KIET GROUP OF INSTITUTION

Name: Prashant Jain

Branch: CSE AIML

Section: B

University Roll No. : 202401100400139

Class Roll No. : 66

# Problem Statement : 8 - Puzzle solver

**8-Puzzle Problem Explanation**

## 1 What is the 8-Puzzle Problem?

The **8-puzzle problem** is a classic **sliding puzzle** that consists of a **3×3 grid** with **8 numbered tiles** and **one empty space (0)**. The tiles can be moved into the empty space to rearrange them.

○ **Goal of the Problem**

The objective is to arrange the tiles in a specific order, starting from a **random initial configuration**, by sliding tiles into the empty space.

The **goal state** is:

1 2 3

4 5 6

7 8 0

where 0 represents the empty space.

---

## 2 How the Puzzle Works

- The player can slide a tile **up, down, left, or right** into the empty space.
- The goal is to **reach the correct arrangement** in the **fewest moves possible**.
- The puzzle is **solvable only if the number of tile inversions is even** (an inversion occurs when a larger number appears before a smaller one in a row-wise reading).

---

## 3 Example of the Problem

○ **Given Input (Initial State)**

1 2 3

4 0 5

6 7 8

Here, 0 represents the empty space.

# Approach to Solve the 8-Puzzle Problem

## 1 Understanding the Problem

The **8-Puzzle** consists of a 3×3 **grid** containing **8 numbered tiles** and **one empty space** (0). The goal is to move tiles by sliding them into the empty space to reach the **goal state**:

```
CopyEdit
1 2 3
4 5 6
7 8 0
```

The task is to find the **shortest sequence of moves** that leads to this goal state.

---

## 2 Choosing the Right Algorithm

To solve this problem efficiently, we use the *A (A-star) Search Algorithm** because:

- ✓ It guarantees finding the **shortest path** (optimal solution).
- ✓ It explores **only relevant moves**, reducing computation time.

---

## *3 Approach - A Search Algorithm**

### ○ Step 1: Define the Initial and Goal State

- The user provides the **initial board configuration**.
- The **goal state** is predefined as:

  ```
  CopyEdit
  1 2 3
  4 5 6
  7 8 0
  ```

---

### ○ Step 2: Define the Cost Function

The A* algorithm uses the cost function:

```
CopyEdit
f(n) = g(n) + h(n)
```

Where:

- `g(n)`: **The cost to reach** the current state (number of moves so far).

- `h(n)`: The **heuristic estimate** of how close we are to the goal.
- **Manhattan Distance Heuristic** is used for `h(n)`:
  - It calculates the sum of the distances of all misplaced tiles from their correct positions.

---

### ○ Step 3: Use a Priority Queue (Min-Heap)

- A **priority queue** is used to store puzzle states, always expanding the **most promising state first** (one with the lowest `f(n)`).

---

### ○ Step 4: Generate Valid Moves

- Find the **position of the empty tile (0)**.
- Move the empty tile **Up, Down, Left, or Right**, if possible.
- Generate **new board states** for each valid move.

---

### ○ Step 5: Track Visited States

- Use a **set** to store already visited board configurations.
- This prevents unnecessary re-exploration and speeds up the search.

---

### ○ Step 6: Backtrack to Print the Solution

- Once the goal state is reached, backtrack through the **parent states** to reconstruct the **shortest path**.
- Print each step of the solution.

