Intro to Artificial Intelligence 16:198:520

Mazerunner: Path Planning and Search Algorithms

Baber Khalid, Muhammad Aunns Shabbir, Jimmy Jorge, Kyungsuk Lee

https://github.com/baber-sos/Maze-Runner

Part 1: Path Planning

1. For each of the implemented algorithms, how large the maps can be (in terms of dim) for the algorithm to return an answer in a reasonable amount of time (less than a minute) for a range of possible p values? Select a size so that running the algorithms multiple times will not be a huge time commitment, but that the maps are large enough to be interesting.

Dimension	**500**	**1000**	**2000**	**500**	**1000**	**2000**	**500**	**1000**	**2000**	**500**	**1000**	**2000**
DFS	>1min	>1min	>1min	>1min	>1min	>1min	>1min	>1min	>1min	Not solvable		
BFS	5.0s	28s	>1min	3.58s	32.0s	>1min	4.2s	26.1s	>1min			
Manhattan A*	1.58s	6.0s	33.0s	1.23s	9.15s	18.9s	1.02s	6.56s	5.57s			
Euclidean A*	2.08s	6.0s	33.0s	1.32s	14.2s	25.9s	1.07s	6.44s	26.4s			
Probability	**p = 0.1**			**p = 0.2**			**p = 0.3**			**p = 0.4**		

2. Find a random map with p ≈ 0.2 that has a path from corner to corner. Show the paths returned for each algorithm. (Showing maps as ASCII printouts with paths indicated is sufficient; however 20 bonus points are available for coding good visualizations.)

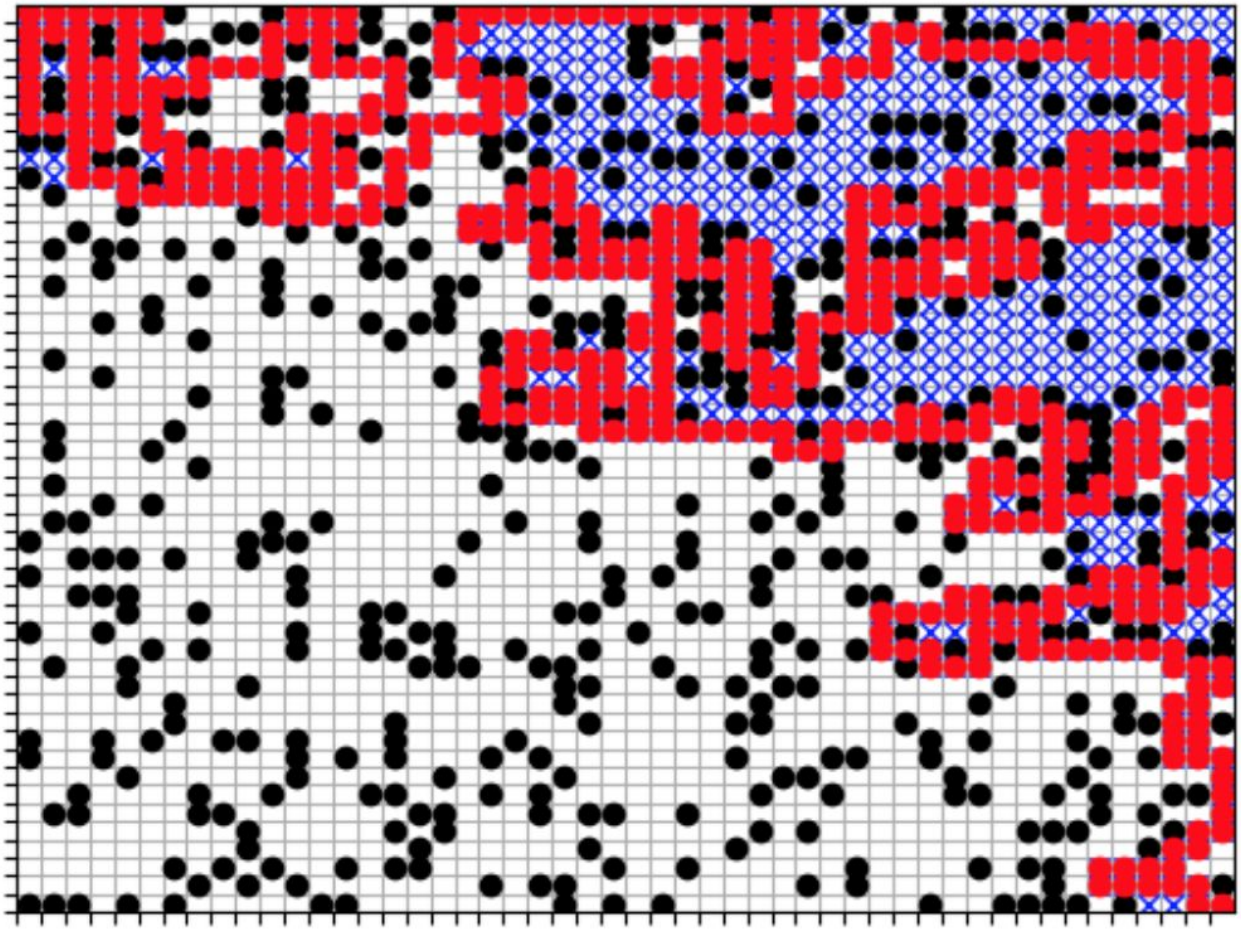
Following are the paths returned for each algorithm with p = 0.2 of being an obstacle:

Note:

- White indicates open space
- Red indicates solution path taken from start to the goal
- Blue indicates node expanded

DFS:

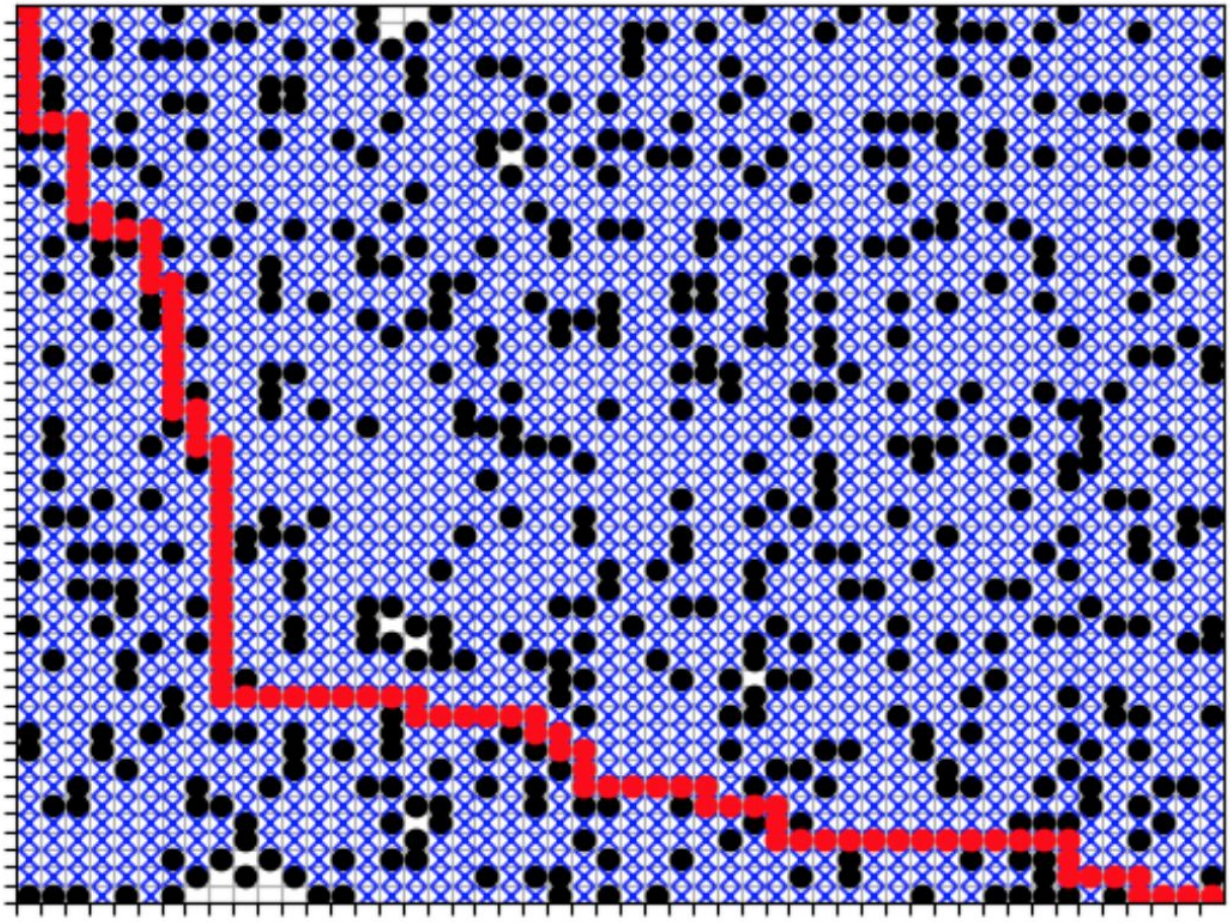
DFS returns the path as soon as it reaches the goal through any node, hence the reason the path looks very windy and spread out until it hits the end point.



50x50 Maze solved with DFS

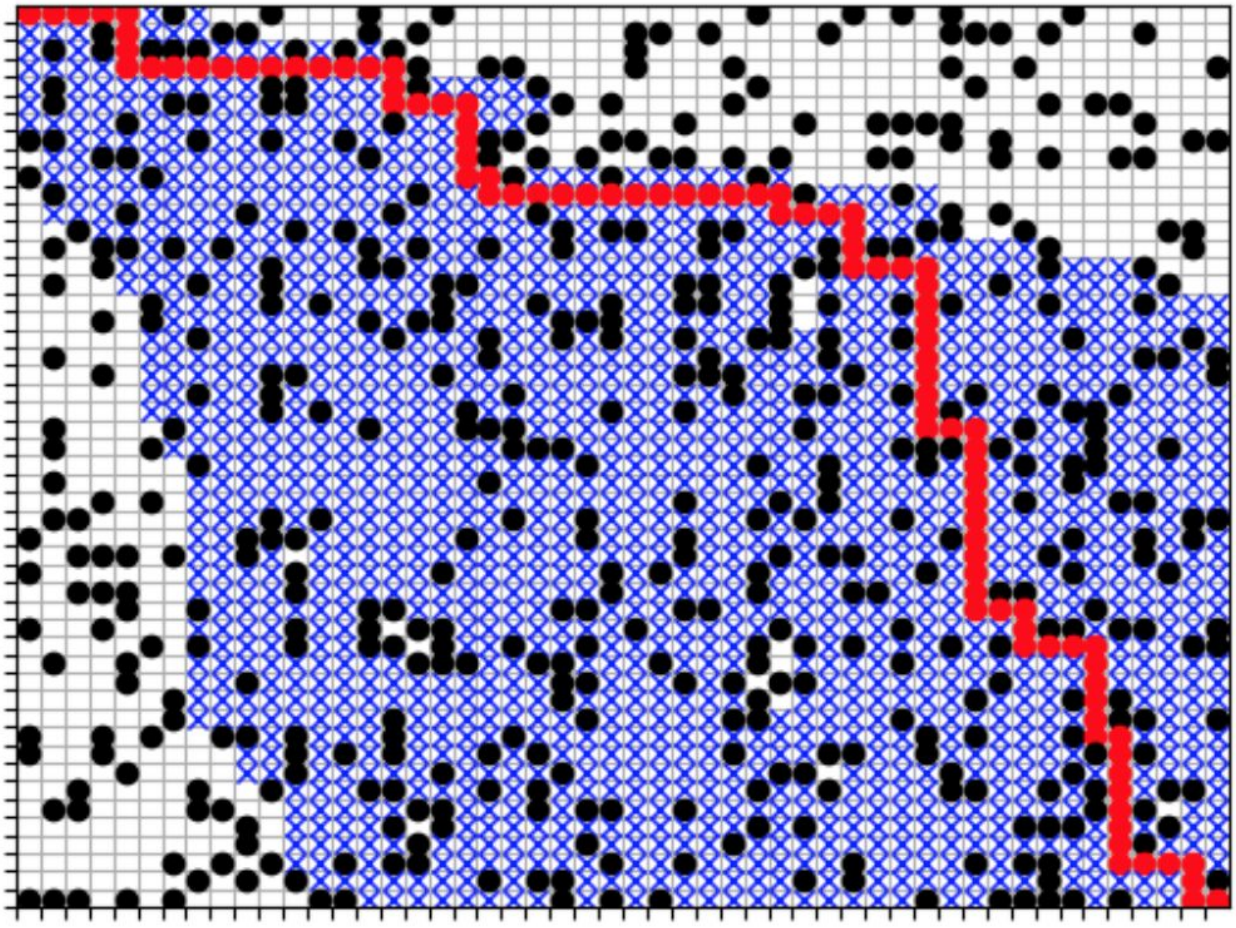
BFS:

It is evident from the path taken by BFS algorithm to reach goal that it explores the maximum number of nodes.



