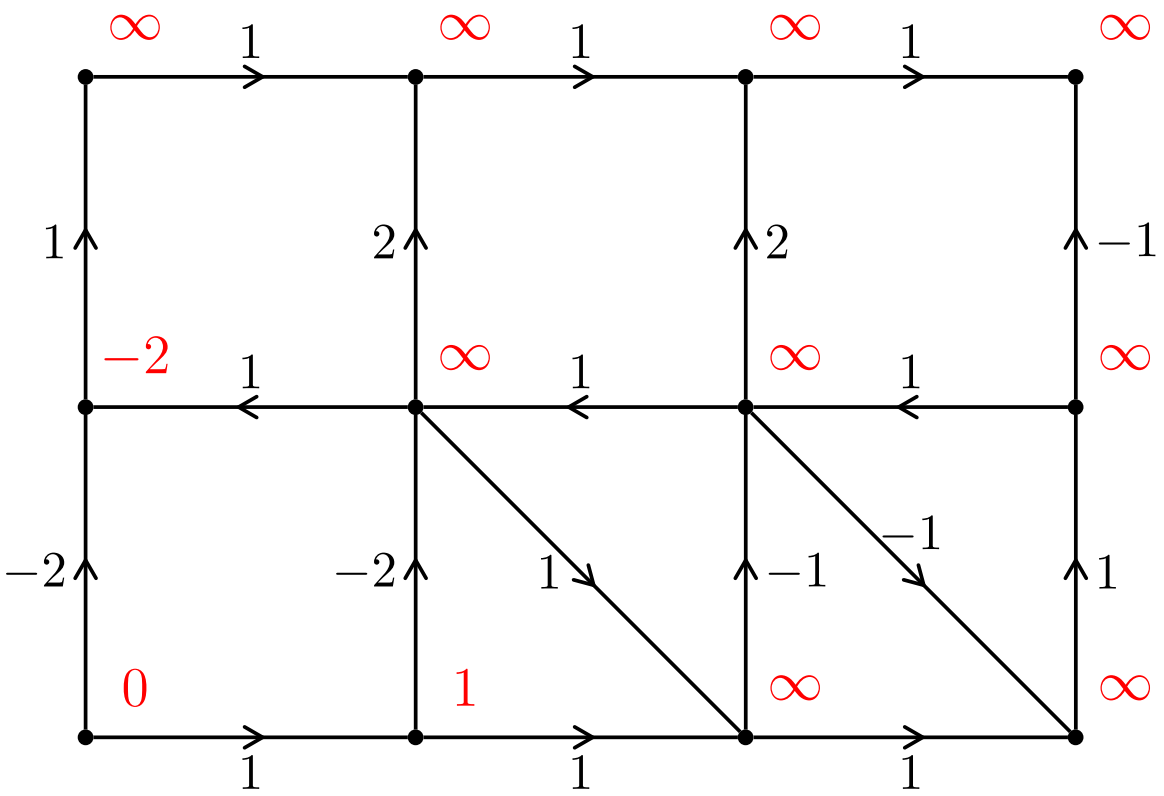


Illustration de l'algorithme de Bellman–Ford

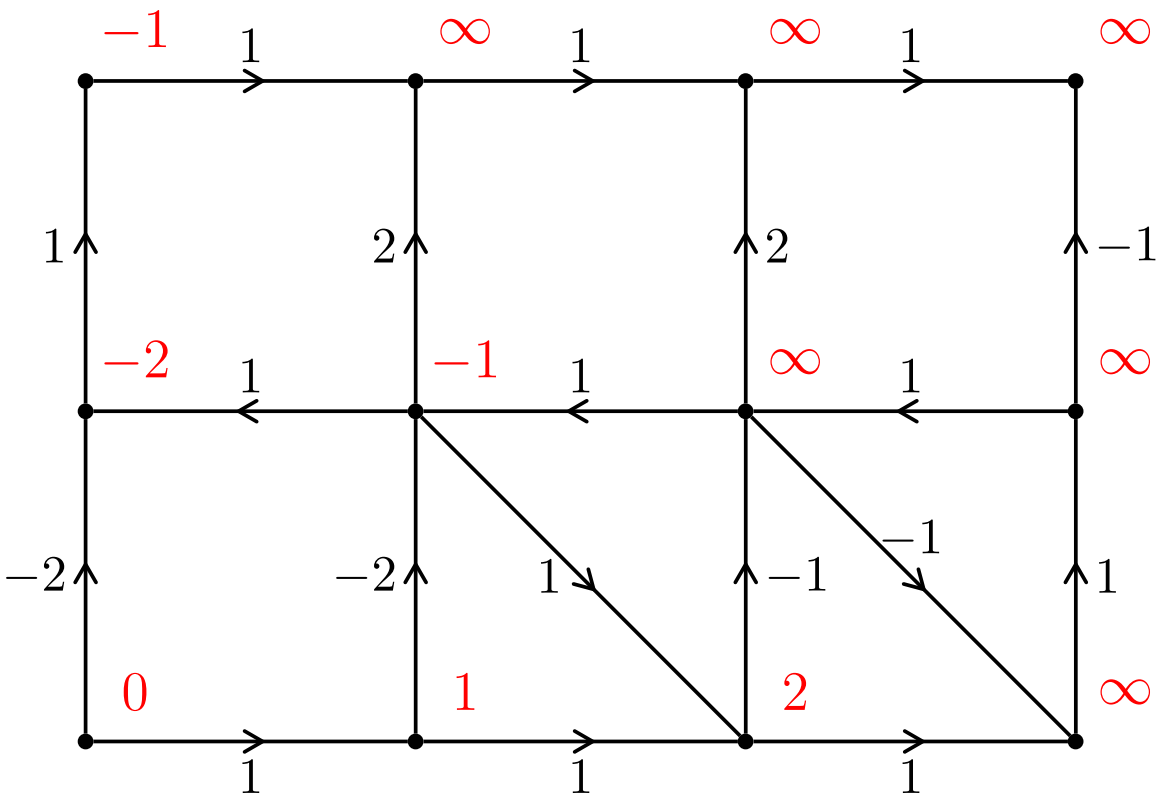


Illustration de l'algorithme de Bellman–Ford

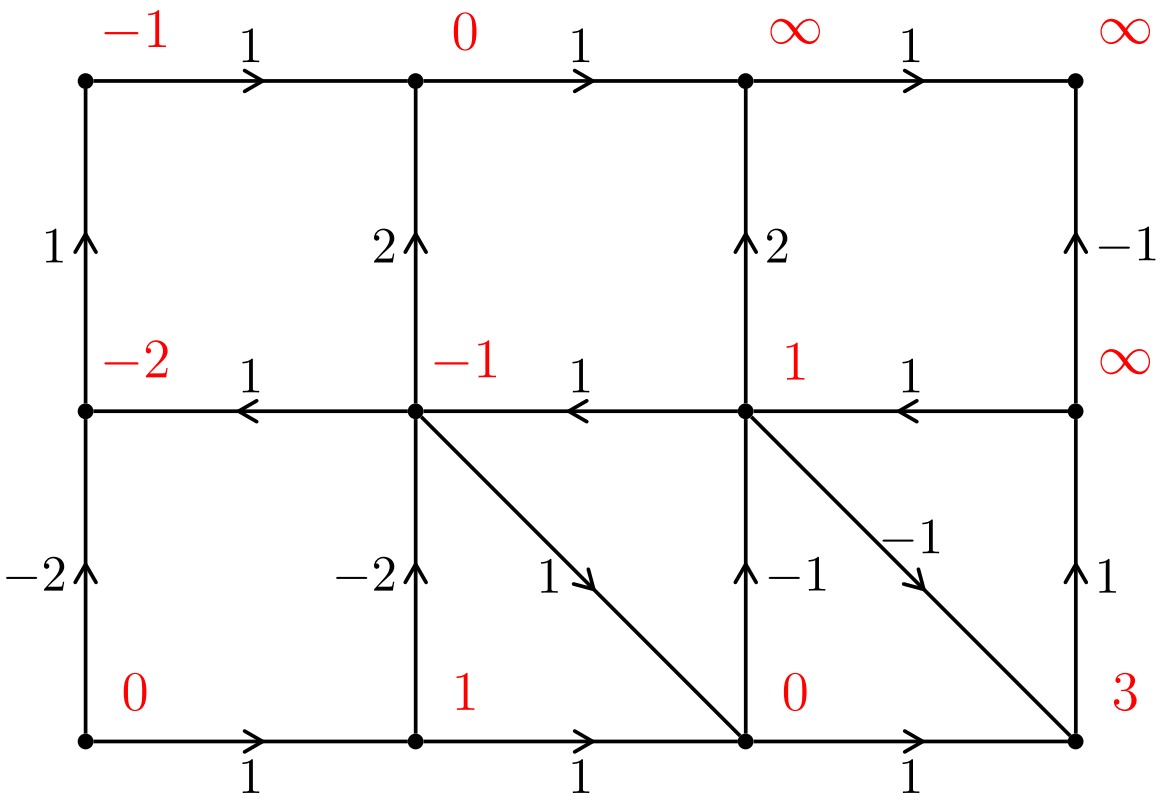


Illustration de l'algorithme de Bellman–Ford

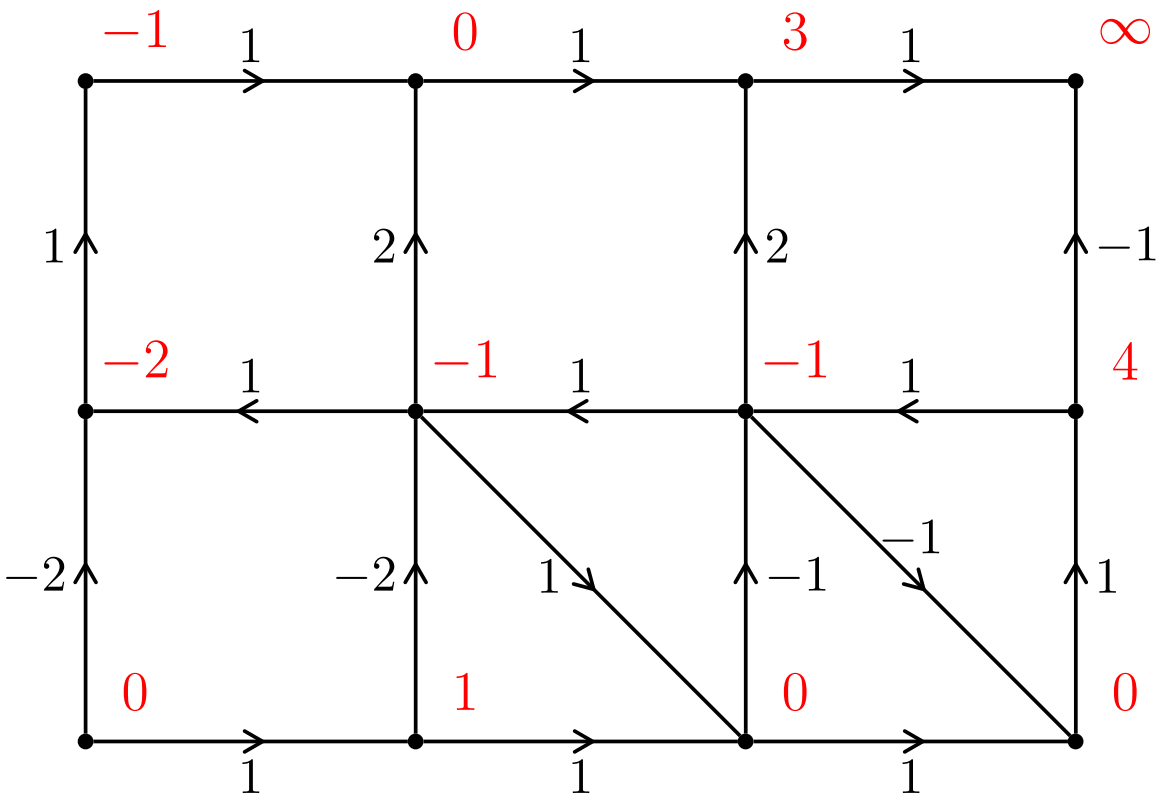


Illustration de l'algorithme de Bellman–Ford

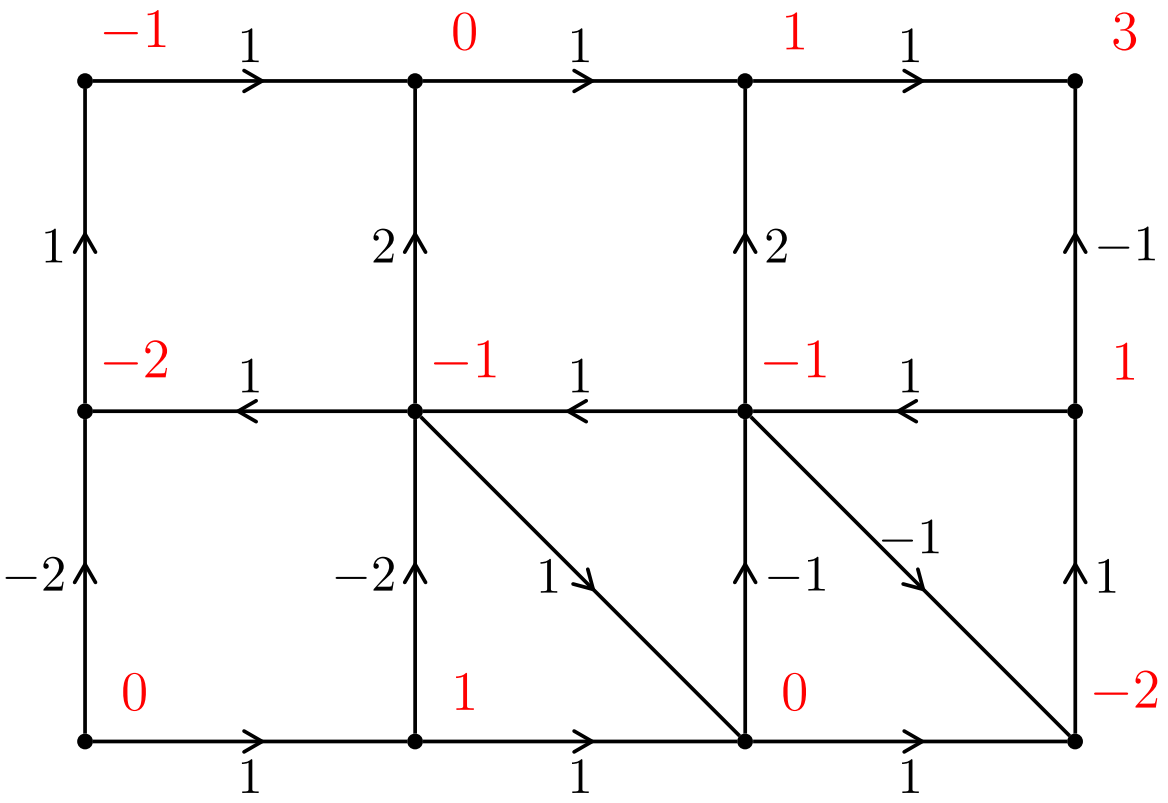


