CS 440

Project 1: Maze Runner

Professor Wes Cowan

February 16, 2019

Krupal Patel (kp766)

Zain Siddiqui (zas30)

Sunehar Sandhu (sss311)

Part 1: Environments and Algorithms

Generating Environments:

Below are visualizations of the maze generating algorithm,

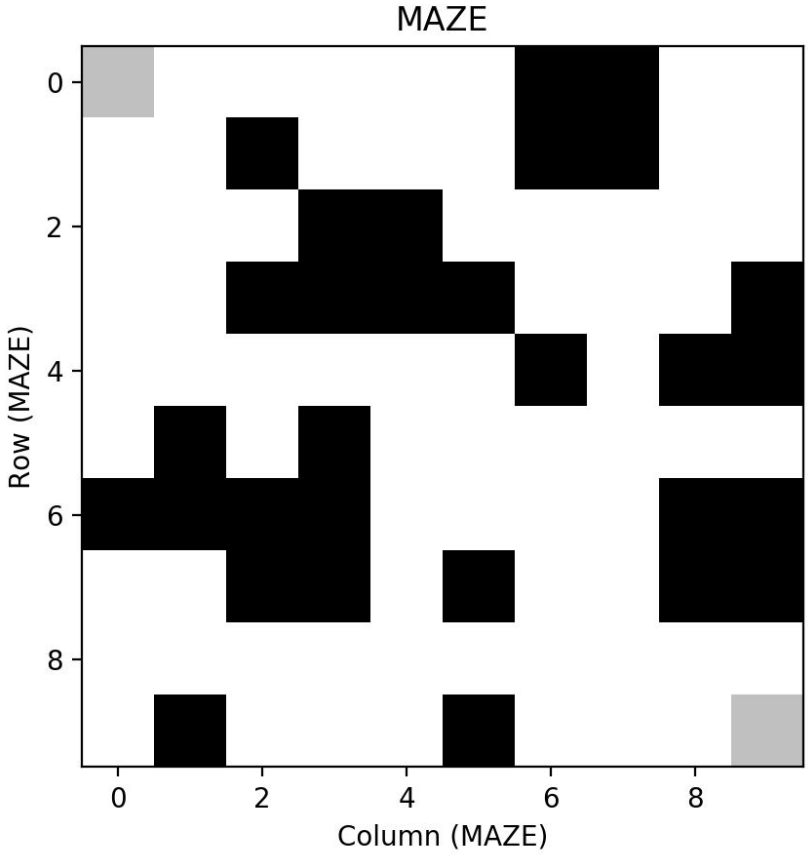


Figure: Map of dim = 10, p = 0.3

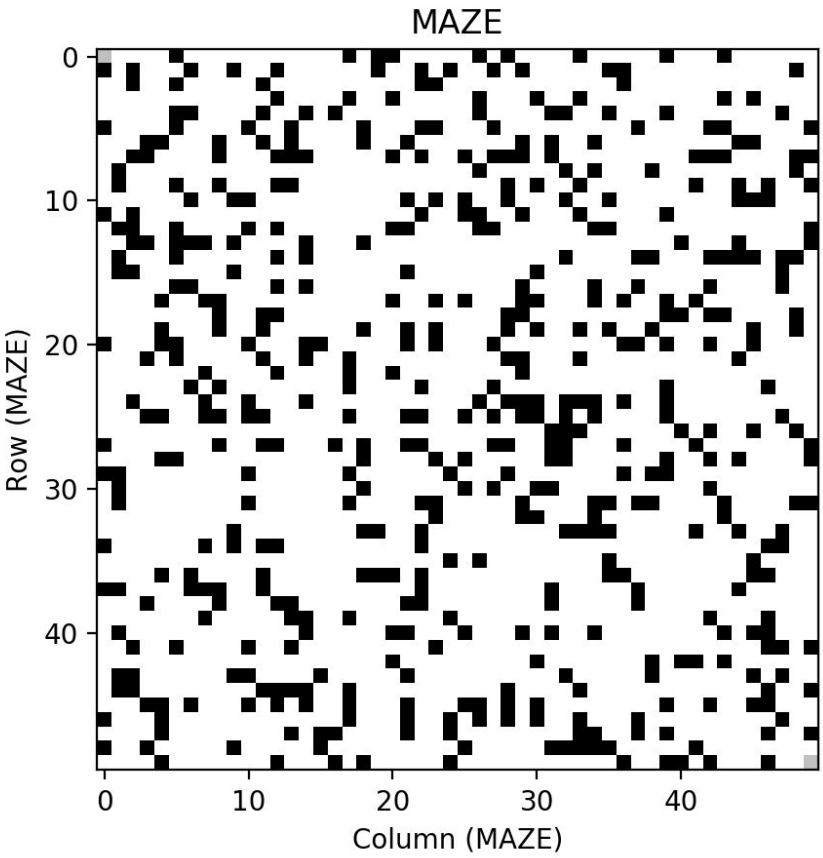


Figure: Map of dim = 50, p = 0.2

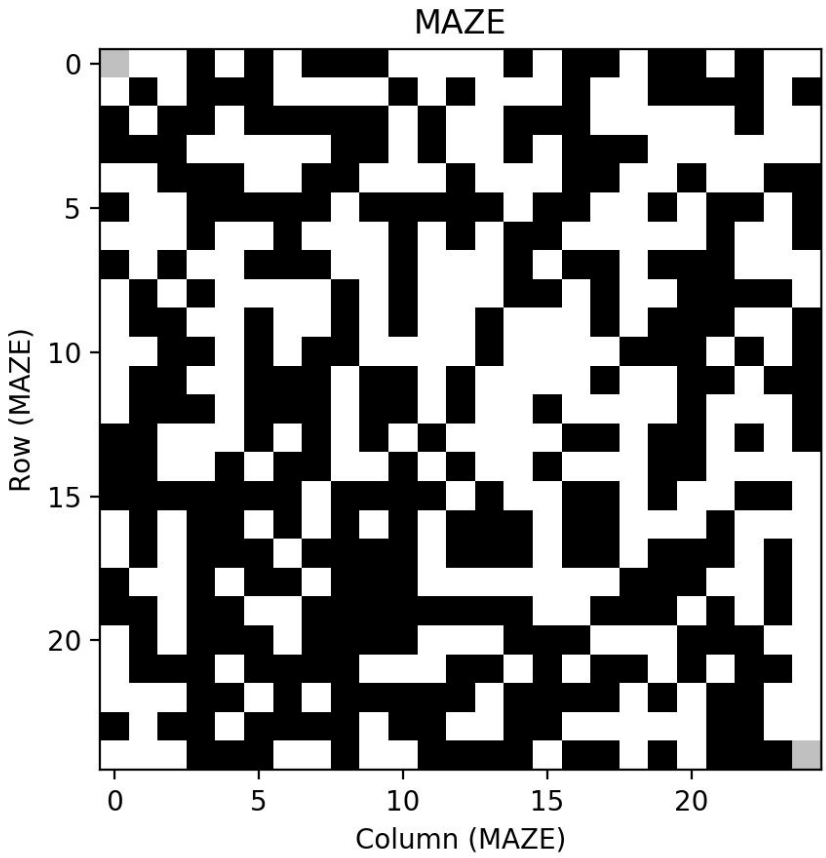


Figure: Map of dim = 25, p = 0.5

Path Planning:

Below are visualizations of the various path planning algorithms implemented, (Note: the grey cells are the agent's path, while the black cells are the filled cells)

