# Informatics Large Practical - Coursework 2

s1858928

December 4, 2020

## Contents



## 1 Software Architecture

The goal of this application is to send a Drone to visit Sensors around George Square. It makes sense that a **Drone** and **Sensor** Class would be required to keep track of the Drone's position on the map and the number of moves it is allowed it make as well as the Sensor's position, reading, battery and information on whether whether it has been read or not.

The **DroneController** class was identified in order to store all of the relevant information for the task, which includes the confinement area, no-fly-zones, the drone to be flown on the desired date, as well as the sensors which should be visited. This class is needed to send the drone on its flight for the day, using the Drone Control Algorithm specified in this report to control the path it takes. Having all of this information held in a single controller class makes more sense intuitively, improving readability and maintainability, and rather than storing it all in the Drone class this makes the code more open to development if, for example, multiple drones were to be flown on a given day.

When parsing the json files from the web server, Sensor objects could be created directly from air-quality-data.json files however in order to find the Coordinates, in longitude and latitude, of the Sensor, the details.json files from the web server would need to be parsed, requiring the **Details** class.

Mapbox's Java Geo-JSON library does not provide any explicit methods of detecting whether one LineString intersects another LineString or Polygon. This functionality would be important in the development of the application since we could model a single move of 0.0003 degrees, by the drone, as

a LineString and check whether this move intersects any of the LineStrings or Polygons representing no-fly-zones or the confinement area. Assuming the drone was started in an allowed area, if the drone doesn't cross any of the lines representing the no-fly-zones or confinement area, we can keep it in the allowed areas with this functionality. So **LineSegment** was needed to model the lines which create the no-fly-zones or confinement area and find whether lines representing moves intersect them so that it could be avoided.

**PointUtils** was identified as a class which would help improve the readability and maintainability of the code. It was simply needed to provide commonly used methods which interact with objects of the Point class and it has allowed the avoidance of code duplication.

Instead of coding the pathfinding algorithm in DroneController, a new utility class, **AStarUtils**, was created to reduce clutter in DroneController so that it would be more readable. There could be multiple different algorithms to find allowed paths from one point to another and so if somebody were to develop a different pathfinding algorithm this could be done by creating another utility class and replacing calls to AStarUtils with calls to the new class which would prevent the already lengthy DroneController class from gaining more lines of code and becoming less readable.

The A* pathfinding algorithm inside of AStarUtils uses a PriorityQueue to improve efficiency. This required the **Node** and **NodeComparator** class so that points could be sorted according to their fScore, with the lowest fScore Node coming first in the PriorityQueue.

# 2 Class Documentation

**Note that getter methods have not been included in this documentation.**

## 2.1 AStarUtils

### 2.1.1 Summary

The AStarUtils class is a utility/helper class which provides a public static method which uses the A* algorithm in order to return the optimal path that the drone should take from a starting point to a target point while making sure not to intersect any of the given line segments on the path.

### 2.1.2 Static Methods:

**public static findBestPath(Point start, Point target, List<LineSegment> noFlyLineSegments, double closeEnough)**

Pseudocode from Wikipedia (`https://en.wikipedia.org/wiki/A*_search_algorithm#Pseudocode`), accessed on 25 of October 2020, was followed in the coding and commenting of this method.

findBestPath is a public static helper method, used in DroneController, which returns a list of Points which indicates each Point which should be visited on the path from the start Point to the target Point, where the target Point is considered any Point within the distance of the closeEnough parameter from the target Point. The first element of the returned list is the start Point and the last is the target Point (or a Point close enough to the target Point). This is used to find the path that the drone should take from one point to another while avoiding buildings and staying in the confinement area. Due to the heuristic function used, this method will always return the optimal path (for more detail see the Drone Control Algorithm section).

Parameters:
Point start - The Point where the returned path should start.
Point target - The Point which should be reached by the returned path.
List<LineSegment> noFlyLineSegments - LineSegments which the returned path should never cross.
double closeEnough - The distance from the target at which it is considered close enough to have reached the desired Point.

**private static reconstructPath(Map<Point,Point> cameFrom, Point current)**

Pseudocode from Wikipedia (`https://en.wikipedia.org/wiki/A*_search_algorithm#Pseudocode`), accessed on 25 of October, was followed in the coding of this method.

reconstructPath is a private static helper method, used by findBestPath. It reconstructs the optimal path to current found by findBestPath based on the cameFrom Map. It is called once findBestPath has found a path to it's target (or a Point close enough to it) which is then passed to this method as the Point current.

