Assignment 1-Search

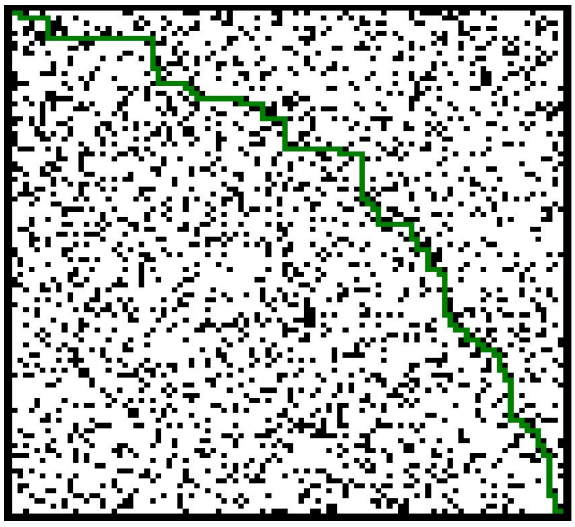
Team Members: Vaibhav Malik, Jonathan Diaz

Responsibilities: We worked on every aspect of the project together. We collaborated on algorithm development, plotting, maze representations, etc. with Jon using C and Vaibhav using Python. We also worked on typing up the final report together.

Analysis and Comparison

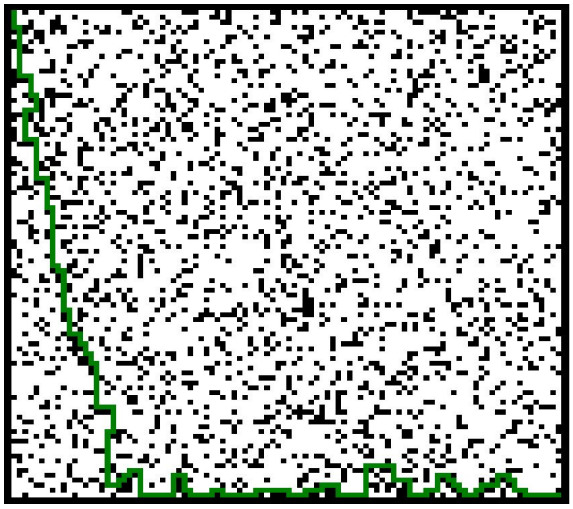
1. We decided to use a dim of 100 to create 100x100 mazes. This maze size ensured that testing functions such as plotting could be completed in a reasonable amount of time, but at the same time the solutions and paths that are discovered by the algorithm would be impossible to find using the human eye. Plotting density vs success rate for the algorithms using 20 density values and running 100 iterations of a specific algorithm for each density value took about 4-5 minutes to run in Python.

2. BFS



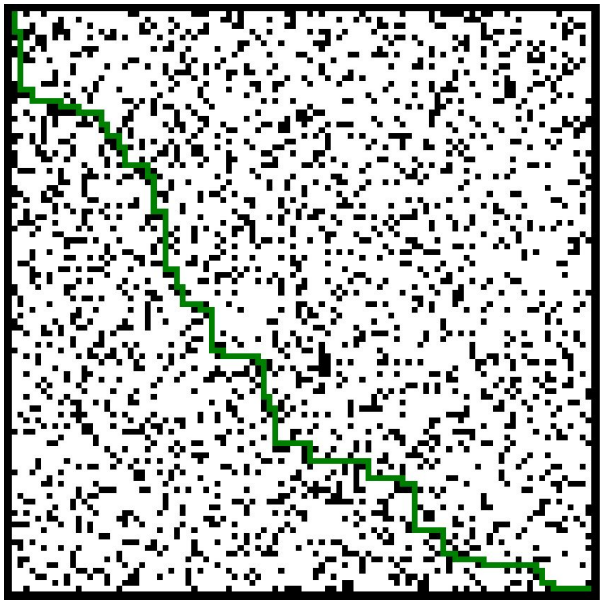
BFS performed as expected as it returned a shortest path. This can be seen by the fact that the path never goes left or up.

DFS



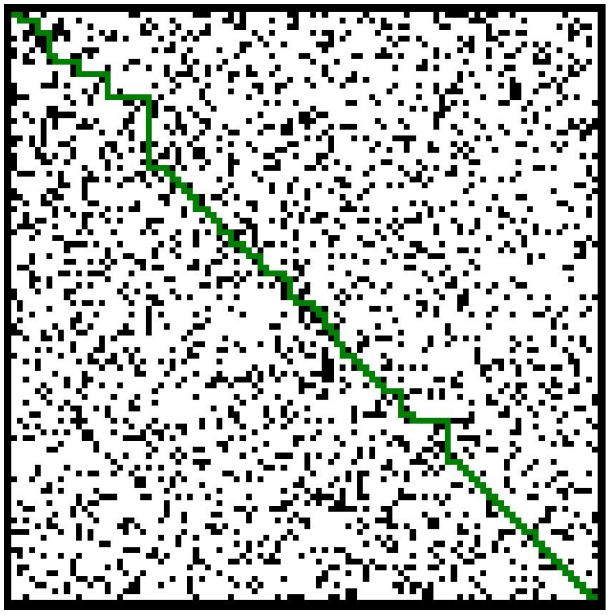
DFS performed as expected as it first prioritizes going down then right then up then left . It follows this priority throughout the whole path.

