Teoria dos Grafos

Aula 9

Caminho mínimo

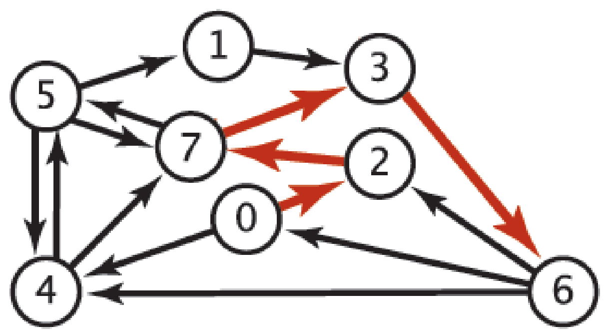
Marcos Muniz

Caminho mínimo

- Dado um digrafo, encontre o menor caminho (direcionado) de s até t.

edge-weighted digraph

4->5	0.35
5->4	0.35
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93

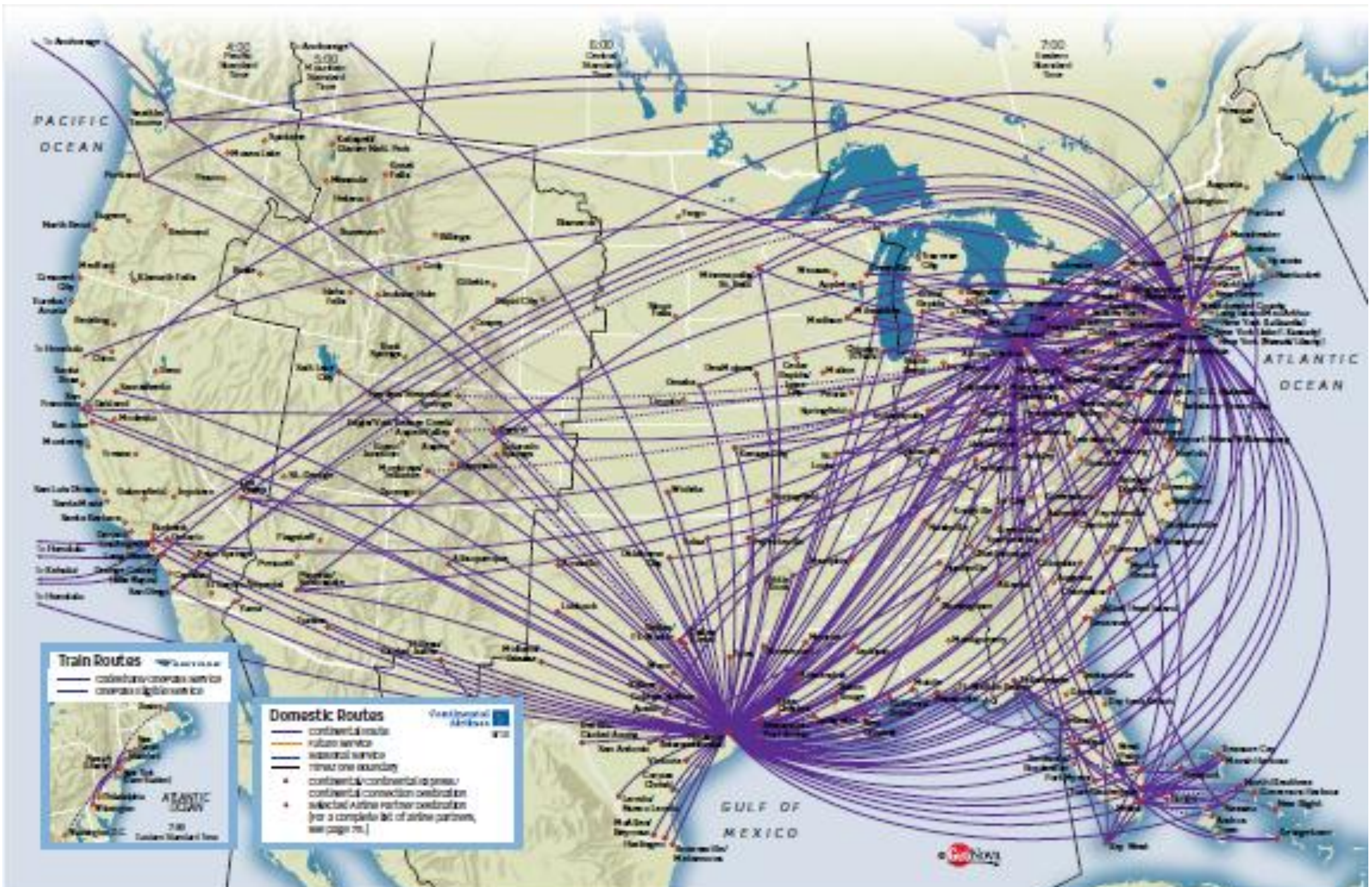


shortest path from 0 to 6

0->2	0.26
2->7	0.34
7->3	0.39
3->6	0.52

Caminho mínimo

Caminho mínimo



http://www.continental.com/web/en-US/content/travel/routes

Caminho mínimo

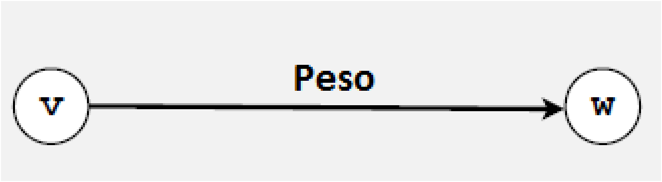
- Aplicações:
  - Pert/CPM;
  - Roteamento em mapas;
  - Navegação de robôs;
  - Planejamento de tráfego urbano;
  - Agendamento de tarefas;
  - Roteamento de mensagens em telecomunicações;
  - Roteamento de protocolos de rede;
  - Entre outras...

Caminho mínimo

- Variações do problema:
  - Quais vértices?
    - Origem-destino: De um vértice para outro;
    - Fonte única: De um vértice para todos os demais;
    - Todos os pares: Entre todos os pares de vértices.
  - Restrições nos pesos das arestas?
    - Pesos não negativos;
    - Pesos arbitrários;
    - Pesos euclidianos.
  - Ciclos?
    - Sem ciclos direcionados;
    - Sem ciclos negativos.

API de Digrafo ponderado

`public class DirectedEdge`		
	`DirectedEdge(int v, int w, double weight)`	Aresta ponderada de v para w
`int`	`from()`	Vértice V
`int`	`to()`	Vértice W
`double`	`weight()`	Peso da aresta
`String`	`toString()`	Representação textual



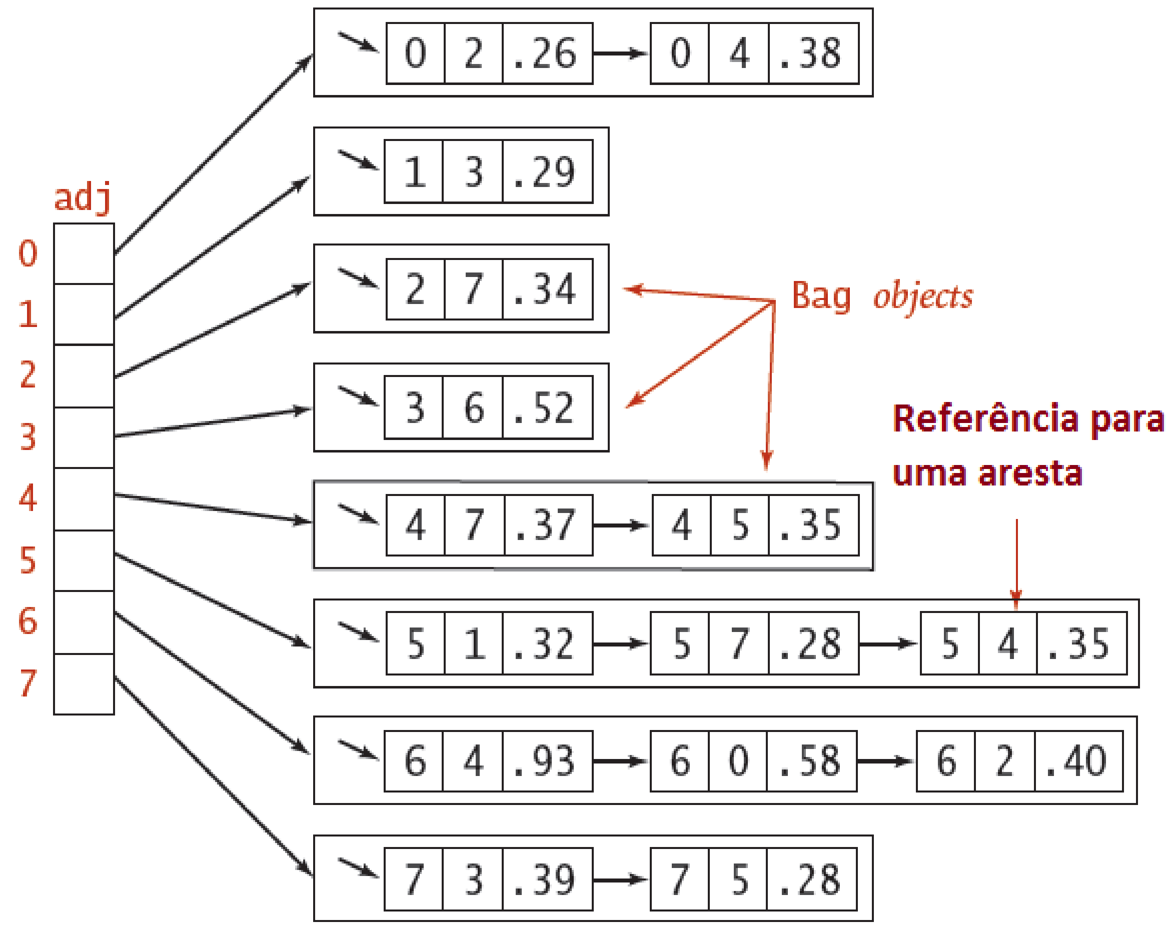
API de Digrafo ponderado

```
public class DirectedEdge
{
   private final int v, w;
   private final double weight;

   public DirectedEdge(int v, int w, double weight)
   {
      this.v = v;
      this.w = w;
      this.weight = weight;
   }

   public int from()
   {  return v;  }

   public int to()
   {  return w;  }

   public int weight()
   {  return weight; }
}
```

API de Digrafo ponderado

tinyEWD.txt

```
V  8
   15  E
   4 5  0.35
   5 4  0.35
   4 7  0.37
   5 7  0.28
   7 5  0.28
   5 1  0.32
   0 4  0.38
   0 2  0.26
   7 3  0.39
   1 3  0.29
   2 7  0.34
   6 2  0.40
   3 6  0.52
   6 0  0.58
   6 4  0.93
```



API de Digrafo ponderado

```
public class EdgeWeightedDigraph
{
   private final int V;
   private final Bag<Edge>[] adj;

   public EdgeWeightedDigraph(int V)
   {
      this.V = V;
      adj = (Bag<DirectedEdge>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<DirectedEdge>();
   }

   public void addEdge(DirectedEdge e)
   {
      int v = e.from();
      adj[v].add(e);
   }

   public Iterable<DirectedEdge> adj(int v)
   {  return adj[v];  }
}
```

A MARCA DA EDUCAÇÃO

API de Digrafo ponderado

```
public EdgeWeightedDigraph(int V)
{
   this.V = V;
   adj = (Bag<DirectedEdge>[]) new Bag[V];
   for (int v = 0; v < V; v++)
      adj[v] = new Bag<DirectedEdge>();
}
public void addEdge(DirectedEdge e)
{
   int v = e.from();
   adj[v].add(e);
}
public Iterable<DirectedEdge> adj(int v)
{  return adj[v];  }
```

API de único Caminho

```
public class SP
```

	`SP(EdgeWeightedDigraph G, int s)`
`double`	`distTo(int v)`
`Iterable <DirectedEdge>`	`pathTo(int v)`
`boolean`	`hasPathTo(int v)`

API de único Caminho

```
SP sp = new SP(G, s);
for (int v = 0; v < G.V(); v++)
{
   StdOut.printf("%d to %d (%.2f): ", s, v, sp.distTo(v));
   for (DirectedEdge e : sp.pathTo(v))
      StdOut.print(e + "  ");
   StdOut.println();
}
```

uni-bh
A MARCA DA EDUCAÇÃO

API de único Caminho

```
% java SP tinyEWD.txt 0
0 to 0 (0.00):
0 to 1 (1.05): 0->4 0.38  4->5 0.35  5->1 0.32
0 to 2 (0.26): 0->2 0.26
0 to 3 (0.99): 0->2 0.26  2->7 0.34  7->3 0.39
0 to 4 (0.38): 0->4 0.38
0 to 5 (0.73): 0->4 0.38  4->5 0.35
0 to 6 (1.51): 0->2 0.26  2->7 0.34  7->3 0.39  3->6 0.52
0 to 7 (0.60): 0->2 0.26  2->7 0.34
```

Caminho mínimo

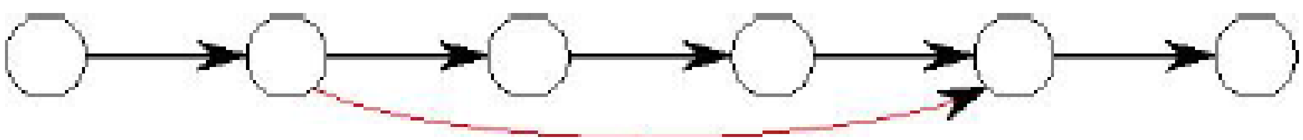
- O problema do caminho mínimo consiste em encontrar, caso exista, o menor caminho de um ponto de origem até um ponto de destino.
- O algoritmo de busca em profundidade encontrar o menor caminho,caso exista, em número de arestas.

Caminho mínimo

- Caminho mais curto $\delta(s, v)$:
  - Número mínimo de arestas em qualquer caminho de s para v, no caso de ser alcançável, ou ∞ se não for.
  - d[v] = $\delta(s, v)$, para todo $v \in V$.
- Caminho mais curto entre s e v.

Caminho mínimo

- **<u>Teorema:</u>** Sub-caminhos de caminhos mais curtos são caminhos mais curtos.
- **<u>Prova:</u>** Se algum sub-caminho não gerar o caminho mais curto, ele poderia ser substituído por um sub-caminho atalho e gerar um caminho total mais curto.

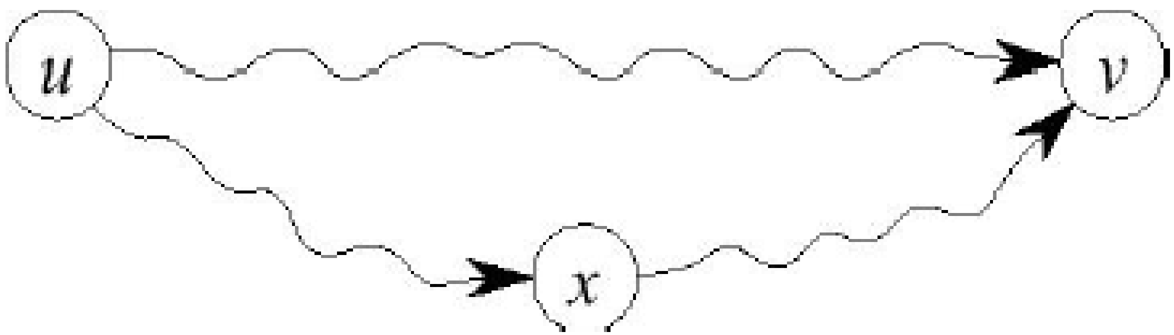


Caminho mínimo

- **Teorema da inequação triangular:**

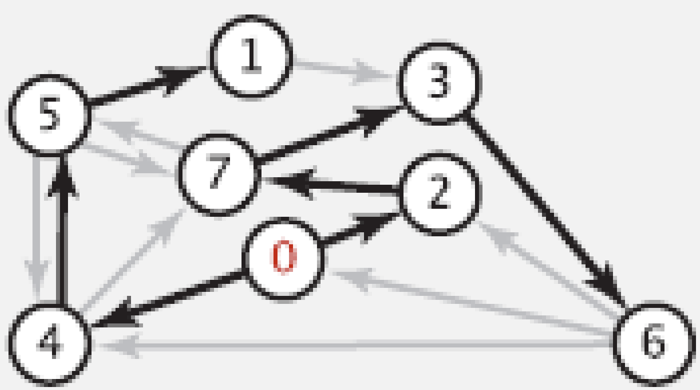
$$\delta(u, v) \leq \delta(u, x) + \delta(x, v)$$

  – Caminho mais curto $u \to v$ não é mais longo que qualquer outro caminho $u \to v$ - em particular, o caminho concatenando o caminho mais curto $u \to x$ com o caminho mais curto $x \to v$



Caminho mínimo

- distTo[v] é o caminho mínimo de s para v.
- edgeTo[v] é a última aresta no caminho mínimo de s para v.



	edgeTo[]		distTo[]
0	null		0
1	5->1	0.32	1.05
2	0->2	0.26	0.26
3	7->3	0.37	0.97
4	0->4	0.38	0.38
5	4->5	0.35	0.73
6	3->6	0.52	1.49
7	2->7	0.34	0.60

Caminho mínimo

```
public double distTo(int v)
{  return distTo[v];  }
```

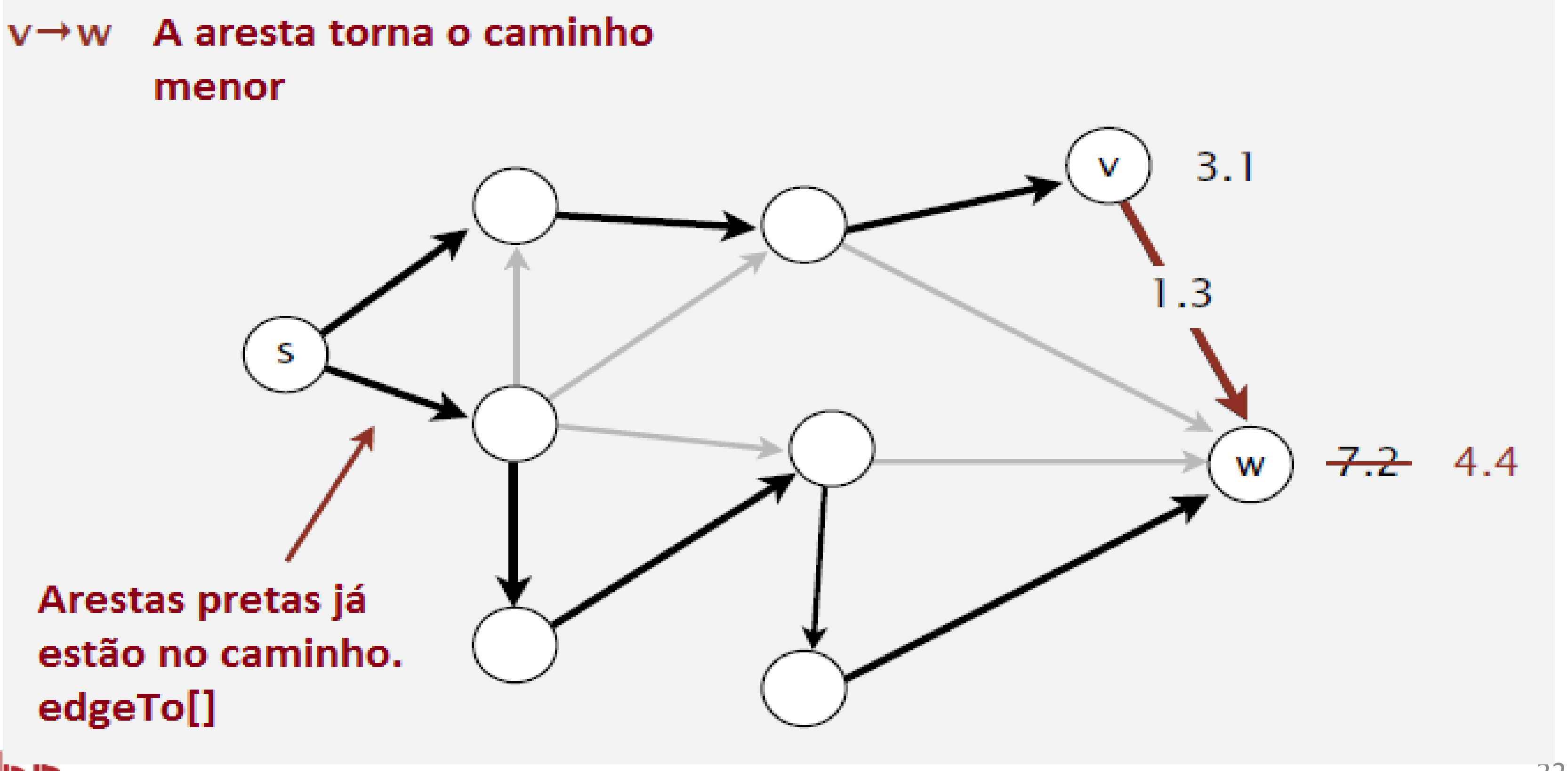
```
public Iterable<DirectedEdge> pathTo(int v)
{
   Stack<DirectedEdge> path = new Stack<DirectedEdge>();
   for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()])
      path.push(e);
   return path;
}
```

Caminho mínimo

- Relaxação de aresta:
  - distTo[v] é o tamanho do menor caminho **conhecido** de s para v.
  - distTo[w] é o tamanho do menor caminho **conhecido** de s para w.
  - edgeTo[w] é a última aresta no menor caminho **conhecido** de s para w.

Caminho mínimo

- Se $e = v \to w$ retornar o menor caminho para w através de v, atualize distTo[w] e edgeTo[w]



Caminho mínimo

- Se $e = v \rightarrow w$ retornar o menor caminho para w através de v, atualize distTo[w] e edgeTo[w]

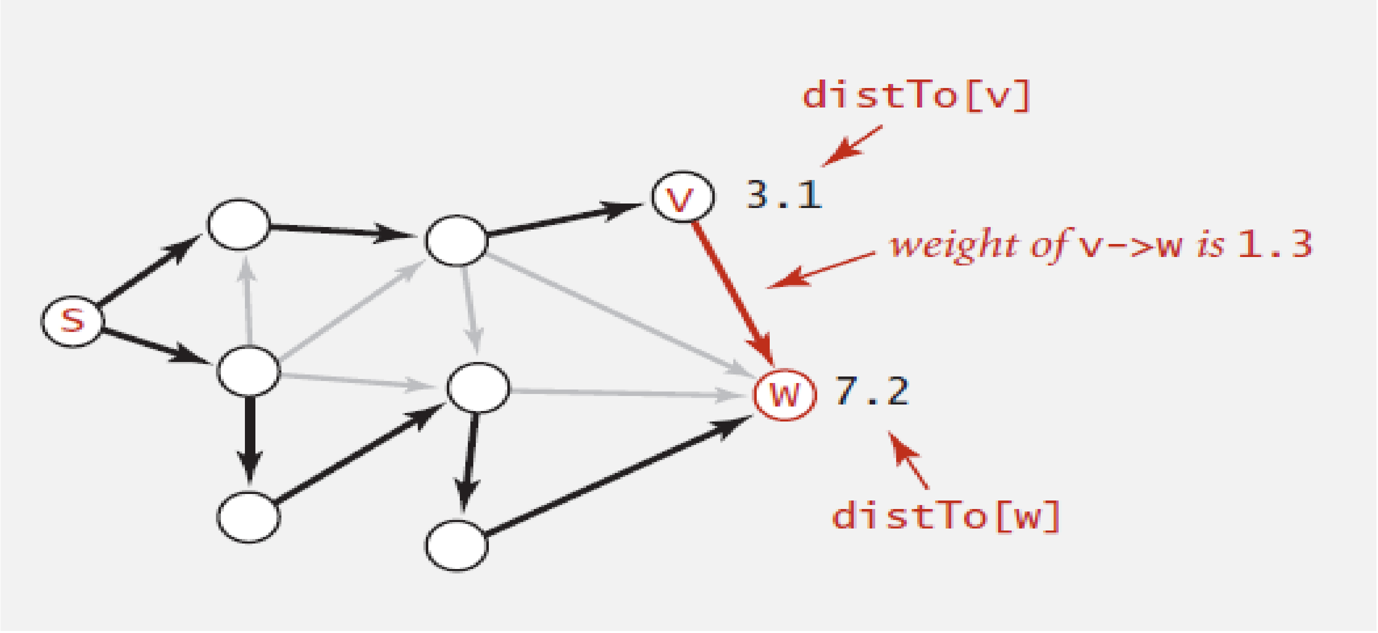
```
private void relax(DirectedEdge e)
{
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight())
    {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
    }
}
```

Caminho mínimo

- Proposição: Seja G um digrafo ponderado. Então distTo[] é o verto com o menor caminho a partir de s, se e somente se:
  - Para cada vértice v, distTo[v] é o tamanho de algum caminho de s para v.
  - Para cada aresta $e = v \to w$,
    $$distTo[w] \leq disTo[v] + e.weight()$$

Caminho mínimo

- Prova:
  - Suponha que $\text{distTo}[w] > \text{disTo}[v] + e.\text{weight}()$ para alguma aresta $e = v \rightarrow w$.
  - Então, e informa um caminho de s para w (através de v) de tamanho menor que $distTo[w]$



Caminho mínimo

- Prova:
  - Suponha que $s = v_0, v_1, v_2, \ldots, v_k = w$ é um caminho mínimo de s para w.
  - Então ,

$$distTo[V_k] \leq distTo[V_{k-1}] + e_k.weight()$$
$$distTo[V_{k-1}] \leq distTo[V_{k-2}] + e_{k-1}.weight()$$
$$\ldots$$
$$distTo[V_1] \leq distTo[V_0] + e_1.weight()$$

  - Adicione as inequações; Simplifique e substitua $distTo[V_0] = distTo[s] = 0$;

$$distTo[w] = distTo[V_k] \leq e_k.weight() + e_{k-1}.weight(\quad) + \ldots + e_1.weight(\quad)$$

  - Dessa maneira, $distTo[w]$ é o peso do menor caminho de w.

Caminho mínimo

- Algoritmo genérico para computar o caminho mínimo a partir de s:
  - Inicialize $distTo[s] = 0$ e $distTo[v] = \infty$ para todos os demais vértices.
  - Repita até que as condições de otimalidade sejam atendidas:
    - Relaxe alguma aresta.

Caminho mínimo

- **Proposição:** O algoritmo genérico calcula (se existe) o caminho mínimo para s.
- Prova:
  - Através do algoritmo, distTo[v] é o tamanho de um caminho simples de s para v(e edgeTo[v] é a última aresta no caminho).
  - Cada relaxação com sucesso decrementa distTo[v] para algum v.
  - distTo[v] pode ser decrementada no máximo um finito número de vezes.

Caminho mínimo

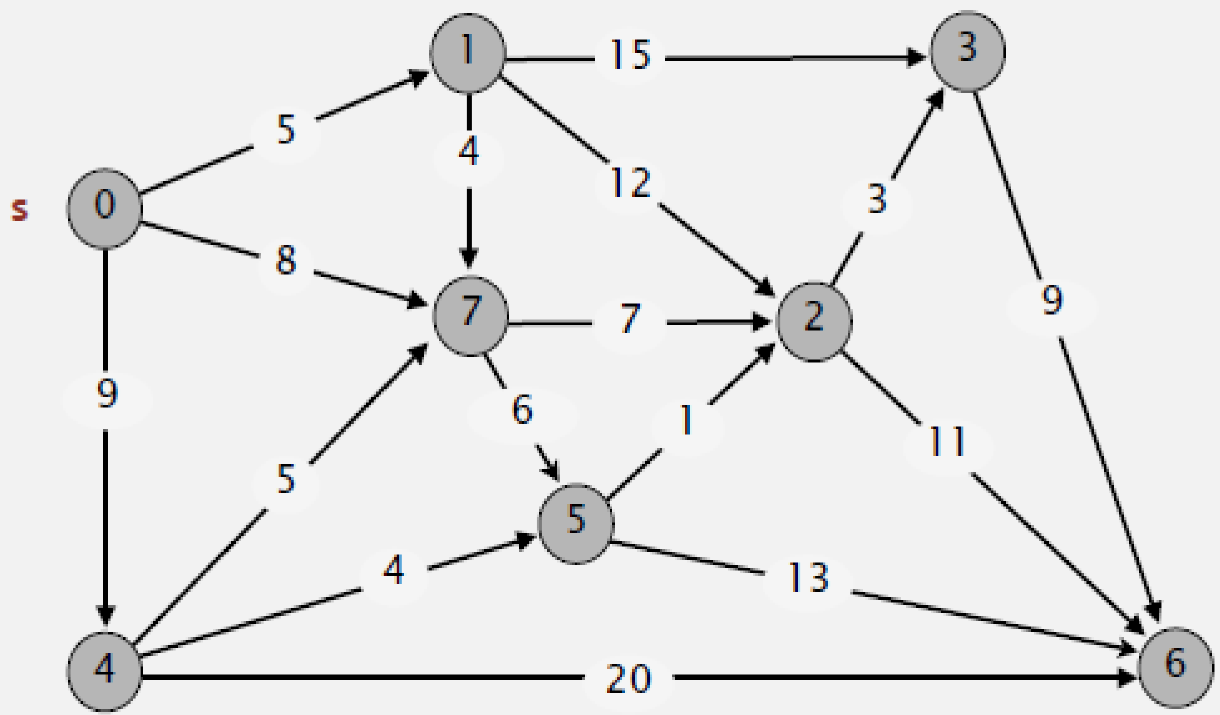
- Implementações eficientes: Como escolher qual aresta a relaxar?

1. Algoritmo de Dijkstra (pesos não negativos)
2. Algoritmo de ordenação topológica (Sem ciclos direcionados)
3. Algoritmo de Bellman-Ford (Sem ciclos negativos)

Algoritmo de Dijkstra

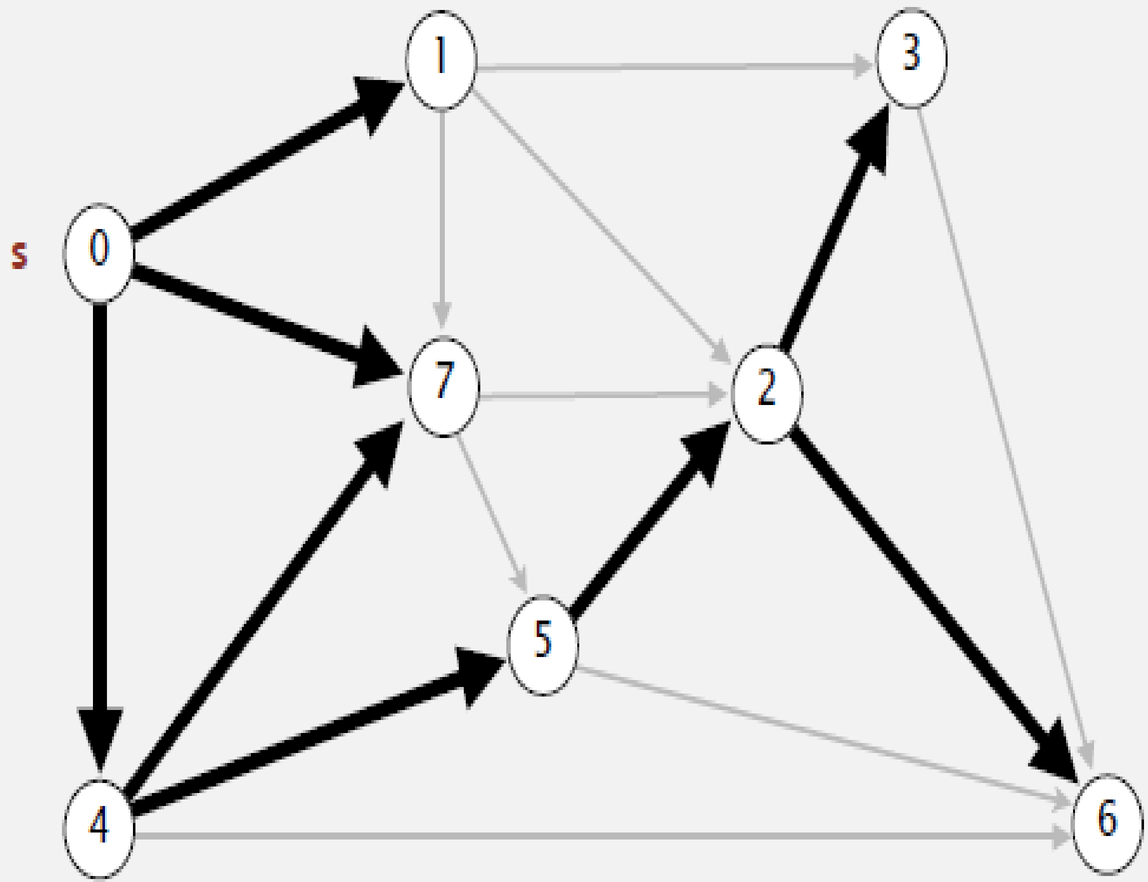
- Considere os vértices mais próximos de s. (Vértices que não estão na árvore com o menor distTo[])
- Adiciona o vértice na árvore e relaxe todas as arestas que saem daquele vértice.

Algoritmo de Dijkstra



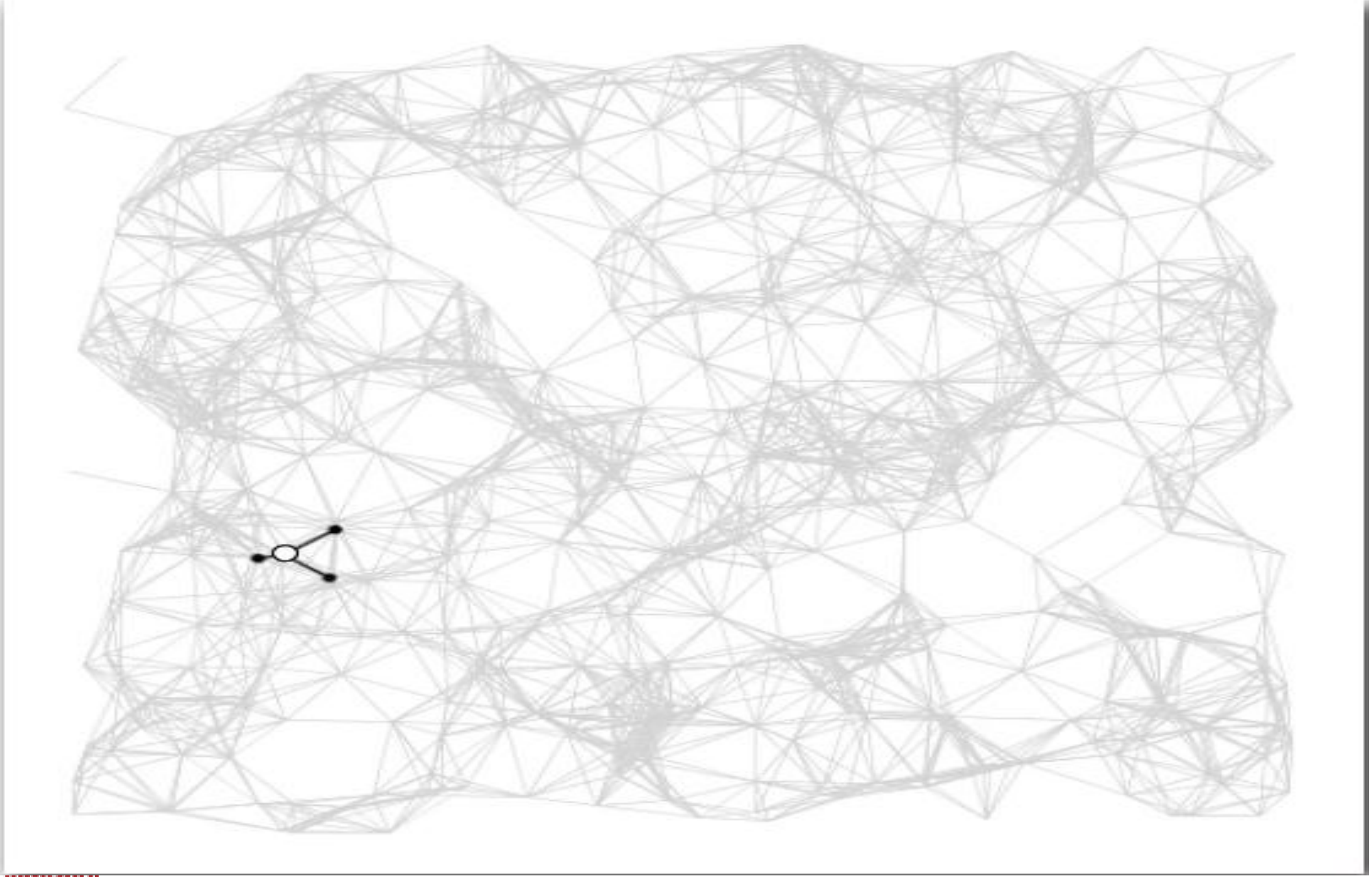
0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

