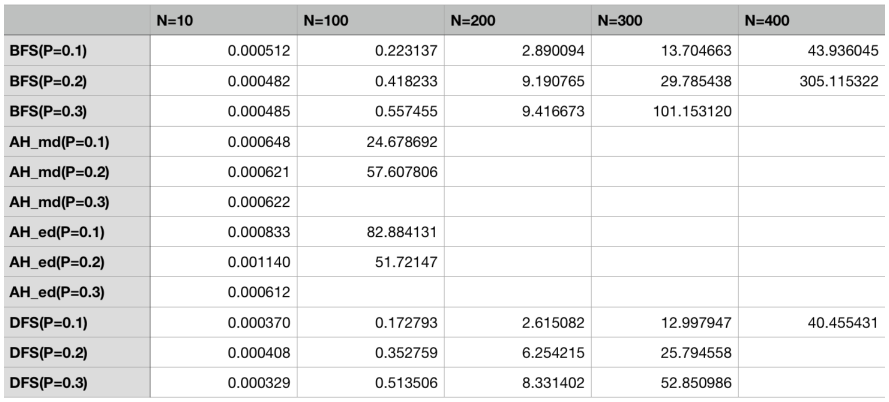
AI Assignment-1

October 3, 2017

1 Part 1

1.1 Answer 1.

For the algorithms that we implemented for different values of p and size ($dimXdim$) of the map, the following table shows the run times. Time represented is in **seconds**.

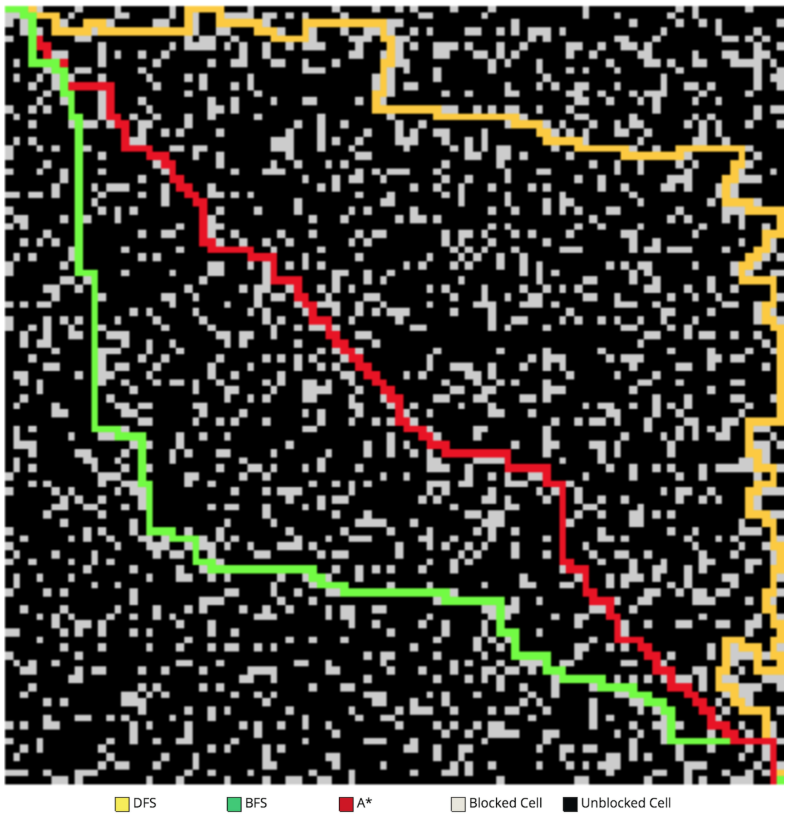


	N=10	N=100	N=200	N=300	N=400
BFS(P=0.1)	0.000512	0.223137	2.890094	13.704663	43.936045
BFS(P=0.2)	0.000482	0.418233	9.190765	29.785438	305.115322
BFS(P=0.3)	0.000485	0.557455	9.416673	101.153120	
AH_md(P=0.1)	0.000648	24.678692			
AH_md(P=0.2)	0.000621	57.607806			
AH_md(P=0.3)	0.000622				
AH_ed(P=0.1)	0.000833	82.884131			
AH_ed(P=0.2)	0.001140	51.72147			
AH_ed(P=0.3)	0.000612				
DFS(P=0.1)	0.000370	0.172793	2.615082	12.997947	40.455431
DFS(P=0.2)	0.000408	0.352759	6.254215	25.794558	
DFS(P=0.3)	0.000329	0.513506	8.331402	52.850986	

Since we needed to run the code multiple times, we concluded that working with maps of size ($100X100$) is most reasonable for every algorithm.

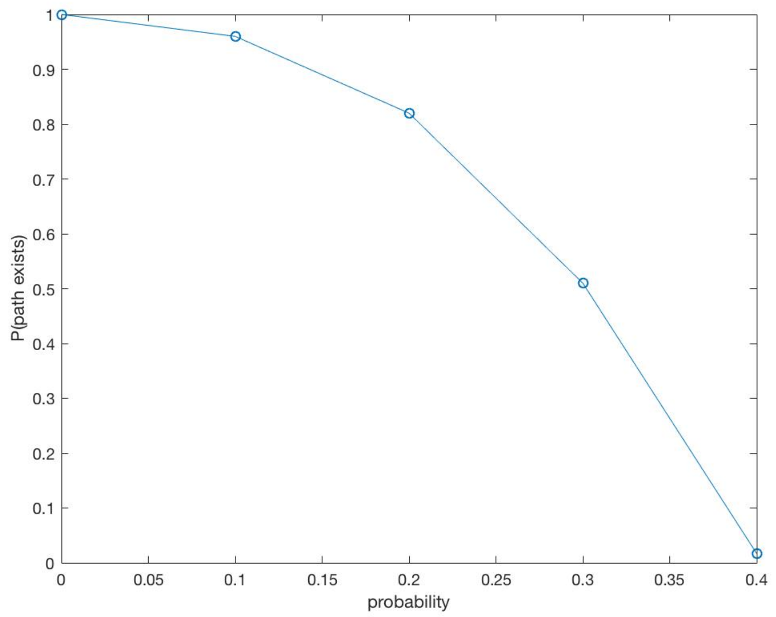
1.2 Answer 2.

