Introduction to Artificial Intelligence

Assignment 1 - Maze Runner

Group: Tatsat Vyas(thv1), Adarsh Gogineni(sg1255), Andy Guo (ag1394)

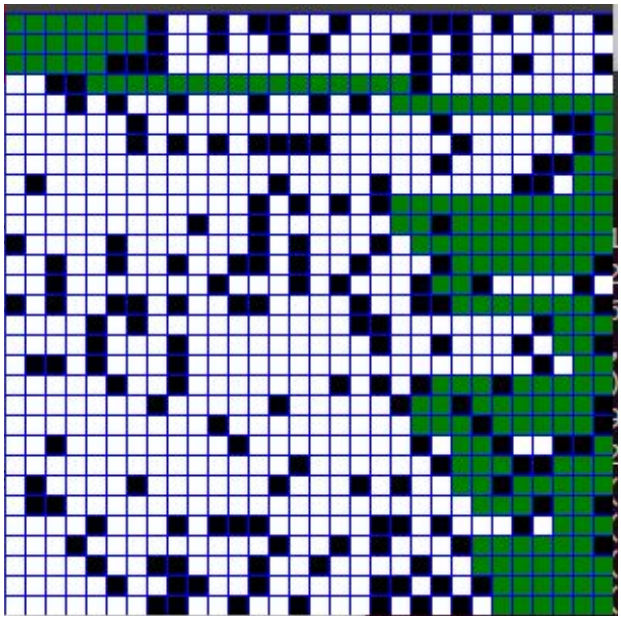
Introduction

This project requires implementing file different Graph searching algorithms. The algorithms are BFS, DFS, Bidirectional BFS, A* Euclidian distance, and A* manhattan distance. We have used python for our choice of programing language because of the simplicity of the language as well as the graphics library. We made a simple chess board like maze(meaning black and white) and colored the blocks that were blocked. More information about the algorithm and the project is in particular sections.

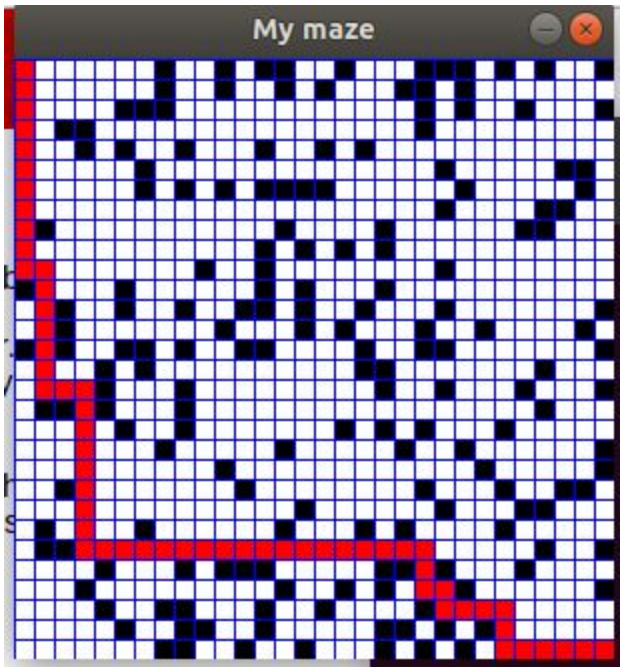
Part 2

1) The Map size that we found that was perfect was 30. This is small enough that it requires each algorithm some wok to solve but large enough to see the absolute probability.

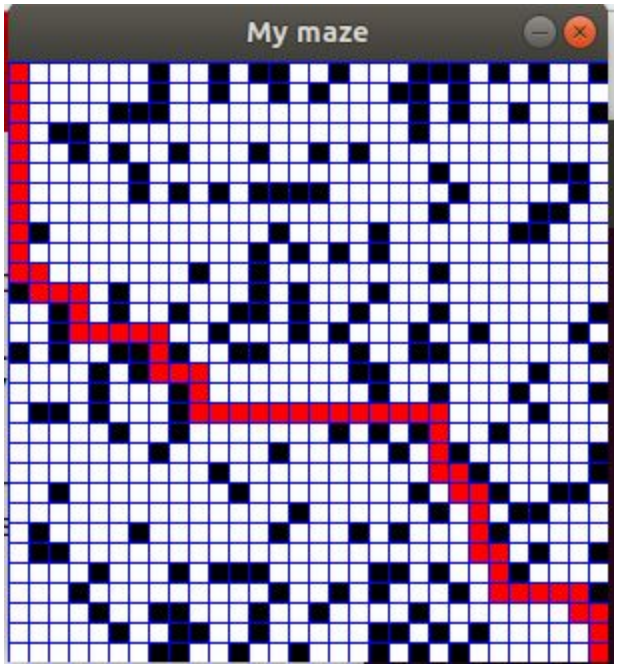
2) The path the algorithms took with the randomly generated maze is shown below.



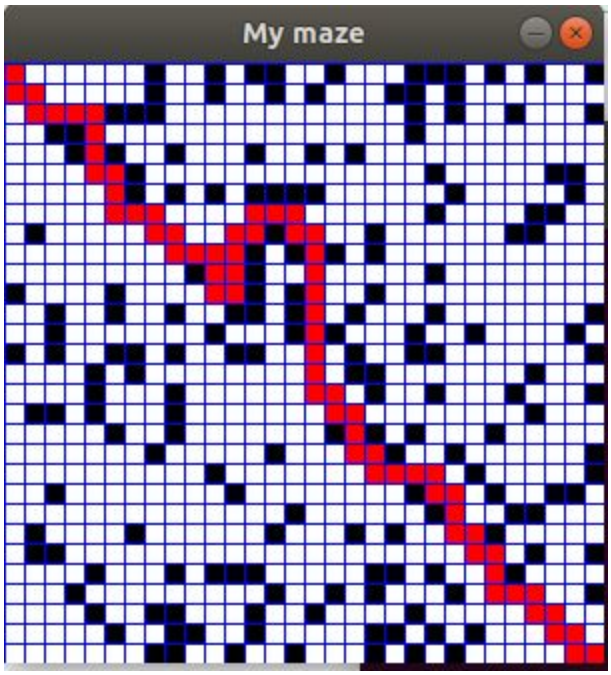
This is the return path of **DFS,** because DFS only cares about finding if the path exists we showed all the nodes that it visits in order to do it. As you can see it makes a random choice depending on the state of the neighbors. If more than one neighbor is open it makes the choice randomly.



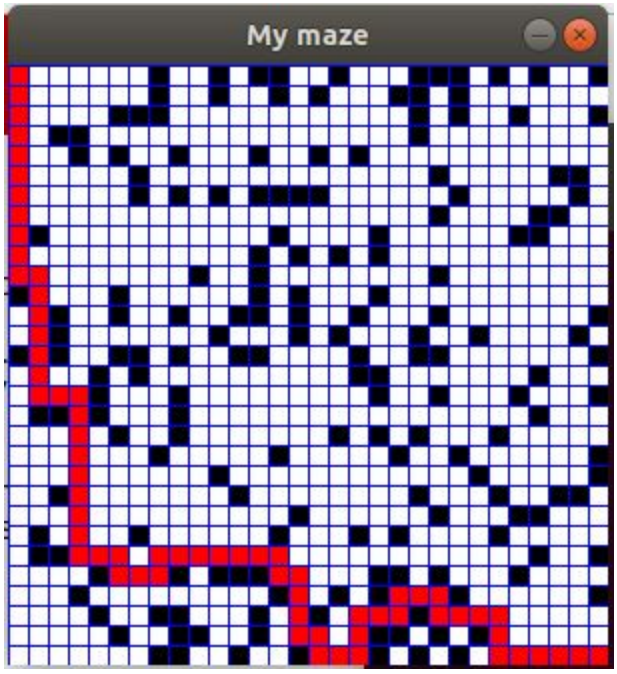
This is the return path for BSF. As you can see it is much better looking than DFS. the way we designed the code is it keeps exploring neighbors until it hits the destination and then it basically backtracks the nodes. We used a queue representing a fringe and we would add the neighbors to the fringe each time we explore the node. This makes sure that it finds the shortest path. It stops as soon as it encounters the destination node(in this case the last node [n-1][n-1]). For backtracking, we have a class that stores each of the nodes and each node have a reference to the previous nodes. For coloring, we basically make the final node and backtrack from it.



