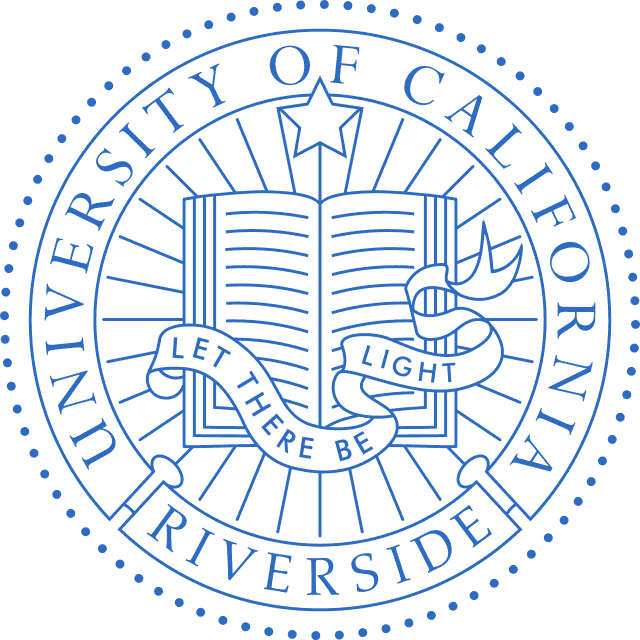
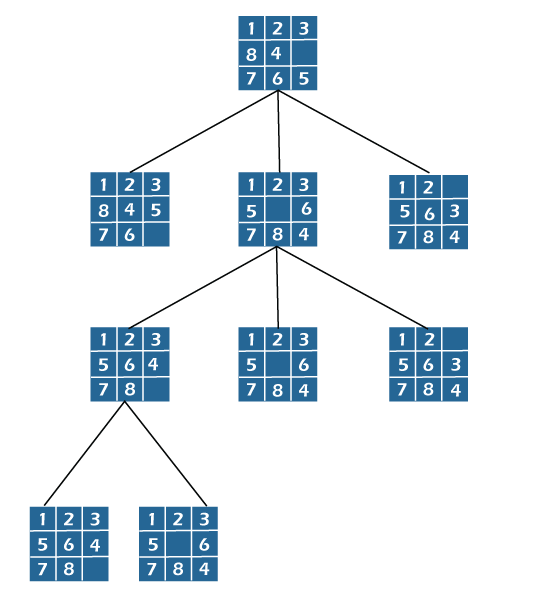
# **CS170 Project 1: The EIGHT PUZZLE**

****

**Name: Alfonso Mayoral**

**SID: 862550010**

**NetID: amayo026**

**Date: 02/08/25**

**GitHub Project Link:** [**https://github.com/alfonsomayoral/8\_Puzzle\_Code/tree/main**](https://github.com/alfonsomayoral/8_Puzzle_Code/tree/main)

## **Specifications of the Project**

In order to perform this project I consulted:

* The Blind Search (part 1 and 2) and Heuristic Search lecture slides and notes annotated from Dr. Eamonn’s lecture
* Python 3.6 Documentation

All the code is original except for:

* All subroutines used from heapq, to handle the node structure of states.
* All subroutines used from copy, to deepcopy and correctly modify states.

Moreover, the dataset used was created by myself using Excel through the different metrics obtained after using my code with different initial states and different maximum depths. This dataset is attached in this project in the correspond section, as well as the code implemented.

