Capítulo IX

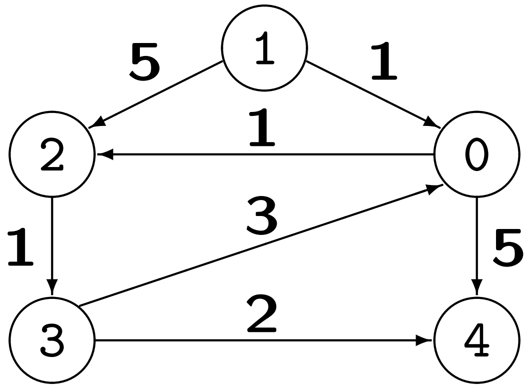
Caminhos Mais Curtos de um vértice a todos os vértices

—

Algoritmo de Dijkstra

Problema

Dado um grafo **orientado** (e pesado) e um vértice o, como encontrar, para cada vértice x para o qual há caminho a partir de o, um **caminho (pesado) mais curto de o para x**?

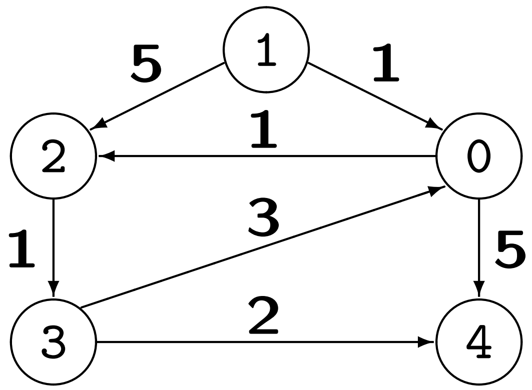


Caminho mais curto de 1 para 2

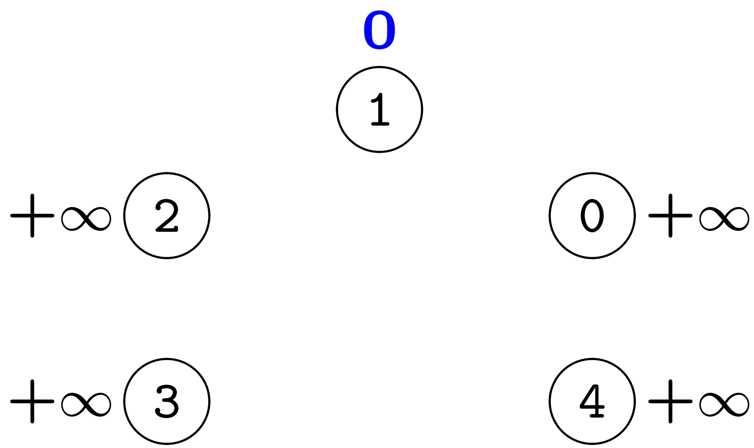
Caminho não pesado:	1, 2	Compr.: 1	Compr. pesado: 5
Caminho pesado:	1, 0, 2	Compr.: 2	Compr. pesado: 2

Restrição: os pesos dos arcos não são negativos.

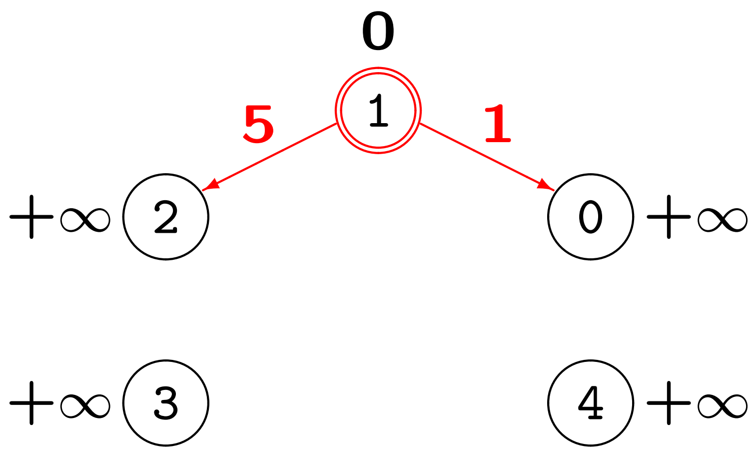
Algoritmo de Dijkstra [1959]

