A* Puzzle Solver Agent

Khalid Almotaery

Introduction

The goal of this project is to understand the fundamentals of Artificial Intelligence by implementing the A* algorithm. This algorithm solves problems by moving from one state of the problem to another based on a heuristic function and a cost function. The algorithm keeps transitioning between states until the current state is the goal state. The heuristic function and cost function allows the algorithm to pick the next best state. The optimal choice is the state with the lowers F value. A state x in the problem has an F value given by the formula 1.

$$f(x)=g(x)+h(x)$$

(1)

There are different types of heuristics that can be embedded in the algorithm. The main requirement for these heuristics is that they are optimistic and never overestimate the cost of the next state.

This project uses the A* algorithm to solve the 3, 8, and 15 puzzle games. In this context, the states are the different configurations of each puzzle. The algorithm was implemented using four heuristics: number of misplaced tiles, manhattan distance, max sort swaps count, and the Euclidean distance.

Program design

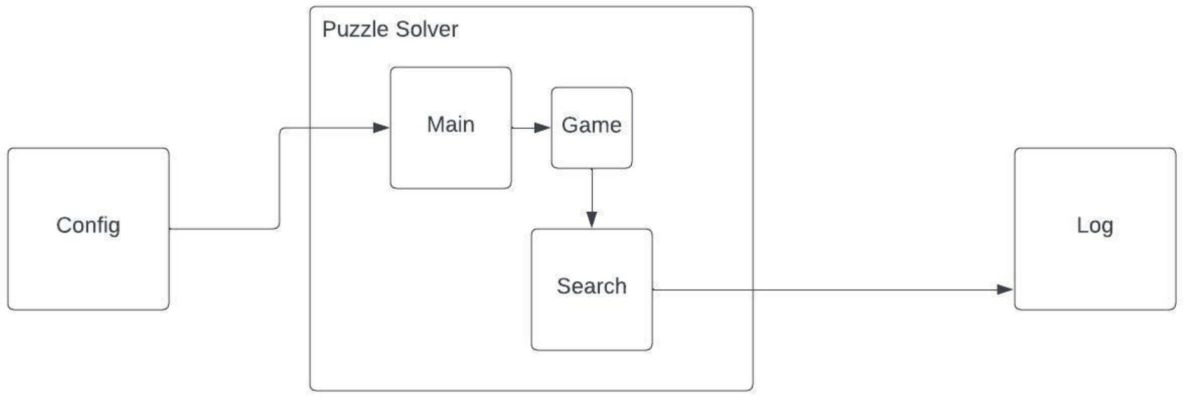


Figure 1

The high-level abstraction of the program is shown in Figure 1. The input of the program is supplied by the configuration file and the output of the program is sent to the log file. The program is constructed using three classes: Main, Game, and Search.

The Main class is responsible for starting the program, getting the inputs from the configuration file, creating initial Game objects, running the search algorithm with created puzzles, and putting the outputs in the log file.

The Game class represents the board, keeps count of variables related to the board, and has functions such as moving the space and printing the board. Each game object has its own F function. Depending on the heuristic selected in the configuration file, F will have a different value.

The Search class implements the A* Algorithm. Taking in the initial state, the algorithm creates 4 substates where the space is moving up, down, right, and left from its current position. The configuration with the lowest F value will be selected as the next state. This continues until the next state is the goal state.

Because the algorithm creates new objects for each substate, there needs to be a mechanism to keep track of visited configurations and not objects. Thus the Search class creates a dictionary where the key is the hash of the configuration and the value is the first game object with that configuration. Thus, a new state has created the algorithm does the following:

1. Hashes the configuration of this new state.
2. Checks if the dictionary has a key that matches the hashed configuration
3. If the key exists, then it gets the object associated with that key and increases its g value (the cost to get to the node)
4. If the key does not exist, then it adds the key and links it to the current state

User manual

To run the program please follow these steps:

1. Go to the directory where the project is at.
2. Simply write “Make run” to execute the program. This command will run the python program based on the configuration file and would output results in the log file.

The configuration file allows the user to select the mode of the program and the puzzles to be solved. There are three modes of the program: an experiment mode a one-game mode and a puzzle generator mode. The user can select mode under the “Mode” section of the configuration file. Note the user may select both modes of the program.

The experiment mode solves multiple puzzles of different edge sizes with all heuristics. The edge sizes to be tested and the initial configurations are declared in the “Experiment” section of the configuration file. The user may edit the initial configurations array to produce new puzzles for the program. Also, the user may choose which edge sizes to be tested. Note that running all edge sizes at once may take so much time and it is recommended that the user runs one edge at a time.

The one-game mode solves only one puzzle. The specification of the puzzle such as the edge size, the heuristic to be used, and the configuration of the initial state can be edited under the “SolveOne” section.

The puzzle generator mode simply outputs random puzzles in the log file so that the user can use them in the other modes for the program. The user can select the size of the puzzles to be generated under the “Generator” section of the configuration file.