## **Outline**

**Introduction ……………………………………………………………………………………. page 3**

**Background Review …………………………………………………………………………… page 3**

**Search Algorithms ……………………………………………………………………………... page 4**

**Methodology …………………………………………………………………………………… page 4**

**Search Algorithms Performance Analysis …………………………………………………… page 5**

**Conclusion ……………………………………………………………………………………… page 7**

**Dataset ………………………………………………………………………………………….. page 8**

**Code …………………………………………………………………………………………….. page 9**

**Graphs …………………………………………………………………………..……………… page 11**

**References ……………………………………………………………………………………… page 12**

## **Introduction**

Sliding puzzles, also called Gem Puzzle, Boss Puzzle, Mystic Square, and more, are a class of combinatorial problems characterized by a set of numbered or patterned pieces arranged in a grid, with a single empty space that allows the movement of the tiles. The objective is to rearrange the pieces from an initial configuration to a goal state, see Figure 1, where all the tiles are ordered from smallest to largest, starting from the top left corner, through a sequence of moves. The 8-puzzle, which is a smaller version of the 15-puzzle, is one of the most studied cases of sliding tile puzzles. It features a 3x3 grid with eight tiles and one empty space, which poses a more computationally manageable problem, but not trivial.

This project aims to develop a robust solution for the 8-puzzle problem using three different search algorithms: Uniform Cost Search, A\* with the Misplaced Tile heuristic, and A\* with the Manhattan Distance heuristic. Additionally, the project evaluates these algorithms by comparing their efficiency through different performance metrics obtained in various initial states.

A screenshot of a game

Descripción generada automáticamente

*Figure 1:*

*A representation of the Goal State of the 15-puzzle*

This is the first project of the Introduction to Artificial Intelligence course, taught by Dr. Eamonn Keogh, at the University of California, Riverside during the Winter 2025 quarter. The following report details my results and conclusions obtained throughout the completion of the project. This report explores Uniform Cost Search and the heuristics of Misplaced Tile and Manhattan Distance applied to A\* to solve the 8-puzzle problem. My language of choice was Python (version 3.6), and the complete project code is included, along with a dataset where all the results and metrics obtained for different maximum depths of the search space are recorded.

## **Background Review**

Historically, the sliding puzzle was first introduced as the "15 puzzle" in the late 19th century. Its invention is attributed to Noyes Palmer Chapman, a postman from Canastota, New York, who is credited with creating the puzzle around 1874 [1]. The puzzle consists of a 4x4 grid with 15 numbered tiles and a single blank space, challenging players to arrange the tiles in sequential order through sliding movements enabled by the free space. The puzzle gained great popularity in the United States during the 1880s, captivating the public's interest and becoming a widely recognized mathematical curiosity in a short period of time.

The mathematical importance of the puzzle became evident as researchers analyzed its properties. In 1879, William Woolsey Johnson and William Ernest Story independently provided a mathematical explanation for the solvability of this puzzle, introducing concepts such as permutations and inversions [2]. They demonstrated that the puzzle could be solved if and only if the number of inversions was even, laying the groundwork for understanding its underlying structure. This discovery led to further studies in combinatorial game theory, turning this puzzle into a crucial case for the study of permutation groups and problem-solving complexity [3].

Nowadays, in the field of artificial intelligence, the sliding puzzle has become a standard for evaluating search algorithms and heuristic methods. Its simplicity, combined with the computational challenges it presents, makes it an ideal problem for testing the efficiency and effectiveness of algorithms. The 8-puzzle variant, set on a 3x3 grid, offers 181,440 unique states, providing a manageable yet complex environment for algorithmic exploration. Researchers have used this puzzle to develop and refine various search strategies, including uninformed search algorithms such as Breadth-First Search (BFS) and Depth-First Search (DFS), as well as informed search techniques like A\* [4].

## **Search Algorithms**

The project implements three distinct search algorithms to solve the 8-puzzle problem: Uniform Cost Search (UCS), A\* Search with the Misplaced Tile heuristic, and A\* Search with the Manhattan Distance heuristic [4].

**Uniform Cost Search (UCS)**

Uniform Cost Search (UCS) is an uninformed search algorithm that expands nodes based solely on their path cost, without considering any heuristic. This is a specific case of the A\* algorithm, where the heuristic function h(n) is set to zero [6]. In the case of the 8-puzzle, UCS explores neighboring states by sliding tiles adjacent to the empty space, using a priority queue to prioritize the least-cost path [9]. Although UCS guarantees finding the optimal solution, it can be computationally expensive since it explores a large number of possible paths, especially in more complex problems [8].

