An Analysis of an A* Sliding Puzzle Solver

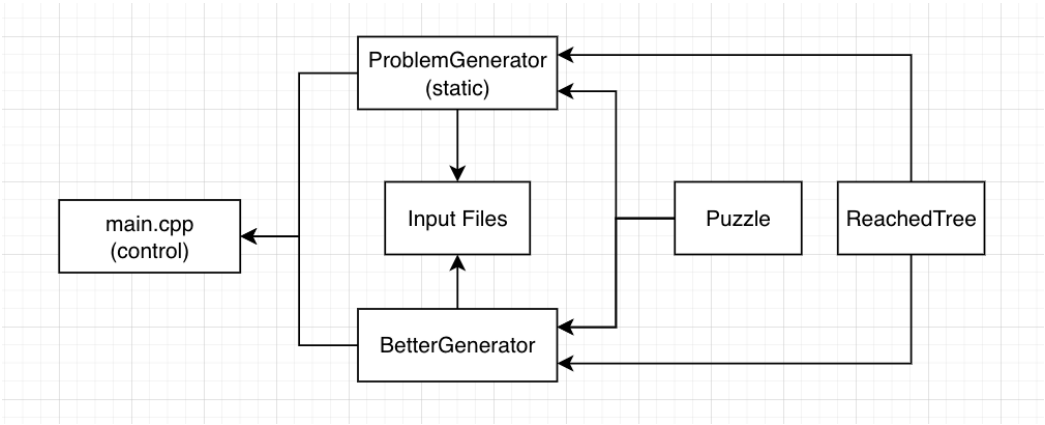
🎀 Claire Liu, CS 420: Artificial Intelligence, Lafayette College, Fall 2022 🎀

👩‍💻Program Design

This project was written in C++.
Project File Organization:

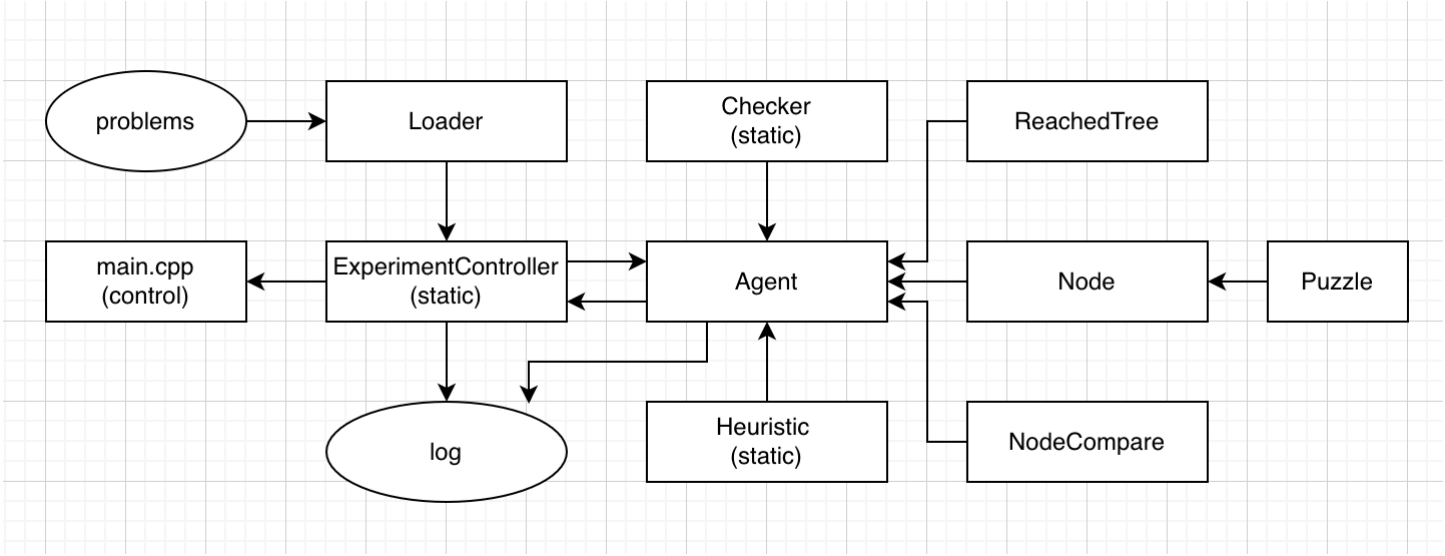
```
|---AI_Project1 (Project root folder)
    |---log (output files)
    |---problems (problem generation code and input files)
    |---main project code
```

The problem generation code follows the following architecture:



- **main.cpp** is the file that drives the problem generation
- **ProblemGenerator** is a static class that can be used to generate every permutation of 3-puzzles and every permutation of 8-puzzles, solvable or unsolvable.
- The **BetterGenerator** class can be used to generate solvable, unique and relies on the Puzzle and ReachedTree classes to do so.
- Problems generated by the ProblemGenerator or BetterGenerator classes are dumped into **input files** that live in the same directory.
- The **Puzzle** class encapsulates the vector representation of a sliding puzzle and all the legal moves.
- The **ReachedTree** is essentially a dictionary for looking up vectors (puzzles) to see if this puzzle was generated before. This prevents duplicate states in problem generation.

The main project code follows the following architecture:



- **main.cpp** is the file that drives the experiment
- **ExperimentController** is a static class where experiment parameters (which input files, which heuristic to use) can be configured
- The **Loader** class takes files of problems and parses them, reading them into vectors. These problems are then fed to the ExperimentController
- The **Agent** class is where the actual A* Search is done, which is phase 3 of the problem solving process.
- The static **Heuristic** class calculates h(n) for each of the heuristics and supplies it to the Agent
- The **Node** class represents a state, including a Puzzle, parent nodes, children nodes, estimated costs, and depth.
- The **NodeCompare** supports the priority queue of nodes within the Agent to overload the compare for the min priority queue.
- The **Puzzle** class encapsulates the vector representation of a sliding puzzle and all the legal moves.
- The **ReachedTree** is essentially a dictionary for looking up vectors (puzzles) to see if this puzzle was generated before. This prevents duplicate states in problem generation
- The **Checker** static class covers phase 4 of the problem solving process, executing the path proposed by the Agent's search and validating that it's a solution.
- The ExperimentController outputs averaged results to the **log** folder. The Agent outputs problem by problem data to files in the log as well.