Depth-First Search

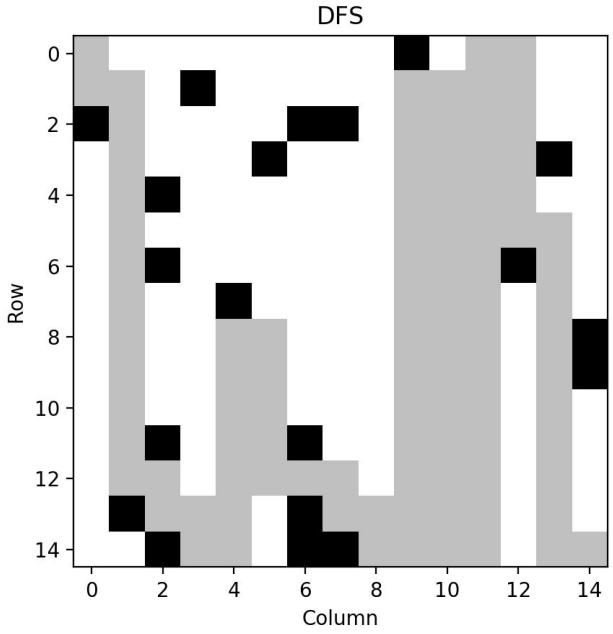


Figure: Solved map of dim = 15, p = 0.1

Breadth-First Search

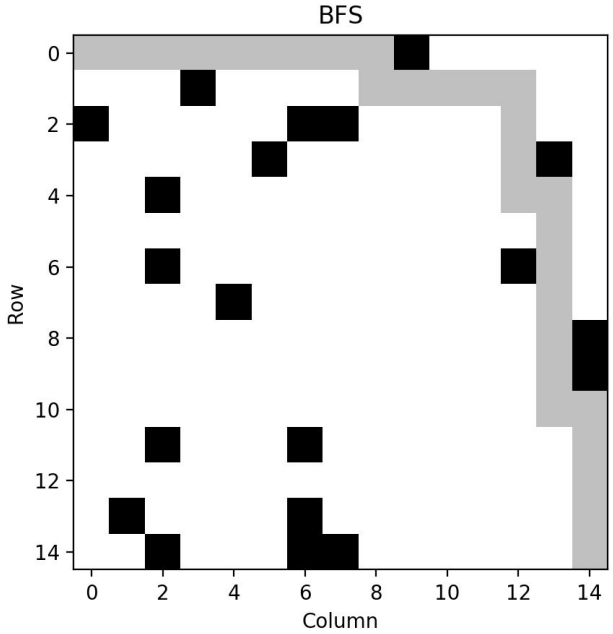


Figure: Solved map of dim = 15, p = 0.1

A* with Euclidean Heuristic

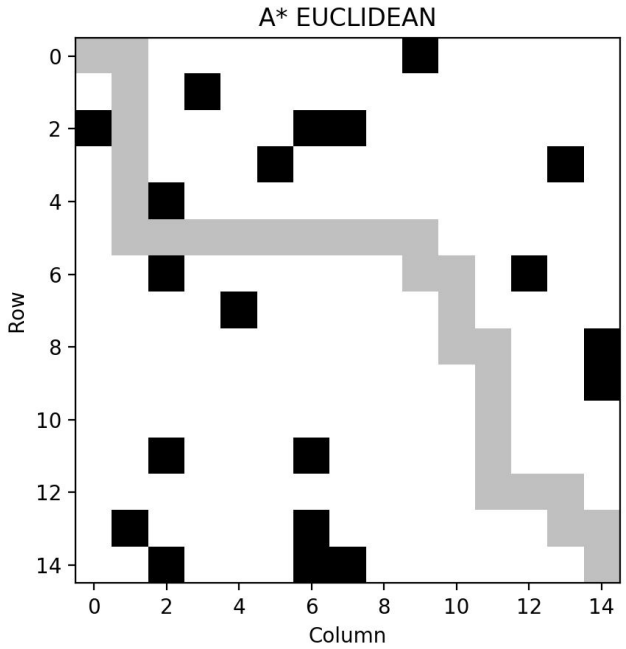


Figure: Solved map of dim = 15, p = 0.1

A* with Manhattan Heuristic

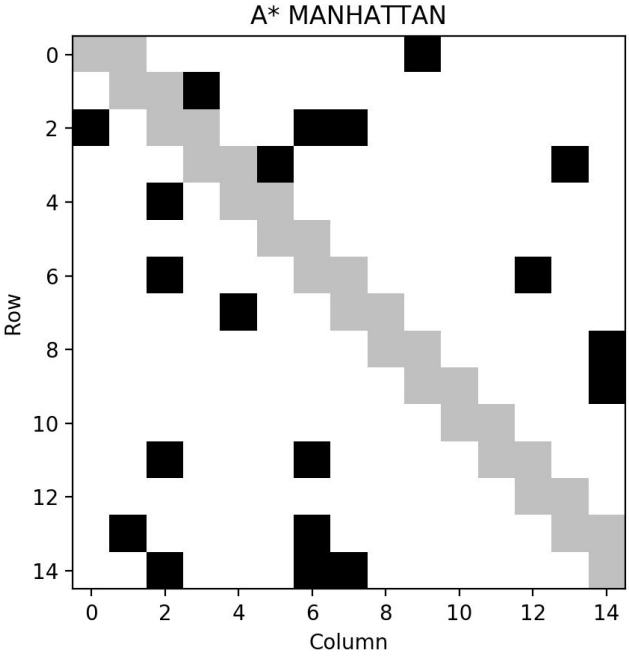


Figure: Solved map of dim = 15, p = 0.1

Bi-Directional Breadth-First Search

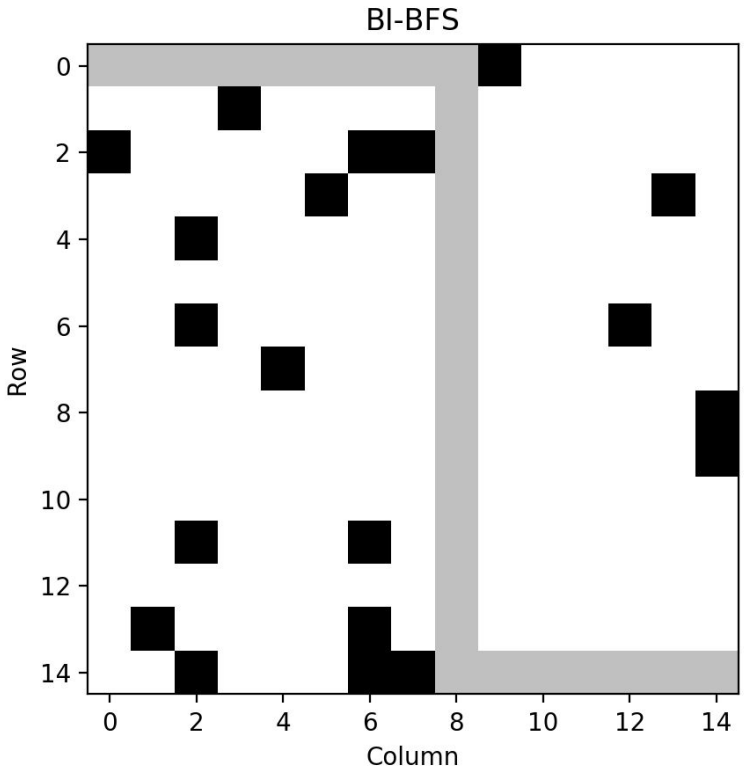
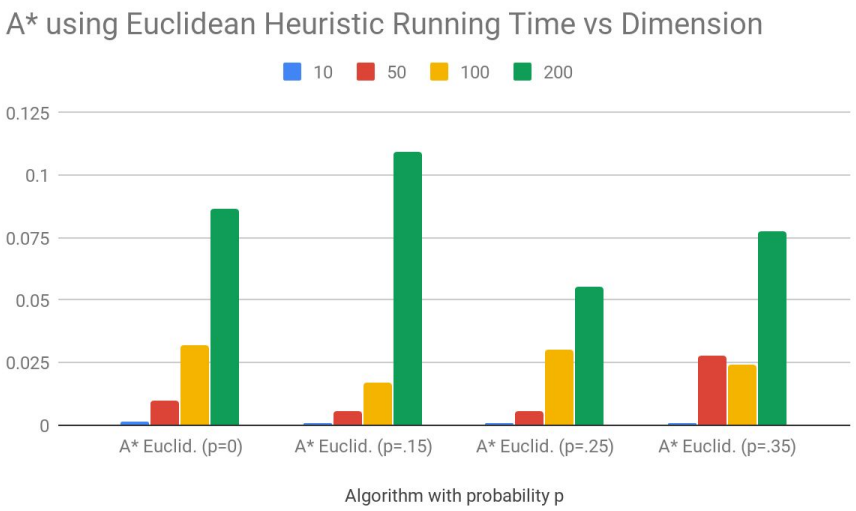


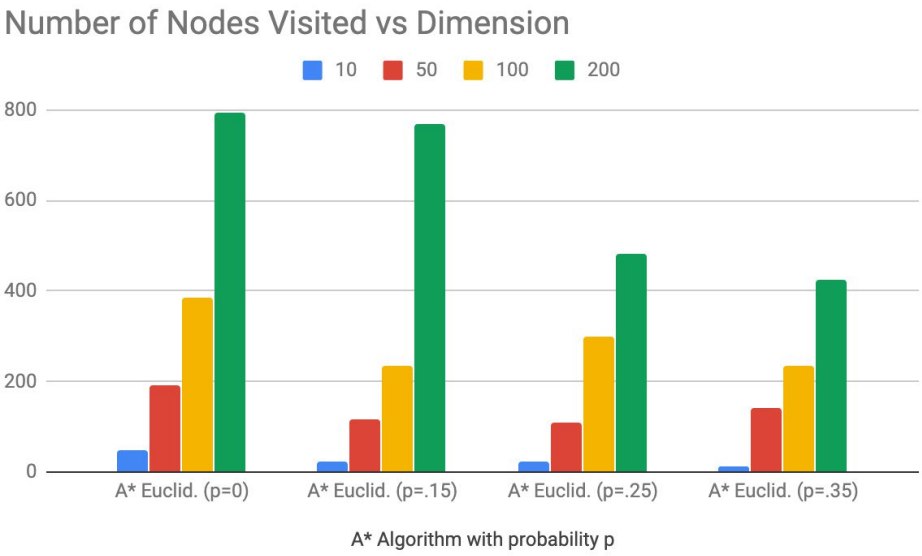
Figure: Solved map of dim = 15, p = 0.1

Part 2: Analysis and Comparison

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?

   In order to find the optimal size (dim) of a map, we must take into account two factors,
   1. The execution time for the algorithm in order to find the complete path, as the running time of the algorithm determines whether it can be run multiple times efficiently.
   2. The number of nodes explored in order to find the complete path as the more nodes an algorithm explores the more work is required to find a valid path from start to goal.





We need an algorithm that explores relatively fewer nodes and executed in a shorter time so that the function can efficiently compute the path even for higher dimensions. Out of all the algorithms we wrote, the A* algorithm runs optimally, hence, we have chosen this algorithm for this part.

The map of size dim = 200 seems to produce a steady amount of work with a somewhat consistent amount of time to solve. Thus, picking this dimension will allow us to repeatedly produce maps for a range of possible p values {0, 0.15, 0.25, 0.35}.

2. For p ≈ 0.2, generate a solvable map, and show the paths returned for each algorithm. Do the results make sense? (ASCII printouts are fine, but good visualizations are a bonus)

