UNIVERSIDADE FEDERAL DE ITAJUBÁ
Instituto de Matemática e Computação

SMAC03 – Grafos

4. Caminho Mínimo

Rafael Frinhani
frinhani@unifei.edu.br

1º Semestre de 2025

OBJETIVOS

Apresentar os conceitos e algoritmos para obtenção de caminhos mínimos em grafos.

AGENDA

4. Caminho Mínimo

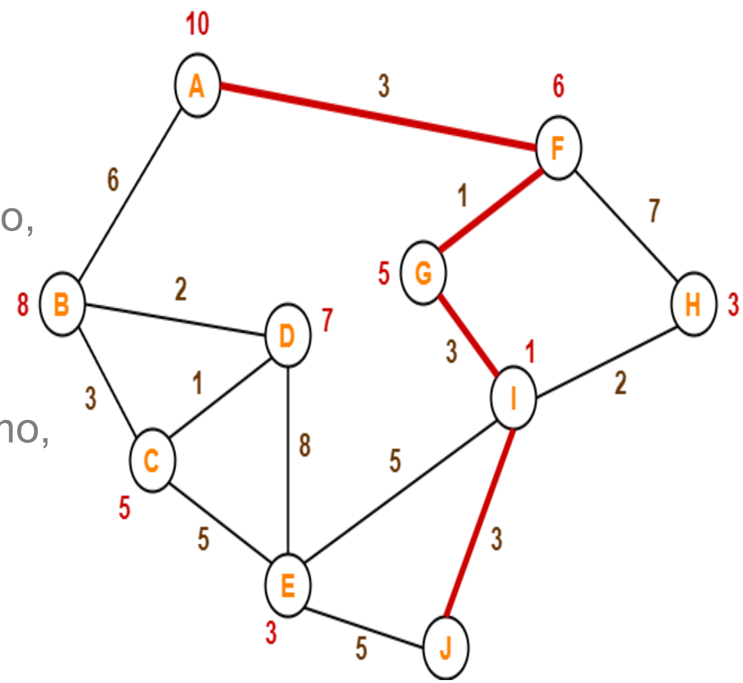
Contextualização, Algoritmo Guloso.

4.1. Algoritmo de Dijkstra

Definições, Princípio de Funcionamento, Algoritmo, Exemplo, Comentários, Limitação.

4.2. Algoritmo de Bellman-Ford

Definições, Princípio de Funcionamento, Algoritmo, Exemplo, Comentários.



Contextualização

O problema do caminho mínimo, ou caminho mais curto, consiste na minimização do custo de travessia de um grafo entre um vértice origem e um destino.

Em um grafo não-ponderado, o caminho mínimo entre dois vértices v (origem) e u (destino) é aquele que possui a menor quantidade de arestas entre eles (pode ser obtido com BFS).

Contextualização

O problema do caminho mínimo, ou caminho mais curto, consiste na minimização do custo de travessia de um grafo entre um vértice origem e um destino.

Em um grafo não-ponderado, o caminho mínimo entre dois vértices v (origem) e u (destino) é aquele que possui a menor quantidade de arestas entre eles (pode ser obtido com BFS).

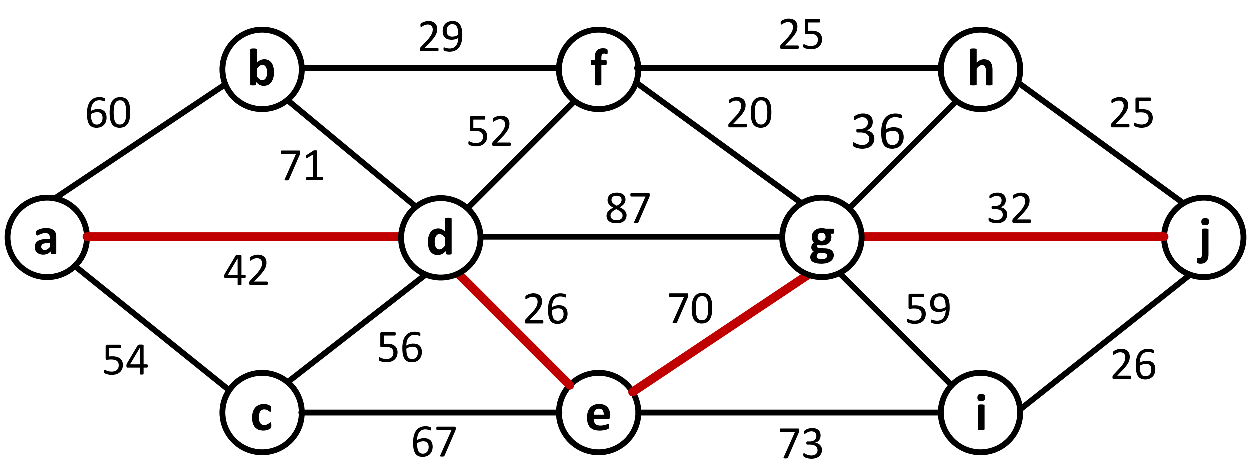
No grafo ponderado leva-se em conta que o relacionamento entre os objetos não é idêntico (ex. distância física, tempo de translado etc.). O custo da travessia a partir de um vértice v até alcançar u é dado pela soma dos pesos de cada aresta (ou arco) percorrida.

O caminho mínimo é aquele que possui o menor custo entre todos os caminhos existentes entre v e u. Poderá existir mais de um caminho mínimo entre v e u com custos mínimos iguais. O caminho mínimo pode não ser aquele com menor número de arestas.

Algoritmo Guloso

Um algoritmo para o caminho mínimo baseado em uma estratégia gulosa seleciona a aresta de menor custo a cada instante.

A estratégia gulosa restringe a visão do custo do caminho para uma perspectiva local. Como o método não volta atrás na sua decisão, caminhos melhores poderão ser desconsiderados na análise.



Custo do Caminho

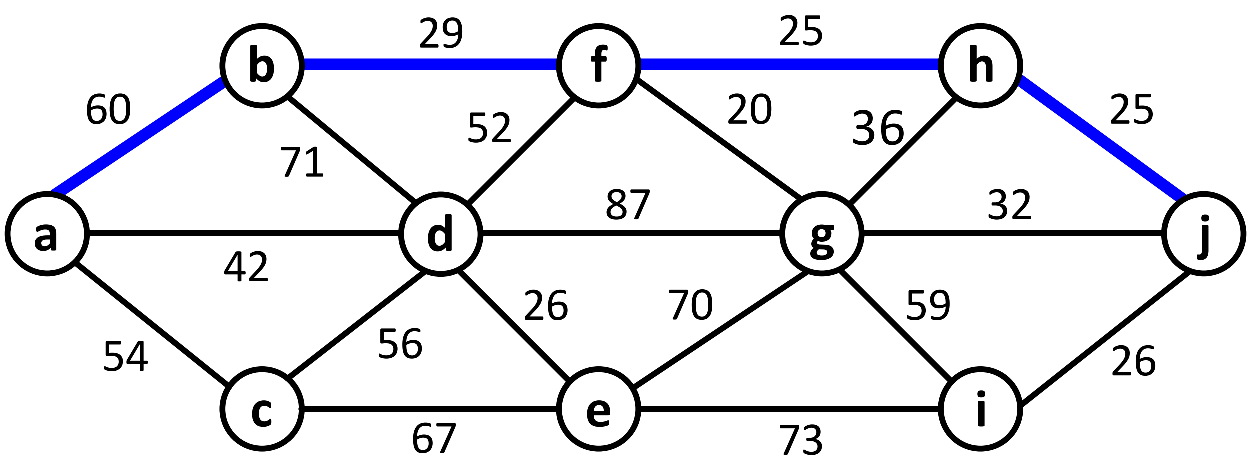
{a, d} = 42
{d, e} = 26
{e, g} = 70
{g, j} = 32

170

Algoritmo Guloso

Um algoritmo para o caminho mínimo baseado em uma estratégia gulosa seleciona a aresta de menor custo a cada instante.

