INFORMATICS LARGE PRACTICAL REPORT

Powergrab Game

TAI Yintao

S1891075 | y.tai-6@sms.ed.ac.uk

Table of Contents

1. Software Architecture Description

1.1 Architectural Pattern

Why I choose to these classes to define? Reason as follows.

The Powergrab is designed in a layered pattern. Each layer has different abstraction and realize different functions. The lower layer provides service to the next higher layer, and the higher layer's service depends on the lower layers service. There are 4 abstraction layers in Powergrab, from bottom to the top, they are Data representation layer, Information Query layer, Drone layer and the Application layer.

	Layer	Classes	Description
Layer 4	Application Layer	App	execute the application procedure
Layer 3	Drone Layer	Drone Stateless Stateful	operations and strategy logic of drones
Layer 2	Information Query Layer	Geography Map LineDrawer	calculate information which cannot be stored
Layer 1	Data Representation Layer	Position Direction Charger	store basic information

table of 4 layers

The task of Data Representation Layer is to store basic information. The Position class, Direction class and Charger class work in this layer for expressing different information. A Position object represents a Position on the game map, expressed in latitude and longitude. It's the most basic geographic object, only have position attributes. Normally, a Position object is used to transport location information between objects and stores the location information of an object. A Charger object represents a charger on the game map, hence it stores power coins and also location information. The Direction class is an enumeration with 16 major compass direction, and also provides methods related to direction.

The Information Query Layer's objective is to provide information which cannot be stored. When a drone is making decision, it would need to acquire some information which are not in storage but can be calculated from stored locations. Such like the distance to a Charger and which charger is the nearest charger and so on. Geography interface, Map class and LineDrawer class work in this layer. Any class with geographic information, namely the location information, should implement Geography interface. This interface defines methods to calculate distance, direction and angle between two geographic objects. Some

information we can only get through iterating all chargers. The Map class is defined for this case. A Map object is initialized from a map FeatureCollection and keeps all chargers information. When a drone needs to determine which charger it connects to, or which charger is the nearest positive charger, querying the Map object is a convenient way. Since the final results, the trace, coins and power records, are output to files, the LineDrawer class is defined. It records drone's states of every step and return records as FeatureCollection or String of specified format.

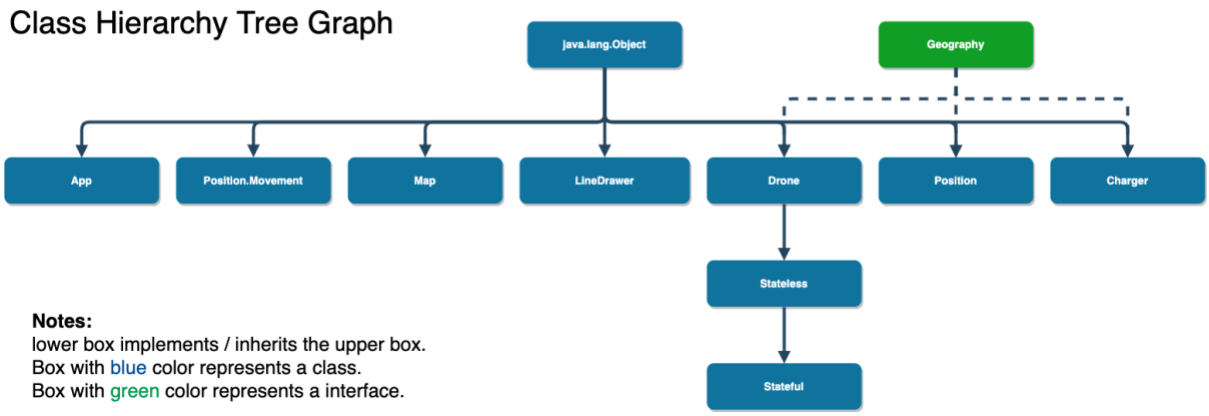
The Drone Layer defines operations and strategy of drones. There are two kinds of drones, the Stateless and the Stateful drones. They have different strategy but same operations, decide which position to go and go to that position. Hence they heritage these methods from the abstract Drone class and then override them to adapt different strategies. In some cases, the Stateful drone will also adapt Stateless' strategy, so a convenient way is let the Stateful drone to heritage the Stateless drone.

