Experiment No.4
Implement 8-Puzzle problem using A* Search algorithm.
Date of Performance: 12/02/2025
Date of Submission: 17/02/2025

Aim: Study and Implementation of 8-Puzzle problem using A* Search algorithm.

Objective: To study the informed searching techniques and its implementation for problem solving.

Theory:

8-Puzzle Problem:

The 8-puzzle problem consists of a 3×3 grid containing 8 numbered tiles (1 to 8) and one empty space. The goal is to reach a predefined arrangement by moving the tiles in the available space.

Initial State: A given unsolved configuration.

Goal State: The desired arrangement of tiles.

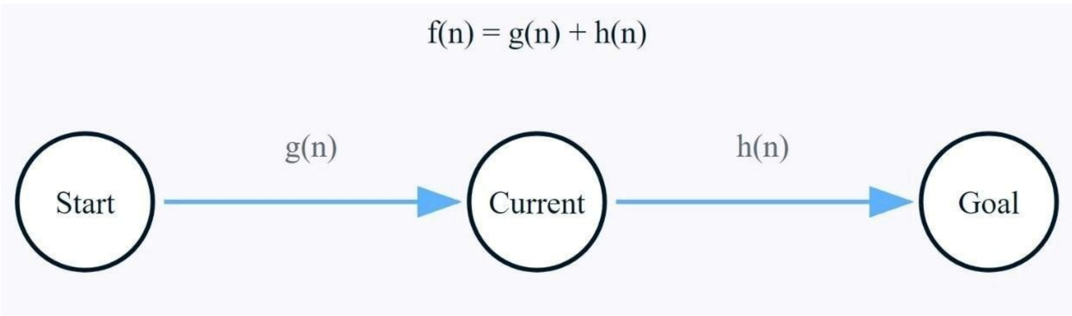
Operators: Movement of tiles in four possible directions (up, down, left, right).

Cost: Each tile movement has a uniform cost.

A Search Algorithm:

A* is an informed search algorithm that combines:

g(n): The cost from the start node to the current node. h(n): The estimated cost from the current node to the goal (heuristic function). f(n) = g(n) + h(n): The total estimated cost of the cheapest path.



Heuristic Functions:

1. **Manhattan Distance:**
   - Sum of the absolute differences between the current position and the goal position of each tile.
2. **Misplaced Tiles:**
   - Counts the number of misplaced tiles compared to the goal state.

A* uses these heuristics to prioritize nodes with the lowest estimated total cost, ensuring an optimal solution.

A* Algorithm-

- The implementation of A* Algorithm involves maintaining two lists- OPEN and CLOSED.
- OPEN contains those nodes that have been evaluated by the heuristic function but have not been expanded into successors yet.
- CLOSED contains those nodes that have already been visited.

Step 1: Define a list OPEN.

- Initially, OPEN consists solely of a single node, the start node S.

Step 2: If the list is empty, return failure and exit.

Step 3:

- Remove node n with the smallest value of f(n) from OPEN and move it to list CLOSED.
- If node n is a goal state, return success and exit.

Step 4: Expand node n.

Step 5:

- If any successor to n is the goal node, return success and the solution by tracing the path from goal node to S.
- Otherwise, go to Step-06. **Step 6:** For each successor node,
- Apply the evaluation function f to the node.
- If the node has not been in either list, add it to OPEN.

Step 7:

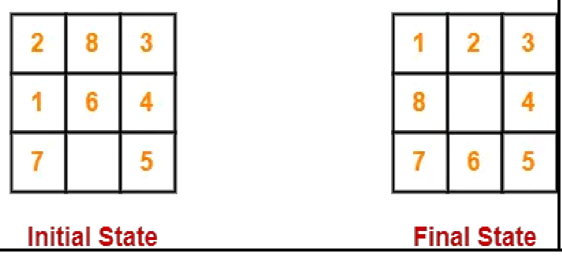
Go back to Step-02.

