Joseph Jaeschke, Anthony Polo
CS 440
Assignment 1: Local Search

Task 1: Puzzle Representation

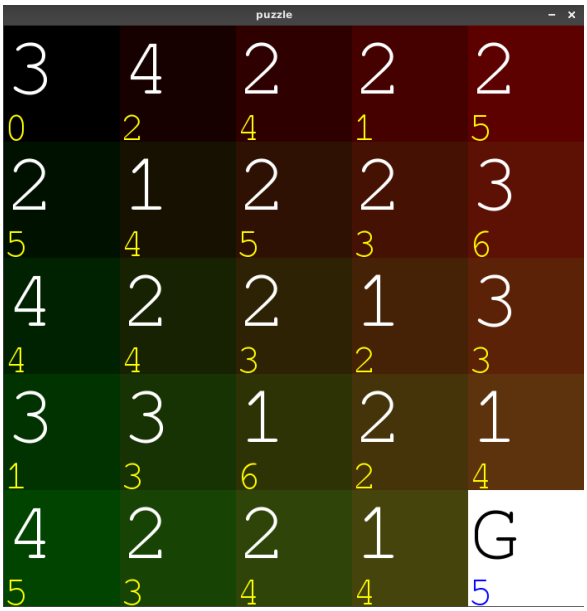
To represent an individual puzzle, we choose to use a GUI which shows both the move number of the cell and how many moves it took to reach that cell. The number of cells to move from the current cell is represented by the larger white number in the top left corner of each cell. The number of moves it took to reach a cell is represented by the smaller yellow number in the left corner. Two special cells in this representation are the start cell in the top left and the goal cell in the bottom left. The start cell will always have a yellow number of zero. This is because there are no number of moves required to reach the start cell. The other special case is the white goal cell. This cell does not have a white move number, rather it has a black “G” to represent the goal. The number of moves it takes to reach the goal is represented by the blue number in the bottom left corner of the cell. This number is important because it is the score of the puzzle (if the goal is reachable). If a given cell is not reachable, it will still have a white move number, but the yellow number will be negative one showing that it is impossible to be reached.

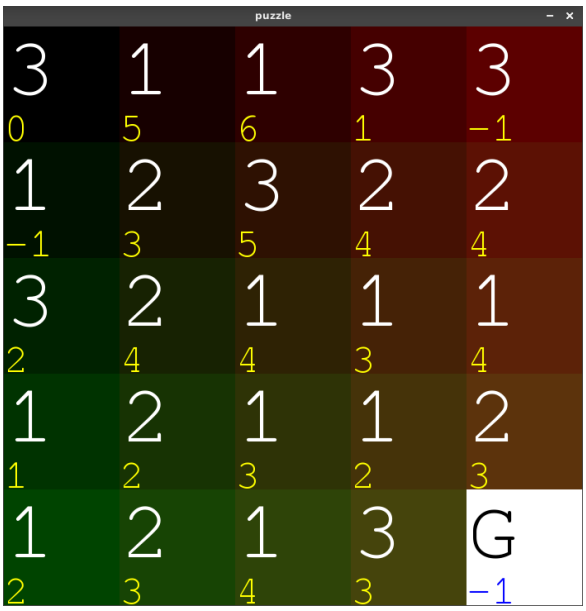
In the program, each cell is represented by a node. Each node contains three fields: a move number, a number of steps it took to reach it, and its parent. The move numbers are initialized to either a random number or a number from a puzzle that was given to the program at startup. The number of steps it took to reach the cell is initialized to negative one. This is because at the time of making the puzzle, it has not been solved and all of those numbers are unknowns. The parent field is set to None for the same reason. A whole puzzle is made of a two dimensional array of nodes of size n by n where n is the length of the puzzle. The length of the puzzle is determined at the start of the program either from the puzzle input from a text file or through the command line if there is no file to input. This size cannot be changed once the program starts.

Task 2: Puzzle Evaluation

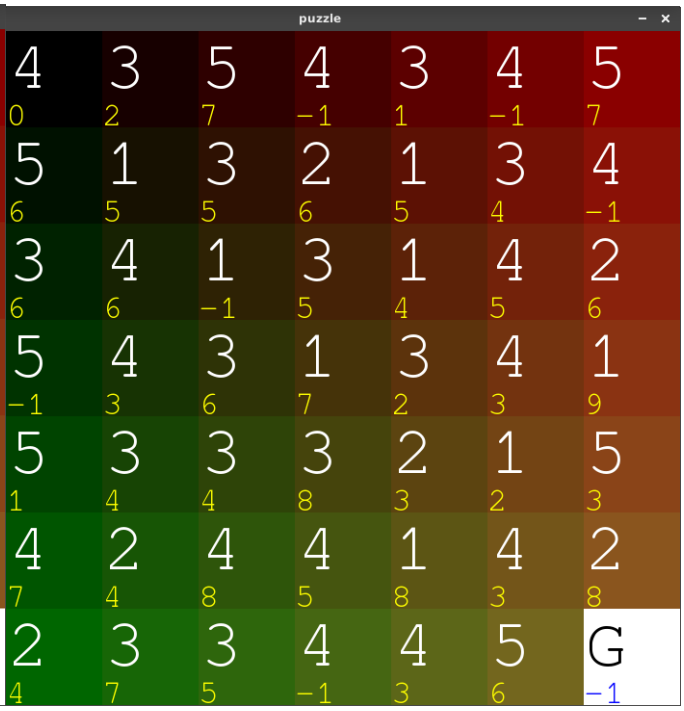
After a puzzle can be represented, it is now needed to solve the puzzle. This is done through a breadth first search algorithm. Before the algorithm runs, the . For every cell in the puzzle, starting with the starting cell in the top left, it's neighbors will be conditionally added to a first-in-first-out queue. The conditions of a cell being added to a queue are that it has not been visited or it can be reached in fewer moves than it previously was. In order to tell how many moves were used to reach a cell, it was necessary to keep track of the parent of each cell. If the number of moves it took to reach the parent, then the number of moves it takes to reach the current cell is just one more. This can be traced all the way back to the start where the number would be zero. This algorithm will end when all the reachable cells have been visited.

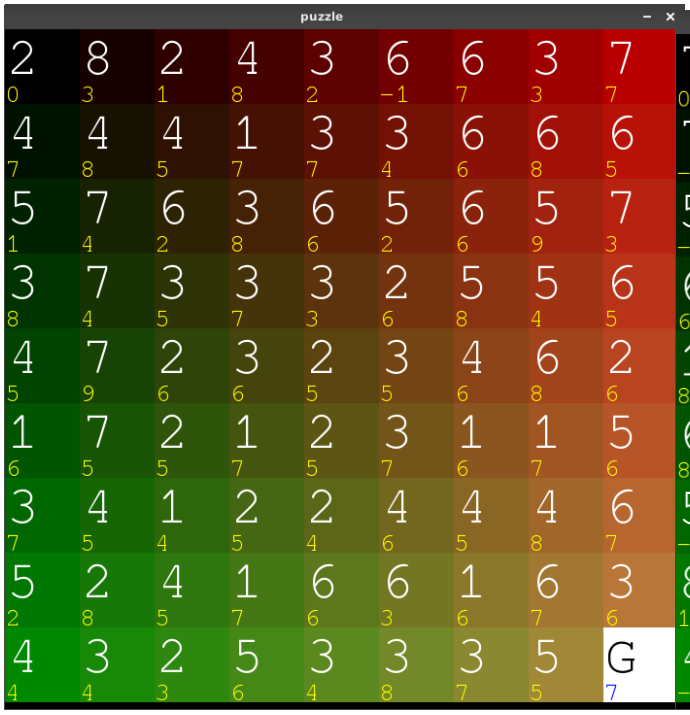
Once the puzzle has been solved, it is then possible to score it based on how many moves it took to reach the goal cell. If the goal cell is reachable, then the score of the puzzle is simply how many moves it took to reach the end. If the goal is not reachable, then the scoring is more complicated. In this case, the score is the number of unreachable cells times negative one. Using this scoring system allows very difficult puzzles to receive a high score while easy undo-able puzzles to receive a low score. Below are examples of solvable graphs and unsolvable graphs for different sizes of n



