Informatics Large Practical
Coursework 2
s1803764

Table of Contents

SOFTWARE ARCHITECTURE DESCRIPTION

Project description

To program a drone to fly around and collect sensor readings from sensors in a defined confinement area. This drone can only move in steps of 0.0003 degrees and can only turn in degrees of 10. This drone must also avoid no-fly-zones in the area whilst mapping its route. After collecting readings from all the sensors this drone must return to its start point and output the readings of the sensors (to a .geojson file) and a log of the movements it made during its flight (to a .txt file).

Software architecture summary

1) Retrieving command line inputs

This program works by first retrieving all the relevant command line arguments that specify the date we want to map a route for, the starting point (in longitude and latitude), and the port of connection. Upon retrieval these inputs are validated for data type and correctness (e.g. does not allow impossible dates). If one of these inputs is invalid, the program terminates.

2) Connecting to the web server

Once these inputs are retrieved and validated our program first tries to connect to the web server at the specified port using the java HTTP class. If this fails, the program terminates.

3) Retrieving the relevant data from the web server and parsing it into Java objects

3.1) Retrieving and parsing the maps file

Assuming a successful connection to the web server our program retrieves the data for all the sensors to be read for the specified date. It does this by retrieving the *air-quality-data.json* file for the given date (in the *maps/YYYY/MM/DD/* directory) which stores the data for all the sensors: the What3Words location, the battery percentage, and the air-quality reading. These are parsed using substring indexing and are stored in custom Sensor objects.

3.2) Retrieving and parsing the What3Words files

Now that we have the sensor data, we must retrieve the respective coordinates for each What3Words location. Our program does this by iterating through all the different sensors to retrieve the *details.json* file for each given What3Words location (in the *words/W1/W2/W3/* directory). The only data we need from this file is the 'coordinates' object which represents the centre of the respective What3Words tile. These are parsed using substring indexing and are stored in their respective Sensor objects. All the sensors are then stored in one global ArrayList of Sensor objects called *sensor*.

3.3) Retrieving and parsing the no-fly-zones file

The last thing we need to retrieve from the web server is the no fly zones for the drone, these represent tall buildings which the drone must avoid in order to prevent a crash. Our program does this by retrieving the *no-fly-zones.geojson* file (in the *buildings/* directory). This file contains a GeoJSON FeatureCollection which stores each no-fly-zone as a feature. The only data we need from each of these features is the Polygon geometry object which stores an array of coordinates representing the vertices for the given no-fly-zone. These are parsed using substring indexing and are stored in custom NoFlyZone objects. All the no-fly zones are then stored in one global ArrayList of NoFlyZone objects called *noFlyZones*.

4) Find optimal sensor route

We use an algorithm to determine the optimal route to visit sensors. I implemented many different algorithms to find the best based on both performance and execution time.

5) Calculate and record valid moves for the drone to follow this route

We start at the coordinates specified by the input arguments. Our program then iterates through the *findNextMove()* method which returns a valid move that takes the drone closer to the next sensor in the queue. Each move is recorded for the log file, and every time a sensor is read it is stored as a GeoJSON air-quality marker. This is iterated until the drone has visited all the sensors in the route. Then, all the sensors that are not visited by the end of the drone's route are recorded as blank GeoJSON air-quality markers. These log and GeoJSON files are then outputted to the current working directory.

Java class descriptions:

Object classes

These classes are used to represent data structures effectively and create reusable methods.

- Fragment
  This is a custom class for the *temperate()* algorithm. Simply put, this object represents an edge between 2 sensors. Find a full explanation of its implementation by reading the description of the *temperate()* algorithm in the 'Drone control algorithm' section.
- Point
  This object represents geographical coordinates.
- NoFlyZone
  This object represents a no-fly zone.
- Sensor
  This object represents an air-quality sensor.
- LineGraph
  This object represents a straight-line function. It is extremely useful and effective for calculating whether a given drone path goes across a no-fly zone boundary. A full description of this utilisation can be found in the description for 'Avoiding no-fly zones and staying within the confinement area' in the 'Drone control algorithm' section.
- Move
  This object represents the movement of a drone between two points. This is very useful when calculating moves in *findNextMove()* as it allows us to easily store/log all of our drone's movements.
- Route
  This object represents a given drone route. I decided to represent a route as an object as it allows us to neatly store all the relevant data and functions. This not only promotes good readability but also gives us the ability to easily map many different routes and compare their results.