A estratégia gulosa restringe a visão do custo do caminho para uma perspectiva local. Como o método não volta atrás na sua decisão, caminhos melhores poderão ser desconsiderados na análise.



Custo do Caminho

{a, b} = 60
{b, f} = 29
{f, h} = 25
{h, j} = 25

139

Caminho Mínimo a – j
a, b, f, h, j

Algoritmo de Dijkstra

- O algoritmo proposto por Edsger Wybe Dijkstra em 1959.
- Aplicado a grafos simples ou dígrafos, e ponderados (apenas pesos positivos);
- Estruturalmente semelhante à BFS, adota o mesmo conceito de camadas:
  - Obtém a menor distância do vértice origem até seus vizinhos, depois dos vizinhos do vértice origem até seus próprios vizinhos e assim por diante.
  - Atualiza as distâncias sempre que descobre uma menor.
  - Atualiza o vértice precedente sempre que atualiza uma distância.

Algoritmo de Dijkstra

- O algoritmo proposto por Edsger Wybe Dijkstra em 1959.
- Aplicado a grafos simples ou dígrafos, e ponderados (apenas pesos positivos);
- Estruturalmente semelhante à BFS, adota o mesmo conceito de camadas:
  - Obtém a menor distância do vértice origem até seus vizinhos, depois dos vizinhos do vértice origem até seus próprios vizinhos e assim por diante.
  - Atualiza as distâncias sempre que descobre uma menor.
  - Atualiza o vértice precedente sempre que atualiza uma distância.

Princípio de Funcionamento

1. Mantém conjuntos de vértices abertos (A) e fechados (F). Vértice Fechado = caminho mínimo do vértice origem até ele já foi calculado. Caso contrário é considerado aberto.
2. Mantém o menor caminho conhecido até o momento para cada vértice descoberto.
3. Adiciona o vértice de menor caminho ao conjunto fechado.
4. Atualiza as distâncias considerando este vértice, descobrindo novos vértices.

Algoritmo

dijkstra(G, vInicio, vFim)

```
for i = 0 até |V| do
    custo[i] = ∞;
    rota[i] = vInicio;
end
custo[vInicio] = 0;
A = V;
F = ∅;
while A ≠ ∅ do
    v = v ∈ A com menor custo[v] entre todos;
    F = F ∪ {v};
    A = A − {v};
    N = N − F;
    for cada u ∈ N do
        if custo[v] + w_vu < custo[u] then
            custo[u] = custo[v] + w_vu;
            rota[u] = v;
        end
    end
end
caminho = obtemCaminho(rota, vInicio, vFim);
caminho.reverse;
return caminho, custo[vFim]
```

ESTRUTURAS

- F : conjunto de vértices Fechados.
- A : conjunto de vértices Abertos.
- N : conjunto de vértices vizinhos ao vértice atual.
- **custo[v]** : vetor que armazena o custo (distância) entre o vértice origem e os demais vértices de G.
- **rota[v]** : vetor que armazena o índice do vértice que precede o vértice v, no caminho cuja distância consta no vetor $custo[v]$.
- w_{vu} : valor do peso da aresta ou arco que conecta v e u.

Algoritmo

dijkstra(G, vInicio, vFim)

```
1  for i = 0 até |V| do
2      custo[i] = ∞;
3      rota[i] = vInicio;
4  end
5  custo[vInicio] = 0;
6  A = V;
7  F = ∅;
8  while A ≠ ∅ do
9      v = v ∈ A com menor custo[v] entre todos;
10     F = F ∪ {v};
11     A = A − {v};
12     N = N − F;
13     for cada u ∈ N do
14         if custo[v] + w_vu < custo[u] then
15             custo[u] = custo[v] + w_vu;
16             rota[u] = v;
17         end
18     end
19 end
20 caminho = obtemCaminho(rota, vInicio, vFim);
21 caminho.reverse;
22 return caminho, custo[vFim]
```

ESTRUTURAS

- F : conjunto de vértices Fechados.
- A : conjunto de vértices Abertos.
- N : conjunto de vértices vizinhos (adjacentes) ao vértice atual.
- ***custo*[*v*]** : vetor que armazena o custo (distância) entre o vértice origem e os demais vértices de G.
- ***rota*[*v*]** : vetor que armazena o índice do vértice que precede o vértice v, no caminho cuja distância consta no vetor *custo*[*v*].
- w_{vu} : valor do peso da aresta ou arco que conecta v e u.

Os vetores de custo e rota são inicializados conforme *vInício* (linhas 1 a 4).

O custo de *vInicio* é definido como 0 (linha 5), são inicializados os conjuntos A e F (linhas 6 e 7).

Algoritmo

```
dijkstra(G, vInicio, vFim)
1  for i = 0 até |V| do
2      custo[i] = ∞;
3      rota[i] = vInicio;
4  end
5  custo[vInicio] = 0;
6  A = V;
7  F = ∅;
8  while A ≠ ∅ do
9      v = v ∈ A com menor custo[v] entre todos;
10     F = F ∪ {v};
11     A = A − {v};
12     N = N − F;
13     for cada u ∈ N do
14         if custo[v] + w_vu < custo[u] then
15             custo[u] = custo[v] + w_vu;
16             rota[u] = v;
17         end
18     end
19 end
20 caminho = obtemCaminho(rota, vInicio, vFim);
21 caminho.reverse;
22 return caminho, custo[vFim]
```

ESTRUTURAS

- F : conjunto de vértices Fechados.
- A : conjunto de vértices Abertos.
- N : conjunto de vértices vizinhos (adjacentes) ao vértice atual.
- ***custo*[*v*]** : vetor que armazena o custo (distância) entre o vértice origem e os demais vértices de G.
- ***rota*[*v*]** : vetor que armazena o índice do vértice que precede o vértice v, no caminho cuja distância consta no vetor *custo*[v].
- w_{vu} : valor do peso da aresta ou arco que conecta v e u.

Os vetores de custo e rota são inicializados conforme *vInício* (linhas 1 a 4).

O custo de *vInicio* é definido como 0 (linha 5), são inicializados os conjuntos A e F (linhas 6 e 7).

Enquanto existirem vértices abertos (linhas 8 a 19):

Obter o vértice v a partir de A que tenha o menor custo entre todos os demais (linha 9).

