

# Composantes fortement connexes ; plus court chemin

CM n°4 — Algorithmique (AL5)

Matěj Stehlík

13/10/2022

## Une conséquence du tri topologique



- Le premier sommet dans un tri topologique de  $G$  est une source (c'est-à-dire, aucun arc n'entre le sommet).
- De même, le dernier sommet est un puits (c'est-à-dire, aucun arc ne sort du sommet).

### **Théorème**

Tout graphe orienté acyclique contient au moins une source et au moins un puits.

Ce théorème est la base d'une autre approche au tri topologique :

- Trouver un sommet source de  $G$  et supprimer-le de  $G$ .
- Répéter jusqu'à ce que le graphe devienne vide.

# Implémentation naïve de l'algorithme

**Entrées** : graphe orienté acyclique  $G = (V, E)$

$L \leftarrow \emptyset$

**tant que**  $V \neq \emptyset$  **faire**

    trouver un sommet  $v$  t.q.  $d^-(v) = 0$   
     $G \leftarrow G - v$   
     $L \leftarrow L + v$

**return**  $L$

## Quelques idées pour améliorer l'algorithme...

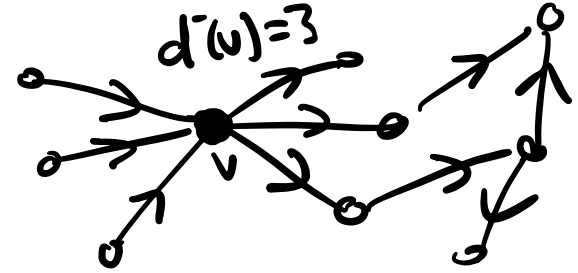
Voisins "entrants" de  $v$  :  $N^-(v)$

" " "sortants" " " :  $N^+(v)$

Le degré "entrant" de  $v$  :  $d^-(v)$

" " "sortant" " " :  $d^+(v)$

(source)



- Garder une file avec les sommets de degré entrant 0 pour ne pas avoir à rechercher ces sommets plusieurs fois.
- Le degré entrant d'un sommet ne change que lorsque l'un de ses voisins entrants (correspondant à des conditions préalables) ne soit supprimé.
- Garder une liste des degrés entrants des sommets pour ne pas avoir à modifier le graphe.

# Algorithme de Kahn

**Entrées :** graphe orienté acyclique  $G = (V, E)$

$L \leftarrow \emptyset$

**pour tous les**  $u \in V$  **faire**

└  $d^-(u) \leftarrow$  degré entrant de  $u$

créer file( $Q$ )

**pour tous les**  $u \in V$  **faire**

└ **si**  $d^-(u) = 0$  **alors**

└ └ enfiler( $Q, u$ )

**tant que**  $Q \neq \emptyset$  **faire**

└  $v \leftarrow$  défiler( $Q$ )

└  $L \leftarrow L + v$

└ réduire le degré de tous les voisins sortants de  $v$  de 1

}  $O(m)$

└ **pour tous les**  $u \in V$  **faire**

└ └ **si**  $d^-(u) = 0$  **alors**

└ └ └ enfiler( $Q, u$ )

**return**  $L$

## Complexité de l'algorithme de Kahn

- L'initialisation prend temps  $O(n)$ ,
- La boucle **tant que** est parcourue  $O(n)$  fois.
- Réduire le degré de tous les voisins sortants de  $v$  est de complexité  $O(m)$ .
- On obtient donc une complexité de  $O(mn)$ .

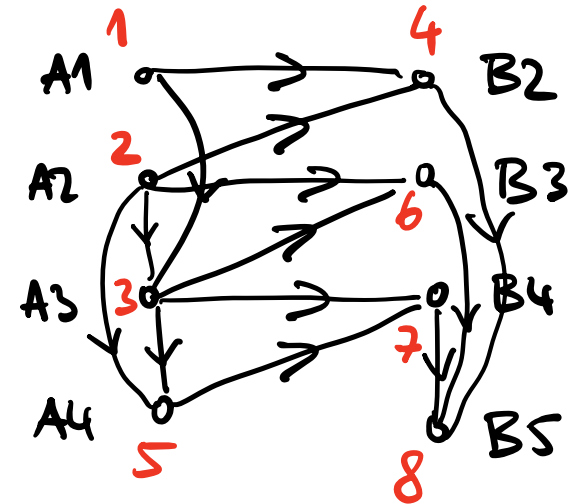
## Complexité de l'algorithme de Kahn

- L'initialisation prend temps  $O(n)$ ,
- La boucle **tant que** est parcourue  $O(n)$  fois.
- Réduire le degré de tous les voisins sortants de  $v$  est de complexité  $O(m)$ .
- On obtient donc une complexité de  $O(mn)$ .
- Or, si on est attentif, on remarque que chaque arc n'est traité qu'une seule fois, donc finalement la boucle while prend temps  $O(n + m)$ .
- On conclut que l'algorithme de Kahn est de complexité  $O(n + m)$ , donc la même que si l'on utilise DFS.

# Application du tri topologique aux tableurs

- Cellules dont les formules font référence à d'autres cellules ont des dépendances.
- On peut utiliser le tri topologique pour mettre à jour efficacement les cellules!

	A	B
1	1	
2	1	=A2 / A1
3	=A1+A2	=A3 / A2
4	=A2+A3	=A4 / A3
5		=AVERAGE (B2 : B4)





## Application du tri topologique aux tableurs

- Cellules dont les formules font référence à d'autres cellules ont des dépendances.
- On peut utiliser le tri topologique pour mettre à jour efficacement les cellules!

	A	B
1	1	
2	1	=A2 / A1
3	2	=A3 / A2
4	=A2 + A3	=A4 / A3
5		=AVERAGE (B2 : B4)

# Application du tri topologique aux tableurs

- Cellules dont les formules font référence à d'autres cellules ont des dépendances.
- On peut utiliser le tri topologique pour mettre à jour efficacement les cellules!

	A	B
1	1	
2	1	1
3	2	=A3/A2
4	=A2+A3	=A4/A3
5		=AVERAGE (B2 : B4)

## Application du tri topologique aux tableaux

- Cellules dont les formules font référence à d'autres cellules ont des dépendances.
- On peut utiliser le tri topologique pour mettre à jour efficacement les cellules!

	A	B
1	1	
2	1	1
3	2	=A3/A2
4	3	=A4/A3
5		=AVERAGE (B2 : B4)

# Application du tri topologique aux tableurs

- Cellules dont les formules font référence à d'autres cellules ont des dépendances.
- On peut utiliser le tri topologique pour mettre à jour efficacement les cellules!

	A	B
1	1	
2	1	1
3	2	2
4	3	=A4/A3
5		=AVERAGE (B2 : B4)

## Application du tri topologique aux tableurs

- Cellules dont les formules font référence à d'autres cellules ont des dépendances.
- On peut utiliser le tri topologique pour mettre à jour efficacement les cellules!

	A	B
1	1	
2	1	1
3	2	2
4	3	1.5
5		=AVERAGE (B2 : B4)

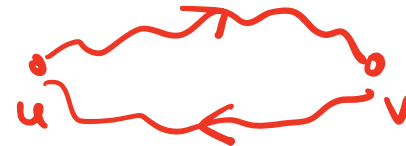
## Application du tri topologique aux tableurs

- Cellules dont les formules font référence à d'autres cellules ont des dépendances.
- On peut utiliser le tri topologique pour mettre à jour efficacement les cellules!

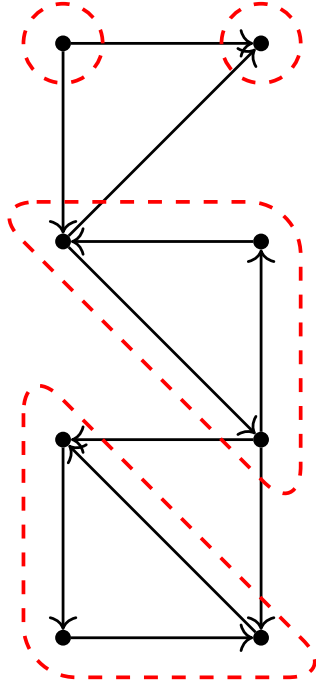
	A	B
1	1	
2	1	1
3	2	2
4	3	1.5
5		1.5

# Connexité dans les graphes orientés

- Nous avons déjà vu la définition de graphes connexes et des composantes connexes.
- Intuitivement, un graphe est connexe s'il ne peut pas être “séparé” sans casser des arêtes.
- Pour les graphes orientés, la notion de connexité est un peu plus subtile.
- Soit  $\sim$  la relation suivante :  $u \sim v$  ssi il existe un chemin de  $u$  à  $v$  et aussi un chemin de  $v$  à  $u$ .



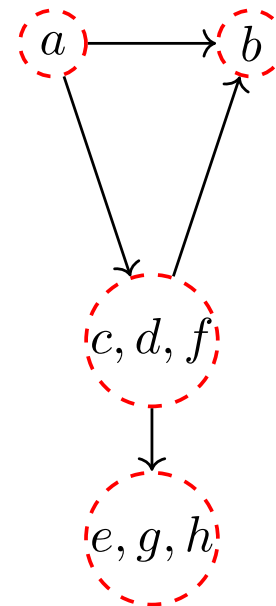
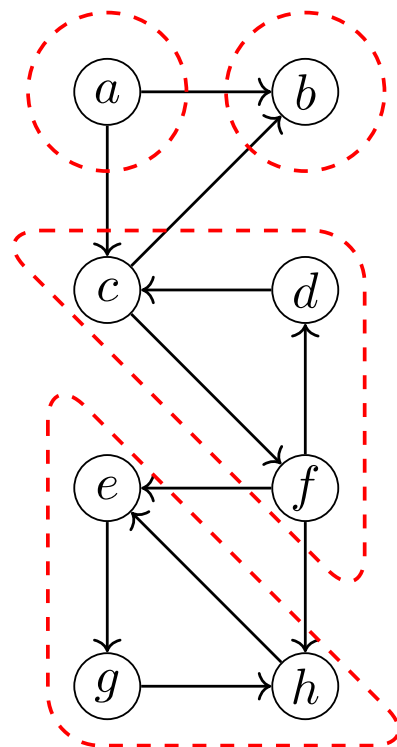
# Composantes fortement connexes



- Il est facile de vérifier que  $\sim$  est une relation d'équivalence :
  - $\sim$  est symétrique
  - $\sim$  est réflexive
  - $\sim$  est transitive.
- Les classes d'équivalence forment une partition de  $V(G)$ .
- On les appelle les *composantes fortement connexes*.
- D'une façon informelle, une composante connexe consiste de tous les sommets t.q. pour chaque paire de sommets  $u, v$ , il existe un chemin de  $u$  à  $v$ .