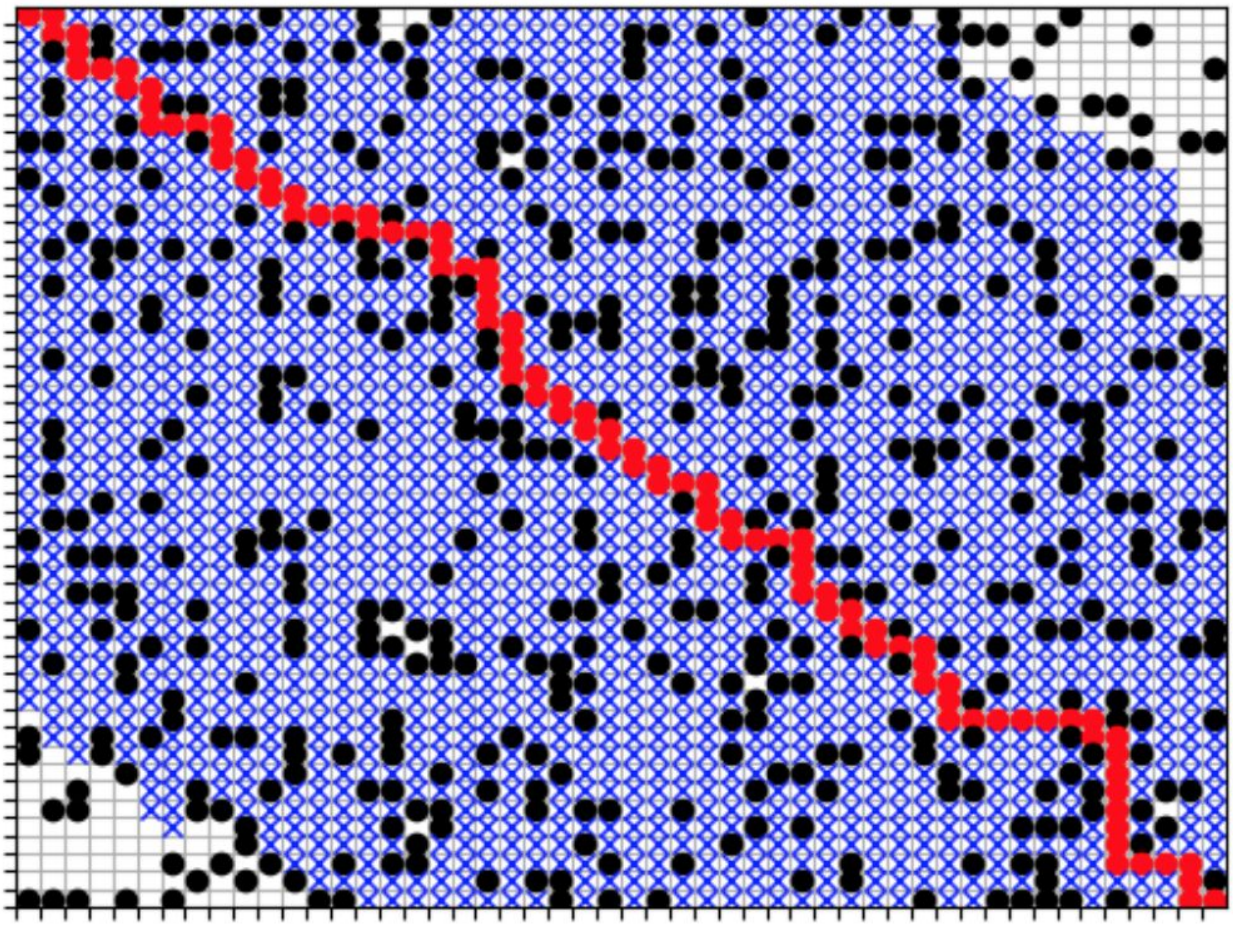
50x50 Maze solved with BFS

Manhattan A*:



50x50 Maze solved with A (Manhattan heuristic)*

Euclidean A*:



50x50 Maze solved with A (Euclidean heuristic)*

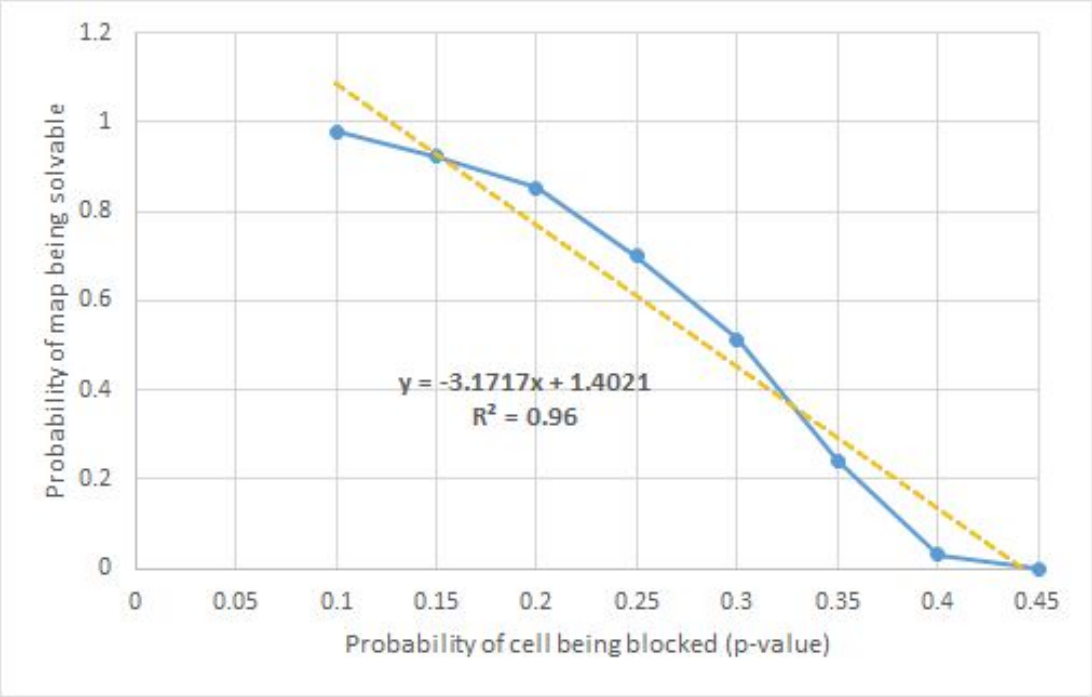
3. For a fixed value of dim as determined in Question (1), for each p = 0.1, 0.2, 0.3, . . . , 0.9, generate a number of maps and try to find a path from start to goal - estimate the probability that a random map has a complete path from start to goal, for each value of p. Plot your data. Note that for p close to 0, the map is nearly empty and the path is clear; for p close to 1, the map is mostly filled and there is no clear path. There is some threshold value p_0 so that for $p < p_0$, there is usually a clear path, and $p > p_0$ there is no path. Estimate p_0. Which path finding algorithm is most useful here, and why?

The path finding algorithm that's the most useful in this situation would be A* because it expands the least number of nodes to be cost efficient. Therefore, to see if a solution to a maze exists, we will run A*. The probability that each map has a complete path from start to goal can be computed as follows:

$$Probability\ of\ finding\ a\ path\ =\ \frac{Number\ of\ maps\ that\ were\ solved}{Total\ number\ of\ maps\ generated}$$

The dimension of every map generated is 100x100 with 1000 trials to find average rates.

Probability of cell being blocked	Number of maps that were solved	Total number of maps generated	Success rate
0.1	978	1000	97.8%
0.15	923	1000	92.3%
0.2	853	1000	85.3%
0.25	699	1000	69.9%
0.3	512	1000	51.2%
0.35	242	1000	24.2%
0.4	32	1000	3.2%
0.45	0	1000	0.0%



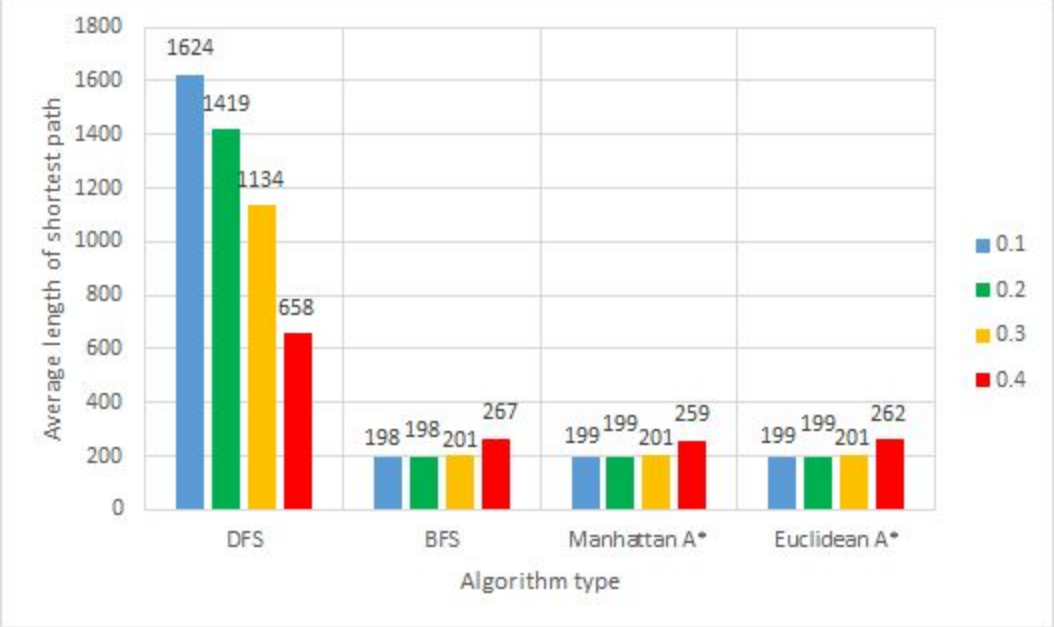
After plotting the probability that a random map has a complete path from start to goal for each value of p (probability that a cell is blocked), we implemented a line of regression with the data. With this, we were able to figure out the value of p_0 to be ~0.44. As seen with our tests of 1000 trials for each p-value, at p = 0.4, very few maps were able to be solved, but there were still a few. At p = 0.45, a total of zero were solvable; hence our solved value of p_0 seems to be very accurate.

4. For a range of p values (up to p_0), generate a number of maps and estimate the average or expected length of the shortest path from start to goal. You may discard all maps where no path exists. Plot your data. What path finding algorithm is most useful here?

Average length of the shortest path from start to goal was calculated with the average of 1000 trials of maze generations. Dim = 100.

Path length				
DFS	1624	1419	1134	658
BFS	198	198	201	267
Manhattan A*	199	199	201	259
Euclidean A*	199	199	201	262
Probability	**p = 0.1**	**p = 0.2**	**p = 0.3**	**p = 0.4**

Both A* algorithms and BFS ended up being very similar in average lengths of the shortest path from start to goal. For p = 0.1, 0.2, and 0.3, BFS was more efficient in finding the shortest path, by a slight margin. The A* algorithms performed well overall, but in the case of finding the shortest path, the BFS algorithm is the most useful prior to p approaching p_0. With the information gathered in the previous questions, it was evident that when p > 0.4, the chance of the maze being solvable was very small. Thus, we kept 0.4 as the cut off.

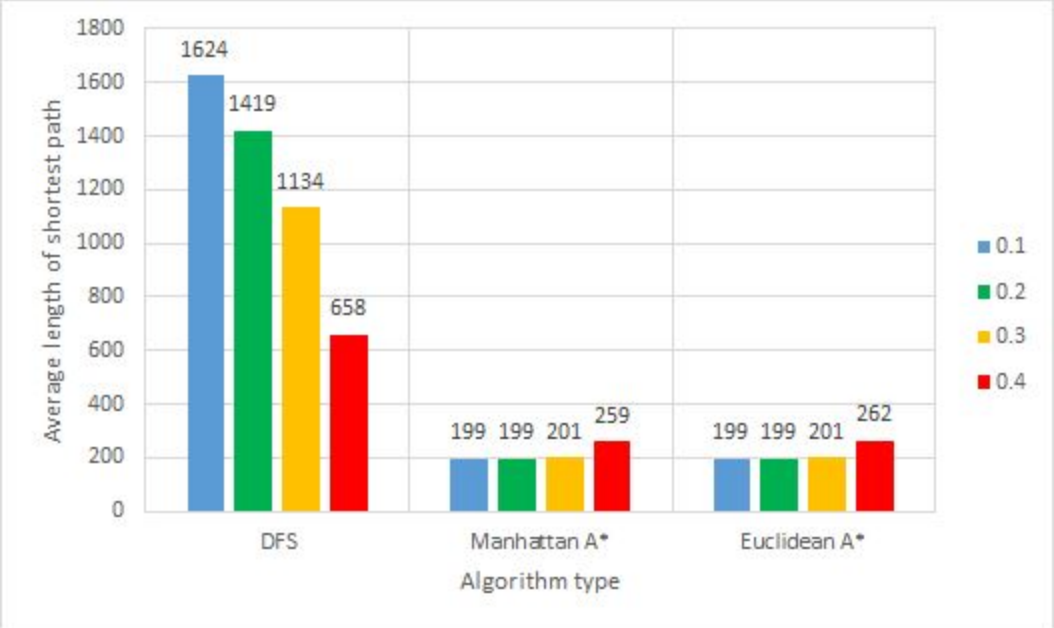


5. For a range of p values (up to p_0), estimate the average length of the path generated by A* from start to goal (for either heuristic). Similarly, for the same p values, estimate the average length of the path generated by DFS from start to goal. How do they compare? Plot your data.

