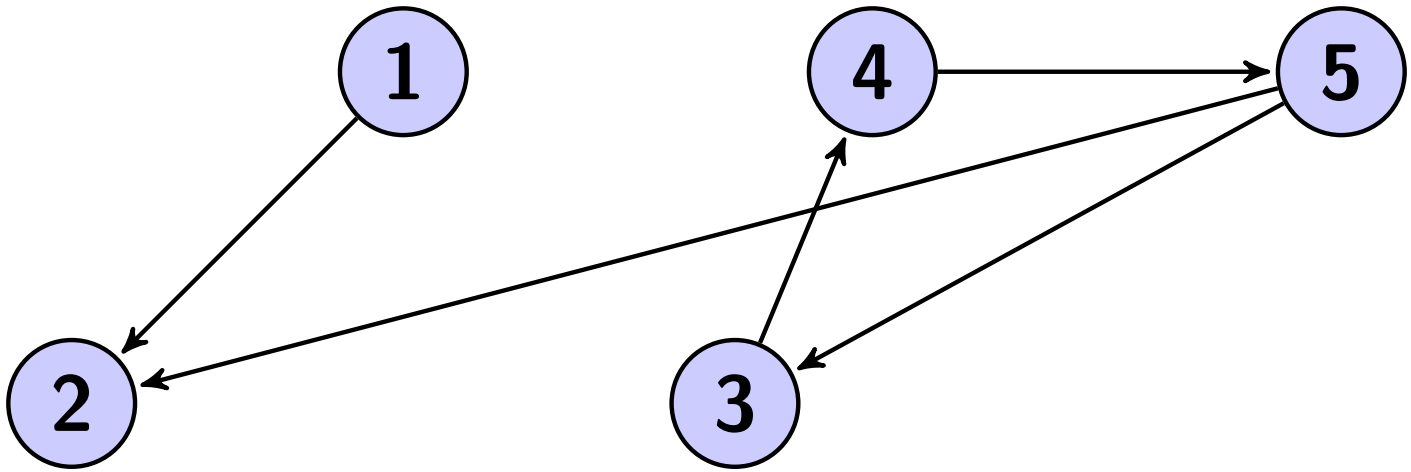
Dígrafos, arestas com pesos e menor caminho

Marcelo K. Albertini

16 de agosto de 2023

Grafo direcionado ou **Dígrafo**

Conjunto de vértices conectados por arestas direcionadas.



Em dígrafo existe **grau de saída** e de **grau de entrada**: grau de saída de 5 é 2 e grau de entrada de 5 é 1.

Aplicações de dígrafos

dígrafo	**vértice**	**aresta direcionada**
transportes	cruzamentos	ruas de mão-única
web	páginas web	links
cadeia alimentar	espécies	predador-presa
escalonamento	tarefa	restrição de precedência
finanças	banco	transação
celulares	pessoa	ligação
DST	pessoa	infecção
jogos	posição tabuleiro	movimento
literatura científica	artigo científico	citação

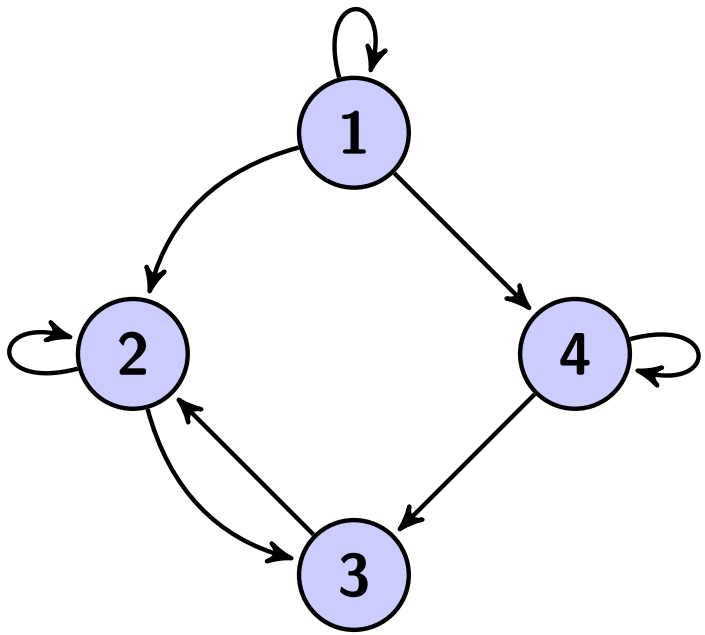
API de Dígrafos

API similar à de Grafos

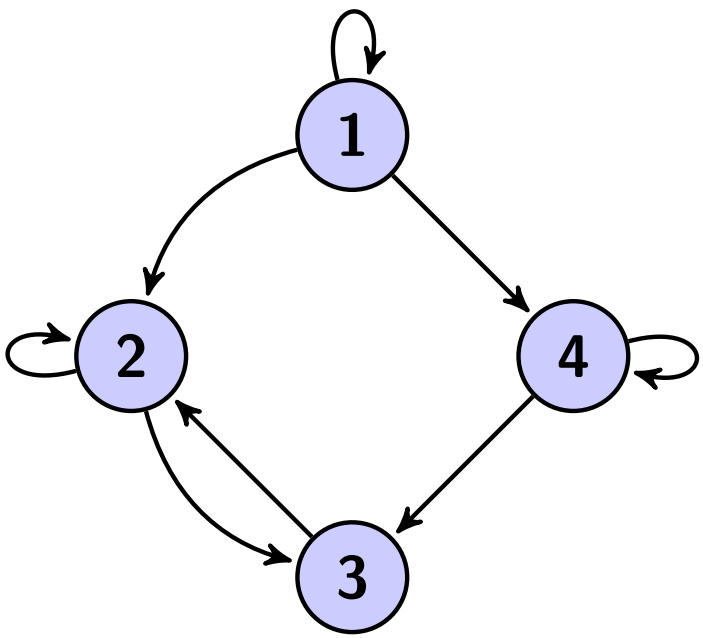
public class	**Digrafo**	
	Digrafo(int V)	criar dígrafo vazio com V vértices
	Digrafo(InputStream in)	criar dígrafo a partir de entradas
void	novaAresta(int v, int w)	criar aresta
Iterable<Integer>	adj(int v)	vértices adjacentes a v
int	V()	número de vértices
int	E()	número de arestas
Digrafo	reverter()	digrafo com arestas reversas

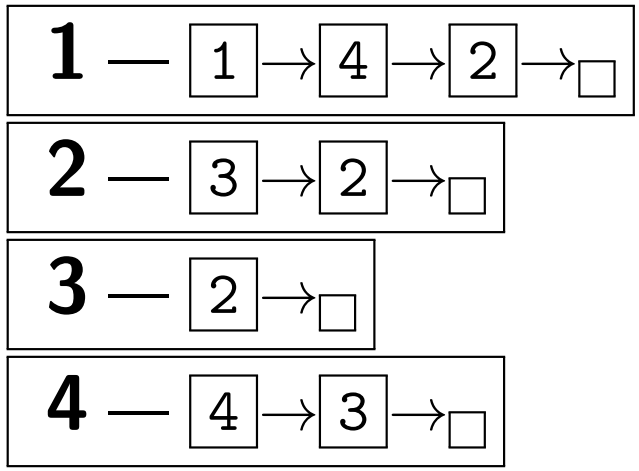
Problemas de dígrafos

- Caminho (alcançabilidade). Existe um caminho direcionado de `s` para `t`?
- Ordenação topológica. É possível organizar o dígrafo de tal forma que nenhuma aresta aponta para baixo?
- Menor caminho. Qual é o menor caminho direcionado de `s` a `t`?



Representação de um dígrafo com listas de adjacências





Implementação muito similar ao `Grafo`, porém ao inserir aresta, só define o sentido pedido e não os dois.

Problema: alcançabilidade

Problema

Encontrar todos os vértices **alcançáveis** a partir de s.

Problema: alcançabilidade

Problema

Encontrar todos os vértices **alcançáveis** a partir de s.

Solução

Busca em profundidade em dígrafos. O código é igual ao utilizado em grafos.

- **DFS**(para visitar um vértice v)
  - Marcar v como visitado
  - Recursivamente visitar vértices adjacentes a v que não foram marcados
- Vértices marcados são os vértices **alcançáveis**

Aplicação de alcançabilidade

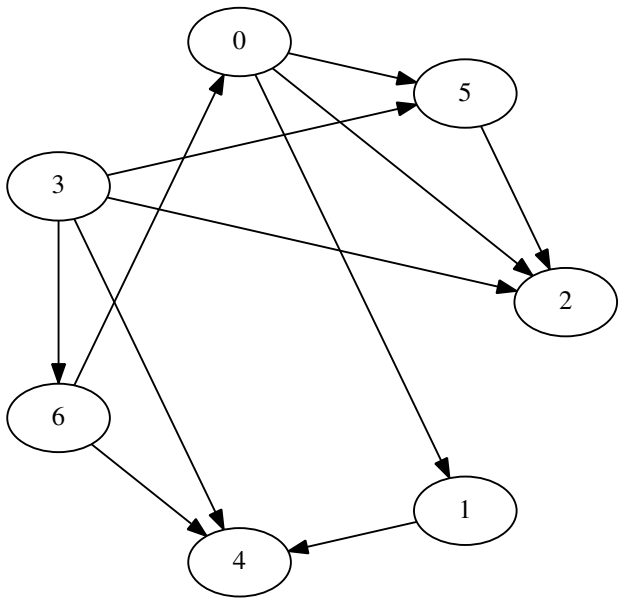
Aplicação

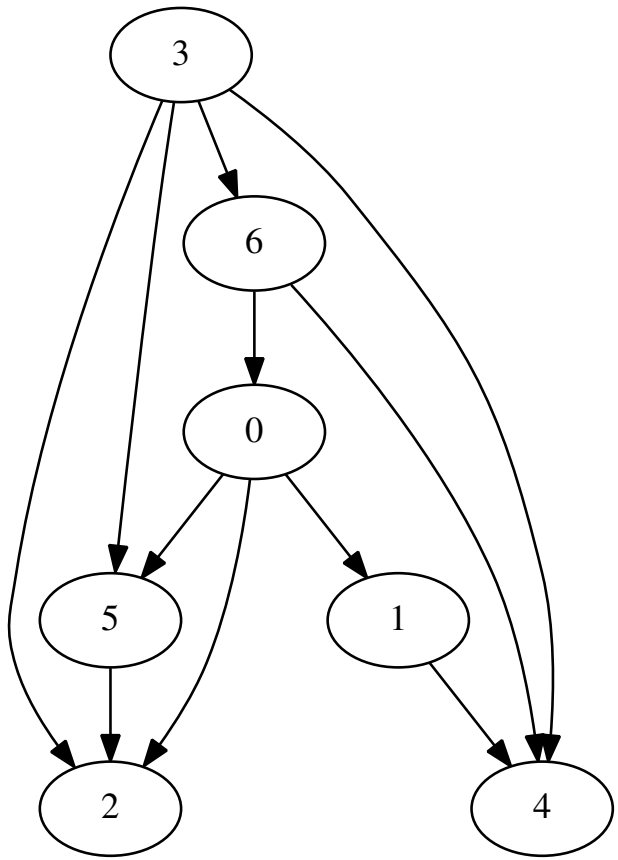
Análise do fluxo de controle de programas para verificação de `bugs`.

- Código é representado por dígrafo
  - Vértice: bloco de instruções sem desvios
  - Aresta: desvio do fluxo de operações (`if`, `for`, `while`, `função`, `...`)
- É possível detectar
  - Regiões mortas - nunca alcançáveis
  - Loops infinitos - nunca é possível ao vértice de saída (return)
- Coletor de Lixo Marcar-Limpar (*mark-sweep*)
  - Vértices: objetos. Arestas: Referências.
  - **Raízes**: objetos acessíveis a partir da stack
  - Objetos alcançáveis a partir das **raízes** são marcados
  - Objetos não alcançáveis são **lixo** e são liberados

Ordenação topológica

- Problema da precedência
  - Dado um cronograma, quais tarefas posso fazer antes?
  - Dados disciplinas e pré-requisitos, qual ordem de disciplinas posso fazer?





Ordenação topológica

Problema

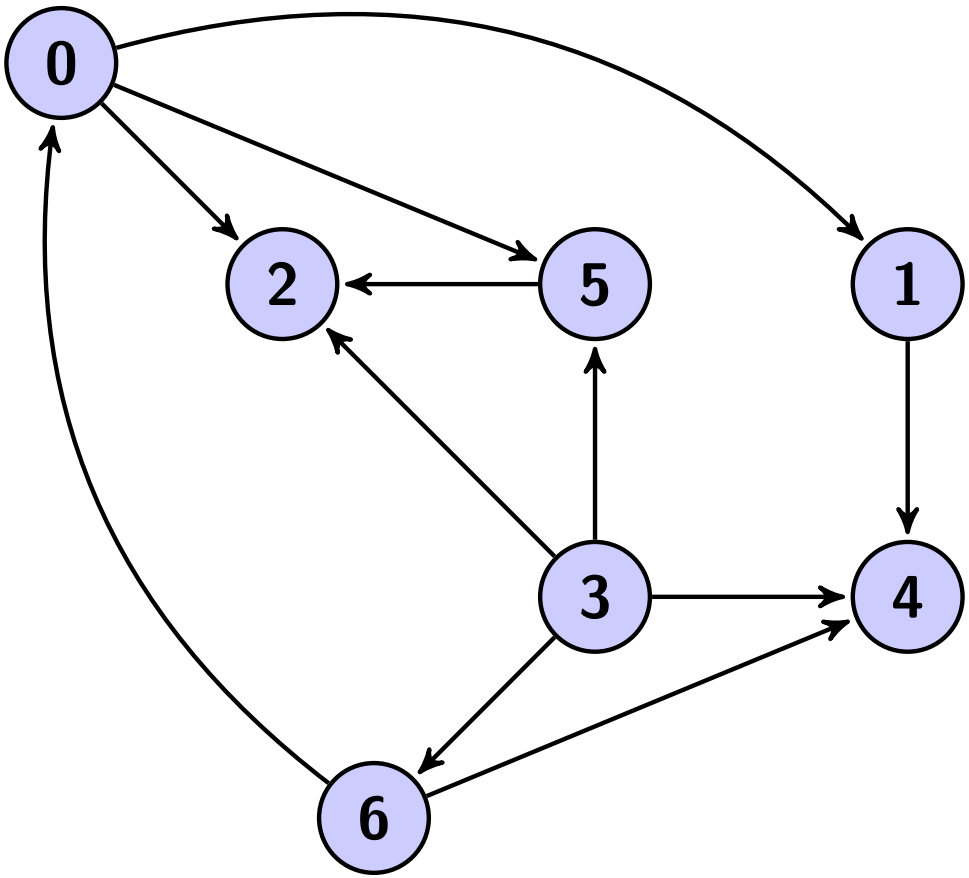
- Temos um grafo direcionado sem ciclos
- Objetivo: Redesenhar grafo, tal que todas as arestas apontam para o mesmo sentido.

Solução

- Usar busca em profundidade.
- Retornar vértices em pós-ordem reversa
  - Pós-ordem ocorre após usar cada vértice

```
1  class OrdemTopologica {
2    boolean[] marcado;
3    Stack<Integer> posOrdemRev;// ordem topologica
4
5    public OrdemTopologica(Digrafo G) {
6     posOrdemRev = new Stack<Integer>();
7     marcado = new boolean[G.V()];
8     for (int v = 0; v < G.V(); v++)
9       if (!marcado[v]) dfs(G, V);
10   }
11
12   void dfs(Digrafo G, nt v) { // busca em profundidade
13     marcado[v] = true;
14     for (int w : G.adj(v))
15       if (!marcado[w]) dfs(G, w);
16     posOrdemRev.push(v); // poe no topo da pilha
17   }
18
19   public Iterable<Integer> ordem() {
20     return posOrdemRev;
21   }
22 }
```

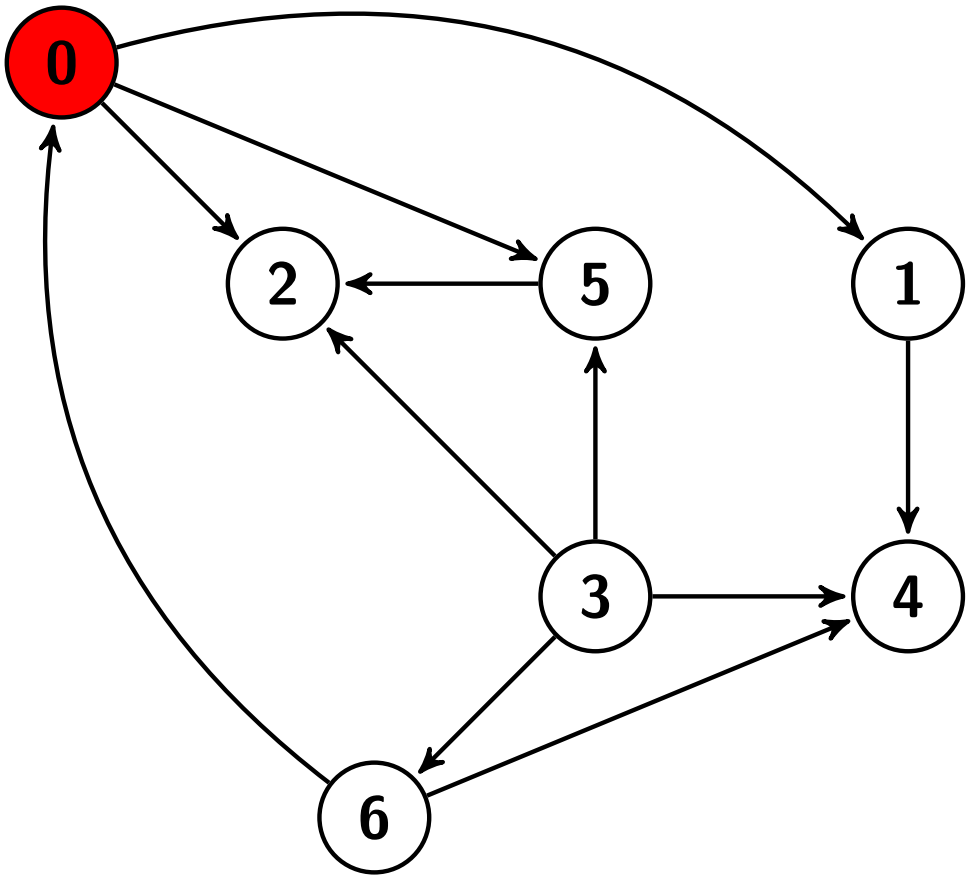
Ordenação topológica



$0 \to 5$
$0 \to 2$
$0 \to 1$
$3 \to 6$
$3 \to 5$
$3 \to 4$
$5 \to 4$
$6 \to 4$
$6 \to 0$
$3 \to 2$
$1 \to 4$

Um grafo direcionado sem ciclos.

Ordenação topológica



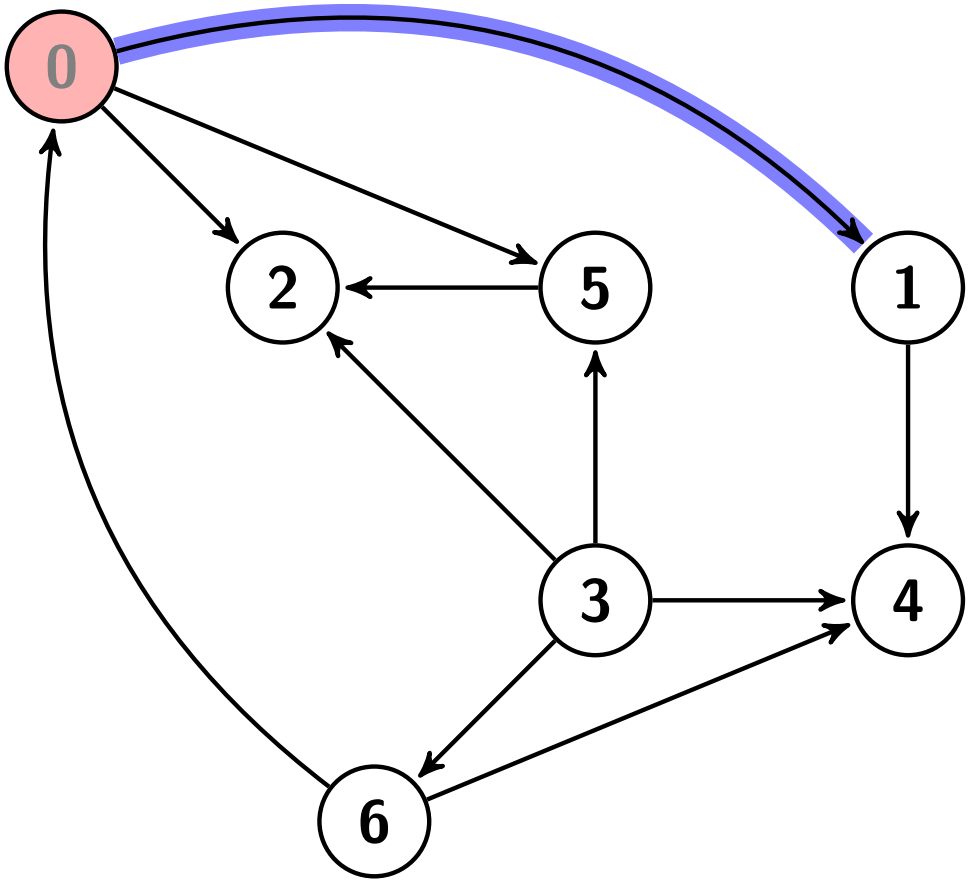
Pós-ordem - pilha

- {}

Ação

visita 0; avaliar 1; avaliar 2; e avaliar 5

Ordenação topológica



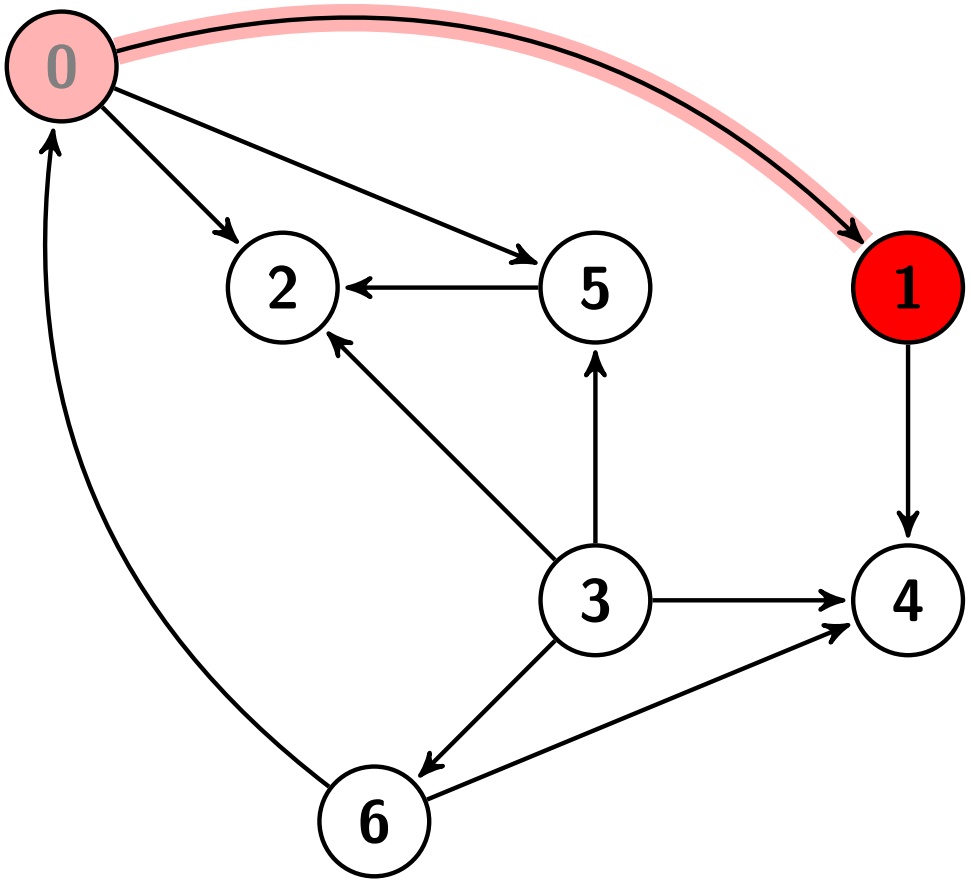
Pós-ordem - pilha

- {}

Ação

visita 0; avaliar 1; avaliar 2; e avaliar 5

Ordenação topológica



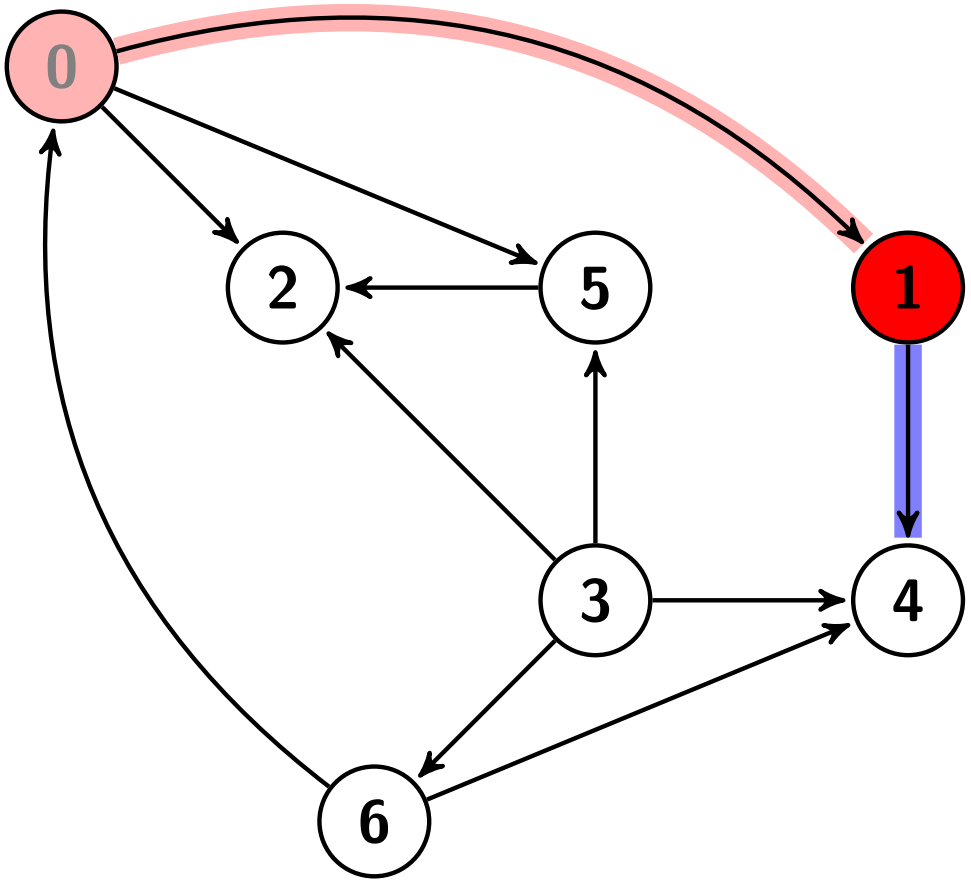
Pós-ordem - pilha

- {}

Ação

visita 1; avaliar 4;

Ordenação topológica



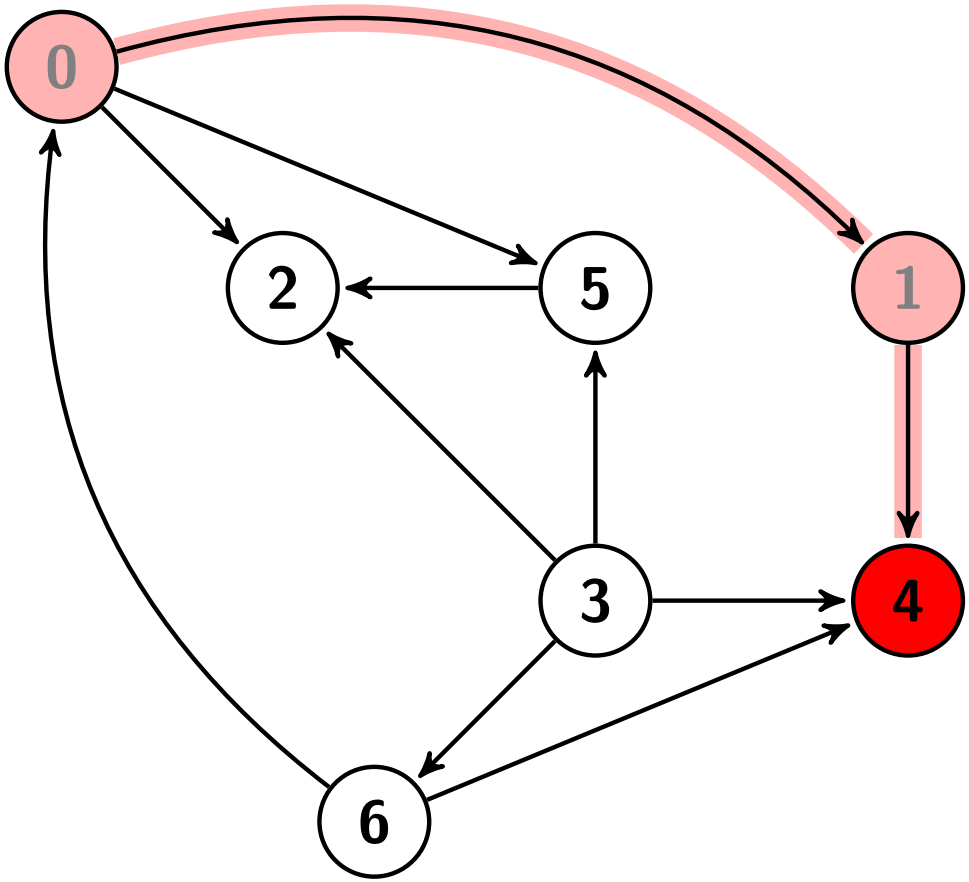
Pós-ordem - pilha

- {}

Ação

visita 1; avaliar 4;

Ordenação topológica



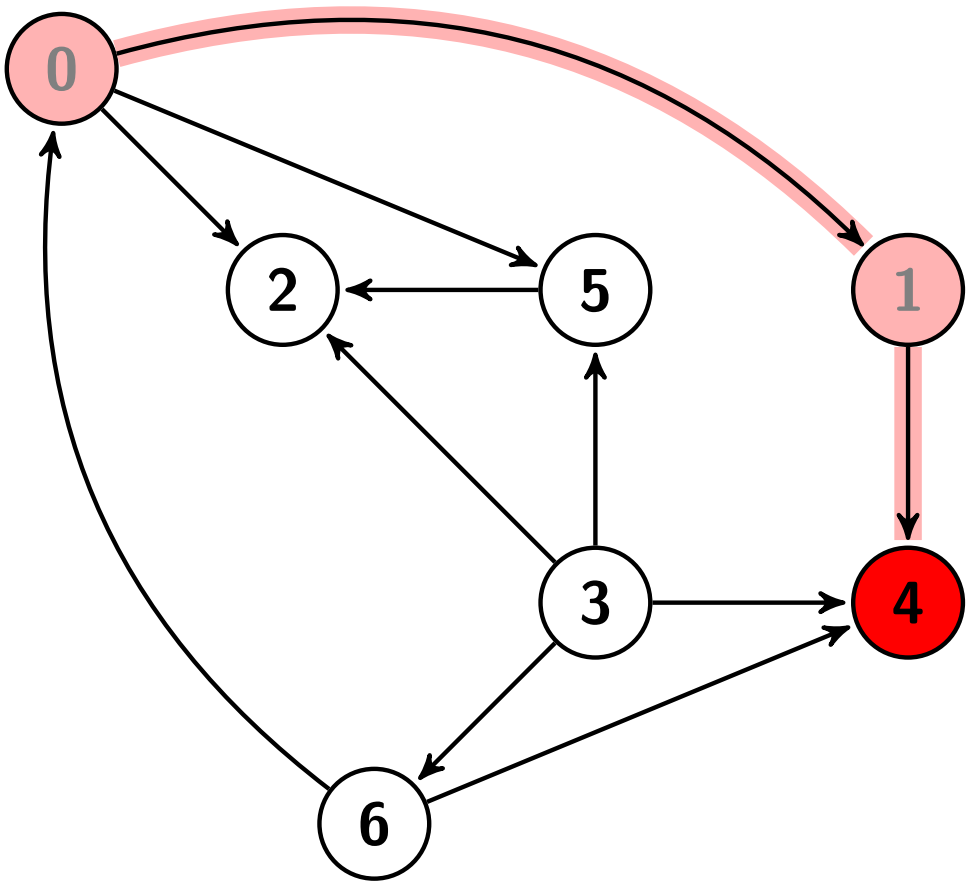
Pós-ordem - pilha

- {}

Ação

- visita 4;

