Question 1.1:

The maze solver could be framed as a search problem, and solved by searching the graph representation of the maze for a goal node (the exit node). To do this we take all positions where there is a path (represented by a "-") and create nodes to represent them, which we add to a graph. We can then find the neighbours for each node by checking for paths directly above, below, and to either side of the node, and by adding edges between the two nodes in the graph. We can then specify the entrance to the maze as the start node and specify all exits from the maze as goal states. This will allow a search algorithm such depth-first search or A* search to traverse the graph and find a path from the start node to any of the goal nodes (in our case there is just one exit so just one goal node) and therefore identify a path via which we could travel through the maze.

Question 1.2:

Note: In order to be able to run the Maze_DFS.py program, python3 must be installed and you may also need to install the timeit module. Additionally the program must be executed from terminal and not an IDE as IDE's may impose their own max recursion limits so the program may not execute to required recursion depth and hence not find a solution. The program will ask the user for the input name of the maze file to use and will solve the maze represented in the given txt file. The maze-file option that must be inputted for each answer is specified in the answer.

Part 1:

The depth-first search algorithm is a search algorithm that finds a path between the start node and goal node. It is important to note that the path found is not necessarily the optimal path between the two nodes. In order to carry out the depth-first search algorithm, we require each node to have a visited attribute to determine whether or not each node has been visited yet.

The depth-first search starts by setting the start node as visited and checking whether it is the goal node. If it is then it will return itself as a path. Otherwise the top (unvisited) neighbour is popped from the neighbours stack (note the order of this stack is dependent on the order in which the graph was created from the maze file). The algorithm then recursively calls depth-first search on the popped neighbour. If the recursion returns a non-empty path, then the depth-first search will return the path with itself appended to the front (as this signifies the goal node was found further along in the recursive calls). If the recursion returns an empty path (this signifies the goal node was not found further along in the recursive calls), the next node is popped from the stack and similarly processed. Once the neighbours stack is empty, the algorithm will return an empty path (since the goal node was not found further along in the recursive calls).

Part 2:

Maze Name: maze-Easy.txt
Path Found:
Node(0, 1)→Node(1, 1)→Node(1, 2)→Node(1, 3)→Node(1, 4)→Node(1, 5)→
→Node(2, 5)→Node(3, 5)→Node(4, 5)→Node(5, 5)→Node(5, 6)→Node(5, 7)→
→Node(5, 8)→Node(6, 8)→Node(6, 9)→Node(6, 10)→Node(6, 11)→Node(6, 12)→
→Node(6, 13)→Node(6, 14)→Node(6, 15)→Node(6, 16)→Node(6, 17)→Node(7, 17)→
→Node(8, 17)→Node(8, 18)→Node (9, 18)

Part 3:

Easy Maze Analysis:

Maze Name: maze-Easy.txt
Analysis:

Run Number	Nodes Explored	Time of execution/s	Path Length
1	82	0.000234	27
2		0.000261	
3		0.000264	
4		0.00023	
5		0.000242	
Average	82	0.0002462	27

Part 4:

Medium Maze Analysis:

Maze Name: maze-Medium.txt
Analysis:

Run Number	Nodes Explored	Time of execution/s	Path Length
1	6,876	0.020407	509
2		0.020654	
3		0.020194	
4		0.024046	
5		0.021215	
Average	6,876	0.0213032	509

Large Maze Analysis:

Maze Name: maze-Large.txt

Analysis:

Run Number	Nodes Explored	Time of execution/s	Path Length
1	13,580	0.196723	1,120
2		0.228194	
3		0.19329	
4		0.193977	
5		0.232639	
Average	13,580	0.2089646	1,120

VLarge Maze Analysis:

Maze Name: maze-VLarge.txt

Analysis:

Run Number	Nodes Explored	Time of execution/s	Path Length
1	620,707	3.132061	5,725
2		3.024883	
3		3.140343	
4		3.087107	
5		3.107731	
Average	620,707	3.098425	5,725

Question 1.3:

Note: In order to be able to run the Maze_A_Star_Search.py program, python3 must be installed and you may also need to install the math, timeit and queue modules. Additionally the program must be executed from command line and not an IDE as IDE's impose their own max recursion limits so the program may not execute to required recursion depth and not find a solution. The program will ask the user for the input name of the maze file to use and will solve the maze represented in the given txt file as well as a choice between euclidean and manhattan for the heuristic to be used.

