Informatics Large Practical Report

Lewis Hamilton (s1807298)

Contents:

-page 2: Software Architecture Description

-page 4: Class Documentation

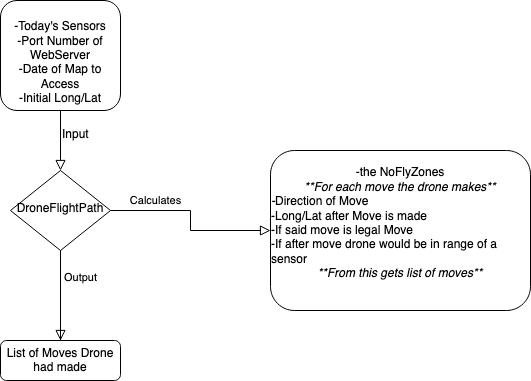
-page 11: Drone Control Algorithm

-page :References/Appendix

*Software Architecture Description*

My implementation contains a number of classes, each of which I see as valuable to the overall project.

*DroneFlightPath/FlightPathOutput*

To start with I’d describe what I believe to be the centre of the implementation in terms of classes. While there are no hierarchical structures in my final implementation these classes serve as the “brain” of it. Rather than have model a class based on the Drone itself, where data, such as its current location, would be constantly changing, I viewed it based on an *input/output* idea. In this case the input is data from the WebServer, such as sensors, noflyzones etc and the output, for each time the application, is run would be a flightpath for the drone. To me this makes it much clearer what is being calculated and where it is being calculated each time the program is run.

*\*\*High level idea of*

*What*

*DroneFlightPath is*

*doing\*\**

One could also view the DroneFlightPath class as an example of the *Factory Pattern*. It is only instantiated once during the program and creates a number of Move Objects that the end user does not know about as it is only sees the output from FlightOutput that the Move objects contribute towards. (Move class is discussed later).

The DroneFlightPath class is where the bulk of the calculations for finding the Flightpath take place, with data from other classes such as TodaysSensors, NoFlyZones being the *input* and the overall *output* of the DroneFlightPath class being an ArrayList of type Move.

The decision to split the part of the program that calculates the output files is for 2 reasons. The first is to reduce the size/scope of DroneFlightPath class to make it easier to read, understand and maintain. The second is so that future users of the program can easily update the formats in how the output is generated. For example, say a user wants to have the flightpath output as a pdf format then it would simply be the case of writing an additional method in the FlightOutput class to parse the data given into said class into this format. This allows for having unique methods for each type of format of output making it easy for end users to pick how they want the data outputted as- they need only create another method in this class.

*SensorsToday/Sensor/Words*

Starting with Sensor, as a class it is simply the data structure which represents the real-life object of the sensors. Unlike say the Drone, to which data surrounding it would be constantly changing, the Sensor’s location, battery reading etc do not change over the course of calculating the flightpath. Hence modelling the Sensor as a class fits the problem.

The purpose of SensorsToday is to create a batch of sensors that represent the sensors the drone is travelling to and their battery/readings for that particular day. Each time this class is called it parses the JSON file from the WebServer for that particular day (SensorsToday’s Input) and gets an ArrayList of the Sensors with their battery/readings for that day (SensorsToday’s Output). This is a cleaner implementation than say creating every possible Sensor on the map each time the application is run. This is because you are cutting down on any unnecessary data such as redundant sensors that won’t be visited that day or calculations within the program to get the sensors to be visited for that day. Essentially SensorsToday *decouples the dependancy Sensor would have on the DroneFlightPath* class by creating an intermediary class that filters the relevant Sensor data to DroneFlightPath. Like DroneFlightPath it can also be considered a part of the *Factory Pattern* with SensorsToday also only being instantiated once during the program to create/parse a number of Sensor objects.

The Words class exists as a class in order to facilitate easy JSON parsing of the different fields of Sensors in the maps JSON Files. The Words class mimics the data structure of the each Sensor in the maps files with the static inner class coordinates acting as the field within a field for the coordinates.