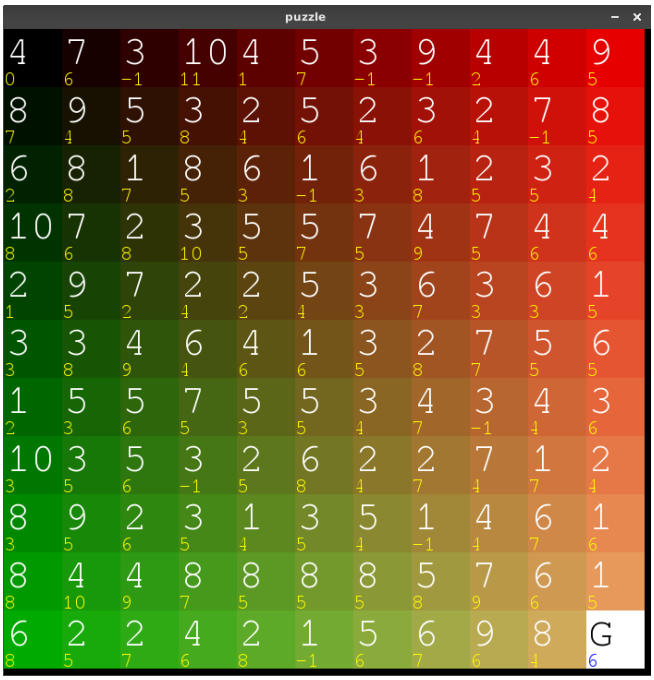


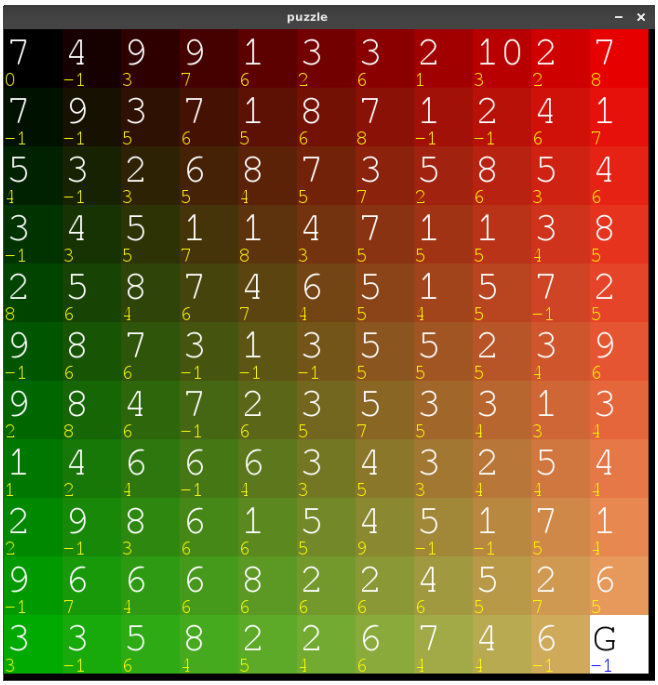

puzzle
1 5 6 1 2 3 4
0 1 3 5 6 −1 2
5 2 3 1 3 5 3
1 5 5 4 5 2 −1
4 1 2 3 2 2 4
6 −1 6 5 7 −1 6
5 4 1 1 1 4 3
3 3 5 5 6 4 −1
1 5 4 1 3 2 4
−1 6 4 4 5 −1 3
6 2 2 4 3 2 1
−1 2 −1 3 −1 −1 −1
3 5 3 2 3 5 G
2 4 4 3 6 3 5


puzzle
4 3 5 4 3 4 5
0 2 7 −1 1 −1 7
5 1 3 2 1 3 4
6 5 5 6 5 4 −1
3 4 1 3 1 4 2
6 6 −1 5 4 5 6
5 4 3 1 3 4 1
−1 3 6 7 2 3 9
5 3 3 3 2 1 5
1 4 4 8 3 2 3
4 2 4 4 1 4 2
7 4 8 5 8 3 8
2 3 3 4 4 5 G
4 7 5 −1 3 6 −1


puzzle
2 8 2 4 3 6 6 3 7
0 3 1 8 2 −1 7 3 7
4 4 4 1 3 3 6 6 6
7 8 5 7 7 4 6 8 5
5 7 6 3 6 5 6 5 7
1 4 2 8 6 2 6 9 3
3 7 3 3 3 2 5 5 6
8 4 5 7 3 6 8 4 5
4 7 2 3 2 3 4 6 2
5 9 6 6 5 5 6 8 6
1 7 2 1 2 3 1 1 5
6 5 5 7 5 7 6 7 6
3 4 1 2 2 4 4 4 6
7 5 4 5 4 6 5 8 7
5 2 4 1 6 6 1 6 3
2 8 5 7 6 3 6 7 6
4 3 2 5 3 3 3 5 G
4 4 3 6 4 8 7 5 7


puzzle
7 7 4 2 6 2 4 2 3
0 4 8 3 −1 2 7 1 3
7 2 3 7 3 4 5 2 1
−1 7 7 8 7 6 6 8 8
5 6 2 5 1 2 1 4 2
−1 5 −1 3 6 3 5 2 4
6 7 4 4 4 5 3 1 4
6 8 8 7 5 5 6 8 4
1 1 5 3 2 1 1 4 6
8 9 5 7 5 4 5 6 5
6 6 2 2 2 4 2 5 4
8 5 7 −1 7 5 6 6 7
5 4 5 5 1 3 3 6 4
−1 4 6 7 6 5 7 3 6
8 1 3 5 5 6 4 3 7
1 3 4 4 6 5 7 −1 2
4 3 4 7 5 2 3 5 G
−1 4 8 7 5 6 9 7 −1

