Capítulo IX

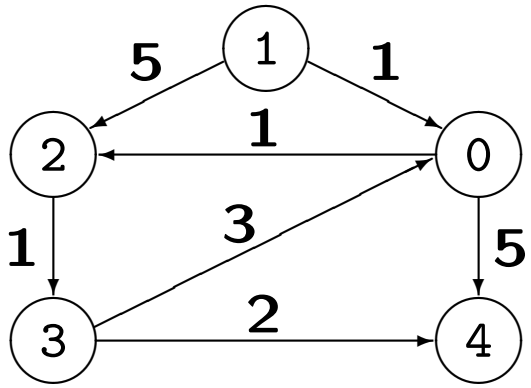
Caminhos Mais Curtos de um vértice a todos os vértices

—

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**?

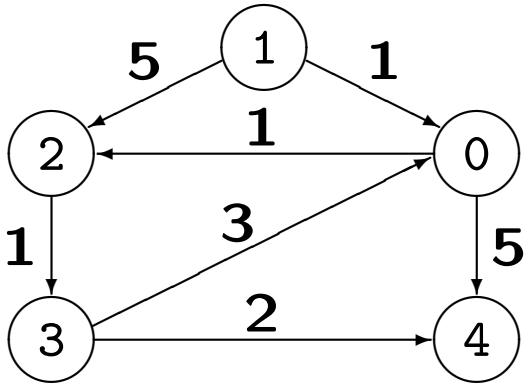


Caminho mais curto de 1 para 2

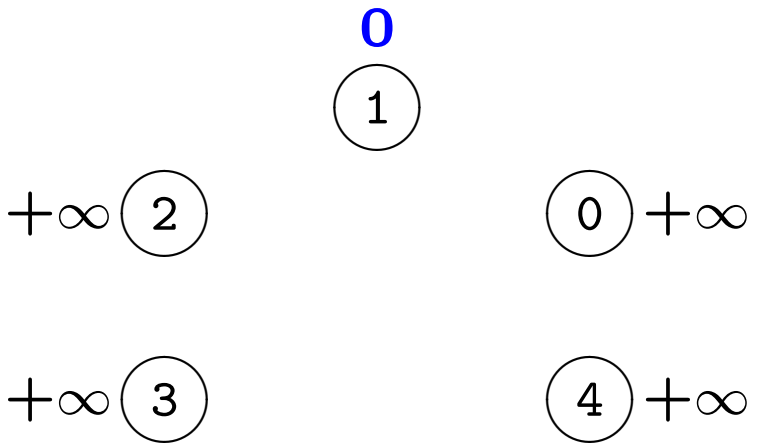
Caminho não pesado:	1, 2	Compr.: 1	Compr. pesado: 5
Caminho pesado:	1, 0, 2	Compr.: 2	Compr. pesado: 2

Restrição: os pesos dos arcos não são negativos.

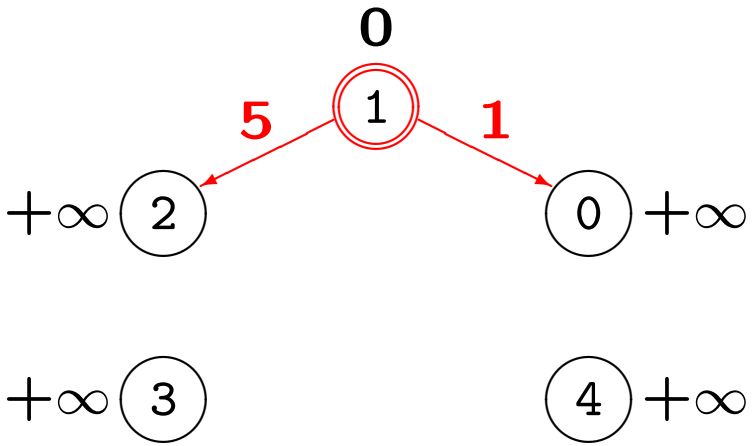
Algoritmo de Dijkstra [1959]

