Informatics Large Practical 2019-2020

Coursework 2

1 Introduction

This project is to develop a simulation framework of a location-based strategy game called “Powergrab”. In this game, we are to develop two versions of autonomous drones to play against a human player. The first version is stateless which has limitations, suitable for novice player. The second version is stateful which has no limitations and is suitable for expert player.

2 Software Architecture Description

2.1 UML Class model

See figure 1 on page 2.

2.2 High Level Description

• DroneStrategy is an interface. Any class for a particular drone strategy which implements this interface should be able to run and rerun the simulation, print the result and save the result to a specified “.txt” file as well as a “.geojson” file.

• Two kinds of stateful drone strategy are created in this project - StatefulNaive and StatefulACO. The final output files of the stateful drone are produced by StatefulACO. Both strategies have excellent performance(100% coin rate), and they are compared in part 3. We keep StatefulNaive as an alternative choice.

• A object of Drone records its flightpath in the track property so that we can print it on the map. This memory can not be used in a Stateless object. The log property of Drone records the state of the drone at each move in a format as *.txt output files required.

• The datatype of latitude, longitude, coins and power depends on the data read from map source. However it is also a trade-off between precision and speed, so they may be changed to float type to get a higher speed.

• The properties and methods of AntColonyOpmization and Ant are omit because they are relatively a independent part working as an algorithm for StatefulACO. For more information about these two classes, see part 2 and part 3.

