Informatics Large Practical (ILP) Report

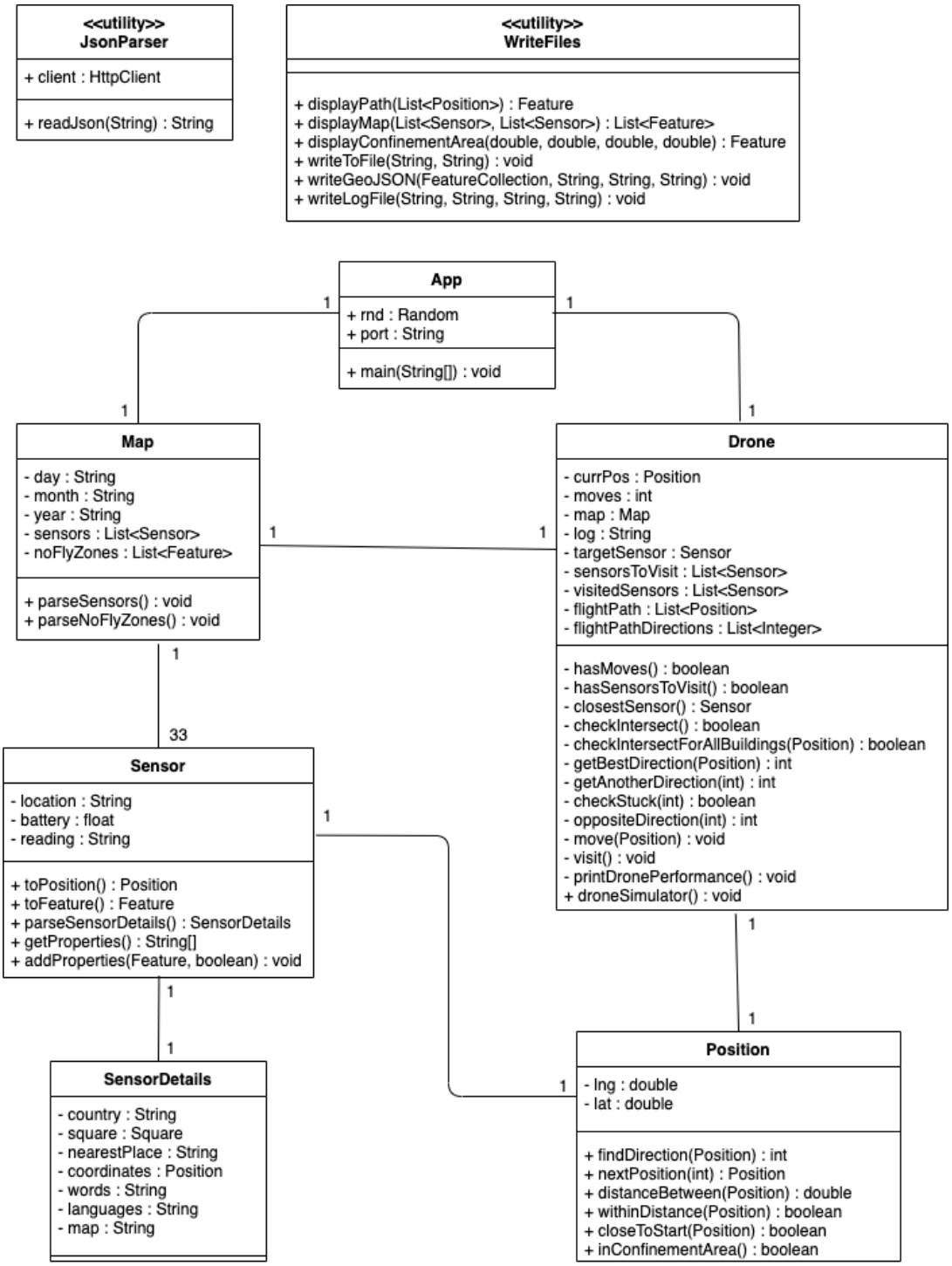
s1869672

CONTENTS

1. Software Architecture Description

This section will explore the overall software architecture of my application, as well as the reasoning behind the decision I made in identifying the Java classes that are suitable for my application. The UML diagram below will show the hierarchical relationships between the classes.