Average length of the shortest path from start to goal was calculated with the average of 1000 trials of maze generations. Dim = 100.

Average Path Length				
DFS	1624	1419	1134	658
Manhattan A*	199	199	201	259
Euclidean A*	199	199	201	262
Probability	**p = 0.1**	**p = 0.2**	**p = 0.3**	**p = 0.4**

It is clear that A*, with either Manhattan or Euclidean heuristics, outperforms DFS in finding the shortest path from start to goal for all values of p. This is because A* can be made to be equivalent to other optimal search algorithms, such as BFS for shortest path. It is worth noting that as p increases, the path length of DFS decreases while the path length of A* increases. This is because as more cells become obstacles, DFS has less nodes to traverse. However for A*, with more cells being blocked, the shortest possible path being a perfect diagonal becomes more improbable.

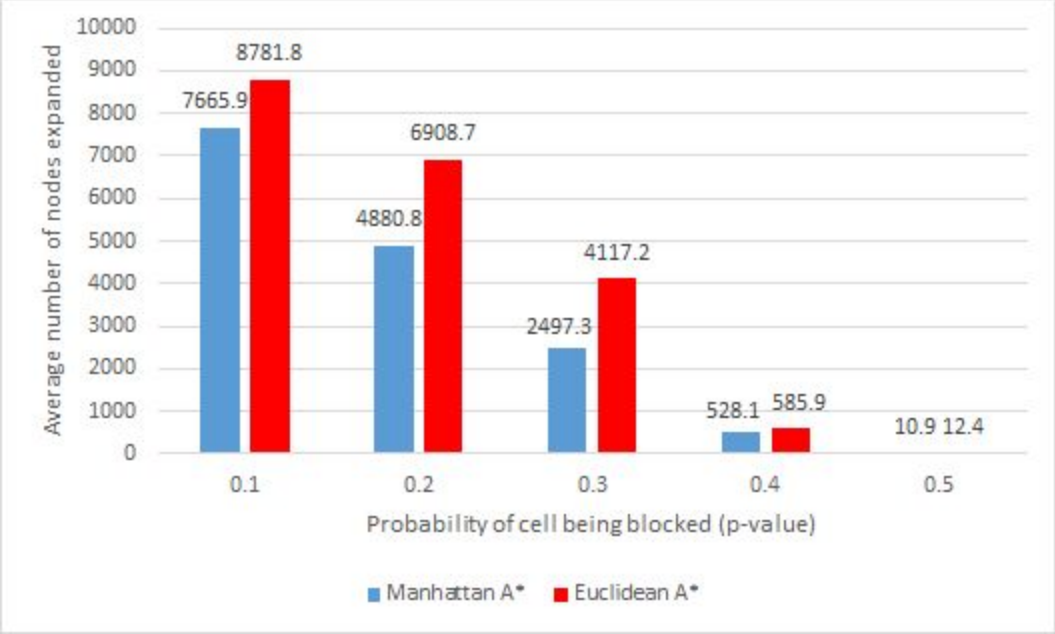


6. For a range of p values (up to p_0), estimate the average number of nodes expanded in total for a random map, for A* using the Euclidean Distance as the heuristic, and using the Manhattan Distance as the heuristic. Plot your data. Which heuristic typically expands fewer nodes? Why? What about for p values above p_0?

Average number of nodes expanded was calculated with the average of 1000 trials of maze generations. Dim = 100.

Average Nodes Expanded					
Manhattan A*	7665.9	4880.8	2497.3	528.1	10.9
Euclidean A*	8781.8	6908.7	4117.2	585.9	12.4
Probability	**p = 0.1**	**p = 0.2**	**p = 0.3**	**p = 0.4**	**p = 0.5**

Based on the data recorded, we see that using the Manhattan heuristic expands fewer nodes across all values of p including $p > p_0$. Both values of the nodes expanded seem to decrease towards a similar value for both A* algorithms as the p approaches p_0. In the Euclidean A*, the nodes expanded tend to explore more of the top and bottom "halves" as the diagonal from the start to goal is being formed. In the Manhattan A*, the nodes explored stay along the diagonal that is formed from the start to goal nodes more efficiently, thus resulting in a lot less nodes expanded in the end.

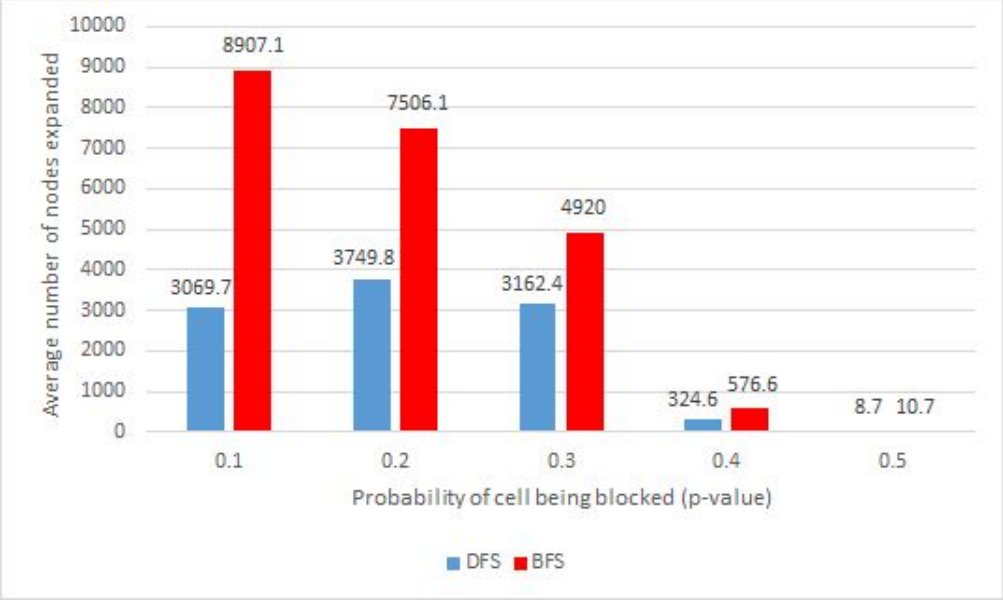


7. For a range of p values (up to p_0), estimate the average number of nodes expanded in total for a random map by DFS and by BFS. Plot your data. Which algorithm typically expands fewer nodes? Why? How does either algorithm compare with A∗ in Question (6)?

Average number of nodes expanded was calculated with the average of 1000 trials of maze generations. Dim = 100.

Average Nodes Expanded					
DFS	3069.7	3749.8	3162.4	324.6	8.7
BFS	8907.1	7506.1	4920.0	576.6	10.7
Probability	**p = 0.1**	**p = 0.2**	**p = 0.3**	**p = 0.4**	**p = 0.5**

Based on the data recorded, we see that DFS expands fewer nodes than BFS across all values of p, including $p > p_0$. This is because BFS will expand all neighbor nodes of the current node and then expand the neighbor nodes of those neighbors, and the process is repeated until no node is available. This leads to an exponential increase of the nodes expanded until the goal node is reached. Conversely, as DFS expands the nodes, it is sequentially moving closer and closer to the goal one path at a time.
In comparison to the A* algorithms, DFS expands less nodes than both A* algorithms, while BFS expands more nodes than both A* algorithms.



Bonus 1. Why were you not asked to implement UFCS?

The uniform cost search algorithm takes into account the cost of expanding a node and prioritizes cumulative cost. In the case of our maze where the edge from cell to cell is is by default equal everywhere, there is no priority given to the cost. Therefore, UFCS would be the same as an implementation of breadth first search.

Part 2: Building Hard Mazes

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

We chose genetic algorithms as the algorithm of choice to find the hardest maze. Our maze is being represented as a 2D grid of zeros and ones where a zero means an open space and a one means an obstacle. The process starts with finding the two solvable random mazes which act as the starting generation. Our genetic algorithm employs a single point crossover technique where it takes a range of rows from a point until the last row in both mazes, swaps them and generates two new children which may be more difficult to solve by a given algorithm. The independent probability of mutation for each child is 0.5 (after testing with different ones) and as for the mutation process, we take a range of cells randomly in the maze and invert them. The maximum number of cells which can be mutated is 50% of the total number of cells in the maze. For a generation to be eligible to act as the parents for the new generation, one of them must have a higher difficulty than both of its parents. The reason we did not enforce this condition on both of the children was that we thought if a child had less of a difficulty it may have less obstacles on the grid itself because of some reason like mutation. That may help the algorithm in filling out tiny spaces or relax some difficult portions of the maze and eventually reach a harder but solvable maze. One other option was the problem of children being unsolvable. For now, we have enforced that both the children should be solvable for the generation to carry on, although this can be changed to replace the less suitable parent with the child and carry on with the process. The reason we did not do this was because of the flexibility condition stated earlier. Therefore, we thought it might be better not to decrease the cost difference between parents to deal with the solvability problem in new generations.

9. Unlike the problem of solving a maze, for which the 'goal' is well-defined, it is difficult to know when we have constructed the 'hardest' maze. That being so, what kind of termination conditions can you apply to your search algorithm to generate 'hard' if not the 'hardest' mazes? What kind of shortcomings or advantages do you anticipate your approach having?

Since a genetic algorithm relies on its parents to generate new children which may be better than the parents itself, the termination condition of the algorithm is the number of tries in which parents fail to produce offspring better than themselves. We can increase or decrease this parameter in the algorithm to generate mazes which are close to hardest or just a little harder than where they started from. Since the process of generating children depends on the chance and the structure of the parents itself (which is random too), we may get lucky and get a maze close to hardest in few tries or we may get unlucky and never get close to the hardest maze. We can increase mutation rate to introduce more uncertainty in hopes that children will be better than the parents but if its too high we might end up generating a lot of unsolvable mazes. However, we believe that with the proper tweaking of the randomness parameters it should be relatively easier to obtain close to the hardest mazes in few tries. This algorithm could also be modified easily to increase the generation size at any given point, using something like a multi-point crossover and choose the best among them to carry on to the next generation.

10. For each of the following algorithms, do the following: Using your local search algorithm, for each of the following properties indicated, generate and present three mazes that attempt to maximize the indicated property. Do you see any patterns or trends? How can you account for them? What can you hypothesize about the 'hardest' maze, and how close do you think you got to it?

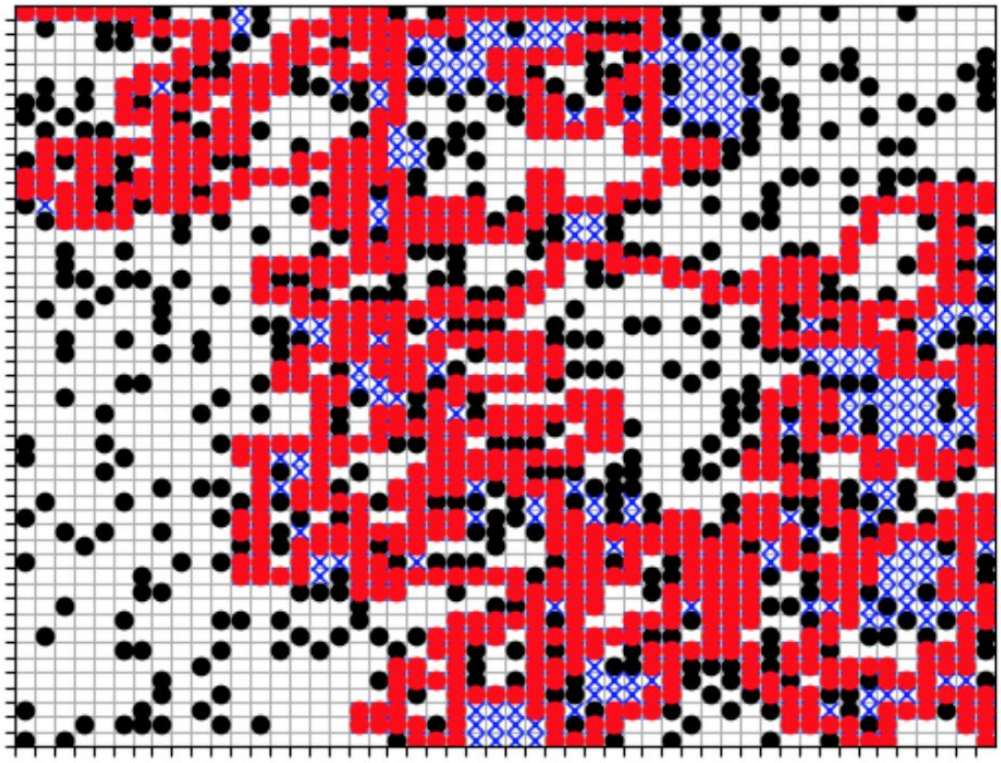
Every result produced is from a randomly generated maze (parents) of DIM = 50. Our algorithm generates different shaped children depending on the cost type, so it inherently performs well in each kind of situation presented. In each case, the search algorithms seem to be converging to a single value evident from the relatively close data of the hardest costs. The notion of the hardest maze is different depending on the cost we are trying to maximize. Our algorithm does not reach the maximum state itself as can be seen from the multiple runs, but the final costs of the generated children are close enough to conclude that it is reaching a point which is can be considered the hardest.

