**Question 1: (Chapter 29)**

Write a program that reads a connected weighted graph from a user. Then the program will prompt the user to enter the name of two vertices, check if they exist and if so, the program will display the shortest path between these two vertices and the weight from going from one vertex to the other.

**Driver Class**

import java.io.File;

import java.util.ArrayList;

import java.util.Scanner;

public class Driver {

public static void main(String[] args) throws Exception {

Scanner input = new Scanner (System.in);

System.out.println("Enter the file name to import the data");

String s = input.nextLine();

//Users/techyouknow/eclipse-workspace/Assignment14/dat.txt

/\*

\* 6

0 1 5 0 2 29

1 0 30 1 3 12

2 0 32 2 3 22 2 4 7

3 1 15 3 2 8 3 4 22 3 5 10

4 2 7 4 3 6 4 5 12

5 3 2 5 4 100

\*/

File file = new File (s);

String[] vertices = assignVertices(file);

ArrayList<WeightedEdge> edgeList = new ArrayList<>();

edgeList = assiginEdges(file);

WeightedGraph<String> graph1 =

new WeightedGraph<>(java.util.Arrays.asList(vertices), edgeList);

System.out.println("Checking for Two Vertices");

System.out.println("Please enter the first Vertice");

String x = String.valueOf(input.nextInt());

System.out.println("Please enter the second Vertice");

String y = String.valueOf(input.nextInt());

shortPath(x,y,graph1);

}

private static void shortPath(String x, String y, WeightedGraph<String> graph1) {

boolean a = validVertices(x,graph1);

boolean b = validVertices(y,graph1);

if (a && b) {

WeightedGraph<String>.ShortestPathTree tree1 =

graph1.getShortestPath(graph1.getIndex(x));

/\*

//alternative method - Display shortest paths

System.out.print(x + " to " + y +" Shortest path is ");

java.util.List<String> path

= tree1.getPath(graph1.getIndex(y));

for (String s: path) {

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

}

\*/

/\*displaying all path from root

/ System.out.println("\n");

tree1.printAllPaths();\*/

System.out.println("\nShortest Path & cost (weight)");

tree1.printWeightPath(y);

}

}

private static boolean validVertices(String x, WeightedGraph<String> graph2) {

boolean set = false;

if (graph2.getIndex(x) == -1) {

System.out.println("Vertice " + x + " doesnt exists");

set = false;

}

else {

System.out.println("Vertice " + x + " does exists ");

set = true;

}

return set;

}

private static ArrayList<WeightedEdge> assiginEdges(File file) throws Exception {

ArrayList<WeightedEdge> edgeList = new ArrayList<>();

Scanner read = new Scanner (file);

read.nextLine();

while(read.hasNext()) {

int j = read.nextInt();

//System.out.print(j + ",");

int k = read.nextInt();

//System.out.print(k + ",");

int l = read.nextInt();

//System.out.print(l + "\n");

edgeList.add(new WeightedEdge (j,k,l));

}

return edgeList;

}

private static String[] assignVertices(File file) throws Exception {

Scanner read = new Scanner (file);

String[] vertices = new String [read.nextInt()];

read.nextLine();

while(read.hasNextLine()) {

for (int i = 0 ; i<vertices.length; i++) {

vertices [i] = String.valueOf(read.nextInt());

read.nextLine();

}

}

read.close();

return vertices;

}

}

**WeightedEdge class**

//Code taken from Lecture slide 29

public class WeightedEdge extends Edge

implements Comparable<WeightedEdge> {

public double weight;

public WeightedEdge(int u, int v, double weight) {

super(u, v);

this.weight = weight;

}

@Override

public int compareTo(WeightedEdge edge) {

if (weight > edge.weight) {

return 1;

}

else if (weight == edge.weight) {

return 0;

}

else {

return -1;

}

}

}

**WeightedGraph class**

// modified code from Lecture 29 Slide

import java.util.\*;

public class WeightedGraph<V> extends UnweightedGraph<V> {

public WeightedGraph() {

}

public WeightedGraph(V[] vertices, int[][] edges) {

createWeightedGraph(java.util.Arrays.asList(vertices), edges);

