

# Estrutura de dados 2

## PRIMEIRO SEMESTRE DE 2024

INSTRUTOR: JOÃO DALLYSON

ALUNOS: GABRIEL BELO PEREIRA DOS REIS, RONDINELI SEBA SALOMAO

---

**Link apresentação: programa 1 e 4 - Gabriel Belo**

[https://youtu.be/Be\\_HpLda7Zs](https://youtu.be/Be_HpLda7Zs)

**Link apresentação: programa 2 e 3 - Rondineli Seba**

<https://youtu.be/gpZYii7CFjU>

### QUESTÃO 1:

**Implemente o algoritmo de busca em profundidade:**

1. Implemente um método para retornar o número de arestas existentes entre o vértice de origem da DFS e um outro vértice destino.
  2. Implemente um método para retornar um caminho de um vértice de origem a um vértice destino.
  3. Implemente um método para imprimir todas as arestas do tipo retorno.
-

---

## **Discussão:**

A busca em profundidade (DFS) é uma técnica fundamental na exploração de grafos, permitindo visitar todos os vértices e arestas de maneira sistemática. O DFS funciona explorando o mais longe possível em cada ramificação antes de retroceder, utilizando uma abordagem recursiva ou uma pilha explícita para gerenciar os vértices a serem visitados. Esta implementação do DFS é otimizada para incluir funcionalidades adicionais, como a identificação de arestas de retorno e a reconstrução de caminhos entre vértices específicos. A DFS é especialmente útil para:

- 1. Detectar ciclos em grafos.**
- 2. Encontrar caminhos entre pares de vértices.**
- 3. Explorar componentes conectados.**
- 4. Determinar a profundidade de cada vértice a partir da raiz.**

Nesta implementação, detalhamos cada método e explicamos como cada parte do algoritmo contribui para alcançar esses objetivos.

## **Descrição das Classes e Métodos:**

### **DFS.java:**

#### **Método dfs:**

Executa a busca em profundidade em todos os vértices do grafo, inicializando as estruturas necessárias.

- 
- **Linhas 1-4:** Inicializa todas as cores dos vértices como BRANCO (não visitados) e define seus pais como -1.
  - **Linhas 5-7:** Reinicializa o contador de tempo e itera sobre todos os vértices, iniciando a visitação dos vértices que ainda não foram visitados.

### **Método dfsVisit:**

Método recursivo que realiza a visitação dos vértices, explorando suas adjacências e identificando arestas de retorno.

- Linhas 1-3: Incrementa o contador de tempo e marca o vértice atual como CINZENTO (em processo de visitação), registrando o tempo de descoberta.
- Linhas 4-11: Explora todos os vértices adjacentes. Se um vértice adjacente não foi visitado (BRANCO), define o vértice atual como pai e chama recursivamente o método dfsVisit. Se um vértice adjacente está em processo de visitação (CINZENTO) e não é o pai do vértice atual, identifica uma aresta de retorno.
- Linhas 12-15: Após explorar todas as adjacências, marca o vértice atual como PRETO (completamente visitado), incrementa o tempo e registra o tempo de finalização.

### **Método getNumeroDeArestas:**

Retorna o número de arestas entre dois vértices, utilizando um método auxiliar recursivo para contar as arestas.

- Linhas 1-2: Inicializa um array de vértices visitados e chama o método recursivo dfsNumeroDeArestasUtil.

- 
- Linhas 3-9: Método auxiliar que percorre recursivamente o grafo a partir do vértice atual, contando as arestas até alcançar o vértice de destino.

### **Método getCaminho:**

Retorna o caminho entre dois vértices, reconstruindo-o a partir do array de pais.

- Linhas 1-3: Inicializa um array de vértices visitados e uma lista para armazenar o caminho. Chama o método auxiliar recursivo dfsCaminhoUtil.
- Linhas 4-15: Método auxiliar que percorre recursivamente o grafo a partir do vértice atual, adicionando os vértices ao caminho até alcançar o vértice de destino.

### **Método getArestasDeRetorno:**

Retorna todas as arestas de retorno encontradas durante a execução da DFS.

- Linhas 1-2: Retorna uma nova lista contendo as arestas de retorno identificadas durante a visitação dos vértices.

## **Grafo.java:**

### **Atributos:**

- **int vertices:** Número de vértices no grafo.
- **LinkedList<Integer>[] listaAdj:** Lista de adjacências para armazenar as arestas.

### **Métodos:**

- **Grafo(int vertices):** Construtor para inicializar o grafo.

