Project Report | Informatics Large Practical

Ragnor Comerford[1]

[1]University Of Edinburgh

November 27, 2019

Abstract

The Informatics Large Practical is a project with the aim of developing a Java-based solution to an optimisation problem consisting of an autonomous drone that tries to collect as many coins as possible on a given constrained map. In the following report, the drone strategy shall first be outlined in order to ease the understanding of the software architecture in the second part. In the last part of the report, a comprehensive documentation of the individual classes shall be provided.

Drone Strategy

Stateless drone

The stateless drone is limited in its abilities and hence does not allow much scope for optimisation. However, we can improve the outcome by trying to avoid negative stations and seek positive stations that are just next to the drone. The strategy of the stateless drone works as follows:

It iterates through all valid next positions that are still in the play area and computes their respective utility. The utility of a position is the sum of the coins and power of the nearest station. If there is no nearest station in that position, the utility is simply zero.

Depending on the availability, the drone first tries to pick a position with the highest positive utility. If it does not exist, it tries to pick a zero-utility position uniformly at random. And if still does not exist, it must pick the position with the least negative utility.

Due to the randomness involved in the strategy, the flight path of the stateless drone is highly dependent on the initial seed.

Year	Month	Day	Coins Collected	Max Coins	Ratio		
2019	1	1	823.36	1880.44	0.44	Minimum Ratio	0.24
2019	2	2	787.66	2136.19	0.37	Median Ratio	0.37
2019	3	3	664.42	2177.08	0.31	Average Ratio	0.39
2019	4	4	840.82	2274.81	0.37		
2019	5	5	1109.14	1877.16	0.59		
2019	6	6	965.36	2137.46	0.45		
2019	7	7	462.79	1903.87	0.24		
2019	8	8	639.36	1815.79	0.35		
2019	9	9	664.05	1800.36	0.37		
2019	10	10	771.01	2301.38	0.34		
2019	11	11	649.07	1974.8	0.33		
2019	12	12	1284.63	2492.09	0.52		

Table 1: Performance of stateless drone on test maps.

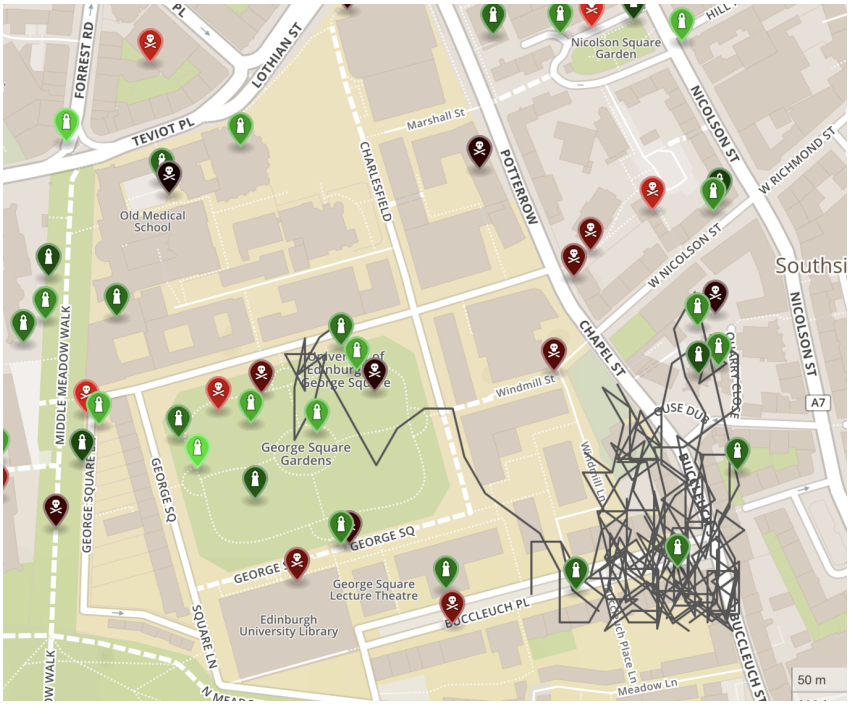

Figure 1: Flight path of the stateless drone (Map: 01.02.2019)

Stateful drone

It should be noted that the constraints of this optimisation problem are to lose to show the algorithm's abilities. Making power more scarce in the game, would show its advantages over simple greedy algorithms.

