Informatics Large Practical Report

1. Software Architecture Description

My application is made up of the following classes: *ReadWebServer*, *SensorDetails*, *SensorLocation*, *MarkerProperties*, *NoFlyZones*, *DroneConstraints*, *DroneMovement*, *WriteFlightpath*, *WriteReadings* and *App*. Each class extends the class preceding it, displaying multi-level inheritance. The order was chosen in such a way so that it reflects the pipeline my application. These classes can be grouped into 5 overlapping categories:

1. *ReadWebServer* deals with retrieving the content from the webserver
2. *SensorDetails*, *SensorLocation*, *MarkerProperties* and *NoFlyZones* receive and/or manipulate the data from the webserver, in order to make it useful for my application.
3. *NoFlyZones*, *DroneConstraints* and *DroneMovement* deal with the flightpath of the drone
4. *WriteFlightpath* and *WriteReadings* make use of the previous classes to produce the output files
5. *App* calls the methods in 4. in order to produce the files

ReadWebServer is made up of one method which is needed by some of its sub classes, namely *SensorDetails*, *SensorLocation* and *NoFlyZones*. The class *SensorDetails* collects the data related to the details of all the sensors visited on a given day, which includes their location, battery and reading. Furthermore, *SensorLocation* gets the location of each sensor in terms of coordinates, given its location in terms of What3Words. Since having the locations of the sensors to be visited on a given day in the form of coordinates is useful, using methods from *SensorDetails* in *SensorLocations* makes this possible. Similarly, in order for the markers on the map to have the specified properties, it has to look at the reading and battery of each sensor, making it useful that it extends *SensorLocation*. Lastly, *NoFlyZones* retrieves the coordinates that define the no-fly zones of the drone, allowing the drone to avoid them. Some of its methods therefore become useful in *DroneConstraints*, which ensures the drone's movements are legal during its flightpath. *DroneMovement* in turn makes use of methods from previous classes in order to visit sensors, while satisfying the constraints defined in *DroneConstraints*. Finally, all classes combined allows the application to produce the file documenting the drone's flight and the file producing a visualisation of this flight.

In each of the super classes, the visibility modifiers of the methods and variables which are inherited by the sub classes are set to public. On the other hand, the methods which are used as helper functions, as well as the global variables, in a specific class are set to private. This makes the code more readable, as someone can immediately identify the methods which directly contribute to the final output. Using several helper functions also contributes to the readability of the code, as this breaks down big tasks into smaller task that are more interpretable and therefore easier to process by someone. Similarly, using several classes instead of one big class has the same effect.

2. Class documentation

I. *ReadWebServer*

Method name: `webServerContent`
Input: String urlSegment, String port
Output type: String
Description: Given the *port* to receive the content from and the non-constant part of the URL of the webserver, i.e. everything after "http://localhost:*port*", the method makes an HTTP request, sends it to an HTTP client and receives an HTTP response. In the case that the request is not successful, an error message is printed and the application is exited. In the case that it is successful, the content of the webserver is returned as a String.

II. *SensorDetails*

Method name: `getSensors`
Input: String port, String year, String month, String day
Output type: List<SensorDetails>
Description: Given the port to receive the content from and the date specified by day, month and year, the method uses `webServerContent` to receive a JSON list as String and then deserialises this list of many SensorDetails into an ArrayList<SensorDetails>. It does this by creating a SensorDetails object, with properties location, battery and reading. The List<SensorDetails> returned represents the details of the sensors to be visited on the given date.

Method name: `location`
Input: SensorDetails details
Output type: String
Description: Given the details defining a sensor, the property location of this object is returned.

Method name: `getSensorWords`
Input: SensorDetails details
Output type: List<String>
Description: Given the details defining a sensor, it returns a list of the 3 words defining the location of the sensor.