Parameters:
Map<Point,Point> cameFrom - A mapping of a key of Points to values which that Point came from on it's cheapest path to that Point.
Point current - The Point at the end of the path which reconstructPath returns.

**private static getNeighbourList(Point point, List<LineSegment> noFlyLineSegments)**

getNeighbourList is a private static helper method, used by findBestPath. It returns a list of Points which are all of the possible Points the drone could reach from a given point in a single move (the neighbours). The returned list of Points will not include the result of any move which intersects any of the noFlyLineSegments.

Parameters:
Point point - The point who's neighbours are to be found.
List<LineSegment> noFlyLineSegments - The Line Segments for which the move from point to the returned neighbours should not intersect.

## 2.2 Details

### 2.2.1 Summary

This class enables Sensor to use Google's Gson parser in order to find the details.json file corresponding to its What3Words address and extract the location from it. The Details class stores all the fields of information held in the details.json files.

### 2.2.2 Instance Variables

Note that Details contains many unused instance variables, so this documentation will only list the one which is used in the application to conserve space.
private Coordinates coordinates - These are the coordinates of the sensor. Coordinates contains two fields: double lng and double lat representing the longitude and latitude.

## 2.3 Drone

### 2.3.1 Summary

The Drone class is used to keep track of the Drone's position as it moves as well as the number of moves it is allowed to make. It also updates the properties (held in the Sensor objects) to be used for the markers representing the sensors in the geojson output file of this application.

### 2.3.2 Instance Variables

private Point position - The Drone's position.
private int moveAllowance - The number of moves the drone is allowed to make.
private Map<Integer,String> colourMap - A map of the colours to be used for the markers, representing sensors, for each of the sensor's reading ranges.
private Map<Integer,String> symbolMap - A map of the symbols to be used for the markers, representing sensors, for each of the sensor's reading ranges.

### 2.3.3 Constructors

**public Drone(Point position, int moveAllowance)**

Constructs a Drone object with the given position, moveAllowance, and default colourMap and symbolMap which follows the standards given in the coursework document.

Parameters:
Point position - The position of the drone.
int moveAllowance - The number of moves the drone is allowed to make.

**public Drone(Point position)**

Constructs a Drone object with the given position, a moveAllowance of 150, and the default colourMap and symbolmap which follows the standards given in the coursework document.

Parameters:
Point position - The position of the drone.

### 2.3.4 Instance Methods

**private void setColours()**

setColours is a setter for colourMap which sets the colourMap according to the coursework specifications. Called in the constructors.

**private void setSymbols()**

setSymbols is a setter for symbolMap which sets the symbolMap according to the coursework specifications. Called in the constructors.

**public void readSensor(Sensor sensor)**

Updates the sensor's rgbString and markerSymbol instance variables according to the values specified by colourMap and symbolMap. If the battery of the sensor is less than 10% then the rgbString and markerSymbol instance variables of the sensor are updated without using colourMap or symbolMap.

Parameters:
Sensor sensor - the sensor to be read.

Throws:
IllegalStateException - if the distance between the Drone and Sensor objects' positions is not less than 0.0002 degrees.

**public void move(Point newPosition)**

Performs one move of 0.0003 degrees with the Drone object, moving it to newPosition and updating moveAllowance.

Parameters:
Point newPosition - The new position which the drone ends in as the result of a single move of 0.0003 degrees at an angle of 0-350 degrees where the angle is a multiple of 10 and the possible angles corresponds to moves at the bearings specified in the coursework document (0 east, 90 North, 180 West, 270 South).

Throws:
IllegalArgumentException - if newPosition is not a distance of 0.0003 degrees away from the Drone's current position. Accepts error less than or equal to $1 \times 10^{-14}$ degrees from the accepted distance.
IllegalArgumentException - if the angle (going counter-clockwise) between the line, from the drone's current position to newPosition, with the east pole is not a value between 0 and 350 (inclusive) or not a multiple of 10. Accepts error less than or equal to 0.00001 degrees from an accepted angle.
IllegalArgumentException - if the moveAllowance of the Drone object is not greater than 0.

