

# Projeto e Análise de Algoritmos

Caminhos mínimos

Lehilton Pedrosa

Primeiro Semestre de 2020

Caminhos mínimos com uma origem

# Problema do Caminho Mínimo

Considere  $(G, \omega)$ ,

- ▶  $G$  é um grafo direcionado
- ▶  $\omega$  associa um peso  $\omega(u, v)$  para cada aresta  $(u, v)$

Definimos os seguintes problemas:

## 1. Problema do Caminho Mínimo entre Dois Vértices

Dados  $s$  e  $t$ , encontre um caminho de peso mínimo de  $s$  a  $t$ .

## 2. Problema dos Caminhos Mínimos com Mesma Origem

Dado  $s$ , encontrar um caminho de peso mínimo de  $s$  a  $v$  para **cada** vértice  $v$  de  $G$ .

# Problema do Caminho Mínimo

Considere  $(G, \omega)$ ,

- ▶  $G$  é um grafo direcionado
- ▶  $\omega$  associa um peso  $\omega(u, v)$  para cada aresta  $(u, v)$

Definimos os seguintes problemas:

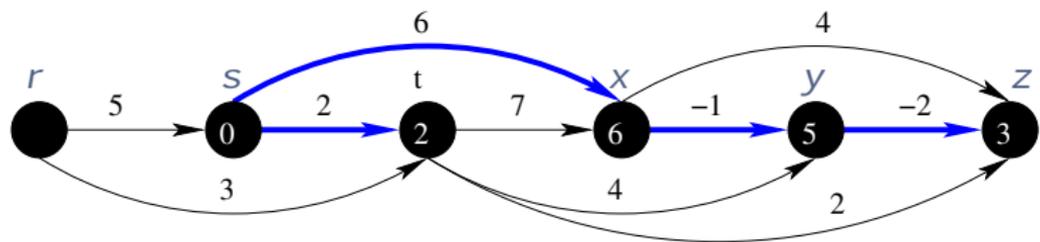
## 1. Problema do Caminho Mínimo entre Dois Vértices

Dados  $s$  e  $t$ , encontre um caminho de peso mínimo de  $s$  a  $t$ .

## 2. Problema dos Caminhos Mínimos com Mesma Origem

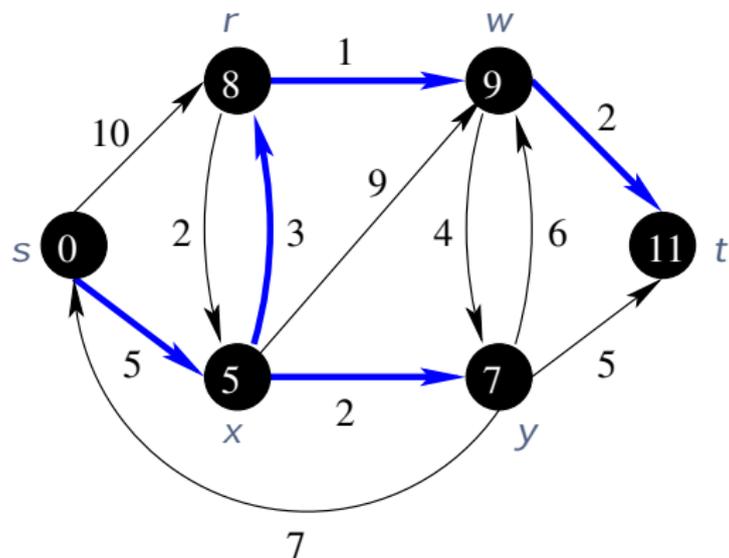
Dado  $s$ , encontrar um caminho de peso mínimo de  $s$  a  $v$  para **cada** vértice  $v$  de  $G$ .

# Exemplo: grafo direcionado acíclico



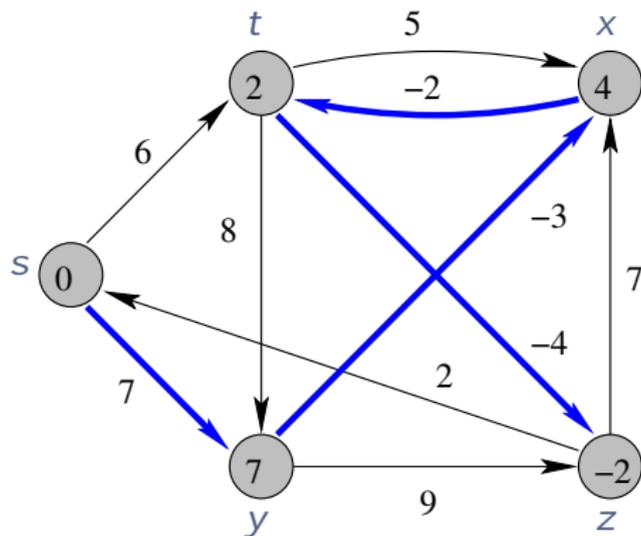
$v$	$s$	$r$	$t$	$x$	$y$	$z$
$\text{dist}(s, v)$	0	$\infty$	2	6	5	3

# Exemplo: grafo direcionado sem arestas negativas



$v$	$s$	$r$	$x$	$y$	$w$	$t$
$\text{dist}(s, v)$	0	8	5	7	9	11

# Exemplo: grafo direcionado com arestas negativas



$v$	$s$	$t$	$x$	$y$	$z$
$\text{dist}(s, v)$	0	2	4	7	-2

# Ideias comuns a todos os algoritmos

- ▶ Ideia similar à Busca em Largura a partir de  $s$ .
- ▶ Para cada vértice  $v \in V[G]$  associamos predecessor  $\pi[v]$ .
- ▶ A saída é uma **árvore de caminhos mínimos** com raiz  $s$ .
- ▶ Um caminho de  $s$  a  $v$  nessa árvore é um caminho mínimo de  $s$  a  $v$  em  $G$ .

# Ideias comuns a todos os algoritmos

- ▶ Ideia similar à Busca em Largura a partir de  $s$ .
- ▶ Para cada vértice  $v \in V[G]$  associamos predecessor  $\pi[v]$ .
- ▶ A saída é uma **árvore de caminhos mínimos** com raiz  $s$ .
- ▶ Um caminho de  $s$  a  $v$  nessa árvore é um caminho mínimo de  $s$  a  $v$  em  $G$ .

# Ideias comuns a todos os algoritmos

- ▶ Ideia similar à Busca em Largura a partir de  $s$ .
- ▶ Para cada vértice  $v \in V[G]$  associamos **predecessor**  $\pi[v]$ .
- ▶ A saída é uma **árvore de caminhos mínimos** com raiz  $s$ .
- ▶ Um caminho de  $s$  a  $v$  nessa árvore é um caminho mínimo de  $s$  a  $v$  em  $G$ .

# Ideias comuns a todos os algoritmos

- ▶ Ideia similar à Busca em Largura a partir de  $s$ .
- ▶ Para cada vértice  $v \in V[G]$  associamos **predecessor**  $\pi[v]$ .
- ▶ A saída é uma **árvore de caminhos mínimos** com raiz  $s$ .
- ▶ Um caminho de  $s$  a  $v$  nessa árvore é um caminho mínimo de  $s$  a  $v$  em  $G$ .

# Ideias comuns a todos os algoritmos

- ▶ Ideia similar à Busca em Largura a partir de  $s$ .
- ▶ Para cada vértice  $v \in V[G]$  associamos **predecessor**  $\pi[v]$ .
- ▶ A saída é uma **árvore de caminhos mínimos** com raiz  $s$ .
- ▶ Um caminho de  $s$  a  $v$  nessa árvore é um caminho mínimo de  $s$  a  $v$  em  $G$ .

# Subestrutura ótima de caminhos mínimos

## Teorema

Seja  $(G, \omega)$  um grafo direcionado **sem ciclos negativos** e seja

$$P = (v_1, v_2, \dots, v_k)$$

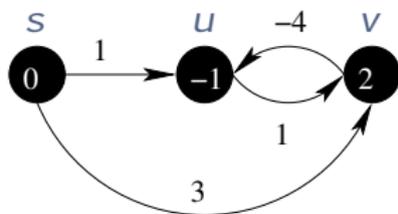
um caminho mínimo de  $v_1$  a  $v_k$ . Então para quaisquer  $i, j$  com  $1 \leq i \leq j \leq k$ ,

$$P_{ij} = (v_i, v_{i+1}, \dots, v_j)$$

é um caminho mínimo de  $v_i$  a  $v_j$ .

# Subestrutura ótima de caminhos mínimos

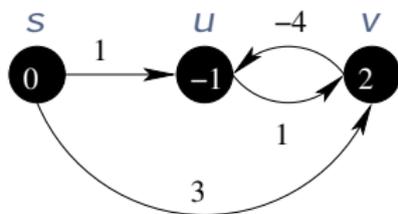
A subestrutura ótima **não** vale se  $(G, \omega)$  contiver **ciclos negativos**.



- ▶ O caminho mínimo de  $s$  a  $v$  é  $(s, u, v)$  com peso  $1 + 1 = 2$ .
- ▶ Mas  $(s, u)$  **não** é um caminho mínimo de  $s$  a  $u$ .
- ▶ O caminho mínimo de  $s$  a  $u$  é  $(s, v, u)$  com peso  $3 - 4 = -1$ .

# Subestrutura ótima de caminhos mínimos

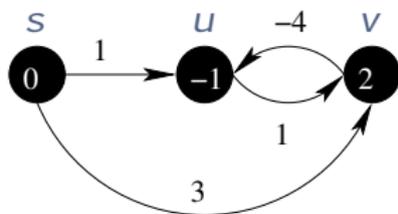
A subestrutura ótima **não** vale se  $(G, \omega)$  contiver **ciclos negativos**.



- ▶ O caminho mínimo de  $s$  a  $v$  é  $(s, u, v)$  com peso  $1 + 1 = 2$ .
- ▶ Mas  $(s, u)$  **não** é um caminho mínimo de  $s$  a  $u$ .
- ▶ O caminho mínimo de  $s$  a  $u$  é  $(s, v, u)$  com peso  $3 - 4 = -1$ .

# Subestrutura ótima de caminhos mínimos

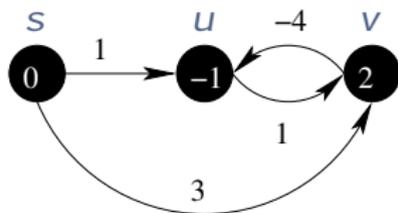
A subestrutura ótima **não** vale se  $(G, \omega)$  contiver **ciclos negativos**.



- ▶ O caminho mínimo de  $s$  a  $v$  é  $(s, u, v)$  com peso  $1 + 1 = 2$ .
- ▶ Mas  $(s, u)$  **não** é um caminho mínimo de  $s$  a  $u$ .
- ▶ O caminho mínimo de  $s$  a  $u$  é  $(s, v, u)$  com peso  $3 - 4 = -1$ .

# Subestrutura ótima de caminhos mínimos

A subestrutura ótima **não** vale se  $(G, \omega)$  contiver **ciclos negativos**.



- ▶ O caminho mínimo de  $s$  a  $v$  é  $(s, u, v)$  com peso  $1 + 1 = 2$ .
- ▶ Mas  $(s, u)$  **não** é um caminho mínimo de  $s$  a  $u$ .
- ▶ O caminho mínimo de  $s$  a  $u$  é  $(s, v, u)$  com peso  $3 - 4 = -1$ .

# Estimativa de distâncias

- ▶ Para cada  $v \in V[G]$ , a distância  $\text{dist}(s, v)$  é o peso de um caminho mínimo de  $s$  a  $v$ .
- ▶ Para cada  $v \in V[G]$ , os algoritmos irão associar um **estimativa da distância** na variável  $d[v]$ .

# Estimativa de distâncias

- ▶ Para cada  $v \in V[G]$ , a distância  $\text{dist}(s, v)$  é o peso de um caminho mínimo de  $s$  a  $v$ .
- ▶ Para cada  $v \in V[G]$ , os algoritmos irão associar um **estimativa da distância** na variável  $d[v]$ .

# Estimativa de distâncias

- ▶ Para cada  $v \in V[G]$ , a distância  $\text{dist}(s, v)$  é o peso de um caminho mínimo de  $s$  a  $v$ .
- ▶ Para cada  $v \in V[G]$ , os algoritmos irão associar um **estimativa da distância** na variável  $d[v]$ .

# Inicialização

```
INITIALIZE-SINGLE-SOURCE( $G, s$ )  
1  para cada vértice  $v \in V[G]$  faça  
2     $d[v] \leftarrow \infty$   
3     $\pi[v] \leftarrow \text{NIL}$   
4     $d[s] \leftarrow 0$ 
```

Mantemos as invariantes

- ▶  $d[v]$  é sempre maior ou igual a  $\text{dist}(s, v)$
- ▶ até aquele momento, o algoritmo encontrou algum caminho de  $s$  a  $v$  com peso  $d[v]$ .
- ▶ esse caminho pode ser recuperado por meio dos predecessores  $\pi[\ ]$ .

```
INITIALIZE-SINGLE-SOURCE( $G, s$ )  
1  para cada vértice  $v \in V[G]$  faça  
2       $d[v] \leftarrow \infty$   
3       $\pi[v] \leftarrow \text{NIL}$   
4       $d[s] \leftarrow 0$ 
```

Mantemos as invariantes

- ▶  $d[v]$  é sempre maior ou igual a  $\text{dist}(s, v)$
- ▶ até aquele momento, o algoritmo encontrou algum caminho de  $s$  a  $v$  com peso  $d[v]$ .
- ▶ esse caminho pode ser recuperado por meio dos predecessores  $\pi[\ ]$ .

```
INITIALIZE-SINGLE-SOURCE( $G, s$ )  
1  para cada vértice  $v \in V[G]$  faça  
2     $d[v] \leftarrow \infty$   
3     $\pi[v] \leftarrow \text{NIL}$   
4     $d[s] \leftarrow 0$ 
```

Mantemos as invariantes

- ▶  $d[v]$  é sempre maior ou igual a  $\text{dist}(s, v)$
- ▶ até aquele momento, o algoritmo encontrou algum caminho de  $s$  a  $v$  com peso  $d[v]$ .
- ▶ esse caminho pode ser recuperado por meio dos predecessores  $\pi[\ ]$ .

# Inicialização

```
INITIALIZE-SINGLE-SOURCE( $G, s$ )  
1  para cada vértice  $v \in V[G]$  faça  
2     $d[v] \leftarrow \infty$   
3     $\pi[v] \leftarrow \text{NIL}$   
4     $d[s] \leftarrow 0$ 
```

Mantemos as invariantes

- ▶  $d[v]$  é sempre maior ou igual a  $\text{dist}(s, v)$
- ▶ até aquele momento, o algoritmo encontrou algum caminho de  $s$  a  $v$  com peso  $d[v]$ .
- ▶ esse caminho pode ser recuperado por meio dos predecessores  $\pi[\ ]$ .

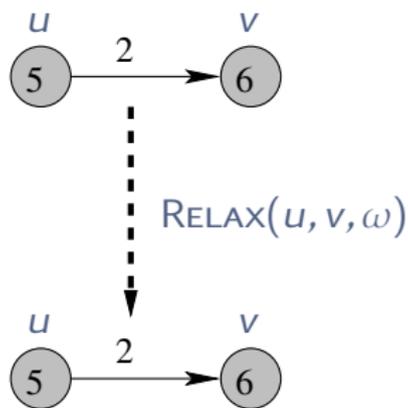
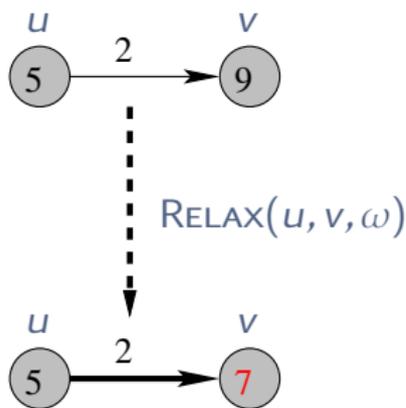
```
INITIALIZE-SINGLE-SOURCE( $G, s$ )  
1  para cada vértice  $v \in V[G]$  faça  
2     $d[v] \leftarrow \infty$   
3     $\pi[v] \leftarrow \text{NIL}$   
4     $d[s] \leftarrow 0$ 
```

Mantemos as invariantes

- ▶  $d[v]$  é sempre maior ou igual a  $\text{dist}(s, v)$
- ▶ até aquele momento, o algoritmo encontrou algum caminho de  $s$  a  $v$  com peso  $d[v]$ .
- ▶ esse caminho pode ser recuperado por meio dos predecessores  $\pi[\ ]$ .

# Relaxação

Tenta melhorar a estimativa  $d[v]$  examinando  $(u, v)$ .

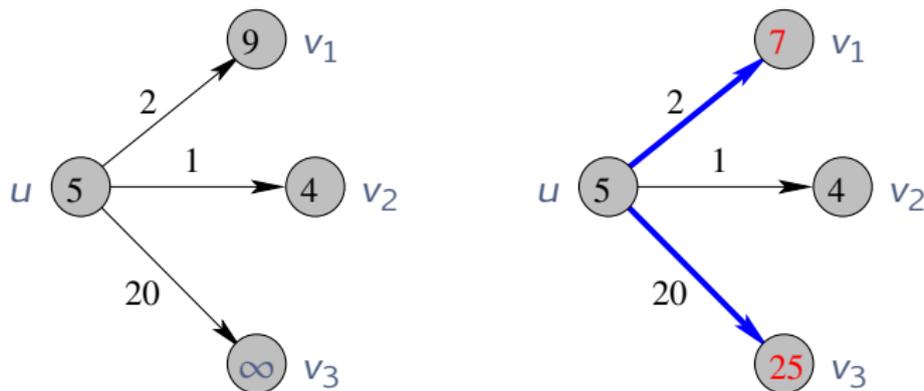


**RELAX** $(u, v, \omega)$

- 1 se  $d[v] > d[u] + \omega(u, v)$  então
- 2  $d[v] \leftarrow d[u] + \omega(u, v)$
- 3  $\pi[v] \leftarrow u$

## Relaxação dos vizinhos

Em cada iteração o algoritmo seleciona um vértice  $u$  e para cada vizinho  $v$  de  $u$  aplica  $\text{RELAX}(u, v, \omega)$ .



**RELAX**( $u, v, \omega$ )

- 1 se  $d[v] > d[u] + \omega(u, v)$  então
- 2  $d[v] \leftarrow d[u] + \omega(u, v)$
- 3  $\pi[v] \leftarrow u$

Veremos três algoritmos baseados em **relaxação** para tipos de subcasos diferentes do Problema de Caminhos Mínimos.

- ▶  $G$  é acíclico: aplicação de ordenação topológica
- ▶  $(G, \omega)$  não tem arestas de peso negativo: algoritmo de Dijkstra
- ▶  $(G, \omega)$  pode ter arestas de peso negativo, mas não contém ciclos negativos: algoritmo de Bellman-Ford.

Veremos três algoritmos baseados em **relaxação** para tipos de subcasos diferentes do Problema de Caminhos Mínimos.

- ▶  $G$  é acíclico: aplicação de ordenação topológica
- ▶  $(G, \omega)$  não tem arestas de peso negativo: algoritmo de Dijkstra
- ▶  $(G, \omega)$  pode ter arestas de peso negativo, mas não contém ciclos negativos: algoritmo de Bellman-Ford.

Veremos três algoritmos baseados em **relaxação** para tipos de subcasos diferentes do Problema de Caminhos Mínimos.

- ▶  $G$  é acíclico: aplicação de ordenação topológica
- ▶  $(G, \omega)$  não tem arestas de peso negativo: algoritmo de Dijkstra
- ▶  $(G, \omega)$  pode ter arestas de peso negativo, mas não contém ciclos negativos: algoritmo de Bellman-Ford.

Veremos três algoritmos baseados em **relaxação** para tipos de subcasos diferentes do Problema de Caminhos Mínimos.

- ▶  $G$  é acíclico: aplicação de ordenação topológica
- ▶  $(G, \omega)$  não tem arestas de peso negativo: algoritmo de Dijkstra
- ▶  $(G, \omega)$  pode ter arestas de peso negativo, mas não contém ciclos negativos: algoritmo de Bellman-Ford.

## Caminhos mínimos em grafos acíclicos

# Caminhos mínimos em grafos acíclicos

## Entrada:

- ▶ grafo direcionado acíclico  $G = (V, E)$
- ▶ função de peso  $\omega$  nas arestas
- ▶ origem  $s$ .

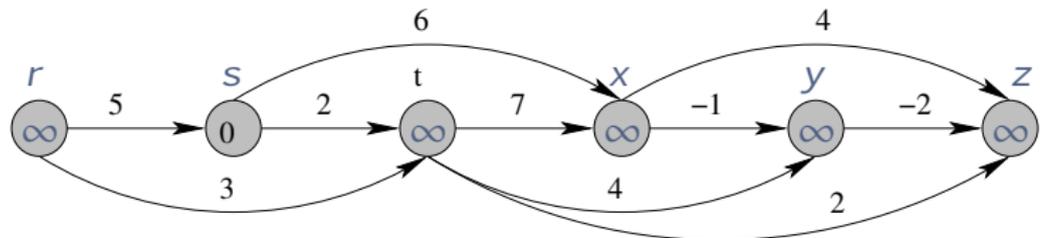
## Saída:

- ▶ vetor  $d$  com  $d[v] = \text{dist}(s, v)$  para  $v \in V$
- ▶ vetor  $\pi$  definindo uma **árvore de caminhos mínimos**

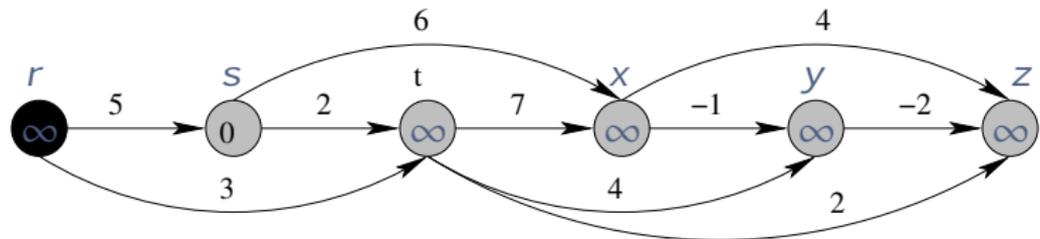
### DAG-SHORTEST-PATHS( $G, \omega, s$ )

- 1 Ordene topologicamente os vértices de  $G$
- 2 INITIALIZE-SINGLE-SOURCE( $G, s$ )
- 3 para cada vértice  $u$  na ordem topológica faça
- 4     para cada  $v \in \text{Adj}[u]$  faça
- 5         RELAX( $u, v, \omega$ )
- 6 devolva  $d, \pi$

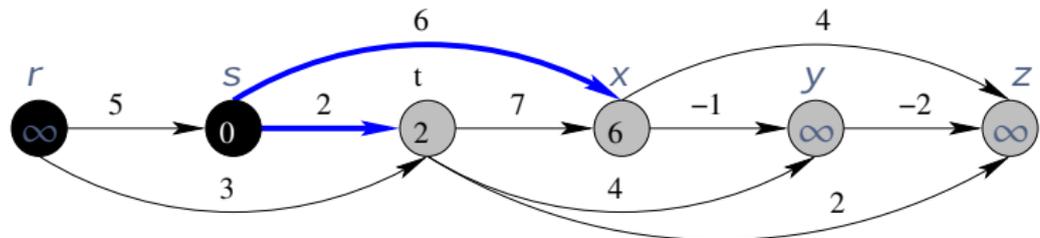
# Exemplo



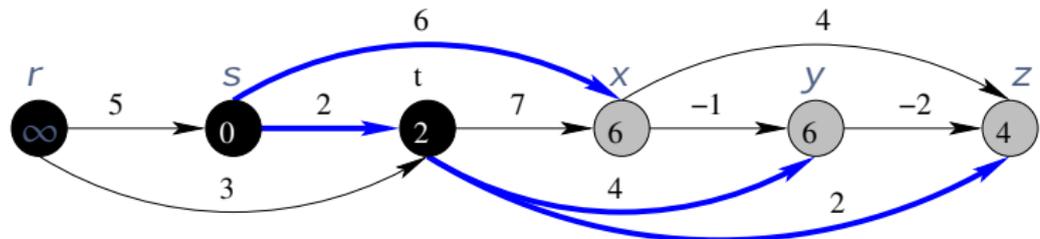
# Exemplo



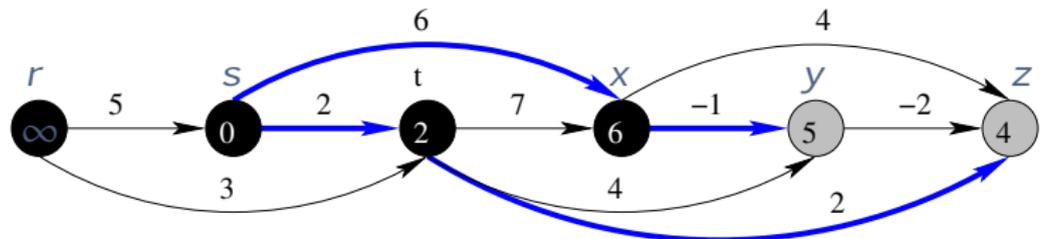
# Exemplo



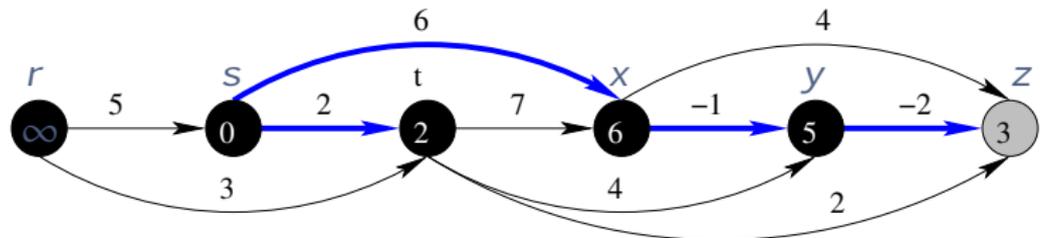
# Exemplo



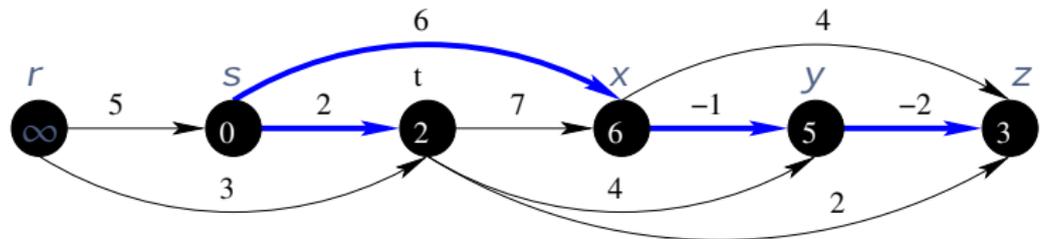
# Exemplo



# Exemplo



# Exemplo



## DAG-SHORTEST-PATHS( $G, \omega, s$ )

- 1 Ordene topologicamente os vértices de  $G$
- 2 INITIALIZE-SINGLE-SOURCE( $G, s$ )
- 3 para cada vértice  $u$  na ordem topológica faça
- 4     para cada  $v \in \text{Adj}[u]$  faça
- 5         RELAX( $u, v, \omega$ )
- 6 devolva  $d, \pi$

Linha(s)	Tempo total
1	$O(V + E)$
2	$O(V)$
3-5	$O(V + E)$

Complexidade de DAG-SHORTEST-PATHS:  $O(V + E)$

## DAG-SHORTEST-PATHS( $G, \omega, s$ )

- 1 Ordene topologicamente os vértices de  $G$
- 2 INITIALIZE-SINGLE-SOURCE( $G, s$ )
- 3 para cada vértice  $u$  na ordem topológica faça
- 4     para cada  $v \in \text{Adj}[u]$  faça
- 5         RELAX( $u, v, \omega$ )
- 6 devolva  $d, \pi$

Linha(s)	Tempo total
1	$O(V + E)$
2	$O(V)$
3-5	$O(V + E)$

Complexidade de DAG-SHORTEST-PATHS:  $O(V + E)$

## DAG-SHORTEST-PATHS( $G, \omega, s$ )

- 1 Ordene topologicamente os vértices de  $G$
- 2 INITIALIZE-SINGLE-SOURCE( $G, s$ )
- 3 para cada vértice  $u$  na ordem topológica faça
- 4     para cada  $v \in \text{Adj}[u]$  faça
- 5         RELAX( $u, v, \omega$ )
- 6 devolva  $d, \pi$

Linha(s)	Tempo total
1	$O(V + E)$
2	$O(V)$
3-5	$O(V + E)$

Complexidade de DAG-SHORTEST-PATHS:  $O(V + E)$

- ▶ A correção de `DAG-SHORTEST-PATHS` pode ser demonstrada de várias formas.
- ▶ Vamos demonstrar alguns lemas e observações que serão úteis na análise de correção deste e dos outros algoritmos.

