American University of Armenia

Solving Rubik’s Race

Marina Igitkhanian

Nina Lazaryan

Yeva Manukyan

Mariam Yayloyan

December 9, 2023

Abstract

The Rubik's Race game was created by Ernő Rubik, the inventor of the famous Rubik's Cube. Although those two games may seem similar at first glance, they are pretty different. Rubik's Race may be considered an alternate version of the N-puzzle problem, which is already solved with various artificial intelligence algorithms. In the project, techniques that were applied to the N-puzzle problem will be implemented to solve the Rubik's Race. Informed and uninformed algorithms with different heuristics and hill climbing are going to be considered. The performance of those methods will be compared empirically. Through this research, our aim is to find the most effective approach for winning the Rubik's Race game, if it exists.

Contents

1 Introduction

1.1 Problem Setting and Description

Rubik's Race was created by Ernő Rubik 8 years after the invention of the viral Rubik's Cube in 1982. It didn't become as popular as the cube, but it still was played by many board game fans and is known to this day. The basic idea is that a player must repeat some given pattern of colors from the scrambler on their board.

Originally, the Rubik's Race was supposed to be played by two people, meaning a multi-agent (competitive) environment. However, players' actions are independent of each other since, to win the game, a player needs to get the desired pattern of colors before the opponent. Taking into account the independence of players' actions, in this paper, we are reformulating the game, assuming only one person participates in it.

Also, by people, the 5x5 grid version is normally played, and a 3x3 centered square must satisfy the pattern given on the scrambler. For the sake of scalability in this paper, the general case for the n x n grid is considered.

1.2 Rules of the Game

In the modified version of the Rubik's Race game, the rules are as follows:
There is an n x n (where $n > 2$) board with $n^2 - 1$ number of colored tiles and one empty space. The number of tiles of each color should be equal. As there are six different colors of the tiles, there should be $(n^2-1)/6$ number of tiles from each color (where $(n^2-1)/6$ is a natural number).

A scrambler is used within the game to generate the goal state randomly. The scrambler is of size (n-2) x (n-2) with $(n-2)^2$ colored tiles, and there is no more than $(n^2-1)/6$ of each color tile in it. We aim to achieve a state in our n x n board where the central $(n-2)^2$ tiles mirror the generated goal state. The actions we can apply on each tile are Left, Right, Up, and Down. It is important to note that these actions can only be applied if there is an adjacent empty space, and they should be performed to interchange the positions of the

empty space and the targeted tile. The initial state of the game is random; however, there are such states that are unsolvable, and we are going to consider only the initial states that are solvable.

1.3 Similarities with the N-puzzle problem

In fact, for the AI community, this game is not popular, meaning there is very little research done to test different AI techniques for solving the problem of Rubik's Race.

However, we can see there are some similarities between Rubik's Race modified version and the well-known N-Puzzle game. Let us briefly define the rules of the N-Puzzle game. The game consists of an n x n board with (n-1) number of tiles and an empty space. The tiles are numbered from 1 to (n-1). The aim is to achieve the goal state, which is the state where the tiles are ordered in row-major order, by applying actions Left, Right, Up, and Down on each tile that has an empty space next to it and swapping the locations of the tile and the empty space.

Compared to the N puzzle, where we only have one fixed goal state, Rubik's Race has multiple goal states, as the colors of the tiles outside the central $(n-2)^2$ tiles are irrelevant. Moreover, in the Rubik's Race, the goal state is randomized with the Scramble. Another difference is the tiles. While in the N puzzle, the tiles are uniquely numbered, in Rubik's Race, we have six colors of tiles. In our goal state, we do not care which X-colored tile (X being one of the six colors) is in the central part. Apart from those differences, the game setting is very similar. The actions and the results of the actions are identical for both games. In both games, we are trying to get a given configuration of tiles using the same actions.

1.4 The Structure of the Project Paper

The project will consist of the following parts: introduction, literature review, methodology, implementation of the algorithms, results, and further work and discussion. In the introduction, the history and the rules of the game are stated, and another problem - N-puzzle, which is similar to Rubik's racing but more popular, is introduced. The literature

review part covers the algorithms that are applied for solving N-puzzle problems that are chosen to be applied for Rubik's racing. The methodology part covers how the known algorithms for N-Puzzle are implemented for our problem. The results part contains empirical data obtained from running algorithms and their analysis.