## 2.4 DroneController

### 2.4.1 Summary

DroneController is responsible for accessing the web server on the desired day in order to retrieve the corresponding sensors to be visited as well as areas to avoid. It is also responsible for controlling the drone so that it visits all the required sensors, reading them when neccessary, and returns back to it's initial point while avoiding areas it is not allowed to be in. DroneController also outputs the necessary files for information on the drone's flightpath. DroneController's main method is the entry point for this application.

### 2.4.2 Instance Variables

private LineString confinementArea - The area which the drone should remain inside of.
private List<Polygon> noFlyZones - The areas which the drone should never be inside of.
private List<LineSegment> noFlyLineSegments - The LineSegments which the drone should never cross.
private List<Sensor> sensorList - The list of sensors to be visited on the given date containing information for the markers to be written to the geojson file.
private List<Feature> featureList - The features to be written to the geojson output file.
private Drone drone - The drone object which the instance of DroneController controls.
private String flightPathString - The String to be written to the flightpath txt output file.
private String year - The year component of the date for the corresponding sensors to be visited.
private String month - The month component of the date for the corresponding sensors to be visited.
private String day - The day component of the date for the corresponding sensors to be visited.

### 2.4.3 Constructors

**public DroneController(LineString confinementArea, String year, String month, String day, Drone drone, int webServerPort)**

Constructs a DroneController object and initialises the instance variables: drone, confinementArea, year, month, and day with the given arguments. sensorList is initialised for the given date by accessing the web server at the port given by webServerPort and using Google's Gson parser to parse the json file containing the sensors to be visited. noFlyZones is also initialised by accessing the same web server. flightPathString is initialised to be the empty string. noFlyLineSegments is initialised to be the line segments which make up the no-fly-zones and confinement area. featureList is initialised as an empty list.

Parameters:
LineString confinementArea - The LineString which defines the area which the drone should remain inside.
String year - The year component of the date which determines the sensors to be visited.
String month - The month component of the date which determines the sensors to be visited. This String should always be of length 2 so for the 5th month of a year this String would be "05".
String day - The day component of the date which determines the sensors to be visited. This String should always be of length 2 so for the 2nd day of a month this String would be "02".
Drone drone - The drone object who's position is used as the initial position meaning that when DroneController is used to send the drone on its flight path to visit all sensors, it will attempt to return to this initial position once it has visited them all.
int webServerPort - The port on which the web server is running.

### 2.4.4 Instance Methods

**private void setSensorList(String year, String month, String day, int webServerPort)**

Accesses the web server running at the port given by webServerPort and uses Google's Gson parser in order to parse the air-quality-data.json file for the given date and create a list of Sensor objects for the drone to visit on that date. Called in the constructor.

Parameters:
String year - The year component of the date which determines the sensors to be visited.
String month - The month component of the date which determines the sensors to be visited.
String day - The day component of the date which determines the sensors to be visited.
int webServerPort - The port at which the web server to be accessed is running.

**private void setNoFlyZones(int webServerPort)**

Accesses the web server running at the port given by webServerPort in order to find the no-fly-zones (area(s) which the drone should never be inside).

Parameters:
int webServerPort - The port at which the web server to be accessed is running.

**private void setNoFlyLineSegments()**

A setter for noFlyLineSegments. Uses the confinementArea and noFlyZones instance variables in order to set noFlyLineSegments (the line segments which the drone should never cross). Called in the constructor.

**private void writeReadings(String fileName)**

Adds markers to featureList for each of the sensors in sensorList and writes the featureList instance variable to a geojson file with a file name given by the argument fileName.

Parameters:
String fileName - The name of the geojson file to be written.

**private void writeFlightPath(String fileName)**

Writes the flightPathString instance variable to a txt file with a file name given by fileName.

Parameters:
String fileName - The name of the txt file to be written.

**private void moveDrone(Point newPosition)**

Moves the drone to a new position which must be of length 0.0003 degrees from the drone's current position and at an angle of 0-350 (inclusive) with the east pole, where the angle is a multiple of 10. This method also updates the flightPathString instance variable to contain information of the move. Note that this calls drone's move function - see the documentation on Drone's move method to understand when this will throw exceptions.

Parameters:
Point newPosition - The new position which the drone should move to.

**private void droneRead(Sensor sensor)**

Makes the drone read a sensor by calling drone's readSensor method which updates the sensor's rgbString and markerSymbol instance variables and updates the flightPathString instance variable of DroneController accordingly. Note that the drone must be within 0.0002 degrees of the sensor to read it - see the documention on drone's readSensor method to understand when this will throw exceptions.

