PizzaDronz:

Software Architecture and Drone Control Algorithm

B140495

Informatics Large Practical

Software Architecture

This section provides an overview of architectural decisions made on the design of the PizzaDronz system. Good software architecture design is important as it ensures software projects are functional, scalable, and maintainable. PizzaDronz is designed with the following in mind: afforming to object-oriented programming principles and good software architecture characteristics such as modularity, loose coupling, high cohesion, abstraction, and reusability [1].

Classes within the system are created so that they all serve distinct purposes thus improving the modularity of the system (meaning software system functionality is separated into independent components) and promotes high cohesion (which refers to all elements within a single component works towards a single well-defined goal). They are also loosely coupled, this means components in the system are as least dependent on each other as practically possible, which makes them easier to modify, test and maintain. This design allows future development on one area of the system to have the least impact on other areas. All important data in the system is stored in private fields, with appropriate getters and setters defined, even if they are not all necessarily used. This makes the code more complete and easier to understand for future developers as when they need to use getters and setters, they do not need to figure out which are missing. Furthermore, order validation methods are also kept private to promote abstraction of the system (which refers to hiding specific implementations of the system and only shows required information to the user), this is important as we do not want the user to know how we validate information such as credit card details which may allow for potential abuse. Lastly, the implementation of certain classes, such as *Client*, *FileOutput*, and *ValidationHelper* allows for reusability of components in multiple contexts of the system, therefore code duplication is minimized allowing the code base to be understood easier.

Classes

Client

All dynamic content the PizzaDronz system needs is provided by the REST-server, therefore the action of connecting to and retrieving data from the server is essential. As this objective is distinct from other purposes of the system, a *Client* class is created to encapsulate methods for requesting, receiving, and parsing data. The class constructor will take in a base URL so that this can be used in all following requests rather than taking it in as a parameter each time. However, one potential design flaw of the current implementation is that checks for determining how to read data i.e., which class to read it to, from each of the subdirectories is hard coded. This could be generalized to improve reusability if subdirectories were to be edited, added, or removed.

LngLat

Another crucial aspect of the system is having a means to represent coordinate locations so that the real world around the drone can be modeled. This is another distinct objective of the system so must be created in a separate class. The *LngLat* class was originally implemented as a record, which had benefits of better readability and immutability, where automatically, the values of their fields could not be changed after creation of the record. However, this had to be sacrificed for better functionality as more details were incorporated to a LngLat coordinate representation. To utilize the A* algorithm for finding flight paths for the drone, the LngLat definition needed to be expanded to represent a node. The node must contain a reference to its parent node, neighboring nodes, and corresponding variables used in the A* algorithm, amongst other things. These variables can change at any time, so a record class was not sufficient, and a regular class implementing the Comparable interface is used instead (allowing comparisons of LngLat objects, again required for A*).

MenuItem, NoFlyZone & Location

These classes are similar in that they serve a purpose of solely reading into and storing data. These are all necessary as they represent separate elements of the system, that other components need to read data from.

CentralArea

This class is implemented with a singleton access pattern, which means there is only one instance of a class, and a corresponding global access point is provided. As central area is quite compact, it is sufficient to set this data once and allow this instance to be shared globally.

Direction, OrderOutcome & Subdirectory

These are all implemented as enum classes. The directions, order outcomes and subdirectories are all set from the beginning and do not depend on any input, so it is appropriate to implement them as enum classes. The “INVALID” enum from the original specification was removed as no cases matching this was identified after checks for the rest of the order outcome enums. However, a new enum of “VALID” was added to signify that an order had passed validation and is ready for further calculations. This means only calculations for valid orders are carried out later in the system, rather than for all orders, increasing efficiency which is important in a real-world system. The subdirectory enum pairs a REST service

subdirectory with its corresponding string path, so the path can be easily modified without breaking the system.

Restaurant

This class allows restaurant data to be read in and stored. It also contains two static methods which perform operations on a list of restaurants, namely matching a list of orders with its corresponding restaurant and returning a list of restaurants with at least one order (*matchRestaurantsAndOrders*, *getOnlyRestaurantsWithOrders*). These methods relate to more than one restaurant so it can be argued that they do not belong in the Restaurant class, the other option would be to move these two methods to a helper class. However, creating a class with no state and only consisting of two small, use case specific static methods introduces unnecessary complications and can make the code base harder to understand. Therefore, the current approach is sufficient, and a helper class can be considered if more related methods are required to be incorporated in future development.

