PowerGrab

Informatics Large Practical Report

Name: Keyi Liu
Matriculation number: s1703084
Date: 01/12/2019

PART I: Introduction

This project aims to develop a prototype of a location-based strategy game named PowerGrab. My implementation is to complete an autonomous component which includes two different level of drones: stateless and stateful to play against novice players and expert players respectively. The game's goal is to focus on collecting power and cryptocurrency coins form 2D virtual charging stations around UoE central Area. The following is the description and evaluation of functionalities in my Maven project.

PART II: Software Architecture Description

My application is made up of a collection of 11 Java classes. The General Structure is represented by simplified UML class diagram shown below (See Figure 1), and I will explain why I identified each class in my implementation.

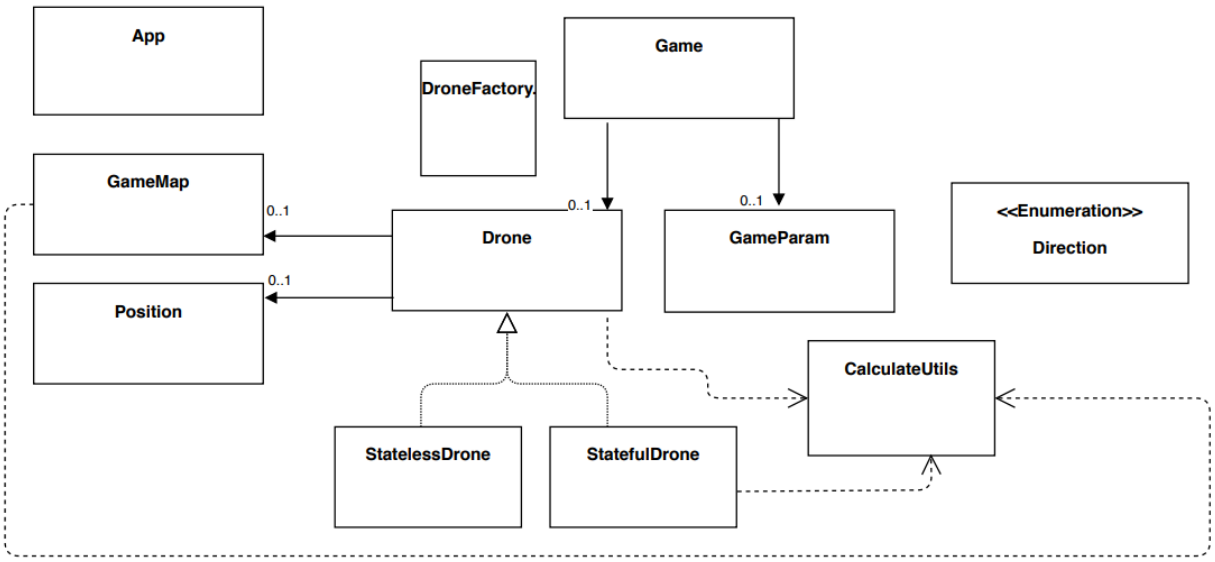

Figure 1: Simplified UML Class Diagram

◆ App.java

This class aims to receive the parameters from command-line arguments by the main method which is working as entry point of the whole application. So in this class checking the argument array is very important, if any argument is missing or out of range, the application stops immediately to report the related error message on console. The command-line should obey the format shown below:

DD MM YYYY 55.944425 -3.188396 SEED DroneType

DD means Date, MM means Month, YYYY means Year, (55.944435, -3.188396) is the starting point of the drone on the map. SEED is to initialise pesudo-random number generator, it can be 5678, 4567 and so on. DroneType can only be the stateful or stateless.

◆ GameParam.java

This class is used to ensure safety and data hiding by wrapping up the parameters(data) received from App.java under this single class to prevent the parameters from being changed by other codes outside this protecting mechanism — Encapsulation.

◆ DroneFactory.java

This Factory class I create here is used to generate the corresponding object according to subclasses (either statefulDrone.java or statelessDrone.java) based on the parameters stored in GameParam.java which is originally from command-line argument(DroneType). The advantage of introducing the Factory class is to encapsulate the details of how we create the specific object, working as multiplexer module to choose single signal from multiple signals in hardware description language.

◆ Direction.java

Using an enum to represent this class as a collection of the all constant values, they are 16 compass directions that are available for the drone to choose from before each movement.

◆ Position.java

This class aims to calculate the next position in the form of (latitude, longitude) when the drone decides to make a movement in the specified direction and check if the next position is still inside the range of the map.