Parameters:
Sensors sensor - the sensor to be read by the drone.

**private void droneDontRead()**

Updates the flightPathString instance variable to record that the drone did not read any sensor.

**public void greedyFlightPath()**

Sends the drone to visit each of the sensors, visiting the closest sensor to it's current position first according to the straight line distance from the drone's position to the sensor. Once the closest sensor has been found, the visitSensor instance method is used in order to move within 0.0002 degrees of the sensor and take it's reading (see Drone Control Algorithm section for more detail). Once it has done this with all of the sensors, this method uses the returnDrone instance method to move the drone back within 0.0003 degrees of it's initial position. The instance variables flightPathString, featureList, and sensorList (as well as drone) will all be updated according to the moves the drone made, and the sensors it visited. This method finally calls writeFlightPath and writeReadings in order to produce the necessary output files.

**private void visitSensor(Sensor sensor)**

Moves the drone on a path from it's current position to within 0.0002 degrees of the given sensor, using the moveDrone instance method to move the drone and update flightPathString as well as the droneRead and droneDontRead methods to update flightPathString accordingly so that when moveDrone is called it is always followed by a call to droneRead or droneDontRead. moveDrone also updates the featureList instance variable and droneRead updates the sensorList instance variable. The path which the drone takes is found using AStarUtils' findBestPath function so that the drone takes the best path while not passing through any of the line segments in the noFlyLineSegments instance variable.

Parameters:
Sensor sensor - The desired sensor for the drone to move within 0.0002 degrees of and read.

**private void returnDrone(Point startPosition)**

returnDrone moves the drone on a path from it's current position to a Point within 0.0003 degrees of startPosition. The drone will never read a sensor along this path. The method uses droneMove and dron-DontRead in order to update the flightPathString and featureList instance variables. The path which the drone takes is found using AStarUtils' findBestPath function so that the drone takes the best path while not passing through any of the line segments in the noFlyLineSegments instance variable.

Parameters:
Point startPosition - The desired Point for the drone to move within 0.0003 degrees of.

### 2.4.5 Static Methods

**public static String getResponseBody(String urlString)**

Returns the file held on the web server at the given urlString as a String.

Parameters:
String urlString - Address of the file on the web server.

### 2.4.6 Main Method

**public static void main(String[] args)**

This is the entry point of the whole application. This provides a fixed confinement area set according to the coursework guidelines and calls DroneController's greedyFlightPath method to produce the flightpath.txt and readings.geojson files for the given date using the implemented drone control algorithm.

Parameters:
args[0] - Day of the drone's flight as a String of length 2. (So for the 1st day of the month use "01" rather than "1")
args[1] - Month of the drone's flight as a String of length 2. (So for the 3rd month of the year use "03" rather than "3")
args[2] - Year of the drone's flight.
args[3] - Latitude of the drone's initial position.
args[4] - Longitude of the drone's initial position.
args[5] - Random number seed for the application - this is unused so it has no effect on the application.
args[6] - Port which the web server is running on.

## 2.5 LineSegment

### 2.5.1 Summary

LineSegment is used to model a line segment on the earth's surface. It provides an instance method which finds whether another LineSegment object is intersecting with it. This is helpful because it allows the AStarUtils' function, getNeighbourList, to return the list of points which are a result of moves which do not cross any of the lines defining the confinement area or no-fly-zones.

### 2.5.2 Instance Variables

private Boolean isVertical - If the line segment has an undefined gradient, it is a vertical line segment and so this boolean will be true.
private double gradient - Defines the gradient of non-vertical line segments.
private double yIntercept - The y-intercept of the straight non-vertical line which the line segment is a part of.
private Point endPoint1 - The Point at one end of the line segment.
private Point endPoint2 - The Point at the other end of the line segment.

### 2.5.3 Constructors

**public LineSegment(Point endPoint1, Point endPoint2)**

Constructs the line segment of a straight line between endPoint1 and endPoint2. The order of endPoint1 and endPoint2 does not matter. Checks if the line segment is a vertical and if not then, it uses the equation of the gradient, $m = \frac{y_2 - y_1}{x_2 - x_1}$, to initialise the gradient and the equation of the line, $y = mx + c \iff c = y - mx$ to initialise the y-intercept.

Parameters:
Point endPoint1 - The Point at one end of the line segment.
Point endPoint2 - The Point at the other end of the line segment.

