CS 520: Introduction to AI

Wes Cowan

Assignment 1 - Search

(September 24, 2019)

197001190: Aditya Lakra (al1247)
196003272: Hari Priya (hp467)
197001859: Twisha Naik (tn268)

TABLE OF CONTENTS

1. Environments and Algorithms **2**
Grid Implementation: 2
Path Planning: 2
Implementation Logic: 2

2. Analysis and Comparison **3**
Fixing a dim 3
Visualization and comparison of paths 7
Maze Solvability dependence on p 10
Average Shortest path length v/s p 11
Comparison of heuristics 12
Algorithm Behavioral Analysis 13
Improved DFS 14
A* v/s BD-BFS 15
Bonus: Relation of p0 and dim 16

3. Generating Hard Mazes **18**
Choice of Local Search Algorithm 18
Design Choices 18
Termination Conditions 20
Hardest maze using Algorithm with paired metric 22
DFS with Maximal Fringe Size 22
A*-Manhattan with Maximal Nodes Expanded 23
Result Intuition Comparison 25

4 What If The Maze Were On Fire? **30**
Idea 30
Approach 30

Division of Work **36**

1. Environments and Algorithms

Grid Implementation:
2D matrix of an object called cells.

```
"""
value = 0 => empty cell
value = 1 => occupied cell
value = 2 => burning cell
"""
cell.value = value
cell.parent = None
cell.visited = False
```

The object approach to the problem makes it easier to code it without having to store many lists. A maze is generated by making a cell occupied with a certain probability p.

Path Planning:
Given a maze, our goal is to search for a path from source to destination:

Source = top left corner (0, 0)
Destination = bottom right cell (dim-1, dim-1)

Implementation Logic:
For each of the algorithms, we have a fringe (which keeps track of the unexplored cells) and a closed set that maintains the cells that have already been explored i.e. all the neighbors of that particular cell are added on the fringe.

Each algorithm returns a dictionary which stores the following information which can be used for important analysis and comparisons of these algorithms.

- ➔ Success/Failure (If there is a path from source to goal)
- ➔ Exploration steps
- ➔ Maximum fringe size
- ➔ Average fringe size
- ➔ Closed set (The length of closed set will give the number of exploration steps)

The algorithms implemented with the corresponding code file is as follows:

1. Depth-First Search (dfs.py)
2. Breadth-First Search (BFS.py)
3. A∗ (a_star.py): where the heuristic is to estimate the distance remaining via the Euclidean Distance => d((x1, y1), (x2, y2)) = □√(x1 − x2)^2 + (y1 − y2)^2.
4. A∗ (a_star.py): where the heuristic is to estimate the distance remaining via the Manhattan Distance => d((x1, y1), (x2, y2)) = |x1 − x2| + |y1 − y2| (a_star.py)
5. Bi-Directional Breadth-First Search (BiDirectionalBFS.py)

2. Analysis and Comparison

- **Fixing a dim**

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?

Approach:

Keeping the probability constant, we calculated the total time taken to generate 100 solvable mazes of dimensions in the range of [60, 240] and the time taken to solve these 100 mazes using the BFS algorithm. This gives us an idea about the feasibility of choosing a specific dimension.

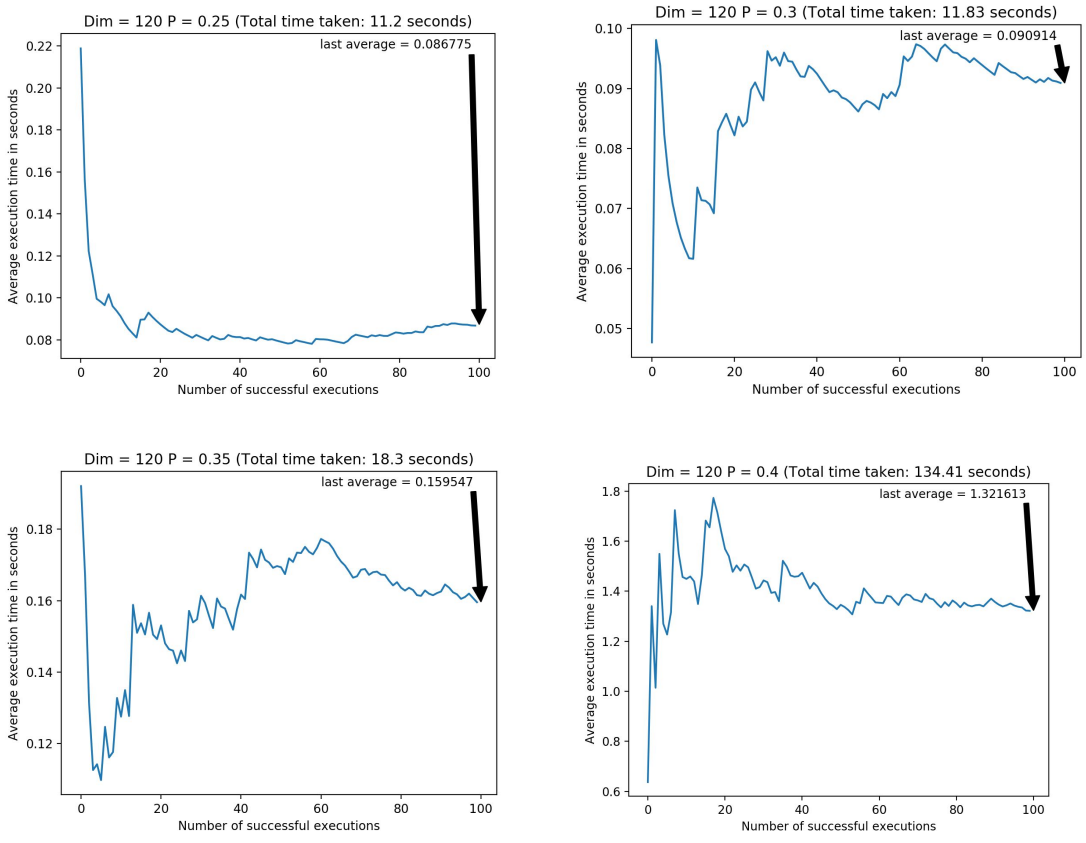


Fig:1 Plots for average execution time for dim=120 and four probabilities

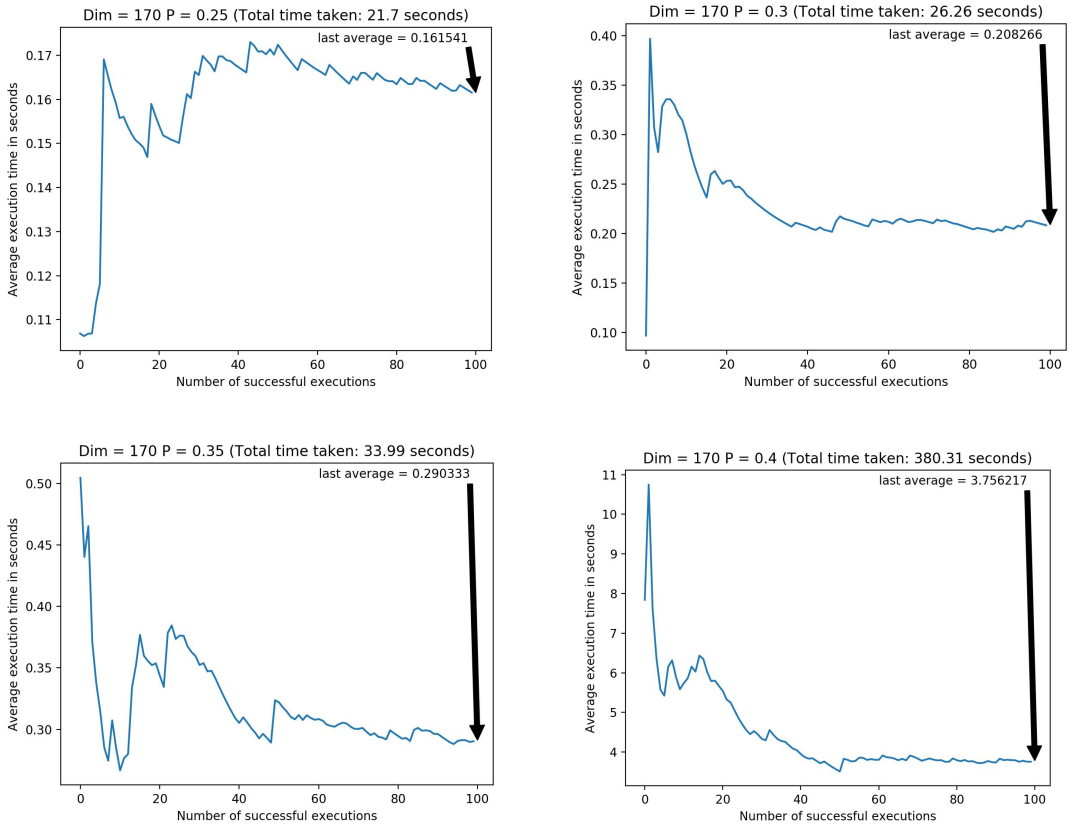
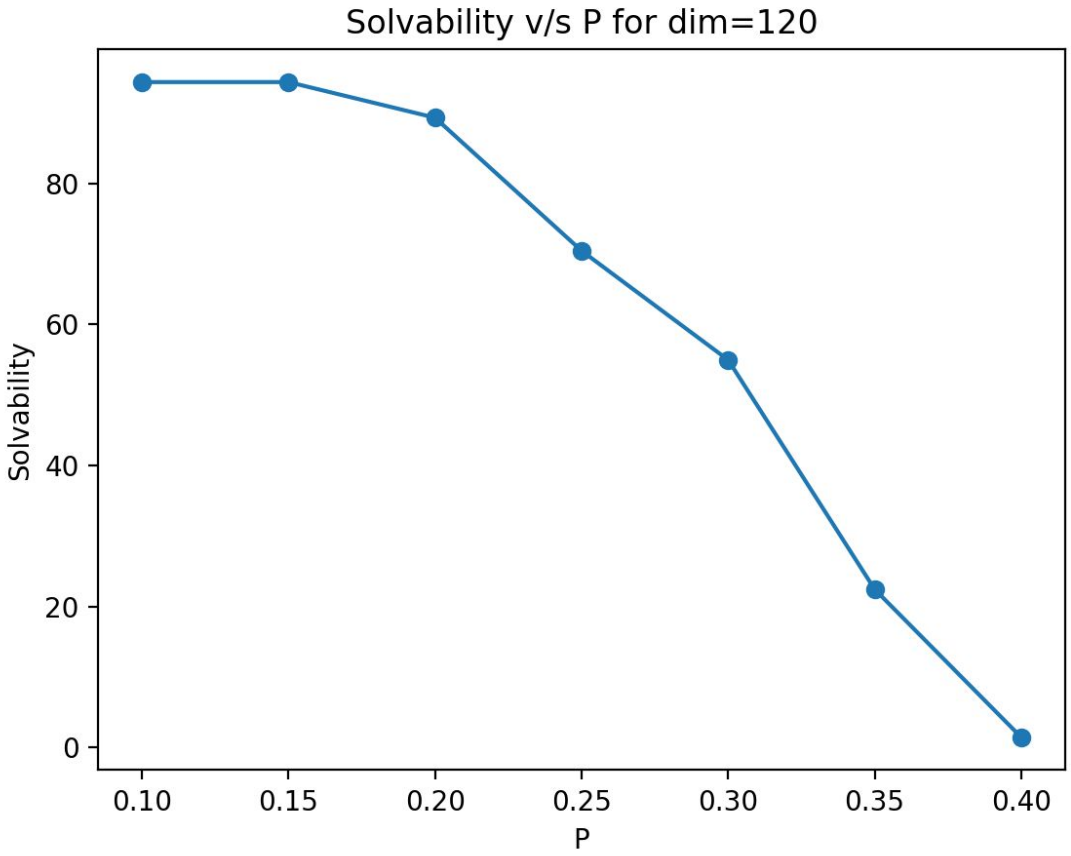


Fig:2 Plots for average execution time for dim=170 and four probabilities

Probability range chosen = [0.1, 0.4]

- p~0.1 most of the mazes were solvable
- p~0.4 most of the mazes were unsolvable (too much work to generate a solvable maze)



NOTE: Given a dimension, the effort to solve a maze is not directly proportional to p. But the solvability is directly proportional to p. There might be very few open cells that can be explored while searching a path from source to destination when more cells are blocked i.e. when p is high.

p∝ solvability of maze

p not proportional to effort (exploration to solve the maze)

From the graphs below, it can be observed that there are significant spikes in the execution time starting from dim = 120. This indicates a significant and reasonable amount of work required to solve the mazes.

