Informatics Large Practical 2019-20 - Coursework 2

Implementing an Autonomous Agent to play against a Human Agent in a location-based Strategy Game

Author: Valentine Dragan s1710228

December 2019

Contents

1 Introduction **2**

2 Software architecture description **2**
2.1 Class Diagram . . . 3
2.2 Class Hierarchical Relationships . . . 4

3 Class Documentation **4**
3.0.1 App . . . 4
3.0.2 InputOutputManager . . . 5
3.0.3 GameManager . . . 5
3.0.4 Map . . . 6
3.0.5 MapFunctions . . . 6
3.0.6 Drone . . . 6
3.0.7 DroneLogic . . . 7
3.0.8 StatelessLogic . . . 7
3.0.9 StatelessDrone . . . 7
3.0.10 StatefulLogic . . . 8
3.0.11 StatefulDrone . . . 8
3.0.12 SortByMoney . . . 9
3.0.13 Node . . . 9
3.0.14 Station . . . 9
3.0.15 Position . . . 10
3.0.16 Direction . . . 10
3.0.17 DistanceComparator . . . 10
3.0.18 Debugger . . . 10

4 Stateful Drone Strategy **11**
4.1 Similarity with Travelling Salesman Problem . . . 11
4.2 The Strategy . . . 11
4.2.1 Description . . . 11
4.2.2 Step by Step Implementation . . . 12
4.3 Results . . . 13
4.4 Further Improvements . . . 13

5 Personal Conclusions **14**

6 References **15**

1 Introduction

This document presents the design and implementation of the PowerGrab game framework, in which players (human or computer) compete in controlling a Drone around a Map in order to collect the most coins from randomly located Stations. A full description of the PowerGrab game, rules and specifications can be found in the *PowerGrab project specifications.pdf* document.

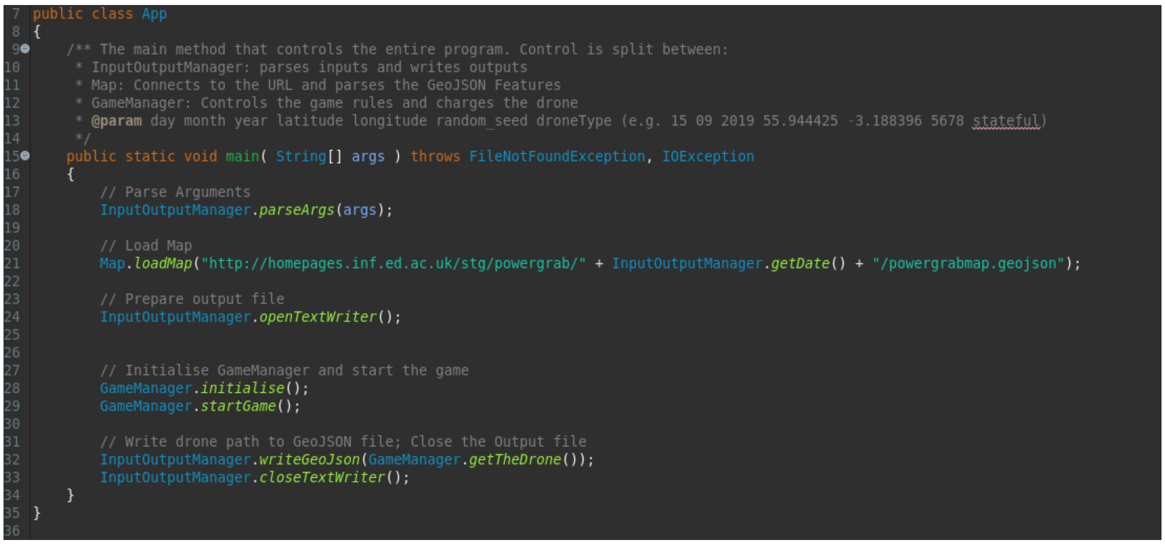
Moreover, the document presents the implementation of two versions of an Autonomous Agent (computer player) which plays against the human player. The first version, referred to as Stateless Drone, is appropriate for playing against a novice human player. The second version, referred to as Stateful Drone, is appropriate for playing against an expert human player.

The project is available for use on GitHub: https://github.com/ValntinDragan/PowerGrab.git (Will be Private until the coursework marks are released)

This document captures the Software Architecture, Class Documentation and Implementation of the Stateful Drone, which should be read in order to understand and maintain the application.

2 Software architecture description

For this project I have decided to use the **Separation of Concerns**[1] design principle: separating the program into distinct sections such that each section addresses a separate concern. This principle was first introduced to me in the course *Informatics 2C: Software Engineering at The University of Edinburgh.* This design concept provides several benefits such as: making the code easier to read, modify and test.



```java
public class App
{
    /** The main method that controls the entire program. Control is split between:
     * InputOutputManager: parses inputs and writes outputs
     * Map: Connects to the URL and parses the GeoJSON Features
     * GameManager: Controls the game rules and charges the drone
     * @param day month year latitude longitude random_seed droneType (e.g. 15 09 2019 55.944425 -3.188396 5678 stateful)
     */
    public static void main( String[] args ) throws FileNotFoundException, IOException
    {
        // Parse Arguments
        InputOutputManager.parseArgs(args);

        // Load Map
        Map.loadMap("http://homepages.inf.ed.ac.uk/stg/powergrab/" + InputOutputManager.getDate() + "/powergrabmap.geojson");

        // Prepare output file
        InputOutputManager.openTextWriter();


        // Initialise GameManager and start the game
        GameManager.initialise();
        GameManager.startGame();

        // Write drone path to GeoJSON file; Close the Output file
        InputOutputManager.writeGeoJson(GameManager.getTheDrone());
        InputOutputManager.closeTextWriter();
    }
}
```