}

public WeightedGraph(int[][] edges, int numberOfVertices) {

List<V> vertices = new ArrayList<>();

for (int i = 0; i < numberOfVertices; i++)

vertices.add((V)(new Integer(i)));

createWeightedGraph(vertices, edges);

}

public WeightedGraph(List<V> vertices, List<WeightedEdge> edges) {

createWeightedGraph(vertices, edges);

}

public WeightedGraph(List<WeightedEdge> edges,

int numberOfVertices) {

List<V> vertices = new ArrayList<>();

for (int i = 0; i < numberOfVertices; i++)

vertices.add((V)(new Integer(i)));

createWeightedGraph(vertices, edges);

}

private void createWeightedGraph(List<V> vertices, int[][] edges) {

this.vertices = vertices;

for (int i = 0; i < vertices.size(); i++) {

neighbors.add(new ArrayList<Edge>());

}

for (int i = 0; i < edges.length; i++) {

neighbors.get(edges[i][0]).add(

new WeightedEdge(edges[i][0], edges[i][1], edges[i][2]));

}

}

private void createWeightedGraph(

List<V> vertices, List<WeightedEdge> edges) {

this.vertices = vertices;

for (int i = 0; i < vertices.size(); i++) {

neighbors.add(new ArrayList<Edge>());

}

for (WeightedEdge edge: edges) {

neighbors.get(edge.u).add(edge);

}

}

public double getWeight(int u, int v) throws Exception {

for (Edge edge : neighbors.get(u)) {

if (edge.v == v) {

return ((WeightedEdge)edge).weight;

}

}

throw new Exception("Edge does not exit");

}

public void printWeightedEdges() {

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

System.out.print(getVertex(i) + " (" + i + "): ");

for (Edge edge : neighbors.get(i)) {

System.out.print("(" + edge.u +

", " + edge.v + ", " + ((WeightedEdge)edge).weight + ") ");

}

System.out.println();

}

}

public boolean addEdge(int u, int v, double weight) {

return addEdge(new WeightedEdge(u, v, weight));

}

public MST getMinimumSpanningTree() {

return getMinimumSpanningTree(0);

}

public MST getMinimumSpanningTree(int startingVertex) {

double[] cost = new double[getSize()];

for (int i = 0; i < cost.length; i++) {

cost[i] = Double.POSITIVE\_INFINITY;

}

cost[startingVertex] = 0;

int[] parent = new int[getSize()];

parent[startingVertex] = -1;

double totalWeight = 0;

List<Integer> T = new ArrayList<>();

while (T.size() < getSize()) {

int u = -1;

double currentMinCost = Double.POSITIVE\_INFINITY;

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

if (!T.contains(i) && cost[i] < currentMinCost) {

currentMinCost = cost[i];

u = i;

}

}

if (u == -1) break; else T.add(u);

totalWeight += cost[u];

for (Edge e : neighbors.get(u)) {

if (!T.contains(e.v) && cost[e.v] > ((WeightedEdge)e).weight) {

cost[e.v] = ((WeightedEdge)e).weight;

parent[e.v] = u;

}

}

}

return new MST(startingVertex, parent, T, totalWeight);

}

public class MST extends SearchTree {

private double totalWeight;

public MST(int root, int[] parent, List<Integer> searchOrder,

double totalWeight) {

super(root, parent, searchOrder);

this.totalWeight = totalWeight;

}

public double getTotalWeight() {

return totalWeight;

}

}

public ShortestPathTree getShortestPath(int sourceVertex) {

double[] cost = new double[getSize()];

for (int i = 0; i < cost.length; i++) {

cost[i] = Double.POSITIVE\_INFINITY;

}

cost[sourceVertex] = 0;

int[] parent = new int[getSize()];

parent[sourceVertex] = -1;

List<Integer> T = new ArrayList<>();

while (T.size() < getSize()) {

int u = -1;

double currentMinCost = Double.POSITIVE\_INFINITY;

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

if (!T.contains(i) && cost[i] < currentMinCost) {

currentMinCost = cost[i];

u = i;

}

}

if (u == -1) break; else T.add(u);

for (Edge e : neighbors.get(u)) {

if (!T.contains(e.v)

&& cost[e.v] > cost[u] + ((WeightedEdge)e).weight) {

cost[e.v] = cost[u] + ((WeightedEdge)e).weight;

parent[e.v] = u;

