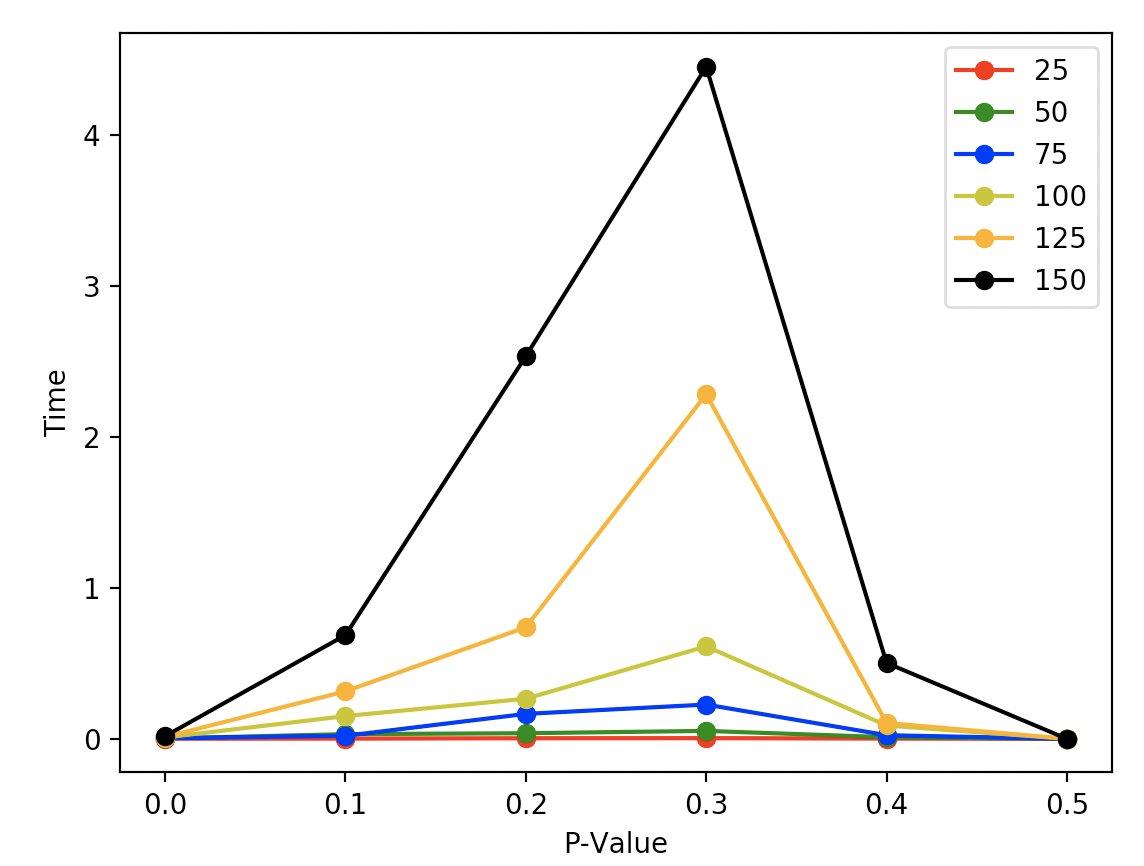
**Part 2:**

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**

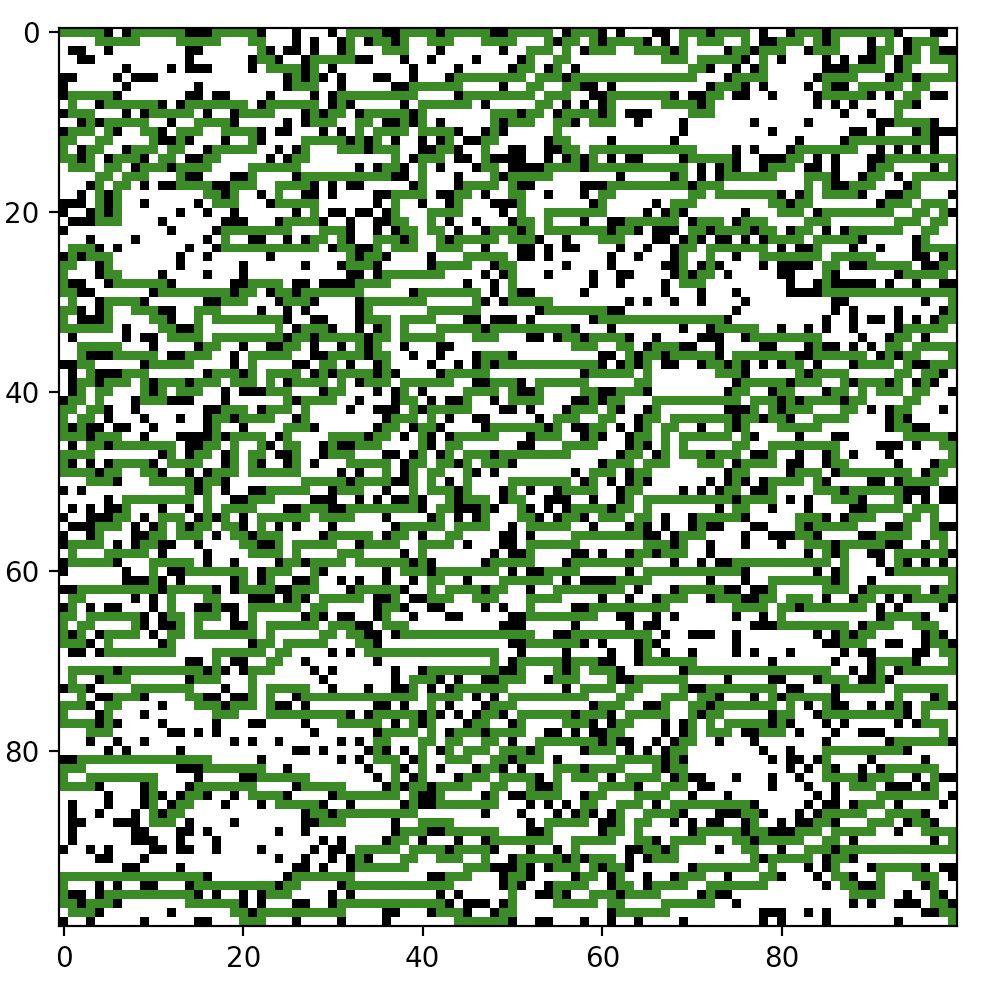
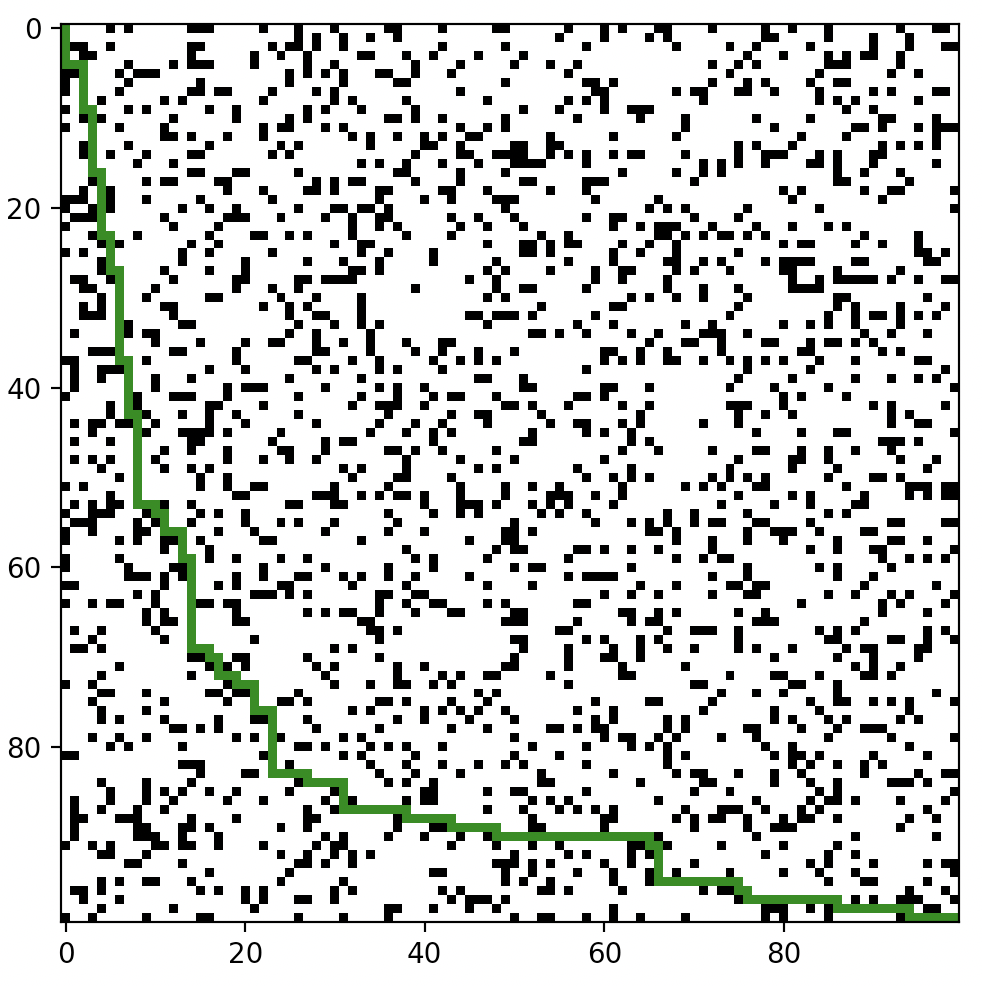