Method classes

These classes are used to neatly group different types of static methods to promote maximum readability.

- Algorithms
  This class is used to store all the sensor route optimisation algorithms. The workings of all these algorithms are explained in depth in the descriptions under 'Algorithms used' in the 'Drone control algorithm' section.
- FileReading
  This class is used to store all the methods used for parsing web server files.
- FileWriting
  This class is used to store all the methods that are used to write to files.
- GeometricalCalcs
  This class is used to store all the geometrical calculation methods.
- MoveCalcs
  This class is used to store all the single move calculation methods.
- Webserver
  This class stores the methods which allow us to connect to the web server and retrieve files.

Main class (App.java)

This is the primary class. Here we retrieve the command line arguments, store global constants and variables, and utilize our various other classes to effectively map a drone route

Java imports used:

- Java.io.FileWriter
- Java.io.IOException
- Java.util.ArrayList
- Java.util.Arrays
- Java.net.URI
- Java.net.http.HttpClient
- Java.net.http.HttpRequest
- Java.net.http.HttpResponse.BodyHandlers

CLASS DOCUMENTATION

Object classes

Note that **all** the variables for each of the following objects are private and non-static. Private because they can be accessed via getter methods, and non-static as no default values are used meaning an object instance is required. Along with this, **all** class methods are non-static as they all require data from object instances. Class methods are only made public if they are accessed outside of their native class.

***NOTE:** getter and setter methods will be omitted from each class' 'Methods' description. If a given class has no 'Methods' description this is because it has no methods.*

Fragment

Variables:

`Sensor sensor`: represents the original sensor to be expanded upon.

`Double avgDist`: represents the mean distance *sensor* is from all the other sensors for the given day.

`Sensor bestDestSensor`: represents the best sensor for the drone to visit after *sensor* (closest).

Point

Variables:

`Double lng`: represents the longitude coordinate

`Double lat`: represents the latitude coordinate

Methods:

`Boolean isEqual(Point pointA)`

Checks whether the given point instance is equivalent to pointA. Returns true if so, false otherwise.

`Boolean checkConfinement()`

Checks whether the given point instance is within the confinement area. Returns true if so, false otherwise.

NoFlyZone

Variables:

`ArrayList<Point> points`: represents the vertices of the given no fly zone.

Sensor

Variables:

`String location`: represents the What3Words location of the given sensor.

`Double battery`: represents the battery reading of the sensor.

`Double reading`: represents the air-quality reading made by the sensor.

`Point point`: represents the geographical coordinates of the sensor.

Methods:

`String getReadingColour()`

Air-quality classification method which returns an rgb-string based on the given sensor's air-quality reading.

`String getReadingSymbol()`

Air-quality classification method which returns the name of a symbol based on the given sensor's air-quality reading.

LineGraph

Variables:

`Double gradient`: represents the gradient of the function.
`Double yint`: represents the y-intercept of the function.
`Point p1`: represents the first point that was used to define the given straight-line function.
`Point p2`: represents the second point that was used to define the given straight-line function.

Move

Variables:

`Point origin`: represents the point of the drone at the beginning of the move.
`Point dest`: represents the point of the drone at the end of the move.
`Double angle`: represents the angle of movement from *origin* to *dest*.

Route

Variables:

`ArrayList<Sensor> sensorRoute`: represents the optimized route of sensors for the drone to follow.
`ArrayList<Point> pointRoute`: represents the entire route the drone takes.
`ArrayList<Sensor> unreadSensors`: represents the sensors the drone has not visited yet.
`String dataGeojson`: represents GeoJSON code which stores the markers for all of the sensors as Point features and the drone route as a LineString feature.
`String flightpathTxt`: represents the contents of the drone movement log file. Each line represents a move and stores data in the following order; the move number, coordinates before the move, angle of movement, coordinates after the move, and the What3Words location of the sensor visited in that move ("null" if none).

Methods:

`void findMoves()`
Given the starting point and route of sensors to follow, this method iterates through the *findNextMove()* function which takes the drone's current position and finds the optimal next position given what sensor needs to be visited next. Each of these found moves are then appended to the log file variable *flightpathTxt*, and every time a sensor is visited a classification marker is appended to the GeoJSON variable *dataGeojson*. This is iterated until either all the sensors have been read or the drone has completed 150 moves. If the drone did not manage to visit all the sensors then the sensors that were not visited are appended to *dataGeojson* as blank markers.
`void writeOutputFiles()`
This method outputs the route's resulting GeoJSON and text files in the current working directory.

