CS 205 – Project 1

Name: Yijian Chai
SID – 862392329

INDEX

My code pages is 8-12, URL is https://github.com/YijianChai/CS205

The eight-puzzle

Introduction

The 8-puzzle problem is classic sliding tile puzzle where there are 8 tiles numbered 1-8 and an empty tile arranged in a 3×3 grid. The goal is to move blocks by swapping empty blocks with its adjacent blocks until all the blocks are ordered in a specific goal state.

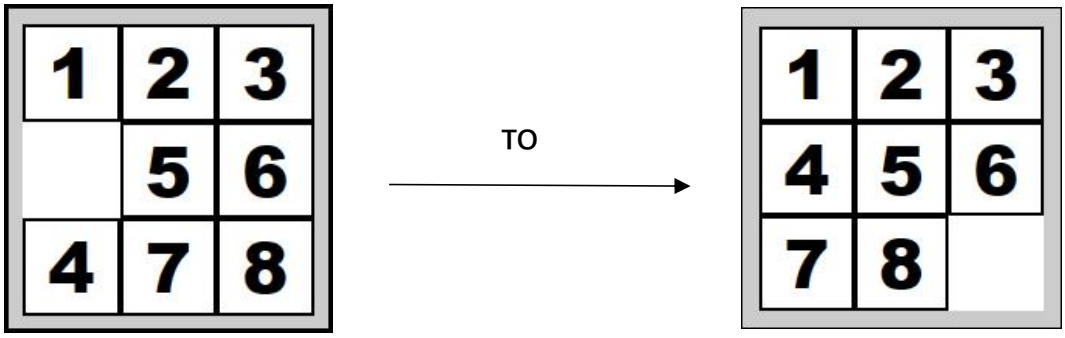

Figure 1: An example of solving the 8-puzzle problem [1]

We can see from Figure 1 that a simple initial state is randomly selected on the left, and the goal state of all 8-puzzle problems is shown on the right.

In this project, we aim to solve the 8-puzzle problem using three search algorithms: Uniform Cost Search (UCS), A* with Mismatched Tiling Heuristic (MTH) and A* with Manhattan Distance Heuristic (MDH). After designing these algorithms and developing corresponding user interface. I will compare the performance of these algorithms in terms of the number of nodes scaled and the maximum queue size under a series of given solution depth.

Algorithms

1. Uniform Cost Search (UCS)
   This algorithm is a fundamental search algorithm in which the cost (g(n)) of each move is 1. It can be regarded as an A* heuristic algorithm whose h(n) is equal to 0. All possible states from a given state are inserted into a priority queue according to their g(n). The state with the lowest g(n) is dequeued, compared to the goal, and its child nodes are enqueued based on their respective g(n) values.
2. A* with Misplaced Tile Heuristic (MTH)
   A* algorithms combine heuristics (h(n)) and cost (g(n)) to make a function f(n) (i.e., $f(n) = g(n) + h(n)$). All the puzzle states are sorted by this f(n) and the puzzle state with the lowest f(n) is dequeued, checked, and its child nodes are enqueued according to their f(n) values. Specifically, Misplaced Tile Heuristic calculates the total number of tiles in incorrect positions, assuming that a smaller heuristic means the state is closer to the goal.
3. A* with Manhattan Distance Heuristic (MDH)

Manhattan Distance Heuristic calculates h(n) by adding up the number of moves each tile would require to reach its correct position, assuming all other tiles are empty. The f(n) value is calculated as the sum of g(n) and h(n). Then the puzzle states are inserted into the priority queue and sorted by their f(n) values.

Implementation

I inserted a state of a puzzle into the priority queue (heapq) with its current state (1-8 and empty tile), g(n) value, h(n) value, and its previous path [2]. Various methods were implemented to print the current puzzle, compare f(n) with different puzzles, check if it's the solution, and generate its next states (child nodes). In order to avoid to consider visited states, I created a set to record all the visited puzzle states. Then I converted the puzzle state to bypes to compare two states and test whether they are equal.

In order to facilitate the evaluation of whether the algorithm can execute correctly and get the expected results, I developed a user interface for the n-puzzle problem solver. After entering the user interface, you can choose to use the default puzzle or create a new puzzle yourself. As shown in Figure 2, if the default puzzle is used, then we continue to select the test cases given by the project [3]. Here I choose a case where the actual depth is 20. Next, a choice of three algorithms will be given.

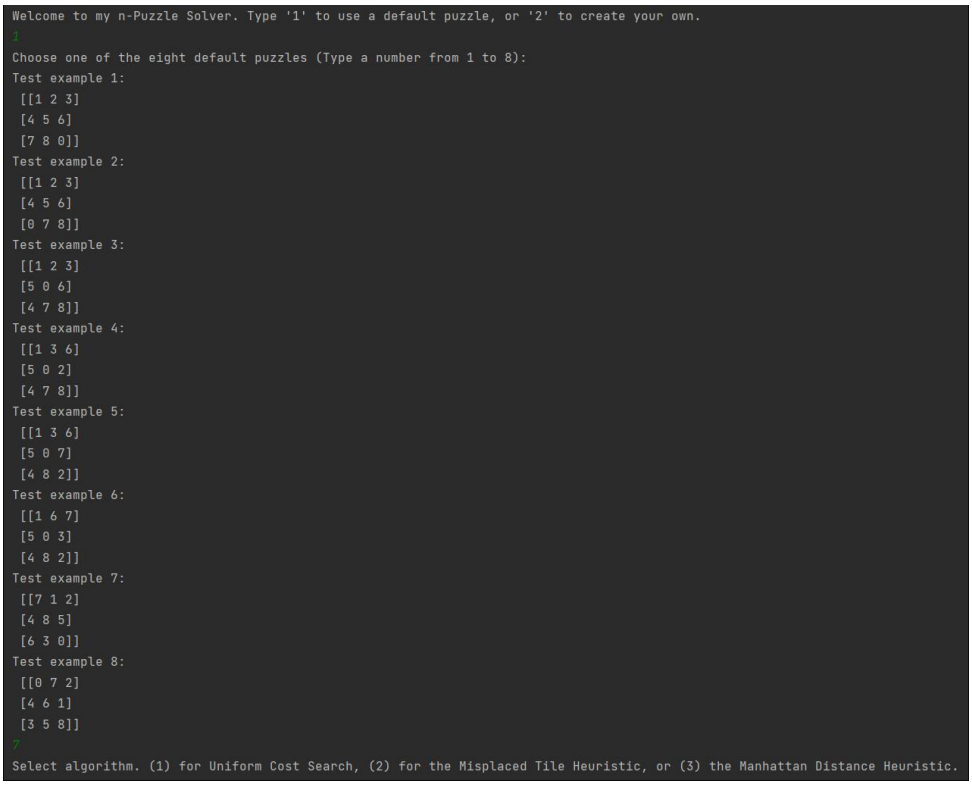

Figure 2: Use default cases (provided by the guide) to test

Subsequently, the choice of a specific algorithm needs to be made. As shown in Figure 3, here I chose Misplaced Tile Heuristic, and then got the path to transform the

state of this test case into the target state step by step. And get the depth of the solution, the number of nodes expanded and the maximum queue size.

```
7
Select algorithm. (1) for Uniform Cost Search, (2) for the Misplaced Tile Heuristic, or (3) the Manhattan Distance Heuristic.
2
Solution:
((7, 1, 2), (4, 8, 5), (6, 0, 3))
((7, 1, 2), (4, 8, 5), (0, 6, 3))
((7, 1, 2), (0, 8, 5), (4, 6, 3))
((7, 1, 2), (8, 0, 5), (4, 6, 3))
((7, 1, 2), (8, 5, 0), (4, 6, 3))
((7, 1, 2), (8, 5, 3), (4, 6, 0))
((7, 1, 2), (8, 5, 3), (4, 0, 6))
((7, 1, 2), (8, 5, 3), (0, 4, 6))
((7, 1, 2), (0, 5, 3), (8, 4, 6))
((0, 1, 2), (7, 5, 3), (8, 4, 6))
((1, 0, 2), (7, 5, 3), (8, 4, 6))
((1, 2, 0), (7, 5, 3), (8, 4, 6))
((1, 2, 3), (7, 5, 0), (8, 4, 6))
((1, 2, 3), (7, 0, 5), (8, 4, 6))
((1, 2, 3), (7, 4, 5), (8, 0, 6))
((1, 2, 3), (7, 4, 5), (0, 8, 6))
((1, 2, 3), (0, 4, 5), (7, 8, 6))
((1, 2, 3), (4, 0, 5), (7, 8, 6))
((1, 2, 3), (4, 5, 0), (7, 8, 6))
((1, 2, 3), (4, 5, 6), (7, 8, 0))
Solution depth: 20
Number of nodes expanded: 3303
Max queue size: 1896
```

Figure 3: Test the selected case and get the solution

Next, in order to evaluate whether these three algorithms have sufficient generalization, I will try to run the algorithms in a more generalized scenario to obtain the results. Specifically, first I chose to create my own puzzle instead of using the default ones. Then, I can decide whether I need to solve an 8-puzzle problem, a 15-puzzle problem, or something more complicated. As shown in Figure 4, here I choose 4, which means I try to solve a 15 puzzle. Next, I customized a 15 puzzle state and chose to use Manhattan Distance Heuristic (MDH) to solve it. In the end, I succeeded to get the solution paths, the depth, etc.

```
Welcome to my n-Puzzle Solver. Type '1' to use a default puzzle, or '2' to create your own.
2
Enter the size of the puzzle (e.g., '3' for 8-puzzle, '4' for 15-puzzle, '5' for 24-puzzle, etc.)
4
Enter your puzzle, using a zero to represent the blank. Please only enter valid n-puzzles.
Enter the puzzle demilimiting the numbers with a space. Type RETURN only when finished.
Enter the 1st row: 0 2 3 4
Enter the 2nd row: 1 5 7 8
Enter the 3rd row: 9 6 10 12
Enter the 4th row: 13 14 11 15
Select algorithm. (1) for Uniform Cost Search, (2) for the Misplaced Tile Heuristic, or (3) the Manhattan Distance Heuristic.
3
Solution:
((1, 2, 3, 4), (0, 5, 7, 8), (9, 6, 10, 12), (13, 14, 11, 15))
((1, 2, 3, 4), (5, 0, 7, 8), (9, 6, 10, 12), (13, 14, 11, 15))
((1, 2, 3, 4), (5, 6, 7, 8), (9, 0, 10, 12), (13, 14, 11, 15))
((1, 2, 3, 4), (5, 6, 7, 8), (9, 10, 0, 12), (13, 14, 11, 15))
((1, 2, 3, 4), (5, 6, 7, 8), (9, 10, 11, 12), (13, 14, 0, 15))
((1, 2, 3, 4), (5, 6, 7, 8), (9, 10, 11, 12), (13, 14, 15, 0))
Solution depth: 6
Number of nodes expanded: 6
Max queue size: 10
```

Figure 4: More generalized scenarios (a customized 15-puzzle)

The main function runs trials for each depth in a specified range, generating a random

state of puzzle for each depth and solving it using the three algorithms. The number of nodes expanded and the maximum queue size are recorded for each algorithm. After completing all the trials, I calculated the average values of them to prepare for further visualization.

Results and Discussion

After recording the average values of some important indicators (the number of nodes expanded and the maximum queue size). The algorithms were compared based on these indicators with respect to increasing depth.

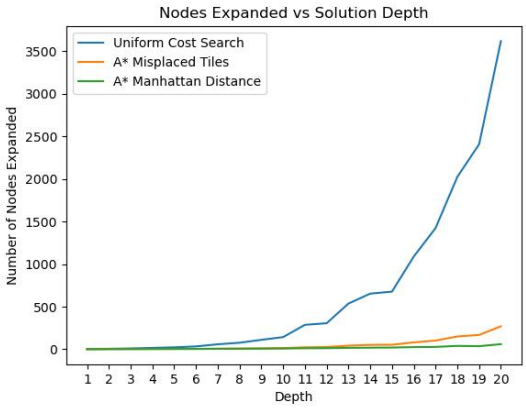

Figure 5

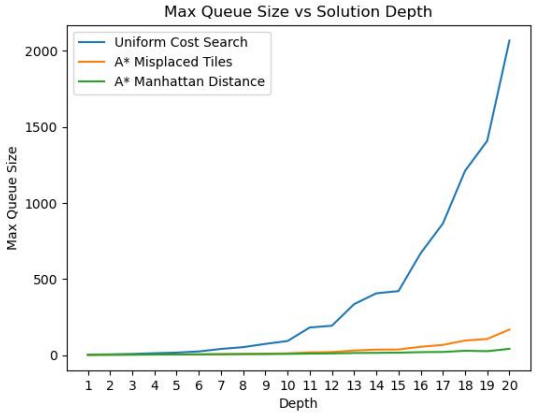

Figure 6

Fortunately, all three algorithms were found to be successfully implemented, but their performance varies significantly when comparing the number of nodes expanded and the maximum queue size. As shown in Figure 5 and Figure 6, if the depth is less than 5, all three algorithms perform similarly. However, when the depth is greater than 5, the performance of UCS is obviously worse than the other two A* algorithms. UCS has significantly larger number of expanded nodes and maximum queue size, which means that solving the same problem may require longer time (as the number of expanded nodes is more) and more memory (as its max queue size is larger).

Despite both MTH and MDH perform much better than UCS, the performance of MTH starts to decline after depth 12, making MDH a better heuristic for larger depths.

However, even though the number of expanded nodes and the maximum queue size can reflect the performance of the algorithms to a large extent, when I really started to analyze the actual performance (such as the time used by the algorithms), I found some interesting phenomena. The evaluation of these algorithms based on the average solve time is shown as Figure 7:

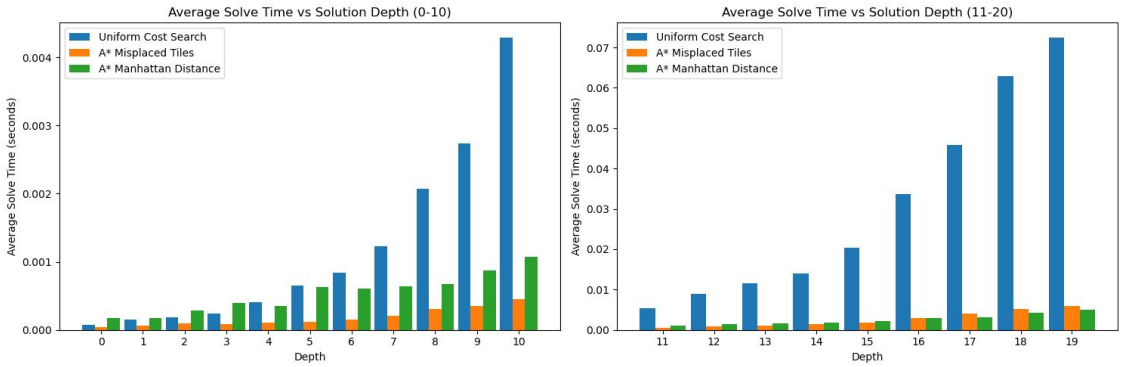

Figure 7: Average Solve Time vs Solution Depth

(As the depth increases, the Average Time of UCS becomes too large, causing some data to be displayed unclearly, so here I grouped them according to different Depths. Please note that the scale of the y-axis in the two subfigures is different)

According to my previous analysis, the performance of MDH should always be the best (because the number of expanded nodes and the maximum queue size of MDH are both the smallest). But in fact, the time required to achieve the goal is always higher than MTH before Depth reaches 17, and even when Depth is less than or equal to 5, it is similar to UCS. After carefully analyzing the reason for this uncommon sense phenomenon, I think it is because MDH needs to calculate the Manhattan distance as its h(n). Compared to MTH's heuristic function (the number of misplaced tiles), it takes longer to calculate the Manhattan distance. Therefore, in practical applications, when the Depth value is small, the performance of MTH may be better than that of MDH.

Summary

In this project, I successfully implemented three search algorithms (Uniform Cost Search, A* with Misplaced Tile Heuristic and A* with Manhattan Distance Heuristic) to solve 8-puzzle problem. By comparing the performance of these three algorithms, we found their advantages and disadvantages in different depths and scenes. At small depths, MTH and MDH perform similarly and are more effective than UCS; however, at large depths, MDH outperforms MTH. Also, I found that in terms of actual running time, MTH can outperform MDH at smaller depths.

Through the practice of this project, I have learned the following experiences and lessons:

1. Choosing an appropriate heuristic algorithm is critical to improving search performance. In different problem scenarios, a suitable heuristic algorithm can greatly reduce the search space and thus improve the search efficiency.
2. In practical applications, in addition to considering the theoretical performance of

the algorithm, it is also necessary to pay attention to the actual running time. Sometimes an algorithm with better theoretical performance may not perform as expected in practice.

3. By implementing visualization and user interface, it can help me better understand the principle of the algorithm, thus providing valuable guidance for further optimization of the algorithm.

Reference

[1]Figures and customized puzzles taken from https://deniz.co/8-puzzle-solver/
[2]priority queue structure (heap queue) imported to insert heuristic function, cost, puzzle states, documentation at https://docs.python.org/3.9/library/heapq.html
[3]default test cases taken from project 1 pdf given by professor
[4]imported numpy library to use array(),arrayequal() and where(), documentation at https://numpy.org/doc/stable/reference/
[5]imported random library to generate samples in different depth for visualization analysis, documentation at https://docs.python.org/3.9/library/random.html
[6]imported time library to calculate the time of solving 8-puzzle problem for visualization analysis, documentation at https://docs.python.org/3.9/library/time.html
[7]imported matplotlib library to draw different charts for visualization analysis documentation at https://matplotlib.org/api/matplotlib_api.html#matplotlib.matplotlib

n_puzzle.py

```
import heapq
import numpy as np

def neighbors(puzzle, n):
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    empty_position = tuple(np.argwhere(puzzle == 0)[0])
    # Given the previous position of empty block, we can get every possible new positions.
    for move in moves:
        new_position = (empty_position[0] + move[0], empty_position[1] + move[1])
        # Valid range
        if (0 <= new_position[0] < n and 0 <= new_position[1] < n):
            new_puzzle = puzzle.copy()
            # Swap the positions
            new_puzzle[empty_position], new_puzzle[new_position] =
new_puzzle[new_position], new_puzzle[empty_position]
            yield new_puzzle, 1

def uniform_cost_search(puzzle, goal, n):
    visited = set()
    queue = [(0, tuple(map(tuple, puzzle)), [])]
    nodes_expanded = 0
    max_queue_size = 1

    while queue:
        # In Uniform Cost Search f(n) = 0, so f(n) = g(n). We only need to consider the actual
cost.
        (cost, current_state, path) = heapq.heappop(queue)
        current_state = np.array(current_state)
        # If we achieve our goal
        if np.array_equal(current_state, goal):
            return path, cost, nodes_expanded, max_queue_size

        # Convert the current state to hash code to test if it is visited.
        current_hash = current_state.tobytes()
        if current_hash not in visited:
            visited.add(current_hash)
            nodes_expanded += 1

            for neighbor, edge_cost in neighbors(current_state, n):
                neighbor_hash = neighbor.tobytes()
                # Only consider unvisited states
                if neighbor_hash not in visited:
                    heapq.heappush(queue, (cost + edge_cost, tuple(map(tuple, neighbor)),
```

```
path + [tuple(map(tuple, neighbor))]))
                    max_queue_size = max(max_queue_size, len(queue))

    return None, None, None, None

# Approach 2:
def misplaced_tiles(puzzle, goal):
    return np.sum(puzzle != goal) - 1  # to exclude the blank tile

def MTH(puzzle, goal, n):
    visited = set()
    queue = [(0 + misplaced_tiles(puzzle, goal), 0, tuple(map(tuple, puzzle)), [])]
    nodes_expanded = 0
    max_queue_size = 1

    while queue:
        (priority, cost, current_state, path) = heapq.heappop(queue)
        current_state = np.array(current_state)
        # If we achieve our goal
        if np.array_equal(current_state, goal):
            return path, cost, nodes_expanded, max_queue_size

        # convert the current state to hash code to test if it is visited.
        current_hash = current_state.tobytes()
        if current_hash not in visited:
            visited.add(current_hash)
            nodes_expanded += 1

            for neighbor, edge_cost in neighbors(current_state, n):
                neighbor_hash = neighbor.tobytes()
                # Only consider unvisited states
                if neighbor_hash not in visited:
                    # f(n) = g(n) + h(n)
                    total_cost = cost + edge_cost + misplaced_tiles(neighbor, goal)
                    heapq.heappush(queue, (total_cost, cost + edge_cost, tuple(map(tuple,
neighbor)), path + [tuple(map(tuple, neighbor))]))
                    max_queue_size = max(max_queue_size, len(queue))

    return None, None, None, None


# Approach 3
def manhattan_distance(puzzle, goal, n):
    distance = 0
```

```
    goal = np.array(goal)
    for i in range(n):
        for j in range(n):
            tile = puzzle[i][j]
            if tile != 0:
                goal_i, goal_j = np.where(goal == tile)
                distance += abs(i - goal_i) + abs(j - goal_j)
    return distance

def MDH(puzzle, goal, n):
    visited = set()
    queue = [(0 + manhattan_distance(puzzle, goal, n), 0, tuple(map(tuple, puzzle)), [])]
    nodes_expanded = 0
    max_queue_size = 1

    while queue:
        (priority, cost, current_state, path) = heapq.heappop(queue)
        current_state = np.array(current_state)
        # If we achieve our goal
        if np.array_equal(current_state, goal):
            return path, cost, nodes_expanded, max_queue_size

        # convert the current state to hash code to test if it is visited.
        current_hash = current_state.tobytes()
        if current_hash not in visited:
            visited.add(current_hash)
            nodes_expanded += 1

            # Only consider unvisited states
            for neighbor, edge_cost in neighbors(current_state, n):
                neighbor_hash = neighbor.tobytes()
                if neighbor_hash not in visited:
                    # f(n) = g(n) + h(n)
                    total_cost = cost + edge_cost + manhattan_distance(neighbor, goal, n)
                    heapq.heappush(queue, (total_cost, cost + edge_cost, tuple(map(tuple,
neighbor)), path + [tuple(map(tuple, neighbor))]))
                    max_queue_size = max(max_queue_size, len(queue))

    return None, None, None, None




goal = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 0]])
```

```
test_cases = [
    np.array([[1, 2, 3], [4, 5, 6], [7, 8, 0]]),
    np.array([[1, 2, 3], [4, 5, 6], [0, 7, 8]]),
    np.array([[1, 2, 3], [5, 0, 6], [4, 7, 8]]),
    np.array([[1, 3, 6], [5, 0, 2], [4, 7, 8]]),
    np.array([[1, 3, 6], [5, 0, 7], [4, 8, 2]]),
    np.array([[1, 6, 7], [5, 0, 3], [4, 8, 2]]),
    np.array([[7, 1, 2], [4, 8, 5], [6, 3, 0]]),
    np.array([[0, 7, 2], [4, 6, 1], [3, 5, 8]])
]


def main():
    print("Welcome to my n-Puzzle Solver. Type '1' to use a default puzzle, or '2' to create
your own.")
    choice = int(input())
    if choice == 1:
        print("Choose one of the eight default puzzles (Type a number from 1 to 8):")
        for i, test_case in enumerate(test_cases, start=1):
            print(f"Test example {i}:\n {test_case}")
        puzzle_choice = int(input())
        n = 3
        puzzle = test_cases[puzzle_choice - 1]
    elif choice == 2:
        print("Enter the size of the puzzle (e.g., '3' for 8-puzzle, '4' for 15-puzzle, '5'
for 24-puzzle, etc.)")
        n = int(input())
        puzzle = []
        print("Enter your puzzle, using a zero to represent the blank. Please only enter
valid n-puzzles. \n"
              "Enter the puzzle demilimiting the numbers with a space. Type RETURN only when
finished.")
        for i in range(n):
            row = list(map(int, input(f"Enter the {ordinal(i + 1)} row: ").split()))
            puzzle.append(row)

    print("Select algorithm. (1) for Uniform Cost Search, (2) for the Misplaced Tile
Heuristic, or (3) the Manhattan Distance Heuristic.")
    algorithm_choice = int(input())

    # Generate the goal array based on different n
    goal = np.array(list(range(1, n * n)) + [0]).reshape(n, n)
```

```
    if algorithm_choice == 1:
        solution, depth, nodes_expanded, max_queue_size = uniform_cost_search(puzzle, goal,
n)
    elif algorithm_choice == 2:
        solution, depth, nodes_expanded, max_queue_size = MTH(puzzle, goal, n)
    elif algorithm_choice == 3:
        solution, depth, nodes_expanded, max_queue_size = MDH(puzzle, goal, n)

    print("Solution:")
    for state in solution:
        print(state)
    print(f"Solution depth: {depth}")
    print(f"Number of nodes expanded: {nodes_expanded}")
    print(f"Max queue size: {max_queue_size}")

# Determine the ordinal of the given row, as we need to consider more complex conditions,
such as 24-puzzle, 35-puzzle, etc.
def ordinal(n):
    if 10 <= n % 100 <= 20:
        suffix = "th"
    else:
        suffix = {1: "st", 2: "nd", 3: "rd"}.get(n % 10, "th")
    return str(n) + suffix


if __name__ == "__main__":
    main()
```