*NoFlyZone*

The NoFlyZone class is an example of allowing me separate out data from the DroneFlightPath class to allow me to reduce the scope of DroneFlightPath. I could have easily just had the NoFlyZones as an ArrayList in DroneFlightPath and have all the GeoJSON parsing in that class but this separation allows for better readability overall. This class now contains all the data and calculations to check if the drone went through a NoFlyZone and DroneFlightPath only needs to instantiate it and call one of its methods throughout the whole application.

*Move*

The purpose of the Move class is to represent as a kind of data structure, each move the Drone makes across the map. It allows me to keep all the data of current/previous Longitudes,Latitudes, if the Drone visited a sensor and if so what sensor, in one place. This is useful as since a List of Moves is essentially the output of DroneFlightPath class, it means the FlightPathOutput class only needs to read in the information from each move to create the final output files.

*Direction*

The Direction class is another example of a class being used as a structured way to store data and also to ensure a clean way of input validation. This class is set up so that it can only store the direction in degrees in increments of ten so whenever a new direction for the Drone to go in is calculated within the DroneFlightPath class, any input validation is moved to this specific class.

*StraightLine*

The purpose of this class is to represent the calculate the straight line equation of the line a drone takes from one location to another. Since only the current and previous longitudes and latitudes are stored/calculated with in the Move and DroneFlightPath classes, this class allows us to calculate any other longitude and latitude on that line. This is useful when calculating whether or not the Drone has gone over the NoFlyZone in one move as this class lets us calculate multiple points along the line of travel and feed that information into the NoFlyZone class to ensure none of these points are in a NoFlyZone

*MapDate*

The function of the MapDate class might seem redundant given their are other Java classes such as Calendar that do a similar purpose. However I believe such classes have too much ceremonial processes in trying to store the date data and in the case of Calendar seems to serve a purpose of getting the current data/time and calculating times. My class simply stores the user inputted data in a structured way with relevant getters to get the date information when it is needed in other classes. I feel this custom Date class makes my overall application more readable

*Class Documentation*

App

Description:

The main method of this class is run when the program is run via terminal. It serves to create the singleton objects for SensorsToday, DroneFlightPath and FlightOutput then call relevant methods to create the two output files

Fields:

—None—

Methods:

-main(String[] args)

Parameters

The main method takes in a string array and will parse the first 7 elements of said array, it corresponds the following arguments

* args[0] -> the day of the date of sensor data you want to access, should always be written as a double digit e.g 07 instead of 7
* args[1] -> the month of the date of sensor data you want to access, should always be written as a double digit e.g 09 instead of 9
* args[2]-> the year of the date of sensor data you want to access, should always be written in full e.g 2021
* args[3]-> initial latitude of drone
* args[4]-> initial longitude of drone
* args[5]-> random seed \*\*unused in this implementation but a arbitrary integer should be added anwyay\*\*
* args[6]-> port number to connect to WebServer, 80 is the usual value given to it

Returns

-Creates two files, a txt file and a GeoJson File of the Drone’s Flightpath, with naming conventions flightpath-DD-MM-YYYY.txt and readings-DD-MM-YYYY.geojson respectively

-Logs if the files have been created or not

DroneFlightPath

Description:

A singleton object used to take in data of what Sensors to visit, Sensor Readings and NoFlyZones and generate a flightpath based on that data, making sure the drone has as few moves as possible

Fields:

All have private access modifiers

* todaysSensors, Type: ArrayList<Sensor>, Purpose: all the sensors the drone should visit
* movesMade, Type: ArrayList<Move>, Purpose: all the moves the drone makes to form the final flightpath
* flightDate, Type: MapDate, Purpose: is which map date should be accessed to get sensor information
* initialLong, Type: double, Purpose: initial longitude of drone
* initialLat, Type: double, Purpose: initial latitude of drone
* portNo, Type: int, Purpose: which port number of the WebServer to access
* nFZ Type: NoFlyZone Purpose: data on all NoFlyZones for that flightpath
* longitudeBounds Type: final double[]. Purpose: constant array that that the lower (element 0) and upper (element 1) longitude bounds of the map
* latitudeBounds. Type: final double[]. Purpose: constant array that that the lower (element 0) and upper (element 1) latitude bounds of the map
* (r) Type: final double. Purpose: constant of the distance a drone can travel in a straight line

