Projeto e Análise de Algoritmos

Caminhos mínimos

Lehilton Pedrosa

Primeiro Semestre de 2023

Caminhos mínimos com uma origem

Problema do Caminho Mínimo

Considere um par (G, w) em que

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

Estamos interessados nos seguintes problemas:

1. **Problema do caminho mínimo entre dois vértices:** dados s e t, encontrar 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 todo vértice v de G.

Problema do Caminho Mínimo

Considere um par (G, w) em que

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

Estamos interessados nos seguintes problemas:

1. **Problema do caminho mínimo entre dois vértices:** dados s e t, encontrar 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 todo vértice v de G.

Problema do Caminho Mínimo

Considere um par (G, w) em que

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

Estamos interessados nos seguintes problemas:

1. **Problema do caminho mínimo entre dois vértices:** dados s e t, encontrar 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 todo vértice v de G.

Problema do Caminho Mínimo

Considere um par (G, w) em que

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

Estamos interessados nos seguintes problemas:

1. **Problema do caminho mínimo entre dois vértices:** dados s e t, encontrar 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 todo vértice v de G.

Problema do Caminho Mínimo

Considere um par (G, w) em que

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

Estamos interessados nos seguintes problemas:

1. **Problema do caminho mínimo entre dois vértices:** dados s e t, encontrar 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 todo vértice v de G.

Problema do Caminho Mínimo

Considere um par (G, w) em que

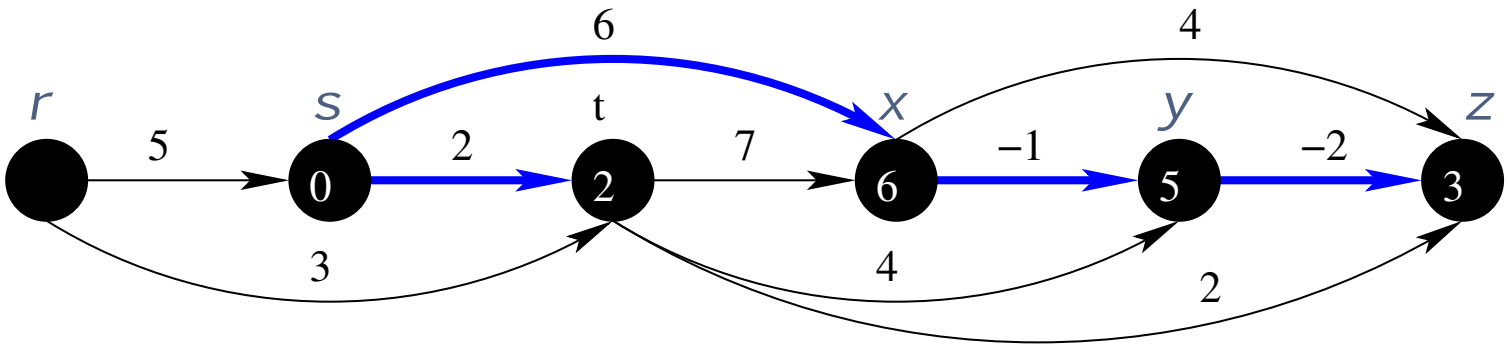
- G é um grafo direcionado
- w associa um peso $w(u, v)$ para cada aresta (u, v)

Estamos interessados nos seguintes problemas:

1. **Problema do caminho mínimo entre dois vértices:** dados s e t, encontrar um caminho de peso mínimo de s a t.

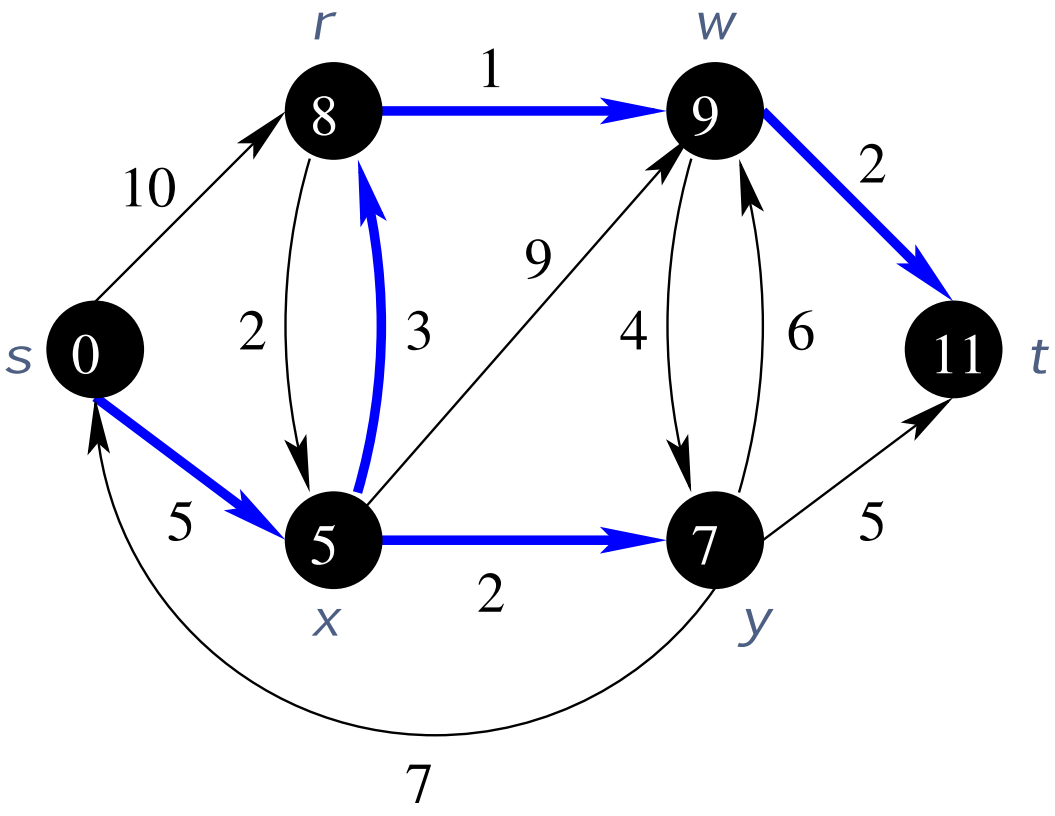
2. **Problema dos caminhos mínimos com mesma origem:** dado s, encontrar um caminho de peso mínimo de s a v para todo vértice v de G.

Exemplo: grafo direcionado acíclico



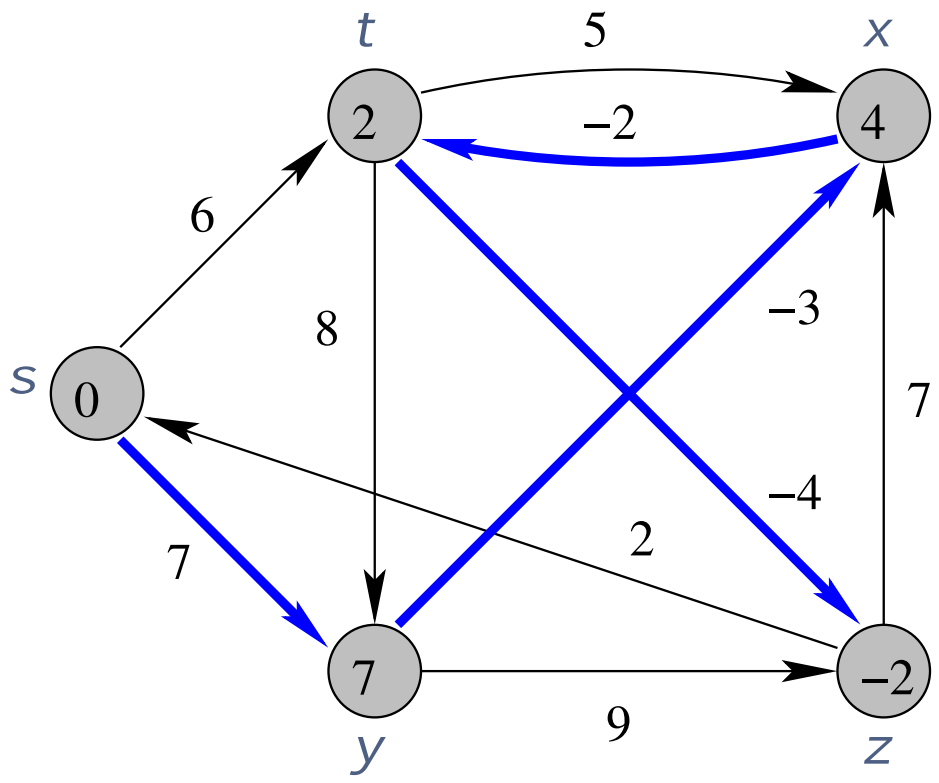
v	s	r	t	x	y	z
dist(s, v)	0	∞	2	6	5	3

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

Representando caminhos mínimos

A saída é similar à da busca em largura a partir de s:

- para cada $v \in V[G]$, associamos um predecessor $\pi[v]$
- o vetor π induz uma árvore de caminhos mínimos com raiz em s
- um caminho de s a v na árvore é um caminho mínimo de s a v no grafo G

Representando caminhos mínimos

A saída é similar à da busca em largura a partir de s:

- para cada $v \in V[G]$, associamos um predecessor $\pi[v]$
- o vetor π induz uma árvore de caminhos mínimos com raiz em s
- um caminho de s a v na árvore é um caminho mínimo de s a v no grafo G

Representando caminhos mínimos

A saída é similar à da busca em largura a partir de s:

- para cada $v \in V[G]$, associamos um predecessor $\pi[v]$
- o vetor π induz uma árvore de caminhos mínimos com raiz em s
- um caminho de s a v na árvore é um caminho mínimo de s a v no grafo G

Representando caminhos mínimos

A saída é similar à da busca em largura a partir de s:

- para cada $v \in V[G]$, associamos um predecessor $\pi[v]$
- o vetor π induz uma árvore de caminhos mínimos com raiz em s
- um caminho de s a v na árvore é um caminho mínimo de s a v no grafo G

Definição

Problema dos caminhos mínimos com mesma origem

Entrada:

- grafo direcionado acíclico $G = (V, E)$
- função de peso w nas arestas
- vértice de origem s

Saída:

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

Definição

Problema dos caminhos mínimos com mesma origem

Entrada:

- grafo direcionado acíclico $G = (V, E)$
- função de peso w nas arestas
- vértice de origem s

Saída:

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

Definição

Problema dos caminhos mínimos com mesma origem

Entrada:

- grafo direcionado acíclico $G = (V, E)$
- função de peso w nas arestas
- vértice de origem s

Saída:

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

Definição

Problema dos caminhos mínimos com mesma origem

Entrada:

- grafo direcionado acíclico $G = (V, E)$
- função de peso w nas arestas
- vértice de origem s

Saída:

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

Definição

Problema dos caminhos mínimos com mesma origem

Entrada:

- grafo direcionado acíclico $G = (V, E)$
- função de peso w nas arestas
- vértice de origem s

Saída:

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

Definição

Problema dos caminhos mínimos com mesma origem

Entrada:

- grafo direcionado acíclico $G = (V, E)$
- função de peso w nas arestas
- vértice de origem s

Saída:

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

Definição

Problema dos caminhos mínimos com mesma origem

Entrada:

- grafo direcionado acíclico $G = (V, E)$
- função de peso w nas arestas
- vértice de origem s

Saída:

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

Definição

Problema dos caminhos mínimos com mesma origem

Entrada:

- grafo direcionado acíclico $G = (V, E)$
- função de peso w nas arestas
- vértice de origem s

Saída:

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

Subestrutura ótima de caminhos mínimos

Teorema

Seja (G, w) um grafo direcionado sem ciclos negativos e seja

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

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

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

é um caminho mínimo de v_i a v_j.

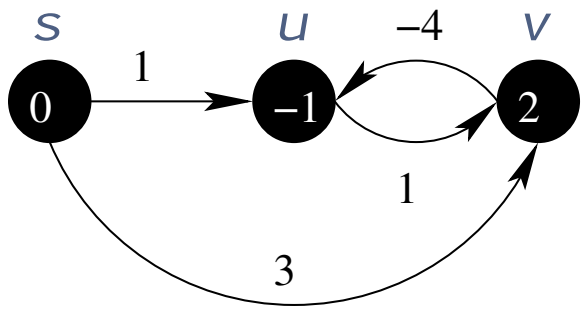
Subestrutura ótima de caminhos mínimos

A subestrutura ótima não vale se o grafo tiver ciclos negativos:

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

Subestrutura ótima de caminhos mínimos

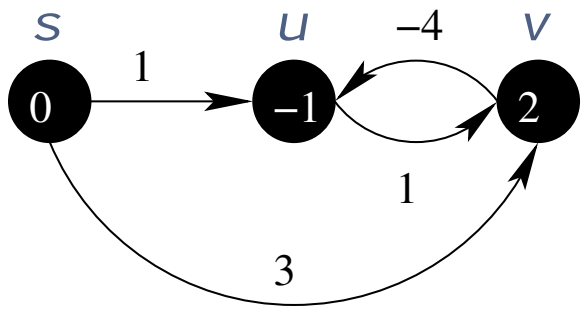
A subestrutura ótima não vale se o grafo tiver ciclos negativos:



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

Subestrutura ótima de caminhos mínimos

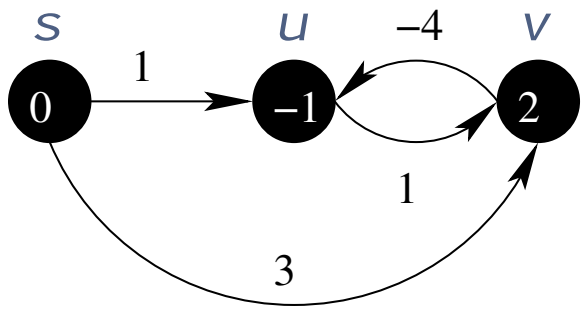
A subestrutura ótima não vale se o grafo tiver ciclos negativos:



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

Subestrutura ótima de caminhos mínimos

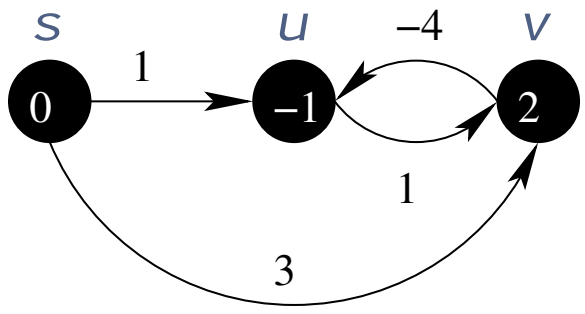
A subestrutura ótima não vale se o grafo tiver ciclos negativos:



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

Subestrutura ótima de caminhos mínimos

A subestrutura ótima não vale se o grafo tiver ciclos negativos:



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

Algoritmos baseados em relaxação

Veremos algoritmos com as seguintes características:

1. Inicializam d e π com uma sub-rotina INITIALIZE-SINGLE-SOURCE
2. Alteram d e π apenas com uma sub-rotina RELAX

Esses algoritmos mantêm algumas invariantes:

- existe um caminho de s a v com peso $d[v]$
- esse caminho pode ser recuperado por meio π
- assim, $d[v]$ é sempre maior ou igual a $\text{dist}(s, v)$
- queremos que no final valha $d[v] = \text{dist}(s, v)$

Algoritmos baseados em relaxação

Veremos algoritmos com as seguintes características:

1. Inicializam d e π com uma sub-rotina INITIALIZE-SINGLE-SOURCE
2. Alteram d e π apenas com uma sub-rotina RELAX

Esses algoritmos mantêm algumas invariantes:

- existe um caminho de s a v com peso $d[v]$
- esse caminho pode ser recuperado por meio π
- assim, $d[v]$ é sempre maior ou igual a $\text{dist}(s, v)$
- queremos que no final valha $d[v] = \text{dist}(s, v)$

Algoritmos baseados em relaxação

Veremos algoritmos com as seguintes características:

1. Inicializam d e π com uma sub-rotina INITIALIZE-SINGLE-SOURCE
2. Alteram d e π apenas com uma sub-rotina RELAX

Esses algoritmos mantêm algumas invariantes:

- existe um caminho de s a v com peso $d[v]$
- esse caminho pode ser recuperado por meio π
- assim, $d[v]$ é sempre maior ou igual a $\text{dist}(s, v)$
- queremos que no final valha $d[v] = \text{dist}(s, v)$

Algoritmos baseados em relaxação

Veremos algoritmos com as seguintes características:

1. Inicializam d e π com uma sub-rotina INITIALIZE-SINGLE-SOURCE
2. Alteram d e π apenas com uma sub-rotina RELAX

Esses algoritmos mantêm algumas invariantes:

- existe um caminho de s a v com peso $d[v]$
- esse caminho pode ser recuperado por meio π
- assim, $d[v]$ é sempre maior ou igual a $\text{dist}(s, v)$
- queremos que no final valha $d[v] = \text{dist}(s, v)$

Algoritmos baseados em relaxação

Veremos algoritmos com as seguintes características:

1. Inicializam d e π com uma sub-rotina INITIALIZE-SINGLE-SOURCE
2. Alteram d e π apenas com uma sub-rotina RELAX

Esses algoritmos mantêm algumas invariantes:

- existe um caminho de s a v com peso $d[v]$
- esse caminho pode ser recuperado por meio π
- assim, $d[v]$ é sempre maior ou igual a $\text{dist}(s, v)$
- queremos que no final valha $d[v] = \text{dist}(s, v)$

Algoritmos baseados em relaxação

Veremos algoritmos com as seguintes características:

1. Inicializam d e π com uma sub-rotina INITIALIZE-SINGLE-SOURCE
2. Alteram d e π apenas com uma sub-rotina RELAX

Esses algoritmos mantêm algumas invariantes:

- existe um caminho de s a v com peso $d[v]$
- esse caminho pode ser recuperado por meio π
- assim, $d[v]$ é sempre maior ou igual a $\text{dist}(s, v)$
- queremos que no final valha $d[v] = \text{dist}(s, v)$

Algoritmos baseados em relaxação

Veremos algoritmos com as seguintes características:

1. Inicializam d e π com uma sub-rotina INITIALIZE-SINGLE-SOURCE
2. Alteram d e π apenas com uma sub-rotina RELAX

Esses algoritmos mantêm algumas invariantes:

- existe um caminho de s a v com peso $d[v]$
- esse caminho pode ser recuperado por meio π
- assim, $d[v]$ é sempre maior ou igual a $\text{dist}(s, v)$
- queremos que no final valha $d[v] = \text{dist}(s, v)$

Algoritmos baseados em relaxação

Veremos algoritmos com as seguintes características:

1. Inicializam d e π com uma sub-rotina INITIALIZE-SINGLE-SOURCE
2. Alteram d e π apenas com uma sub-rotina RELAX

Esses algoritmos mantêm algumas invariantes:

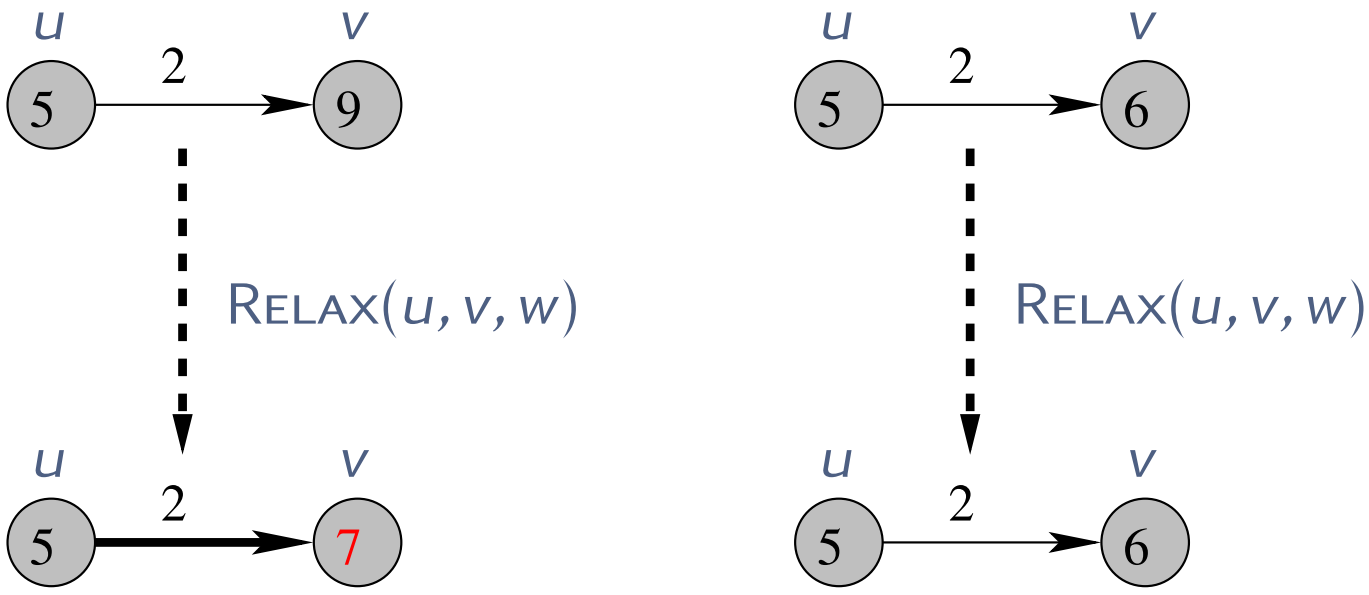
- existe um caminho de s a v com peso $d[v]$
- esse caminho pode ser recuperado por meio π
- assim, $d[v]$ é sempre maior ou igual a $\text{dist}(s, v)$
- queremos que no final valha $d[v] = \text{dist}(s, v)$

Inicialização

```
INITIALIZE-SINGLE-SOURCE(G, s)
1   para cada vértice v ∈ V[G]
2       d[v] ← ∞
3       π[v] ← NIL
4   d[s] ← 0
```

Relaxação

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

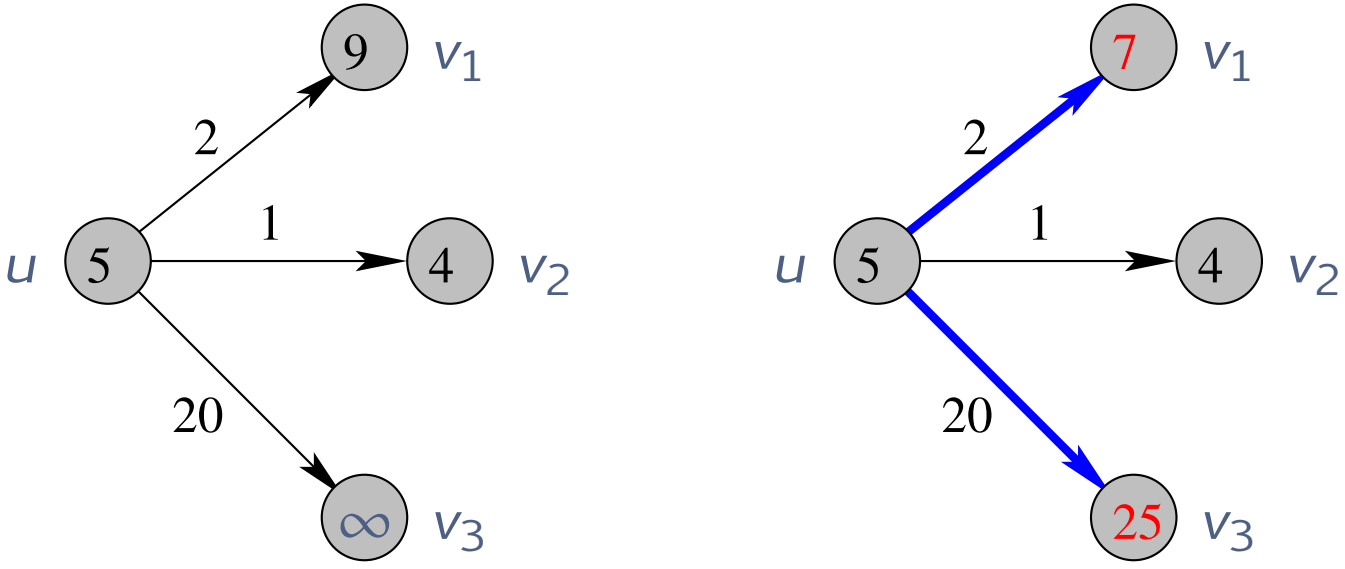


RELAX(u, v, w)

1 se $d[v] > d[u] + w(u, v)$
2 $\quad d[v] \leftarrow d[u] + w(u, v)$
3 $\quad \pi[v] \leftarrow u$

Relaxação dos vizinhos

Em cada iteração o algoritmo seleciona um vértice u e para cada vizinho v de u aplica RELAX(u, v, w).



RELAX(u, v, w)
1 se $d[v] > d[u] + w(u, v)$
2 $\quad d[v] \leftarrow d[u] + w(u, v)$
3 $\quad \pi[v] \leftarrow u$

Casos do problema de caminhos mínimos

Veremos três algoritmos baseados em relaxação para tipos de subcasos diferentes do Problema de caminhos mínimos.

1. aplicação de ordenação topológica: G é acíclico:
2. algoritmo de Dijkstra: (G, w) não tem arestas de peso negativo
3. algoritmo de Bellman-Ford: (G, w) pode ter arestas de peso negativo, mas não contém ciclos negativos

Casos do problema de caminhos mínimos

Veremos três algoritmos baseados em relaxação para tipos de subcasos diferentes do Problema de caminhos mínimos.

1. aplicação de ordenação topológica: G é acíclico:
2. algoritmo de Dijkstra: (G, w) não tem arestas de peso negativo
3. algoritmo de Bellman-Ford: (G, w) pode ter arestas de peso negativo, mas não contém ciclos negativos

Casos do problema de caminhos mínimos

Veremos três algoritmos baseados em relaxação para tipos de subcasos diferentes do Problema de caminhos mínimos.

1. aplicação de ordenação topológica: G é acíclico:
2. algoritmo de Dijkstra: (G, w) não tem arestas de peso negativo
3. algoritmo de Bellman-Ford: (G, w) pode ter arestas de peso negativo, mas não contém ciclos negativos

Casos do problema de caminhos mínimos

Veremos três algoritmos baseados em relaxação para tipos de subcasos diferentes do Problema de caminhos mínimos.

1. aplicação de ordenação topológica: G é acíclico:
2. algoritmo de Dijkstra: (G, w) não tem arestas de peso negativo
3. algoritmo de Bellman-Ford: (G, w) pode ter arestas de peso negativo, mas não contém ciclos negativos

Caminhos mínimos em grafos acíclicos

Caminhos mínimos em grafos acíclicos

Entrada:

- grafo direcionado acíclico $G = (V, E)$
- função de peso w nas arestas
- origem s

Saída:

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

```
DAG-SHORTEST-PATHS(G, w, s)
1  Ordene topologicamente os vértices de G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  para cada vértice u na ordem topológica
4      para cada v ∈ Adj[u]
5          RELAX(u, v, w)
6  devolva d, π
```

Caminhos mínimos em grafos acíclicos

Entrada:

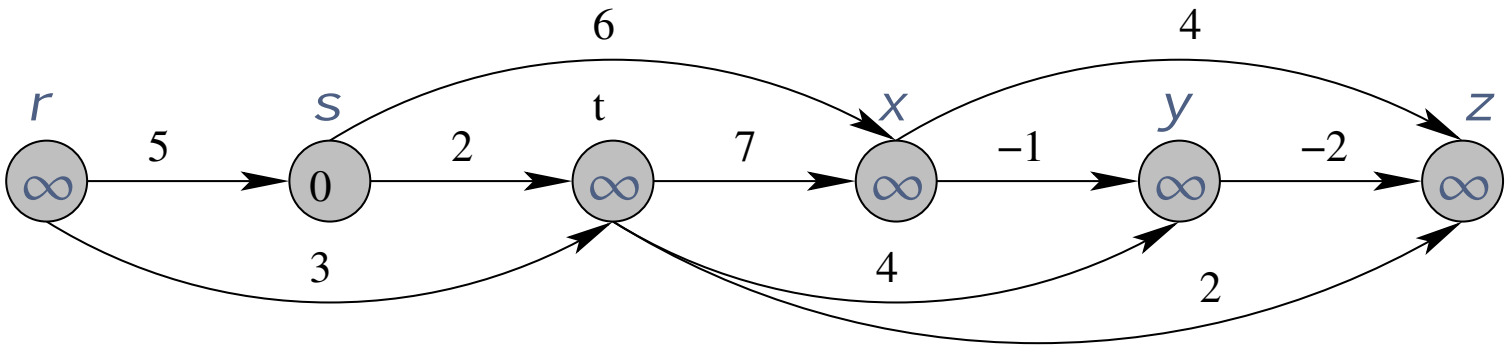
- grafo direcionado acíclico $G = (V, E)$
- função de peso w nas arestas
- origem s

Saída:

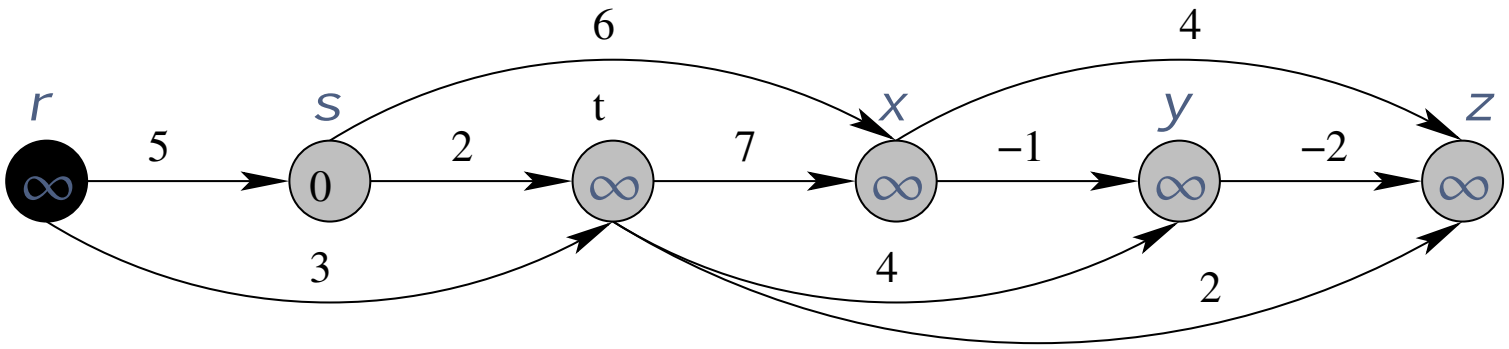
- vetor d com $d[v] = \text{dist}(s, v)$ para $v \in V$
- vetor π definindo uma árvore de caminhos mínimos

```
DAG-SHORTEST-PATHS(G, w, s)
1  Ordene topologicamente os vértices de G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  para cada vértice u na ordem topológica
4      para cada v ∈ Adj[u]
5          RELAX(u, v, w)
6  devolva d, π
```

