4. Solve 8 puzzle problem using A* algorithm

1.Using Misplaced tiles

```
import heapq
print("Varsha P(1BM22CS320)")

# Define the goal state for the 8-puzzle
GOAL_STATE = [
    [2, 8, 1],
    [0, 4, 3],
    [7, 6, 5]
]

# Define the position moves (up, down, left, right)
MOVES = [
    (-1, 0),  # Up
    (1, 0),   # Down
    (0, -1),  # Left
    (0, 1)    # Right
]

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

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

def misplaced_tiles(state):
    """Heuristic function that counts the number of misplaced tiles."""
    misplaced = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != GOAL_STATE[i][j]:
                misplaced += 1
    return misplaced

def get_zero_position(state):
    """Find the position of the zero (empty tile) in the puzzle."""
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
```

```
    return None

def generate_successors(node):
    """Generate successors by moving the empty tile in all possible directions."""
    successors = []
    zero_x, zero_y = get_zero_position(node.state)

    for move_x, move_y in MOVES:
        new_x, new_y = zero_x + move_x, zero_y + move_y
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in node.state]
            new_state[zero_x][zero_y], new_state[new_x][new_y] =
new_state[new_x][new_y], new_state[zero_x][zero_y]
            h = misplaced_tiles(new_state)
            successors.append(PuzzleNode(new_state, parent=node, g=node.g + 1,
h=h))
    return successors

def is_goal(state):
    """Check if the current state is the goal state."""
    return state == GOAL_STATE

def reconstruct_path(node):
    """Reconstruct the path from the start state to the goal state."""
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return path[::-1]

def a_star(start_state):
    """A* algorithm to solve the 8-puzzle problem."""
    start_node = PuzzleNode(start_state, g=0, h=misplaced_tiles(start_state))
    open_list = []
    closed_set = set()

    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)

        if is_goal(current_node.state):
            return reconstruct_path(current_node)

        closed_set.add(tuple(map(tuple, current_node.state)))

        for successor in generate_successors(current_node):
```

```python
            if tuple(map(tuple, successor.state)) in closed_set:
                continue
            heapq.heappush(open_list, successor)

    return None

def get_user_input():
    """Get a valid 8-puzzle input state from the user."""
    print("Enter your 8-puzzle configuration (0 represents the empty tile):")
    state = []
    values = set()

    for i in range(3):
        row = input(f"Enter row {i+1} (space-separated numbers between 0 and 8): ").split()
        if len(row) != 3:
            print("Each row must have exactly 3 numbers. Please try again.")
            return None

        row = [int(x) for x in row]

        if not all(0 <= x <= 8 for x in row):
            print("Values must be between 0 and 8. Please try again.")
            return None

        state.append(row)
        values.update(row)

    if values != set(range(9)):
        print("All numbers from 0 to 8 must be present exactly once. Please try again.")
        return None

    return state

# Main function
def main():
    start_state = None
    while start_state is None:
        start_state = get_user_input()

    solution = a_star(start_state)

    # Print the solution steps
    if solution:
        print("Solution found in", len(solution) - 1, "moves:")
        for step in solution:
```

```python
            for row in step:
                print(row)
            print()
    else:
        print("No solution found.")

if __name__ == "__main__":
    main()
```

Output

```
Varsha P(1BM22CS320)
Enter your 8-puzzle configuration (0 represents the empty tile):
Enter row 1 (space-separated numbers between 0 and 8): 1 2 3
Enter row 2 (space-separated numbers between 0 and 8): 4 0 6
Enter row 3 (space-separated numbers between 0 and 8): 7 8 5
Solution found in 17 moves:
[1, 2, 3]
[4, 0, 6]
[7, 8, 5]

[1, 2, 3]
[4, 8, 6]
[7, 0, 5]

[1, 2, 3]
[4, 8, 6]
[0, 7, 5]

[1, 2, 3]
[0, 8, 6]
[4, 7, 5]

[1, 2, 3]
[8, 0, 6]
[4, 7, 5]

[1, 0, 3]
[8, 2, 6]
[4, 7, 5]

[0, 1, 3]
[8, 2, 6]
[4, 7, 5]

[8, 1, 3]
[0, 2, 6]
[4, 7, 5]

[8, 1, 3]
[2, 0, 6]
[4, 7, 5]

[8, 1, 3]
[2, 6, 0]
[4, 7, 5]

[8, 1, 0]
[2, 6, 3]
[4, 7, 5]

[8, 0, 1]
[2, 6, 3]
[4, 7, 5]
```

```
[0, 8, 1]
[2, 6, 3]
[4, 7, 5]

[2, 8, 1]
[0, 6, 3]
[4, 7, 5]

[2, 8, 1]
[4, 6, 3]
[0, 7, 5]

[2, 8, 1]
[4, 6, 3]
[7, 0, 5]

[2, 8, 1]
[4, 0, 3]
[7, 6, 5]

[2, 8, 1]
[0, 4, 3]
[7, 6, 5]
```

