| L  **Universidade Luterana do Brasil**  **ULBRA Torres**  **Pró-Reitoria de Graduação** | | Tipo de atividade:  Prova ( ) Trabalho ( ) ..... ( )  Avaliação:AP1 ( )AP2 ( ) AS ( ) AF ( ) | |
| --- | --- | --- | --- |
| Curso: | Disciplina: | | Data: |
| Turma: | Professor(a): | | Valor da Avaliação:  Nota: |
| Acadêmico(a): n°: | | |

1. Uma rede social permite que os usuários se conectem entre si. Cada usuário pode ter vários amigos. Qual é a estrutura de dados mais adequada para representar essa rede social?

a) Árvore

b) Grafo ponderado

c) Pilha

d) Lista encadeada

1. Um mapa de uma cidade é representado por um grafo, onde as ruas são as arestas e os cruzamentos são os vértices. Qual das seguintes opções representa melhor uma aresta direcionada no grafo?

a) Uma rua sem saída

b) Uma rua de mão única

c) Um cruzamento sem sinal de trânsito

d) Uma rua com várias saídas

1. Um sistema de entrega de pacotes precisa encontrar a rota mais curta entre dois pontos em um mapa. Qual algoritmo de busca é comumente usado para resolver esse problema?

a) Algoritmo de Dijkstra

b) Algoritmo de ordenação rápida (Quicksort)

c) Algoritmo de busca em profundidade (DFS)

d) Algoritmo de busca em largura (BFS)

1. Um sistema de recomendação de filmes precisa encontrar os filmes mais similares com base nos interesses do usuário. Qual algoritmo é comumente usado para encontrar grupos de vértices em um grafo?

a) Algoritmo de Kruskal

b) Algoritmo de Prim

c) Algoritmo de busca em profundidade (DFS)

d) Algoritmo de busca em largura (BFS)

1. Um sistema de recomendação de músicas precisa encontrar músicas similares com base nos gostos do usuário. Qual algoritmo é comumente usado para encontrar a similaridade entre vértices em um grafo?

a) Algoritmo de Kruskal

b) Algoritmo de Prim

c) Algoritmo de busca em profundidade (DFS)

d) Algoritmo de busca em largura (BFS)

1. Você foi designado para desenvolver um algoritmo que permita a montagem de um grafo não direcionado para representar as amizades entre um grupo de pessoas. O grafo deve ser representado utilizando a técnica de lista de adjacências. Implemente uma classe chamada "Grafo" que possua os seguintes métodos:

* Um construtor que recebe como parâmetro o número total de pessoas no grupo e inicializa a estrutura de dados necessária para representar o grafo.
* Um método chamado "adicionarAresta" que recebe dois parâmetros: o identificador das duas pessoas que são amigas e adiciona uma aresta entre elas no grafo.
* Um método chamado "imprimirGrafo" que imprime no console as conexões de cada pessoa do grupo.

Utilize o algoritmo desenvolvido para criar um grafo com as seguintes amizades:

* Pessoa 0 é amiga de pessoa 1.
* Pessoa 1 é amiga de pessoa 2.
* Pessoa 2 é amiga de pessoa 3.
* Pessoa 3 é amiga de pessoa 4.

Após criar o grafo, utilize o método "imprimirGrafo" para visualizar as conexões entre as pessoas. Desenvolva o algoritmo em Java e teste seu funcionamento utilizando o exemplo de amizades fornecido.

import java.util.ArrayList;

import java.util.List;

class Grafo {

private int numPessoas;

private List<List<Integer>> adjacencias;

public Grafo(int numPessoas) {

this.numPessoas = numPessoas;

this.adjacencias = new ArrayList<>(numPessoas);

for (int i = 0; i < numPessoas; i++) {

adjacencias.add(new ArrayList<>());

}

}

public void adicionarAresta(int pessoa1, int pessoa2) {

adjacencias.get(pessoa1).add(pessoa2);

adjacencias.get(pessoa2).add(pessoa1);

}

public void imprimirGrafo() {

for (int i = 0; i < numPessoas; i++) {

System.out.print("Pessoa " + i + " é amiga de: ");

for (Integer amigo : adjacencias.get(i)) {

System.out.print(amigo + " ");

}

System.out.println();

}

}

}

public class Main {

public static void main(String[] args) {

Grafo grafo = new Grafo(5);

grafo.adicionarAresta(0, 1);

grafo.adicionarAresta(1, 2);

grafo.adicionarAresta(2, 3);

grafo.adicionarAresta(3, 4);

grafo.imprimirGrafo();

}

}

1. Você foi contratado para criar um sistema de navegação para uma empresa de transporte público. O sistema deve encontrar a rota mais curta entre duas paradas de ônibus, considerando o tempo de viagem como peso das arestas. Implemente um algoritmo do Algoritmo de Dijkstra que receba como entrada um grafo ponderado representando as paradas de ônibus e o tempo de viagem entre elas. O algoritmo deve calcular a rota mais curta entre uma parada de origem e uma parada de destino.

