UNIVERSIDADE
CATÓLICA
DE PERNAMBUCO



# Estudo Dirigido

# Algoritmo de Dijkstra

Aluno: Victor Souza Santos
R.A.: 00000844512
Professor: Marcos Canejo
Curso: Ciências da Computação

# 1. Introdução

O roteamento eficiente de pacotes de dados é um dos pilares fundamentais para garantir o funcionamento adequado das redes de telecomunicações, como a Internet. Nesse contexto, o algoritmo de Dijkstra desempenha um papel crucial, permitindo encontrar o caminho mais curto entre nós em um grafo valorado, onde cada aresta representa uma conexão de rede com um determinado custo associado.

O algoritmo de Dijkstra, proposto pelo cientista da computação Edsger W. Dijkstra em 1956, é amplamente utilizado em uma variedade de aplicações, incluindo sistemas de navegação, logística, redes de computadores e, especialmente, em telecomunicações. Sua capacidade de encontrar caminhos mais curtos de forma eficiente em grafos valorados o torna uma ferramenta essencial para otimizar o roteamento de pacotes em redes de telecomunicações.

Este projeto tem como objetivo explorar o algoritmo de Dijkstra, fornecendo uma descrição detalhada de seu funcionamento, uma implementação prática em Java e sua aplicação em um problema real relacionado ao roteamento de redes de telecomunicações. Ao longo deste projeto, abordarei os principais conceitos do algoritmo de Dijkstra, mostrarei uma implementação passo a passo em Java e será demonstrado sua utilidade na solução de problemas práticos no contexto de telecomunicações.

Na próxima seção, será fornecido uma descrição detalhada do algoritmo de Dijkstra, explicando seus principais passos e destacando sua importância na resolução de problemas de roteamento de redes de telecomunicações.

- ## Descrição do Algoritmo de Dijkstra

O algoritmo de Dijkstra é um método eficiente para encontrar o caminho mais curto de um nó de origem para todos os outros nós em um grafo valorado, onde cada aresta possui um peso associado. Desenvolvido por Edsger W. Dijkstra em 1956, este algoritmo é amplamente utilizado em uma variedade de aplicações práticas devido à sua eficácia e simplicidade.

O funcionamento do algoritmo de Dijkstra pode ser dividido em quatro passos principais:

A. Inicialização:
- O algoritmo começa selecionando um nó de origem a partir do qual calculará os caminhos mais curtos para todos os outros nós no grafo;
- A distância do nó de origem até ele mesmo é definida como 0, enquanto a distância para todos os outros nós é inicializada como infinito;
- Uma estrutura de dados de fila de prioridade (geralmente implementada como uma heap binária) é utilizada para armazenar os nós a serem explorados, ordenados pela distância atual até o nó de origem.

B. Exploração dos Nós:
- Enquanto a fila de prioridade não estiver vazia, o algoritmo continua explorando os nós;
- O nó com a menor distância atual (ou seja, o próximo nó mais próximo do nó de origem) é removido da fila de prioridade e marcado como visitado.

C. Relaxamento das Arestas:
- Para cada nó adjacente ao nó removido, o algoritmo verifica se a distância até esse nó pode ser reduzida passando pelo nó removido;

- Se a distância até o nó adjacente for menor do que a distância atualmente conhecida, a distância é atualizada e o nó adjacente é adicionado à fila de prioridade.

D. Repetição:
- Os passos de exploração de nós e relaxamento de arestas são repetidos até que todos os nós tenham sido explorados ou até que o nó de destino tenha sido alcançado.

Esses passos garantem que, ao final do algoritmo, terei as menores distâncias de todos os nós até o nó de origem e o caminho mais curto de cada nó até o nó de origem. O algoritmo de Dijkstra é eficaz para grafos com pesos não negativos e produz resultados ótimos em termos de caminhos mais curtos. No entanto, é importante destacar que ele não lida adequadamente com grafos que contenham arestas com pesos negativos.

- Implementação de Fila de Prioridade em Java

A fila de prioridade, também conhecida como heap, é uma estrutura de dados que mantém os elementos de acordo com uma ordem de prioridade específica. No Java, podemos implementar uma fila de prioridade usando a classe `PriorityQueue`, que ordena os elementos por padrão em ordem ascendente. No entanto, pode-se personalizar a ordem de prioridade fornecendo um comparador personalizado.