◆ GameMap.java

The class is all about dealing with Geo-JSON maps. By using java.net package to download the specified map based on the information provided by the input parameters. Working with the useful classes provided from Mapbox JAR files can update information of coin and power of each station (feature) after the drone finishes visiting, and can export the flying path and a copy of the map documents. Also, Calculating distance method between positions is defined here through the measurement of Pythagorean distance(See Figure 2). In addition, every time a map is downloaded, all information of station(features) will be saved in a collection and the total number of positive coins of a specified map will be calculated for later analysis.

$$\sqrt{(Latitude1 - Latitude2)^2 - (Longitude1 - Longitude2)^2}$$

Figure 2: Pythagorean Distance Measurement

◆ CalculateUtils.java

To avoid losing the precision of calculated data in this application, I need to import the java.math.BigDecimal and defined methods for four basic arithmetics operations to ensure the precision. Further, In order to avoid method declarations duplicating and to do concise calculation in position, power and coin in the game, all the arithmetics operations will be called in this application defined here.

◆ Game.java

This class aims to initialise the game at the beginning including getting the type of the drone and the specified map for this new game. Define the conditions to check if the game is over, which are the drone making 250 steps or running out of the power that it holds. And export the .txt file and .geojson file as required. Moreover, the console will print the result of this game, using 2019-05-05 as an example.

```
GameOver! 2019-05-05, stateful drone, step:250, coins:1877.159738 ,
MaxCoins:1877.159738  CollectiveRate:100.000000%
```

◆ Drone.java

To keep this project organised, understandable and avoid duplicating codes, I use the ideas of abstract classes to group two classes together which are stateful.java and stateless.java. There are lots of code shared between two classes such as assigning the initial steps, power and other related information to the drone, writing flying log after each movement, assigning the power or coin to zero when they are reduced to negative value, updating the station after visiting and so on. However, the two drones will use different strategies to achieve move(), collection() and find next direction, define the common part in these methods and two subclasses will override them to complete their different implementations.

◆ StatelessDrone.java

This class is the subclass of the Done.java, providing the implementation of the abstract methods, set a “dumb” strategy for the stateless drone to let this component are suitable for new beginners of this game.

◆ StatefulDrone.java

This class is the subclass of the Done.java, providing the implementation of the abstract methods, set a “smart “strategy for the stateless drone to let this component are suitable for experienced player this game. The strategy for the stateful drone will be introduced at part IV and the detailed implementations on abstract methods will be described in part III.

PART III: Class Documentation

◆ Field concise explanation

Java Class Name	Field Name	Field Type	Access_modifier	Static	Final	Description
Direction	X	Enum	Public	No	No	Uses the Enum class to define the 16 compass direction which are N, NNE, NE, ENE, E, ESE, SE, SSE, S, SSW, SW, WSW, W, WNW, NW, NNW. These constant can be used by all classes in this application through Direction.N, Direction.values() and so on.
Game	ALL_STEP	int	Public	Yes	Yes	Sets the ALL_STEP to 250 and cannot be changed anymore, representing the maximum movements of the drone can make.
Drone	STEP_FUEL	double	Public	Yes	Yes	Sets the STEP_FUEL to 1.25 and cannot be changed anymore, representing the energy cost for each step(movement)
Drone	COLLECT_DISTANCE	double	Public	Yes	Yes	Sets the COLLECT_DISTANCE to 0.00025 and cannot be changed anymore, representing the distance of automatically transferring coin and power between stations and drone
Drone	power	double	Protected	No	No	Initialises power to 250 at first, representing the holding power of drone, this variable will be changed multiple times in the game.
Drone	coin	double	Protected	No	No	Initialises coin to 0.0 at first, representing the the holding coin of drone, this variable will be changed multiple times in the game.

