Chakshu Tandon and Zackary Mages
Intro to Artificial Intelligence
Dr. Troy McMahon
February 24, 2020

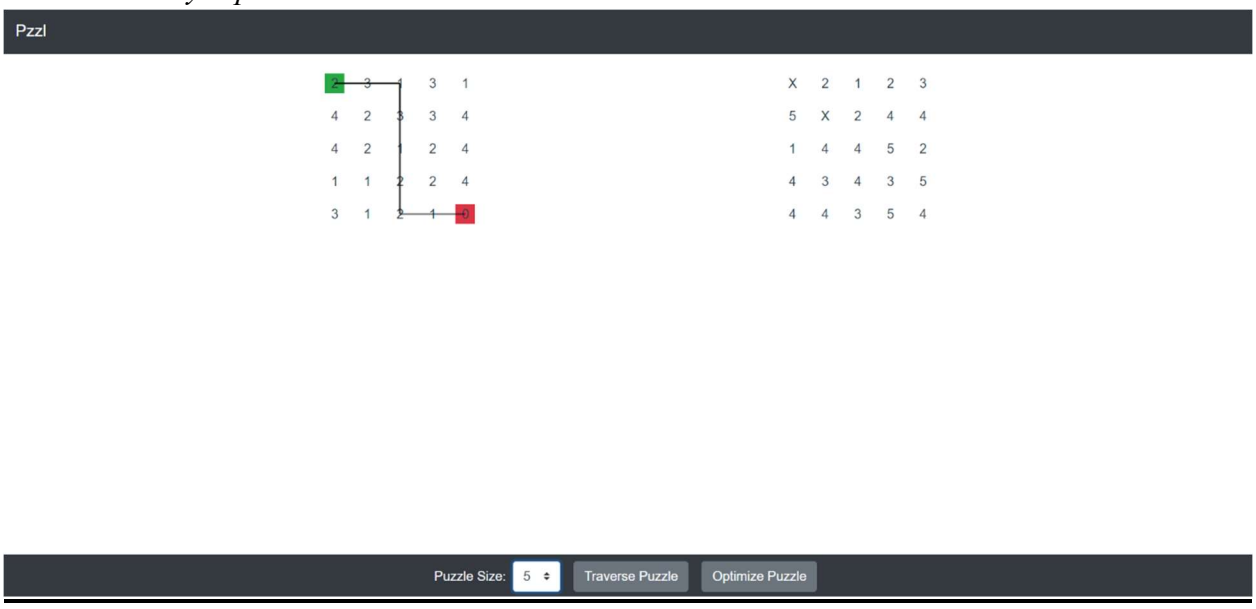
Project 1 Report

Task 1

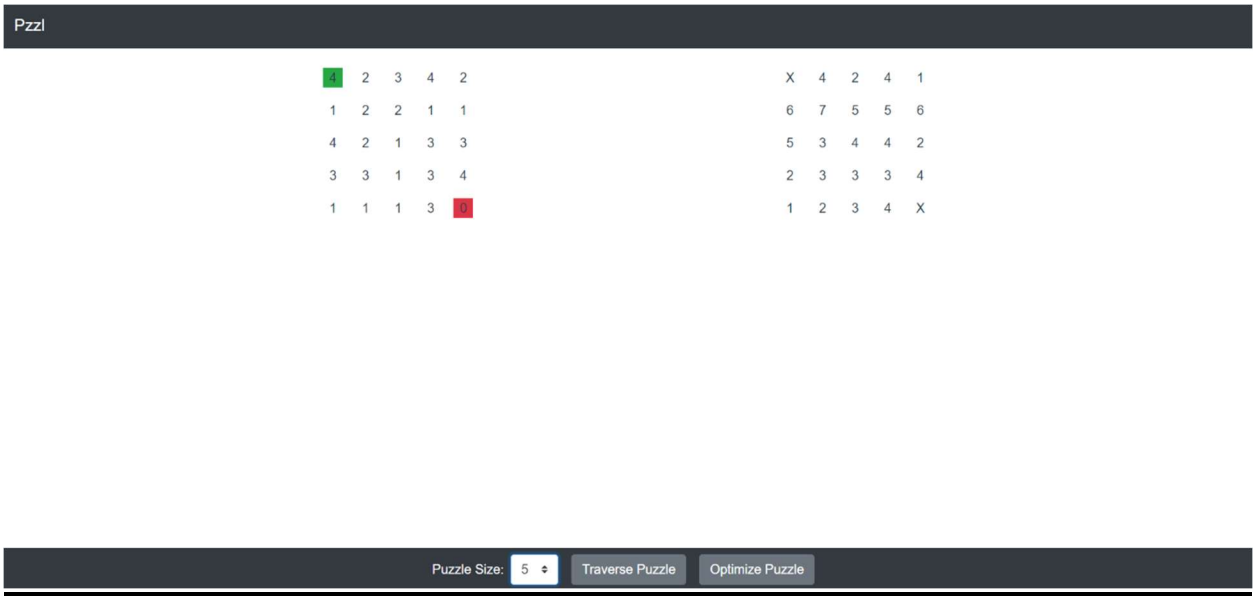
Environment: The n-by-n puzzle
State Space: Depending on the task, an array of n arrays of length n, or a directed graph where vertices are cells in the puzzle, edges are legal moves, and the out-degree is 0 for the goal cell vertex and positive otherwise. We defined a method to convert a array to a graph.
Actions: A valid horizontal or vertical move based on the value of the cell the agent is in
Perception/Observations: The cell the agent is in
Transition Function: Once we move the agent (the action), we must update the array to indicate which cell the agent is now in, and we have to update the graph to indicate which node the agent is now in.
Evaluation Metric: For the cell you are in, we evaluate a move based on how optimal it is for the path length to the goal. For the evaluation of the entire puzzle (for hill climbing and genetic algorithms), we evaluate a puzzle based on the path length to the goal and the number of unreachable cells from the start position. This evaluation function is bounded by $-n^2$ and n^2.