Aqui está um exemplo de implementação de fila de prioridade em Java, onde os elementos são ordenados em ordem descendente:

```java
package Dijkstra;

import java.util.Collections;
import java.util.PriorityQueue;
import java.util.Queue;

// Uma fila de prioridade, é um tipo de fila em que podemos determinar uma condição
// de prioridade. No Java essa prioridade é por padrão ascendente, mas podemos
// usar uma outra classe e formar um comparador personalizado.

public class FilaDePrioridade {
    Run | Debug
    public static void main(String[] args) {
        Queue<Integer> filaDePrioridade = new PriorityQueue<>(Collections.reverseOrder());
        filaDePrioridade.add(e:2);
        filaDePrioridade.add(e:3);
        filaDePrioridade.add(e:1);
        while (!filaDePrioridade.isEmpty()) {
            System.out.println(filaDePrioridade.poll());
        }
    }
}
```

Output:

```
3
2
1
```

Neste exemplo, criei uma fila de prioridade `filaDePrioridade` com ordem descendente usando `Collections.reverseOrder()`. Em seguida, adicionei alguns elementos à fila e os removi um por um usando o método `poll()`, imprimindo-os em ordem de prioridade.

Outra forma de implementar uma fila em Java é usando a interface `Queue`, que segue o princípio FIFO (First In, First Out). Aqui está um exemplo:

```java
package Dijkstra;

import java.util.LinkedList;
import java.util.Queue;

// Uma queue ou fila, é uma estrutura de dados linear que segue o princípio FIFO (First In,
// First Out). Sobre a implementação dela em Java, temos a biblioteca Java util
// que apresenta a classe Queue.

public class Fila {
    Run | Debug
    public static void main(String[] args) {
        Queue<Integer> fila = new LinkedList<>();
        fila.add(e:1);
        fila.add(e:2);
        fila.add(e:3);
        while (!fila.isEmpty()) {
            System.out.println(fila.poll());
        }
    }
}
```

Output:

```
1
2
3
```

Neste exemplo, utiliza-se a classe `LinkedList` para implementar uma fila simples. Os elementos são adicionados usando `add()` e removidos usando `poll()`, mantendo a ordem em que foram inseridos.

Além disso, pode-se criar um comparador personalizado para definir a ordem de prioridade de acordo com nossos critérios. Aqui está um exemplo de um comparador simples que compara os inteiros:

Este

```java
package Dijkstra;

import java.util.Comparator;

public class MyComparator implements Comparator<Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o1.compareTo(o2);
    }
}
```

comparador pode ser usado para definir a ordem de prioridade em uma fila de prioridade de acordo com as necessidades específicas do problema.

## 2. Fluxograma do Algoritmo de Dijkstra

O fluxograma do algoritmo de Dijkstra é uma representação visual dos passos que o algoritmo segue para encontrar o caminho mais curto de um vértice de origem para todos os outros vértices em um grafo valorado. Este fluxograma detalha a inicialização das estruturas de dados, a exploração dos vértices, a marcação dos vértices visitados e a atualização das distâncias. A seguir, descrevemos o fluxograma e explicamos cada uma de suas partes.

- Descrição do Fluxograma

A. Início:

- O algoritmo começa com a inicialização das variáveis e estruturas de dados necessárias.

B. Inicialização:

- Distância Inicial: Para cada vértice `v` no grafo, `dist[v]` é definido como infinito (`∞`) e `previous[v]` é definido como indefinido (`undefined`);
- Origem: A distância do vértice de origem `s` é definida como 0 (`dist[s] = 0`);
- Fila de Prioridade: Todos os vértices são inseridos na fila de prioridade `Q`, ordenados pela distância atual (`dist`).

C. Exploração dos Vértices:

- Vértice com Menor Distância: Enquanto a fila de prioridade `Q` não estiver vazia, o vértice `u` com a menor `dist[u]` é removido de `Q`;
- Verificação de Pesos Negativos: Para cada vizinho `v` de `u`, se o peso da aresta entre `u` e `v` for negativo, o algoritmo retorna um erro.

D. Relaxamento das Arestas:

- Cálculo de Distância Alternativa: Para cada vizinho `v` de `u`, a distância alternativa `alt` é calculada como `dist[u] + peso da aresta(u, v)`;
- Atualização de Distâncias: Se `alt` for menor que `dist[v]`, `dist[v]` é atualizado para `alt` e `previous[v]` é atualizado para `u`. O vértice `v` é então reordenado na fila de prioridade `Q`.