This is the output of the bidirectional BFS, as you can see it is taking a different path than the regular BFS. That is because we have BFS running from the destination and the start. When we stop the algorithm whenever they both meet. This still makes sure the shortest path but has a faster run time. When you run BFS and bidirectional BFS together. You can notice the time complexity different when the value of N is large enough.

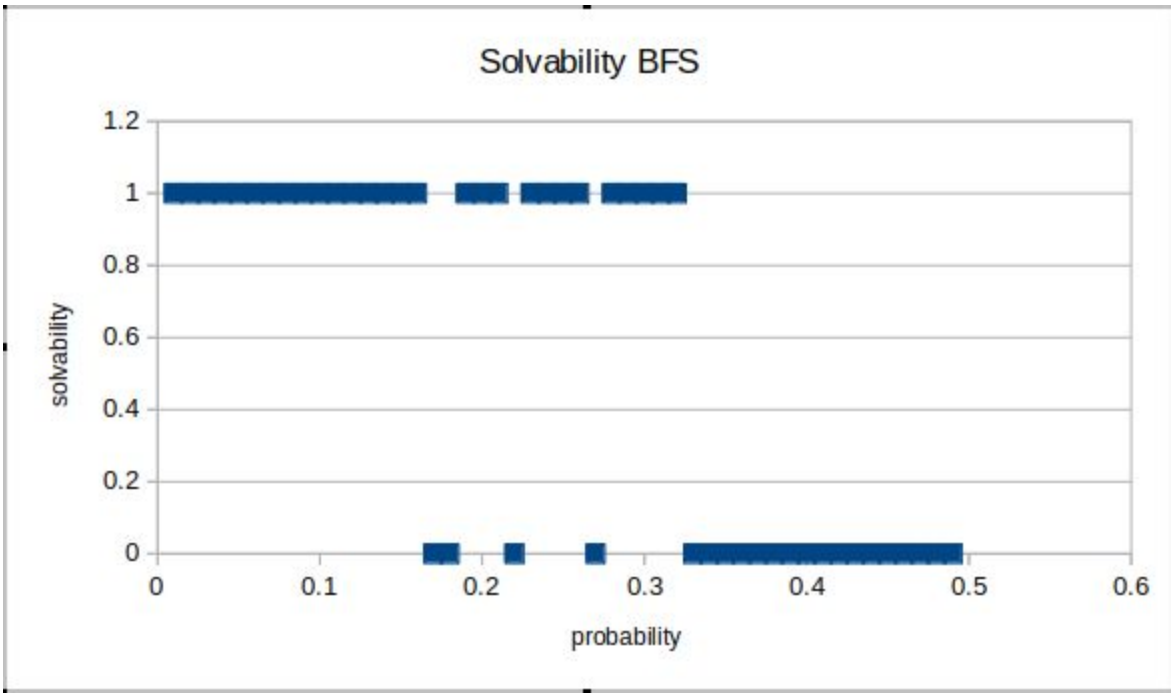


This is the maze generated when using A-star with euclidian distance heuristics. It basically has the same strategy of exploring neighbors but it has another value called heuristics for each node which decides what node the algorithm will explore next. This is not the shortest path as you can see from the rectangle made in the middle because A-start does not explore all the nodes, it saves time and space by exploring the nodes with the smallest estimated value. In our case, however, two blocks can have the same heuristic values(down and right). In this case the algorithm picks the next block to explore by when it was added to the fringe. This results in the rectangle phenomenon you see in the maze.



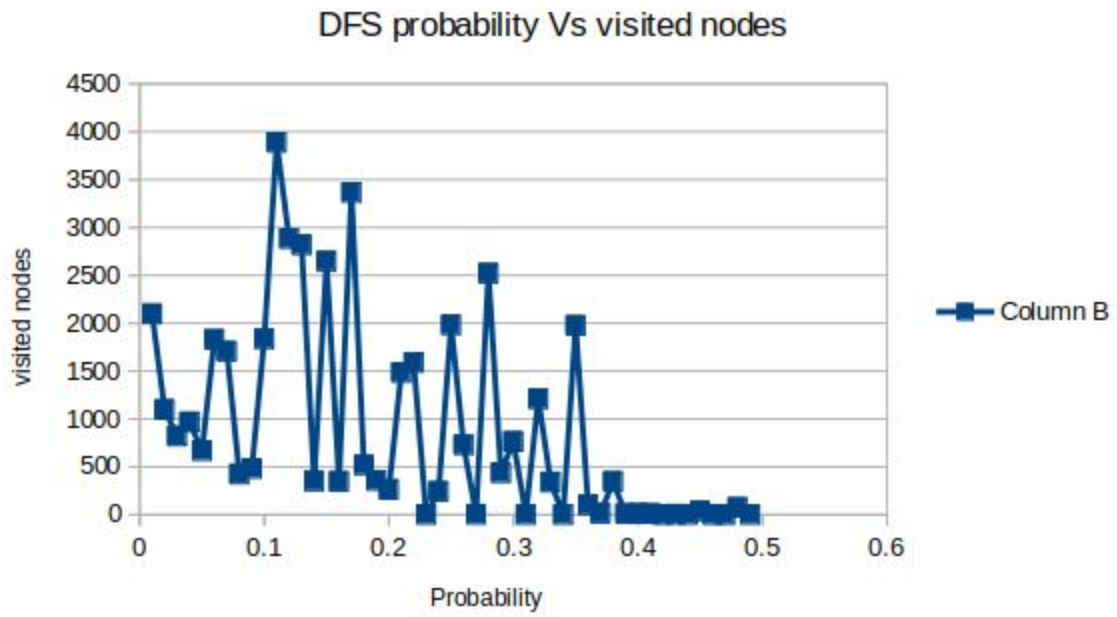
This is the maze generated when using A-star with manhattan distance. As you can see, it has a different path than the one before. The only difference in the code is the way it calculates the heuristics.

3)