🛠️ User Manual

To run the program, first generate problems by running the following:

```
cd problems
make three // generates every three puzzle, solvable and unsolvable
make eight // generates 100 solvable 8-puzzles each for 6, 7, and 8 misplaced tiles
make fifteen // generates 30 solvable 15-puzzles for 10, 11, and 12 misplaced tiles
```

This will generate files in the problems folder.

You can also run `make clean` to remove .o files and the executable `./generate`

The problems file have the following naming conventions:

- For 3-puzzles, the file is called `every_three_puzzle.txt`
- For 8-puzzle and 15-puzzles, the file is called `test<edgeSize>_<#OfMisplacedTiles>.txt`

Problems files have the following format:

```
<edgeSize> <numberOfProblems>
// ....
// followed by one problem per line
//...
```

To run the experiment, run the following commands.

```
cd .. // to project root folder (AI_Project1)
make three // solve every three puzzle using every heuristic
make eight // solve 8-puzzles for 6, 7, and 8 misplaced tiles using every heuristic
make fifteen // solve 15-puzzles for 12 and 15 misplaced tiles using every heuristic.
Please note that running make fifteen results in a REALLY high runtime.
```

These will generate log files in the log folder. One log file is generated for every puzzle level (# misplaced tiles) and every heuristic. The file is overwritten each run. The naming convention is `results<edgeSize>_<#OfMisplacedTiles>_<Heuristic>.txt`

Log files files output the following all on one line for each puzzle solved:

```
timeToSolve averageBranchingFactor  averageFrontierSize     totalNumberOfNodes
solutionDepth     deepestDepth      solutionVerifiedbyChecker    solved (T/F)
```

One file in the averages folder is also generated for each puzzle level. The file is NOT overwritten each run, results are appended. The naming convention is `results<edgeSize>_<#OfMisplacedTiles>.txt` Averages files output the following on each line:

```
heuristic   averageTimeToSolve      averageBranchingFactor  averageFrontierSize
averageNumberOfNodes    averageParentNodes      averageSolutionDepth    b*
```

Additionally, you can also run the following command in the project root folder

```
make clean
make tests // outputs to testHeuristics.txt in problems folder
make cleanLog // deletes all .txt files in the log
```

Program code can also be found on github at https://github.com/lee-anh/AI_Project1, please email me at liucl@lafayette.edu if you'd like permissions.

🐣 Theory: Four-Phase Problem Solving Process & Formal Definition

Four-Phase Problem Solving Process

The Sliding Tile Problem operates in an episodic, single agent, fully observable, deterministic, static, discrete, known environment. Since the environment is known, the agent can follow the four-phase problem solving process.

1. **Goal Formulation**

The agent adopts the goal of reaching the goal state in which the tiles are organized in ascending order, from left to right, to bottom, with the blank tile in the top left corner.

2. **Problem Formulation**

The agent devises an abstract model for the world, a grid world where sliding tiles are represented by swaps. The two-dimensional nature of the n-puzzle is further abstracted to a one dimensional vector of size n+1 for any n-puzzle. Using a simple modulus conversion to calculate the 2d index from the 1d index, the agent can perform swaps that would represent the tiles moving up, down, left, and right.

3. **Search**

Before taking action, the agent simulates each possible move (up, down, left, right) for each possibility. It checks to see if the move is legal and uses an A* search to estimate the cost of taking certain actions. Since the heuristics we are using in the A* search are admissible, theoretically only the states that have greater chance of leading to the optimal solution are explored.

4. **Execution**

The agent can now execute the action sequence it decided on. The Checker class takes care of this.

Formal Definition

The formal definition of an n-puzzle problem is as follows:

State space:
A state includes the arrangement of the tiles, a reference to the parent state that resulted in the state, the action that led from the parent state to the current state, the number of moves it took to get to the current state from the initial state (which is essentially the depth), and the estimate of the cost of taking the state (f(n)).

In implementation, each state is represented by a Node with the puzzle itself represented by a 1d vector, with a simple modulus equation mapping the 1d index to the 2d index.

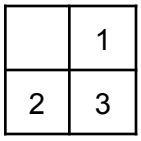
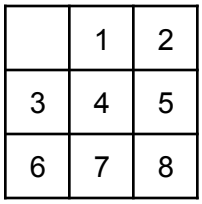
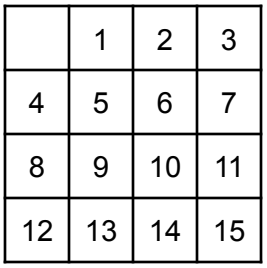
The state space is represented by a tree where each node has at most 4 children (results of moving up, down, left, and right) and includes every possible arrangement of the puzzle within the sliding puzzle rules (you can only slide tiles into the blank space to rearrange, you cannot just swap any two tiles).

Initial State:

A random permutation of integers {-1, 1, 2, …., n-1, n}, representing a random arrangement of the tiles. -1 is included to represent the blank space. Note that not every permutation is solvable. Since it's the inital state, it has no parent state, no previous action, and a depth of 1.

Goal States:
For every n puzzle, there is one goal state. The top left corner is where the blank space should be, and the tiles should be arranged in ascending order, left to right, top to bottom.

In implementation, this is simply the vector in sorted order, ascending.

3-Puzzle	8-Puzzle	15-Puzzle
		

Goal states for 3-Puzzle, 8-Puzzle, and 15-Puzzle.

Applicable actions:
Actions(Puzzle) = {MoveDown, MoveUp, MoveLeft, MoveRight}
Move Down is where the blank moves down, so the tile below the blank slides up.
Move Up is where the blank moves up, so the tile above the blank slides down.
Move Left is where the blank moves to the left, so the tile left of the blank slides right.
Move Right is where the blank moves to the right, so the tile right of the blank slides left.

Not every action is applicable to every state. The agent should test to see if an action is applicable before exploring the possibility to avoid segmentation faults in implementation. In implementation these are simply swaps of array elements.

Transition model:
The state resulting from taking an action will result in the blank moving to the specified position of the action (ex. *Move Down* will result in the blank tile moving down and swapping places with the tile that is below it)
Result(Puzzle, MoveDown) = $Puzzle_d$
Result(Puzzle, MoveUp) = $Puzzle_u$
Result(Puzzle, MoveLeft) = $Puzzle_l$
Result(Puzzle, MoveRight) = $Puzzle_r$

Action cost function:
Action-Cost(s, a, s′) = 1 for any given transition, since the cost of a move is simply one unit.

