# Name: Kirubel Temesgen

ID: C00260396

## Computing 2: 2022-23

## DSA I- Assignment 1

Walking Tour Application

This application first prompts the user with 3 options.

* To enter a file name (absolute path)
* To search for a site/node
* To exit

On choosing the first option the file details(contents) are outputted, considering it is formatted correctly. If the file is not found the user will be prompted with the following message “File not found or is not formatted correctly”. If the file is found the user will be prompted with “Graph imported from file” the graph will then be stored as a graph object and node objects will be created from this. The user is then once again prompted with the 3 original options. On choosing the second option Search for sit by name. The user will be prompted to enter the site name. Once entered the site details and its adjacent and closest sites will be outputted. The user will then again be prompted with the 3 same options and on choosing the third option the user will be prompted with “Exiting…” and the program will end.

Description

The program is using two classes, Node and Edge, to represent a graph data structure. The graph itself is stored as a list of nodes and a list of edges, both of which are implemented using ArrayLists. This means that the graph is being stored in memory as a collection of nodes and edges, which are objects with various properties, like a name or a distance.

Its also using lists to store the graph data. Each node in the graph has an ID, a name, and a latitude/longitude coordinate, which are all stored in a Node object. Each edge has a source and destination node, as well as a distance, which are stored in an Edge object. These nodes and edges are then stored in ArrayLists.

Pseudocode

• Create a Node class with fields for id, name, latitude, and longitude

• Create a constructor for Node that takes id, name, latitude, and longitude as parameters and initializes the fields

• Create methods in Node to get the id, name, latitude, and longitude

• Create a method in Node to calculate the distance to another Node

• Create an Edge class with fields for source Node, destination Node, and distance

• Create a constructor for Edge that takes source Node, destination Node, and distance as parameters and initializes the fields

• Create methods in Edge to get the source Node, destination Node, and distance

• Create a Graph class to represent the graph

• Add a field to Graph to store a list of Nodes

• Add a field to Graph to store a list of Edges

• Create methods in Graph to add Nodes and Edges to the right lists

• Create a method in Graph to find the closest and second closest Node to a given latitude and longitude

• In the closest Node method, iterate through the list of Nodes and calculate the distance to each Node using the distanceTo method in Node

• Return the Node with the smallest distance

• Create a method in Graph to find all adjacent Nodes to a given Node

• In the adjacent Nodes method, iterate through the list of Edges and check if the given Node is the source or destination of each Edge

• If the given Node is the source of an Edge, add the destination Node to a list of adjacent Nodes

• If the given Node is the destination of an Edge, add the source Node to the list of adjacent Nodes

• Return the list of adjacent Nodes

Java Code

MENU CLASS

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

import java.util.List;

// Menu class which contains the main method and menu options

public class Menu {

    private static Scanner scanner = new Scanner(System.in);

    private static Graph graph; // Graph object to hold imported graph data

    public static *void* main(String[] *args*) {

*boolean* exit = false;

        while (!exit) {

            // Print menu options

            System.out.println("1. Import graph from file");

            System.out.println("2. Search site by name");

            System.out.println("3. Exit");

            System.out.print("Enter choice: ");

            // Read user choice

*int* choice = scanner.nextInt();

            scanner.nextLine(); // consume newline character

            switch (choice) {

                case 1:

                    System.out.print("Enter absolute filepath: ");

                    String filename = scanner.nextLine();

                    try {

                        // Try to read graph from file

                        graph = readGraphFromFile(filename);

                        System.out.println(graph);

                        System.out.println("Graph imported from file.");

                    } catch (FileNotFoundException e) {

                        // File not found or not formatted correctly

                        System.out.println("File not found or not formatted correctly.");

                    }

                    break;

                case 2:

                    if (graph == null) {

                        // Check if graph is not found and prompt user to import it first

                        System.out.println("Graph not found. Please import graph from file first.");

                        break;

                    }

                    System.out.print("Enter site name: ");

                    String siteName = scanner.nextLine();

                    Node siteNode = graph.getNodeByName(siteName);

                    if (siteNode == null) {

                        // Check if site is not found

                        System.out.println("Site not found.");

                        break;

                    }

                    System.out.println("\nClosest site:");

                    // Find the closest node to the site

                    Node closestNode = graph.getClosestNode(siteNode);

                    System.out.println(closestNode);

                    System.out.println("\nNeighbour sites:");

                    // Find the adjacent sites node to the site

                        System.out.println(siteNode.getAdjacentNodes());

                    // Create a 2D array of edges

*int* edgeSite0 = (*int*) Math.abs(siteNode.getLatitude() - closestNode.getLatitude());

*int* edgeSite1 = (*int*) Math.abs(siteNode.getLongitude() - closestNode.getLongitude());

                    Edge[][] edges = new Edge[edgeSite0][edgeSite1];

