Sagar Bangari(1BM22CS231)

Lab – 3 A* Implementation of 8 – Puzzle Problem

1) Misplaced tiles :-

Code:

```
import heapq

class Node:

  def __init__(self, state, parent=None):

    self.state = state

    self.parent = parent

    self.g = 0  # Cost from start to current node

    self.h = 0  # Heuristic cost to goal

    self.f = 0  # Total cost


  def __lt__(self, other):

    return self.f < other.f


def calculate_misplaced_tiles(state, goal):

  return sum(1 for i in range(9) if state[i] != goal[i] and state[i] != 0)


def get_possible_moves(state):

  moves = []

  zero_index = state.index(0)

  row, col = divmod(zero_index, 3)


  directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
```

```
    direction_names = ["Up", "Down", "Left", "Right"]

    for (dr, dc), dir_name in zip(directions, direction_names):
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_index = new_row * 3 + new_col
            new_state = list(state)
            new_state[zero_index], new_state[new_index] = new_state[new_index],
new_state[zero_index]
            moves.append((tuple(new_state), dir_name))  # Store new state with direction
    return moves

def print_state(state):
    """Print the 2D matrix representation of the state."""
    print("Current state:")
    for i in range(3):
        print(f"| {' | '.join(str(x) for x in state[i*3:(i+1)*3])} |")
    print()  # Blank line for readability

def a_star(start, goal):
    start_node = Node(start)
    start_node.h = calculate_misplaced_tiles(start, goal)
    start_node.f = start_node.g + start_node.h

    open_list = []
    closed_set = set()
```

```
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)

        if current_node.state == goal:
            path = []
            while current_node:
                path.append(current_node.state)
                current_node = current_node.parent
            return path[::-1]  # Return reversed path

        closed_set.add(tuple(current_node.state))

        for move, direction in get_possible_moves(current_node.state):
            if tuple(move) in closed_set:
                continue

            child_node = Node(move, current_node)
            child_node.g = current_node.g + 1
            child_node.h = calculate_misplaced_tiles(move, goal)
            child_node.f = child_node.g + child_node.h

            if not any(open_node.state == move and open_node.g <= child_node.g for open_node
in open_list):
                heapq.heappush(open_list, child_node)
```

```
    return None  # No solution found


def get_user_input(prompt):
    state = []
    for i in range(3):
        while True:
            try:
                row = input(f"{prompt} (row {i + 1}): ")
                row_values = list(map(int, row.split()))
                if len(row_values) == 3 and all(0 <= x <= 8 for x in row_values):
                    state.extend(row_values)
                    break
                else:
                    print("Invalid input. Please enter 3 integers (0-8) for this row.")
            except ValueError:
                print("Invalid input. Please enter integers only.")
    return tuple(state)


# Main execution
print("Using number of misplaced tiles.")
start_state = get_user_input("Enter the start state")
goal_state = get_user_input("Enter the goal state")


solution_path = a_star(start_state, goal_state)
```

```
if solution_path:
    total_cost = len(solution_path) - 1  # Total cost is the number of moves
    moves = []  # Store moves for final output
    for i in range(1, len(solution_path)):
        move_direction = ""
        for move, direction in get_possible_moves(solution_path[i - 1]):
            if move == solution_path[i]:
                move_direction = direction
                break
        moves.append(move_direction)
        current_h = calculate_misplaced_tiles(solution_path[i], goal_state)  # Calculate
heuristic for current state
        print(f"Moved {move_direction} | Heuristic value: {current_h}")
        print_state(solution_path[i])  # Print the 2D matrix representation of the state

    print(f"Total cost: {total_cost}")
    print("Goal reached!")
    print("Moves taken:", " -> ".join(moves))  # Print the entire path of moves
else:
    print("No solution found.")
```

Output:

```
Using number of misplaced tiles.
Enter the start state (row 1): 1 2 3
Enter the start state (row 2): 4 5 0
Enter the start state (row 3): 7 8 6
Enter the goal state (row 1): 1 2 3
Enter the goal state (row 2): 4 5 6
Enter the goal state (row 3): 0 7 8
Moved Down | Heuristic value: 2
Current state:
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 0 |

Moved Left | Heuristic value: 1
Current state:
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 0 | 8 |

Moved Left | Heuristic value: 0
Current state:
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 0 | 7 | 8 |

Total cost: 3
Goal reached!
Moves taken: Down -> Left -> Left
```

