Question 1 | Maze Solver.

1.1:

One can frame the maze solver problem as a search problem where the aim is to locate a path from the starting point to the end point of the maze. This could be done by representing each cell in the maze as a node, and the possible movements from each cell are the edges one would use to connect the nodes in a graph. This would then be solved by the algorithm by visiting the nodes and then following the connected edges until the end point is found, if this isn’t possible then we know that there is no path to the end point.

1.2:

A depth-first search (DFS) is an algorithm for searching tree data structures. The way this algorithm works is it starts at the ‘root’ node (arbitrarily chosen), it will then mark it as a visited node and add it to memory which is usually implemented as a stack data structure. Then while there are still items in the stack it should pop the top node from the stack, if the node is the end node, then it will end here as it is a success. If not, add every unvisited neighbouring cell of the popped node add it to the stack. And if all neighbours have been visited and the stack is empty then no path could be found. So essentially a depth-first search explores as far as it can to the end of each branch before it backtracks and looks at other branches, this makes the algorithm more memory efficient then others. Yet this algorithm does not always locate the most efficient path.

Corresponding code can be run by the file “mazeSearch.py”, please enter the path of the maze to run (if the file is in the same folder as the python script, then just file name), then select “1” for which algorithm to run the depth-first search.

For the ‘maze-Easy.txt’ the depth first search was able to locate a valid path with 41 steps from the starting point of the maze to the end point, with a total of 51 nodes explored. This search took 0.0003 seconds. Yet just by looking at this maze we can see that this was not the most efficient way to get to the end point. The path taken by this algorithm is represented bellow by ‘\*’ added as the valid path. Also see image.

# \* # # # # # # # # # # # # # # # # # #   
# \* \* \* \* \* - - - - - - - - - - - - # #   
# # # # # \* # # - # # - # # # - # - - #   
# - # # # \* # - - # # - # # - - # - # #   
# - # \* \* \* # # # # # # # # # - # - - #   
# \* \* \* # - \* \* \* # - # # - # - # # # #   
# \* # # # # \* # \* \* \* \* \* \* \* \* \* \* - #   
# \* # # \* \* \* # # # # # # # # # # \* # #   
# \* \* \* \* # - - # # - - - - - - - \* \* #   
# # # # # # # # # # # # # # # # # # \* #

Qr code

Description automatically generated with medium confidence

This algorithm also works with any maze of the same format. The statistics for each of the example mazes have been calculated bellow.

Path found with 625 steps in 0.0104 seconds. With 5757 nodes explored. Using Depth First Search. With maze-Medium.txt  
Path found with 1142 steps in 0.0225 seconds. With 14688 nodes explored. Using Depth First Search. With maze-Large.txt  
Path found with 4053 steps in 0.1347 seconds. With 88693 nodes explored. Using Depth First Search. With maze-VLarge.txt

The paths for each of these maze’s can be seen bellow in their generated image representation. The blue line represents the path that the algorithm takes, the black lines is the maze itself (‘#’), the white lines are valid places (‘-‘) and the locations highlighted in red are the visited nodes. These images can be regenerated when running the program.