                    // Populate the array with edges

                    for (Edge edge : graph.getEdges()) {

*int* row = (*int*) Math.abs(siteNode.getLatitude() - closestNode.getLatitude());

*int* col = (*int*) Math.abs(siteNode.getLongitude() - closestNode.getLongitude());

                        edges[row][col] = edge;

                    }

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

                        for (*int* j = 0; j < edges[i].length; j++) {

                            System.out.print(edges[i][j] + " ");

                        }

                        System.out.println(); // Move to next line after printing all elements in the row

                    }

                    break;

                case 3:

                    // Exit the program

                    System.out.println("Exiting...");

                    exit = true;

                    break;

                default:

                    // Invalid choice

                    System.out.println("Invalid choice. Try again.");

                    break;

            }

        }

        scanner.close();

    }

    private static Graph readGraphFromFile(String *filename*) throws FileNotFoundException {

        Graph graph = new Graph();

        File file = new File(*filename*);

        Scanner scanner = new Scanner(file);

        while (scanner.hasNextLine()) {

            String line = scanner.nextLine();

            String[] parts = line.split(",");

*int* id = Integer.parseInt(parts[0]);

            String name = parts[1];

*double* latitude = Double.parseDouble(parts[2]);

*double* longitude = Double.parseDouble(parts[3]);

            Node node = new Node(id, name, latitude, longitude);

            if (parts.length > 4) { // check if there are edges

                String[] edgeParts = parts[4].split("\\|"); // split edges by pipe symbol

                for (String edgePart : edgeParts) {

                    String[] edgeData = edgePart.split(":"); // split edge data by colon symbol

*int* neighborId = Integer.parseInt(edgeData[0]);

*double* distance = Double.parseDouble(edgeData[1]);

                    node.addEdge(neighborId, distance); // add each edge to the node

                }

            }

            graph.addNode(node);

        }

        scanner.close();

        return graph;

    }

    public static *void* printEdges(Graph *graph*) {

        List<Edge> edges = *graph*.getEdges();

        for (Edge edge : edges) {

            System.out.println(edge.getSource() + " -> " + edge.getDestination() + " : " + edge.getDistance());

        }

    }

}

GRAPH CLASS

import java.util.ArrayList;

import java.util.List;

public class Graph {

    private List<Node> nodes; // List of nodes in the graph

    private List<Edge> edges; // List of edges in the graph

    public Graph() {

        nodes = new ArrayList<>(); // Initialize the nodes list

        edges = new ArrayList<>(); // Initialize the edges list

    }

    public *void* addNode(Node *node*) {

        nodes.add(*node*); // Add a node to the nodes list

    }

    public *void* addEdge(*int* *nodeId1*, *int* *nodeId2*, *double* *distance*) {

        Node node1 = nodes.get(*nodeId1*);

        Node node2 = nodes.get(*nodeId2*);

        node1.addEdge(node2.getId(), *distance*);

        node2.addEdge(node1.getId(), *distance*);

    }

    public List<Node> getNodes() {

        return nodes; // Get the nodes list

    }

    public List<Edge> getEdges() {

        return edges; // Get the edges list

    }

    public Node getNodeByName(String *name*) {

        for (Node node : nodes) { // Iterate over the nodes list

            if (node.getName().equalsIgnoreCase(*name*)) { // If the node's name matches the given name (case-insensitive)

                return node; // Return the node

            }

        }

        return null; // If no node is found, return null

    }

    public Node getClosestNode(Node *node*) {

        Node closest = null; // Initialize the closest node to null

*double* minDistance = Double.MAX\_VALUE; // Initialize the minimum distance to the maximum possible value of a double

        for (Node otherNode : nodes) { // Iterate over the nodes list

            if (*node*.equals(otherNode)) { // If the current node is the same as the given node, skip it

                continue;

            }

*double* distance = *node*.distanceTo(otherNode); // Calculate the distance between the current node and the given node

            if (distance < minDistance) { // If the calculated distance is less than the current minimum distance

                closest = otherNode; // Update the closest node to the current node

                minDistance = distance; // Update the minimum distance to the calculated distance

            }

        }

        return closest; // Return the closest node

    }

    public Node getSecondClosestNode(Node *node*) {

        Node closest = null; // Initialize the closest node to null

        Node secondClosest = null; // Initialize the second closest node to null

*double* minDistance = Double.MAX\_VALUE; // Initialize the minimum distance to the maximum possible value of a double

*double* secondMinDistance = Double.MAX\_VALUE; // Initialize the second minimum distance to the maximum possible value of a double

        for (Node otherNode : nodes) { // Iterate over the nodes list

            if (*node*.equals(otherNode)) { // If the current node is the same as the given node, skip it

                continue;