The above figure represents the different DIM’s with different P-Values. For each DIM and P-Value, the test was run 50 times and the average was taken. We only tested for P-Values 0.0 – 0.5 because past 0.5 it is almost guaranteed to not generate a map that is possible to finish, the results would all 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.

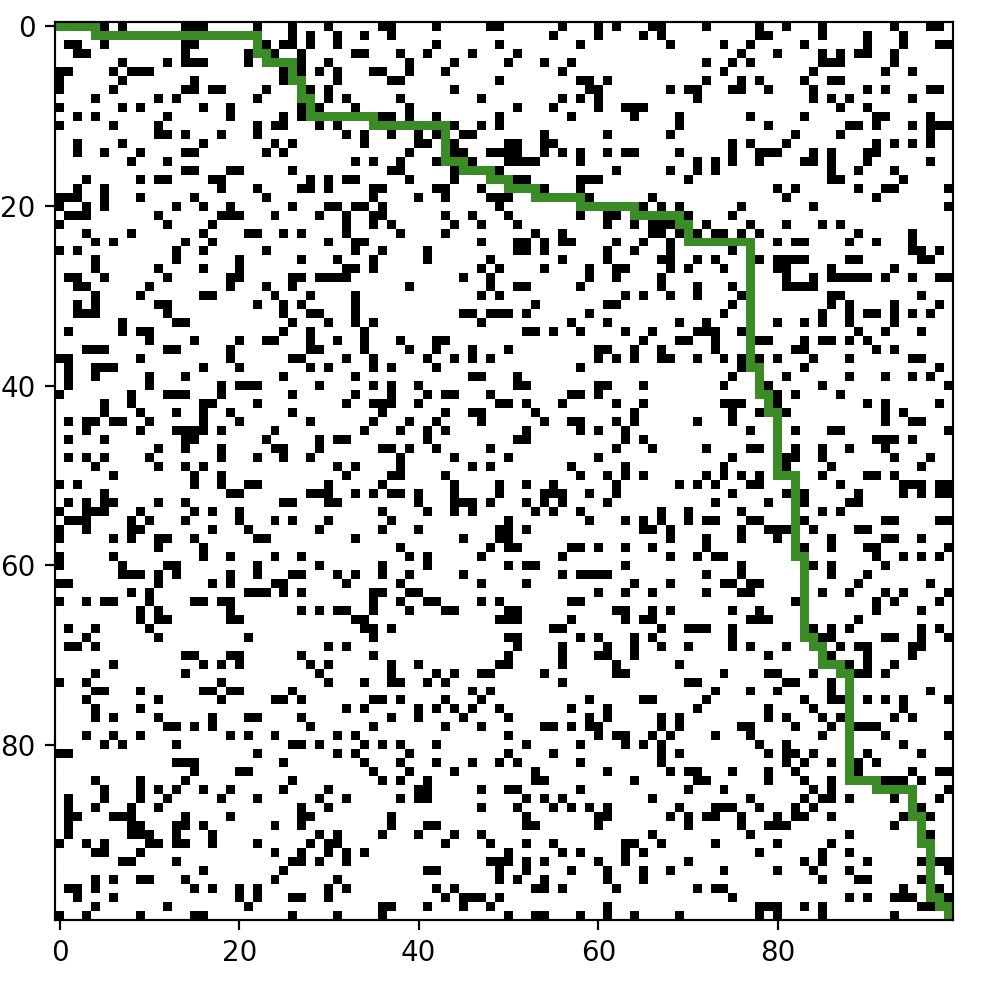
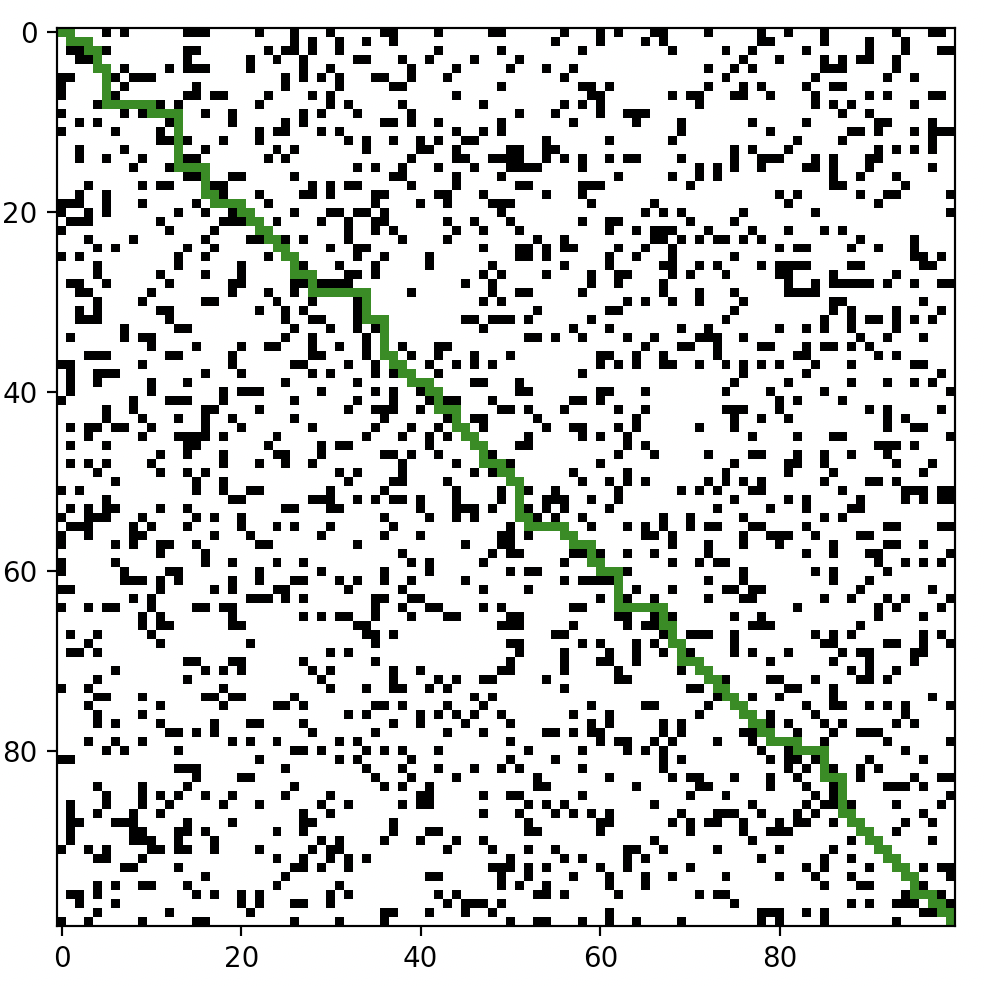
**DIM chosen: 100**

