Teoria dos Grafos

Menor Caminho de Origem Única

Menor Caminho com Origem Única

- Problema: encontrar o caminho, mais curto possível, entre Campina Grande e Natal.
- Suponha que temos um mapa com todas as estradas do Nordeste, com a distância entre todos os pares de cidades adjacentes.
- Como determinar a menor distância entre as duas cidades?

Uma maneira de resolver este problema é encontrar a menor distância entre Campina Grande e todas as outras cidades do Nordeste.

Em grafos: Menor Caminho com Origem Única

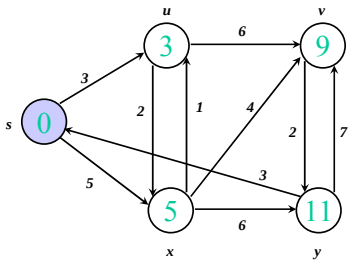
Menor Caminho com Origem Única

- Consideramos um grafo dirigido $G = (V, E)$, com uma função peso $w(u, v): E \rightarrow \mathbb{R}$, que mapeia pesos a arestas.
- O peso do caminho $p = \langle v_0, v_1, \ldots, v_k \rangle$ é a soma dos pesos das arestas que o constituem:
  - $w(p) = \sum_{i=1}^{k} w(v_{i-1}, w_i)$
- Definimos o menor caminho entre u e v como:
  - $\delta(u, v) = min\{w(p): u \Rightarrow v\}$, se existir um caminho.
  - $\delta(u, v) = \infty$, se não existir caminho.

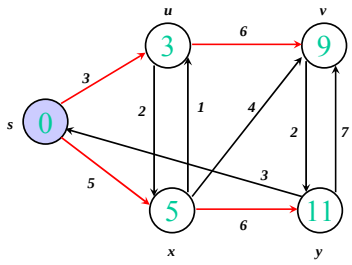
Menor Caminho com Origem Única

- O problema de menor caminho em um grafo consiste em determinar o menor caminho entre um vértice inicial $s \in V$ e todos os demais vértices de V.
- Algumas variações do problema:
  - Menor caminho com destino único.
  - Menor caminho entre um par de vértices.
  - Menor caminho entre todos os pares de vértices.
- Em algumas instância é possível encontrar pesos negativos.

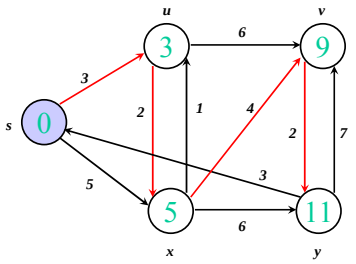
Exemplo de Menor Caminho



Exemplo de Menor Caminho



Exemplo de Menor Caminho



Algoritmo de Dijkstra

- Assumir que qualquer cidade é infinitamente distante da cidade origem:
- Abordagem otimista:
  - Sempre que chegar em uma cidade, assumir que encontrou o menor caminho.
  - Se depois encontra algo melhor, atualizar os dados.
- A partir da cidade de origem, visitar primeiro as cidades adjacentes, depois as adjacentes das adjacentes, e assim por diante.
- Algoritmo bastante conhecido, utilizado no OSPF.

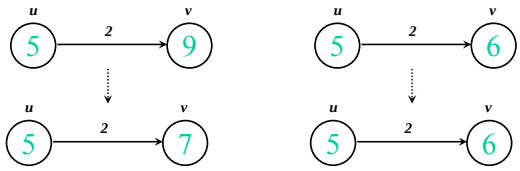
Algoritmo de Dijkstra

- Não considera pesos negativos.
- Uso de duas variáveis auxiliares:
  - *d[v]* (estimativa de menor distância)
  - *π[v]* (predecessor)

```
IniciaMenorCaminho(G, s)
for ∀u ∈ V[G] do
        d[u] ← ∞
        π[u] ← NIL
d[s] ← 0
```

Algoritmo de Dijkstra

- Atualização dos dados utiliza a técnica de relaxamento.