- ▶ A correção de `DAG-SHORTEST-PATHS` pode ser demonstrada de várias formas.
- ▶ Vamos demonstrar alguns lemas e observações que serão úteis na análise de correção deste e dos outros algoritmos.

As propriedades que veremos são para algoritmos que satisfazem as restrições abaixo.

- ▶ O algoritmo começa com `INITIALIZE-SINGLE-SOURCE( $G, s$ )`.
- ▶ Os valores de  $d[v]$  e  $\pi[v]$  só são modificados por meio de uma chamada de `RELAX( $u, v, w$ )` para alguma aresta  $(u, v)$ .

# Algoritmos baseados em relaxação

As propriedades que veremos são para algoritmos que satisfazem as restrições abaixo.

- ▶ O algoritmo começa com `INITIALIZE-SINGLE-SOURCE( $G, s$ )`.
- ▶ Os valores de  $d[v]$  e  $\pi[v]$  só são modificados por meio de uma chamada de `RELAX( $u, v, w$ )` para alguma aresta  $(u, v)$ .

# Algoritmos baseados em relaxação

As propriedades que veremos são para algoritmos que satisfazem as restrições abaixo.

- ▶ O algoritmo começa com `INITIALIZE-SINGLE-SOURCE( $G, s$ )`.
- ▶ Os valores de  $d[v]$  e  $\pi[v]$  só são modificados por meio de uma chamada de `RELAX( $u, v, \omega$ )` para alguma aresta  $(u, v)$ .

Ao longo de um algoritmo baseado em relaxação sempre vale:

- ▶ **Limite superior**

Vale  $d[v] \geq \text{dist}(s, v)$  e, tão logo  $d[v]$  alcança  $\text{dist}(s, v)$ , nunca mais muda.

- ▶ **Inexistência de caminho**

Se não existe caminho de  $s$  a  $v$ , então  $d[v] = \infty$ .

- ▶ **Subgrafo de predecessores**

Se  $d[v] < \infty$ , então o subgrafo dos predecessores induzido por  $\pi$  é um caminho de peso  $d[v]$ .

Ao longo de um algoritmo baseado em relaxação sempre vale:

- ▶ **Limite superior**

Vale  $d[v] \geq \text{dist}(s, v)$  e, tão logo  $d[v]$  alcança  $\text{dist}(s, v)$ , nunca mais muda.

- ▶ Inexistência de caminho

Se não existe caminho de  $s$  a  $v$ , então  $d[v] = \infty$ .

- ▶ Subgrafo de predecessores

Se  $d[v] < \infty$ , então o subgrafo dos predecessores induzido por  $\pi$  é um caminho de peso  $d[v]$ .

# Algoritmos baseados em relaxação

Ao longo de um algoritmo baseado em relaxação sempre vale:

- ▶ **Limite superior**

Vale  $d[v] \geq \text{dist}(s, v)$  e, tão logo  $d[v]$  alcança  $\text{dist}(s, v)$ , nunca mais muda.

- ▶ **Inexistência de caminho**

Se não existe caminho de  $s$  a  $v$ , então  $d[v] = \infty$ .

- ▶ **Subgrafo de predecessores**

Se  $d[v] < \infty$ , então o subgrafo dos predecessores induzido por  $\pi$  é um caminho de peso  $d[v]$ .

# Algoritmos baseados em relaxação

Ao longo de um algoritmo baseado em relaxação sempre vale:

- ▶ **Limite superior**

Vale  $d[v] \geq \text{dist}(s, v)$  e, tão logo  $d[v]$  alcança  $\text{dist}(s, v)$ , nunca mais muda.

- ▶ **Inexistência de caminho**

Se não existe caminho de  $s$  a  $v$ , então  $d[v] = \infty$ .

- ▶ **Subgrafo de predecessores**

Se  $d[v] < \infty$ , então o subgrafo dos predecessores induzido por  $\pi$  é um caminho de peso  $d[v]$ .

## Algoritmos baseados em relaxação (cont)

- ▶ **Convergência**

Se  $p$  é um caminho mínimo de  $s$  até  $v$  terminando com a aresta  $(u, v)$  e  $d[u] = \text{dist}(s, u)$ , então ao relaxar  $(u, v)$ ,  $d[v] = \text{dist}(s, v)$ , que nunca mais muda.

- ▶ **Relaxamento de caminho**

Se  $p = (v_0, v_1, \dots, v_k)$  é um caminho mínimo de  $s = v_0$  a  $v_k$  e relaxamos as arestas de  $p$  na ordem  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$ , então  $d[v_k] = \text{dist}(s, v_k)$ . A propriedade vale mesmo se tivermos realizado quaisquer outras relaxações durante a execução.

## Algoritmos baseados em relaxação (cont)

### ► Convergência

Se  $p$  é um caminho mínimo de  $s$  até  $v$  terminando com a aresta  $(u, v)$  e  $d[u] = \text{dist}(s, u)$ , então ao relaxar  $(u, v)$ ,  $d[v] = \text{dist}(s, v)$ , que nunca mais muda.

### ► Relaxamento de caminho

Se  $p = (v_0, v_1, \dots, v_k)$  é um caminho mínimo de  $s = v_0$  a  $v_k$  e relaxamos as arestas de  $p$  na ordem  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$ , então  $d[v_k] = \text{dist}(s, v_k)$ . A propriedade vale mesmo se tivermos realizado quaisquer outras relaxações durante a execução.

## Algoritmos baseados em relaxação (cont)

### ► Convergência

Se  $p$  é um caminho mínimo de  $s$  até  $v$  terminando com a aresta  $(u, v)$  e  $d[u] = \text{dist}(s, u)$ , então ao relaxar  $(u, v)$ ,  $d[v] = \text{dist}(s, v)$ , que nunca mais muda.

### ► Relaxamento de caminho

Se  $p = (v_0, v_1, \dots, v_k)$  é um caminho mínimo de  $s = v_0$  a  $v_k$  e relaxamos as arestas de  $p$  na ordem  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$ , então  $d[v_k] = \text{dist}(s, v_k)$ . A propriedade vale mesmo se tivermos realizado quaisquer outras relaxações durante a execução.

- ▶ Seja  $v$  um vértice e suponha que  $p = (v_0, v_1, \dots, v_k)$  é um caminho mínimo de  $s = v_0$  a  $v = v_k$ .
- ▶ Como  $v_0, v_1, \dots, v_k$  aparecem em ordem na ordenação topológica, as arestas  $(v_0, v_1), \dots, (v_{k-1}, v_k)$  são relaxadas em ordem.
- ▶ Logo, pela propriedade do relaxamento de caminho, o algoritmo computa corretamente  $d[v] = \text{dist}(s, v)$  para cada  $v \in V$ .

Também é fácil ver que  $\pi[\ ]$  define uma árvore de caminhos mínimos. (Por quê?)

- ▶ Seja  $v$  um vértice e suponha que  $p = (v_0, v_1, \dots, v_k)$  é um caminho mínimo de  $s = v_0$  a  $v = v_k$ .
- ▶ Como  $v_0, v_1, \dots, v_k$  aparecem em ordem na ordenação topológica, as arestas  $(v_0, v_1), \dots, (v_{k-1}, v_k)$  são relaxadas em ordem.
- ▶ Logo, pela propriedade do relaxamento de caminho, o algoritmo computa corretamente  $d[v] = \text{dist}(s, v)$  para cada  $v \in V$ .

Também é fácil ver que  $\pi[\ ]$  define uma árvore de caminhos mínimos. (Por quê?)

- ▶ Seja  $v$  um vértice e suponha que  $p = (v_0, v_1, \dots, v_k)$  é um caminho mínimo de  $s = v_0$  a  $v = v_k$ .
- ▶ Como  $v_0, v_1, \dots, v_k$  aparecem em ordem na ordenação topológica, as arestas  $(v_0, v_1), \dots, (v_{k-1}, v_k)$  são relaxadas em ordem.
- ▶ Logo, pela propriedade do relaxamento de caminho, o algoritmo computa corretamente  $d[v] = \text{dist}(s, v)$  para cada  $v \in V$ .

Também é fácil ver que  $\pi[\ ]$  define uma árvore de caminhos mínimos. (Por quê?)

## Correção de DAG-SHORTEST-PATHS

- ▶ Seja  $v$  um vértice e suponha que  $p = (v_0, v_1, \dots, v_k)$  é um caminho mínimo de  $s = v_0$  a  $v = v_k$ .
- ▶ Como  $v_0, v_1, \dots, v_k$  aparecem em ordem na ordenação topológica, as arestas  $(v_0, v_1), \dots, (v_{k-1}, v_k)$  são relaxadas em ordem.
- ▶ Logo, pela propriedade do relaxamento de caminho, o algoritmo computa corretamente  $d[v] = \text{dist}(s, v)$  para cada  $v \in V$ .

Também é fácil ver que  $\pi[]$  define uma árvore de caminhos mínimos. (Por quê?)

- ▶ Seja  $v$  um vértice e suponha que  $p = (v_0, v_1, \dots, v_k)$  é um caminho mínimo de  $s = v_0$  a  $v = v_k$ .
- ▶ Como  $v_0, v_1, \dots, v_k$  aparecem em ordem na ordenação topológica, as arestas  $(v_0, v_1), \dots, (v_{k-1}, v_k)$  são relaxadas em ordem.
- ▶ Logo, pela propriedade do relaxamento de caminho, o algoritmo computa corretamente  $d[v] = \text{dist}(s, v)$  para cada  $v \in V$ .

Também é fácil ver que  $\pi[]$  define uma árvore de caminhos mínimos. (Por quê?)

1. Como se resolve o problema de encontrar um caminho de peso **máximo** de  $s$  a  $t$  em um grafo direcionado acíclico  $(G, \omega)$ ?
2. Como se resolve o problema do caminho mínimo de  $s$  a  $t$  em **tempo linear** para um grafo direcionado em que todas as arestas têm o mesmo peso  $C > 0$ ?

1. Como se resolve o problema de encontrar um caminho de peso **máximo** de  $s$  a  $t$  em um grafo direcionado acíclico  $(G, \omega)$ ?
2. Como se resolve o problema do caminho mínimo de  $s$  a  $t$  em **tempo linear** para um grafo direcionado em que todas as arestas têm o mesmo peso  $C > 0$ ?

## Algoritmo de Dijkstra

# Algoritmo de Dijkstra

Veremos agora um algoritmo para caminhos mínimos em grafos que podem conter ciclos, mas **sem arestas de pesos negativo**.

O algoritmo foi proposto por E.W. Dijkstra e é bastante similar ao algoritmo de Prim para o problema da árvore geradora mínima.

O algoritmo também é **baseado em relaxação**.

# Algoritmo de Dijkstra

Veremos agora um algoritmo para caminhos mínimos em grafos que podem conter ciclos, mas **sem arestas de pesos negativo**.

O algoritmo foi proposto por E.W. Dijkstra e é bastante similar ao algoritmo de Prim para o problema da **árvore geradora mínima**.

O algoritmo também é **baseado em relaxação**.

# Algoritmo de Dijkstra

Veremos agora um algoritmo para caminhos mínimos em grafos que podem conter ciclos, mas **sem arestas de pesos negativo**.

O algoritmo foi proposto por E.W. Dijkstra e é bastante similar ao algoritmo de Prim para o problema da **árvore geradora mínima**.

O algoritmo também é **baseado em relaxação**.

## Revisão: algoritmos baseados em relaxação

```
INITIALIZE-SINGLE-SOURCE( $G, s$ )  
1  para cada vértice  $v \in V[G]$  faça  
2     $d[v] \leftarrow \infty$   
3     $\pi[v] \leftarrow \text{NIL}$   
4     $d[s] \leftarrow 0$ 
```

- ▶  $d[v]$  é o comprimento do menor caminho conhecido até o momento
- ▶  $\pi[]$  induz uma árvore que testemunha  $d$

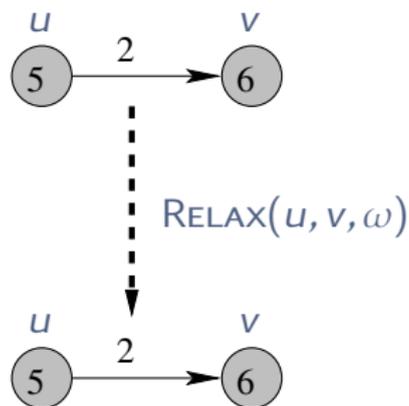
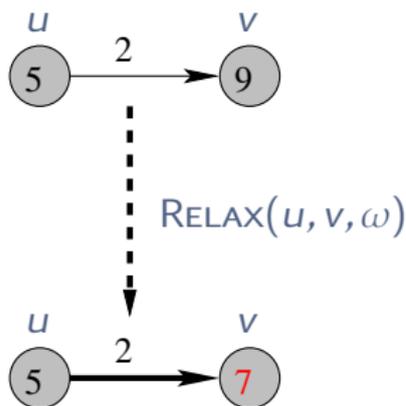
## Revisão: algoritmos baseados em relaxação

```
INITIALIZE-SINGLE-SOURCE( $G, s$ )  
1  para cada vértice  $v \in V[G]$  faça  
2     $d[v] \leftarrow \infty$   
3     $\pi[v] \leftarrow \text{NIL}$   
4     $d[s] \leftarrow 0$ 
```

- ▶  $d[v]$  é o comprimento do menor caminho conhecido até o momento
- ▶  $\pi[]$  induz uma árvore que testemunha  $d$

## Revisão: relaxação

Tenta melhorar a estimativa  $d[v]$  examinando  $(u, v)$ .

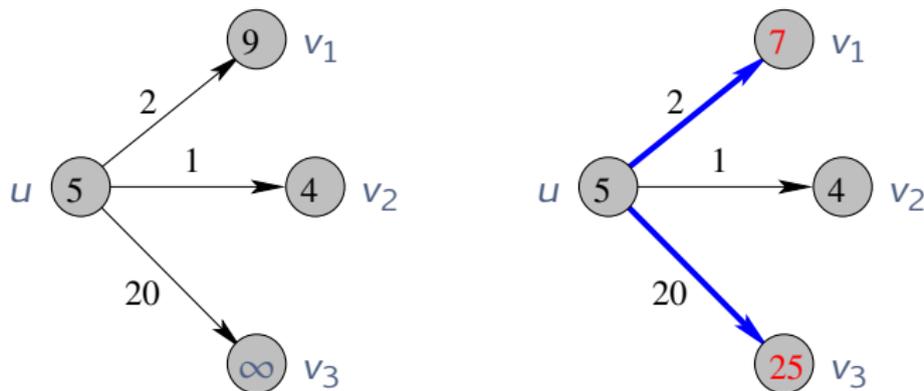


**RELAX** $(u, v, \omega)$

- 1 se  $d[v] > d[u] + \omega(u, v)$  então
- 2  $d[v] \leftarrow d[u] + \omega(u, v)$
- 3  $\pi[v] \leftarrow u$

## Revisão: relaxação dos vizinhos

Em cada iteração o algoritmo seleciona um vértice  $u$  e para cada vizinho  $v$  de  $u$  aplica  $\text{RELAX}(u, v, \omega)$ .



**RELAX**( $u, v, \omega$ )

- 1 se  $d[v] > d[u] + \omega(u, v)$  então
- 2  $d[v] \leftarrow d[u] + \omega(u, v)$
- 3  $\pi[v] \leftarrow u$

Os algoritmos de caminho mínimo baseados nas rotinas `INITIALIZE-SINGLE-SOURCE` e `RELAX` mantêm as propriedades:

- ▶ Limite superior
- ▶ Inexistência de caminho
- ▶ Subgrafo de predecessores
- ▶ Convergência
- ▶ Relaxamento de caminho

# Algoritmo de Dijkstra

## Entrada:

- ▶ grafo direcionado  $G = (V, E)$
- ▶ função de peso  $\omega$  nas arestas (sem arestas de peso negativo)
- ▶ origem  $s$ .

## Saída:

- ▶ vetor  $d$  com  $d[v] = \text{dist}(s, v)$  para  $v \in V$
- ▶ vetor  $\pi$  definindo uma **árvore de caminhos mínimos**

# Algoritmo de Dijkstra

**DIJKSTRA**( $G, \omega, s$ )

```
1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  $S \leftarrow \emptyset$ 
3  $Q \leftarrow V[G]$ 
4 enquanto  $Q \neq \emptyset$  faça
5      $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
6      $S \leftarrow S \cup \{u\}$ 
7     para cada vértice  $v \in \text{Adj}[u]$  faça
8         RELAX( $u, v, \omega$ )
9 devolva  $d, \pi$ 
```

- ▶ O conjunto  $Q$  é implementado como uma fila de prioridade com chave  $d$ .
- ▶ O conjunto  $S$  não é realmente necessário, mas simplifica a análise do algoritmo.

# Algoritmo de Dijkstra

**DIJKSTRA**( $G, \omega, s$ )

```
1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  $S \leftarrow \emptyset$ 
3  $Q \leftarrow V[G]$ 
4 enquanto  $Q \neq \emptyset$  faça
5      $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
6      $S \leftarrow S \cup \{u\}$ 
7     para cada vértice  $v \in \text{Adj}[u]$  faça
8         RELAX( $u, v, \omega$ )
9 devolva  $d, \pi$ 
```

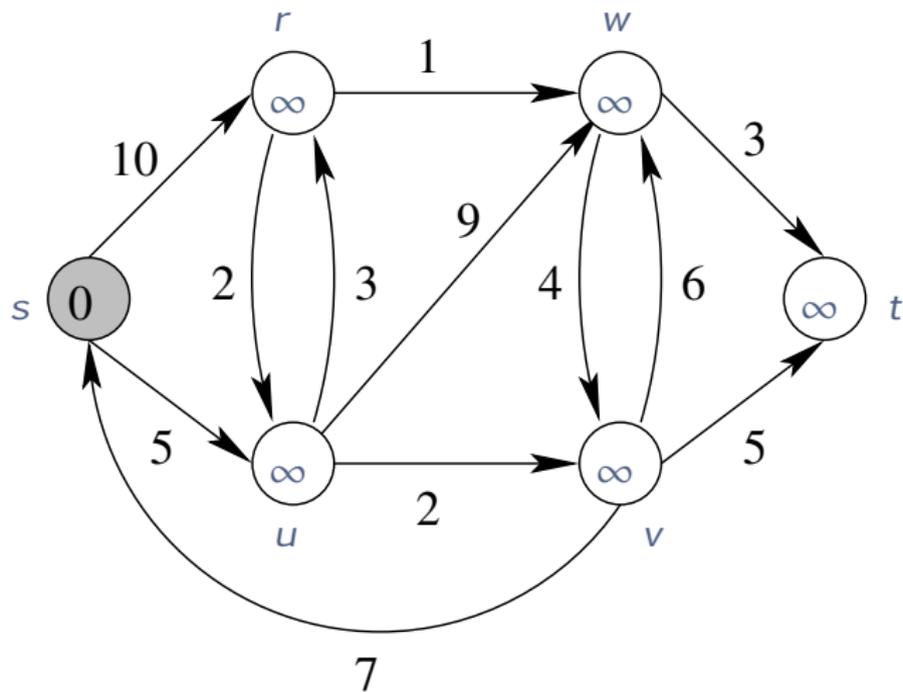
- ▶ O conjunto  $Q$  é implementado como uma fila de prioridade com chave  $d$ .
- ▶ O conjunto  $S$  não é realmente necessário, mas simplifica a análise do algoritmo.

# Algoritmo de Dijkstra

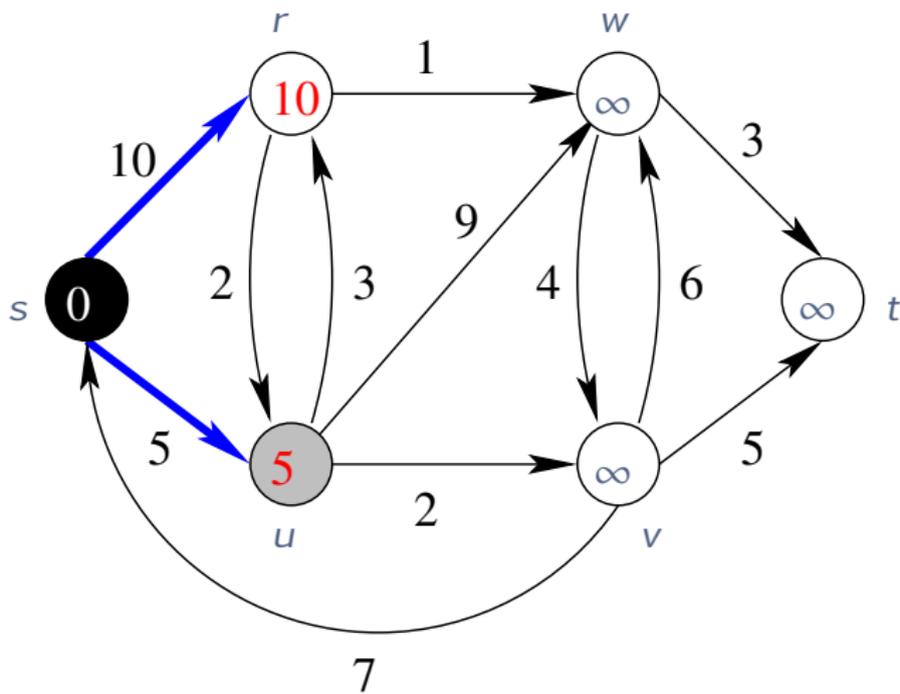
```
DIJKSTRA( $G, \omega, s$ )
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2   $S \leftarrow \emptyset$ 
3   $Q \leftarrow V[G]$ 
4  enquanto  $Q \neq \emptyset$  faça
5       $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
6       $S \leftarrow S \cup \{u\}$ 
7      para cada vértice  $v \in \text{Adj}[u]$  faça
8          RELAX( $u, v, \omega$ )
9  devolva  $d, \pi$ 
```

- ▶ O conjunto  $Q$  é implementado como uma fila de prioridade com chave  $d$ .
- ▶ O conjunto  $S$  não é realmente necessário, mas simplifica a análise do algoritmo.