### 2.5.4 Instance Methods

**public Boolean intersectsWith(LineSegment otherLine)**

Returns true if the instance of LineSegment intersects with the parameter otherLine and false otherwise. Makes use of the other helper instance methods isPointOnLineSegment and doEndPointsOverlap.

Parameters:
LineSegment otherLine - This method checks whether this instance of LineSegment intersects with this parameter, otherLine.

**private Boolean isPointOnLineSegment(double lng, double lat)**

Returns true if the Point made up of the arguments lng and lat is a part of the instance of LineSegment and false otherwise. When the LineSegment is not vertical, this method relies on the triangle inequality to determine whether the Point is on the line. If it is on the line then the sum of the distance from one endPoint to the given Point and the other endPoint to the given Point should be equal to the distance from one endPoint to the other endPoint otherwise it is not. If the line is vertical then an if condition is used to reduce the number of computations.

Parameters:
double lng - longitude of the Point which may or may not be on the LineSegment.
double lat - latitude of the Point which may or may not be on the LineSegment.

**private Boolean doEndPointsOverlap(LineSegment otherLine)**

Returns true if either this instance of LineSegment or otherLine has an endPoint which is on the Line Segment which it is being compared with and false otherwise. This method uses the isPointOnLineSegment method.

Parameters:
LineSegment otherLine - The LineSegment which this instance of LineSegment is being compared with.

## 2.6 Node

### 2.6.1 Summary

The purpose of the Node class is to allow the use of a PriorityQueue in the AStarUtils' findBestPath method so that the application runs faster.

### 2.6.2 Instance Variables

private Point point - The Point which the node represents.
private double fCost - The fCost of the Node where fCost = gCost + hCost (given by the heuristic function used - see Drone Control Algorithm section for more detail)

### 2.6.3 Constructors

**public Node(Point point, double fCost)**

Initialises the instance variables point and fCost to be that of the arguments point and fCost respectively.

## 2.7 NodeComparator

### 2.7.1 Summary

This class implements the Comparator interface and allows the use of the PriorityQueue by allowing the Nodes to be ordered by fCost.

### 2.7.2 Instance Methods

**@Override**
**public int compare(Node node1, Node node2)**

This overrides Comparator's compare method and creates an ordering such that the Node with the greater fCost is considered greater than a Node with a lower fCost. This means that the Node with the lowest fCost will appear at the front of a PriorityQueue.

## 2.8 PointUtils

### 2.8.1 Summary

PointUtils is a utility/helper class which provides multiple public static methods for calculating useful information to do with Point objects which are used across the application.

### 2.8.2 Static Methods

**public static double findDistanceBetween(Point point1, Point point2)**

Returns the straight line distance (euclidean distance) between two points.

Parameters:
Point point1 - The first of the two points whose euclidean distance between is being measured.
Point point2 - The second of the two points whose euclidean distance between is being measured.

**public static Point pointAfterMove(Point startPoint, int angle)**

Returns the Point which is the result of a move of 0.0003 degrees at the given angle (in degrees), where an angle of 0 degrees is east, 90 North, 180 West, and 270 South from the given startPoint. Uses basic trigonometry to find the resulting Point.

Parameters:
Point startPoint - The Point from which the move occurs.
int angle - The angle of the move with respect to the east pole (as described above) in degrees.

**public static Point subtractPoints(Point point1, Point point2)**

Subtracts the longitude of point 2 from the longitude of point 1 and does the same with the latitude and returns a point with the resulting longitude and latitude. Note that the order of the parameters matters.

Parameters:
Point point1 - The point from which values are subtracted.
Point point2 - The point whose values are used in the subtraction from point1.

**public static double pointMagnitude(Point point)**

Returns the magnitude of the Point as if it were a vector i.e. $Magnitude = \sqrt{longitude^2 + latitude^2}$.

Parameters:
Point point - The point for which the magnitude is calculated.

**public static double angleBetweenPoints(Point startPoint, Point newPoint)**

Returns the angle (in radians) that the move from the startPoint to newPoint makes with the east pole assuming that the move is of length 0.0003 degrees. Uses the equation for the dot product, $\theta = cos^{-1}(\frac{a \cdot \mathbf{b}}{\|a\|\|b\|})$, to find this angle.

Parameters:
Point startPoint - The point before the move.
Point newPoint - The resulting point after the move.

