ALGORITHMES ET STRUCTURES DE DONNÉES

IFT-2008

Chapitre 5 : Graphe (partie 2)

Thierry Eude, Ichrak Hamdi

Département d'informatique et de génie logiciel

UNIVERSITÉ LAVAL

Sommaire

- Recherche de plus courts chemins dans un graphe valué
- Algorithme de Dijkstra
- Algorithme de Bellman-Ford
- Un autre algorithme de Tri topologique pour graphes orientés acycliques
- Belleman-Ford pour graphes orientés acycliques
- Plus courts chemins pour tout couple de sommets: algorithme de Floyd/Warshall

Problématique des plus courts chemins

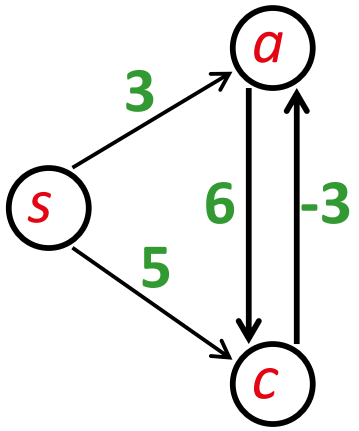
- Algorithmes nous permettant de trouver tous les plus courts chemins **d'une origine unique**. Nous allons voir:
  - Algorithme de Dijkstra en $O(n^2)$ mais tous les poids doivent être ≥ 0
  - Algorithme de Bellman-Ford en $O(nm)$ et permet les poids négatifs
  - Algorithme Bellman-Ford pour graphe orientés acyclique en $O(n+m)$.

- Donne aussi, indirectement, des algorithmes permettant de trouver **le plus court chemin pour un couple (origine s, destination t)**
  - On ne connaît pas d'algorithme pour ce problème spécifique qui soit asymptotiquement meilleur en pire cas qu'un algorithme à origine unique.
  - L'algorithme A^* permet de trouver possiblement plus rapidement le plus court chemin entre (s,t) mais seulement dans certaines conditions:
    - ✓ On doit fournir une heuristique (i.e., une fonction bornant inférieurement la distance de tout nœud à la destination) valide qui est adaptée au réseau.

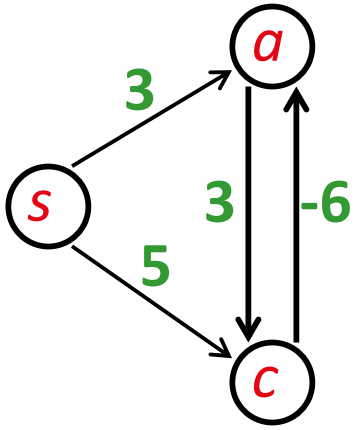
Problématique des arcs (ou arêtes) de poids négatifs

- Certains algorithmes (comme Bellman-Ford et Floyd) tolèrent la présence de poids négatifs. Mais d'autres (comme Dijkstra) ne les tolèrent pas.
- Mais s'il existe des arcs de poids négatifs, il ne faut pas que ceux-ci engendrent **un cycle de longueur négative** car, dans ce cas, **il n'existe pas de plus court chemin** entre la source s et les sommets de ce cycle!

Le plus court chemin entre s et a existe. C'est $s - c - a$.

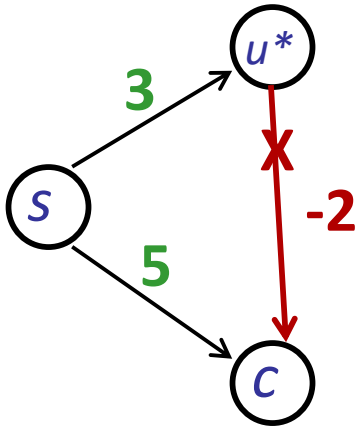


Le plus court chemin entre s et a n'existe pas! Nous pouvons toujours trouver un plus court chemin en ajoutant un cycle $a - c - a$ de plus…



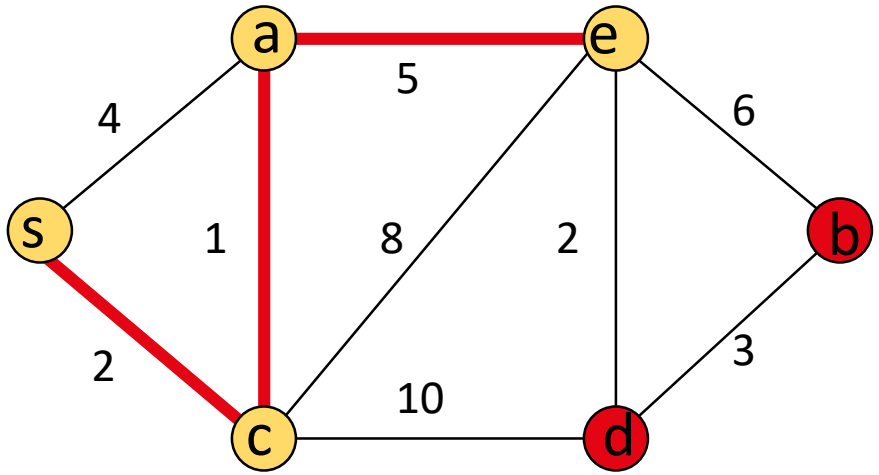
Algorithme de Dijkstra: stratégie utilisée

- **Problème à résoudre:** on a un graphe orienté valué $G(S, A, w)$ et un sommet source s de S. On veut obtenir le plus court chemin entre s et chacun des sommets de $S \setminus \{s\}$.
  - $w(u, v)$ dénote le poids de l'arc allant de u à v.
- **Dijkstra utilise une approche gloutonne (vorace)** :
  - On trouve d'abord le nœud u^* le plus près de s.
  - **s'il n'existe pas de poids négatifs**, u^* est le nœud adjacent à s dont $w(s, u^*)$ est minimal !



Algorithme de Dijkstra: stratégie utilisée

- Ensuite on ajoute u^* à l'ensemble des nœuds solutionnés les plus près de s
- Observation: après i itérations, les i nœuds les plus près de s formeront un arbre T_i.
- Arbre T_3 formé par la source s et ses trois nœuds les plus près: c, e et a.



Algorithme de Dijkstra: stratégie utilisée

- Problème: À l'itération $i + 1$ on devra trouver le nœud u^* le plus près de s parmi les nœuds qui ne sont pas dans T_i.
  - Observation: u^* est forcément un des nœuds adjacents à T_i car, étant donné que les poids sont tous non négatifs, tout nœud non adjacent à T_i ne peut que s'éloigner d'avantage (ou demeurer à la même distance) de T_i par rapport à un nœud adjacent de T_i.
  - Comment alors trouver ce u^* ?
  - Réponse: on utilise la technique du relâchement

Algorithme de Dijkstra: le relâchement

- Technique consistant à mettre à jour, pour certains nœuds u de S, un majorant (une borne supérieure) $d(u)$ de la distance du plus court chemin allant de s à u.
  - Initialement on a: $d(s) = 0$ et $d(u) = +\infty$ pour tout u dans $S \setminus \{s\}$. Car, au début, c'est le plus petit majorant que l'on a pour le plus court chemin allant de s à u.
- Afin de pouvoir reconstruire les plus courts chemins on détermine également le prédécesseur $p(u)$ de u pour notre meilleur estimé du plus court chemin de s à u.

Algorithme de Dijkstra: le relâchement

- Soit un nœud v dans l'arbre T_i et un nœud u adjacent à v qui n'est pas dans T_i, le relâchement pour l'arc (v, u) est la séquence d'opérations suivante:
  - Si $d(v) + w(v,u) < d(u)$ ALORS
    - ✓ $d(u) = d(v) + w(v,u)$;//une estimation moins pessimiste de $d(u)$ a été trouvée
    - ✓ $p(u) = v$; //le prédécesseur de u a changé
  - //sinon ne rien faire car on n'a pas trouvé un chemin plus court vers u
- Notez qu'après un relâchement de (v, u), $d(u)$ demeure encore un majorant (borne supérieur) de la distance du plus court chemin allant de s à u.

Algorithme de Dijkstra (suite)

- L'estimation $d(v)$ est exacte pour tous les nœuds v dans l'arbre T_i (i.e, $d(v)$ est vraiment la distance du plus court chemin de s à v)
- Après avoir relâcher tous les arcs (v, u) tel que v est dans T_i et u est adjacent à v sans être dans T_i , l'estimation $d(u^*)$ du nœud u^* le plus près de s qui n'est pas dans T_i , est également exacte!
  - Si v^* est le prédécesseur de u^* sur le plus court chemin de s à u^*, on a que $\mathrm{d}(u^*) = d(v^*) + w(v^*, u^*)$.
- Ainsi, après avoir relâché tous les arcs (v, u) tel que v est dans T_i et que u est adjacent à v sans être dans T_i , le noeud u^* minimisant $d(u)$ est le nœud qui est situé le plus près de s parmi ceux qui ne sont pas dans T_i
  - Ce nœud u^* sera donc le prochain nœud solutionné.
  - Donnant alors $T_{i+1} = T_i \cup \{u^*\}$

Algorithme de Dijkstra (suite)

- L'algorithme de Dijkstra, (pseudo-code page suivante), consiste simplement à construire cette séquence $T_0, T_1, \ldots, T_{n-1}$ d'arbres de noeuds solutionnés.
  - Donc, lorsque T_{n-1} est obtenu, pour tout v dans S, on a que $d(v)$ est égal à la distance du plus court chemin de s à v et le prédécesseur de v sur ce chemin est donné par $p(v)$.

Algorithme de Dijkstra: pseudo code

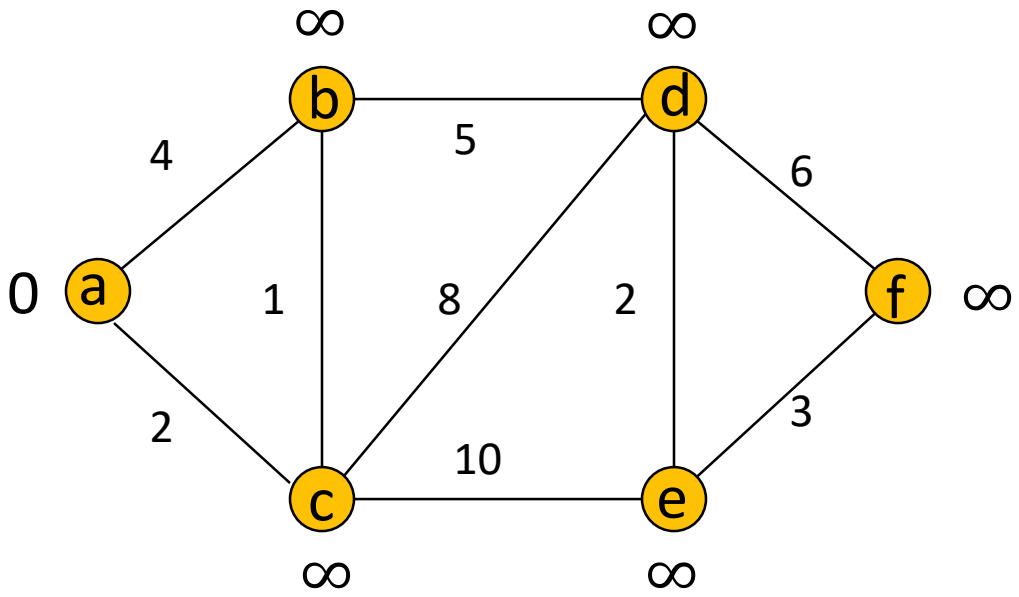
- Entrée: un Graphe orienté valué $G(S, A, w)$ avec poids non négatifs et un sommet s.
- Sortie: la longueur $d(v)$ et le prédécesseur $p(v)$ du plus court chemin allant de s à v pour tous les sommets v de S.
  - POUR tout v dans S FAIRE //initialisation de d et p
    - ✓ $d(v) = +\infty$;
    - ✓ $p(v) = NIL$;
  - $d(s) = 0$;
  - $T = \{\}$; // T est l'ensemble des nœuds solutionnés
  - $Q = S$; //Q est l'ensemble des nœuds non solutionnés
  - RÉPÉTER $|S|$ FOIS
    - ✓ u^* = Le nœud u dans Q tel que $d(u)$ est minimal;
    - ✓ $Q = Q \setminus \{u^*\}$; //enlever de Q le nœud solutionné u^*
    - ✓ $T = T \cup \{u^*\}$; //mettre u^* dans l'ensemble des nœuds solutionnés T
    - ✓ POUR tout u dans Q $(= S \setminus T)$ adjacent à u^* FAIRE
      - $temp = d(u^*) + w(u^*, u)$;
      - Si $temp < d(u)$ ALORS //relâchement pour (u^*, u)
        - $d(u) = temp$;
        - $p(u) = u^*$;

Algorithme de Dijkstra: remarques

- Lorsque l'on enlève l'élément u^* de Q tel que $d(u^*)$est minimal, on le place dans l'arbre T (des nœuds solutionnés). Par la suite, il suffit de relâcher les arcs (u^*, u) tel que u est adjacent à u^* sans être dans T.
- Il n'est pas nécessaire de relâcher les autres arcs quittant T car cela a été fait aux itérations précédentes. Donc chaque arc de A sera relâché une seule fois durant l'exécution de l'algorithme.
- L'algorithme fonctionne aussi bien avec des arcs ou des arêtes.

Algorithme de Dijkstra : trace

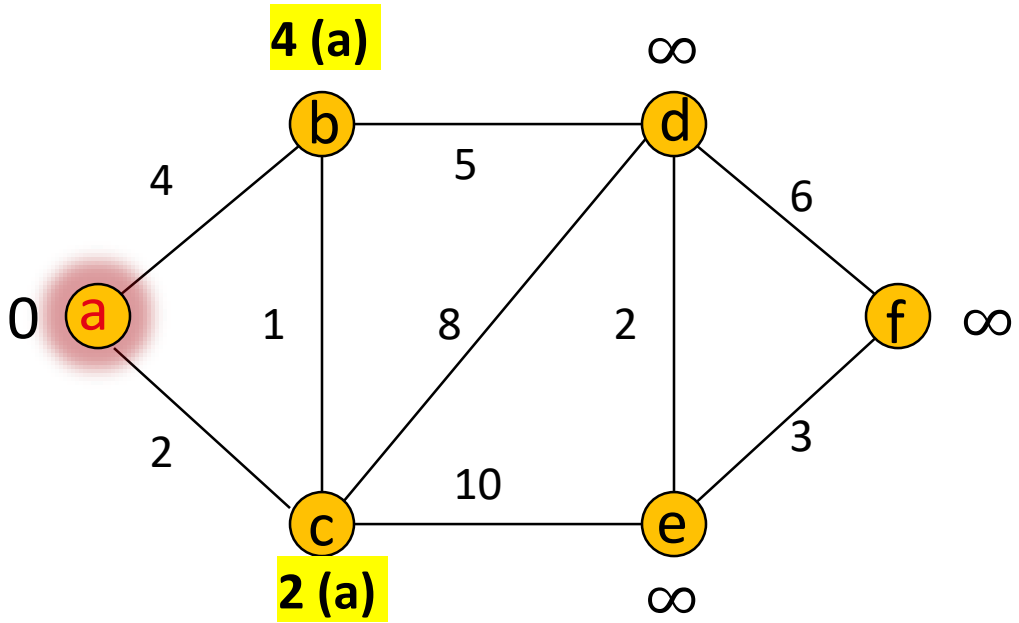
- Initialisation :
  - POUR tout v dans S FAIRE
    - ✓ $d(v) = +\infty;$
    - ✓ $p(v) = NIL;$
  - $d(s) = 0;$
  - $T = \{\};$
  - $Q = S;$



T						
Q	a	b	c	d	e	f
D	0	∞	∞	∞	∞	∞
P	-	-	-	-	-	-

Algorithme de Dijkstra : trace

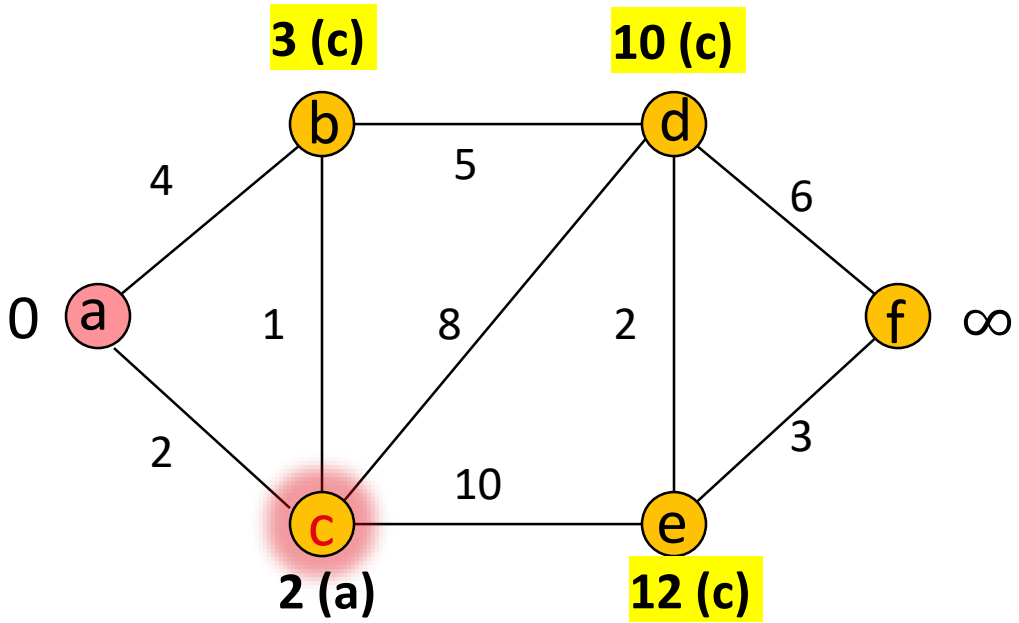
- RÉPÉTER $|S|$ FOIS
  - u^* = Le nœud u dans Q tel que $d(u)$ est minimal;
  - $Q = Q \setminus \{u^*\}$;
  - $T = T \cup \{u^*\}$;
  - POUR tout u dans Q $(= S \setminus T)$ adjacent à u^*
    - ✓ $temp = d(u^*) + w(u^*, u)$;
    - ✓ Si $temp < d(u)$
      - $d(u) = temp$;
      - $p(u) = u^*$;



