Contents

Optimizing the 8-Puzzle: Algorithms, Heuristics, and God's Number

Author: Tommaso Senatori
University: Università degli studi Roma TRE
Github: `https://github.com/tommasosenatori/AI_8-puzzle`
Date: December 2024

0 Abstract

In this work, I explore the well-known 8-puzzle problem, focusing on solving it optimally using various AI search algorithms. I demonstrate the computation of the "God's Number" for the 8-puzzle, which is the maximum number of moves required to solve any scrambled configuration. Additionally, I optimize the computation process by improving the search speed and introducing an efficient method for calculating the optimal solution of all possible 8-puzzle states.

In Section 2, I briefly describe the algorithms implemented to solve the 8-puzzle, dividing them into uninformed and informed search categories. Section 3 provides a detailed performance analysis, comparing the efficiency of the algorithms and evaluating the performance of A* with different heuristics. Section 4 presents the brute force calculation of God's Number for the 8-puzzle and highlights the methods used to compute this value efficiently. Section 5 examines the distribution of optimal move solutions for all solvable states, showcasing the depth of the puzzle's solution space. Finally, in Section 6 I wrap up the study, looking at the key findings and thinking about future directions, like possibly exploring the 15-puzzle or the Rubik's Cube.

1 Introduction

The 8-Puzzle is a fascinating problem in the field of artificial intelligence and computer science. It involves sliding numbered tiles within a 3x3 grid to achieve a target configuration, starting from a randomized initial state. The puzzle consists of eight tiles, each labeled with a number from 1 to 8, and an empty space. The goal is to reach the target configuration, where the tiles are ordered from left to right, top to bottom, with the empty space in the bottom-right corner.

1	2	3
4	5	6
7	8	

Figure 1: 8-puzzle solved state

Solving the 8-Puzzle is **NP-hard** because the state space grows factorially with the number of tiles. More specifically, there are 9! (362,880) possible configurations of the grid. Although the theoretical time complexity remains exponential, efficient heuristics and search strategies can significantly reduce the number of states explored, making some algorithms much faster in practice.

One of the challenges in solving the 8-Puzzle is determining whether a given initial state is solvable. This can be deduced by examining the **parity of tile inversions**, which refers to the number of pairs of tiles where a higher-numbered tile appears before a lower-numbered one. A state is **solvable** (or admissible) if the total number of inversions is even. Out of the 9! (362,880) possible configurations of the grid, only half are solvable, amounting to **181,440 configurations**. This problem's combinatorial complexity and logical nature make it particularly appealing to those, like me, who enjoy solving complex puzzles and brain teasers.

The complete **Python code** for this project is available on my GitHub repository. My approach and understanding of these methods were built upon the knowledge gained from my University's Artificial Intelligence course.

2 Algorithms and Heuristics

During my work on the 8-Puzzle, I implemented several search algorithms to explore different ways of solving the puzzle. These algorithms can be divided into two categories: Informed and Uninformed search.

2.1 Uninformed Search Algorithms

Uninformed search algorithms have no knowledge of how close a state is to the goal. They explore the search space blindly, expanding nodes without any heuristic guidance. I implemented the following uninformed search algorithms:

- **Breadth-First Search (BFS)**: This algorithm expands nodes level by level, meaning it will explore all nodes at depth i before expanding nodes at depth $i + 1$. It uses a queue, so the first node to be added to the queue is also the first one to be expanded.
- **Depth-First Search (DFS)**: DFS uses a stack and always expands the deepest node in the search tree, continuing down a path until it encounters a deadend. When that happens, it backtracks to the previous node and continues exploring other paths.
- **Uniform Cost Search (or Dijkstra)**: This algorithm always expands the node with the smallest cost from the root.
- **Iterative Deepening Depth-First Search (IDDFS)**: IDDFS performs a series of DFS searches, gradually increasing the depth limit after each iteration, until the goal state is found.

