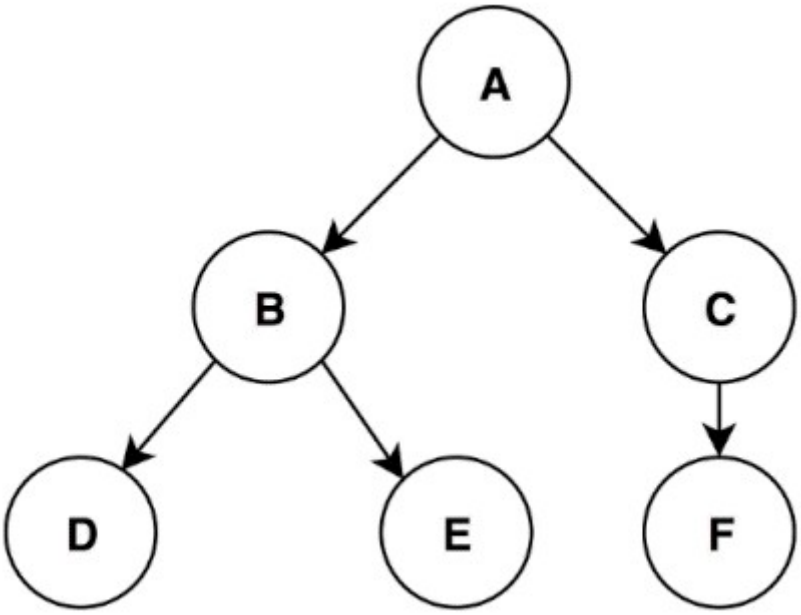
1.1) Describing How Solving A Maze Can Be A Search Problem

The maze problem can be turned into a search problem if we use a tree to represent each point we are at in the maze and each branch of the tree is a path through the maze. Each node in our tree will represent a potential legal point where we can be at each point in the maze. The leafs of that node will then represent each move we can make from the parent node's point. This will create a tree that represents all possible paths through the maze. We can then use a search algorithm on the tree to traverse through the tree to find a possible path in the tree that leads to the exit of the maze.

1.2.1) Brief Outline Of Depth-First Search

Depth-First search is a graph traversal algorithm. Depth-First will visit each node of a graph until the goal is reached or all the nodes have been visited. Depth-First will start at the root node and will see if that node has any children nodes, if so, it will visit each node left to right, but will always check if a node has any children and will always visit all the children before the siblings of the root node. Once it finds the last node with no children, it will then backtrack up the tree to the last known root node with unvisited siblings and continue checking all the nodes in the same fashion.

Consider the tree below. To traverse this tree with Depth-First we will start at A and see that A has two children nodes. B and C. We go left to right, so we will visit B next. We then see B also has two children nodes D and E. We proceed to visit D, which has no children nodes. So we backtrack up to B and then visit E. We see that E has no children nodes so we once again backtrack up to B. We have now visited all of B's children so we then backtrack up to A. We can now visit C, which has a child node of F. F has no child node so we backtrack to C, which has no more children nodes so we backtrack to A and thus we have successful traversed our graph with Depth-first search



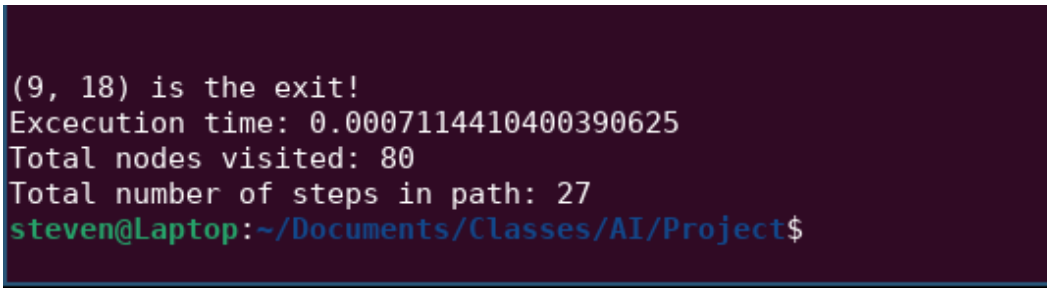
Depth First Search traversing through tree. (Source: https://dev.to/externconst/dfsubviews-5182)