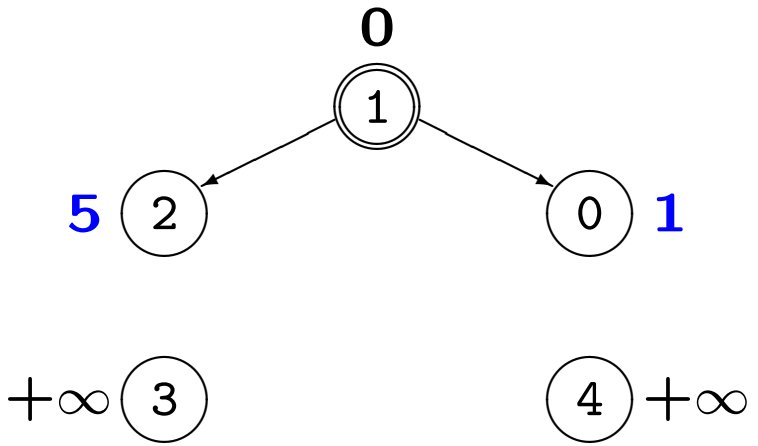


Inicialização origem 1

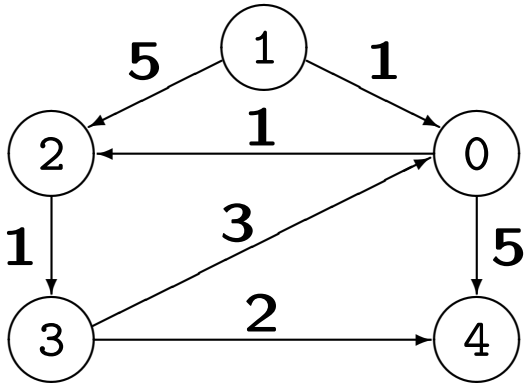


Seleção de 1

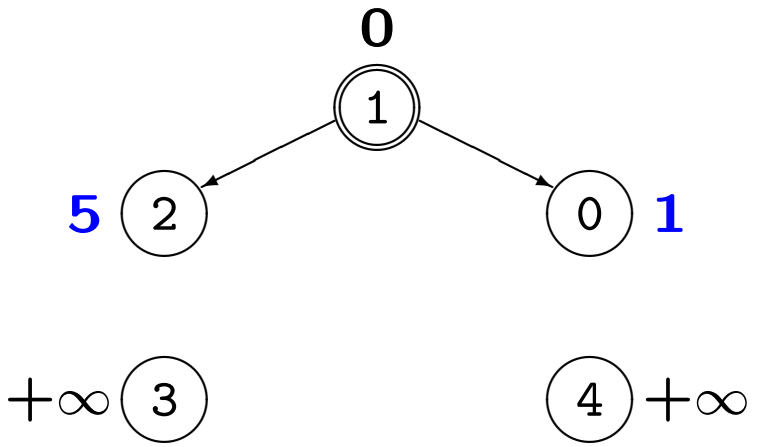




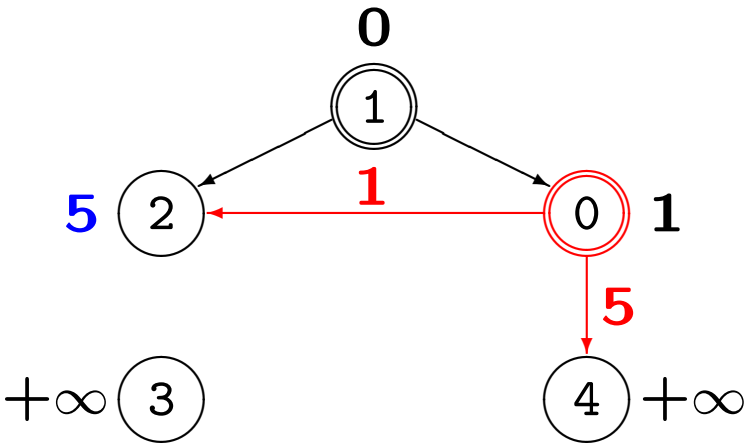
Algoritmo de Dijkstra (2)

