Maksym Turkot
10/07/22

A* Search n-Puzzle Solver and Heuristic Domination Analysis

Introduction

Searching algorithms are at the foundation of the methods that AI agents use to solve problems. In this project, I am implementing an agent that uses A* searching algorithm with two heuristics to search through 3-, 8-, and 15-puzzles, and comparing the performance results for both heuristics.

This report includes the following sections:

- **Program Design 1**
  Explains the organization of the program.
- **User Manual 8**
  Guide to how the program should be configured and run.
- **Four-Phase Problem Solving Design 10**
  Description of the high-level ideas behind the agent behavior using FPPS.
- **Heuristics Design 11**
  Explanation of how each heuristic and how it is implemented.
- **Program Performance Results 11**
  Presentation of aggregated performance data and discussion.
- **Results Summary 17**
  Conclusion of the results and determination of heuristic dominance.

Program Design

Program contains two major modules:

- The first module is the puzzle generator that reads the puzzle configuration files and creates sets of puzzles based on specified parameters. These puzzles are stored to the puzzle set file.
- The second module reads the puzzle set file and runs the search on each of the puzzles in the file. The summary of program execution based on each heuristic used, as well as parameters used to generate the respective puzzle set are then logged to the log file.

Project was developed using Java programming language. The following pages provide a detailed description of each class that builds up the program. This section may be glanced over for familiarity with the program structure. Class diagram is provided in Figure 1.

