Informatics Large Practical Report

s1740055

September 10, 2021

Contents

1 Software Architecture Description

1.1 Software Architecture Design

The architecture of my software is relatively simple and straightforward. The classes are App, ChargingStation, Drone, StatefulDrone, StatelessDrone, Position and Direction.

The App class is the main body of the software. It contains a drone and all charging stations. I treat the App as a map, because there is nothing more than a map in this application and all of the stations and the drone are on the map. Therefore, a App class is enough to hold all data.

There are 50 charging stations on the map and stations have two types, the positive value charging stations (light houses) and negative value charging stations (skulls). So the ChargingStation class encapsulates state information of a charging station, such as its position, utilities and the type of the charging station. It also has basic functions to handle transformations and calculate the distance between the station and the drone.

The drone is the only moving object on this map. A Drone class is responsible for status and movement of a drone. Since the Drone class is a abstract base class, a Drone instance should encapsulate state information of the drone and provides some helper functions for both stateful and stateless drones. This class also provides a abstract class for the drone to implement the strategy of the drone.

Because there are two types of drones and there strategies are totally different. The StatelessDrone and StatefulDrone classes handle the actual decision making of stateless and stateful drones respectively.

The Position and Direction classes are given. The Position class is like a data type. A Position instance represents a coordinates on the map. Direction class is a enumeration class which indicates 16 fixed compass directions used in the practical.

Initially, I had two classes, Util and Constants, in order to store any utility functions and constants respectively. The Util class, like a centre server, carries out different computation requests and the Constants class just stores several fixed values for easy management. However, I noticed that the constants and functions can be placed in different classes because a function/constant is only used in a specific class under my architecture. For example, the function about computing Euclidean distance is only called in ChargingStation class and POWER_CONSUMPTION variable is only used in the Drone class. If we think about real world scenario, the drone may have limited computing performance and memory. It is difficult for a stateful drone to perform any strategy itself. But from software development perspective, more classes we have, the higher complexity of the architecture is. Therefore I removed these two classes for "high cohesion, low coupling". Moreover, since the software does not provide any API, all functions and variables are either private or package private in terms of encapsulation, apart from the main in the App.

1.2 Class Relationships

Figure 1 shows the relationships between each class. The relationship between App and ChargingStation is one-to-many, because there are many stations in the application. Drone class and App class are one-to-one related. Only one drone exists on the map and the drone needs to access some data in the App. For most of the cases, the drone only needs data about charging stations and there supports to be a relationship between ChargingStation and Drone, but I think they are completely independent and the communication between these two classes should be through App. The relationship between Position and ChargingStation is one-to-one and the relationship between Position and Drone is also one-to-one, since a station or a drone can only have one related location. The only Hierarchical relationship in my architecture is that a abstract base class, Drone, is inherited by StatefulDrone and StatelessDrone classes.