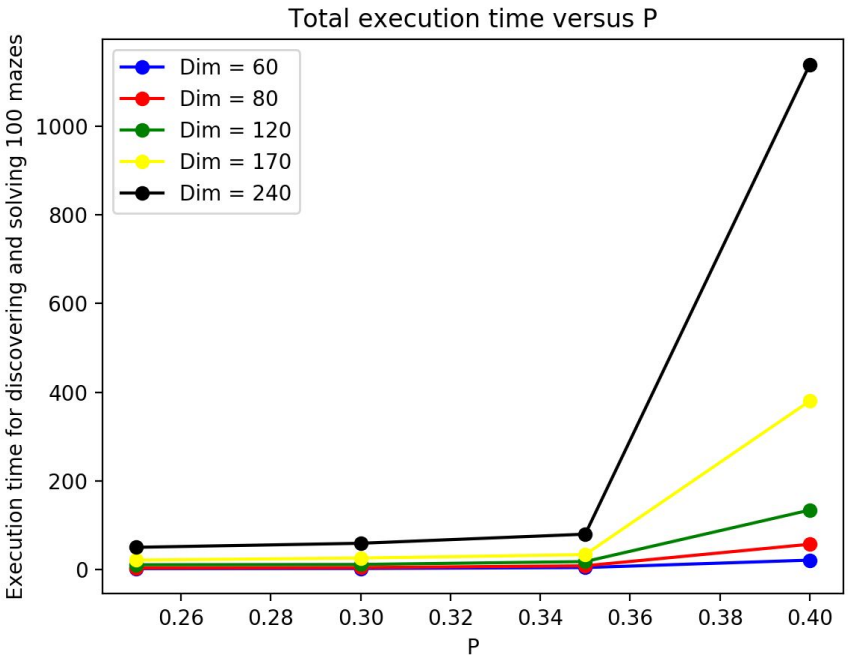


Fig: 3a Total execution time to generate and solve 100 solvable mazes using BFS for different dimensions and vale of p.

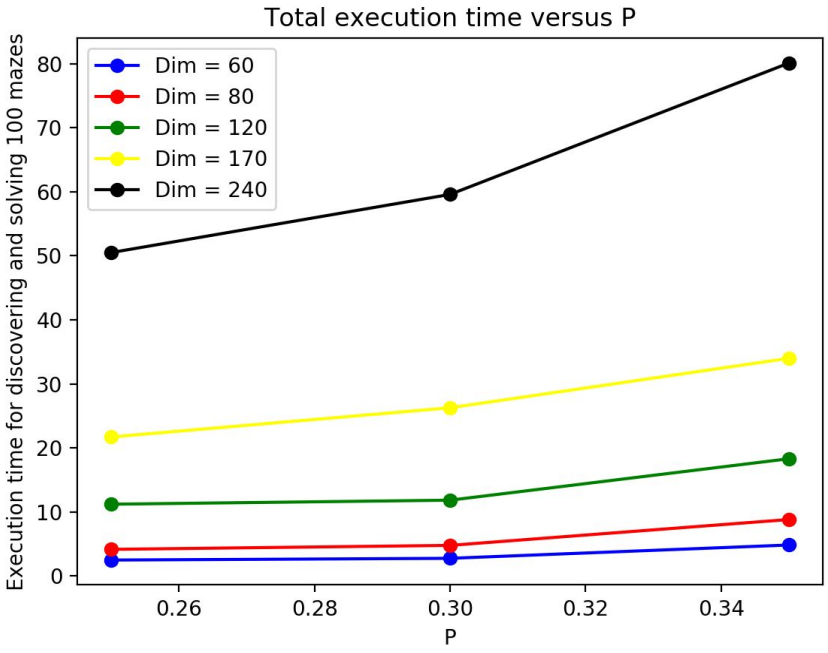


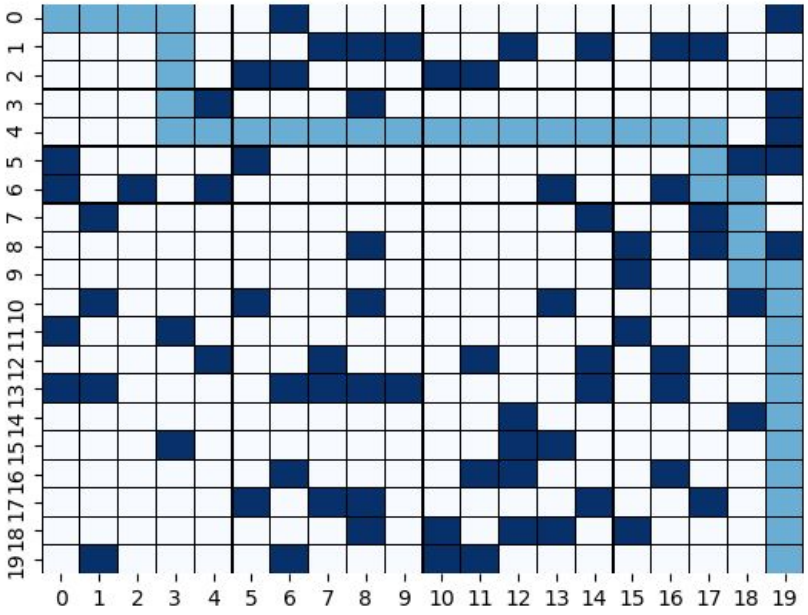
Fig: 3b Removed p=0.4 because we will mostly be working with p<=0.35 to get a better comparison.

- **Visualization and comparison of paths**

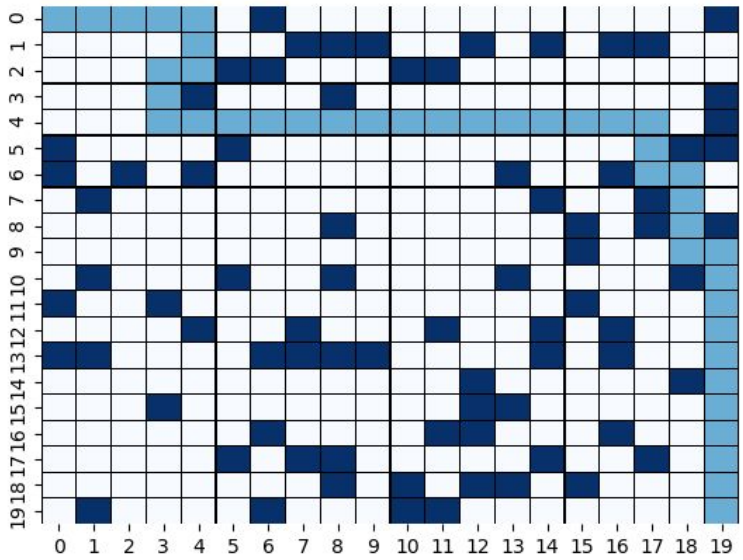
For p ≈ 0.2, generate a solvable map, and show the paths returned for each algorithm. Do the results make sense?

dim = 20 -> For clear visualization

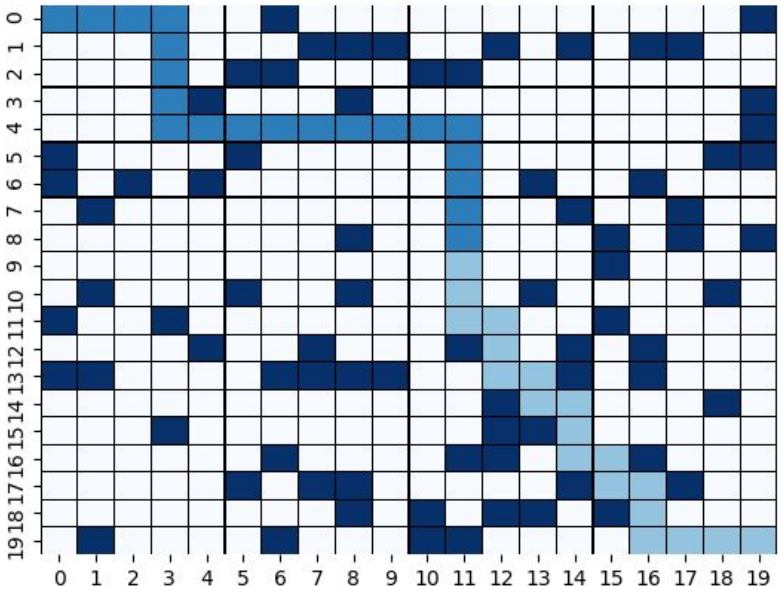
1. **BFS** - The path is the shortest path possible and it tries going right and down first because of the way it is coded. Neighbors are added in the **(right - bottom - left - bottom)** order in the queue. Hence, it is popped in the same order.



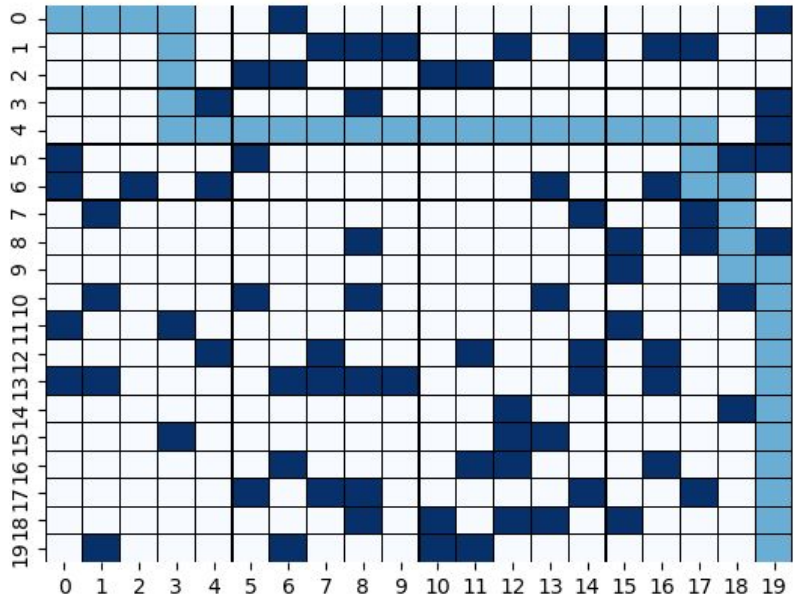
2. **DFS** - The path is as expected. It is not the shortest path but the path that always explored the right and bottom neighbors first.



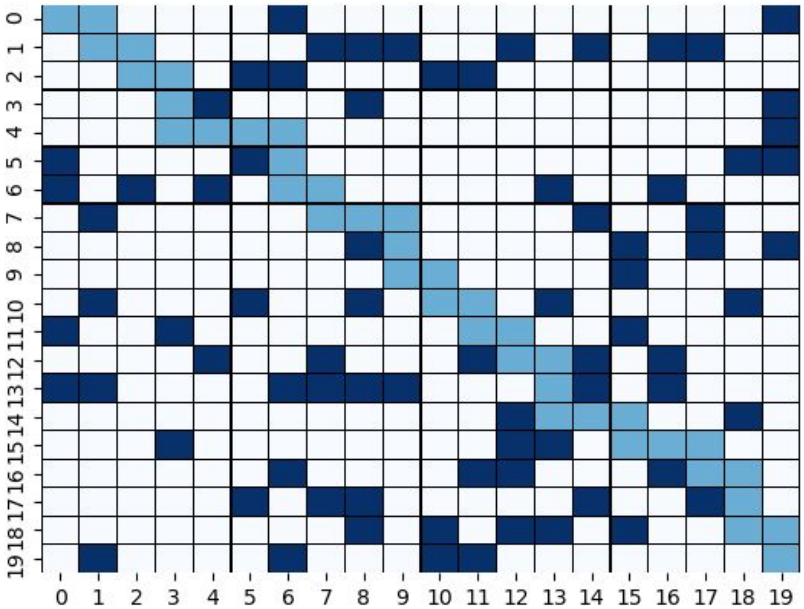
3. **BD-BFS** - The path computed by the two independent BFS one from source and other from destination merges to return a final path.