**A\* Search with the Misplaced Tile Heuristic**

The second algorithm implemented is A\* with the Misplaced Tile Heuristic. In this approach, the heuristic function is based on the sum of the number of tiles that are not in their correct positions, excluding the empty tile [4]. The A\* algorithm combines this heuristic with the path cost which represents the number of moves made to reach a particular state [6].

Initial State:  
[[1, 2, 3],  
 [5, 6, 0],  
 [7, 8, 4]]

Goal State:  
[[1, 2, 3],  
 [4, 5, 6],  
 [7, 8, 0]]

*Figure 2:*

*Practical example of misplaced tile heuristic with .*

A\* search expands nodes based on the total cost function , which is a combination of both the actual path cost and the estimated remaining cost to reach the goal [5]. In the example of Figure 2, the puzzle node has 3 misplaced tiles, without counting 0, therefore h(n) = 3. By incorporating the Misplaced Tile heuristic, A\* can focus its search on states that are closer to the goal, reducing the number of explored nodes compared to UCS, since queue will expand the node with the cheapest cost.

**A\* Search with the Manhattan Distance Heuristic**

A\* Search with the Manhattan Distance Heuristic improves upon the Misplaced Tile heuristic by considering the actual distances that each tile must travel to reach the goal state [7]. The Manhattan Distance heuristic calculates the sum of the absolute differences in the horizontal and vertical distances between each tile’s current position and its goal position [4]. This heuristic is more precise than the Misplaced Tile heuristic and explores fewer nodes, as it provides a more accurate estimate of the remaining distance to the goal [9]. A\* uses this heuristic in conjunction with the path cost to guide the search, effectively prioritizing nodes that are closer to the goal. [5]

Using the practical example of the initial state shown in Figure 2, it can be seen that tiles 5, 6 and 4 are not in their goal positions (not counting the 0 tile). Computing the Manhattan Distance Heuristic, we will reach that

## **Methodology**

To properly evaluate the performance of the different algorithms, a database was created to measure various metrics across different initial states and depths (attached later in the Database section). This approach allowed for a comprehensive comparison of the efficiency and effectiveness of the algorithms under different conditions. By testing with a variety of initial states (showed in Figure 3) and varying the maximum depth, we were able to evaluate how each algorithm performs in terms of exploration, time complexity, and solution quality, providing valuable insights into their strengths and limitations.

1 6 7

5 0 3

4 8 2

1 2 3

4 5 6

7 8 0

8 6 7

2 5 4

3 0 1

0 7 2

4 6 1

3 5 8

7 1 2

4 8 5

6 3 0

8 1 3

4 0 2

7 6 5

1 3 6

5 0 7

4 8 2

1 3 6

5 0 2

4 7 8

1 2 3

5 0 6

4 7 8

1 2 3

4 5 6

0 7 8

**Depth 0 Depth 2 Depth 4 Depth 8 Depth 12 Depth 14 Depth 16 Depth 20 Depth 24 Depth 31**

*Figure 3: Set of initial states selected to measure the performance of the different algorithms shorted by “hardness”*

Before start with the comparison of the performance of each search algorithm, let’s define the several key concepts problem. First of all, the initial state represents the starting configuration of the puzzle, where tiles are arranged in an arbitrary manner, and serves as the input for the search algorithm. The goal state is the predefined arrangement where all tiles are in numerical order, with the empty space in the bottom-right corner, marking the completion of the puzzle. Operators define the valid moves in the puzzle, allowing adjacent tiles to slide into the empty space, with movements constrained by the grid’s boundaries.

To evaluate the efficiency of each algorithm, four key metrics were analyzed. Nodes Expanded measures the number of states generated and explored, indicating search efficiency. Maximum Queue Size tracks the peak number of states stored, reflecting memory usage. Time Taken records the duration required to find a solution, assessing computational speed. Cost of Solution (Depth) counts the number of moves needed to reach the goal, highlighting solution optimality.