2.Using Manhattan distance

```
import heapq
print("Varsha P(1BM22CS320)")

# Define the goal state for the 8-puzzle
GOAL_STATE = [
     [2, 8, 1],
     [0, 4, 3],
     [7, 6, 5]

]

# Define the position moves (up, down, left, right)
MOVES = [
    (-1, 0),  # Up
    (1, 0),   # Down
    (0, -1),  # Left
    (0, 1)    # Right
]

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

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

def manhattan_distance(state):
    """Heuristic function that calculates the Manhattan distance for each tile."""
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:  # Skip the empty tile
                # Calculate goal position for this value
                goal_x, goal_y = (value - 1) // 3, (value - 1) % 3
                # Add the Manhattan distance for this tile
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance

def get_zero_position(state):
    """Find the position of the zero (empty tile) in the puzzle."""
    for i in range(3):
```

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

def generate_successors(node):
    """Generate successors by moving the empty tile in all possible directions."""
    successors = []
    zero_x, zero_y = get_zero_position(node.state)

    for move_x, move_y in MOVES:
        new_x, new_y = zero_x + move_x, zero_y + move_y
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in node.state]
            new_state[zero_x][zero_y], new_state[new_x][new_y] =
new_state[new_x][new_y], new_state[zero_x][zero_y]
            h = manhattan_distance(new_state)
            successors.append(PuzzleNode(new_state, parent=node, g=node.g + 1,
h=h))
    return successors

def is_goal(state):
    """Check if the current state is the goal state."""
    return state == GOAL_STATE

def reconstruct_path(node):
    """Reconstruct the path from the start state to the goal state."""
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return path[::-1]

def a_star(start_state):
    """A* algorithm to solve the 8-puzzle problem."""
    start_node = PuzzleNode(start_state, g=0, h=manhattan_distance(start_state))
    open_list = []
    closed_set = set()

    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)

        if is_goal(current_node.state):
            return reconstruct_path(current_node)
```

```python
        closed_set.add(tuple(map(tuple, current_node.state)))

        for successor in generate_successors(current_node):
            if tuple(map(tuple, successor.state)) in closed_set:
                continue
            heapq.heappush(open_list, successor)

    return None

def get_user_input():
    """Get a valid 8-puzzle input state from the user."""
    print("Enter your 8-puzzle configuration (0 represents the empty tile):")
    state = []
    values = set()

    for i in range(3):
        row = input(f"Enter row {i+1} (space-separated numbers between 0 and 8): 
").split()
        if len(row) != 3:
            print("Each row must have exactly 3 numbers. Please try again.")
            return None

        row = [int(x) for x in row]

        if not all(0 <= x <= 8 for x in row):
            print("Values must be between 0 and 8. Please try again.")
            return None

        state.append(row)
        values.update(row)

    if values != set(range(9)):
        print("All numbers from 0 to 8 must be present exactly once. Please try 
again.")
        return None

    return state

# Main function
def main():
    start_state = None
    while start_state is None:
        start_state = get_user_input()

    solution = a_star(start_state)

    # Print the solution steps
```

```python
    if solution:
        print("Solution found in", len(solution) - 1, "moves:")
        for step in solution:
            for row in step:
                print(row)
            print()
    else:
        print("No solution found.")

if __name__ == "__main__":
    main()
```

Output

```
Varsha P(1BM22CS320)
Enter your 8-puzzle configuration (0 represents the empty tile):
Enter row 1 (space-separated numbers between 0 and 8): 1 2 3
Enter row 2 (space-separated numbers between 0 and 8): 4 0 6
Enter row 3 (space-separated numbers between 0 and 8): 7 8 5
Solution found in 17 moves:
[1, 2, 3]
[4, 0, 6]
[7, 8, 5]

[1, 2, 3]
[4, 8, 6]
[7, 0, 5]

[1, 2, 3]
[4, 8, 6]
[0, 7, 5]

[1, 2, 3]
[0, 8, 6]
[4, 7, 5]