Incluir v no conjunto F e remover do conjunto A (linhas 10 e 11).

Obter os adjacentes a v que ainda não foram fechados (linha 12)

Algoritmo

```
dijkstra(G, vInicio, vFim)
1  for i = 0 até |V| do
2      custo[i] = ∞;
3      rota[i] = vInicio;
4  end
5  custo[vInicio] = 0;
6  A = V;
7  F = ∅;
8  while A ≠ ∅ do
9      v = v ∈ A com menor custo[v] entre todos;
10     F = F ∪ {v};
11     A = A − {v};
12     N = N − F;
13     for cada u ∈ N do
14         if custo[v] + w_vu < custo[u] then
15             custo[u] = custo[v] + w_vu;
16             rota[u] = v;
17         end
18     end
19 end
20 caminho = obtemCaminho(rota, vInicio, vFim);
21 caminho.reverse;
22 return caminho, custo[vFim]
```

ESTRUTURAS

- ***F*** : conjunto de vértices Fechados.
- ***A*** : conjunto de vértices Abertos.
- ***N*** : conjunto de vértices vizinhos (adjacentes) ao vértice atual.
- ***custo*[*v*]** : vetor que armazena o custo (distância) entre o vértice origem e os demais vértices de *G*.
- ***rota*[*v*]** : vetor que armazena o índice do vértice que precede o vértice *v*, no caminho cuja distância consta no vetor *custo*[*v*].
- w_{vu} : valor do peso da aresta ou arco que conecta *v* e *u*.

Para cada vértice *u* adjacente a *v* que ainda esteja em aberto (linha 13).

Se o custo de *v* mais o peso da aresta {*v*, *u*} for menor que o custo de *u* (linha 14):

Atribuir como custo de *u* (linha 15) o custo do caminho até *v* mais o peso da aresta {*v*, *u*}.

Atribuir *v* como predecessor (antes) de *u* no vetor rota (linha 16).

4.1. Dijkstra

Algoritmo

dijkstra(G, vInicio, vFim)

```
1  for i = 0 até |V| do
2      custo[i] = ∞;
3      rota[i] = vInicio;
4  end
5  custo[vInicio] = 0;
6  A = V;
7  F = ∅;
8  while A ≠ ∅ do
9      v = v ∈ A com menor custo[v] entre todos;
10     F = F ∪ {v};
11     A = A − {v};
12     N = N − F;
13     for cada u ∈ N do
14         if custo[v] + w_vu < custo[u] then
15             custo[u] = custo[v] + w_vu;
16             rota[u] = v;
17         end
18     end
19 end
20 caminho = obtemCaminho(rota, vInicio, vFim);
21 caminho.reverse;
22 return caminho, custo[vFim]
```

ESTRUTURAS

- F : conjunto de vértices Fechados.
- A : conjunto de vértices Abertos.
- N : conjunto de vértices vizinhos (adjacentes) ao vértice atual.
- $custo[v]$: vetor que armazena o custo (distância) entre o vértice origem e os demais vértices de G.
- $rota[v]$: vetor que armazena o índice do vértice que precede o vértice v, no caminho cuja distância consta no vetor $custo[v]$.
- w_{vu} : valor do peso da aresta ou arco que conecta v e u.

Obtém o caminho mínimo percorrendo o vetor de rota a partir de $vFim$, passando pelos precedentes até alcançar $vInicio$ (linha 20).

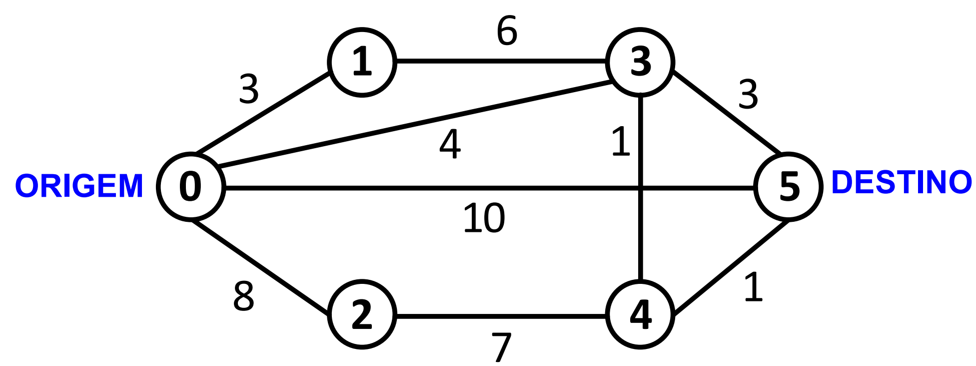
Inverte o caminho mínimo (linha 21).

Retorna a sequência de vértices e o custo do caminho (linha 22).

Exemplo

```
dijkstra(G, vInicio, vFim)

for i = 0 até |V| do
    custo[i] = ∞;
    rota[i] = vInicio;
end
custo[vInicio] = 0;
A = V;
F = ∅;
while A ≠ ∅ do
    v = v ∈ A com menor custo[v] entre todos;
    F = F ∪ {v};
    A = A − {v};
    N = N − F;
    for cada u ∈ N do
        if custo[v] + w_vu < custo[u] then
            custo[u] = custo[v] + w_vu;
            rota[u] = v;
        end
    end
end
caminho = obtemCaminho(rota, vInicio, vFim);
caminho.reverse;
return caminho, custo[vFim]
```



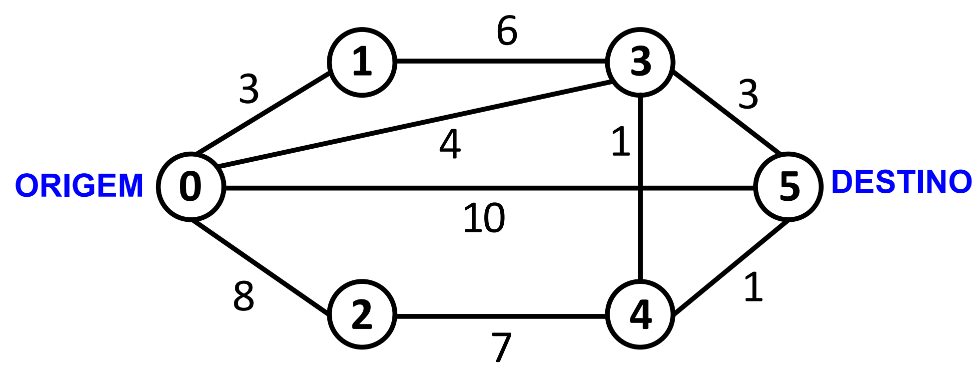
	0	1	2	3	4	5
custo	0	3	8	4	∞	10
rota	-	0	0	0	0	0

v = 0 (origem) F = {0}

Exemplo

dijkstra(G, vInicio, vFim)

```
for i = 0 até |V| do
    custo[i] = ∞;
    rota[i] = vInicio;
end
custo[vInicio] = 0;
A = V;
F = ∅;
while A ≠ ∅ do
    v = v ∈ A com menor custo[v] entre todos;
    F = F ∪ {v};
    A = A − {v};
    N = N − F;
    for cada u ∈ N do
        if custo[v] + w_vu < custo[u] then
            custo[u] = custo[v] + w_vu;
            rota[u] = v;
        end
    end
end
caminho = obtemCaminho(rota, vInicio, vFim);
caminho.reverse;
return caminho, custo[vFim]
```



