ILP Coursework 2 Report 2020

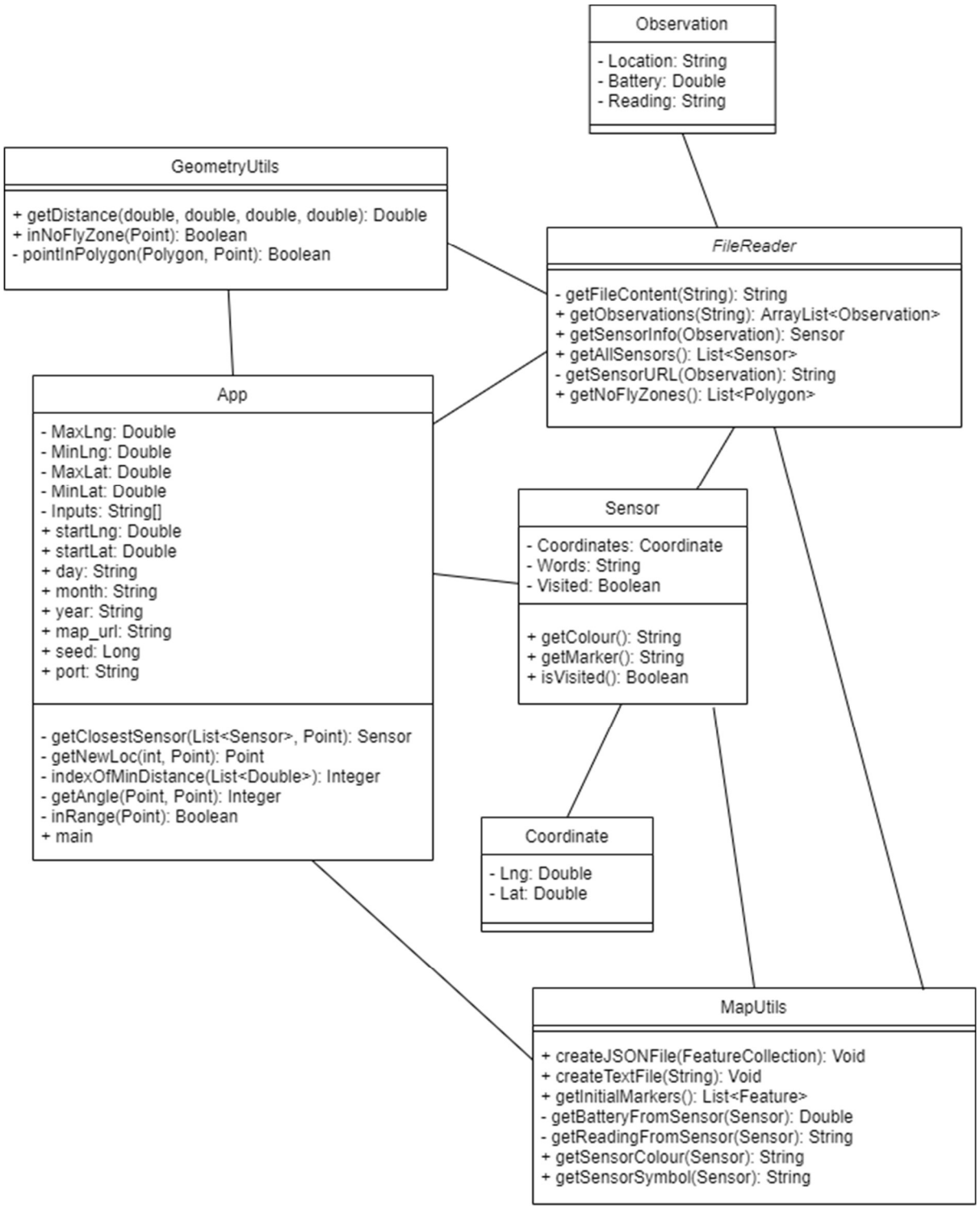
By Amy Rafferty s1817812

CONTENTS

Page Number	Topic
1	Title and Contents Page
2	Software Architecture Description – Class Diagram
3	Software Architecture Description – Justification Class Documentation – Sensor
4	Class Documentation – Sensor Cont., Coordinates, Observation, FileReader
5	Class Documentation – FileReader Cont., MapUtils
6	Class Documentation – MapUtils Cont., GeometryUtils
7	Class Documentation – App
8	Class Documentation – App cont. Drone Control Algorithm
9	Drone Control Algorithm Cont.
10	Drone Control Algorithm Cont.
11	GEOJSON Example 1
12	GEOJSON Example 2, Sources

SOFTWARE ARCHITECTURE DESCRIPTION.