Method name: `get3WordsList`
Input: String port, String year, String month, String day
Output type: List<List<String>>
Description: Given the port to receive the content from and the date specified by day, month and year, the method uses `getSensors` and `getSensorWords` to return a list of lists, where each list contains the 3 words defining the location of a sensor to be visited on the given date.

Method name: `getBatteryList`
Input: String port, String year, String month, String day
Output type: List<Double>
Description: Given the port to receive the content from and the date specified by day, month and year, the method uses `getSensors` and returns a list of the battery values of the sensors to be visited on the given date.

Method name: `getReadingList`
Input: String port, String year, String month, String day
Output type: List<String>
Description: Given the port to receive the content from and the date specified by day, month and year, the method uses `getSensors` and returns a list of the readings of the sensors to be visited on the given date.

III. *SensorLocation*

Method name: `getCoords`
Input: String port, String firstWord, String secondWord, String thirdWord
Output type: Coordinates
Description: Given the port to receive the content from and the location specified by the three words, the method firstly uses `webServerContent` to receive a JSON String. It then uses this to create a SensorLocation object, with property coordinates, which is a Coordinates object which has properties lat and lng. The Coordinates returned represents the coordinates of a sensor defined by the given three words.

Method name: `coordsAsPoint`
Input: String port, String firstWord, String secondWord, String thirdWord
Output type: Point
Description: Given the port to receive the content from and the location specified by the three words, the method firstly uses `getCoords`. Using the properties lng and lat, it then converts the Coordinates of a sensor into type Point.

Method name: `getSensorsCoords`
Input: String port, String year, String month, String day
Output type: List<Point>
Description: Given the port to receive the content from and the date specified by day, month and year, the method firstly uses `get3WordsList`. Then by using `coordsAsPoint`, a list of the coordinates of all the sensors to be visited on the given date is returned.

Method name: `pointToLocation`
Input: String port, String year, String month, String day
Output type: HashMap<Point, String>
Description: Given the port to receive the content from and the date specified by day, month and year, the method uses `getSensorsCoords` and `getSensors` to create a mapping of the sensor Point positions to their String locations. This mapping is then returned.

Method name: `pointToBattery`
Input: String port, String year, String month, String day
Output type: HashMap<Point, Double>
Description: Given the port to receive the content from and the date specified by day, month and year, the method uses `getSensorsCoords` and `getBetteryList` to create a mapping of the sensor Point positions to their battery values. This mapping is then returned.

Method name: `pointToReading`
Input: String port, String year, String month, String day
Output type: HashMap<Point, String>
Description: Given the port to receive the content from and the date specified by day, month and year, the method uses `getSensorsCoords` and `getReadingList` to create a mapping of the sensor Point positions to their readings. This mapping is then returned.

IV. *MarkerProperties*

Method name: `getColour`
Input: String sensorReading, double sensorBattery
Output type: String
Description: Given the sensor reading and battery, the method returns the corresponding RGB string (according to Figure 5).

Method name: `getSymbol`
Input: String sensorReading, double sensorBattery
Output type: String
Description: Given the sensor reading and battery, the method returns the corresponding marker symbol (according to Figure 5).

V. *NoFlyZones*

Method name: `getGradient`
Input: Point p1, Point p2
Output type: double
Description: Given two Points, the method returns the gradient of the line joining them.

Method name: `getYint`
Input: Point p1, Point p2
Output type: double
Description: Given two Points, the method uses `getGradient` and p1 to compute the y-intercept of the line joining them. The y-intercept is then returned.

Method name: `getIntersectionX`
Input: double droneYint, double buildingSideYint, double droneGrad, double buildingSideGrad Output type: double
Description: Given the y-intercepts and gradients of the line joining the drone's current and next position and the line representing one side of a building, the method returns the x coordinate (longitude) where these two lines intersect. The equation used to get this x is obtained by equating $y=m_1x+c_1$ and $y=m_2x+c_2$ and solving for x.