Parent 1 and parent 2 for each algorithm represents the solutions of the corresponding algorithm. From these, harder children are created by maximizing "hardness" parameters (number of nodes expanded, maximum length of solution, and maximum size of fringe). The children's costs are then calculated and plotted, and after looking at them, we can definitely see the increase in hardness.

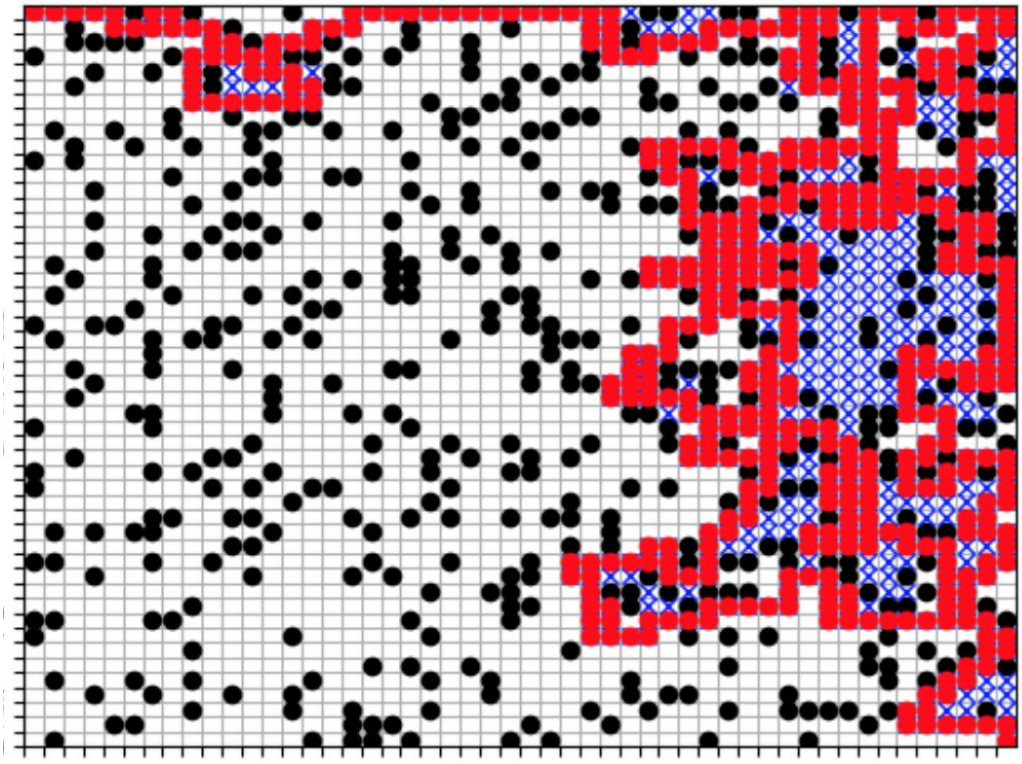
a) DFS

DFS	Total number of nodes expanded	Length of solution path returned	Maximum size of fringe during runtime
Parent maze cost	**993**	**800**	**1324**
Hard child #1	**1872**	**800**	**1645**
Hard child #2	**1830**	**862**	**1664**
Hard child #3	**1922**	**868**	**1677**

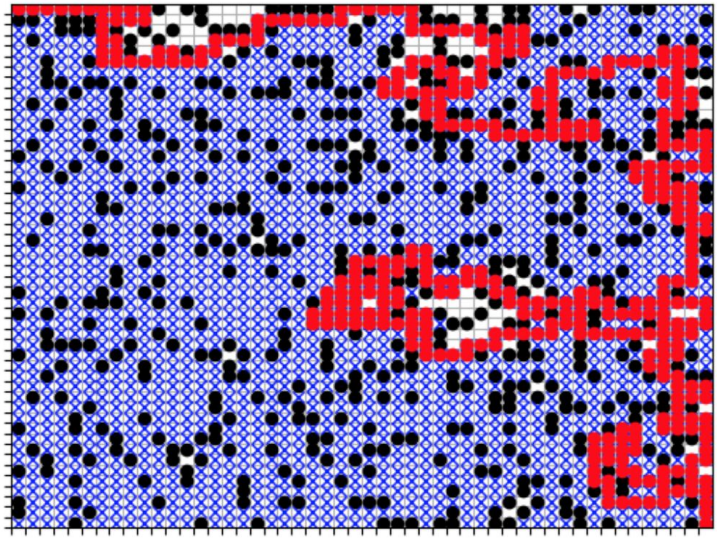
Below parent #1 and parent #2 are plotted. On each subsequent page, the 3 children for max number of nodes expanded, max length of solution path, and max size of fringe are all also plotted, respectively.