The following Class Diagram details the classes I have chosen to use for this project.



The Observation and Sensor classes are simply for representing the data that is to be read in from the JSON files on the Web Server.

The Observation class for data from an *air-quality-data.json* file for a particular date, and representing each piece of data as an Observation, containing a location, battery level and pollution reading.

The Sensor class does the same for the corresponding *details.json* file for a particular Observation, where the location string from an Observation links to a specific Sensor. Each sensor has a set of coordinates specifying its longitude and latitude, as well as a boolean value representing whether the sensor has been visited or not on a particular day. Within this class is the Coordinate class, which groups the longitude and latitude together for legibility purposes.

At this point, once the Objects were taken care of, I decided to split up my code according to functionality.

The FileReader class takes care of converting the data from the JSON files into a String, then into a list of Observation instances, from each of which a Sensor instance is created. It also contains a method which returns a list of 33 Sensor instances for a given day. This class also takes care of finding the URL for the sensor information given the location string from an Observation, which allows my algorithm to pull information from the correct file on the webserver. Finally, it also contains a method to return the no fly zones as a list of Polygons.

The MapUtils class contains methods that help with creating the GEOJSON graph, as well as the flightpath text file. It contains a method for both, which take a FeatureCollection for the GEOJSON file, or a String for the text file, and write the information to the appropriate file type. It also contains a method that creates grey markers for each of the sensors for a given day – these markers are later removed and replaced with coloured markers as each sensor is read from. The MapUtils class also contains methods that return the battery and pollution reading for a sensor, and methods that return the required colour and symbol for a marker given these two values.

I also felt the need for a GeometryUtils class, which takes care of some of the mathematical methods required for my algorithm. It contains a method for finding the Euclidean distance between two points, and for checking whether a point is within a polygon. It also contains a method that returns whether a point is within any of the no fly zones, using the latter method.

Everything else is within the App class – these are the methods that are integral to the algorithm. This class contains a method for finding the closest unvisited sensor to a particular point, finding the new location of the drone given its current position and angle of travel, and a method for finding the best angle for the drone to travel in. It also contains a method that checks whether a point is within the confinement area for the drone.

CLASS DOCUMENTATION

Sensor Class

Every instance of this class is a representation of the data in one *details.json* file – that is, the relevant information about a particular sensor, identifiable by its unique location. The class comes with getter methods for retrieving the W3W words , the longitude, and the latitude of the sensor.

The class also contains getter methods for retrieving the colour and symbol of the marker corresponding to that sensor. These methods use methods from MapUtils, which will be explained in the MapUtils subsection, but exist in the Sensor class for readability purposes.

Finally, each sensor has a boolean attribute that tells us whether the drone has visited it on the current day. The class comes with a getter method, isVisited(), and a setter method for this attribute, to be used in the main algorithm.

Coordinates Class

This class simply exists so that the longitude and latitude of the sensor can be accessed in a readable way, and is not used by the wider algorithm.

Observation Class

Every instance of this class is a representation of one of 33 pieces of data in an *air-quality-data.json* file. Each of these data pieces contains a location, a battery level, and a reading. The class contains getter methods for all three of these attributes.

FileReader Class

This class contains methods for reading data from JSON files into Sensor and Observation instances, and retrieving the No Fly Zones.

- **private static String getFileContent(String url)**
  This method takes the URL of a JSON file on the Web Server, and converts its contents to a String. This is the method that will throw an error if there is an issue with the Web Server or the command line inputs provided by the user.
  - Error Message “Check Inputs!” – A command line input is invalid, check that 1-digit days and months are prefixed by a 0.
  - Error Message “WebServer Down!” – The Web Server is not running.
  - Error Message “Check URL!” – Issue with the file URL itself, could have an invalid port number in the command line.

- **public static ArrayList<Observation> getObservations(String url)**
  This method gets the String content of the *air-quality-data.json* file as specified by the input date. It then converts this String into an ArrayList of 33 Observation instances for that day.

- **private static String getSensorURL(Observation observation)**
  This method finds the URL of the *details.json* file corresponding to a particular Observation, using its W3W words string. This URL points to the information for the Sensor at that location.

- **public static Sensor getSensorInfo(Observation observation)**
  This method first converts the information from the *details.json* file found by the previous method getSensorURL into a String. It then creates a Sensor instance using this information.

