Informatics Large Practical

Report - Coursework 2

Lorna Armstrong - s1812323

Contents

Software Architecture Description

Structure Overview

The application is formed through a collection of 8 vital classes: App, GeographicalArea, Word, Sensor, Line, NearestInsertion, Drone and Word. Together, they form an application that takes in a starting position, a date, port number and a random seed, and outputs the result of an autonomous drone's flight around a defined area, collecting air quality sensor readings.

The **App** class is the main class, containing the simulation of the drone flight over the geographical area of the campus; the class coordinates the methods and links together the Nearest Insertion algorithm, the map of sensors and no fly zones for a given day, and the drone flight. The user input is read into the program through the App class (day, month, year, starting latitude, starting longitude, random seed and port number.). The App class then sets up the map by calling the *setUp* method in *GeographicalArea* class*,* and calls the *generateSensorOrder* method in *NearestInsertion* class. Calling these methods is the precondition for the drone flight, as it needs the map of sensors and no fly zones to be loaded from the server, the distance matrix to be populated, and the order in which sensors should aim to be visited, in order to begin its flight. It is also responsible for writing the output files[1], after the drone has completed its flight.

The **GeographicalArea** class is used to hold all information about the map and its contents. When an instance of *GeographicalArea* is constructed, it is passed in a date, in three Strings of *date, month, year.* This date makes the GeographicalArea unique, as it represents the area on that given date (this affects the sensors to be visited). The map information includes the No Fly Zone polygons (there are 4 polygons[2]) and the list of sensors to be visited on that day, both of which are fetched from the server by methods within the class (*getSensorListFromServer* and *getNoFlyZonesFromServer).* A *distanceMatrix* is calculated in a method, *calculateDistanceMatrix*, containing the Euclidean distances between all sensors, each other and the start point. This is then stored as an attribute. The distance matrix logic is within the *GeographicalArea* class because it is an attribute that logically belongs to the area as it depends on the list of sensors for the specific map, and their positions. It is needed because the Nearest Insertion Heuristic requires the distances between all sensors and each other, and the starting position, and storing this as a matrix makes for both easy access and an efficient solution (rather than calculating the distance between two sensors every time it's required).

The **Word** class has no methods and only has one attribute - *coordinates*. It is required, however, because it is used when parsing the data for the location of a sensor, based on its

[1] readings-DD-MM-YYYY.geojson (containing the Geo-JSON for 33 sensor markers and the flightpath of the drone as a LineString), and flightpath-DD-MM-YYYY.txt (containing the details for each move of the drone).
[2] According to requirements in the specification.

What3Words value, from one of the top-level folders of the server (words). The response JSON string is parsed from JSON to Gson, into a Word object, in the *translateLocation* method of the *Sensor* class.

The **Sensor** class is used to represent the air quality sensors from around the geographical area. The sensors are each represented as an instance of the Sensor class, with attributes for each property: location (What3Words), coordinate (latitude, longitude), battery, reading, and sensor range (= 0.0002 degrees according to the initial requirements). It is used in many other classes, which require interaction with the Sensor. There is a method in this class for key functionality - *translateLocation.* This is used to get the latitude and longitude equivalent of a What3Words address, using the information on the web server. This is needed because distances are calculated using the latitude and longitude, as the What3Words address cannot be used for this purpose. In addition, there is an equality method for checking if two sensors are equal.

The **NearestInsertion** class is used to implement the Nearest Insertion Heuristic algorithm (for explanation, see Drone Control Algorithm section). This is a key part of the program because the initial order of sensors to visit is determined by the result of this algorithm, before the drone begins its flight. It is called by the *App* class as part of the set up before the drone's flight, and returns a list of ordered sensors when the method *generateSensorOrder* is called.

Another vital class is the **Drone** class, which holds the main functionality of the drone. It contains the methods needed to fly the drone, collect information from the sensors, and record the flight path. It also contains methods to check if the drone is within range of a sensor (or of its starting position), to take a reading of a sensor, and to check that the drone movement is legal (i.e. doesn't leave the confinement zone, form a large loop, or cross any No Fly Zone boundaries).