Methods:

-droneFlightPath(MapDate flDate, ArrayList<Sensor> tSensors,double iLong, double iLat, int portN)

Description

Constructor class for object

Parameters

* flDate -> date of sensor map you want to access as MapDate type
* tSensors -> all the sensors the drone should visit during its flight, as an ArrayList of Sensor objects
* iLong -> initial Longitude of drone
* iLat -> initial Latitude of drone
* portN -> portNumber of WebServer to access

-calculateFlightPath()

Description

Calculates the flightpath of the drone for the date given by field flightDate and assigns all the moves made, in order, to the movesMade field

Parameters

None

Returns

None

-getNearestSensor(double cLong, double cLat,ArrayList<Sensor> sRemaining)  
 Description

Given a List of sensors, finds the sensor with the longitude and latitude closest to current longitude and latitude

Parameters

* cLong -> current Longitude of drone
* cLat -> current Latitude of drone
* sRemaining -> number of sensors remaining for drone to visit

Returns

The nearest sensor to the drone as a Sensor object

-calculateEuclidDist(double xLong, double xLat, double yLong, double yLat)

Description

Returns the Euclidean distance between two pairs of longitudes and latitudes

Parameters

* aLong, aLat -> the longitude and latitude pair for some point a
* bLong, blat -> the longitude and latitude pair for some point b

Returns

Value of Euclidean distance as type double

-getCoordsOfLine(double currLong, double currLat,double newLong, double newLat)

Description

Given the current longitude and latitude of the drone and the new longitude and latitude of the drone after it moves, this method gets coordinates across the line it travels. These values are used to check if the Drone has gone over any NoFlyZones.

Parameters

* currLong, currLat -> current longitude and latitude of drone
* newLong, newLat -> the longitude and latitude of the drone after it moves to a new location

Returns

A 2D Array of type double with the number of coordinate you want to access as the rows(200 coordinates in total), and the longitude, latitude of coordinate on the line as the columns with [0] corresponding to longitude, [1] latitude. 200x2 double 2D Array

-inRangeSensor(double newLong, double newLat, Sensor cSensor)

Description

Checks if location of drone in terms of longitude and latitude is in range of the sensor (range is <0.0002)

Parameters

* newLong,newLat -> longitude and latitude of the drone after it has made a move
* cSensor -> Sensor that is currently being checked if in range

Returns

Boolean value of if in range or not

-isALegalDroneMove(Direction d, double newLong, double newLat, double currLong, double currLat, double[][] lineCoords, Sensor nearestSensor)

Description

Checks, that the move the drone is making is not going over any NoFlyZones or out the boundaries of the map.

Parameters

* d -> the Direction, as an object, the drone is going in for this move
* newLong,newLat -> the longitude and latitude after the drone has moved
* currLong,currLat -> longitude and latitude before the drone has moved
* lineCoords -> 200 coordinates across the line the drone makes going from its current long/lat to its new long/lat
* nearestSensor -> the nearestSensor to the drone’s current location

Returns

The Direction object of the direction of a legal move that can be made from that drone’s position

-calculateNewDroneAngle(double sensX, double sensY, double droneX, double droneY)

Description

Calculates the angle of direction the drone will travel

Parameters

* sensX, sensY -> the longitude (or x coordinate), and latitude (or y coordinate) of Sensor the drone is travelling to
* droneX,droneY -> the longitude and latitude respectively of where the drone currently is on the map

