Analyzing pathfinding algorithms on a Two-Dimensional Maze

Abstract:

Pathfinding algorithms have been a topic of wide discussions in the field of computer science. They have used widely to solve real-world issues of finding and optimizing path between two places, and have countless applications. To better understand this problem, I used various data structures as the core of finding cells to reach out to, to find the actual path between a provided start and end cell. I used Algorithms such as Depth First Search, Breadth First Search and A*, implemented using Stacks, Queues, Heaps and Priority Queues. My key findings were that having knowledge of start and end points greatly improved the efficiency, and the other important finding was the existence and usage of better heuristic functions to solve our problems quicker.

Results:

For all of the portion of the results, I have set the grid size to be **30*30** with varying density, most of the data presented here are graphs and charts.

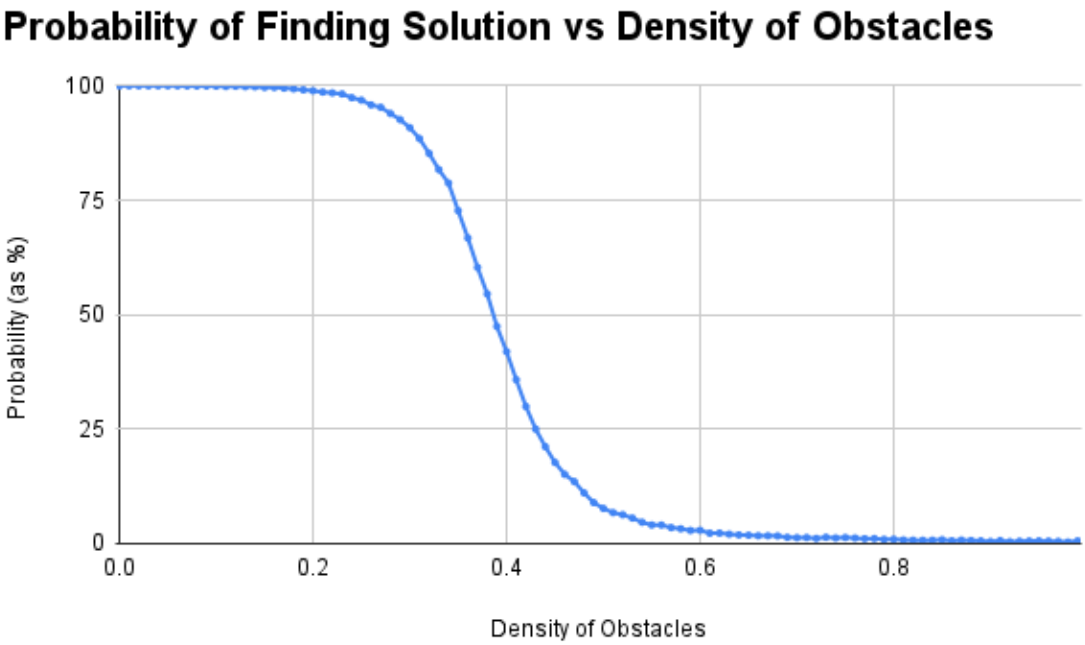
For all of the simulation, I ran DFS, BFS and A* algorithms 10000. DFS went along a single path until it reached the target, if locked it went back to last traversable state. BFS explored once in every direction it could until it reached the target.

For my A* algorithm however, I used the Manhattan distance between given cell and target as my heuristic function, alongside the number of cells required to reach the given cell. In my extensions, I will discuss how I improved this function for a much faster and efficient algorithm.

To run these results, run the Exploration.java file.