Positions of both the sensors and the drone are represented using the **Coordinate** class, where there are two attributes: latitude and longitude. Although these positions can be represented using Point from Mapbox SDK, the creation of a Coordinate class enabled more control over adding methods and making the Coordinate more specific to this program and its requirements. To make the translation from Coordinate to Point, there is a method *toPoint* which returns the Point with the same latitude and longitude values. It is necessary because the final output files require the use of Point to create the relevant Features. This enables the smooth use of Coordinates throughout, and translation to Point to utilise any necessary methods from Mapbox SDK, whilst also using highly-relevant methods from the Coordinate class.

A useful class within the program is the **Line** class. There are two attributes in the Line class: *coordinateA* and *coordinateB*. These are both of type *Coordinate*, and they represent the two points at each end of a line. This class is used in checking if two line segments intersect, when checking if a potential drone move intersects with a boundary of a No Fly Zone. The methods in this class are: getters and setters for both *coordinateA* and *coordinateB*, and a *toString* method used to create a string representation of the *Line* object. The *Line* class represents both a move and a boundary, but these are not broken down further into separate classes because all attributes and methods are relevant to both a move and a boundary, thus using inheritance would not aid the structure or design of the application.

Overall, all of these classes are important in defining the structure of the program, ensuring the code is readable and easy to maintain. All methods and attributes are highly relevant to their class, leading to high cohesion. Encapsulation is also used by restricting the access to some attributes and methods by using the public and private modifiers, as well as static and final where appropriate. The classes: *Drone*, *Sensor* and *GeographicalArea* are objects that can represent physical aspects of the problem. *NearestInsertion* contains the algorithmic functionality of finding a possible sensor order, which is separate from the Drone class. *App* is the class that coordinates the overall simulation, and *Coordinate* and *Line* are important for creating a smooth, logical, encapsulated program.

Class Documentation

App Class

Attributes:

The **App** class has the attributes:

● int *portNumber*	inputted into the program
● Drone *drone*	the instance of the drone.
● GeographicalArea *map*	stores the instance of the map

Methods;

- *main()* is the main method of the class and of the overall program. It contains the core of the program, collecting user input, creating an instance of each of **GeographicalArea** and **Drone** classes, needed for the program, triggering methods required for finding the route (using **NearestInsertion** class) and then calling the methods *setSensors, startRoute* and *visitSensors* (of **Drone** class). The main method also contains the writing of the output files (flightpath-DD-MM-YYYY.txt and readings-DD-MM-YYYY.geojson).
- *getFeaturesForGeoJson(): FeatureCollection*[3] is a method used to generate the collection of Features for the Geo-JSON file, containing a marker Feature for each sensor and a LineString of the drone's flight path. Within this function, it calls

[3] This notation represents a function, called getFeaturesForGeoJson, which takes no arguments and returns a FeatureCollection.

createMarkers to get the marker Features, and then creates a LineString using the *route* attribute of drone (in **Drone**).
FeatureCollection - this contains all sensor markers and the drone flight path.

- *createMarkers(GeographicalArea): ArrayList<Feature>* takes a **GeographicalArea** object as input. The *sensors* list of this GeographicalArea is iterated through, and for each sensor, a Feature is created to represent it as a marker. All sensor markers are created as Features, with properties: location, rgb-string and marker-color. If the sensor is contained in *checkedSensors* in the **Drone** class, it is also given a marker-symbol using the *getMarkerSymbol* method, as specified in the initial requirements. The Feature of each sensor marker is added to an ArrayList, which is then returned by the method once all sensors have been represented as a marker.
- *getRGBString(double): String* takes a double value and returns the corresponding RGB String of hexadecimal values, according to the given mapping[4]. It is used in *createMarkers* to set the "rgb-string" and "marker-colour" properties of the sensor markers.
- *getMarkerSymbol(double): String* takes a value and returns the corresponding marker symbol for the value, according to the given mapping.
- *writeJsonFile(String, String): void* takes a filename and a JSON string as input. It then writes the JSON to the file (.geojson).
- *writeFlightpathFile(String): void* takes a filename as input and writes each String from *flightpathData* from the **Drone** class into the file, each terminated with a newline character.

Coordinate Class

Coordinates are used throughout the program, to represent positions of a drone or sensor. They correspond to the GPS coordinate of the location (latitude and longitude).