2 Literature Review

2.1 Solutions to a similar problem

For some reasons, one of which can be the game's unpopularity, very few papers are published on Rubik's Race solving algorithms. To solve this shortage of literature, we will be looking at works done on N-puzzle only. As mentioned, these two games have a lot of similarities; hence, the algorithms used to solve the N-puzzle can also be applied to our Rubik's Race.

2.1.1 Solvability of initial states

In the N-Puzzle problem, there are n +1 tiles, which can result in (n+1)! initial configurations. Out of these, only half of the configurations are solvable. So, only (n+1)! /2 Initial states can lead to the goal state in a limited number of moves. [1]

How to check if the initial state is solvable:

Step 1: Shift the blank tile at the bottom right corner of the grid.

Step 2: Calculate permutation inversion for each tile. Permutation inversion of a tile is the number of tiles with numbers less than the number on the tile.

Step 3: Calculate the sum of inversions for all the tiles; if it's even, then the initial state is solvable; otherwise, it's not solvable.

2.1.2 Uninformed algorithms

Breadth-First Search

As an uninformed algorithm, Breadth-First Search (BFS) begins without prior knowledge of the state space. The process starts at the root node, representing the initial state, and systematically explores all successors of the root node. Employing a first-in-first-out (FIFO) queue, BFS begins expanding successors from the left side. It will stop once it reaches the goal state, ensuring that the discovered path to the goal node is the shortest. The algorithm is complete in the case of using it for solving an N-puzzle problem. Despite finding the optimal solution, the algorithm tends to execute more nodes/states than necessary, making it inefficient for solving the N-puzzle problem. [2]

Depth First Search

Unlike the BFS algorithm, the Depth-First Search (DFS) algorithm initially explores all leftmost successors. It uses a first-in-last-out (FILO) queue or a stack. The algorithm prompts a backtracking process once it reaches a dead end unless the goal is found. DFS is more likely to enter an infinite loop as long as a depth limit is not imposed. Moreover, the DFS algorithm is not complete.

Since both of these algorithms are not efficient in the case of solving the N-puzzle game, which is similar to our modified version of Rubik's Race, we will not implement them to solve our game. [2]

2.1.3 Informed algorithms

The next class of techniques implemented for solving the N-puzzle are informed algorithms. For the particular problem, the most commonly used one is A*. In addition, some researchers considered IDA*. The difference between those and the previous ones is that heuristics functions are applied for informed algorithms, which must make the solution to the

problem more effective. This project considers the cases when A* and IDA* are implemented with the following heuristic functions: the number of misplaced tiles, the Manhattan distance, and linear conflict heuristics. [3] Before analysis of the implementation of those two algorithms for solving the N-puzzle problem, the heuristic functions must be introduced.

For a given state, the number of misplaced tiles is the sum of the tiles appearing in the wrong position - not where they appear in the goal state.

Manhattan distance is one of the popular methods used as a heuristic function when it comes to solving problems with the A* algorithm. For the N-puzzle, it is the linear distance that a tile needs to traverse from its initial position to reach the goal position. The heuristic function is the sum of each tile's Manhattan distance.

The linear conflict heuristic function is derived from the Manhattan distance by adding a new term: Two tiles, j(t□) and k(t□), are said to be in a linear conflict if they both appear in the same line (either in a row or a column) and their goal positions are in that same line as well. The heuristic function for a given state s is the number of all linear conflicts multiplied by two and the Manhattan distance:
H(s) = M(s) + 2*N(s) where M(s) is the Manhattan distance heuristic, and N(s) is the number of linear conflicts for a given state s.

A star (A*) Algorithm

One of the best-performing search algorithms for the N puzzle is the A* algorithm. A* is an informed search algorithm that uses heuristics to get more insights into how close a node is to the goal. It combines the logic of the Greedy Best First Search, which relies on heuristic value only, and the Uniform Cost Search, which depends on the path cost only. Hence, A* the evaluation function f(n) = h(n) + g(n), where g(n) is the path cost from the initial state to the node n, and h(n) is the heuristic value associated with the n node. The f(n) value indicates the closeness to the goal state, so A* favors the nodes with the least f(n) values.