Order

This class allows order data to be read in and stored and contains methods for validation of orders. For our current purposes of date validation, we are assuming a specific locale for date formatting, however there are multiple to choose from. The system would not be able to adapt to a change in locale used for date formatting, future development should aim to deal with this ambiguity and make the system more resilient to change. Furthermore, the current system only allows for specific credit cards to be used (16-digit Mastercard and Visa), this should also be modified to allow for scalability and adaptability. One thing to note about the order outcomes is that the corresponding validations are checked in an order that ensures later checks can be completed. For example, the pizzas are ensured to be defined first before validating that the total cost is correct. Problems may arise if this order is disrupted, therefore future development should pay careful attention to this or refactor the implementation so that the order of operations no longer matters. Lastly, this class also contains two static methods that operate on a list of orders (*validateOrders*, *findOnlyValidOrders*), which may again be argued that they are more appropriately placed in helper classes, however again, creating classes with no state and only consisting of two small, use case specific static methods introduces unnecessary complications and can make the code base harder to understand, therefore the current approach is sufficient until more static methods are incorporated.

ValidationHelper

This class contains some helper methods for validation. These methods are not specific to validating an order, for example, *isNumeric* can be used whenever to check if a string is numeric, so it makes sense to create a separate class to house them. Although they are currently not used in any other class than the *Order* class, this structural decision allows for potential reusability in future development.

Drone

This class models a drone and contains relevant information. Important attributes such as the drone's battery life, path taken, and computation time is stored here. Methods for path calculation is also contained here as it is the drone's responsibility to figure out and execute a path from origin to destination. Methods and the path taken can be retrieved and used from other classes but the important attributes related to the inner workings of the algorithm is kept private, preventing any potential misuse of

the system by an external user. The specifications require a measurement for keeping track of length of time of path computation, this time measurement is currently started in one function and ends in the middle of another. This is not the best design as it may cause confusion when trying to understand the code, however this is required to obtain an accurate time measurement.

FileOutput

The function of outputting files is distinct from other functionalities of the system and justifies its own class. The constructor for this class allows a date string to be taken in and stored, so that each time a file with date signature is created, it can just use the date stored within the class. The methods of this class are split into those that creates output JSON objects corresponding to the desired output content and a generate file method that takes in a JSON object as a string then generates a file. Methods for creating JSON objects can be changed, added, or removed according to future specifications and it will not affect the *generateFile* method, as long as the parameters are of the correct type.

App

This class does not have any inherit logic, the main purpose this class serves is to direct the flow of the system. *App* will instruct other classes to perform operations in a sequential fashion until the goal is reached. One potential issue to note is that there are two checks performed in this class, namely the check that three arguments are taken in and the check that the date provided is valid. It can be argued that these checks should not be in the *App* class as no logic should be present here, and instead be in a helper class. However, these checks are very specific to one use case, namely that of the arguments being of length 3 and the date is of a predetermined date format. If more checks are required at a later stage of development, it may then be more appropriate to create a separate class to house tests. Therefore, these methods would not be reusable if put in a helper functions, thus the current approach is sufficient. Another potential issue is that if data retrieved from *Client* is empty, such as if the order for a particular day is empty, the system still performs all computations and returns empty files. Since the specification does not detail how to handle this case, the current approach is sufficient.

Drone Control Algorithm

The drone control algorithm is built upon the A* algorithm, with modifications to ensure a valid path according to the specifications is found. This section will first discuss the A* algorithm, then explore how it fits into the PizzaDronz system and finally, present possible improvements.

A* Algorithm

The A* algorithm is a heuristic graph search algorithm used to compute the shortest path between two nodes, it is an extension to Dijkstra's algorithm (by addition of a heuristic) [2]. This algorithm is well suited to the PizzaDronz system as each *LngLat* can be thought of as nodes and we will be able to obtain the Euclidean distance between nodes to use as the heuristic (a method to guide the algorithm to an optimal solution most of the time) to calculate the shortest path from origin (Appleton Tower) to destination (restaurant) or vice versa.

The most important part of the A* algorithm is the cost function *f* as defined below, where n is a current node.

$$f(n) = g(n) + h(n)$$

The cost function *f* is what we use to determine how good a candidate node is to be selected for the optimal path. The cost function f is the sum of the move function *g* and the heuristic function *h.* The move function *g* represents the cost it takes to get to node n, and the heuristic function *h* represents the estimated closeness between n and the target/destination.