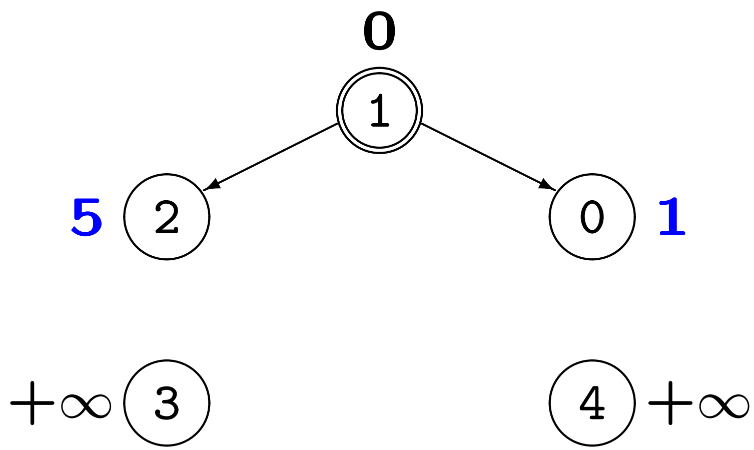


Inicialização origem 1

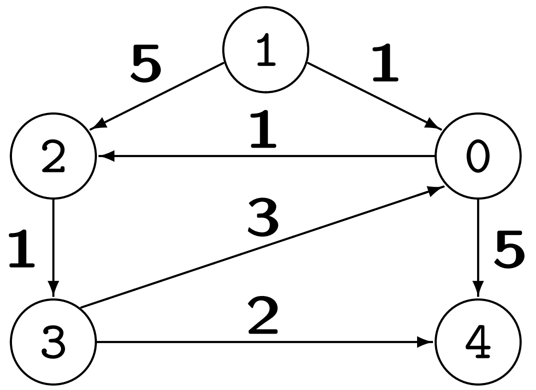


Seleção de 1

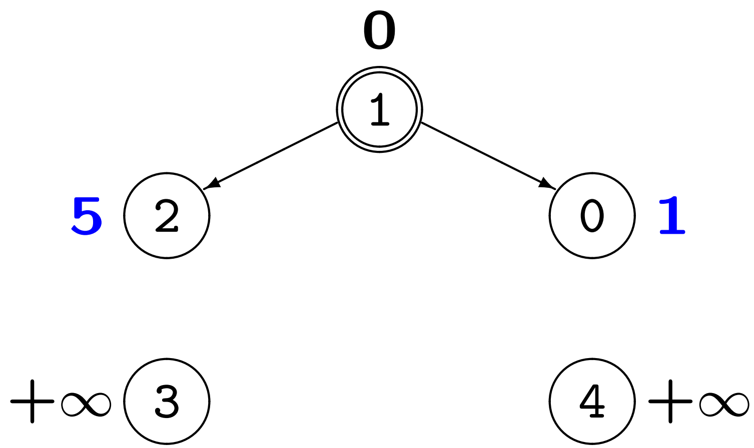




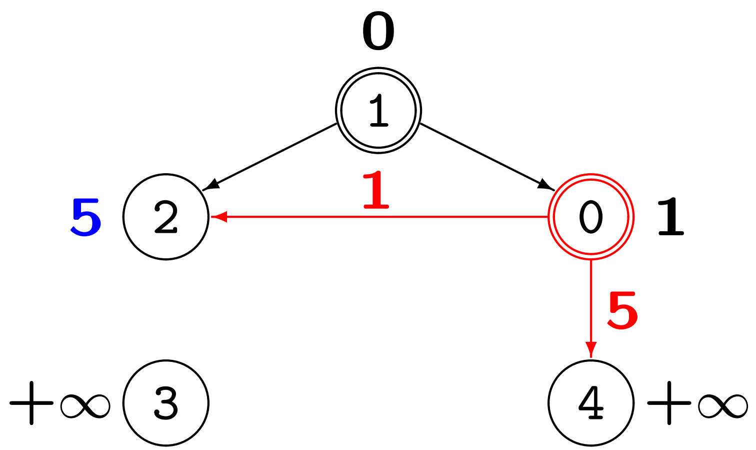
Algoritmo de Dijkstra (2)