Attributes:
In the **Coordinate** class, there are two private, final attributes:

- double *latitude* the latitude value of the GPS coordinate.
- double *longitude* the longitude value of the GPS coordinate.

Methods:

- *getLatitude(): double* is used to get the private attribute *latitude.*
- *getLongitude(): double* is used to get the private attribute *longitude.*
- *isInConfinementZone(Coordinate, Coordinate, Coordinate): boolean* returns true if both the **Coordinate**'s latitude and longitude are within the confinement zone, otherwise false. As input, it takes three **Coordinate** instances, corresponding to the top-left, bottom-left and bottom-right vertices used to define the confinement zone. It is used in: *moveDrone* (in **Drone**).

[4] See Figure 5 in the document "Informatics Large Practical" by Stephen Gilmore and Paul Jackson School of Informatics, University of Edinburgh for given mapping of sensor readings to hexadecimal RGB strings.

- *getEuclideanDistance(Coordinate): double* returns the Euclidean distance between the **Coordinate** and the passed-in **Coordinate**. This method is a key method, used in *calculateDistanceMatrix* (in **Geographical Area**), *insertIntoOrder* and *selectNearestSensor* (in **Nearest Insertion**), and *withinRange* (in **Drone**).
- *getAngle(Coordinate): int* returns the angle from the **Coordinate** to the passed-in **Coordinate**, rounded to the nearest 10 degrees, where angles are calculated anti-clockwise from a baseline of East (angle = 0). It is called in the *visitSensors* method of **Drone** class.
- *getNextPosition(int, double): Coordinate* takes in the angle (int) and the move length (double). It then uses trigonometric rules of sine and cosine to calculate and return the result of moving from the **Coordinate**, at the inputted angle, for the distance of the move length. It returns the **Coordinate** with the latitude and longitude values of the resulting position. It is called in the *moveDrone* method in **Drone** class.
- *toPoint(): Point* converts a Coordinate to a Point object. It is used in *createMarkers* (in **App**), and *startRoute* and *moveDrone* (in **Drone**).
- *equals(Coordinate): boolean* is an equality method used for comparing two Coordinates, used in *insertIntoOrder* in **NearestInsertion** class.
- *toString(): String* is a simple method useful for logging and printing out the String representation of the Coordinate.

Drone Class

A **Drone** instance is created by passing a **Coordinate** starting position for the drone and a **GeographicalArea** map containing the information for the urban area within which the drone must fly.

<u>Attributes:</u>
The **Drone** class represents a drone, with private attributes:

● Coordinate *startPosition*	the start position of the drone
● Coordinate *currentPosition,*	the current position of the drone, updated throughout flight
● int *moves*	the number of moves that the drone has enough battery to complete, reduced with every move the drone makes.
● GeographicalArea *map*	the map that the drone must fly over, with sensors and No Fly Zones for the day of the drone flight
● double *moveLength*	the length of a move by the drone (constant, 0.0003)
● boolean *returningToStart*	the state of drone: visiting sensors (F[5]) / returning to start (T).

and public attributes:

● Line<Point> *route*	the list of every Point visited by the drone during its flight.
● List<Sensor> *sensors*	the list of all Sensors the drone needs to visit

[5] F = false, T = true

- List<Sensor> *checkedSensors* the list of every Sensor the drone has visited
- List<String> *flightpathData* contains the data of the flight. Each String is a line to be later written to the output .txt file, added with every move that the drone makes.

Methods:

- *getCurrentPosition(): Coordinate* is a getter for the private attribute *currentPosition,* called in the *main* function of **App**.
- *getMoves(): int* is a getter for the private attribute *move,* used in the *main* function of **App.**
- *setSensors(List<Sensor>): void* is used to set the value of sensors list to be that of the passed-in list of sensors.
- *startRoute(): void* is a method that adds the starting position, *startPosition,* of the drone to the *route*, called in *main* in **App.**
- *visitSensors(): void* is a method used to initiate the drone beginning its flight. Until the drone has no remaining moves or has visited all sensors and returned to the start, it continues to call *getDestination, getAngle* (from **Coordinate** of current position), and *moveDrone.* The *visitSensors* method is called in the *main* method in the **App** class**.**
- *getDestination(): Coordinate* returns the next coordinate that the drone is aiming for, according to the ordered list of sensors. This is either the next sensor in the list of remaining sensors to visit, or the start point depending on the size of the remaining list of sensors. It is used in the *visitSensors* method.
- *moveDrone(int, Coordinate): void* takes an integer angle and a Coordinate destination as inputs. There are several cases checked: if the move involves flying through a No Fly Zone, a new angle is found and *moveDrone* is recursively called. Else, if the suggested move has been repeated within the last 5 moves, update the angle to be 20 degrees less than the currently-attempted angle (20 degrees enables the drone to make it out of alcoves in the No Fly Zones), and recursively call *moveDrone*. Else, if the move would lead the drone outside of the confinement zone, reduce the attempted angle by 10 and recursively call *moveDrone*. When the move is legal, the drone's position is updated, the new position is added to the *route***,** the flightpath details are added to *flightpathData*[6]*,* and if the drone is within a sensor range, it takes the reading and removes the sensor from the list of sensors to visit (*sensors*).
- *isRepeatedMove(Point, Point): boolean* is used to identify loops in the drone's movements. It checks whether the suggested move (from one passed-in point to the other) has been made in the past 5 moves, returning true if it has, else false. This is used to limit drone movement so that the drone cannot get stuck. It is called in the *moveDrone* method.
- *getNewAngleClockwise(int): int* returns the next angle anti-clockwise from the passed-in angle, using the modulo operator to ensure the angle is kept within the required range of 0 to 350. It is called in *moveDrone.*

[6] Flightpath data is in the form: line number, longitude before, latitude before, direction, longitude after, latitude after, sensor location or 'null' if no sensor is read.

- *moveIntersectsNoFlyZone(Coordinate, Coordinate): boolean* checks if the line formed by moving from the first Coordinate passed-in as an input, to the second Coordinate, crosses any boundary of the No Fly Zones, using the *isIntersecting* method of Line class, with the drone movement line and each boundary of each No Fly Zone. It is called in *moveDrone.*
- *takeReading(Sensor): void* takes the reading from the Sensor, checking the battery value, reading and getting the location. It then creates a new Sensor with this information, adding it to the *checkedSensor* list. The method is called in *moveDrone.*
- *hasCheckedSensor(Sensor): boolean* returns true if the Sensor passed-in has already been checked by the drone; if there is a sensor with the same attribute values in the *checkedSensors* variable of the drone.
- *withinRange(Coordinate, double): boolean* takes a Coordinate destination and checks whether the coordinate is within range of the destination, where the range is the passed-in double value. It is called in *moveDrone.*

GeographicalArea Class

A *GeographicalArea* instance is constructed with passed-in values of *day, month,* and *year.* An instance of GeographicalArea is created in the **App** class.

<u>Attributes:</u>
A **GeographicalArea** object has many attributes:

● String *day*	the calendar day of the chosen date for the drone flight
● String *month*	the calendar month of the chosen date for the drone flight
● String *year*	the calendar year of the chosen date for the drone flight
● List<Sensor> *sensors*	the list of 33 sensors to be visited for the chosen date
● List<Feature> *noFlyZones*	the list of 4 No Fly Zones
● double[][] *distanceMatrix*	the 34x34 matrix of distances from sensors and the start
● HttpClient *httpClient*	the HTTP client used for sending requests to the web server

It also has four attributes for each of the vertices of the confinement zone, all of which are final:

● double *topLeftConfinement*	the North-West corner of confinement zone[7]
● double *topRightConfinement*	the North-East corner of confinement zone
● double *bottomLeftConfinement*	the South-West corner of confinement zone
● double *bottomRightConfinement*	the South-East corner of confinement zone

<u>Methods:</u>

- *setUp(int, Coordinate): void* calls all methods within the class that are needed to set up the map for a drone flight: *getSensorListFromServer, getNoFlyZonesFromServers, translateLocation* (for each **Sensor**), and *calculateDistanceMatrix.* As input, it takes

[7] Confinement zone: NW (55.946233, -3.192473), NE (55.946233, -3.184319), SE (55.942617, -3.184319), SW(55.942617, -3.192473).

the int port number, passed into the functions that require the port number to form a connection to the server.

