Thomas Fiorilla, Anthony Tiongson

Professor McMahon

CS440 Intro to AI

February 22, 2020

Assignment 1 Search and Genetic Algorithms

Environment

An n-by-n grid puzzle, where a chosen value n can be greater than or equal to 5 or a randomly generated n value of 5, 7, 9, or 11. n represents the number of row and column positions on the puzzle. The row and column indices are $(r_{min}, \ldots, r_{max})$ and $(c_{min}, \ldots, c_{max})$, so for a puzzle where $n = 5$, $r_{min} = c_{min} = 1$ and $r_{max} = c_{max} = 5$. The starting position of the puzzle is the upper left most position $r_{min} \times c_{min}$, or 1×1, and the goal position is the lower right most position $r_{max} \times c_{max}$, or $n \times n$. There exists an integer value representing a legal move for every non-goal position, with the goal position designated with the letter G.

Example 1: A 5×5 puzzle generated from a manipulated random puzzle using a custom genetic algorithm. Solves in 16 moves.

3	2	2	3	2
2	2	3	2	3
2	2	1	2	2
4	3	3	2	4
3	4	1	3	G

Path to goal: (0, 0), (0, 3), (3, 3), (3, 1), (0, 1), (2, 1), (2, 3), (4, 3), (4, 0), (1, 0), (1, 2), (4, 2), (3, 2), (0, 2), (0, 4), (2, 4), (4, 4)

There is one agent tasked to evaluate the puzzle by attempting to solve it to derive a solvability score for it. The agent may then be tasked to manipulate the puzzle to increase its solvability score. The puzzle is static unless the agent is manipulating it, so the next state is determined by either the agent's position on the puzzle during an

evaluation or in the resulting manipulation of it. Every randomly generated puzzle for the agent to evaluate and possibly manipulate is independent from the last.

Thus, this is a fully observable deterministic episodic static environment with discrete states for a single agent.

State Space

A generated puzzle is represented by the two-dimensional matrix $boardBuilt[i][j]$ where i represents a position in the row range $r_{min} - 1 = 0$ to $r_{max} - 1 = n - 1$ and j represents a position in the column range $c_{min} - 1 = 0$ to $c_{max} - 1 = n - 1$. The values stored in the array $boardBuilt[i][j]$ each represent a legal move m at that particular i-th row and j-th position of the puzzle. In general, a legal move for a non-goal position ranges from 1 to the $max\{r_{max} - r, r - r_{min}, c_{max} - c, c - c_{min}\}$. The puzzle starting position is always at $boardBuilt[0][0]$ and its goal position is always at $boardBuilt[n - 1][n - 1]$; the goal position will always have a legal move value of 0 and a displayed value of G to represent the final desired destination.

An agent can evaluate the randomly generated $n \times n$ puzzle $boardBuilt[i][j]$ starting at position $(0, 0)$. Generally, when arriving at position (i, j), the agent stores the minimum number of moves required to reach it $depth$ in $steps[i][j]$, takes note that its visited it in $visited[i][j]$, and finds out its move value m. The agent has two different techniques to continue its analysis of the puzzle.

The agent can make an exhaustive analysis using a breadth-first-search of the possible moves from its current position and put every subsequent possible position and their $depth + 1$ level as $(i_2, j_2, steps[i][j] + 1)$ into a queue q and (i_2, j_2) into the set v to remember not to evaluate that position again. The agent will evaluate every position in q in FIFO order. Once the q is empty, the agent will derive a solvability score and assign it to the puzzle's $value$.