2.2 Informed Search Algorithms

In contrast to uninformed search, informed search algorithms use **heuristics** to guide the search process. These algorithms have additional information about how promising a node is, thanks to an **evaluation function**, which helps prioritize which nodes to expand. I implemented two informed search algorithms:

- **Greedy Best-First Search**: This algorithm expands the node that seems closest to the goal, based on the heuristic value $h(n)$. It uses the evaluation function $f(n) = h(n)$, which considers only the estimated cost to the goal.
- **A* Search**: A* expands the node with the best $f(n)$, where $f(n) = g(n) + h(n)$. Here, $g(n)$ is the path cost from the initial state to the current node, and h(n) is the heuristic estimate of the cost to the goal. A* is **optimal** if the heuristic is **admissible**, meaning it is **optimistic**. An optimistic heuristic means that for every node, the heuristic value is always less than or equal to the actual cost from the node to the goal state:

$$h(n) \leq h^*(n), \quad \forall n.$$

3 Performance Analysis

3.1 Algorithm Comparisons

In my evaluation, I compared the performance of several search algorithms based on an average of 1,000 solutions derived from 1,000 randomly generated initial states. This approach allowed me to observe the behavior of each algorithm over a wide range of problem instances, providing a fair comparison of their performance in terms of node expansion, execution time, and optimality.

Algorithm	**Mean Nodes Expanded**	**Average Time**	**Std Dev**	**Optimal**
BFS	92535.42	0.21120s (4.73 states/s)	0.12265s	Yes
IDDFS	309502.13	0.45441s (2.20 states/s)	0.39428s	No
DFS	69419.65	0.16524s (6.05 states/s)	0.10623s	No
Dijkstra	87616.57	0.40473s (2.47 states/s)	0.25153s	Yes
Greedy BFS	300.22	0.00239s (417.65 states/s)	0.00142s	No
A*	1645.02	0.01261s (79.28 states/s)	0.01533s	Yes

Table 1: Comparison of search algorithms for the 8-puzzle, averaged over 1000 runs.

The slowest algorithm in the comparison was IDDFS, with a mean execution time of 0.4544 seconds. On the other hand, the fastest overall algorithm was Greedy Best-First Search, which found a solution in just 2 milliseconds on average.
The most impactful result from this comparison is the contrast between informed and uninformed search algorithms in terms of the number of nodes explored. The informed algorithms, such as A* and Greedy BFS, demonstrated a significant reduction in the number of nodes generated, with a decrease of over

90% compared to the uninformed algorithms. While uninformed algorithms like BFS, DFS, and IDDFS explore hundreds of thousands of nodes, the informed algorithms manage to reduce this to just over a thousand nodes in the case of A*. This highlights the efficiency of informed search strategies, especially when good heuristics are used to guide the search.
One of the key aspects of search algorithms is their optimality, meaning their ability to always find the best possible solution. In the case of the 8-puzzle, an optimal solution is one that takes the least number of moves to reach the goal state. Among the algorithms I tested, A*, BFS, and Dijkstra are considered optimal because they always find the solution that requires the fewest moves. Notably, A* is the best among these, as it guarantees an optimal solution while expanding the least number of nodes and in the fastest time. While A* is generally optimal when using an admissible heuristic, BFS and Dijkstra are also optimal in the 8-puzzle domain due to the uniform cost of each move (all transitions have the same cost of 1).

3.2 A* Performance with Different Heuristics

I wanted to determine the best possible solution for any initial state of the 8-puzzle in the shortest amount of time, which is why I focused on the A* algorithm, as it was the fastest optimal algorithm in my previous analysis. I experimented with various heuristics specific to the 8-puzzle domain. Each heuristic used is optimistic (admissible), ensuring that A* optimality is guaranteed.
I tested 4 different heuristics in total:
Let's evaluate a specific example to see how each heuristic performs. Consider the following 8-puzzle configuration:

1	2	3
4	5	6
	7	8

Figure 2: 8-puzzle example state

Here's how each heuristic evaluates this configuration:

1. **Number of Misplaced Tiles**: This heuristic counts the number of tiles that are not in their goal positions. In this case: Misplaced Tiles = 2 (Tiles 7 and 8 are misplaced)

2. **Euclidean Distance**: The Euclidean distance for each tile is calculated as the straight-line distance from its current position to its goal position. Summing these distances gives: Euclidean Distance $= \sqrt{(2-2)^2 + (1-0)^2} + \sqrt{(2-2)^2 + (2-1)^2} = 1 + 1 = 2$