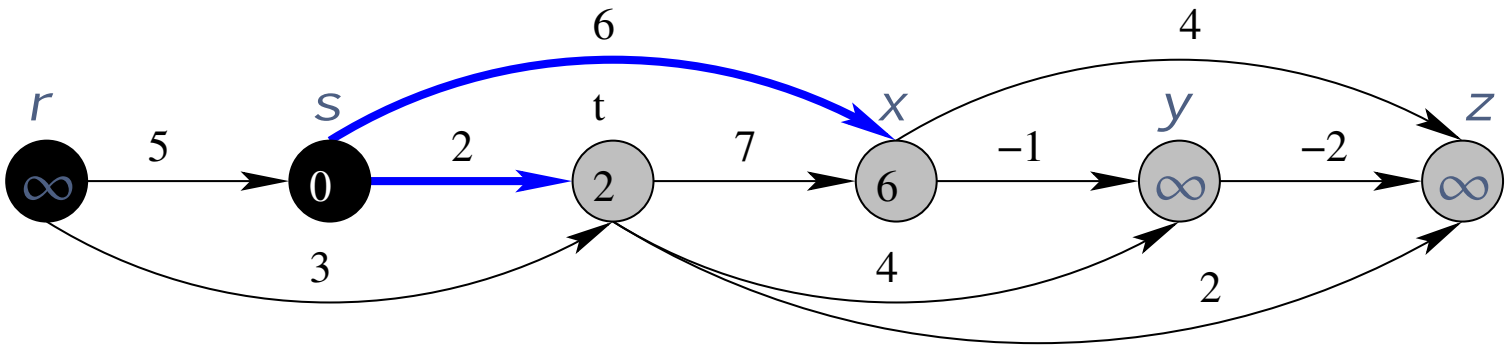
Exemplo



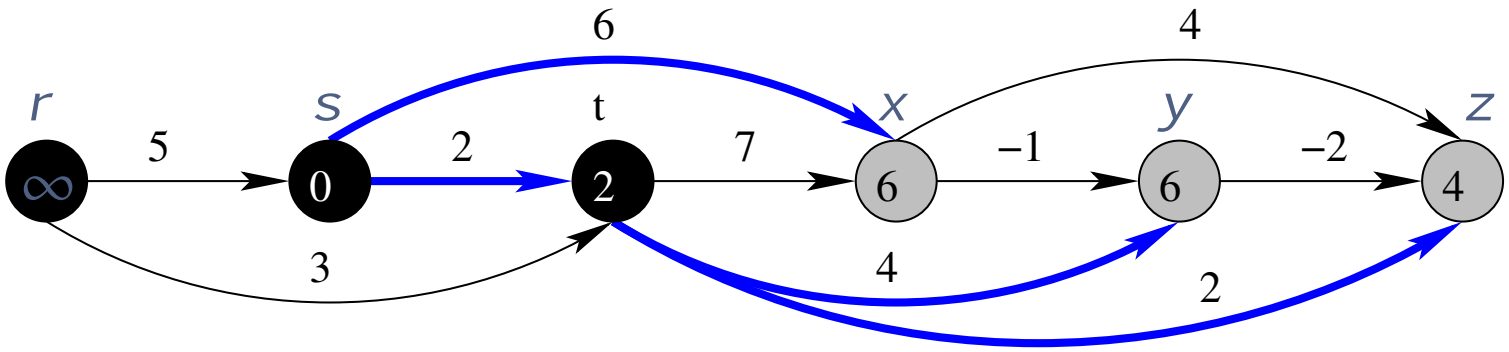
Exemplo



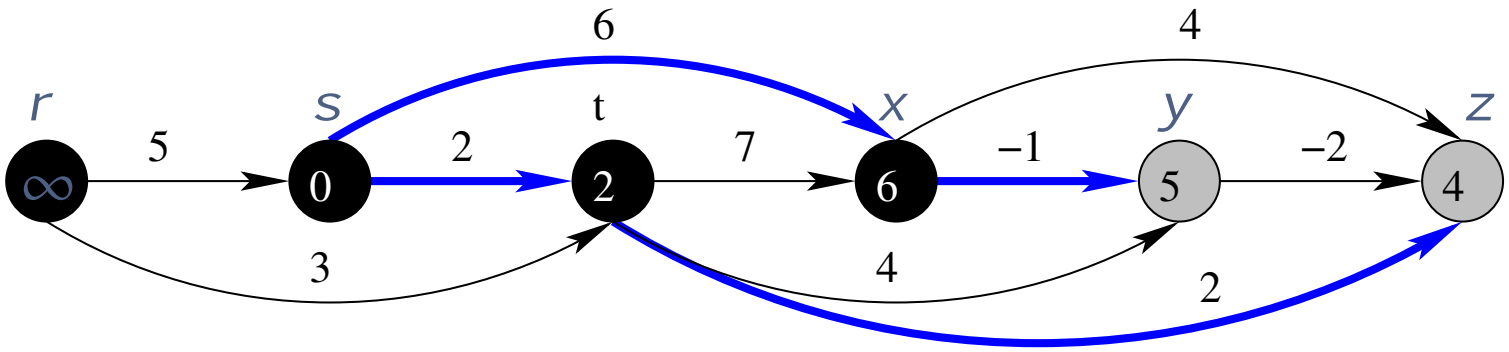
Exemplo



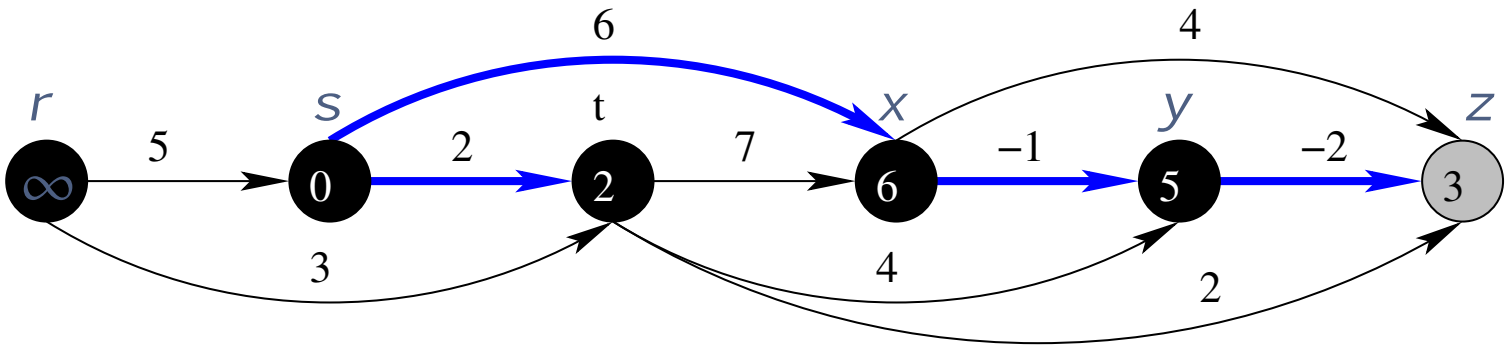
Exemplo



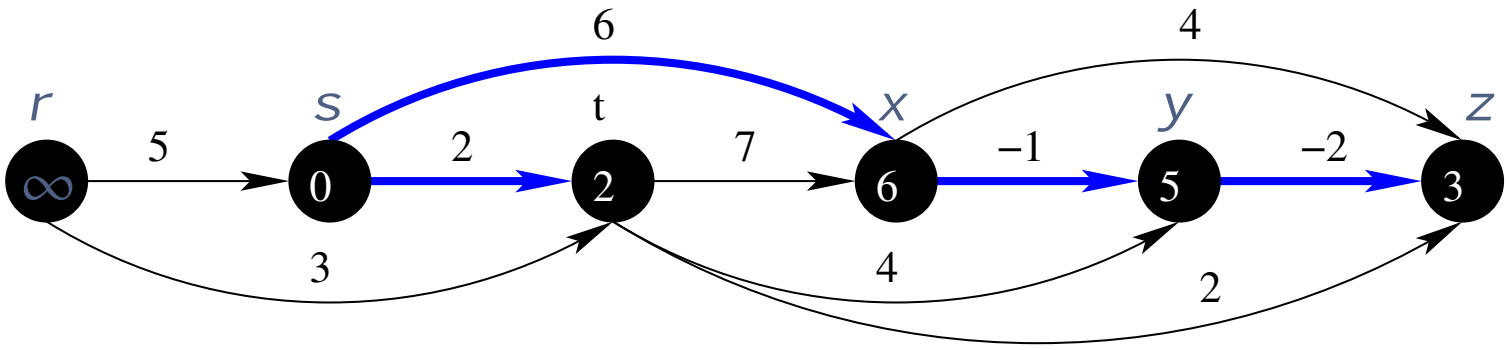
Exemplo



Exemplo



Exemplo



Complexidade

```
DAG-SHORTEST-PATHS(G, w, s)
1  Ordene topologicamente os vértices de G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  para cada vértice u na ordem topológica
4      para cada v ∈ Adj[u]
5          RELAX(u, v, w)
6  devolva d, π
```

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

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

Complexidade

```
DAG-SHORTEST-PATHS(G, w, s)
1  Ordene topologicamente os vértices de G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  para cada vértice u na ordem topológica
4      para cada v ∈ Adj[u]
5          RELAX(u, v, w)
6  devolva d, π
```

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

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

Complexidade

```
DAG-SHORTEST-PATHS(G, w, s)
1  Ordene topologicamente os vértices de G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  para cada vértice u na ordem topológica
4      para cada v ∈ Adj[u]
5          RELAX(u, v, w)
6  devolva d, π
```

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

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

Correção

- Há diversas estratégias para demonstrar que DAG-SHORTEST-PATHS esteja correto
- Vamos demonstrar algumas propriedades que valem porque o algoritmo é baseado em relaxação
- Essas propriedades também serão úteis para analisar os outros algoritmos depois

Correção

- Há diversas estratégias para demonstrar que DAG-SHORTEST-PATHS esteja correto
- Vamos demonstrar algumas propriedades que valem porque o algoritmo é baseado em relaxação
- Essas propriedades também serão úteis para analisar os outros algoritmos depois

Correção

- Há diversas estratégias para demonstrar que DAG-SHORTEST-PATHS esteja correto
- Vamos demonstrar algumas propriedades que valem porque o algoritmo é baseado em relaxação
- Essas propriedades também serão úteis para analisar os outros algoritmos depois

Propriedade de 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 π é um caminho de peso $d[v]$.

Propriedade de 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 π é um caminho de peso $d[v]$.

Propriedade de 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 π é um caminho de peso $d[v]$.

Propriedade de 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 π 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 $p = (v_0, v_1, \ldots, v_k)$ é um caminho mínimo de $s = v_0$ a v_k e relaxamos as arestas de p na ordem $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, 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 $p = (v_0, v_1, \ldots, v_k)$ é um caminho mínimo de $s = v_0$ a v_k e relaxamos as arestas de p na ordem $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, 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 $p = (v_0, v_1, \ldots, v_k)$ 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), $\ldots$, (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.

Correção de Dag-Shortest-Paths

- Seja v um vértice e suponha que $p = (v_0, v_1, \ldots, v_k)$ é um caminho mínimo de $s = v_0$ a $v = v_k$.
- Como v_0, v_1, ..., v_k aparecem em ordem na ordenação topológica, as arestas $(v_0, v_1), \ldots, (v_{k-1}, v_k)$ 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 π define uma árvore de caminhos mínimos (por quê?)

Correção de Dag-Shortest-Paths

- Seja v um vértice e suponha que $p = (v_0, v_1, \ldots, v_k)$ é um caminho mínimo de $s = v_0$ a $v = v_k$.
- Como v_0, v_1, ..., v_k aparecem em ordem na ordenação topológica, as arestas $(v_0, v_1), \ldots, (v_{k-1}, v_k)$ 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 π define uma árvore de caminhos mínimos (por quê?)

Correção de Dag-Shortest-Paths

- Seja v um vértice e suponha que $p = (v_0, v_1, \ldots, v_k)$ é um caminho mínimo de $s = v_0$ a $v = v_k$.
- Como v_0, v_1, ..., v_k aparecem em ordem na ordenação topológica, as arestas $(v_0, v_1), \ldots, (v_{k-1}, v_k)$ 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 π define uma árvore de caminhos mínimos (por quê?)

Correção de Dag-Shortest-Paths

- Seja v um vértice e suponha que $p = (v_0, v_1, \ldots, v_k)$ é um caminho mínimo de $s = v_0$ a $v = v_k$.
- Como v_0, v_1, ..., v_k aparecem em ordem na ordenação topológica, as arestas $(v_0, v_1), \ldots, (v_{k-1}, v_k)$ 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 π define uma árvore de caminhos mínimos (por quê?)

Correção de Dag-Shortest-Paths

- Seja v um vértice e suponha que $p = (v_0, v_1, \ldots, v_k)$ é um caminho mínimo de $s = v_0$ a $v = v_k$.
- Como v_0, v_1, ..., v_k aparecem em ordem na ordenação topológica, as arestas $(v_0, v_1), \ldots, (v_{k-1}, v_k)$ 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 π define uma árvore de caminhos mínimos (por quê?)

Perguntas

1. Como se resolve o problema de encontrar um caminho de peso máximo de s a t em um grafo direcionado acíclico (G, w)?

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 $C > 0$?

Perguntas

1. Como se resolve o problema de encontrar um caminho de peso máximo de s a t em um grafo direcionado acíclico (G, w)?

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 ∈ V[G]
2       d[v] ← ∞
3       π[v] ← NIL
4   d[s] ← 0
```

- $d[v]$ é o peso do menor caminho conhecido até o momento
- π induz uma árvore que testemunha d

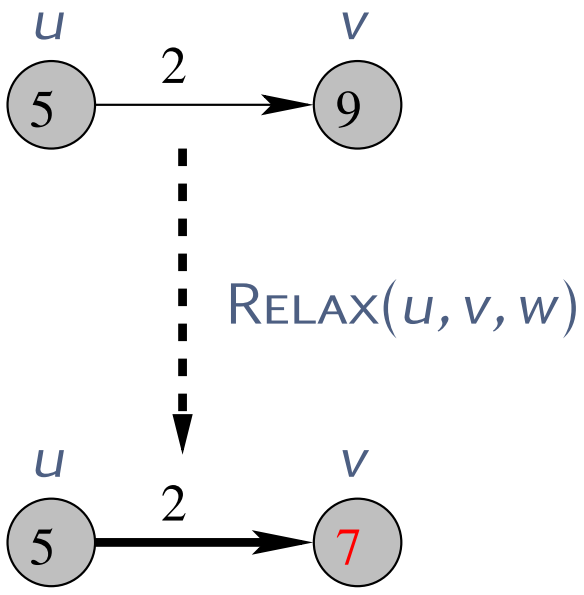
Revisão: algoritmos baseados em relaxação

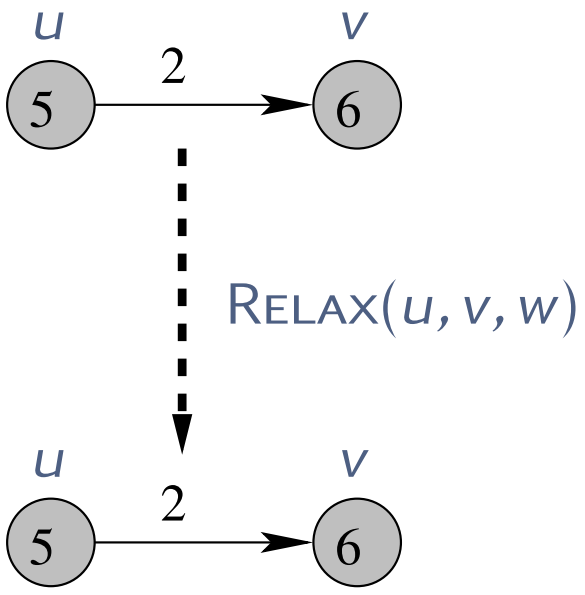
```
INITIALIZE-SINGLE-SOURCE(G, s)
1   para cada vértice v ∈ V[G]
2       d[v] ← ∞
3       π[v] ← NIL
4   d[s] ← 0
```

- $d[v]$ é o peso do menor caminho conhecido até o momento
- π induz uma árvore que testemunha d

Revisão: relaxação

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

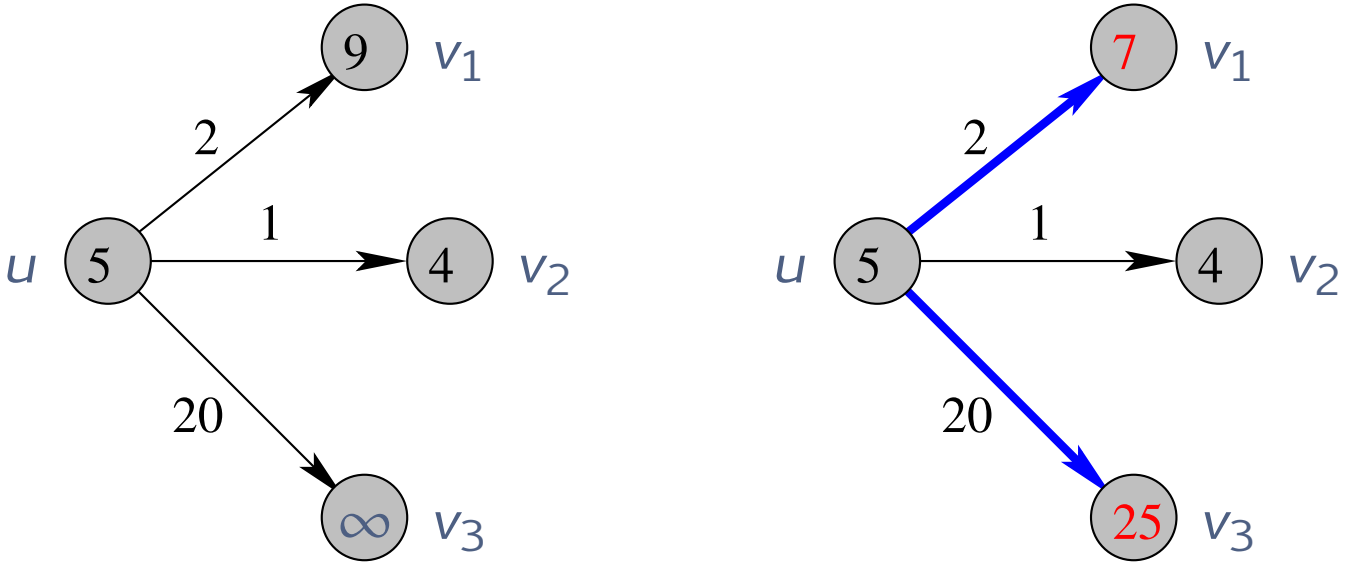





```
RELAX(u, v, w)
1   se d[v] > d[u] + w(u, v)
2       d[v] ← d[u] + w(u, v)
3       π[v] ← u
```

Revisão: relaxação dos vizinhos

Em cada iteração o algoritmo seleciona um vértice u e para cada vizinho v de u aplica RELAX(u, v, w).



RELAX(u, v, w)
1 se $d[v] > d[u] + w(u, v)$
2 $\quad d[v] \leftarrow d[u] + w(u, v)$
3 $\quad \pi[v] \leftarrow u$

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**

Algoritmo de Dijkstra

Entrada:

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

Saída:

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

Algoritmo de Dijkstra

```
DIJKSTRA(G, w, s)
1   INITIALIZE-SINGLE-SOURCE(G, s)
2   S ← ∅
3   Q ← V[G]
4   enquanto Q ≠ ∅
5       u ← EXTRACT-MIN(Q)
6       S ← S ∪ {u}
7       para cada vértice v ∈ Adj[u]
8           RELAX(u, v, w)
9   devolva d, π
```

- 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, w, s)
1   INITIALIZE-SINGLE-SOURCE(G, s)
2   S ← ∅
3   Q ← V[G]
4   enquanto Q ≠ ∅
5       u ← EXTRACT-MIN(Q)
6       S ← S ∪ {u}
7       para cada vértice v ∈ Adj[u]
8           RELAX(u, v, w)
9   devolva d, π
```

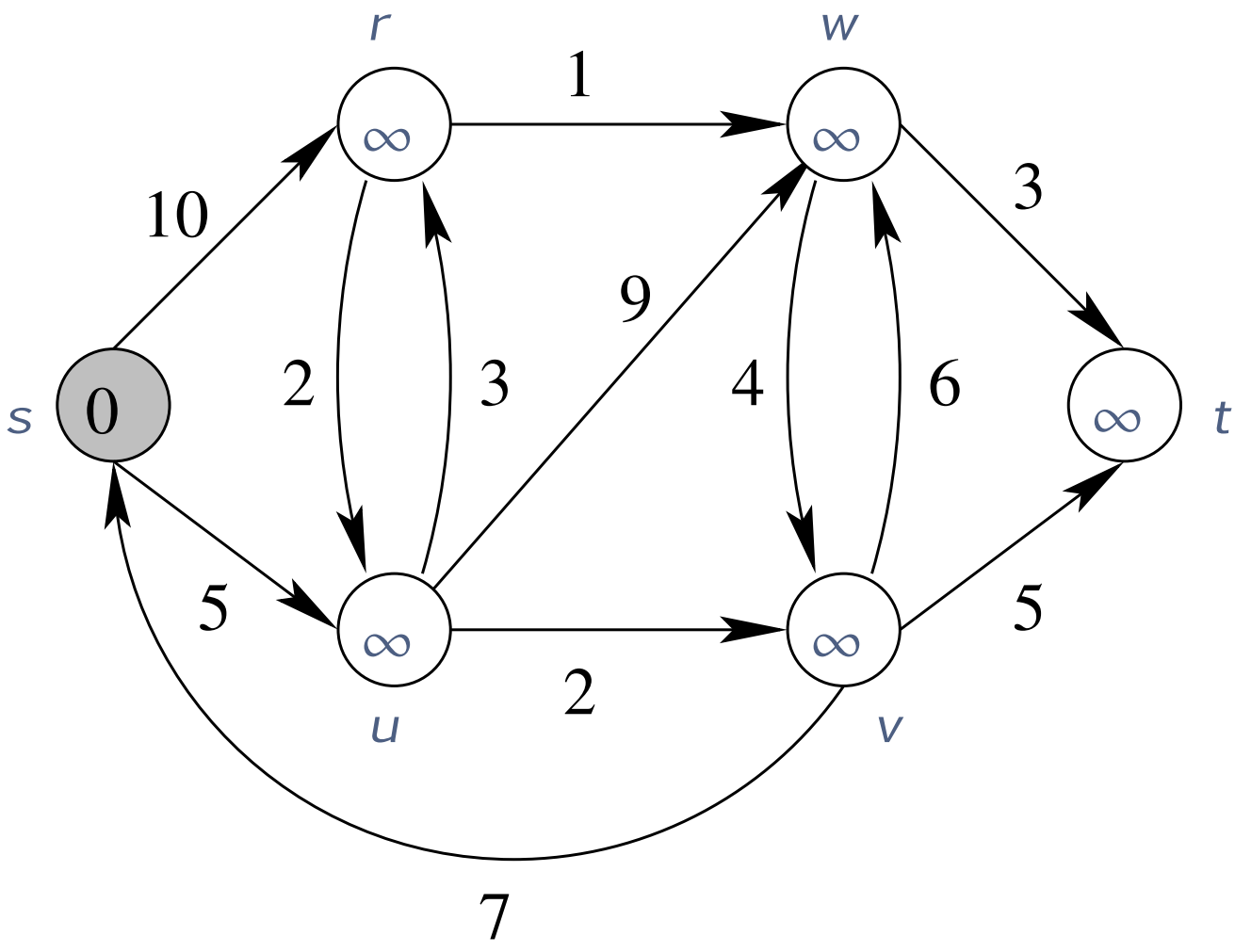
- O conjunto Q é implementado como uma fila de prioridade com chave d.
- O conjunto S não é realmente necessário, mas simplifica a análise do algoritmo.