The agent can alternatively take a targeted, more efficient analysis using an A* search with an *estimate*, a *heuristic*, and a priority queue pQ. The *estimate* can be considered the function $f(x) = g(x) + heuristic$, where $g(x)$ is the number of moves the agent has made *depth* and the *heuristic* is an estimated guess of the remaining minimum amount of moves the agent has left to reach goal. The *heuristic* says that for the agent's current position (i, j) and its possible next position (i_2, j_2), if $i_2 = size - 1$ and $j_2 = size - 1$, the estimated amount of moves left to the goal is only 1. If $i_2 = size - 1$ or $j_2 = size - 1$, the estimated amount of moves left to the goal is at least 2. Otherwise, the estimated amount of moves to goal is at least 3. Using this *heuristic*, the agent calculates an *estimate* to be used with the next possible position and its $depth + 1$ level to put it in pQ as $(estimate, i_2, j_2, self.steps[i][j] + 1)$. A lower *estimate* will give a position (i_2, j_2) a higher priority in the queue pQ, so the agent will evaluate positions estimated to be closer to goal ahead of those with higher estimates. Unlike the breadth-first-search, the A* search will end the agent's traversal once it arrives to the goal position, and the agent will derive a positive solvability score m at this moment. Otherwise, the agent will derive a $-k$ score when all possible moves are exhausted.

During either techniques, the agent will use the dictionary $prevPos[(i_2, j_2)] = (i , j)$ to keep track of a possible path to the goal position. If the puzzle is solvable, the agent will populate a list *shortestPath* to store and display the puzzle's solution.

After an initial evaluation of the puzzle, the agent can then be tasked to manipulate it in attempt to improve its solvability score. The agent may take a basic hill-climbing approach, randomly choosing a non-goal position (i_r, j_r) and assigning a random valid move to that position. The agent will then compare that randomly manipulated puzzle with the original by evaluating it, and if the new puzzle's solvability score *value* is greater than or equal to the original it will replace the original with it. If not, the agent will reject the new puzzle for the original. The agent may run this manipulation for many iterations.

The agent may also manipulate the puzzle using a more targeted, less random approach with the hill-climbing as its basis. If the agent evaluates a randomly generated puzzle as unsolvable, the agent will take the hill-climbing approach but also make sure that whatever random valid move it chooses for position (i_r, j_r) will be different than the one already assigned to it. If the agent is manipulating a solvable puzzle, it will first evaluate the *shortestPath* with both the BFS and A* search traversals to compare and see if there are at least two different shortest paths. If so, the agent will investigate a random position (i_r, j_r) from the BFS *shortestPath* and a random position (i_{ra}, j_{ra}) from the A* search *shortestPath*, making sure $(i_r, j_r) \neq (i_{ra}, j_{ra})$. The agent will randomly choose a valid move for both positions different from their respective existing values. The agent will then create and evaluate a puzzle board that represents a mutation where both shortest paths are manipulated, another board where only the A* search *shortestPath* is modified, and lastly a puzzle where only the BFS *shortestPath* is changed. All three boards are evaluated and compared against each other by the agent. The puzzle with the highest solvability score *value* is then compared against the original to see if it should be accepted or rejected. In the other case where there is only one *shortestPath* evaluated by the agent, the agent will similarly target that *shortestPath* and make an evaluative comparison with another puzzle manipulated with the less random hill-climbing algorithm. Once again, the accepted puzzle will be compared with the original board to validate any improvements. The agent may run this manipulation for many iterations.

If $\mathbf{S}$ is the set of all states, then $\mathbf{s}$ where $\mathbf{s} \in \mathbf{S}$ is any state representing the combination of the array $boardBuilt[i][j]$, the instance of the agent's possible evaluation of $boardBuilt[i][j]$, and the instance of the agent's possible mutation of $boardBuilt[i][j]$ at that given moment.

Actions

If $\mathbf{A}$ is the set of all possible actions, then $\mathbf{a}$ where $\mathbf{a} \in \mathbf{A}$ is any action an agent can make to evaluate the puzzle, like traversing it, or to mutate the puzzle, like applying a hill-climbing algorithm to it.

Perception/Observations

The input the agent receives is the puzzle in the form of an array $boardBuilt[i][j]$. If $\mathbf{O}$ is the set of all possible observations for a given puzzle, then $\mathbf{o}$ where $\mathbf{o} \in \mathbf{O}$ is for that given moment any observable conclusions an agent can derive during evaluation or any observable change an agent can make to the puzzle's solvability score $value$ during manipulation.