[1, 2, 3]
[8, 0, 6]
[4, 7, 5]

[1, 0, 3]
[8, 2, 6]
[4, 7, 5]
```

```
[0, 1, 3]
[8, 2, 6]
[4, 7, 5]

[8, 1, 3]
[0, 2, 6]
[4, 7, 5]

[8, 1, 3]
[2, 0, 6]
[4, 7, 5]

[8, 1, 3]
[2, 6, 0]
[4, 7, 5]

[8, 1, 0]
[2, 6, 3]
[4, 7, 5]

[8, 0, 1]
[2, 6, 3]
[4, 7, 5]
```

```
[0, 8, 1]
[2, 6, 3]
[4, 7, 5]

[2, 8, 1]
[0, 6, 3]
[4, 7, 5]

[2, 8, 1]
[4, 6, 3]
[0, 7, 5]

[2, 8, 1]
[4, 6, 3]
[7, 0, 5]

[2, 8, 1]
[4, 0, 3]
[7, 6, 5]

[2, 8, 1]
[0, 4, 3]
[7, 6, 5]
```

Observation

LAB4 08/10/24

A* Algorithm

function A* search (problem) returns a Solution or failure

node ← a node n with n.state = problem.initialState
n.g = 0

frontier ← a priority queue ordered by ascending g+h only element n

loop do
  if empty (frontier) then return failure
  n ← pop (frontier)
  if problem.goalTest (n.state) then return Solution(n)
  for Each action a in problem.action (n.state) do
    n' ← childNode (problem.n, a)
    insert (n', g(n') + h(n'), frontier)

~~output~~
~~Solved in 3 move~~

Input : [[1, 2, 0], [5, 3, 4], [6, 7, 8]] initial state

goal state : [[1, 3, 2], [5, 4, 0], [6, 7, 8]]

Output

→ Solved in 3 moves using Misplaced Tiles

→ Solved in 3 moves using Manhattan Distance

State space tree

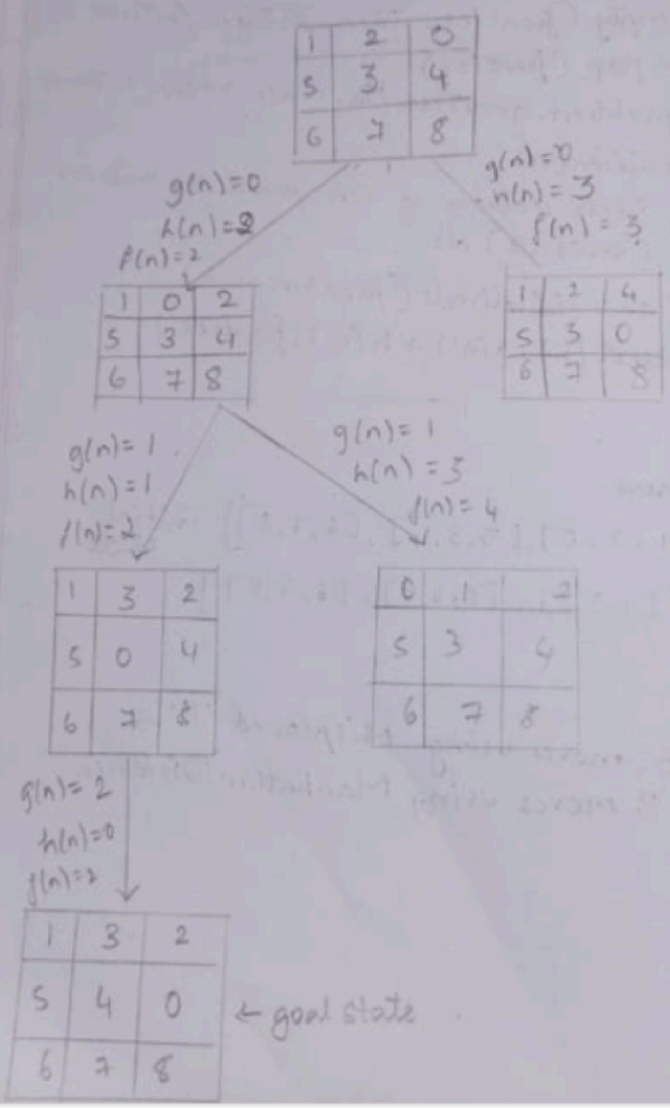
① - A* - Misplace Tiles

Initial state

1	2	0
5	3	4
6	7	8

goal state

1	3	2
5	4	0
6	7	8



② A* - Manhattan distance