We generated a random map for $p = 0.2$ and ran BFS/DFS/A* on it and the paths along with the map are shown below:



1.3 Answer 3.

For dim = 100, we generated 100 random maps as a sample for estimating the mean probability of a path existing in the map for $p = 0.1, 0.2, ...$ and so on and computed the number of maps which had a path. We then generated the sample of 100 mazes multiple times to get a better estimation of probability values and to reduce the risk of working with an outlier in the sample mean population. The estimated probability say $p^{'}$ such that a random map generated with $p = k$ where $k = \{0.1, 0.2, ..., 0.9\}$ are:



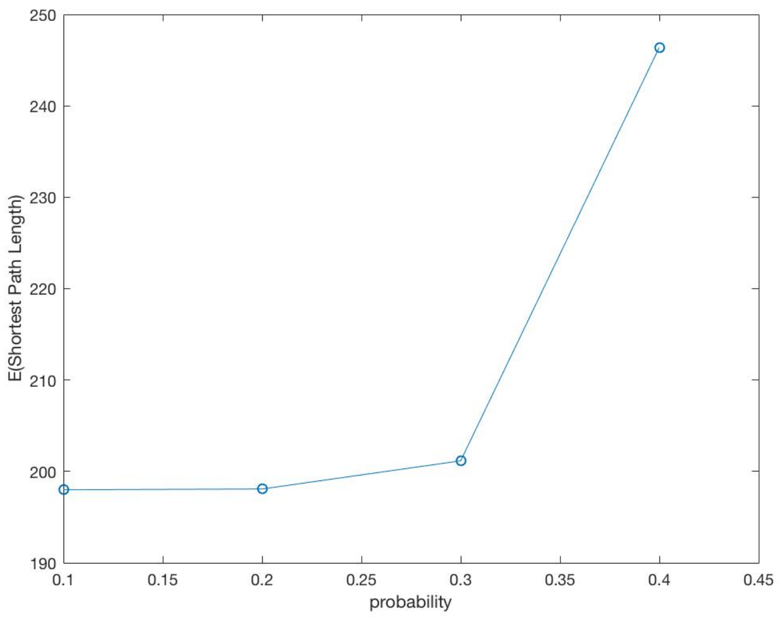
At $p = 0.1, 0.2, 0.3, 0.4$, the average number of maps for which a path was found were $96, 82, 51$ and 5. Therefore, the mean probability of finding a path are $0.96, 0.82, 0.51$ and 0.017 respectively. For values of probability greater than 0.4, the maps become filled more and the path usually does not exist for a sample of 100 random maps chosen for experiment.

As we can see, for $p = 0.3$, 51 maps out of 100 were solvable and the number drastically dropped down to 1.7 at $p = 0.4$. Therefore, the threshold probability p_0 for which there is $'usually'$ a path for all $p < p_0$ is 0.3 as about half the number of maps have a path from start to goal.

1.4 Answer 4.

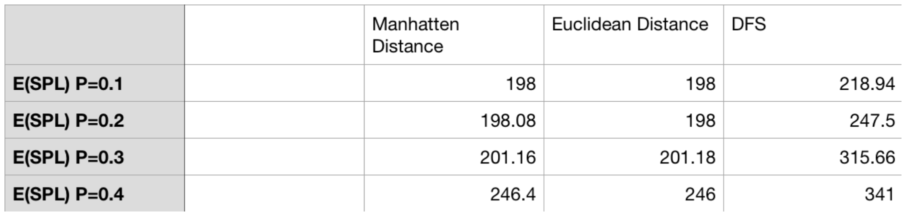
For estimating expected value of shortest path length from start to goal, We kept generating random maps until the sample size reached significant (100 in our experiment) for $p < p_0$ i.e. up to $p = 0.3$ and we computed average length of shortest path returned for each p.

We used A^* as our choice of algorithm for this experiment as A^* is optimal and returns the shortest possible path for a given maze.



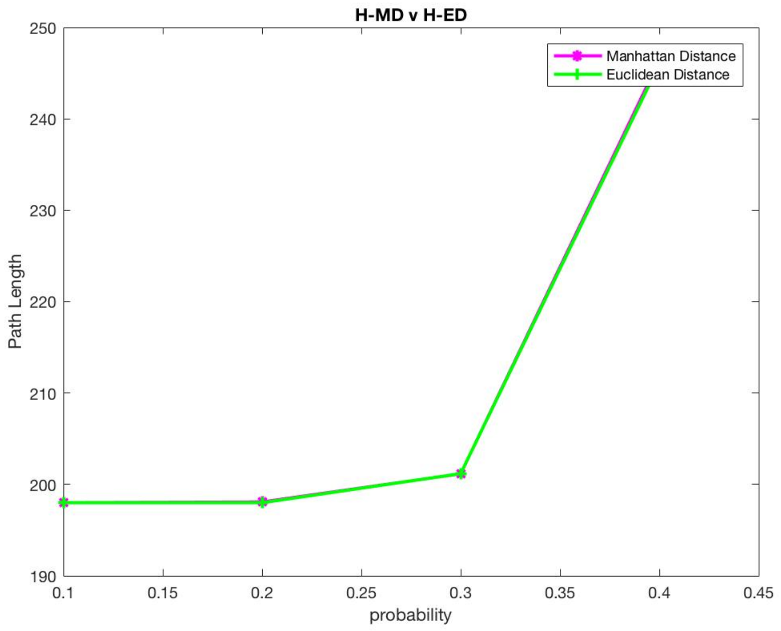
1.5 Answer 5.

We generated the maps and solved it to find a path until we get a significant number of mazes (100) to compute mean path length. Average length of path Generated by A^* using heuristic functions involving Manhattan Distance and Euclidean Distance and DFS are as follows:

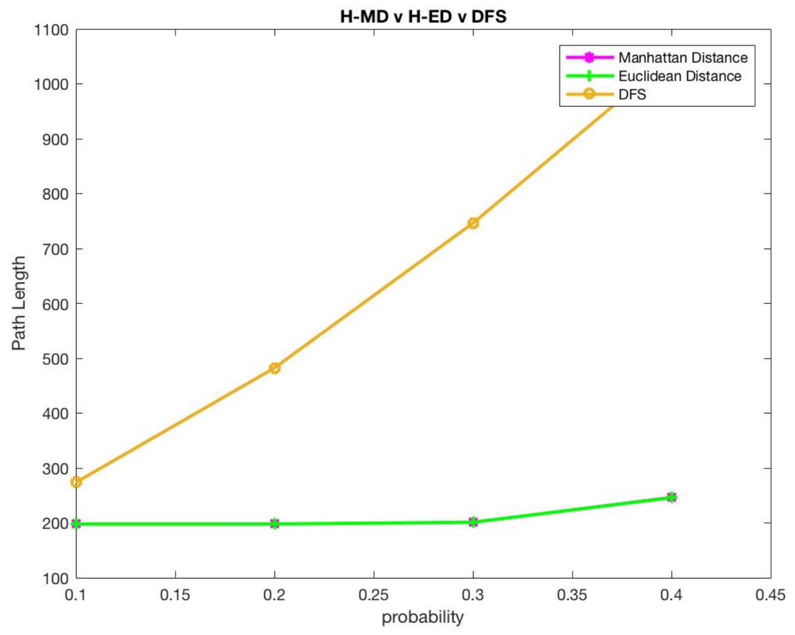


		Manhatten Distance	Euclidean Distance	DFS
E(SPL) P=0.1		198	198	218.94
E(SPL) P=0.2		198.08	198	247.5
E(SPL) P=0.3		201.16	201.18	315.66
E(SPL) P=0.4		246.4	246	341

As we can see, the lengths of paths returned by each heuristic are very close as both heuristics are admissible for the environment finds the optimal path.