1.1. UML Diagram



1.2. Reasoning in choosing the classes

The **Sensor** class is used to represent an air quality sensor, hence includes the What3Words address, battery level and the reading as private attributes. It includes the method to parse the JSON file from the words folder, which give the What3Words information corresponding to the What3Words address of the sensor. Although the only relevant key in the JSON data is the coordinates, I decided to represent this information as a **SensorDetails** object which contains all the keys as attributes in case it would be useful in the future.

The **Map** class is used to represent the current map for the drone, hence includes the day, month, year, sensors to be visited and the no-fly zones as attributes. One of the main responsibilities of this class is to parse the JSON file in the maps folder. With the given date of the map, we will obtain the corresponding sensors to be visited by the drone and store it as a list of Sensor objects. Another responsibility is to parse the GeoJSON file in the buildings folder to represent the no-fly zones within the map.

The three folders (maps, words and buildings) are stored on a web server on a particular port. Therefore, I have decided to create a utility class **JsonParser** that has a public static final attribute named client, which is an HttpClient with default settings. It contains a method to read the JSON/GeoJSON data from the given URL and returns it as a String object to be used by the **Sensor** and **Map** class.

The **Position** class is used to provide a way to hold the positions of a drone and sensors on the map through their respective longitudes and latitudes. It includes methods such as whether a drone is within the confinement area, whether a drone is within distance of the sensors to download readings, and so on. These methods are mainly utilized across the **Drone** class.

The **Drone** class represents a drone and is responsible of its flight path in visiting the sensors and returning close to its initial position. Therefore, it includes a map, its current position, its target sensor, the number of moves it has left, its flight path, its flight path directions, the list of sensors to be visited, the list of sensors that has been visited as attributes and a log to record the drone's movements. It contains the method droneSimulator() which implements the drone control algorithm and where all the attributes (except map) will be modified.

The **App** class serves as the main entry point of the application as it contains the main() method. It has the Random object attribute set by the input seed and String object attribute set by the input port. These are set public static as they are accessed by other classes. Running this class will parse input arguments, instantiate a **Map** and **Drone** objects, run the simulator and create output files. The creation of output files is done through the public static methods in the utility class **WriteFiles**. This class is responsible of creating the output files, including the creation of Feature objects for the .geojson output files.

2. Class Documentation

This section will have a more detailed explanation on each of the classes that I have chosen above, including descriptions on the attributes and methods and their visibility. Attributes that are private have their respective accessor methods if necessary for this coursework.

2.1. Position.java

Position object has two private attributes:

- The **lng** attribute which is of type double that represents the longitude of an object.
- The **lat** attribute which is of type double that represents the latitude of an object.

Both attributes are final as they should be immutable, that is they are not supposed to change throughout the course of running the application.

There are several public methods defined in this class:

- The **getDirection(Position *destination*)** method has a Position object as the parameter and calculates the direction to move to the ***destination***. This is done by calculating the arc tangent of the change in latitude over the change in longitude using the inbuilt method Math.atan2, which returns a value between -π and π. Since negative values are not allowed according to the coursework specification, 2π is added to the value returned to obtain an equivalent positive direction. This is then converted to the nearest ten degrees and if it is equal to 360°, an equivalent direction of 0° is assigned.
- The **nextPosition(int *direction*)** method takes in an integer ***direction*** and calculates a new position after moving the drone from its current position to the specified ***direction***. This is done by adding the current longitude and latitude values with the change in longitude and change in latitude respectively. The new values are then used to create a new Position object and is returned to indicate the next position of the drone.
- The **distanceBetween(Position *pos*)** method takes in a Position object and returns the Euclidean distance between the current position and that object. The Euclidean distance formula to compute the distance between points *p* and *q* is given by:

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$$

- The **withinDistance(Position *pos*)** method takes in a Position object and returns a Boolean to check if Euclidean distance between the input position and the position object the method is called on is less than 0.0002 degrees. This method is used to check whether a drone is close to a sensor (in order to download readings).
- The **closeToStart(Position *pos*)** method takes in a Position object and returns a boolean to check if the current position (i.e. of a drone) is "close to" its starting position. According to the coursework specification, "close to" means the distance between the two positions should be less than 0.0003 degrees.

- The **inConfinementArea()** method returns a boolean to check whether or not a certain position is within the drone confinement area, that is between (55.946233, -3.184319) and (55.942617, -3.192473).

2.2. Map.java

This class handles reading the JSON data consisting of the list of sensors to be visited on a particular day.