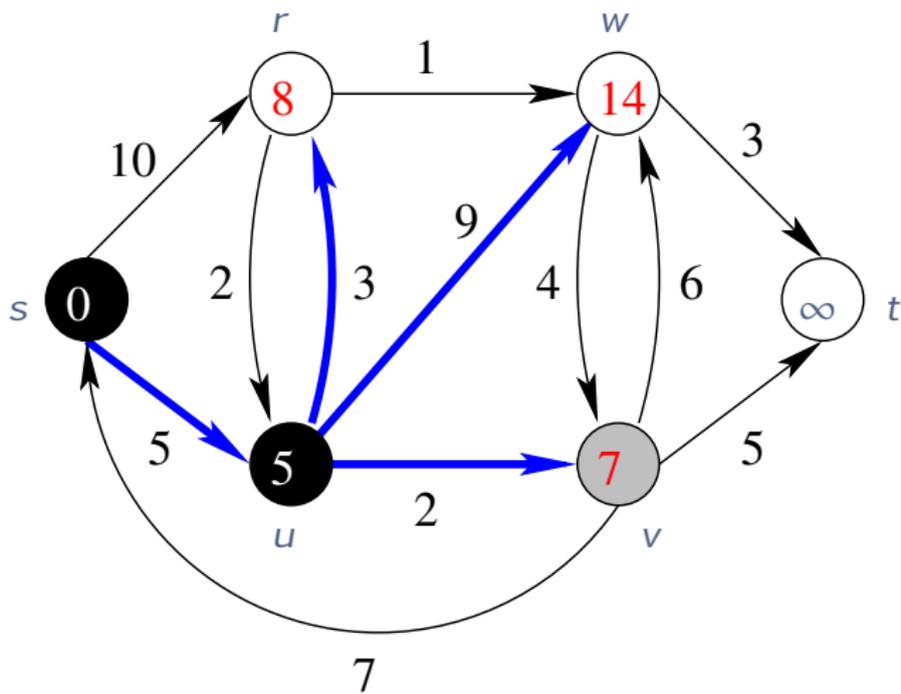
# Exemplo (CLRS modificado)



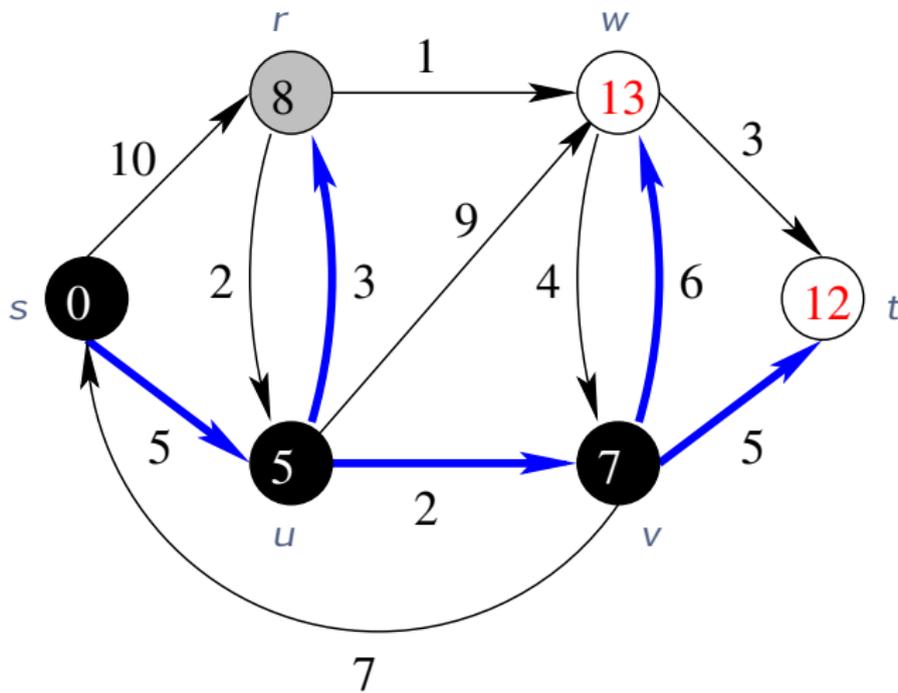
# Exemplo (CLRS modificado)



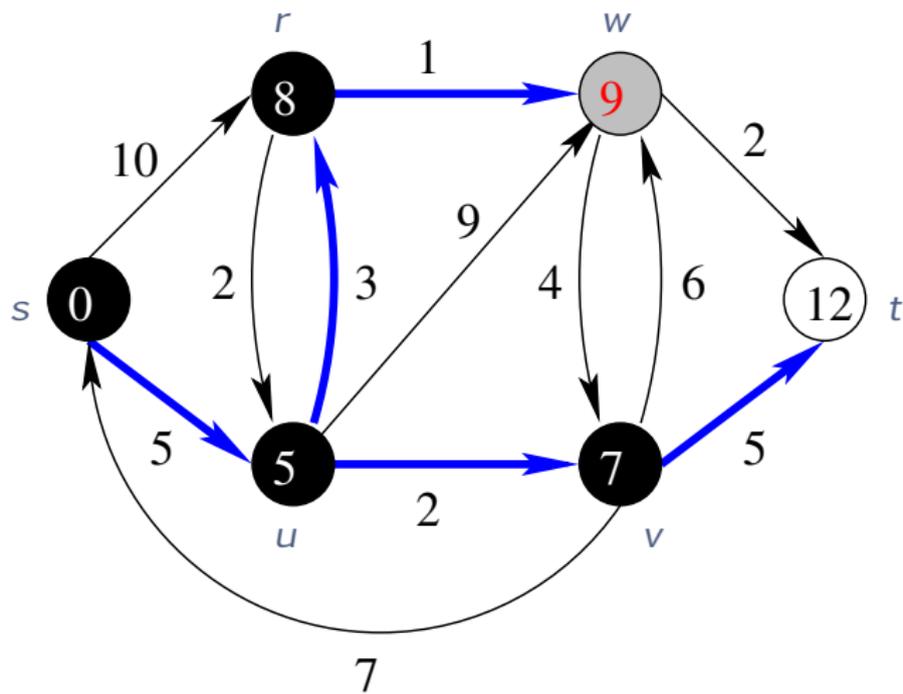
# Exemplo (CLRS modificado)



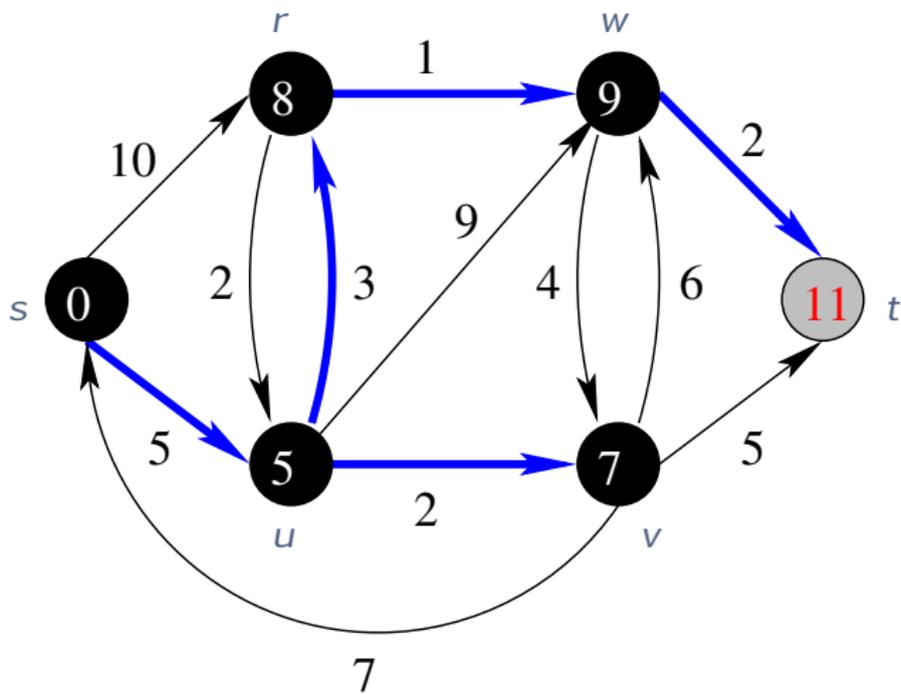
# Exemplo (CLRS modificado)



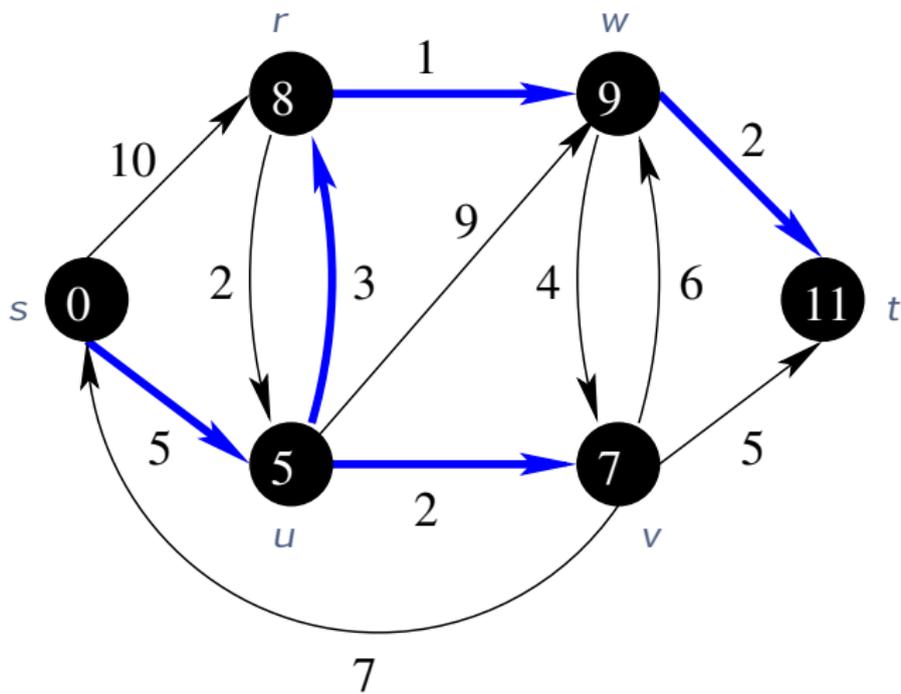
# Exemplo (CLRS modificado)



# Exemplo (CLRS modificado)



# Exemplo (CLRS modificado)



# Correção do algoritmo

Precisamos provar que algoritmo está correto, temos que mostrar que, quando o algoritmo termina

1.  $d[v] = \text{dist}(s, v)$  para todo  $v \in V[G]$  e
2.  $\pi[]$  induz uma **árvore de caminhos mínimos**.

Observe que

- ▶ DIJKSTRA é baseado em relaxação
- ▶ pela propriedade dos subgrafo de predecessores,  $\pi$  é uma árvore que testemunha  $d$
- ▶ assim, basta mostrar que de fato  $d[v] = \text{dist}(s, v)$

# Correção do algoritmo

Precisamos provar que algoritmo está correto, temos que mostrar que, quando o algoritmo termina

1.  $d[v] = \text{dist}(s, v)$  para todo  $v \in V[G]$  e
2.  $\pi[]$  induz uma **árvore de caminhos mínimos**.

Observe que

- ▶ DIJKSTRA é baseado em relaxação
- ▶ pela propriedade dos subgrafo de predecessores,  $\pi$  é uma árvore que testemunha  $d$
- ▶ assim, basta mostrar que de fato  $d[v] = \text{dist}(s, v)$

# Correção do algoritmo

Precisamos provar que algoritmo está correto, temos que mostrar que, quando o algoritmo termina

1.  $d[v] = \text{dist}(s, v)$  para todo  $v \in V[G]$  e
2.  $\pi[]$  induz uma **árvore de caminhos mínimos**.

Observe que

- ▶ DIJKSTRA é baseado em relaxação
- ▶ pela propriedade dos subgrafo de predecessores,  $\pi$  é uma árvore que testemunha  $d$
- ▶ assim, basta mostrar que de fato  $d[v] = \text{dist}(s, v)$

# Correção do algoritmo

Precisamos provar que algoritmo está correto, temos que mostrar que, quando o algoritmo termina

1.  $d[v] = \text{dist}(s, v)$  para todo  $v \in V[G]$  e
2.  $\pi[]$  induz uma **árvore de caminhos mínimos**.

Observe que

- ▶ DIJKSTRA é baseado em relaxação
- ▶ pela propriedade dos subgrafo de predecessores,  $\pi$  é uma árvore que testemunha  $d$
- ▶ assim, basta mostrar que de fato  $d[v] = \text{dist}(s, v)$

# Correção do algoritmo

Precisamos provar que algoritmo está correto, temos que mostrar que, quando o algoritmo termina

1.  $d[v] = \text{dist}(s, v)$  para todo  $v \in V[G]$  e
2.  $\pi[]$  induz uma **árvore de caminhos mínimos**.

Observe que

- ▶ DIJKSTRA é baseado em relaxação
- ▶ pela propriedade dos subgrafo de predecessores,  $\pi$  é uma árvore que testemunha  $d$
- ▶ assim, basta mostrar que de fato  $d[v] = \text{dist}(s, v)$

# Correção do algoritmo

Precisamos provar que algoritmo está correto, temos que mostrar que, quando o algoritmo termina

1.  $d[v] = \text{dist}(s, v)$  para todo  $v \in V[G]$  e
2.  $\pi[]$  induz uma **árvore de caminhos mínimos**.

Observe que

- ▶ DIJKSTRA é baseado em relaxação
- ▶ pela propriedade dos subgrafo de predecessores,  $\pi$  é uma árvore que testemunha  $d$
- ▶ assim, basta mostrar que de fato  $d[v] = \text{dist}(s, v)$

# Invariante

Iremos mostrar que no início de cada iteração da linha 4 no algoritmo DIJKSTRA, vale  $d[x] = \text{dist}(s, x)$  para cada  $x \in S$ .

- ▶ Inicialização

No início,  $S = \emptyset$ , então a invariante vale trivialmente.

- ▶ Manutenção

- ▶ suponha que a invariante vale para  $S$  no início da iteração
- ▶ nessa iteração, DIJKSTRA escolhe um vértice  $u$  com menor  $d[u]$  em  $Q$  e adiciona a  $S$
- ▶ queremos mostrar que a invariante vale para  $S \cup \{u\}$
- ▶ assim, basta verificar então que neste momento  $d[u] = \text{dist}(s, u)$

# Invariante

Iremos mostrar que no início de cada iteração da linha 4 no algoritmo DIJKSTRA, vale  $d[x] = \text{dist}(s, x)$  para cada  $x \in S$ .

- ▶ Inicialização

No início,  $S = \emptyset$ , então a invariante vale trivialmente.

- ▶ Manutenção

- ▶ suponha que a invariante vale para  $S$  no início da iteração
- ▶ nessa iteração, DIJKSTRA escolhe um vértice  $u$  com menor  $d[u]$  em  $Q$  e adiciona a  $S$
- ▶ queremos mostrar que a invariante vale para  $S \cup \{u\}$
- ▶ assim, basta verificar então que neste momento  $d[u] = \text{dist}(s, u)$

# Invariante

Iremos mostrar que no no início de cada iteração da linha 4 no algoritmo DIJKSTRA, vale  $d[x] = \text{dist}(s, x)$  para cada  $x \in S$ .

- ▶ **Inicialização**

No início,  $S = \emptyset$ , então a invariante vale trivialmente.

- ▶ **Manutenção**

- ▶ suponha que a invariante vale para  $S$  no início da iteração
- ▶ nessa iteração, DIJKSTRA escolhe um vértice  $u$  com menor  $d[u]$  em  $Q$  e adiciona a  $S$
- ▶ queremos mostrar que a invariante vale para  $S \cup \{u\}$
- ▶ assim, basta verificar então que neste momento  $d[u] = \text{dist}(s, u)$

# Invariante

Iremos mostrar que no início de cada iteração da linha 4 no algoritmo DIJKSTRA, vale  $d[x] = \text{dist}(s, x)$  para cada  $x \in S$ .

- ▶ **Inicialização**

No início,  $S = \emptyset$ , então a invariante vale trivialmente.

- ▶ **Manutenção**

- ▶ suponha que a invariante vale para  $S$  no início da iteração
- ▶ nessa iteração, DIJKSTRA escolhe um vértice  $u$  com menor  $d[u]$  em  $Q$  e adiciona a  $S$
- ▶ queremos mostrar que a invariante vale para  $S \cup \{u\}$
- ▶ assim, basta verificar então que neste momento  $d[u] = \text{dist}(s, u)$

Iremos mostrar que no início de cada iteração da linha 4 no algoritmo `DIJKSTRA`, vale  $d[x] = \text{dist}(s, x)$  para cada  $x \in S$ .

- ▶ **Inicialização**

No início,  $S = \emptyset$ , então a invariante vale trivialmente.

- ▶ **Manutenção**

- ▶ suponha que a invariante vale para  $S$  no início da iteração
- ▶ nessa iteração, `DIJKSTRA` escolhe um vértice  $u$  com menor  $d[u]$  em  $Q$  e adiciona a  $S$
- ▶ queremos mostrar que a invariante vale para  $S \cup \{u\}$
- ▶ assim, basta verificar então que neste momento  $d[u] = \text{dist}(s, u)$

# Invariante

Iremos mostrar que no início de cada iteração da linha 4 no algoritmo `DIJKSTRA`, vale  $d[x] = \text{dist}(s, x)$  para cada  $x \in S$ .

- ▶ **Inicialização**

No início,  $S = \emptyset$ , então a invariante vale trivialmente.

- ▶ **Manutenção**

- ▶ suponha que a invariante vale para  $S$  no início da iteração
- ▶ nessa iteração, `DIJKSTRA` escolhe um vértice  $u$  com menor  $d[u]$  em  $Q$  e adiciona a  $S$
- ▶ queremos mostrar que a invariante vale para  $S \cup \{u\}$
- ▶ assim, basta verificar então que neste momento  $d[u] = \text{dist}(s, u)$

# Invariante

Iremos mostrar que no início de cada iteração da linha 4 no algoritmo `DIJKSTRA`, vale  $d[x] = \text{dist}(s, x)$  para cada  $x \in S$ .

- ▶ **Inicialização**

No início,  $S = \emptyset$ , então a invariante vale trivialmente.

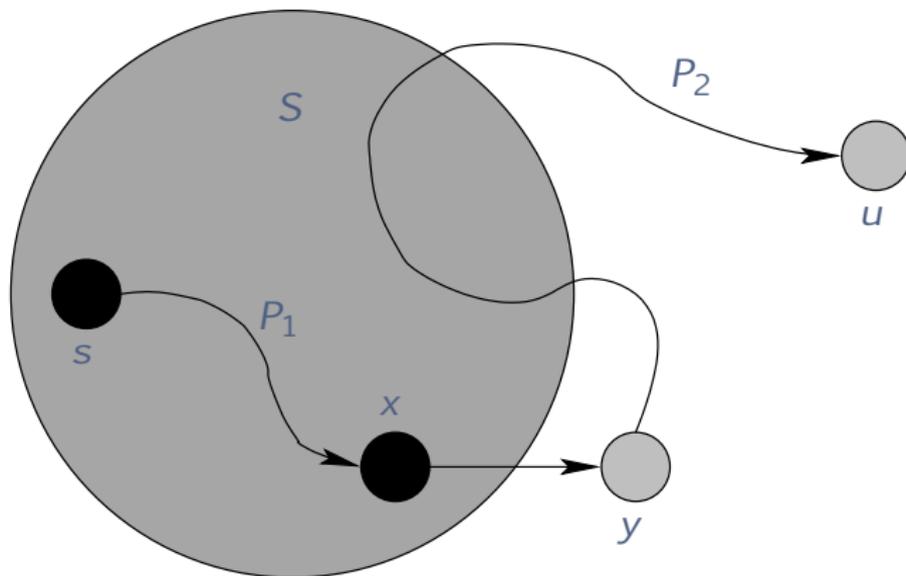
- ▶ **Manutenção**

- ▶ suponha que a invariante vale para  $S$  no início da iteração
- ▶ nessa iteração, `DIJKSTRA` escolhe um vértice  $u$  com menor  $d[u]$  em  $Q$  e adiciona a  $S$
- ▶ queremos mostrar que a invariante vale para  $S \cup \{u\}$
- ▶ assim, basta verificar então que neste momento  $d[u] = \text{dist}(s, u)$

# Demonstração

Seja

- ▶  $P$  um caminho mínimo de  $s$  a  $u$  (i.e., com peso  $\text{dist}(s, u)$ )
- ▶  $y$  o primeiro vértice de  $P$  que não pertence a  $S$
- ▶  $x$  o vértice em  $P$  que precede  $y$



# Demonstração

- ▶ Suponha que  $d[u] > \text{dist}(s, u)$  por contradição!
- ▶ Pela hipótese de indução,  $d[x] = \text{dist}(s, x)$  pois  $x \in S$ .
- ▶ Então

$$\begin{aligned}d[y] &\leq d[x] + \omega(x, y) \text{ (pois relaxamos } (x, y)) \\ &= \text{dist}(s, x) + \omega(x, y) \text{ (invariante)} \\ &\leq \text{dist}(s, x) + \omega(x, y) + \omega(P_2) \\ &= \omega(P_1) + \omega(x, y) + \omega(P_2) \\ &= \text{dist}(s, u) < d[u].\end{aligned}$$

- ▶ Mas daí  $d[y] < d[u]$ , o que contraria a escolha de  $u$ .
- ▶ Então, na verdade,  $d[u] \leq \text{dist}(s, u)$ .

Concluimos que  $d[u] = \text{dist}(s, u)$ , demonstrando a invariante.

# Demonstração

- ▶ Suponha que  $d[u] > \text{dist}(s, u)$  por contradição!
- ▶ Pela hipótese de indução,  $d[x] = \text{dist}(s, x)$  pois  $x \in S$ .
- ▶ Então

$$\begin{aligned}d[y] &\leq d[x] + \omega(x, y) \text{ (pois relaxamos } (x, y)\text{)} \\ &= \text{dist}(s, x) + \omega(x, y) \text{ (invariante)} \\ &\leq \text{dist}(s, x) + \omega(x, y) + \omega(P_2) \\ &= \omega(P_1) + \omega(x, y) + \omega(P_2) \\ &= \text{dist}(s, u) < d[u].\end{aligned}$$

- ▶ Mas daí  $d[y] < d[u]$ , o que contraria a escolha de  $u$ .
- ▶ Então, na verdade,  $d[u] \leq \text{dist}(s, u)$ .

Concluimos que  $d[u] = \text{dist}(s, u)$ , demonstrando a invariante.

# Demonstração

- ▶ Suponha que  $d[u] > \text{dist}(s, u)$  por contradição!
- ▶ Pela hipótese de indução,  $d[x] = \text{dist}(s, x)$  pois  $x \in S$ .
- ▶ Então

$$\begin{aligned}d[y] &\leq d[x] + \omega(x, y) \text{ (pois relaxamos } (x, y)) \\ &= \text{dist}(s, x) + \omega(x, y) \text{ (invariante)} \\ &\leq \text{dist}(s, x) + \omega(x, y) + \omega(P_2) \\ &= \omega(P_1) + \omega(x, y) + \omega(P_2) \\ &= \text{dist}(s, u) < d[u].\end{aligned}$$

- ▶ Mas daí  $d[y] < d[u]$ , o que contraria a escolha de  $u$ .
- ▶ Então, na verdade,  $d[u] \leq \text{dist}(s, u)$ .

Concluimos que  $d[u] = \text{dist}(s, u)$ , demonstrando a invariante.

# Demonstração

- ▶ Suponha que  $d[u] > \text{dist}(s, u)$  por contradição!
- ▶ Pela hipótese de indução,  $d[x] = \text{dist}(s, x)$  pois  $x \in S$ .
- ▶ Então

$$\begin{aligned}d[y] &\leq d[x] + \omega(x, y) \text{ (pois relaxamos } (x, y)) \\ &= \text{dist}(s, x) + \omega(x, y) \text{ (invariante)} \\ &\leq \text{dist}(s, x) + \omega(x, y) + \omega(P_2) \\ &= \omega(P_1) + \omega(x, y) + \omega(P_2) \\ &= \text{dist}(s, u) < d[u].\end{aligned}$$

- ▶ Mas daí  $d[y] < d[u]$ , o que contraria a escolha de  $u$ .
- ▶ Então, na verdade,  $d[u] \leq \text{dist}(s, u)$ .

Concluimos que  $d[u] = \text{dist}(s, u)$ , demonstrando a invariante.

# Demonstração

- ▶ Suponha que  $d[u] > \text{dist}(s, u)$  por contradição!
- ▶ Pela hipótese de indução,  $d[x] = \text{dist}(s, x)$  pois  $x \in S$ .
- ▶ Então

$$\begin{aligned}d[y] &\leq d[x] + \omega(x, y) \text{ (pois relaxamos } (x, y)) \\ &= \text{dist}(s, x) + \omega(x, y) \text{ (invariante)} \\ &\leq \text{dist}(s, x) + \omega(x, y) + \omega(P_2) \\ &= \omega(P_1) + \omega(x, y) + \omega(P_2) \\ &= \text{dist}(s, u) < d[u].\end{aligned}$$

- ▶ Mas daí  $d[y] < d[u]$ , o que contraria a escolha de  $u$ .
- ▶ Então, na verdade,  $d[u] \leq \text{dist}(s, u)$ .

Concluimos que  $d[u] = \text{dist}(s, u)$ , demonstrando a invariante.

# Demonstração

- ▶ Suponha que  $d[u] > \text{dist}(s, u)$  por contradição!
- ▶ Pela hipótese de indução,  $d[x] = \text{dist}(s, x)$  pois  $x \in S$ .
- ▶ Então

$$\begin{aligned}d[y] &\leq d[x] + \omega(x, y) \text{ (pois relaxamos } (x, y)) \\ &= \text{dist}(s, x) + \omega(x, y) \text{ (invariante)} \\ &\leq \text{dist}(s, x) + \omega(x, y) + \omega(P_2) \\ &= \omega(P_1) + \omega(x, y) + \omega(P_2) \\ &= \text{dist}(s, u) < d[u].\end{aligned}$$

- ▶ Mas daí  $d[y] < d[u]$ , o que contraria a escolha de  $u$ .
- ▶ Então, na verdade,  $d[u] \leq \text{dist}(s, u)$ .

Concluimos que  $d[u] = \text{dist}(s, u)$ , demonstrando a invariante.