3. **Manhattan Distance**: The Manhattan distance measures the total number of moves required for each tile to reach its goal position, assuming no obstacles. In this case: Manhattan Distance $= |2-2| + |1-0| + |2-2| + |2-1| = 1 + 1 = 2$

4. **Manhattan Distance with Linear Conflict**: Linear conflict adds a penalty for tiles that are in the same row or column as their goal positions but are blocking each other. In this example, Tiles 7 and 8 form a linear conflict in the last row. The heuristic is calculated as:

   $h(n) = \text{Manhattan Distance} + 2 \times \text{Linear Conflicts}$

   Here, Linear Conflicts = 1, so:

   $$h(n) = 2 + 2 \times 1 = 4$$

To assess the performance of the heuristics, I averaged the results over 1000 random initial states and recorded the following metrics: Mean Nodes Expanded, **Mean Effective Branching Factor (EBF)**, Average Time, and Standard Deviation. The Mean EBF is a metric used to estimate how many nodes each node in the search tree expands on average. It is calculated using the formula:

$$EBF = \left(\frac{N}{L}\right)^{\frac{1}{L-1}}$$

where:

- N is the total number of nodes expanded by the A* algorithm,
- L is the length of the solution (or the depth of the goal state in the search tree).

The lower the Mean EBF, the more efficient the heuristic is at pruning the search tree, and thus, the more “informed” it is.
The results of the experiments are summarized in the table below:

Heuristic	**Mean Nodes Expanded**	**Mean EBF**	**Average Time**	**Std Dev**
Wrong Tiles	16048.851	1.3200	0.08194s (12.20 states/s)	0.09195s
Euclidean	2535.219	1.2094	0.02267s (44.11 states/s)	0.02785s
Manhattan	1653.049	1.1888	0.01226s (81.56 states/s)	0.01442s
Manh. & Linear Conflict	909.461	1.1571	0.01103s (90.65 states/s)	0.01276s

Table 2: Comparison of heuristics performance with A* on the 8-puzzle, averaged over 1000 runs.

I have ordered the heuristics from the least informed to the most informed: starting with “Wrong Tiles” and ending with “Manhattan Distance with Linear Conflict”. In general, if two A* algorithms, say A_1* and A_2*, use different heuristics where $h1 \leq h2$ for each node, then A_2* is considered more **informed** than A_1*. In other words, A_2* will expand fewer nodes than A_1*.
As observed in the table, the heuristic based on Manhattan Distance with Linear Conflict is the most informed, as it expands the least number of nodes on average, with a mean of less than 1000 nodes. As a result, it has a significantly lower mean branching factor (EBF) and performs faster compared to the other heuristics. This shows that the algorithm using this heuristic is more efficient, expanding fewer nodes and reaching the goal state faster.

4 God’s Number Brute Force Calculation

In this section, I focus on determining **God’s Number** for the 8-puzzle. God’s Number refers to the maximum number of moves required to solve any given scrambled state of the puzzle optimally, regardless of how complicated it is. For instance, the God’s Number of the Rubik’s Cube has been proven to be 20.
After discovering that my algorithm could solve 90 random scrambled 8-puzzle states optimally per second, I realized that, with a total of $\frac{9!}{2} = 181,440$ possible configurations, it would take just over 30 minutes to solve every possible starting position optimally. Motivated by this, I ran an instance of the A* algorithm for each admissible permutation of the puzzle and kept track of the longest optimal solution found.
As a result of this brute force analysis, I proved that the **God’s Number for the 8-puzzle is 31**. This means that, no matter how scrambled a given configuration of the puzzle is, it can always be solved in 31 moves or less.
Calculating God’s Number using this approach required 35 minutes with the A* algorithm and the best heuristic I tested, Manhattan Distance with Linear Conflict. For comparison, if I were to use the Wrong Tiles heuristic, it would have taken more than 4 hours. Using a simple, uninformed Breadth-First Search would have taken more than 10 hours. Thankfully, since the total number of possible scrambles for the 8-puzzle is relatively small, this approach remains feasible even with less efficient search algorithms.