Method classes

Note that **all** the variables and methods for the following method classes are static. This is due to the fact they are not treated as objects and thus evidently do not rely on data stored in object instances. The only data they need are retrieved via public variable/method calls and input arguments. Again, class methods are only made public if they are accessed outside of their native class.

Algorithms

Methods:

`ArrayList<Sensor> greedy()`
This method outputs an optimized sensor route by prioritizing the closest sensors.
`ArrayList<Sensor> temperate()`
This method outputs an optimized route of sensors by prioritizing sensors with the largest mean distance and expanding them with the closest available sensor.
`ArrayList<Sensor> twoOpt(ArrayList<Sensor> sensorRoute)`
This method outputs an optimized version of the input route *sensorRoute* by naively swapping sensors in the route to see if it improves the overall route cost.
`ArrayList<Sensor> swap(ArrayList<Sensor> sensorRoute)`
This method outputs an optimized version of the input route *sensorRoute* by naively swapping adjacent sensors in the route to see if it improves the overall route cost.
`Sensor getClosestSensor(Sensor sens, ArrayList<Sensor> sensorRoute)`
Custom helper method for the temperate algorithm. This outputs the closest sensor to *sens* that is not in the *sensorRoute* (excluding itself of course).
`ArrayList<Sensor> getClosestSensors(Sensor sens)`
Custom helper method for the temperate algorithm. This returns an ordered list of the closest sensors to *sens* (ascending in distance).

FileReading

Methods:

`ArrayList<Sensor> parseJsonSensors(String fileContents)`
This method parses the contents of the *air-quality-data.json* file for the given date. It iterates through the lines of *fileContents* and retrieves data using substring indexing. Data for each sensor in the file is stored in a Sensor object. Each of these sensors is then appended to a Sensor ArrayList which is outputted by this method.
`Point parseJsonW3Wtile(String fileContents)`
This method parses the contents of a given What3Words *details.json* file. The only data we desire here is the "coordinates" parameter as this represents the central point of the given What3Words tile. This data is retrieved using substring indexing and is then stored in a Point object which is outputted by this method.
`ArrayList<Sensor> getSensorCoords(ArrayList<Sensor> inputSensors)`
Each of the sensors in the *inputSensors* ArrayList only have a What3Words location. This method allows us to retrieve the geographical coordinates for each of these sensors by iterating through the sensors and calling *parseJsonW3Wtile()* for each one. The Point object returned is then stored in the respective Sensor object. Once all sensors coordinates have been retrieved the fulfilled Sensor objects are all returned in an ArrayList.
`ArrayList<NoFlyZone> parseNoFlyZones(String fileContents)`
This method parses the contents of the *no-fly-zones.geojson* file. Each no-fly zone is represented by a list of points which represent its vertices. This method iterates through the lines of *fileContents* and retrieves the point data using substring indexing. Data for each each no-fly zone is stored in a NoFlyZone object. Each of these no-fly zones is then append to a NoFlyZone ArrayList which is outputted by this method.

FileWriting

Variables:

`final String endFeatureCollectionGeojson`: constant used to represent the closing syntax of a GeoJSON FeatureCollection.
`final String startMarkerGeojson`: constant used to represent the opening syntax of a GeoJSON Point.

`final String startLineStringGeojson`: constant used to representing the opening syntax of a GeoJSON LineString.

Methods:

`String getGeojsonMarker(Sensor sens, Boolean beenVisited)`
This method parses the input Sensor *sens* into code for a GeoJSON Point. This is used to geographically represent an air-quality marker. The *beenVisited* parameter specifies if the input Sensor has been visited and thus denotes the colour and symbol the marker will take. If this sensor has not been visited the marker will be blank and grey, otherwise, it will have a custom colour and symbol based on the classification of its air-quality reading.

`String getGeojsonRoute(ArrayList<Point> route)`
This method parses the input ArrayList of Points *route* into code for a GeoJSON LineString. This is used to geographically represent the drone path. It iterates through each Point in the ArrayList and stores their coordinates in a list representing the points on the line.

`void writeToFile(String filePath, String fileContents)`
This method is used for writing to files. The *filePath* parameter represents the desired path (and name) of the output file, this path is appended to the path of the current working directory. The *fileContents* parameter represents the data to be written to the output file.