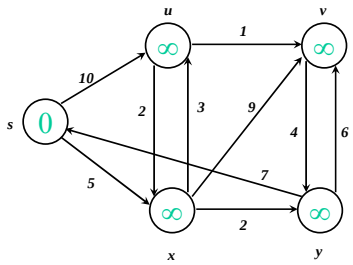



```
Relaxa(u, v, w)
if d[v] > d[u] + w(u,v) then
        d[v] ← d[u] + w(u,v)
        π[v] ← u
```

Algoritmo de Dijkstra

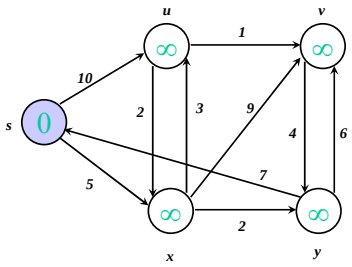
```
Dijkstra(G, w, s)
IniciaMenorCaminho(G, s)
S ← ∅
Q ← V[G]
while Q ≠ ∅ do
        u ← ExtraiMenor[Q]
        S ← S ∪ {u}
        for ∀v ∈ Adj[u] do
                Relaxa(u, v, w)
```

Execução do Algoritmo de Dijkstra



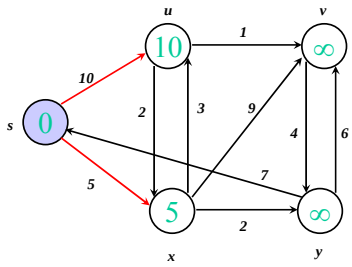
Q: | s | u | v | x | y |

Execução do Algoritmo de Dijkstra



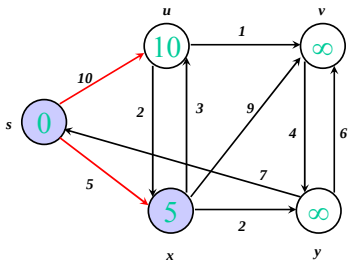
Q: | *u* | *v* | *x* | *y* |

Execução do Algoritmo de Dijkstra



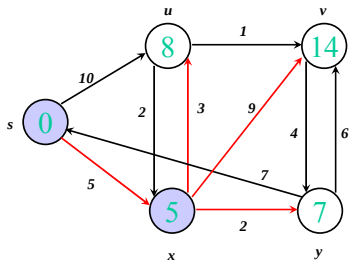
Q: | *x* | *u* | *v* | *y* |

Execução do Algoritmo de Dijkstra



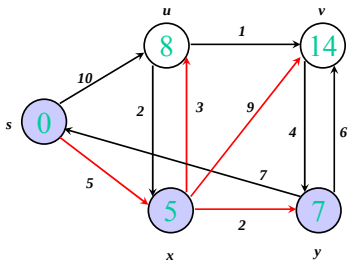
Q: | *u* | *v* | *y* |

Execução do Algoritmo de Dijkstra



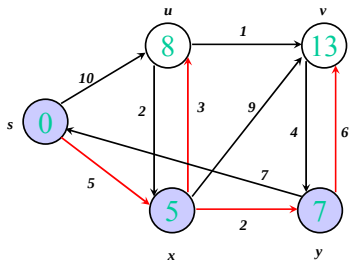
Q: | *y* | *u* | *v* |

Execução do Algoritmo de Dijkstra



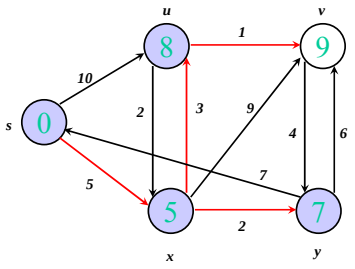
Q: | *u* | *v* |

Execução do Algoritmo de Dijkstra



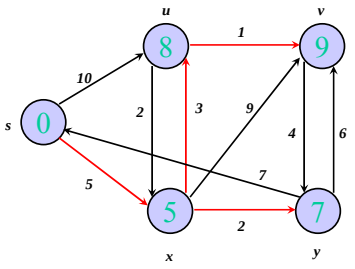
Q: | *u* | *v* |

Execução do Algoritmo de Dijkstra



Q: v

Execução do Algoritmo de Dijkstra



Q: Ø

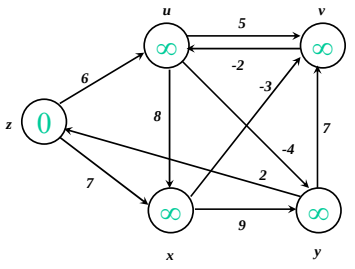
Algoritmo Bellman-Ford

- Resolve menor caminho de forma mais genérica, incluindo pesos negativos.
- Verifica a existência de ciclos de peso negativo:
  - Se existir, indica que não é possível achar menor caminho.
- Também utiliza a técnica de relaxamento de arestas.
- A idéia principal: progressivamente ir diminuindo a estimativa de distância até encontrar o menor caminho.