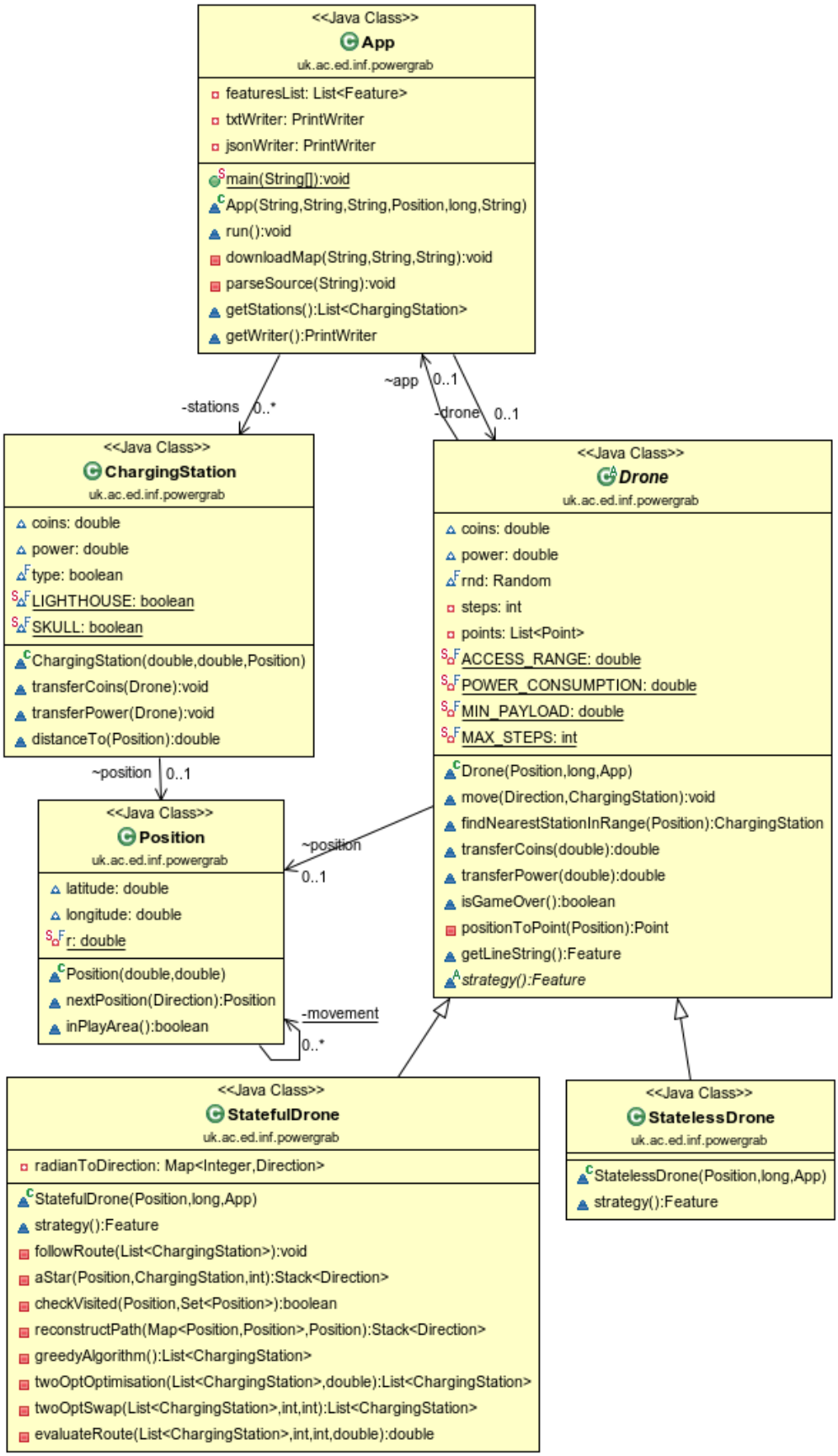

Figure 1: The UML Class Diagram of Architecture

2 Class Documentation

2.1 App

The App class is the main part of the application. Charging stations and a drone are encapsulated in a App instance. The App initiates charging stations by downloading GeoJSON file online according to the first three arguments which specify which date the map is. Then it initiates a drone by last three arguments, which specify the initial position where the drone starts, a random seed for generating random movement of the drone and the type of this drone. After the execution, a text file and a GeoJSON file are created and appended with status changes of the drone and a map information for visualising the path of the drone respectively.