Below are map visualizations of size dim = 20 and p = 0.2

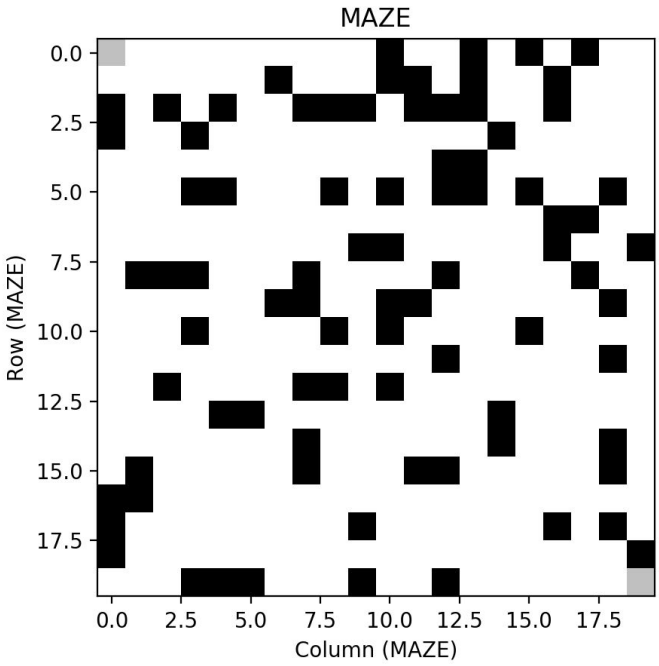


Figure: Unsolved map of dim = 20, p = 0.2

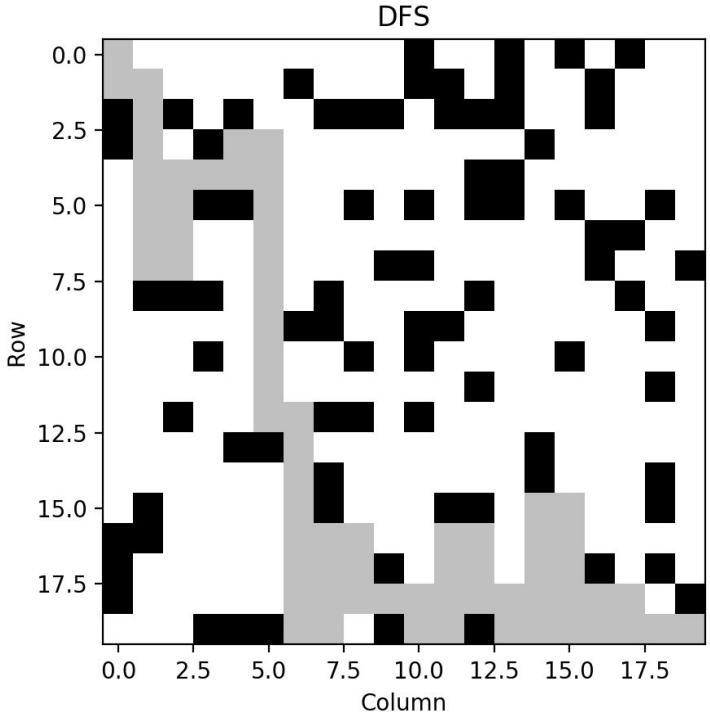


Figure: Solved map of dim = 20, p = 0.2

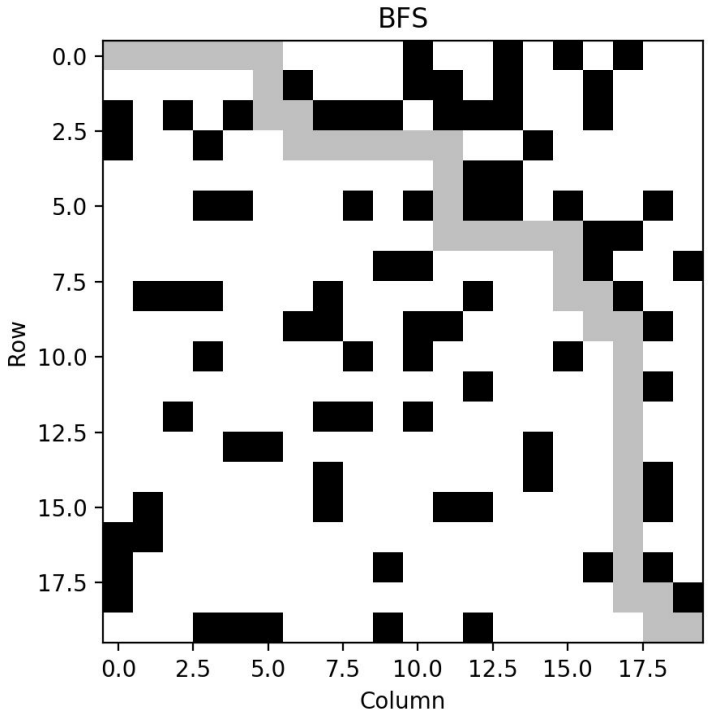


Figure: Solved map of dim = 20, p = 0.2

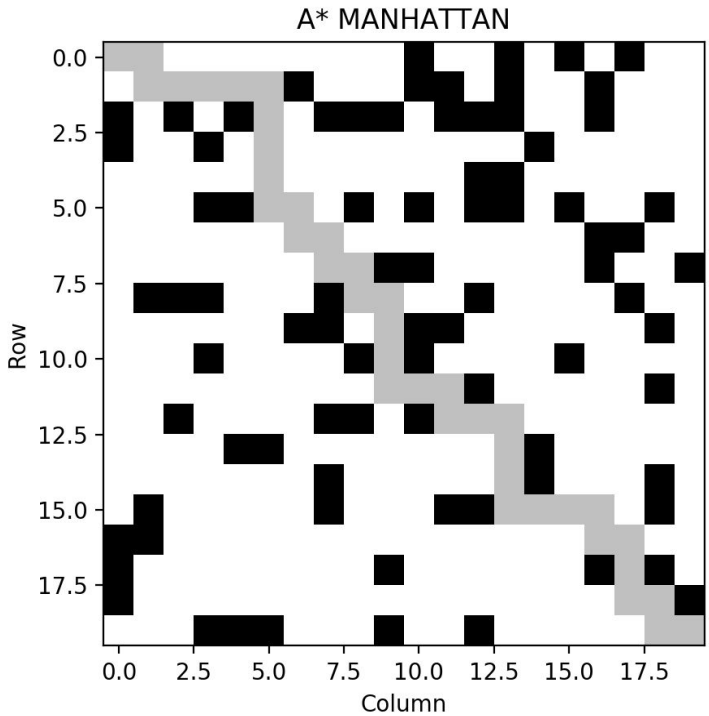


Figure: Solved map of dim = 20, p = 0.2

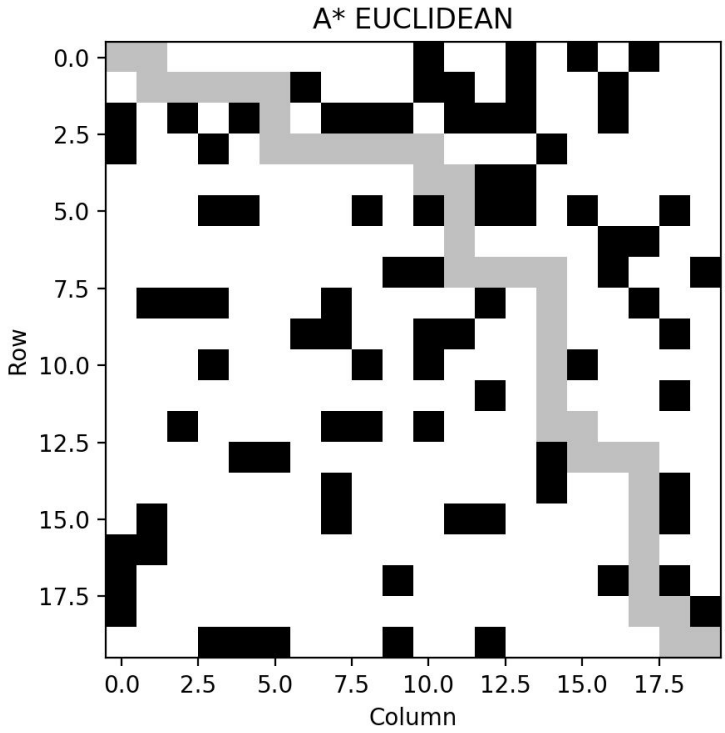


Figure: Solved map of dim = 20, p = 0.2

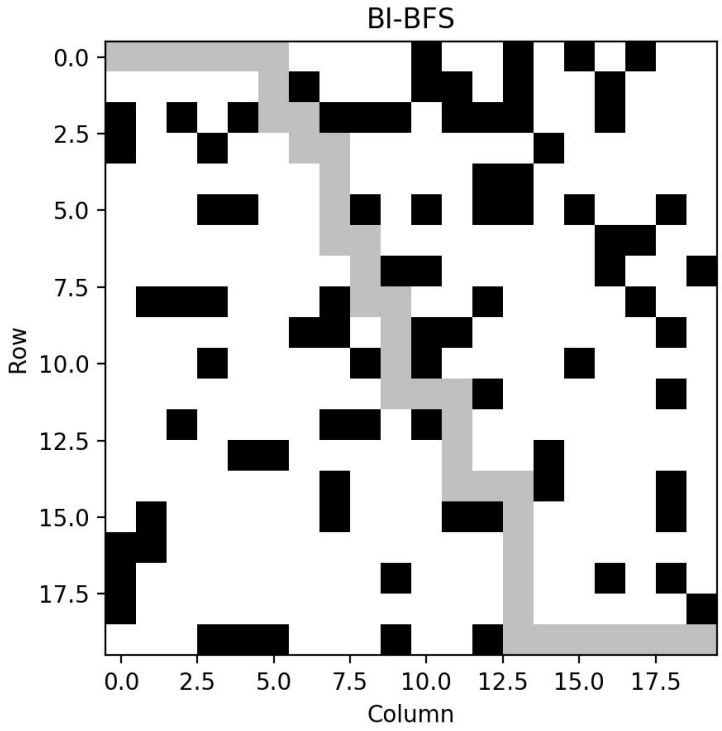
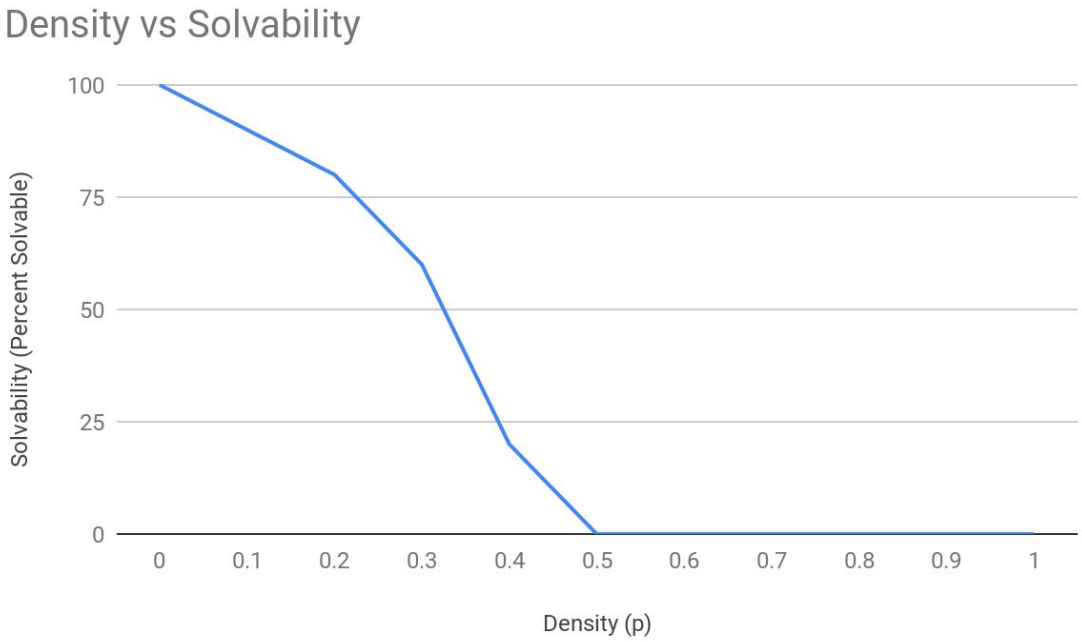


Figure: Solved map of dim = 20, p = 0.2

3. 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.