	~~0~~	1	2	3	4	5
custo	0	3	8	4	∞	10
rota	-	0	0	0	0	0

v = 0 (origem) $F = \{0\}$

v = 1 (menor) $F = \{0, 1\}$

ATUALIZAÇÃO DOS CUSTOS E ROTAS

$custo[2] = min(8, 3 + A[1, 2]) = min(8, \infty) = 8$

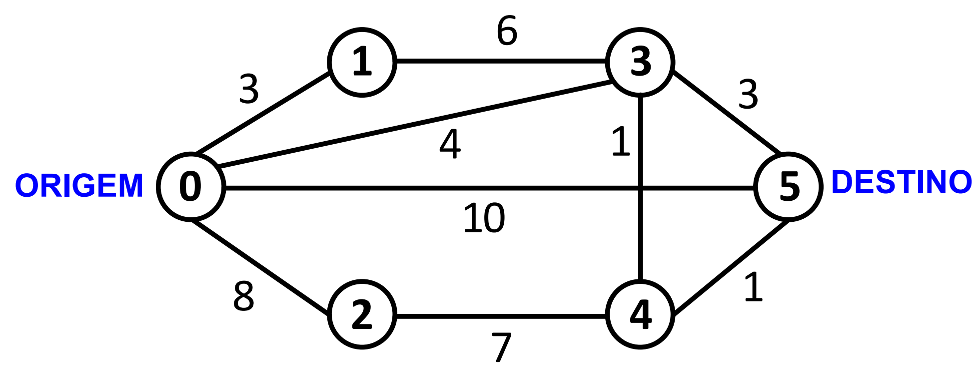
$custo[3] = min(4, 3 + A[1, 3]) = min(4, 9) = 4$

$custo[4] = min(\infty, 3 + A[1, 4]) = min(\infty, \infty) = \infty$

$custo[5] = min(10, 3 + A[1, 5]) = min(10, \infty) = 10$

Exemplo

```
dijkstra(G, vInicio, vFim)
for i = 0 até |V| do
    custo[i] = ∞;
    rota[i] = vInicio;
end
custo[vInicio] = 0;
A = V;
F = ∅;
while A ≠ ∅ do
    v = v ∈ A com menor custo[v] entre todos;
    F = F ∪ {v};
    A = A − {v};
    N = N − F;
    for cada u ∈ N do
        if custo[v] + w_vu < custo[u] then
            custo[u] = custo[v] + w_vu;
            rota[u] = v;
        end
    end
end
caminho = obtemCaminho(rota, vInicio, vFim);
caminho.reverse;
return caminho, custo[vFim]
```



	~~0~~	~~1~~	2	3	4	5
custo	0	3	8	4	∞	10
rota	-	0	0	0	0	0

v = 3 (menor) F = {0, 1, 3}

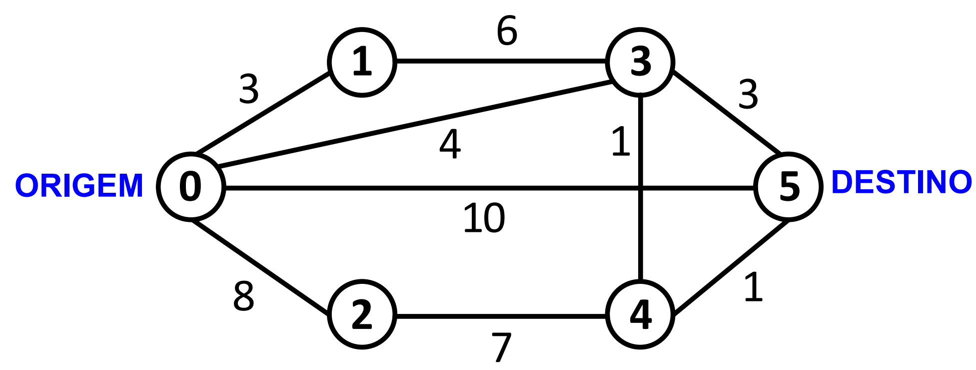
$custo$[2] = min(8, 4 + A[3, 2]) = min(8, 4+∞) = 8

$custo$[4] = min(∞, 4 + A[3, 4]) = min(∞, 4+1) = 5 (atualizar custo/rota)

$custo$[5] = min(10, 4 + A[3, 5]) = min(10, 4+3) = 7 (atualizar custo/rota)

Exemplo

```
dijkstra(G, vInicio, vFim)
for i = 0 até |V| do
    custo[i] = ∞;
    rota[i] = vInicio;
end
custo[vInicio] = 0;
A = V;
F = ∅;
while A ≠ ∅ do
    v = v ∈ A com menor custo[v] entre todos;
    F = F ∪ {v};
    A = A − {v};
    N = N − F;
    for cada u ∈ N do
        if custo[v] + w_vu < custo[u] then
            custo[u] = custo[v] + w_vu;
            rota[u] = v;
        end
    end
end
caminho = obtemCaminho(rota, vInicio, vFim);
caminho.reverse;
return caminho, custo[vFim]
```



	~~0~~	~~1~~	2	3	4	5
custo	0	3	8	4	5	7
rota	-	0	0	0	3	3

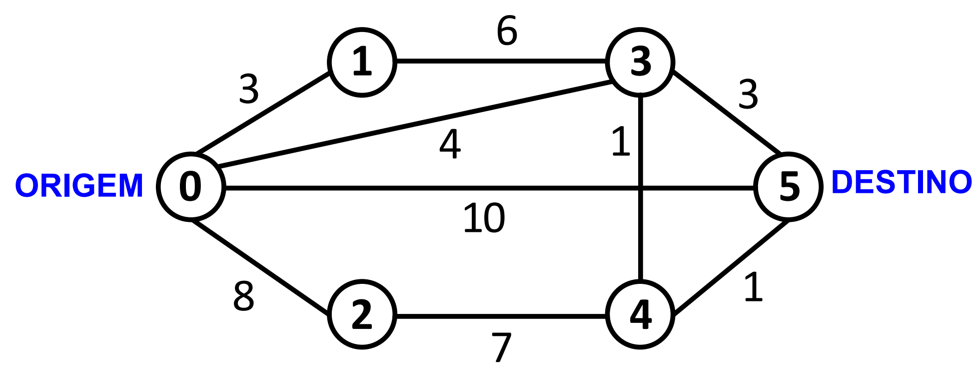
v = 3 (menor) F = {0, 1, 3}

$custo$[2] = min(8, 4 + A[3, 2]) = min(8, 4+∞) = 8
$custo$[4] = min(∞, 4 + A[3, 4]) = min(∞, 4+1) = 5 (atualizar custo/rota)
$custo$[5] = min(10, 4 + A[3, 5]) = min(10, 4+3) = 7 (atualizar custo/rota)