DFS, on the other hand, finds first available path out of many paths in the given map which is random based on a random map and is solely dependent on arrangement of blocked nodes in the map. If the goal node is reached early, it stops and if not, it keeps on expanding the nodes until the goal nodes is included in the fringe of a node. DFS compares with A^* for heuristics as follows:

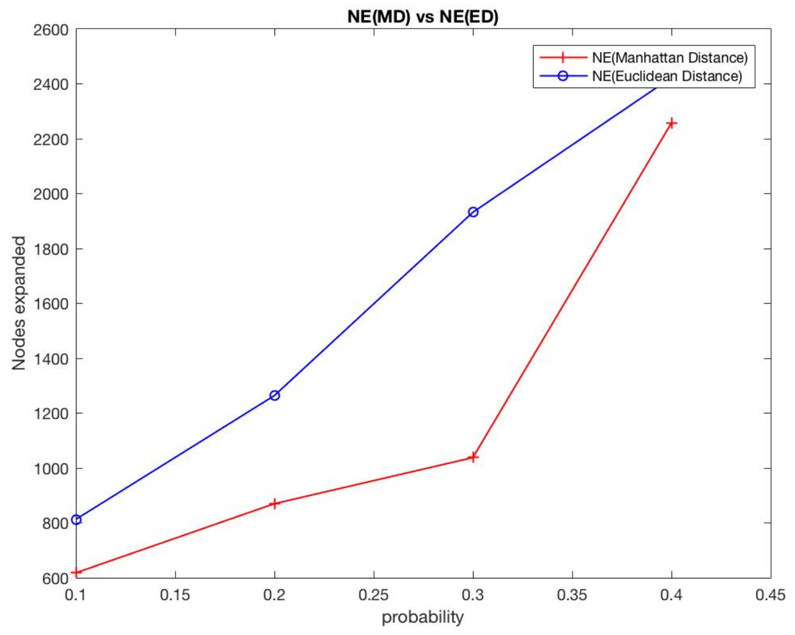


1.6 Answer 6.

We ran the algorithm A^* multiple times for both heuristics, Manhattan Distance and Euclidean Distance until we found a path for a significant number of maps and took the mean on number of nodes expanded for each map in which a path was found. We discarded the mazes where where no path was returned.

		MD	ED
E(NE) P=0.1		618	813
E(NE) P=0.2		870	1264
E(NE) P=0.3		1038	1932
E(NE) P=0.4		2257	2423

Since both the heuristics are admissible, both returns optimal paths but through different patterns of traversals in the maze which leads to different number of nodes traversed. Both the heuristics compare as follows:



As evident from the comparison plot, using Manhattan distance as the heuristic expands lesser nodes than using Euclidean distance as the heuristic. In our algorithm A^* for both the heuristics, we prioritize the nodes in priority queue based on score value of the node computed from the sum of true distance up to that node and estimated distance to the goal. The node with a greater score is given lower priority than the node with the lesser score and is visited after the nodes having score values lesser than it's score value.

Claim: Euclidean distance gives a greater score to the nodes nearer to the goal except nodes in last row and last column and therefore, nodes farther from the goal are visited before eventually leading to more node expansions.

Proof: It is trivial that the scores will be higher and the priorities will be lower for the upper and left neighbor for a given node in a map or 2-D array in this case thus it suffices to prove the claim for the lower and right neighbor. Let the random map generated be a (dim X dim) 2-D array where i, j^{th} element represents a node. Suppose, score values for each node are computed using function $f(i,j) = g(i,j) + h(i,j)$ where $g(i,j)$ represents the true distance or distance covered until i, j^{th} node and $h(i,j)$ represents the estimated distance using either heuristic.

For Manhattan Distance, $h(i,j) = (dim - i) + (dim - j)$. When expanding a node and computing scores of it's neighbors in it's fringe, we will now prove that the $(i+1), j$ node and $(i, (j+1))$ nodes get a score value higher and hence a lower priority in the priority queue. Let $g(i,j) = k$ and $g(i+1,j) = g(i,j+1) = k+1$. $h(i,j) = \sqrt{(dim-i)^2 + (dim-j)^2}$ for node (i,j).

$$h(i+1,j) = \sqrt{(dim-1-i)^2 + (dim-j)^2}$$
$$h(i,j+1) = \sqrt{(dim-i)^2 + (dim-1-j)^2}$$

Therefore,

$$f(i+1,j) = k+1+\sqrt{(dim-1-i)^2 + (dim-j)^2}$$
$$f(i,j+1) = k+1+\sqrt{(dim-i)^2 + (dim-1-j)^2}$$

The term in the square root decreases for both the closer to the node neighbors but adding 1 dominates the decrement. We will prove it below.

Let $Q = (dim-i)^2 + (dim-j)^2$. We will try to prove the claim using contradiction. Assume $f(i,j) > f(i+1,j)$. Therefore,

$$k + \sqrt{Q} > (k+1) + \sqrt{Q + 1 - 2(dim-i)}$$
$$\sqrt{Q} - 1 > \sqrt{Q + 1 - 2(dim-i)}$$
$$Q + 1 - 2\sqrt{Q} > Q + 1 - 2(dim-i)$$
$$\sqrt{Q} < (dim-i)$$
$$Q < (dim-i)^2$$
$$(dim-i)^2 + (dim-j)^2 < (dim-i)^2$$

If $dim \neq j$, then we get a contradiction. Similar proof can be given for $f(i,j+1)$. This implies, the two closer nodes to the goal in the fringe of a certain node gets higher score and hence lower priority except the nodes in last row and last column where the closest neighbor to the goal gets equal priority and gets visited next after the current node. Thus using Euclidean distance as heuristic will always return lower priority values for every node in the fringe of a given node and therefore, the nodes at the exact same level of the given node will get visited first than moving down a level in search tree. Thus, it almost imparts BFS like characteristics to the algorithm.

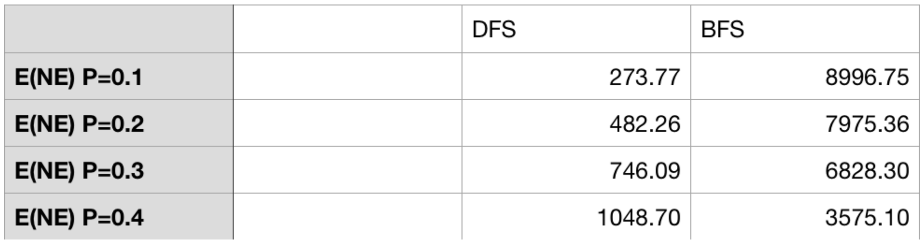
On the other hand, for Manhattan distance as the heuristic, $f(i+1,j) = f(i,j+1) = (k+1) + 2(dim) - i - j + 1$ which is $f(i+1,j) = f(i,j+1) = k + 2(dim) - i - j$. It is the same score as node (i,j) and therefore, the two closest neighbors get the same priority in the priority queue. Thus, being smart with the algorithm and making the algorithm visit the nodes in the next lower level in search tree first, before making it visit the other nodes in with the same priority in the same level as given current node whose fringe is being explored, decreases the number of nodes expanded . Thus using Manhattan distance as heuristic imparts DFS like characteristics to the algorithm with optimality.

Thus, we can say that average number of nodes expanded in Manhattan Distance as heuristic function are lesser than the average number of nodes expanded in Euclidean Distance as heuristic function.

The similar pattern follows for p values above p_0 as more nodes will be blocked/filled and average length of fringe for each node will decrease but the traversal behavior remains same.

