JAYDEN YAP (2112790) NATHAN YEO (2123082)

DAAA/2B/04

Abstract

Pizza Runners is a startup company that has revolutionized the food delivery industry by incorporating drone technology for door-step pizza delivery. However, recent customer complaints regarding late deliveries and failed deliveries prompted an investigation by the company's IT department. The investigation revealed that the left-hand algorithm used for drone navigation was at fault, with shortcomings such as inability to find a path between two points and not always taking the shortest path. In light of these findings, this report will outline the steps taken to address these issues and improve the overall delivery process. This report will detail the revised approach taken to enhance the drone navigation system, ensuring accurate and efficient deliveries to meet customer expectations.

GROUP 9: REPORT

DSAA CA2 PIZZA RUNNERS PROJECT

**1.0 USER GUIDELINES**

**1.1 City Map Files**

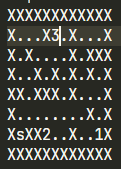
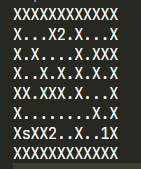
Please place any map files in the ‘maps’ folder. The map can have **multiple destinations** with multiple usage scenarios.

* General requirements:
  + Max supported size: 140 columns x 70 rows
  + ‘X’ indicates a building/unavailable square
  + ‘.’ indicates a available square
  + ‘s’ indicates the starting square for the drone
    - If more than 1 s is detected, the most bottom-right square will be used
  + ‘e’ indicates a destination
  + A number from 1-9 will indicate a destination using priority

Map examples:

Text

Description automatically generated**Multiple ‘e’ letters:** Shortest-path finder will go through in order of top to bottom, then left to right.

* **Priority destinations:** Use numbers from (1-9) instead of ‘e’ to designate the order of destinations for the shortest-path finder to follow.
* Example: Dijkstra will go to destination 1, then 2, then 3
* If there are multiple destinations under one number, it will follow the rules stated in a. but for those specific destinations only.
* Example: Dijkstra will go to destination 1, then the top-most destination then the bottom one.

**1.2 Loading the Program:**

Text

Description automatically generatedYou can Load the program by opening Anaconda Prompt and going to the project folder, from there just type “python main.py (and a map of your choice e.g. map1.txt)”.

A picture containing graphical user interface

Description automatically generated

This will bring up a pop-up turtle graphics window where the program will start, depending on the size of the map it will take a few seconds to load. (Note: city map .txt files have to be in maps folder to be used)

**1.3 Program Usage**

* When loading the program, the default Algorithm is the Left-Hand Algorithm as denoted by the title of the pop-up window at the top left, it will also begin in a paused state



* To switch to other algorithms, you can use the “Tab” key. We have a total of 4 modes for you to choose from and it goes in this order as a cycle:
  + Text

    Description automatically generated Left-Hand Rule Algorithm
  + Text

    Description automatically generated Dijkstra Algorithm (shortest path)
  + Text

    Description automatically generated with medium confidence Manual User Input

Text

Description automatically generated with low confidence

* + Random Movement (Note: Left-Hand algorithm will switch to Random Movement automatically to unstuck itself)
* When you have selected the mode which you want, you can press “Space” key to start running the algorithm.
* Graphical user interface

  Description automatically generatedFor Manual mode, you can just press the arrow keys to move the turtle in the corresponding directions.

A picture containing graphical user interface

Description automatically generated

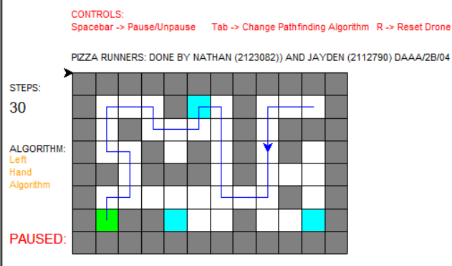
* Pressing “Space” at any time during the running of the algorithms will Pause or Un pause if it is already paused. Note: The title shows whether the algorithm is paused and there is a display when paused too

Graphical user interface

Description automatically generated

* Pressing “Tab” at any time will change the modes as earlier mentioned. After changing the modes, the program will automatically pause. So, remember to unpause with “Space” or press “Tab” to continue switching/cycling between modes
* If you wish to reset the pathfinder at any time, press “R” to reset the drone to its start position

A picture containing text, wall, keyboard, square

Description automatically generated**Before:** **After:** Table

Description automatically generated with medium confidence

**Diagram

Description automatically generated2.0 OBJECT ORIENTED PROGRAMMING (OOP) OVERVIEW:**

**2.1 General OOP Practices**

We practice Encapsulation by doing the following:

* By using classes as blueprints for object instances
* Making use of privated variables like \_\_windowSize and private functions like \_\_reset.
  1. Enhanced data security (users cannot as easily access those variables and functions, only the classes themselves can assess them through its own methods)