Task 2 and Task 3

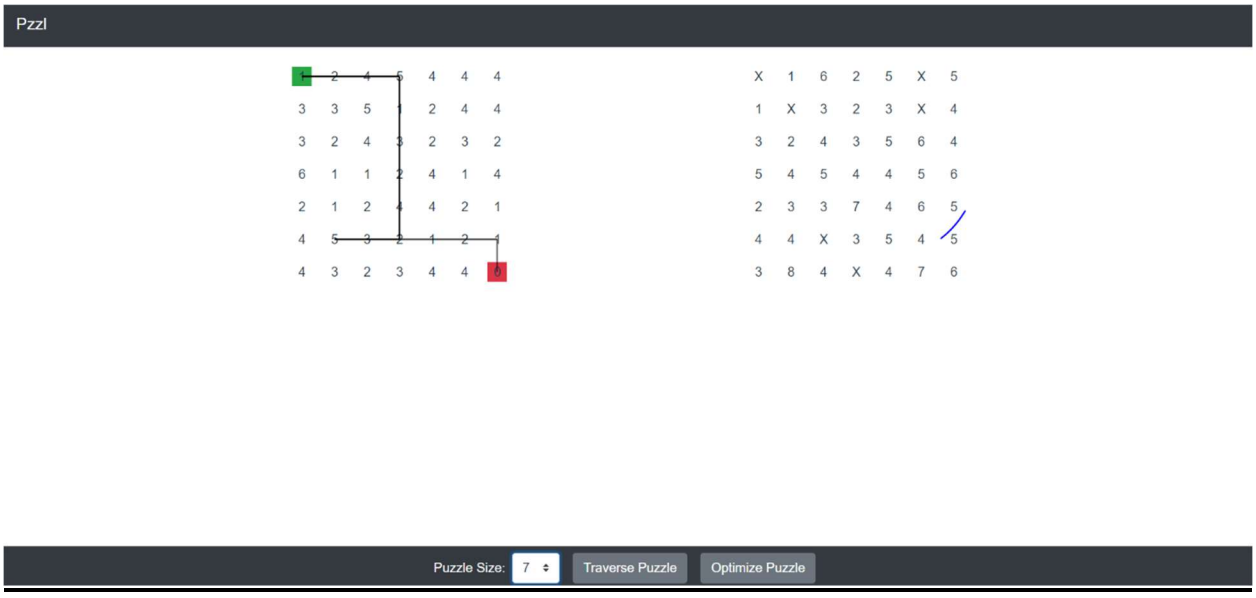
A solvable 5-by-5 puzzle



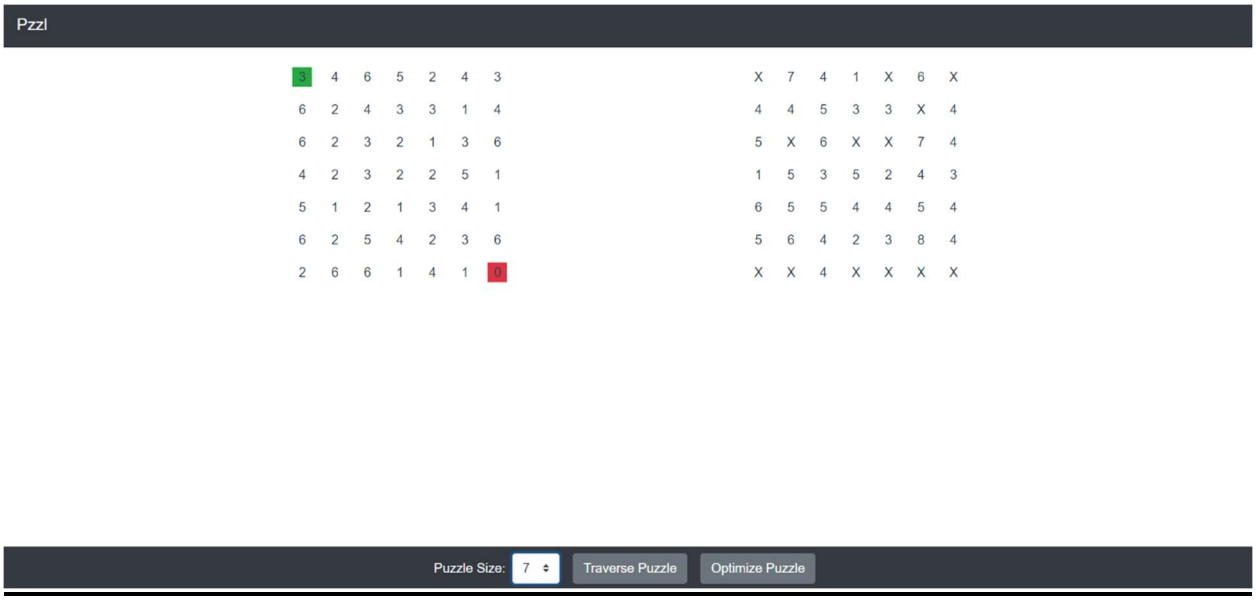
An unsolvable 5-by-5 puzzle



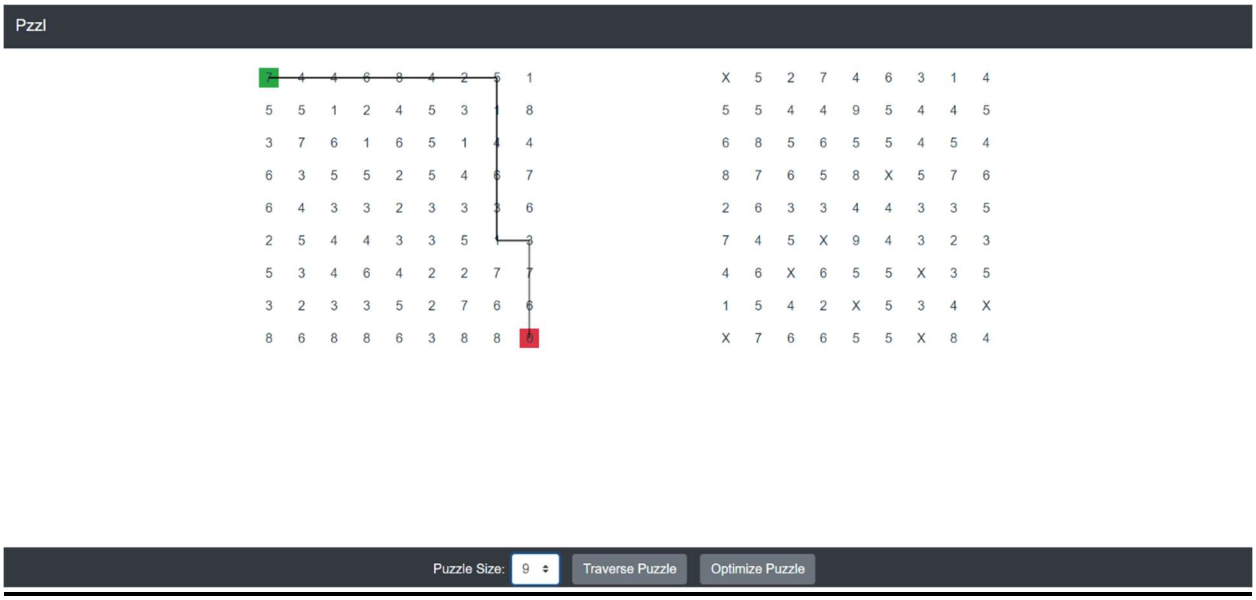
A solvable 7-by-7 puzzle



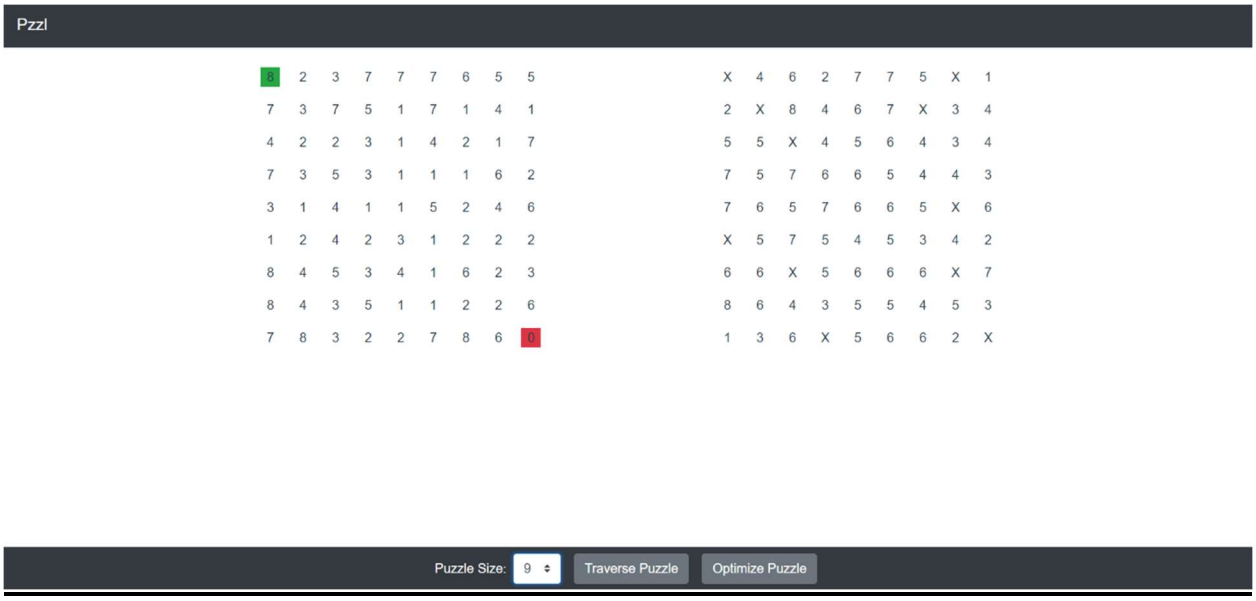
An unsolvable 7-by-7 puzzle



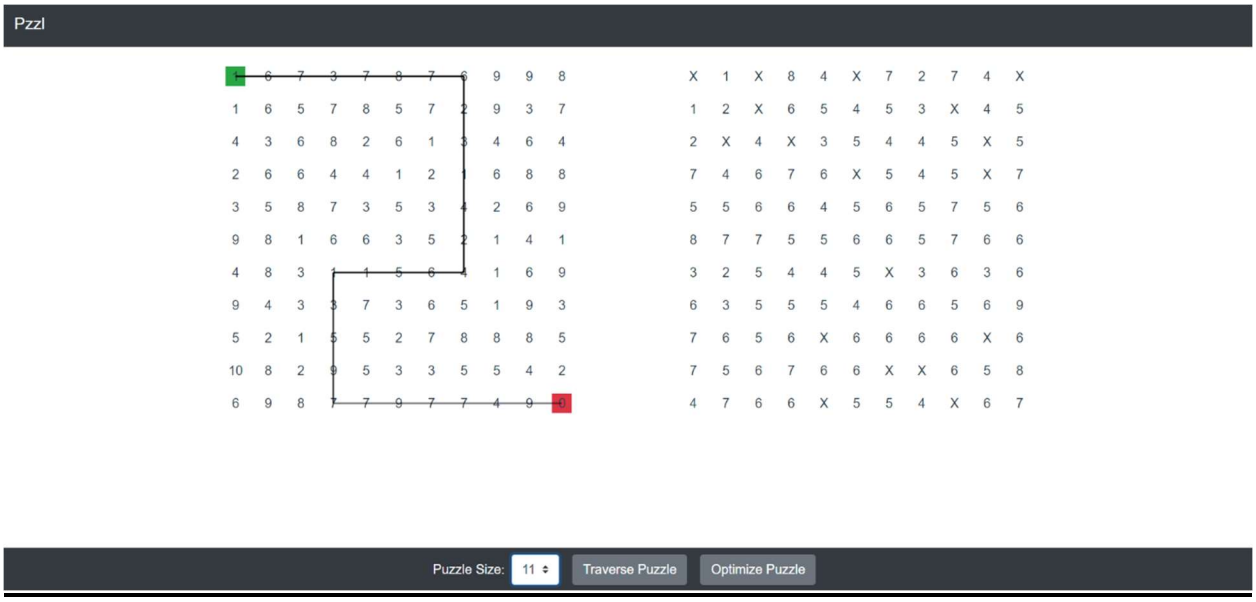
A solvable 9-by-9 puzzle



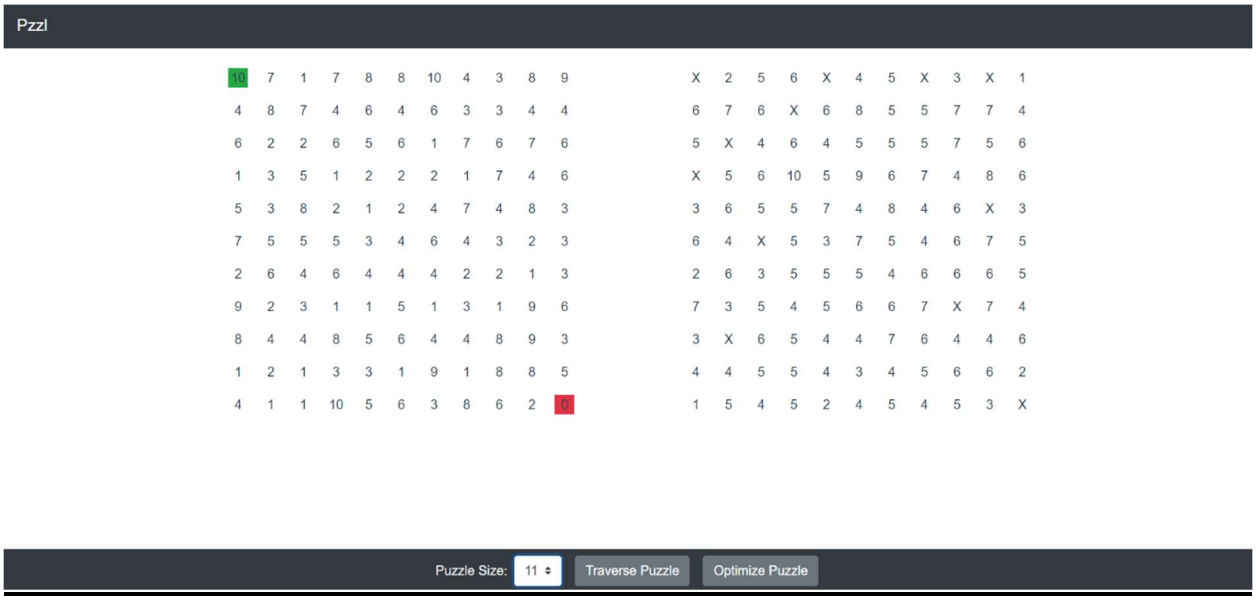
An unsolvable 9-by-9 puzzle



A solvable 11-by-11 puzzle

