# Informatics Large Practical

### Implementation Report

Kuenzang Losel – s1732459

Contents

[Software architecture description 2](#_Toc26282788)

[Class Documentation 3](#_Toc26282789)

[**Stations Class** 3](#_Toc26282790)

[**Direction Class** 4](#_Toc26282791)

[**Position Class** 4](#_Toc26282792)

[**App class** 5](#_Toc26282793)

[**Drone Class:** 5](#_Toc26282794)

[**Stateless Class:** 7](#_Toc26282795)

[**Stateful Class :** 8](#_Toc26282796)

[Stateful Strategy 10](#_Toc26282797)

# Software architecture description

Stations

Before I even started coding, my initial thoughts on the architecture of the application was to have a class such that the object of that class would contain all the details of a station, such as the Coordinates, coins and power. However, as this would require us to create 50 objects of that class and it'd mean that I would need to call that object anytime I want to access any of its variables.

As this would have made looping through a variable of that class for all stations a bit more complicated, I decided not to do this and have a class which initialises 3 array type variables that would store the variables that each station can have, so Position, an object created by using the coordinates of the station, coins and power. Then the data type Double was used to represent the coins and power variables as Double returned a higher precision then a Float type would.

This overall decision made iterating through the coins or the power of the station or calling a variable of the station very easy to understand and use, as an index in one array type of the variable would represent the variables of the same station in other arrays as well (the index 10 in all Three arrays will return information from the same station). In addition to that, the class where the Map is downloaded is in, “Stations” and this is where the three arrays will be initialised as well. Therefore, implementing all the decisions regarding the stations in one single class.

Drone

As both the stateful and stateless type drones share the same basic characteristics, movement styles (Charging to closest station after moving) and only differ in their ability to use the geoJSON map to map their movements. I decided to create a Drone class to be the Parent class of the stateful and stateless class. This would mean that the Stateful and Stateless class are the **subclasses** of the Drone class, therefore will inherit all the properties of the Drone class. Doing this meant, I only needed to create the methods shared by both the classes, Stateful and Stateless, once in the Drone class, and they would be able to inherit those methods.

The drone class will also have methods and variables that are private, so the two subclasses will have to use the getters method to access it. This is mainly done in order to avoid duplication and the two subclasses should not be able to change these variables and methods at all, e.g. in order to avoid false incrementation of counters.

In addition, for all of the classes, a LinkedHashMap was used so the application would maintain the insertion order. This was done in order to prevent different flight paths in of the same map in different devices, as a normal HashMap will not the maintain the order of inputs, sometimes resulting in different flight paths.

# Class Documentation

\*\*Note for the Class Documentation section\*\*

• LHMap = LinkedHashMap •AL = ArrayList

• Instance Variables

• Instance Methods

## **Stations Class**

#### Downloads the map from a server then initialises array variables that contains properties of all the features in the map.

#### Instance Methods

|  |  |
| --- | --- |
| Name and Types | Description |
| Stations(String link) | Initialises the instance variable link with the String variable passed to the constructor |
| Connect() | Opens a communication link to the URL of type string, then by using a bufferedReader, reads the contents of the webpage then saves the JSON as a type String. Then extracts and returns a List of type Feature from the formatted JSON string. |
| Init(AL<Feature> features ) | Loops through a List of Feature then initialises 3 instance variables with the Properties (coins and power) and the Coordinates of each Feature, which has been used to create a Position object. An index in all the 3 variables will refer to the variables/properties of the same feature. |

#### Instance Variables

|  |  |
| --- | --- |
| Name and Types | Description |
| Private Position [ ] allPos | Stores a list of objects of type Position, which represents the coordinates of each feature |
| Private AL<Double> allPower | Stores a list of type Double, for higher precision, which represents the property *power* of each feature. |
| Private AL<Double> allCoins | Stores a list of type Double, for higher precision, which represents the property *coin* of each feature. |

## **Direction Class**

An Enum class that defines a collection of constants of type Direction and contains an instance variable which returns an opposite direction of a given direction.