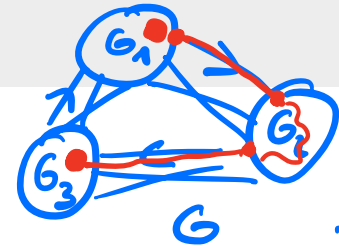
## Contracter les composantes fortement connexes



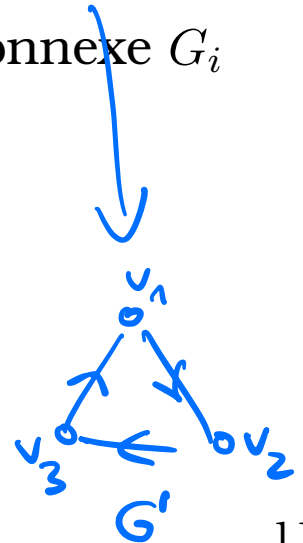
# Le graphe contracté est un DAG

## Théorème

Soit  $G$  un graphe orienté quelconque, et soit  $G'$  le graphe obtenu en contractant chaque composante fortement connexe à un seul sommet. Alors,  $G'$  est un graph orienté acyclique (un DAG).



- Soit  $C = (G_1, G_2, \dots, G_k)$  un circuit dans  $G'$ .
- Alors, tous les sommets de  $G$  dans la composante fortement connexe  $G_i$  sont atteignable depuis tous les sommets de  $G_j$ , pour tous  $i, j \in \{1, \dots, k\}$ .
- Donc,  $V(G_1) \cup V(G_2) \cup \dots \cup V(G_k)$  appartiennent à une seule composante connexe.
- Donc  $k = 1$ .

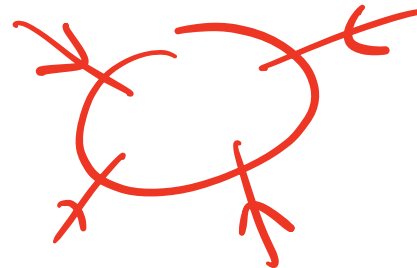


# Composantes fortement connexes et parcours en profondeur

## Propriété 1

Si la procédure `explorer` est lancée à partir du sommet  $u$ , elle se terminera précisément lorsque tous les sommets atteignables depuis  $u$  auront été visités.

- Donc, si  $u$  est un sommet dans une composante fortement connexe  $G_i$  qui est un puits dans le graphe contracté  $G'$  (tous les arcs incidents à  $G_i$  pointent vers  $G_i$ ), alors `explore( $u$ )` va parcourir précisément les sommets de  $G_i$ .



composante f.-c.  
de type puits

# Composantes fortement connexes et parcours en profondeur

## Propriété 2

Soient  $G_i$  et  $G_j$  des composantes fortement connexes de  $G$ . S'il existe un arc d'un sommet de  $G_i$  à un sommet de  $G_j$ , alors

$$\max\{\text{post}(v) : v \in G_i\} \geq \max\{\text{post}(v) : v \in G_j\}.$$

- Il y a deux cas à considérer.
- Si le DFS visite la composante  $G_i$  avant la composante  $G_j$ , alors tous les sommets de  $G_i$  et  $G_j$  seront visités avant que `explorer` coince.
- Par conséquent, le nombre `post` du premier sommet visité dans  $G_i$  sera supérieur à celui de tout sommet de  $G_j$ .
- Si  $G_j$  est visité en premier, `explorer` va coincer après avoir visité l'ensemble de  $G_j$  mais avant d'avoir visité l'ensemble de  $G_i$ .

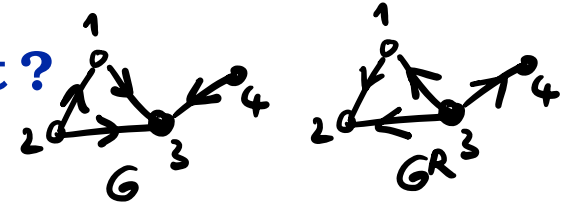
## Conséquence de la Propriété 2

### Propriété 3

Le sommet avec la valeur maximum de  $\text{post}(\cdot)$  dans une recherche en profondeur appartient à une composante fortement connexe de type source.

- On peut trier les composantes fortement connexes par ordre décroissant de leurs nombres post maximaux.

... et si on veut trouver un sommets dans un puit ?



- La Propriété 2 nous aide à trouver un sommet dans une composante fortement connexe de  $G$  de type « source ».
- Or, nous avons besoin d'un sommet dans une composante fortement connexe de  $G$  de type « puits ».
- Soit  $G^R$  le graphe orienté *inverse*, défini comme suit :  $G^R = (V, E^R)$ , où  $(u, v) \in E^R$  ssi  $(v, u) \in E$ . C'est-à-dire, on reverse la direction des arcs.
- $G^R$  a les mêmes composantes fortement connexes que  $G$ .
- En effectuant un parcours en profondeur sur  $G^R$ , le sommet avec la valeur maximale de  $\text{post}(\cdot)$  appartient à une composante fortement connexe de  $G^R$  de type « source », c'est-à-dire, à une composante fortement connexe de  $G$  de type « puits ».


## Vers un algorithme de composantes fortement connexes

- Comment continuer après l'identification de la première composante fortement connexe de type puits ?
- Il suffit d'utiliser la Propriété 2.
- Une fois que nous avons trouvé la première composante fortement connexe  $G_1$  et que nous l'avons supprimée du graphe, le sommet avec le nombre post maximum dans  $G - V(G_1)$  appartiendra à une composante fortement connexe de  $G - V(G_1)$ .
- Par conséquent, nous pouvons continuer à utiliser les nombres  $\text{post}(\cdot)$  du parcours en profondeur initial sur  $G^R$  pour produire successivement la deuxième composante fortement connexe, la troisième composante fortement connexe, etc.

# Un algorithme de composantes fortement connexes

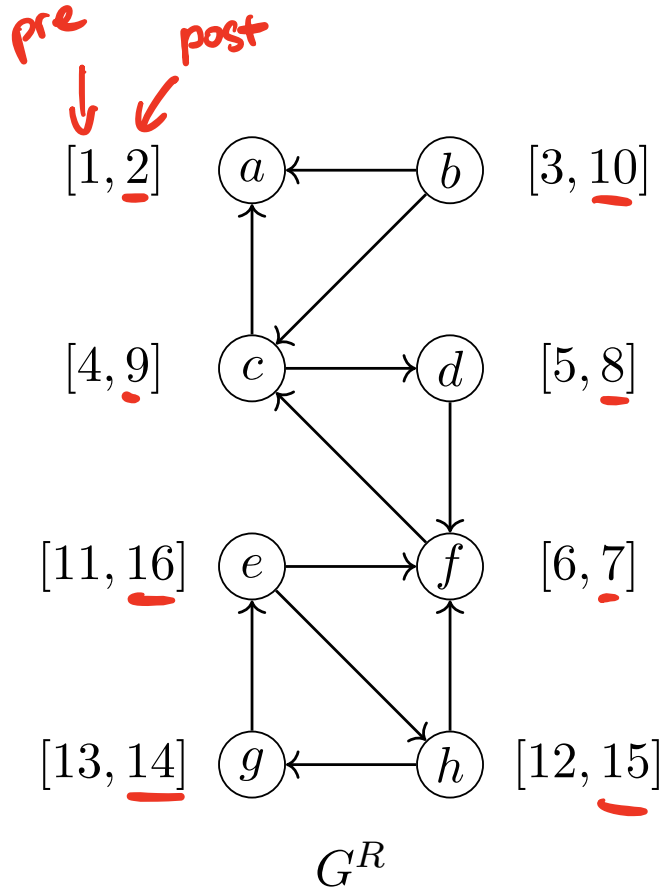
Kosaraju (?)

Voici donc un algorithme pour déterminer les composantes fortement connexes de  $G$  :

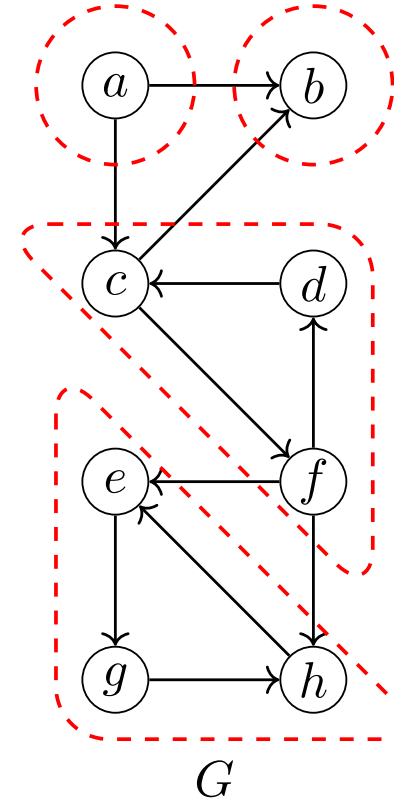
1. Exécuter un parcours en profondeur sur  $G^R$  et garder pour chaque sommet son valeur post.
  2. Trier les sommets selon leurs valeurs post.
  3. Exécuter un parcours en profondeur sur  $G$  selon l'ordre inverse.  
(En particulier, chaque fois que la procédure DFS appelle la procédure explore, commencer par la premier sommet non visité dans l'ordre inverse.)
- 



# Example



$e, h, g, b, c, d, f, a$



$\{e, h, g\}, \{b\}, \{c, d, f\}, \{a\}$

# Applications du parcours en profondeur : résumé

Nous avons vu 3 applications du parcours en profondeur :

1. Détection de circuits dans les graphes orientés :

- Un DFS révèle un arc retour ssi le graphe contient un circuit.