Returns

A Direction object of the direction the drone should travel to get to the sensor

-calculateNewDronePosition(Direction dDirect,Sensor sClose, double cX, double cY)

Description

Given parameters, gets new position of location of the drone once it has finished its move

Parameters

* dDirect -> the Direction, as an object, the drone is going in for this move
* sClose -> the closest sensor the drone is trying to move within range o
* cX,cY -> the current longitude(cX) and latitude(cY) of the drone, before it moves

Returns

An array, of size 2, of type double wherein the *first* element is the new longitude after the drone has moved and the *second* element is the new latitude after the drone moved

SensorsToday

Description:

A class that, given a date, connects to a Webserver, gets the information on sensors for that date then parses that information to an ArrayList of Sensors

Fields:

All fields have private access modifiers

* todaysSensors Type: ArrayList<Sensor> Purpose: Essentially the ‘output’ of the object of this class, a list of all Sensors the drone has to visit
* todaysDate. Type: MapDate Purpose: The date of the map of Sensors, to be access from the WebServer
* portNo Type: int Purpose: The port number to connect to the WebServer on

Methods

-SensorsToday(MapDate tDate,int pNo)

Description

Public Constructor to generate SensorsToday object

Parameters

* tDate -> the date of which map to be accessed
* pNo. -> the port number of which we access the WebServer

Returns

-None-

-createUrlString()

Description

Creates the url string to of the Web Server address and the specific file we want to return

Parameters

-None-

Returns

String of full url with pathname and file type at the end

-parseSensorData(String sData)

Description

Parses then assigns to the field todaysSensors all the sensor information from a map file

Parameters

* sData -> all the data from the file recieved from the WebServer as a string

Returns

-None-

Sensor

Description:

Class that represents each Sensor object

Fields:

All fields have private modifiers except for sensorWord

* battery, Type: double Purpose: field stores parsed data on battery level of the drone
* reading, Type:String Purpose: field stores parsed data of sensor reading, is string due to possibility of ‘null’ or ‘NaN’ readings
* location, Type String Purpose: field stores parsed data on What3Words location of sensor
* sensorWord, Type: Words Purpose:field stores static inner class structure which is used to store latitude and longitude coordinates of the Sensor location e.g sensor\_object.sensorWord.coordinates.lng

Methods

-Sensor(double initLong, double initLat)

Description

Public Constructor used during DroneFlightPath to assign return location information to a sensor object

Parameters

* initLong, initLat -> the longitude and latitude coordinates of the Sensor

Returns

-None-

-ThreeWordsToLongLat(int pNo)

Description

Creates an Http Request to get longitude/latitude details from the Words Folder on the WebServer, then assigns these details to the sensorWord.coordinates.lat/lng field

Parameters

* pNo -> port no of WebServer to connect to

Returns

-None-

-get3WordsUrl(int pNo)

Description

Gets 3WordsUrl for location of sensor

Parameters

* pNo -> port no of WebServer to connect to

Returns

String of Url to access some 3WordsLocation details on the WebServer

-parseThreeWords(String tWords)

Description

Given string output from WebServer parses this information, assigns these location details to the sensorWord.coordinates.lat/lng field

Parameters

* tWords -> the Stirng output of the What3Words location in terms of longitude and latitude

Returns

-None-

NoFlyZone

Description:

Stores the details of the NoFlyZones for the map and has methods to detect if a line of points is in a NoFlyZone

Fields

* noFlyZones Type: ArrayList<Polygon> Purpose: ArrayList of Mapbox polygon objects of all the NoFlyZones

Methods

-NoFlyZone(int portNo)

Description

Public Constructor for class

Parameters

* pNo -> port no of WebServer to connect to

Returns

-None-

-parseGeoJson(String geoString)

Description

Parses GeoJson String received from WebServer, assigning its Polygon contents to the field NoFlyZones

Parameters

* geoString -> String of GeoJson file received from WebServer

Returns

-None-

-inNoFlyZone(double newLong, double newLat, double[][] lineCoords)