4.1. Dijkstra

Exemplo

```
dijkstra(G, vInicio, vFim)
for i = 0 até |V| do
    custo[i] = ∞;
    rota[i] = vInicio;
end
custo[vInicio] = 0;
A = V;
F = ∅;
while A ≠ ∅ do
    v = v ∈ A com menor custo[v] entre todos;
    F = F ∪ {v};
    A = A − {v};
    N = N − F;
    for cada u ∈ N do
        if custo[v] + w_vu < custo[u] then
            custo[u] = custo[v] + w_vu;
            rota[u] = v;
        end
    end
end
caminho = obtemCaminho(rota, vInicio, vFim);
caminho.reverse;
return caminho, custo[vFim]
```



	~~0~~	~~1~~	2	~~3~~	4	5
custo	0	3	8	4	5	7
rota	-	0	0	0	3	3

v = 3 (menor) F = {0, 1, 3}

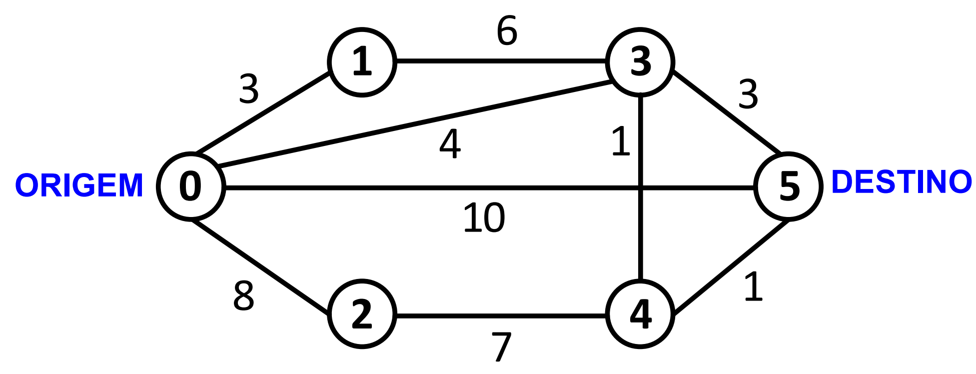
$custo$[2] = min(8, 4 + A[3, 2]) = min(8, 4+∞) = 8
$custo$[4] = min(∞, 4 + A[3, 4]) = min(∞, 4+1) = 5 (atualizar custo/rota)
$custo$[5] = min(10, 4 + A[3, 5]) = min(10, 4+3) = 7 (atualizar custo/rota)

v = 4 (menor) F = {0, 1, 3, 4}

$custo$[2] = min(8, 5 + A[4, 2]) = min(8, 5 + 7) = 8
$custo$[5] = min(7, 5 + A[4, 5]) = min(7, 5 + 1) = 6 (atualizar custo/rota)

Exemplo

```
dijkstra(G, vInicio, vFim)
for i = 0 até |V| do
    custo[i] = ∞;
    rota[i] = vInicio;
end
custo[vInicio] = 0;
A = V;
F = ∅;
while A ≠ ∅ do
    v = v ∈ A com menor custo[v] entre todos;
    F = F ∪ {v};
    A = A − {v};
    N = N − F;
    for cada u ∈ N do
        if custo[v] + w_vu < custo[u] then
            custo[u] = custo[v] + w_vu;
            rota[u] = v;
        end
    end
end
caminho = obtemCaminho(rota, vInicio, vFim);
caminho.reverse;
return caminho, custo[vFim]
```



	~~0~~	~~1~~	2	~~3~~	4	5
custo	0	3	8	4	5	6
rota	-	0	0	0	3	4

v = 3 (menor) F = {0, 1, 3}

$custo$[2] = min(8, 4 + A[3, 2]) = min(8, 4+∞) = 8
$custo$[4] = min(∞, 4 + A[3, 4]) = min(∞, 4+1) = 5 (atualizar custo/rota)
$custo$[5] = min(10, 4 + A[3, 5]) = min(10, 4+3) = 7 (atualizar custo/rota)

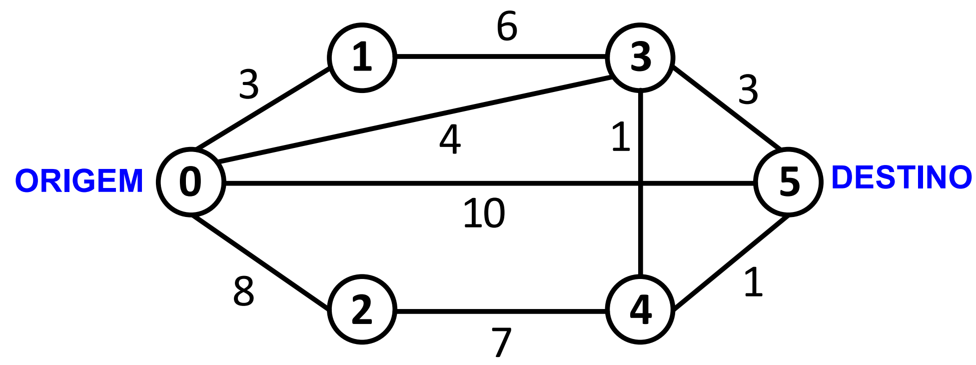
v = 4 (menor) F = {0, 1, 3, 4}

$custo$[2] = min(8, 5 + A[4, 2]) = min(8, 5 + 7) = 8
$custo$[5] = min(7, 5 + A[4, 5]) = min(7, 5 + 1) = 6 (atualizar custo/rota)

Exemplo

```
dijkstra(G, vInicio, vFim)

for i = 0 até |V| do
    custo[i] = ∞;
    rota[i] = vInicio;
end
custo[vInicio] = 0;
A = V;
F = ∅;
while A ≠ ∅ do
    v = v ∈ A com menor custo[v] entre todos;
    F = F ∪ {v};
    A = A − {v};
    N = N − F;
    for cada u ∈ N do
        if custo[v] + w_vu < custo[u] then
            custo[u] = custo[v] + w_vu;
            rota[u] = v;
        end
    end
end
caminho = obtemCaminho(rota, vInicio, vFim);
caminho.reverse;
return caminho, custo[vFim]
```



	~~0~~	~~1~~	2	~~3~~	~~4~~	5
custo	0	3	8	4	5	6
rota	-	0	0	0	3	4

v = 5 (menor) F = {0, 1, 3, 4, 5}

$custo$[2] = min(8, 6 + ∞) = 8

Para obter o caminho mínimo inicie pelo vértice destino e percorra a rota dos precedentes até alcançar a origem (5 ⟶ 4 ⟶ 3 ⟶ 0).

O caminho mínimo entre 0 e 5 é: 0, 3, 4, 5.

Comentários

- O algoritmo de Dijkstra possui complexidade $O(n^2)$.

CRÍTICAS AO ALGORITMO

- O algoritmo é incapaz de calcular os caminhos mínimos caso existam arestas com custo negativo.

- O algoritmo só calcula os caminhos mínimos a partir de uma única origem.

Comentários

- O algoritmo de Dijkstra possui complexidade $O(n^2)$.

CRÍTICAS AO ALGORITMO

- O algoritmo é incapaz de calcular os caminhos mínimos caso existam arestas com custo negativo.

- O algoritmo só calcula os caminhos mínimos a partir de uma única origem.