Drone	step	int	Protected	No	No	Initialises step to 0 first, recording the steps(movement) of the drone, every time the drone make a movement, step+1.
Drone	flyPath	List<Point>	Protected	No	No	Every time the drone is ready to make a movement, flyPath will add the line between current position and next position
Drone	flyLog	String	Protected	No	No	Every time the drone makes a movement, flyLog will add a new String with format: old position, next direction, new position, coin, power.
GameParam	day	String	Private	No	No	Gets the value from App.java by using gameParam.setDay(args[0]). Hiding this parameter by declaring private can only be used by other classes though calling getDay().
GameParam	month	String	Private	No	No	Gets the value from App.java by using gameParam.setMonth(args[1]). Hiding this parameter by declaring private can only be used by other classes though calling getMonth().
GameParam	year	String	Private	No	No	Gets the value from App.java by using gameParam.setYear(args[2]). Hiding this parameter by declaring private can only be used by other classes though calling getYear()
GameParam	latitude	double	Private	No	No	Gets the value from App.java by using gameParam.setLatitude(args[3]). Hiding this parameter by declaring private can only be used by other classes though calling getLatitude()
GameParam	longitude	double	Private	No	No	Gets the value from App.java by using gameParam.setLongitude(args[4]). Hiding this parameter by declaring private can only be used by other classes though calling getLongitude()
GameParam	seed	long	Private	No	No	Gets the value from App.java by using gameParam.setSeed(args[5]). Hiding this parameter by declaring private can only be used by other classes though calling getSeed()
GameParam	droneType	String	Private	No	No	Gets the value from App.java by using gameParam.setDroneType(args[6]). Hiding this parameter by declaring private can only be used by other classes though calling getDroneType()

Position	R	double	Public	Yes	Yes	Sets R to 0.0003, representing the fixed displacement of each movement of the drone
Position	latitude	double	Public	No	No	The variable of latitude is to represent the latitude of position(latitude, longitude)
Position	longitude	double	Public	No	No	The variable of latitude is to represent the latitude of position(latitude, longitude)
GameMap	BASE_URL_TPL	String	Public	Yes	Yes	Defines the format of the map downloading address "http://homepages.inf.ed.ac.uk/stg/powergrab/%s/powergrabmap.geojson"
GameMap	FEATURE_MAX_Coin	double	Public	Yes	Yes	Declares the constant of the maximum coin stored in a station(Feature) on the map
GameMap	downloadUrl	String	Private	No	No	Assigns the downloadUrl in the constructor. String.format(BASE_URL_TPL,gameParam yyyymmdd("/")) to form an address
GameMap	allCoin	double	Private	No	No	Represents the total number of coin of all the positive stations in a specified map.
GameMap	features	List<Feature>	Private	No	No	A collection of all the stations(features) in a specified map
StatefulDrone	collectedFeatures	List<String>	Private	No	No	A collection of the id of the features(stations) we visited
StatefulDrone	nextFeature	Feature	Private	No	No	The next feature(stations) the drone decides to visit

◆ Operation concise explanation

Java Class Name	Constructor or Method	Description
App	main(String []): void	Reads the command-line argument, each parameters will be stored directly to the new GameParam object. All the parameters will be checked if it is valid, use try-catch and printStackTrace to locate the error parameters and print the reason on the console
DroneFactory	createDrone(GameParam, GameMap):Drone	Uses switch-case to generate the object for subclass by **gameParam.getDroneType()**. The new object concludes the initial position and game map information.
CalculateUtils	DoubleAdd(double, double):double	In order to keep concise. To calculate addition of big decimal numbers and use **.doubleValue()** to convert to double type.

CalculateUtils	DoubleSubtract(double, double):double	In order to keep concise. To calculate subtraction of big decimal numbers and use **.doubleValue()** to convert to double type
CalculateUtils	DoubleMult(double, double):double	In order to keep concise. To calculate Multiplication of big decimal numbers and use **.doubleValue()** to convert to double type
CalculateUtils	DoubleDivide(double, double):double	In order to keep concise. To calculate division of big decimal numbers. **b1.divide(b2,8,BigDecimal.ROUND_HALF_UP)** is to round the result to nearest 8 decimal places. And use **.doubleValue()** to convert the result to double type
Game	Game(GameParam)	Intialises newly instance of Game by using parameters stored in GameParam.java
Game	init():void	Initialises the map and creates a type of drone
Game	play():void	Uses while to continue the game until **isGameOver()== True,** wrapping the output files name by using String.format and export them. Using **System.out.println()** to print the final score of this game for evaluating the performance of the drone.
Game	isGameOver():boolean	Uses the conditions to stop the game when **steps == Game.ALL_STEP**(which is 250) or power is equal to 0.
GameParam	getPosition():Position	Uses the receiving parameters to create a new instance of starting position of the drone
GameParam	yyyymmdd(String):String	Defines a format to represent the date description in URL address, this String format is in the form of year, separator, month, separator, day.
GameParam	ddmmyyyy(String):String	Defines a format to represent the date description of the name of output files, this String format is in form of day, separator, month, separator, year.
GameParam	Setter and Getter	Generates Setter and Getter for all private variables declared in this class including day, month, year, latitude, longitude, seed and droneType.
GameMap	updateFeature(Feature):void	After the drone has visited the station, updates the coin and power of that station.
GameMap	getDistance(Position, point):double	Gets the distance between the drone and station
GameMap	getDistance(Position,Position):double	Gets the distance between the old position and the new position of the drone, calling the **CalculateUtils.DoubleSubtract**.