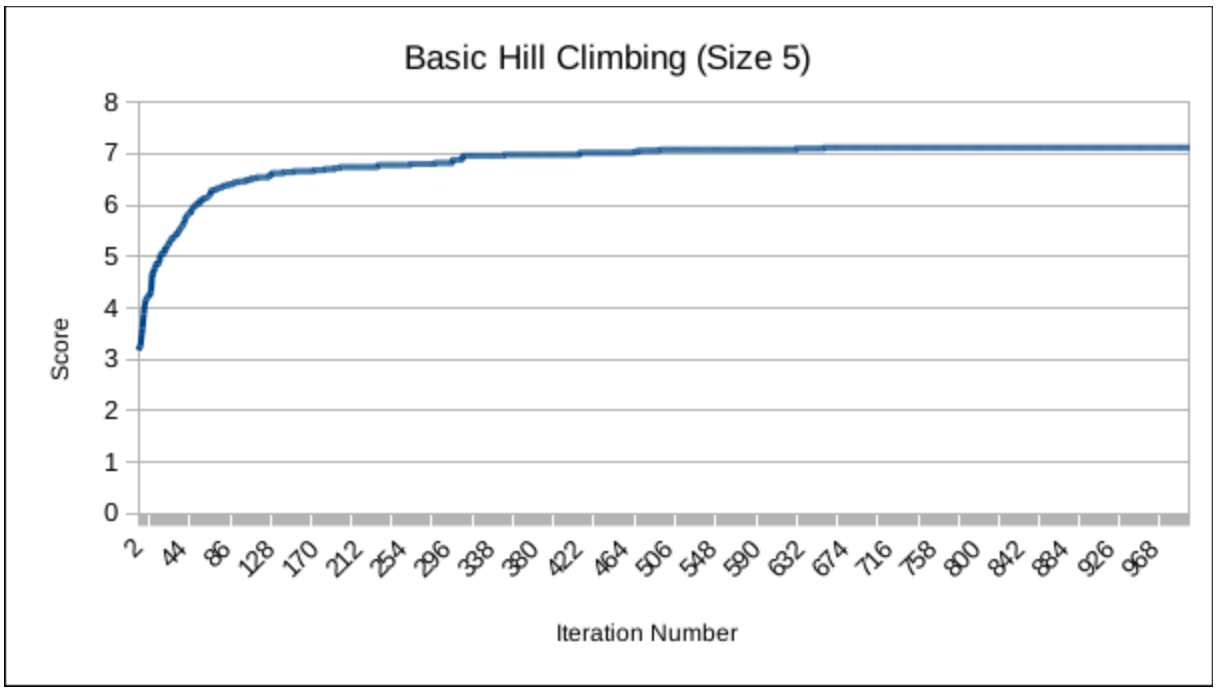




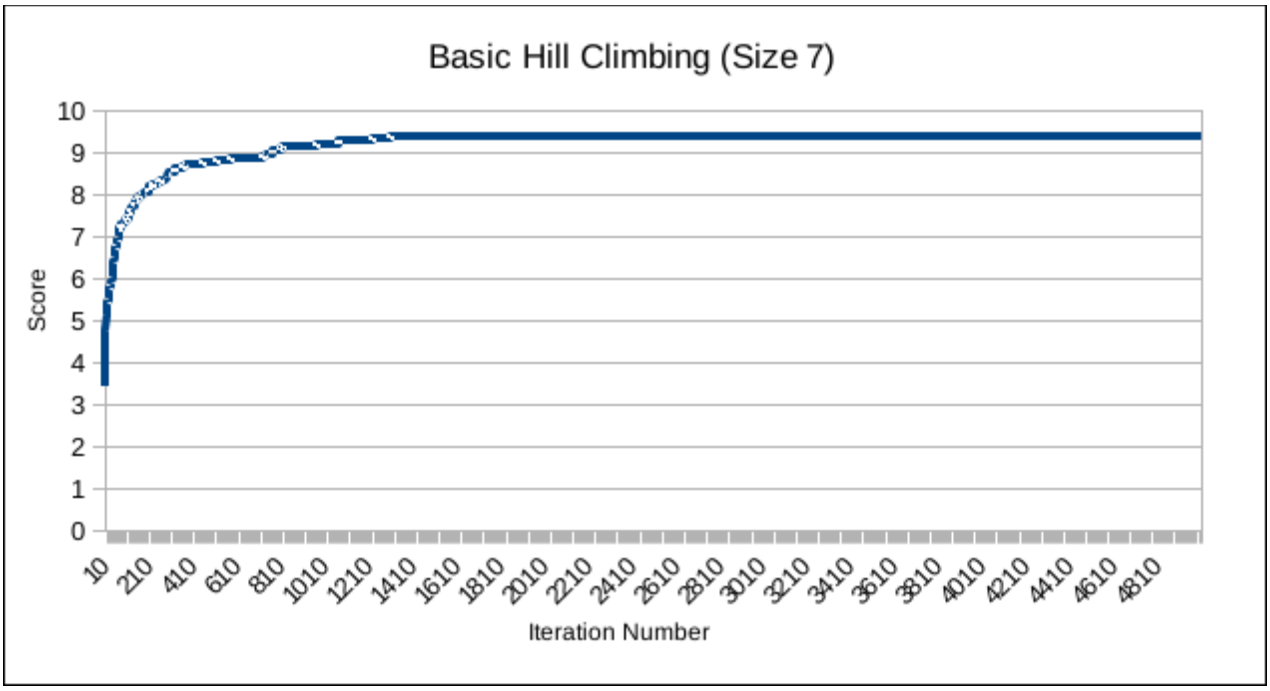
Task 3: Basic Hill Climbing

On average, the basic hill climbing approach was not very good at improving puzzles. For each possible size of puzzle (n=5, 7, 9, 11) for fifty runs each iteration in the hill climbing was averaged together to get a rough average of how well the algorithm improved the puzzle. For many of the puzzles, the hill climbing will plateau fairly early on. This is because there has been a local maximum reached in the state space. All of the puzzles that are one change away from the current one are all worse therefore the algorithm will not make a change. This is a clear limitation because much of the state space will not get sampled. This method of improving puzzles was not very good as the bellow graphs demonstrate.

For n=5, hill climbing was run for a total of 1000 iterations. All of the first iterations were averaged, then all the second iterations, and so on to generate the following graph.

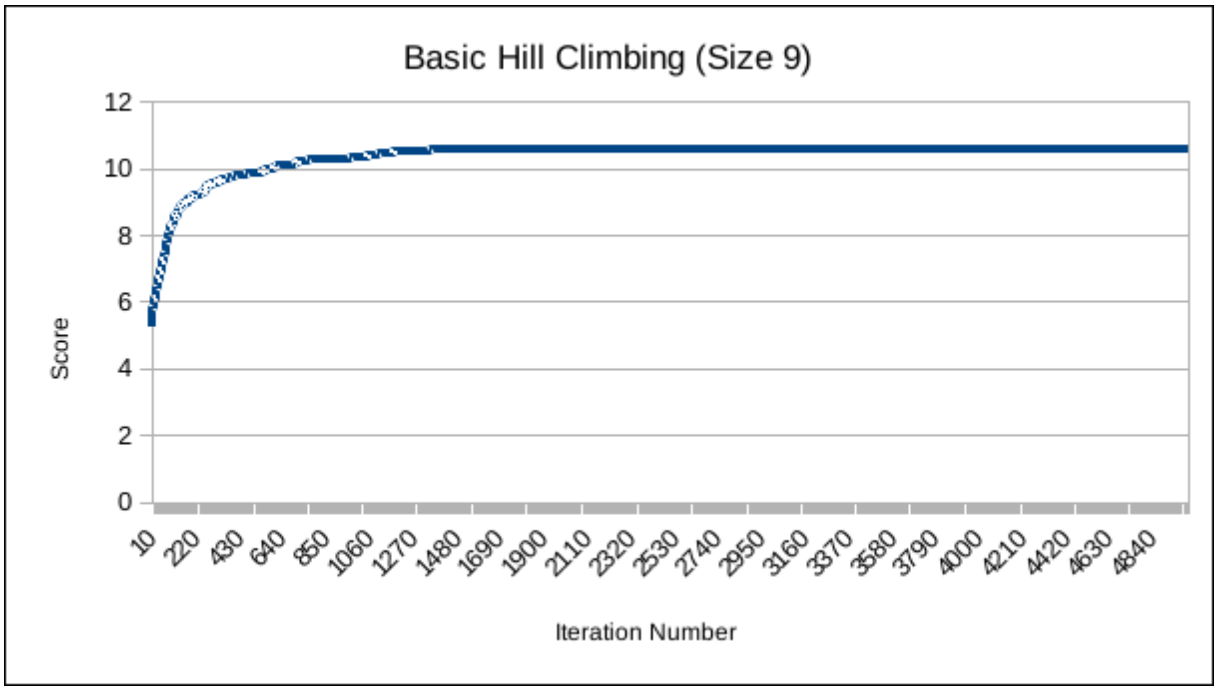


For n=7, hill climbing was run for a total of 5000 iterations. All of the first iterations were averaged, then all the second iterations, and so on to generate the following graph.

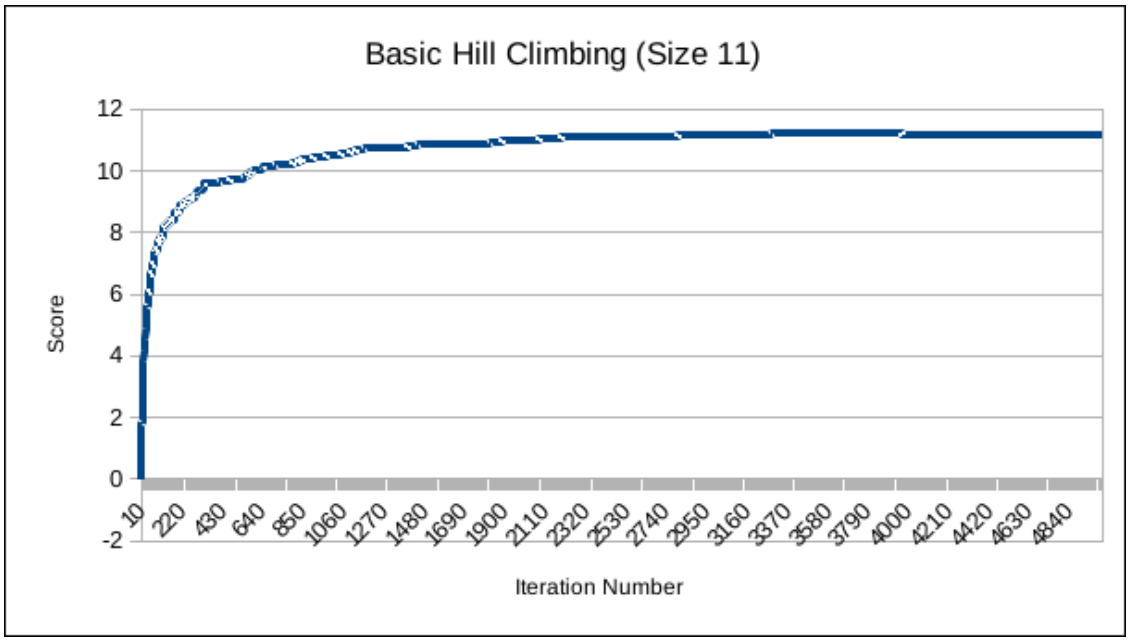


This graph shows a rather large plateau after the 2000th iteration meaning the last three thousand iterations did not have a large impact on average. The algorithm could have been stopped at this point.

For n=9, hill climbing was run for a total of 5000 iterations. All of the first iterations were averaged, then all the second iterations, and so on to generate the following graph.



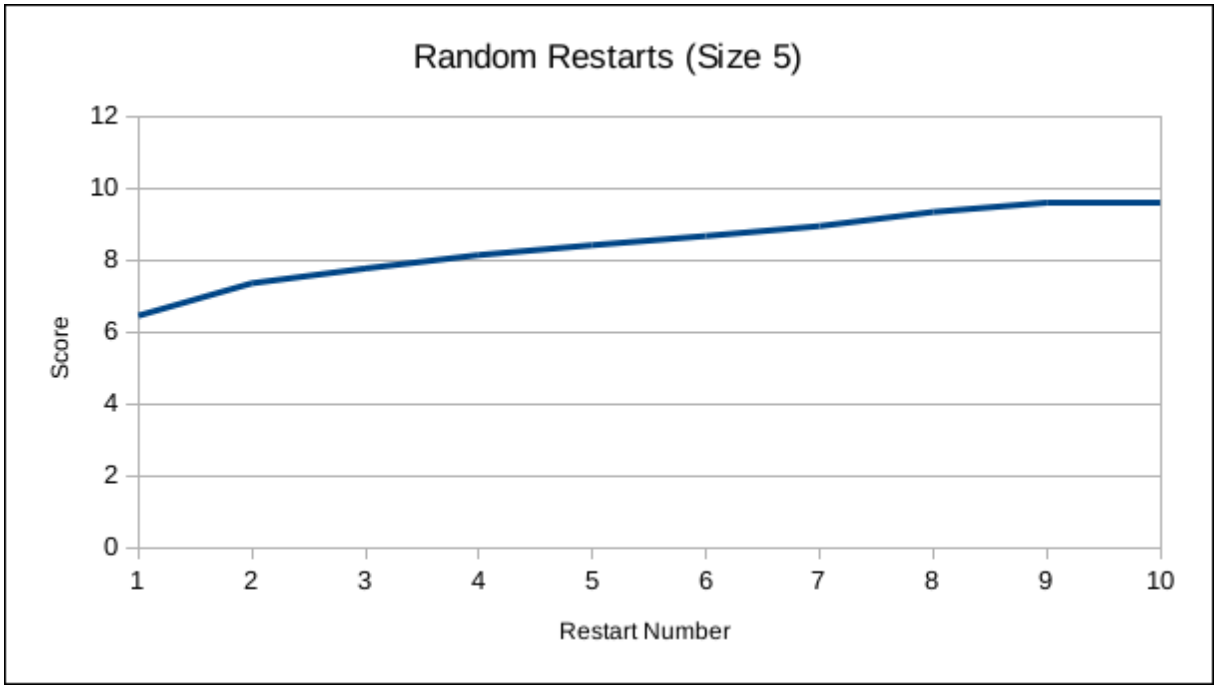
For n=11, hill climbing was run for a total of 5000 iterations. All of the first iterations were averaged, then all the second iterations, and so on to generate the following graph.