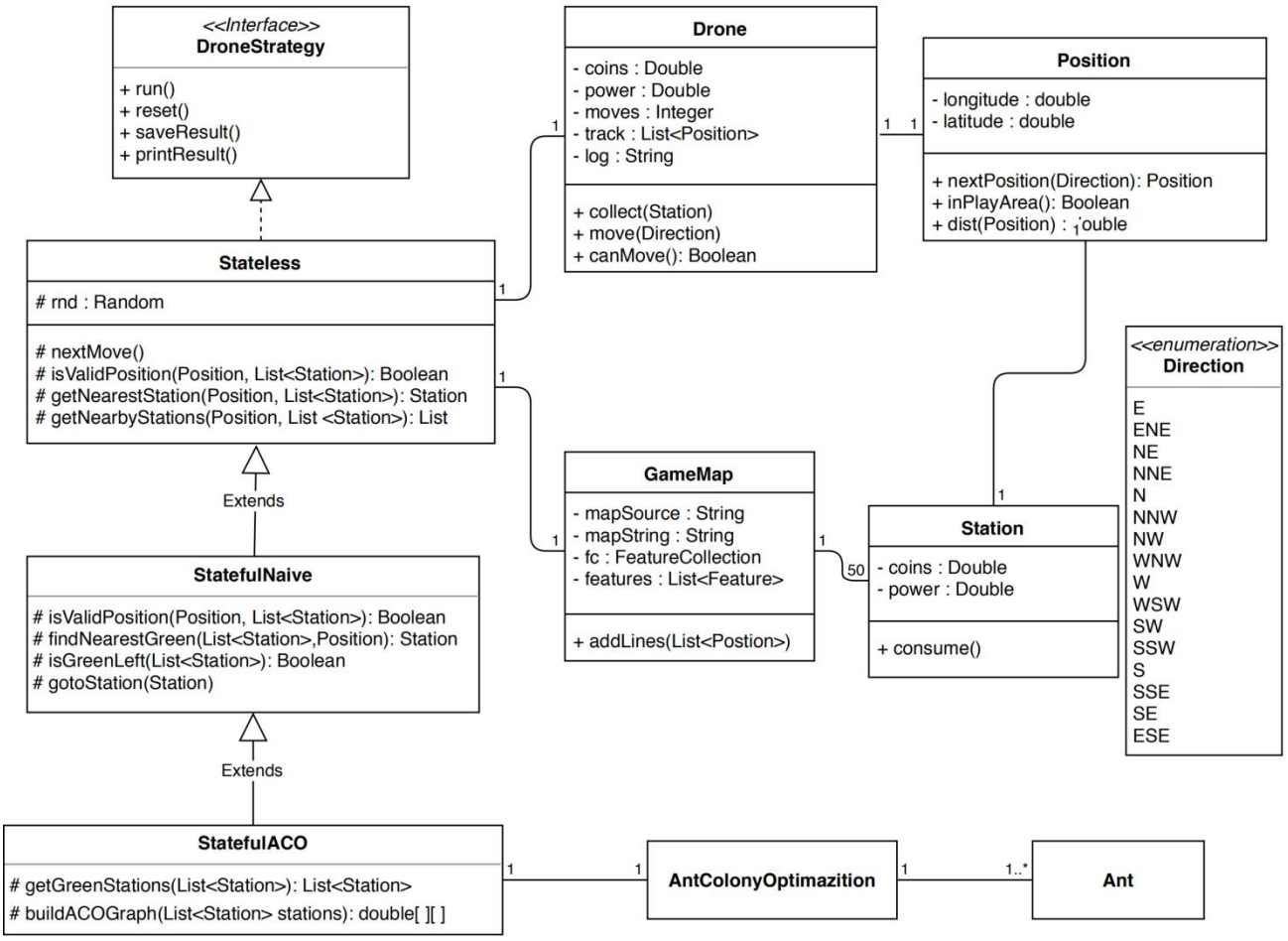

Figure1 : UML Class Diagram

3 Class Documentation

Class

3.1 Position

Description

* Represents a position on the map with a latitude and a longitude.
* Note that all positions are treated as they were on a plane instead of a sphere.
* This class also predefines the border of the map.

Properties

static final public double ***leftBorder, rightBorder, topBorder, downBorder***: These constants defines the PowerGrab playing area. All points on every map are within this area. Note that the drone must not fly out of this area. The borders are given by latitudes and longitudes based on a real world map.

static final public double ***moveUnitDistance*** = 0.0003: The drone travels a unit distance every time, which is 0.0003 degrees. Note that due to considering small changes if longitude or latitude, we approximate the earth's surface as plane.

Constructor

Position(**double** latitude, **double** longitude)

Constructor, the initial location is always required.

Methods

public Position nextPosition(Direction direction)

* Finds the next position of the drone according to the current position and the specified * direction.

* **@param** direction the direction that the drone flies in.

* **@return** the next position of the drone.

public boolean inPlayArea()

Checks if the object is in the play area.

@return Returns true only if the object is in the play area. Note that the method will return false if the object is right on the border.

public Double dist(Position pos)

Calculates the Euclidean distance between position 1 and position 2. Note that it is a static method.

@return the distance between this position and pos

3.2 Station

Description

* Represents a station that is automatically linked to drone when the drone is close enough. Station
* can provide power and coins to the drone however station can have negative coins and power.
* A drone can only connect to at most one station after each move, which is the nearest station.

Properties

private Double coins: The amount of coins in the station.
private Double power: The amount of power in the station.
private Position position: The position of the station, should never be changed.

Constructor
public Station(Double coins, Double power, Position position)
* Same as the drone, the position of the station must be in the play area.

Methods
public void consume()
* When a station is consumed, the coins and power of it is set to 0.

3.3 Drone
Description
* Represents the autonomous drone in the game. A drone object can be driven by an
* object of class that implements the DroneStrategy interface.

Properties
private Double coins: Total coins the drone has collected, including negative coins.
private Double power: Amount of power the drone has for now.
private Integer moves: Total number of moves the drone has made.
public Position position: Current Position of the drone.
private List<Position> track: Records the position at each move plus initial position
private String log = **null:** Records the information in coinsRecord, powerRecord and track in string format.
private final static double *moveUnitPower* = 1.25: The amount of power consumed at each move.
private final static int *maxMoveNumber* = 250: The maxmium number of moves the drone can make.

Constructor
public Drone(Position position)
* A drone is initialized with 250 units of power, 0 coin and a initial position. Note that * the initial position must be in the play area.
* **@param** position the initial position of the drone.

Methods
public void collect(Station station)
* Let the drone collects coins and power from a specified station. Coins and power in the station *could be negative. Coins and power of that station are set to 0 when they are collected.
* **@param** station the station linking with the drone.

public boolean canMove()
* Finds out whether the drone can continue to move. If both moves and
* power are left, the drone can continue to move.

* @return Returns true if the drone can continue to move.
public void move(Direction direction)
* Moves the drone towards a specified direction.
public void record()
* Records the current state of the drone. Note that track records the flight path of the drone, and
* log records all information required for the output *.txt files.
* Note that log is written in a specified format:
* last position, direction, coins, power, current position

3.4GameMap

Description

* Represents the map of the game. Maps are always download via Url address.
* You need a valid date to download a map.