Illustration de l'algorithme de Bellman–Ford

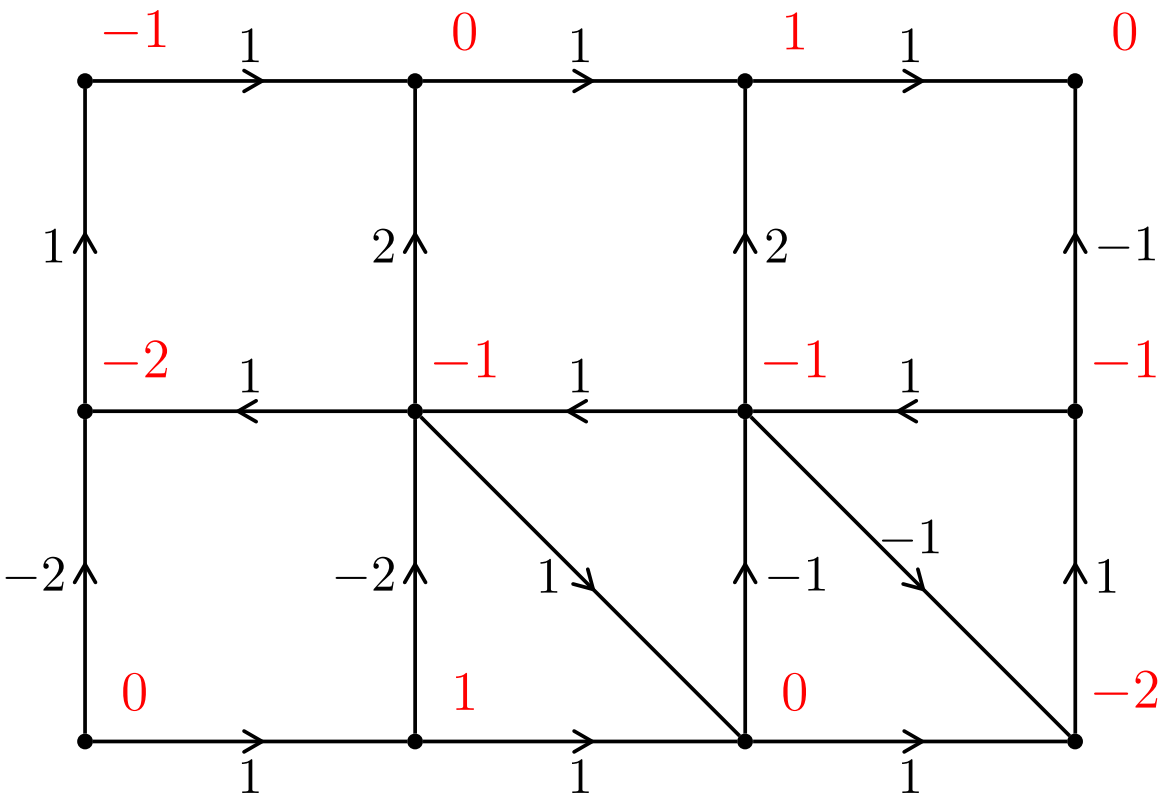
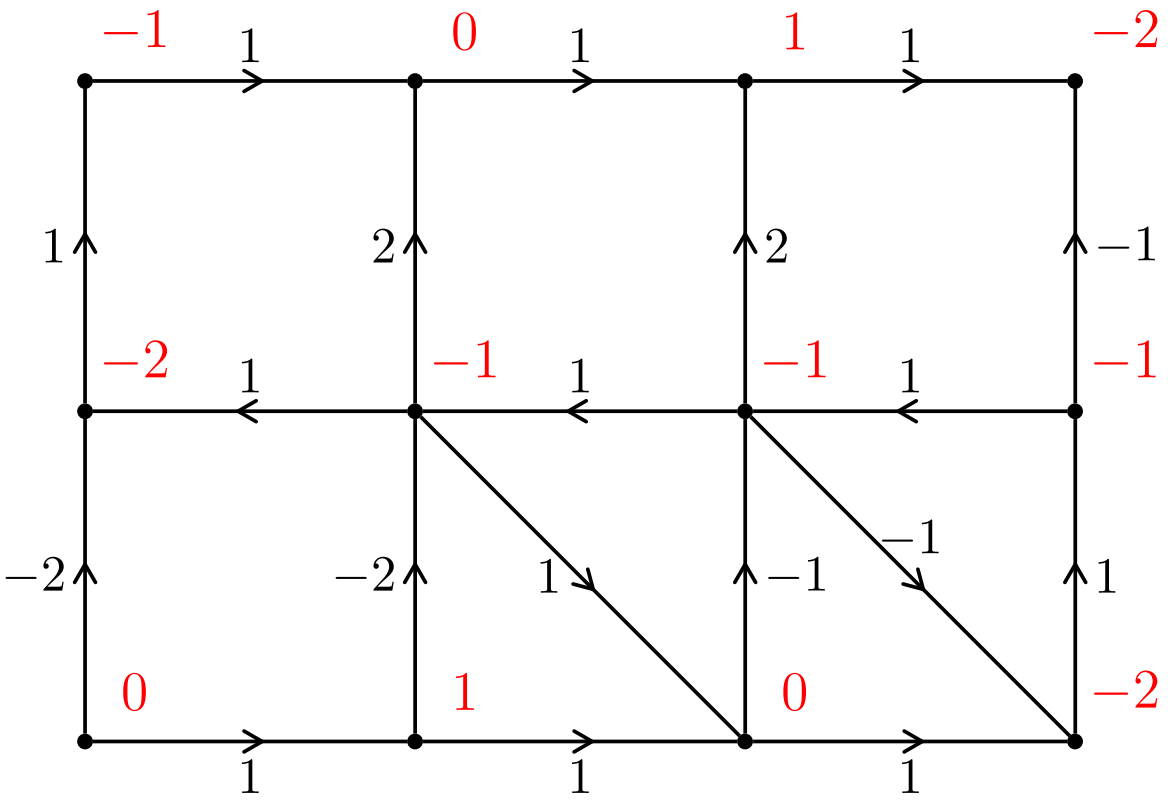


Illustration de l'algorithme de Bellman–Ford



Complexité et correction de l'algorithme de Bellman–Ford

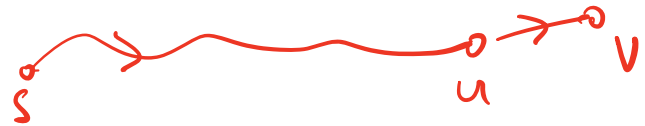
- La complexité de Bellman–Ford est de $O(nm)$.
- Pour la correction de l'algorithme de Bellman–Ford, il suffit de prouver le lemme suivant.