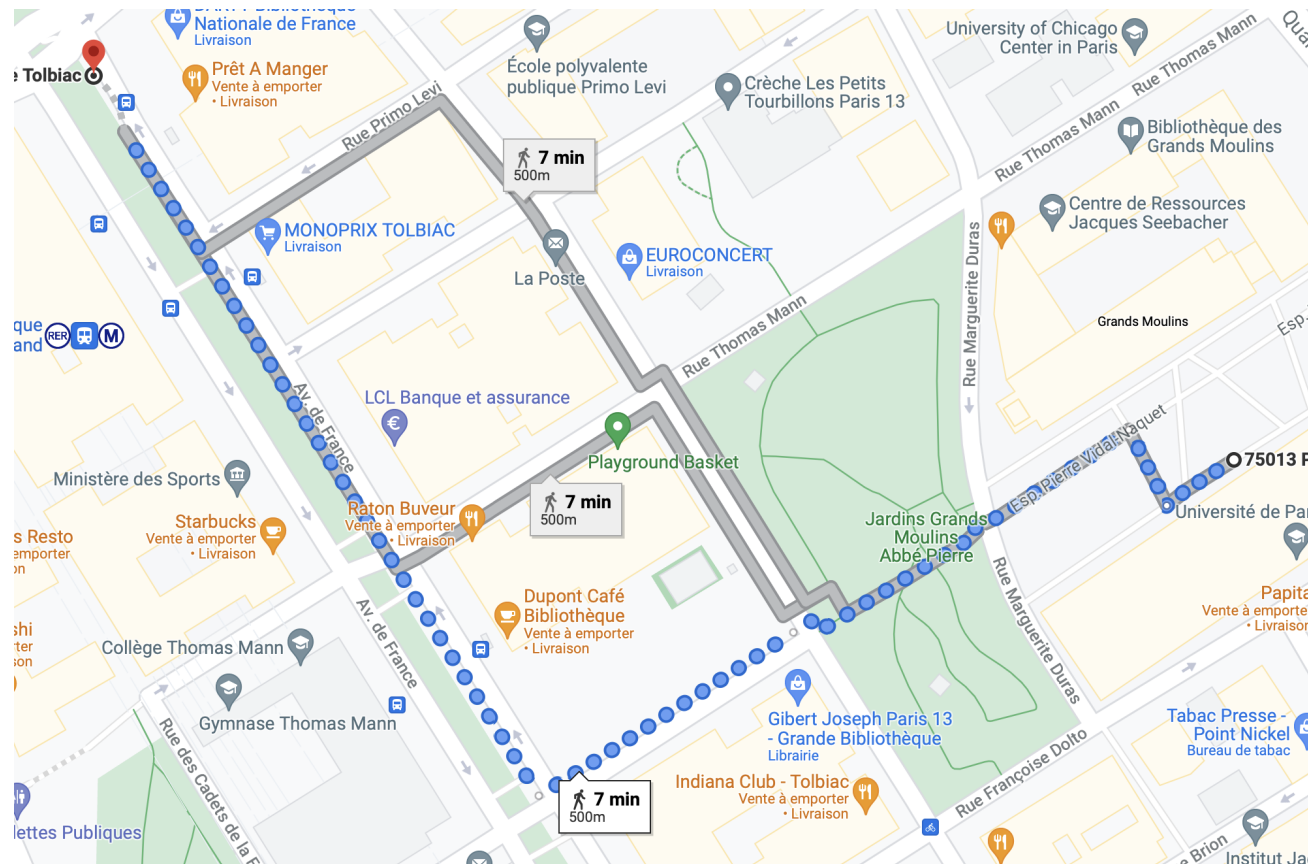
2. Tri topologique des graphes acycliques orientés :

- Il suffit de faire un DFS qui garde trace des temps de <sup>post-</sup>visite, et ensuite trier les sommets par nombre post décroissant.

3. Calcul des composantes <sup>fortement</sup> connexes d'un graphe orienté :

- On fait DFS sur le graphe inverse  $G^R$  et on trie les sommets selon l'ordre décroissant de nombre post
- Ensuite on fait DFS sur  $G$  en utilisant l'ordre trouvé dans la première étape.

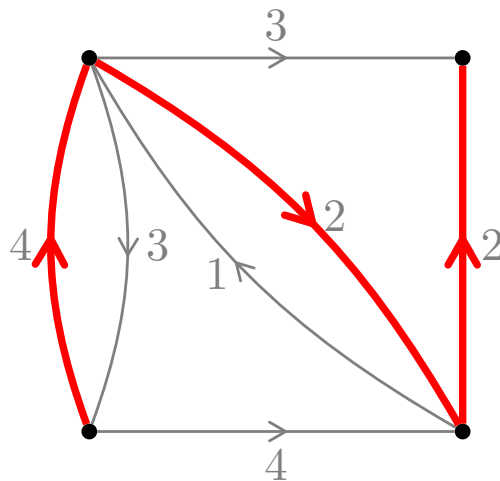
# Plus court chemin



# Chemins et circuits pondérés

## Définition

- Soit  $G = (V, E)$  un graphe orienté pondéré (avec pondération  $w \in \mathbb{R}^{|E|}$ ).
- Soit  $P \subseteq$  un chemin dans  $G$ .
- La *longueur* (ou poids) du chemin  $P$  est définie comme  $\sum_{e \in E(P)} w_e$ .

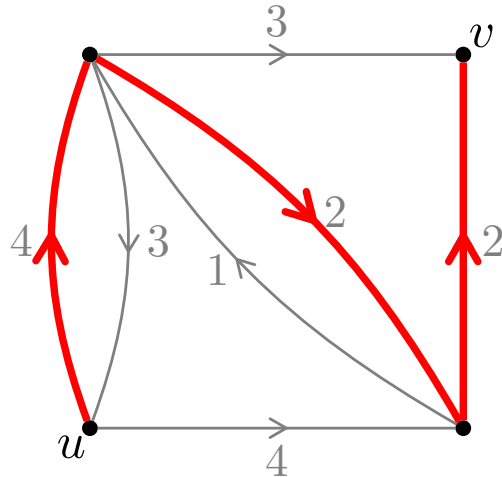


# Distance

## Définition

Soient  $u, v$  deux sommets dans un graphe orienté pondéré  $G = (V, E)$  (avec pondération  $w \in \mathbb{R}^{|E|}$ ). La *distance* de  $u$  à  $v$  est définie comme

$$\text{dist}(u, v) = \min\{w(P) : P \text{ est un chemin de } u \text{ à } v\}$$



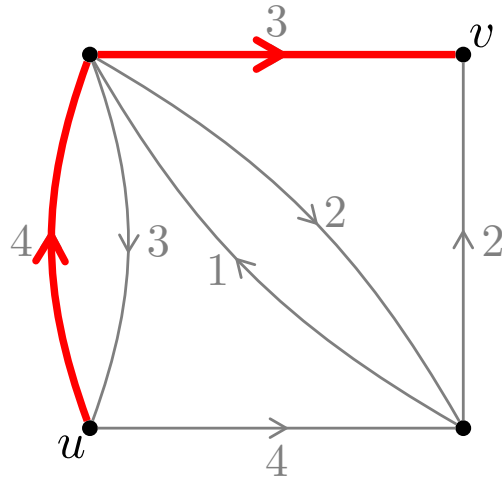
$$\text{dist}(u, v) \leq 8$$

# Distance

## Définition

Soient  $u, v$  deux sommets dans un graphe orienté pondéré  $G = (V, E)$  (avec pondération  $w \in \mathbb{R}^{|E|}$ ). La *distance* de  $u$  à  $v$  est définie comme

$$\text{dist}(u, v) = \min\{w(P) : P \text{ est un chemin de } u \text{ à } v\}$$



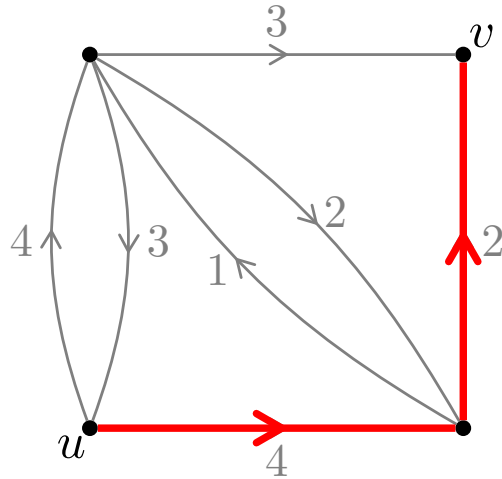
$$\text{dist}(u, v) \leq 7$$

# Distance

## Définition

Soient  $u, v$  deux sommets dans un graphe orienté pondéré  $G = (V, E)$  (avec pondération  $w \in \mathbb{R}^{|E|}$ ). La *distance* de  $u$  à  $v$  est définie comme

$$\text{dist}(u, v) = \min\{w(P) : P \text{ est un chemin de } u \text{ à } v\}$$



$$\text{dist}(u, v) = 6$$

# Le problème du plus court chemin

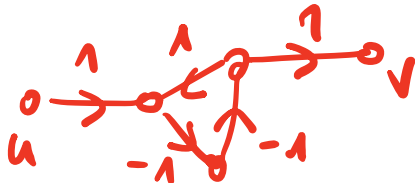
## Problème

Étant donné un graphe orienté  $G = (V, E)$  pondéré (avec pondération  $w \in \mathbb{R}^{|E|}$ ) et deux sommets  $u \neq v$  dans  $V$ , trouver un plus court chemin (« chaîne orientée ») de  $u$  vers  $v$ .