The strategy of the stateful drone uses stochastic optimisation technique that can be described as a combination of the Monte Carlo Method and Probabilistic Greediness. The stateful drone runs a specified amount of simulations in which it initiates a rollout drone which performs a whole flight path and then reports the score achieved. The stateful drone then picks the simulation with the highest return and performs its entire flight path. The rollout drone itself constructs its path by iteratively selecting available stations using a probabilistic greedy policy. It picks stations with probability proportional to their utility and then computes the path to the selected station while at the same time avoiding negative stations on its path. This approach allows us to tackle the exploration-exploitation tradeoff by striving to strike a balance between sufficiently exploring the variant space and exploiting the optimal path.

There also exists a tradeoff between computation time and optimality. We can observe that as we increase the number of simulations, our score converges to the optimum but also increases the computation time. The stateful drone class allows setting that parameter as required by the user. Using 100 simulations, we can report an average ratio of collected coins on the test maps of 99.96% and a median of 100%.

The utility of a station is a linear combination of its power, coins and distance to the drone whose weights can be adjusted to the type of maps that the algorithm can expect. E.g. In a setting where power is very scarce, we could penalize long distances much more and increase the weight of power in the utility function.

A further improvement can be made by using the Monte Carlo Tree Search (MCTS) algorithm, in which for each move a tree is constructed whose nodes represent stations that the drone could move to. Instead of building a complete tree that would require O(n!) computational steps, the search tree is built, node-

by-node, according to the outcomes of the simulated playouts. The process can be broken down into the following steps:

1. **Selection** Selecting good child nodes, starting from the root node R, that represent the stations leading to a better overall path (high number of coins).
2. **Expansion** If L is a not a terminal node (i.e. the drone still has power and has not exceeded the 250 moves), then create one or more child nodes and select one (C).
3. **Simulation (rollout)** Run a simulated playout from C until a result is achieved.
4. **Backpropagation** Update the current move sequence with the simulation result.

Since the algorithm presented in the first part of this section already leads to a nearly optimal result, the MCTS algorithm was dropped in favor of reduced complexity. However, the code of the implementation can be found in the Appendix.

Year	Month	Day	Coins Collected	Max Coins	Ratio		
2019	1	1	1880.44	1880.44	1.0	Minimum Ratio	1.0
2019	2	2	2136.19	2136.19	1.0	Median Ratio	1.0
2019	3	3	2168.18	2177.08	1.0	Average Ratio	1.0
2019	4	4	2274.81	2274.81	1.0		
2019	5	5	1876.63	1877.16	1.0		
2019	6	6	2137.46	2137.46	1.0		
2019	7	7	1903.87	1903.87	1.0		
2019	8	8	1815.79	1815.79	1.0		
2019	9	9	1799.65	1800.36	1.0		
2019	10	10	2301.38	2301.38	1.0		
2019	11	11	1974.8	1974.8	1.0		
2019	12	12	2492.09	2492.09	1.0		

Table 2: Performance of stateful drone on test maps.