|  |  |
| --- | --- |
| Public final static LHMAP<Direction, Direction> oppositeDirections | A LinkedHashMap with the keys and values both having the type Direction. Contains 16 keys each with a value that refers to the direction opposite to it. Variable is final, so is not changeable. |

## **Position Class**

Where coordinates of a feature is used to create a new object of class Position, which contains methods to move the drone from one position to another.

|  |  |
| --- | --- |
| Position(double longitude, double latitude) | Initialises the two instance variables of the class, longitude and latitude |
| nextPosition(Direction d) | moves a position in the *d* direction then returns the new Position |
| inPlayArea() | Checks if the drone is within the play area. |
| calcMove(double a | Calculates moves in two directions, x and y plane, with a radius of r. Then returns it by adding the two moves in an array of type double |

|  |  |
| --- | --- |
| double Latitude | stores the latitude, y-axis, of the position |
| double longitude | stores the longitude, x- axis, of the position |
| double xMin, xMax | stores the minimum and maximum boundaries of the map in the x direction. the variables are final, so cannot be changed |
| double yMin, yMax | stores the minimum and maximum boundaries of the map in the y direction. the variables are final, so cannot be changed |

## **App class**

Only function of this class is to run the application through the main method

|  |  |
| --- | --- |
| Main | When called, it creates objects of different classes and executes the movements of the drones then outputs two files, a txt file containing details of every move and a valid formatted Geojson file. |

## **Drone Class:**

Holds basic characteristics, initialiser and methods of all drone types.

|  |  |
| --- | --- |
| Drone(Position a, AL<Double> b, AL<Double> c, Position[] d, int e) | initialises the instance variables of the drone, the random number generator and adds the initial Position of the drone to the instance variable History. |
| indexOfStations() | initialises the two instance variable of type AL<Integer> with the index of the positive and negative stations in the map. |
| totalCoins() | Returns the total number of positive coins that can be collected from the map |
| calcDistance(Position a, Position b) | Returns the distance between two positions which is calculated using the Euclidean distance formula. |
| gameOver() | If the drone cannot make any more moves, it returns true else false; |
| move(Direction d) | Moves the drone in a certain Direction, and transfers coins from the nearest station if in range. It also keeps track of the movement of the drone and its direction. |
| closestStation() | Returns the index of a closest station in range, else it returns the maximum value an Integer can hold |
| transfer() | If a station exists in range, it connects to it and transfers all the coins and power from the station to the drone. Then gives the station a new value of coins and power, the difference from transferring |
| moveLog(Position a, Position b): | Logs the movements of the drones and its coins, power and the direction it moved in by storing it in a String variable. Then appends the variable to the instance variable output. |
| randomNumber() | Uses the Random seed and returns the next random number with the bound of 16. |

#### 

|  |  |
| --- | --- |
| Position current | Holds the current position of the drone |
| AL<Direction> allDirections | Holds the 16 direction that the drone can move in, and this variable is final, so cannot be changed. |
| Random rand | the random number generator that'll produces random numbers that some drones uses to help them choose a direction of movement. |
| double power | Holds the current power of the drone |
| double coins | Holds the current coins of the drone |
| AL<Position> history | Stores all the positions that the drone has visited. |
| AL<Double> allCoins | Each index in the array corresponds to a power station and how much coins it contains. |
| AL<Double> allPower | Each index in the array corresponds to a power station and how much power it contains. |
| Position [] allPos | Each index in the array corresponds to a power station and the object Position which holds its longitude and latitude values |
| int moves | An Integer value of the number of moves the drone has made. |
| int NEG | Stores the number of negative stations the drone has visited |
| int POS | Stores the number of positive stations the drone has visited |
| int counter | Stores a number that keeps track of the stations for a Stateful drone |
| Direction prevD: | Stores the previous Direction that the drone moved in, only used by some Drones |
| Direction chosenDirection | Stores the direction that the drone has decided to move in |
| int seed: | Stores the seed for the random number generator |
| AL<Integer> indexPositives | Stores the indexes of all the positive power stations in the map |
| AL<Integer>indexNegatives | Stores the indexes of all the negative power stations in the map |
| int dummyCounter | A counter that counts the number of moves since the drone last connected to a power station. |