Algoritmo de Dijkstra

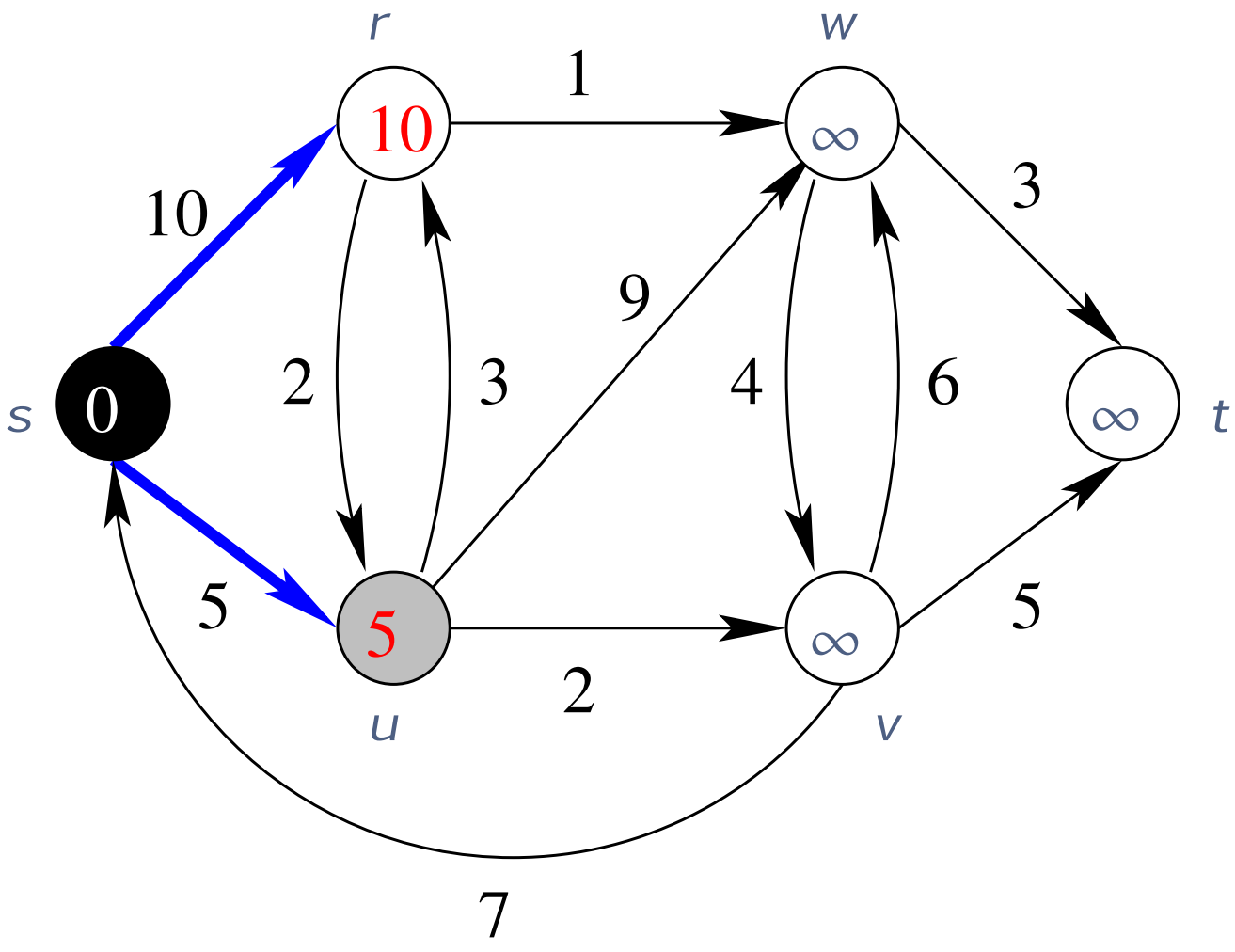
```
DIJKSTRA(G, w, s)
1   INITIALIZE-SINGLE-SOURCE(G, s)
2   S ← ∅
3   Q ← V[G]
4   enquanto Q ≠ ∅
5       u ← EXTRACT-MIN(Q)
6       S ← S ∪ {u}
7       para cada vértice v ∈ Adj[u]
8           RELAX(u, v, w)
9   devolva d, π
```

- 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.

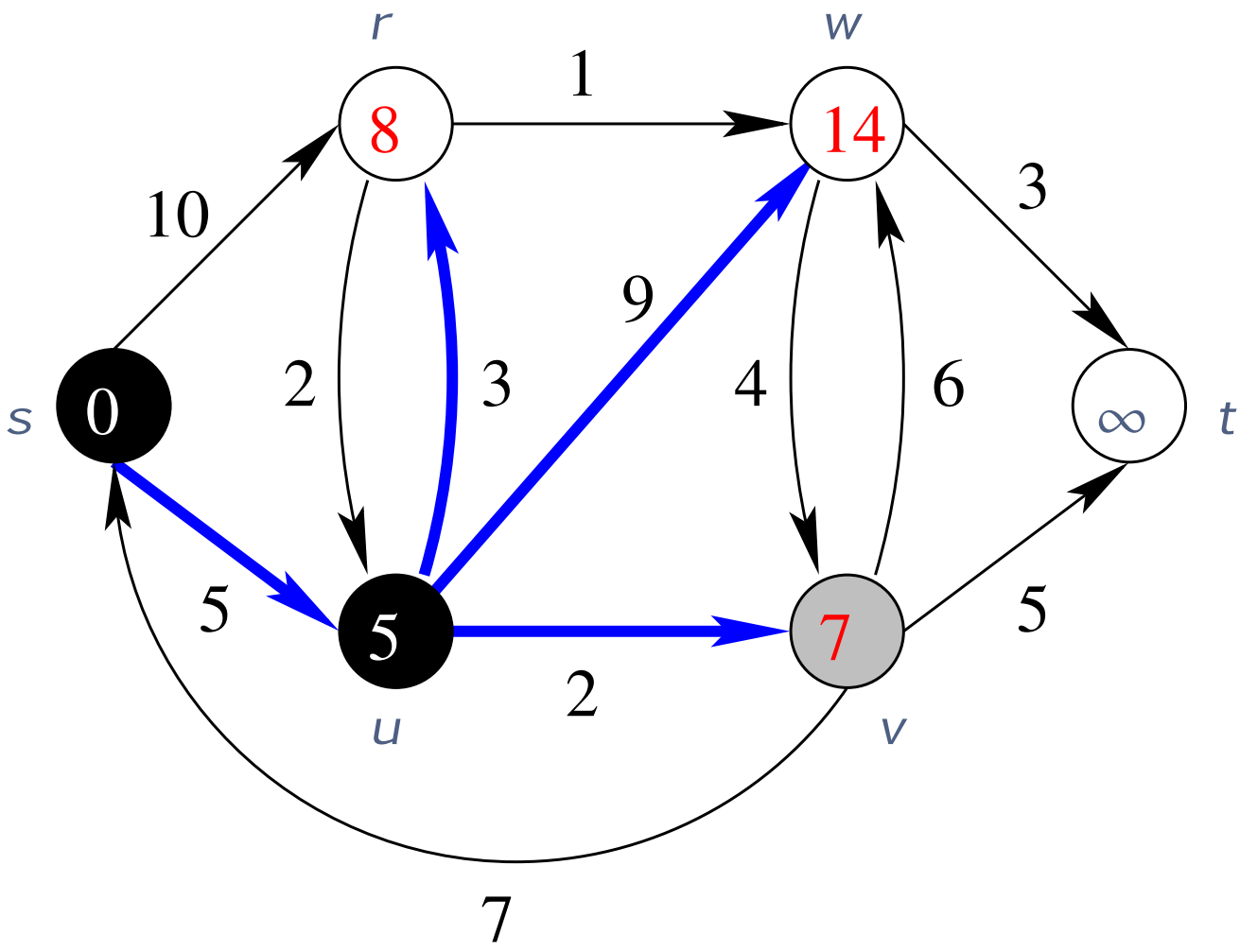
Exemplo (CLRS modificado)



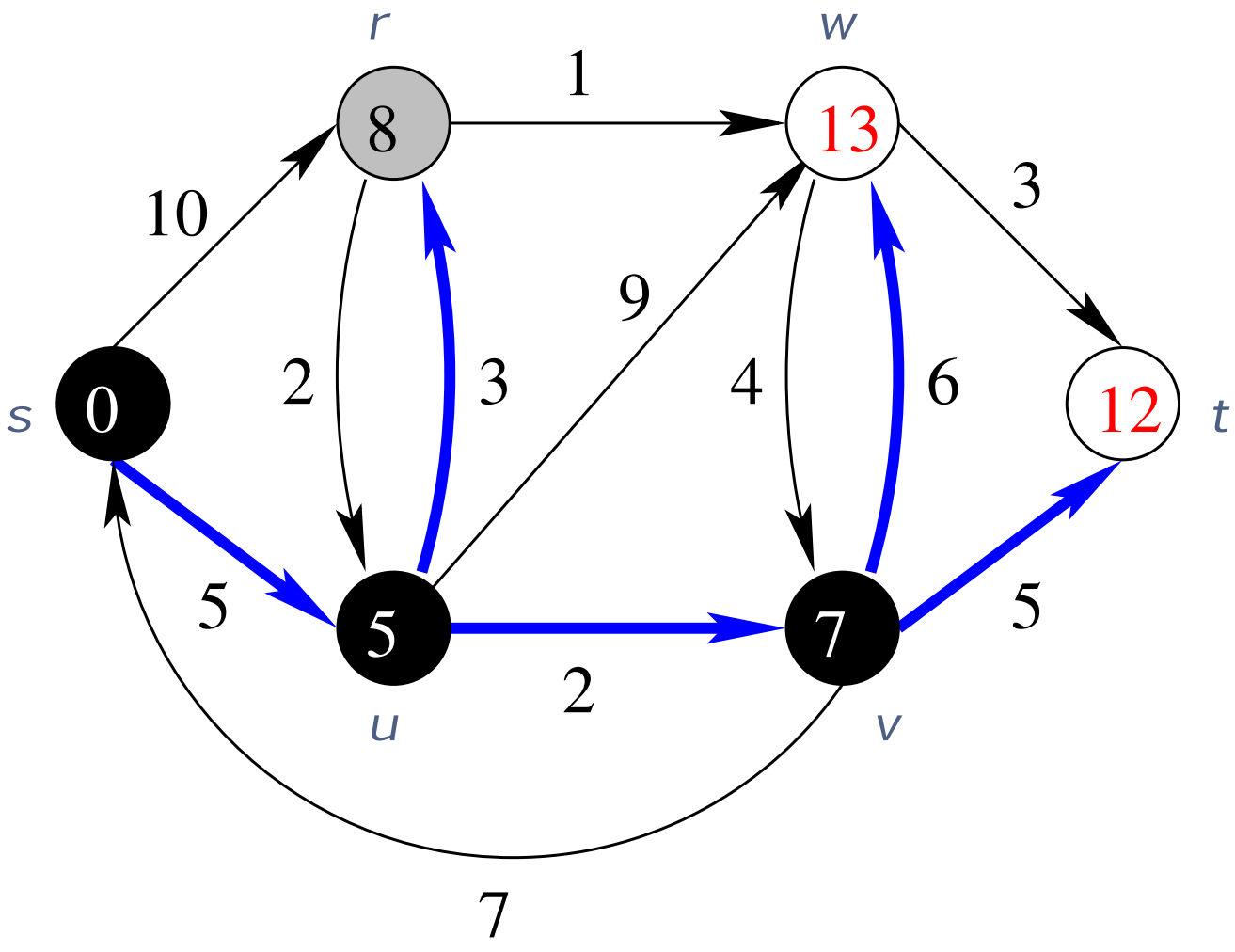
Exemplo (CLRS modificado)



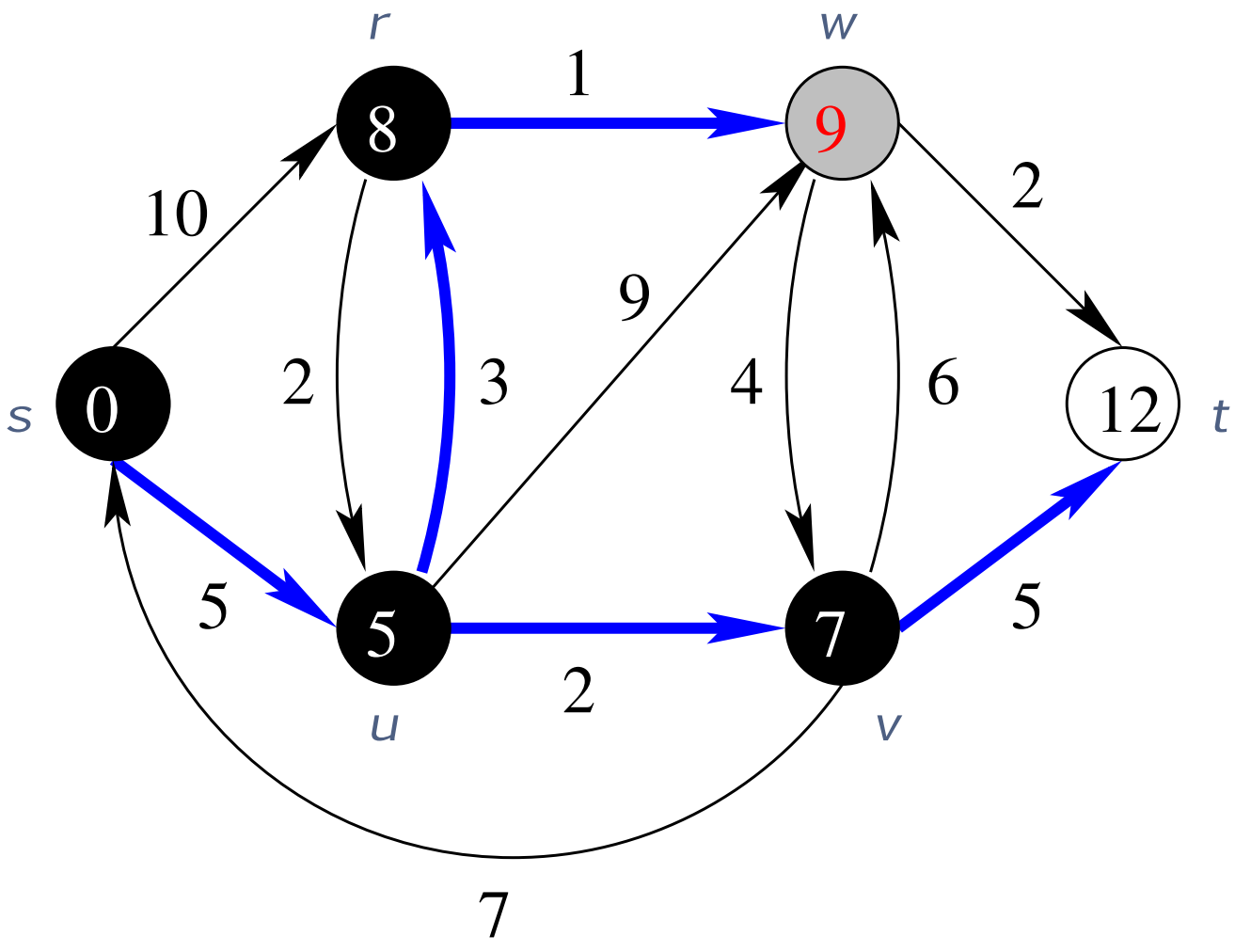
Exemplo (CLRS modificado)