Description

Given both the longitude/latitude of the drone once it has travelled as part of its move and other coordinates along its line travel, determines if any of those points fall within a NoFlyZone

Parameters

* newLong,newLat -> the longitude and latitude of the drone once it has finished its move
* lineCoords -> 2D 200x2 array storing coordinates along the line the drone has travelled. The rows stores what number of coordinate and the columns if it is longitude (element 0) or latitude (element 1)

Returns

Boolean value if any points were found to be in any NoFlyZone Polygons

Move

Description:

Class that represents each move made by the drone as it goes across the map as an object

Fields:

* prevLong, Type: double, Purpose: stores previous longitude location of the drone before the move was made
* prevLat, Type: double, Purpose: stores previous latitude location of the drone before the move was made
* currLong, Type: double, Purpose: stores longitude location of drone after move has been made
* currLat, Type: double, Purpose: stores latitude location of drone after move has been made
* currentDirection, Type: Direction Purpose: stores direction object of the direction the drone went in to end up in current location
* sensorLocation, Type: String, Purpose : if the drone visited a Sensor, stores the What3Words location of that sensor, otherwise stores null

Methods

-Move(double pLong,double pLat,double cLong,double cLat,Direction cDirection,String sLocation)

Description

Public Constructor for Move Class

Parameters

* pLong, pLat -> previous longitude and latitudes of drone location before move was made
* cLong,cLat -> current longitude and latitudes of drone location
* cDirection -> direction drone had to go in to get to current location
* sLocation -> if drone visited a sensor, stores What3Words location of sensor

Returns

-None-

Direction

Description:

Class represents the direction a drone can travel in, that being direction in degrees in multiples of 10 up to 350 degrees.

Fields:

* northIsAt, Type: final int, Purpose: A constant value to remind other users that North is measure from 90 degrees
* directionDegree, Type: int. Purpose: holds the degree value, of the direction for the drone to go in, can only be a multiple of 10

Methods:

-Direction(int dir)

Description

Public Constructor

Parameters

* Dir -> direction in degrees, that the drone has travelled

Returns

-None but rounds input dir to nearest multiple of ten degrees in incorrect input given

MapDate

Description:

Holds the date of the which Map and its Sensor information to access to get a particular flightpath

Fields:

All fields are private access

* year, Type: int, Purpose: holds year of map date to access
* month, Type:int, Purpose: holds month of map date to access
* day, Type:int, Purpose: holds day of map date to access

Methods

-MapDate(String y, String m, String d)

Description

Public Constructor

Parameters

* (y) -> year of date
* (m) -> month of date
* (d) -> day of date

Returns

-None-

Words

Description: A class set up to mimic the field layout of the air-quality.json files to allow for easier parsing of data. Contains a static inner class called Coordinates that serves as a field within a field, like in air-quality.json, to store the altitude and longitude coordinates

Fields:

* coordinates, Type: coordinates object Purpose: acts as a field within a field structure for Json parsing

Fields for coordinates:

* lng, Type: double Purpose: stores longitude of Sensor Location
* lat, Type:double. Purpose: stores latitude of Sensor Location

Methods

-Words()

Description

Public Constructor, only used for when assigning return location as final sensor

Parameters

-None-

Returns

-None-

*Drone Control Algorithm*

Overview

My drone control algorithm implements a ‘Greedy’ Approach to find the order in which the sensors should be visited. This means that after each move made the algorithm determines what the nearest sensor is and endeavours to reach this nearest sensor.

1.The algorithm starts with a basic outer for loop, that ensures it can only iterate 149 times (from 0) for the maximum 150 moves. Within the for loop the algorithm works as such

2.To get the nearest sensor it goes through a list of the sensors remaining and calculates the Euclidean distance between current position of the drone and the sensor location. It takes the minimum of all of these distances using a basic find minimum algorithm[[1]](#footnote-1)

3.Once it finds the minimum, using trigonometry it calculates the angle direction which the drone should travel in order to reach the sensor. It then rounds that angle, in degrees, to a multiple of ten then, using trigonometry again, calculates the new position the drone would be in after travelling along that direction for 0.0003 degrees.

