ILP Report – Ioulios Patmanidis

Software architecture

Classes

The application consists of 15 classes. I decided to structure these classes according to their functionality, as shown on Figure 1. The reason for choosing these specific classes will be analysed from top to bottom as shown in the figure.

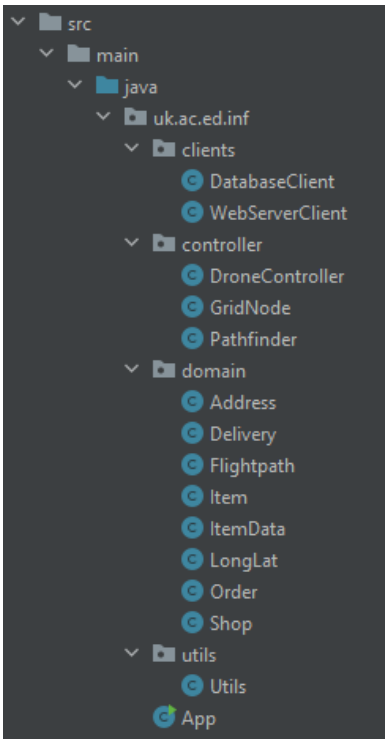

Figure 1: Application structure

DatabaseClient - clients package

This class is responsible for all the interfacing between the application and the Derby database server. It contains functions that allow read and write operations to be performed on the database.

By having all interactions with the database abstracted within this class, the application becomes much easier to debug, and the reusability of the code greatly increases, since in case the database server is changed, only this class needs to be altered.

WebServerClient - clients package

This class is responsible for all the interfacing between the application and the Web Server running on the local machine. It contains functions that fetch and store information in the appropriate format, ready to be used by the rest of the application.

Similar to the Database class, abstracting the web server allows for easier debugging during development, as well as increases the maintainability of the code.

DroneController - controller package

This class is used to control and log the movement of the drone. It utilizes a Pathfinder instance to find the path the drone should follow, and then attempts to match it as closely as possible, given the limitations of the drone movement.

I decided to use this class in order to abstract the methods and data structures that control the drone, allowing for easier debugging during development. This class also allows me to test the pathfinding algorithm and the drone movement separately.

GridNode - controller package

This class represents a single node on the virtual graph utilized by the Pathfinder class. It contains methods utilized by the pathfinding algorithm used, Theta Star, as well as some helper methods.

The main purpose of this class is to conceptually help with the implementation of the Theta Star algorithm. By encapsulating all information about a node in the virtual grid inside this class, it becomes much simpler to provide helper functions and all the data that each node requires can be grouped into one place, making the code more readable and providing structure.

Pathfinder - controller package

This class implements the algorithm used to find a path between two points on the map, while avoiding no-fly-zones and staying inside the confinement area. The algorithm used is Theta Star.

The Pathfinder class allows all the pathfinding logic to be abstracted, and as such its functionality can be reused in the future for other systems, as it is not tied to the drone controller.

Address - domain package

This class represents addresses stored in the web server. It is used only for parsing address information from JSON formatted strings found on the web server.

This class is needed because it allows the use of the Gson library to parse appropriately formatted JSON strings into Address objects.

Delivery - domain package

This class represents a completed delivery made by the drone, containing the exact fields that also appear in the “deliveries” table on the Derby database.

Its purpose is to simplify adding delivery entries on the database by grouping all the fields of each entry into a single object.

Flightpath - domain package

This class represents a single move or step made by the drone, containing the exact fields that also appear in the “flightpath” table on the Derby database.

Similar to the Delivery class, its purpose is to simplify adding flightpath entries on the database by grouping all the fields of each entry into a single object.

Item - domain package

This class represents individual items that stored in the web server. It is used in conjunction with the Shop class for parsing shop information from appropriately formatted JSON strings found on the web server.

This class is needed because it allows the use of the Gson library to parse JSON strings into item objects.