A* Manhattan Distance



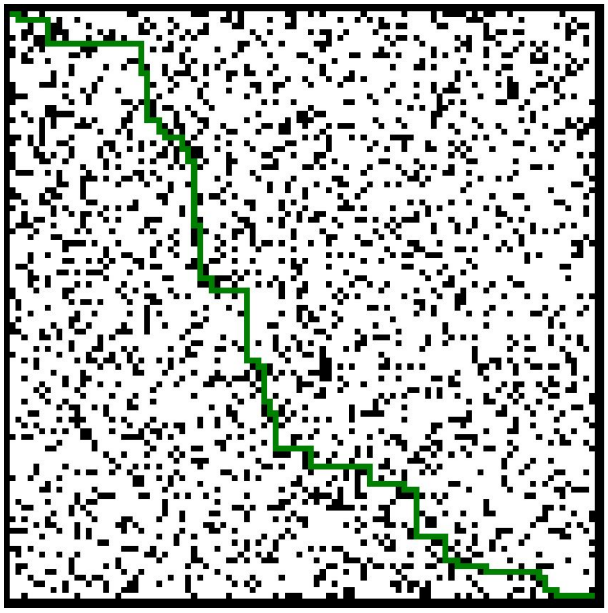
A* following the Manhattan Distance performed as expected because the tiebreaker for cells with the same priority is broken using the same directional pattern as in DFS above. So it has the same tendency to go down then right but also gave us a shortest path unlike DFS.

A* Euclidean Distance



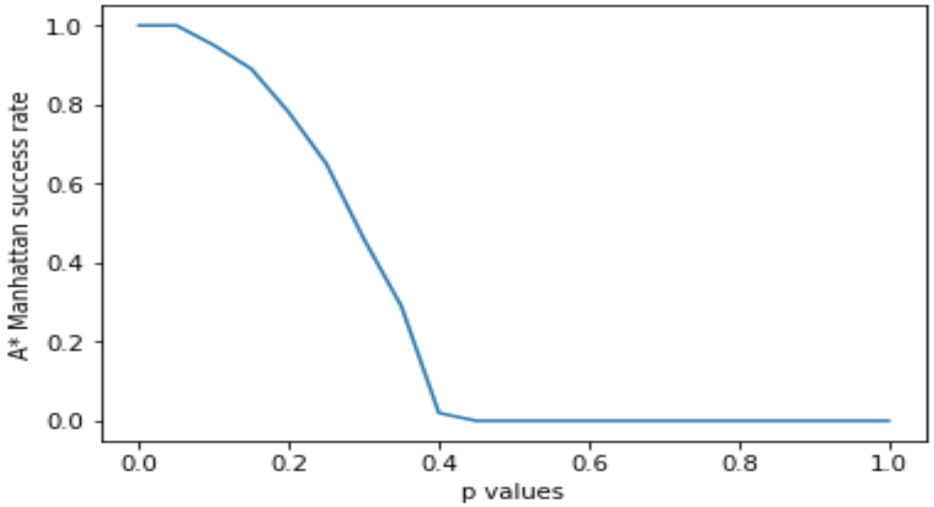
This algorithm performed as expected as it prioritizes the diagonal since that gives the fastest decreases to the heuristic. It also returned a shortest path which is to be expected.