Algoritmo de Dijkstra

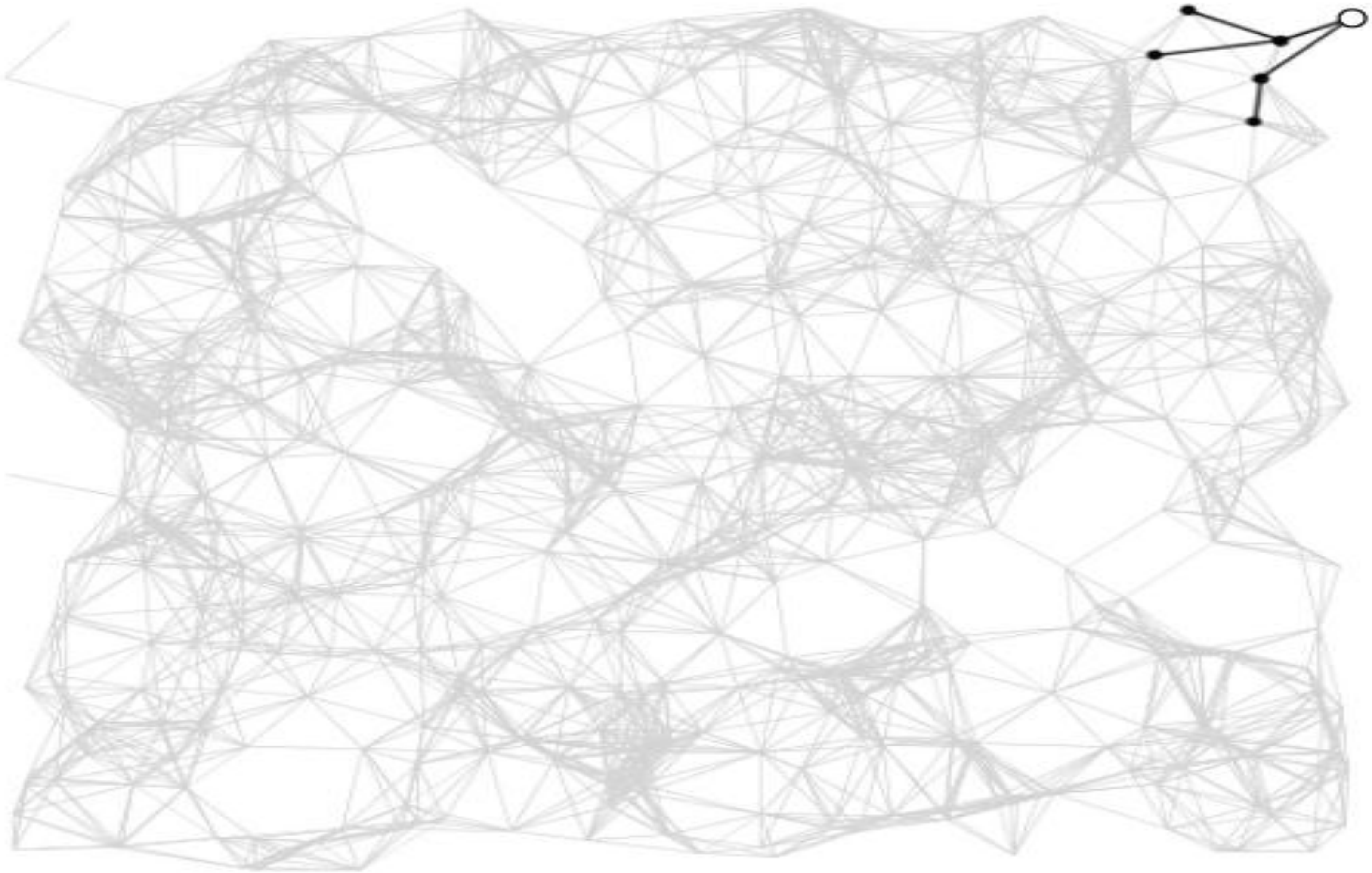


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

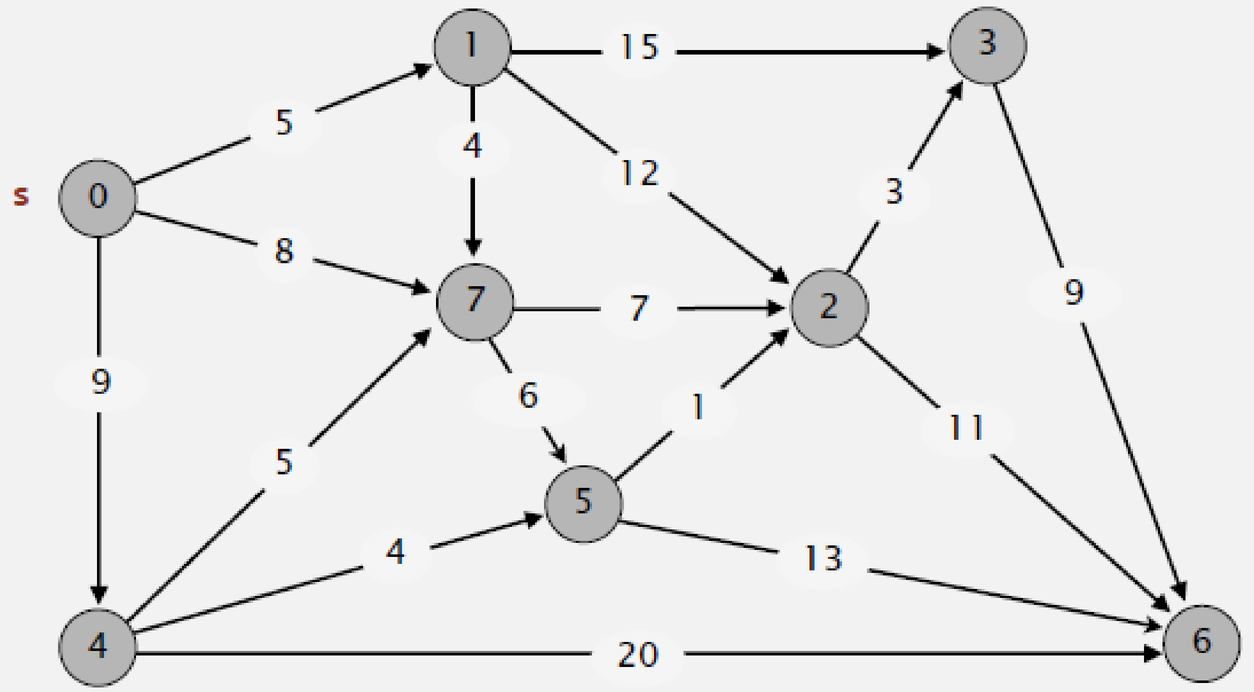
Algoritmo de Dijkstra



Algoritmo de Dijkstra

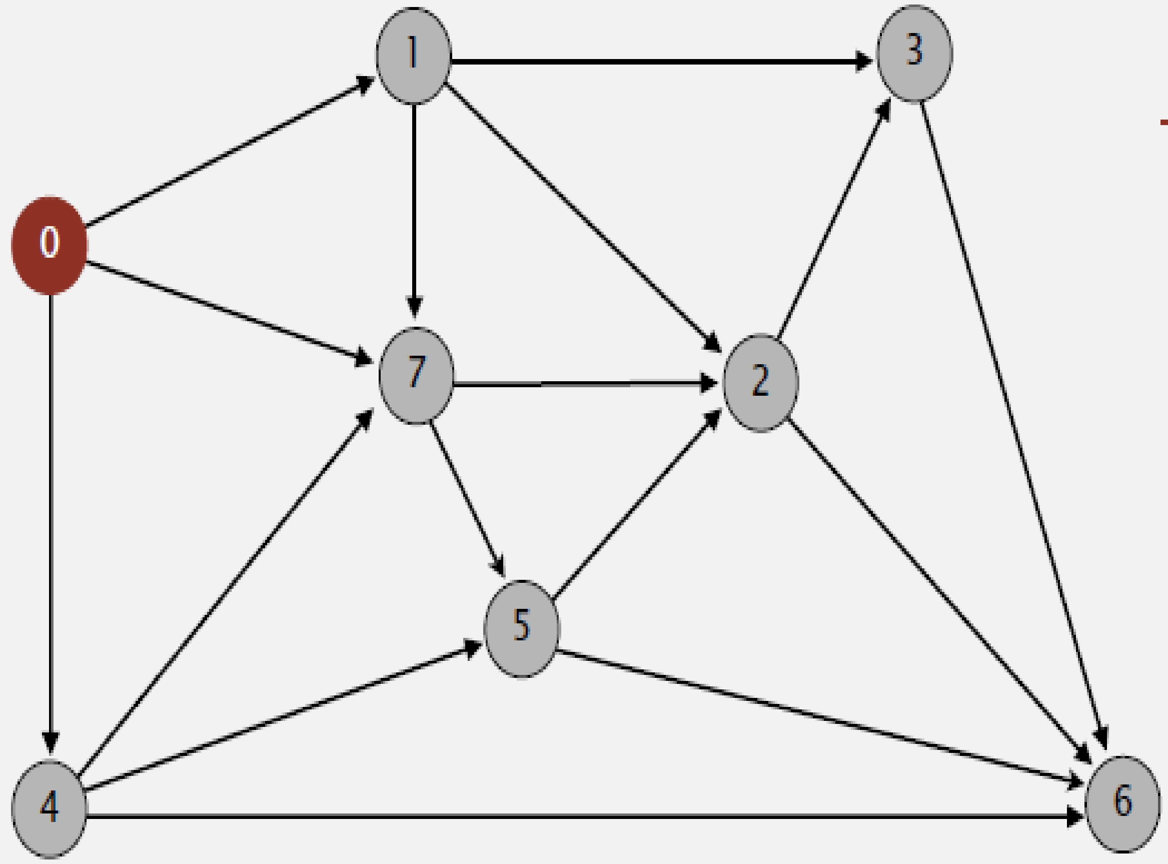


Algoritmo de Dijkstra



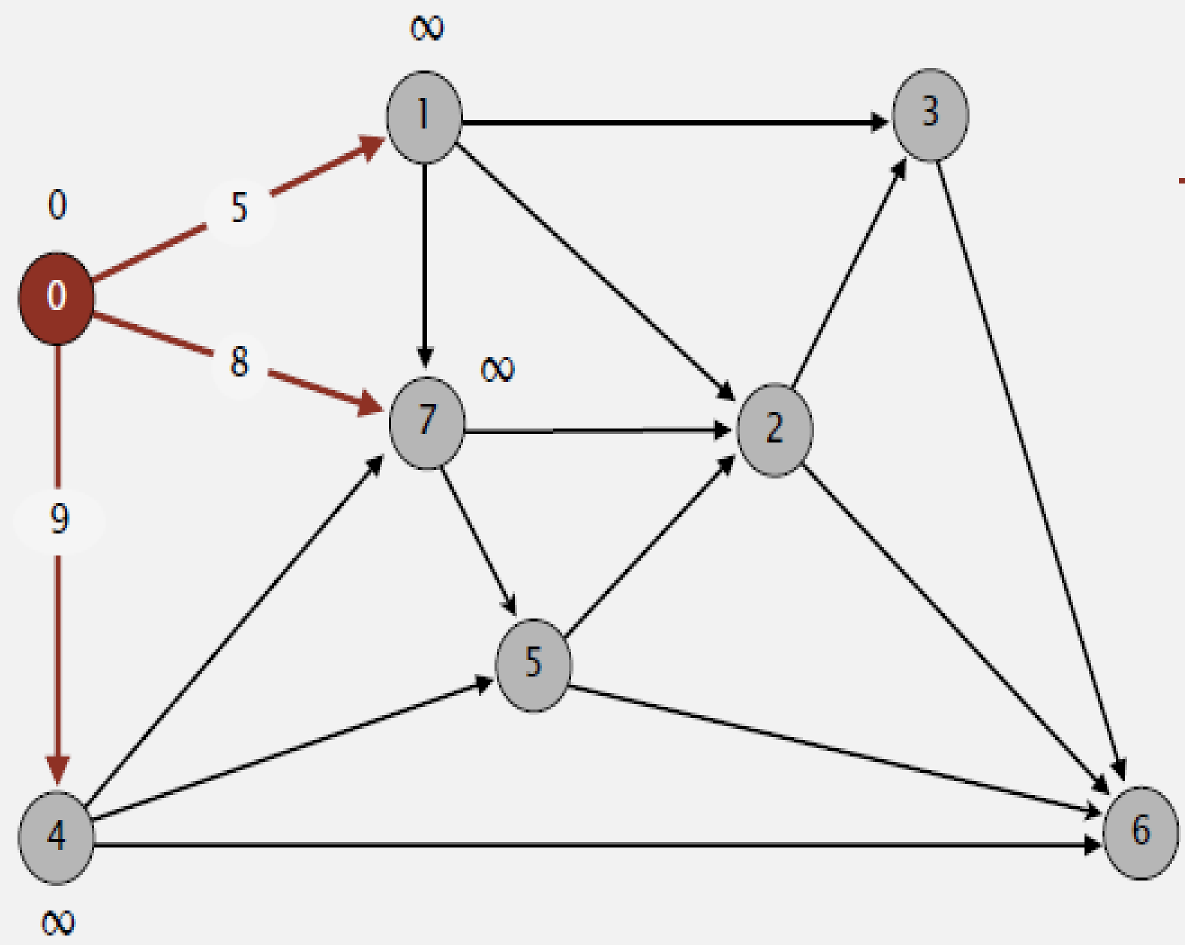
0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

Algoritmo de Dijkstra



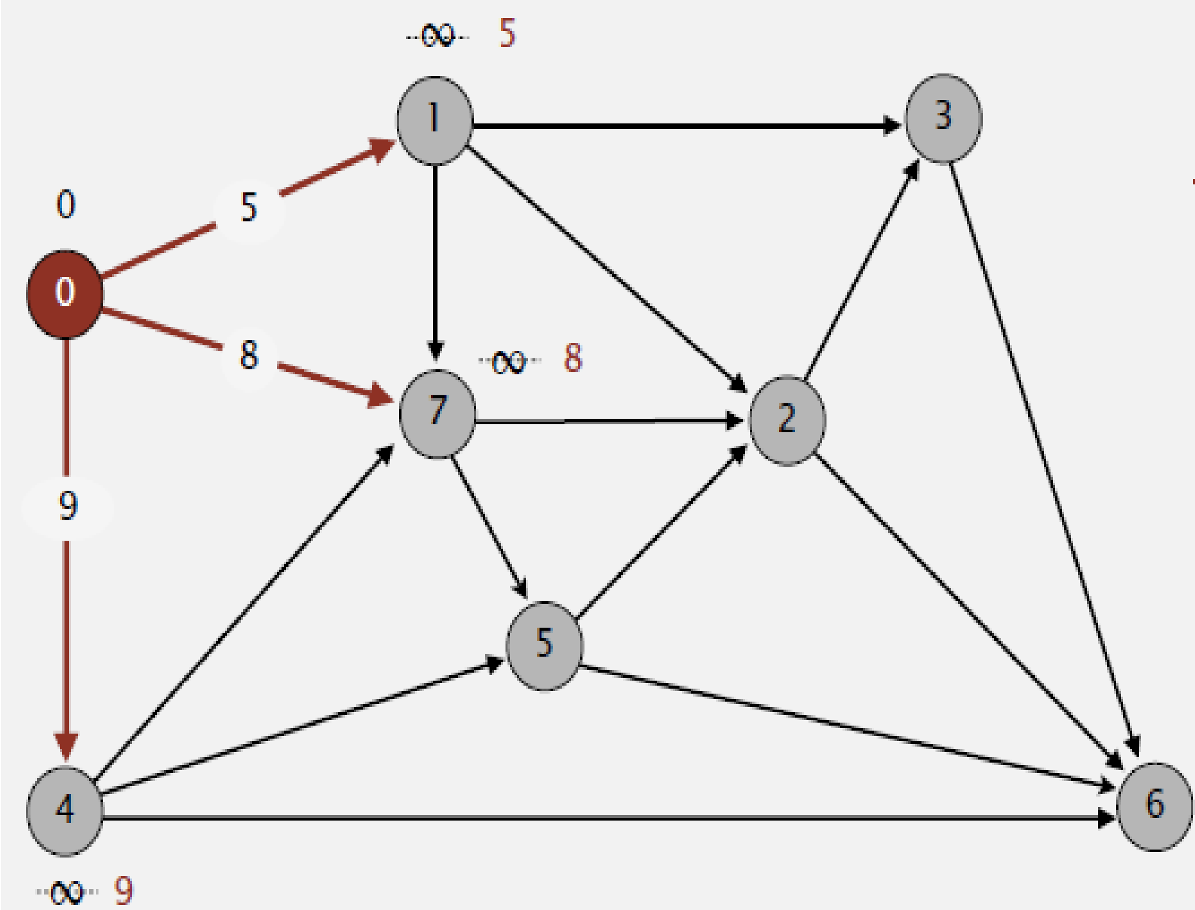
v	distTo[]	edgeTo[]
0	0.0	-
1		
2		
3		
4		
5		
6		
7		

Algoritmo de Dijkstra



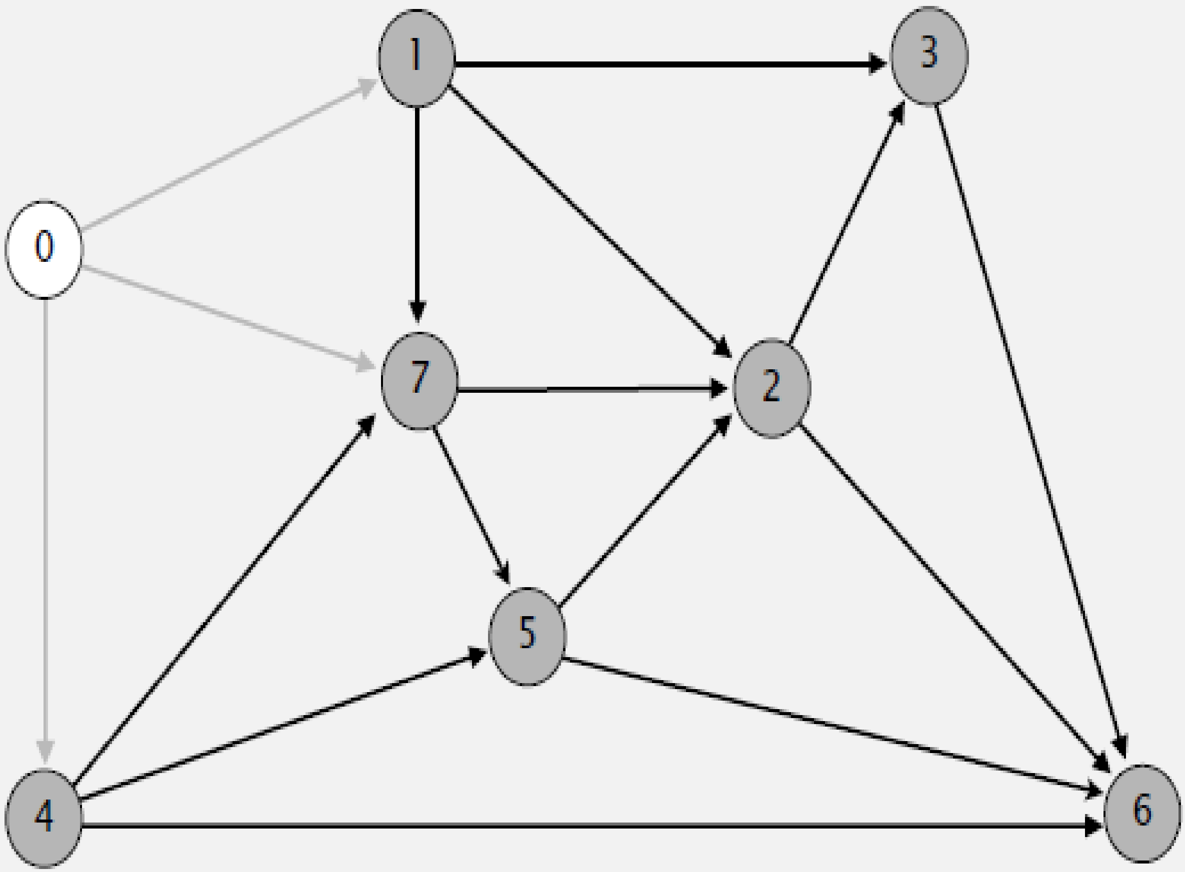
v	distTo[]	edgeTo[]
0	0.0	-
1		
2		
3		
4		
5		
6		
7		

Algoritmo de Dijkstra



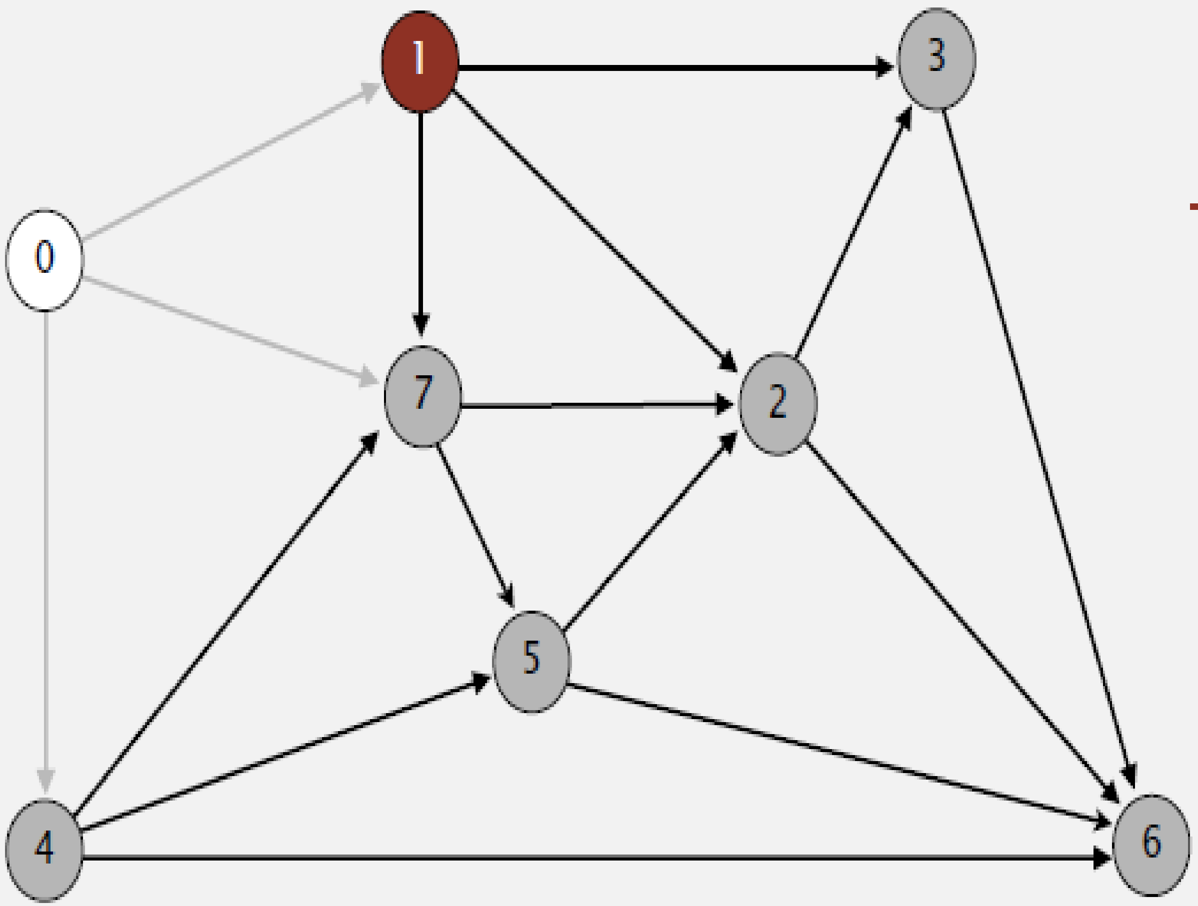
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

Algoritmo de Dijkstra



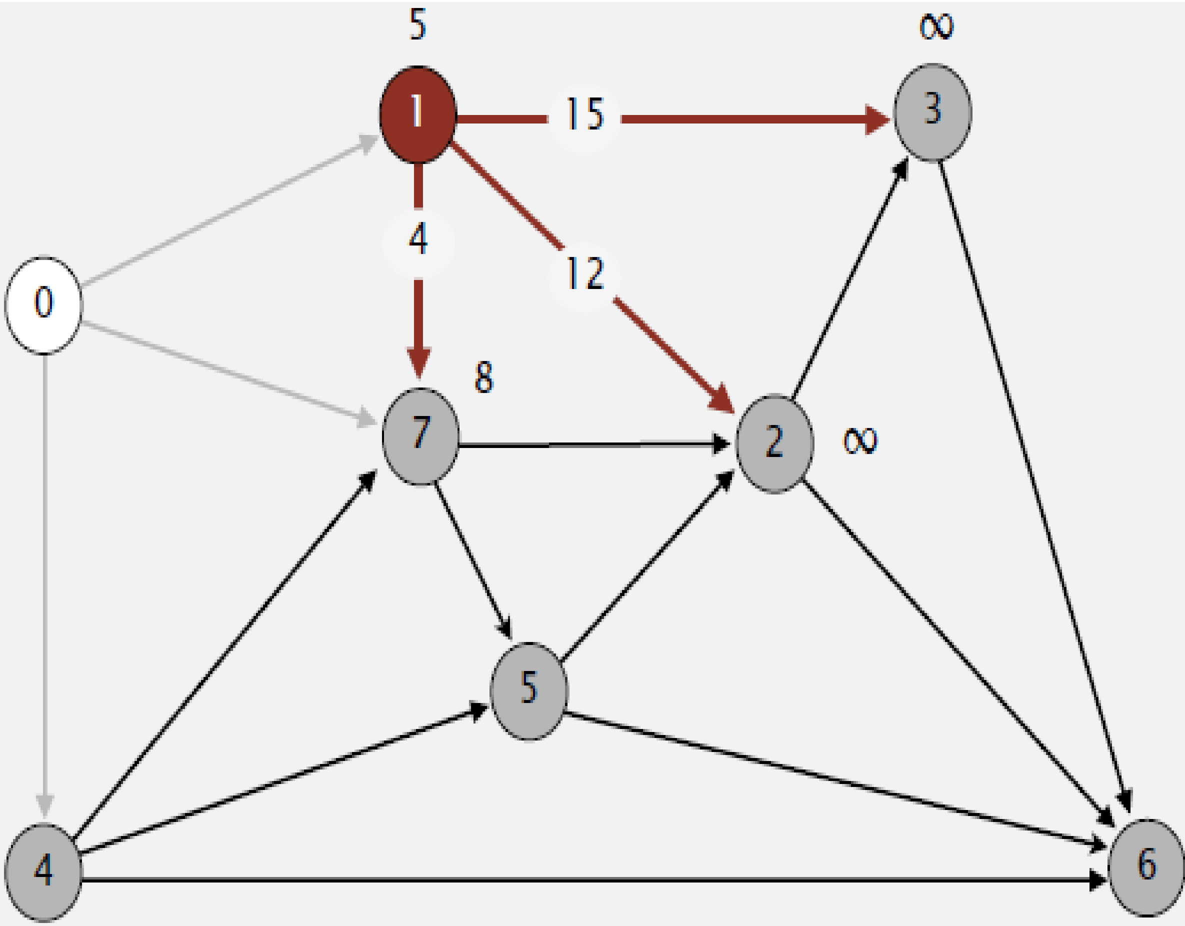
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

Algoritmo de Dijkstra



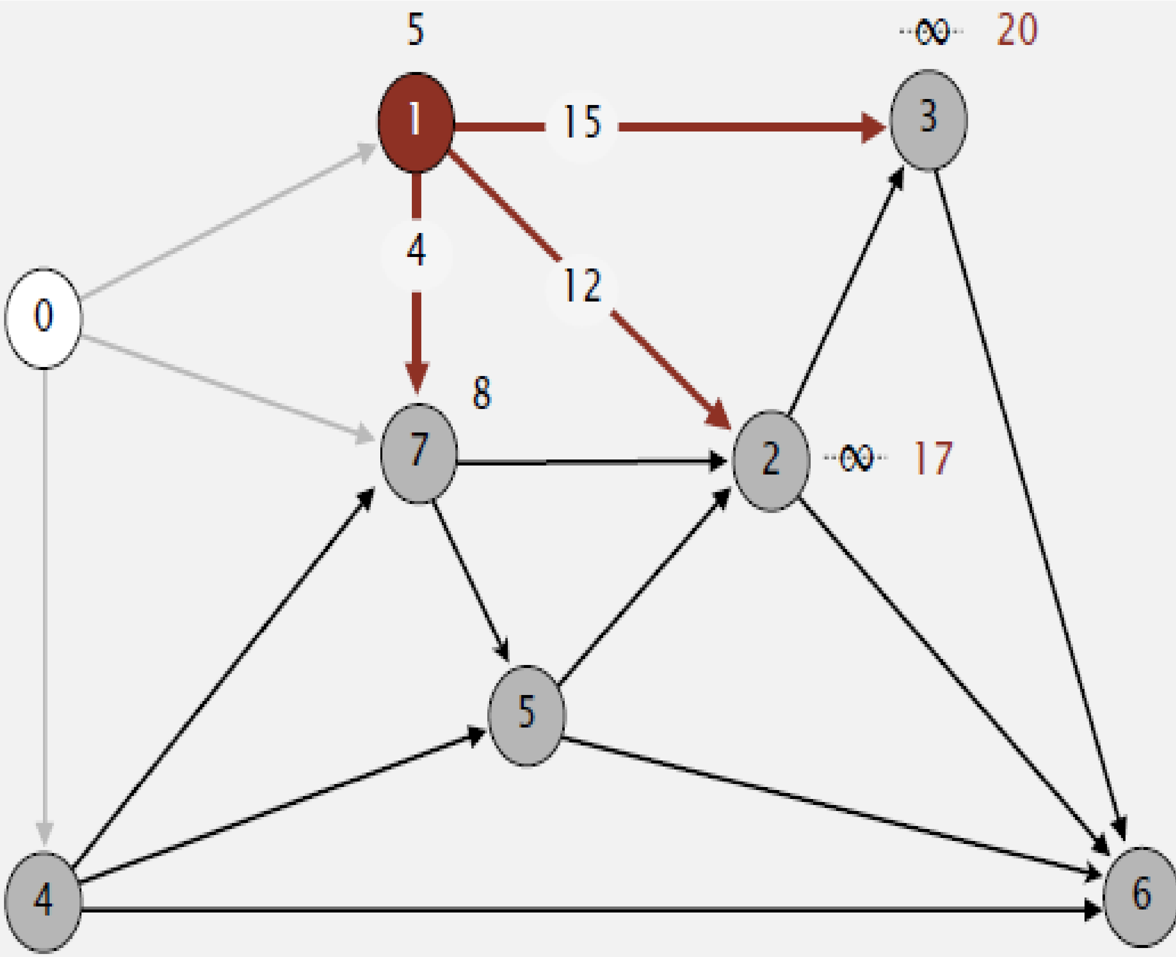
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

Algoritmo de Dijkstra



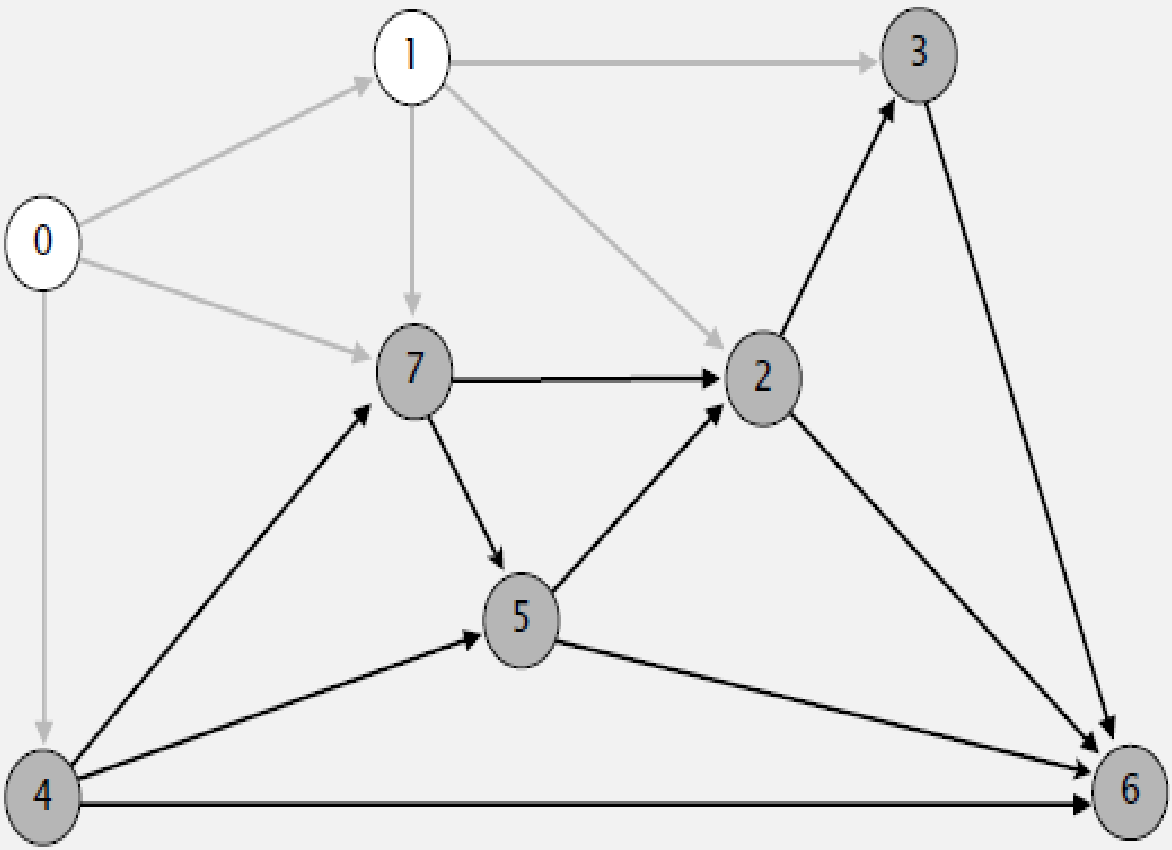
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

Algoritmo de Dijkstra



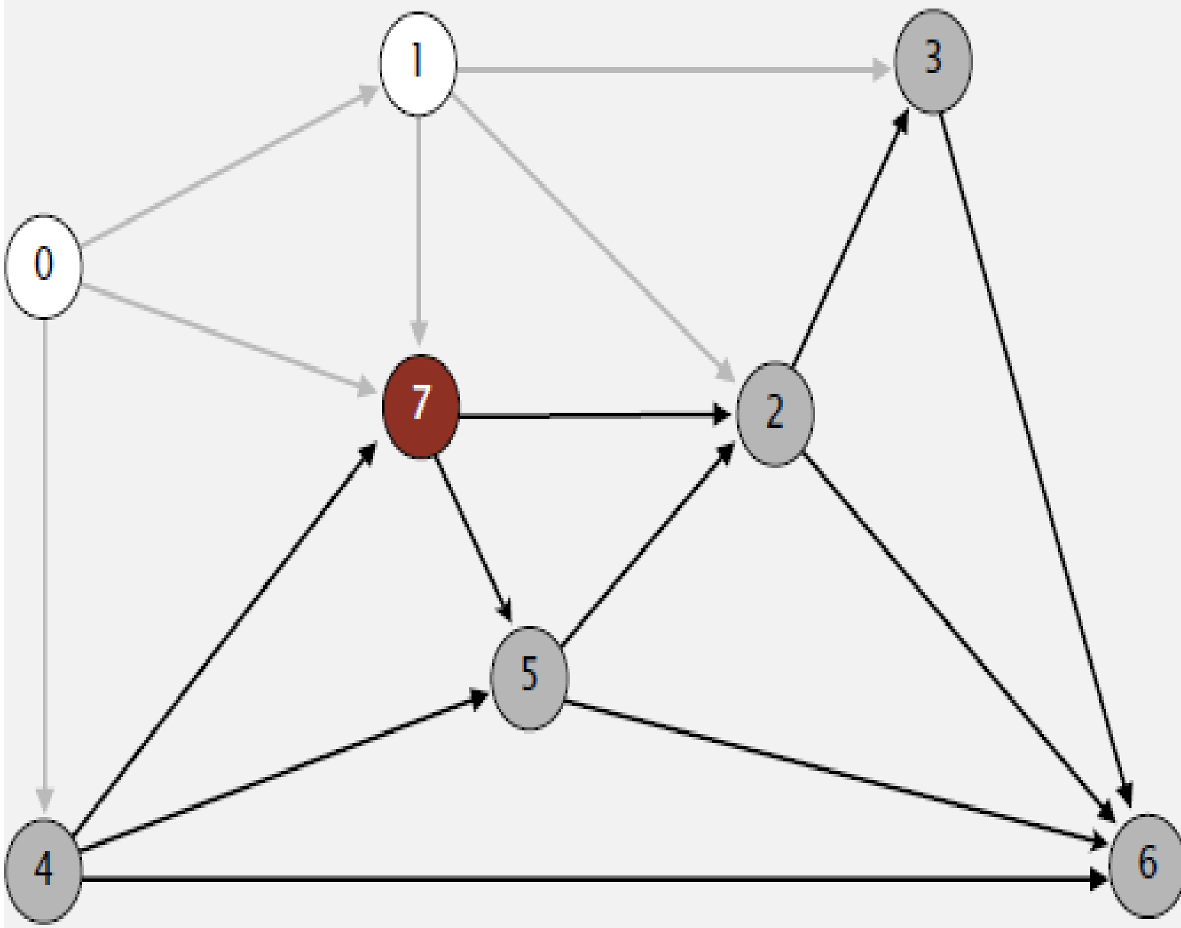
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0 ✔	0→7

Algoritmo de Dijkstra



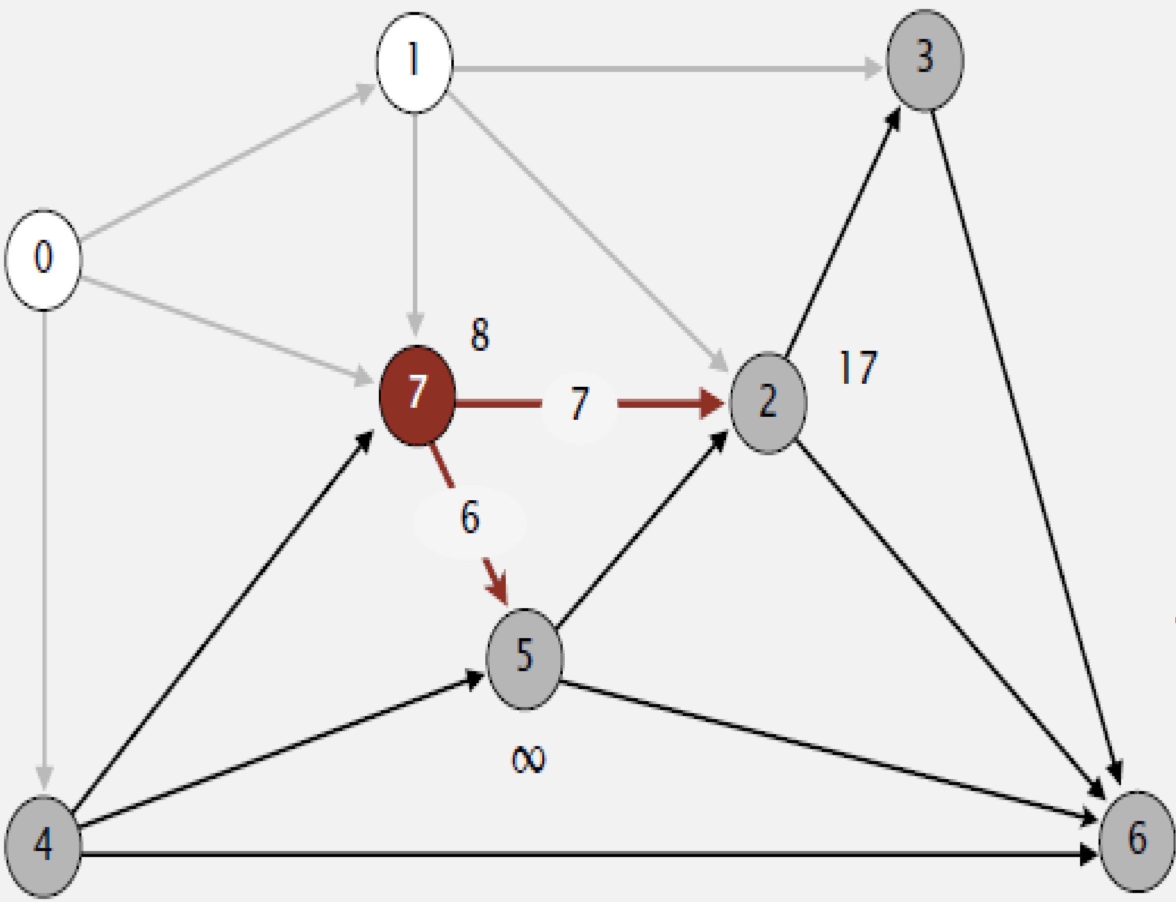
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