🔭 Heuristics

An A*Search uses the following formula to estimate f(n), the cost of a state: f(n) = g(n) + h(n) where g(n) is the path cost from the initial state to node n (# of moves = the depth of the node) and h(n) is the estimated cost of the shortest path from n to a goal state (the heuristic). In this project, we considered using 4 different heuristics: Misplaced Tiles, Manhattan Distance, Swaps in MAXSORT, and Euclidean Distance.

h1: Number of Misplaced Tiles

The number of misplaced tiles (blank not included) from the goal state. h1 is an admissible heuristic because any tile that is out of place will require at least one move to get it to the right place. h1 is not consistent because a successor state generated from an applicable action could result in an additional tile being misplaced.

h1 was implemented in the static Heuristic class and simply iterates through the puzzle vector and counts up the number of tiles that are misplaced. The run time for h1 is linear.

```
float Heuristic::misplacedTiles(Puzzle* puzzle) {
 int count = 0;
 for (int i = 0; i < puzzle->getDimension(); i++) {
   if (puzzle->getPuzzleArray().at(i) != i) {
     count++;
   }
 }
 // subtract 1 from the total because blank will always be out of place, don't count it
 return (count - 1);
}
```

h2: Manhattan Distance

The sum of the distance of the tiles from their goal positions, where the distance is the sum of the horizontal and vertical distances. h2 is an admissible heuristic because a move can only move the tile one step closer to the goal. To move a tile from its current position to its goal position would of course take more moves than what the Manhattan Distance estimates because we only have one blank tile. h2 is not consistent because a successor state generated from an applicable action could result in a tile moving farther away from its goal position.

h2 was implemented in the static Heuristic class and iterates through the puzzle vector, calculating the Manhattan distance for each tile using some utility functions to convert the 1d vector indices to 2d vector indices and adding it to the sum, which is returned at the end. The runtime for h1 is linear.

```
float Heuristic::manhattanDistance(Puzzle* puzzle) {
 int sum = 0;
 for (int i = 0; i < puzzle->getDimension(); i++) {
   int tileNumber = puzzle->getPuzzleArray().at(i);
   if (tileNumber != -1) {  // don't count the blank tile
     pair<int, int> tileCoordinate = puzzle->getTwoDimensionIndexFromOneDimension(tileNumber);
     pair<int, int> target = puzzle->getTwoDimensionIndexFromOneDimension(i);
     int xDiff = abs(tileCoordinate.first - target.first);
     int yDiff = abs(tileCoordinate.second - target.second);
     sum += xDiff + yDiff;
   }
```

```
 }
 return sum;
}
```

h3: Swaps in MAXSORT

The number of swaps needed to sort the array according to the MAXSORT algorithm, which is a modification of selection sort. MAXSORT is an admissible heuristic sinceMisplaced Tiles is already an admissible heuristic and there are always at most Misplaced Tiles number of swaps for the MAXSORT algorithm. However, MAXSORT has some disadvantages to the other algorithms. First, MAXSORT is a quadratic algorithm, so it has slow runtime. Additionally, the MAXSORT algorithm tends to underestimate the costs too much. MAXSORT sometimes swaps more than one tile into place resulting in an estimate that is at time lower than the Misplaced Tiles estimates. This could mislead the agent into exploring non-optimal solutions. h3 is not consistent because a successor state generated from an applicable action could result in an additional tile being misplaced, resulting in more swaps.

h3 was implemented in the static Heuristic class and follows the quadratic algorithm specified in the project description.

```
float Heuristic::maxSort(Puzzle* puzzle) {
 int numberOfSwaps = 0;
 vector<int> puzzleArr = puzzle->getPuzzleArray();
 for (int i = (puzzle->getDimension() - 1); i >= 1; i--) {
   int max = 0;
   for (int j = 1; j <= i; j++) {
     if (puzzleArr.at(max) < puzzleArr.at(j)) {
       max = j;
     }
   }
   // swap if out of place
   if (puzzleArr.at(max) != puzzleArr.at(i)) {
     int temp = puzzleArr.at(max);
     puzzleArr.at(max) = puzzleArr.at(i);
     puzzleArr.at(i) = temp;
     numberOfSwaps++;
   }
 }
 return numberOfSwaps;
}
```

h4: Euclidean Distance

The sum of the distance of the tiles from their goal positions, where distance is the real valued Euclidean distance from a tile's current location to the final goal state location. The Euclidean distance is an admissible heuristic because a tile's Euclidean distance will always be less than its Manhattan distance (because of the Pythagorean Theorem) so since the Manhattan distance heuristic is admissible, then the Euclidean distance heuristic is admissible as well. h4 is not consistent because a successor state generated from an applicable action could result in an additional tile being misplaced, resulting in a higher euclidean distance sum.

h4 was implemented in the static Heuristic class and iterates through the puzzle vector, calculating the euclidean distance from a tile's current location to the final goal location using the distance formula and adding it to the sum, which is returned at the end.

```
float Heuristic::geometricDistance(Puzzle* puzzle) {
float sum = 0;
for (int i = 0; i < puzzle->getDimension(); i++) {
  int tileNumber = puzzle->getPuzzleArray().at(i);
  if (tileNumber != -1) {  // don't count the blank tile
    pair<int, int> tileCoordinate = puzzle->getTwoDimensionIndexFromOneDimension(tileNumber);
    pair<int, int> target = puzzle->getTwoDimensionIndexFromOneDimension(i);
    int xDiff = abs(tileCoordinate.first - target.first);
    int yDiff = abs(tileCoordinate.second - target.second);
    // distance formula
    sum += sqrt(pow(xDiff, 2) + pow(yDiff, 2));
  }
}

return sum;
}
```

📊 Program Performance and Analysis

The following performance measures were considered for the agent:

- Average time to solve
- Average solution depth
- Average number of nodes explored
- Average branching factor
- Approximate effective branching factor, b*

Considering the average time to solve the puzzle gives a broad overview of the performance of a heuristic. Average frontier size, average number of nodes explored, and approximate b* shed light into why the heuristic performed the way it did.

