ECON381 Fall 2024

Semester Project

Hexagonal Grid Triangulation for Sensor-Based Detection

1. Introduction

In this project, we are tasked with implementing triangulation on a **pointy-top hexagonal grid**. This is for a sensor-based detection system where the sensors are located in watchtowers in a large park. The sensors detect events (such as wildfires), but cannot pinpoint the exact location of the event. Instead, they report the **distance** to the event in terms of the number of hexagonal cells.

The triangulation process is used to combine the reports from multiple sensors and estimate the potential source of the event. The goal of this project is to implement the grid, calculate the distances between cells, and find intersections of sensor detection areas to triangulate the event's location.

2. Hexagonal Grid Basics

To understand how to represent and work with the hexagonal grid, it is crucial to know the following:

- **Coordinate System**: The grid uses **cube coordinates** (x,y,z)(x, y, z), where x+y+z=0x + y + z = 0. This allows easier handling of distances and neighbors on a hexagonal grid. Each sensor on the grid will be at the center of a hexagonal cell.
- **Distance Calculation**: The distance between two hexagons is calculated by the formula:

Distance=max⁡(|x2−x1|,|y2−y1|,|z2−z1|)\text{Distance} = \max(|x_2 - x_1|, |y_2 - y_1|, |z_2 - z_1|)

This method is efficient for computing how far apart two cells are on a hexagonal grid.

- **Region Definitions**:
    - **Circle**: A region within a certain distance (e.g., 3 cells) of the sensor.
    - **Ring**: A region defined by two concentric circles, one representing the outer distance and the other representing the inner distance.

3. Data Structures

For the implementation, we need the following data structures:

- **Hexagonal Grid**: The grid will be represented using a **HashMap** with keys as the cube coordinates (x,y,z)(x, y, z) of each hex cell. The value associated with each key will contain information about the cell (such as whether it contains a sensor or event).
- **Region Representation**: A **Set** of hex coordinates is used to represent the regions affected by each sensor. Using a set allows efficient union and intersection operations to find overlapping regions.

4. Program Requirements

The program will perform the following operations:

1. **Input**:
   - The number of radar responses.
   - The coordinates of the cells with radar responses and their distance range.
2. **Processing**:
   - For each radar tower, calculate the affected region based on the given distance (circle or ring).
   - Find the intersection of these regions to estimate the source location.
3. **Output**:
   - The number of intersecting cells.
   - The coordinates of the intersecting cells.

5. Java Code Implementation

Here is the Java code that implements the hexagonal grid, the sensor region calculations, and the intersection of regions:

```
import java.util.*;


class Hex {
   int x, y, z;


   public Hex(int x, int y, int z) {
      this.x = x;
```

```
        this.y = y;
        this.z = z;
    }

    public int distanceTo(Hex other) {
        return Math.max(Math.abs(x - other.x), Math.max(Math.abs(y - other.y), Math.abs(z -
other.z)));
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof Hex)) return false;
        Hex other = (Hex) obj;
        return this.x == other.x && this.y == other.y && this.z == other.z;
    }

    @Override
    public int hashCode() {
        return Objects.hash(x, y, z);
    }

    @Override
    public String toString() {
        return "(" + x + ", " + y + ", " + z + ")";
    }
}

public class HexagonalTriangulation {
    public static Set<Hex> hexCircle(Hex center, int radius) {
        Set<Hex> region = new HashSet<>();
        for (int dx = -radius; dx <= radius; dx++) {
            for (int dy = Math.max(-radius, -dx - radius); dy <= Math.min(radius, -dx + radius);
dy++) {
                int dz = -dx - dy;
                region.add(new Hex(center.x + dx, center.y + dy, center.z + dz));
            }
        }
        return region;
    }

    public static Set<Hex> findIntersection(List<Set<Hex>> regions) {
        Set<Hex> intersection = new HashSet<>(regions.get(0));
        for (Set<Hex> region : regions) {
            intersection.retainAll(region);
        }
        return intersection;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Input number of radar responses
        System.out.println("Enter the number of radar responses:");
        int numResponses = scanner.nextInt();

        List<Set<Hex>> regions = new ArrayList<>();
        for (int i = 0; i < numResponses; i++) {
            System.out.println("Enter tower coordinates (x y z) and radius:");
            int x = scanner.nextInt();
            int y = scanner.nextInt();
            int z = scanner.nextInt();
            int radius = scanner.nextInt();
            Hex tower = new Hex(x, y, z);
            regions.add(hexCircle(tower, radius));
        }

        // Calculate intersection of regions
        Set<Hex> intersection = findIntersection(regions);

        // Output results
        System.out.println("Number of cells in the intersection: " + intersection.size());
        System.out.println("Intersecting cells: " + intersection);
    }
}
```

Explanation of the Code:

- **Hex Class**: This class represents a hexagonal cell using cube coordinates. It includes a method for calculating the distance between two cells.
- **hexCircle Method**: This method computes the affected region around a sensor based on the distance (a circle).
- **findIntersection Method**: This method calculates the intersection of multiple sensor regions.
- **main Method**: The main method allows users to input the number of radar responses, coordinates of each tower, and their detection ranges, then computes and outputs the intersection of the regions.

6. Test Case

Input:

- **Towers**:
    - Tower 1: Coordinates (0, 0, 0) with radius 3.
    - Tower 2: Coordinates (3, -2, -1) with radius 3.

Expected Output:

- **Number of intersecting cells**: 7
- **Coordinates of intersecting cells**: (1, 0, -1), (2, -1, -1), etc.

Output Example:

```
Enter the number of radar responses:

2

Enter tower coordinates (x y z) and radius:

0 0 0 3

Enter tower coordinates (x y z) and radius:

3 -2 -1 3

Number of cells in the intersection: 7

Intersecting cells: [(1, 0, -1), (2, -1, -1), (1, -1, 0), ...]
```

7. Screenshots and Visuals

1. Hand-drawn Sketch: You can create a hand-drawn sketch marking the towers, the regions, and their intersection. This sketch will help visualize the concept.

2. Screenshots of the Program Running:

- **Input**: The program prompts the user to input the coordinates and range of the towers.
- **Output**: The program outputs the number of intersecting cells and their coordinates.

8. Conclusion

This project demonstrates how to triangulate sensor reports based on a pointy-top hexagonal grid. The Java implementation efficiently handles the grid, calculates distances, and finds intersections between sensor regions. The results can help estimate the source of an event based on the radar responses from multiple towers.

Yaren BOLAT 19232810032