Transition Function

Any agent action $\mathbf{a}$ during any state $\mathbf{s}$ for a transition function τ gives a new state $\mathbf{s}'$, or $\mathbf{s}' = \tau(\mathbf{s}, \mathbf{a})$. During an agent's evaluation, the next state $\mathbf{s}'$ depends on any legal moves $\mathbf{a}$ the agent has and which search analysis its using. Once an action $\mathbf{a}$ is chosen, the current position of the agent and its current available legal moves are updated to reflect the new state $\mathbf{s}'$. During an agent's manipulation of a puzzle, the next state $\mathbf{s}'$ depends on which algorithm $\mathbf{a}$ it uses to mutate the puzzle and whether or not the subsequent puzzle has a higher solvability score.

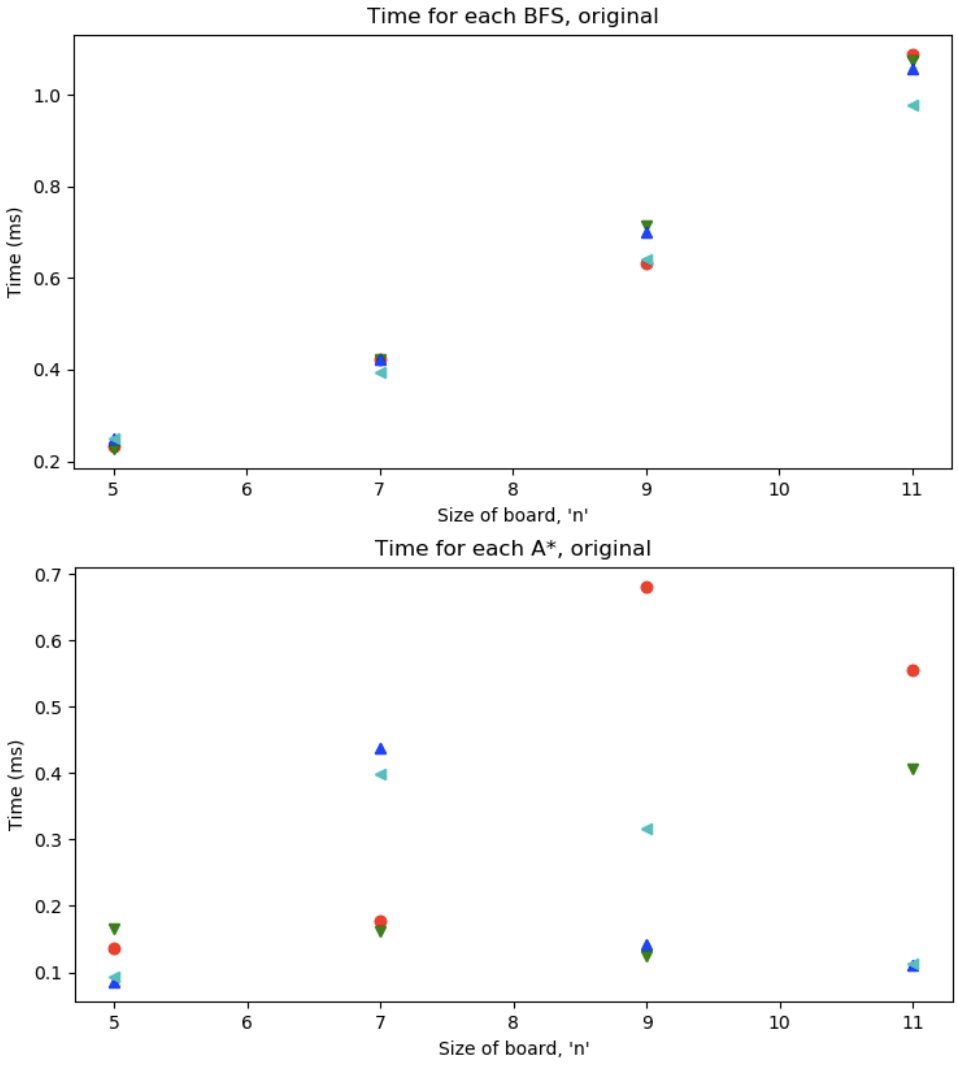
Evaluation Metric

After an agent evaluates a puzzle, it will assign a solvability score $value$ to it. Upon completion of either search techniques, the agent uses $steps[size - 1][size - 1]$ or $visited[i][j]$ to derive a solvability score m or $-k$ for the puzzle's $value$, respectively. If the puzzle is solvable, the agent will set the $value$ of the puzzle to the minimum m number of moves it takes to solve the puzzle. A higher value m is defined as a more difficult puzzle. If the puzzle is unsolvable, then the agent will set the value of the puzzle as $-k$, where k is the number of positions not reachable from the start.

The agent will use this evaluation metric to govern its decisions if tasked to manipulate the puzzle. If a resulting mutated puzzle is evaluated to have a solvability score greater than its parent, the agent will accept that puzzle as an improved replacement. If not, the agent will reject the resulting puzzle and keep the original.

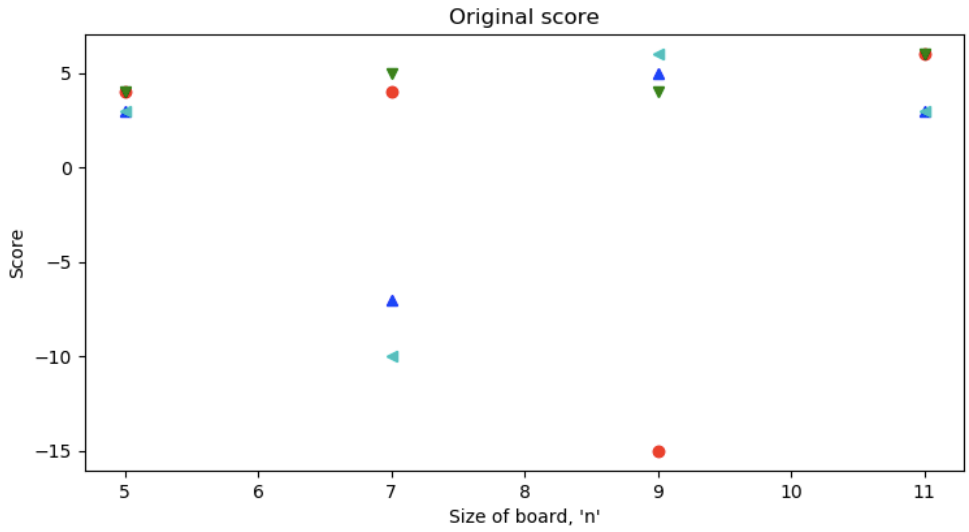
Final Evaluation

Four puzzles are generated for each sizes 5, 7, 9, and 11. In all, there are sixteen random puzzles of varying solvability scores. Each puzzle is evaluated by the agent with a BFS and A* search analysis before and after 50 iterations of the basic hill-climbing algorithm. The agent repeats the evaluation with the more elaborately designed genetic algorithm for 50 iterations from the original starting puzzles.