Exploration 1:

This runs a simulation of 10000 mazes across densities from 0.00 to 0.99 trying to find the solution to the maze. Since the existence of path does not depend on the algorithm used, I implemented A* for quick results. Here, I present the data obtained as a line graph:

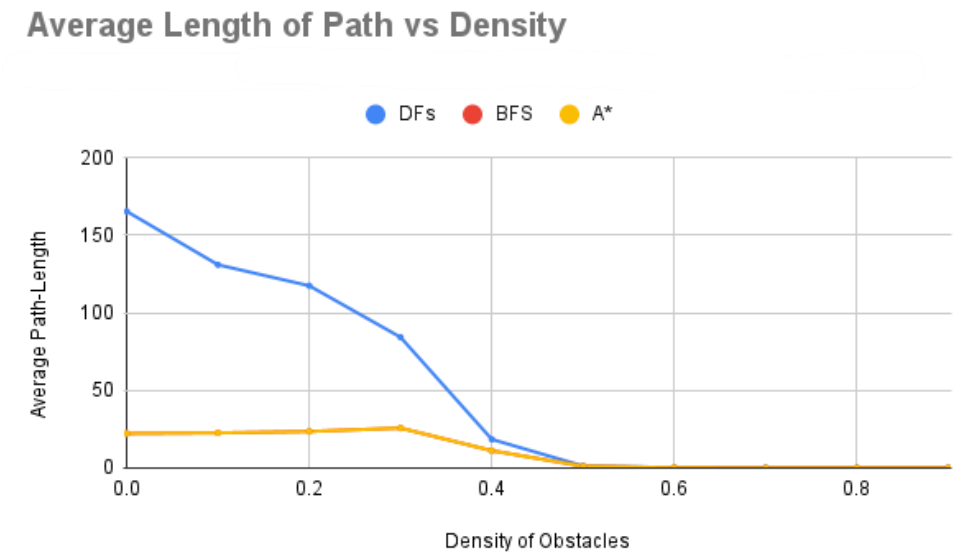


We can see that the probability starts plummeting around 0.3 and is almost 0 around the 0.6 mark. In fact, most mazes are not solvable around 0.5 but since I am using a large simulation, some solutions were found due to randomness. This graph also very closely resembles what looks like a sigmoid function due to the similar curves around 0.3 and 0.5 mark.

Exploration 2:

(Once again, the results from A* were suboptimal which I improved in the extensions later for consistency)

Here, I simply calculated length of ArrayLists containing the Paths from start to target.