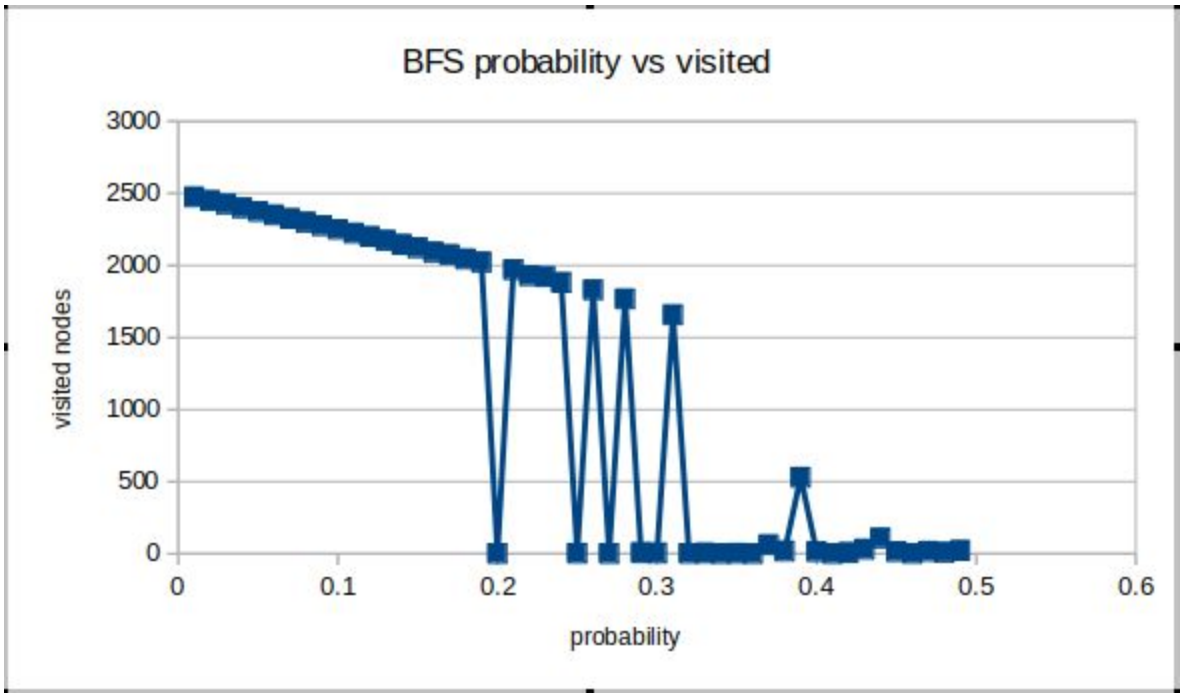
The above graph shows the solvability of the randomly generated maze. As you can see when the probability gets to a certain threshold point, the maze becomes unsolvable. We found this threshold probability of $p' \cong 0.34$. We found this threshold probability by generating multiple mazes and checking each for an estimated threshold. We also used each algorithm on the range of probability and measured the visited nodes. We kept the N value to 50 in order to generate a 50x50 maze. This was our way to see the performance of each algorithm as the probability increases. The graph of probability vs the nodes visited is shown below for each algorithm.

DFS



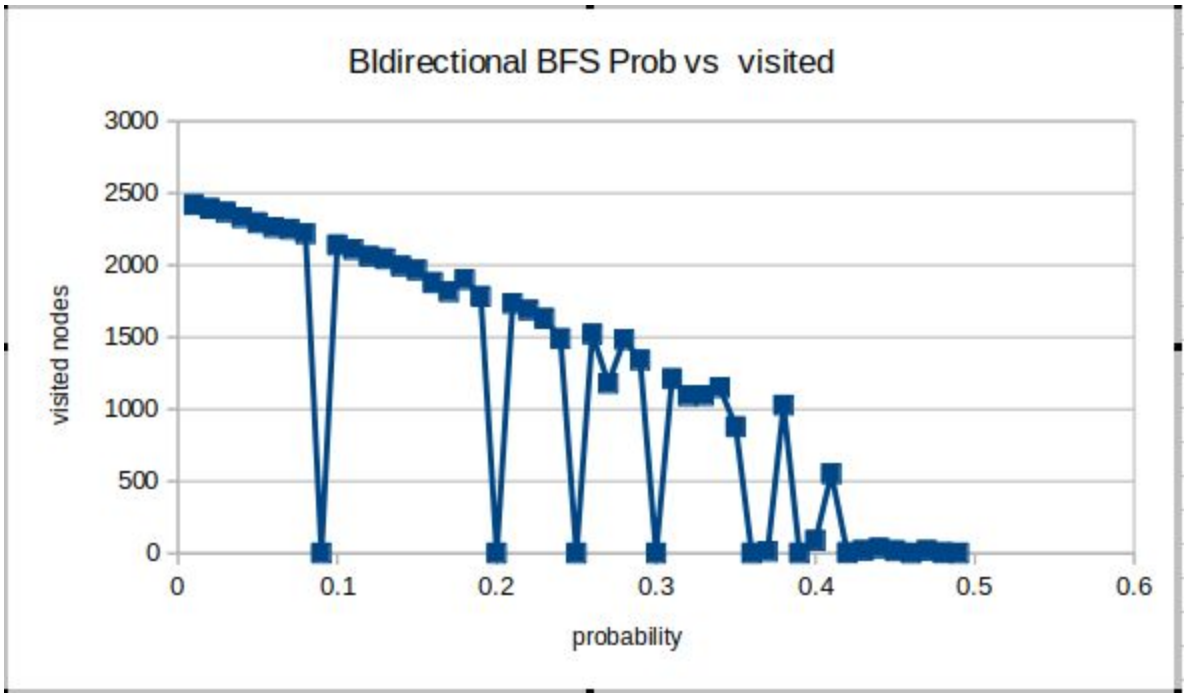
As you can see the performance of DFS seems to be very bad for low probabilities. This is when the maze is solvable. This makes sense because DFS is mainly used to identify if the path exists or not.

BFS



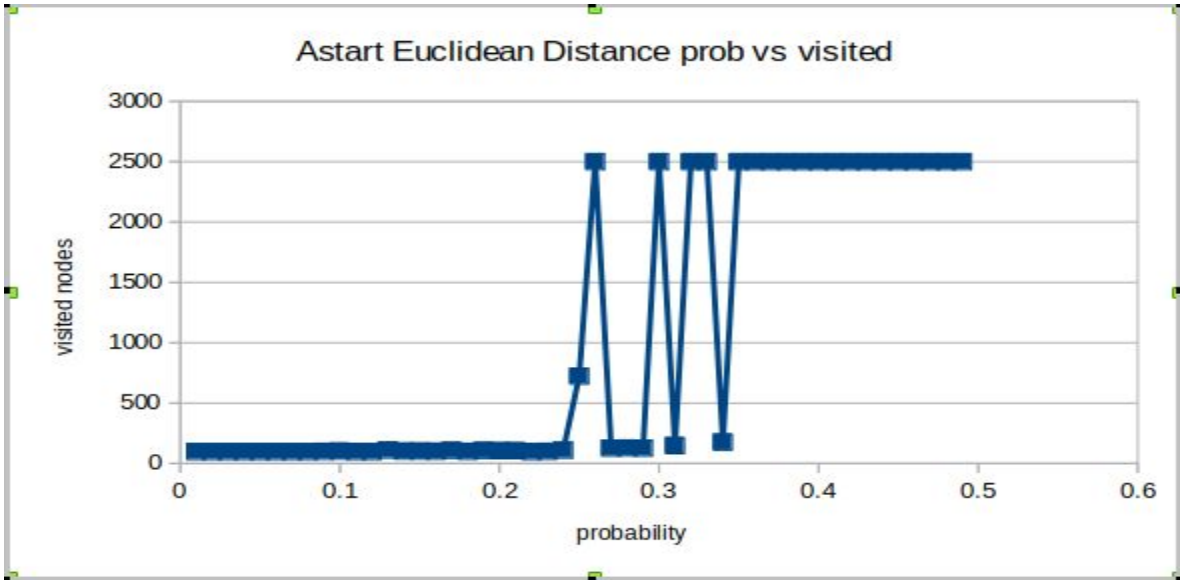
As you can see with very small probability, BFS visits almost every single node. This is not good for the run time since it will check neighbors for every single node.