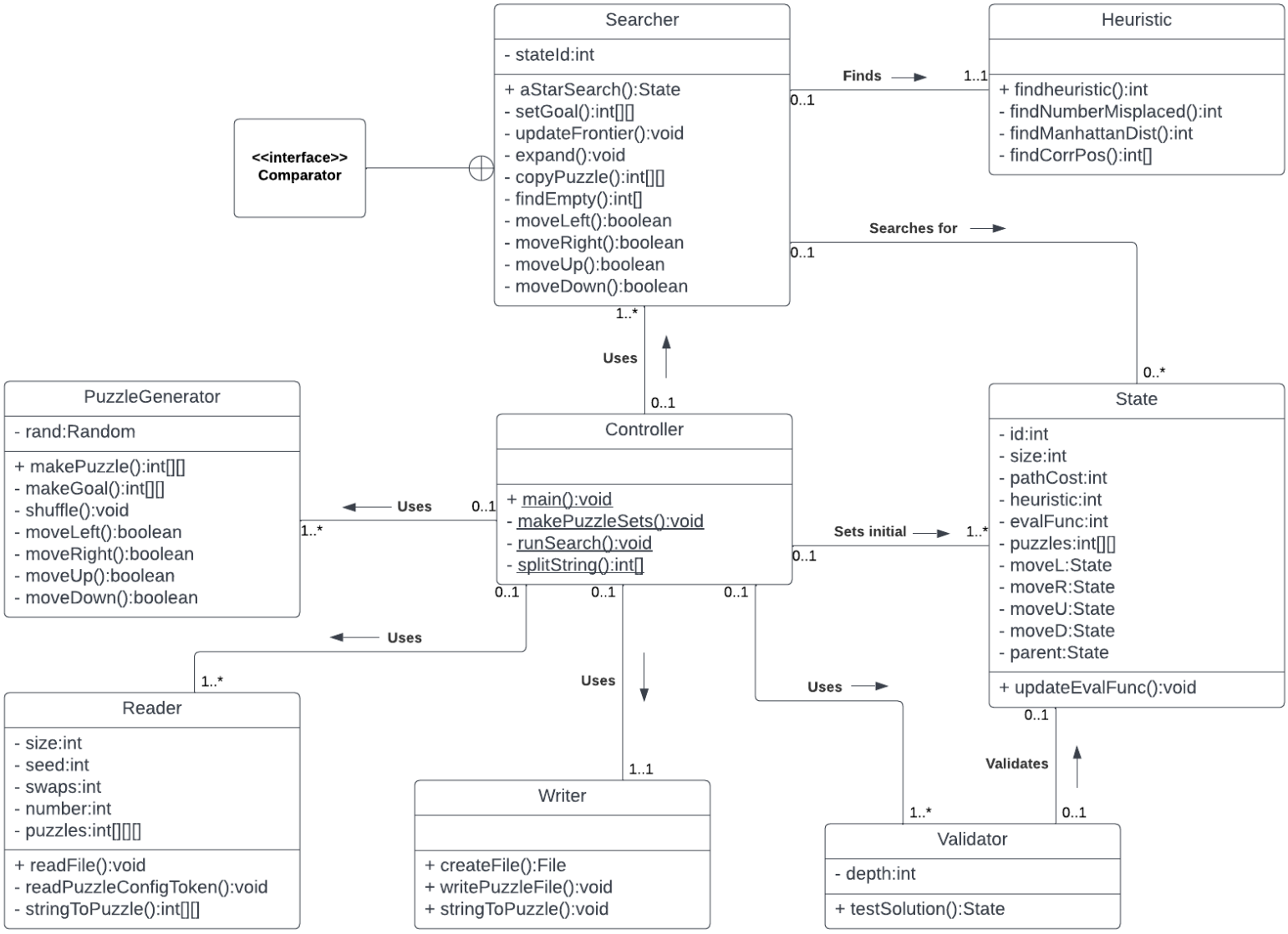

Figure 1. Class Diagram

Controller

Contains the **main()** method that starts the program execution, as well as two methods that split the program into two modules: **makePuzzleSets()** that generates sets of random initial states, and **runSearch()** that executes the search, validates the solution, and outputs the result of the program. A helper method **splitString()** is also present.

Methods

main()
Creates the reader and writer objects and calls the makePuzzleSets() and runSearch() methods.

makePuzzleSets()
Reads all of the puzzle configuration files stored in the data/puzzleConfig directory, calls puzzleGenerator to create the random initial states, and writes these states as strings to a file in the data/puzzleSet directory.

runSearch()
Reads all of the puzzle set files stored in the data/puzzleConfig directory, runs A* search using each heuristic, validates the solutions, and writes the aggregated results to the data/log.txt file.

splitString()
Used to split the filename string of the puzzleSet file to retrieve puzzle configuration info.

PuzzleGenerator

Includes methods **makePuzzle()** to generate a random puzzle, **makeGoal()** to generate the goal state, **shuffle()** to shuffle the goal state to a solvable initial state, and **moveLeft()**, **moveRight()**, **moveUp()**, and **moveDown()** methods used for shuffling.

Fields

Random **rand** used for random number generation.

Methods

makePuzzle()
Initializes the goal state and runs the shuffle on it to produce the initial state.

makeGoal()
Generates a goal state for passed puzzle size.

shuffle()
Generates a stream of random integers (ranging from 0 to 3) used for shuffling. For each integer, either left, right, top, or bottom tile is moved, generating the next state, provided such move is possible.

moveLeft()
Moves the left tile in the current state, provided such move is possible.

moveRight()
Moves the right tile in the current state, provided such move is possible.

moveUp()
Moves the top tile in the current state, provided such move is possible.

moveDown()
Moves the bottom tile in the current state, provided such move is possible.

State

This class constructs state objects that store information about the puzzle state. It includes data fields as well as getters and setters for those fields.

Fields

int **id** of the puzzle.
int **size** of the puzzle.
int **heuristic** value of the puzzle.
int **pathCost** value of the path cost to get from the initial state to the current state.
int **evalFunc** sum of the path cost and heuristic fields.
int[][] **puzzle** array of the puzzle.
State **moveL** left tile move state, derived from puzzle.
State **moveR** right tile move state, derived from puzzle.
State **moveU** top tile move state, derived from puzzle.
State **moveD** bottom tile move state, derived from puzzle.
State **parent** state from which this puzzle was derived.

Methods
Constructor, getters, and setters.

Heuristic

This class implements the logic of heuristic calculation. The **findHeuristic()** processes the heuristic string and activates the correct heuristic construction method, **findNumberMisplaced()** or **findManhattanDist().** The former computes the number of misplaced tiles in the current state, while the latter finds the distance along row and columns of the puzzle from each tile's current position to the goal position. A helper method **findCorrPos()** is also present.