### 2.9 Sensor

#### 2.9.1 Summary

The Sensor class is used to store the information of sensors from the web server as well as update when they are visited by the drone.

#### 2.9.2 Instance Variables

private String location - The What3Words location of the sensor.
private double battery - The battery of the sensor.
private double reading - The reading of the sensor.
private Point position - The Point where the sensor is located.
private String rgbString - The String to be used to set the "rgb-string" and "marker-color" property of the marker, which represents this Sensor, by DroneController.
private String markerSymbol - The String to be used to set the "marker-symbol" property of the marker, which represents this Sensor, by DroneController.

#### 2.9.3 Instance Methods

**public void setPosition(int webServerPort)**

Uses Google's Gson parser to parse the json string held in the web server file named details.json for the What3Words location of the sensor in order to create objects of the class Details and set the position instance variable. Note that location must be initialised before this method is called.

Parameters:
int webServerPort - The port at which the web server is running.

**public void setMarkerProperties(String rgbString, String markerSymbol)**

Sets the rgbString and markerSymbol instance variables given by the arguments. This is used when recording that a drone has read a sensor.

Parameters:
String rgbString - The String to be used to set the instance variable rgbString.
String markerSymbol - The String to be used to set the instance variable markerSymbol.

## 3 Drone Control Algorithm

The Drone Control Algorithm has two aspects to it.

#### A* algorithm

Firstly, the control algorithm uses the A* algorithm in order to find the optimal path from one point to another while avoiding the no-fly-zones and staying inside of the confinement area. This is used when finding the path that the drone should take from it's initial point to a sensor, from one sensor to another sensor and from a sensor back to the initial point. The A* algorithm assigns an f-score to each node where f-score = g-score + h-score. The g-score of the node is the cheapest known distance travelled to reach that node and the h-score of the node is the heuristic function of that node. For this implementation of A*, the heuristic of a node, h(n), is the straight line distance (SLD) from n to the target minus

the distance from the target that is considered close enough. The A* algorithm explores the node with the lowest f-score first until it reaches the target and for each node it visits, it keeps track of the node that it came from on the cheapest path to the node it is visiting and updates this information as necessary.

So, for example, when visiting sensors, the drone must be within 0.0002 degrees of the sensor so the heuristic function of a node $n$ is given by, $h(n) = SLD(n, target) - 0.0002$ which is equivalent to $h(n) = SLD(n, closest\ point\ within\ 0.0002\ of\ target)$ as shown by figure 1 below where $SLD(x, y)$ is the straight line distance between two points $x$ and $y$. Furthermore, the SLD between two points is the shortest possible distance that must be travelled to reach one point from another. In figure 2, the distance of the straight line path from A to C is SLD(A,C) so any path other than the straight line from A to C must visit an intermediate point given by B (which is not a part of the line from A to C or else the path from A to B to C is the straight line from A to C) and so this will form a triangle. Since the triangle inequality states that $SLD(A, C) < SLD(A, B) + SLD(B, C)$, the straight line path from A to C must be the shortest path possible. This proves that the heuristic used will never overestimate the distance that it must travel to reach it's target (or close enough to it) so it is admissible meaning that A* will always return the optimal path to the area or point which we want to reach. The value of 0.0002 can be replaced with 0.0003 when the Drone is sent to return to it's initial position and this still holds.

