15-10-24 AI lab
Devanshi Slathia
1BM22CS083

Misplaced tiles

```
class Puzzle:
    def __init__(self, initial_state, goal_state):
        self.board = initial_state
        self.goal = goal_state
        self.n = len(initial_state)

    # To find the index of '0' (blank tile)
    def find_blank(self, board):
        for i in range(self.n):
            for j in range(self.n):
                if board[i][j] == 0:
                    return (i, j)

    # Heuristic function: h(n) - number of misplaced tiles
    def misplaced_tiles(self, board):
        misplaced = 0
        for i in range(self.n):
            for j in range(self.n):
                if board[i][j] != 0 and board[i][j] != self.goal[i][j]:
                    misplaced += 1
        return misplaced

    # Generate possible moves (neighbors) from the current state
    def get_neighbors(self, board):
        neighbors = []
        blank_pos = self.find_blank(board)
        x, y = blank_pos
        # Possible moves (up, down, left, right)
        moves = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]
        for move in moves:
            new_x, new_y = move
            if 0 <= new_x < self.n and 0 <= new_y < self.n:
                new_board = [row[:] for row in board]  # Copy the board
                # Swap the blank with the adjacent tile
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
                neighbors.append(new_board)
        return neighbors
```

```
    # A* Search Algorithm
    def a_star(self):
        start = self.board
        goal = self.goal
        open_list = [(start, 0)]  # List of tuples (board, g(n))
        closed_list = set()
        iteration = 0

        while open_list:
            # Sort open list by f(n) = g(n) + h(n)
            open_list.sort(key=lambda x: x[1] + self.misplaced_tiles(x[0]))  # Sort by f(n)
            current_board, g = open_list.pop(0)  # Get the board with the lowest f(n)

            iteration += 1
            print(f"\nIteration {iteration}:")
            self.print_board(current_board)
            print(f"g(n): {g}, h(n): {self.misplaced_tiles(current_board)}, f(n): {g +
self.misplaced_tiles(current_board)}")

            # If we reach the goal, return the solution
            if current_board == goal:
                print("\nGoal reached!")
                return g

            # Add the current state to the closed list
            closed_list.add(tuple(map(tuple, current_board)))

            # Get all possible moves (neighbors)
            for neighbor in self.get_neighbors(current_board):
                if tuple(map(tuple, neighbor)) in closed_list:
                    continue
                # g(n) is the depth (number of moves from the start)
                g_new = g + 1
                # Add neighbor to the open list
                open_list.append((neighbor, g_new))

        return -1  # If no solution is found

    # Print the 3x3 board
    def print_board(self, board):
        for row in board:
            print(" ".join(str(tile) if tile != 0 else "_" for tile in row))
```

```
# Helper function to take input from the user
def take_input():
    print("Enter the initial state (3x3 grid) row by row, use '0' for the blank tile:")
    initial_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        initial_state.append(row)

    print("Enter the goal state (3x3 grid) row by row, use '0' for the blank tile:")
    goal_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        goal_state.append(row)

    return initial_state, goal_state

# Main
if __name__ == "__main__":
    initial_state, goal_state = take_input()

    puzzle = Puzzle(initial_state, goal_state)
    moves = puzzle.a_star()

    if moves != -1:
        print(f"\nNumber of moves to solve: {moves}")
    else:
        print("\nNo solution found.")
```

Output

```
Enter the initial state (3x3 grid) row by row, use '0' for the blank tile:
 5 4 0
 6 1 8
 7 3 2
Enter the goal state (3x3 grid) row by row, use '0' for the blank tile:
 0 1 2
 3 4 5
 6 7 8

Iteration 1:
5 4 _
6 1 8
7 3 2
g(n): 0, h(n): 12, f(n): 12

Iteration 2:
```

```
5 4 8
6 1 _
7 3 2
g(n): 1, h(n): 13, f(n): 14

Iteration 3:
5 _ 4
6 1 8
7 3 2
g(n): 1, h(n): 13, f(n): 14

Iteration 4:
5 4 8
6 1 2
7 3 _
g(n): 2, h(n): 12, f(n): 14

Iteration 5:
```

```
Iteration 2587:
_ 4 2
5 3 8
6 7 1
g(n): 16, h(n): 8, f(n): 24

Iteration 2588:
_ 1 2
3 4 5
6 7 8
g(n): 24, h(n): 0, f(n): 24

Goal reached!

Number of moves to solve: 24
```