The Application Layer's objective is to execute the application procedure and output results. Only one class in this layer, the App class. Each time run this program, firstly it will analyse the legality of program parameters. Then initialize the Map, corresponding Drone and LineDrawer. After successfully initialized them, run the drone until it has no next step. Finally, output the result to files. If any step failed, the program will return and report the failure.

1.2 Class Hierarchy

Class tree:
java.lang.Object
- uk.ac.ed.inf.powergrab.App
- uk.ac.ed.inf.powergrab.Charger (implements uk.ac.ed.inf.powergrab.Geography)
- uk.ac.ed.inf.powergrab.Drone (implements uk.ac.ed.inf.powergrab.Geography)
  - uk.ac.ed.inf.powergrab.Stateless
    - uk.ac.ed.inf.powergrab.Stateful
- uk.ac.ed.inf.powergrab.LineDrawer
- uk.ac.ed.inf.powergrab.Map
- uk.ac.ed.inf.powergrab.Position (implements uk.ac.ed.inf.powergrab.Geography)

uk.ac.ed.inf.powergrab.Position.Movement



Class Hierarchy Tree Graph

Note 1: Drone is an abstract class, it defined a method to move the drone to the next position and update all states, but the strategy to decide which direction to go is undefined. Hence the Stateless and Stateful drone can adapt different strategies by overriding one method. In some cases the Stateful drone will also adapt Stateless strategy, so it heritage the Stateless.

Note 2: Both of the Drone, Charger and Position have position attributes, therefore they implement the Geography interface.

2. Class Documentation

Private fields are not included in this documentation.

public class Position

implements `Geography`

The Position class used to indicate a geographic position, and provides position related methods. A object of Position class is a basic geographic element, which implemented Geography interface. Also provides function to calculate the next position in a specific direction, and to judge whether this position is in play area.

*public Position **nextPosition**(Direction `direction`)*

This method is used to generate a new Position instance with a given direction, and without change the coordinate of itself.

Parameters: `direction` - The new position's direction of current position

Returns: `nxtPos` The instance of next position

*public boolean **inPlayArea**()*

Check whether the current coordinate is within the play area

Returns: If it's in the area, return true, otherwise return false.

*public boolean **equals**(java.lang.Object o)*

Judge whether another object have the same Position with an precision of 10^-12

Overrides: equals in class java.lang.Object

public double ***latitude****()*

Return the latitude of this geographic object.
Specified by: `latitude` in interface `Geography`
Returns: the latitude.

public double ***longitude****()*

Return the longitude of this geographic object.
Specified by: `longitude` in interface `Geography`
Returns: the longitude.

public enum Direction

extends java.lang.Enum<Direction>
Defines 16 major compass directions and some related methods. A Direction represents a major compass point. They are 16 enumeration constants from N to NNW and ordered in clockwise direction. This class also provides two methods, one for transfer an angle to a direction, one for iterating 16 directions in pseudorandom order.

Enum Constants
16 major compass directions
E, ENE, ESE, N, NE, NNE, NNW, NW, S, SE, SSE, SSW, SW, W, WNW, WSW

public static Direction ***angleToDirection****(double angle)*

Find the closest direction of an angle. Return the direction corresponding to the input angle. The angle is from North to some compass point in clockwise direction, expressed in radians.
Parameters: `angle -` the angle from North in radians.
Returns: the direction closest to the input angle.

public static java.util.Iterator<Direction> ***randomDirections****(java.util.Random rand)*

Return an iterator which iterates a randomly sorted direction list. The list has 16 directions from the Direction class with no repeats.
Parameters: `rand -` the pseudorandom generator used to shuffle the direction list.
Returns: an iterator which can iterates the direction list.

public class Charger

implements Geography
The charger class, an implementation of Geography interface with coins and power attributes. A charger object is a geographic object with coins and power attributes. It represents a charger on the map. Once it initialized, the location of it will not change.

public ***Charger****(double* `latitude`*, double* `longitude`*, double* `coins`*, double* `power`*)*

Constructs a charger with given latitude, longitude, coins and power. This is the only constructor of a charger, the latitude and longitude are given by parameters and stored in a Position object. The charger also have coins and power attributes.