## Remarque

Il peut ne pas exister de plus court chemin de  $u$  à  $v$  :

- S'il n'y a aucun chemin de  $u$  à  $v$  :  $\text{dist}(u, v) = \infty$
- S'il y a un circuit négatif sur le chemin de  $u$  à  $v$  :  $\text{dist}(u, v) = -\infty$

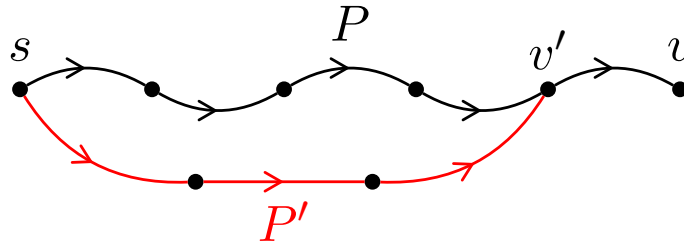




# Principe de sous-optimalité

## Observation

Si  $P$  est un plus court chemin de  $s$  vers  $v$  alors, en notant  $v'$  le prédécesseur de  $v$  dans ce chemin, le sous-chemin de  $P$  qui va de  $s$  vers  $v'$  est un plus court chemin de  $s$  vers  $v'$ .



## Démonstration par l'absurde

S'il existe  $P'$  de  $s$  vers  $v'$  de poids strictement inférieur au sous-chemin de  $P$  de  $s$  vers  $v'$  alors en concaténant  $P'$  à  $(v, v')$  on aurait un chemin de  $s$  à  $v'$  de poids strictement inférieur à celui de  $P$ , contradiction.

# Algorithme de Dijkstra

nombre réels  
non-négatifs

**Entrées** : Graphe orienté  $G = (V, E)$  avec pondération  $\ell \in \mathbb{R}^+$ , un sommet  $s \in V$

**Sorties** : Distances de  $s$  aux autres sommets

$S \leftarrow \emptyset$

$D[s] \leftarrow 0$

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

$D[u] \leftarrow +\infty$

**tant que**  $S \neq V$  **faire**

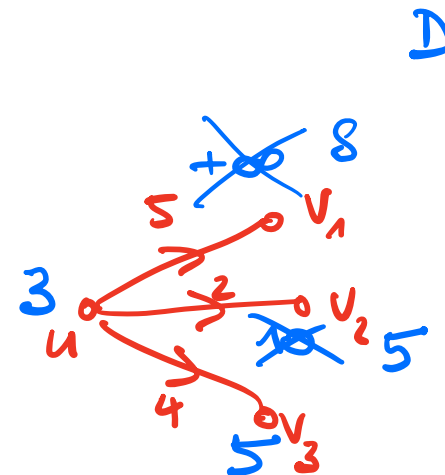
    Trouver  $u \in V \setminus S$  tel que  $D[u]$  est minimum

$S \leftarrow S \cup \{u\}$

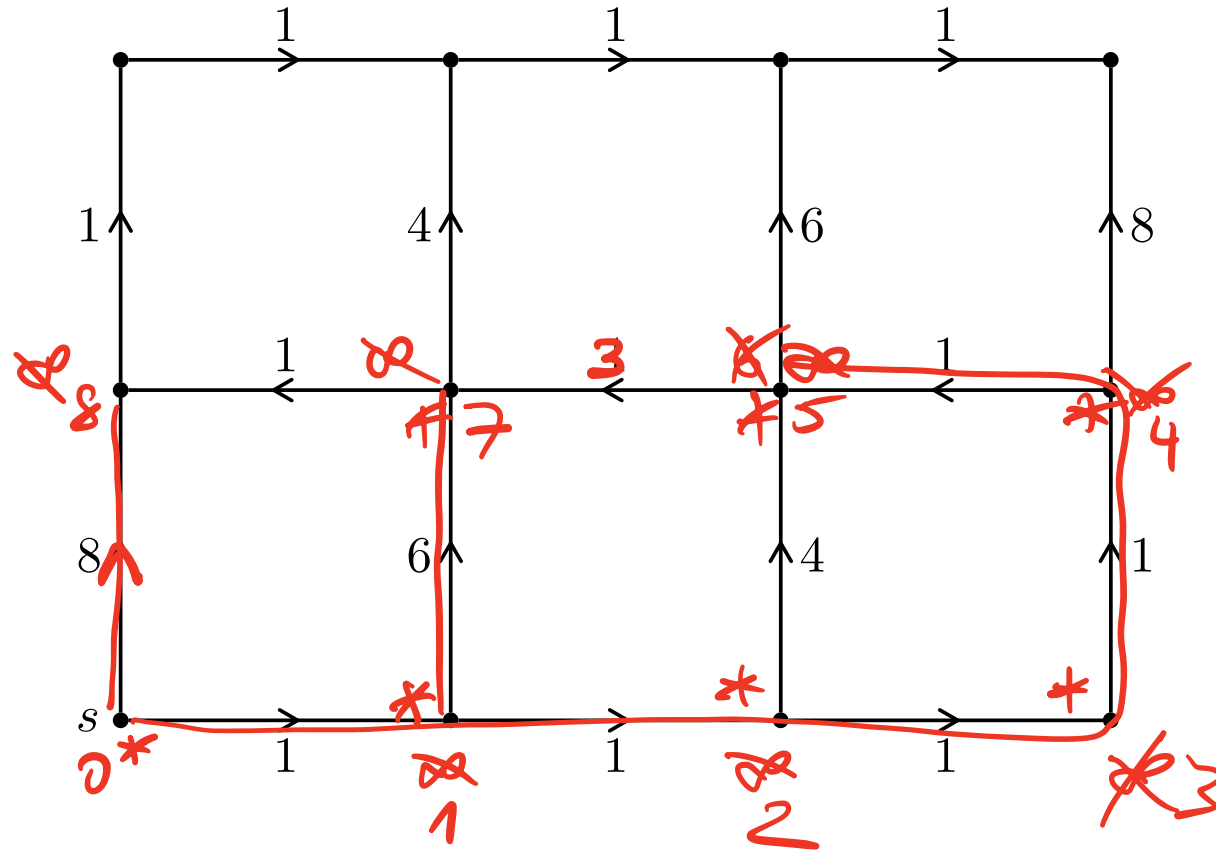
**pour tous les**  $v \in V \setminus S$  tels que  $(u, v) \in E$  **faire**