Bi-Directional BFS

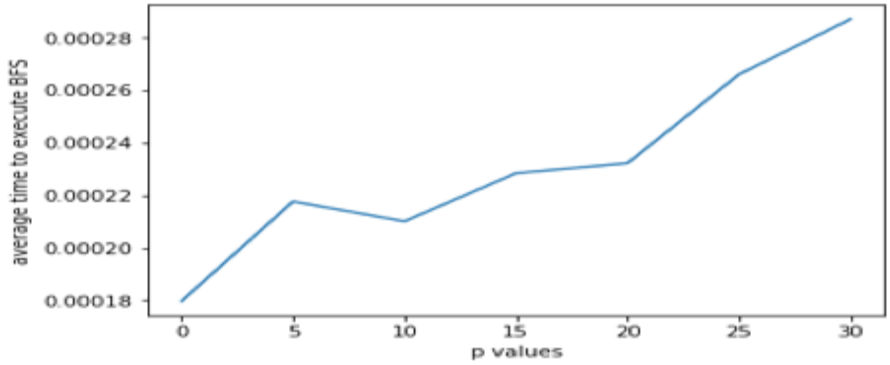


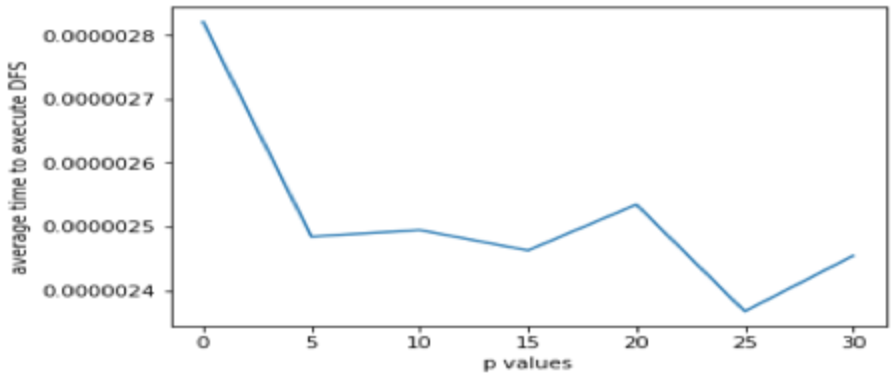
This algorithm performed as expected because the two fringes were likely to meet close to the middle of the maze and once they do the path runs through both of them and this path is a shortest path as was guaranteed.

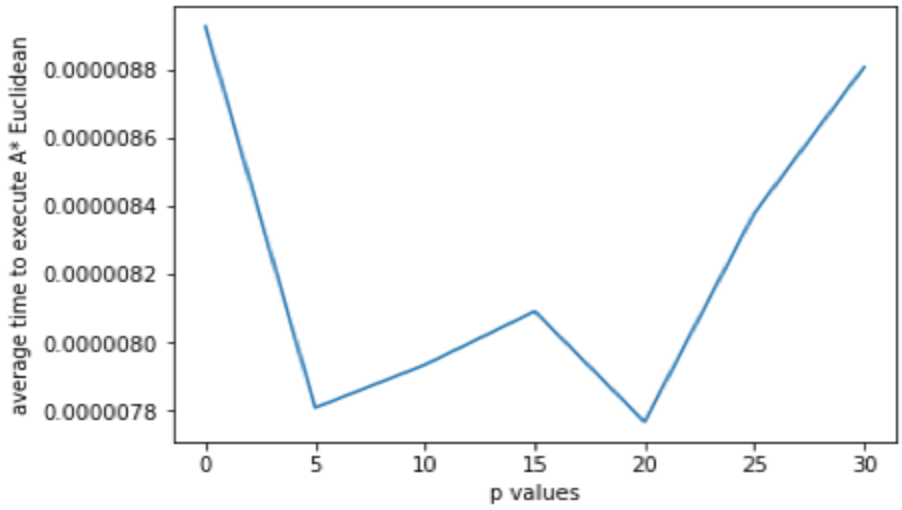
3.

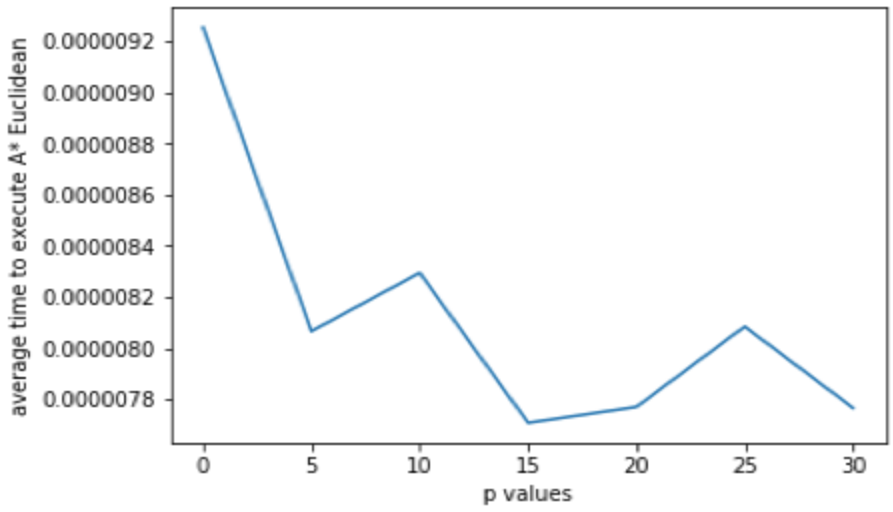


Displayed above is the plot for density values versus solvability using the DFS algorithm. We iterate over 20 different p values, and then generate 100 distinct 100x100 mazes using that given p value. We check for solvability by running the maze through the DFS function and then divide the number of successful mazes by 100. Based on the graph, the p0 threshold where the majority of mazes become unsolvable/the success rate drops below 50% is at p~0.3. After density value~ 0.4, every maze is unsolvable. We believe DFS is the best algorithm for plotting purposes by taking time and memory complexity into consideration. If every algorithm runs without exceeding the memory capacity of our computer then they are all on the same level. We don't need to exceed the memory resources available in order for the algorithms to work. On the other hand, the DFS algorithm outperformed the BFS and A* algorithms in execution times as shown below. We aren't constrained by some fixed amount of time to run our program, so if one algorithm takes less time to run than the other, it is more beneficial.