# Demonstração

- ▶ Suponha que  $d[u] > \text{dist}(s, u)$  por contradição!
- ▶ Pela hipótese de indução,  $d[x] = \text{dist}(s, x)$  pois  $x \in S$ .
- ▶ Então

$$\begin{aligned}d[y] &\leq d[x] + \omega(x, y) \text{ (pois relaxamos } (x, y)) \\ &= \text{dist}(s, x) + \omega(x, y) \text{ (invariante)} \\ &\leq \text{dist}(s, x) + \omega(x, y) + \omega(P_2) \\ &= \omega(P_1) + \omega(x, y) + \omega(P_2) \\ &= \text{dist}(s, u) < d[u].\end{aligned}$$

- ▶ Mas daí  $d[y] < d[u]$ , o que contraria a escolha de  $u$ .
- ▶ Então, na verdade,  $d[u] \leq \text{dist}(s, u)$ .

Concluimos que  $d[u] = \text{dist}(s, u)$ , demonstrando a invariante.

# Demonstração

- ▶ Suponha que  $d[u] > \text{dist}(s, u)$  por contradição!
- ▶ Pela hipótese de indução,  $d[x] = \text{dist}(s, x)$  pois  $x \in S$ .
- ▶ Então

$$\begin{aligned}d[y] &\leq d[x] + \omega(x, y) \text{ (pois relaxamos } (x, y)) \\ &= \text{dist}(s, x) + \omega(x, y) \text{ (invariante)} \\ &\leq \text{dist}(s, x) + \omega(x, y) + \omega(P_2) \\ &= \omega(P_1) + \omega(x, y) + \omega(P_2) \\ &= \text{dist}(s, u) < d[u].\end{aligned}$$

- ▶ Mas daí  $d[y] < d[u]$ , o que contraria a escolha de  $u$ .
- ▶ Então, na verdade,  $d[u] \leq \text{dist}(s, u)$ .

Concluimos que  $d[u] = \text{dist}(s, u)$ , demonstrando a invariante.

# Demonstração

- ▶ Suponha que  $d[u] > \text{dist}(s, u)$  por contradição!
- ▶ Pela hipótese de indução,  $d[x] = \text{dist}(s, x)$  pois  $x \in S$ .
- ▶ Então

$$\begin{aligned}d[y] &\leq d[x] + \omega(x, y) \text{ (pois relaxamos } (x, y)) \\ &= \text{dist}(s, x) + \omega(x, y) \text{ (invariante)} \\ &\leq \text{dist}(s, x) + \omega(x, y) + \omega(P_2) \\ &= \omega(P_1) + \omega(x, y) + \omega(P_2) \\ &= \text{dist}(s, u) < d[u].\end{aligned}$$

- ▶ Mas daí  $d[y] < d[u]$ , o que contraria a escolha de  $u$ .
- ▶ Então, na verdade,  $d[u] \leq \text{dist}(s, u)$ .

Concluimos que  $d[u] = \text{dist}(s, u)$ , demonstrando a invariante.

# Demonstração

- ▶ Suponha que  $d[u] > \text{dist}(s, u)$  por contradição!
- ▶ Pela hipótese de indução,  $d[x] = \text{dist}(s, x)$  pois  $x \in S$ .
- ▶ Então

$$\begin{aligned}d[y] &\leq d[x] + \omega(x, y) \text{ (pois relaxamos } (x, y)) \\ &= \text{dist}(s, x) + \omega(x, y) \text{ (invariante)} \\ &\leq \text{dist}(s, x) + \omega(x, y) + \omega(P_2) \\ &= \omega(P_1) + \omega(x, y) + \omega(P_2) \\ &= \text{dist}(s, u) < d[u].\end{aligned}$$

- ▶ Mas daí  $d[y] < d[u]$ , o que contraria a escolha de  $u$ .
- ▶ Então, na verdade,  $d[u] \leq \text{dist}(s, u)$ .

Concluimos que  $d[u] = \text{dist}(s, u)$ , demonstrando a invariante.

# Demonstração

- ▶ Suponha que  $d[u] > \text{dist}(s, u)$  por contradição!
- ▶ Pela hipótese de indução,  $d[x] = \text{dist}(s, x)$  pois  $x \in S$ .
- ▶ Então

$$\begin{aligned}d[y] &\leq d[x] + \omega(x, y) \text{ (pois relaxamos } (x, y)) \\ &= \text{dist}(s, x) + \omega(x, y) \text{ (invariante)} \\ &\leq \text{dist}(s, x) + \omega(x, y) + \omega(P_2) \\ &= \omega(P_1) + \omega(x, y) + \omega(P_2) \\ &= \text{dist}(s, u) < d[u].\end{aligned}$$

- ▶ Mas daí  $d[y] < d[u]$ , o que contraria a escolha de  $u$ .
- ▶ Então, na verdade,  $d[u] \leq \text{dist}(s, u)$ .

Concluimos que  $d[u] = \text{dist}(s, u)$ , demonstrando a invariante.

# Demonstração

- ▶ Suponha que  $d[u] > \text{dist}(s, u)$  por contradição!
- ▶ Pela hipótese de indução,  $d[x] = \text{dist}(s, x)$  pois  $x \in S$ .
- ▶ Então

$$\begin{aligned}d[y] &\leq d[x] + \omega(x, y) \text{ (pois relaxamos } (x, y)) \\ &= \text{dist}(s, x) + \omega(x, y) \text{ (invariante)} \\ &\leq \text{dist}(s, x) + \omega(x, y) + \omega(P_2) \\ &= \omega(P_1) + \omega(x, y) + \omega(P_2) \\ &= \text{dist}(s, u) < d[u].\end{aligned}$$

- ▶ Mas daí  $d[y] < d[u]$ , o que contraria a escolha de  $u$ .
- ▶ Então, na verdade,  $d[u] \leq \text{dist}(s, u)$ .

Concluimos que  $d[u] = \text{dist}(s, u)$ , demonstrando a invariante.

Para terminar a demonstração, observe:

- ▶ Após a última iteração,  $S$  é o conjunto dos vértices atingíveis por  $s$ .
- ▶ Pela propriedade de inexistência de caminho, se um vértice  $v$  não é atingível, então  $d[v] = \infty$
- ▶ Portanto, para todo  $v \in V$ , vale  $d[v] = \text{dist}(s, v)$ .

# Demonstração

Para terminar a demonstração, observe:

- ▶ Após a última iteração,  $S$  é o conjunto dos vértices atingíveis por  $s$ .
- ▶ Pela propriedade de inexistência de caminho, se um vértice  $v$  não é atingível, então  $d[v] = \infty$
- ▶ Portanto, para todo  $v \in V$ , vale  $d[v] = \text{dist}(s, v)$ .

# Demonstração

Para terminar a demonstração, observe:

- ▶ Após a última iteração,  $S$  é o conjunto dos vértices atingíveis por  $s$ .
- ▶ Pela propriedade de inexistência de caminho, se um vértice  $v$  não é atingível, então  $d[v] = \infty$
- ▶ Portanto, para todo  $v \in V$ , vale  $d[v] = \text{dist}(s, v)$ .

# Demonstração

Para terminar a demonstração, observe:

- ▶ Após a última iteração,  $S$  é o conjunto dos vértices atingíveis por  $s$ .
- ▶ Pela propriedade de inexistência de caminho, se um vértice  $v$  não é atingível, então  $d[v] = \infty$
- ▶ Portanto, para todo  $v \in V$ , vale  $d[v] = \text{dist}(s, v)$ .

# Dijkstra precisa de arestas com peso não negativo

Note que na demonstração foi importante o fato de não haver arestas negativas no grafo.

De fato, não se pode garantir que o algoritmo de Dijkstra funciona se esta hipótese não for válida.

**Exercício.** Encontre um grafo direcionado ponderado com 4 vértices para o qual o algoritmo de Dijkstra **não** funciona. Há pelo menos um exemplo com apenas uma única aresta negativa e sem ciclos de peso negativo.

# Dijkstra precisa de arestas com peso não negativo

Note que na demonstração foi importante o fato de não haver arestas negativas no grafo.

De fato, não se pode garantir que o algoritmo de Dijkstra funciona se esta hipótese não for válida.

Exercício. Encontre um grafo direcionado ponderado com 4 vértices para o qual o algoritmo de Dijkstra **não** funciona. Há pelo menos um exemplo com apenas uma única aresta negativa e sem ciclos de peso negativo.

# Dijkstra precisa de arestas com peso não negativo

Note que na demonstração foi importante o fato de não haver arestas negativas no grafo.

De fato, não se pode garantir que o algoritmo de Dijkstra funciona se esta hipótese não for válida.

**Exercício.** Encontre um grafo direcionado ponderado com 4 vértices para o qual o algoritmo de Dijkstra **não** funciona. Há pelo menos um exemplo com apenas uma única aresta negativa e sem ciclos de peso negativo.

# Complexidade de tempo

```
DIJKSTRA( $G, \omega, s$ )  
1 INITIALIZE-SINGLE-SOURCE( $G, s$ )  
2  $S \leftarrow \emptyset$   
3  $Q \leftarrow V[G]$   
4 enquanto  $Q \neq \emptyset$  faça  
5      $u \leftarrow \text{EXTRACT-MIN}(Q)$   
6      $S \leftarrow S \cup \{u\}$   
7     para cada vértice  $v \in \text{Adj}[u]$  faça  
8         RELAX( $u, v, \omega$ )  
9 devolva  $d, \pi$ 
```

Depende de como a fila de prioridade  $Q$  é implementada:

- ▶ Operações INSERT, EXTRACT-MIN, DECREASE-KEY

# Complexidade de tempo

```
DIJKSTRA( $G, \omega, s$ )
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2   $S \leftarrow \emptyset$ 
3   $Q \leftarrow V[G]$ 
4  enquanto  $Q \neq \emptyset$  faça
5       $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
6       $S \leftarrow S \cup \{u\}$ 
7      para cada vértice  $v \in \text{Adj}[u]$  faça
8          RELAX( $u, v, \omega$ )
9  devolva  $d, \pi$ 
```

Depende de como a fila de prioridade  $Q$  é implementada:

- ▶ Operações INSERT, EXTRACT-MIN, DECREASE-KEY

# Complexidade do algoritmo de Dijkstra

- ▶ Os passos INITIALIZE-SINGLE-SOURCE e  $Q \leftarrow V[G]$  escondem chamadas a INSERT
- ▶ RELAX esconde chamada a DECREASE-KEY

Linha(s)	Tempo total
1-3	$ V $ chamadas a INSERT
5	$ V $ chamadas a EXTRACT-MIN
8	$ E $ chamadas a DECREASE-KEY

Complexidade de Dijkstra:

$$O(|V| \times \text{INSERT} + |V| \times \text{EXTRACT-MIN} + |E| \times \text{DECREASE-KEY})$$

# Complexidade do algoritmo de Dijkstra

- ▶ Os passos INITIALIZE-SINGLE-SOURCE e  $Q \leftarrow V[G]$  escondem chamadas a INSERT
- ▶ RELAX esconde chamada a DECREASE-KEY

Linha(s)	Tempo total
1-3	$ V $ chamadas a INSERT
5	$ V $ chamadas a EXTRACT-MIN
8	$ E $ chamadas a DECREASE-KEY

Complexidade de Dijkstra:

$$O(|V| \times \text{INSERT} + |V| \times \text{EXTRACT-MIN} + |E| \times \text{DECREASE-KEY})$$

# Complexidade do algoritmo de Dijkstra

- ▶ Os passos INITIALIZE-SINGLE-SOURCE e  $Q \leftarrow V[G]$  escondem chamadas a INSERT
- ▶ RELAX esconde chamada a DECREASE-KEY

Linha(s)	Tempo total
1-3	$ V $ chamadas a INSERT
5	$ V $ chamadas a EXTRACT-MIN
8	$ E $ chamadas a DECREASE-KEY

Complexidade de Dijkstra:

$$O(|V| \times \text{INSERT} + |V| \times \text{EXTRACT-MIN} + |E| \times \text{DECREASE-KEY})$$

# Complexidade de tempo

Total:  $O(|V| \times \text{INSERT} + |V| \times \text{EXTRACT-MIN} + |E| \times \text{DECREASE-KEY})$

Tipo de fila	INSERT	EXTRACT-MIN	DECREASE-KEY	TOTAL
<b>Vetor</b>	$O(1)$	$O(V)$	$O(1)$	$O(V^2)$
<b>Min-Heap</b>	$O(\lg V)$	$O(\lg V)$	$O(\lg V)$	$O((V + E) \lg V)$
<b>Fibonacci</b>	$O(1)$	$O(\lg V)$	$O(1)$	$O(V \lg V + E)$

## Algoritmo de Bellman-Ford

## Arestas vs. ciclos de peso negativo

- ▶ O algoritmo de Dijkstra resolve o Problema dos Caminhos Mínimos quando  $(G, \omega)$  **não tem arestas de peso negativo**.
- ▶ Quando  $(G, \omega)$  possui arestas negativas, o algoritmo de Dijkstra não funciona.
- ▶ Uma das dificuldades com arestas negativas é a possível existência de **ciclos de peso negativo** ou simplesmente ciclos negativos.

## Arestas vs. ciclos de peso negativo

- ▶ O algoritmo de Dijkstra resolve o Problema dos Caminhos Mínimos quando  $(G, \omega)$  **não tem arestas de peso negativo**.
- ▶ Quando  $(G, \omega)$  possui arestas negativas, o algoritmo de Dijkstra não funciona.
- ▶ Uma das dificuldades com arestas negativas é a possível existência de **ciclos de peso negativo** ou simplesmente ciclos negativos.

## Arestas vs. ciclos de peso negativo

- ▶ O algoritmo de Dijkstra resolve o Problema dos Caminhos Mínimos quando  $(G, \omega)$  **não tem arestas de peso negativo**.
- ▶ Quando  $(G, \omega)$  possui arestas negativas, o algoritmo de Dijkstra não funciona.
- ▶ Uma das dificuldades com arestas negativas é a possível existência de **ciclos de peso negativo** ou simplesmente ciclos negativos.

## Ciclos negativos — uma dificuldade

- ▶ O Problema dos Caminhos Mínimos para instâncias com ciclos negativos é **NP-difícil**.
  - ▶ Acreditamos que **não** existe algoritmo eficiente para resolver problemas NP-difíceis.
  - ▶ Assim, vamos nos restringir ao Problema de Caminhos Mínimos **sem ciclos negativos**.
- ▶ Um algoritmo que resolve o problema restrito é o algoritmo de Bellman-Ford, que também é baseado em relaxação.

## Ciclos negativos — uma dificuldade

- ▶ O Problema dos Caminhos Mínimos para instâncias com ciclos negativos é **NP-difícil**.
  - ▶ Acreditamos que **não** existe algoritmo eficiente para resolver problemas **NP-difíceis**.
  - ▶ Assim, vamos nos restringir ao Problema de Caminhos Mínimos **sem ciclos negativos**.
- ▶ Um algoritmo que resolve o problema restrito é o algoritmo de Bellman-Ford, que também é baseado em relaxação.

## Ciclos negativos — uma dificuldade

- ▶ O Problema dos Caminhos Mínimos para instâncias com ciclos negativos é **NP-difícil**.
  - ▶ Acreditamos que **não** existe algoritmo eficiente para resolver problemas **NP-difíceis**.
  - ▶ Assim, vamos nos restringir ao Problema de Caminhos Mínimos **sem ciclos negativos**.
- ▶ Um algoritmo que resolve o problema restrito é o algoritmo de Bellman-Ford, que também é baseado em relaxação.

## Ciclos negativos — uma dificuldade

- ▶ O Problema dos Caminhos Mínimos para instâncias com ciclos negativos é **NP-difícil**.
  - ▶ Acreditamos que **não** existe algoritmo eficiente para resolver problemas **NP-difíceis**.
  - ▶ Assim, vamos nos restringir ao Problema de Caminhos Mínimos **sem ciclos negativos**.
- ▶ Um algoritmo que resolve o problema restrito é o algoritmo de Bellman-Ford, que também é baseado em relaxação.

## Revisão: algoritmos baseados em relaxação

```
INITIALIZE-SINGLE-SOURCE( $G, s$ )  
1  para cada vértice  $v \in V[G]$  faça  
2     $d[v] \leftarrow \infty$   
3     $\pi[v] \leftarrow \text{NIL}$   
4     $d[s] \leftarrow 0$ 
```

- ▶  $d[v]$  é o comprimento do menor caminho conhecido até o momento
- ▶  $\pi[]$  induz uma árvore que testemunha  $d$

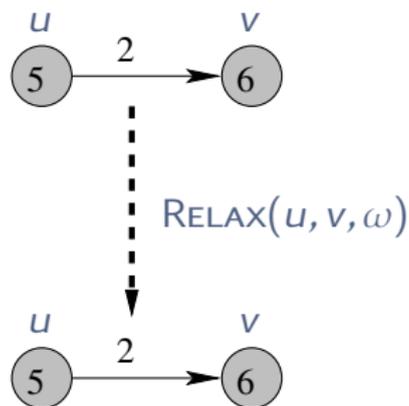
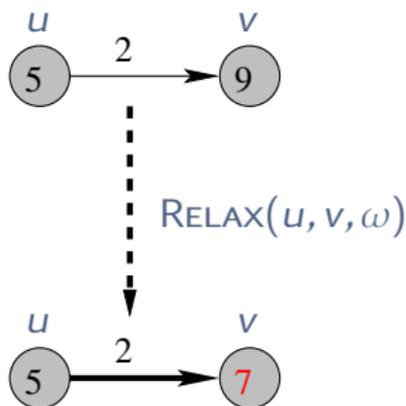
## Revisão: algoritmos baseados em relaxação

```
INITIALIZE-SINGLE-SOURCE( $G, s$ )  
1  para cada vértice  $v \in V[G]$  faça  
2     $d[v] \leftarrow \infty$   
3     $\pi[v] \leftarrow \text{NIL}$   
4     $d[s] \leftarrow 0$ 
```

- ▶  $d[v]$  é o comprimento do menor caminho conhecido até o momento
- ▶  $\pi[]$  induz uma árvore que testemunha  $d$

## Revisão: relaxação

Tenta melhorar a estimativa  $d[v]$  examinando  $(u, v)$ .

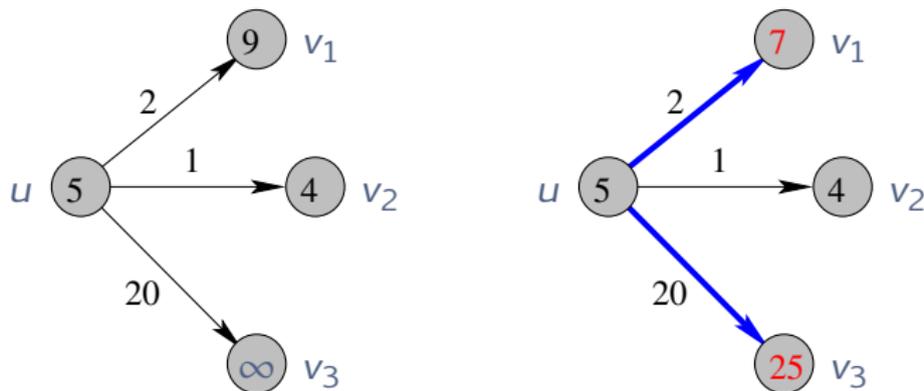


**RELAX** $(u, v, \omega)$

- 1 se  $d[v] > d[u] + \omega(u, v)$  então
- 2  $d[v] \leftarrow d[u] + \omega(u, v)$
- 3  $\pi[v] \leftarrow u$

## Revisão: relaxação dos vizinhos

Em cada iteração o algoritmo seleciona um vértice  $u$  e para cada vizinho  $v$  de  $u$  aplica  $\text{RELAX}(u, v, \omega)$ .



**RELAX**( $u, v, \omega$ )

- 1 se  $d[v] > d[u] + \omega(u, v)$  então
- 2  $d[v] \leftarrow d[u] + \omega(u, v)$
- 3  $\pi[v] \leftarrow u$

## Revisão: algoritmos baseados em relaxação

Os algoritmos de caminho mínimo baseados nas rotinas `INITIALIZE-SINGLE-SOURCE` e `RELAX` mantêm as propriedades:

- ▶ Limite superior
- ▶ Inexistência de caminho
- ▶ Subgrafo de predecessores
- ▶ Convergência
- ▶ Relaxamento de caminho

# Ideia do algoritmo de Bellman-Ford

**Relaxamento de caminho:** Para **qualquer** caminho mínimo  $(v_0, v_1, \dots, v_k)$ , queremos relaxar  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  em ordem

1. Executamos RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1)$  relaxada
2. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2)$  relaxadas em ordem
3. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2), (v_2, v_3)$  relaxadas em ordem
4. Repetimos esse passo até  $|V| - 1$  vezes. (Por quê?)  
 $\Rightarrow (v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  relaxadas em ordem

Podemos verificar se o grafo contém **ciclos negativos** executando mais uma vez:

- ▶ se algum valor  $d[v]$  diminuir, então há ciclo negativo

# Ideia do algoritmo de Bellman-Ford

**Relaxamento de caminho:** Para **qualquer** caminho mínimo  $(v_0, v_1, \dots, v_k)$ , queremos relaxar  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  em ordem

1. Executamos RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1)$  relaxada
2. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2)$  relaxadas em ordem
3. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2), (v_2, v_3)$  relaxadas em ordem
4. Repetimos esse passo até  $|V| - 1$  vezes. (Por quê?)  
 $\Rightarrow (v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  relaxadas em ordem

Podemos verificar se o grafo contém **ciclos negativos** executando mais uma vez:

- ▶ se algum valor  $d[v]$  diminuir, então há ciclo negativo

# Ideia do algoritmo de Bellman-Ford

**Relaxamento de caminho:** Para **qualquer** caminho mínimo  $(v_0, v_1, \dots, v_k)$ , queremos relaxar  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  em ordem

1. Executamos RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1)$  relaxada
2. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2)$  relaxadas em ordem
3. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2), (v_2, v_3)$  relaxadas em ordem
4. Repetimos esse passo até  $|V| - 1$  vezes. (Por quê?)  
 $\Rightarrow (v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  relaxadas em ordem

Podemos verificar se o grafo contém **ciclos negativos** executando mais uma vez:

- ▶ se algum valor  $d[v]$  diminuir, então há ciclo negativo

# Ideia do algoritmo de Bellman-Ford

**Relaxamento de caminho:** Para **qualquer** caminho mínimo  $(v_0, v_1, \dots, v_k)$ , queremos relaxar  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  em ordem

1. Executamos RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1)$  relaxada
2. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2)$  relaxadas em ordem
3. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2), (v_2, v_3)$  relaxadas em ordem
4. Repetimos esse passo até  $|V| - 1$  vezes. (Por quê?)  
 $\Rightarrow (v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  relaxadas em ordem

Podemos verificar se o grafo contém **ciclos negativos** executando mais uma vez:

- ▶ se algum valor  $d[v]$  diminuir, então há ciclo negativo

# Ideia do algoritmo de Bellman-Ford

**Relaxamento de caminho:** Para **qualquer** caminho mínimo  $(v_0, v_1, \dots, v_k)$ , queremos relaxar  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  em ordem

1. Executamos RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1)$  relaxada
2. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2)$  relaxadas em ordem
3. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2), (v_2, v_3)$  relaxadas em ordem
4. Repetimos esse passo até  $|V| - 1$  vezes. (Por quê?)  
 $\Rightarrow (v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  relaxadas em ordem

Podemos verificar se o grafo contém **ciclos negativos** executando mais uma vez:

- ▶ se algum valor  $d[v]$  diminuir, então há ciclo negativo

# Ideia do algoritmo de Bellman-Ford

**Relaxamento de caminho:** Para **qualquer** caminho mínimo  $(v_0, v_1, \dots, v_k)$ , queremos relaxar  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  em ordem

1. Executamos RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1)$  relaxada
2. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2)$  relaxadas em ordem
3. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2), (v_2, v_3)$  relaxadas em ordem
4. Repetimos esse passo até  $|V| - 1$  vezes. (Por quê?)  
 $\Rightarrow (v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  relaxadas em ordem

Podemos verificar se o grafo contém **ciclos negativos** executando mais uma vez:

- ▶ se algum valor  $d[v]$  diminuir, então há ciclo negativo

# Ideia do algoritmo de Bellman-Ford

**Relaxamento de caminho:** Para **qualquer** caminho mínimo  $(v_0, v_1, \dots, v_k)$ , queremos relaxar  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  em ordem

1. Executamos RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1)$  relaxada
2. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2)$  relaxadas em ordem
3. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2), (v_2, v_3)$  relaxadas em ordem
4. Repetimos esse passo até  $|V| - 1$  vezes. (Por quê?)  
 $\Rightarrow (v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  relaxadas em ordem

Podemos verificar se o grafo contém **ciclos negativos** executando mais uma vez:

- ▶ se algum valor  $d[v]$  diminuir, então há ciclo negativo

# Ideia do algoritmo de Bellman-Ford

**Relaxamento de caminho:** Para **qualquer** caminho mínimo  $(v_0, v_1, \dots, v_k)$ , queremos relaxar  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  em ordem

1. Executamos RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1)$  relaxada
2. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2)$  relaxadas em ordem
3. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2), (v_2, v_3)$  relaxadas em ordem
4. Repetimos esse passo até  $|V| - 1$  vezes. (Por quê?)  
 $\Rightarrow (v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  relaxadas em ordem

Podemos verificar se o grafo contém **ciclos negativos** executando mais uma vez:

- ▶ se algum valor  $d[v]$  diminuir, então há ciclo negativo

# Ideia do algoritmo de Bellman-Ford

**Relaxamento de caminho:** Para **qualquer** caminho mínimo  $(v_0, v_1, \dots, v_k)$ , queremos relaxar  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  em ordem

1. Executamos RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1)$  relaxada
2. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2)$  relaxadas em ordem
3. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2), (v_2, v_3)$  relaxadas em ordem
4. Repetimos esse passo até  $|V| - 1$  vezes. (Por quê?)  
 $\Rightarrow (v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  relaxadas em ordem

Podemos verificar se o grafo contém **ciclos negativos** executando mais uma vez:

- ▶ se algum valor  $d[v]$  diminuir, então há ciclo negativo

# Ideia do algoritmo de Bellman-Ford

**Relaxamento de caminho:** Para **qualquer** caminho mínimo  $(v_0, v_1, \dots, v_k)$ , queremos relaxar  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  em ordem

1. Executamos RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1)$  relaxada
2. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2)$  relaxadas em ordem
3. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2), (v_2, v_3)$  relaxadas em ordem
4. Repetimos esse passo até  $|V| - 1$  vezes. (Por quê?)  
 $\Rightarrow (v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  relaxadas em ordem

Podemos verificar se o grafo contém **ciclos negativos** executando mais uma vez:

- ▶ se algum valor  $d[v]$  diminuir, então há ciclo negativo

# Ideia do algoritmo de Bellman-Ford

**Relaxamento de caminho:** Para **qualquer** caminho mínimo  $(v_0, v_1, \dots, v_k)$ , queremos relaxar  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  em ordem

1. Executamos RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1)$  relaxada
2. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2)$  relaxadas em ordem
3. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2), (v_2, v_3)$  relaxadas em ordem
4. Repetimos esse passo até  $|V| - 1$  vezes. (Por quê?)  
 $\Rightarrow (v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  relaxadas em ordem

Podemos verificar se o grafo contém **ciclos negativos** executando mais uma vez:

- ▶ se algum valor  $d[v]$  diminuir, então há ciclo negativo

# Ideia do algoritmo de Bellman-Ford

**Relaxamento de caminho:** Para **qualquer** caminho mínimo  $(v_0, v_1, \dots, v_k)$ , queremos relaxar  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  em ordem

1. Executamos RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1)$  relaxada
2. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2)$  relaxadas em ordem
3. Executamos novamente RELAX para todas as arestas:  
 $\Rightarrow (v_0, v_1), (v_1, v_2), (v_2, v_3)$  relaxadas em ordem
4. Repetimos esse passo até  $|V| - 1$  vezes. (Por quê?)  
 $\Rightarrow (v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$  relaxadas em ordem

Podemos verificar se o grafo contém **ciclos negativos** executando mais uma vez:

- ▶ se algum valor  $d[v]$  diminuir, então há ciclo negativo

# O algoritmo de Bellman-Ford

## Entrada:

- ▶ grafo direcionado  $G = (V, E)$
- ▶ função de peso  $\omega$  nas arestas
- ▶ origem  $s$ .