}

}

}

return new ShortestPathTree(sourceVertex, parent, T, cost);

}

public class ShortestPathTree extends SearchTree {

private double[] cost;

public ShortestPathTree(int source, int[] parent,

List<Integer> searchOrder, double[] cost) {

super(source, parent, searchOrder);

this.cost = cost;

}

public double getCost(int v) {

return cost[v];

}

public void printAllPaths() {

System.out.println("All shortest paths from " +

vertices.get(getRoot()) + " are:");

for (int i = 0; i < cost.length; i++) {

printPath(i);

System.out.println("(cost: " + cost[i] + ")");

}

}

public void printWeightPath(String d) {

for (int i = Integer.parseInt(d); i < cost.length; i++) {

printPath(i);

System.out.println("(cost: " + cost[i] + ")");

break;

}

}

}

}

**Edge.Class**

//code from chapter 28

public class Edge {

int u;

int v;

public Edge (int u, int v){

this.u = u;

this.v =v;

}

public boolean equals(Object o){

return u == ((Edge)o).u && v == ((Edge)o).v;

}

}

**Graph.class**

// code from Lecture Slides 28

public interface Graph<V> {

public int getSize();

public java.util.List<V> getVertices();

public V getVertex(int index);

public int getIndex(V v);

public java.util.List<Integer> getNeighbors(int index);

public int getDegree(int v);

public void printEdges();

public void clear();

public boolean addVertex(V vertex);

public boolean addEdge(int u, int v);

public boolean addEdge(Edge e);

public boolean remove(V v);

public boolean remove(int u, int v);

public UnweightedGraph<V>.SearchTree dfs(int v);

public UnweightedGraph<V>.SearchTree bfs(int v);

}

**UnweightedGraph class**

// code from Lecture Slides 28

import java.util.\*;

public class UnweightedGraph<V> implements Graph<V> {

protected List<V> vertices = new ArrayList<>();

protected List<List<Edge>> neighbors

= new ArrayList<>();

public UnweightedGraph() {

}

public UnweightedGraph(V[] vertices, int[][] edges) {

for (int i = 0; i < vertices.length; i++)

addVertex(vertices[i]);

createAdjacencyLists(edges, vertices.length);

}

public UnweightedGraph(List<V> vertices, List<Edge> edges) {

for (int i = 0; i < vertices.size(); i++)

addVertex(vertices.get(i));

createAdjacencyLists(edges, vertices.size());

}

public UnweightedGraph(List<Edge> edges, int numberOfVertices) {

for (int i = 0; i < numberOfVertices; i++)

addVertex((V)(new Integer(i)));

createAdjacencyLists(edges, numberOfVertices);

}

public UnweightedGraph(int[][] edges, int numberOfVertices) {

for (int i = 0; i < numberOfVertices; i++)

addVertex((V)(new Integer(i)));

createAdjacencyLists(edges, numberOfVertices);

}

private void createAdjacencyLists(

int[][] edges, int numberOfVertices) {

for (int i = 0; i < edges.length; i++) {

addEdge(edges[i][0], edges[i][1]);

}

}

private void createAdjacencyLists(

List<Edge> edges, int numberOfVertices) {

for (Edge edge: edges) {

addEdge(edge.u, edge.v);

}

}

@Override

public int getSize() {

return vertices.size();

}

@Override

public List<V> getVertices() {

return vertices;

}

@Override

public V getVertex(int index) {

return vertices.get(index);

}

@Override

public int getIndex(V v) {

return vertices.indexOf(v);

}

@Override

public List<Integer> getNeighbors(int index) {

List<Integer> result = new ArrayList<>();

for (Edge e: neighbors.get(index))

result.add(e.v);

return result;

}

@Override

public int getDegree(int v) {

return neighbors.get(v).size();

}

@Override

public void printEdges() {

for (int u = 0; u < neighbors.size(); u++) {

System.out.print(getVertex(u) + " (" + u + "): ");

for (Edge e: neighbors.get(u)) {

System.out.print("(" + getVertex(e.u) + ", " +

getVertex(e.v) + ") ");

}

System.out.println();

