University of Edinburgh

INFR09051 PizzaDronz Report

THE UNIVERSITY of EDINBURGH
informatics

Alvaro Martin-Angulo: B

December 9, 2022

Section One: Software Archiecture Description

1.1 Program Flow and Main Method Execution

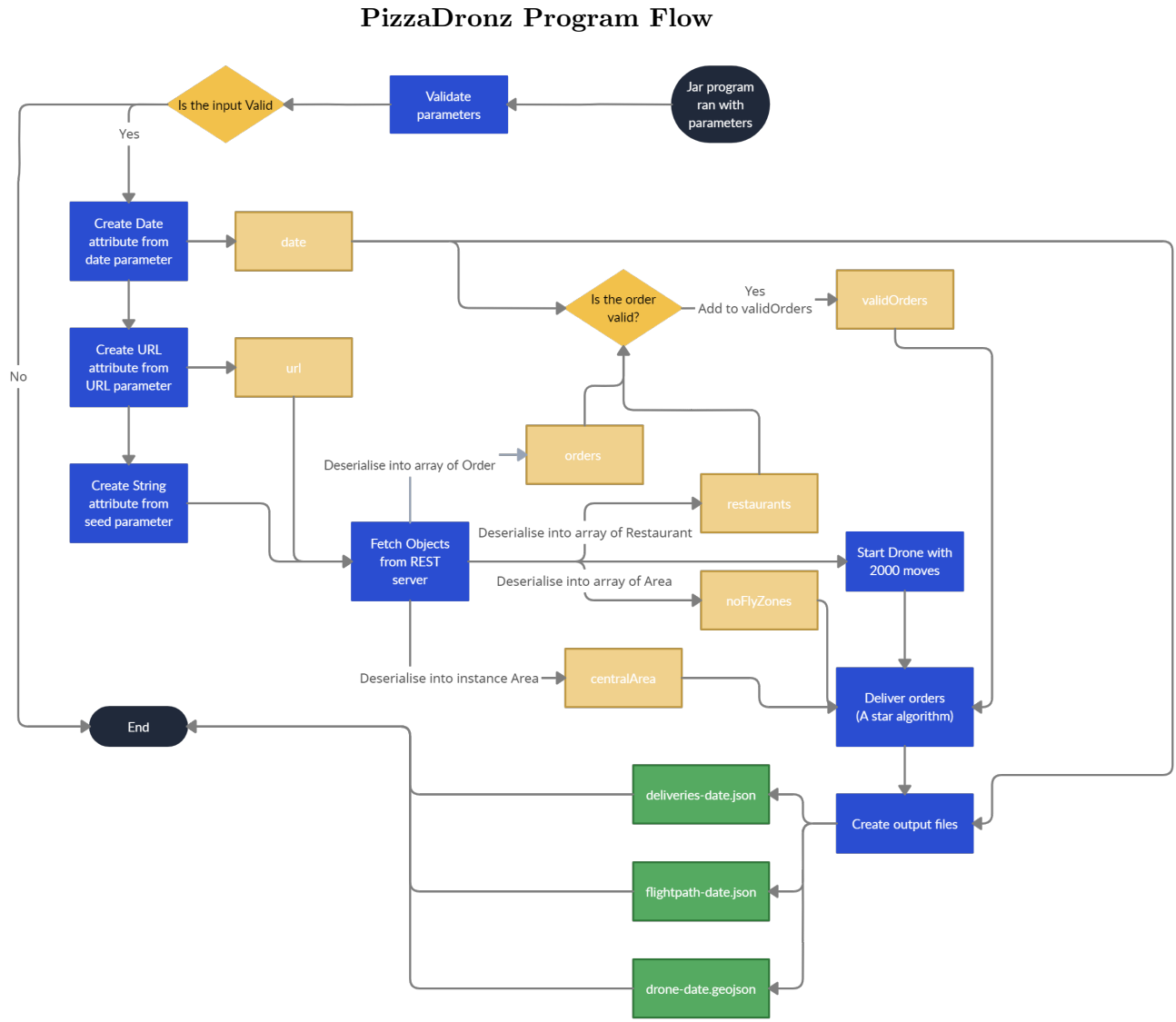

Figure 1: *Simple diagram to show overview of Program flow and execution*

The Entry point of the program is at the main method in the App class. When the Jar file is run with the arguments, the program executes the main method which follows the program flow shown in the diagram above.

Arguments passed in the command line are validated using the InputValidation class, which makes use of the DateValidator class to validate the date argument.