Method name: `getIntersectionY`
Input: double droneYint, double buildingSideYint, double droneGrad, double buildingSideGrad
Output type: double
Description: Given the y-intercepts and gradients of the line joining the drone's current and next position and the line representing one side of a building, the method returns the y coordinate (latitude) where these two lines intersect. It firstly uses `getIntersectionX` and then substitutes this into the equation y=mx+c, where m=droneGrad and c=droneYint.

Method name: `getIntersection`
Input: double droneYint, double buildingSideYint, double droneGrad, double buildingSideGrad
Output type: Point
Description: Given the y-intercepts and gradients of the line joining the drone's current and next position and the line representing one side of a building, the method uses `getIntersectionX` and `getIntersectionY` and returns the Point where the drone path intersects with a side of a building.

Method name: `noIntersection`
Input: Point droneCurr, Point droneNext, Point coord1, Point coord2
Output type: boolean
Description: Given the drone's current and next position and two coordinates (coord1 and coord2) defining a side of a building in the no fly zone, the method returns true if the line joining the first two points does not intersect with the line joining the next two points. It does this by firstly using `getGradient` and `getYint` on droneCurr and droneNext as well as coord1 and coord2. Then there are three scenarios to check:

1. In the case that the lines have the same gradient and different y-intercept, the lines are parallel and therefore do not intersect.
2. If the lines fall onto each other, then the lines intersect (at finitely many points). In order to see if the lines fall onto each other, firstly the method checks if the two lines have the same gradient and y-intercept. If this is true, it then compares the longitudes of the drone path and side of the building. If

the drone starts off the building side, but during its flight to its next position its path lies on the side of the building, then the lines intersect.

3. If the x coordinate of the point at which the two lines intersect is within the range of longitude values of the drone path and building side, then the lines intersect. This x coordinate is obtained using `getIntersectionX`.

Method name: `noFlyZoneCoords`
Input: String port
Output type: List<List<Point>>
Description: Given the port to receive the content from, the method firstly uses `webServerContent` to receive a JSON String. It uses this to create a list of lists of Points, where each list contains the Points defining one no-fly zone (one building). This list is then returned.

Method name: `noFlyZoneCoordPairs`
Input: String port
Output type: List<List<List<Point>>>
Description: Given the port to receive the content from, the method firstly uses `noFlyZoneCoords`. Using this, it creates a list of lists, where each list contains the Points defining each no-fly zone. The Points are in pairs (a list of two Points), each pair representing the two coordinates defining one side of that building. This list is then returned.

Method name: `linesWhichIntersect`
Input: Point droneCurr, Point droneNext, List<List<List<Point>>> noFlyZonesCoords
Output type: List<Point>
Description: Given the drone's current and next position, and the list of pairs of coordinates defining each no-fly zone, the method uses `noIntersection` to check if the line joining the drone's current and next position intersects with any line joining two coordinates defining a side of a building in the no fly zone. In this case that it does, a list of the coordinates defining those sides is returned.

Method name: `noIntersections`
Input: Point droneCurr, Point droneNext, List<List<List<Point>>> noFlyZonesCoords
Output type: boolean
Description: Given the drone's current and next position, and the list of pairs of coordinates defining each no-fly zone, the method returns true if the line joining the drone's current and next position does not intersect with any line joining two coordinates defining a side of a building in the no fly zone. It does this by using `noIntersection`.

Method name: `buildingIndex`
Input: Point coord1, String port
Output type: int
Description: Given the port to receive the content from and one of the coordinates defining a building, the method uses `noFlyZoneCoords` to return the index of the building in the no-fly zones list which contains this point.

Method name: `sumPoints`
Input: Point point1, Point point2
Output type: Point
Description: Given two Points, it returns the sum of them.

Method name: `averagePoint`
Input: Point pointsSummed, int numCoordsInZone
Output type: Point
Description: Given the summation total of the points and number of coordinates which were summed, the method returns the average point.