Lemme

Après i itérations de la boucle principale :

- Si $D[u] \neq +\infty$, alors $D[u]$ est la longueur d'un chemin de s à u ;
- S'il existe un chemin de s à u comprenant au plus i arcs, alors la valeur de $D[u]$ est inférieure ou égale à la longueur d'un plus court chemin de s à u comprenant au plus i arcs.

Correction de Bellman–Ford



- Cas de base : $i = 0$.
- On a $D[s] = 0$ et $D[u] = +\infty$ pour tout $u \neq s$.
- $D[s] = 0$ est bien la longueur d'un chemin de s à s (le chemin trivial).
- Supposons que le lemme est vrai pour $i \geq 0$, et considérons l'état après $i + 1$ itérations.
- Soit v un sommet quelconque t.q. $D[v]$ change de $+\infty$ à un nombre fini à la $(i + 1)$-ème itération.
- Donc, $D[v] = D[u] + \ell((u, v))$, où $D[u] < +\infty$ après i itérations.
- Par l'hypothèse de récurrence, $D[u]$ est la longueur d'un chemin P de s à u.
- En concatenant P avec l'arc (u, v), on obtient un chemin de s à v de longueur $D[v]$.
- Le lemme est donc démontré par récurrence.

Détection de cycles négatifs

- Une légère modification de l'algorithme de Bellman–Ford nous permet de détecter des cycles négatifs.
- Après avoir fait les $|V| - 1$ itérations de la boucle, faire une itération supplémentaire.
- Un cycle négatif existe dans G ssi il y a au moins un changement dans le tableau D lors de la dernière itération.
- Détecter des cycles négatifs a des applications importantes dans la vie réele.
- Une application classique est l'arbitrage de devises (voir aussi le TD).

Une application (qui pourrait vous rendre fabuleusement riche)

- Vous avez un ensemble de taux de change entre certaines devises.
- Vous voulez déterminer si un *arbitrage* est possible, c'est-à-dire s'il existe un moyen par lequel vous pouvez commencer avec une unité d'une certaine devise C et effectuer une série d'échanges qui aboutit à avoir plus d'une unité de C.
- Supposons que les coûts de transaction sont nuls et que les taux de change ne fluctuent pas.

Exemple

	USD	**EUR**	**GBP**	**CHF**	**CAD**
USD	1	0.741	0.657	1.061	1.005
EUR	1.349	1	0.888	1.433	1.366
GBP	1.521	1.126	1	1.614	1.538
CHF	0.942	0.698	0.619	1	0.953
CAD	0.995	0.732	0.650	1.049	1

Exemple

	USD	**EUR**	**GBP**	**CHF**	**CAD**
USD	1	0.741	0.657	1.061	1.005
EUR	1.349	1	0.888	1.433	1.366
GBP	1.521	1.126	1	1.614	1.538
CHF	0.942	0.698	0.619	1	0.953
CAD	0.995	0.732	0.650	1.049	1

- Échangeons 10000 USD dans la séquence suivante : USD $\rightarrow$ EUR $\rightarrow$ CAD $\rightarrow$ USD

Exemple

	USD	**EUR**	**GBP**	**CHF**	**CAD**
USD	1	0.741	0.657	1.061	1.005
EUR	1.349	1	0.888	1.433	1.366
GBP	1.521	1.126	1	1.614	1.538
CHF	0.942	0.698	0.619	1	0.953
CAD	0.995	0.732	0.650	1.049	1

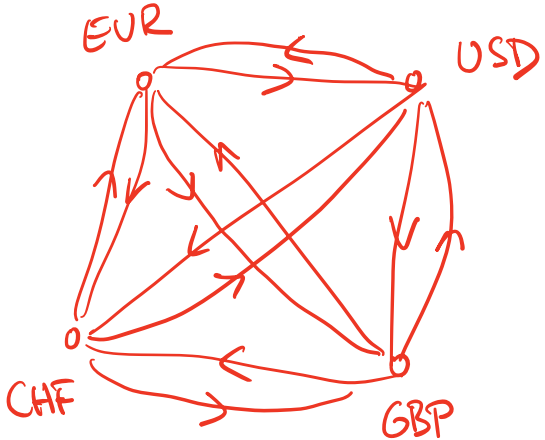
- Échangeons 10000 USD dans la séquence suivante : USD $\rightarrow$ EUR $\rightarrow$ CAD $\rightarrow$ USD
- Nous obtenons $10000 \times 0.741 \times 1.366 \times 0.995 \approx 10071$

Exemple

	USD	**EUR**	**GBP**	**CHF**	**CAD**
USD	1	0.741	0.657	1.061	1.005
EUR	1.349	1	0.888	1.433	1.366
GBP	1.521	1.126	1	1.614	1.538
CHF	0.942	0.698	0.619	1	0.953
CAD	0.995	0.732	0.650	1.049	1

- Échangeons 10000 USD dans la séquence suivante : USD $\rightarrow$ EUR $\rightarrow$ CAD $\rightarrow$ USD
- Nous obtenons $10000 \times 0.741 \times 1.366 \times 0.995 \approx 10071$
- Nous avons ainsi fait un bénéfice de 71 USD !
- Bellman-Ford peut servir pour trouver une telle séquence de changes.

Réduction au problème de circuit négatif



Il faut trouver un moyen de transformer l'opération de multiplication à l'opération d'addition.

$$\log(ab) = \log a + \log b$$

$-\log(0{,}741)$

Soit c_{ab} le taux de change de la devise a à la devise b. Alors, on met le poids $-\log c_{ab}$ sur l'arc (a,b). Un cycle négatif correspond à une séquence de changes où vous gagnez de l'argent.

Deux classes naturelles de graphes orientés sans circuits négatifs

- Rappelons que le concept du plus court chemin n'a pas de sens s'il existe un circuit négatif.
- Nous nous intéressons donc aux graphes orientés *sans circuits négatifs*.
- Il y en a deux classes naturelles :
  - les graphes sans arcs négatifs
  - les graphes sans circuits orientés (les DAG).
- Dans les graphes sans arcs négatifs, on peut utiliser l'algorithme de Dijkstra, de complexité $O(m + n \log n)$.
- Peut-on faire mieux que Bellman–Ford (de complexité $O(nm)$) pour les DAG ?

Plus court chemin dans les graphes orientés acycliques (DAG)

- Rappelons qu'il faut effectuer une séquence de mises à jour qui inclut chaque plus court chemin comme sous-séquence.
- Dans tout chemin d'un DAG, les sommets apparaissent dans un ordre topologique croissant.
- Par conséquent, il suffit de faire un tri topologique du DAG par une recherche en profondeur, et puis de parcourir les sommets dans l'ordre topologique, en mettant chaque fois à jour tous les arcs sortants du sommet.
- La complexité de cet algorithme est de $O(n + m)$.

Algorithme de plus court chemin dans les DAG

Entrées : Graphe orienté $G = (V, E)$, pondération $w \in \mathbb{R}^m$, source $s \in V$