Figure 1: Example: easy to read code in the App.main() because of Separation of Concerns

Following this principle, I have identified 3 main areas of my program, each separated further by distinct concerns. 15 Classes and 1 Interface were created in order to implement the project in a clean and accurate manner. The software architecture can be described as follows:

- Control of the Program
  - **App**: responsible for initialising and coordinating all other Control classes
  - **InputOutputManager**: handling input/output operations
  - **Map**: loading the map features, storing and updating stations
  - **GameManager**: playing the game, checking the rules, responding to the drone's moves

- Drone Components
  - **Drone**: containing the drone's variables (model)
  - **DroneLogic**: interface describing the drone's decision-making process
  - **StatelessLogic** and **StatefulLogic** implementations of the Logic
  - **StatelessDrone** and **StatefulDrone** extensions of the Drone
- Spatial Features
  - **Station**: storing information about the stations on the Map
  - **Position**: describing the geographical coordinates of Drone and Stations on the Map
  - **Direction**: enumerating the 16 possible directions in which the Drone can move
  - **MapFunctions**: providing useful methods for calculating distances and directions

  Additionally, the **Debugger** class contains several useful methods for debugging the project. This class should not be added to a public release, but is very useful for debugging the code in a clean way.

2.1 Class Diagram

The figure below presents the Class Diagram of the application. As it has been described above, each class addresses one of the 3 distinct sections: *Control of the Program*, *Drone Components* or *Spatial Features*. A high-level description of the classes is presented in the next section.

For ease of visualisation, some of the relationships have been truncated (i.e. the Drone, Station and Node all have relationships with the Position class).