The advantages of Depth-First search is that it has a time complexity of O(V), where V is the number of nodes in the tree. However, if we are using it on a graph that is not a tree, we have the time complexity of O(V+E) where V is the number of vertexes and E is the number of edges. It is also a relatively fast algorithm and is not memory intensive as we only have to store the current path through the tree. When finding a solution through a tree, Depth-First likely will not have to explore much of tree to return us a path through the tree we desire.

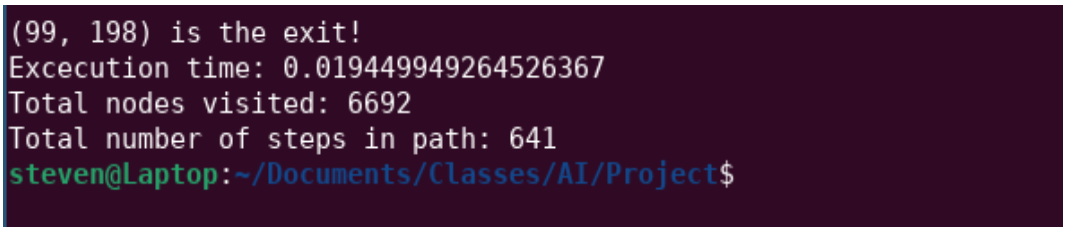
However, the disadvantage of Depth-First is that is will very likely not return us the most optimal path through the tree as it will not explore the entire tree. It is not even guaranteed to find a solution depending on what our problem is.

1.2.4) Implementation of Depth-First and Results.

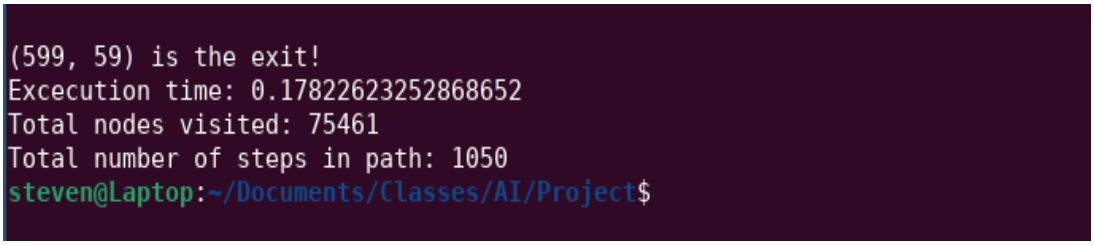
Small Maze :



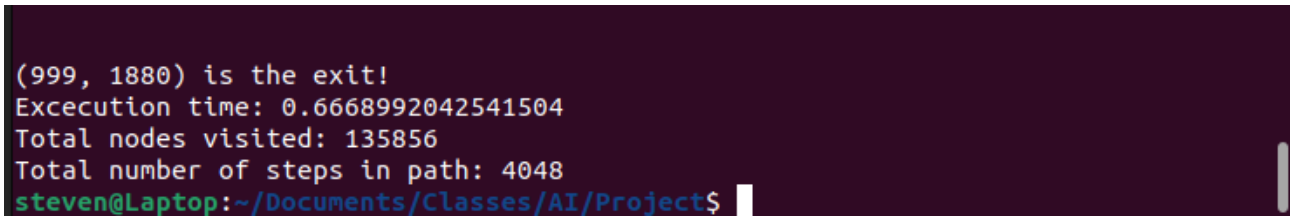
Medium Maze:



Large Maze:

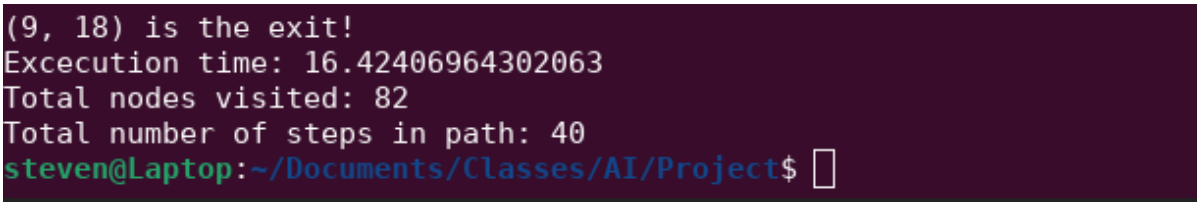


Very Large Maze:



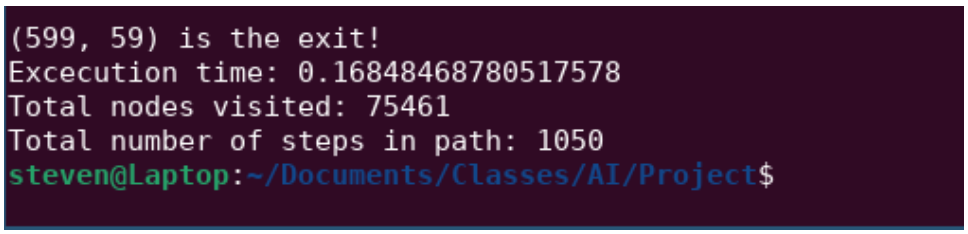
One thing to note about implementing Depth-first search is that the order we search potential moves does change the efficiency of our algorithm. At a given point in a maze, we can potentially have up to 4 possible moves. We can go up, down, left, or right, assuming that there isn't a wall in the way. When implementing Depth-first we can pick which direction we explore first given that the next move in that direction is valid. This can drastically change how optimal the solution is that Depth-first search provides us. We can see different results before as I have change the order in which the algorithm considers moves, based on what will be less optimal for our provided mazes.

Small Maze with going to the right as the first move and upwards as the second move:



Medium Maze with first move upwards.

Large Maze with first move upwards.



Very Large maze with first move upwards:

```
(999, 1880) is the exit!
Excecution time: 29.59155249595642
Total nodes visited: 779808
Total number of steps in path: 6130
steven@Laptop:~/Documents/Classes/AI/Project$
```

As we can see from these results above, changing the order of the move we look at first depending on the maze does either increase or decrease how optimal the route is that depth-first produces, but it isn't really practical to find what the most optimal moves to make for each individual maze is and change the move order for each individual maze. More importantly, this still doesn't guarantee us to receive the most optimal path through the maze.

1.3.1) Different Algorithms to Consider

While Depth-First provides us with a path through the maze, due to how Depth-First traverses trees, there is not guarantee that this path through the maze is an optimal path. Since Depth-First search searches Depth-First, it is just going to return us the very first path through the maze it finds. However, there could be several paths through the maze, where some are a lot more useful and viable than others. As seen above we can modify Depth-First to make it more likely to provide us with a more optimal path depending on the maze we have, however, this becomes annoying as we would have to study each maze individually and even if we optimize Depth-First for each individual maze, it still doesn't promise to give us the short path of the maze. And if we edit our Depth-First algorithm to always provide us with the shortest path, we will likely have to search almost the entire tree every-time so this will become incredibly time consuming for very large mazes and very memory intensive. When looking to solve our maze we want a general algorithm that we can apply to any maze of our format that will guarantee to provide us with the shortest path in an efficient manner.