Constructor Detail

App(String day, String month, String year, Position initDronePos, long seed, String droneType)

Constructs the App with the specific map and a drone with the specific state information. The constructor returns a App instance after the app is initiated. The processes of construction includes downloading the specific map, initiating the specific type of drone and two PrintWriters for recording the path and the status of the drone.

parameters:

day - the day of the map
month - the month of the map
year - the year of the map
initDronePos - the position where the drone starts
seed - the initial seed
droneType - the type of the drone

Method Detail

void run()

Executes the drone and appends the LineString returned by drone to the feature collection of the map. Then writes the feature collection to a GeoJSON file.

private void downloadMap(String year, String month, String day)

Requests a map from the specific date online and converts the InputStream from the website into a JSON String.

parameters:

year - the year of the map
month - the month of the map
day - the day of the map

private void parseSource(String mapSource)

Retrieves information about charging stations from specific JSON string and stores each charging station into a ArrayList.

parameters:

mapSource - the JSON string about the map.

List<ChargingStation> getStations()

Returns a array of charging stations.

PrintWriter getWriter()

Returns a PrintWriter of a text file.

2.2 Position

A Position instance is a coordinate on the map, described by latitude and longitude. It is specified in double precision. This class also provides functions to create nearby positions from sixteen compass directions and check if the current position is inside the play area. The function about creating next position is used a lot in this application. So a static block is applied to initialise a HashMap between sixteen compass directions and related unit movements $(\delta y, \delta x)$:
δy = the length of each move * sin(the angle of the direction)

δx = the length of each move * cos(the angle of the direction)
When new position is required, only addition is manipulated.

Constructor Detail
public Position (double latitude, double longitude)
Constructs and initializes a position at the specified coordinate.
parameters:
latitude - the Y coordinate of the newly constructed Point
longitude - the X coordinate of the newly constructed Point

Method Detail
public Position nextPosition(Direction direction)
Creates a new Position instance nearby current location, towards specific direction.
parameters:
direction - the direction where the next position should be.
public boolean inPlayArea()
Returns true if current position is in the play area.

2.3 Direction

A enumeration class includes sixteen fixed compass directions.

2.4 ChargingStation

The ChargingStation class represents charging stations. A ChargingStation instance encapsulates state information about a charging station. These information includes:

> the number of coins/debts in the station
> the power stored in the station
> the position of the station
> the type of the station

Moreover, two static Boolean constants, LIGHTHOUSE and SKULL, indicates two types of charging station.

Constructor Detail
ChargingStation(double coins, double power, Position position)
Constructs a new charging station according to the specific data.
parameters:
coins - the number of coins/debts stored in the station
power - the power stored in the station
position - the position of the station

Method Detail

void transferCoins(Drone drone)

Transfers coins/debts to the specific drone and updates the number of coins in the charging station by how many coins the drone takes.

parameters:

drone - the drone that is connected to the charging station

void transferPower(Drone drone)

Transfers power to the specific drone and updates the quantity of power in the charging station by how much power the drone takes.

parameters:

drone - the drone that is connected to the charging station

double distanceTo(Position p)

Returns the Euclidean distance from the charging station to the specific position.

parameters:

p - the destination position for computing distance

2.5 Drone

The Drone class is a abstract base class. A Drone instance represents a drone with its status. The state information for a drone includes:

> the position of the drone
> the number of coins in the drone
> the power stored in the drone
> the initial seed for random move
> the step counter counts how many steps the drone has made
> a list of Points tracks the path of the drone
> the instance of App that the drone is in

Also, some constants in the Drone class represents some limitations to the drone. The maximum steps for a drone is 250, the power consumption of each step is 1.25, the minimum payload of the drone is 0 and a drone is able to connect to the closest charging station and send transformation request if and only if the distance between the station and the drone is smaller than or equal to 0.00025.

The drone can only move to one of the sixteen directions placed in the Direction class and it should try to move towards charging stations with positive value, while avoiding charging stations with negative value if possible.