}

}

@Override

public void clear() {

vertices.clear();

neighbors.clear();

}

@Override

public boolean addVertex(V vertex) {

if (!vertices.contains(vertex)) {

vertices.add(vertex);

neighbors.add(new ArrayList<Edge>());

return true;

}

else {

return false;

}

}

@Override

public boolean addEdge(Edge e) {

if (e.u < 0 || e.u > getSize() - 1)

throw new IllegalArgumentException("No such index: " + e.u);

if (e.v < 0 || e.v > getSize() - 1)

throw new IllegalArgumentException("No such index: " + e.v);

if (!neighbors.get(e.u).contains(e)) {

neighbors.get(e.u).add(e);

return true;

}

else {

return false;

}

}

@Override

public boolean addEdge(int u, int v) {

return addEdge(new Edge(u, v));

}

@Override

public SearchTree dfs(int v) {

List<Integer> searchOrder = new ArrayList<>();

int[] parent = new int[vertices.size()];

for (int i = 0; i < parent.length; i++)

parent[i] = -1;

boolean[] isVisited = new boolean[vertices.size()];

dfs(v, parent, searchOrder, isVisited);

return new SearchTree(v, parent, searchOrder);

}

private void dfs(int v, int[] parent, List<Integer> searchOrder,

boolean[] isVisited) {

searchOrder.add(v);

isVisited[v] = true;

for (Edge e : neighbors.get(v)) {

int w = e.v;

if (!isVisited[w]) {

parent[w] = v;

dfs(w, parent, searchOrder, isVisited);

}

}

}

@Override

public SearchTree bfs(int v) {

List<Integer> searchOrder = new ArrayList<>();

int[] parent = new int[vertices.size()];

for (int i = 0; i < parent.length; i++)

parent[i] = -1;

java.util.LinkedList<Integer> queue =

new java.util.LinkedList<>();

boolean[] isVisited = new boolean[vertices.size()];

queue.offer(v);

isVisited[v] = true;

while (!queue.isEmpty()) {

int u = queue.poll();

searchOrder.add(u);

for (Edge e: neighbors.get(u)) {

int w = e.v;

if (!isVisited[w]) {

queue.offer(w);

parent[w] = u;

isVisited[w] = true;

}

}

}

return new SearchTree(v, parent, searchOrder);

}

public class SearchTree {

private int root;

private int[] parent;

private List<Integer> searchOrder;

public SearchTree(int root, int[] parent,

List<Integer> searchOrder) {

this.root = root;

this.parent = parent;

this.searchOrder = searchOrder;

}

public int getRoot() {

return root;

}

public int getParent(int v) {

return parent[v];

}

public List<Integer> getSearchOrder() {

return searchOrder;

}

public int getNumberOfVerticesFound() {

return searchOrder.size();

}

public List<V> getPath(int index) {

ArrayList<V> path = new ArrayList<>();

do {

path.add(vertices.get(index));

index = parent[index];

}

while (index != -1);

return path;

}

public void printPath(int index) {

List<V> path = getPath(index);

System.out.print("A path from " + vertices.get(root) + " to " +

vertices.get(index) + ": ");

for (int i = path.size() - 1; i >= 0; i--)

System.out.print(path.get(i) + " ");

}

public void printTree() {

System.out.println("Root is: " + vertices.get(root));

System.out.print("Edges: ");

for (int i = 0; i < parent.length; i++) {

if (parent[i] != -1) {

System.out.print("(" + vertices.get(parent[i]) + ", " +

vertices.get(i) + ") ");

}

}

System.out.println();

}

}

@Override

public boolean remove(V v) {

return true;