- 
- Inicializa um grafo com um número específico de vértices, criando uma lista de adjacências para cada vértice.
  - **void adicionarAresta(int v, int w):** Adiciona uma aresta entre os vértices v e w.
    - Adiciona uma aresta dirigindo do vértice v para o vértice w, registrando essa conexão na lista de adjacências.
  - **LinkedList<Integer>[] getListaAdj():** Retorna a lista de adjacências.
    - Fornece acesso à estrutura interna do grafo, retornando a lista de adjacências.
  - **int getVertices():** Retorna o número de vértices no grafo.
    - Retorna o número total de vértices presentes no grafo.

### **Main.java:**

### **Método main:**

Executa a demonstração e os testes de busca em profundidade.

- **Linhas 1-5:** Inicializa o grafo pedindo ao usuário o número de vértices e as arestas a serem adicionadas.
- **Linhas 6-13:** Recebe as arestas do usuário e as adiciona ao grafo.
- **Linhas 14-16:** Cria uma instância do DFS e executa a busca em profundidade.
- **Linhas 17-40:** Apresenta um menu de opções para o usuário escolher diferentes funcionalidades, como mostrar o número de arestas, mostrar o caminho, mostrar arestas de retorno e exibir o grafo.

---

## PRINTS:

```
Digite o número de vértices no grafo: 6
Digite as arestas do grafo no formato 'origem destino'. Digite '-1 -1' para parar.
Aresta: 1 2
Aresta adicionada: 1 -> 2
Aresta: 1 3
Aresta adicionada: 1 -> 3
Aresta: 3 4
Aresta adicionada: 3 -> 4
Aresta: 4 5
Aresta adicionada: 4 -> 5
Aresta: 5 6
Aresta adicionada: 5 -> 6
Aresta: -1 -1
Construção do grafo concluída.
-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todas as arestas do tipo retorno
4. Exibir o grafo atual
5. Sair
Opção: 4
Grafo atual:
1 -> 2 3
2 ->
3 -> 4
4 -> 5
5 -> 6
6 ->
```

```
8 7 6
6 ->

-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todas as arestas do tipo retorno
4. Exibir o grafo atual
5. Sair
Opção: 1
Digite o vértice de origem: 1
Digite o vértice de destino: 6
Número de arestas de 1 a 6: 4

-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todas as arestas do tipo retorno
4. Exibir o grafo atual
5. Sair
Opção: 2
Digite o vértice de origem: 1
Digite o vértice de destino: 6
Caminho de 1 a 6: 1 3 4 5 6

-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todas as arestas do tipo retorno
```

---

```
5. Sair
Opção: 1
Digite o vértice de origem: 1
Digite o vértice de destino: 3
Número de arestas de 1 a 3: 2

-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todas as arestas do tipo retorno
4. Exibir o grafo atual
5. Sair
Opção: 3
Nenhuma aresta de retorno encontrada.

-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todas as arestas do tipo retorno
4. Exibir o grafo atual
5. Sair
Opção: 4
Grafo atual:
1 -> 2
2 -> 3
3 ->
4 -> 5
5 -> 6
6 ->
```

```

Digite o número de vértices no grafo: 6
-----
Comece com arestas maiores que 0! exemplo: 1, 2, 3
-----
Digite as arestas do grafo no formato 'origem destino'. Digite '-1 -1' para parar.
Aresta: 1 2
Aresta adicionada: 1 -> 2
Aresta: 1 3
Aresta adicionada: 1 -> 3
Aresta: 2 4
Aresta adicionada: 2 -> 4
Aresta: 3 5
Aresta adicionada: 3 -> 5
Aresta: 4 6
Aresta adicionada: 4 -> 6
Aresta: -1 -1
Construção do grafo concluída.
-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todas as arestas do tipo retorno
4. Exibir o grafo atual
5. Sair
Opção: 1
Digite o vértice de origem: 1
Digite o vértice de destino: 6
Número de arestas de 1 a 6: 3

-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
```

```
-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todas as arestas do tipo retorno
4. Exibir o grafo atual
5. Sair
Opção: 2
Digite o vértice de origem: 1
Digite o vértice de destino: 6
Caminho de 1 a 6: 1 2 4 6
```

```
-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todas as arestas do tipo retorno
4. Exibir o grafo atual
5. Sair
Opção: 3
Nenhuma aresta de retorno encontrada.
```

```
-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todas as arestas do tipo retorno
4. Exibir o grafo atual
5. Sair
Opção: 4
Grafo atual:
1 -> 2 3
```

```
Caminho de 1 a 6: 1 2 4 6
```

```
-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todas as arestas do tipo retorno
4. Exibir o grafo atual
5. Sair
Opção: 3
Nenhuma aresta de retorno encontrada.
```

```
-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todas as arestas do tipo retorno
4. Exibir o grafo atual
5. Sair
Opção: 4
Grafo atual:
1 -> 2 3
2 -> 4
3 -> 5
4 -> 6
5 ->
6 ->
```

```
-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
```



```

Digite o número de vértices no grafo: 6
-----
Comece com arestas maiores que 0! exemplo: 1, 2, 3
-----
Digite as arestas do grafo no formato 'origem destino'. Digite '-1 -1' para parar.
Aresta: 1 2
Aresta adicionada: 1 -> 2
Aresta: 1 3
Aresta adicionada: 1 -> 3
Aresta: 4 5
Aresta adicionada: 4 -> 5
Aresta: 5 6
Aresta adicionada: 5 -> 6
Aresta: -1 -1
Construção do grafo concluída.
-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todas as arestas do tipo retorno
4. Exibir o grafo atual
5. Sair
Opção: 1
Digite o vértice de origem: 1
Digite o vértice de destino: 5
Caminho não encontrado.
-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices

```

```

5. Sair
Opção: 2
Digite o vértice de origem: 1
Digite o vértice de destino: 5
Caminho não encontrado.
-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todas as arestas do tipo retorno
4. Exibir o grafo atual
5. Sair
Opção: 3
Nenhuma aresta de retorno encontrada.
-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todas as arestas do tipo retorno
4. Exibir o grafo atual
5. Sair
Opção: 4
Grafo atual:
1 -> 2 3
2 ->
3 ->
4 -> 5
5 -> 6
6 ->

```

*Ao desenvolver a implementação do algoritmo de busca em profundidade (DFS) no grafo, decidimos, ao final do algoritmo, restringir o número inicial dos vértices a 1 para simplificar a interface de usuário e evitar confusão. Como os grafos são frequentemente apresentados e manipulados utilizando índices baseados em 1, essa restrição vai garantir que os usuários forneçam entradas válidas. Essa abordagem reduziu possíveis erros com aresta inicial sendo 0)*

---

## QUESTÃO 2: PROGRAMA MST

O programa implementa o algoritmo de Kruskal para encontrar a Árvore Geradora Mínima (MST) de um grafo. Utilizamos uma estrutura de união e busca (UnionFind) para gerenciar a união e busca. A algoritmo de Kruskal utilizado para encontrar a Árvore Geradora Mínima, ou MST, de um grafo. O algoritmo de Kruskal é uma abordagem clássica em teoria dos grafos que encontra a MST ao adicionar arestas de menor peso, evitando ciclos.

### Estrutura do Programa

O programa foi estruturado de forma que cada classe encontra-se em um arquivo distinto, seguindo a estrutura:

#### **Classe Aresta.java**

A classe Aresta.java representa uma aresta no grafo, contendo os vértices de origem e destino, além do peso da aresta. Ela implementa a interface *Comparable* para permitir a ordenação das arestas pelo peso em ordem crescente.

- **Funcionalidade:** Representa uma aresta do grafo com origem, destino e peso.

- **Métodos:**

- Construtor: Inicializa a aresta com origem, destino e peso.
- Getters: Obtêm a origem, destino e peso da aresta.
- compareTo: Compara duas arestas pelo peso (ordem crescente).

---

## **Classe Grafo.java**

**- Funcionalidade: Representa o grafo contendo todas as arestas.**

**- Métodos:**

- adicionarAresta: Adiciona uma aresta ao grafo.
- getAdjacentes: Obtém a lista de arestas adjacentes a um vértice.
- getArestas: Obtém a lista de todas as arestas do grafo.
- getVertices: Obtém o conjunto de todos os vértices do grafo.
- getArestas:  $O(E)$ , onde  $E$  é o número de arestas.

## **Classe UnionFind.java**

A classe UnionFind.java é uma estrutura de dados para encontrar e unir conjuntos disjuntos, essencial para verificar se a adição de uma aresta criará um ciclo no grafo. Ela usa a compressão de caminhos e a união por tamanho para otimizar essas operações.

**- Funcionalidade: Implementa a estrutura de união e busca.**

**- Métodos:**

- Construtor: Inicializa as estruturas de pai e tamanho para cada elemento.
- encontrar: Encontra o representante do conjunto de um elemento.
- unir: Une dois conjuntos distintos.
- conectados: Verifica se dois elementos pertencem ao mesmo conjunto.

**Complexidade:**

---

## Classe MST.java

A classe **MST** implementa o algoritmo de Kruskal. Ela ordena as arestas pelo peso e utiliza a estrutura **UnionFind** para adicionar as arestas mais leves ao MST, garantindo que não haja ciclos. O método **encontrarMST** retorna a lista de arestas que compõem a MST.

**Funcionalidade:** Implementa o algoritmo de Kruskal para encontrar a MST.

### - Métodos:

- **encontrarMST:** Encontra a MST do grafo.
- **Complexidade:**  $O(E \log E)$ , onde  $E$  é o número de arestas (dominado pela ordenação das arestas).