The algorithm starts off with initializing two priority queues: open list and closed list. The open list will contain nodes that have been encountered but not analyzed yet, and the closed list will contain nodes that have been analyzed and have had their neighbor nodes added to the open list. Nodes with lowest *f* values will be at the start of the priority queue, and hence analyzed first

The process of analyzing a node includes first finding its neighboring nodes, in our case, this will create and return 16 *LngLat* objects (stored in the current *LngLat* node instance) obtained by travelling one unit distance (0.00015 degrees) in each of the 16 specified directions from the current node. Then for each of the neighboring nodes, if a path from the current node to the neighboring node does not violate conditions in relation to no-fly zones and reentering central area as defined in the specification, we will do either of the following. If we have not encountered this node before (i.e., not in open or closed list) then we will update its *g* and *f* functions (*h* is calculated during execution each time by obtaining the Euclidean distance from this neighboring node to the target/destination) and its parent node, then add it onto the open list. Otherwise, if we have encountered this node before, and the path we are using to get to the node now is more optimal than the previous path we encountered the node at, we will update its *g* and *f* functions and parent node, then move it from the closed list to the open list. After analyzing a node, we move it from the open list to the closed list.

We do this analysis for each of the nodes in the open list, beginning with the start node, until we reach a *LngLat* that is close to the target/destination (within 0.00015 degrees in distance). Now we can backtrack through the final node's parent to obtain a path. This will give a path from the destination node to the start node, therefore we need to reverse this to obtain a path from start to target/destination.

Problems and Solutions

The first problem encountered was that the algorithm took an extremely long time to compute a simple test path from Appleton Tower to the main library. The problem was the neighbours for each node was only created and added to the open list (if it satisfied conditions mentioned above). Since neighbours

are created each time for a node, it is not possible to determine if we have already encountered a neighbor as even if the new neighbor has same values as one already in either the open or closed lists, it will not be of the same instance. This was resolved by storing the neighbours of a LngLat node in a field of its own instance, therefore neighbours are distinguishable.

This increased the run time dramatically, however it still took longer than expected. The intuitive problem here is since there are 16 directions a node can travel, there will exist 16 neighbours for each node. This can become an extremely expensive computation, therefore the solution was to give a stronger bias towards nodes that are closer to the target/destination. To do this, the cost function *f* was adjusted to the following.

$$f(n) = g(n) * 0.8 + h(n)$$

This is also known as the weighted A* algorithm. The move function *g* accounts for less in the overall cost function whilst the heuristic function h accounts for more, therefore it is less biased towards travelling backwards (in the direction of the origin node).

Overall Implementation

The runtime of this implementation is largely dependent on the initial path calculations for each restaurant. This is because only one path is found from the origin (Appleton Tower) to the restaurant, then a reversed copy is combined with the first path, resulting in a complete path from origin to restaurant and back to origin. The last *LngLat* in this list is duplicated to account for a hover move at origin (Appleton Tower) for pizza delivery. Since the path is reversed, the restaurant position will be duplicated, therefore accounting for a hover move, for pizza collection. This path is stored in the corresponding instance for the restaurant, which will be used as the only path for orders at this restaurant.

One important thing to note is that only restaurants with at least one order for that day is used in the above calculation (this is determined by *getOnlyRestaurantsWithOrders* in the *Restaurant* class), as there is no point using computation power to calculate paths to a restaurant if there are no orders to be fulfilled.

Then, given a list of valid orders, the algorithm will sort these in order of the flight path lengths for their corresponding restaurants so that the maximum number of orders can be delivered before the drone battery runs out. The algorithm will iterate through the sorted orders starting with the lowest flight path length for the corresponding restaurant, retrieve its path and combine it to the total flight path for the day until a path can no longer be added following the constraint of maximum 2000 moves. The computation time for this section is not too large as we are just retrieving the precomputed flight path for each restaurant and not performing any flight path calculations.

To check that the drone does not leave the central area once it goes back in, it is sufficient to do the following. As we only find one path from the origin (Appleton Tower) to a restaurant, we can check that when the path starts at the origin and leaves, it cannot come back in, meaning it can only cross the central area boundaries at most once. This is also true when the path is reversed and hence still fulfils the central area check as mentioned in the specifications.

A slightly different approach is taken for obtaining a path that avoids no-fly zones. This is done by making sure the line between a node and a potential next node does not intersect with any of the edges of no-fly zones. If it does intersect, we discard this potential next node and move on.

The runtime of this implementation is O(find path for each restaurant. + add path for each order), this will run particularly well if there are many orders to a few restaurants, as the “find path for each restaurant” is the dominating term here.

Improvements

Currently the central area check is implemented by ensuring that if the start node is in the central area, and the current node is not, then the potential next node must also not be in the central area. There exist certain edge cases where this check may not work as intended. Namely, the case where although the start node is in central area, and current node and potential next node are both not in central area, but the path taken between the current and next nodes may cross the central area (for example it may cross a corner of central area). This is a potential robustness issue to be resolved in future development.