Manhattan Distance

```
class Puzzle:
    def __init__(self, initial_state, goal_state):
        self.board = initial_state
        self.goal = goal_state
        self.n = len(initial_state)

    # To find the index of '0' (blank tile)
    def find_blank(self, board):
        for i in range(self.n):
            for j in range(self.n):
                if board[i][j] == 0:
                    return (i, j)

    # Heuristic function: h(n) - Manhattan distance
    def manhattan_distance(self, board):
        distance = 0
        for i in range(self.n):
            for j in range(self.n):
                if board[i][j] != 0:  # Don't calculate for the blank tile (0)
                    goal_x, goal_y = self.find_position(self.goal, board[i][j])
                    distance += abs(i - goal_x) + abs(j - goal_y)
        return distance

    # Find the position of a tile in the goal state
    def find_position(self, board, value):
        for i in range(self.n):
            for j in range(self.n):
```

```
                if board[i][j] == value:
                    return (i, j)

    # Generate possible moves (neighbors) from the current state
    def get_neighbors(self, board):
        neighbors = []
        blank_pos = self.find_blank(board)
        x, y = blank_pos
        # Possible moves (up, down, left, right)
        moves = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]
        for move in moves:
            new_x, new_y = move
            if 0 <= new_x < self.n and 0 <= new_y < self.n:
                new_board = [row[:] for row in board]  # Copy the board
                # Swap the blank with the adjacent tile
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
                neighbors.append(new_board)
        return neighbors

    # A* Search Algorithm
    def a_star(self):
        start = self.board
        goal = self.goal
        open_list = [(start, 0)]  # List of tuples (board, g(n))
        closed_list = set()
        iteration = 0

        while open_list:
            # Sort open list by f(n) = g(n) + h(n)
            open_list.sort(key=lambda x: x[1] + self.manhattan_distance(x[0]))  # Sort by f(n)
            current_board, g = open_list.pop(0)  # Get the board with the lowest f(n)

            iteration += 1
            print(f"\nIteration {iteration}:")
            self.print_board(current_board)
            print(f"g(n): {g}, h(n): {self.manhattan_distance(current_board)}, f(n): {g +
self.manhattan_distance(current_board)}")

            # If we reach the goal, return the solution
            if current_board == goal:
                print("\nGoal reached!")
                return g
```

```
            # Add the current state to the closed list
            closed_list.add(tuple(map(tuple, current_board)))

            # Get all possible moves (neighbors)
            for neighbor in self.get_neighbors(current_board):
                if tuple(map(tuple, neighbor)) in closed_list:
                    continue
                # g(n) is the depth (number of moves from the start)
                g_new = g + 1
                # Add neighbor to the open list
                open_list.append((neighbor, g_new))

        return -1  # If no solution is found

    # Print the 3x3 board
    def print_board(self, board):
        for row in board:
            print(" ".join(str(tile) if tile != 0 else "_" for tile in row))

# Helper function to take input from the user
def take_input():
    print("Enter the initial state (3x3 grid) row by row, use '0' for the blank tile:")
    initial_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        initial_state.append(row)

    print("Enter the goal state (3x3 grid) row by row, use '0' for the blank tile:")
    goal_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        goal_state.append(row)

    return initial_state, goal_state

# Main
if __name__ == "__main__":
    initial_state, goal_state = take_input()

    puzzle = Puzzle(initial_state, goal_state)
    moves = puzzle.a_star()

    if moves != -1:
        print(f"\nNumber of moves to solve: {moves}")
```

```
    else:
        print("\nNo solution found")
```

Output

```
Enter the initial state (3x3 grid) row by row, use '0' for the blank tile:
 5 4 0
 6 1 8
 7 3 2
Enter the goal state (3x3 grid) row by row, use '0' for the blank tile:
 0 1 2
 3 4 5
 6 7 8

Iteration 1:
5 4 _
6 1 8
7 3 2
g(n): 0, h(n): 12, f(n): 12

Iteration 2:
5 4 8
6 1 _
7 3 2
g(n): 1, h(n): 13, f(n): 14

Iteration 3:
5 _ 4
6 1 8
7 3 2
g(n): 1, h(n): 13, f(n): 14

Iteration 4:
5 4 8
6 1 2
7 3 _
g(n): 2, h(n): 12, f(n): 14

Iteration 5:
5 1 4
6 _ 8
7 3 2
g(n): 2, h(n): 12, f(n): 14

Iteration 6:
_ 5 4
6 1 8
7 3 2
g(n): 2, h(n): 12, f(n): 14
```

```
6 3 8
g(n): 20, h(n): 4, f(n): 24

Iteration 2587:
_ 4 2
5 3 8
6 7 1
g(n): 16, h(n): 8, f(n): 24

Iteration 2588:
_ 1 2
3 4 5
6 7 8
g(n): 24, h(n): 0, f(n): 24

Goal reached!

Number of moves to solve: 24
```