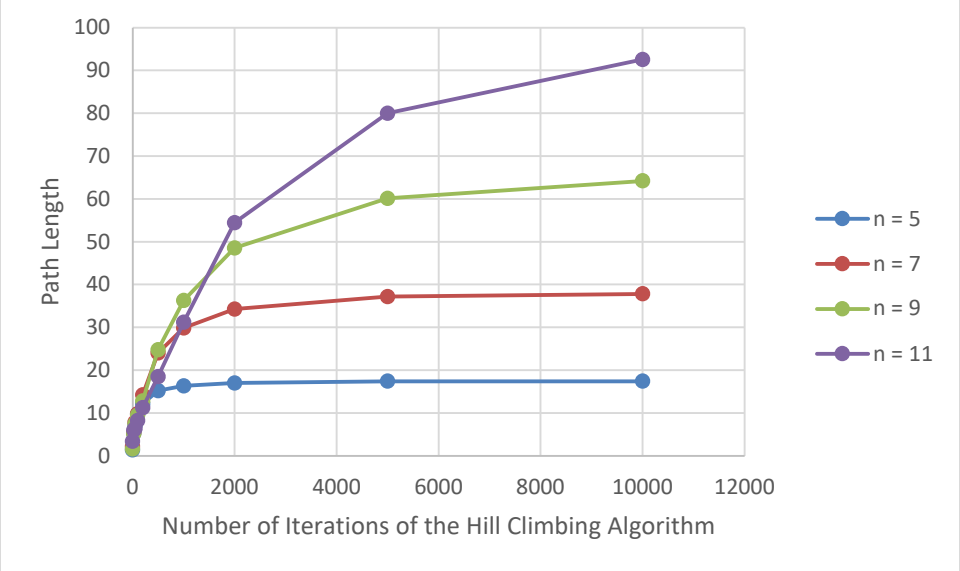


An unsolvable 11-by-11



On our GUI, you click the “Puzzle Size” tab on the bottom of the page. You choose a puzzle size, and the output above it is the puzzle itself and the grid of path lengths from the start position. The optimal path length is also outputted on the original puzzle each time as well, courtesy of BFS and SPF. In the grid on the right, the number in the goal cell corresponds to the optimal path length from the start position. When the puzzle is not solvable, no path will appear on the puzzle. You can then click on the “Optimize Puzzle” tab to generate an improved, harder version of the puzzle with its corresponding path to the goal.

Task 4



Above is the plot of the number of iterations of the hill climbing algorithm and the path length of the corresponding n-by-n puzzle. To produce this, 100 n-by-n puzzles were created, and the hill climbing algorithm with a specific number of iterations was performed on each of them. The path length is the average of those 100 puzzles after a certain number of hill climbing iterations.

It appears that we were approaching or at the limit for the path lengths of 5-by-5 puzzles and 7-by-7 puzzles. However, the functions for 9-by-9 and 11-by-11 puzzles seem to still be increasing, so if we ran 200,000, 500,000, or even 1,000,000 hill climbing iterations on these two, we may have gotten larger average path lengths.