## Saída: um valor booleano

- ▶ FALSE se existe um ciclo negativo atingível a partir de  $s$ , ou
- ▶ TRUE e neste caso devolve também
  - ▶ vetor  $d$  com  $d[v] = \text{dist}(s, v)$  para  $v \in V$
  - ▶ vetor  $\pi$  definindo uma **árvore de caminhos mínimos**

# O algoritmo de Bellman-Ford

## Entrada:

- ▶ grafo direcionado  $G = (V, E)$
- ▶ função de peso  $\omega$  nas arestas
- ▶ origem  $s$ .

## Saída: um valor booleano

- ▶ FALSE se existe um ciclo negativo atingível a partir de  $s$ , ou
- ▶ TRUE e neste caso devolve também
  - ▶ vetor  $d$  com  $d[v] = \text{dist}(s, v)$  para  $v \in V$
  - ▶ vetor  $\pi$  definindo uma **árvore de caminhos mínimos**

# O algoritmo de Bellman-Ford

## Entrada:

- ▶ grafo direcionado  $G = (V, E)$
- ▶ função de peso  $\omega$  nas arestas
- ▶ origem  $s$ .

## Saída: um valor booleano

- ▶ FALSE se existe um ciclo negativo atingível a partir de  $s$ , ou
- ▶ TRUE e neste caso devolve também
  - ▶ vetor  $d$  com  $d[v] = \text{dist}(s, v)$  para  $v \in V$
  - ▶ vetor  $\pi$  definindo uma **árvore de caminhos mínimos**

# O algoritmo de Bellman-Ford

## Entrada:

- ▶ grafo direcionado  $G = (V, E)$
- ▶ função de peso  $\omega$  nas arestas
- ▶ origem  $s$ .

## Saída: um valor booleano

- ▶ FALSE se existe um ciclo negativo atingível a partir de  $s$ , ou
- ▶ TRUE e neste caso devolve também
  - ▶ vetor  $d$  com  $d[v] = \text{dist}(s, v)$  para  $v \in V$
  - ▶ vetor  $\pi$  definindo uma **árvore de caminhos mínimos**

# O algoritmo de Bellman-Ford

## Entrada:

- ▶ grafo direcionado  $G = (V, E)$
- ▶ função de peso  $\omega$  nas arestas
- ▶ origem  $s$ .

## Saída: um valor booleano

- ▶ FALSE se existe um ciclo negativo atingível a partir de  $s$ , ou
- ▶ TRUE e neste caso devolve também
  - ▶ vetor  $d$  com  $d[v] = \text{dist}(s, v)$  para  $v \in V$
  - ▶ vetor  $\pi$  definindo uma **árvore de caminhos mínimos**

# O algoritmo de Bellman-Ford

```
BELLMAN-FORD( $G, \omega, s$ )  
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )  
2  para  $i \leftarrow 1$  até  $|V[G]| - 1$  faça  
3    para cada aresta  $(u, v) \in E[G]$  faça  
4      RELAX( $u, v, \omega$ )  
5  para cada aresta  $(u, v) \in E[G]$  faça  
6    se  $d[v] > d[u] + \omega(u, v)$   
7      então devolva FALSE  
8  devolva TRUE,  $d, \pi$ 
```

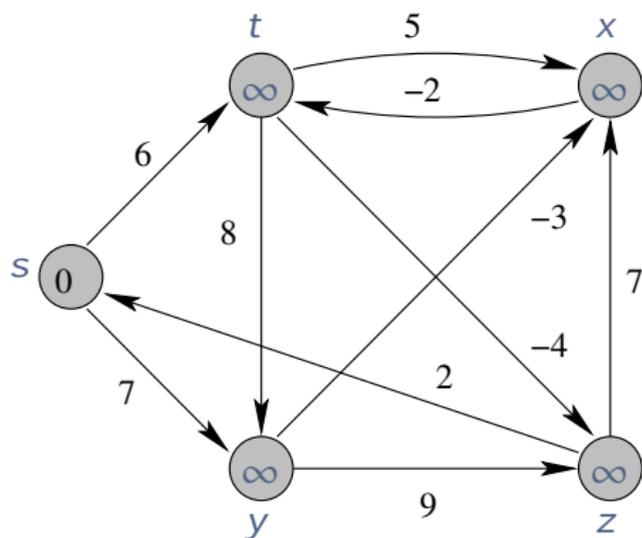
Complexidade de tempo:  $O(VE)$

# O algoritmo de Bellman-Ford

```
BELLMAN-FORD( $G, \omega, s$ )
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  para  $i \leftarrow 1$  até  $|V[G]| - 1$  faça
3    para cada aresta  $(u, v) \in E[G]$  faça
4      RELAX( $u, v, \omega$ )
5  para cada aresta  $(u, v) \in E[G]$  faça
6    se  $d[v] > d[u] + \omega(u, v)$ 
7      então devolva FALSE
8  devolva TRUE,  $d, \pi$ 
```

Complexidade de tempo:  $O(VE)$

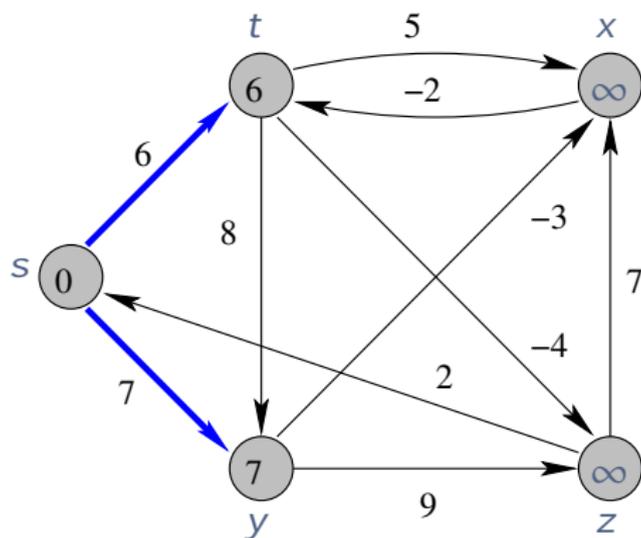
# Exemplo (CLRS)



Ordem:

$(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$ .

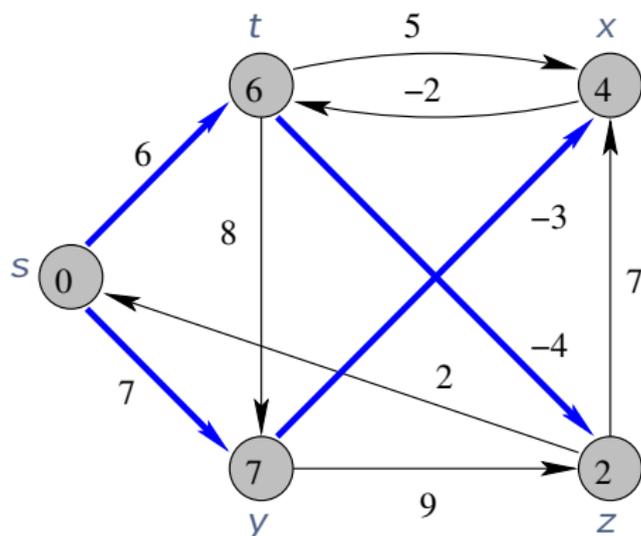
# Exemplo (CLRS)



Ordem:

$(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$ .

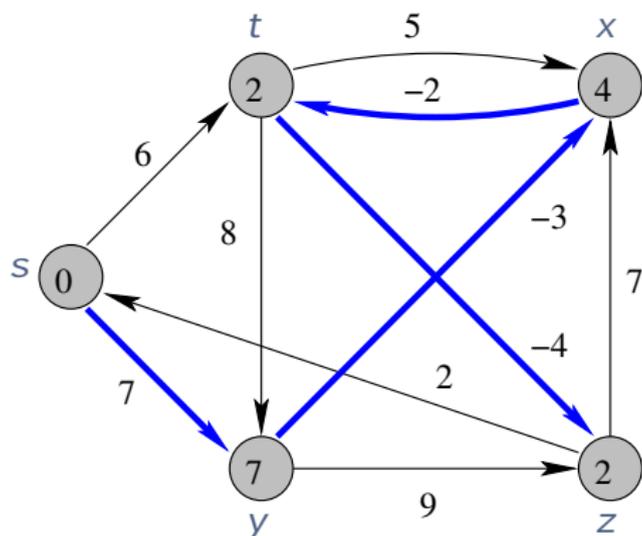
# Exemplo (CLRS)



Ordem:

$(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$ .

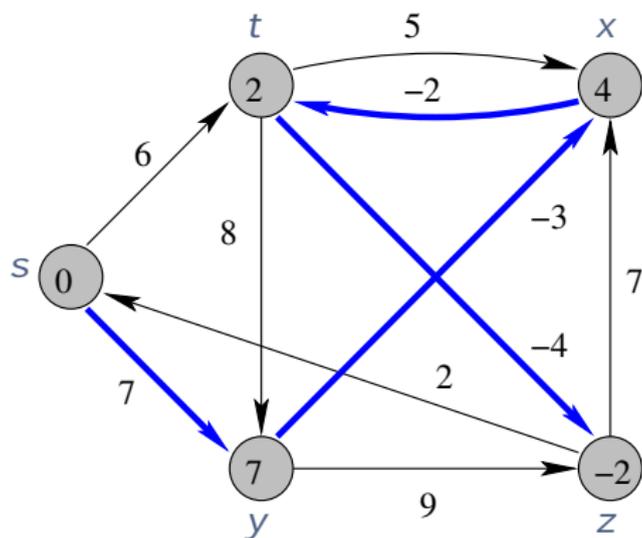
# Exemplo (CLRS)



Ordem:

$(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$ .

# Exemplo (CLRS)



Ordem:

$(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$ .

## Teorema

BELLMAN-FORD devolve:

- ▶ FALSE se existe um ciclo negativo atingível a partir de  $s$ ,
- ▶ TRUE caso contrário; neste caso devolve também
  - ▶ vetor  $d$  com  $d[v] = \text{dist}(s, v)$  para  $v \in V$
  - ▶ vetor  $\pi$  definindo uma **árvore de caminhos mínimos**

# Correção do algoritmo Bellman-Ford

Primeiro, suponha que o grafo não possui ciclos negativos atingíveis por  $s$ .

Considere  $v \in V[G]$  e os valores de  $d$  e  $\pi$  após o primeiro laço:

- ▶ se  $v$  não é atingível,  $d[v] = \infty$  por Inexistência de caminho
- ▶ senão, existe caminho mínimo  $(v_0, v_1, \dots, v_k)$  de  $s = v_0$  a  $v = v_k$ .
- ▶ como  $k \leq |V| - 1$ ,  $(v_0, v_1)$ , então  $(v_1, v_2), \dots, (v_{k-1}, v_k)$  foram relaxadas **nesta ordem**
- ▶ por Relaxamento de arestas,  $d[v] = \text{dist}(s, v)$
- ▶ também, por Sugrafo de predecessores,  $\pi$  induz um caminho mínimo de  $s$  a  $v$ .

# Correção do algoritmo Bellman-Ford

Primeiro, suponha que o grafo não possui ciclos negativos atingíveis por  $s$ .

Considere  $v \in V[G]$  e os valores de  $d$  e  $\pi$  após o primeiro laço:

- ▶ se  $v$  não é atingível,  $d[v] = \infty$  por Inexistência de caminho
- ▶ senão, existe caminho mínimo  $(v_0, v_1, \dots, v_k)$  de  $s = v_0$  a  $v = v_k$ .
- ▶ como  $k \leq |V| - 1$ ,  $(v_0, v_1)$ , então  $(v_1, v_2), \dots, (v_{k-1}, v_k)$  foram relaxadas **nesta ordem**
- ▶ por Relaxamento de arestas,  $d[v] = \text{dist}(s, v)$
- ▶ também, por Sugrafo de predecessores,  $\pi$  induz um caminho mínimo de  $s$  a  $v$ .

# Correção do algoritmo Bellman-Ford

Primeiro, suponha que o grafo não possui ciclos negativos atingíveis por  $s$ .

Considere  $v \in V[G]$  e os valores de  $d$  e  $\pi$  após o primeiro laço:

- ▶ se  $v$  não é atingível,  $d[v] = \infty$  por **Inexistência de caminho**
- ▶ senão, existe caminho mínimo  $(v_0, v_1, \dots, v_k)$  de  $s = v_0$  a  $v = v_k$ .
- ▶ como  $k \leq |V| - 1$ ,  $(v_0, v_1)$ , então  $(v_1, v_2), \dots, (v_{k-1}, v_k)$  foram relaxadas **nesta ordem**
- ▶ por **Relaxamento de arestas**,  $d[v] = \text{dist}(s, v)$
- ▶ também, por **Sugrafo de predecessores**,  $\pi$  induz um caminho mínimo de  $s$  a  $v$ .

# Correção do algoritmo Bellman-Ford

Primeiro, suponha que o grafo não possui ciclos negativos atingíveis por  $s$ .

Considere  $v \in V[G]$  e os valores de  $d$  e  $\pi$  após o primeiro laço:

- ▶ se  $v$  não é atingível,  $d[v] = \infty$  por **Inexistência de caminho**
- ▶ senão, existe caminho mínimo  $(v_0, v_1, \dots, v_k)$  de  $s = v_0$  a  $v = v_k$ .
- ▶ como  $k \leq |V| - 1$ ,  $(v_0, v_1)$ , então  $(v_1, v_2), \dots, (v_{k-1}, v_k)$  foram relaxadas **nesta ordem**
- ▶ por Relaxamento de arestas,  $d[v] = \text{dist}(s, v)$
- ▶ também, por Sugrafo de predecessores,  $\pi$  induz um caminho mínimo de  $s$  a  $v$ .

# Correção do algoritmo Bellman-Ford

Primeiro, suponha que o grafo não possui ciclos negativos atingíveis por  $s$ .

Considere  $v \in V[G]$  e os valores de  $d$  e  $\pi$  após o primeiro laço:

- ▶ se  $v$  não é atingível,  $d[v] = \infty$  por **Inexistência de caminho**
- ▶ senão, existe caminho mínimo  $(v_0, v_1, \dots, v_k)$  de  $s = v_0$  a  $v = v_k$ .
- ▶ como  $k \leq |V| - 1$ ,  $(v_0, v_1)$ , então  $(v_1, v_2), \dots, (v_{k-1}, v_k)$  foram relaxadas **nesta ordem**
- ▶ por Relaxamento de arestas,  $d[v] = \text{dist}(s, v)$
- ▶ também, por Sugrafo de predecessores,  $\pi$  induz um caminho mínimo de  $s$  a  $v$ .

# Correção do algoritmo Bellman-Ford

Primeiro, suponha que o grafo não possui ciclos negativos atingíveis por  $s$ .

Considere  $v \in V[G]$  e os valores de  $d$  e  $\pi$  após o primeiro laço:

- ▶ se  $v$  não é atingível,  $d[v] = \infty$  por **Inexistência de caminho**
- ▶ senão, existe caminho mínimo  $(v_0, v_1, \dots, v_k)$  de  $s = v_0$  a  $v = v_k$ .
- ▶ como  $k \leq |V| - 1$ ,  $(v_0, v_1)$ , então  $(v_1, v_2), \dots, (v_{k-1}, v_k)$  foram relaxadas **nesta ordem**
- ▶ por **Relaxamento de arestas**,  $d[v] = \text{dist}(s, v)$
- ▶ também, por **Sugrafo de predecessores**,  $\pi$  induz um caminho mínimo de  $s$  a  $v$ .

# Correção do algoritmo Bellman-Ford

Primeiro, suponha que o grafo não possui ciclos negativos atingíveis por  $s$ .

Considere  $v \in V[G]$  e os valores de  $d$  e  $\pi$  após o primeiro laço:

- ▶ se  $v$  não é atingível,  $d[v] = \infty$  por **Inexistência de caminho**
- ▶ senão, existe caminho mínimo  $(v_0, v_1, \dots, v_k)$  de  $s = v_0$  a  $v = v_k$ .
- ▶ como  $k \leq |V| - 1$ ,  $(v_0, v_1)$ , então  $(v_1, v_2), \dots, (v_{k-1}, v_k)$  foram relaxadas **nesta ordem**
- ▶ por **Relaxamento de arestas**,  $d[v] = \text{dist}(s, v)$
- ▶ também, por **Sugrafo de predecessores**,  $\pi$  induz um caminho mínimo de  $s$  a  $v$ .

# Correção do algoritmo Bellman-Ford

Também temos que mostrar que nesse caso **BELLMAN-FORD** devolve **TRUE**.

- ▶ considere  $d$  imediatamente após o primeiro laço
- ▶ nesse momento,  $d[v] = \text{dist}(s, v)$  para todo vértice  $v$
- ▶ por **Convergência**, sabemos que  $d$  nunca mais muda
- ▶ portanto, o teste da linha 6 falha sempre
- ▶ concluímos que o algoritmo devolve **TRUE**.

# Correção do algoritmo Bellman-Ford

Também temos que mostrar que nesse caso **BELLMAN-FORD** devolve **TRUE**.

- ▶ considere  $d$  imediatamente após o primeiro laço
- ▶ nesse momento,  $d[v] = \text{dist}(s, v)$  para todo vértice  $v$
- ▶ por **Convergência**, sabemos que  $d$  nunca mais muda
- ▶ portanto, o teste da linha 6 falha sempre
- ▶ concluímos que o algoritmo devolve **TRUE**.

# Correção do algoritmo Bellman-Ford

Também temos que mostrar que nesse caso `BELLMAN-FORD` devolve `TRUE`.

- ▶ considere  $d$  imediatamente após o primeiro laço
- ▶ nesse momento,  $d[v] = \text{dist}(s, v)$  para todo vértice  $v$
- ▶ por `Convergência`, sabemos que  $d$  nunca mais muda
- ▶ portanto, o teste da linha 6 falha sempre
- ▶ concluímos que o algoritmo devolve `TRUE`.

# Correção do algoritmo Bellman-Ford

Também temos que mostrar que nesse caso `BELLMAN-FORD` devolve `TRUE`.

- ▶ considere  $d$  imediatamente após o primeiro laço
- ▶ nesse momento,  $d[v] = \text{dist}(s, v)$  para todo vértice  $v$
- ▶ por **Convergência**, sabemos que  $d$  nunca mais muda
- ▶ portanto, o teste da linha 6 falha sempre
- ▶ concluímos que o algoritmo devolve `TRUE`.

# Correção do algoritmo Bellman-Ford

Também temos que mostrar que nesse caso `BELLMAN-FORD` devolve `TRUE`.

- ▶ considere  $d$  imediatamente após o primeiro laço
- ▶ nesse momento,  $d[v] = \text{dist}(s, v)$  para todo vértice  $v$
- ▶ por **Convergência**, sabemos que  $d$  nunca mais muda
- ▶ portanto, o teste da linha 6 falha sempre
- ▶ concluímos que o algoritmo devolve `TRUE`.

# Correção do algoritmo Bellman-Ford

Também temos que mostrar que nesse caso **BELLMAN-FORD** devolve **TRUE**.

- ▶ considere  $d$  imediatamente após o primeiro laço
- ▶ nesse momento,  $d[v] = \text{dist}(s, v)$  para todo vértice  $v$
- ▶ por **Convergência**, sabemos que  $d$  nunca mais muda
- ▶ portanto, o teste da linha 6 falha sempre
- ▶ concluímos que o algoritmo devolve **TRUE**.

# Correção do algoritmo Bellman-Ford

Suponha agora que  $(G, \omega)$  contém **ciclo negativo** alcançável por  $s$ .

Queremos mostrar que o algoritmo devolve FALSE:

- ▶ seja  $C = (v_0, v_1, \dots, v_k = v_0)$  um ciclo tal que

$$\omega(C) = \sum_{i=1}^k \omega(v_{i-1}, v_i) < 0$$

- ▶ suponha por contradição que o algoritmo devolve TRUE
- ▶ daí, como relaxamos cada aresta  $(v_{i-1}, v_i)$ ,

$$d[v_i] \leq d[v_{i-1}] + \omega(v_{i-1}, v_i)$$

# Correção do algoritmo Bellman-Ford

Suponha agora que  $(G, \omega)$  contém **ciclo negativo** alcançável por  $s$ .

Queremos mostrar que o algoritmo devolve FALSE:

- ▶ seja  $C = (v_0, v_1, \dots, v_k = v_0)$  um ciclo tal que

$$\omega(C) = \sum_{i=1}^k \omega(v_{i-1}, v_i) < 0$$

- ▶ suponha por contradição que o algoritmo devolve TRUE
- ▶ daí, como relaxamos cada aresta  $(v_{i-1}, v_i)$ ,

$$d[v_i] \leq d[v_{i-1}] + \omega(v_{i-1}, v_i)$$

# Correção do algoritmo Bellman-Ford

Suponha agora que  $(G, \omega)$  contém **ciclo negativo** alcançável por  $s$ .

Queremos mostrar que o algoritmo devolve FALSE:

- ▶ seja  $C = (v_0, v_1, \dots, v_k = v_0)$  um ciclo tal que

$$\omega(C) = \sum_{i=1}^k \omega(v_{i-1}, v_i) < 0$$

- ▶ suponha por contradição que o algoritmo devolve TRUE
- ▶ daí, como relaxamos cada aresta  $(v_{i-1}, v_i)$ ,

$$d[v_i] \leq d[v_{i-1}] + \omega(v_{i-1}, v_i)$$

# Correção do algoritmo Bellman-Ford

Suponha agora que  $(G, \omega)$  contém **ciclo negativo** alcançável por  $s$ .

Queremos mostrar que o algoritmo devolve **FALSE**:

- ▶ seja  $C = (v_0, v_1, \dots, v_k = v_0)$  um ciclo tal que

$$\omega(C) = \sum_{i=1}^k \omega(v_{i-1}, v_i) < 0$$

- ▶ suponha por contradição que o algoritmo devolve **TRUE**
- ▶ daí, como relaxamos cada aresta  $(v_{i-1}, v_i)$ ,

$$d[v_i] \leq d[v_{i-1}] + \omega(v_{i-1}, v_i)$$

# Correção do algoritmo Bellman-Ford

Suponha agora que  $(G, \omega)$  contém **ciclo negativo** alcançável por  $s$ .

Queremos mostrar que o algoritmo devolve FALSE:

- ▶ seja  $C = (v_0, v_1, \dots, v_k = v_0)$  um ciclo tal que

$$\omega(C) = \sum_{i=1}^k \omega(v_{i-1}, v_i) < 0$$

- ▶ suponha por contradição que o algoritmo devolve TRUE
- ▶ daí, como relaxamos cada aresta  $(v_{i-1}, v_i)$ ,

$$d[v_i] \leq d[v_{i-1}] + \omega(v_{i-1}, v_i)$$

# Correção do algoritmo Bellman-Ford

Suponha agora que  $(G, \omega)$  contém **ciclo negativo** alcançável por  $s$ .

Queremos mostrar que o algoritmo devolve FALSE:

- ▶ seja  $C = (v_0, v_1, \dots, v_k = v_0)$  um ciclo tal que

$$\omega(C) = \sum_{i=1}^k \omega(v_{i-1}, v_i) < 0$$

- ▶ suponha por contradição que o algoritmo devolve TRUE
- ▶ daí, como relaxamos cada aresta  $(v_{i-1}, v_i)$ ,

$$d[v_i] \leq d[v_{i-1}] + \omega(v_{i-1}, v_i)$$

# Correção do algoritmo Bellman-Ford

- ▶ Vamos somar as desigualdades anteriores para cada aresta do ciclo:

$$\begin{aligned}\sum_{i=1}^k d[v_i] &\leq \sum_{i=1}^k (d[v_{i-1}] + \omega(v_{i-1}, v_i)) \\ &= \sum_{i=1}^k d[v_{i-1}] + \sum_{i=1}^k \omega(v_{i-1}, v_i).\end{aligned}$$

- ▶ Como  $v_0 = v_k$ , temos que  $\sum_{i=1}^k d[v_i] = \sum_{i=1}^k d[v_{i-1}]$ .  
Daí,

$$0 \leq \sum_{i=1}^k \omega(v_{i-1}, v_i) = \omega(C).$$

- ▶ Mas isso é uma contradição, pois  $C$  é ciclo negativo.
- ▶ Concluimos que, de fato, o algoritmo devolve FALSE.