ItemData - domain package

This class is used to store Shop and Item data more efficiently, as well as provide a function to calculate the price of a collection of items. The way it achieves that is by using two HashMaps to map item names to prices and item names to Shops. This allows for more efficient searching for item prices and Shops that sell the item.

Its purpose is simply to encapsulate item and shop data from the web server in a more efficient manner.

LongLat - domain package

This class describes the standard spatial unit used throughout the application. This unit is a pair of longitude and latitude values. It also contains all the necessary helper functions regarding the coordinate system used in the application.

This class helps encapsulate the coordinate system along with any useful functions into one place.

Order - domain package

This class represents a customer order that needs to be delivered by the drone, containing the exact fields that also appear in the “orders” table on the Derby database.

Its purpose is to simplify the reading process of Order entries on the database by grouping all the fields of each entry into a single object.

Shop - domain package

This class represents shops which stored in the web server, where the drone can pick up items from. It is used in conjunction with the Item class for deserializing shop information from the web server.

This class is needed because it allows the use of the Gson library to parse JSON strings into shop objects.

Utils - utils package

This class contains any utility functions that did not fit into a particular class, and they could be reused in the future.

App – uk.ed.ac.inf package

This is the running class of the application that contains the main method. Its purpose is to combine the functionality the other classes provide in order to run the application according to the arguments it receives from the user.

Packages

As the number of classes I used increased, I decided it would be best if I separated the classes into packages. This way, classes will be grouped according to their functionality and the application structure will become more organized, as opposed to having all 15 classes into one place.

The packages I used were clients, controller, domain and utils.

The clients package contains the classes that need to connect to an external source in order to operate. These classes are the WebServerClient and the DatabaseClient.

The controller package contains classes that are responsible for the navigation of the drone. These classes are DroneController, Pathfinder and GridNode.

The domain package groups all classes that are made to simply store information. These classes are, Address, Delivery, Flightpath, Item, ItemData, LongLat, Order and Shop.

The utils package contains the Utils class. It is not necessary for this package to exist, but it makes the class grouping more consistent.

Finally, all the packages mentioned above are inside the uk.ed.ac.inf package, which also holds the App class of the application. This is the top-level package of the app.

Drone control algorithm

During the planning phase of the Drone control algorithm, I had in mind 2 basic characteristics that I wanted the algorithm to have. Firstly, the algorithm must navigate the drone inside the containment area while avoiding the no-fly-zones, and secondly, the algorithm should aim to maximize the Sampled Average Percentage Monetary Value delivered by the drone. To maximize that value, the drone must be as efficient as possible with its movement, since the main limiting factor is the number of steps it can perform, which is currently 1500 steps.

With these two characteristics in mind, I decided that the best choice would be to implement the Theta Star (or Theta*) algorithm, which will be utilized by the drone to find near optimal paths

towards its destinations. The Theta* algorithm is an any-angle pathfinding algorithm that works on grids and builds up on the ideas of the A Star (or A*) algorithm.

Since this algorithm requires a grid to operate on, the program converts the confinement area into a virtual grid, with nodes evenly spaced out. This means that multiple LongLat points are mapped to the same node, and each node essentially occupies a square on the map. I decided that the spacing or granularity of the grid needs to be smaller than the closeTo distance because if it was greater, some points on the map would be unreachable.

However, the Theta* algorithm alone cannot fulfil the needs of the drone since it can only move at specific angles. Due to this limitation, I also developed an algorithm that tries to move the drone as closely as possible along the path provided by the Theta Star algorithm.

A* algorithm

In order to fully explain the implementation of the Theta* algorithm, the A* algorithm must be presented first.

The A* algorithm begins from the starting node or cell on the grid, and successively explores neighbouring nodes, until it finds the destination node. The neighbours of a single node are simply the 8 nodes that are adjacent to it.

The order at which the neighbours are explored dictate the characteristics of the path. The two metrics taken into account are the length of the shortest known path from the neighbour back to the start, which I call *scoreFromStart*, and the distance of the neighbour to the destination, which I call *distanceScore*. These two metrics are combined into one final score which I call *totalScore*. Neighbours with the least *totalScore* are considered first. (Rabin, Nash and Koenig 2015).