Algoritmo de Dijkstra



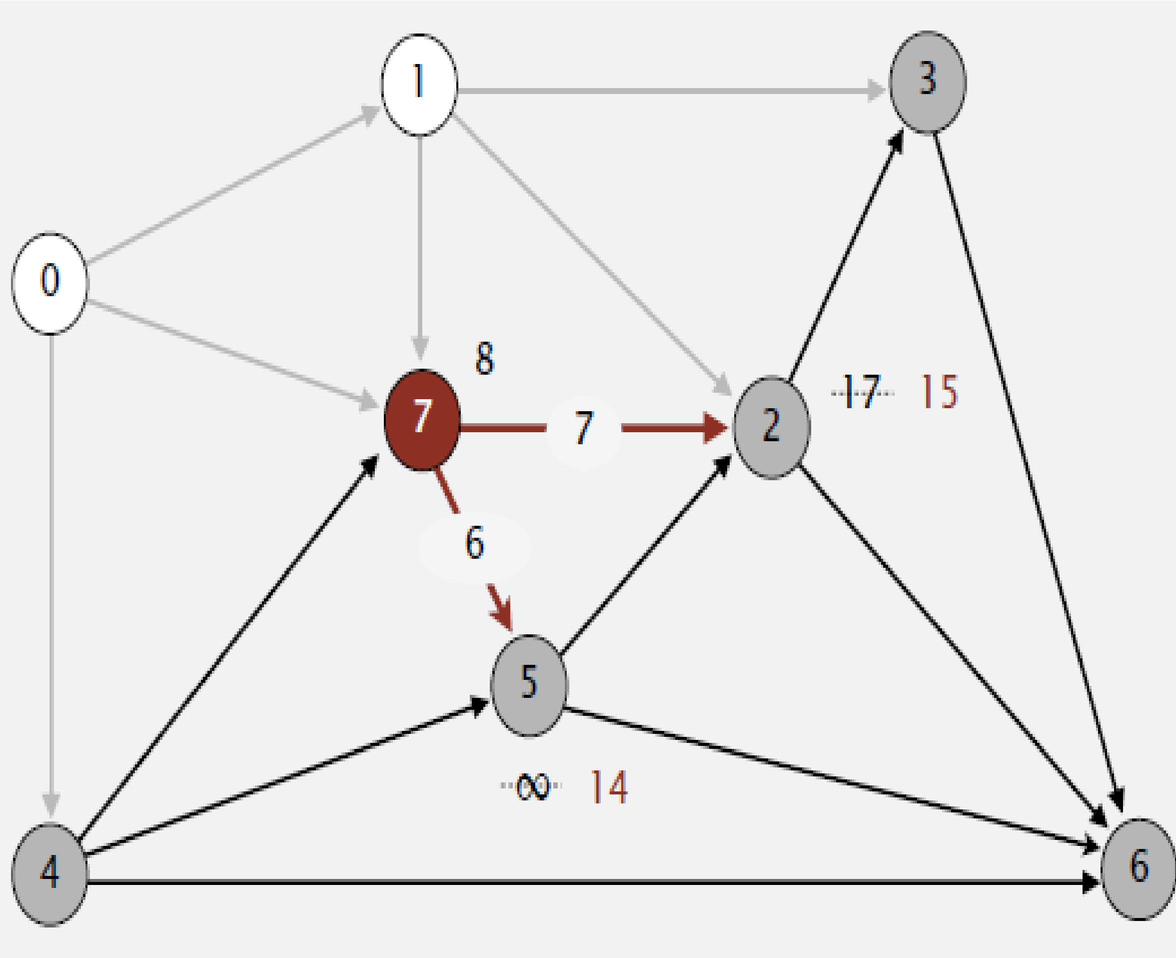
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

Algoritmo de Dijkstra



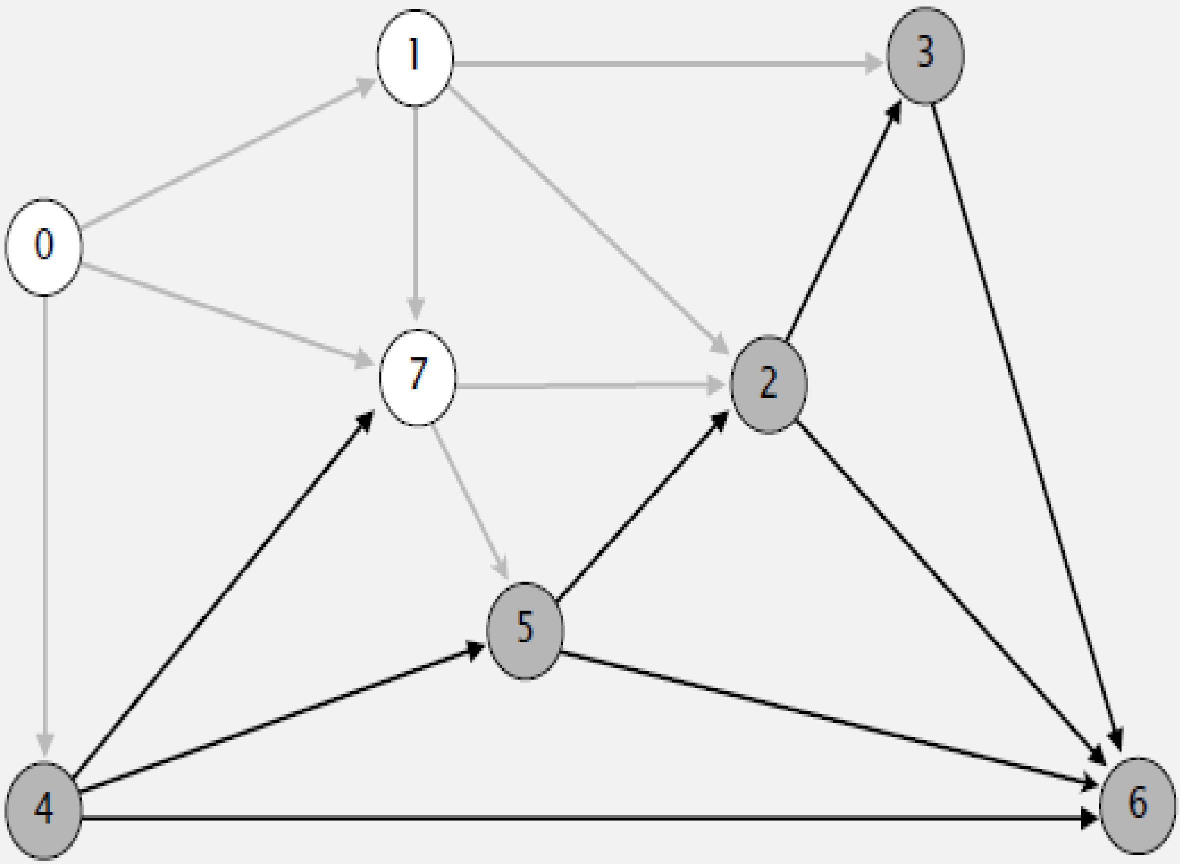
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
→ 7	8.0	0→7

Algoritmo de Dijkstra



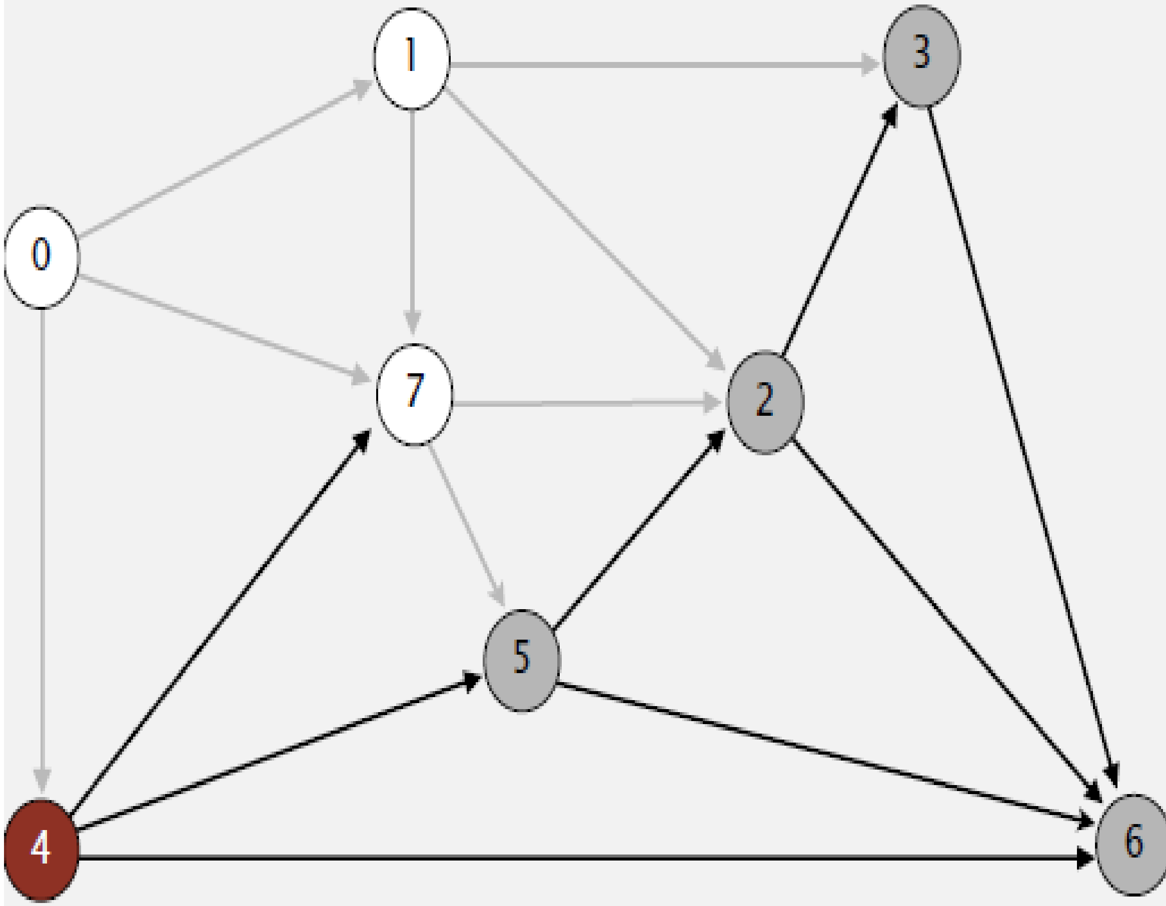
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	14.0	7→5
6		
→ 7	8.0	0→7

Algoritmo de Dijkstra



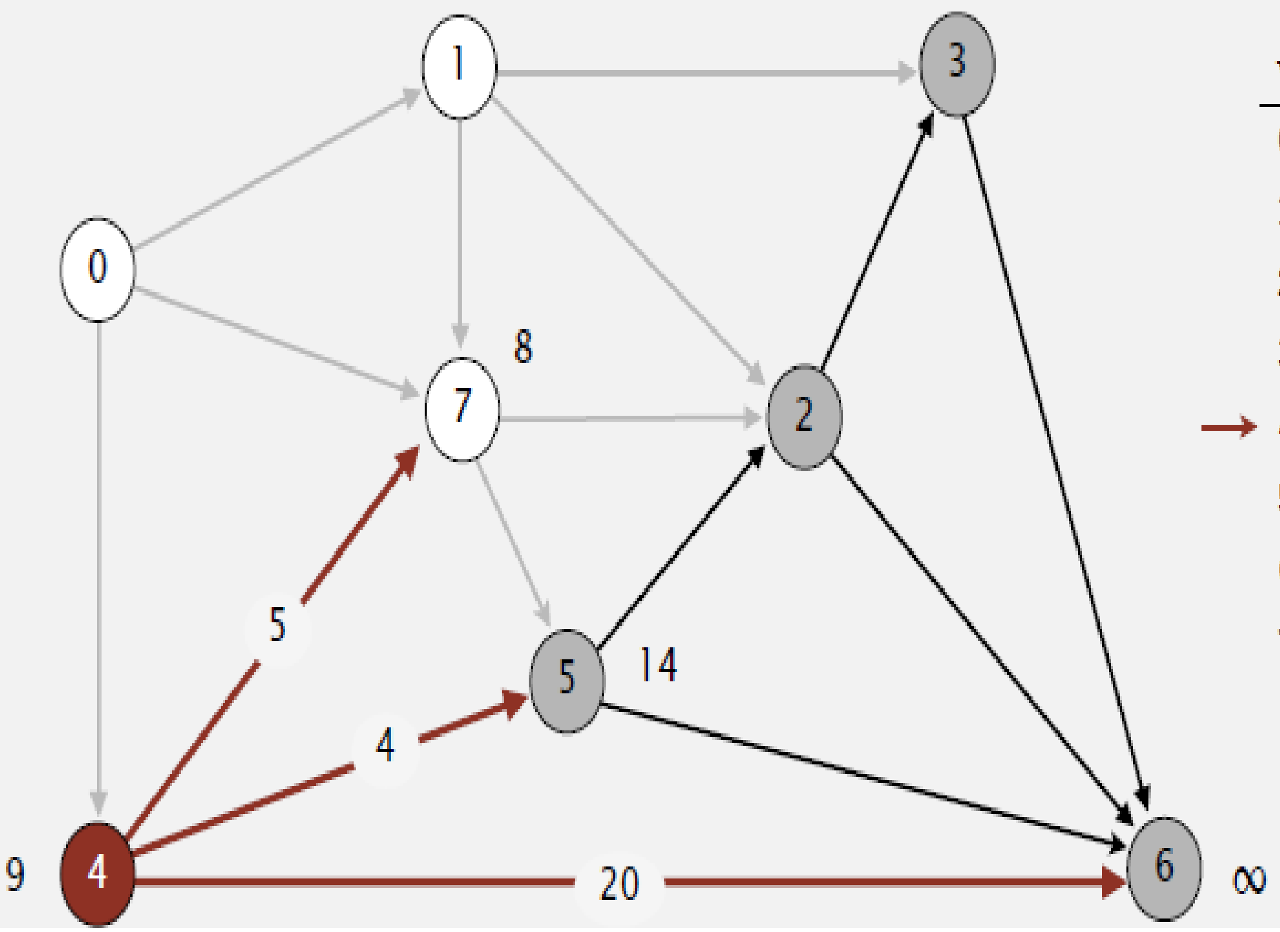
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	14.0	7→5
6		
7	8.0	0→7

Algoritmo de Dijkstra



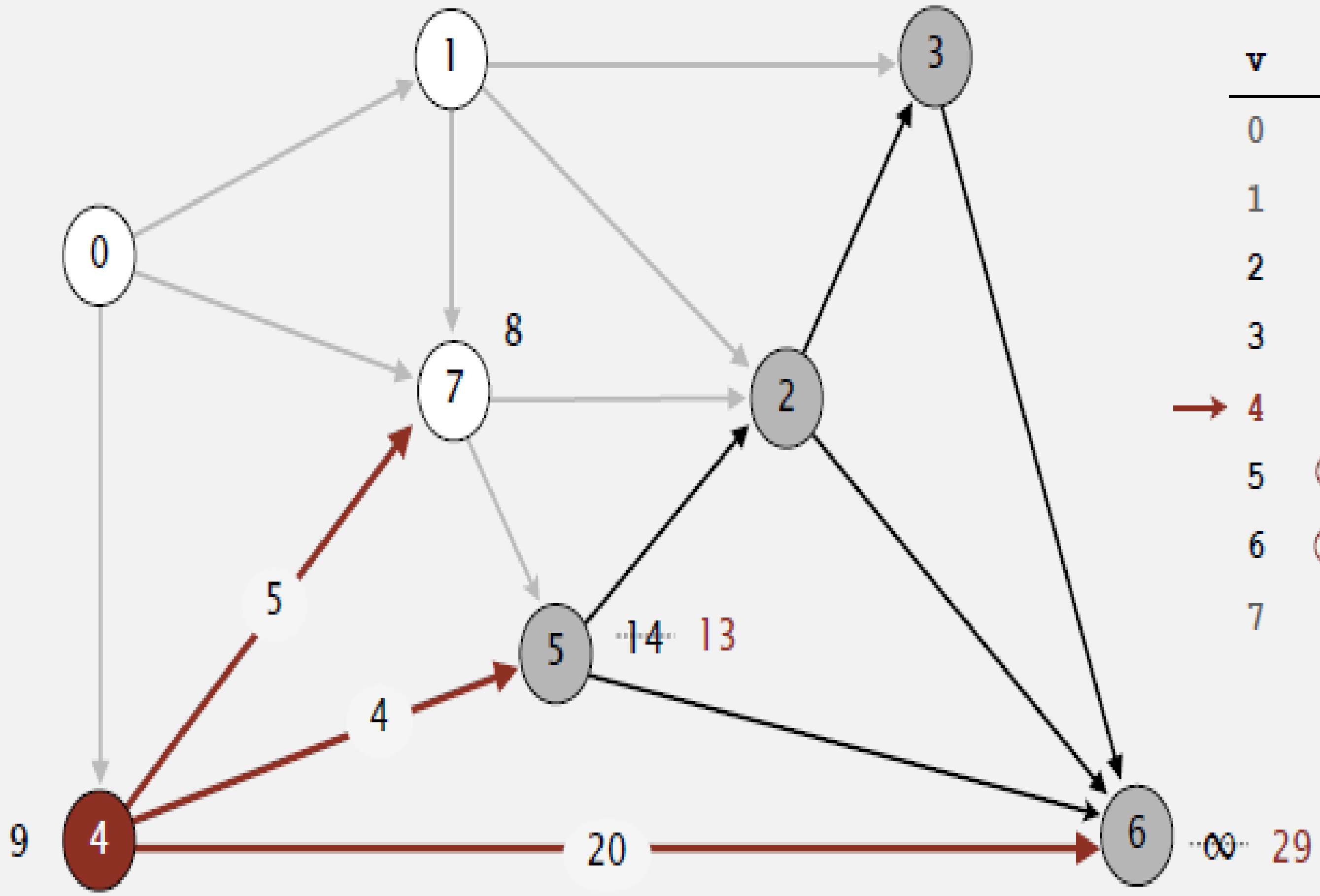
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	14.0	7→5
6		
7	8.0	0→7

Algoritmo de Dijkstra



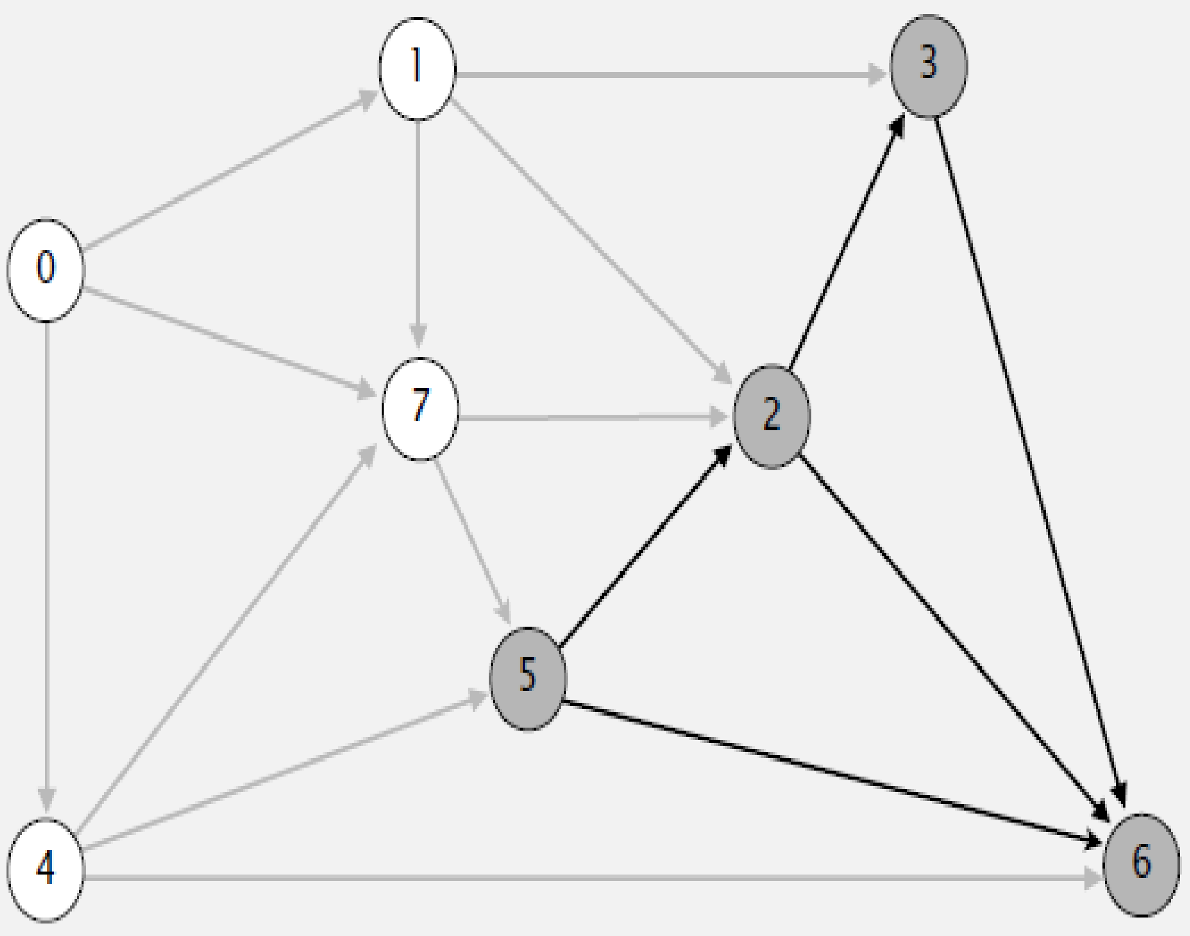
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	14.0	7→5
6		
7	8.0	0→7

Algoritmo de Dijkstra



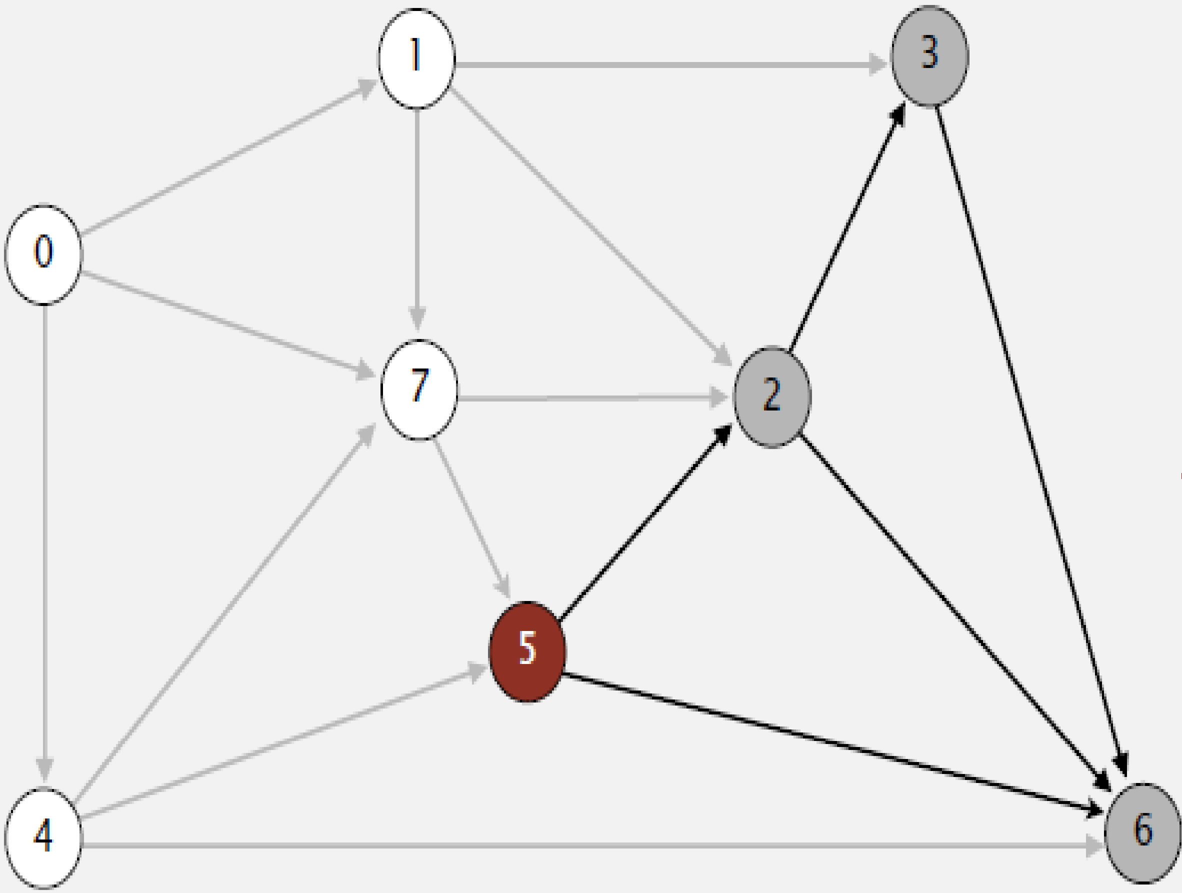
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0 ✔	0→7

Algoritmo de Dijkstra



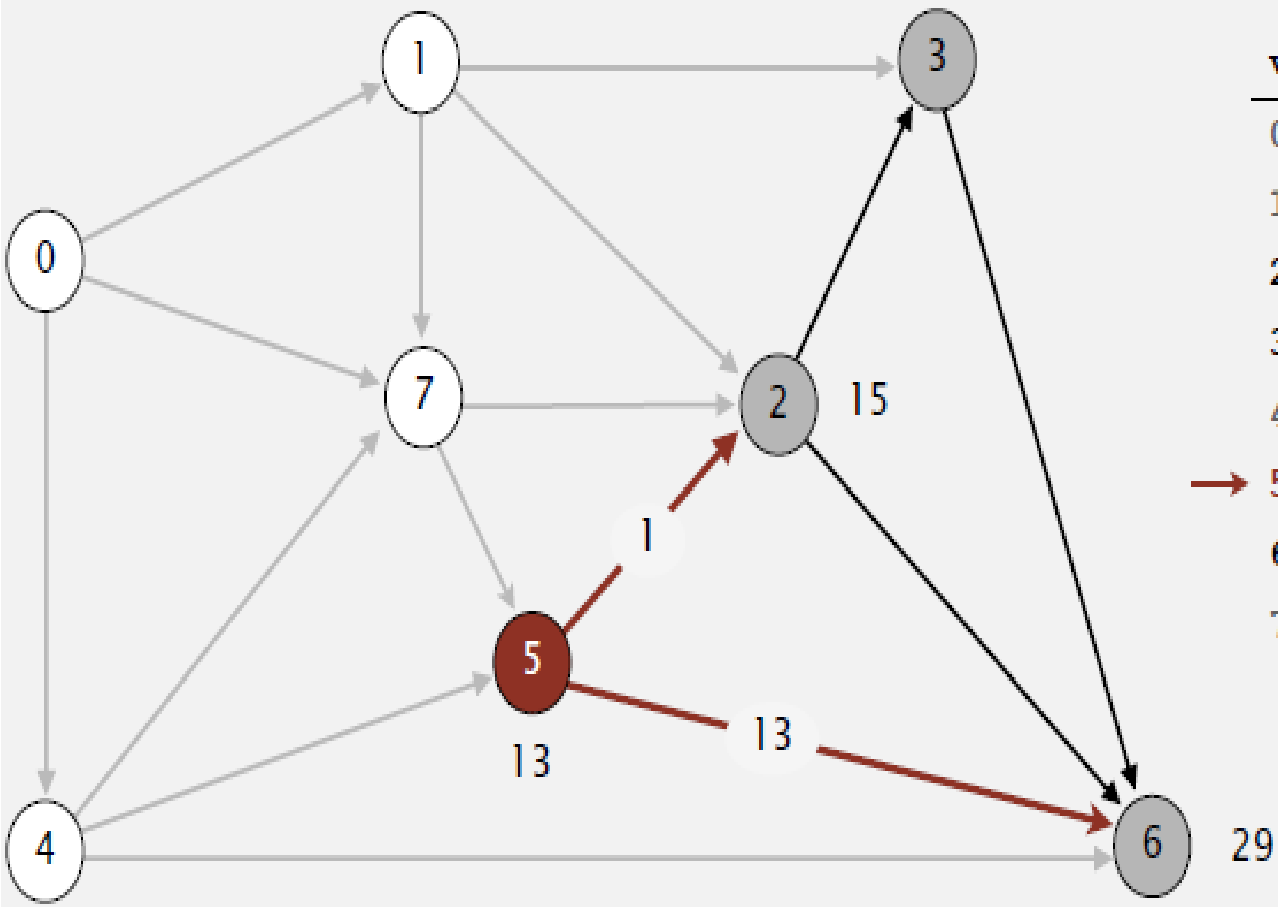
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

Algoritmo de Dijkstra



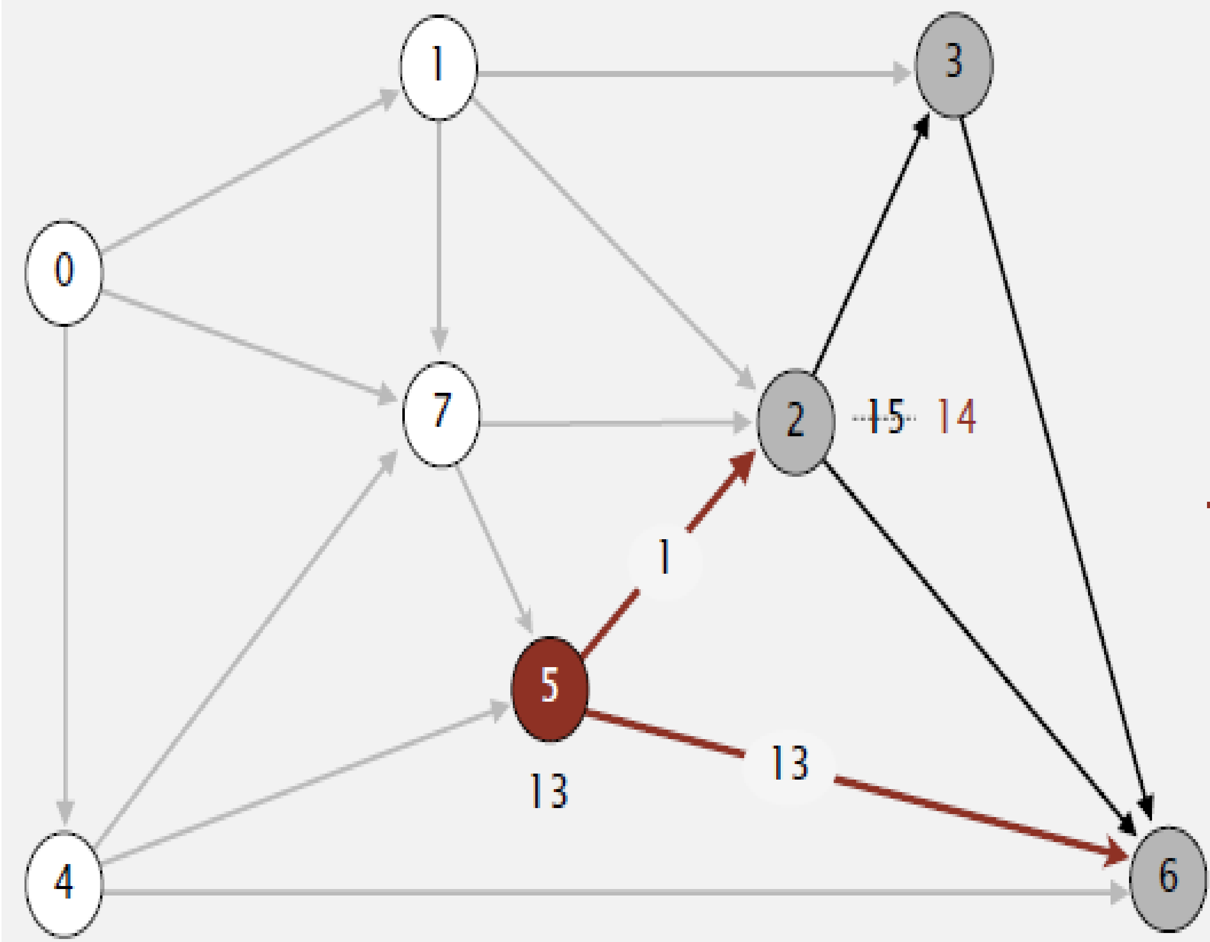
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

Algoritmo de Dijkstra



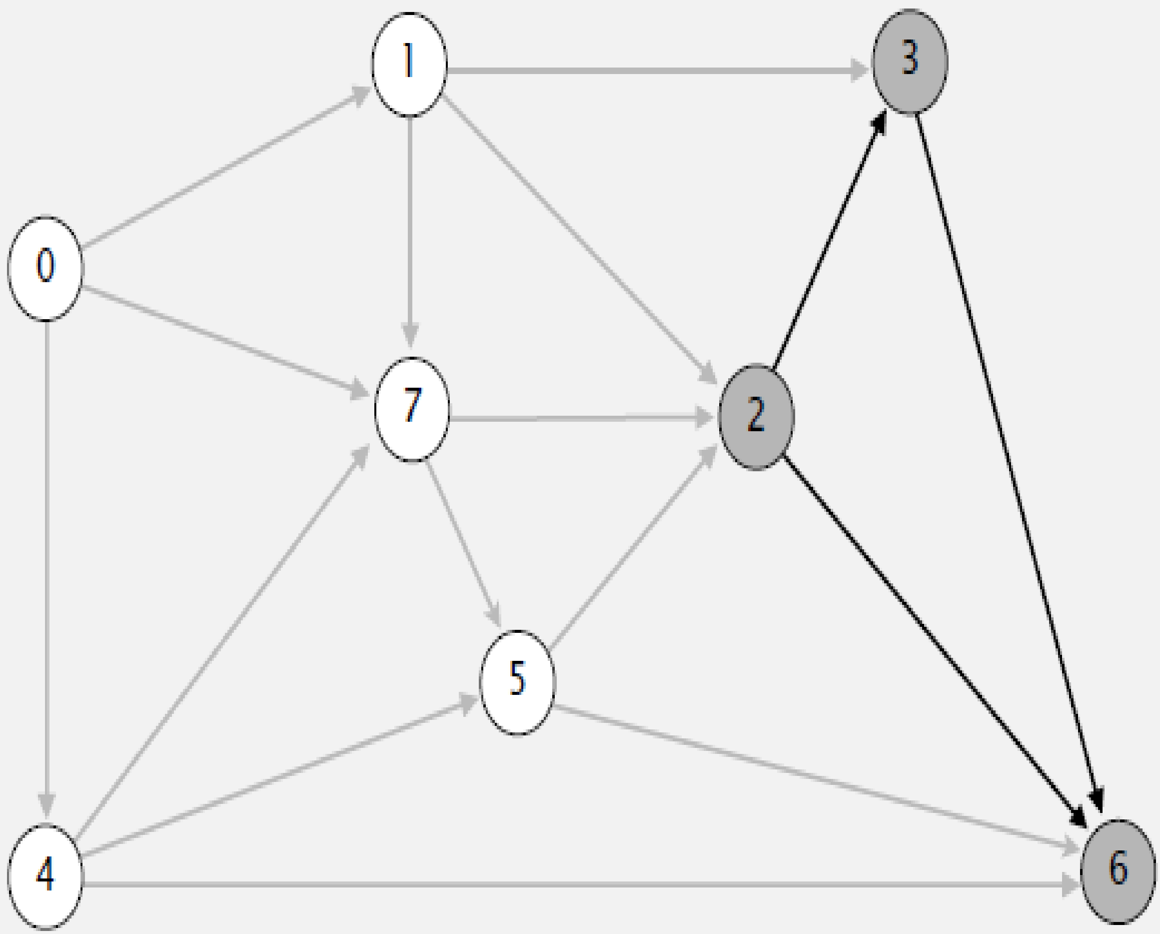
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
→ 5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

Algoritmo de Dijkstra



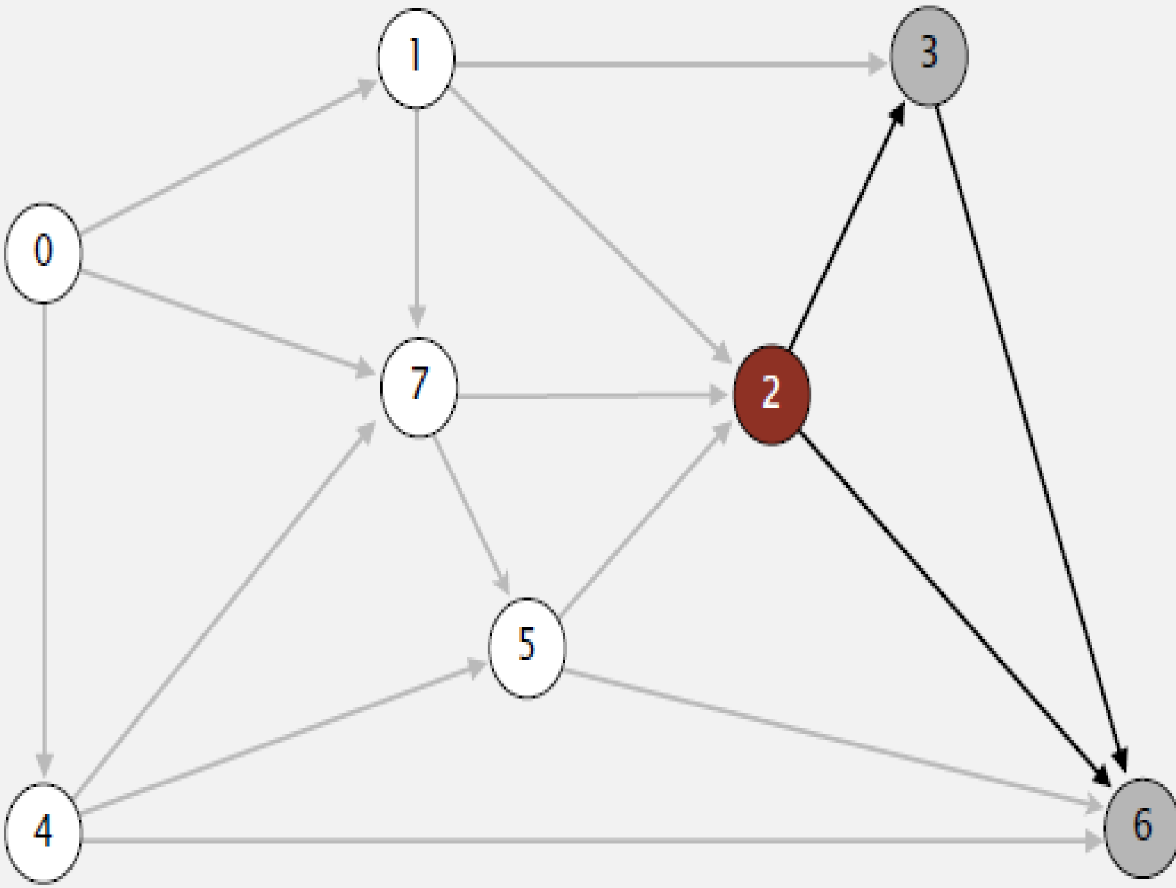
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