The logic of the program solves the 8-puzzle problem by implementing three search algorithms, depending on the user choice. It begins by defining the initial and goal states, then generates neighboring states by moving adjacent tiles into the empty space. Using a priority queue, the algorithm expands nodes based on their cost function , where is the path cost and is the heuristic estimate. The search continues until the goal state is reached, returning the path taken to the solution along with the key performance metrics, or returns failure in the case of not finding a solution.

## **Search Algorithms Performance Analysis**

This section goes on to explain the analysis conducted to compare the performance of each of the search algorithms implemented based on the metrics measured after executing the proposed code in this project for each of the test cases, shown in Figure 3.

Up to the easier puzzles, it can be clearly observed that the difference between the three algorithms is relatively minimal, but as we increase the depth, the results show a trend in the efficiency of the different search algorithms, both in time, as well as in expanded nodes and maximum queue size.

In terms of time, shown in Figure 4, the Uniform Cost Search algorithm is the most inefficient as the depth increases. Its time grows exponentially, especially at greater depths.

A\* (Misplaced Tile) significantly improves its search time results compared to UCS, but it still shows exponential growth as the depth increases. However, as expected, A\* (Manhattan Distance) is the most efficient algorithm in terms of time, with a more controlled growth even at high depths. This is because it is a more precise heuristic compared to A\* (Misplaced Tiles), which makes reaching the goal state much faster.

A graph with green and orange lines

El contenido generado por IA puede ser incorrecto.

*Figure 4: Depth vs Time Analysis Line Chart. Green line represents UCS, Orange line represents A\* Misplaced Tiles and Blue line represents A\* Manhattan Distance*

On the other hand, if we analyze the results in terms of nodes expanded in each of the algorithms, shown in Figure 5, Uniform Cost Search expands significantly more nodes than the other two algorithms as the depth of the problem increases. Its growth is the most pronounced; it can be observed how it shifts from exponential growth from a depth of 12 to 20, to slightly linear growth from 20 to the diameter of the problem.

However, by using any of the heuristics, it can be observed that the number of expanded nodes is drastically reduced compared to UCS. Instead, A\* (Misplaced Tiles) shows exponential growth in the expanded nodes as the depth increases. A\* (Manhattan Distance) expands the fewest number of nodes in all test cases, showing exponential growth starting from a depth of 24 but much less pronounced than any other algorithm. Just like in the previous analysis, this is because being a more precise heuristic than A\* (Misplaced Tiles), this method achieves the solution with less node exploration, which directly impacts the time taken.

A graph with text and numbers

El contenido generado por IA puede ser incorrecto.

*Figure 5: Depth vs Nodes Expanded Analysis Line Chart. Green line represents UCS, Orange line represents A\* Misplaced Tiles and Blue line represents A\* Manhattan Distance*

Finally, regarding the max queue size, it can be observed in Figure 6 how all the algorithms experience exponential growth in memory usage as the depth increases.

After analyzing the results, it is obvious that the USC algorithm is the least efficient in terms of memory, reaching values close to 60,000 nodes in the queue at depth 31.

A\* (Misplaced Tiles) grows more gradually than UCS, but when it reaches the problem's diameter, its maximum queue size significantly increases, outpacing UCS's memory usage at a depth of 31. In addition, A\* (Manhattan Distance) is the most memory-efficient of the three algorithms, as evidenced by the fact that its maximum queue size is the smallest across all assessed depths.

A graph of a graph with text and numbers

El contenido generado por IA puede ser incorrecto.

*Figure 6: Depth vs Max Queue Size Analysis Line Chart. Green line represents UCS, Orange line represents A\* Misplaced Tiles and Blue line represents A\* Manhattan Distance*

Overall, the analysis of search algorithms highlights clear differences in performance as the depth of the problem increases. Although all algorithms show exponential growth in resource consumption at greater depths, the use of heuristics in A\* significantly improves efficiency in terms of time, expanded nodes, and memory usage.