## Classe Main

**- Funcionalidade:** Classe principal para testar a implementação.

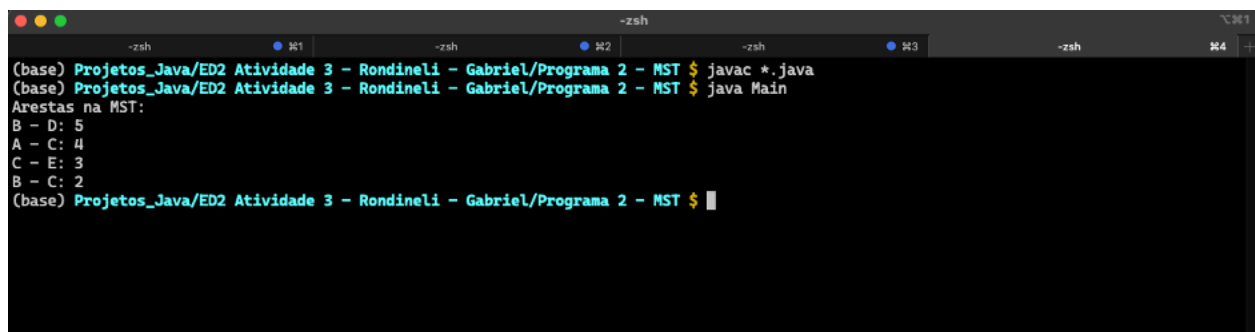
### - Métodos:

- **main:** Testa a implementação criando um grafo e chamando os métodos da classe **MST**.
- **Complexidade:**  $O(V + E)$ , devido às iterações iniciais e chamadas aos métodos de outras classes.

## Funcionamento Geral

- 
1. Aresta: Define uma aresta com origem, destino e peso. A implementação de *Comparable* permite a ordenação das arestas.
  2. Grafo: Armazena as arestas em uma lista e os vértices em um conjunto. Oferece métodos para adicionar arestas e obter vértices e suas adjacências.
  3. UnionFind: Fornece uma estrutura de dados para gerenciar a união e busca de conjuntos disjuntos, essencial para o algoritmo de MST.
  4. MST: Utiliza o algoritmo de Kruskal, ordenando as arestas em ordem crescente e adicionando as mais leves primeiro para encontrar a Árvore Geradora Mínima.
  5. Main: Testa a implementação dos algoritmos criando um grafo e chamando os métodos das classes MST.

## PRINTS



```
(base) Projetos_Java/ED2 Atividade 3 - Rondineli - Gabriel/Programa 2 - MST $ javac *.java
(base) Projetos_Java/ED2 Atividade 3 - Rondineli - Gabriel/Programa 2 - MST $ java Main
Arestas na MST:
B - D: 5
A - C: 4
C - E: 3
B - C: 2
(base) Projetos_Java/ED2 Atividade 3 - Rondineli - Gabriel/Programa 2 - MST $
```

```

1  import java.util.List;
2
3  public class Main {
4      public static void main(String[] args) {
5          Grafo<String> grafo = new Grafo<>();
6          grafo.adicionarAresta("A", "B", 1);
7          grafo.adicionarAresta("A", "C", 4);
8          grafo.adicionarAresta("B", "C", 2);
9          grafo.adicionarAresta("B", "D", 5);
10         grafo.adicionarAresta("C", "D", 1);
11         grafo.adicionarAresta("C", "E", 3);
12         grafo.adicionarAresta("D", "E", 2);
13
14         MST<String> mst = new MST<>();
15         List<Aresta<String>> resultadoMST = mst.encontrarMST(grafo);
16
17         System.out.println("Arestas na MST:");
18         for (Aresta<String> aresta : resultadoMST) {
19             System.out.println(aresta.getOrigem() + " - " + aresta.getDestino() + ": " + aresta.getPeso());
20         }
21     }
22 }

```

```

1  import java.util.*;
2
3  public class MST<T> {
4      public List<Aresta<T>> encontrarMST(Grafo<T> grafo) {
5          List<Aresta<T>> arestas = grafo.getArestas();
6          Set<T> vertices = grafo.getVertices();
7
8          // Ordena as arestas em ordem crescente
9          Collections.sort(arestas);
10
11          UnionFind<T> uf = new UnionFind<>(vertices);
12
13          // Lista para armazenar as arestas da MST
14          List<Aresta<T>> mst = new ArrayList<>();
15
16          for (Aresta<T> aresta : arestas) {
17              T origem = aresta.getOrigem();
18              T destino = aresta.getDestino();
19
20              // Verifica se a aresta conecta dois componentes diferentes
21              if (!uf.conectados(origem, destino)) {
22                  uf.unir(origem, destino);
23              }
24          }
25          return mst;
26      }
27  }