Parameters:
`latitude` – the latitude of the new charger
`longitude` – the longitude of the new charger
`coins` – the initial coins of the new charger
`power` – the initial power of the new charger

public double ***latitude****()*
Return the latitude of this geographic object.
Specified by: `latitude` in interface `Geography`
Returns: the latitude.

public double ***longitude****()*
Return the longitude of this geographic object.
Specified by: `longitude` in interface `Geography`
Returns: the longitude.

interface Geography

Provides methods to get or calculate geographic information. Any class implemented this interface means its object has geographic information. Geographic information includes latitude, longitude, the distance to another object and the angle to another object.

double ***latitude****()*
Return the latitude of this geographic object.
Returns: the latitude.

double ***longitude****()*
Return the longitude of this geographic object.
Returns: the longitude.

default double ***distance****(Geography* `g`*)*
Calculate the distance to another geographic object. The distance is the Pythagorean distance calculated from latitude and longitude. It's an approximated distance, assuming the earth as a plane and latitude and longitude are equally distributed.
Parameters: `g` – the geographic object to calculate the distance of.
Returns: the distance.

default double ***distance****(double* `latitude`*, double* `longitude`*)*
Calculate the distance to a position with given latitude and longitude
Parameters: `latitude` – the given latitude**,** `longitude` – the given longitude
Returns: the distance.

default double ***angle****(Geography* `g`*)*
Calculate the angle to another geographic object. The angle from North to some compass point in clockwise direction, expressed in radians.
Parameters: `g` – the geographic object to calculate the angle of.
Returns: the angle expressed in radians.

public class Map

The game map stores all charger information and provides methods to query them efficiently. A map can be constructed from a GeoJson FeatureCollection or from a specified date. The Map will download the FeatureCollection according to the date. Different query method also defined in this class, drones can express their strategies intuitively by using them.

public ***Map****(com.mapbox.geojson.FeatureCollection* `featureMap`*)*

Constructs a map from a GeoJson FeatureCollection.

Parameters: `featureMap` – a GeoJson FeatureCollection with 50 chargers information.

public ***Map****(int year, int month, int day) throws java.io.IOException*

Constructs a map with a given date. The constructor download the map GeoJson file from http://homepages.inf.ed.ac.uk/stg/powergrab, then analyse the file and initialize chargers.

Parameters:

`year` – the year of the map.

`month` – the month of the map.

`day` – the date of the map in a month.

Throws: `java.io.IOException` – throw the error if anything wrong.

public *Charger* ***nearestPositiveCharger****(*Position* `pos`*)*

Find the nearest positive charger of a position.

Parameters: `pos` – the position we are currently considering.

Returns: return the nearest positive charger, if no positive charger, return null.

public *Charger* ***nearestCharger****(*Position* `pos`*)*

Find the nearest charger of a position.

Parameters: `pos` - the position we are currently considering.

Returns: return the nearest charger.

public *Charger* ***connectedCharger****(*Position* `pos`*)*

Find the connected charger of a position if it has.

Parameters: `pos` – the position we are currently considering.

Returns: return the connected charger. If no charger connected, return false.

public class LineDrawer

Provides methods to record the drone and add the trace to map FeatureCollection. Also to generate formatted trace String and FeatureCollection.

public ***LineDrawer****(com.mapbox.geojson.FeatureCollection* `fc`*)*

Constructs a LineDrawer with given map features

Parameters: `fc` – a map FeatureCollection

public void ***recordDrone****(Drone* `d`*)*

Record the drone's states and position

Parameters: `d` – the drone to be recorded

public com.mapbox.geojson.FeatureCollection ***mapWithLines****()*

Return a map FeatureCollection with flight trace and initial chargers.

Returns: the map FeatureCollection with recorded trace.

public java.lang.String ***flightTrace****()*

Write the drone movement records to a string. Each line in the string represents one record/move of the drone, in the format of <latitude before move>,<longitude before move>,<direction of this move>, <latitude after move>,<longitude after move>,<coins after move>,<power after move>

Returns: the movement records String.

public abstract class Drone

implements Geography

The abstract drone class, provides common operations and defines common attributes of a drone. A Drone object is a geographic object implements the geography interface. The basic operation ***goNextPosition*** is already defined in this class. So any drone heritages this drone can only override the ***findNextPosition*** to adapt a strategy.

public ***Drone****(Position* `startPosition`*, long* `seed`*, Map* `map`*, LineDrawer* `tracer`*)*

Constructs a drone with given position, seed, map and tracer.

Parameters:

`startPosition` – the initial position of this drone.