A Map object has the following private attributes:

- The **day** attribute which is of type String.
- The **month** attribute which is of type String.
- The **year** attribute which is of type String.

  These three attributes represent the date in which will be used to generate the sensors to be visited by the drone.

- The **sensors** attribute is a List of Sensor objects, which represents a list of sensors to be visited on the particular day, month and year.
- The **noFlyZones** attribute is a List of Feature objects, which represents the buildings that form the no-fly zones.

This class has the following private methods as they are only used within this class:

- The **parseSensors()** method is used to initialize the **sensors** attribute. It parses the JSON data stored inside the files in the maps folder.
- The **parseNoFlyZones()** method is used to initialize the **noFlyZones** attribute. It parses the GeoJSON data stored inside the files in the buildings folder.

  Both of these methods utilizes the public static method **readJson(String)** in the **JsonParser** class.

2.3. Sensor.java

A Sensor object has three private attributes:

- The **location** attribute is a String object that represents the What3Words address of a sensor. This attribute is final because a sensor's location should be changed, i.e. a sensor does not move around.
- The **battery** attribute is a float that represents the percentage battery charge of a sensor.
- The **reading** attribute is a String object that represents the reading of a sensor. It is not of type double/float since a reading can also be "null" or "NaN".

In addition, this class has the following public methods:

- The **parseSensorDetails()** method is used to parse the JSON file containing the What3Words information corresponding to the What3Words address (location) of the Sensor object that the method is called on. It returns a SensorDetails object.

- The **toPosition()** method is used to obtain the position of the Sensor object that the method is called on. This is done by splitting the What3Words address (location) of the sensor and use getSensorDetails() to obtain the coordinates. It returns a Position object.
- The **toFeature()** method is used to convert the Sensor object that the method is called on into a Feature object. This is done by creating a Point object from the position of the sensor (obtained by toPosition() method), and then converting it into a Feature. This method is used to display the sensors in the output GeoJSON map.
- The **getProperties()** method returns a length 2 array of String objects. The first element represents the rgb-string and the second element represents the marker-symbol. These two properties depend on the reading and the battery level of the Sensor object that the method is called on.
- The **addProperties(Feature *feature*, boolean *hasVisited*)** method adds the properties to *feature* depending whether or not the sensor has been visited by the drone. Some of the properties are specified in the coursework specification and some are obtained through the method getProperties().

2.4. SensorDetails.java

This class is used to store the information in a JSON file from the words folder. It stores the information corresponding to a What3Words address and has (private) attributes that are the same as the keys in the file. The only relevant attribute for this coursework is **coordinates** which is a Position object, representing the position of a sensor.

2.5. WriteFiles.java

This is a utility class to handle the writing of output files. This includes methods to help construct the output .geojson and .txt files. It has the following public static methods:

- The **displayPath(List<Position> *path*)** method takes in a list of Position objects which represents the flightpath of the drone. This is used to construct a LineString object which is then converted to a Feature object and returned by the method.
- The **displayMap(List<Sensor> *visitedSensors*, List<Sensor> *unvisitedSensors*)** method takes in a list of Sensor objects that has been visited and those that are failed to be visited by the drone. Each sensor is converted into a Feature object and the addProperties() method is then invoked to assign appropriate properties to the sensors. The output will be a list of Feature objects.
- The **displayConfinementArea(double *lng1*, double *lng2*, double *lat1*, double *lat2*)** method takes in four doubles which represent the borders of the confinement area. It ouputs the confinement area as a Feature object used for debugging.
- The **writeToFile(String *fname*, String *str*)** method creates/opens the file specified by the input file name, *fname*, and writes the *str* parameter into the file.

- The **writeGeoJSON(FeatureCollection *fc*, String *day*, String *month*, String *year*)** takes in a FeatureCollection object and converts it to a GeoJSON string. The input date will be used to create the filename. These will be the parameters to the writeToFile method and the output .geojson file is created.
- The **writeLogFile(String *log*, String *day*, String *month*, String *year*)** creates the file name with the input date. Together with the input String log, these will be passed to the writeToFile method and the output .txt file is created.

2.6. JsonParser.java

This utility class is responsible of reading the JSON/GeoJSON data from a specified URL. It contains an attribute **client**, which is a HttpClient object shared between all HttpRequests. This attribute is public as it will be used in the Sensor and Map class. It is static final because we only need one HTTP client and it will not be updated.