maze-Medium.txt

A picture containing qr code

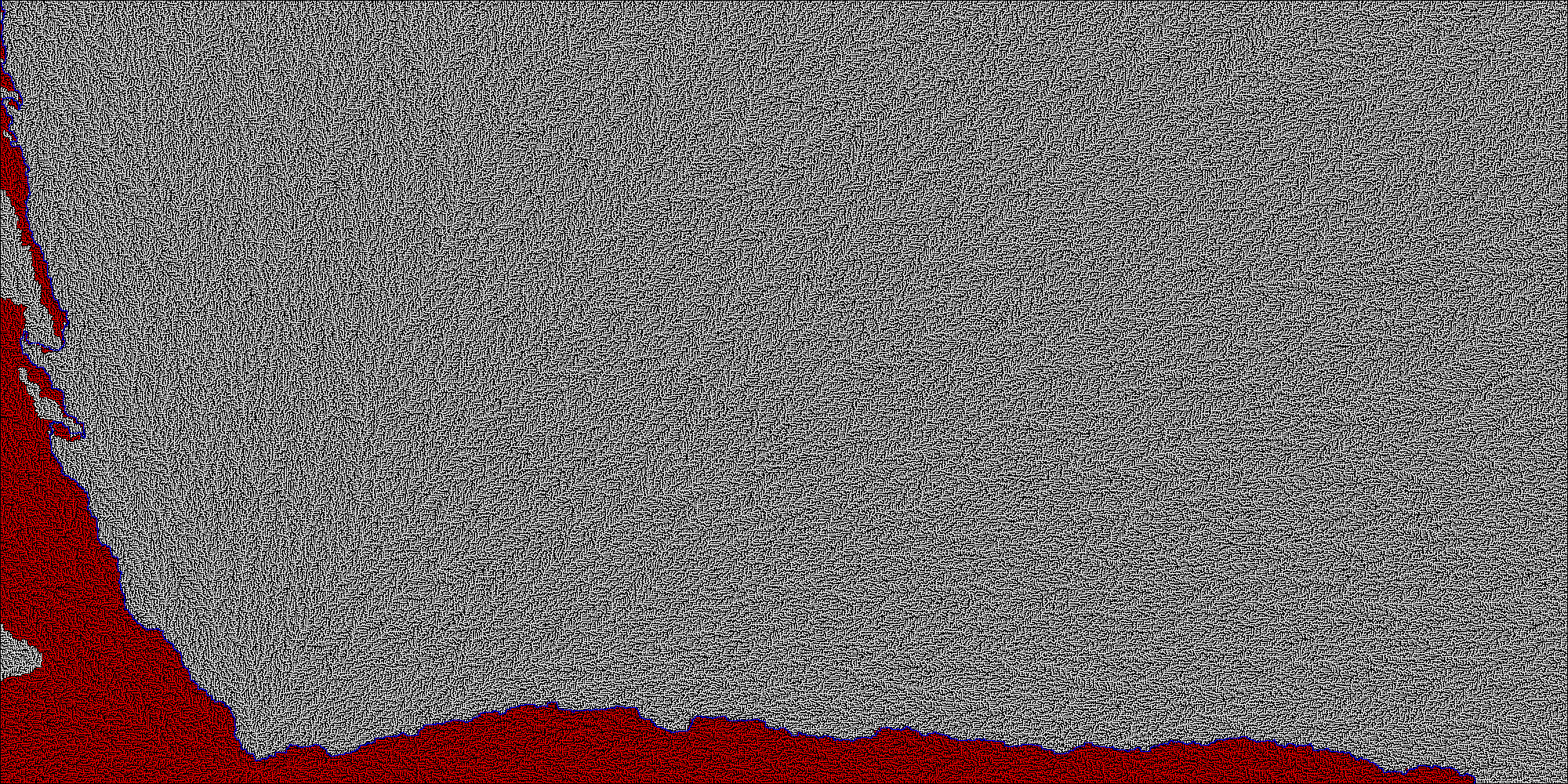
Description automatically generated

maze-Large.txt:

Map

Description automatically generated

maze-VLarge.txt:



After analysing these statistics and paths we can clearly see that the number of nodes explored, and time taken increase as the size of the maze increases. But this is expected as the larger mazes have more available paths, and thus require more exploration to locate a path to the end goal this is also why the number of steps for the path increases with size as the maze itself is longer. With the larger mazes it is hard to see by eye if these paths are the most efficient, but we can clearly see in the ‘maze-Medium.txt’ that the path is not the optimal and there is a shorter cut through in the centre that this algorithm does not find. We can also calculate the average time spent per node at each maze size and we can see that it is relatively consistent across all mazes regardless of size.