Properties

private String mapSource: The original map source string in Geojson format.
private List<Station> stations: All stations in the map.

Constructor

public GameMap(String dd, String mm, String yyyy)
* Extract informations and create a game map object by using the date of map
* to find the original map source.

Methods

public void addLines(List<Position> lines)
* Adds lines to the map. This function is used to add flightpath to the map.
public List<Station> getStations()
* Returns a copy of the stations of the map. It means that the stations property can not be changed
* from outside. If you want to change them, just return the property stations itself.
* @return a copy of the stations property

3.5 Stateless implements DroneStrategy

Description

* Realize the stateless drone strategy. All stateful classes should extend this class
* and base on this class.
* A stateless drone is limited and designed for novice players. It is memoryless, and
* only watches stations that can be reached within one step instead of the whole map.

Properties

protected Drone drone: The drone which is manipulated using this strategy.
protected Random rnd: Random number generator, always using the same seed so that the result can be reproduced.
protected GameMap theMap: The map which the drone flies on.
protected final static double *chargeRange* = 0.0002: If the distance of the drone and station is less

than charge range, the drone is able to collect coins and power from that station instantly.

Constructor

```
public Stateless(Position position, Integer seed, GameMap theMap)
* @param position The initial position of the drone.
* @param seed The seed used to initialize random number generator.
* @param theMap The map this simulation running on.
```

Methods

```
public void run()
* Moves the drone step by step until it can not move. Save the track of the drone. Nothing will
*happen if you call this method again without resetting the simulator.
public void reset()
* Resets the simulator. Apply run() method again will get same result.
public void printResult()
* Prints the simulation result on console.
public void saveResult()
* Save result to files.
protected void nextMove()
* Decides the next move of the drone and then move it. If the drone doesn't have nearby
* green stations(stations with positive coins and power), it will move completely randomly.
* Else it will move to the green station that can be reached within one step and collects
* coins and power from that station.
protected boolean isValidPosition(Position pos, List<Station> stations)
* If at such a position the drone won't collect negative coins and power from a station
* and won't go out the play area, then this position is valid.
* @param pos Position of the drone
* @param stations Stations on the map or just stations nearby the drone
* @return Returns true if it is safe to move to this position

protected Station getNearestValidStation(Position currentPos, List<Station> stations)
* Gets the station which is nearest to the drone and within charge range. Returns null if there is no
* such station.
* @param currentPos Current position of the drone
* @param stations All stations on the map or just stations nearby the drone
* @return Returns the station which is nearest to the drone and within charge
* range. Returns null if there is no such station.

protected List <Station> getNearbyStations(Position position, List <Station> stations)
* Find all position that the drone may have chance to get charged from within one move.
* This kind of stations are called nearby stations.
* @param position Current position of the drone.
* @param stations All stations on the map.
```

3.6 StatefulNaive extends Stateless implements DroneStrategy

Description

* Realizes a kind of stateful drone strategy via simple greedy strategy.
* It is a simple and natural version of stateful strategy, class that implements more complex stateful
* strategy may extend this class.
*
* A stateful drone has memory and should perform as good as possible to beat expert human
* players.
*
* This stateful drone records its former moves to avoid arriving at same position multiple times, and
* it always heads to the nearest green station.
* (Green stations means it has positive coins and power)

Methods

```
public void run()
```
* Applying greedy strategy, let the the drone always move towards the nearest green station until
* no green left. Call same method in Stateless class when it collects coins and power from all green
* stations.

```
protected boolean isValidPosition(Position pos, List<Station> stations)
```
* Rewrites the method of Stateless, adds a new function - prevents the drone from going back to
* last position.

```
protected Station findNearestGreen(Position dronePos, List<Station> stations)
```
* Finds the nearest green station to the drone.
* @param dronePos Current position of the drone
* @param stations At least contains all green stations on the map

```
protected boolean isGreenLeft(List<Station> stations)
```
* Finds out if there is any green station left.
* @param stations All stations on the map.
* @return Returns true if there is at least one green station left.

```
protected void gotoStation(Station nextStation)
```
* Keeps moving the drone until is reaches within charge range to the
* next station and is connected to this station.
* @param nextStation The next station you want the drone to move towards.