$D[v] \leftarrow \min(D[v], D[u] + \ell(u, v))$

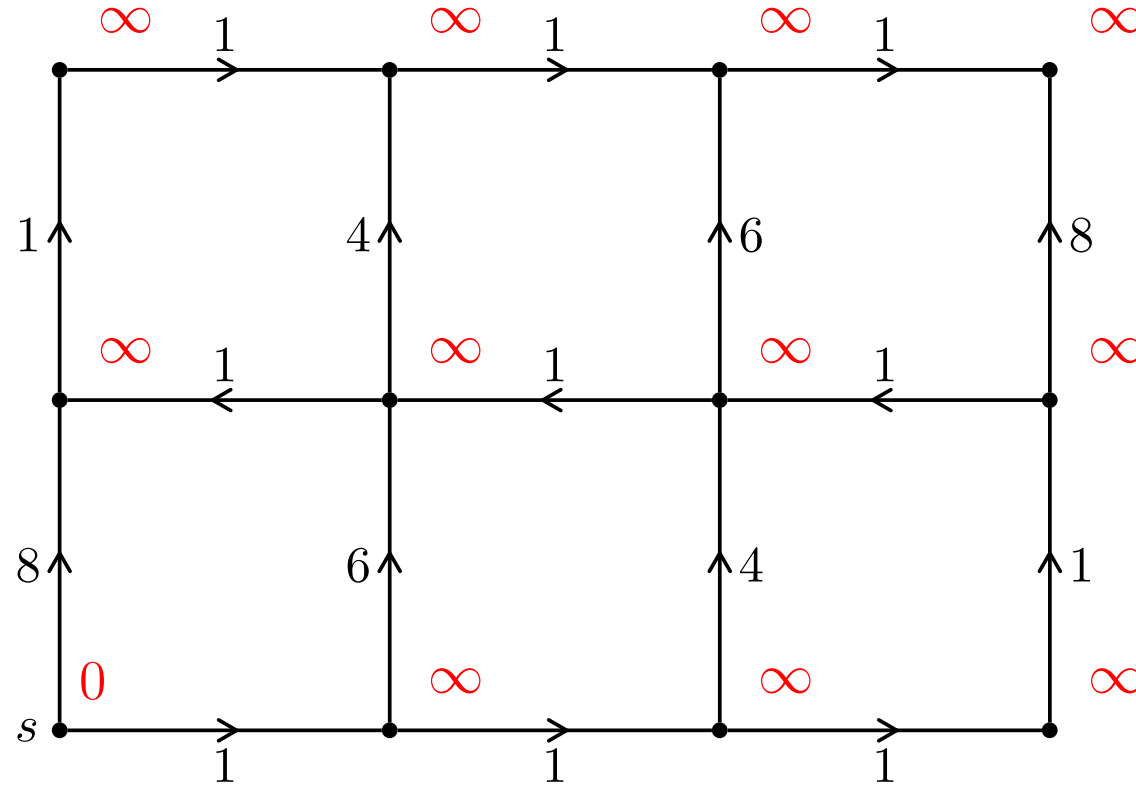
**retourner**  $D$



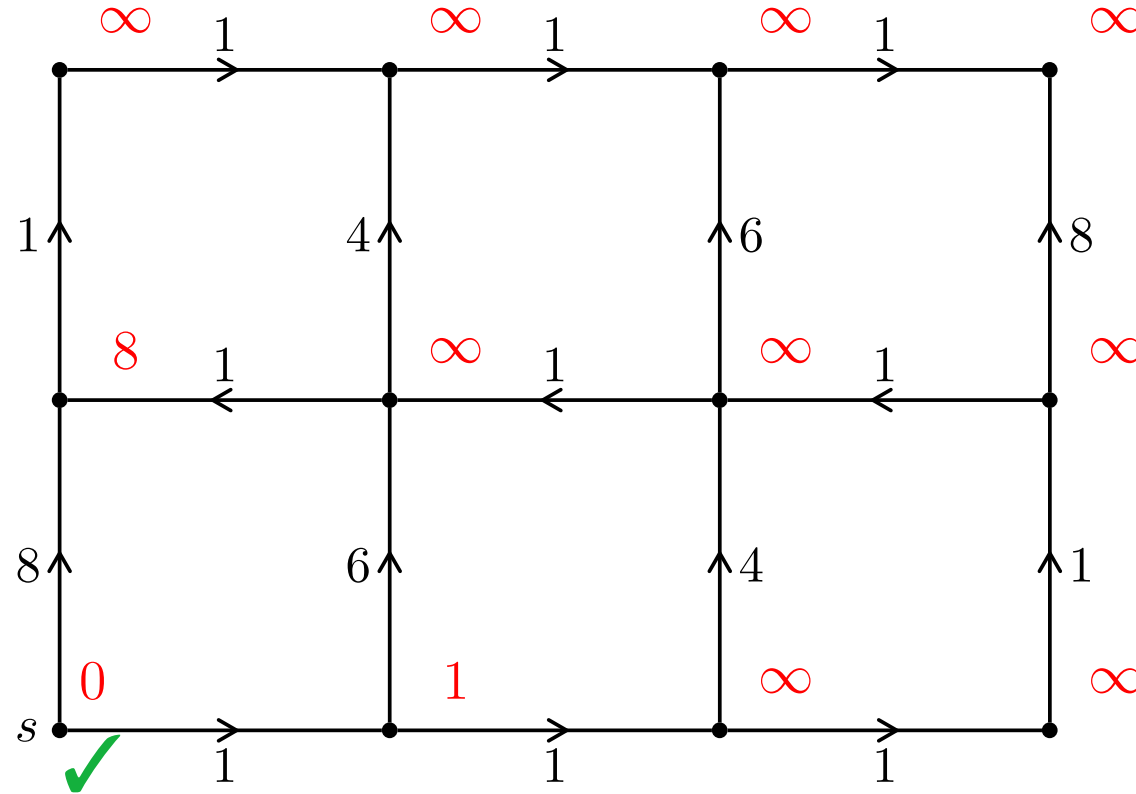
# Illustration de l'algorithme de Dijkstra



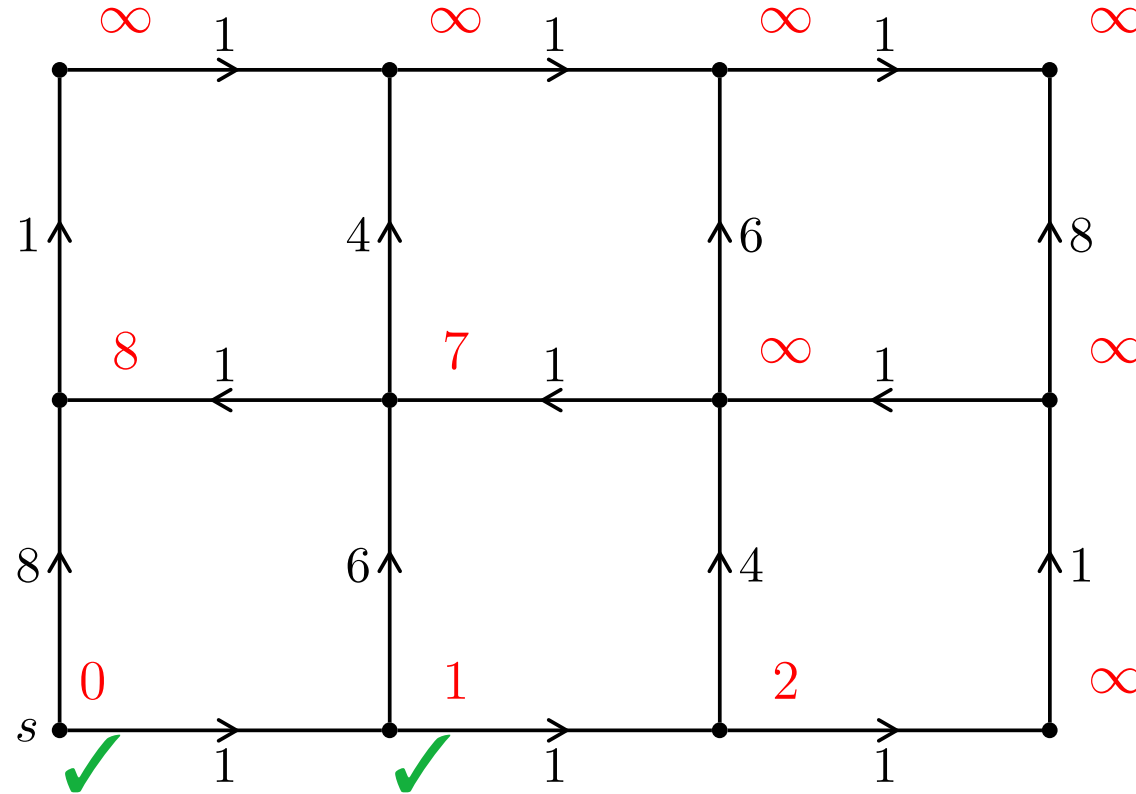
# Illustration de l'algorithme de Dijkstra



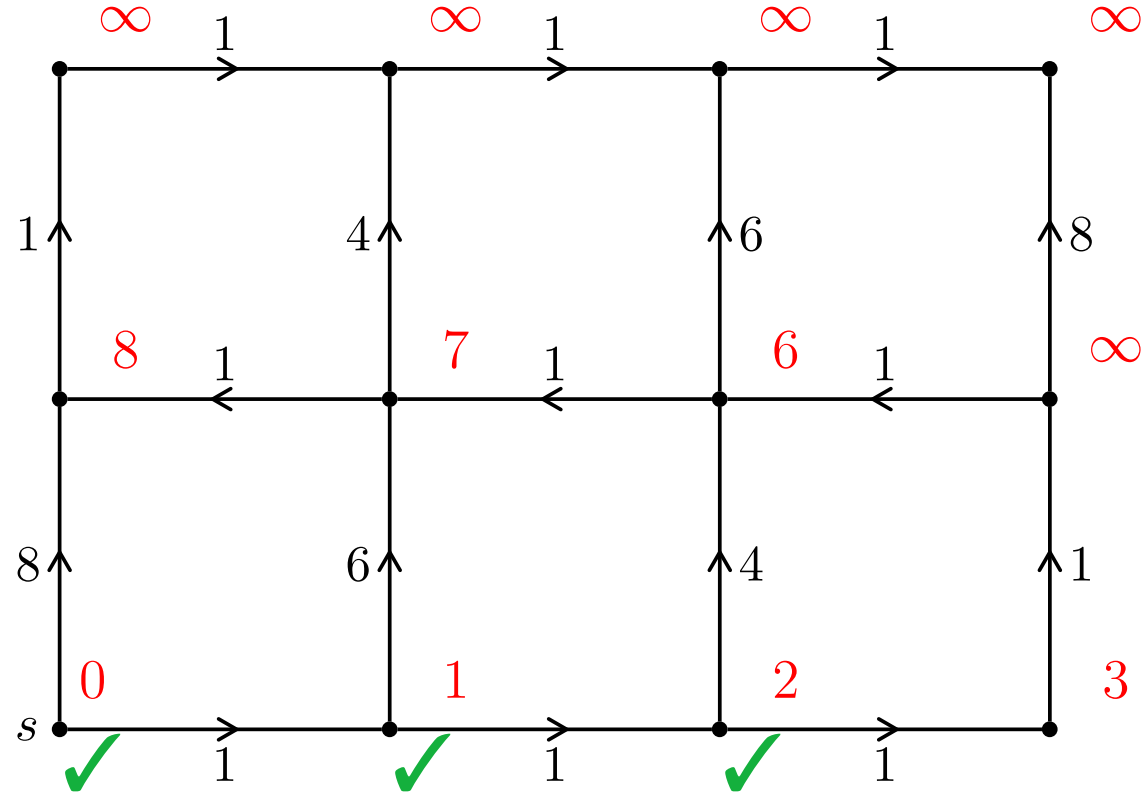
# Illustration de l'algorithme de Dijkstra



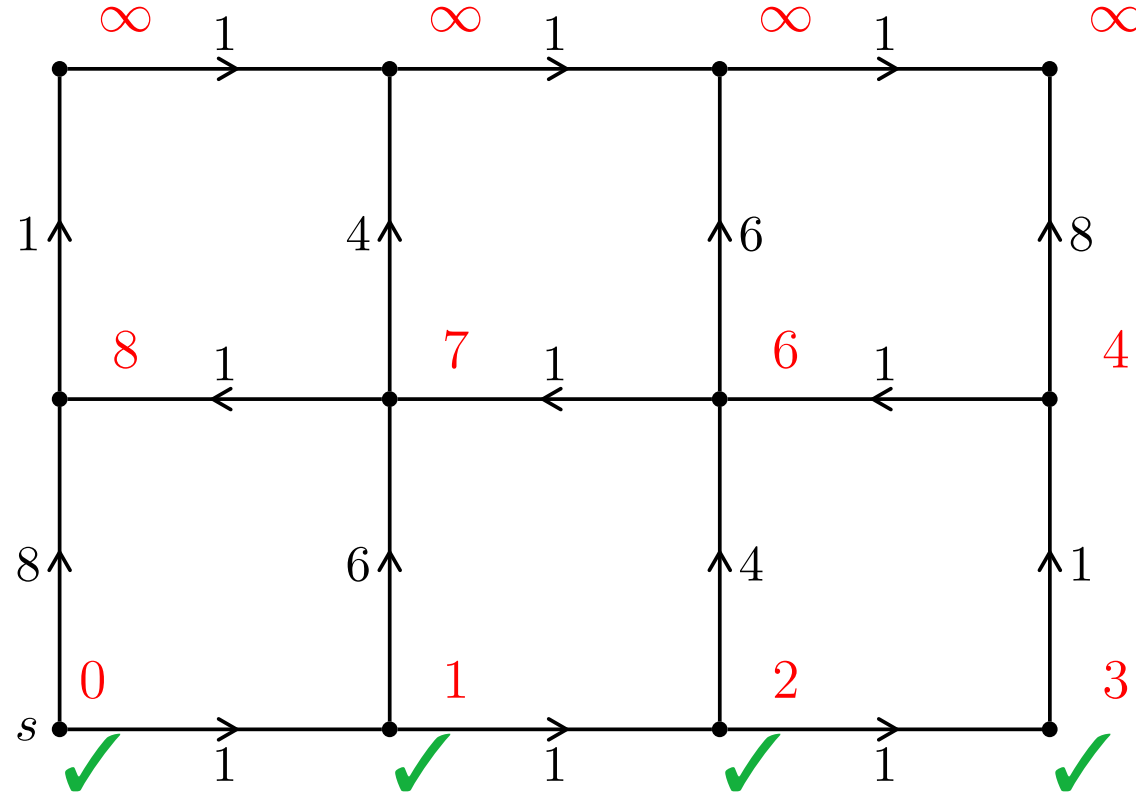
# Illustration de l'algorithme de Dijkstra



