Informatics Large Practical

Coursework 2: Report

1st December 2022

Part 1. *Software architecture description*

The application consists of 23 public classes. 7 of them are navigation related classes, 7 are exception related, and the remaining 9 are for other various purposes (for example storing orders/restaurants, getting input, writing output, etc.). Instead of simply listing each class alphabetically and describing its purpose, they will be described as they appear in the program execution to help understand their purpose and allow easier readability of the report.

Retrieving console input and Rest server data

Firstly, the ***App*** class stores the *main* method which takes the console input. If exactly 3 valid String arguments were given, they are then processed in the ***OrderProcessor*** class, otherwise the system is stopped with an appropriate error message. The ***OrderProcessor*** class controls other classes to get the desired result (like Controller in Model-View-Controller pattern). The ***OrderProcessor*** class then retrieves the orders, restaurants and no-fly zones for the given day using static factory methods in ***Order***, ***Restaurant*** and ***Area*** classes respectively. These static factory methods (*getOrdersFromRestServer, getRestaurantsFromRestServer and getNoFlyZones*) all make use of the ***RestClient*** class method *deserialize*. This approach was taken as the *RestClient* class allows multiple classes to deserialize data from the Rest server in one place, with the rest of the deserialization and Rest server access details encapsulated in the *RestClient* class. Note that the *RestClient* class method *deserialize* is overloaded to allow two kinds of inputs: with date (for orders) and without (everything else).

Order validation

Each order validation functionality is in the ***Order*** class *validateOrder* method. The ***Order*** class has its own method for validating the order pizza names, price and count. The rest of the validation consists of validating the credit card details (card number, expiry date, cvv) – this functionality is grouped together in the ***CreditCardValidation*** class to reduce clutter in the ***Order*** class. The ***CreditCardValidation*** class has one public method *validateCreditCard*. It confirms that the credit card details are valid, otherwise throws an appropriate exception. This design allows abstraction of the credit card validation in the Order class – it does not need to care how the validation is done in the ***CreditCardValidation*** class, only cares about the outcome. Hence, it makes sense to have it in one publicly accessible method which informs the ***Order*** class about the outcome of validation. Combined with the pizza name/price/count exceptions, there are 6 exceptions in total:

- ***InvalidCardNoException*** – when the credit card number is invalid
- ***InvalidCreditCardExpiryException*** – when the credit card expiration is invalid
- ***InvalidCvvException*** – when the credit card cvv number is invalid
- ***InvalidPizzaCombinationException*** – when the pizza combination cannot be delivered by the same restaurant
- ***InvalidPizzaCountException*** – when the ordered pizza count is outside of the valid range [1; 4].
- ***InvalidPizzaNotDefinedException*** – when an ordered pizza does not exist.

These exceptions are all caught in the ***Order*** class *validateOrder* method where an appropriate outcome is written into the Order outcome field. All possible outcomes are classified in the ***OrderOutcome*** enum as per specification section 0.1.8. The *validateOrder* method is then called by the ***OrderProcessor*** class for each order.

More about restaurants and menus

Having validated the orders, the ***OrderProcessor*** class filters the correct orders and assigns them to a restaurant to which they belong (one valid order belongs to exactly one restaurant). Each restaurant, as mentioned previously, is stored in a ***Restaurant*** class. It includes details about location of the restaurant, distance from Appleton Tower, orders made to the restaurant and the available menus, that are stored in ***Menu*** objects. ***Menu*** is a record that stores pizza name and price from the server. Since we can have multiple pizzas served in one restaurant, it makes sense to group data about each pizza into a ***Menu*** object (as per specification section 1.4) and store menus available in a restaurant in an ArrayList inside the relevant ***Restaurant*** object. The ArrayList of restaurants is then sorted according to the distance to Appleton Tower in order to maximize the number of delivered orders.

Navigator class

All Navigation related methods are grouped inside the ***Navigator*** class. There are two publicly accessible methods: *deliverOrders*, which takes the ordered ArrayList of restaurants and generates drone flightpath according to the specification, and *navigateTo* that can also be used to calculate the distance to any given restaurant before sorting. Note that the *navigateTo* method throws a custom ***MoveLimitReachedException*** if the drone cannot reach the desired endpoint due to insufficient battery capacity, which is then caught when measuring the distance to restaurants or delivering orders. The ***Navigator*** class also stores details about the drone: maximum and current remaining number of moves, drone's position, its moves, visited points and no-fly zones. This design allows simple extension of the software in case the project expands and there are multiple drones delivering the orders, as each ***Navigator*** object stores details that are relevant for just one drone. Additionally, the specification limits the number of directions the drone can take to 16, so they are all enumerated inside ***Direction*** enum together with a hover action (representing no direction with null as per specification). This allows easier looping through the possible directions when choosing next move (explained more in Part 2). Alternatively they could simply be listed in a switch (resulting in 16 cases), but the current design is more simple, concise and easily extendable if the number of direction changes in the future.