Ordenação topológica



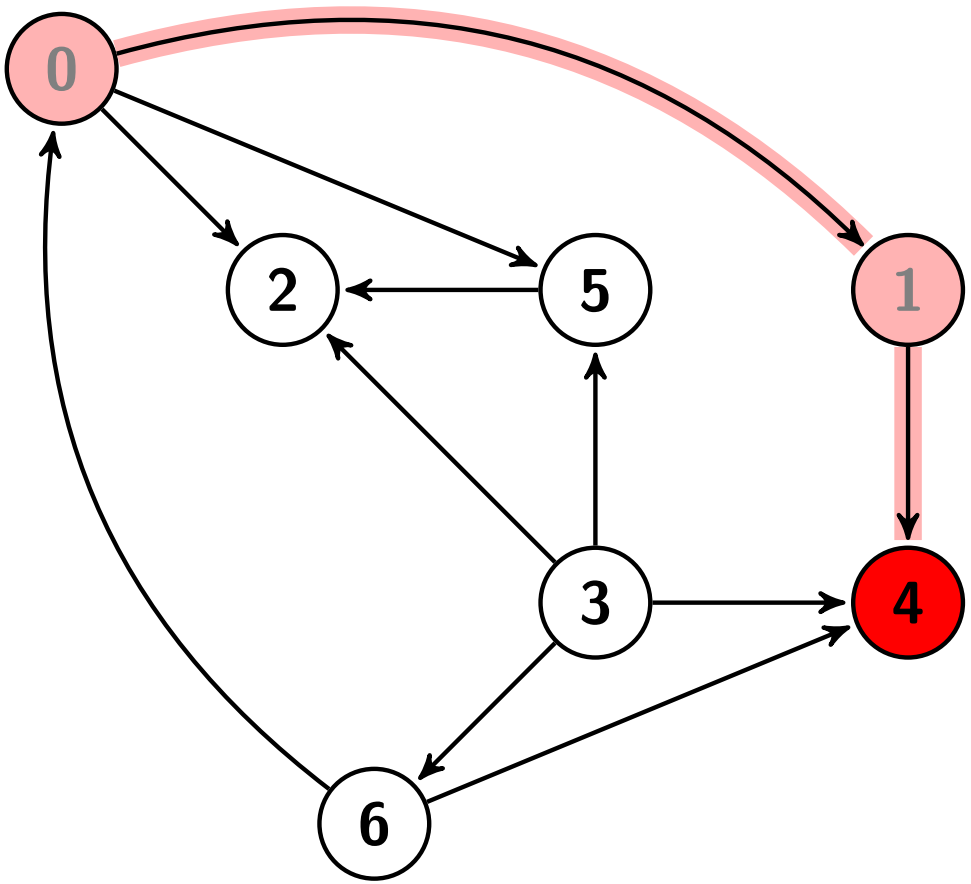
Pós-ordem - pilha

- {}

Ação

- visita 4;
- Não há adjacentes para avaliar

Ordenação topológica



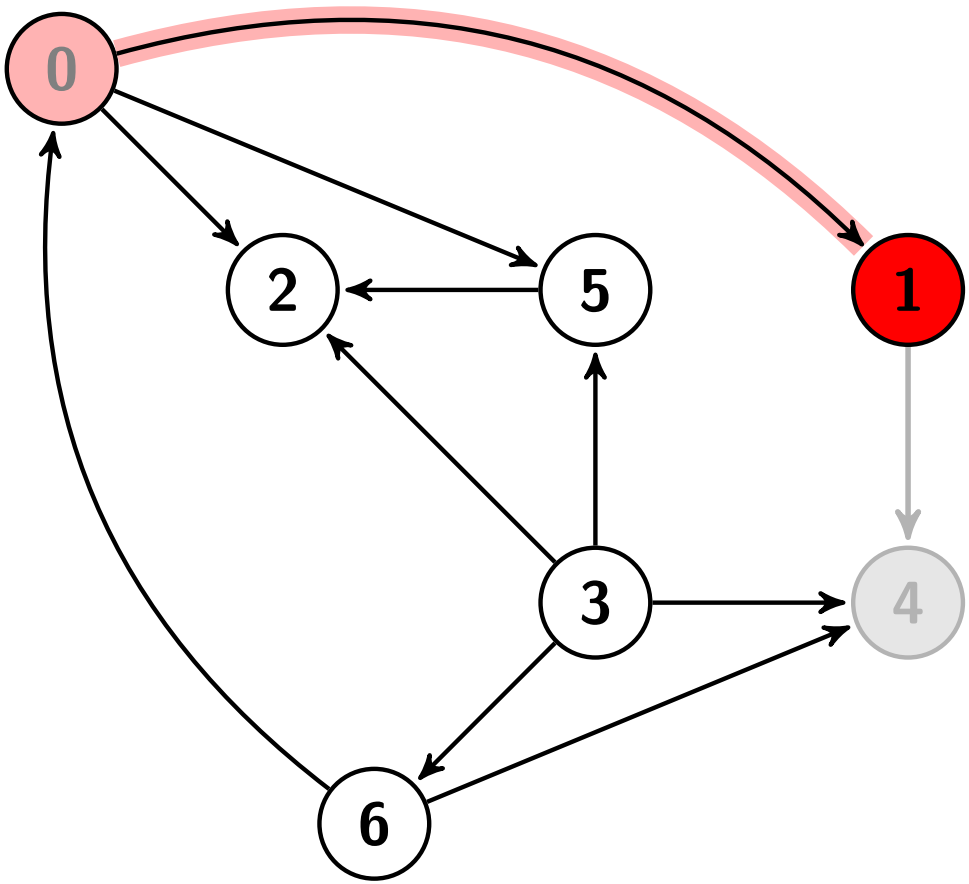
Pós-ordem - pilha

- {4}

Ação

- Não há adjacentes para avaliar
- Adicionar no caminho em pós-ordem

Ordenação topológica



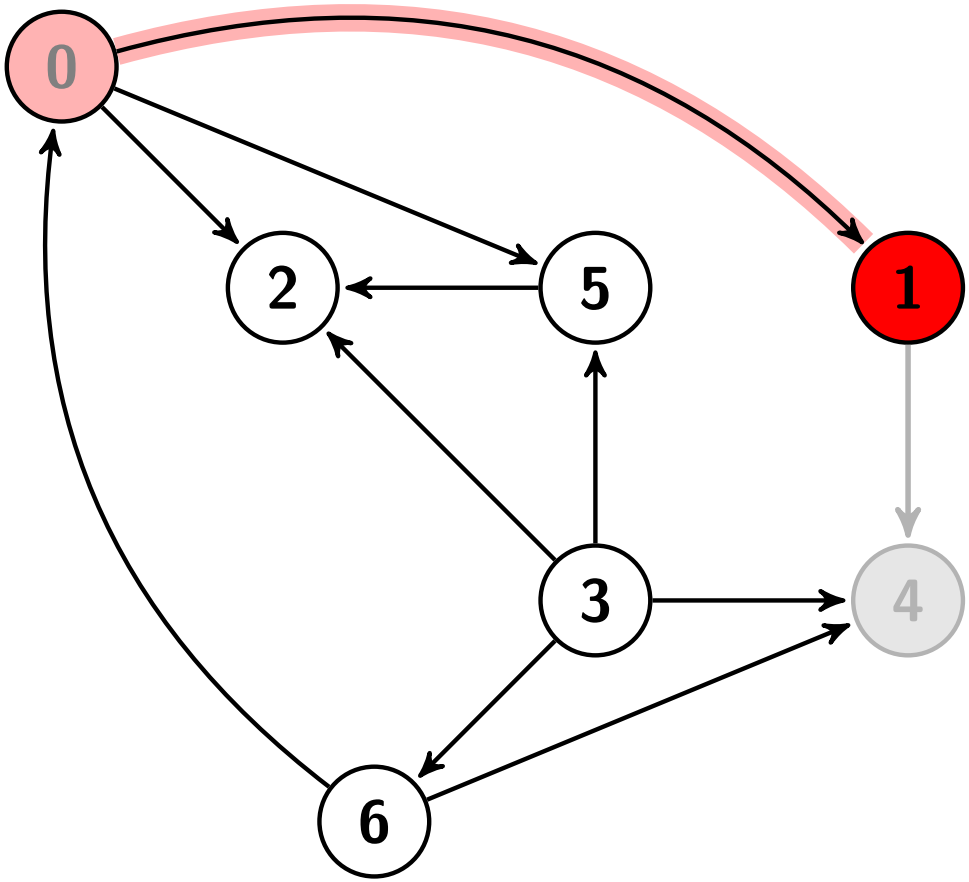
Pós-ordem - pilha

- $\{4\}$

Ação

- vértice 4 terminou
- **volta na recursão**

Ordenação topológica



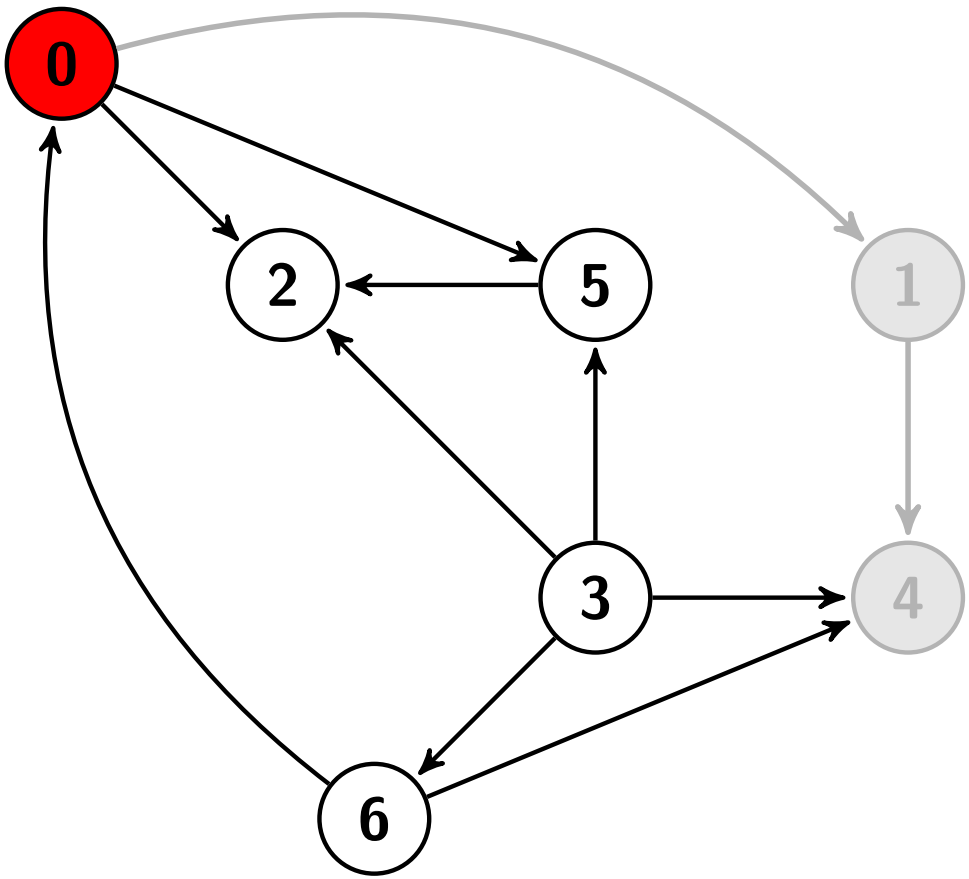
Pós-ordem - pilha

- $\{4, 1\}$

Ação

- vértice 1 terminou, põe na pilha

Ordenação topológica



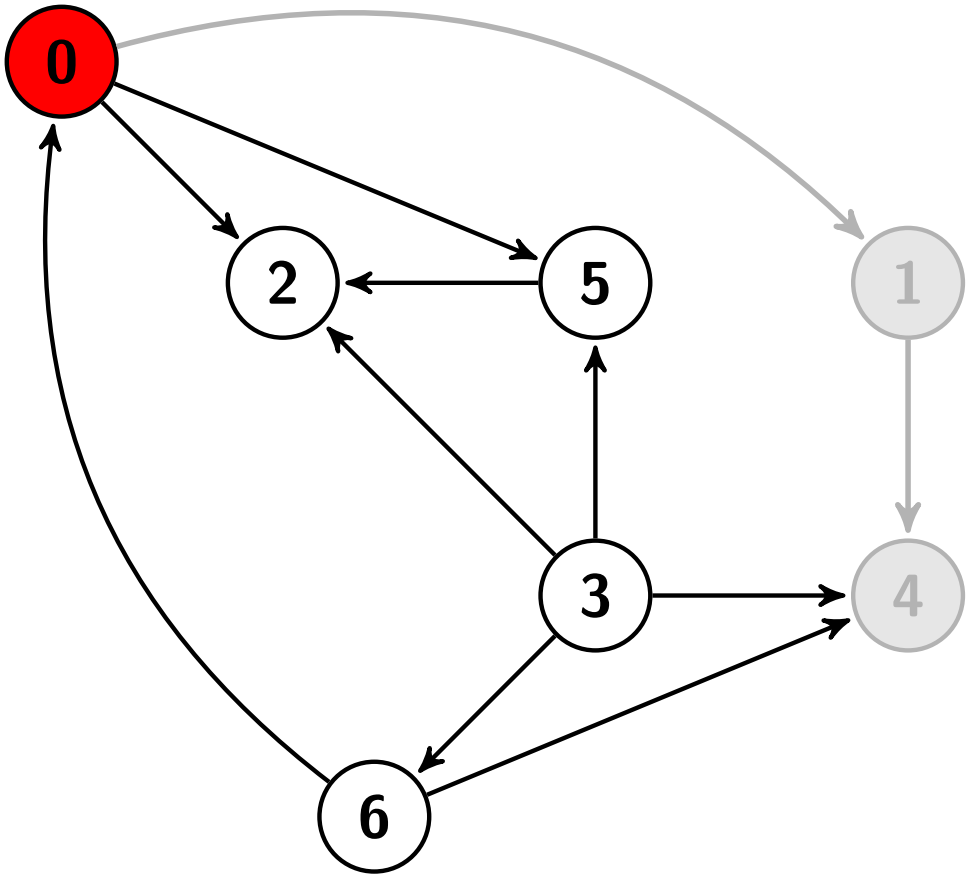
Pós-ordem - pilha

- $\{4, 1\}$

Ação

- vértice 1 terminou, põe na pilha
- **volta na recursão**

Ordenação topológica



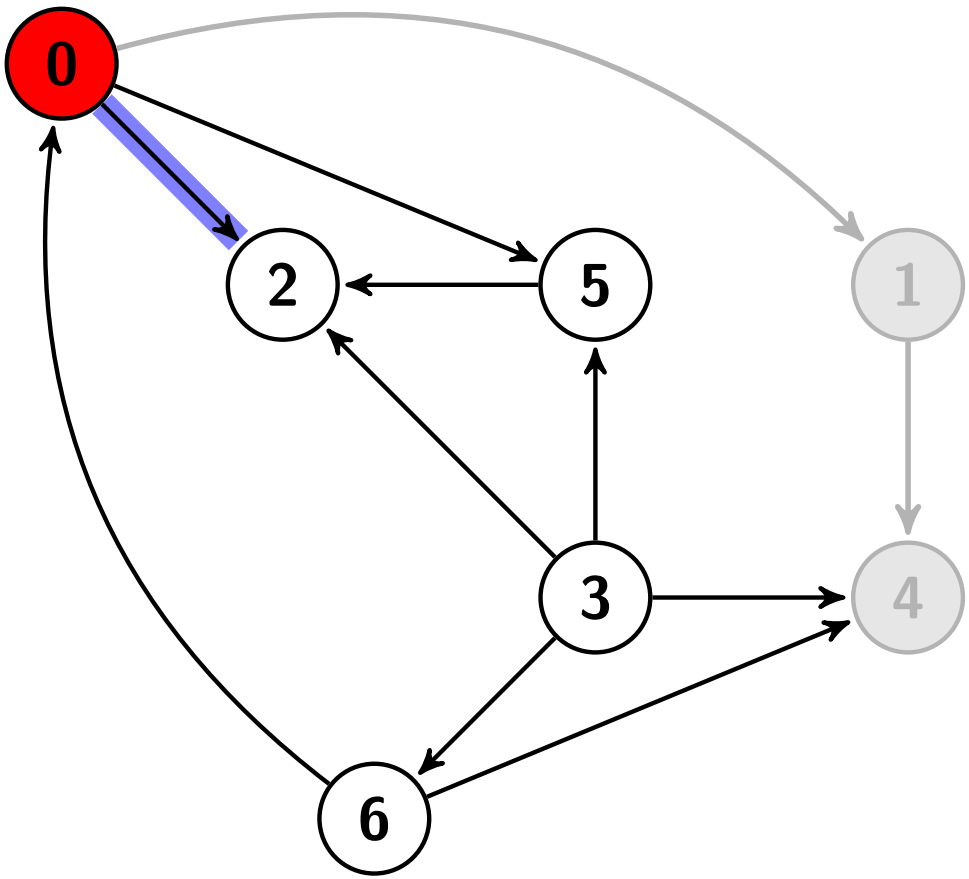
Pós-ordem - pilha

- $\{4, 1\}$

Ação

- visita 0; avaliar 1; avaliar 2; e avaliar 5

Ordenação topológica



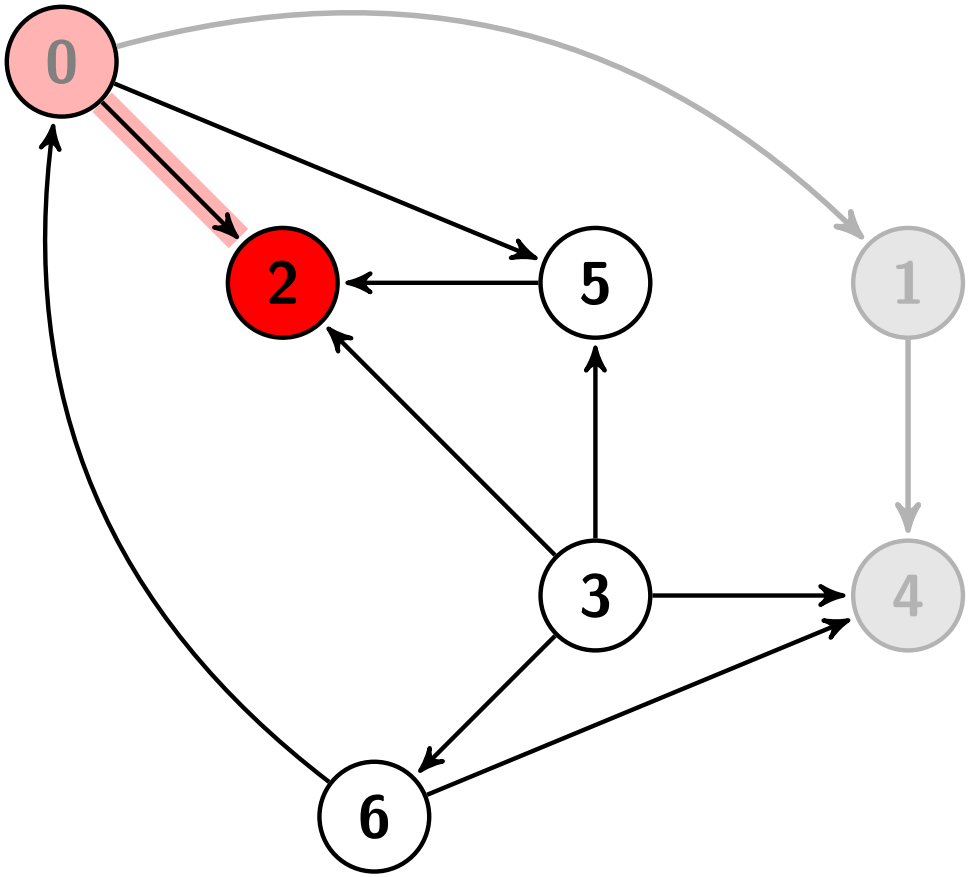
Pós-ordem - pilha

- $\{4, 1\}$

Ação

- visita 0; avaliar 1; avaliar 2; e avaliar 5

Ordenação topológica



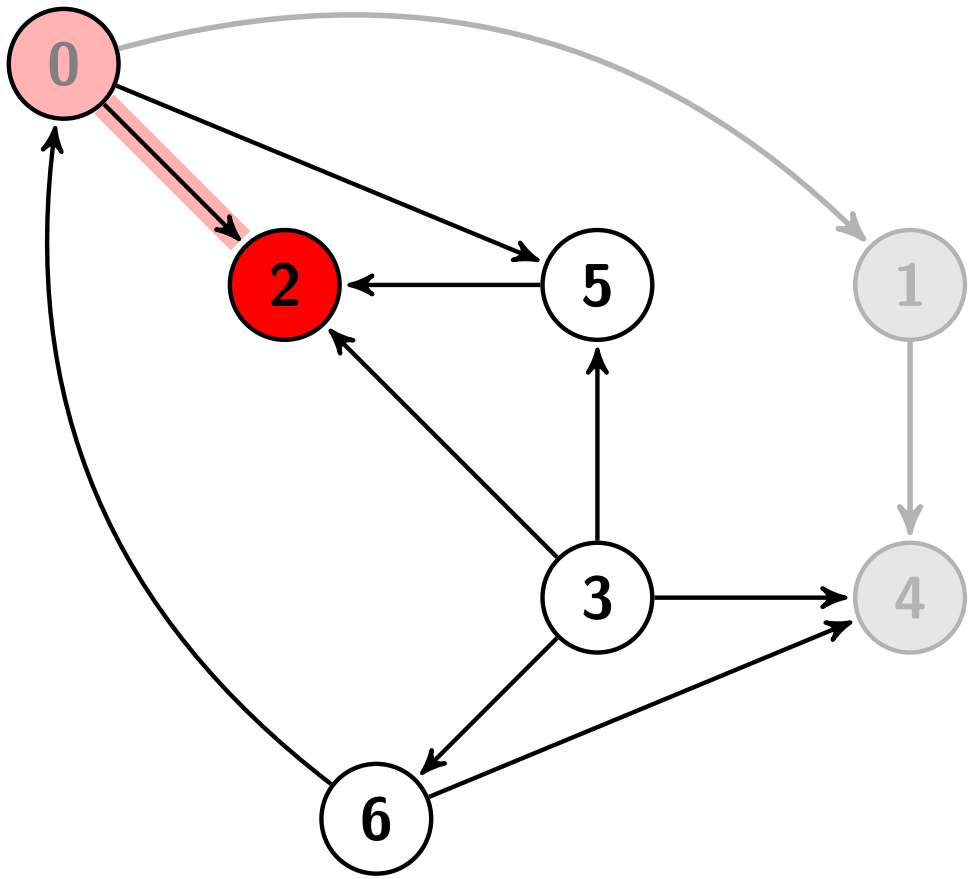
Pós-ordem - pilha

- $\{4, 1\}$

Ação

- visita 2;

Ordenação topológica



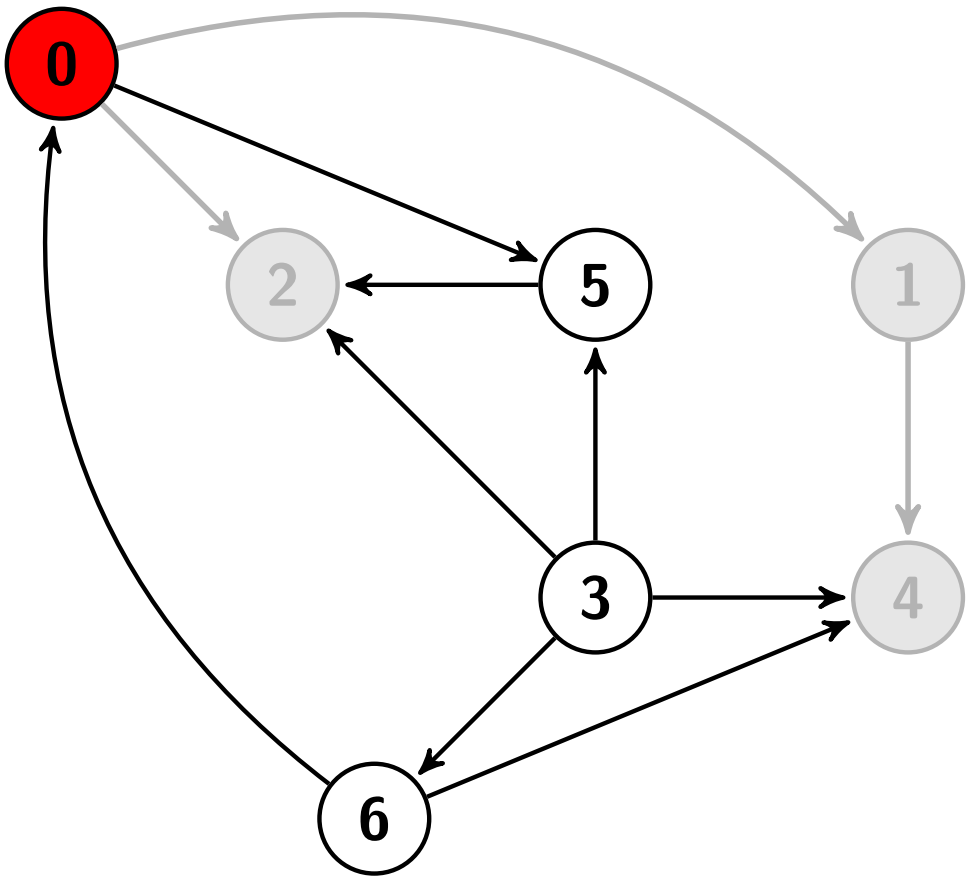
Pós-ordem - pilha

- $\{4, 1, 2\}$

Ação

- visita 2;
- Terminou o **2**. Empilha.

Ordenação topológica



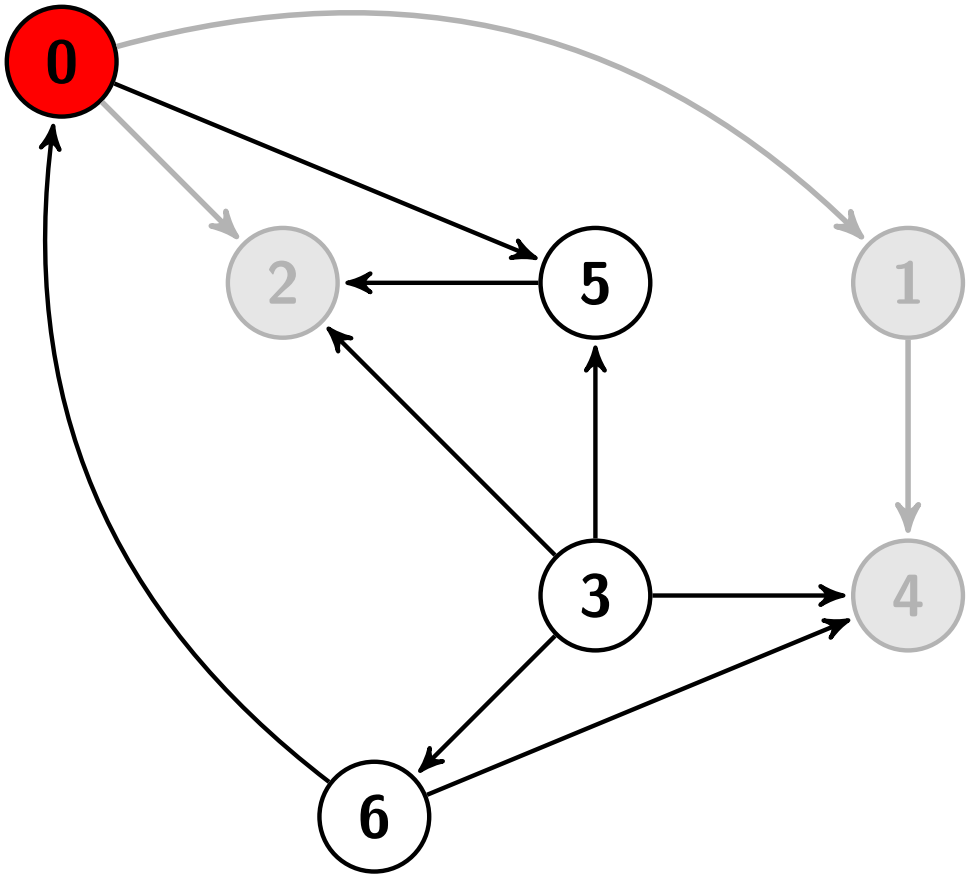
Pós-ordem - pilha

- $\{4, 1, 2\}$

Ação

- Retorna na recursão.