* 'maze-Medium.txt': Average time spent per node = 0.0104 seconds / 5757 nodes = 0.00000181 seconds/node
* 'maze-Large.txt': Average time spent per node = 0.0225 seconds / 14688 nodes = 0.00000153 seconds/node
* 'maze-VLarge.txt': Average time spent per node = 0.1347 seconds / 88693 nodes = 0.00000152 seconds/node

1.3:

Another algorithm that can be used to solve this maze is breadth first search (BFS), this algorithm works by exploring all the neighbouring nodes of a given node (nodes at the same level) before moving the next level of nodes. This way ensures that the shortest possible path to the goal is found which is an improvement over Depth First Search as this will not always be guaranteed. It is also less likely to get stuck in an infinite loop or a long path before finding the destination. Essentially breadth first search is a uniform-cost search algorithm that systematically explores all possible paths from the starting point to the goal point. It first explores all paths at a depth of one from the starting point, then all paths at a depth of two, and so on until the goal point is found.

Corresponding code can be run by the file “mazeSearch.py”, please enter the path of the maze to run (if the file is in the same folder as the python script then just file name), then select “2” for which algorithm to run the breadth-first search.

This algorithm was able to run and find a path on all versions of the maze. Statistics for how each of the mazes where solved are bellow.

* Path found with 27 steps in 0.0005 seconds. With 86 nodes explored. Using maze-Easy.txt
* Path found with 321 steps in 0.1698 seconds. With 143502 nodes explored. Using maze-Medium.txt
* Path found with 974 steps in 0.2184 seconds. With 162500 nodes explored. Using maze-Large.txt
* Path found with 3691 steps in 2.3642 seconds. With 1603843 nodes explored. Using maze-VLarge.txt

From this we can see that all mazes had a shorter path than we could previously see with the other algorithm, with the ‘maze-Medium.txt’ required 94.7% more steps to find a solution with the depth-first search algorithm. Yet with this we can see that this algorithm is exploring a considerable amount more of the maze with the average number of nodes explored increasing by around 1300%.

* For 'maze-Easy.txt', the percentage increase was ((86 - 51) / 51) x 100% = 68.63%
* For 'maze-Medium.txt', the percentage increase was ((143502 - 5757) / 5757) x 100% = 2405.61%
* For 'maze-Large.txt', the percentage increase was ((162500 - 14688) / 14688) x 100% = 1009.34%
* For 'maze-VLarge.txt', the percentage increase was ((1603843 - 88693) / 88693) x 100% = 1707.12%.

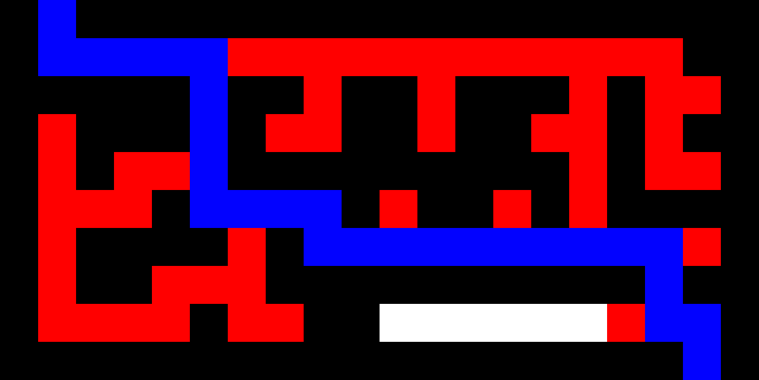
Which is an extremely high increase in memory usage. Another comparison between both algorithms is the average time spent per node. For this algorithm it is also relatively consistent at:

* 'maze-Medium.txt': Average time spent per node = 0.1698 seconds / 143502 nodes = 0.00000118 seconds/node
* 'maze-Large.txt': Average time spent per node = 0.2184 seconds / 162500 nodes = 0.00000134 seconds/node
* 'maze-VLarge.txt': Average time spent per node = 2.3642 seconds / 1603843 nodes = 0.00000147seconds/node

These figures are insignificantly different from the DFS algorithm.