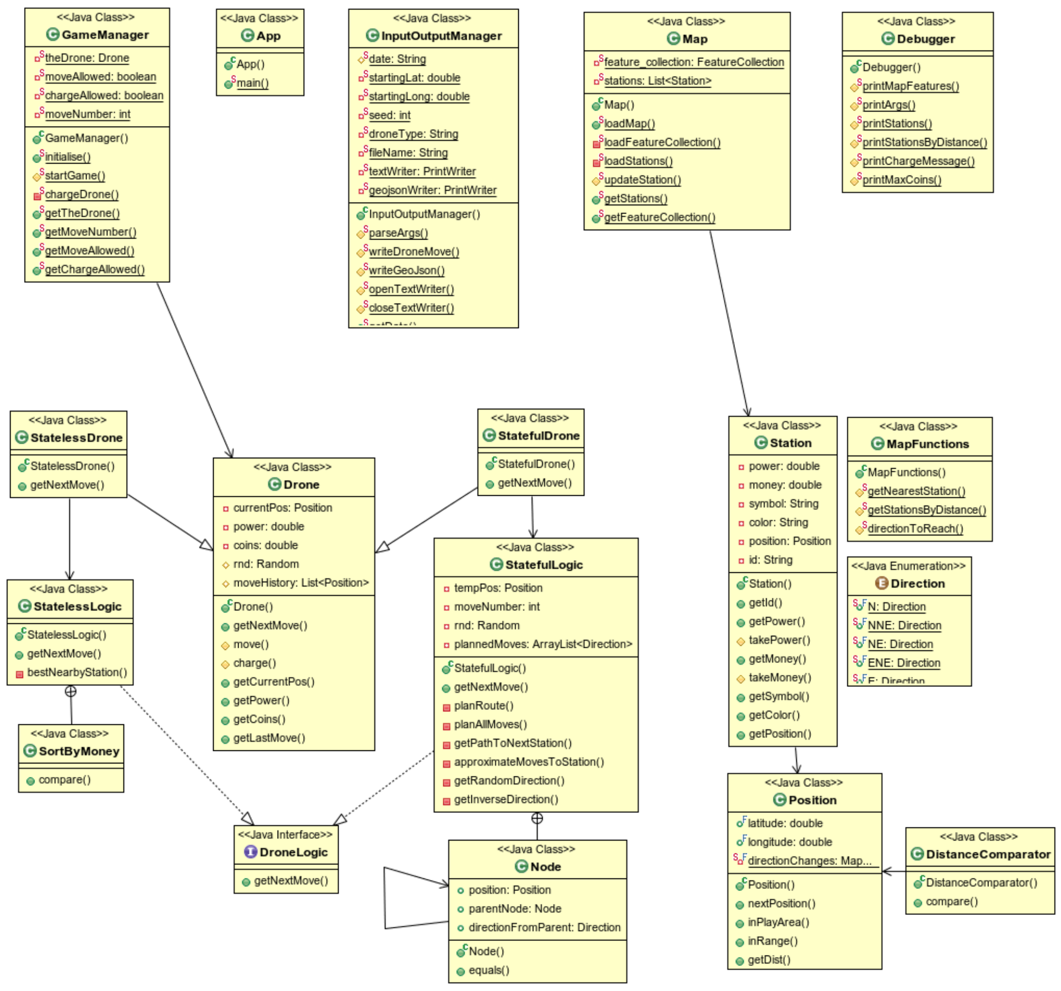


Figure 2: UML Class diagram

2.2 Class Hierarchical Relationships

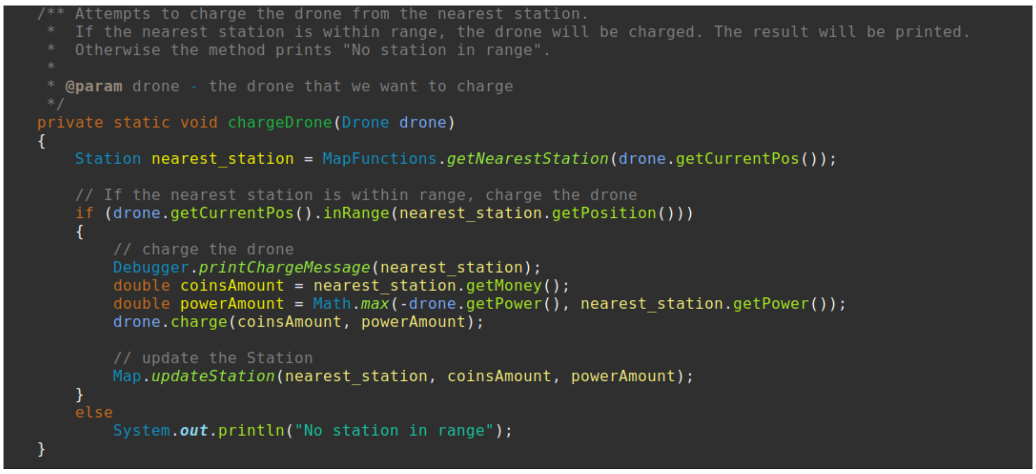
Drone is the parent-class of the **StatelessDrone** and **StatefulDrone**. This is because both Stateless and Stateful drones have similar fields (e.g. current position, power, coins) and similar functionality (e.g. move, charge). It is therefore appropriate to define a parent-class which contains the common information, and then create Stateless and Stateful sub-classes of it with specific modifications.

StatelessLogic and **StatefulLogic** are implementations of the **DroneLogic** Interface. Since both the Stateless and Stateful drone have the same decision goal: finding the best move, it is appropriate to create an Interface for the drone's logical component, and then provide separate Stateless and Stateful implementations of it.

3 Class Documentation

This section shows the definitions of class members and how to use them. It presents method signatures and descriptions, but not their internal implementations. It includes links between classes so that you can rapidly follow the logic of the code.

For code implementations, please open the .java class files. Each method contains comments and JavaDoc which should help in understanding how each method operates.



```java
/** Attempts to charge the drone from the nearest station.
 *  If the nearest station is within range, the drone will be charged. The result will be printed.
 *  Otherwise the method prints "No station in range".
 *
 * @param drone - the drone that we want to charge
 */
private static void chargeDrone(Drone drone)
{
    Station nearest_station = MapFunctions.getNearestStation(drone.getCurrentPos());

    // If the nearest station is within range, charge the drone
    if (drone.getCurrentPos().inRange(nearest_station.getPosition()))
    {
        // charge the drone
        Debugger.printChargeMessage(nearest_station);
        double coinsAmount = nearest_station.getMoney();
        double powerAmount = Math.max(-drone.getPower(), nearest_station.getPower());
        drone.charge(coinsAmount, powerAmount);

        // update the Station
        Map.updateStation(nearest_station, coinsAmount, powerAmount);
    }
    else
        System.out.println("No station in range");
}
```

Figure 3: Example of JavaDoc contained in the code

3.0.1 App

Description: This class is the main *Control Component* which starts and manages the entire program. Control is further divided between InputOutputManager, Map and GameManager.

Methods:

- **main(String[] args)**: Manages and runs the entire program
  1. Tells *InputOutputManager* to parse the input arguments.
  2. Tells the *Map* to read the GeoJSON and load all the Stations.
  3. Tells *InputOutputManager* to open the text file.
  4. Initialises the GameManager which initialises the *Drone* and starts the game.
  5. Tells *InputOutputManager* to draw the drone path.

3.0.2 InputOutputManager

Description: This class is one of the *Control Components*, where input and output messages are directed to. The inputs are the arguments entered by the *User* when starting the App. The outputs are the Drone's moves and path written down in the text and geojson files.

Fields:

- *date*: the date of the generated Map
- *startingLat* and *startingLong*: the *Drone*'s starting position
- *seed*: the seed of the Random Number Generator used by the *Drone*
- *droneType*: the type of the *Drone*
- *fileName*: the name of the text and geojson files
- *textWriter*: the PrintWriter object used for writing in the text file
- *geojsonWriter* : the PrintWriter object used for writing in the geojson file

Methods:

- **parseArgs(String args[])** used to parse the user inputs into the fields mentioned above, which are necessary for running the rest of the *Control Classes*.
- **openTextWriter** is called before writing the drone moves in the text file.
- **closeTextWriter** is called after all the moves have been written in the text file.
- **writeDroneMove(Drone drone, Direction direction)** is used to write down a *Drone*'s move in the text file. This method is called by the GameManager after the drone executes a move.

- **writeGeoJson(Drone drone)** is called by the App after the game has ended to write down the drone's path. The Drone's *moveHistory* is transformed into a geojson feature and appended to the Map's $feature_collection$.