- *getSensorListFromServer(int): void* takes a port number as input, sends a HTTP request to the server, collects the list of sensors to be visited on the given day (using attributes *day, month, year*) and adds each Sensor to *sensors***.** It is called in *setUp.*
- *getNoFlyZonesFromServer(int): void* takes a port number as input, sends a HTTP request to the server, gets a FeatureCollection containing all No Fly Zone Polygon Features, and adds each one to *noFlyZones.* It is called in *setUp.*
- *calculateDistanceMatrix(Coordinate): void* takes the starting position of a drone as input, and then fills the *distanceMatrix* with the Euclidean distances from one sensor to another, and from one sensor to the start; index of (0,x | x != 0) refers to the distance from the start node to any other sensor. The matrix is symmetrical, since the distance from A to B is the same as B to A in this context. The distance is calculated using *getEuclideanDistance* method from **Coordinate** class, on each coordinate (either a sensor or the start position of the drone).

Line Class

The Line constructor takes in two Coordinate values *coordinateA* and *coordinateB* and sets the corresponding attribute values. The threshold is a constant, and is already set for all lines. Instances of this class are used to represent the line of movement of the drone and also the boundary lines of No Fly Zones.

<u>Attributes:</u>

The **Line** class represents a segment of a straight line, with two private attributes;

● Coordinate *coordinateA*	the Coordinate of the point at one end of the line
● Coordinate *coordinateB*	the Coordinate of the point at the other end of the line.
● double *THRESHOLD*	used to control tolerance when comparing double values in *isIntersecting* method.

<u>Methods</u>

- *getCoordinateA(): Coordinate* is used to get the private *coordinateA* attribute.
- *getCoordinateB(): Coordinate* is used to get the private *coordinateB* attribute.
- *isIntersecting(Line): boolean* checks if the passed-in line intersects with the current instance of Line by checking all possible cases of two line segments intersecting. It returns the boolean value of true (if intersecting, else false). This public method is used in *moveInterceptsNoFly* method in the **Drone** class.
- *toString()* is a simple method useful for logging and printing out the String representation of the Line.

NearestInsertion Class

<u>Attributes:</u>

The **NearestInsertion** has private attributes:

Coordinate *startNode*	the starting position of the drone; the first coordinate in the Nearest Insertion route.
List<Sensor> *sensorsInOrder*	starts as null, then contains the ordered list of sensors to visit, determined by the algorithm.
GeographicalArea *map*	contains the unique map for a given date

Methods

- *generateSensorOrder(): List<Sensor>* is a method that calls the other functions needed to calculate the order that a list of sensors should be visited in, using Nearest Insertion Heuristic. The methods it calls are *nearestSensorToStart,* and then *nextSensorToInclude* until all sensors have been put into the order.
- *findNearestSensorToStart(Coordinate, GeographicalArea): Sensor* loops through all of the sensors in the map (GeographicalArea) and finds the sensor closest to the passed-in start point. This is used in *generateSensorOrder*, as the closest sensor to the start is the first to be added into the route.
- *selectNearestSensor(GeographicalArea, Coordinate): Sensor* is a method that is called as part of *generateSensorOrder.* It finds the sensor that is closest to any of the sensors already added to sensorsInOrder, using the *distanceMatrix* from the **GeographicalArea** passed-in.
- *insertIntoOrder(Sensor, Coordinate): void* receives a sensor, passed in, which is the next Sensor**,** N, to add to the route, and the start position as a Coordinate**.** The sensor is inserted into *sensorsInOrder,* between nodes I and J, such that d(I,N)[8] + d(N,J) - d(I,J) is the smallest. It is called in *generateSensorOrder.*

Sensor Class

The Constructor takes in values for the *location*, *battery* and *reading*, and sets the *coordinate* to null. The coordinate is assigned in the *translateLocation* method.

Attributes:

The **Sensor** class has private attributes:

String *location*	the What3Words address of the sensor
Coordinate *coordinate*	the GPS coordinate in latitude and longitude of the sensor
double *battery*	the sensor battery value on the given day
String *reading*	the air quality reading for the sensor on the given
double *RANGE*	the constant range value (0.0002).

Methods

- *getLocation(): String* is a getter for the private attribute *location*.
- *getCoordinate(): Coordinate* is a getter for the private attribute *coordinate.*
- *setCoordinate(Coordinate): void* is a setter for the private attribute *coordinate.*

[8] d(I,N) represents the distance from I to N.