}

@Override

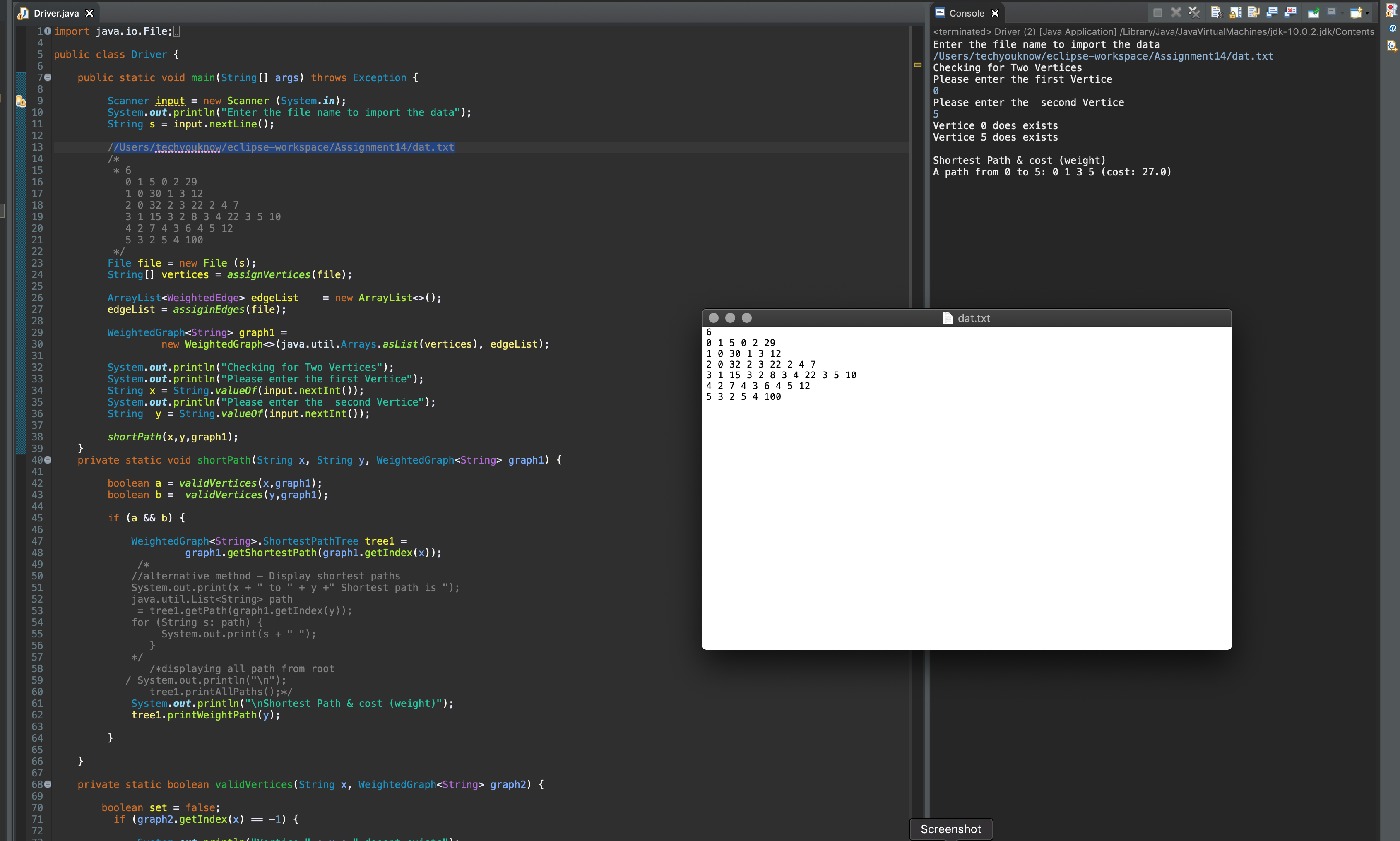
public boolean remove(int u, int v) {

return true;

}

}

Screenshot



**Question 2: (Chapter 30)**

Write a program to count the occurrence of characters in a sentence using streams.

Test your program on the following sentence:

You can use the groupingBy collector to group the elements in the stream and apply aggregate methods for the elements in the group.

Code

import java.util.\*;

import java.util.stream.Collectors;

import java.util.stream.Stream;

public class CountOccurrence {

private static int count = 0;

public static void main(String [] args) {

String s = "You can use the groupingBy collector to group the elements in the stream and apply aggregate methods for the elements in the group.";

count = 0;

System.out.println("The occurrences of each character are:");

Stream.of(toCharacterArray(s.toCharArray()))

.filter(ch -> Character.isLetterOrDigit(ch))

.map(ch -> Character.toString(ch))

.collect(Collectors.groupingBy(e -> e,

TreeMap::new, Collectors.counting()))