GeometricalCalcs

Methods:

`Double calcDistance(Point p1, Point p2)`
This method returns the Euclidean distance between the input Point objects.

`Double calcAngle(Point origin, Point dest)`
This method returns the angle from the *origin* Point to the *dest* Point.

`Point transformPoint(Point origin, Double angle)`
This method returns the resultant point when moving from the *origin* Point at an angle of *angle* degrees. This is calculated using simple planar trigonometry given we have a fixed path length.

MoveCalcs

Variables:

`Move lastMove`: variable used to store the last move made in the *findNextMove()* method.

`Point lastSensorPoint`: variable used to store the Point of the last destination sensor in the *findNextMove()* method.

Methods:

`Move findNextMove(Point currPoint, Point nextPoint)`
This method returns the next valid drone move given the current point of the drone and the point of the destination sensor. These calculations will be discussed in depth under 'Calculating moves' in the 'Drone control algorithm' section.

`Boolean isMoveRedundant(Move current, Point currSensorPoint)`
This method takes inputs for the current Move and current destination sensor Point. These inputs are compared with the class variables *classMove* and *lastSensorPoint* to check if the input move is redundant (opposite of last). A move is redundant if the destination sensor point is the same as last but the angle is opposite ($|angle_{input} - angle_{lastMove}| = 180$) . A full description of why this is needed can be found under 'Calculating a single move' in the 'Drone control algorithm' section.

`Boolean isPointInRange(Point destination, Point actual)`
This method checks if the input Point *actual* is in range of the other input Point *destination*. These points are in range of each other if their distance is less than the global constant *errorMargin*. Returns true if so, false otherwise.

`Double calcRouteCost(ArrayList<Sensor> sens)`

This returns the total distance of the route specified by the input ArrayList of Sensor objects.

`Double calcEdgeCost(Point origin, Point dest)`

This returns the real distance between the input Point objects. What I mean by a "real" distance is that if the path between these points is obstructed by a no-fly zone (checked using the *isPathValid()* method) then we calculate the distance of the path it would take the drone to fly around this no-fly zone and reach the *dest* Point. This is calculated by calling the *calcActualDist()* method which is explained below. If this path is not obstructed then the Euclidean distance between these points is returned.

`Double calcActualDist(Point origin, Point destination)`

This method is used to accurately calculate the distance the drone would actually have to fly from the *origin* Point to the *destination* Point. This is used to calculate the distance between points which are obstructed by no-fly zones. This method works by iterating through the *findNextMove()* method until we are in range of the *destination* Point. Distance is taken as the number of moves multiplied by the global constant *pathLength*.

`Boolean isPathValid(Point origin, Point dest)`

This method checks whether the path between the input points is valid. It does this by ensuring the *dest* Point is within the confinement area and checking that the path between the *origin* Point and the *dest* Point does not intersect any of the no-fly zone boundaries. Returns true if valid, false otherwise.

`Boolean checkNoFlyZones(Point p1, Point p2)`

This method checks if the path between the two input points intersects with any of the no-fly zone boundaries. It does this by iterating through each boundary for each no-fly zone and calling *isIntersection()* which returns whether the input path and the no-fly zone boundary intersect at all. If a single intersection is found this method returns false, otherwise it returns true.

`Boolean isIntersection(LineGraph path, LineGraph bound)`

This method checks if the input LineGraph objects intersect at all. This is calculated using simple coordinate geometry and will be explained in depth in the description of 'Avoiding no-fly zones and staying in the confinement area' in the 'Drone control algorithm' section.

Webserver

Variables:

`String wsURL`: represents the URL to connect to the web server at.

`final HttpClient client`: represents the Java HttpClient we use to connect to the web server. Made final as it is never adapted.

Methods:

`void initWebserver()`

This method tries to connect to the web server at the specified input argument port. If this connection fails the user is notified and the program is terminated.

`String getWebserverFile(String path)`

This method is used to retrieve data from the web server. It returns the String contents of the web server file at the path specified by the input *path*. If this fails, the user is notified, and the program is terminated.

Main Class (App.java)

Note that like the previous method classes **all** variables and methods here are static. This main class is where we keep all the input arguments, constants, and web server data, thus most variables here are made public as all classes need access to this data. The only private variables being *randomSeed* (as it is unused) and *sensorRoute* (as if it is needed by a function it is passed as an argument). All methods besides the main are private as they are not utilized in any other classes.

Variables:

`final double maxLat`, `final double minLat`, `final double maxLng`, `final double minLng`: these constants represent the boundaries of our confinement area.