3.0.3 GameManager

Description: This class is one of the Control Components, whose purpose is to start the game, interact with the Drone and respond to its moves.

Fields:

- *theDrone*: the Drone (Stateless or Stateful) which is used for playing the game
- *moveNumber*: keeps track of what move the *Drone* is at
- *moveAllowed*: represents whether or not the *Drone* is allowed to *move()*. This ensures that the *Drone* doesn't try to cheat.
- *chargeAllowed*: represents whether or not the *Drone* is allowed to *charge()*. This ensures that the *Drone* doesn't try to cheat.

Methods:

- **initialise()** is used for initialising *theDrone* field and inspecting the starting Position.
- **startGame()** is used to start playing the *Drone*'s game. Each turn, the method will:
  1. Get the Drone's next move by calling its *getNextMove()* method
  2. Move the drone in the specified direction
  3. Charge from the nearest station, if within range
  4. Write down the move in the output file

  (until *moveNumber* reaches 250, or *theDrone* runs out of power):

3.0.4 Map

Description: This class is one of the *Control Components*, responsible for loading the features from the GeoJSON website, storing and updating the Stations.

Fields:

- *feature_collection*: the *FeatureCollection* representing the GeoJSON code pulled from the url.
- *stations*: the *List* of Stations present on the map.

Methods:

- **loadMap(String urlString)**: calls the next two methods described below, which extract the GeoJSON from the website, load it into the *feature_collection* and initialise the *stations*.
- **loadFeatureCollection(String mapString)**: connects to the specified URL and extracts the features into the *feature_collection* field.
- **loadStations()**: returns a *List* of all the Stations on the map.
- **updateStation(Station station, double coinsAmount, double powerAmount)**: is used to update a *Station*'s fields after the *Drone* has charged from it. This method is called by the GameManager after charging the Drone.

3.0.5 MapFunctions

Description: This is a singleton class which provides useful methods for calculating distances and directions.

Methods:

- **getNearestStation(Position origin)**: returns the nearest Station from the origin. This method is used by the GameManager to determine the *Station* that's closest to the Drone.
- **getStationsByDistance(Position origin)**: returns a *List* of all Stations ordered by distance from the origin. This method makes use of the DistanceComparator.
- **directionToReach(Position origin, Position destination)**: return the Direction which would bring the origin Position (e.g. the *Drone's position*) within range of the destination (e.g. a *Station's position*). This is used by the StatelessLogic to determine which *Direction* reaches the nearest positive station.

3.0.6 Drone

Description: This is a Parent-class which holds information about a drone's status and basic functionality. This class is essential for representing the drone's model.

Fields:

- *currentPos*: the drone's current Position on the Map
- *power*: the drone's accumulated power
- *coins*: the drone's accumulated coins
- *rnd*: the Random Number Generator with the seed corresponding to the *User*'s input
- *moveHistory*: an *ArrayList* of Positions representing the *Drone*'s *Position* after each move

Methods:

- **getNextMove()**: this method is designed to return the Direction where the drone wants to go. The method will be called by the GameManager each turn in order to move and charge the drone. The method will be overridden by the StatefulDrone and StatelessDrone which use their own algorithms.

- **move(Direction direction)**: moves the drone in the specified direction by updating *currentPos* and consuming 1.25 *power*. Additionally, adds the move to *moveHistory*. The drone can only move when allowed by the GameManager (indicated by *GameManager.moveAllowed)* and can't move outside the play area.
- **charge(double coinsAmount, double powerAmount)**: charges the drone with *coins* and *power*. The drone can only charge when the GameManager allows it (indicated by GameManager.chargeAllowed).

3.0.7 DroneLogic

Description: This is an interface representing the Drone's decision-making algorithm. Stateful and Stateless implementations of this interface are described below.

Methods:

- **getNextMove(Position currentPos)**: should return the Direction where the *Drone* wants to move, based on its current Position.

3.0.8 StatelessLogic

Description: This class is the stateless implementation of DroneLogic, and thus represents the StatelessDrone's decision-making algorithm. It has no memory and thus only makes decisions based on what's around it.

Fields:

- *rnd*: The Drone's Random Number Generator

Methods:

- **getNextMove(Position currentPos)** returns the Direction that the stateless drone considers to be the best. The algorithm performs the following steps:
  1. Finds which *Stations* are within 1 move away
  2. If there is at least one positive *Station*, returns the *Direction* which would reach the best *bestNearbyStation()*.
  3. Otherwise, returns a random *Direction* which avoids negative *Stations*.

  A full description of the algorithm's strategy can be found in the *StatelessLogic.java* file.
- **bestNearbyStation(List<Station>nearbyStations)**: given a list of nearby Stations, returns the *Station* with the highest *money* value. The SortByMoney Comparator is used for this method.

3.0.9 StatelessDrone

Description: This class represents the stateless drone. It is a sub-class of the Drone, which also contains a StatelessLogic component.

Fields:

- *droneLogic*: the StatelessLogic component used for deciding the next move.

Methods:

- **getNextMove()**: calls the droneLogic.getNextMove() method to return the Direction where the stateless drone wants to move. This method is used by the GameManager when requesting the drone's next move.

3.0.10 StatefulLogic

Description: This class is the stateful implementation of DroneLogic, and thus represents the StatefulDrone's decision-making algorithm. It uses memory to plan a route which will visit all the positive Stations on the Map and avoid all negative *Stations*. All the drone's moves are planned from initialisation. Please read the Stateful Drone Strategy for an understanding of how the algorithm works.