Constructor Detail

Drone(Position position, long seed, App app)

Constructs a drone at specific position. The power of the drone is initialised to 250. The number of coins in the drone and the step are 0 initially. The initial position is also added in the list of Points.

parameters:

position - the position where the drone starts
seed - the initial seed
app - the instance of App that the drone is in

Method Detail

void move(Direction direction, ChargingStation station)

Moves the drone to the position at specific direction and update its status. The power is reduced by 1.25 and the step counter increments. If the nearest charging station in the access range is not pre-calculated or not found, this method will try to find the closest charging station in the range again after every movement and send transformation requests to the station if a station is found. This checking pre-calculation feature is designed for stateless drone in order to avoid duplicate computation. Finally, new position is added to the list of points and new status of the drone is written to text file by the print writer.

parameters:

direction - the direction of the drone flies to

station - the closest charging station in the access range, it can be null

ChargingStation findNearestStationInRange(Position p)

Returns the nearest charging station in the access range of the specific position. This method implements Euclidean distance from every charging stations in the App to the specific position.

parameters:

p - the destination position for computing distances

double transferCoins(double amount)

Returns the number of coins that the drone retrieves from a charging station. The coins change based on how many coins/debts are stored in the station and how many coins are stored in the drone. The minimum payload of a drone is zero.

parameters:

amount - the total number of coins/debts stored in the charging station

double transferPower(double amount)

Returns the power that the drone retrieves from a charging station. The power changes based on how much power are stored in the station and how much power are stored in the drone. The minimum payload of a drone is zero.

parameters:

amount - the total quantity of power stored in the charging station

boolean isGameOver()

Return true if the power of the drone is less than 1.25 or the steps of the drone has made reaches 250.

Point positionToPoint(Position position)

Returns a Point data type with same latitude and longitude of the specific position

parameters:

position - the position to convert to Point.

Feature getLineString()

Returns a LineString that represents the path of the drone, which is a list of Points.

abstract Feature strategy()

A abstract class returns a LineString feature. The implementation of this method is in subclasses.

2.6 StatelessDrone

The StatelessDrone class implements the strategy of a stateless drone. A stateless drone is a memoryless drone and designed to against amateurs. Its decision of the next move to make can only be based on information about the charging stations which are within range of the sixteen positions where the drone can be after one move from its current position, and guided by the general gameplay of tries to move towards charging stations with positive value, while avoiding charging stations with negative value if possible.

Constructor Detail

StatelessDrone(Position p, long seed, App app)

Constructs a stateless drone by calling the Drone class (super class) constructor.

parameters:

position - the position where the drone starts

seed - the initial seed

app - the instance of App that the stateless drone is in.

Method Detail

Feature strategy()

Returns a LingString feature represented the path of the drone. From the start position, the drone scans its neighbours of the sixteen positions one by one. Searches for the closest charging stations with positive quantity of coins in the range for every neighbour. If such charging station is found, the drone will move to that direction directly and transfer coins and power. Otherwise, if no such station is found or there is a light house with zero coin in that direction, this direction will be added to a list which represents safe directions. After all directions have been searched, a list with safe directions is established and the drone picks a direction from the list randomly. If the list is null which means the current position of the drone is surrounded by negative value charging stations, the drone moves to a random direction. Processes above iterates until the game is over.

2.7 StatefulDrone

The StatefulDrone class implements the strategy of a stateful drone. A stateful drone is designed to against experts. The drone flies more purposefully and tries to collect as many coins as possible. Several algorithms are implemented in this class for different purposes. The greedy search algorithm and 2-opt optimisation algorithm are combined to solve the access order of light houses. A* search algorithm is deployed to find a path from current position of the drone to a charging station. We shall discuss this in more detail later in the Stateful Drone Strategy.

Constructor Detail

StatefulDrone(Position p, long seed, App app)

Constructs a stateful drone by calling the Drone class (super class) constructor. A HashMap is also generated for converting the radian between two positions to the relative direction and the HashMap will be used in A* algorithm for every path to charging stations.

parameters:

position - the position where the drone starts

seed - the initial seed

app - the instance of App that the stateless drone is in.

