159.372

Intelligent Machines

Assignment 3 Report

Chris McDonald

20009360

**Part 1:**

To integrate the mazes into the search algorithm, I decided to transform the data into a format where it would be usable by the existing search function, rather than creating my own search function to access the maze data in its original format.

To begin with, I imported (using NumPy), the maze data for the maze to be searched and the starting node locations for the mazes of that particular connectivity. After converting the array items to integers, I sliced the lists to remove the first 3 items of the array, which only contain information about the maze size and its connectivity/number of starting locations.

To facilitate navigation between nodes in the maze, I elected to give each node a unique name consisting of a letter followed by a number. So for each of the 15 maze rows, they were given the letters A-O from top to bottom. Then for each of the columns, they were given the numbers 0-14 from left to right. So for example, the top-left node was given the name ‘A0’ whilst the bottom-right node was given the name ‘O14’. This would ensure each node had a unique name and were also named in a way that would indicate position and path movement throughout the maze.

I then made an array of all possible letter values from A to O whilst also creating an array of bit values that would be used to indicate the walls surrounding a particular node. This would come in handy later when it came to transforming the maze data.

To randomly choose a starting position, I first retrieved a random number that corresponded to an index in the array of possible start locations. Given that each starting point was attributed to two consecutive numbers in the array (corresponding to the relevant column and row respectively), if the moduli of the chosen index was zero, the following number would be the relevant one for that data-point. Conversely, if the moduli was not zero, then it would be the preceding number in the array that corresponded to the one chosen. Once the relevant numbers have been obtained from the array, the node’s name could then be obtained (concatenating the letter at the row index of the starting node with the column number of the starting node), giving us our starting node in the maze.

In order to assign names to each node in the maze, I decided to create a Node class which would then be initialized with a name and the number in the maze data array corresponding to that particular node. So while iterating through the array of maze data, the node’s name was assigned, a Node object was created and then appended to a list. So upon completion, I had an array of 150 Node objects, each with its own name and data number.

From here, a set was initialized to hold the names of every node in the maze.

The most crucial part of adapting the maze data to make it usable with the search function was making sure all the arcs between nodes were correct and not breaking any walls/boundaries.

The following is pseudo-code describing the process of defining arcs between nodes:

*For each node object:*

*If node is not the goal node (node number value is under 16):*

*Get the bit value corresponding to the node’s number (to define the walls)*

*If node is goal node (node number value above 15):*

*Assign node as goal node*

*Get the bit value corresponding to the node’s number (to define the walls)*

*If node doesn’t have a left wall and isn’t in the left-most column:*

*Find the node to the left of the current node*

*Get this node’s bit value*

*If this node doesn’t have a right wall (no barrier between nodes): Create an Arc between the two nodes and append to arc set*

*If node doesn’t have a bottom wall and isn’t in the bottom row:*

*Find the node directly below the current node*

*Get this node’s bit value*

*If this node doesn’t have an upper wall (no barrier between nodes):*

*Create an Arc between the two nodes and append to arc set*

*If node doesn’t have a right wall and isn’t in the right-most column:*

*Find the node to the right of the current node*

*Get this node’s bit value*

*If this node doesn’t have a left wall (no barrier between nodes): Create an Arc between the two nodes and append to arc set*

*If node doesn’t have a top wall and isn’t in the top row:*

*Find the node directly above the current node*

*Get this node’s bit value*

*If this node doesn’t have an bottom wall (no barrier between nodes):*

*Create an Arc between the two nodes and append to arc set*

Once the data has been iterated through, we now have a set that constitutes every single possible arc between neighbouring nodes in the maze.

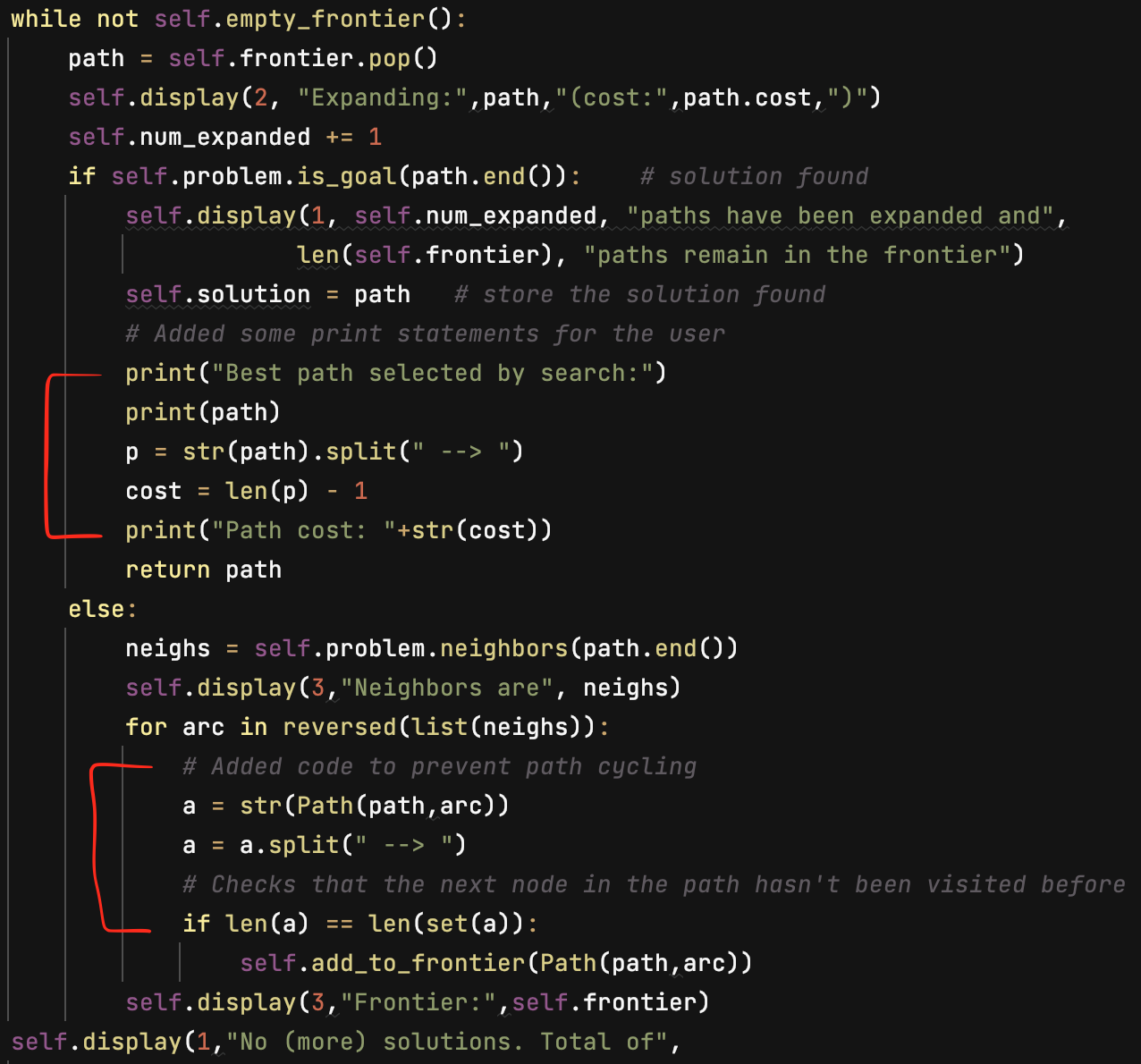
The final variable needed to search the maze (apart from the heuristic function) is a set of all the node positions. This was simple as the maze is laid out like a grid with each node being equidistant from its neighbours. So it was simply the row number (the letter’s position in the alphabet minus one) and then the column value.