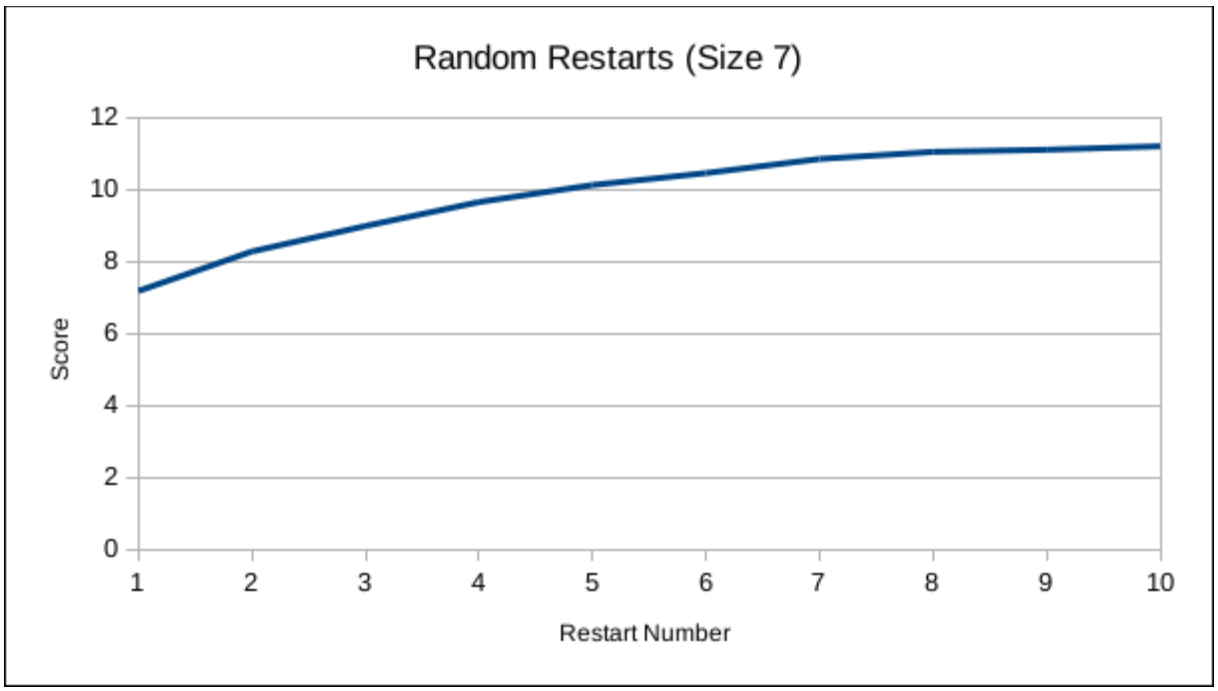
Task 4: Hill Climbing with Random Restarts

One way to improve the hill climbing algorithm is to have restarts after the iterations are over. This allows for more puzzles to be seen and therefore a higher chance of seeing good ones. For the random restarts and random walks, the total number of iterations are the same with n=5, 7 having a total of 1000 iterations and n=9, 11 having a total of 3000 iterations. In the case of the hill climbing, all sizes had ten restarts, so for n=5, 7, there were 100 iterations and when n=9, 11, there were 300 iterations. The graphs of the evaluation function were recorded after the hill climbing was done so the score shown is after every restart of the hill climbing. The random restarts could have benefited from more iterations per restart as there is not a real plateau shown. In order to keep it comparable to the random walks, the total iterations were kept the same. This being said, the random restarts faired better than the random walks, but were overall not too great.

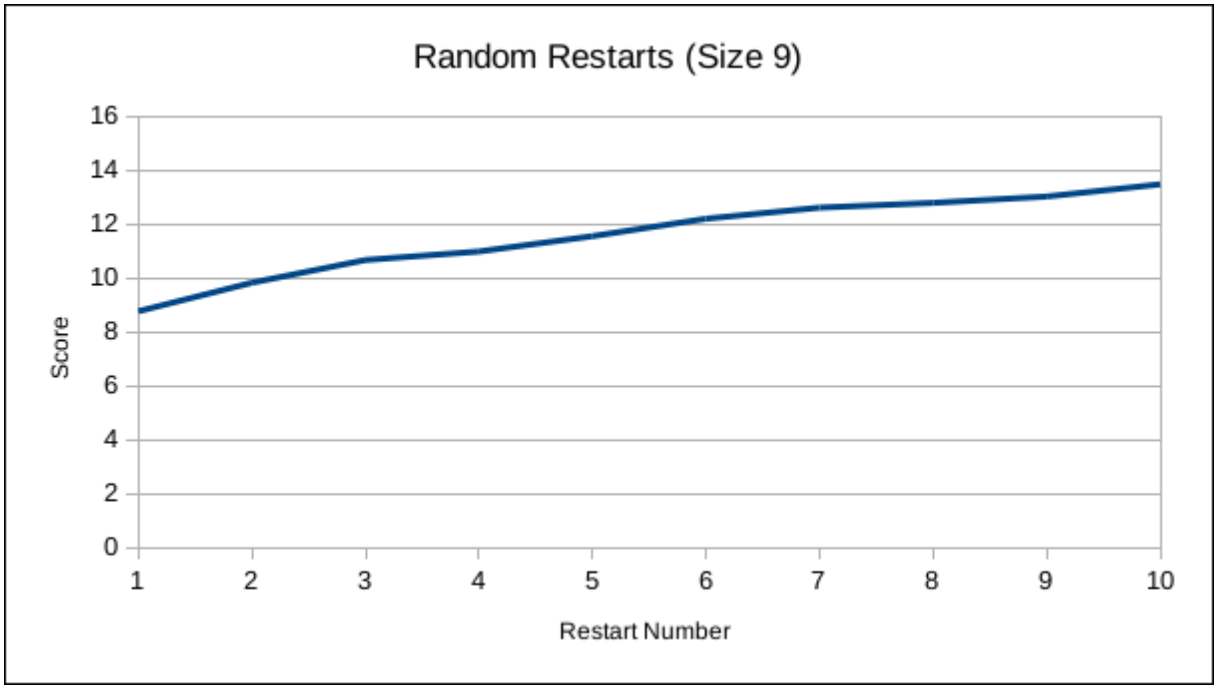
For n=5, hill climbing with random restarts was run for a total of 1000 iterations. All of the first iterations were averaged, then all the second iterations, and so on to generate the following graph.



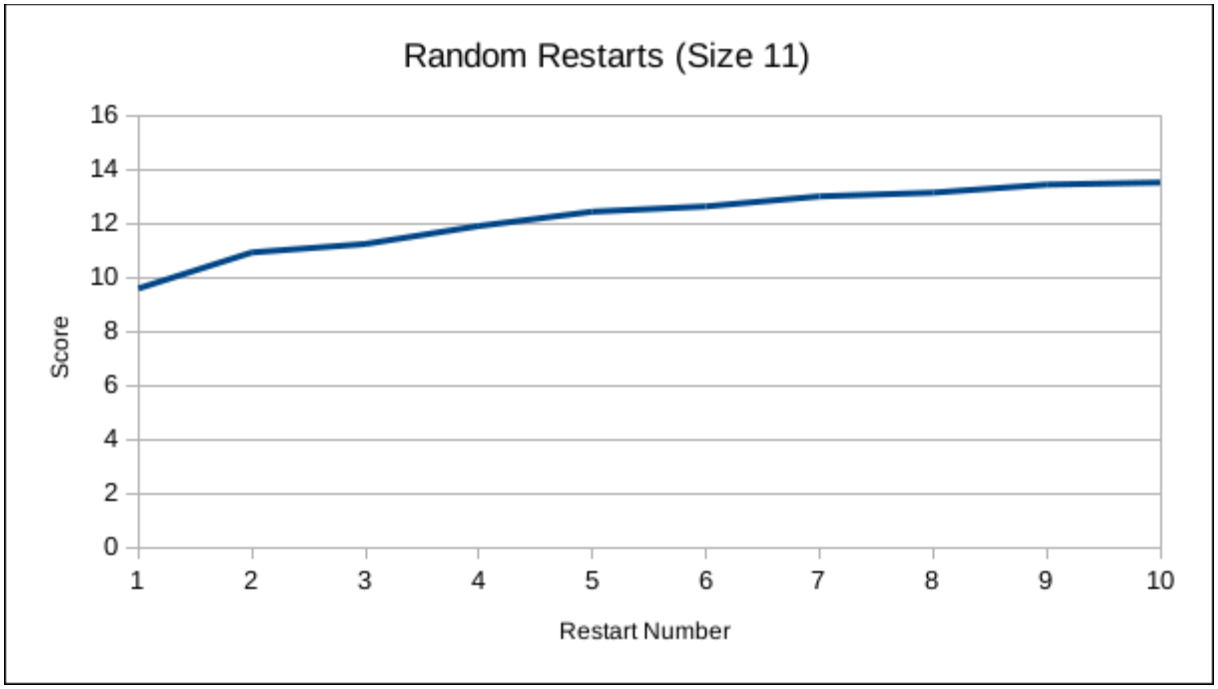
For n=7, hill climbing with random walks was run for a total of 1000 iterations. All of the first iterations were averaged, then all the second iterations, and so on to generate the following graph.