Sorties : Distances de s aux autres sommets

$D[s] \leftarrow 0$

$\texttt{prev}[s] \leftarrow s$

pour tous les $u \in V \setminus \{s\}$ **faire**

- $D[u] \leftarrow +\infty$
- $\texttt{prev}[u] \leftarrow \emptyset$

Tri topologique de G

pour tous les $u \in V$ *dans l'ordre topologique* **faire**

- **pour tous les** $(u, v) \in E$ **faire**
  - $\texttt{maj}(u, v)$

retourner $D, \texttt{prev}$

Illustration de l'algorithme de plus court chemin dans les DAG

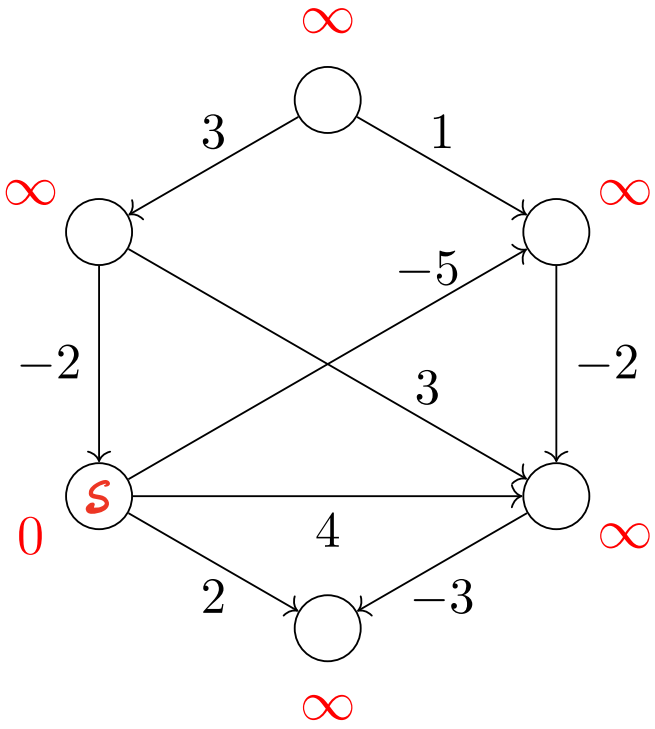


Illustration de l'algorithme de plus court chemin dans les DAG

$S = v_3$

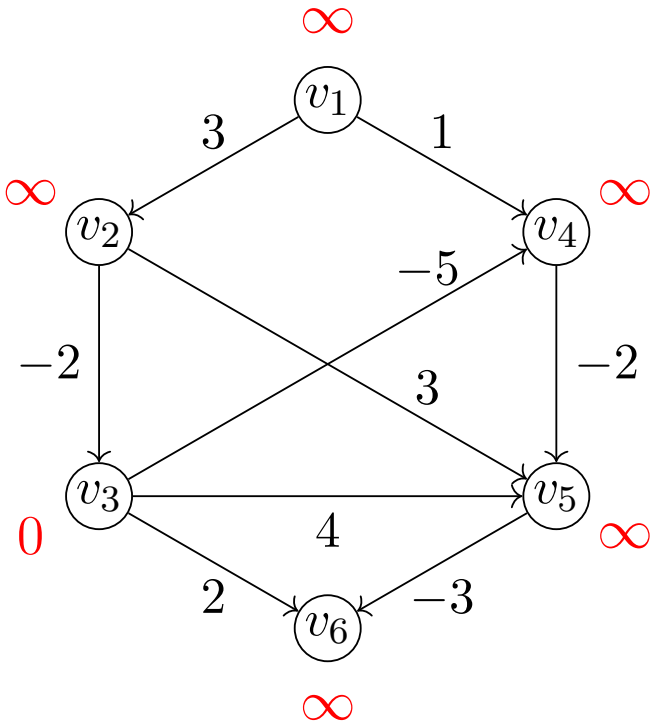


Illustration de l'algorithme de plus court chemin dans les DAG

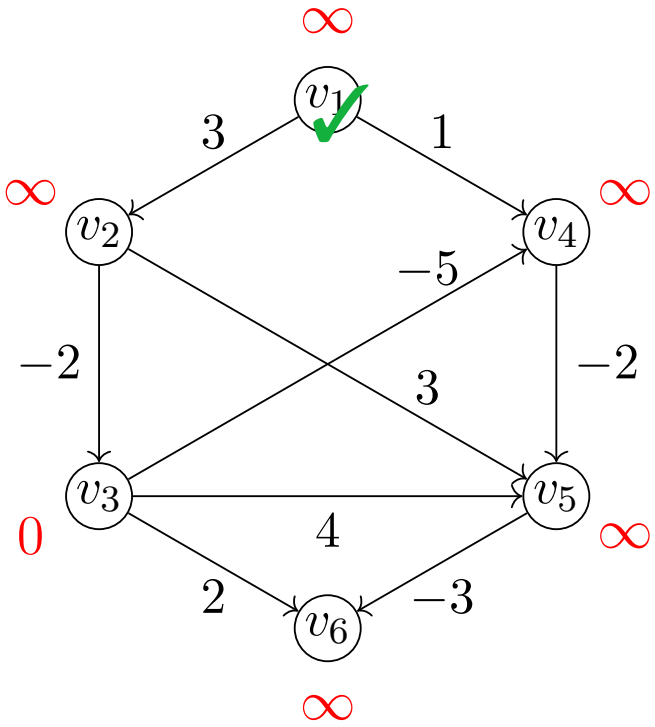


Illustration de l'algorithme de plus court chemin dans les DAG

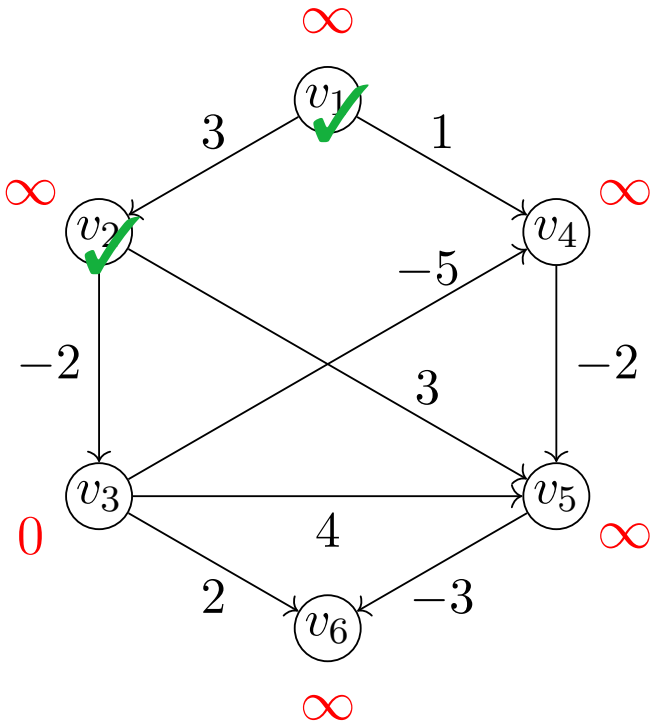


Illustration de l'algorithme de plus court chemin dans les DAG

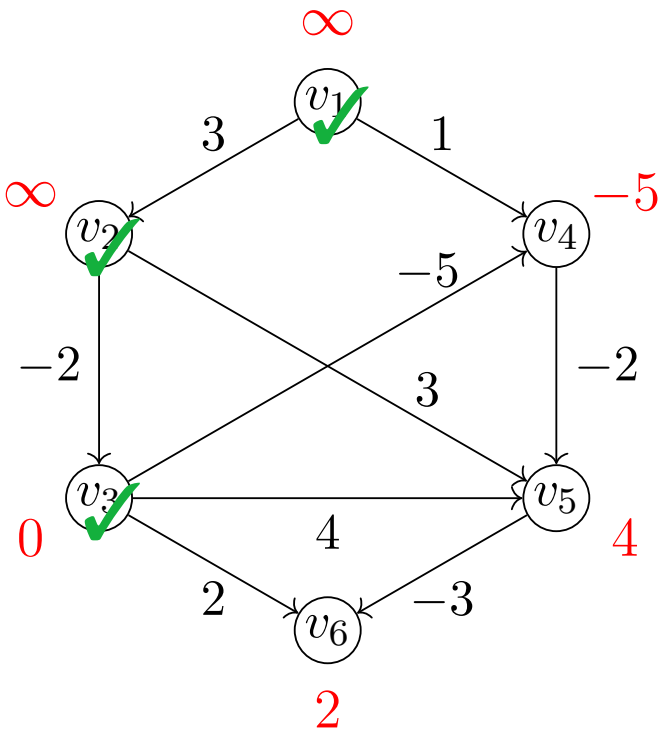


Illustration de l'algorithme de plus court chemin dans les DAG

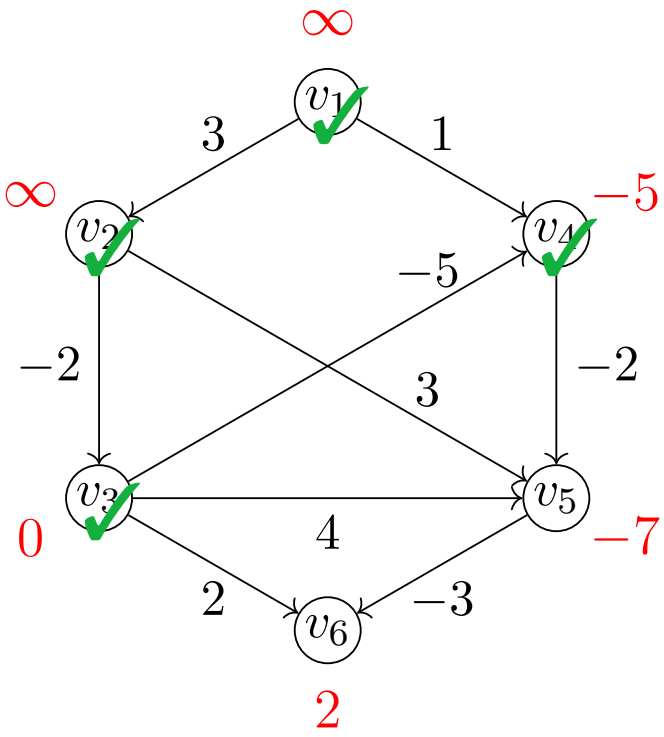


Illustration de l'algorithme de plus court chemin dans les DAG

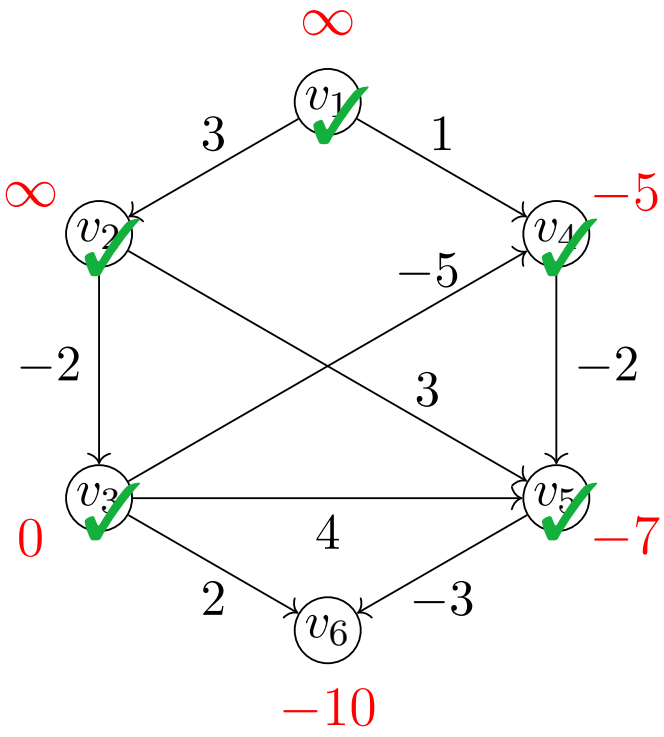
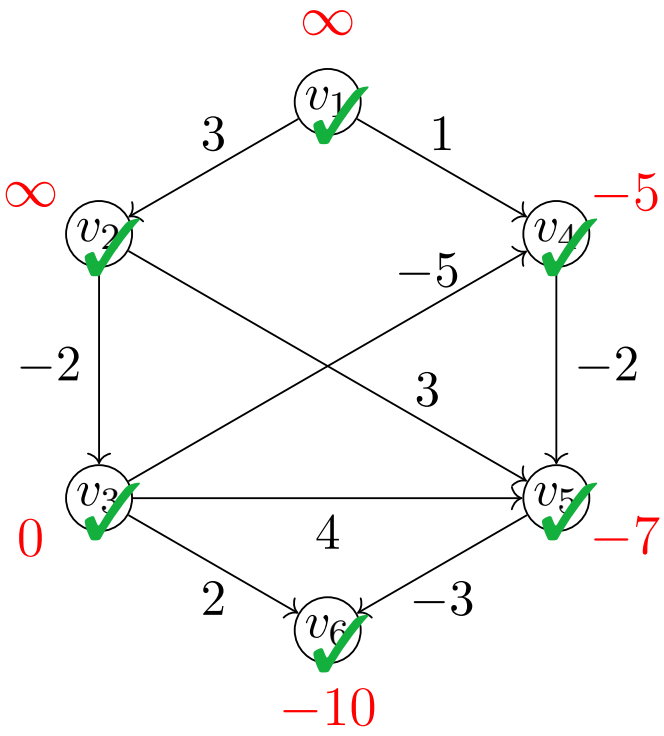


Illustration de l'algorithme de plus court chemin dans les DAG



Distances entre toutes les paires de sommets

- Dijkstra et Bellman–Ford trouvent la distance d'un sommet fixe (la source) aux autres sommets.
- Et si on veut trouver la distance entre *toutes les paires* de sommets ?
- Une approche naïve : exécuter Dijkstra ou Bellman–Ford n fois : une fois pour chaque sommet.
- La complexité de l'algorithme ainsi obtenu est de :
  - $O(nm + n^2 \log n)$ (cas avec poids non négatifs)
  - $O(n^2 m)$ (cas général)
- Si l'on ignore le terme logarithmique, le premier algorithme (poids non négatifs) est de la même complexité que Bellman–Ford.
- Pour les graphes denses, la complexité du deuxième algorithme est de $O(n^4)$.
- Peut-on faire mieux ?

Sommets intermédiaires

- Le plus court chemin $(u, w_1, \ldots, w_\ell, v)$ de u à v utilise un certain nombre de sommets “intermédiaires”.
- Supposons que nous n’autorisions aucun sommet intermédiaire.
- Nous pouvons alors trouver les plus courts chemins entre toutes les paires en un seul coup : le plus court chemin de u à v est simplement l’arc (u, v), si il existe.
- On élargit progressivement (d’un sommet à chaque étape) l’ensemble des sommets intermédiaires autorisés, en mettant à jour les longueurs des plus courts chemins à chaque étape.