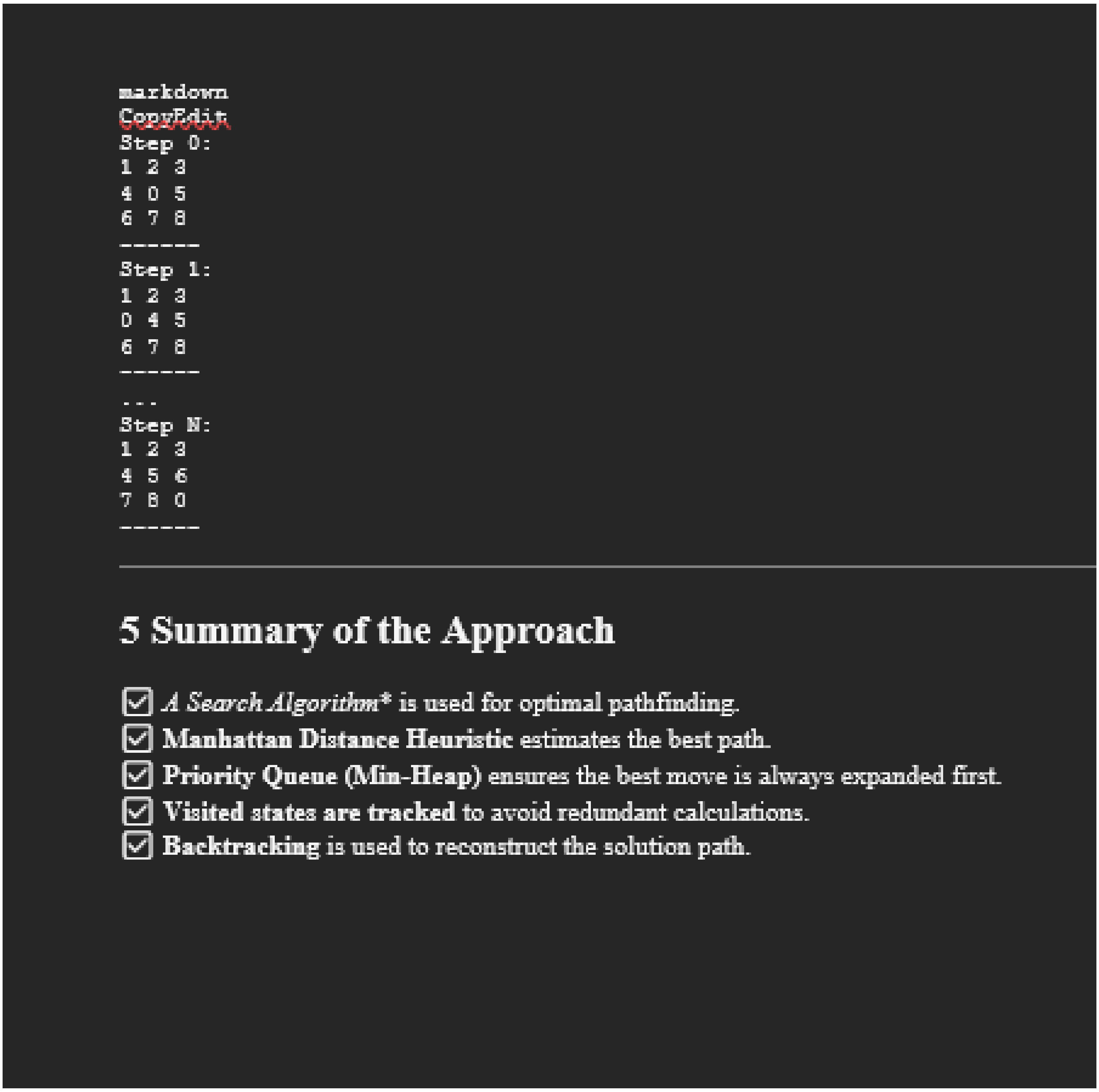
---

## 4️Example Execution

### ○ User Input

```
CopyEdit
1 2 3
4 0 5
6 7 8
```

### ○ Output (Solution Steps)



markdown