Bidirectional BFS



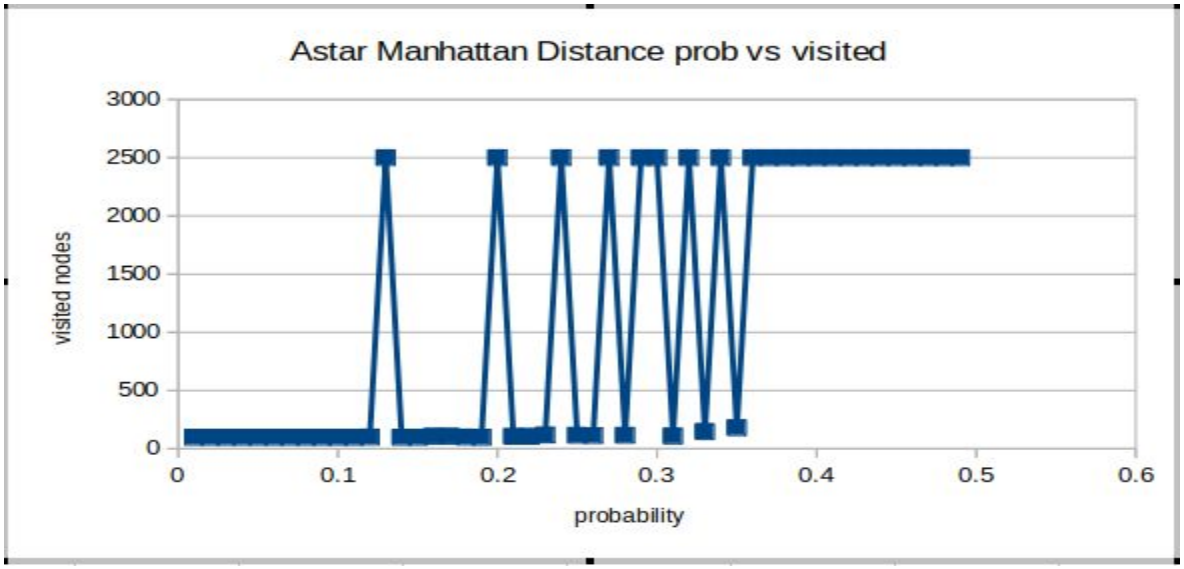
The above graph is for bidirectional BFS, it performs better than BFS since it starts running from both ends. The algorithm is faster, however, the combined visited nodes are just as many as the regular BFS.

A-star Euclidean Distance



As you can see from the above graph, A-star performs completely different. The maze that is solvable, A-start is very fast. This is because A-start makes a decision at each step by looking at the heuristics. When maze probability gets higher, however, A-star has a hard time finding if the maze is solvable or not.

A-star Manhattan Distance

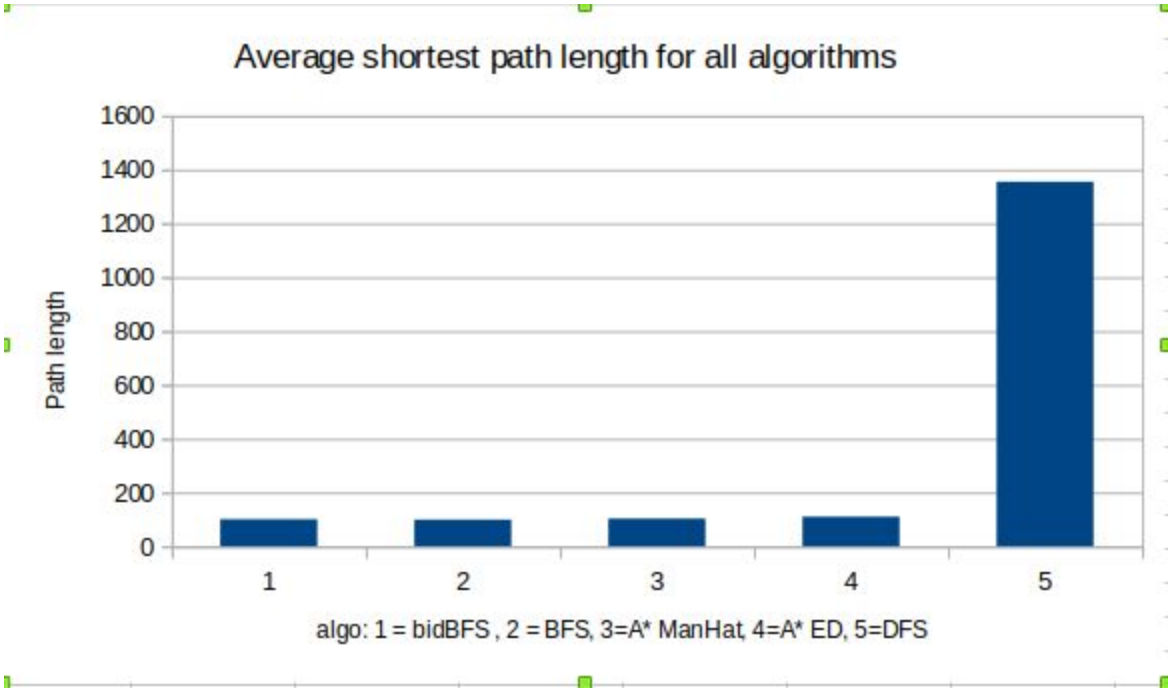


A-star using manhattan distance has a similar but slightly better performance than the euclidean distance. This is because the design of the algorithm is very similar and the way we are restricted in that(only going up/down/left/right) it is better suitable for manhattan distance.

Conclusion:

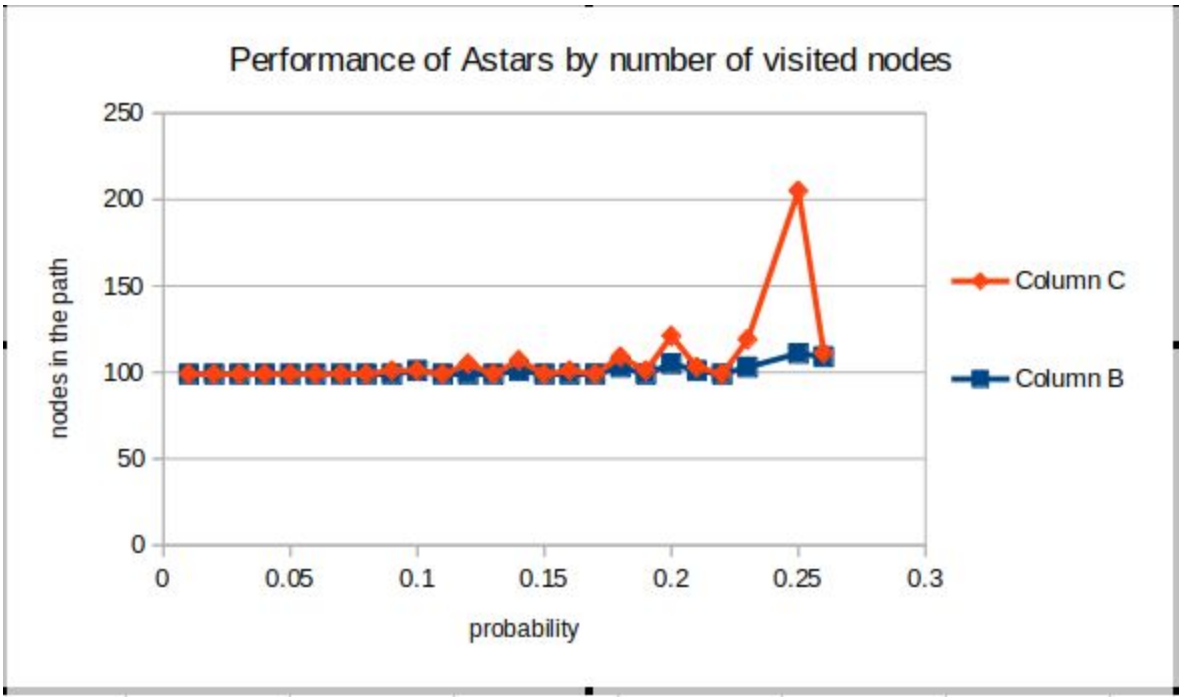
The best Algorithm to use here is the A-star using manhattan distance because the overall performance of the algorithm is good for $p < p'$.