# Illustration de l'algorithme de Dijkstra

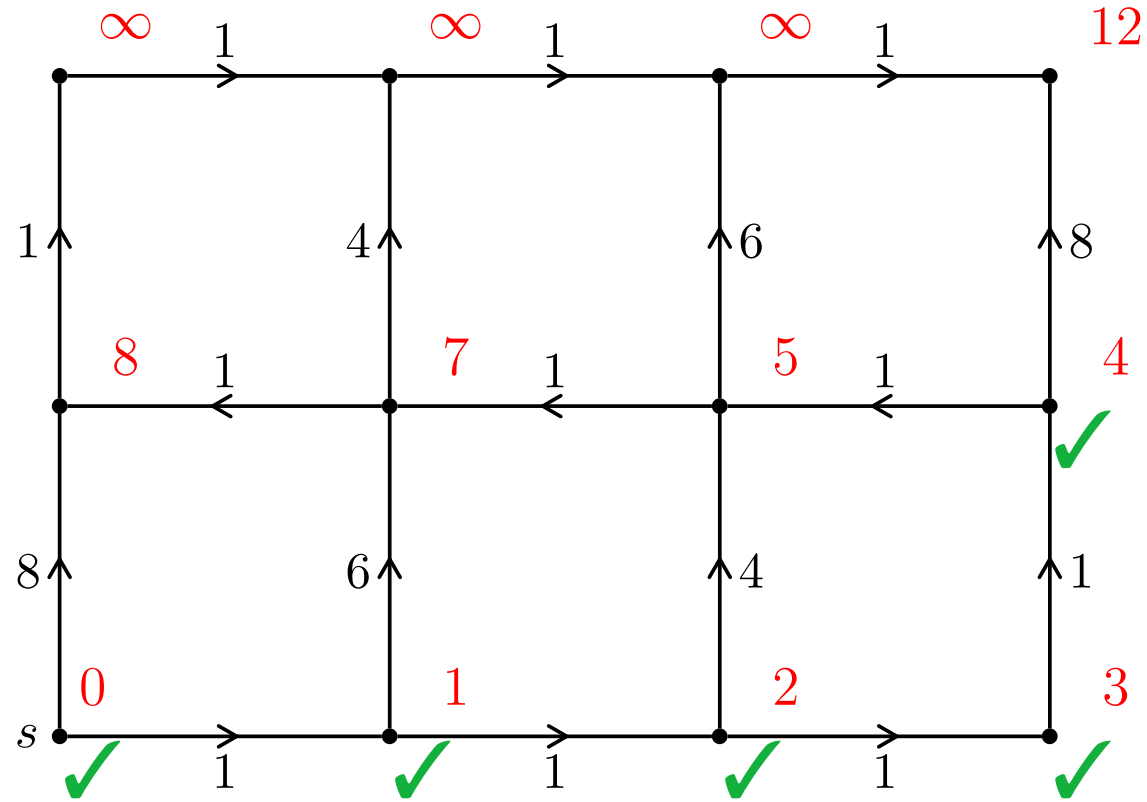


# Illustration de l'algorithme de Dijkstra

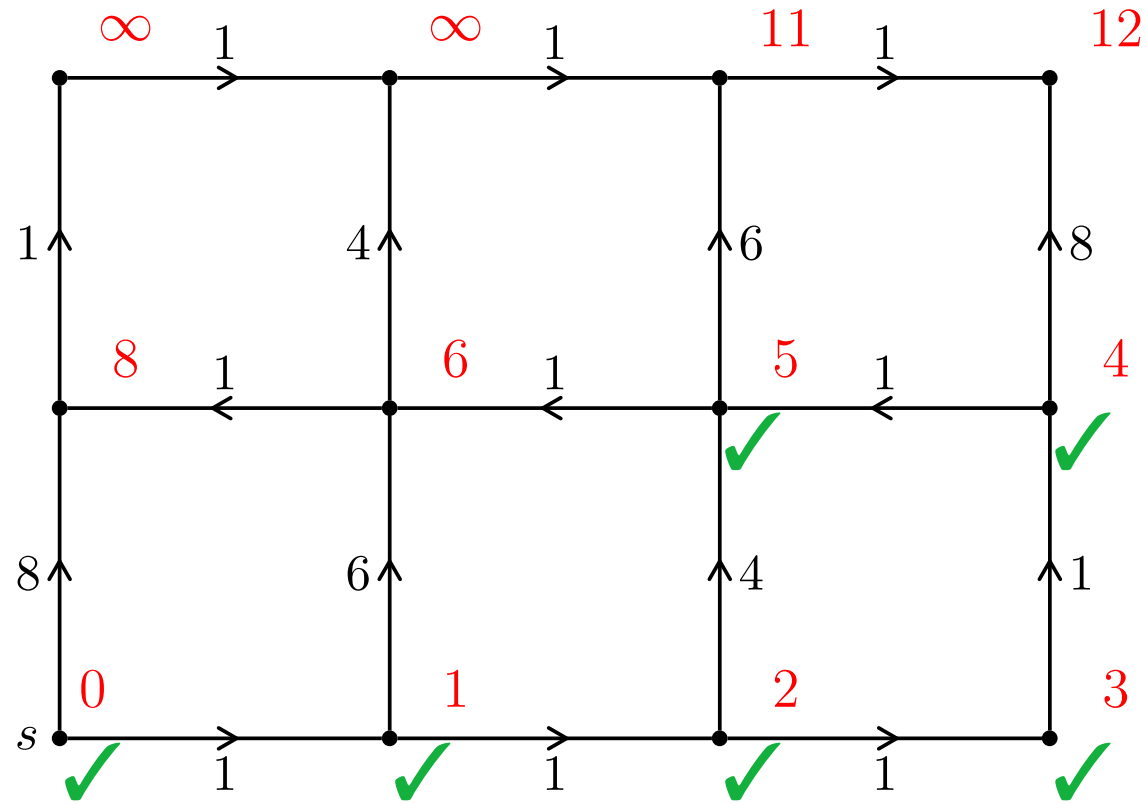




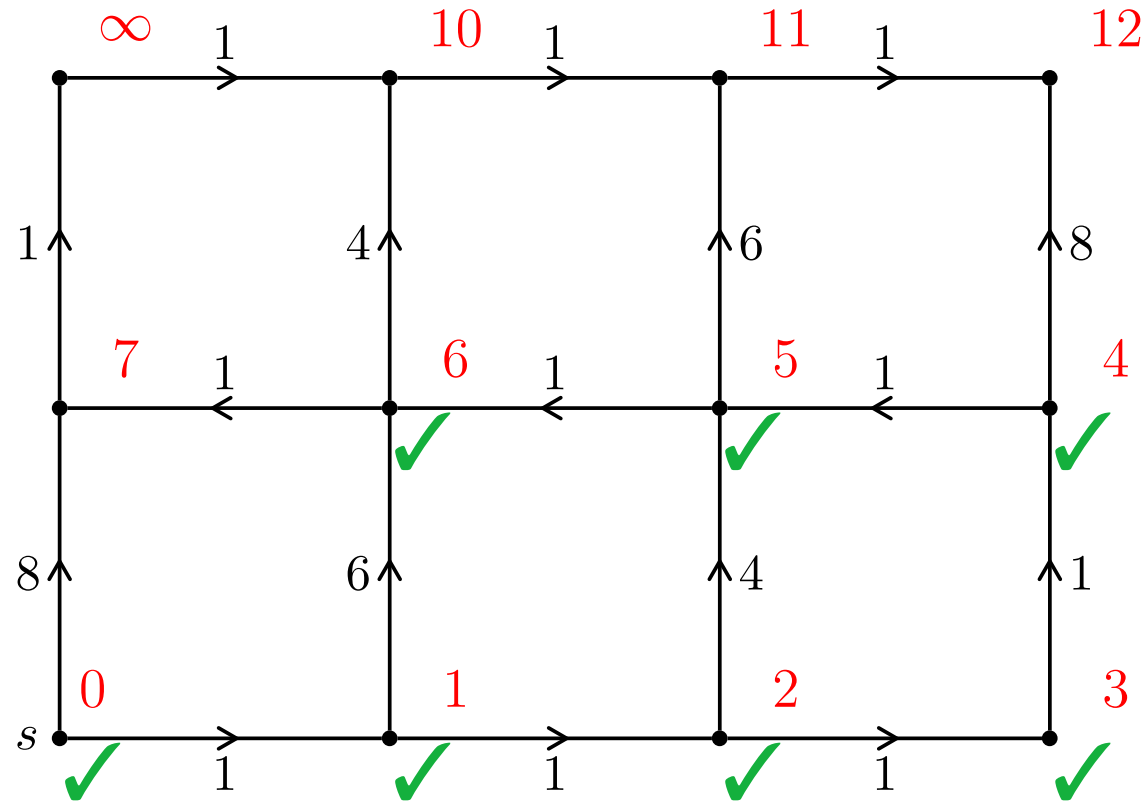
# Illustration de l'algorithme de Dijkstra



