CS205. Introduction to Artificial Intelligence [Project 1] AKASH SURESH BILGI
862395080 abilg003

Introduction:

The 8-puzzle, a captivating sliding puzzle game, has garnered considerable attention from both researchers and AI enthusiasts. Its setup is simple yet deceptive: a 3x3 grid filled with numbered tiles and a mischievous empty space. The quest? Rearrange the tiles, one sneaky slide at a time, until they align perfectly with the desired configuration.

In this project report, our focus revolves around the creation of a program that tackles the 8-puzzle using three distinct search algorithms: the trusty Uniform Cost Search (UCS), the resourceful A* with Manhattan Distance Heuristic, and the crafty A* with Misplaced Tiles Heuristic. These algorithms each offer a unique approach to navigating the puzzle's intricate search space, seeking the most optimal or nearly optimal solution.

Our main objective lies in comparing the performance of these algorithms and scrutinizing their efficiency and effectiveness in solving the 8-puzzle. By assessing crucial factors such as the number of expanded nodes and space requirements, we aim to unravel the strengths and weaknesses inherent in each algorithm, ultimately discerning their suitability for varying problem instances.

This analysis not only sheds light on the capabilities and limitations of these search algorithms in conquering the 8-puzzle but also has the potential to extend its findings to other enigmatic combinatorial problems. By grasping the delicate balance between exploration, efficiency, and solution quality, we can confidently make well-informed decisions when faced with similar perplexing challenges. This project embarks on a delightful journey into the realm of the 8-puzzle and its mesmerizing search algorithms. Through rigorous evaluation and analysis, we unravel their hidden secrets, paving the way for future puzzle-solving triumphs. So brace yourselves, for within these puzzle-filled pages lies a tale of exploration, ingenuity, and the pursuit of optimal solutions. Let us dive into the intricacies of the 8-puzzle and its enthralling search algorithms, leaving no tile unturned in our quest for enlightenment.

Note: I have added link to my git repo at the end of the report, my choice of language is Python 3.

Algorithms:

Uniform Cost Search (UCS): UCS is a best-first search algorithm that expands nodes based on their path cost. In the context of the 8-puzzle problem, the path cost represents the number of moves required to reach the current state. UCS maintains a priority queue of nodes and expands the node with the lowest path cost first, guaranteeing an optimal solution. However, UCS can be computationally expensive due to the vast number of possible states in the search space.

UCS guarantees finding the optimal solution because it explores the search space based on the path cost. It always expands the node with the lowest cost, ensuring that the optimal path is found before considering higher-cost paths. However, this optimality comes at the cost of potentially high time and memory requirements, especially when dealing with large search spaces.

Misplaced Tiles Heuristic: The Misplaced Tiles heuristic is an informed search algorithm that estimates the distance from a given state to the goal state by counting the number of tiles that are not in their desired positions. By summing the number of misplaced tiles, the algorithm guides the search towards the goal state. This heuristic is admissible, ensuring it never overestimates the actual cost to reach the goal. The Misplaced Tiles heuristic prioritizes states that are closer to the goal state, potentially leading to faster solutions.

Manhattan Distance Heuristic: The Manhattan Distance heuristic is another informed search algorithm for the 8-puzzle problem. It estimates the distance between a given state and the goal state by calculating the sum of the Manhattan distances for each tile. Manhattan distance is the sum of the vertical and horizontal distances between the current position of a tile and its desired position. The Manhattan Distance heuristic considers the total sum of Manhattan distances for all the tiles, providing a guide for the search. Like the Misplaced Tiles heuristic, the Manhattan Distance heuristic is admissible and facilitates an efficient search.

The Manhattan Distance Heuristic guides the A* search algorithm by prioritizing states that are closer to the goal state based on the sum of the Manhattan distances for each tile. By considering this heuristic value, the search is biased towards states that are more likely to lead to the goal state. The A* algorithm combines the heuristic value with the cost of reaching a state, ensuring that the optimal solution is found by considering both the estimated distance and the actual cost.

Working Steps:

UCS, Misplaced Tiles Heuristic, and Manhattan Distance Heuristic are three algorithms used in solving the 8-puzzle problem. While they share some common steps, they differ in how they calculate the heuristic values and prioritize the nodes for expansion.

Initialization:

1. Start with the initial state of the puzzle.
2. Create an empty priority queue to store the nodes to be expanded.
3. Assign a cost of 0 to the initial state.

Expansion:

1. Select the node with the lowest priority from the priority queue.
2. Expand the selected node by generating its neighboring states.

3. Calculate the cost of each neighboring state by adding the cost of reaching the current state and the cost of the transition to that state.
4. Calculate the heuristic value for each neighboring state using the respective heuristic (Misplaced Tiles Heuristic for the Misplaced Tiles algorithm, Manhattan Distance Heuristic for the Manhattan Distance algorithm).
5. Add the neighboring states to the priority queue with their corresponding costs and heuristic values.

Goal Test:

1. Check if the expanded node is the goal state.
2. If it is, the algorithm terminates, and the optimal solution is found.
3. If not, continue to the next step.

Repeat:

1. If the goal state is not reached, go back to the Expansion step and repeat the process.
2. Select the node with the lowest priority from the priority queue and expand it.
3. Generate its neighboring states and calculate their costs and heuristic values.
4. Add the neighboring states to the priority queue.
5. Continue until the goal state is found.

The key differences between the algorithms lie in the heuristic calculation and the priority used for node selection:

Misplaced Tiles Heuristic:

1. Calculate the heuristic value for the initial state using the Misplaced Tiles Heuristic, which counts the number of tiles that are not in their desired positions.
2. The priority queue stores nodes based on the sum of their cost and the heuristic value.
3. The node with the lowest priority (cost + heuristic value) is selected for expansion.

Manhattan Distance Heuristic:

1. Calculate the heuristic value for the initial state using the Manhattan Distance Heuristic.
2. For each tile, calculate the Manhattan distance between its current position and its desired position.
3. The Manhattan distance is the sum of the absolute differences between the x-coordinates and y-coordinates of the two positions.
4. Sum up the Manhattan distances for all the tiles to obtain the heuristic value.

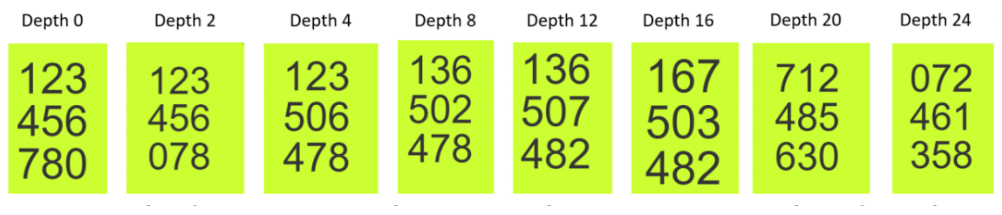
5. The priority queue stores nodes based on the sum of their cost and the heuristic value.
6. The node with the lowest priority (cost + heuristic value) is selected for expansion.

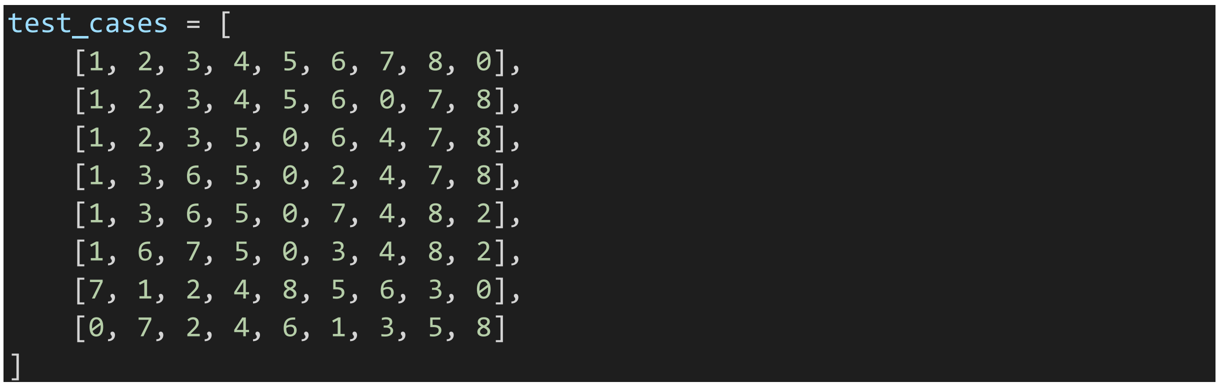
It's important to note that while UCS explores the most nodes and uses the most space, both the Misplaced Tiles Heuristic and the Manhattan Distance Heuristic algorithms strike a balance between exploration and efficiency. The Misplaced Tiles Heuristic expands fewer nodes compared to UCS, while the Manhattan Distance Heuristic expands the fewest nodes. Additionally, in terms of space usage, UCS and the Misplaced Tiles Heuristic generally require less space compared to the Manhattan Distance Heuristic.

The choice of algorithm depends on the specific requirements of the problem and the available resources. UCS is suitable when finding the optimal solution is crucial and computational resources are not a constraint. The Misplaced Tiles Heuristic offers a compromise between speed and solution quality when efficiency and limited computational resources are a priority. The Manhattan Distance Heuristic, with its focus on informed search, is suitable when faster solutions are desired, even at the expense of higher computational requirements.

Comparison of the 3 Algorithms:

Here I have plotted the 8 test cases given by Dr. Keogh (1-8) in the handout with depth 0-24.



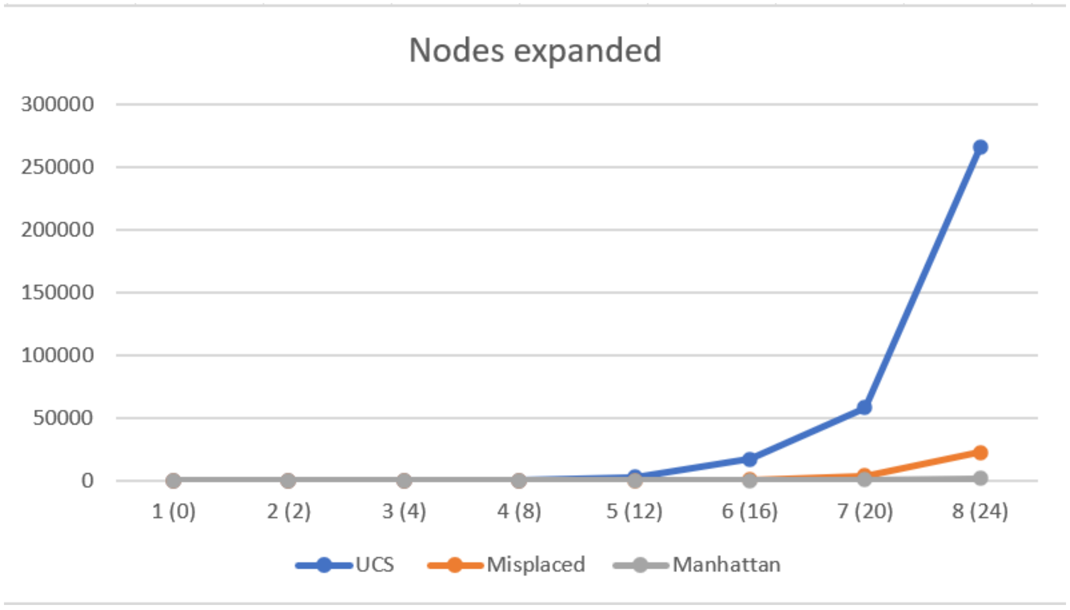


```
test_cases = [
    [1, 2, 3, 4, 5, 6, 7, 8, 0],
    [1, 2, 3, 4, 5, 6, 0, 7, 8],
    [1, 2, 3, 5, 0, 6, 4, 7, 8],
    [1, 3, 6, 5, 0, 2, 4, 7, 8],
    [1, 3, 6, 5, 0, 7, 4, 8, 2],
    [1, 6, 7, 5, 0, 3, 4, 8, 2],
    [7, 1, 2, 4, 8, 5, 6, 3, 0],
    [0, 7, 2, 4, 6, 1, 3, 5, 8]
]
```

The first shows number of nodes each algorithms had to expand to get the final state.

X-axis = testcase(depth)

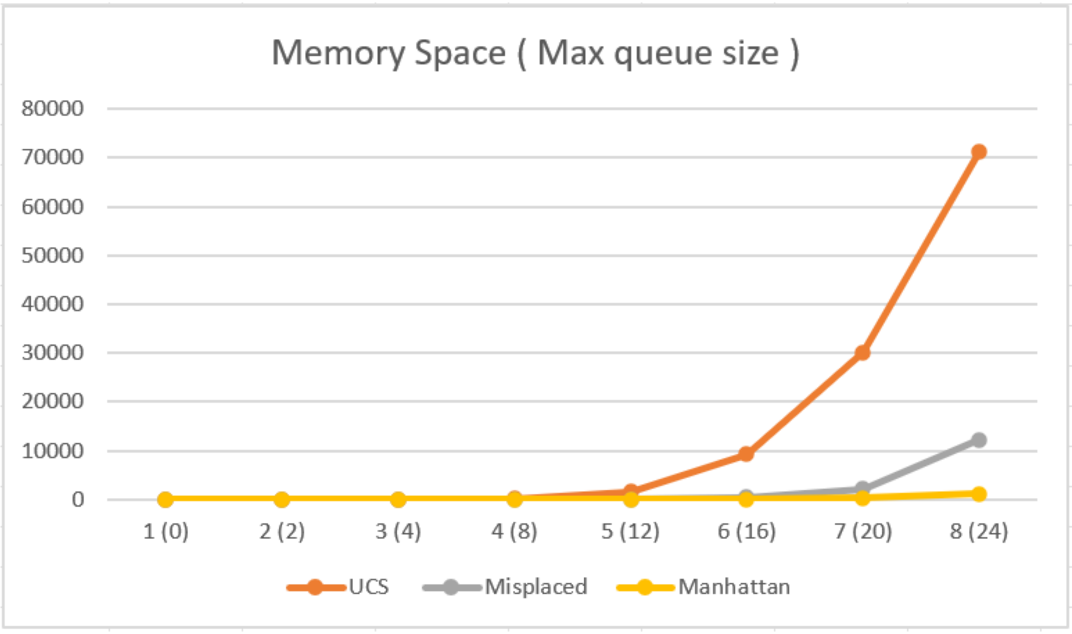
Y-axis = Number of Nodes



Second one shows the usage of memory space by each algorithm:

X-axis = testcase(depth)

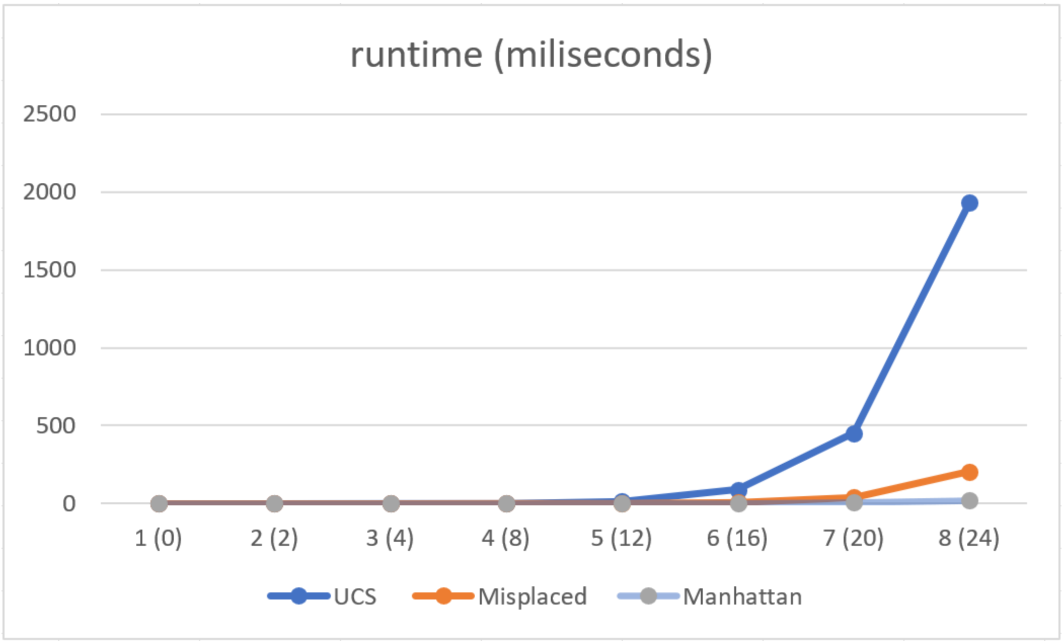
Y-axis = maximum size of queue



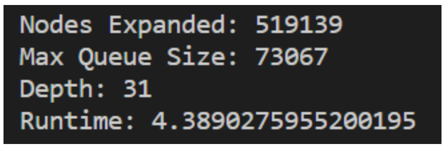
We see the same pattern with runtime as well:

X-axis = testcase(depth)

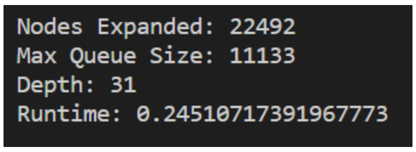
Y-axis = runtime in miliseconds



Sample (Hardest) 31-depth testcase(UCS vs A* Manhattan) i.e. : 8 6 7 2 5 4 3 0 1 [3]



```
Nodes Expanded: 519139
Max Queue Size: 73067
Depth: 31
Runtime: 4.3890275955200195
```



```
Nodes Expanded: 22492
Max Queue Size: 11133
Depth: 31
Runtime: 0.24510717391967773
```

Based on the analysis of the data, here is a comparison of the three algorithms:

- Uniform Cost Search (UCS): UCS expands the most nodes among the three algorithms and uses the most space. It only considers the path cost and does not incorporate any heuristic information. However, UCS may take longer to find a solution due to the exhaustive exploration of the search space.

- Misplaced Tiles Heuristic: The Misplaced Tiles heuristic expands fewer nodes compared to UCS and uses less space than UCS. By using the number of misplaced tiles as a guide, this heuristic focuses on states that are more likely to lead to the goal state. It strikes a balance between exploration and efficiency.

- Manhattan Distance Heuristic: The Manhattan Distance heuristic expands the fewest nodes and uses less space compared to both UCS and the Misplaced Tiles heuristic. It combines path cost and the estimated distance to the goal state using the Manhattan distance heuristic. While it explores a smaller number of nodes, it benefits from the informed search approach, potentially leading to faster solutions.

Regarding space usage, the data indicates that UCS and the Misplaced Tiles heuristic generally require more space compared to the Manhattan Distance heuristic. UCS needs to maintain information about the path cost, while the Misplaced Tiles heuristic considers the number of misplaced tiles. These algorithms may require additional storage for heuristic calculations or data structures.

On the other hand, the Manhattan Distance heuristic expands the fewest nodes, which means it needs to store information about a smaller number of states in the search space. This results in lower space requirements compared to UCS and the Misplaced Tiles heuristic.

Conclusion:

In this project, I conducted a comparative analysis of three search algorithms - Uniform Cost Search (UCS), Misplaced Tiles Heuristic, and Manhattan Distance Heuristic - in solving the 8-puzzle problem. Each algorithm offers a different approach to navigate the search space and find an optimal or near-optimal solution.

Based on the analysis of the data, Uniform Cost Search expands the most nodes and uses the most space, while the Misplaced Tiles heuristic expands fewer nodes and uses less space. The Manhattan Distance heuristic, with its focus on informed search, expands the fewest nodes and uses the least space.

The choice of algorithm depends on the specific requirements of the problem and the available resources. For example, if finding the optimal solution is crucial and computational resources are not a constraint, Uniform Cost Search may be the preferred choice. On the other hand, if efficiency is a priority and computational resources are limited, the Misplaced Tiles heuristic can offer a good compromise between speed and solution quality. The Manhattan Distance heuristic, with its focus on informed search, is suitable when faster solutions are desired, even at the expense of higher computational requirements.

In conclusion, the study of search algorithms in solving the 8-puzzle problem provides valuable insights into their performance characteristics. The Uniform Cost Search, Misplaced Tiles heuristic, and Manhattan Distance heuristic each offer distinct advantages and trade-offs. By comparing their performance in terms of nodes expanded and space usage, we can make informed decisions when selecting the most suitable algorithm for a given problem.

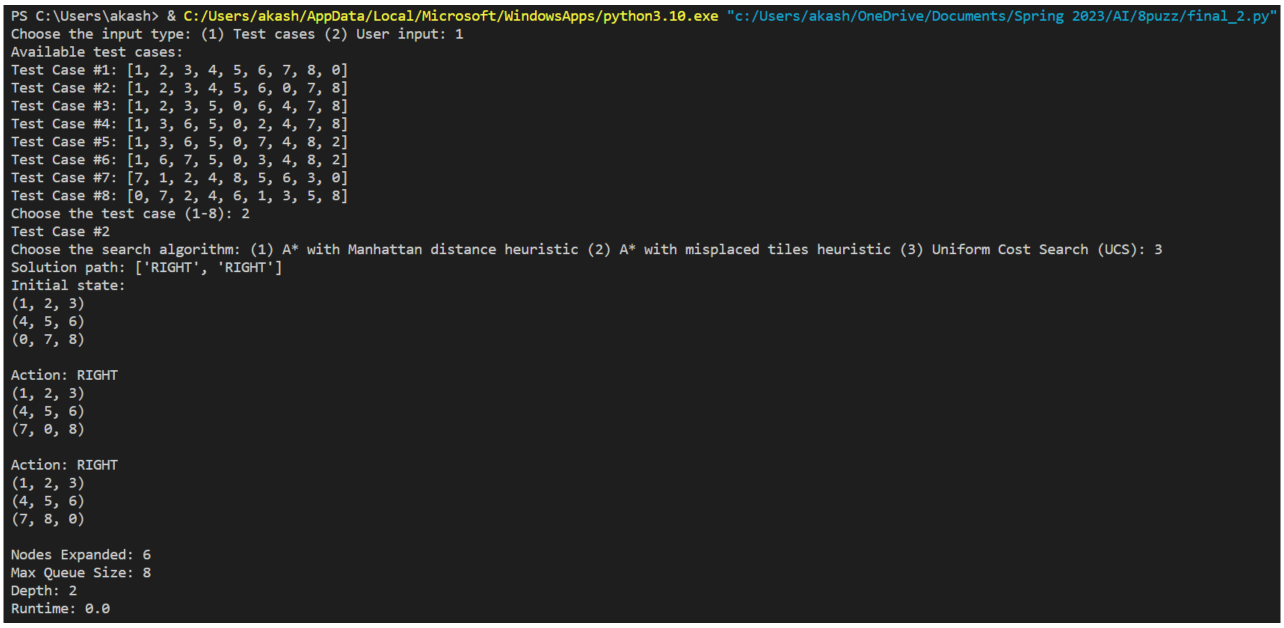
Example stack trace:

```
PS C:\Users\akash> & C:/Users/akash/AppData/Local/Microsoft/WindowsApps/python3.10.exe "c:/Users/akash/OneDrive/Documents/Spring 2023/AI/8puzz/final_2.py"

Choose the input type: (1) Test cases (2) User input: 1
Available test cases:
Test Case #1: [1, 2, 3, 4, 5, 6, 7, 8, 0]
Test Case #2: [1, 2, 3, 4, 5, 6, 0, 7, 8]
Test Case #3: [1, 2, 3, 5, 0, 6, 4, 7, 8]
Test Case #4: [1, 3, 6, 5, 0, 2, 4, 7, 8]
Test Case #5: [1, 3, 6, 5, 0, 7, 4, 8, 2]
Test Case #6: [1, 6, 7, 5, 0, 3, 4, 8, 2]
Test Case #7: [7, 1, 2, 4, 8, 5, 6, 3, 0]
Test Case #8: [0, 7, 2, 4, 6, 1, 3, 5, 8]

Choose the test case (1-8): 2
Test Case #2
```

Choose the search algorithm: (1) A* with Manhattan distance heuristic (2) A* with misplaced tiles heuristic (3) Uniform Cost Search (UCS): 3
Solution path: ['RIGHT', 'RIGHT']
Initial state:
(1, 2, 3)
(4, 5, 6)
(0, 7, 8)
Action: RIGHT
(1, 2, 3)
(4, 5, 6)
(7, 0, 8)
Action: RIGHT
(1, 2, 3)
(4, 5, 6)
(7, 8, 0)
Nodes Expanded: 6
Max Queue Size: 8
Depth: 2



```
PS C:\Users\akash> & C:/Users/akash/AppData/Local/Microsoft/WindowsApps/python3.10.exe "c:/Users/akash/OneDrive/Documents/Spring 2023/AI/8puzz/final_2.py"
Choose the input type: (1) Test cases (2) User input: 1
Available test cases:
Test Case #1: [1, 2, 3, 4, 5, 6, 7, 8, 0]
Test Case #2: [1, 2, 3, 4, 5, 6, 0, 7, 8]
Test Case #3: [1, 2, 3, 5, 0, 6, 4, 7, 8]
Test Case #4: [1, 3, 6, 5, 0, 2, 4, 7, 8]
Test Case #5: [1, 3, 6, 5, 0, 7, 4, 8, 2]
Test Case #6: [1, 6, 7, 5, 0, 3, 4, 8, 2]
Test Case #7: [7, 1, 2, 4, 8, 5, 6, 3, 0]
Test Case #8: [0, 7, 2, 4, 6, 1, 3, 5, 8]
Choose the test case (1-8): 2
Test Case #2
Choose the search algorithm: (1) A* with Manhattan distance heuristic (2) A* with misplaced tiles heuristic (3) Uniform Cost Search (UCS): 3
Solution path: ['RIGHT', 'RIGHT']
Initial state:
(1, 2, 3)
(4, 5, 6)
(0, 7, 8)

Action: RIGHT
(1, 2, 3)
(4, 5, 6)
(7, 0, 8)

Action: RIGHT
(1, 2, 3)
(4, 5, 6)
(7, 8, 0)

Nodes Expanded: 6
Max Queue Size: 8
Depth: 2
Runtime: 0.0
```

My code:

```
from queue import PriorityQueue
import time

def make_node(state, parent=None, action=None, depth=0, cost=0):
    return {
        'STATE': state,    # the current state of the node
        'PARENT': parent,  # the parent node
        'ACTION': action,  # the action taken to get to the current state from
the parent node
```

AKASH SURESH BILGI
abilg003

```
        'DEPTH': depth,    # the depth of the node in the search tree
        'COST': cost       # the cost to reach the current node from the
initial state
    }

def expand(node, operators):
    expanded_nodes = []
    for operator in operators:
        new_state = apply_operator(node['STATE'], operator)  # apply each
operator to the current state to generate a new state
        if new_state:
            child_node = make_node(new_state, parent=node, action=operator,
depth=node['DEPTH'] + 1, cost=node['COST'] + 1)  # create a child node with
the new state and other information
            expanded_nodes.append(child_node)   # add the child node to the
list of expanded nodes
    return expanded_nodes   # return the list of expanded nodes

def apply_operator(state, operator):
    # Implement the logic to apply an operator on the state
    # and return the new state if valid, otherwise return None
    if state is None:
        return None

    new_state = list(state)
    empty_tile_index = new_state.index(0)

    if operator == 'UP':
        # Check if the empty tile can move up
        if empty_tile_index >= 3:
            # Swap the empty tile with the tile above it
            new_state[empty_tile_index], new_state[empty_tile_index - 3] =
new_state[empty_tile_index - 3], new_state[empty_tile_index]
            return tuple(new_state)
    elif operator == 'DOWN':
        # Check if the empty tile can move down
        if empty_tile_index < 6:
            # Swap the empty tile with the tile below it
            new_state[empty_tile_index], new_state[empty_tile_index + 3] =
new_state[empty_tile_index + 3], new_state[empty_tile_index]
            return tuple(new_state)
    elif operator == 'LEFT':
        # Check if the empty tile can move left
        if empty_tile_index % 3 != 0:
            # Swap the empty tile with the tile to the left of it
            new_state[empty_tile_index], new_state[empty_tile_index - 1] =
new_state[empty_tile_index - 1], new_state[empty_tile_index]
            return tuple(new_state)
    elif operator == 'RIGHT':
        # Check if the empty tile can move right
        if empty_tile_index % 3 != 2:
            # Swap the empty tile with the tile to the right of it
            new_state[empty_tile_index], new_state[empty_tile_index + 1] =
new_state[empty_tile_index + 1], new_state[empty_tile_index]
            return tuple(new_state)

    return None

def goal_test(state):
    return state == (1, 2, 3, 4, 5, 6, 7, 8, 0)

def a_star_heuristic_manhattan(state):
    # Manhattan distance heuristic
    h = 0
    for i in range(len(state)):
        if state[i] != 0:
            goal_row = (state[i] - 1) // 3  # Calculate the row index of the
goal position for the current tile value
            goal_col = (state[i] - 1) % 3   # Calculate the column index of
the goal position for the current tile value
            current_row = i // 3            # Calculate the current row index
for the tile value
            current_col = i % 3             # Calculate the current column
index for the tile value
            h += abs(goal_row - current_row) + abs(goal_col - current_col)  #
Compute the Manhattan distance between the current and goal positions
    return h

def a_star_heuristic_misplaced(state):
    # Misplaced tiles heuristic
    misplaced = 0
    for i in range(len(state)):
        if state[i] != 0 and state[i] != i + 1:  # Check if the tile value is
not 0 and is misplaced
            misplaced += 1  # Increment the count of misplaced tiles
    return misplaced

def general_search(problem, queueing_function, use_heuristic=False):
    nodes = PriorityQueue()
    initial_node = make_node(problem['INITIAL_STATE'])
    if use_heuristic == 1:
        initial_cost = initial_node['COST'] +
a_star_heuristic_manhattan(problem['INITIAL_STATE'])
    elif use_heuristic == 2:
         initial_cost = initial_node['COST'] +
a_star_heuristic_misplaced(problem['INITIAL_STATE'])
```

```
    else:
        initial_cost = initial_node['COST']
    nodes.put((initial_cost, id(initial_node), initial_node))
    visited = set()
    max_queue_size = 1
    nodes_expanded = 0

    while not nodes.empty():
        _, _, node = nodes.get()

        if goal_test(node['STATE']):
            return node, max_queue_size, nodes_expanded

        visited.add(node['STATE'])
        nodes_expanded += 1

        for child_node in expand(node, problem['OPERATORS']):
            if child_node['STATE'] not in visited:
                if use_heuristic == 1:
                    cost = child_node['COST'] +
a_star_heuristic_manhattan(child_node['STATE'])
                elif use_heuristic == 2:
                    cost = child_node['COST'] +
a_star_heuristic_misplaced(child_node['STATE'])
                else:
                    cost = child_node['COST']
                nodes.put((cost, id(child_node), child_node))
                if nodes.qsize() > max_queue_size:
                    max_queue_size = nodes.qsize()

    return "failure", max_queue_size, nodes_expanded

# Define the problem
problem = {
    'INITIAL_STATE': None,
    'OPERATORS': ['UP', 'DOWN', 'LEFT', 'RIGHT'],  # Define the available
operators for moving the tiles
}

# Function to print the state
def print_state(state):
    if state is None:
        print("Invalid state")
        return

    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()
```

```python
# Default test cases given ny Dr. Keogh - (all solvable)
test_cases = [
    [1, 2, 3, 4, 5, 6, 7, 8, 0],
    [1, 2, 3, 4, 5, 6, 0, 7, 8],
    [1, 2, 3, 5, 0, 6, 4, 7, 8],
    [1, 3, 6, 5, 0, 2, 4, 7, 8],
    [1, 3, 6, 5, 0, 7, 4, 8, 2],
    [1, 6, 7, 5, 0, 3, 4, 8, 2],
    [7, 1, 2, 4, 8, 5, 6, 3, 0],
    [0, 7, 2, 4, 6, 1, 3, 5, 8]
]

# Prompt the user to choose the input type
input_type = input("Choose the input type: (1) Test cases (2) User input: ")

if input_type == '1':
    # Use test cases
    print("Available test cases:")
    for i, test_case in enumerate(test_cases):
        print(f"Test Case #{i+1}: {test_case}")

    TC = int(input("Choose the test case (1-8): "))
    print(f"Test Case #{TC}")
    problem['INITIAL_STATE'] = tuple(test_cases[TC - 1])

elif input_type == '2':
    # User input
    user_input = input("Enter the initial state of the 3x3 grid (space-
separated numbers from 0 to 8, e.g., '1 2 3 4 5 6 7 8 0'): ")
    initial_state = tuple(map(int, user_input.split()))

    problem['INITIAL_STATE'] = initial_state

else:
    print("Invalid input type. Please try again.")
    #return

# Prompt the user to choose the search algorithm
algorithm = input("Choose the search algorithm: (1) A* with Manhattan distance
heuristic (2) A* with misplaced tiles heuristic (3) Uniform Cost Search (UCS):
")
use_heuristic = int(algorithm)

start_time = time.time() # start timer
solution, max_queue_size, nodes_expanded = general_search(problem,
queueing_function=PriorityQueue, use_heuristic=use_heuristic)
runtime = time.time() - start_time # end timer
```

```python
# Print the solution
if solution == "failure":
    print("Failed to find a solution.")
else:
    path = []
    while solution:
        if solution['ACTION'] is not None:
            path.insert(0, solution['ACTION'])
        solution = solution['PARENT']

    print("Solution path:", path)

    # Display the final state
    print("Initial state:")
    print_state(problem['INITIAL_STATE'])

    # Apply actions to the initial state to reach the goal state
    current_state = problem['INITIAL_STATE']
    for action in path:
        current_state = apply_operator(current_state, action)
        if current_state is None:
            print(f"Action: {action}")
            print("Invalid state")
        else:
            print(f"Action: {action}")
            print_state(current_state)

    # Print additional information
    print("Nodes Expanded:", nodes_expanded)
    print("Max Queue Size:", max_queue_size)
    print("Depth:", len(path))
    print("Runtime:", runtime)
    print()
```

GitHub Repo: https://github.com/akashbilgi/8_puzzle/blob/main/8_puzzle.py

References:
[1] https://www.geeksforgeeks.org/check-instance-8-puzzle-solvable/

[2] https://blog.goodaudience.com/solving-8-puzzle-using-a-algorithm-7b509c331288

[3] https://52xenos.blogspot.com/2016/11/the-hardest-8-puzzle.html