E. Verificação da Fila:

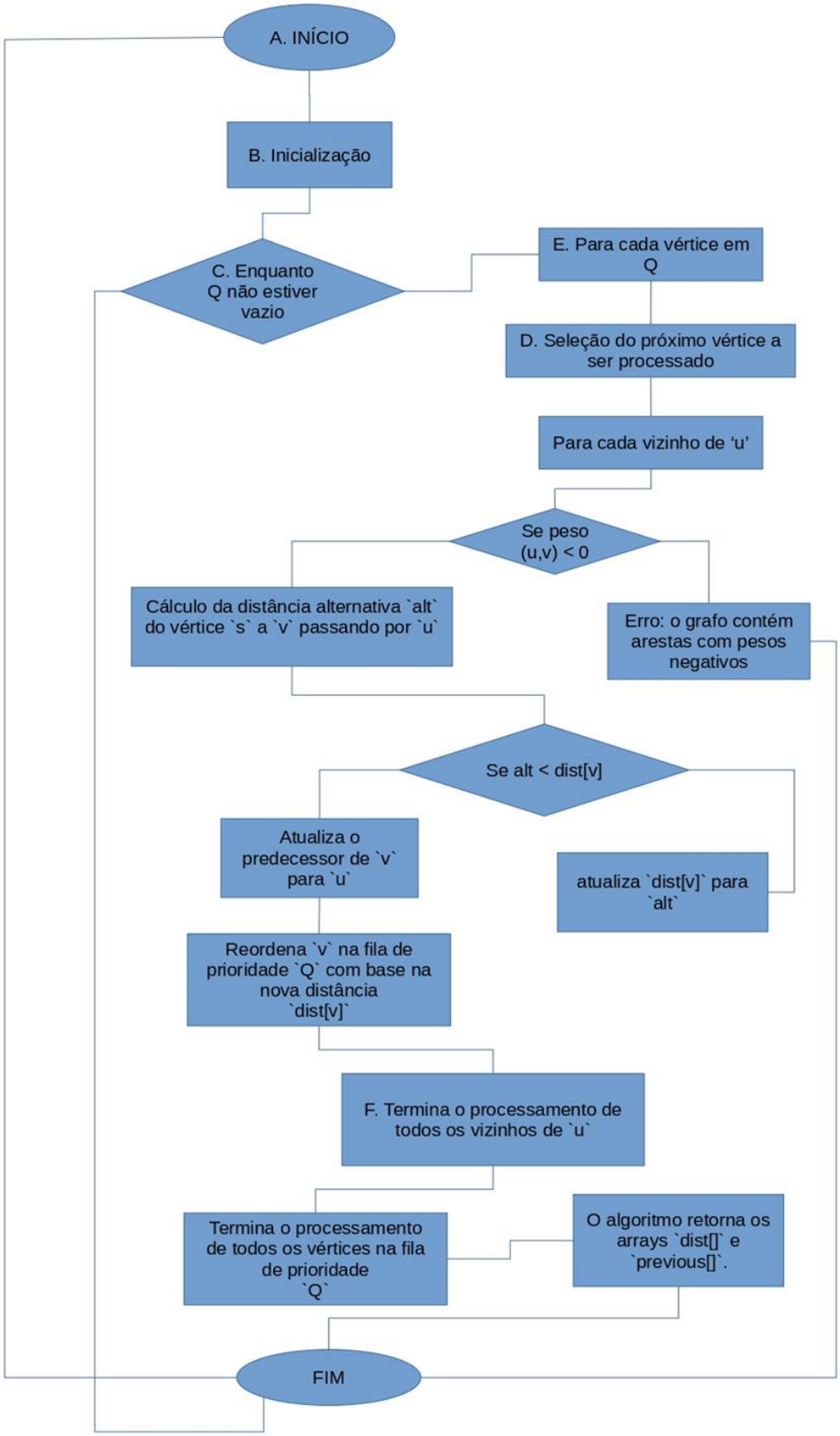
- Fila Vazia. O processo de exploração continua até que a fila de prioridade `Q` esteja vazia.

F. Fim:

- Retorno dos Resultados. Ao final, o algoritmo retorna os arrays `dist[]` e `previous[]`, que contêm as menores distâncias e os predecessores de cada vértice, respectivamente.