For n=9, hill climbing with random walks was run for a total of 3000 iterations. All of the first iterations were averaged, then all the second iterations, and so on to generate the following graph.



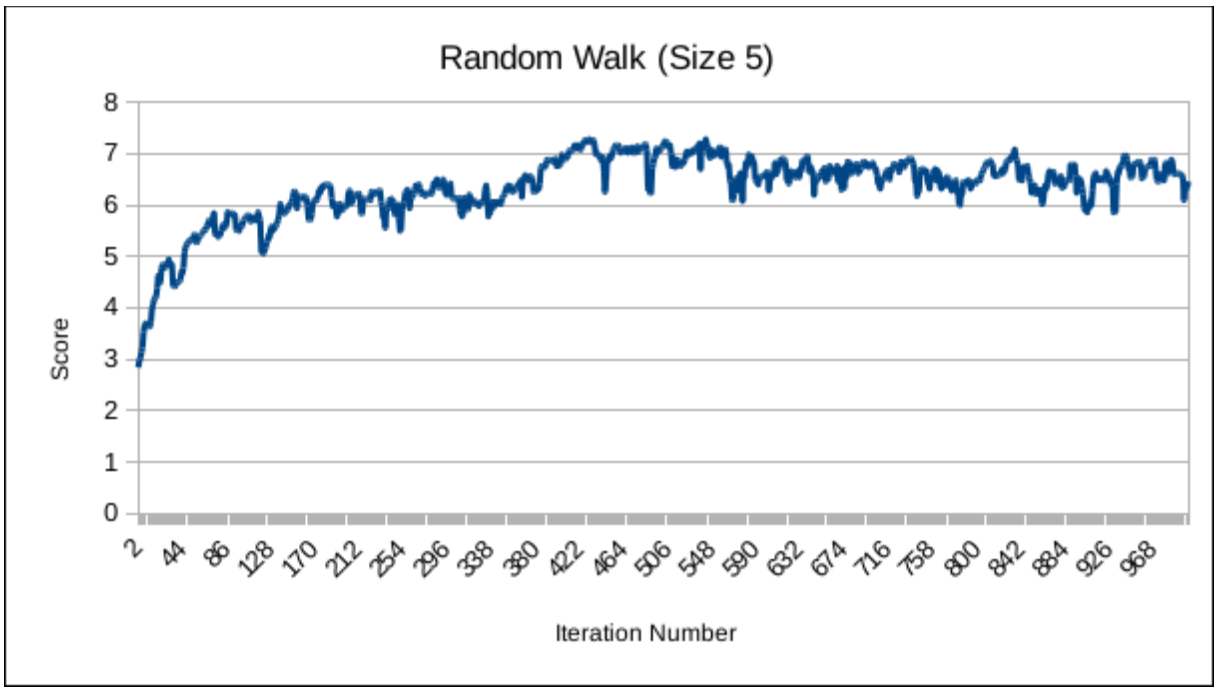
For n=11, hill climbing with random walks was run for a total of 3000 iterations. All of the first iterations were averaged, then all the second iterations, and so on to generate the following graph.



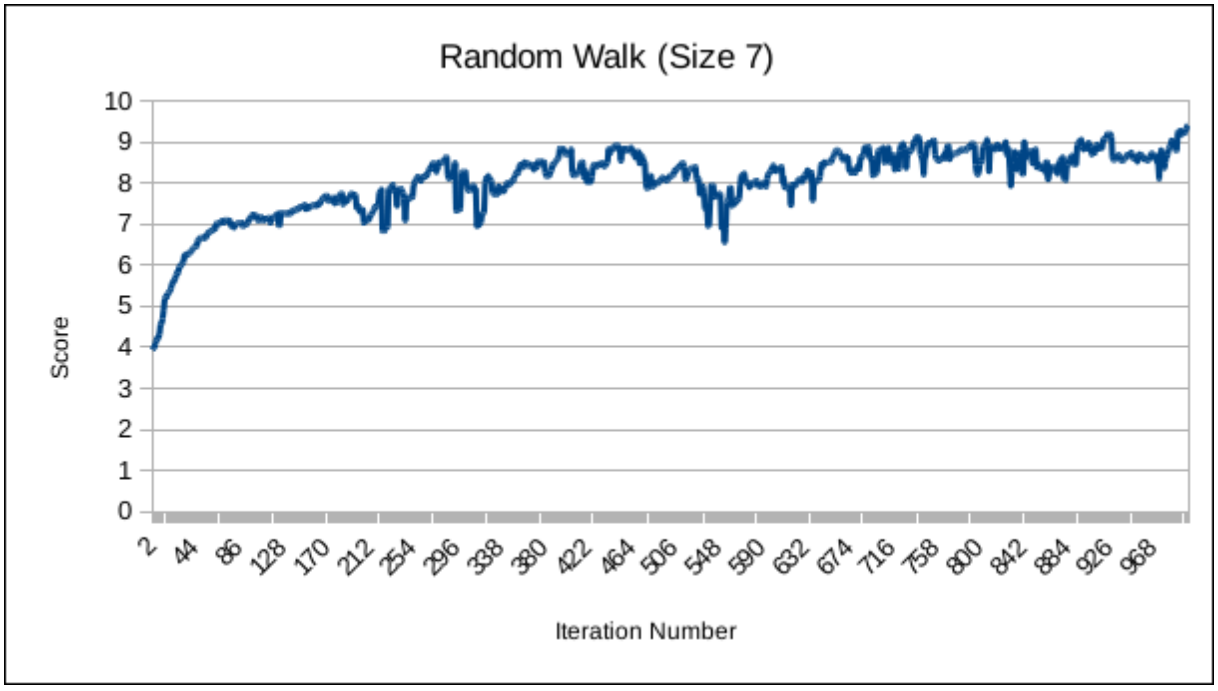
Task 5: Hill Climbing with Random Walks

Adding an element of random to the normal hill climbing algorithm allows for the program to make changes to the puzzle that will worsen the score. When starting the process, a probability is defined which tells the program to accept bad changes a certain percentage of the time. By introducing this kind of randomness it is possible to escape a local maximums. For the examples given, the probability of accepting a worse puzzle was 0.05.

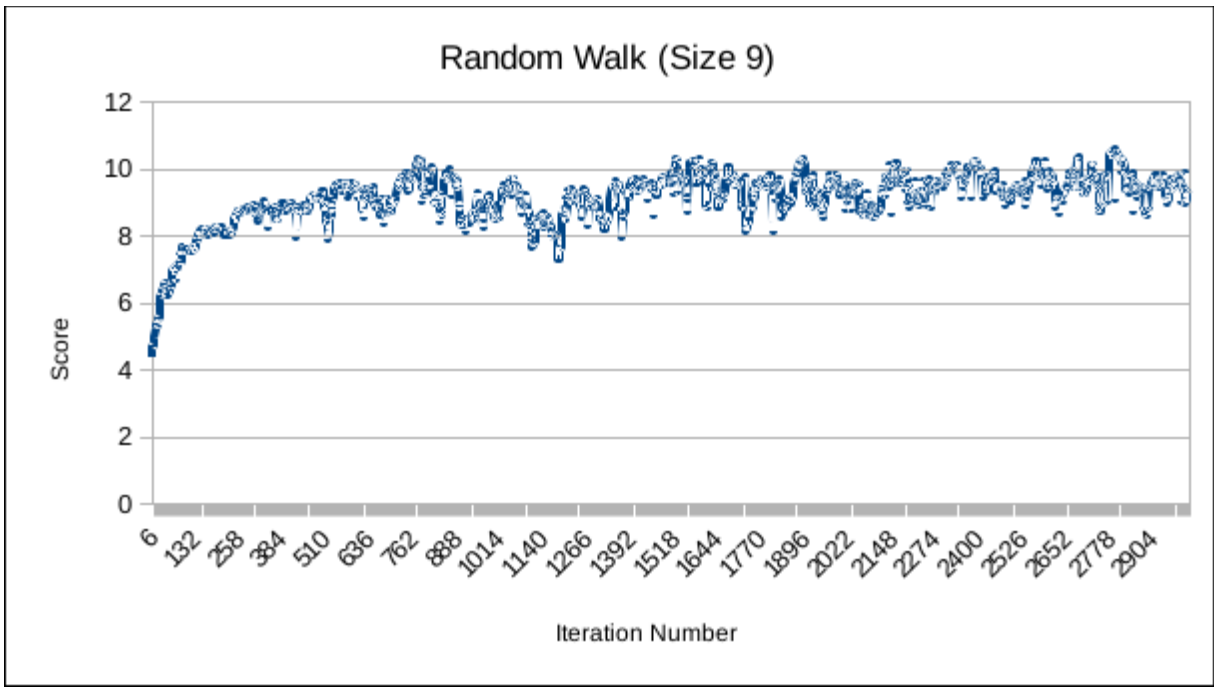
For n=5, hill climbing with random walks was run for a total of 1000 iterations. All of the first iterations were averaged, then all the second iterations, and so on to generate the following graph.



