INFR09051
Informatics Large Practical

Coursework 2

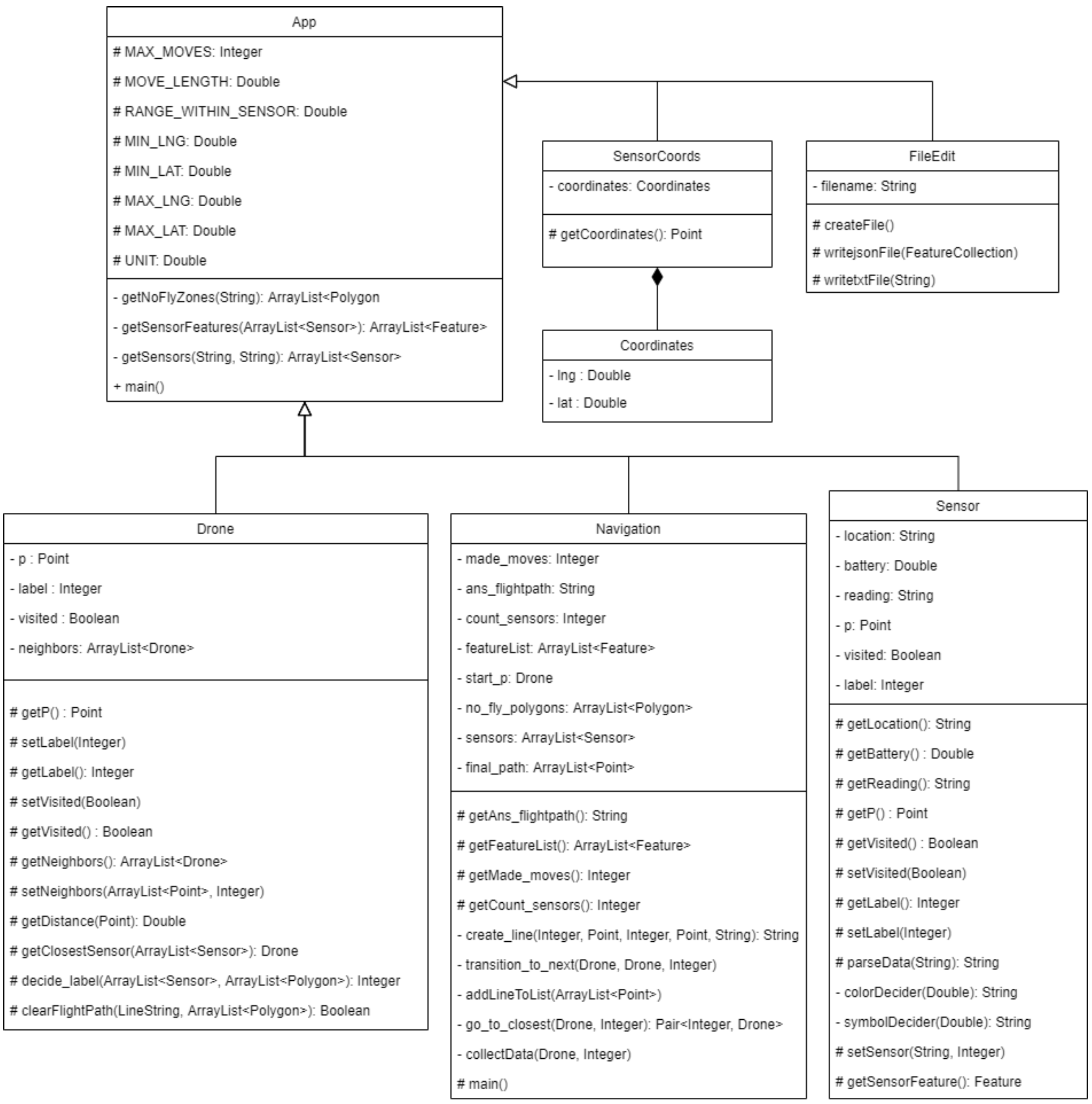
Alexandru-Ioan Chelba
s1865890

School of Informatics
University of Edinburgh
Scotland, UK
4 December 2020

Contents

1 Software Architecture Description

Below is the UML diagram of the classes I have:


App
# MAX_MOVES: Integer
# MOVE_LENGTH: Double
# RANGE_WITHIN_SENSOR: Double
# MIN_LNG: Double
# MIN_LAT: Double
# MAX_LNG: Double
# MAX_LAT: Double
# UNIT: Double
- getNoFlyZones(String): ArrayList<Polygon
- getSensorFeatures(ArrayList<Sensor>): ArrayList<Feature>
- getSensors(String, String): ArrayList<Sensor>
+ main()
SensorCoords
- coordinates: Coordinates
# getCoordinates(): Point
FileEdit
- filename: String
# createFile()
# writejsonFile(FeatureCollection)
# writetxtFile(String)
Coordinates
- lng : Double
- lat : Double
Drone
- p : Point
- label : Integer
- visited : Boolean
- neighbors: ArrayList<Drone>
# getP() : Point
# setLabel(Integer)
# getLabel(): Integer
# setVisited(Boolean)
# getVisited() : Boolean
# getNeighbors(): ArrayList<Drone>
# setNeighbors(ArrayList<Point>, Integer)
# getDistance(Point): Double
# getClosestSensor(ArrayList<Sensor>): Drone
# decide_label(ArrayList<Sensor>, ArrayList<Polygon>): Integer
# clearFlightPath(LineString, ArrayList<Polygon>): Boolean
Navigation
- made_moves: Integer
- ans_flightpath: String
- count_sensors: Integer
- featureList: ArrayList<Feature>
- start_p: Drone
- no_fly_polygons: ArrayList<Polygon>
- sensors: ArrayList<Sensor>
- final_path: ArrayList<Point>
# getAns_flightpath(): String
# getFeatureList(): ArrayList<Feature>
# getMade_moves(): Integer
# getCount_sensors(): Integer
- create_line(Integer, Point, Integer, Point, String): String
- transition_to_next(Drone, Drone, Integer)
- addLineToList(ArrayList<Point>)
- go_to_closest(Drone, Integer): Pair<Integer, Drone>
- collectData(Drone, Integer)
# main()
Sensor
- location: String
- battery: Double
- reading: String
- p: Point
- visited: Boolean
- label: Integer
# getLocation(): String
# getBattery() : Double
# getReading(): String
# getP() : Point
# getVisited() : Boolean
# setVisited(Boolean)
# getLabel(): Integer
# setLabel(Integer)
# parseData(String): String
- colorDecider(Double): String
- symbolDecider(Double): String
# setSensor(String, Integer)
# getSensorFeature(): Feature

My application has one main class: App.java, and 5 subclasses of it: Sensor, SensorCoords, FileEdit, Navigation, Drone. These classes have well defined roles and each support the application in particular tasks which need to be performed.

Class App is the main driver of the project. It is important since it acts as the scheduler of tasks and puts all parts of the project in the right order, so as to solve the given problem.

Class Drone encapsulates the drone, all its attributes and capabilities. It is a key class since it defines all the properties of the drone and each function defines a capability that the drone has, such as to recognise the neighboring areas, appreciate whether there is a no-fly zone on the path to any neighboring area, to find the closest sensor to its current location and to calculate the distance to a certain position on the map.

Class Navigation is essential since it provides the main body of the algorithm. It makes use of the processing abilities of the drone to construct the solution for the problem, and sketches the required answers in variables ans_flightpath and featureList.

Class Sensor is used to memorize the details of each sensor: location, reading, battery. I added to each sensor a label (a number from 1 to 33) and a Boolean variable which is true if the drone read details from this sensor, or false otherwise. This class is important because it encapsulates every necessary detail about a sensor.

SensorCoords is a utility class and is used to read the coordinates of each sensor from "details.json" files. These are then passed to the corresponding object of class Sensor, which memorizes the coordinates in a Point object.

FileEdit is another utility class and contains all functions which deal with creating and writing into files. It is responsible with putting the solutions found in class Navigation into the required files: "flightpath.txt" and "readings.geojson" in the correct format.

When I created these classes, my thoughts were that I need to have a super class which calls all the sub-tasks in order, then I need to have the sensors and the drone as definite entities, and, lastly, I need to have a class which implements the algorithm, and one which deals with creating the necessary files and outputting the solution in the required format.

2 Class Documentation

1. **App**:

   In App class, there are some final variables declared: MAX_MOVES, MOVE_LENGTH, RANGE_WITHIN_SENSOR, MIN_LAT, MIN_LNG, MAX_LAT, MAX_LNG. These are immutable, because they are given as such in the coursework sheet. It also has a UNIT immutable variable, used mainly to initialize maximums where the maximum needs calculated. Since all these variables are used in different classes in the project, they have visibility set to protected status.

   - *ArrayList⟨Polygon⟩ getNoFlyZones(String port)*: takes input one string, representing the server's port number and outputs a list of polygons, each corresponding to a no-fly zone.
   - *ArrayList⟨Feature⟩ getSensorFeatures(ArrayList¡Sensor¿ sensors)*: takes input one list of sensors and returns a list of features, each feature corresponding to each sensor.
   - *ArrayList⟨Sensor⟩ getSensors(String port, String date)*: takes input 2 strings: the server's port number and the date for which data is to be collected. It returns a list of the sensors found in the file corresponding to the required date.
   - *void main(String[] args)*: takes input a list of strings, representing the input of the user to the application. Calls each necessary class and function to fulfill the overall objective of the application.

2. **Drone**:

   Class Drone has variables that help describe any of the drone's possible states: the location's coordinates are held in Point p, the label of the position (if it is in a sensor's area, or inside a no-fly zone, or outside), the status of the position (if the drone has visited it or not) and the list of neighbors. All variables are private, getters are provided for every variable, setters are provided only for visited and label.

   - *void setNeighbors(ArrayList⟨Point⟩ path, int angle)*: Overwritten method of the setter for the list of neighbors. It takes input a list of points, corresponding to the path traversed by the drone, and an integer, corresponding to the angle at which the drone traveled last time. Creates the neighbor's list, marking as visited the neighbors which are on the traversed path, or opposite the drone's direction of traveling, at an angle varying ±20 degrees.

- *double getDistance(Point y)*: takes input a point and outputs the Euclidean distance of current object's position to that point.
- *Drone getClosestSensor(ArrayList⟨Sensor⟩ sensors)*: takes input a list of sensors and returns the Drone object of the closest sensor to the current object. If all sensors have been visited, it returns an object at a location out of the map.
- *int decide_label(ArrayList⟨Sensor⟩ sensors, ArrayList⟨Polygon⟩ nfz)*: takes input a list of sensors and a list of polygons, representing the no-fly zones. It returns the label of the current object:
  (a) -1, if the object is outside the confinement area or in a no-fly zone;
  (b) label of a sensor, if the object is within the required range of that sensor;
  (c) 0, otherwise.
- *boolean clearFlightPath(LineString li, ArrayList⟨Polygon⟩ polys)*: takes input a LineString, bounded by the drone and a neighbor, and a list of polygons, with each polygon representing a no-fly zone. It returns true if the LineString does not cross any of the no-fly zones.

3. **Navigation**:

In Navigation class, I declare variables that are needed in order to build the solution: starting coordinates in start_p, no-fly zones in no_fly_zones and the sensors' list in sensors. I also declare variables which help remember the solution: made_moves helps at counting the number of moves the drone made, counting_sensors is used to count the number of sensors the drone has collected data from, ans_flightpath is used to remember the string that will be written into flightpath.txt, final_path retains all points of the whole path traversed by the drone, and featureList which is used to remember the LineString of the path which the drone traverses. These are all private variables, and only getters are provided only for the latter kind of variables: made_moves, count_sensors, ans_flightpath, featureList.

- *String create_line(int idx, Point ps, int angle, Point pf, String loc)*: takes input an integer, representing the index of the line in flightpath.txt; 2 points, representing the initial point (ps) and the destination point (pf); an integer, representing the angle at which pf is found from ps; and a string, representing the location of the destination point. It returns a string, representing a line with index idx, with all the required details.
- *void transition_to_next(Drone actual_pnt, Drone next_pnt, int angle)*: Takes input an integer, representing the angle at which the drone travels, and 2 Drone

objects, representing the drone's actual state (actual_pnt) and the drone's next state (next_pnt). It modifies ans_flightpath and increases made_moves.

- *void addLineToList(ArrayList⟨Point⟩ path)*: takes input a list of points, representing the path traveled by the drone, and creates a feature out of it to then add it to featureList.
- *Pair⟨Integer, Drone⟩ go_to_closest(Drone actual_pnt, int closest_sensor)*: takes input a Drone object, representing the drone's present state, and an integer, representing the label of the closest sensor to the drone. It returns a tuple formed of:
  (a) the angle at which the possible next state is found
  (b) the drone's next possible state
- *void collectData(Drone actual_pnt, int closest_sensor)*: takes input a Drone object, representing the actual state of the drone, and an integer, representing the label of the closest sensor to the drone. if closest_sensor = 34, that means I seek to return to the starting point with the remaining available number of moves. Otherwise, it gets the drone into the area of the sensor with label closest_sensor, marks the sensor as visited and chooses the label for the next closest_sensor. If it is called with closest_sensor¡34, it does this until all sensors are visited. If closest_sensor=34, it works until the drone gets in the starting point's area. In both cases, it also stops if the maximum number of moves is reached.
- *void main()*: calls the function collectData with the necessary input.
  The algorithm constructs a path for the drone, using a Greedy logic:
  (a) The drone flies towards the closest sensor;
  (b) It marks it as visited;
  (c) Calculates the new closest sensor to the drone's current position, not visited yet;
  (d) If all sensors are visited, drone goes back to the starting point.

  The algorithm stops when either:
  (a) it has reached the maximum number of moves available
  OR
  (b) it is within range of the starting point.

4. **Sensor**:

Class Sensor has variables that help describe any of the sensor's states: the sensor's location name is held in location, the value that can be read from the sensor is held in reading, the sensor's battery level is held in battery, the location's coordinates are held in Point p, the label of the sensor in label and the status of the position (if the sensor has been visited, or not) in visited. All variables are private, getters are provided for every variable, setters are provided only for visited and label.

- *String parseData(String urlString)*: takes input an URL string and returns a string, corresponding to the data found at the provided URL address. This function throws error if it can't retrieve the data or if the provided URL address doesn't exist.
- *String colorDecider(double x)*: takes input a double and, based on coursework sheet, it returns a string representing the corresponding rgb-string:
  (a) if x is between 0 and 256, it returns the appropriate rgb-string;
  (b) If x is -1, it returns the rgb-string for ”low battery”;
  (c) If x is -2, it returns the rgb-string for ”not visited”.
- *String symbolDecider(double x)*: takes input a double and, based on coursework sheet, it returns a string representing the corresponding symbol-string:
  (a) if x is between 0 and 256, it returns the appropriate symbol-string;
  (b) If x is -1, it returns the symbol-string for ”low battery”;
  (c) If x is -2, it returns the empty string.
- *void setSensor(String port, int lbl)*: takes input a string, representing server's port number, and an integer, representing the label's value. It sets the current location and the label of the current object, and marks the current object as not visited.
- *Feature getSensorFeature()*: Function returns a Feature, corresponding to current object's feature, with the properties corresponding to the level of battery and the reading value, as described in the coursework sheet.

5. **SensorCoords**:

   This class has a Coordinates class variable declared as private, with a getter provided for it. This variable is used to read any sensor's coordinates and pass them to class Sensor.

   - *class Coordinates*: used to parse the coordinates data from the json file for each sensor.
   - *Point getCoordinates()*: getter for Coordinates object. Returns a Point object from the longitude and latitude provided.

6. **FileEdit**:

   This class has a single private variable declared: filename, through which it receives the name of the file that needs created and written.

   - *void createFile()*: takes a string as input and checks if a file with that name exists. if it does not exist, it creates one. This function throws error if it cannot create the file.
   - *void writejsonFile(FeatureCollection fc)*: writes the geojson-formatted string representation of the FeatureCollection object in the file with name filename. This function throws error if it cannot write in the file.
   - *void writetxtFile(String text)*: writes the given string in filename. It throws error if it cannot write in the file

3 Drone Control Algorithm

Overview

Overall, the algorithm is based on a Greedy approach. At each step, the drone selects the sensor which is closest to its current position in Euclidean distance terms, and has not been visited yet. The drone then tries to reach its area and, once it does, the sensor is marked as visited and the whole process restarts. At the end, The sensors which have been visited are marked on the map according to the reading value and the battery level, and those which have not been visited, are marked accordingly.

Steps of the algorithm

1. **Pre-processing**

   - *Declare immutable variables*

     At the very beginning of App.java, I declare immutable variables for a move's length, the number of maximum moves available, the range within which the drone needs to be in order to collect data from a sensor, a unit used to initialize maximums where the maximum needs calculated, and the limits of the confinement area.

   - *Memorize no-fly zones in a list of Polygon objects*

     Since no-fly zones are represented as Polygon objects in json, I extract them all in a list of Polygon objects, using the function App.getNoFlyZones.

   - *Encapsulate sensors and put them all in a list*

     In Sensor.setSensor, I read a sensor's data using Sensor class, and the coordinates at which the sensor is located, using SensorCoords. I unite these pieces of information in Sensor class, having the coordinates passed as a Point object to Sensor class. I label it and mark it as not visited. Finally, once the object is created for a sensor, I add the new Sensor object to the list of all Sensor objects in App.getSensors.

   - *Encapsulate the drone*

     The coordinates of the starting position of the drone are inputted from keyboard, hence I take them in App.main, encapsulate them in a Point object and then add the Point object to the Drone class. The Drone class tells what label the current location of the drone has, if it has been visited before and the nearby points that are accessible from this location. The nearby accessible points are calculated using the function Drone.setNeighbors.

- *Decide label for the position of the drone*

  The label of the current location is calculated using the function Drone.decide_label. If the drone is in the area of a sensor, it marks it as visited, then looks for the closest sensor not yet visited.

2. **Select the closest sensor**

- *Calculate Euclidean distance to each sensor*
- *Memorize the label of the sensor for which the Euclidean distance is smallest*

  Both these steps are done in Drone.getClosestSensor, with the help of Drone.getDistance. Drone.getClosestSensor returns an integer corresponding to the label of the closest sensor which has not been visited yet.

3. **Reach the sensor's area**

- *Calculate all the next possible states*

  Drone.setNeighbors deals with this. The next possible states are the neighboring accessible locations. We calculate each neighbor's position, using simple trigonometry, and look to build its object. For simplicity, the algorithm adds all neighbors objects to the list and differentiates them through the setting of the visited status. If:

  – the neighbor has already been visited (i.e. it is in the list path, provided as parameter), OR
  – the neighbor is opposite the direction of traveling, at an angle varying ± 20 degrees:

  Then the function marks them as visited, so that the drone knows those points as inaccessible.

- *Select the next state*

  Once the drone has established the closest sensor to its current position and has the list of neighbors set up, it selects to move towards the neighbor that is closest to the target sensor, and which has not been visited yet. In case a neighbor has the same label as the target sensor, the drone chooses to go there without checking if it is the closest point to the sensor. All these happen in Navigation.go_to_closest.

- *Avoid no-fly-zones*

  Drone.decide_label can label whether the point is in a no-fly zone and the algorithm knows not to go there. However, it is Drone.clearFlightPath that makes sure the drone does not fly above a no-fly zone while flying. Drone.clearFlightPath needs the line between the drone and the neighbor to be drawn beforehand and then not keep it if it goes above any no-fly zone.

- *Avoid getting trapped in small areas by restricting drone's movement*

  Drone's movement is restricted in a number of ways:

  (a) The drone cannot go to a state in which it has been before, i.e. any previously visited points.

  (b) The drone cannot fly backwards. It is only at the start that it can fly in all directions, after that it has interdiction to fly backwards, i.e. if its current flight happened at 0 degrees, the next flight cannot happen at 180 ± 20 degrees. These restrictions which are set up in Drone.setNeighbors. The case in which it needs to access those areas in order to exit a certain area on the map, is dealt with in Navigation.collectData: The drone's orientation is reset in the opposite way.

- *Visit the sensor*

  Once the drone is in the closest sensor's area, it visits the sensor, counts it and resets closest sensor's label. This happens in Navigation.collectData.

4. **Go back to the starting position**

   Going back happens only if the drone visits all sensors and still has some remaining moves. I took this decision after I tested on over 100 of the provided instances, and found that the algorithm finishes the whole circuit with less than 150 moves made in each case. Going back to the starting point happens when collectData is called with parameter closest_sensor = 34.

   - *Mark all the points on the map as not visited*

     All visited points on the map are marked as not visited. This is because I want the drone to go in as few steps as possible to the starting point, hence it would not help to have certain points not accessible. I still maintain the interdiction to go backwards.

   - *Select the next state*

     The next state's selection happens exactly as when the drone flies towards a sensor: it selects the neighbor which is closest to the starting point.

   - *When it stops*

     I chose to consider the circuit is completed when the drone is within the same range of the starting point, as if the starting point was a sensor and it needed to collect the data from it. This decision came after I did a short mathematical calculation and found out that with a move's length of 0.0003, the drone cannot miss an area of a circle with 0.0002 radius, but can miss one with a smaller radius and just loop around it until the maximum number of moves is reached.