Four-Phase Problem Solving Process

- Goal Formulation
    - A board configuration where the numbers are ordered in increasing order.
- Problem Formulation
    - State space: all of the possible board configurations.
    - Initial State: any board configuration can be the initial state.
    - Actions: the space can move either up, down, left, or right.
    - Transition Model: maps a configuration and an action to a new configuration.
- Search
    - Implementing the A* algorithm to find the goal configuration.
    - Solution: a set of states that start from the initial state to the goal state.
- Execution
    - Using the python program, the agent executes the actions that lead to the goal state.

Heuristics Design

The number of misplaced tiles is H1. The heuristic goes through each cell in the goal state and checks the corresponding digit in the current state. If the cells do not match, then the H1 value would increase by 1. The heuristic repeats this process until all the cells have been checked.

H2 is the Manhattan distance heuristic. The heuristic creates x, and y coordinates for each cell in the goal state and the current state. For each cell in the current state, the heuristic uses the manhattan distance formula shown below to calculate the distance between the current cell and where it should be.

$$|x1-x2|+|y1-y2| \tag{2}$$

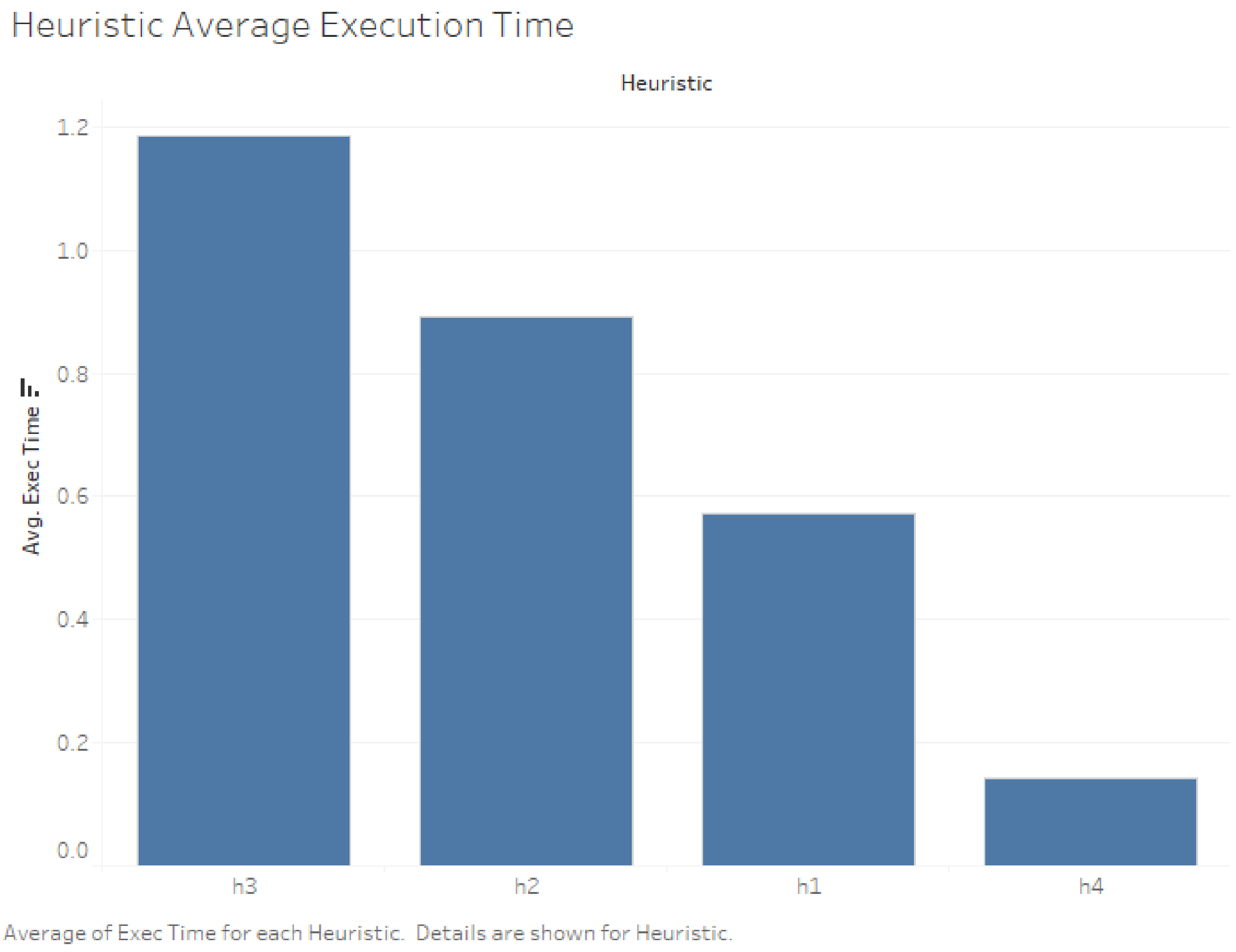
Then, the heuristic sums all the manhattan distances for all the cells.

