ECM 2423 Coursework

Maze Solver

Prinal Patel

Question 1.1 : Describe how you would frame the maze solver as a search problem

The goal of a search problem is to find a particular solution to that problem by searching through a large space of possible solutions. A maze solver would constitute as a search problem by having:

1. An initial state
2. Current state
3. A goal/final state
4. A set of all possible moves from the current state

The Maze Solver can be framed as a search problem where the initial state is the entry/start point of the maze, the goal state is the exit point of the maze, and the set of possible moves can be interpreted as travelling in the defined directions (up, down, left, right)

The Maze also includes obstacles in the form of walls which can lead to deadends / dead-paths. The search problem can be solved by finding the desired path from start to end, whilst avoiding the walls.

Question 1.2 : Solve the maze using depth-first search

1.2.1 : Briefly outline the depth-first algorithm.

Depth-First Search is a search algorithm used to traverse tree or graph data structures. It starts at the root node and explores a branch until it either reaches its goal, or reaches a dead-end, after which it backtracks and explores the next branch. (Can be seen in Figure 1).

So it simply,

- Starts at the starting node and marks it as visited
- Explores the adjacent unvisited nodes and chooses one to visit next.
- If there are no adjacent unvisited nodes, backtrack to the previous node and select another unvisited node to visit
- The last two steps are repeated until either
    - All nodes are visited
    - Exit condition on reaching a goal state is set.

It can be implemented recursively or iteratively using a stack. Recursive calls would use the call stack to keep track of the path being explored, whereas iteratively, a stack data structure could be used to keep track of nodes to be visited and their paths.

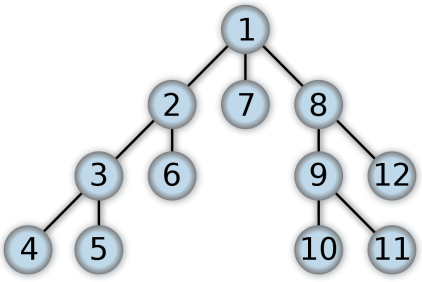

Figure 1 : Depth-First Search (numbered in order of traversal)

Limitations with implementations:

- Recursively → Less efficient for large datasets (max recursive call error).
- Iteratively → Hard to implement where recursive nature is apparent, specifically while backtracking.

Limitation of DFS:

- Does not guarantee finding the shortest path, as it prioritises exploring one path as far as possible before backtracking.
- Can get stuck in infinite loops if the graph contains cycles.

1.2.2 : Implementation of DFS

The code for the implementation of DFS can be found in the file *"DFS_Search.py"*. The path is created on runtime and is contained in a txt file in the *Paths* subdirectory.

I have used an iterative implementation, as using a recursive implementation led to the recursion error (maximum recursion depth exceeded).

A stack is defined to keep track of the nodes to be visited. By default the start node is added to the stack. The validity of the current node is checked (if it is not visited and not a wall), after which it is marked as visited. The for loop explores each possible move (up, down, left, right) and adds all the unexplored neighbors into the stack, along with the update path.

Initially I had defined *"visited"* as a list, which had a significant increase on the execution time. By changing *"visited"* to a set, the execution time improved drastically. The reason for the improvement is that sets are implemented using a hash table which allows for O(1) membership testing, whereas membership testing in lists takes O(n). I found this out by experimenting, and *"maze-Large.txt"* which had a 3800% decrease in execution time; from 3.97 seconds, to 0.11 seconds.

Another key observation made was regarding the order of the exploration of nodes. The nodes are explored in a specific predefined sequence.

For example:

```
#(right, left, down, up)
for moveR, moveC in ((0, 1), (0, -1), (1, 0), (-1, 0)):
```

This would be explored in the order (-1, 0) → (1, 0) → (0, -1) → (0, 1)
Up → Down → Left → Right

The final implemented order is **Down→ Right→ Left → Up**

```
for moveR, moveC in ((-1, 0), (0, -1), (0, 1), (1, 0)):
```

This inarguably makes it more efficient due to the structure of the mazes. As the starting points are at the top left quadrant, and their exit points are in the bottom right quadrant, so movements that focus on heading to the bottom right quadrant are more efficient.

1.2.3 and 1.2.4 : Statistics

This is the first combination of directions I had implemented.