Method Detail

Feature strategy()

Returns a LingString feature represented the path of the drone. Firstly, it calls the greedy search algorithm to generate a optimised "greedy" route which is a list of charging stations to let the drone know the access order. Then the drone tries to fly to every station following a path created by A* search algorithm. If the game is still on after all deterministic processes, the drone enters random mode and starts moving randomly and safely until the game is over.

private void followRoute(List<ChargingStation> route)
Lets the drone follows the specific route. A* search algorithm is used to find a path from current position of the drone to a charging station in the route and the drone follows the path returned by A*. After the drone is within the range of charging station, the drone moves to the next charging station by the guide of A* and so on iteratively, until the game is over or all charging stations on the route are visited.
parameters:
route - a list of charging stations for the drone to visit

private Stack<Direction> aStar(Position drone, ChargingStation station, int attempt)
Returns a stack of directions which indicates the direction of each movement. A* search algorithm is implemented in this method. For more detail, please check Stateful Drone Strategy.
parameters:
drone - the start position
station - the destination
attempt - the counter represents how many times the drone have tried to get to the station.

private boolean checkVisited(Position current, Set<Position> visited)
A helper function of A*. Returns true if the specific position is in the visited positions set.
parameters:
current - the position A* currently searching
visited - the set contains visited positions

private Stack<Direction> reconstructPath(Map<Position, Position> cameFrom, Position current)
A helper function of A*. Returns a stack of directions for the drone to fly. It backtracks from the destination position to the start position by a HashMap and gets the direction between every adjacent pair of positions.
parameters:
cameFrom - a HashMap contains positions and the parent of positions
current - the destination position of A*

private List<ChargingStation> greedyAlgorithm()
Returns a list of charging stations that indicates the optimised access order of light houses. The method implements greedy search algorithm to find a route and then optimises the route by 2-opt optimisation algorithm.

private List<ChargingStation> twoOptOptimisation(List<ChargingStation> route, double routeLength)
Returns a list of charging stations that indicates the optimised access order of light houses. This method optimises the route generated by greedy search algorithm. It tries every possible combinations of the swapping mechanism and finds the two-opt route.
parameters:
route - the route calculated by greedy
routeLength - the length of the route

private List<ChargingStation> twoOptSwap(List<ChargingStation> current, int a, int b)
Returns a list of charging stations which represents a new route. This method implements the swapping mechanism of 2-opt.
parameters:
current - the list of charging stations represents the old route
a - the index of the start of the sub-list
b - the index of the end of the sub-list

private double evaluateRoute(List<ChargingStation> route, int a, int b, double routeLength)
Returns the length of new route created by 2-opt. It implements simple subtractions and additions from the length of old route in order to get the length of the new route.
parameters:
route - the list of charging stations represents the old route
a - the index of the start of the sub-list
b - the index of the end of the sub-list
routeLengthe - the length of the old route

3 Stateful Drone Strategy

The spirit of this problem is a (physical) travelling salesman problem (tsp). Therefore, the strategy for a stateful drone is splitted into two parts.

The first is a travelling salesman solver about to visit all light houses on the map. Dangerous stations are not taken into account at this step. The greedy search algorithm is the simplest and also efficient way to solve the problem. The heuristic of the greedy search algorithm is nearest unvisited neighbour. The algorithm starts at the initial position of the drone and looks for the closest light house to it. Assuming the drone is on the exact position of the closest light house after the charging station is found, it searches for the closest light house again from current position and so on, until all light houses are visited. The greedy search algorithm only looks good at each step, but the final solution is not optimal. It only considers the optimal solution locally instead of globally and can generates some intersections and also long paths between some pairs of light houses. The more optimal solution should be a Hamiltonian path.