A* is the fastest algorithm. It expends fewer nodes than any other search algorithm with a given heuristic. The heuristics mentioned above, number of misplaced tiles, Manhattan distance, and linear conflict heuristic, are admissible, meaning that the solution found by A* will always be optimal. It is important to mention that from an experiment done by Debasish Nayak (2014), where the mentioned heuristics were tested using an A* search on N-puzzle, the Linear Conflict heuristic dominated the Manhattan Distance heuristic and the number of misplaced tiles, having average path lengths 21.9. [3] As a result, it is a more accurate heuristic function than ManhattanDistance, which is a desirable property for solving the problem. Also ManhattanDistance dominates the number of misplaced tiles.

With all these advantages of the A* algorithm, its space complexity is very large.

Iterative Deepening A* algorithm (IDA*)

Another informed search algorithm broadly used for the N-puzzle problem is IDA* (Iterative Deepening A star). With a similar logic to the Iterative Deepening Depth-First search, where the search depth is increased with each iteration, IDA* also uses the advantage of computing the closeness to the goal state of the A* search algorithm. IDA* uses the same f(n) function defined for the A* search, called F-score. With each iteration, the minimum allowed F- score or the threshold becomes greater.

Like A* IDA* finds the optimal solution if the heuristic function is admissible. So, with the Manhattan distance and linear conflict heuristic, and number of misplaced tiles IDA* returns the optimal solution. Unlike A*, which uses a large amount of memory, IDA* uses less compared to A* [4]. However, IDA* may be slower because it expands the same nodes repeatedly.

An interesting study was done by Alexander Reinefeld, where he tested a different variation of IDA* on the N-puzzle. In the experiment, he tried different ways of choosing a node to expand and found that for the N-puzzle, random selection works the best compared to fixed node selection. [5]

2.1.4 Local Search

Another approach that can be considered in solving the N-puzzle is local search. Both hill climbing and genetic algorithms can be used to solve the N-puzzle problem.

Hill climbing

Hill climbing is an informed search algorithm that uses heuristic functions to perform the best step at the given moment. The algorithm starts with an initial state and makes changes step by step to it in order to get closer to the goal state. These changes are based on a heuristic function that evaluates the quality of the solution. The algorithm keeps making small changes in each step until no further improvement can be made. The same heuristics as mentioned above (Manhattan distance, number of misplaced tiles) can be used for the hill climbing method; however, there is a high possibility of the algorithm getting stuck in local minima.

Genetic Algorithm

Another approach to using local search is using genetic algorithms to reach the goal state. Genetic algorithms solve the problem by mimicking natural selection. They use variation-inducing operators such as mutation and crossover to implement variation. The fitness function is used to evaluate the fitness of a chromosome and to perform reproduction. 1) The algorithm randomly generates an initial population, 2) computes the fitness score of each individual in the current population, 3) selects two members from the current population based on their fitness score, 4) generates new chromosome using genetic operators, repeats step 3) until a new generation is completed, repeats step 2) until the goal state is found. The algorithm performs better in terms of time complexity.

In this particular case, the genetic algorithm is applied in the following way. It first generates a population of n chromosomes, where each of them has 2^m cells, where m is an integer equal to the proportionate size of the chromosome. Then for a particular

chromosome, pairs of two cells are made. Each pair is converted into a binary number using modulo 3. The result can be 0, 1, or 2. 0 stands for the first child of the present node in the state space tree, 1 stands for the second child, and 2 stands for the third child. If there is a blank tile at the edge, then there will be two cases, so we take modulo 2. If there is a blank tile at the corner, there will be only one case and modulo 1 will be taken. The step will return us ~m levels since each chromosome has 2^m cells. These levels are traversed and if the configuration becomes unsolvable while traversing then the chromosome is rejected. From the remaining chromosomes, if any path gives us the solution, then the algorithm stops. There can only be 1 solution for a particular instance of the problem. Otherwise, from all the evaluated chromosomes, we take two configurations of chromosomes and apply the crossover operator to them. This gives us a new path that needs to be traversed. If crossover does not result in a solution, then a mutant configuration is generated by randomly making a move. This can be implemented by using a Pseudo Random Number Generator and changing the path at i-th level. Starting from m = 2, m is incremented. This provides the power of Iterative Deepening mixed with a heuristic search. If the resources or time exceeds a particular threshold, it means it reached futility and the algorithm stops. [1]

3 Methodology

While multiple papers discuss the effectiveness of different algorithms for the famous N-puzzle, very little similar work is done on our Rubik's Race.

