**Informatics Large Practical**

**Coursework 2**  
s1803764

# **Table of Contents**

1. **Software architecture description**
2. **Class documentation**
3. **Drone control algorithm**

# **SOFTWARE ARCHITECTURE DESCRIPTION**

# **Project description** To program a drone to fly around and collect sensor readings from sensors in a defined confinement area. This drone can only move in steps of 0.0003 degrees and can only turn in degrees of 10. This drone must also avoid no-fly-zones in the area whilst mapping its route. After collecting readings from all the sensors this drone must return to it’s start point and output the readings of the sensors (to a .geojson file) and a log of the movements it made during its flight (to a .txt file).

# **Software architecture summary**

**1) Retrieving command line inputs**  
This program works by first retrieving all the relevant command line arguments that specify the date we want to map a route for, the starting point (in longitude and latitude), and the port of connection. Upon retrieval these inputs are validated for data type and correctness (eg. does not allow impossible dates). If one of these inputs is invalid, the program terminates.

**2) Connecting to the web server**  
Once these inputs are retrieved and validated our program first tries to connect to the web server at the specified port using the java HTTP class. If this fails, the program terminates.

**3) Retrieving the relevant data from the web server and parsing it into Java objects**

**3.1) Retrieving and parsing the maps file**  
Assuming a successful connection to the web server our program retrieves the data for all the sensors to be read for the specified date. It does this by retrieving the *air-quality-data.json* file for the given date (in the *maps/YYYY/MM/DD/* directory) which stores the data for all the sensors: the What3Words location, the battery percentage, and the air-quality reading. These are parsed using substring indexing and are stored in custom ‘Sensor’ objects.

**3.2) Retrieving and parsing the What3Words files**  
Now that we have the sensor data, we must retrieve the respective coordinates for each What3Words location. Our program does this by iterating through all the different sensors to retrieve the *details.json* file for each given What3Words location (in the *words/W1/W2/W3/* directory). The only data we need from this file is the ‘coordinates’ object which represents the centre of the respective What3Words tile. These are parsed using substring indexing and are stored in their respective ‘Sensor’ objects. All of the sensors are then stored in one global ArrayList of Sensor objects called ‘sensors’.

**3.3) Retrieving and parsing the no-fly-zones file**  
The last thing we need to retrieve from the web server is the no fly zones for the drone, these represent tall buildings which the drone must avoid in order to prevent a crash. Our program does this by retrieving the *no-fly-zones.geojson* file (in the *buildings/* directory). This file contains a Geo-JSON FeatureCollection which stores each no-fly-zone as a feature. The only data we need from each of these features is the ‘Polygon’ geometry object which stores an array of coordinates representing the vertices for the given no-fly-zone. These are parsed using substring indexing and are stored in custom ‘Building’ objects. All of the buildings are then stored in one global ArrayList of Building objects called ‘buildings’.

**4) Find optimal sensor route**  
We use an algorithm to determine the optimal route to visit sensors. I implemented many different algorithms to find the best based on both performance and execution time.

**5) Calculate and record valid moves for the drone to follow this route**  
We start at the coordinates specified by the input arguments. Our program then iterates through the *findPoint()* method which returns a valid move that takes the drone closer to the next sensor in the queue. This is iterated until the drone has visited all the sensors in the route. These moves are calculated using planar trigonometry and will be discussed in detail further on.

Java classes used:

* Java.io.FileWriter
* Java.io.IOException
* Java.util.ArrayList
* Java.util.Arrays
* Java.net.URI
* Java.net.http.\*

# **CLASS DOCUMENTATION**

# **Custom classes**

# **Point**

s

# **Sensor** (extends Point)

s

# **Building**

s

# **LineGraph**

s

# **Move**

s

# **Fragment** (custom class for the *temperate()* algorithm) s

Draw a class diagram

Note usage of Java’s object hierarchy

# **DRONE CONTROL ALGORITHM**

# **Finding the optimal sensor route** In order to decide the optimal sensor route I decided to try lots of different algorithms to find what was optimal.

# **Algorithms used** After experience with the travelling salesman problem I had a good idea of what algorithms I wanted to try use for this drone. **Initial route setting algorithms:** these set a route based purely on distances between sensors.

* **Greedy**  
  The Greedy algorithm works by iterating through the sensors the drone needs to visit and chooses a route based on which sensor is closest to the last.  
    