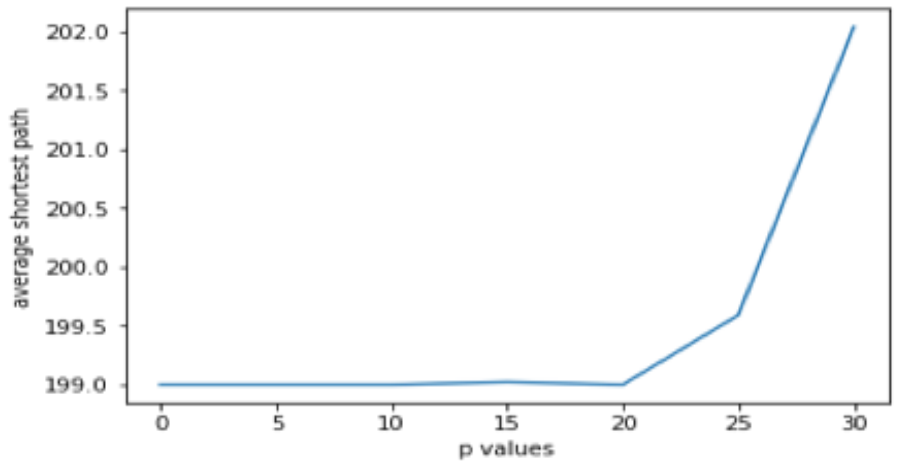



average time to execute DFS
0.0000028
0.0000027
0.0000026
0.0000025
0.0000024
0
5
10
15
20
25
30
p values


average time to execute A* Euclidean
0.0000088
0.0000086
0.0000084
0.0000082
0.0000080
0.0000078
0
5
10
15
20
25
30
p values


average time to execute A* Euclidean
0.0000092
0.0000090
0.0000088
0.0000086
0.0000084
0.0000082
0.0000080
0.0000078
0
5
10
15
20
25
30
p values

4.



Using a similar procedure from the previous question, we found the average shortest path for solvable mazes through 100 iterations.We could no longer use DFS for this purpose as DFS does not guarantee a shortest path. Therefore we used the next fastest algorithm A* using Manhattan distance as every algorithm besides DFS guarantees shortest paths and A* is the fastest among those.

5. In order to determine if the Manhattan or Euclidean heuristic was better we used the maximal nodes expanded measure from the next section. In order to compare them we ran each of them at the same time on 10,000 different mazes. Then in the event of a success we compared the number of maximal nodes expanded for each algorithm and gave a point to whichever one had fewer. We repeated this process for q values ranging from 0 to 0.75 increasing by 0.05 each time. We didn't go all the way to 1 as the mazes are frequently unsolvable at that difficulty. There is no graph for this section as it would be entirely uninteresting since the manhattan heuristic never had more nodes expanded than the euclidean heuristic and in fact it was never even a tie. This makes sense because in the implementation of the A* algorithms that were being compared we break ties in priority of squares by whichever was newest added and in the case where we are using the Manhattan heuristic every square that is reachable without walking away from the goal (either left or up) will have the same distance+heuristic value as every step down or to the right increases distance by 1 but decreases the heuristic by 1. If ties were broken the other way then we would expect the Manhattan heuristic to perform much worse than it currently does as it would expand nearly every node that doesn't require stepping away from the goal more than the shortest path does.

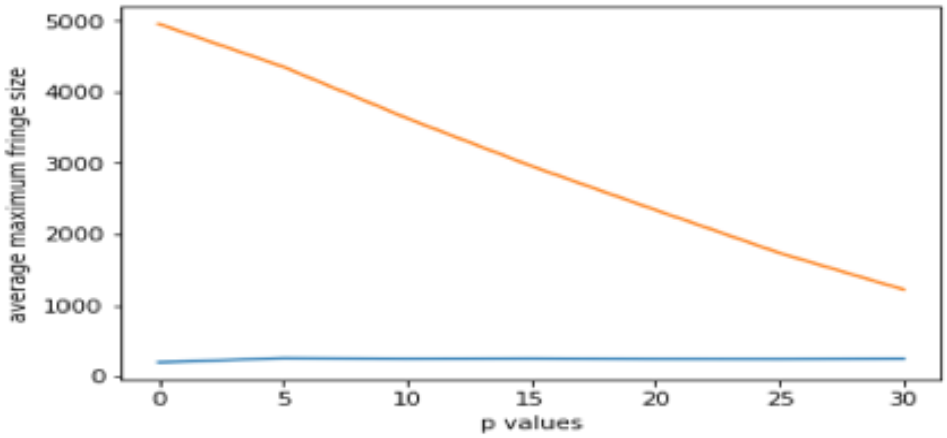
The Euclidean heuristic does not decrease by the same amount that distance increases as you step towards the goal and in fact it decreases by less. For instance in our 100x100 maze when A* is

very close to the goal it's heuristic will be nearly 1 but its distance will be nearly 200 at a minimum. It is easy to see that this will result in it taking the time to check nodes much further away from the goal as their heuristics cant be higher than about 140 and so every node that can be reached in a distance of 60 or less from the start will be expanded and in fact almost every node will be expanded.