`seed` – the seed used to initialize the pseudorandom number generator **.**

`map` – the game map

`tracer` – used to record the flight trace and states.

public Position ***goNextPosition****()*

Move this drone to the next position. The drone decides which direction to go. Change the drone's position, reduce the power consumption and remaining steps. Each step consumes 1.25 unit power.

Returns: the position after move.

abstract Position ***findNextPosition****()*

Decide which position to go next. This abstract method should be implemented by inheriting classes depending on what strategy it adapts.

Returns: the position the drone decides to go.

public boolean ***hasNext****()*

Determine whether the drone has next step to go. If the drone has enough power and has steps remaining, returns true, otherwise, returns false

Returns: true means has next step, false means no next step.

boolean ***dangerous****(Position* `pos`*)*

Determine whether the given position will connect to a negative charger.
Parameters: `pos` – the position
Returns: true means dangerous position, false means safe position.

boolean ***isPositive****(Position* `position`*)*

Determine whether the given position will connect to a positive charger.
Parameters: `position` – the position
Returns: true means positive position, false means no positive charger connected.

boolean ***isPositive****(Charger* `charger`*)*

Determine whether the charger has positive coins and power.
Parameters: `charger` – the charger
Returns: true means positive charger, false means negative or empty charger.

public double ***currentCoins****()*

Return the coins of this drone.
Returns: the coins.

public double ***currentPower****()*

Return the power of this drone.
Returns: the power.

public class Stateless

extends Drone

The stateless drone class, inherits the abstract drone, and defines strategy of the stateless drone. Strategy of the stateless drone is go to next position randomly, but if it detected a nearby positive charger, fly to that charger. If it detected a negative charger, avoid it unless no other positions to go.

public ***Stateless****(Position* `startPosition`*, long* `seed`*, Map* `map`*, LineDrawer* `tracer`*)*

Constructs a stateless drone with given position, seed, map and tracer.
Parameters:
`startPosition` – the initial position of this drone.
`seed` – the seed used to initialize the pseudorandom number generator .
`map` – the game map
`tracer` – used to record the flight trace and states.

Position ***findNextPosition****()*

The stateless drone randomly iterates the positions of 16 directions. Once find a position connecting to a positive charger, return this position. If no position connects to positive chargers, return the first safe position. If all positions are dangerous, return the position with least damage. Any return position will within the play area.
Specified by: `findNextPosition` in class `Drone`
Returns: position the drone decides to go.

public class Stateful

extends Stateful

The stateful drone class, extends the stateless drone, and defines strategy of the stateful drone. The stateful drone adapts a greedy strategy. It always try to get the nearest positive charger. If there are any negative charger block the drone's way, the stateful drone will bypass negative chargers in clockwise or anti-clockwise direction. If one direction is not useful or the drone meet the border, it will try another direction.

public ***Stateful****(Position* `startPosition`*, long* `seed`*, Map* `map`*, LineDrawer* `tracer`*)*

Constructs a stateful drone with given position, seed, map and tracer.

Parameters:

`startPosition` – the initial position of this drone.

`seed` – the seed used to initialize the pseudorandom number generator .

`map` – the game map

`tracer` – used to record the flight trace and states.

Position ***findNextPosition****()*

Decide which position to go next using greedy strategy. If the drone can connect to a positive charger within 1 step, return that position connects to the charger. Otherwise the drone will go with the direction to the nearest positive charger. If the direction lead to a negative charger, rotate the direction until it lead to a safe or positive position.

Overrides: `findNextPosition` in class `Stateless`

Returns: position the drone decides to go.

public class App

Main procedure of the powergrab game.