Methods

findHeuristic()
Accepts the string indicating the heuristic, and based on it activates the correct heuristic construction function.

findNumberMisplaced()
Computes the number of tiles misplaced from the goal state by comparing value of each cell to the incremented counter.

findManhattanDistance()
Computes the Manhattan distance. For each tile, it compares its value to the expected correct value. If they are different, it scans the puzzle for the correct position of the tile, and calculates the distance between the expected and current tiles.

findCorrPos()
Finds the coordinates of the correct position of a puzzle tile.

Searcher

This class searchers the puzzle states for the desired goal state. The **aStarSearch()** method runs the search logic; **setGoal()** method creates a reference goal state based on the puzzle size; **updateFrontier()** adds four children states to the frontier PriorityQueue and computes the heuristic and pathCost values; **expand()** method creates children states from the current state with up to four possible tile moves. Additional helper methods are present, including **copyPuzzle()**, **findEmpty()** which finds coordinates of the empty tile, and **moveLeft()**, **moveRight()**, **moveUp()**, and **moveDown()** methods. Includes an inner class **StateCoparator** with **compare()** function to compare two states based on evalFun for the PriorityQueue.

Fields

Heuristic **heuristicFinder** used to compute heuristic values.
int **stateId** used to keep track of and identify the states as they are expanded.

Methods

aStarSearch()
Keeps track of the frontier of candidate states as well as a lookup table of reached states for performance optimization. The state from the frontier with the smallest evalFunc value is checked for being a goal; if it is not, it is expanded, and its children are added to the frontier. Before expanding a state, its puzzle value is checked against reached states in the lookup table.

setGoal()
For the puzzle size, creates a goal state to be used by the searching algorithm.

updateFrontier()
For each child state, this method computes the evaluation function and adds the child to the frontier.

expand()
Expands a state by creating a child state for each possible tile move.

copyPuzzle()
Creates a copy of the puzzle array.

findEmpty()
Finds the coordinates of the empty tile.

moveLeft()
Creates a new puzzle state by moving the left tile.

moveRight()
Creates a new puzzle state by moving the right tile.

moveUp()
Creates a new puzzle state by moving the top tile.

moveDown()
Creates a new puzzle state by moving the bottom tile.

Validator

This class implements the **testSolution()** method that walks up the parent references from the result state to the initial state to make sure that the solution is valid.

Fields

int **depth** of the solution tree.

Methods

testSolution()
Walks up the parent references of the result state to validate a solution and find the depth of the tree.

Reader

This class implements the reading features of the project. Configuration files fully determine how the program is run. The **readFile()** method reads a specified file, and calls the **readPuzzleConfigToken()** or **stringToPuzzle()** to extract data from the file into the system.

Fields

int **seed** used to generate the initial puzzles.
int **size** of the puzzle.
int **swaps** used to generate the initial state.
int **number** of puzzles to be generated.
int[][][] **puzzles** array to be generated.

Methods

readFile()
Reads the specified file by calling a necessary helper function to store the information into the system.

readPuzzleConfigToken()
Splits the read string into tokens and stores each one as a variable to the system.

stringToPuzzle()
Converts a string specifying the puzzle state to an array and stores it to the array of states.

Writer

This class implements the reading functionality of the program. The method **createFile()** creates a new file, overwriting the existing ones with the same name. Method **writePuzzleFile()** creates and writes a file containing initial states of the puzzle. Method **writeLogFile()** writes program execution results to the log.txt file.

Methods

createFile()
Creates a new file with a specified path. If the file already exists, it is overwritten.

writePuzzleFile()
Creates and writes the passed puzzle state set to the file with a filename reflecting the properties of the puzzle set.

writeLogFile()
Writes program execution data to the log file, appending to the existing file.

User Manual

The entire program is in the turkotm folder. Source code is located in the src/main folder, and data files are in the data folder.

Configuring the program

Program is completely configured by the puzzleConfig files, and it takes no input from the terminal. To change the behavior of the program, user should modify or add puzzleConfig files.

Configuration files must be located in the project1/data/puzzleConfig folder. The structure of the file is as follows:

size: <size> // Size of the puzzle (2/3/4…).
seed: <seed> // Random seed.
swaps: <swaps> // Number of random swaps from goal state (randomization).
number: <number> // Number of initial states generated.

