INTRODUCTION TO AI MSE 1 REPORT

1. Title Page

- **Title**: 8-Puzzle Solver Using A* Algorithm
- **Your Name**: Ram Ji
- **Roll Number**: 202401100400152
- **Course Name**: Introduction to AI
- **Institution Name**: KIET GROUP OF INSTITUTIONS
- **Date**: 10-03-2025

---

2. Introduction

- **What is the 8-Puzzle Problem?**
    - The 8-puzzle is a classic sliding puzzle consisting of a 3x3 grid with 8 numbered tiles and one empty space (represented by 0).
    - The goal is to rearrange the tiles from a given initial state to reach the goal state ([[0, 1, 2], [3, 4, 5], [6, 7, 8]]) by sliding the tiles into the empty space.
- **Why is it Important?**

- It is a fundamental problem in artificial intelligence and search algorithms.
- It helps in understanding heuristic search techniques like the A* algorithm.

- **Objective of the Project**:
  - To implement an 8-puzzle solver using the A* algorithm in Python.
  - To find the shortest path from the initial state to the goal state.

---

3. Methodology

- *A Algorithm Overview**:
  - A* is a search algorithm that finds the shortest path from an initial state to a goal state.
  - It uses two key components:
    1. **Cost to reach the current state (g)**: The number of moves taken to reach the current state from the initial state.
    2. **Heuristic estimate (h)**: An estimate of the cost to reach the goal state from the current state (we used **Manhattan Distance**).
- *Steps in the A Algorithm**:
  1. Start with the initial state.
  2. Add the initial state to the open_set (priority queue).
  3. While the open_set is not empty:

        - Remove the state with the lowest cost (g + h) from the open_set.
        - If it is the goal state, return the solution.
        - Add the state to the closed_set (to avoid revisiting).
        - Generate all possible neighbors (next states) by moving the empty tile.
        - Add valid neighbors to the open_set.
    4. If no solution is found, return "No solution."
- **Heuristic Used: Manhattan Distance**:
    - For each tile, calculate the distance between its current position and its goal position.
    - Sum these distances for all tiles to get the heuristic value.

4. Code Typed

Include the full Python code here. You can copy the code I provided earlier and format it neatly in your report. Use a monospace font (like Courier New) for the code to make it look clean.

```
from queue import PriorityQueue


# 8-Puzzle Solver using A* Algorithm


# Class to represent the state of the puzzle
```

```
class PuzzleState:
    def __init__(self, board, parent=None, move=""):
        self.board = board  # Current state of the board
        self.parent = parent  # Parent state
        self.move = move  # Move taken to reach this state
        self.g = 0  # Cost to reach this state
        self.h = self.calculate_heuristic()  # Heuristic value

    # Calculate the heuristic (Manhattan distance)
    def calculate_heuristic(self):
        distance = 0
        goal = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]  # Goal state
        for i in range(3):
            for j in range(3):
                if self.board[i][j] != 0:  # Ignore empty tile
                    x, y = divmod(self.board[i][j] - 1, 3)  # Expected position
                    distance += abs(x - i) + abs(y - j)  # Manhattan distance
        return distance

    # Check if the current state is the goal state
    def is_goal(self):
        return self.h == 0

    # Generate all possible next states
    def get_neighbors(self):
```