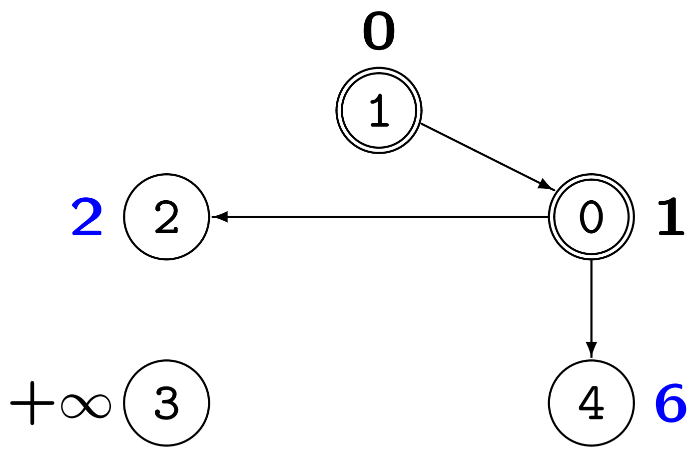


Situação Corrente

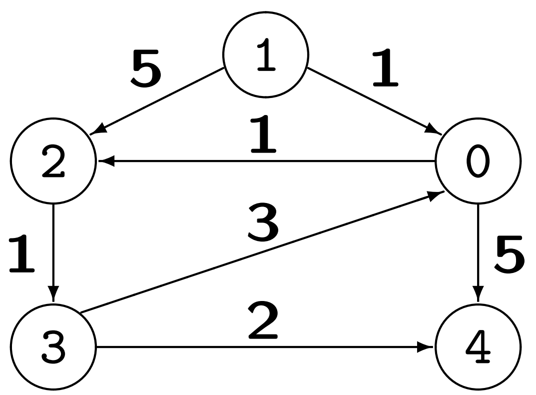


Seleção de 0

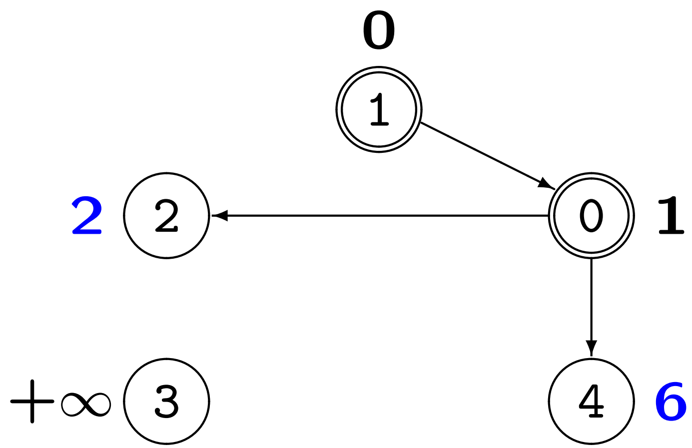




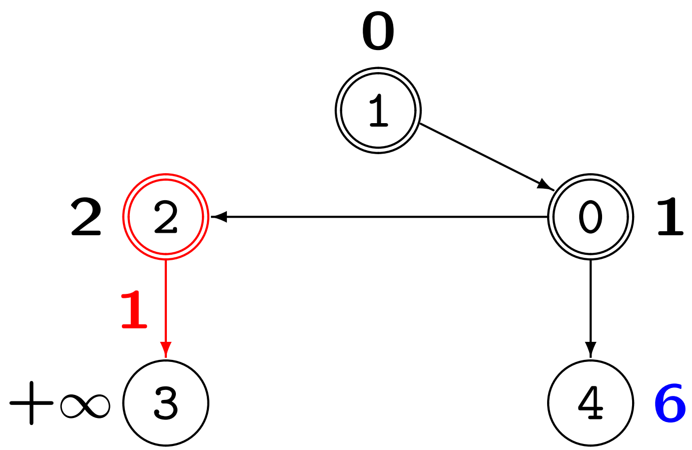
Algoritmo de Dijkstra (3)

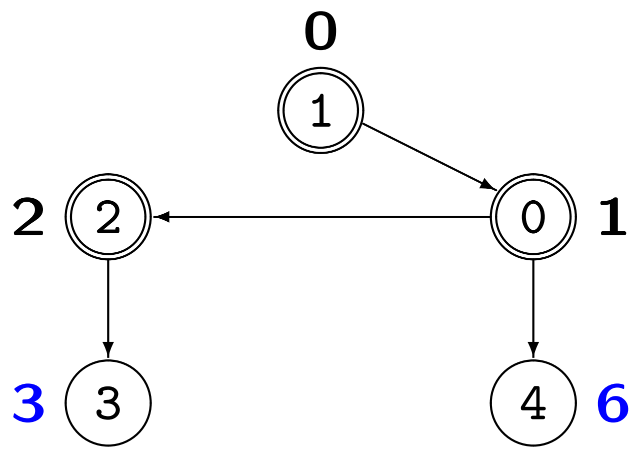


Situação Corrente

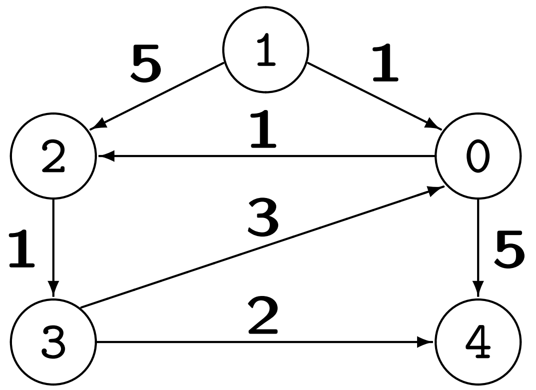


Seleção de 2

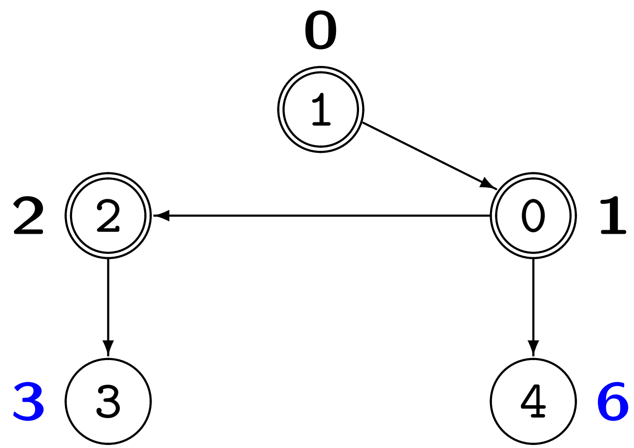




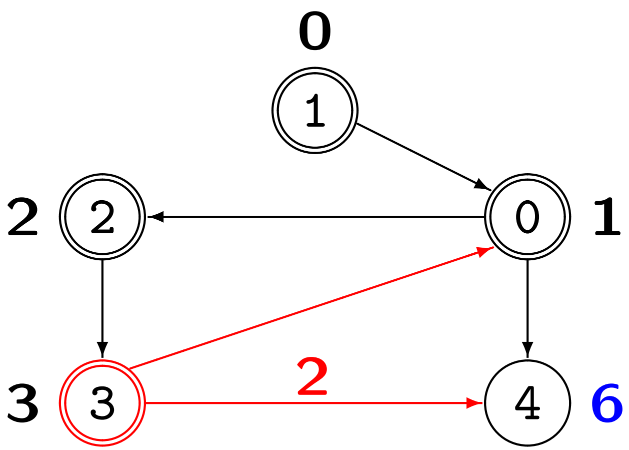
Algoritmo de Dijkstra (4)