5 Optimal Move Distribution

To determine the optimal move distribution efficiently, I used a smarter approach than executing 181,440 independent instances of A* for each possible configuration. Instead, I computed the optimal solution for every state using **A* in reverse**, starting from the solved state.
The idea behind this method is straightforward: begin from the solved state, which serves as the base case, and incrementally compute solutions for more complex states by exploring their neighbors in reverse. This approach ensures that every configuration is solved optimally without redundant computations.
To illustrate the difference: if a state can be solved optimally in k moves, the brute force approach would solve it by running a new A* instance starting from that state, recomputing the solution path entirely. In contrast, in the reverse approach, when solving a state that is k moves away from the solved state, I can rely on the fact that I have already computed the solutions for states that are $k-1$ moves away. This means that instead of continuously opening nodes, I can use the precomputed results of these simpler cases to directly determine the solution. This eliminates redundant computations and dramatically speeds up the process.
Using this method, I achieved a huge speedup compared to the brute force approach. While solving all configurations individually would have taken 35

minutes with the best heuristic (Manhattan with linear conflict), this optimized method completed the computation in just 1.2 seconds: a **99.94% speedup**.

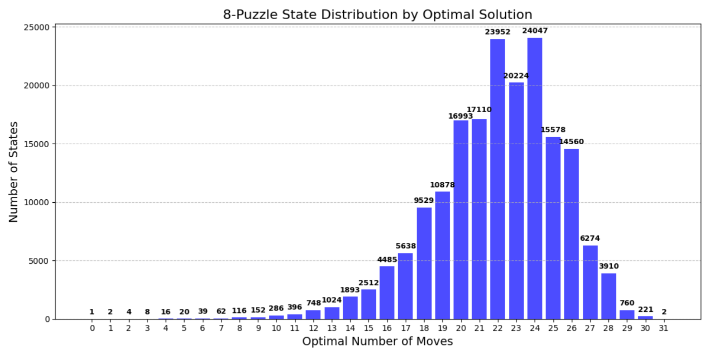

Figure 3: Distribution of optimal move solutions for all 8-puzzle configurations.

The graph I present here is a unique result. It shows, for each possible number of moves, how many distinct 8-puzzle configurations can be solved optimally in that exact number of moves. For example, when $x = 29$ and $y = 760$, this means that I calculated there are 760 different states in the 8-puzzle that have an optimal solution (the shortest solution, which requires the least number of moves) that takes exactly 29 moves to reach the solved state.
The analysis of the 181440 total states shows the following results:

- **Mean**: 21.97
- **Median**: 22.0
- **Mode**: 24
- **Standard Deviation**: 3.37

On average, a random 8-puzzle state can be solved optimally in about 22 moves. 95% of the states can be solved in **15 - 28 moves**, 2.5% can be solved in less than 15 moves, and the other 2.5% can only be solved with more than 28 moves.
Moreover, I discovered that there are only two states that require at least 31 moves to be solved. These two states are responsible for determining the God's Number for the 8-puzzle, which is 31. This means that the worst-case scenario for solving the 8-puzzle optimally is only due to these two specific configurations. The two states that require the maximum number of moves to solve are:

8	6	7
2	5	4
3		1

6	4	7
8	5	
3	2	1

Figure 4: The 2 most difficult states in the 8-puzzle, both requiring a minimum of 31 moves to be solved.

6 Conclusions and Future Work

I am very pleased to have experimented with the 8-puzzle, a benchmark puzzle I have long enjoyed. Throughout this project, I implemented various algorithms and was able to determine the most efficient one for solving the puzzle optimally. Through this process, I demonstrated that the God's number for the 8-puzzle is 31 moves. In addition to this, I successfully improved the computational speed and efficiency of solving the puzzle by developing a more effective approach, which reduced the time required to compute optimal solutions from 35 minutes to just 1.2 seconds, achieving a 99.94% speedup.

In the future, I may consider experimenting with other, more complex puzzles. One possibility is the 15-puzzle, which has $\frac{16!}{2} = 10,461,394,944,000$ possible configurations. Another challenge could be the Rubik's Cube, which has a staggering $43,252,003,274,489,856,000$ possible configurations. Both of these puzzles present fascinating challenges and would push the limits of current search algorithms and computational capabilities. I will certainly have a lot of fun tackling these puzzles, as solving them has always been a passion of mine.

References

[1] Artificial Intelligence Course, academic year 2024/2025, Università degli Studi di Roma Tre.

[2] Manhattan and Linear Conflicts Heuristic Examples: YouTube Video.