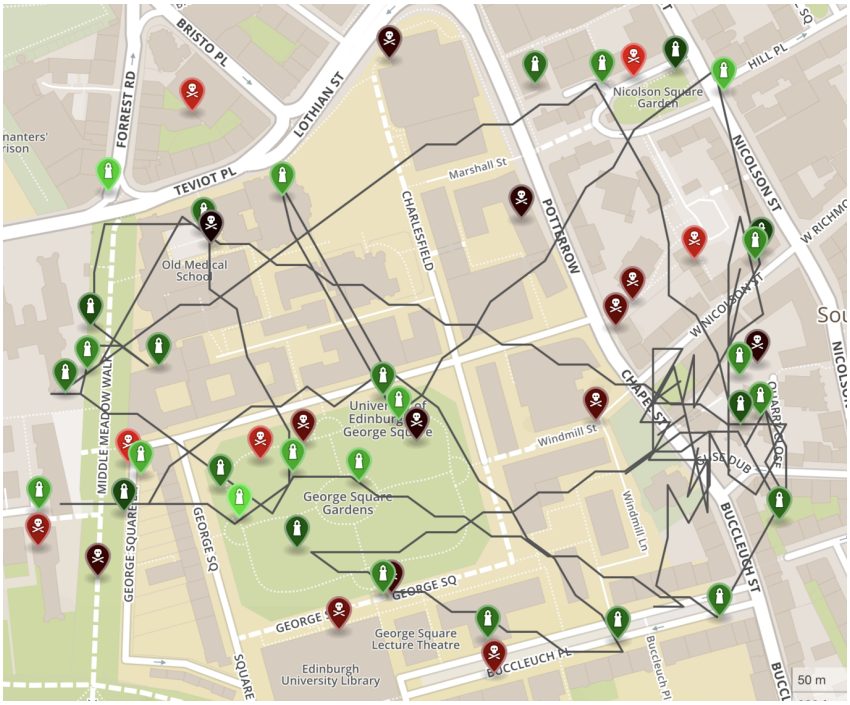

Figure 2: Flight path of the stateful drone (Map: 01.02.2019)

Software Architecture

The class *App* represents the entry point of our application. It implements the desired game by parsing the initial settings of the game from the command line arguments, downloading the required map and instantiating the appropriate drone type with the given initial position and power.

Direction is an enum and specifies the discrete directions the drone is allowed to move in. The *Position* class then represents the position of the drone using geographical coordinates and contains methods that can determine the next position of the drone based on the direction it is moving to or can compute distances between itself and another position.

The *Drone* class is the abstract parent class of *StatefulDrone, StatelessDrone* and *MonteCarloDrone* which cannot be instantiated on its own since the drone either has to be stateful, stateless or the simulation drone. However, the class *Drone* defines the methods that need to be implemented by the child classes (e.g. the method *nextMoves* to compute the next moves of the drone) and directly implements the methods that are common to all drone types (E.g. performing a move and updating the drone's coins, writing the flight path to a text file, computing the utility of a station).

The child classes *StatelessDrone* and *StatefulDrone* only contain the method *nextMoves* in which the strategy of the respective drone type is implemented. In this method, StatelessDrone also iteratively instantiates an object of type *MonteCarloDrone* in order to perform the simulations as described in the previous section. *MonteCarloDrone*'s *nextMoves* method returns the paths it selected using the rollout policy also described above in which it picks a station with probability proportional it its utility. This is made possible by a custom data structure implemented in the class *WeightedRandomCollection.*

Finally, the software architecture also contains a class *Move* which holds certain immutable properties of a move like coin and power gain, utility, direction and the potential station that the drone would connect to on performing that move. The class makes it easier and cleaner to perform operations on the move (writing the move to the text file, updating the drone's state) without having to repeatedly recompute these properties or having to pass them to every single method as arguments.