Ordenação topológica



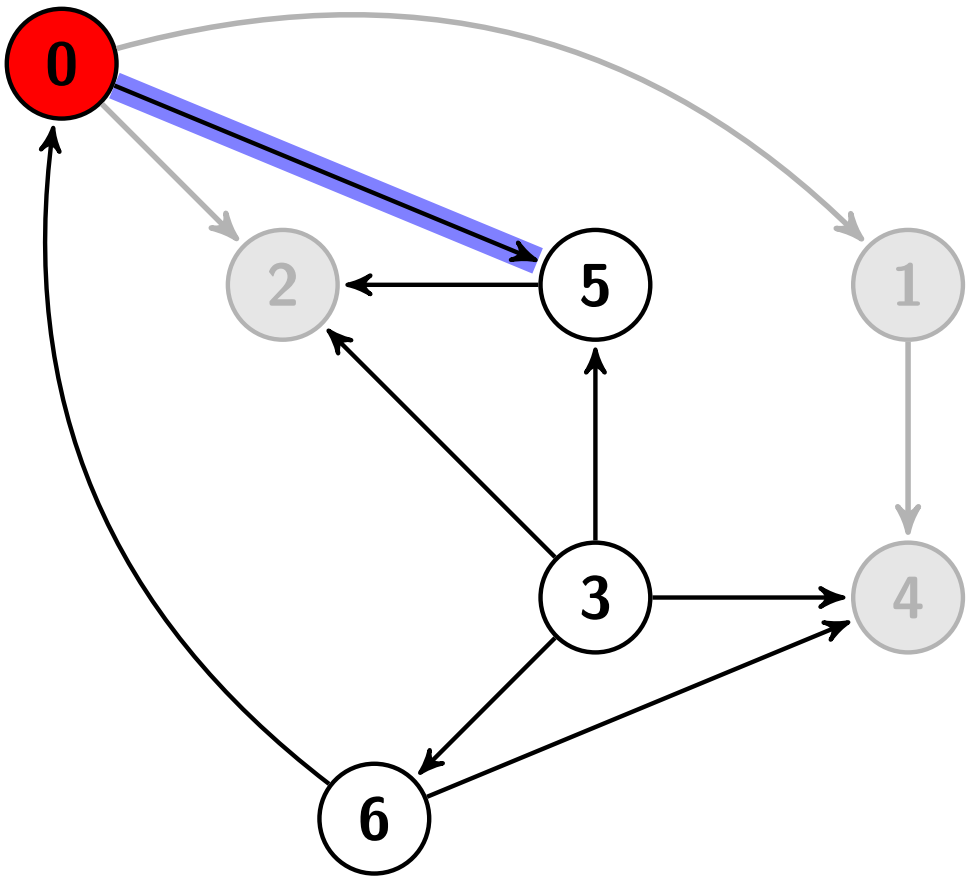
Pós-ordem - pilha

- $\{4, 1, 2\}$

Ação

- visita 0; avaliar 1; avaliar 2; e avaliar 5

Ordenação topológica



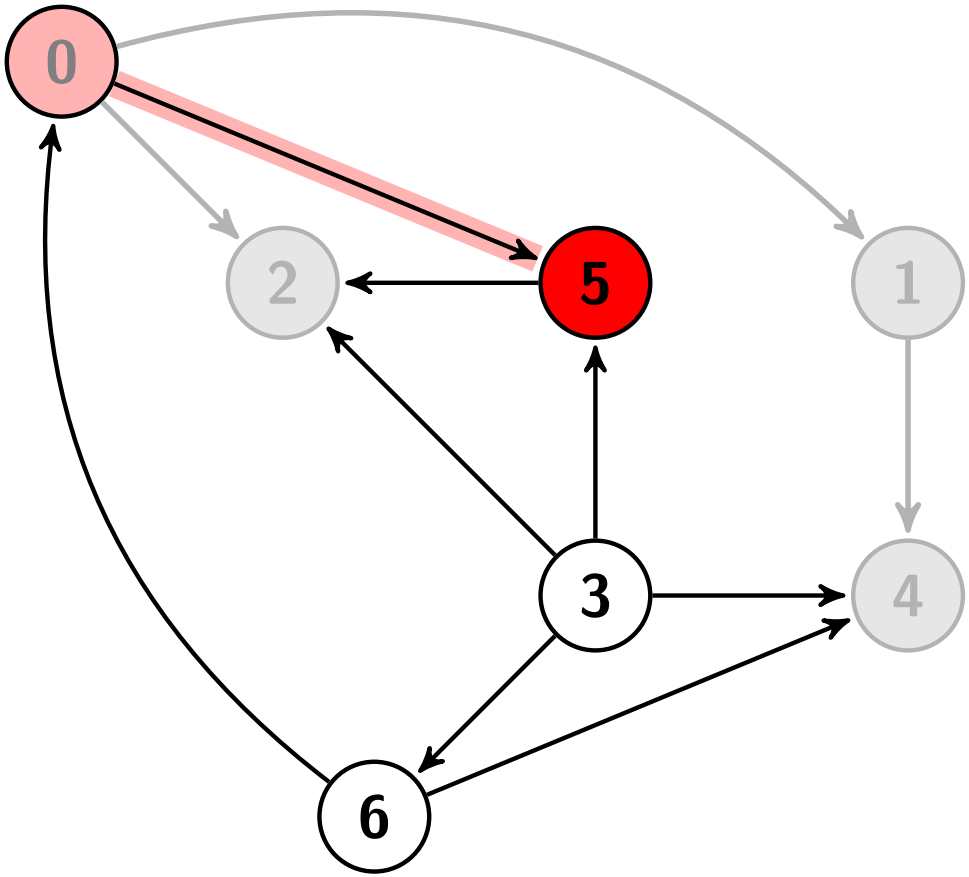
Pós-ordem - pilha

- $\{\mathbf{4, 1, 2}\}$

Ação

- visita 0; avaliar 1; avaliar 2; e avaliar **5**

Ordenação topológica



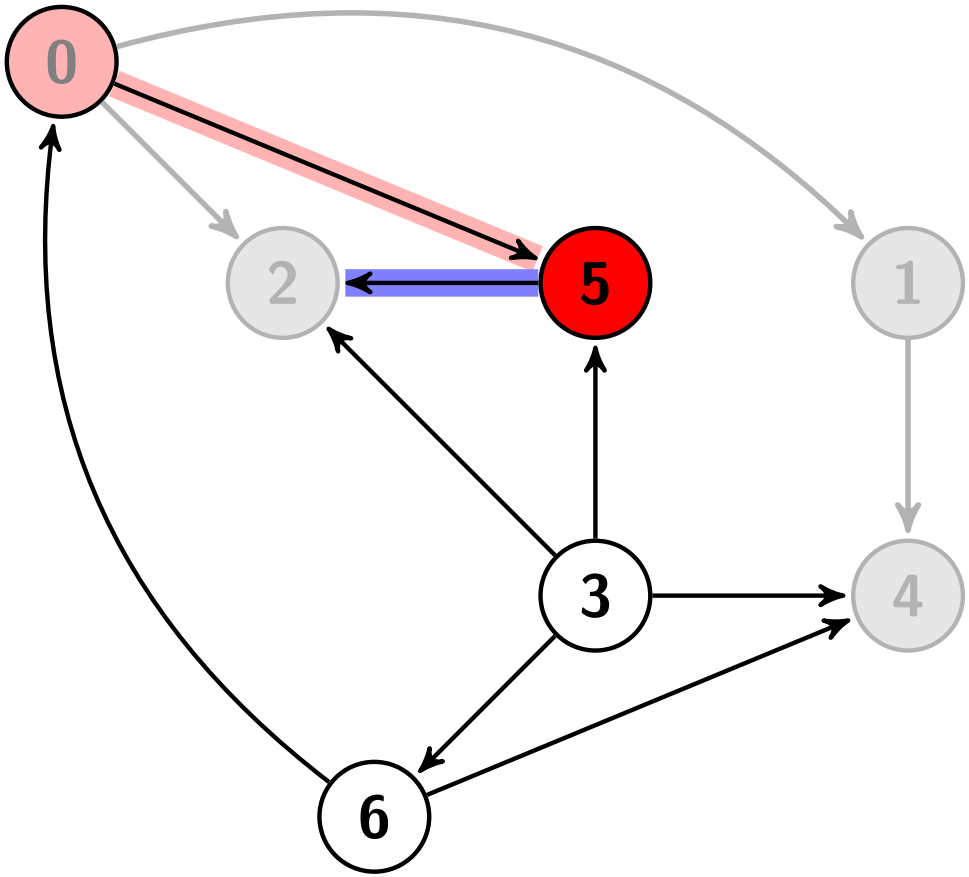
Pós-ordem - pilha

- $\{4, 1, 2\}$

Ação

- visita 5; avaliar 2

Ordenação topológica



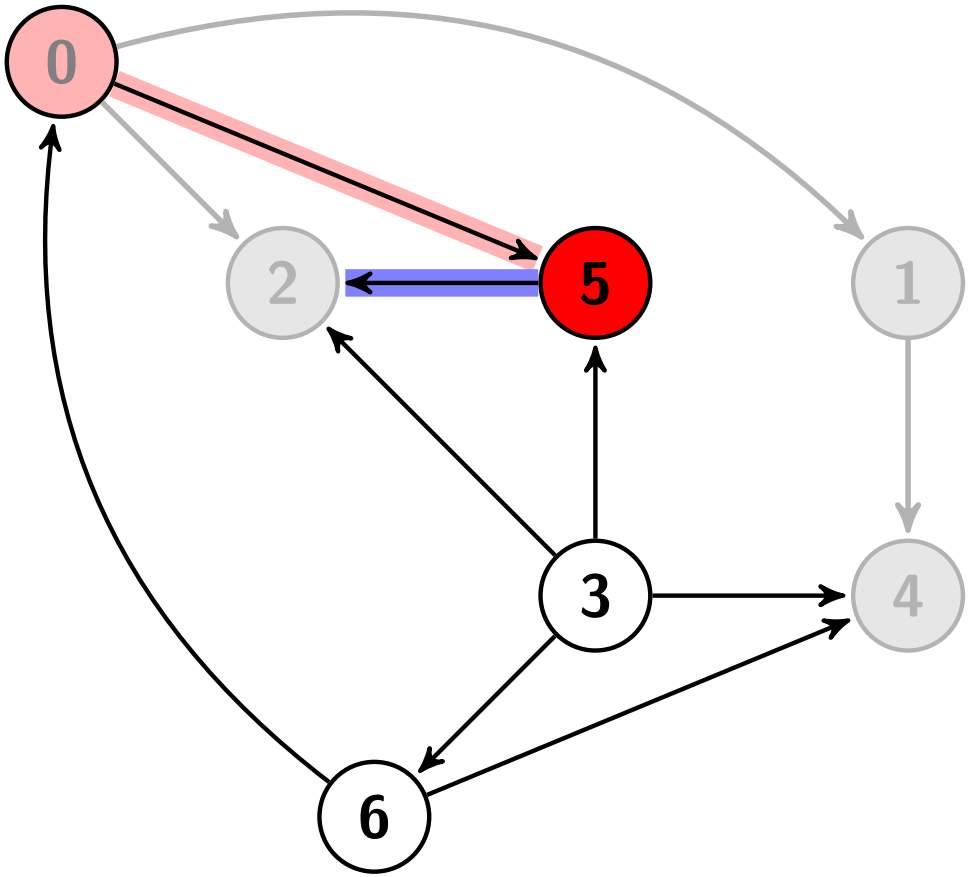
Pós-ordem - pilha

- $\{4, 1, 2\}$

Ação

- visita 5; avaliar **2**

Ordenação topológica



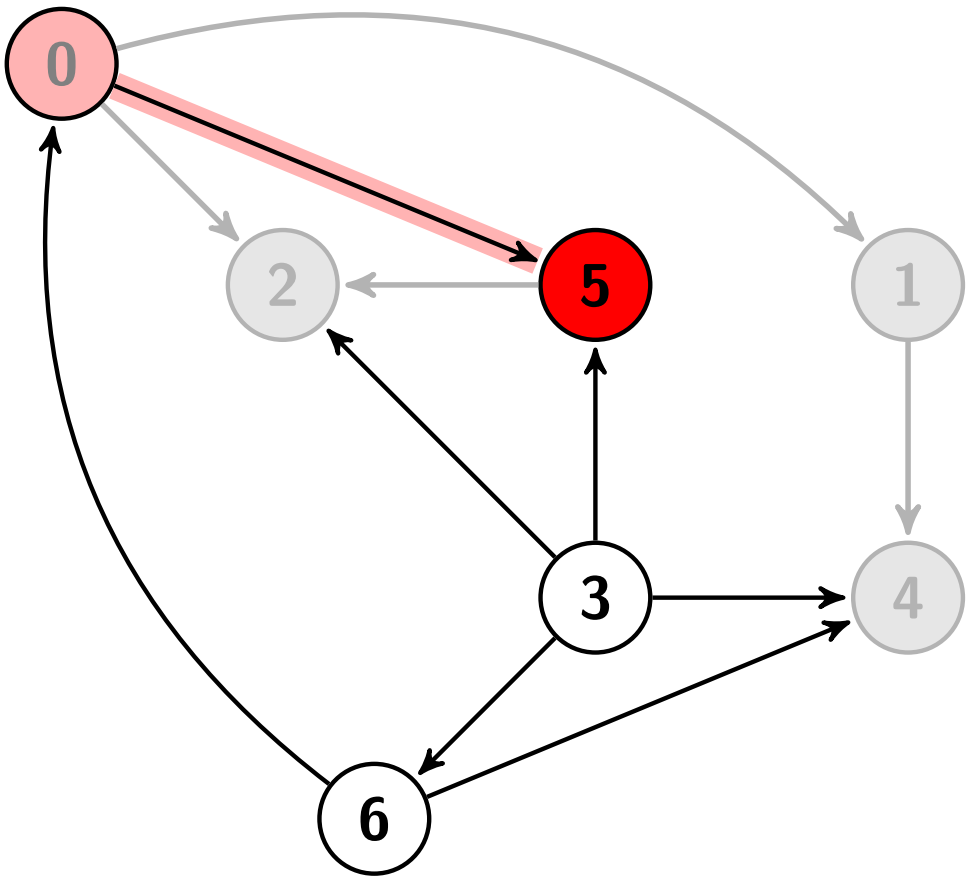
Pós-ordem - pilha

- $\{4, 1, 2\}$

Ação

- visita 5; avaliar **2**
- **2 já está marcado**

Ordenação topológica



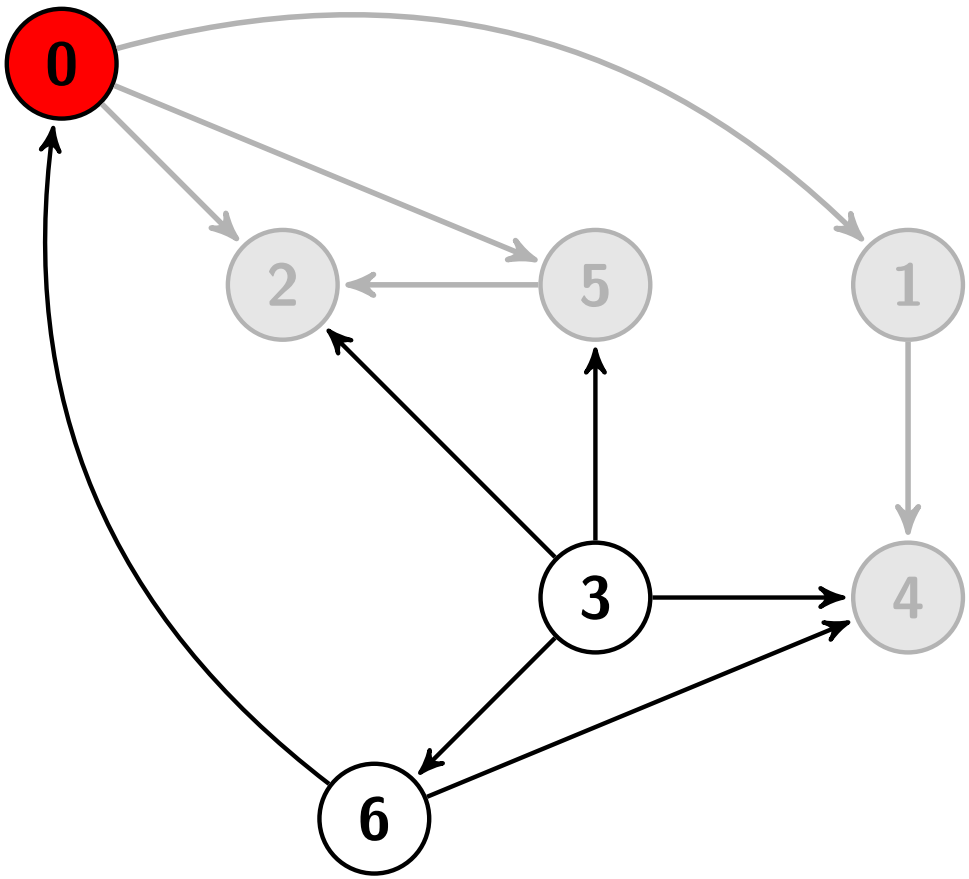
Pós-ordem - pilha

- $\{4, 1, 2, 5\}$

Ação

- 2 já está marcado.
- Retorna recursão. Terminou o 5. Empilha.

Ordenação topológica



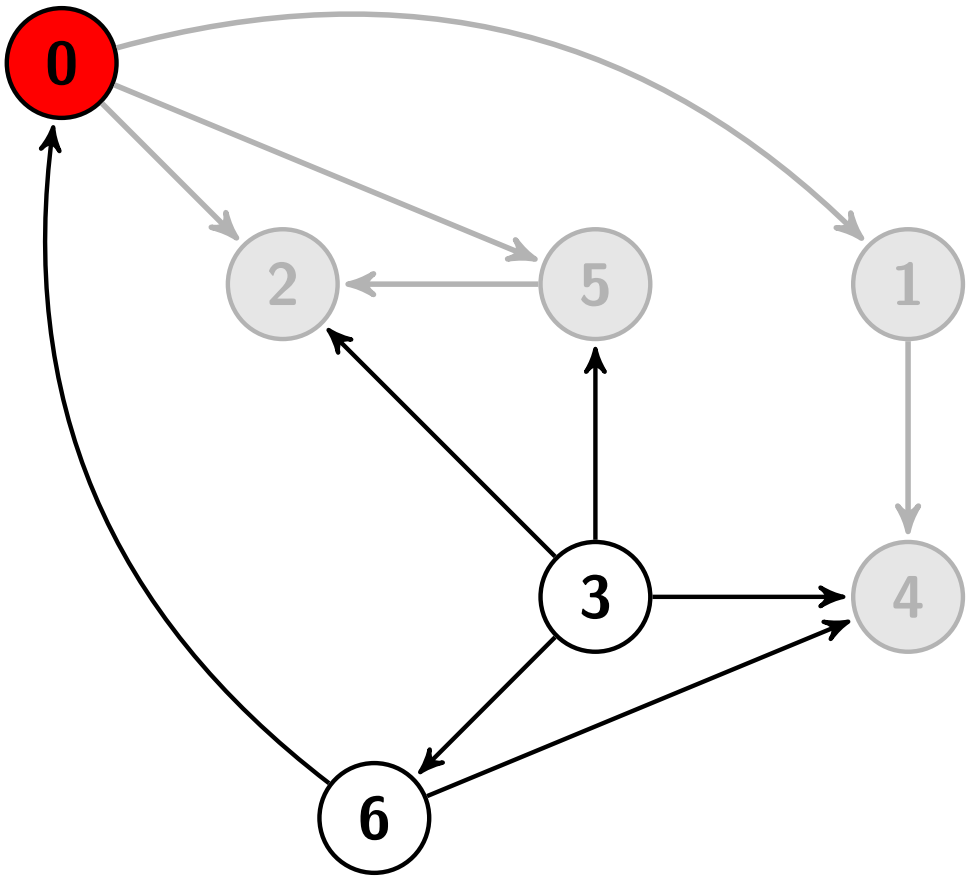
Pós-ordem - pilha

- $\{4, 1, 2, 5\}$

Ação

- Terminou o 5. Retorna recursão.

Ordenação topológica



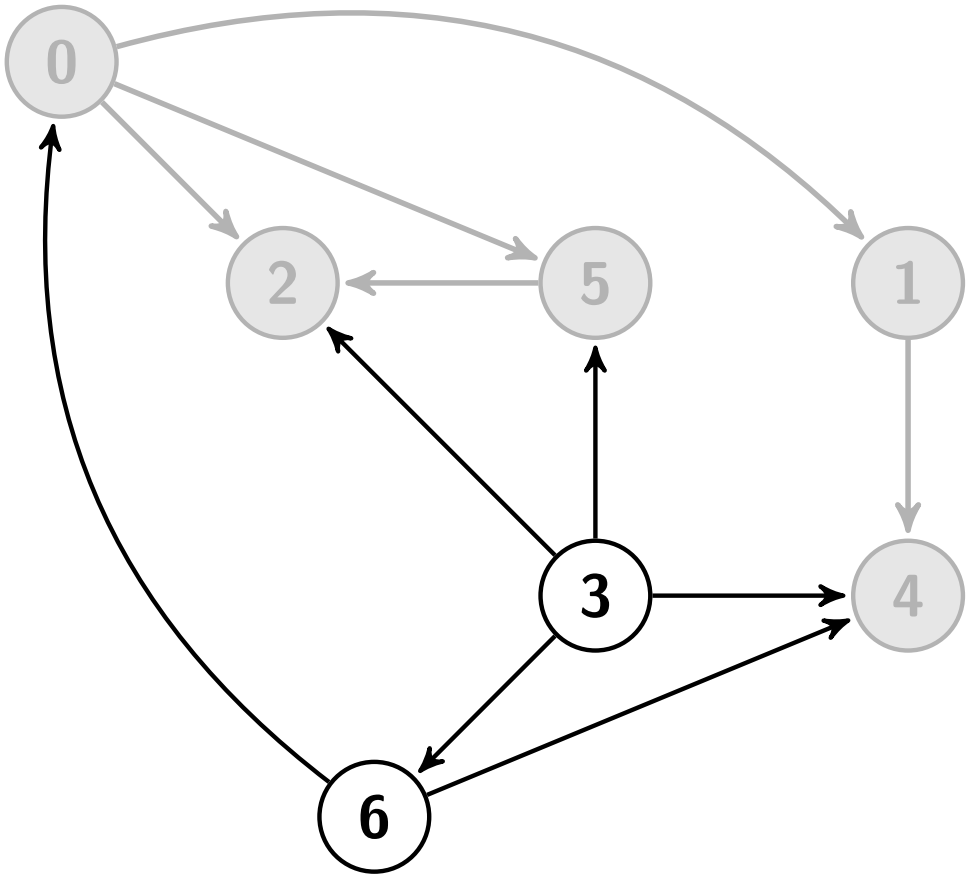
Pós-ordem - pilha

- $\{\mathbf{4, 1, 2, 5, 0}\}$

Ação

- Terminou o **0**. Empilha.

Ordenação topológica



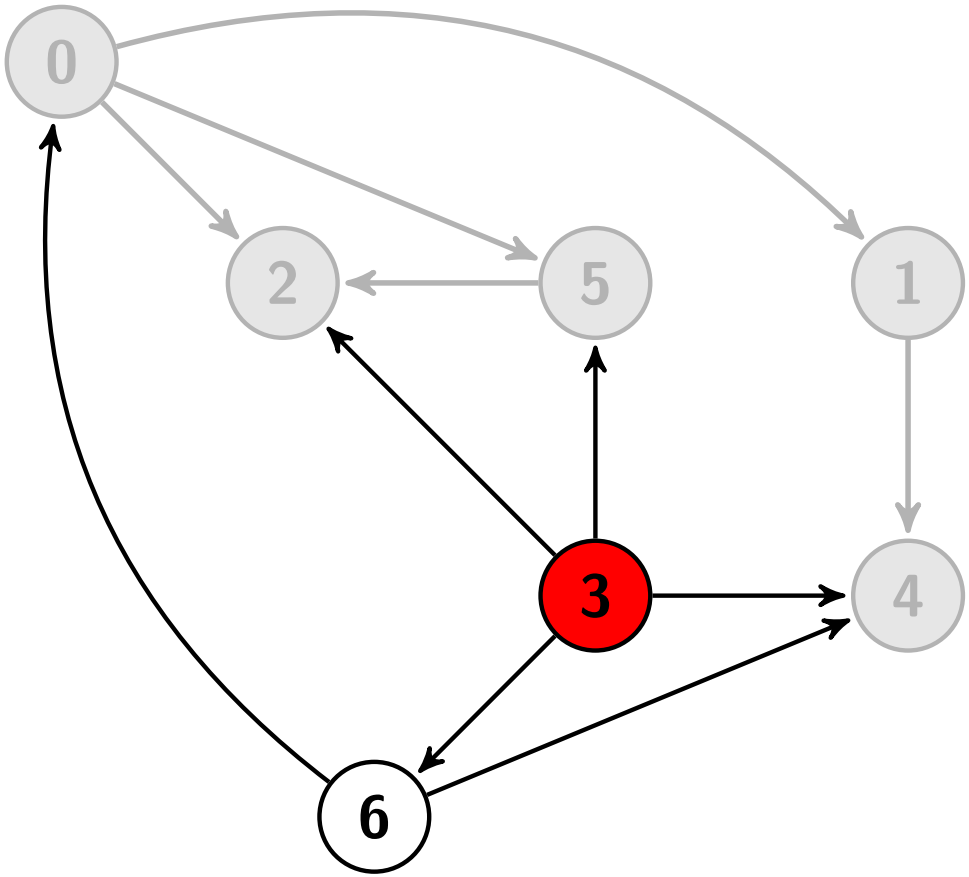
Pós-ordem - pilha

- $\{\mathbf{4, 1, 2, 5, 0}\}$

Ação

- Terminou a pós-ordem a partir de **0**
- Ainda temos **vértices não marcados**.

Ordenação topológica



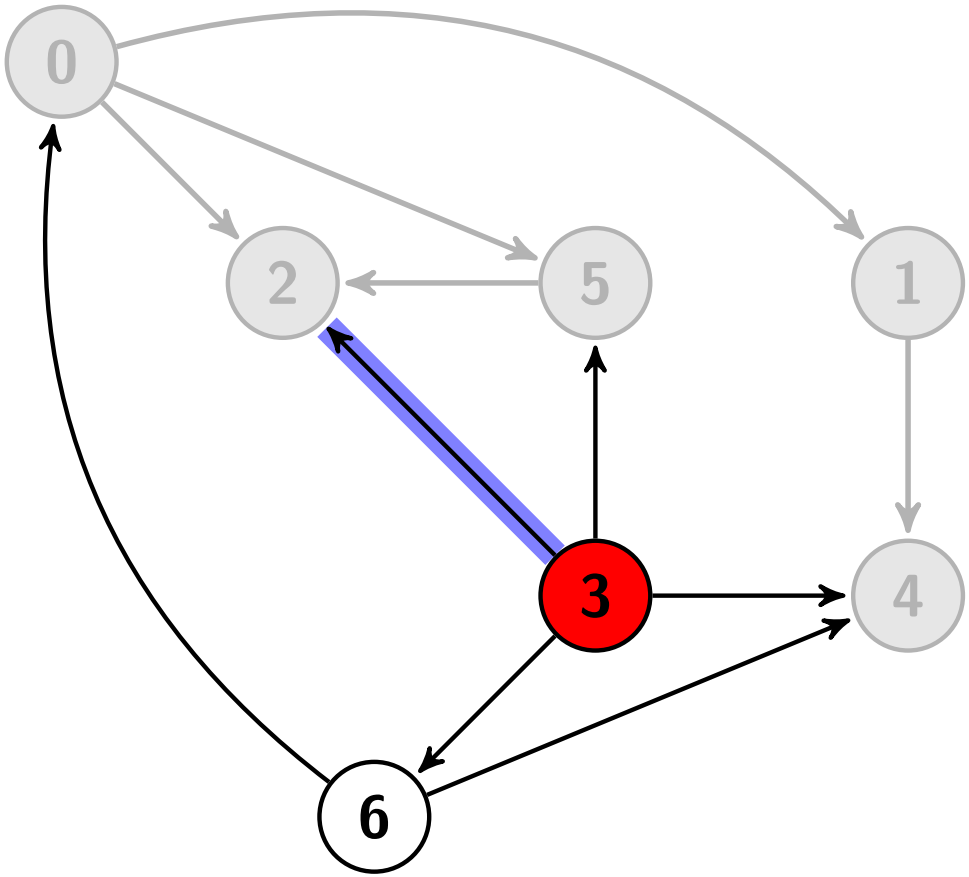
Pós-ordem - pilha

- $\{\mathbf{4, 1, 2, 5, 0}\}$

Ação

- Procurar próximo não marcado: **3**
- visita 3; avaliar 2; avaliar 4; avaliar 5; e avaliar 6.

Ordenação topológica



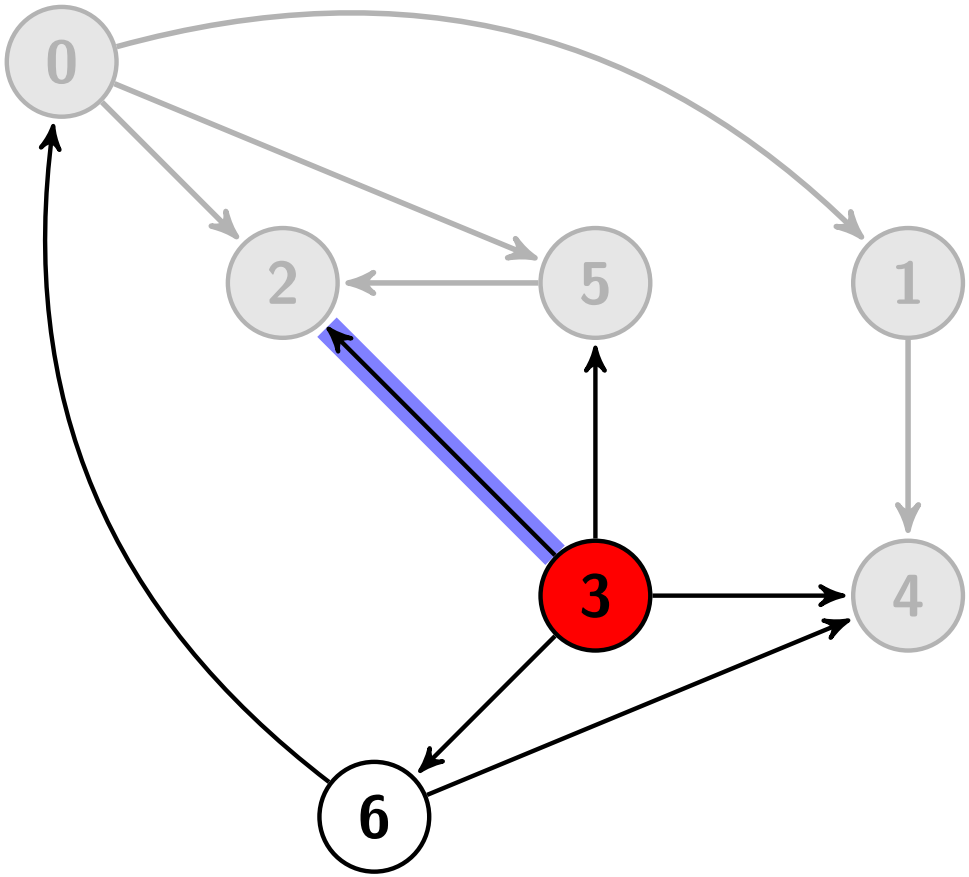
Pós-ordem - pilha

- $\{\mathbf{4, 1, 2, 5, 0}\}$

Ação

- visita 3; avaliar **2**; avaliar 4; avaliar 5; e avaliar 6.

Ordenação topológica



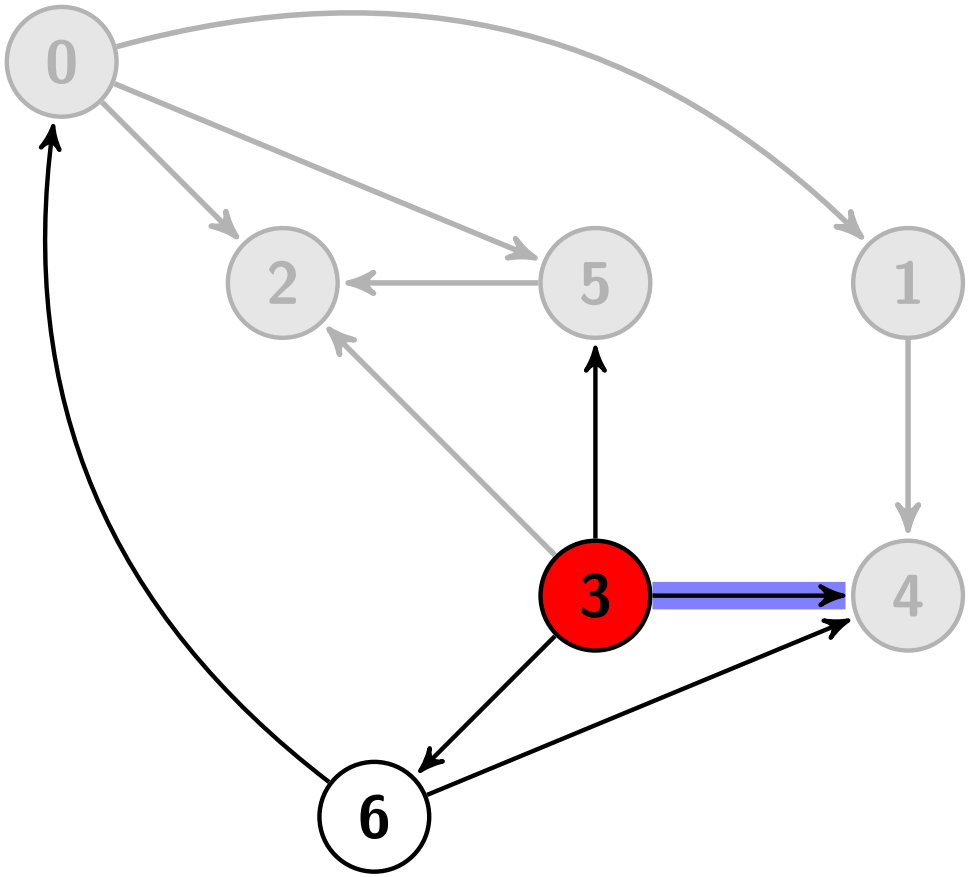
Pós-ordem - pilha

- $\{\mathbf{4, 1, 2, 5, 0}\}$

Ação

- visita 3; avaliar **2**; avaliar 4; avaliar 5; e avaliar 6.
- **2** está marcado.

Ordenação topológica



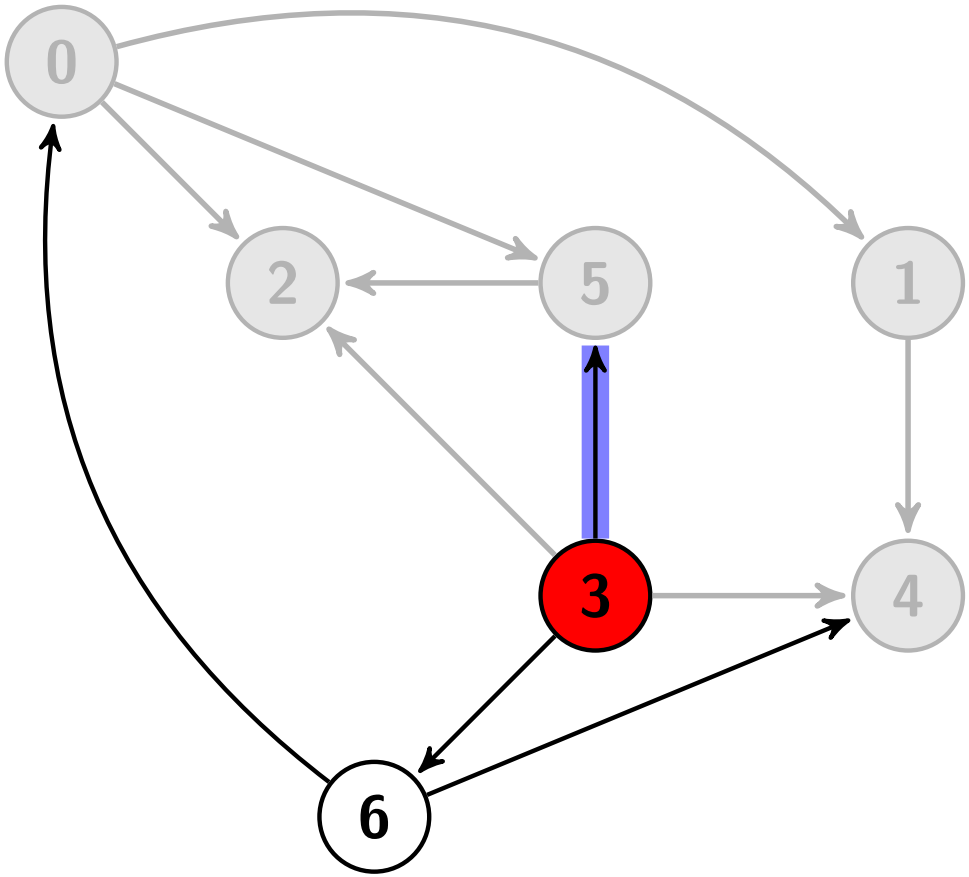
Pós-ordem - pilha

- $\{\mathbf{4, 1, 2, 5, 0}\}$

Ação

- visita 3; avaliar 2;avaliar **4**; avaliar 5; e avaliar 6.
- **4** está marcado.