Method name: `buildingCentres`
Input: String port
Output type: List<Point>
Description: Given the port to receive the content from, the method firstly uses `noFlyZoneCoords`. It then uses this list to sum all the points in each no-fly zone, using `sumPoints`. Once the summation total of the points in a particular no-fly zone is obtained, it uses `averagePoint` to get the centre of mass of the building. A list of the centres of all buildings is returned.

VI. *DroneConstraints*

Method name: `withinArea`
Input: Point dronePos
Output type: boolean
Description: Given the position of the drone, the method returns true if the drone is within the confinement area.

Method name: `areaCentre`
Input: -
Output type: Point
Description: Returns the centre of the confinement area.

Method name: `multipleOfTen`
Input: double direction
Output type: boolean
Description: Given the direction (angle), the method returns true if it is a multiple of 10 i.e. the modulo 10 of it is 0.

Method name: `withinDirRange`
Input: double direction
Output type: boolean
Description: Given the direction (angle), the method returns true if it is within the correct range

Method name: `validDir`
Input: double direction
Output type: boolean
Description: Given the direction (angle), the method returns true if both `multipleOfTen` and `withinDirRange` return true, and therefore the direction is valid.

Method name: `nextPos`
Input: int dirInDeg, Point currPos
Output type: Point
Description: Given the current position of the drone and its direction of travel, the method returns the new position of the drone.

Method name: `computeDir`
Input: Point dronePos, Point desPos
Output type: int
Description: Given the position of the drone and the desired position it wants to move towards, the method returns a direction for the drone to move in such that it approaches this desired position. The desired position of the drone is either in range of: the closest sensor or its initial position (such that we have a closed loop path). In order to determine this direction, the method firstly finds the angle between the line joining the drone's current and desired position, and the positive horizontal. It then checks the properties of this angle and adjusts it accordingly. The scenarios that are checked are as follows:

1. If `multipleOfTen` is false and `withinDirRange` is true, the angle is rounded down to the nearest 10.

2. If `multipleOfTen` is true and `withinDirRange` is false, the angle either has a magnitude greater than 360, is negative, or both. The first case is dealt with by computing the modulus 360 of the angle, in order to convert the angle to be between -360 and 360. The second scenario is dealt with by adding 360 to this angle, such that the negative angle is converted into its equivalent positive angle. The third case is dealt with by applying both.
3. If `validDir` is false, the steps followed in 2. and then 1. are executed.

In the case that none of these conditions are satisfied, this means the direction is valid and therefore this angle is returned without making any adjustments.

Method name: `possibleNextPos`
Input: Point currPos, Point coord1, Point coord2, double buildingSideGrad, Point buildingCentre
Output type: List<Point>
Description: The method is given the current position of the drone, the two coordinates defining the side of the building the drone's path intersects with, the gradient of this side and the centre of this building. Using these, it returns a list of the two possible next positions the drone can have, without intersecting with the building. The two possibilities comes from the fact that the drone can either fly in the direction from: coord1 to coord2, or coord2 to coord1. Using `computeDir` followed by `nextPos`, the next position of the drone is computed for each of these directions. Since these directions are never valid directions for the drone to fly in, the angles are rounded down, which in some cases results in the drone still intersecting with the building side. The method deals with these cases, by adding 10 degrees to these directions.

Method name: `euclidDist`
Input: Point p1, Point p2
Output type: double
Description: Given two points, the method calculates and returns the Euclidean distance between them.

Method name: `chooseNextPos`
Input: Point currPos, Point droneNextDes, Point coord1, Point coord2, double buildingSideGrad, Point buildingCentre, List<List<List<Point>>> noFlyZonesCoords, Point previous1, Point previous2
Output type: Point
Description: The method is given the current and next desired position of the drone, the two coordinates defining the side of the building the drone's path intersects with, the gradient of this side, the centre of this building, the list of pairs of coordinates defining each no-fly zone and the previous two positions of the drone. Using these, it returns the next position of the drone, such that it does not fly over the side of the building defined by coord1 and coord2. The method uses `possibleNextPos` and then proceeds to choose between the two possible positions. It does this by firstly using `noIntersections` on each of these positions and proceeds as follows:

1. In the case that `noIntersections` is true for both possible positions, the position chosen is the one that minimises the distance (uses `euclidDist`) to the drone's desired next position and does not result in a repeated movement.
2. If `noIntersections` is true for only one of the two possible positions, it chooses that position.

Method name: `avoidNoFlyZones`
Input: Point droneCurr, Point droneNextDes, List<List<List<Point>>> noFlyZonesCoords, List<Point> dronePositions, String port
Output type: Point
Description: The method is given the current and next desired position of the drone, the list of pairs of coordinates defining each no-fly zone, the drone's positions so far and the port to receive the content from. Using these, it returns the new position of the drone such that it avoids the no-fly zones. It does this by firstly using `linesWhichIntersect` and proceeds depending on how many coordinates the list returned contains. The list size can take the values 0, 2, 4 and 6. The different sizes can be interpreted as follows: if the list is of size 2, 4 and 6, this indicates the drone path intersects with 1, 2 and 3 building sides, respectively.
In the case that there are 0 intersections, the drone's desired next position is returned. In the case that there is 1 intersection, `buildingCentres` is used, and the correct building centre is obtained using

`buildingIndex`. Next, `getGradient` is used on the pair of coordinates defining the building side the drone's path intersects with. These values are then used in `chooseNextPos`, and this next position is returned. In the two remaining cases the method proceeds similarly, as follows:

1. `getGradient` and `getYint` are used on the points given by the drone's current position and its next desired position.
2. For each pair of coordinates defining a building side the drone's path intersects with, `getGradient` and `getYint` are used.
3. Using the values obtained in 1. and 2., `getIntersection` is used for each pair of points.
4. `euclidDist` is used on the points given by the drone's current position and each of the points of intersection in 3.
5. The pair of coordinates that result in the smallest distance in 4. correspond to the building side the drone's path intersects with first. This is the direction that needs adjusting by `chooseNextPos`.
6. In order to use `chooseNextPos`, `buildingCentres` and `buildingIndex` need to be used as above. This next position is returned.

Method name: `avoidLeavingArea`
Input: Point droneCurr, Point droneNextDes, List<List<List<Point>>> noFlyZonesCoords, List<Point> dronePositions, String port
Output type: Point
Description: The method is given the current and next desired position of the drone, the list of pairs of coordinates defining each no-fly zone, the drone's positions so far and the port to receive the content from. Using these, it returns the new position of the drone, in the case that the desired next position results in the drone leaving the confined area. It does this by moving towards the centre of the confined area, given by `areaCentre`, using `computeDir` and `nextPos`. At the same time, it avoids the no-fly zones, using `avoidNoFlyZones`.

Method name: `avoidIllegalMove`
Input: Point droneCurr, Point droneNextDes, List<List<List<Point>>> noFlyZonesCoords, List<Point> dronePositions, String port
Output type: Point
Description: The method is given the current and next desired position of the drone, the list of pairs of coordinates defining each no-fly zone, the drone's positions so far and the port to receive the content from. Using these, it returns the new position of the drone, which is legal. It does this by using `avoidNoFlyZones` followed by `avoidLeavingArea`.

VII. *DroneMovement*

Method name: `getMaxMoves`
Input: -
Output type: int
Description: Returns the maximum number of moves the drone can make.

Method name: `incrementMoves`
Input: -
Output type: -
Description: Increases the number of moves the drone has made by 1.

Method name: `getMoves`
Input: -
Output type: int
Description: Returns the number of moves the drone has made.

Method name: `withinRange`
Input: Point dronePos, Point sensorPos
Output type: boolean
Description: Given the positions of the drone and a sensor, this method returns true if the drone is within range to connect to the sensor.