Fields:

- *tempPos*: the Position of the stateful drone as it simulates its moves around the Map. Since the drone can only move when the GameManager allows it, planning its moves requires using this variable to simulate moving.
- *moveNumber*: keeps track of the drone's turns executed in the GameManager
- *rnd*: the Random Number Generator with the seed specified by the *User*
- *plannedMoves*: an *ArrayList* of Directions containing all the pre-planned moves of the stateful drone.

Methods:

- **getNextMove(Position currPos)**: returns the Direction where the stateful drone wants to go. Since all moves are pre-planned, the *currPos* field is not used and the method returns the *plannedMoves* element of index *moveNumber*.
- **planRoute(Position startingPos)**: returns an *ArrayList* containing all positive Stations in the order they will be visited by the Drone. The algorithm used is a greedy nearest-station.
- **getPathToNextStation(Position startingPos, Station goalStation)**: returns an *ArrayList* of Directions representing the drone's best path from its starting Position to the goal Station. The method uses an A* Search Algorithm [2] and a Node class in order to represent nodes in the search graph.
- **planAllMoves(Position startingPos, ArrayList<Station >route)**: returns an *ArrayList* of Directions containing all the 250 moves that the stateful drone has pre-planned. The algorithm uses the *route* obtained from *planRoute()* and calls the *getPathToNextStation()* for each *Station* in the *route*. If the drone finishes early, it will zigzag between its last move.
- **approximateMovesToStation(Position pos, Station station)**: returns approximately how many moves it takes to get from the specified Position to the Station. This method is used as a heuristic in the A* search.
- **getRandomDirection(Position currentPos)**: returns a random Direction for the drone to move to. This method is only used as a precaution for the A* search algorithm getting stuck.
- **getInverseDirection(Direction dir)**: returns the inverse of a Direction (e.g. NE - SW). This method is used by the drone when it finishes early, so it can zigzag until it finishes 250 moves.

3.0.11 StatefulDrone

Description: This class represents the stateful drone. It is a sub-class of the Drone, which also contains a StatefulLogic component.

Fields:

- *droneLogic*: the StatefulLogic component used for deciding the drone's moves

Methods:

- **getNextMove()**: calls the droneLogic.getNextMove() method to return the Direction where the stateful drone wants to move. In the *StatefulLogic* component, all the moves are pre-computed. This method is used by the GameManager when requesting the drone's next move.

3.0.12 SortByMoney

Description: A comparator used for comparating Stations based on their *money* field. This is used by the StatelessLogic class when determining the best Station to move to.

3.0.13 Node

Description: This class is used for representing nodes in the A* search algorithm used in StatefulLogic's *getPathToNextStation()* method. In this case, a Node represent a Position where the drone could move on the map. They are necessary for finding the path from the drone to a Station.

Fields:

- *position*: the Position of the *Node*
- *parentNode*: the parent *Node*
- *directionFromParent*: the Direction in which the *parentNode* had to move to reach the current *Node*

Methods:

- **equals(Object o)**: this method overrides the standard *equals()* method. Two nodes are considered equal if they have the same *position*. Returns true or false.

3.0.14 Station

Description: This class is used for storing information about the stations on the Map and updating them as the Drone charges around.

Fields:

- *power*: the amount of power contained
- *money*: the amount of money contained
- *symbol*: the station's symbol (”danger” or ”lighthouse”)
- *color*: the color of the corresponding feature on the GeoJSON map
- *position*: the Position of the station on the Map
- *id*: the id of the station

Methods:

- **takePower(double amount)**: takes away an *amount* of power after the Drone charges. This method is called by the GameManager after the *Drone* has charged. The *amount* can be negative if the *Station* is negative.
- **takeMoney(double amount)**: takes away an *amount* of money after the Drone charges. This method is called by the GameManager after the *Drone* has charged. The *amount* can be negative if the *Station* is negative.

3.0.15 Position

Description: This class describes the geographical coordinates of the Drone and Stations on the Map, as well as providing geographical functionality.

Fields:

- *latitude*: the latitude position
- *longitude*: the longitude position
- *directionChanges*: a *HashMap* of Direction keys and *ArrayList* $<double>$ values, representing the changes in position from moving in each of the 16 *Directions*. The amounts are pre-computed in a static final variable, since they are the same for all positions.

Methods:

- **nextPosition(Direction direction)**: returns the next position of the Drone after it would move in the specified Direction.
- **inPlayArea()**: returns whether or not this position is in the play area.
- **inRange(Position destination)**: returns whether or not this Position lies within 0.00025 degrees of the destination. This is used for detecting whether a Drone is in range of a Station.
- **getDist(Position destination)**: returns the euclidean distance between this position and the *destination*. This is used for calculating the distance between the Drone and a Station.
- **sinOfAngle(double angDeg)**: returns the sine value of an angle. Used in computing *directionChanges*.
- **cosOfAngle(double angDeg)**: returns the cosine value of an angle. Used in computing *directionChanges*.

3.0.16 Direction

Description: This is an Enumerator describing the 16 different compass directions in which the *Drone* can move.

3.0.17 DistanceComparator

Description: this is a Comparator used for sorting Stations based on their distance from a *Position* of origin (usually the Drone's position).

3.0.18 Debugger

Description: This class contains several useful methods for debugging the project. Please note that this class **should not be added to a public release** and should only be used for debugging purposes.