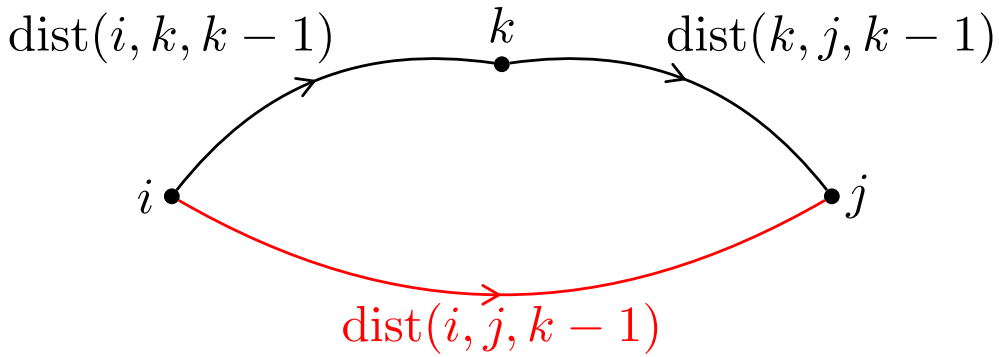
Distances partielles

- Soit $V = \{1, 2, \ldots, n\}$ l'ensemble des sommets.
- Soit $\mathrm{dist}(i, j, k)$ la longueur minimum d'un chemin de i à j dont tous les sommets intermédiaires sont dans $\{1, 2, \ldots, k\}$.
- En particulier,

$$\mathrm{dist}(i, j, 0) = \begin{cases} \ell(i, j) & \textbf{si } (i, j) \in E \\ \infty & \textbf{si } (i, j) \notin E. \end{cases}$$

- Un plus court chemin de i à j qui emprunte k (et éventuellement d'autres sommets intermédiaires qui précédent k) passe par k une seule fois.

Mise à jour des distances partielles



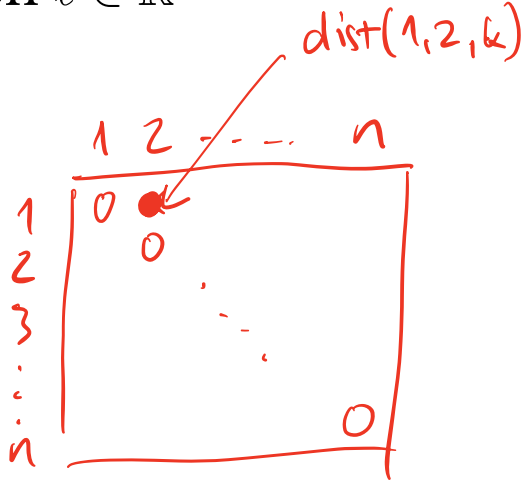
- On a déjà calculé la longueur d'un plus court chemin passant uniquement par les sommets intermédiaires dans $\{1,\ldots,k-1\}$.
- Passer par k donne un chemin plus court de i à j ssi

$$\mathrm{dist}(i,k,k-1)+\mathrm{dist}(k,j,k-1)<\mathrm{dist}(i,j,k-1).$$

Algorithme de Floyd–Warshall

Entrées : Graphe orienté $G = (V, E)$ avec pondération $\ell \in \mathbb{R}^{|E|}$

Sorties : Distances entre chaque paire de sommets



pour tous les $i \in \{1, \ldots, n\}$ **faire**

  pour tous les $j \in \{1, \ldots, n\}$ **faire**

    $\mathrm{dist}(i, j, 0) \leftarrow \infty$

pour tous les $(i, j) \in E$ **faire**

  $\mathrm{dist}(i, j, 0) \leftarrow \ell(i, j)$

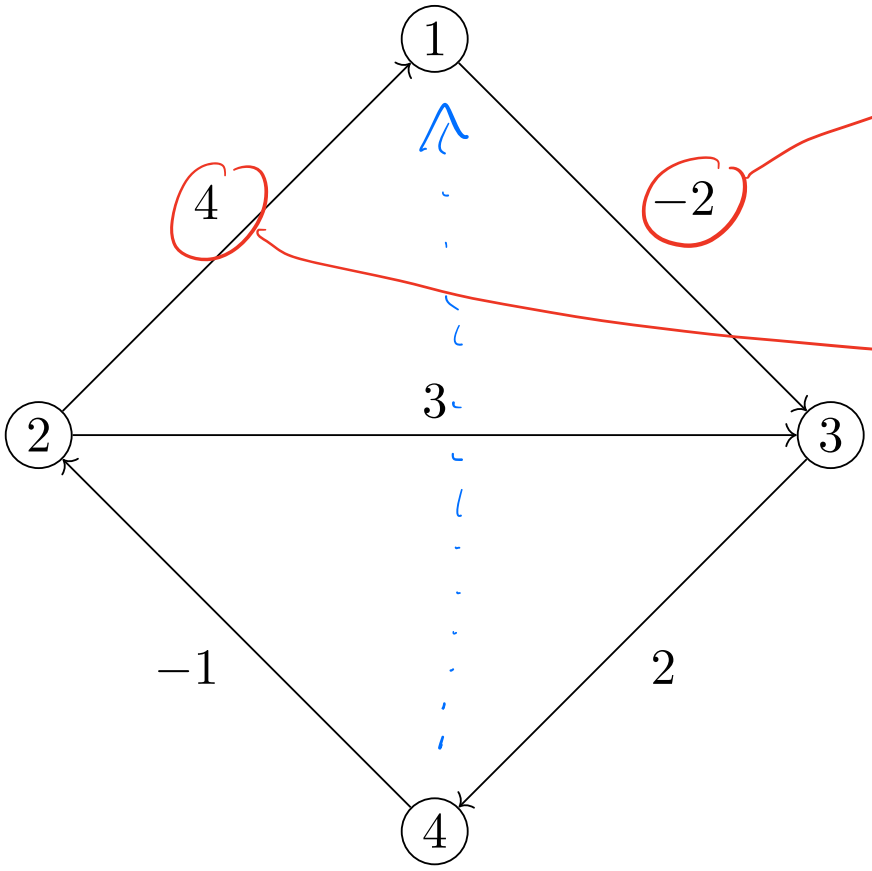
pour tous les $k \in \{1, \ldots, n\}$ **faire**

  pour tous les $i \in \{1, \ldots, n\}$ **faire**

    pour tous les $j \in \{1, \ldots, n\}$ **faire**

      $\mathrm{dist}(i, j, k) = \min\{\mathrm{dist}(i, k, k-1) + \mathrm{dist}(k, j, k-1), \mathrm{dist}(i, j, k-1)\}$

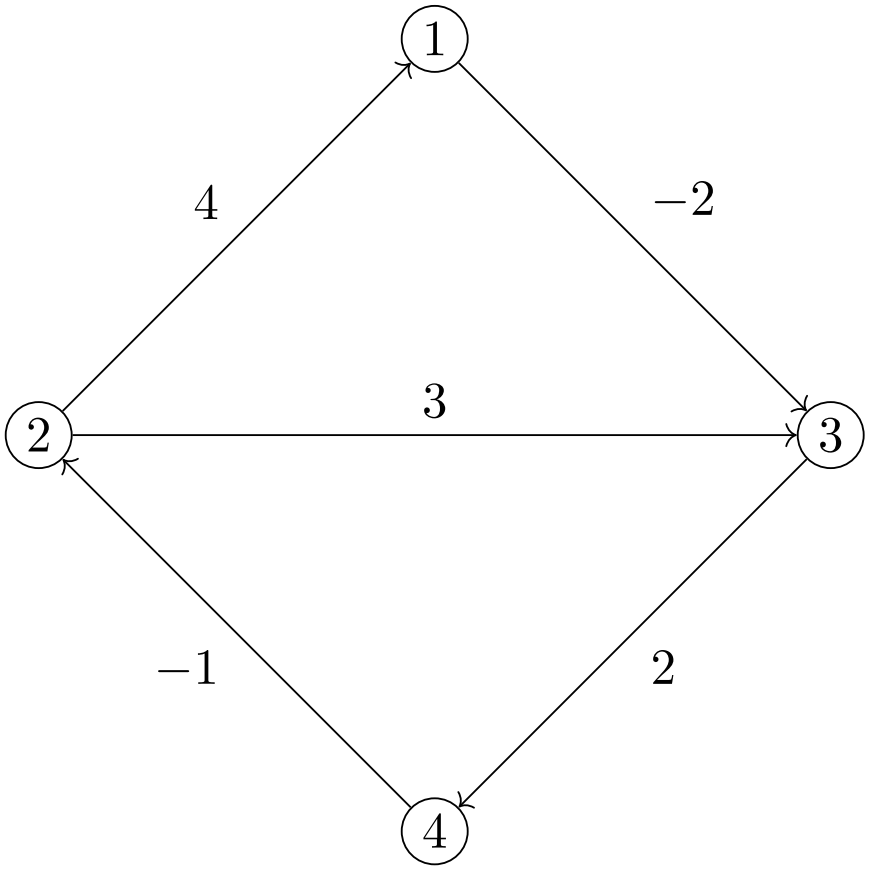
Illustration de l'algorithme de Floyd–Warshall



	1	2	3	4
1	0	∞	-2	∞
2	4	0	3	∞
3	∞	∞	0	2
4	∞	-1	∞	0

$\mathrm{dist}(\cdot, \cdot, 0)$

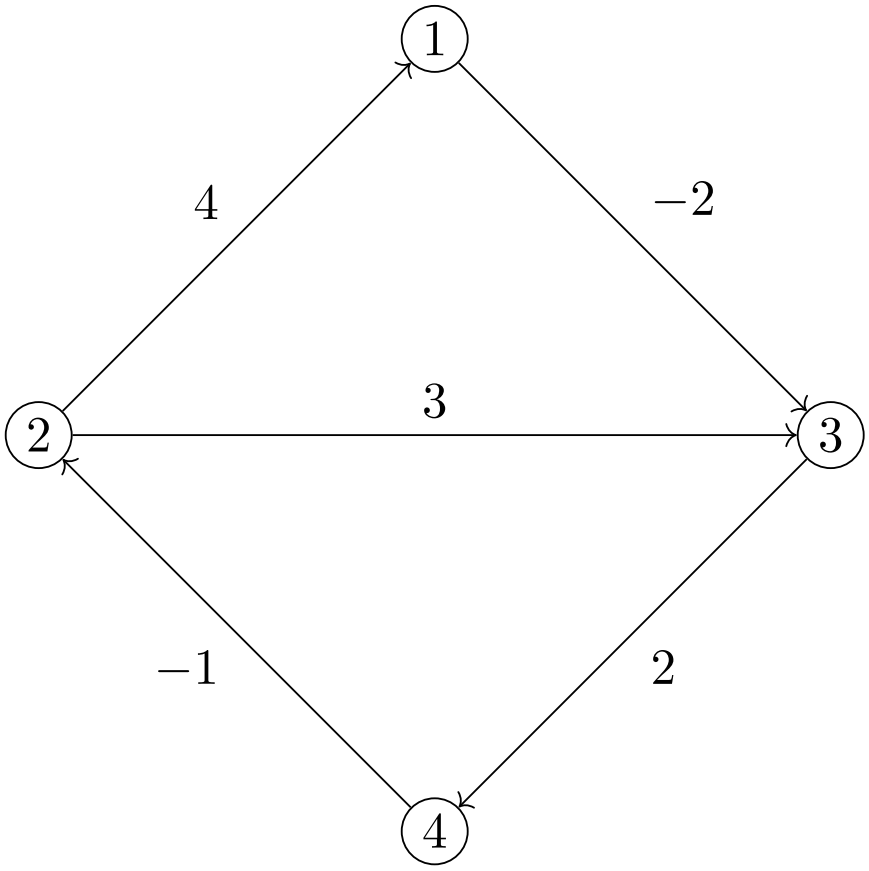
Illustration de l'algorithme de Floyd–Warshall