Algoritmo de Dijkstra



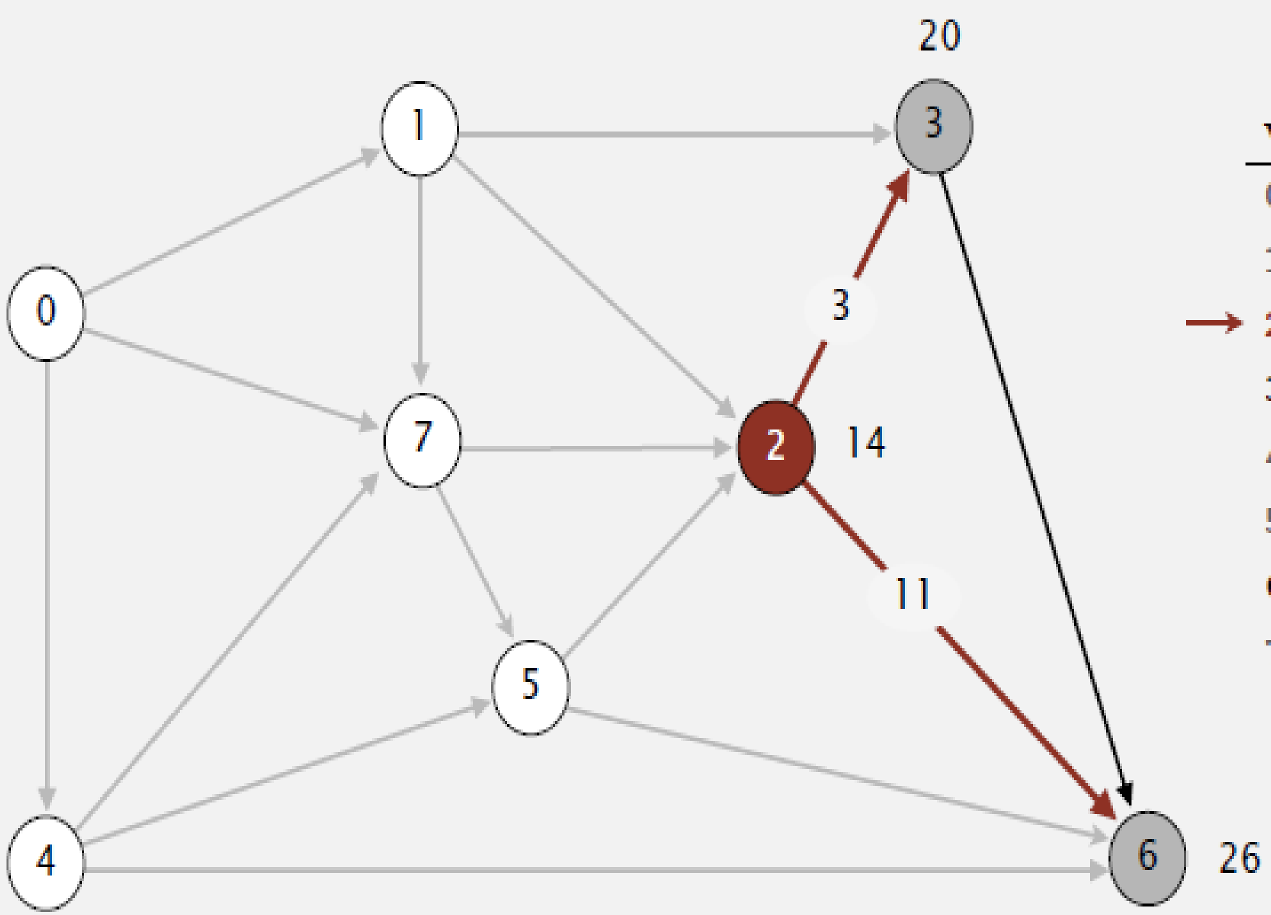
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

Algoritmo de Dijkstra



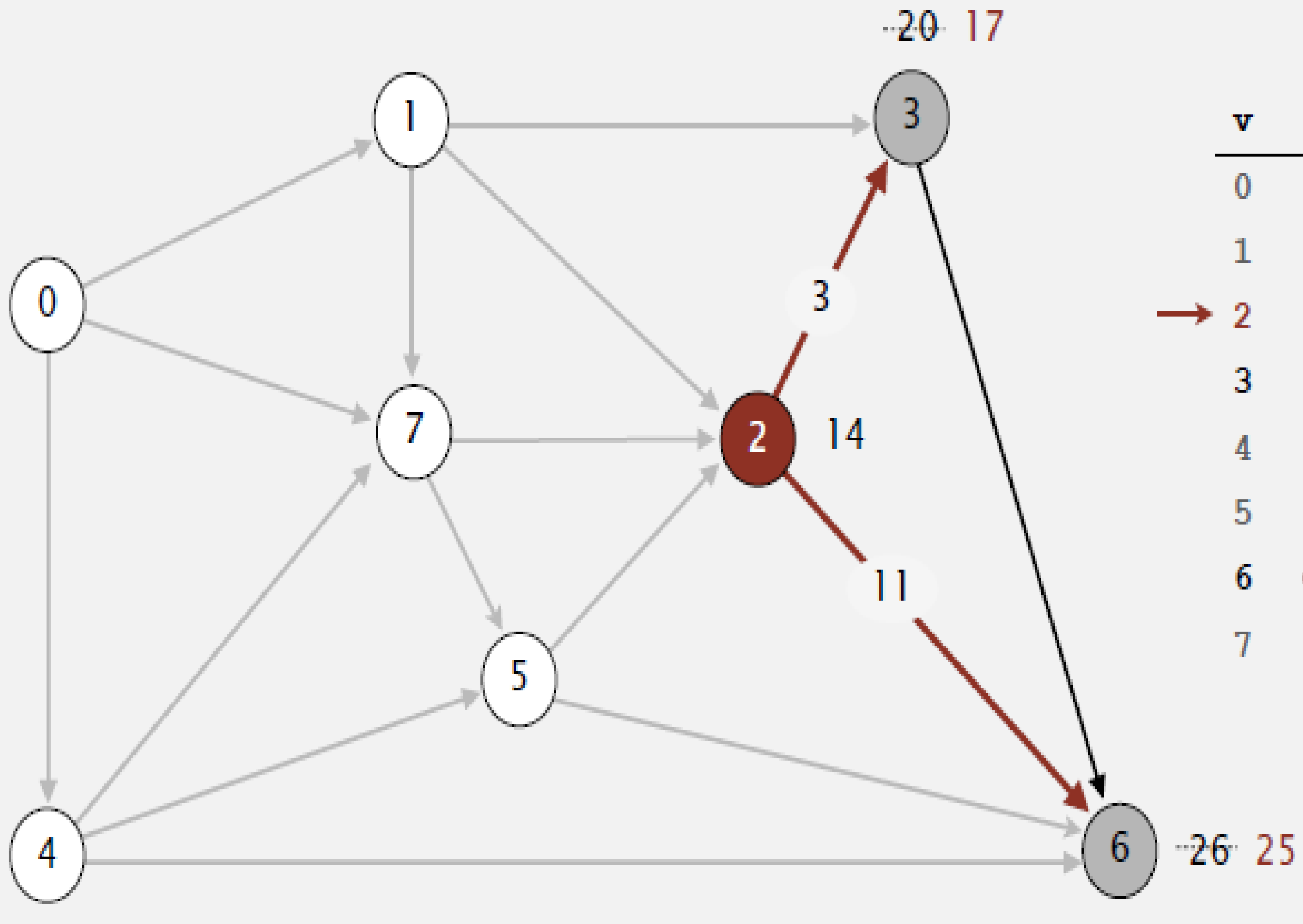
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

Algoritmo de Dijkstra



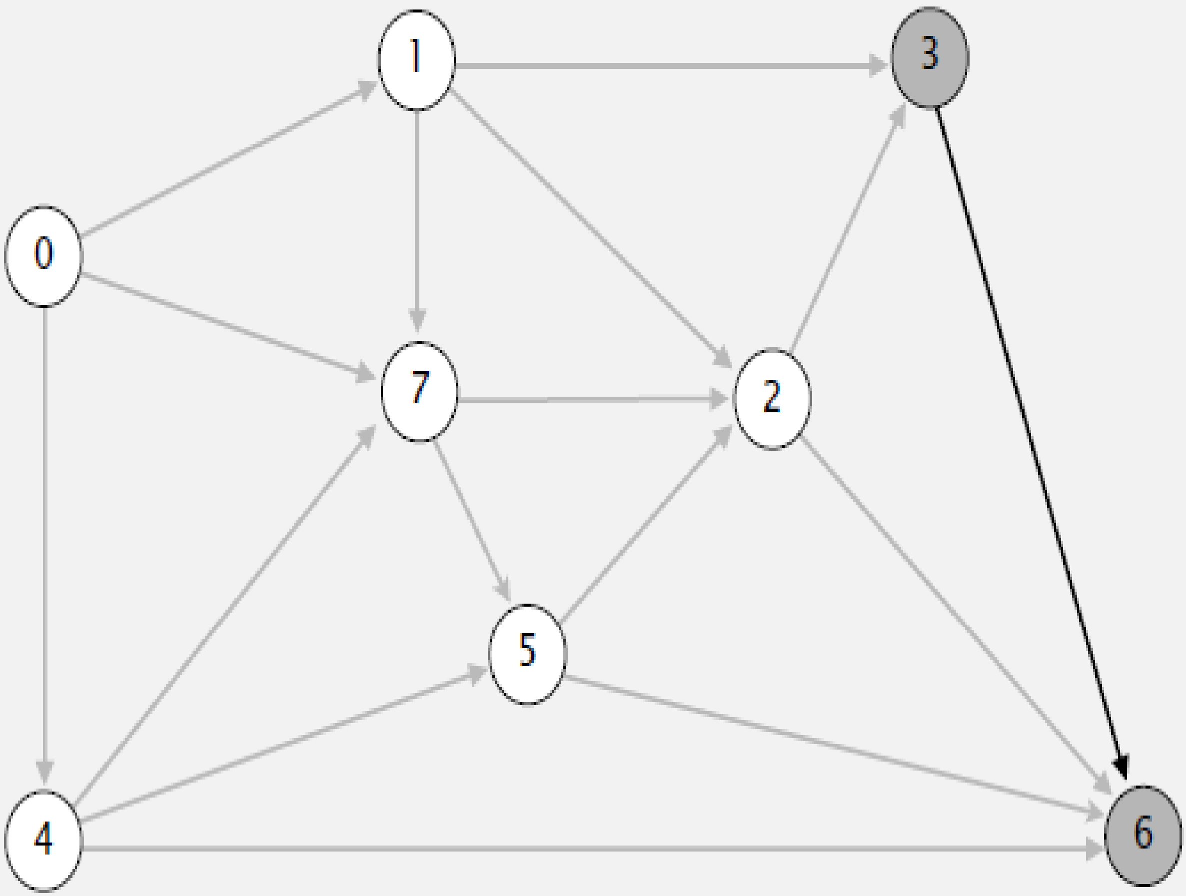
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

Algoritmo de Dijkstra



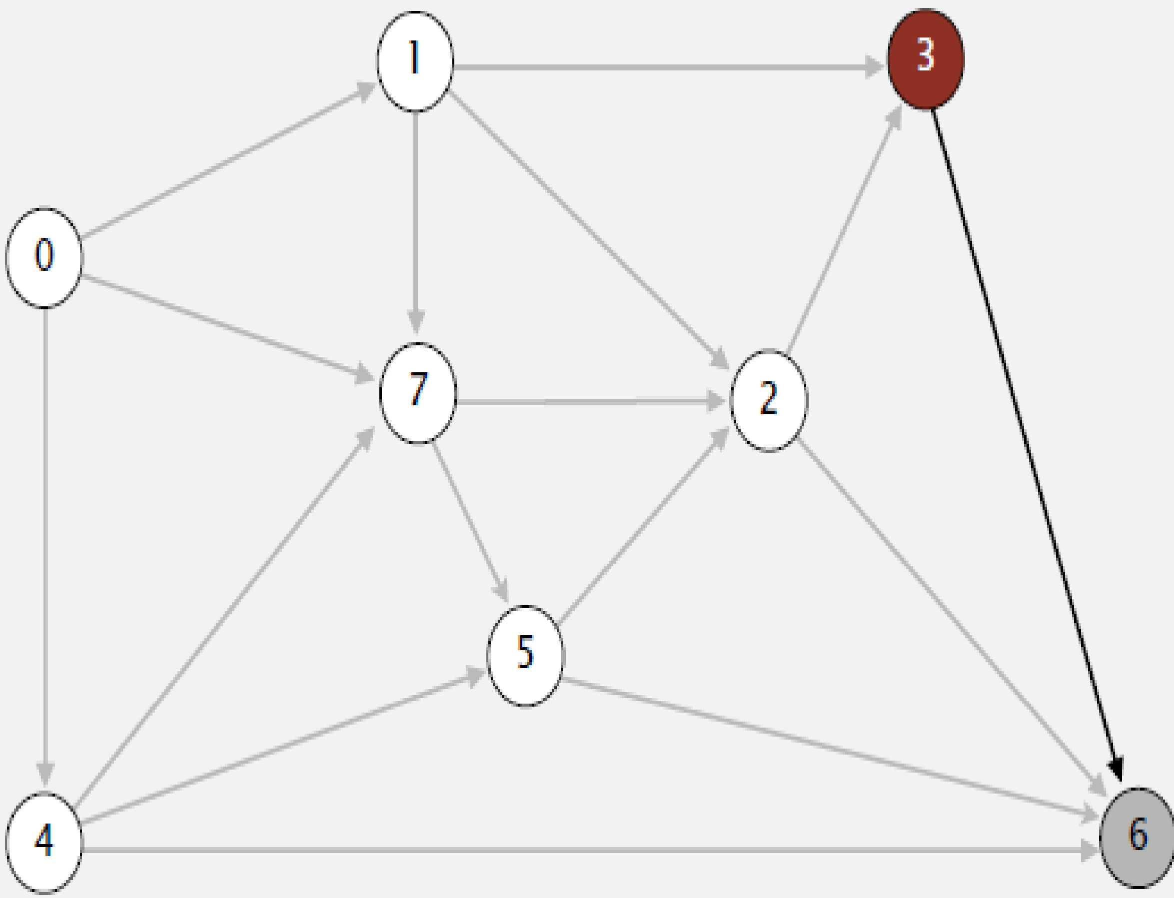
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Algoritmo de Dijkstra



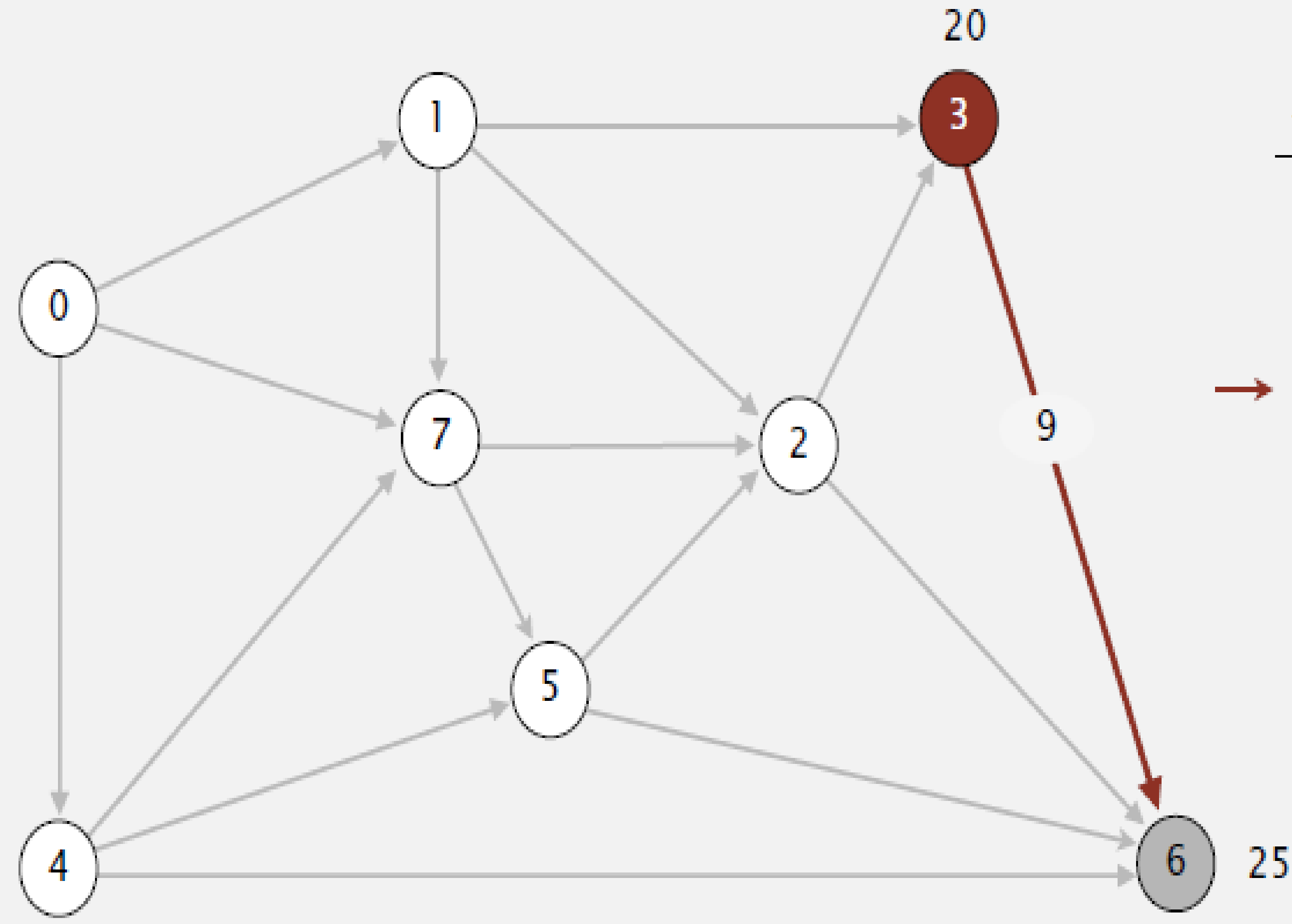
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Algoritmo de Dijkstra



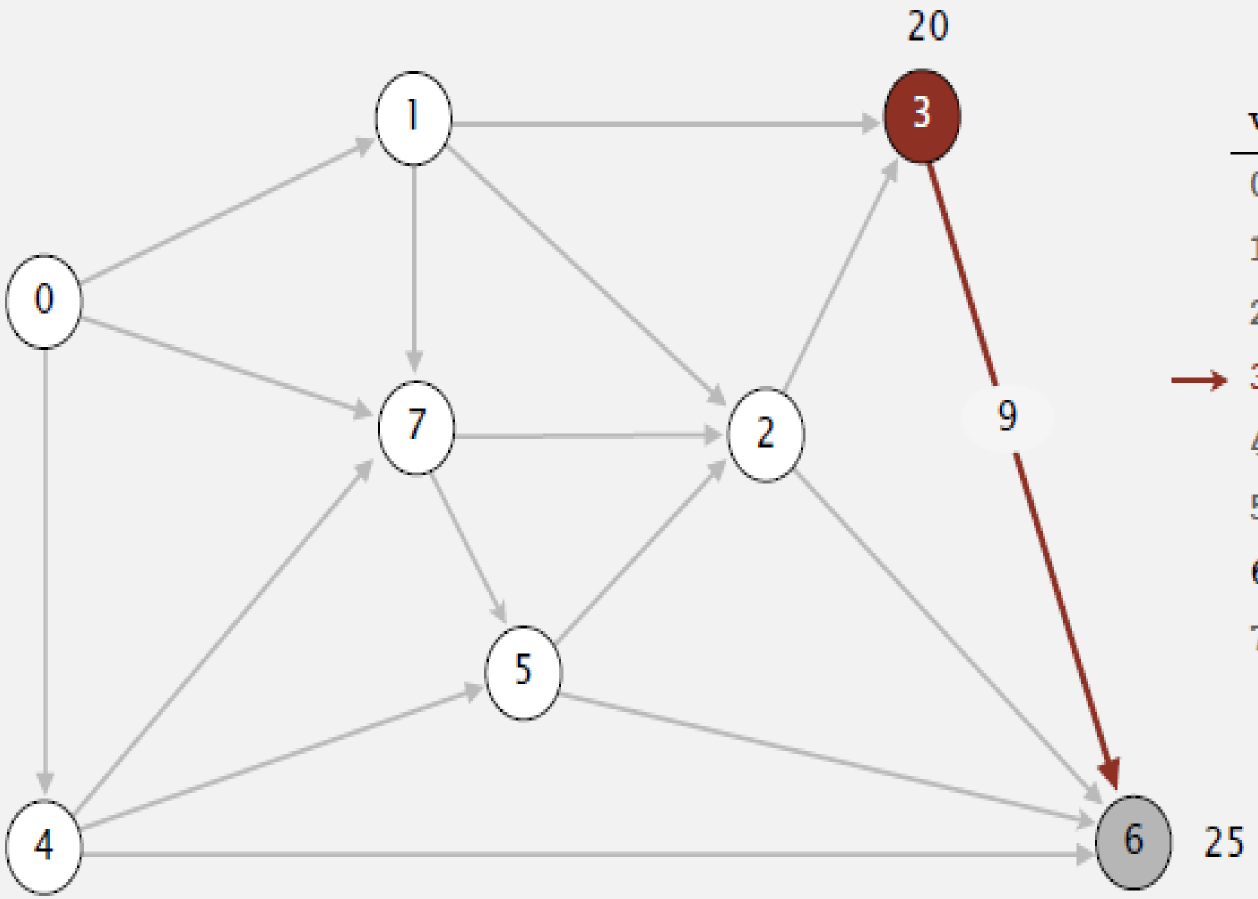
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Algoritmo de Dijkstra



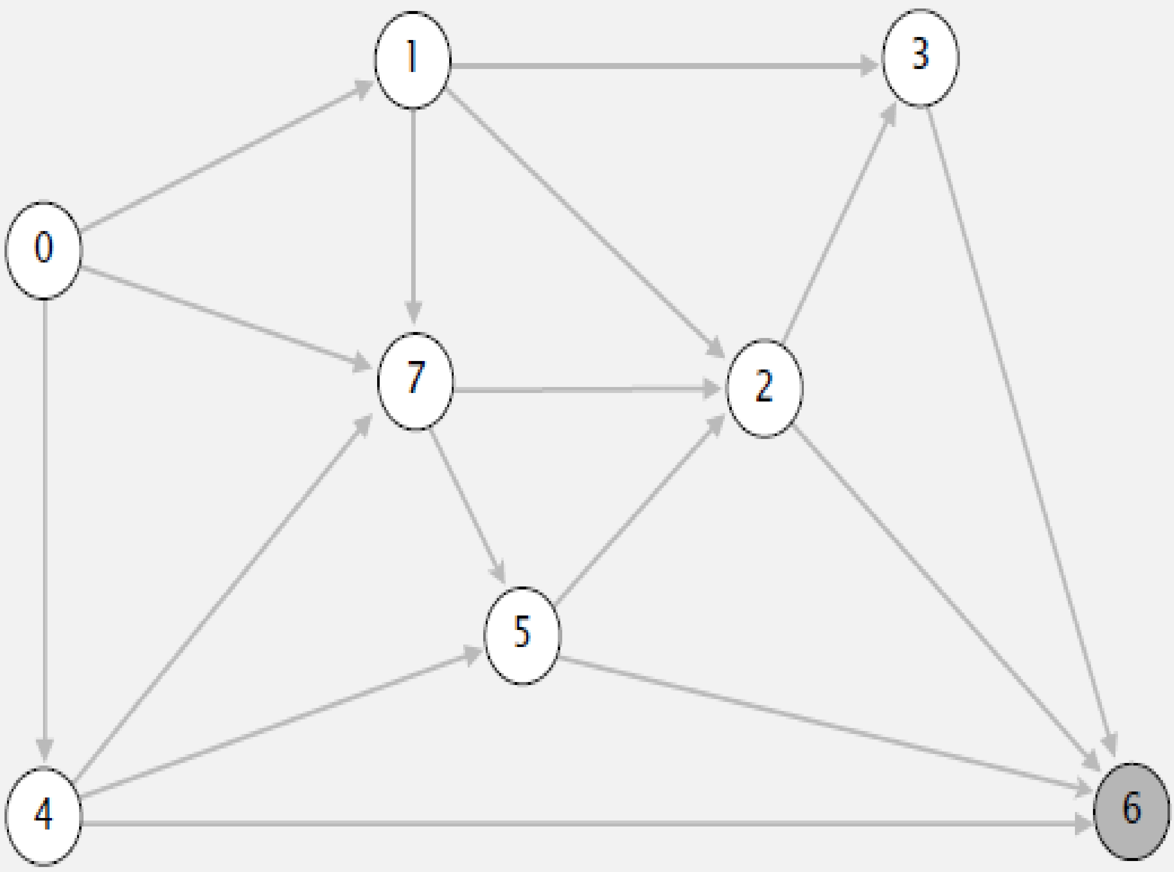
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Algoritmo de Dijkstra



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
→ 3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0 ✔	2→6
7	8.0	0→7

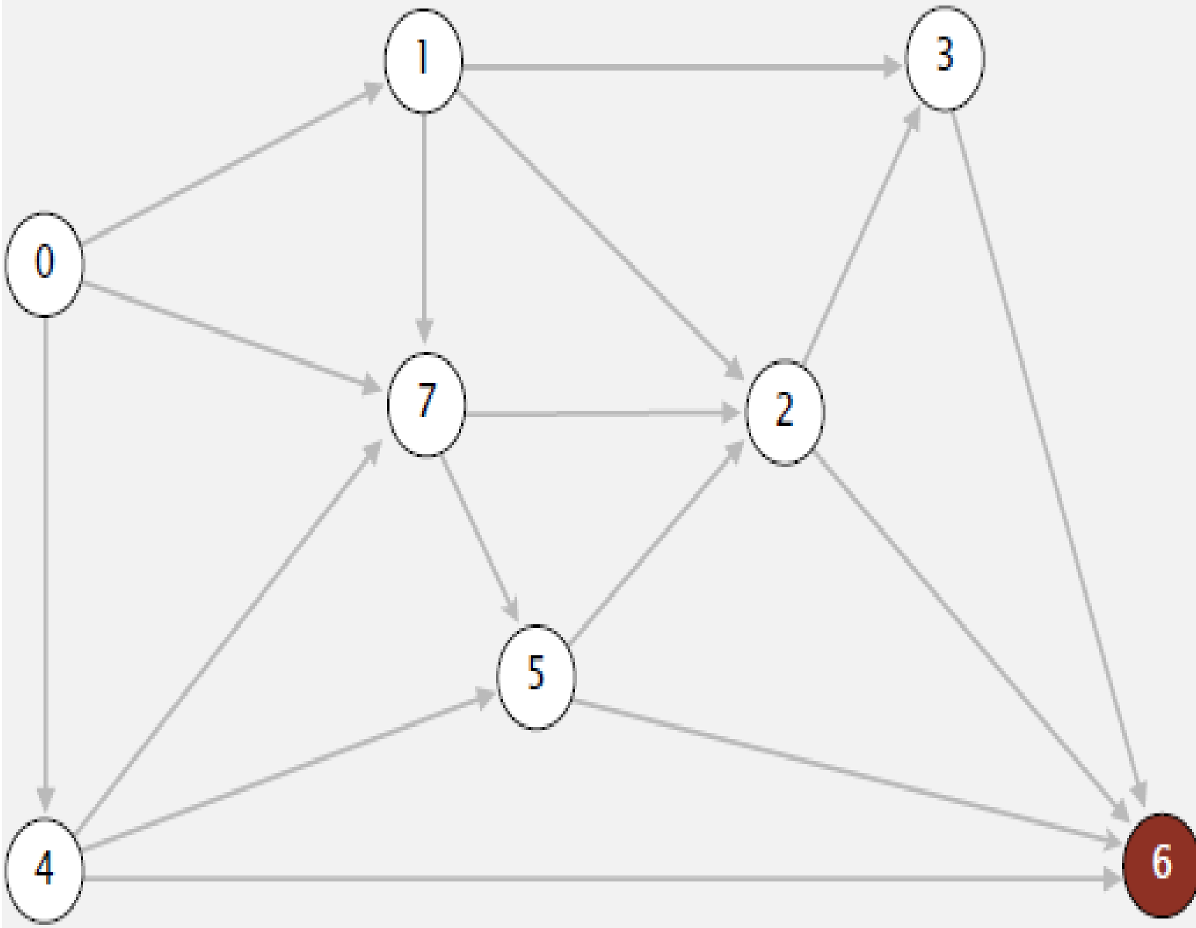
Algoritmo de Dijkstra



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

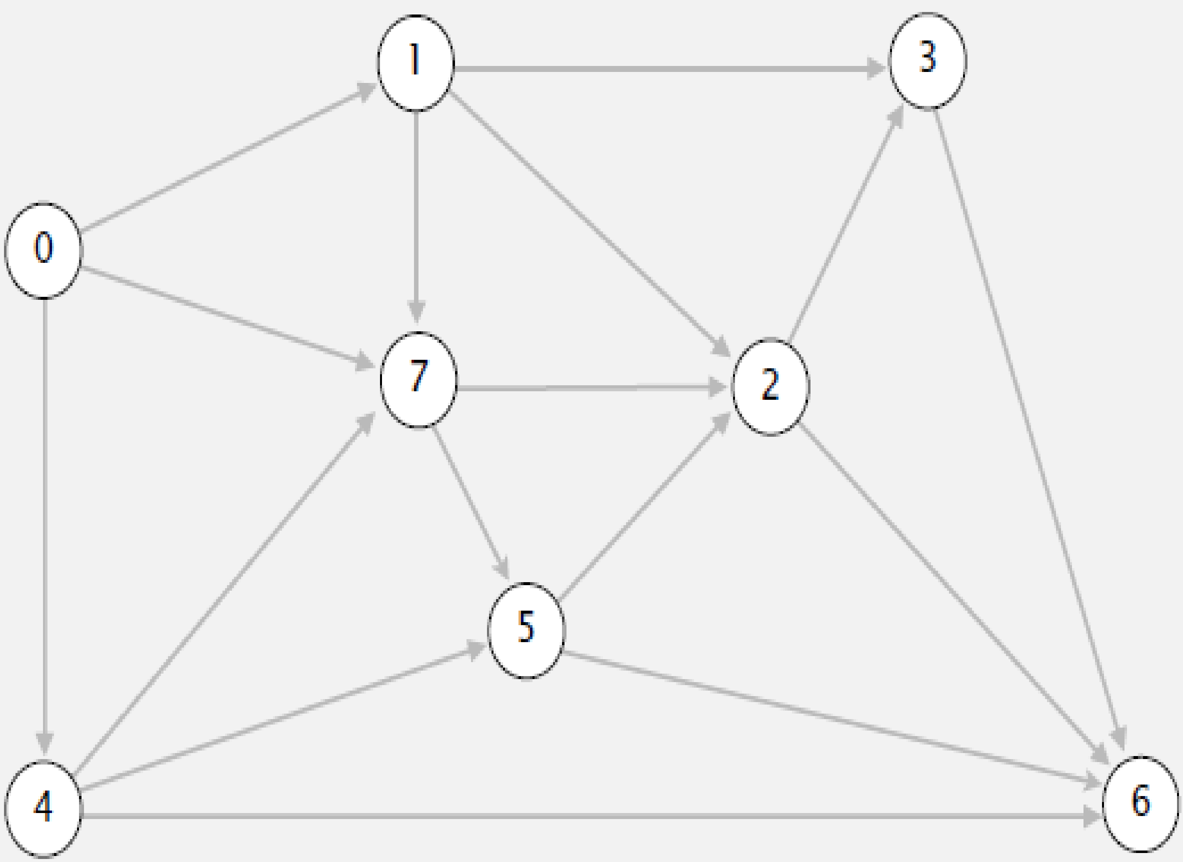
uni-bh
A MARCA DA EDUCAÇÃO

Algoritmo de Dijkstra



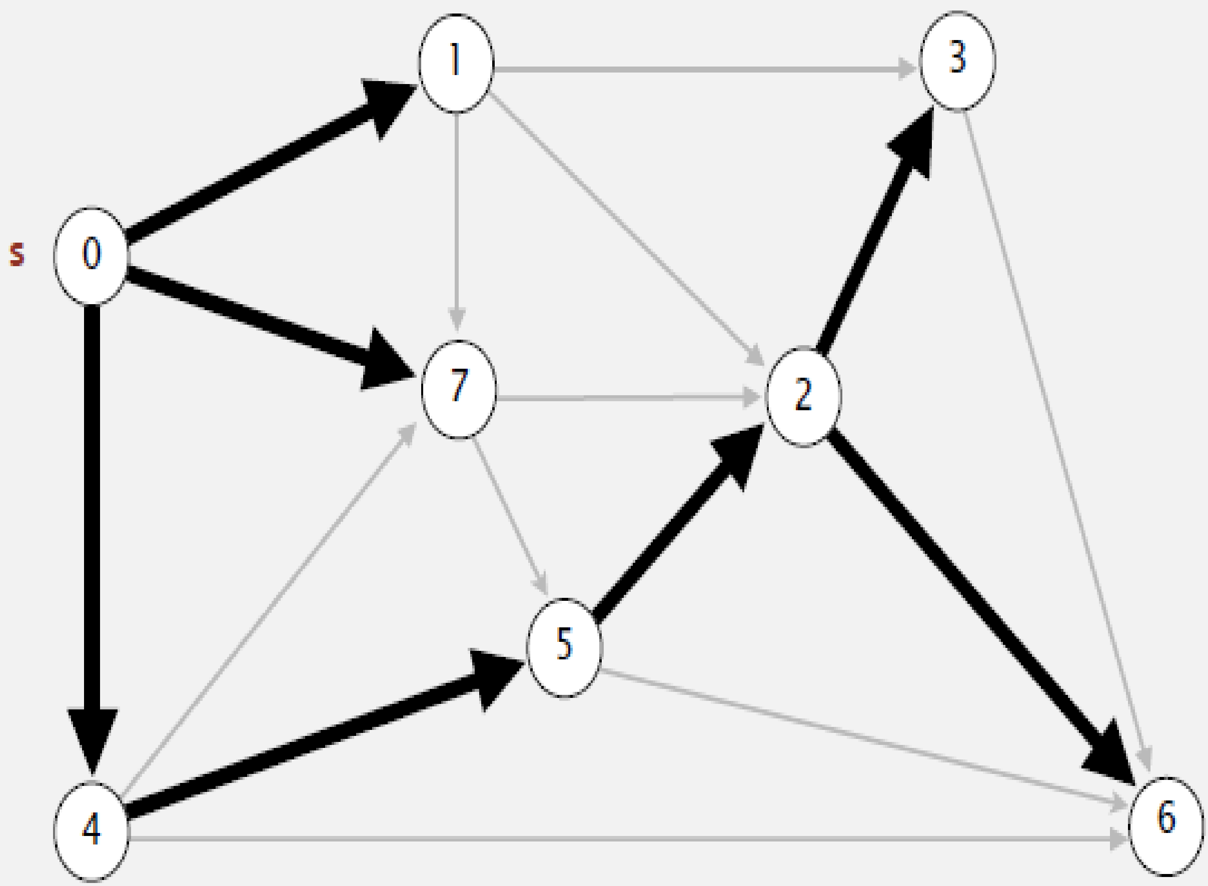
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
→ 6	25.0	2→6
7	8.0	0→7

Algoritmo de Dijkstra



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Algoritmo de Dijkstra



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Algoritmo de Dijkstra

- Proposição: O Algoritmo de Djikstra calcula o caminho mínimo em qualquer digrafo com pesos positivos.
- Prova:
  – Cada aresta $e = v \rightarrow w$ é relaxada exatamente uma vez(quando v é relaxada).

Algoritmo de Dijkstra

- Prova (continuação):
  - A condição de desigualdade é atendida até que o algoritmo termine porque:
    - distTo[w] não pode aumentar
      - O valor poderá diminuir a cada iteração.
    - distTo[v] não muda
      - As arestas são ponderadas e positivas, e a menor aresta é o menor distTo [] é escolhido a cada passo.
  - Desta maneira, até o término, as condições de otimalidade do menor caminho são mantidas.

Algoritmo de Dijkstra

```
public class DijkstraSP
{
   private DirectedEdge[] edgeTo;
   private double[] distTo;
   private IndexMinPQ<Double> pq;

   public DijkstraSP(EdgeWeightedDigraph G, int s)
   {
      edgeTo = new DirectedEdge[G.V()];
      distTo = new double[G.V()];
      pq = new IndexMinPQ<Double>(G.V());

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

      pq.insert(s, 0.0);
      while (!pq.isEmpty())
      {
          int v = pq.delMin();
          for (DirectedEdge e : G.adj(v))
             relax(e);
      }
   }
}
```

Relaxa os vértices em ordem de distância de s

Algoritmo de Dijkstra

```
private void relax(DirectedEdge e)
{
   int v = e.from(), w = e.to();
   if (distTo[w] > distTo[v] + e.weight())
   {
       distTo[w] = distTo[v] + e.weight();
       edgeTo[w] = e;
       if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
       else                pq.insert     (w, distTo[w]);
   }
}
```

Algoritmo de Djikstra

- Análise de Complexidade:
  - Em linhas gerais:
    - Implementação de array ótima para grafos densos.
    - Heap binário muito mais rápido para grafos esparsos.
    - D-way Heap compensa o esforço de implementação em situações onde a performance é crítica.
    - Heap de Fibonacci é o melhor na teoria, mas não compensa implementar.