Part 1:

The A* search algorithm is a search algorithm that finds a path between the start node and goal node. In order to carry out the A* search algorithm, we require a set of three values (***g***, ***h*** and ***f***). The ***g value*** is the cost of getting to a node and is calculated by finding the number of preceding nodes in the path between the start node and the current node. The ***h value*** is a heuristic and is an estimate for the cost to travel between the current node and the end node. This can be done using a series of methods such as euclidean or manhattan distance which I will describe later. The ***f value*** is calculated as the sum of the ***g value*** and the ***h value*** for a given node and is used to decide which node should be explored next.

The A* search algorithm starts by setting the start node as visited and checking whether it is the goal node. If it is then it will return itself as a path. Otherwise the algorithm will consider all the current node's neighbours and calculate a ***g, h and f value*** for them. It will then select the neighbouring node with the lowest ***f value***. This can be implemented by using a priority queue where smaller values have higher priorities. The algorithm is then called on the popped node until a path to the end goal is returned, in which case it will append itself to the start of the returned path and return this new extended path. However if no path is returned then the current node will return no path.

One key thing to consider in an A* search is the choice of heuristic (the ***h value***). In order for the A* algorithm to return an optimal path, the heuristic chosen must be admissible meaning that the generated cost is always and underestimate of the actual cost to travel to the goal state.

I have therefore decided to compare the performances of two different heuristics (Euclidean and Manhattan) to try and observe how the differences in heuristic choice affect the performance of the algorithm.

The Manhattan distance is the sum of the vertical distance and horizontal distance between the current node and the goal node. This is an admissible heuristic since it assumes there are no walls blocking the path between the current and goal nodes. This makes the estimate an underestimate of the cost to reach the goal node.

The Euclidean distance is the diagonal distance between the current and goal nodes. This is also an admissible heuristic since it assumes there are no walks blocking the path between the current and goal nodes as well as that we are able to move diagonally, which is an underestimate of the cost to reach the goal node.

In my implementation, I have also included an additional step in the decision of which neighbour should be selected in the case of a draw in the ***f value***, by selecting the ***h value*** as a secondary key since it provided generally better results.

Part 2:

Easy Maze:

Maze Name: maze-Easy.txt

Path Found Using Manhattan Distance:

Node(0, 1)→Node(1, 1)→Node(1, 2)→Node(1, 3)→Node(1, 4)→
→Node(1, 5)→Node(1, 6)→Node(1, 7)→Node(1, 8)→Node(1, 9)→
→Node(1, 10)→Node(1, 11)→Node(1, 12)→Node(1, 13)→Node(1, 14)→
→Node(1, 15)→Node(2, 15)→Node(3, 15)→Node(4, 15)→Node(5, 15)→
→Node(6, 15)→Node(6, 16)→Node(6, 17)→Node(7, 17)→Node(8, 17)→
→Node(8, 18)→Node(9, 18)

Path Found Using Euclidean Distance:

Node(0, 1)→Node(1, 1)→Node(1, 2)→Node(1, 3)→Node(1, 4)→
→Node(1, 5)→Node(1, 6)→Node(1, 7)→Node(1, 8)→Node(1, 9)→
→Node(1, 10)→Node(1, 11)→Node(1, 12)→Node(1, 13)→Node(1, 14)→
→Node(1, 15)→Node(2, 15)→Node(3, 15)→Node(4, 15)→Node(5, 15)→
→Node(6, 15)→Node(6, 16)→Node(6, 17)→Node(7, 17)→Node(8, 17)→
→Node(8, 18)→Node(9, 18)

Medium Maze:

Maze Name: maze-Medium.txt

Path Found Using Manhattan Distance:

Node(0, 1)→Node(1, 1)→Node(1, 2)→Node(1, 3)→Node(1, 4)→…→Node(97, 198)→Node(98, 198)→Node(99, 198)

Path Found Using Euclidean Distance:

Node(0, 1)→Node(1, 1)→Node(1, 2)→Node(1, 3)→…→Node(97, 198)→Node(98, 198)→Node(99, 198)

Large Maze:

Maze Name: maze-Large.txt

Path Found Using Manhattan Distance:

Node(0, 1)→Node(1, 1)→Node(2, 1)→…→Node(597, 59)→Node(598, 59)→Node(599, 59)

Path Found Using Euclidean Distance:

Node(0, 1)→Node(1, 1)→Node(2, 1)→…→Node(597, 59)→Node(598, 59)→Node(599, 59)

VLarge Maze:

Maze Name: maze-VLarge.txt

Path Found Using Manhattan Distance:

Node(0, 1)→Node(1, 1)→Node(2, 1)→…→Node(997, 1880)→Node(998, 1880)→Node(999, 1880)

Path Found Using Euclidean Distance:

Node(0, 1)→Node(1, 1)→Node(2, 1)→…→Node(997, 1880)→Node(998, 1880)→Node(999, 1880)

Part 3:

In this section I will be analysing the performance of the A* search algorithm as well as the performance of the two different heuristics when applied to the same mazes.