One search algorithm we could consider to use is Breadth-First search. As stated above, we want to find the most optimal path, which is the shortest path from the start to the exit. This is where Breadth-first search has its advantages over Depth-first as Breadth-first is guaranteed to return the shortest path through the maze, which will be the most optimal. However, this comes with some drawbacks in comparison to Depth-first. The two big issues we will have when implementing Breadth-first search is that in comparison to Depth-first, Breadth-first uses more memory and is slower than Depth-first. This differences in these algorithms is due to how each of them search through a tree structure. Depth-first will search through a tree by searching child first, meaning if it finds valid node, it will keep searching that node's children and the children's children and so on… until either there are no more valid nodes and it starts backtracking or a path through the maze it found. Comparatively, Breadth-first will visit all the siblings nodes before visiting the children. This means even if Breadth-First finds a valid node to continue searching it will still search all of its sibling nodes to see if any of them are valid nodes before going on to search the valid nodes' children. This is how Breadth-first is able to guarantee it will return the shortest path, while Depth-first will only return a path. But obviously, we can see the how this causes drawbacks. Breadth-first will have to search much more of the tree to guarantee that it is returning the shortest, which means it will likely be much slower than Depth-first who will search much less nodes in the tree. Breadth-

first will also have memory drawbacks in comparison to Depth-first since Breadth-first will need to store multiple paths from the root node to the current node, while Depth-first will only need to store one path from the root node to the current node. This means Breadth-first will use up considerably more memory than Depth-first search.

While, Breadth-first search gives us a more preferred solution it doesn't come without its drawbacks and depending on how large our maze becomes, although unlikely, it may not be feasible to use Breadth-first due to time and memory constraints. However, despite the speed and memory usage, Depth-first search isn't very appealing as a solution either because it is very likely it isn't the most optimal solution. What we want is an algorithm that provides us with the best of both. We want the most optimal path through the maze, but we want it done efficiently. For this reason we are going to explore a search algorithm called A* search.

A* search is a graph traversal algorithm that will provide us with the shortest path, which is what we want, however, it will do so efficiently, so we don't get the drawbacks of Breadth-first search. This is achieved by implementing a heuristic function which will aid it in deciding which nodes to explore next in its search. A* search can still potentially explore nodes that lead to no solution, with the help of a heuristic function, we can hopefully reduce the amount of nodes we have to explore to find the most optimal path through the maze, thus increasing the speed of the algorithm compared to Breadth-first.

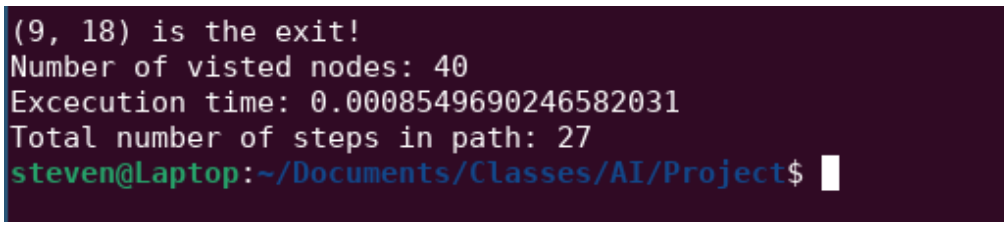
What A* search does is that when searching our graph, it will assign a cost to each of the nodes and will select the path that has the minimum total cost to reach from the start node to the end node. The cost function that A* search will use is F(n) = G(n) + H(n) where G(n) is the cost that it takes to reach the next node from our starting node. H(n) is our heuristic function, which is the cost it takes to reach the end node from our node as efficiently as possible. Adding these two functions will give us F(n) for each node, which is the cost that it takes to reach the end node. A* search will go through our graph and select the path that is that is the most efficient, meaning we want to minimize F(n).

This makes our search much more efficient than that of Breadth-first search, while still providing us with the most optimal path through the maze. The one drawback of using A* search is that it is memory intensive. This is because like Breadth-first, A* needs to save all the paths from the root to the current node.

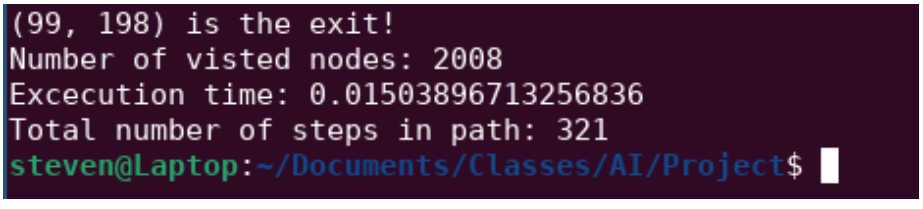
We can see the results of A* below:

1.3.3) Implementation of Suggested Algorithm and Results

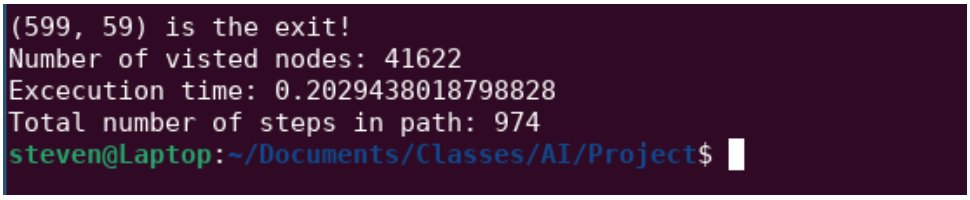
Small Maze:



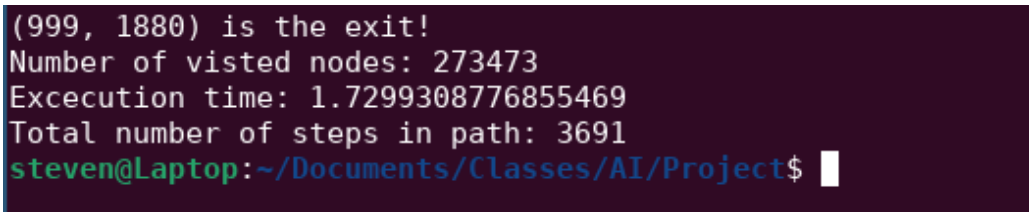
Medium Maze:



Large Maze:



Very Large Maze:



With the result above we can already see a big improvement in our results. We have been given much shorter paths through the maze and A* is doing so as quickly as our Depth-First algorithm was. As expected, as our maze grows larger and larger the time to compute the shortest path and the amount of nodes we have to visit to do so is growing exponentially. This could cause problems if our maze becomes to large as it may become too computationally expensive to compute.

It is also worth noting that A*'s performance it heavily dependent on how good the heuristic function is. In this implementation of A*, I have used a very well known heuristic called Manhattan distance. This is the sum of the absolute value of the difference of both the coordinates. In this case, it seems we have chosen a good heuristic that has made the algorithm perform much better than Depth-First.

1.3.4) Performance Versus Depth-first Search

As we can see from the results of running both algorithms on the 4 mazes, A* search provides us with a much more optimal solution in 3 out of 4 and the first maze, Depth-first and A* provided the optimal solution , however, A* did so in half the nodes visited. We can see in both medium and large mazes that that not only does A* provide us with a much more optimal path in comparison to Depth-first, it also did so in much less time and visiting much less nodes than Depth-first. On the very large maze, once again, we see we have a much more optimal solution with A*, but it does take longer and does visit more than twice the number of nodes than Depth-first search to find this path.

This could be because the very large maze is most optimally suited for Depth-first to find a path very quickly in its search, thus it doesn't need to search much of the tree to find a path through. This is likely because as the maze gets larger and larger, the longer A* will have to search to make sure it will provide us with the most optimal path.

Since A* search is looking to provide us with the most optimal path it is expected that as the maze continues to grow larger and larger, A* will likely start to take longer than Depth-First search will. We can see this with the difference in the time complexity of the two algorithms With A*'s time complexity being O(b^d). Where b is the branching factor of the tree and d is the depth of the goal node. Depth-First's worst case time complexity is O(bd). Thus, it is expected that on large mazes,

Depth-First will still have a speed advantage over A*, but A* will likely provide a path through the maze that is more and more optimal as the maze grows larger and larger.