H3: converts the current board into an array and performs max sort. Whenever a swap is made the objective value is increased by 1.

Finally, the Euclidean distance, H4. The heuristic is similar to H2, but it uses the Euclidean distance formula (shown below) instead of the Manhattan distance formula.

$$\sqrt{(x_2 - x_2)^2 + (y_2 - y_2)^2}$$
(3)

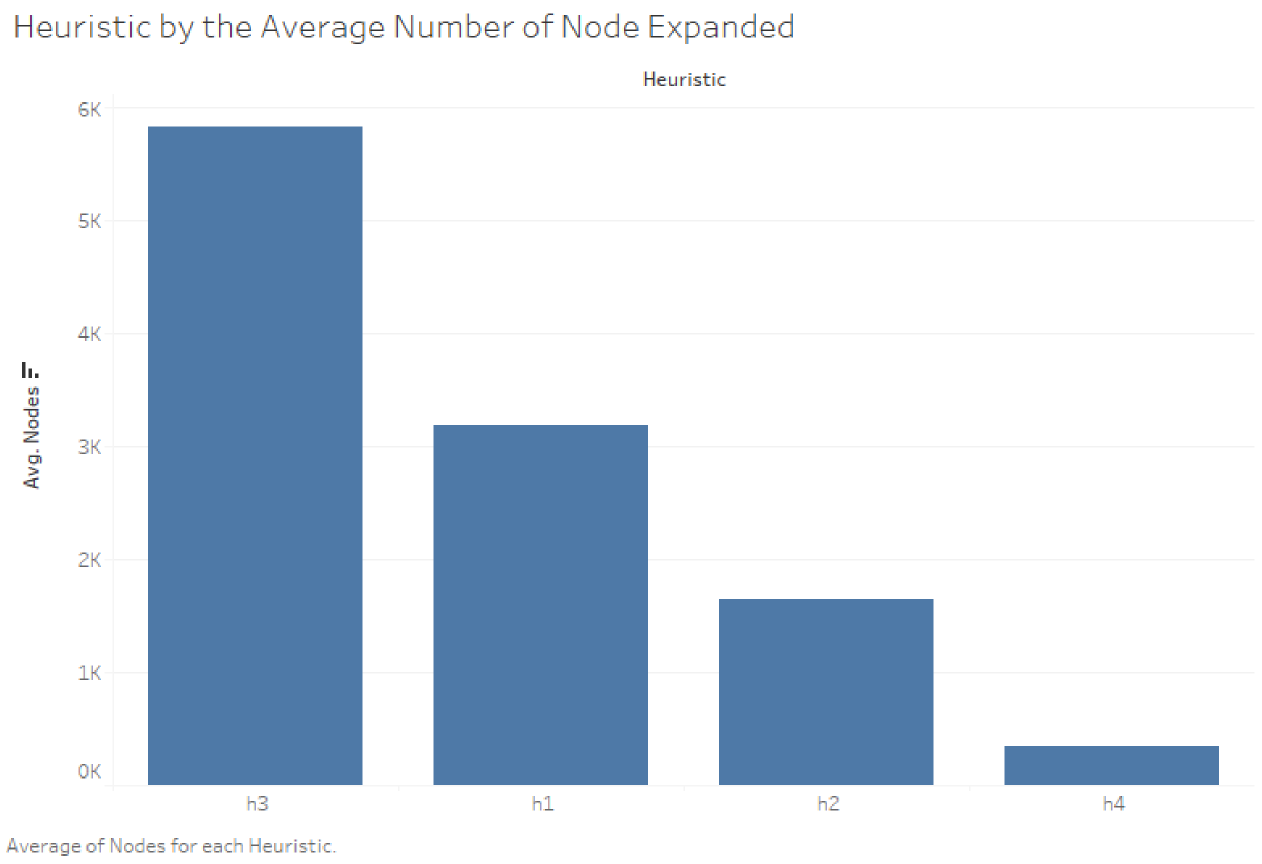
Program Performance
The program compares execution time, and the number of nodes expanded when using h1, h2, h3, and h4 for the different edge sizes based on the percent of misplaced tiles. The results of the experiments are displayed in this section of the report.