T	a					
Q		b	c	d	e	f
D	0	4	2	∞	∞	∞
P	-	a	a	-	-	-

Algorithme de Dijkstra : trace

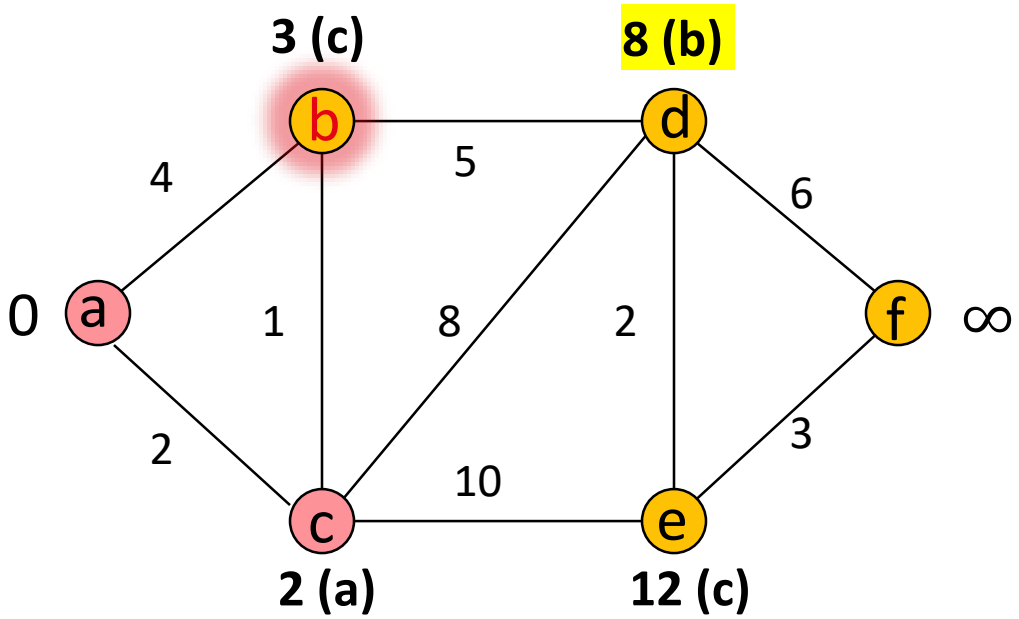
- RÉPÉTER $|S|$ FOIS
  - u^* = Le nœud u dans Q tel que $d(u)$ est minimal;
  - $Q = Q \setminus \{u^*\}$;
  - $T = T \cup \{u^*\}$;
  - POUR tout u dans Q $(= S \setminus T)$ adjacent à u^*
    - ✓ $temp = d(u^*) + w(u^*, u)$;
    - ✓ Si $temp < d(u)$
      - $d(u) = temp$;
      - $p(u) = u^*$;



T	a		c			
Q		b		d	e	f
D	0	3	2	10	12	∞
P	-	c	a	c	c	-

Algorithme de Dijkstra : trace

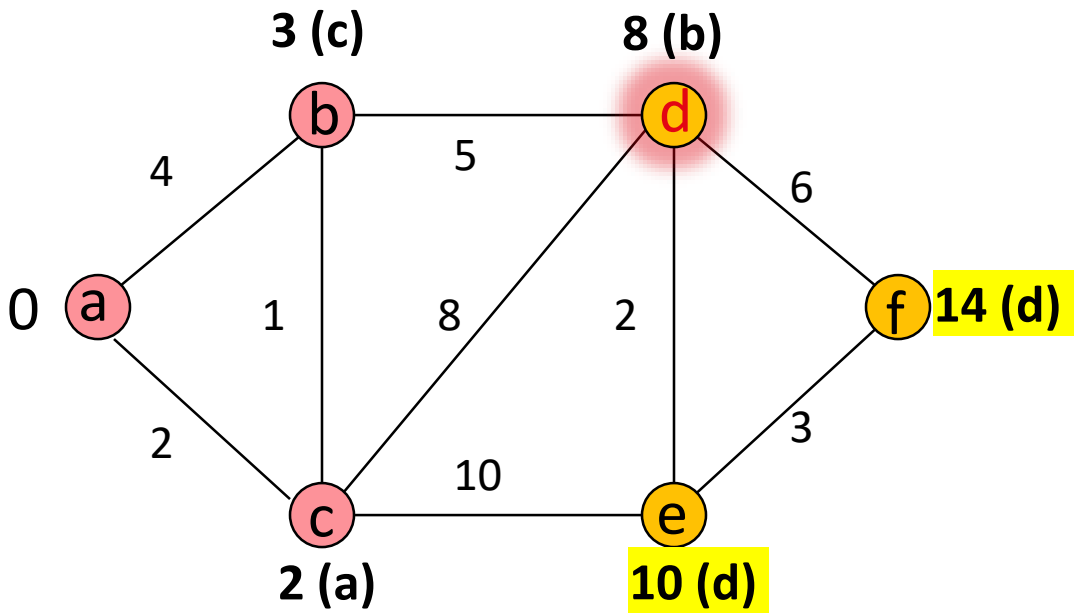
- RÉPÉTER $|S|$ FOIS
  - u^* = Le nœud u dans Q tel que $d(u)$ est minimal;
  - $Q = Q \setminus \{u^*\}$;
  - $T = T \cup \{u^*\}$;
  - POUR tout u dans Q $(= S \setminus T)$ adjacent à u^*
    - ✓ $temp = d(u^*) + w(u^*, u)$;
    - ✓ Si $temp < d(u)$
      - $d(u) = temp$;
      - $p(u) = u^*$;



T	a	b	c			
Q				d	e	f
D	0	3	2	8	12	∞
P	-	c	a	b	c	-

Algorithme de Dijkstra : trace

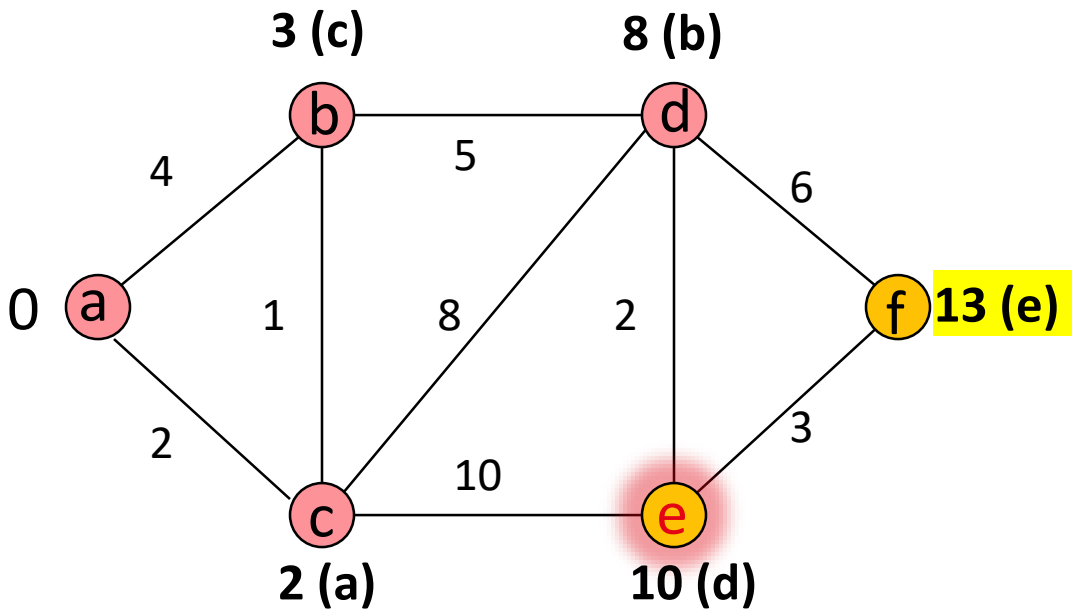
- RÉPÉTER $|S|$ FOIS
  - u^* = Le nœud u dans Q tel que $d(u)$ est minimal;
  - $Q = Q \setminus \{u^*\}$;
  - $T = T \cup \{u^*\}$;
  - POUR tout u dans Q $(= S \setminus T)$ adjacent à u^*
    - ✓ $temp = d(u^*) + w(u^*, u)$;
    - ✓ Si $temp < d(u)$
      - $d(u) = temp$;
      - $p(u) = u^*$;



T	a	b	c.	d		
Q					e	f
D	0	3	2	8	10	14
P	-	c	a	b	d	d

Algorithme de Dijkstra : trace

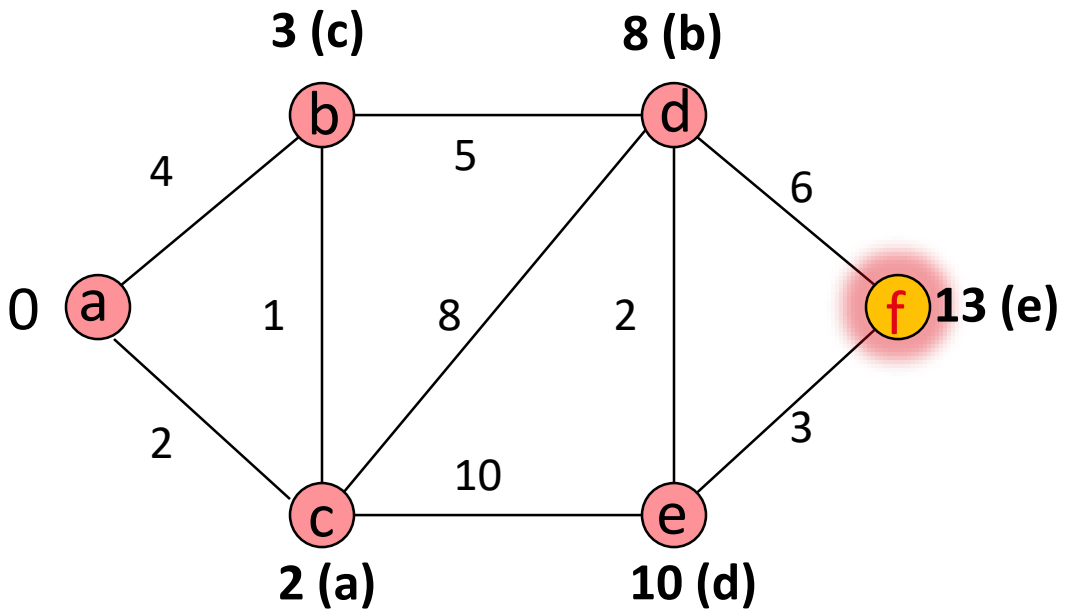
- RÉPÉTER $|S|$ FOIS
  - u^* = Le nœud u dans Q tel que $d(u)$ est minimal;
  - $Q = Q \setminus \{u^*\}$;
  - $T = T \cup \{u^*\}$;
  - POUR tout u dans Q $(= S \setminus T)$ adjacent à u^*
    - ✓ $temp = d(u^*) + w(u^*, u)$;
    - ✓ Si $temp < d(u)$
      - $d(u) = temp$;
      - $p(u) = u^*$;



T	a	b	c	d	e	
Q						f
D	0	3	2	8	10	13
P	-	c	a	b	d	e

Algorithme de Dijkstra : trace

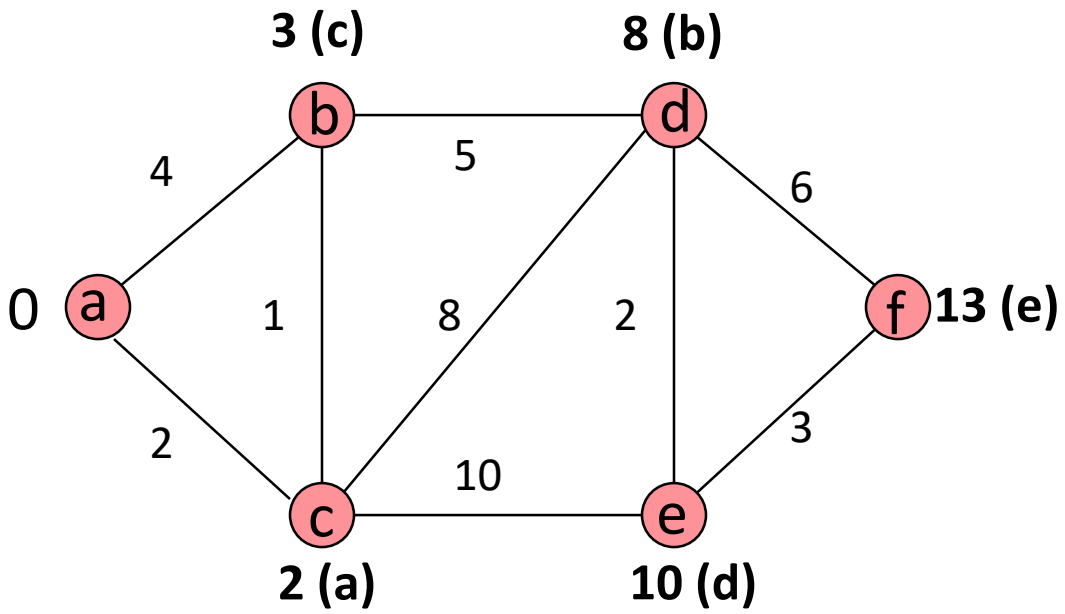
- RÉPÉTER $|S|$ FOIS
  - u^* = Le nœud u dans Q tel que $d(u)$ est minimal;
  - $Q = Q \setminus \{u^*\}$;
  - $T = T \cup \{u^*\}$;
  - POUR tout u dans Q $(= S \setminus T)$ adjacent à u^*
    - ✓ $temp = d(u^*) + w(u^*, u)$;
    - ✓ Si $temp < d(u)$
      - $d(u) = temp$;
      - $p(u) = u^*$;



T	a	b	c	d	e	f
Q						
D	0	3	2	8	10	13
P	-	c	a	b	d	e

Algorithme de Dijkstra : trace

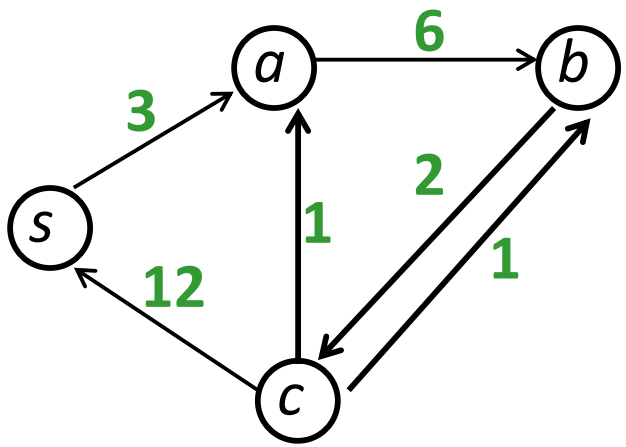
- RÉPÉTER $|S|$ FOIS
  - u^* = Le nœud u dans Q tel que $d(u)$ est minimal;
  - $Q = Q \setminus \{u^*\}$;
  - $T = T \cup \{u^*\}$;
  - POUR tout u dans Q $(= S \setminus T)$ adjacent à u^*
    - $temp = d(u^*) + w(u^*, u)$;
    - Si $temp < d(u)$
      - $d(u) = temp$;
      - $p(u) = u^*$;



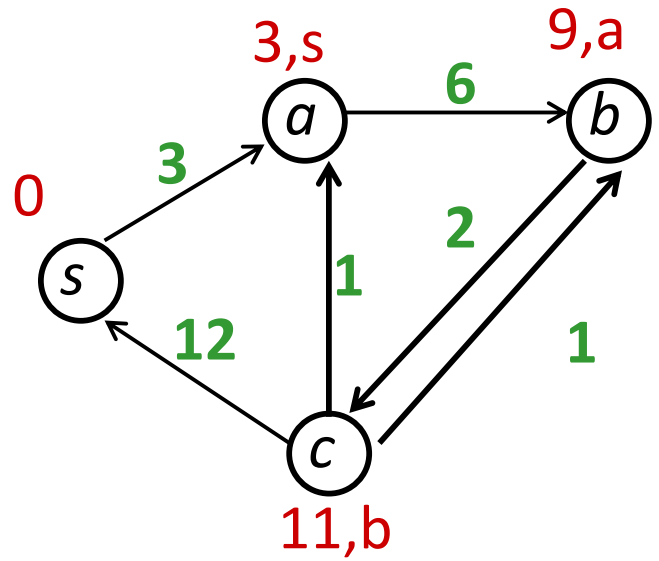
T	a	b	c	d	e	f
Q						
D	0	3	2	8	10	13
P	-	c	a	b	d	e

Exercice

- Appliquer l'algorithme de Dijkstra sur ce graphe à partir du sommet s



Solution



Algorithme de Dijkstra: analyse du temps d'exécution

- À chacune des $|S| = n$ itérations: ça prend un temps $O(|Q|)$ pour trouver u dans Q minimisant $d(u)$.
- Si on utilise une matrice W d'adjacence:
  - Le relâchement des arcs sortant de u^* se fait en $O(|Q|)$ car pour tous les sommets v dans Q, il faut examiner si l'arc (u^*, v) est dans G.
  - Puisque $|Q|$ décroît de 1 à chaque itération, le temps total est donc en

$$O(\, n \, + \, (n-1) \, + \, \ldots \, + \, 1 \,) \, = O(n^2) \, = O(|S|^2).$$

Algorithme de Dijkstra: analyse du temps d'exécution

- Si on utilise des listes d'adjacence et un vecteur $b[1,..n]$ de booléens tel que $b[u]$ = «vrai» si et seulement si le sommet u est solutionné:
  - Le relâchement des arcs sortant de u^* se fait en $O(|adjacents(u^*)|)$
  - Mais, en raison de la recherche dans Q à chaque itération, on a quand même un temps d'exécution en $O(\ n\ +\ (n-1) +\ \ldots\ +\ 1\ ) = O(n^2) = O(|S|^2)$.
- Si on utilise un tas min (voir chapitre sur les monceaux) pour Q et qu'on utilise des listes d'adjacence pour les arcs, on peut démontrer que le temps d'exécution est en $O((n+m)\ log(n))$ pour un graphe de n sommets et m arcs.
  - Ce qui constitue un avantage seulement pour les graphes peu denses.

Algorithme de Bellman-Ford (objectif)

- L'algorithme de Dijkstra ne supporte pas des poids négatifs.
- L'algorithme Bellman-Ford fonctionne avec des poids négatifs mais au prix d'un temps d'exécution plus long que celui de Dijkstra.
- Rappel: il ne doit pas exister de cycle de longueur négative qui soit accessible depuis la source s.
  - Dans ce cas, les plus courts chemins entre s et certains autres sommets de S n'existent pas.
- L'algorithme de Bellman-Ford examine si cela est le cas
  - retourne FAUX s'il existe un cycle de longueur négative accessible depuis la source s.

Algorithme de Bellman-Ford (pseudo code)

- Entrée: un Graphe orienté $valué$ $G(S, A, w)$ et un sommet source s de S.
- Sortie: Retourne FAUX s'il existe un cycle de poids négatif accessible depuis s. Sinon, retourne VRAI et retourne la longueur $d(v)$ et le prédécesseur $p(v)$ du plus court chemin allant de s à E pour tout sommet v de S.
- POUR tout v dans S FAIRE //initialisation de d et p
  - $d(v) = +\infty$; $p(v) = NIL$;
- $d(s) = 0$;
- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;
- POUR tout (u, v) de A FAIRE //vérification de l'existence d'un cycle de longueur < 0
  - Si $d(v) > d(u) + w(u, v)$ ALORS retourner FAUX;
- retourner VRAI;

Algorithme de Bellman-Ford (analyse)

- En raison de la double boucle principale, l'algorithme s'exécute en un temps $O(|S|\,|A|) = O(nm)$.
  - La boucle de vérification à la fin de l'algorithme nécessite un temps $O(|A|)$ et ne change donc pas l'ordre de croissance du temps d'exécution car, selon la règle du maximum, on a que $O(nm + m) = O(\max(nm, m)) = O(nm)$.