            }

*double* distance = *node*.distanceTo(otherNode); // Calculate the distance between the current node and the given node

            if (distance < minDistance) { // If the calculated distance is less than the current minimum distance

                secondClosest = closest; // Update the second closest node to the previous closest node

                secondMinDistance = minDistance; // Update the second minimum distance to the previous minimum distance

                closest = otherNode; // Update the closest node to the current node

                minDistance = distance; // Update the minimum distance to the calculated distance

            } else if (distance < secondMinDistance) { // If the calculated distance is less than the second minimum distance but greater than or equal to the minimum distance

                secondClosest = otherNode; // Update the second closest node to the current node

                secondMinDistance = distance; // Update the second minimum distance to the calculated distance

            }

        }

        return secondClosest; // Return the second closest node

    }

    public String toString() {

        StringBuilder sb = new StringBuilder();

        sb.append("Nodes:\n");

        for (Node node : nodes) { // Iterate over the nodes list

            sb.append(String.format("%d: %s (%f, %f)\n", node.getId(), node.getName(), node.getLatitude(), node.getLongitude())); // Append the node's id, name, latitude, and longitude to the StringBuilder

        }

        sb.append("\n");

        for (Edge edge : edges) { // Iterate over the edges list

            sb.append(String.format("%s -- %f --> %s\n", edge.getSource().getName(), edge.getDistance(), edge.getDestination().getName())); // Append the source node's name, distance, and destination node's name to the StringBuilder

        }

        return sb.toString(); // Return the StringBuilder as a String

    }

}

EDGE CLASS

import java.util.List;

import java.util.ArrayList;

public class Edge {

    // Fields

    private Node source; // starting node of the edge

    private Node destination; // ending node of the edge

    private *double* distance; // distance between the two nodes in the edge

    // Constructor

    public Edge(Node *source*, Node *destination*, *double* *distance*) {

        this.source = *source*;

        this.destination = *destination*;

        this.distance = *distance*;

    }

    public Edge(Node *source*, Node *destination*) {

        this.source =*source*;

        this.destination =*destination*;

    }

    // Getter methods

    public Node getSource() {

        return source;

    }

    public Node getDestination() {

        return destination;

    }

    public *double* getDistance() {

        return distance;

    }

    public String toString() {

        return String.format("%s to %s (%f)", source, destination, distance);

    }

}

NODE CLASS

import java.util.List;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.Map;

public class Node {

    private *int* id;

    private String name;

    private *double* latitude;

    private *double* longitude;

    private Map<Integer, Double> adjacencyList;

    public Node(*int* *id*, String *name*, *double* *latitude*, *double* *longitude*) {

        this.id = *id*;

        this.name = *name*;

        this.latitude = *latitude*;

        this.longitude = *longitude*;

        adjacencyList = new HashMap<Integer, Double>();

    }

    public *int* getId() {

        return id;

    }

    public String getName() {

        return name;

    }

    public *double* getLatitude() {

        return latitude;

    }

    public *double* getLongitude() {

        return longitude;

    }

    public *double* distanceTo(Node *otherNode*) {

*double* xDiff = this.latitude - *otherNode*.latitude;

*double* yDiff = this.longitude - *otherNode*.longitude;

        return Math.sqrt(xDiff \* xDiff + yDiff \* yDiff);

    }

    public *void* addEdge(*int* *nodeId*, *double* *distance*) {

        adjacencyList.put(*nodeId*, *distance*);

    }

    public List<Integer> getAdjacentNodes() {

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

        for (Integer nodeId : adjacencyList.keySet()) {

            adjacentNodes.add(nodeId);

        }

        return adjacentNodes;

    }

    public *double* getDistanceTo(*int* *nodeId*) {

        if (adjacencyList.containsKey(*nodeId*)) {

            return adjacencyList.get(*nodeId*);

        } else {

            return Double.POSITIVE\_INFINITY;

        }

    }

    @*Override*

    public String toString() {

        return String.format("%s (%f, %f)", name, latitude, longitude);

    }

}

Short Description

The methods used in the code include creating nodes and edges, calculating the distance between nodes, getting a list of adjacent nodes, finding the closest node, and getting the shortest path between two nodes.

Test Data

nodes.csv

0,Carlow Castle,52.8364,-6.9336,1:20.0|6:40.0

1,Browneshill Dolmen,52.8291,-6.7846,0:20.0|2:30.0

2,Altamont Gardens,52.7327,-6.8823,1:30.0|3:40.0

3,Carlow County Museum,52.8366,-6.9333,2:40.0|4:10.0

4,Delta Sensory Gardens,52.8541,-6.9089,3:10.0|5:25.0

5,Huntington Castle and Gardens,52.6531,-6.6014,4:25.0|6:35.0

6,St. Mullins,52.5105,-6.9295,0:40.0|5:35.0

Screen Captures