The phrase “puzzleConfig” must be present in the filename. The following naming convention is recommended:

puzzleConfig-<size>-<seed>-<swaps>-<number>.txt

For example:

puzzleConfig-3-111-200-100.txt
puzzleConfig-4-2323-30-50.txt

The configuration file controls the program in the following way:

- **Size** variable tells the program what size puzzles should be generated from this particular configuration file.
- **Seed** variable is used to generate a sequence of integers to perform random swaps on the goal state, generating a necessarily solvable initial game state.
- **Swaps** is the length of the random number sequence generated, and hence the level of randomization of the puzzle. Please keep in mind that each random number maps to a specific move, which is carried out only when it is possible (left move on a central tile. Move up on the top tile is not possible). Hence, the actual number of swaps will be smaller than the entered swap value.
- **Number** variable tells how many initial puzzle states should be generated and later searched.

Running the program

Program was developed using Java.

To run the program, please make sure you are located in the turkotm folder. The Makefile is provided, so you can run the program using the following commands:
- **make** or **make compile** - compiles the program
- **make run** - runs the program
- **make clean** - removes compiled .class files.

When running, program will first read puzzleConfig files and generate puzzle sets, and then save them to the data/puzzleSet folder. The puzzleSet file simply stores string representations of puzzle 2D arrays. The naming format of a puzzleSet file is similar to the puzzleConfig one, and *must not be modified*:

puzzleSet-3-111-200-100.txt

Each puzzleSet file corresponds to one puzzleConfig file. User has no need to edit or add puzzleSet files, even though they may inspect files to see what puzzles are being solved.

As the program is executed, a progress bar will appear for each file, with a "#" indicating a solved puzzle.

Program Output

Upon completion, program will record results in the log.txt file, located in the data folder. Each entry in the log file corresponds to one puzzleConfig/puzzleSet pair of files. The structure of a log file entry is as follows:

```
************************
size:           3
seed:           2424
swaps:          80
number:         100
========================
H1: Number Misplaced
------------------------
expanded:       543.39
treeDepth:      11.37
execTime (ms): 4.61750079
------------------------
H2: Manhattan Distance
------------------------
expanded:       151.51
treeDepth:      11.37
execTime (ms): 0.44277134
************************
```

The top section specifies parameters used to generate the initial states from the puzzleConfig. The lower section provides the following performance data for both heuristics:

- **expanded** - average number of states that were expanded when running the search.
- **treeDepth** - average depth of the tree from the solution to the initial node.
- **execTime (ms)** - average execution time per puzzle, in milliseconds.

Four-Phase Problem Solving Design

1. Goal Formulation

The program (agent) formulates goal at the beginning of the search when generating a goal state of the appropriate puzzle size.

2. Problem Formulation

An agent can progress from the initial state by making moves, which generate subsequent puzzle states that can bring the agent closer to or farther from the goal. In the current state, agent can move one of the adjacent tiles in place of the empty tile. This action generates the next puzzle state, and this is the only action that can influence the environment.

3. Search

At each step in the search agent checks if the current state is the goal state. If it is not, agent considers all possible (up to 4) states it can reach by moving one of the tiles into the empty space. It makes a choice on which state to consider next based on the evaluation function value for that state which consists of the path cost and heuristic. Path cost is the number of moves it took the agent to reach that state. Heuristic is an evaluation of the state based on the knowledge of the game and of what features of the state make it closer to the goal state, for example, the number of misplaced tiles in the puzzle.

4. Execution

Once the goal state is reached, the program is ready to produce a solution. This is done by using the Validator class, which walks backwards from the goal state following the parent references of each state in the solution path. This lets the agent construct a sequence of states that will take it from the initial to the goal state, and can be followed after the search is completed.

Heuristics Design

Heuristic is a function that gives the agent an estimate of how close the state is to the goal. The program implements two types of heuristics: number of misplaced tiles and Manhattan distance.

H1: Number of misplaced tiles

This is the simplest heuristic. It counts how many tiles in the puzzle are out of place. The smaller the number, the closer this particular state is to the goal state.

H2: Manhattan distance

This heuristic is more sophisticated but still simple. It computes the block, or Manhattan, distance from the current position of a tile to its correct position. Again, the smaller is the sum of distances from the current positions of tiles to the final ones, the closer that states is to the goal.