Algoritmo de Dijkstra

- Análise de complexidade

PQ implementation	insert	delete-min	decrease-key	total
array	1	V	1	V^2
binary heap	log V	log V	log V	E log V
d-way heap (Johnson 1975)	$d \log_d V$	$d \log_d V$	$\log_d V$	$E \log_{E/V} V$
Fibonacci heap (Fredman-Tarjan 1984)	1 †	log V †	1 †	E + V log V

† amortized

Redimensionamento de conteúdo

- Costura de escultura [Avidan and Shamir]: Redimensiona uma imagem sem distorção para exibir em celulares e navegadores web.

uni-bh
A MARCA DA EDUCAÇÃO

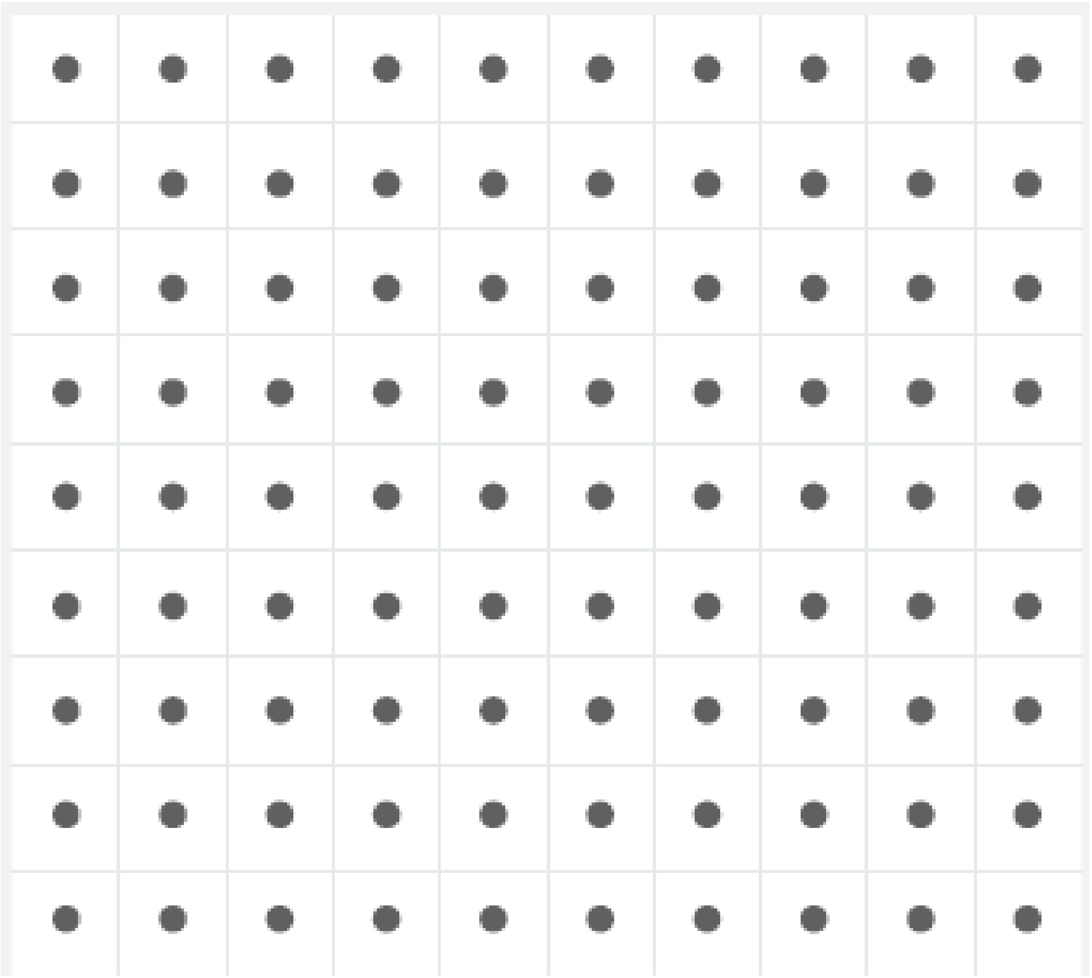
Redimensionamento de conteúdo

- Utilizado no Photoshop, Imagemagic, GIMP, …

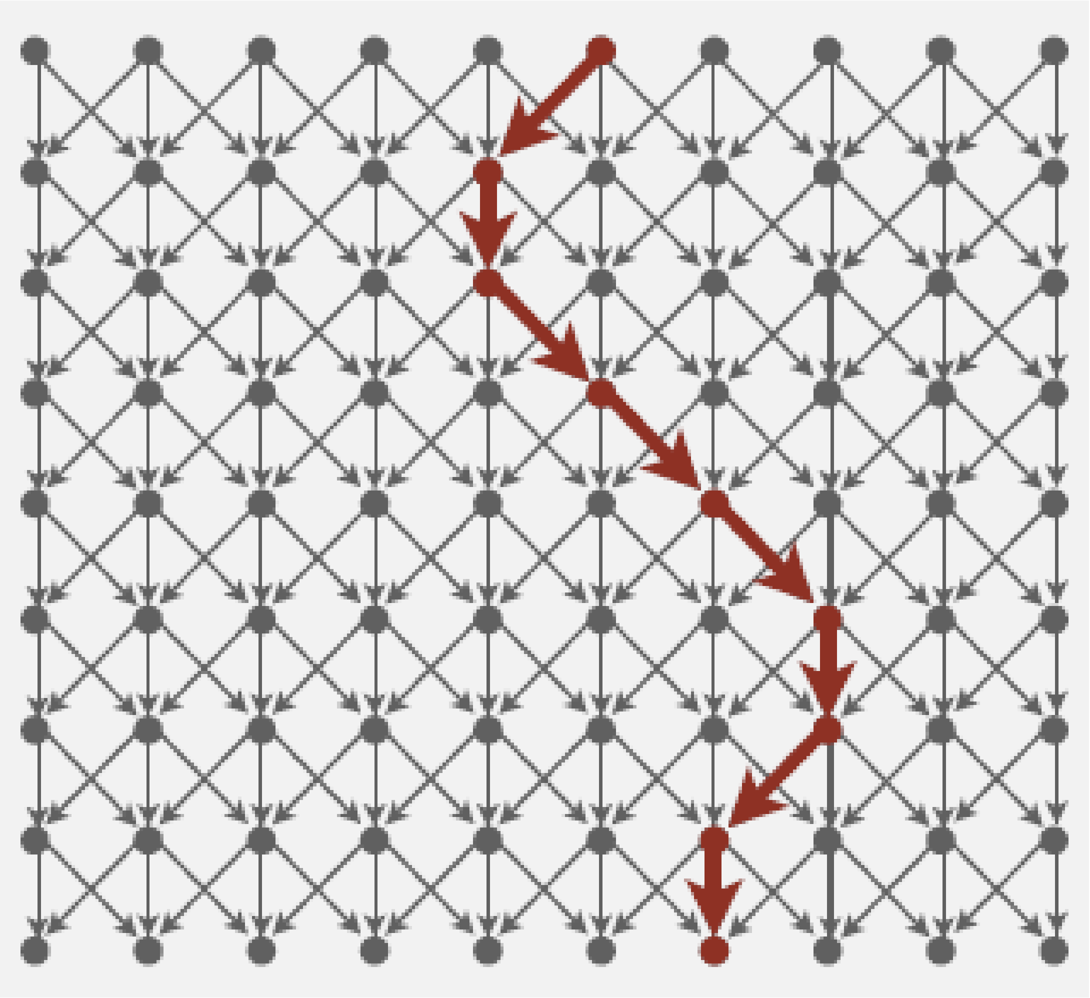
Redimensionamento de conteúdo

- Para encontrar a costura vertical:
  - Grid Grafo direcionado acíclico:
    - vértice = pixel,
    - Aresta = do pixel para os 3 vizinhos para baixo.
  - Peso do pixel: função de energia dos 8 pixels vizinhos
  - Costura = menor caminho do topo até a parte de baixo da figura

Redimensionamento de conteúdo

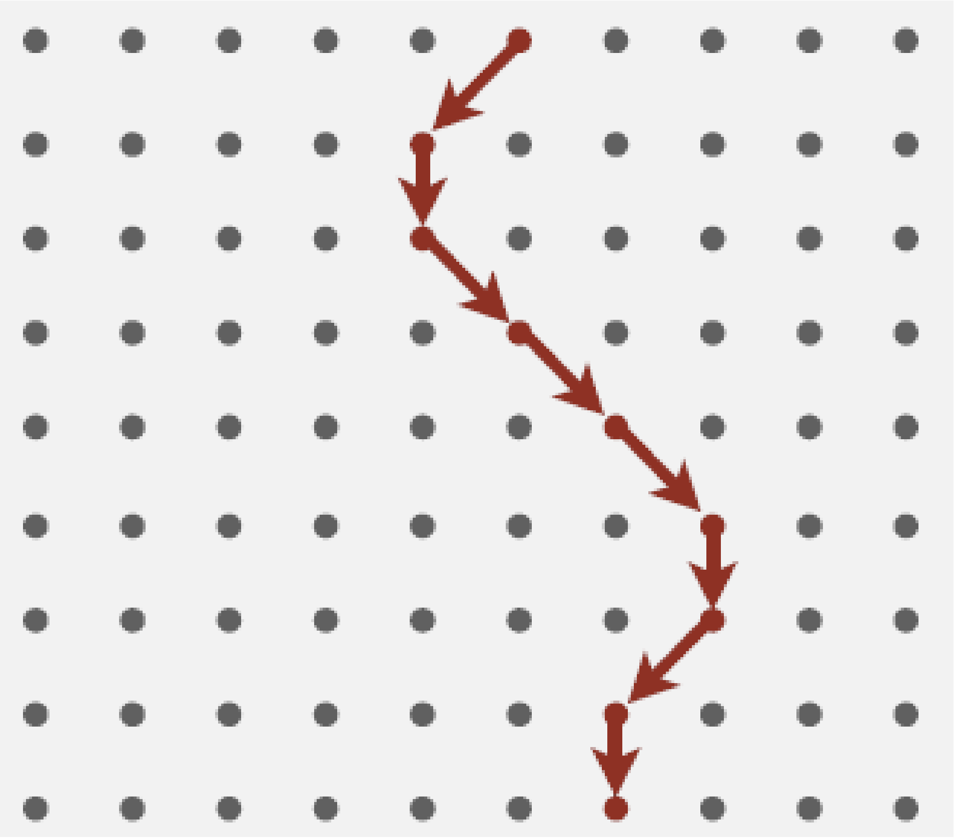


Redimensionamento de conteúdo



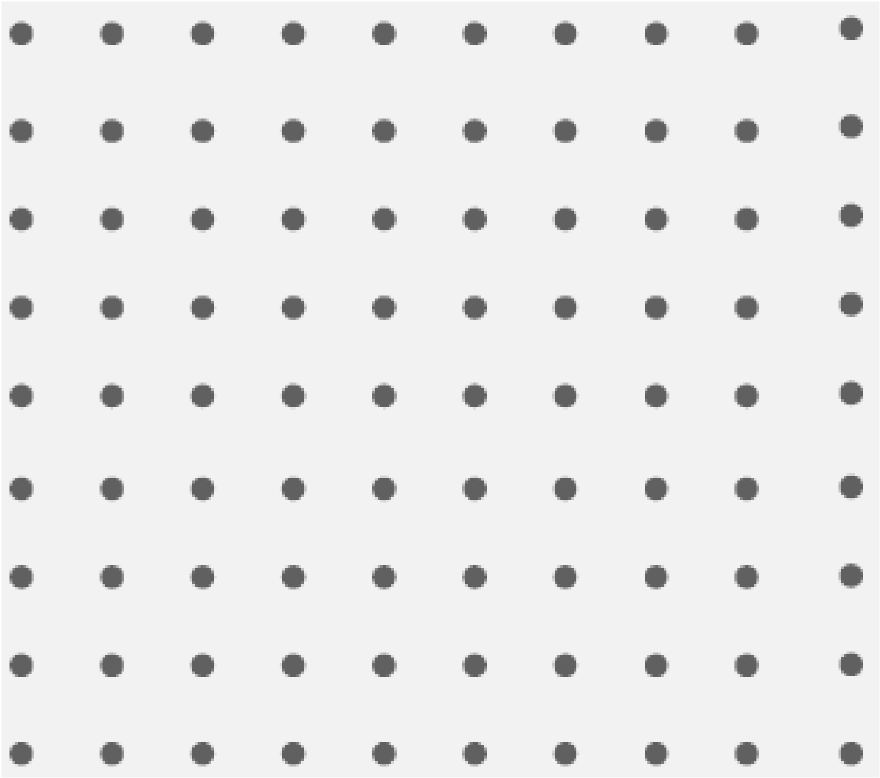
Redimensionamento de conteúdo

- Após encontrar o caminho, remova os vértices da costura (um por linha)



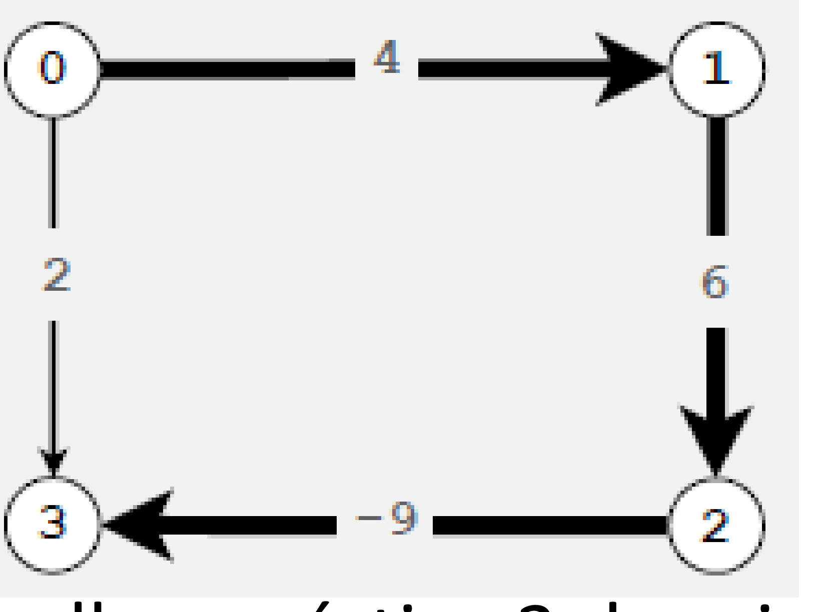
Redimensionamento de conteúdo

- Após encontrar o caminho, remova os vértices da costura (um por linha)



Caminhos mínimos com peso negativos

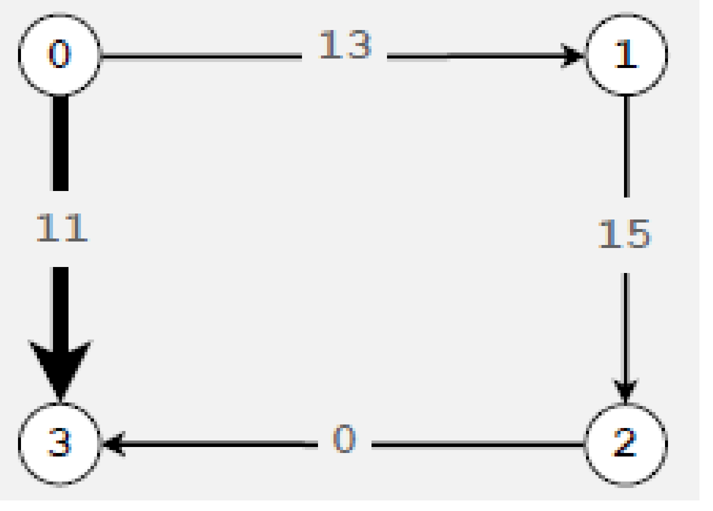
- Djikstra: Não funciona com arestas com peso negativo.



- Djikstra escolhe o vértice 3 depois do 0. Mas o caminho mínimo a partir de 0 seria 0, 1, 2, 3.

Caminhos mínimos com peso negativos

- Re-pesando: Adicionar um restrição para cada aresta não funciona.



- Adicionando 9 para cada aresta muda o menor caminho de 0, 1, 2, 3 para 0, 3.

Caminhos mínimos com peso negativos

- Más notícias:

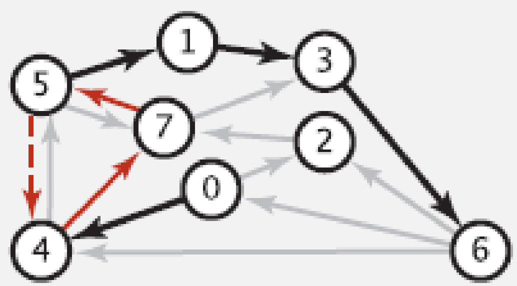
- Precisamos de um algoritmo diferente!

Ciclos negativos

- Definição: Um ciclo negativo é um ciclo direcionado cujo a soma das arestas é negativa.

digraph

4->5	0.35
5->4	-0.66
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93



negative cycle (-0.66 + 0.37 + 0.28)

5->4->7->5

shortest path from 0 to 6

0->4->7->5->4->7->5...->1->3->6

Ciclos negativos

- Proposição: Um caminho mínimo só existe se e somente se não existem ciclos negativos.
- Assumindo que todos os vértices são alcançáveis através da origem.

Algoritmo de Bellman-Ford

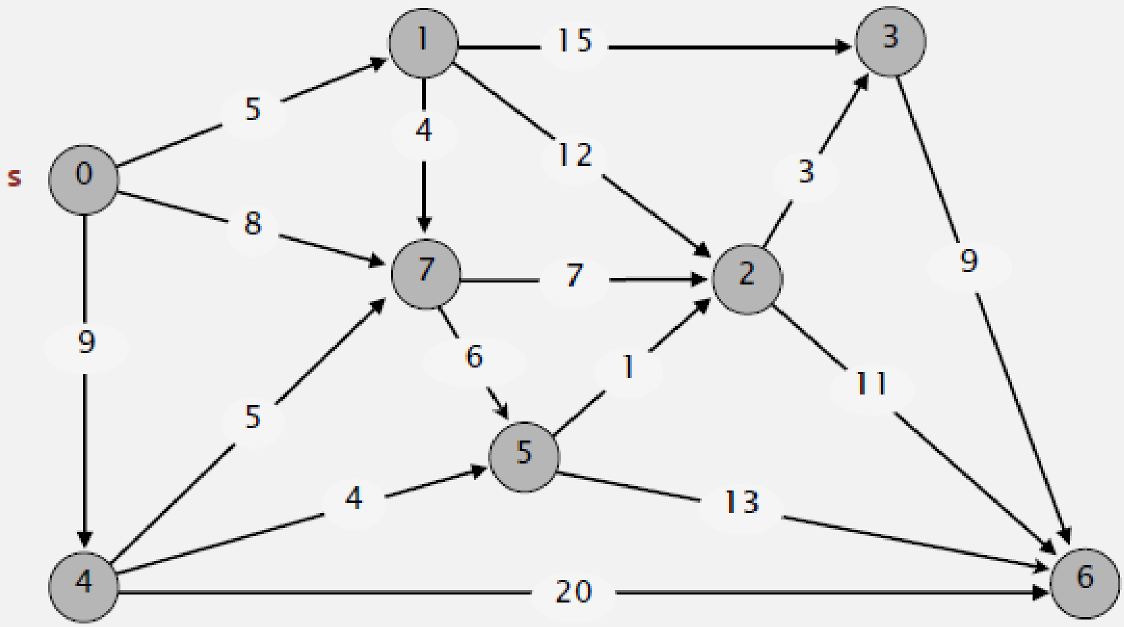
- Inicialize $distTo[s] = 0$ e $distTo[v] = \infty$ para todos os demais vértices.
- Repita V vezes:
  – Relaxe cada aresta.

```
for (int i = 0; i < G.V(); i++)
   for (int v = 0; v < G.V(); v++)
      for (DirectedEdge e : G.adj(v))
         relax(e);
```

Algoritmo de Bellman-Ford

- Inicialize $distTo[s] = 0$ e $distTo[v] = \infty$ para todos os demais vértices.
- Repita V vezes:
  - Relaxe cada aresta.