- **public static List<String> getAllSensors()**
  This method first receives the list of 33 Observations for the input date using the getObservations method above, then creates a Sensor for each of these Observations, and returns the resulting list of 33 Sensor instances for that day.

- **public static List<Polygon> getNoFlyZones()**
  This method reads the *no-fly-zones.geojson* file from the Web Server, and converts its contents into a list of Polygons, each representing a section of the map that is out of bounds for the drone.

MapUtils Class

This class contains the functions used to create both the GEOJSON map and the Flight Path text file.

- **public static void createJSONFile(FeatureCollection fc)**
  This method takes the FeatureCollection generated by the algorithm in App, which will be explained in the App subsection. It writes this data to a GEOJSON file named "*readings-day-month-year.geojson*" where day, month and year are taken from the command line inputs.

- **public static void createTextFile(String flightpath)**
  This method takes the flightpath String generated by the algorithm in App, which will be explained in the App subsection. It writes the data to a .txt file named "*flightpath-day-month-year.txt*" where day, month and year are taken from the command line inputs.

- **public static List<Feature> getInitialMarkers()**
  This method first finds the 33 Observation instances for the input date, then for each corresponding Sensor object, converts its position to a Point object. It then converts these Points to GEOJSON Features, and adds the default properties – a rgb-string and marker-color of grey (#aaaaaa). The method returns the list of these features, which when rendered in GEOJSON, produces 33 grey markers at the positions of the Sensors for the day. These markers are later replaced with coloured markers in the algorithm in App as the Sensors are read from, but any Sensors that are missed will remain as created by this method.

- **private static double getBatteryFromSensor(Sensor sensor)**
  This method takes a Sensor, and finds its corresponding Observation by finding the Observation with the same W3W words as its location, out of all the Observations for the input day. It then returns the battery level of this Observation.

- **private static String getReadingFromSensor(Sensor sensor)**
  This method takes a Sensor, and finds its corresponding Observation by finding the Observation with the same W3W words as its location, out of all the Observations for the input day. It then returns the pollution level reading from this Observation.

- **public static String getSensorColour(Sensor sensor)**
  This method takes a Sensor, and using the battery and reading values from the above two methods, getBatteryFromSensor and getReadingFromSensor, and determines what colour the marker for that Sensor should be. For reference:
  - If the battery level is under 10%, the marker should be Black (#000000)
  - Otherwise, the colour depends solely on the pollution level reading:
    - 0 –> 31 Green (#00ff00)
    - 32 -> 63 Medium Green (#40ff00)
    - 64 -> 95 Light Green (#80ff00)
    - 96 -> 127 Lime Green (#c0ff00)
    - 128 -> 159 Gold (#ffc000)
    - 160 -> 191 Orange (#ff8000)
    - 192 -> 223 Red/Orange (#ff4000)
    - 224 -> 255 Red (#ff0000)

  This method also takes care of Out Of Bounds errors, which ensure that the battery level is a value 0 -> 100, and providing that the battery level is over 10%, the pollution reading is a value 0 -> 255. Note that for Observations with battery levels under 10%, the pollution reading can be NaN, but this is not an issue because of the way this method is constructed.

- **public static String getSensorSymbol(Sensor sensor)**
  This method takes a Sensor, and using the battery and reading levels from the getter methods above, determines what symbol should be shown on the GEOJSON marker that corresponds to it.
  - If the battery level is below 10%, the marker should be a “cross”
  - Otherwise, it depends solely on the pollution level reading:
    - 0 -> 127 “lighthouse”
    - 128 -> 255 “danger”

  The symbol for an unvisited sensor is None, shown in this method as an empty String. Since invalid battery and reading values are taken care of in getSensorColour above, and this method is always called after it, there is no need for error catching here.

GeometryUtils Class

This class contains solely geometric methods that will be used when creating the algorithm in App.

- **public static double getDistance(double lng1, double lat1, double lng2, double lat2)**
  This method finds the Euclidean distance between two sets of (lat, lng) coordinates.

- **private static boolean pointInPolygon(Polygon polygon, Point point)**
  This method finds out whether a Point is within a Polygon. For this, an algorithm I found online was used (1). First, the Polygon is converted to a 2D double array where [i][0] is the longitude of point i in the Polygon, and [i][1] is the latitude. The Point is converted to a 1D double array {longitude, latitude}. The underlying theory here is that a Point is within a Polygon only if a line from the Point to infinity crosses the Polygon an odd number of times. This method is implemented here, and returns a boolean value that tells us the outcome.