# Correção do algoritmo Bellman-Ford

- ▶ Vamos somar as desigualdades anteriores para cada aresta do ciclo:

$$\begin{aligned}\sum_{i=1}^k d[v_i] &\leq \sum_{i=1}^k (d[v_{i-1}] + \omega(v_{i-1}, v_i)) \\ &= \sum_{i=1}^k d[v_{i-1}] + \sum_{i=1}^k \omega(v_{i-1}, v_i).\end{aligned}$$

- ▶ Como  $v_0 = v_k$ , temos que  $\sum_{i=1}^k d[v_i] = \sum_{i=1}^k d[v_{i-1}]$ .  
Daí,

$$0 \leq \sum_{i=1}^k \omega(v_{i-1}, v_i) = \omega(C).$$

- ▶ Mas isso é uma contradição, pois  $C$  é ciclo negativo.
- ▶ Concluimos que, de fato, o algoritmo devolve FALSE.

# Correção do algoritmo Bellman-Ford

- ▶ Vamos somar as desigualdades anteriores para cada aresta do ciclo:

$$\begin{aligned}\sum_{i=1}^k d[v_i] &\leq \sum_{i=1}^k (d[v_{i-1}] + \omega(v_{i-1}, v_i)) \\ &= \sum_{i=1}^k d[v_{i-1}] + \sum_{i=1}^k \omega(v_{i-1}, v_i).\end{aligned}$$

- ▶ Como  $v_0 = v_k$ , temos que  $\sum_{i=1}^k d[v_i] = \sum_{i=1}^k d[v_{i-1}]$ .  
Daí,

$$0 \leq \sum_{i=1}^k \omega(v_{i-1}, v_i) = \omega(C).$$

- ▶ Mas isso é uma contradição, pois  $C$  é ciclo negativo.
- ▶ Concluimos que, de fato, o algoritmo devolve FALSE.

# Correção do algoritmo Bellman-Ford

- ▶ Vamos somar as desigualdades anteriores para cada aresta do ciclo:

$$\begin{aligned}\sum_{i=1}^k d[v_i] &\leq \sum_{i=1}^k (d[v_{i-1}] + \omega(v_{i-1}, v_i)) \\ &= \sum_{i=1}^k d[v_{i-1}] + \sum_{i=1}^k \omega(v_{i-1}, v_i).\end{aligned}$$

- ▶ Como  $v_0 = v_k$ , temos que  $\sum_{i=1}^k d[v_i] = \sum_{i=1}^k d[v_{i-1}]$ .  
Daí,

$$0 \leq \sum_{i=1}^k \omega(v_{i-1}, v_i) = \omega(C).$$

- ▶ Mas isso é uma contradição, pois  $C$  é ciclo negativo.
- ▶ Concluimos que, de fato, o algoritmo devolve FALSE.

# Correção do algoritmo Bellman-Ford

- ▶ Vamos somar as desigualdades anteriores para cada aresta do ciclo:

$$\begin{aligned}\sum_{i=1}^k d[v_i] &\leq \sum_{i=1}^k (d[v_{i-1}] + \omega(v_{i-1}, v_i)) \\ &= \sum_{i=1}^k d[v_{i-1}] + \sum_{i=1}^k \omega(v_{i-1}, v_i).\end{aligned}$$

- ▶ Como  $v_0 = v_k$ , temos que  $\sum_{i=1}^k d[v_i] = \sum_{i=1}^k d[v_{i-1}]$ .  
Daí,

$$0 \leq \sum_{i=1}^k \omega(v_{i-1}, v_i) = \omega(C).$$

- ▶ Mas isso é uma contradição, pois  $C$  é ciclo negativo.
- ▶ Concluimos que, de fato, o algoritmo devolve FALSE.

# Correção do algoritmo Bellman-Ford

- ▶ Vamos somar as desigualdades anteriores para cada aresta do ciclo:

$$\begin{aligned}\sum_{i=1}^k d[v_i] &\leq \sum_{i=1}^k (d[v_{i-1}] + \omega(v_{i-1}, v_i)) \\ &= \sum_{i=1}^k d[v_{i-1}] + \sum_{i=1}^k \omega(v_{i-1}, v_i).\end{aligned}$$

- ▶ Como  $v_0 = v_k$ , temos que  $\sum_{i=1}^k d[v_i] = \sum_{i=1}^k d[v_{i-1}]$ .  
Daí,

$$0 \leq \sum_{i=1}^k \omega(v_{i-1}, v_i) = \omega(C).$$

- ▶ Mas isso é uma contradição, pois  $C$  é ciclo negativo.
- ▶ Concluimos que, de fato, o algoritmo devolve FALSE.

# Correção do algoritmo Bellman-Ford

- ▶ Vamos somar as desigualdades anteriores para cada aresta do ciclo:

$$\begin{aligned}\sum_{i=1}^k d[v_i] &\leq \sum_{i=1}^k (d[v_{i-1}] + \omega(v_{i-1}, v_i)) \\ &= \sum_{i=1}^k d[v_{i-1}] + \sum_{i=1}^k \omega(v_{i-1}, v_i).\end{aligned}$$

- ▶ Como  $v_0 = v_k$ , temos que  $\sum_{i=1}^k d[v_i] = \sum_{i=1}^k d[v_{i-1}]$ .  
Daí,

$$0 \leq \sum_{i=1}^k \omega(v_{i-1}, v_i) = \omega(C).$$

- ▶ Mas isso é uma contradição, pois  $C$  é ciclo negativo.
- ▶ Concluimos que, de fato, o algoritmo devolve FALSE.

# Correção do algoritmo Bellman-Ford

- ▶ Vamos somar as desigualdades anteriores para cada aresta do ciclo:

$$\begin{aligned}\sum_{i=1}^k d[v_i] &\leq \sum_{i=1}^k (d[v_{i-1}] + \omega(v_{i-1}, v_i)) \\ &= \sum_{i=1}^k d[v_{i-1}] + \sum_{i=1}^k \omega(v_{i-1}, v_i).\end{aligned}$$

- ▶ Como  $v_0 = v_k$ , temos que  $\sum_{i=1}^k d[v_i] = \sum_{i=1}^k d[v_{i-1}]$ .  
Daí,

$$0 \leq \sum_{i=1}^k \omega(v_{i-1}, v_i) = \omega(C).$$

- ▶ Mas isso é uma contradição, pois  $C$  é ciclo negativo.
- ▶ Concluimos que, de fato, o algoritmo devolve **FALSE**.

## Sistemas de restrições de diferenças

# Sistemas de restrições de diferenças

Uma aplicação de caminhos mínimos é encontrar  $x_1, x_2, \dots, x_n$  que satisfaçam:

$$x_1 - x_2 \leq 0$$

$$x_1 - x_5 \leq -1$$

$$x_2 - x_5 \leq 1$$

$$x_3 - x_1 \leq 5$$

$$x_4 - x_1 \leq 4$$

$$x_4 - x_3 \leq -1$$

$$x_5 - x_3 \leq -3$$

$$x_5 - x_4 \leq -3$$

Exemplo:

- ▶  $x_i$  representa a hora do evento  $i$
- ▶  $x_j - x_i \leq b_k$  significa que deve haver um intervalo de pelo menos  $b_k$  horas entre eventos  $i$  e  $j$

# Sistemas de restrições de diferenças

Uma aplicação de caminhos mínimos é encontrar  $x_1, x_2, \dots, x_n$  que satisfaçam:

$$x_1 - x_2 \leq 0$$

$$x_1 - x_5 \leq -1$$

$$x_2 - x_5 \leq 1$$

$$x_3 - x_1 \leq 5$$

$$x_4 - x_1 \leq 4$$

$$x_4 - x_3 \leq -1$$

$$x_5 - x_3 \leq -3$$

$$x_5 - x_4 \leq -3$$

Exemplo:

- ▶  $x_i$  representa a hora do evento  $i$
- ▶  $x_j - x_i \leq b_k$  significa que deve haver um intervalo de pelo menos  $b_k$  horas entre eventos  $i$  e  $j$

# Sistemas de restrições de diferenças

Uma aplicação de caminhos mínimos é encontrar  $x_1, x_2, \dots, x_n$  que satisfaçam:

$$x_1 - x_2 \leq 0$$

$$x_1 - x_5 \leq -1$$

$$x_2 - x_5 \leq 1$$

$$x_3 - x_1 \leq 5$$

$$x_4 - x_1 \leq 4$$

$$x_4 - x_3 \leq -1$$

$$x_5 - x_3 \leq -3$$

$$x_5 - x_4 \leq -3$$

Exemplo:

- ▶  $x_i$  representa a hora do evento  $i$
- ▶  $x_j - x_i \leq b_k$  significa que deve haver um intervalo de pelo menos  $b_k$  horas entre eventos  $i$  e  $j$

# Sistemas de restrições de diferenças

Uma aplicação de caminhos mínimos é encontrar  $x_1, x_2, \dots, x_n$  que satisfaçam:

$$x_1 - x_2 \leq 0$$

$$x_1 - x_5 \leq -1$$

$$x_2 - x_5 \leq 1$$

$$x_3 - x_1 \leq 5$$

$$x_4 - x_1 \leq 4$$

$$x_4 - x_3 \leq -1$$

$$x_5 - x_3 \leq -3$$

$$x_5 - x_4 \leq -3$$

Exemplo:

- ▶  $x_i$  representa a hora do evento  $i$
- ▶  $x_j - x_i \leq b_k$  significa que deve haver um intervalo de pelo menos  $b_k$  horas entre eventos  $i$  e  $j$

# Sistemas de restrições de diferenças

Podemos reescrever as restrições de forma matricial:

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}$$

Algumas soluções:

- ▶  $x = (-5, -3, 0, -1, -4)$ ,
- ▶  $x' = (0, 2, 5, 4, 1), \dots$

# Sistemas de restrições de diferenças

Podemos reescrever as restrições de forma matricial:

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}$$

Algumas soluções:

- ▶  $x = (-5, -3, 0, -1, -4)$ ,
- ▶  $x' = (0, 2, 5, 4, 1), \dots$

# Sistemas de restrições de diferenças

Podemos reescrever as restrições de forma matricial:

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}$$

Algumas soluções:

- ▶  $x = (-5, -3, 0, -1, -4)$ ,
- ▶  $x' = (0, 2, 5, 4, 1), \dots$

# Sistemas de restrições de diferenças

Podemos reescrever as restrições de forma matricial:

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}$$

Algumas soluções:

- ▶  $x = (-5, -3, 0, -1, -4)$ ,
- ▶  $x' = (0, 2, 5, 4, 1), \dots$

## Lema

Seja  $x = (x_1, \dots, x_n)$  uma solução de um sistema de restrições de diferença  $Ax \leq b$  e  $d$  uma constante. Então

$$x + d = (x_1 + d, \dots, x_n + d)$$

também é uma solução de  $Ax \leq b$ .

Mostraremos a seguir como encontrar uma solução de um sistema  $Ax \leq b$  de restrições de diferença resolvendo um problema de caminhos mínimos.

# Sistemas de restrições de diferenças

## Lema

Seja  $x = (x_1, \dots, x_n)$  uma solução de um sistema de restrições de diferença  $Ax \leq b$  e  $d$  uma constante. Então

$$x + d = (x_1 + d, \dots, x_n + d)$$

também é uma solução de  $Ax \leq b$ .

Mostraremos a seguir como encontrar uma solução de um sistema  $Ax \leq b$  de restrições de diferença resolvendo um problema de caminhos mínimos.

# Grafo de restrições

Construímos o chamado **grafo de restrições** fazendo o seguinte:

1. Primeiro criamos um grafo em que:
  - ▶ cada vértice  $v_i$  corresponde a uma variável  $x_i$
  - ▶ cada aresta  $(v_i, v_j)$  tem peso  $b_k$  e corresponde a uma restrição  $x_j - x_i \leq b_k$ 
    - ⇒ A matriz  $A$  do sistema de restrições corresponde à transposta matriz de incidência desse grafo obtido
2. Agora adicionamos um vértice especial  $v_0$  e uma aresta de  $v_0$  a cada outro vértice  $v_i$  com peso 0.

Construímos o chamado **grafo de restrições** fazendo o seguinte:

1. Primeiro criamos um grafo em que:
  - ▶ cada vértice  $v_i$  corresponde a uma variável  $x_i$
  - ▶ cada aresta  $(v_i, v_j)$  tem peso  $b_k$  e corresponde a uma restrição  $x_j - x_i \leq b_k$ 
    - ⇒ A matriz  $A$  do sistema de restrições corresponde à transposta matriz de incidência desse grafo obtido
2. Agora adicionamos um vértice especial  $v_0$  e uma aresta de  $v_0$  a cada outro vértice  $v_i$  com peso 0.

# Grafo de restrições

Construímos o chamado **grafo de restrições** fazendo o seguinte:

1. Primeiro criamos um grafo em que:
  - ▶ cada vértice  $v_i$  corresponde a uma variável  $x_i$
  - ▶ cada aresta  $(v_i, v_j)$  tem peso  $b_k$  e corresponde a uma restrição  $x_j - x_i \leq b_k$ 
    - ⇒ A matriz  $A$  do sistema de restrições corresponde à transposta matriz de incidência desse grafo obtido
2. Agora adicionamos um vértice especial  $v_0$  e uma aresta de  $v_0$  a cada outro vértice  $v_i$  com peso 0.

# Grafo de restrições

Construímos o chamado **grafo de restrições** fazendo o seguinte:

1. Primeiro criamos um grafo em que:
  - ▶ cada vértice  $v_i$  corresponde a uma variável  $x_i$
  - ▶ cada aresta  $(v_i, v_j)$  tem peso  $b_k$  e corresponde a uma restrição  $x_j - x_i \leq b_k$ 
    - ⇒ A matriz  $A$  do sistema de restrições corresponde à transposta matriz de incidência desse grafo obtido
2. Agora adicionamos um vértice especial  $v_0$  e uma aresta de  $v_0$  a cada outro vértice  $v_i$  com peso 0.

# Grafo de restrições

Construímos o chamado **grafo de restrições** fazendo o seguinte:

1. Primeiro criamos um grafo em que:
  - ▶ cada vértice  $v_i$  corresponde a uma variável  $x_i$
  - ▶ cada aresta  $(v_i, v_j)$  tem peso  $b_k$  e corresponde a uma restrição  $x_j - x_i \leq b_k$ 
    - ⇒ A matriz  $A$  do sistema de restrições corresponde à transposta matriz de incidência desse grafo obtido
2. Agora adicionamos um vértice especial  $v_0$  e uma aresta de  $v_0$  a cada outro vértice  $v_i$  com peso 0.

# Grafo de restrições

Construímos o chamado **grafo de restrições** fazendo o seguinte:

1. Primeiro criamos um grafo em que:
  - ▶ cada vértice  $v_i$  corresponde a uma variável  $x_i$
  - ▶ cada aresta  $(v_i, v_j)$  tem peso  $b_k$  e corresponde a uma restrição  $x_j - x_i \leq b_k$ 
    - ⇒ A matriz  $A$  do sistema de restrições corresponde à transposta matriz de incidência desse grafo obtido
2. Agora adicionamos um vértice especial  $v_0$  e uma aresta de  $v_0$  a cada outro vértice  $v_i$  com peso 0.

# Grafo de restrições

Formalmente, dado o sistema  $Ax \leq b$  de restrições de diferença, construímos o grafo  $G = (V, E)$  tal que

- ▶  $V = \{v_0, v_1, \dots, v_n\}$
- ▶  $E = \{(v_i, v_j) : x_j - x_i \leq b_k \text{ é restrição}\} \cup \{(v_0, v_1), \dots, (v_0, v_n)\}$

E associamos pesos:

- ▶  $\omega(v_i, v_j) = \begin{cases} b_k & \text{se } x_j - x_i \leq b_k \text{ é restrição} \\ 0 & \text{se } i = 0 \end{cases}$

# Grafo de restrições

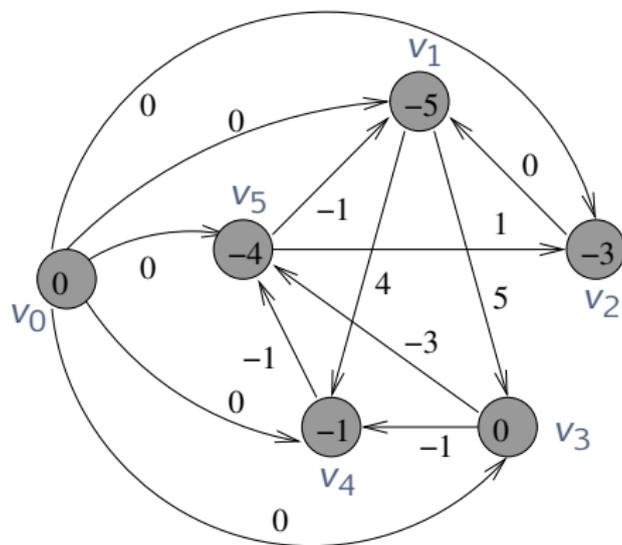
Formalmente, dado o sistema  $Ax \leq b$  de restrições de diferença, construímos o grafo  $G = (V, E)$  tal que

- ▶  $V = \{v_0, v_1, \dots, v_n\}$
- ▶  $E = \{(v_i, v_j) : x_j - x_i \leq b_k \text{ é restrição}\} \cup \{(v_0, v_1), \dots, (v_0, v_n)\}$

E associamos pesos:

- ▶  $\omega(v_i, v_j) = \begin{cases} b_k & \text{se } x_j - x_i \leq b_k \text{ é restrição} \\ 0 & \text{se } i = 0 \end{cases}$

# Grafo de restrições



Uma solução viável é  $x = (-5, -3, -0, -1, -4)$ .

## Teorema

Seja  $Ax \leq b$  um sistema de restrições de diferença e  $G = (V, E)$  o grafo de restrições associado. Então:

- ▶ se  $G$  não contém ciclos negativos, então

$$x = (\text{dist}(v_0, v_1), \text{dist}(v_0, v_2), \dots, \text{dist}(v_0, v_n))$$

é uma solução viável do sistema;

- ▶ se  $G$  contém ciclos negativos, então o sistema não possui solução viável.

# Resolvendo um sistema de restrições de diferença

Para **resolver o sistema**, basta resolver caminhos mínimos:

- ▶ Executamos BELLMAN-FORD a partir de  $v_0$  no grafo de restrições  $G$
- ▶ Como todo vértice é atingível de  $v_0$ , se houver ciclo negativo, o algoritmo devolve FALSE
- ▶ Se não existir ciclo negativo, então obtemos a solução  $x = (d[v_1], d[v_2], \dots, d[v_n])$

O **tempo de execução** do algoritmo é calculado como:

- ▶ a matriz  $A$  tem dimensões  $m \times n$
- ▶ então  $G$  possui  $n + 1$  vértices e  $n + m$  arestas.
- ▶ O tempo de execução de BELLMAN-FORD em  $G$  é  $O((n + 1)(n + m)) = O(n^2 + nm)$

# Resolvendo um sistema de restrições de diferença

Para **resolver o sistema**, basta resolver caminhos mínimos:

- ▶ Executamos BELLMAN-FORD a partir de  $v_0$  no grafo de restrições  $G$
- ▶ Como todo vértice é atingível de  $v_0$ , se houver ciclo negativo, o algoritmo devolve FALSE
- ▶ Se não existir ciclo negativo, então obtemos a solução  $x = (d[v_1], d[v_2], \dots, d[v_n])$

O **tempo de execução** do algoritmo é calculado como:

- ▶ a matriz  $A$  tem dimensões  $m \times n$
- ▶ então  $G$  possui  $n + 1$  vértices e  $n + m$  arestas.
- ▶ O tempo de execução de BELLMAN-FORD em  $G$  é  $O((n + 1)(n + m)) = O(n^2 + nm)$

# Resolvendo um sistema de restrições de diferença

Para **resolver o sistema**, basta resolver caminhos mínimos:

- ▶ Executamos BELLMAN-FORD a partir de  $v_0$  no grafo de restrições  $G$
- ▶ Como todo vértice é atingível de  $v_0$ , se houver ciclo negativo, o algoritmo devolve FALSE
- ▶ Se não existir ciclo negativo, então obtemos a solução  $x = (d[v_1], d[v_2], \dots, d[v_n])$

O **tempo de execução** do algoritmo é calculado como:

- ▶ a matriz  $A$  tem dimensões  $m \times n$
- ▶ então  $G$  possui  $n + 1$  vértices e  $n + m$  arestas.
- ▶ O tempo de execução de BELLMAN-FORD em  $G$  é  $O((n + 1)(n + m)) = O(n^2 + nm)$

# Resolvendo um sistema de restrições de diferença

Para **resolver o sistema**, basta resolver caminhos mínimos:

- ▶ Executamos BELLMAN-FORD a partir de  $v_0$  no grafo de restrições  $G$
- ▶ Como todo vértice é atingível de  $v_0$ , se houver ciclo negativo, o algoritmo devolve FALSE
- ▶ Se não existir ciclo negativo, então obtemos a solução  $x = (d[v_1], d[v_2], \dots, d[v_n])$

O **tempo de execução** do algoritmo é calculado como:

- ▶ a matriz  $A$  tem dimensões  $m \times n$
- ▶ então  $G$  possui  $n + 1$  vértices e  $n + m$  arestas.
- ▶ O tempo de execução de BELLMAN-FORD em  $G$  é  $O((n + 1)(n + m)) = O(n^2 + nm)$

# Resolvendo um sistema de restrições de diferença

Para **resolver o sistema**, basta resolver caminhos mínimos:

- ▶ Executamos **BELLMAN-FORD** a partir de  $v_0$  no grafo de restrições  $G$
- ▶ Como todo vértice é atingível de  $v_0$ , se houver ciclo negativo, o algoritmo devolve **FALSE**
- ▶ Se não existir ciclo negativo, então obtemos a solução  $x = (d[v_1], d[v_2], \dots, d[v_n])$

O **tempo de execução** do algoritmo é calculado como:

- ▶ a matriz  $A$  tem dimensões  $m \times n$
- ▶ então  $G$  possui  $n + 1$  vértices e  $n + m$  arestas.
- ▶ O tempo de execução de **BELLMAN-FORD** em  $G$  é  $O((n + 1)(n + m)) = O(n^2 + nm)$

# Resolvendo um sistema de restrições de diferença

Para **resolver o sistema**, basta resolver caminhos mínimos:

- ▶ Executamos **BELLMAN-FORD** a partir de  $v_0$  no grafo de restrições  $G$
- ▶ Como todo vértice é atingível de  $v_0$ , se houver ciclo negativo, o algoritmo devolve **FALSE**
- ▶ Se não existir ciclo negativo, então obtemos a solução  $x = (d[v_1], d[v_2], \dots, d[v_n])$

O **tempo de execução** do algoritmo é calculado como:

- ▶ a matriz  $A$  tem dimensões  $m \times n$
- ▶ então  $G$  possui  $n + 1$  vértices e  $n + m$  arestas.
- ▶ O tempo de execução de **BELLMAN-FORD** em  $G$  é  $O((n + 1)(n + m)) = O(n^2 + nm)$

# Resolvendo um sistema de restrições de diferença

Para **resolver o sistema**, basta resolver caminhos mínimos:

- ▶ Executamos BELLMAN-FORD a partir de  $v_0$  no grafo de restrições  $G$
- ▶ Como todo vértice é atingível de  $v_0$ , se houver ciclo negativo, o algoritmo devolve FALSE
- ▶ Se não existir ciclo negativo, então obtemos a solução  $x = (d[v_1], d[v_2], \dots, d[v_n])$

O **tempo de execução** do algoritmo é calculado como:

- ▶ a matriz  $A$  tem dimensões  $m \times n$
- ▶ então  $G$  possui  $n + 1$  vértices e  $n + m$  arestas.
- ▶ O tempo de execução de BELLMAN-FORD em  $G$  é  $O((n + 1)(n + m)) = O(n^2 + nm)$

# Resolvendo um sistema de restrições de diferença

Para **resolver o sistema**, basta resolver caminhos mínimos:

- ▶ Executamos **BELLMAN-FORD** a partir de  $v_0$  no grafo de restrições  $G$
- ▶ Como todo vértice é atingível de  $v_0$ , se houver ciclo negativo, o algoritmo devolve **FALSE**
- ▶ Se não existir ciclo negativo, então obtemos a solução  $x = (d[v_1], d[v_2], \dots, d[v_n])$

O **tempo de execução** do algoritmo é calculado como:

- ▶ a matriz  $A$  tem dimensões  $m \times n$
- ▶ então  $G$  possui  $n + 1$  vértices e  $n + m$  arestas.
- ▶ O tempo de execução de **BELLMAN-FORD** em  $G$  é  $O((n + 1)(n + m)) = O(n^2 + nm)$

Caminhos mínimos entre todos os pares de vértices

# Caminhos mínimos entre todos os pares

**Novo problema:** Dado grafo  $(G, \omega)$  sem ciclos negativos, queremos encontrar um caminho mínimo de  $u$  a  $v$  para **todo** par de vértices  $u, v$ .