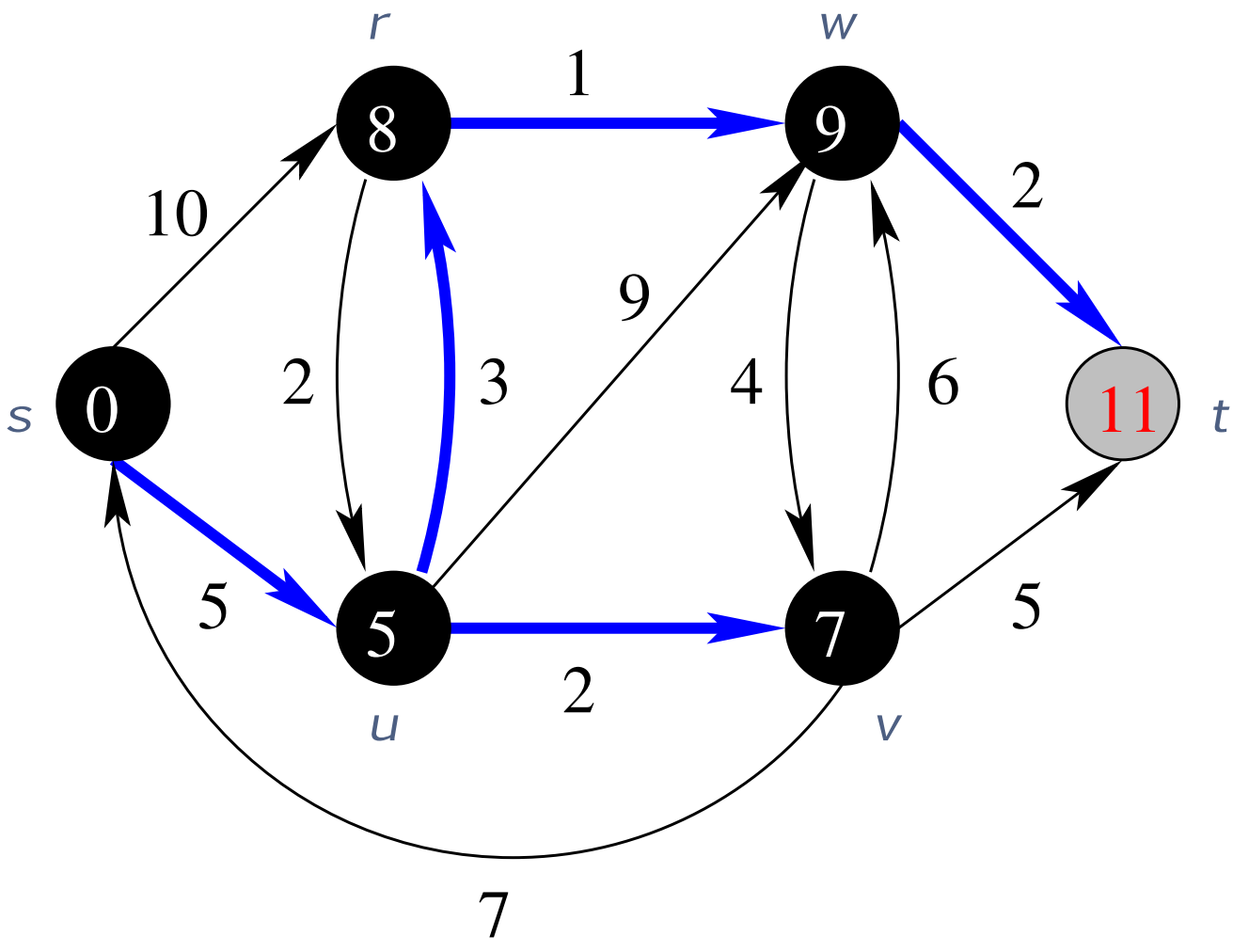
Exemplo (CLRS modificado)



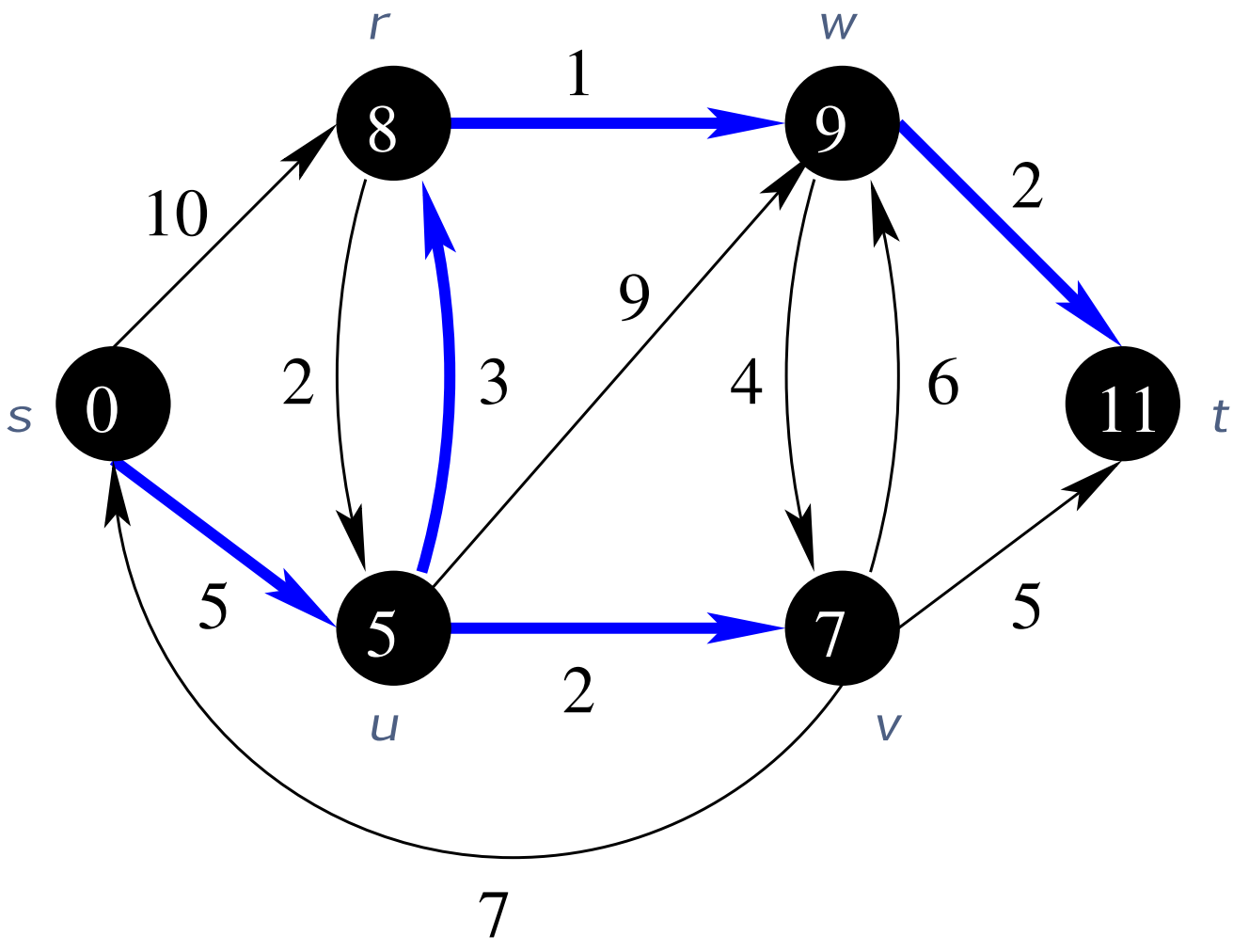
Exemplo (CLRS modificado)



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. π induz uma árvore de caminhos mínimos.

Observe que

- DIJKSTRA é baseado em relaxação
- pela propriedade dos subgrafo de predecessores, sabemos que o vetor π induz 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. π induz uma árvore de caminhos mínimos.

Observe que

- DIJKSTRA é baseado em relaxação
- pela propriedade dos subgrafo de predecessores, sabemos que o vetor π induz 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. π induz uma árvore de caminhos mínimos.

Observe que

- DIJKSTRA é baseado em relaxação
- pela propriedade dos subgrafo de predecessores, sabemos que o vetor π induz 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. π induz uma árvore de caminhos mínimos.

Observe que

- DIJKSTRA é baseado em relaxação
- pela propriedade dos subgrafo de predecessores, sabemos que o vetor π induz 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. π induz uma árvore de caminhos mínimos.

Observe que

- DIJKSTRA é baseado em relaxação
- pela propriedade dos subgrafo de predecessores, sabemos que o vetor π induz 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. π induz uma árvore de caminhos mínimos.

Observe que

- Dijkstra é baseado em relaxação
- pela propriedade dos subgrafo de predecessores, sabemos que o vetor π induz uma árvore que testemunha d
- assim, basta mostrar que de fato $d[v] = \text{dist}(s, v)$

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 instante $d[u] = \text{dist}(s, u)$

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 instante $d[u] = \text{dist}(s, u)$

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 instante $d[u] = \text{dist}(s, u)$

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 instante $d[u] = \text{dist}(s, u)$

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 instante $d[u] = \text{dist}(s, u)$

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 instante $d[u] = \text{dist}(s, u)$

Invariante

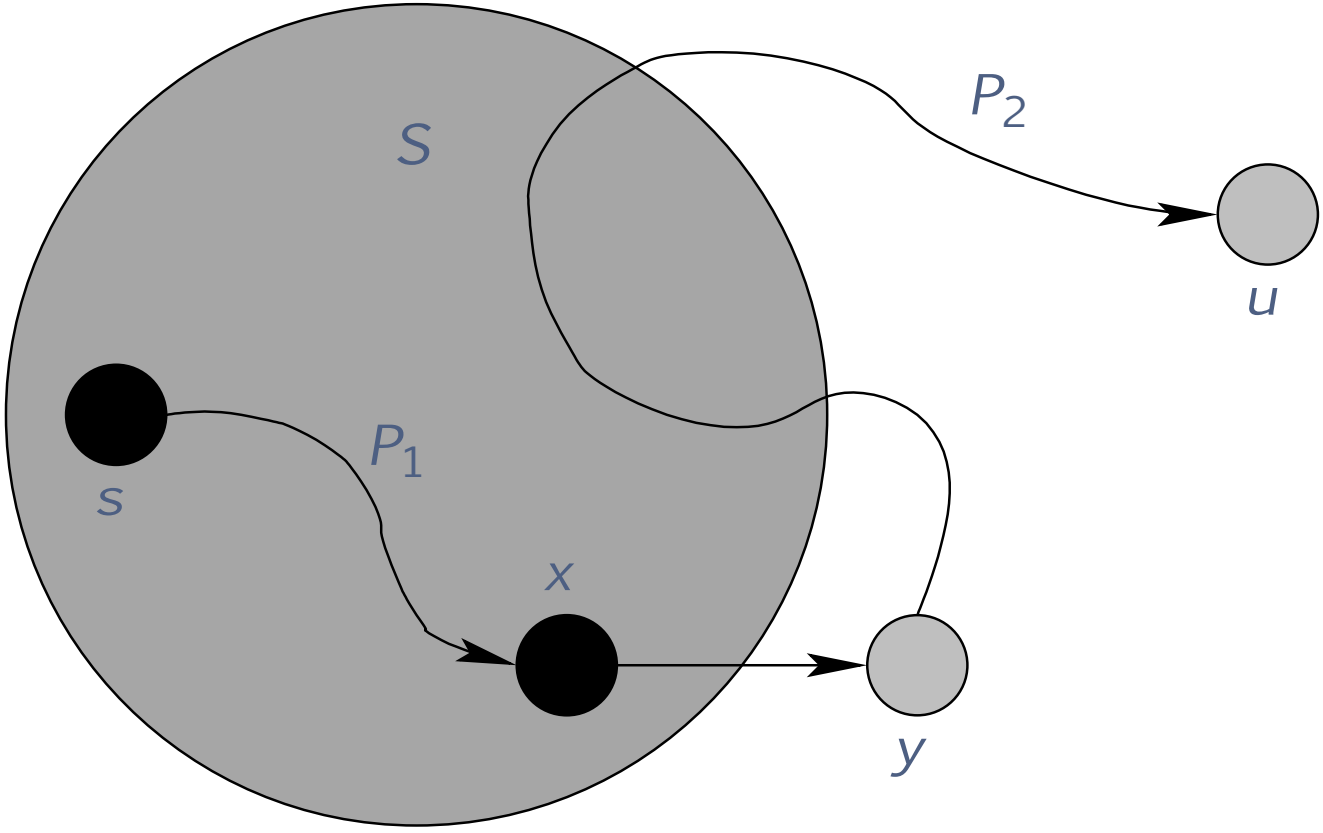
Iremos mostrar que no no início de cada iteração da linha 4 no algoritmo DIJKSTRA, vale $d[x] = \text{dist}(s, x)$ para cada $x \in S$.

- **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 instante $d[u] = \text{dist}(s, u)$

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

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

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

Concluímos 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

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

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

Concluímos 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

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

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

Concluímos 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

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

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

Concluímos 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

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

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

Concluímos 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

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

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

Concluímos 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

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

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

Concluímos 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

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

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

Concluímos 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

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

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

Concluímos 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

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

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

Concluímos 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

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

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

Concluímos 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

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

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

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

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 $d[v] = \text{dist}(s, v)$.

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

Na demonstração, supomos que não há arestas negativas

- se a hipótese não valer, o algoritmo de Dijkstra pode falhar
- encontre um grafo com arestas negativas para o qual o algoritmo de Dijkstra não funciona (exercício)
- existe um exemplo com 4 vértices, com apenas uma aresta negativa e sem ciclos de peso negativo

Dijkstra precisa de arestas com peso não negativo

Na demonstração, supomos que não há arestas negativas

- se a hipótese não valer, o algoritmo de Dijkstra pode falhar
- encontre um grafo com arestas negativas para o qual o algoritmo de Dijkstra não funciona (exercício)
- existe um exemplo com 4 vértices, com apenas uma aresta negativa e sem ciclos de peso negativo

Dijkstra precisa de arestas com peso não negativo

Na demonstração, supomos que não há arestas negativas

- se a hipótese não valer, o algoritmo de Dijkstra pode falhar
- encontre um grafo com arestas negativas para o qual o algoritmo de Dijkstra não funciona (exercício)
- existe um exemplo com 4 vértices, com apenas uma aresta negativa e sem ciclos de peso negativo

Dijkstra precisa de arestas com peso não negativo

Na demonstração, supomos que não há arestas negativas

- se a hipótese não valer, o algoritmo de Dijkstra pode falhar
- encontre um grafo com arestas negativas para o qual o algoritmo de Dijkstra não funciona (exercício)
- existe um exemplo com 4 vértices, com apenas uma aresta negativa e sem ciclos de peso negativo

Complexidade de tempo

```
DIJKSTRA(G, w, s)
1   INITIALIZE-SINGLE-SOURCE(G, s)
2   S ← ∅
3   Q ← V[G]
4   enquanto Q ≠ ∅
5       u ← EXTRACT-MIN(Q)
6       S ← S ∪ {u}
7       para cada vértice v ∈ Adj[u]
8           RELAX(u, v, w)
9   devolva d, π
```

Depende de como a fila de prioridade Q é implementada:

- Operações INSERT, EXTRACT-MIN, DECREASE-KEY

Complexidade de tempo

```
DIJKSTRA(G, w, s)
1   INITIALIZE-SINGLE-SOURCE(G, s)
2   S ← ∅
3   Q ← V[G]
4   enquanto Q ≠ ∅
5       u ← EXTRACT-MIN(Q)
6       S ← S ∪ {u}
7       para cada vértice v ∈ Adj[u]
8           RELAX(u, v, w)
9   devolva d, π
```

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: $O(|V| \times \textsc{Insert} + |V| \times \textsc{Extract-Min} + |E| \times \textsc{Decrease-Key})$

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

Algoritmo de Bellman-Ford

Arestas vs. ciclos de peso negativo

- O algoritmo de Dijkstra resolve o Problema dos Caminhos Mínimos quando (G, w) não tem arestas de peso negativo.
- Quando (G, w) 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, w) não tem arestas de peso negativo.
- Quando (G, w) 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, w) não tem arestas de peso negativo.
- Quando (G, w) 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 ∈ V[G]
2       d[v] ← ∞
3       π[v] ← NIL
4   d[s] ← 0
```

- $d[v]$ é o peso do menor caminho conhecido até o momento
- π induz uma árvore que testemunha d

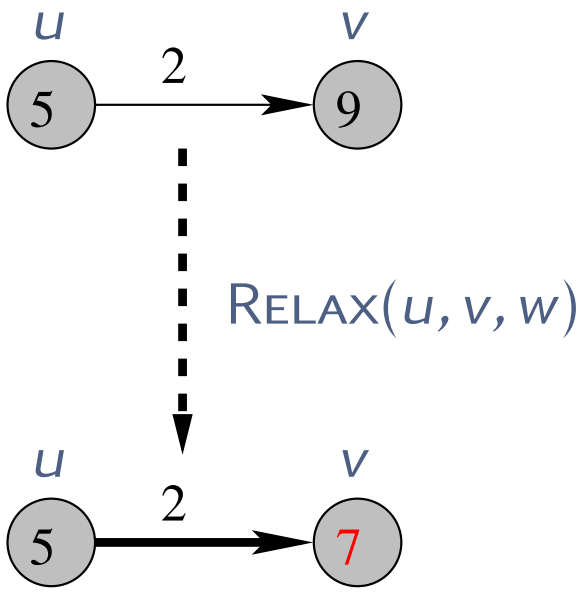
Revisão: algoritmos baseados em relaxação

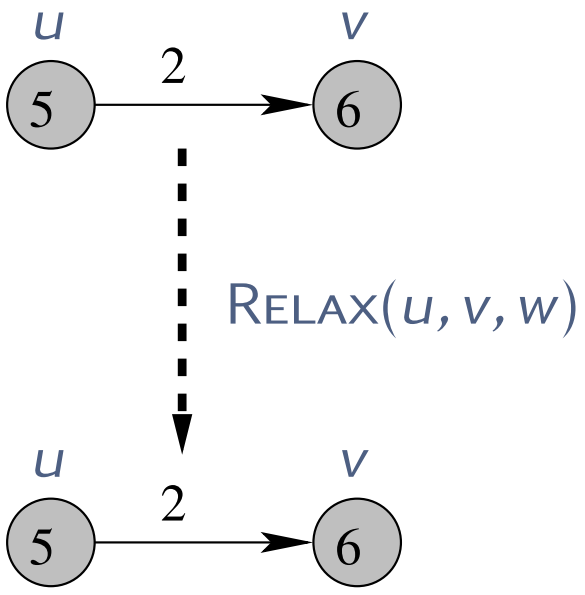
```
INITIALIZE-SINGLE-SOURCE(G, s)
1   para cada vértice v ∈ V[G]
2       d[v] ← ∞
3       π[v] ← NIL
4   d[s] ← 0
```

- $d[v]$ é o peso do menor caminho conhecido até o momento
- π induz uma árvore que testemunha d

Revisão: relaxação

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

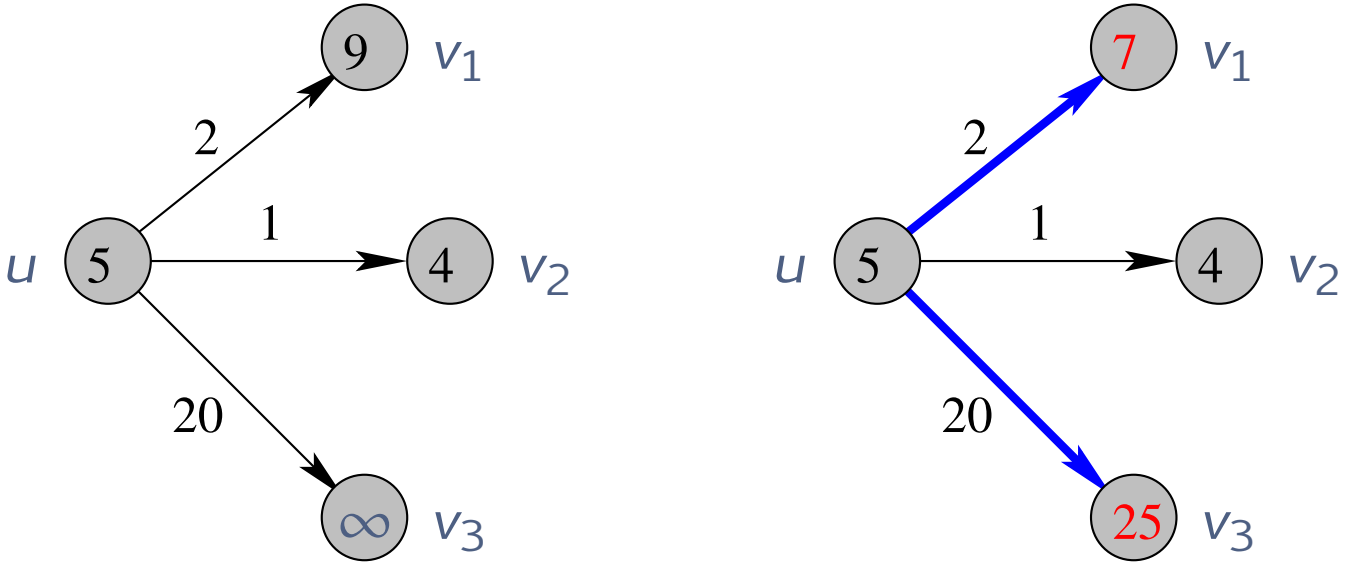





```
RELAX(u, v, w)
1   se d[v] > d[u] + w(u, v)
2       d[v] ← d[u] + w(u, v)
3       π[v] ← u
```

Revisão: relaxação dos vizinhos

Em cada iteração o algoritmo seleciona um vértice u e para cada vizinho v de u aplica RELAX(u, v, w).



RELAX(u, v, w)
1 se $d[v] > d[u] + w(u, v)$
2 $\quad d[v] \leftarrow d[u] + w(u, v)$
3 $\quad \pi[v] \leftarrow u$

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 $(v_0, v_1, \ldots, v_k)$, queremos relaxar (v_0, v_1), (v_1, v_2), $\ldots$, (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?)
   $\Rightarrow (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ 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 $(v_0, v_1, \ldots, v_k)$, queremos relaxar (v_0, v_1), (v_1, v_2), $\ldots$, (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?)
   $\Rightarrow (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ 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 $(v_0, v_1, \ldots, v_k)$, queremos relaxar (v_0, v_1), (v_1, v_2), $\ldots$, (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?)
   $\Rightarrow (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ 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 $(v_0, v_1, \ldots, v_k)$, queremos relaxar (v_0, v_1), (v_1, v_2), $\ldots$, (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?)
   $\Rightarrow (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ 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 $(v_0, v_1, \ldots, v_k)$, queremos relaxar (v_0, v_1), (v_1, v_2), $\ldots$, (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?)
   $\Rightarrow (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ 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 $(v_0, v_1, \ldots, v_k)$, queremos relaxar (v_0, v_1), (v_1, v_2), $\ldots$, (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), $\ldots$, (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 $(v_0, v_1, \ldots, v_k)$, queremos relaxar (v_0, v_1), (v_1, v_2), $\ldots$, (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?)
   $\Rightarrow (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ 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 $(v_0, v_1, \ldots, v_k)$, queremos relaxar (v_0, v_1), (v_1, v_2), $\ldots$, (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?)
   $\Rightarrow (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ 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 $(v_0, v_1, \ldots, v_k)$, queremos relaxar (v_0, v_1), (v_1, v_2), $\ldots$, (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?)
   $\Rightarrow (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ 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 $(v_0, v_1, \ldots, v_k)$, queremos relaxar (v_0, v_1), (v_1, v_2), $\ldots$, (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), $\ldots$, (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 $(v_0, v_1, \ldots, v_k)$, queremos relaxar (v_0, v_1), (v_1, v_2), $\ldots$, (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), $\ldots$, (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 $(v_0, v_1, \ldots, v_k)$, queremos relaxar (v_0, v_1), (v_1, v_2), $\ldots$, (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), $\ldots$, (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 w nas arestas
- origem s.

Saída:

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

O algoritmo de Bellman-Ford

Entrada:

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

Saída:

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

O algoritmo de Bellman-Ford

Entrada:

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

Saída:

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

O algoritmo de Bellman-Ford

Entrada:

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

Saída:

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

O algoritmo de Bellman-Ford

Entrada:

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

Saída:

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

O algoritmo de Bellman-Ford

```
BELLMAN-FORD(G, w, s)
1   INITIALIZE-SINGLE-SOURCE(G, s)
2   para i ← 1 até |V[G]| − 1
3       para cada aresta (u, v) ∈ E[G]
4           RELAX(u, v, w)
5   para cada aresta (u, v) ∈ E[G]
6       se d[v] > d[u] + w(u, v)
7           devolva FALSE
8   devolva TRUE, d, π
```

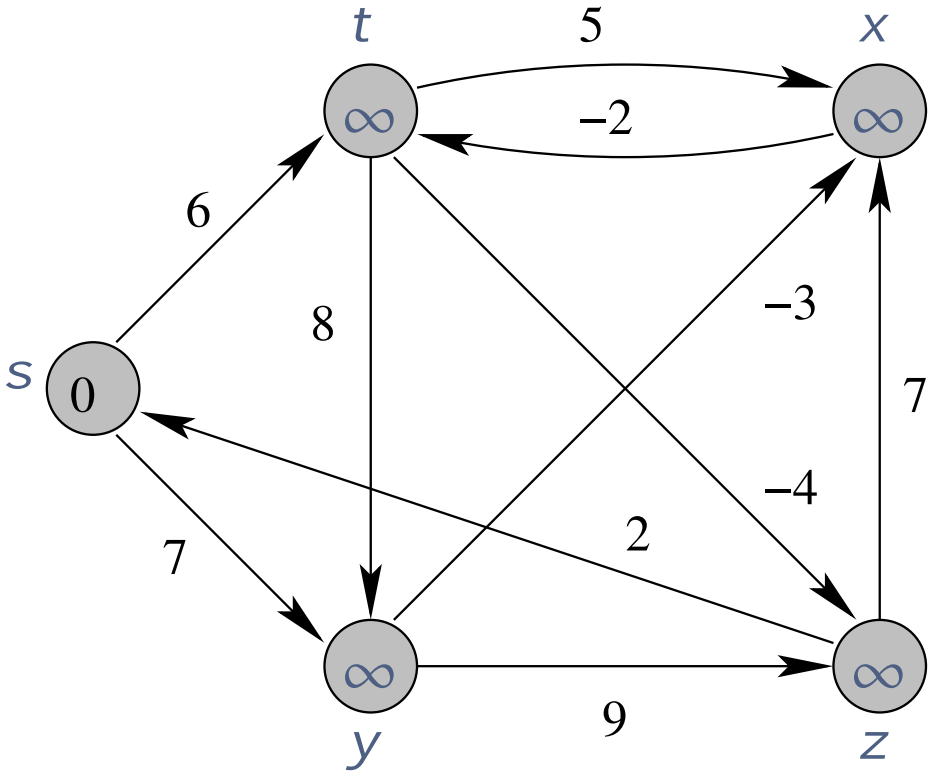
Complexidade de tempo: $O(VE)$

O algoritmo de Bellman-Ford