Representation of a point

The specification allows representing a point as longitude and latitude values on a 2D plane. These longitude and latitude values for one point are both stored in a ***LngLat*** record. It is the right class for storing points because there are no circumstances when only latitude or longitude is needed in the application, and storing longitude and latitude separately would bring additional complexity to the application code. The ***LngLat*** class also stores point related methods, such as *nextPosition* for calculating the next position after moving a constant distance (given in specification) in a given direction, *closeTo* to check if a point is within the distance tolerance to another point and additional checks to make sure the point is not inside or on the border of a (no-fly) area.

Visibility graph representation

A visibility graph is used to generate drone's flightpath around obstacles (explained further in Part 2). The visibility graph is stored inside a ***VisibilityGraph*** class. The shortest path algorithm described Part 2 is implemented in the ***VisibilityGraph*** class. The graph is represented using adjacency lists. This approach (and not, for example, adjacency matrix) was chosen because it provides better efficiency when listing the neighbours of each vertex – this is especially relevant as the shortest path algorithm (Dijkstra algorithm) often has to loop through neighbouring vertices. The graph edges are represented by the ***LineSegment*** class. Its purpose is to group pairs of ***LngLat*** objects together, since edges always have two endpoint ***LngLat*** objects and also offers some line-segment related functionality (for example intersection check). Compared to keeping the two endpoints separately (in two ArrayLists for example) introduces more clutter and makes the application code less readable in comparison to the current design. Additionally, the Dijkstra algorithm makes use of a priority queue for higher efficiency. However, since the priority queue entries should contain a tuple of vertex and distance and be ordered in terms of distance, a new helper private ***QueueEntry*** record was created inside the VisibilityGraph class which contains both of these fields. This class implements a Comparable interface and offers a compareTo method, that allows the ***QueueEntry*** vertex/distance pairs be ordered in terms of distance, so that the closest vertex is always the first in the priority queue for the Dijkstra algorithm.

Area/polygon representation

The obstacles (no-fly zones) in the visibility graph as stored as a ArrayList of ***Area*** records. ***Area*** object stores a list of vertices represented as ***LngLat*** objects in an ArrayList. Note that the first and the last ***LngLat*** vertices in the ArrayList are repeating. This provides 2 main benefits. Firstly, it matches the format on the Rest server (where the first and last vertices are also repeating), so the deserialization is simple. Secondly, the methods for finding adjacent vertices (*getAdjacent*) and generating the edges (*getEdges*) are slightly easier to implement, as there is no wraparound edge. A possible improvement could be that the edges should be generated immediately after creating the ***Area*** object instead of on demand, as it would slightly increase efficiency if needed.

Resulting output

The class which writes output to files is named ***Output***. The specification requires 3 output files, which are the public methods in the Output class. One output file is a json file with deliveries – it can be easily serialized from ***Order*** class by providing only the relevant getters for serialization in the ***Order*** class. Geojson file can also be serialized from a list of ***LngLat*** objects that represent the visited locations. However, for the flightpath json file, a very specific set of attributes was required. Hence, to facilitate simpler design (in terms of having attributes for one record bundled in one object) and easier serialization a new ***Move*** class was created. One ***Move*** object stores all relevant details to be written into the flightpath json file record, and a list of ***Move*** objects is then serialized into an array of json records. The serialization from a list to an array of records for delivery and flightpath json files is provided by a private ***ListSerializer*** class that is implemented inside the ***Output*** class, as its functionality is only relevant for the ***Output*** class. The ***ListSerializer*** class extends the

Jackson package ***JsonSerializer*** class and overrides its serialize method to allow serialization of any list of objects into the required format.

Part 2. *Drone control algorithm*

The main objective of the drone control algorithm is to deliver as many orders as possible for any given day while adhering to the drone movement constraints. These constraints include:

(1) Avoid no-fly zones.
(2) Choose from 16 compass directions to move each time.
(3) Not leave the central area after entering it for each order.
(4) The drone can move at most 2000 steps of length 0.00015 degrees and must finish on Appleton Tower.