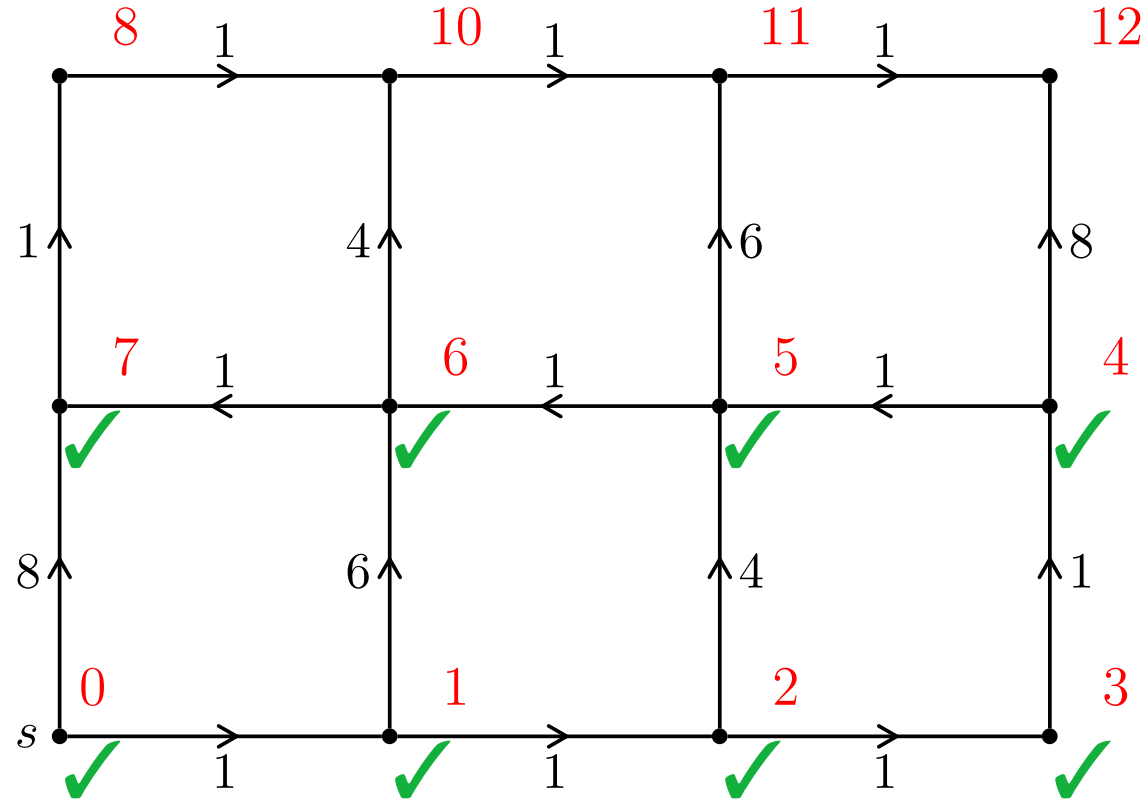
# Illustration de l'algorithme de Dijkstra



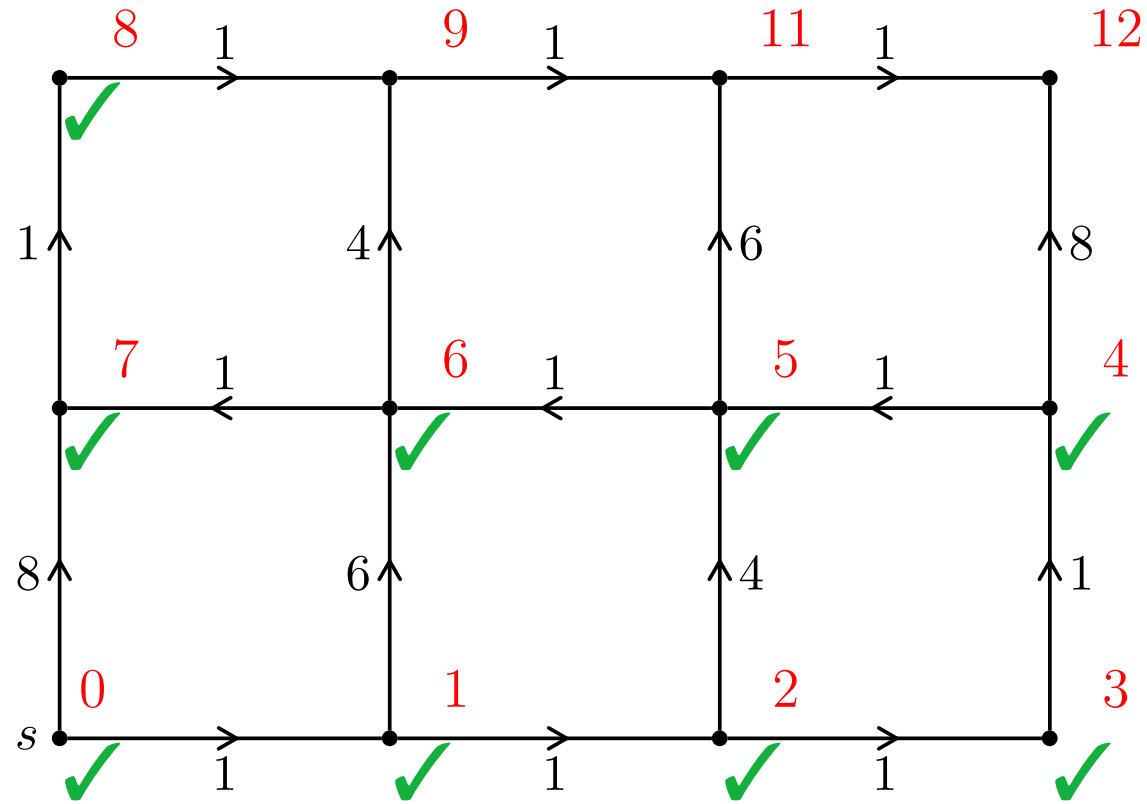
# Illustration de l'algorithme de Dijkstra



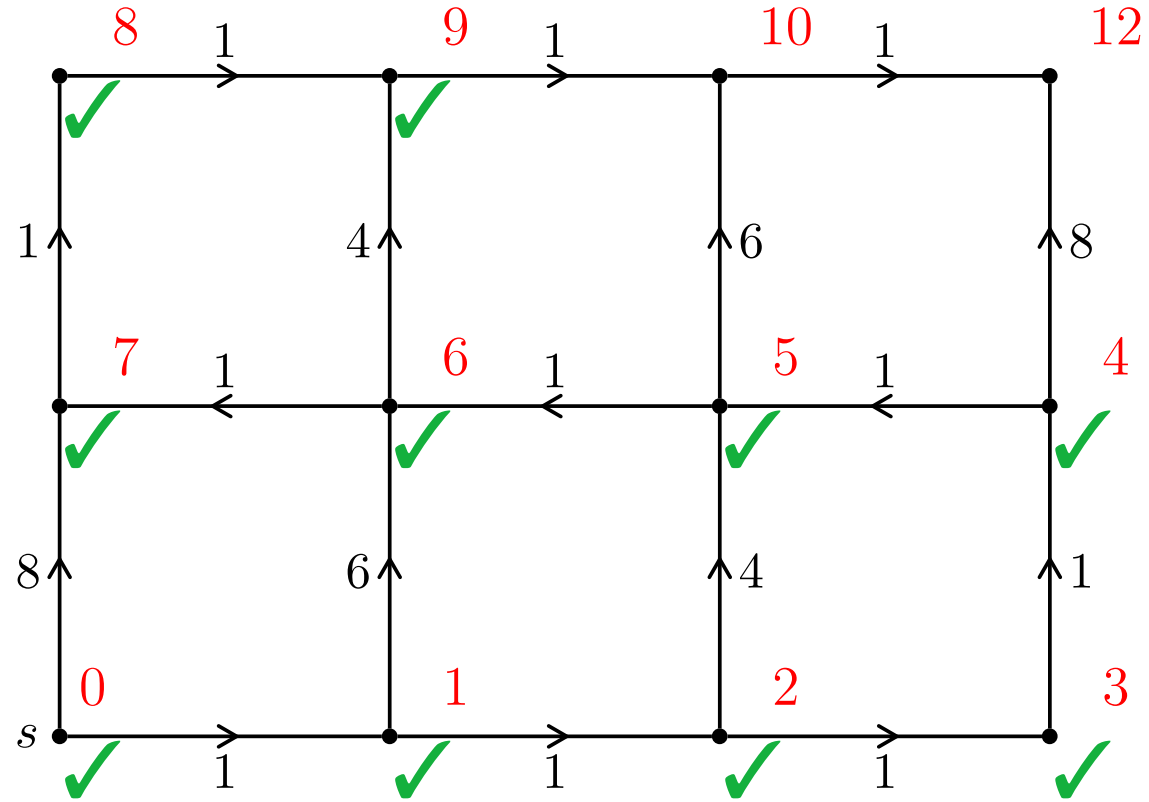
# Illustration de l'algorithme de Dijkstra