Easy Maze Analysis:

Maze Name: maze-Easy.txt

Manhattan Distance Analysis:

Run Number	Nodes Explored	Time of execution/s	Path Length
1	35	0.000809	27
2		0.000839	
3		0.000902	
4		0.000782	
5		0.000841	
Average	35	0.0008346	27

Euclidean Distance Analysis:

Run Number	Nodes Explored	Time of execution/s	Path Length
1	60	0.000806	27
2		0.000772	
3		0.000799	
4		0.000815	
5		0.000802	
Average	60	0.0007988	27

Medium Maze Analysis:

Maze Name: maze-Medium.txt

Manhattan Distance Analysis:

Run Number	Nodes Explored	Time of execution/s	Path Length
1	1935	0.039655	321
2		0.046973	
3		0.03959	
4		0.050145	
5		0.041045	
Average	1935	0.0434816	321

Euclidean Distance Analysis:

Run Number	Nodes Explored	Time of execution/s	Path Length
1	3880	0.056156	321
2		0.05617	
3		0.055385	
4		0.055577	
5		0.054685	
Average	3880	0.0555946	321

Large Maze Analysis:

Maze Name: maze-Large.txt

Manhattan Distance Analysis:

Run Number	Nodes Explored	Time of execution/s	Path Length
1	41381	1.144351	974
2		1.287576	
3		1.245075	
4		1.150871	
5		1.224067	
Average	41381	1.210388	974

Euclidean Distance Analysis:

Run Number	Nodes Explored	Time of execution/s	Path Length
1	63547	2.124916	974
2		2.254605	
3		2.107483	
4		2.094149	
5		2.083816	
Average	63547	2.1329938	974

VLarge Maze Analysis:

Maze Name: maze-VLarge.txt

Manhattan Distance Analysis:

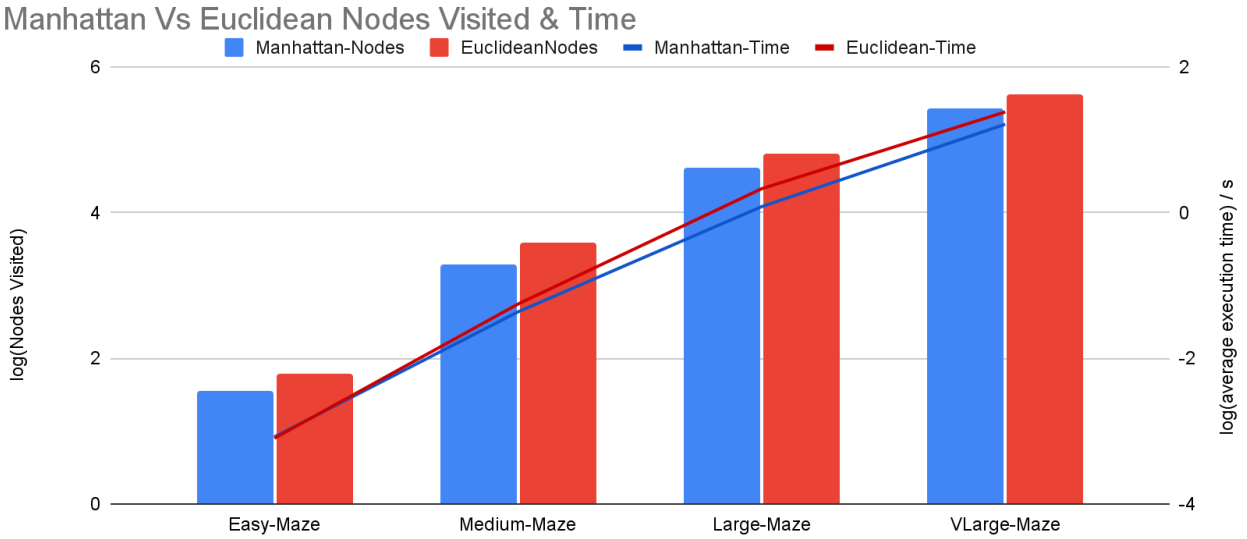
Run Number	Nodes Explored	Time of execution/s	Path Length
1	273000	15.838001	3691
2		15.988876	
3		16.599291	
4		16.475513	
5		16.971618	
Average	273000	16.3746598	3691

Euclidean Distance Analysis:

Run Number	Nodes Explored	Time of execution/s	Path Length
1	413129	25.718841	3691
2		23.834774	
3		22.300462	
4		26.297485	
5		22.781216	
Average	413129	24.1865556	3691

Written Analysis:

From the data collected, we can show that both Euclidean and Manhattan distance heuristics were able to identify optimal solutions for the mazes, since for all the mazes they produced paths of the same length (these might not necessarily be the same path but have the same number of nodes). This is expected given that both heuristics are admissible and therefore always reach optimal solutions. As part of my investigation I also explored the difference in the number of nodes visited by the A* algorithm when using different heuristics.