The algorithm will be described using a top-down approach: the high-level idea and building upon it, finally reaching the complete working solution.

2.1. Visibility graph approach

General idea

The first requirement is to avoid no fly-zones. The objective is to do that while having as short paths as possible. According to de Berg et al. the theoretical shortest path while avoiding obstacles follows straight line segments except at the vertices of the obstacles, where it may turn, so the Euclidean shortest path is the shortest path in a visibility graph that has as its nodes the start and destination points and the vertices of the obstacles. [https://en.wikipedia.org/wiki/Visibility_graph#cite_note-robot-2]. Therefore, in our case, building a visibility graph that includes the current drone location, the desired endpoint (which is either restaurant or Appleton Tower) and the vertices of no-fly zones gives the shortest possible path from drone's position to the endpoint. This satisfies the main requirement of maximizing the number of delivered orders, as the optimal shortest paths will theoretically take the shortest distance for the drone to move while also adhering to the first requirement of avoiding no-fly zones. Notice that the (3) constraint is also satisfied, because none of the no fly-zones exist outside of the Central Area [1], and there is enough space for drone to maneuver around [2]. This implies that the shortest path also has no reason to leave the central area, as there will always be a possibility to go around any no-fly zone without leaving the central area, which satisfies the (3) constraint.

Building graph edges

Next, for each two vertices (represented by LngLat objects) in the graph the connection is made if and only if the connecting edge does not intersect any no-fly zone (in other words, it does not go inside of any no-fly zones and does not cross any corners). This check is performed in the LineSegment class doNotIntersectsAreas method. Overall, this ensures that if the drone (that for now moves in an arbitrary direction) follows any of the edges, it does not cross any no-fly zones. This satisfies the (1) constraint for now.

Choice of the shortest path algorithm

Now having populated the vertices and valid edges of the graph, we can reach the main task of the visibility graphs – to find the shortest path from the start point to the end point. The chosen algorithm is Dijkstra, as it is guaranteed to find the shortest path [3]. An alternative choice could be A* search algorithm – it can be more efficient than Dijkstra and can also be made to find the optimal solution in all cases [4][5]. The Dijkstra algorithm was chosen because it is simple, easy and quick to implement, that can easily be extended into A* search algorithm if more time efficiency is needed in the future.

Implementation of Dijkstra algorithm

The Dijkstra algorithm was implemented using priority queues for greater efficiency, as mentioned in Part 1. The algorithm starts with only the starting vertex visited with distance from starting vertex 0. All other vertices are initialized to have infinite distance from starting vertex (represented by the positive infinity value in the Double class). Then for each fringe vertex that is still in the priority queue, the closest one *v* is taken out of the queue and a loop running through each of its neighbour *u* checks if the distance to *it* from the start through *v* is less than the current distance to the *u* (which was initialised to infinity), and takes the minimum of the two, updating the priority queue entry if necessary. There is also a list initialised for storing the previous nodes of the graph in order to backtrack the path. This process is repeated until the checked priority queue entry is the desired endpoint. Then a backtracking algorithm runs backwards through the resulting list and returns a path of LngLat objects the drone has to go through to reach the desired end point.

Possible efficiency improvements

Currently, a new visibility graph is constructed, and a new path is computed for each order. This choice was done primarily due to floating point errors – they might accumulate over time and change the path the drone needs to take. This design is a trade-off between accuracy and efficiency and accuracy was prioritised in this application. However, in case more efficiency is needed, it is possible to store unique visibility graphs for each restaurant, with common no-fly zone vertices, Appleton Tower as the starting point and restaurant coordinates as the ending point.

2.2. Adapting the algorithm to the drone constraints

General idea

Having found the theoretical shortest path which maximises the number of delivered orders and satisfies the (1) and (3) constraints, the remaining task is to adapt it to the constraint set of directions the drone can take (2) and constrained number of moves the drone can use (4). The task of converting from optimal paths using straight lines to paths of the constrained length and direction is not easy. The chosen approach is to use a Greedy Best First Search algorithm to move between visibility graph vertices in the shortest path. The rough idea is that the drone should follow the theoretical straight-line shortest path around no-fly zones that satisfies (1) and (3) constraints, while also adhering to the (2) and (4) constraints around limited drone movement direction and distance.