Ordenação topológica



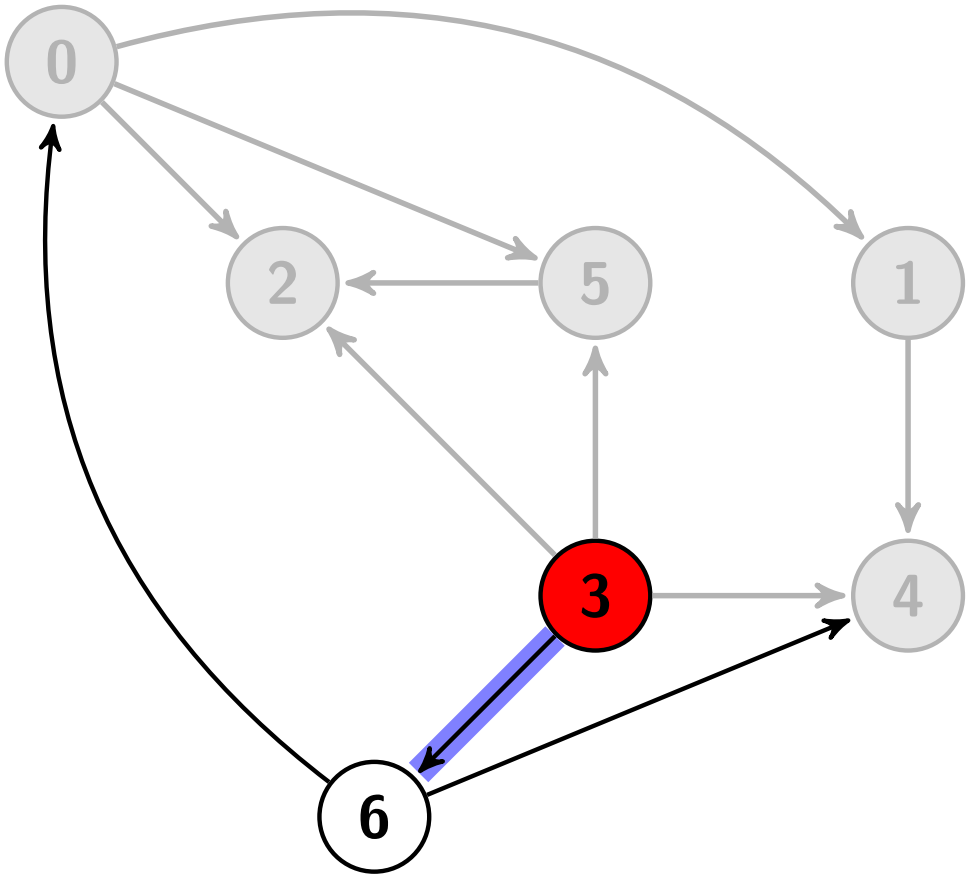
Pós-ordem - pilha

- $\{\mathbf{4, 1, 2, 5, 0}\}$

Ação

- visita 3; avaliar 2; avaliar 4; avaliar **5**; e avaliar 6.
- **5** está marcado.

Ordenação topológica



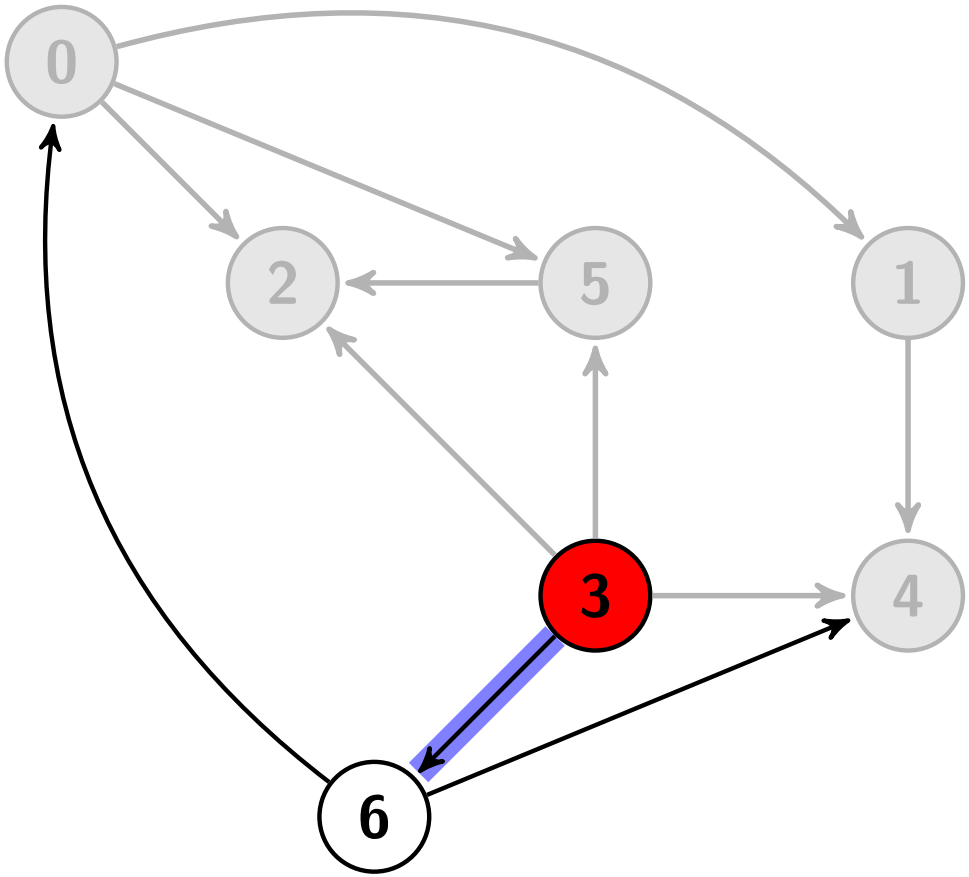
Pós-ordem - pilha

- $\{\mathbf{4, 1, 2, 5, 0}\}$

Ação

- visita 3; avaliar 2; avaliar 4; avaliar 5; e avaliar **6**.

Ordenação topológica



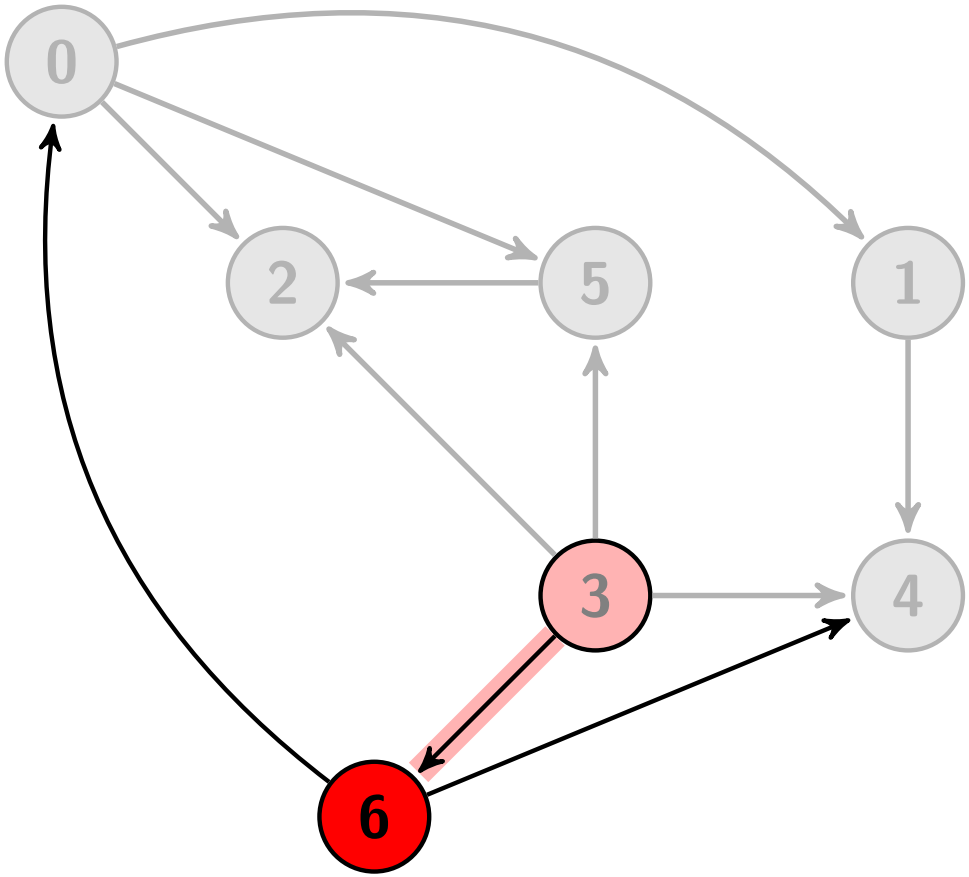
Pós-ordem - pilha

- $\{\mathbf{4, 1, 2, 5, 0}\}$

Ação

- visita 3; avaliar 2; avaliar 4; avaliar 5; e avaliar **6**.
- vértice **6 não** está marcado.

Ordenação topológica



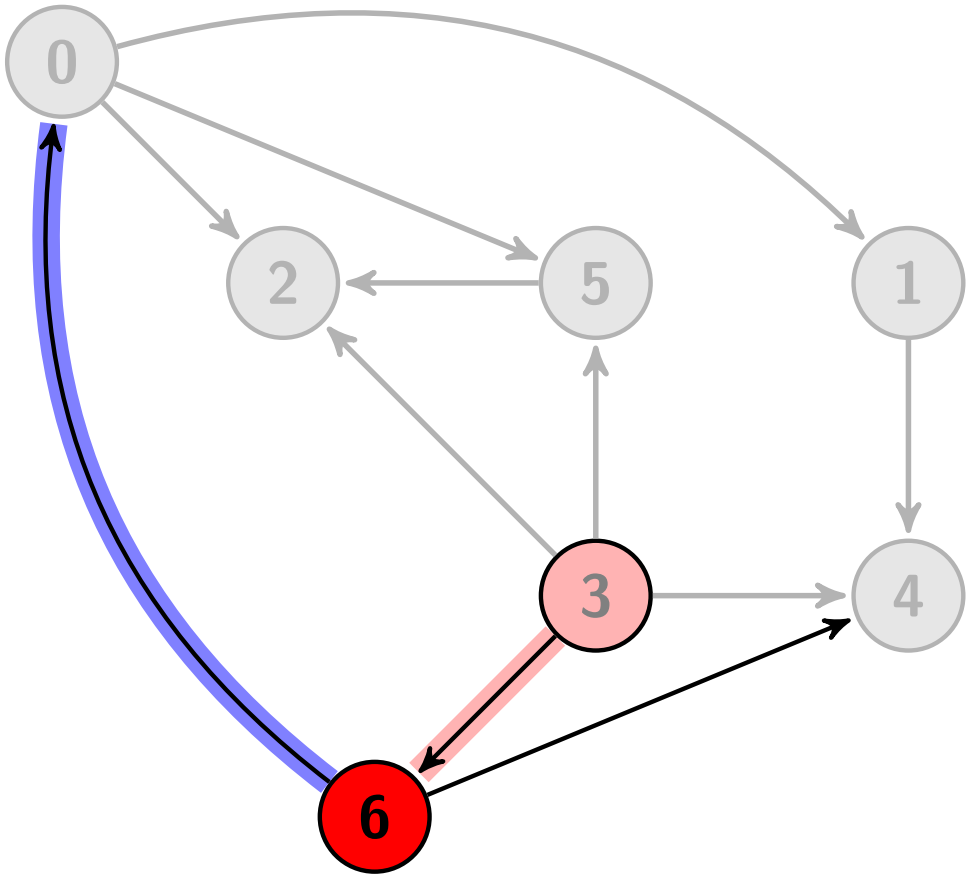
Pós-ordem - pilha

- $\{\mathbf{4, 1, 2, 5, 0}\}$

Ação

- visita 6; avaliar 0; e avaliar 4.

Ordenação topológica



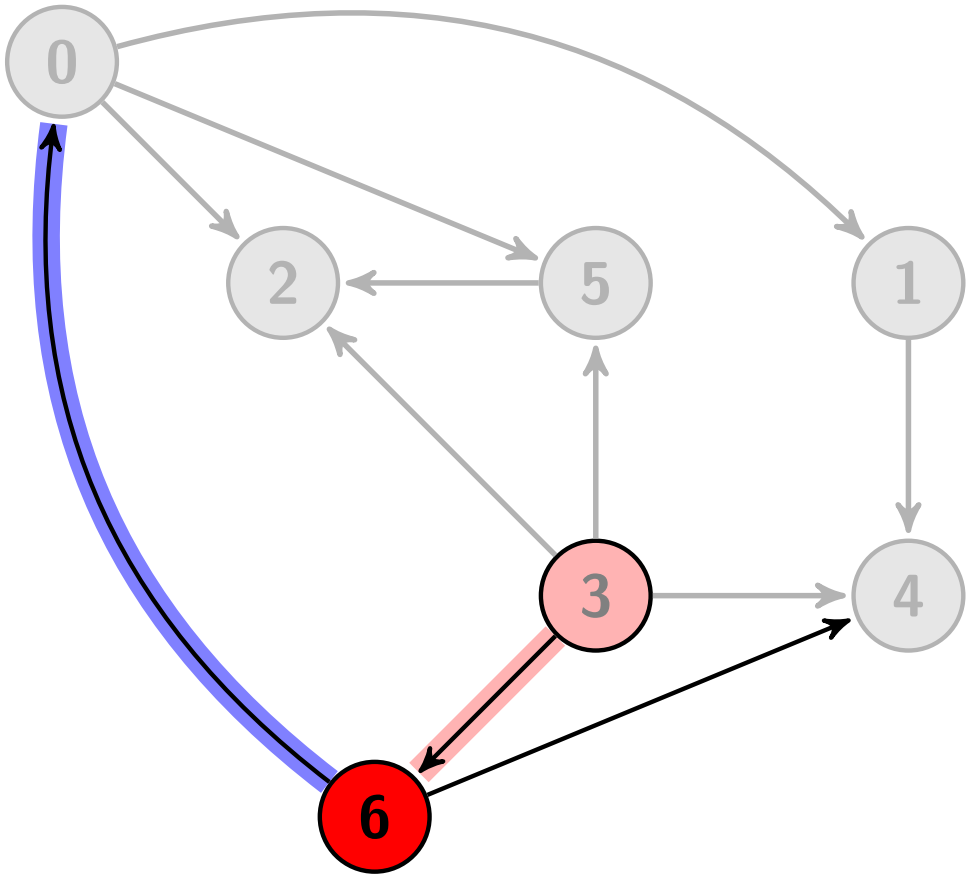
Pós-ordem - pilha

- $\{\mathbf{4, 1, 2, 5, 0}\}$

Ação

- visita 6; avaliar 0; e avaliar 4.

Ordenação topológica



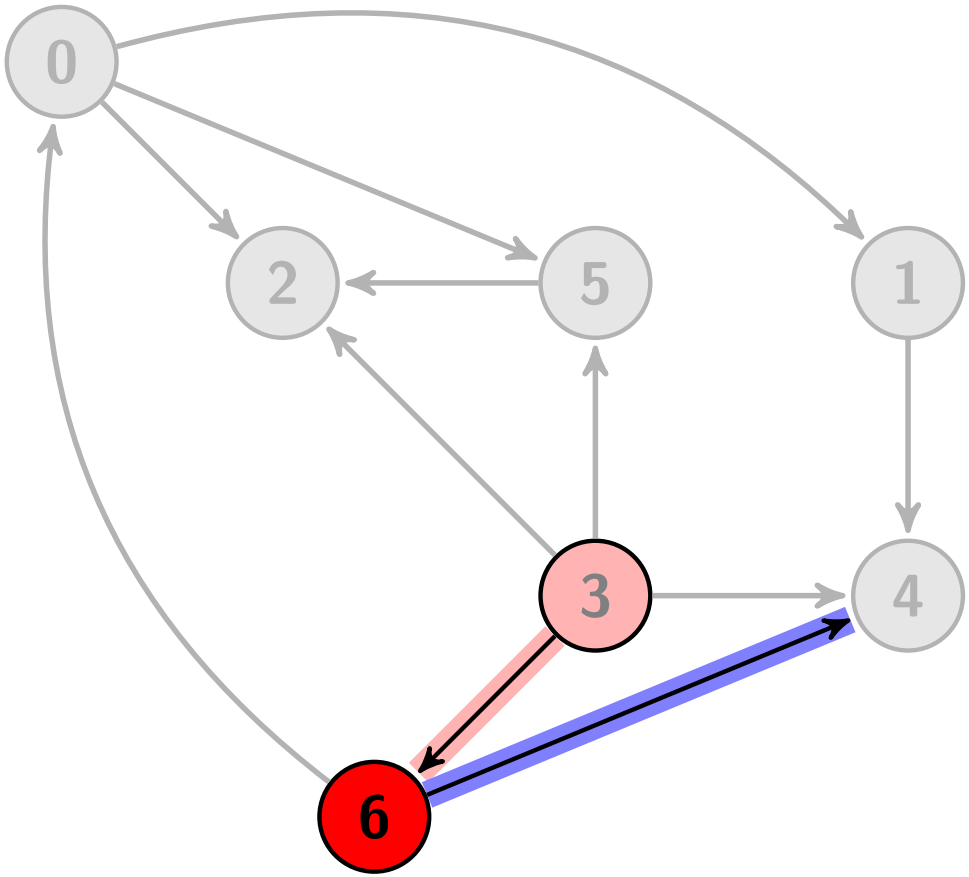
Pós-ordem - pilha

- $\{\mathbf{4, 1, 2, 5, 0}\}$

Ação

- visita 6; avaliar 0; e avaliar 4.
- vértice **0** está marcado.

Ordenação topológica



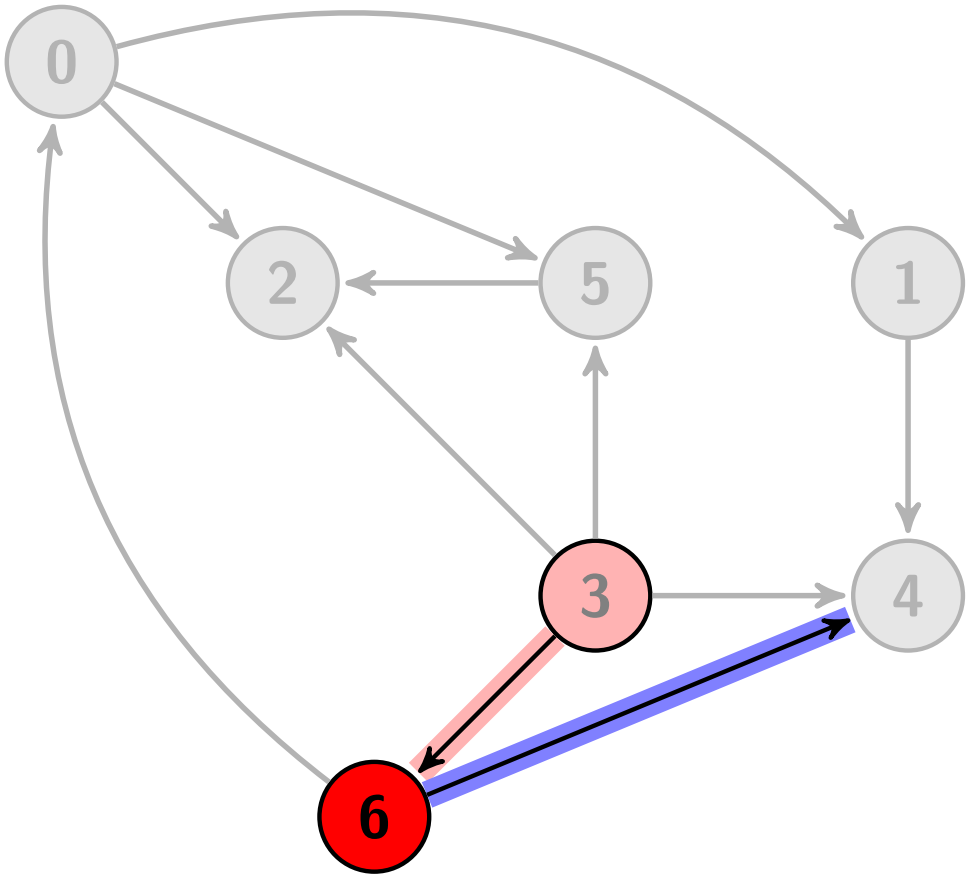
Pós-ordem - pilha

- $\{\mathbf{4, 1, 2, 5, 0}\}$

Ação

- visita 6; avaliar 0; e avaliar 4.

Ordenação topológica



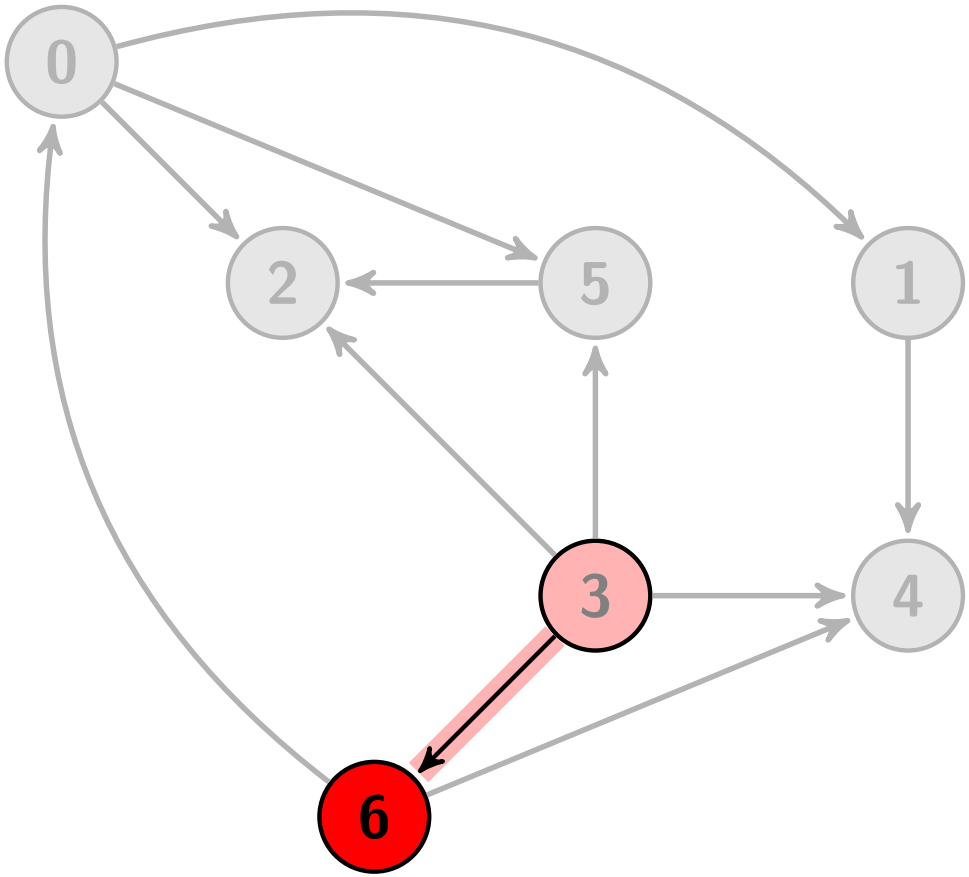
Pós-ordem - pilha

- $\{\mathbf{4, 1, 2, 5, 0}\}$

Ação

- visita 6; avaliar 0; e avaliar 4.
- vértice **4** está marcado.

Ordenação topológica



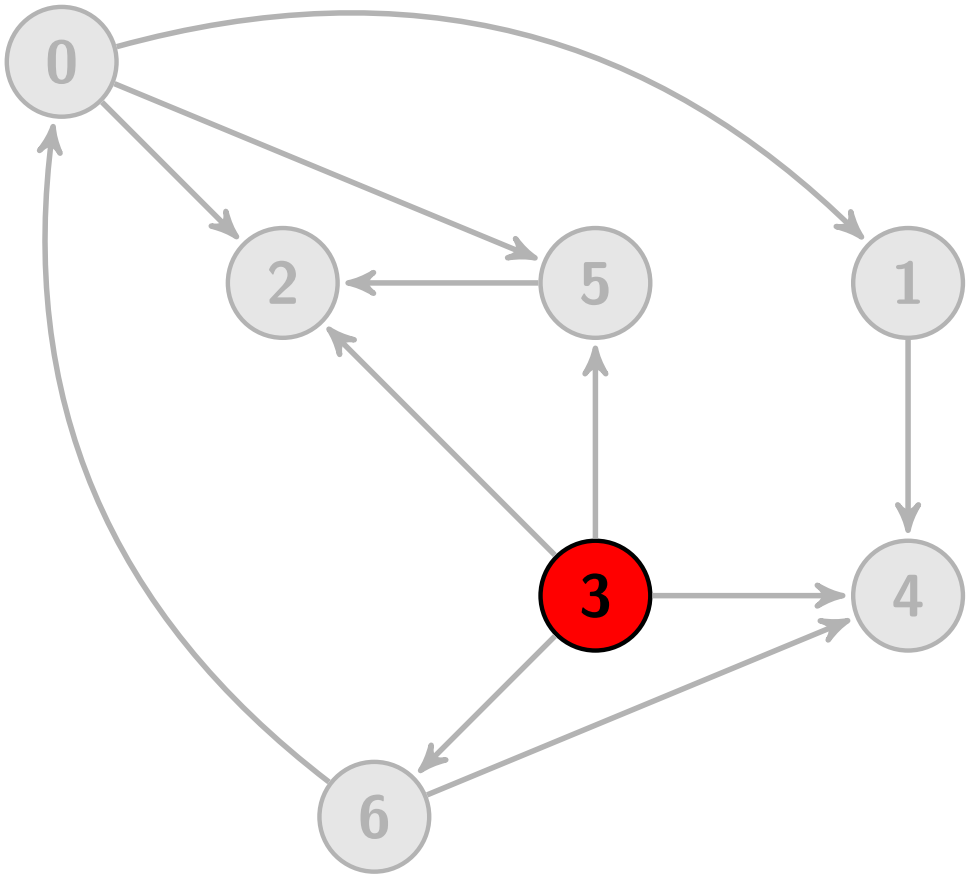
Pós-ordem - pilha

- $\{\mathbf{4, 1, 2, 5, 0, 6}\}$

Ação

- visita 6; avaliar 0; e avaliar 4.
- Terminou o **6**. Empilha.

Ordenação topológica



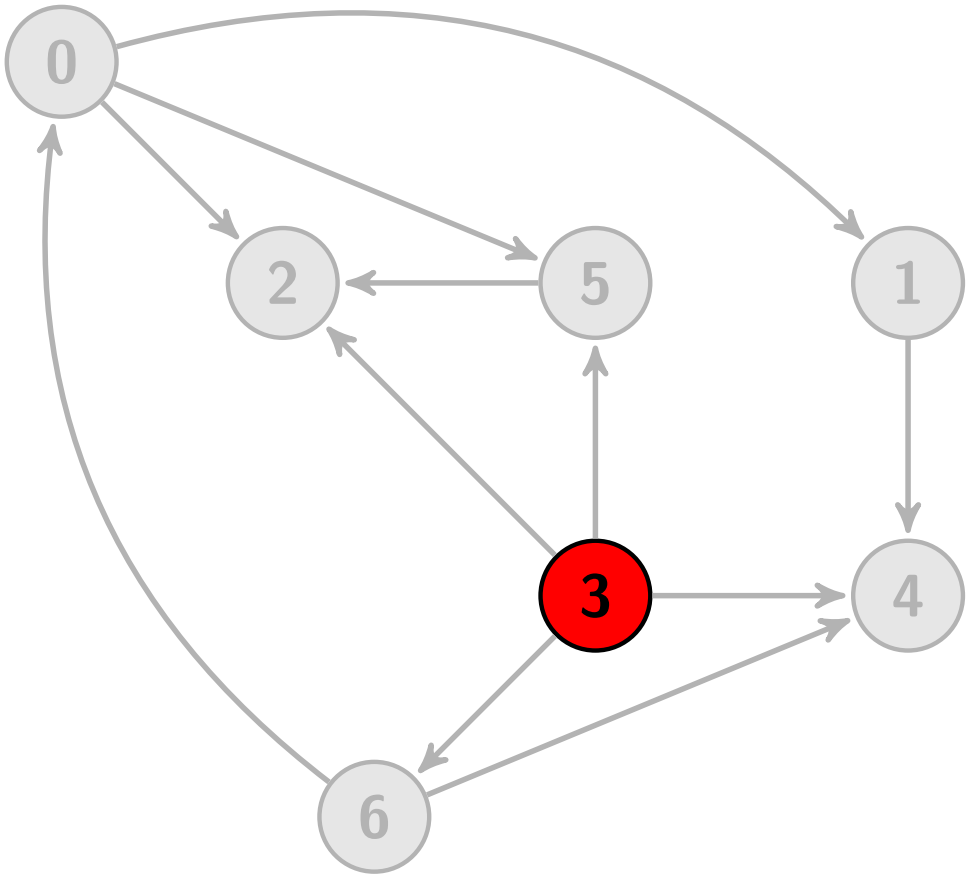
Pós-ordem - pilha

- $\{\mathbf{4, 1, 2, 5, 0, 6}\}$

Ação

- Terminou o 6.
- Retorna na recursão.

Ordenação topológica



Pós-ordem - pilha

- $\{\mathbf{4, 1, 2, 5, 0, 6, 3}\}$

Ação

- Terminou o **3**. Empilha.
- Retorna nas recursões e fim do algoritmo.

Problema do menor caminho

- Roteamento em mapas
- Navegação de robôs
- Planejamento de tráfico urbano
- Planejamento de atividades em uma empresa
- Algoritmos de protocolos de roteamento
- Redimensionamento inteligente de imagens

É necessário

- Em dígrafos sem pesos usar BFS
- Em dígrafos com pesos, necessário representar pesos em arestas
- Caminhar no dígrafo e atualizar distâncias
  - ordem topológica
  - algoritmo de Dijkstra

Menor Caminho em Dígrafos Sem Pesos

Menor caminho a partir de vértice de origem **s**

- Colocar **s** em uma FIFO e **marcar** como *visitado*
- Repetir até terminar FIFO
  - remover um vértice **v** da FIFO
  - Para cada vértice não marcado **u** partindo de **v**
    - adicionar **u** e marcá-lo como visitado
    - guardar a distância até **u** como 1 mais a distância até **v**

BFS para menor caminho sem pesos

```
1  void bfs(Digraph G, int s) {
2    Queue<Integer> q = new Queue<Integer>();
3    marked[s] = true;
4    distTo[s] = 0;
5    q.enqueue(s);
6    while (!q.isEmpty()) {
7      int v = q.dequeue();
8      for (int w : G.adj(v)) {
9        if (!marked[w]) {
10         edgeTo[w] = v;
11         distTo[w] = distTo[v] + 1;
12         marked[w] = true;
13         q.enqueue(w);
14       }
15     }
16   }
17 }
```

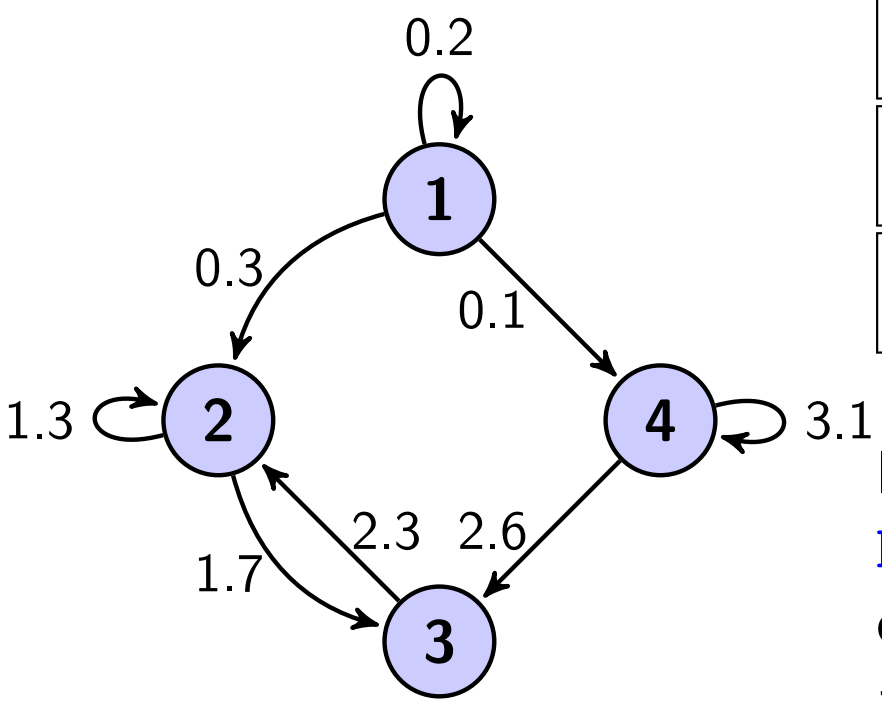
API: Arestas com pesos

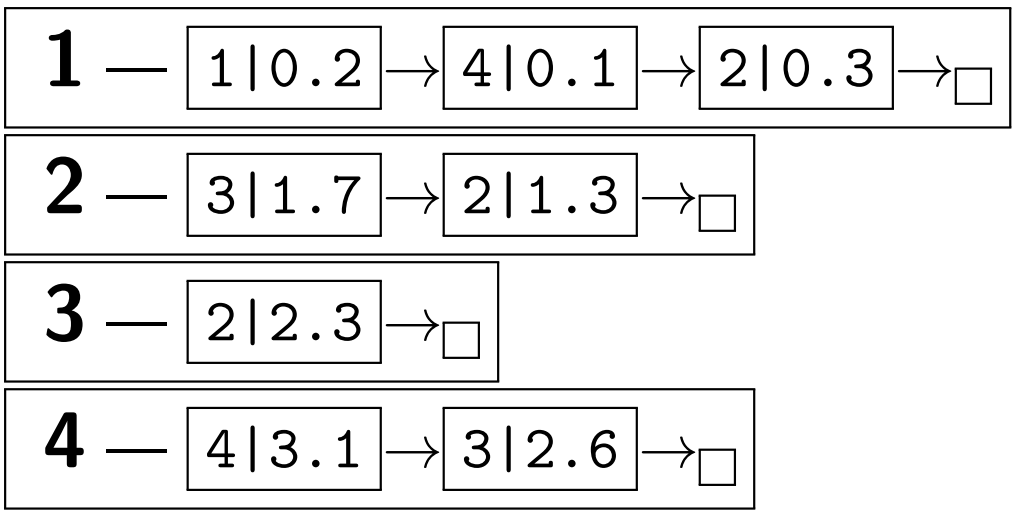
class	**Aresta**	implements Comparable<Aresta>
	Aresta(int v, int w, double peso)	criar aresta v-w com peso
int	de()	vértice origem da aresta
int	para()	vértice destino da aresta
double	peso()	peso associado
int	compareTo(Aresta a)	compara pesos

API de Dígrafos com pesos

public class	**DigrafoComPeso**	
	DigrafoComPeso(int V)	criar dígrafo vazio com V vértices
void	novaAresta(Aresta a)	criar aresta
Iterable<Aresta>	adj(int v)	vértices adjacentes a v

Representação de um dígrafo com listas de adjacências com pesos





Implementação muito similar ao `Dígrafo`, porém ao inserir aresta, deve-se `criar uma aresta com peso` antes.

Menor caminho em dígrafos com pesos

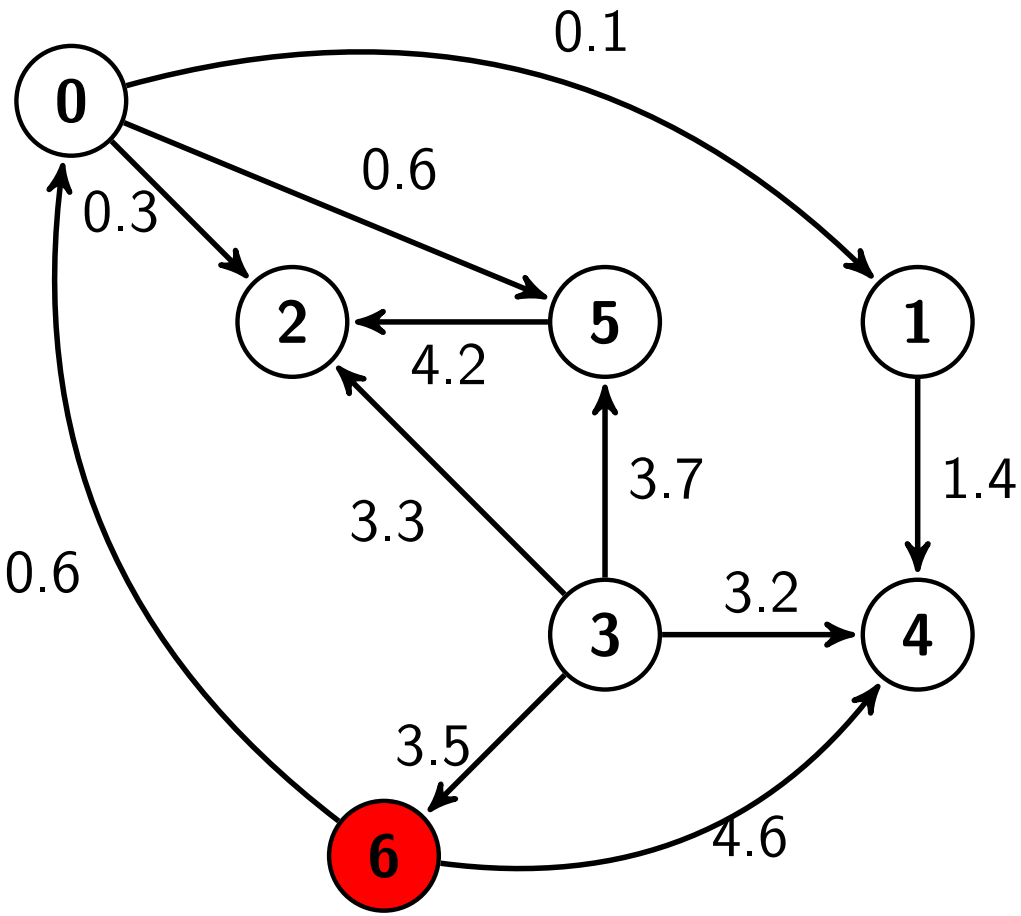
Existem diferentes modalidades

- Uma origem, todos os destinos
- Uma origem, um destino
- Todos os pares

Queremos o custo e o caminho.

```java
class MenorCaminho {//em dígrafo acíclico c/ pesos
 double[] distPara; Aresta[] arestaPara;

 public MenorCaminho(Digrafo G, int origem) {
  arestaPara = new Aresta[G.V()];
  distPara = new double[G.V()];

  for (int v = 0; v < G.V(); v++)
    distPara[v] = Double.POSITIVE_INFINITY;
  distPara[origem] = 0.0;

  OrdemTopologica topo = new OrdemTopologica(G);
  for (int v: topo.ordem())
    for (Aresta a: G.adj(v)) expandir(a);
 }

 void expandir(Aresta a) {
  int v = a.de(), w = a.para();
  if (distPara[w] > distPara[v] + a.peso()) {
    distPara[w] = distPara[v] + a.peso();
    arestaPara[w] = a;
  }
 }
}
```

Menor caminho

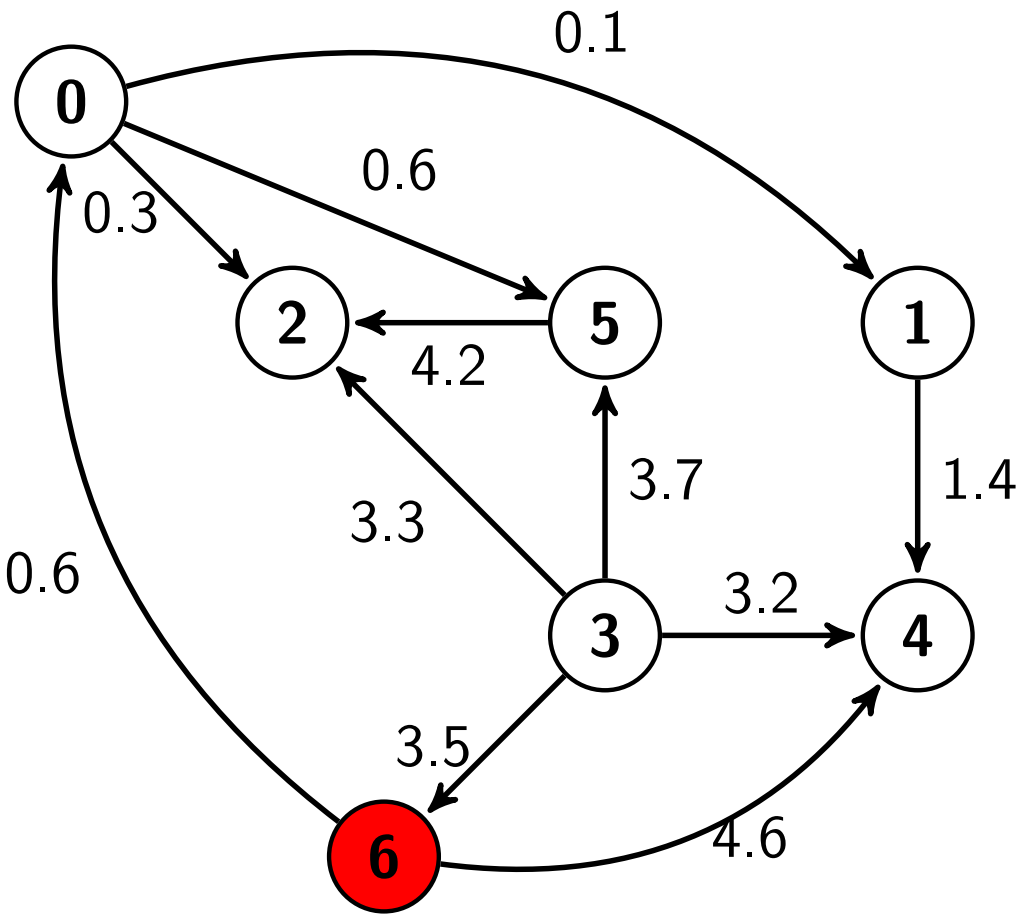


Pós-ordem topológica - pilha

- $\{\mathbf{4, 1, 2, 5, 0, 6, 3}\}$

	distPara	arestaPara
0	inf	
1	inf	
2	inf	
3	inf	
4	inf	
5	inf	
6	inf	

- **Origem** é o vértice 6

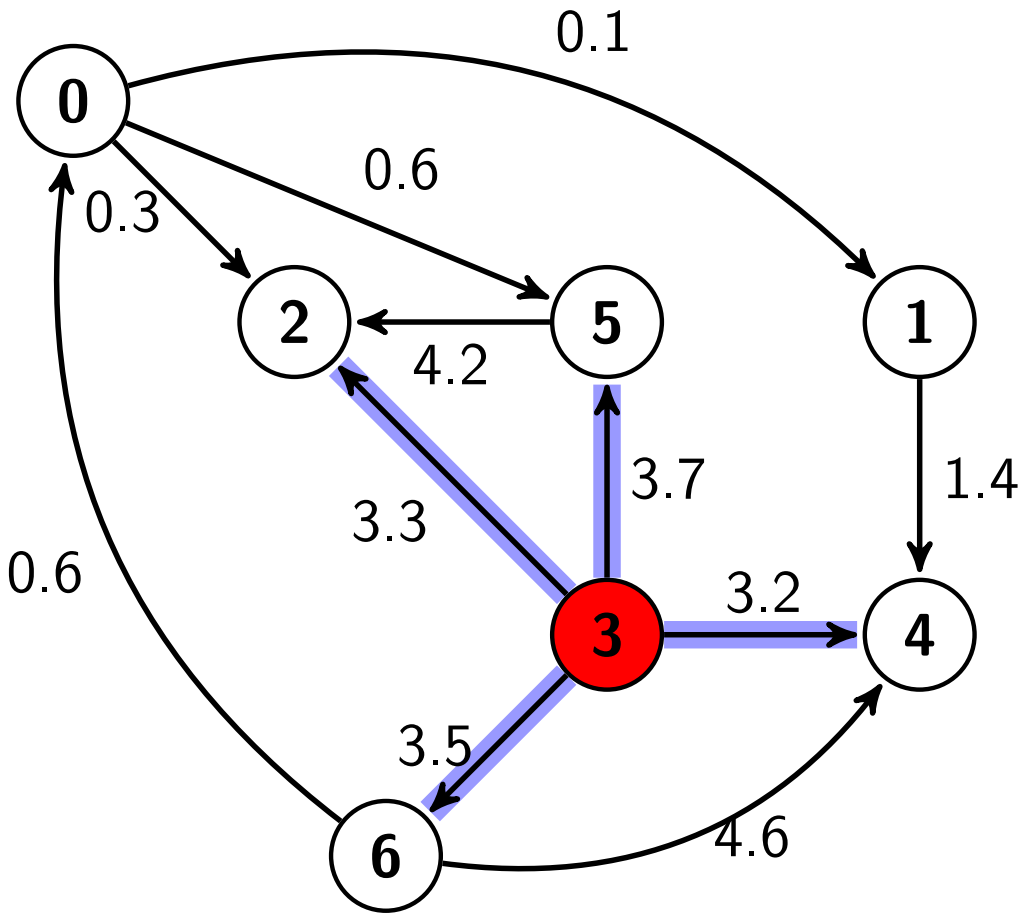


Ordem topológica - pilha

- {**4, 1, 2, 5, 0, 6, 3**}

	distPara	arestaPara
0	inf	
1	inf	
2	inf	
3	inf	
4	inf	
5	inf	
6	0	

- Origem é o vértice 6
- **Distância à origem é 0**

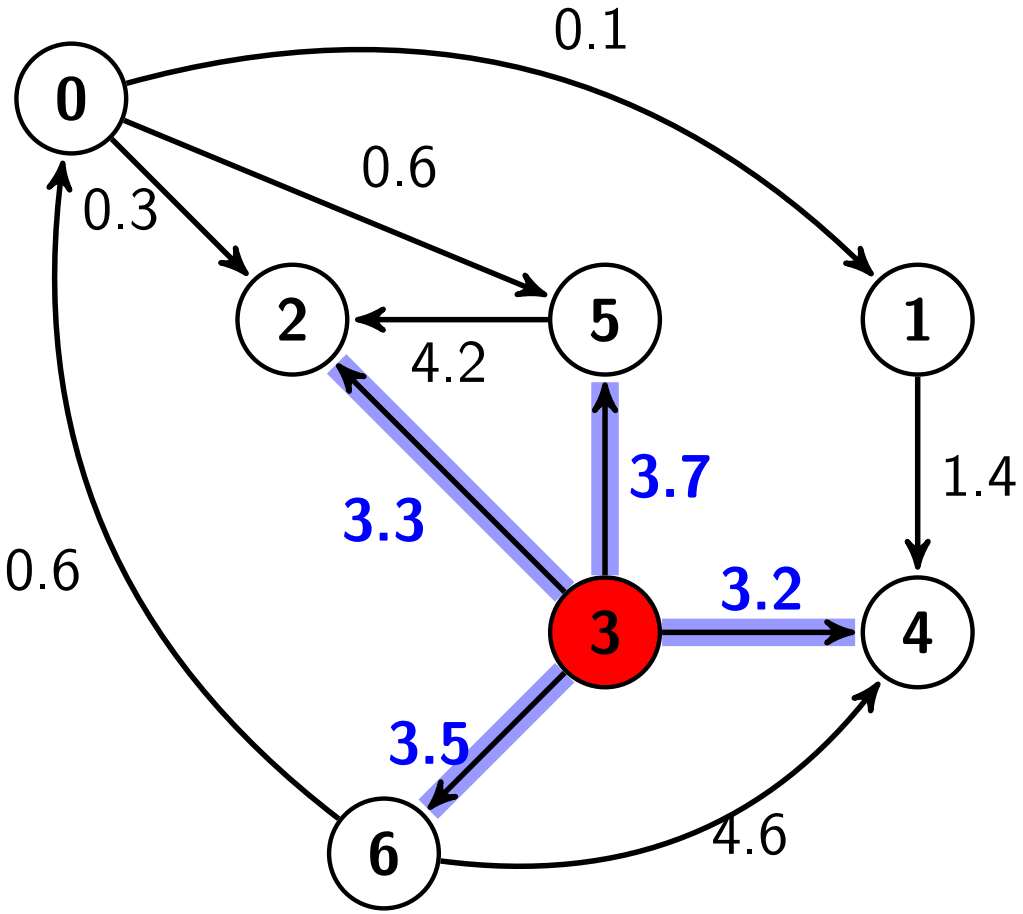


Ordem topológica - pilha

- {**4, 1, 2, 5, 0, 6, 3**}

	distPara	**arestaPara**
0	inf	
1	inf	
2	inf	
3	inf	
4	inf	
5	inf	
6	0	

- Pegar da pilha um vértice: **3**
- Expandir arestas de **3** para **2**, **4**, **5**, **6**



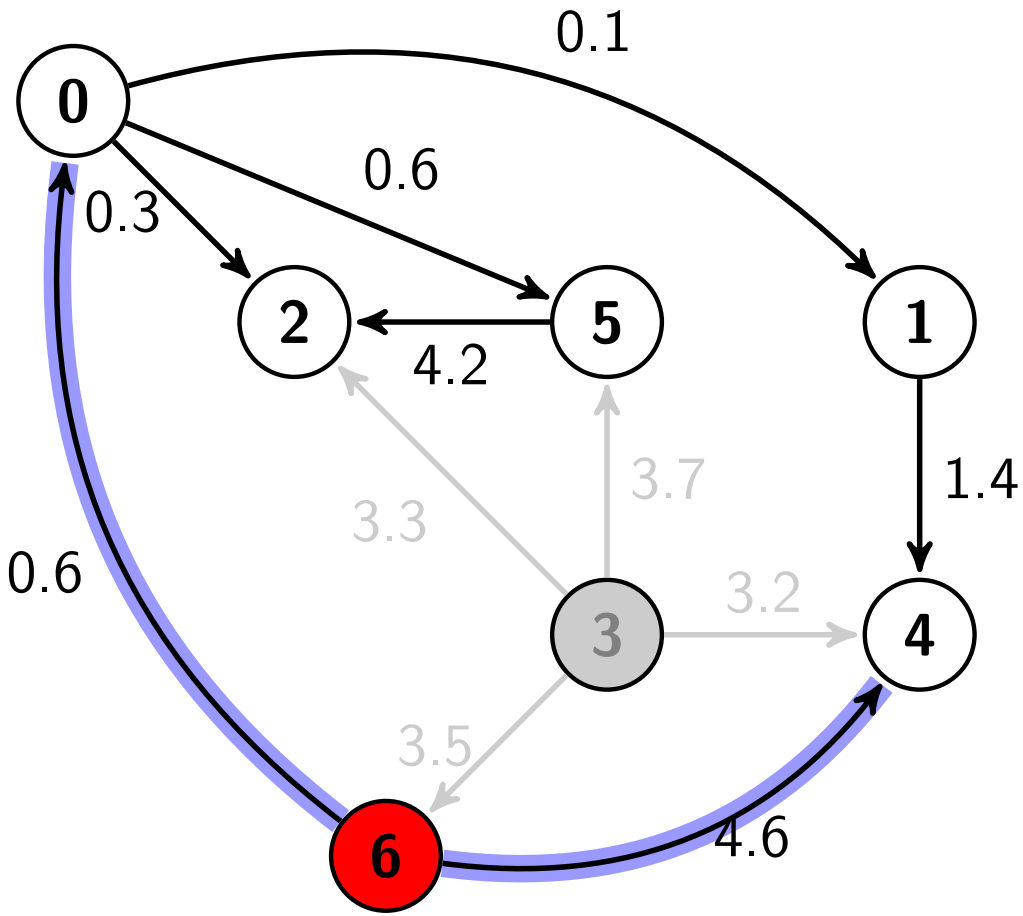

```
1 if (distPara[w]>distPara[v]+a.peso()){
2   distPara[w] = distPara[v] + a.peso();
3   arestaPara[w] = a;
4 }
```

Ordem topológica - pilha

- {**4**, **1**, **2**, **5**, **0**, **6**, **3**}

	distPara	arestaPara
0	inf	
1	inf	
w **2**	inf	
v **3**	inf	
w **4**	inf	
w **5**	inf	
w **6**	0	

- Expandir arestas de **3** para **2**, **4**, **5**, **6**
- Tabela **não** é alterada.

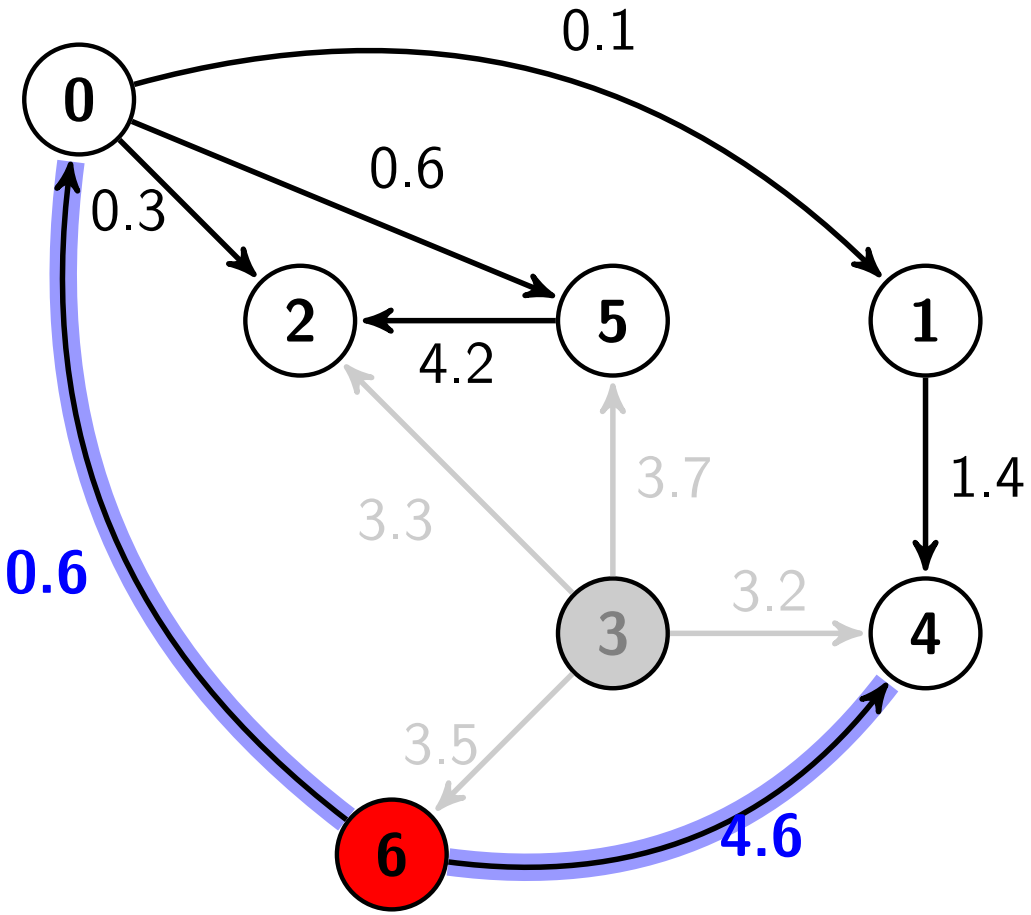


Ordem topológica - pilha

- {**4, 1, 2, 5, 0, 6**}

	distPara	arestaPara
0	inf	
1	inf	
2	inf	
3	inf	
4	inf	
5	inf	
6	0	

- Pegar da pilha um vértice: **6**
- Expandir arestas de **6** para **0** e **4**



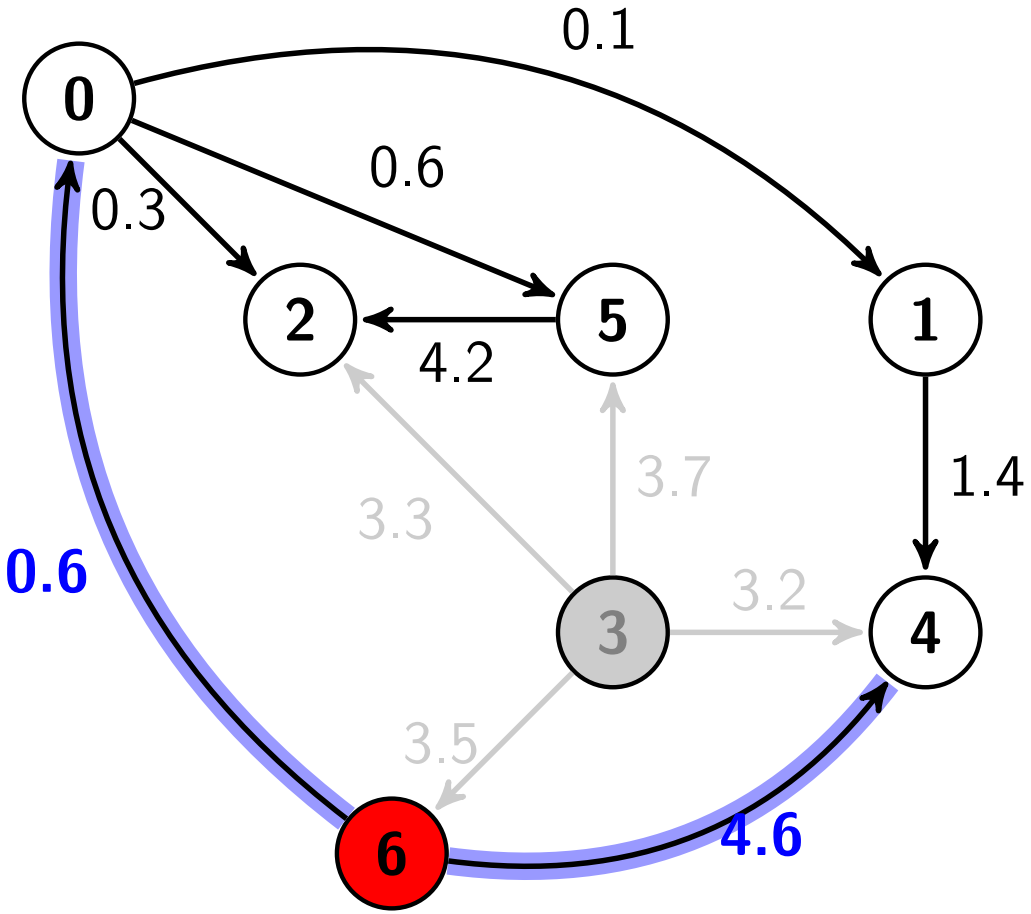
Ordem topológica - pilha

- {**4, 1, 2, 5, 0, 6**}

	distPara	arestaPara
w **0**	inf	
1	inf	
2	inf	
3	inf	
w **4**	inf	
5	inf	
v **6**	0	

```
1 if (distPara[w]>distPara[v]+a.peso()){
2   distPara[w] = distPara[v] + a.peso();
3   arestaPara[w] = a;
4 }
```

- Expandir de **6** para **0** e **4**



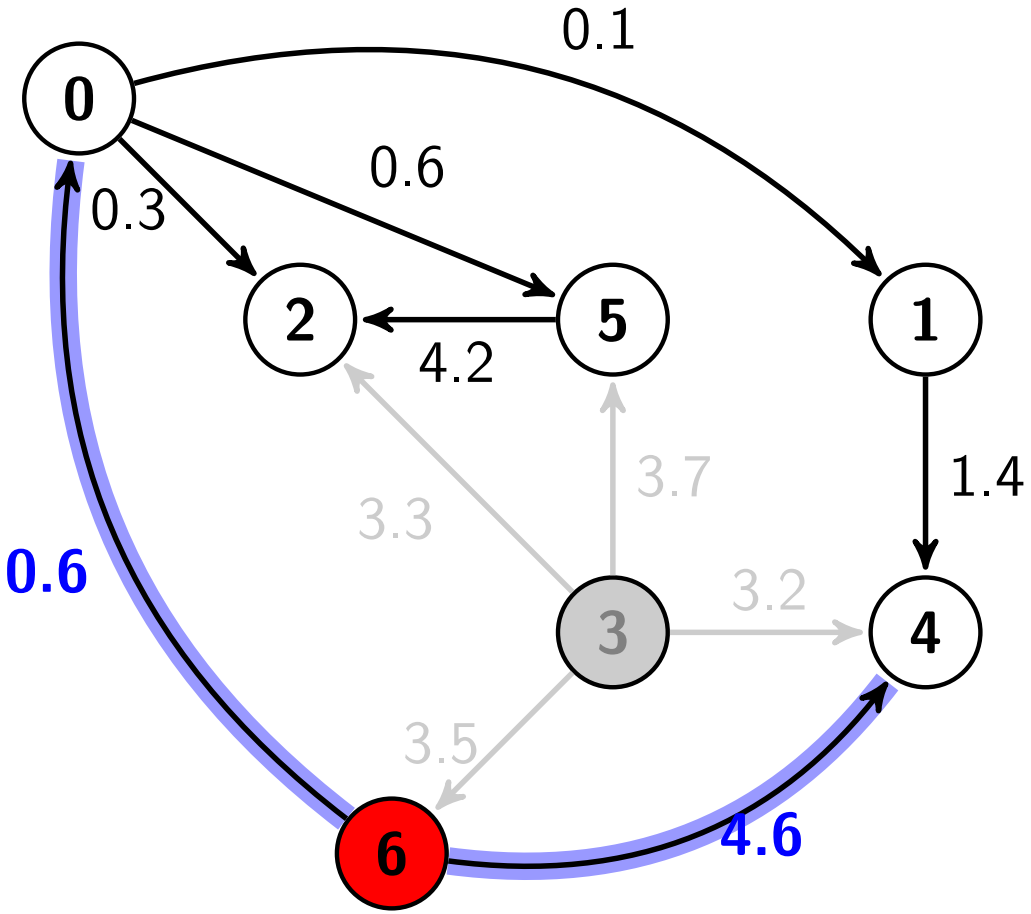
Ordem topológica - pilha

- {**4, 1, 2, 5, 0, 6**}

	distPara	arestaPara
w **0**	**0.6**	**6**
1	inf	
2	inf	
3	inf	
w **4**	inf	
5	inf	
v **6**	0	

```
1 if (distPara[w]>distPara[v]+a.peso()){
2   distPara[w] = distPara[v] + a.peso();
3   arestaPara[w] = a;
4 }
```