  The route is initialized with the first sensor in the list. Thereafter route expansion is done by adding the next closest available sensor. This process is continued until a complete route is formed.
* **Temperate (custom algorithm)**  
  I wanted to create an algorithm that finds optimal paths by prioritizing the sensors which­   
  have the highest mean distances from other sensors. This prioritization works by creating fragments/edges (represents the transition between two sensors) between the sensor with the highest mean distance and expanding it with its best possible path (the closest available sensor). This is iterated for all sensors from highest to lowest mean distance and placed in a priority queue (descending order of mean values). To prevent redundancy in this priority queue we only allow a given sensor to be in a maximum of two fragments (as a single sensor can be connected to a maximum of two other sensors). Once a priority queue of fragments/edges is found then the route can begin to be created. The route is initialized with the first fragment in the priority queue. Thereafter route expansion is based upon availability in the priority queue. By this I mean that based upon the given sensor we need to expand (last sensor in the current route), the algorithm first checks if this sensor can be found in a fragment in the priority queue. If so, the other sensor from this given fragment is added to the route. Otherwise, the algorithm finds the best available transition for the given sensor and adds it to the route. This process is continued until a complete route is formed. In order to ensure maximum efficiency redundant fragments (fragments which contain sensors that are not available) are deleted when found upon each iteration.  
    
  This algorithm almost works in the opposite way to the greedy algorithm in which rather than prioritizing using the shortest distances possible, mine prioritizes **not** using the longest distances possible, this is why I decided to name this algorithm ‘Temperate’ (the antonym of greedy).

**Route refinement algorithms:** these naively switch points in the route and see if this improves the overall route cost (total distance travelled).

* **Swap heuristic**  
  The Swap heuristic algorithm works by swapping adjacent sensors in the route to see if it improves the cost.  
    
  This algorithm iterates through each element in the given route to try all possible adjacent swaps. If a single swap in the loop is successful (improves the overall cost) we iterate through the route again (because the route has been changed). If no successful swaps are made throughout an entire loop the algorithm is terminated.
* **2-Opt heuristic**  
  The 2-Opt heuristic algorithm works by flipping the path between two sensors in the route to see if it improves the cost.  
    
  This algorithm uses a nested loop in order to get two indexes that represent sensors in the route. For every iteration we then try reverse the path between these two sensors. If a single reversal from the entire nested loop was successful (improves the overall cost) we iterate through the route again (because the route has changed). If no successful path reversals are made throughout an entire nested loop the algorithm is terminated.

**Calculating distances between sensors**If the path between two sensors is obstructed by a no-fly zone then we try calculate the actual distance it would take our drone to go around these obstructions.

# **Testing**

**Table

Description automatically generated**

Upon testing I came to realize that the same sensors were used every 92 days. This is evident by the repetitive pattern shown in the plot below which shows the performance of all the different algorithms for all the days.

**Chart

Description automatically generated**

Therefore, for the sake of clarity, when plotting scatter plots to show the performance for each individual day I will only these unique 92 days.

Timeline

Description automatically generated

**Chart, line chart

Description automatically generated**

**Chart

Description automatically generated**

**Optimal algorithm for both performance and time**

**Optimal algorithm to use for scalability (noting the number of sensors that would be required for all of Edinburgh)**

**Calculating moves**

**Calculating a single move**

**Avoiding no fly zones**

**Map outputs**

Map

Description automatically generated**Best # moves map: 49**We can attribute this extremely low move count to be due to the lack of complexity in this example.

This is evident by the particularly close proximity of all the sensors and lack of buildings in the way.

**Worst # moves map: 109**Map

Description automatically generatedThe high move count in this map was predominantly due to two sections in our route. We can visually identify these as being the path inefficiently wrapping around both the Informatics Forum and Appleton Tower, and the squiggle in the bottom left next to the library.

Firstly, the path wrapping around the library is due to the way our *findNextMove()* method works. This method retrieves the 2 next **valid** angles closest to the angle from the current point to the destination sensor (counter clockwise and clockwise from the original angle). These angles are then used to calculate the coordinates of the 2 possible next points. We then choose the point that is closest to the sensor (Euclidean distance) for the next move. It is evident to see that although this works very well most of the time, when faced with a no-fly zone it is possible the distance measure may not choose the optimal angle/point and may end up going around the building the longer way.

Secondly, the squiggle in the bottom left is due to how close one of the sensors is to the boundaries of the confinement area and the library (no-fly zone). This ultimately made the drone make moves in lots of irregular directions until it was located close enough to the sensor.

Compare the execution time and performance of all our different models over all the different dates

I noticed that the sensor locations were repeated every 92 days the only difference being readings. This is evident as shown by the graph below which shows the performance for all days in 2020 and 2021.

Therefore, for performance analysis I decided for the sake of readability and redundancy to just include these 92 days to compare different algorithm performances.