Estrutura de dados 2

PRIMEIRO SEMESTRE DE 2024

DOCENTE: DR. JOÃO DALLYSON

DISCENTE: GABRIEL BELO PEREIRA DOS REIS, RONDINELI SEBA SALOMAO

linha horizontal

**PROGRAMA 1: DFS**

## Implemente o algoritmo de busca em profundidade:

## Implemente um método para retornar o número de arestas existentes entre o vértice de origem da DFS e um outro vértice destino.

## Implemente um método para retornar um caminho de um vértice de origem a um vértice destino.

## 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:**

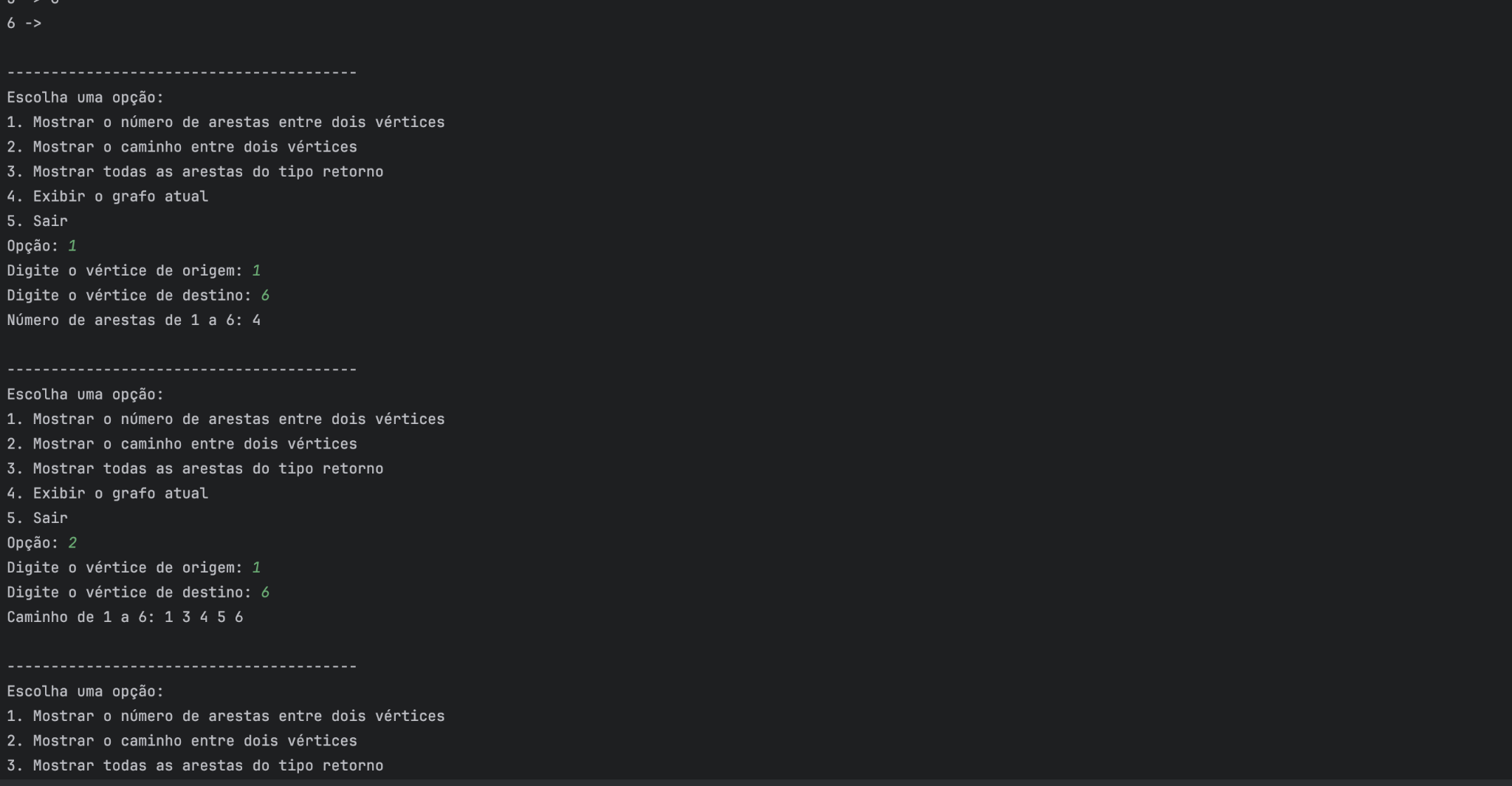
##### **Método main:**

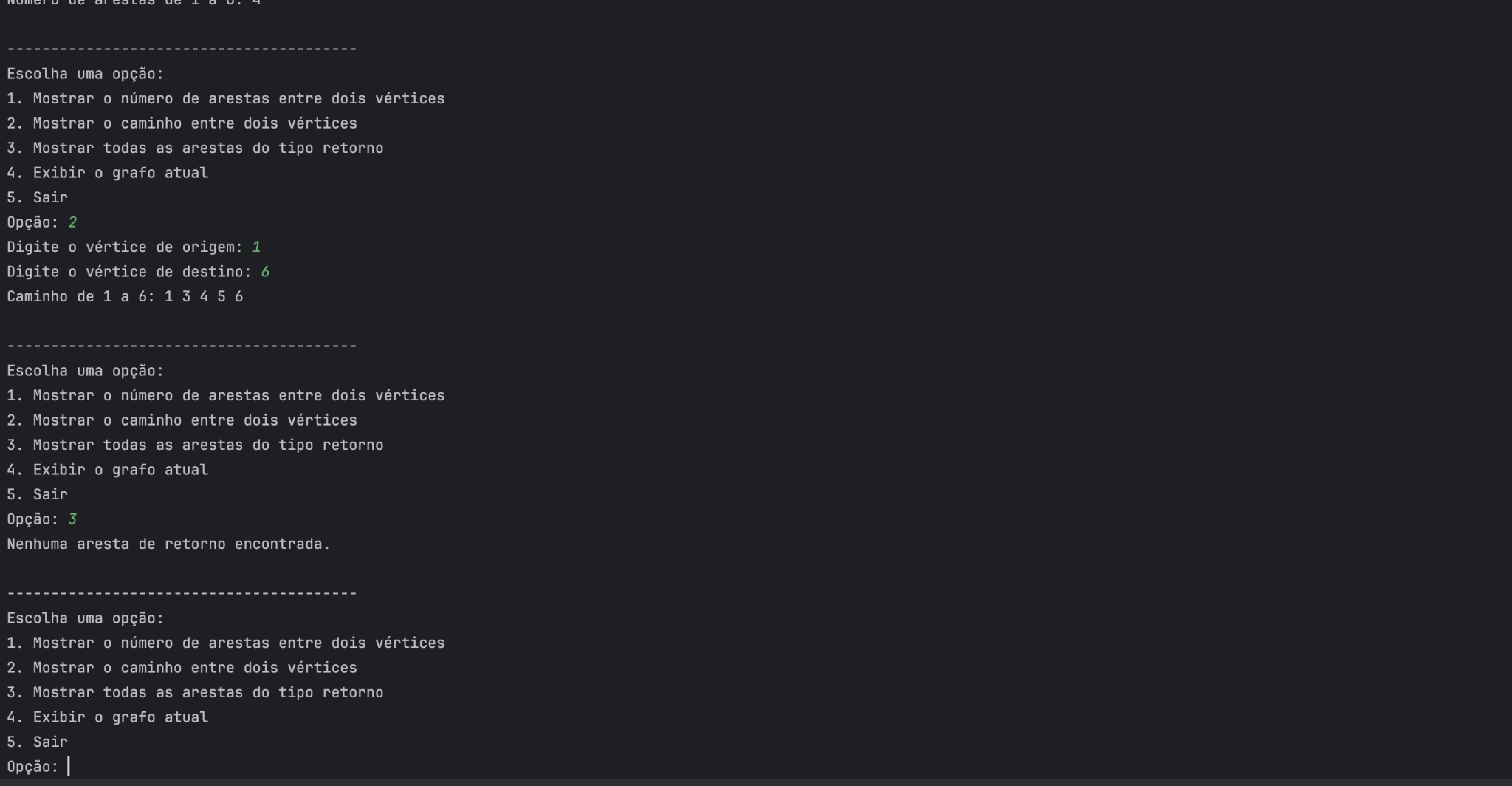
Executa a demonstração e os testes de busca em profundidade.

* **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:**





****

**PROGRAMA 2: 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:

Texto

Descrição gerada automaticamente

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 encontrarMSTretorna 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

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Texto

Descrição gerada automaticamente

Texto

Descrição gerada automaticamente

Texto

Descrição gerada automaticamente

**PROGRAMA 3: 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.

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

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**

**Texto

Descrição gerada automaticamente**

**Texto

Descrição gerada automaticamente**

**PROGRAMA 4: PROGRAMA BUSCA EM LARGURA**

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:**