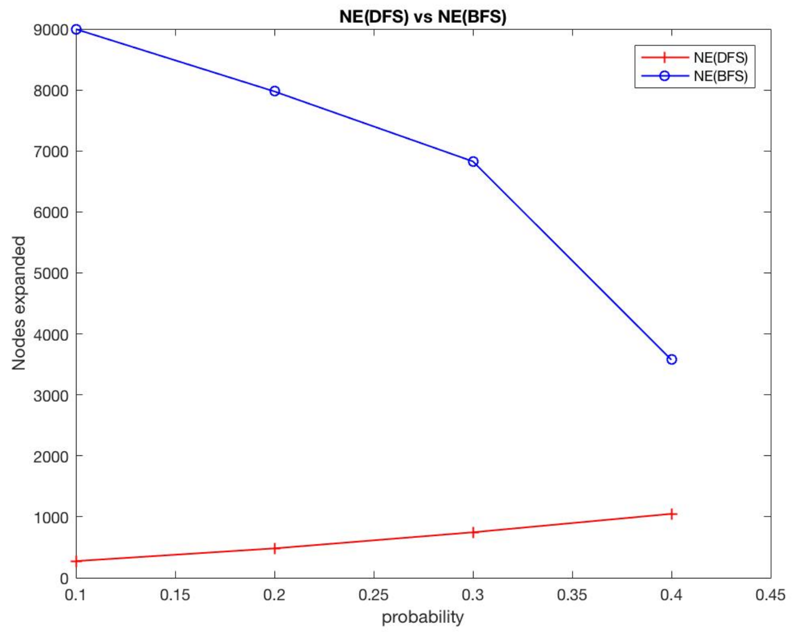
1.7 Answer 7.

We ran BFS and DFS on multiple maps until we found 100 maps with paths from start to goal and we took the mean on number of nodes expanded for each such map. The average number of nodes expanded in BFS and DFS are as follows:



		DFS	BFS
E(NE) P=0.1		273.77	8996.75
E(NE) P=0.2		482.26	7975.36
E(NE) P=0.3		746.09	6828.30
E(NE) P=0.4		1048.70	3575.10

Clearly, DFS expands fewer nodes than BFS because it moves down the search tree and returns the first path found first visiting all the closer nodes to the goal while it backtracks. BFS, on the other hand, visits every other node on the same level of search tree thus expanding more nodes than DFS. BFS for increasing values of p encounters more blocked/filled nodes and therefore, skips them in the process of fringe exploration and therefore, it expands lesser nodes as p increases. They compare as follows:



A^* is optimal, therefore, it visits a node first through which the shortest path is more likely using heuristic functions. Thus, it expands more nodes than DFS in an average case for a randomly generated map as DFS keeps going down the search tree levels but A^* may need to explore more number of nodes farther from the goal than DFS as DFS always first explores nearest nodes to the goal while backtracking.

BFS, on the other hand, explores every node on the same level and has no possibility of jumping a level down in the search tree until all the nodes in the same level are exhausted. A^* does not have this constraint and can jump down a level in the search tree, infact, it 'intends' to, based on the score value returned from heuristic functions by giving higher priority values to nearer nodes to the goal, if possible. Thus, BFS explores more nodes than A^*.

2 Part 2

2.1 Answer 8.

Our choice of local search algorithm is **Genetic algorithm**.
The space to be searched is not so well understood and relatively unstructured since the arrangement of the blocked cells in the maze changes randomly every time a maze is created. In this case an effective Genetic algorithm representation of this space can be developed, and Genetic algorithm can provide powerful heuristic for this large, complex space. Genetic algorithm detects biases

in lower-order schema, creates difficult maze partitions, and converges on this part of the search space. Eventually, biases in higher-order schema are detected by combining information from low-order schema via crossover, and a difficult maze is generated. We implemented it as follows.

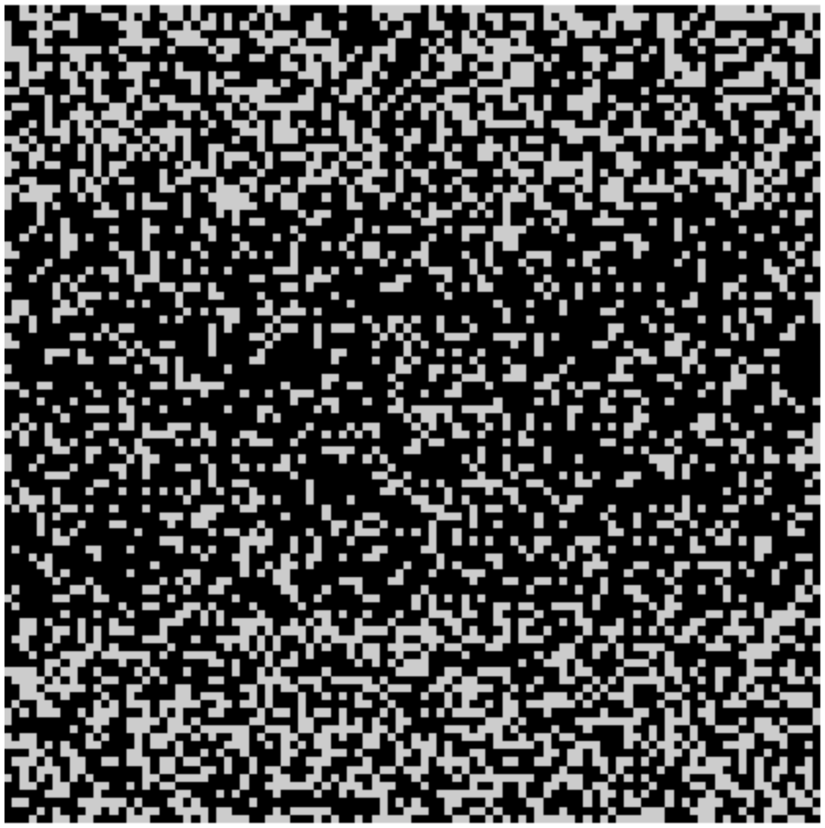
The maze is divided into three sections.
First section is from index 1 to n/4.
Second section is from (n/4)+1 to 3(n/4).
Third section is from 3(n/4)+1 to n.

The first and the third sections have a different difficulty level than the second part. That is, the Probability of a cell being blocked in the first and the third sections is given by "top_prob" and the probability of a cell being blocked in the middle section is given by "middle_prob".

"top_prob" and "middle_prob" have different initial values and also their rate of change overtime is different. Initially, top_prob = 0.4 and middle_prob = 0.6 Every time the search algorithm is able to solve the maze with the current values of "top_prob" and "middle_prob", their values are incremented by 0.005 and 0.01 respectively. Overtime, "top_prob" and "middle_prob" will reach a value for which our search algorithm will not be able to find a path even after running on 100 different mazes. Then we will conclude that a hard maze has been constructed with the given "top_prob" and "middle_prob".

We have designed our maze in such a way that it can be thought of as a maze with different sections of different and varying probabilities of a cell being blocked. The important decisions we had to make included determining the initial values of the probabilities, ratio of partitions and rate of change of probabilities.