So now the maze data is in a format that allows it to be read by the search function.

**Part 2:**

The only pre-existing function that I changed for this assignment was the *search* function in the *Searcher* class of the *searchGeneric* file. The original implementation of this function did not do cycle checking, so it was necessary to implement it to ensure the DFS algorithm did not get stuck in any cycles.

The functionality added by me is shown below (highlighted sections only):



The first section is simply printing relevant information to the console whilst also calculating the cost of the path chosen as the solution (done by subtracting 1 from the total number of nodes in the path), then printing that to the console too.

The second part is checking for cycles. In the original code, the path was automatically added to the frontier, even if the final node in the path had been visited before. This could cause the code to enter a cycle (for example, if the path reached a dead end, there was a chance it would just continually move between the preceding node and the dead end infinitely). So before a path is added to the frontier, I ran a check to see if the next node in the path had been visited before. I did this by stringifying the current path, splitting it on the arrow so I was left with a list of node values, then comparing the length of this list to a set with the same elements. The reason for doing this is that a set can only have unique values, so if there was a difference in the lengths, this would mean that there was a node in the path that had been (or will be) visited twice. If it was the case that the list and set were the same length, the path would be added to the frontier, otherwise it wouldn’t be, which would remove that path from the frontier altogether since it was popped from the frontier at the start of the function and not added back in.

The only other significant alteration to the original code was to implement multi-path pruning to the A\* search. While cycle checking was necessary for DFS, multi-path pruning is better suited for A\* searches. But the original code in the *searchGeneric* file explicitly stated that it did not have MPP. But the search function in the *searchMPP* file did. So I simply brought that function over to the *searchGeneric* file and instructed the A\* search algorithm to use that search function as opposed to the one for DFS which implemented cycle checking.

I had thought about changing the way the DFS chooses paths because it does so in a uniform way. As seen in the code above, it gets the arcs to the neighbouring nodes, which always follow the same order: left neighbour, bottom neighbour, right neighbour, top neighbour (assuming all 4 neighbours can be travelled to directly). When this list is iterated through, it is done so in reverse, so the last arc iterated through will be added to the frontier last (assuming the node hasn’t been visited already). This means that the node will always have a movement priority of left, down, right, then up. I considered shuffling the list of arcs to neighbouring nodes so that the direction the path will take to the following node will always be random. But this gave very varied results. For example, running a search on a low-connectivity maze from the same start node would be different every time in terms of the path chosen, runtime and paths explored. So for the purpose of collecting data to compare to the A\* search, I decided to keep the DFS as is, so it will always search in the same way.

**Part 3:**

For my heuristic function, I treated it similarly to a best-first search. For each node in the maze, I calculated both the row and column differences between this node and the goal node. Then I added these two values together to give the node its heuristic value.

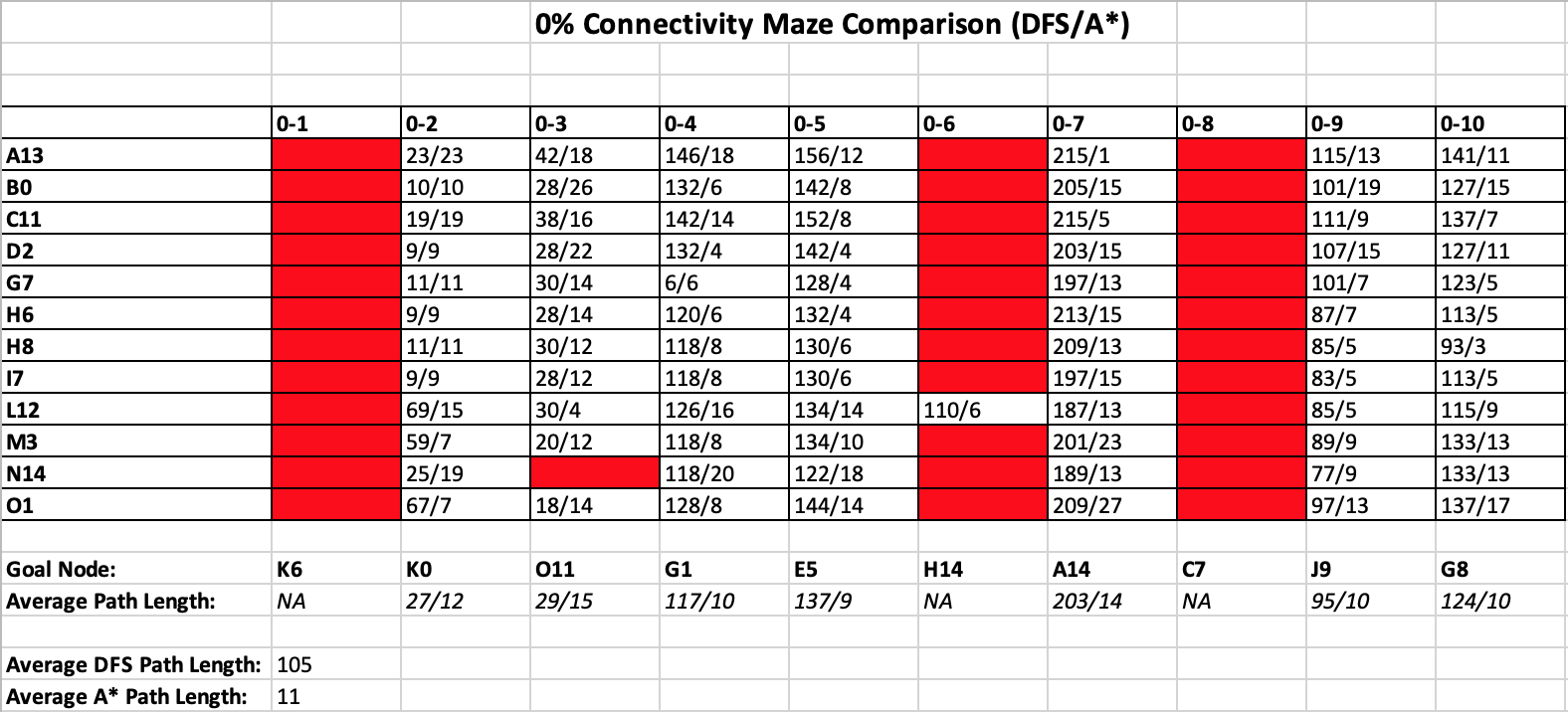
I had initially thought about calculating the heuristic value to be the exact cost it would take to get from the current node to the goal node, taking the maze walls into account. But this goes against the idea of a heuristic function being an **estimate** of the cost of the shortest path. It wouldn’t be an estimate if it was calculated exactly and essentially solved. Plus, it would be computationally expensive to do this for every node in the maze. The heuristic function I decided on is efficient to compute since it’s just adding two numbers that are both calculated easily by taking the difference between the positions of two nodes.