```

---

```
1 import java.util.*;
2
3 public class Grafo<T> {
4     private Map<T, List<Aresta<T>>> adjacencia;
5
6     public Grafo() {
7         this.adjacencia = new HashMap<>();
8     }
9
10    public void adicionarAresta(T origem, T destino, int peso) {
11        Aresta<T> aresta = new Aresta<>(origem, destino, peso);
12        adjacencia.computeIfAbsent(origem, k -> new ArrayList<>()).add(aresta);
13        adjacencia.computeIfAbsent(destino, k -> new ArrayList<>()); // Garante que o destino também seja
14    }
15
16    public List<Aresta<T>> getAdjacentes(T vertice) {
17        return adjacencia.getOrDefault(vertice, new ArrayList<>());
18    }
19
20    public List<Aresta<T>> getArestas() {
21        List<Aresta<T>> arestas = new ArrayList<>();
22        for (List<Aresta<T>> lista : adjacencia.values()) {
23            // ...
24        }
25    }
26 }
```

---

## QUESTÃO 3: PROGRAMA DIJKSTRA MODIFICADO

O programa implementa uma versão modificada do algoritmo de Dijkstra para encontrar os dois menores caminhos entre um par de vértices em um grafo. Utiliza-se uma fila de prioridade para gerenciar os caminhos.

### Estrutura do Programa

O programa está organizado nas seguintes classes:

1. Aresta: Representa uma aresta do grafo com peso.
2. Grafo: Representa o grafo contendo todas as arestas.
3. Caminho: Representa um caminho e seu peso total.
4. DijkstraModificado: Implementa o algoritmo de Dijkstra modificado para encontrar os dois menores caminhos.
5. Main: Classe principal para testar a implementação.



The screenshot shows an IDE with a project named "Programa 3 - Dijkstra Modificado". The left sidebar displays the project structure under a "src" folder, listing the following files: Aresta.class, Aresta.java, Caminho.class, Caminho.java, DijkstraModificado.class, DijkstraModificado.java, Grafo.class, Grafo.java, Main.class, and Main.java. The main editor window shows Java code for the DijkstraModificado class, with lines 17 through 22 visible. The code includes a print statement for MST edges and a loop to iterate over the results. The bottom of the IDE shows a terminal window with the command prompt "(base) Projetos\_Java/ED2 Atividade 3 - Rondineli - Gabriel/src \$".

```
17 System.out.println("Arestas na MST:");
18 for (Aresta<String> aresta : resultadoMST) {
19     System.out.println(aresta.getOrigem() + " - " + aresta.getDestino() +
20 }
21 }
22 }
```



---

## Funcionalidades

### Classe Aresta

Funcionalidade: Representa uma aresta do grafo com origem, destino e peso.

#### Métodos:

- **Construtor:** Inicializa a aresta com origem, destino e peso.
- **Getters:** Obtêm a origem, destino e peso da aresta.

### Classe Grafo

Funcionalidade: Representa o grafo contendo todas as arestas.

#### Métodos:

- **adicionarAresta:** Adiciona uma aresta ao grafo.
- **getAdjacentes:** Obtém a lista de arestas adjacentes a um vértice.
- **getArestas:** Obtém a lista de todas as arestas do grafo.
- **getVertices:** Obtém o conjunto de todos os vértices do grafo.
- **Complexidade getArestas:**  $O(E)$ , onde  $E$  é o número de arestas.

### Classe Caminho

Funcionalidade: Representa um caminho em termos de uma lista de vértices e o peso total do caminho.

#### Métodos:

- **Construtores:** Inicializa o caminho.

- 
- adicionarVertice: Adiciona um vértice ao caminho e atualiza o peso.
  - getVertices: Obtém a lista de vértices do caminho.
  - getPesoTotal: Obtém o peso total do caminho.
  - compareTo: Compara dois caminhos pelo peso total.

### **Classe DijkstraModificado**

Funcionalidade: Implementa o algoritmo de Dijkstra modificado para encontrar os dois menores caminhos.

#### **Métodos:**

- encontrarDoisMenoresCaminhos: Encontra os dois menores caminhos entre um par de vértices.
- Complexidade:  $O((V + E) \log V)$ , onde  $V$  é o número de vértices e  $E$  é o número de arestas.

### **Classe Main**

Funcionalidade: Classe principal para testar a implementação.