These findings reinforce the importance of selecting appropriate heuristics when solving search problems like the 8-puzzle. That is why the ability to reduce the expanded nodes and manage memory more efficiently has a direct impact on the practical viability of the algorithm. Based on these results, the foundations are established to reach the final conclusions in the next section of this project report. Additionally, the metrics of each algorithm can be visualized together in the graphs attached in the appendix section.

## **Conclusion**

Through an analytical comparison of the search algorithms employed in this research, we were able to assess the effectiveness and efficiency of the three search algorithms in solving the 8-puzzle problem: Uniform Cost Search (UCS), A\* with Misplaced Tiles, and A\* with Manhattan Distance. Each of these approaches uses different strategies to explore the search space and find the optimal solution. Through the analysis of the results obtained in terms of expanded nodes, execution time, and max queue size, it can be concluded that:

The **Uniform Cost Search (UCS)** algorithm, by not using any specific heuristic (h(n) is hardcoded to equal 0), explores the nodes uniformly according to the accumulated cost. Despite being a complete and optimal algorithm, it is extremely slow and inefficient in terms of performance. Its exhaustive search strategy generates a significantly high number of expanded nodes, which increases execution time and memory consumption, reaching 181,440 expanded nodes assuming we track repeated states, what implies a memory consumption becomes unsustainable. While it finds the optimal solution, its high computational cost makes it impractical for more complex or larger-scale problems.

On the other hand, the **A\* (Misplaced Tiles)** algorithm introduces a heuristic strategy that measures the number of pieces that are not in their correct position. In this way, efficiency is greatly improved compared to UCS by directing the search towards more promising states. However, this heuristic turns out to be relatively weak as the depth of the problem increases, as it does not take into account the actual distance each piece needs to move. Consequently, although it expands fewer nodes than UCS and shows great performance in low-depth problems, it requires a significant amount of memory and execution time as the problem increases in depth, making it a less efficient algorithm compared to more informed heuristics.

Finally, the **A\* (Manhattan Distance)** algorithm proves to be the best option in terms of execution time, expanded nodes, and consequently, memory usage. When considering the actual distance in terms of the movements required to place each piece in its final position, this heuristic provides a more informed guide, thereby reducing the search space. After analyzing the results, it can be concluded that this algorithm expands significantly fewer nodes compared to UCS and A\* (Misplaced Tiles) at any depth, significantly reducing execution time without compromising the quality of the solution. Its ability to balance exploration and efficiency makes it the most effective option for solving the 8-puzzle problem.

As a final conclusion, this project report has demonstrated that the choice of a good heuristic is fundamental to improving the performance of search algorithms. It has been demonstrated that UCS is too costly in computational terms and proves to be an inefficient search algorithm in high-depth problems. On the other hand, A\* (Misplaced Tiles) offers a moderate improvement by introducing a heuristic based on the number of pieces placed in incorrect positions, but it is evident that it does not provide optimal results in terms of memory and time as the depth increases. However, A\* (Manhattan Distance) turns out to be the best option thanks to using a much more precise heuristic for informed search, drastically reducing the number of expanded nodes and optimizing execution time. Therefore, the importance of using informed and well-designed heuristics to efficiently tackle search problems is demonstrated.

## **Dataset**



## **CODE**

import heapq

import copy

import time

# Define the goal state of the puzzle as a 3x3 grid where the tiles are in order

goal\_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

def find\_zero(state):

    """Find the position of the empty tile (0) in the puzzle."""

    for i in range(len(state)):

        for j in range(len(state[i])):

            if state[i][j] == 0: # Identify the position of the blank tile

                return i, j

def is\_goal(state):

    """Check if the current state matches the goal state."""

    return state == goal\_state

def get\_neighbors(state):

    """Generate all possible moves by sliding tiles adjacent to the empty space."""