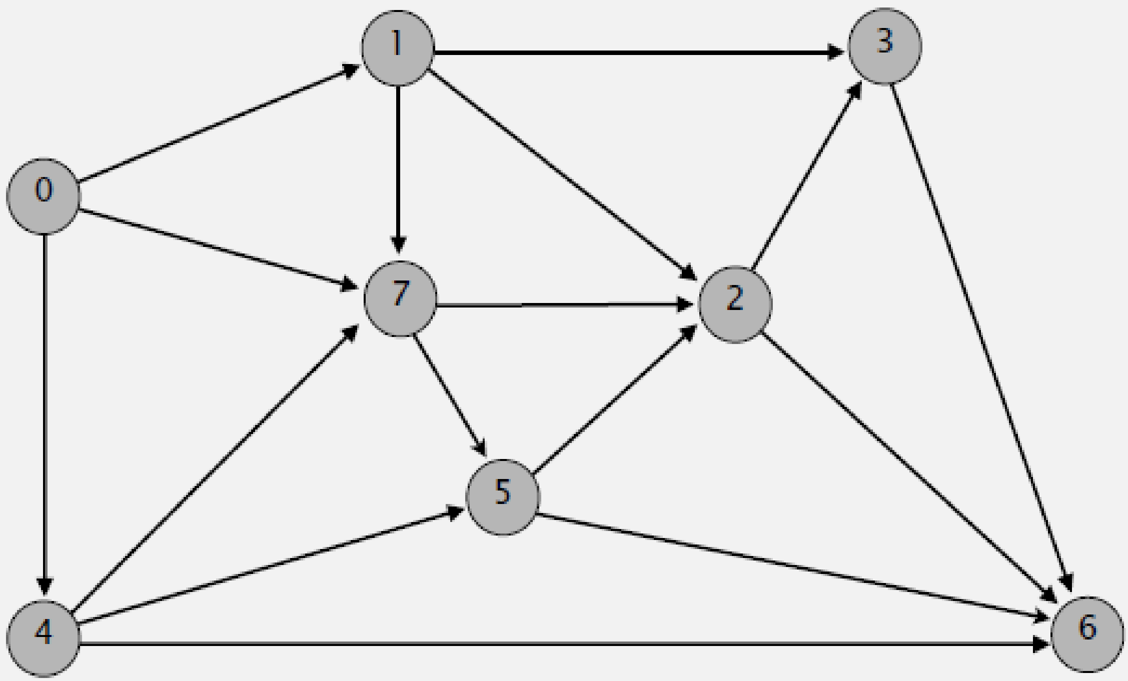
Algoritmo de Bellman-Ford



an edge-weighted digraph

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

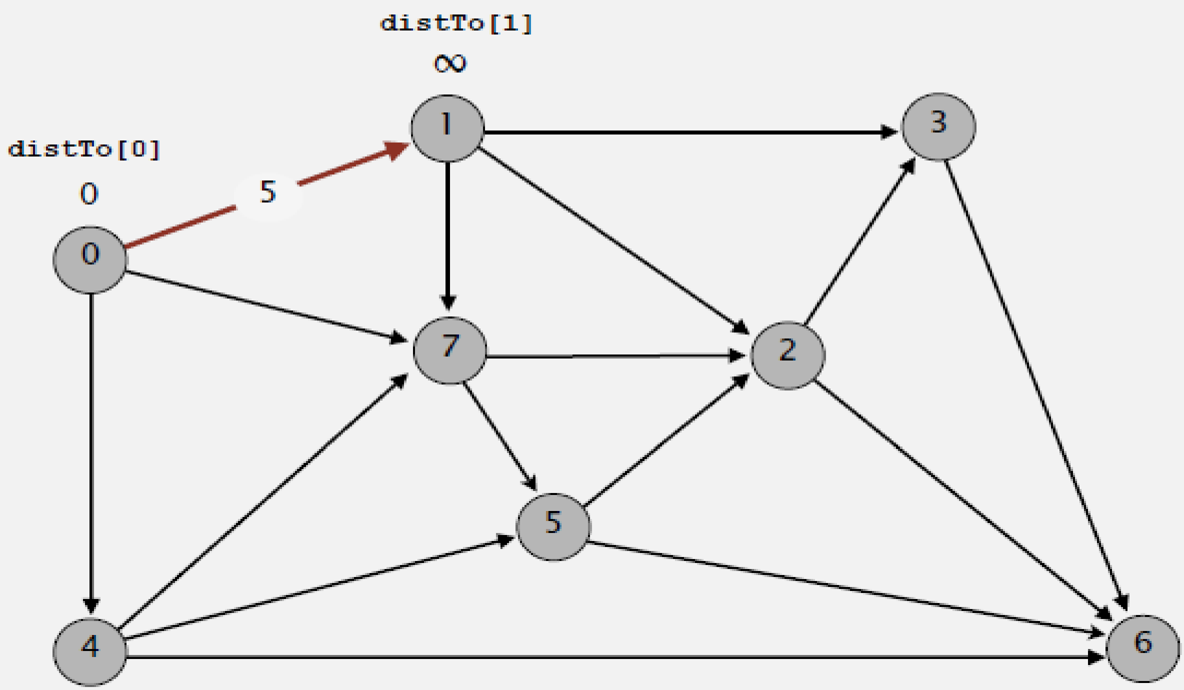
Algoritmo de Bellman-Ford



v	distTo[]	edgeTo[]
0	0.0	-
1		
2		
3		
4		
5		
6		
7		

initialize

Algoritmo de Bellman-Ford

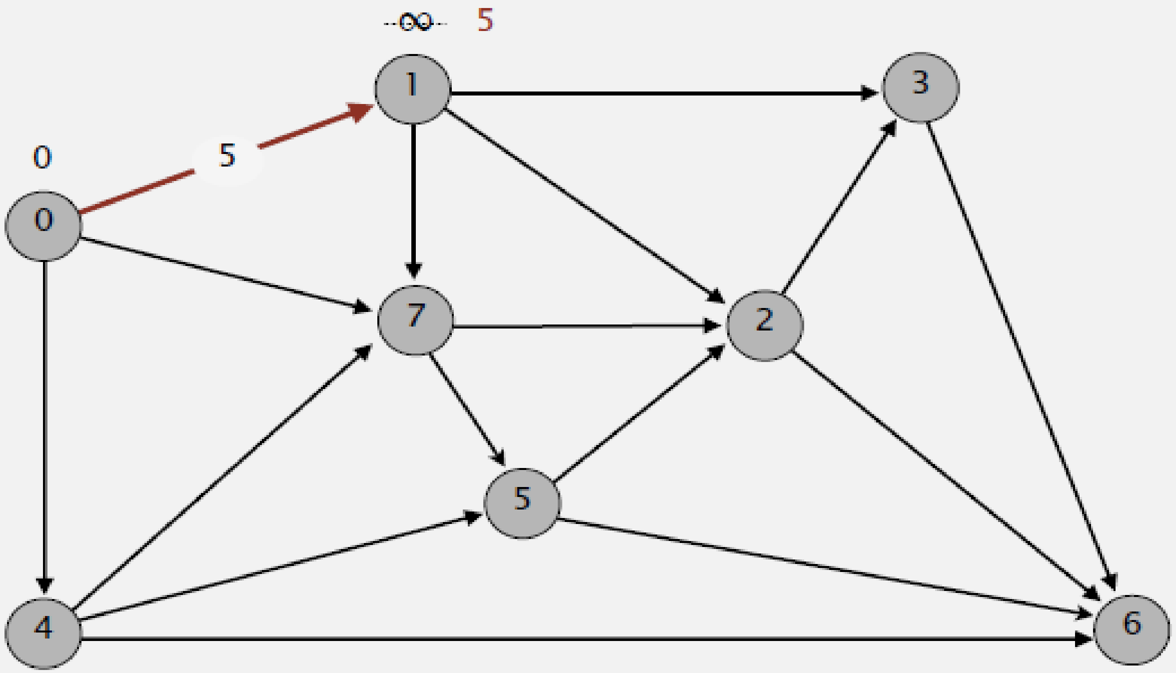


v	distTo[]	edgeTo[]
0	0.0	–
1		
2		
3		
4		
5		
6		
7		

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

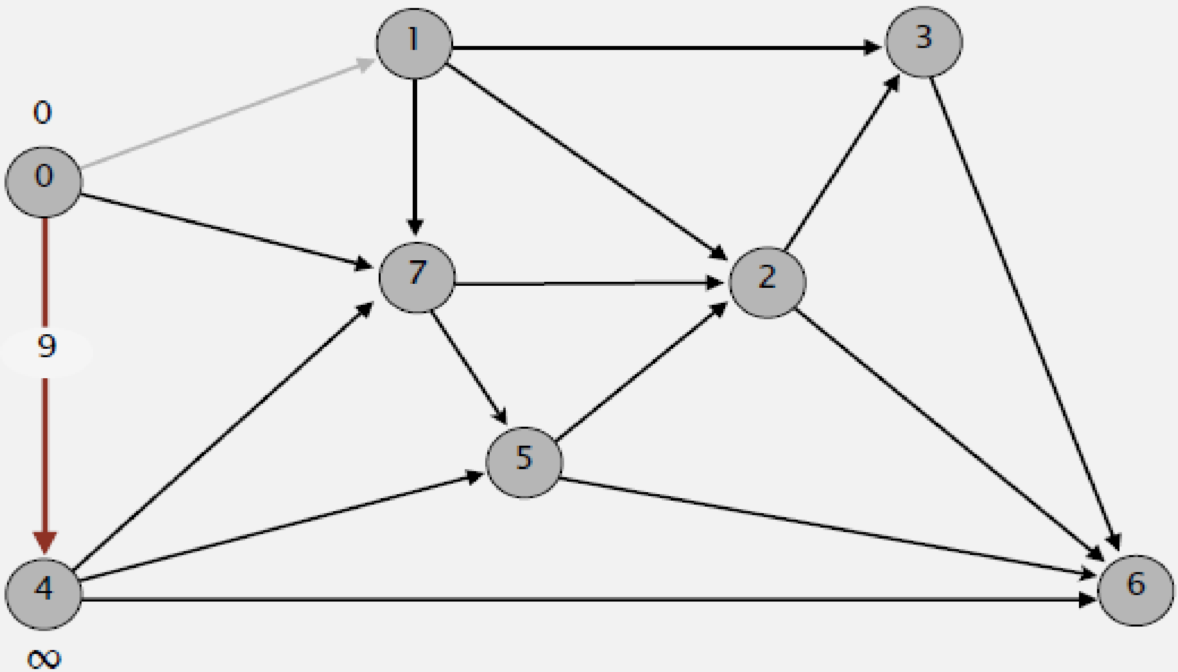


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4		
5		
6		
7		

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

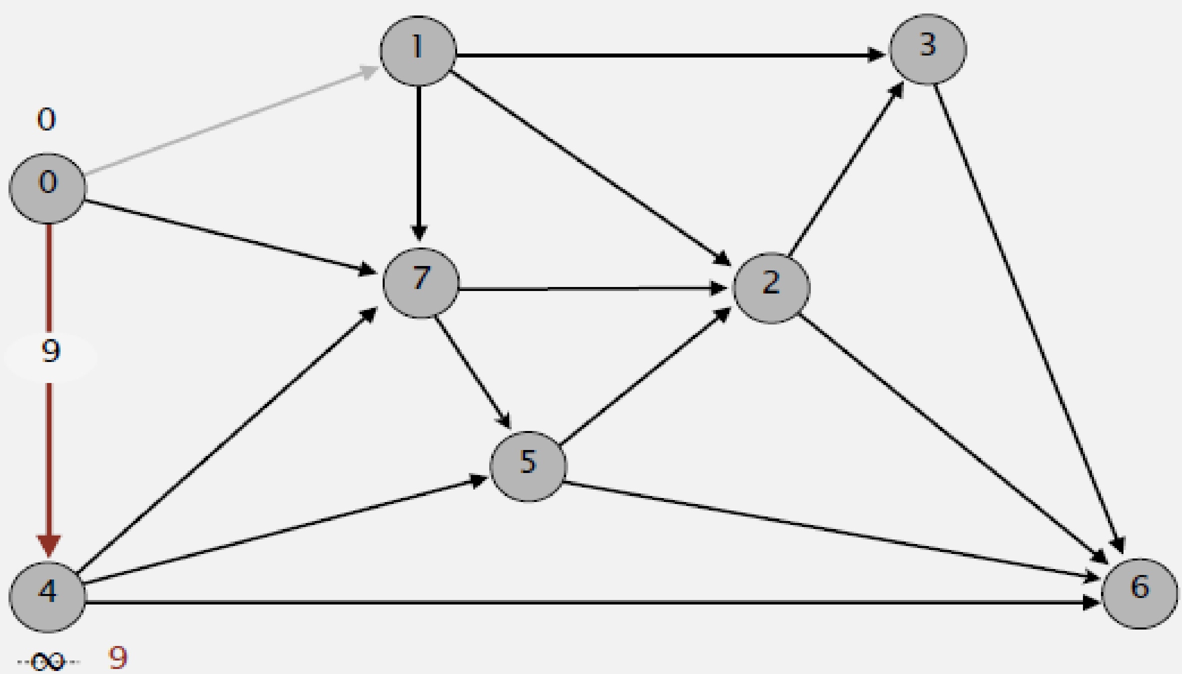


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4		
5		
6		
7		

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

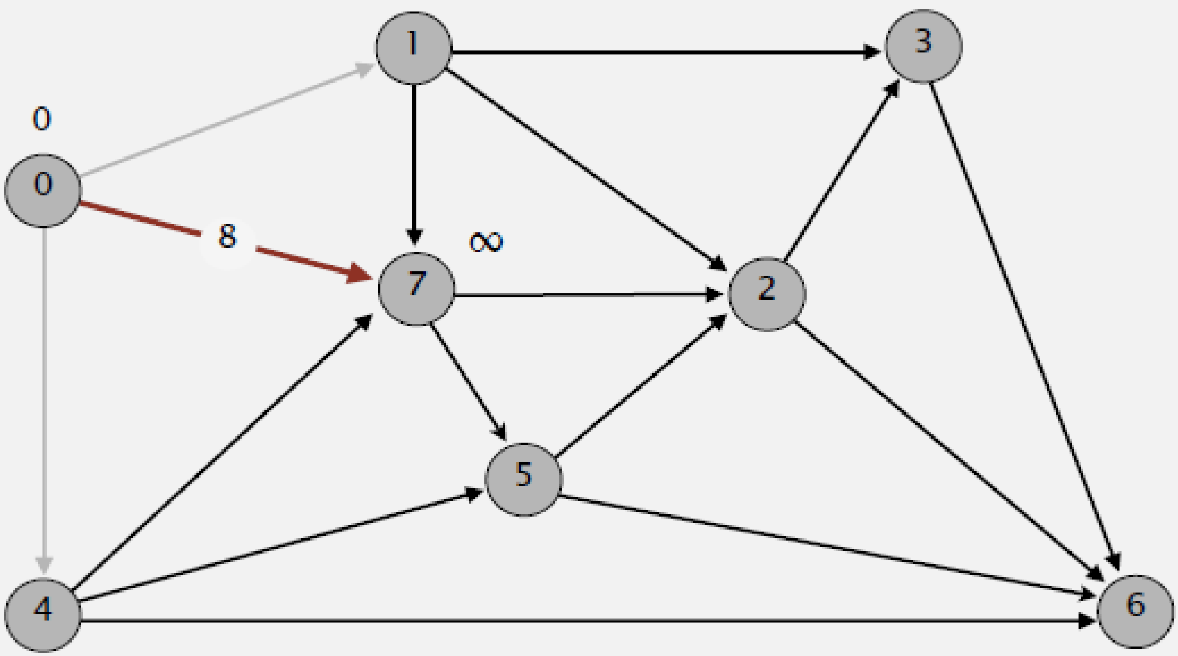


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7		

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

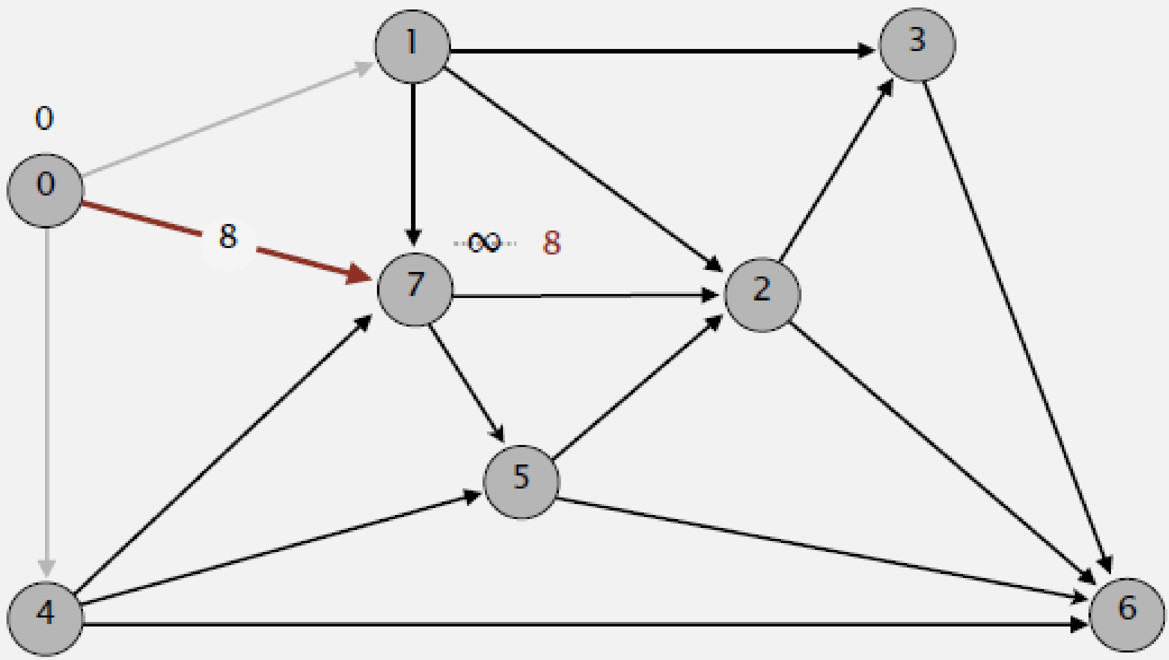


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7		

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

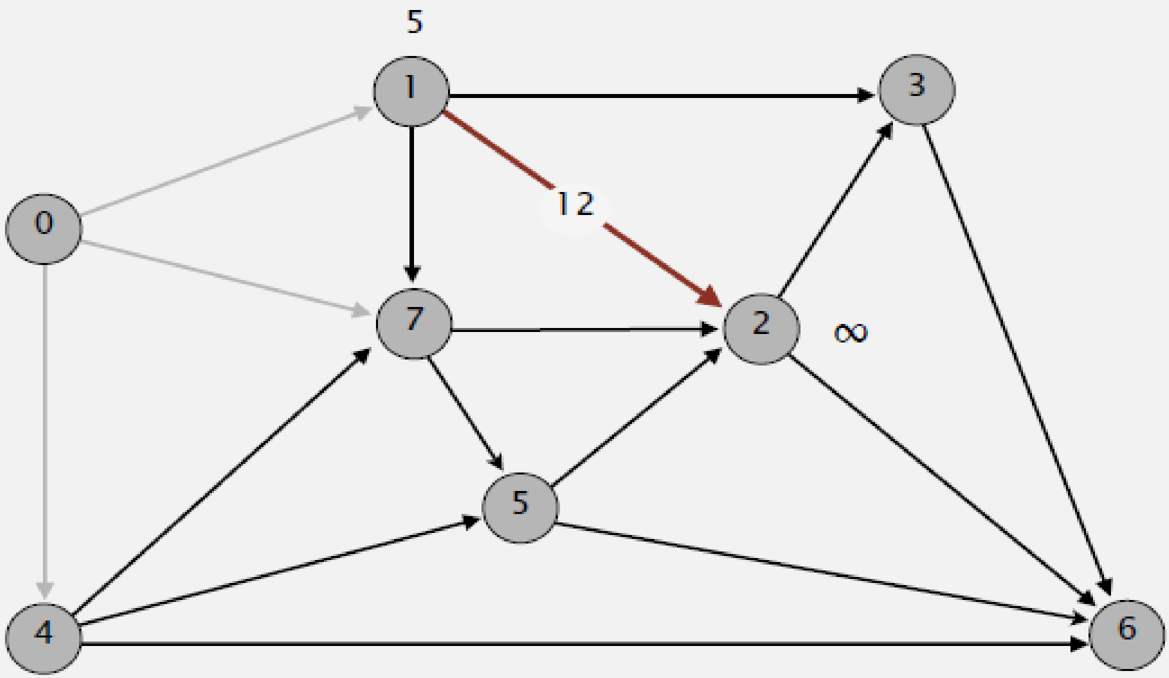


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

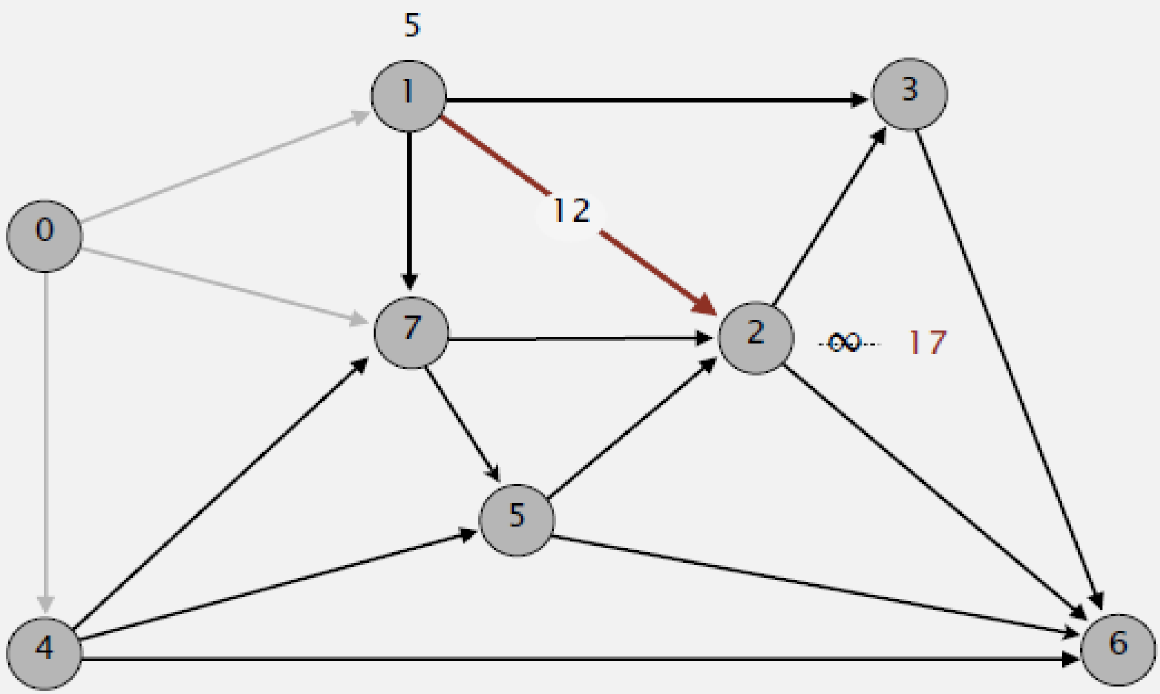


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

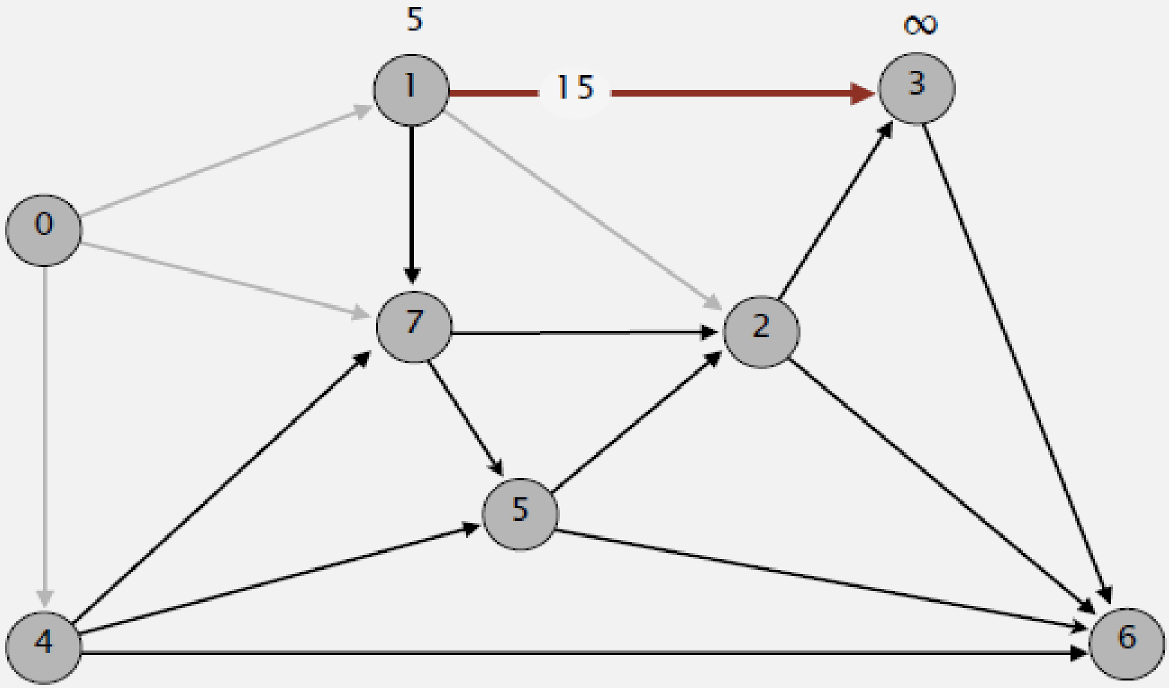


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

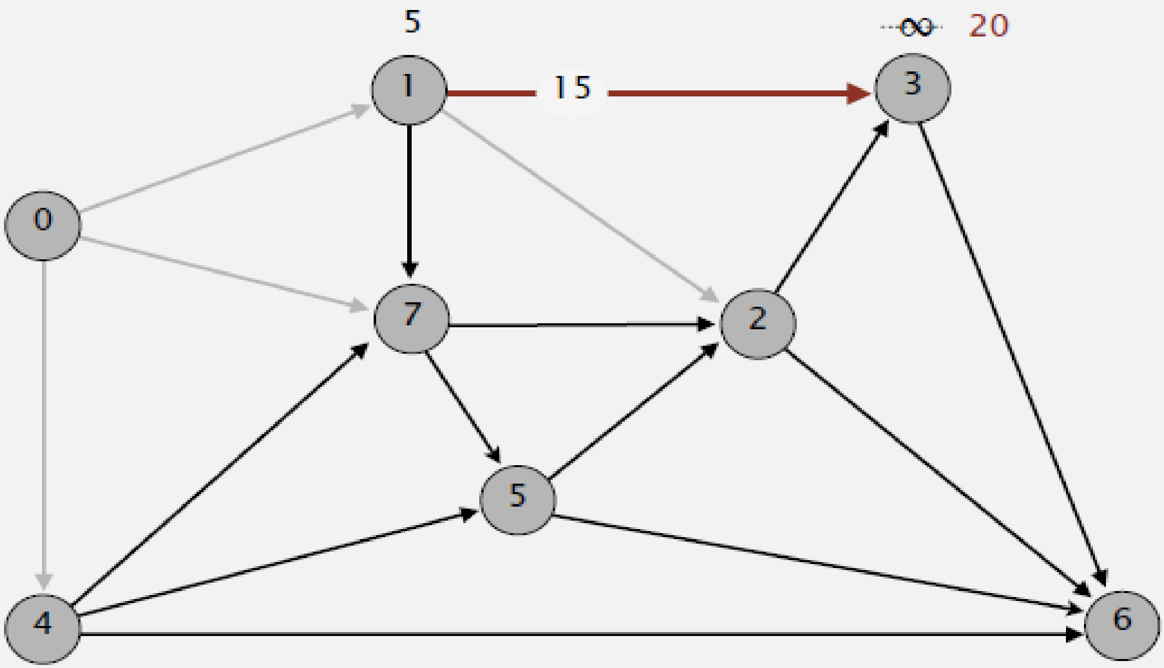


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

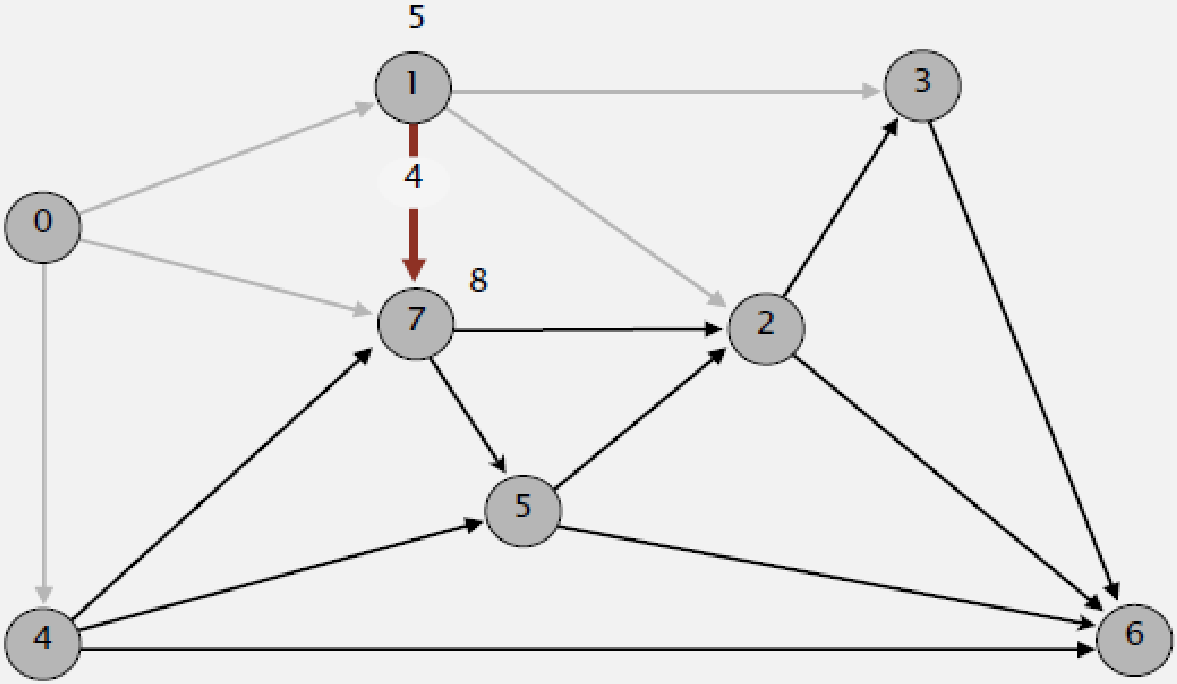


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

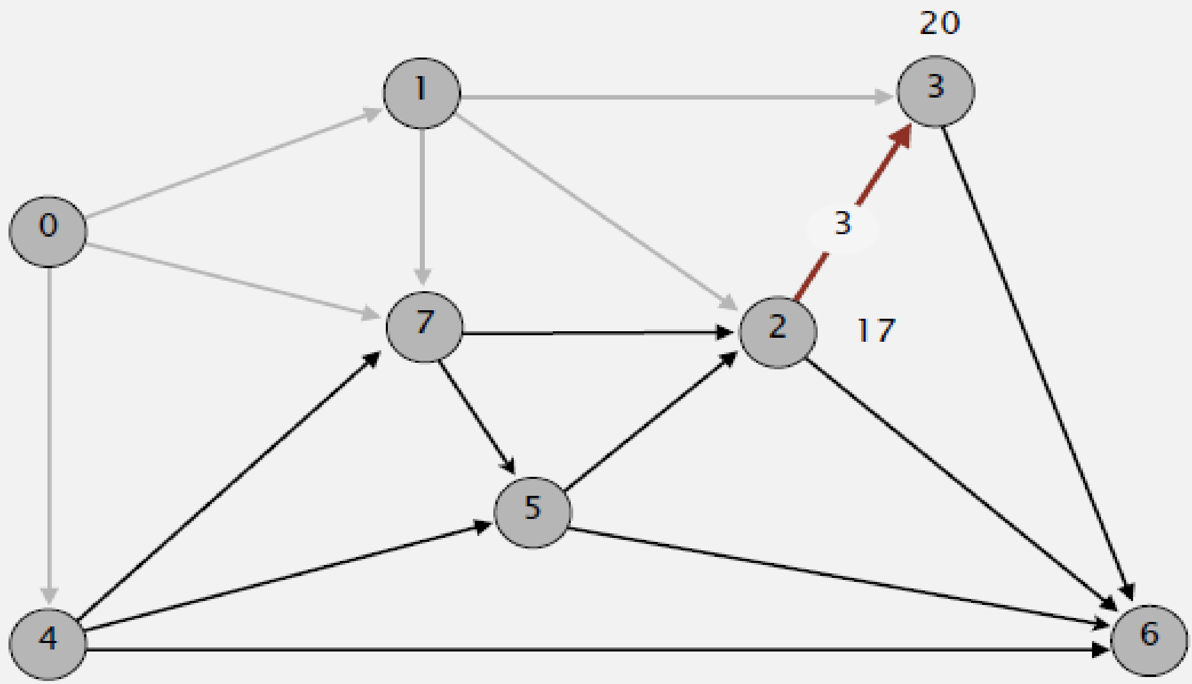


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

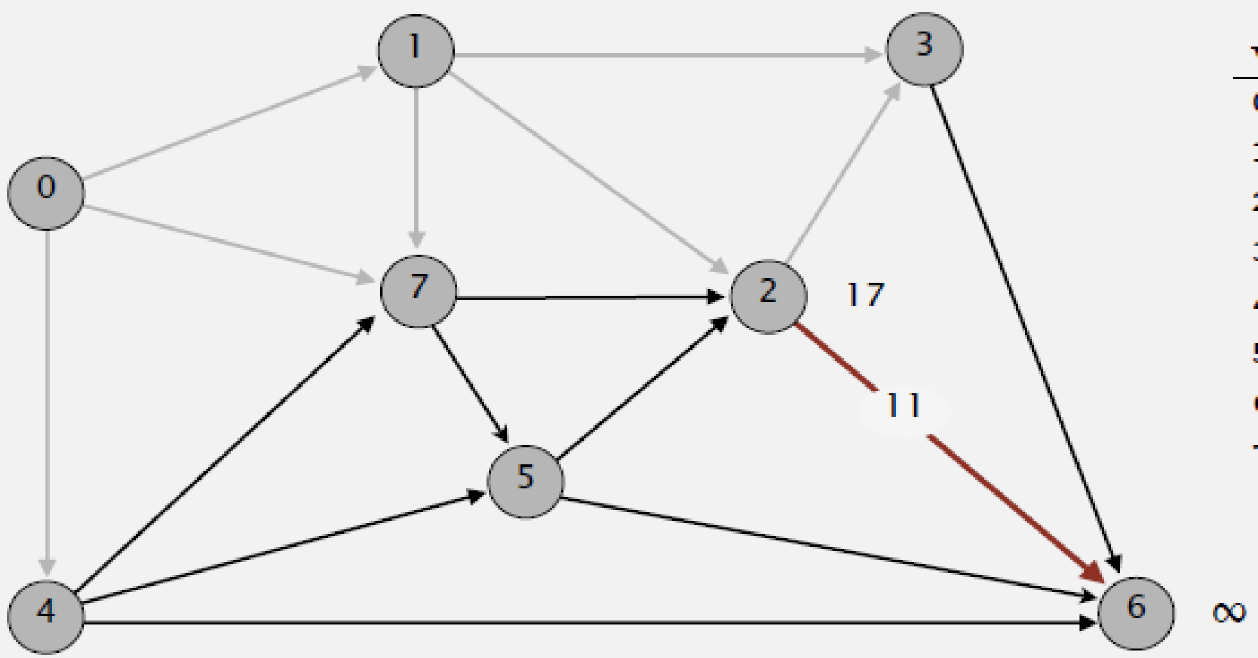


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

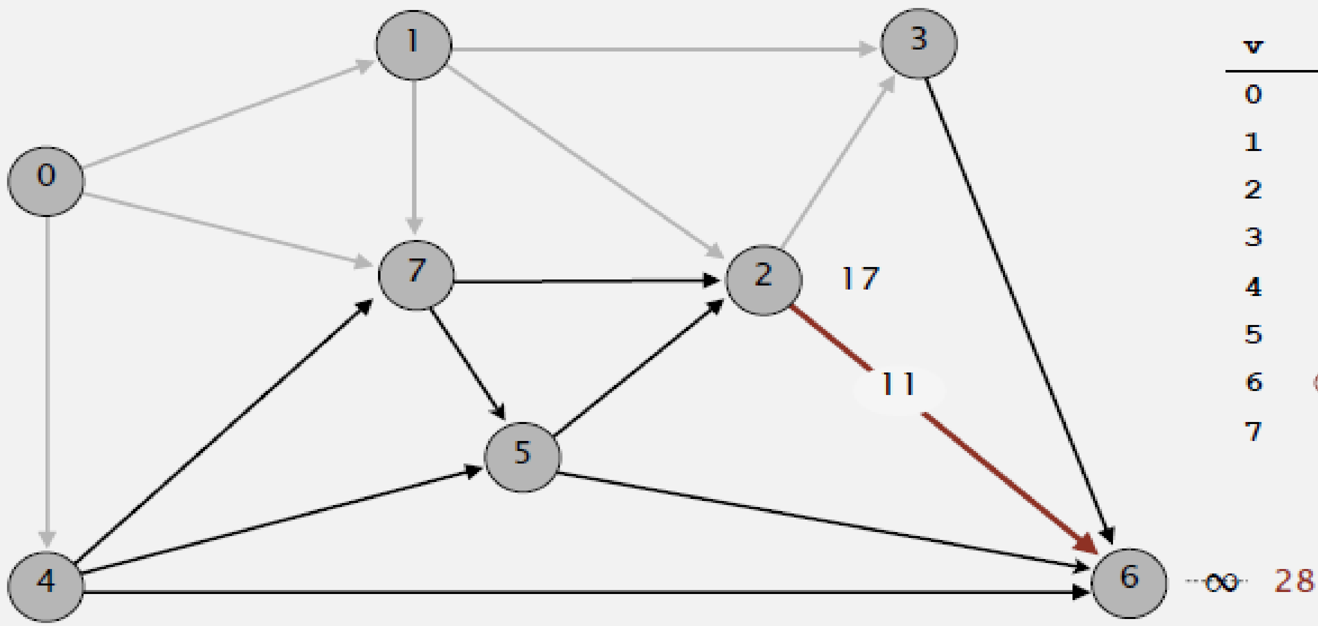


v	distTo[]	edgeTo[]
0	0.0	–
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

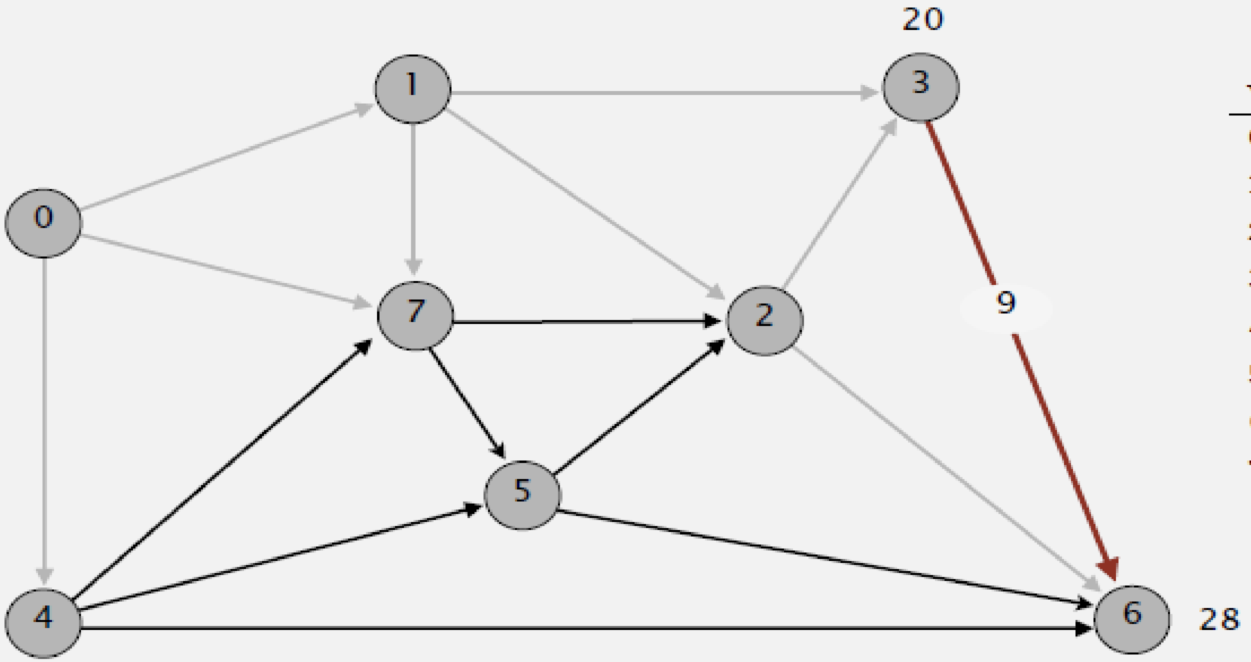


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6	28.0	2→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford



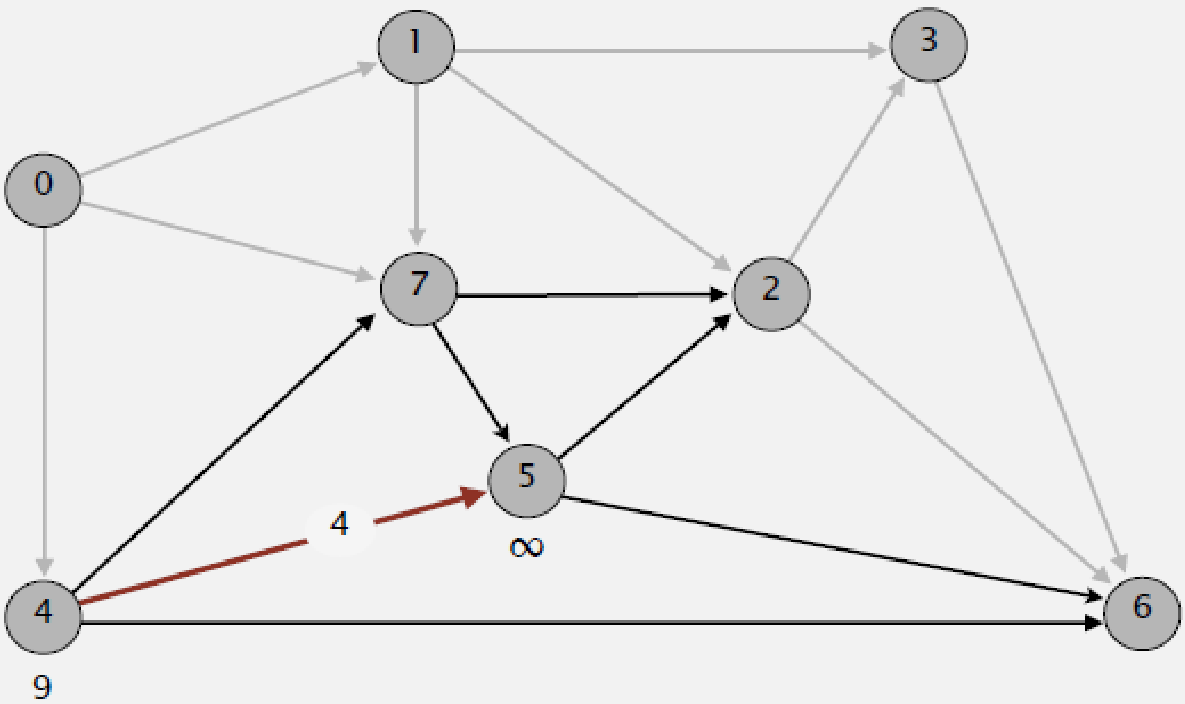
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6	28.0	2→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Algoritmo de Bellman-Ford

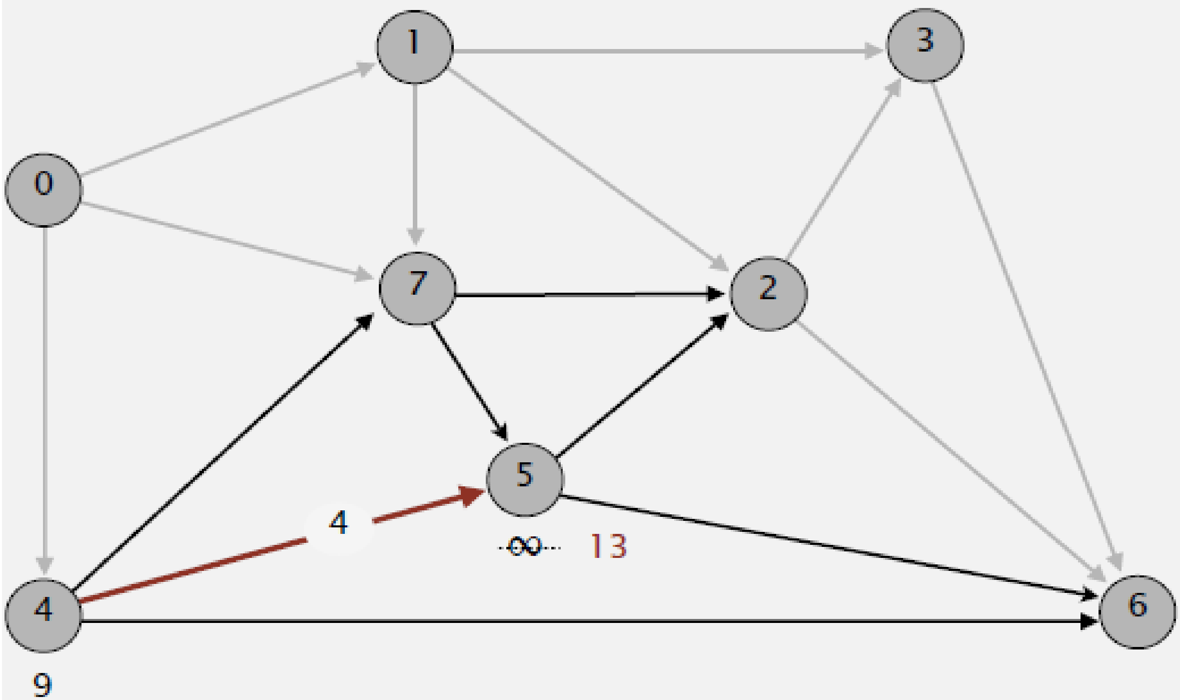


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6	28.0	2→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