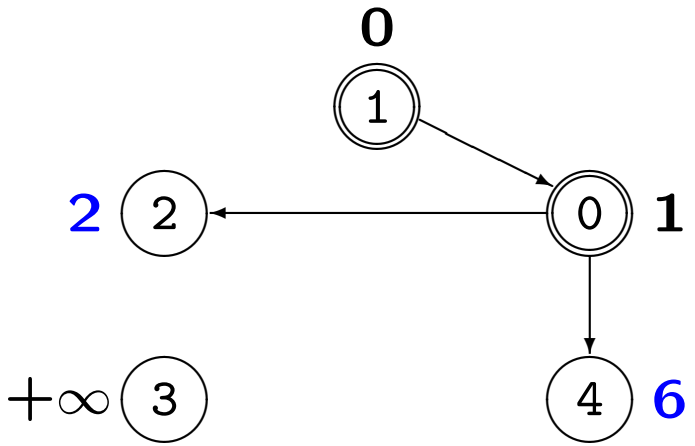


Situação Corrente

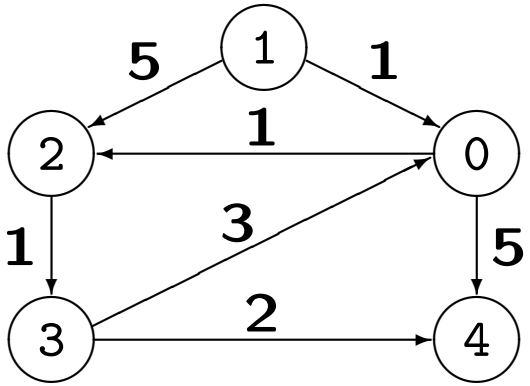


Seleção de 0

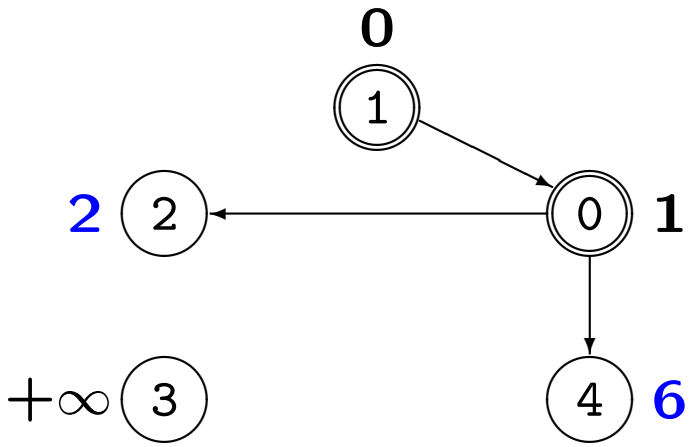




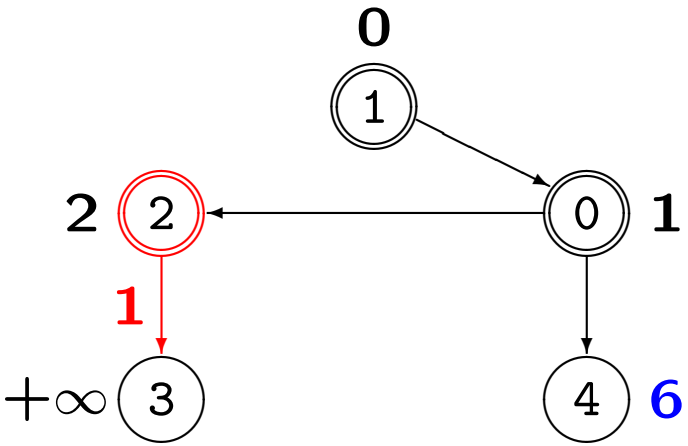
Algoritmo de Dijkstra (3)