CopyEdit

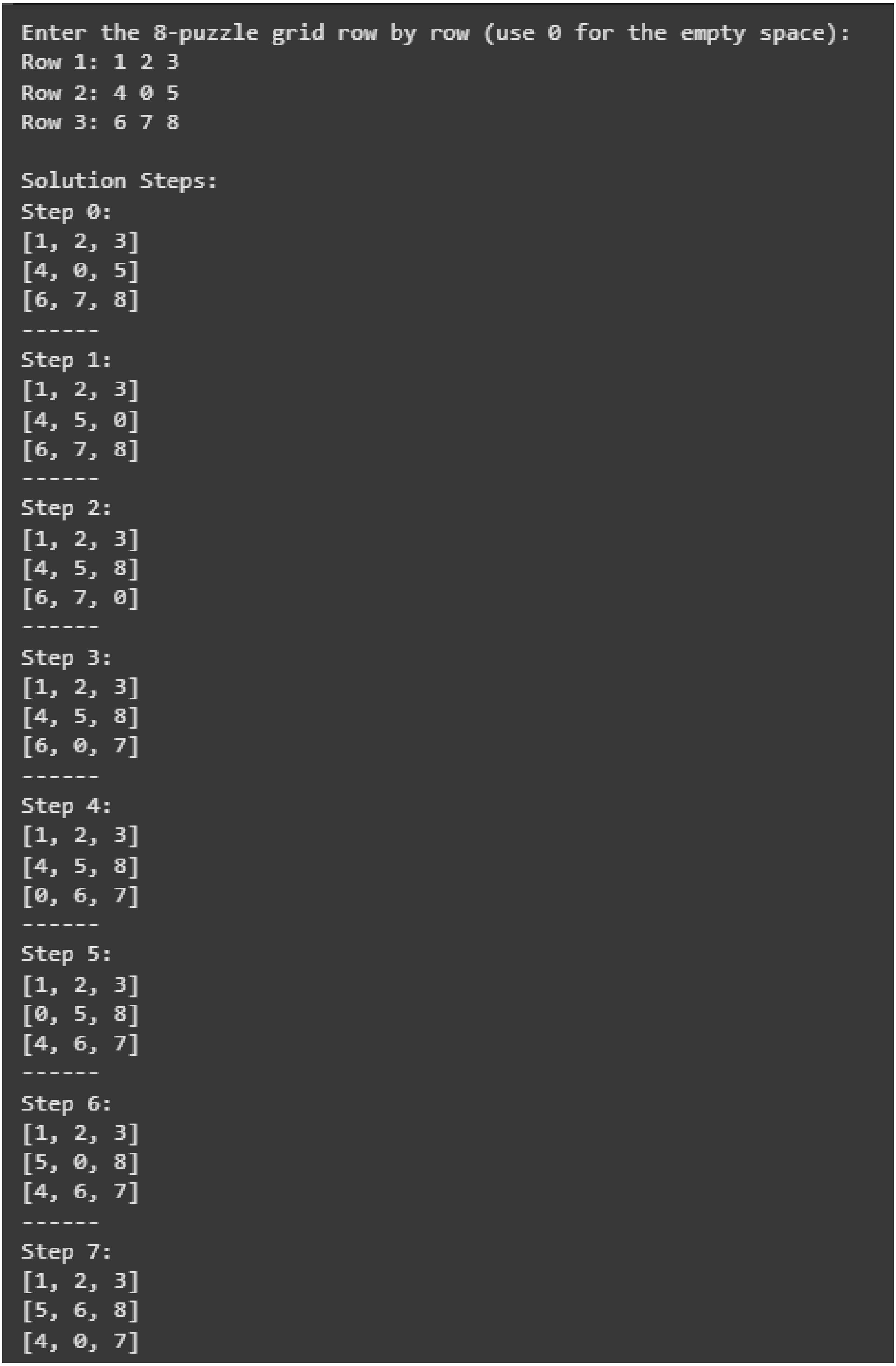
```
Step 0:
1 2 3
4 0 5
6 7 8
------
Step 1:
1 2 3
0 4 5
6 7 8
------
...
Step N:
1 2 3
4 5 6
7 8 0
------
```