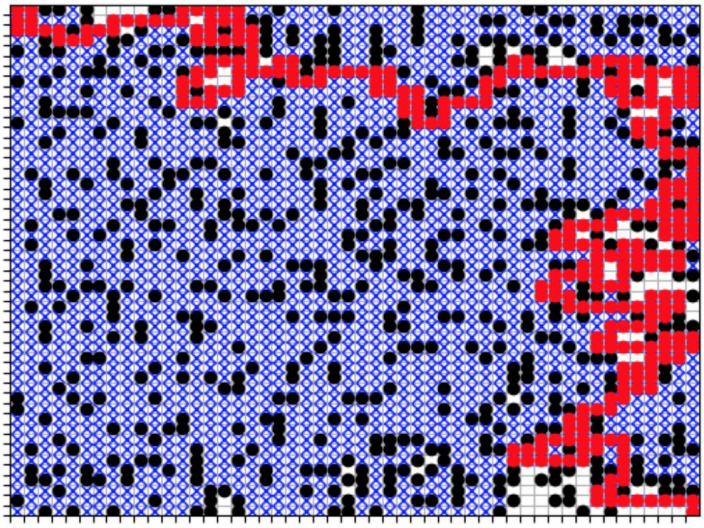
Parent 1



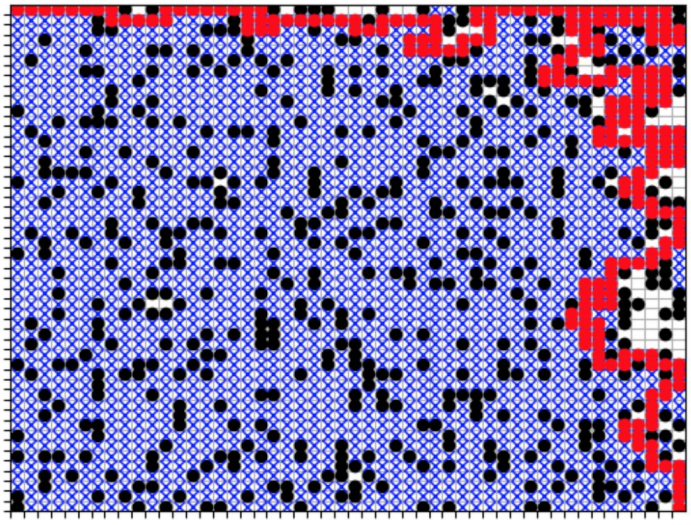
Parent 2



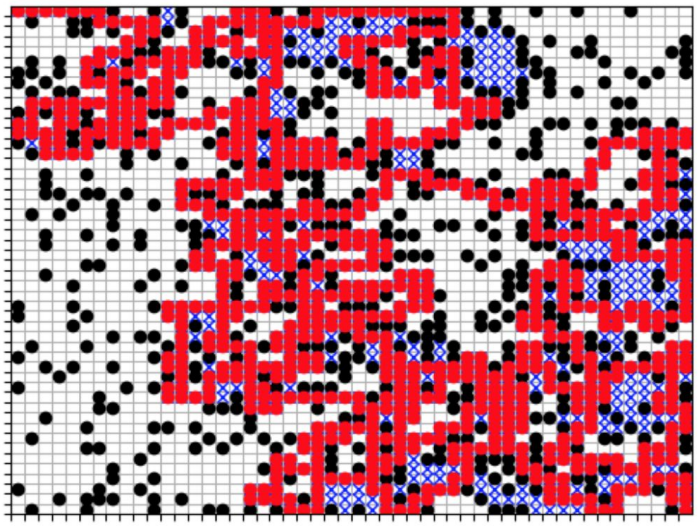
Hard child #1 with max nodes



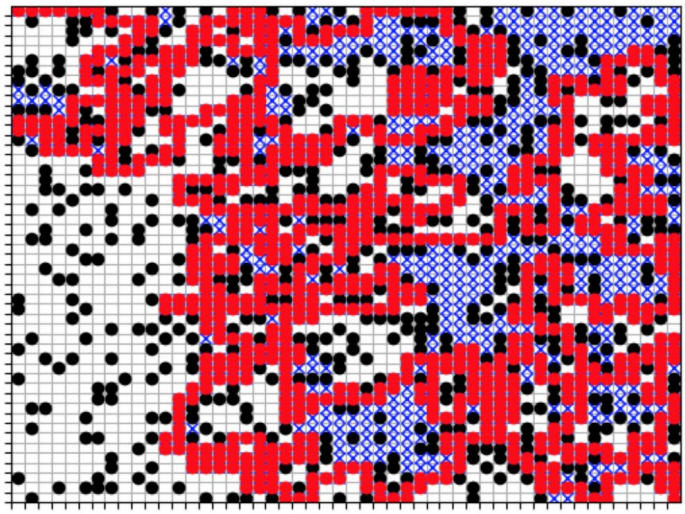
Hard child #2 with max nodes



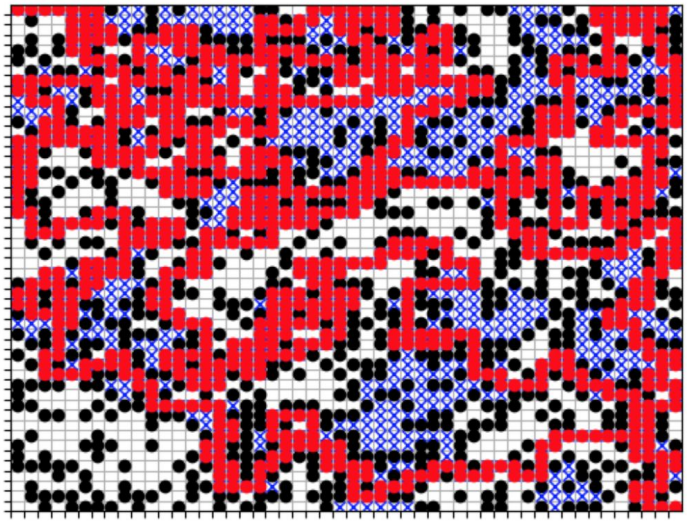
Hard child #3 with max nodes



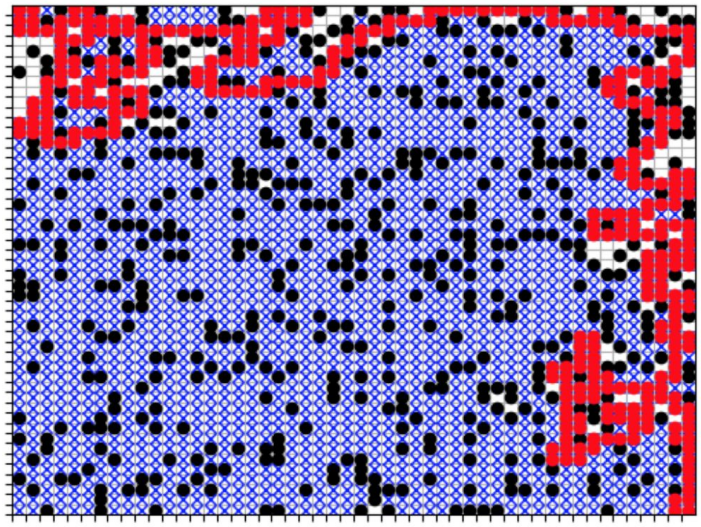
Hard child #1 with max length



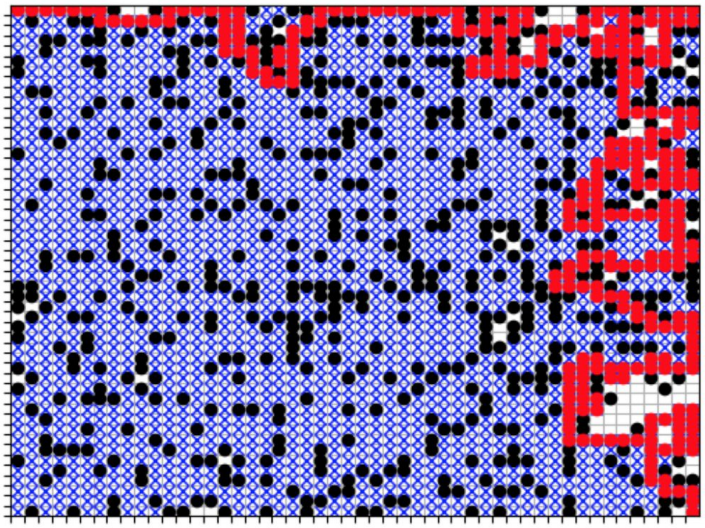
Hard child #2 with max length



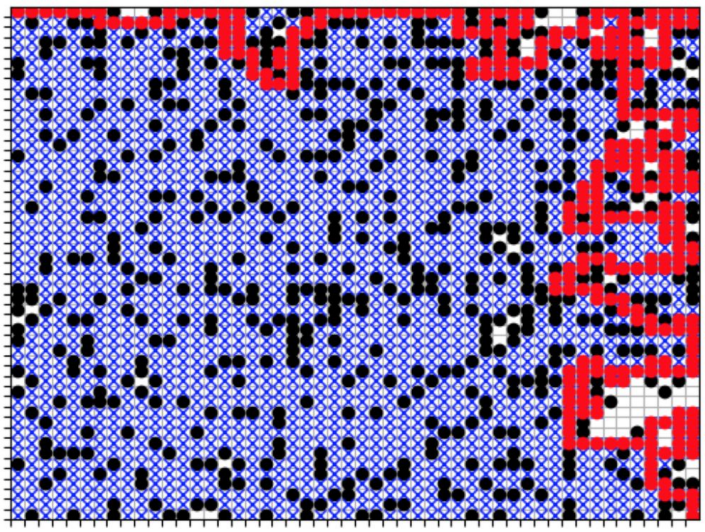
Hard child #3 with max length



Hard child #1 with max fringe



Hard child #2 with max fringe

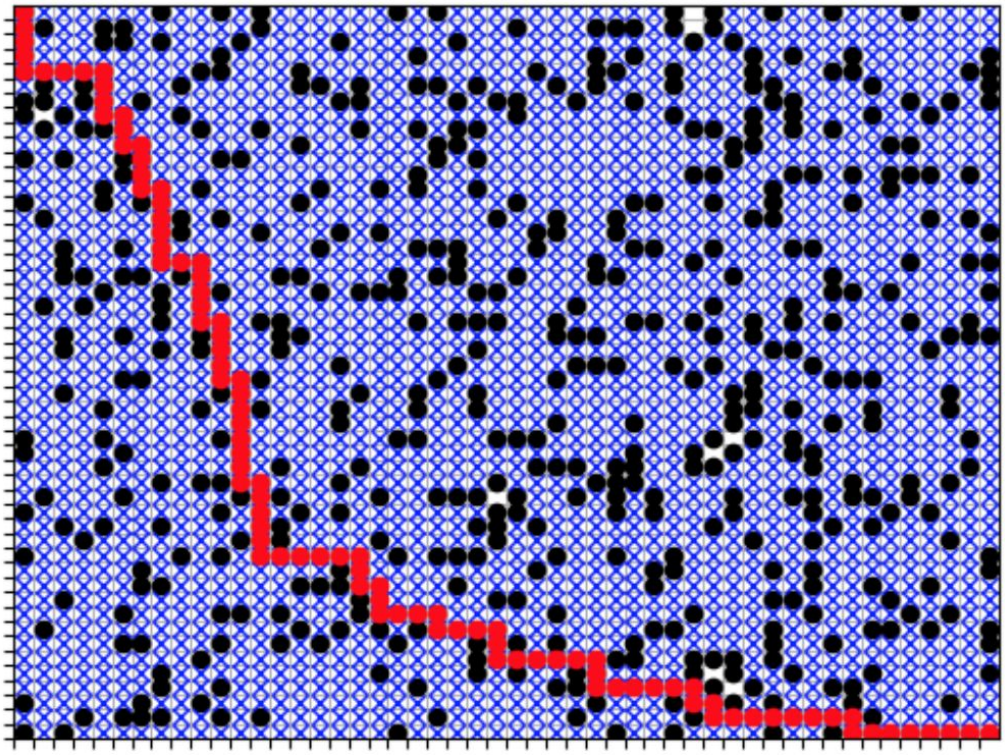


Hard child #3 with max fringe

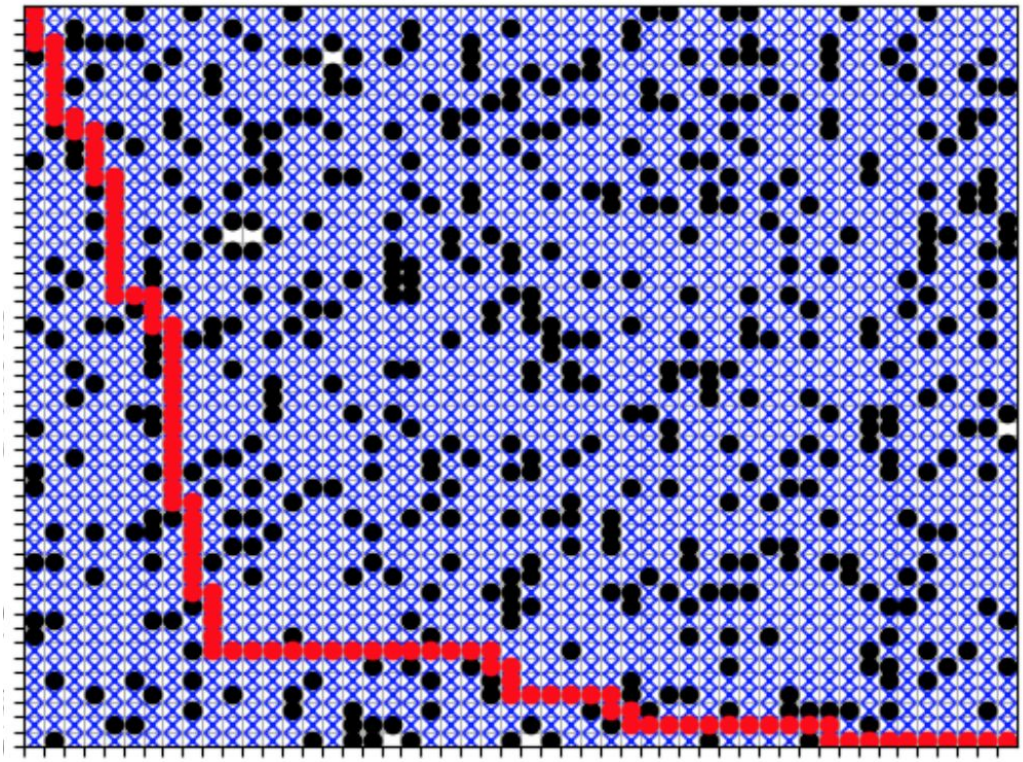
b) BFS

BFS	Total number of nodes expanded	Length of solution path returned	Maximum size of fringe during runtime
Parent maze cost	**2003**	**98**	**162**
Hard child #1	**2017**	**158**	**200**
Hard child #2	**2020**	**186**	**225**
Hard child #3	**2014**	**164**	**213**

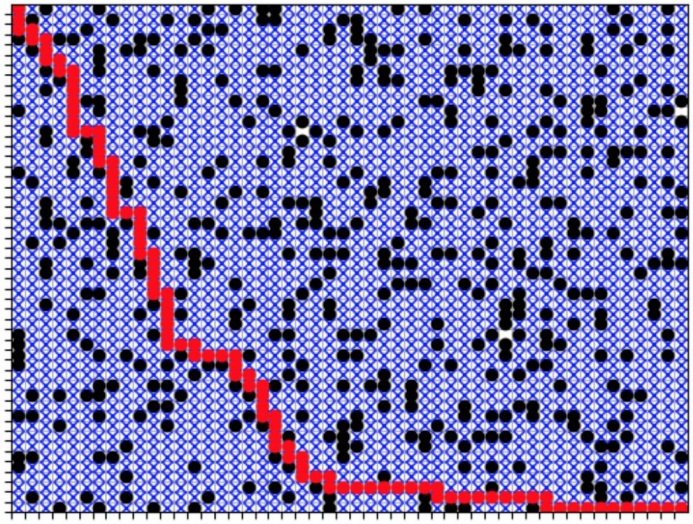
Below parent #1 and parent #2 are plotted. On each subsequent page, the 3 children for max number of nodes expanded, max length of solution path, and max size of fringe are all also plotted, respectively.