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

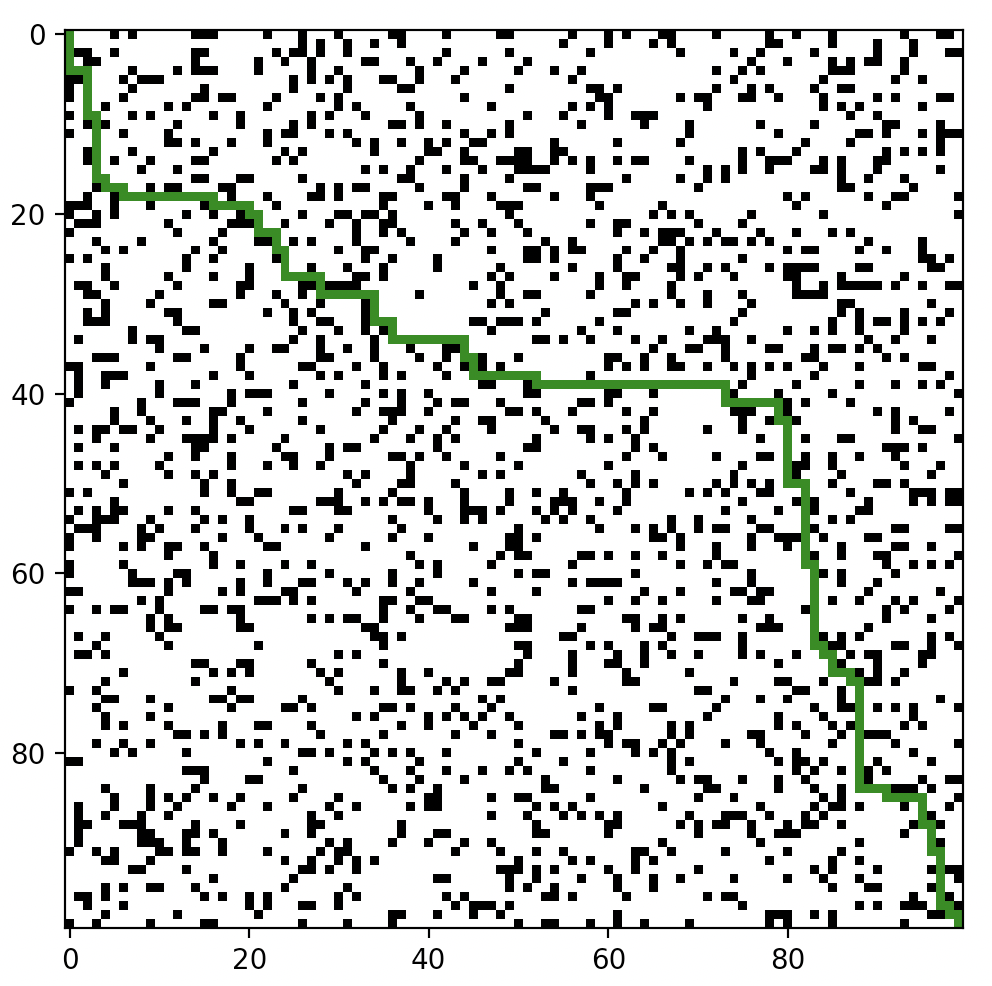
BFS DFS



A-Star Euclidean A-Star Manhattan

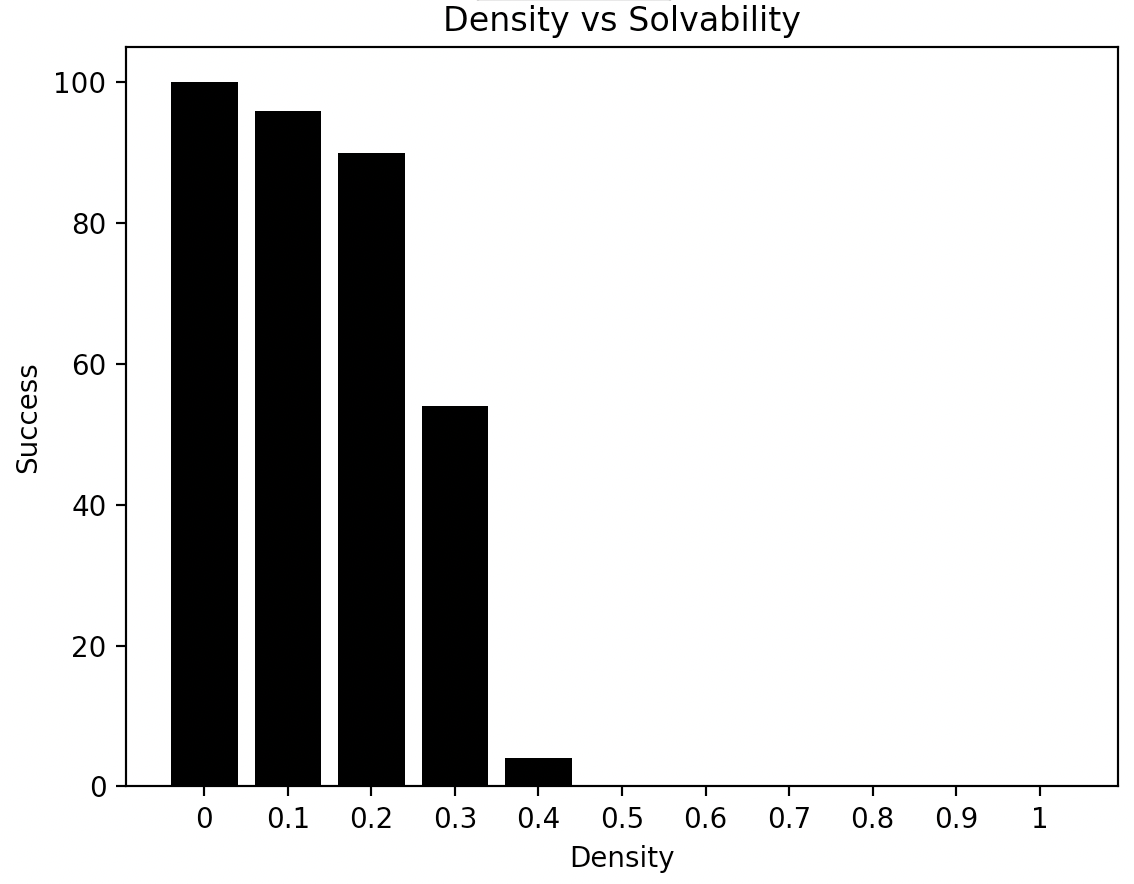


Bi-Directional BFS



The results for each of the algorithm do make sense. For BFS we see boomerang shape as it discovers nearly all cells 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.