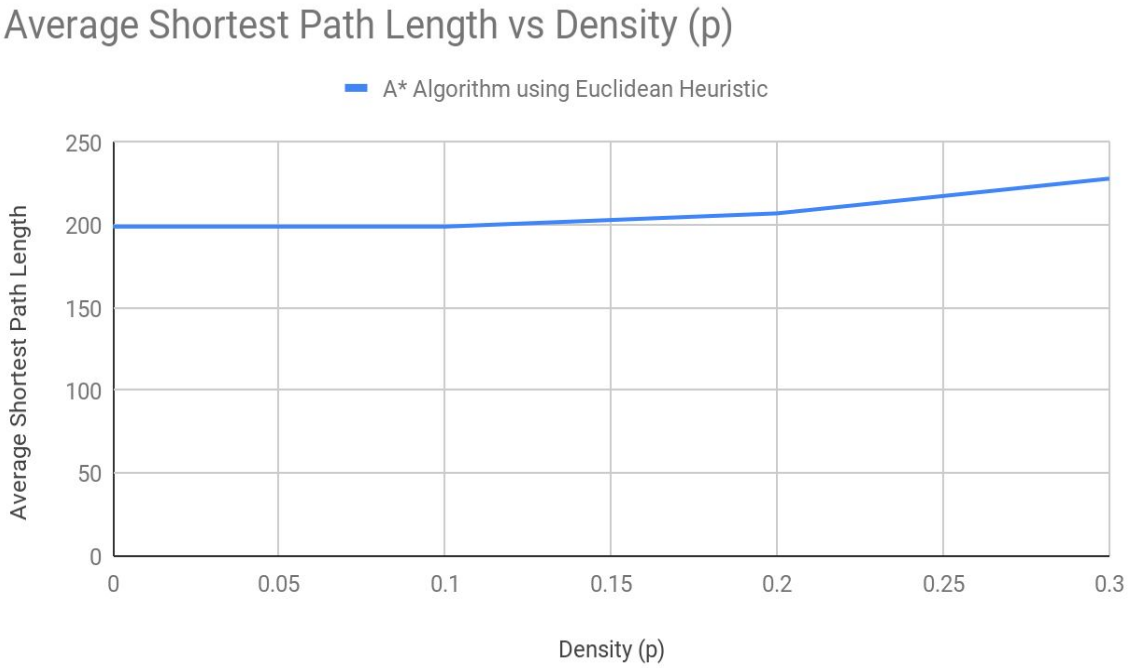
In order to see how maze-solvability depends on p, we can generate a map of dim = 100 with a range of p values consisting of {0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1}. For a particular p value, we will run the algorithm 50 times and count the number of solvable maps. The best algorithm to use in this instance is an algorithm that solves maps in the least amount of time. We know that BFS finds a path from one point to another the quickest out of all algorithms. However, Bi-Directional BFS is even faster, as it essentially runs two BFS from each point simultaneously, which in turn decreases the number of nodes that need to be explored. Thus, we can use Bi-Directional BFS for generating the Density vs Solvability plot:



Based on the data from the plot, it appears that most maps are solvable, with solvability > 50%, with p <= 0.3. With maps of p > 0.3, we see that the solvability of the map drops dramatically. Thus, we can choose $p_0 = 0.3$.

4. 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?

In order to estimate the average or expected length of the shortest path from start position to goal position, we generated 100 random mazes of dim = 100 with p <= 3 and then computed the average for every p value. Thus, we can plot the Average Shortest Path Length vs Density:



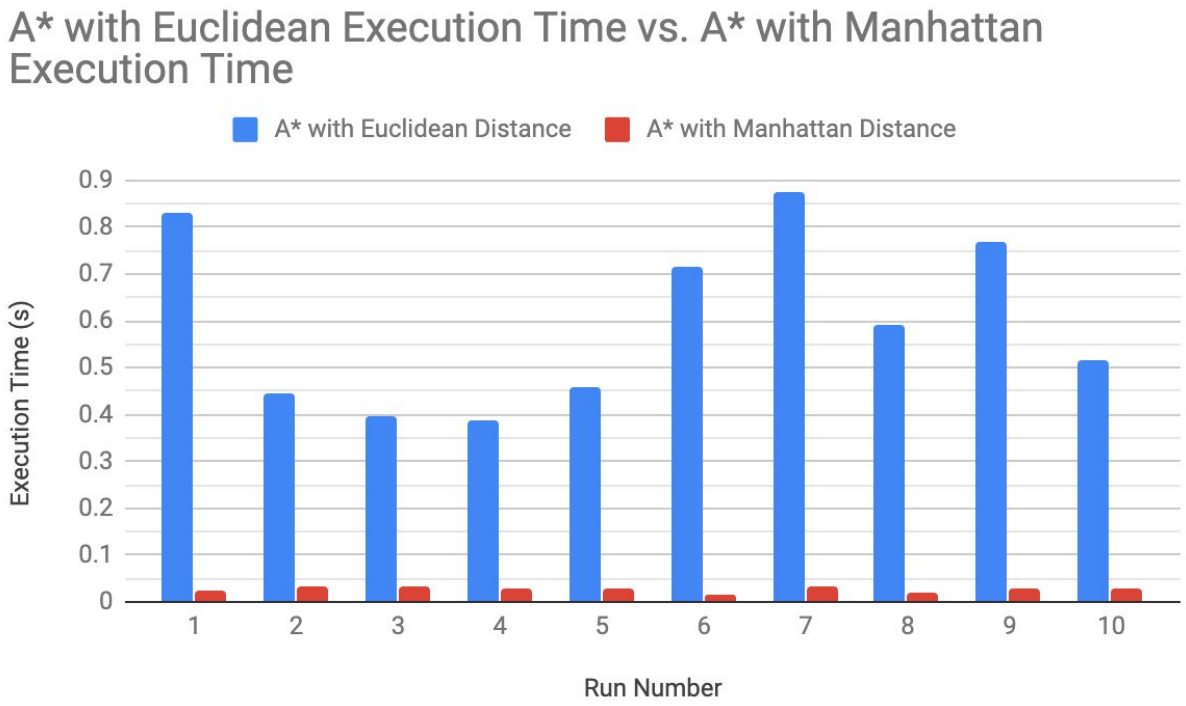
The algorithm most useful for this task is the A* Algorithm, as it is optimal which suggests that it will find the shortest path from start to finish, if a shortest path exists. We can observe that the average shortest path length remains almost the same for p <= 0.1. For p > 0.15, we can see the length of the path begin to increase.

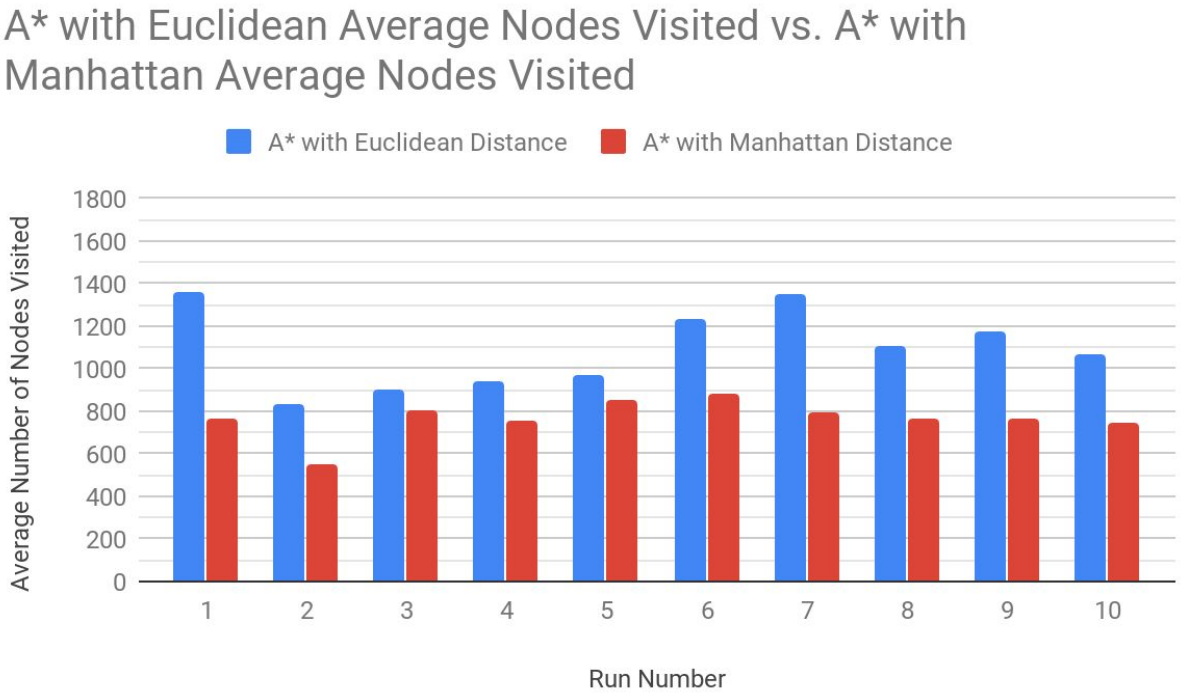
5. Is one heuristic uniformly better than the other for running A*? How can they be compared? Plot the relevant data and justify your conclusions.