Task 6

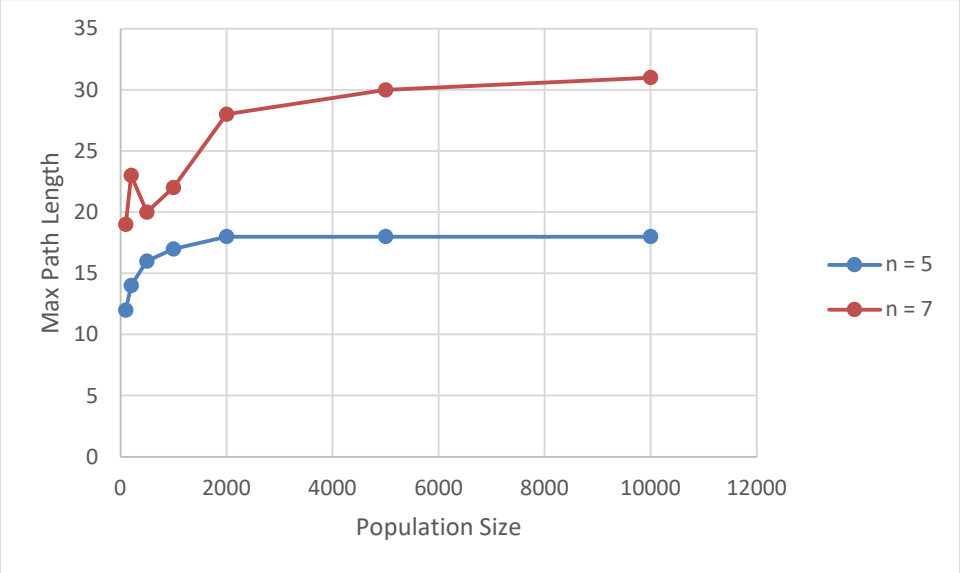
Our heuristic was the inverse of the value at a particular cell. We prioritized moving to a cell with a higher move value. The reason we chose this is because larger moves make it easier to search and traverse the puzzle. It turns out our heuristic was not strong computationally, as it was much slower than BFS and SPF (see below). Ideally, A* would provide the optimal path at the quickest time, but we did not create a good enough heuristic.

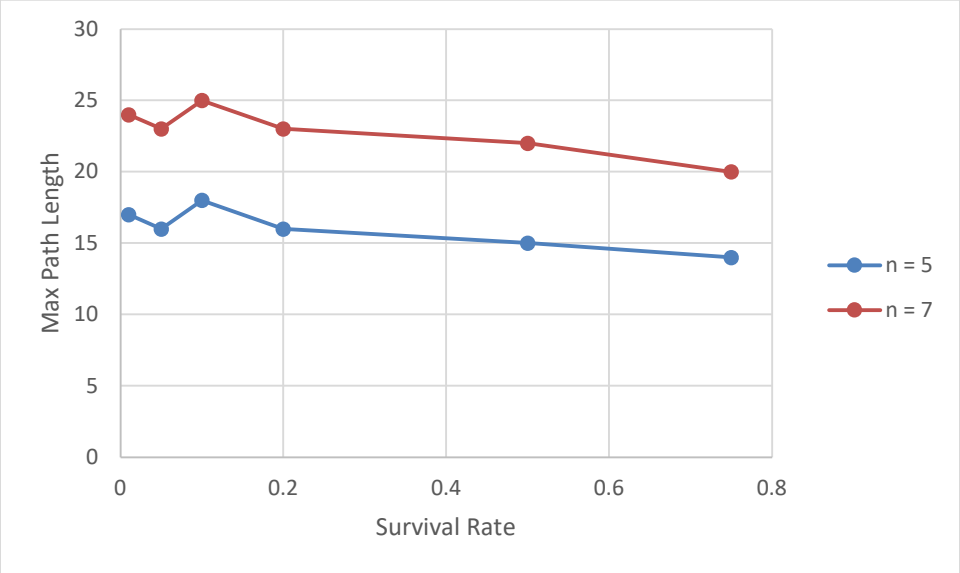
Task 7

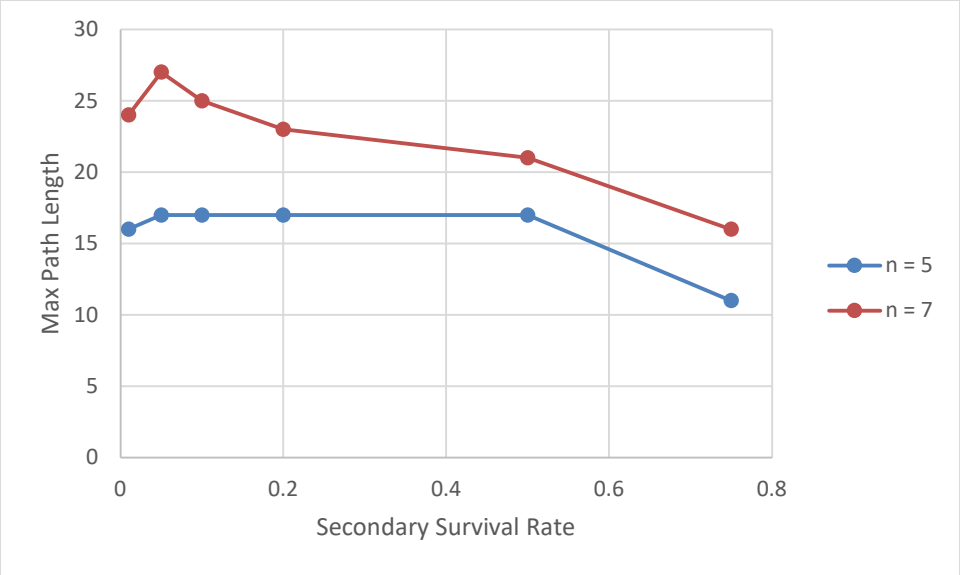
For our population-based approach, or genetic algorithm, we decided to implement a process that contained selection with and without elitism, mutation and breeding. The parameters we use are population size, a survival rate, a secondary survival rate, a mutation rate, and number of epochs/generations. To start, we create a population of n-by-n puzzles based on the number we provide for the population size parameter. We then use our evaluation function to calculate the path length of each of the members of the population. All members are subsequently sorted in descending order by path length, and we picked the top percentage based on the survival rate. This parameter is given as a decimal, and we multiple it by the population size to get the number of "elites" we want. From the remaining population, we randomly choose more members who were simply lucky and survived natural selection. We determine this number by multiplying the secondary survival rate (given as a decimal) by the population size. The last thing we do to this parent population is throw in some mutations. We multiply the mutation rate (given as a decimal) by the population size, and based on this number, we randomly pick a puzzle from the parent population and replace it with a new puzzle. Once this is done, we generate a children population to bring the entire population back to its original size. We randomly pick two distinct parents, and we create a child that has either the mother's first row or father's first row and so on, thus creating a puzzle of the same size and ensuring the child has all legal moves. This becomes the new start population, and the algorithm starts over on that population. In total, it runs the amount of times we designate in the epoch parameter, which indicates the number of generations we