It also contains the method **readJson(String *urlString*)**, which takes in a URL as a String object, reads the JSON/GeoJSON data from that URL, and returns it as a String object (which will be deserialized in the Map and Sensor class).

2.7. App.java

This class contains the main() method, which serves as the entry point to the whole application. It has the following public static attributes:

- The **rnd** attribute is a Random object which depends on the input seed argument.
- The **port** attribute is a String object that stores the port number specified by one of the command line input arguments.

This class has one public static method:

- The **main(String[] *args*)** method reads the input arguments and assigns the last input argument to **port**. Next, a random seed is generated which is assigned to **rnd**. A Map and Drone object is then instantiated according to the input arguments and the **droneSimulator()** method is called. Finally, the output files are then created with the help of some of the methods in **WriteFiles** class.

2.8 Drone.java

This class is used to implement the drone control algorithm. It contains the following attributes:

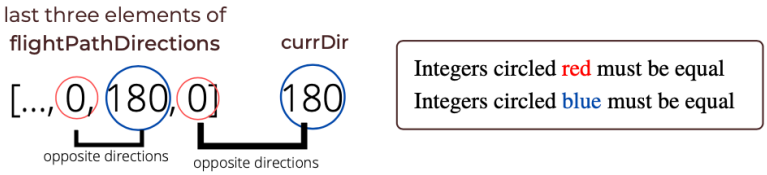
- The **currPos** attribute is a Position object which represents the drone's current position.
- The **moves** attribute is an integer that represents the amount of moves that the drone has left.

- The **map** attribute is a Map object that represents the map which includes the sensors that the drone will be required to collect readings from.
- The **log** attribute is a String object that stores the string to be written to the output *.txt file.
- The **targetSensor** attribute is a Sensor object that represents the sensor that the drone is currently trying to visit.
- The **sensorsToVisit** attribute is a list of Sensor objects that represents the sensors that the drone has to collect readings from.
- The **visitedSensors** attribute is a list of Sensor objects that represents the sensors that has been visited by the drone.
- The **flightpath** attribute is a list of Position objects that represents the positions that the drone has moved to.
- The **flightPathDirections** attribute is a list of integers that represents the directions that the drone has moved to.

This class has the following private methods which serve as helper methods for the drone control algorithm:

- The **hasMoves()** method checks if the drone has moves left, that is moves > 0.
- The **closestSensor()** method the closest Sensor object that has not been visited to the drone. The distances of each sensor in sensorsToVisit from the drone and returns the sensor that has the smallest distance.
- The **checkIntersect(List<Point> *building*, Position *nextPos*)** method checks whether moving from the currPos to nextPos will bring the drone to the specified building. This method uses the java.awt.geom package to create Line2D objects that represent the sides of the building and a Line2D object that represents moving from currPos to nextPos. If the latter intersects with any of the former, the method returns true. Otherwise, it returns false.
- The **hasSensorsToVisit()** method checks if the drone still has sensors to be visited.
- The **move(Position *nextPos*)** method assigns nextPos to currPos and decrements the moves attribute, indicating that the drone has moved.
- The **visit()** method removes the targetSensor from sensorsToVisit list and adds it to visitedSensors list, indicating that the sensor has been visited by the drone.
- The **checkIntersectForAllBuildings(Position *nextPos*)** method checks whether moving the currPos to nextPos will bring the drone to any of the buildings that make up the entire no-fly zones.
- The **getBestDirection(Position *targetPos*)** method is used to get the direction that brings the drone towards the *targetPos*. This method also handles cases where the movement of drone may cause it to fly in the no-fly zones or outside the confinement area. If this is the case, we loop through all possible directions and get the direction that does not violate the rules and brings the drone closest to the *targetPos*.

- The **getAnotherDirection(int *direction*)** method is used to get another direction in case the drone gets stuck. We first try to get the opposite direction to the input direction, but if it violates the rules, we generate a random valid direction such that it is not equal to the input direction.
- The **checkStuck(int *currDir*)** method checks whether the drone is stuck by getting the last three directions that the drone has moved towards (from flightPathDirections) and check whether moving towards the direction *currDir* satisfies the following constraints:



  If it fulfills all the constraints, the drone is stuck. Otherwise, the drone is not stuck.
- The **oppositeDirection(int *direction*)** method returns the opposite direction of the given input direction. This is used in the checkStuck method above.
- The **printDronePerformance()** method is used to print the performance of the drone, such as the number of moves taken. It is used for debugging purposes.