---

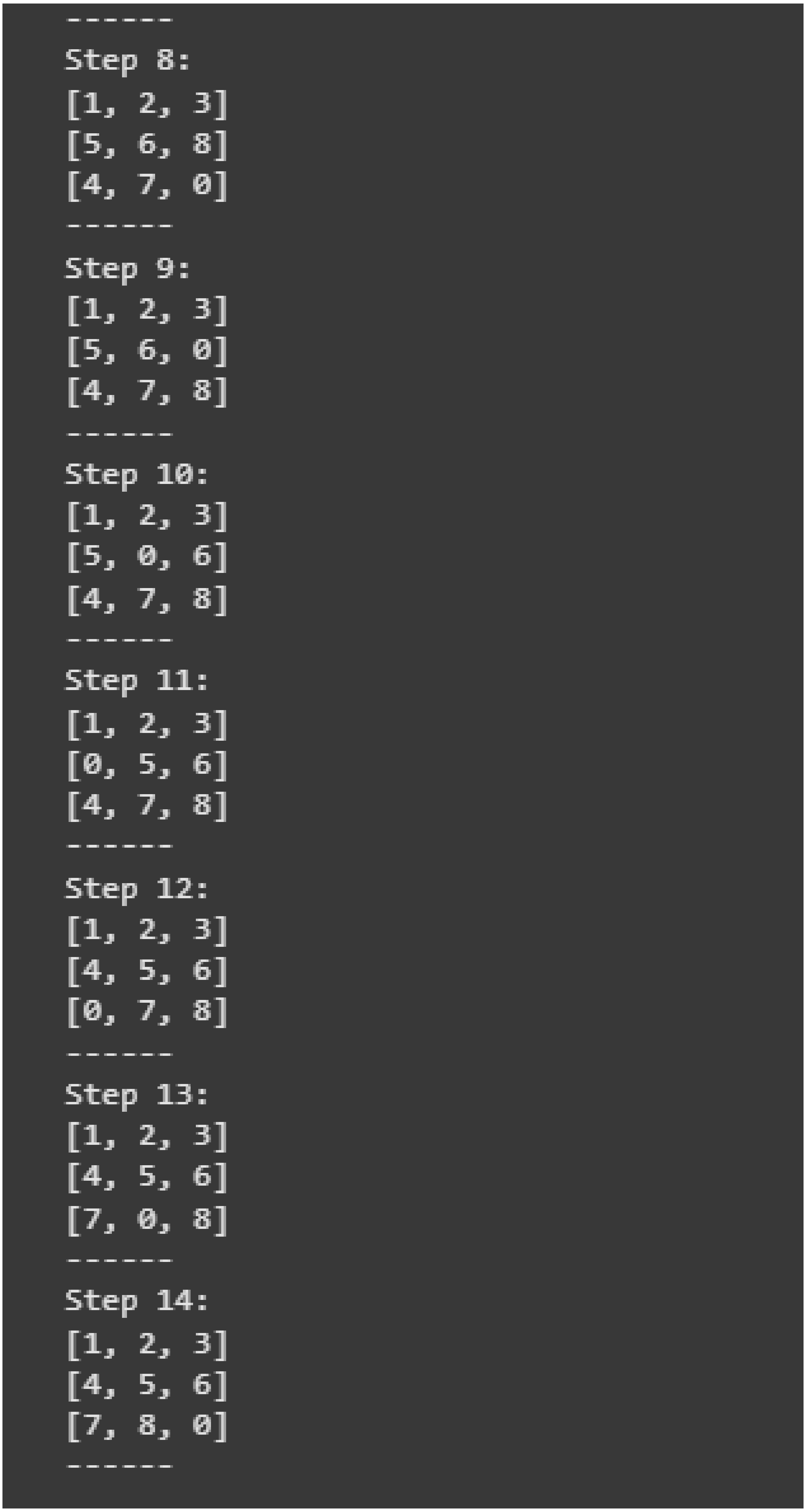
## 5 Summary of the Approach

☑ *A Search Algorithm** is used for optimal pathfinding.
☑ **Manhattan Distance Heuristic** estimates the best path.
☑ **Priority Queue (Min-Heap)** ensures the best move is always expanded first.
☑ **Visited states are tracked** to avoid redundant calculations.
☑ **Backtracking** is used to reconstruct the solution path.

```
from heapq import heappush, heappop

class Puzzle:
    def __init__(self, board, moves=0, previous=None):
        self.board = board
        self.moves = moves
        self.previous = previous
        self.cost = self.moves + self.heuristic()

    def __lt__(self, other):
        return self.cost < other.cost

    def heuristic(self):
        """ Manhattan Distance heuristic """
        distance = 0
        for i in range(3):
            for j in range(3):
                value = self.board[i][j]
                if value != 0:
                    goal_x, goal_y = (value - 1) // 3, (value - 1) % 3
                    distance += abs(i - goal_x) + abs(j - goal_y)
        return distance

    def get_neighbors(self):
        """ Generate possible next moves """
        neighbors = []
        x, y = next((i, j) for i in range(3) for j in range(3) if
self.board[i][j] == 0)
        moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left,
Right

        for dx, dy in moves:
            nx, ny = x + dx, y + dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                new_board = [row[:] for row in self.board]
                new_board[x][y], new_board[nx][ny] = new_board[nx][ny],
new_board[x][y]
                neighbors.append(Puzzle(new_board, self.moves + 1,
self))

        return neighbors

def solve_puzzle(start_board):
    open_set = []
    heappush(open_set, Puzzle(start_board))
    visited = set()

    while open_set:
        current = heappop(open_set)
        if current.heuristic() == 0:
            path = []
            while current:
                path.append(current.board)
                current = current.previous
            return path[::-1]

        state_tuple = tuple(map(tuple, current.board))
        if state_tuple in visited:
            continue
        visited.add(state_tuple)

        for neighbor in current.get_neighbors():
            heappush(open_set, neighbor)

    return None

def get_user_input():
    print("Enter the 8-puzzle grid row by row (use 0 for the empty space):")
    board = []
    for i in range(3):
        row = list(map(int, input(f"Row {i+1}: ").split()))
        board.append(row)
    return board

# Get input from the user
start_state = get_user_input()

# Solve the puzzle
solution = solve_puzzle(start_state)

# Print the solution steps
if solution:
    print("\nSolution Steps:")
    for step, state in enumerate(solution):
        print(f"Step {step}:")
        for row in state:
            print(row)
        print("------")
else:
    print("No solution found.")
```



```
Enter the 8-puzzle grid row by row (use 0 for the empty space):
Row 1: 1 2 3
Row 2: 4 0 5
Row 3: 6 7 8

Solution Steps:
Step 0:
[1, 2, 3]
[4, 0, 5]
[6, 7, 8]
------
Step 1:
[1, 2, 3]
[4, 5, 0]
[6, 7, 8]
------
Step 2:
[1, 2, 3]
[4, 5, 8]
[6, 7, 0]
------
Step 3:
[1, 2, 3]
[4, 5, 8]
[6, 0, 7]
------
Step 4:
[1, 2, 3]
[4, 5, 8]
[0, 6, 7]
------
Step 5:
[1, 2, 3]
[0, 5, 8]
[4, 6, 7]
------
Step 6:
[1, 2, 3]
[5, 0, 8]
[4, 6, 7]
------
Step 7:
[1, 2, 3]
[5, 6, 8]
[4, 0, 7]
```



```
------
Step 8:
[1, 2, 3]
[5, 6, 8]
[4, 7, 0]
------
Step 9:
[1, 2, 3]
[5, 6, 0]
[4, 7, 8]
------
Step 10:
[1, 2, 3]
[5, 0, 6]
[4, 7, 8]
------
Step 11:
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]
------
Step 12:
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
------
Step 13:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
------
Step 14:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
------
```