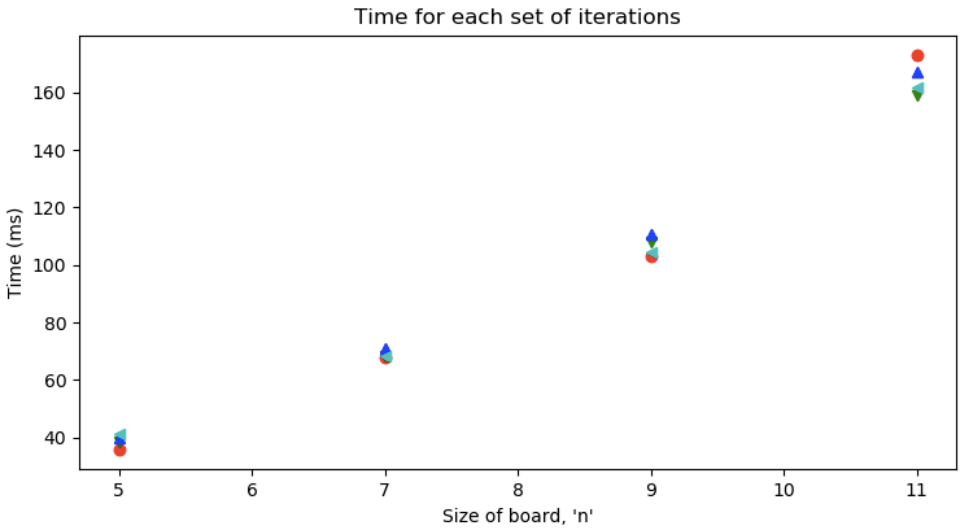


Each color/shape represents one of the four randomly generated puzzles.

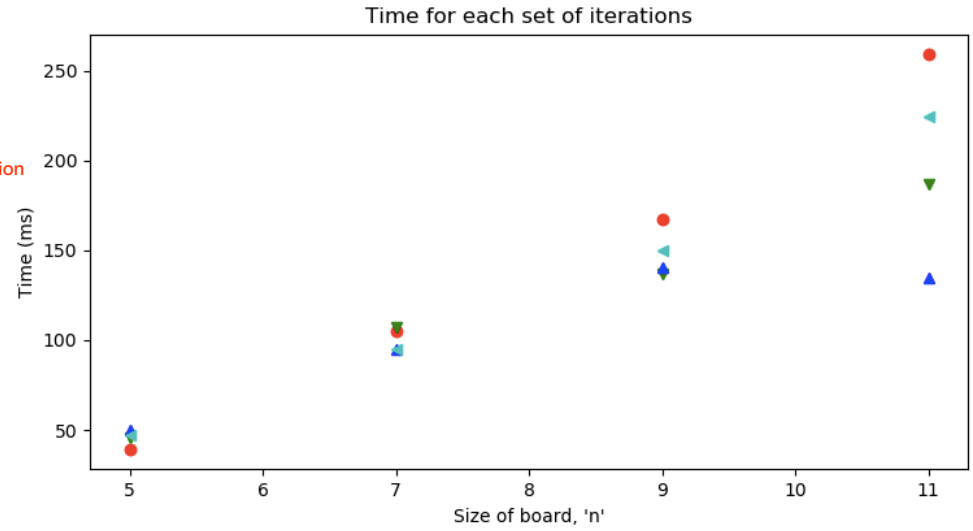
From the original starting puzzles, the agent on average evaluates the puzzles faster with the A* search. This is expected since the A* search is not exhaustive by design like the BFS. The BFS, however, gives a complete analysis of the board by showing the entire move, or *depth*, level for all reachable positions. Because of this, the BFS is capable of showing an alternative *shortestPath* not recognized by the A* search. The A* search's advantage in speed can be observed at larger boards where it can effectively evaluate a board by a factor of 10 compared to the BFS. This speed advantage does not occur for unsolvable puzzles, since the A* search will only terminate once a *shortestPath* is found. For initial evaluation of puzzles, the A* is ideal whereas the BFS is useful for manipulated or high solvability score puzzles to find multiple shortest paths.



Solvability score results from agent analysis of the randomly generated puzzles.
Note the red 9×9 -15 puzzle and its respective evaluations speeds from the graphs above.