- Para calcular os caminhos mínimos de todos os vértices para todos os vértices, o algoritmo deve ser executado uma vez para cada vértice do grafo, com complexidade total $O(n^3)$ na implementação simples.

- Se for utilizada uma estrutura *Heap* e listas de adjacências na representação do grafo a complexidade é reduzida para $O((V + E)\ logE)$ pois determinar o menor elemento e atualizar a *Heap* pode ser feito em tempo logarítmico.

- Melhor implementação (ano 2000) possui complexidade $O(nlog\ logm)$.

Algoritmo de Bellman-Ford

Arestas de peso negativo: Além das distâncias geográficas, caminhos mínimos podem modelar situações reais que necessitam de arestas com pesos negativos.

- Movimentações financeiras, nas quais pode ocorrer a obtenção de lucro ou prejuízo, principalmente quando em operações de câmbio;
- Um taxista que recebe mais dinheiro do que gasta com combustível a cada viagem: se o táxi roda vazio, ele gasta mais do que recebe;
- Um entregador que necessita atravessar um pedágio e pode acabar pagando mais do que recebe para entregar encomendas;
- A energia gerada e consumida durante uma reação química.

Algoritmo de Bellman-Ford

Arestas de peso negativo: Além das distâncias geográficas, caminhos mínimos podem modelar situações reais que necessitam de arestas com pesos negativos.

- Movimentações financeiras, nas quais pode ocorrer a obtenção de lucro ou prejuízo, principalmente quando em operações de câmbio;
- Um taxista que recebe mais dinheiro do que gasta com combustível a cada viagem: se o táxi roda vazio, ele gasta mais do que recebe;
- Um entregador que necessita atravessar um pedágio e pode acabar pagando mais do que recebe para entregar encomendas;
- A energia gerada e consumida durante uma reação química.

Alguns autores denominam o algoritmo de Ford-Moore-Bellman, em homenagem a outros três autores que propuseram o mesmo algoritmo em anos diferentes: Lester Ford (1956), Edward Moore (1957), Richard Bellman (1958).

Princípio de Funcionamento

- Calcula caminhos mais curtos via programação dinâmica *bottom-up*.
- Ao invés de fechar um vértice por iteração como faz Dijkstra, examina todos os vértices de um grafo orientado por iteração até que atualizações não sejam mais possíveis.
- Em um grafo com n vértices, qualquer caminho possui no máximo n–1 arestas, portanto, cada vértice é examinado no máximo n–1 vezes. Com esta estratégia, é possível calcular caminhos mínimos em grafos com arestas de peso negativo.

Princípio de Funcionamento

- Calcula caminhos mais curtos via programação dinâmica *bottom-up*.
- Ao invés de fechar um vértice por iteração, como faz Dijkstra, examina todos os vértices de um grafo orientado por iteração até que atualizações não sejam mais possíveis.
- Em um grafo com n vértices, qualquer caminho possui no máximo n–1 arestas, portanto, cada vértice é examinado no máximo n–1 vezes. Com esta estratégia, é possível calcular caminhos mínimos em grafos com arestas de peso negativo.
- Assim como o algoritmo de Dijkstra, baseia-se no princípio de relaxação: uma aproximação da distância da origem até cada vértice é gradualmente atualizada por valores mais baixos até que a solução seja obtida.
- Se, em alguma iteração do algoritmo os caminhos até cada um dos vértices permanecerem inalterados, não haverá atualizações nas próximas iterações e o algoritmo pode terminar.
- Entretanto, se houver atualizações na última iteração do algoritmo, é sinal de que há pelo menos um ciclo negativo no grafo, dado que algum caminho terá n arestas ou mais.

Algoritmo

```
bellmanFord(G, vInicio, vFim)

for i = 0 até |V| do
    custo[i] = ∞;
    rota[i] = vInicio;
end
custo[vInicio] = 0;
for i = 0 até |V| do
    for cada aresta (v, u) ∈ E do
        if custo[u] > custo[v] + w_vu then
            custo[u] = custo[v] + w_vu;
            rota[u] = v;
        end
    end
end
for cada aresta (v, u) ∈ E do
    if custo[u] > custo[v] + w_vu then
        return False
    end
end
caminho = obtemCaminho(rota, vInicio, vFim);
caminho.reverse;
return caminho, custo[vFim]
```

ESTRUTURAS

- ***custo*[*v*]** : vetor que armazena o custo (distância) entre o vértice origem e o vértice v.
- ***rota*[*v*]** : vetor que armazena o índice do vértice que precede o vértice v, no caminho cuja distância consta no vetor $custo[v]$.
- w_{vu} : valor do peso da aresta ou arco que conecta v e u.

Algoritmo

```
bellmanFord(G, vInicio, vFim)
1  for i = 0 até |V| do
2      custo[i] = ∞;
3      rota[i] = vInicio;
4  end
5  custo[vInicio] = 0;
6  for i = 0 até |V| do
7      for cada aresta (v, u) ∈ E do
8          if custo[u] > custo[v] + w_vu then
9              custo[u] = custo[v] + w_vu;
10             rota[u] = v;
11         end
12     end
13 end
14 for cada aresta (v, u) ∈ E do
15     if custo[u] > custo[v] + w_vu then
16         return False
17     end
18 end
19 caminho = obtemCaminho(rota, vInicio, vFim);
20 caminho.reverse;
21 return caminho, custo[vFim]
```

ESTRUTURAS

- ***custo*[*v*]** : vetor que armazena o custo (distância) entre o vértice origem e o vértice v.
- ***rota*[*v*]** : vetor que armazena o índice do vértice que precede o vértice v, no caminho cuja distância consta no vetor $custo[v]$.
- w_{vu} : valor do peso da aresta ou arco que conecta v e u.

Os vetores de custo e rota são inicializados considerando o vértice de id 0 como origem (linhas 1 a 5).

Considerando o caminho de s para cada vértice de G. Para cada aresta (v, u):

Se o custo de adicionar o vértice u no caminho for maior que o custo do vértice v mais o peso da aresta que conecta os vértices v e u (linha 8).

Atualiza o custo do vértice u para o custo do vértice v mais o peso da aresta (v, u).

Atualiza a rota do vértice u considerando o vértice v como precedente.

Algoritmo

```
bellmanFord(G, vInicio, vFim)
1  for i = 0 até |V| do
2      custo[i] = ∞;
3      rota[i] = vInicio;
4  end
5  custo[vInicio] = 0;
6  for i = 0 até |V| do
7      for cada aresta (v, u) ∈ E do
8          if custo[u] > custo[v] + w_vu then
9              custo[u] = custo[v] + w_vu;
10             rota[u] = v;
11         end
12     end
13 end
14 for cada aresta (v, u) ∈ E do
15     if custo[u] > custo[v] + w_vu then
16         return False
17     end
18 end
19 caminho = obtemCaminho(rota, vInicio, vFim);
20 caminho.reverse;
21 return caminho, custo[vFim]
```

ESTRUTURAS

- ***custo*[*v*]** : vetor que armazena o custo (distância) entre o vértice origem e o vértice v.
- ***rota*[*v*]** : vetor que armazena o índice do vértice que precede o vértice v, no caminho cuja distância consta no vetor $custo[v]$.
- w_{vu} : valor do peso da aresta ou arco que conecta v e u.