#### **Métodos:**

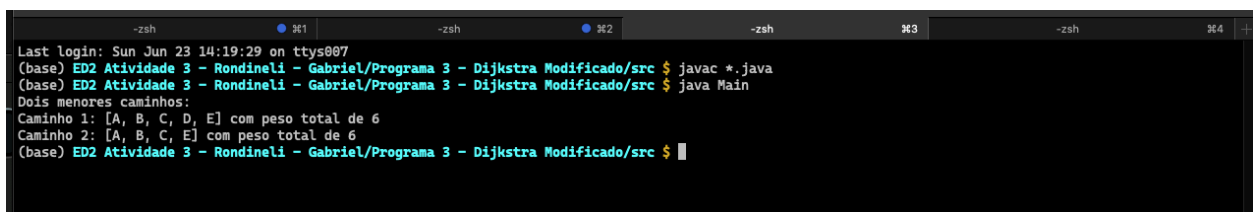
- main: Testa a implementação criando um grafo e chamando os métodos da classe DijkstraModificado.
- Complexidade:  $O(V + E)$ , devido às iterações iniciais e chamadas aos métodos de outras classes.

---

## Funcionamento Geral

1. Aresta: Define uma aresta com origem, destino e peso.
2. Grafo: Armazena as arestas em uma lista e os vértices em um conjunto.
3. Caminho: Representa um caminho em termos de uma lista de vértices e o peso total do caminho.
4. DijkstraModificado: Utiliza uma fila de prioridade para gerenciar os caminhos e encontrar os dois menores caminhos entre um par de vértices.
5. Main: Testa a implementação dos algoritmos criando um grafo e chamando os métodos das classes DijkstraModificado.

## PRINTS



```
-zsh 001 -zsh 002 -zsh 003 -zsh 004 +
Last login: Sun Jun 23 14:19:29 on ttys007
(base) ED2 Atividade 3 - Rondineli - Gabriel/Programa 3 - Dijkstra Modificado/src $ javac *.java
(base) ED2 Atividade 3 - Rondineli - Gabriel/Programa 3 - Dijkstra Modificado/src $ java Main
Dois menores caminhos:
Caminho 1: [A, B, C, D, E] com peso total de 6
Caminho 2: [A, B, C, E] com peso total de 6
(base) ED2 Atividade 3 - Rondineli - Gabriel/Programa 3 - Dijkstra Modificado/src $
```

```
1 import java.util.*;
2
3 public class DijkstraModificado<T> {
4     public List<Caminho<T>> encontrarDoisMenoresCaminhos(Grafo<T> grafo, T origem, T destino) {
5         PriorityQueue<Caminho<T>> pq = new PriorityQueue<>();
6         Caminho<T> caminhoInicial = new Caminho<>();
7         caminhoInicial.adicionarVertice(origem, 0);
8         pq.add(caminhoInicial);
9
10        List<Caminho<T>> melhoresCaminhos = new ArrayList<>();
11        Map<T, Integer> melhoresPesos = new HashMap<>();
12
13        while (!pq.isEmpty() && melhoresCaminhos.size() < 2) {
14            Caminho<T> caminhoAtual = pq.poll();
15            List<T> verticesAtuais = caminhoAtual.getVertices();
16            T ultimoVertice = verticesAtuais.get(verticesAtuais.size() - 1);
17
18            if (ultimoVertice.equals(destino)) {
19                melhoresCaminhos.add(caminhoAtual);
20                continue;
21            }
22        }
23    }
24 }
```

PROBLEMAS SAÍDA TERMINAL GITLENS CONSOLE DE DEPUAÇÃO

(base) Projetos\_Java/ED2 Atividade 3 - Rondineli - Gabriel/src \$

```
1 import java.util.List;
2 import java.util.ArrayList;
3
4 public class Main {
5     public static void main(String[] args) {
6         Grafo<String> grafo = new Grafo<>();
7         grafo.adicionarAresta("A", "B", 1);
8         grafo.adicionarAresta("A", "C", 4);
9         grafo.adicionarAresta("B", "C", 2);
10        grafo.adicionarAresta("B", "D", 5);
11        grafo.adicionarAresta("C", "D", 1);
12        grafo.adicionarAresta("C", "E", 3);
13        grafo.adicionarAresta("D", "E", 2);
14
15        DijkstraModificado<String> dijkstraMod = new DijkstraModificado<>();
16        List<Caminho<String>> resultados = dijkstraMod.encontrarDoisMenoresCaminhos(grafo, "A", "E");
17
18        System.out.println("Dois menores caminhos:");
19        for (int i = 0; i < resultados.size(); i++) {
20            Caminho<String> caminho = resultados.get(i);
21            System.out.println("Caminho " + (i + 1) + ": " + caminho.getVertices() + " com peso total de ");
22        }
23    }
24 }
```