Once input is validated, arguments are assigned to instance objects of their respective class (Date date, URL baseUrl, and String seed).

Making use of the IlpRestClient Class, the program fetches information needed to complete the orders for that date from the REST server. The information is deserialized into instances

of the respective Classes:

- orders deserialized into an array of Order.
- restaurants deserialized into an array of Restaurant.
- noFlyZones deserialized into an array of Area.

At this point, the program has all of the information needed in order to create the files for the date provided. To avoid unnecessary work, the program now makes a sorted arrayList of all valid orders and initializes a drone from the Drone class to deliver the orders.

The drone runs the path-finding algorithm for the valid orders, setting their state to delivered until the 2000 moves are consumed. This produces an ArrayList of the RouteNode class, which contains all the necessary information to form all the necessary files for that date.

Finally, making use of the FileManager class, the program creates all the necessary files for the date.

1.2 Utility and Validation Classes

In this section, the uses of the different Utility and Validation classes are explained. Most of the classes here are non-instantiable and hence have a private constructor. This is done because the methods declared are static, and are independent of the instance of that class. For example, since the information for validating inputs is provided in the specification, the class for input validation is declared as final and has a private constructor, so no instances can be created. It contains the information as static attributes which can be accessed without creating an instance of the class, and that is the same for any program run.

The utility and validation classes are implemented to modularise the code. Different validation classes validate the different types of classes, while the utility classes handle the processing of a certain type of class. Even though all the code that is related to handling orders could be placed in the Order class, having an OrderUtil class makes the code more readable, workable, and modularised. Orders can be passed as arguments to the static methods of OrderUtil to perform validation for example. This makes it so that the Order class only contains the code dealing with the attributes of the instance, and the handling of orders is done externally.

1.2.1 InputValidation

All of the methods within this class are used to validate inputs, and the methods and attributes are static so that they can be used without instantiating this class.

The class contains static attributes for validation of the input representing :

- ISO8601 date format,
- Base URL backup to be used if the URL provided in the program run is invalid,
- Minimum and maximum dates for program execution.

The arguments passed at the entry point of the program are passed into the public method "isValidInput" of the class, which makes use of other private methods in the class to ensure every argument is valid. This execution checks that the Date is in a valid format, and is in the range allowed of dates, the URL is a valid URL and appends a slash to the URL if it does not contain one in the end.

1.2.2 DateValidator

The class which is used for validating date strings, and converting between Date and String classes. A public constructor allows for instantiating of this class, and a String is dateFormat is the only parameter allowed. This is done so the program can make use of the different date formats needed in the spec (yyyy-mm-dd for program input, and mm/yy for validating credit card expiry dates). Makes use of SimpleDateFormat to be able to parse the date strings and convert them into dates of that format and vice versa.

1.2.3 MovementUtil

Class with a private constructor to avoid instantiation. This class contains the path-finding algorithm that will be discussed in section 2.

It is used to handle the movement of the drone and contains static methods to work out some of the needed information. This will be further discussed in section 2.

1.2.4 OrderUtil

Class with a private constructor to avoid instantiation. It serves the purpose of validating, setting outcomes, and creating deliveries from orders. The most important use is the method "getSortedValidOrdersFromOrders". This method validates each order with the method checkOrder, which has validation for all of the possible outcomes of an order, and sets the outcome to "ValidButNotDelivered" if the order is valid.

This is done prior to the execution of the path-finding algorithm, as we only want to find paths for orders that are valid. The method also assigns a restaurant to valid orders and sorts them based on the distance from Appleton tower (location specified as a constant of the class) to the restaurant. This is done to maximize the metric of the specification, where they ask to deliver the maximum number of orders each day. Since the drone has a limited battery life of 2000 moves, the best way to maximize the orders delivered is to deliver those that are closest to Appleton tower first.

1.2.5 IlpRestClient

The class contains a baseUrl parameter to instantiate depending on the URL provided at the program entry - if the URL is invalid, a backup URL is used. It is used to deserialize data fetched from a URL formed from the base URL and an endpoint.

It implements the method deserialize, which takes an endpoint and a class to deserialize the data. It uses ObjectMapper to deserialize the data, and the DeserializationFeature of fail on unknown properties is set to false to avoid the program from crashing when an unknown property is found. This method is used throughout classes of the program, such as Order, and Restaurant, in a static manner, to fetch orders and restaurants from the rest server and deserialize them into instances to be used.

1.2.6 FileManager

The class implements three static methods to create the files at the end of a program run. Makes use of FileWriter, JSONObject and JSONArray to create the files.