**2.2 Main Program Class**

Main Program class sets up the turtle graphics window and handles all the individual turtle pens. It also handles the switching of pathfinding algorithms, pausing, resetting the program, as well as the Tkinter pathfinding speed widget.

main () function:

* Verify that there is a path
* Read map file, then draw out the map on turtle, set up the Tkinter slider widget used for the dyanmic pen speed
* Then In a loop:
  1. We use the findPath() function of the different Pathfinder classes which will run until either
     1. the user pauses
     2. user wants to switch algorithm
     3. user wants to reset program
     4. all destinations were reached
  2. Then we call the handleKeypress function which checks for a)i,a)ii,a)iii and does actions accordingly
     1. Continue and let the while loop run until the user unpauses
     2. Switch the self.pathfinder to next pathfinder in the cycle
     3. Reset pathfinder to original position
  3. Then check if a) iv was reached (i.e all destinations reached), if so, flash a message, this will only flash once
  4. Repeat loop

**2.3 Pathfinder Class (parent class)**

The pathfinder class is a parent class which provides a basic framework for 4 algorithms (Left Hand, Dijkstra, Manual and Random) and handles the visual display of the navigation process. The reason we have this is to reduce redundancy by using **Inheritance**. The class has several attributes, including the pathfinder’s name, the map, the number of steps taken, the current position, the list of destinations and a verifier for the Dijkstra Algorithm. It also has several turtle objects for visualizing the navigation process, including a turtle for the actual path, one for displaying the number of steps taken, and another for indicating whether the navigation is in a paused state.

The 4 Pathfinder classes/algorithms will be further discussed in Section 3.0. They all inherit from the Pathfinder parent class and share a drawPath function but with different implementations. This is a form of **Polymorphism** as the drawPath function from the parent acts as the interface while the different Pathfinder child classes have their own implementation.

**2.4 Map Class**

Contains functions for reading .txt files to initialise the grid as a 3D list, representing the city map. Functions for calculating the size and appropriate scale to be used for drawing the grid based on the number of columns and rows (dynamic scaling)

The readFile function takes in a .txt file and extracts variables such as the start and destinations and stores the destinations in a Sorted Linked List.

The drawMap function draws out the map using the turtle pen passed by main program.

**2.5 Sorted Linked List**

The Sorted Linked List is used for the **Priority Destinations** system. It sorts the destinations by the numbers given by the user in the text file. We utilise **operator overloading** to define custom behaviour for in-built methods like \_\_str\_\_ using magic methods. The data structure will be further discussed in section 4.0.

**3.0 PATHFINDING ALGORITHMS**

**3.1 Left Hand Algorithm**

The left-hand rule algorithm is a simple method for finding an exit from a maze. It involves holding

one's left hand on the wall and always turning left at intersections until the exit is found. In case

the algorithm realises its stuck in a loop, it will automatically switch to a random algorithm to get

itself unstuck. This ensures that if there is a valid path, the left-hand algorithm will always find it

eventually. Other methods of improving the algorithm might not follow the idea of the left

hand rule being used by blind people to find their way out of a maze. The use case for this

in our case of the pizza delivery drone is that this algorithm could be used as a backup

in case the drone's sensors fail. For example, its visions sensors might fail so in that case,

it can revert to using the left-hand algorithm to always get it to where it needs to go eventually.

Algorithm:

1. Check the square to the left, if available, go there and append to path list, if not continue

2. Check the square in front, if available, go there and append to path list, if not, then turn to the right

3. Repeat. If at a square that has been in the path list more than twice, change the algorithm to random

algorithm for a random number of steps (between 8 and 20) to get itself unstuck. After switching back to left

hand algorithm, refresh the path list again.

The Big O Notation for the left-hand rule is O(N) where N is the number of available squares. This is linear time complexity. However, the real performance largely depends on the type of maze given, there will be times when it needs to backtrack, others where it gets stuck in a loop.

**3.2 Dijkstra Algorithm**

Dijkstra's algorithm is a method for finding the shortest path in a graph, where the distances between nodes are represented as weights. The algorithm works by starting from the source node, then repeatedly selecting the node with the lowest distance and updating the distances of its neighbours until the shortest path has been determined for all nodes.

We also utilised **Function Overloading** by specifying default arguments for ‘verifier’. When its false which is default, the pathfinder acts as normal. When it is a verifier, it is only used to check if there is a valid path available.

To calculate the shortest path between two points using Dijkstra's algorithm. We create a graph using the NetworkX library, adds nodes and edges to the graph, and finally returns the shortest path.