In summary the manhattan heuristic is better because the euclidean heuristic is too optimistic resulting in unnecessary nodes being expanded.

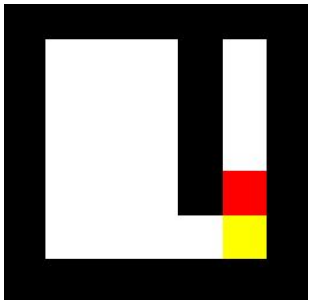
As stated under the images of the paths our algorithms returned, every algorithm is behaving as it should. All algorithms guaranteed to give shortest paths are doing so**,** and the paths they are returning are consistent with how the algorithm is expected to execute and prioritize nodes.

6.



Yes, we can improve the performance of DFS based on the order in which we load neighboring cells onto the fringe. Since the goal cell is located on the bottom right corner of the maze, it would be logical to add the bottom and right cell neighbors to the stack last. We want to direct our traversal in the same direction that the goal cell is located. This would mean whenever we pop a cell off of our stack, our algorithm should prioritize exploring right and bottom cells first. We would expect that following this procedure would lead to a small average max fringe size because we are always looking to move closer to the goal. The graph above shows the massive difference in the two approaches. For a given p value, we ran two different DFS algorithms: one that prioritizes going right and down(blue) and the other which wants to move away from the goal by going up and left(orange). Since the second approach is moving away from the goal, it will naturally encounter more cells and add them to the fringe. As density increases, the “dumb” DFS approach improves because the blocked cells are preventing it from following its preferred direction. We would expect some random process of adding cells to the stack without preference to fall somewhere in between the two graphs.

7. It is possible for BD-BFS to expand nodes that A* won't. We can prove it with an example maze depicted below:



Because of the wall of blocked cells at the right of the maze, A* Manhattan is unable to expand the red square above the goal because it must go through the goal to do so and will terminate after reaching the goal. On the other hand, BD-BFS works from both ends of the maze and will expand the red square above the goal on the first iteration of bfs from the goal.

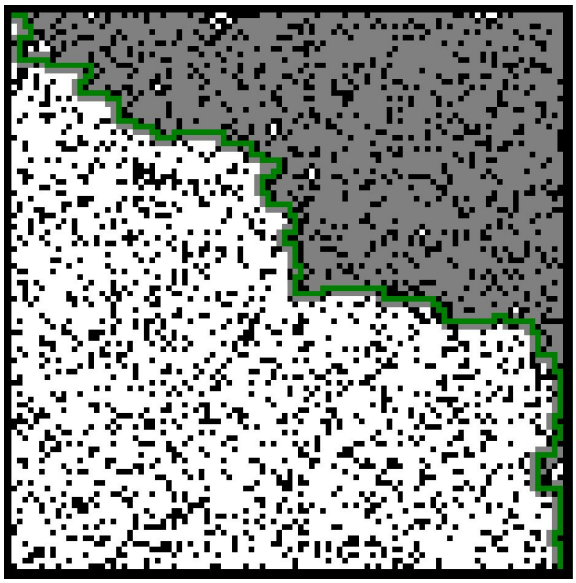
Generating Hard Mazes

1. We decided to use a genetic local search algorithm, similar to the one used for the 8 Queens problem. The main reason for picking this algorithm is because the environment for the maze problem is essentially the same as the environment for 8 Queens: both are a board full of cells. Also 8 Queens relies on splicing boards and adding them together, an operation that is very easy to do in Python using list indexing and list addition. Therefore, it was important that our representation of the maze consisted of Cell objects initialized inside of a list object. We needed to create separate functions for creation of the maze and forming neighbor pointers for the Cells within the maze. This was necessary because after splicing the maze sections, we needed to "reattach" them to another maze section which would contain different Cells. Lastly, this algorithm depends on sorting the population by some metric i.e. max fringe size for DFS. We needed to create a Maze class that has two attributes: the physical maze and the value of the metric when we run an algorithm on that maze. Doing so would allow us to sort a list of mazes in the population using the metric value as the sorting parameter. This would also give us the ability to print an ASCII representation of any maze in the population. Which we could then convert to an image using other code in C.