Problem-01:

Given an initial state of a 8-puzzle problem and final state to be reached-

---

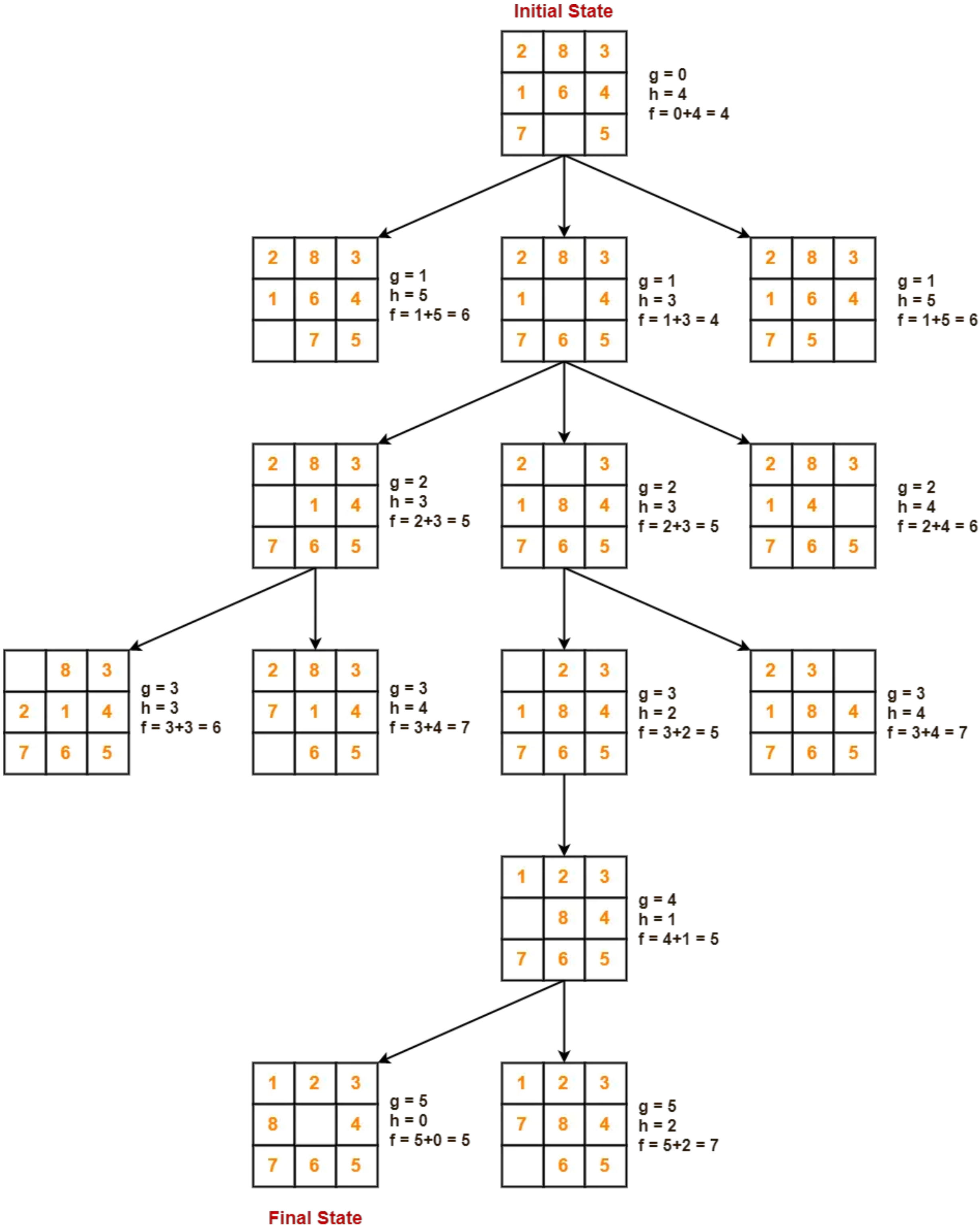
- Find the most cost-effective path to reach the final state from initial state using A* Algorithm.



Consider g(n) = Depth of node and h(n) = Number of misplaced tiles.

Solution:

- A* Algorithm maintains a tree of paths originating at the initial state.
- It extends those paths one edge at a time.
- It continues until the final state is reached.