    x, y = find\_zero(state)

    neighbors = []

    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # up, down, left, right

    for dx, dy in directions:

        nx, ny = x + dx, y + dy

        if 0 <= nx < 3 and 0 <= ny < 3:

            new\_state = copy.deepcopy(state)

            new\_state[x][y], new\_state[nx][ny] = new\_state[nx][ny], new\_state[x][y]

            neighbors.append(new\_state)

    return neighbors

def misplaced\_tile\_heuristic(state):

    """Calculate the number of tiles that are not in their correct positions."""

    count = 0

    for i in range(3):

        for j in range(3):

            # Exclude the blank tile and count tiles that are not in their goal positions

            if state[i][j] != 0 and state[i][j] != goal\_state[i][j]:

                count += 1

    return count

def manhattan\_distance\_heuristic(state):

    """Calculate the sum of Manhattan distances of tiles from their goal positions."""

    distance = 0

    for i in range(3):

        for j in range(3):

            if state[i][j] != 0:

                goal\_x, goal\_y = divmod(state[i][j] - 1, 3)

                distance += abs(goal\_x - i) + abs(goal\_y - j)

    return distance

def general\_search(initial\_state, heuristic\_fn):

    """General search algorithm that expands nodes based on a given heuristic."""

    start\_time = time.time()

    pq = []

    heapq.heappush(pq, (0, 0, initial\_state, []))  # (f(n), g(n), state, path)

    visited = set()

    max\_queue\_size = 0

    nodes\_expanded = 0

    while pq:

        max\_queue\_size = max(max\_queue\_size, len(pq))

        fn, gn, current\_state, path = heapq.heappop(pq)

        if tuple(map(tuple, current\_state)) in visited:

            continue

        visited.add(tuple(map(tuple, current\_state)))

        if is\_goal(current\_state):

            elapsed\_time = time.time() - start\_time # End timing

            return path, gn, nodes\_expanded, max\_queue\_size, elapsed\_time, len(visited)

        nodes\_expanded += 1

        for neighbor in get\_neighbors(current\_state):

            new\_path = path + [(neighbor, gn + 1, heuristic\_fn(neighbor))]  # Store g(n) and h(n)

            hn = heuristic\_fn(neighbor)

            heapq.heappush(pq, (gn + 1 + hn, gn + 1, neighbor, new\_path))

    elapsed\_time = time.time() - start\_time

    return "Failure", -1, nodes\_expanded, max\_queue\_size, elapsed\_time, len(visited)

def uniform\_cost\_search(initial\_state):

    """Algorithm 1: Perform Uniform Cost Search."""

    return general\_search(initial\_state, lambda \_: 0) # No heuristic (h(n) = 0)

def a\_star\_misplaced\_tile(initial\_state):

    """Algorithm 2: Perform A\* search using the Misplaced Tile heuristic."""

    return general\_search(initial\_state, misplaced\_tile\_heuristic)

def a\_star\_manhattan\_distance(initial\_state):

    """Algorithm 3: Perform A\* search using the Manhattan Distance heuristic."""

    return general\_search(initial\_state, manhattan\_distance\_heuristic)

if \_\_name\_\_ == "\_\_main\_\_":

    print("Welcome to the 8-Puzzle Solver!")

    choice = input("Type '1' to use a default puzzle, or '2' to input your own: ")

    if choice == '1':

        initial\_state = [[8, 6, 7], [2, 5, 4], [3, 0, 1]]

    else:

        print("Enter your puzzle (3x3 grid, use space to separate numbers, and 0 for blank):")

        initial\_state = [list(map(int, input(f"Enter row {i + 1}: ").split())) for i in range(3)]

    print("Select the algorithm:")

    print("1. Uniform Cost Search")

    print("2. A\* with Misplaced Tile Heuristic")

    print("3. A\* with Manhattan Distance Heuristic")