Greedy Best First Search algorithm

The simplest way to move from point A to point B is to greedily choose a direction that minimizes the remaining distance to point B. It is possible to satisfy the (2) restriction by restricting the possible number of directions to 16. The simplest approach is to loop through the 16 directions and for each direction check if it allows for a bigger gain than the current maximum one – this way the best direction out of the 16 will be chosen that minimizes the remaining distance to B. The directions that would lead drone to cross a border of any no-fly zone are ignored to ensure the (1) constraint is still satisfied. Once the drone is within the provided distance tolerance of the end point, it can start moving to the next point in the shortest distance list.

Possible efficiency improvements

It is possible to improve this by also looking at the coordinates – for example, if the endpoint longitude is greater than the current drone longitude, do not consider the directions that will decrease the longitude (in other words, any westward direction). However, while this helps increase efficiency of the code, it reduces simplicity. For example, the directions should be logically grouped together beforehand (for example, group all westward and eastern directions into two groups, etc.) and then additional checks have to be made to choose the appropriate group. The groups can be made even smaller, but the code complexity would increase further. The current implementation was chosen because of its simplicity, readability, and extensibility (which will be shown later). It is also possible to backtrack the moves when going back to the Appleton Tower instead of greedily computing them each time. While these changes would allow the program to have better efficiency, additional care must be taken to ensure that the order reaches the desired endpoint as per specification.

Issue with the approach and implemented fix

Since the drone has constraints on its movement direction (2) and distance (4), it cannot follow the theoretical path perfectly. Sometimes this deviation can be large (see Figure 1). This can lead the drone to deviate enough, so that it encounters unexpected new no-fly zones and gets stuck behind them. The direction choice algorithm was changed to address this issue:

1. Drone greedily chooses the best direction (minimizing the remaining distance) that allows the drone to see the next path point without any no-fly zones in the way.
2. If this is not possible choose the best direction (minimizing the remaining distance).

The second choice may be required when the straight-path line is between two no-fly zones with a small margin. Then the drone moves for some time without seeing the next endpoint but chooses the visible direction as it navigates close to the second no-fly zone.

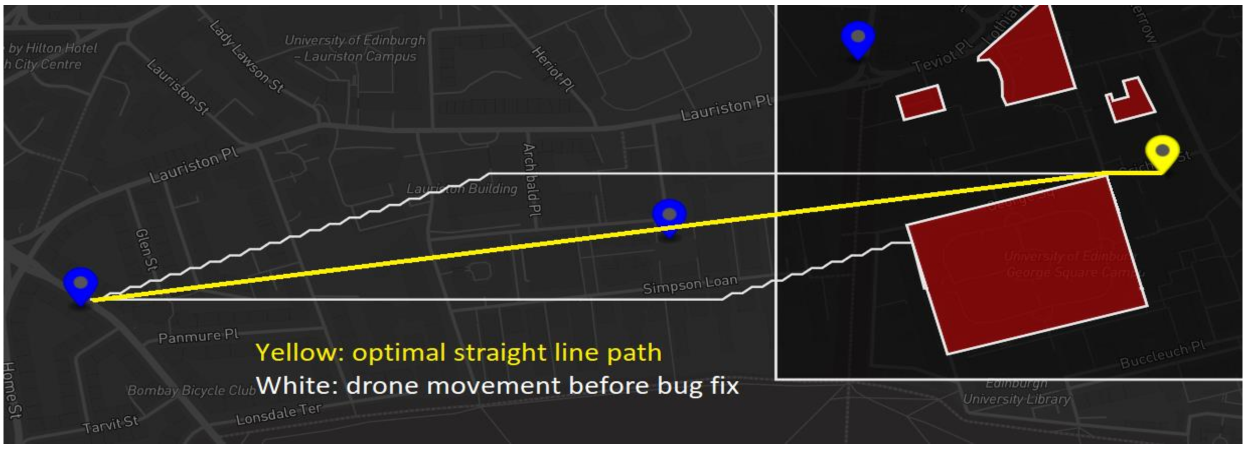
Yellow: optimal straight line path
White: drone movement before bug fix

Figure 1: Drone movement issue due to different straight-line and actual flightpaths before it was fixed.

Another possible failure and fix

A possible failure can happen if some specific circumstances align: the drone chooses the last move closest to the endpoint and chooses the shortest path, which could theoretically be behind a no-fly zone if an angle is awkward and the distance to the next path point is just above the distance tolerance. Then the drone tries to continue going to the next point behind the no-fly zone while trying to minimize the distance, but cannot because then it would fly into this no-fly zone. While no example cases where identified when this causes a problem, it is possible to fix this issue in future improvements by choosing the last step so that the next shortest path point is visible as well.