Algoritmo de Bellman-Ford

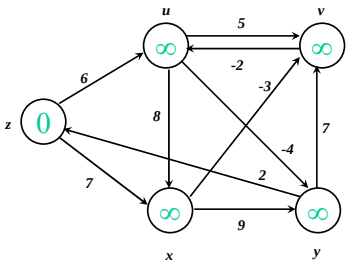
```
Bellman-Ford(G, w, s)
IniciaMenorCaminho(G, s)
for i ← 1 to |V[G]| - 1 do
        for ∀(u, v) ∈ E[G] do
                Relaxa(u, v, w)
for ∀(u, v) ∈ E[G] do
        if d[v] > d[u] + w(u, v) then
                return FALSE
return TRUE
```

Execução do Algoritmo de Bellman-Ford



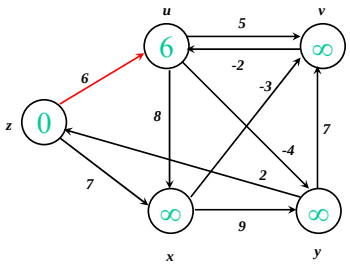
uv, ux, uy, vu, xv, xy, yv, yz, zu, zx

Execução do Algoritmo de Bellman-Ford



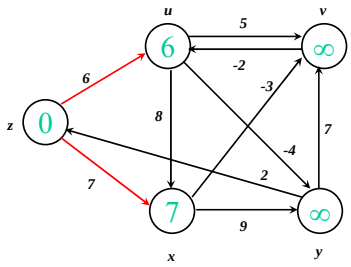
uv, ux, uy, vu, xv, xy, yv, yz, zu, zx

Execução do Algoritmo de Bellman-Ford



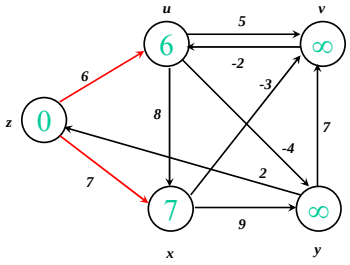
uv, ux, uy, vu, xv, xy, yv, yz, zu, zx

Execução do Algoritmo de Bellman-Ford



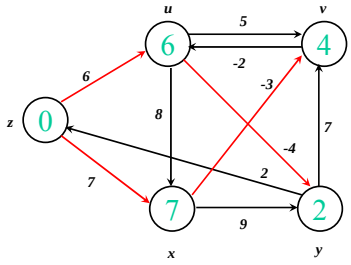
uv, ux, uy, vu, xv, xy, yv, yz, zu, zx

Resultado Após Primeira Passada



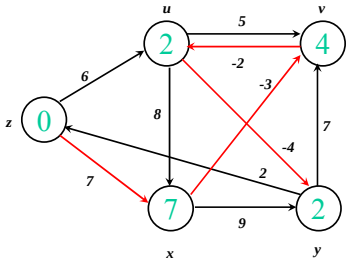
uv, ux, uy, vu, xv, xy, yv, yz, zu, zx

Resultado Após Segunda Passada



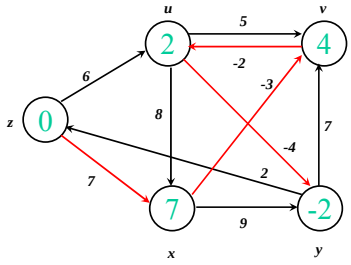
uv, ux, uy, vu, xv, xy, yv, yz, zu, zx

Resultado Após Terceira Passada



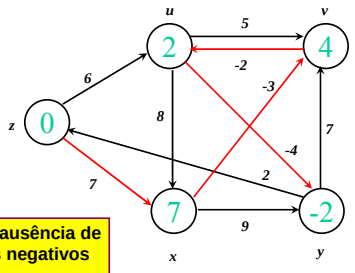
uv, ux, uy, vu, xv, xy, yv, yz, zu, zx

Resultado Após Quarta Passada



uv, ux, uy, vu, xv, xy, yv, yz, zu, zx

Resultado Final



TRUE: ausência de ciclos negativos

uv, ux, uy, vu, xv, xy, yv, yz, zu, zx

Algoritmo de Floyd-Warshall

- Calcula a menor distância entre todos os pares de vértices.
- Considera uma matriz n×n e utiliza a técnica de programação dinâmica.
- Seja A[i,j] = c[i,j] ∀ i,j & i≠j.
- Se (i,j) não é uma aresta, fazer A[i,j]=∞ e A[i,i]=0
- $A_k[i,j] = \min (A_{k-1}[i,j] , A_{k-1}[i,k] + A_{k-1}[k,j])$

Algoritmo de Floyd-Warshall

```
Floyd-Warshall(G, w)
Iniciar A[i,j] = w[i,j]
Iniciar A[i,i] = 0
for k ← 1 to n do
        for i ← 1 to n do
                for j ← to n do
                        if A[i,j] > A[i,k] + A[k,j] then
                                A[i,j] = A[j,k] + A[k,j]
```