In this paper, we will implement A* and IDA* with Manhattan Distance, Number of Misplaced Tiles and Hill Climbing algorithms and Breadth First Search from the uninformed search strategies. The scalability of algorithms is an important factor, so apart from considering the number of nodes generated, the performance of the algorithms on the classical 5 x 5 board, larger sizes of boards will be considered to evaluate the algorithms. For now, 3 different sizes will be tested. Moreover, the initial states that will be tested on will not

be so random. On each board size, initial states that are specific moves away from the goal states will be considered and compared. Naimly, states that were generated by moving away from the goal by 10,20,30,40,50 steps. Each distanced type of state was considered 10 times to avoid biases as the moves away are random. At first, everything was tested on a 5x5 and 7x7 grid, and the algorithm with the best performance was tested on an 11x11 grid.

3.1. Solvability of the problem

Another important factor that must be considered is the solvability of the initial state. As was mentioned in the introduction, only solvable initial states must be considered for the game; otherwise, it doesn't make sense.

3.2. Heuristic functions

The logic of the Manhattan Distance was modified for our problem. As mentioned above, in this problem, the goal is determined by only the central part of the state. Hence, the Manhattan distance will only be calculated for the colors on the goal window. For each tile in the goal window, the Manhattan Distance from all other tiles of the same color in the state will be calculated, and the minimum of them will be taken as the Manhattan Distance. All the Manhattan Distances will be summed up at the end. The heuristic is admissible and also dominates the Number of Misplaced Tiles heuristic. The number of misplaced tiles works with the same logic as for N-puzzle.

3.3. Algorithms

Below, the implementation of each algorithm is explained:

Classes are in **bold**

Functions are in *italics*

Modules are <u>underlined</u>

BFS

1. board
    - *flatten(lst)* → flattens a list
    - *generate_random_state(n)* → generates a random state. It takes value n to generate the n x n grid. We take this as the "goal state." This function prevents the state from having a black(empty) tile in the central (n-2) x (n-2) part.
    - *display_grid(grid)* → displays a grid
    - *getGoalState(grid)* → this function takes as in input a grid (in our case, the grid is the random state generated by*generate_random_state(n)*) and returns its central (n-2) x (n-2) part since, as we defined our goal state is an (n-2) x (n-2) grid.
    - *display_scrambler(grid)* → displays the goal grid
    - *apply_moves_to_black_block(grid, moves)* → takes as a value a grid and a list with moves ("up," "down," "right," "left") and applies them on the black block (empty tile) of the grid
    - *move_away(grid, num_steps)* → this is a function to generate our initial state. So, we take the state generated by the function *generate_random_state(n),* and this is our input for the grid. The num_steps takes as an input a number equal to how many moves away we want our initial state to be from the goal state.
    - **Puzzle**
        - Parameters:
            - n → the size of our puzzle, an n x n grid
            - goal → The goal state we want to achieve for the puzzle
            - puzzle → the initial state of the puzzle
        - *find_black_block()* → returns a tuple containing the row and column indices of the black block, showing its position in the puzzle
        - *applyMove()* → swaps the black block in the grid with an adjacent block in the specified direction ("up," "down," "right," "left"). It updates the black block's position and the path cost(the cost from initial/root to that node) attribute accordingly
        - *is_goal_state()* → checks if the current state is the goal state

- get_possible_moves() → checks the possible moves that we can apply on the black block of the grid by checking its position. Return a list containing all the possible values we can apply on the black tile at that state
- *manhattanHeuristic()* → calculates the Manhattan Distance Heuristic. Each color tile in the goal state calculates the distance between the colored tile and every tile of that color in the initial state. From these, it chooses the one with the minimum distance. So, for each color tile in the goal state, we find one in the initial state closest to that position. Return all of them summed up
- *numberOfMisplacedHeuristic()* → calculates the number of misplaced tiles in the grid. Only checks the grid's central (n-2) x (n-2) part.

2. informed_search
   - **AStarSearchTree**
     - *searchManhattan* → A* tree search using the Manhattan Distance heuristic
     - *searchNumOfMisplaced* → A* tree search using the Number of Misplaced Tiles heuristic
   - **AStarSearchgraph**
     - *searchManhattan* → A* tree search using the Manhattan Distance heuristic
     - *searchNumOfMisplaced* → A* tree search using the Number of Misplaced Tiles heuristic

   - **IDA**(iterative deepening A*)
     - *searchNumMisplacedTiles* → search using the Number of Misplaces Tiles heuristic. It uses the iterative deepening A* algorithm. The threshold is updated at each iteration, setting it to the minimum

heuristic of the visited nodes (but not expanded as their heuristic was higher than the previous threshold), and the search continues until the goal state is reached.