# Algoritmos para grafos esparsos

Podemos executar  $|V|$  vezes um algoritmo de Caminhos Mínimos com Mesma Origem:

- ▶ Se  $(G, \omega)$  não possui arestas negativas, usamos DIJKSTRA:

Tipo de fila	Uma vez	$ V $ vezes
Heap	$O(E \lg V)$	$O(VE \lg V)$
Fibonacci	$O(V \lg V + E)$	$O(V^2 \lg V + VE)$

- ▶ Se  $(G, \omega)$  possui arestas negativas, usamos BELLMAN-FORD:

Uma vez	$ V $ vezes
$O(VE)$	$O(V^2E)$

# Algoritmos para grafos esparsos

Podemos executar  $|V|$  vezes um algoritmo de Caminhos Mínimos com Mesma Origem:

- ▶ Se  $(G, \omega)$  não possui arestas negativas, usamos DIJKSTRA:

Tipo de fila	Uma vez	$ V $ vezes
Heap	$O(E \lg V)$	$O(VE \lg V)$
Fibonacci	$O(V \lg V + E)$	$O(V^2 \lg V + VE)$

- ▶ Se  $(G, \omega)$  possui arestas negativas, usamos BELLMAN-FORD:

Uma vez	$ V $ vezes
$O(VE)$	$O(V^2E)$

# Algoritmos para grafos esparsos

Podemos executar  $|V|$  vezes um algoritmo de Caminhos Mínimos com Mesma Origem:

- ▶ Se  $(G, \omega)$  não possui arestas negativas, usamos DIJKSTRA:

Tipo de fila	Uma vez	$ V $ vezes
Heap	$O(E \lg V)$	$O(VE \lg V)$
Fibonacci	$O(V \lg V + E)$	$O(V^2 \lg V + VE)$

- ▶ Se  $(G, \omega)$  possui arestas negativas, usamos BELLMAN-FORD:

Uma vez	$ V $ vezes
$O(VE)$	$O(V^2E)$

# O algoritmo de Floyd-Warshall

**Floyd-Warshall:** é um algoritmo que resolve o problema diretamente e é melhor se  $G$  for **denso**.

- ▶ um grafo é denso se  $|E| = \Omega(V^2)$
- ▶ é baseado em programação dinâmica
- ▶ resolve o problema em tempo  $O(V^3)$
- ▶ supomos que  $G$  é completo:  
⇒ se  $(i,j)$  não é aresta, definimos  $\omega(i,j) = \infty$

# O algoritmo de Floyd-Warshall

**Floyd-Warshall:** é um algoritmo que resolve o problema diretamente e é melhor se  $G$  for **denso**.

- ▶ um grafo é denso se  $|E| = \Omega(V^2)$
- ▶ é baseado em programação dinâmica
- ▶ resolve o problema em tempo  $O(V^3)$
- ▶ supomos que  $G$  é completo:  
⇒ se  $(i,j)$  não é aresta, definimos  $\omega(i,j) = \infty$

# O algoritmo de Floyd-Warshall

**Floyd-Warshall:** é um algoritmo que resolve o problema diretamente e é melhor se  $G$  for **denso**.

- ▶ um grafo é denso se  $|E| = \Omega(V^2)$
- ▶ é baseado em programação dinâmica
- ▶ resolve o problema em tempo  $O(V^3)$
- ▶ supomos que  $G$  é completo:  
⇒ se  $(i,j)$  não é aresta, definimos  $\omega(i,j) = \infty$

# O algoritmo de Floyd-Warshall

**Floyd-Warshall:** é um algoritmo que resolve o problema diretamente e é melhor se  $G$  for **denso**.

- ▶ um grafo é denso se  $|E| = \Omega(V^2)$
- ▶ é baseado em programação dinâmica
- ▶ resolve o problema em tempo  $O(V^3)$
- ▶ supomos que  $G$  é completo:  
⇒ se  $(i,j)$  não é aresta, definimos  $\omega(i,j) = \infty$

# O algoritmo de Floyd-Warshall

**Floyd-Warshall:** é um algoritmo que resolve o problema diretamente e é melhor se  $G$  for **denso**.

- ▶ um grafo é denso se  $|E| = \Omega(V^2)$
- ▶ é baseado em programação dinâmica
- ▶ resolve o problema em tempo  $O(V^3)$
- ▶ supomos que  $G$  é completo:  
⇒ se  $(i,j)$  não é aresta, definimos  $\omega(i,j) = \infty$

# O algoritmo de Floyd-Warshall

**Floyd-Warshall:** é um algoritmo que resolve o problema diretamente e é melhor se  $G$  for **denso**.

- ▶ um grafo é denso se  $|E| = \Omega(V^2)$
- ▶ é baseado em programação dinâmica
- ▶ resolve o problema em tempo  $O(V^3)$
- ▶ supomos que  $G$  é completo:  
⇒ se  $(i,j)$  não é aresta, definimos  $\omega(i,j) = \infty$

# Subproblema

Para simplificar a discussão, suponha que  $V = \{1, 2, \dots, n\}$

Considere um caminho  $P = (v_1, v_2, \dots, v_{l-1}, v_l)$ :

- ▶ os **vértices internos** de  $P$  são  $\{v_2, \dots, v_{l-1}\}$
- ▶  $P$  é chamado  **$k$ -interno** se  $\{v_2, \dots, v_{l-1}\} \subseteq \{1, \dots, k\}$

## Subproblema ótimo

Sejam  $i$  e  $j$  vértices de  $G$  e  $k$  um inteiro com  $k \geq 0$ . Dentre todos os caminhos  $k$ -internos de  $i$  até  $j$ , encontre algum que tenha custo mínimo.

# Subproblema

Para simplificar a discussão, suponha que  $V = \{1, 2, \dots, n\}$

Considere um caminho  $P = (v_1, v_2, \dots, v_{l-1}, v_l)$ :

- ▶ os **vértices internos** de  $P$  são  $\{v_2, \dots, v_{l-1}\}$
- ▶  $P$  é chamado  **$k$ -interno** se  $\{v_2, \dots, v_{l-1}\} \subseteq \{1, \dots, k\}$

## Subproblema ótimo

Sejam  $i$  e  $j$  vértices de  $G$  e  $k$  um inteiro com  $k \geq 0$ . Dentre todos os caminhos  $k$ -internos de  $i$  até  $j$ , encontre algum que tenha custo mínimo.

# Subproblema

Para simplificar a discussão, suponha que  $V = \{1, 2, \dots, n\}$

Considere um caminho  $P = (v_1, v_2, \dots, v_{l-1}, v_l)$ :

- ▶ os **vértices internos** de  $P$  são  $\{v_2, \dots, v_{l-1}\}$
- ▶  $P$  é chamado  **$k$ -interno** se  $\{v_2, \dots, v_{l-1}\} \subseteq \{1, \dots, k\}$

## Subproblema ótimo

Sejam  $i$  e  $j$  vértices de  $G$  e  $k$  um inteiro com  $k \geq 0$ . Dentre todos os caminhos  $k$ -internos de  $i$  até  $j$ , encontre algum que tenha custo mínimo.

# Subproblema

Para simplificar a discussão, suponha que  $V = \{1, 2, \dots, n\}$

Considere um caminho  $P = (v_1, v_2, \dots, v_{l-1}, v_l)$ :

- ▶ os **vértices internos** de  $P$  são  $\{v_2, \dots, v_{l-1}\}$
- ▶  $P$  é chamado  **$k$ -interno** se  $\{v_2, \dots, v_{l-1}\} \subseteq \{1, \dots, k\}$

## Subproblema ótimo

Sejam  $i$  e  $j$  vértices de  $G$  e  $k$  um inteiro com  $k \geq 0$ . Dentre todos os caminhos  $k$ -internos de  $i$  até  $j$ , encontre algum que tenha custo mínimo.

# Subproblema

Para simplificar a discussão, suponha que  $V = \{1, 2, \dots, n\}$

Considere um caminho  $P = (v_1, v_2, \dots, v_{l-1}, v_l)$ :

- ▶ os **vértices internos** de  $P$  são  $\{v_2, \dots, v_{l-1}\}$
- ▶  $P$  é chamado  **$k$ -interno** se  $\{v_2, \dots, v_{l-1}\} \subseteq \{1, \dots, k\}$

## Subproblema ótimo

Sejam  $i$  e  $j$  vértices de  $G$  e  $k$  um inteiro com  $k \geq 0$ . Dentre todos os caminhos  $k$ -internos de  $i$  até  $j$ , encontre algum que tenha custo mínimo.

# Estrutura de um caminho mínimo

O algoritmo de Floyd-Warshall explora a relação entre

- ▶ um caminho  $k$ -interno  $P$  de  $i$  até  $j$  que tem custo mínimo
- ▶ e outros caminhos  $(k - 1)$ -internos

Caso 1: Se  $k$  não é um vértice interno de  $P$ :

- ▶ todos os vértices internos de  $P$  estão em  $\{1, \dots, k - 1\}$
- ▶ então  $P$  é um caminho  $(k - 1)$ -interno de custo mínimo

# Estrutura de um caminho mínimo

O algoritmo de Floyd-Warshall explora a relação entre

- ▶ um caminho  $k$ -interno  $P$  de  $i$  até  $j$  que tem custo mínimo
- ▶ e outros caminhos  $(k - 1)$ -internos

**Caso 1:** Se  $k$  não é um vértice interno de  $P$ :

- ▶ todos os vértices internos de  $P$  estão em  $\{1, \dots, k - 1\}$
- ▶ então  $P$  é um caminho  $(k - 1)$ -interno de custo mínimo

# Estrutura de um caminho mínimo

O algoritmo de Floyd-Warshall explora a relação entre

- ▶ um caminho  $k$ -interno  $P$  de  $i$  até  $j$  que tem custo mínimo
- ▶ e outros caminhos  $(k - 1)$ -internos

**Caso 1:** Se  $k$  não é um vértice interno de  $P$ :

- ▶ todos os vértices internos de  $P$  estão em  $\{1, \dots, k - 1\}$
- ▶ então  $P$  é um caminho  $(k - 1)$ -interno de custo mínimo

# Estrutura de um caminho mínimo

O algoritmo de Floyd-Warshall explora a relação entre

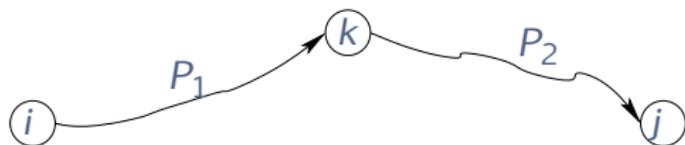
- ▶ um caminho  $k$ -interno  $P$  de  $i$  até  $j$  que tem custo mínimo
- ▶ e outros caminhos  $(k - 1)$ -internos

**Caso 1:** Se  $k$  não é um vértice interno de  $P$ :

- ▶ todos os vértices internos de  $P$  estão em  $\{1, \dots, k - 1\}$
- ▶ então  $P$  é um caminho  $(k - 1)$ -interno de custo mínimo

## Estrutura de um caminho mínimo

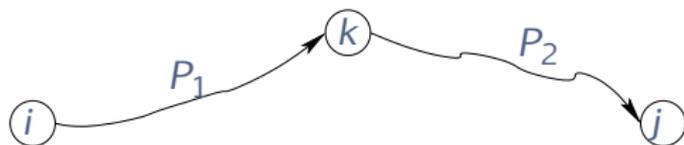
**Caso 2:** Se  $k$  é um vértice interno de  $P$  então  $P$  pode ser dividido em dois caminhos  $P_1$  (com início em  $i$  e fim em  $k$ ) e  $P_2$  (com início em  $k$  e fim em  $j$ ).



- ▶  $P_1$  é um caminho mínimo de  $i$  a  $k$  com vértices internos em  $\{1, \dots, k-1\}$
- ▶  $P_2$  é um caminho mínimo de  $k$  a  $j$  com vértices internos em  $\{1, \dots, k-1\}$ .

## Estrutura de um caminho mínimo

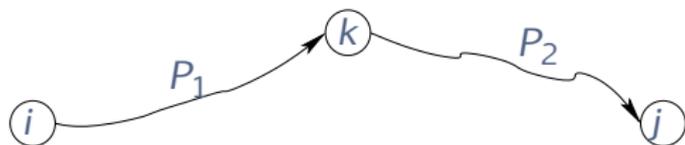
**Caso 2:** Se  $k$  é um vértice interno de  $P$  então  $P$  pode ser dividido em dois caminhos  $P_1$  (com início em  $i$  e fim em  $k$ ) e  $P_2$  (com início em  $k$  e fim em  $j$ ).



- ▶  $P_1$  é um caminho mínimo de  $i$  a  $k$  com vértices internos em  $\{1, \dots, k-1\}$
- ▶  $P_2$  é um caminho mínimo de  $k$  a  $j$  com vértices internos em  $\{1, \dots, k-1\}$ .

## Estrutura de um caminho mínimo

**Caso 2:** Se  $k$  é um vértice interno de  $P$  então  $P$  pode ser dividido em dois caminhos  $P_1$  (com início em  $i$  e fim em  $k$ ) e  $P_2$  (com início em  $k$  e fim em  $j$ ).



- ▶  $P_1$  é um caminho mínimo de  $i$  a  $k$  com vértices internos em  $\{1, \dots, k-1\}$
- ▶  $P_2$  é um caminho mínimo de  $k$  a  $j$  com vértices internos em  $\{1, \dots, k-1\}$ .

# Recorrência para caminhos mínimos

Seja  $d_{ij}^{(k)}$  o peso de um caminho  $k$ -interno mínimo de  $i$  a  $j$ .

- ▶ se  $k = 0$  então  $d_{ij}^{(0)} = \omega(i, j)$
- ▶ senão, caímos em algum dos dois casos anteriores

Obtemos a seguinte recorrência:

$$d_{ij}^{(k)} = \begin{cases} \omega(i, j) & \text{se } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{se } k \geq 1. \end{cases}$$

Note que  $d_{ij}^{(n)} = \text{dist}(i, j)$ . (Por quê?)

- ▶ Calculamos as matrizes  $D^{(k)} = (d_{ij}^{(k)})$  para  $k = 1, 2, \dots, n$ .
- ▶ A resposta do problema é  $D^{(n)}$ .

# Recorrência para caminhos mínimos

Seja  $d_{ij}^{(k)}$  o peso de um caminho  $k$ -interno mínimo de  $i$  a  $j$ .

- ▶ se  $k = 0$  então  $d_{ij}^{(0)} = \omega(i, j)$
- ▶ senão, caímos em algum dos dois casos anteriores

Obtemos a seguinte recorrência:

$$d_{ij}^{(k)} = \begin{cases} \omega(i, j) & \text{se } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{se } k \geq 1. \end{cases}$$

Note que  $d_{ij}^{(n)} = \text{dist}(i, j)$ . (Por quê?)

- ▶ Calculamos as matrizes  $D^{(k)} = (d_{ij}^{(k)})$  para  $k = 1, 2, \dots, n$ .
- ▶ A resposta do problema é  $D^{(n)}$ .

# Recorrência para caminhos mínimos

Seja  $d_{ij}^{(k)}$  o peso de um caminho  $k$ -interno mínimo de  $i$  a  $j$ .

- ▶ se  $k = 0$  então  $d_{ij}^{(0)} = \omega(i, j)$
- ▶ senão, caímos em algum dos dois casos anteriores

Obtemos a seguinte recorrência:

$$d_{ij}^{(k)} = \begin{cases} \omega(i, j) & \text{se } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{se } k \geq 1. \end{cases}$$

Note que  $d_{ij}^{(n)} = \text{dist}(i, j)$ . (Por quê?)

- ▶ Calculamos as matrizes  $D^{(k)} = (d_{ij}^{(k)})$  para  $k = 1, 2, \dots, n$ .
- ▶ A resposta do problema é  $D^{(n)}$ .

# Recorrência para caminhos mínimos

Seja  $d_{ij}^{(k)}$  o peso de um caminho  $k$ -interno mínimo de  $i$  a  $j$ .

- ▶ se  $k = 0$  então  $d_{ij}^{(0)} = \omega(i, j)$
- ▶ senão, caímos em algum dos dois casos anteriores

Obtemos a seguinte recorrência:

$$d_{ij}^{(k)} = \begin{cases} \omega(i, j) & \text{se } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{se } k \geq 1. \end{cases}$$

Note que  $d_{ij}^{(n)} = \text{dist}(i, j)$ . (Por quê?)

- ▶ Calculamos as matrizes  $D^{(k)} = (d_{ij}^{(k)})$  para  $k = 1, 2, \dots, n$ .
- ▶ A resposta do problema é  $D^{(n)}$ .

# Recorrência para caminhos mínimos

Seja  $d_{ij}^{(k)}$  o peso de um caminho  $k$ -interno mínimo de  $i$  a  $j$ .

- ▶ se  $k = 0$  então  $d_{ij}^{(0)} = \omega(i, j)$
- ▶ senão, caímos em algum dos dois casos anteriores

Obtemos a seguinte recorrência:

$$d_{ij}^{(k)} = \begin{cases} \omega(i, j) & \text{se } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{se } k \geq 1. \end{cases}$$

Note que  $d_{ij}^{(n)} = \text{dist}(i, j)$ . (Por quê?)

- ▶ Calculamos as matrizes  $D^{(k)} = (d_{ij}^{(k)})$  para  $k = 1, 2, \dots, n$ .
- ▶ A resposta do problema é  $D^{(n)}$ .

# Recorrência para caminhos mínimos

Seja  $d_{ij}^{(k)}$  o peso de um caminho  $k$ -interno mínimo de  $i$  a  $j$ .

- ▶ se  $k = 0$  então  $d_{ij}^{(0)} = \omega(i, j)$
- ▶ senão, caímos em algum dos dois casos anteriores

Obtemos a seguinte recorrência:

$$d_{ij}^{(k)} = \begin{cases} \omega(i, j) & \text{se } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{se } k \geq 1. \end{cases}$$

Note que  $d_{ij}^{(n)} = \text{dist}(i, j)$ . (Por quê?)

- ▶ Calculamos as matrizes  $D^{(k)} = (d_{ij}^{(k)})$  para  $k = 1, 2, \dots, n$ .
- ▶ A resposta do problema é  $D^{(n)}$ .

# Recorrência para caminhos mínimos

Seja  $d_{ij}^{(k)}$  o peso de um caminho  $k$ -interno mínimo de  $i$  a  $j$ .

- ▶ se  $k = 0$  então  $d_{ij}^{(0)} = \omega(i, j)$
- ▶ senão, caímos em algum dos dois casos anteriores

Obtemos a seguinte recorrência:

$$d_{ij}^{(k)} = \begin{cases} \omega(i, j) & \text{se } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{se } k \geq 1. \end{cases}$$

Note que  $d_{ij}^{(n)} = \text{dist}(i, j)$ . (Por quê?)

- ▶ Calculamos as matrizes  $D^{(k)} = (d_{ij}^{(k)})$  para  $k = 1, 2, \dots, n$ .
- ▶ A resposta do problema é  $D^{(n)}$ .

# Recorrência para caminhos mínimos

Seja  $d_{ij}^{(k)}$  o peso de um caminho  $k$ -interno mínimo de  $i$  a  $j$ .

- ▶ se  $k = 0$  então  $d_{ij}^{(0)} = \omega(i, j)$
- ▶ senão, caímos em algum dos dois casos anteriores

Obtemos a seguinte recorrência:

$$d_{ij}^{(k)} = \begin{cases} \omega(i, j) & \text{se } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{se } k \geq 1. \end{cases}$$

Note que  $d_{ij}^{(n)} = \text{dist}(i, j)$ . (Por quê?)

- ▶ Calculamos as matrizes  $D^{(k)} = (d_{ij}^{(k)})$  para  $k = 1, 2, \dots, n$ .
- ▶ A resposta do problema é  $D^{(n)}$ .

# Algoritmo de Floyd-Warshall

## Entrada:

- ▶ matriz  $W = (\omega(i, j))$  com dimensões  $n \times n$

## Saída:

- ▶ a matriz  $D^{(n)}$

```
FLOYD-WARSHALL( $W, n$ )  
1   $D^{(0)} \leftarrow W$   
2  para  $k \leftarrow 1$  até  $n$  faça  
3      para  $i \leftarrow 1$  até  $n$  faça  
4          para  $j \leftarrow 1$  até  $n$  faça  
5               $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$   
6  devolva  $D^{(n)}$ 
```

Complexidade:  $O(V^3)$

# Algoritmo de Floyd-Warshall

## Entrada:

- ▶ matriz  $W = (\omega(i, j))$  com dimensões  $n \times n$

## Saída:

- ▶ a matriz  $D^{(n)}$

**FLOYD-WARSHALL**( $W, n$ )

1  $D^{(0)} \leftarrow W$

2 para  $k \leftarrow 1$  até  $n$  faça

3     para  $i \leftarrow 1$  até  $n$  faça

4         para  $j \leftarrow 1$  até  $n$  faça

5              $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

6 devolva  $D^{(n)}$

Complexidade:  $O(V^3)$

# Como encontrar os caminhos?

Devolvemos matriz de predecessores  $\Pi = (\pi_{ij})$

- (a)  $\pi_{ij} = \text{NIL}$  se  $i = j$  ou se não existe caminho de  $i$  a  $j$ ,
- (b)  $\pi_{ij}$  é o predecessor de  $j$  em algum caminho mínimo a partir de  $i$ , caso contrário.
  - ▶ Calculamos do mesmo modo que  $D^{(k)}$ .
  - ▶ Obtemos uma sequência de matrizes  $\Pi^{(0)}, \Pi^{(1)}, \dots, \Pi^{(n)}$

Quando  $k = 0$ :

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{se } i = j \text{ ou } \omega(i, j) = \infty, \\ i & \text{se } i \neq j \text{ e } \omega(i, j) < \infty. \end{cases}$$

# Como encontrar os caminhos?

Devolvemos matriz de predecessores  $\Pi = (\pi_{ij})$

- (a)  $\pi_{ij} = \text{NIL}$  se  $i = j$  ou se não existe caminho de  $i$  a  $j$ ,
- (b)  $\pi_{ij}$  é o predecessor de  $j$  em algum caminho mínimo a partir de  $i$ , caso contrário.
  - ▶ Calculamos do mesmo modo que  $D^{(k)}$ .
  - ▶ Obtemos uma sequência de matrizes  $\Pi^{(0)}, \Pi^{(1)}, \dots, \Pi^{(n)}$

Quando  $k = 0$ :

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{se } i = j \text{ ou } \omega(i, j) = \infty, \\ i & \text{se } i \neq j \text{ e } \omega(i, j) < \infty. \end{cases}$$

# Como encontrar os caminhos?

Devolvemos matriz de predecessores  $\Pi = (\pi_{ij})$

- (a)  $\pi_{ij} = \text{NIL}$  se  $i = j$  ou se não existe caminho de  $i$  a  $j$ ,
- (b)  $\pi_{ij}$  é o **predecessor** de  $j$  em algum caminho mínimo a partir de  $i$ , caso contrário.