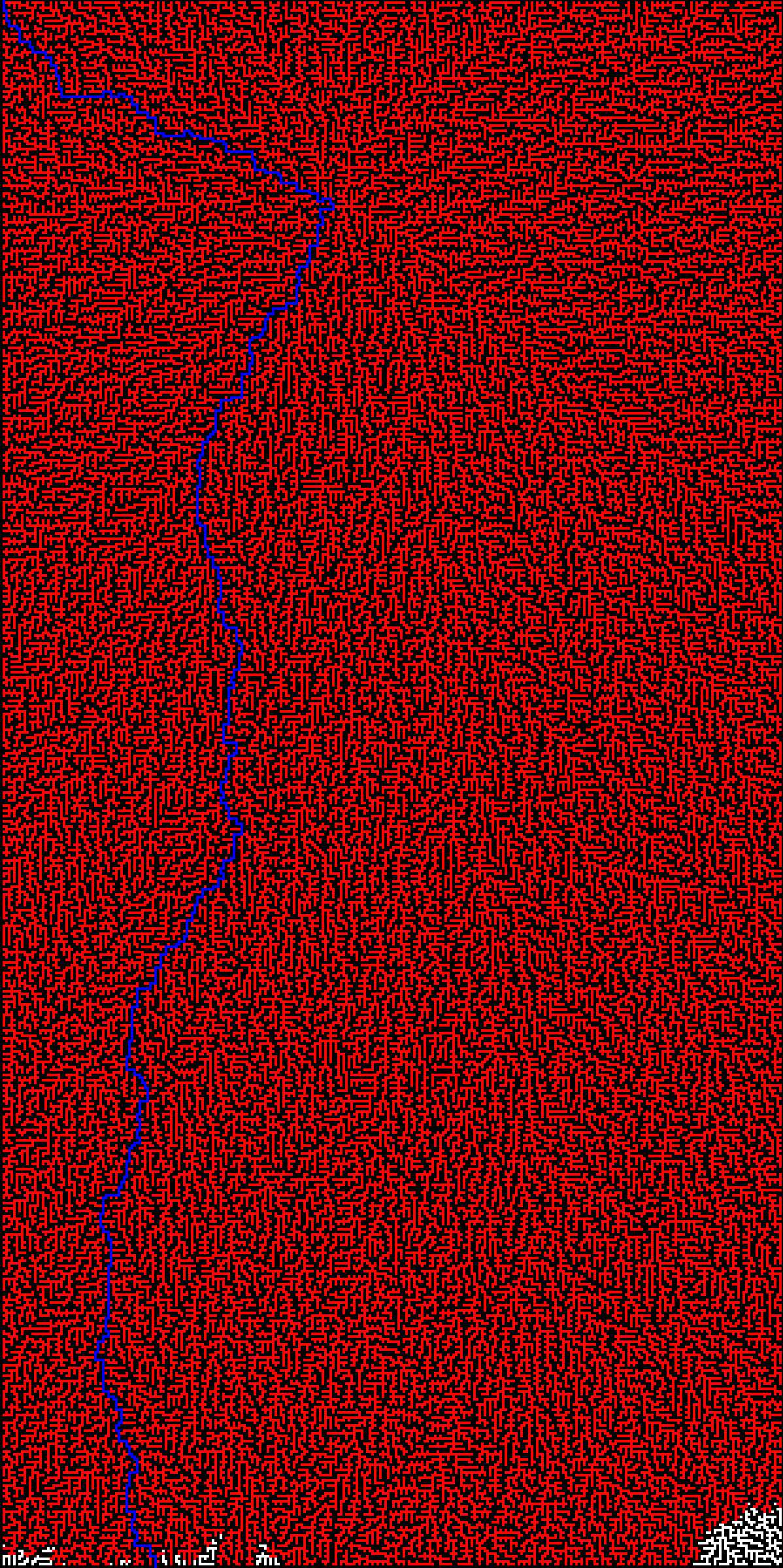
Furthermore, with this algorithm we can see compared to the Depth-First search the amount of time to find the solution is longer. Yet all of this was expected as a Breadth-First search algorithm is not used for efficiency in finding the solution, but the efficiency of the solution itself. The paths that the algorithm found are visible below as images as it is easier to visualise the path, these images are in the same key as before. The paths also output as a new .txt file in the same style as the maze-XXX.txt but it uses ‘\*’ to mark the path to the end goal.