3.7 StatefulACO extends StatefulNaive implements DroneStrategy

Description

* Realizes a kind of stateful drone strategy via ant colony optimization algorithm.
*
* This stateful drone will first use this algorithm to design an good order to traverse
* all green stations(stations with positive coins and power), and then move the drone
* to these stations one by one.

Methods

public void run()

* Finds an order of all green stations using Ant Colony Optimization and then move the drone
* towards them one by one.

protected List<Station> getGreenStations(List<Station> stations)

* Finds all green stations on the map.
* **@param** stations At least contains all green stations on the map.
* **@return** Returns a list of stations contains all green stations on the map.

protected double[][] buildACOGraph(List<Station> stations)

* Builds a graph that records the distance between each pair of
* the stations in the given list plus the position of the drone.
* **@param** stations A list of stations.
* **@return** Returns 2-dimension array representing the graph.

3.8 AntColonyOptimization

Description

* This class implements an algorithm which can be used to find a short path
* for traversing a graph. In this project it is used to find a short path
* to traverse all green stations on the map.

Properties

private double c = 1.0: Parameter c indicates the original number of trails, at the start of the simulation.

private double alpha = 1: Alpha controls the pheromone importance. In general, the beta parameter should be greater than alpha for the best results.

private double beta = 5: Beta variable controls the distance priority.

private double evaporation = 0.6: the evaporation variable shows the percent how much the pheromone is evaporating in every iteration.

private double Q = 500: Q provides information about the total amount of pheromone left on the trail by each Ant.

private double antFactor = 0.8: AntFactor defines how many ants we will use for each city.

private double randomFactor = 0.01: We need some randomness, but not much.

private int maxIterations = 1000: Defines how many iterations we use until get the final result.

Methods

public int[] solve()

* Runs the main logic to find the best tour order.

3.9Ant

Description

* This class defines an abstract ant used in ant colony optimization.

Interface

3.10 DroneStrategy

Description

* An interface that all kinds of stateful or stateless strategy should implement.
* Contains basic public method for a drone strategy class.

Methods

```
void run()
* Run the simulation on map, exits when the drone can not move.
void reset()
* Reset the drone and clear all simulation results.
void saveResult()
* Save the simulation result into two files. A .geojson file saves original map and flightpath, and
* a .txt file save the log of the drone.
void printResult()
* Prints the result to console.
```

Enumeration

3.11 Direction

Description

* This class defines a total number of 16 directions that the drone is able to fly in. The drone cannot
*fly in other directions.

Member

```
E,ENE,NE,NNE,N,NNW,NW,WNW,W,WSW,SW,SSW,S,SSE,SE,ESE
```

4 Stateful Drone Strategy

4.1 Salesman Problem(TSP)

The salesman problem asks the following question: “Given a list of cities and the distance between each pair of cities, find the shortest route to visit every city and then go back to the original city.” This is an NP- hard problem in combinatorial optimization.

In Powergrab game, if we just look at all green stations first, to win the game, the drone need to collect all coins from greens stations. However, the power and steps can make are limited, so we have to choose a route as short as possible to traverse all green stations. Then we have a problem similar to the Salesman Problem. There are some differences though - the drone can only move towards limited directions, the stations are not a point but a circle(have charge range), and we don’t need the drone to return the initial position. Despite differences, we can still make approximations and first treat the stateful drone strategy as a salesman problem.

4.2 Brute force

It is easy to think of an idea to find all possible ways to visit all stations, but that will have exponential time complexity, so we cannot use brute force.

4.3 Greedy

It is natural to try a greedy strategy on this problem. That is, we always let the drone move towards a nearest station until the drone is connected with that station. So we also call the nearest green station the target station. This strategy is implemented by the **StatefulNaive** class.

Then we need to take red stations into consider, because we must never have the drone connect with a red station. As the drone can move towards 16 directions, moving towards one of them will be closest to the target green station, but if that move will lead the drone to connect with a read station, we choose the second closest direction and so on.

4.4 Trap

If we only apply the logic in 4.3 to avoid red stations, the drone may get trapped in some special situation.