4)



The above graph is the average shortest path for each algorithm. As you can see DFS performs the worst and BFS performs the best for the shortest path. This makes sense since BFS is guaranteed to find the shortest path each time. The Data was taken over increasing probability(**increasing Density**) so This also caused the other algorithms to perform worst. The reason that the expected shortest path values are not as far apart is that the randomly generated maze only gets difficult up to a certain point before becoming unsolvable. In order to make it more difficult a manual input is required.

5)

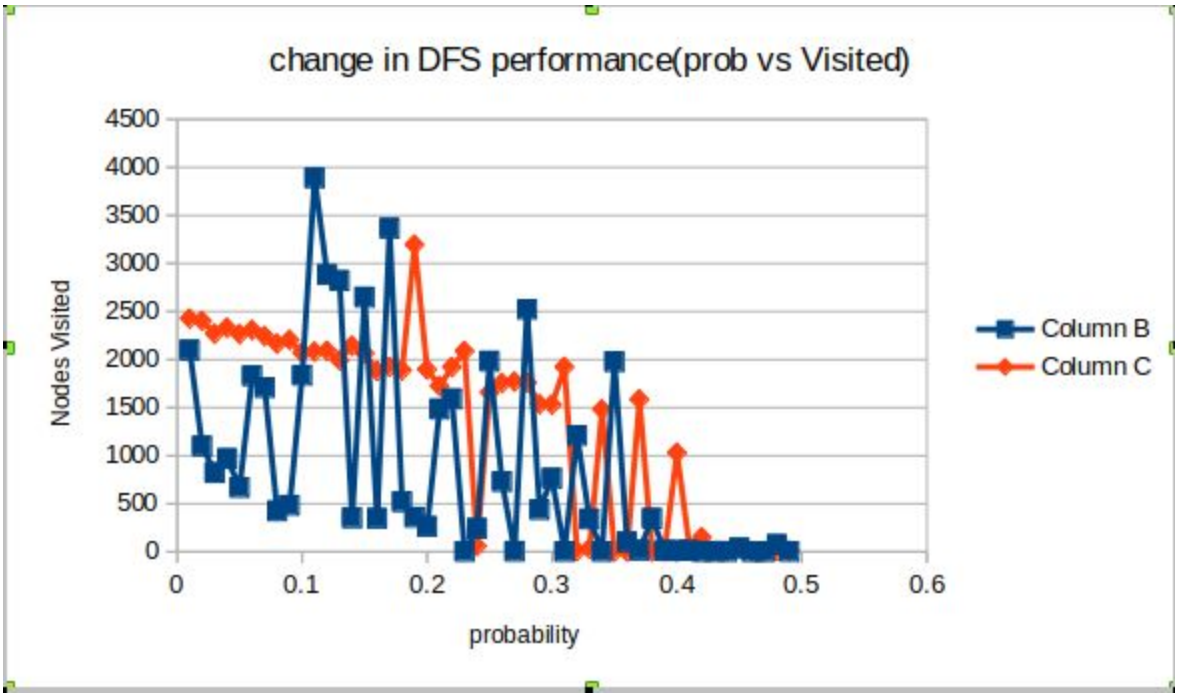


Column C = A-star Euclidian Distance , Column B = A-star Manhattan Distance

As you can see from the above graph the manhattan distance has a sightly better overall performance.

6) Yes, The algorithms do seem to behave as they should on average. A-star euclidian distance can perform better if the diagonal path was permitted to be utilized. BIdirectional BFS seem to be a lot faster than both DFS and BFS as expected.

7)



Column C = right/down/left/up , column B = left/right/up/down

As you can see the way you load neighbors to the fringe does affect the performance of the algorithm. The overall performance of the algorithm seems to be better on the blue(column B) curve. It has some outliers but overall performance is much better.

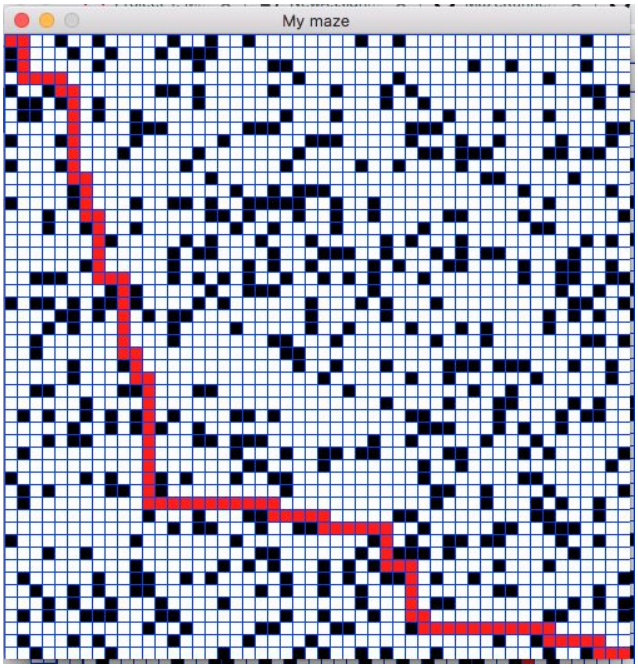
8) Yes, there are nodes in a maze that bidirectional BFS expends that A* doesn't, this is because of the heuristics factor of A*. bidirectional BFS is basically BFS running on the both sides of graph. When you see the combined expanded nodes of the algorithm it will be very similar to the regular BFS. A* expands only the once that make sense to expand depending on the heuristics.

Part 3 - Generating Hard Mazes

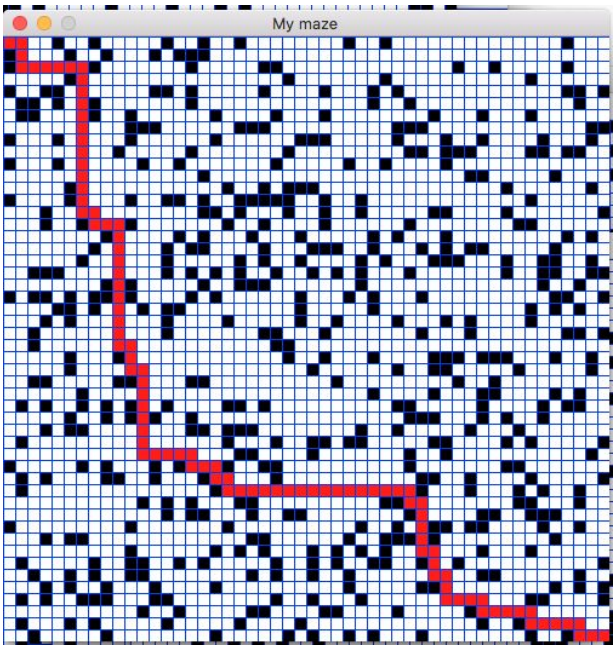
1) To solve this problem we used the idea of Steepest-Ascent hill climbing to solve this problem. Using a similar idea to our bfs we implemented this hill climb idea. We have a node and that is our current node and we will be checking all the neighboring nodes and select the one that will bring you closer to the goal state of the problem. By selecting that node, we will then check the next neighbors of that node and continue from there. To determine if the maze is harder after completing the maze, we created a queue that will search the neighbors of each of the next nodes on the graph and choose the path that is the most optimal. The size of the queue will be recorded and this will be known as our max fringe of the node given while trying to solve this problem. Once the maze is solved we will add a new or remove a block from the maze. This is done through a random number generator where if we get one number it will remove and if we get another it will add to the maze a block. To prevent the map from turning out to have zero changes we added a possibility of the map to add a block to the path. This will enable the search algorithm to have to increase its overall fringe size to solve for the new maze. While running through the algorithm it will find the new shortest path of the maze. And this step will help us determine if the new path has more fringes in its overall path. If it does it will update the new shortest path and give us the new max fringe size. However this is very time consuming and slow process to run on our laptops, therefore we made a stop to our code where if we fail to change the maze 100 times, this means after 100 attempts of randomly placing and removing blocks, if no new path is found of the fringe size does not increase, we will assume that maze is our hardest maze. Any changes to the maze is saved even if it doesn't affect the overall fringe size and then it will continue to add and remove. This code only shows new mazes when a new fringe size of something bigger is shown.
2) The main focus of our hard maze is determined by the maximum size of the fringe at any given point in the algorithm. As a new maze with better fringe size is discovered through our random placement of blocks and deletions of blocks, this change will be recorded for us to see in our graphs. Our code has two terminations that will stop running when our code reaches a certain point. First termination for