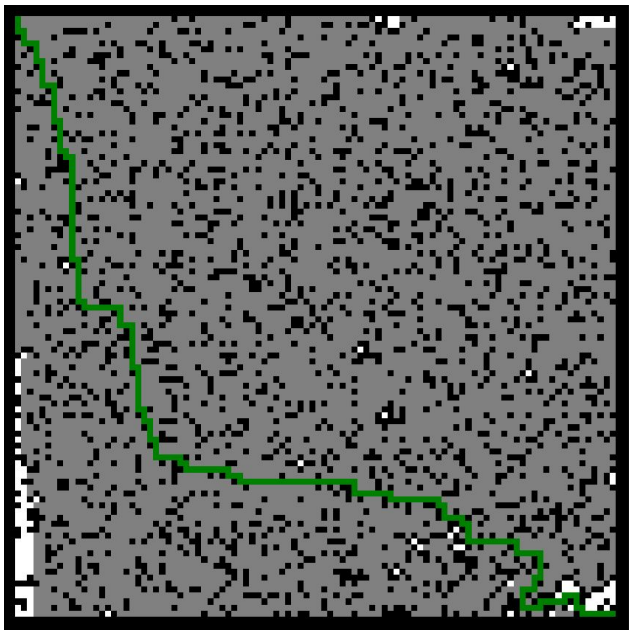
2. In order to determine a termination condition for the function, we plotted the average DFS max fringe size for the population versus generation iteration. We did the same for A* Manhattan maximal nodes expanded in order to find the point where the value of the metric plateaus and reaches its cap. Both metrics peaked at around 40 generations because most mazes in the next generation would become unsolvable beyond that point and those that were solvable were worse. We decided to use of a termination condition of iterating up to 100 generations because it would allow us to visually prove that after 40 generations, the population wasn't

improving and we could be fairly sure we found a local maxima. The advantage of this algorithm is that we are guaranteed that the average metric for the population will only increase or remain constant as the generations increase. We can also uncover patterns from the mazes that remain after the metrics plateau. The downfall of this approach is that it is computationally expensive because when we generate children, we need to initialize brand new maze lists and then iterate over all the Cells in those mazes to initialize attributes. Running the function once for a 100x100 maze with a density of 0.2 takes approximately 30-40 minutes to run.

3.

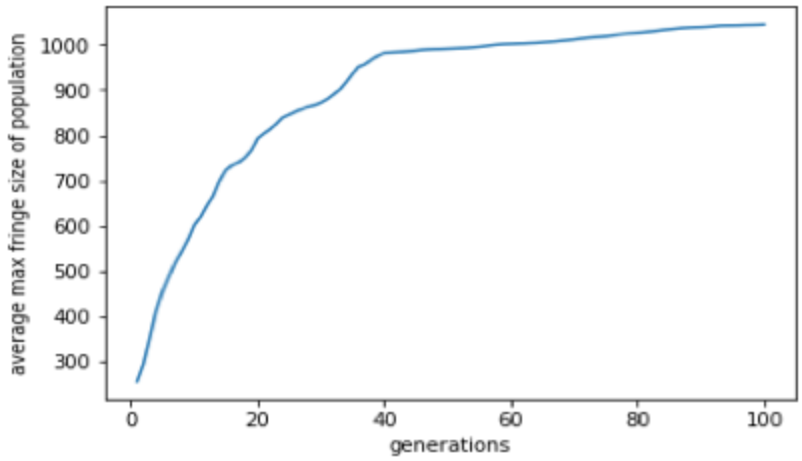


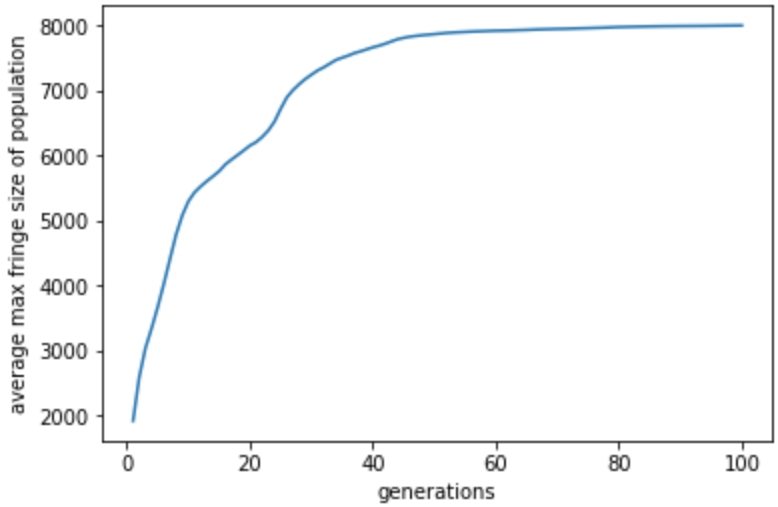
Displayed above is the hardest maze after running our genetic algorithm for 100 generations using DFS max fringe size as the population ranking metric. This maze resulted in a max fringe size of 1047. The squares colored grey do not represent the maximum fringe, merely they represent every node that was expanded.



Displayed above is the hardest maze after running our genetic algorithm for 100 generations using A* Manhattan maximal nodes expanded for the population ranking metric. This maze resulted in maximal nodes expanded of 8017.

4.