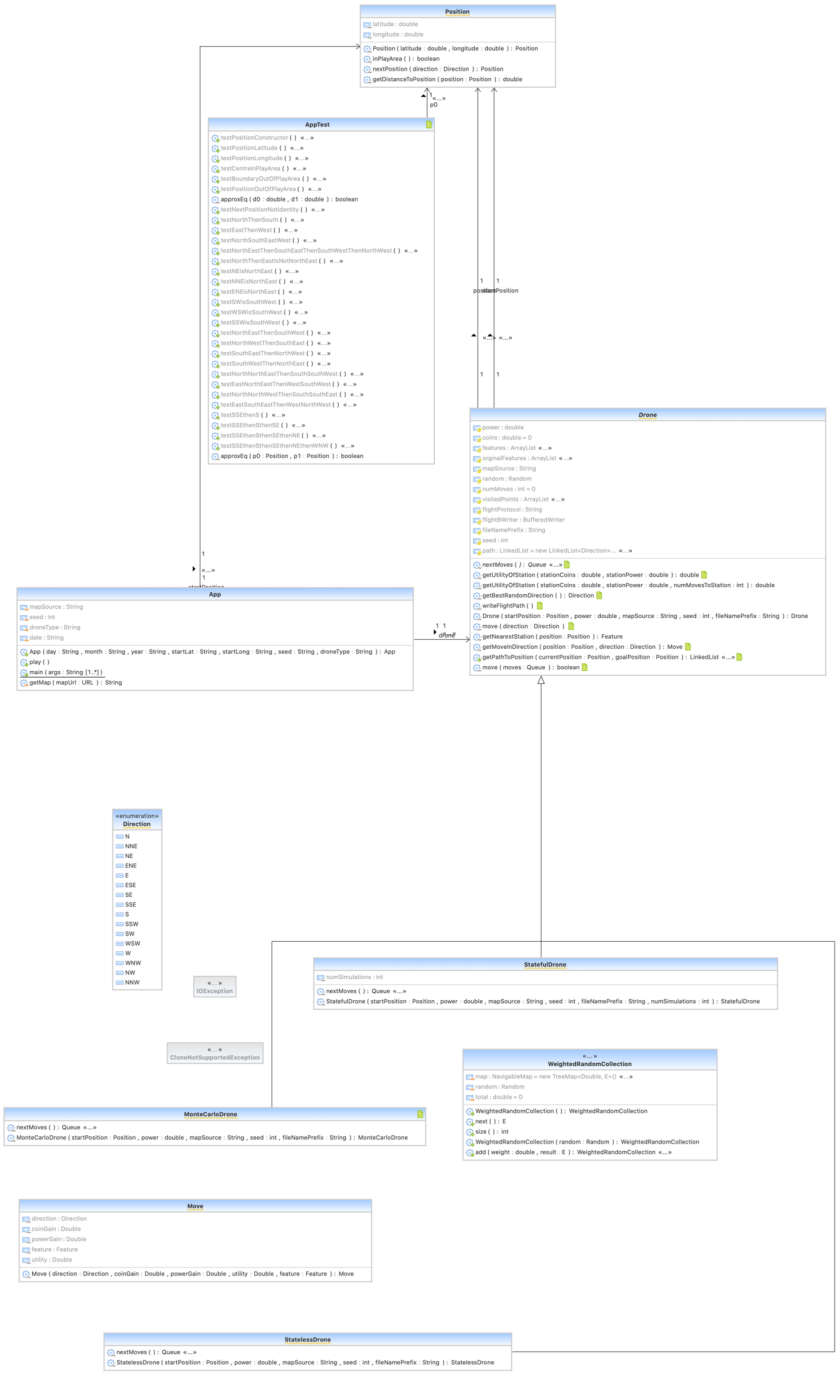

Figure 3: UML diagram of Software Architecture

Class Documentation

App

The class *App* implements the desired game by parsing the initial settings of the game from the command line arguments in *main* and instantiating a game of type *App* using the parsed parameters. The constructor of *App* then calls the function *getMap* to download the map for the given date parameters and instantiates the appropriate drone type. Finally, the *play* method is called which repeatedly gets the next moves determined by the drone and performs them using the *move* method of Drone.

Drone, StatefulDrone, StatelessDrone, MonteCarloDrone

Drone is an abstract class that implements the attributes and methods that are shared by the different drone types. The class uses a variety of attributes to keep track of the drone's state in the game. These include its position, number of coins and power, visited points and properties of the map it runs on.

The class method *getMoveInDirection* returns a move object containing the properties of the move such as coin and power gain or utility. The coin and power gain is determined by the station the drone will be closest to after moving from the given position in the given direction.

The function *getPathToPosition* is used by some of the drone types and computes the best path from the current position of the drone to a given station (goal position). It iterates through the directions, picks the direction that brings the drone closest to the goal and that doesn't lead to a loss of coins and then repeats until it reaches the goal. In case the function can't find a path that avoids negative stations (e.g it revisits a certain point which would lead to an infinite loop), it just returns null. The drone then ignores that station for now.

The method *nextMoves* implements the different type of drone's strategy of determining the next move. It is defined as abstract since the strategy is not general but depends on the drone type and hence needs to be implemented by the subclasses.

The stateful implementation of the *nextMoves* method works by repeatedly instantiating a drone of type *MonteCarloDrone* to simulate a whole run for a specified number of times. The methods then selects the simulation drone with the highest final amount of coins and return its entire path. The number of simulations can be adjusted to the required computation time by passing it to the constructor of the stateful drone.