- *NumMisplacedTiles_search_depth(self, current_state, g, visited)*→ runs a depth-first search using the Number of Misplaced Tiles heuristic. This explores the search space until the threshold is exceeded(returns None) or the goal state is reached. We call this function in the *searchNumMisplacedTiles* function. During each iteration, if None was returned, we aff the children nodes to a list, and from that list in the *searchNumMisplacedTiles,* we update the threshold to the minimum heuristic from that list.
- *searchManhattanDist(self)* → Similar to searchNumMisplacedTiles, this function uses the Manhattan Distance heuristic to perform a search.
- *ManhattanDist_search_depth(self, current_state, g, visited=0)* → Performs a depth-first search using the Manhattan Distance heuristic.

3. hill_climbing_and_bfs

- **HillClimbingSearch**
  - *convert_to_hashable(self, puzzle)* → Makes the list into a tuple so that it's hashable and faster to work within a set (hashable formats can also be used as keys in a set).
  - *search_misplaced(self)* → Implements the hill climbing search using the number of misplaced tiles as a heuristic function, allowing sideways moves and not allowing state repetition. It uses a set to keep all the visited states and generates the neighbors list based on all the possible moves from the current state. Chooses the best neighbor based on the minimum heuristic value. If the best neighbor has a higher heuristic than the current state, the search stops its execution; otherwise, it adds the current state to the visited set, and the current state is

set to the best neighbor. If the best neighbor is already in the visited set, it checks for the second-best neighbor. In the case of ties, it selects the best neighbor randomly.

- *search_manhattan(self)* → Implements the hill climbing search using the Manhattan distance as a heuristic function; everything else is the same as in search_misplaced.
- *generate_neighbors(self, current_state)* → Takes the current state and generates its neighbors based on all the possible moves.

- **BreadthFirstSearch**
  - *convert_to_hashable* → flattens a nested list structure into a tuple
  - *search* → breadth-first search algorithm

4 Results

4.1 Results

Before going further with the analyses, an important note should be made. When testing the algorithms and encountering problems with the time that it takes to run or other technical problems, NAs were added to our data. NA does not show if a problem was solved or not. It indicates that the computer was not managing the problem, which is also valuable data. However, only when looking at the data collected from testing Hill Climbing, will the NA mean a failure and not a technical issue.

BFS

As was expected, an uninformed search is not efficient for solving such types of problems. The number of expanded nodes is exceeding thousands (Figure 1).

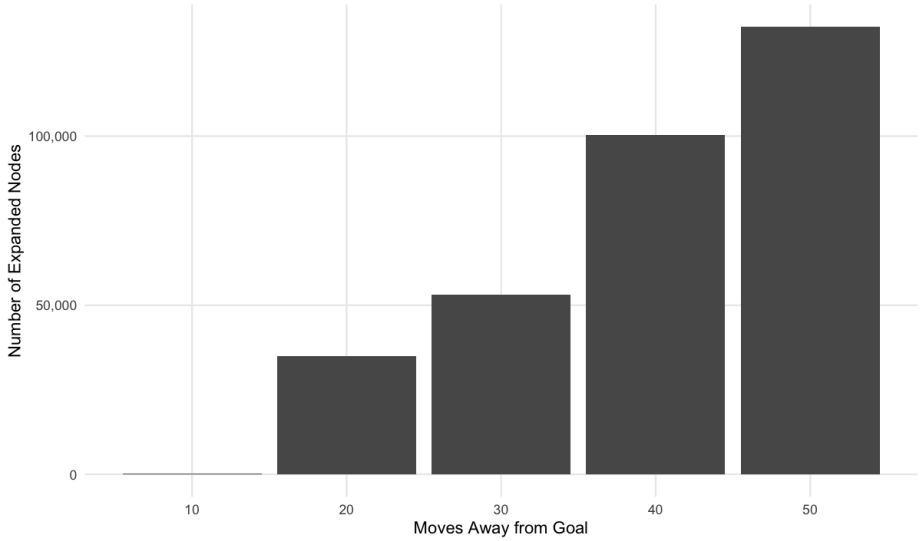

Figure 1: Average Number of Expanded Nodes for BFS

Informed search strategies

Both variants of A* - graph and tree are implemented and, obviously, the graph search is more efficient (Figure 2).