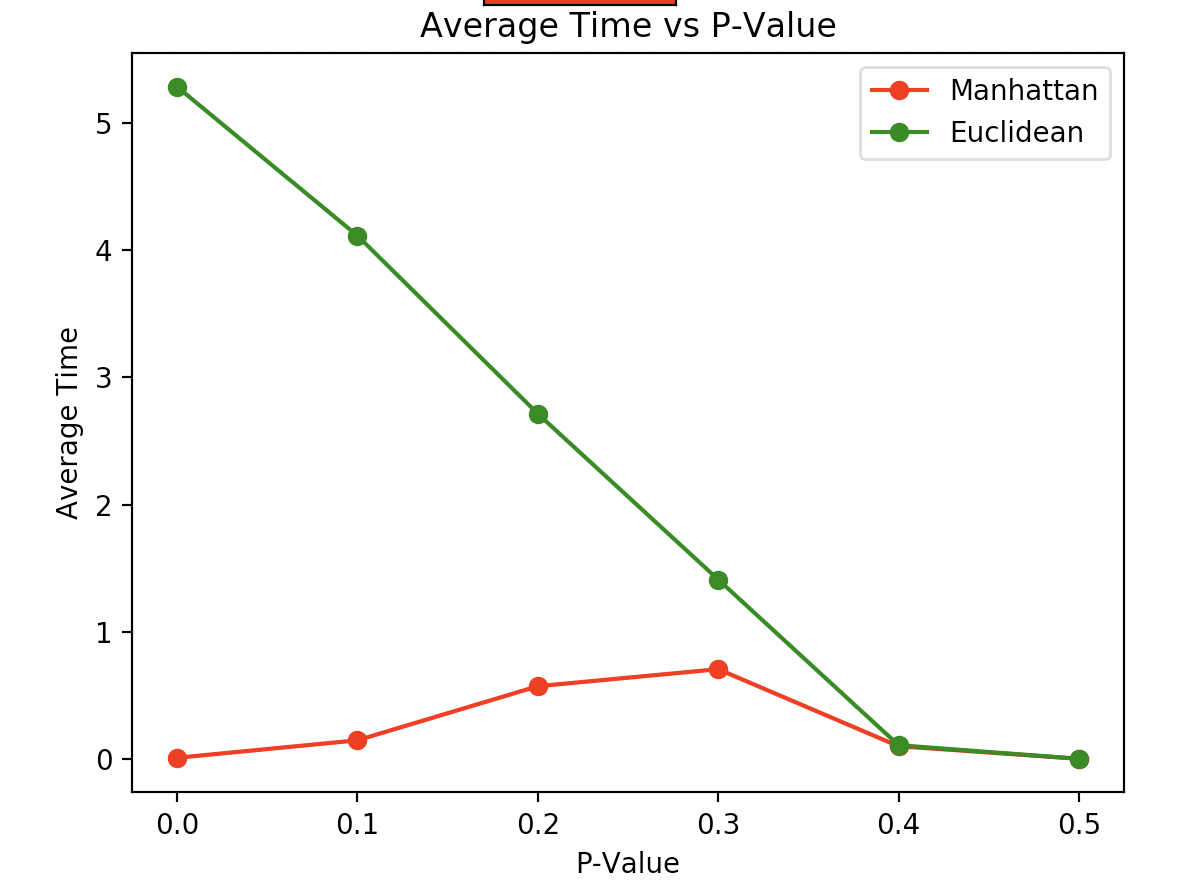
The threshold where p < p0 where most mazes are solvable, but p > p0 where most mazes are not solvable is most closely looking for a success rate of around 50%. From the graph we can clearly see p = 0.3 most closely resembles this

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.**

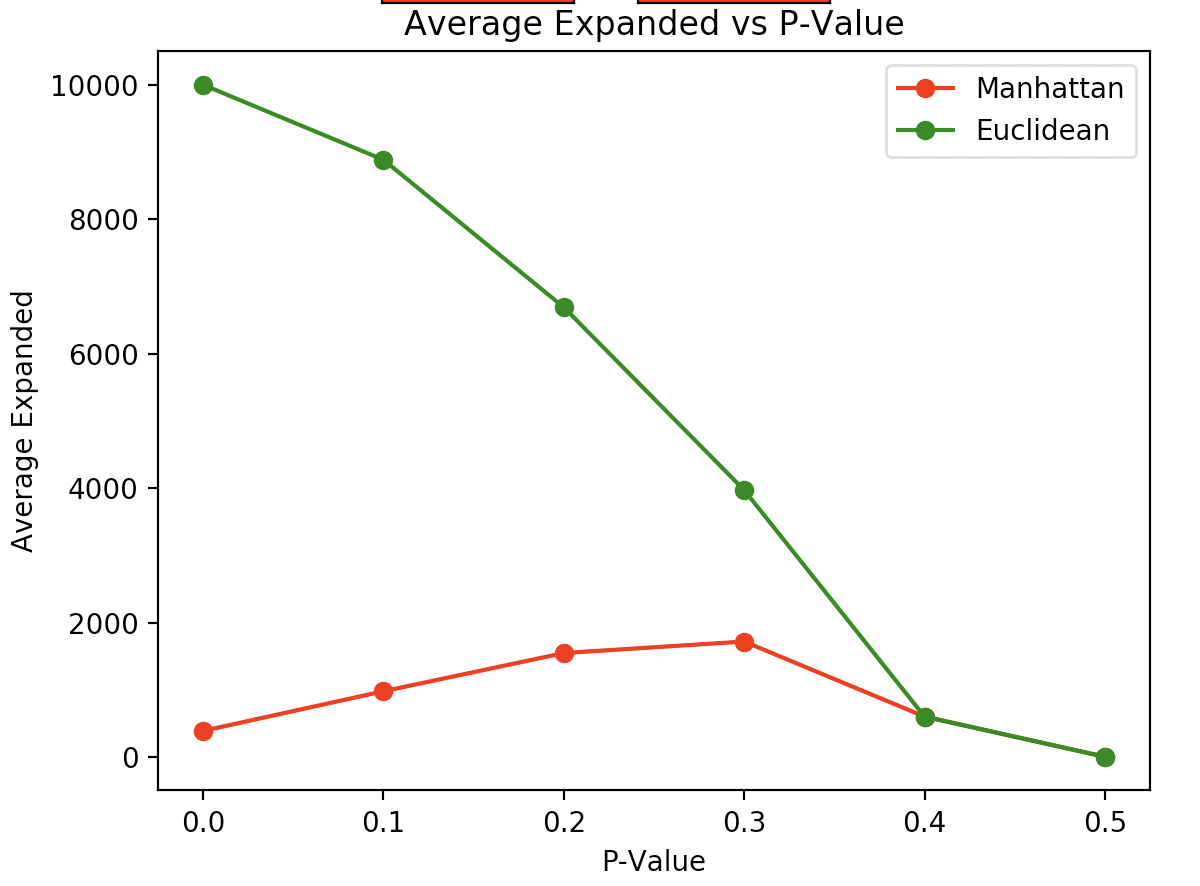
The small change of calculation in the heuristic between A\* Manhattan and A\* Euclidean can make a very large difference in the results. We know that both Manhattan and Euclidean are Admissible heuristics, meaning they are guaranteed to find the optimal path once reached the goal cell. So to compare them, we can pit them against each other in 3 separate ways:

1. The average time taken for completion
2. The number of cells expanded
3. Average max fringe size

**From part2/astar\_benchmark.py**



DIM = 100, Each P-Value ran 50 times



DIM = 100, Each P-Value ran 50 times

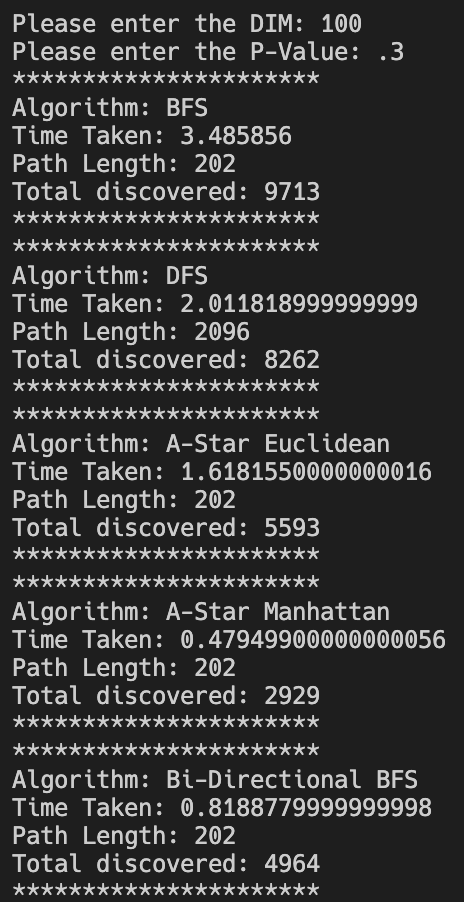


DIM = 100, Each P-Value ran 50 times

From the 3 charts we can conclude the following: A\* Manhattan takes less time and explores less cells, but it takes up more space than A\* Euclidean. If we were trying to optimize space efficiency, then we would say A\* Euclidean is better than A\* Manhattan. However, 2 out of the 3 competitions were won by A\* Manhattan, and in the cases for this project with extensive testing it would be much more beneficial to have a faster algorithm than a slower one. So, in conclusion, we can say that A\* Manhattan is generally better than A\* Euclidean.

**6. Do these algorithms behave as they should?**

Just looking at the generated paths from the mazes in part 1, we know that what we expect the maze to look like and what we get is pretty much exactly the same. However, we can quantify this data more by showing the data for each algorithm on the same maze: **Using part1.py**



From this data we can see that the algorithms are behaving as we expect them to. BFS taking the longest and expanding the most cells out of them all, but it does find the shortest path. DFS taking a long route and expanding nearly as much as BFS but taking less time since it brute forces its way down a branch and does not have to do much computational thinking compared to BFS. A\* Euclidean and A\* Manhattan behaving as we expected them to, same as with the last question. Manhattan runs faster and expands less cells than Euclidean. Bi-Directional BFS is also interesting, it takes a fraction of the time BFS takes and expands about half the number of cells BFS expands. This makes sense since we are starting BFS off in 2 different sides looking to meet each other. Because of this, BFS does not need to expand out as much as it would regularly resulting in less nodes needed to be explored and a faster run time.

**7. For DFS, can you improve the performance of the algorithm by**

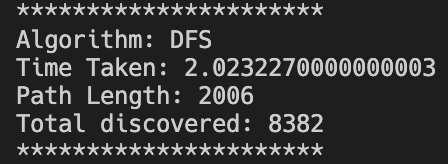
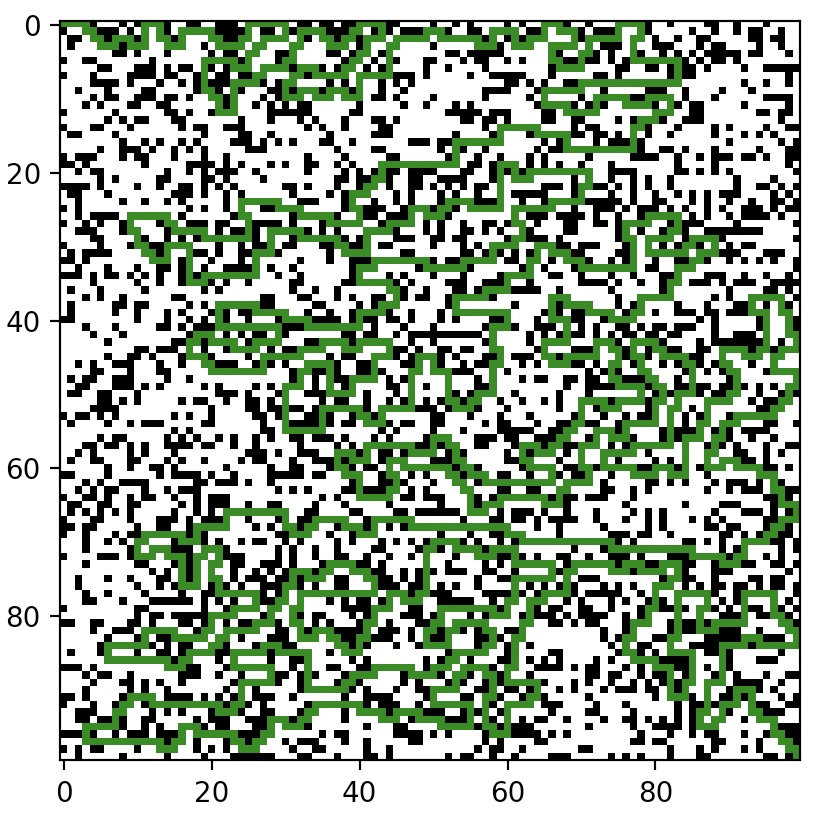
**choosing what order to load the neighboring rooms into the fringe?**

**What neighbors are ‘worth’ looking at before others? Be thorough and**

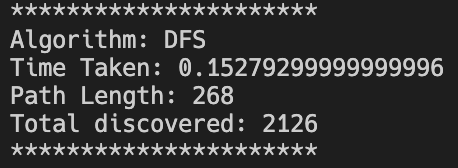
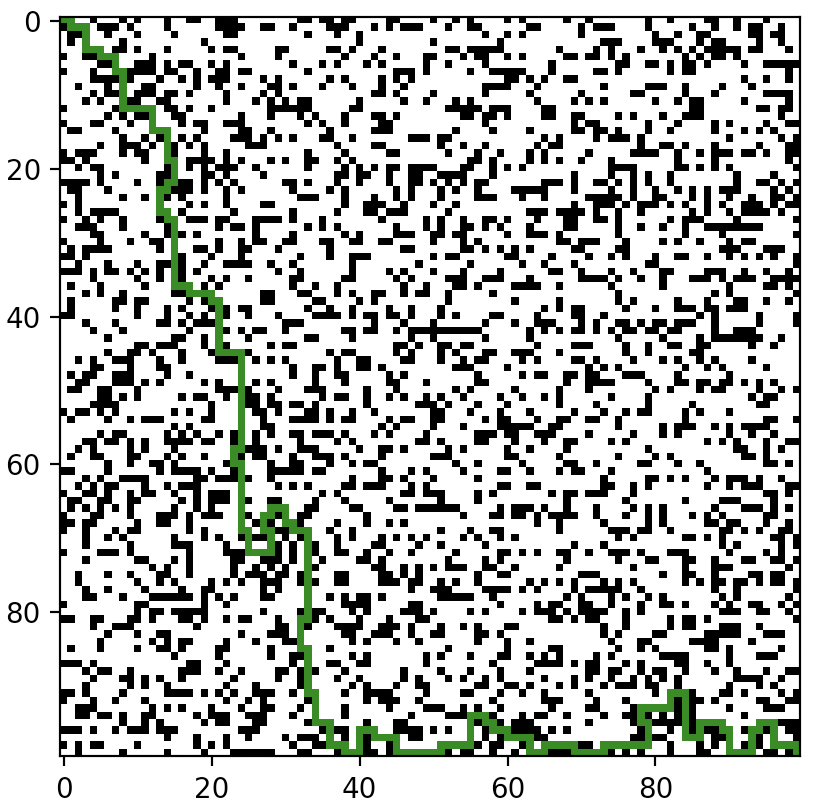
**justify yourself.**