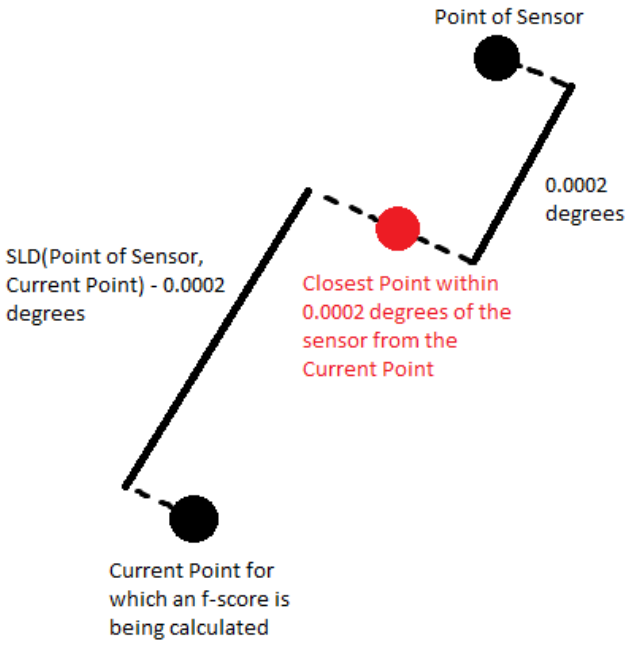


Figure 1

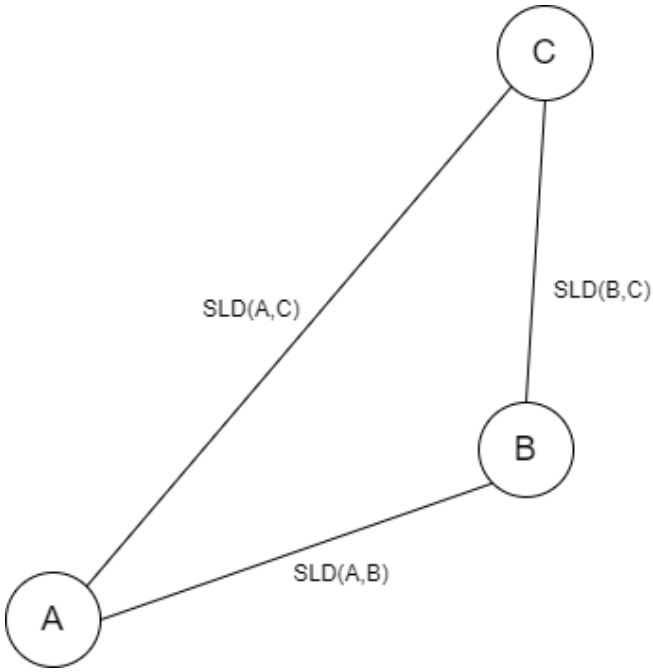


Figure 2

The worst case time complexity of A* is $O(b^d)$ where $b$ is the maximum branching factor and $d$ is the depth of the tree. The maximum branching factor, $b$, in this case is 36 since we consider a move of a fixed magnitude at 36 different angles (0-350 and a multiple of 10). The depth, $d$, would depend on the number of moves it would take to reach the target which is affected by the distance between the start point and target point. As we can see below, in figure 3, there is a call to the A* algorithm to find the sequence of moves which brings the drone from the top left corner of the confinement area to the bottom right corner. This is among the greatest distances that the drone can travel in the confinement area and as a result $d$ will be large and, running the application on 10/10/2020 will take a longer time to run than other dates without such a long sequence of moves.

Figure 3 - Flight of drone on 10/10/2020 starting at (55.944425, -3.188396) as (latitude,longitude)

### Greedy Algorithm

Secondly, the control algorithm uses the greedy algorithm to decide which sensor to visit next, calling the A* algorithm in order to find the best path to within 0.0002 degrees of the next sensor. As we know, the task of visiting all the sensors is a variant of the travelling salesman problem which is NP-hard. Therefor there is no algorithm which will find the best order in which the sensors should be visited in any reasonable amount of time. The greedy algorithm makes the drone visit the sensor with the shortest straight line distance between the drone's current position and the sensor's position and calls the A* algorithm to move the drone along the optimal path to that sensor. It repeats this process after it has read that sensor until it has visited all sensors. The control algorithm then calls the A* algorithm to send the drone back to it's initial position, if it is still allowed to make more moves. The greedy algorithm does not often give the best order of sensors to visit but it has a reasonable time complexity of $O(n^2)$, where $n$ is the number of sensors, with the implementation in this application. This is because it looks at all $n$ sensors to be visited and picks the closest one according to the SLD. Then once it has reached that sensor it looks at the remaining $n-1$ sensors and picks the closest again. It repeats this until there are no sensors left. Therefor it will have to loop $\sum_{k=1}^{n}(k) = \frac{n(n+1)}{2}$ times. The use of the greedy algorithm works reasonably well with the A* algorithm as well. By choosing to travel to the sensor that is the lowest distance from the drone, this minimizes the time complexity of A* by reducing d. This can, however, result in unfavourable moves when visiting the last few sensors as shown above, in figure 3, where the problem of travelling from the top left to bottom right corner could have been avoided if the drone was sent to visit the sensor at the bottom right while it was already in the area rather than a different sensor which was closer.