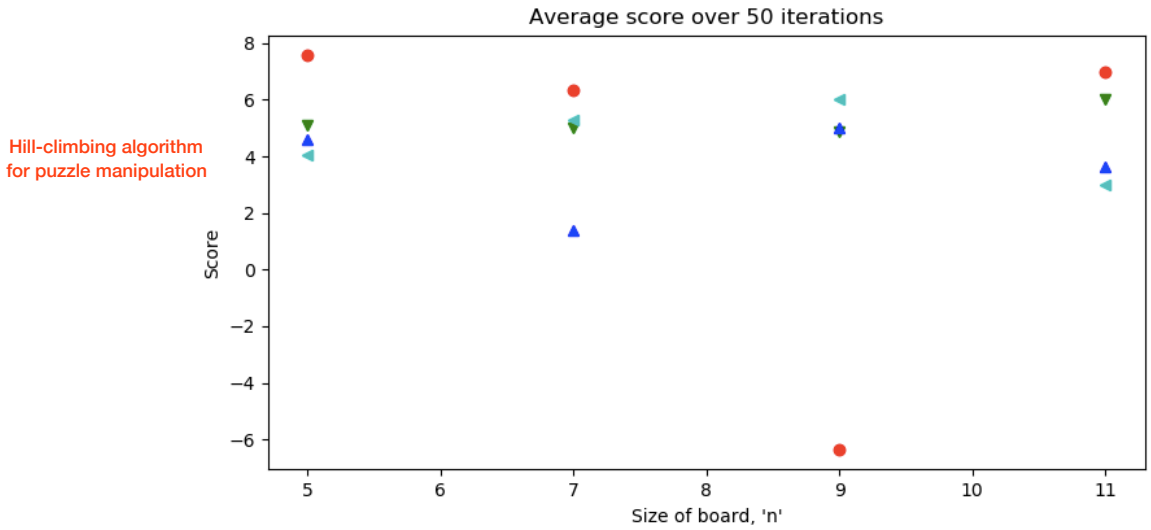


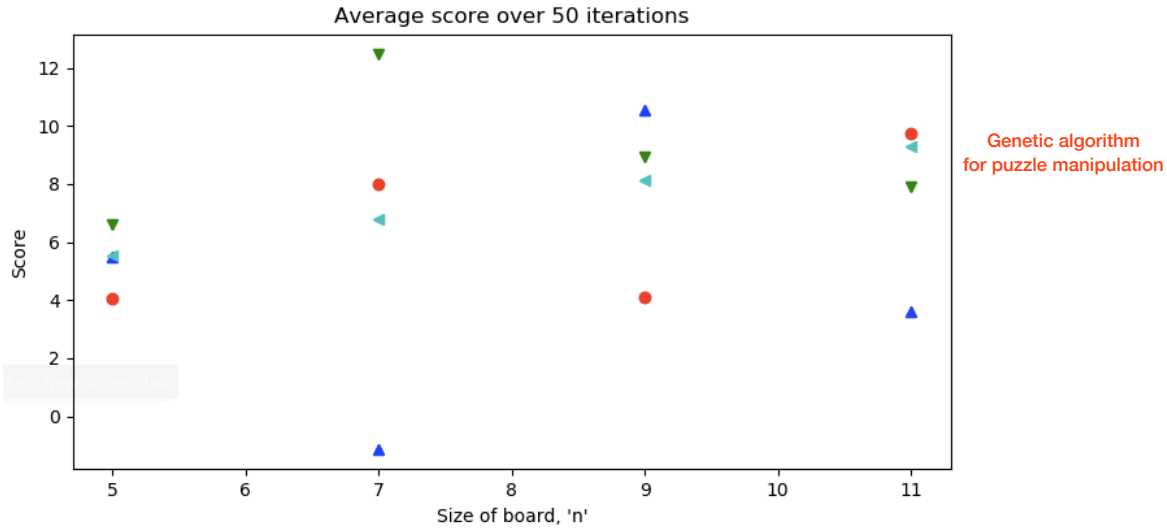
Hill-climbing algorithm for puzzle manipulation



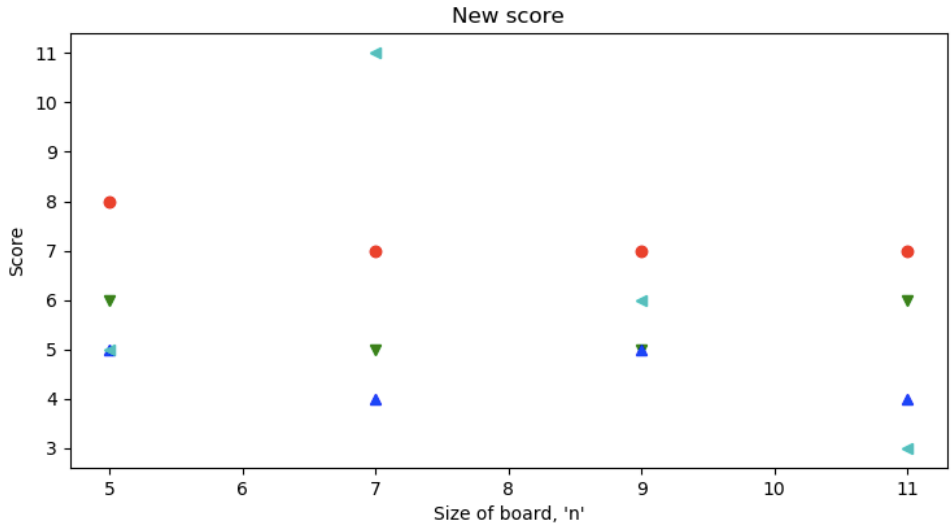
Genetic algorithm for puzzle manipulation