	0	1	2	3	4
custo	0	∞	∞	∞	∞
rota	-	0	0	0	0

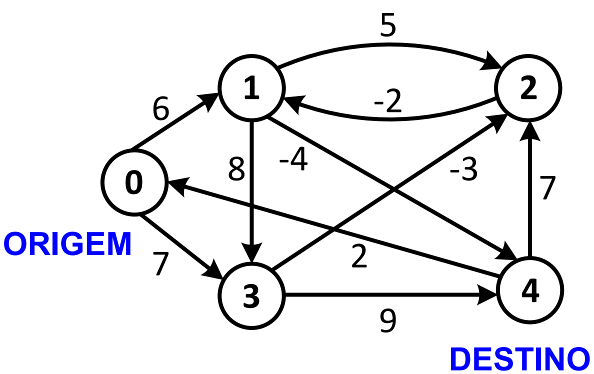
Para cada aresta (v, u) verifica se o grafo possui um ciclo de peso negativo que seja acessível a partir do vértice origem.

Caso exista ao menos um ciclo negativo o algoritmo retorna False (linhas 14 a 18).

Caso não exista nenhum ciclo negativo o algoritmo obtém o caminho e o retorna junto com o custo (linhas 19 a 21).

Exemplo

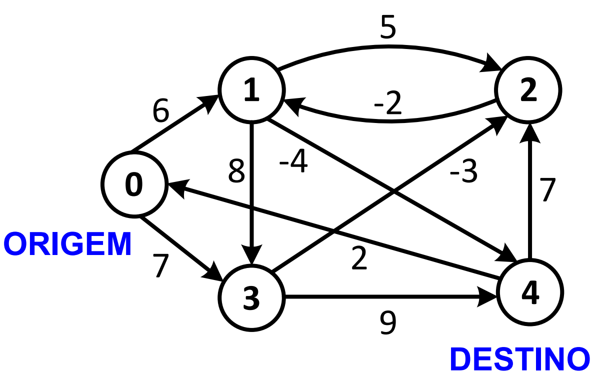
E = (0,1), (0,3), (1,2), (1,3), (1,4), (2,1), (3,2), (3,4), (4,0), (4,2)



	0	1	2	3	4
custo	0	∞	∞	∞	∞
rota	-	0	0	0	0

```
bellmanFord(G, vInicio, vFim)
for i = 0 até |V| do
    custo[i] = ∞;
    rota[i] = vInicio;
end
custo[vInicio] = 0;
for i = 0 até |V| do
    for cada aresta (v, u) ∈ E do
        if custo[u] > custo[v] + w_vu then
            custo[u] = custo[v] + w_vu;
            rota[u] = v;
        end
    end
end
for cada aresta (v, u) ∈ E do
    if custo[u] > custo[v] + w_vu then
        return False
    end
end
caminho = obtemCaminho(rota, vInicio, vFim);
caminho.reverse;
return caminho, custo[vFim]
```

Exemplo




```
bellmanFord(G, vInicio, vFim)
for i = 0 até |V| do
    custo[i] = ∞;
    rota[i] = vInicio;
end
custo[vInicio] = 0;
for i = 0 até |V| do
    for cada aresta (v,u) ∈ E do
        if custo[u] > custo[v] + w_vu then
            custo[u] = custo[v] + w_vu;
            rota[u] = v;
        end
    end
end
for cada aresta (v,u) ∈ E do
    if custo[u] > custo[v] + w_vu then
        return False
    end
end
caminho = obtemCaminho(rota, vInicio, vFim);
caminho.reverse;
return caminho, custo[vFim]
```

E = (0,1), (0,3), (1,2), (1,3), (1,4), (2,1), (3,2), (3,4), (4,0), (4,2)

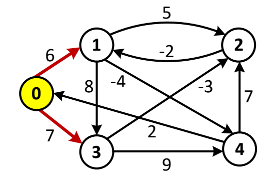
PASSO 1

	0	1	2	3	4
custo	0	6	∞	7	∞
rota	0	0	0	0	0

v = 0

custo[1] = min(∞, 0 + 6) = 6 (atualiza custo e rota)

custo[3] = min(∞, 0 + 7) = 7 (atualiza custo e rota)



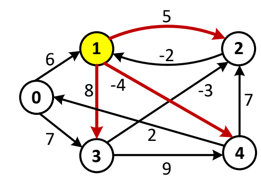
	0	1	2	3	4
custo	0	6	11	7	2
rota	0	0	1	0	1

v = 1

custo[2] = min(∞, 6 + 5) = 11 (atualiza custo e rota)

custo[3] = min(7, 6 + 8) = 7

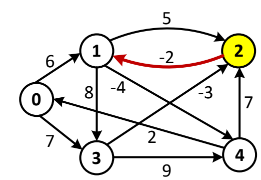
custo[4] = min(∞, 6 + (-4)) = 2 (atualiza custo e rota)



	0	1	2	3	4
custo	0	6	11	7	2
rota	0	0	1	0	1

v = 2

custo[1] = min(6, 11 + (-2)) = 6

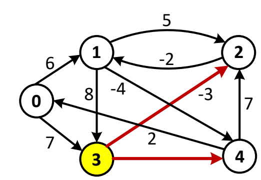


	0	1	2	3	4
custo	0	6	4	7	2
rota	0	0	3	0	1

v = 3

custo[2] = min(11, 7 + (-3)) = 4 (atualiza custo e rota)

custo[4] = min(2, 7 + 9) = 2

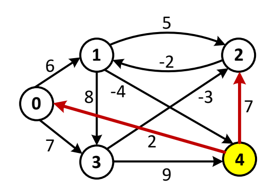


	0	1	2	3	4
custo	0	6	4	7	2
rota	0	0	3	0	1

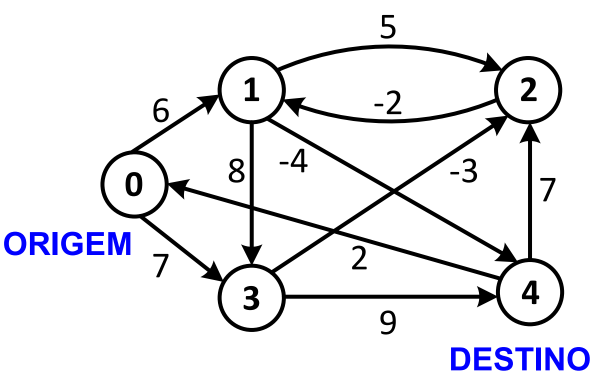
v = 4

custo[0] = min(0, 2 + 2) = 0

custo[2] = min(4, 2 + 7) = 4



Exemplo




```
bellmanFord(G, vInicio, vFim)
for i = 0 até |V| do
    custo[i] = ∞;
    rota[i] = vInicio;
end
custo[vInicio] = 0;
for i = 0 até |V| do
    for cada aresta (v, u) ∈ E do
        if custo[u] > custo[v] + w_vu then
            custo[u] = custo[v] + w_vu;
            rota[u] = v;
        end
    end
end
for cada aresta (v, u) ∈ E do
    if custo[u] > custo[v] + w_vu then
        return False
    end
end
caminho = obtemCaminho(rota, vInicio, vFim);
caminho.reverse;
return caminho, custo[vFim]
```