It also has one public method which is the drone control algorithm that is going to be called in the main() method in App class:

- The **droneSimulator()** method implements the drone control algorithm. The idea is that the drone will always try to move towards the sensor closest to it (targetSensor): The drone will move as long as it **hasMoves()**. For each move, it is checked whether the drone **hasSensorToVisit()** or not. If it does, the method **getBestDirection(Position *targetPos*)** is invoked (where targetPos is the position of the targetSensor), the next position is computed from the returned direction and we then invoke **checkStuck(int *direction*)** to check if the drone is stuck. Note that a drone can only be stuck if it has moved 3 or more times. If it is stuck, the method **getAnotherDirection(int *direction*)** is invoked to obtain another valid direction to free the drone from being stuck. We then update the output **log** string and invoke **move(Position *nextPos*)**. At the end of a move, the drone's **flightPath** and **flightPathDirections** are updated.
  After a move, we will check if the **targetSensor** is within distance of the drone for it to be visited. If it is, we invoke **visit()**. If it isn't, we check if there is another closer sensor that the **targetSensor** and try to visit it if it is within distance. We then update the drone's sensor visit to the output **log** string accordingly.
  If all sensors has been visited, drone will move back to its starting position with similar strategy to moving towards a sensor.

3. Drone Control Algorithm

The drone's objective is to visit all 33 sensors and return to its initial position, all while avoiding the no-fly zones. An optimal solution would be one that allows the drone to complete its objective in the least number of moves.

3.1. Pseudocode

The pseudocode of my **droneSimulator()** method is as follows:

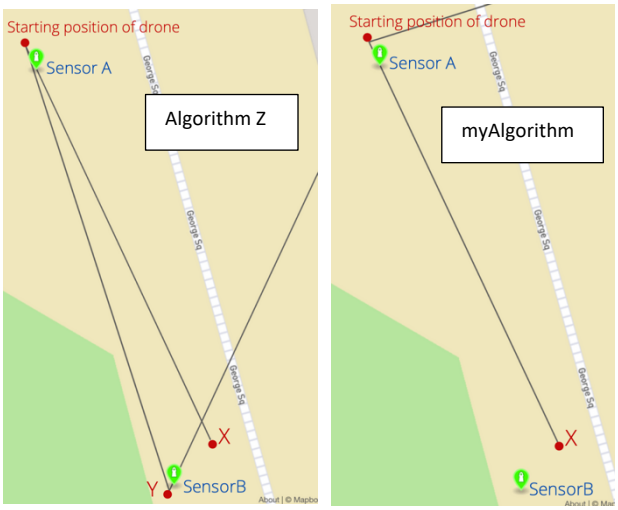
1. Add starting position, stored in *currPos*, of the drone to *flightPath*
2. Get the position of the *targetSensor*
3. **REPEAT**
4. | **IF** drone still has sensors to visit:
5. | | **IF** drone visits a sensor this move:
6. | | Find the closest sensor to the drone and its position
7. | | **ENDIF**
8. | | Invoke *getBestDirection()* and attempt to move to direction returned
9. | | **IF** drone is stuck:
10. | | Invoke *getAnotherDirection()*
11. | | **ENDIF**
12. | | Update string to be written to the ouput .txt file
13. | | Move to the next position
14. | | Record the direction & path to *flightPathDirections* and *flightPath* respectively
15. | | **IF** *targetSensor* is within distance to the drone:
16. | | Visit sensor
17. | | **ELSE**
18. | | Try to see if there is a closer *targetSensor* and visit if within distance
19. | | **ENDIF**
20. | **ELSE IF** all sensors have been visited:
21. | | Move towards starting position using *getBestDirection()*
22. | | **IF** drone is stuck:
23. | | Invoke *getAnotherDirection()*
24. | | **ENDIF**
25. | | Update string to be written to the output .txt file
26. | | Move to the next position
27. | | Record the direction & path to *flightPathDirections* and *flightPath* respectively
28. | | **IF** drone is close to starting position:
29. | | **BREAK**
30. | | **ENDIF**
31. | **ENDIF**
32. **UNTIL** drone has no more moves left

3.2. Finding the target sensor

The core idea of the algorithm is to try to visit the sensor that is closest to the drone. This will be the ***targetSensor*** of the drone. We use the method **closestSensor()** to obtain the closest sensor (that has yet to be visited) to the drone. We update the ***targetSensor*** (i). when we have visited it, and (ii). during the case where we encounter a closer sensor when we are trying to move towards it. I will show why the latter is important with two algorithms running on a map for the date 02/02/2020.