maze-Easy.txt:

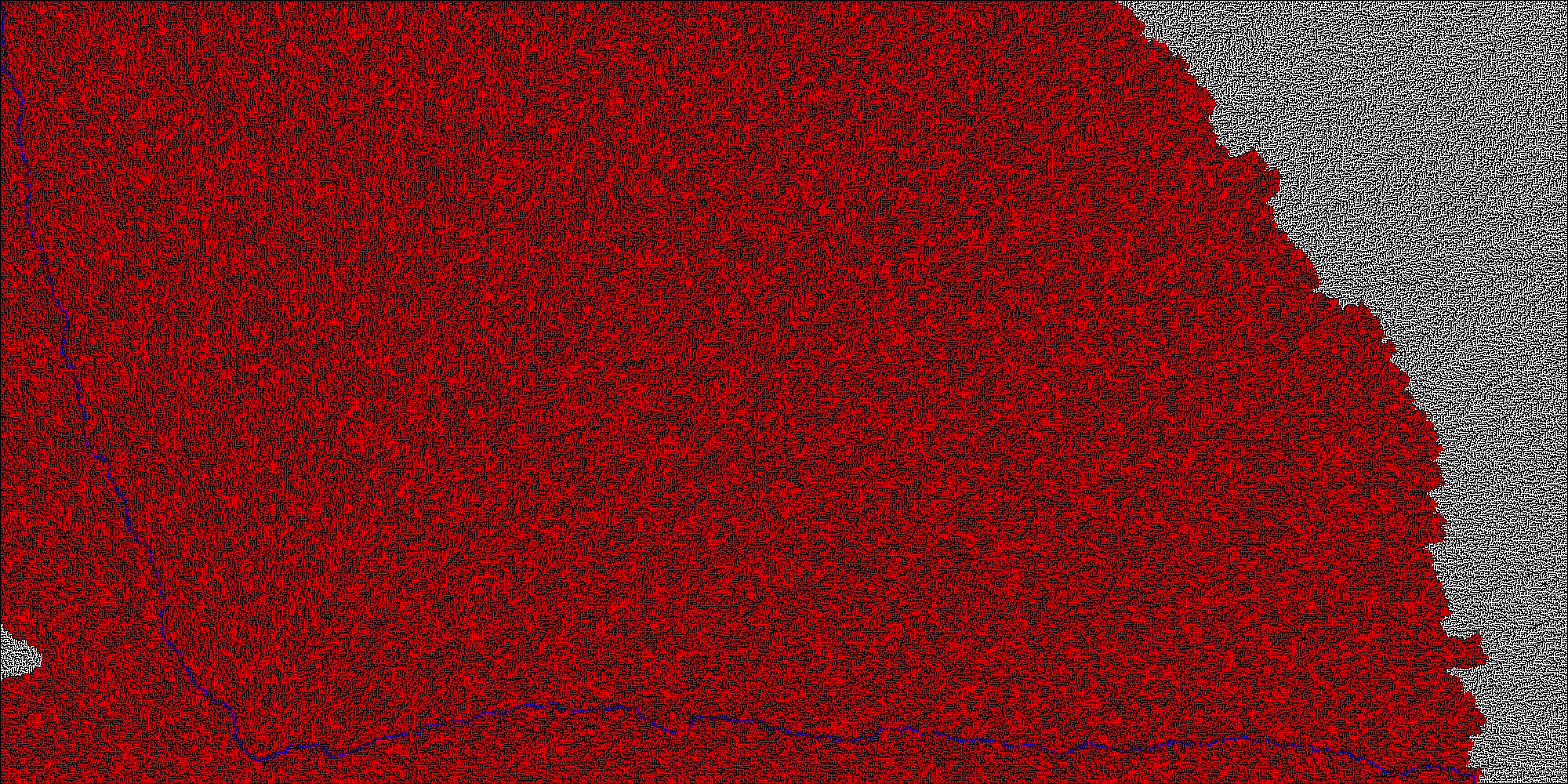


maze-Medium.txt: A picture containing background pattern

Description automatically generated

maze-Large.txt (rotated 90 degrees clockwise):

maze-VLarge.txt:



When visually analysing these images, we can see that many more nodes are explored on the BFS rather than DFS. We can also see, more clearly in the smaller mazes, that there does not seem to be a shorter path which should be the case for this algorithm, and in conjunction with the statistical ‘steps in path’ thus we can say that this algorithm performed better than depth-first in this way.