4.It then checks if it would go over any NoFlyZones by making this move, by first calculating the equation of a straight line from the previous longitude, latitude of the drone and the new longitude and latitude. From this straight line equation it calculates a whole number of coordinates along the line and then sends this information to a method in the NoFlyZone class which uses TurfJoins.PointsWithinPolygon method to check *if any of these points* that make up the straight line line *within any of the NoFlyZone polygons*.

4.1 - If not, then the drone can travel to this new location which is does so then checks if it is in range of the sensor it is travelling to,

4.1.1 - if so it creates a Move object, detailing all the longitudes/latitudes of the move and the What3Words name of sensor it visited. It then removes this sensor form the list of sensors the drone is yet to visit.

4.1.2 - If it is not in range of a sensor it creates a Move object just of all the longitudes and latitudes.

4.2 - If it is in a NoFlyZone then the algorithm goes into a while loop and, using the drone’s current location *tries a different direction by incrementing the current direction* by 10 degrees. It then calculates the drone’s location if it travelled 0.0003 degrees in this new direction and checks again if this new location is in a NoFlyZone. If it is repeat this process(2) until it finds a direction whose end location does not end in a NoFlyZone. If not then the algorithm can exit the while loop and go to 4.1.

5.The algorithm then adds the created Move object to a list of Move objects and assigns the new drone location from this move as current drone location for the next iteration

6.Basic outer for loop ends

7.Algorithm then checks if there are still under 150 moves and all sensors have been visited, i,e the sensors remaining list is empty. If so it takes the initial longitude and latitude of the drone before it made any moves and makes a Sensor object from this.

8.A while loop is then started which goes either until the number of moves runs out or the drone is in range of this final ‘Sensor’, that being the return location.

9.Within the while loop it uses the same methods used to calculate the direction, new position of the drone and whether or not any moves cause the drone to go over any NoFlyZones with the obvious edit of that it is not checking for the nearest sensor each time, it is just trying to get back to its initial location in as few moves as possible. (essentially retreating steps 3-5)

10.Once the while loop ends the algorithm is over and a flightpath, made up of the list of moves, has been generated

Possible Issues/Exceptions

In this section I’ll talk about the possible issues and exceptions that arise from using a naive ‘greedy’ algorithm and the ways I have dealt with such issues.

The first issue someone might have with the algorithm is simply the fact that it is a Greedy Algorithm. Someone might point to trying to use say a Convex-Hull Algorithm or some kind of dynamic programming approach to get an order in which to visit the sensors. However I have chosen a greedy algorithm, as for one, it is a lot faster than some other choices of algorithms with only one loop inside a loop for the calculations (about O(n^2) although with certain other time complexity issues with Java, this is purely a loose guide to the theoretical time cost) and also while it may not always get the best possible solution it those cases it does usually return a good approximation of the solution.

The second issue is to do with the case in checking is a line the drone travels is in the NoFlyZones. This being that there may not be as many equally spaced points calculated on the line so that part of the line might intersect with the polygon but there are not points calculated for that part of the line for the algorithm to recognise that does go through the polygon. My solution to this, is not particularly elegant but works in that the sheer number of points I have calculated (200) means that the case is unlikely to happen unless part of the NoFly Polygon itself was very small. If this was the case then again this would simply require more points to be added on the line to account for this exception.

Example Output of Algorithm

This is output for date 4/4/2021

This is the output for date 4/4/2020

References

1. -Title Maximum and minimum of an array using minimum number of comparisons, GeeksForGeeks, <https://www.geeksforgeeks.org/maximum-and-minimum-in-an-array/>, Last Accessed: Thursday 3rd December

1. Link in Refs; not the code nor exact algorithm I used to find minimum but gives a general idea [↑](#footnote-ref-1)