The average time to solve will give a sense of the time complexity involved in solving for

The effective branching factor, b*, is found from solving for b* from the equation $N+1 = 1 + b^* + (b^*)^2 + \ldots. + (b^*)^d$ where N is the total number of nodes generated by a A* search, d is the solution depth and b* is the branching factor that a uniform tree of depth d would have to have in order to contain N+1 nodes. According to the textbook, a well designed heuristic has a b* close to 1. Solving for b* can be tricky, so an approximation $b^* = N^{1/d}$, proposed by Sentry on stackoverflow, was used in this analysis.

Average branching factor is pretty consistent for sliding puzzles because you only ever have 4 applicable actions, so while this was recorded, it did not shed light on differences in heuristic performance.

3-Puzzles

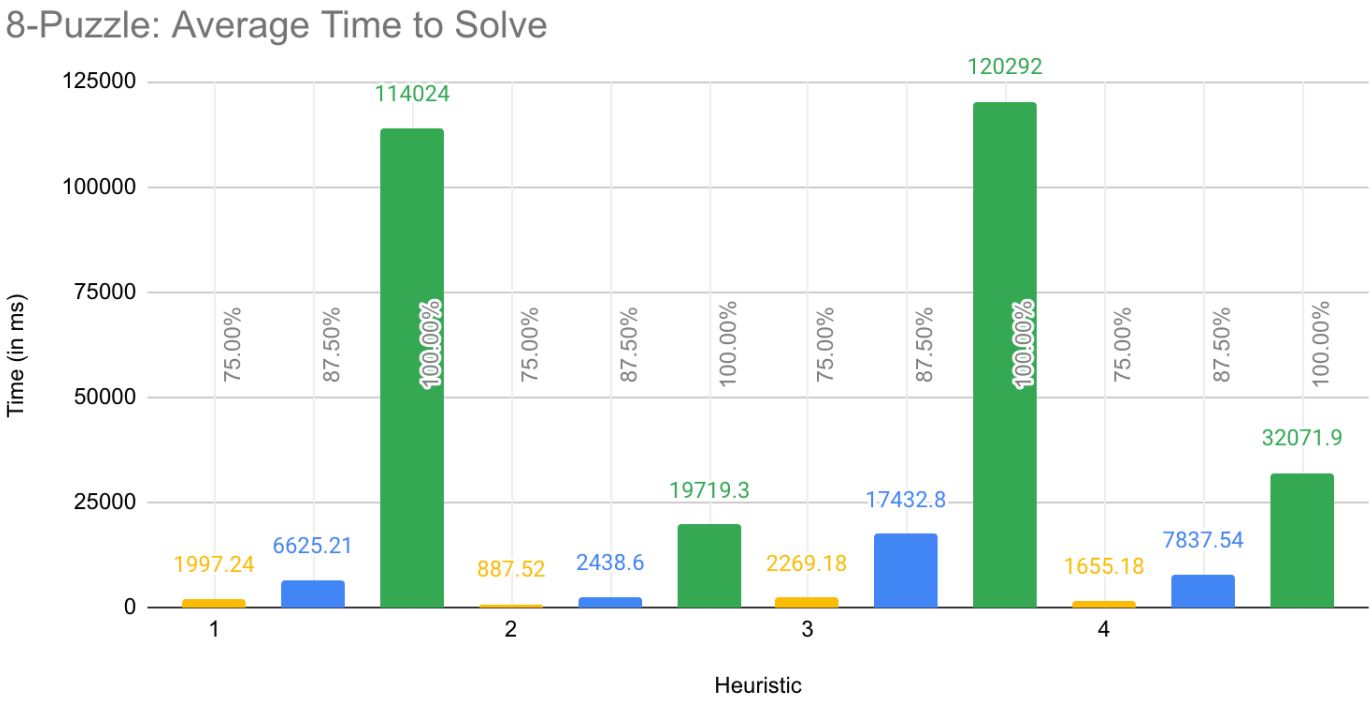
Solving the 3-Puzzle is pretty trivial and there are only 24 possible permutations of a 3-Puzzle, so I had the agent try to solve all of them, whether the puzzle was solvable or not. The results of the experiment are in the table below.

Heuristic	averageTimeToSolve	averageBranchingFactor	averageNumberOfNodes	AverageSolutionDepth	b*
1	94.9583	0.958333	9.91667	4.5	1.665
2	112.833	0.958333	9.66667	4.5	1.65558
3	96.875	0.958333	10.1667	4.5	1.67424
4	104.083	0.958333	9.91667	4.5	1.665

It's evident that all the heuristics performed roughly the same across all the performance measures, with minor differences because of the computational complexity of the different heuristics, but the state space is only 24 states maximum, so even if the puzzle didn't have heuristics to guide it, it would be trivial to solve the puzzle brute force. Also, from the raw data, 50% of the 24 puzzles were solvable, and the rest unsolvable for the given goal state.

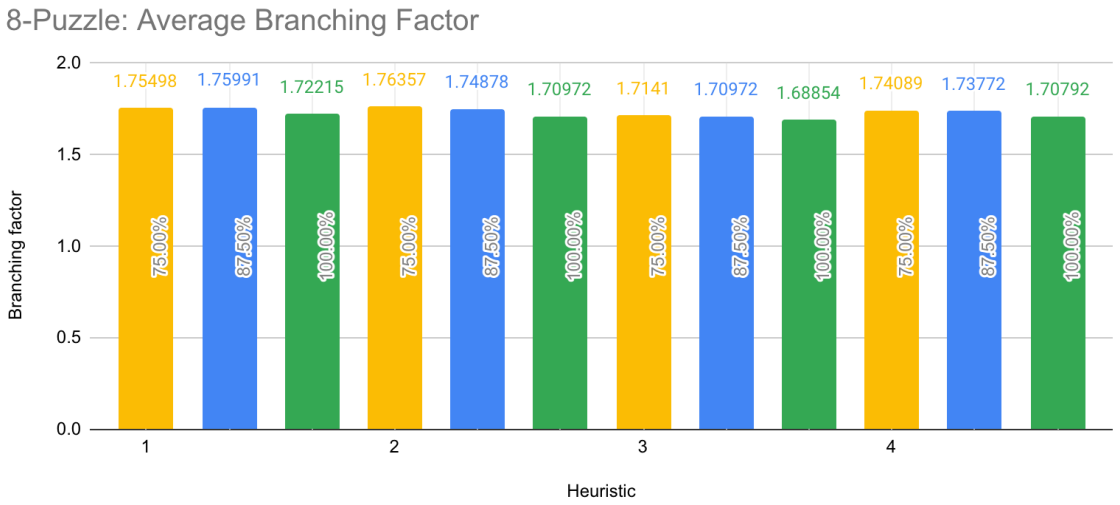
8-Puzzles

For 8-Puzzles, the agent solved 100 8-Puzzles where 75% of the tiles (6 tiles) were out of place, 100 8-Puzzles where 87.5% of the tiles (7 tiles) were out of place, and 100 8-Puzzles where 100% of the tiles (8 tiles) were out of place. All of the problems the agent solved were guaranteed to be solvable.