In the case of *MonteCarloDrone,* the nextMoves method iterates through all the stations on the map that are available and adds paths to positive-utility stations alongside their respective utilities as weights to the custom probabilistic data structure *WeightedRandomCollection.* It then randomly picks a path from the random collection and returns it. If the random collection is empty, usually when all positive stations have been visited, it just picks a random move in the same way the stateless drone would pick a random move. This is done in *Drone*'s method *getBestRandomDirection.* It computes the best direction to take by maximising the utility of the move. It first picks the move with highest non-zero utility among the moves that get in reach of a station. In the case that the highest utility is negative (due to a negative station) it tries to randomly pick a move that does not encounter any station (zero utility). In case no such move exists, it must remain with the negative-utility move. *getBestRandomDirection* is the main function called in the *nextMove* implementation of the stateless drone to compute its path.

The polymorphic method *move* is overloaded and upon receiving a single direction as an input it performs the actions that are required by the rules of the game when moving in a certain direction. It is implemented in the superclass *Drone* because the procedure is the same for all drone types. Given a direction that the drone has decided to move to, this method updates the position of the drone and decreases its power by 1.25 as specified in the instructions and performs the exchange of coins and power in case a station is in reach. Additionally, this method writes the current move to a text file as required for the submission.

However, upon receiving a queue of moves that the drone has planned to move to, this method iteratively pops a direction from the queue and passes it to the other move function which performs the required actions. In the case of the stateless drone, this function is always passed a singleton queue since the stateless is only allowed limited look-ahead. A queue is used instead of an array as it reduces the runtime of getting the next move from O(n) to O(1).

This method writeFlightPath is responsible for writing the flight path after the drone has completed its trajectory. The method takes the preserved original features of the map, adds the visited positions (instance variable *visitedPoints*) as a feature and then writes it to a geojson file.

WeightedRandomCollection

The class *WeightedRandomCollection* implements a data structure that facilitates randomly selecting an element with a probability proportional to its assigned weight. It uses a Treemap to store the cumulative sums of the stations' utilities and picks an element by generating a number uniformly at random between 0 and the total sum and returns the element with the least cumulative weight greater than that number as to emulate a bucket in which the elements take up space according to their weight. The probability is then proportional to the space.

Appendix

```
 /*****************************************************************************
 *  File:   StatefulDrone.java
 *  Author:  Ragnor Comerford (s1614102)
 *****************************************************************************/
package uk.ac.ed.inf.powergrab;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Random;

import com.mapbox.geojson.Feature;

class StatefulDrone extends Drone {

    StatefulDrone(Position startPosition, double power, String mapSource, int seed,
    String fileNamePrefix)
            throws IOException {
        super(startPosition, power, mapSource, seed, fileNamePrefix);
    }

    Move nextMove() throws IOException, CloneNotSupportedException {

        if (this.numMoves < 250 && this.power > 0) {

            if (this.pathToFollow.size() > 0) {

                return this.getMoveInDirection(this.position, this.pathToFollow.remove(0));

            }
```

```
        else {

            MonteCarloNode root = new MonteCarloNode(null, this.position, this.mapSource,
            this.features, this.path);

            int i = 0;
            while (i < 5000) {
                MonteCarloNode leaf = selection(root);
                double simulation_result = rollout(leaf);
                backpropagation(leaf, simulation_result);
                i++;

            }

            if (root.children.size() == 0) {

                return null;
            }

            MonteCarloNode best = this.bestChild(root);

            this.pathToFollow.addAll(this.position
            .getPathToPosition(best.goalPosition));
            return this.getMoveInDirection(this.position, this.pathToFollow.remove(0));
        }

    }

    else {
        return null;
    }

}

MonteCarloNode selection(MonteCarloNode node) throws IOException, CloneNotSupportedException {

    while (node.isFullyExpanded()) {

        MonteCarloNode maxUCBChild = node.children.get(0);

        for (MonteCarloNode child : node.children) {

            if (child.getUCB1(300) > maxUCBChild.getUCB1(300)) {

                maxUCBChild = child;

            }

        }

        node = maxUCBChild;
    }

    MonteCarloNode child = node.getNextChild();

    return child == null ? node : child;
```

```
    }

    double rollout(MonteCarloNode node) throws IOException, CloneNotSupportedException {
        RolloutDrone rolloutDrone = new RolloutDrone(this.startPosition, 250, this.mapSource,
                1000 + this.random.nextInt(9000), node.pathFromRoot, "sim" + node.goalPosition.latit
        Move nextMove = rolloutDrone.nextMove();
        int i = 1;
        while (nextMove != null) {
            rolloutDrone.move(nextMove);
            nextMove = rolloutDrone.nextMove();
            i++;

        }

        rolloutDrone.writeFlightPath();

        rolloutDrone.flightBWriter.close();
        return rolloutDrone.coins;

    }

    void backpropagation(MonteCarloNode node, double result) {
        if (node != null) {
            node.num_plays = node.num_plays + 1;
            node.total_coins = node.total_coins + result;
            backpropagation(node.parent, result);

        }

    }

    MonteCarloNode bestChild(MonteCarloNode node) {

        MonteCarloNode maxChild = node.children.get(0);

        for (MonteCarloNode child : node.children) {
            if (child.num_plays > maxChild.num_plays) {

                maxChild = child;

            }

        }
        return maxChild;

    }

}
 /*****************************************************************************
 *  File:   MonteCarloNode.java
 *  Author:  Ragnor Comerford (s1614102)
 *****************************************************************************/

package uk.ac.ed.inf.powergrab;
```