Heuristic Average Execution Time

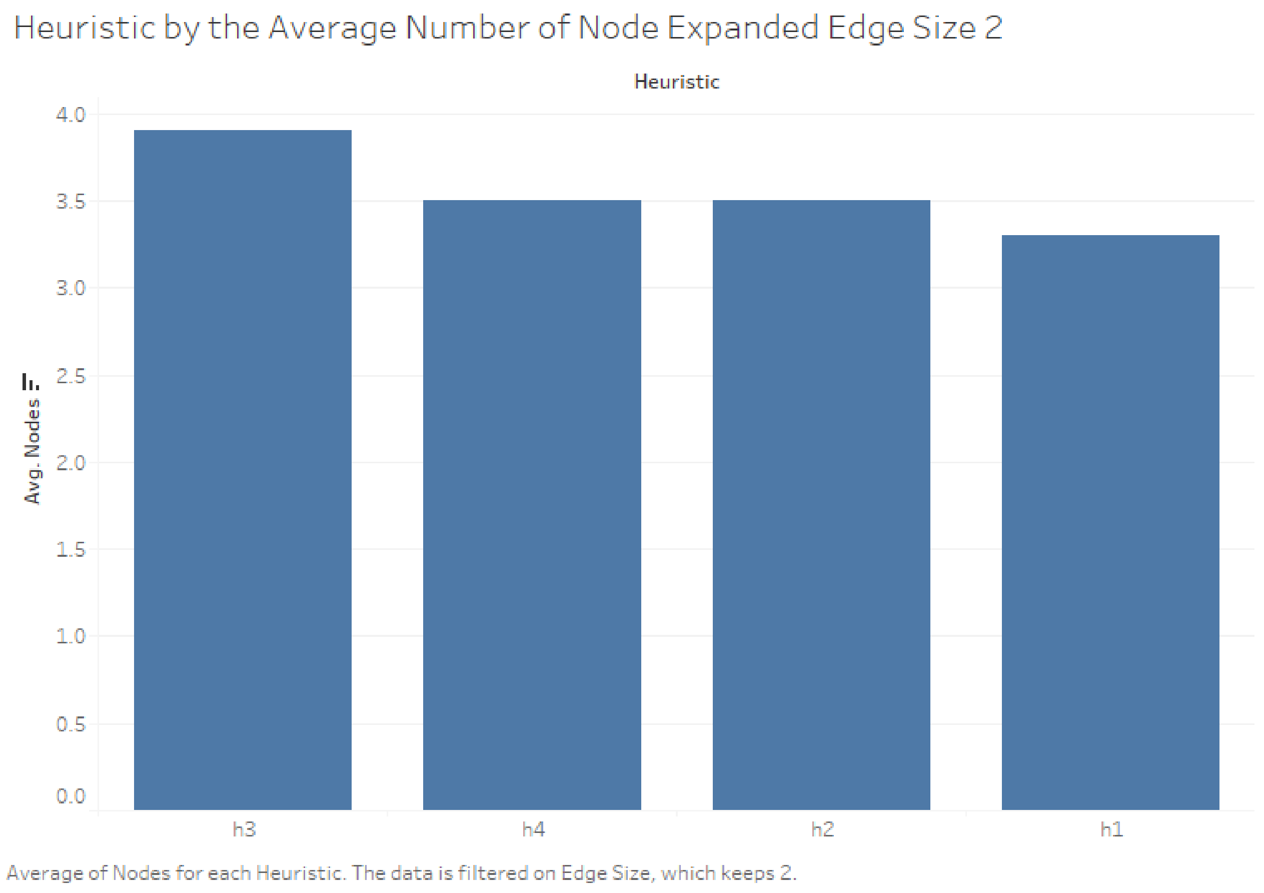
Figure 2

The above figure shows the average execution time for each heuristic when used for puzzles of edge sizes 2,3, and 4. The puzzles had different misplaced tile percentages.