PROBLEMAS SAÍDA TERMINAL GITLENS CONSOLE DE DEPUAÇÃO

(base) Projetos\_Java/ED2 Atividade 3 - Rondineli - Gabriel/src \$

---

## QUESTÃO 4:

**Implemente o algoritmo de busca em largura; a. Implemente um método para retornar o número de arestas existentes entre o vértice de origem da BFS e um outro vértice destino. b. Implemente um método para retornar um caminho de um vértice de origem a um vértice destino. c. Implemente um método para retornar todos os vértices que estão uma dada distância d.**

### Discussão:

A busca em largura (BFS) é uma técnica fundamental na exploração de grafos, permitindo visitar todos os vértices e arestas de maneira sistemática. O algoritmo BFS é eficaz na descoberta de caminhos mínimos em grafos não ponderados, o que significa que ele encontra a menor distância (em termos de número de arestas) entre o vértice inicial e qualquer outro vértice no grafo. O funcionamento do BFS pode ser descrito como:

1. Inicialização: O BFS começa marcando todos os vértices como não visitados (BRANCO), definindo suas distâncias como infinitas e seus predecessores como -1.
2. Fila de Visitação: O vértice inicial é marcado como em processo de visitação (CINZA), sua distância é definida como 0, e ele é inserido em uma fila.
3. Exploração: Enquanto a fila não estiver vazia, o BFS remove o vértice na frente da fila, explora todos os seus vizinhos não visitados, marca-os como em processo de visitação, atualiza suas distâncias e predecessores, e os adiciona à fila.

- 
4. Finalização: Após explorar todos os vizinhos, o vértice é marcado como completamente visitado (PRETO).

O BFS é particularmente útil para:

- Encontrar o caminho mais curto em grafos não ponderados.
- Descobrir todos os vértices a uma determinada distância do vértice inicial.
- Verificar a conectividade de um grafo.

## Descrição das Classes e Métodos:

**BFS.java:**

### Método **bfs**:

Executa a busca em largura a partir de um vértice inicial, explorando todos os vértices do grafo.

- **Linhas 1-4:** Inicializa as cores dos vértices como BRANCO (não visitados), define a distância como infinita e os predecessores como -1.
- **Linhas 5-7:** Define a cor do vértice inicial como CINZA (em processo de visitação), a distância como 0 e sem predecessor. Coloca o vértice inicial na fila.
- **Linhas 8-16:** Enquanto a fila não estiver vazia, remove o primeiro vértice da fila e explora todos os seus vizinhos. Se um vizinho não foi visitado

---

(BRANCO), marca-o como CINZA, atualiza a distância e define o predecessor. Após explorar todos os vizinhos, marca o vértice atual como PRETO (completamente visitado).

### **Método `getNumeroDeArestas`:**

Retorna o número de arestas entre o vértice inicial e um vértice de destino.

- **Linhas 1-2:** Retorna a distância armazenada no array de distâncias para o vértice especificado, que representa o número de arestas do vértice inicial até o vértice de destino.

### **Método `getCaminho`:**

Retorna o caminho entre o vértice inicial e um vértice de destino, reconstruindo-o a partir do array de predecessores.

- **Linhas 1-3:** Inicializa uma lista para armazenar o caminho e segue os predecessores desde o vértice de destino até o vértice inicial, adicionando os vértices ao caminho.
- **Linhas 4-5:** Reverte a lista do caminho para obter a ordem correta e retorna a lista.

### **Método `getVerticesNaDistancia`:**

Retorna todos os vértices que estão a uma determinada distância do vértice inicial.

- 
- **Linhas 1-5:** Percorre o array de distâncias e coleta todos os vértices que têm uma distância igual à especificada, retornando uma lista desses vértices.

## **Grafo.java:**

### **Atributos:**

- **int vertices:** Número de vértices no grafo.
- **LinkedList<Integer>[] listaAdj:** Lista de adjacências para armazenar as arestas.

### **Métodos:**

- **Grafo(int vertices):** Construtor para inicializar o grafo.
  - Inicializa um grafo com um número específico de vértices, criando uma lista de adjacências para cada vértice.
- **void adicionarAresta(int v, int w):** Adiciona uma aresta entre os vértices v e w.
  - Adiciona uma aresta dirigindo do vértice v para o vértice w, registrando essa conexão na lista de adjacências.
- **LinkedList<Integer>[] getListaAdj():** Retorna a lista de adjacências.
  - Fornece acesso à estrutura interna do grafo, retornando a lista de adjacências.
- **int getVertices():** Retorna o número de vértices no grafo.
  - Retorna o número total de vértices presentes no grafo.



---

## **Main.java:**

### **Método main:**

Executa a demonstração e os testes de busca em largura.