GameMap	GameMap(GameParam)	Initialises a new instance, assign the variable downloadUrl with downloading address format and downloading the map. using **FeatureCollection fc = FeatureCollection.fromJson(mapJsonStr)** to get all stations information and put them into **List<Feature>**. In addition, accumulating the total number of positive coin by filtering the **List<Feature>** if **feature.getProperty("coins").getAsDouble()>0.**
GameMap	exportMap(String, List<Point>): void	use Geometry class to draw the line of a collection of moving position by using **LineString.fromLngLats()**. And adds the original copy of the map, by using **Files.write(Paths.get(path) , mapInfo.toJson().toString().getBytes());** to write the .geojson file.
GameMap	downLoadMap():String	**mapUrl.openConnection()** return a java.net.URLConnection object which we can downcast to java.net.HttpURLConnection. And throw a java,io,IOException.
GameMap	Setter and Getter	Generates Setter and Getter for all private variables declared in this class including downloadUrl, features and allCoin.
Position	Position(double, double)	Initialises a newly object of Position with parameters of latitude and longitude.
Positon	nextPosition(Direction): Position	Using switch-case to calculate the next position by applying specified compass direction.
Position	isValidLatitude():boolean	Check if the latitude are in the valid latitude range which is between 55.942617 and 55.946233.
Position	isValidLongitude():boolean	Checks if the longitude are in the valid longitude range which is between -3.192473 and -3.184319
Position	inPlayArea():boolean	Determines if the position satisfies **isValidLatitude()** and **isValidLongitude()**.
Drone	Drone()	Executes the **init()** to initialise the variables
Drone	Drone(Position, GameMap)	Initialises a newly object of Drone with parameters of position and GameMap.
Drone	init():void	Assigns power = 250.0, coin = 0.0, step = 0, fly path = new ArrayList<>() and flyLog = ""when a concrete drone is created.

Drone	init(Position, GameMap):void	Assigns the position as current position and initialises the gameMap.
Drone	playing():void	This method is used during the game, it will execute **move()**, **collection()** and adding flyLog.
Drone	nextDirection():Direction	This is an abstract method, the subclass must complete implementation of all the abstract classes.
Drone	move(Direction)	Updates the currentLocation and step and power after each movement.
Drone	collection():void	Calls the **calculate()** after collecting, the holding coin and power cannot be smaller than 0. Interrupts Automatic transferring when coin or power decrease to zero.
Drone	collectionFeature():Feature	If the drone is in the range of two stations at the same time, still collecting from the closest station. Comparing the distance of two stations by using the **Double.MAX_VALUE**.
Drone	calculate(Feature):void	Calculates the holding power and holding coin when the drone is in range of a station. If the coin or power held by the drone is negative, it will be recorded as 0, the station will still keep the rest of the negative power or coin.
Drone	nextDirection(): Map<Direction, Double>	Finds the next position where the drone is ready to go. Get all possible 16 next positions by applying 16 directions in **nextPosition(direction)**. Gets the position of stations, calculate distance between all possible next positions with stations, if the distance inside the the collective range. Return the **directionFeatures**.
Drone	exportLog(String):void	By using **Files.write(Paths.get(path), flyLog.getBytes());**
Drone	Setter and Getter	Generates Setter and Getter for all private variables declared in this class including power, coin, step, flyPath and flyLog.
StatelessDrone	nextDirection(): Direction	Overrides the abstract method To avoid the negative station by using **directionFeatures.entrySet().removeIf(v -> v.getValue() < 0);** and filter the stations which has no coins left. After filtering, using **collection.sort()** to rank the Map<Direction,Double> in descending order, Choose the station with the most coin. Otherwise, random choosing a direction.

