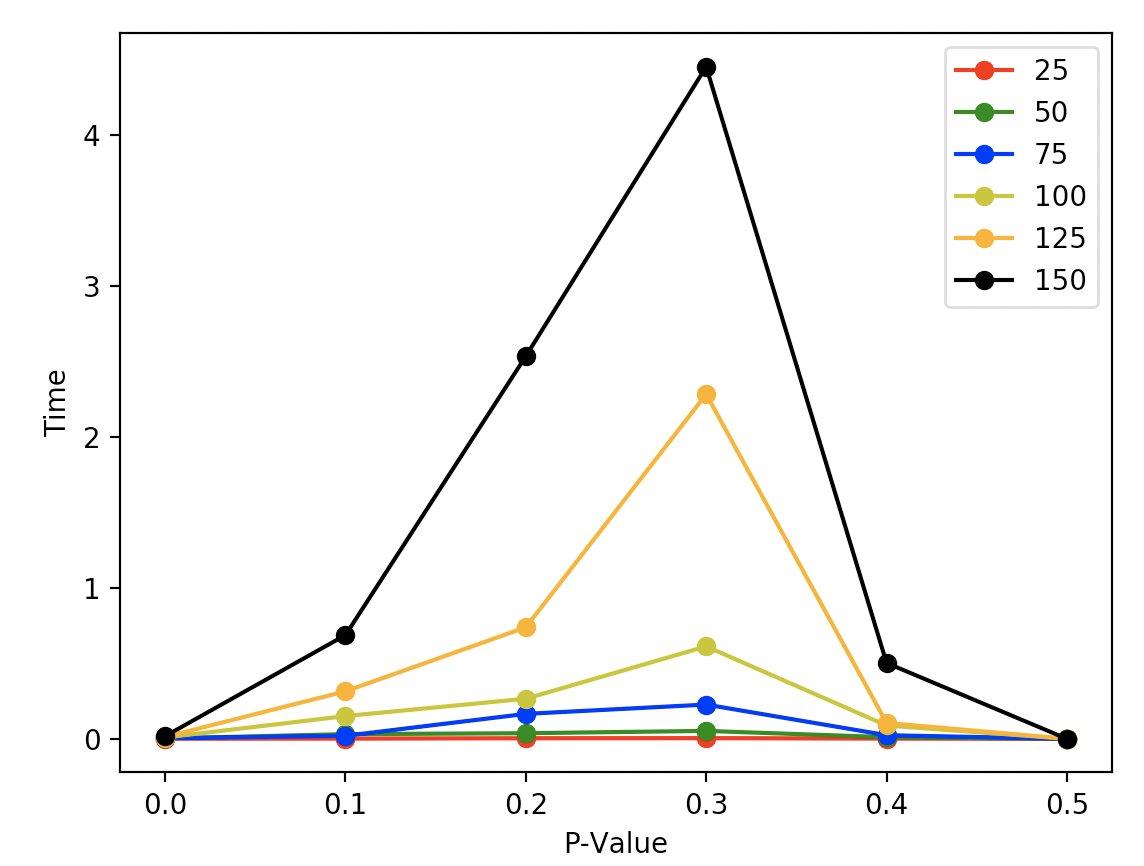
**Part 2:**

1. **Find a map size (dim) that is large enough to produce maps that require some work to solve, but small enough that you can run each algorithm multiple times for a range of possible p values. How did you pick a dim?**

Finding a good size dim is important for analyzing the different algorithms both separately and against each other. If the DIM is too small, not only will we not get a good representation on how the algorithm performs in a larger, more complex scale, it will also generate too many impossible to solve mazes for higher P values. If the DIM is too large, then we could be waiting minutes for a single algorithm to complete. We are looking for the middle ground where it will not take too long for the slowest algorithm to complete if ran multiple times, and will be complex enough that we can get a good representation of each algorithm.

We can do a benchmark for different size mazes at different P values to find the best middle ground. We are looking for something that requires work to solve, which we can justify by the time it takes the algorithm to complete. We can make this justification because the more time it takes an algorithm to complete, the more work the algorithm had to do. So, we will create a graph of **Time vs P-Value** with different DIM’s on the same graph to compare the results. We use A\* Manhattan for this test since it is very fast and we can get a good average with many tests.

**Results: Using part2/DIM\_benchmark.py**

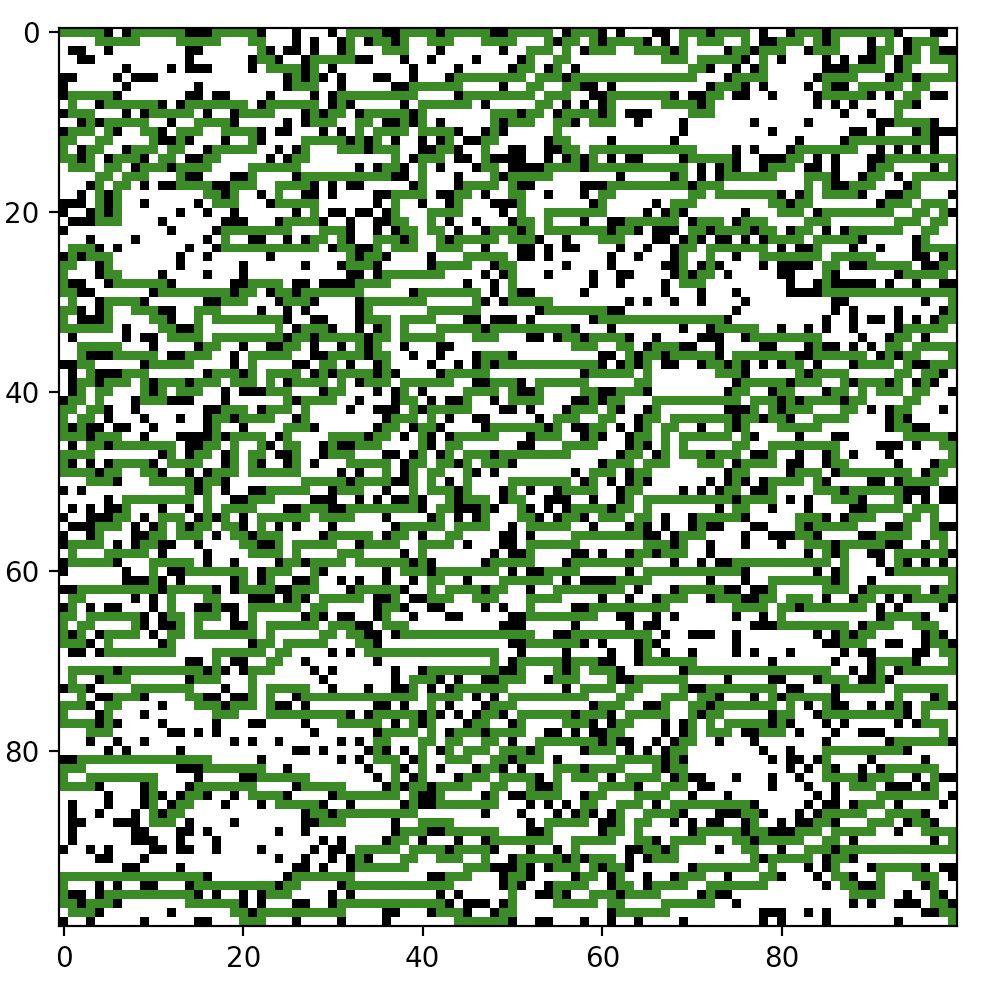
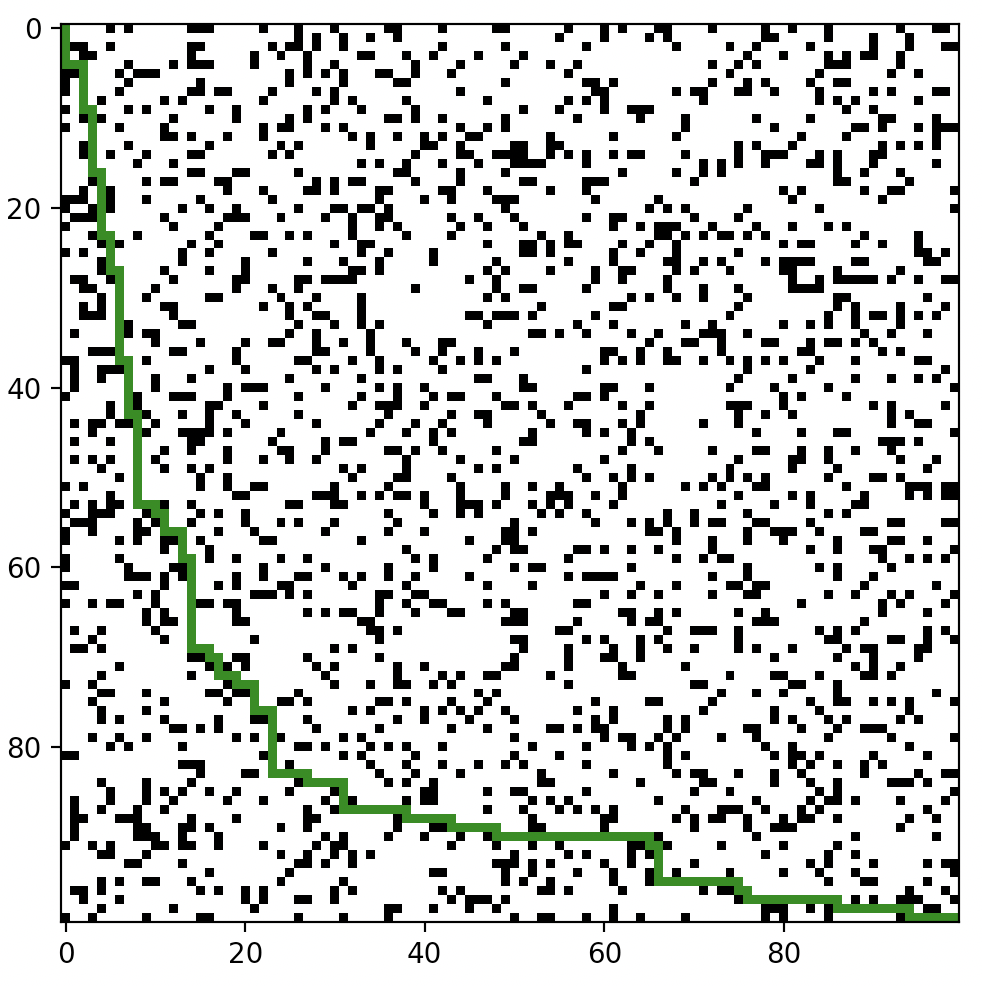