.forEach((k, v) -> { System.out.println( "Number of Occurrance of Character: " + k + " is " + v

);

});

}

public static Character[] toCharacterArray(char[] list) {

Character[] result = new Character[list.length];

for (int i = 0; i < result.length; i++) {

result[i] = list[i];

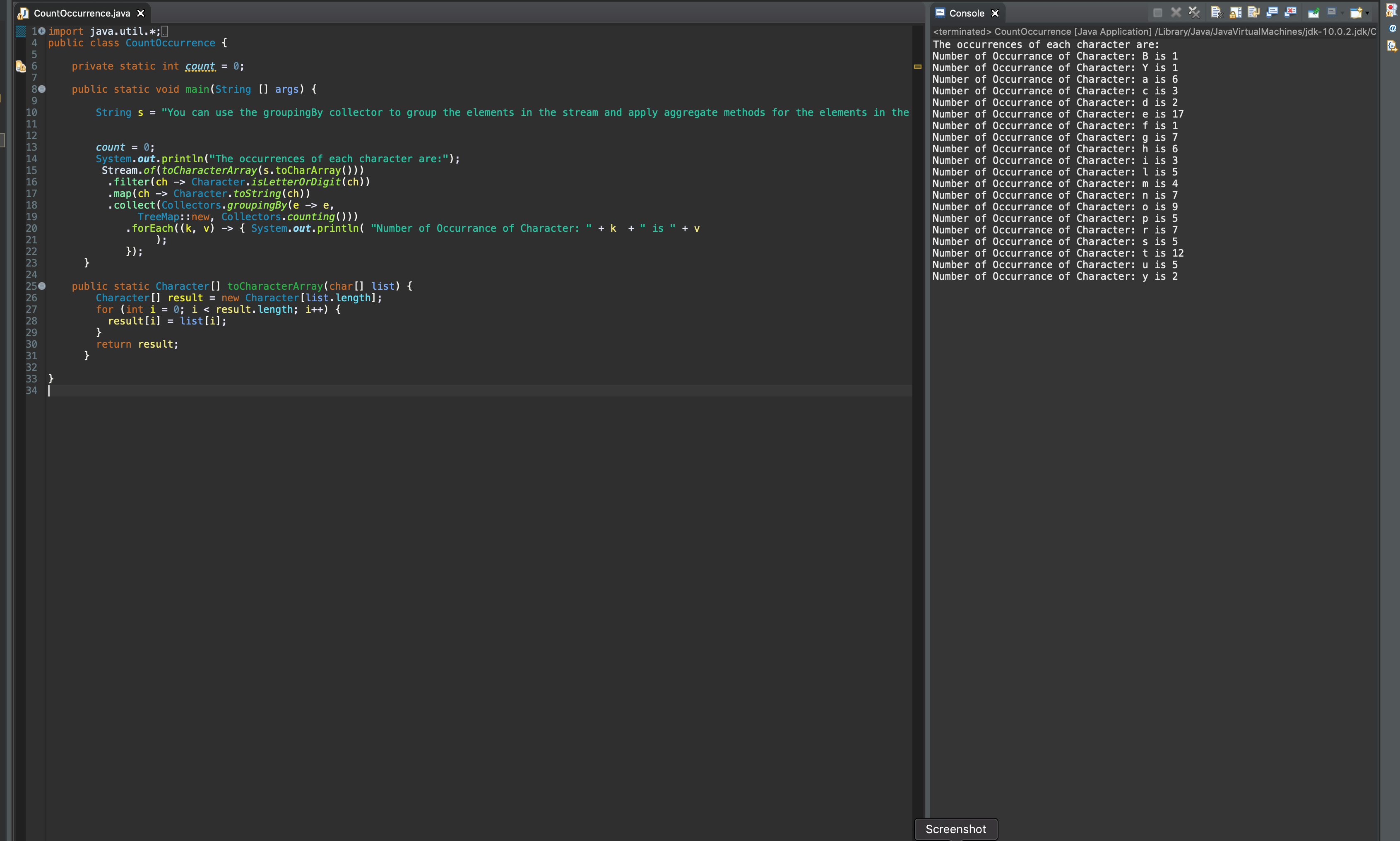
}

return result;

}

}

Screenshot

****