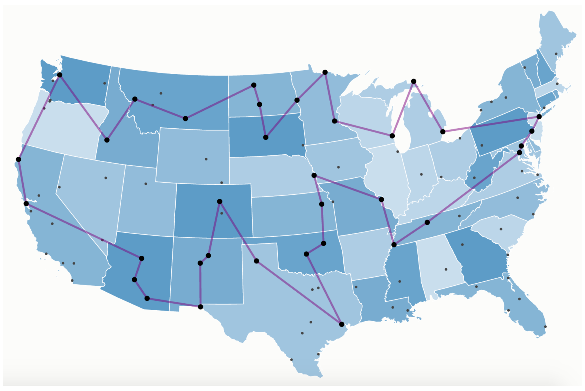
Methods:

- **printMapFeatures()**: prints out all the features extracted from the GeoJSON map.
- **printArgs()**: prints out the input arguments: date, latitude, longitude, seed, droneType
- **printStations()**: prints out the latitude, longitude, money, power values of each station in the Map.
- **printStationsByDistance(Position)**: prints out all Stations' IDs, ordered by their distance from the origin.
- **printChargeMessage(Station)**: prints out whether the drone charged from a Positive, Negative or Empty station. This method is called by the GameManager to track the Drone's performance.
- **printMaxCoins()**: prints out the sum of coins from all positive Stations on the Map.

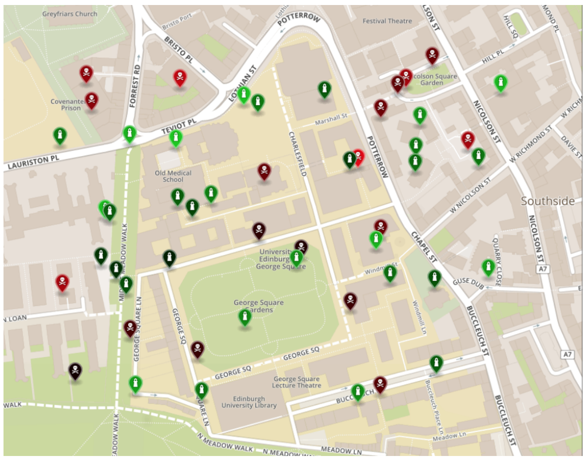
4 Stateful Drone Strategy

4.1 Similarity with Travelling Salesman Problem

The Traveling Salesman Problem (often called TSP) is a classic algorithmic problem in the field of computer science. The Travelling Salesman Problem describes a salesman who must travel between N cities. The order in which he does so is something he does not care about, as long as he visits each once during his trip. There are several variants of this problem, but the task focuses on optimization and finding the shortest (or in some cases cheapest) travel path.[2]



(a) visual example of the TSP



(b) a PowerGrab map

Figure 4: Visual similarity between TSP and PowerGrab

After looking at the PowerGrab map we can notice that our task is very similar to the Travelling Salesman Problem: finding a path that collects as many (preferably all) coins on the map. In other words, finding a path that goes through all positive stations.

After testing the Stateless Drone, I have realised that most stations lie within less than 5 moves of another station. Since each map contains around 30 positive stations, I have anticipated that the drone should be able to visit all positive stations in less than 150 moves by using a greedy nearest-station algorithm.

In conclusion, the algorithm used for the Stateful Drone is an implementation of the Travelling Salesman Problem. It attempts to find a path which goes through all positive stations while avoiding negative stations.

4.2 The Strategy

4.2.1 Description

The Stateful Drone algorithm (implemented in the StatefulLogic class) plans all its 250 moves at initialisation. In summary, the algorithm is divided into two parts:

1. **Construct a Route**: an order in which the positive stations will be visited by the drone. The route is similar to the *TSP figure above*: it represents a graph of stations, which the drone will use to travel from station to station. This part is implemented through a Greedy (nearest-station) algorithm.

2. **Plan a path from Station to Station**: a set of moves that the drone will execute to move from Station to Station in the planned Route. Planning a path is necessary for the drone to move efficiently and avoid negative stations. This part is implemented through an A* search algorithm.

This strategy should result in a large performance improvement over the Stateless Drone. As compared to moving randomly and 'seeing' only what's surrounding it, the Stateful Drone memorises a Route, simulates its movements to construct a Path, and memorises all its 250 moves from initialisation.

4.2.2 Step by Step Implementation

1. **Construct the Route (Greedy nearest-station)**
   (a) Take the drone's starting position
   (b) Find the nearest positive station, add it to the Route
   (c) Find the next nearest positive station, add it to the Route. Repeat until all positive stations have been added
2. **Plan a path to each Station in the Route (A* search)**
   We will need a Node class (containing position, f-value, parent node, direction from the parent) to represent nodes in the search.
   We will also need a List of nodes which are/will be visited *(called visitedList)* - this prevents nodes from being visited twice - and a List of nodes to expand in the search algorithm *(called nodeList).*
   (a) Take the drone's starting position
   (b) Create a Node containing the starting position (we can initialise f to 0). Add it to the nodeList and visitedList.
   (c) Get the node with the least f-value. Remove it from the nodeList.
   (d) Generate the node's successors (16 Nodes corresponding to moving in the 16 Directions)
   (e) Remove successors which go outside the play area, or are contained in the visitedList
   (f) For each remaining successor:
      - If the successor reaches the goal Station, stop the search
      - Calculate the successor's f-value.
        g = 1000 if the Node is within range of a negative station, 1 otherwise. This ensures the drone avoids negative stations.
        h = approximate moves to reach the Station = (Station's position - Node's position) / 0.00025
        f = g + h
   (g) Add the successors to the nodeList
   (h) Repeat from step 2(c) until the destination is reached
   (i) Return the path (*List of Directions*) by backtracking through the Node's *parentNode* field
3. **Add each path to a the list of *plannedMoves***
4. **If all positive stations have been visited in less than 250 moves, add ZigZaggin moves to the list of plannedMoves until it contains 250 moves**
5. **The plannedMoves list now contains all 250 pre-planned moves, which the Drone can execute**