As expected, the agent performs the simple hill-climbing algorithm faster than the more complex genetic algorithm on average. This is not surprising since the genetic algorithm is built upon, and even uses, the hill-climbing algorithm. The difference in speed gets more profound as the size of the puzzle increases.

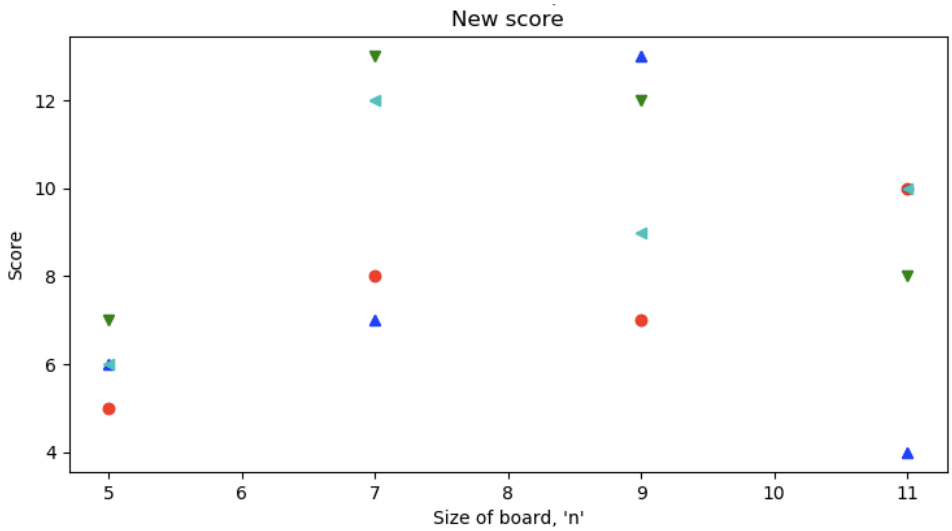




The average score over 50 iterations between the two algorithms shows that the genetic algorithm is capable of reaching a higher solvability score than the hill-climbing, however it is not a guarantee that it will. In some instances, the hill-climbing algorithm is capable of outperforming the genetic algorithm within the window of iterations. Although much simpler in design, the hill-climbing algorithm is still effective.



Final hill-climbing algorithm solvability score after 50 iterations.



Final genetic algorithm solvability score after 50 iterations.

Because of its less random and more directed approach, the genetic algorithm was able to effectively raise the solvability score of each puzzle higher than the hill-climbing algorithm on average. It is noteworthy that these differences are more noticeable for larger puzzles. Smaller puzzles seems to respond as effectively to the hill-climbing algorithm.

Puzzle Examples

4	3	3	4	2
2	3	2	2	4
2	3	1	3	4
3	3	3	3	1
2	4	4	3	G

Example 2: A 5×5 puzzle generated from a manipulated random puzzle using a custom genetic algorithm. Solves in 16 moves.