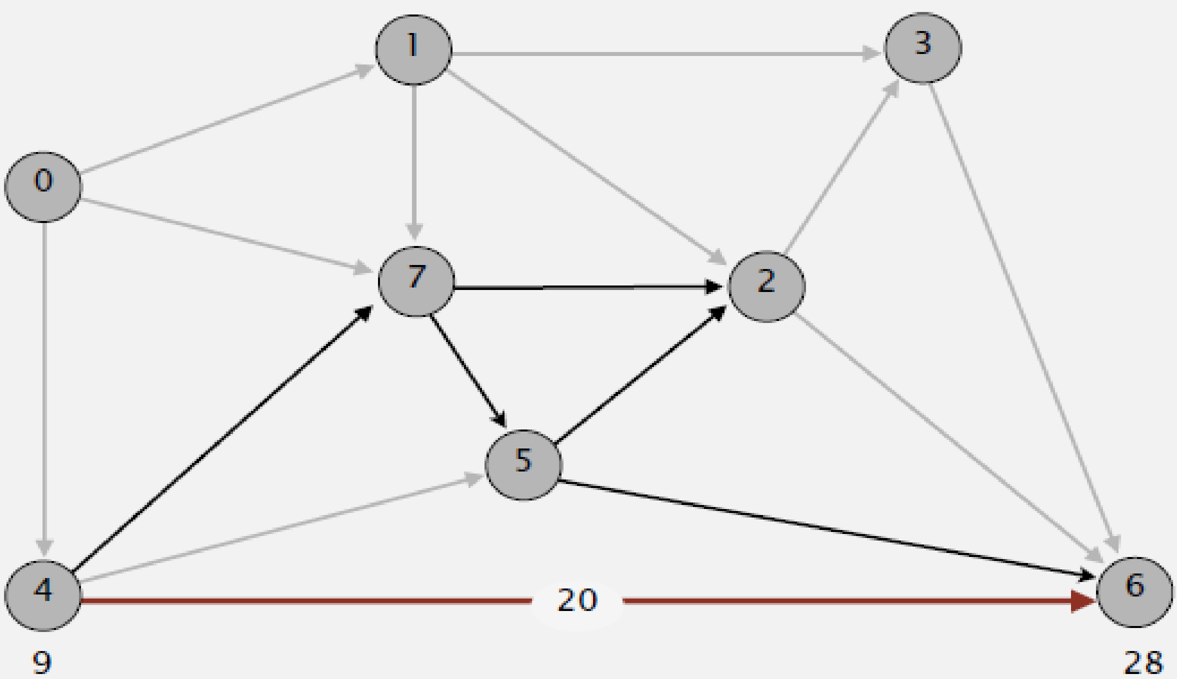


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

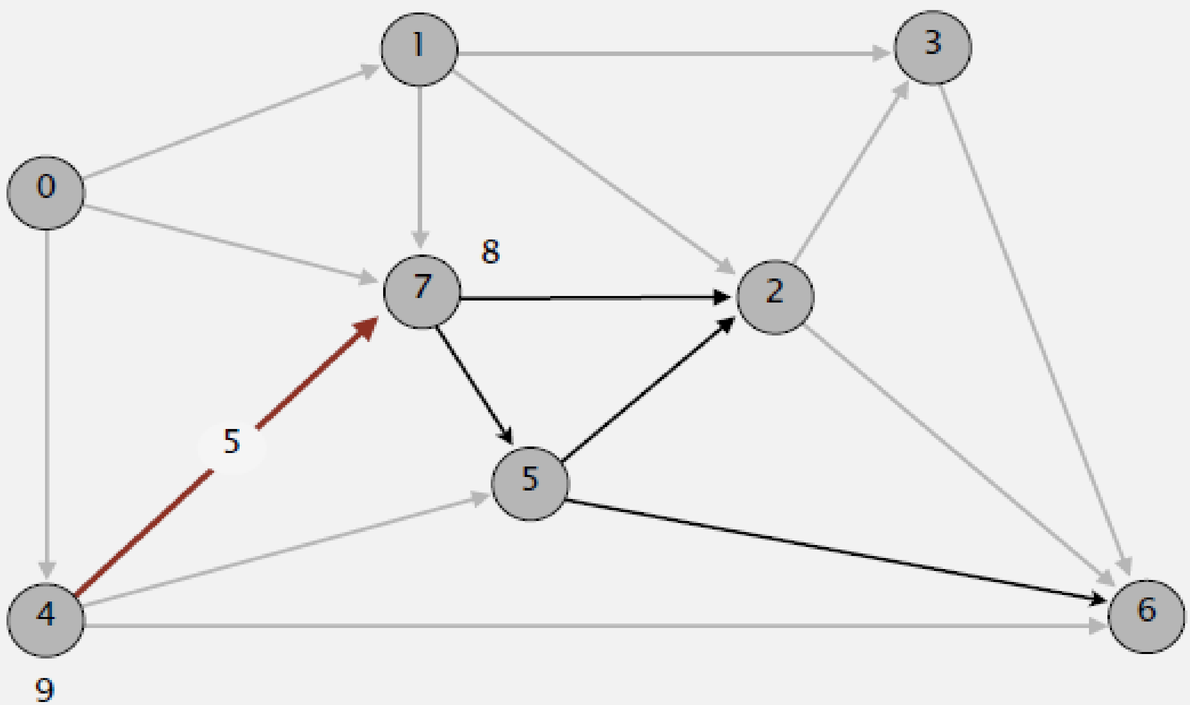


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

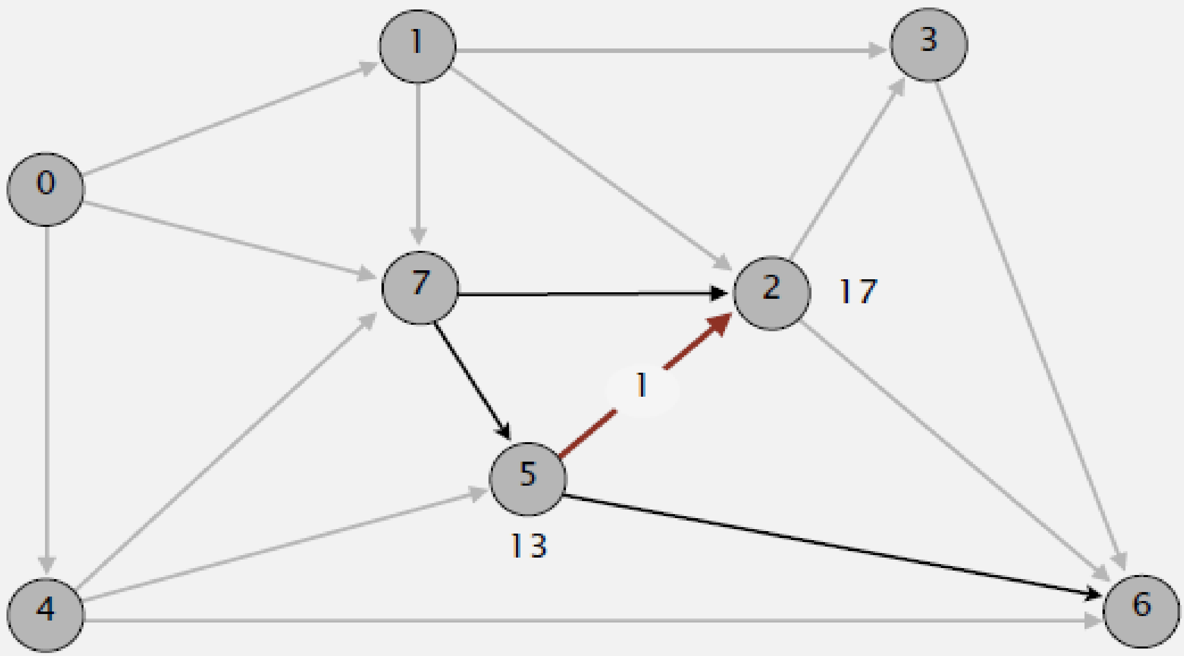


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

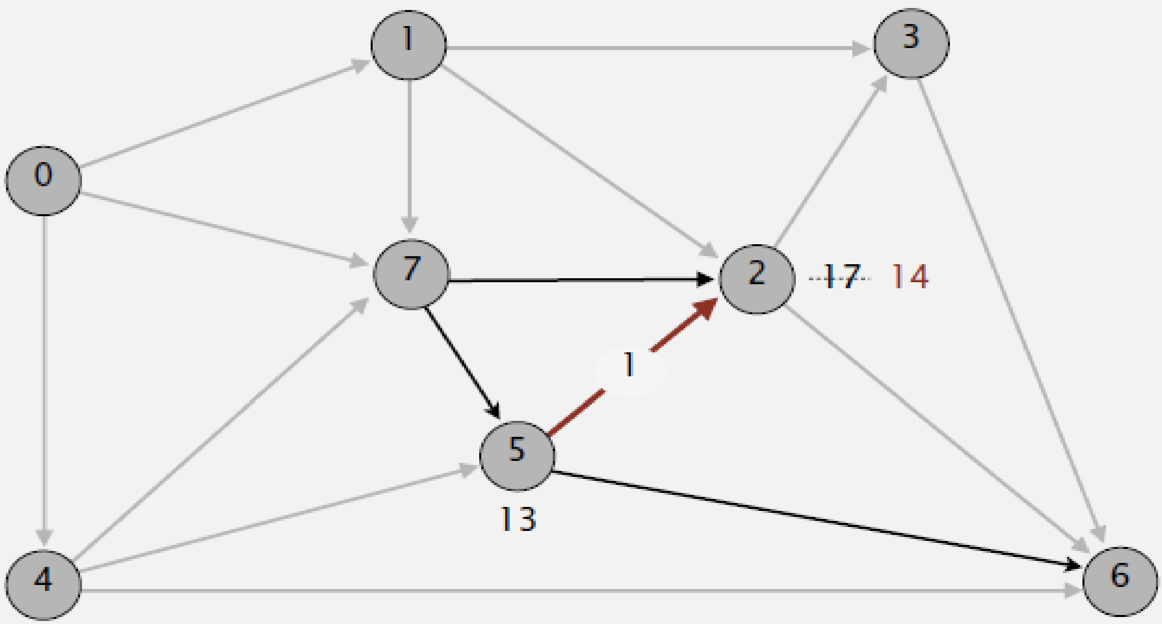


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

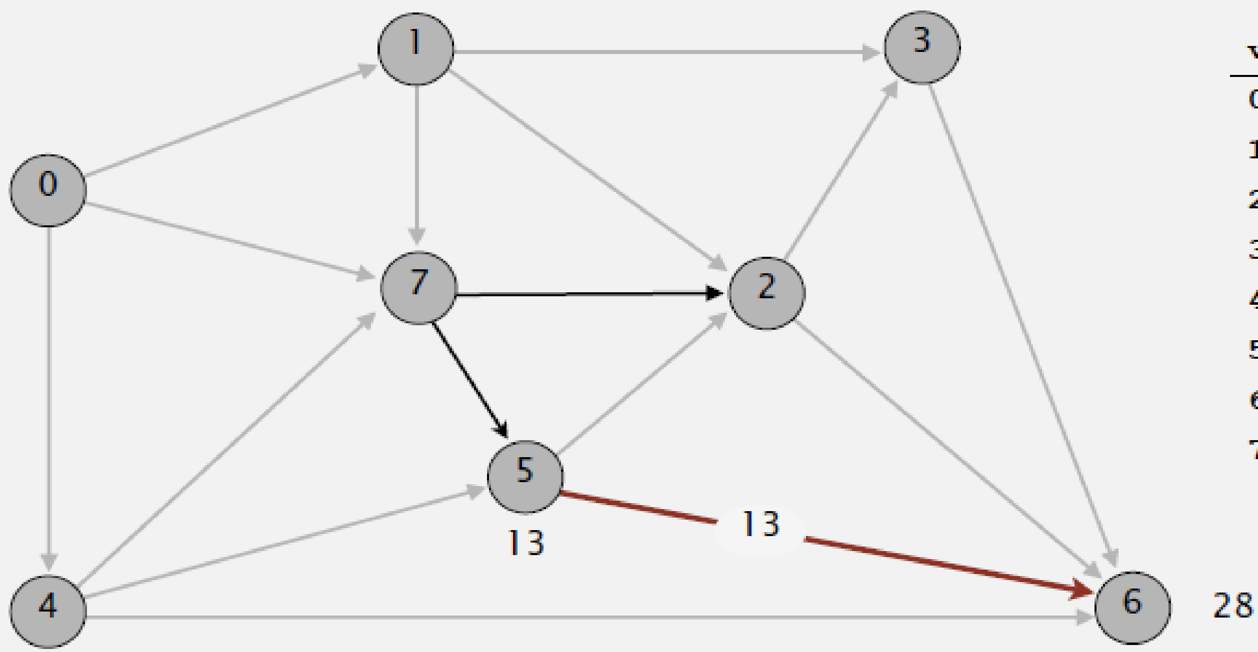


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

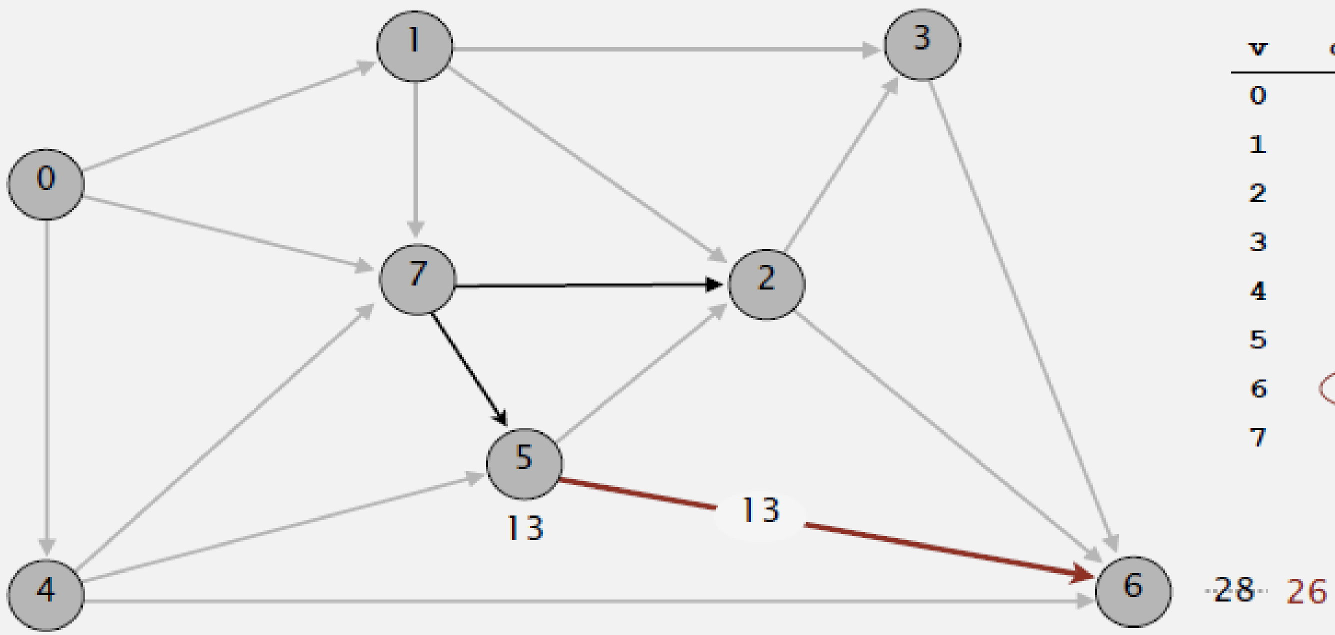


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

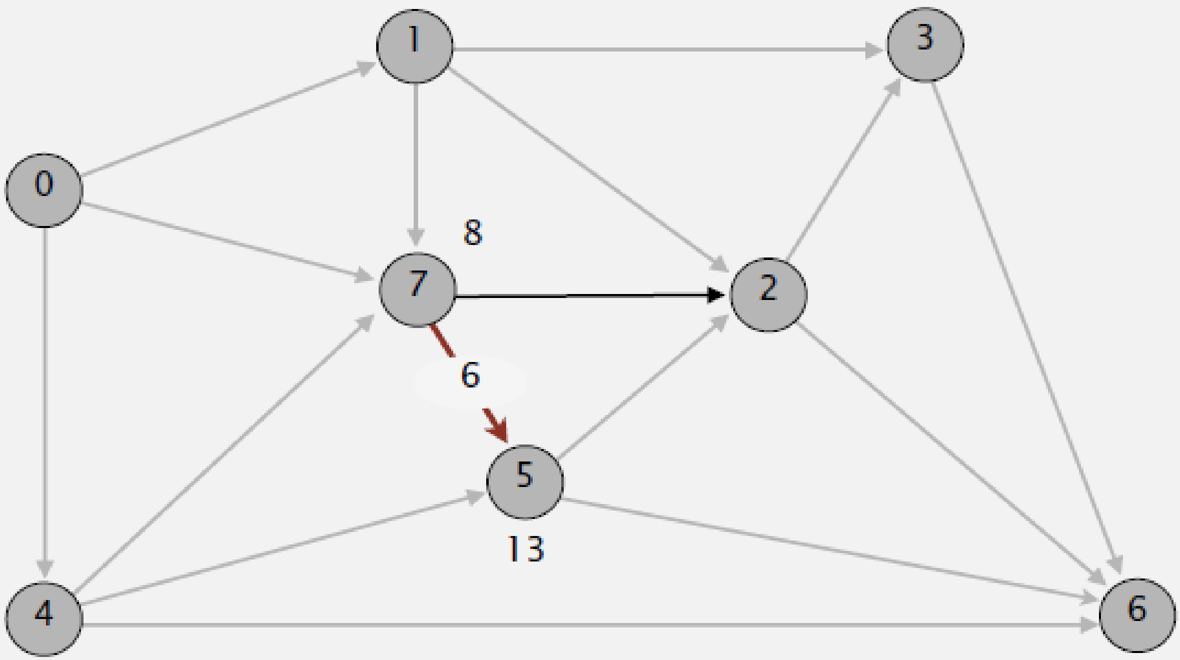


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

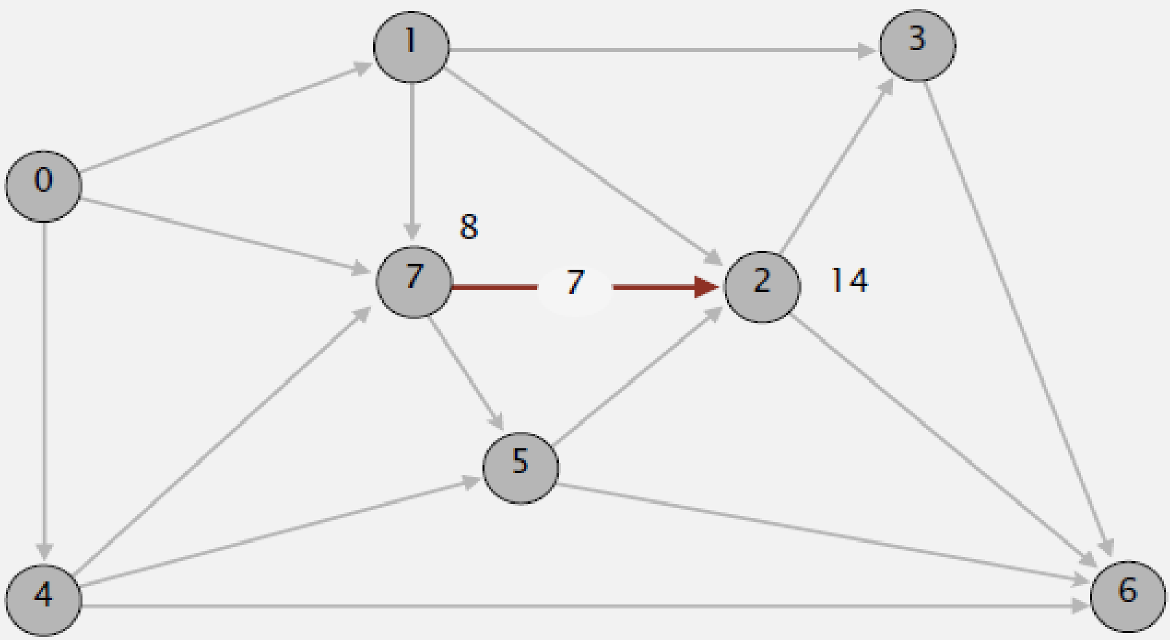


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

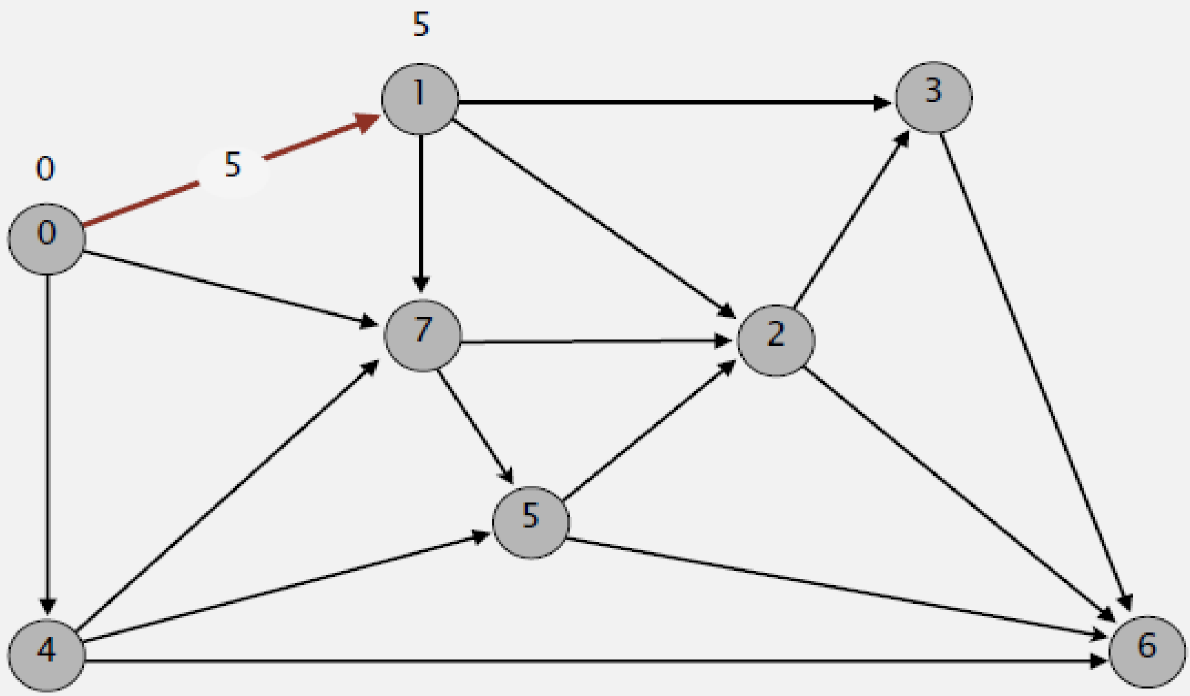


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford



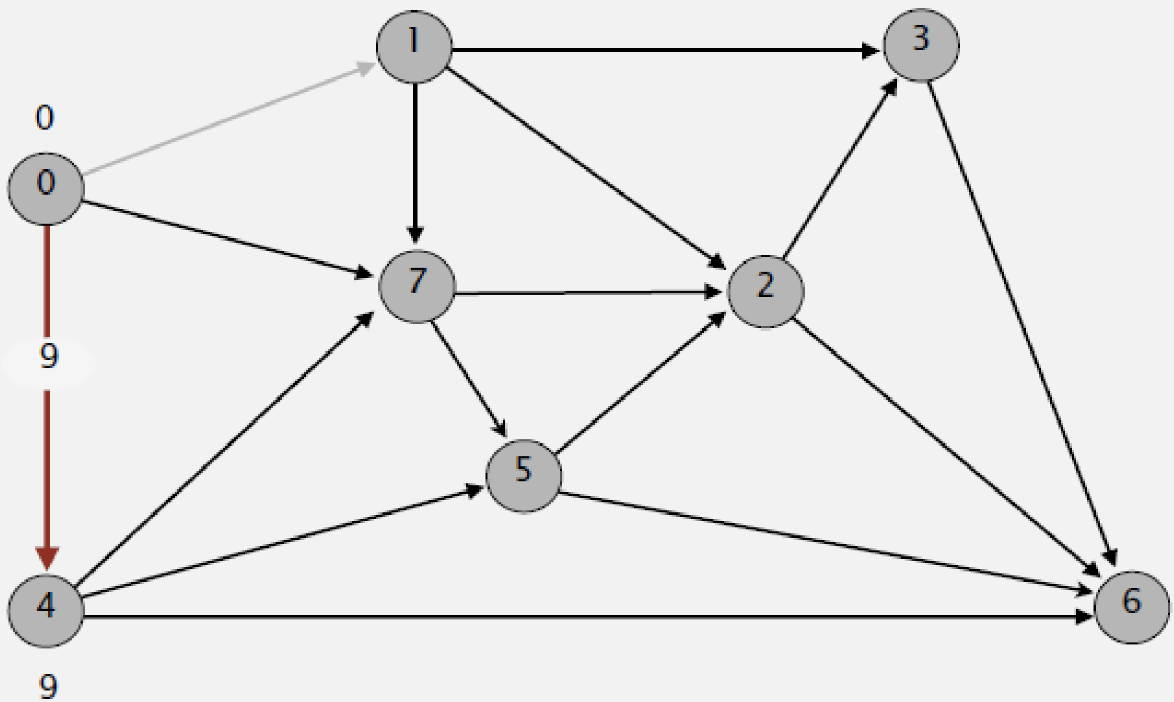
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Algoritmo de Bellman-Ford

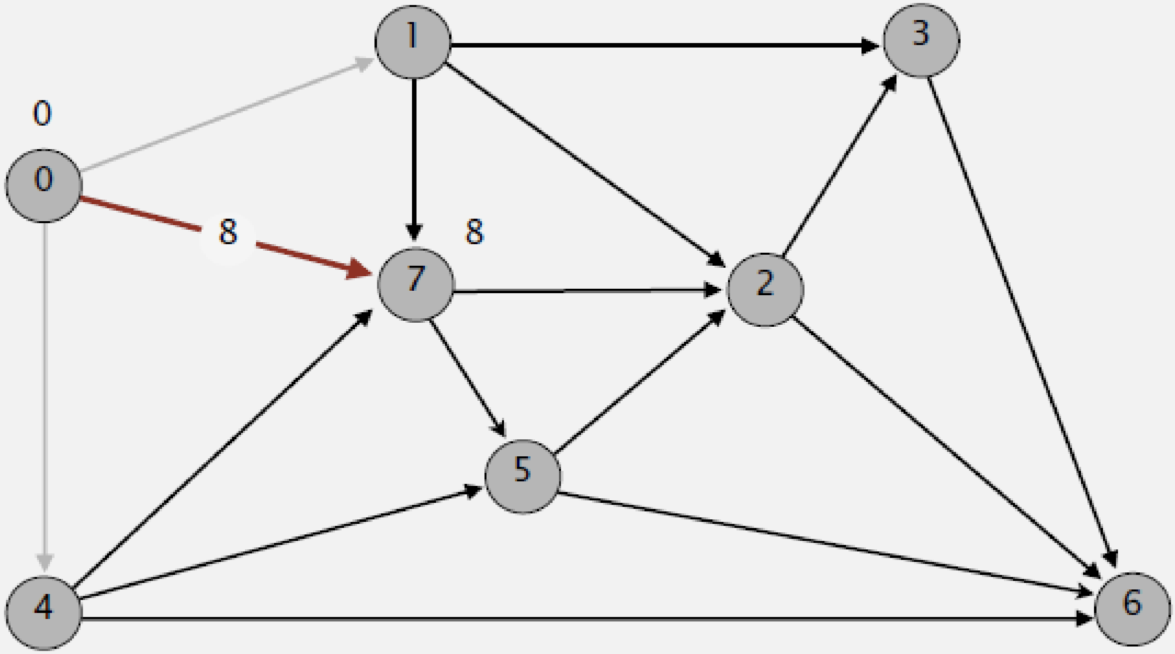


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford



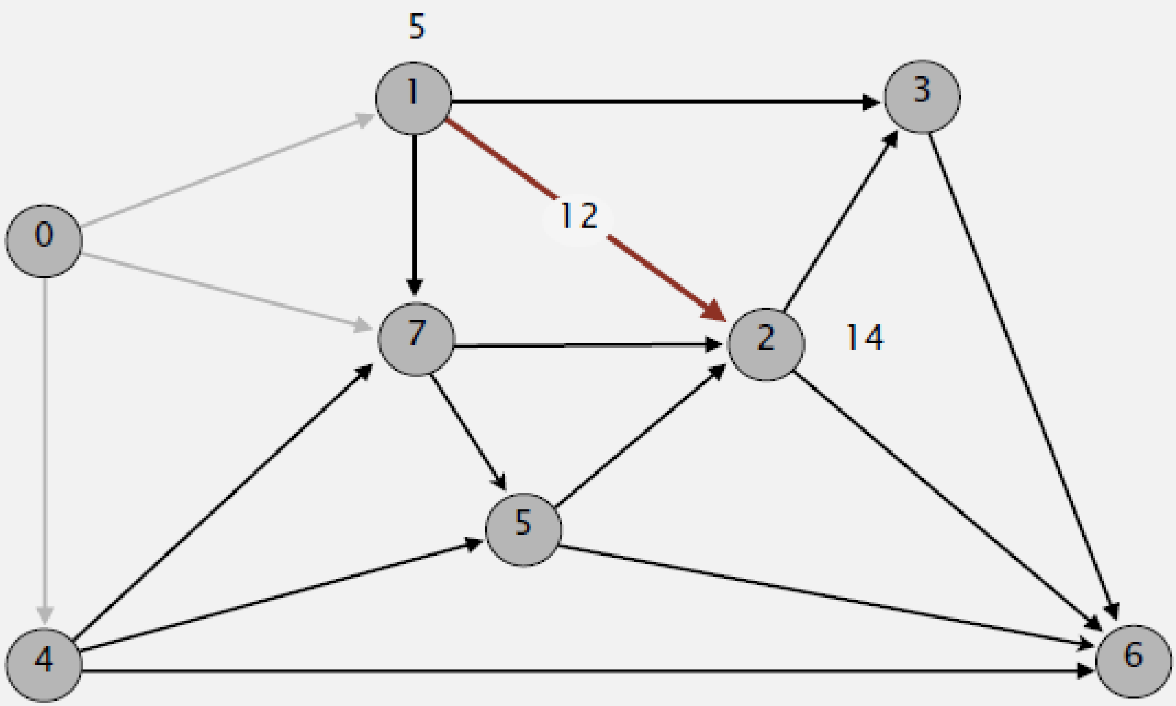
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Algoritmo de Bellman-Ford

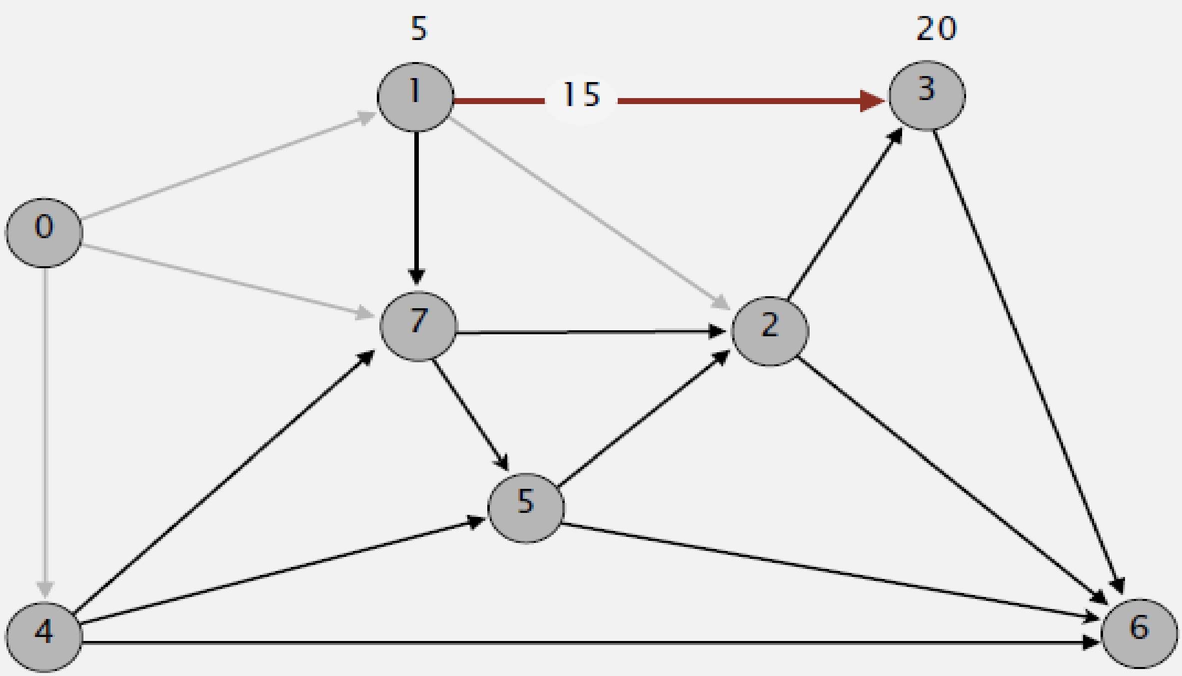


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

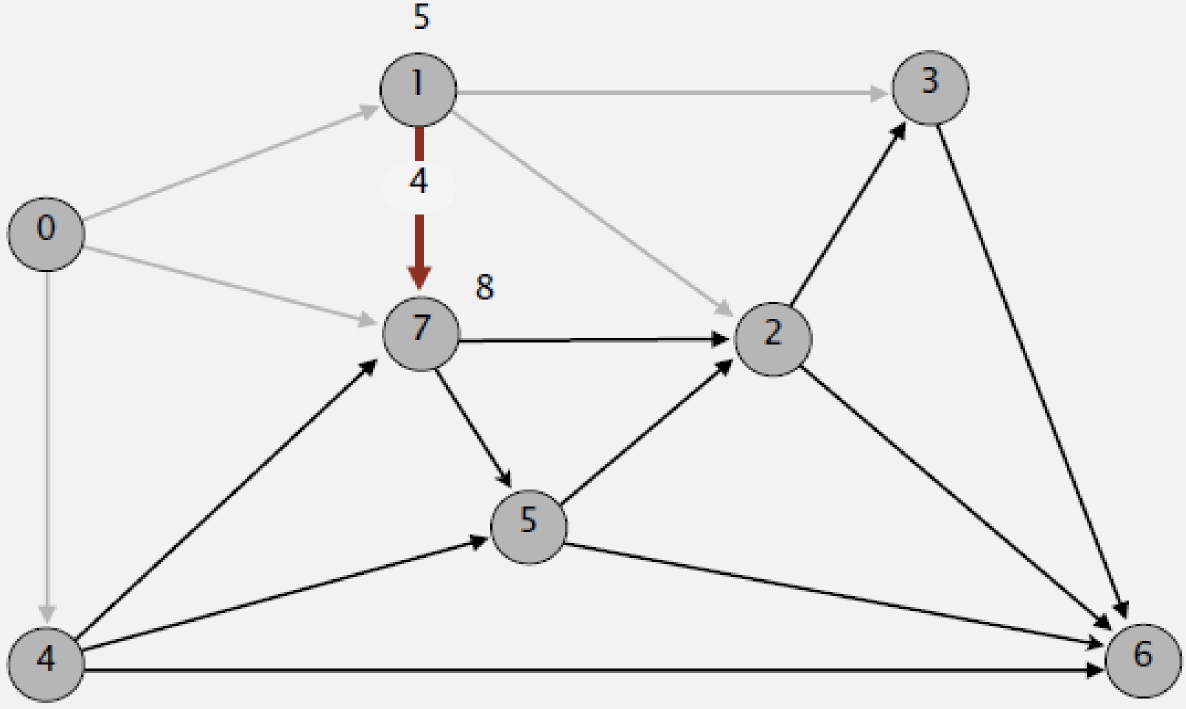


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford



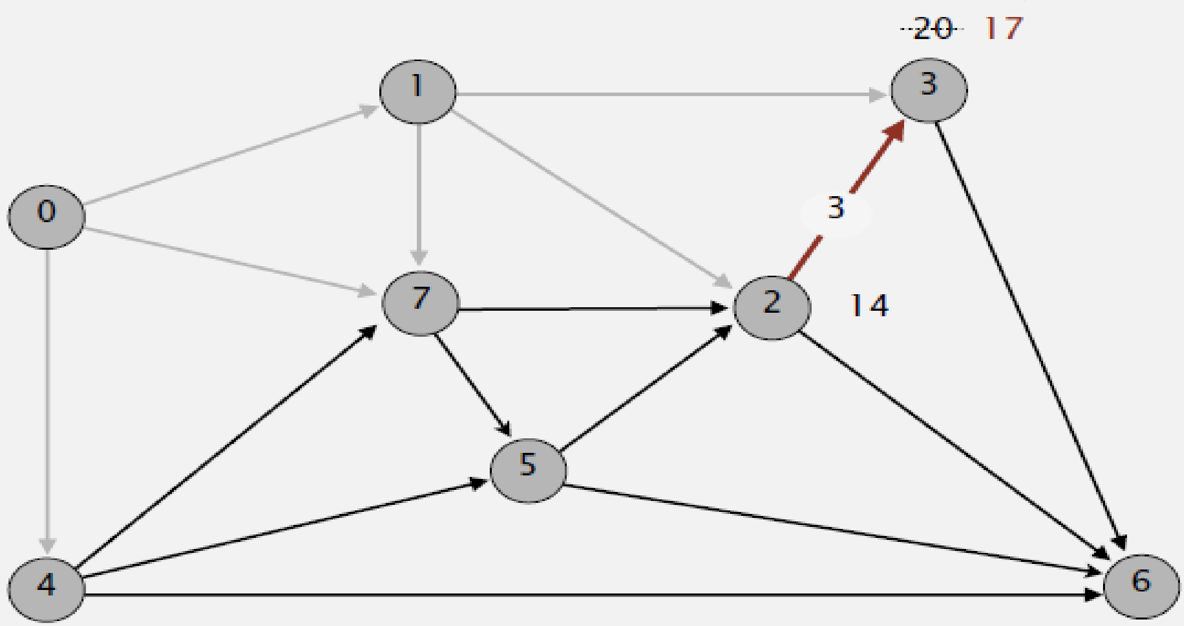
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

2-3 successfully relaxed
in pass 1, but not pass 0

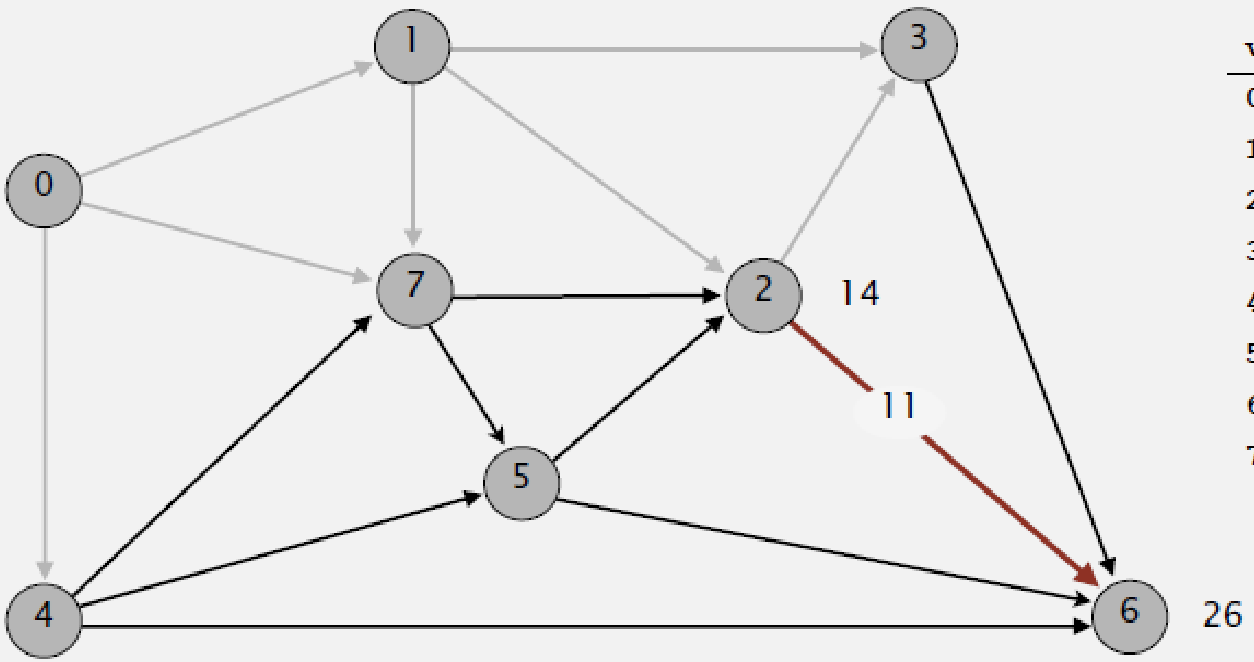


v	distTo[]	edgeTo[]
0	0.0	–
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

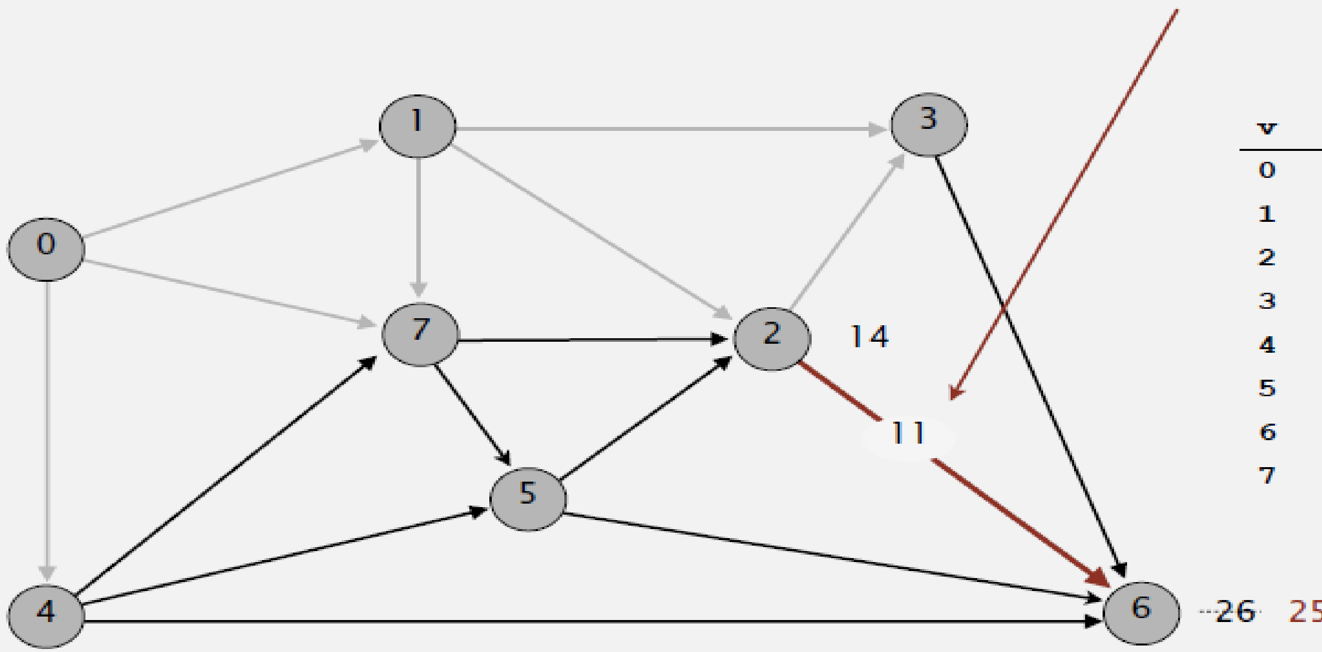
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