- **public static boolean inNoFlyZone(Point point)**
  This method takes a Point, and determines whether it is within any of the No-Fly Zones as found by the getNoFlyZones method in the FileReader class. To do this, it simply implements the above method, pointInPolygon, for every No-Fly Zone Polygon. The outcome is then returned as a boolean.

App Class

This is the main class that contains the core algorithm for the drone, and its supporting internal functions. It includes attributes representing the command line inputs (day, month, year, start latitude, start longitude, random seed, port number), as well as the values bounding the confinement area for the drone. These are represented as final values as they should never change unless more sensors are added outside of the current confinement area or the drone is required elsewhere. This area is bounded by the minimum and maximum longitudes and latitudes, as specified in the coursework document.

- **private static Sensor getClosestSensor(List<Sensor> unvisited_sensors, Point point)**
  This method takes a list of Sensors that have not yet been visited on the input day, and the Point representing the current position of the drone. It finds the closest unvisited Sensor to the drone by first finding the distance from the drone to every Sensor in the list, using getDistance from the GeometryUtils class, and then finding the minimum distance.

- **private static Point getNewLoc(int angle, Point currPosition)**
  This method takes the current position of the drone, and the angle that it is going to travel in, and returns its new position after the movement has occurred. The angle values fed into this method are calculated by getAngles, which will be explained later. The new position of the drone is calculated using trigonometry, with the distance travelled each turn being 0.0003 degrees.
  The mathematics behind this method is summarised as follows, where h = 0.0003, and a is the angle travelled in:
  - a = 0 (East) – latitude unchanged, longitude increases by h
  - a = 10 -> 80 (North East) – latitude increases by h*cos(a), longitude increases by h*sin(a)
  - a = 90 (North) – latitude increases by h, longitude unchanged
  - a = 100 -> 170 (North West) – latitude increases by h*cos(a), longitude decreases by h*sin(a)
  - a = 180 (West) – latitude unchanged, longitude decreases by h
  - a = 190 -> 260 (South West) – latitude decreases by h*cos(a), longitude decreases by h*sin(a)
  - a = 270 (South) – latitude decreases by h, longitude unchanged
  - a = 280 -> 350 (South East) – latitude decreases by h*sin(a), longitude increases by h*cos(a)

- **private static int indexOfMinDistance(List<Double> distances)**
  This method takes a list of doubles and find the minimum double in the list. This was separated into its own method to avoid repeatability, as in the next method, getAngle, this process is repeated three times.

- **private static int getAngle(Point currPosition, Point target)**
  This method is fundamental to the algorithm, and provides the angle values for getNewLoc above. It takes two Points – the current position of the drone, and where the drone is trying to go. The target Point will either be a Sensor (if there are Sensors left to read from), or the start location of the drone (if there's not), but the way that this method is structured means that it works for both scenarios.
  First, for every possible angle (0, 10, …, 340, 350) the hypothetical new position of the drone Is calculated, as well as the new distance from the drone to its target. Then, the minimum distance to the target is found using the above helper method. The angle that corresponds to the minimum distance to the target is therefore the best angle for the drone to travel in. However, we also need to consider the No-Fly Zones and the confinement area of the drone. So, if the new position generated by getNewLoc using the best angle found here is either in a No-Fly Zone, or not in the confinement area, this move is invalid. In this case, the drone will move in the next best direction, check whether the new move is valid, and repeat until a valid move is found. Then, the optimum valid angle is returned.

- **private static boolean inRange(Point point)**
  This is a simple method that takes a Point, and checks whether it is within the confinement area for the drone. To do this, it simply compares the latitude and longitude of the Point to the maximum and minimum dimensions specified by the final attributes in the App class.

- **public static void main(String[] args)**
  This is the main algorithm for the drone and will be explained in the next section.

DRONE CONTROL ALGORITHM

In this section, the main method in the App class, which is where the overall algorithm for the drone is located, will be explained in stages.

1. First, the algorithm takes the inputs from the command line, and assigns them to variables (day, month, year, startLat, startLng, seed, port). The URL for the *air-quality-data.json* file is then generated using these inputs.
2. Then we initialise variables. This includes creating a Point representing the start position of the drone from its initial coordinates, and an empty String for the text file contents. An ArrayList of Sensors is created from the getAllSensors method in FileReader, which represents the list of unvisited Sensors and will gradually decrease in size. A list of Points representing the path the drone will take is initialised to contain only the starting Point, and a list of Features containing the grey markers from getInitialMarkers in MapUtils is created. Finally, an empty list of Features representing what will be the updated coloured markers is initialised.
3. For up to 150 turns:
    a. If there are still Sensors to be visited:
        i. The algorithm first finds the closest Sensor to the drone using the getClosestSensor method, and converts its location to a Point.