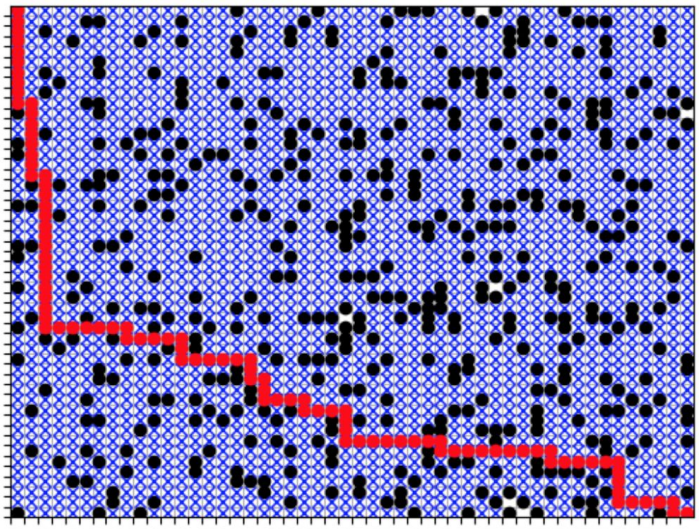
Parent #1



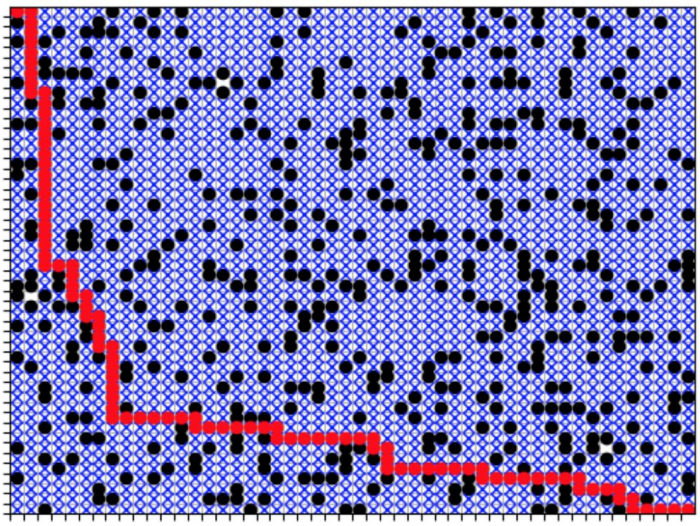
Parent #2



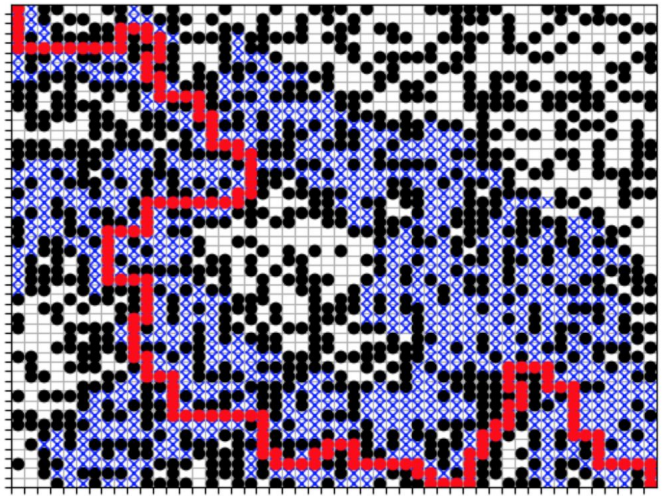
Hard child #1 with max nodes



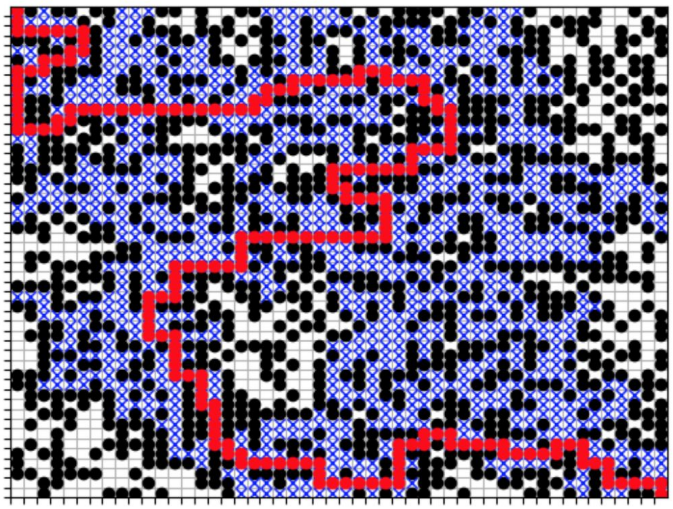
Hard child #2 with max nodes



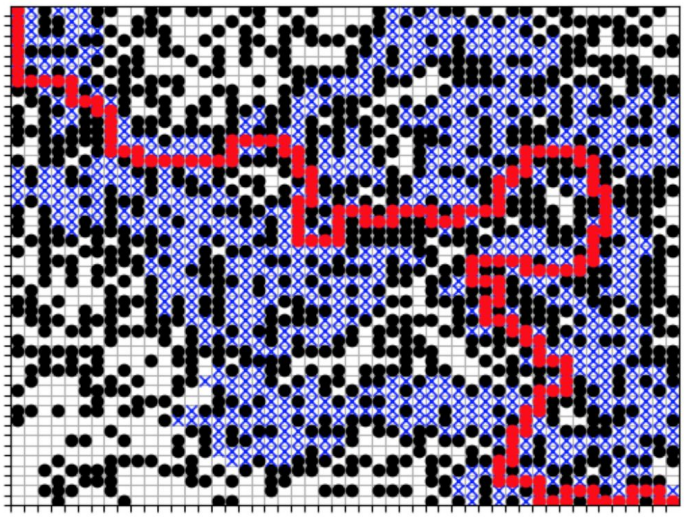
Hard child #3 with max nodes



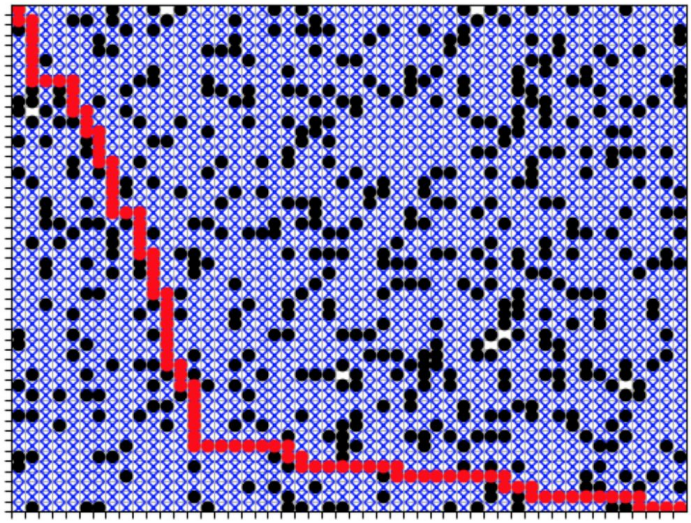
Hard child #1 with max length



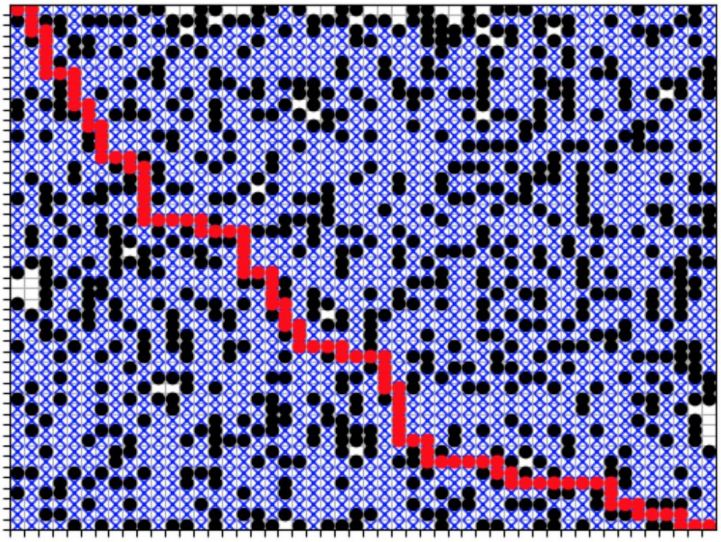
Hard child #2 with max length



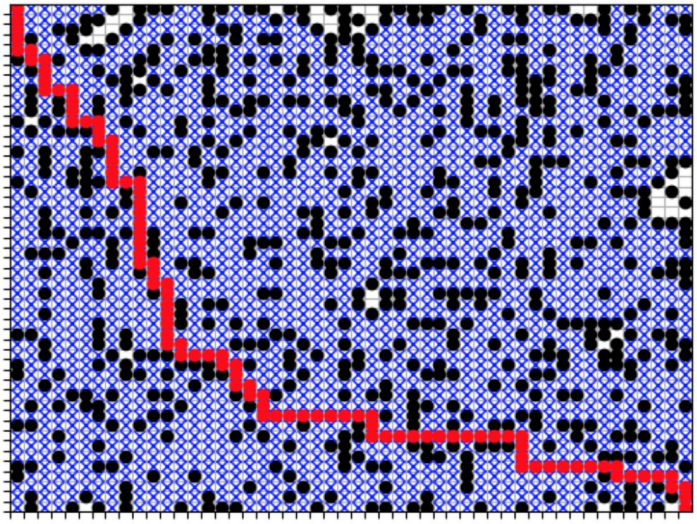
Hard child #3 with max length



Hard child #2 with max fringe



Hard child #2 with max fringe

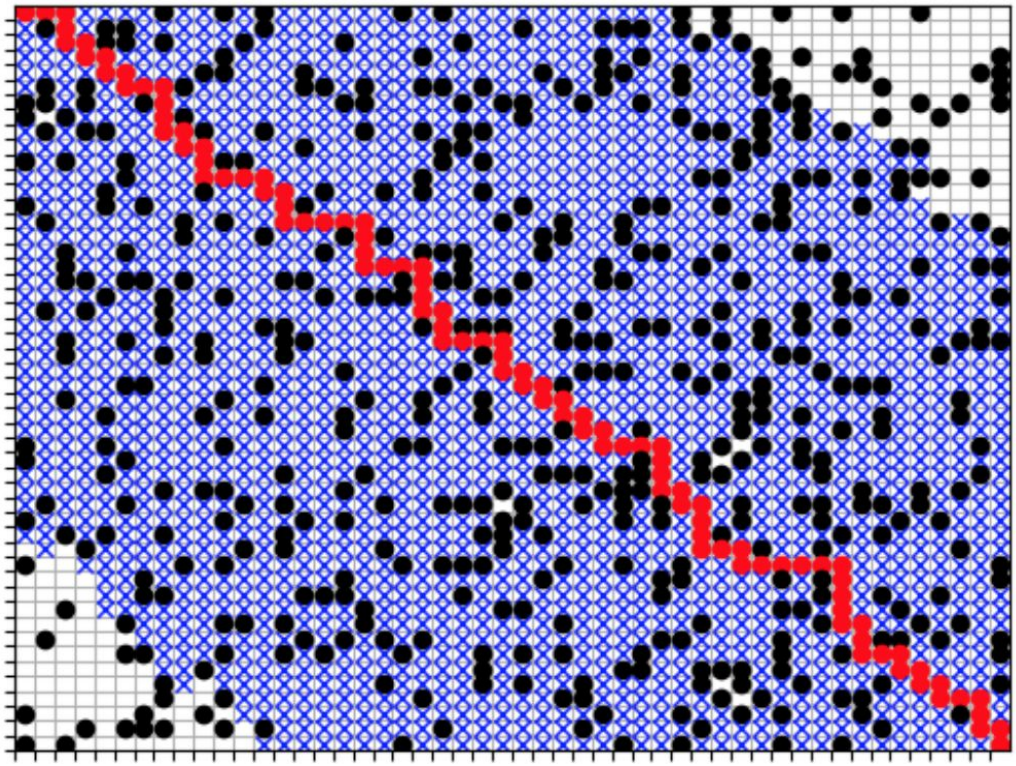


Hard child #3 with max fringe

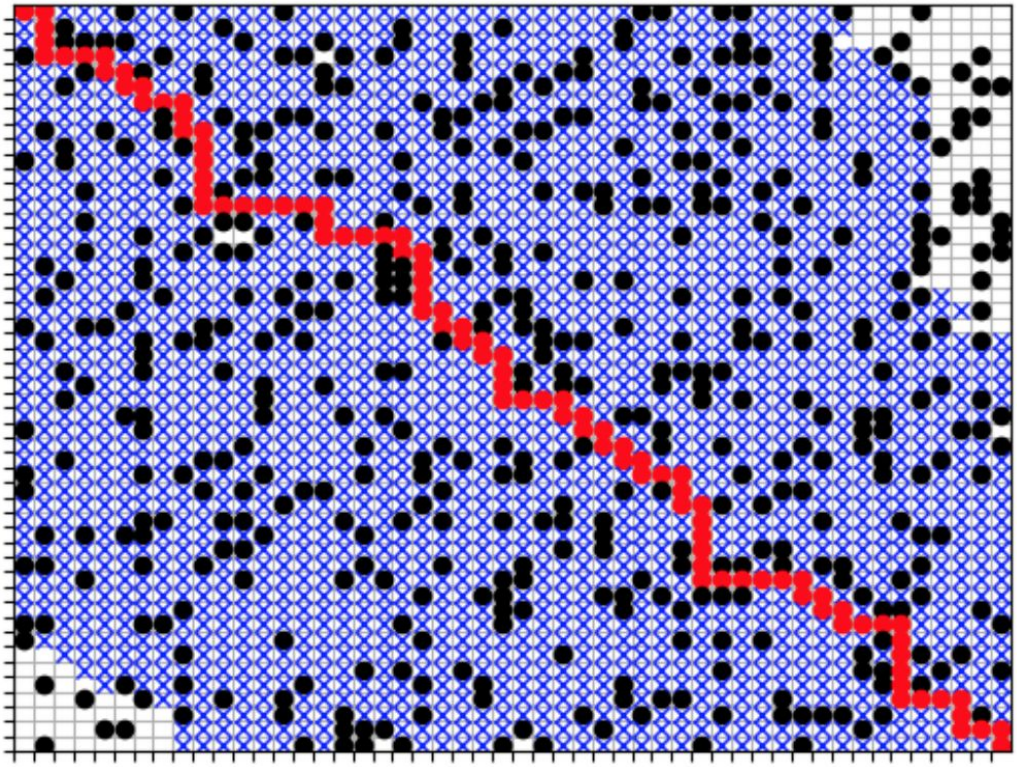
c) A* with Euclidean Distance Heuristic

A* (Euclidean Heuristic)	Total number of nodes expanded	Length of solution path returned	Maximum size of fringe during runtime
Parent maze cost	**1855**	**99**	**83**
Hard child #1	**1929**	**177**	**108**
Hard child #2	**1919**	**155**	**109**
Hard child #3	**1932**	**163**	**119**

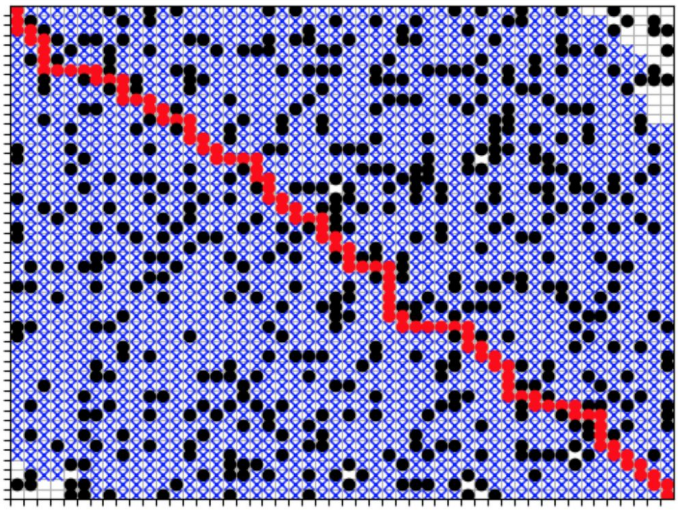
Below parent #1 and parent #2 are plotted. On each subsequent page, the 3 children for max number of nodes expanded, max length of solution path, and max size of fringe are all also plotted, respectively.