Further Experimentation:

From analysing both algorithms I wanted to see if there was an algorithm that I could say was the ‘best of both worlds’ where it would try to find the shortest path like BFS, but also not use as much memory and searched nodes as BFS and more to the amount of usage as DFS.

One algorithm that I have found that is useful is a A\* Search. This algorithm is a lot more complex than DFS or BFS as it runs a heuristic calculation to see which nodes would be better to move to. It does this by considering the “g score” (distance from start to current) of a node and its “h score” (distance from current point to goal). A\* is generally considered better than both other algorithms discussed as it is more efficient and guarantees to find the optimal path (the path with the lowest cost). Additionally, A\* search can avoid exploring paths that are unlikely to lead to the goal by using the “h score” as an estimate of the remaining distance, which makes it more efficient than both others. Overall, A\* search strikes a good balance between completeness and efficiency, making it a popular choice for maze solving problems.

I was able to implement this algorithm into my program using a heap. The corresponding code can be run by the file “mazeSearch.py”, please enter the path of the maze to run (if the file is in the same folder as the python script, then just file name), then select “3” for which algorithm to run the A\* search.

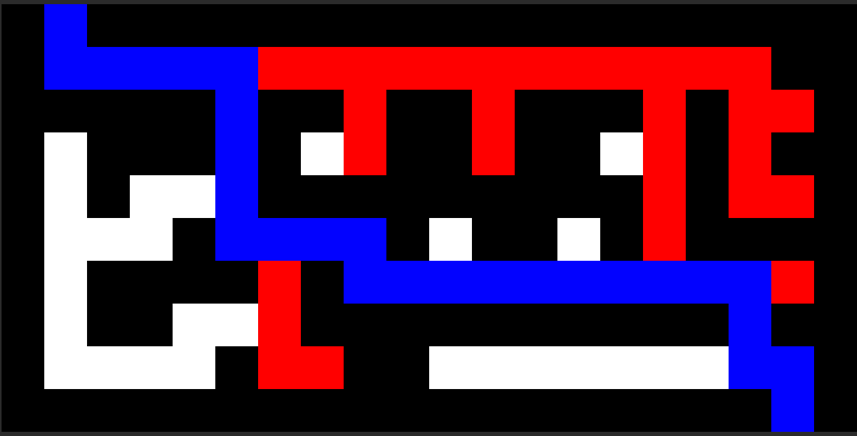
After running this algorithm on all example mazes we can see that all the statistics have moved to where expected.

* Path found with 27 steps in 0.0006 seconds. With 64 nodes explored. Using maze-Easy.txt
* Path found with 321 steps in 0.0509 seconds. With 24085 nodes explored. Using maze-Medium.txt
* Path found with 974 steps in 0.0989 seconds. With 43770 nodes explored. Using maze-Large.txt
* Path found with 3691 steps in 1.4611 seconds. With 550033 nodes explored. Using maze-VLarge.txt

We can see that in comparison the A\* search algorithm finds the same shortest solution as the BFS in all mazes, but it also explores are fewer nodes than BFS does. This algorithm also runs faster on average than BFS, yet it is still not as fast as DFS as an A\* search has more calculations to do and DFS just tries to find any path to the goal and does not care for efficiency of its route.