`double errorMargin`: this represents the maximum distance away (exclusive) our drone has to be from a sensor in order to visit/read it.

`final double pathlength`: this constant represents the length each drone movement must be.

`ArrayList<NoFlyZone> noFlyZones`: this is used to store all the NoFlyZone objects.

`ArrayList<Sensor> sensors`: this is used to store all the Sensor objects for the given day.

`String dateDD`: this stores the day specified by a command line argument.

`String dateMM`: this stores the month specified by a command line argument.

`String dateYY`: this stores the year specified by a command line argument.

`Point startPoint`: this stores the starting point specified by the longitude and latitude command line arguments.

`int randomSeed`: this stores the random seed specified by a command line argument. I have not used this in any of my route optimisation algorithms, but kept it in case of further developers who might.

`String portNumber`: this stores the port number to connect to the web server at, specified by a command line argument.

`ArrayList<Sensor> sensorRoute`: this privately stores the optimized route of Sensor objects.

Methods:

`void checkDateIsValid(String day, String month, String year)`

This method checks the input date exists and repairs any input formatting issues. If the input date does not exist, the user is notified, and the program is terminated. This method then repairs any formatting of the input date by checking for any single character day or month inputs. We cannot use single characters here as all day and month directories on the web server use two characters, thus a "0" is appended onto these and the appropriate class variables are updated.

`Integer checkIsNumber(String date, String name)`

This method checks whether the input *date* is a integer. The *name* argument is used for identifying what the *date* input represents for effective error logging. If *date* is not an integer the user is notified and the program is terminated, otherwise this integer is returned.

`void getSensorData()`

This method retrieves and parses the sensor data from the web server for the given date and stores it in the class variable *sensors*.

`void getNoflyzoneData()`

This method retrieves and parses the no-fly zone data from the web server and stores it in the class variable *noFlyZones*.

`void findOptimalRoute()`

This method is used to find the optimal route between sensors. It first creates a Sensor object for the starting point of the drone which is then appended to the 'sensors' class variable. This is extremely important as it ensures the starting point is accounted for when mapping an optimal route. After this route optimisation algorithms are called, and the output routes are stored in the *sensorRoute* class variable. This structure is very useful as it allows us to easily swap in different sensor route optimisation algorithms if wanted/needed.

DRONE CONTROL ALGORITHM

Finding the optimal sensor route

In order to decide the optimal sensor route I decided to try lots of different algorithms to find what was optimal in this context.

Algorithms used

After experience with the travelling salesman problem I had a good idea of what algorithms I wanted to try use for this drone.

Initial route setting algorithms: these set a route based purely on distances between sensors.

- Greedy
  The Greedy algorithm works by iterating through the sensors the drone needs to visit and chooses a route based on which sensor is closest to the last.
  The route is initialized with the first sensor in the list. Thereafter route expansion is done by adding the next closest available sensor. This process is continued until a complete route is formed.

- Temperate (custom algorithm)
  I wanted to create an algorithm that finds optimal paths by prioritizing the sensors which have the highest mean distances from other sensors. This prioritization works by creating fragments/edges (represents the transition between two sensors) between the sensor with the highest mean distance and expanding it with its best possible path (the closest available sensor). This is iterated for all sensors from highest to lowest mean distance and placed in a priority queue (descending order of mean values). To prevent redundancy in this priority queue we only allow a given sensor to be in a maximum of two fragments (as a single sensor can be connected to a maximum of two other sensors). Once a priority queue of fragments/edges is found then the route can begin to be created. The route is initialized with the first fragment in the priority queue. Thereafter route expansion is based upon availability in the priority queue. By this I mean that based upon the given sensor we need to expand (last sensor in the current route), the algorithm first checks if this sensor can be found in a fragment in the priority queue. If so, the other sensor from this given fragment is added to the route. Otherwise, the algorithm finds the best available transition for the given sensor and adds it to the route. This process is continued until a complete route is formed. In order to ensure maximum efficiency redundant fragments (fragments which contain sensors that are not available) are deleted when found upon each iteration.
  This algorithm almost works in the opposite way to the greedy algorithm in which rather than prioritizing using the shortest distances possible, mine prioritizes **not** using the longest distances possible, this is why I decided to name this algorithm 'Temperate' (the antonym of greedy).

Route refinement algorithms: these naively switch points in the route and see if this improves the overall route cost (total distance travelled).