## **Stateless Class:**

Contains the methods to move the drone around the map in a random directions whilst avoiding negative stations.

|  |  |
| --- | --- |
| Stateless2(Position a, AL<Double> b, AL<Double> c, Position[] d, int e) | calls the constructor, methods and the properties of the parent Drone class and initializes a drone of type Stateless. |
| inBigRange(AL<Integer> index): | Scans a big circle of radius 0.00025+0.0003 then returns a LHMap containing the Position and the index of the stations within the circle. |
| inSmallRnage(LHMAP<Position, Integer> a) | Loops through a Map of Position and indexes of stations and checks if the station is within 0.00025 from all the directions. It then returns a Map of the direction the station is in and the stations Index. |
| positiveMovesOnly() | Loops through possible moves the drone can make and returns the directions with the indexes of the stations, which is the closest to the move and has more than 0coins, so hasn't been visited before |
| bestMove(LHMAP<Direction, Integer> possibleMoves): | Returns the direction of the station, which is in range and has the highest coins, compared to the other stations in range.  Returns the direction of the station that has the highest coins and will be charged when drone is moved. |
| bestNegMove(LHMAP<Direction, Integer> possibleMoves) | Returns the direction which contains a negative station which will cause the drone to lose the least amount of coins. |
| listWithoutNegatives(AL<DIrection> directions, LHMAP<Direction, Integer> allNeg ) | Returns a List of Directions that are in range, not negative and is in the play area of the map. |
| closeNegative(Direction d, int pos, int neg) | Compares the two station indexes, usually one positive and one negative. It returns true if the negative station is closer and false if the positive station is closer if the drone moved in a certain direction |
| moving() | finds the best possible move for the drone from its current position. Moves to a direction where the positive is the closest station or moves to a direction which will cause the drone to lose the least coins, if surrounded by all negative stations or moves to a random direction which does not contain any negative and is valid. |
| execute() | Executes the movement of the drones until the game is over and whilst moving, logs each move made by the drone and saves it to the parent instance variable, output. |

## **Stateful Class :**

Contains the methods to execute the movements of the drone in order to maximise the number of coins collected from the map.

|  |  |
| --- | --- |
| Stateful2(Position a, AL<Double> b, AL<Double> c, Position[] d, int e) | calls the constructor, methods and the properties of the parent Drone class and initializes a drone of type Stateful. |
| nextCounter(Position a) | Checks if another positive station is in range of the drone **and** closer than the station the counter variable is referring to. Returns true if it is, else false. |
| shortestMove() | Returns the direction that moves the drone closest to the station the counter variable refers to. It also, considers cases where the drone repeats a move multiple times or takes too long to connect to the next station and makes a move regardless of negatives. |
| inRangeVisited(Position a) | Returns true if the drone is in range of a station that has already been visited before else, false. |
| inRangeNegative(Position a) | Checks if any negative stations in range then compares the distance with the distance from the drone to the station counter variable refers to. If positive is closer and in range then returns false, else true. If the positive is not in range, then returns true if it found a negative station within range of the drone. |
| closestNegative(Position temp) | Checks for the closest station from a position, and returns true if negative and false for positive |
| fillUpGraph() | Initialises the 2D double graph instance variable then fills up the array with the distances between positive power stations. |
| findRoute() | Returns the index of the station closest to another station or a position, then appends the index to a list of already visited List and updates the variable startIndex to the index of the closest station. |
| allRoute(): | Fills up the instance Variable route with a list of station indexes, which connects the initial position to its closest positive station then closest station to that station. Then repeats until the size of route variable is the same as the number of positive stations in the map. |
| possibleMoves(): | Returns a list of directions that contains valid moves and moves that are not in range of a negative station. |
| moving(): | Moves the drone by one step in a direction that brings it one step closer to the station counter is referring to and if it has visited all the positive stations, it moves in a random direction whilst avoiding negatives |
| execute(): | Executes the movement of the drones until the game is over and whilst moving, logs each move made by the drone and saves it to the parent instance variable, output. |

|  |  |
| --- | --- |
| int startIndex | Stores the index of a power station. |
| AL<integer> visited | stores the indexes of all visited stations |
| double [ ] [ ] graph | Once initialised, stores the matrix of n by n, where n is the size of positive stations in the map. graph[i] [j] will store the distance between the station with the i'th and j'th index. |
| AL<Integer> route | Once filled, stores the indexes of positive station that the drone will try to move in order |
| int counter | an integer that guides the drone to a specific power station in the map. |