So this simple heuristic function calculates the absolute shortest path that would be possible if there were no walls in the maze. But obviously with walls in the maze, just because a node has a lower heuristic value doesn’t necessarily mean that it is part of the best path, as there may be a wall or many walls between it and the goal node that it needs to circumvent. But the point of a heuristic value is that it should be an **underestimate** if there’s no path that costs less than the heuristic value, which is the case here, there is no possible path that could cost less than what the heuristic function for that node is. So this is an example of an **admissible heuristic**.

**Part 4:**

To illustrate how the DFS and A\* compare, I have compiled spreadsheets for each level of maze connectivity. These spreadsheets show a comparison between the path costs of the paths selected by the DFS and A\* for each starting point and for each maze. Some things to note:

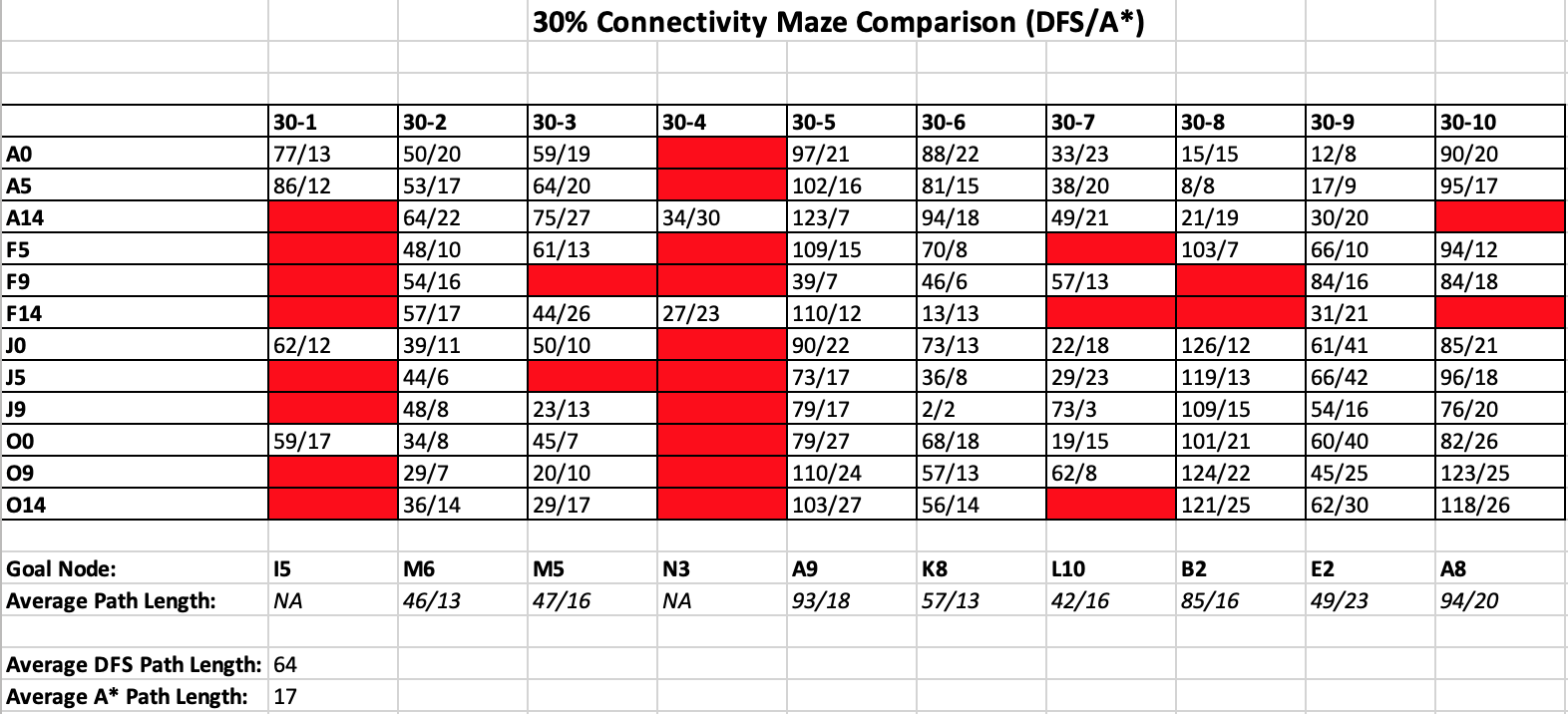
* Cells in red are where the DFS did not find a solution within 30 seconds, so they were excluded.
* Maze averages were not taken if the maze had a majority of red cells.
* All average path costs were rounded to the nearest whole number.
* The values along the y-axis are the starting nodes for the maze.



As you can see, 2 of the mazes had every instance of a DFS take in excess of 30 seconds, while one had all but one do the same. One reason for this is due to the fact that these 3 mazes had a goal node that could only be approached from one direction (1 & 8 would need to be approached from below, and since upward movement is the lowest priority for this DFS, it would take significant time to find a path). Interestingly, Maze 2 had a significant number of starting points that would yield an equal path length for DFS and A\*, despite it also only being accessible from one direction. But the reason for this is relatively simple. When the DFS starts, it immediately creates a path travelling left through the maze until it hits a wall, then it will switch to travelling downwards until it is able to travel left again. We see that the goal node is K0, which sits on the left-most column in the maze. Also, the goal node can only be accessed from above. So the DFS travels left from the start node until it hits the maze wall, then it travels straight down to the goal node, which is also incidentally the shortest path it could have taken. You can also see that if the start node is situated on a lower row than the goal node, the DFS suddenly yields a much higher path cost than its counterpart A\* search.