The class serializes the data for each file into the respective JSONObjects and creates the files for the date provided at the program entry.

1.3 Enumerators

1.3.1 CompassDirection

Holds all the possible directions a drone can move in, used throughout the program to ensure the robustness of always using the correct angles.

1.3.2 OrderOutcome

Holds all the possible outcomes of orders. It is used in the OrderUtil class to validate the orders, and each order has a value from OrderOutcome assigned. Used for the robustness of outcomes.

1.4 Area and its sub-classes

Class Area is the only class in the program where inheritance is used. It contains the coordinates of the points enclosing an area as an ArrayList of LngLat. The instances of this class are used to check for points being inside certain areas. Having both NoFlyZones and CentralArea extend this class allows for the same method to check if a point is in an area.

Area has two constructors: one is a JsonCreator, for deserializing the noFlyZones, and the other one is for creating an area given an array of points. The need for two constructors comes from the different JSON formats for the areas provided in the Rest server.

1.4.1 CentralArea

Extends the Area class, and contains a method for deserializing the centralArea endpoint from the Rest server. This method deserializes the endpoint into an array of LngLat, which makes use of the Area constructor with an array of LngLat to create an Area instance with these points.

1.4.2 NoFlyZones

Extends the Area class, and contains a method for deserializing the noFlyZones endpoint from the Rest server. Makes use of the JsonCreator constructor of the Area class to deserialize the endpoint into an array of Areas.

1.5 Drone

Class used to deliver the orders for a day. On instantiation, the number of moves is set to 2000, stored as a constant in the class. Its purpose is to deliver orders until it runs out of moves.

1.6 LngLat

Record class to represent points in Longitude and Latitude. It contains methods to check if a point is in a given area, check distances to other points, check if the points are close to one another (within the constant tolerance defined in the specification), and to find the next position after a move in a direction.

It is used throughout the program to represent all of the coordinates at any point, allowing for all the methods being used for any coordinate.

1.7 Instantiable Classes

In this sub-section, the remaining instantiable classes only contain getter and setter methods, and some contain the method to fetch members of that class from the Rest server. These classes are used to either deserialize data from the rest server, help with finding the deliveries, or for serializing data to write to the output files.

1.7.1 Order

Class for deserializing orders endpoint for a specific date.

1.7.2 Restaurant

Class for deserializing restaurants endpoint.

1.7.3 Menu

Class for deserializing restaurants endpoint, as the menu is another record that needs a class to be able to deserialize into.

1.8 Move

Contains all of the attributes of a specific Drone Move. Used to serialize into a JSONObject to write to the flightpath file.

1.9 Delivery

Contains all of the attributes for order delivery. Used to serialize into a JSONObject to write to the deliveries file.

1.10 RouteNode

Contains all of the attributes for a route node. This class will be discussed in section 2 along with the drone control algorithm.

Section Two: Drone Control Algorithm

2.1 Maximising Orders

According to the specification, one of the important metrics is to maximize the orders delivered in a day. Before explaining the drone controlling algorithm, it is important to see how the program maximizes the orders delivered in one day.

Before running the drone control algorithm, the program runs the method "getSortedValidOrdersFromOrders" from the OrderUtil class on all the orders. This method sets the orderOutcome of orders by checking if they are valid. If the Order is valid, they are set to ValidButNotDelivered, which can be changed to Delivered after the Drone delivers that order. The method returns an array list of orders with the outcome of ValidButNotDelivered. This array list is sorted based on the distance from the restaurant of that order to Appleton Tower so that most orders are fulfilled before the drone runs out of moves.

2.2 Initalising Drone

Before the orders are delivered, the drone is initialized. The remaining moves are set to the constant 2000, and the flag isLastOrder is set to false. The drone has the method deliverOrders which is run with the array list of ordered valid orders.

2.3 A star pathfinding algorithm

For each of the orders, the drone runs the calculateRoute method in the MovementUtil class, from either Appleton to a restaurant or vice versa. This method is an implementation of the A star pathfinding algorithm. The method returns an array list of route nodes, which have all the necessary information in order to build the files for output

2.3.1 How it works

A star pathfinding algorithm is an algorithm to find the shortest path from one node to another node in a weighted graph. This algorithm ensures always finding the best route for that graph but may be memory and run-time inefficient.

By iteratively picking the best route so far, and looking at the possible moves from that route, the algorithm works out the most efficient route.

In order for the algorithm to find what the best route so far is, we need a metric to analyze how good the routes calculated so far are. To do this, we need to give nodes an estimated score and compare them based on this.