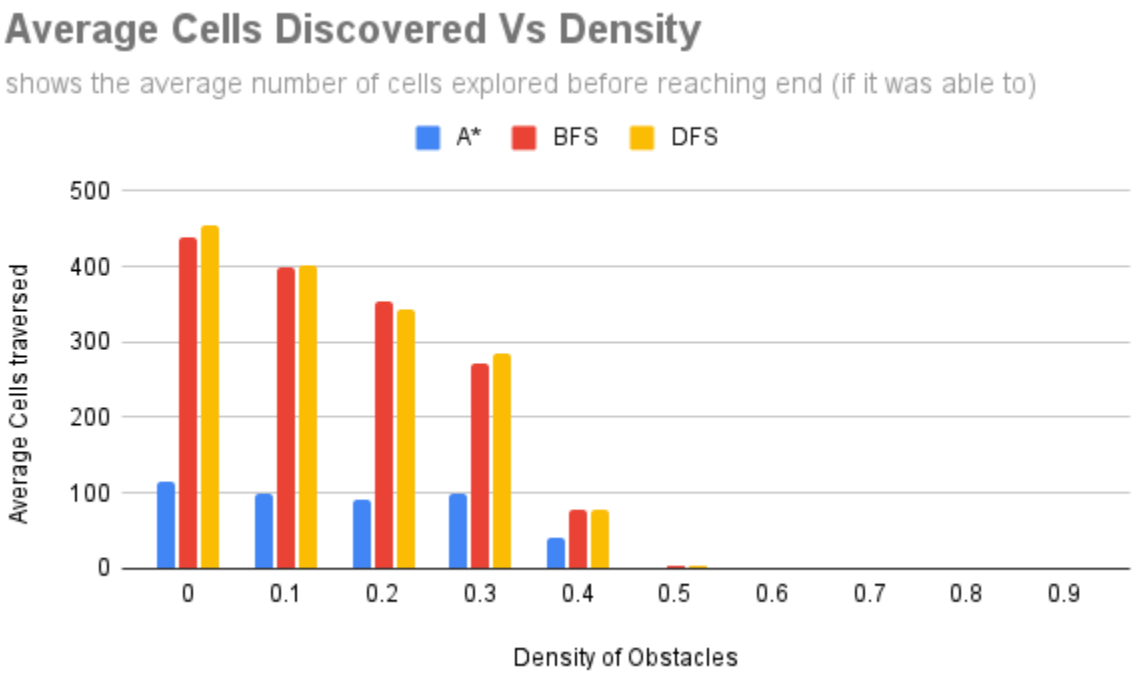


BFS and A* always give the shortest path between start and finish, due to which they appear as overlapping in the graph. For smaller densities, path found by DFS is extremely large compared to them. But in the longer run, their efficiency doesn't matter as they drop down to 0 very quickly since they are no longer able to find solutions to the maze.

Generally, path found by DFS > BFS = A*

Exploration 3:

To find the number of cells explored by each algorithm, after the maze was analyzed, I just counted the number of cells which had a previous cell.



Even for the suboptimal A*, it had a very high efficiency compared to DFS and BFS.

Surprisingly, on average the number of cells explored by DFS and BFS were very close which is ironic considering their completely different ways to search through the maze.

Once again, on increasing the density of obstacles, the number of cells discovered before finding the target went down. **This does not necessarily mean the algorithms perform better with higher densities, rather it could be from the fact that there were less cells that we could even explore.**

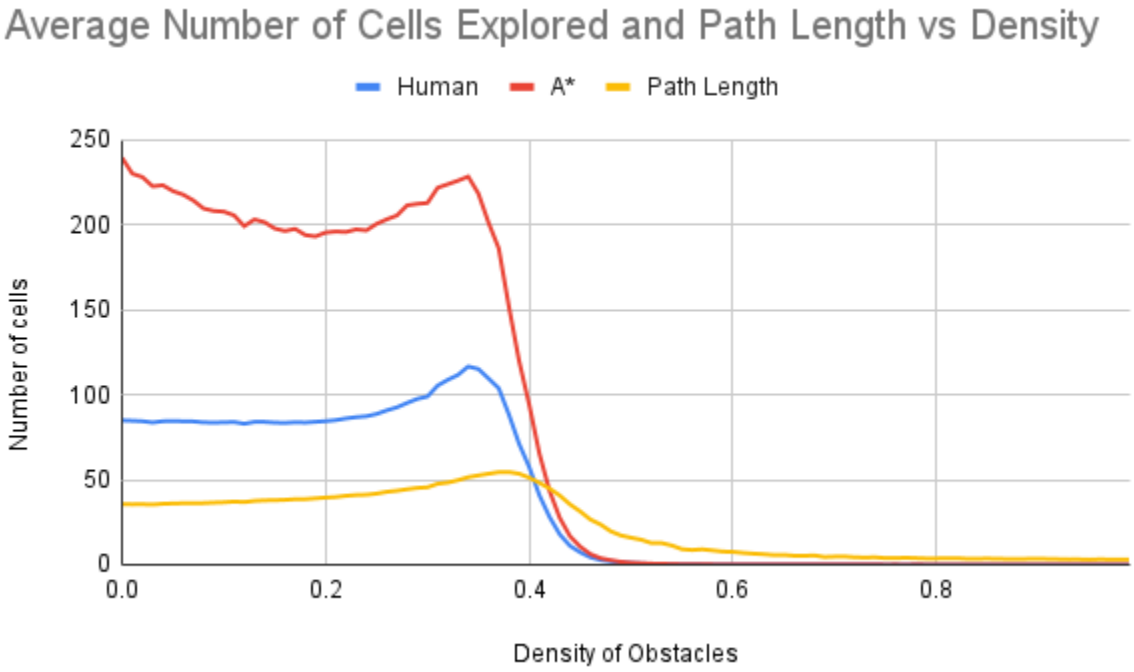
Similar to the previous case, once the densities start hitting the critical point, the number of cells needed to explore drops down since we can quickly tell if a solution exists for the given maze.

Extensions:

The entire portion of the extensions uses a size of maze 50*50 for even accurate representations. To get these results, run the **MazeHumanSearch.java main method**

For my extensions, I decided to form an algorithm as a combination of A* along with a human touch.