Passos:

* ~~Defina uma função chamada Dijkstra que receba como parâmetros o grafo, a parada de origem e a parada de destino.~~
* ~~Inicialize uma estrutura de dados para armazenar as distâncias mínimas até cada parada de ônibus. Inicialize todas as distâncias como infinito, exceto a distância da parada de origem, que será 0.~~
* ~~Inicialize uma estrutura de dados para armazenar os nós visitados.~~
* ~~Enquanto houver nós não visitados:~~
* ~~Escolha o nó com a menor distância mínima que ainda não foi visitado.~~
* ~~Marque o nó como visitado.~~
* ~~Para cada vizinho não visitado do nó atual:~~
* ~~Calcule a distância até o vizinho somando a distância mínima do nó atual com o peso da aresta entre eles.~~
* ~~Se a nova distância for menor do que a distância mínima atual do vizinho, atualize a distância mínima.~~
* ~~Retorne a distância mínima até a parada de destino.~~
* ~~Desafio extra: Modifique o algoritmo para que ele também retorne a rota completa entre a parada de origem e a parada de destino.~~

Você pode usar o trecho do código.

import java.util.\*;

public class DijkstraAlgorithm {

public static int dijkstra(Graph graph, int source, int destination) {

////implemente aqui

}

public static void main(String[] args) {

// Exemplo de uso do algoritmo

// Criação do grafo e adição de arestas com pesos

Graph graph = new Graph(6);

graph.addEdge(0, 1, 2);

graph.addEdge(0, 2, 4);

graph.addEdge(1, 2, 1);

graph.addEdge(1, 3, 7);

graph.addEdge(2, 4, 3);

graph.addEdge(3, 4, 1);

graph.addEdge(3, 5, 5);

graph.addEdge(4, 5, 2);

int source = 0;

int destination = 5;

int shortestDistance = dijkstra(graph, source, destination);

System.out.println("A menor distância entre a parada " + source + " e a parada " + destination + " é: " + shortestDistance);

}

}

class Graph {

private int numVertices;

private int[][] adjacencyMatrix;

public Graph(int numVertices) {

this.numVertices = numVertices;

adjacencyMatrix = new int[numVertices][numVertices];

}

public void addEdge(int source, int destination, int weight) {

adjacencyMatrix[source][destination] = weight;

adjacencyMatrix[destination][source] = weight;

}

public int getNumVertices() {

return numVertices;

}

public int getWeight(int source, int destination) {

return adjacencyMatrix[source][destination];

}

}

import java.util.\*;

public class DijkstraAlgorithm {

public static int dijkstra(Graph graph, int source, int destination) {

int numVertices = graph.getNumVertices();

int[] distanciasMinimas = new int[numVertices];

boolean[] visitados = new boolean[numVertices];

int[] antecessores = new int[numVertices];

// Inicializa as distâncias mínimas com infinito e os antecessores como -1

Arrays.fill(distanciasMinimas, Integer.MAX\_VALUE);

Arrays.fill(antecessores, -1);

// A distância da parada de origem para ela mesma é 0

distanciasMinimas[source] = 0;

for (int i = 0; i < numVertices - 1; i++) {

int verticeMinimo = encontrarVerticeMinimo(distanciasMinimas, visitados);

visitados[verticeMinimo] = true;

for (int vertice = 0; vertice < numVertices; vertice++) {

int pesoAresta = graph.getWeight(verticeMinimo, vertice);

if (pesoAresta > 0 && !visitados[vertice]) {

int novaDistancia = distanciasMinimas[verticeMinimo] + pesoAresta;

if (novaDistancia < distanciasMinimas[vertice]) {

distanciasMinimas[vertice] = novaDistancia;

antecessores[vertice] = verticeMinimo;

}

}

}

}

return distanciasMinimas[destination];

}

private static int encontrarVerticeMinimo(int[] distanciasMinimas, boolean[] visitados) {

int minDistancia = Integer.MAX\_VALUE;

int minVertice = -1;

for (int vertice = 0; vertice < distanciasMinimas.length; vertice++) {

if (!visitados[vertice] && distanciasMinimas[vertice] < minDistancia) {

minDistancia = distanciasMinimas[vertice];

minVertice = vertice;

}

}

return minVertice;

}

public static void main(String[] args) {

// Exemplo de uso do algoritmo

// Criação do grafo e adição de arestas com pesos

Graph graph = new Graph(6);

graph.addEdge(0, 1, 2);

graph.addEdge(0, 2, 4);

graph.addEdge(1, 2, 1);

graph.addEdge(1, 3, 7);

graph.addEdge(2, 4, 3);

graph.addEdge(3, 4, 1);

graph.addEdge(3, 5, 5);

graph.addEdge(4, 5, 2);

int source = 0;

int destination = 5;

int shortestDistance = dijkstra(graph, source, destination);

System.out.println("A menor distância entre a parada " + source + " e a parada " + destination + " é: " + shortestDistance);

}

}

class Graph {

private int numVertices;

private int[][] adjacencyMatrix;

public Graph(int numVertices) {

this.numVertices = numVertices;

adjacencyMatrix = new int[numVertices][numVertices];

}

public void addEdge(int source, int destination, int weight) {

adjacencyMatrix[source][destination] = weight;

adjacencyMatrix[destination][source] = weight;

}

public int getNumVertices() {

return numVertices;

}

public int getWeight(int source, int destination) {

return adjacencyMatrix[source][destination];

}

}