The two heuristic functions we have implemented for the A* algorithm consists of the Manhattan distance and Euclidean distance. The Manhattan distance is the distance between two points on a grid based on a vertical and horizontal path. However, the Euclidean distance is the straight-line distance between two points on a grid. The context heavily determines which heuristic will be better for running A*. In this map environment, we can only move up, down, left, and right which means that the Manhattan distance will be a better heuristic as it accounts for vertical and horizontal path calculations. In the case that we could also move diagonally, then the Euclidean distance would be a better heuristic as it accounts for the straight-line distance between two points.

The heuristics can be compared by plotting their average execution time along with the average number of nodes visited when running A* a total of 10 times on a map of dim = 100 and p = 0.15,

	Average Execution Time (s)	Average Number of Nodes Visited
A* with Euclidean Distance	0.5979343176	1093.2
A* with Manhattan Distance	0.02691214085	767.6





From the multiple visualizations, we can see that A* with Manhattan distance is a better heuristic. The execution time of the algorithm for the Manhattan heuristic is significantly faster than with the Euclidean heuristic. Furthermore, the algorithm with the Manhattan heuristic explored less nodes throughout all the trial runs. Thus, we can see that the Manhattan heuristic provides us with a greater approximate estimate, which in turn causes overall less nodes to be put into the priority queue and later expanded. As a result, based on the given context and dynamics of our map environment, the Manhattan distance is a better heuristic for finding an optimal path with efficient use of execution time and space.

6. Do these algorithms behave as they should?

In order to see if the algorithms are behaving as they should, we can compare their execution time and path length for a given dim = 150 and p = .15 with their expected behavior.

Algorithm:	Execution Time	Path Length	Max Size of Fringe
DFS	0.1090407372	969	865
BFS	1.08996892	299	188
A* Manhattan	0.10708498	163	234
A* Euclidean	0.1239228249	163	348
Bidirectional-BFS	0.0879609108	299	139

According to the data, we can see that DFS is not optimal, as its path length is much greater than every other algorithm, this is because DFS visits significantly more cells. However, DFS is known to be more efficient computationally, which is evident in its relatively short execution time compared to the other algorithms. Furthermore, both A* algorithms found the most optimal path of length 163, which is the shortest path from start cell to goal cell. Also, we can observe that A* using the Manhattan heuristic executes faster than with the Euclidean heuristic as expected within the context of this environment. Lastly, we see that Bidirectional-BFS is expected to be theoretically faster as it runs two BFS simultaneously. Thus, we see that the algorithm runs significantly faster than normal BFS.

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.

Instead of having DFS choose which neighbors to load into the fringe systematically or randomly, we can incorporate a heuristic to make desirable decisions first. Hence, we can use the Manhattan distance as a heuristic to load neighboring cells into the fringe, as it accounts for vertical and horizontal path calculations which are path movements our environment supports. By loading the neighboring cells into the fringe by the size of the heuristic in increasing order, the performance of DFS can be significantly improved. This is because the heuristic will give an estimate for the length of the path from one position to the goal. Thus, we can make desirable decisions by loading cells with smaller heuristics to find a path faster. Lastly, by including heuristics, the algorithm no longer makes uninformed decisions which will eliminate the chance that the algorithm will roam undesirable cells.

8. On the same map, are there ever nodes that BD-DFS expands that A* doesn't? Why or why not? Give an example, and justify.

In order to conclude whether Bidirectional-BFS expands cells that A* does not, we must understand that Bi-directional BFS conducts a BFS from the start cell to the goal cell and another BFS from the goal cell to the start cell simultaneously. Fundamentally, BFS explores all cells in until the goal cell is found. In contrast, A* uses heuristics to find a path from start cell to goal cell. These heuristics allow A* to estimate which cells are better than others, and to explore those desirable cells. Thus, we can see, by nature, that Bidirectional-BFS can expand cells that A* will overlook. We can solidify this statement by looking at the paths returned from both algorithms on a given map of dim = 20 and p = 0.1,

A* with Euclidean Heuristic:
[(0, 0), (0, 1), (0, 2), (1, 2), (1, 3), (1, 4), (2, 4), (3, 4), (4, 4), (4, 5), (5, 5), (5, 6), (6, 6), (6, 7), (7, 7), (7, 8), (8, 8), (8, 9), (9, 9), (10, 9), (10, 10), (10, 11), (11, 11), (11, 12), (11, 13), (12, 13), (13, 13), (13, 14), (14, 14), (14, 15), (15, 15), (15, 16), (15, 17), (16, 17), (17, 17), (17, 18), (18, 18), (18, 19), (19, 19)]
a* euclidean path length: 39

Bi-directional BFS:
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 9), (0, 10), (0, 11), (0, 12), (0, 13), (0, 14), (0, 15), (0, 16), (1, 16), (2, 16), (3, 16), (4, 16), (5, 16), (6, 16), (7, 16), (8, 16), (9, 16), (9, 17), (9, 18), (10, 18), (11, 18), (12, 18), (13, 18), (14, 18), (15, 18), (16, 18), (17, 18), (18, 18), (19, 18), (19, 19)]
bi-bfs path length: 39

Cells explored by Bi-directional BFS but not with A* with Euclidean Heuristic:
[(0, 12), (0, 14), (0, 7), (0, 16), (7, 16), (0, 10), (9, 18), (0, 3), (19, 18), (0, 15), (1, 16), (14, 18), (10, 18), (0, 11), (9, 17), (15, 18), (0, 4), (3, 16), (11, 18), (8, 16), (16, 18), (2, 16), (4, 16), (12, 18), (9, 16), (0, 5), (0, 8), (5, 16), (0, 13), (0, 6), (13, 18), (0, 9), (6, 16)]

As we can see, even though the length of the path returned by both algorithms is 39, Bi-directional BFS explores cells that A* does not due to their fundamental differences explained previously.

Bonus: How does the threshold probability p0 depend on dim? Be as precise as you can.

In order to see the relationship, we must plot Solvability vs Density plots for a variety of dimensions as well as a range of densities.