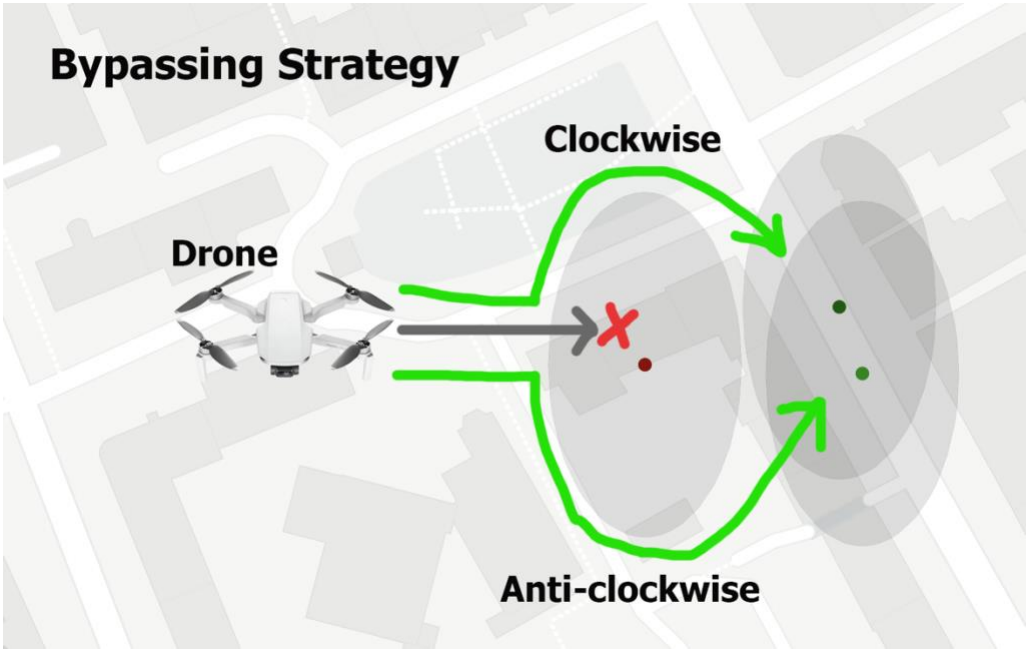
public static void ***main****(java.lang.*String[] `args`*)*

The program entry method.

3. Stateful Drone Strategy

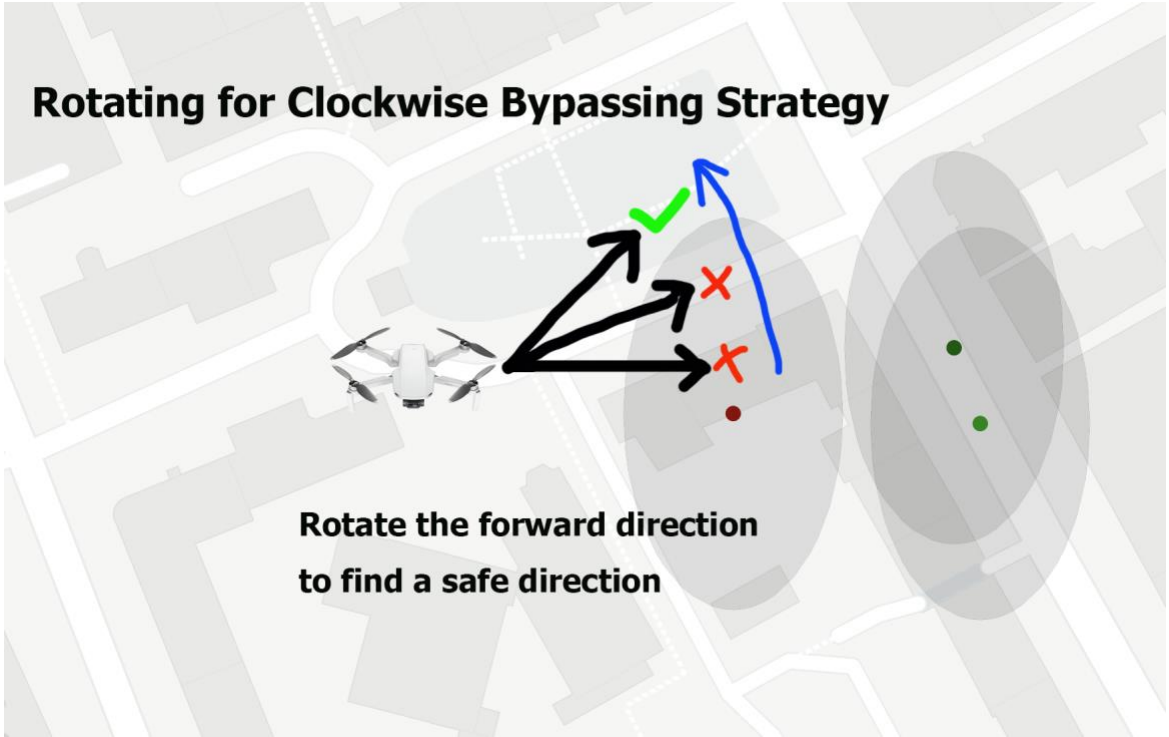
3.1 Common Strategy

The stateful drone adapts a greedy strategy. It always tries to get the nearest positive charger. To find the right direction to the next positive charger, the drone will first calculate the angle to that charger, then convert the angle to the closest direction. But this direction may lead the drone to a dangerous position, where the drone will connect to a negative charger or out of play area. Hence, the drone will try to bypass these barriers, in clockwise or anticlockwise direction.