Yes, we can absolutely improve the performance of the algorithm by choosing what order to load the neighboring rooms into the fringe. Our default algorithms all use the same ordering of Down, Right, Up, Left in that order. For this reason, DFS takes a very long path for the mazes.

What we are trying to accomplish is to get from the top left to the bottom right. As it is now, our DFS algorithm is loading Down, Right, Up, Left into the fringe (stack) in that order. Since this is a stack, what we are essentially doing is prioritizing looking Left first, then Up, then Right, then Down. This is the worst-case scenario for trying to go from the top left to the bottom right. If we were to switch the order the DFS checks the children to Left, Up, Right, Down then we will see improvements in every aspect of this algorithm. **Using part1.py**



DIM = 100, P = 0.3, DFS order Down, Right, Up, Left



DIM = 100, P = 0.3, DFS order Left, Up, Right, Down

As we can see, changing the load order of the neighbors can drastically change the performance of DFS. In terms of what neighbors are ‘worth’ looking at, it depends on where the goal is. If we started from the bottom left and the goal was the top right, then it would be worth looking and prioritizing cells that are up or right of the current position. We would not want to look at cells left and downwards because then we would be going back to where we started. Another thing to note is with DFS having optimal load order, it actually becomes the fastest algorithm we have. Before, A\* Manhattan was the fastest, but the optimal load order DFS beats it still, but still does not guarantee shortest path:

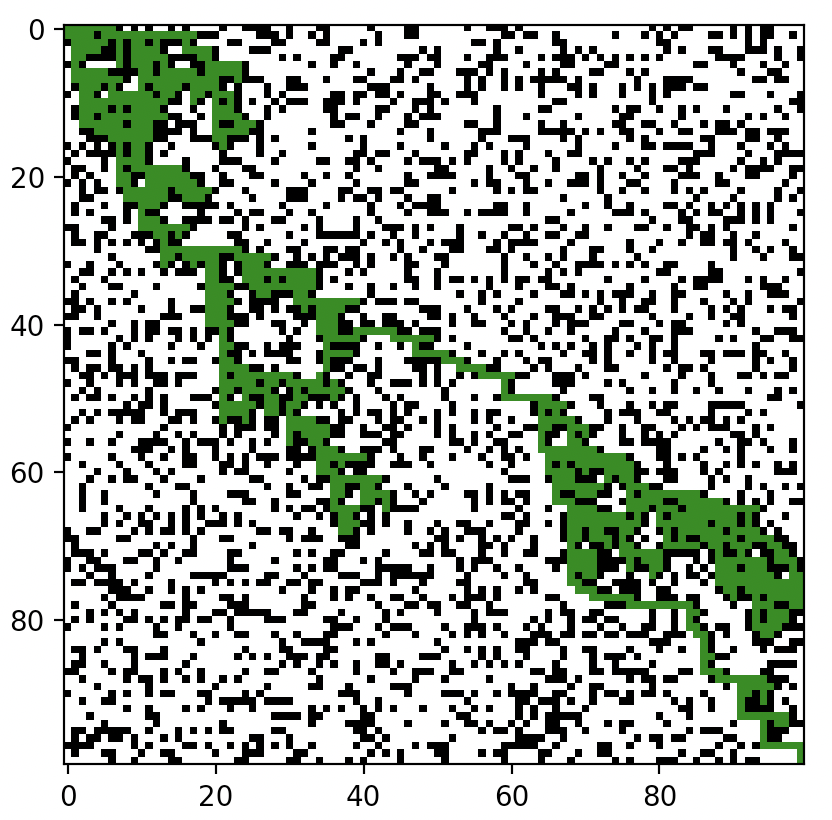
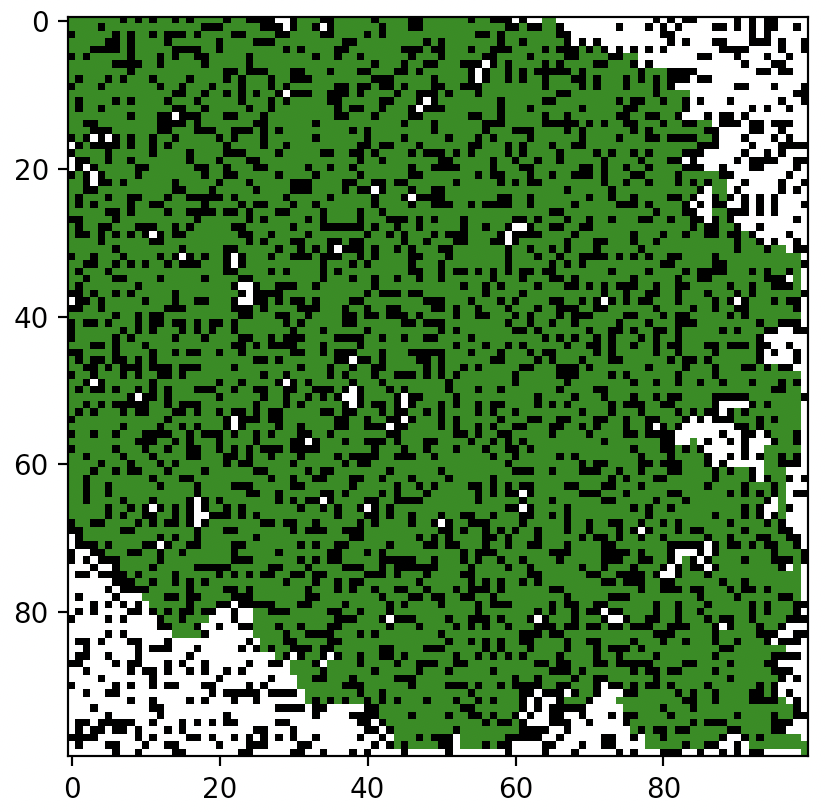