- Swap heuristic
  The Swap heuristic algorithm works by swapping adjacent sensors in the route to see if it

improves the cost.
This algorithm iterates through each element in the given route to try all possible adjacent swaps. If a single swap in the loop is successful (improves the overall cost) we iterate through the route again (because the route has been changed). If no successful swaps are made throughout an entire loop the algorithm is terminated.

- 2-Opt heuristic
  The 2-Opt heuristic algorithm works by flipping the path between two sensors in the route to see if it improves the cost.
  This algorithm uses a nested loop in order to get two indexes that represent sensors in the route. For every iteration we then try reverse the path between these two sensors. If a single reversal from the entire nested loop was successful (improves the overall cost) we iterate through the route again (because the route has changed). If no successful path reversals are made throughout an entire nested loop the algorithm is terminated.

**NOTE: All of these algorithm implementations can be found in *Algorithms.java* and can be easily swapped between by using the *findOptimalRoute*() function in *App.java*

Calculating distances between sensors

Calculating the distances between sensors accurately is a very critical part of producing an effective algorithm in this context. This is evident as all our route optimisation algorithms use distance to map a route with the smallest overall distance and thus minimal number of moves.

Typically we would just be able to measure the direct distance between two points, however, given our drone has to avoid no-fly zones this makes a direct distance not very reliable as there could be a no-fly zone between these points.

To solve this problem, when calculating the distance between sensors I would check if the path between these sensors intersected any of the no-fly zone boundaries. If so, I would then map out the actual distance the drone would have to follow to go around these no-fly zones. This was done by iterating *findMove()* until we were in range of the destination sensor, the resulting distance would just be the number of moves multiplied by the path length. Otherwise, if there was no obstruction between these sensors, we would just take the direct Euclidean distance.

This was an extremely effective factor in optimizing the route as it maximizes both the performance and accuracy of our algorithms.

Calculating moves

Given we want to minimize the total number of moves our drone takes, calculating moves efficiently is just as or even more important than finding an optimal sensor route. This is evident as our drone must be able to navigate its way between sensors in an optimal way to maximize efficiency whilst conforming to its constraints on movement length and direction.

Calculating a single move

This is all done in the *findNextMove()* method.

To calculate a single move, we must first know the current point and destination point (destination sensor coordinates) of the drone. With these points we will then be able to calculate the angle between our drone and the destination sensor using planar trigonometry:

$$angle = \tan^{-1}(m_{AB}) \quad , \qquad where \ m_{AB} = gradient \ between \ points \ A \ \& \ B$$

Although this angle is most likely not in degrees of 10 it allows us to find the next best options. Given this movement direction constraint we realize our drone must zigzag towards the sensor. To do this we can just take the nearest angles on either side of the real angle:

$$newAngle = (realAngle - (realAngle \ \% \ 10)) \pm 10$$

We can then use these two angle options to transform the current point and thus find the two possible next points. If an invalid point is found (outside confinement area or crossing a no-fly zone) the angle is changed until the point is valid. We then choose our move based on which point is closest to the destination point (given that this move is not redundant; discussed below).

A problem with this approach is when our drone needs to go around a no-fly zone. This is because it is likely our drone will have to move further away from the destination sensor to get around the given no-fly zone. This would evidently not work given the way our algorithm chooses the next move (based on which point is closest to the destination sensor) and thus would end up getting caught in an infinite loop of redundant moves. To prevent this, after every call of *findNextMove()* we store the move and destination sensor in global variables. Then when choosing which point is best we first check each of the moves are not redundant using the *isMoveRedundant()* function, and then compare distances. The *isMoveRedundant()* function works by checking if the angle of the input move is opposite of the last move ($|angle_{input} - angle_{lastMove}| = 180$) and if the input destination sensor is the same as the last destination sensor. If these are both true, then we know this input move is redundant, and thereby we would choose the other available move in *findMove()*.

Avoiding no-fly zones and staying in the confinement area

In order to avoid no-fly zones and stay in the confinement area when calculating a move in *findMove()* I created the *isPathValid()* method. This method takes the current point and the next point as inputs.

Using the *checkConfinement()* method it first checks if the 'next point' is within the confinement area by ensuring its coordinates are within a range of latitudes and longitudes defined by the global constants *maxLat*, *minLat*, *maxLng*, and *minLng*.