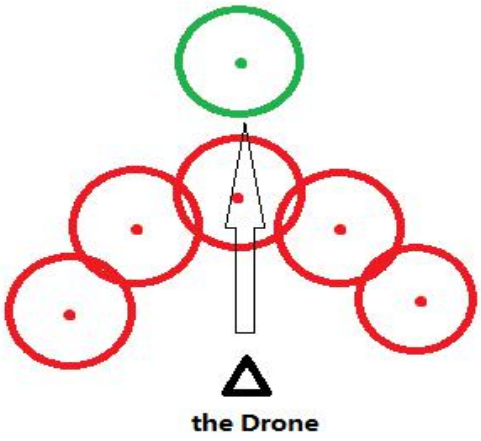

figure 2 special situation

As figure 2 shows, if we always choose the closest possible direction, the drone may get trapped at a corner formed by several red stations. To solve this special case, we let the drone move several random steps(avoid red stations) when it detects it is getting trapped.

4.5 Heuristic Algorithm - Ant Colony Optimization

The Ant Colony Optimization algorithm can be used for finding good paths through graphs, and it can apply on the Salesman Problem. An ant tries to find optimal solution by moving through a parameter space representing all possible solutions. Like real ants, the abstract 'ants' similarly record their positions and the quality of their solutions, so that in later simulation iterations more ants find better solutions. Those records are called pheromones.

The stateful strategy using ACO is implemented by class **StatefulACO**, **AntColonyOptimization** and

Ant. We trained some key parameters to get better result:

AntFactor : for n stations, we use (antFactor * n) ants.
Evaporation: defines how much percent of pheromone is evaporating every iteration.