- ▶ Calculamos do mesmo modo que  $D^{(k)}$ .
- ▶ Obtemos uma sequência de matrizes  $\Pi^{(0)}, \Pi^{(1)}, \dots, \Pi^{(n)}$

Quando  $k = 0$ :

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{se } i = j \text{ ou } \omega(i, j) = \infty, \\ i & \text{se } i \neq j \text{ e } \omega(i, j) < \infty. \end{cases}$$

# Como encontrar os caminhos?

Devolvemos matriz de predecessores  $\Pi = (\pi_{ij})$

- (a)  $\pi_{ij} = \text{NIL}$  se  $i = j$  ou se não existe caminho de  $i$  a  $j$ ,
- (b)  $\pi_{ij}$  é o **predecessor** de  $j$  em algum caminho mínimo a partir de  $i$ , caso contrário.
  - ▶ Calculamos do mesmo modo que  $D^{(k)}$ .
  - ▶ Obtemos uma sequência de matrizes  $\Pi^{(0)}, \Pi^{(1)}, \dots, \Pi^{(n)}$

Quando  $k = 0$ :

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{se } i = j \text{ ou } \omega(i, j) = \infty, \\ i & \text{se } i \neq j \text{ e } \omega(i, j) < \infty. \end{cases}$$

# Como encontrar os caminhos?

Devolvemos matriz de predecessores  $\Pi = (\pi_{ij})$

- (a)  $\pi_{ij} = \text{NIL}$  se  $i = j$  ou se não existe caminho de  $i$  a  $j$ ,
- (b)  $\pi_{ij}$  é o **predecessor** de  $j$  em algum caminho mínimo a partir de  $i$ , caso contrário.
  - ▶ Calculamos do mesmo modo que  $D^{(k)}$ .
  - ▶ Obtemos uma sequência de matrizes  $\Pi^{(0)}, \Pi^{(1)}, \dots, \Pi^{(n)}$

Quando  $k = 0$ :

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{se } i = j \text{ ou } \omega(i, j) = \infty, \\ i & \text{se } i \neq j \text{ e } \omega(i, j) < \infty. \end{cases}$$

# Como encontrar os caminhos?

Se  $k \geq 1$ :

- ▶ Considere um caminho  $k$ -interno mínimo  $P$  de  $i$  a  $j$ .
- ▶ Obtemos o predecessor de  $j$ :
  - ▶ Se  $k$  não aparece em  $P$ , usamos o predecessor de um caminho  $(k-1)$ -interno de  $i$  a  $j$
  - ▶ Se  $k$  aparece em  $P$ , usamos o predecessor  $d$  de um caminho  $(k-1)$ -interno de  $k$  a  $j$

Formalmente,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{se } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{se } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

# Como encontrar os caminhos?

Se  $k \geq 1$ :

- ▶ Considere um caminho  $k$ -interno mínimo  $P$  de  $i$  a  $j$ .
- ▶ Obtemos o predecessor de  $j$ :
  - ▶ Se  $k$  não aparece em  $P$ , usamos o predecessor de um caminho  $(k-1)$ -interno de  $i$  a  $j$
  - ▶ Se  $k$  aparece em  $P$ , usamos o predecessor  $d$  de um caminho  $(k-1)$ -interno de  $k$  a  $j$

Formalmente,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{se } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{se } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

# Como encontrar os caminhos?

Se  $k \geq 1$ :

- ▶ Considere um caminho  $k$ -interno mínimo  $P$  de  $i$  a  $j$ .
- ▶ Obtemos o predecessor de  $j$ :
  - ▶ Se  $k$  não aparece em  $P$ , usamos o predecessor de um caminho  $(k-1)$ -interno de  $i$  a  $j$
  - ▶ Se  $k$  aparece em  $P$ , usamos o predecessor  $d$  de um caminho  $(k-1)$ -interno de  $k$  a  $j$

Formalmente,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{se } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{se } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

# Como encontrar os caminhos?

Se  $k \geq 1$ :

- ▶ Considere um caminho  $k$ -interno mínimo  $P$  de  $i$  a  $j$ .
- ▶ Obtemos o predecessor de  $j$ :
  - ▶ Se  $k$  não aparece em  $P$ , usamos o predecessor de um caminho  $(k-1)$ -interno de  $i$  a  $j$
  - ▶ Se  $k$  aparece em  $P$ , usamos o predecessor  $d$  de um caminho  $(k-1)$ -interno de  $k$  a  $j$

Formalmente,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{se } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{se } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

# Como encontrar os caminhos?

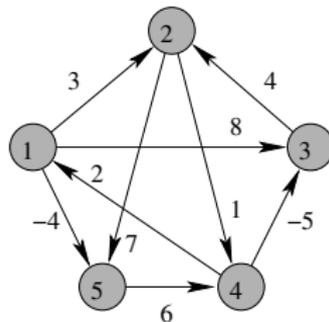
Se  $k \geq 1$ :

- ▶ Considere um caminho  $k$ -interno mínimo  $P$  de  $i$  a  $j$ .
- ▶ Obtemos o predecessor de  $j$ :
  - ▶ Se  $k$  não aparece em  $P$ , usamos o predecessor de um caminho  $(k-1)$ -interno de  $i$  a  $j$
  - ▶ Se  $k$  aparece em  $P$ , usamos o predecessor  $d$  de um caminho  $(k-1)$ -interno de  $k$  a  $j$

Formalmente,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{se } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{se } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

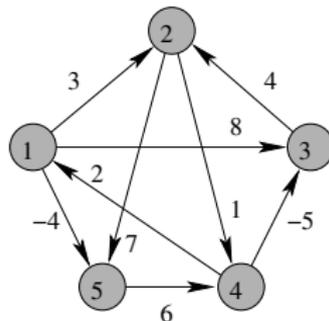
# Exemplo



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} N & 1 & 1 & N & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & N & N \\ 4 & N & 4 & N & N \\ N & N & N & 5 & N \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} N & 1 & 1 & N & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & N & N \\ 4 & 1 & 4 & N & 1 \\ N & N & N & 5 & N \end{pmatrix}$$

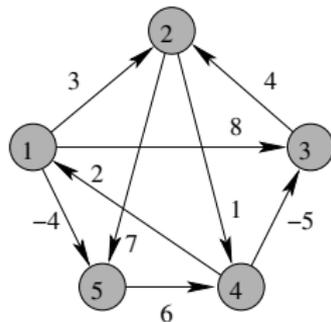
# Exemplo



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} N & 1 & 1 & N & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & N & N \\ 4 & N & 4 & N & N \\ N & N & N & 5 & N \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \mathbf{5} & -5 & 0 & \mathbf{-2} \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} N & 1 & 1 & N & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & N & N \\ 4 & \mathbf{1} & 4 & N & \mathbf{1} \\ N & N & N & 5 & N \end{pmatrix}$$

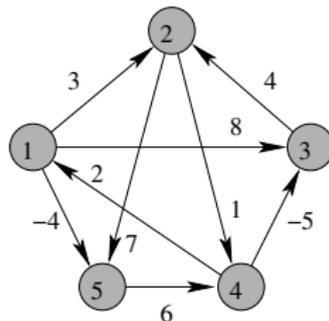
# Exemplo



$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} N & 1 & 1 & N & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & N & N \\ 4 & 1 & 4 & N & 1 \\ N & N & N & 5 & N \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} N & 1 & 1 & 2 & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & 2 & 2 \\ 4 & 1 & 4 & N & 1 \\ N & N & N & 5 & N \end{pmatrix}$$

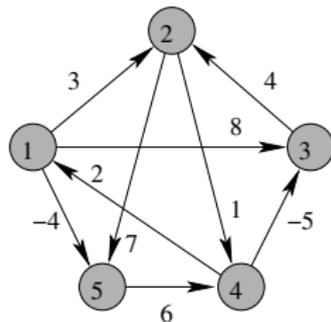
# Exemplo



$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} N & 1 & 1 & N & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & N & N \\ 4 & 1 & 4 & N & 1 \\ N & N & N & 5 & N \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} N & 1 & 1 & 2 & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & 2 & 2 \\ 4 & 1 & 4 & N & 1 \\ N & N & N & 5 & N \end{pmatrix}$$

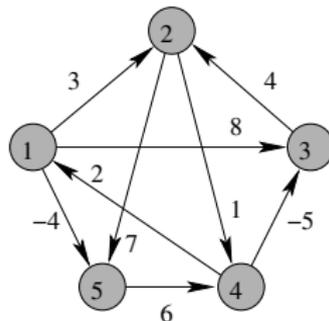
# Exemplo



$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} N & 1 & 1 & 4 & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & 2 & 2 \\ 4 & 1 & 4 & N & 1 \\ N & N & N & 5 & N \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} N & 1 & 1 & 2 & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & 2 & 2 \\ 4 & 3 & 4 & N & 1 \\ N & N & N & 5 & N \end{pmatrix}$$

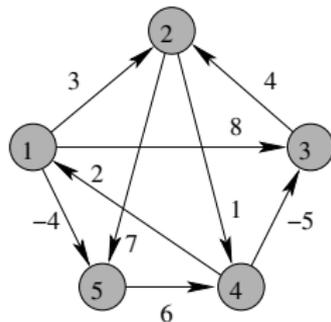
# Exemplo



$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} N & 1 & 1 & 4 & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & 2 & 2 \\ 4 & 1 & 4 & N & 1 \\ N & N & N & 5 & N \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} N & 1 & 1 & 2 & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & 2 & 2 \\ 4 & 3 & 4 & N & 1 \\ N & N & N & 5 & N \end{pmatrix}$$

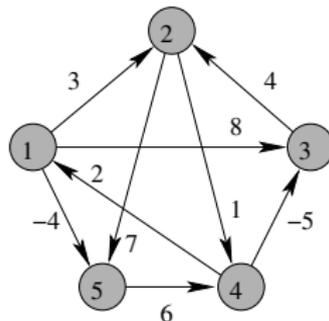
# Exemplo



$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} N & 1 & 1 & 2 & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & 2 & 2 \\ 4 & 3 & 4 & N & 1 \\ N & N & N & 5 & N \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} N & 1 & 4 & 2 & 1 \\ 4 & N & 4 & 2 & 1 \\ 4 & 3 & N & 2 & 1 \\ 4 & 3 & 4 & N & 1 \\ 4 & 3 & 4 & 5 & N \end{pmatrix}$$

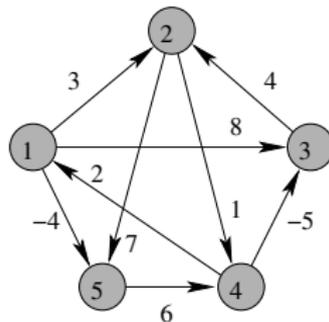
# Exemplo



$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} N & 1 & 1 & 2 & 1 \\ N & N & N & 2 & 2 \\ N & 3 & N & 2 & 2 \\ 4 & 3 & 4 & N & 1 \\ N & N & N & 5 & N \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} N & 1 & 4 & 2 & 1 \\ 4 & N & 4 & 2 & 1 \\ 4 & 3 & N & 2 & 1 \\ 4 & 3 & 4 & N & 1 \\ 4 & 3 & 4 & 5 & N \end{pmatrix}$$

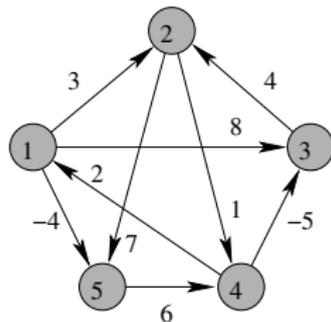
# Exemplo



$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} N & 1 & 4 & 2 & 1 \\ 4 & N & 4 & 2 & 1 \\ 4 & 3 & N & 2 & 1 \\ 4 & 3 & 4 & N & 1 \\ 4 & 3 & 4 & 5 & N \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} N & 3 & 4 & 5 & 1 \\ 4 & N & 4 & 2 & 1 \\ 4 & 3 & N & 2 & 1 \\ 4 & 3 & 4 & N & 1 \\ 4 & 3 & 4 & 5 & N \end{pmatrix}$$

# Exemplo



$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} N & 1 & 4 & 2 & 1 \\ 4 & N & 4 & 2 & 1 \\ 4 & 3 & N & 2 & 1 \\ 4 & 3 & 4 & N & 1 \\ 4 & 3 & 4 & 5 & N \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & \mathbf{1} & \mathbf{-3} & \mathbf{2} & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} N & \mathbf{3} & \mathbf{4} & \mathbf{5} & 1 \\ 4 & N & 4 & 2 & 1 \\ 4 & 3 & N & 2 & 1 \\ 4 & 3 & 4 & N & 1 \\ 4 & 3 & 4 & 5 & N \end{pmatrix}$$

Fecho transitivo de grafos direccionados

## Fecho transitivo de grafos direcionados

Seja  $G = (V, E)$  um grafo direcionado com  $V = \{1, 2, \dots, n\}$ .

O fecho transitivo de  $G = (V, E)$  é o grafo  $G^* = (V, E^*)$  onde

$$E^* = \{(i, j) : \text{existe um caminho de } i \text{ a } j \text{ em } G\}.$$

## Fecho transitivo de grafos direcionados

Seja  $G = (V, E)$  um grafo direcionado com  $V = \{1, 2, \dots, n\}$ .

O **fecho transitivo** de  $G = (V, E)$  é o grafo  $G^* = (V, E^*)$  onde

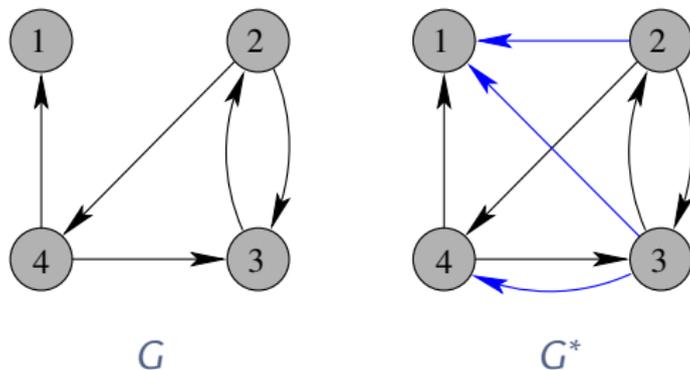
$$E^* = \{(i, j) : \text{existe um caminho de } i \text{ a } j \text{ em } G\}.$$

# Fecho transitivo de grafos direcionados

Seja  $G = (V, E)$  um grafo direcionado com  $V = \{1, 2, \dots, n\}$ .

O **fecho transitivo** de  $G = (V, E)$  é o grafo  $G^* = (V, E^*)$  onde

$$E^* = \{(i, j) : \text{existe um caminho de } i \text{ a } j \text{ em } G\}.$$



# Um detalhe de implementação

Determinando o fecho transitivo de  $G = (V, E)$ :

1. atribuímos peso 1 a cada aresta
2. executar FLOYD-WARSHALL em tempo  $\Theta(V^3)$
3. existe um caminho de  $i$  a  $j$  se e somente se  $d_{ij} < |V|$

Na prática:

- ▶ executamos FLOYD-WARSHALL substituindo:
  - ▶ min por  $\vee$  (OU lógico)
  - ▶ + por  $\wedge$  (E lógico)
- ▶ tem a mesma complexidade assintótica
- ▶ mas economiza tempo e espaço

# Um detalhe de implementação

Determinando o fecho transitivo de  $G = (V, E)$ :

1. atribuímos peso 1 a cada aresta
2. executar FLOYD-WARSHALL em tempo  $\Theta(V^3)$
3. existe um caminho de  $i$  a  $j$  se e somente se  $d_{ij} < |V|$

Na prática:

- ▶ executamos FLOYD-WARSHALL substituindo:
  - ▶ min por  $\vee$  (OU lógico)
  - ▶ + por  $\wedge$  (E lógico)
- ▶ tem a mesma complexidade assintótica
- ▶ mas economiza tempo e espaço

# Um detalhe de implementação

Determinando o fecho transitivo de  $G = (V, E)$ :

1. atribuímos peso 1 a cada aresta
2. executar FLOYD-WARSHALL em tempo  $\Theta(V^3)$
3. existe um caminho de  $i$  a  $j$  se e somente se  $d_{ij} < |V|$

Na prática:

- ▶ executamos FLOYD-WARSHALL substituindo:
  - ▶ min por  $\vee$  (OU lógico)
  - ▶ + por  $\wedge$  (E lógico)
- ▶ tem a mesma complexidade assintótica
- ▶ mas economiza tempo e espaço

# Um detalhe de implementação

Determinando o fecho transitivo de  $G = (V, E)$ :

1. atribuímos peso 1 a cada aresta
2. executar FLOYD-WARSHALL em tempo  $\Theta(V^3)$
3. existe um caminho de  $i$  a  $j$  se e somente se  $d_{ij} < |V|$

Na prática:

- ▶ executamos FLOYD-WARSHALL substituindo:
  - ▶ min por  $\vee$  (OU lógico)
  - ▶ + por  $\wedge$  (E lógico)
- ▶ tem a mesma complexidade assintótica
- ▶ mas economiza tempo e espaço

# Um detalhe de implementação

Determinando o fecho transitivo de  $G = (V, E)$ :

1. atribuímos peso 1 a cada aresta
2. executar FLOYD-WARSHALL em tempo  $\Theta(V^3)$
3. existe um caminho de  $i$  a  $j$  se e somente se  $d_{ij} < |V|$

Na prática:

- ▶ executamos FLOYD-WARSHALL substituindo:
  - ▶ min por  $\vee$  (OU lógico)
  - ▶ + por  $\wedge$  (E lógico)
- ▶ tem a mesma complexidade assintótica
- ▶ mas economiza tempo e espaço

# Um detalhe de implementação

Determinando o fecho transitivo de  $G = (V, E)$ :

1. atribuímos peso 1 a cada aresta
2. executar FLOYD-WARSHALL em tempo  $\Theta(V^3)$
3. existe um caminho de  $i$  a  $j$  se e somente se  $d_{ij} < |V|$

Na prática:

- ▶ executamos FLOYD-WARSHALL substituindo:
  - ▶ min por  $\vee$  (OU lógico)
  - ▶ + por  $\wedge$  (E lógico)
- ▶ tem a mesma complexidade assintótica
- ▶ mas economiza tempo e espaço

# Um detalhe de implementação

Determinando o fecho transitivo de  $G = (V, E)$ :

1. atribuímos peso 1 a cada aresta
2. executar FLOYD-WARSHALL em tempo  $\Theta(V^3)$
3. existe um caminho de  $i$  a  $j$  se e somente se  $d_{ij} < |V|$

Na prática:

- ▶ executamos FLOYD-WARSHALL substituindo:
  - ▶ min por  $\vee$  (OU lógico)
  - ▶ + por  $\wedge$  (E lógico)
- ▶ tem a mesma complexidade assintótica
- ▶ mas economiza tempo e espaço

# Um detalhe de implementação

Determinando o fecho transitivo de  $G = (V, E)$ :

1. atribuímos peso 1 a cada aresta
2. executar FLOYD-WARSHALL em tempo  $\Theta(V^3)$
3. existe um caminho de  $i$  a  $j$  se e somente se  $d_{ij} < |V|$

Na prática:

- ▶ executamos FLOYD-WARSHALL substituindo:
  - ▶  $\min$  por  $\vee$  (OU lógico)
  - ▶  $+$  por  $\wedge$  (E lógico)
- ▶ tem a mesma complexidade assintótica
- ▶ mas economiza tempo e espaço

# Fecho transitivo de grafos direcionados

Defina  $t_{i,j}^{(k)}$  o valor booleano:

- ▶ TRUE se existe caminho  $k$ -interno de  $i$  a  $j$
- ▶ FALSE se **não** existe caminho  $k$ -interno de  $i$  a  $j$

Para  $k = 0$

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{se } i \neq j \text{ e } (i,j) \notin E, \\ 1 & \text{se } i = j \text{ ou } (i,j) \in E, \end{cases}$$

e para  $k \geq 1$ ,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}).$$

Como em FLOYD-WARSHALL, calculamos matrizes  $T^{(k)} = (t_{ij}^{(k)})$ .

# Fecho transitivo de grafos direcionados

Defina  $t_{i,j}^{(k)}$  o valor booleano:

- ▶ TRUE se existe caminho  $k$ -interno de  $i$  a  $j$
- ▶ FALSE se não existe caminho  $k$ -interno de  $i$  a  $j$

Para  $k = 0$

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{se } i \neq j \text{ e } (i,j) \notin E, \\ 1 & \text{se } i = j \text{ ou } (i,j) \in E, \end{cases}$$

e para  $k \geq 1$ ,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}).$$

Como em FLOYD-WARSHALL, calculamos matrizes  $T^{(k)} = (t_{ij}^{(k)})$ .

# Fecho transitivo de grafos direcionados

Defina  $t_{i,j}^{(k)}$  o valor booleano:

- ▶ TRUE se existe caminho  $k$ -interno de  $i$  a  $j$
- ▶ FALSE se **não** existe caminho  $k$ -interno de  $i$  a  $j$

Para  $k = 0$

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{se } i \neq j \text{ e } (i,j) \notin E, \\ 1 & \text{se } i = j \text{ ou } (i,j) \in E, \end{cases}$$

e para  $k \geq 1$ ,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}).$$

Como em FLOYD-WARSHALL, calculamos matrizes  $T^{(k)} = (t_{ij}^{(k)})$ .

# Fecho transitivo de grafos direcionados

Defina  $t_{i,j}^{(k)}$  o valor booleano:

- ▶ TRUE se existe caminho  $k$ -interno de  $i$  a  $j$
- ▶ FALSE se **não** existe caminho  $k$ -interno de  $i$  a  $j$

Para  $k = 0$

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{se } i \neq j \text{ e } (i,j) \notin E, \\ 1 & \text{se } i = j \text{ ou } (i,j) \in E, \end{cases}$$

e para  $k \geq 1$ ,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}).$$

Como em FLOYD-WARSHALL, calculamos matrizes  $T^{(k)} = (t_{ij}^{(k)})$ .

# Fecho transitivo de grafos direcionados

Defina  $t_{i,j}^{(k)}$  o valor booleano:

- ▶ TRUE se existe caminho  $k$ -interno de  $i$  a  $j$
- ▶ FALSE se **não** existe caminho  $k$ -interno de  $i$  a  $j$

Para  $k = 0$

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{se } i \neq j \text{ e } (i,j) \notin E, \\ 1 & \text{se } i = j \text{ ou } (i,j) \in E, \end{cases}$$

e para  $k \geq 1$ ,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}).$$

Como em FLOYD-WARSHALL, calculamos matrizes  $T^{(k)} = (t_{ij}^{(k)})$ .

# Fecho transitivo de grafos direcionados

Defina  $t_{i,j}^{(k)}$  o valor booleano:

- ▶ TRUE se existe caminho  $k$ -interno de  $i$  a  $j$
- ▶ FALSE se **não** existe caminho  $k$ -interno de  $i$  a  $j$

Para  $k = 0$

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{se } i \neq j \text{ e } (i,j) \notin E, \\ 1 & \text{se } i = j \text{ ou } (i,j) \in E, \end{cases}$$

e para  $k \geq 1$ ,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}).$$

Como em FLOYD-WARSHALL, calculamos matrizes  $T^{(k)} = (t_{ij}^{(k)})$ .

# Algoritmo de Transitive-Closure

## Entrada:

- ▶ matriz de adjacência  $A$  de  $G$

## Saída:

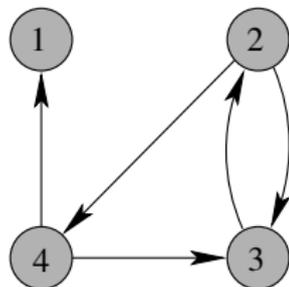
- ▶ a matriz de adjacência  $T^{(n)}$  de  $G^*$

### TRANSITIVE-CLOSURE( $A, n$ )

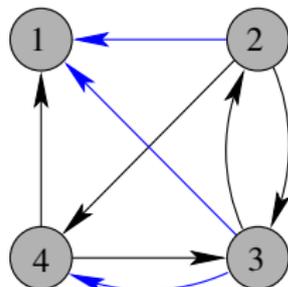
```
1   $T^{(0)} \leftarrow A + I_n$ 
2  para  $k \leftarrow 1$  até  $n$  faça
3      para  $i \leftarrow 1$  até  $n$  faça
4          para  $j \leftarrow 1$  até  $n$  faça
5               $t_{ij}^{(k)} \leftarrow t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$ 
6  devolva  $T^{(n)}$ 
```

Complexidade:  $O(V^3)$

# Exemplo



$G$

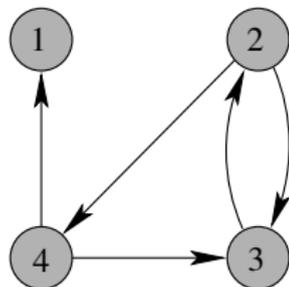


$G^*$

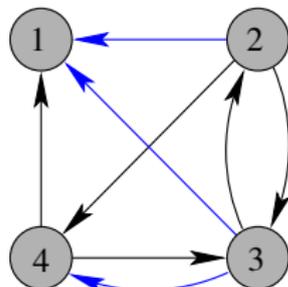
$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

# Exemplo



$G$

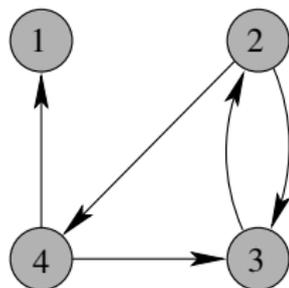


$G^*$

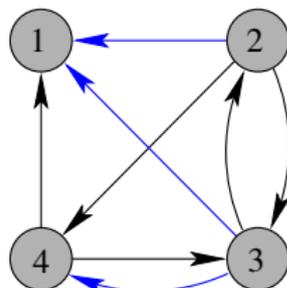
$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

# Exemplo



$G$

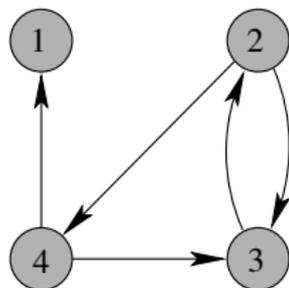


$G^*$

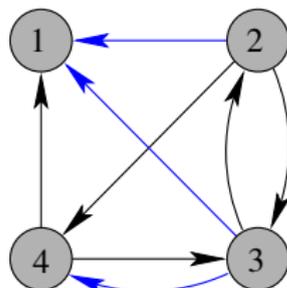
$$T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

# Exemplo



$G$

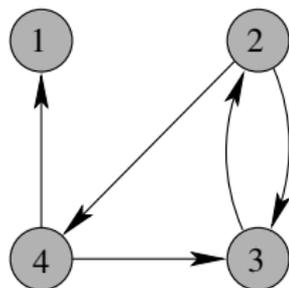


$G^*$

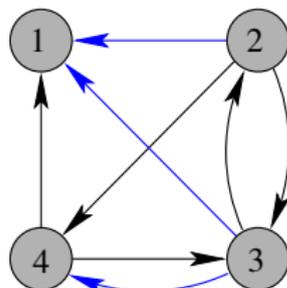
$$T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & \mathbf{1} \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

# Exemplo



$G$

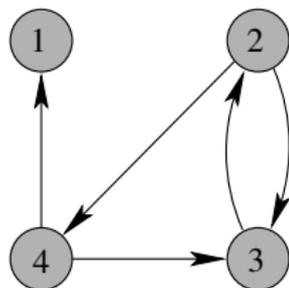


$G^*$

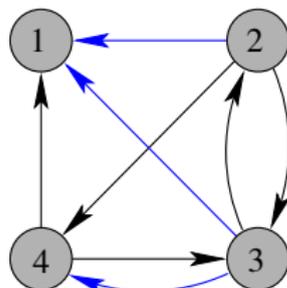
$$T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

# Exemplo



$G$

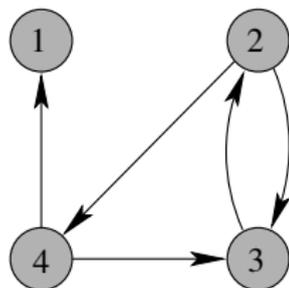


$G^*$

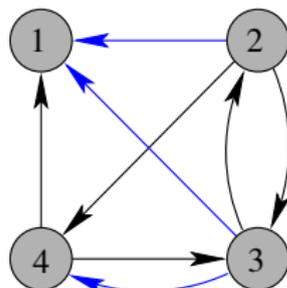
$$T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

# Exemplo



$G$

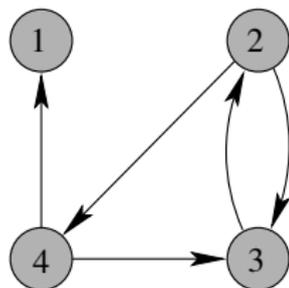


$G^*$

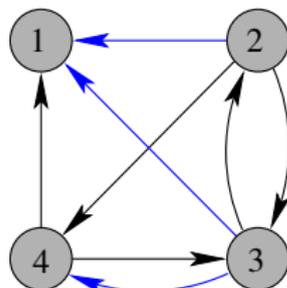
$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

$$T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

# Exemplo



$G$



$G^*$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

$$T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$