In order of find a more optimal route that visits all light houses, 2-opt optimisation is introduced to resolve the crossings of the greedy route. The main idea is to take a route that crosses over itself and reorder it so that it does not. The idea of this algorithm is that it looks for a 2-adjacent tour with lower cost than the current tour. If one is found, then it replaces the current tour. This continues until there is a 2-optimal tour. Be more specific, a complete 2-opt local search compares every possible valid combination of the swapping mechanism. For every combination, the algorithm takes two indices from the array and reverses the order of the sub-array between these two indices to create a new route. If there is a new route which performs better than the old route, it replaces the old one. A example of how the swapping mechanism works is shown below:

example route: A -> B -> C -> D -> E -> F -> G -> H -> A
example i = 4, example k = 7 (starting index 1)
new_route:
1. (A -> B -> C)
2. A -> B -> C -> (G -> F -> E -> D)
3. A -> B -> C -> G -> F -> E -> D (-> H -> A)

After the 2-opt, a Hamiltonian path that goes through every light house is found and is returned to the drone.

The second part of strategy is to find a path from the position of the drone to a light house. The A* search algorithm is chosen due to its completeness, optimality, and optimal efficiency.

The heuristic function in this case is the Euclidean distance from the position of the drone to the destination. The heuristic function is optimal for graph search because it is consistent. However, in terms of cost function, two different cost functions are required to handle different scenarios. One cost function is simply the actual distance, a multiply of the step length of the drone, from the start position to the current position of the drone. Another is based on the actual distance but a penalty is added to it when the drone steps into a negative value charging station. The penalty is proportional to the quantity of coins stored in the negative value charging stations and the minimum penalty is three extra steps. The combination of a cost function and the heuristic function produces the f-score of a position.

For each iteration of A* algorithm, the current searching position is replaced by a position with the minimum f-score and the new current position is added to visited positions.

After the position with the minimum f-score is retrieved, A* checks if it has searched for a long time right. If iteration counter reaches the threshold, the current search will stop to start a new A* search with the same destination beginning at the current position. This means the drone will move to the position as far as the position that A* search reaches and will start a new A* search from that position to the same destination. The threshold is set to 1250, which is a empirical result. Because the searching of every path takes fewer than 1250 iterations on all given maps. A light house (destination) will be searched for three times and then it is considered

unreachable after three attempts.

If the current iteration is not timed out, it expands the current position. The goal test is applied to every neighbour. If one of the neighbours is within the range of the destination charging station and the charging station is the closest to the neighbour, A* terminates and returns a path from the start position to the current position. Otherwise, if this is the first A* search, all neighbours that are within the range of dangerous charging stations are ignored, along with visited neighbours. Then A* applies the cost function and the heuristic function without penalty to compute the f-scores to the rest of neighbours. If this is the second or third attempt, all neighbours apart from visited ones are considered and the cost function with penalty is applied when a neighbour is within the range of a negative value charging station and there are still some coins to gain if the drone flies through the negative charging station. The second and third attempt let the drone go through negative value charging stations with minimum loss. If the drone notices all available neighbours are visited in the first attempt, which means it is surrounded by negative charging stations, it increments the attempt counter and looks for a path with minimum loss to breakthrough.

After the A* terminates, a path from the start position to the terminate position is construct by backtracking. Because all searched positions and their parents are stored in a HashMap during A* searches and the ancestor of positions is the start position. The drone goes through the HashMap to find a list of positions from the terminate position to the start position of A*. A HashMap for mapping radians to directions is also created when the stateful drone is constructed. The direction of a position based on its parent position can be found by computing the radian between two positions. Then directions are pushed into a Stack in reverse order since the path is constructed by backtracking. Finally, the drone is able to follow the path by popping directions from the Stack.

After all positive value charging stations are visited, the drone starts moving randomly and it is able to avoid negative charging stations as usual. The drone stops when it runs out of power or it has moved 250 steps.

The images about the performance of Stateless Drone and Stateful Drone are illustrated in figure 2 and figure 3 respectively. From the images, the performance of stateful drone is much better than that of stateless drone.