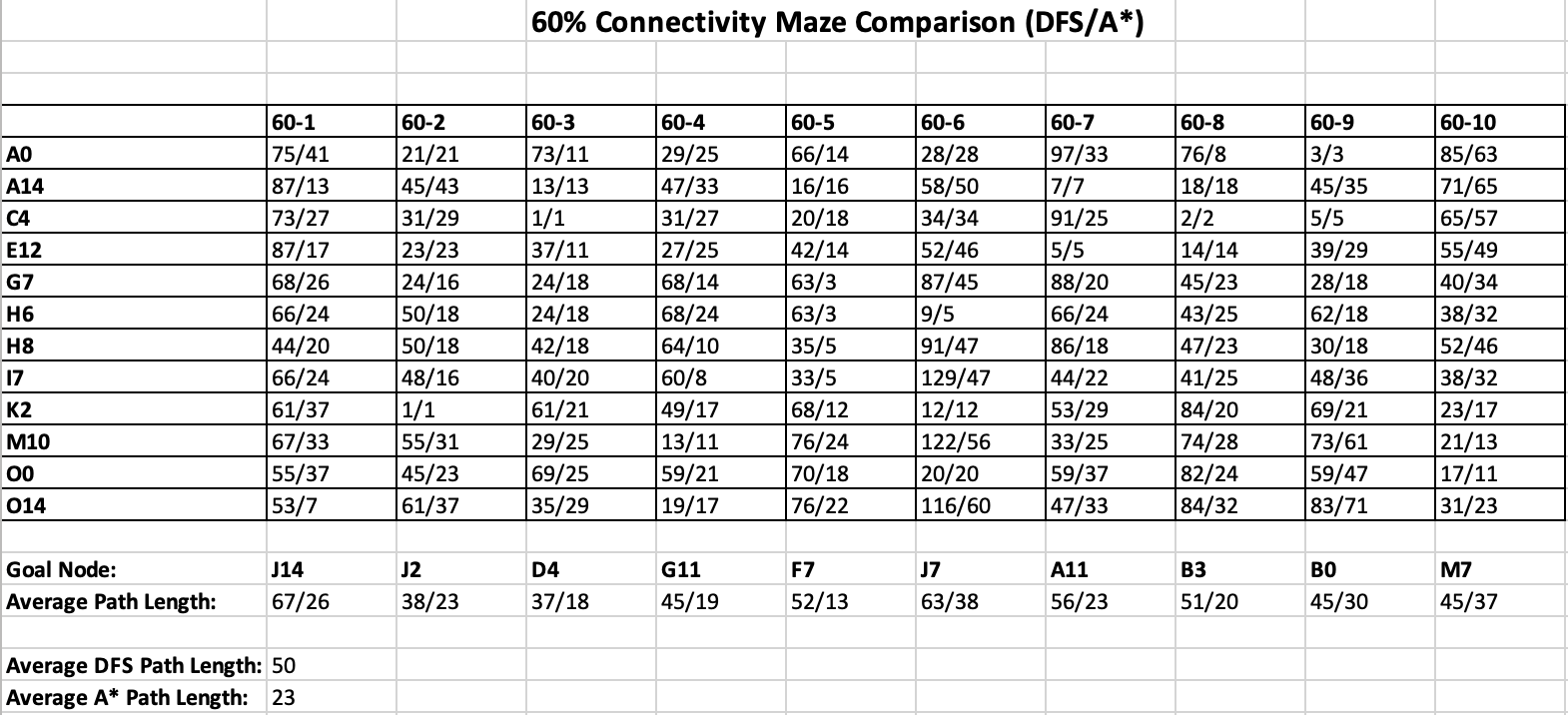
There were only a few instances where the DFS and A\* selected paths of the same cost, but for every other one, the DFS path had a significantly higher cost than the A\* search. As seen from the total path average for this maze connectivity, the DFS had an average path cost almost 10 times higher than that of the A\* search. In each scenario, the DFS also had to explore a far greater number of paths than the A\* search did, making the A\* search better and more efficient at this level of maze connectivity.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



While we see a stronger performance from the DFS here in terms of path cost when compared to A\*, more the half the mazes had at least one start point where the DFS failed to find the goal node within 30 seconds. Mazes 1 and 4 both had high rates of failure (in terms of finding the goal node within the timeframe) and Maze 4’s could most easily be explained by the position of the goal node. Because the DFS gravitates towards the bottom-left of the maze first and foremost, it passes by the entrance to the goal node relatively early on. So in order for the DFS to have access to this node again in order to explore any of its neighbours, it will have to explore the majority of all other possible paths first. This, coupled with the fact that the goal node can only be accessed from below, is one explanation for the poor DFS performance.

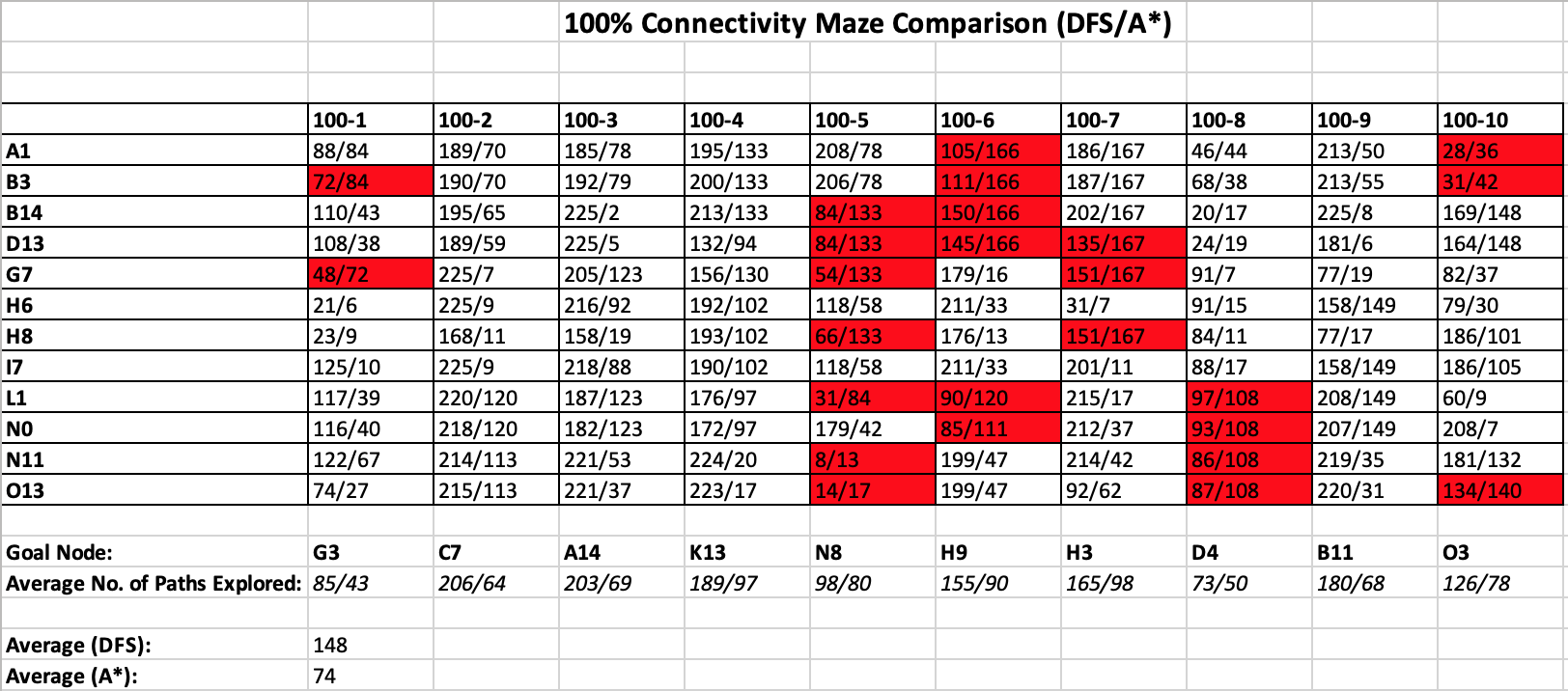
But across the board, we see that DFS has massively improved its average path cost. It now only comes to about 4 times the average path cost as A\*, which is a big improvement from the 0% connectivity mazes.