E = (0,1), (0,3), (1,2), (1,3), (1,4), (2,1), (3,2), (3,4), (4,0), (4,2)

PASSO 1

	0	1	2	3	4
custo	0	6	∞	7	∞
rota	0	0	0	0	0

v = 0

custo[1] = min(∞, 0 + 6) = **6** (atualiza custo e rota)
custo[3] = min(∞, 0 + 7) = **7** (atualiza custo e rota)

	0	1	2	3	4
custo	0	6	11	7	2
rota	0	0	1	0	1

v = 1

custo[2] = min(∞, 6 + 5) = **11** (atualiza custo e rota)
custo[3] = min(7, 6 + 8) = 7
custo[4] = min(∞, 6 + (-4)) = **2** (atualiza custo e rota)

	0	1	2	3	4
custo	0	6	11	7	2
rota	0	0	1	0	1

v = 2

custo[1] = min(6, 11 + (-2)) = 6

	0	1	2	3	4
custo	0	6	4	7	2
rota	0	0	3	0	1

v = 3

custo[2] = min(11, 7 + (-3)) = **4** (atualiza custo e rota)
custo[4] = min(2, 7 + 9) = 2

	0	1	2	3	4
custo	0	6	4	7	2
rota	0	0	3	0	1

v = 4

custo[0] = min(0, 2 + 2) = 0
custo[2] = min(4, 2 + 7) = 4

PASSO 2

	0	1	2	3	4
custo	0	6	4	7	2
rota	0	0	3	0	1

v = 0

custo[1] = min(6, 0 + 6) = 6
custo[3] = min(7, 0 + 7) = 7

	0	1	2	3	4
custo	0	6	4	7	2
rota	0	0	3	0	1

v = 1

custo[2] = min(4, 6 + 5) = 4
custo[3] = min(7, 6 + 8) = 7
custo[4] = min(2, 6 + (-4)) = 2

	0	1	2	3	4
custo	0	2	4	7	2
rota	0	2	3	0	1

v = 2

custo[1] = min(6, 4 + (-2)) = **2** (atualiza custo e rota)

	0	1	2	3	4
custo	0	2	4	7	2
rota	0	2	3	0	1

v = 3

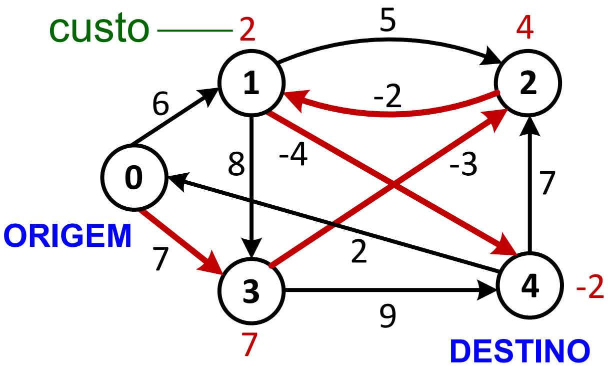
custo[2] = min(4, 7 + (-3)) = 4
custo[4] = min(2, 7 + 9) = 2

	0	1	2	3	4
custo	0	2	4	7	2
rota	0	2	3	0	1

v = 4

custo[0] = min(0, 2 + 2) = 0
custo[2] = min(4, 2 + 7) = 4

Exemplo




```
bellmanFord(G, vInicio, vFim)
for i = 0 até |V| do
    custo[i] = ∞;
    rota[i] = vInicio;
end
custo[vInicio] = 0;
for i = 0 até |V| do
    for cada aresta (v, u) ∈ E do
        if custo[u] > custo[v] + w_vu then
            custo[u] = custo[v] + w_vu;
            rota[u] = v;
        end
    end
end
for cada aresta (v, u) ∈ E do
    if custo[u] > custo[v] + w_vu then
        return False
    end
end
caminho = obtemCaminho(rota, vInicio, vFim);
caminho.reverse;
return caminho, custo[vFim]
```

E = (0,1), (0,3), (1,2), (1,3), (1,4), (2,1), (3,2), (3,4), (4,0), (4,2)

PASSO 3

	0	1	2	3	4
custo	0	2	4	7	2
rota	0	2	3	0	1

v = 0

custo[1] = min(2, 0 + 6) = 2

custo[3] = min(7, 0 + 7) = 7

	0	1	2	3	4
custo	0	2	4	7	-2
rota	0	2	3	0	1

v = 1

custo[2] = min(4, 2 + 5) = 4

custo[3] = min(7, 2 + 8) = 7

custo[4] = min(2, 2 + (-4)) = -2 (atualiza custo e rota)

	0	1	2	3	4
custo	0	2	4	7	-2
rota	0	2	3	0	1

v = 2

custo[1] = min(2, 4 + (-2)) = 2

	0	1	2	3	4
custo	0	2	4	7	-2
rota	0	2	3	0	1

v = 3

custo[2] = min(4, 7 + (-3)) = 4

custo[4] = min(-2, 7 + 9) = -2

	0	1	2	3	4
custo	0	2	4	7	-2
rota	0	2	3	0	1

v = 4

custo[0] = min(0, -2 + 2) = 0

custo[2] = min(4, -2 + 7) = 4

PASSO 4

	0	1	2	3	4
custo	0	2	4	7	-2
rota	0	2	3	0	1

v = 0

custo[1] = min(2, 0 + 6) = 2

custo[3] = min(7, 0 + 7) = 7

	0	1	2	3	4
custo	0	2	4	7	-2
rota	0	2	3	0	1

v = 1

custo[2] = min(4, 2 + 5) = 4

custo[3] = min(7, 2 + 8) = 7

custo[4] = min(-2, 2 + (-4)) = -2

	0	1	2	3	4
custo	0	2	4	7	-2
rota	0	2	3	0	1

v = 2

custo[1] = min(2, 4 + (-2)) = 2

	0	1	2	3	4
custo	0	2	4	7	-2
rota	0	2	3	0	1

v = 3

custo[2] = min(4, 7 + (-3)) = 4

custo[4] = min(-2, 7 + 9) = -2

	0	1	2	3	4
custo	0	2	4	7	-2
rota	0	2	3	0	1

v = 4

custo[0] = min(0, -2 + 2) = 0

custo[2] = min(4, -2 + 7) = 4

Comentários

O algoritmo de Bellman-Ford possui complexidade $O(n^3)$.

Uma versão proposta por Yen (1970) possui complexidade $O(nm)$ no pior caso:

- Se $custo[v]$ não se alterar desde a última vez que seus antecessores foram analisados, então não é necessário examinar novamente seus arcos de saída.
- O comprimento do laço externo é reduzido de n–1 para n/2 através de uma ordenação linear dos vértices e posterior partição dos mesmos.
- Yen, Jin Y. (1970). "*An algorithm for finding shortest routes from all source nodes to a given destination in general networks*". Quarterly of Applied Mathematics 27: 526–530.

Perguntas? Sugestões?