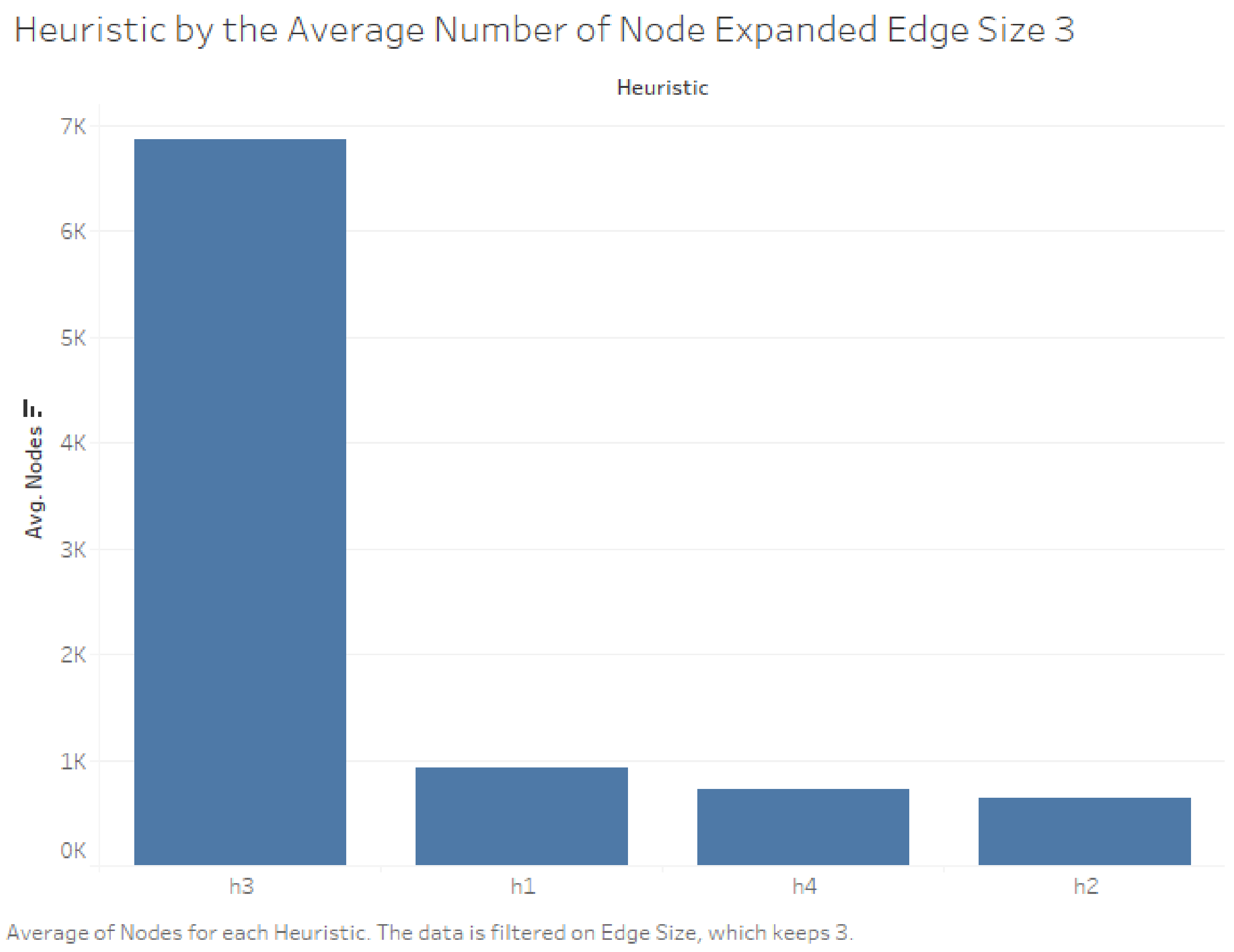
Heuristic by the Average Number of Node Expanded
Figure 3

The above figure shows the average number of nodes expanded for each heuristic when used for puzzles of edge sizes 2,3, and 4. The puzzles had different misplaced tile percentages.



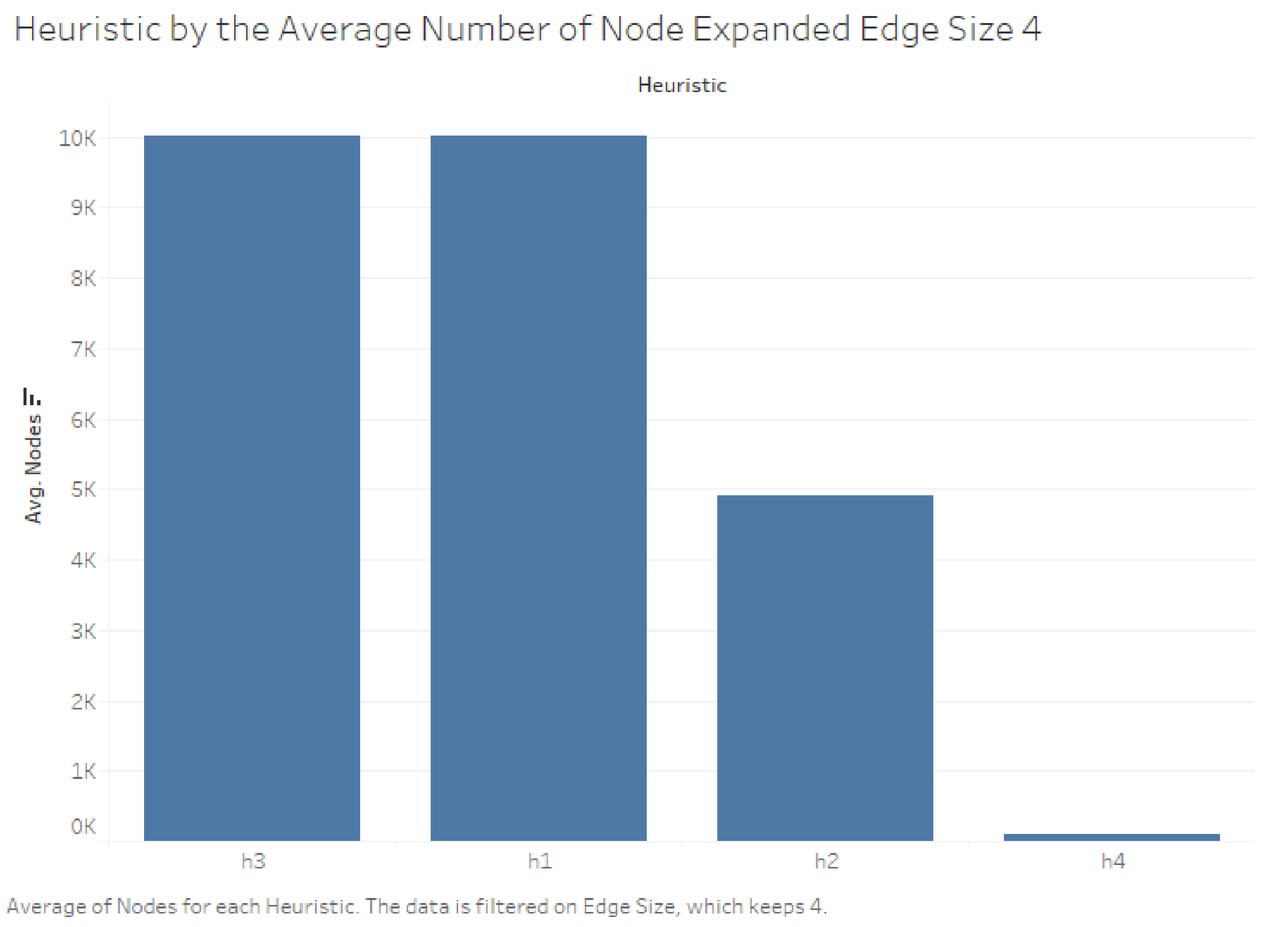
Heuristic by the Average Number of Node Expanded Edge Size 2
Figure 4