- Fluxograma

A seguir está o fluxograma detalhado do algoritmo de Dijkstra. Este fluxograma ilustra as etapas do algoritmo de forma sequencial e lógica:


A. INÍCIO
B. Inicialização
C. Enquanto Q não estiver vazio
E. Para cada vértice em Q
D. Seleção do próximo vértice a ser processado
Para cada vizinho de 'u'
Se peso (u,v) < 0
Cálculo da distância alternativa `alt` do vértice `s` a `v` passando por `u`
Erro: o grafo contém arestas com pesos negativos
Se alt < dist[v]
Atualiza o predecessor de `v` para `u`
atualiza `dist[v]` para `alt`
Reordena `v` na fila de prioridade `Q` com base na nova distância `dist[v]`
F. Termina o processamento de todos os vizinhos de `u`
Termina o processamento de todos os vértices na fila de prioridade `Q`
O algoritmo retorna os arrays `dist[]` e `previous[]`.
FIM

- Detalhamento dos Principais Passos do Fluxograma

- A. Inicialização: Configura as distâncias iniciais e a fila de prioridade.
- B. Exploração dos Vértices: Seleciona o vértice com a menor distância e explora seus vizinhos.
- C. Relaxamento das Arestas: Atualiza as distâncias para os vizinhos se um caminho mais curto for encontrado.
- D. Verificação da Fila: Continua o processo até que todos os vértices tenham sido explorados.
- E. Retorno dos Resultados: Após a exploração completa, retorna as distâncias mínimas e os predecessores.

# 3. Problema Prático

- Aplicação do Algoritmo de Dijkstra em Telecomunicações

Neste tópico vou aplicar o algoritmo de Dijkstra em um problema prático de telecomunicações. Considere um cenário onde uma empresa de telecomunicações deseja otimizar a rota de transmissão de dados entre diferentes centros de dados em uma rede de fibra óptica. Cada centro de dados é representado por um vértice no grafo, e cada conexão de fibra óptica entre centros de dados é representada por uma aresta com um peso correspondente ao tempo de transmissão (latência) entre esses centros.

O objetivo é determinar o caminho mais curto (com menor latência) de um centro de dados de origem para todos os outros centros de dados na rede.

- Modelo do Problema

- Vértices: Representam os centros de dados.
- Arestas: Representam as conexões de fibra óptica entre os centros de dados.
- Pesos das Arestas: Representam a latência (tempo de transmissão) das conexões.

EXEMPLO: Considere um grafo valorado com os seguintes centros de dados e conexões:

- Vértices: A, B, C, D, E
- Arestas:
  - ➢ - A - B (latência 4)
  - ➢ - A - C (latência 2)
  - ➢ - B - C (latência 1)
  - ➢ - B - D (latência 5)
  - ➢ - C - D (latência 8)
  - ➢ - C - E (latência 10)
  - ➢ - D - E (latência 2)

- Seguindo os Passos do Algoritmo de Dijkstra:

A. Inicialização:
- Definir o centro de dados de origem, por exemplo, `A`;

- Atribuir `dist[A] = 0` e `dist[v] = ∞` para todos os outros vértices;
- `previous[v]` é indefinido para todos os vértices;
- Inserir todos os vértices na fila de prioridade `Q`.

B. Exploração dos Vértices:
- Selecionar o vértice com a menor distância da fila de prioridade `Q` e removê-lo;
- Atualizar as distâncias para os vizinhos do vértice selecionado.

C. Relaxamento das Arestas:
- Para cada vizinho, calcular a distância alternativa e atualizar `dist` e `previous` se um caminho mais curto for encontrado.

D. Repetição:
- Continuar até que todos os vértices tenham sido processados.

- Implementação em Java

O código DjikstraPRoblemaPrático.java, presente no repositório do GitHib, resolve exatamente o problema descrito acima e sua resposta/output é:

Output:

```
Caminho mais curto de A para A: A(unreached)
Caminho mais curto de A para B: A(unreached) -> B(4)
Caminho mais curto de A para C: A(unreached) -> C(2)
Caminho mais curto de A para D: A(unreached) -> B(4) -> D(9)
Caminho mais curto de A para E: A(unreached) -> B(4) -> D(9) -> E(11)
```

- Resumo do Processo de Aplicação

- **Inicialização**: Distâncias definidas e fila de prioridade criada;
- **Exploração**: Seleção do vértice com menor distância, remoção da fila e processamento dos vizinhos;
- **Relaxamento**: Atualização das distâncias alternativas e reordenação na fila de prioridade;
- **Resultado**: Impressão dos caminhos mais curtos de `A` para todos os outros vértices, indicando a rota e a latência total.

Esta implementação demonstra como o algoritmo de Dijkstra pode ser utilizado para otimizar rotas de transmissão de dados em uma rede de telecomunicações, minimizando a latência entre centros de dados.