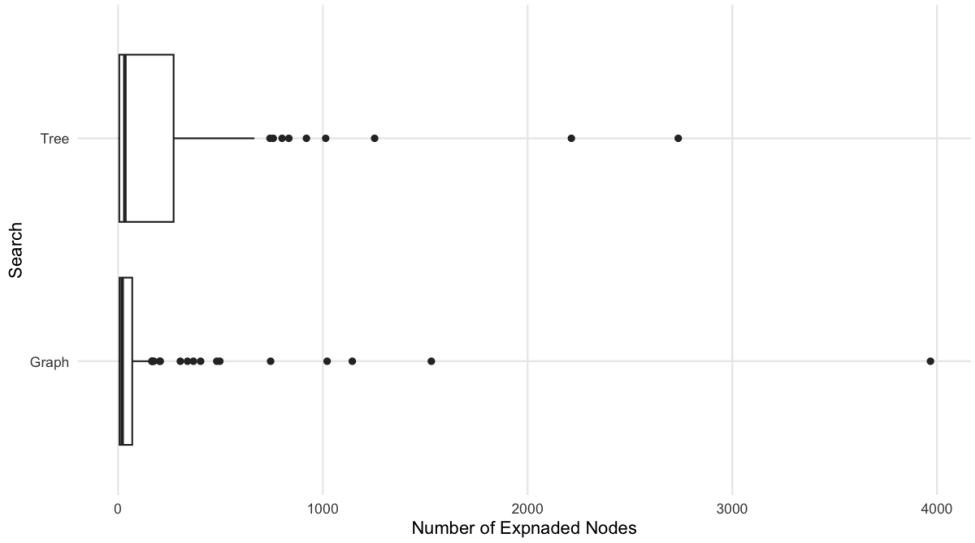

Figure 2: Number of Expanded Nodes for A* Algorithm: Graph vs Tree

After this experiment it was decided to implement only graph version of IDA* since graph search is more efficient for solving Rubik's race.

As a result, number of expanded nodes for IDA* graph in average is 12.356 while for A* graph it is 188.9664 (Figure 3). The difference is quite large so, in conclusion, IDA* graph is more efficient.

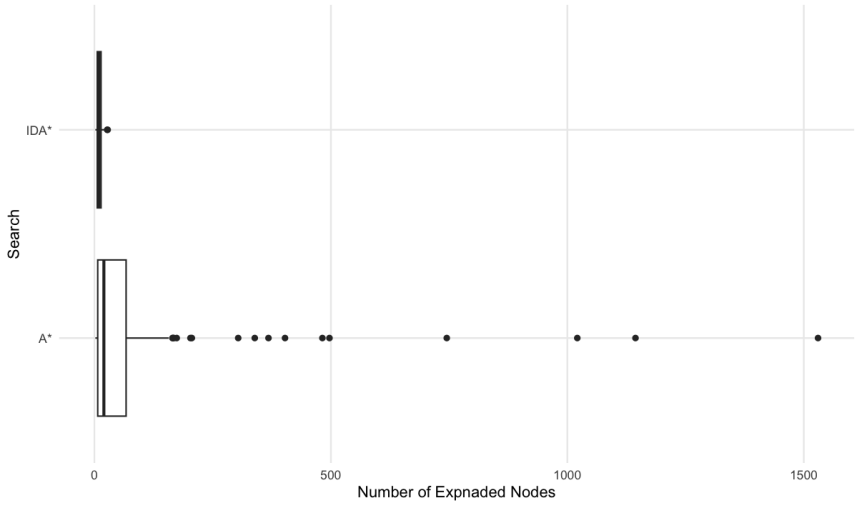

Figure 3: Number of Expanded Nodes for A* Graph and IDA*

Before that we considered implementations of all the algorithms with both the number of misplaced tiles and Manhattan distance heuristic functions. Since we already made a conclusion that IDA* graph search is the most efficient algorithm so far, heuristic functions must be considered at the next step of analysis.

In average, the amount of expanded nodes for IDA* with those heuristic functions are close: for number of misplaced tiles it's 10.65753 and for Manhattan distance it's 11.15385. Both functions are suitable for IDA* and make the solution efficient.