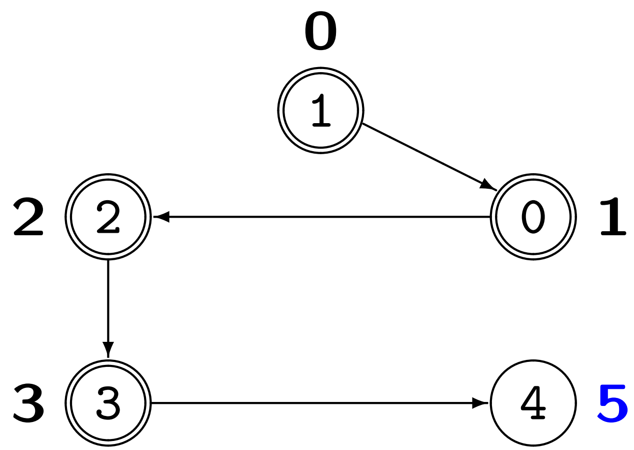


Situação Corrente

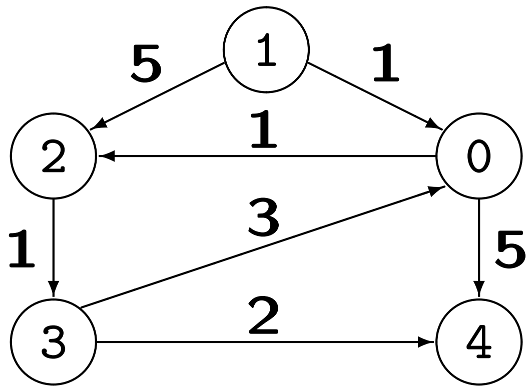


Seleção de 3

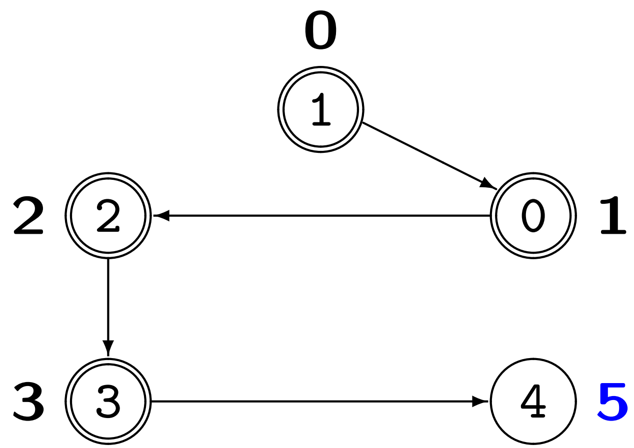




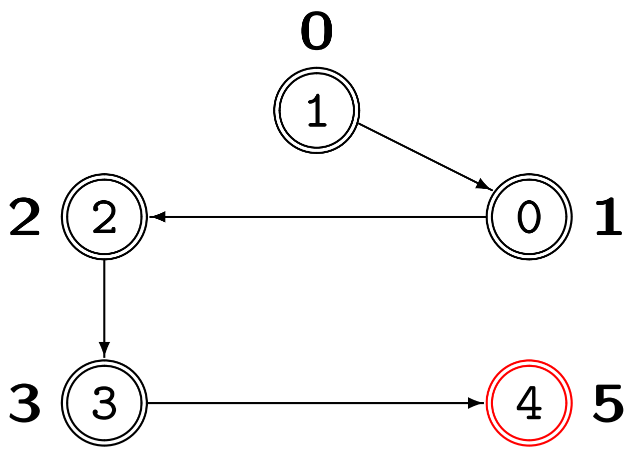
Algoritmo de Dijkstra (5)