**8. On the same map, are there ever nodes that BD-BFS expands that A∗**

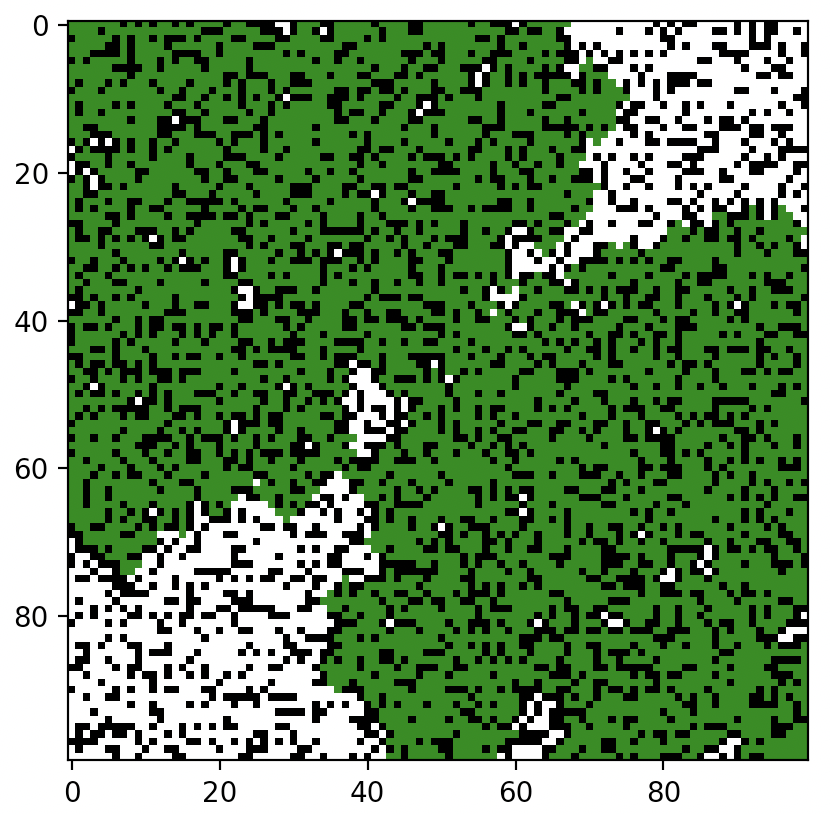
**doesn’t? Why or why not? Give an example, and justify.**

Given the same map, there are nodes that BD-BFS will expand that both A\* algorithms would not. For visual purposes, we can see that in the following captures: **Using part1.py**

A\* Euclidean A\* Manhattan



ALL 3: DIM = 100, P = 0.3



BD-BFS

At first glance it might be hard to see, but if you look on the edges of the BD-BFS compared to A\* you can see it expands nodes A\* does not. For instance, BD-BFS expands more nodes on the upper right side that A\* does not. This is due to the fact that BD-BFS is look at all nodes from both sides. A\* prioritizes based on a heuristic, so as it gets closer to the edges of the map it will not expand those nodes as there are almost guaranteed to be better available nodes nearer the goal. BD-BFS, however, will expand every node it sees until it reaches its goal. This means it would generally span out wider than A\* usually would, as well as explore nodes A\* would not.

**Bonus: How does the threshold probability p0 depend on dim? Be as precise**

**as you can.**

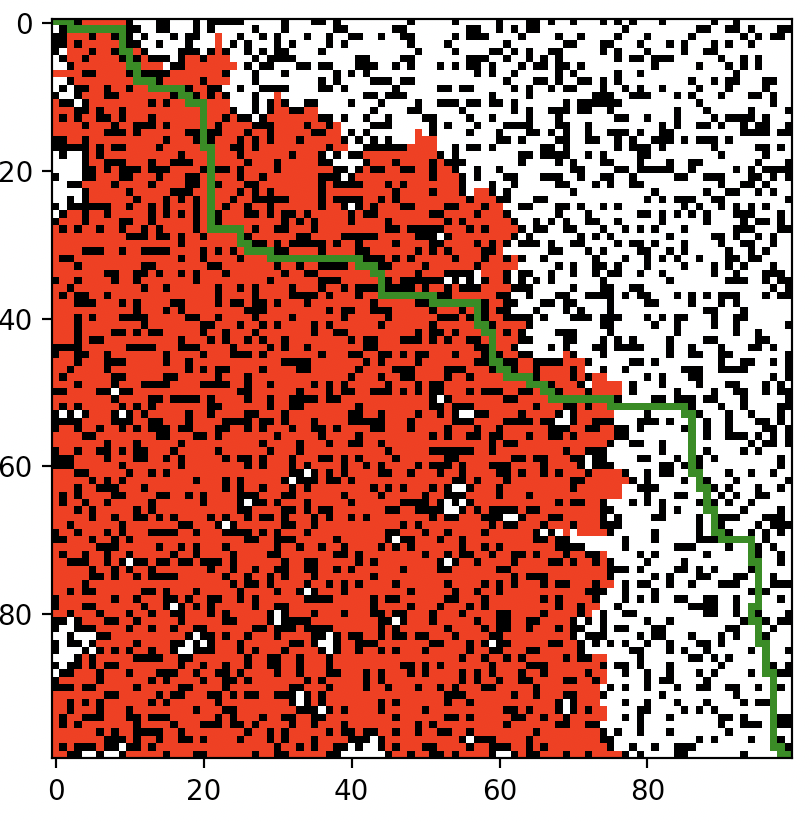
**Part 4:**

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 their 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.

Implementing strategy 1 yields:

A\* Manhattan



DIM = 100, P = 0.3, Q = 0.1

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:

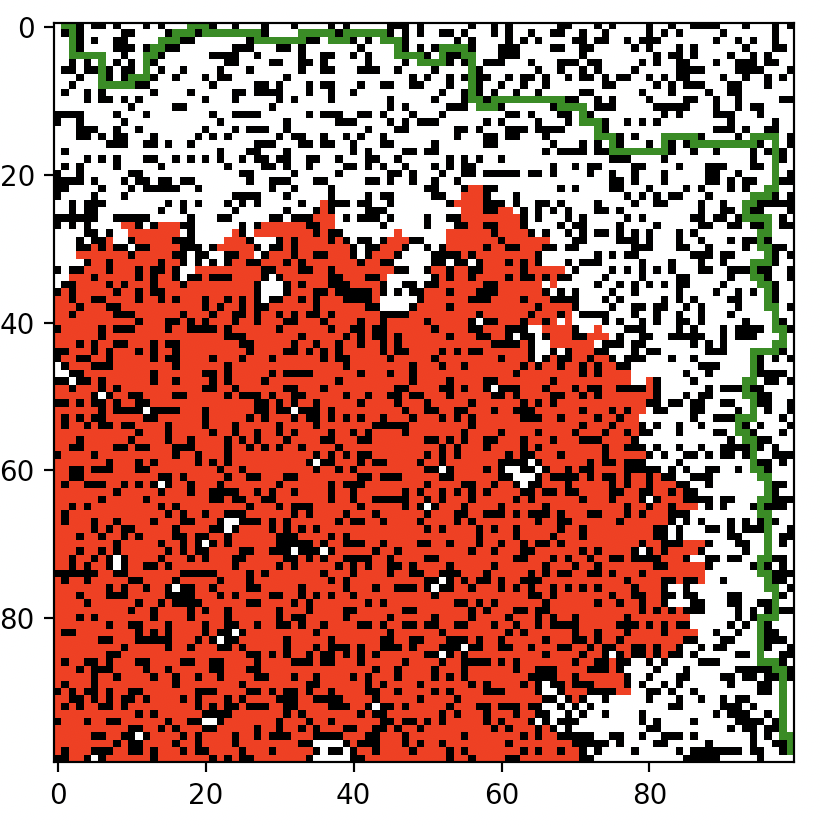
F = H – FireDistance

FireDistance = Euclidean Distance from current cell 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 cell 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 cell rather than going back near the start.