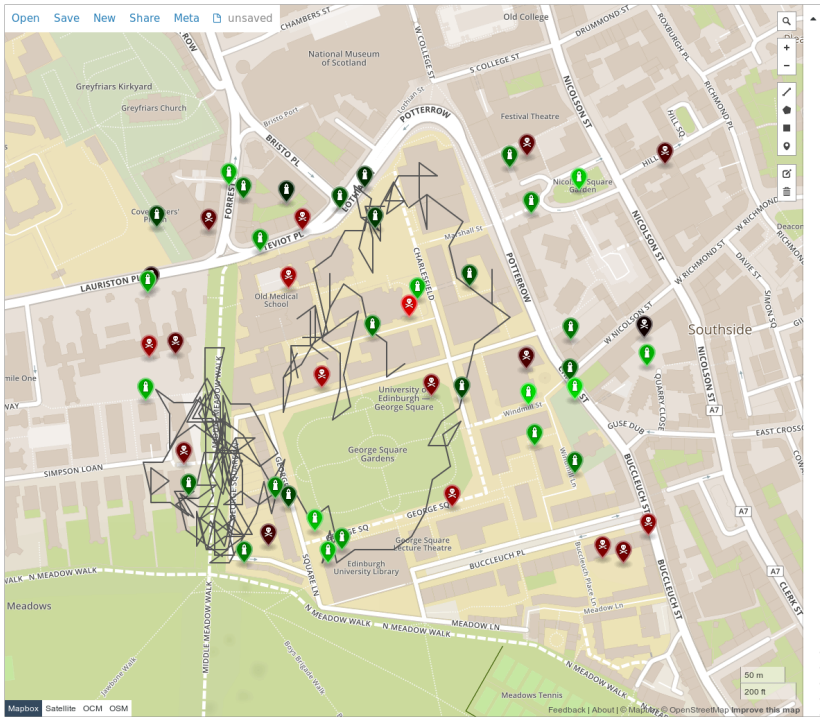

Figure 2: The path of stateless drone in 09 09 2019

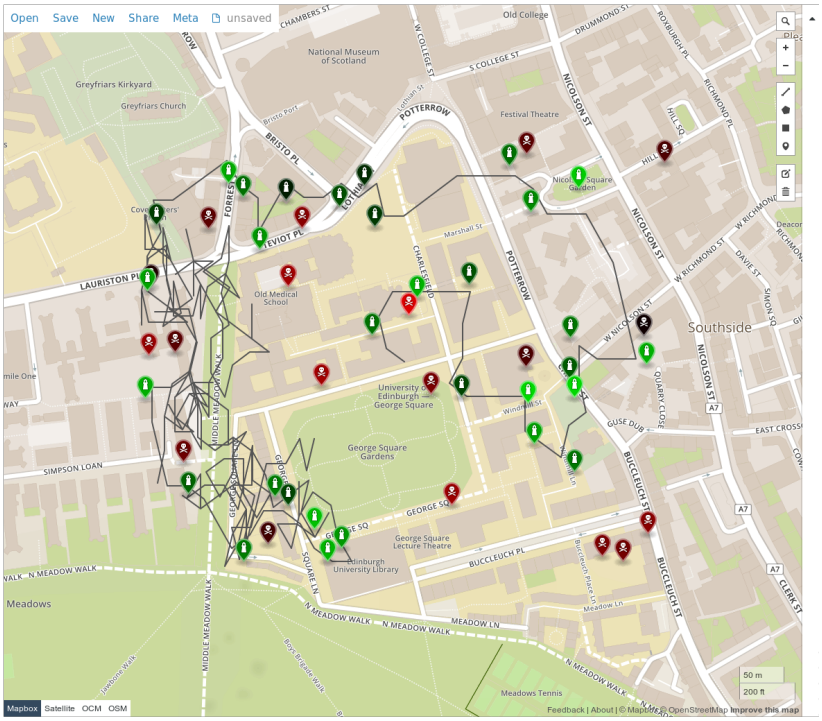

Figure 3: The path of stateful drone in 09 09 2019