Hill Climbing

The next strategy that was implemented is a local search - Hill Climbing. As we can see from the plot (Figure 4), almost half of the tests using the algorithm returned failure. Still, it is important to mention that when the algorithm gives us a solution, it is fast and expands fewer nodes. With this advantage, however, the local search algorithm turned out to be less suitable for our problem as the lack of guarantee of finding a solution is a big problem.

Additionally, plot (Figure 4) again shows that IDA* performs the best among the considered algorithms having less terminated iterations.

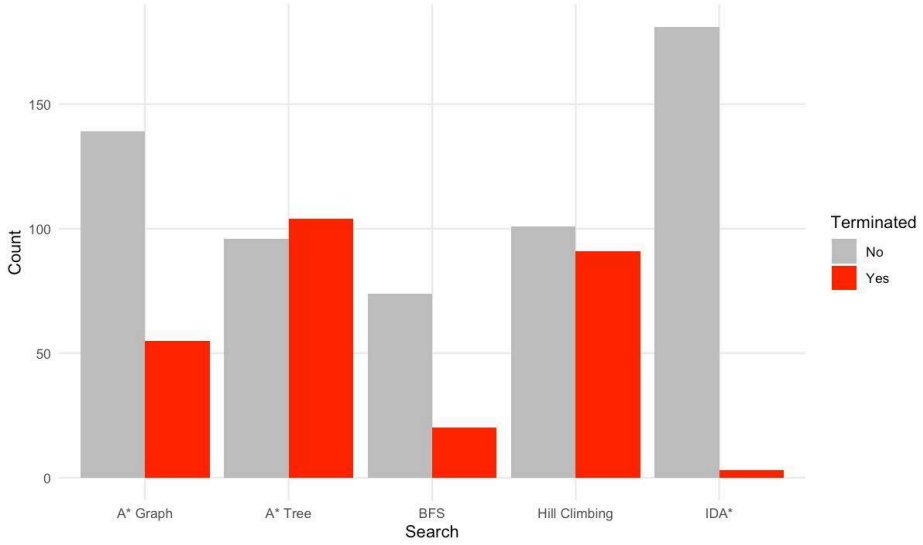

Figure 4: Histogram of Terminated Implementations for Each Search

We would also like to mention that the plots are not based on the grid size at all, and no analyses were done based on the grid size because, during our tests, we found that the algorithms' performance is more dependent on how far away the state is from the goal state and very little affected by the grid size.

5 Conclusion

5.1. Conclusion

In conclusion, after testing A* tree and graph searches, IDA* graph search and hill climbing with 2 heuristic functions: number of misplaced tiles and Manhattan distance, IDA* with both of the heuristic functions is found out to be the most efficient strategy for solving Rubik's race.

5.2. Future work

This paper didn't discuss the implementation of solving Rubik's race by applying the genetic algorithm which was quite efficient for the N-puzzle problem. Additionally, the linear conflict heuristic function was not applied. Those strategies may be efficient for solving the game problem as well. In fact, there may be more algorithms for the Rubik's race in the world of AI but in the paper were introduced the most popular and easy to implement ones.

References

[1] Bhasin, H., Singla, N. (2012, August). *Genetic based Algorithm for N-Puzzle Problem.* International Journal of Computer Applications (0975 – 8887) Volume 51– No.22. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=44a86bb2cd7f65a300b725a050a3a1614d006ca3

[2] Chen, Y., & Kuo, T. (2013). *Experimental Comparison of Uninformed and Heuristic AI Algorithms for N-Puzzle Solution. International Journal of Innovative Computing, Information, and Control*https://www.researchgate.net/publication/259694537_Experimental_Comparison_of_Uninformed_and_Heuristic_AI_Algorithms_for_N_Puzzle_Solution

[3] Nayak, D. (2014, May). *Analysis and Implementation of Admissible Heuristics in 8 Puzzle Problem.* https://core.ac.uk/download/pdf/53190059.pdf

[4] Jiang, W. (2021). *Analysis of Iterative Deepening A* Algorithm.* IOP Conf. Ser.: Earth Environ. Sci. 693 012028. https://iopscience.iop.org/article/10.1088/1755-1315/693/1/012028/pdf

[5] Reinefeld, A. *Complete Solution of the Eight-Puzzle and the Benefit of Node Ordering in IDA*.* Paderborn Center for Parallel Computing. https://www.ijcai.org/Proceedings/93-1/Papers/035.pdf