- Expandir de **6** para **0** e **4**



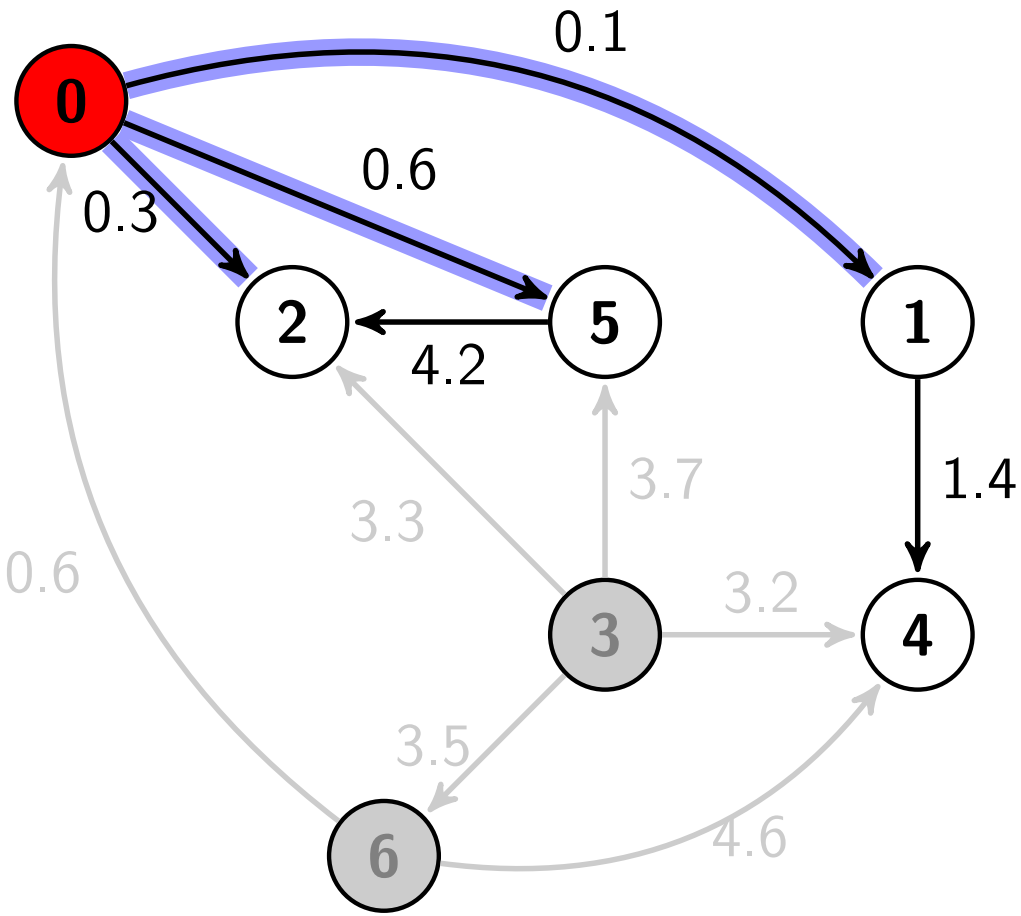
Ordem topológica - pilha

- {**4, 1, 2, 5, 0, 6**}

	distPara	arestaPara
w **0**	**0.6**	**6**
1	inf	
2	inf	
3	inf	
w **4**	**4.6**	**6**
5	inf	
v **6**	0	

```
1 if (distPara[w]>distPara[v]+a.peso()){
2   distPara[w] = distPara[v] + a.peso();
3   arestaPara[w] = a;
4 }
```

- Expandir de **6** para **0** e **4**

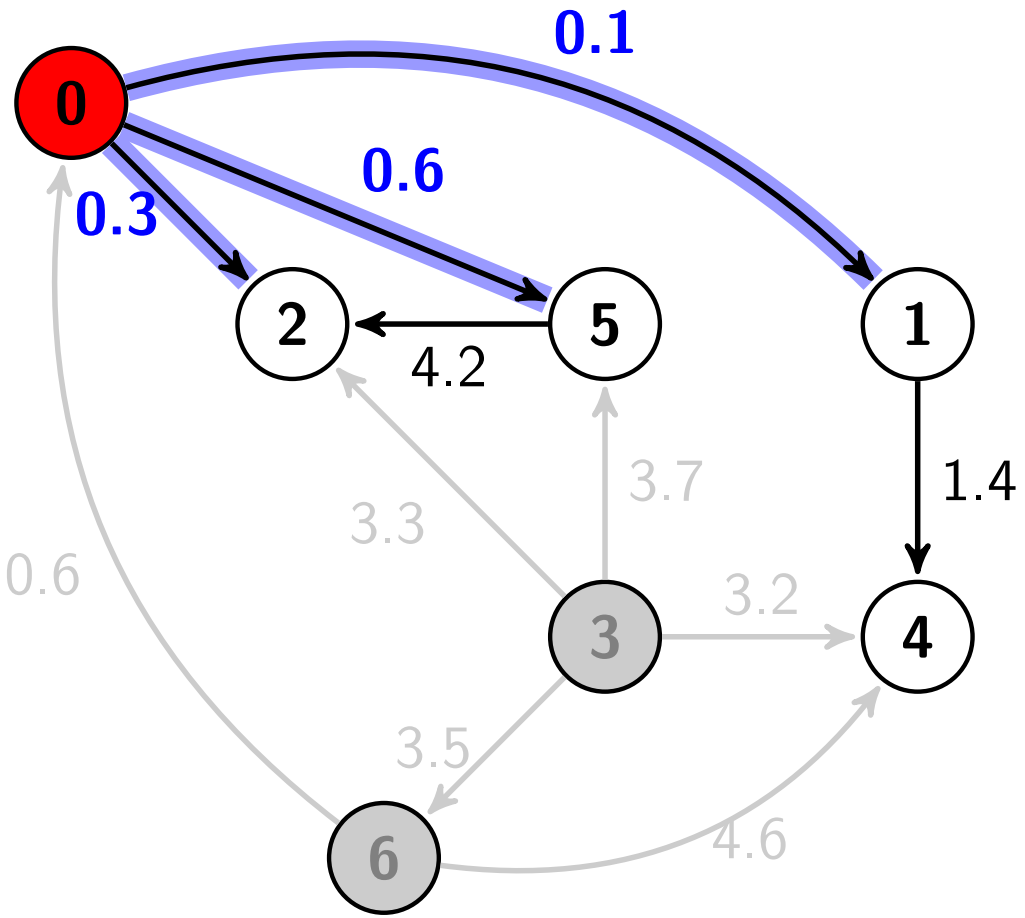


Ordem topológica - pilha

- {**4, 1, 2, 5, 0**}

	distPara	arestaPara
0	0.6	6
1	inf	
2	inf	
3	inf	
4	4.6	6
5	inf	
6	0	

- Pegar da pilha um vértice: **0**
- Expandir arestas de **0** para **1**, **2** e **5**



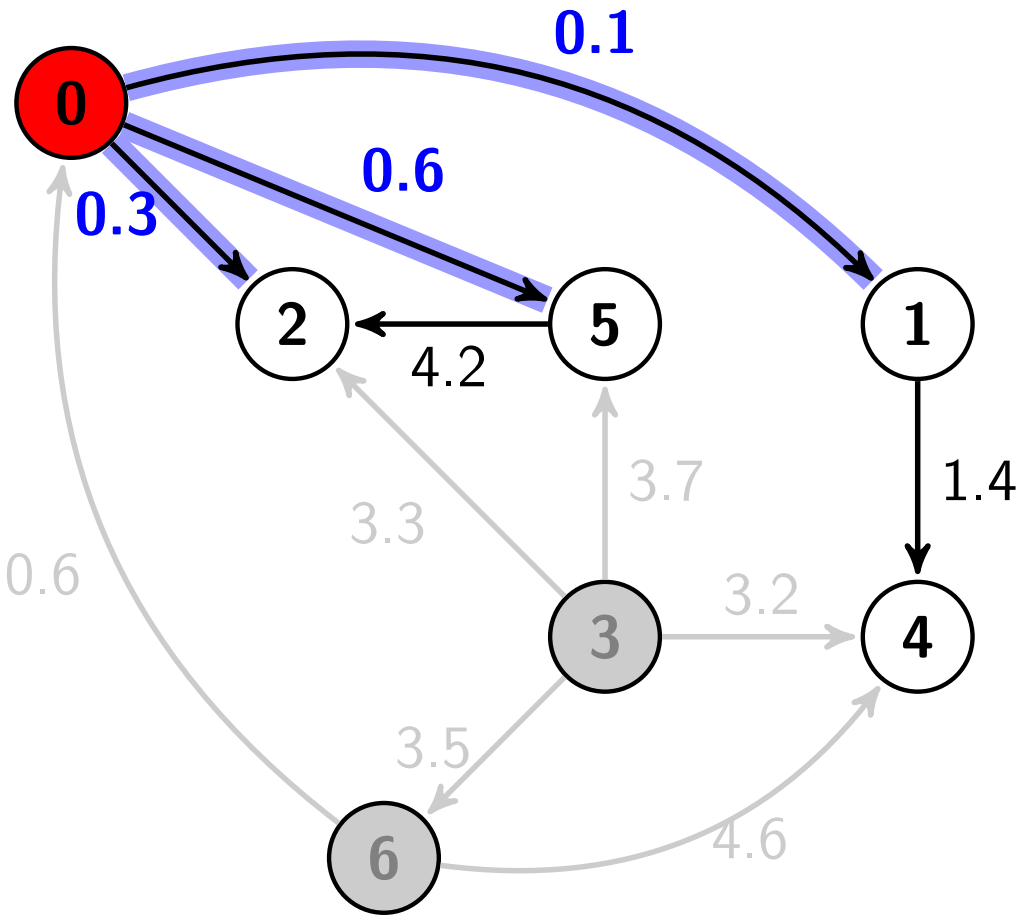
Ordem topológica - pilha

- {**4**, **1**, **2**, **5**, **0**}

	distPara	arestaPara
v 0	0.6	6
w 1	0.6 + 0.1	0
w 2	0.6 + 0.3	0
3	inf	
4	4.6	6
w 5	0.6 + 0.6	0
6	0	

```
1 if (distPara[w]>distPara[v]+a.peso()){
2   distPara[w] = distPara[v] + a.peso();
3   arestaPara[w] = a;
4 }
```

- Expandir de **0** para **1**, **2** e **5**



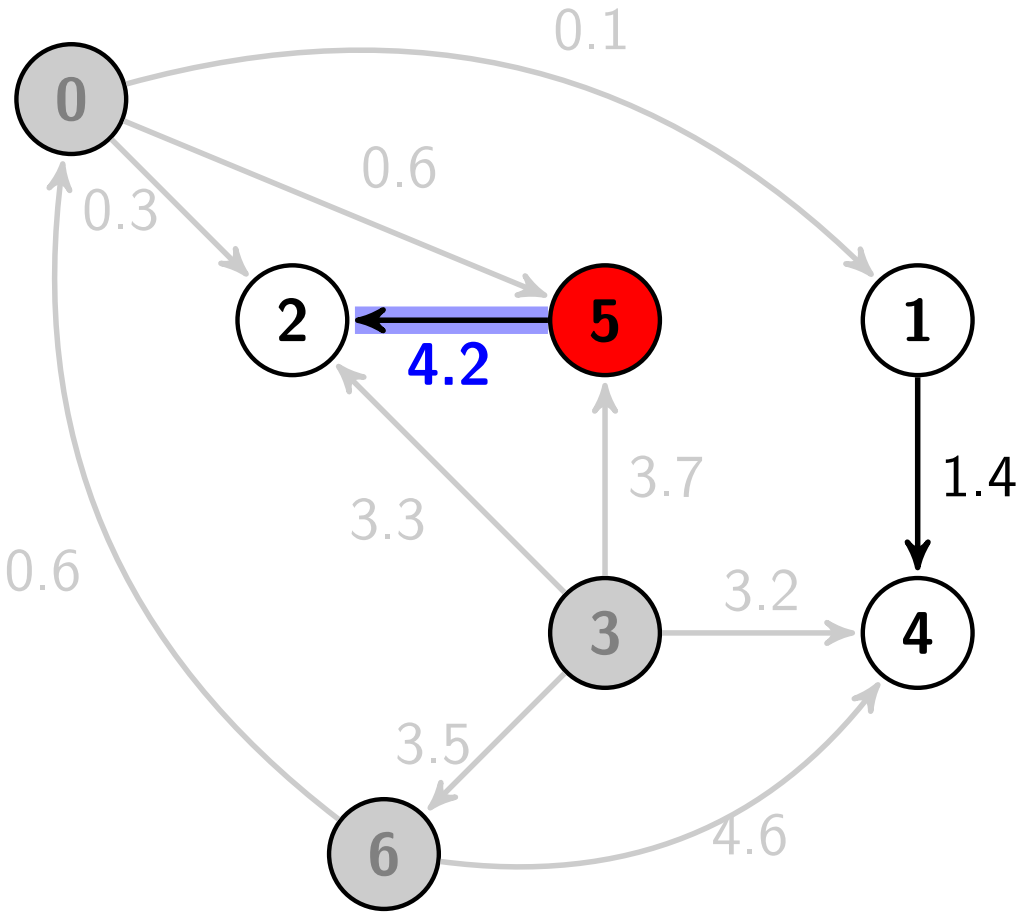
Ordem topológica - pilha

- {**4**, **1**, **2**, **5**, **0**}

	distPara	arestaPara
0	0.6	6
1	0.7	0
2	0.9	0
3	inf	
4	4.6	6
5	1.2	0
6	0	

```
1 if (distPara[w]>distPara[v]+a.peso()){
2   distPara[w] = distPara[v] + a.peso();
3   arestaPara[w] = a;
4 }
```

- Expandir de **0** para **1**, **2** e **5**

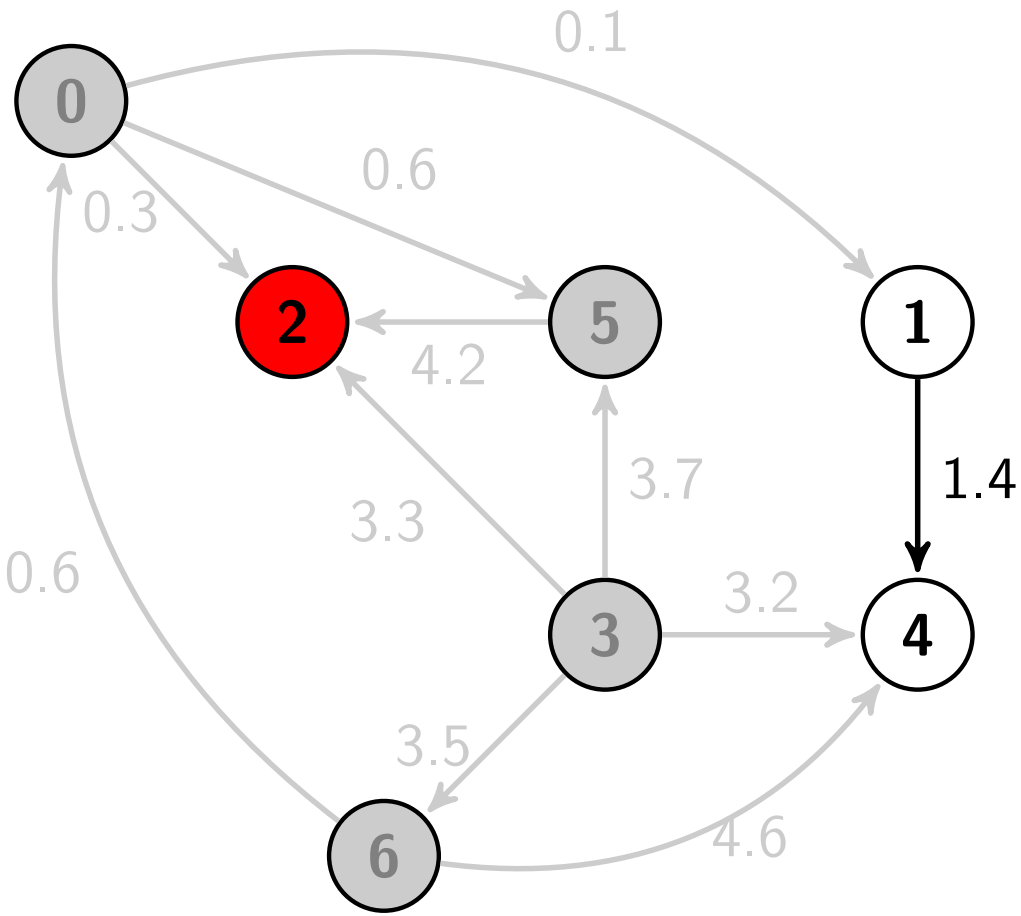


Ordem topológica - pilha

- $\{4, 1, 2, 5\}$

	distPara	arestaPara
0	0.6	6
1	0.7	0
2	0.9	0
3	inf	
4	4.6	6
5	1.2	0
6	0	

- Pegar da pilha um vértice: **5**
- Expandir arestas de **5** para **2**.
- Não muda nada: `0.9 < 1.2+4.2`!

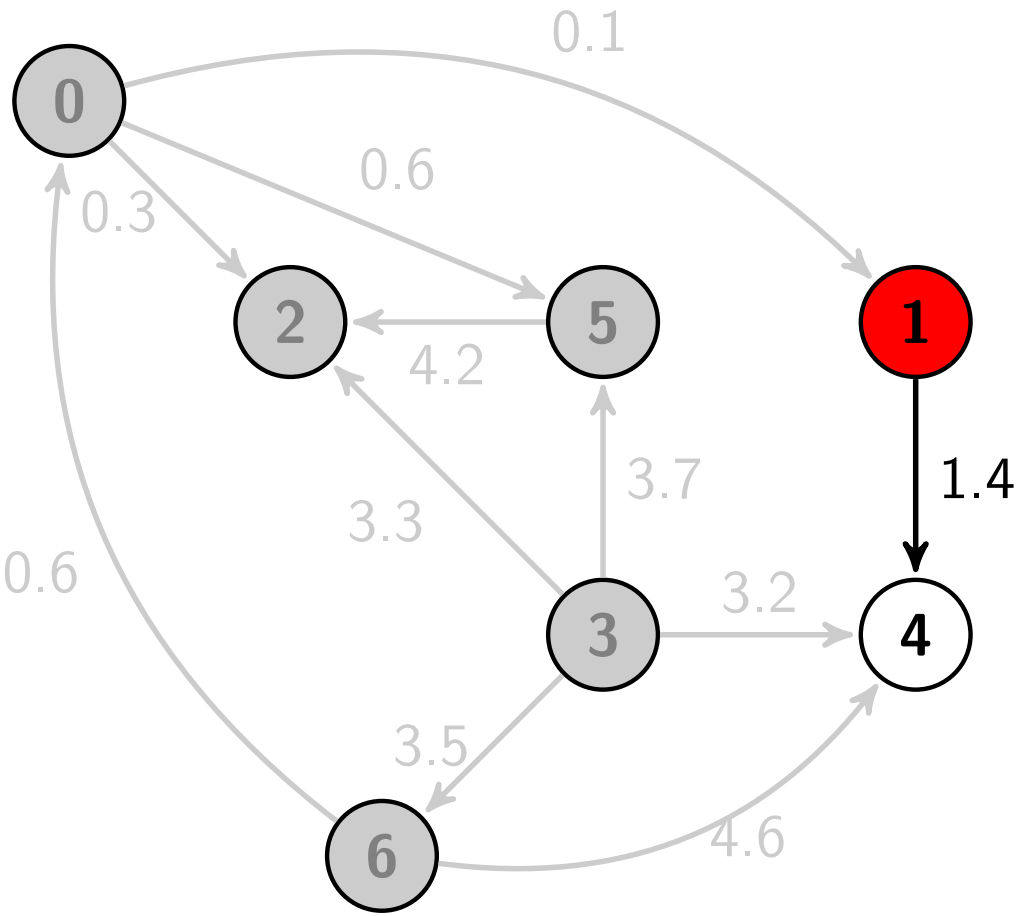


Ordem topológica - pilha

- $\{4, 1, 2\}$

	distPara	arestaPara
0	0.6	6
1	0.7	0
2	0.9	0
3	inf	
4	4.6	6
5	1.2	0
6	0	

- Pegar da pilha um vértice: **2**
- **Não** tem para onde expandir.

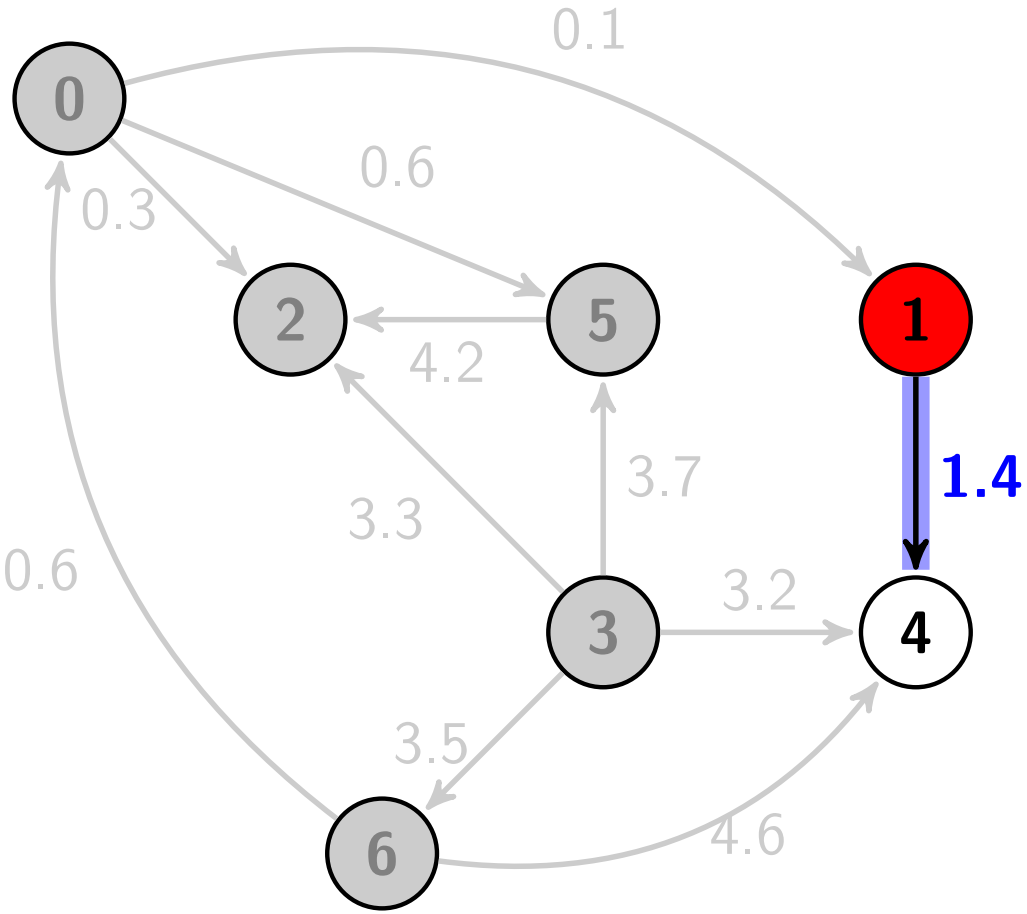


Ordem topológica - pilha

- {**4**, **1**}

	distPara	arestaPara
0	0.6	6
1	0.7	0
2	0.9	0
3	inf	
4	4.6	6
5	1.2	0
6	0	

- Pegar da pilha um vértice: **1**
- Expandir de **1** para **4**



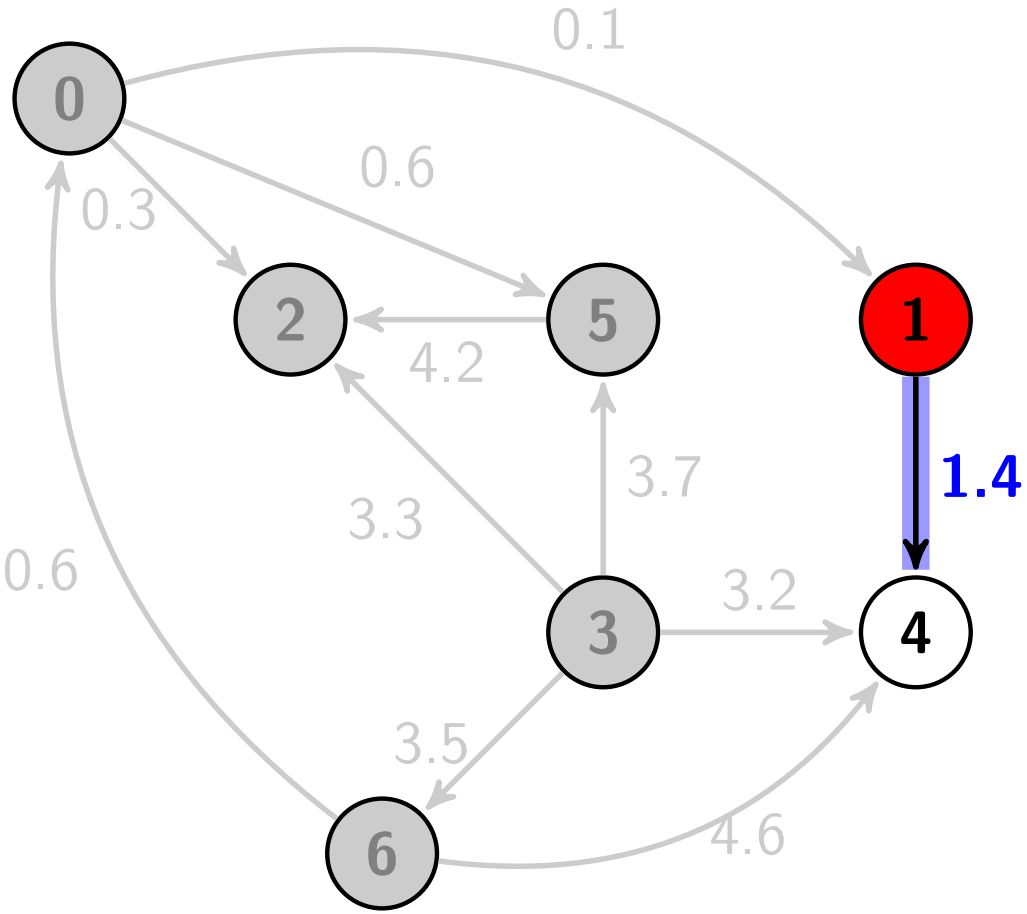
Ordem topológica - pilha

- $\{4, 1\}$

	distPara	arestaPara
0	0.6	6
v 1	0.7	0
2	0.9	0
3	inf	
w 4	4.6	6
5	1.2	0
6	0	

```
1 if (distPara[w]>distPara[v]+a.peso()){
2   distPara[w] = distPara[v] + a.peso();
3   arestaPara[w] = a;
4 }
```

- Expandir de **1** para **4**.
- `4.6 > 0.7 + 1.4`, atualizar tabela.



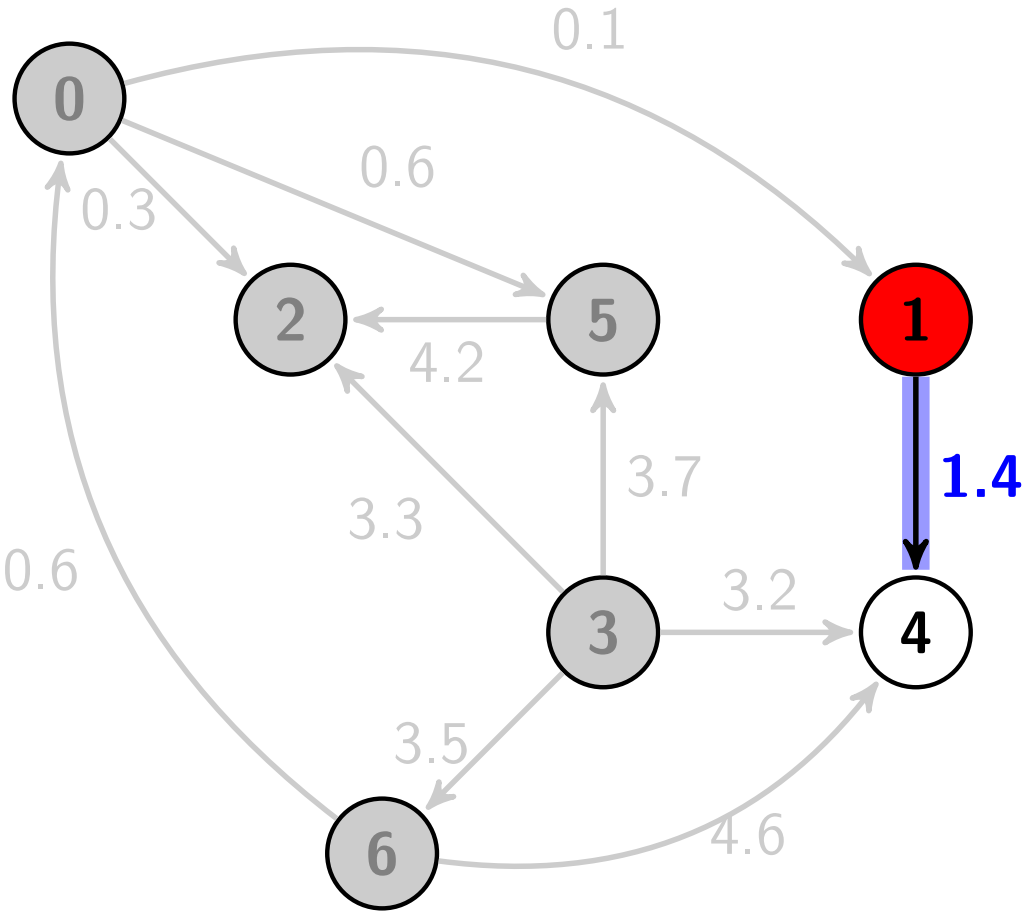
Ordem topológica - pilha

- $\{4, 1\}$

	distPara	arestaPara
0	0.6	6
v 1	0.7	0
2	0.9	0
3	inf	
w 4	$0.7 + 1.4$	1
5	1.2	0
6	0	

```
1 if (distPara[w]>distPara[v]+a.peso()){
2   distPara[w] = distPara[v] + a.peso();
3   arestaPara[w] = a;
4 }
```

- Expandir de 1 para 4.
- 4.6 > 0.7 + 1.4, atualizar tabela.



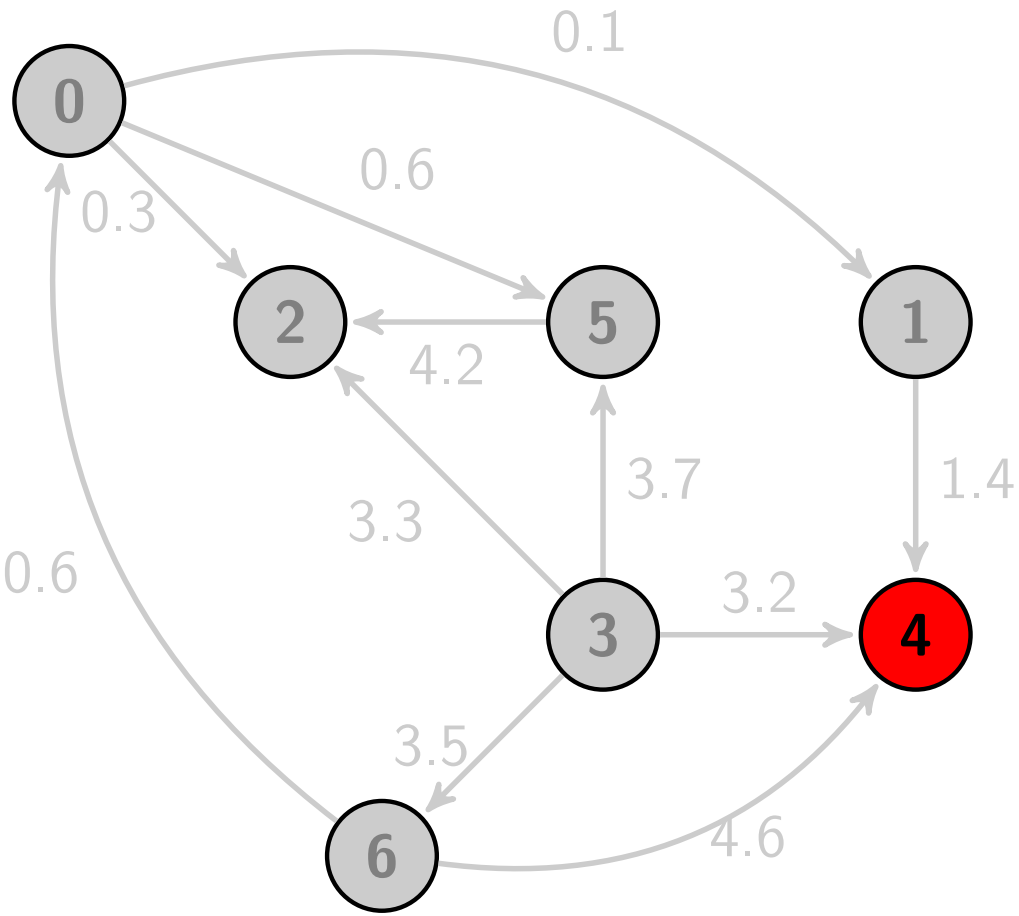
Ordem topológica - pilha

- $\{4, 1\}$

	distPara	arestaPara
0	0.6	6
v 1	0.7	0
2	0.9	0
3	inf	
w 4	2.1	**1**
5	1.2	0
6	0	

```
1 if (distPara[w]>distPara[v]+a.peso()){
2   distPara[w] = distPara[v] + a.peso();
3   arestaPara[w] = a;
4 }
```

- Expandir de **1** para **4**.
- `4.6 > 0.7 + 1.4`, atualizar tabela.