this is after running the code through 100 iterations our code will stop going any further, this can be changed for a higher number, however due to limited memory we have to stop the generation of harder mazes. The advantage of this enables us to reduce memory and allows us to terminate the code instead of going on forever at a certain point. However the shortcoming of doing this is that the hardest solution of the maze is not the actual hardest solution, there might be one that's harder. By stopping early we would be unable to determine if it actually is the shortest path. Another termination process that could've been done was if there was a block that resulted in no path found we can assume that the previous graph before this case will be our hardest maze. However with this again is terminating too early as we can't determine if there actually is another maze that will result in a more overall fringe size.

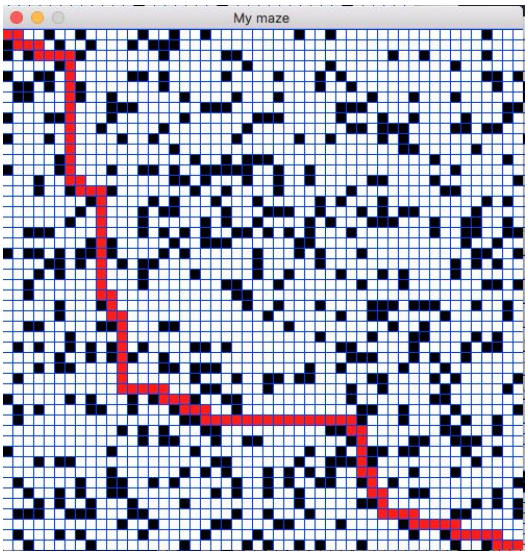
3) Results:



This was our graph at the start of our hardest maze problem. This is a 50x50 matrix and the shortest path is shown, this shortest path also stores the maximum fringe size that was determined in our algorithm.



After the 10th iteration of the maze, this means after a certain change we save in a greater fringe size at this point, this is the 10th change we saw in our code. But this is not the 10th iteration in our code. After randomly selecting blocks to add or remove. This would be the 10th step. As you can see as we add parts into the maze, therefore our maximal fringe size expends and our path is also changed.



This is our 20th new hardest map that was created. As you can see again we have more and more blocked spaces. Therefore our we have to look at more in our queue resulting in a bigger fringe size.

In conclusion, due to us terminating the code early due to the graph having 100 iterations our code will eventually stop. As we assumed the longer we run the code it will generate harder mazes and this case is shown as the maze has more fringes expanded in its queue. What was interesting about this finding was that the path of the shortest length is unaffected most of the time through the algorithm. This is because most of the changes does not affect the overall direction sense in a large grid we have other options to move. As long as we move right and down without any direction upward of left this will not affect our total distance travelled. This was our theory by generating new mazes we went through more nodes in our queue. However this does not affect, most of the time our shortest path. Due to this reason we determined it was better to use the maximum fringe expanded rather than the length of the shortest path. At a certain point adding another block will start to decrease our overall fringe size. But over shortest path still remains the same due to there being no change in the maze.

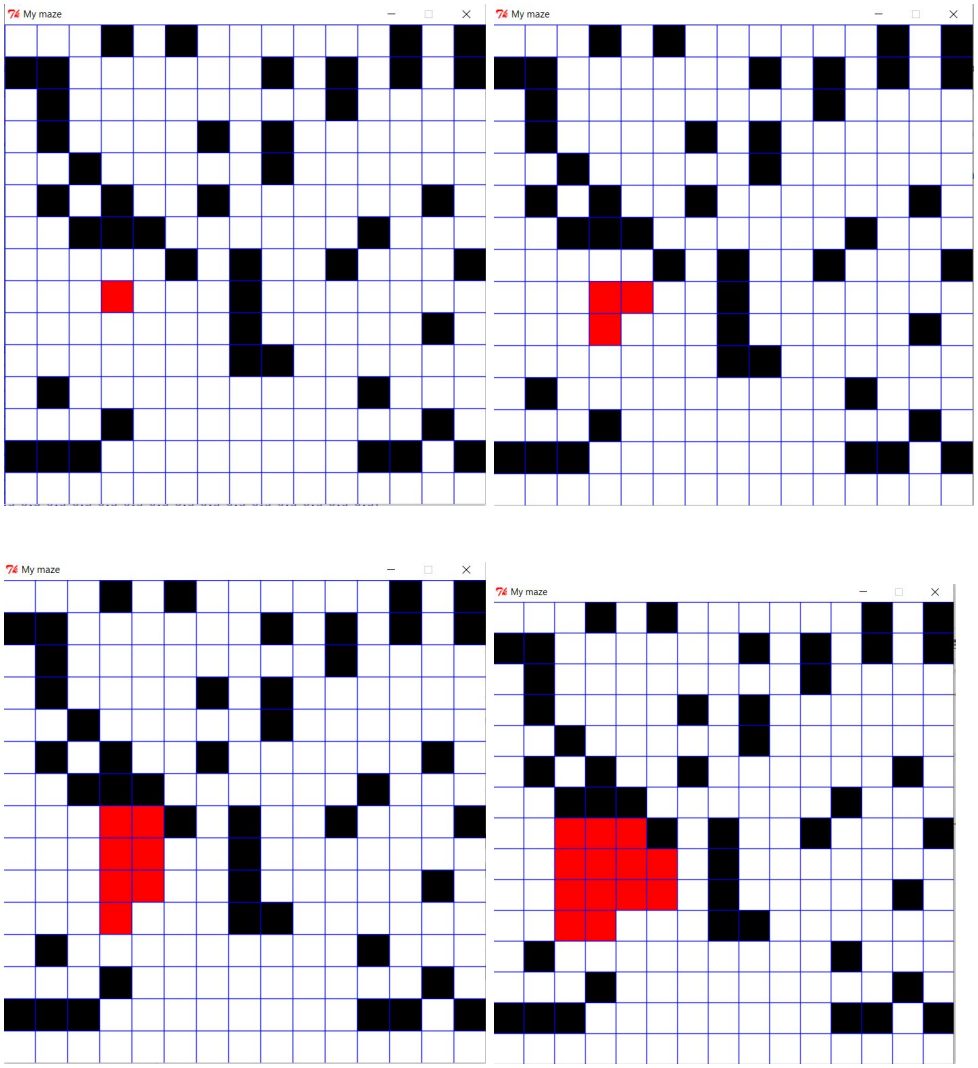
Part 4 - What If The Maze Were On Fire?