In the next section, performance of both of these heuristics will be compared.

Performance Results

Analysis was performed by running large numbers of puzzles of increasing randomness (complexity), and comparing the performance of h1 and h2 based on three key features:

Search Cost (Nodes Expanded)
Each node that was expanded during execution was counted. This measure gives us a good idea of how many attempts the agent made to find the goal, and what is the space complexity.

Effective Branching Factor
It is a quantity computed using the formula:

$$N + 1 = 1 + b + b^2 + \ldots + b^d$$

where N is the number of nodes expanded, d is the solution depth, and b is the branching factor itself. In combination with empirical evidence, this is a powerful tool for the analysis of usefulness of the heuristic.

Execution Time
Time it took for an agent to solve the given puzzle in milliseconds. Time is the average time to solve a single puzzle across all instances.

8-Puzzle Performance Analysis

The tree key features described above for each heuristic and solution depth (complexity) of a puzzle are summarized in Table 1. For the analysis I generated 100 random initial states, where the number of swaps during puzzle generation gave the desired solution depth. Raw data can be found in the file 8-puzzleReportLog.txt in the docs folder.

	Search Cost (Nodes Expanded)		Effective Branching Factor		Execution Time (ms)	
d	**h1 (N Misplaced)**	**h2 (Manhattan)**	**h1 (N Misplaced)**	**h2 (Manhattan)**	**h1 (N Misplaced)**	**h2 (Manhattan)**
6	25.8	17.2	1.44	1.31	0.180	0.120
8	90.8	33.2	1.55	1.32	0.718	0.171
10	334	80.2	1.63	1.37	3.10	0.269
12	748	176	1.60	1.39	9.36	0.616
14	1915	400	1.60	1.41	44.6	1.81
16	4695	760	1.60	1.40	477	5.44
18	8866	1419	1.57	1.40	919	15.7
20	14367	2257	1.53	1.38	1675	33.9
22	25087	2873	1.51	1.35	3725	41.5

Table 1. 8-Puzzle Average Heuristic Performance Comparison

We can readily see that for all three features, h2 consistently demonstrates higher performance. We may further explore this by plotting each feature separately, as is done in Figures 2-4. Please note that Figures 2 and 4 use logarithmic scales.

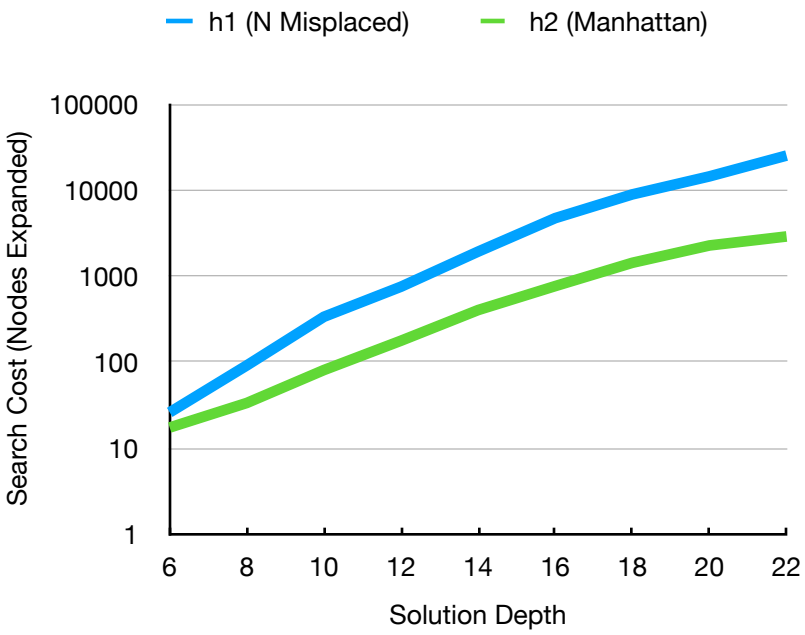

Figure 2. 8-Puzzle Search Cost (Nodes Expanded) for h1 vs h2.