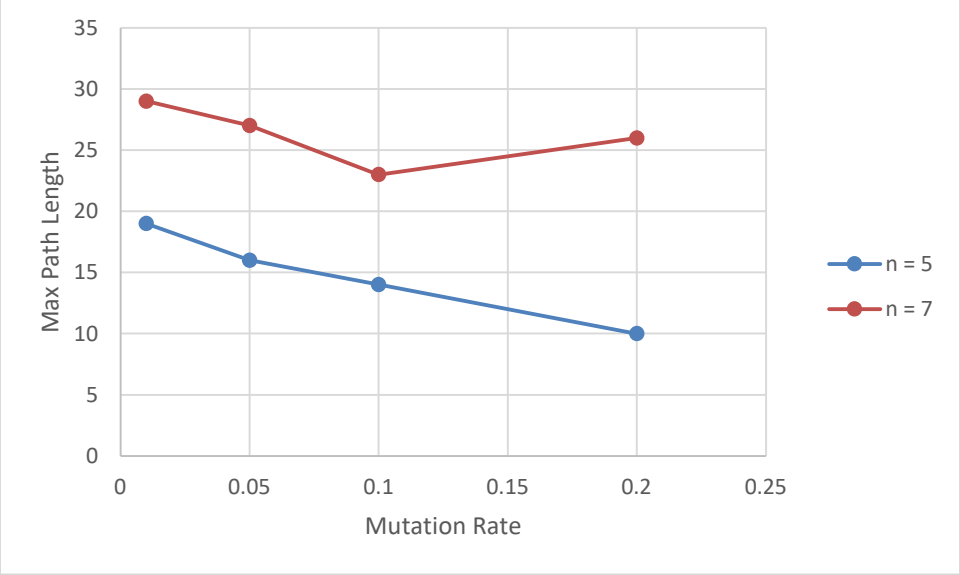
want. We find the maximum path length of the members of the final population as our indicator of how the algorithm performed.

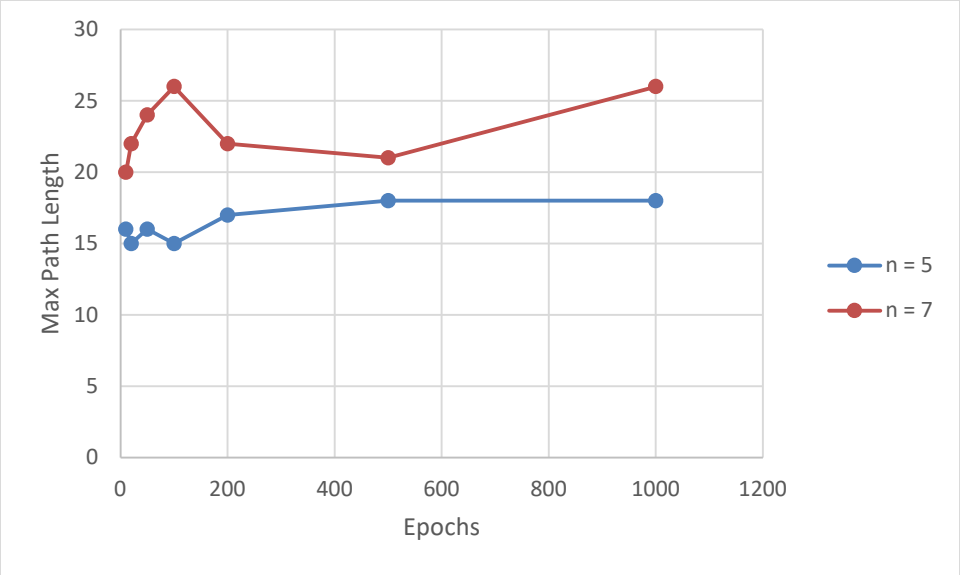
To analyze the impact of the parameters of our algorithm, we considered the base state for each of the parameters to be as follows: population size = 1000, survival rate = 0.2, secondary survival rate = 0.05, mutation rate = 0.01, epochs = 100. We evaluate the parameters by seeing their impact on the maximum path length of the final population after changing each of them and holding the other variables constant. We do this for 5-by-5 and 7-by-7 puzzles.