Writing results

Each time a move is made, the location and the move details for flightpath json file are saved in temporary lists named visitedSinceAT and movesSinceAT respectively. The values stored in these temporary lists are moved to the final result lists visited and moves respectively only if the drone can finish delivering the order back to Appleton Tower. This makes sure that the final results do not have moves that make drone run out of battery when it is out for delivery. The final results are illustrated in Figure 2 and Figure 3.

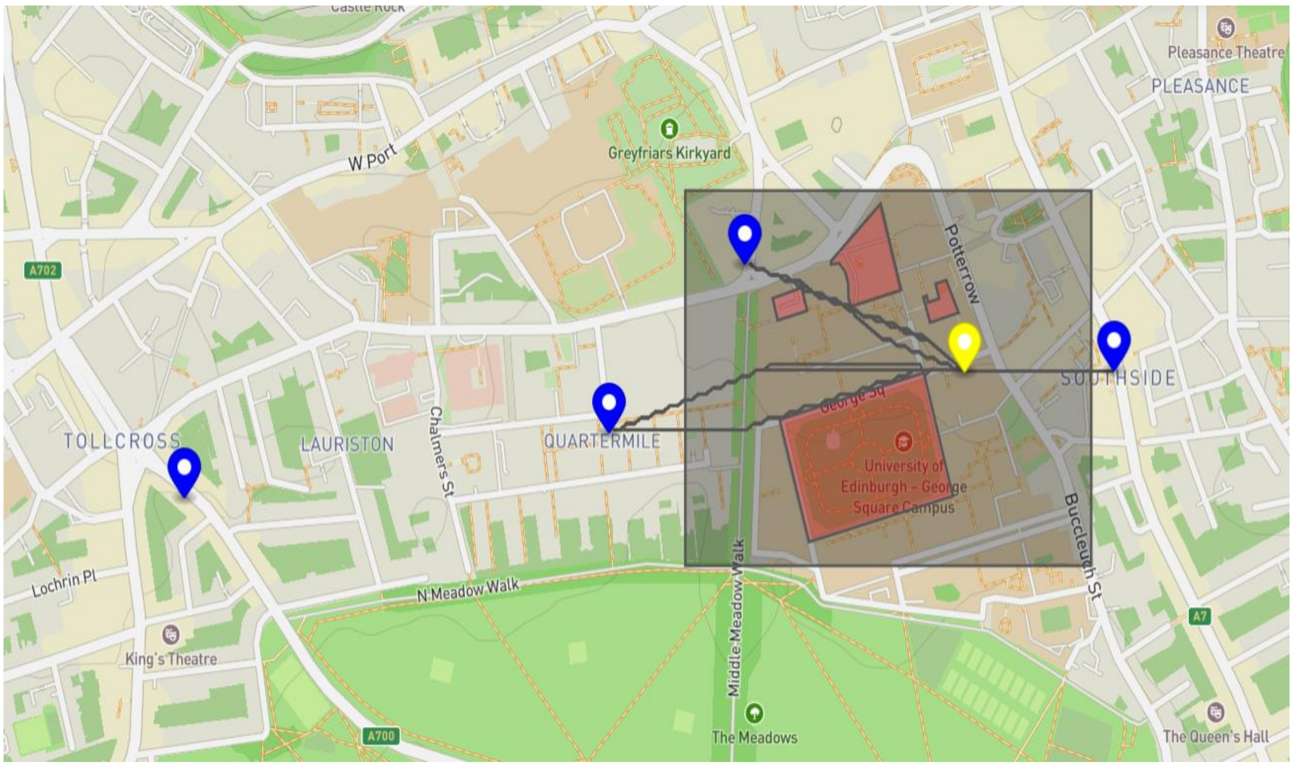

Figure 2: Drone movement in Central Edinburgh

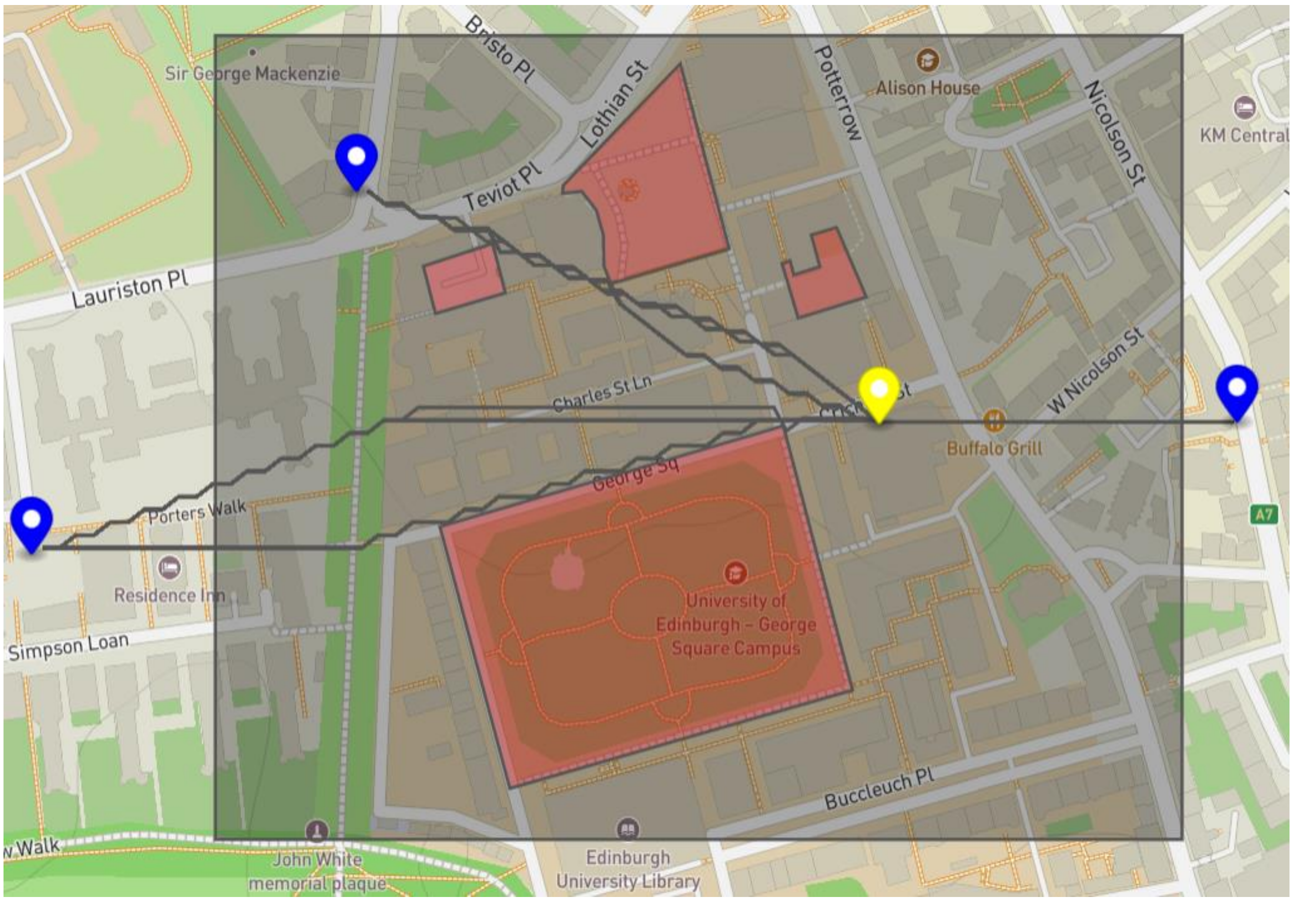

Figure 3: Delivered orders by drone

References

[1] Piazza question @168. Retrieved 1st December 2022. https://piazza.com/class/l82xsgm1u243sc/post/168

[2] Piazza question @205. Retrieved 1st December 2022. https://piazza.com/class/l82xsgm1u243sc/post/205

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 24.3: Dijkstra's algorithm, pp.595–601.

[4] František Duchoň, Andrej Babinec, Martin Kajan, Peter Beňo, Martin Florek, Tomáš Fico, Ladislav Jurišica. Path Planning with Modified a Star Algorithm for a Mobile Robot, Procedia Engineering, Vol. 96, 2014, Pages 59-69, ISSN 1877-7058. Retrieved 1st December 2022. https://www.sciencedirect.com/science/article/pii/S187770581403149X

[5] Zhanying Zhang and Ziping Zhao. A Multiple Mobile Robots Path planning Algorithm Based on A-star and Dijkstra Algorithm. Vol.8, No.3, 2014, pp.75-86. Retrieved 1st December 2022. https://gvpress.com/journals/IJSH/vol8_no3/7.pdf