Initial State
g = 0
h = 4
f = 0+4 = 4
g = 1
h = 5
f = 1+5 = 6
g = 1
h = 3
f = 1+3 = 4
g = 1
h = 5
f = 1+5 = 6
g = 2
h = 3
f = 2+3 = 5
g = 2
h = 3
f = 2+3 = 5
g = 2
h = 4
f = 2+4 = 6
g = 3
h = 3
f = 3+3 = 6
g = 3
h = 4
f = 3+4 = 7
g = 3
h = 2
f = 3+2 = 5
g = 3
h = 4
f = 3+4 = 7
g = 4
h = 1
f = 4+1 = 5
g = 5
h = 0
f = 5+0 = 5
g = 5
h = 2
f = 5+2 = 7
Final State

Code:

```
import heapq import
networkx as nx
import matplotlib.pyplot as plt


# Define the given initial and goal states
INITIAL_STATE = [[1, 2, 3], [4, 0, 6], [7, 5, 8]]
GOAL_STATE = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]


MOVES = {"UP": (-1, 0), "DOWN": (1, 0), "LEFT": (0, -1), "RIGHT": (0, 1)}


def manhattan_distance(state):
    distance = 0 for
    i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:
                goal_x, goal_y = [(row, col) for row in range(3) for col in range(3) if
GOAL_STATE[row][col] == value][0]
                distance += abs(goal_x - i) + abs(goal_y - j)
    return distance
```

```
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None


def generate_new_state(state, move):
    x, y = find_blank(state) dx, dy = MOVES[move] new_x, new_y = x + dx, y + dy if 0 <=
    new_x < 3 and 0 <= new_y < 3: new_state = [row[:] for row in state] new_state[x][y],
    new_state[new_x][new_y] = new_state[new_x][new_y], new_state[x][y] return new_state
    return None


def state_to_tuple(state):
    return tuple(tuple(row) for row in state)


def is_solvable(state):
    flat_list = [num for row in state for num in row if num != 0]
    inversions = sum(1 for i in range(len(flat_list)) for j in range(i + 1, len(flat_list)) if flat_list[i]
> flat_list[j])
    return inversions % 2 == 0
```