1. The distance of the source node is set to zero and all other nodes are given an infinite distance.
2. A priority queue is created to store the nodes that need to be processed, with the source node being the first to be added.
3. The algorithm continues until the priority queue is empty:

1) The node with the smallest distance is taken from the priority queue.

2) The distances of its neighbours are then analysed.

A) The tentative distance to each neighbour is calculated.

B) If the tentative distance is less than the current distance of the neighbour, its distance is updated, and it is added to the priority queue.

1. The final distance values found by the algorithm represent the shortest distances from the source node to all other nodes in the graph.

The big O notation for the Dijkstra algorithm is O (E + V log V) where E is the number of edges in the graph and V is the number of vertices in the grid, it is linear but with a logarithmic factor.

Compared to an algorithm like Breadth-first search with a Big O of O (V + E) which is also linear but with no logarithmic factor, it is going to perform better.

**3.3 Manual Input**

The main purpose of the manual input is to let the user freely explore the possibilities. For example, instead of trying to start only at the start point, the user can move it to another location and change the algorithm to see how the other pathfinders react to different scenarios.

Algorithm:

1. Listens for input from keyboard

2. Checks whether the path is valid

3. The drone will then move to the specified direction if the path is valid, if not it won’t move

4. Listens for input again

Manually moving a drone is not a mathematical function, so it is not appropriate to describe its performance using big O notation. Instead, the time taken to complete the order is based on the user.

**3.4 Random Algorithm**

The main purpose of the random pathfinder is to improve the left-hand algorithm, but it is also a usable pathfinder like the others. When the left-hand algorithm is stuck in a loop, it will automatically switch to this random algorithm in the hopes of getting itself unstuck/finding the destination.

Algorithm:

1. Calculates which of the 4 possible directions are valid (in bounds and not a building)

2. Appends the direction to a list twice, if the last direction moved by the algorithm was the opposite, append once

a. This makes the algorithm disfavour going backwards compared to the other directions, hence higher chance of making progress

3. Randomly sample a direction from the list

4. Move in that sampled direction

5. Repeat

The big O Notation for the random algorithm is O(N) where N is the number of squares, but this should be taken with a grain of salt because the random algorithm is in most cases going to be the worst one. It can sometimes go in loops and take an arbitrarily long time to find the destinations.

**4.0 DATA STRUCTURES**

|  |  |  |  |
| --- | --- | --- | --- |
| Data Structure | Description | Insertion O | Deletion O |
| List | An ordered data structure that stores a collection of items, where each item can be of any data type | O (n) | O (n) |
| Sorted Linked List | Each item, called a node, contains a data element and a reference to the next node in the list, these nodes are linked in such a way that the data elements are in ascending order. | O (n) | O (n) |
| NetworkX Graph | A graph in NetworkX is a data structure that represents a set of nodes and edges connecting these nodes. | O (1) | O (1) |

**5.0 GROUP REFLECTION**

**5.1 Challenges faced**

One of the challenges that we faced was the lack of time to develop our program, the program can still be developed in many ways such as more User friendly interfaces, better code efficiency and more features that we wish to apply.

Another problem that we faced was how to do our work on separate devices. We encountered this problem early on and found a way to get around it, we simply uploaded the files on MS Teams and with a little cohesion, it was not a problem anymore.

**5.2 Takeaways**

**Nathan:** I learnt how different algorithms work, like Left-Hand Algorithms and Dijkstra Algorithm and how to incorporate them into a program. I also learnt how to work alongside teammates online.

**Jayden**: I learned how to apply OOP practices to a larger project as well as how to cooperate with teammates to implement many features that contribute to a large program with many different parts.

**5.3 Roles and contributions**

**Nathan:** Improved the left-hand rule by incorporating the random moving algorithm to make it unstuck. Implemented reading in the map and the variables. Implemented the switching of pathfinders and the pause key.

**Jayden:** Implemented left-hand-rule algorithm. Implemented Dijkstra (shortest-path) algorithm. Implemented the Sorted Linked list class. Implemented Reset program key. Created OOP diagram. Created labels for Step Count and Algorithm used. Implemented the draw map function to draw the map on turtle.

**APPENDIX: SOURCE CODE**

|  |  |
| --- | --- |
| **Source Code File** | **Written by** |
| main.py | Jayden Yap |
| logic.py | Jayden Yap, Nathan Yeo |
| map.py | Jayden Yap, Nathan Yeo |
| path\_finder.py | Jayden Yap, Nathan Yeo |
| pathfinder\_leftHand.py | Jayden Yap, Nathan Yeo |
| pathfinder\_manual.py | Nathan Yeo |
| pathfinder\_random.py | Nathan Yeo |
| pathfinder\_shortest.py | Jayden Yap |
| Sorted\_linked\_list.py | Jayden Yap |