The above figure represents the different DIM’s with different P-Values. For each DIM and P-Value, the test was run 50 times and the average was taken. We only tested for P-Values 0.0 – 0.5 because past 0.5 it is almost guaranteed to not generate a map that is possible to finish, the results would all be 0 like 0.5 is. Since we did this test with A-Star Manhattan, so 3 seconds for it to complete could be a minute or more for BFS to finish. For that reason, we can see 150 takes a bit too much time to compute for our purposes. We can also see 25,50, and 75 do not have to do much work to solve, so we can narrow down our choices to 100 and 125. P-125 looks promising but similar to 150 it has more of an exponential growth which could cause problems with more extensive testing. So we will chose 100 DIM as it offers a nice middle ground between everything.

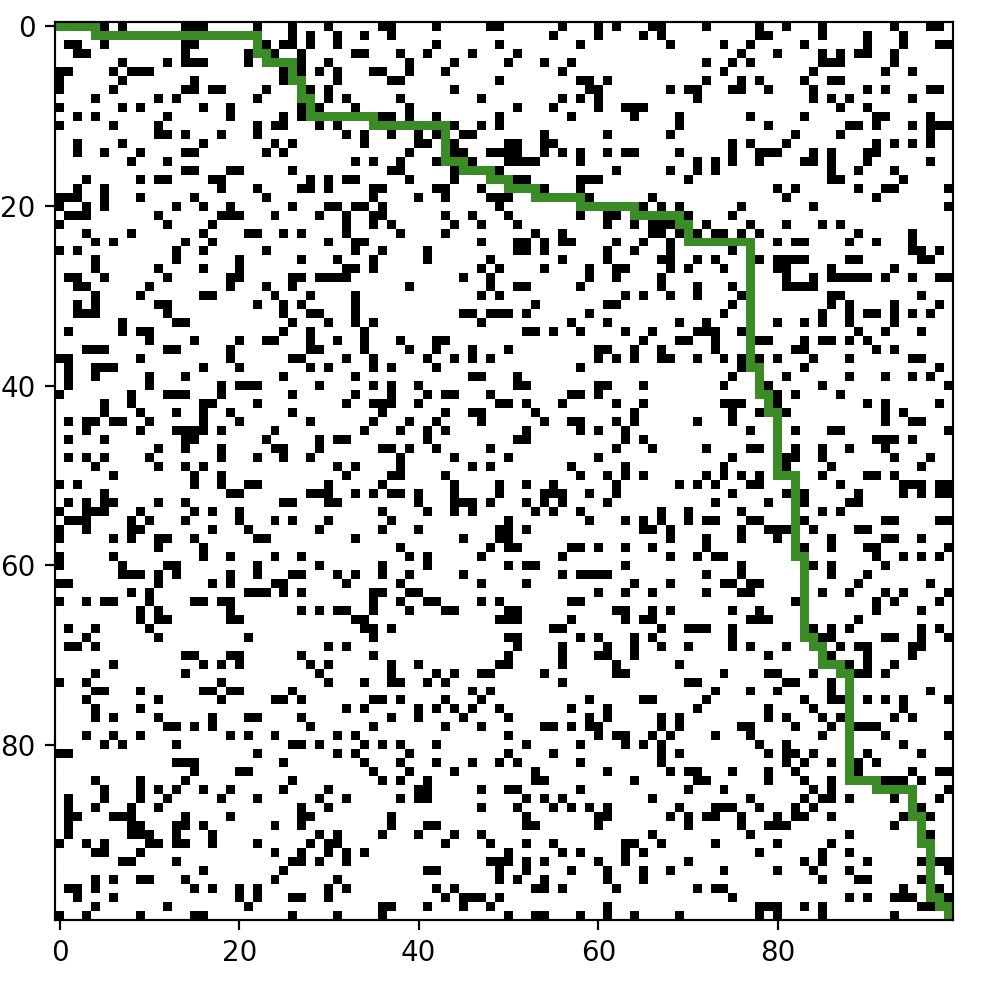
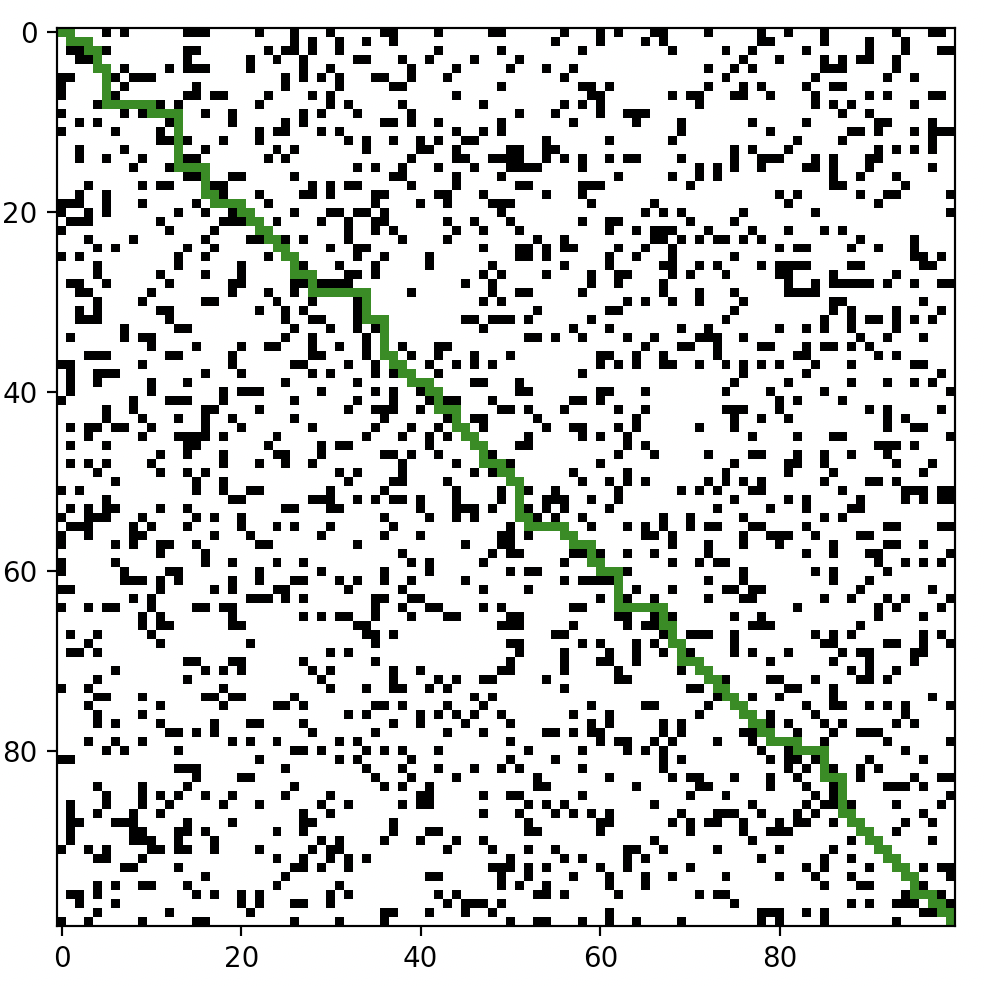
**DIM chosen: 100**

1. **For p ≈ 0.2, generate a solvable map, and show the paths returned for each algorithm. Do the results make sense?**

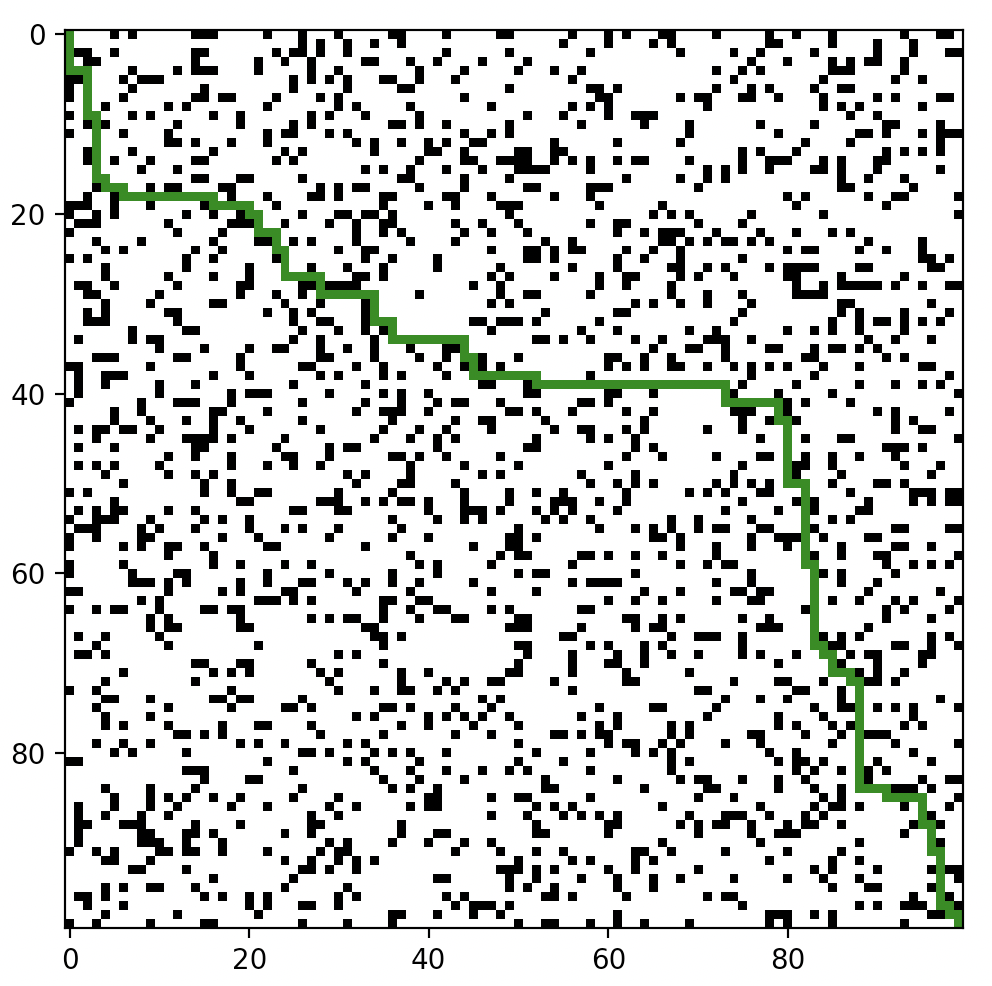
BFS DFS



A-Star Euclidean A-Star Manhattan

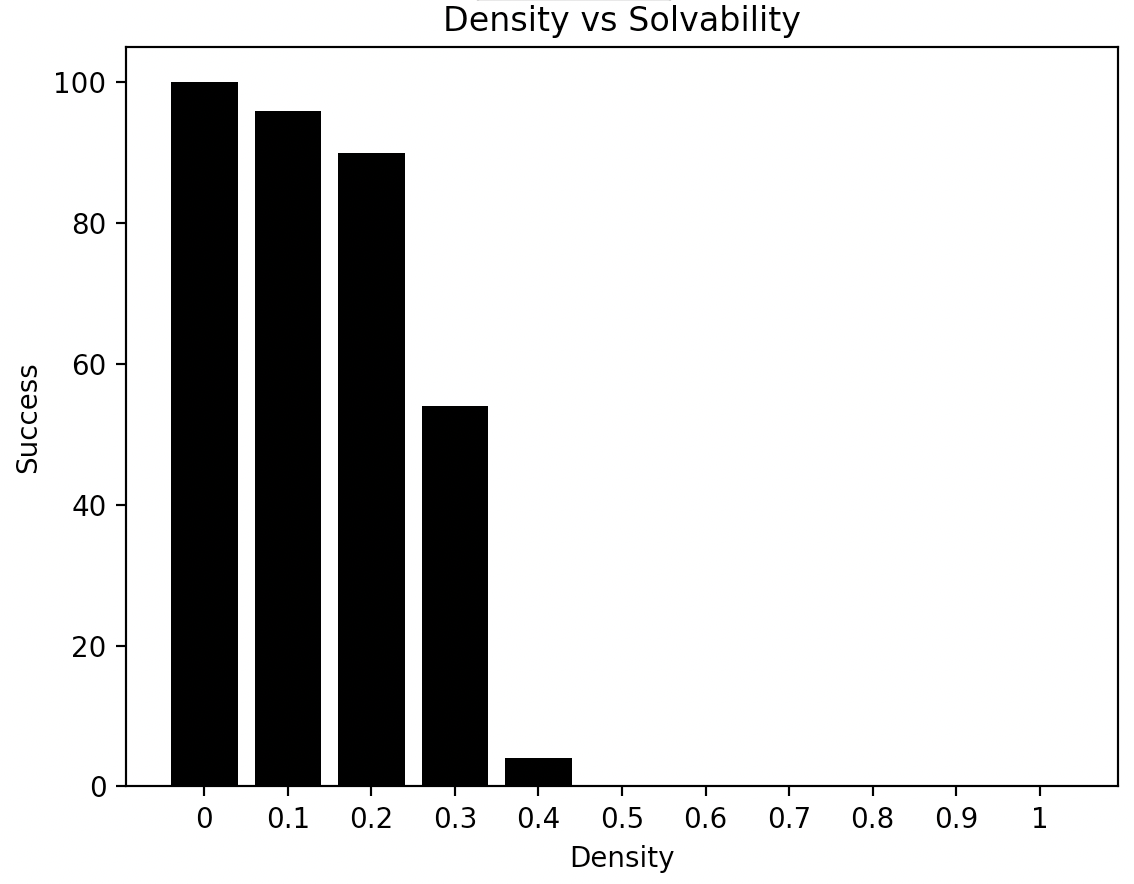


Bi-Directional BFS



The results for each of the algorithm do make sense. For BFS we see boomerang shape as it discovers nearly all nodes in the map, spreading out until it reaches the end as the queue does. DFS is exactly what is expected since it uses a stack and simply goes down the cells with the order it checks them and adds them into the stack. We can see from the picture DFS is prioritizing Right, Up, Left, Down in that order. A-Star Euclidean looks as it should, going diagonally across as the heuristic is based on a diagonal distance. A-Star Manhattan also looks like it should, with Manhattan distance heuristic prioritizing going right and downwards. It could have also gone down and to the right similar to BFS, but it just was the way equal f scores went into the priority queue choosing right over down. Lastly, Bi-Direction BFS, which is also looks as expected. The top left first goes downwards and the bottom right looks upwards, they end up meeting in the middle first which is what we see in the picture.

1. **Given dim, how does maze-solvability depend on p? For a range of p values, estimate the probability that a maze will be solvable by generating multiple mazes and checking them for solvability. What is the best algorithm to use here? Plot density vs solvability, and try to identify as accurately as you can the threshold p0 where for p < p0, most mazes are solvable, but p > p0, most mazes are not solvable.**



DIM = 100, Each P-Value ran 100 times. # of success recorded

P is the main part that determines maze solvability. With a p at 0, that means no walls are generated which means a solvable maze is created every time. As p value increases, the solvability of the maze decreases. The algorithm we used is A-Star Manhattan as it the fastest algorithm we have without modifying DFS. DFS could also be used if we modify it to prioritize down and to the right. Doing so will make it run faster than A-Star Manhattan, but the difference in values is in the hundredths place so it does not matter too much.