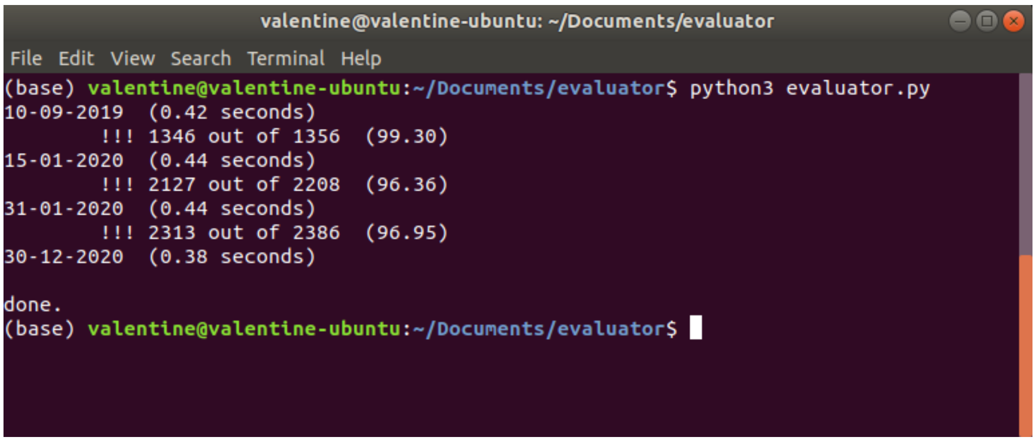
When implementing the code for the algorithm described above, a few modifications have been made:

- A *TreeMap <Integer, ArrayList<Node>>* has been used to represent the *nodeList* containing Nodes and their respective f-values. This representating allows us to find the node with the least f-value in logarithmic time, rather than search through the entire list (which expands at a branching factor of 16).
  The *Integer* key represents to the f-value. The *ArrayList<Node >* contains all the Nodes which have the corresponding f-value.
- To ensure the algorithm doesn't enter an infinite-loop, the A* search will stop after 100 iterations and will return a random legal Direction.

4.3 Results

Testing on the Powergrab Evaluator (Author: Bora M Alper), the Stateful Drone manages to collect 100% of the coins on 727 Maps, and more than 96% of coins on the remaining 3 Maps.

The drone collects maximum coins on 99.5% of the Maps.
The drone usually finishes collecting all coins in less than 150 moves (The rest are ZigZags).
The average runtime of the program is around 0.40 seconds (0.60 seconds on DICE).

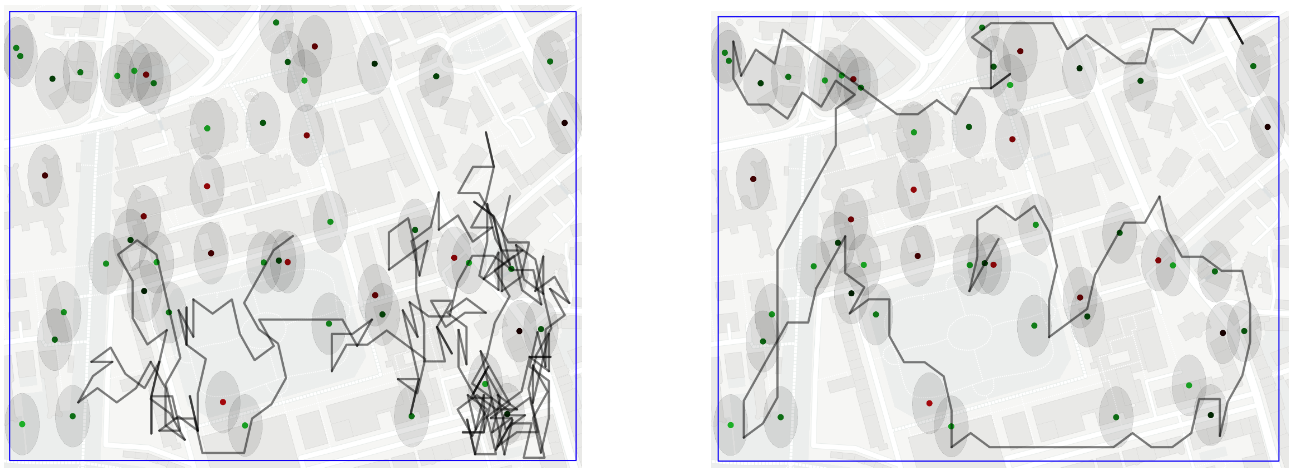

valentine@valentine-ubuntu: ~/Documents/evaluator
File Edit View Search Terminal Help

```
(base) valentine@valentine-ubuntu:~/Documents/evaluator$ python3 evaluator.py
10-09-2019  (0.42 seconds)
        !!! 1346 out of 1356  (99.30)
15-01-2020  (0.44 seconds)
        !!! 2127 out of 2208  (96.36)
31-01-2020  (0.44 seconds)
        !!! 2313 out of 2386  (96.95)
30-12-2020  (0.38 seconds)

done.
(base) valentine@valentine-ubuntu:~/Documents/evaluator$ 
```

Figure 5: Result on the Evaluator: Collects all the coins on 99.5% of the Maps

The figure below shows the comparison between the Stateless and Stateful Drone playing on the same map.