Method name: `isClosedLoop`
Input: Point initPos, Point currPos
Output type: boolean
Description: Given the initial and current position of the drone, this method returns true if two positions are close to each other and therefore the drone's flightpath is a closed loop.

Method name: `closestSensor`
Input: Point dronePos, List<Point> sensors
Output type: Point
Description: Given the drone's position and the list of sensors to be visited, this method returns the closest sensor to the drone.

Method name: `dronePath`
Input: String day, String month, String year, String latStr, String lonStr, String port
Output type: List<List<Point>>
Description: The method is given the date specified by day, month and year, the initial position of the drone specified by latStr and lonStr, and the port to receive the content from, which are the command-line arguments used. Using these, it returns a list of two lists, where the first list contains all the drone's positions and the second contains all the sensors visited, during its flight. It does this by firstly using `getSensorsCoords`. Then as long as `getMaxMoves` and `getMoves` are not equal:

1. `closestSensor` is used.
2. `computeDir` is used on the point defining the drone's current position and the closest sensor, to find the desired direction for the drone to fly in.
3. Using the direction in 2., the desired position of the drone is found using `nextPos`.
4. Using the position in 3. and the output of `noFlyZoneCoordPairs`, `avoidIllegalMove` is used.
5. `incrementMoves` is used.
6. This new position is added to the list of the drone's positions.
7. `closestSensor` is used again.
8. If `withinRange` is true, this sensor found in 7. is added to the list of the sensors visited.
9. Steps 1.-8. are followed until all sensors are visited.
10. When all sensors are visited, the method proceeds to check if `isClosedLoop` is true. If it is, the two lists are added to a list and are returned.
11. If `isClosedLoop` is false, steps 2.-6. are followed, but this time with the initial position of the drone instead of the closest sensor. This is so the drone moves towards its initial position.
12. The steps in 10. are followed until the drone `isClosedLoop` is true.

Method name: `chosenDirections`
Input: List<Point> dronePositions
Output type: List<Integer>
Description: Given a list of the drone's positions during its flight, the method returns a list of directions the drone chose to move in.

Method name: `sensorsNotVisited`
Input: String port, String day, String month, String year, List<Point> sensorsVisited
Output type: List<Point>
Description: Given the port to receive the content from, the date specified by day, month and year, and a list of the sensors visited by the drone during its flight, the method returns a list of the sensors not visited by the drone. It does this by comparing the inputted list with the list of all sensors from `getSensorsCoords`.

VIII. *WriteFlightpath*

Method name: `writeTxtFile`
Input: String day, String month, String year, String lat, String lon, String port
Output type: -
Description: The method is given the date specified by day, month and year, the initial position of the drone specified by latStr and lonStr, and the port to receive the content from, which are the command-line arguments used. Using these, it creates a flightpath-dd-mm-yyyy.txt file, which lists where the drone has been and which sensors it connected to. It does this by firstly making use of `dronePath`. Using the list the drone's positions during its flight, `chosenDirections` is used. Next, using the list of sensors the drone visited and `pointToLocation`, the string location of each sensor is obtained.

IX. *WriteReadings*

Method name: `createMarkerFeature`
Input: Point sensorPos, String location, String colour, String symbol
Output type: Feature
Description: Given the sensor position, its string location, and the colour and symbol of the marker representing this sensor should have, this method returns a Feature consisting of the Point geometry with properties "location", "rgb-string", "marker-color" and "marker-symbol".

Method name: `createFeatureCollection`
Input: String day, String month, String year, String lat, String lon, String port
Output type: FeatureCollection
Description: The method is given the date specified by day, month and year, the initial position of the drone specified by latStr and lonStr, and the port to receive the content from, which are the command-line arguments used. Using these, it returns a Feature Collection of the drone's flight. It does this by firstly using `dronePath` to obtain the two lists. Using the list of the drone's positions during its flight, a Line String geometry is created to show the drone's path. Using the list of sensors the drone visited, for each sensor, `pointToBattery` and `pointToReading` are used such that `getColour` and `getSymbol` can be used. Using the list of sensors the drone visited, for each sensor, `pointToLocation` is also used. Then, using these values, `createMarkerFeature` is used. Using `sensorsNotVisited`, for each sensor, `pointToLocation` followed by `createMarkerFeature` is used. A Feature Collection is then created using all the Features created above.