Path to goal: (0, 0), (4, 0), (2, 0), (2, 2), (1, 2), (1, 0), (3, 0), (3, 3), (0, 3), (4, 3), (1, 3), (1, 1), (4, 1), (0, 1), (3, 1), (3, 4), (4, 4)

11	5	4	9	19	4	17	18	10	9	7	18	3	19	3	6	17	18	20	12	17
1	16	8	11	2	9	16	4	19	9	12	17	7	3	3	6	7	8	2	10	6
10	19	7	14	7	14	4	14	10	18	3	7	2	9	14	14	9	1	16	9	19
11	1	15	14	14	11	2	13	14	5	16	12	12	12	1	9	9	8	3	13	12
1	3	11	9	13	1	1	2	13	14	14	5	14	15	4	12	8	7	14	17	4
8	5	6	3	2	14	1	5	11	10	15	5	10	5	10	2	5	12	6	5	16
5	5	10	2	2	6	12	7	12	12	12	7	9	7	5	4	13	14	4	4	6
8	9	6	8	8	1	11	2	4	4	5	6	8	2	6	3	5	8	12	17	9
13	18	15	15	5	5	8	11	2	11	11	3	9	1	1	9	14	15	2	18	18
5	11	17	5	4	15	11	9	12	1	6	9	4	6	14	3	6	12	8	9	14
3	17	2	10	3	13	9	11	8	11	1	8	10	7	8	2	10	14	3	10	12
12	13	7	8	11	3	13	12	3	8	11	11	6	6	12	11	8	11	11	17	4
15	6	5	13	4	10	3	13	1	6	8	5	10	4	11	3	3	17	6	17	16
12	13	2	2	3	6	1	4	10	13	5	12	3	7	9	11	3	8	18	3	7
6	11	6	15	7	13	2	6	12	4	14	2	5	14	4	4	10	14	2	13	12
20	11	16	17	4	1	9	8	14	13	11	1	10	10	13	8	11	11	5	14	17
10	17	10	1	1	6	3	15	1	15	6	8	14	6	8	6	10	2	15	11	20
6	12	7	6	11	4	7	1	12	9	11	6	11	2	9	1	16	8	11	17	5
1	17	4	1	7	11	4	4	2	11	1	17	9	14	4	5	11	15	5	17	3
7	19	4	6	14	3	12	12	19	14	4	3	11	4	16	12	6	9	8	15	14
17	6	17	8	13	9	1	1	11	3	17	10	4	9	20	19	6	13	10	17	G

Example 3: A 21×21 puzzle generated from a manipulated random puzzle using a custom genetic algorithm. Solves in 20 moves.

Path to goal: (0, 0), (11, 0), (11, 12), (17, 12), (17, 1), (17, 13), (17, 15), (17, 14), (8, 14), (7, 14), (7, 20), (16, 20), (16, 0), (16, 10), (16, 16), (6, 16), (19, 16), (13, 16), (13, 13), (13, 20), (20, 20)

5	10	10	2	7	1	6	1	10	5	2
7	6	5	5	6	4	8	2	7	4	10
2	6	3	6	6	7	1	5	4	8	1
10	9	8	6	4	5	2	3	8	6	9
10	2	5	6	4	4	4	2	8	7	9
7	4	6	7	3	5	2	5	7	5	6
6	9	7	3	3	3	6	7	5	2	9
9	6	2	5	6	4	5	7	3	8	4
5	9	3	6	3	1	7	2	6	6	1
8	4	9	4	6	4	4	1	5	4	9
5	9	5	10	10	3	2	8	8	2	G

Example 4: A 11×11 puzzle generated from a manipulated random puzzle using a custom genetic algorithm. Solves in 21 moves, 2 paths.

Path to goal: (0, 0), (5, 0), (5, 7), (0, 7), (0, 8), (10, 8), (2, 8), (6, 8), (6, 3), (9, 3), (9, 7), (9, 8), (4, 8), (4, 0), (4, 10), (4, 1), (4, 3), (10, 3), (0, 3), (0, 1), (10, 1), (10, 10)

Alt: (0, 0), (0, 5), (1, 5), (5, 5), (10, 5), (10, 8), (2, 8), (6, 8), (6, 3), (9, 3), (9, 7), (9, 8), (4, 8), (4, 0), (4, 10), (4, 1), (4, 3), (10, 3), (0, 3), (0, 1), (10, 1), (10, 10)