(a) Stateless Drone performance (b) Stateful Drone performance

Figure 6: Comparison between Stateless and Stateful Drone, on 04/04/2019 Map

As we can see the Stateless Drone has a jittery path with quite a lot of back-and-forth motion (because, being memoryless, it cannot remember where it has been). It attempts to avoid negative stations while visiting positive stations, but leaves many stations uncharged and manages to **collect only 900 coins** (39% of the maximum).
In comparison, the Stateful Drone manages to visit all positive stations in only 116 Moves, **collecting all 2274 coins** and avoiding all negative stations.

4.4 Further Improvements

While the Stateful Drone's performance is very good, it is also clear that it could be improved even further. For example, the path it follows on certain maps does not seem optimal (see figure below). This is because the Greedy nearest-station route (described in the algorithm above) does not necessarily make the best decisions.

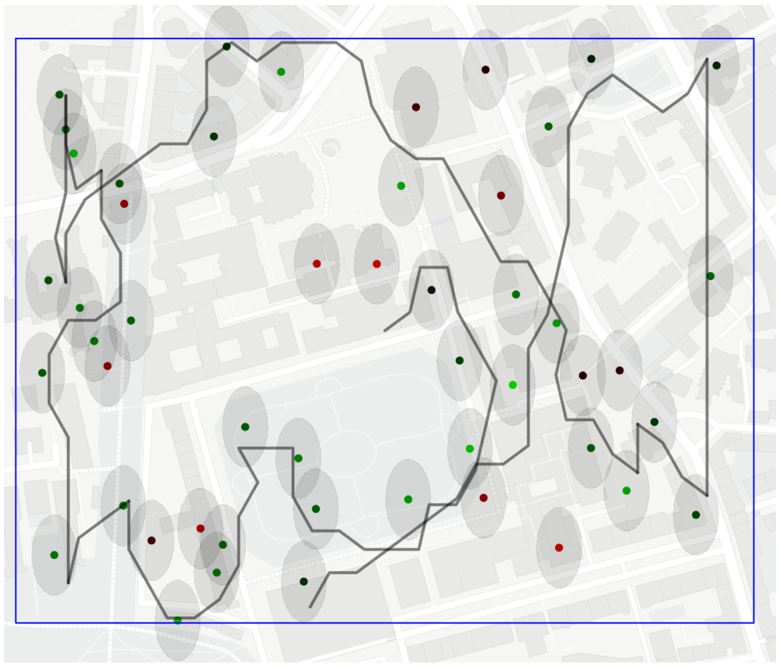


Figure 7: Stateful Drone's path on the 10/10/2019 Map

One way to improve this would be to implement a 2-Opt Swap Optimization Algorithm [3] on the initial greedy route to rearrange it into an even shorter route. To improve even further, we could run the 2 Opt-Swap optimization algorithm several times and pick the shortest path overall, but this would increase runtime. While this solution will not be a global optima, it should be noticeably better than the initial greedy solution.

However, the Drone manages to collect all coins in less than 150 moves in almost all the maps, so for the purpose of this project the current implementation seems to be sufficient.

5 Personal Conclusions

I found this project to be very enjoyable, mainly because I challenged myself to abide by the *Separation of Concerns* design principle. Implementing an efficient Stateful Drone was not as difficult as I expected. Initially, I planned on using the 2-Opt Swap algorithm to optimize the Drone's route, but decided not to after seeing that the Greedy algorithm is good enough.

Overall I am happy that I learned new things and I feel more confident in my Software Engineering capabilities.

6 References

[1] Separation of Concerns desgin principle - Wikipedia

[2] A* Search Algorithm - geeksforgeeks.org

[3] 2-Opt Swap Optimization Algorithm - technical-recipes.com

This project wouldn't have been possible wihtout the following resources:

`https://stackoverflow.com/` - It features questions and answers on a wide range of topics in computer programming. It's a great place to find answers when you're stuck.

`https://www.geeksforgeeks.org` - A computer science portal for geeks. It contains well written, well thought and explained programming articles and algorithms. It's a great place to learn how to use methods, libraries or algorithms.

`https://docs.oracle.com/javase/7/docs/api/j` - Oracle's API specification for the Java^{TM} Platform, Standard Edition

`https://www.youtube.com/watch?v=SC5CX8drAtU&t=10s` - A visually comparison of the Greedy, Local Search, and Simulated Annealing strategies for addressing the Traveling Salesman problem. [Author: n Sanity channel]

`https://piazza.com/` - Used by the Classroom for answering questions related to the coursework. Professor Stephen Gilmore has been incredibely helpful.

Lecture Slides from Professors Stephen Gilmore and Paul Jackson (School of Informatics, The University of Edinburgh)

Bora M. Alper's Powergrab Visualiser and Powergrab Evaluator