StatefulDrone	nextDirection():Direction	Overrides the abstract method but Inherits all Methods from Drone.java. Get next direction by calling the **getNextDirectionByFeature(nextFeature)** If the nextFeature is null, executing the nextFeature to do re-searching.
StatefulDrone	Collection(): void	Overrides the method in Drone.java, the stateful drone will record the station it has already visited. If the chosen next feature has been visited before, the **feature** will be null. Otherwise, records the **id** of the this feature and execute collecting, then update the information of the drone and the station.
StatefulDrone	getRandomDirection():Direction	If the **feature** = null by somehow, choose random direction to choose but still avoid the negative station by using: **directionFeatures.entrySet().removeIf(v -> v.getValue() < 0);**
StatefulDrone	getWeight(double, double):double	Calculates the stepNum by distance/0,0003 Calculates the stepRatio by stepNum/ remaining steps Calculates the CoinRatio by: coins/FEATURE_MAX_COIN Calculates the Earnings yield by: CoinRatio-stepNum*35
StatefulDrone	nextFeature():Feature	Scans the whole map to choose all the positive features and calculates the distance between the current position with these features separately then stores in **featureDistance**. If the **featureDistance.size() == 0**, return feature =null.Otherwise, sort the featureDistance in descending order, then choose the top 5 from the featureDistance collection. Gets the coin information and distance among the top 5, using the **getWeight()** to calculate each station's **Earnings yield**. By using a new hashMap<>() to store the **<Feature, Earnings yield>** of the top 5 stations. And sort this collection and return the station with largest Earnings yield.

PART IV: Strategy And Evaluation

◆ Strategy Applied on Stateful Drone

The idea of achieving the smart component is to let the drone "think" as human beings. The stateful drone I have implemented can scan the whole map, memorise the station position it has been to and can choose directions and its next positions wisely. The detailed strategy shows below:

1. Scan the whole map apart from all the negative stations and visited stations, store the stations still with positive coins to a ***collection named A***.
2. Calculate ***distance*** between all the stations and current position respectively.
3. Create a subclass ***collection B*** by choosing five closest station from ***collection A*** based on current position and store the distance information.
4. Get the ***coin*** information from the five closest station and store the ***coin*** information.
5. Calculate EarningsYield of each station in ***collection B*** by using the formula I create (See Figure 3). ***Coin*** and ***distance*** are information stored in each station in ***collection B.*** The constant ***maxCoin*** represents the maximum payload which a charging station can have and the constant ***R*** represents the fixed displacement of the each movement. Using the ratio of ***coin/maxCoin*** to represent the coin percentage of this station, the ratio of ***(distance/R)/remaining steps*** to represent the percentage of the cost of steps to this station.

$$EarningsYield = \frac{coin}{maxCoin} - \frac{distance \div R}{remaining\ steps} * weight$$

$$where\ maxCoin = 125.0,\ R = 0.0003$$

Figure 3: EarningsYield For Stateful Drone

- The ***EarningsYield = coin percentage — stepsCost percentage * weight***. It is used to measure how two ratios influence the decisions made by the stateful drone before choosing the next station it intends to visit. Obviously when remaining steps are too many, the stepsCost percentage will be extremely small, the coin percentage seems have more influence on the drone when choosing stations, the drone will choose the station with more coins even though not very close among the ***collection B***. However, as the remaining steps are becoming smaller and smaller, StepsCost percentage will become very large for example 80% or 90%, at this

situation, choosing the closest station is the best choice. Overall, the way of the drone to make a decision is floating by the remaining steps. If there are plenty of steps left, the drone intends to visit the station with more coins, otherwise the drone would prefer the closest station.

- The ***weight*** in this formula is for us to optimize and adjust the weight of the StepsCost percentage in order to achieve better performance of the stateful drone. And how to choose the best weight will be introduced under the subtitle of *Optimization and Improve The Performance* later.

6. Choose the station with the maximum value of EarningsYield from ***collection B*** and get the position information of that station. The station will be chosen as the next feature that the stateful drone is going to visit if it has not been visited.
7. Calculate the next position of all possible moving directions, filtering the next position in the range of the negative station. Then choose the next moving direction which is closest to the next station we are going to.
8. If the no station chosen from ***collection B*** that means the drone has already visited all the available positive stations, then executing random moving but it still needs to avoid the negative stations.

◆ Optimization and Improve The Performance

Weight is used to adjust the importance of the stepsCost percentage when the drone makes decisions. By testing different Weights to check which one is the best value to work with EarningYield formula to let the performance of stateful drone more stable. The performance of the stateful drone is determined by the collection rate of the coins. [Collection Rate = coin collected / allCoin] Using 12 maps as sample deal with weight = 10, 25, 30, 35 and 50 respectively, the line chart shown below (See Figure 4). We can find when weight = 25 or weight = 35, the stateful drone has better and more stable performance on collecting coin. From the data recorded on Appendix A(See page 15), we can find the average collection rate based on weight = 25 and weight = 35 are both around 93%. However, the stateful drone achieves 100% when applying weight = 35.