- *getBattery(): double* is a getter for the private attribute *battery.*
- *getReading(): String* is a getter for the private attribute *reading.*
- *getRange(): double* is a getter for the private, constant value of *RANGE.*
- *translateLocation(): void* creates a request for the JSON file containing information about the sensor, using the What3Words value of the sensor's *location* attribute. The response includes the corresponding latitude ("lat") and longitude ("lng") values, which are then passed into a Word object, setting the Coordinate value of *coordinates* in Word, and then setting the *coordinate* attribute of Sensor to this Coordinate. The method is called in the *setUp* method, in **GeographicalArea** class.

Word Class

Attributes:

The Word class is used when Word objects are received in the response from the server request, in *translateLocation()* in **Sensor** class. It contains one attribute:

● Coordinate *coordinates*	the coordinate value of the lat and lng from word folder on server

Methods

- *getCoordinates(): Coordinate* is a getter for the private attribute *coordinates.*
- *setCoordinates(Coordinate): void* is a setter for the private attribute *coordinates.*

Drone Control Algorithm

Visiting the air sensors in an efficient way is an example of the Travelling Salesperson Problem, requiring the drone to complete a Hamiltonian cycle (starting and finishing at the same point, and visiting every sensor exactly once if possible). It is a metric TSP, since there are no negative weights as the distance between sensors is always positive. The 33 sensors that need to be visited each day can be viewed as an unconnected graph, with each sensor being a node. The order in which the sensors are to be visited by the drone is calculated using the Nearest Insertion Heuristic.

Distance Matrix

Considering the 33 sensors that need to be visited in a given day as nodes, the program calculates a 34x34 matrix of Euclidean distances, to create a connected graph, containing the distances between the starting position and all sensors, and every sensor to every other sensor. These distances are vital to the Nearest Insertion Heuristic implementation, and can be seen in the **NearestInsertion** class.

The Nearest Insertion Heuristic

Since TSP cannot be optimally solved, the program uses the Nearest Insertion Heuristic to find an efficient route around the sensors. Before the drone moves, the order with which sensors should be visited is calculated. This then is the main route the drone aims for.

Algorithm:
Consider the ‘nodes’ to refer to the starting position and the sensors.

1. Add the drone starting position, s, to the partial tour.
2. Find the sensor, a, for which d(n,i)[9] is minimum and add this sensor to the path. The resulting tour is (s, a).
3. Find the sensor n from the sensors not yet added, such that n is the closest sensor to any node in the tour.
4. Choose the edge (i, j) from the tour with the minimum: d(i, n) + d(n, j) - d(i, j), where i and j are nodes already in the tour.
5. Insert node n between nodes i and j in the tour.
6. Check if all nodes are inserted.
    a. If yes, add the starting position to the end of the tour and finish algorithm.
    b. If not, go back to (4).

Moving the Drone

The ordered list of sensors to visit is passed into the drone. The drone can make a move of 0.0003 degrees in a straight line, at any angle between 0 and 350, where the angle is a multiple of 10, measured clockwise from 0 where 0 represents travelling East. It sets out to visit the sensors in order, but to improve efficiency, if the drone moves within range of any sensor, it takes the reading and removes it from the list of sensors left to visit.

Algorithm:

1. Get the destination.
    a. If there are no sensors in the list to visit, set the destination to the start point and the returningToStart tracker as true.
    b. Else, if there are sensors to visit, get the first sensor in the list as the destination.
2. Calculate the angle to the destination from the current position of the drone, and round the angle to the nearest 10.
3. Use this calculated angle to find the proposed next position, with move length of 0.0003.
4. Check if the move is a legal move (defined by the constraints of the problem).
    a. If the move involves flying into a No Fly Zone, it is illegal. Increase the angle by 10 anticlockwise and go back to (3).
    b. If the move would cause the drone to leave the confinement zone, it is illegal. Increase the angle by 20 degrees anticlockwise and go back to (3).
    c. If the move involves flying a repeated move within the most recent 5 moves (a small loop), it is illegal. Increase the angle by 30 degrees anticlockwise (to break out of looping areas with small gaps in the No Fly Zones) and go back to (3).