The images for to display the path are also available bellow.

maze-Easy.txt:

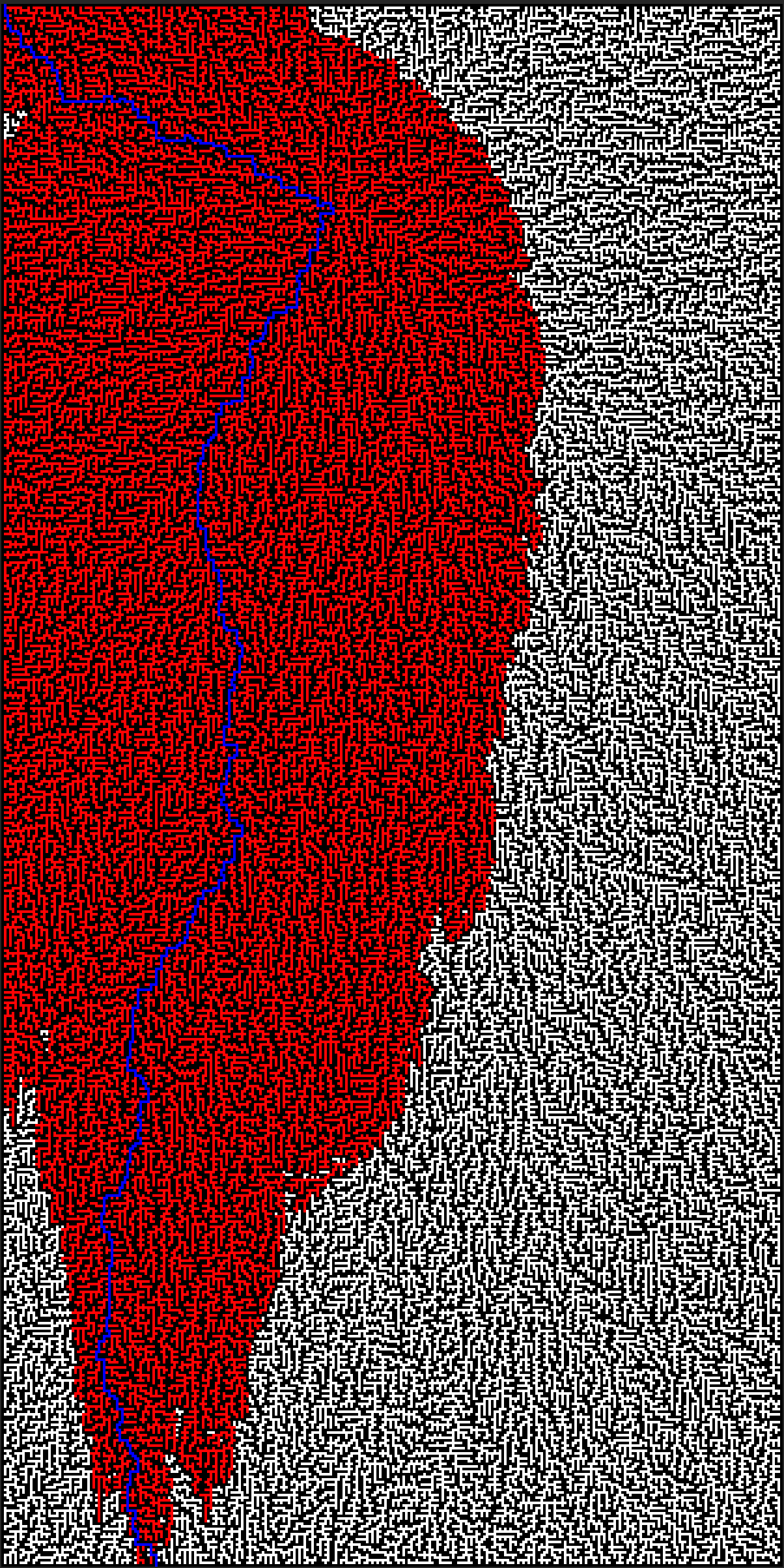


maze-Medium.txt:

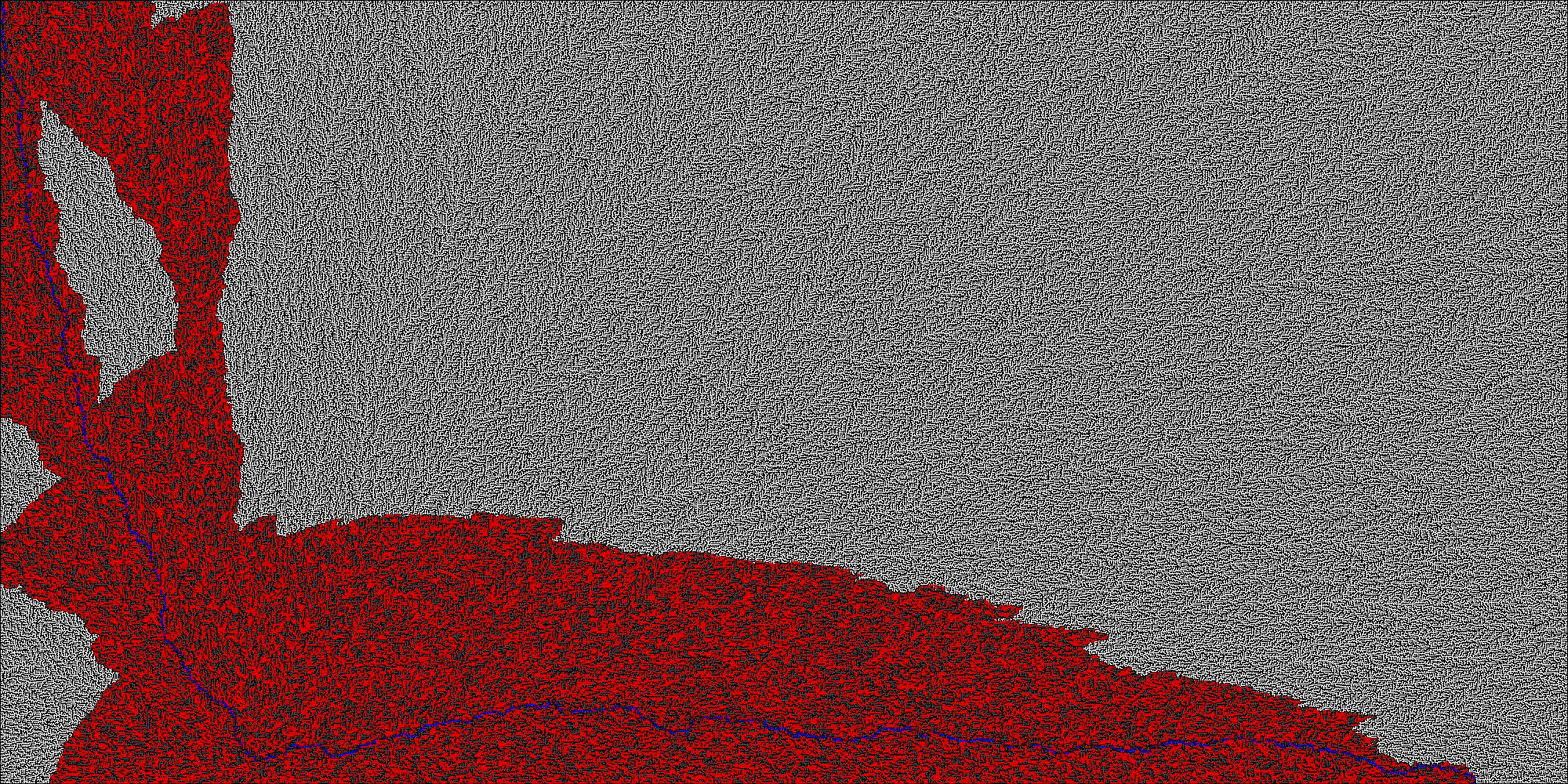
Qr code

Description automatically generated with medium confidence

maze-Large.txt (rotated 90 degrees clockwise):



maze-VLarge.txt:



Again, by visually analysing these images we can see that the number of nodes explored in the A\* search algorithm is much less when comparing to BFS yet is still more than DFS.

Because of the way the algorithm is written there could be room for potential improvements by changing the heuristic function if speed is needed rather than accuracy for example.