It seems like using weight = 35 is the best choice. We then need to increase sample size to do further evaluation. I have chosen 3 months in a year, which are February, May and August. The 90 raw data can be checked on Appendix A (See Page 15), we can find 69 days (77%) among the three months in which the stateful drone achieves the score above 90%, 17 days with the score being 100% and 84 days (93%) with the score above 80%. From these 90 days data, we can find this implementation with weight = 35 is very successful.

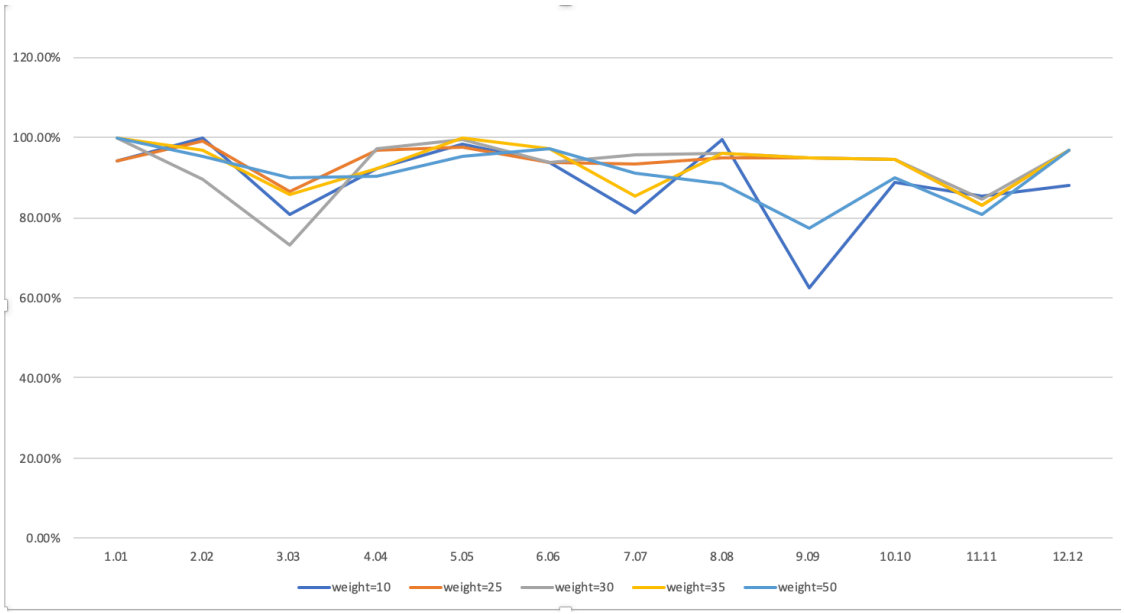

Figure 4: Line Chart of the Performance on Different Weights

◆ FlyPath of Stateless Drone VS Stateful Drone on 2019-12-01

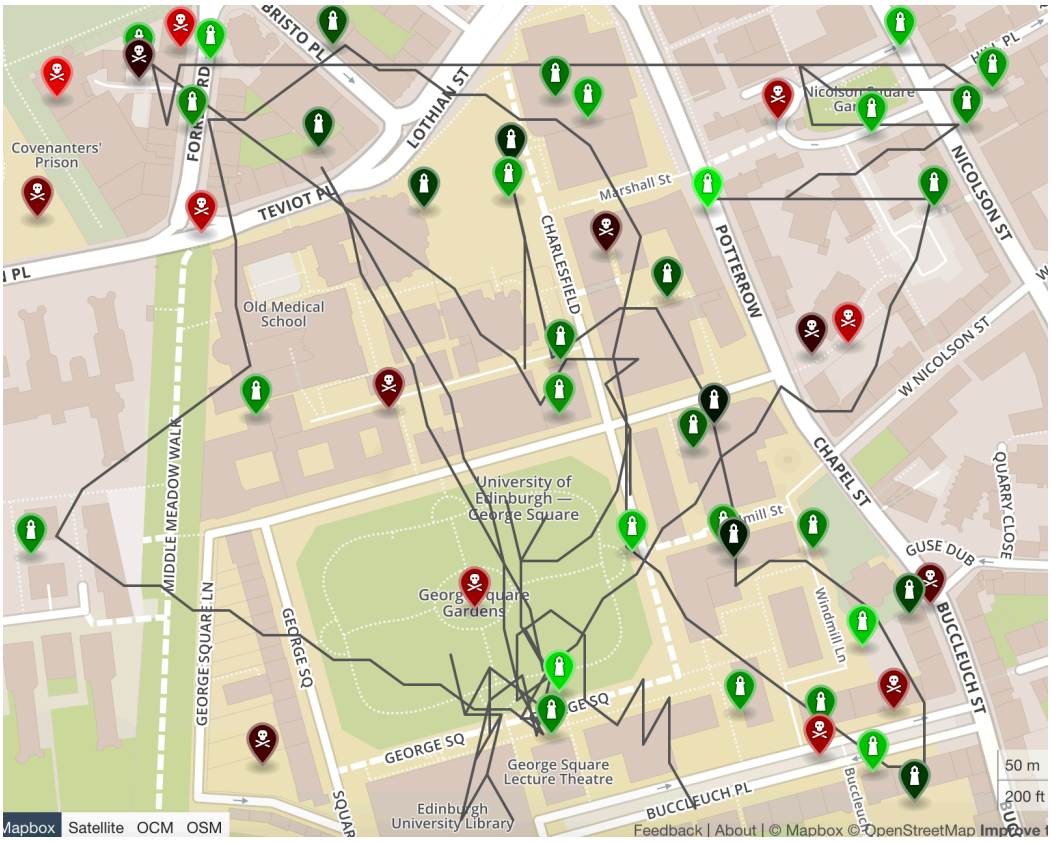

Figure 5: Stateful Drone FlyPath on 2019-12-01

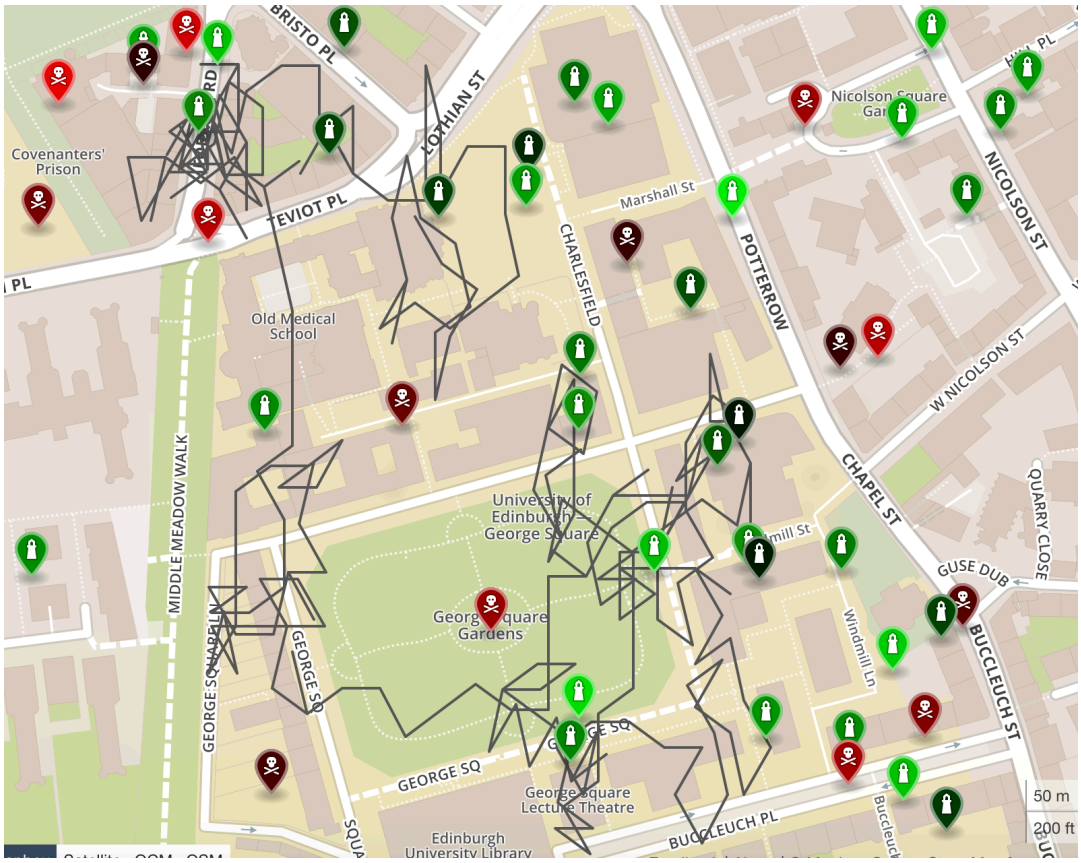

Figure 6: Stateless Drone FlyPath on 2019-12-01

PART V: Acknowledgements

GeeksforGeeks: enum in java [online]
Available at: https://www.geeksforgeeks.org/enum-in-java/
[Accessed 01 Oct. 2019]