# Stateful Strategy

Initially, I choose the greedy method as my Stateful strategy, it calculated all the possible route that would visit all the positive power stations from an initial position. Then I would choose the route that required the least cost, which is power in this case. However, this strategy was very expensive to compute with a time complexity of O(n!), in our case where the value of n could is most likely to be greater than 30, that would be 2e32 computations. As this greedy method was also a solution to an already existing problem, the Travelling salesman problem, I found another method that would reduce the time complexity of the strategy to O(n2 \* 2n) and this was achieved through dynamic programming. However, this was still very expensive and inefficient.

Then I realised that, I just need to make it so the drone collects as much coins as it can without costing too much. Therefore, the strategy I choose was the closest neighbour strategy. As the stateful drone is able to scan the entire map before moving, I created a method called fillUpGraph() which would initialise a 2D array, called graph with the size of n by n where n is the size of the positive power stations in the map. Then fill up the array with distances from one positive station to the other by using the calcDistance(Position a, Position b) method, which uses the Euclidean formula to calculate the distance. For a graph[i] [j] the i and j would represent the indexes of the two positive stations in the Array list of power stations.

Then through the method findRoute(), for the initial position, it will return the index of the positive Station which is the closest to it, then updates the variable “startIndex” to the index of the closest station and also adds it to an ArrayList variable, visited, which contains the indexes of drones that has already been visited. This will save us from unnecessary computations and prevents us from visiting the same power station multiple times. It then repeats it for the above, by using the Position of the station that startIndex refers to as the initial position.

\*\**startIndex is the closest station found from the previous move. For the very first move, as it does not have a startIndex, it’ll use the current position instead*.\*\* It also refers to the graph variable instead of calculating the distance between two stations multiple times. Then the method allRoute() runs the findRoute() and adds the value it returns to an ArrayList variable, route, **until** the size of route equals to the size of positive power stations in the map.

This method results to a complexity of O(n^2 + (n\*n/2 + n/2)) , for the fillUpGraph and allroute(). Then the cost of the strategy would be O(n^2), as it chooses the highest cost from a summation of costs. Though this strategy does not provide the quickest route to visit all the power stations, it does however, provides a route that will visit all the power station with a very modest runtime.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

The cost to calculate the route is similar to figure 1, for example for a table of length n=4 after each cell is found, the next compute n-1 iterations. Then the one after that (n-1)-1. The total amount of iterations combining up to , which holds for any value of n.