Next, using the *checkNoFlyZones()* method it checks if the path from the current point to the next point intersects any of the buildings/no-fly zones. It does this by iterating through each boundary for all of the no-fly zones and sees if the drone path intersects any of these boundaries using the *isIntersection()* function. If an intersection is found the loop is broken and the path is returned as invalid (false).

This *isIntersection()* function works by expressing this drone path and the given boundary of the no-fly zone as straight-line functions. Then using simple coordinate geometry, we can see if these lines intersect at a point in the range of the given boundary:

isIntersection(): calculating an intersection between the LineGraph *path* and LineGraph *bound* inputs

Case 1: if *path* is a vertical line

1. Check if the range of longitude values for *bound* includes *path*'s longitude value.
2. Check if the range of latitude values for *bound* overlaps *path*'s range of latitude values.

Case 2: If *bound* is a vertical line

1. Check if the range of longitude values for *path* includes *bound*'s longitude value.
2. Check if the range of latitude values for *path* overlaps *bound*'s range of latitude values.

<u>Case 3:</u> <u>If the net gradient is zero (meaning path and bound are not parallel)</u>
We then calculate the coordinates of intersection

1. Check if the intercept longitude/latitude value lies in both *bound*'s and *path*'s range of longitudes/latitudes.

<u>Case 4:</u> <u>If the straight-line functions described by *path* and *bound* are equivalent</u>

1. Check if *path*'s range of latitude values overlaps with *bound*'s range of latitude values.

**If all the checks for a given case pass, then there is an intersection between path and bound. Otherwise, there is no intersection.

Algorithm testing

	Performance Metrics (over 731 maps)					
Algorithms	**Avg. # moves**	**Best # moves**	**Worst # moves**	**Standard deviation**	**Avg. execution time (s)**	**Time complexity**
Swap	149.42	89	150	NA	0.108071135	**O(n)**
2-Opt	95.48	45	150	10.54	0.236662107	**$O(n^2)$**
Swap -> 2-Opt	94.73	48	113	8.55	0.29001368	**$O(n^2 + n)$**
Greedy	101.93	63	150	11.56	**0.017783858**	**$O(n^2)$**
Greedy -> Swap	99.42	53	121	10.43	0.02872777	**$O(n^2 + n)$**
Greedy -> 2-Opt	88.83	48	**102**	**7.03**	0.149110807	**$O(n^2)$**
Greedy -> Swap -> 2-Opt	88.90	50	104	7.09	0.151846785	**$O(n^2 + n)$**
Temperate	103.81	54	137	11.68	0.04377565	**$O(n^2 + n)$**
Temperate -> Swap	101.54	52	133	11.8	0.058823529	**$O(n^2 + n)$**
Temperate -> 2-Opt	**88.75**	**47**	107	7.54	0.17373461	**$O(n^2 + n)$**
Temperate -> Swap -> 2-Opt	89.00	**47**	107	7.09	0.177838577	**$O(n^2 + n)$**

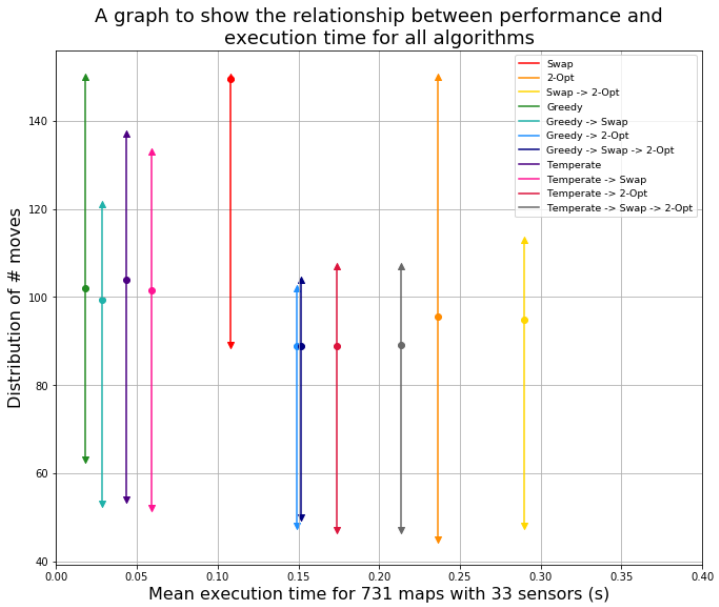
Note these arrows (->) represent sequential executions of the different algorithms. In which the next one takes the optimized route from the previous.