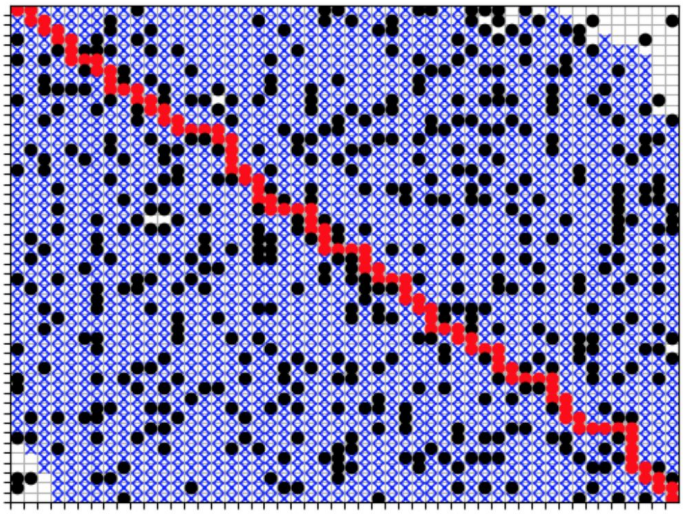
Parent #1



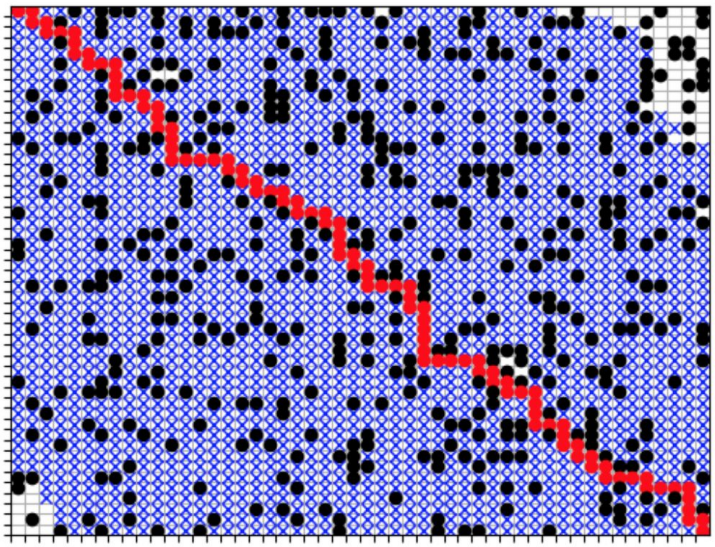
Parent #2



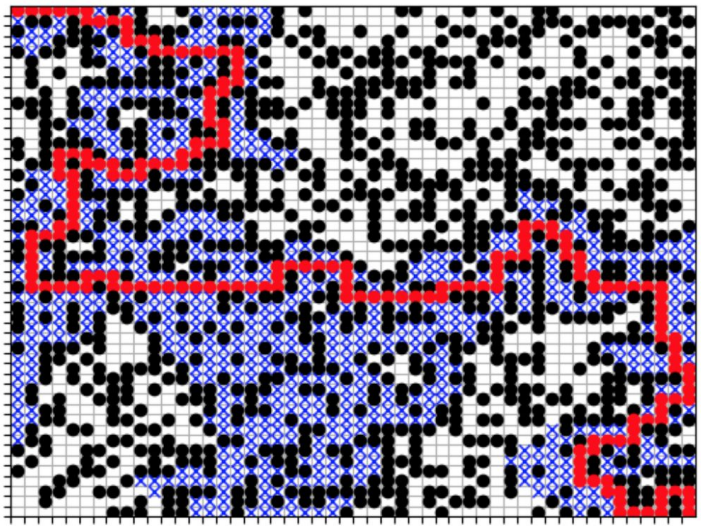
Hard child #1 with max number of nodes



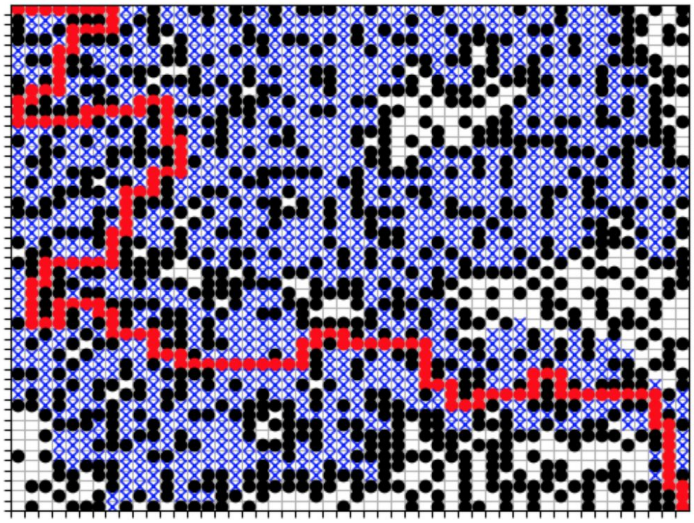
Hard child #2 with max number of nodes



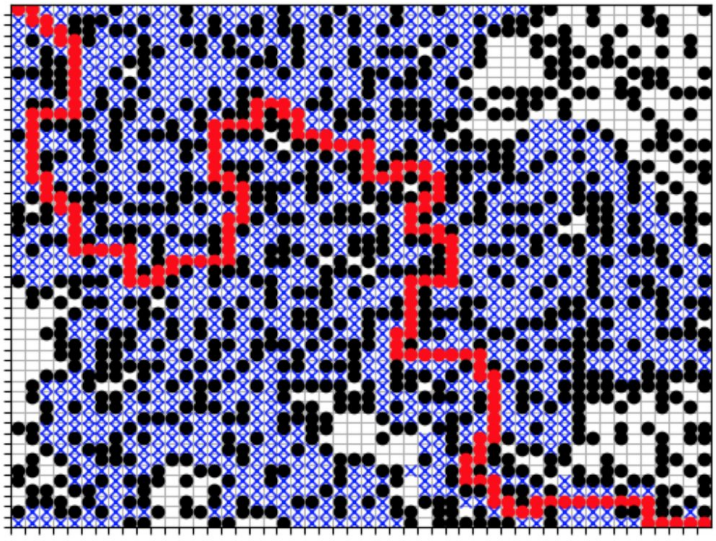
Hard child #3 with max number of nodes



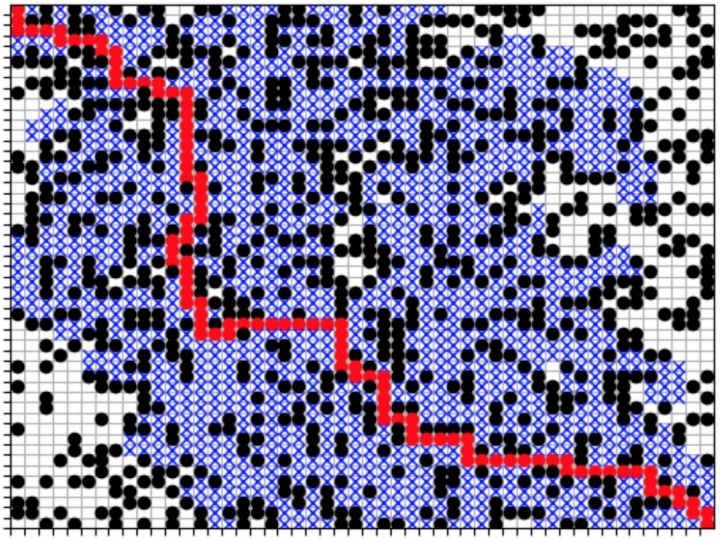
Hard child #1 with max length



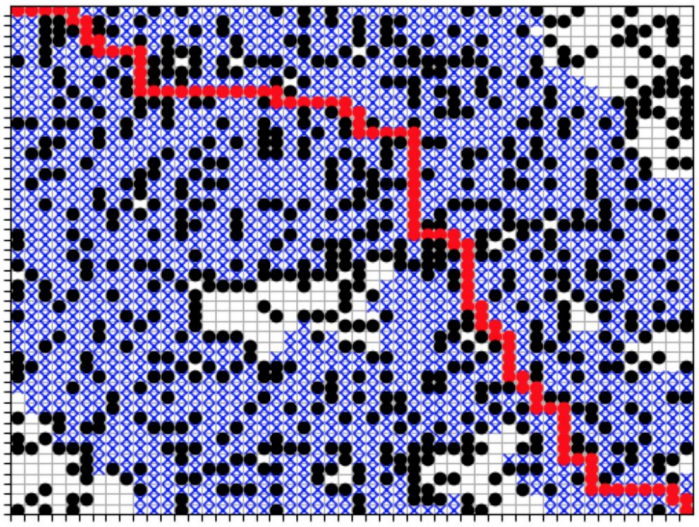
Hard child #2 with max length



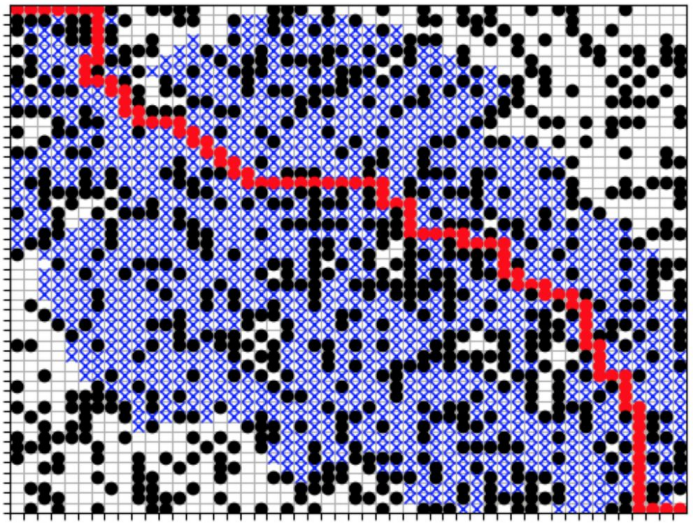
Hard child #3 with max length



Hard child #1 with max fringe



Hard child #2 with max fringe

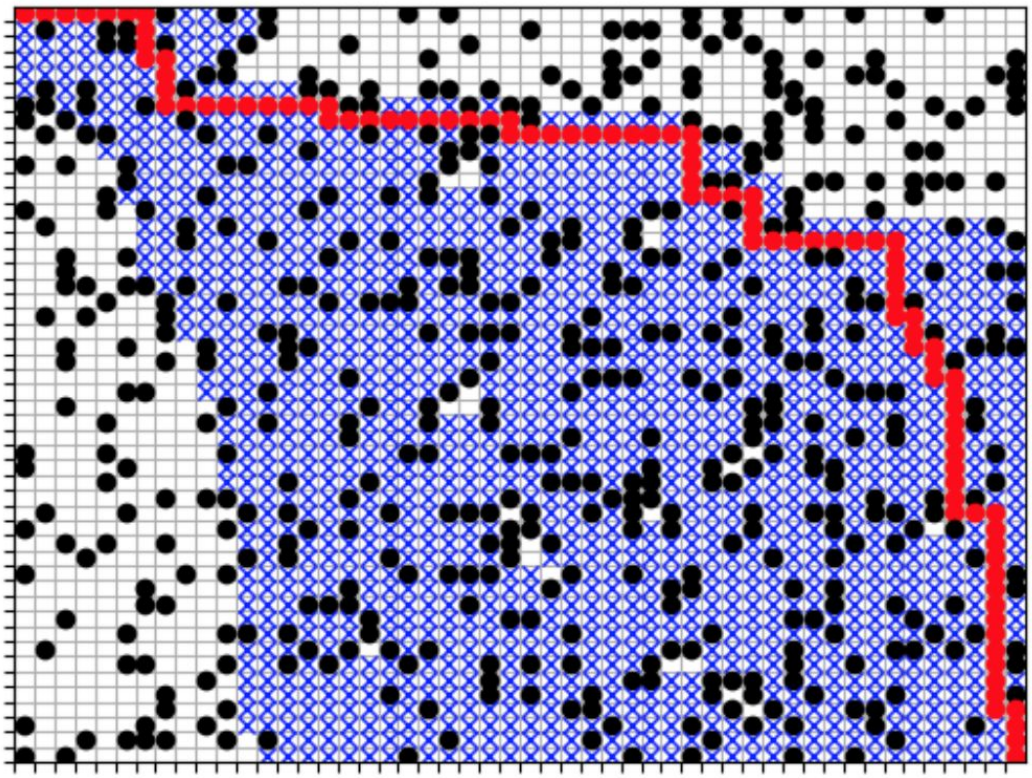


Hard child #3 with max fringe

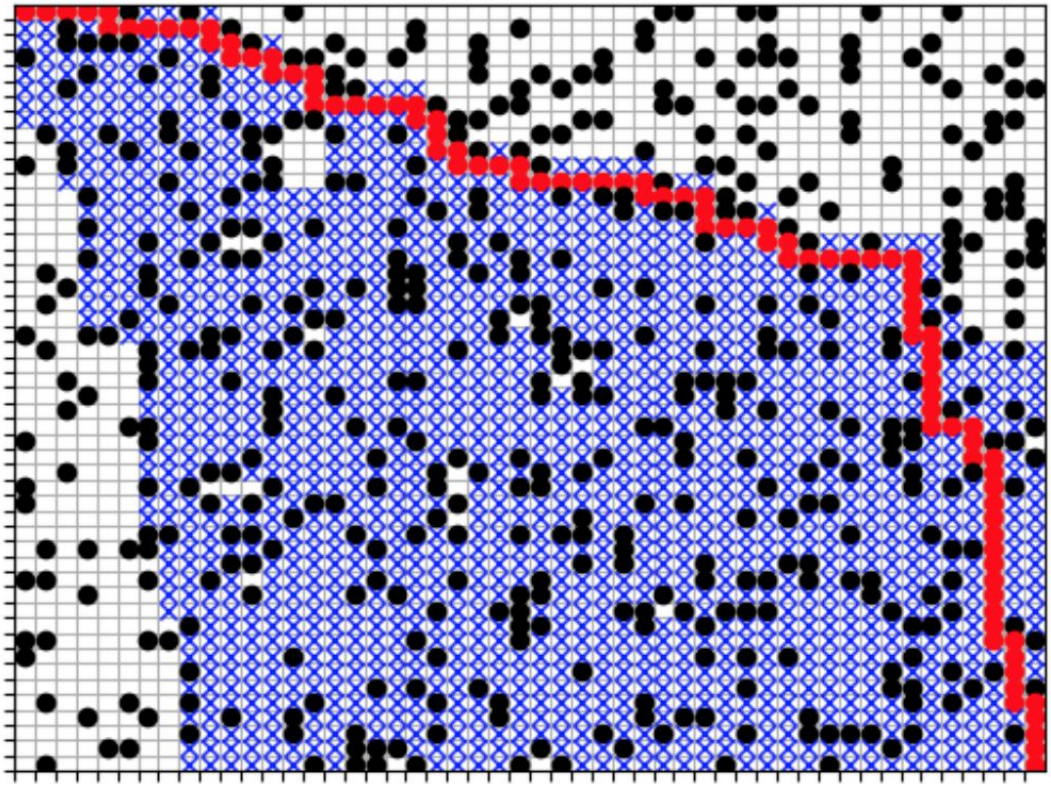
d) A* with Manhattan Distance Heuristic

A* (Manhattan Heuristic)	Total number of nodes expanded	Length of solution path returned	Maximum size of fringe during runtime
Parent maze cost	**1329**	**99**	**150**
Hard child #1	**1798**	**187**	**196**
Hard child #2	**1785**	**157**	**180**
Hard child #3	**1643**	**203**	**255**