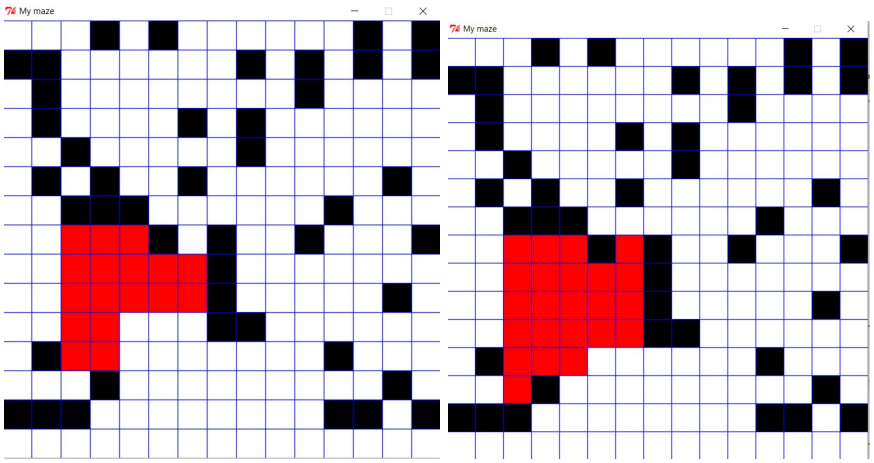
In this part, we are tasked to find a path from the start to the goal, while the maze is continually on fire. The fire has a few rules on how it spreads as well. The spread of the fire depends on the q value, which is the flammability constant, and the number of neighbors, k, that are on fire, given the formula 1-(1-q)^k.

Creating The Maze:

In order to create the maze, we selected a random block on the maze to start the fire. Then, for each time-step, we computed the probability of the current block catching on fire based off of the rules specified above. You can find this code under fire.py, as this will

create a maze that will catch on fire. For example, the spread of the fire can be documented in this way:

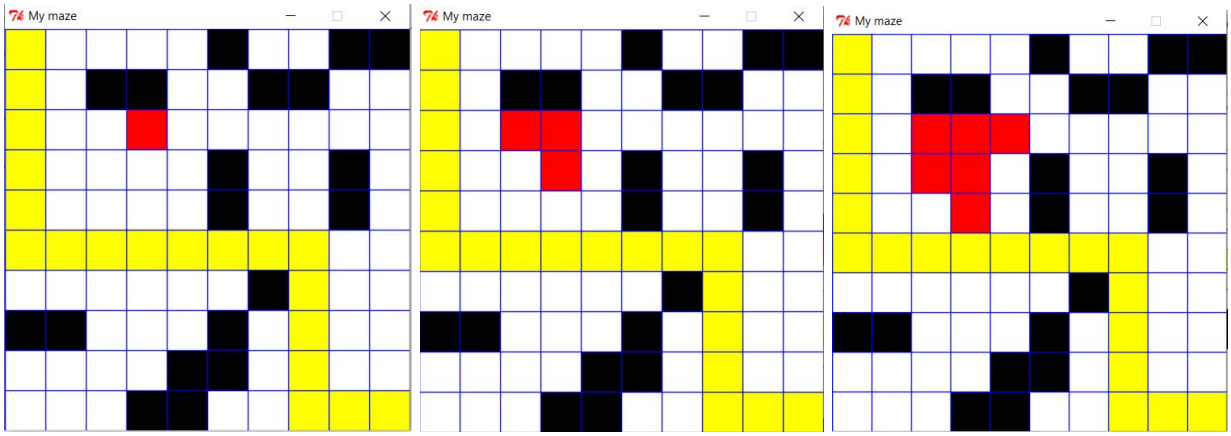




Depending on the q value, the fire spread per timestep will be higher or lower, because the flammability of that specific block will be different. For a higher value of q, blocks will burn faster, and for a lower value of q, they will burn slower.

Strategy 1:

Strategy 1 is to run a shortest path algorithm, without regard for where the fire is. The initial path will be the final path, and the path will not care where the fire is. If the fire catches the path, the "robot" will die. The code for this strategy is in the file strategy1.py, and will find the shortest path first, and will keep that path even if it dies. Documentation of this is as follows:

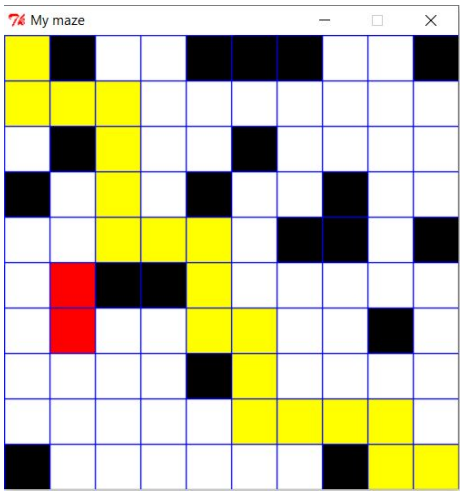


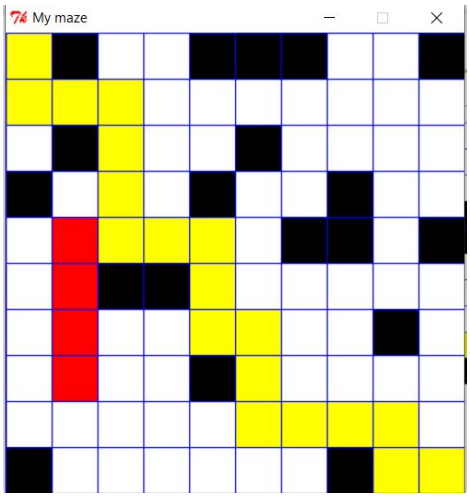
```
you are burned
```

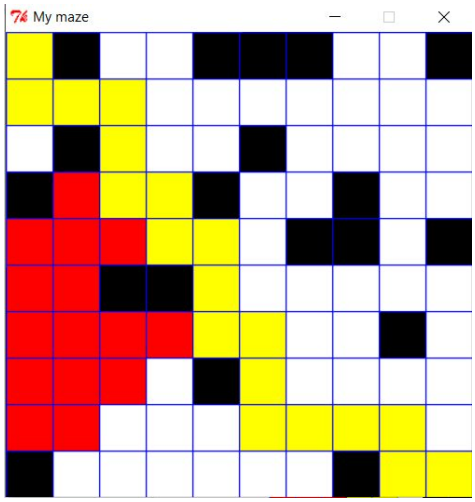
The way this algorithm works is that it runs BFS on the maze first to find the shortest path. The fire will still spread with every click (simulating a time-step). As it goes on, when the fire crosses the path that BFS found, the maze exits, and a message pops up that tells you that the path has been burned. In this strategy, the path does not account for the cells that are on fire at all, and pretends the maze is completely fine until the path itself gets burned.

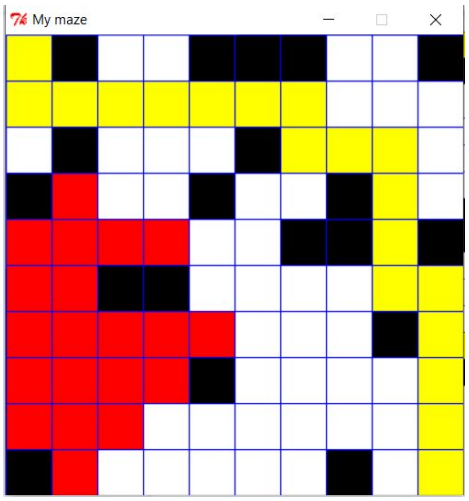
Strategy 2:

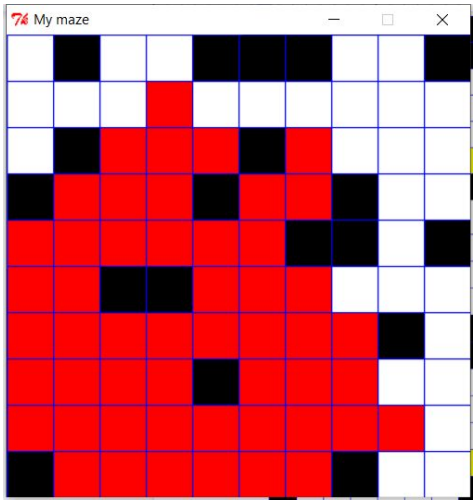
Strategy 2 is to recompute the shortest path every time the fire extends. The way we did this, is by running BFS on the maze at every time step, and making sure that the path is recomputed to account for the fire burning. This will find a shortest path and avoid the fire at every time-step. Unlike strategy 1, this strategy actually updates itself based on which blocks are on fire and actively tries to find paths around them. The code for this can be found under strategy2.py and the documentation is as follows:







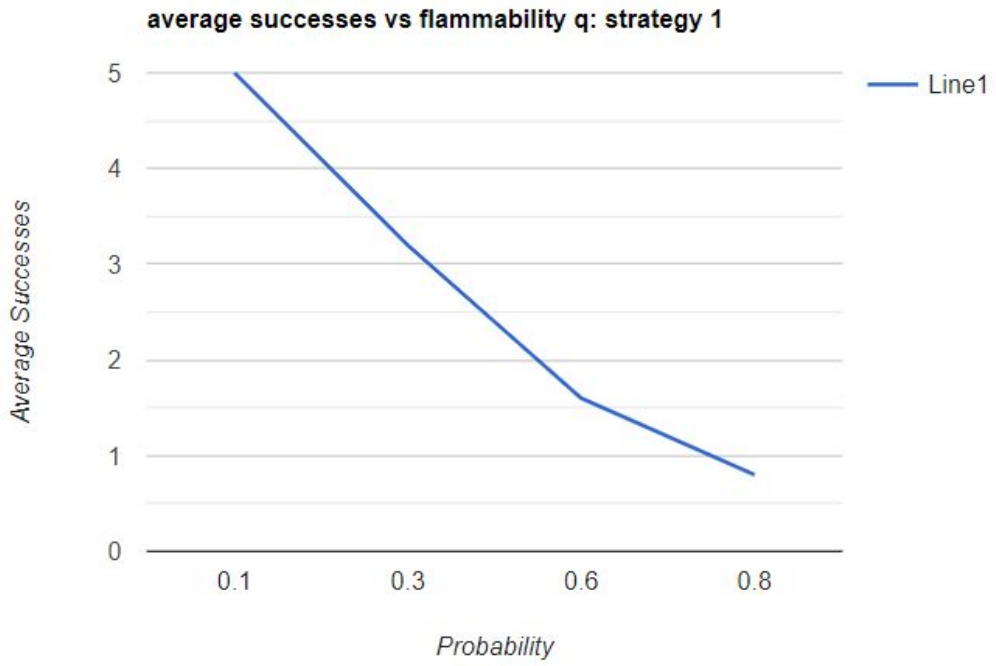


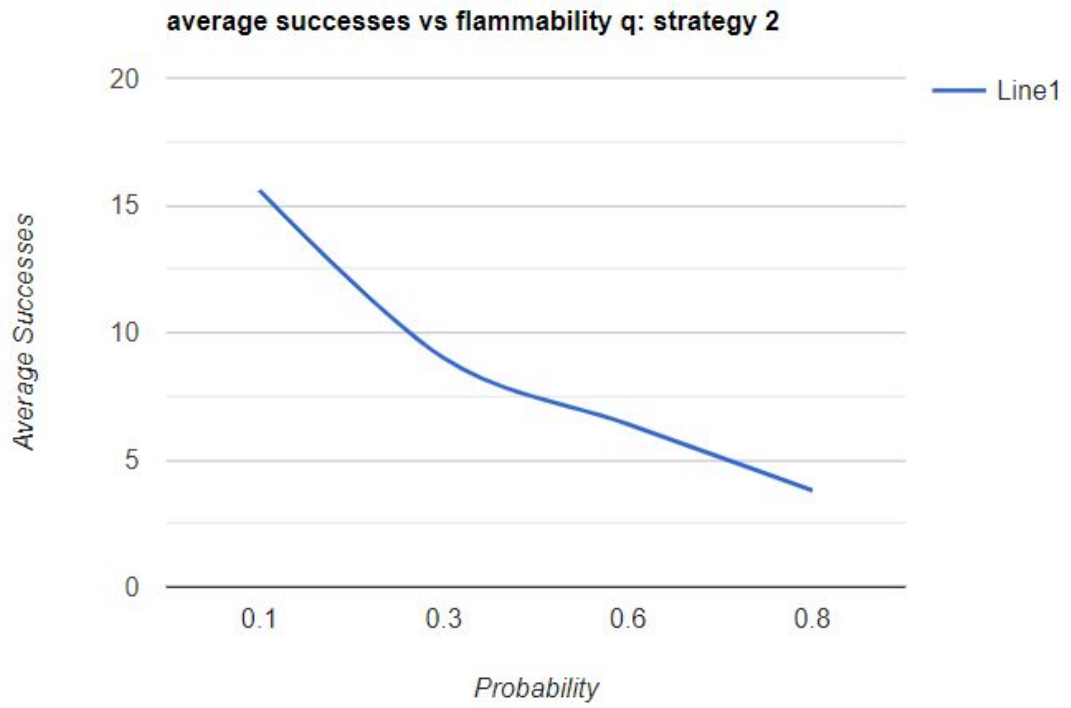



```
no path to be found
```

As you can see above the algorithm first runs BFS to find the optimal path. Then, as the fire spreads into the path that the original BFS found, the algorithm recomputes the shortest path around the fire, and keeps doing so, until the fire blocks the path to the goal entirely.

Comparing Strategy 1 and Strategy 2:





As we can see above, Strategy 1's successes are a lot fewer than Strategy 2's. As the q values increase, the flammability of each cell increases, which means that each cell can turn into a fire block much faster because it has a higher probability of turning into a fire block. This means that when the q values are fairly low, say in the 0.1 to less than 0.5 range, the successes are a lot higher due to the slowness of the fire spread. As q values increase, the rate of the fire spreading will be greater, so this means that more blocks will burn at a faster rate. This lowers the chances of the algorithms finding a correct path towards the goal, and thus the average successes decrease. In both cases, the trend line is similar, in that as the q values increase, it becomes harder for the algorithm to find successes. This is as expected because the faster the grid burns, the less paths the algorithm will find to get to the goal. However, as you can see above, the actual average number of successes are way higher in Strategy 2.

Benefits of Recomputing:

There are clear benefits to recomputing every time the fire spreads. When the fire spreads, the number of blocks that are available for the path to go to decreases. This means that the total number of paths that the algorithm can take will decrease with each time step. Due to this, recomputing the path at each time step will be a lot better at finding successes. Strategy 2 will compute the path that it cannot go to each time step. The fire spreading will reduce the number of possible paths, but the second strategy will still try to find an optimal path, while strategy 1 does not do this. The benefit of computing here outweighs not computing at each step, as summarized by the plots above as well.

Conclusion:

Overall, the burning maze is a way to dynamically solve the maze problem. The maze will change every time step, and this will reduce the total number of paths available in that maze. Because of this, an optimal solution will take into account the changing maze and choose a path based off of that.

Strategy 2 will find an optimal solution based off of the information that is currently present in each time step of the maze. However, there is another solution that we didn't completely have time to implement. This solution will take into account the future "predicted" maze changes as well, and make the optimal solution based off of both current changes as well as future changes. We can think of this as a heuristic, a guess based off of where the fire is right now and where it "might" spread. Wherever the fire is currently, it can only spread to the neighbors of that fire block. We also have the q values which gives us probabilities that the blocks around the fire will burn as well. Due to this, we can guess that the fire will reach these blocks somewhere in the next few time steps, and we can inform our BFS algorithm to avoid these blocks as well. This will give us a larger number of successes overall, and improve our algorithm even further. Taking into account, not only the information that we currently have but also guessing where the possible paths are in the future would ensure the best and most optimal solution to this problem.

Group Distributions:

Part 1&2: Tatsat Vyas

Part3: Andy Guo

Part 4: Adarsh Gogineni