4. **A* (Manhattan)** - For manhattan the bottom and right direction will be prioritized over the left and top. But, the **bottom cell and right cell will have the same priority**. Because the right element is added to the queue before the bottom element, the path moves right and then down towards the goal.



5. **A* (Euclidean)** - The path will try to follow the diagonal as that will be the least euclidean distance from the destination.



- **Maze Solvability dependence on p**

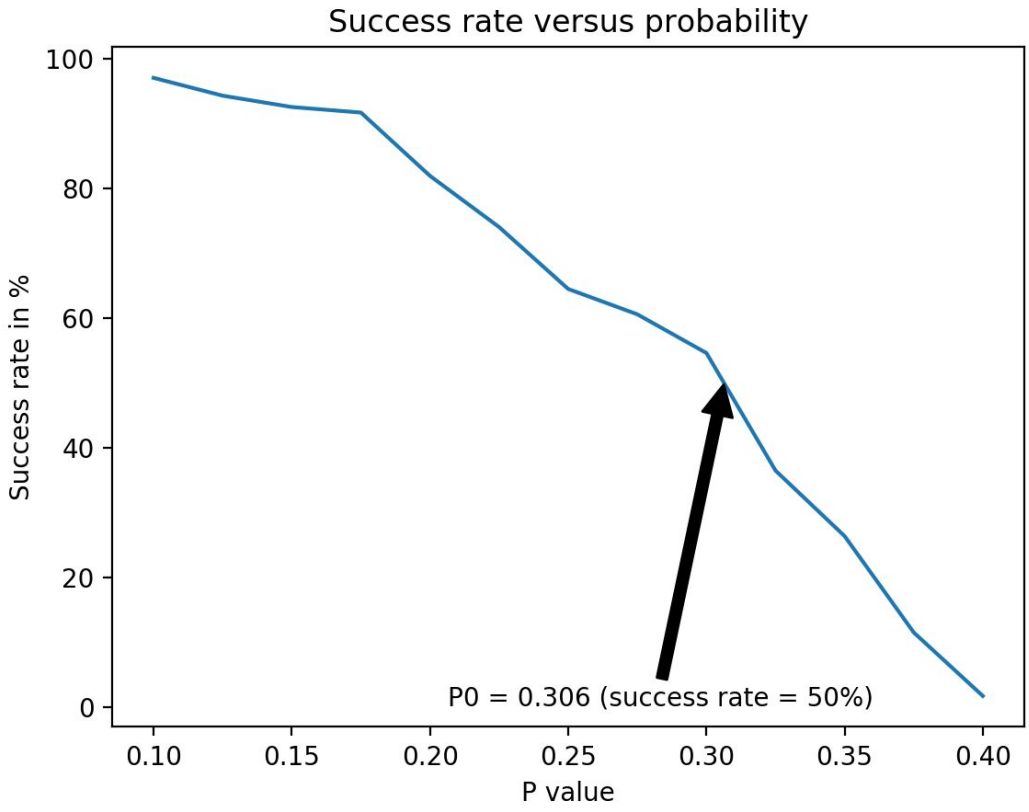
Given dim, how does maze-solvability depend on p? For a range of p values, estimate the probability that a maze will be solvable by generating multiple mazes and checking them for solvability. Plot density vs solvability, and try to identify as accurately as you can the threshold p0 where for p < p0,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

What is the best algorithm to use here? - Here, we are not concerned about the shortest path, we just want to quickly check if a maze is solvable. Time is directly proportional to the number of exploration steps. As A* with Manhattan has the least exploration, it would be the best choice (Assuming that the maze is not large enough to create memory issues)

Keeping the **dimension constant at 120,** we found out the maze solvability percentage for a different range of probabilities.

As the graph has a negative slope, we can observe:

- p < 0.306: success rate > 50%
- p > 0.306: success rate < 50% => **p0=0.306**



- **Average Shortest path length v/s p**

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?

Choice of algorithm:

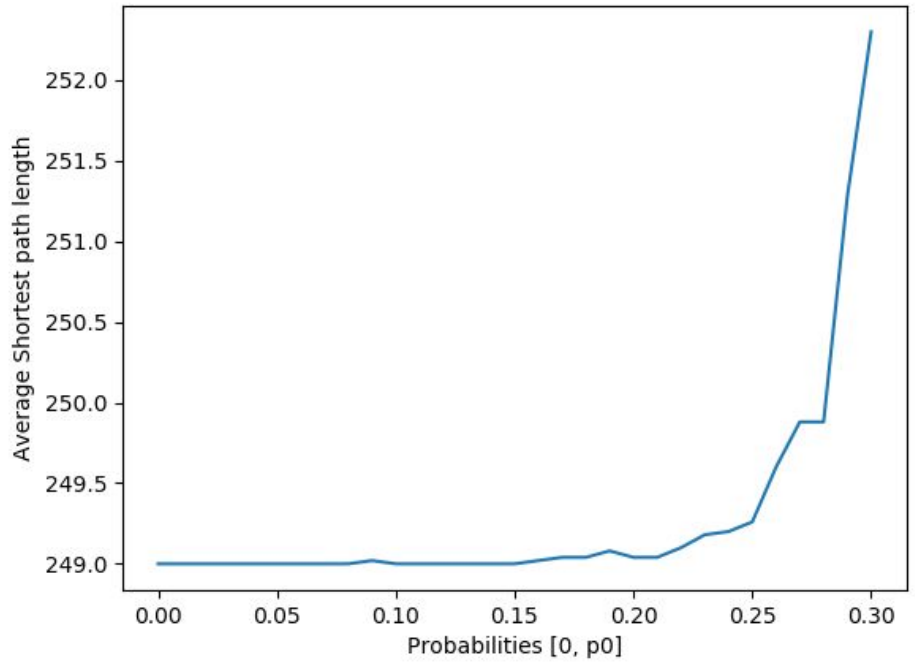
The algorithms BFS, BD-BFS and A* return the shortest path from source to goal. However, BD-BFS is better than BFS due to lesser time complexity and A* is better than BD-BFS due to lesser number of node exploration steps. Hence, we have chosen A* with Manhattan heuristic for calculation of the average shortest path. Reason for choosing Manhattan over Euclidean is presented in the next sub-part.

Implementation details:

To calculate the average shortest path, we generated 100 solvable mazes for each probability in the range [0, p0] and took the average of these shortest path lengths.

Observation:

The average shortest path length keeps increasing with increasing values of probability.



- **Comparison of heuristics**

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

Exploration and Runtime: Manhattan heuristic explores lesser number of nodes as compared to Euclidean. And as a result of this, Manhattan runs faster than Euclidean.

The max fringe size of Manhattan heuristic based A* is more than that of Euclidean based A*. Hence, in terms of memory, Euclidean is slightly better.

For dimension 40 and probability 0.2, below is how each heuristic explores the nodes to reach the goal.

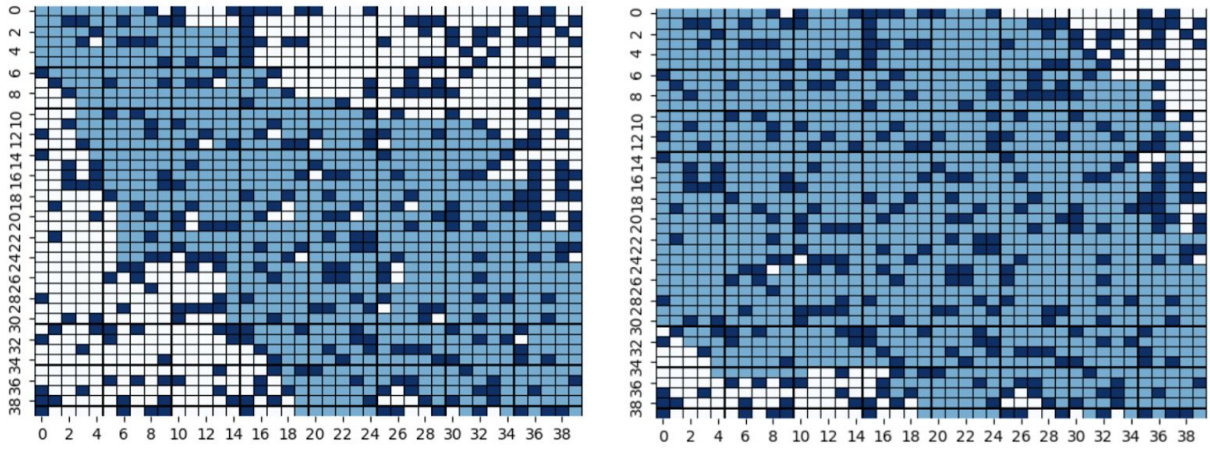


Fig: Left - Node exploration by Manhattan, Right - Node exploration by Euclidean

The reason for this is intuitive because **Manhattan distance contains less relaxation** as compared to Euclidean distance and is therefore **closer to the actual distance** from the goal node.

Space: The average maximum fringe size of A* with Manhattan is slightly more than than of Euclidean heuristic.

- **Algorithm Behavioral Analysis**

The statistics for different algorithms run for 100 mazes of dimension **125x125.**

	Avg. Path Length	Avg. Exploration Steps	Avg. Max Fringe Size	Avg. of Avg Fringe Size
BFS	249	12469.58	120.72	1.07
DFS	313	640.44	326.90	20.04
BD-BFS	249	11364.93	*199.04	1.76
astar_manhattan	249	7671.57	654.22	2.50
astar_euclidean	249	11850.16	205.24	1.53

*For calculating max fringe of BD-BFS, we have added the fringes of both the BFS

Behavioural Confirmations:

❏ BFS, A* and BD-BFS return the shortest path, while DFS does not return the shortest path.

❏ DFS explores lesser nodes as compared to the other algorithms as it tends to travel towards the goal without considering alternate routes on its way.

❏ BFS explores maximum number of nodes as expected. BD-BFS and a_star are improvements over the normal BFS in terms of exploration.

❏ Max fringe size depends on the ratio of elements added in the fringe and elements removed from the fringe. The results for this parameter also confirms the intuition.

- **Improved DFS**

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, as DFS picks one neighbor and thoroughly explores it, it becomes important to make sure it picks the best possible neighbor first. If it picks the one which is closer to the goal, it will reach the goal faster.

The right and bottom neighbors are worth looking at before as compared to the bottom and top ones. We want to explore the nodes in the following order so that we move in the direction of the goal and reach the goal faster. (This is based on Manhattan distance from the goal => (Right and Bottom) have the same priority. So does (Top and Left)

➔ Right
➔ Bottom
➔ Left
➔ Top

Since DFS uses the stack data structure which pops the most recently added nodes first, we added the neighbours into the fringe in the opposite order as below.