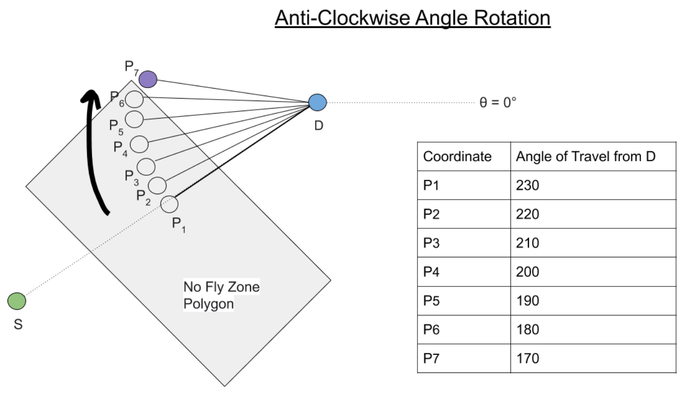
[9] d(n,j) is the distance from n to j in degrees.

5. Reduce the number of moves left for the drone by 1.
6. Move the drone to the new position and add the new position as a point in the route tracker.
7. If there is at least one sensor left, check if the drone is within range of any sensor not yet visited.
    a. If the drone is in range of a sensor, take the sensor reading and record the flightpath information into the drone's data list. Then, remove the sensor from the list left to visit.
    b. If the drone is not in range, append null to the flightpath information to record this.

Avoiding No Fly Zones

There are four No Fly zones, each a polygon in shape. When the drone calculates its next move, the line connecting the drone's current position and the proposed next position is checked to see if it intersects with any of the No Fly Zones (step 4a in movement algorithm). If there is no intersection, the drone updates its position by moving to the next position and the move is complete. If there is an intersection, the drone cannot fly to this Point because it is either inside a building or involves flying through a No Fly Zone.

In Figure 1, you can see the pivoting of the potential move, decreasing the angle, by 10 degrees, every time that the proposed new point, P, intersects the No Fly Zone. The drone then moves to the new position, the angle to the sensor is calculated and if the next move based on this is in the No Fly Zone, the process repeats. This is not the most efficient solution, but in the timeframe and respecting the project deadlines, the decision was made to focus on ensuring the overall specification was entirely met, as efficiently wasn't highlighted in the given specification.



Coordinate	Angle of Travel from D
P1	230
P2	220
P3	210
P4	200
P5	190
P6	180
P7	170

Figure 1: Anticlockwise Angle Rotation to Avoid No Fly Zones

The intersection algorithm has several cases. First, it checks whether there are any possible X values the two line segments have in common - if they don't, intersect is false. It then uses the infinite lines passing through the line segments, with equations $y = mx + c$ to check all possible cases. If the lines are parallel and don't have the same X values, then they must not intersect. If the gradient of either line is infinite, then the possible values of Y are analysed to check if an intersection occurs within the line segment Y value interval. Otherwise, the intersection point is calculated using basic linear algebra and then, if the point is within the interval, the lines intersect, otherwise they don't.

Future development of the solution could include implementing a check for both anticlockwise and clockwise approaches, to find the most efficient, resulting in a more efficient route in regards to the final destination being close to the start and all sensors being visited.

Checking Within Sensor Range and Taking Readings

After every move, the algorithm checks whether the drone is within 0.0002 degrees of any sensor. If this is the case, then the sensor reading is read, collecting the location, battery and reading. If not, the drone doesn't take a reading at this position. The choice of including any sensor that the drone flies past adapts the Nearest Insertion Heuristic and improves it, making the drone more efficient.

Maximum Moves

The drone has a maximum of 150 moves. The aim is for the drone to return close to the starting position, but the importance of visiting all sensors is greater than returning to the start, in this solution. For this reason, the drone continues to make moves until it has either used up all 150 moves or it has returned to the starting position and has visited all sensors. The final position of the drone is printed and accessible, so the team can locate the drone and collect it, even if it has not returned to the start point.

Staying in Confinement Zone

The drone must stay within the confinement zone. To achieve this, if a move would result in the drone leaving the confinement zone, it is not made. Instead, a new angle is calculated repeatedly through pivoting, as described in the movement algorithm, until the move can be made without breaking any requirements.