I implemented the concept of good and bad neighbors. Basically, a neighboring cell is good if it has more free cells neighboring it than obstacles. Additionally, I decided to use Euclidean distance over Manhattan just out of curiosity. Adding a component to the heuristic function of A* gave a slightly better result for large number of simulations. I also included 5* weight for Euclidean distance and half of that to number of neighbors, which I will discuss later.



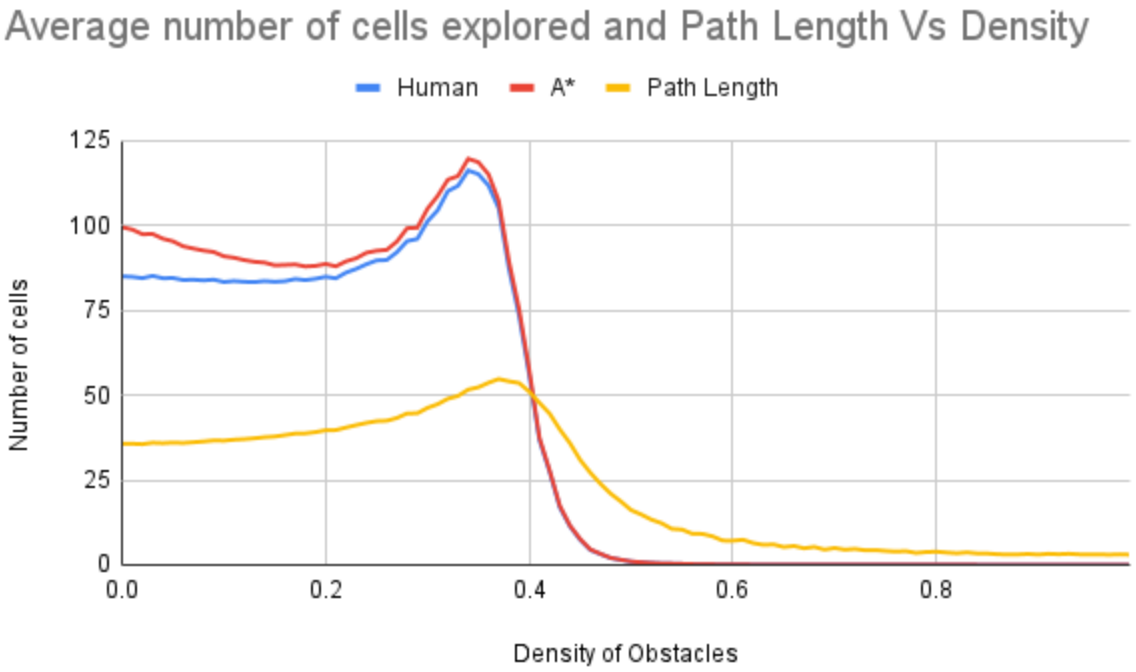
Here, we can observe a suspiciously large improvement of the Human Search from A*. This led me to fix the A* algorithm which I will discuss later.

We can see that the algorithms' cells searching dive at a certain number of obstacles and later grow in size until they have difficulty finding the solutions, after which everything drops to 0.

As predicted earlier, the early dip could be due to having less cells to explore (or rather move to unnecessarily), but the proof for that is beyond the scope of this project.

Fixing the amateur A* implementation using weight factors

I realized that having the same weight factor for the Manhattan distance and traceback for a given cell did not result in good performance. Thus, following my human search implementation, I decided to have 5 times for weight for estimated distance. This led to a more efficient solution:



After including the weight factors, the initial A* algorithm started performing almost as good as the human intuitive search. But, for initial densities, human search was better. Their difference went down for larger values of densities and following the pattern, after 0.5 everything dropped to 0. Also notice how these graphs are exactly identical but A* is scaled down by almost 2.5.

Acknowledgements:

In completing this project, I consulted with Max and Allen, but did not look at any code. I also looked up A* algorithms videos from YouTube channels Polylog and Reducible. For extension ideas, I consulted Claude sonnet from Anthropic.