Question 1.1

The maze solver can be seen as a search problem as it contains a search state, represented by the ‘-’, a goal state, the ‘-’ in the bottom line, and a start state, the ‘-’ on the topmost line. It’s a search problem as the search state includes multiple paths and cycles meaning some kind of algorithm must be undertaken to successfully navigate from the start state to the goal state.

Question 1.2

1. The depth-first algorithm is an algorithm used to navigate or search tree or graph data structures. The algorithm starts at a certain root node (for the maze solver this would be the start state) and explores the graph or tree by going as far down a branch as possible. Only backtracking if it encounters a dead end, at which point it backtracks to the nearest junction which leads to unexplored nodes. The depth-first algorithm finishes once it reaches the goal node or every explorable node has been searched. It is important to note that depth first search only works in graphs with cycles if it keeps track of explored nodes. If explored nodes are not taken into consideration depth first will find itself stuck in cycles going around in circles.

|  |  |  |  |
| --- | --- | --- | --- |
| Maze | Time to solve (s) | Total explored nodes | Final path length |
| Easy | 0.00015 | 50 | 41 |
| Medium | 0.01819 | 5740 | 625 |
| Large | 0.03963 | 14685 | 1142 |
| Very Large | 0.25731 | 88663 | 4053 |

4.

Table 1: Measured performance statistics for the Depth-First algorithm on varying mazes

Table 1 shows the results that Depth first achieved for each maze size. There is a dramatic increase in time needed to solve the mazes as they increase in size but this increase is proportional to the increase in total explored nodes. Overall Depth-First solves these mazes rapidly solving a maze of 2’000’000 nodes in 0.25 seconds. It should be noted that despite solving the largest maze in 0.25 seconds it only explored 88’663 nodes from the total 2’000’000. The overall speed of the algorithm might be accredited to the order in which depth first looks at neighbouring nodes for a certain node. This is because Depth-First always picks the last found valid neighbour from the stack, hence depending on the order that depth first finds valid neighbours the last valid neighbour will have a higher probability of being in a certain direction towards the current node. Looking at line 25 of depthFirstSearch.py:

for direction in [(-1,0),(0,1),(0,-1),(1,0)]:

We can see that the last valid neighbour to be added would be the node at the position -1, 0) relative to the current node. This represents the node 1 line below the current node. If the node in that position is not valid the next most likely node to be found is at the position (0, -1). This represents the node 1 column to the left of the current node. This is quite advantageous as we know for certain that the starting node is in the top left of the maze, and the end node is in the bottom line. Overall this small code decision influences the speed at which depth first can solve the mazes.

Comparing the above order with the following:

for direction in [(1, 0), (0, -1), (0, 1), (-1, 0)]:

Inversing the order in which neighbours are added to the stack made the time to solve jump to 2 seconds for the largest maze, exploring 794’393 nodes from a total of 2’000’000. Almost ten times more than with the previous order. This shows that variations in the location of the start and end node can massively influence the time-efficiency of this algorithm.

Finally, it should be noted that Depth-First does not find the shortest path and so cannot be relied on for applications requiring the shortest path.

Question 1.3

1. I will be implementing A\* as the second algorithm. A\* is a pathfinding algorithm like Dijkstra’s algorithm, in the sense it uses a heuristic to choose which node to look at next. A\* is also guaranteed to find the shortest path in a maze unlike Depth-First. I would also expect A\* to be faster due to the heuristic value guiding it making smarter choices when picking a node.

A\* works by looking at valid neighbours to the node its currently on. These neighbours are given two heuristic values, the first being the same as Dijkstra’s algorithm, the distance between the starting node and the neighbour node. This can be easily found by keeping track of the number of nodes explored before reaching a specific one. The second value is the distance between the neighbour node and the end node. Multiple methods can be used for this, this includes calculating the Euclidean distance by comparing their coordinates or their Manhattan distance by comparing their coordinates again. I opted for Manhattan distance as the overheads required to calculate Euclidean distance such as finding the square root outweighed having a more accurate distance.

These two heuristic values are then added together and in the following iteration of the algorithm the node with the lowest combined heuristic is picked. This represents the next unexplored node in the most direct line between the start and end node.

To make the node picking process time efficient I opted to use a priority queue in order find the desired node in O(log(n)) rather than O(n) for an array.

|  |  |  |  |
| --- | --- | --- | --- |
| Maze | Time to solve (s) | Total explored nodes | Final path length |
| Easy | 0.00070 | 57 | 27 |
| Medium | 0.02426 | 2060 | 321 |
| Large | 0.46515 | 42006 | 974 |
| Very Large | 3.24538 | 274137 | 3691 |

3.

Table 2: Measured performance statistics for the A\* algorithm on varying mazes

Table 2 shows the results that A\* achieved for each maze size.

Figure 1: Comparing time to solve to number of explored nodes by the A\* algorithm

Figure 1 shows the correlation between time to solve each maze and the number of nodes explored for those mazes. As we can see the line of best fit represents a non-linear increase, most likely representative of the O(Log(n)) cost when choosing the next node. This small overhead allows for the algorithm to consider far fewer nodes than other algorithms such as breadth-first or Dijkstra’s, resulting in an overall quicker program.

It's worth noting that due to a maze’s nature A\*’s heuristic struggles to be effective. As the quickest path through a maze is rarely a direct line between the start and end node. In a more open setting or a graph A\* would perform a lot better, as each wall between the start and end node is a massive obstacle for A\*’s heuristic. The walls make A\*’s heuristic perform very similar to Dijkstra’s algorithm as the Manhattan distance from any certain neighbour node the end node will undoubtedly be very inaccurate from the actual distance between those two nodes.

4. Comparing Table 1 with Table 2 we see that Depth-First was far quicker at solving the mazes than A\*. Although A\* did find a shorter path for every maze than Depth-First could. It’s important to note that for a random selection of mazes, where the start and end node are not in the top left and bottom line respectively both algorithms would perform at similar speeds, with depth-first being slightly faster due to no overheads when choosing the next node. It would be interesting to see if modifications to the heuristic A\* uses would increase the speed at which it solves the mazes, and possibly be faster than Depth-first. This would come down to how many cycles the mazes have, as in the event they had little to no cycles, any heuristic used would become nearly useless.

Overall mazes make very interesting search problems due to obstacles being almost everywhere in the search space, hindering the effectiveness of any heuristic used. This is shown by looking at the number of explored nodes by both A\* and Depth-First. For the very large maze depth first looked at 88’663 nodes compared to A\*’s 274’137 nodes. This highlights that trying to beeline directly towards the end goal made the algorithm look at over three times more nodes as it was forced to increase its search space due to obstacles completely blocking off possible paths.

In conclusion both algorithms have significant strengths and drawbacks, notably Depth-First’s speed compared to A\* finding the shortest path. But the effectiveness of both algorithms would heavily depend on the maze structure, with the presence of cycles, longer or shorter pathways, the location of the start and end node all having huge consequences on the effectiveness of both algorithms.