Once we get to the 60% connectivity mazes, we see that all DFS runs succeeded in the relevant timeframe, and while they performed even better still, they still fall short of A\* searches. Aside from a few cases where both searches yielded paths of equal cost (typically when the goal node was very close by), A\* still found a path of less cost and did so by exploring less paths than DFS did.

Mazes of 100% connectivity are where DFS finally proves its worth. In every instance, DFS found the same exact path as the A\* search. But this is due to the fact that these mazes have only one possible path to the goal node, so as long as DFS found the goal node, it would match whatever path A\* found.

So rather than a spreadsheet of path costs, I decided to do one showing the number of paths explored before finding the goal node, which yielded some interesting results:



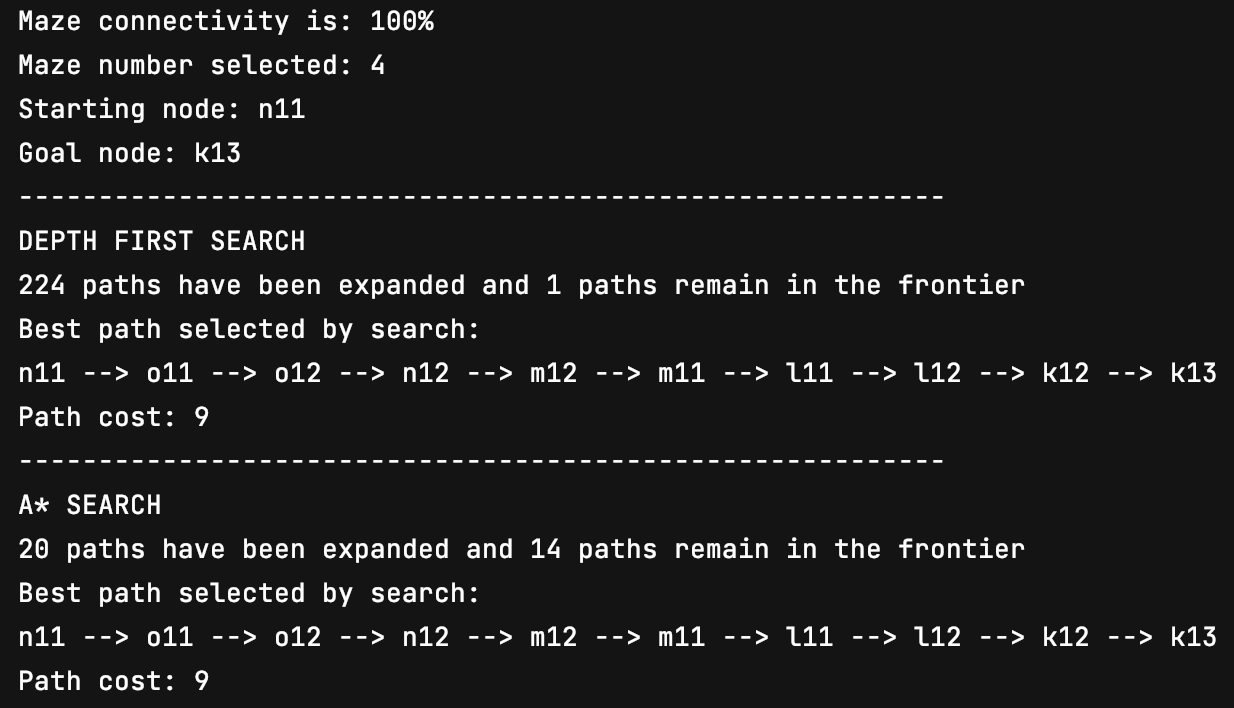
Of the 120 variations tested, 25 of them actually yielded better results for the DFS than A\* (these are highlighted in red). In these instances, the DFS actually explored less paths than the A\* search whilst searching for the goal node. So in roughly 20% of cases, DFS actually outperformed A\*. But when taking averages for all starting nodes on a particular maze, the A\* search still explored less paths on average than DFS for every single one of the mazes of this connectivity.

**Part 5:**

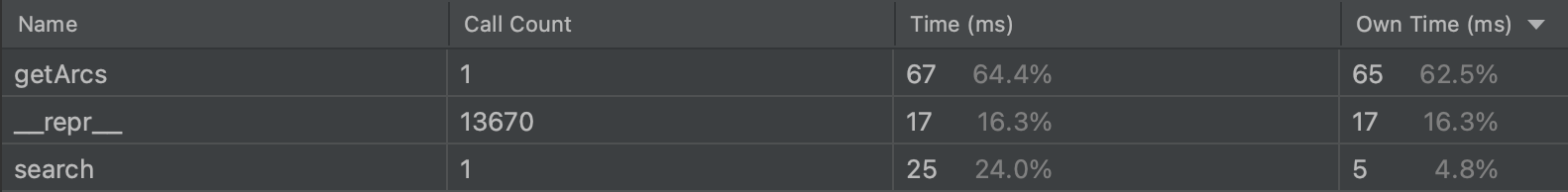
Connectivity plays a big part in determining both the runtime of the searches and also the cost of the paths when comparing DFS to A\*.

For example, let’s look at a maze with 100% connectivity. Because there is only one possible path to the solution from the starting node (all the rest would result in dead-ends), both the DFS and A\* searches will find this optimum path in roughly the same runtime length.

