Informatics Large Practical

Coursework 2

Report on Implementation

1. Software architecture description. This section provides a description of the software architecture of your application. Your application is made up of a collection of Java classes; explain why you identified these classes as being the right ones for your application. Identify class hierarchical relationships between classes: which classes are subclasses of others?

* Explain why I choose to store the stations in the Stations class
* Drone class and why, something like stateless and stateful shares the same data but the way they move and how they use the data is different
* Say stateless and stateful are subclasses of drone

Stations Class

I choose the Station class to be the class where the maps would be downloaded from the server. This was done as, all the data fetched from the server will be related to the power stations.

Drone class

The Stateful and the Stateless classes are the subclasses of the Drone class. This was done as, the Stateful and the Stateless class shares the same characteristics and data as the Drone class. The only difference is on how they make use of the data to move around the map.

2. Class documentation. Provide concise documentation for each class in your application. Explain each class as through providing documentation for a developer who will be maintaining your application in the future.

* Explain what each of the classes does, instead of how its used? How its used will go in the next section
* Explain each method in each class, explain the inputs and outputs

Station Class

Init method: Takes in a List of features, then stores different data of all the features in 3 different lists. allPower, allCoins and allPos.

Stateful drone strategy

**\2. Class documentation. Provide concise documentation for each class in your application. Explain each class as through providing documentation for a developer who will be maintaining your application in the future.**

· Explain what each of the classes does, instead of how its used? How its used will go in the next section

· Explain each method in each class, explain the inputs and outputs

Stations

* Connect
  + Opens a communication link to the url of type String, provided in the argument, then . Then by using a BufferedReader,
* init
  + Loops through all the features then stores the properties; coins, power and the coordinates of each feature in a List associated with each of the values. All values of the same feature will share the same index in all 3 Lists.
* txtFile
  + Returns a GeoJson map of type string that contains a featureCollection with one feature of type LineString, that contains all the Positions form the ArrayList position in the argument and all the features from the ArrayList of features in the arguement to it.
* getters for allCoins, allPower, allPos
  + These returns the Lists of values of all the features.

Drone

* indexOfPositives
  + Initialises two ArrayList which contains the indexs of all the power stations with negative and Positive value of coins;
* totalCoins
  + Returns the total value of coins from all the Positive power stations
* calcDistance
  + returns a type double which contains the distance between two coordinates which has been calculated using the Euclidean formula.
* gameOver
  + Returns a Boolean to represent the status of the game. when power or moves is less than 1.25 or 250, respectively, the game is over.
* move
  + Moves the position of the drone to the Direction given then carries out any tranfers possible. It also increments and decrement the values of moves and power for each move. If the given Direction is opposite, so drone taking a complete 180, it increments for repeatCounter.
* closestStation
  + returns the index of the closest Station, positive or negative, if in range, else it returns a very big number.
* transfer
  + Checks if there is a station nearby in range, if not does not do anything else, it checks the value of the coins of the station then carries out a transaction on the coins and power of the station.
* moveLog
  + Logs the data of move after each move then stores it in a string variable.
* getters for history, coin, power, currentMove, current position and output(the thing for txt)
  + returns the variable, todo// say what the variables mean?

Stateless

* inBigRange
  + Scans through a circle of radius (0.0003 +0.00025) and checks if there are any stations within that circle, if yes adds them to a HashMap, with it's position and index then returns the HashMap.
* inSmallRange
  + If the size of the HashMap containing positions of Stations within the big circle is greater than one, it compares each of those position and with the drone moved in all 16 directions then checks if the station is in that direction. Then adds it to a hashMap of Direction and int index and then returns it
* positivesMoveOnly
  + Returns a HashMap of direction and Integer, that only contains directions where the closest station is a positive station.
* bestMove
  + returns the direction to the station that's in range and contains the highest value of coin and coin is greater than 0 and also checks that the Direction is not backwards( complete opposite) as this just results into loops
* bestNegMove
  + returns the direction to the negative station which is in range and has the highest coin
* listWithoutNegatives
  + Returns a list of direction that is not in range of any negative power stations and are inside the play area
* closeNegative
  + checks whether a positive or negative station is closer if we moved in a direction
* moving
  + If theres a direction where the positive station is closer than a negative station, it moves there. else moves in a direction without any negatives in it.
* execute
  + Moves the drone until the game is over and also logs the movement and the current power and coins of the drone.

Stateful

* nextCounter
  + Returns true if another station apart from the one counter variable is referring to is in range. else false
* shortestMove
  + Returns the direction that gives the shortest distance to the next station in route. It checks if the move is valid, and makes decisions if the drone is stuck in a position going back and forward. In case of a null move, it moves in a direction closest to the station in route, regardless of negatives.
* inRangeVisited2
  + Returns true if the drone is in range of an already visited station else false
* inRangeNegative2
  + Returns a type Boolean if the drone the closest station to the position is a negative, else false;
* inRangeNegative
  + Returns a Type Boolean if a negative station is in range of the position
* fillUpGraph
  + Initialises the 2D double array field then fills up the array with distances from one station to the other. if the same station, distance = 0;
* findRoute
  + Returns the index of the closest station from the variable, startIndex or the currentPosition if the drone hasn't visited any stations
* allRoute
  + Fills up the route field with the index of closest station
* possibleMoves
  + returns an Array of directions, that only contains Directions that are in play area and not in range of any negative stations;
* moving
  + Moves in a direction closest to the station route is referring to and moves in a random direction without negatives once all Positive stations has been visited.
* execute
  + Moves the drone until the game is over and also logs the movement and the current power and coins of the drone.

**3**

Stateful drone strategy This section explains the strategy which is used by your stateful drone to improve their score relative to the stateless drone. You should explain what is remembered in the state of the stateful drone and how this is used to improve the drone’s score.