➔ Top
➔ Left
➔ Bottom
➔ Right

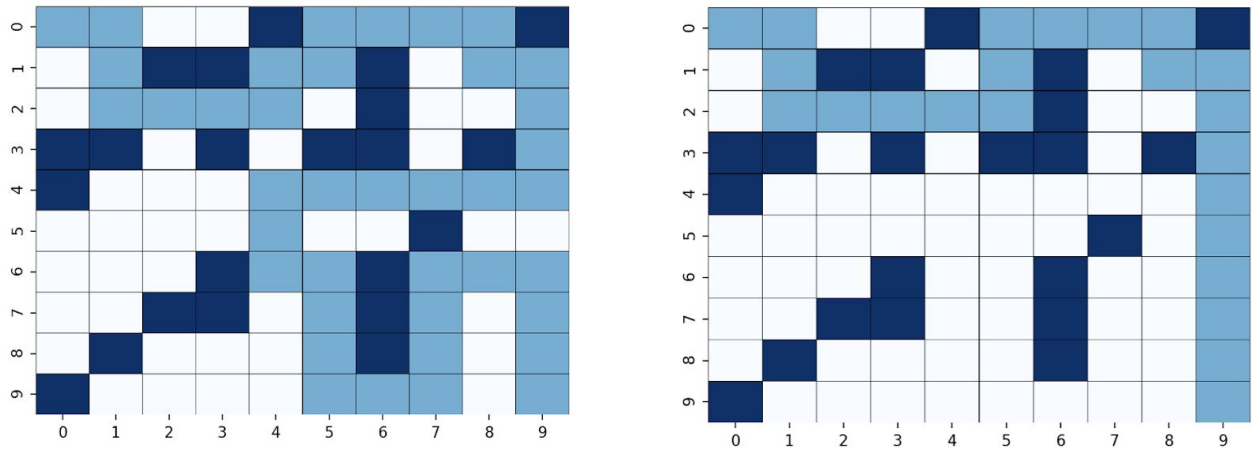


Fig: Left - DFS with random neighbor order, Right - Improved DFS version

- **A* v/s BD-BFS**

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.

Yes.

This is because A* only explores the nodes only based on their heuristic values while BD-BFS explores all the nodes and their neighbours. Also, as it is bi-directional, it will explore nodes from both the ends, source and destination. Most of the nodes explored by the BFS running from the destination would not be explored by A*.

The following map of dimension 10 is an example for this.

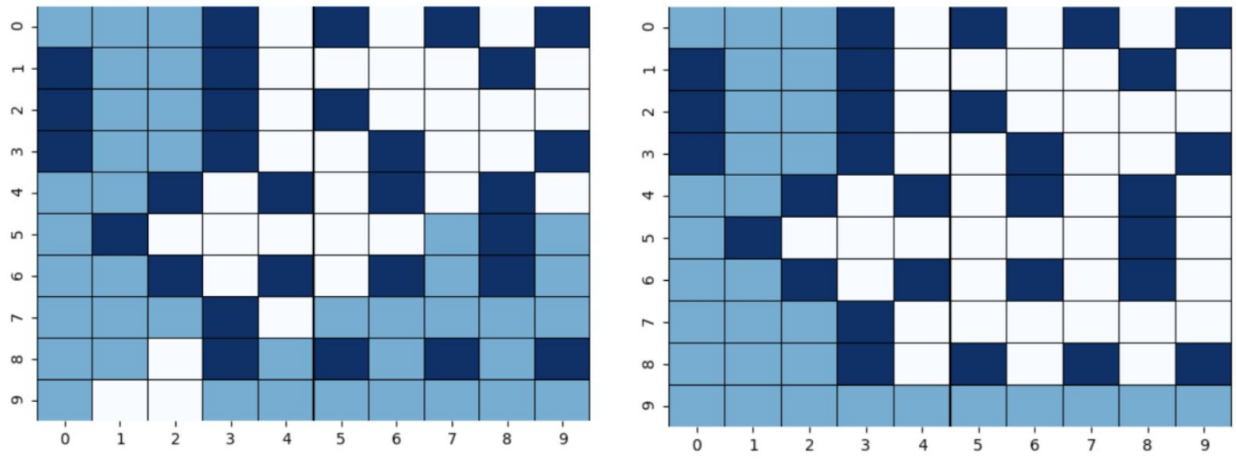


Fig: Left - BD-DFS Node Expansion, Right - A* (Manhattan Heuristic) Node Expansion

Bonus: Relation of p0 and dim

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

p0 depends on the solvability of the graph. The solvability of the graph depends on the probability of the cell being blocked.

Suppose, the p is very high. If the number of blockages are too high, regardless of the dimension of the maze, it would not be solvable for most of the cases.

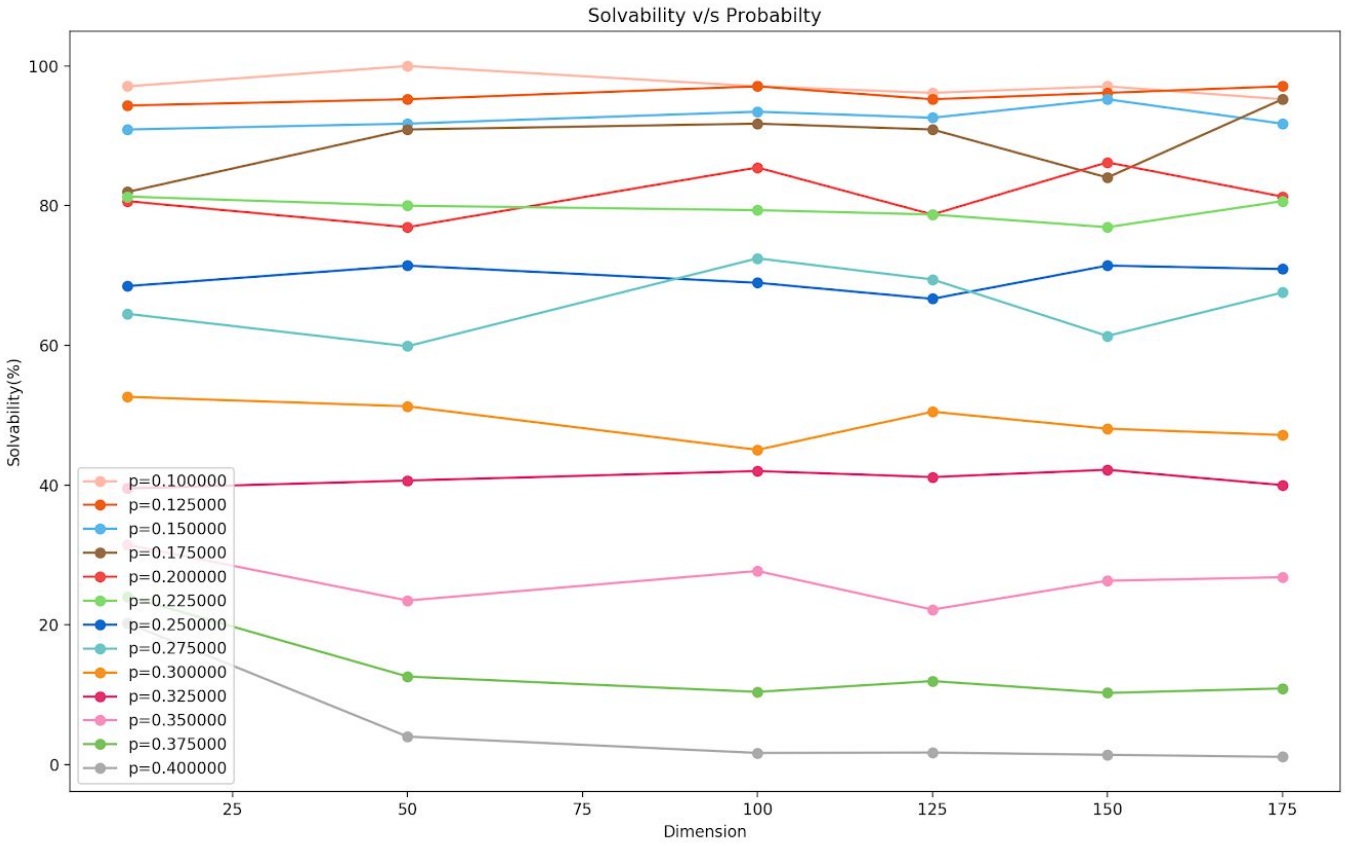


Fig: Dimension vs Solvability

We had p0 = 0.306 for dimension 120. (Plotted in one of the previous sub-part)
From the graph above, for p0=0.3 (orange line), we can see that the solvability is around 50% for all dimensions.

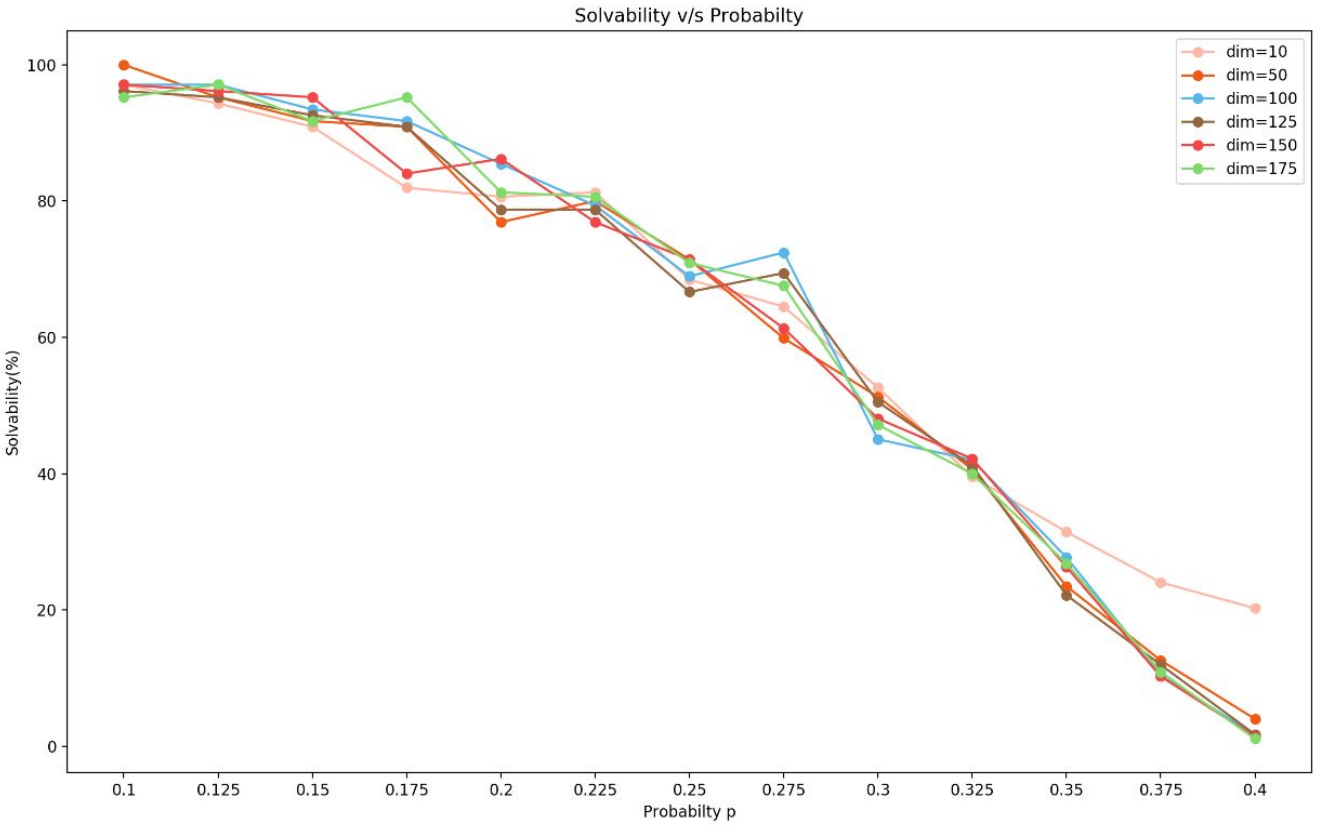


Fig: Probability vs Solvability

From the above plot, we can see that for p = 0.3, the maze solvability is ~50% for all dimensions.

From these two observations, we can conclude that p0 does not depend much on the dim.

3. Generating Hard Mazes

Three possible ways to quantify hard are:
a) how long the shortest path is
b) the total number of nodes expanded during solving
c) the maximum size of the fringe at any point in solving.

- **Choice of Local Search Algorithm**

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 make to apply this search algorithm to this problem?

We picked the **Genetic algorithm** for generating the hardest maze. Genetic algorithm works on the principle of evolution "Survival of the fittest". Where in each generation the fittest of the population survives, reproduces and passes on its characteristics to the next generation.

Since taking portions of the mazes from the parents to create a new maze contains some properties of the parents, the algorithm can help us carry forward the favourable properties and find new ones by obtaining new combinations help us move towards our goal.

Design Choices

1. **Fitness Function** - Hardness of the maze based on the evaluation metric chosen.
For genetic algorithm with A*, the hardness factor is the maximum number of nodes expanded.
For genetic algorithm with DFS, it is the maximum fringe size.