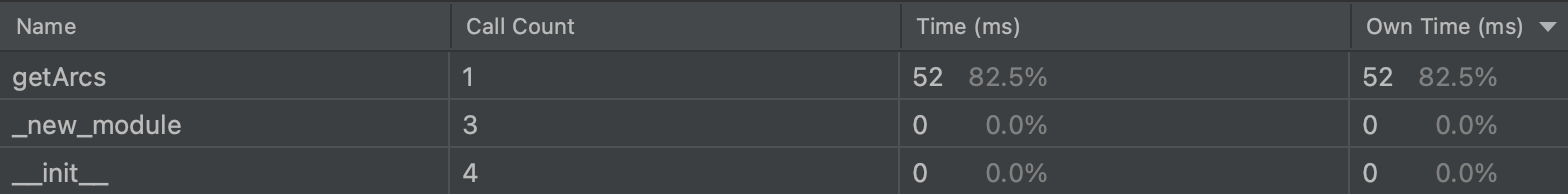
Console Output:



DFS Runtime:



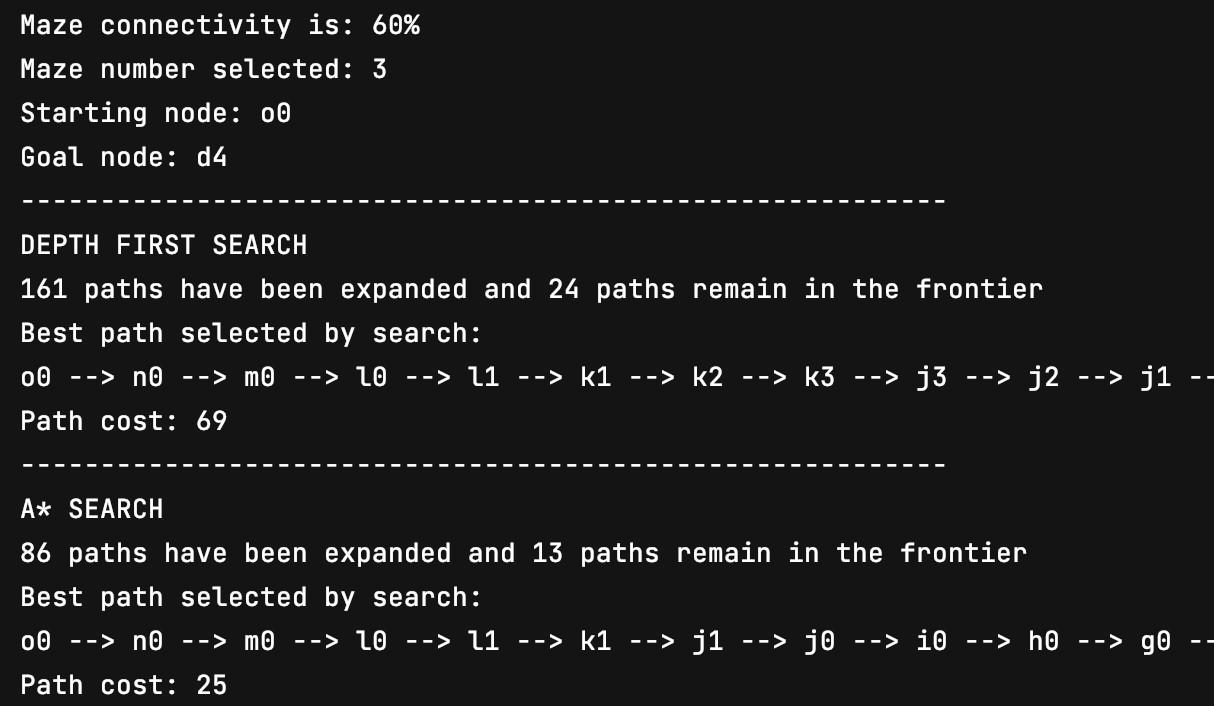
A\* Runtime:



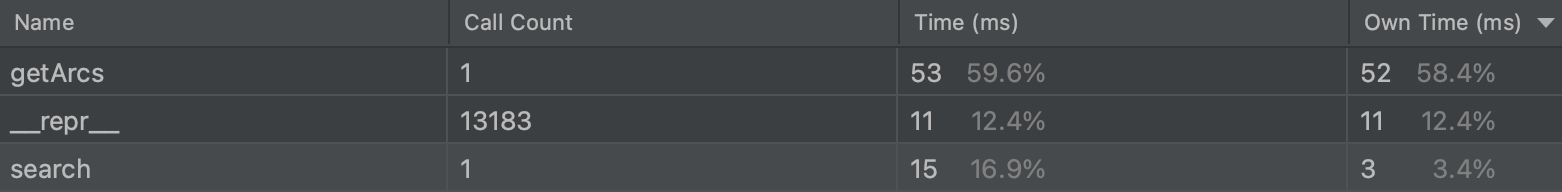
So in this example, there is little difference between the two search methods (except for the higher number of paths DFS had to expand in order to find the solution).

Now, let’s go down to a maze with 60% connectivity and see what happens:

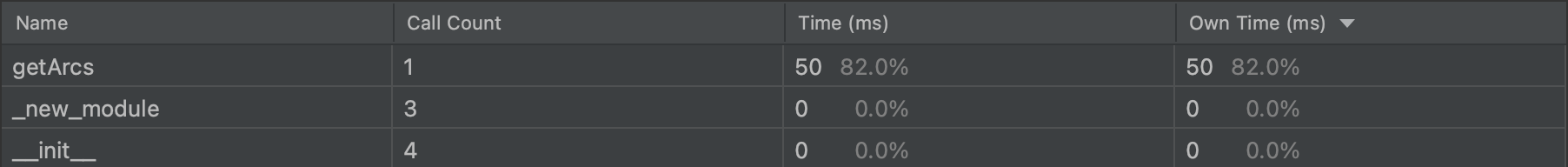
Console Output:



DFS Runtime:



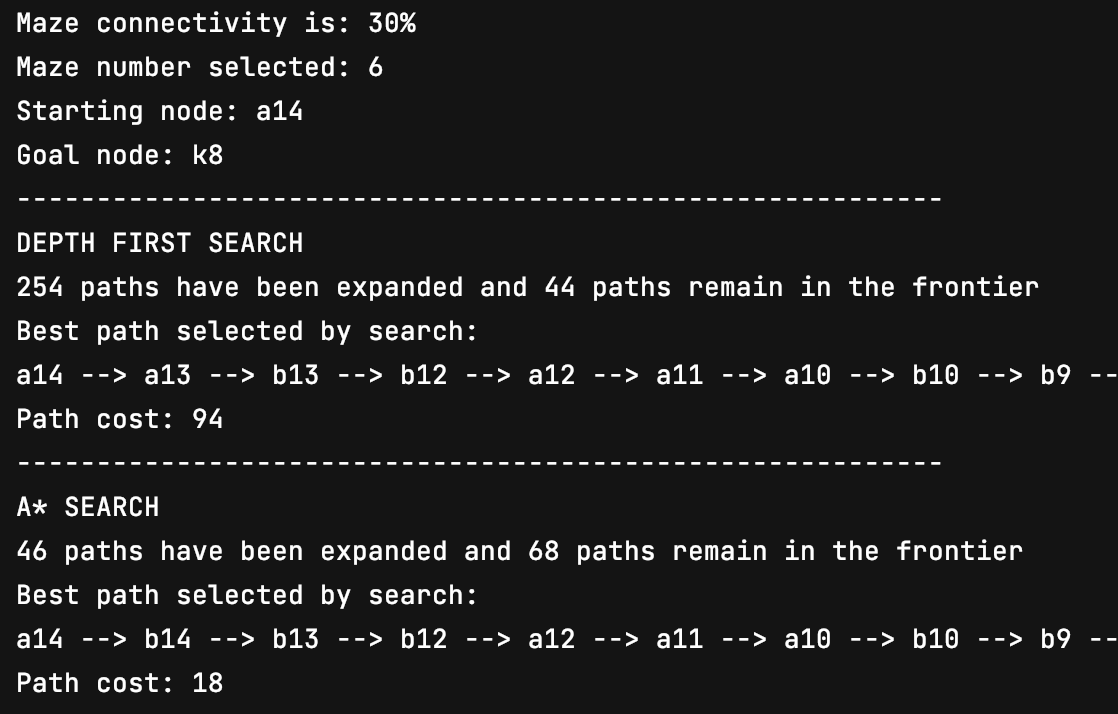
A\* Runtime:



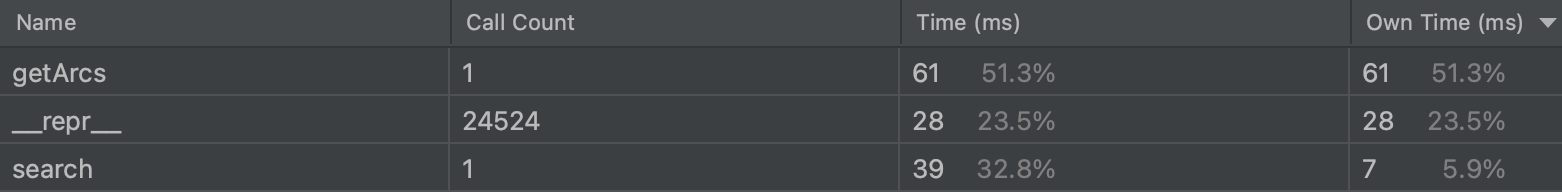
As you can see, even though the runtimes are relatively comparable (A\* still has a slight edge), the DFS chose a path that had a significantly higher cost than the path the A\* search chose, as well as expanding almost double the amount of paths. So this is a pretty big step down in performance for the DFS with a lower maze connectivity.

Next, we’ll go down another level to a maze with 30% connectivity:

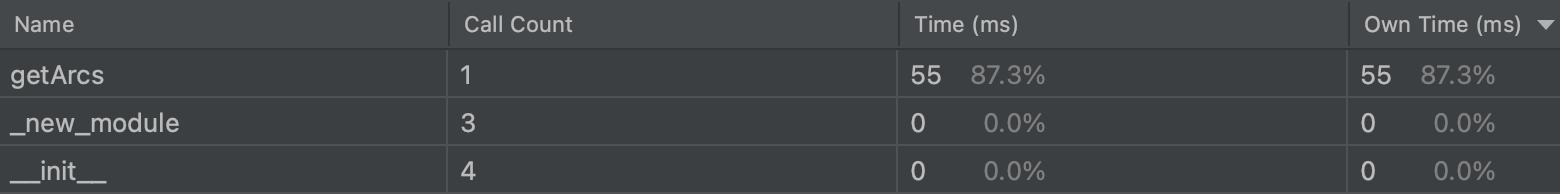
Console Output:



DFS Runtime:



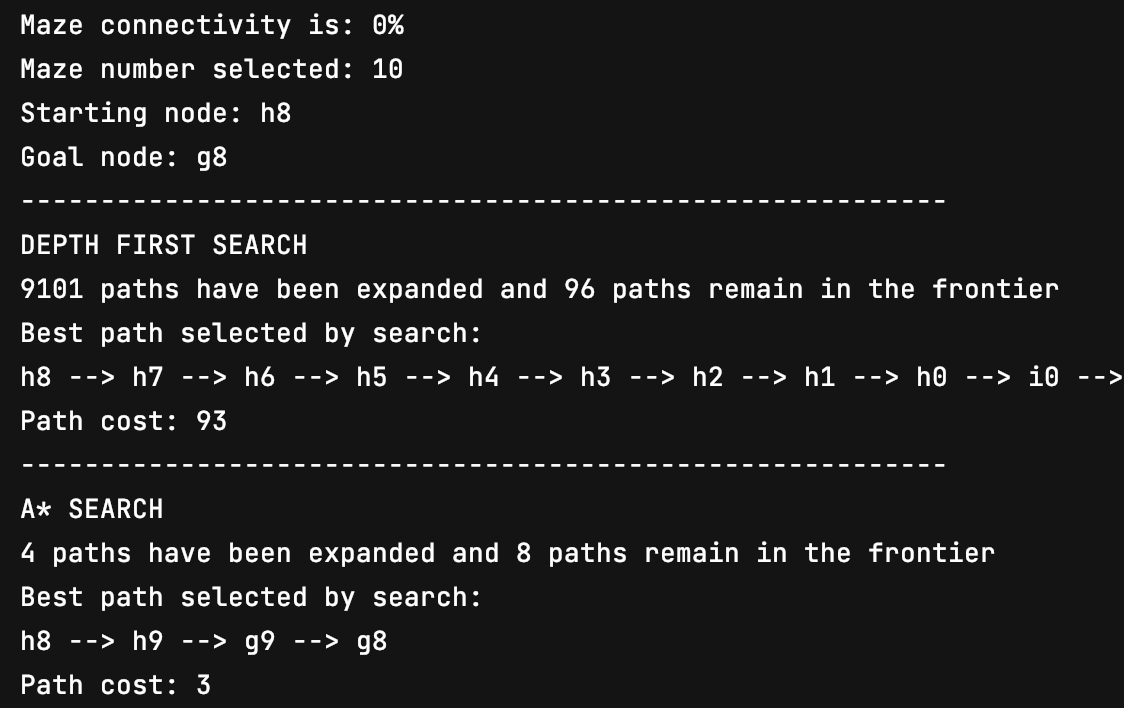
A\* Runtime:



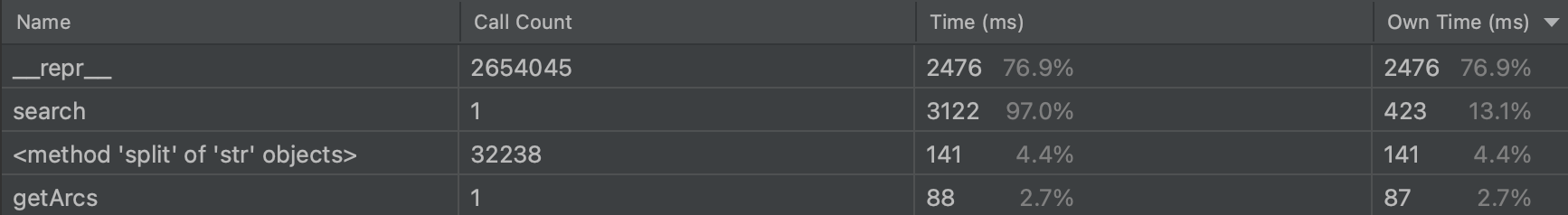
Again we see a big downgrade in the performance of the DFS. The runtime, while still taking milliseconds, is quite a bit higher than the A\* runtime now. The path selected by the DFS is over 5 times longer than the path selected by the A\* search and searched over 5 times as many paths. Now obviously, this is just one example from one maze, but from testing multiple mazes from different start points, this behaviour is typical for searches done on mazes at 30% connectivity.