Another potential issue is that the implementation currently assumes a path will always be found, therefore checks should be put in place to catch cases where determining a path is not possible given specified constraints, allowing for more robustness of the system.

Lastly as the implementation currently just calculates one path from origin (Appleton Tower) to restaurant and uses the reverse of this path to get a combined path, this combined path may not always be optimal. Instead, we could implement a bidirectional search which starts two A* searches in parallel one from Appleton Tower to restaurant and one from restaurant to Appleton Tower, to yield an optimal path when they meet from both directions [3].

Flight Path Results

Below shows results of drone flight path calculations for two separate days, 2023/01/01 and 2023/01/02. We can see the drone runs out of moves before being able to deliver orders for the furthest away restaurant, as the orders are sorted as described above.

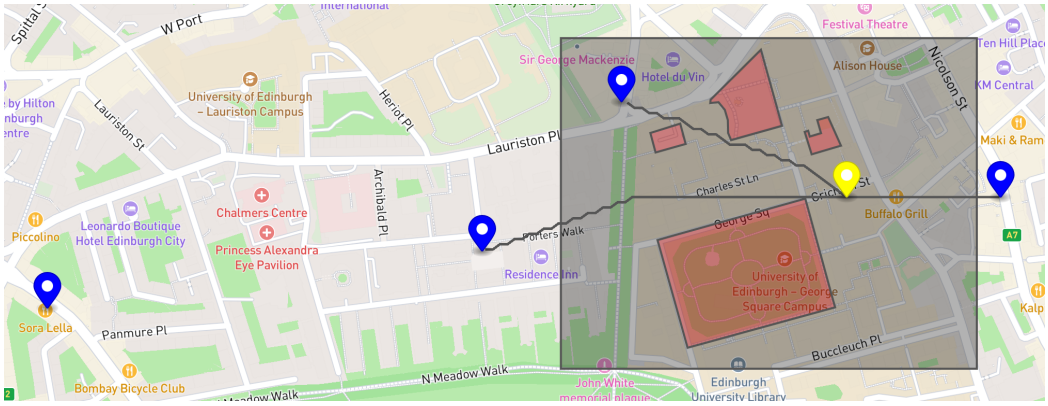

Figure 1. Results of the drone flight path for 2023/01/01

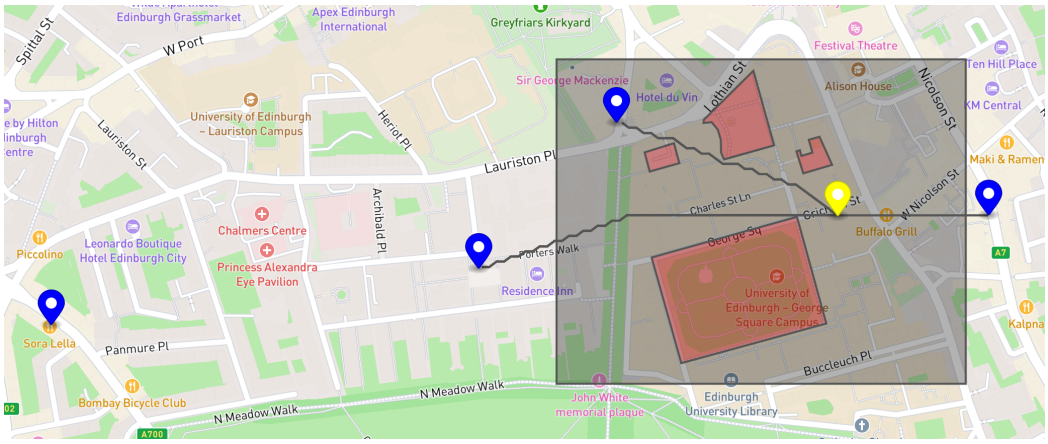

Figure 2. Results of the drone flight path for 2023/01/02

References

[1] *Fundamentals of Software Architecture* (2022) *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/fundamentals-of-software-architecture/ (Accessed: December 10, 2022).

[2] Zobenica, D. (2021) *Graphs in Java - A* algorithm*, *Stack Abuse*. Stack Abuse. Available at: https://stackabuse.com/graphs-in-java-a-star-algorithm/ (Accessed: December 10, 2022).

[3] Patel, A. (2011) *Variants of A**. Available at: https://theory.stanford.edu/~amitp/GameProgramming/Variations.html (Accessed: December 10, 2022).