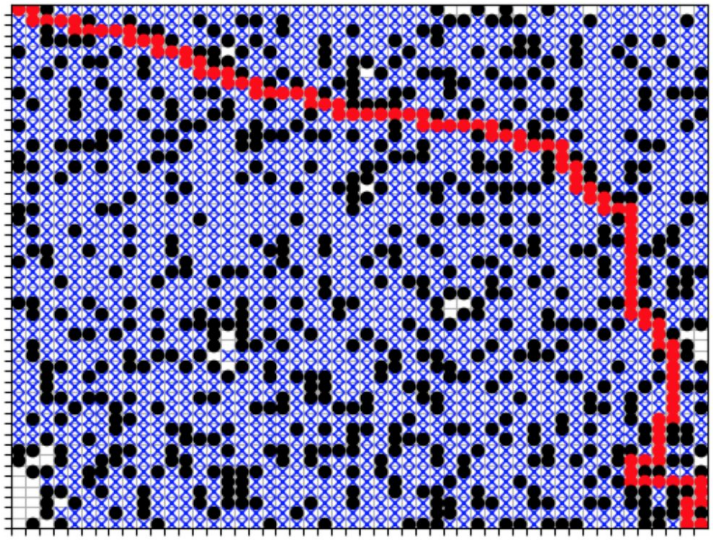
Below parent #1 and parent #2 are plotted. On each subsequent page, the 3 children for max number of nodes expanded, max length of solution path, and max size of fringe are all also plotted, respectively.



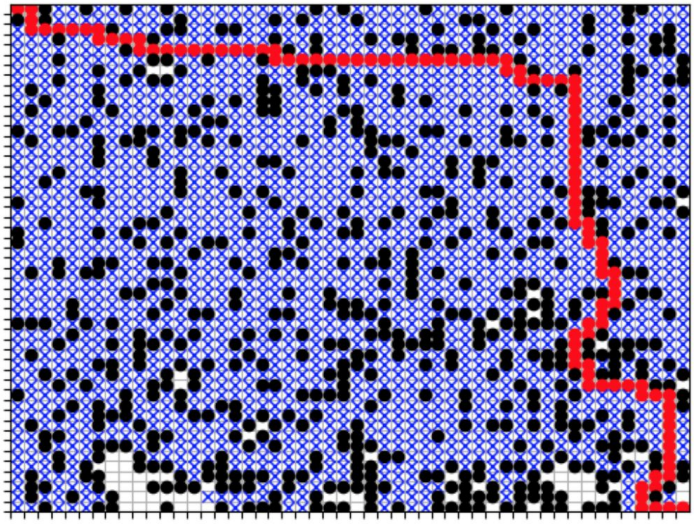
Parent #1



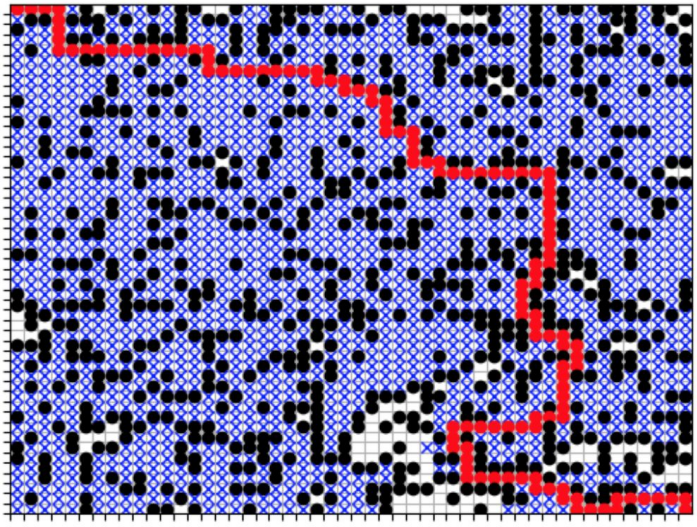
Parent #2



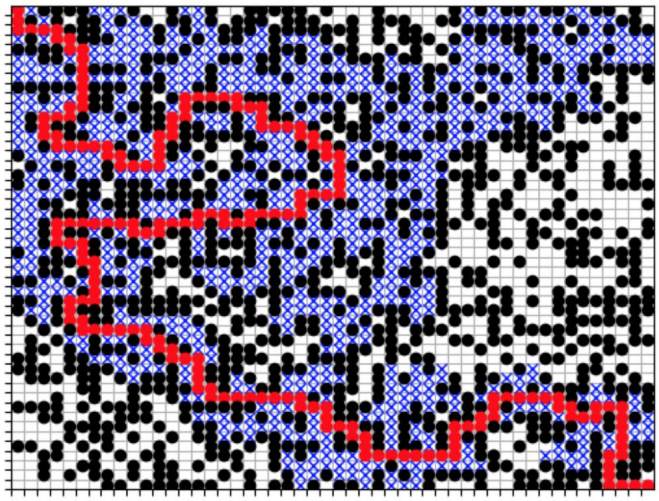
Hard child #1 with max nodes



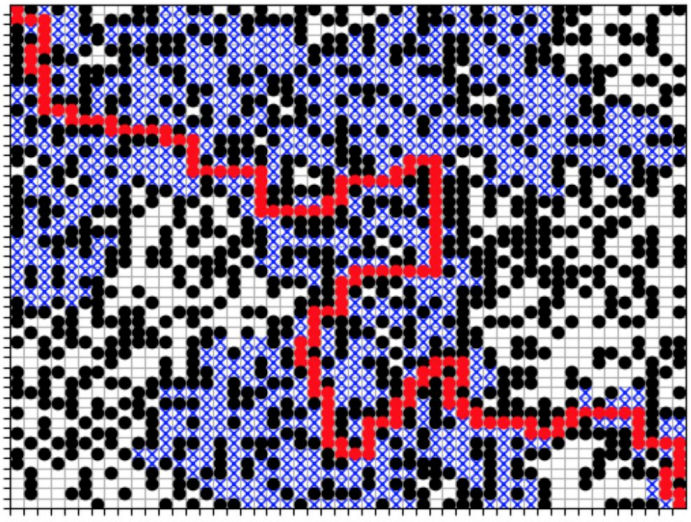
Hard child #2 with max nodes



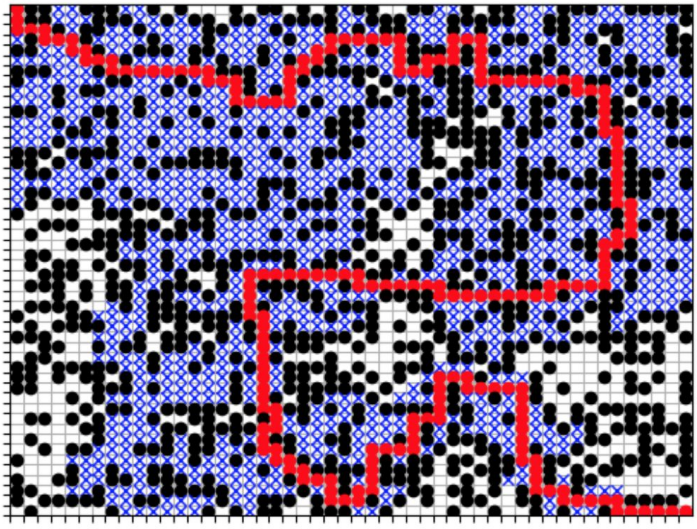
Hard child #3 with max nodes



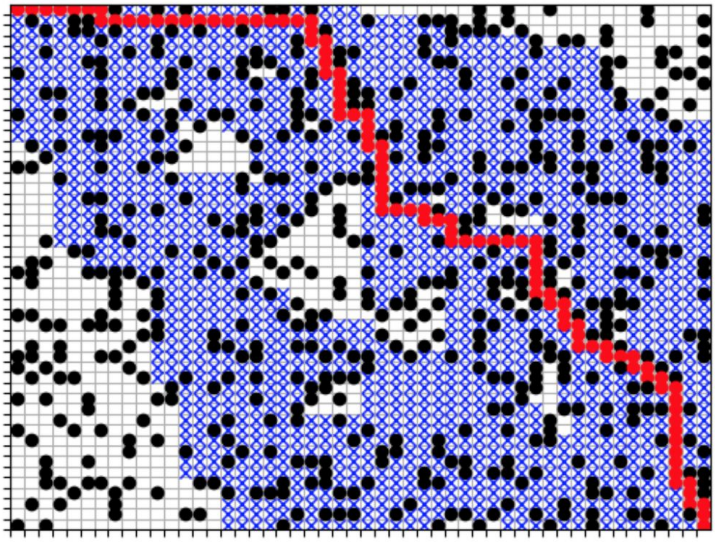
Hard child #1 with max length



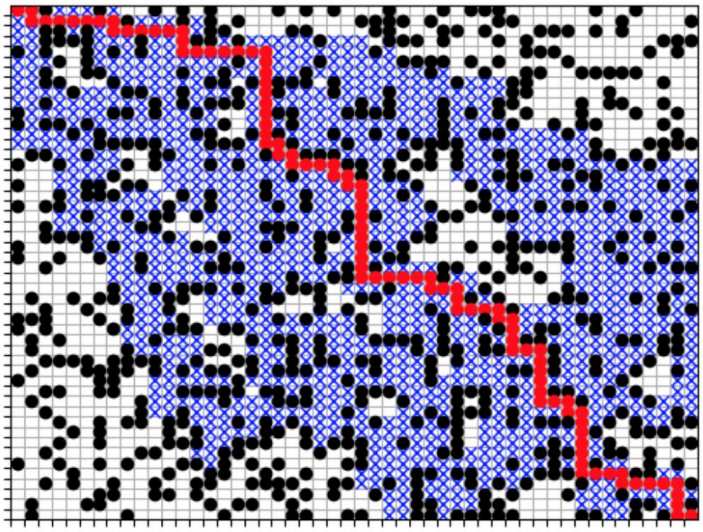
Hard child #2 with max length



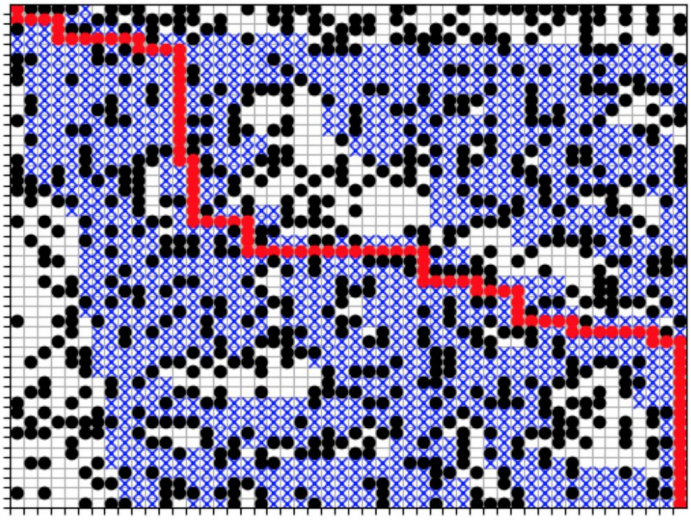
Hard child #3 with max length



Hard child #1 with max fringe



Hard child #2 with max fringe



Hard child #3 with max fringe

Contribution breakdown:

To start off, we split the team into 2 different groups: BFS/DFS, and A* Euclidean/A* Manhattan. Jimmy and Kyung focused on BFS/DFS while Muhammad Aunns and Baber focused on the two A* algorithms. After we got each algorithm working properly, we then worked together to make the maze generation and individual algorithms run smoothly with each other. When part two came around, we each had a different idea for what type of algorithm we could utilize to create a “hard” maze, but in the end, after a lot of brainstorming and planning, we were able to create a very efficient genetic algorithm that we believe gave us accurate solutions to the questions. The questions were split evenly among every group member. Afterwards, every member in the group went over the others’ answers to confirm the accuracy of each response. In order to efficiently collect data, every member ran the algorithms to record costs and generate graphs.