The threshold where p < p0 where most mazes are solvable, but p > p0 where most mazes are not solvable is most closely looking for a success rate of around 50%. From the graph we can clearly see p = 0.3 most closely resembles this

1. **For p in [0,p0] as above, estimate the average or expected length of the shortest path from start to goal. You may discard unsolvable maps. Plot density vs expected shortest path length. What algorithm is most useful here?**
2. **Is one heuristic uniformly better than the other for running A∗? How can they be compared? Plot the relevant data and justify your conclusions.**

**Part 4:**

Adding fire that is spreading in the maze drastically changes the entire paradigm of the maze solving algorithms. Until now the maze has been static, so each iteration of the algorithms had the same maze before and after computing, step by step. Now the maze has a chance to change, which means that the possible shortest path earlier in the path finding algorithm might not be the fastest once it reaches the end.

Strategy 1 has the fundamental problem of not considering the fire at all. The probability of the path finding being successful and not burning is entirely up to where the fire starts and how fast it spreads. In other words, it is completely left to luck with no accounting for the fire. Since the fire can spawn anywhere in the maze, it has a very high chance of spawning near the middle. This means that even with a probability of 0.1 of the fire spreading, it will be detrimental to most of the algorithms that brute force there way down the middle. Since they do not take to account the fire, by the time they realize they needed to go around it is too late and the fire is spread too much.

Strategy 2 is miles better than strategy 1 as it considers the fire and changing maze. From the start it can make choices of which direction to go is best suited

Implementing strategy 1 yields:

- results as expected, fails to complete maze nearly every time

Implementing strategy 2 yields:

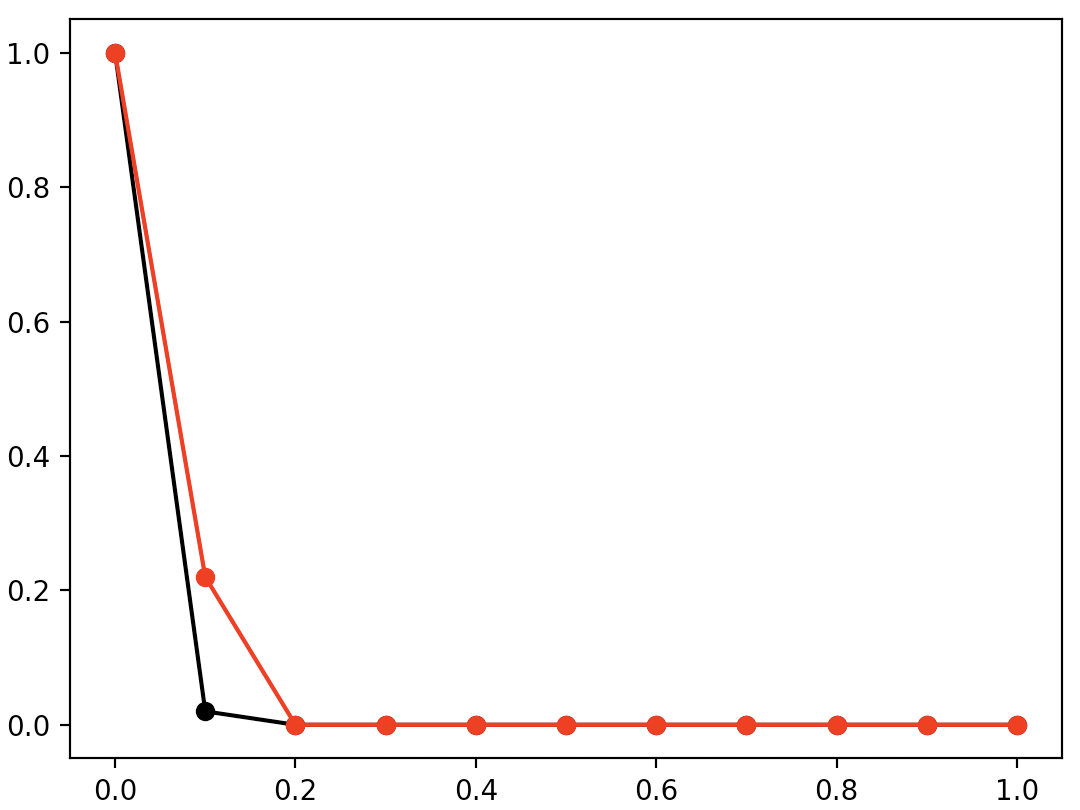
For implementing strategy 2 we need to add the fire into the algorithm so it can take the proper steps to flee and find a way around the growing fire. However, we also do not want it to completely go the opposite direction, ignoring the goal and giving us a path too long or taking the algorithm on a detour that takes too long and ends up burned in flames. We use A star with Euclidean distance with a modified heuristic. For every step, we want to pick the cell that is farthest from the nearest fire cell but closest to the goal. We get rid of the G value (distance from start to current location) because we do not care anymore about picking the fastest path, but rather the safest. We calculate our new F value as follows:

F = H – FireDistance

FireDistance = Euclidean Distance from current node to nearest fire block

With this new heuristic, the algorithm now resembles more of a DFS search than a A-star, but this is exactly what we wanted. Every time the algorithm goes far back from the current node to find a better path, it is wasting valuable time that we cannot afford to lose. Instead, our algorithm takes a path away from the fire and near the goal, and will look for new paths in an identical way to DFS near the current node rather than going back near the start.

**Does re-computing your path like this have any benefit, ultimately?** Yes, it absolutely has added benefit. There is still the large play of randomness for where the fire will first spawn, but on average strategy 2 will perform better than strategy 1 as shown in this graph:



Average

Q-Value

Black = Strategy1, Red = Strategy2. DIM = 100, p = 0.3

Each Q ran 50 times on unique solvable mazes

As we can see Strategy 2 does perform better at lower Q Values but not by a whole lot. This is mainly due to the fact that even though strategy 2 recomputes the path every step accounting for the fire, it does not predict well. For example, at a certain step it might be at a cell close to the end but near a fire cell. If it made a straight path directly to the end without considering the fire anymore it would make it, but instead it chooses to move farther away from the fire and leads it too long, causing the fire to spread and for it to fail.

Let’s greatly improve upon both strategies. We have a maze that has the potential to change at every time step with some probability Q. Strategy 1 does not account for the change and thus fails frequently, but it has the benefit of going directly to the end without regards to the fire which can lead it to success if the fire spawns far away. Strategy 2 accounts for the change but may incorrectly chose to move away from the fire when it should instead dismiss it and go right to the goal like strategy 1 does. What we are looking for is a merge between these two strategies. At any point, the algorithm needs to know when to prioritize going for the goal or moving away from the fire. The only way to accomplish this would be to predict what the fire spread will look like in the future, ascertain the threat of that fire to the current position, and make the decision based on that.