2) Manhattan distance:-

Code:

```
import heapq

class Node:
    def __init__(self, state, parent=None):
        self.state = state
        self.parent = parent
        self.g = 0  # Cost from start to current node
        self.h = 0  # Heuristic cost to goal (Manhattan distance)
        self.f = 0  # Total cost

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

def calculate_manhattan_distance(state, goal):
    distance = 0
    for i in range(9):
        if state[i] != 0:  # Ignore the blank tile
            goal_index = goal.index(state[i])
            current_row, current_col = divmod(i, 3)
            goal_row, goal_col = divmod(goal_index, 3)
            distance += abs(current_row - goal_row) + abs(current_col - goal_col)
    return distance
```

```
def get_possible_moves(state):
    moves = []
    zero_index = state.index(0)
    row, col = divmod(zero_index, 3)

    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
    direction_names = ["Up", "Down", "Left", "Right"]

    for (dr, dc), dir_name in zip(directions, direction_names):
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_index = new_row * 3 + new_col
            new_state = list(state)
            new_state[zero_index], new_state[new_index] = new_state[new_index],
new_state[zero_index]
            moves.append((tuple(new_state), dir_name))  # Store new state with direction
    return moves

def print_state(state):
    """Print the 2D matrix representation of the state."""
    print("Current state:")
    for i in range(3):
        print(f"| {' | '.join(str(x) for x in state[i*3:(i+1)*3])} |")
    print()  # Blank line for readability
```

```
def a_star(start, goal):
    start_node = Node(start)
    start_node.h = calculate_manhattan_distance(start, goal)
    start_node.f = start_node.g + start_node.h

    open_list = []
    closed_set = set()
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)

        if current_node.state == goal:
            path = []
            while current_node:
                path.append(current_node.state)
                current_node = current_node.parent
            return path[::-1]  # Return reversed path

        closed_set.add(tuple(current_node.state))

        for move, direction in get_possible_moves(current_node.state):
            if tuple(move) in closed_set:
                continue

            child_node = Node(move, current_node)
```

```
        child_node.g = current_node.g + 1
        child_node.h = calculate_manhattan_distance(move, goal)
        child_node.f = child_node.g + child_node.h

        if not any(open_node.state == move and open_node.g <= child_node.g for open_node
in open_list):
          heapq.heappush(open_list, child_node)

  return None  # No solution found


def get_user_input(prompt):
  state = []
  for i in range(3):
    while True:
      try:
        row = input(f"{prompt} (row {i + 1}): ")
        row_values = list(map(int, row.split()))
        if len(row_values) == 3 and all(0 <= x <= 8 for x in row_values):
          state.extend(row_values)
          break
        else:
          print("Invalid input. Please enter 3 integers (0-8) for this row.")
      except ValueError:
        print("Invalid input. Please enter integers only.")
  return tuple(state)
```

```
# Main execution
print("Using Manhattan distance.")
start_state = get_user_input("Enter the start state")
goal_state = get_user_input("Enter the goal state")

solution_path = a_star(start_state, goal_state)

if solution_path:
    total_cost = len(solution_path) - 1  # Total cost is the number of moves
    moves = []  # Store moves for final output
    for i in range(1, len(solution_path)):
        move_direction = ""
        for move, direction in get_possible_moves(solution_path[i - 1]):
            if move == solution_path[i]:
                move_direction = direction
                break
        moves.append(move_direction)
        current_h = calculate_manhattan_distance(solution_path[i], goal_state)  # Calculate
heuristic for current state
        print(f"Moved {move_direction} | Heuristic value: {current_h}")
        print_state(solution_path[i])  # Print the 2D matrix representation of the state

    print(f"Total cost: {total_cost}")
    print("Goal reached!")
    print("Moves taken:", " -> ".join(moves))  # Print the entire path of moves
else:
```

```
    print("No solution found.")
```

Output:

```
Using number of misplaced tiles.
Enter the start state (row 1): 1 2 3
Enter the start state (row 2): 4 5 0
Enter the start state (row 3): 7 8 6
Enter the goal state (row 1): 1 2 3
Enter the goal state (row 2): 4 5 6
Enter the goal state (row 3): 0 7 8
Moved Down | Heuristic value: 2
Current state:
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 0 |

Moved Left | Heuristic value: 1
Current state:
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 0 | 8 |

Moved Left | Heuristic value: 0
Current state:
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 0 | 7 | 8 |

Total cost: 3
Goal reached!
Moves taken: Down -> Left -> Left
```