4.6 Evaluating Results

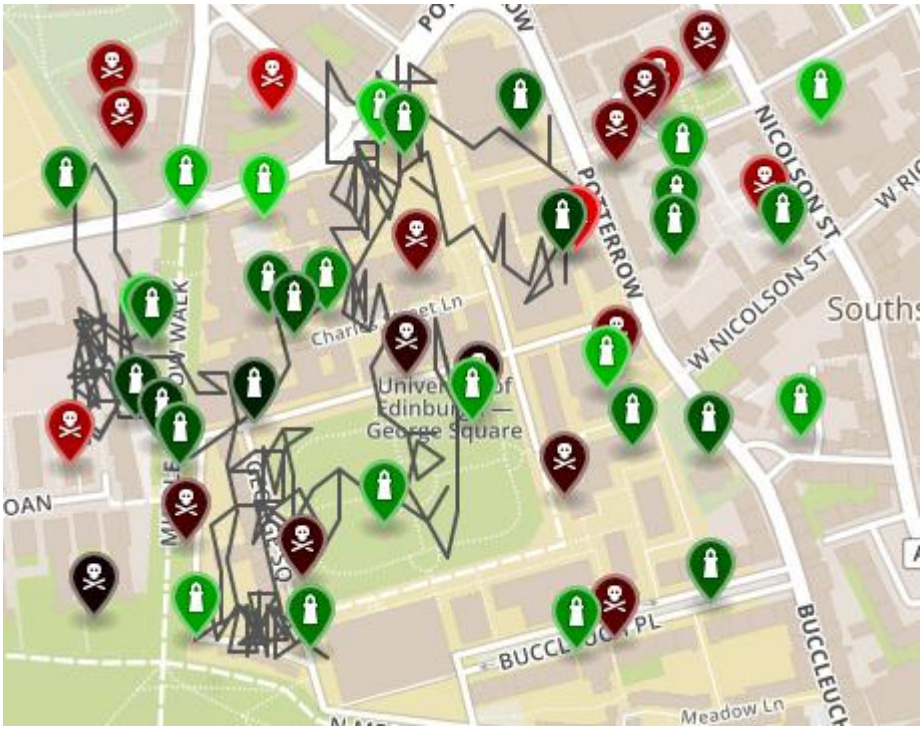

Figure 3 stateless drone flightpath on map 01-01-2019

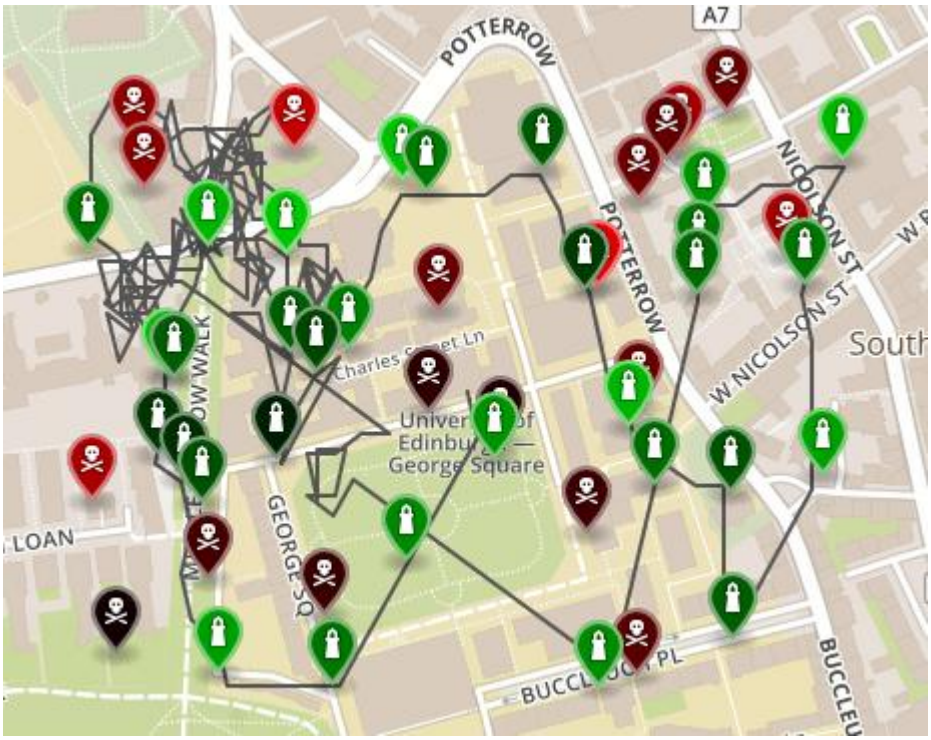

Figure 4 greedy strategy drone flightpath on map 01-01-2019

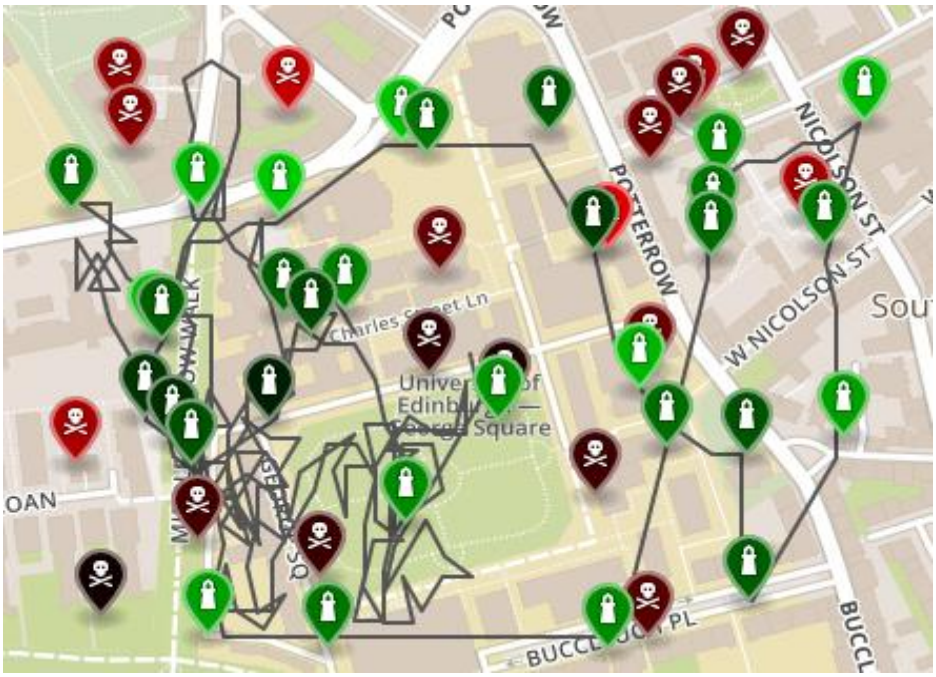

Figure 5 ACO strategy drone flightpath on map 01-01-2019

Strategy\result	Average coin ratio	Average elapsed	Average moves
greedy	100%	0.61	108
ACO	100%	0.69	94

*Average moves: how many moves the drone made to collect all coins from the map.
The chart show that the greedy strategy runs faster while the ACO strategy collect all coins faster - Because it gets a shorter route. So which stateful strategy to choose is a trade-off between speed and short route.