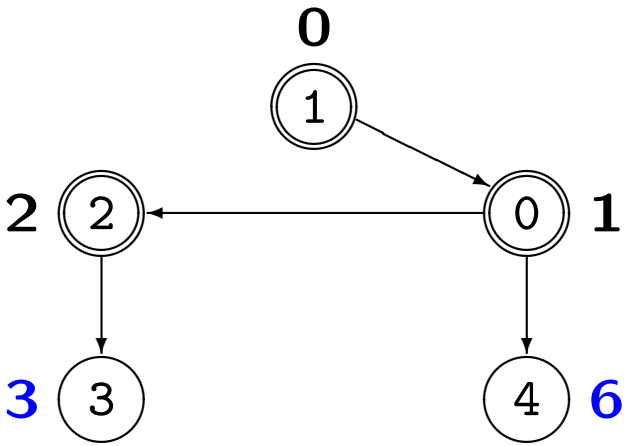


Situação Corrente

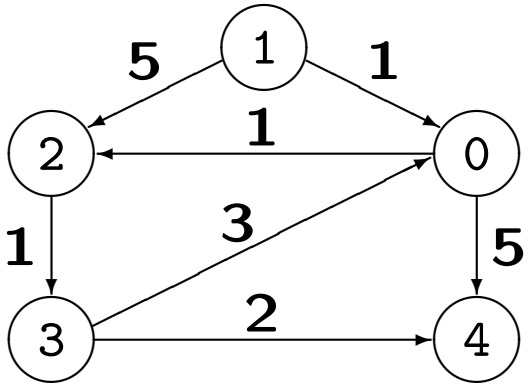


Seleção de 2

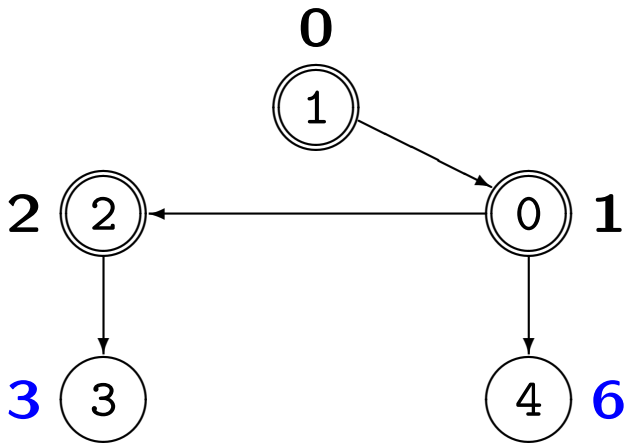




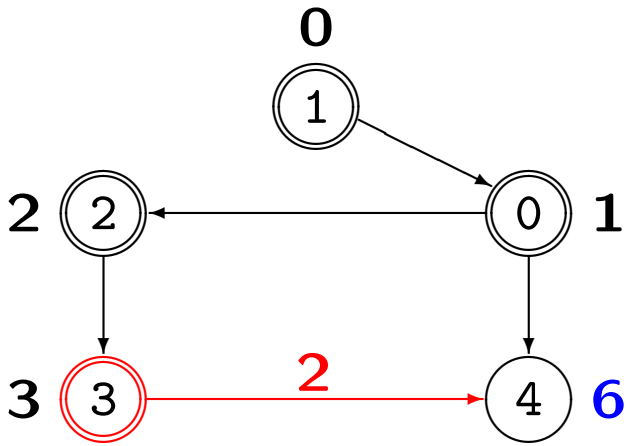
Algoritmo de Dijkstra (4)