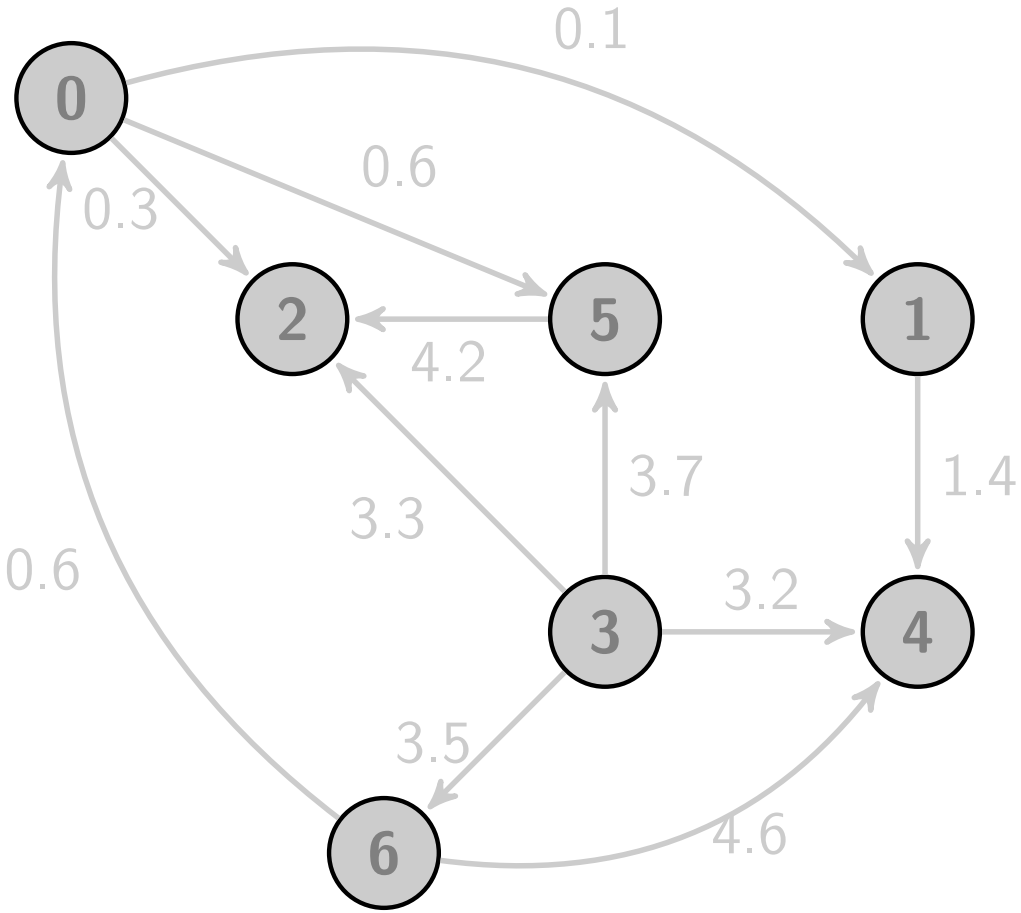


Ordem topológica - pilha

- {**4**}

	distPara	arestaPara
0	0.6	6
1	0.7	0
2	0.9	0
3	inf	
v 4	2.1	1
5	1.2	0
6	0	

- Pegar da pilha um vértice: **4**
- **Não** tem para onde expandir



Ordem topológica - pilha

- {}

	distPara	arestaPara
0	0.6	6
1	0.7	0
2	0.9	0
3	inf	
4	2.1	1
5	1.2	0
6	0	

- Pilha está vazia. Fim do algoritmo.
- Menores caminhos a partir de **6** estão prontos para serem consultados.

Aplicações - Ordenação topológica

- Detecção de ciclos em grafos
- Verificar se sequência de atividades é possível de respeitar
- Paralelização de algoritmos
- Linguagem de programação: herança cíclica em Java
- Recálculo de planilhas de dados

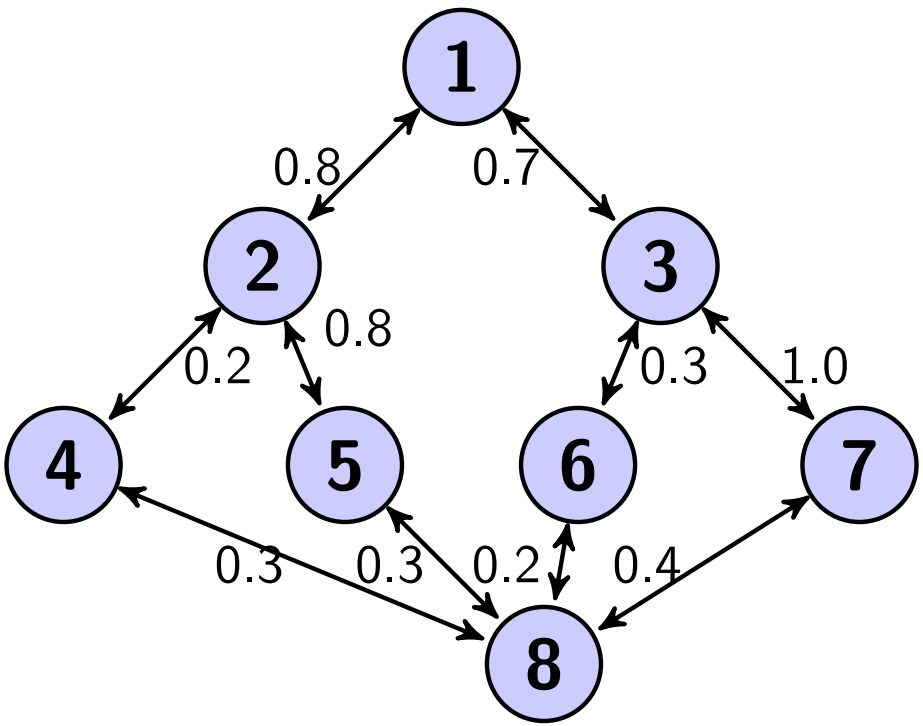
Aplicações - Menor caminho

- Roteamento em mapas (GPS)
- Gerência de projetos - caminho crítico
- Redimensionamento de imagens
- Planejamento de tráfego urbano
- Protocolos de rede (OSPF, BGP)
- Comércio de moedas estrangeiras

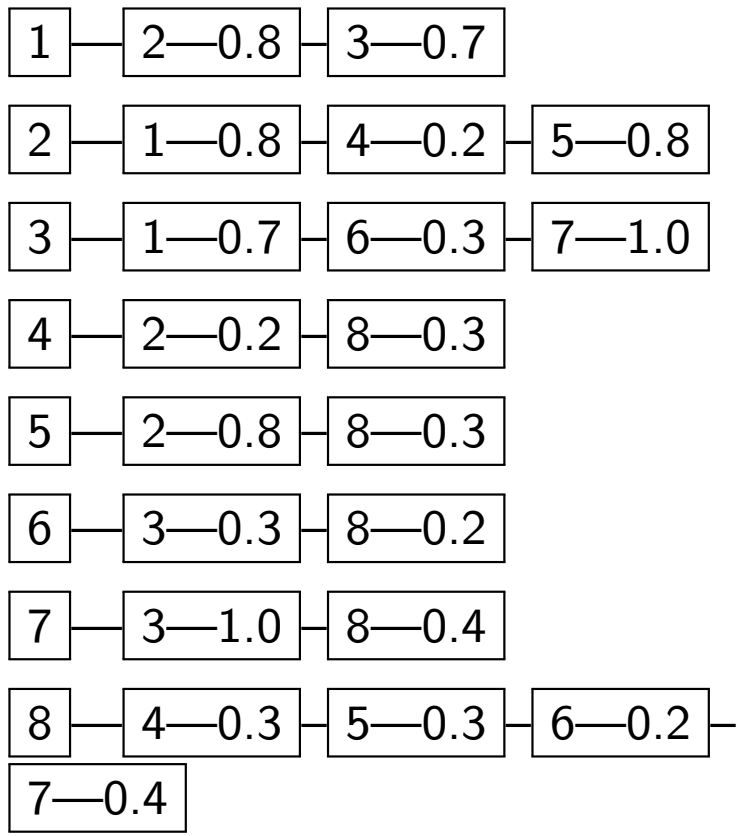
Outros algoritmos relacionados

- Ordenação topológica: dígrafos acíclicos direcionados
- Algoritmo de Dijkstra: dígrafos com arestas não-negativas
- Algoritmo de Floyd-Warshall (encontra todos os pares de menores caminhos)

Exemplo de dígrafo cíclico com pesos >= 0



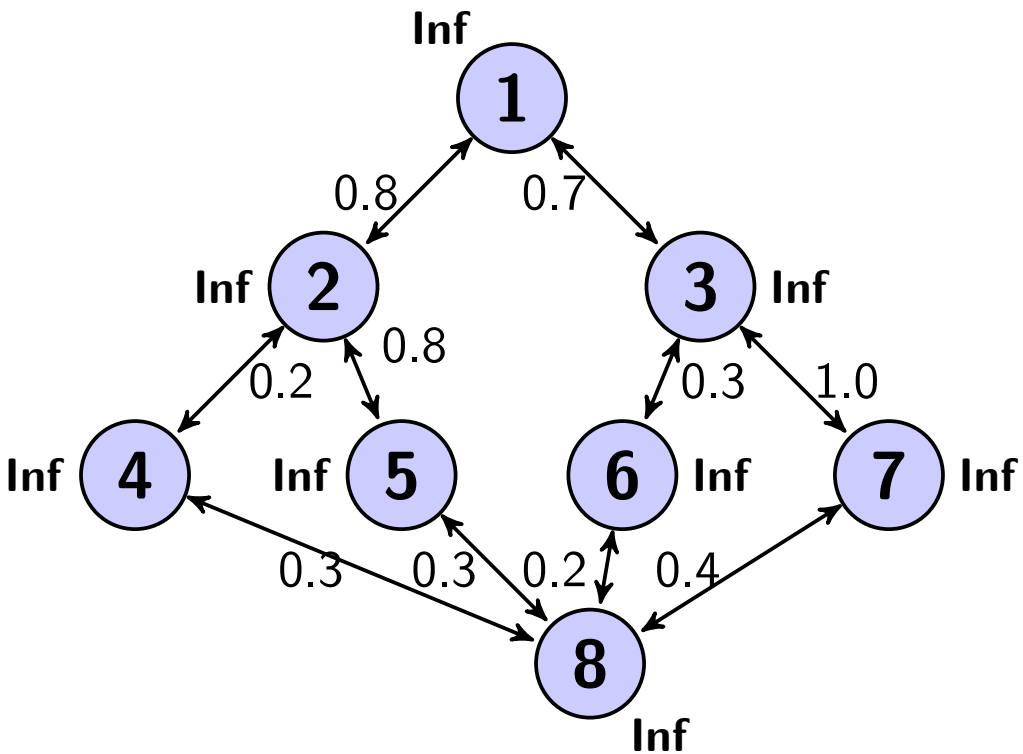
Listas de adjacência



Menor caminho por Edsger Dijkstra

Condições

- arestas com pesos não-negativos
- cada vértice guarda um valor de distância
- distâncias são registradas nos vértices, em relação ao vértice de origem
- distâncias começam infinitas



Algoritmo Djikstra: pré-requisitos

- Tipo Abstrato de Dados: Fila de Prioridades
  - void add(int u, double value)
  - void update(int i, double d)
  - int removeMin()
  - double getValue(int i)

Algoritmo Djikstra: inicialização

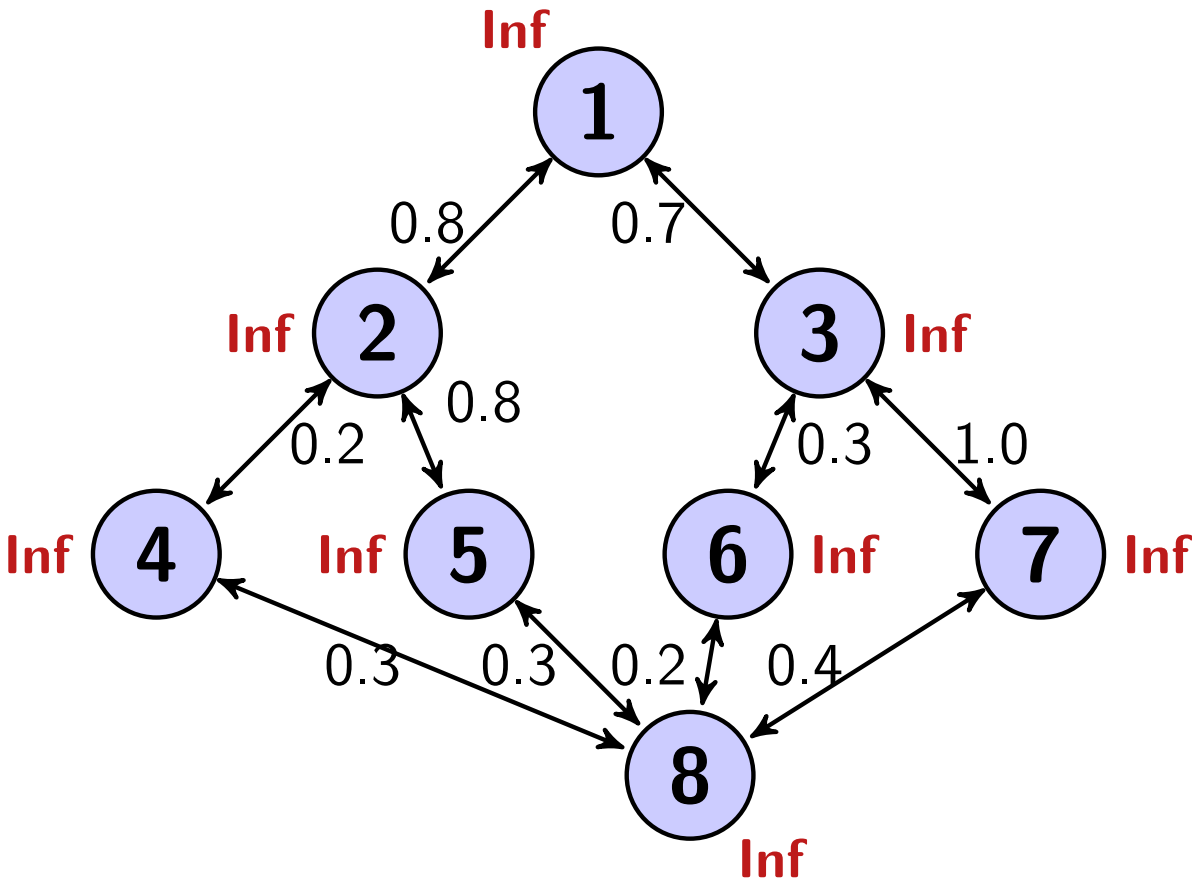
```
1  void iniciaDijkstra(int origem) {
2    filaP = new FilaPrioridades<>();
3    for (int u = 0; u < G.V(); u++)
4     filaP.add(u, Double.MAX_VALUE);
5
6    Double[] precedente = new Double[g.V()];
7    for (int u = 0; u < G.V(); u++) {  // inicialização
8      precedente[u] = null; //vertice precedente
9    }
10   filaP.update(origem, 0);//dist. à origem
11 }
```

Algoritmo Menor Caminho Dijkstra

```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     double altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

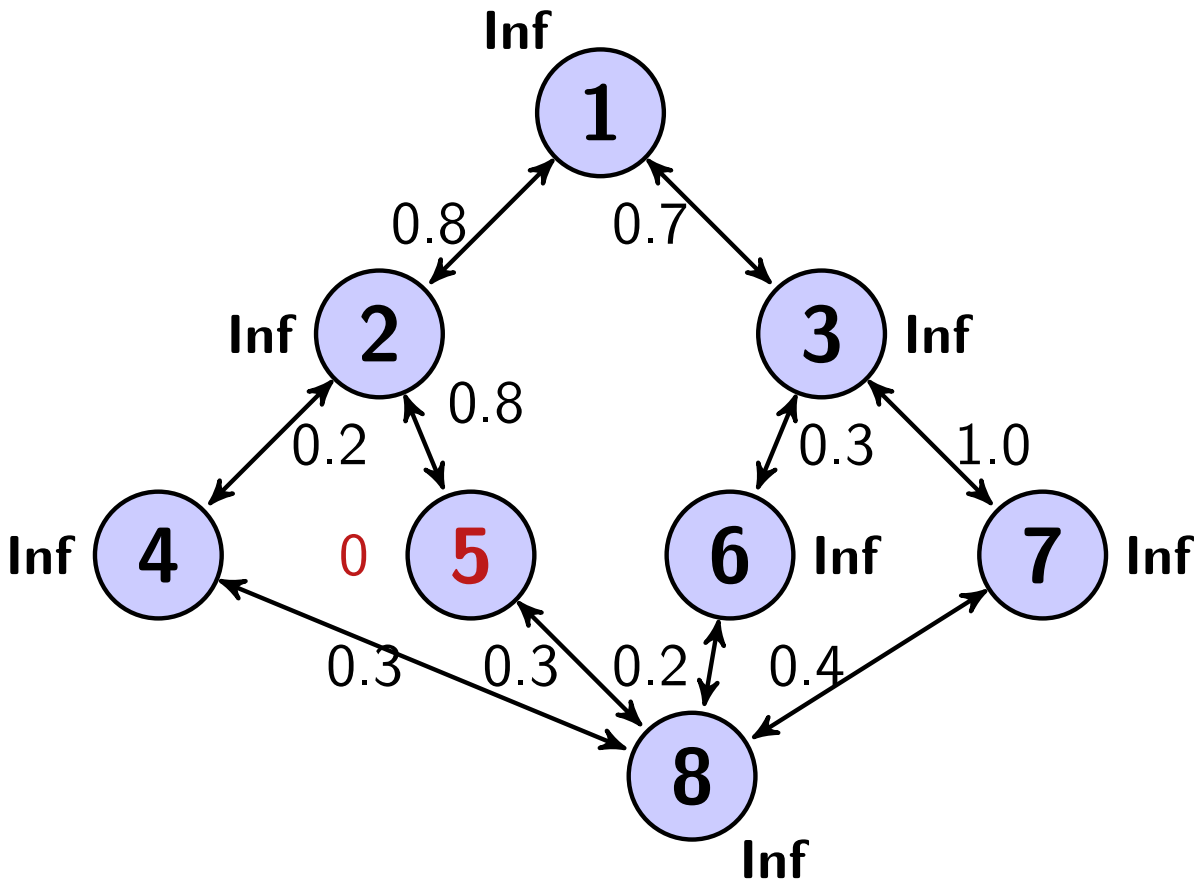
Menor caminho por Edsger Dijkstra

Inicialização: distâncias começam com Infinito

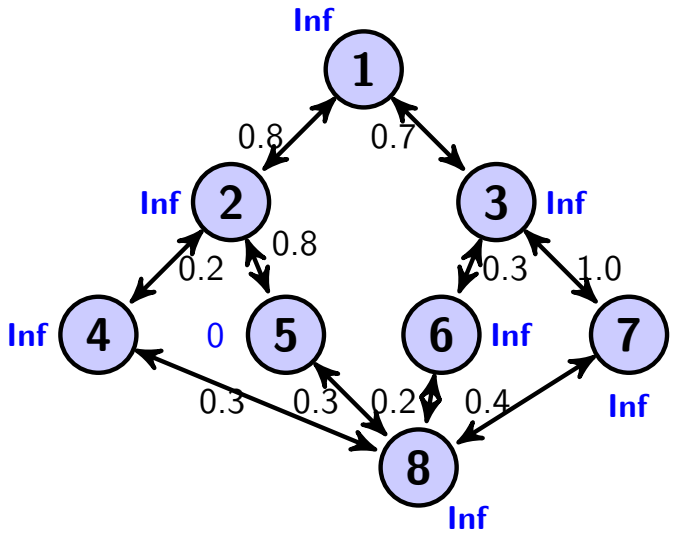


Menor caminho por Edsger Dijkstra

Inicialização: distância da origem é zero. Vértice de origem é o 5.



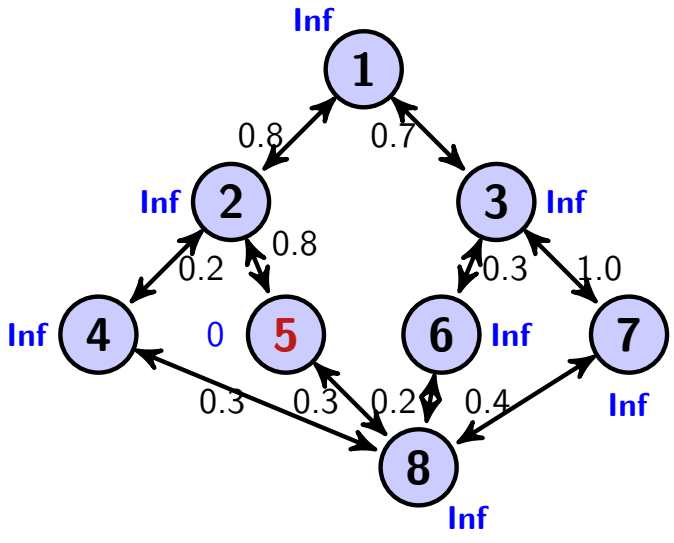
Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 5/0 | 1/Inf | 2/Inf | 3/Inf | 4/Inf | 6/Inf | 7/Inf | 8/Inf |

Execução.



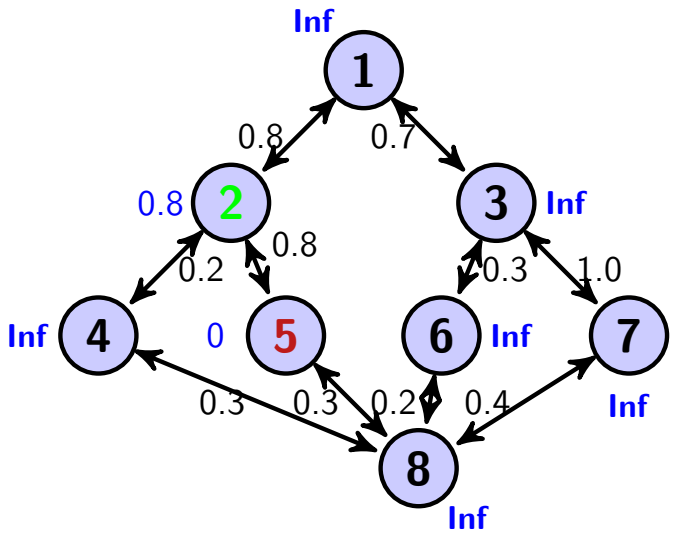

```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 1/Inf | 2/Inf | 3/Inf | 4/Inf | 6/Inf | 7/Inf | 8/Inf |

u = | 5 |

G.adj(u) = | (5, 2, 0.8) | (5, 8, 0.3) |

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 2/0.8 | 1/Inf | 3/Inf | 4/Inf | 6/Inf | 7/Inf | 8/Inf |

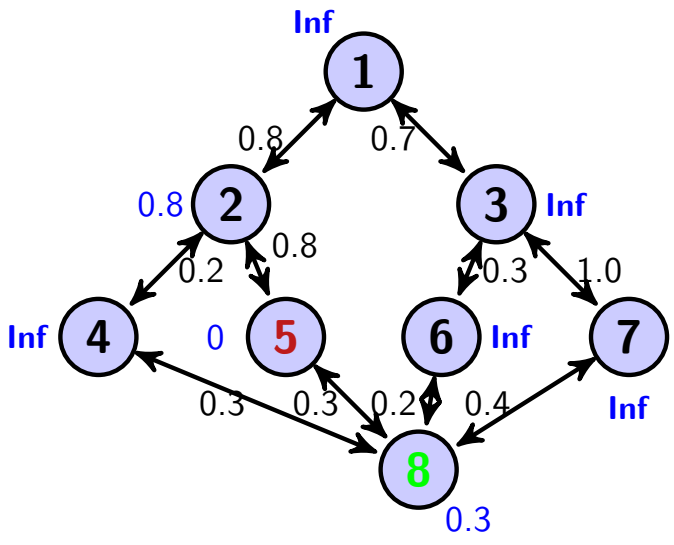
u = | 5 |

G.adj(u) = | (5,2, 0.8) | (5, 8, 0.3) |

a = | (5, 2, 0.8) |

Atualiza a distância ao vértice 2.

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 8/0.3 | 2/0.8 | 1/Inf | 3/Inf | 4/Inf | 6/Inf | 7/Inf |

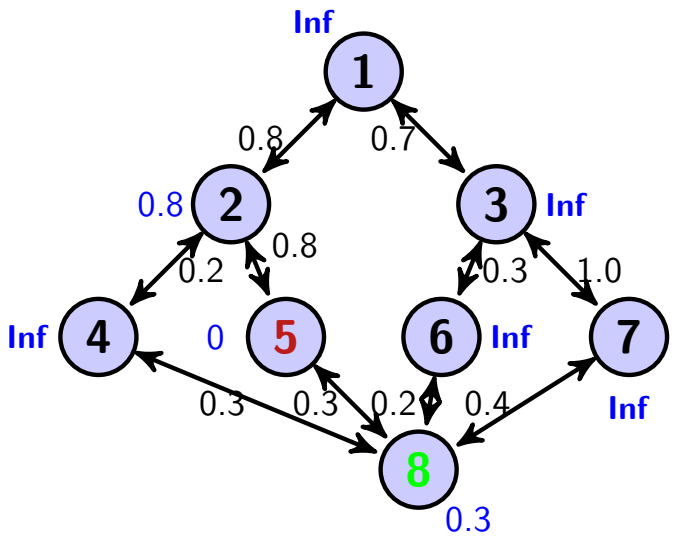
u = | 5 |

G.adj(u) = | (5, 2, 0.8) | (5, 8, 0.3) |

a = | (5, 8, 0.3) |

Atualiza distância ao vértice 8.

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 8/0.3 | 2/0.8 | 1/Inf | 3/Inf | 4/Inf | 6/Inf | 7/Inf |

u = | 5 |

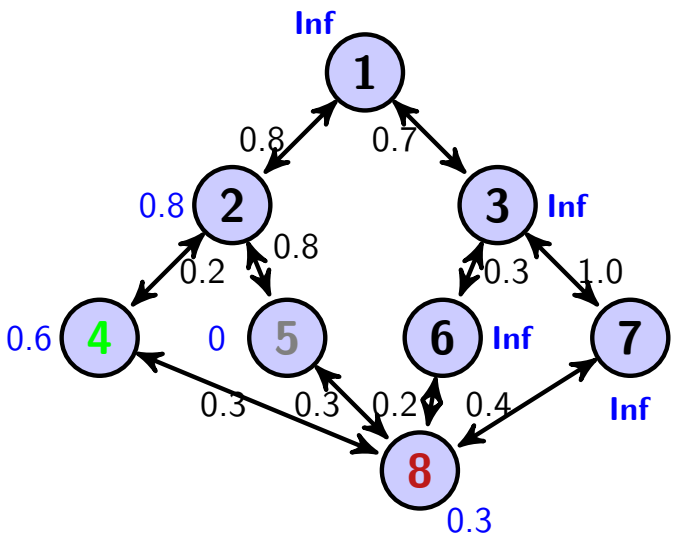
G.adj(u) = | (5, 2, 0.8) | (5, 8, 0.3) |

a = | (5, 8, 0.3) |

Atualiza distância ao vértice 8.

Os adjacentes de | 5 | acabaram, então procurar próximo a visitar.

Execução.

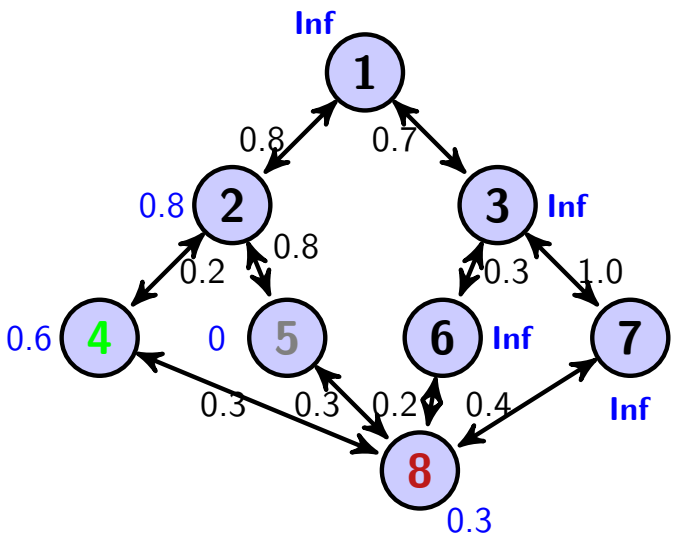



```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 2/0.8 | 1/Inf | 3/Inf | 4/Inf | 6/Inf | 7/Inf |

u = 8, pois tem a menor distância até o momento

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 2/0.8 | 1/Inf | 3/Inf | 4/Inf | 6/Inf | 7/Inf |

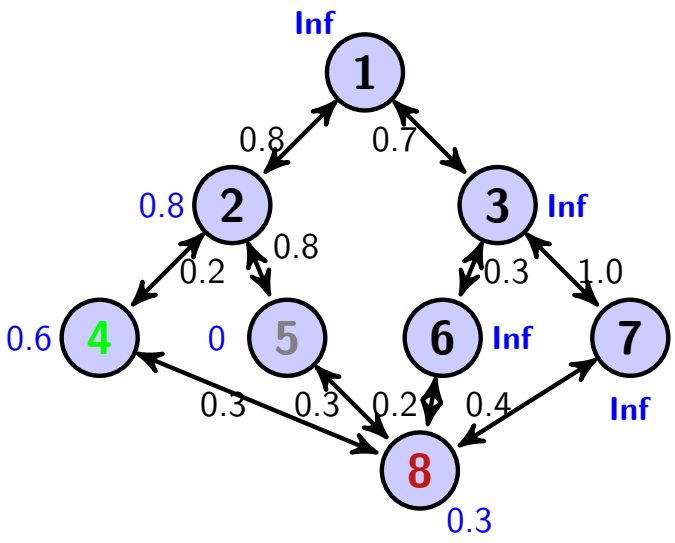
u = | 8 |, pois tem a menor distância até o momento

G.adj(u) = | (8, 4, 0.3) | (8, 6, 0.2) | (8, 7, 0.4) |

a = | (8, 4, 0.3) |

Atualiza distância do vértice 4 para $0.3 + 0.3 = 0.6$.

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 4/0.6 | 2/0.8 | 1/Inf | 3/Inf | 6/Inf | 7/Inf |

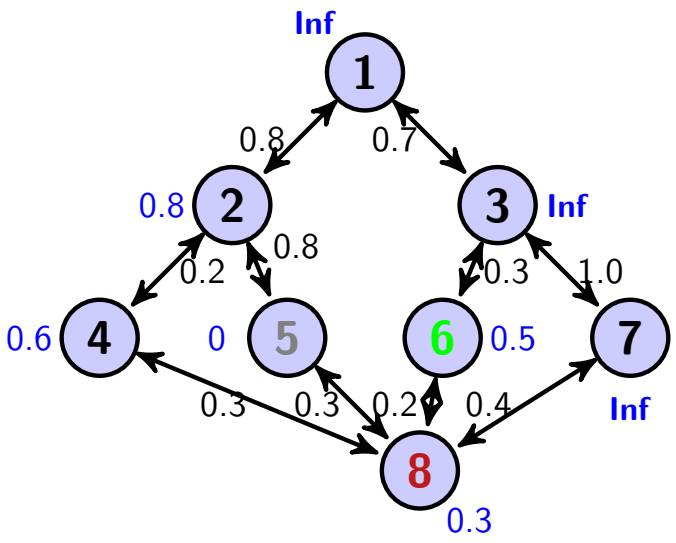
u = | 8 |, pois tem a menor distância até o momento

G.adj(u) = | (8, 4, 0.3) | (8, 6, 0.2) | (8, 7, 0.4) |

a = | (8, 4, 0.3) |

Atualiza distância do vértice 4 para $0.3 + 0.3 = 0.6$.

Execução.

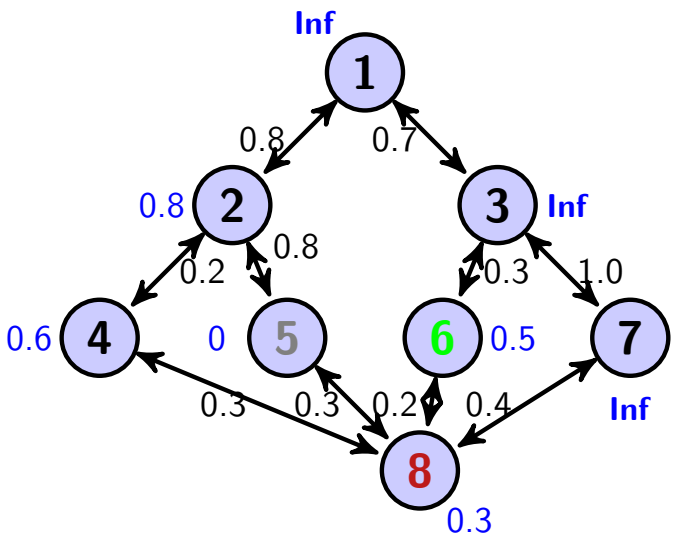



```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 4/0.6 | 2/0.8 | 1/Inf | 3/Inf | 6/Inf | 7/Inf |

u = 8, continua atualizando distâncias aos adjacentes

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

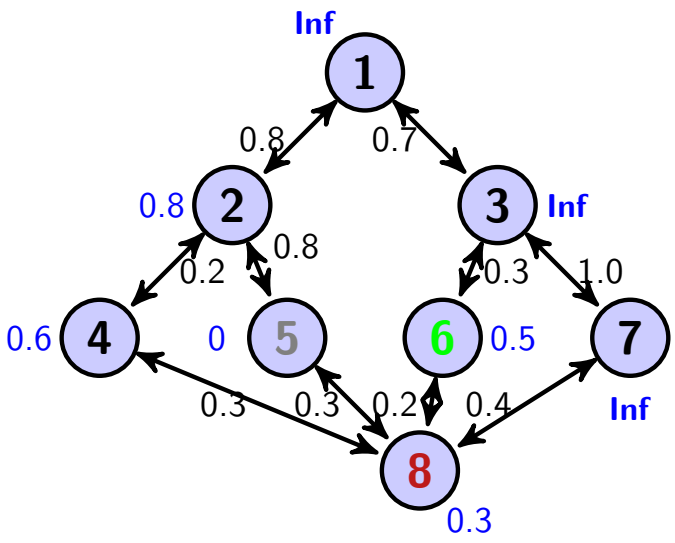
filaP = | 4/0.6 | 2/0.8 | 1/Inf | 3/Inf | 6/Inf | 7/Inf |

u = | 8 |, continua atualizando distâncias aos adjacentes

G.adj(u) = | (8, 4, 0.3) | (8, 6, 0.2) | (8, 7, 0.4) |

a = | (8, 6, 0.2) |

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 6/0.5 | 4/0.6 | 2/0.8 | 1/Inf | 3/Inf | 7/Inf |

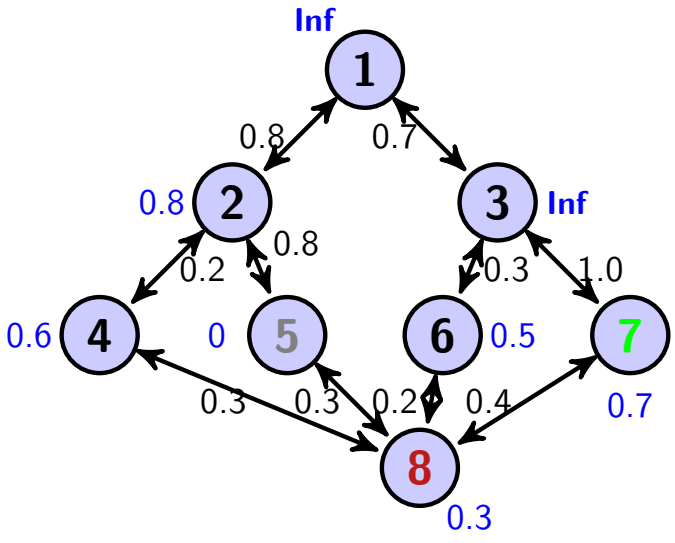
u = | 8 |, continua atualizando distâncias aos adjacentes

G.adj(u) = | (8, 4, 0.3) | (8, 6, 0.2) | (8, 7, 0.4) |

a = | (8, 6, 0.2) |

Atualiza distância do vértice 6 para 0.3 + 0.2 = 0.5

Execução.