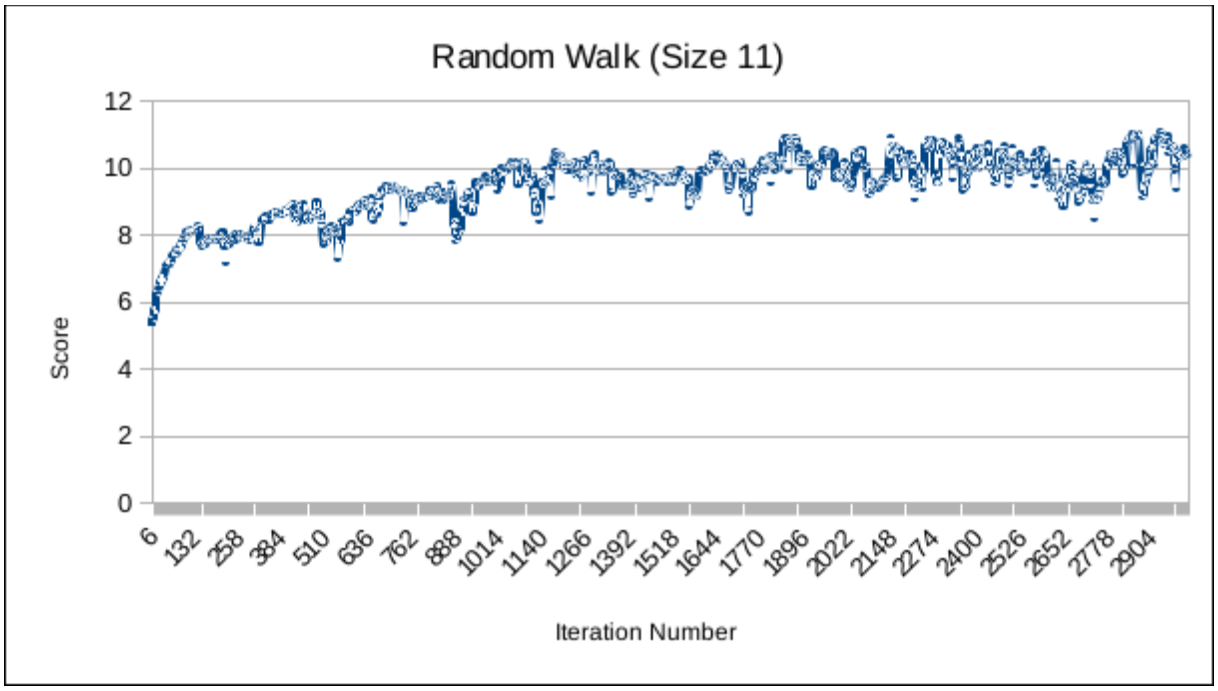
For n=7, hill climbing with random walks was run for a total of 1000 iterations. All of the first iterations were averaged, then all the second iterations, and so on to generate the following graph.



For n=9, hill climbing with random walks was run for a total of 3000 iterations. All of the first iterations were averaged, then all the second iterations, and so on to generate the following graph.



For n=11, hill climbing with random walks was run for a total of 3000 iterations. All of the first iterations were averaged, then all the second iterations, and so on to generate the following graph.

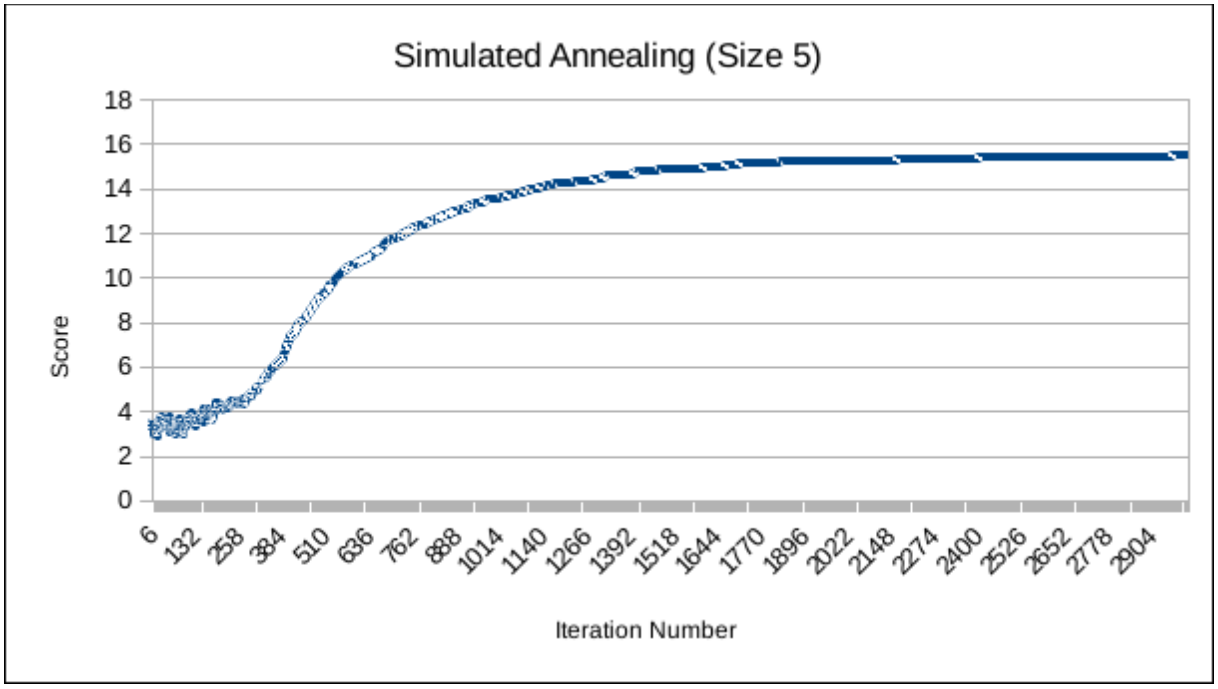


Task 6: Simulated Annealing

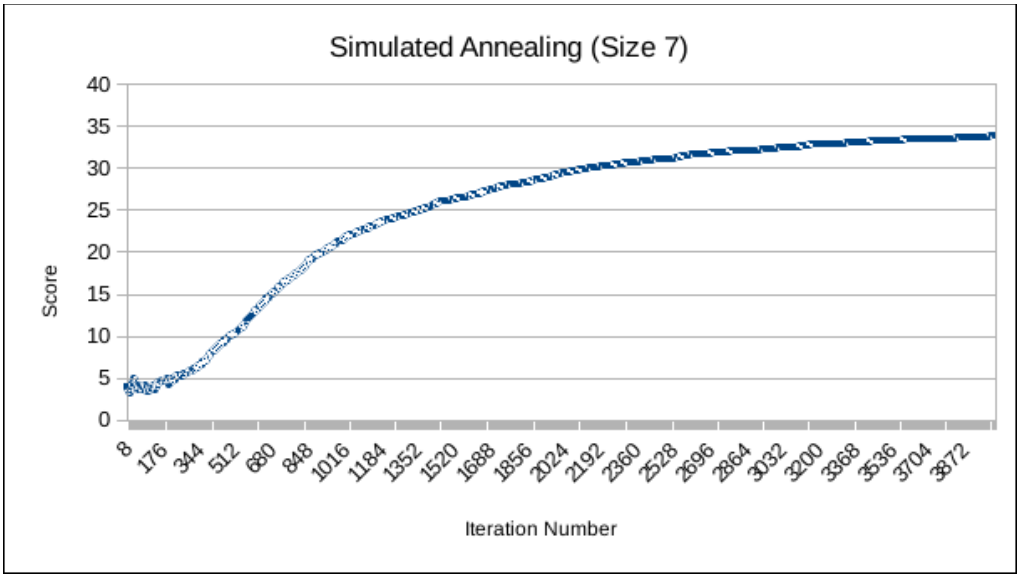
One way to take advantage of the best of both random restarts and random walks is to start with one and transition to the other. In the case of simulated annealing, there is a “temperature” that controls

the probability of randomly accepting a change. At the start of the program the temperature is hot which makes it more likely to accept a change which lowers the puzzle's score. This setup provided the best results all sizes of puzzles. In all of the graphs below, the initial temperature was set to 100 and the cool rate was set to 0.985 as these numbers produced decent results.