Two direction to bypass barriers

To achieve this. The drone firstly try the best direction. If the best direction is dangerous, rotate it to next direction, if still dangerous, rotate to next ... until meet a safe direction.



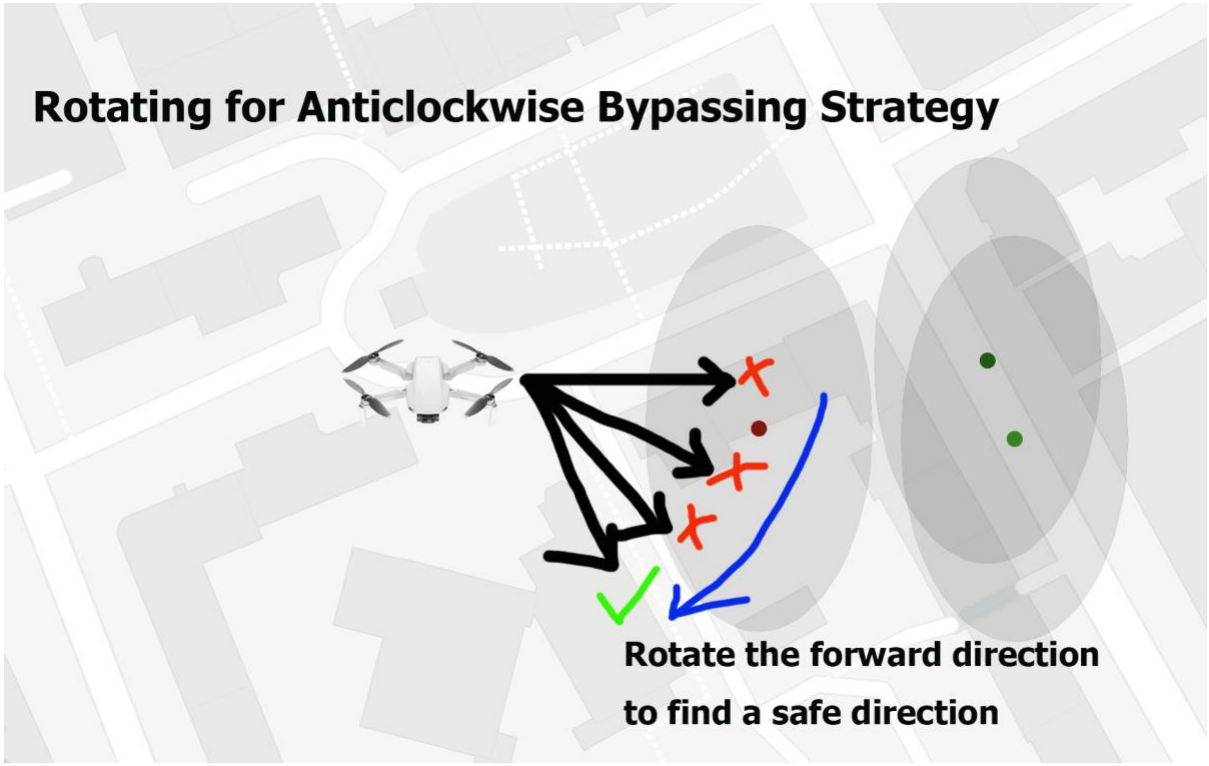
clockwise bypassing strategy rotation
anticlockwise bypassing strategy rotation

Only adapt one direction rotation will meet problems. There are some cases the drone need to change rotation direction. If the next safe direction is the last step position, or outside the boundary of play area, the drone will change the direction to rotate and repeat the procedure again until find a safe direction.
When there are no coins left in the current map, the stateful drone will always go to the previous step position for safe.

3.2 Strategy for Special Cases

When the drone is trapped, all 16 directions are dangerous, the drone will go to the position with least reduction of coins. This strategy is also adapted by stateless drone.

When the drone is trying to bypass the negative charger, both clockwise or anticlockwise direction will rotate to the previous position. In another words, the drone only have one way to go. In this case the common strategy is not effective because the drone will go to the previous position, then go to the same trapped position.

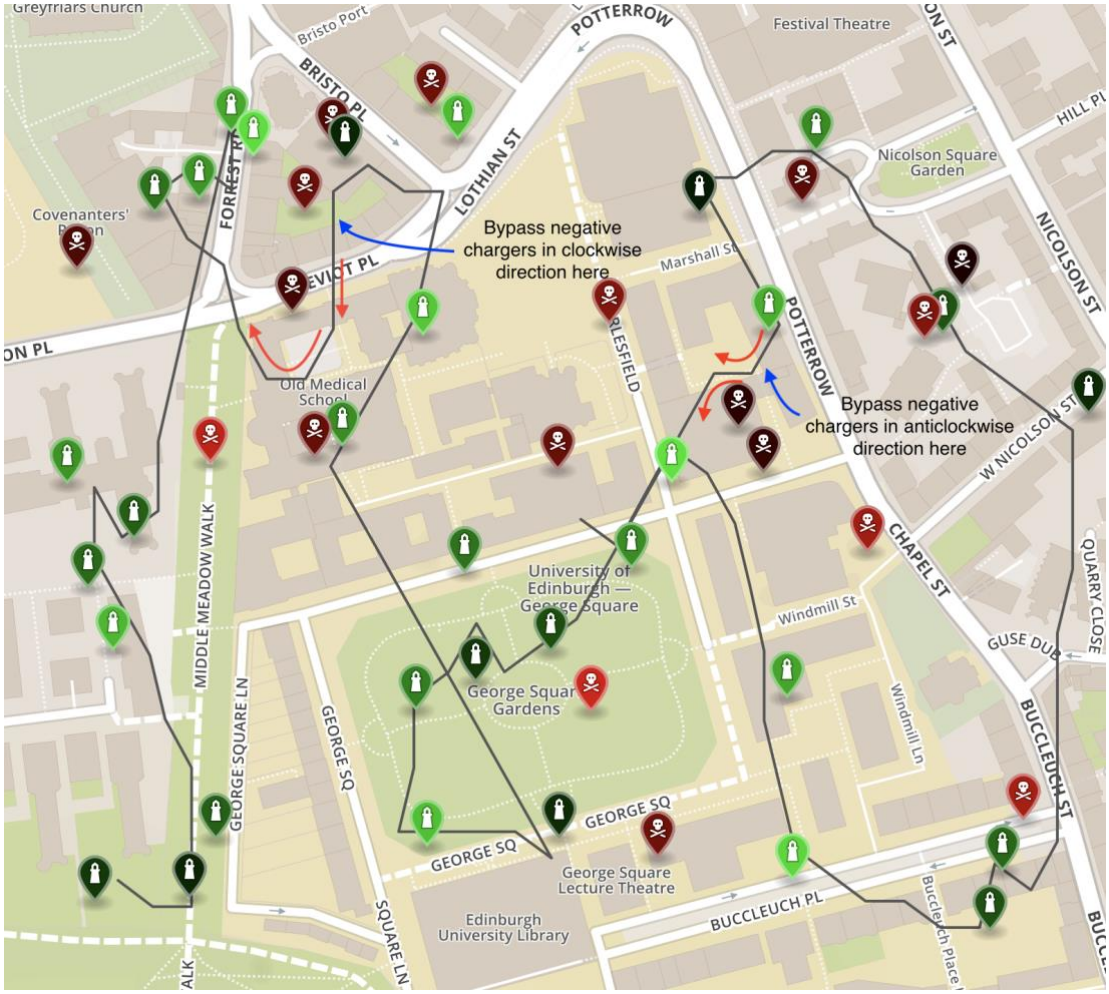


In this case the stateful drone will randomly run 10 steps using stateless drone strategy. So that the drone can get out of the trapping situation. This situation is very rare, I tested all 730 map but never meet one.