5. **Output the solution**

   - *Prepare data for output*

     Before writing to the 2 required files, data needs to be gathered and prepared. In App.java, I gather the LineString which draws the path that the drone traversed, and the Sensor Feature representations, which are constructed with the help of the function App.getSensorFeatures and Sensor.getSensorFeature. I unite these in a single Feature list and transform this list in a FeatureCollection. For flightpath.txt, I gather the string ans_flightpath from Navigation class, since this contains all the necessary details in the correct format for output.

   - *Create files*

     This happens in FileEdit.createFile. The class takes as constructor parameter the name of the file to be created and creates it only if it does not exist. In App.java, I create two instances of FileEdit, one for each necessary file to be created.

   - *Write the solution in flightpath.txt*

     Writing into flightpath.txt happens with the help of FileEdit.writetxtFile. Since the string that is passed to the function is already in the correct format, it is just a matter writing it in the existing file.

   - *Write the solution in readings.geojson*

     Writing into readings.geojson happens with the help of FileEdit.writejsonFile. The function takes input a FeatureCollection, hence before writing to the file, it needs to construct the Json string. I chose to print the code in the "pretty" way for easier debugging.

Why does it work?

The algorithm follows a Greedy approach. It does not intend to find a best solution, instead it will always look to find a local optima. This ensures that the algorithm terminates in a reasonable number of steps.

The traveling restrictions help the drone not get stuck in small loops around the no-fly zones, and emergency changes of direction help it not get blocked in certain areas on the map. In function Navigation.collectData, the variable path_added ensures the traveled path is added to the answer even if the objective is not fully completed, hence partial solutions are provided in case the algorithm fails to visit all sensors and reach back to the starting point in the maximum number of moves available.

I have added the following figures which illustrate that the algorithm terminates and provides viable paths. At the same time, it is visible the solutions are not optimal in both cases.

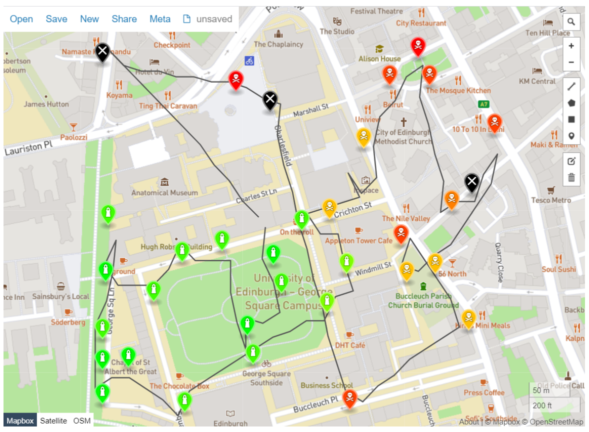


Figure 1: Output for date 12/12/2020. Took 112 moves.

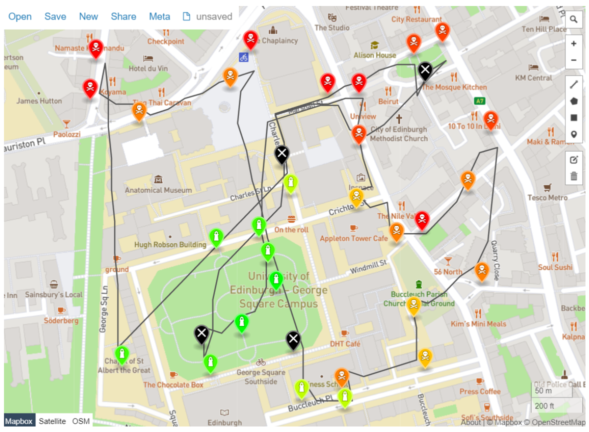


Figure 2: Output for date 08/04/2020. Took 122 moves.

4 Bibliography

- `https://github.com/maxogden/geojson-js-utils`

  The code for Drone.clearFlightPath is inspired from this link, specifically from the geojson-js-utils.lineStringsIntersect function. My function is a Java translation of what is there.

- MapBox TurfJoins documentation

  I used TurfJoins.inside(Point, Polygon) in Drone.decide_label, to see if a point is inside any of the no-fly zones.

- `https://www.baeldung.com/java-tuples`

  From this website, I learned how to use javaTuples.Pair, which I use in class Navigation.