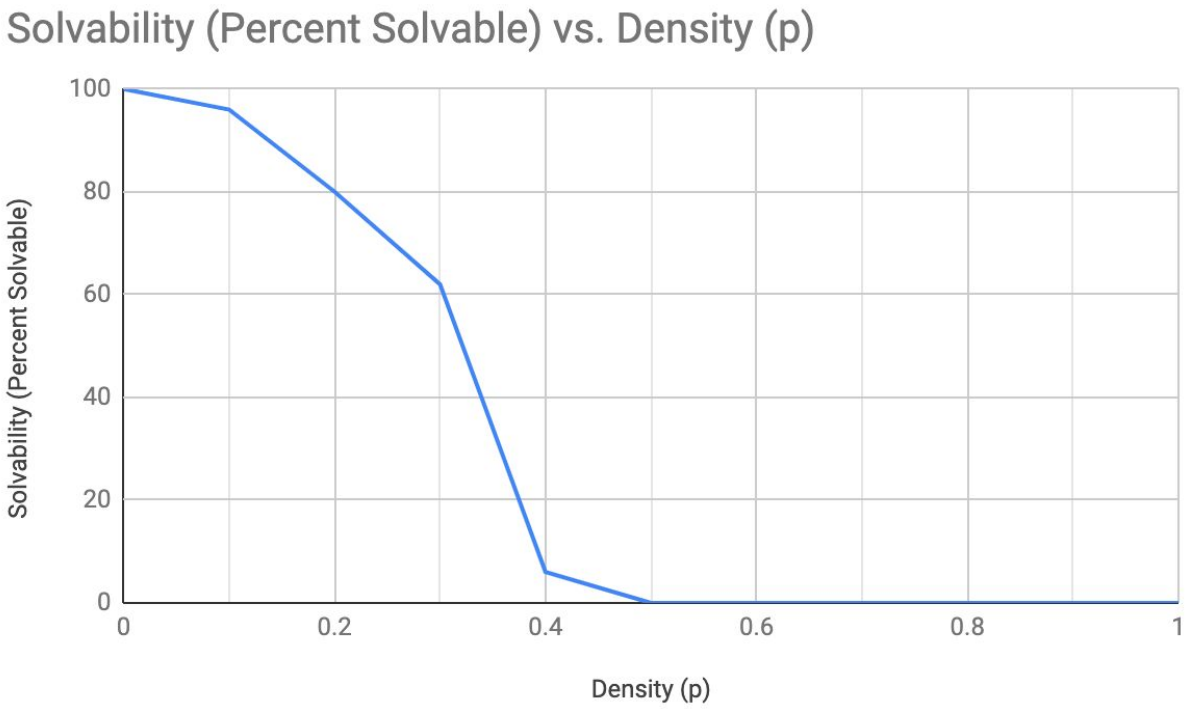


Figure: Plot for dim = 50 over 50 trials

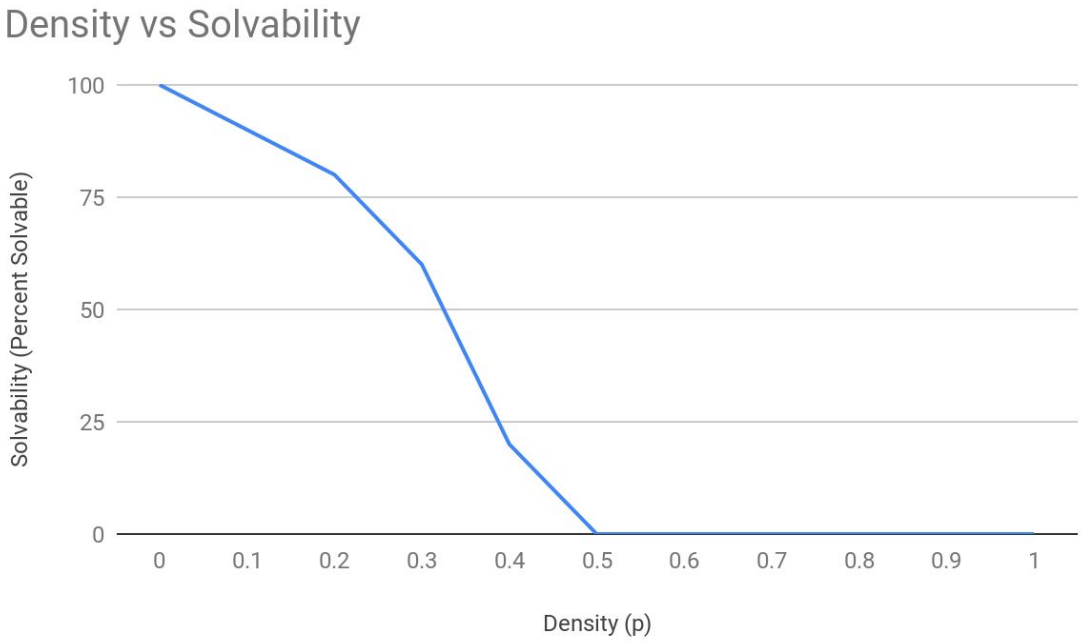


Figure: Plot for dim = 100 over 50 trials

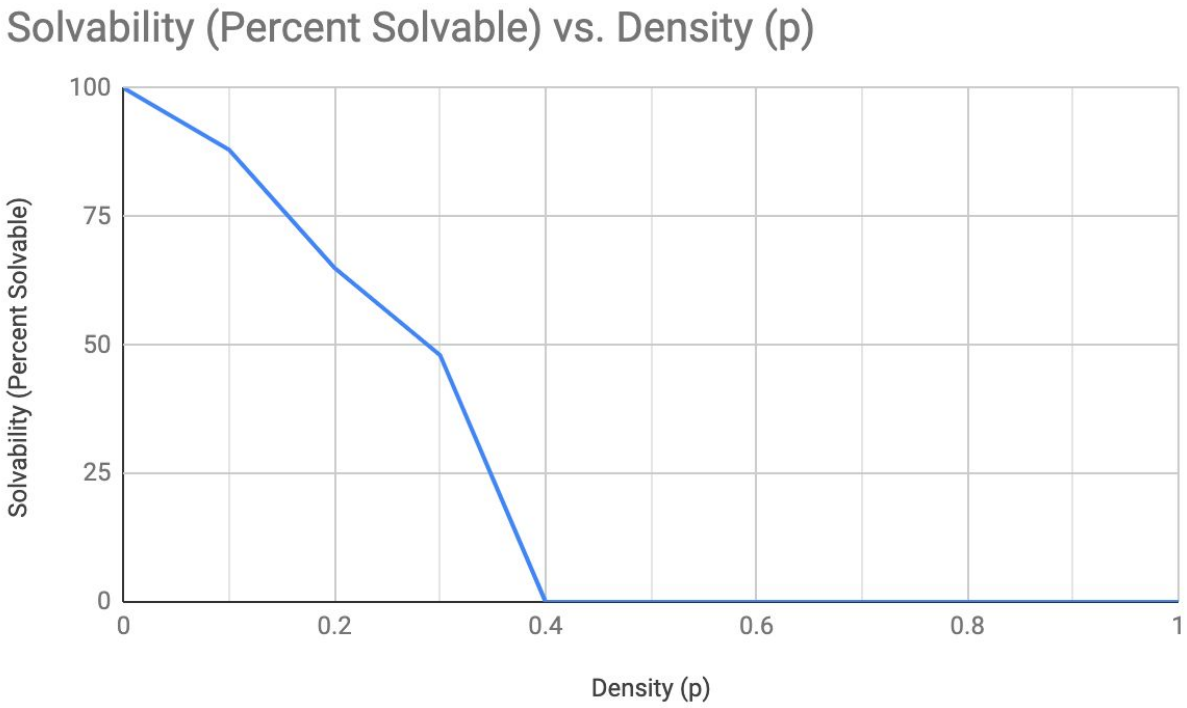


Figure: Plot for dim = 250 over 50 trials

From the Solvability vs Density plots, we can see that the threshold probability p0 very decreases. However, this decrease is very insignificant, as the threshold probability remains at just about 0.3. Thus, we can conclude that the increase in dim causes a slight decrease in threshold probability p0.

Part 3: Generating Hard Mazes

What local search algorithm did you pick, and why? How are you representing the maze/environment to be able to utilize this search algorithm? What design choices did you have to make to apply this search algorithm to this problem?

- We decided to construct a genetic algorithm for our local search. We believe that the environment of a maze is unique in that for a certain maze a subset of that maze might be ideal, but another subset might cause a dead path. A genetic algorithm allows us to choose the hardest/ideal mazes from a population and try to build children of both and create even harder and more ideal mazes. We are essentially taking mazes that have the highest score for a certain heuristic, cutting them each in half and building a new child to see if it will have a higher score than its parents. We picked this one because intuitively it made the most sense to us that if we combine halves of two hard mazes in some scenarios it would generate an increasingly hard maze due to factors of both difficult mazes combing and forming into one.
- For our design we constructed our maze environment as 2D arrays which we later portrayed as visuals for better readability. However, the backend of the algorithm uses a 2D array to represent the state of the maze. The maze itself is created by taking the probability input and filling in the elements of the 2D array randomly. The only other parameter for the genetic algorithm we used was a string that indicates which paired metric it wanted to evaluate the algorithm for: DFS with Max Fringe Size or A* Manhattan with Max Nodes Expanded.
- Design Choices
    - Population: We stored our population as an array. Each element of the population is another array, which we will call maze for this explanation, which contains 3 elements. The first element of each maze element in the population will have a path that leads to the start to finish of a generated map and if a path does not exist it will have None in maze[0]. The second element of maze will hold the heuristic score, so for example for dfs it will return the max fringe size at any point while running dfs on the map of the specific maze. The final element of maze will simply be the original map which we ran the search algorithm hard, also known as our original snapshot of the maze.
    - Fitness Function/Parameter: Given our population we need some way to quantify fitness or in our case hardness of a maze. We do this by using a specified heuristic which for this assignment was max fringe size for dfs and max nodes expanded for A*.
    - Selection: Since we have our population constructed we then sort our population by the second element of every population element which will be our

fitness/heuristic. This makes sure the hardest mazes based on our heuristic will be the first ones in our population. Then we can easily select the first and second element to be parents.

- Recombination: Now that we have two parents that are the hardest mazes in our population we will perform a unique crossover. We will take the first half of the dad maze and second half of the mom maze to create a child that inherits attributes from both parents. With hard maze elements form both parents the likelihood of the child being very hard is high.
- Mutation: After we have the child of both our hardest mazes we can now first calculate his fitness and find his proper place in the population. We mutate the newly created child by adding blocks where all neighbors are open and see if we can make the generated child even harder than its parents.
- Then, we update our population accordingly, if the child was an unsolvable maze we discard him or we put the solvable child maze into our population
- Finally, we run our genetic algorithm with a termination (which is described in the next question) then we visualize the first element of the population which will be the hardest maze in our population

Unlike the problem of solving the maze, for which the 'goal' is well-defined, it is difficult to know if you have constructed the 'hardest' maze. What kind of termination conditions can you apply here to generate hard if not the hardest maze? What kind of shortcomings or advantages do you anticipate from your approach?

- A simple way of terminating or getting a very good hard maze is to terminate after a certain amount of time/iterations. Since we are working in a very randomized environment of mazes, a lot of time we will be subjected to mazes that have no path. But, we decided to incorporate a system where we make sure we have 100 hard and valid mazes in our population at all times. Next, we have another iteration termination where we will make sure we have generated 50 new valid child mazes. This means that our algorithm will have a long threshold to mate and create a new hard maze. Our approach is good because we will be pairing the most fit members of our population and generating a new fitness each time, but over time we will hit a stagnant fitness value. Which is why we terminate after a certain time rather than a specified alternate condition. If we have dealt with hard mazes and their children for a reasonably long amount of iterations we can be confident that we have a very hard maze. A shortcoming of this could be that we end up with children that are not properly mutated. We could have a child that would be the hardest maze, but one blocked node could be preventing this. Due to inheritance our genetic algorithm might never improve in a drastic way and we might stay stagnant on a certain fitness score.

Try to find the hardest mazes for the following algorithms using the paired metric:
– DFS with Maximal Fringe Size
– A* -Manhattan with Maximal Nodes Expanded