MAPBOX | Docs: Class summary of com.mapbox.geojson [online]
Available at: https://docs.mapbox.com/android/api/mapbox-java/libjava-geojson/3.0.1/com/mapbox/geojson/package-summary.html
[Accessed 02 Nov. 2019]

TutorialsPoint: Design pattern and factory pattern [online]
Available at: https://www.tutorialspoint.com/design_pattern/factory_pattern.htm
[Accessed 09 Nov. 2019]

Oracle: Class BigDecimal [online]
Available at: https://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html
[Accessed 15 Nov. 2019]

Appendix A: Raw Testing Data

(I) The recorded raw data used in the line graph on Figure 4

Date	weight=10	weight=25	weight=30	weight=35	weight=50
1.01	94.23%	94.23%	99.76%	99.76%	99.76%
2.02	100.00%	99.17%	89.36%	96.86%	95.19%
3.03	80.91%	86.51%	73.28%	85.58%	90.03%
4.04	92.29%	96.95%	97.23%	92.29%	90.42%
5.05	98.32%	97.41%	99.47%	100.00%	95.27%
6.06	93.91%	93.91%	93.91%	97.22%	97.22%
7.07	81.28%	93.35%	95.82%	85.20%	91.07%
8.08	99.49%	94.78%	95.96%	95.96%	88.45%
9.09	62.51%	94.85%	94.85%	94.85%	77.37%
10.10	88.72%	94.58%	94.58%	94.58%	89.92%
11.11	85.17%	83.06%	84.61%	83.06%	80.88%
12.12	87.93%	96.62%	96.62%	96.62%	96.65%
Average	88.73%	93.78%	92.95%	93.50%	91.02%

(II) Due to limited space, only the first 23 days of the three months are shown here.

Date	score	Date	Score	Date	Score
2019-02-01	83.42%	2019-05-01	91.14%	2019-08-01	92.02%
2019-02-02	96.86%	2019-05-02	99.98%	2019-08-02	96.03%
2019-02-03	100.00%	2019-05-03	98.79%	2019-08-03	94.98%
2019-02-04	100.00%	2019-05-04	100.00%	2019-08-04	10.31%
2019-02-05	96.33%	2019-05-05	97.95%	2019-08-05	96.59%
2019-02-06	32.08%	2019-05-06	94.21%	2019-08-06	93.33%
2019-02-07	100.00%	2019-05-07	96.34%	2019-08-07	30.86%
2019-02-08	99.62%	2019-05-08	100.00%	2019-08-08	95.96%
2019-02-09	94.76%	2019-05-09	100.00%	2019-08-09	88.60%
2019-02-10	93.65%	2019-05-10	91.60%	2019-08-10	100.00%
2019-02-11	94.78%	2019-05-11	100.00%	2019-08-11	93.61%
2019-02-12	94.33%	2019-05-12	98.75%	2019-08-12	88.50%
2019-02-13	95.19%	2019-05-13	87.13%	2019-08-13	95.96%
2019-02-14	97.11%	2019-05-14	91.34%	2019-08-14	92.29%
2019-02-15	86.54%	2019-05-15	96.91%	2019-08-15	100.00%
2019-02-16	90.90%	2019-05-16	23.06%	2019-08-16	90.42%
2019-02-17	93.31%	2019-05-17	95.96%	2019-08-17	98.60%
2019-02-18	95.65%	2019-05-18	87.62%	2019-08-18	36.39%
2019-02-19	96.67%	2019-05-19	95.20%	2019-08-19	92.98%
2019-02-20	100.00%	2019-05-20	90.72%	2019-08-20	87.87%
2019-02-21	71.77%	2019-05-21	14.40%	2019-08-21	95.17%
2019-02-22	98.87%	2019-05-22	91.57%	2019-08-22	96.48%
2019-02-23	100.00%	2019-05-23	100.00%	2019-08-23	95.35%