Method name: `writeGeojsonFile`
Input: String day, String month, String year, String lat, String lon, String port
Output type: -
Description: The method is given the date specified by day, month and year, the initial position of the drone specified by latStr and lonStr, and the port to receive the content from, which are the command-line arguments used. Using these, it creates a readings-dd-mm-yyyy.geojson file of the drone's flight. It does this by making use of `createFeatureCollection`.

X. *App*

This class calls the methods `writeTxtFile` and `writeGeojsonFile` in order to generate the output files.

3. Drone control algorithm

The drone starts off at the point provided by the command line argument. By searching through the list of all sensors to be visited on the day provided, the algorithm finds the closest sensor. The angle between the positive horizontal, and the line joining the current position of the drone and the closest sensor is then found. If this angle is not a valid angle to be used for the drone, i.e. if it is not a multiple of 10 and is not within the range 0 to 350, the angle is adjusted as follows:

1. If the angle is not a multiple of 10 but is within the correct range, the angle is rounded down to the nearest 10.
2. If the angle is a multiple of 10 but is not within the correct range, the angle either has a magnitude greater than 360, is negative, or both. The first case is dealt with by computing the modulus 360 of the angle, in order to convert the angle to be between -360 and 360. The second scenario is dealt with by adding 360 to this angle, such that the negative angle is converted into its equivalent positive angle. The third case is dealt with by applying both.
3. If the angle is neither a multiple of 10, nor within the correct range, the steps followed in 2. and then 1. are executed.

Using this angle, the algorithm then uses it to estimate the drone's desired next position. This is the ideal next position for the drone to have as this results in it approaching the closest sensor identified earlier. However, since this next position might result in the drone flying over a no-fly zone or leaving the confinement area, this cannot be the finalised next position of the drone yet.

The algorithm therefore proceeds by checking if this next position results in the drone flying over a no-fly zone. It does this by checking whether the line from the drone's current position to the drone's desired next position intersects with any line defining a side of a building in the no-fly zone. These two lines are said to intersect if they cross over at a single point, or if the lines lie on each other. However, if there are intersections, due to the drone only being able to fly in a finite range of directions, the two lines never lie on each other.

In the case that there are no intersections, the algorithm proceeds by checking if this desired next position results in the drone leaving the confinement area. In the case that there are intersections, which could be from 1 to 3 intersections, the coordinates of the line it intersects with first is used to adjust the drone's movement. These two coordinates are used to re-compute the direction for the drone to fly in, such that it flies almost parallel to the side of the building it would have intersected with if it flew to the desired next position. Since no side of a building defining a no-fly zone is a multiple of 10, it is impossible for the drone to fly parallel to them. Instead, the drone can only fly almost parallel to them, by flying at a direction which is the next smallest multiple of 10. Because of this, there will be some situations where this rounding down will result in the drone still intersecting with that side of the building. These are illustrated in Figure 1 below. The algorithm takes care of these situations by increasing the direction by 10 degrees. It identifies each of these scenarios by looking at the drone's position relative to the side to the building, as well as the gradient of the side of the building. In the case that the gradient has a magnitude less than 1, in the problematic cases, the drone is either at the top or bottom of the building. Therefore the latitude value of the drone's position relative to the latitude value of the centre of the building is looked at. Similarly, if the gradient has a magnitude greater than or equal to 1, the drone is either to the left or right of the building. Therefore the longitude value of the drone's position relative to the longitude value of the centre of the building is looked at. Since the drone can also fly in two directions given any side of a building, the direction chosen is the one that minimises the distance to the drone's desired next position, and also does not result in the drone continuing to go back and forth in a loop. This direction is then used to calculate the drone's new desired next position, which now avoids no-fly zones.