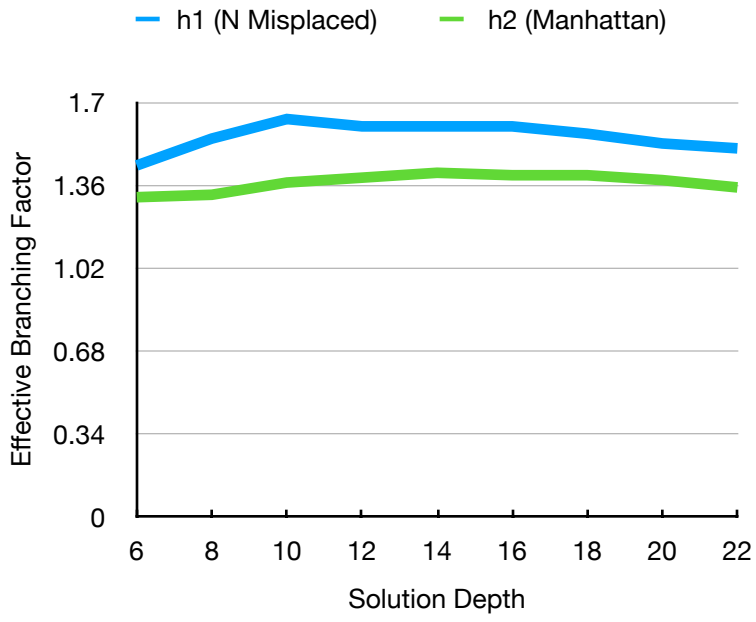

Figure 3. 8-Puzzle Effective Branching Factor for h1 vs h2.

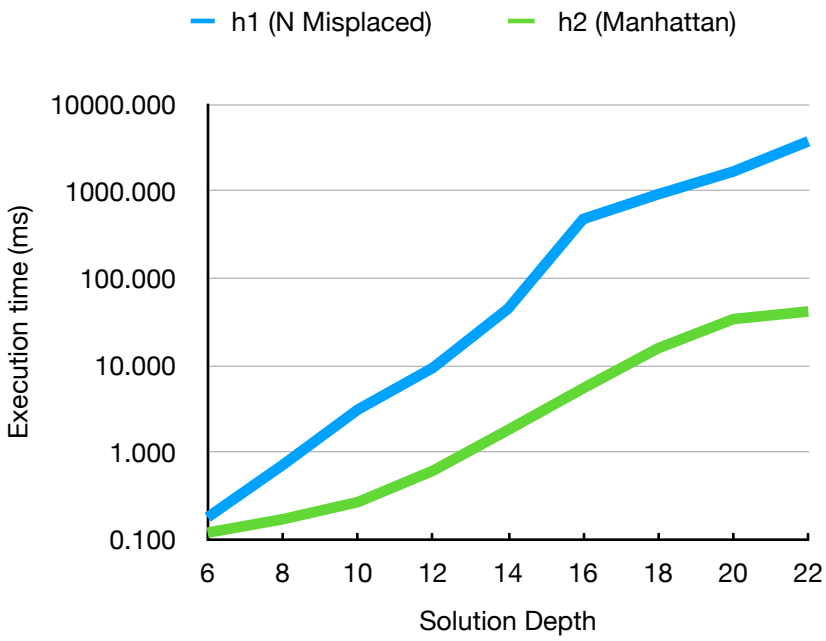

Figure 4. 8-Puzzle Execution time (ms) for h1 vs h2.

We see that indeed, search cost increases much faster for h1, and is never smaller than h2. Difference in time complexity is the most dramatic with h1 becoming exponentially slower, much slower than h2. Figure 3 also confirms that for every solution depth, h1 always has larger effective branching factor, indicating a greater utility of h2.

3-Puzzle Performance Analysis

Below I also analyzed the performance of the program on solving 3-puzzle instances, with results summarized in Table 2. As in the previous example, for each solution depth 100 random puzzle instances were generated. Raw data can be found in the 3-puzzleReportLog.txt file in the docs folder.

	Search Cost (Nodes Expanded)		Effective Branching Factor		Execution Time (ms)	
d	**h1 (N Misplaced)**	**h2 (Manhattan)**	**h1 (N Misplaced)**	**h2 (Manhattan)**	**h1 (N Misplaced)**	**h2 (Manhattan)**
1	1.99	1.95	1.99	1.95	0.0510	0.0385
2	3.74	3.13	1.49	1.34	0.0701	0.0498
3	6.20	4.58	1.41	1.23	0.0648	0.0432

Table 2. 8-Puzzle Average Heuristic Performance Comparison

We see that as for the 3-puzzle, h2 shows consistently better performance than h1.