30
25
20
15
10
5
0
Max Path Length
0
0.2
0.4
0.6
0.8
Secondary Survival Rate
n = 5
n = 7

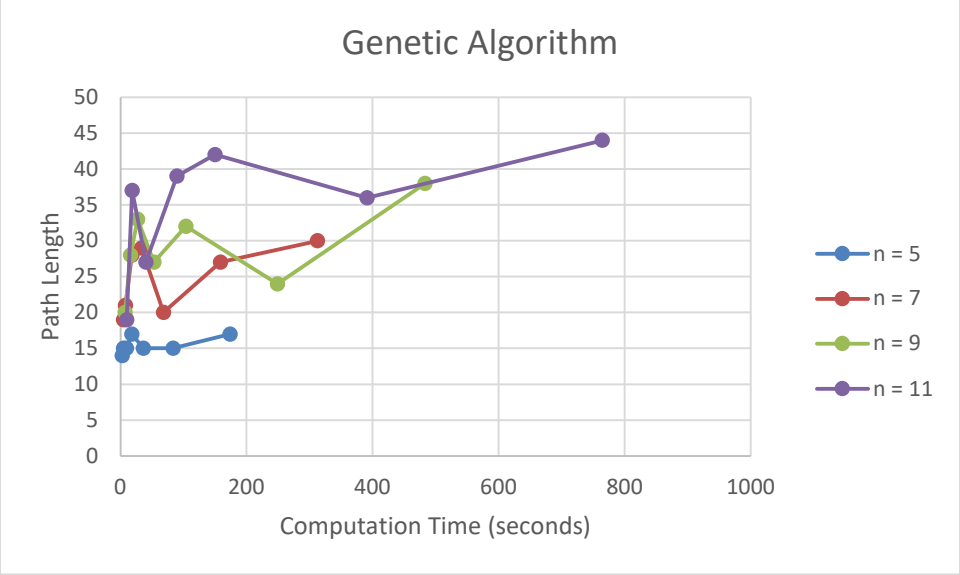

35
30
25
20
15
10
5
0
Max Path Length
0
0.05
0.1
0.15
0.2
0.25
Mutation Rate
n = 5
n = 7



Based on the instructions, we decided only to compare our genetic algorithm to the hill-climbing algorithm since they both involve optimizing the puzzle (or making it more difficult). It is not ideal to compare the genetic algorithm with BFS, SPF or A* because those are search algorithms.

To evaluate hill climbing, we created 50 n-by-n puzzles and ran the hill climbing algorithm on each puzzle. The algorithm was run using 10,20,50,100,200,500 and 1000 iterations. At the end, we took the average of the path lengths of the 50 puzzles. The computation time corresponds to the amount of time it took to calculate the path of length of the 50 puzzles using a certain number of hill climbing iterations. The results are below.

To evaluate the genetic algorithm, we kept all of the parameters constant except the epochs. We set the population size to 1000, the survival rate to .2, the secondary survival rate to .05 and the mutation rate to .01. We ran certain iterations (epochs) of the genetic algorithm on the population of n-by-n puzzles, and we chose the epochs to be the same as the hill climbing algorithm above to make easier comparisons (10,20,50,100,200,500,1000). The output was the maximum path length of one or more of the members of the final population. We did not have time to run through each puzzle size and epoch size 50 or more times to take an average of the path length like hill climbing. Ergo, comparisons would be difficult between the two plots, but relationships can begin to be seen.



Our genetic algorithm is slower to a solution than the hill climbing algorithm, but when it does get there, it creates a tougher puzzle. For the most part, the path lengths of the n-by-n puzzles are larger after the genetic algorithm is implemented as compared to the hill climbing algorithm. It also clear that as n increases, the iterations of hill climbing and our genetic algorithm need to increase as well to reach a harder solution. For example, the 11-by-11 puzzle generated from the 1000 iteration solution of our genetic algorithm only had a path length of ~45. This is not great compared to the optimal 120. This is even worse in the hill climbing plot, as the 11-by-11 puzzle generated from the 1000 iteration solution had a path length of ~32.

Task 8

We did an extensive evaluation of the genetic algorithm and the hill climbing algorithm above. It appears that our genetic algorithm is slower than hill climbing for most puzzle sizes and difficulties. However, the genetic algorithm does, in general, create harder puzzles.

It turns out that our BFS algorithm was the fastest for all puzzle sizes, followed by SPF and then A*. This is not at all what we expected. We ran each algorithm on 5000 5-by-5 puzzles, 5000 7-by-7 puzzles, 5000 9-by-9 puzzles and 5000 11-by-11 puzzles. We calculated the total time that took and divided by 5000. The y-axis of the plot below is the time per iteration. Normally, A* search is the quickest because it prioritizes a path based on its "cost" and its heuristic and always returns the optimal path. Dijkstra's prioritizes the shortest path, so it's generally faster than BFS. The fastest BFS can run is equivalent to Dijkstra's.