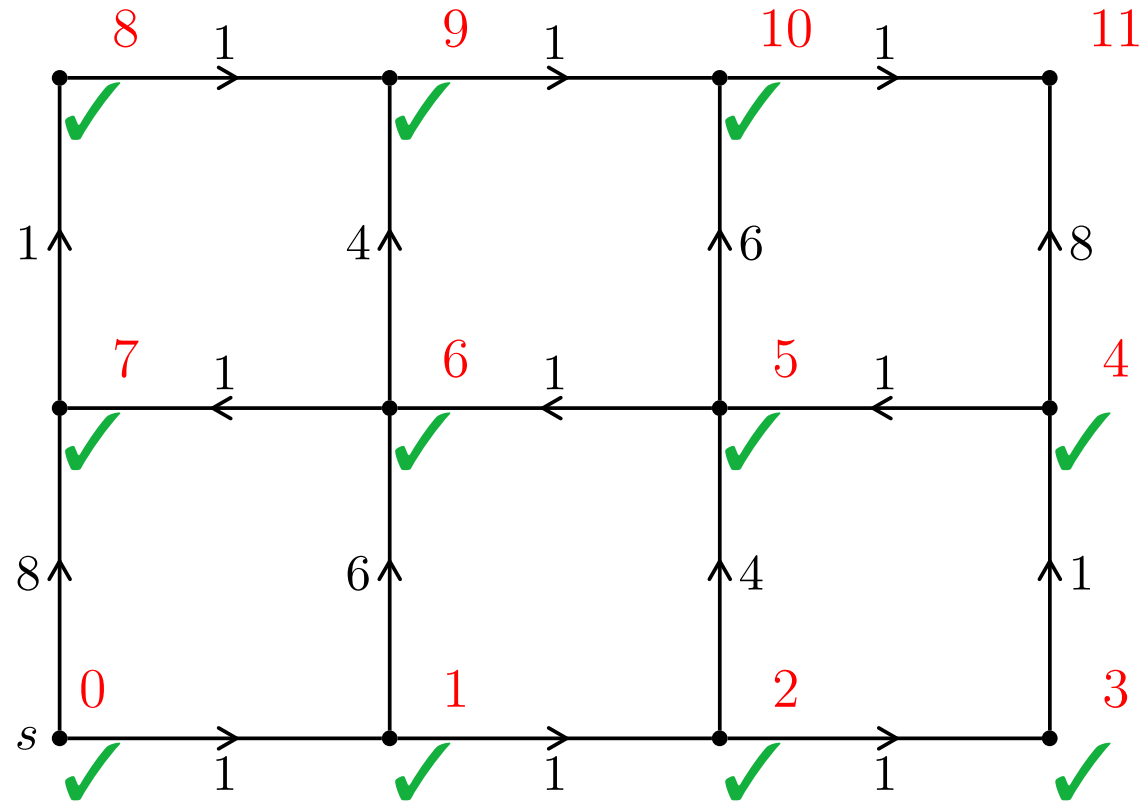
# Illustration de l'algorithme de Dijkstra



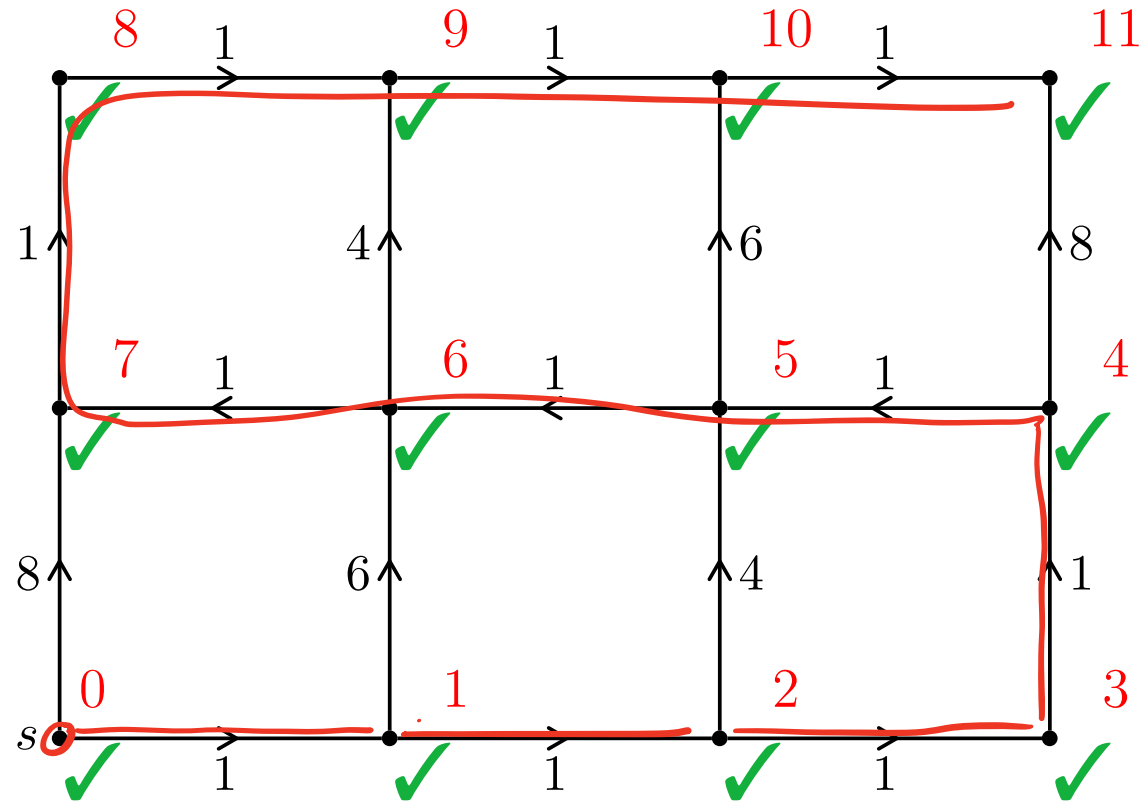
# Illustration de l'algorithme de Dijkstra



# Illustration de l'algorithme de Dijkstra



# Illustration de l'algorithme de Dijkstra



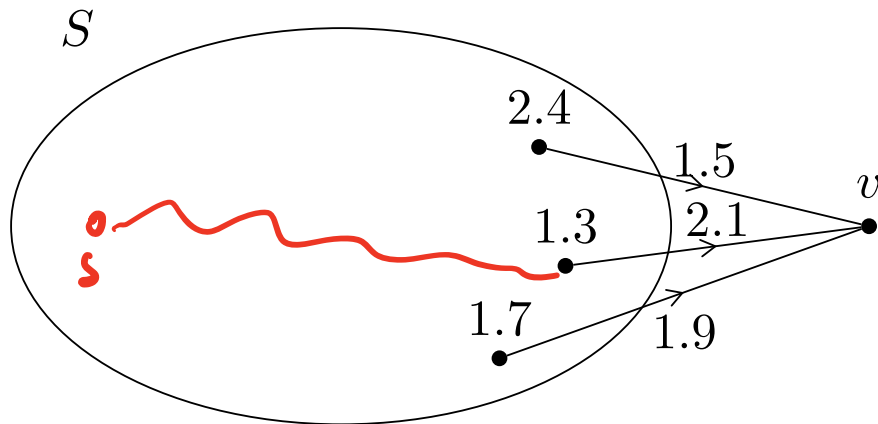


# Distances partielles

## Définition

Soit  $G = (V, E)$  un graphe orienté avec pondération  $\ell \in \mathbb{R}^+$ , et  $s \in S \subseteq V$ . Pour tout  $u \in V$ , on note  $d(u) = \text{dist}(s, u)$ . Pour tout sommet  $v \notin S$ , on définit

$$D(v) = \min \{d(u) + \ell(u, v) : u \in S \text{ et } (u, v) \in E\}.$$



$$D(v) = \min \{3.9, 3.4, 3.6\} = 3.4$$

## Justification de l'algorithme de Dijkstra (1/3)

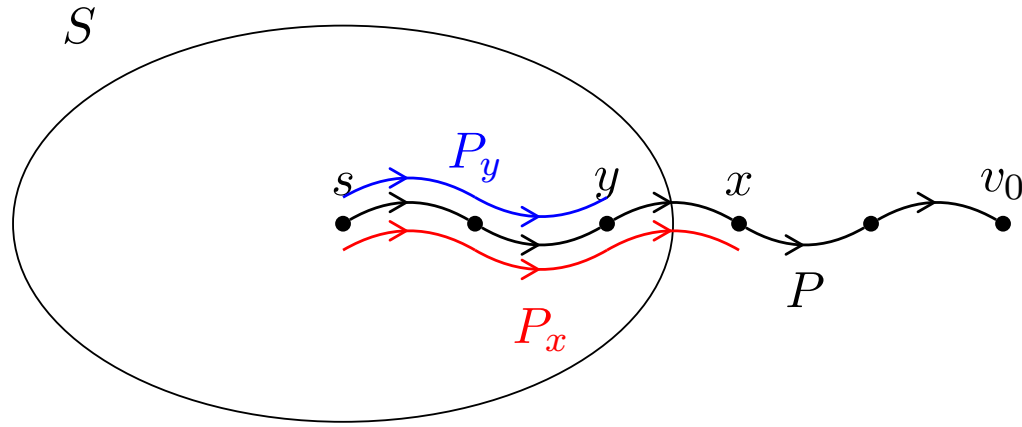
### Lemme

Soit  $v_0$  tel que  $D(v_0)$  est minimum, parmi tous les sommets dans  $V \setminus S$ .  
Alors,  $D(v_0) = d(v_0)$ .

### Démonstration

- Soit  $v_0$  tel que  $D(v_0)$  est minimum.
- Comme  $D(v_0)$  est la longueur d'un chemin de  $s$  à  $v_0$ , on a  $D(v_0) \geq d(v_0)$ .
- Si  $D(v_0) \leq d(v_0)$  alors la preuve est terminée.
- Sinon, supposons par l'absurde que  $D(v_0) > d(v_0)$ .
- Soit  $P$  un plus court chemin de  $s$  vers  $v_0$  (son poids est alors  $d(v_0)$ ).
- Soit  $x$  le premier sommet de  $P$  qui n'appartient pas à  $S$ .

## Justification de l'algorithme de Dijkstra (2/3)



### Démonstration (suite)

- Soit  $y \in S$  le prédécesseur de  $x$  dans ce chemin.
- Soit  $P_y$  le sous-chemin de  $P$  de  $s$  vers  $y$ .
- $P_y$  est un plus court chemin de  $s$  vers  $y$  (principe de sous optimalité).
- Soit  $P_x$  le sous chemin de  $P$  de  $s$  vers  $x$ .

## Justification de l'algorithme de Dijkstra (3/3)

### Démonstration (suite)

- La longueur de  $P_x$  est  $d(y) + \ell(y, x)$ .
- Ceci entraîne que  $D(x) \leq d(y) + \ell(y, x) \leq d(v_0) < D(v_0)$ .
  - Première inégalité : par définition de  $D$ .
  - Deuxième inégalité : tous les poids sont  $\geq 0$ .
  - Troisième inégalité : par hypothèse.
- Implique  $D(x) < D(v_0)$  — contradiction avec le choix de  $D(v_0)$ .