2.3.2 Heuristic to compare nodes

To compare the nodes, they are given an estimated score heuristic. In this context, the chosen heuristic is the distance from the node to the endpoint, plus the total distance traveled to get to that node. For this, it is needed to store a route score and an estimated store in all nodes.

2.4 RouteNode and its attributes

As mentioned above, the route nodes need these two attributes in order to calculate which nodes to take as the best route, but other attributes are also necessary. Since the program needs to build files for flightpaths, which consist of move records, the nodes also include information on what direction they come from, the orderNo for that route, and the ticks since the start of the

calculation. The route nodes also contain an attribute current, which is the LngLat instance of the coordinates of that node, and the previous, which is the LngLat coordinate of the previous node in the graph. The last attribute is a Boolean to check if the path of the node has left the central area, to ensure it does not enter it again.

RouteNode has two constructors: One to initialize the first node of the algorithm, and one to initialize all the other RouteNodes in the main loop of the algorithm. The constructor for the first node contains parameters for current, previous, routeScore, estimatedScore, leftCentralArea, orderNo. The constructor for nodes in the main loop contains only parameters for current, direction, orderNo. This is because when nodes are created in the main loop, the routeScore and estimatedScore are set to infinity so that they are always considered. The remaining attributes are set accordingly once the algorithm considers the node for the route.

2.4.1 Ticks since the start of calculation

To calculate the ticks since the start of calculation, the algorithm sets a start tick when it is executed. Every time a node is considered for the final route, the difference in ticks at that point is what is set as the attribute. The ticks are set using System.nanoTime(). A static method is used to change nanoseconds into ticks by dividing the value by 100.

2.4.2 Need for the previous attribute

Since the algorithm tries many routes to get to the endpoint, the final route is not known until one of the nodes is close to the endpoint. For this reason, each node has to have its previous node as an attribute. When a node reaches the end, the algorithm uses backtracking to build the route from the start point to that node by recursing through the previous nodes.

2.5 Implementing A star without a weighted graph

When looking to implement the A star algorithm in the context of this problem, we do not have a clear graph: i.e There are no predefined routes from a certain node to another one.

A possible solution would be to define a grid and create nodes for all of the squares of the grid. The problem with this approach is that the drone moves in 16 directions, so it would not be possible to create a grid of squares for the drone to move in.

Instead, the approach taken is to create the nodes as the algorithm is taking place. At each position, nodes are created at the 16 positions the drone could move into.

2.6 Storing nodes

2.6.1 Open set

For this algorithm, an open set is implemented as a priority queue. This open set consists of the nodes that are currently being considered as potential nodes to be included in the final route. For every iteration of the program, the head of the open set is considered the next possible node. At the start of the algorithm, the open set is just a node consisting of the starting point of the route.

The compareTo method in RouteNode is overridden to make the nodes with the lowest estimated score have the least value in the priority queue. Since the priority queue's head is taken to be the node with the lowest value, it always takes the node with the lowest estimated score as the next node to check.

2.6.2 All Nodes

All nodes is implemented as a hashmap, which has LngLat objects as keys, and RouteNodes as their values. Every node that is created is added to the all nodes hashmap, with its location

as the key. This is used for backtracking and for checking if a node has already been visited. When the program eventually gets to the end node, to build back the route, it gets the previous route node objects by searching the hashmap for the LngLat keys of their previous attribute.

2.7 Algorithm execution

At the start, the open set is just set to a route node representing the starting point. The program then enters the main loop, which is run until the open set is empty, or the algorithm has found a node that is close to the objective location.

2.7.1 Main loop of the algorithm

The main loop is implemented with a while loop that exits if the open set is empty. At every iteration, the algorithm removes the head from the open set priority queue and uses this as the node to check for the next possible moves.

With the node from the head of the open set, the first operation is to check if the node has reached the endpoint. To do this, the algorithm gets the current location of the node, and using the method "closeTo" it checks if the distance to the end location is less than the tolerance. If it is, according to the specification we can assume that the points are equal and therefore the path has reached the end location.

If the node is not at the end location, however, the algorithm has to find new nodes to append to the open set. To do this, as we do not have a graph with connections, we create the connections "on the fly" by looping through all the directions. Nodes are created at the 16 coordinates the drone could move into from the current position. Each of these is 0.00015 degrees away from the current node, and each represents a move in a direction from the CompassDirection class. These nodes are initialized with the constructor that does not have an estimated score or route score, and these are set to positive infinity. Before creating a node for a location, the algorithm checks if the node already exists in allNodes hashmap for that location. If it does, it takes that node, and if it does not, a node is created as mentioned above. After validating that the node is valid, the node is added to the allNodes hashmap.