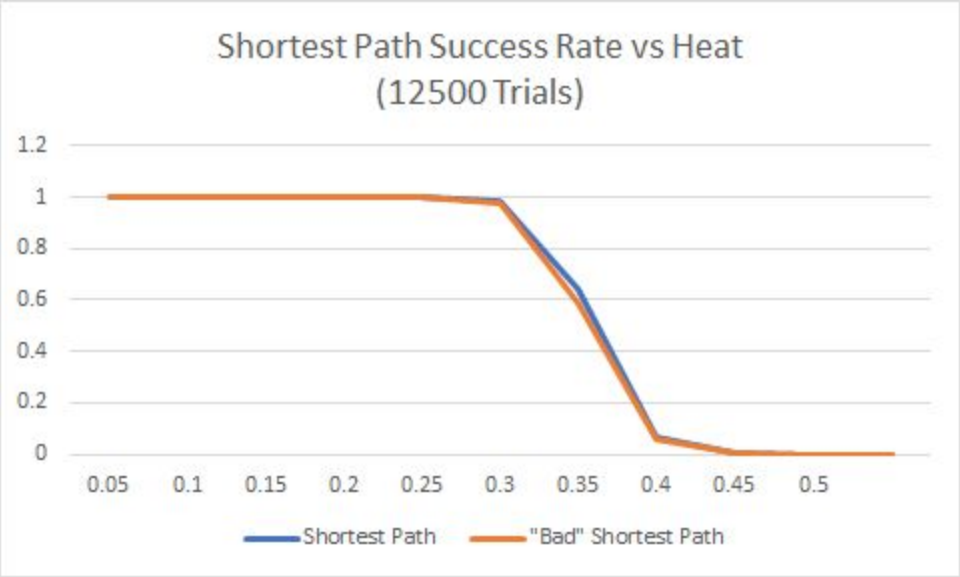
The graphs do agree with our intuition about how the genetic algorithm should behave. We would expect that as the generations go on and the "survival of the fittest" idea is implemented, the average metric value for the population has to either increase or reach a steady state. This is displayed in the graph by the steep climb up to 40 generations, after which the metrics level out. We also predicted that there would be some kind of plateau at a certain point because the mazes can only improve so much before they become unsolvable and reach a local maximum. Once the population contains the hardest maze patterns, their children will likely become unsolvable and the genetic algorithm will continue to stick with that same population as the generations go on. We didn't expect both the graphs for the different searching algorithms and metrics to have such similar trajectories. It was also interesting that the metrics for both DFS and A* Manhattan peaked at 40 generations.

What If the Maze Were on Fire?

All of the paths in this section faced the same maze and the same fire spreading pattern as each other for each individual trial. There were 12500 trials done on 150 different mazes. 100 of the mazes were run simulating the fire 100 times and 50 of the mazes were run simulating the fire 50 times. Ensuring each algorithm was subject to the same maze and fire pattern helped to limit the impact that random chance has on the success of each algorithm. Also no algorithm managed to succeed when the heat value q was higher than 0.5 so that has been omitted from the graphs.

When considering an algorithm for completing the maze while on fire there are two somewhat related goals that should direct your path. Finishing the maze and avoiding the fire. The faster you finish the maze the less time the fire has to spread and catch either you or the goal on fire therefore some priority should still be placed on finding short paths. Since there can be multiple shortest paths through the maze the success rate of picking the shortest path and running it can

depend on how “good” of a shortest path you chose where “good” paths try to avoid the region the fire starts in. By changing the order items that are tied in priority are added to the queue you can have slight control over if the path goes towards or away from where the fire started. Below is a graph of the success rate of running the shortest path and hoping for the best for two different shortest paths generated with one algorithm implicitly trying to avoid the top left corner and the other trying to head towards it.