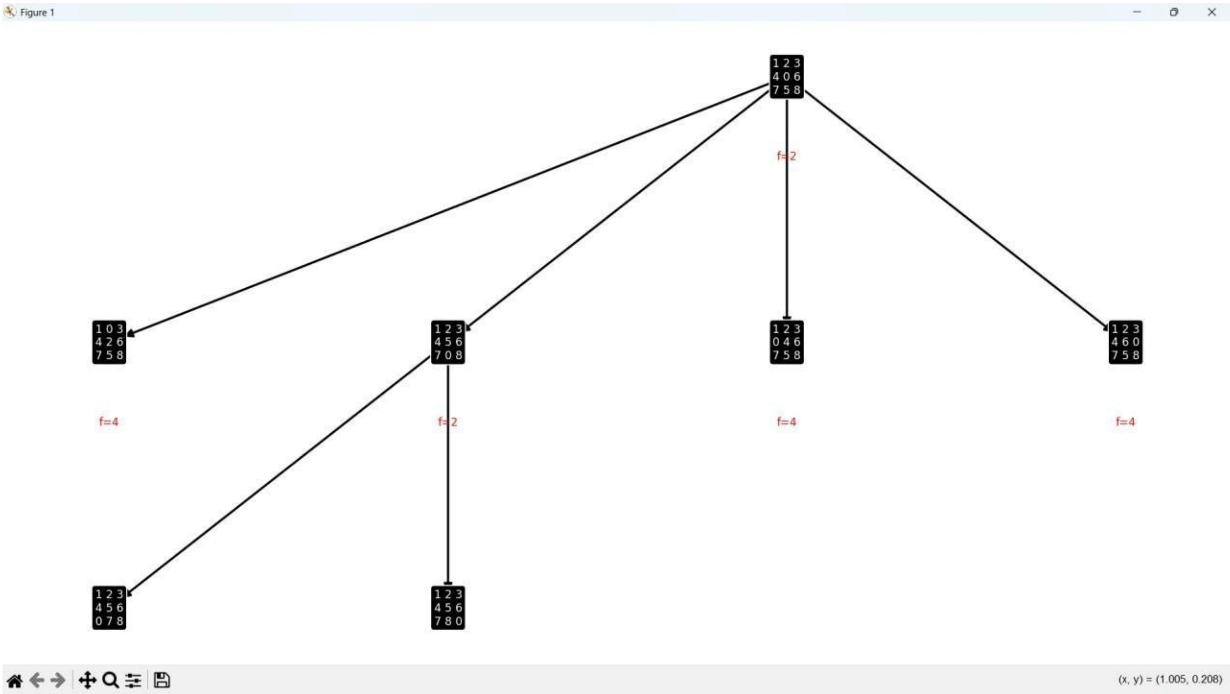
```
def a_star_search(initial_state): if
    not is_solvable(initial_state):
    print("Given initial state is
    unsolvable!") return None,
    None

    open_list = [] heapq.heappush(open_list, (manhattan_distance(initial_state), 0,
    initial_state, [])) visited = set()
    parent_map = {}

    while open_list:
        _, cost, current_state, path = heapq.heappop(open_list)
        if current_state == GOAL_STATE:
            return path + [current_state], parent_map

        visited.add(state_to_tuple(current_state)) for
        move in MOVES.keys():
            new_state = generate_new_state(current_state, move) if
            new_state and state_to_tuple(new_state) not in visited:
                new_cost = cost + 1
                heapq.heappush(open_list, (new_cost + manhattan_distance(new_state), new_cost,
new_state, path + [new_state]))
```

```
            parent_map[state_to_tuple(new_state)] = state_to_tuple(current_state)

    return None, None

def visualize_tree_with_f_values(initial_state, solution_path, parent_map, max_depth=6):
    G = nx.DiGraph() pos
    = {}
    f_values = {}

    queue = [(state_to_tuple(initial_state), (0, 0), 0, 0)]
    pos[state_to_tuple(initial_state)] = (0, 0)
    f_values[state_to_tuple(initial_state)] = manhattan_distance(initial_state)

    while queue:
        node, position, depth, f_value = queue.pop(0) if
        depth >= max_depth:
            continue

        children = [child for child, parent in parent_map.items() if parent == node]
        num_children = len(children) start_x = position[0] - num_children / 2 y =
        position[1] - 1
```

```
        for i, child in enumerate(children):
            child_state = tuple(map(tuple, child)) g_value = solution_path.index(child)
            if child in solution_path else depth + 1 h_value =
            manhattan_distance(child) f_value = g_value + h_value
            f_values[child_state] = f_value
            child_pos = (start_x + i, y) pos[child_state] =
            child_pos queue.append((child, child_pos, depth +
            1, f_value))
            G.add_edge(node, child_state)

    # Improved visualization plt.figure(figsize=(12,
    8))
    nx.draw(G, pos, with_labels=False, node_size=800, edge_color="black", width=2,
arrows=True, arrowstyle='-|>', arrowsize=15)

    # Node colors
    colors = [] for node
    in G.nodes:
        if node == state_to_tuple(GOAL_STATE):
            colors.append("red") # Goal node in red elif node in
        [state_to_tuple(state) for state in solution_path]:
            colors.append("blue") # Solution path in blue else:
```

```
        colors.append("green") # Other nodes in green

    nx.draw_networkx_nodes(G, pos, node_color=colors, node_size=800, edgecolors="black")

    # Add text labels with f-values for
    node, (x, y) in pos.items():
        matrix_str = "\n".join([" ".join(map(str, row)) for row in node])
        f_text = f"f={f_values[node]}" plt.text(x, y, matrix_str, fontsize=10, ha="center",
        va="center", color="white", bbox=dict(facecolor="black", edgecolor="white",
        boxstyle="round,pad=0.3"))
        plt.text(x, y - 0.3, f_text, fontsize=10, ha="center", va="center", color="red")

    # Graph title plt.title("8-Puzzle A* Search Visualization with f-values", fontsize=14,
    fontweight="bold") plt.show()

solution_path, parent_map = a_star_search(INITIAL_STATE) if
solution_path:
    visualize_tree_with_f_values(INITIAL_STATE, solution_path, parent_map, max_depth=6)
else:
    print("No solution found")
```

Output:



Conclusion:

The A* search algorithm is a powerful tool for solving the 8-puzzle, as it efficiently finds the shortest path by combining both the actual path cost and the estimated cost to the goal. The Manhattan distance heuristic is particularly well-suited for this puzzle, as it accurately measures how far each tile is from its goal position.