Algorithme de Bellman-Ford (validité)

- Pour l'analyse de la validité de l'algorithme: démontrons d'abord qui si G ne contient aucun cycle de longueur négative qui soit accessible depuis s, alors pour tout v dans S, lorsque la boucle principale aura terminé, nous aurons que $d(v)$ sera égal à la distance $D(s, v)$ du plus court chemin de s à v.
  - Notez que dans ce cas, $p(v)$ sera le bon prédécesseur de v sur le plus court chemin de s à v.
- Pour prouver cet énoncé nous utiliserons la propriété du relâchement des plus courts chemins:
- Si c = $< v_0, v_1, .., v_k >$ est un plus court chemin de s = v_0 à v_k, et si les arcs sont relâchés dans l'ordre (v_0, v_1), (v_1, v_2), …, (v_{k-1}, v_k), alors d(v_k) = D(s, v_k) = distance du plus court chemin de s à v_k. Cela est vrai indépendamment de toutes les autres étapes intermédiaires de relâchement pouvant se produire.

Algorithme de Bellman-Ford (validité)

- Pour prouver cet énoncé nous utiliserons la propriété du relâchement des plus courts chemins:
  - Si $c = < v_0, v_1, .., v_k >$ est un plus court chemin de $s = v_0$ à v_k, et si les arcs sont relâchés dans l'ordre (v_0, v_1), (v_1, v_2), …, (v_{k-1}, v_k),
  - alors $d(v_k) = D(s, v_k)$ = distance du plus court chemin de s à v_k. Cela est vrai indépendamment de toutes les autres étapes intermédiaires de relâchement pouvant se produire.

Algorithme de Bellman-Ford (validité)

- Preuve de la validité :
  - Soit un sommet v accessible depuis s et soit c = $< v_0, v_1, .., v_k >$ un plus court chemin de s = v_0 à v = v_k.
  - Ce plus court chemin c ne peut contenir de cycle car tous les cycles ont une longueur non négative.
  - Donc c contient au plus $|S| - 1$ arcs. Donc on a que $k \leq |S| - 1$.
  - Or, chacune des $|S| - 1$ itérations relâche tous les $|A|$ arcs.
  - Donc, l'arc (v_0, v_1) a été relâché à l'itération 1.
  - Et l'arc (v_{i-1}, v_i) a été relâché à l'itération i.
  - Après $|S| - 1$ itérations les arcs ont donc été relâchés dans l'ordre $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ avec, possiblement, d'autres relâchements intermédiaires.
  - D'après la propriété du relâchement des plus courts chemins, on a que $d(v) = D(s, v)$ pour tout plus court chemin allant de s à v. CQFD.

Algorithme de Bellman-Ford (validité)

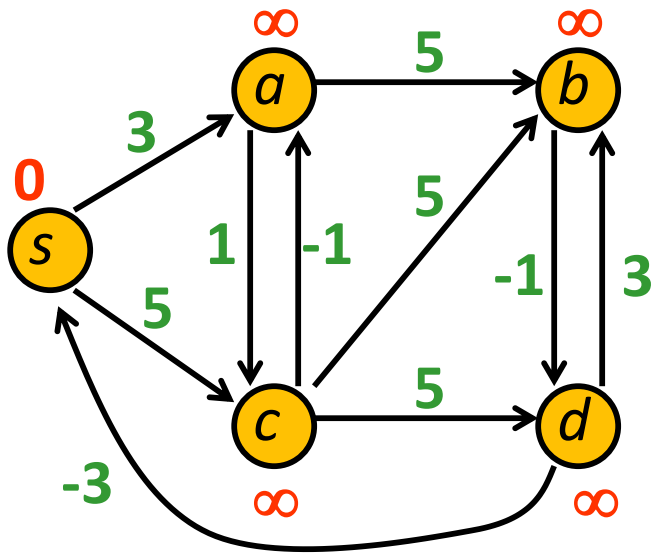
- Donc, si G ne contient aucun cycle de longueur négative qui soit accessible depuis s, alors pour tout v accessible de s, on a $d(v) = d(p(v)) + w(p(v), v) \leq d(u) + w(u, v)$ pour tous les autres u connectés à v.
- Si v n'est pas accessible de s, on a que $d(v) = +\infty = d(u) + w(u, v)$ pour tout u connecté à v
  - (car, u est alors non accessible depuis s et, dans ce cas, $d(u) = +\infty$)
- Donc, pour tous les arcs (u, v) de A on a $d(v) \leq d(u) + w(u, v)$
- Donc aucun des arcs (u, v) de A ne satisfera $d(v) > d(u) + w(u, v)$ et l'algorithme retournera VRAI.

Algorithme de Bellman-Ford (validité)

- Inversement, si l'algorithme retourne VRAI, nous avons $d(v_i) \leq d(v_{i-1}) + w(v_{i-1}, v_i)$ pour tout cycle $< v_0, v_1, .., v_k >$ (avec $v_k = v_0$) accessible de s. Cela implique que
  - $d(v_1) + \ldots + d(v_k) \leq d(v_0) + .. + d(v_{k-1}) + w(v_0, v_1) + \ldots + w(v_{k-1}, v_k)$
- Puisque $d(v_0) = d(v_k)$, cela implique que $w(v_0, v_1) + \ldots + w(v_{k-1}, v_k) \geq 0$.
- Donc si l'algorithme retourne VRAI, tous les cycles accessibles de s doivent avoir un poids non négatif.
- À la page précédente, nous avons montré que si tous les cycles accessibles depuis s ont un poids non négatif alors l'algorithme retourne VRAI.
- Donc l'algorithme retourne VRAI si et seulement si tous les cycles accessibles depuis s ont un poids non négatif.
- Cela prouve la validité de l'algorithme

Algorithme de Bellman-Ford : trace

- Initialisation :
- POUR tout v dans S FAIRE
  //initialisation de d et p
  - $d(v) = +\infty$; $p(v) = NIL$;
- $d(s) = 0$;

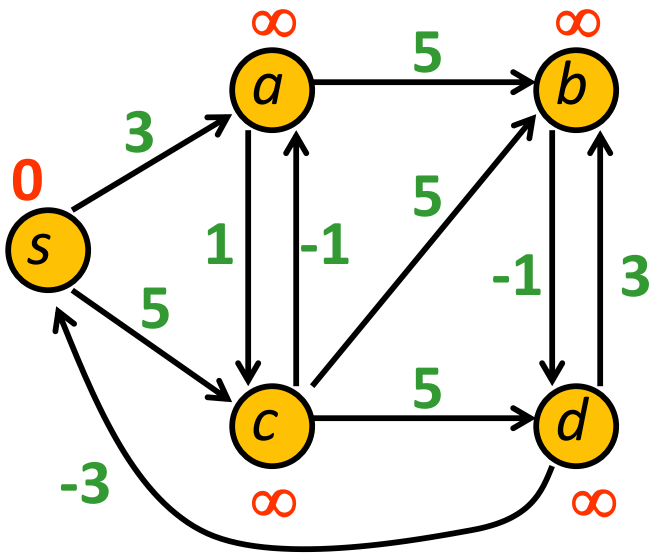


D	0	∞	∞	∞	∞
P	-	-	-	-	-

s	a	b	c	d

Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



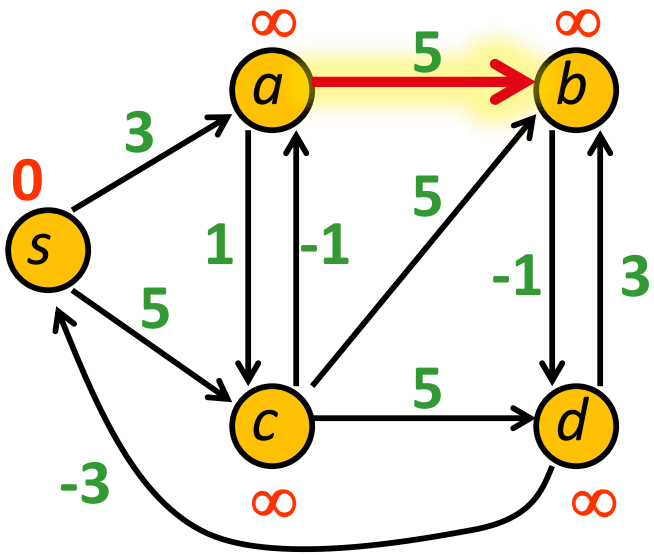
Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

D	0	∞	∞	∞	∞
P	-	-	-	-	-

s	a	b	c	d

Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

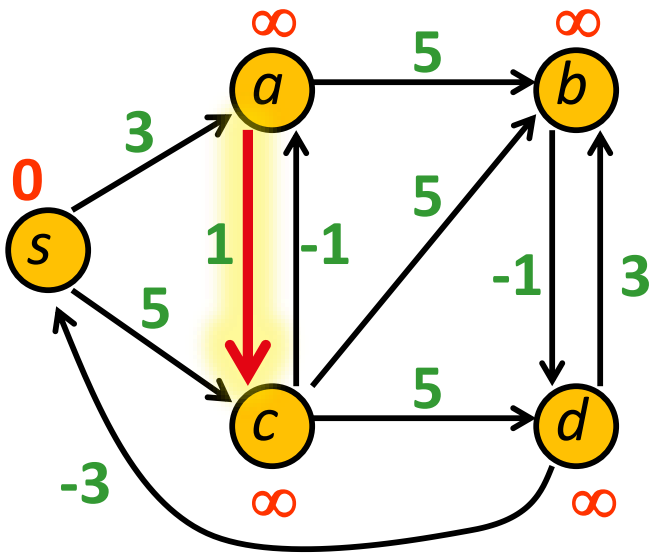
$$temp = \infty + 5 = \infty$$
$$\infty \not< \infty$$

D	0	∞	∞	∞	∞
P	-	-	-	-	-
	s	a	b	c	d

Itération #1

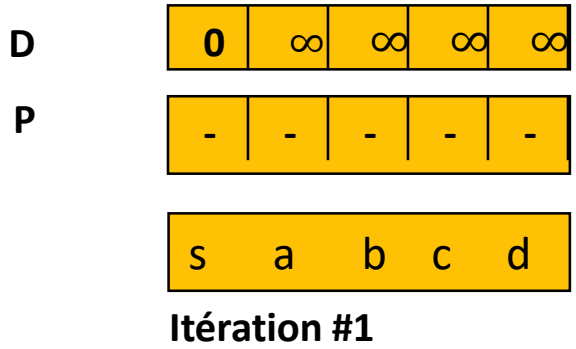
Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

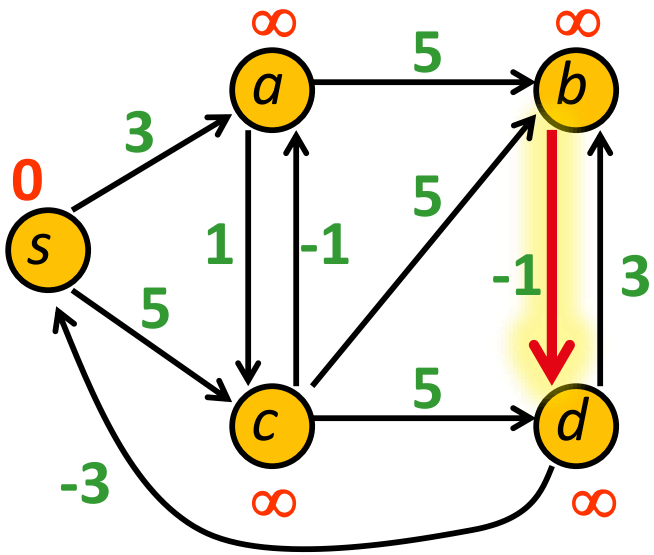
$$temp = \infty + 1 = \infty$$
$$\infty \not< \infty$$



Itération #1

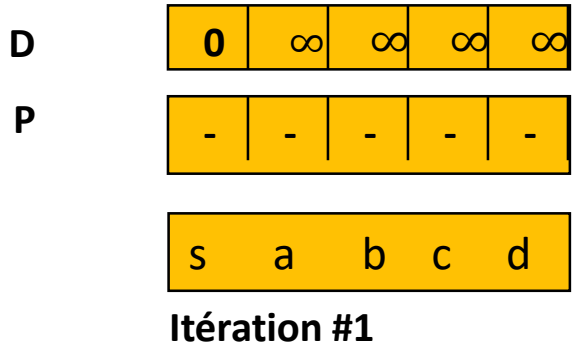
Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



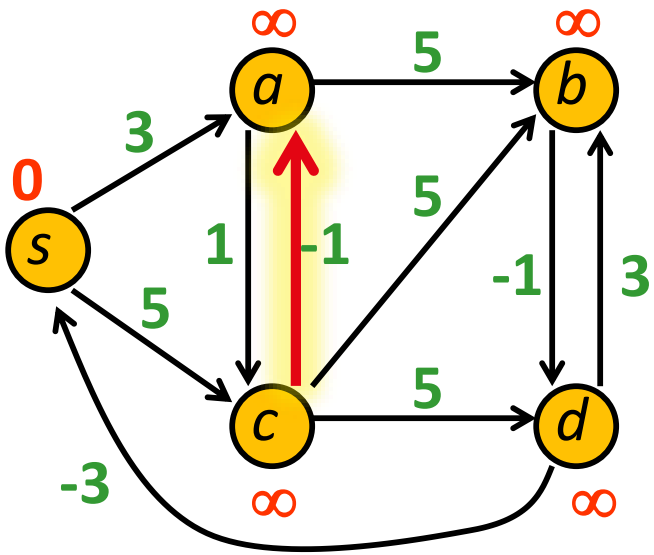
Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$temp = \infty + (-1) = \infty$$
$$\infty \not< \infty$$



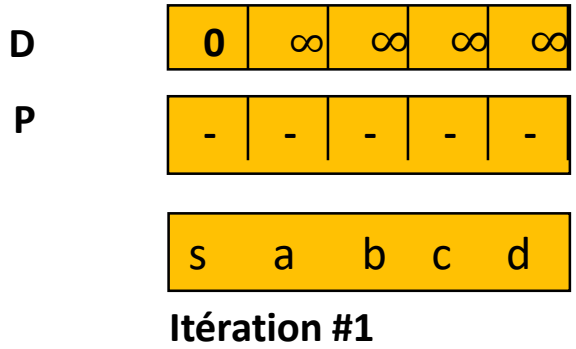
Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

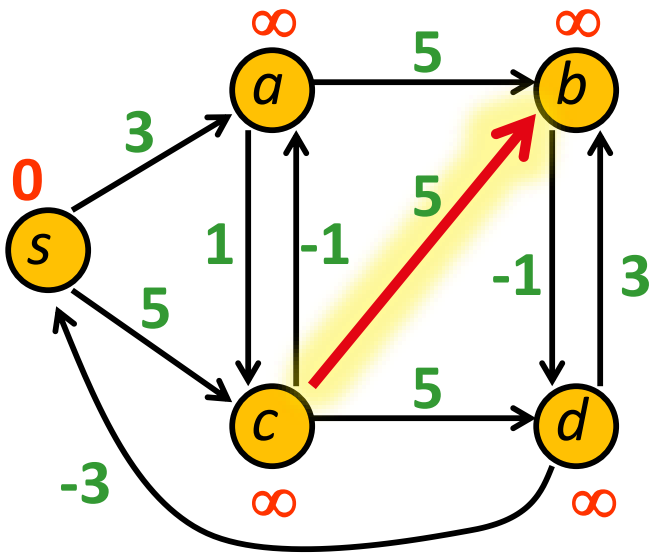
$$temp = \infty + (-1) = \infty$$
$$\infty \not< \infty$$



Itération #1

Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

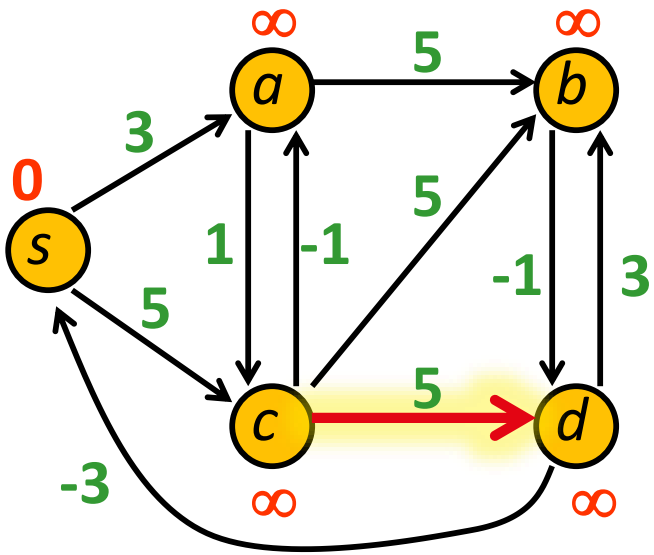
$$temp = \infty + 5 = \infty$$
$$\infty \not< \infty$$

D	0	∞	∞	∞	∞
P	-	-	-	-	-
	s	a	b	c	d

Itération #1

Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

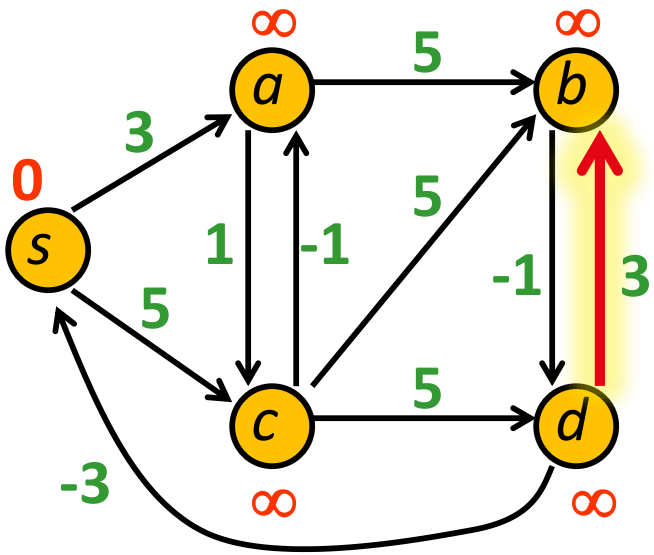
$$temp = \infty + 5 = \infty$$
$$\infty \not< \infty$$