Below are visualizations of hard mazes for each metric,

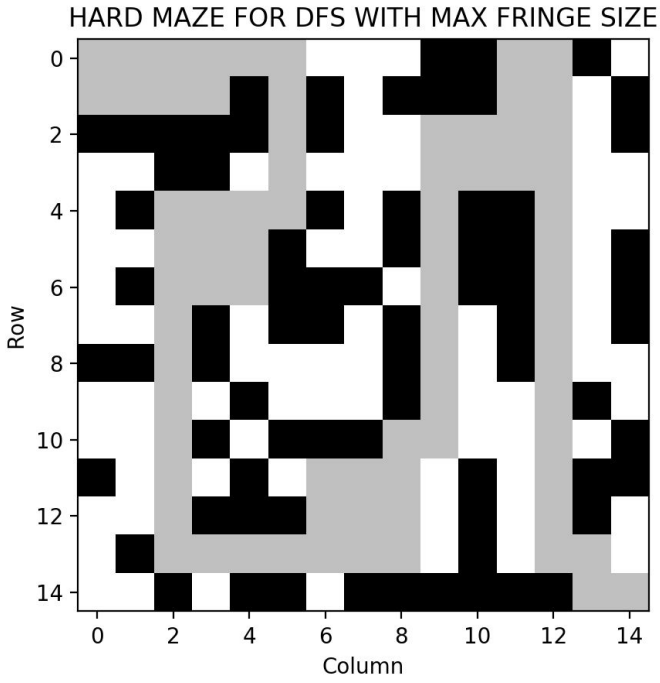


Figure: Map of dim = 15, p = 0.4

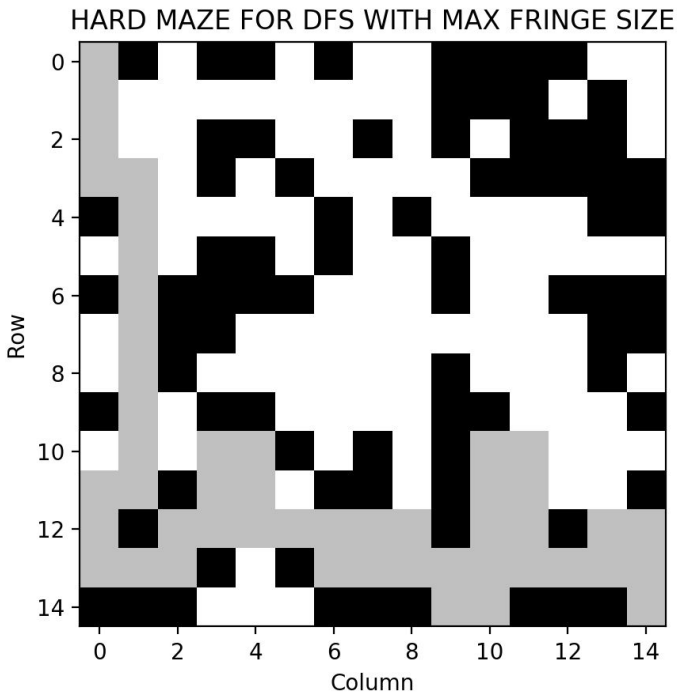


Figure: Map of dim = 15, p = 0.4

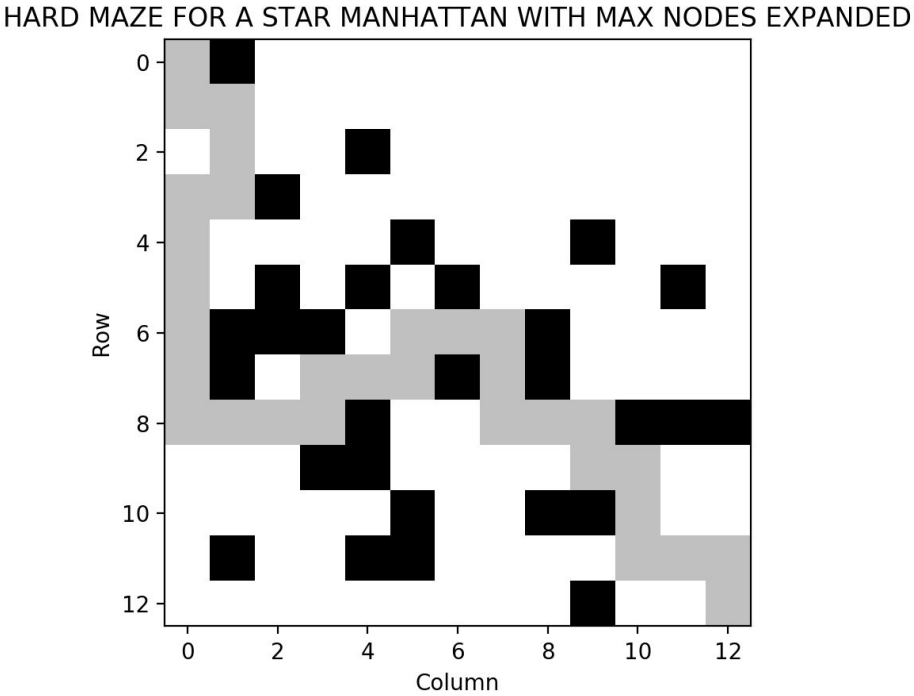


Figure: Map of dim = 13, p = 0.2

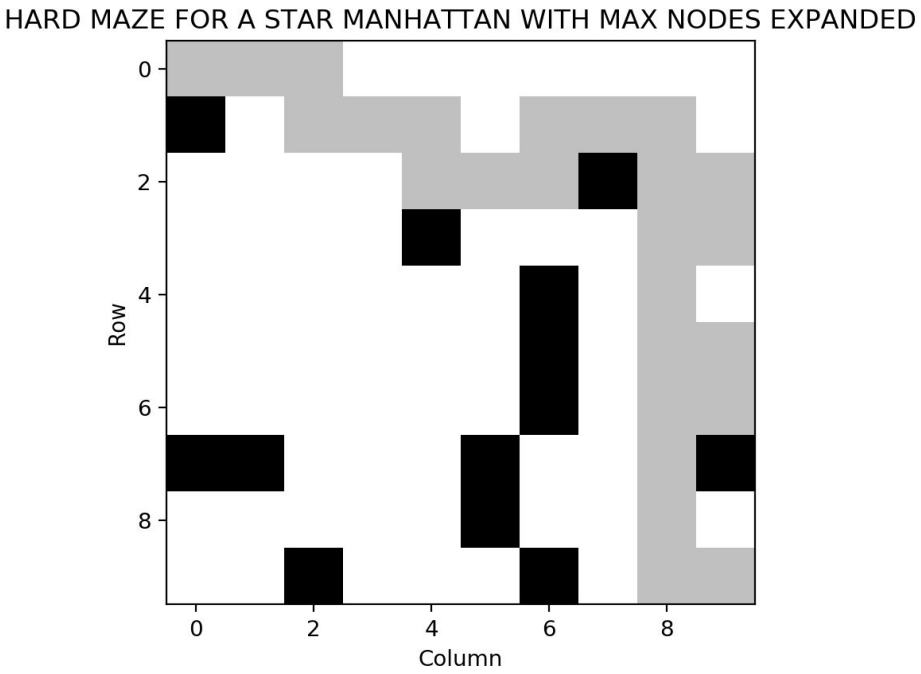


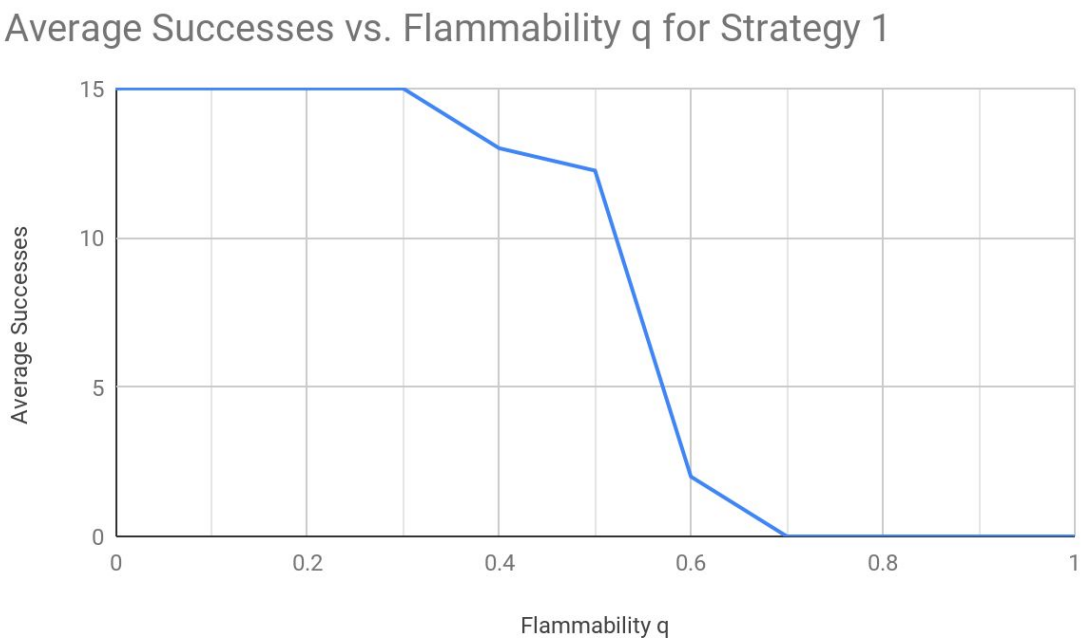
Figure: Map of dim = 10, p = 0.2

Part 4: What If the Maze Were On Fire?

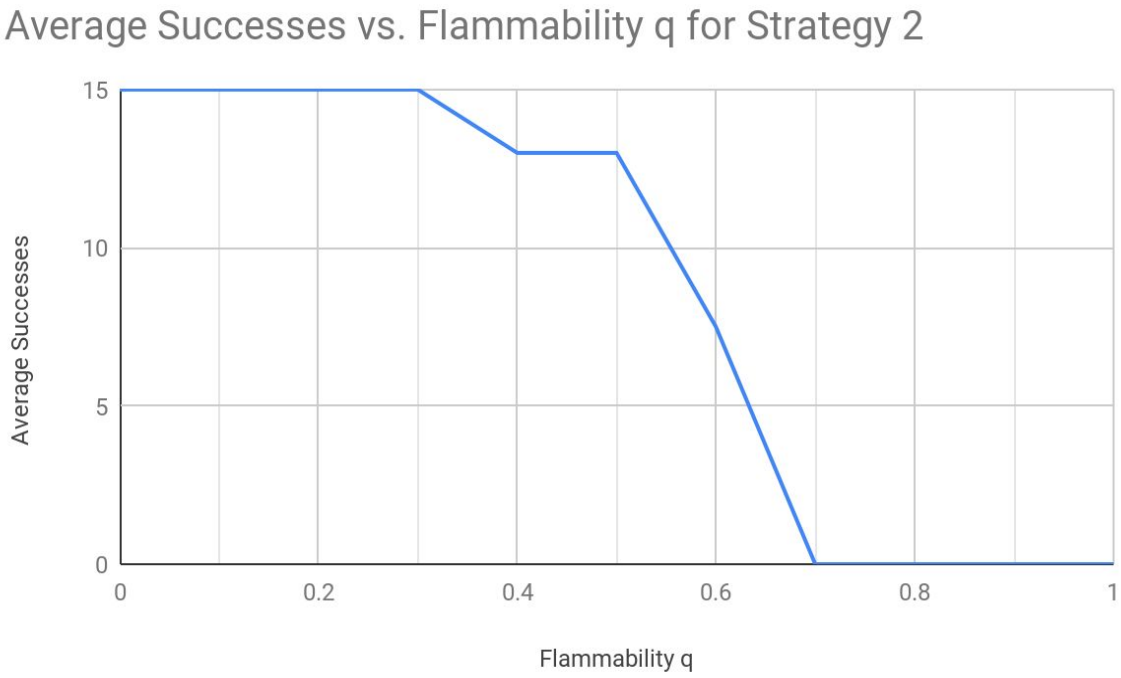
Generate a number of mazes at the dimension dim and density p0 as in Section 2. Be sure to generate a new maze and a new starting location for the fire each time. Please discard any maze where there is no path from the initial position of the agent to the initial position of the fire - for these mazes, the fire will never catch the agent and the agent is not in any danger. For each strategy, plot a graph of 'average successes vs flammability q'. Note, for each test value of q, you will need to generate multiple mazes to collect data. Does re-computing your path like this have any benefit, ultimately?