This ordering essentially combines two algorithms, the Greedy Best-First Search which only takes into account the distanceScore and it focuses mainly on speed, and Dijkstra's Algorithm which only takes into account the scoreFromStart and focuses on finding an optimal path (Patel 2021).

This results in an algorithm that balances between speed and accuracy, making it very versatile as the score weighting can be adjusted to favour one over the other.

Going into more detail, the algorithm uses two sets, the OPEN set, and the CLOSED set. OPEN contains all the nodes that are to be examined or "explored", while the CLOSED set contains the nodes that have already been examined. Initially, OPEN contains only the starting node and CLOSED is empty. Each node also keeps a pointer to its parent which is the node it was expanded from. This helps with the calculation of the scoreFromStart and the construction of the final path.

After initialising the OPEN and CLOSED sets and adding the starting node to OPEN, the main loop of the algorithm starts, and runs until the OPEN set is empty.

Inside the main loop, the node with the lowest totalScore is picked and if it is not the destination, its neighbours are added to the OPEN set, given that they can be traversed (how this is checked will be discussed later in the document). The current node is also added to CLOSED.

Now the algorithm loops through all neighbours retrieved. If a neighbour is already in the CLOSED set or the OPEN set through a shorter path (with a smaller scoreFromStart), this neighbour is not considered again. Otherwise, the totalScore of the neighbour is calculated and set. If the neighbour has not been visited before (it is not in either CLOSED or OPEN), then the neighbour is added to the OPEN set.

Since every node also stores a pointer to its parent, reconstructing the final path just requires to recursively follow each parent node from the destination all the way back to the start.

Pseudocode for the basic idea behind the algorithm can be seen in Figure 2 below.

```
// Pseudocode for A* algorithm

Initialise OPEN, CLOSED
OPEN.add(start)
while (OPEN is not empty)
    currentNode = OPEN.getBest()
    OPEN.remove(currentNode)
    if currentNode == destination
        return PATH
    ClOSED.add(currentNode)
    neighbours = currentNode.getNeighbours()
    for (neighbour in neighbours)
        newScoreFromStart = currentNode.scoreFromStart + dist(currentNode, neighbour)
        if (CLOSED.contains(neighbour) AND neighbour.scoreFromStart < newScoreFromStart)
            skip this neighbour
        if (OPEN.contains(neighbour) AND neighbour.scoreFromStart < newScoreFromStart)
            skip this neighbour

        neighbour.scoreFromStart = newScoreFromStart
        neighbour.distanceScore = dist(neighbour, destination)
        neighbour.totalScore = scoreFromStart + distanceScore

        if (!OPEN.contains(neighbour) && !CLOSED.contains(neighbour))
            OPEN.add(neighbour)
return PATH_NOT_FOUND
```

Figure 2: A implementation pseudocode*

From A* to Theta*

A* does a great job of finding an optimal path, but the path it constructs is limited to moving from a node to an adjacent one, which in the real world is not actually the optimal path as it does not allow movement towards any angle. This is what the Theta* variation tries to fix. This is achieved by allowing the parent of any node to be **any** visible node, not one necessarily adjacent to it (Rabin, Nash and Koenig 2015).

This change does not radically alter the pseudocode for the A* algorithm. The only change needed is to check if the neighbour being explored has line of sight with the parent of the current node. If it has, then the parent of the current node also becomes the parent of the neighbour. This check should be performed before the neighbours' scoreFromStart is set.

The complexity added however is that there needs to be a way to determine whether there is line of sight between two nodes.

Line of sight

In the drone application, the only obstacle that can block line of sight is a no-fly-zone. These zones are parsed from the web server as collections of points that form polygons. A simple way to determine whether two nodes have line of sight with each other is to utilize the built-in java

library for geometry, java.awt.geom, and more specifically the Line2D object, which represents a line in 2 dimensions.