D	0	∞	∞	∞	∞
P	-	-	-	-	-
	s	a	b	c	d

Itération #1

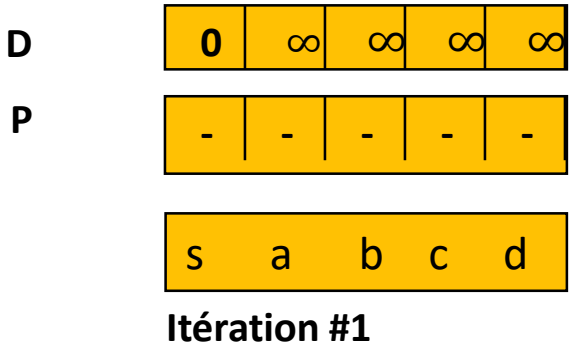
Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



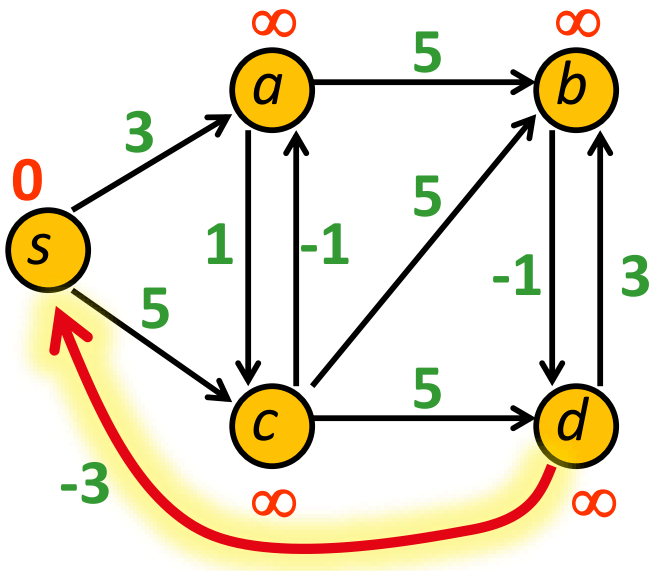
Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$temp = \infty + 3 = \infty$$
$$\infty \not< \infty$$



Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

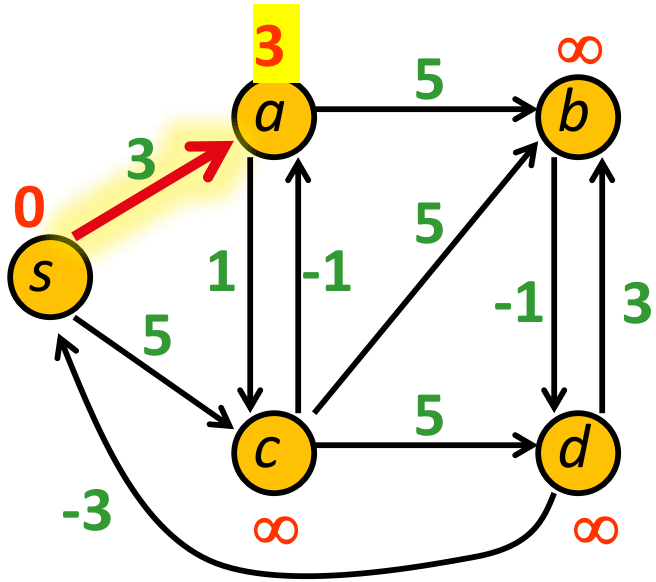
$$temp = \infty + (-3) = \infty$$
$$\infty \not< \infty$$

D	0	∞	∞	∞	∞
P	-	-	-	-	-
	s	a	b	c	d

Itération #1

Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$temp = 0 + 3 = 3$$
$$3 < \infty$$

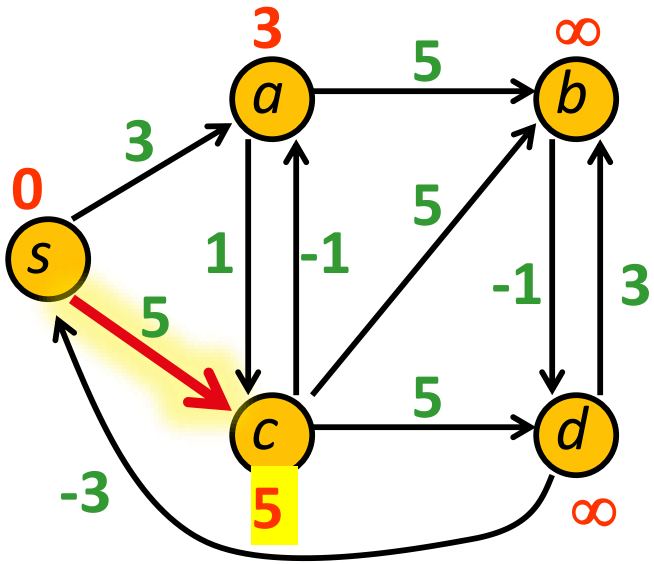
D	0	3	∞	∞	∞
P	-	s	-	-	-

s	a	b	c	d

Itération #1

Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

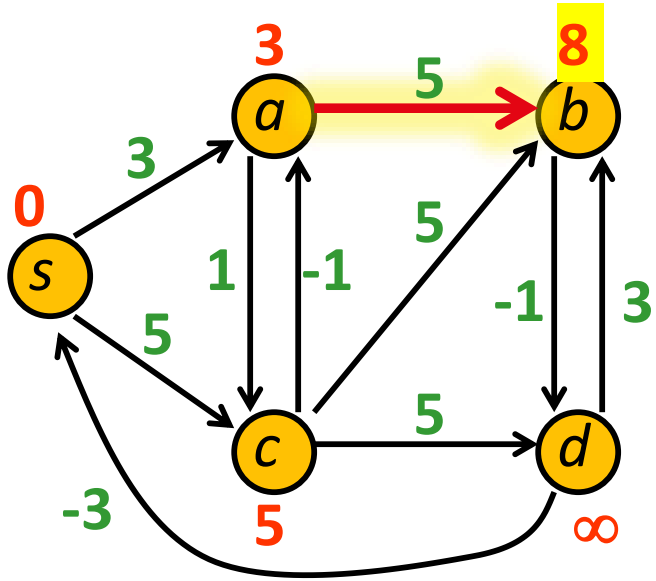
$$temp = 0 + 5 = 5$$
$$5 < \infty$$

D	0	3	∞	5	∞
P	-	s	-	s	-
	s	a	b	c	d

Itération #1

Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

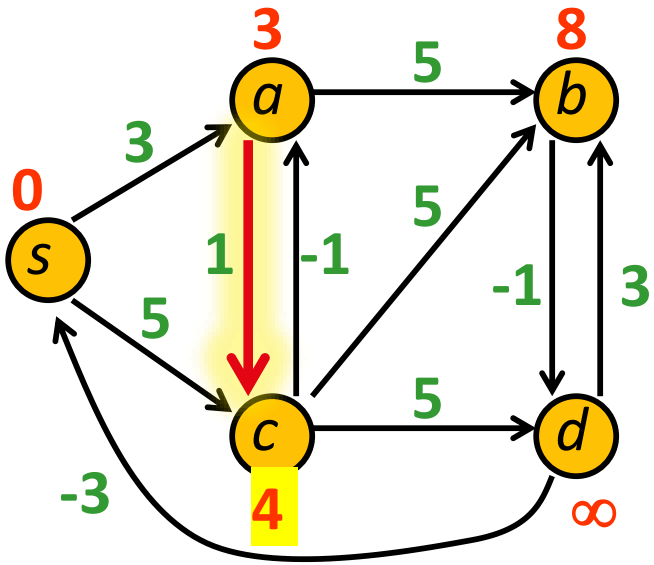
$$temp = 3 + 5 = 8$$
$$8 < \infty$$

D	0	3	8	5	∞
P	-	s	a	s	-
	s	a	b	c	d

Itération #2

Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$temp = 3 + 1 = 4$$
$$4 < 5$$

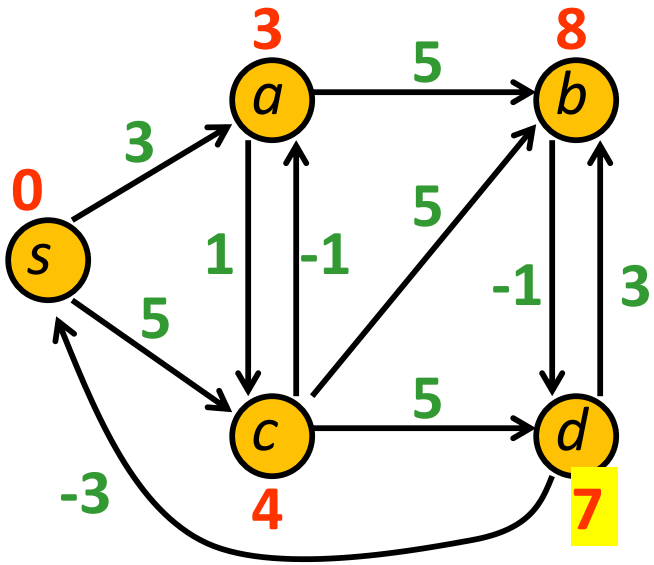
D	0	3	8	4	∞
P	-	s	a	a	-

s	a	b	c	d

Itération #2

Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$temp = 8 + (-1) = 7$$
$$7 < \infty$$

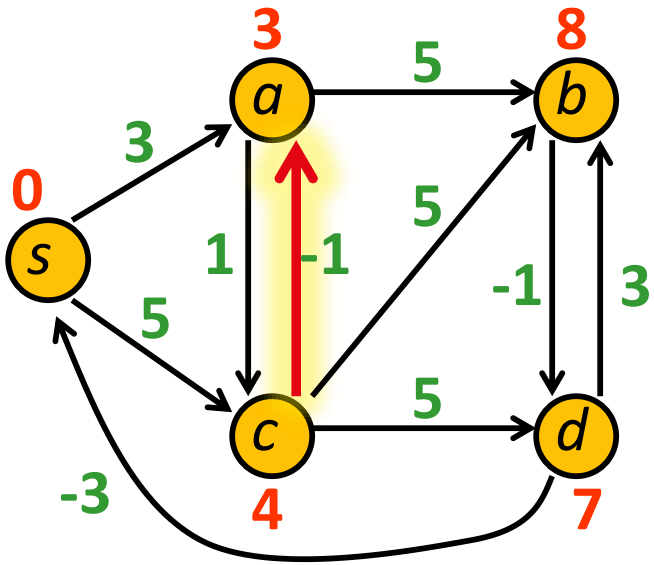
D	0	3	8	4	7
P	-	s	a	a	b

s a b c d

Itération #2

Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$temp = 4 + (-1) = 3$$
$$3 \not< 3$$

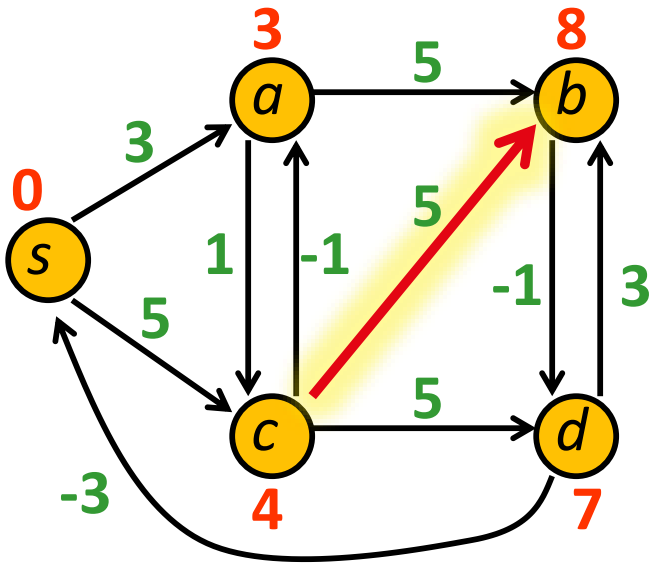
D	0	3	8	4	7
P	-	s	a	a	b

s	a	b	c	d

Itération #2

Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$temp = 4 + 5 = 9$$
$$9 \not< 8$$

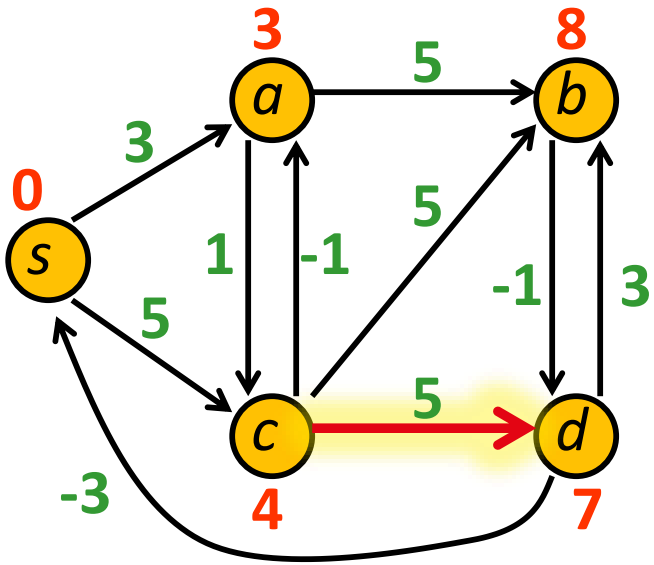
D	0	3	8	4	7
P	-	s	a	a	b

s	a	b	c	d

Itération #2

Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$temp = 4 + 5 = 9$$
$$9 \not< 7$$

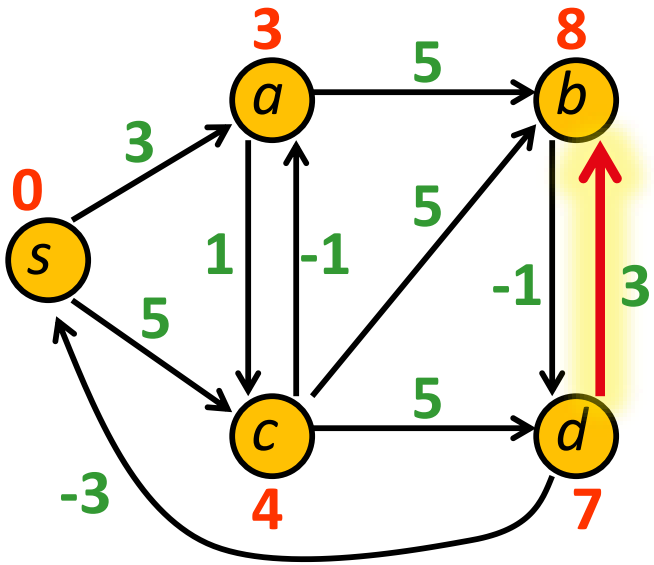
D	0	3	8	4	7
P	-	s	a	a	b

s	a	b	c	d

Itération #2

Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$temp = 7 + 3 = 10$$
$$10 \not< 8$$

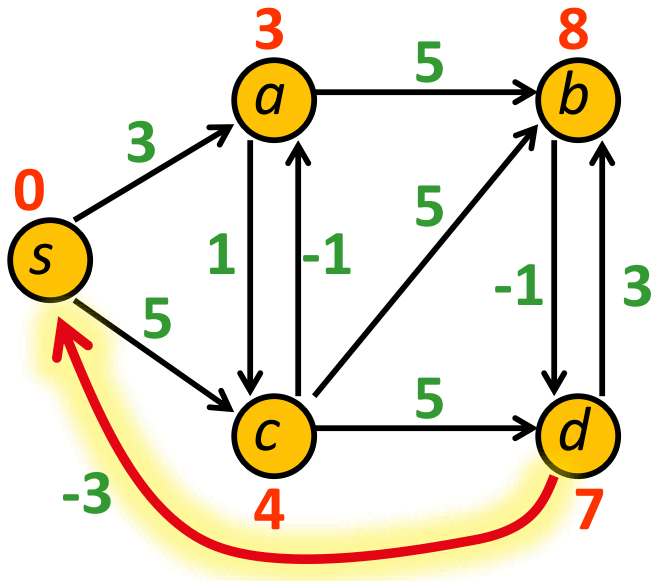
D	0	3	8	4	7
P	-	s	a	a	b

s	a	b	c	d

Itération #2

Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$temp = 7 + (-3) = 4$$
$$4 \not< 0$$

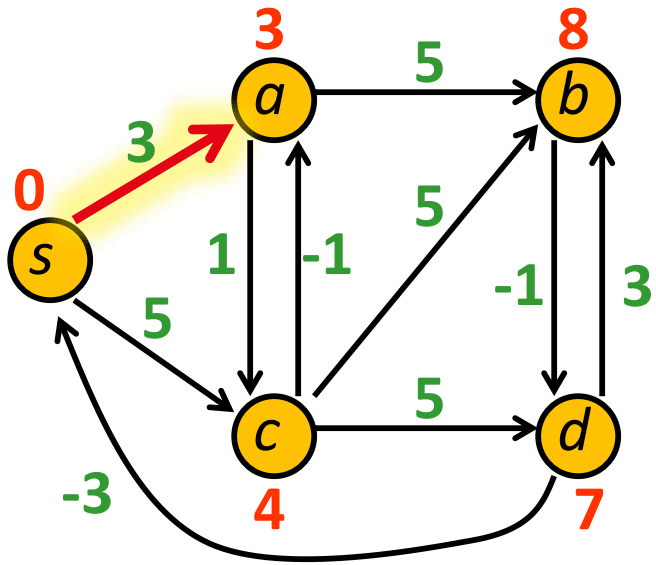
D	0	3	8	4	7
P	-	s	a	a	b

s a b c d

Itération #2

Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$temp = 0 + 3 = 3$$
$$3 \not< 3$$

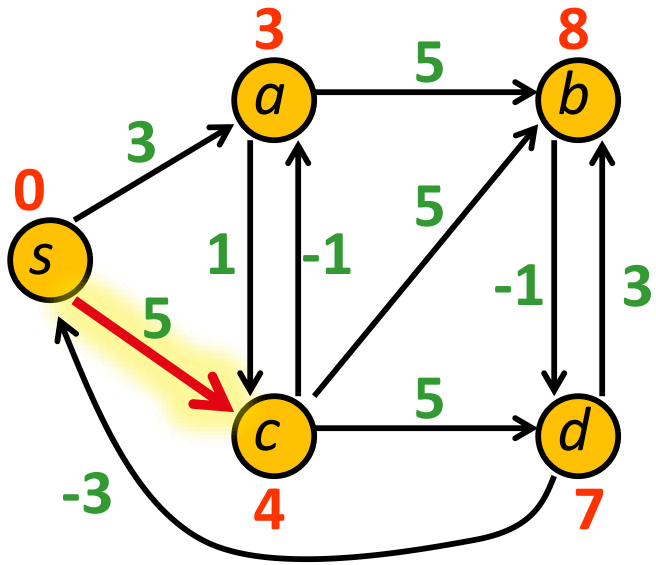
D	0	3	8	4	7
P	-	s	a	a	b

s	a	b	c	d

Itération #2

Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



Ordre des arcs :
(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$temp = 0 + 5 = 5$$
$$5 \not< 4$$

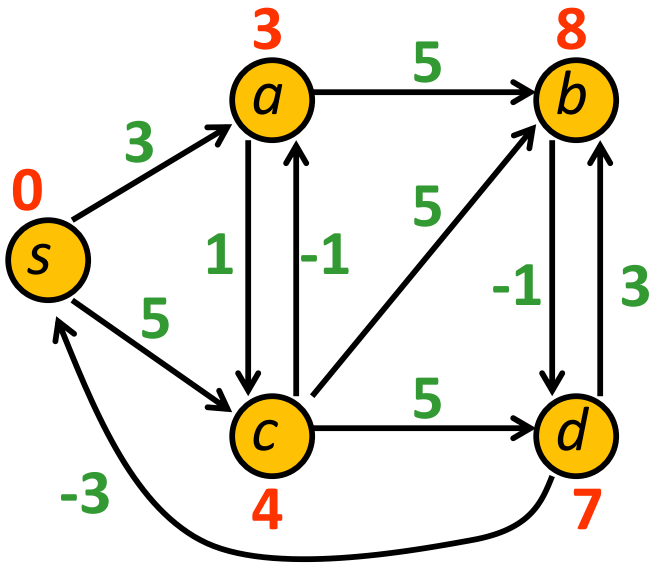
D	0	3	8	4	7
P	-	s	a	a	b

s	a	b	c	d

Itération #2

Algorithme de Bellman-Ford : trace

- RÉPÉTER $|S| - 1$ FOIS //partie principale de l'algorithme
  - ➢ POUR tout (u, v) de A FAIRE
    - ✓ $temp = d(u) + w(u, v)$;
    - ✓ SI $temp < d(v)$ //relâchement pour (u.v)
      - $d(v) = temp$; $p(v) = u$;



- On refait cela encore 2 fois (itération 3-4)
- On trouvera ce graphe.