Loops and Getting Stuck

There are methods in place to stop the drone from becoming stuck in loops or going backwards and forwards from the same two points, using increased angles and no repeating of moves within the past 5 moves.

Reading All Sensors and Returning to the Start

The drone prioritises visiting all of the sensors over returning to the start. It will only return to the start and stop flying, if it has visited all 33 of the sensors, see *Figure 2*. If the drone runs out of moves whilst visiting the sensors, it simply stops at the last position (the position after 150 moves), see *Figure 3.* This position can be accessed using the drone's attribute *currentPosition* after the completion of the *visitSensors* method.

Drone Flight Figures

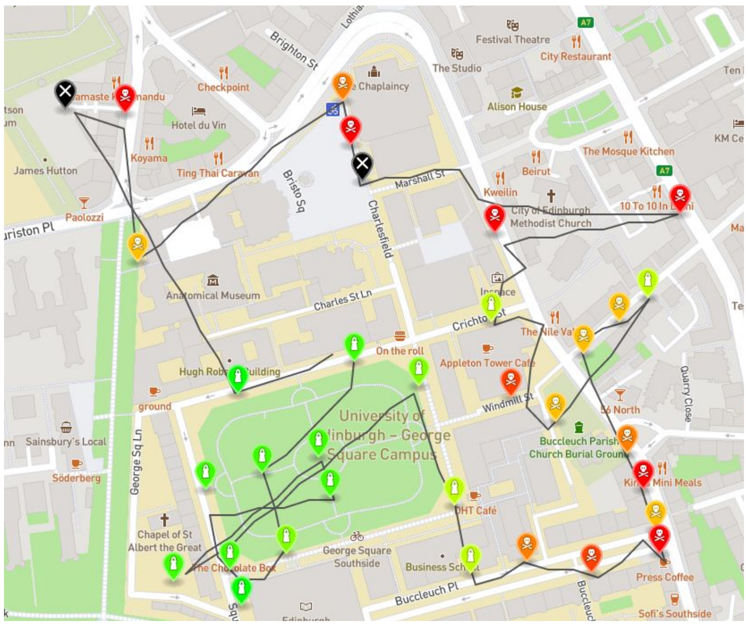


Figure 2: A sample output map for the date 02 02 2021 with a starting position of (55.944425, -3.188396)

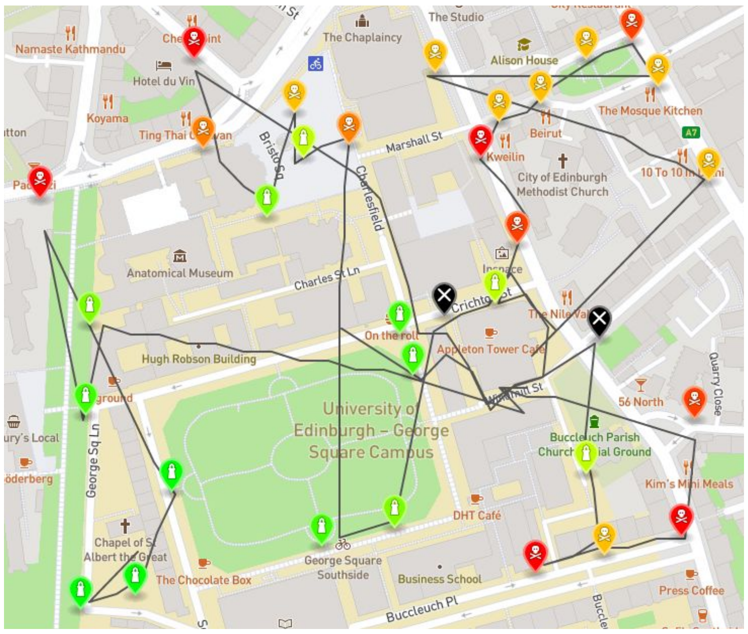


Figure 3: A sample output map for the date 02 02 2020 with a starting position of (55.944425, -3.188396)

References and Resources Used

- Informatics Large Practical *Stephen Gilmore, Paul Jackson, 2020*
- Heuristics for Traveling Salesman Problem *Christian Nilsson, Linköping University, 2003*