ECM2423 AI Coursework
Maze Solving

Harry Findlay - 710026259

March 2023

1 Maze Solver

1.1 Framing maze solving as a search problem

Maze solving is one of the best examples of a search problem. An easy way to visualise this problem is by imagining each position in the maze as being a different state. Each state has neighbours, where each neighbour $\in$ neighbours is a valid move from a given state. Where neighbours is informally defined as neighbours $\subseteq$ all possible moves from a state. I mentioned the word *valid* earlier, and to define this further, a *valid* move is one down a path, whereas an invalid move is to a wall, or where the next state would be outside the bounds of the maze.

Using the idea of states and neighbours one can use search algorithms designed for graphs or trees to find paths to a goal state. In this coursework I have explored depth-first search and A* search algorithms further.

When solving the given mazes using these search algorithms I have made two assumptions, the first being that each maze has a distinct start and end point. The second being that mazes are composed of walls, and these walls are impenetrable.

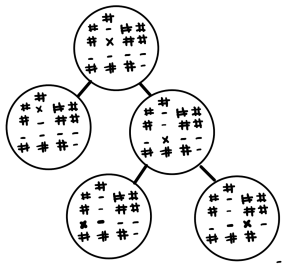

Figure 1: Tree diagram representing states of a maze, where character x represents the current position

Implementation specific features

I have decided to represent the maze as a 2D array, this means that a state represents a position in the array. A position in the array is returned by arr$[i][j]$, where i represents the row, and j represents the column.

I have chosen to represent the maze as a 2D array, rather than an alternative like an adjacency list, which can store a state with all its possible neighbour states, as constructing the adjacency list is an expensive operation. The time complexity of construction is $O(n^2m)$ where n is the number of rows or columns in the maze and m is the number of possible neighbours. The difference in execution times between a 2D array and an adjacency list would become noticeable on the large mazes with many nodes.

To convert a maze text file into a 2D array requires some data cleansing. Newline characters and white space are removed from each of the lines in the file, and then a 2D array is filled with the characters from the text file. The height of the array is how many rows are in the text file, and the width is the number of columns in the file, as we can assume that each row has the same length.

To find the start state of the inputted maze, the function looks at all the positions of the first row in the maze array and searches for a path character, '-'. If the length of the row is n, this is from positions arr[*0*][*0*] to arr[*0*][*n-1*]. Then, to find the goal state, where m is how many rows are in the maze array, it searches positions arr[*m-1*][0] to arr[*m-1*][*n-1*] for a path character.

1.2 Solve the maze using depth-first search

1.2.1 Outlining the depth-first search algorithm

Simply put, depth-first search starts at a given root node and and explores each branch from the root node as far as possible before backtracking.

The theory of depth-first search uses a stack data structure. Firstly, the root node is placed onto the stack which marks it as visited. The function can then be recursively called with the first child of the original root node now being the new root node, while also passing in the visited stack. The function will keep being called until the root node has no more children. If this is the case, a leaf node has been reached, so you pop from the visited stack and call the depth-first search function on the popped node.

The base case of this recursive function is whether the root node is equal to the goal node. If this is the case, you can then return the visited stack, as this is the path from the start to the goal node.

Depth-first search is only concerned with finding the first available path through the maze that the algorithm computes. It is not concerned with the shortest path, other algorithms can be used for this like A* search explained later.

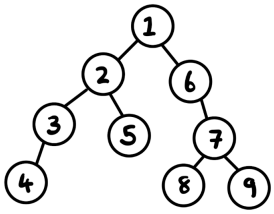

Figure 2: Tree diagram showing the order of traversal when using a depth-first search

1.2.2 Implementation of depth-first search

The code for implementing the depth-first search algorithm can be found in the *ecm2423_ca* folder in the *dfs_solver.py* file.

When implementing the depth-first search in Python a set should be used as well as a stack. This is because a set has a average time complexity of $O(1)$ for checking whether an item is in a set, compared to a stack, which has a $O(n)$ time complexity. A set has a constant lookup time as it utilises a hash table for searching whether an item is in the set. Whereas a stack, which is implemented in Python with a list will have to potentially look through every item.

A set is used along side a stack, as when checking whether a node has been visited, you can utilise the constant lookup time of a set, and then when you want to return an ordered path, you can utilise the fact that a stack is ordered.

1.2.3 Depth-first search statistics

Search ordering: [DOWN, RIGHT, LEFT, UP]

	maze-Small	maze-Medium	maze-Large	maze-VLarge
Start	(0, 1)	(0, 1)	(0, 1)	(0, 1)
Goal	(9, 18)	(99, 198)	(559, 59)	(999, 1880)
Nodes Explored	100	15,806	30,430	3,176,965
Path Length	27	339	1092	3737
Execution Time (S)	$8.297e{-}05$	0.010	0.018	1.458

Search ordering: [UP, LEFT, DOWN, RIGHT]

	maze-Small	maze-Medium	maze-Large	maze-VLarge
Start	(0, 1)	(0, 1)	(0, 1)	(0, 1)
Goal	(9, 18)	(99, 198)	(559, 59)	(999, 1880)
Nodes Explored	215	34,695	52,843	2,681,052
Path Length	41	735	1142	6119
Execution Time (S)	$1.67e{-}04$	0.022	0.028	1.257

An interesting observation from these is that the search ordering, which means which paths we travel down first, drastically changes the number of nodes explored, and the path length. More nodes are explored doing up, left, down, right as you are traversing paths that are less likely to end up with a correct path first. This has a side effect of eliminating many failures quickly, this may explain why the VLarge maze search runs faster than down, right, left, up ordering.