15-Puzzle Performance Analysis

Below is the analysis of the performance of the program on solving 15-puzzle instances. Summary is provided in Table 3. As in the previous example, for each solution depth 100 random puzzle instances were generated. Raw data can be found in the 15-puzzleReportLog.txt file in the docs folder. Because of the increased complexity of the puzzle, problems which required solution depths greater than 14 took too much time.

	Search Cost (Nodes Expanded)		Effective Branching Factor		Execution Time (ms)	
d	**h1 (N Misplaced)**	**h2 (Manhattan)**	**h1 (N Misplaced)**	**h2 (Manhattan)**	**h1 (N Misplaced)**	**h2 (Manhattan)**
5	11.3	10.2	1.28	1.25	0.111	0.101
6	18.4	15.8	1.33	1.29	0.176	0.148
7	31.2	21.3	1.38	1.28	0.222	0.149
8	33.3	27.7	1.32	1.28	0.289	0.216
9	94.0	43.0	1.46	1.31	0.644	0.266
10	151	71.5	1.49	1.35	1.15	0.315
11	858	195	1.71	1.46	35.3	1.53
12	1733	142	1.73	1.36	317	0.535
13	12632	547	1.96	1.49	40741	32.0
14	5156	664	1.73	1.47	1465	28.2

Table 3. 15-Puzzle Average Heuristic Performance Comparison

From this data we can see that h2 demonstrates better performance compared to h1. This is more clearly visible in charts below, Figures 5-7. There is an outlier at the solution depth 14 because one problem instance turned out particularly challenging for the agent. Please note that Figures 5 and 7 use logarithmic scales.

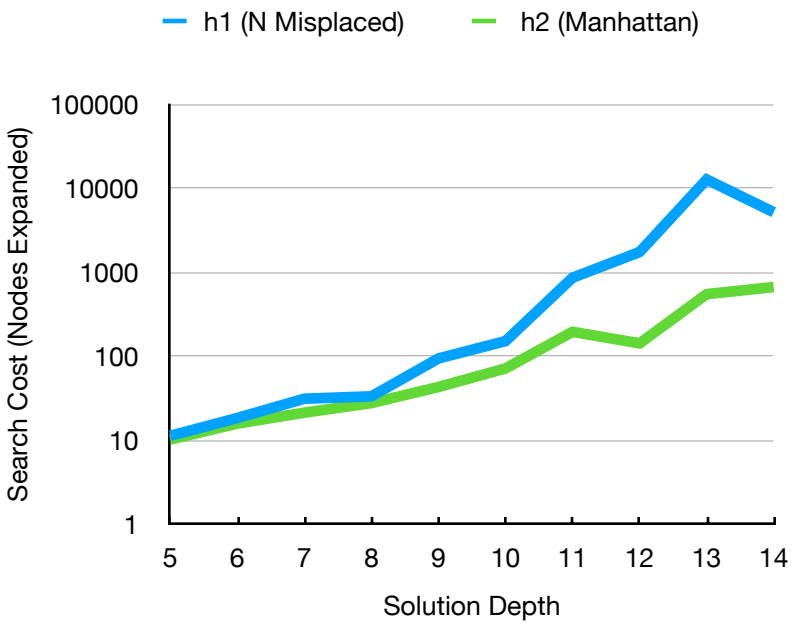

Figure 5. 15-Puzzle Search Cost (Nodes Expanded) for h1 vs h2.