3.3 Strategy Demonstration

The stateful drone trace with initial parameters

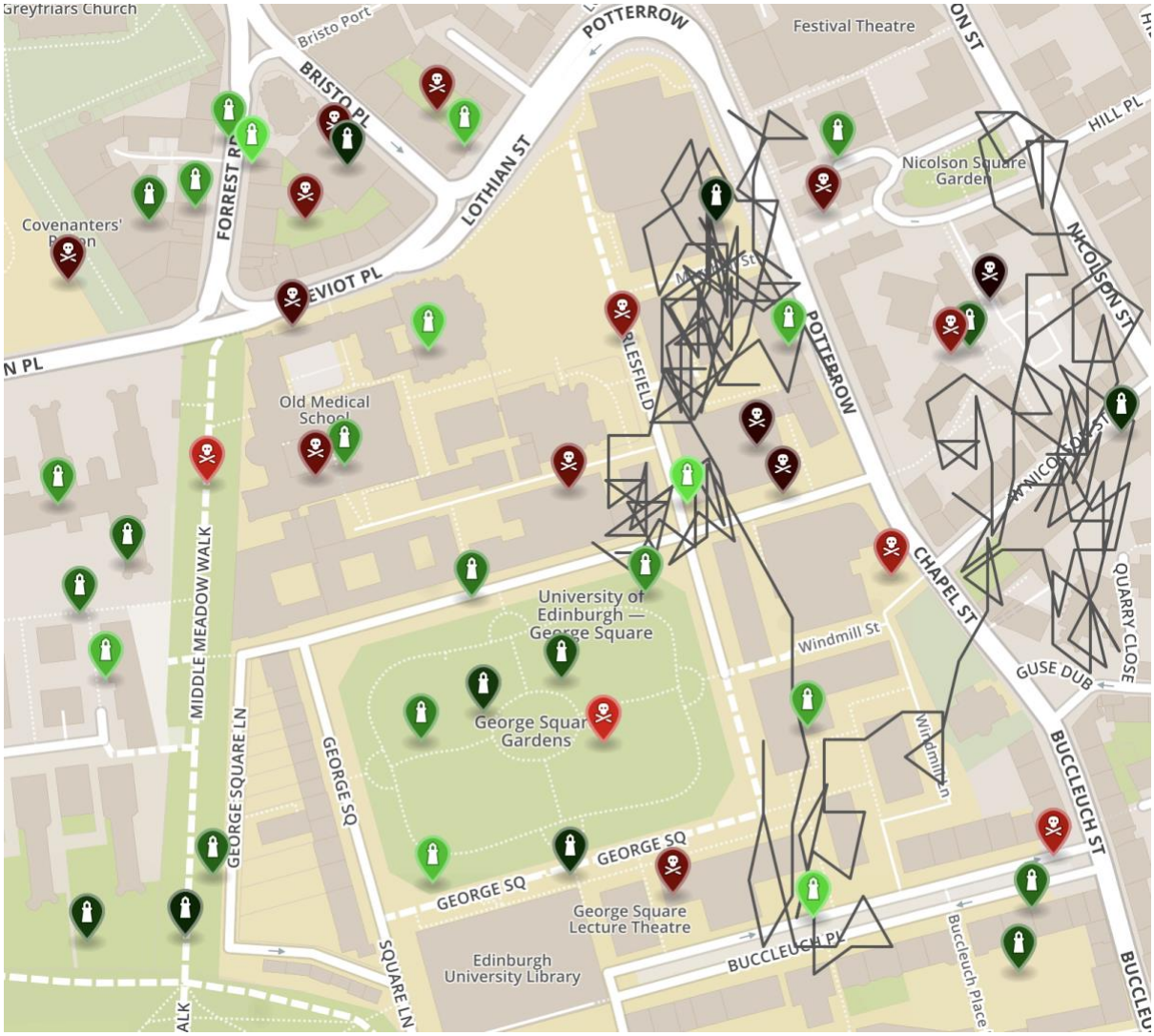
```
14 10 2019 55.944425 -3.188396 5678 stateful
```



Stateful drone demonstration

The stateless drone trace with initial parameters

```
14 10 2019 55.944425 -3.188396 5678 stateful
```



Stateless drone demonstration

3.4 Strategy Quality

The provided evaluator gives the results as below.

Minimum Ratio	100.00%
Median Ratio	100.00%
Average Ratio	100.00%
Maximum Elapsed	2.00
Median Elapsed	0.55
Average Elapsed	0.56

The stateful drone got all coins of 730 maps with initial position at (55.944425 -3.188396) and random seed 5678.