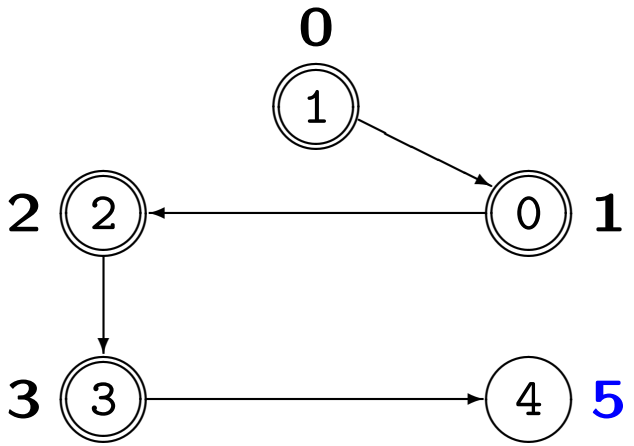


Situação Corrente

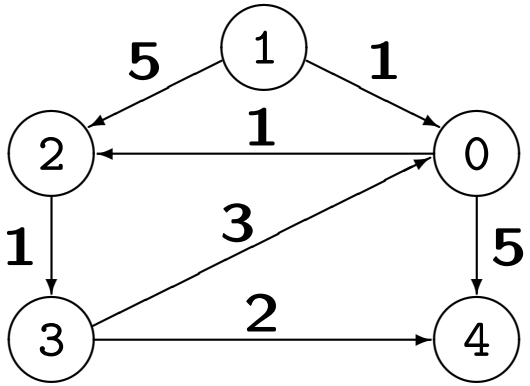


Seleção de 3

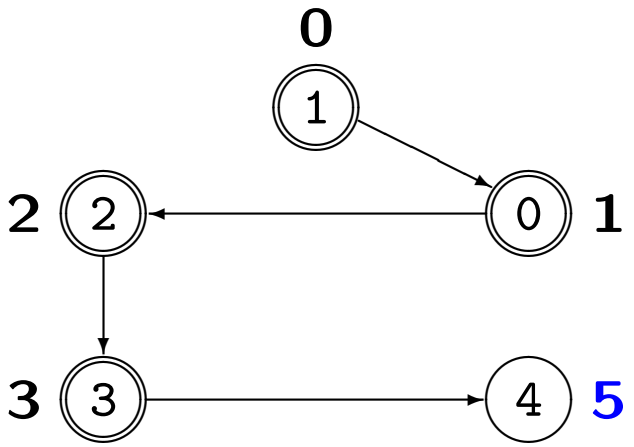




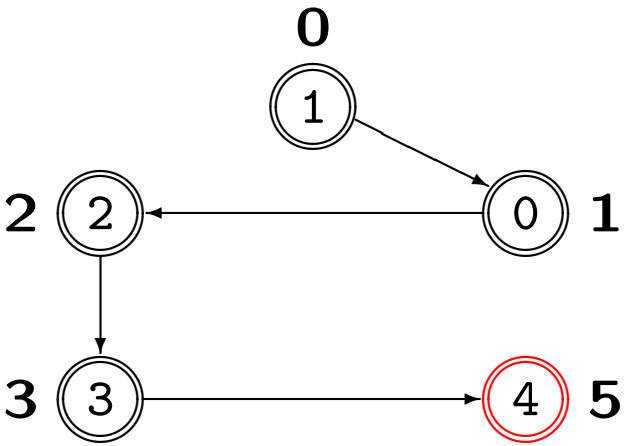
Algoritmo de Dijkstra (5)

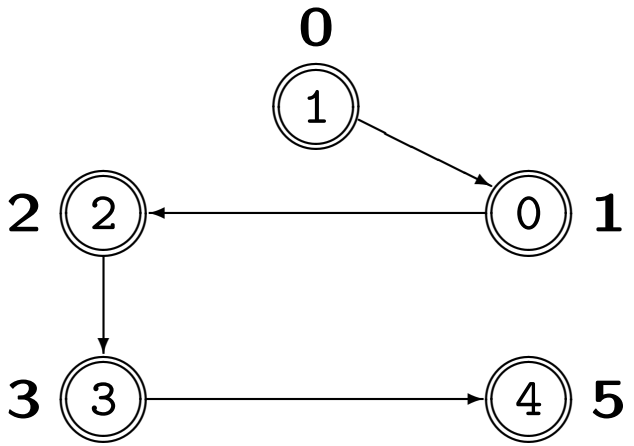


Situação Corrente



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.
- Vertex 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)

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

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

    Vertex[] via = new Vertex[ graph.numVertices() ];

    AdaptMinPriQueue<L, Vertex> connected =
        new AdaptMinHeap<L, Vertex>( graph.numVertices() );
```

Caminhos Mais Curtos (2)

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

Caminhos Mais Curtos (3)

```
    do {
        Vertex vertex = connected.removeMin().getValue();
        selected[vertex] = true;
        exploreVertex(graph, vertex, selected, length, via, connected);
      }
    while ( !connected.isEmpty() );

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

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

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

Complexidade do Algoritmo de Dijkstra

Fila implementada em Heap e Vetor

criar fila	$\Theta(\|V\|)$
inicializar 2 vetores (selected e length)	$\Theta(\|V\|)$
inserir origem na fila	$\Theta(1)$
$\leq \|V\|$ remover mínimo da fila	$O(\|V\| \times \log \|V\|)$
$\leq \|V\|$ percorrer sucessores diretos	$O(\|V\|^2)$ **ou** $O(\|A\|)$
$\leq \|A\|$ inserir na fila ou decrementar chave	$O(\|A\| \times \log \|V\|)$

TOTAL (matriz de adjacências)

TOTAL (listas de adjacências)

Complexidade do Algoritmo de Dijkstra

Fila implementada em Heap e Vetor

criar fila	$\Theta(\|V\|)$
inicializar 2 vetores (selected e length)	$\Theta(\|V\|)$
inserir origem na fila	$\Theta(1)$
$\leq \|V\|$ remover mínimo da fila	$O(\|V\| \times \log \|V\|)$
$\leq \|V\|$ percorrer sucessores diretos	$O(\|V\|^2)$ **ou** $O(\|A\|)$
$\leq \|A\|$ inserir na fila ou decrementar chave	$O(\|A\| \times \log \|V\|)$
TOTAL (matriz de adjacências)	$O(\|V\|^2 + \|A\| \times \log \|V\|)$
TOTAL (listas de adjacências)	$O((\|V\| + \|A\|) \times \log \|V\|)$

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<Vertex> getPath( Vertex[] via, Vertex origin, Vertex destination )
{
    Deque<Vertex> path = new LinkedList<Vertex>();
    Vertex vertex = destination;
    while ( vertex != origin )
    {
        path.addFirst(vertex);
        vertex = via[vertex];
    }
    path.addFirst(vertex);
    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<Vertex> getPath( Vertex[] via, Vertex origin, Vertex destination )
{
    Deque<Vertex> path = new LinkedList<Vertex>();
    Vertex vertex = destination;
    while ( vertex != origin )
    {
        path.addFirst(vertex);
        vertex = via[vertex];
    }
    path.addFirst(vertex);
    return path;
}
```

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