	1	2	3	4
1	0	∞	-2	∞
2	4	0	2	∞
3	∞	∞	0	2
4	∞	-1	∞	0

$\mathrm{dist}(\cdot, \cdot, 1)$

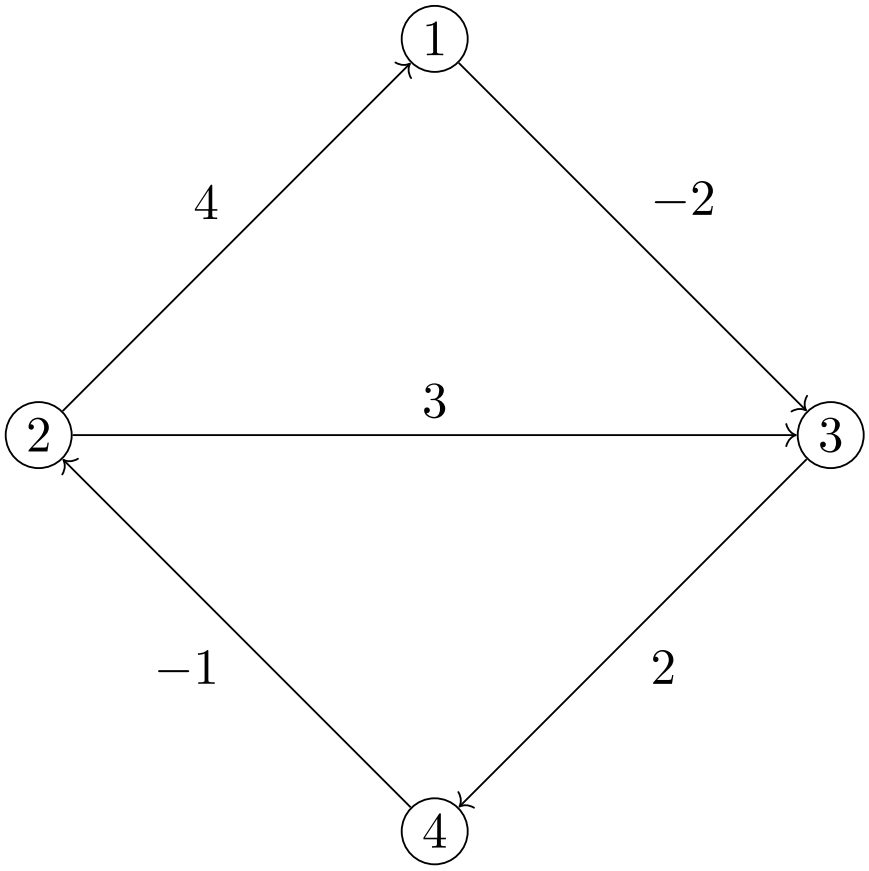
Illustration de l'algorithme de Floyd–Warshall



	1	2	3	4
1	0	∞	-2	∞
2	4	0	2	∞
3	∞	∞	0	2
4	3	-1	1	0

$\mathrm{dist}(\cdot, \cdot, 2)$

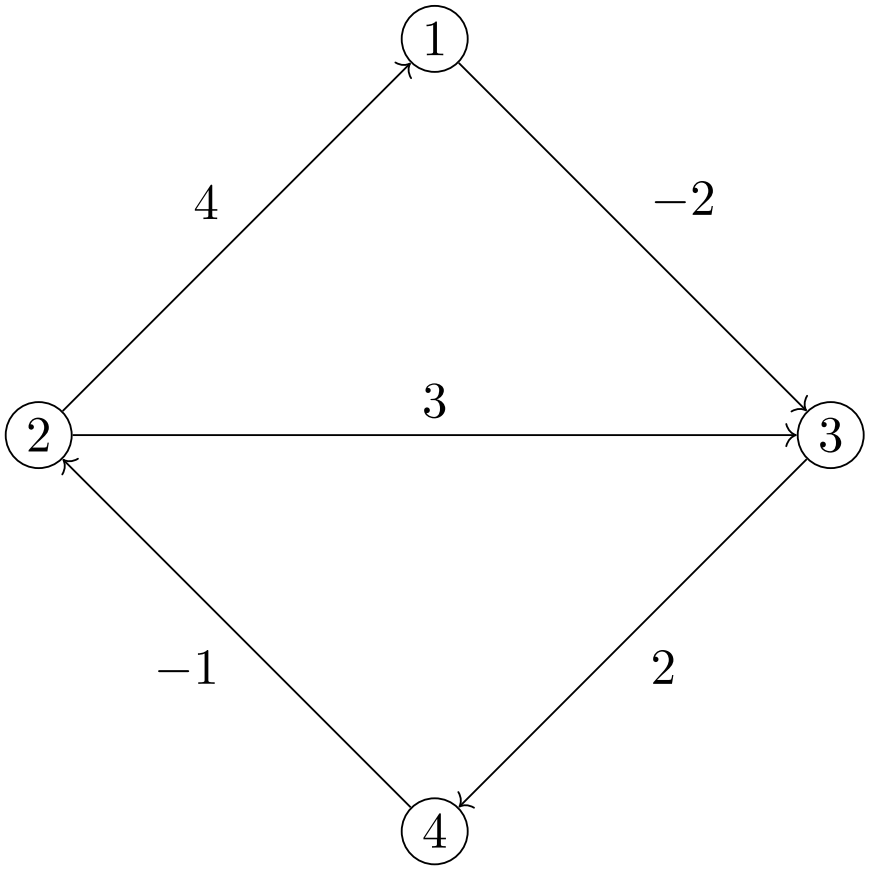
Illustration de l'algorithme de Floyd–Warshall



	1	2	3	4
1	0	∞	-2	0
2	4	0	2	4
3	∞	∞	0	2
4	3	-1	1	0

$\mathrm{dist}(\cdot, \cdot, 3)$

Illustration de l'algorithme de Floyd–Warshall



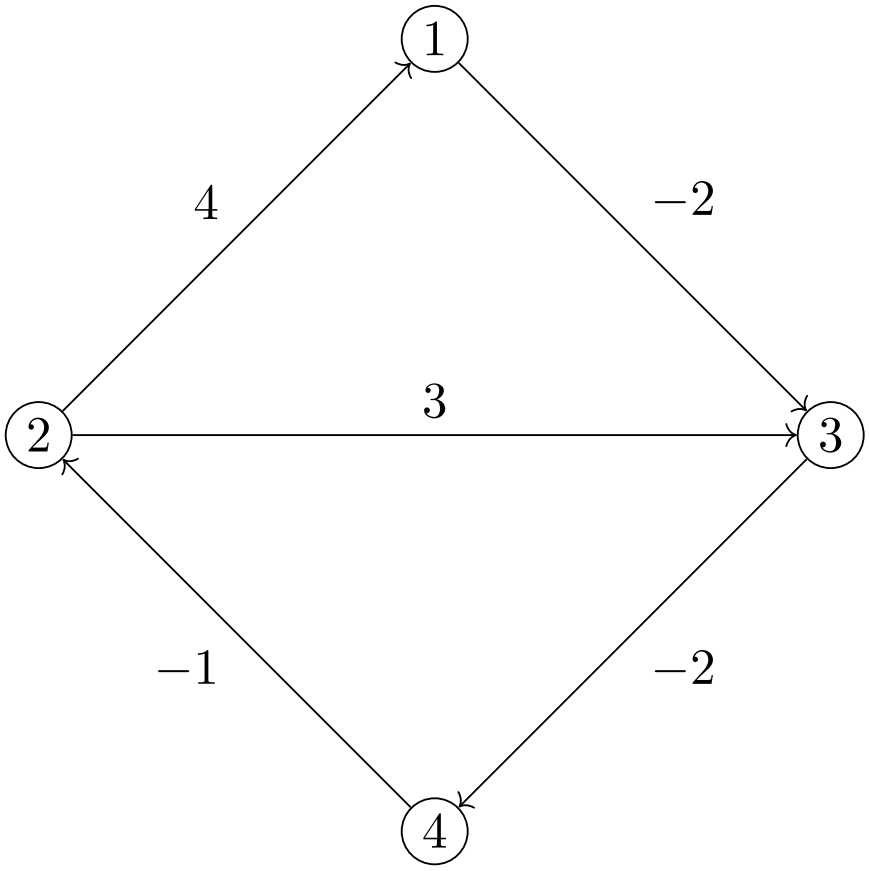
	1	2	3	4
1	0	−1	−2	0
2	4	0	2	4
3	5	1	0	2
4	3	−1	1	0

$\mathrm{dist}(\cdot, \cdot, 4)$

Remarques sur l'algorithme de Floyd–Warshall

- La complexité est de $O(n^3)$.
- Pour les graphes *denses*, cela représente une amélioration d'un facteur de n par rapport à l'approche naïve.
- On verra un autre algorithme (de Johnson) mieux adapté aux graphes *peu denses*.
- L'algorithme de Floyd–Warshall peut être utilisé pour détecter les circuits négatifs.
- Il y a un nombre négatif sur la diagonale de la matrice de distances ssi le graphe contient au moins un circuit négatif.