A calculation is made to see if the node should be added to the open set. If the node already existed in the allNodes hash map, the algorithm checks if the current path considered to get to the node is better than the one that already exists. If it is, it builds the information missing in the node (previous node, route score, estimated score), and then adds it to the open set. In the case that the node did not exist in all nodes, it is always added to the open set, as the route score is set to infinity, which is always bigger than the route score needed to get to that location. At this point, the ticks since the start of the calculation are also set for that node.

2.7.2 Validating node locations

Before checking if a node can be added to the open set, we first have to check if the node is valid. The constraints are that the location of the node cannot be in a no fly zone, and that if it has left the central area, it cannot go back into it. The method "isValidNode" checks if the node adheres to these constraints and returns a Boolean to evaluate if the node should be considered.

2.7.3 Building the route

When the end condition of being at the endpoint is satisfied, the algorithm initializes an array list of route nodes. Using a while loop, the algorithm does backtracking and appends the previous node, until there are no more previous nodes.

2.8 Ensuring the drone does not run out of moves

The drone has an attribute to check if the current order being delivered is the last order. This is done both to ensure that the drone does not run out of battery and to set the orderNo of the path from the last restaurant to Appleton tower as "no-order".

To check if the current order is the last order, the program calculates the route from Appleton to the restaurant to deliver the order. It finds the moves needed to do so. If the moves * 2 (moves taken to go to Appleton and back) is more than the remaining moves minus these moves, the Boolean is set to true, and the order is set to the last order. This is because since the drone is delivering orders in a sorted manner, the next order will always take at least the same amount of moves as the current order being delivered.

After each iteration of delivering an order, the current moves of the drone are updated to subtract the moves taken to deliver that order.

Flightpaths for 2023-01-02

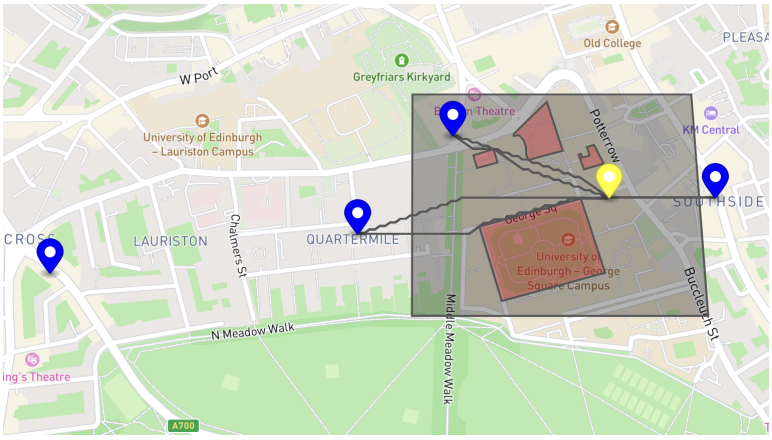

Figure 2: *Flightpaths for 2023-01-02 rendered in geojson.io*

Flightpaths for 2023-04-28

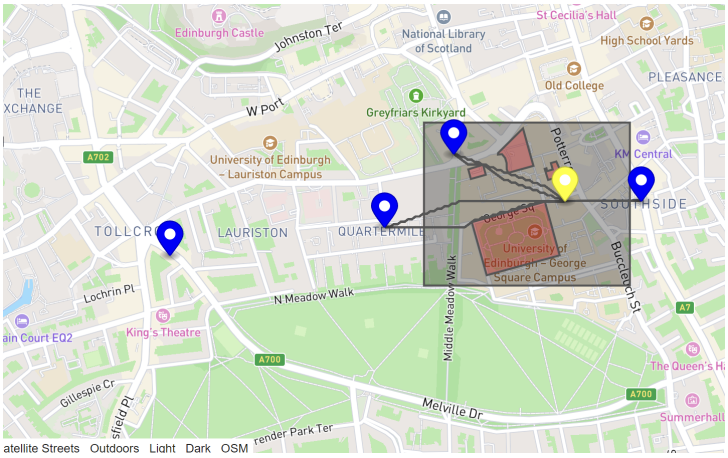

Figure 3: *Flightpaths for 2023-04-28 rendered in geojson.io*

Reference for A star algorithm: https://www.baeldung.com/java-a-star-pathfinding