* dont forget to add the triangle picture, and calculations of the complexity

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 possibly 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(N^2 \* 2^N) 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.

Once the route variable has been filled, the drone will start moving. It’ll move to the first station index in the array and the index of the station it is moving to will be kept track 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. Every time a power station that the counter is referring to is charged, the counter increments by 1, pointing the drone to the next power station.

However, in the rare cases where the drone repeats back and forward movements multiples times because it is stuck due to the algorithm chosen, it’ll move regardless of there being a negative station however, it still checks if it is still in boundaries.

And in the rare cases where the drone takes too long to charge to the next station, the drone decides to skips the station as it is not worth it and increments the counter, which directs the drone to the next station in the route variable. Once all the stations has been visited, the drone moves in a random direction whilst avoiding negative power stations and keeps the drone in play area.

//todo for the above, can probably have something along the line of

//Method

//what it do and how it do

//a description/bullet point at the end of what was saved from this class, or what saved variable did it use.

Class

**Stations Class:** Connects to server and downloads the Geojson map and converts it into type String. Then retrieves the features of that geoJson map and stores the different properties and coordinates of the features in three different List. An index in one list will correspond to an index in the others, all relating to the same feature.

Instance Methods

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 through the use of 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 a formatted JSON string.

*init*(ArrayList<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.

*txtFile*(ArrayList<Position> History, Arraylist<Feature> features, String date) : Creates a valid formatted JSON String, that contains a feature of type LineString with all the coordinates from a specific List and also, appends to it all the Features from a list of Feature. Then this formatted string is returned.

*getCoins()*: returns the instance variable allCoins of the class

*getPower()*: returns the instance variable allPower of the class

*getPos()*: returns the instance variable allPos of the class

Instance Variables

*Position [] allPos*: stores a list of type Position, which represents the coordinates of each feature

*Arraylist<Double> allPower*: stores a list of type Double, which represnts the property power of each feature

*ArrayList<Double> allCoins:* stores a list of type Double, which represents the property coin of each feature

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

Instance variable

oppositeDirection: A HashMap 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.

**Position Class**: A class where Objects type Position is initialised with methods to allow you to check if position is within game area or calculate the position after moving in a certain direction.

Instance methods

Position(double y, double x): initialises the two instance variables of the class

nextPosition(Direction d): moves a position in a certain direction then returns the new Position

inPlayArea(): Checks if the drone is within the game area.

calcMove(double a): Calculates moves in two directions, x and y plane , with a radius of r.

Instance variables

latitude: stores the latitude, y-axis, of the position

longitude: stores the longitude, x- axis, of the position

xMin, xMax: stores the minimum and maximum boundaries of the map in the x direction. the variables are final, so cannot be changed

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 the basic characteristics that all types of drones will share.

Instance methods

Drone(Position a, Arraylist<Double> b, ArrayList<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 List<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 can not 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 it's direction.

closestStation(): returns the index of an 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 transfering.

moveLog(Position a, Position b): logs the movements of the drones and it's coins, power and the direction it moved in by storing it in a String variable. Then appends the variable to the instance variable output.

getHistory(): Returns all the positions that the drone has visited whilst moving

getCoin(): returns the Drone's current coins

getPower(): returns the Drone's current power

getMoves(): Returns the amount of moves the drone has made so far

getCurrent(): Returns the current Position of the drone

getOutput(): Returns the log of the drone's movements and it's coins, power and the directions it moved in.

Instance Variables

Position current: Holds the current position of the drone

ArrayList<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 use 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

ArrayList<Position> history: stores all the positions that the drone has visited.

ArrayList<Double> allCoins: each index in the array corresponds to a power station and how much coins it contains.

ArrayList<Double> allPower: each index in the array corresponds to a power station and how much powerit contains.

Position [] allPos: each index in the array corresponds to a power station and the object Position which holds it's 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

int repeatCounter: stores the amount of time the drone repeats a move, backwards then forwards/ vice versa

Direction chosenDirection: stores the direction that the drone has decided to move in

int seed: stores the seed for the random number generator

ArrayList<Integer> indexPositives: stores the indexes of all the positive power stations in the map

ArrayList<Integer> indexNegative: stores the indexes of all the negative power stations in the map

**Stateless Class**: executes the movement of the drone and moves to a positive station if in range, else makes it so the drone moves in a random direction whilst avoiding negative stations

instance Methods

Stateless2(Position a, ArrayList<Double> b, ArrayList<DOuble> c, Position[] d, int e): calls the constructor, methods and the properties of the parent Drone class and initialies a drone of type Stateless.

inBigRange(ArrayList<Integer> index): Scans a big circle of radius 0.00025+0.0003 then returns a Map containing the Position and the index of the stations within the circle.

inSmallRnage(HashMap<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(HashMap<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.

bestNegMove(HashMap<Direction, Integer> possibleMoves): Returns the direction which contains a negative station which will cause the drone to lose the least amount of coins.

listWithoutNegatives(ArrayList<DIrection> directions,HashMap<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 it's 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** : executes the movement of the drone so it tries to visit as much positive power stations as it can whilst avoiding negative stations when possible.

instance Methods

Stateful2(...):calls the constructor, methods and the properties of the parent Drone class and initialies 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, takes into account of cases where the drone repeats moves 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.

inRangeNegative2(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.

inRnageNegative(Position a): checks if any negative stations are in range of the drone, then returns true if it found any else false.

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.

instance Variables

int startIndex: Stores the index of a power station.

ArrayList<integer> visited: stores the indexes of all visited stations

int dummyCounter: a counter that counts the number of moves since the drone last conncted to a power station.

double [] [] graph: once initilased, 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.

ArrayList<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.