Direction of exploration : **Up** → **Down** → **Left** → **Right**				
	Maze			
	maze-Easy	maze-Medium	maze-Large	maze-VLarge
Execution Time (s)	**=~ 0.0002**	**=~ 0.0350**	**=~ 0.1072**	**=~ 49.4987**
Steps in final path	26	508	1119	5724
Total Nodes (visitable path)	83	9102	82400	918498
Nodes Visited	81	6875	13579	620706
Percent Visited	97.59%	75.53%	16.48%	67.58%

Table 1: DFS Direction Combination 1

Direction of exploration : **Left** → **Up** → **Right** → **Down**				
	Maze			
	maze-Easy.txt	maze-Medium.txt	maze-Large.txt	maze-VLarge.txt
Execution Time (s)	**=~ 0.000211**	**=~ 0.063599**	**=~ 0.482620**	**=~ 151.3690**
Steps in final path	26	736	1049	6130
Total Nodes (visitable path)	83	9102	82400	918498
Nodes Visited	77	8410	72297	778768
Percent Visited	92.77 %	92.40 %	87.74 %	84.79 %

Table 2: DFS Direction Combination 2

According to the structure of the maze entry (top-left) and exit (bottom right) points and the paths, the direction in Table 2 should be the slowest. It has longer execution times and explores a significantly higher number of nodes than the rest of the combinations.

Direction of exploration : **Down→ Right→ Left → Up**				
	Maze			
	maze-Easy.txt	maze-Medium.txt	maze-Large.txt	maze-VLarge.txt
Execution Time (s)	**=~ 0.000093**	**=~ 0.010300**	**=~ 0.091126**	**=~ 4.31625**
Steps in final path	26	338	1091	3736
Total Nodes (visitable path)	83	9102	82400	918498
Nodes Visited	45	2493	10308	113322
Percent Visited	54.27%	27.39%	12.51%	12.34%

Table 3: DFS Direction Combination 3

The direction in Table 3 has the shortest path, fastest execution and visits the least amount of nodes compared to the other combinations, and hence I implemented the search considering this direction.

1.3 : Improved Algorithm

1.3.1 : Brief outline of Breadth-First Search

Breadth-first search explores all nodes at the same level before moving on the next level. It can be visualised as a wave that spreads out from the root vertex and visits all the immediate neighbours that are reachable. By doing this, it essentially explores all possible directions at once, it guarantees that the shortest path is found.

Limitations:

- Requires lots of memory as it has to store all the nodes in a queue.
- It is not efficient when it comes to larger mazes with many paths, as it has to explore unrelated paths.

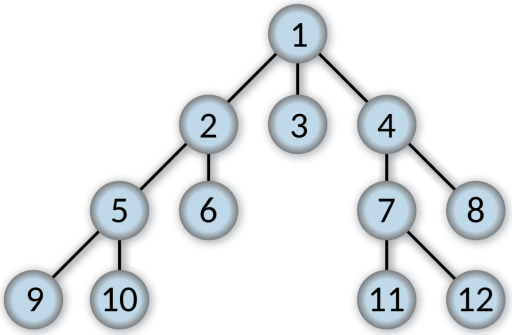

Figure 2 : Breadth-First Search (numbered in order of traversal)

1.3.2 : Implementation of Breadth-First Search

I have used the same implementation as DFS to interpret the given mazes of *'#'* and *'-'*. I have defined a double-ended queue which is used to store the node and its parent node. A double ended queue is more efficient than a queue as it makes it possible to process nodes at a given level faster.

How it works:

When the is not queue empty, unvisited nodes remain:

popleft() is used to retrieve the next node from the front of the queue then mark it as visited. If the node is not visited, then it is added to the visited set and its parent is saved in the parent dictionary (key: (row, col), value: parent).
If the current node is not the goal node, then it loops through the neighbours (down, right, left, up) and checks its validity, then adds it to the queue alongside its parent.

Once the exit node is reached, it backtracks through the nodes from the parent dictionary, starting from the exit node and following the parent nodes. As it traces back through the nodes, it stores each of them in the path, which is then reversed to give the final path

1.3.3 : Statistics

	Maze			
	maze-Easy.txt	maze-Medium.txt	maze-Large.txt	maze-VLarge.txt
Execution Time (s)	**=~ 0.00012**	**=~ 0.00871**	**=~ 0.10486**	**=~ 3.53377**
Steps in final path	26	320	973	3690
Total Nodes (visitable path)	83	9102	82400	918498
Nodes Visited	77	8448	81991	796211
Percent Visited	92.77 %	92.81 %	99.50 %	86.69 %

Table 4: BFS Statistics

1.3.4 : Analysis

When evaluating the effectiveness of a search algorithm, I would consider two important metrics: execution time and the number of steps in the final path.