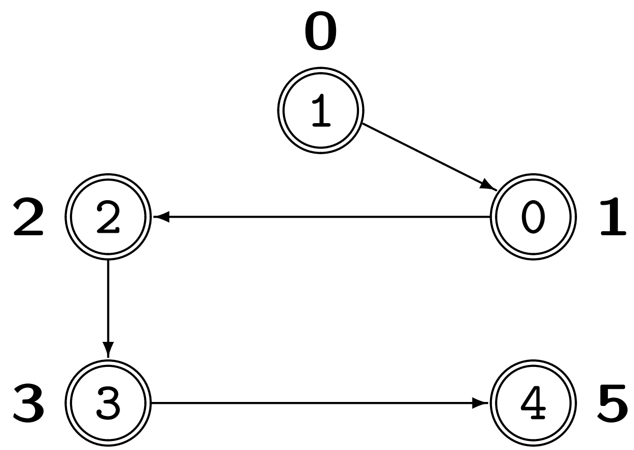


Situação Corrente



Seleção de 4





Ideia Geral

Para todos os vértices x

para os quais há caminho a partir de o,

encontrar um caminho mais curto de o para x,

selecionando, em cada passo, um novo vértice

(e o respetivo caminho).

Informação Necessária

Global: ligados

Conjunto dos vértices nunca selecionados para os quais já há caminho a partir de o.

Por cada vértice x:

- **boolean** selecionado[x]

  Indica se x já foi selecionado, i.e., se já se conhece um caminho mais curto de o para x.

- $\mathbb{R}_0^+ \cup \{+\infty\}$ comprimento[x]

  - **ou** é $+\infty$, quando ainda não há caminho de o para x;
  - **ou** é o comprimento dos caminhos mais curtos de o para x (até ao momento), no caso contrário.

- Node via[x]

  Se estiver definido, indica que um caminho mais curto de o para x (até ao momento) tem a forma o, ..., via[x], x.

Situação Inicial

Global: ligados $= \{o\}$.

Informação para o vértice o:

- selecionado$[o]$ $=$ **false**;
- comprimento$[o]$ $= 0$;
- via$[o]$ $= o$ (poderia ficar indefinido).

Informação para todos os vértices $x \in V \setminus \{o\}$:

- selecionado$[x]$ $=$ **false**;
- comprimento$[x]$ $= +\infty$;
- via$[x]$ não está definido.

Em Cada Iteração

Seleciona-se um vértice x de **ligados** t.q. comprimento$[x]$ é mínimo.

Caminhos Mais Curtos (1)

(Single-source Shortest Paths)

```
Pair<L[], Node[]> dijkstra( Digraph<L> graph, Node origin )
{
    boolean[] selected = new boolean[ graph.numNodes() ];

    L[] length = new L[ graph.numNodes() ];

    Node[] via = new Node[ graph.numNodes() ];

    AdaptMinPriQueue<L, Node> connected =
        new AdaptMinHeap<L, Node>( graph.numNodes() );
```

Caminhos Mais Curtos (2)

```
for every Node v in graph.nodes()
{
    selected[v] = false;
    length[v] = +∞;
}
length[origin] = 0;
via[origin] = origin;
connected.insert(0, origin);
```