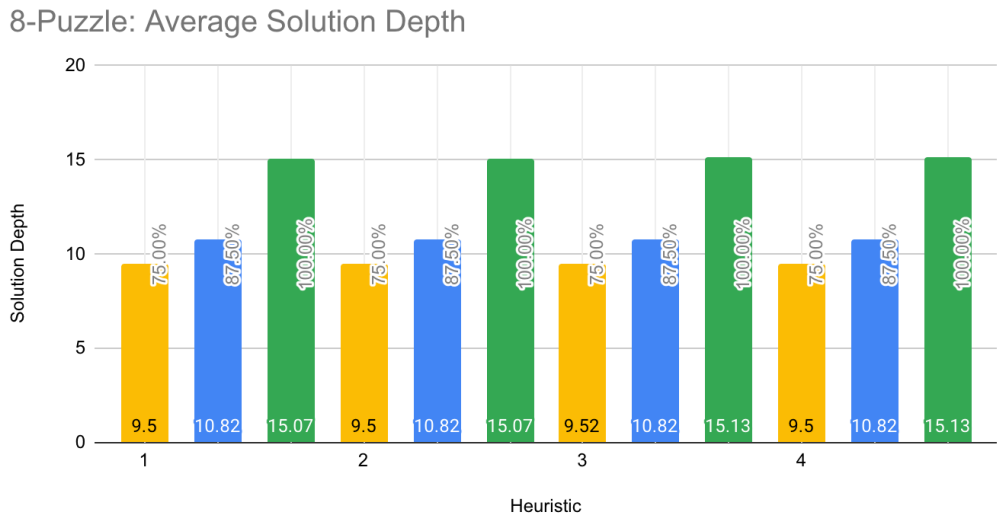


The average time to solve 8-puzzles reveals that having all of the tiles out of place exponentially increases the runtime. We can also see that the heuristics are following the following dominance relationship for time to solve across all percentages sorted: h2 > h4 > h1 > h3 with h1 and h3 taking significantly longer to solve the puzzles.

This is an early indication that h1 and h3 may be guiding the agent to explore paths that aren't optimal while h2 and h4 keep the agent focused on paths that are more optimal.

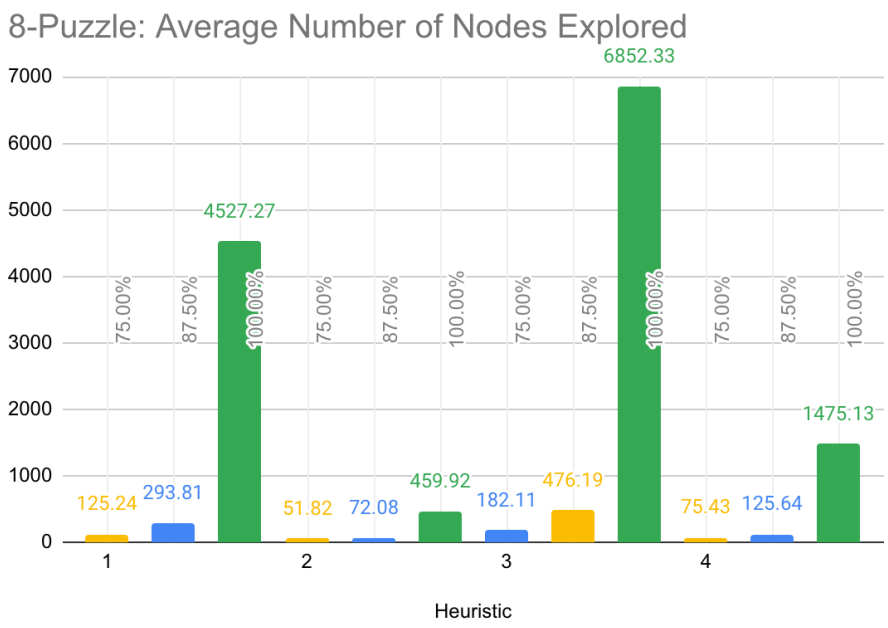


The average branching factor is pretty consistent across the board for the heuristics. This makes sense because creating the children nodes are independent of the heuristic. A child is created if a move is valid. The variations are attributed to the heuristic leading the agent on slightly different paths. It is also logical that the branching factor for 8-puzzles is higher than for 3-puzzles because 3-puzzles have more limited options for movement than 8-puzzles.

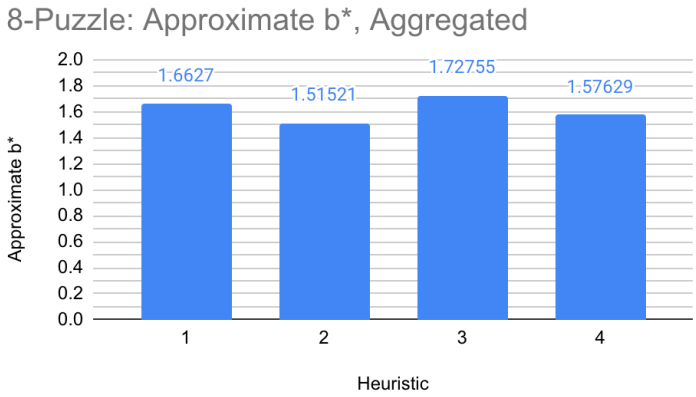


The solution depths are relatively consistent across heuristics and it is logical that the solution depth would increase as the percentage of tiles out of place increases because the problem has increasing difficulty and a greater number of moves would be needed to slide the tile into the goal state.

There appears to be slightly higher depths for h3 and h4, possibly because of some edge cases. This indicates that h3 and h4 may be leading the agent to not find the optimal solution, but perhaps a solution that is close enough. In general though, the heuristics produce solution depths that are relatively consistent across the board.