One of such 'hard' map is shown below:



2.2 Answer 9.

The algorithm generates a maze for the top_prob and middle_prob. Then it runs either DFS or BFS or A^* to determine if a path is found or not for those specific probability values. If a path is found, it alters the values of probabilities of the different partitions to make them even harder.
It increments the probability of the top and bottom section by 0.005 and of the middle section by 0.01.
Then runs it again till the probabilities reach a value when no path is found. If a path is not found, it increments the value of the difficulty counter and runs it again for the same probability values. If no path is found after running it 100 times for the same probability values, then it implies that a hard maze is found.

2.3 Answer 10.

Consider two different mazes **A** and **B**. Also consider that the total number of nodes expanded for A is more than the total number of nodes expanded for B i.e. $NE(A) > NE(B)$. Then, we can conclude that the maximum size of the fringe list for A will be more than the maximum size of the fringe list for B.

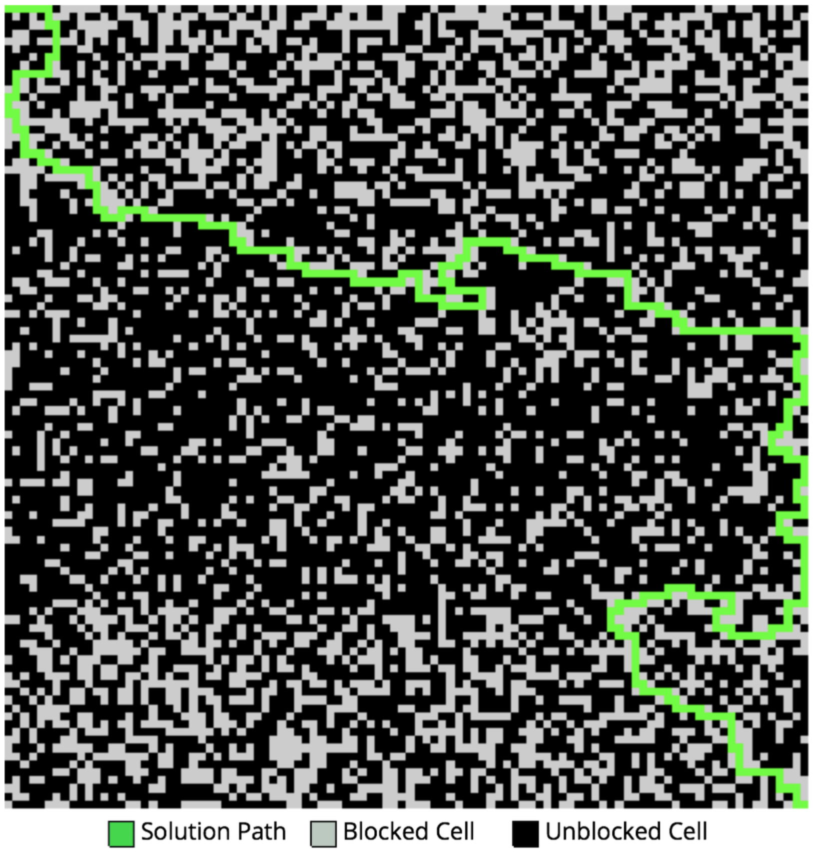
Our algorithm finds the first possible probability values for which no path is possible by creating 100 different configurations for those specific probability values and not being able to find a solution path for those configurations. Thus implying that their is no path possible when the probability is set to those values.

Then the ”top_prob” and ”middle_prob” are decremented by 0.005 and 0.01 to create a maze for which a solution path is possible and concluding that it is possible to solve but is ”hardest” maze.

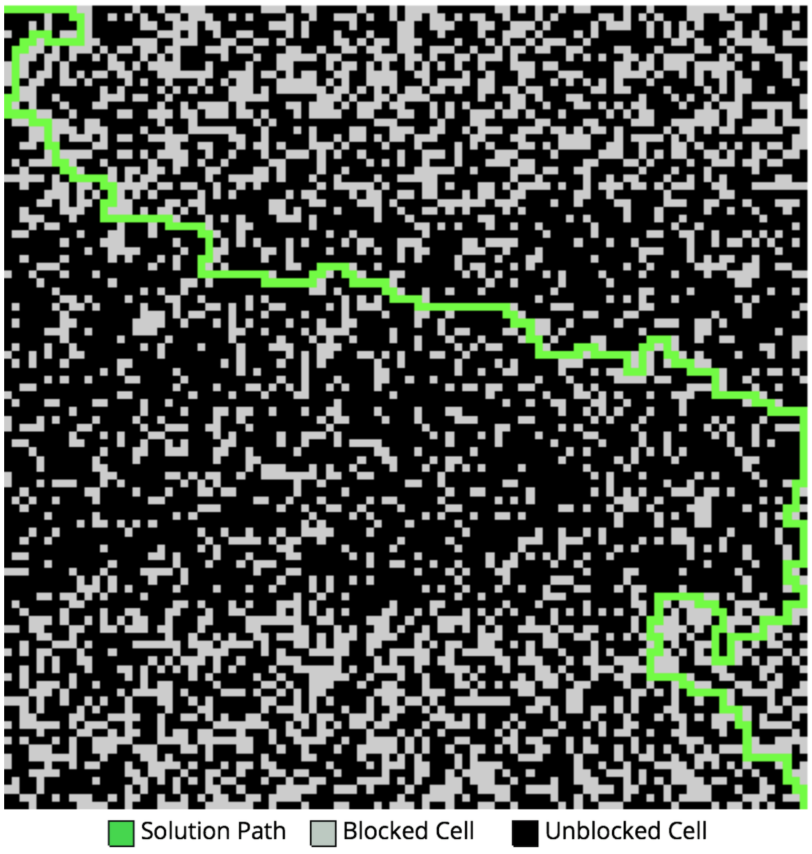
We will now present the maps maximizing the specific properties for various algorithms.

2.3.1 DFS:

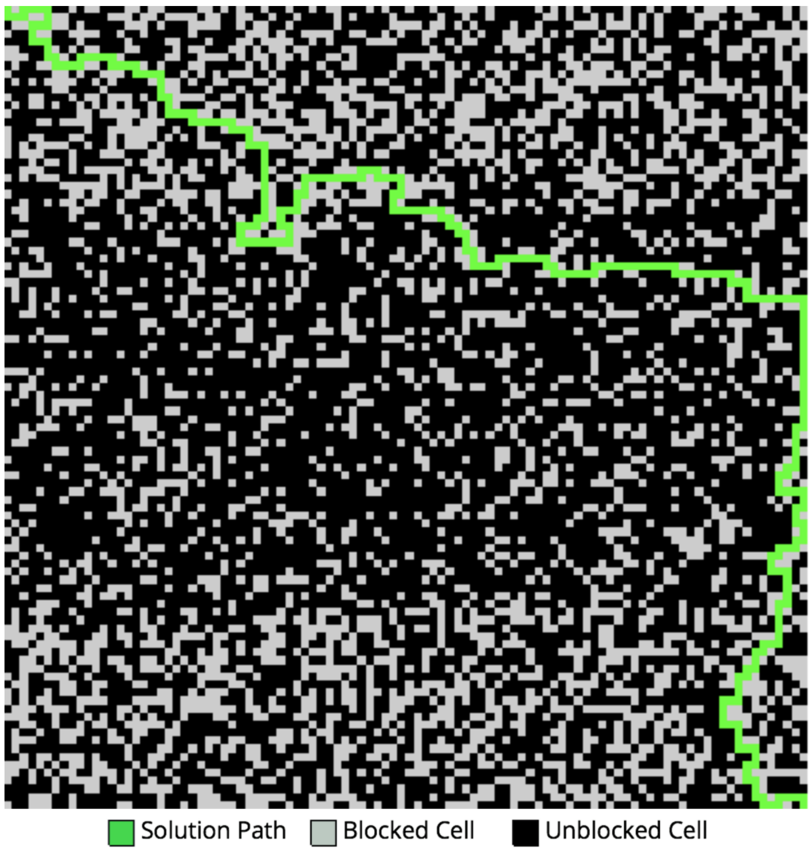
Length of solution path returned:



Total number of nodes expanded:

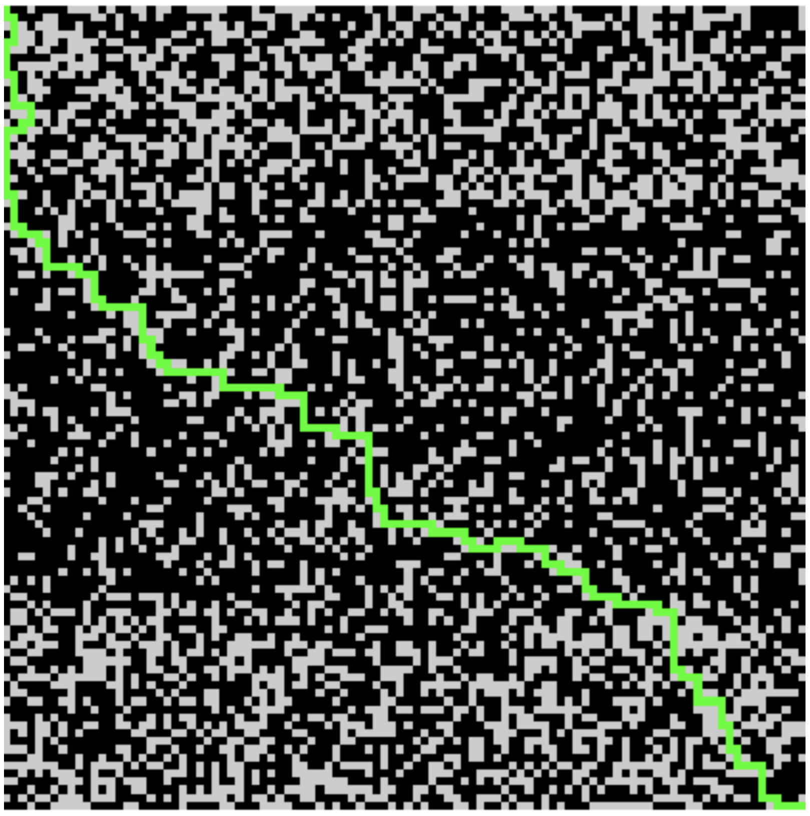


Maximum size of fringe during runtime:

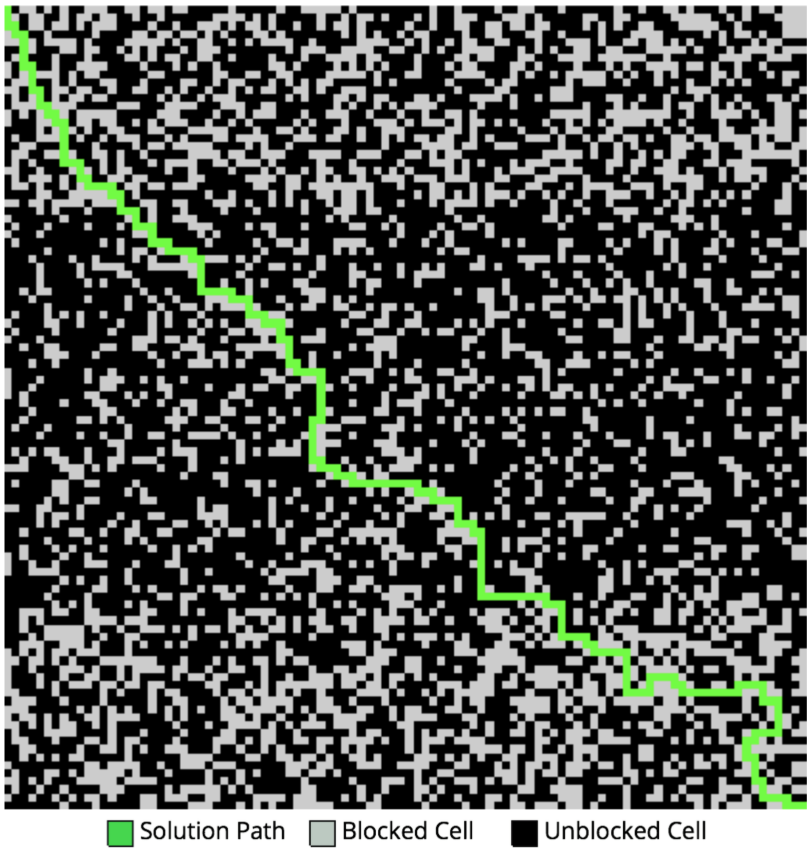


2.3.2 BFS:

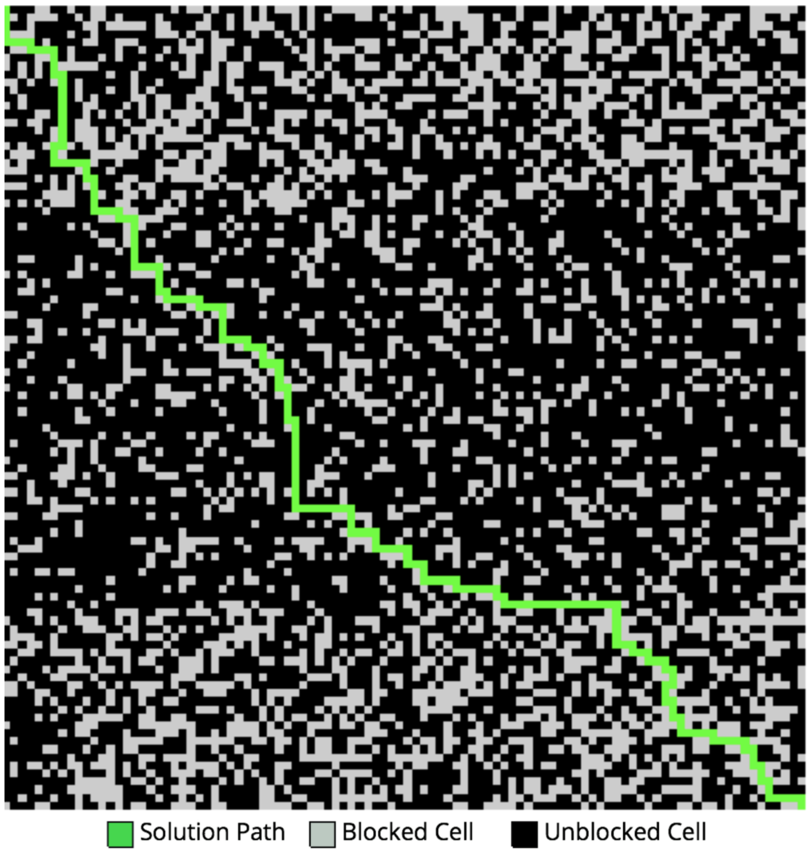
Length of solution path returned:



Total number of nodes expanded:



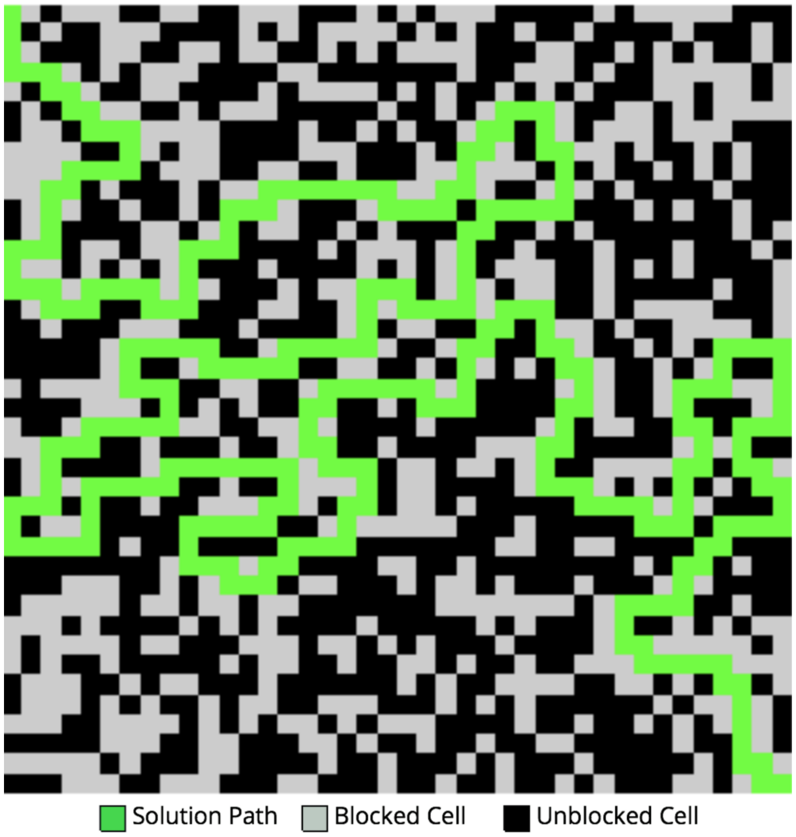
Maximum size of fringe during runtime:



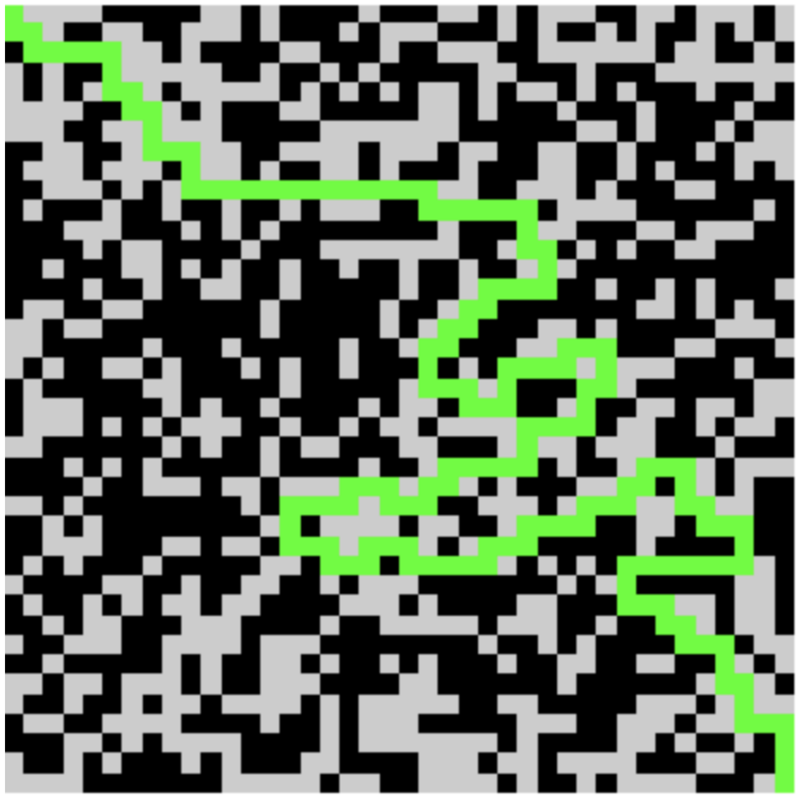
NOTE: For working with A*, we chose to work with maps of smaller dimension ($40X40$) as the running times were high for a higher dimension maps for generating a 'harder' map.

2.3.3 A* with Euclidean Distance as heuristic:

Length of solution path returned:

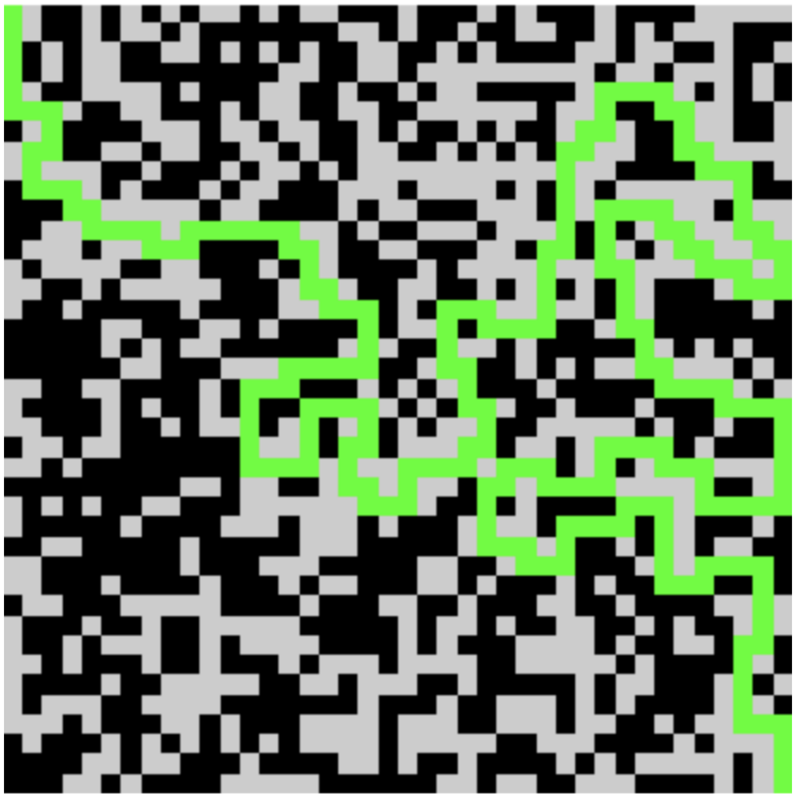


Total number of nodes expanded:



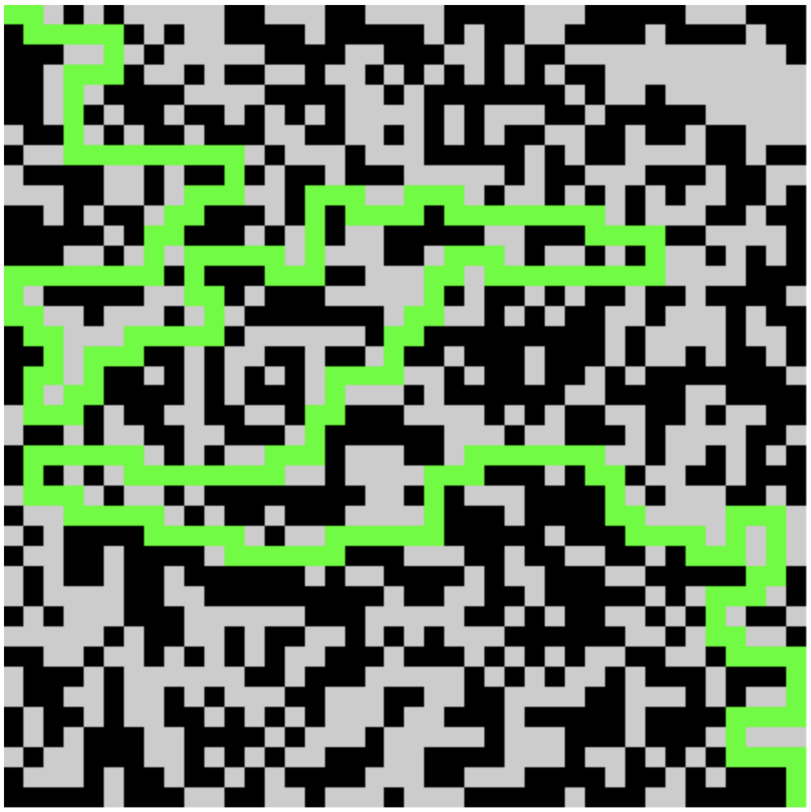
🟩 Solution Path ▨ Blocked Cell ■ Unblocked Cell

Maximum size of fringe during runtime:


🟩 Solution Path ⬜ Blocked Cell ⬛ Unblocked Cell

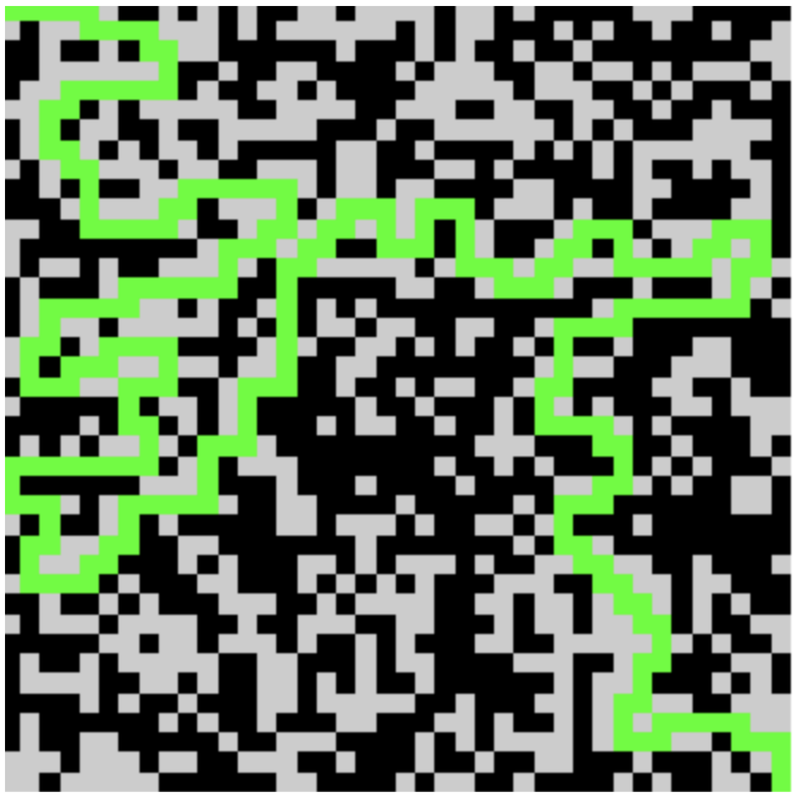
2.3.4 A* with Manhattan Distance as heuristic:

Length of solution path returned:



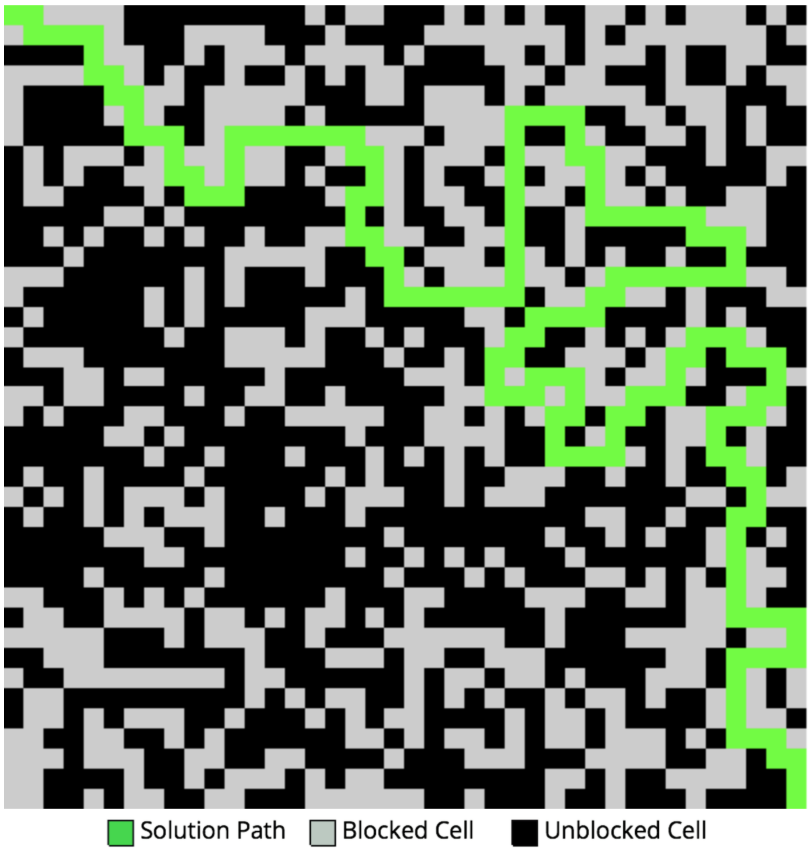
🟩 Solution Path ▨ Blocked Cell ■ Unblocked Cell

Total number of nodes expanded:



🟩 Solution Path ⬜ Blocked Cell ⬛ Unblocked Cell

Maximum size of fringe during runtime:


🟩 Solution Path
▨ Blocked Cell
■ Unblocked Cell