Caminhos Mais Curtos (3)

```
    do {
            Node node = connected.removeMin().getValue();
            selected[node] = true;
            exploreNode(graph, node, selected, length, via, connected);
        }
    while ( !connected.isEmpty() );

    return new PairClass<L[], Node[]>(length, via);
}
```

```
void exploreNode( Digraph<L> graph,   Node source,
    boolean[] selected,   L[] length,   Node[] via,
    AdaptMinPriQueue<L, Node> connected )
{
    for every Edge<L> e in graph.outIncidentEdges(source)
    {
        Node node = e.endNodes()[1];
        if ( !selected[node] )
        {
            L newLength = length[source] + e.label();
            if ( newLength < length[node] )
            {
                // Atualizar menor caminho de origin a node.
            }
        }
    }
}
```

```
// Corpo do método exploreNode.
for every Edge<L> e in graph.outIncidentEdges(source)
{
    Node node = e.endNodes()[1];
    if ( !selected[node] )
    {
        L newLength = length[source] + e.label();
        if ( newLength < length[node] )
        {
            boolean nodeIsInQueue = length[node] < +∞;
            length[node] = newLength;
            via[node] = source;
            if ( nodeIsInQueue )
                connected.decreaseKey(node, length[node]);
            else
                connected.insert(length[node], node);
        }
    }
}
```

Complexidade do Algoritmo de Dijkstra

Grafo em vetor de listas de "incidências"

Fila implementada com Heap e Vetor

criação de 3 vetores	$\Theta(1)$
criação da fila com prioridade	$\Theta(\|V\|)$
inicialização de 2 vetores (selected e length)	$\Theta(\|V\|)$
inserção da origem na fila	$\Theta(1)$
$\leq \|V\|$ remoção do mínimo da fila	$O(\|V\| \times \log \|V\|)$
$\leq \|V\|$ obtenção dos arcos incidentes	$O(\|A\|)$
$\leq \|A\|$ inserção ou decremento da chave na fila	$O(\|A\| \times \log \|V\|)$
TOTAL	$O((\|V\| + \|A\|) \times \log \|V\|)$

Complexidade do Algoritmo de Dijkstra

Grafo em vetor de listas de “incidências”

Fila implem. com Fila de Fibonacci e Vetor

criação de 3 vetores	$\Theta(1)$
criação da fila com prioridade	$\Theta(\|V\|)$
inicialização de 2 vetores (selected e length)	$\Theta(\|V\|)$
inserção da origem na fila	$\Theta(1)$
$\leq \|V\|$ remoção do mínimo da fila	$O(\|V\| \times \log \|V\|)$
$\leq \|V\|$ obtenção dos arcos incidentes	$O(\|A\|)$
$\leq \|A\|$ inserção ou decremento da chave na fila	$O(\|A\| \times 1)$
TOTAL	$O(\|A\| + \|V\| \times \log \|V\|)$

Construção de um Caminho

```
// Assume-se que o método dijkstra já foi executado,
// tendo preenchido e retornado os vetores length e via, e que existe
// caminho da origem para o destino (i.e., length[destination] < +∞).
Deque<Node> getPath( Node[] via, Node origin, Node destination )
{
    Deque<Node> path = new LinkedList<Node>();
    Node node = destination;
    while ( node != origin )
    {
        path.addFirst(node);
        node = via[node];
    }
    path.addFirst(node);
    return path;
}
```

Complexidade:

Construção de um Caminho

```
// Assume-se que o método dijkstra já foi executado,
// tendo preenchido e retornado os vetores length e via, e que existe
// caminho da origem para o destino (i.e., length[destination] < +∞).
Deque<Node> getPath( Node[] via, Node origin, Node destination )
{
    Deque<Node> path = new LinkedList<Node>();
    Node node = destination;
    while ( node != origin )
    {
        path.addFirst(node);
        node = via[node];
    }
    path.addFirst(node);
    return path;
}
```

Complexidade: Θ(número de vértices do caminho) $O(|V|)$