D	0	3	8	4	7
P	-	s	a	a	b

s	a	b	c	d

Itération #2

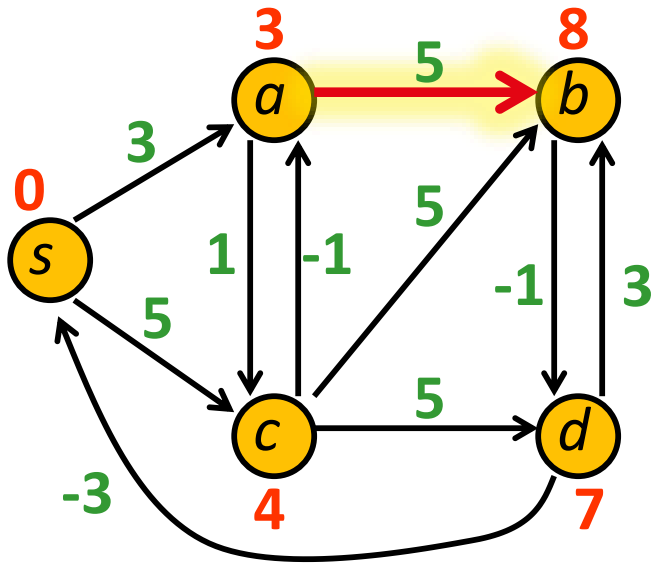
Algorithme de Bellman-Ford : trace

- POUR tout (u, v) de A FAIRE //vérification de l'existence d'un cycle de longueur < 0
  - ➢ Si $d(v) > d(u) + w(u, v)$ ALORS retourner FAUX;

Ordre des arcs :

(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$8 \not> 3 + 5$$



D	0	3	8	4	7
P	-	s	a	a	b

s	a	b	c	d

Itération #2

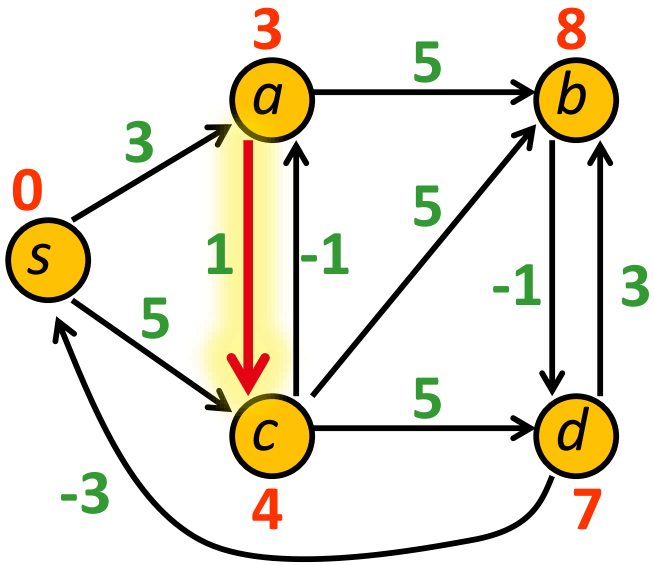
Algorithme de Bellman-Ford : trace

- POUR tout (u, v) de A FAIRE //vérification de l'existence d'un cycle de longueur < 0
  - ➢ Si $d(v) > d(u) + w(u, v)$ ALORS retourner FAUX;

Ordre des arcs :

(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$4 \not> 3 + 1$$



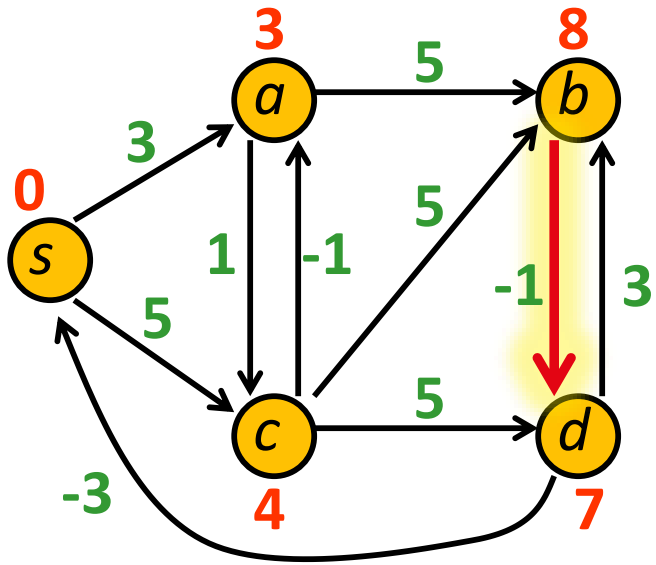
D	0	3	8	4	7
P	-	s	a	a	b

s	a	b	c	d

Itération #2

Algorithme de Bellman-Ford : trace

- POUR tout (u, v) de A FAIRE //vérification de l'existence d'un cycle de longueur < 0
  - ➢ Si $d(v) > d(u) + w(u, v)$ ALORS retourner FAUX;



Ordre des arcs :

(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$7 \not> 8 + (-1)$$

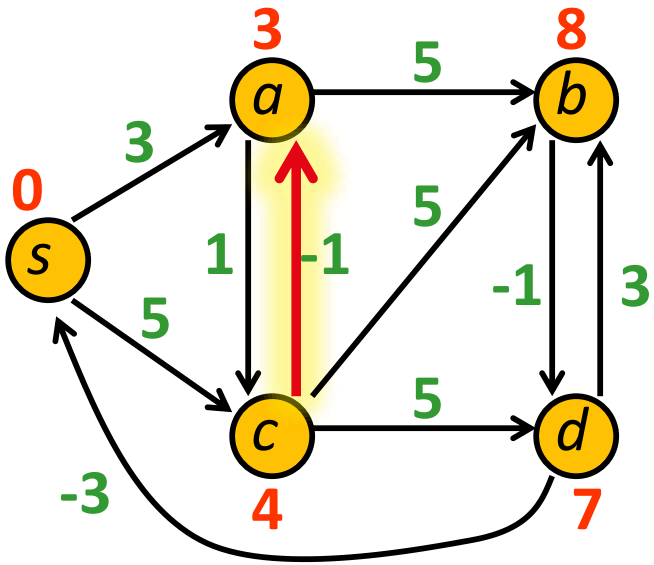
D	0	3	8	4	7
P	-	s	a	a	b

s	a	b	c	d

Itération #2

Algorithme de Bellman-Ford : trace

- POUR tout (u, v) de A FAIRE //vérification de l'existence d'un cycle de longueur < 0
  - ➢ Si $d(v) > d(u) + w(u, v)$ ALORS retourner FAUX;



Ordre des arcs :

(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$3 \not> 4 + (-1)$$

D	0	3	8	4	7
P	-	s	a	a	b

s	a	b	c	d

Itération #2

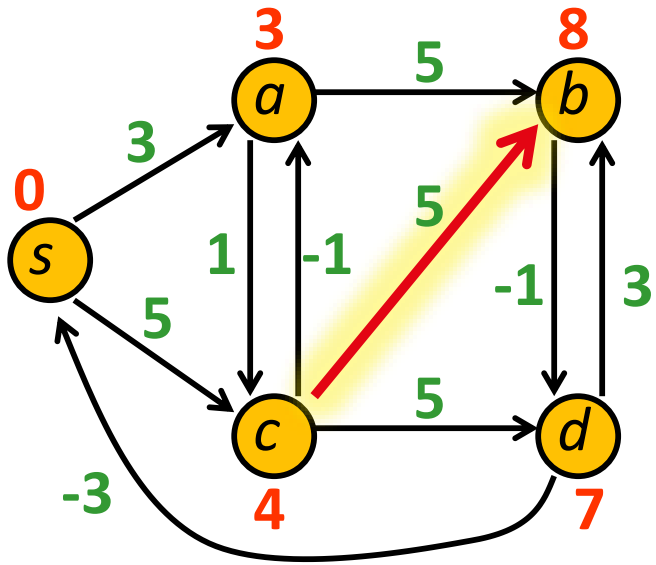
Algorithme de Bellman-Ford : trace

- POUR tout (u, v) de A FAIRE //vérification de l'existence d'un cycle de longueur < 0
  - ➢ Si $d(v) > d(u) + w(u, v)$ ALORS retourner FAUX;

Ordre des arcs :

(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$8 \not> 4 + 5$$



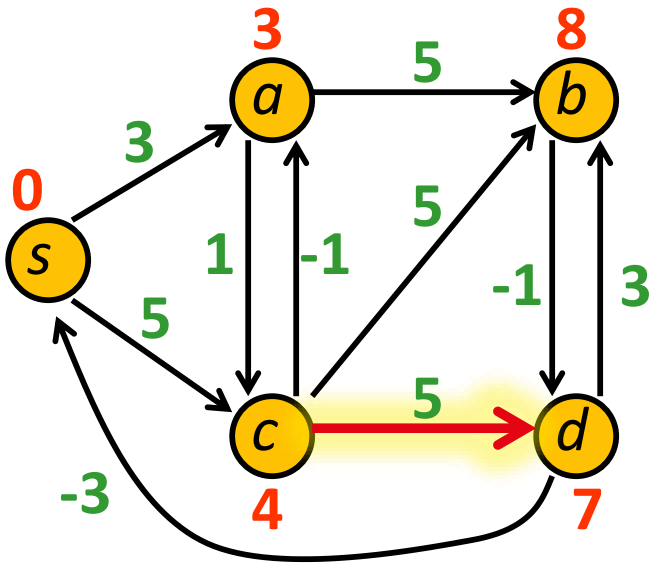
D	0	3	8	4	7
P	-	s	a	a	b

s	a	b	c	d

Itération #2

Algorithme de Bellman-Ford : trace

- POUR tout (u, v) de A FAIRE //vérification de l'existence d'un cycle de longueur < 0
  - ➢ Si $d(v) > d(u) + w(u, v)$ ALORS retourner FAUX;



Ordre des arcs :

(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$7 \not> 5 + 4$$

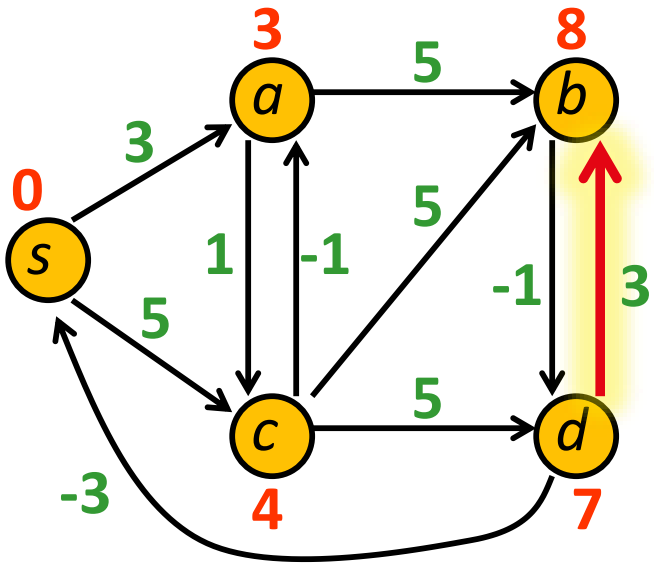
D	0	3	8	4	7
P	-	s	a	a	b

s	a	b	c	d

Itération #2

Algorithme de Bellman-Ford : trace

- POUR tout (u, v) de A FAIRE //vérification de l'existence d'un cycle de longueur < 0
  - ➢ Si $d(v) > d(u) + w(u, v)$ ALORS retourner FAUX;



Ordre des arcs :

(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$8 \not> 7 + 3$$

D	0	3	8	4	7
P	-	s	a	a	b

s	a	b	c	d

Itération #2

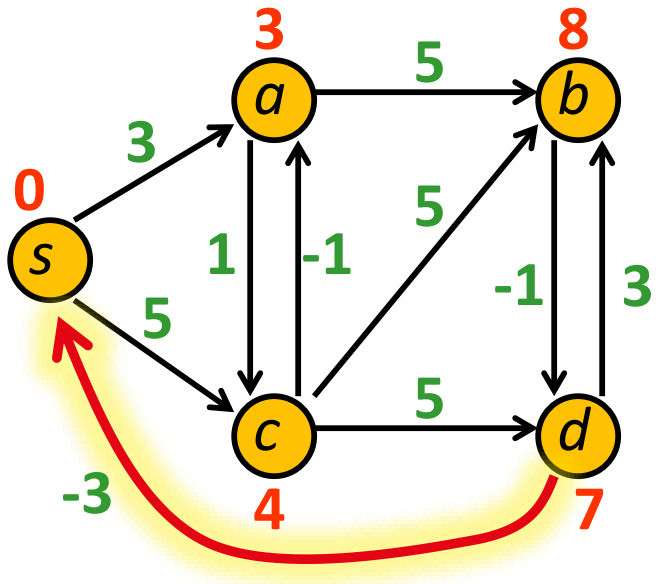
Algorithme de Bellman-Ford : trace

- POUR tout (u, v) de A FAIRE //vérification de l'existence d'un cycle de longueur < 0
  - ➢ Si $d(v) > d(u) + w(u, v)$ ALORS retourner FAUX;

Ordre des arcs :

(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$0 \not> 7 + (3)$$



D	0	3	8	4	7
P	-	s	a	a	b

s	a	b	c	d

Itération #2

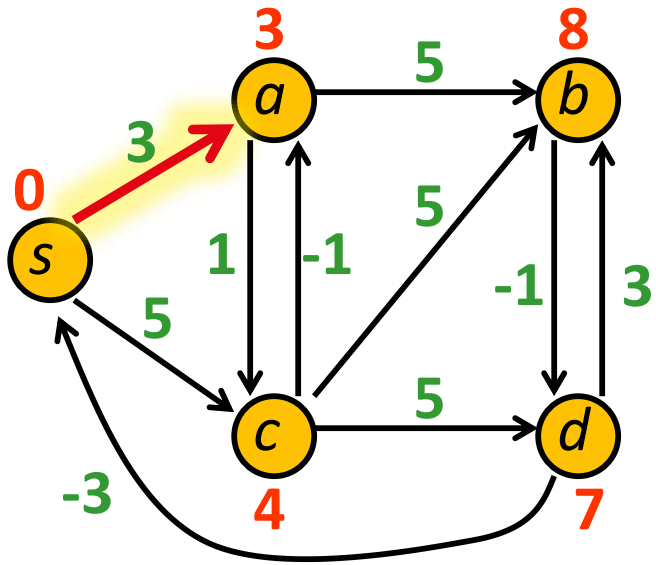
Algorithme de Bellman-Ford : trace

- POUR tout (u, v) de A FAIRE //vérification de l'existence d'un cycle de longueur < 0
  - ➢ Si $d(v) > d(u) + w(u, v)$ ALORS retourner FAUX;

Ordre des arcs :

(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$0 \not> 0 + 3$$



D	0	3	8	4	7
P	-	s	a	a	b

s	a	b	c	d

Itération #2

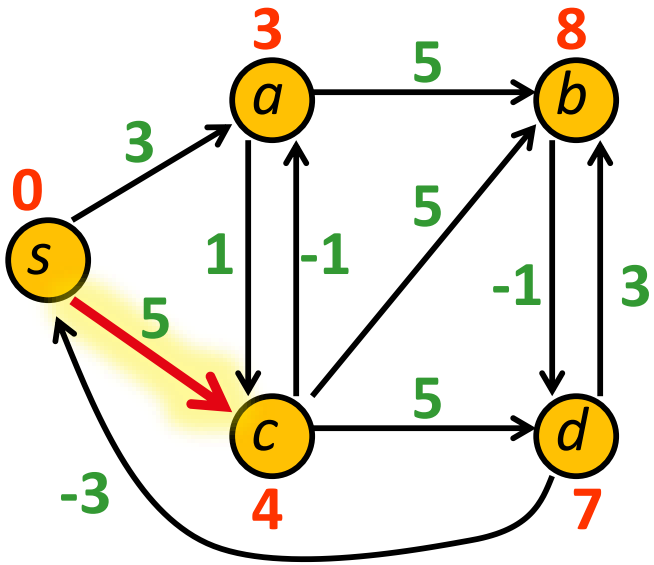
Algorithme de Bellman-Ford : trace

- POUR tout (u, v) de A FAIRE //vérification de l'existence d'un cycle de longueur < 0
  - ➢ Si $d(v) > d(u) + w(u, v)$ ALORS retourner FAUX;

Ordre des arcs :