```
BELLMAN-FORD(G, w, s)
1   INITIALIZE-SINGLE-SOURCE(G, s)
2   para i ← 1 até |V[G]| − 1
3       para cada aresta (u, v) ∈ E[G]
4           RELAX(u, v, w)
5   para cada aresta (u, v) ∈ E[G]
6       se d[v] > d[u] + w(u, v)
7           devolva FALSE
8   devolva TRUE, d, π
```

Complexidade de tempo: $O(VE)$

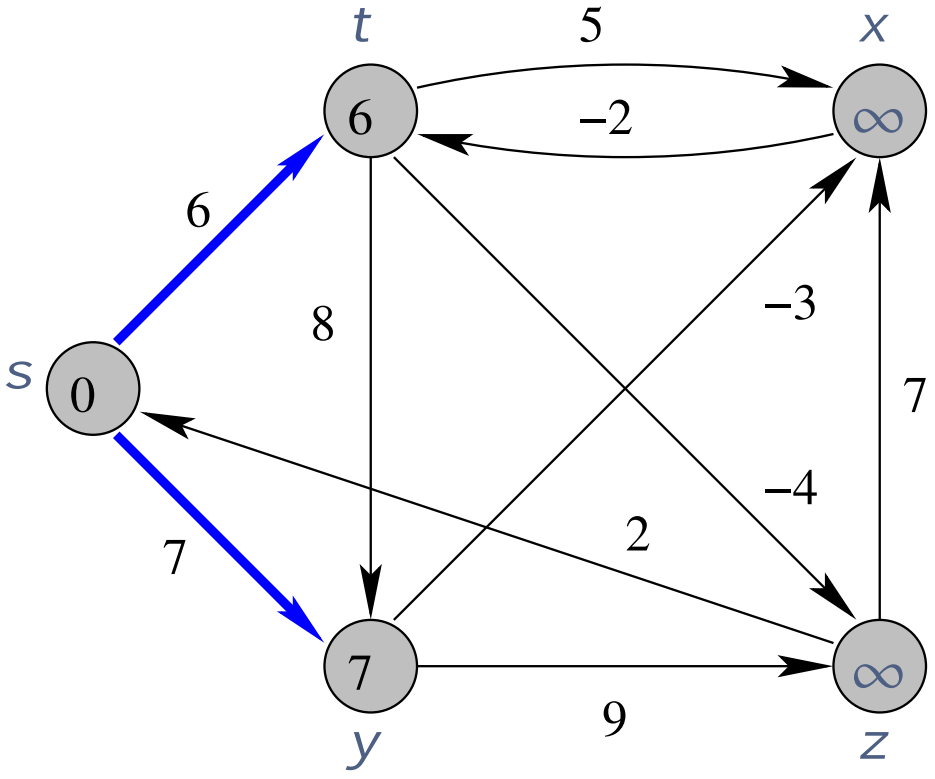
Exemplo (CLRS)



Ordem:

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

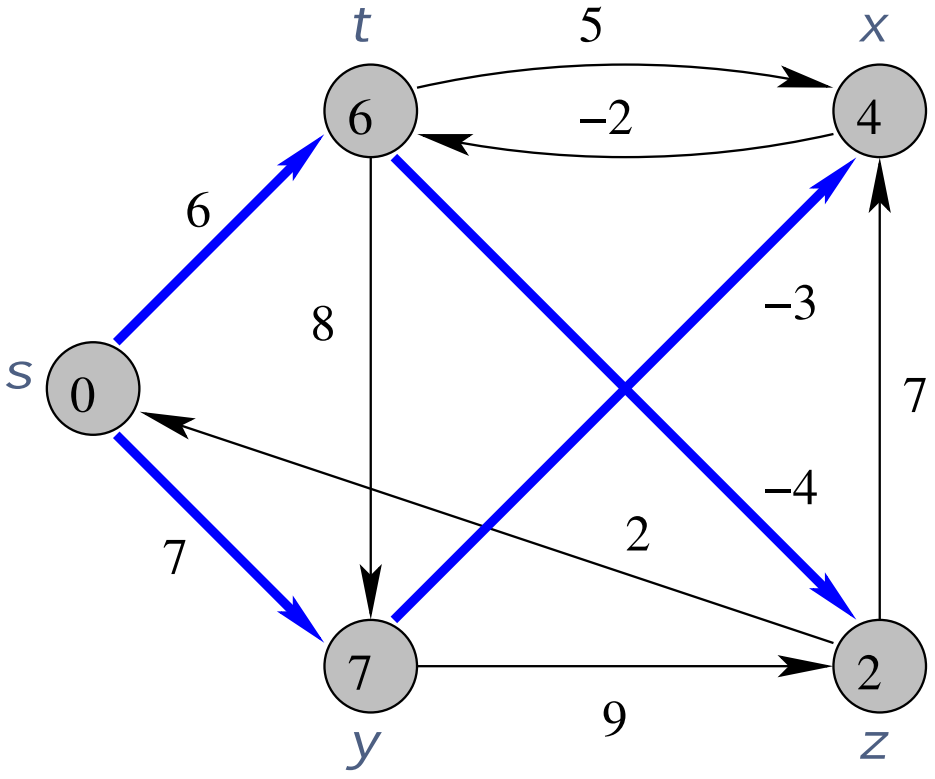
Exemplo (CLRS)



Ordem:

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

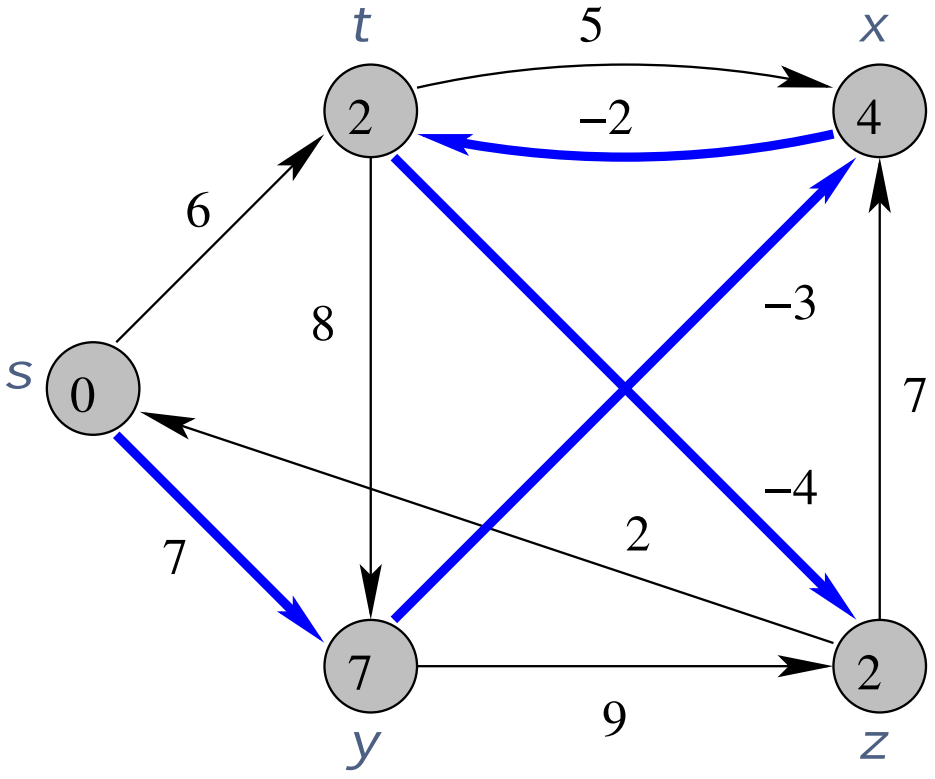
Exemplo (CLRS)



Ordem:

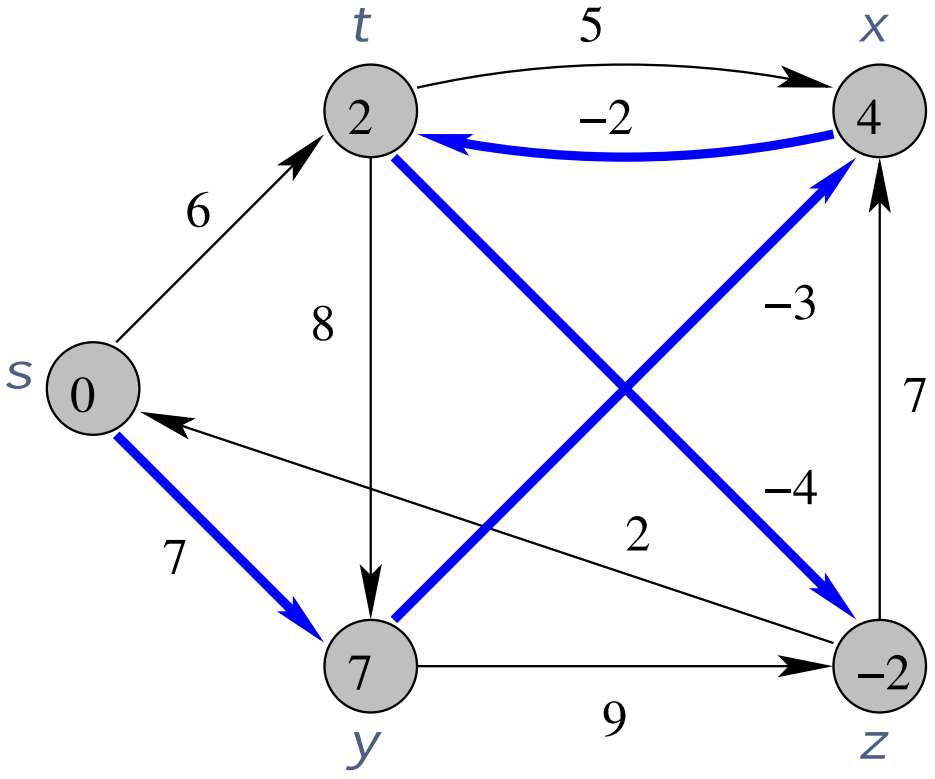
$(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.

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:

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

Correção do algoritmo Bellman-Ford

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 π 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 π após o primeiro laço:

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

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

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

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

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

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

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

Queremos mostrar que o algoritmo devolve FALSE:

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

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

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

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

Correção do algoritmo Bellman-Ford

Suponha agora que (G, w) contenha ciclo negativo alcançável por s.

Queremos mostrar que o algoritmo devolve FALSE:

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

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

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

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

Correção do algoritmo Bellman-Ford

Suponha agora que (G, w) contenha ciclo negativo alcançável por s.

Queremos mostrar que o algoritmo devolve FALSE:

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

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

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

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

Correção do algoritmo Bellman-Ford

Suponha agora que (G, w) contenha ciclo negativo alcançável por s.

Queremos mostrar que o algoritmo devolve FALSE:

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

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

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

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

Correção do algoritmo Bellman-Ford

Suponha agora que (G, w) contenha ciclo negativo alcançável por s.

Queremos mostrar que o algoritmo devolve FALSE:

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

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

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

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

Correção do algoritmo Bellman-Ford

Suponha agora que (G, w) contenha ciclo negativo alcançável por s.

Queremos mostrar que o algoritmo devolve FALSE:

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

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

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

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

Correção do algoritmo Bellman-Ford

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

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i))$$

$$= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

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

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

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

Correção do algoritmo Bellman-Ford

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

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i))$$

$$= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

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

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

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

Correção do algoritmo Bellman-Ford

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

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i))$$
$$= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

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

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

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

Correção do algoritmo Bellman-Ford

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

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i))$$
$$= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

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

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

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

Correção do algoritmo Bellman-Ford

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

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i))$$

$$= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

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

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

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

Correção do algoritmo Bellman-Ford

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

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i))$$

$$= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

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

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

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

Correção do algoritmo Bellman-Ford

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

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i))$$

$$= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

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

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

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

Correção do algoritmo Bellman-Ford

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

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i))$$
$$= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

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

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

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

Sistemas de restrições de diferenças

Sistemas de restrições de diferenças

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

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

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 $x_1, x_2, \ldots, x_n$ que satisfaçam:

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

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 $x_1, x_2, \ldots, x_n$ que satisfaçam:

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

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 $x_1, x_2, \ldots, x_n$ que satisfaçam:

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

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:

- $x = (-5, -3, 0, -1, -4)$,
- $x' = (0, 2, 5, 4, 1)$,...

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:

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

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:

- $x = (-5, -3, 0, -1, -4)$,
- $x' = (0, 2, 5, 4, 1)$, ...

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:

- $x = (-5, -3, 0, -1, -4)$,
- $x' = (0, 2, 5, 4, 1)$,...

Sistemas de restrições de diferenças

Lema

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

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

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

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

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 $x_j - x_i \leq b_k$
     - $\Rightarrow$ 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 $x_j - x_i \leq b_k$
     - $\Rightarrow$ 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 $x_j - x_i \leq b_k$

     $\Rightarrow$ 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 $x_j - x_i \leq b_k$

     $\Rightarrow$ 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 $x_j - x_i \leq b_k$
     - $\Rightarrow$ 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 $x_j - x_i \leq b_k$
     - $\Rightarrow$ 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

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

E associamos pesos:

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

Grafo de restrições

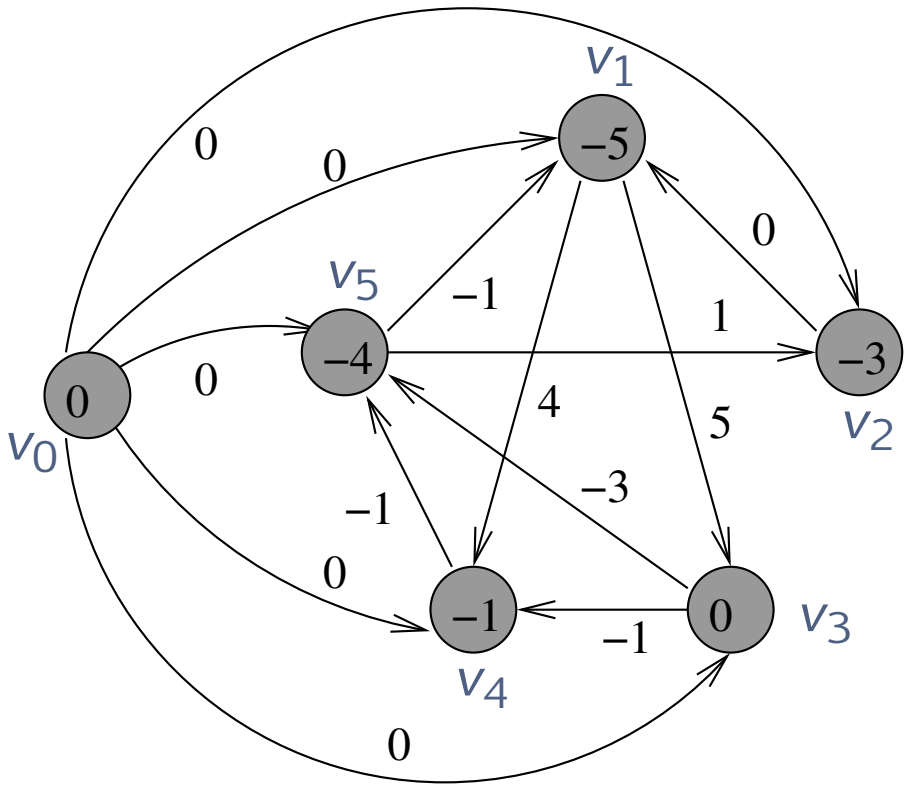
Formalmente, dado o sistema $Ax \leq b$ de restrições de diferença, construímos o grafo $G = (V, E)$ tal que

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

E associamos pesos:

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

Grafo de restrições



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

Grafo de restrições

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

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

  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:

- Executamo 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 $x = (d[v_1], d[v_2], \ldots, d[v_n])$

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:

- Executamo 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 $x = (d[v_1], d[v_2], \ldots, d[v_n])$

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:

- Executamo 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 $x = (d[v_1], d[v_2], \ldots, d[v_n])$

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:

- Executamo 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 $x = (d[v_1], d[v_2], \ldots, d[v_n])$

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:

- Executamo 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 $x = (d[v_1], d[v_2], \ldots, d[v_n])$

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:

- Executamo 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 $x = (d[v_1], d[v_2], \ldots, d[v_n])$

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:

- Executamo 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 $x = (d[v_1], d[v_2], \ldots, d[v_n])$

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:

- Executamo 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 $x = (d[v_1], d[v_2], \ldots, d[v_n])$

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, w) 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, w) não possui arestas negativas, usamos Dijkstra:

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

- Se (G, w) possui arestas negativas, usamos Bellman-Ford:

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

Algoritmos para grafos esparsos

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

- Se (G, w) não possui arestas negativas, usamos Dijkstra:

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

- Se (G, w) possui arestas negativas, usamos Bellman-Ford:

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

Algoritmos para grafos esparsos

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

- Se (G, w) não possui arestas negativas, usamos Dijkstra:

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

- Se (G, w) possui arestas negativas, usamos Bellman-Ford:

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

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 basado em programação dinâmica
- resolve o problema em tempo $O(V^3)$
- supomos que G é completo:
  $\Rightarrow$ se (i, j) não é aresta, definimos $w(i, j) = \infty$

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 basado em programação dinâmica
- resolve o problema em tempo $O(V^3)$
- supomos que G é completo:
  $\Rightarrow$ se (i, j) não é aresta, definimos $w(i, j) = \infty$

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 basado em programação dinâmica
- resolve o problema em tempo $O(V^3)$
- supomos que G é completo:
  $\Rightarrow$ se (i, j) não é aresta, definimos $w(i, j) = \infty$

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 basado em programação dinâmica
- resolve o problema em tempo $O(V^3)$
- supomos que G é completo:
  $\Rightarrow$ se (i, j) não é aresta, definimos $w(i, j) = \infty$

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 basado em programação dinâmica
- resolve o problema em tempo $O(V^3)$
- supomos que G é completo:
  $\Rightarrow$ se (i, j) não é aresta, definimos $w(i, j) = \infty$

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 basado em programação dinâmica
- resolve o problema em tempo $O(V^3)$
- supomos que G é completo:
  $\Rightarrow$ se (i, j) não é aresta, definimos $w(i, j) = \infty$

Subproblema

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

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

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

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 $V = \{1, 2, \ldots, n\}$

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

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

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 $V = \{1, 2, \ldots, n\}$

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

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

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 $V = \{1, 2, \ldots, n\}$

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

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

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 $V = \{1, 2, \ldots, n\}$

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

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

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 $\{1,\ldots,k-1\}$
- 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 $\{1,\ldots,k-1\}$
- 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 $\{1,\ldots,k-1\}$
- 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

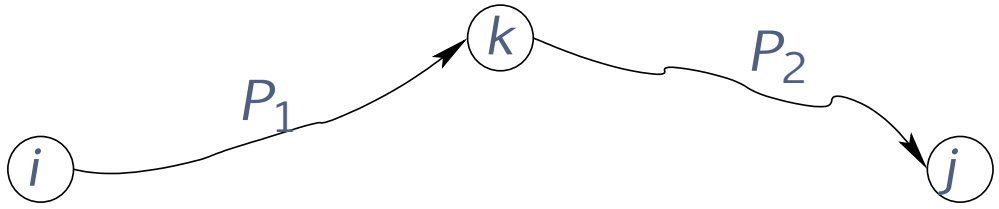
- um caminho k-interno P de i até j que tem custo mínimo
- e outros caminhos $(k-1)$-internos

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

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

Estrutura de um caminho mínimo

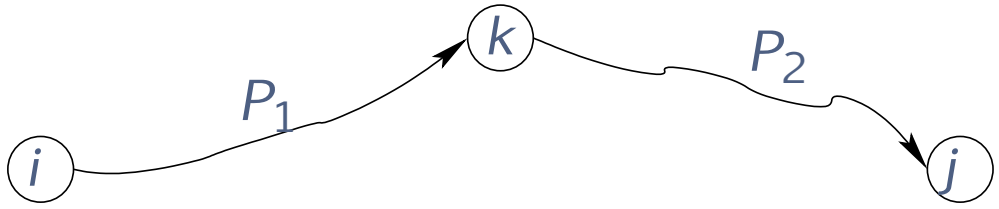
Caso 2: Se k é um vértice interno de P então P pode ser dividido em dois caminhos P_1 (com início em i e fim em k) e P_2 (com início em k e fim em j).



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

Estrutura de um caminho mínimo

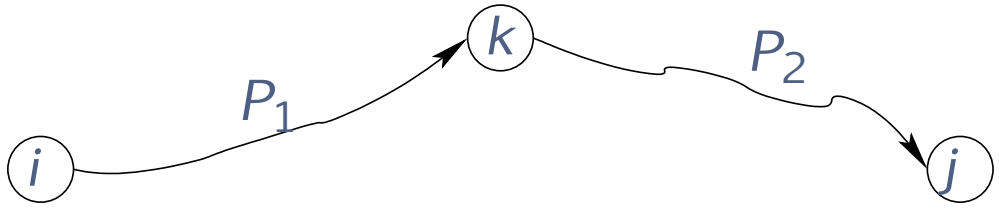
Caso 2: Se k é um vértice interno de P então P pode ser dividido em dois caminhos P_1 (com início em i e fim em k) e P_2 (com início em k e fim em j).



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

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

Recorrência para caminhos mínimos

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

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

Obtemos a seguinte recorrência:

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

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

- Calculamos as matrizes $D^{(k)} = (d_{ij}^{(k)})$ para $k = 1, 2 \ldots, n$.
- 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 $d_{ij}^{(0)} = w(i,j)$
- senão, caímos em algum dos dois casos anteriores

Obtemos a seguinte recorrência:

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

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

- Calculamos as matrizes $D^{(k)} = (d_{ij}^{(k)})$ para $k = 1, 2 \ldots, n$.
- 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 $d_{ij}^{(0)} = w(i,j)$
- senão, caímos em algum dos dois casos anteriores

Obtemos a seguinte recorrência:

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

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

- Calculamos as matrizes $D^{(k)} = (d_{ij}^{(k)})$ para $k = 1, 2 \ldots, n$.
- 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 $d_{ij}^{(0)} = w(i,j)$
- senão, caímos em algum dos dois casos anteriores

Obtemos a seguinte recorrência:

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

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

- Calculamos as matrizes $D^{(k)} = (d_{ij}^{(k)})$ para $k = 1, 2 \ldots, n$.
- 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 $d_{ij}^{(0)} = w(i,j)$
- senão, caímos em algum dos dois casos anteriores

Obtemos a seguinte recorrência:

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

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

- Calculamos as matrizes $D^{(k)} = (d_{ij}^{(k)})$ para $k = 1, 2 \ldots, n$.
- 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 $d_{ij}^{(0)} = w(i,j)$
- senão, caímos em algum dos dois casos anteriores

Obtemos a seguinte recorrência:

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

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

- Calculamos as matrizes $D^{(k)} = (d_{ij}^{(k)})$ para $k = 1, 2 \ldots, n$.
- 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 $d_{ij}^{(0)} = w(i,j)$
- senão, caímos em algum dos dois casos anteriores

Obtemos a seguinte recorrência:

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

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

- Calculamos as matrizes $D^{(k)} = (d_{ij}^{(k)})$ para $k = 1, 2 \ldots, n$.
- 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 $d_{ij}^{(0)} = w(i,j)$
- senão, caímos em algum dos dois casos anteriores

Obtemos a seguinte recorrência:

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

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

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

Algoritmo de Floyd-Warshall

Entrada:

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

Saída:

- a matriz $D^{(n)}$

```
FLOYD-WARSHALL(W, n)
1  D^(0) ← W
2  para k ← 1 até n
3      para i ← 1 até n
4          para j ← 1 até n
5              d_ij^(k) ← 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 $W = (w(i,j))$ com dimensões $n \times n$

Saída:

- a matriz $D^{(n)}$

```
FLOYD-WARSHALL(W, n)
1  D^(0) ← W
2  para k ← 1 até n
3      para i ← 1 até n
4          para j ← 1 até n
5              d_ij^(k) ← 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 $W = (w(i,j))$ com dimensões $n \times n$

Saída:

- a matriz $D^{(n)}$

```
FLOYD-WARSHALL(W, n)
1  D^(0) ← W
2  para k ← 1 até n
3      para i ← 1 até n
4          para j ← 1 até n
5              d_ij^(k) ← 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) π_{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 $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$

Quando $k = 0$:

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

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) π_{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 $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$

Quando $k = 0$:

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

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) π_{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 $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$

Quando $k = 0$:

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

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) π_{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 $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$

Quando $k = 0$:

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

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) π_{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 $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$

Quando $k = 0$:

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

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?