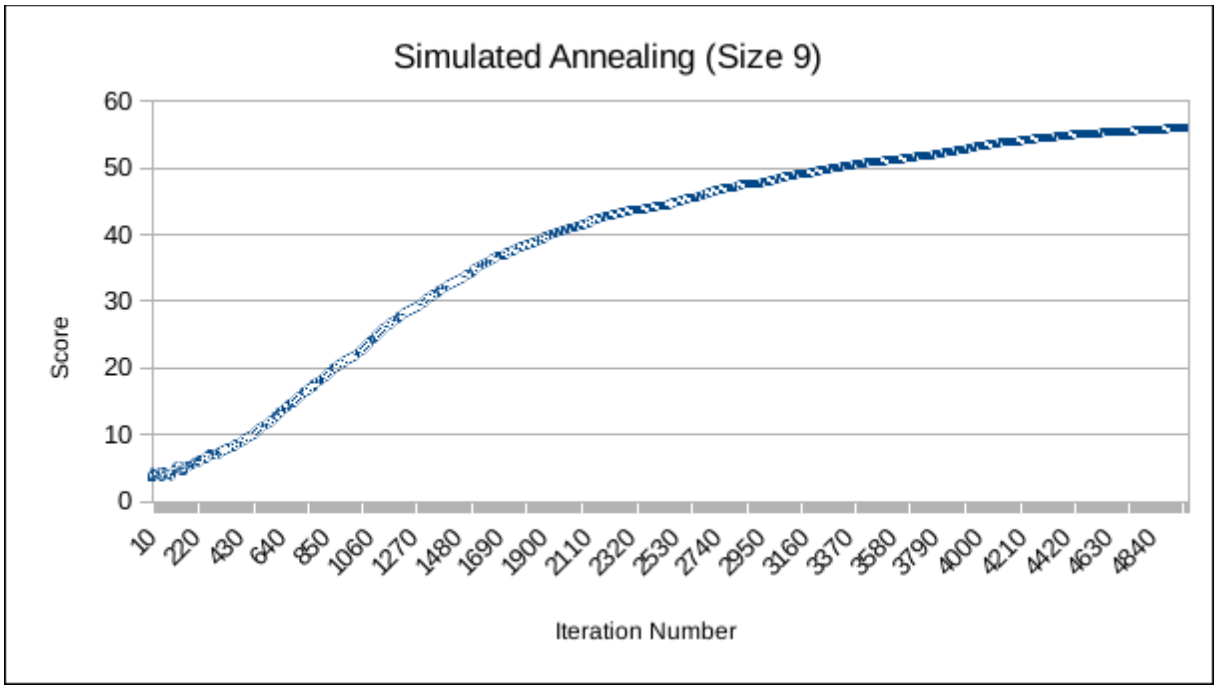
For n=5, simulated annealing was run for a total of 3000 iterations. All of the first iterations were averaged, then all the second iterations, and so on to generate the following graph.



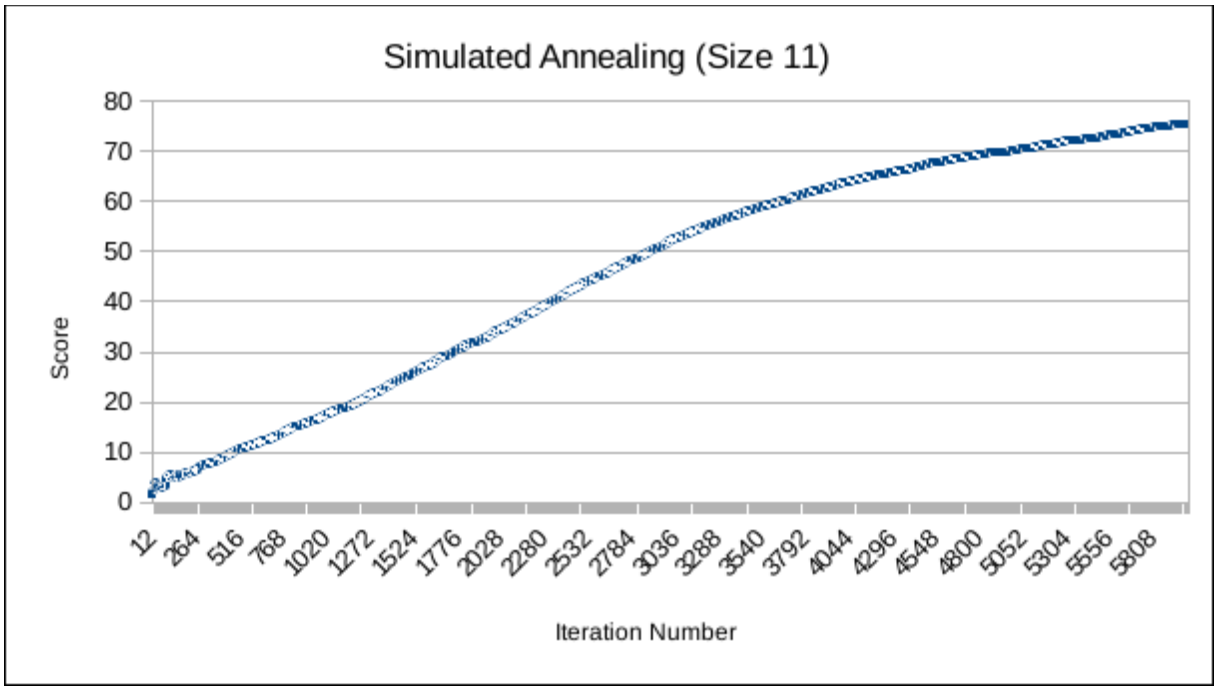
For n=7, simulated annealing was run for a total of 4000 iterations. All of the first iterations were averaged, then all the second iterations, and so on to generate the following graph.



For n=9, simulated annealing was run for a total of 5000 iterations. All of the first iterations were averaged, then all the second iterations, and so on to generate the following graph.



For n=11, simulated annealing was run for a total of 6000 iterations. All of the first iterations were averaged, then all the second iterations, and so on to generate the following graph.