(a,b), (a,c), (b,d), (c,a), (c,b), (c,d), (d,b), (d,s), (s,a), (s,c)

$$4 \not> 0 + 5$$

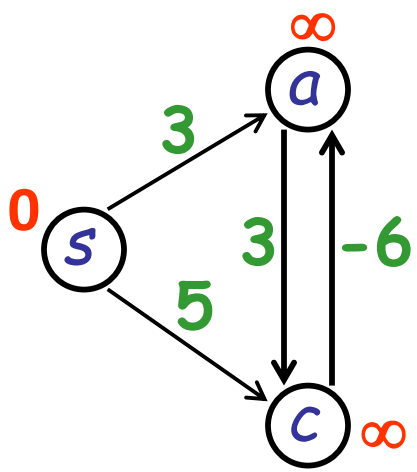


D	0	3	8	4	7
P	-	s	a	a	b

s	a	b	c	d

Itération #2

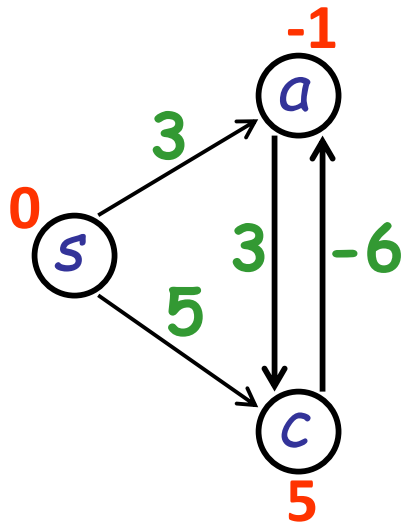
Présence d'un cycle négatif



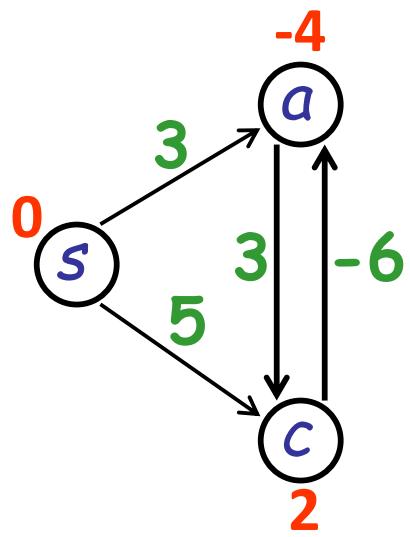
- Ordre des arcs :

(s,a)(s,c)(a,c)(c,a)

|S| = 3



- Itération 1



- Itération 2

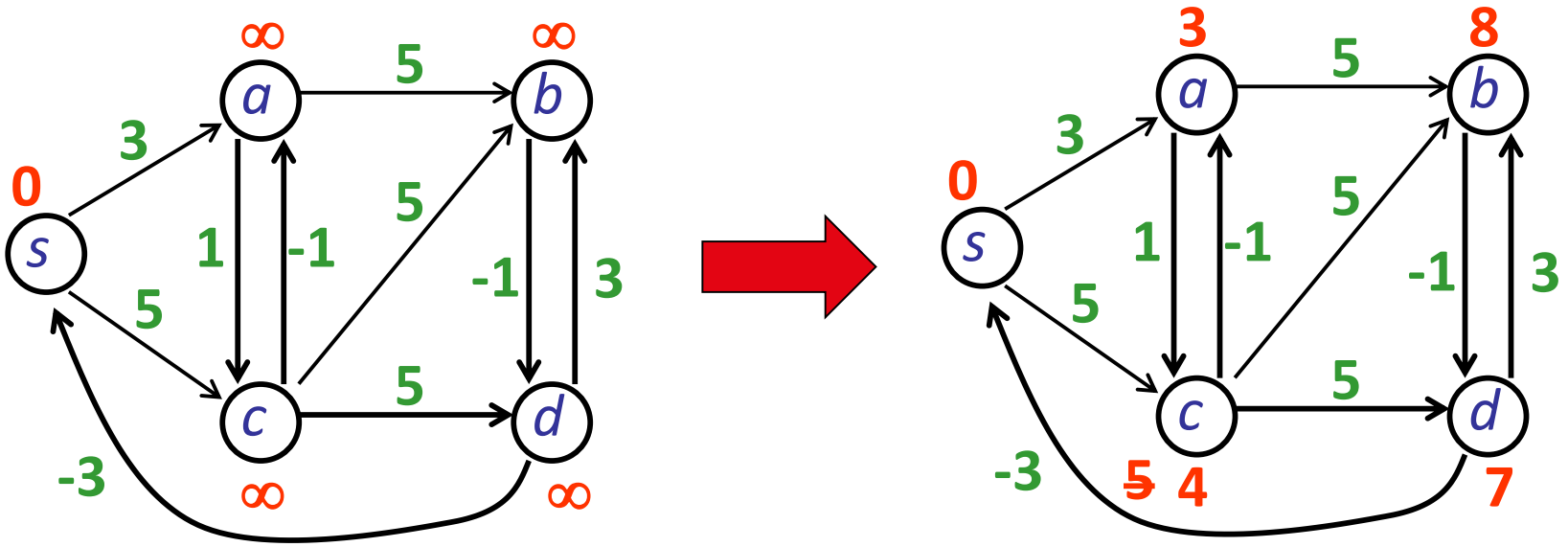
-4 ⊁ 0 + 3

2 ⊁ 0 + 5

2 > -4 + 3

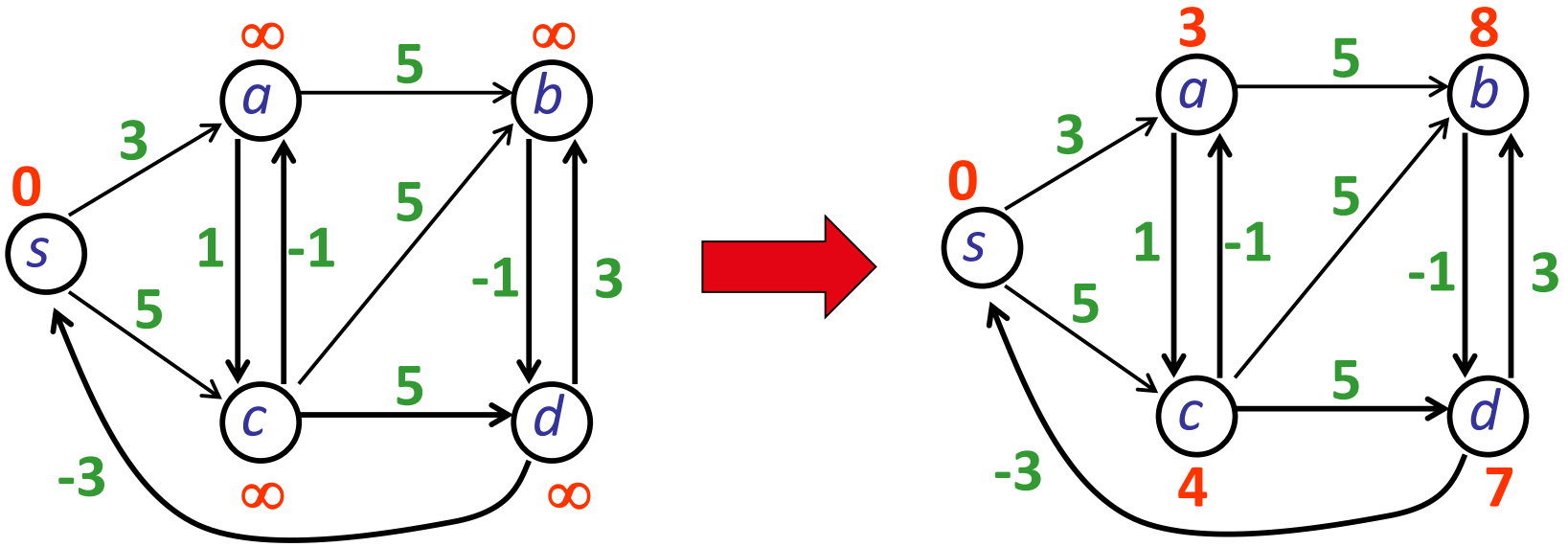
Cycle!

Algorithme de Bellman-Ford (exemple en changeant l'ordre)



Itération #1 : relaxation de tous les arcs dans un autre ordre :
(*s*,*a*) (*s*,*c*) (*a*,*b*) (*a*,*c*) (*b*,*d*) (*c*,*a*) (*c*,*b*) (*c*,*d*) (*d*,*b*) (*d*,*s*)

Algorithme de Bellman-Ford (exemple)



Itération # 2 relaxation de tous les arcs dans l'ordre :
(*s*,*a*) (*s*,*c*) (*a*,*b*) (*a*,*c*) (*b*,*d*) (*c*,*a*) (*c*,*b*) (*c*,*d*) (*d*,*b*) (*d*,*s*)
Pas de réduction possible : distances correctes!

Pour éviter de faire toutes les $|S| - 1$ itérations :
vérifier après itération si $d(v) > d(u) + w(u, v)$ pour tout (u, v) de A

Synthèse

- Plus courts chemins d'une origine unique
  - ➢ Dijkstra
    - ✓ s'il n'existe pas de poids négatifs
    - ✓ relâchement des sommets
    - ✓ fonctionne aussi bien avec des arcs ou des arêtes
    - ✓ pour reconstruire les plus courts chemins : prédécesseur
    - ✓ Si matrice W d'adjacence : $O(|S|^2)$
    - ✓ Si listes d'adjacence : $O(|S|^2)$
    - ✓ Si tas min (voir chapitre sur les monceaux) $en\ O((n+m)\ log(n))$ pour un graphe de n sommets et m arcs

Synthèse

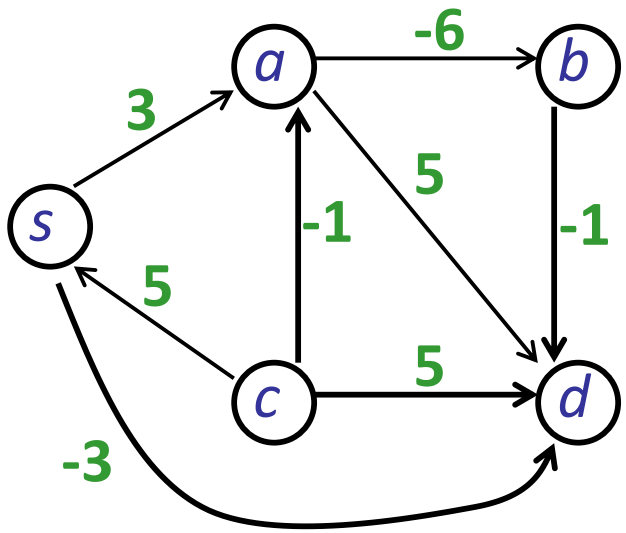
- Plus courts chemins d'une origine unique
  - ➢ Bellman-Ford
    - ✓ fonctionne avec des poids négatifs
    - ✓ relâchement des arcs
    - ✓ en $O(nm)$.
    - ✓ temps d'exécution plus long que Dijkstra.
    - ✓ il ne doit pas exister de cycle de longueur négative qui soit accessible depuis la source s.
    - ✓ Pour éviter de faire toutes les $|S| - 1$ itérations : vérifier après itération si $d(v) > d(u) + w(u, v)$ pour tout (u, v) de A

Les graphes orientés acycliques (rappel)

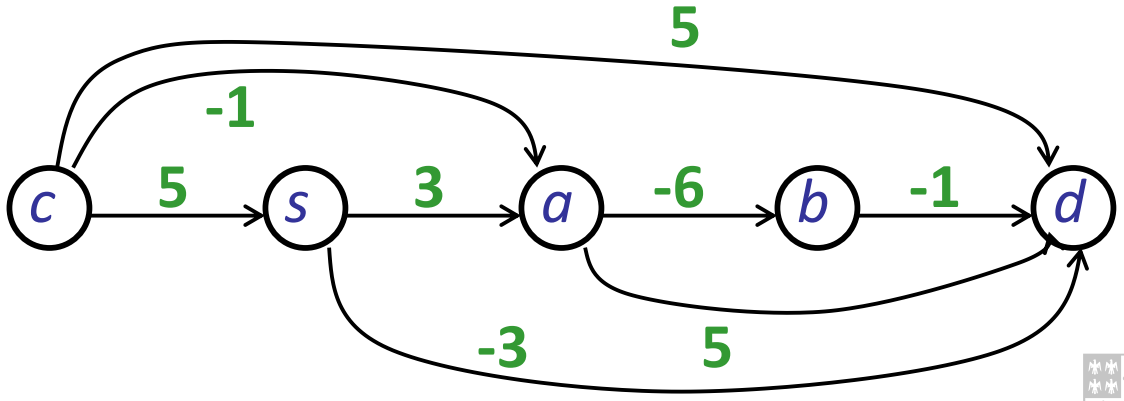
- Il arrive souvent qu'un graphe orienté soit acyclique (sans cycle)
- Exemples:
  - Graphe d'évènements: lorsque chaque nœud représente la réalisation d'un événement et qu'un arc (u, v) existe si et seulement si l'événement u précède l'événement v.
    - Dans ce cas il ne peut y avoir de cycle $< v_0, v_1, .., v_k >$ (avec $v_k = v_0$) car, dans un cycle v_i précède v_i et v_i précède v_i, ce qui viole la causalité des évènements
  - Graphe de tâches: chaque nœud représente une tâche à accomplir et un arc (u, v) existe si et seulement si la tâche u doit précéder la tâche v.
    - Dans ce cas, il ne peut y avoir de cycle car une tâche ne peut pas devoir être précédée d'elle-même.

Ordre topologique d'un graphe orienté acyclique

- Soit un graphe orienté $G = (S, A, w)$, nous disons qu'une séquence de nœuds est selon un ordre topologique de G si et seulement si pour tout arc (u, v) de G, u précède v dans cette séquence.
- Par exemple, la séquence c, s, a, b, d est un ordre topologique pour le graphe acyclique ci-dessous.

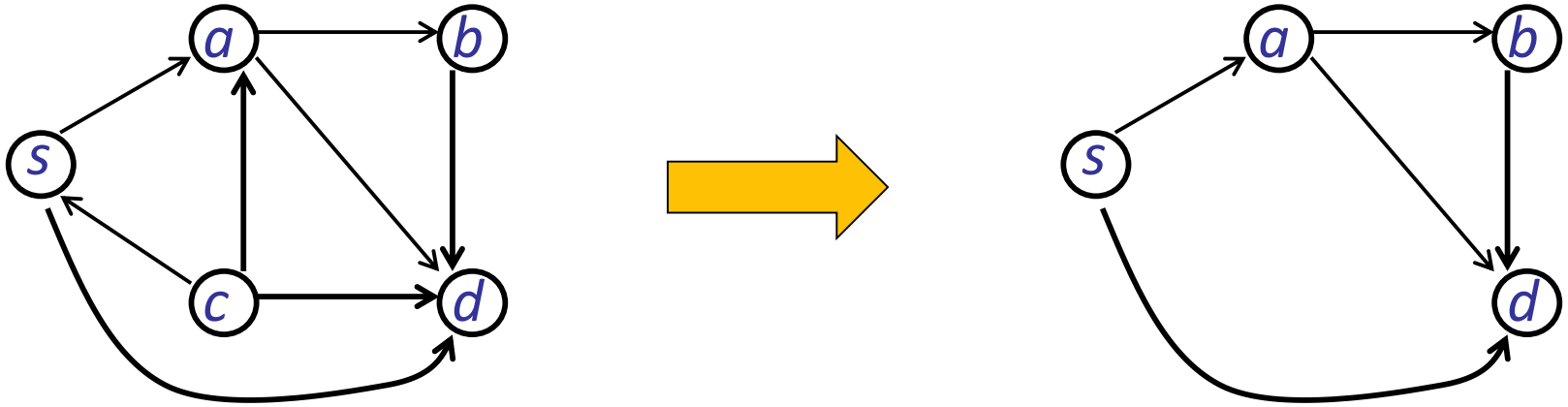


Seuls les graphes orientés acycliques admettent un ordre topologique



Tri topologique: idée de base

- Pour trier un graphe acyclique $G(S, A, w)$ dans un ordre topologique, nous pouvons d'abord identifier tous les nœuds dont l'arité d'entrée est nulle.
  - Ces nœuds doivent se trouver en première position dans le tableau trié T (l'ordre relatif entre ces nœuds n'a pas d'importance).
- Après avoir placé ces nœuds dans T, il suffit d'enlever les arcs sortant de ces nœuds. Cela décroît alors l'arité d'entrée des nœuds adjacents aux nœud placés dans T.



Tri topologique: idée de base

- Et l'on recommence avec le graphe résultant.
- Le temps d'exécution de cet algorithme est en $O(|S|^2)$ en pire cas, car après avoir enlevé les nœuds d'arité d'entrée nulle, il faut visiter tous les autres nœuds.
- Astuce pour accélérer l'algorithme: ne visiter que les nœuds adjacents aux nœuds enlevés! Cela donne l'algorithme à la page suivante.

Algorithme TriTopologique(G)

- Entrée: un Graphe orienté et valué $G(S, A, w)$.
- Sortie: Retourne VRAI et un tableau $T[1, .., |S|]$ des sommets triés dans un ordre topologique si G ne contient pas de cycle. Retourne FAUX autrement.
- $AE[1, .. |S|]$ = tableau des arités d'entrée pour chaque nœud de S;
- $Q = \{\}$; //une file de sommets initialement vide
- POUR tout v dans S FAIRE //enfiler dans Q tous les nœuds dont l'arité d'entrée == 0
  - Si $AE[v] == 0$ ALORS Q.enfiler(v) ;
- Si $Q == \{\}$ ALORS retourner (FAUX,T); //car il existe un **cycle**; T est un tableau bidon
- $i = 0$; //compteur du nombre de nœuds triés dans un ordre topologique
- TANT QUE Q n'est pas vide FAIRE //partie principale de l'algorithme
  - u = Q.défiler(); //nœud d'arité d'entrée nulle
  - T[++i] = u; //insertion du nœud dans sa bonne position dans le tableau trié T
  - POUR TOUT v adjacent à u FAIRE
    - AE[v] --; //ayant enlevé u, on « enlève » l'arc (u, v)
    - SI $AE[v] == 0$ ALORS Q.enfiler(v); //car à insérer dans T
- SI $i != |S|$ ALORS retourner (FAUX, T); //un **cycle** a forcément été détecté, T est bidon
- retourner (VRAI, T);

Tri topologique: Analyse

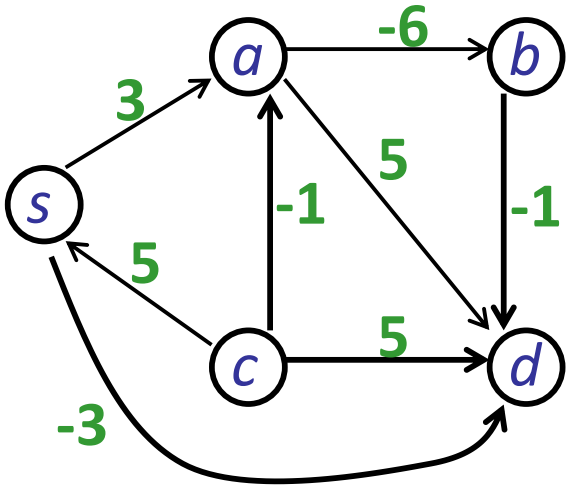
- Soit $n = |S|$ et $m = |A|$. On utilise des listes d'adjacence pour les arcs.
- La construction de $AE[1,..,n]$ se fait en $O(n+m)$ car on visite (une seule fois) tous les nœuds et la liste d'adjacence de chaque noeud.
- La première boucle POUR (qui identifie tous les nœuds de S dont l'arité d'entrée est nulle) s'exécute en temps $O(n)$.
- La boucle principale TANT QUE requiert $O(n)$ étapes, mais pour chaque étape on doit examiner tous les nœuds adjacents au nœud u courant.
  - Ce qui requiert un temps $\mathrm{O}(|adjacents(u)|)$.
- La boucle principale TANT QUE nécessitera alors un temps $O(n+m)$
- TriTopologique() s'exécute donc en $O(n+m)$.

