# Maze Solver

## Question 1.1

A search problem has a state space, nodes and arcs. Therefore, for the maze solver to be seen as a search problem, each of these items needs to be defined by the problem.

The state space of the maze solver would be a tree, with the branches being the arcs.

The arcs of the maze solver would be the adjacencies between the empty ‘-‘ characters.

The nodes in this representation would be the empty ‘-‘ characters.

## Question 1.2

### Part 1

1. The Depth-First algorithm operates by first selecting the first direction it encounters.
2. It then searches all of the nodes along this direction, labelling each as “visited” once they have been passed for the first time.
3. If at any point the goal node is found along this direction, the path up to this node is returned and the algorithm terminates.
4. Otherwise, once it reaches the end of this direction, it backtracks until it finds a new direction to explore, and the search fails if no other directions are found.
5. When a new direction is discovered, the algorithm repeats steps 2-5.

### Part 3

In my implementation, I represented nodes by a 2D cartesian coordinate grid, with (0, 0) being the top left position in the maze.

More specific details are available in the README.txt file.

When testing the performance of my algorithm, I disabled output of the path to the console, as on the larger mazes this can take multiple seconds to do. (Default settings except suppressFinalPath = true)

Easy Maze

Text

Description automatically generated

On a small number of nodes, the depth-first search is very quick, taking less than 0.01 seconds to run from compile time.

### Part 4

Medium Maze



Large Maze

Text

Description automatically generated

Very Large Maze

Text

Description automatically generated

From the above results, I drew the conclusion that depth-first search performs quickly for small maximum depths, however for larger mazes a more intelligent algorithm is more appropriate.

## Question 1.3

### Part 1

* The A\* algorithm would produce a more optimal solution than DFS, provided a good heuristic function is used.
* The A\* algorithm would be an improvement over depth-first search, especially on larger graphs.
* This is because there is no decision making in DFS; it just travels down the next direction until it hits a wall, and this is extremely inefficient in larger graphs, with the number of nodes needing to be explored being in the millions for the “maze-VLarge.txt”.
* The use of a heuristic algorithm (A\*) would drastically reduce this time for larger graphs.
* However, A\* may be slower on trivial graphs, due to the added time cost of a heuristic.

### Part 2

### Part 3

Easy Maze

|  |
| --- |
| Number of nodes visited: 36 |
| Number of steps in final path: 27 |
| Execution time: 0.007s |

Medium Maze

|  |
| --- |
| Number of nodes visited: 513 |
| Number of steps in final path: 331 |
| Execution time: 0.037s |

Large Maze

|  |
| --- |
| Number of nodes visited: 40275 |
| Number of steps in final path: 974 |
| Execution time: 20.232s |

Very Large Maze

|  |
| --- |
| Number of nodes visited: 272249 |
| Number of steps in final path: 3691 |
| Execution time: 217.009s |

Maze Output

* All maze paths output to the file PathOutput.txt, and these results can be seen in the ‘paths’ folder.
* Snippets of the outputs for easy, normal and very large are below:

*as\_easy.txt as\_normal.txt as\_vlarge.txt*

### 

--- A\* SEARCH EASY [mazes/maze-Easy.txt] ---

(1, 0)

(1, 1)

(2, 1)

(3, 1)

(4, 1)

(5, 1)

(5, 2)

(5, 3)

(5, 4)

(5, 5)

(6, 5)

(7, 5)

(8, 5)

(8, 6)

(9, 6)

(10, 6)

(11, 6)

(12, 6)

(13, 6)

(14, 6)

(15, 6)

(16, 6)

(17, 6)

(17, 7)

(17, 8)

(18, 8)

(18, 9)

--- A\* SEARCH MEDIUM [mazes/maze-Medium.txt] ---

(1, 0)

(1, 1)

(1, 2)

(1, 3)

(2, 3)

(3, 3)

(3, 4)

(3, 5)

(4, 5)

(5, 5)

(5, 6)

(6, 6)

(7, 6)

(8, 6)

(8, 7)

(9, 7)

(9, 8)

(9, 9)

(9, 10)

(9, 11)

(9, 12)

(9, 13)

(9, 14)

(9, 15)

(9, 16)

…

(198, 99)

--- A\* SEARCH VLARGE [mazes/maze-VLarge.txt] ---

(1, 0)

(1, 1)

(1, 2)

(1, 3)

(1, 4)

(1, 5)

(1, 6)

(1, 7)

(2, 7)

(2, 8)

(2, 9)

(2, 10)

(2, 11)

(2, 12)

(3, 12)

(4, 12)

(5, 12)

(6, 12)

(6, 13)

(6, 14)

(6, 15)

(6, 16)

(5, 16)

(5, 17)

(4, 17)

(4, 18)

(4, 19)

…

(1880, 999)

### Part 4

Unfortunately, the speed of my implementation of the A\* algorithm was significantly slower than that of the depth-first algorithm. This is due to the need of using Node objects rather than pointers to only 4 direction vectors, and the additional overhead for working out the cost of each path.

An example of the Node class being slower: *2976* nodes explored by A\* for the large maze, and *22372* nodes explored by DFS for the large maze, yet those *22372* DFS nodes took less than a second, while the *2976* A\* nodes took 190.

However, the A\* algorithm produced more optimal results, with a number of steps always lower than or equal to that of the depth-first algorithm.