The above figure shows the average number of nodes expanded for each heuristic when used for puzzles of edge sizes 2. The puzzles had different misplaced tile percentages.



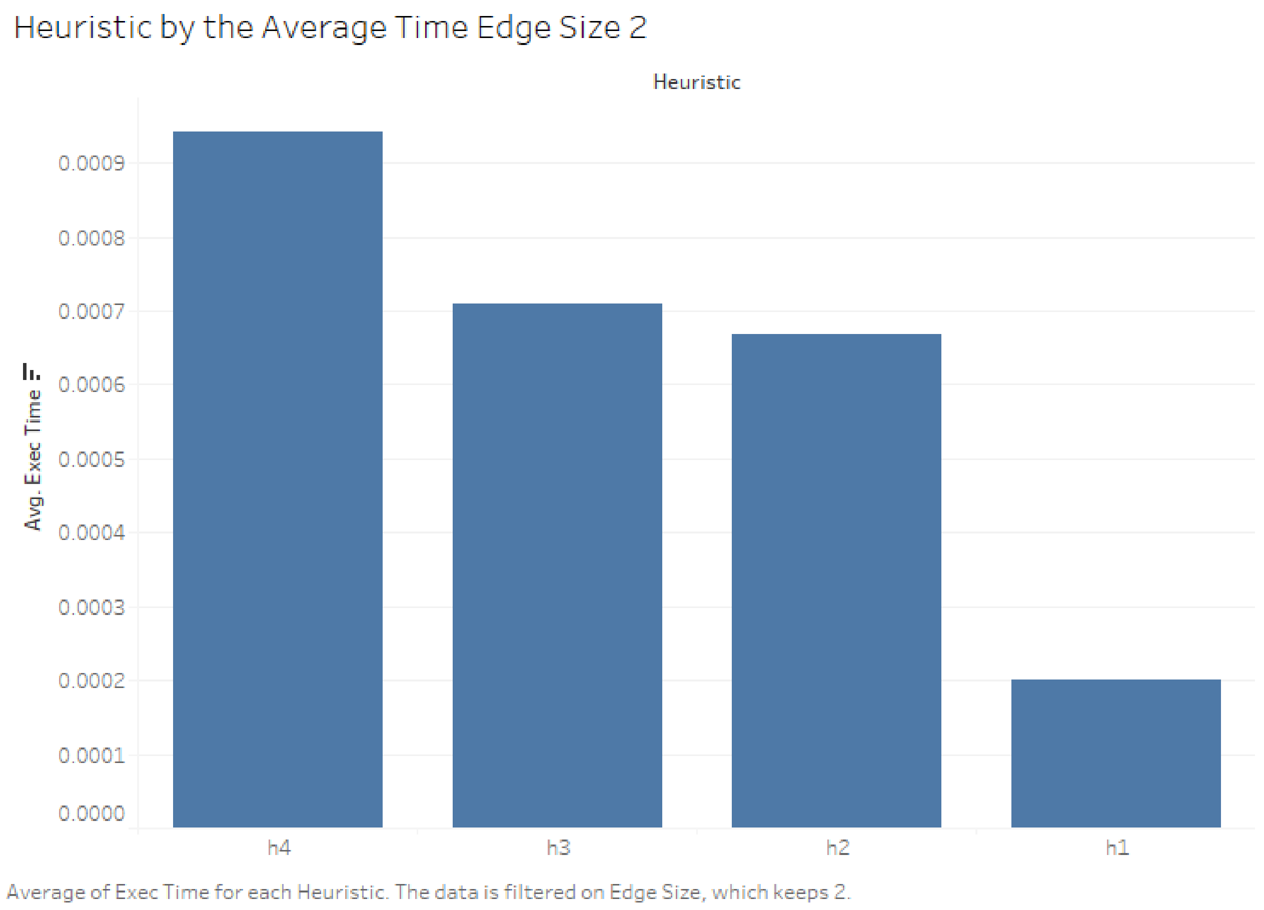
Heuristic by the Average Number of Node Expanded Edge Size 3
Figure 5

The above figure shows the average number of nodes expanded for each heuristic when used for puzzles of edge sizes 3. The puzzles had different misplaced tile percentages.