Tri topologique: Validité

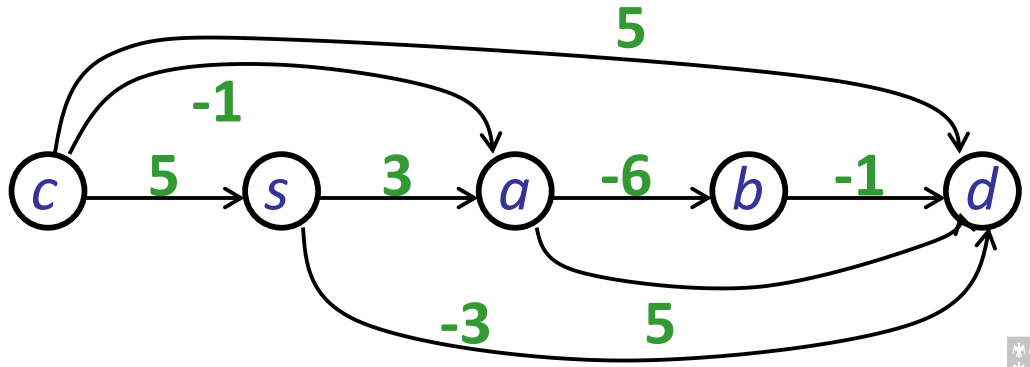
- La validité de TriTopologique() est établie lorsque G est acyclique car un graphe acyclique contient toujours un nœud source et, à chaque itération, lorsque l'on enlève un nœud source et ses arcs sortants, le graphe résultant doit demeurer acyclique.
- Si G contient un cycle, il sera détecté lorsque tous les nœuds de G possédant un arc vers un nœud du cycle auront été enlevés car, dans ce cas, il n'y aura plus de nœuds pouvant être enlevés, la liste Q sera vide, et la boucle principale TANT QUE terminera avec $i < n$.
- La validité de TriTopologique() est donc établie également lorsque G possède un cycle.

Bellman-Ford et les graphes orientés acycliques

- Lorsque le graphe est acyclique, il suffit de parcourir les nœuds dans un ordre topologique une seule fois.
- Lors de ce parcours, il suffit de relâcher (une seule fois) l'arc (u, v) pour chaque nœud v adjacent au nœud courant u.
- Cela donne l'algorithme de la page suivante.



Seuls les graphes orientés acycliques admettent un ordre topologique



Bellman-Ford pour graphes orientés acycliques (pseudo code)

Entrée: un Graphe orienté valué $G(S, A, w)$ et un sommet source s de S.

Sortie: Retourne FAUX si G possède un cycle. Sinon, retourne VRAI et retourne la longueur $d(v)$ et le prédécesseur $p(v)$ du plus court chemin allant de s à v pour tout sommet v de S.

POUR tout v dans S FAIRE //initialisation de d et p

$d(v) = +\infty; p(v) = NIL;$

$d(s) = 0;$

(b, T) = TriTopologique(G);

Si (b==FAUX) retourner FAUX; //un cycle a été détecté

POUR $i = 1$ à $|S|$ FAIRE //partie principale de l'algorithme

$u = T[i];$

POUR tout v adjacent à u FAIRE

$temp = d(u) + w(u, v);$

Si $temp < d(v)$ //relâchement pour $(u.v)$

$d(v) = temp; p(v) = u;$

retourner VRAI;

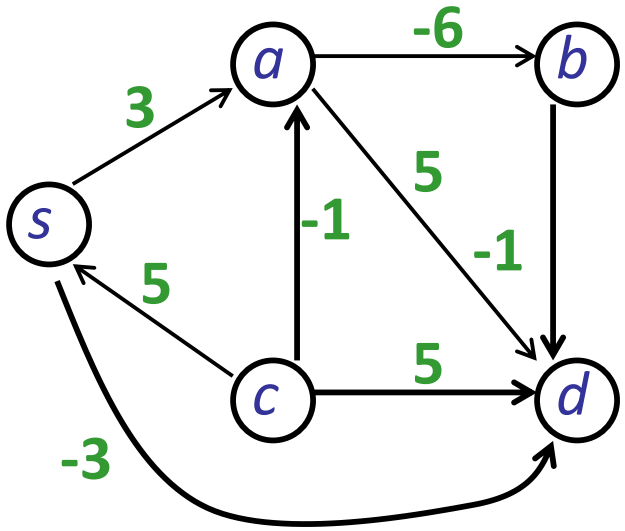
Bellman-Ford pour graphes orientés acycliques: analyse

- Soit $n = |S|$ et $m = |A|$.
- L'initialisation de $p()$ et $d()$ se fait en temps $O(n)$.
- L'exécution de TriTopologique se fait en temps $O(n + m)$.
- La partie principale se fait également en un $temps$ $O(n + m)$ car chaque nœud u de S et chaque nœud v adjacent à u est visité une seule fois.
- L'algorithme au total s'exécute donc en $O(n + m)$.

Bellman-Ford pour graphes orientés acycliques: validité

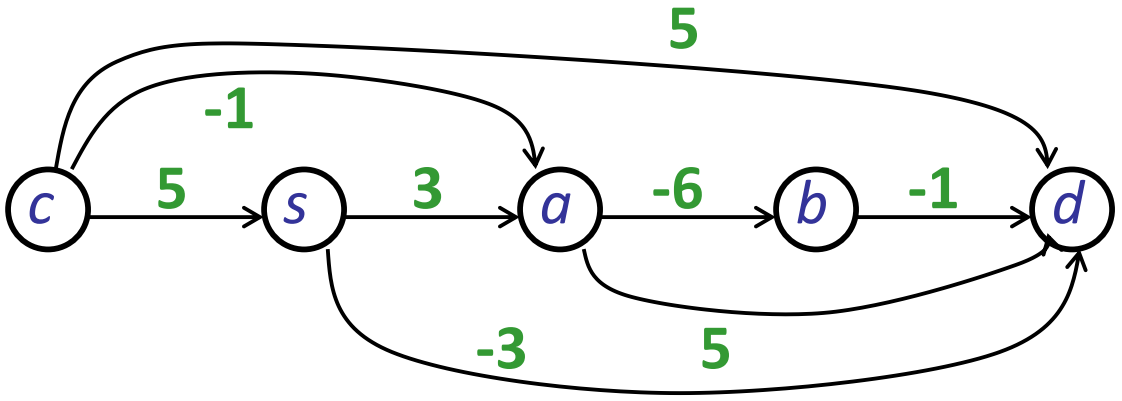
- TriTopologique() nous assure de retourner FAUX lorsque G contient un cycle. L'algorithme est donc valide dans ce cas.
- Lorsque G ne contient pas de cycle, considérez un sommet v accessible depuis s et soit $c =< v_0, v_1, .., v_k >$ un plus court chemin de $s = v_0$ à $v = v_k$.
- Chaque chemin c est conforme au tri topologique: i.e., v_{i-1} précède toujours v_i dans le tri.
- Donc, l'ordre de visite selon le tri topologique nous garantit qu'après avoir visité tous les nœuds, les arcs auront été relâchés exactement dans l'ordre (v_0, v_1), (v_1, v_2) …, (v_{k-1}, v_k) pour tout chemin du graphe.
- D'après la propriété du relâchement des plus courts chemins, on a que $d(v) = D(s, v)$ pour tout plus court chemin allant de s à v.
- Ce qui implique la validité de l'algorithme.

Bellman-Ford pour graphes orientés acycliques: exemple d'exécution



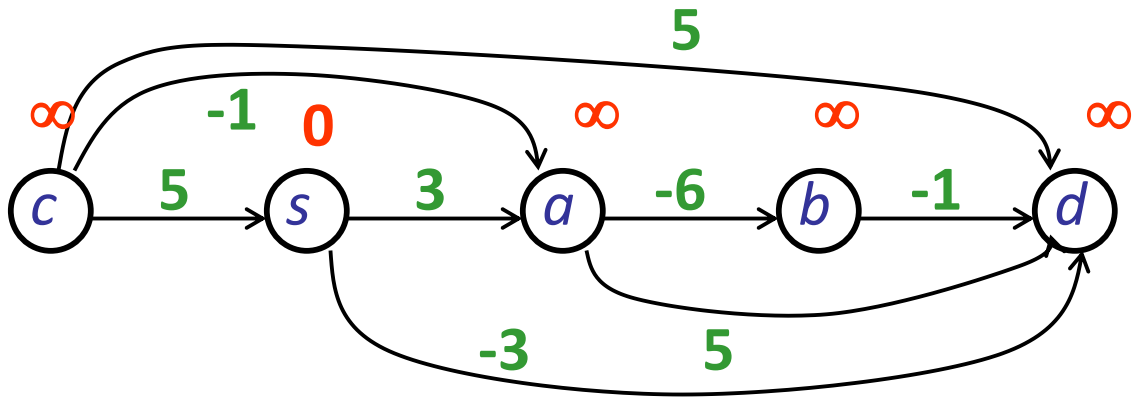
Ordre topologique

c, *s*, *a*, *b*, *d*

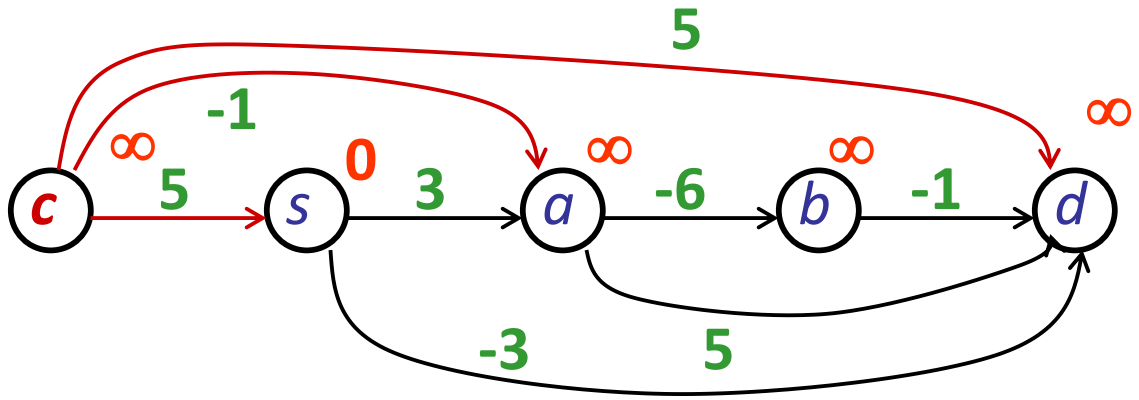


Exemple d'exécution

État initial

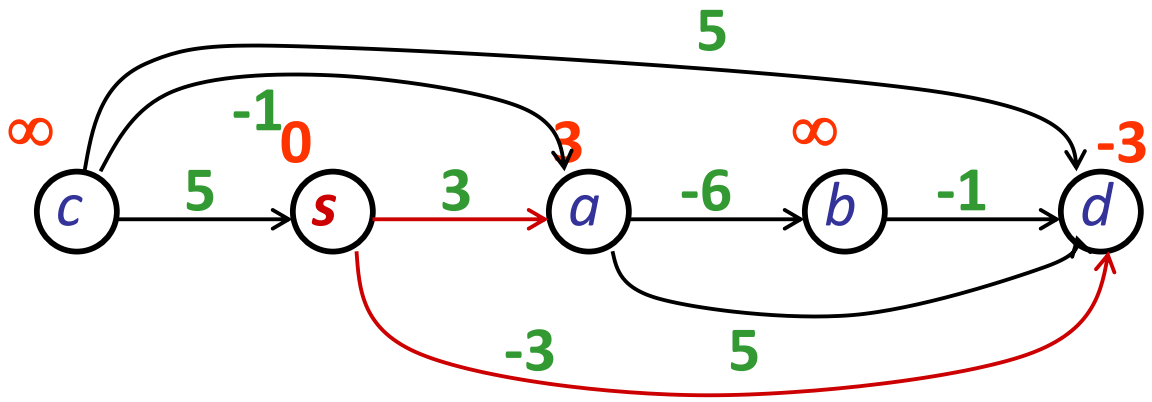


Examen de *c*

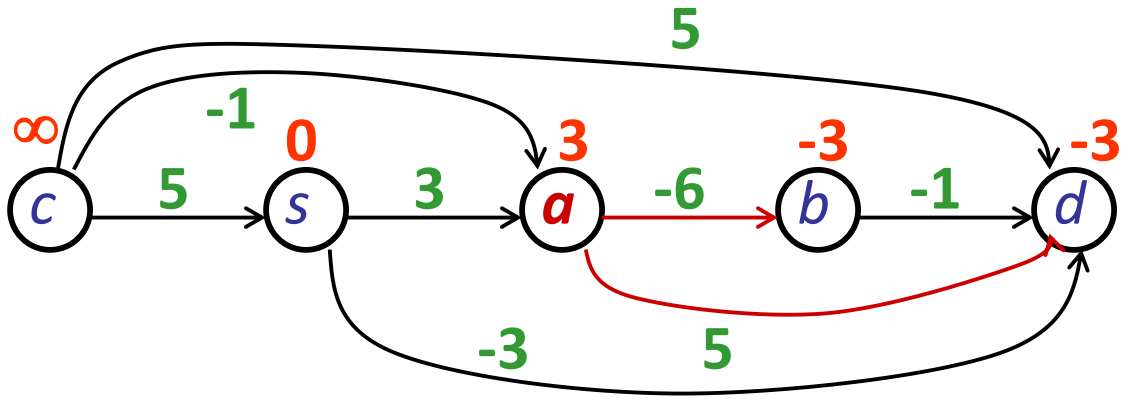


Exemple d'exécution

Examen de *s*

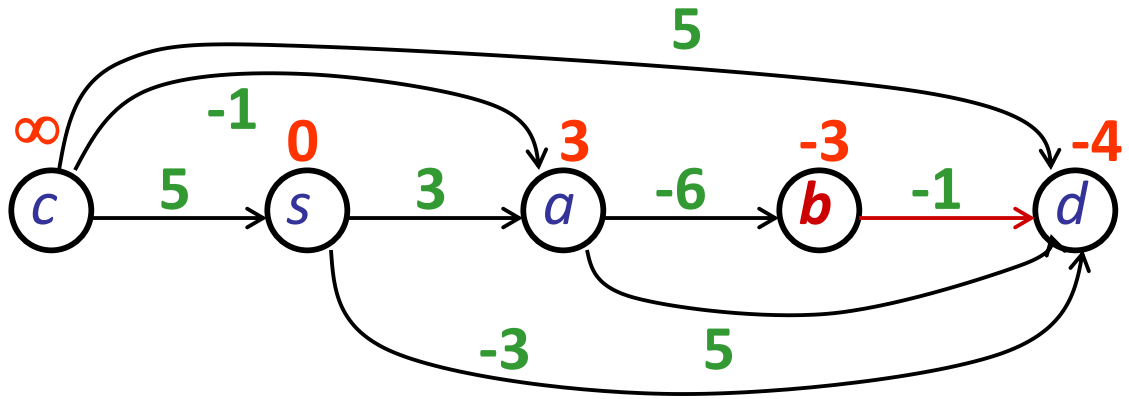


Examen de *a*



Exemple d'exécution

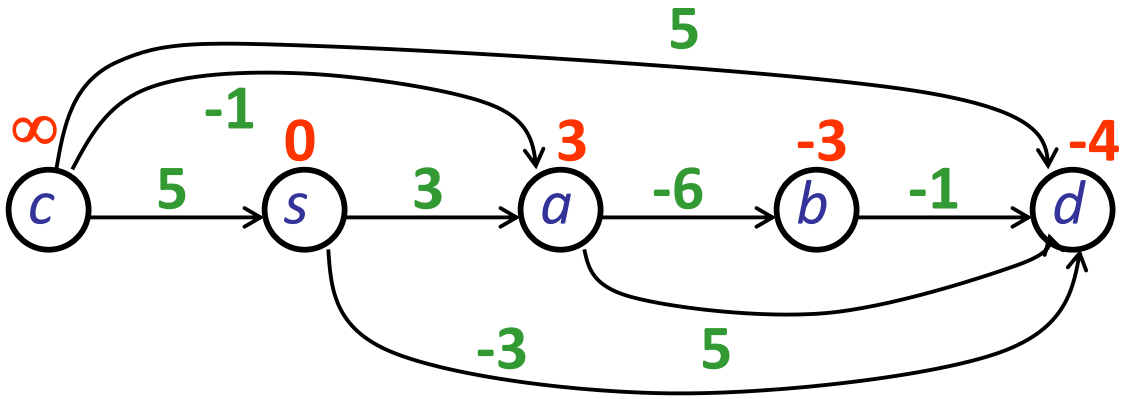
Examen de *b*



Examen de *d* ne produit pas de changement

Exemple d'exécution

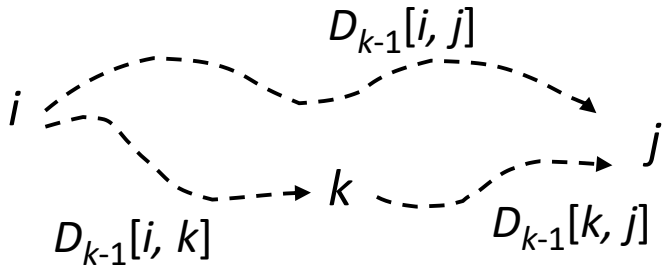
État final:



Algorithme de Floyd

- Cet algorithme (également appelé algorithme Floyd/Warshall) permet de trouver la longueur du plus court chemin entre **toutes les paires de sommets** d'un graphe orienté valué (pondéré).
- Soit $G = (S, A, v)$ un graphe valué, $S = \{1,2, \dots, n\}$, $v : A \to R$(valeur des arcs)
- Nous utiliserons la matrice de valuation W définie par:
  - $W[i,j] \begin{cases} 0 & si\ i = j \\ v(i,j) & si (i,j) \in A \\ \infty & sinon \end{cases}$
- Notons par $D[i,j]$ la distance du plus court chemin allant de i à j.