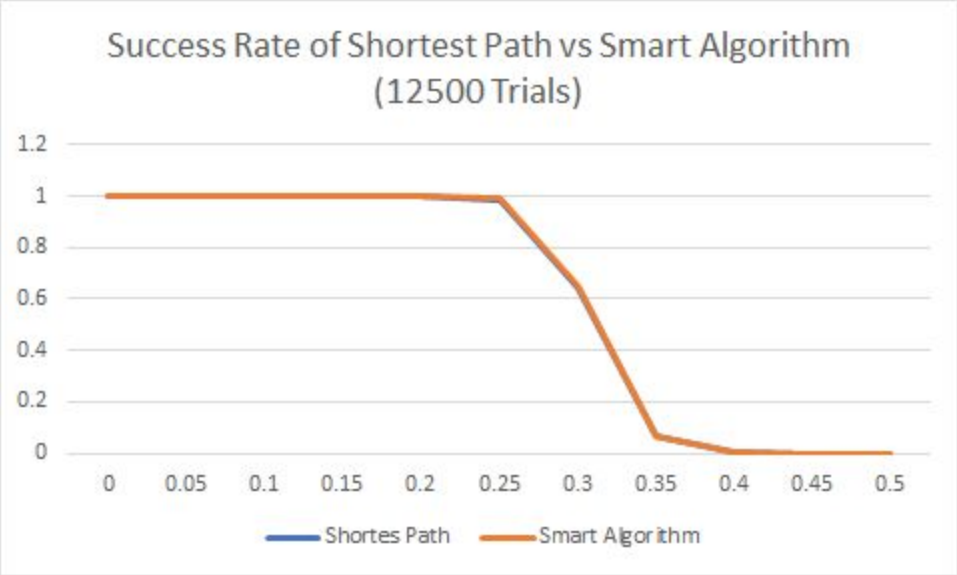


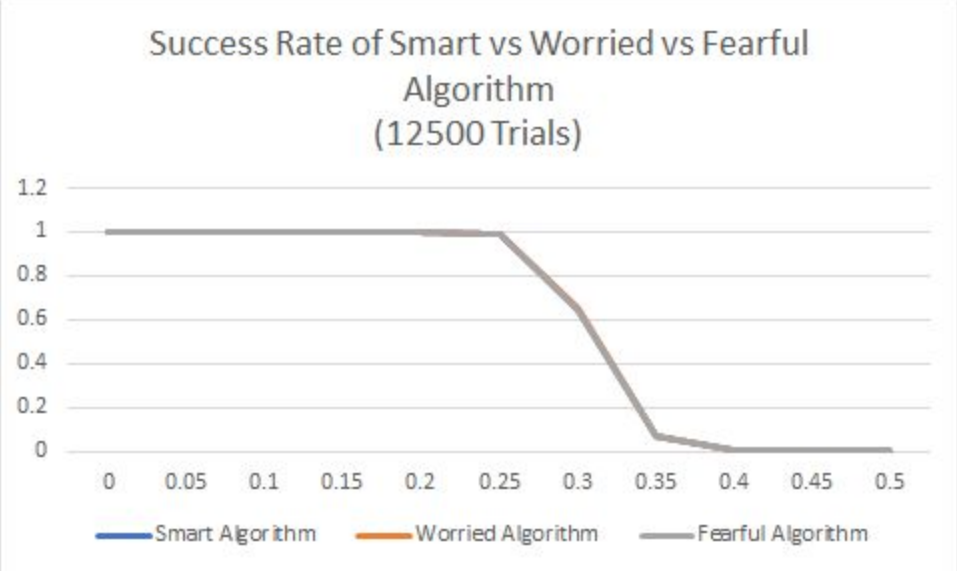
It is worth noting that even though it was skewed towards the fire the “bad” shortest path outperformed the “good” one when heat=0.4.

The idea that we came up with attempted to improve this shortest path approach in two ways. First it re-ran A* after each step keeping track of a rough area that the fire had spread in that step and modified its heuristic for each square based on how close that square was in terms of manhattan distance to this idea of where the fire is. The reason we didn’t use an exact distance to the fire is because it would have been computationally too intensive to find this distance after each step. The new heuristic would take the form: Estimated distance to goal-(p*Estimated distance to fire) where p represents how fearful the algorithm is of fire. The second advantage is that since it re-runs A* each time it can treat squares that are on fire as if they are blocked and avoid walking into a cell that is already on fire. Originally we chose p to be 0.5 but the strategy of following the shortest path was outperforming the scared of fire strategy even though it would never willingly light itself on fire whereas following the shortest path sometimes would. The algorithm was too afraid of the fire and was being steered away from the goal and thus was frequently too slow to reach the end before the end caught fire. We decided to stop devoting resources to analyzing a clearly inferior algorithm and instead analyzed the same algorithm with p values of 0, 0.1, and 0.25 henceforth called the smart algorithm, the worried algorithm and the fearful algorithm respectively. It is worth noting that in the case of the smart algorithm its

heuristic remains unchanged regardless of where the fire is and thus it will follow exactly the same path as the original shortest path unless it is about to step into a square on fire. We would then expect it to perform as well or better than blindly running in identical conditions. The worried and fearful algorithms give no such guarantee as the fire may steer them away from the goal when running straight for it could have worked better.

The following two graphs show the success rates of the smart algorithm vs blindly following the shortest path as heat increases and then the success rates of the smart,worried,and fearful algorithms against each other.





Although it is nearly impossible to see from the graphs the Smart Algorithm (where p=0) did uniformly outperform blindly following the shortest path as was expected.

When comparing the three different p values in the second graph, the one with the higher success rate changed depending on how high the heat value was. This makes sense because as the fire gets more aggressive staying further from the fire becomes a more viable strategy. It is also worth noting that the timid strategy was the only one that managed to finish the maze when the heat reached 0.5.

The algorithm we just analyzed was not our first idea. Our original plan was to calculate the probabilities that any given square would be on fire at any given time given the current state of the fire in the maze. Then using these probabilities finding the probability of success of each potential path through the maze and picking the one that maximized that probability, taking a step down that path, then repeating after the fire has spread. This was infeasible for a number of reasons, firstly being the total number of possible paths is too large to check every single one of them. But even after accounting for this and intelligently choosing paths to consider there is still the problem of calculating the exact success rate of each of those paths as a lot of the steps in the path are conditional on previous steps succeeding. After considering if we could come up with a good estimate of the success rate of a path we also ran into the problem of being unable to calculate all the probabilities at every time we needed to consider and were unable to figure out an efficient way to find those.

The algorithm we settled on could be improved with more computing power as we could find the actual shortest path to the fire from each unlit square instead of using a guess at the shortest path to some region the fire could be in. Although if computing power were of no concern we could implement the algorithm we gave up on in its initial glory and we would expect that to on average be the best performing one there is.