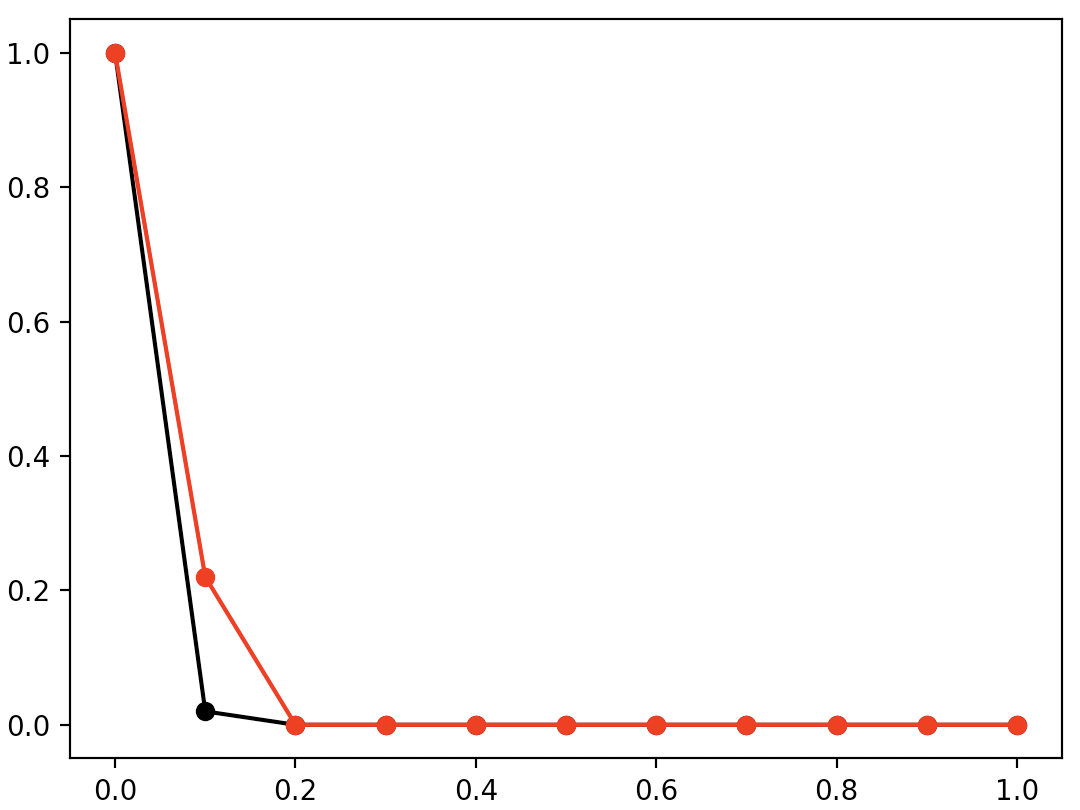
Implementing strategy 2 yields:

A\* Manhattan new heuristic



DIM = 100, P = 0.3, Q = 0.1

**Does re-computing your path like this have any benefit, ultimately?** Yes, it has an added benefit, albeit slightly. 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:



Black = Strategy 1

Red = Strategy 2

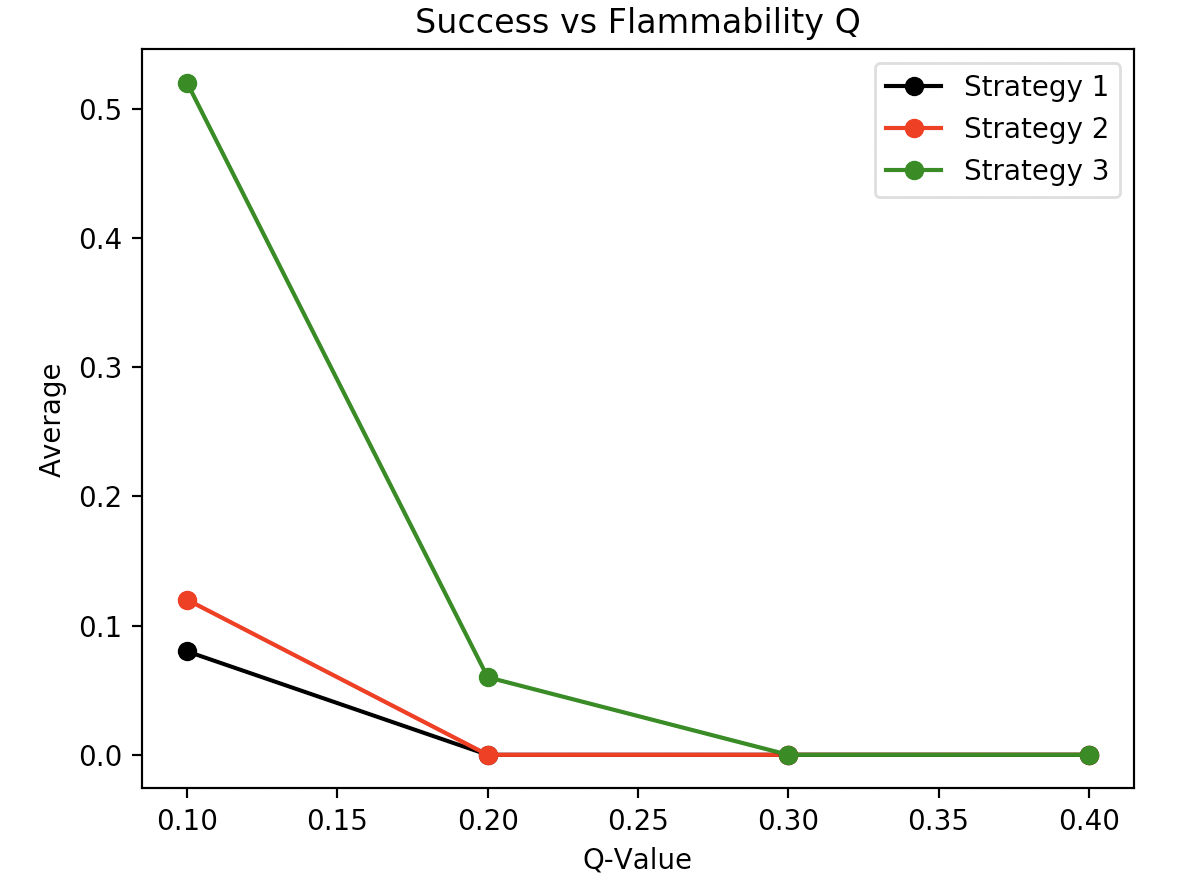
Average

Q-Value

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.

Looking at strategy 2, what it is essentially doing is similar to a min/max adversarial search. The priority queue ensures we are always choosing the min for each cell neighbor, however, there is a chance we can go back to a previous discovered cell which does not make it completely min/max. For making our own strategy, we looked at improving upon strategy 2. Strategy 2’s most fundamental flaw is that although it looks at the changing maze, it does not account for the future changes of the maze and thus can make moves it should not. For strategy 3 we aimed to implement this feature of predicting the future of the maze and acting accordingly. What we do is at every time step, we predict what the fire will look like all the way until it reaches close to our current cell. From there, we use the same heuristic as strategy 2 but now with the predicted future maze as the distance to the closest fire instead of the current maze. What this allows the algorithm to do is make decisions to choose neighbors based on a predicted future state of the maze making it choose safer options that strategy 2. The results:



DIM = 50, p = 0.3, Each Q ran 50 times on unique solvable mazes

As we can see, our new strategy blows the past 2 out of the water. We used DIM = 50 here because although our new strategy beats the past 2, it takes much longer to solve because predicting the future of the map takes time. We see that with Q = 0.1, we can solve the map over 50% of the time and can even solve a few maps at Q = 0.2. Past this though, the fire spreads extremely rapidly, and unless it spawns on the far edges of the maze it is unlikely it will ever be solved.