Algorithme de Floyd



- Soit: $D_k[i,j]$ la longueur du plus court chemin allant du nœud i à j lorsque tous les sommets intermédiaires de ce chemin sont dans $\{1,..,k\}$
- Si k est un sommet intermédiaire => ce chemin est composé d'un chemin allant de i à k dont les nœud intermédiaires sont dans $\{1,..,k-1\}$ et d'un autre chemin allant de k à j et dont les nœuds intermédiaires sont aussi dans $\{1,..,k-1\}$. Dans ce cas on a alors $D_k[i,j] = D_{k-1}[i,k] + D_{k-1}[k,j]$.
- Si k n'est pas un sommet intermédiaire => $D_k[i,j] = D_{k-1}[i,j]$
- Dans tous les cas nous avons alors la récurrence:
  - $D_k[i,j] = Min(D_{k-1}[i,j]; D_{k-1}[i,k] + D_{k-1}[k,j])$
- Notez que la longueur $D_0[i,j]$ du chemin le plus court allant de i à j sans passer par aucun nœud intermédiaire est donné par $D_0[i,j] = W[i,j]$

Algorithme de Floyd

- **Algorithme de Floyd**

Entrée: un graphe $G(S, A, v)$ orienté et pondéré de n noeuds

Sortie: La matrice D des distances des chemins les plus courts (s'il n'y a pas de cycle de poids négatif)

$D = W$; //intialisation: copie de W dans D

for(k = 1, 2, ..., n) //pour tous les nœuds intermédiaires

for(i = 1, 2, ..., n) //pour tous les nœuds source

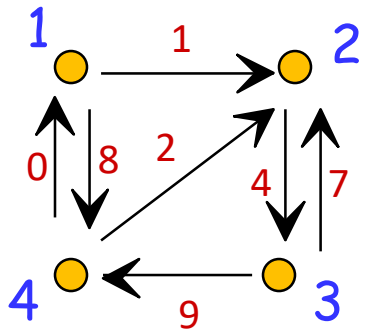
for(j = 1, 2, ..., n) //pour tous les nœuds destination

$D[i,j] = Min(\ D[i,j]\ ;\ D[i,k] + D[k,j]\ )$

return D;

Complexité en $O(n^3)$

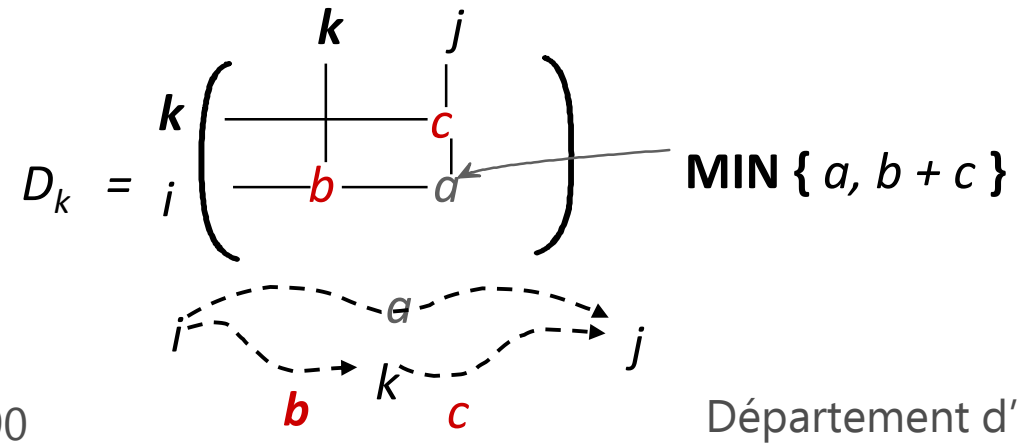
Algorithme de Floyd (exemple)



$$W[i,j]\begin{cases} 0 & si\ i=j \\ v(i,j) & si(i,j)\in A \\ \infty & sinon \end{cases}$$

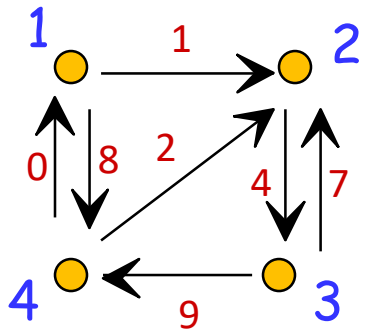
$$D_0 = W = \begin{pmatrix} 0 & \mathbf{1} & \infty & \mathbf{8} \\ \infty & 0 & \mathbf{4} & \infty \\ \infty & \mathbf{7} & 0 & \mathbf{9} \\ \mathbf{0} & \mathbf{2} & \infty & 0 \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 1 & \infty & 8 \\ \infty & 0 & 4 & \infty \\ \infty & 7 & 0 & 9 \\ 0 & \mathbf{1} & \infty & 0 \end{pmatrix}$$



$$D_2 = \begin{pmatrix} 0 & 1 & \mathbf{5} & 8 \\ \infty & 0 & 4 & \infty \\ \infty & 7 & 0 & 9 \\ 0 & 1 & \mathbf{5} & 0 \end{pmatrix}$$

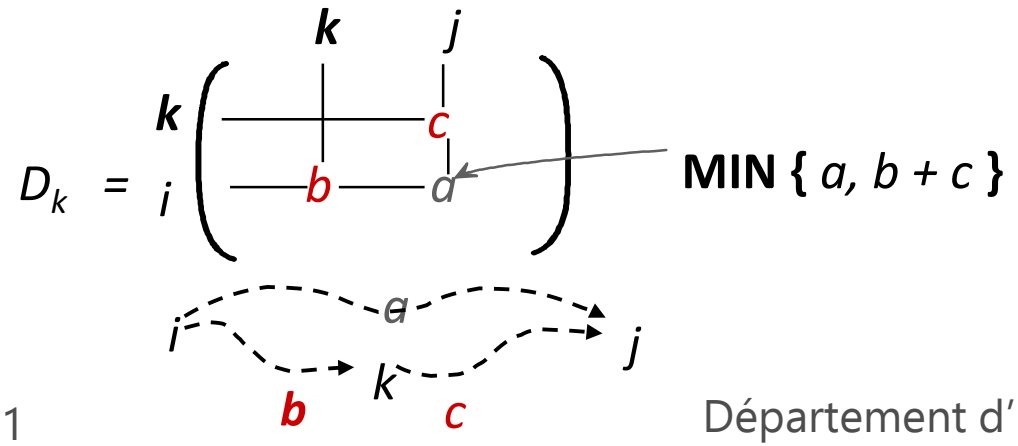
Algorithme de Floyd (exemple)



$$W[i,j]\begin{cases} 0 & si\ i=j \\ v(i,j) & si (i,j)\in A \\ \infty & sinon \end{cases}$$

$$D_3 = \begin{pmatrix} 0 & 1 & 5 & 8 \\ \infty & 0 & 4 & \mathbf{13} \\ \infty & 7 & 0 & 9 \\ 0 & 1 & 5 & 0 \end{pmatrix}$$

$$D_4 = \begin{pmatrix} 0 & 1 & 5 & 8 \\ \mathbf{13} & 0 & 4 & 13 \\ \mathbf{9} & 7 & 0 & 9 \\ 0 & 1 & 5 & 0 \end{pmatrix}$$

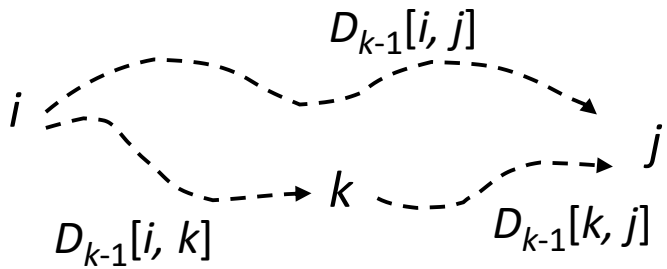


Algorithme de Floyd modifié

- L'algorithme de Floyd nous donne la distance du plus court chemin entre toutes les paires de nœuds possibles d'un graphe.
- Un léger ajout à cet algorithme nous permet d'obtenir **également la séquence de nœuds utilisés par les plus courts chemins**.
- Pour cela, considérons $P_k[i,j]$ = prédécesseur de j sur un plus court chemin de i à j dont les sommets intermédiaires sont tous dans $\{1,..,k\}$.
- On a:
  - $P_0[i,j] = i \; si \; (i,j) \in A$
  - $P_0[i,j]$ = NIL sinon (ie, si $W[i,j] = \infty \; ou \; i = j$)

Algorithme de Floyd modifié

- Maintenant, pour $k > 0$:
  - Si le plus court chemin de i à j (parmi ceux dont les nœuds intermédiaires sont dans $\{1, .., k\}$) passe par k, alors le prédécesseur de j est le même que celui du plus court chemin allant de k à j dont les intermédiaires sont dans $\{1, .., k-1\}$. Dans ce cas $P_k[i,j] = P_{k-1}[k,j]$
  - Si ce plus court chemin ne passe pas par k, alors $P_k[i,j] = P_{k-1}[i,j]$
  - Donc:
    - ✓ $P_k[i,j] = P_{k-1}[k,j]$ si $D_{k-1}[i,k] + D_{k-1}[k,j] < D_{k-1}[i,j]$
    - ✓ $P_k[i,j] = P_{k-1}[i,j]$ sinon



Algorithme de Floyd modifié

- Pour chaque valeur de k (de 1 à n) et pour chaque paire (i, j), l'algorithme de Floyd calcule $D_k[i,j]$et $P_k[i,j]$ de la façon suivante:
  - Si $D_{k-1}[i,k] + D_{k-1}[k,j] < D_{k-1}[i,j]$ alors faire:
    - ✓ $D_k[i,j] = D_{k-1}[i,k] + D_{k-1}[k,j]$;
    - ✓ $P_k[i,j] = P_{k-1}[k,j]$;
    - ✓ Dans ce cas, un nouveau plus court chemin de i à j (et qui passe par k) a été trouvé. Un meilleur prédécesseur de j pour ce nouveau plus court chemin a également été trouvé et c'est $P_{k-1}[k,j]$
  - Si $D_{k-1}[i,k] + D_{k-1}[k,j] \geq D_{k-1}[i,j]$ alors faire:
    - ✓ $D_k[i,j] = D_{k-1}[i,j]$;
    - ✓ $P_k[i,j] = P_{k-1}[i,j]$;
    - ✓ Ce qui revient à ne pas mettre à jour $D_k[i,j]$ et $P_k[i,j]$.

Algorithme de Floyd modifié

- Faire une copie du graphe.
  - ➢ Dans la copie, pour chaque sommet k:
    - ✓ Pour chaque paire de sommets $\{i, j\}$:
      - Vérifier si le coût du chemin va de i à k, additionné à celui qui va de k à j, ne serait pas inférieur à celui précédemment trouvé pour aller de i à j. Si c'est le cas, remplacer le coût de l'arc de i à j par cette somme et prendre en note l'étiquette de k à côté de la case « i, j ».
- Les étiquettes k notées dans chaque case correspondent alors au « meilleur précédent » pour chaque paire de noeud.
- Pour obtenir le plus court chemin d'un nœud à un autre, il suffit alors de partir du dernier nœud, et de remonter jusqu'au début, de case en case, via ces étiquettes « k » notées à côté de chaque case, puis d'inverser la liste de nœuds parcourus par ce processus.

Algorithme de Floyd modifié

Entrée: un graphe $G(S, A, v)$ orienté et pondéré de n noeuds

Sortie: Les matrices D et P pour les chemins les plus courts

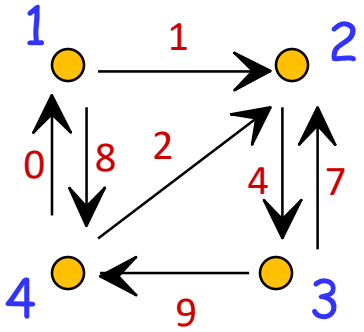
for(i = 1, 2, ..., n) //initialisation de D et P
    for(j = 1, 2, ..., n)
        $D[i,j] = W[i,j];$
        if ($i != j$ et $W[i,j] != \infty$) $P[i,j] = i;$
        else $P[i,j] = NIL;$

for($k = 1, 2, \ldots, n$) //pour tous les nœuds intermédiaires
    for($i = 1, 2, \ldots, n$) //pour tous les nœuds source
        for($j = 1, 2, \ldots, n$) //pour tous les nœuds destination
            $temp = D[i,k] + D[k,j];$
            if($temp < D[i,j]$)
                $D[i,j] = temp;$
                $P[i,j] = P[k,j];$

return (D,P);

Complexité en $O(n^3)$

Algorithme de Floyd modifié (exemple)



Matrice des prédécesseurs

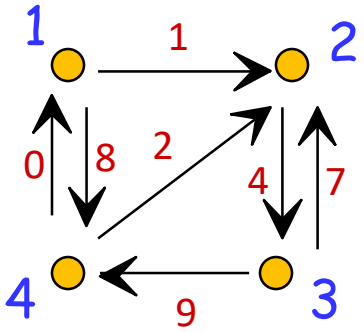
$\boldsymbol{P_k}\,[\boldsymbol{i},\boldsymbol{j}]$ = prédécesseur de j sur un plus court chemin de i à j dont les sommets intermédiaires sont tous $\leq k$

$$D_0 = W = \begin{pmatrix} 0 & \mathbf{1} & \infty & \mathbf{8} \\ \infty & 0 & \mathbf{4} & \infty \\ \infty & \mathbf{7} & 0 & \mathbf{9} \\ \mathbf{0} & \mathbf{2} & \infty & 0 \end{pmatrix} \qquad P_0 = \begin{pmatrix} - & \mathbf{1} & - & \mathbf{1} \\ - & - & \mathbf{2} & - \\ - & \mathbf{3} & - & \mathbf{3} \\ \mathbf{4} & \mathbf{4} & - & - \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 1 & \infty & 8 \\ \infty & 0 & 4 & \infty \\ \infty & 7 & 0 & 9 \\ 0 & \mathbf{1} & \infty & 0 \end{pmatrix} \qquad P_1 = \begin{pmatrix} - & 1 & - & 1 \\ - & - & 2 & - \\ - & 3 & - & 3 \\ 4 & \mathbf{1} & - & - \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & 1 & \mathbf{5} & 8 \\ \infty & 0 & 4 & \infty \\ \infty & 7 & 0 & 9 \\ 0 & 1 & \mathbf{5} & 0 \end{pmatrix} \qquad P_2 = \begin{pmatrix} - & 1 & \mathbf{2} & 1 \\ - & - & 2 & - \\ - & 3 & - & 3 \\ 4 & 1 & \mathbf{2} & - \end{pmatrix}$$

Algorithme de Floyd modifié (exemple)



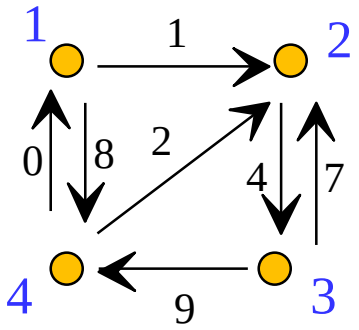
Matrice des prédécesseurs

$\boldsymbol{P_k}\,[\boldsymbol{i},\boldsymbol{j}]$ = prédécesseur de j sur un plus court chemin de i à j dont les sommets intermédiaires sont tous $\leq k$

$$D_3 = \begin{pmatrix} 0 & 1 & 5 & 8 \\ \infty & 0 & 4 & \mathbf{13} \\ \infty & 7 & 0 & 9 \\ 0 & 1 & 5 & 0 \end{pmatrix} \qquad P_3 = \begin{pmatrix} - & 1 & 2 & 1 \\ - & - & 2 & \mathbf{3} \\ - & 3 & - & 3 \\ 4 & 1 & 2 & - \end{pmatrix}$$

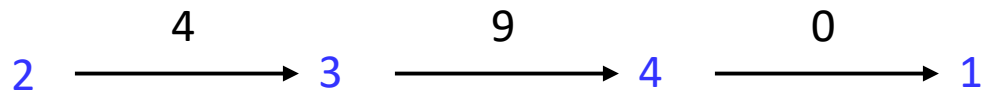
$$D_4 = \begin{pmatrix} 0 & 1 & 5 & 8 \\ \mathbf{13} & 0 & 4 & 13 \\ \mathbf{9} & 7 & 0 & 9 \\ 0 & 1 & 5 & 0 \end{pmatrix} \qquad P_4 = \begin{pmatrix} - & 1 & 2 & 1 \\ \mathbf{4} & - & 2 & 3 \\ \mathbf{4} & 3 & - & 3 \\ 4 & 1 & 2 & - \end{pmatrix}$$

Reconstruction du chemin le plus court



$$D = D_4 = \begin{pmatrix} 0 & 1 & 5 & 8 \\ \mathbf{13} & 0 & 4 & 13 \\ 9 & 7 & 0 & 9 \\ 0 & 1 & 5 & 0 \end{pmatrix} \qquad P = P_4 = \begin{pmatrix} - & 1 & 2 & 1 \\ \mathbf{4} & - & \mathbf{2} & \mathbf{3} \\ 4 & 3 & - & 3 \\ 4 & 1 & 2 & - \end{pmatrix}$$

- Lorsque l'on a les matrices D et P, pour obtenir le plus court chemin de i à j, il suffit de partir de j et d'obtenir son prédécesseur $P[i,j] = p$. Ensuite on trouve le prédécesseur de $P[i,p]$ jusqu'à ce que l'on arrive à i.
- Exemple de reconstruction de plus court chemin de 2 à 1:
  - distance du plus court chemin de 2 à 1 = D[2,1] = 13
    - ✓ P[2,1] = 4 ; P[2,4] = 3 ; P[2,3] = 2 ;



La programmation dynamique

- L'algorithme de Floyd/Warshall est un bel exemple de solution algorithmique obtenue par **programmation dynamique**.
- La programmation dynamique consiste à solutionner un problème en solutionnant séquentiellement des sous-instances de ce problème.
- Chaque sous-instance englobe la sous instance précédente. La solution de chaque sous-instance est obtenue à partir de la sous-instance précédente.
- Il y a donc croissance de la sous-instance jusqu'à ce que l'on obtienne l'instance à traiter.
- Pour l'algorithme de Floyd: on trouve la longueur du chemin le plus court entre i et j en trouvant le plus court chemin dont les nœuds intermédiaires sont dans $\{1, .., k\}$ pour k allant de 1 jusqu'à n. Ce qui permet de construire la solution à l'étape k à partir de celle de l'étape k-1 est la récurrence:
  - $D_k[i,j] = Min(D_{k-1}[i,j]; D_{k-1}[i,k] + D_{k-1}[k,j])$
  - La solution finale est obtenue pour $k = n$, ie, $D[i,j] = D_n[i,j]$.
- Vous verrez d'autres exemples de solution par programmation dynamique dans votre cours de «conception et d'analyse d'algorithmes».

Synthèse

- Tri topologique
  - en $O(n+m)$
- Bellman-Ford et les graphes orientés acycliques
  - en $O(n+m)$
- longueur du plus court chemin entre toutes les paires de sommets d'un graphe orienté valué (pondéré)

  Floyd/Warshall
  - $O(n^3)$
  - Modifié : prédécesseurs
  - Reconstruction du plus court chemin
  - exemple de solution algorithmique obtenue par programmation dynamique