Using this library, all the edges of the no-fly-zones can be stored as Line2D objects, and then to check for line of sight we simply create a new line between the desired points, and we check if that line intersects with any of the edges of the no-fly-zones using the Line2D.intersectsLine() method.

IsWalkable

In addition to line of sight, we need a way to decide whether a node can be "walked-on" or not. This dictates whether a neighbour can be explored or not. To achieves this, during the initialisation of the virtual grid, if any "edge" of the grid node is inside a no-fly-zone, then this node is marked as not walkable. To check if a node is inside a no-fly-zone, I used the Path2D object from the java.awt.geom library which has a built in contains() method for points. By converting and storing the no-fly-zones as Path2D objects, this check becomes very easy.

Theta* implementation

All the behaviour described above is implemented in the Pathfinder and GridNode classes of the project.

The Pathfinder class contains a private method findPathOnGrid(), which given two GridNodes it returns a list of GridNodes that describe a path between them. This is the method where the Theta* algorithm is actually implemented. To increase efficiency, the open list is implemented as a priority queue with custom ordering based on the total score of the nodes. This method is used by the public findPath() method, which given a start and a destination LongLat, returns a list of LongLat objects that represent the path the drone should follow.

The GridNode class represents nodes on the virtual grid and provides functions for updating their scores and accessing their neighbours.

Drone controller algorithm

The Pathfinder class does an excellent job with providing a near optimal path for the drone to follow, but there needs to be an algorithm that actually moves the drone in real world coordinates. This is the purpose of the DroneController class.

First, the algorithm reads the current order, and forms a list of destinations it should move to, based on the position of the shops and the delivery destination.

Then, for every destination, it calls the findPath() method to form the path from the current position of the drone to that destination.

While the drone has not reached the next point on the path, it attempts to move towards it, making sure it moves according to its specification. If the drone reaches the end of the path, findPath() is called again to find the path to the next destination.

Finally, when the order is delivered, the algorithm checks if it can go back to its base with the steps it has left. This check is done by attempting to move the drone towards base and seeing if there are steps left. If it can, the flightpath of the delivery, the current position and the steps left after the delivery was completed get stored, and the algorithm returns true. If not, the delivery that was just completed is not saved, the position of the drone and the steps left get reset to those before attempting the current delivery, and the drone moves back to its base.

```
// Pseudocode for method deliverOrder()

method deliverOrder(orderList)
    currentOrder = orderList.pop()
    initialise List targets, flightpathList
    targets.add(shops)
    targets.add(customer)
    while (targets is not empty)
        currentTarget = targets.pop()
        path = findPath(currentPos, currentTarget)

        while (path is not empty)
            dest = path.pop()
            while (not closeTo(dest))
                currentPos = makeMove(dest)
                stepsLeft--
        hover()
        stepsLeft--

    if (stepsLeft > 0 && canGoBackToBase())
        flightpathList.add(thisFlightpath)
        lastOrderSteps = stepsLeft
        lastOrderPos = currentPos
        return true
    else
        orderList.add(current);
        stepsLeft = lastOrderSteps
        currentPos = lastOrderPos
        goBackToBase()
        return false
```

Figure 3: Pseudocode for deliverOrder()

Pseudocode for the basic idea behind this algorithm can be seen in Figure 3.

However, this algorithm is not complete. Since the drone can only move towards angles that are multiples of 10 degrees it cannot follow the path very accurately. This raises issues when the drone is very close to a no-fly-zone, and instead of going to the position specified, it moves into the no-fly-zone due to the rounding of the angle of movement.

In order to overcome this issue, the makeMove() function checks whether the drone can move towards the rounded angle, and if it cannot, it tries to find an alternative angle that would be valid. The alternative angles are simply angles that deviate 10 degrees from the previous, until a 360 degree circle is formed.