As the puzzle difficulty increases, the average number of nodes explored increases exponentially. This is because as the solution depth increases, the heuristic is not as accurate higher in the search tree, so it may mislead the agent from the optimal solution, causing the agent to explore a greater number of paths and a greater percentage of the state space. We can see the dominance relationship of h2 > h4 > h1 > h3 clearly established across the board in terms of the average number of nodes explored.



The approximate b* varied slightly across the problem difficulties because of the variety of sample puzzles used. Results were aggregated to form a better sense of the b* for each heuristic. From the approximate effective branching factors, we can see that h2 is the closest to 1, which would be an optimal heuristic. The dominance relationship based on b* would be h2 > h4 > h1 > h3.

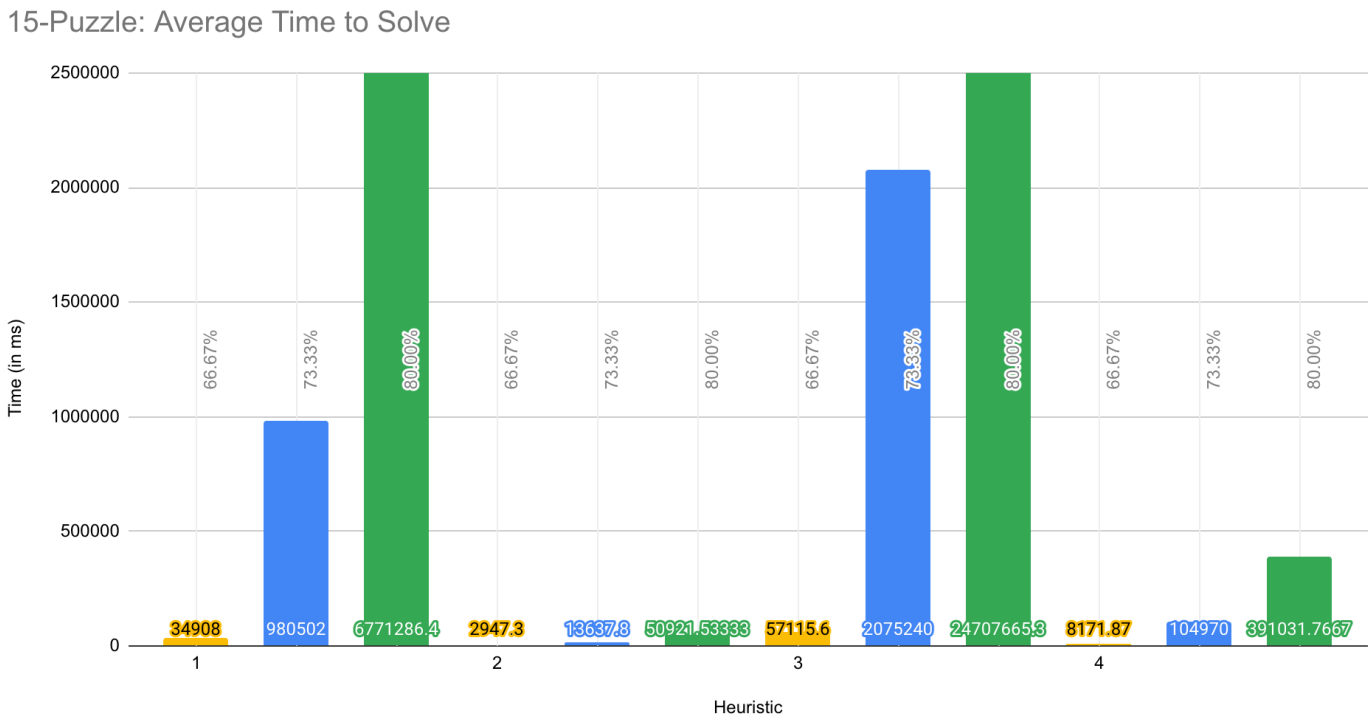
In summary, the average branching factor and the average solution depth was consistent across heuristics, but for the time to solve, the average nodes explored, and the approximate b*, the dominance relationship of h2 > h4 > h1 > h3 was evident.

15-Puzzles

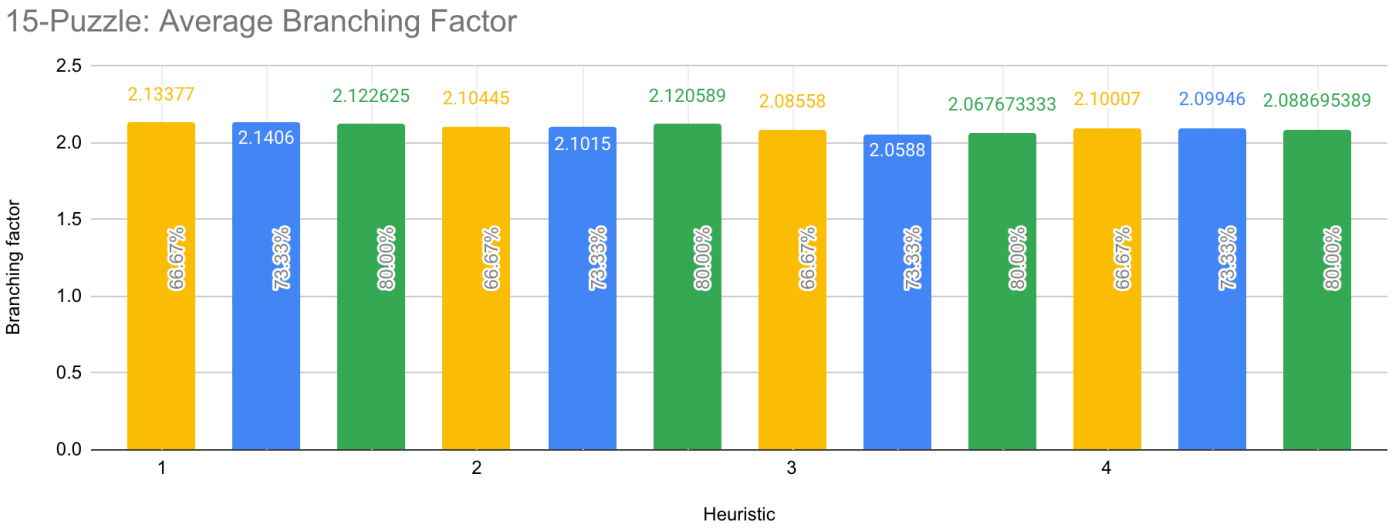
15-Puzzles were much harder to solve than the 8 puzzles. Therefore, the agent was limited to solving only problems that were solvable.