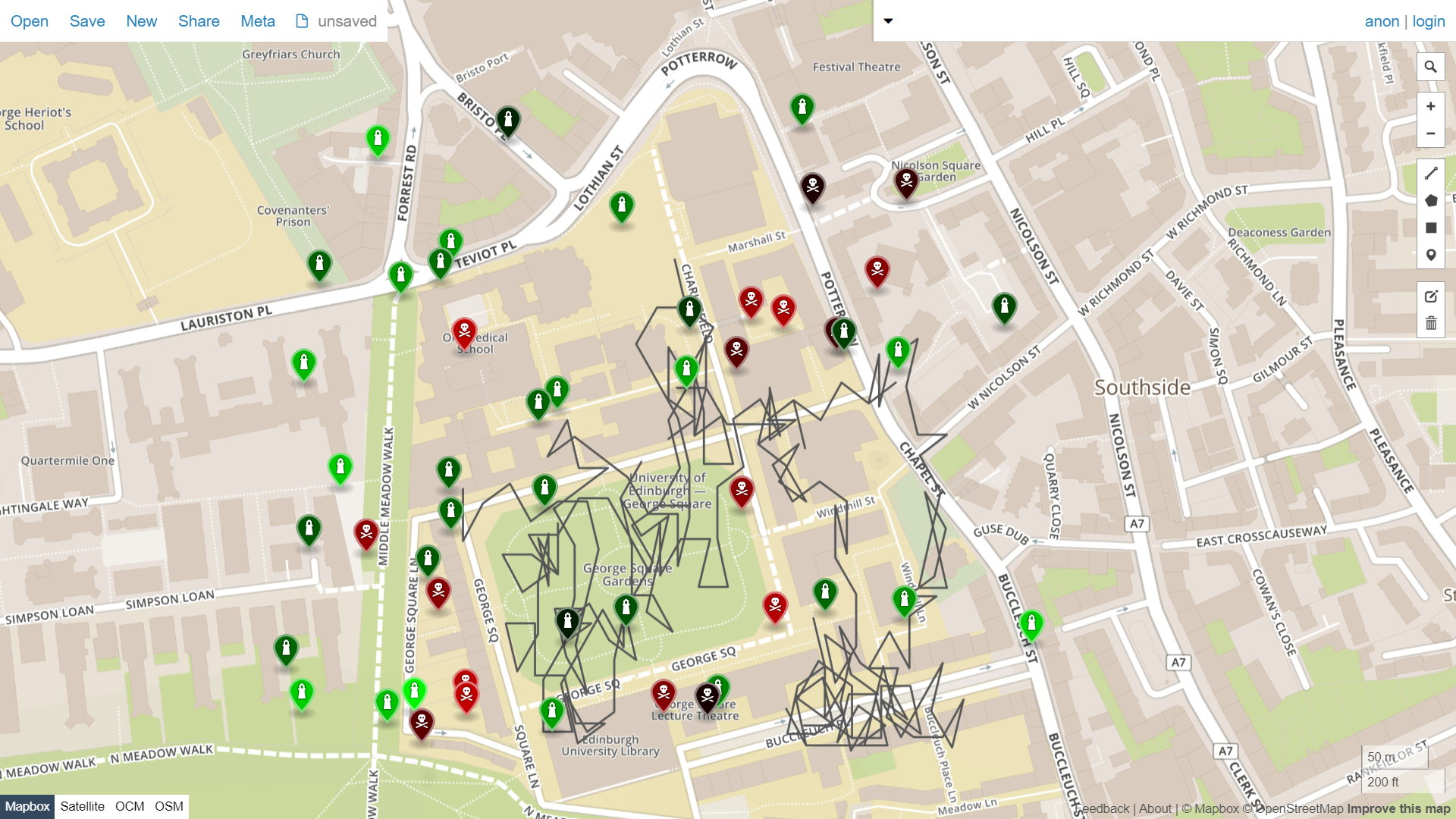
*Figure 1. Route visualisation*

Once the route variable has been filled, the drone will start moving. It’ll move to the first station index in the route array and the index of the station it is moving to will be tracked by an Integer type variable called counter. The drone moves whilst avoiding negatives, out of play moves and making sure that it does not move in a direction which will cause it to charge to a power station with an index that the counter is not referring to, unless it has already charged to that station before. Every time a power station that the counter is referring to is charged, the counter increments by 1, directing the drone to the next closest positive power station.

After instantly charging to a station, the drone is allowed to make backward movements for about 6 steps after that, it tries not to make any more ***repetitive movements***. In the rare events if an already visited station causes the drone to move in a circle and not charge to the desired station, it decides to step back so it is outside the range of the already visited station, then it will then try to charge to the desired station again.

However, in the rare cases where the drones cannot get to a positive station due to a negative station and it has already spent a lot of moves trying to get to the station, it will brute force through the negative stations and charges to it then goes to the positive station. However, if it takes more than 35 moves to reach to the desired station, the drone will decide that it is not worth wasting anymore moves on and moves to the next station in queue.

After visiting all the positive stations, the drone will move in a random direction, whilst avoiding the negative stations and keeping the drone in bounds.



**11 10 2019 55.944425 -3.188396 5678 stateless**

*Figure2. Stateless drone flight rendering*

*Figure 3. Stateful drone flight rendering*



**11 10 2019 55.944425 -3.188396 5678 stateful**