2. **Choosing Parents** - Fitness function based

Approach1: Random choice
Initially, we tried choosing parents randomly out of the whole population without prioritizing based on fitness. That took a longer time to converge.

Approach2: Fitness based
We choose the parents based upon their fitness functions. Harder maze has higher probability to be chosen as the parent.

Implementation steps are as follows:
Let the population be: $\{X_1, X_2 \ldots .. X_N\}$, population size = N with fitness fi
Compute a cumulative population fitness vector V with $V_i = ((f_1+f_2+...+f_i)/ (f_1+f_2+...+f_N)$ for all i = 1, 2, 3….N.

Generate a random number p between 0 and 1. If $V_{i-1} < p < V_i$ then we pick the parent X_i. We repeat this process to pick each parent.

3. **Crossover Point** - Random single crossover strategy

We choose the cross-over points from the parents randomly to simulate the real world scenario of evolution.

- Generate a random probability p in range [0,1] and calculate crossover point, c = p*dim.
- Pick (c) number of rows/ columns from the first parent maze
- Pick (dim-c) number of rows/ columns from the second parent maze
- Combine them to generate a child maze.

This process is repeated until N children are generated.

4. Mutation:

Mutation rate - The rate at which each child is mutated.

Strategies:

a. **Bit flip mutation:** Choose mutation_rate*dim*dim number of random cells and flip the value.
b. **Swapping rows:** Swap mutation_rate*dim number of pairs of rows at random
c. **Swapping columns**: Swap mutation_rate*dim number of pairs of columns at random

Elaborated in the next subpart.

5. New Population Formation - 90% fittest + 10% random

From the total population including the N parents and N children, we pick the new population comprising of 90% fittest mazes (mazes with highest hardness value) and the rest of 10%, is chosen randomly from the remaining population, which comprises of both hard and easy mazes. The random 10% mazes are chosen to maintain diversity in the population.

6. Population size and number of generations:

A population that is too small will not have enough diversity to generate better children. The population might converge soon without much improvement. We tried running the algorithm with a population size of 10, 20, 50, 100. For smaller population size the algorithm did not converge and choosing a very high dimension took a long time to run. Hence, we fixed the **population size to be 50**.

Running genetic algorithm for large mazes was computationally expensive. Hence, we performed our experiments for GA on mazes with **dimension 15x15**.

We have made **different choices for crossover and mutation for the two algorithms suggested**. It is elaborated in the next section when we describe the hardest maze for both the cases.

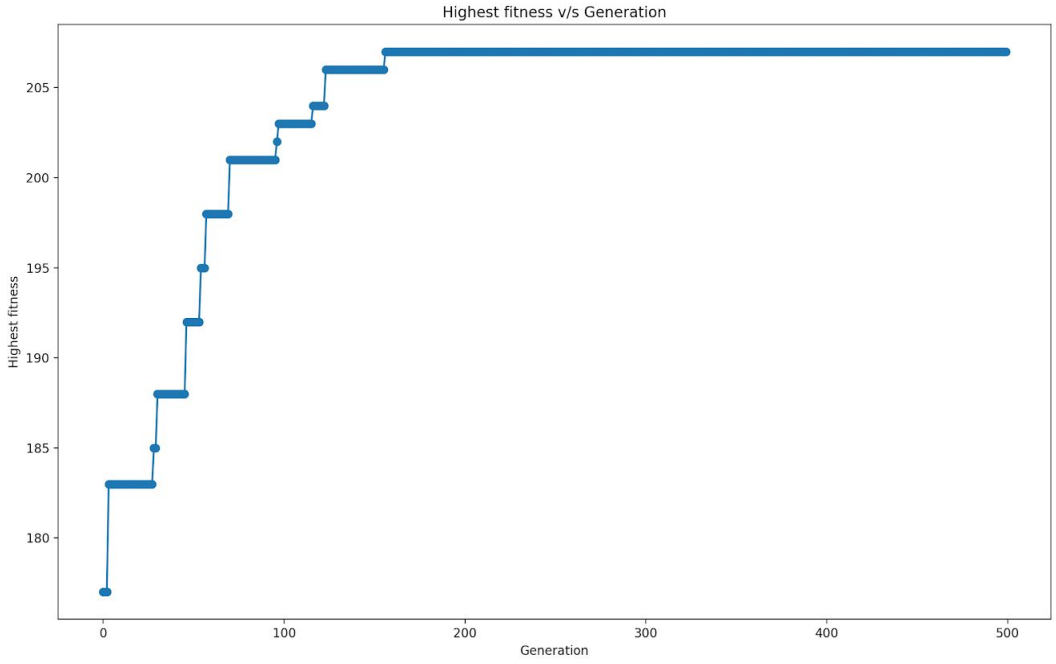
- **Termination Conditions**

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?

Possible Termination Conditions:

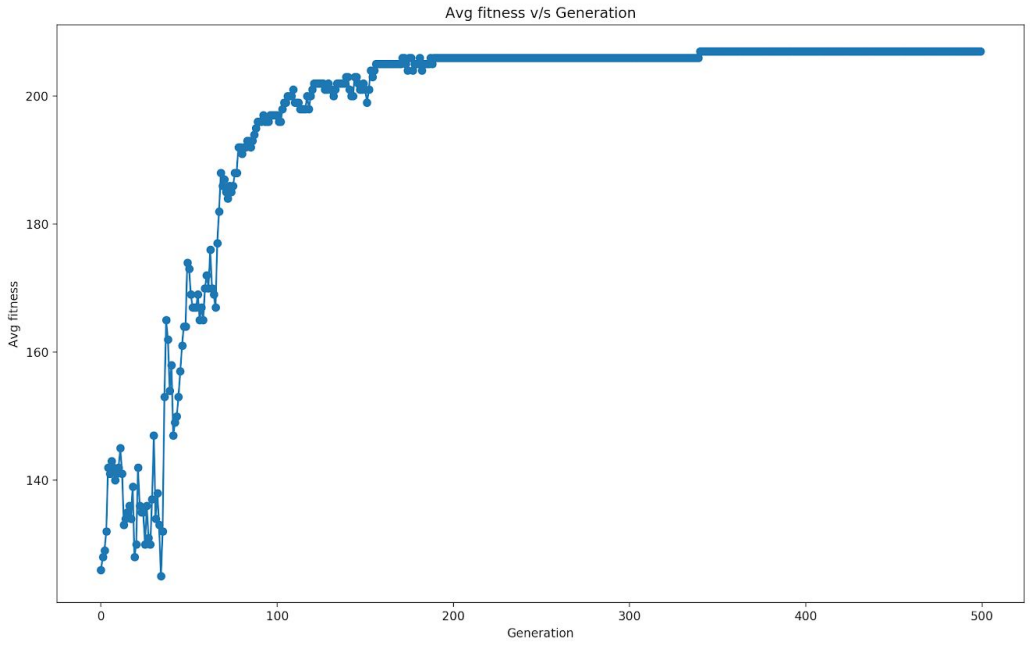
1. The fitness of the fittest maze - If the fitness of the fittest maze remains constant for a certain number of generations, we can conclude that the algorithms has converged.

   Shortcomings: The fitness of the fittest maze can remain constant for a certain number of generations even if the algorithm has not converged. It can be observed in the following plot. If this is used as termination condition, it is possible that we stop at an earlier stage.



2. The average fitness of all the mazes in each population - We can check the improvement in the average fitness of the population

Shortcomings: The average fitness of the population fluctuates a lot. We need to run a few generations of genetic algorithm for it to stabilise before using the average value for judging the population and the fittest solution.



3. Fixing the number of generations - Terminating the algorithm once a few generations have passed. It is difficult to fix the ideal number of generations needed for convergence but for a specific problem we can figure it out by trying out different number of generations.

   Shortcomings: We might end up running the algorithm for a long time unnecessarily or we might end up running it for very less generations.

Potential termination condition: Running the algorithm for a fixed number of generations and after that looking if the average fitness of the population is converging.

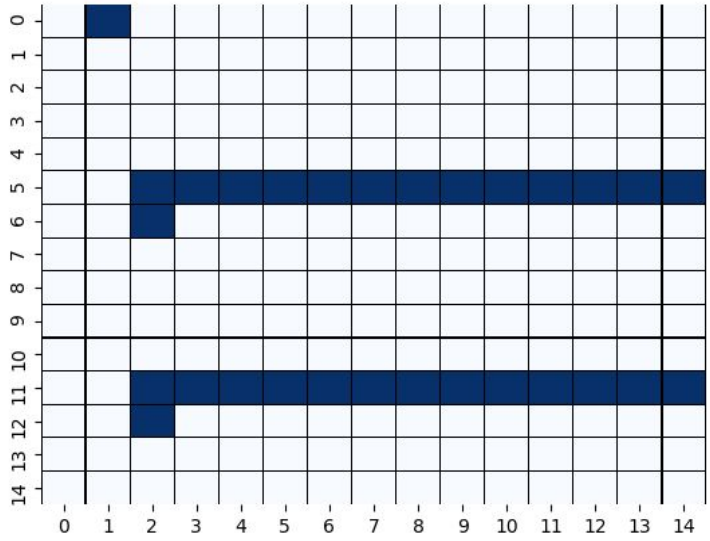
We are using the number of generations as the termination condition for now. We gradually kept increasing the number of generations needed for the average fitness of the population to stabilise. **For 15x15 maze, we are running the algorithm for 500 generations**.

As it can be observed from the graphs above, the average fitness and the fitness of the fittest stops increasing and stabilises after a certain number of generations.

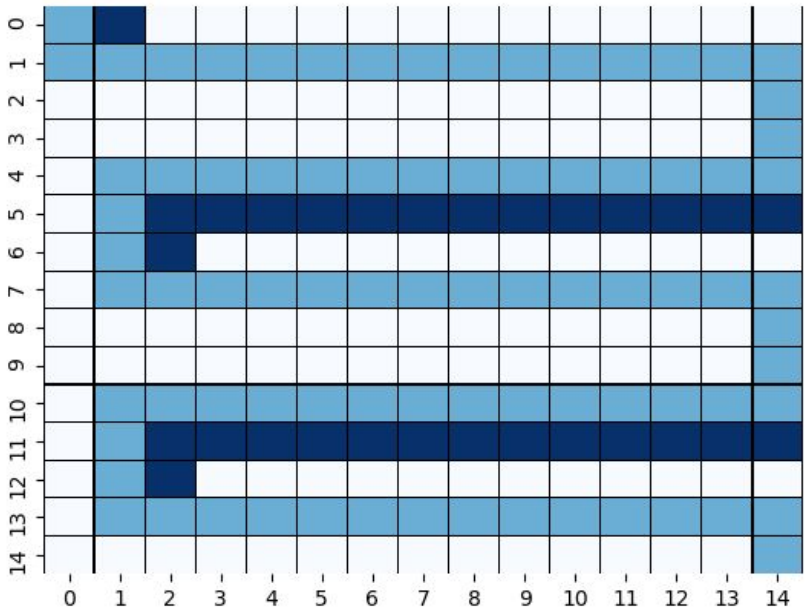
- **Hardest maze using Algorithm with paired metric**

  - **DFS with Maximal Fringe Size**

One of the hardest mazes we could think of for DFS with Maximal Fringe size for dimension 15 is as shown below. (This is also because of our implementation of DFS algorithm, where we traverse the nodes in order *"right=>bottom=>left=>top"*)



The path generated for this maze is shown in the figure below.



The maximum fringe size for solving this maze is 112. The number of cells in the maze we have = 15*15 = 225. Therefore, the number of cells in the maximum fringe is 49.7% of the total number of cells in the maze. This is evident in the figure below.