- **Linhas 1-5:** Inicializa o grafo pedindo ao usuário o número de vértices e as arestas a serem adicionadas.
- **Linhas 6-13:** Recebe as arestas do usuário e as adiciona ao grafo.
- **Linhas 14-16:** Recebe o vértice inicial para a BFS.
- **Linhas 17-19:** Cria uma instância do BFS e executa a busca em largura a partir do vértice inicial.
- **Linhas 20-40:** Apresenta um menu de opções para o usuário escolher diferentes funcionalidades, como mostrar o número de arestas, mostrar o caminho, mostrar vértices a uma distância específica e exibir o grafo.

**PRINTS:**

```

Digite o número de vértices no grafo: 6
-----
0 número de vértices deve começar com 1
-----
Digite as arestas do grafo no formato 'origem destino'. Digite '-1 -1' para parar.
Aresta: 1 2
Aresta adicionada: 1 -> 2
Aresta: 1 3
Aresta adicionada: 1 -> 3
Aresta: 2 4
Aresta adicionada: 2 -> 4
Aresta: 3 5
Aresta adicionada: 3 -> 5
Aresta: 4 6
Aresta adicionada: 4 -> 6
Aresta: -1 -1
Digite o vértice inicial para a BFS: 1
-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todos os vértices a uma dada distância
4. Exibir o grafo atual
5. Sair
Opção: 1
Digite o vértice de destino: 6
Número de arestas de 1 a 6: 3
-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices

```

```
Aresta: -1 -1
Digite o vértice inicial para a BFS: 1
-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todos os vértices a uma dada distância
4. Exibir o grafo atual
5. Sair
Opção: 1
Digite o vértice de destino: 6
Número de arestas de 1 a 6: 3
-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todos os vértices a uma dada distância
4. Exibir o grafo atual
5. Sair
Opção: 2
Digite o vértice de destino: 6
Caminho de 1 a 6: 1 2 4 6
-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todos os vértices a uma dada distância
4. Exibir o grafo atual
5. Sair
Opção:
```

```
-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todos os vértices a uma dada distância
4. Exibir o grafo atual
5. Sair
Opção:
```

```

Digite o número de vértices no grafo: 6
-----
0 número de vértices deve começar com 1
-----

Digite as arestas do grafo no formato 'origem destino'. Digite '-1 -1' para parar.
Aresta: 2 3
Aresta adicionada: 2 -> 3
Aresta: 2 4
Aresta adicionada: 2 -> 4
Aresta: 3 5
Aresta adicionada: 3 -> 5
Aresta: 4
6
Aresta adicionada: 4 -> 6
Aresta: -1 -1
Digite o vértice inicial para a BFS: 2
-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todos os vértices a uma dada distância
4. Exibir o grafo atual
5. Sair
Opção: 1
Digite o vértice de destino: 6
Número de arestas de 2 a 6: 2

-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices

```

```

-----
Digite o número de vértices no grafo: 4
-----
O número de vértices deve começar com 1
-----
Digite as arestas do grafo no formato 'origem destino'. Digite '-1 -1' para parar.
Aresta: 1 2
Aresta adicionada: 1 -> 2
Aresta: 2 3
Aresta adicionada: 2 -> 3
Aresta: 3 1
Aresta adicionada: 3 -> 1
Aresta: 3 4
Aresta adicionada: 3 -> 4
Aresta: -1 -1
Digite o vértice inicial para a BFS: 1
-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todos os vértices a uma dada distância
4. Exibir o grafo atual
5. Sair
Opção: 1
Digite o vértice de destino: 4
Número de arestas de 1 a 4: 3

-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todos os vértices a uma dada distância
4. Exibir o grafo atual

```

```
Número de arestas de 1 a 4: 3

-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todos os vértices a uma dada distância
4. Exibir o grafo atual
5. Sair
Opção: 2
Digite o vértice de destino: 4
Caminho de 1 a 4: 1 2 3 4

-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todos os vértices a uma dada distância
4. Exibir o grafo atual
5. Sair
Opção: 3
Digite a distância: 2
Vértices a distância 2: 3

-----
Escolha uma opção:
1. Mostrar o número de arestas entre dois vértices
2. Mostrar o caminho entre dois vértices
3. Mostrar todos os vértices a uma dada distância
4. Exibir o grafo atual
5. Sair
Opção: |
```

*(Ao desenvolver a implementação do algoritmo de busca em largura (BFS) no grafo, decidimos, ao final do algoritmo, restringir o número inicial dos vértices a 1 para simplificar a interface de usuário e evitar confusão. Como os grafos são frequentemente apresentados e manipulados utilizando índices baseados em 1, essa restrição vai garantir que os usuários forneçam entradas válidas. Essa abordagem reduziu possíveis erros com aresta inicial sendo 0)*