The plot of Average Successes of 15 trials vs Flammability q for each strategy with dim = 100 and density $p_0 = 0.3$ along with flammability q ranging from 0 to 1,

Strategy 1: At the start of the maze, wherever the fire is, solve for the shortest path from upper left to lower right, and follow it until you exit the maze or you burn. This strategy does not modify its initial path as the fire changes.



Strategy 2: At every time step, re-compute the shortest path from your current position to the goal position, based on the current state of the maze and the fire. Follow this new path one time step, then re-compute. This strategy constantly re-adjusts its plan based on the evolution of the fire. If the agent gets trapped with no path to the goal, it dies.



Does re-computing your path like this have any benefit, ultimately?

Yes, recomputing our path at every step improves the average success, this can especially be seen between flammability q ranging from 0.5 to 0.7. Due to the chance that the previous path was closed off by a fire cell, recomputing the path at every step allows the agent to account for this roadblock and find another path to the goal. Thus, we can see that accounting for the changes in the fire cells will be of benefit, especially for larger mazes with high flammability.

Come up with your own strategy to solve this problem, and try to beat both the above strategies. How can you formulate the problem in an approachable way? How can you apply the algorithms discussed? Note, Strategy 1 does not account for the changing state of the fire, but Strategy 2 does. But Strategy 2 does not account for how the fire is going to look in the future. How could you include that?

We can observe that strategy one does not account for the change in the fire. However, strategy two does account for the change in the fire, while it does not account for future spread of the fire. Thus, we must find a way to account for the fire as well as its ability to spread in the future using estimates.

A strategy we have developed in order to beat both of the previous strategies is to perform a modified BFS on the dynamic map. For every neighbor of each cell, we compute the Manhattan distance as the heuristic from the cell to the origin of the fire. After computing the heuristics for every neighboring cell, we choose the cell with the highest calculated heuristic, as that cell is estimated to be farthest from the spread of the fire.

Below are visualizations of the map at the start (left hand side) and at the end (right hand side) of algorithm execution,

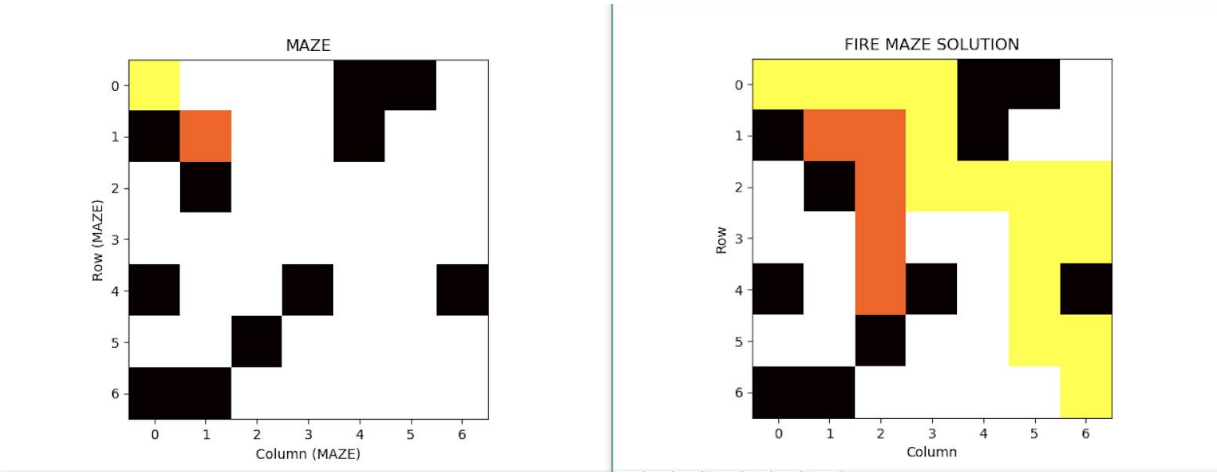


Figure: Map of dim = 7, p = 0.3, q = 0.1

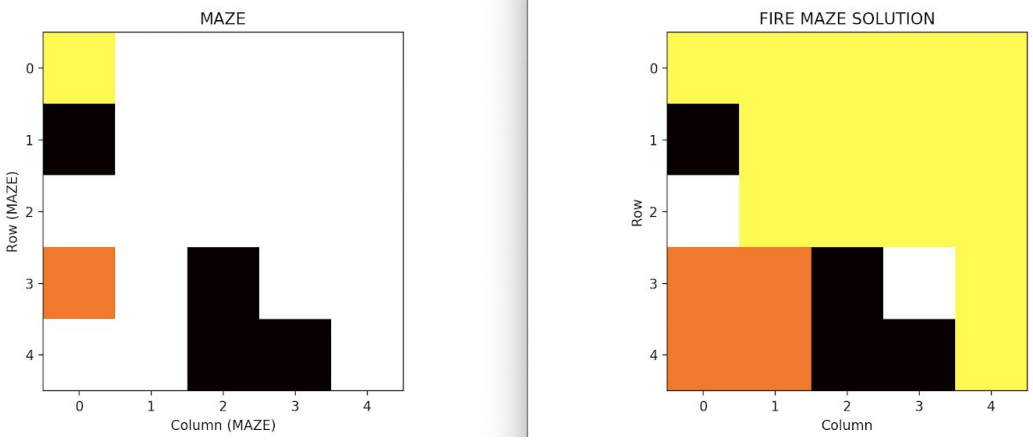


Figure: Map of dim = 5, p = 0.3, q = 0.2

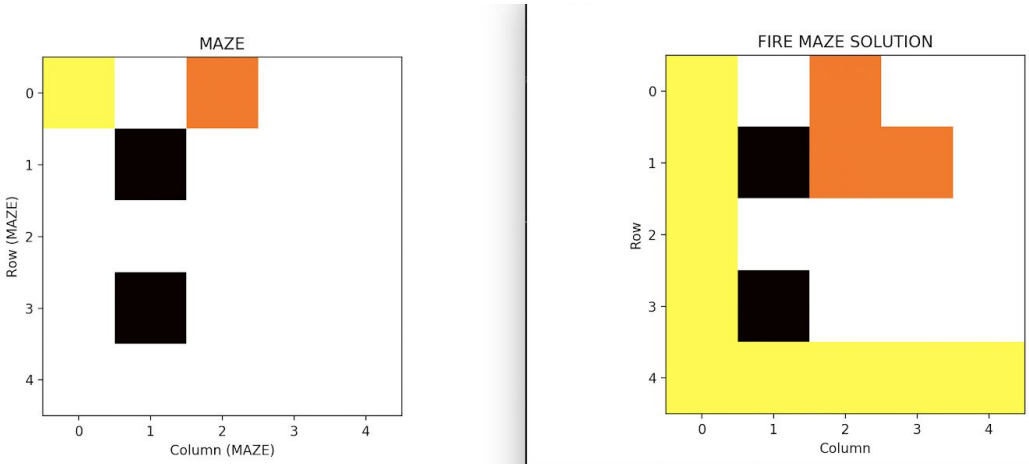
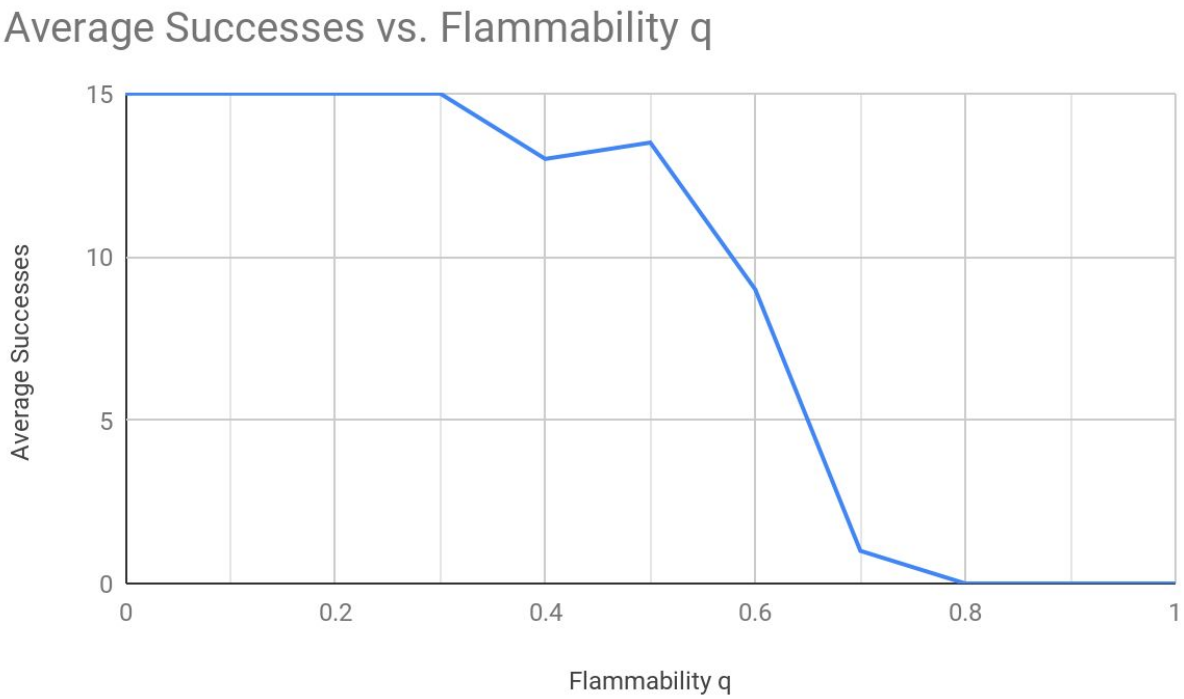


Figure: Map of dim = 5, p = 0.3, q = 0.2



We can observe that our strategy was more efficient as we only computed the Manhattan distance to the spread of the fire. This proves to be computationally more efficient then recomputing the path from the current cell to the goal cell in strategy two. Furthermore, the heuristic allowed the agent to stay farthest away from the spread of the fire as possible. Also, the algorithm's average success rate stays fairly high as our path is fundamentally dependent on the origin of the fire.

Contributions:

Zain Siddiqui - A* Euclidean, A* Manhattan, Part 2: Analysis and Comparison, Part 4: Fire Maze
Krupal Patel - Visuals, DFS, BFS, BI-BFS, Part 3: Hard Mazes, Part 4: Fire Maze
Sunehar Sandhu -