BFS is generally preferred for finding the optimal path, as it guarantees to find the shortest path between two nodes. However, if space-complexity is a priority, DFS may be a better choice, even if it results in a slightly longer path.
For small mazes with few dead-ends, DFS can be faster than BFS by a negligible margin. However, as the maze size and complexity increases, BFS tends to perform better overall.

The space complexity of DFS can be higher in very deep mazes, as it requires more stack space to keep track of the path. However, it may perform well if the search directions align with the paths in the maze.

Finally, BFS would be better for scalability as it explores the nodes in an organised and efficient manner. As the size of the maze increases, DFS may require more memory and time to find the path due to the probability of the increasing dead-paths it may face.

Overall, the choice between DFS or BFS depends on the specific characteristics of the maze being solved. In the case of the given mazes, BFS tends to perform better, as it finds shorter paths with quicker times despite using more memory.

1. 4 : Further Exploration

1.4.1 : Brief outline of A* Search

A* Search algorithm employs heuristic functions to determine the next node in the path. These functions give an estimate of the minimum cost between a particular node and the target node.

The algorithm merges the real cost of moving from the starting node to a given node
(g(n) / gScore),
with the predicted cost of moving from that node to the goal node
(h(n) / hScore),
and the combined result is used to select the next node to evaluate.

A* functions similarly to BFS but utilises a priority queue that sorts nodes based on the provided heuristic function to guide the search towards the goal node.

It guarantees the shortest path as long as the heuristic function is admissible and consistent.

It is used widely in many fields such as robotics, video games and transportation planning.

Limitations:

- High memory consumption
- Difficult to create an admissible and consistent heuristic function
- Poor heuristic functions can lead a unoptimized path

1.4.2 : Implementation of A* Search

- The A* algorithm uses a heuristic function(Manhattan distance) to estimate the minimum cost between a node and the target node.
- The Manhattan distance heuristic is a function that considers only four cardinal directions (Up, Down, Left, Right), making it perfect for this maze solver.
- A priority queue is used to sort nodes based on their f-score (gScore + hScore)
    - *'gScore*' is the actual cost from the start node to the current node.
    - *'hScore'* is the heuristic estimate of the minimum cost from the current node to the goal node.
- A node object is created for each node that is added to the queue. The object allows for it to store the row, col, gScore, hScore and parent of the node.
- At each step, the algorithm selects the node with the lowest f-score from the priority queue and examines its neighbours.
- The algorithm stops when it finds the goal node or when the priority queue is empty.
- If the goal node is found, the shortest path is constructed by backtracking from the exit node to the start node.
- The algorithm keeps track of visited nodes to avoid revisiting them and infinite loops.

1.4.3 : Statistics

	Maze			
	maze-Easy.txt	maze-Medium.txt	maze-Large.txt	maze-VLarge.txt
Execution Time (s)	**=~ 0.00036**	**=~ 0.01054**	**=~ 0.20943**	**=~ 1.70263**
Steps in final path	26	320	973	3690
Total Nodes (visitable path)	83	9102	82400	918498
Nodes Visited	56	2041	41458	273976
Percent Visited	67.47 %	22.42 %	50.31 %	29.83 %

Table 5: A* Statistics

1.4.4 : Analysis

- A* search is faster than BFS and DFS because it uses a heuristic function to guide the search towards the goal node and explore the most promising paths first.
- BFS explores all nodes at a level before moving to the next level, which can be time-consuming and memory-intensive for large search spaces.
- DFS explores paths until it reaches the goal or a dead end, which can be inefficient if the search space has many deep paths that do not lead to the goal.
- For Small, Medium and Large mazes, A* search finds better paths than DFS and is marginally slower than BFS.
- For significantly larger mazes, such as *'maze-VLarge'*, A* outperforms both BFS and DFS in terms of optimal path and speed.
- The space complexity of A* algorithm can be higher than that of BFS or DFS, as it stores all the visited nodes in a priority queue, which can be exponential in the worst-case scenario.
- However, A* often performs better than BFS and DFS in terms of space complexity because it explores the most promising nodes first, which can lead to finding the goal node with fewer visited nodes and less memory usage.
- The efficiency of A* search depends on the quality of the heuristic function and complexity of the search space, and in some cases, BFS or DFS may be more appropriate or efficient.
- For larger mazes, I would recommend using A* search for optimal and quick pathfinding.