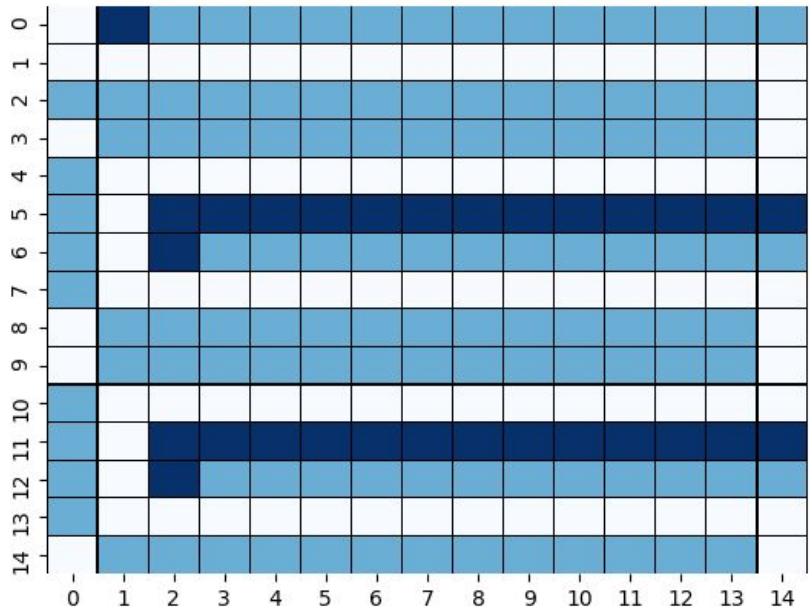


Fig: Cells in the maximum fringe

The fringe size keeps increasing when we add more cells to the fringe than the number of cells which we pop out from it. This happens in the ideal case mentioned above because the DFS algorithm always tries to explore the cells in the right => bottom => left => top order and due to this reason, it ends up ***not exploring most of the neighbours it adds to the fringe when it finds the solution***.

Design choices for generating children for DFS:
By observing the ideal hardest maze for DFS which contains blocked cells in horizontal fashion, we wanted to retain the properties of the rows when we merge parents to create children. Preserving the row property of parent will get us closer to the hardest maze. Hence, we have picked the technique of **combining two parents row-wise** at the crossover point.

Also, we wanted to add diversity to the children by mutating in an opposite way when compared to the parent concatenation. So we picked the technique of **swapping columns for child mutation**.

- **A*-Manhattan with Maximal Nodes Expanded**

Based on the way we have coded our a-star algorithm, a **blank maze is the hardest one** to solve if hardness is calculated based on maximal nodes expanded. For a blank maze, a-star is exploring all the nodes in the maze.

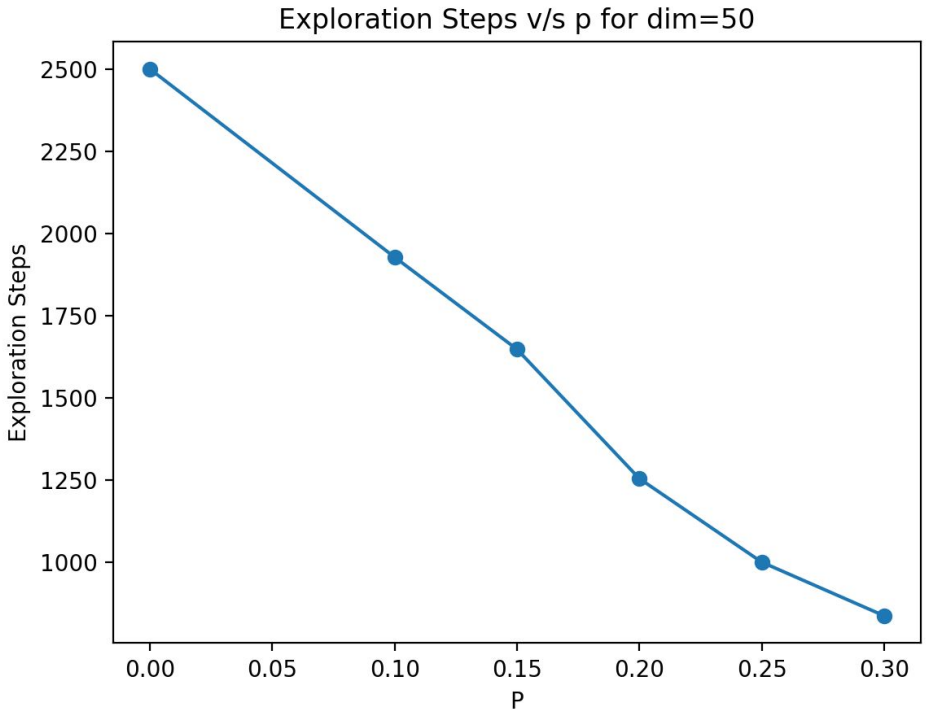
A* works better because instead of exploring all the cells it explores the cells which can potentially give a solution faster. **If the priority of each cell is the same, the A* algorithm will fail to optimise.**

All the cells in the blank maze have a similar priority based on the heuristic we are using currently.
f(n) = g(n) + h(n)

- g(n) = distance traversed from S to n (the actual path)
- h(n) = estimate of the minimum remaining distance from n to any node in G using Manhattan

For a blank maze, f(n) is constant over the entire maze.

This leads to more exploration by the A* algorithm.



For A*, the number of exploration steps is inversely proportional to p. Hence, lesser the blocks, more the exploration steps of A* algorithm.

Also, the way priority queue works is if two elements have the same priority, it prioritises lexicographically. Hence, the cells will be explored row wise. (0,0), (0,1), … (1,0), (1,1), … and so on.

Design choices for generating children for A*:
We have picked the technique of **combining two parents column-wise** at the crossover point. Also, we wanted to add diversity to the children by mutating in an opposite way when compared to the parent concatenation. So we picked the technique of **swapping rows for child mutation**.

We implemented bit flip mutation technique initially. After choosing the cells at random, flipped the values of cells (0=>1 or 1=>0) for a random number of cells, but we observed that the performance of the algorithm was not that good. As most of the cells in the maze were open cells, this mutation technique was simply adding more blocked cells. As we discussed above, the solution for this metric and algorithm is a blank maze. This mutation was taking the children away from the optimum solution by adding more blocks.

- **Result Intuition Comparison**

Yes, the results match our intuition.

1. **For DFS with maximal fringe size:**

We tried generating the hardest maze for DFS using different number of generations. The genetic algorithm implementation details:

1. dimension=15
2. probability = 0.2
3. population size=50
4. 50 crossovers

Few of the mazes we obtained are as follows.
Observation: In each of the samples we can see that, the mazes are trying to reach the hardest maze we described earlier.

Sample 1: Generations = 200, Fringe Length = 58 (Max. fitness)

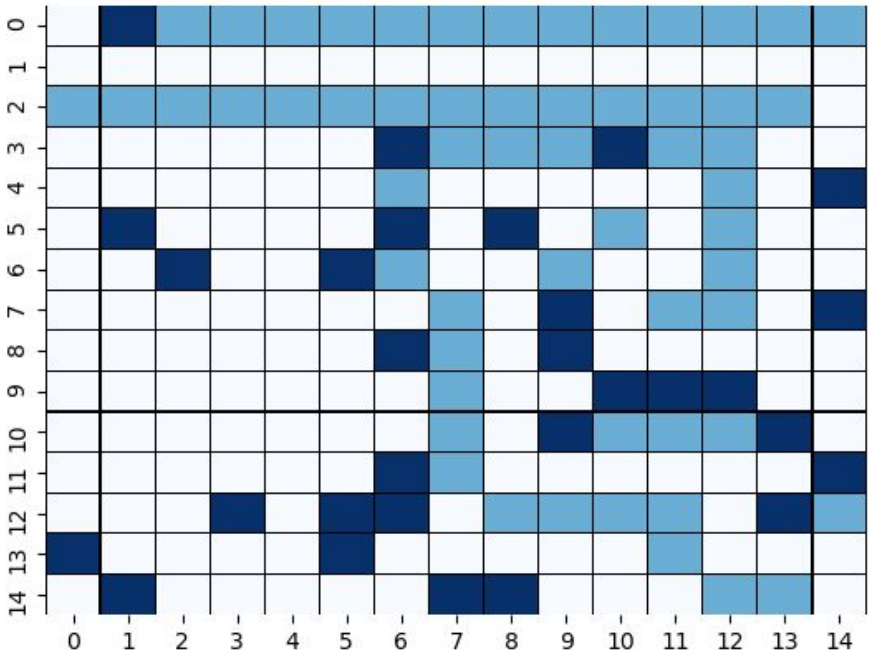


Fig: Cells in the maximum fringe

Sample 2: Generations = 400, Fringe Length = 61

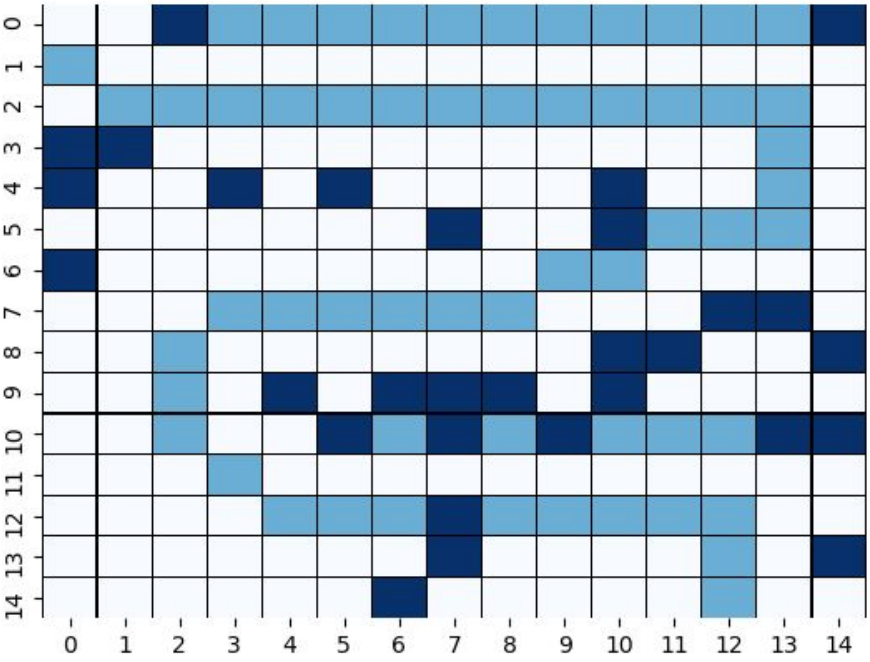


Fig: Cells in the maximum fringe

Sample 3: Generations = 500, Fringe Length = 67

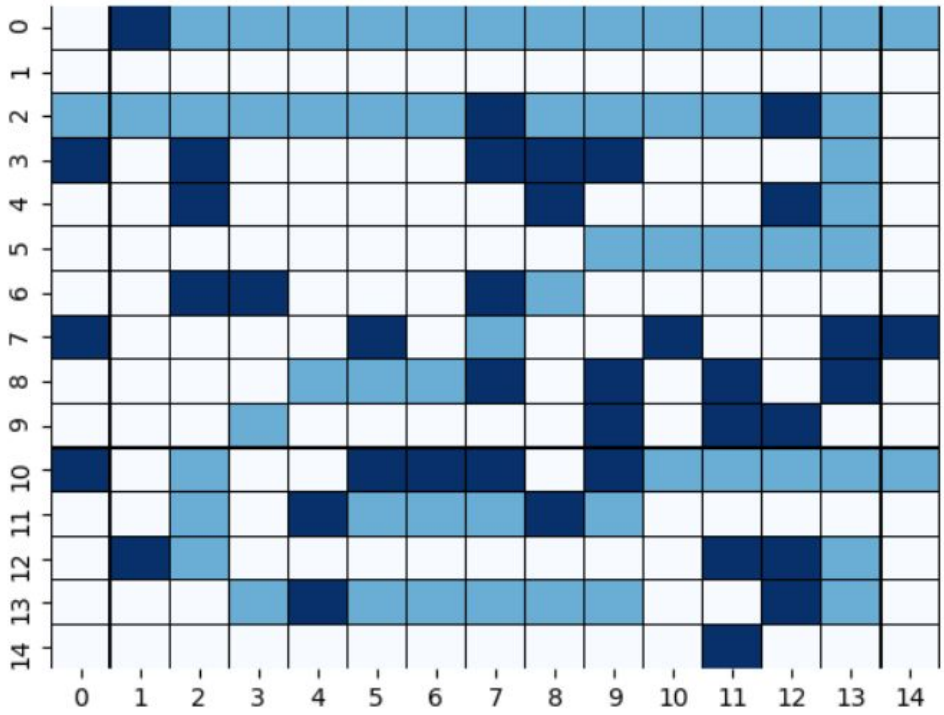


Fig: Cells in the maximum fringe

The maximum number of cells present in the fringe is represented in Sample 3. This is almost similar to the ideal maze scenario. The maximum fringe length is 67 which is equal to 35% of the total number of free cells.