    # Prompt the user to select a search algorithm

    algorithm\_choice = input("Enter your choice (1/2/3): ") # Example default state, depth 31

    if algorithm\_choice == '1':

        path, cost, nodes\_expanded, max\_queue\_size, elapsed\_time, visited\_states = uniform\_cost\_search(initial\_state)

    elif algorithm\_choice == '2':

        path, cost, nodes\_expanded, max\_queue\_size, elapsed\_time, visited\_states = a\_star\_misplaced\_tile(initial\_state)

    elif algorithm\_choice == '3':

        path, cost, nodes\_expanded, max\_queue\_size, elapsed\_time, visited\_states = a\_star\_manhattan\_distance(initial\_state)

    else:

        print("Invalid choice. Exiting.")

        exit()

    if path != "Failure":

        print("Solution found!")

        print("Steps to solution:")

        for step, g\_n, h\_n in path:

            for row in step:

                print(row)  #Print each state in the solution path

            print(f"g(n): {g\_n}, h(n): {h\_n}\n")

        print(f"Depth: {cost}") # Print the depth of the solution

        print(f"Nodes expanded: {nodes\_expanded}")  # Print the total nodes expanded

        print(f"Time taken: {elapsed\_time:.4f} seconds")  # Print the time required to reach the goal state

        print(f"Maximum queue size: {max\_queue\_size}")  # Print the max queue size

        print(f"Total visited states: {visited\_states}") # Print the total nodes visited

    else:

        print("No solution found.")

## **Graphs**

A graph with green and orange lines

El contenido generado por IA puede ser incorrecto. *Figure 4: Depth vs Time Analysis Line Chart. Green line represents UCS, Orange line represents A\* Misplaced Tiles and Blue line represents A\* Manhattan Distance*

A graph of a graph with text and numbers

El contenido generado por IA puede ser incorrecto.

*Figure 6: Depth vs Max Queue Size Analysis Line Chart. Green line represents UCS, Orange line represents A\* Misplaced Tiles and Blue line represents A\* Manhattan Distance*

*A graph on a screen

El contenido generado por IA puede ser incorrecto.*

*Figure 8: A\* (Misplaced Tiles) Metrics*

A graph with text and numbers

El contenido generado por IA puede ser incorrecto.

*Figure 5: Depth vs Nodes Expanded Analysis Line Chart. Green line represents UCS, Orange line represents A\* Misplaced Tiles and Blue line represents A\* Manhattan Distance*

*A graph with green and white text

El contenido generado por IA puede ser incorrecto.*

*Figure 7: Uniform Cost Search Metrics*

*A graph on a computer screen

El contenido generado por IA puede ser incorrecto.*

*Figure 9: A\* (Manhattan Distance) Metrics*

## **REFERENCES**

[1] Slocum, J., & Sonneveld, D. (2006). The 15 Puzzle Book: How It Drove the World Crazy. Slocum Puzzle Foundation.

[2] Johnson, W. W., & Story, W. E. (1879). Notes on the "15" Puzzle. American Journal of Mathematics, 2(4), 397–404.

[3] Archer, A. F. (1999). A Modern Treatment of the 15 Puzzle. The American Mathematical Monthly, 106(9), 793–799.

[4] Russell, S., & Norvig, P. (2020). Artificial Intelligence: A Modern Approach (4th ed.). Pearson.

[5] Dechter, R., & Pearl, J. (1985). *Generalized best-first search strategies and the optimality of A*.\* Journal of the ACM (JACM), 32(3), 505-536.

[6] Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). *A formal basis for the heuristic determination of minimum cost paths.* IEEE Transactions on Systems Science and Cybernetics, 4(2), 100-107.

[7] Korf, R. E. (1985). *Depth-first iterative-deepening: An optimal admissible tree search.* Artificial Intelligence, 27(1), 97-109.

[8] Korf, R. E. (1993). *Linear-space best-first search.* Artificial Intelligence, 62(1), 41-78.

[9] Pearl, J. (1984). *Heuristics: Intelligent search strategies for computer problem solving.* Addison-Wesley.