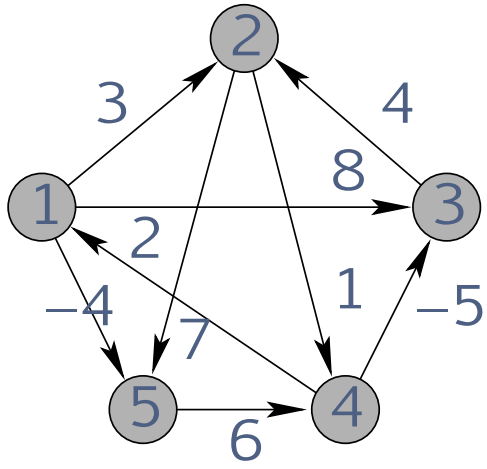
Se $k \geq 1$:

- 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,

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

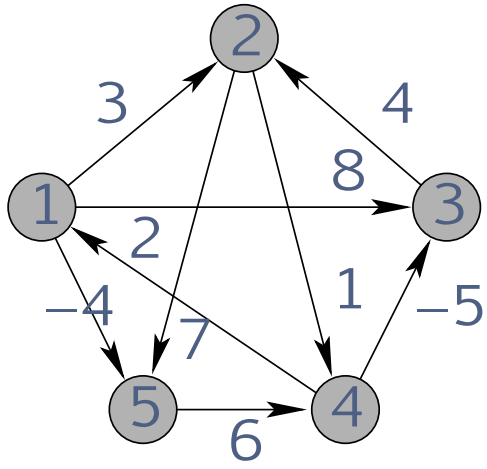
Exemplo



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

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

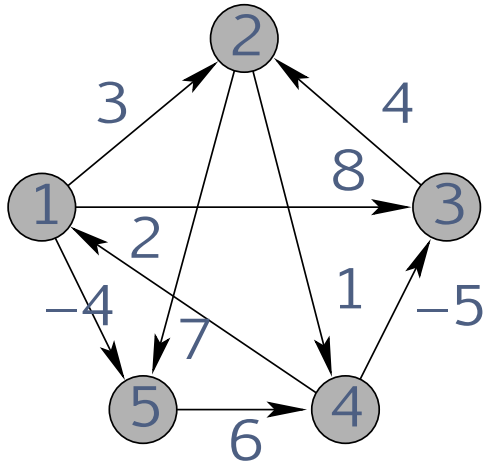
Exemplo



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

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

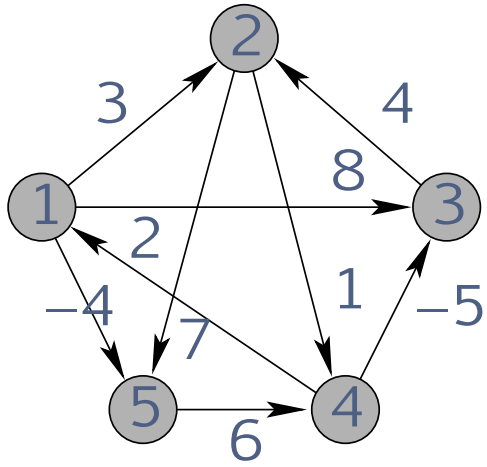
Exemplo



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

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

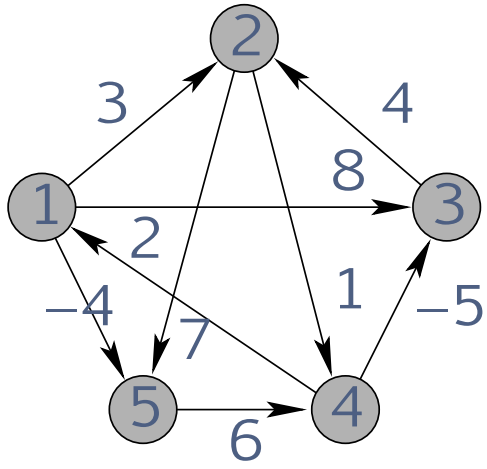
Exemplo



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

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

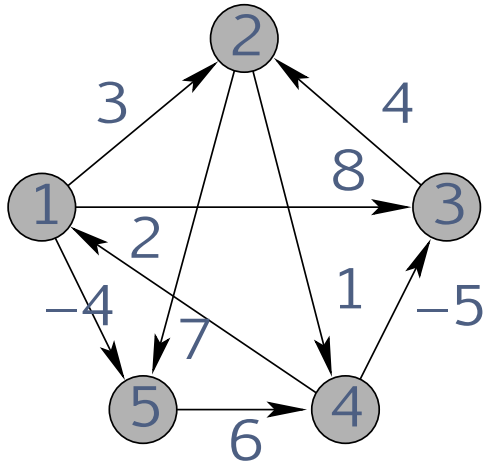
Exemplo



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

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

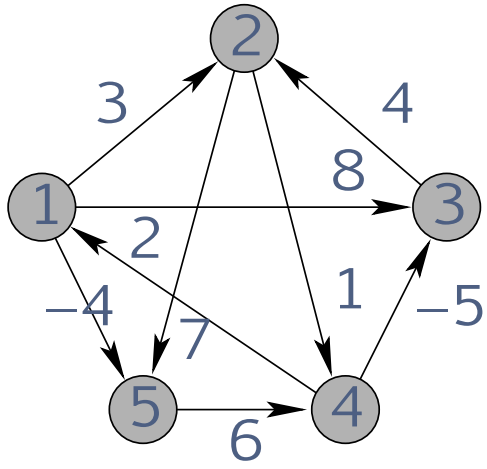
Exemplo



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

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

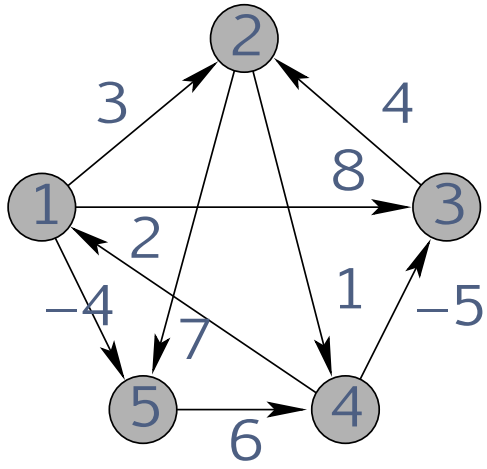
Exemplo



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

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

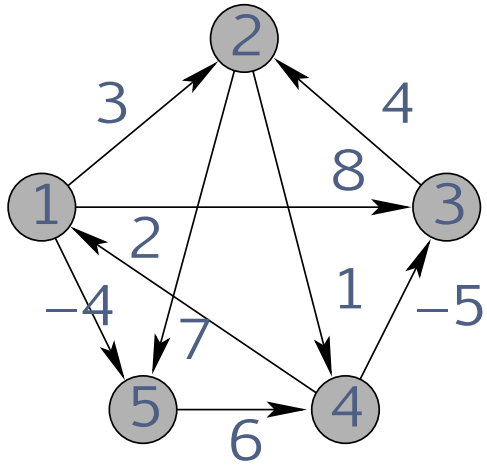
Exemplo



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

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

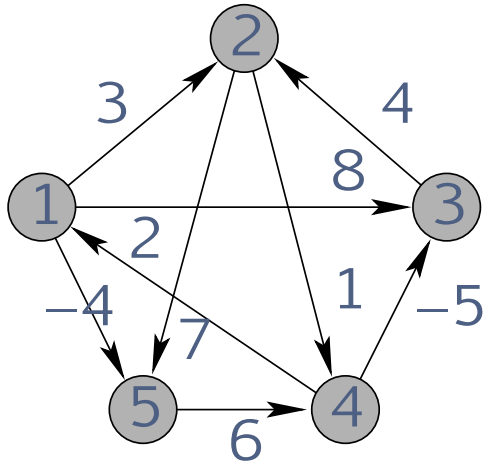
Exemplo



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

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

Exemplo



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

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

Fecho transitivo de grafos direcionados

Fecho transitivo de grafos direcionados

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

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 $V = \{1, 2, \ldots, n\}$.

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

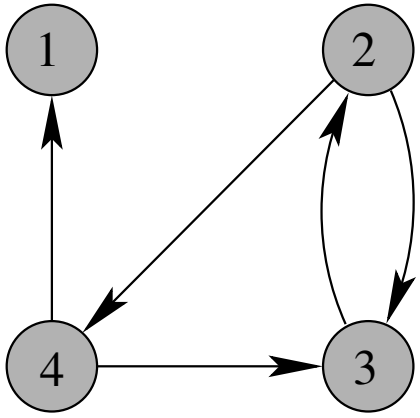
$$E^* = \{(i, j) : \text{existe um caminho de } i \text{ a } j \text{ em } G\}.$$

Fecho transitivo de grafos direcionados

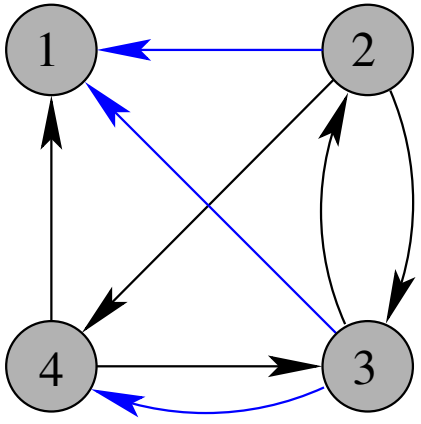
Seja $G = (V, E)$ um grafo direcionado com $V = \{1, 2, \ldots, n\}$.

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

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



G



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 $k \geq 1$,

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

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 $k \geq 1$,

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

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 $k \geq 1$,

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

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 $k \geq 1$,

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

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 $k \geq 1$,

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

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 $k \geq 1$,

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

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

Algoritmo de Transitive-Closure

Entrada:

- matriz de adjacência A de G

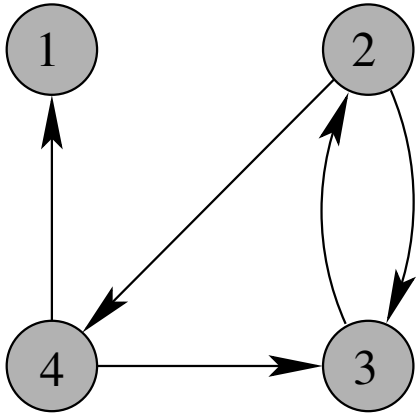
Saída:

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

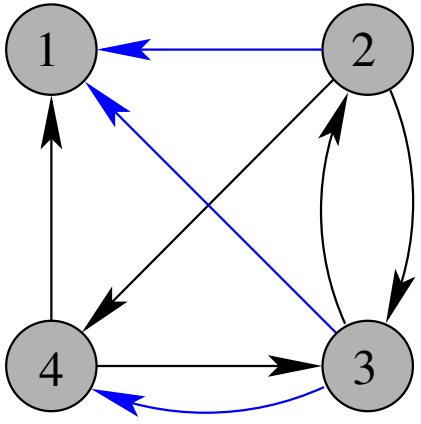
```
TRANSITIVE-CLOSURE(A, n)
1  T^(0) ← A + I_n
2  para k ← 1 até n
3      para i ← 1 até n
4          para j ← 1 até n
5              t_ij^(k) ← t_ij^(k−1) ∨ (t_ik^(k−1) ∧ t_kj^(k−1))
6  devolva T^(n)
```

Complexidade: $O(V^3)$

Exemplo



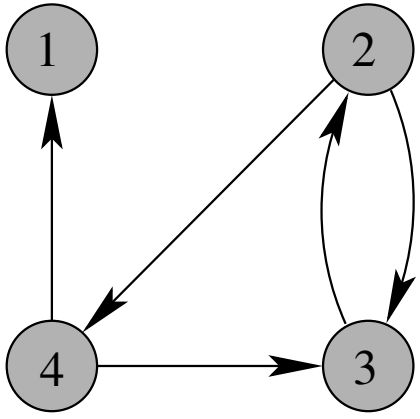
G



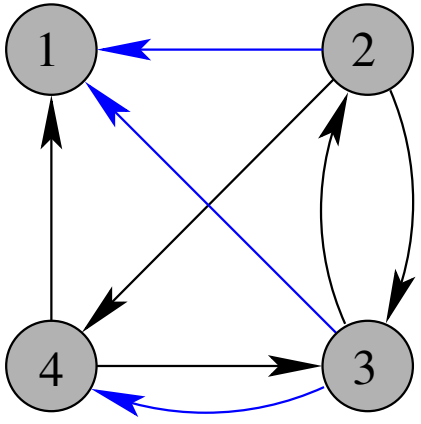
G^*

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

Exemplo



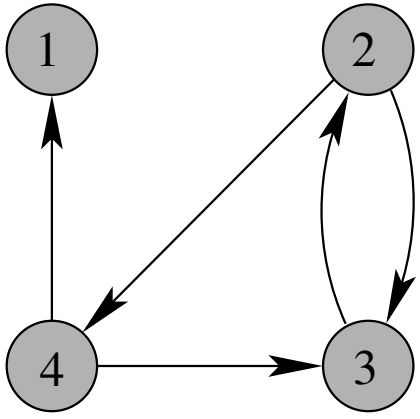
G



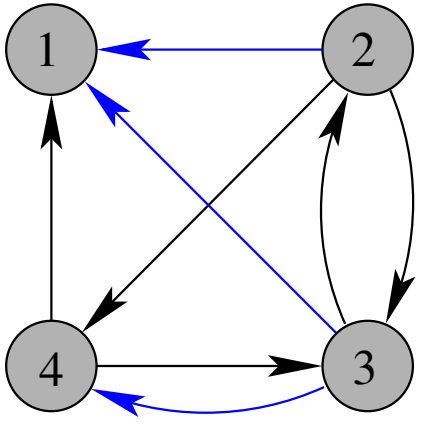
G^*

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

Exemplo



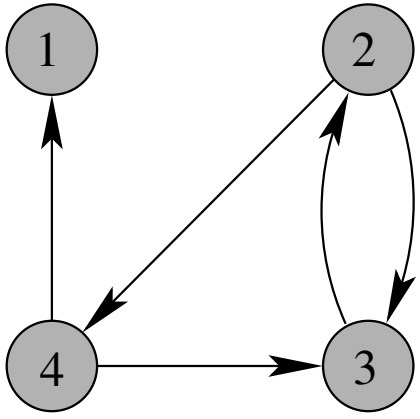
G



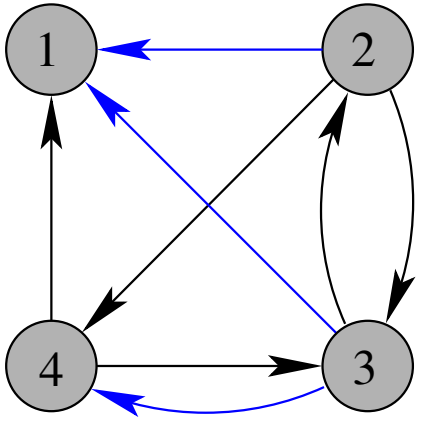
G^*

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

Exemplo



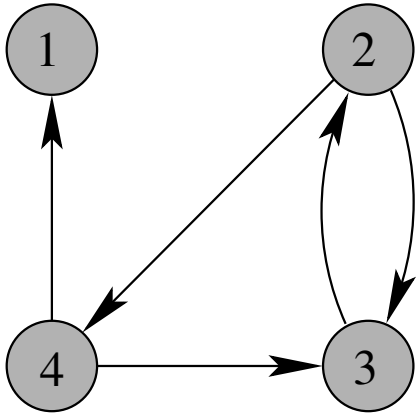
G



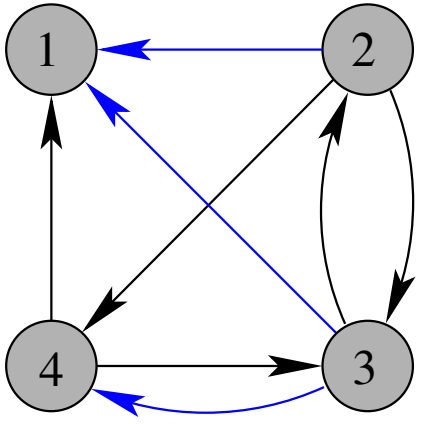
G^*

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

Exemplo



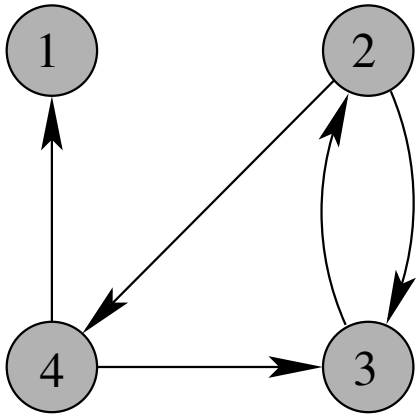
G



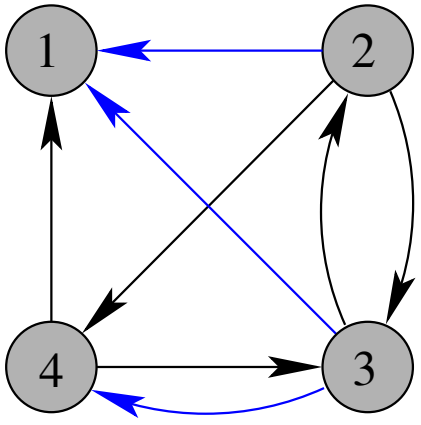
G^*

$$T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \qquad T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Exemplo



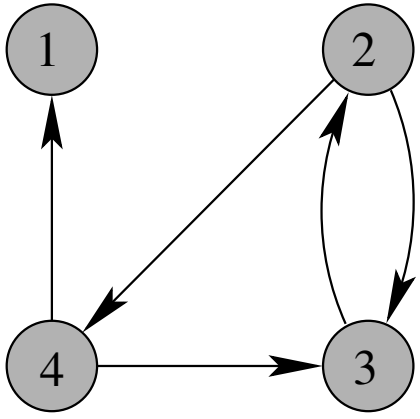
G



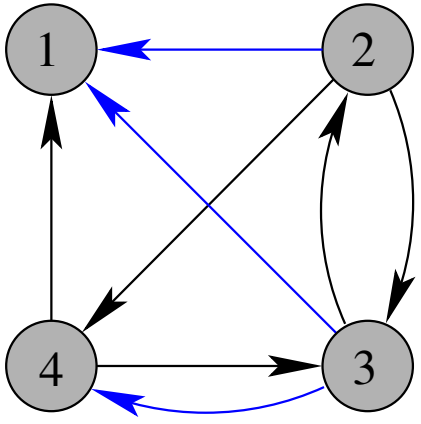
G^*

$$T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \qquad T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Exemplo



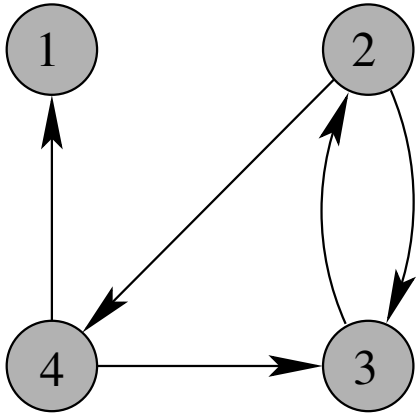
G



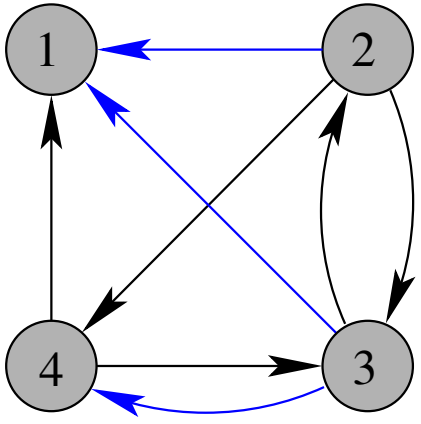
G^*

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \qquad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Exemplo



G



G^*

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \qquad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$