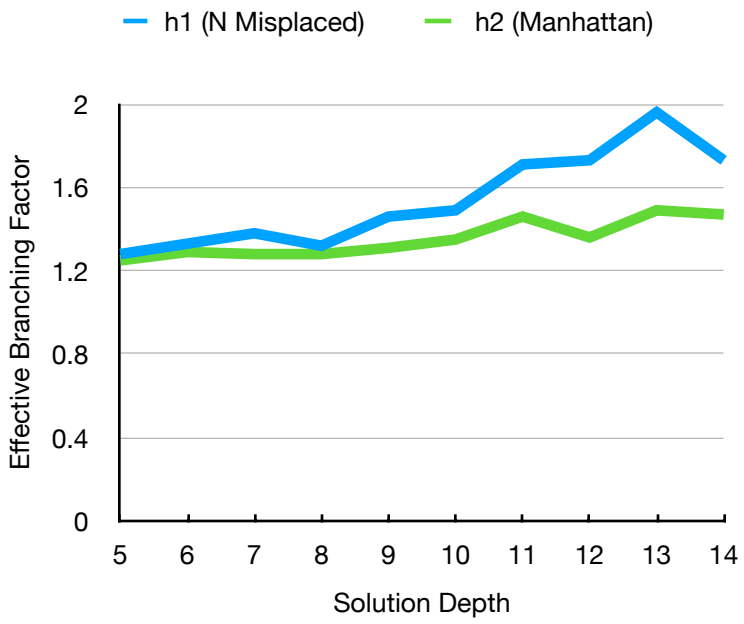

Figure 6. 8-Puzzle Effective Branching Factor for h1 vs h2.

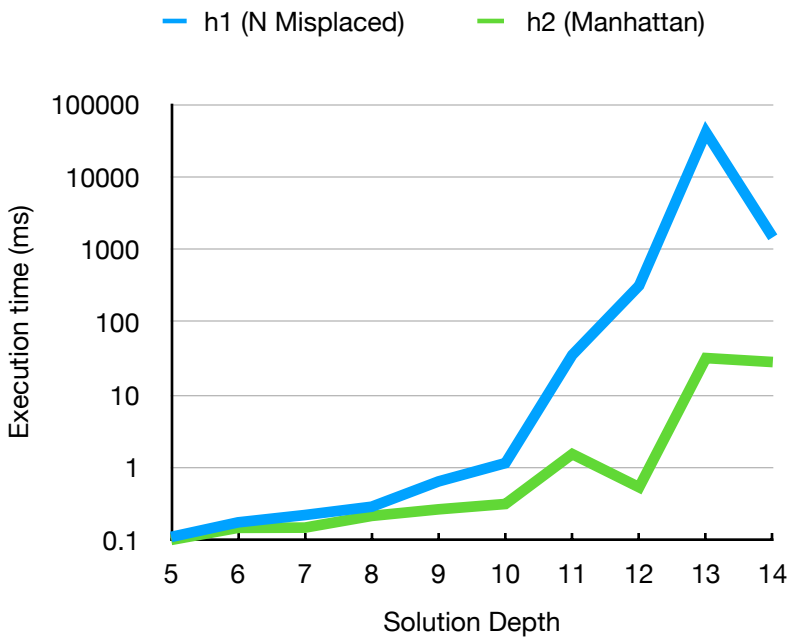

Figure 7. 8-Puzzle Execution time (ms) for h1 vs h2.

It is noticeable that in Figures 5 and 7, at some intervals, space and time complexity of h1 demonstrate increase rates exceeding exponential. This suggests that for large problems, h1 fails to provide sufficient information to make the problem tractable. Heuristic h2 is not very good either, however it does a substantially better job. This is well-reflected in the values of the effective branching factor. It remains fairly stable for h2, while it grows larger and larger for h1. This shows that h2 is much better for solving 15-puzzles for all solution depth between 5 and 14.

Results Summary

As it is visible from the data generated by the agent when solving 3-, 8-, and 15-puzzles, h2 consistently demonstrates better performance than h1 in all three key metrics: nodes expanded (space complexity), execution time (time complexity), and effective branching factor. The difference exacerbates with the size of the puzzle.
The agent was able to successfully solve 8-puzzle problems of high complexities. 15-puzzle quickly became intractable, and could be solved only with a limited number of swaps of the initial state. Heuristic h1 soon began providing insufficient information. As a result, the agent kept getting stuck and expanding thousands of nodes before finally arriving at the goal state.
Since our data shows that for all considered problem sizes and for varying complexity of the puzzle h2 demonstrates petter performance in all key metrics, especially in the effective branching factor, we have enough evidence to conclude that the Manhattan distance heuristic dominates the number-of-misplaced-tiles heuristic.