The green boxes are used to highlight the best value for each metric.

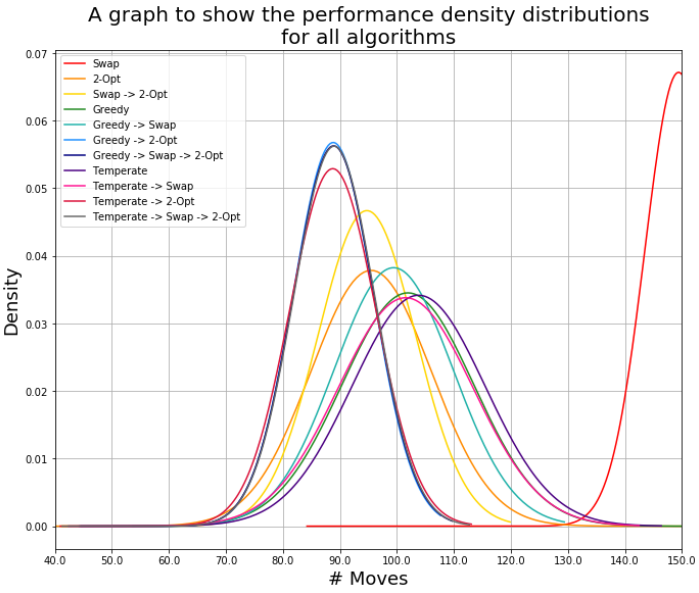
Optimal algorithm for performance, execution time and scalability

I am very happy with the results of my algorithms as they provide many effective solutions with varying trade-offs.

Although, the Temperate -> Swap -> 2-Opt option provides the best average number of moves, in the context of scalability this algorithm is less favourable given its high worst-case time complexity. This would be problematic when mapping a route between sensors across the whole of Edinburgh. Given the area of Edinburgh is roughly 264km² and



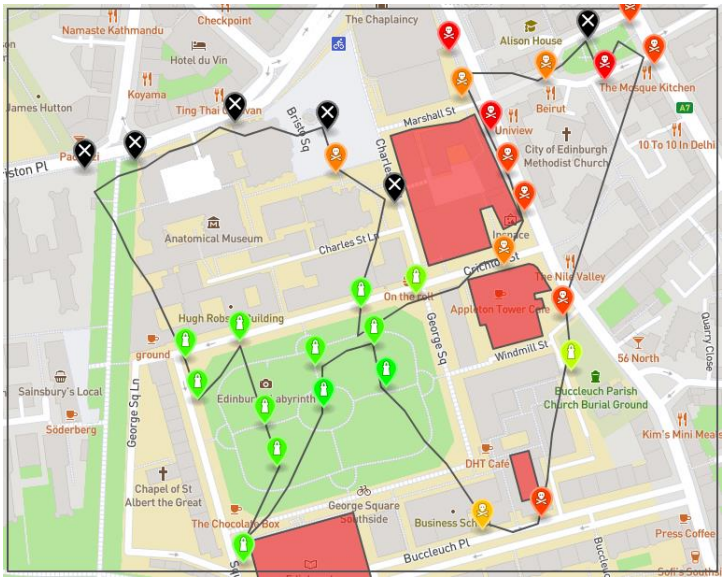
that the density of sensors would be the same as that for our current confinement area we can estimate there would be almost 30000 sensors needed to be visited. This is evidently a massive jump from 33 and would exponentially increase the execution time of our algorithm. So, choosing an algorithm with the right balance of performance and time complexity is extremely important. We must also note that good performance denotes less moves to calculate meaning less execution time.



Furthermore, I believe consistency is very important when choosing an algorithm as it allows for more predictable results. This consistency is measured by the standard deviation of performance, the lower the standard deviation the more consistent the results are.

Given these factors, I believe in this context the Greedy -> 2-Opt algorithm would be the most effective to use. This is evident as it has the lowest standard deviation (excluding Swap given this result is due to poor performance), the second best minimum number of moves achieved, the best maximum number of moves achieved, is only 0.08 moves off the best average, and has the best worst-case time complexity for all the algorithms with sub 100 move averages.

Example map outputs



18-01-2020: 73 moves

06-01-2020: 82 moves

REFERENCES

- INF2: Introduction to Algorithms & Data Structures, Coursework 3, 13th March 2020 Understanding and experience with Swap heuristic, 2-Opt heuristic, and Greedy algorithms through implementation of the travelling salesman problem.