1.3 An improved algorithm, A* search

1.3.1 How A* search works

The simple idea behind the A* search is to include the cost of reaching a node to make an informed choice on which node to visit next. The cost of reaching

the next node will be an estimate, and heuristics can be used to calculate this cost. The most common heuristic used for A* search is Manhattan distance, as this can be only be used when movement occurs only on one axis at a time.

$$|x_1 - x_2| + |y_1 - y_2|$$

Another common heuristic used is the straight line distance between two points (Euclidean distance).

$$d(p, q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}$$

A* expands to paths that are less expensive by using the function

$$f(n) = g(n) + h(n)$$

where $f(n)$ is the total estimated path cost through node n, $g(n)$ is the cost to get to node n from the origin, and $h(n)$ is the estimated cost to the goal node.

The algorithm works, by starting at a given start node, marking it as visited, then computing $f(n)$ for all the *valid* neighbours and setting the parent for all as the current node. Next it will choose the neighbour with the lowest $f(n)$ value out of *all* the expanded to neighbours. Additionally, if one of the neighbours has already been marked as visited, but the newly computed $f(n)$ is lower than the value already stored then update that node with the new $f(n)$ value and update the nodes parent to be the current node.

This will keep running until a list which contains all the valid neighbours is empty, meaning that no path has been found, or a neighbour is the goal node, which means a path has been found and you can reconstruct the path to the start node.

Reconstructing the path is a relatively simple algorithm, it consists of starting at the goal node (if a path was found), adding it to a path list and then recursively calling the reconstruct path method on the current nodes parent.

The base case of this algorithm is if the current nodes parent is empty, meaning that it is the start node, then you can return the reverse of the path list, which will be the path from the start to the goal node.

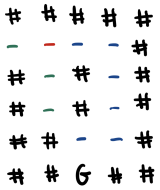

Figure 3: A* will go down the green path, hit a wall, and then jump back to the red node, and then follow the blue path to the goal G.

Why is A* better than depth-first search ?

A* has optimality, which means that the path that it generates is guaranteed to be the shortest path. However, for the algorithm to have the property of optimality, the heuristic used must be admissible.

An *admissible heuristic* is a heuristic that is always optimistic in the estimated path length from a node to the goal. For every node $n, h(n) \leq h^*(n)$ where $h^*(n)$ is the minimum cost to the goal node.

One would also expect that A* will on average explores less nodes than a depth first search to reach the goal node, as as informed choice is used compared to depth-first search which continually visits child nodes.

1.3.2 Implementation of A*

You can find the code that implements the A* search in the *ecm2423_ca* in the *a_star_solver.py* file.

The A* search algorithm makes use of Python's builtin priority queue. A priority queue removes items from the queue based on a priority that an item is given on insertion.

This is extremely useful for the search algorithm as the when all the valid neighbours from a node are each added to the priority queue, they are inserted with a priority set as $f(n) = h(n) + g(n)$. Then the next node to visit can be chosen by just removing an item from the priority queue which will be the node with the lowest priority.

1.3.3 A* search statistics

Manhattan distance heuristic

	maze-Small	maze-Medium	maze-Large	maze-VLarge
Start	(0, 1)	(0, 1)	(0, 1)	(0, 1)
Goal	(9, 18)	(99, 198)	(559, 59)	(999, 1880)
Nodes Explored	64	24,085	43,740	550,033
Path Length	27	321	974	3691
Execution Time (S)	$5e{-}04$	0.144	0.257	3.953

Euclidean distance heuristic

	maze-Small	maze-Medium	maze-Large	maze-VLarge
Start	(0, 1)	(0, 1)	(0, 1)	(0, 1)
Goal	(9, 18)	(99, 198)	(559, 59)	(999, 1880)
Nodes Explored	66	47,787	125,375	828,502
Path Length	27	321	974	3691
Execution Time (S)	$6e{-}04$	0.269	0.719	5.498

1.3.4 Did A* perform better than depth-first search

As you can see from the above tables A* actually ran slower than depth-first search, however, A* has the property of optimality and produces paths with the guaranteed shortest path length. This is a desirable property, as many people would not mind waiting a few seconds to get the shortest route, compared to receiving a route instantaneously, but the route takes them on a much longer journey.

The reason why A* runs slower than depth-first search is likely due to the fact that it uses a custom node class, that has to be instantiated for each neighbour explored, and that a heuristic value has to be calculated along side this. Whereas, during depth-first search only the position in a 2D array is changed, and two collections are updated.

An interesting point to note is that on the medium and large mazes A* using the manhattan and euclidean heuristic actually explored more nodes than depth-first search with down, right, left, up ordering. This is most likely due to these mazes having lots of paths with a low $f(n)$ values near the goal state, that end up being invalid due to walls being met. However, when the maze became very large, A* searched nearly 4 times less nodes than depth-first search to find the path.