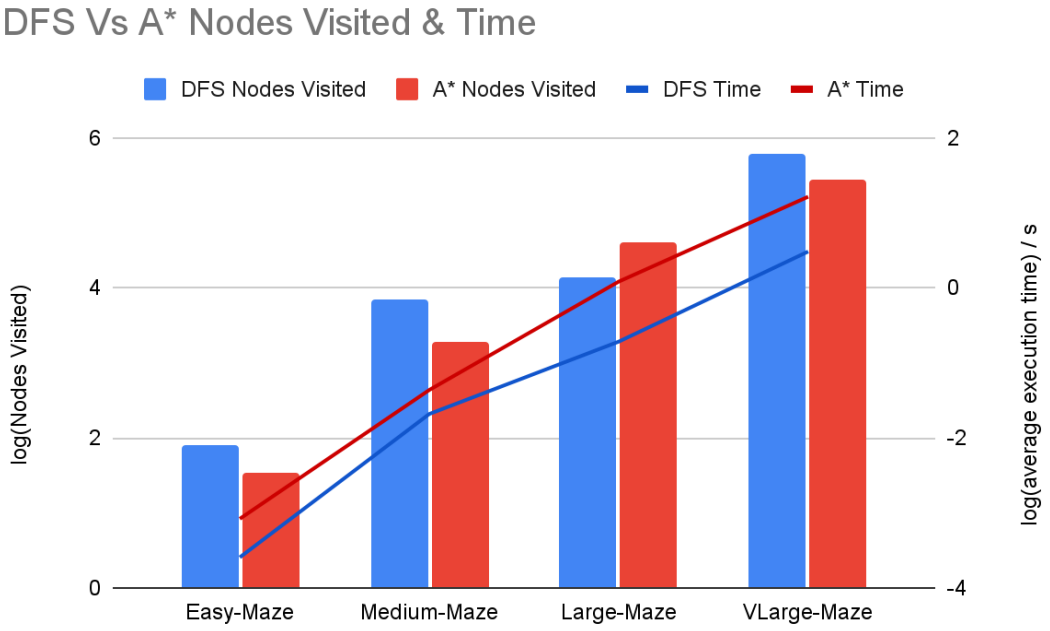
In the plot above, it is important to note that the values for both nodes explored and average time have been logged in order to make the data more visually readable, since the number of nodes changes by multiple orders of magnitude. I then similarly had to log the time to make the time comparable to the nodes explored, and thus showing the directly proportional relationship between the two.

The graph shows how generally (for these specific mazes in all cases) the Manhattan heuristic outperformed (by being lower) the Euclidean heuristic in terms of nodes explored. This is in turn reflected in the average execution time , since this directly impacts the number of recursive calls the program makes and thus the number of operations executed by the processor. Again, this is what I hypothesised given that the Manhattan distance is more representative of the way in which we can traverse the maze and so produces a better estimate of the cost to reach the goal node than the Euclidean distance.

Part 4:

Given the collected data, it is safe to say that the DFS algorithm I implemented was significantly more time efficient than the A* search I implemented, however this is due to the efficiency of my implementations of the algorithms, and not down to the algorithms themselves.

This can be seen in the graph below which shows the log of the nodes visited (again using log for visualisation convenience) as well as the log of the average times for the two algorithms (note I used the manhattan heuristic results since they gave the best performance of my A* search algorithm as discussed above). We can again observe the directly proportional relationship between the number of nodes visited and average time, but can also observe that in all cases apart from the the large maze (where the structure of the maze was beneficial for the DFS algorithm) the A* search algorithm visited significantly fewer nodes than the DFS algorithm.



Another feature of my analysis is that the A* search algorithms I implemented where much more efficient in the paths they found since I used admissible heuristics and therefore the path found by the A* search was always optimal. By contrast, the DFS algorithm depends on the structure of the specific maze for determining which of the possible paths it identifies (exemplified in the maze-Large.txt searches), which results in DFS generally worse performance for identifying efficient paths through the maze. This is highlighted below where the path found by the A* search algorithm for traversing the maze was shorter or equal to that by the DFS algorithm.