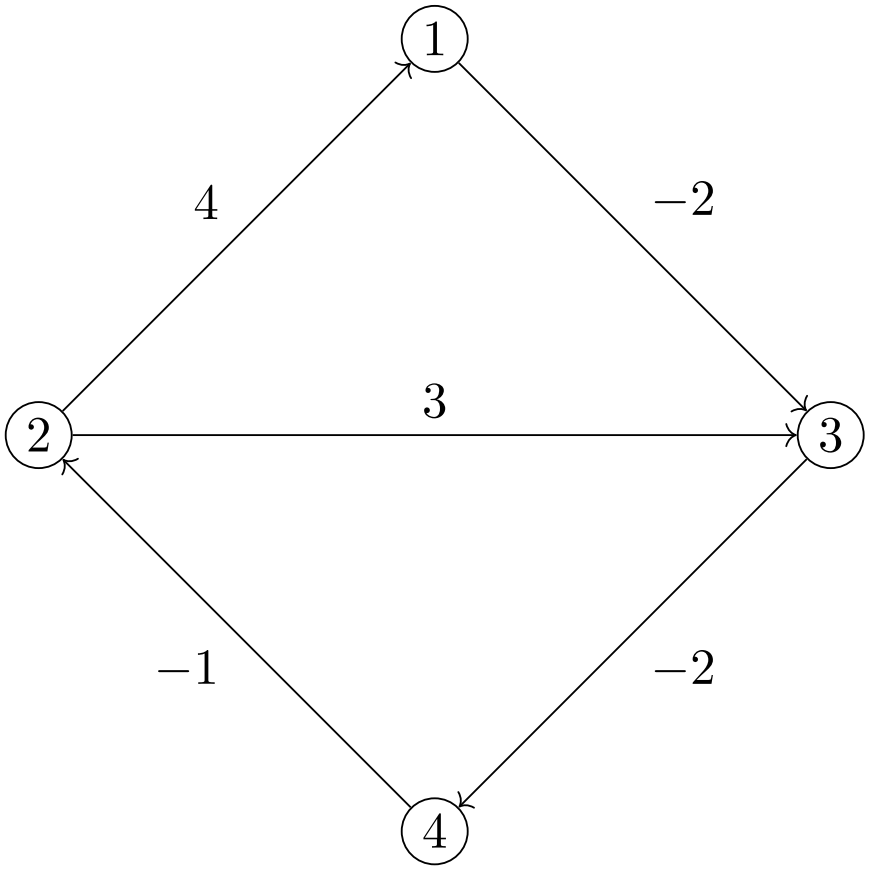
Détection de circuit négatif avec Floyd–Warshall



	1	2	3	4
1	0	∞	-2	∞
2	4	0	3	∞
3	∞	∞	0	-2
4	∞	-1	∞	0

$\mathrm{dist}(\cdot, \cdot, 0)$

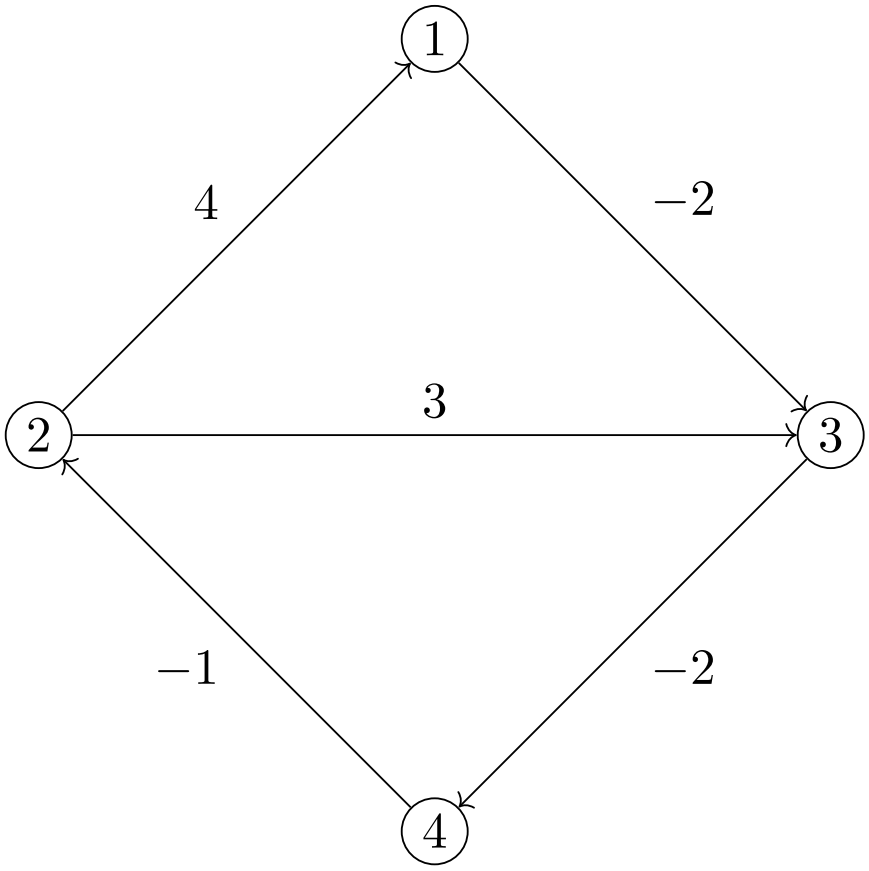
Détection de circuit négatif avec Floyd–Warshall



	1	2	3	4
1	0	∞	-2	∞
2	4	0	2	∞
3	∞	∞	0	-2
4	∞	-1	∞	0

$\mathrm{dist}(\cdot, \cdot, 1)$

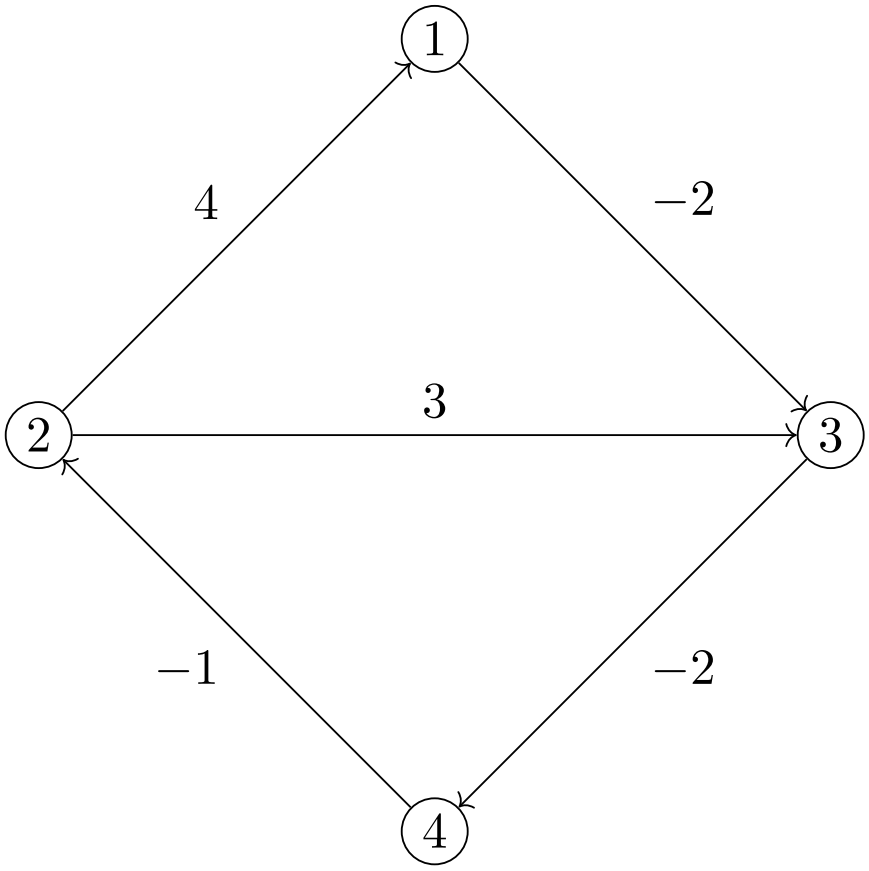
Détection de circuit négatif avec Floyd–Warshall



	1	2	3	4
1	0	∞	-2	∞
2	4	0	2	∞
3	∞	∞	0	-2
4	3	-1	1	0

$\mathrm{dist}(\cdot, \cdot, 2)$

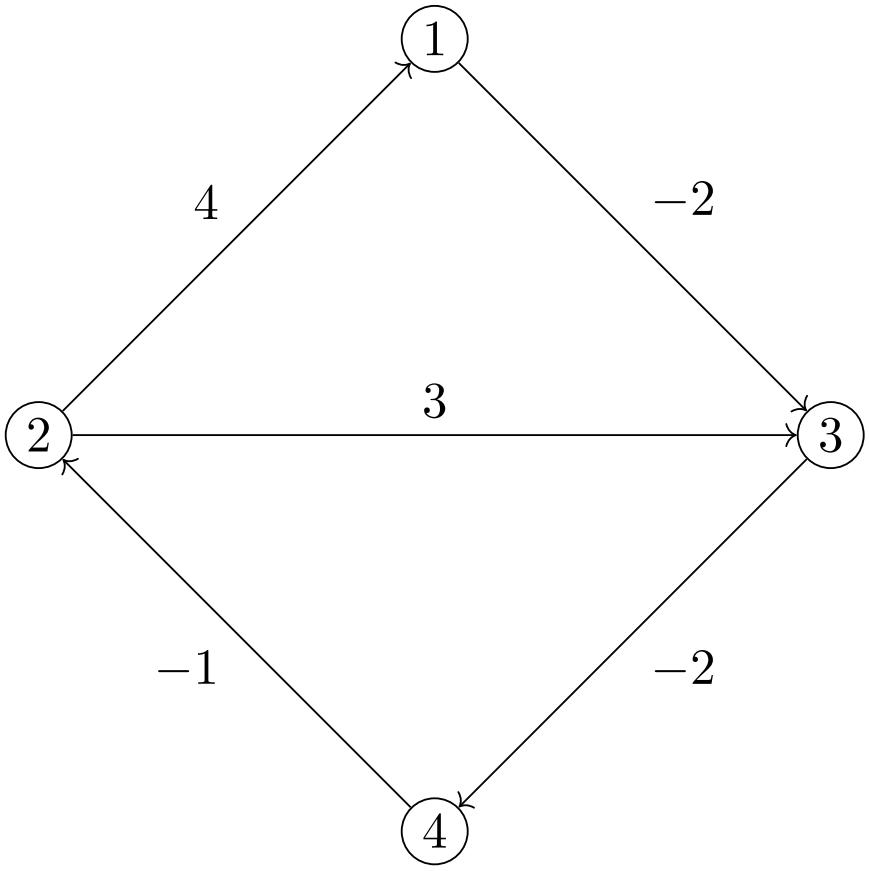
Détection de circuit négatif avec Floyd–Warshall