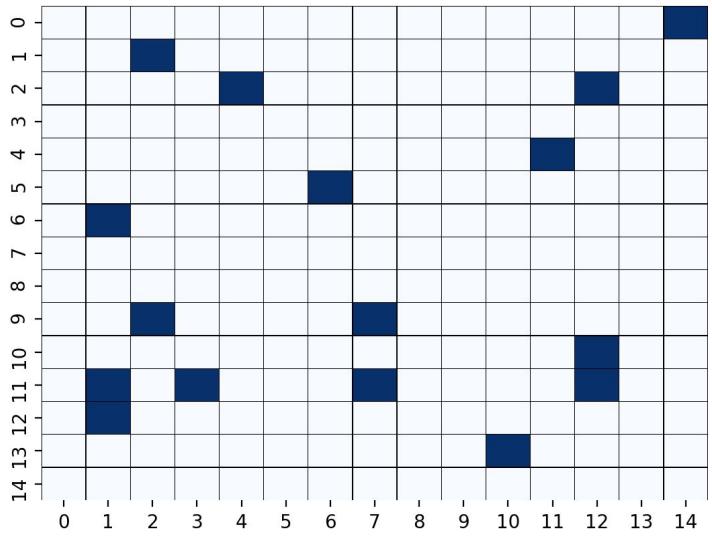
2. For A* with Manhattan:

We tried generating the hardest maze for A* using different number of generations. The genetic algorithm implementation details:

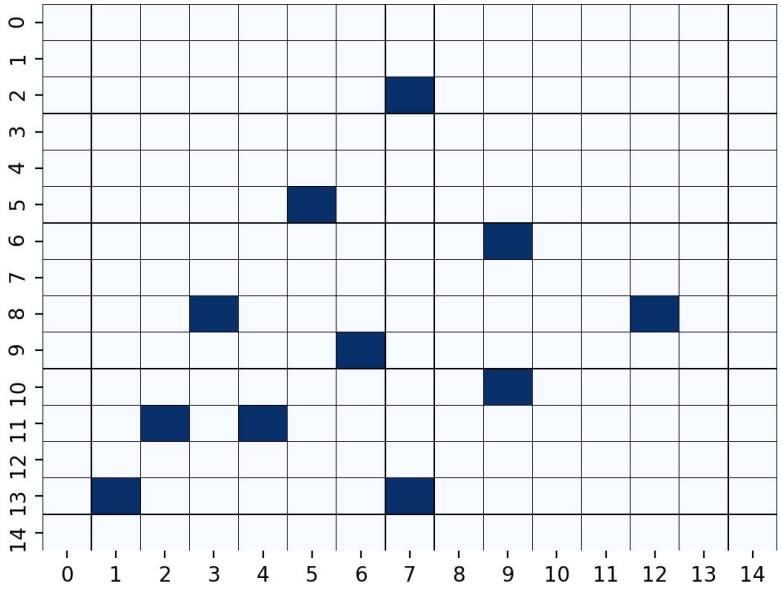
5. dimension=15
6. probability = 0.2
7. population size=50
8. 50 crossovers

Few of the mazes the algorithm generated were as follows.

Sample 1: Generations = 300, Max cells explored = 207
Total cells = 225 => Percent of cells explored = 92%



Sample 2: Generations = 500, Max cells explored = 212
Total cells = 225 => Percent of cells explored = 94.22%

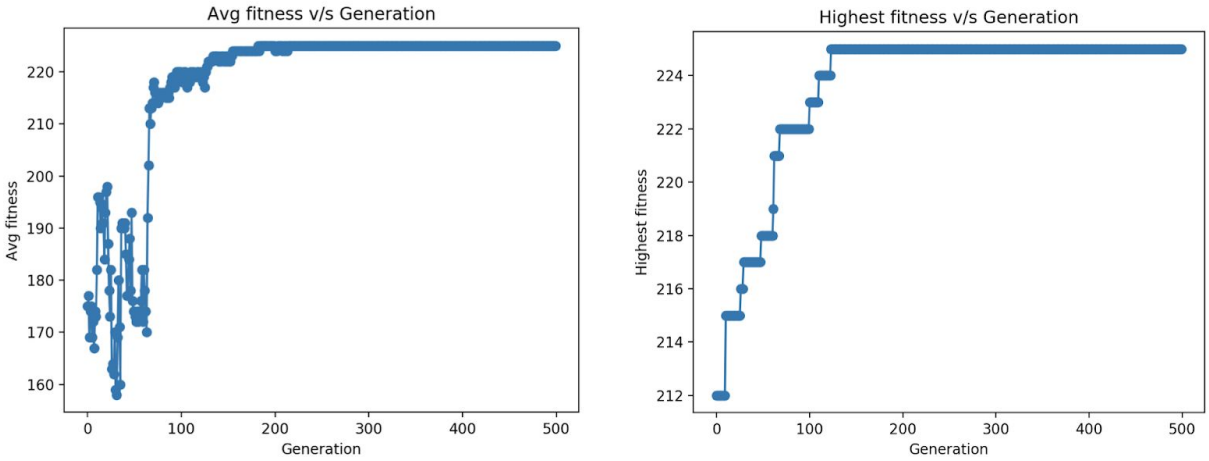


Blank maze is the optimum maze. And these samples are quite close to a blank maze.

Comments on Genetic Algorithm:

- If we know what the solution looks like, we can model the cross over, mutations and initial conditions for population generation in such a way that we converge to the solutions faster.

Consider the example below:
If we start the genetic algorithm with **p=0.1** rather than p=0.2, A* will converge to the best solution faster as it starts with a good initial population.



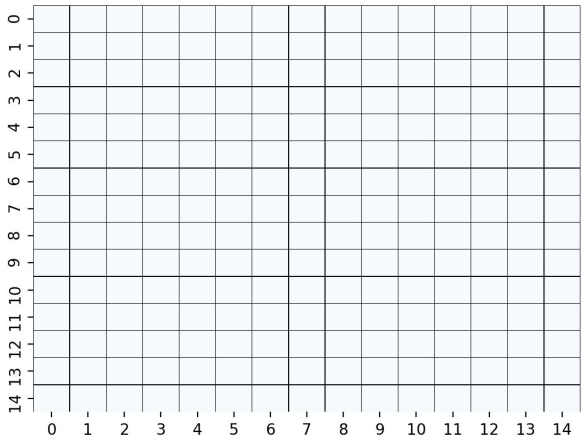


Fig: Hardest maze: Fitness = 225! 100% of the maze.

Future work:
We can run **hill climbing** for finding the hardest maze using a-star evaluating it based on the maximum number of nodes explored. The **neighborhood** of the maze can be defined as the **number of blocked cells in the maze**. This way we can reach the optimum solution which is a maze with zero blocked cells.

4 What If The Maze Were On Fire?

Idea

For performing in a better way, before adding an element in the priority queue, we can simulate the fire for a few time steps i.e. try to think how the fire will spread in the future and then take a decision at a particular time.

In the baseline, we are not considering the presence of fire at all. We are just concerned with finding the shortest path and hoping that we do not catch fire.

When the fire is present and spreading, we need to avoid getting burnt in addition to trying to find the goal state. Including measures to stay clear of fire can help us do better.

With maze being on fire, we now have two goals:

1. reach goal state
2. not get burnt

We can try not to get burnt by avoiding the fire that can be achieved by somehow penalizing the algorithm for taking steps which moves us towards the fire cluster.

Approach

One way we can penalize our algorithm is by modifying the heuristics such that we favor those cells that are further away from fire. However, while doing so, we need to make sure that our first goal is not ignored and we are continuously trying to reach the goal state.

Hence we have used the following function to decide which node should be popped out of the fringe first:

```
For any cell [x, y] in the fringe:
     Find cell [a, b] such that [a, b] is on fire and the
manhattan_distance([x, y], [a, b]) is minimum
     priority([x, y]) = (manhattan_distance([x, y], goal node) -
manhattan_distance([x, y], goal node))
```

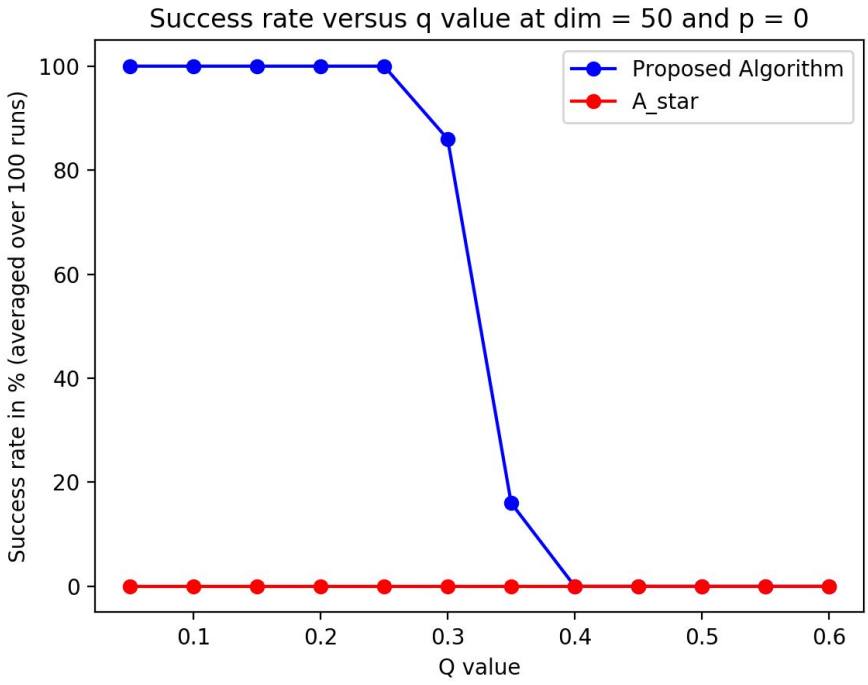
Using the above priority function helps us make sure that we are always trying to move towards the goal while making sure we stay clear of the fire. It is crucial to make sure our moves are not just about avoiding getting burnt, and we are continually moving towards the goal. If we only try staying clear of the fire, we potentially end up taking more time to reach the goal. In that case, we end up giving more time to fire to spread all across the maze and resulting in increasing the probability of us getting burnt.

Since the fire spreads after every time-step, we need to make sure the priority value of the cells in the fringe also get updated after every time-step. On the other hand, continuous updates to the fringe could turn out to be computationally expensive because we would essentially have to find the nearest burning cell each time for each of the cells in the fringe. Instead of using BFS to find the nearest burning cell, we can just keep track of the cells which are present at the edge of the fire cluster and simply pick the one which has the least manhattan distance from the selected cell.

Analysis with varying p

The following figure shows the comparison between the baseline (A* using manhattan distance as the heuristic) and the proposed algorithm.

For p=0:



When there are no blocks in the maze, the baseline approach always fails but the proposed algorithm always works for low q values.

Explanation for the same can be found in the plots for the path below.