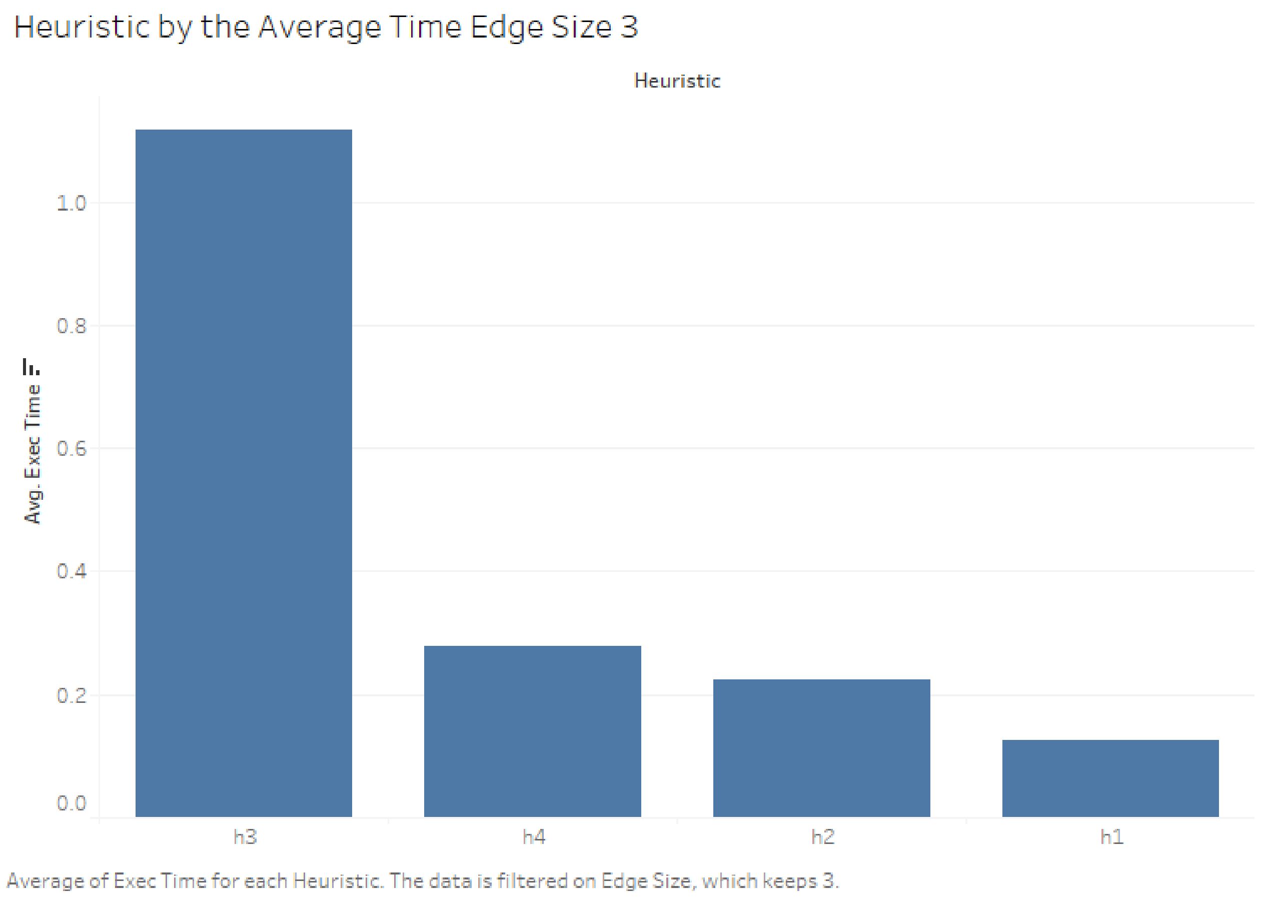
Heuristic by the Average Number of Node Expanded Edge Size 4
Figure 6

The above figure shows the average number of nodes expanded for each heuristic when used for puzzles of edge sizes 4. The puzzles had different misplaced tile percentages.