Pseudocode for this function is shown in Figure 4.

```
// Pseudocode for makeMove() method

method makeMove(Longlat origin, LongLat target)
    initialise list possibleAngles;
    int angle

    angle = origin.calculateAngle(target)
    if (not canMoveTowards(origin, angle))
        // Find a different angle that is in the general direction of the target
        possibleAngles = origin.calculateAngles(target)
        for each possibleAngle in possibleAngles
            if (canMoveTowards(origin, possibleAngle))
                tempPos = origin.makeMove(possibleAngle)
                if lineOfSight(tempPos, target)
                    nextPos = tempPos
    else
        nextPos = origin.nextPosition(angle)
    return nextPos
```

Figure 4: makeMove() method pseudocode

All that is left is for the public method deliverOrders() to call deliverOrder() for each order retrieved from the database for a given date. This function is used in the main method of the app.

Optimizations

After running several tests on the implementation described using the test data provided, the sampled average percentage monetary value of the delivery was hovering around 96%. Even though I was satisfied with that performance, I tried implementing some simple optimizations.

One optimization was to change the order of the shops the drone travels to pick up the item, based on which one is closest to the drone's position and the delivery destination. Since the drone delivers orders with items from maximum 2 shops, this was very simple to implement.

Another optimization applied was to prioritise orders that cost more. Since all orders have relatively similar paths, prioritising those that cost more ensures that the few orders that might not be delivered will have the least impact on the metric. I am aware that it does not always bring about the best results. However, during my testing I found that it increases it, so I decided to include it.

These optimizations increased the sampled average monetary value to approximately 99%.

Results

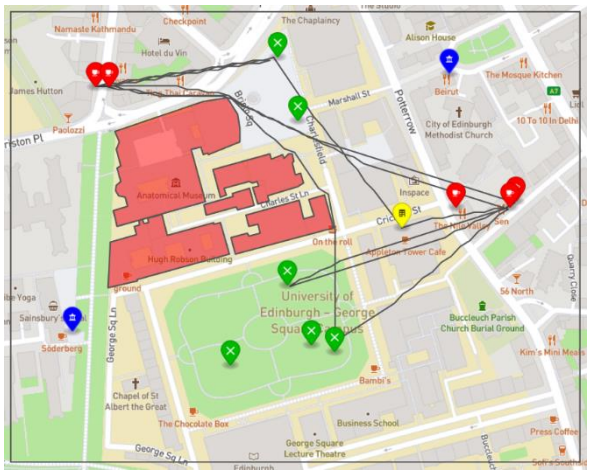

Figure 5: Flightpath of date 14/01/2022

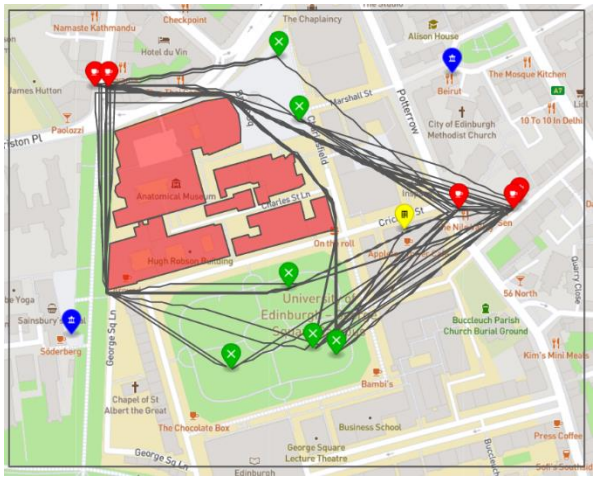

Figure 6: Flightpath of date 07/11/2023

Figure 5 shows a day where all deliveries were successful, and figure 6 shows a day where some orders were not delivered successfully. In both cases the drone manages to avoid the no-fly-zones and return back to Appleton Tower.

BIBLIOGRAPHY

- Patel, Amit. 2021. "Introduction To A*". *Theory.Stanford.Edu*. http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.

- Rabin, Steve, Alex Nash, and Sven Koenig. 2015. *Game AI Pro 2*. Boca Raton: CRC Press.