Algorithm Z: Update ***targetSensor*** only with condition (i).

myAlgorithm: Update ***targetSensor*** with condition (i) and (ii).



For **myAlgorithm**, notice that the drone is able to visit sensors A and B in 2 moves (Move from start to X, then visit B, move back to start, then visit A). Whereas for **Algorithm Z**, the drone takes 3 moves to visit sensors A and B (Move from start to X, then back to start, visit A, move to Y, visit B).

This can also be useful when a drone is moving around a no-fly zone and encounters another sensor along the way.

3.3. Handling no-fly zones and confinement area

This is done in the **getBestDirection(Position *targetPos*)** method. When moving towards the direction of the ***targetPos*** brings the drone to the no-fly zones or outside the confinement area, we will find another valid direction that does not violate these constraints. We will loop through all possible directions 0, 10, 20, …, 350 and get the direction which brings the drone closest to the ***targetPos***, while satisfying all the constraints. This also applies when the drone is moving back to its initial position.

For each move, we check whether a drone crosses the buildings in the no-fly zones using the methods **checkIntersectForAllBuildings()**. For each building, we perform **checkIntersect()** to check whether the movement intersects with any of the sides of the building. We use the java.awt.geom package to create Line2D objects that represent the sides of the building and a Line2D object that represents moving from currPos to nextPos. We use the method from the Line2D class, **intersectsLine(Line2D)**, to check if the latter intersects with any of the former. This is carried out for all buildings that make up the no-fly zones. To check whether a position is outside the drone confinement area, we use the **inConfinementArea()** method in the Position class.

3.4. Handling drones that are stuck

This is dealt with the method **checkStuck(int *currDir*)**. This method checks the last three ***flightPathDirections*** of the drone together with the direction, *currDir*, that the drone is about to move to. A drone is stuck if it does two moves of opposite directions, followed by two same moves. For example, if the last three ***flightPathDirections*** are [0, 180, 0] and *currDir* is 180, then the drone is stuck.

If the drone is stuck, the method **getAnotherDirection(int direction)** is invoked, where we first try to get the opposite direction to the input direction in an attempt to glide through the side of the building that the drone intersects with. If the opposite direction violates no-fly zone/confinement area constraints, then we generate a random direction such that it is not equal to the input direction.

3.5. Drone's life-cycle

The algorithm ensures that the drone makes a move before visiting a sensor (if in range). After getting the direction that does not cause the drone to be stuck, or bring it into no-fly zones or outside the confinement area, the **move()** method is invoked. Next, the drone will check if the ***targetSensor*** is within range of the drone. If it is, the **visit()** method is invoked and the drone finds a new target sensor. If it is not within range, the drone will try to find a closer sensor than the ***targetSensor*** and visits it if in range (as explained in 3.2).

3.6. Flying back to starting position

After all the sensors has been visited, the drone will begin to move back towards its starting position. Again, we use **getBestDirection()** method to get the direction and if the drone gets stuck when moving towards this direction, we use **getAnotherDirection()** method. The algorithm terminates once the drone's currPos is close to the starting position (use **closeToStart()** method in Position class), or when it runs out of moves.

3.7. Sample Output

The following figures show the flight path of the drone on two separate maps.

Figure 2: Output map for the date 02/02/2020 starting from (-3.1878, 55.9444).

Figure 1: A sample output map for the date 11/11/2020 starting from (-3.188396, 55.944425).

3.8. Testing

For the testing of my drone, I create a test in AppTest.java to run the drone on over 700 different maps given from 01/01/2020 to 31/12/2021. In the test method, I added assertTrue statements to check if the drone visits all the sensors required to be visited and check if the drone ends up close to its initial position. I would check the maps where it has poor performance, such as when the drone does not visit all the sensors and inspect the reason behind this.

I found that analyzing the flight path visually to be the most helpful as I was able to see how the drone gets stuck. It also led to my realization that checking for a closer sensor when moving towards a targetSensor can lead to a better performance. (See 3.2)