Since this desired next position can still result in an illegal move, by the drone leaving the confinement area, this is now checked. In the case that it does not leave the confinement area, this next position is the finalised next position of the drone. In the case that it does however, the drone moves towards the centre of the confinement area, while avoiding the no-fly zones as earlier. Then this next position is the finalised next position of the drone.

Now that the drone is in a new position, the algorithm finds the closest sensor to it and checks if it is within range of it to collect readings. If it is, then the readings are collected and that sensor is now considered to have been visited by the drone. All the steps above are repeated until all the sensors have been visited.

When all sensors have been visited, the drone then proceeds to move back to its initial position. It does this similarly to how the drone approaches the closest sensor. The drone finds the direction to move in that results in it approaching its initial point, while avoiding the no-fly zones and leaving the confinement area. Once the drone is close to its initial position, the algorithm reaches an end. However, the algorithm can terminate at any point if the drone reaches its maximum number of moves.

Figure 1: Indicating the directions which will be affected by rounding down the angle defining the side of a building to the nearest 10, on a building from the no-fly zones.

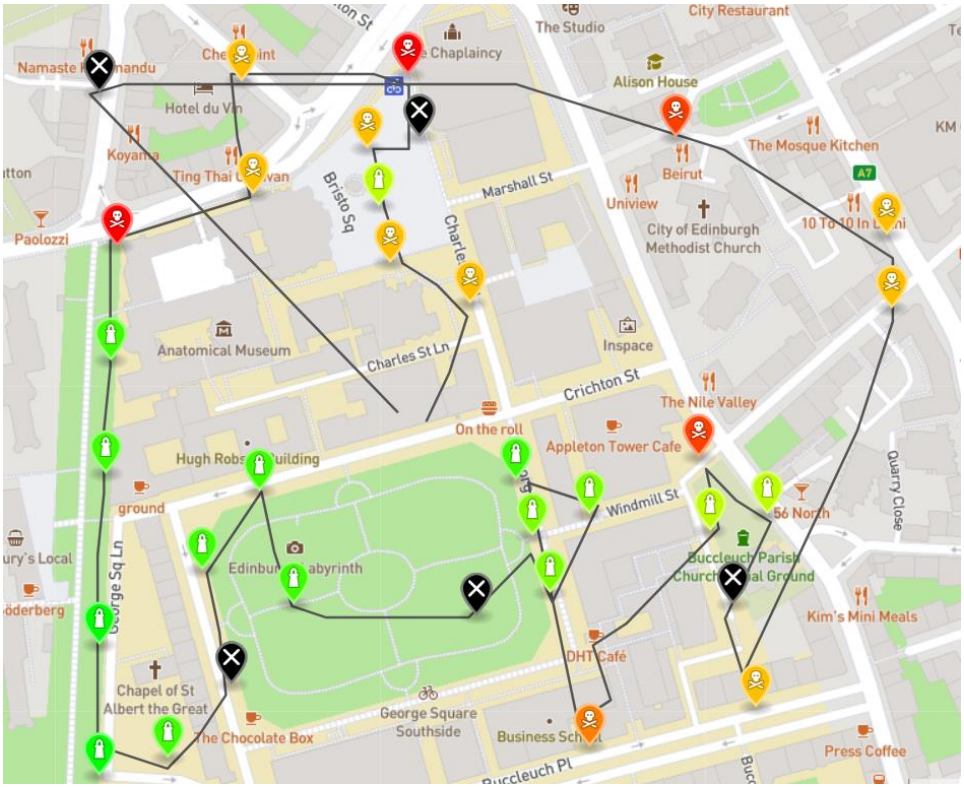

Figure 2: A sample output map for the date 10/04/2020.

Figure 3: A sample output map for the date 15/05/2021.