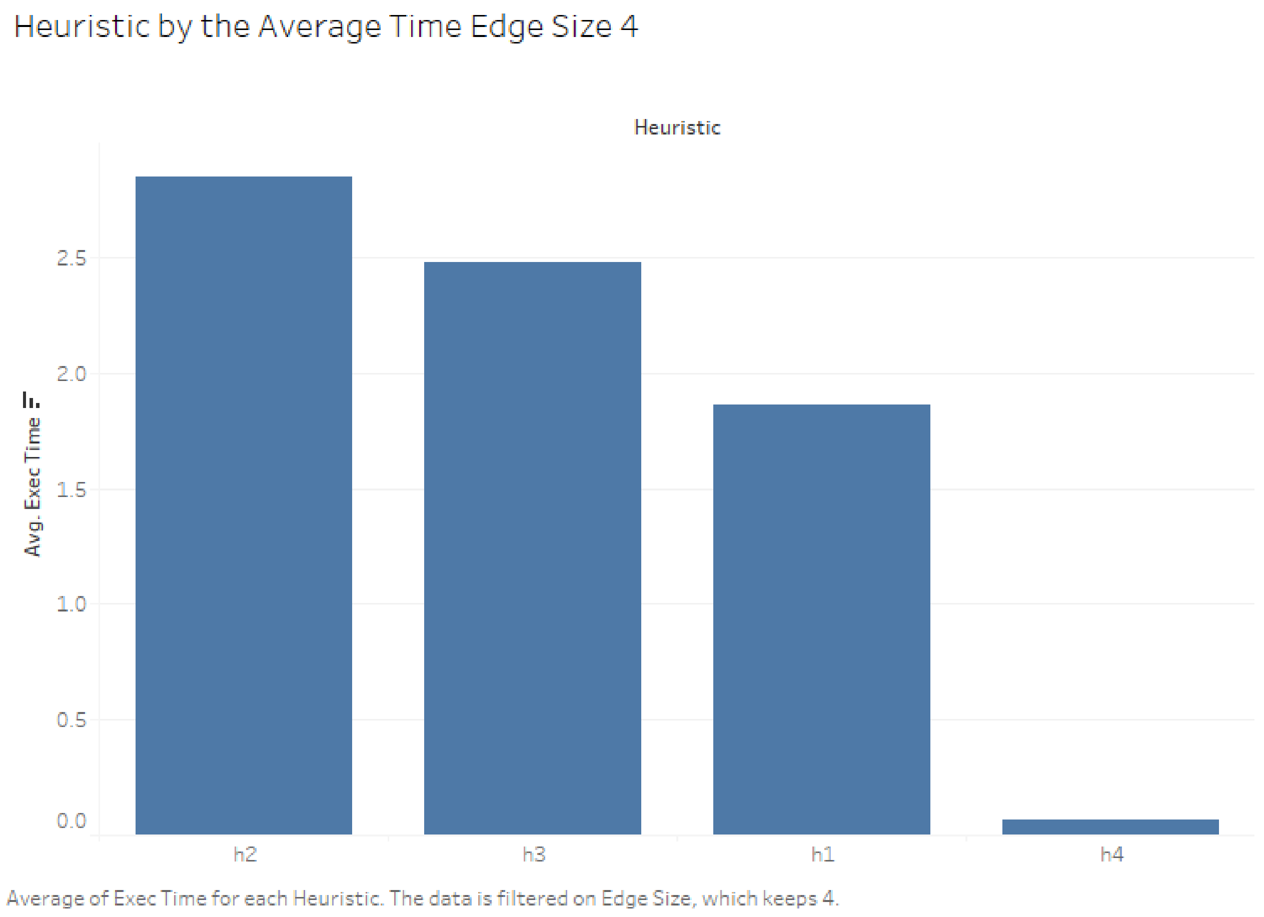
Heuristic by the Average Time Edge Size 2
Figure 7

The above figure shows the average execution time for each heuristic when used for puzzles of edge sizes 2. The puzzles had different misplaced tile percentages.



Heuristic by the Average Time Edge Size 3
Figure 8

The above figure shows the average execution time for each heuristic when used for puzzles of edge sizes 3. The puzzles had different misplaced tile percentages.



Heuristic by the Average Time Edge Size 4
Figure 9

The above figure shows the average execution time for each heuristic when used for puzzles of edge sizes 4. The puzzles had different misplaced tile percentages.

Summary and Result Interpretation

The graphs above show that with different edge sizes there are some changes in the heuristics' ranking. Looking at the overall average execution time and the number of nodes expanded (**Figures 2** and **Figure 3**), we see that H4 expands the least number of nodes and that it is the fastest. However, looking at **Figure 8** and **Figure 7,** we see that H4 is not the fastest heuristics at edge sizes 2 and 3. Thus it seems like H4 does extra work when edge sizes are small which makes it inefficient. In general, H4 seems to do better than the other heuristics.

On the other hand, H3 seems to be the worst algorithm in all measures. It ranked the highest in execution time and the number of nodes expanded in all situations except in **Figure 9.** This low performance may be the result of the algorithm's high complexity time compared with the other heuristics. The Algorithm's complexity time is O(n^2). Even after H3 is able to produce an estimate this estimate may not be as informative as the other heuristics.

Even though H1 expands more nodes on average than the other heuristics, it still remains relatively fast. In **Figure 3**, we see that H1 expands double the amount of nodes that h2 expands. However, looking at **Figure 2**, we see that H1 is faster than H2. This may be because of the heuristic’s simplicity and linear time complexity.