However as shown in figure 4, the flight path produced by the greedy algorithm mostly sense. It skips a sensor in a couple of situations (similarly to the situation in figure 3) and must return to visit it, however,

it does not cause the drone to waste many moves. In this case algorithms such as depth-first-search or breadth-first-search might return a worse solution depending on the order in which the nodes are arranged. The nodes would likely not be sorted with these algorithms either and so it would return a mostly random path which would most likely be worse than the solution returned by the greedy algorithm. On the other hand, in figure 4, the greedy algorithm decides to visit the sensor which it is on first, but because the drone must move before it reads a sensor this results in 2 wasted moves whereas it could have just read the sensor upon its return to the initial position. Another algorithm such as breadth-first-search or depth-first-search may not have had this problem. This proves that different algorithms for deciding the order in which to visit nodes will perform differently depending on the positions of the sensors and the drone's initial position.

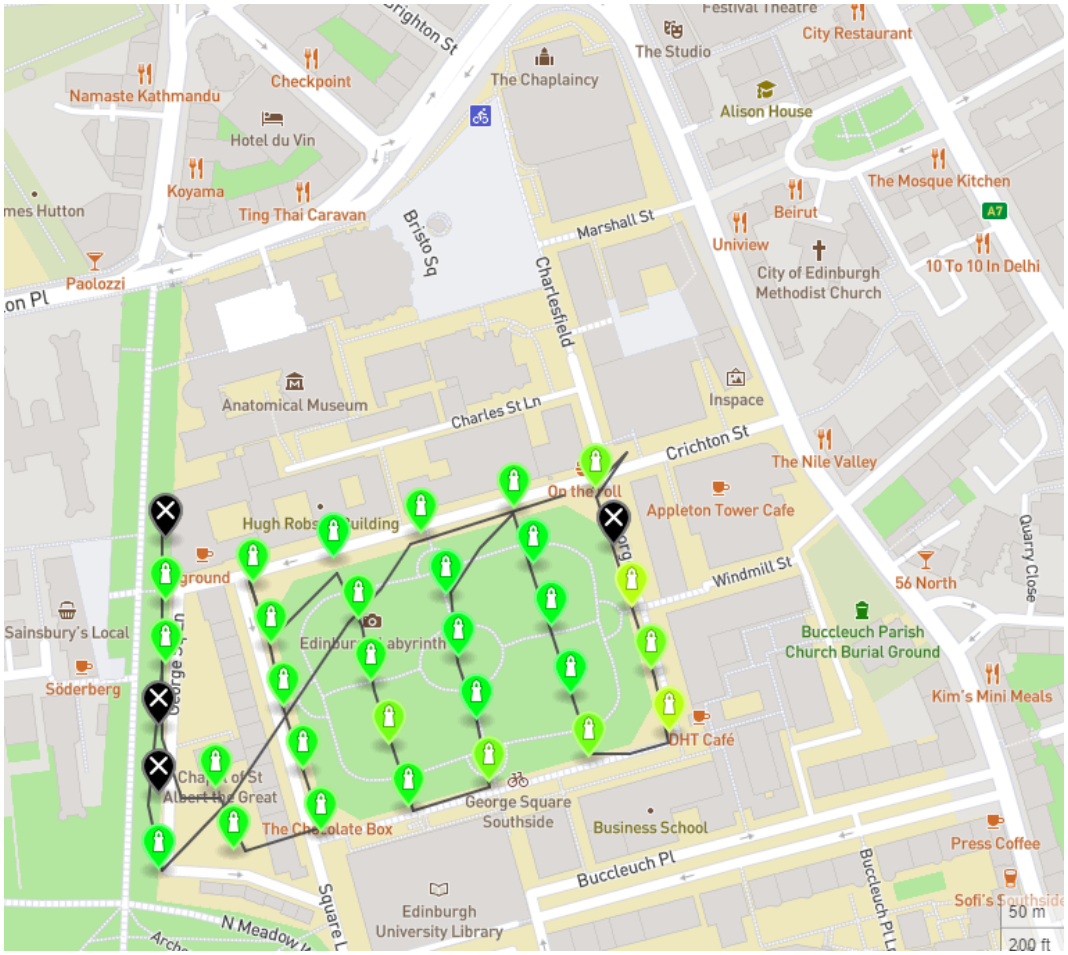

Figure 3 - Flight of drone on 10/10/2020 starting at (55.9444, -3.1878) as (latitude,longitude)