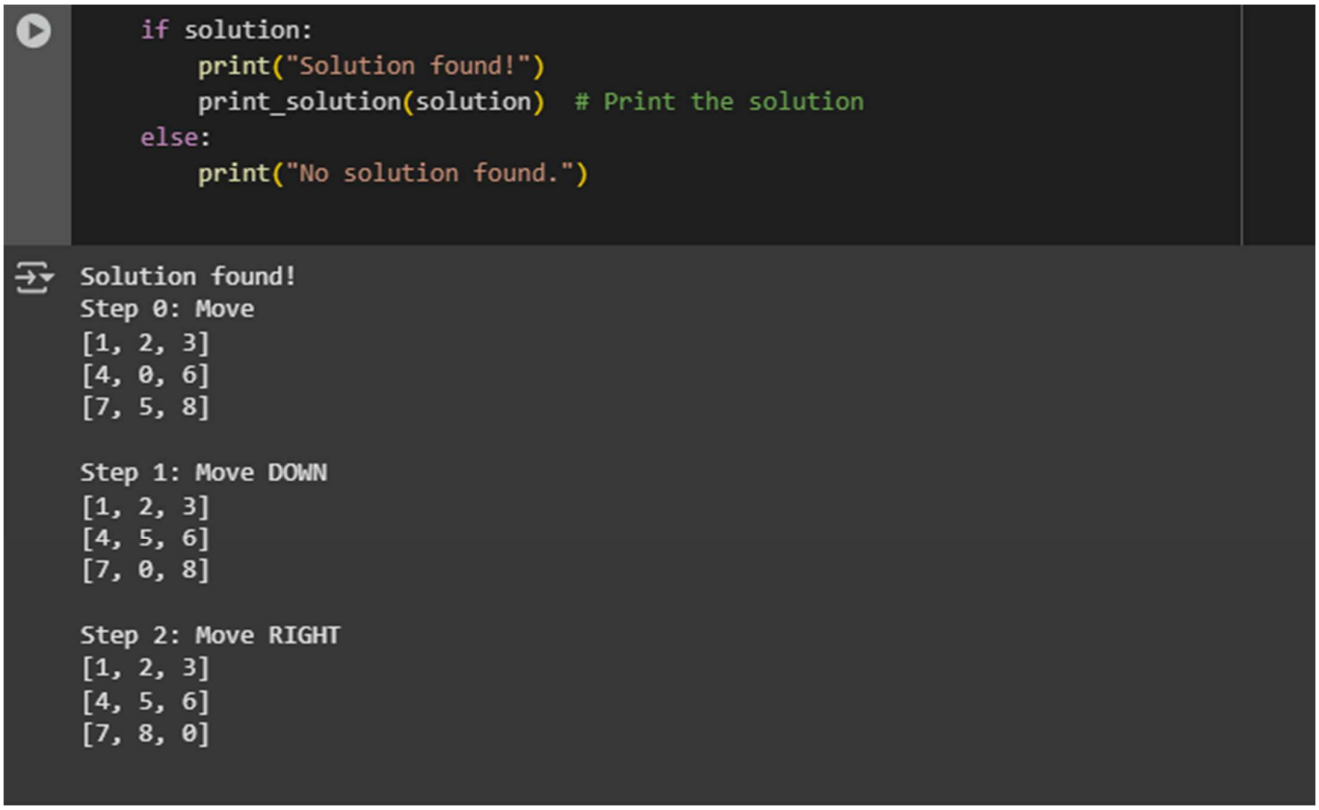
```
        neighbors = []
        x, y = self.find_empty_tile()  # Find the empty tile
        moves = [('UP', x - 1, y), ('DOWN', x + 1, y), ('LEFT', x, y - 1), ('RIGHT', x, y + 1)]
        for move, nx, ny in moves:
            if 0 <= nx < 3 and 0 <= ny < 3:  # Check if move is valid
                new_board = [row[:] for row in self.board]  # Create a copy of the board
                new_board[x][y], new_board[nx][ny] = new_board[nx][ny], new_board[x][y]  # Swap tiles
                neighbors.append(PuzzleState(new_board, self, move))  # Add new state
        return neighbors

    # Find the position of the empty tile (0)
    def find_empty_tile(self):
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return i, j

    # Less than operator for PriorityQueue
    def __lt__(self, other):
        return (self.g + self.h) < (other.g + other.h)

    # Print the board
```

```
    def print_board(self):
        for row in self.board:
            print(row)
        print()

# A* Algorithm to solve the puzzle
def solve_puzzle(initial_state):
    open_set = PriorityQueue()  # Open set for states to explore
    open_set.put(initial_state)  # Add initial state
    closed_set = set()  # Closed set for explored states

    while not open_set.empty():
        current_state = open_set.get()  # Get the state with the lowest cost

        if current_state.is_goal():  # Check if goal state is reached
            return current_state  # Return the goal state

        closed_set.add(tuple(map(tuple, current_state.board)))  # Add current state to closed set

        for neighbor in current_state.get_neighbors():  # Explore neighbors
            if tuple(map(tuple, neighbor.board)) not in closed_set:  # Check if neighbor is not explored
                neighbor.g = current_state.g + 1  # Update cost
                open_set.put(neighbor)  # Add neighbor to open set
```

```
    return None  # If no solution found

# Function to print the solution path
def print_solution(state):
    path = []
    while state:
        path.append(state)
        state = state.parent
    path.reverse()  # Reverse the path to print from start to goal
    for i, s in enumerate(path):
        print(f"Step {i}: Move {s.move}")
        s.print_board()

# Main function
if __name__ == "__main__":
    # Initial state of the puzzle
    initial_board = [
        [1, 2, 3],
        [4, 0, 6],
        [7, 5, 8]
    ]

    initial_state = PuzzleState(initial_board)  # Create initial state
    solution = solve_puzzle(initial_state)  # Solve the puzzle
```

```
if solution:
    print("Solution found!")
    print_solution(solution)  # Print the solution
else:
    print("No solution found.")
```

5.Screenshot of Output



```
if solution:
    print("Solution found!")
    print_solution(solution)  # Print the solution
else:
    print("No solution found.")
```

```
Solution found!
Step 0: Move
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Step 1: Move DOWN
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Step 2: Move RIGHT
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

6. Conclusion

- Summarize the project.
- Mention the key learnings (e.g., understanding A* algorithm, heuristic search, etc.).
- Discuss any challenges faced and how they were overcome.