Finally, we’ll look at a maze that has no connectivity at all:

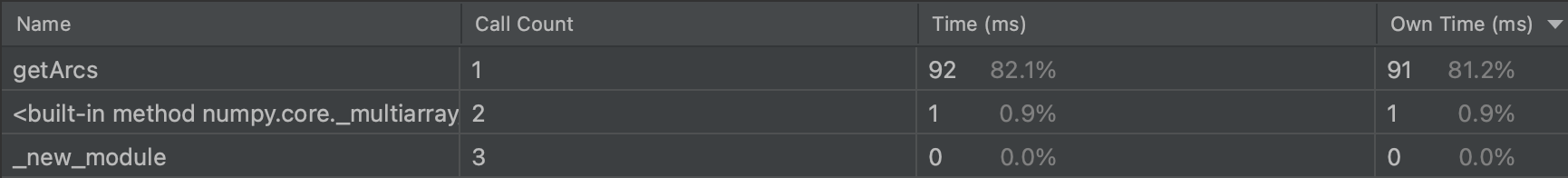
Console Output:



DFS Runtime:



A\* Runtime:



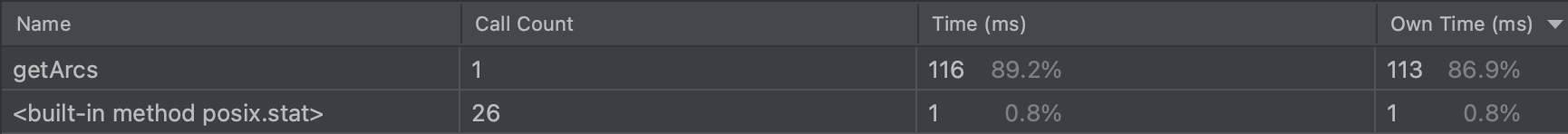
Running both searches in a maze with no connectivity produces wildly varying results. In testing, running a DFS on mazes 1,6 and 8 from any starting point resulted in runtimes in excess of 10 minutes. The DFS will always take a path to the left-hand node if it is able to. This can be seen above, where the DFS takes the path straight to the left until it hits the maze wall. So if the entrance was one node to the right of the start node, it’s entirely possible that when the DFS finally finds that path, it will be one of the very last paths searched in the entire maze. Also, it can sometimes bypass the entrance to the goal node very early on, which means it would take a very significant amount of time for the subsequent paths to be removed from the frontier before it would search from that node again.

But as we can see from the results here, the DFS did find a solution, but it look in excess of 5 seconds and required searching over 9,000 paths, resulting in a path cost of 93. But the A\* search took barely any time, only needed to search 4 paths to find the goal node and managed it with a path cost of 3.

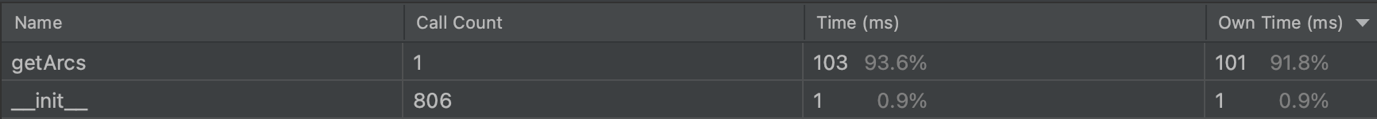
So as we’ve seen from this evidence, it is safe to assume that the lower the connectivity of the maze, the less ideal DFS becomes in terms of runtime, paths searched and path cost.

In all my testing, it was only in mazes with 100% connectivity where DFS would occasionally outperform A\* searches. For all others, A\* reigned supreme. Even on a custom example where the start node was directly to the right of the goal node, the A\* search was still faster, as seen below:

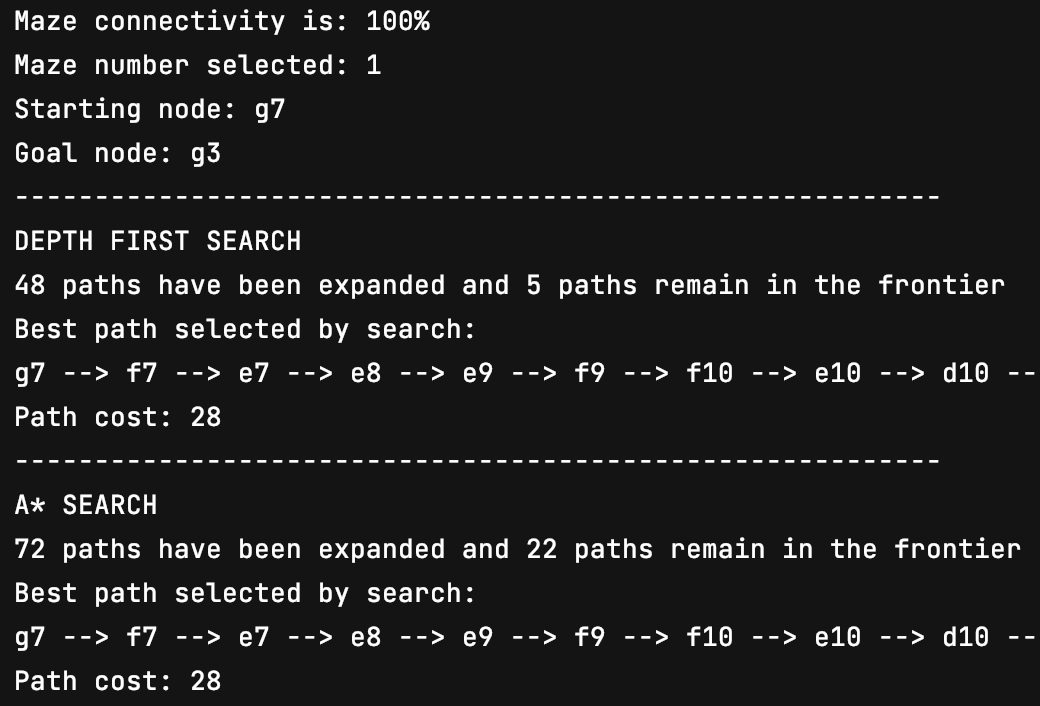
DFS Runtime:



A\* Runtime:



But in mazes of 100% connectivity, there were times where DFS did outperform A\*:



As you can see, the DFS explored less paths while still yielding a path cost equal to the A\* search. The runtimes also confirmed that DFS was slightly faster.

But these instances only happened around 20% of the time and in all other cases, the A\* search outperformed DFS in terms of runtime and number of paths explored.