ii. The distance to between the drone and the Sensor is calculated using getDistance from GeometryUtils, and the optimal angle to travel in to reach this Sensor is calculated by the getAngle method. The algorithm then generates a hypothetical new location using the getNewLoc method with this angle.

iii. If the drone starts going backwards and forwards between two Points, for example if it is stuck behind a No-Fly Zone, we need to change where the drone is trying to go to help it escape:

    1. If there is more than one Sensor left to read, and the drone gets stuck trying to visit its closest Sensor, we simply direct the drone to visit the next closest Sensor. This will change the trajectory of the drone and enable it to escape its loop.

    2. If there is only one Sensor left to read, and the drone can't get to it, we try to move the drone in an arbitrary direction once, and then let it try to reach the Sensor again. Here, the axes (0, 90, 180, 270) are used.

        a. For each of the four axes directions, a hypothetical new location is generated in the same way as previously done. If the new location is within the confinement area, not in a No-Fly Zone, and not equal to the previous position of the drone (indicating a loop), the drone moves.

        b. Moving the drone in this way allows it to unstick itself, and then it can continue progressing towards the desired Sensor on its next turn.

iv. If the new location is within the confinement area of the drone, it is valid, and therefore added to the drone path Point list, and becomes the current location. If not, an **"Error – out of bounds!"** error message is generated, but this should never happen unless a Sensor is outside of the confinement area.

v. If the drone is within 0.0002 degrees of the Sensor closest to it after moving:

    1. The Sensor is converted to a Point, and then to a Feature, so that its formatted marker can be generated. The colour is set to the value from Sensor's getColour method, and the symbol is set to the value from Sensor's getSymbol method.

    2. This formatted Feature is added to the list of visited markers, to be displayed on the GEOJSON graph.

    3. The Sensor is removed from the list of unvisited Sensors, and its corresponding grey initial marker is deleted to ensure that there are always 33 markers in existence.

vi. A line is appended to the String for the flightpath text file. The turn number + 1 (to account for 0 indexing), longitude and latitude at the start of the turn, angle moved in, longitude and latitude at the end of the turn, and the

W3W String of any Sensors visited are formatted in a line separated by commas. Note that if no Sensors are visited, the W3W part will return “null”.

b. If all Sensors have been visited:

   i. The distance from the drone’s current position to its starting position is calculated, along with the optimal angle to move in to get there.

   ii. The hypothetical new location is calculated using getNewLoc, and if it is within the confinement area it is added to the drone path Point list, and the current location is updated. If it is not, an **“Error – out of bounds!”** occurs, but again this should never happen as he target Point is within the confinement area.

   iii. If the current location is within 0.0003 degrees of the starting position, the drone has completed its task and the while loop is terminated. A **“Made it back to start!”** message is also printed to the console for outcome assessment.

4. The algorithm prints out information about its outcome. The date (day/month/year), number of turns used, and number of Sensors missed (length of unvisited Sensors list) are printed to the console, to tell us how well the drone has performed without having to look at the GEOJSON graph.
5. The LineString Feature representing the path of the drone is generated from the list of Points created during the algorithm. This, along with any remaining initial grey markers, and the generated coloured markers, are added to a FeatureCollection. This is then passed into the createJSONFile method from MapUtils, which generates the GEOJSON graph. Note that there will only be grey markers if a Sensor was missed, and there should always be 34 Features (33 Points and the LineString) in the FeatureCollection.
6. The flightpath text file is also generated, using the createTextFile method from MapUtils, with the flightpath String containing a line from each iteration of the algorithm.

Two examples of the GEOJSON graphs generated by this algorithm, for 02/02/2020 and 10/10/2020, are detailed on the next page. The printed console output in Eclipse is also detailed, to show the number of moves needed for the algorithm to complete.

GEOJSON EXAMPLE 1 – 02/02/2020

This map was generated for command line inputs “02 02 2020 55.944425 −3.188396 5678 80” and generates the following console output:

```
Made it back to start!
02/02/2020
Turns: 89
Missed Sensors: 0
```

GEOJSON EXAMPLE 2 – 10/10/2020

This map was generated for command line inputs “10 10 2020 55.944425 −3.188396 5678 80” and generates the following console output:

```
Made it back to start!
10/10/2020
Turns: 116
Missed Sensors: 0
```

SOURCES

(1) https://www.algorithms-and-technologies.com/point_in_polygon/java