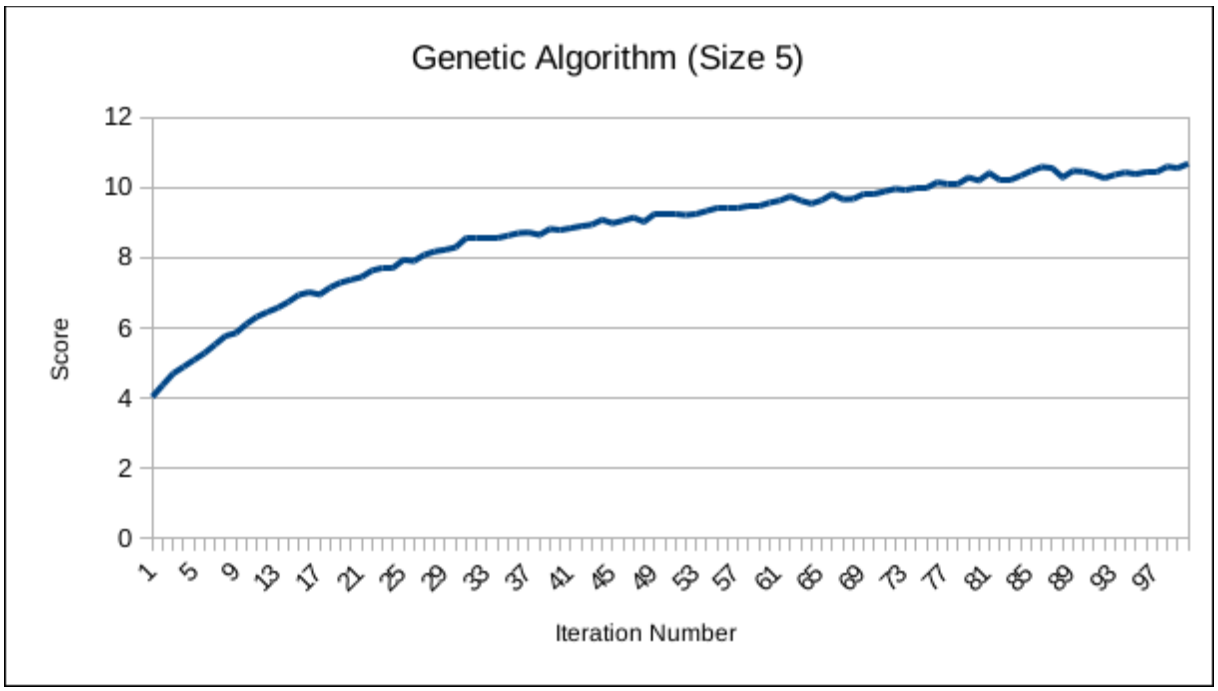


Task 7: Propose and implement a population-based approach

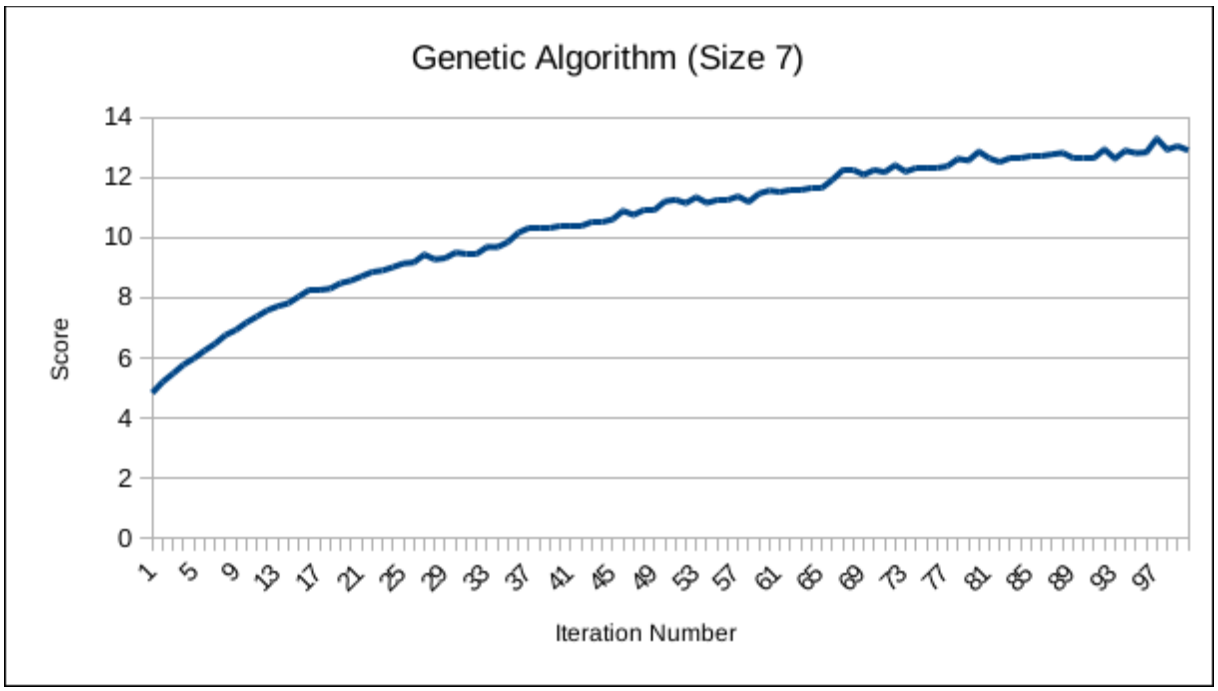
The population-based approach used in this program was better than the hill climbings, but not as good as the simulated annealing. At the start, there is a population of puzzles that are randomly generated and scored. After that, the the scores of individual puzzles are compared against the scores for all the puzzles of the population. The ones that are 80% better than the rest of the scores are selected to be crossed over to create the next generation. If none of the puzzles are substantially better, then two random ones will be selected. Breeding children puzzles for the next generation involves randomly choosing 2 of the selected members of the population with a 50 percent chance a child will instead be a clone of a selected puzzle. After the new population is created, they have a chance of mutating. For every cell in every puzzle of the next generation, there is a 2% that the move value will change to another valid option. This process is then repeated for however many generations are desired.

The graphs shown below are the result of fifty runs of the genetic algorithm. For each generation, the scores of the puzzles were averaged giving a rough average for the entire run. Each run was then averaged together to give the graph shown. The runs were made in comparable time to the simulated annealing runs. This means that the population size times the number of iterations for the genetic algorithm equals the number of iterations for the simulated annealing for the same size n. For all of these graphs, the number of generations per run was fixed at 100.

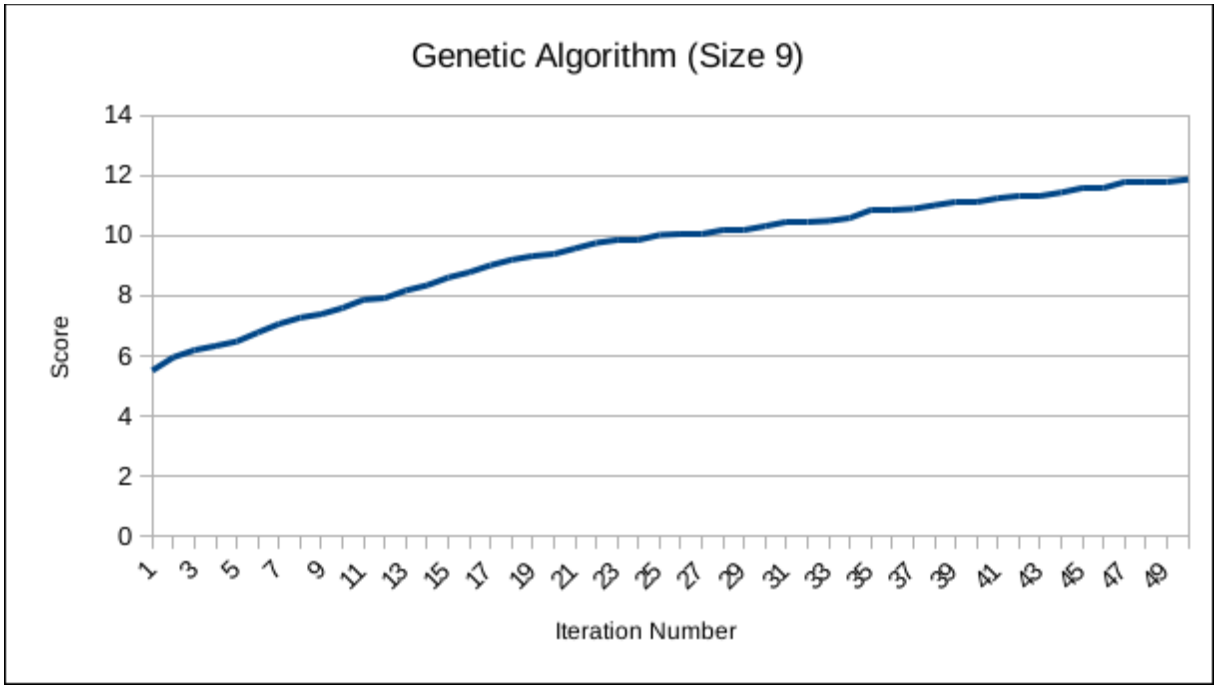
For n=5, genetic algorithm was run for a total of 3000 iterations. There was a population of 30 which went through 100 iterations. All of the members of each generation were averaged which were which were then averaged with all fifty runs. This resulted in the following graph.



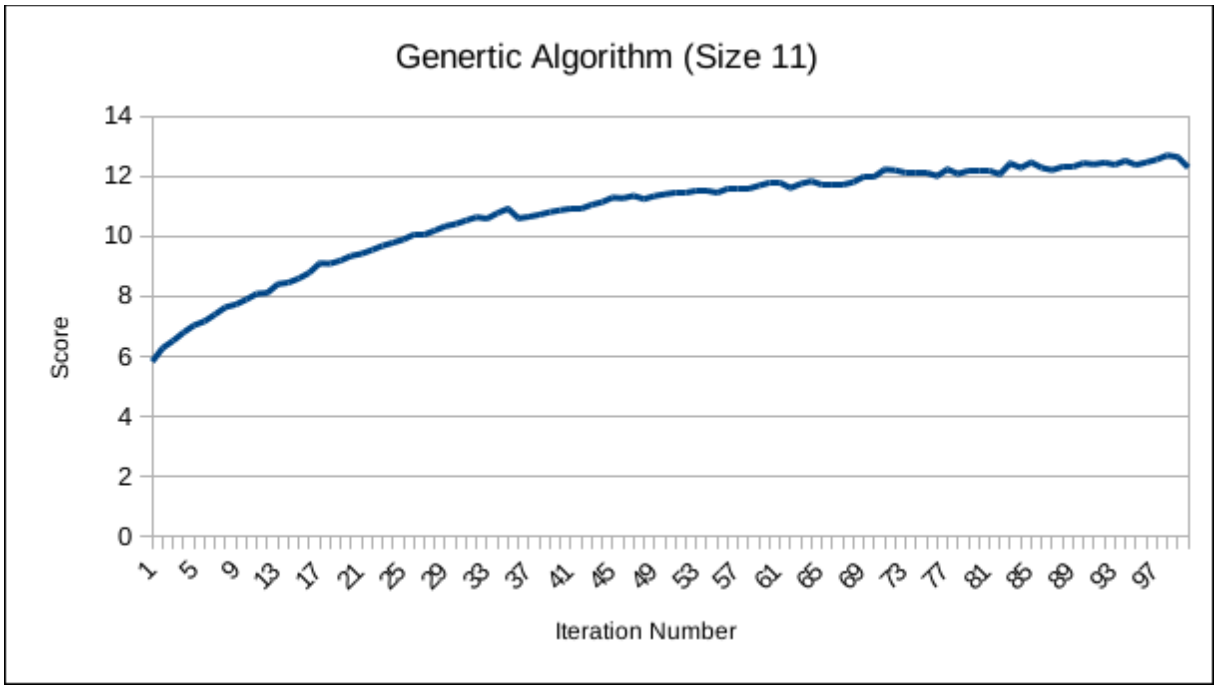
For n=7, genetic algorithm was run for a total of 4000 iterations. There was a population of 40 which went through 100 iterations. All of the members of each generation were averaged which were which were then averaged with all fifty runs. This resulted in the following graph.



For n=9, genetic algorithm was run for a total of 5000 iterations. There was a population of 50 which went through 100 iterations. All of the members of each generation were averaged which were which were then averaged with all fifty runs. This resulted in the following graph.



For n=11, genetic algorithm was run for a total of 6000 iterations. There was a population of 60 which went through 100 iterations. All of the members of each generation were averaged which were which were then averaged with all fifty runs. This resulted in the following graph.



Best puzzle generated: