Java-Based Graph Security Assignment

Title: Route Defense: Navigating Adversarial Paths in El Paso

Introduction:

Graph theory is a cornerstone of computer science, providing a robust framework for tackling complex problems involving networks and paths. An understanding of adversarial risks within graph-based systems is critical, as real-world networks often face threats such as route manipulation or node removal. This assignment focuses on analyzing routing paths across familiar locations in El Paso, considering both efficient pathfinding and adversarial robustness.

Programming Task:

Implement a Java-based program that represents a network of locations in El Paso, modeled as a graph. This network should be equipped to find the fastest route between different points and withstand potential adversarial attacks, such as increased travel times on specific routes. Through this, students will practice remembering and applying graph-theoretical concepts.

Code Requirements:

- **Graph Representation:** Utilize `HashMap` and `ArrayList` to represent the graph's nodes (locations) and edges (routes between locations).

- **Shortest Path Algorithm:** Implement Dijkstra's Algorithm to determine the fastest route between any two locations.

- **Adversarial Simulation:** Develop a function that simulates adversarial attack by increasing weights (travel times) on particular edges to simulate manipulation.

- **Resilience and Detection:** Implement mechanisms to detect suspicious changes in route weights and validate the integrity of discovered paths.

Java Implementation Details:

```java

import java.util.*;

class Location {

String name;

double latitude, longitude;

Location(String name, double latitude, double longitude) {
```

```
this.name = name;
this.latitude = latitude;
this.longitude = longitude;
}
}
class Graph {
private final Map<Location, List<Edge>> adjVertices = new HashMap<>();
void addLocation(Location loc) {
adjVertices.putIfAbsent(loc, new ArrayList<>());
}
void addEdge(Location from, Location to, double distance) {
adjVertices.get(from).add(new Edge(from, to, distance));
adjVertices.get(to).add(new Edge(to, from, distance)); // assuming undirected graph
}
List<Location> getVertices() {
return new ArrayList<>(adjVertices.keySet());
}
List<Edge> getEdges(Location loc) {
return adjVertices.get(loc);
}
double getDistance(Location from, Location to) {
for (Edge edge : adjVertices.get(from)) {
if (edge.to.equals(to)) {
return edge.distance;
}
}
return Double.POSITIVE_INFINITY;
```

```
}
}
class Edge {
Location from, to;
double distance;
Edge(Location from, Location to, double distance) {
this.from = from;
this.to = to;
this.distance = distance;
}
}
public class RouteAnalysis {
private static final double ATTACK_MULTIPLIER = 1.5; // increases edge weight by a factor
public static void main(String[] args) {
Graph elPasoGraph = new Graph();
// Add locations (nodes)
Location buffaloWildWings = new Location("Buffalo Wild Wings", 31.7547607, -106.3475634);
Location viscountDental = new Location("Viscount Dental Associates", 31.7693548, -106.3648582);
// add other locations ...
// Add edges (routes)
elPasoGraph.addLocation(buffaloWildWings);
elPasoGraph.addLocation(viscountDental);
double exampleDistance = calculateDistance(buffaloWildWings, viscountDental);
elPasoGraph.addEdge(buffaloWildWings, viscountDental, exampleDistance);
// Implement the Dijkstra's algorithm call and analyze routes
List<Location> shortestPath = dijkstra(elPasoGraph, buffaloWildWings, viscountDental);
System.out.println("Shortest Path: " + shortestPath);
```

```
// Simulate adversarial attack
simulateAttack(elPasoGraph, buffaloWildWings, viscountDental);
// Detect and respond to attacks
validateGraph(elPasoGraph);
}
private static double calculateDistance(Location loc1, Location loc2) {
// Uses Haversine formula to calculate distance based on coordinates (simplified example)
return Math.sqrt(Math.pow(loc1.latitude - loc2.latitude, 2) + Math.pow(loc1.longitude - loc2.longitude, 2));
}
private static List<Location> dijkstra(Graph graph, Location start, Location end) {
// Implement Dijkstra's algorithm to find shortest path
// ... code for Dijkstra's algorithm ...
return new ArrayList<>();
}
private static void simulateAttack(Graph graph, Location from, Location to) {
List<Edge> edges = graph.getEdges(from);
for (Edge edge : edges) {
if (edge.to.equals(to)) {
edge.distance *= ATTACK_MULTIPLIER;
break;
}
}
System.out.println("Simulated attack on route from " + from.name + " to " + to.name);
}
private static void validateGraph(Graph graph) {
// Check for suspicious changes in edge weights
```

```
// ... add checks and responses ...

System.out.println("Graph validated, no abnormalities detected");

}

}
```

Expected Output:

A Java-based program that accurately calculates the fastest routes among specified locations in El Paso, while also effectively identifying and mitigating route manipulations caused by adversarial attacks.

Adversarial Angle:

Emphasizes detection of manipulated weights within the graph due to attacks, rigorous validation of paths post-attack, and ensures continued network resilience.

Sense of Belonging:

Students will interact with real points of interest around the El Paso area, fostering a connection between academic exercises and the local community where they learn and live.