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Stack;

import com.mapbox.geojson.Feature;
import com.mapbox.geojson.Point;

class MonteCarloNode {

    int num_plays = 0;
    double total_coins = 0;

    MonteCarloNode parent;
    ArrayList<MonteCarloNode> children = new ArrayList<MonteCarloNode>();
    ArrayList<Direction> pathFromRoot = new ArrayList<Direction>();
    ArrayList<Direction> pathFromParent = new ArrayList<Direction>();
    ArrayList<Feature> originalFeatures = new ArrayList<Feature>();
    ArrayList<Feature> features = new ArrayList<Feature>();
    int depth = 0;
    Position goalPosition;
    Direction direction;
    Drone simulationDrone;
    String mapSource;

    MonteCarloNode(MonteCarloNode parent, Position goalPosition, String mapSource, ArrayList<Feature
            ArrayList<Direction> pathFromGameRoot) throws IOException, CloneNotSupportedException {
        this.goalPosition = goalPosition;
        this.parent = parent;

        this.mapSource = mapSource;
        this.depth = (parent == null) ? 0 : parent.depth + 1;

        for (Feature feature : features) {
            if (feature.getProperty("coins").getAsDouble() > 0 || feature.getProperty("power").getAs
                this.features.add(feature);
                this.originalFeatures.add(feature);

            }

        }

        if (parent == null) {
            // Root node
            this.pathFromRoot = ((ArrayList<Direction>) pathFromGameRoot.clone());

        } else {

            this.pathFromRoot = ((ArrayList<Direction>) parent.pathFromRoot.clone());

        }

        if (parent != null) {
            this.pathFromParent = parent.goalPosition.getPathToPosition(goalPosition);
            this.pathFromRoot.addAll(this.pathFromParent);
```

```
        }

    }

    MonteCarloNode getNextChild() throws IOException, CloneNotSupportedException {

        if (this.features.size() == 0) {
            return null;
        }

        Feature childFeature = this.features.remove(0);
        Position goalPosition = new Position(((Point) childFeature.geometry()).latitude(),
                ((Point) childFeature.geometry()).longitude());

        ArrayList<Feature> childFeatures = ((ArrayList<Feature>) this.originalFeatures.clone());
        childFeatures.remove(0);

        MonteCarloNode child = new MonteCarloNode(this, goalPosition, this.mapSource, childFeatures,
        children.add(child);
        return (child.pathFromRoot.size() <= 250) ? child : null;

    }

    double getUCB1(double biasParam) {
        biasParam = 1000;
        return (this.total_coins / this.num_plays)
                + Math.sqrt(biasParam * Math.log(this.parent.total_coins) / this.total_coins);
    }

    boolean isFullyExpanded() {

        return this.features.size() == 0 && this.children.size() > 0;
    }
}
```