↑

Algoritmo de Bellman-Ford

2-6 successfully relaxed
in pass 0 and pass 1

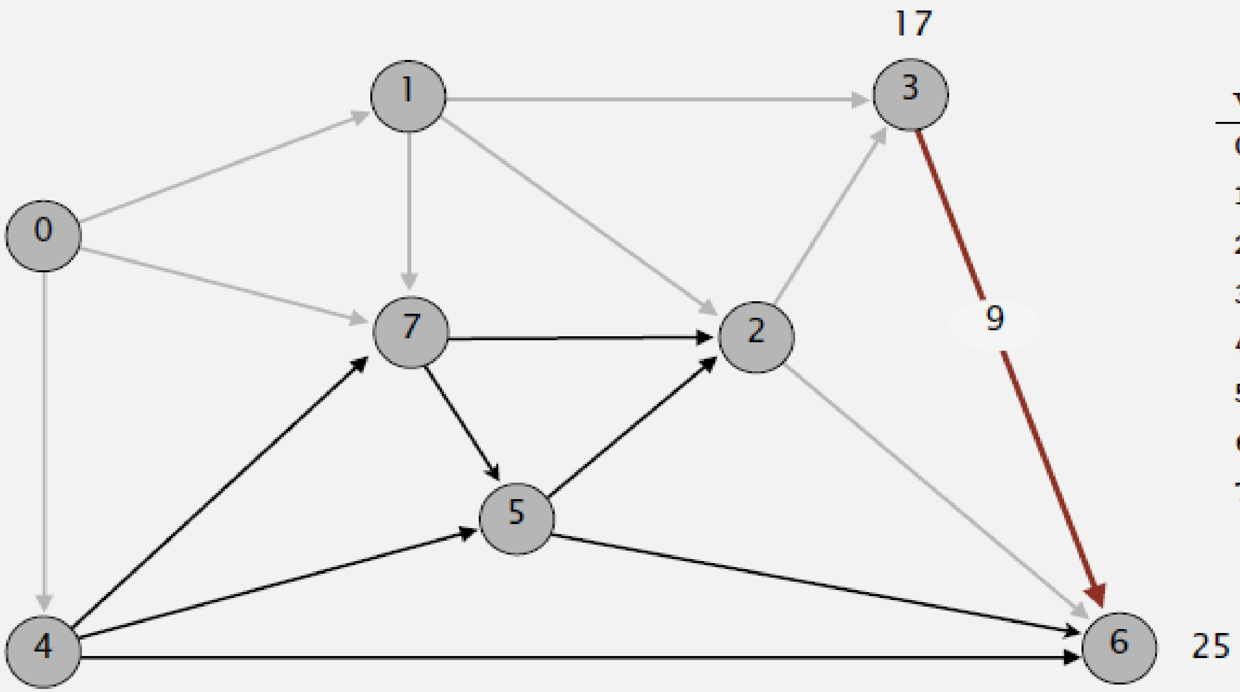


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

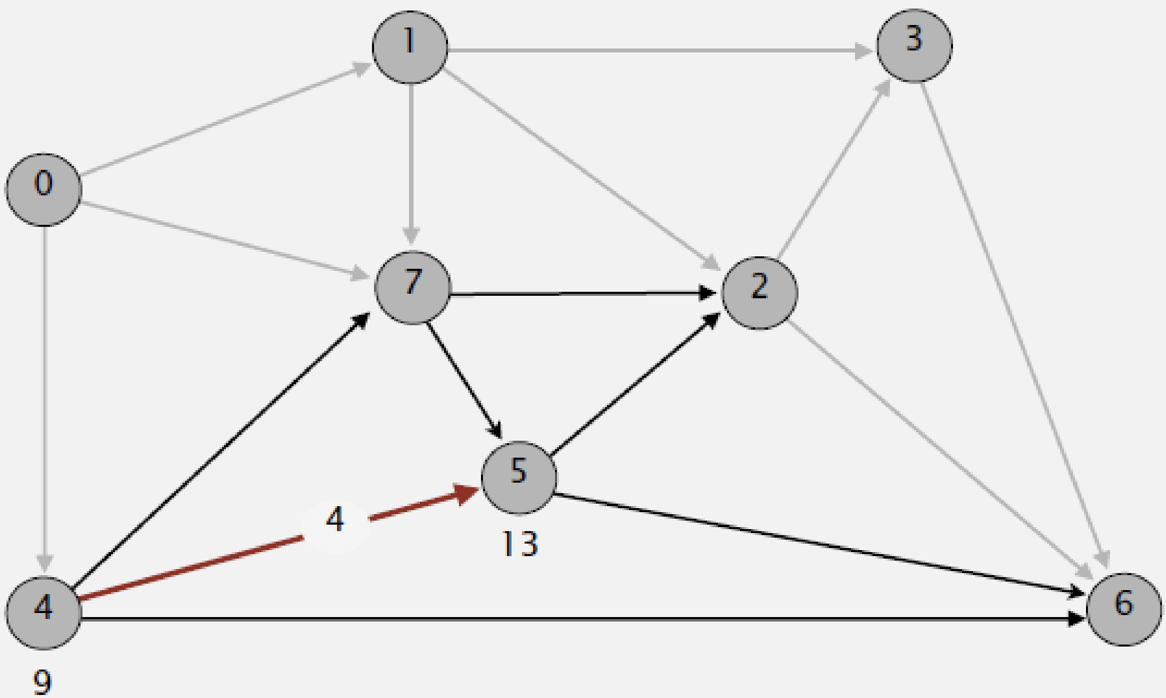


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

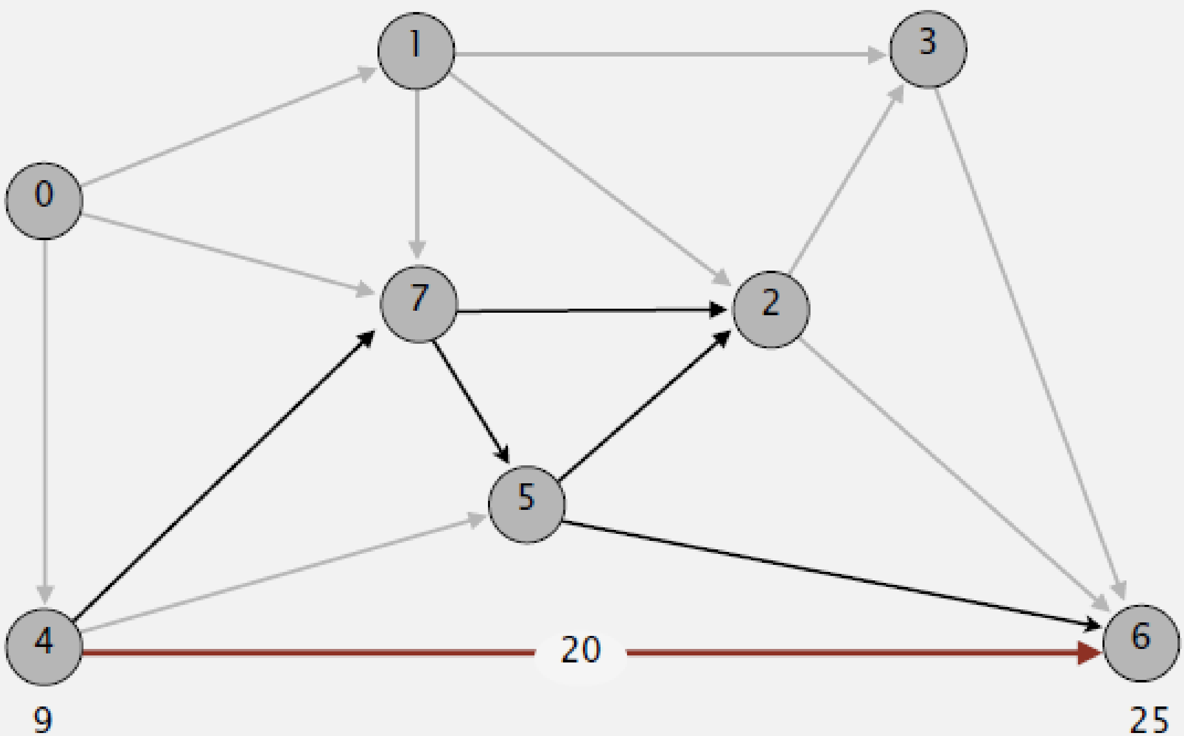


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

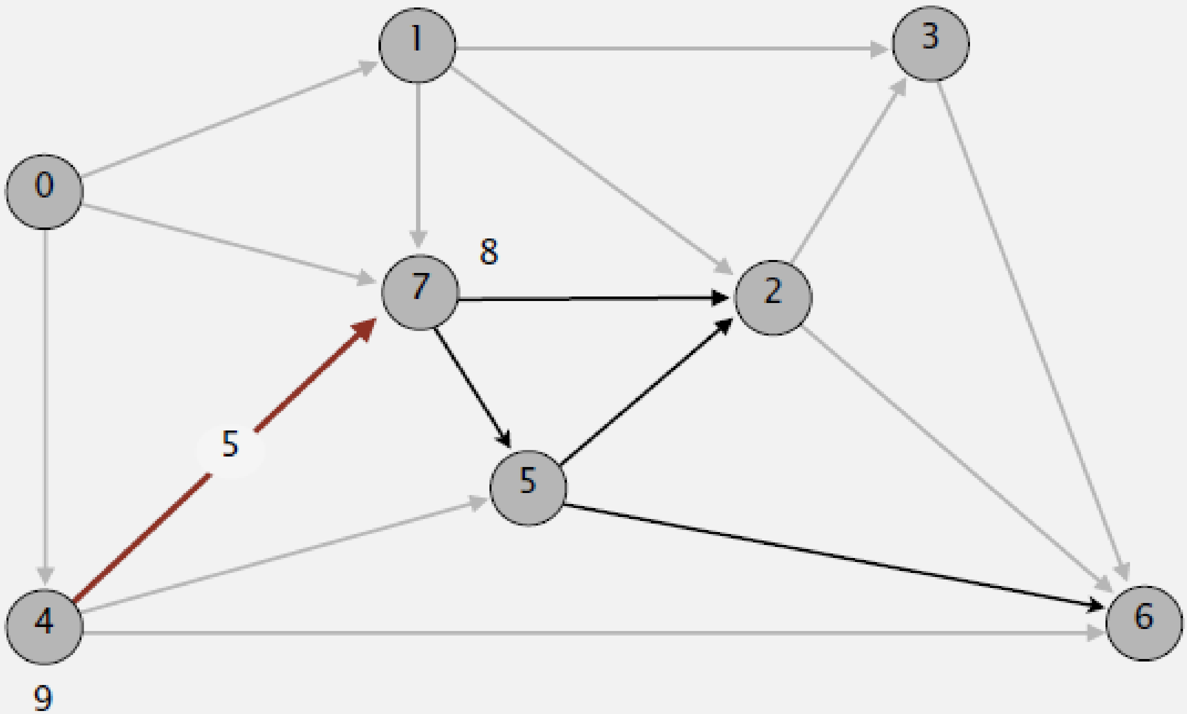


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

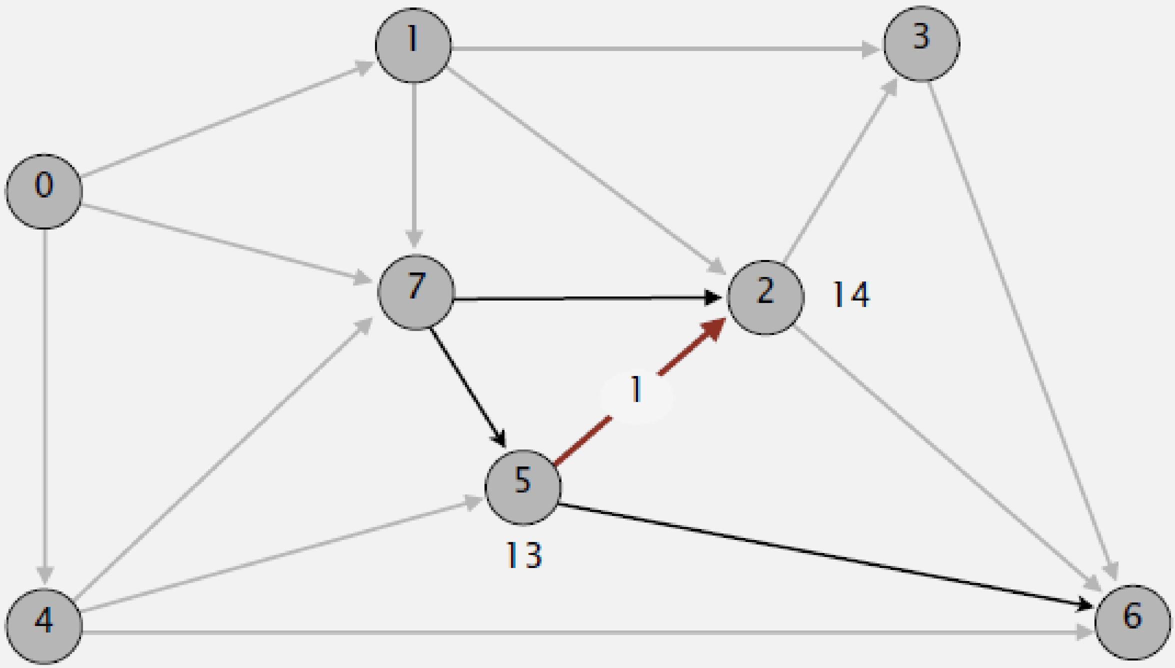


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

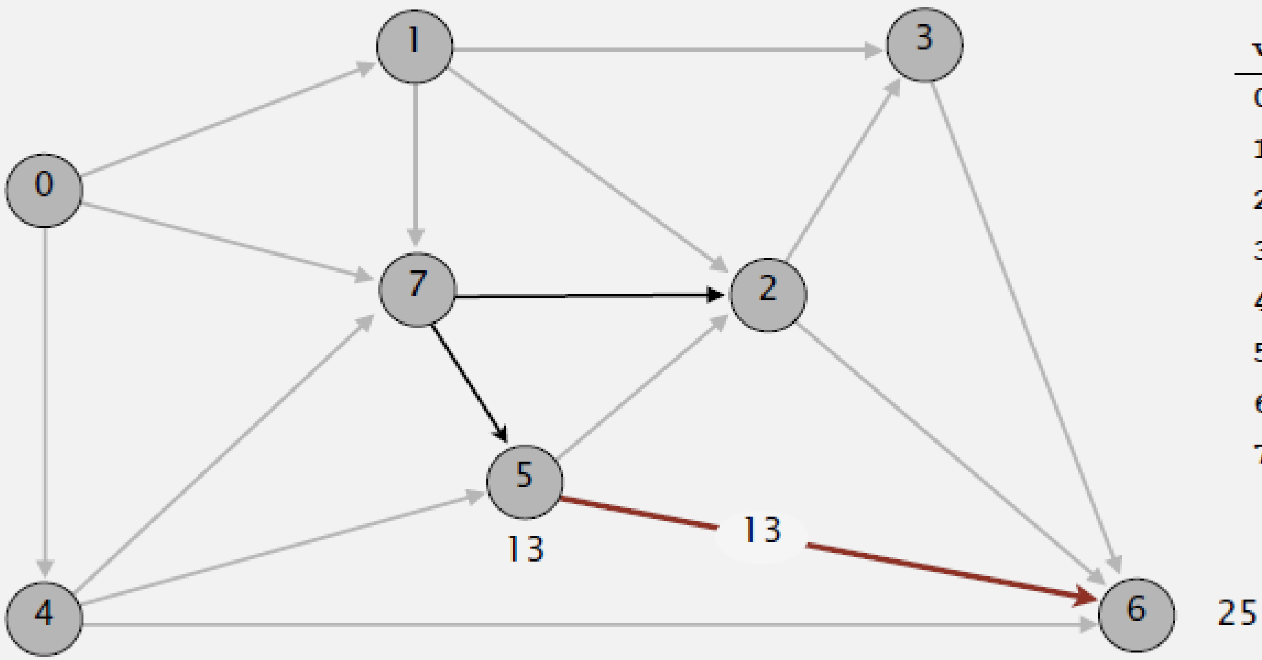


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

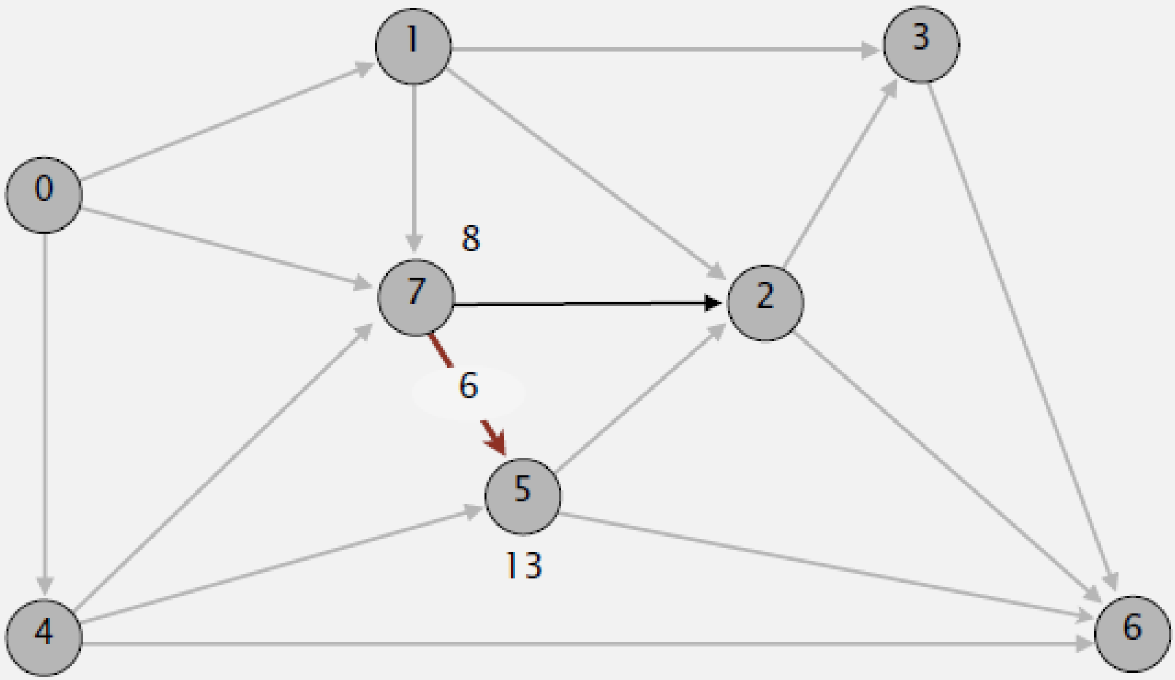


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

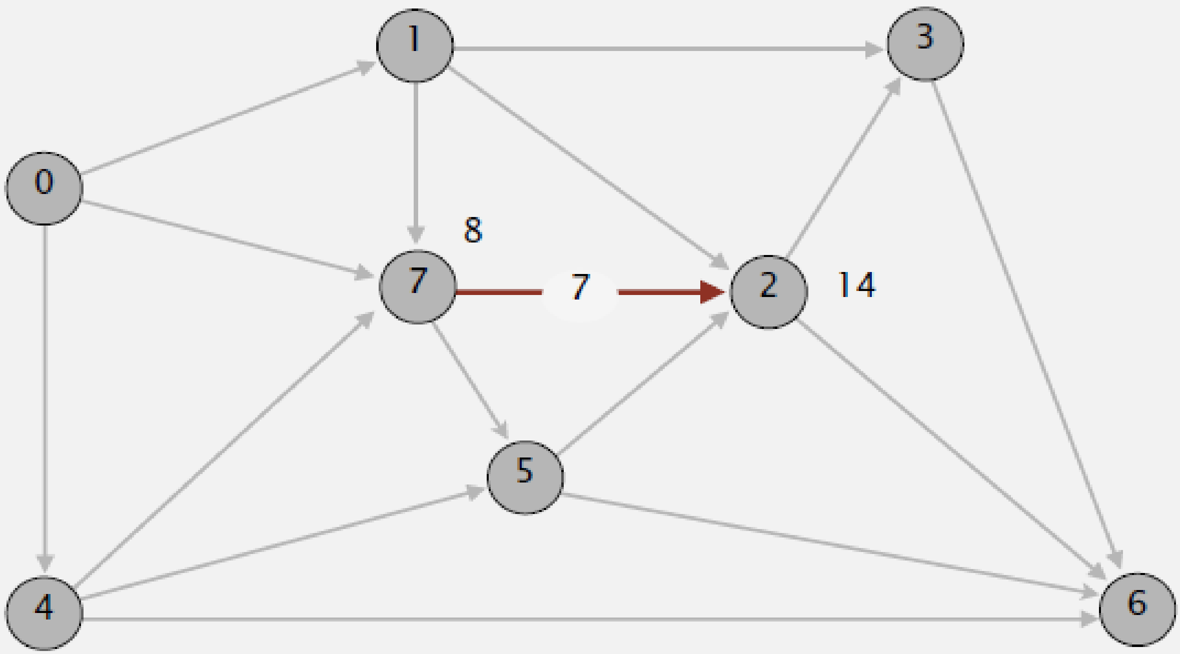


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

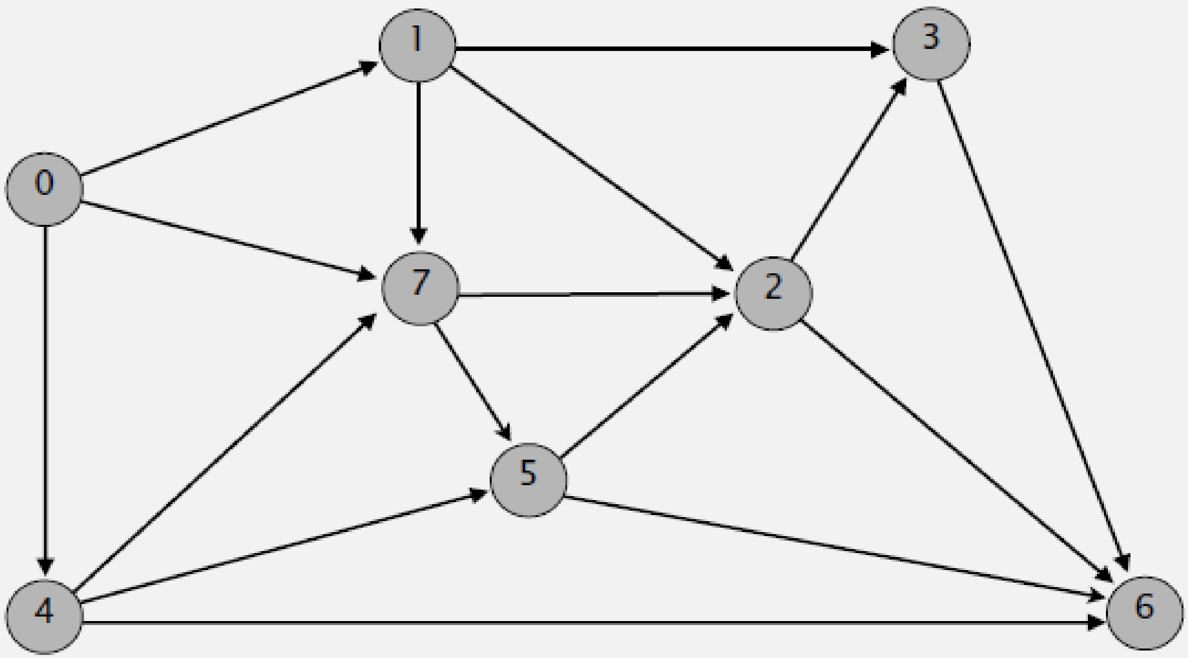


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

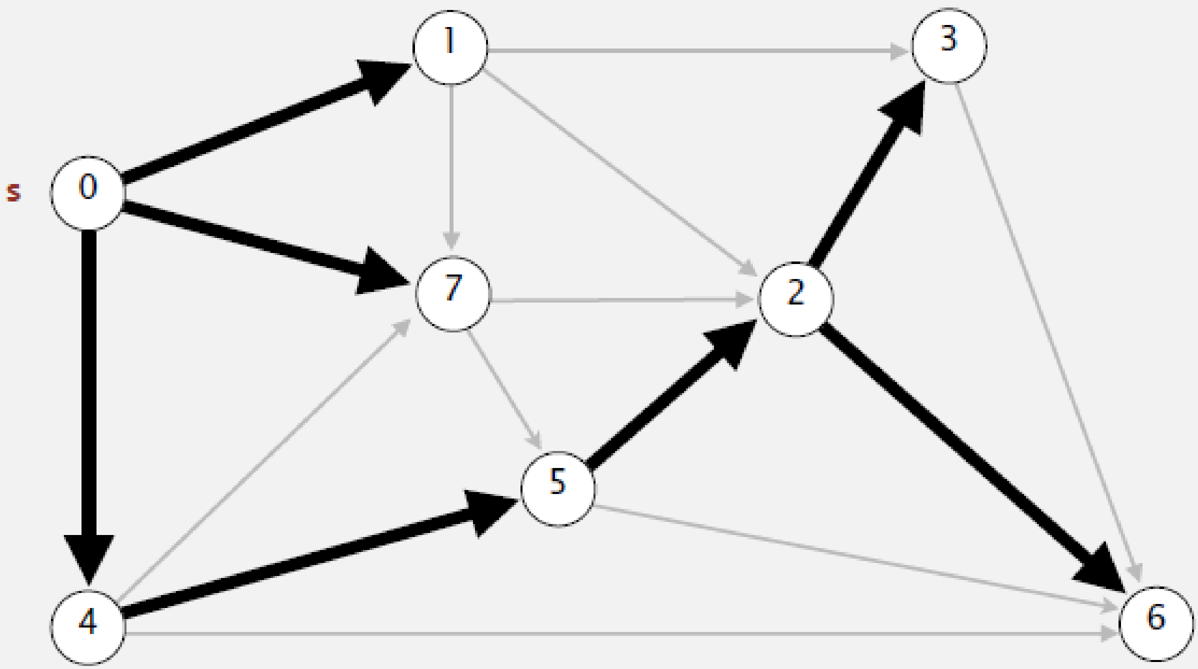


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 2, 3, 4, ... (no further changes)

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Algoritmo de Bellman-Ford

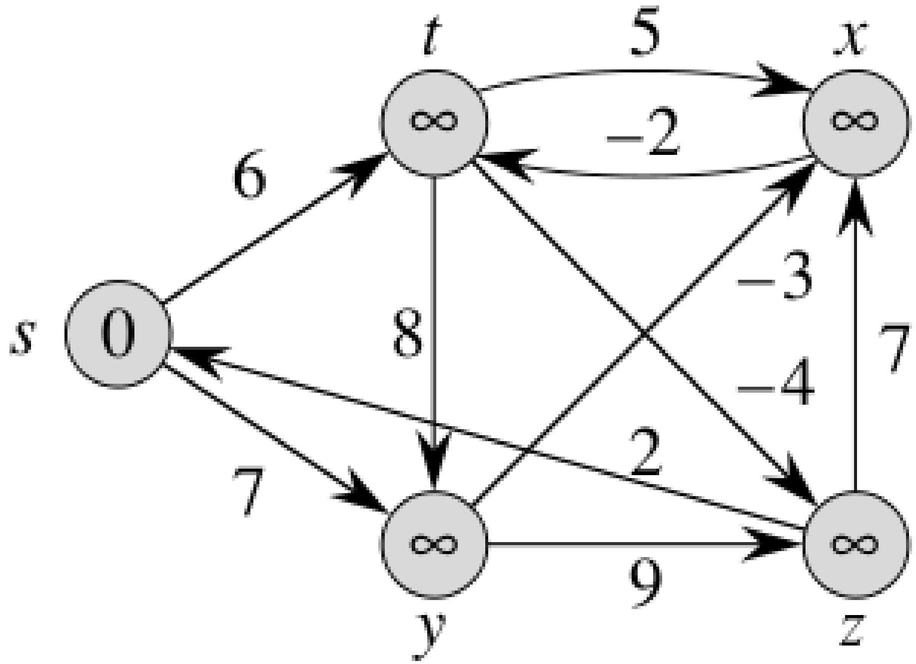


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

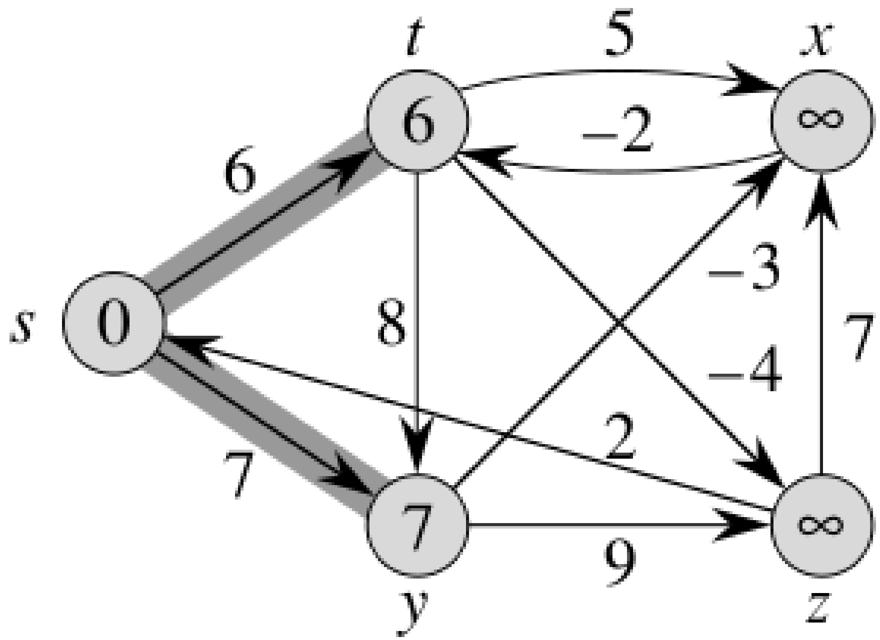
shortest-paths tree from vertex s

Algoritmo de Bellman-Ford

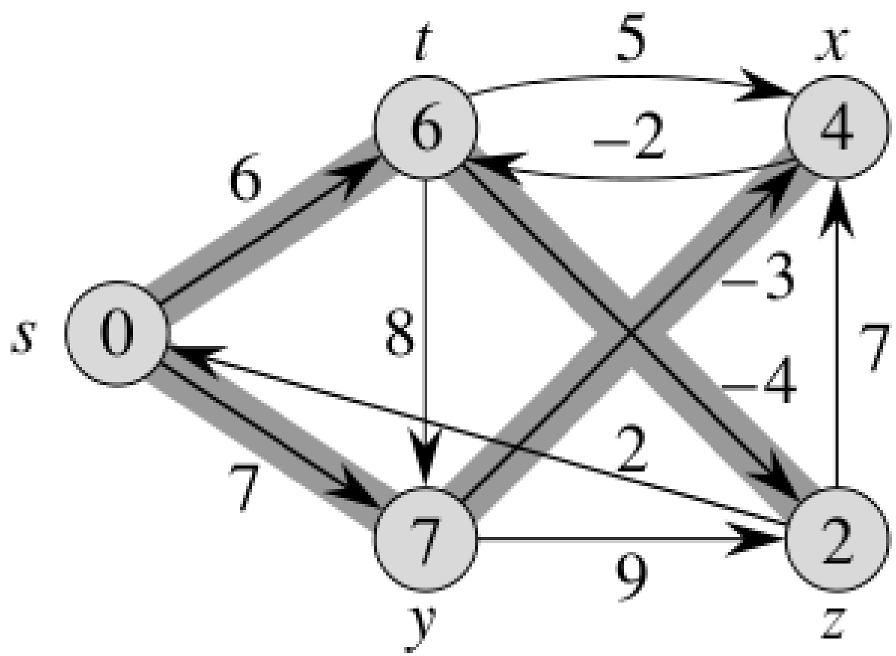
- Encontre o caminho mínimo para o grafo :



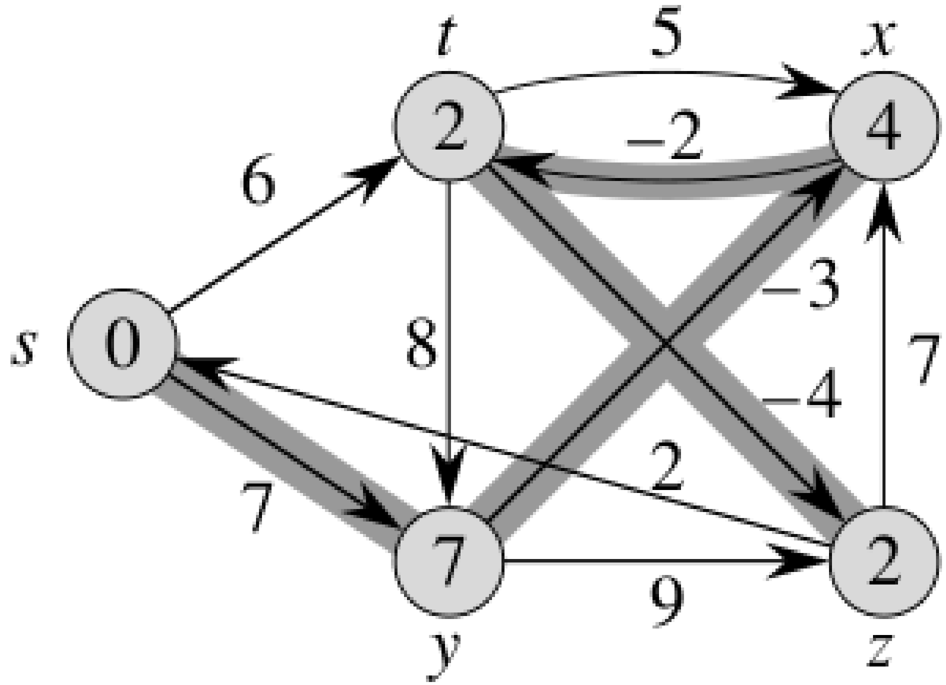
Algoritmo de Bellman-Ford



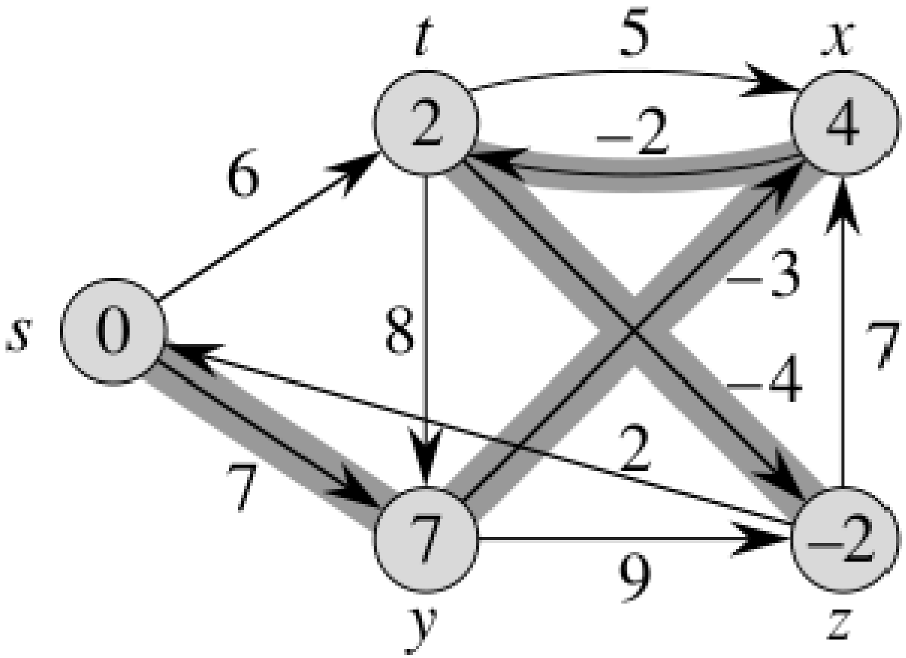
Algoritmo de Bellman-Ford



Algoritmo de Bellman-Ford



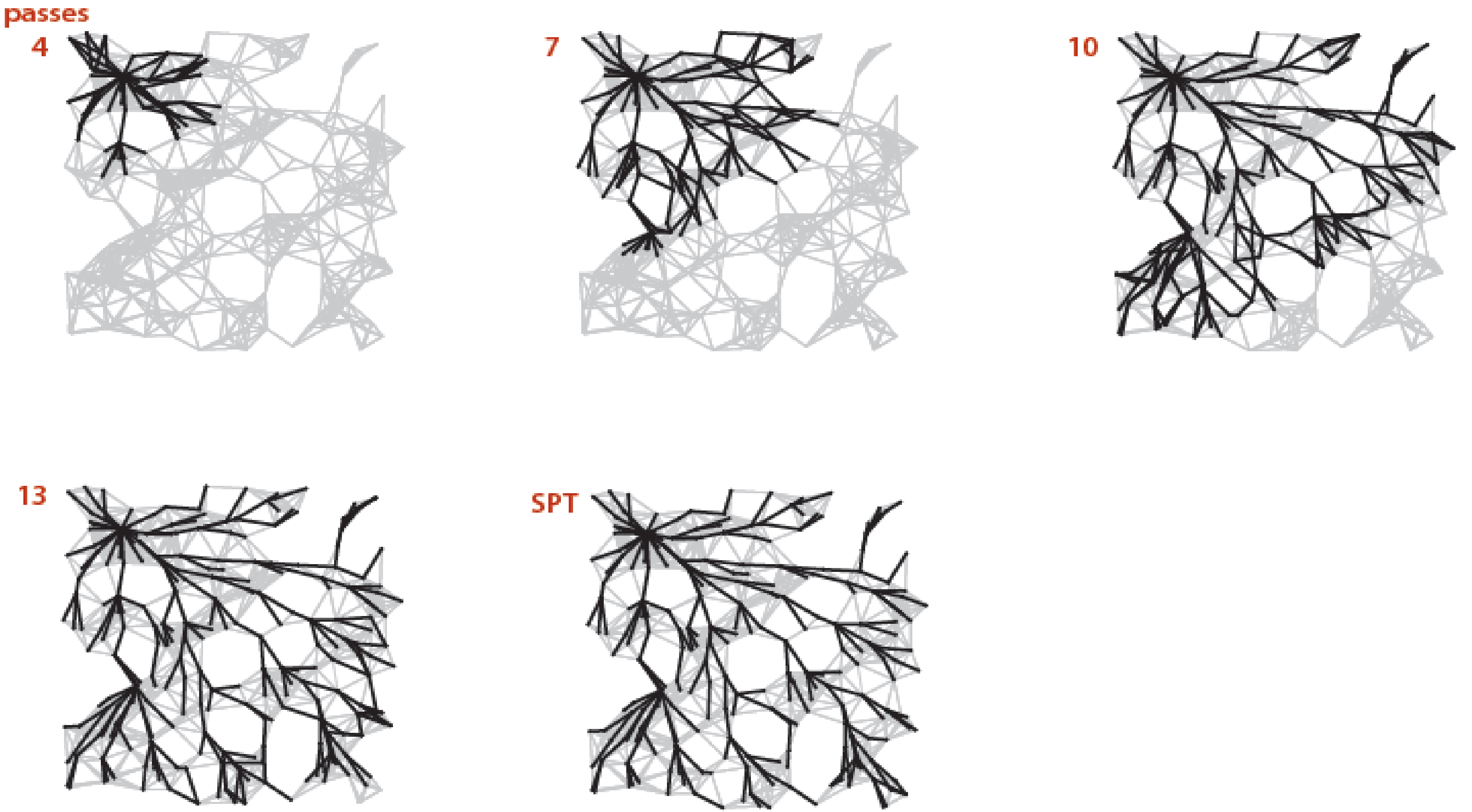
Algoritmo de Bellman-Ford



uni-bh
A MARCA DA EDUCAÇÃO

Algoritmo de Bellman-Ford

Algoritmo de Bellman-Ford



Análise do Algoritmo

- Proposição: O algoritmo de Bellman-Ford calcula o caminho mínimo em qualquer grafo poderado sem ciclos negativos com o tempo proporcional a $E\ x\ V$.
- Idéia para a prova: Após o passo i, o menor caminho encontrado tem no máximo i arestas.

Melhoria Prática

- Observação: Se distTo[v] não muda durante o passo i, não é necessário relaxar qualquer aresta vindo de v no passo $i + 1$.
- Implementação FIFO. Mantém uma fila de vértices cujo distTo[] mudou.
  - Deve-se ficar atento para manter somente uma cópia de cada vértice na fila.
    - (Porquê?)
- Efeito:
  - O tempo de execução do algoritmo é proporcional a $E \; x \; V$ no pior caso.
  - Muito melhor que isso na prática.

Implementação

```
public class BellmanFordSP
{
   private double[] distTo;
   private DirectedEdge[] edgeTo;
   private boolean[] onQ;
   private Queue<Integer> queue;

   public BellmanFordSPT(EdgeWeightedDigraph G, int s)
   {
      distTo = new double[G.V()];
      edgeTo = new DirectedEdge[G.V()];
      onq    = new boolean[G.V()];
      queue  = new Queue<Integer>();

      for (int v = 0; v < V; v++)
         distTo[v] = Double.POSITIVE_INFINITY;
      distTo[s] = 0.0;

      queue.enqueue(s);
      while (!queue.isEmpty())
      {
         int v = queue.dequeue();
         onQ[v] = false;
         for (DirectedEdge e : G.adj(v))
            relax(e);
      }
   }
}
```

queue of vertices whose distTo[] value changes

```
private void relax(DirectedEdge e)
{
   int v = e.from(), w = e.to();
   if (distTo[w] > distTo[v] + e.weight())
   {
       distTo[w] = distTo[v] + e.weight();
       edgeTo[w] = e;
       if (!onQ[w])
       {
          queue.enqueue(w);
          onQ[w] = true;
       }
   }
}
```

Implementação

```
private void relax(DirectedEdge e)
{
   int v = e.from(), w = e.to();
   if (distTo[w] > distTo[v] + e.weight())
   {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
        if (!onQ[w])
        {
           queue.enqueue(w);
           onQ[w] = true;
        }
   }
}
```

Resumo

<table>
<tr><th>algorithm</th><th>restriction</th><th>typical case</th><th>worst case</th><th>extra space</th></tr>
<tr><td>topological sort</td><td>no directed cycles</td><td>E + V</td><td>E + V</td><td>V</td></tr>
<tr><td>Dijkstra (binary heap)</td><td>no negative weights</td><td>E log V</td><td>E log V</td><td>V</td></tr>
<tr><td>Bellman-Ford</td><td rowspan="2">no negative cycles</td><td>E V</td><td>E V</td><td>V</td></tr>
<tr><td>Bellman-Ford (queue-based)</td><td>E + V</td><td>E V</td><td>V</td></tr>
</table>

Resumo

- Lembretes:

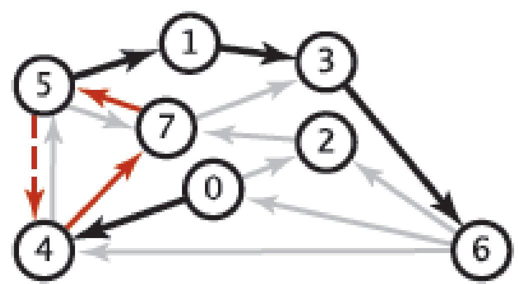
1. Ciclos direcionado tornam o problema mais difícil.
2. Pesos negativos tornam o problema mais difícil.
3. Ciclos negativos tornam o problema intratável.

Encontrando um ciclo negativo

`boolean`	`hasNegativeCycle()`	*is there a negative cycle?*
`Iterable <DirectedEdge>`	`negativeCycle()`	*negative cycle reachable from s*

digraph

4->5	0.35
5->4	-0.66
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93



negative cycle (-0.66 + 0.37 + 0.28)

5->4->7->5

Encontrando um ciclo negativo

```java
// is there a negative cycle reachable from s?
public boolean hasNegativeCycle() {
    return cycle != null;
}

// return a negative cycle; null if no such cycle
public Iterable<DirectedEdge> negativeCycle() {
    return cycle;
}
```

Encontrando um ciclo negativo

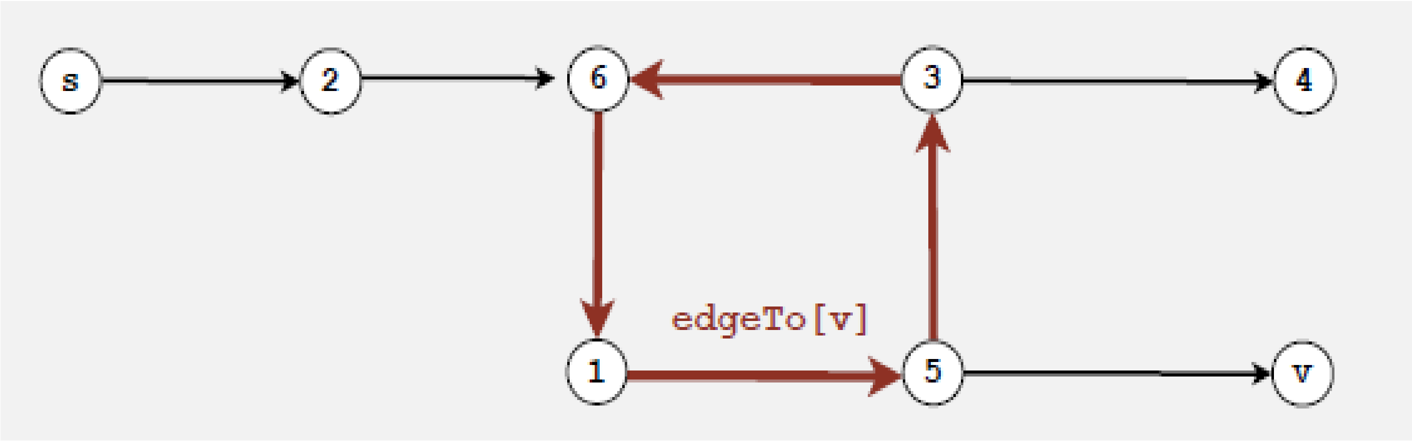
```
// by finding a cycle in predecessor graph
private void findNegativeCycle() {
    int V = edgeTo.length;
    EdgeWeightedDigraph spt = new EdgeWeightedDigraph(V);
    for (int v = 0; v < V; v++)
        if (edgeTo[v] != null)
            spt.addEdge(edgeTo[v]);

    EdgeWeightedDirectedCycle finder = new EdgeWeightedDirectedCycle(spt);
    cycle = finder.cycle();
}
```

Encontrando um ciclo negativo

- Em dúvidas?
  - Dê uma olhada na classe EdgeWeightedDirectedCycle ☺

Encontrando um ciclo negativo

- Observação: Se existe um ciclo negativo, o algoritmo de Bellman-Ford entre em loop infinito atualizando as entradas distTo[] e edgeTo[] dos vértices em ciclo.



Encontrando um ciclo negativo

- Proposição: Se qualquer vértice v é atualizado na fase V, então existe um ciclo negativo (e pode ser rastreado nas entradas edgeTo[v] para ser encontrado).
- Na prática: Verifique por ciclos negativos com mais frequência.