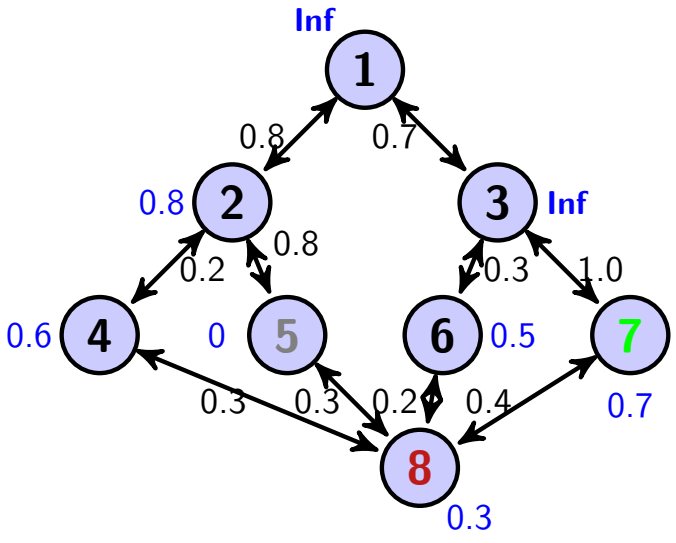



```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 6/0.5 | 4/0.6 | 2/0.8 | 1/Inf | 3/Inf | 7/Inf |

u = 8, continua atualizando distâncias aos adjacentes

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

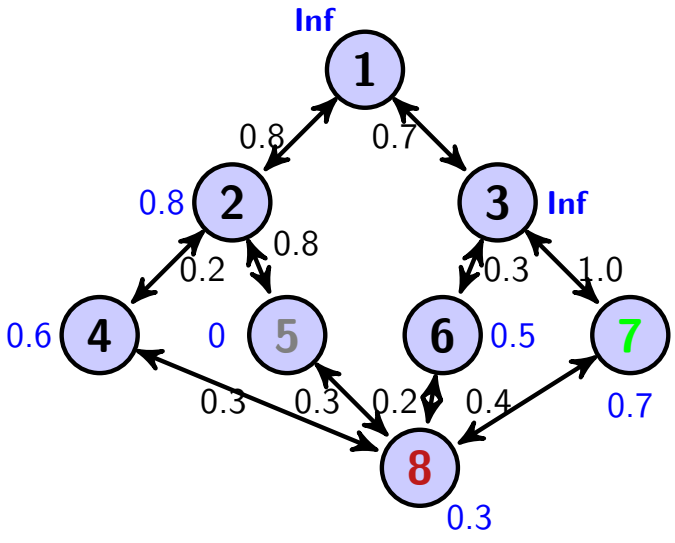
filaP = | 6/0.5 | 4/0.6 | 2/0.8 | 1/Inf | 3/Inf | 7/Inf |

u = | 8 |, continua atualizando distâncias aos adjacentes

G.adj(u) = | (8, 4, 0.3) | (8, 6, 0.2) | (8, 7, 0.4) |

a = | (8, 7, 0.4) |

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 6/0.5 | 4/0.6 | 7/0.7 | 2/0.8 | 1/Inf | 3/Inf |

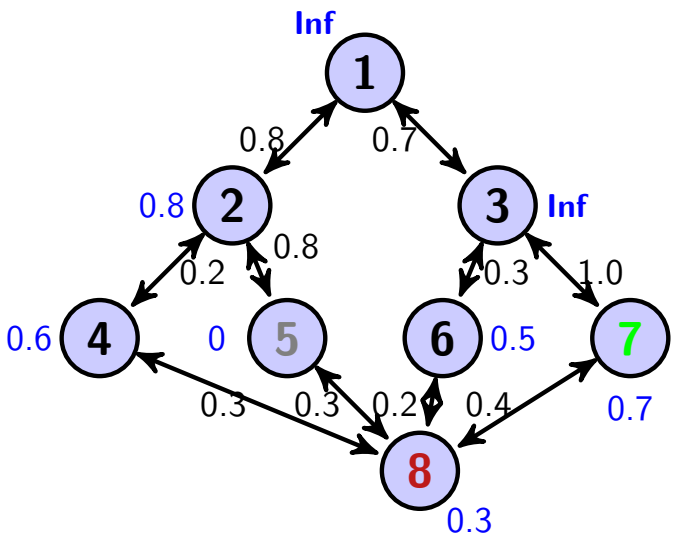
u = | 8 |, continua atualizando distâncias aos adjacentes

G.adj(u) = | (8, 4, 0.3) | (8, 6, 0.2) | (8, 7, 0.4) |

a = | (8, 7, 0.4) |

Atualiza distância do vértice 7 para $0.3 + 0.4 = 0.7$

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 6/0.5 | 4/0.6 | 7/0.7 | 2/0.8 | 1/Inf | 3/Inf |

u = | 8 |, continua atualizando distâncias aos adjacentes

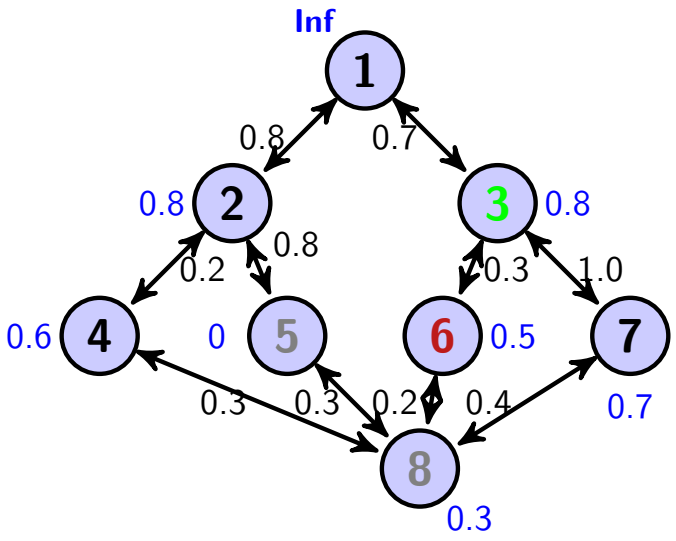
G.adj(u) = | (8, 4, 0.3) | (8, 6, 0.2) | (8, 7, 0.4) |

a = | (8, 7, 0.4) |

Atualiza distância do vértice 7 para $0.3 + 0.4 = 0.7$

Os adjacentes a | 8 | acabaram, então procurar próximo a visitar.

Execução.

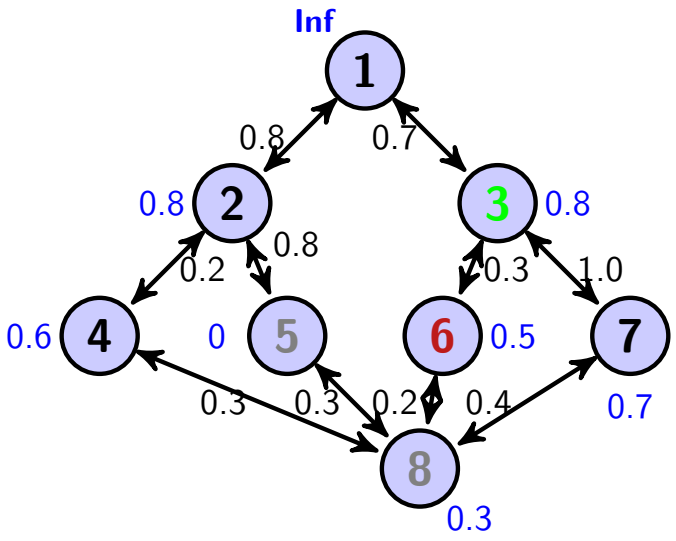



```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 4/0.6 | 7/0.7 | 2/0.8 | 1/Inf | 3/Inf |

u = | 6/0.5 |, remove novo vértice mínimo

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

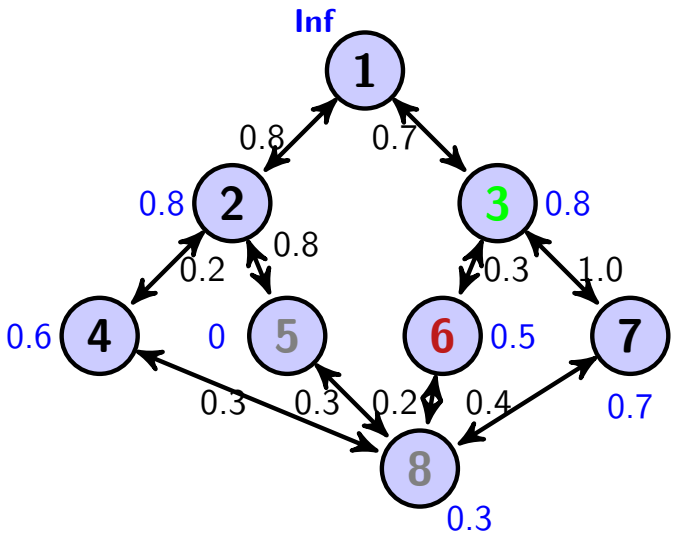
filaP = | 4/0.6 | 7/0.7 | 2/0.8 | 1/Inf | 3/Inf |

u = | 6/0.5 |, remove novo vértice mínimo

G.adj(u) = | (6, 3, 0.3) | (6, 8, 0.2) |

a = | (6, 3, 0.3) |

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 4/0.6 | 7/0.7 | 2/0.8 | 3/0.8 | 1/Inf |

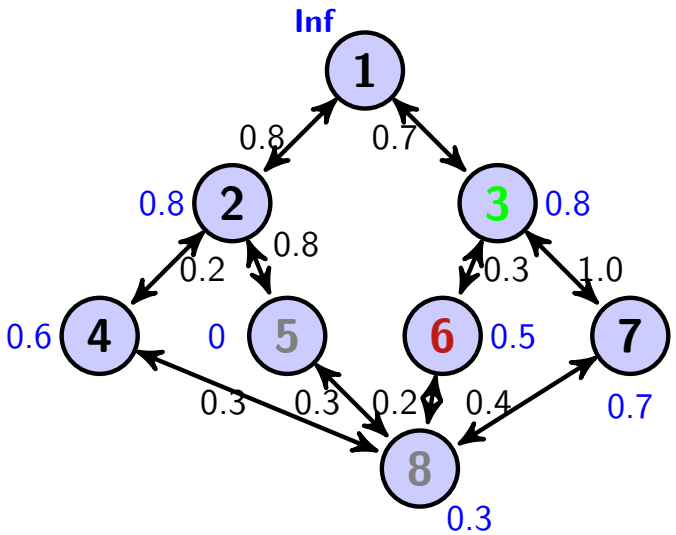
u = | 6/0.5 |, remove novo vértice mínimo

G.adj(u) = | (6, 3, 0.3) | (6, 8, 0.2) |

a = | (6, 3, 0.3) |

Atualiza distância do vértice 3 para 0.5 + 0.3 = 0.8

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 4/0.6 | 7/0.7 | 2/0.8 | 3/0.8 | 1/Inf |

u = | 6/0.5 |, remove novo vértice mínimo

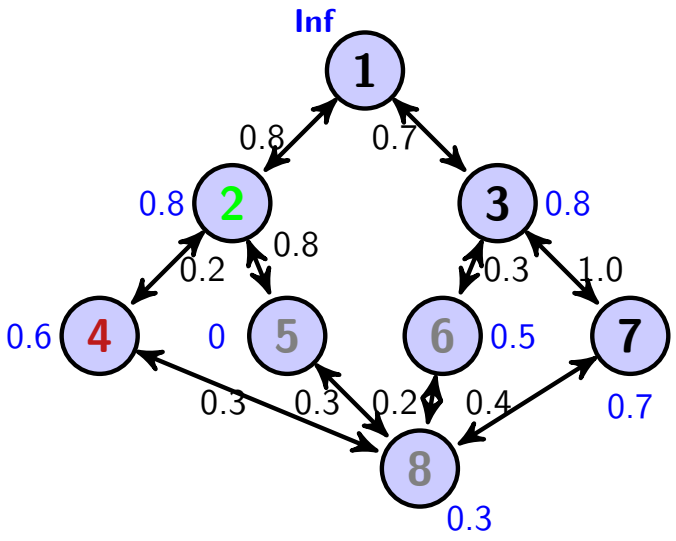
G.adj(u) = | (6, 3, 0.3) | (6, 8, 0.2) |

a = | (6, 3, 0.3) |

Atualiza distância do vértice 3 para 0.5 + 0.3 = 0.8

Os adjacentes acabaram, então procurar próximo a visitar.

Execução.

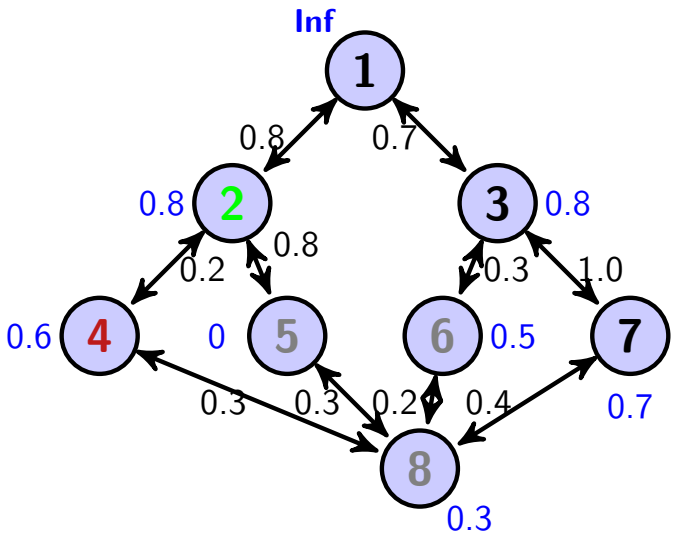



```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 7/0.7 | 2/0.8 | 3/0.8 | 1/Inf |

u = | 4/0.6 |, remove novo vértice mínimo

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

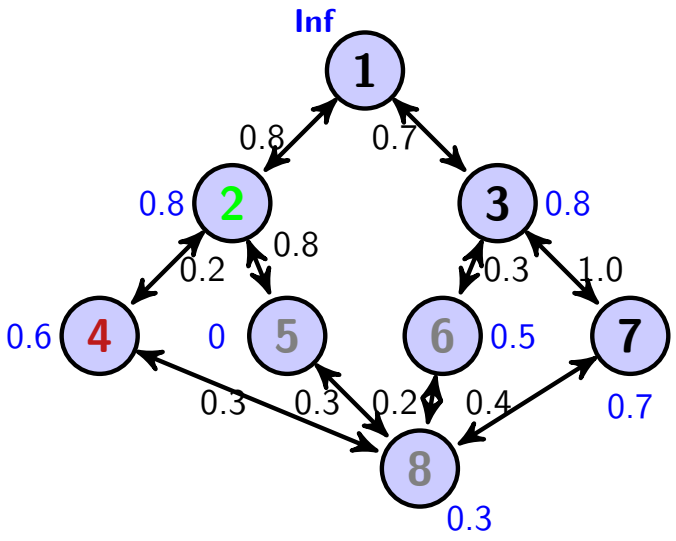
filaP = | 7/0.7 | 2/0.8 | 3/0.8 | 1/Inf |

u = | 4/0.6 |, remove novo vértice mínimo

G.adj(u) = | (4, 2, 0.2) | (4, 8, 0.3) |

a = | (4, 2, 0.2) |

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 7/0.7 | 2/0.8 | 3/0.8 | 1/Inf |

u = | 4/0.6 |, remove novo vértice mínimo

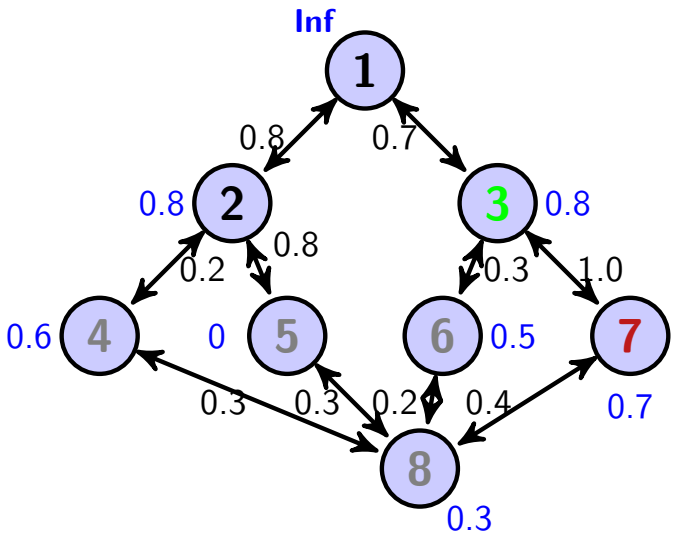
G.adj(u) = | (4, 2, 0.2) | (4, 8, 0.3) |

a = | (4, 2, 0.2) |

Não atualiza distância do vértice 2 pois alternativa não é menor.

Os adjacentes acabaram, então procurar próximo a visitar.

Execução.

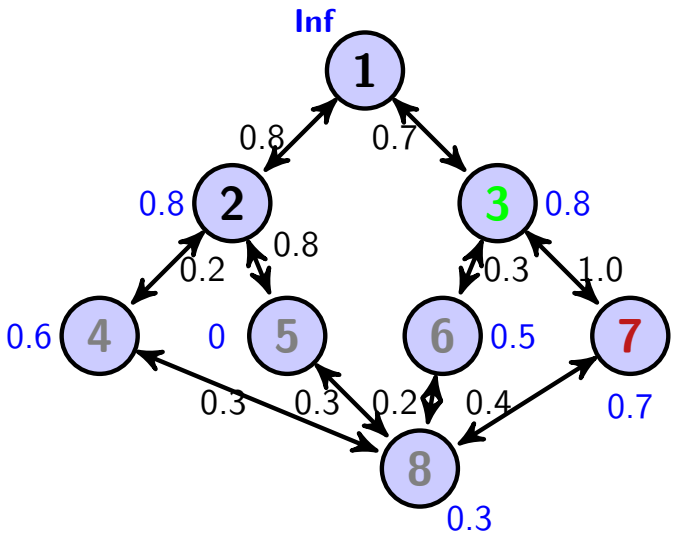



```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 2/0.8 | 3/0.8 | 1/Inf |

u = | 7/0.7 |, remove novo vértice mínimo

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

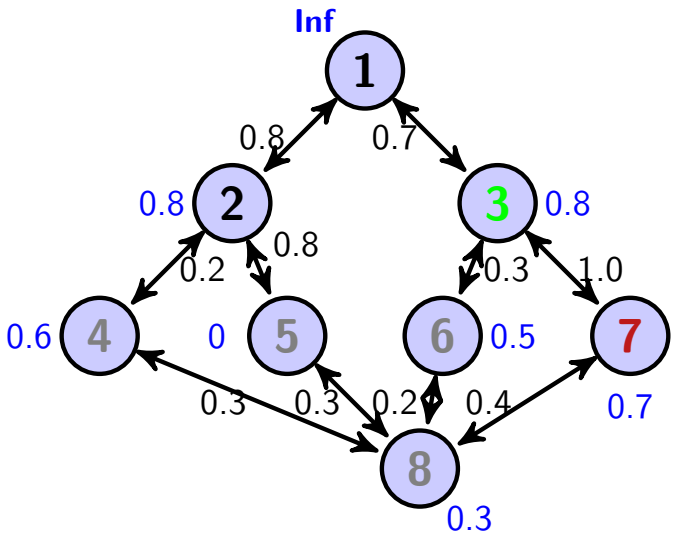
filaP = | 2/0.8 | 3/0.8 | 1/Inf |

u = | 7/0.7 |, remove novo vértice mínimo

G.adj(u) = | (7, 3, 1.0) | (7, 8, 0.4) |

a = | (7, 3, 1.0) |

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 2/0.8 | 3/0.8 | 1/Inf |

u = | 7/0.7 |, remove novo vértice mínimo

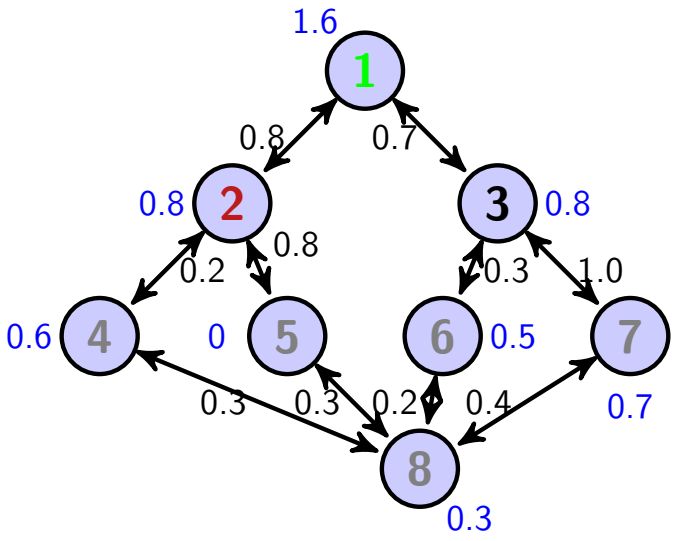
G.adj(u) = | (7, 3, 1.0) | (7, 8, 0.4) |

a = | (7, 3, 1.0) |

Não atualiza distância do vértice 3 pois alternativa não é menor.

Os adjacentes acabaram, então procurar próximo a visitar.

Execução.

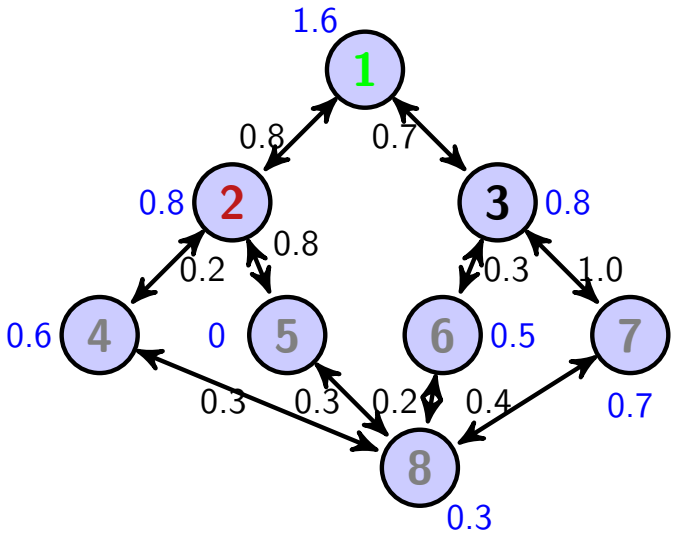



```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = | 3/0.8 | 1/Inf |

u = | 2/0.8 |, remove novo vértice mínimo

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

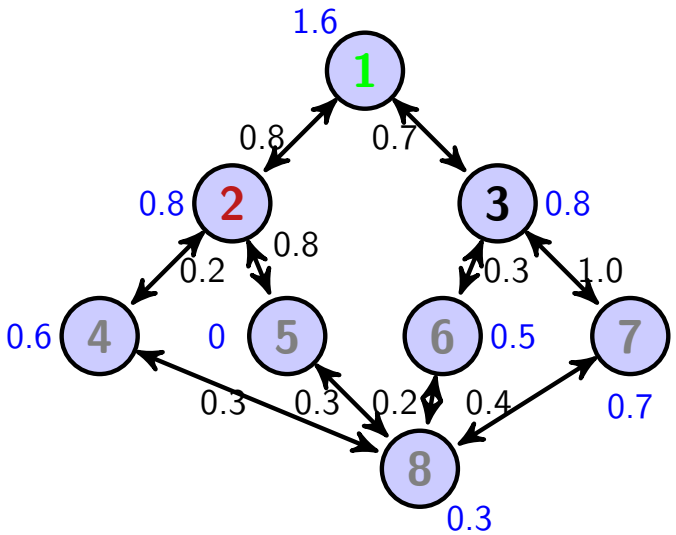
filaP = | 3/0.8 | 1/Inf |

u = | 2/0.8 |, remove novo vértice mínimo

G.adj(u) = | (2, 1, 0.8) | (2, 4, 0.2) | (2, 5, 0.4) |

a = | (2, 1, 0.8) |

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = $\boxed{3/0.8}\boxed{1/1.6}$

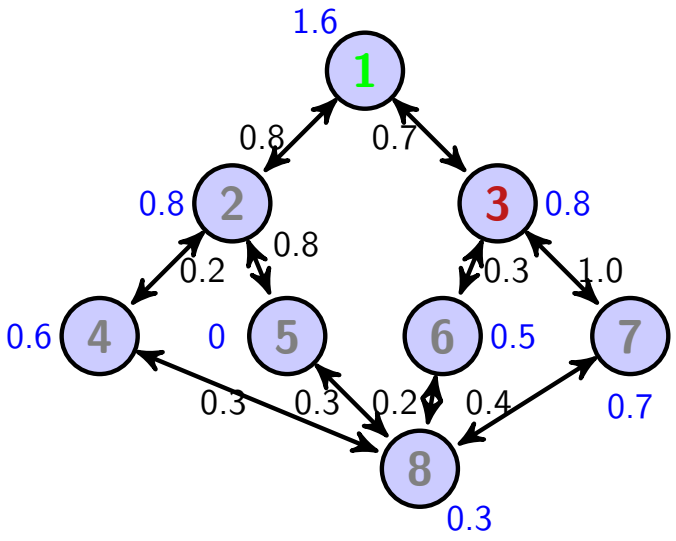
u = $\boxed{2/0.8}$, remove novo vértice mínimo

G.adj(u) = $\boxed{(2, 1, 0.8)}\boxed{(2, 4, 0.2)}\boxed{(2, 5, 0.4)}$

a = $\boxed{(2, 1, 0.8)}$

Atualiza distância do vértice 1 para 1.6. Os adjacentes acabaram, então procurar próximo a visitar.

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = $\boxed{1}$

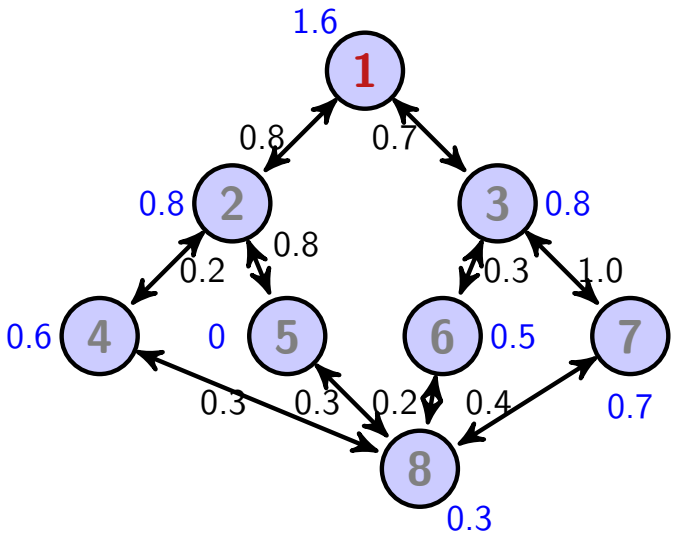
u = $\boxed{3}$, pois tem a menor distância até o momento 0.8

adj(u) = $\boxed{1}$

a.para() = $\boxed{1}$

Atualiza distância do vértice 1 de 1.6 para 1.5.

Execução.




```
1  void dijkstra(int origem) {
2   iniciaDijkstra(origem);// inicialização
3
4   while (!filaP.empty()) {
5    int u = filaP.removeMin();
6    if (Double.isInfinite(filaP.getValue(u)))
7      break; // esta inacessivel
8
9    for (Aresta a: G.adj(u)) {
10     altDist = filaP.getValue(u) + a.peso();
11     if (altDist < filaP.getValue(a.para())) {
12       filaP.update(a.para(), altDist);
13       precedente[a.para()] = u;
14     }
15   }
16  }
17 }
```

filaP = $\boxed{1}$

u = $\boxed{3}$, pois tem a menor distância até o momento 0.8

adj(u) = $\square$

Terminou o cálculo das distâncias.

Menor caminho

- usar variável `precedente`

```
1  Iterable<Integer> menorCaminho(int origem, int destino)
       {
2    dijkstra(origem);
3    Stack<Integer> caminho = new Stack<>();
4
5    int verticeAtual = destino;
6    while (verticeAtual != origem) {
7      caminho.add(verticeAtual);
8      verticeAtual = antecedente[verticeAtual];
9    }
10   caminho.add(origem);
11
12   return(caminho);
13 }
```

Complexidades das operações

- menorCaminho: proporcional ao comprimento do caminho
- dijkstra: depende da estrutura que implementa a FilaPrioridades
  - Se for uma heap (tipo de árvore binária): $O(E \log V)$
  - Se for uma Fibonacci heap: $O(E + V \log V)$ mas possui custos constantes altos demais