On puzzles where more than 12 tiles were out of place, the agent started to run into some serious difficulties with finding the solution in a reasonable amount of time. This was because of memory limits and the frontier/explored states could not all fit into memory so the system encountered thrashing issues.

Therefore, for 15 puzzles the agent solved 30 puzzles where 66% of the tiles (10 tiles) were out of place, 30 puzzles where 73% of the tiles (11 tiles) were out of place, and 30 puzzles where 80% of the tiles (12 tiles) were out of place. The number 30 was chosen so that a statistically significant sample size was used.



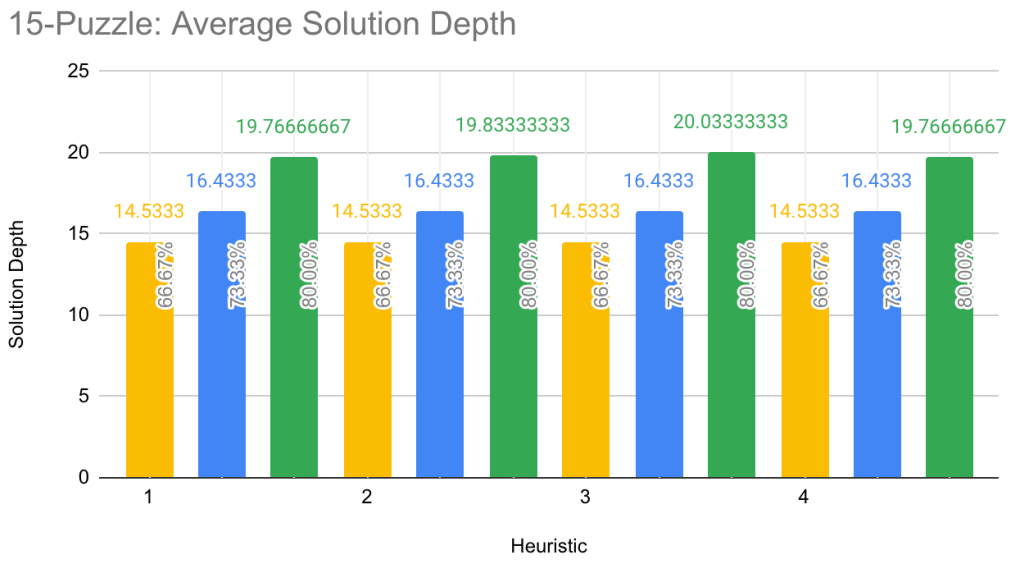
The average time to solve 15-puzzles was quite high across the board, and especially high from h1 and h3 when the puzzle was 80% out of place (it was so high that the actual value is cut off from the graph). The extremely high runtime indicates some signs of thrashing due to a large state space and a large percentage of the state space being explored. We can also see that the heuristics are following the following dominance relationship for time to solve across all percentages sorted quite dramatically: h2 > h4 > h1 > h3. This is in agreement with the results from the 8-puzzles.



Like in the 3-puzzle and the 8-puzzle, the average branching factor is pretty consistent across the board for the heuristics because creating the children nodes are independent of the heuristic. A child is created if a move is valid. The variations are attributed to the heuristic leading the agent on slightly different paths.

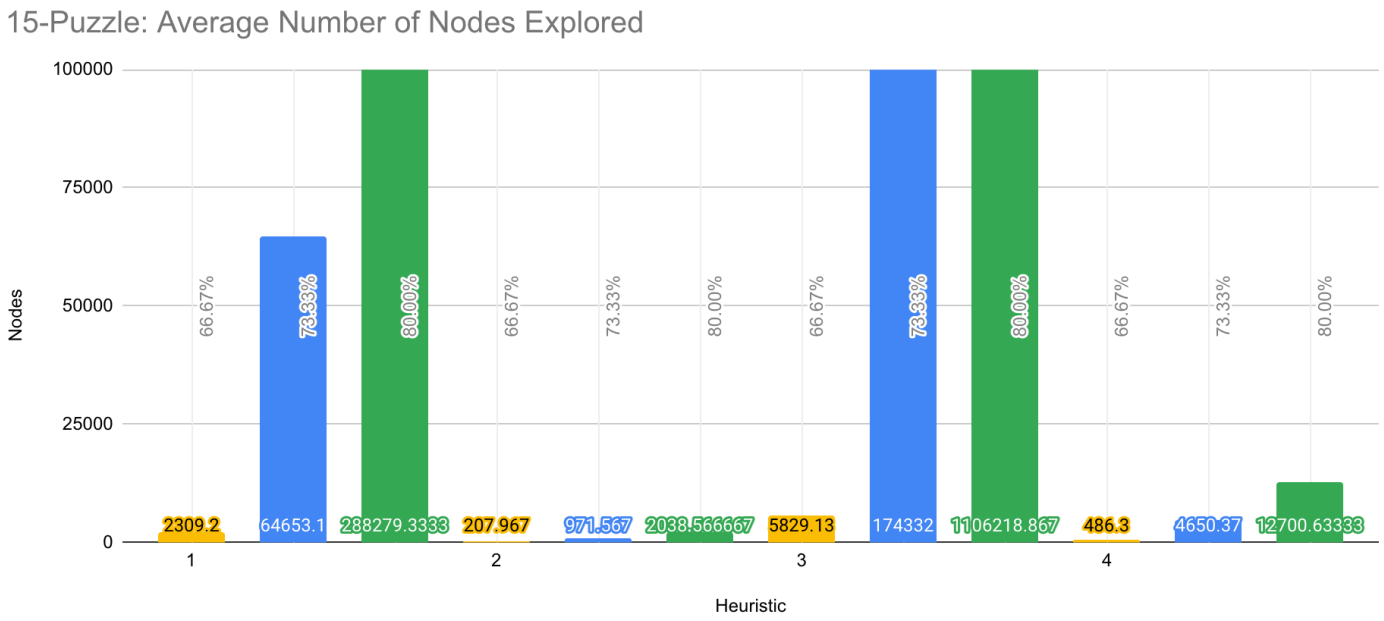
The 15 puzzle has the highest branching factors because it has the most positions where more moves are valid as compared to the 3-Puzzle and 8-Puzzle, as illustrated in the table below:

	3-Puzzle	8-Puzzle	15-Puzzle
# positions where 4 moves are valid	0	1	4
# positions where 3 moves are valid	0	4	8
# positions where 2 moves are valid	4	4	4

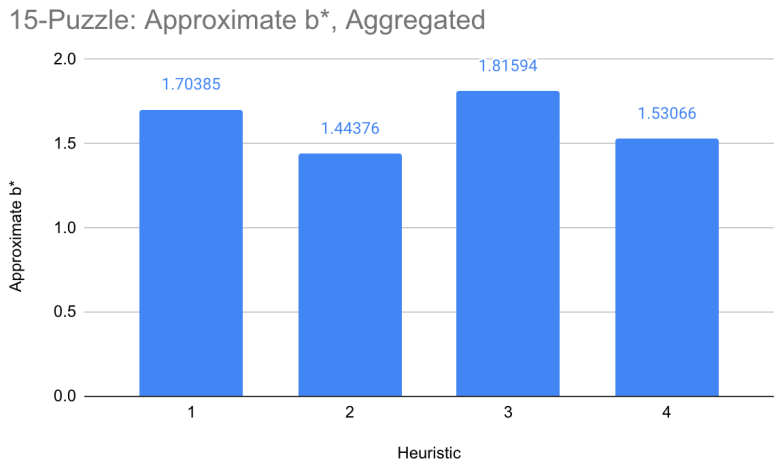


As the case with the 8-Puzzle, the solution depths are relatively consistent across heuristics and the solution depth would increase as the percentage of tiles out of place increases because the problem has increasing difficulty and a greater number of moves would be needed to slide the tile into the goal state.

There is slightly greater variation across the heuristics than in the 8-puzzle, likely because the sample size was n = 30 in the case of the 15 puzzle and n = 100 in the 8-puzzle, so edge cases affected the data a little more in the 15 puzzle experiment.



The performance of the heuristics becomes dramatically evident in this case. The number of nodes explored for h1 and h3 in the 73% and 80% cases are quite literally off the charts. We clearly can see that h2 and h4 perform vastly better than h1 and h3. Instead of getting side tracked and exploring more breadth, h2 and h4 lead the agent more directly down to the optimal solution. The dominance relationship is clearly h2 > h4 > h1 > h3, as in the case of the 8-puzzle.



The approximate b* varied slightly across the problem difficulties because of the variety of sample puzzles used. Results were aggregated to form a better sense of the b* for each heuristic. From the approximate

effective branching factors, we can see that h2 is the closest to 1, which would be an optimal heuristic. The dominance relationship based on b* would be h2 > h4 > h1 > h3, which is in agreement with the dominance relationship suggested by the results from the 8-puzzle.

In summary, conclusions from the data generated by the 15-puzzle agreed with the conclusions from the 8-puzzle. The data was more dramatic in the case of the 15-puzzle in showing the extremes of good and bad performance and there was slightly higher variation because a smaller sample size was used.

🤔 Summary of results

From the results of the experiments, it is evident that the dominance relationship between the heuristics explored is h2 > h4 > h1 > h3. That is, ManhattanDistance > Euclidean Distance > Misplaced Tiles > Swaps on MAXSORT. This dominance relationship was observed across measures of time to solve, average nodes explored, and approximate b* and across experiments. Performance measures that were not as strongly dependent on heuristics, such as branching factor and solution depth were more consistent across heuristics.

Space complexity definitely became an issue in the case of the 15-puzzles as A* search is an extremely memory intensive search. This is because the searched state space and a dictionary of reached nodes all had to be stored until a solution was found. The searched state space took about $O(b^d)$ where b is the branching factor and d is the solution depth. So, it makes sense that 8-puzzles would take much more time to solve than 3-puzzles and 15-puzzles would take longer to solve than 8-puzzles because both the b and the d are higher as n increases. We limited the agent to only solving problems where 80% or less of the tiles were out of place in the 15-puzzle because of space limitations on the machine.

The time complexity of an A* search depended on the heuristic, and more importantly, the size of the state space. h1, h2, and h4 were all linear and h3 was quadratic with respect to n+1 for an n-puzzle. The reached tree could be traversed in O(n).The A* search time was exponential with respect to b and d as discussed in the previous paragraph. Thrashing problems may have affected the time complexity.

Ideally, the agent would find the optimal solution, and this would be the solution found at the shallowest depth. However, the heuristics varied slightly so the cost optimal solution, though we can see that almost optimal solutions were at times found.

Furthermore, on measures of completeness, the agent was able to determine solutions correctly as validated by the Checker class, fulfilling phase 4 of the problem solving process. The agent was also able to determine if a puzzle was solvable or not and this was proven in the 3-puzzle experiments. For the 8-puzzle and 15-puzzle experiments, the agent was only given solvable puzzles in order to focus on the other performance measures for solvable problems.

Conceptually, the dominance relationship also makes a lot of sense. The time complexity for Swaps on MAXSORT were the highest. Manhattan Distance is the strongest heuristic because it most closely reflected what the agent would do while still being admissible. The other heuristics were admissible, but their estimates were too optimistic. For example, the EuclideanDistance is strictly smaller than the Manhattan Distance. Misplaced Tiles is also bound to be an underestimate and the Swaps on MAXSORT were less than or equal to the Misplaced Tiles. Heuristics that were admissible but too optimistic lead the agent astray sometimes to explore paths that another heuristic would not prioritize considering. Thus, the agent sometimes would find non optimal (but almost optimal solutions) while incurring high computational costs. The goal in formulating a strong heuristic is to be admissible while being as close to an actual value as possible.

In conclusion, using an A* search is a vast improvement over a brute force exploration of a state space because the heuristic saves the agent memory space, time, and computational power. Heuristics that are admissible, but closest to simulating the actual action cost are the most desirable and will lead to more optimal solutions efficiently.

💡 References

Russell and Norvig. *Artificial Intelligence: A Modern Approach, 4th Edition*. Hoboken, NJ. Pearson Education Inc, 2021.

Sentry, StackOverflow. https://ai.stackexchange.com/questions/16740/why-is-the-effective-branching-factor-used-for-measuring-performance-of-a-heuris