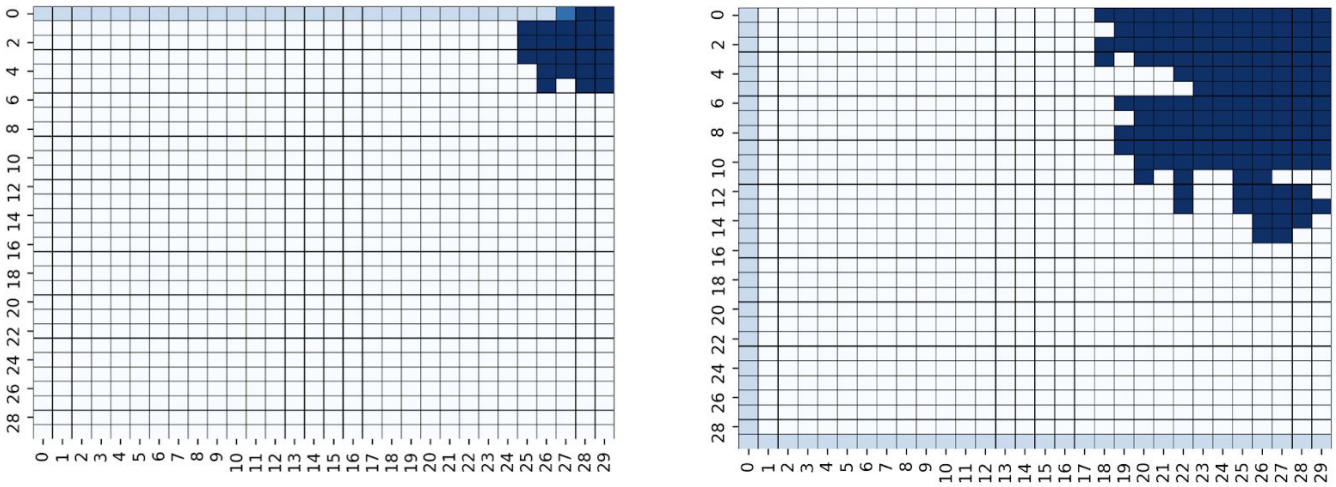
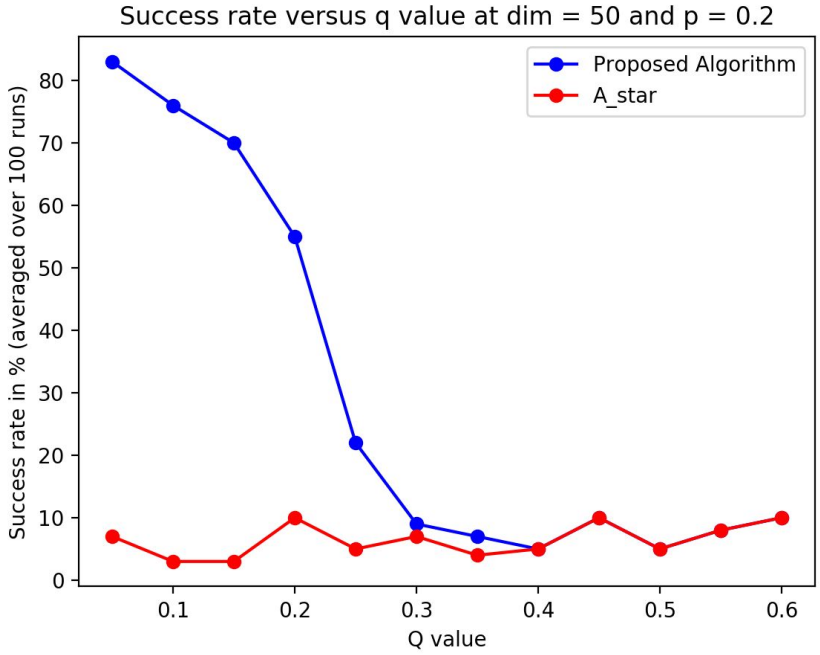


Fig: Left- Path taken by normal a-star, Right- Path taken by proposed algorithm.

As it is clear from the image, a-star baseline runs into the fire, whereas the proposed algorithm tries to stay away from the fire and also reaches the goal.

In the worst case, fire has to travel **dim** steps to block the goal cell whereas the shortest path to be travelled from source is **2*dim.** Hence, for q>=0.5 It is impossible to reach the goal.

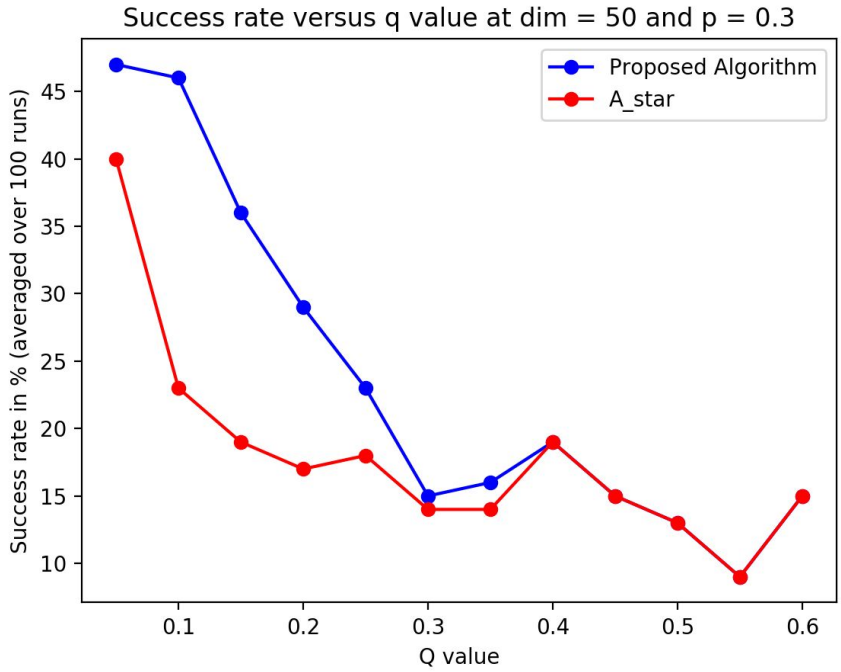
For p=0.2:



It can be observed that the proposed algorithm and A_star have same performance metrics for q >= 0.4. The potential reason behind the pattern could be that the cases which got solved were

those where the fire was blocked by the block cells and thus could not spread. For most of the cases, both the algorithms failed.

For p=0.3:



For the value of p0=0.3, we can see that the success rate of the proposed strategy is nearly 50% which is slightly better than the baseline approach.

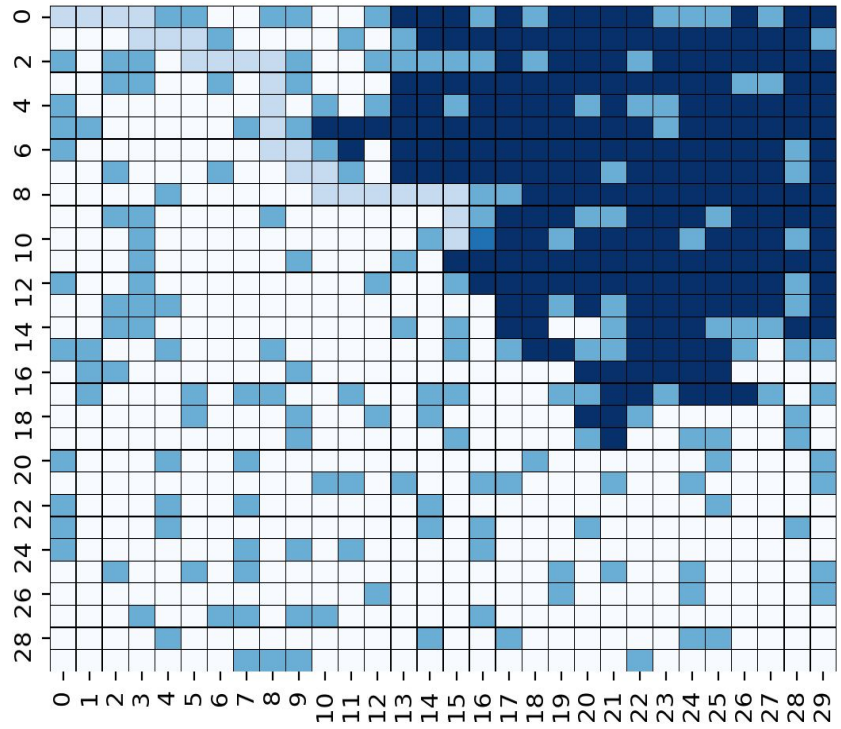
Changes with respect to p:

- Success rate of a-star base line increases with increase in p.
- Success rate of proposed algorithm decreases with increase in p.
- The difference in the success rate for both the algorithms decrease with increase in p.

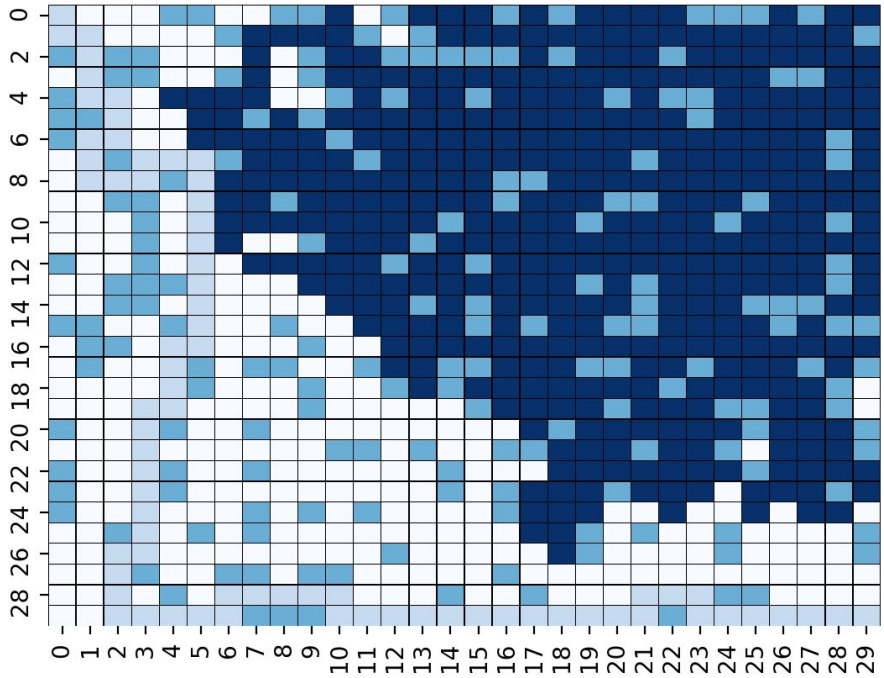
Visualising baseline with proposed approach

The following two figures show a case where we get burnt if we follow the baseline but are able to reach the goal state if we follow the proposed algorithm.

The figure below represents the path and the maze when we follow the a_star algorithm. As can be seen, we end up getting burnt at (10, 16).



The figure below represents the path and the maze when we follow the proposed algorithm.



Key Observations:
In the proposed algorithm as we can see, we are trying to take a path that is trying to follow a path that is away from fire.

Potential Improvements:

- The proposed strategy could be improved if we add the capability of looking into the future (say five time-steps ahead). Let's say we have x nodes in the fringe. Pick top y cells and for each cell, simulate the search for five time-steps using the priority function. This simulation would give us y potential fringes. Compare the fringes generated by each of the y paths and choose the actual next step using the path whose best cell in its fringe is better than all the cells in all the other paths. By using this approach, we are utilizing the information contained in the q value and the behavior of the fire mentioned in the question to make more informed decisions.

- Current priority function considers the least manhattan distance from the fire cell. The function can be made more accurate by running a BFS search for the fire cells to get the actual minimum distance from the fire. We actually implemented this approach but it turned out to be computationally costly so we had to think about alternate approaches.

Division of Work

Work	Team Member
Brainstorming and Discussions	All Team Members
● A* with Manhattan and Euclidean Heuristics ● Genetic Algorithm for A* with maximum number of nodes explored ● Hardest maze analysis for GA ● Fire q_value v/s p analysis ● Visualization methods reusable by all algorithms	Twisha Naik (tn268)
● BFS ● BD-BFS ● Average shortest path length analysis and calculation for [0,p0] ● Genetic Algorithm for DFS with maximal fringe size ● Visualization method for BD-BFS ● Generating hardest maze for GA (DFS)	Hari Priya Ponnakanti (hp467)
● Maze Generation ● Generating dim and p0 values and plots ● DFS ● Fire Maze Simulation: Analysis and Algorithm	Aditya Lakra (al1247)
Documentation	All Team Members
Analysis and Plots	All Team Members