	1	2	3	4
1	0	∞	-2	-4
2	4	0	2	1
3	∞	∞	0	-2
4	3	-1	1	0

$\mathrm{dist}(\cdot, \cdot, 3)$

Détection de circuit négatif avec Floyd–Warshall



	1	2	3	4
1	−1	−1	−2	0
2	4	−1	2	4
3	1	−3	−1	2
4	3	−1	1	−1

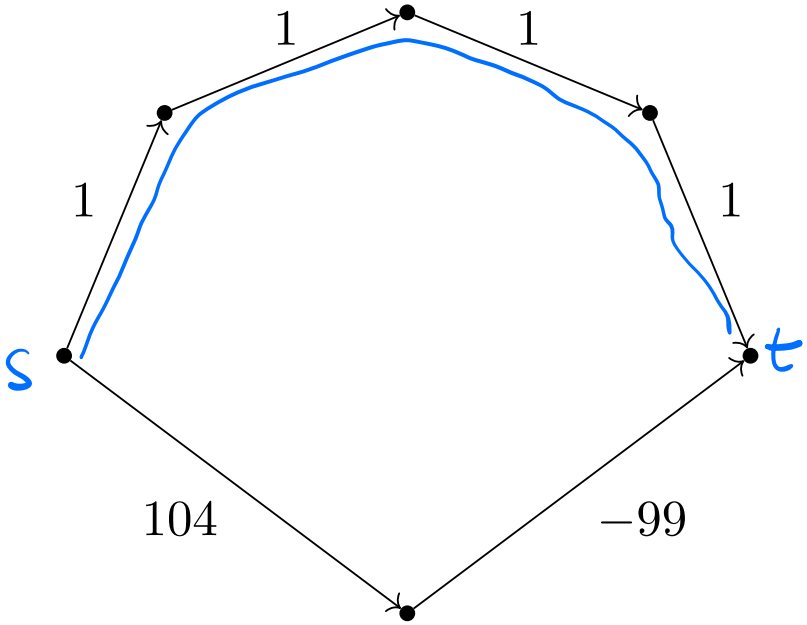
$\mathrm{dist}(\cdot, \cdot, 4)$

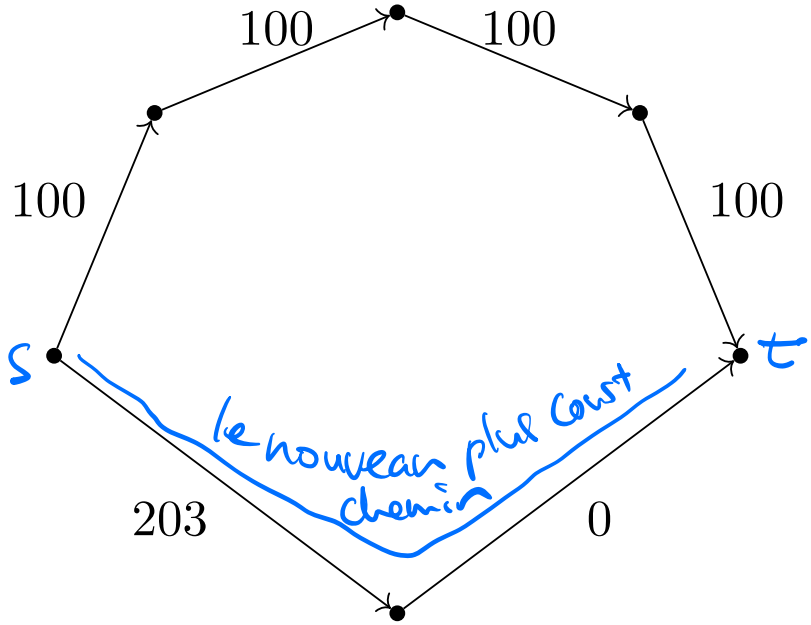
Peut-on faire mieux que $O(n^3)$ dans le cas des graphes peu denses ?

$m = o(n^2)$

- Idée naïve : repondérer le graphe de sorte que les poids deviennent non-négatifs, et les plus courts chemins soient préservés.
- Ensuite, exécuter Dijkstra n fois (une fois par sommet) ; complexité $O(nm + n^2 \log n)$.
- Comment trouver une telle repondération ?
- Première tentative : ajouter une constante au poids de chaque arc de sorte d'éliminer les poids négatifs

Cette repondération naïve ne préserve pas les plus court chemins !





ajouter 99 à tous les arcs

Une repondération préservant les plus courts chemins

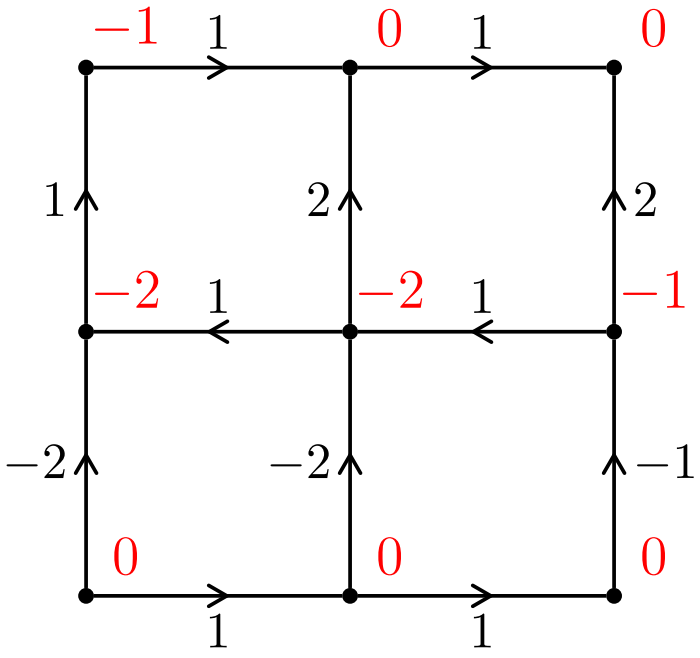
- Soit $G = (V, E)$ un graphe avec pondération $\ell \in \mathbb{R}^m$.
- Soit $h \in \mathbb{R}^n$ un vecteur associant à chaque sommet un nombre réel.
- On définit une nouvelle pondération $\ell' \in \mathbb{R}^m$ de G par $\ell'_{(u,v)} = \ell_{(u,v)} + h_u - h_v$.

Lemme

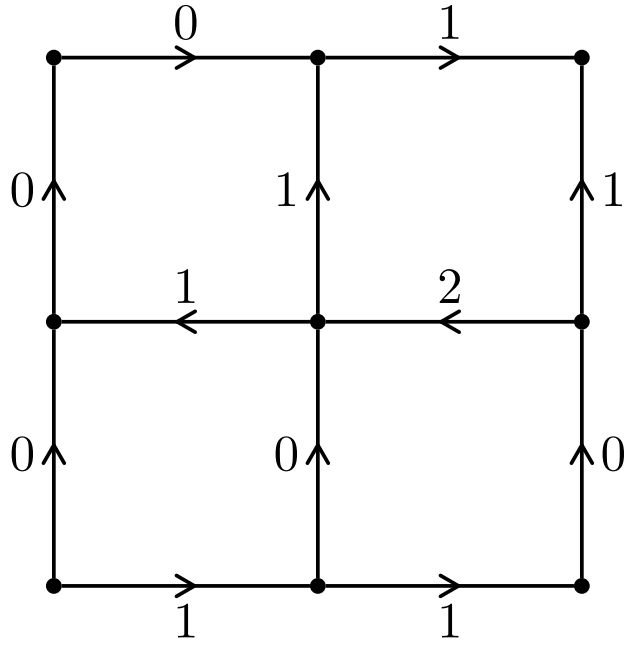
P est un plus court chemin de u à v dans G par rapport à ℓ ssi P est un plus court chemin de u à v dans G par rapport à ℓ'.

Cette repondération préserve les plus courts chemins !

Exemple



ℓ, h



ℓ'

Preuve du lemme (1/2)

- Soit P un chemin quelconque dans G.

$$\begin{aligned}
\ell'(P) &= \sum_{i=1}^{k} \ell'_{(v_{i-1},v_i)} \\
&= \sum_{i=1}^{k} \left(\ell_{(v_{i-1},v_i)} + h_{v_{i-1}} - h_{v_i}\right) \\
&= \sum_{i=1}^{k} \ell_{(v_{i-1},v_i)} + h_{v_0} - h_{v_k} \\
&= \ell(P) + h_{v_0} - h_{v_k}
\end{aligned}$$

$h_{v_0} - h_{v_1} + h_{v_1} - h_{v_2} + \ldots$

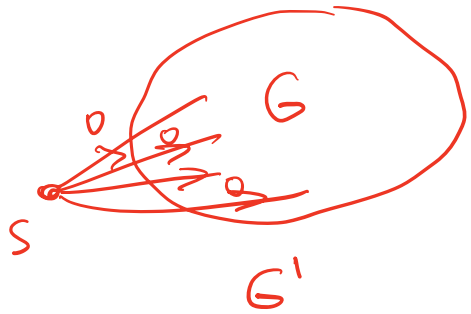
- Donc, non seulement le plus court chemin, mais *tout* chemin P de u à v vérifie $\ell'(P) = \ell(P) + h_u - h_v$.

Preuve du lemme (2/2)

- En particulier, si P est un chemin de u à v, alors $\ell(P) = \mathrm{dist}_\ell(u, v)$ ssi $\ell'(P) = \mathrm{dist}_{\ell'}(u, v)$.
- La longueur de cycles ne change pas si l'on passe de la pondération ℓ à ℓ' (car on a $v_0 = v_k$ dans l'équation de la diapo précédente).
- En particulier, il n'y a pas de cycle négatif par rapport à ℓ ssi il n'y a pas de cycle négatif par rapport à ℓ'.

Comment trouver la repondération ?

- Il suffit de prouver l'existence d'une pondération $\ell' \in \mathbb{R}^m$ t.q. $\ell' \geq 0$.
- Soit G' le graphe construit à partir de G en ajoutant un nouveau sommet s et les arcs $\{(s, v) : v \in V\}$.
- On étend la pondération ℓ à une pondération de G' en posant $\ell_{(s,v)} = 0$ pour tout $v \in V$.
- G' ne contient aucun cycle négatif ssi G ne contient aucun cycle négatif.
- Supposons que G et G' ne contiennent aucun cycle négatif.
- On définit $h_v = \text{dist}(s, v)$ pour tout sommet $v \in V(G')$.
- On a $h_v \leq h_u + \ell_{(u,v)}$ pour tout arc $(u, v) \in E(G')$.
- Donc, $\ell'_{(u,v)} = \ell_{(u,v)} + h_u - h_v \geq 0$.



Algorithme de Johnson

1. Calculer G'.
2. Appliquer Bellman–Ford à G', avec source s, pour calculer $h_v := \mathrm{dist}(s, v)$ pour tout $v \in V(G)$ (ou trouver un cycle négatif)
3. Repondérer chaque arc $(u, v) \in E(G)$ par $\ell'_{(u,v)} = \ell_{(u,v)} + h(u) - h(v)$.
4. Pour chaque $u \in V(G)$, exécuter Dijkstra pour calculer $\mathrm{dist}_{\ell'}(u, v)$ pour tout $v \in V(G)$.
5. Pour chaque couple u, v, on a $\mathrm{dist}_{\ell}(u, v) = \mathrm{dist}_{\ell'}(u, v) + h(v) - h(u)$.

Complexité de l'algorithme de Johnson

- L'étape 1 : $O(n)$
- L'étape 2 : $O(nm)$
- L'étape 3 : $O(m)$
- L'étape 4 : $O(nm + n^2 \log n)$
- L'étape 5 : $O(n^2)$
- Donc, l'algorithme de Johnson est de complexité $O(nm + n^2 \log n)$.
- Pour des graphes *peu denses*, l'algorithme de Johnson est donc plus rapide que l'algorithme de Floyd–Warshall.

Illustration de l'algorithme de Johnson