**LAB PROGRAM 4**

Solve 8 puzzles using A\* algorithm:

1. Number of misplaced tiles:

PSEUDOCODE:

CODE:

from heapq import heappush, heappop

class PuzzleState:

    def \_\_init\_\_(self, board, parent=None, move=None, depth=0):

        self.board = board

        self.parent = parent

        self.move = move

        self.depth = depth

        self.zero\_pos = self.board.index(0)

    def is\_goal(self, goal):

        return self.board == goal

    def get\_moves(self):

        moves = []

        zero = self.zero\_pos

        row, col = zero // 3, zero % 3

        directions = {

            'Up': (row - 1, col),

            'Down': (row + 1, col),

            'Left': (row, col - 1),

            'Right': (row, col + 1)

        }

        for move, (r, c) in directions.items():

            if 0 <= r < 3 and 0 <= c < 3:

                new\_zero = r \* 3 + c

                new\_board = list(self.board)

                new\_board[zero], new\_board[new\_zero] = new\_board[new\_zero], new\_board[zero]

                moves.append(PuzzleState(tuple(new\_board), self, move, self.depth + 1))

        return moves

    def misplaced\_tiles(self, goal):

        return sum(1 for i, tile in enumerate(self.board) if tile != 0 and tile != goal[i])

    def \_\_lt\_\_(self, other):

        return True

def a\_star(start, goal):

    open\_list = []

    closed\_set = set()

    start\_state = PuzzleState(start)

    heappush(open\_list, (start\_state.misplaced\_tiles(goal), start\_state))

    while open\_list:

        \_, current = heappop(open\_list)

        if current.is\_goal(goal):

            return reconstruct\_path(current)

        closed\_set.add(current.board)

        for neighbor in current.get\_moves():

            if neighbor.board in closed\_set:

                continue

            cost = neighbor.depth + neighbor.misplaced\_tiles(goal)

            heappush(open\_list, (cost, neighbor))

    return None

def reconstruct\_path(state):

    path = []

    while state.parent is not None:

        path.append(state.move)

        state = state.parent

    path.reverse()

    return path

if \_\_name\_\_ == "\_\_main\_\_":

    start = (2, 8, 3,

             1, 6, 4,

             7, 0, 5)

    goal = (1, 2, 3,

            8, 0, 4,

            7, 6, 5)

    print("Sinchana Hemanth (1BM23CS330)")

    solution = a\_star(start, goal)

    if solution:

        print(f"Solution found in {len(solution)} moves: {solution}")

    else:

        print("No solution found.")

OUTPUT:



1. Manhattan distance:

PSEUDOCODE:

CODE:

from heapq import heappush, heappop

class PuzzleState:

    def \_\_init\_\_(self, board, parent=None, move=None, depth=0):

        self.board = board

        self.parent = parent

        self.move = move

        self.depth = depth

        self.zero\_pos = self.board.index(0)

    def is\_goal(self, goal):

        return self.board == goal

    def get\_moves(self):

        moves = []

        zero = self.zero\_pos

        row, col = zero // 3, zero % 3

        directions = {

            'Up': (row - 1, col),

            'Down': (row + 1, col),

            'Left': (row, col - 1),

            'Right': (row, col + 1)

        }

        for move, (r, c) in directions.items():

            if 0 <= r < 3 and 0 <= c < 3:

                new\_zero = r \* 3 + c

                new\_board = list(self.board)

                new\_board[zero], new\_board[new\_zero] = new\_board[new\_zero], new\_board[zero]

                moves.append(PuzzleState(tuple(new\_board), self, move, self.depth + 1))

        return moves

    def manhattan\_distance(self, goal):

        distance = 0

        for i, tile in enumerate(self.board):

            if tile != 0:

                goal\_index = goal.index(tile)

                current\_row, current\_col = i // 3, i % 3

                goal\_row, goal\_col = goal\_index // 3, goal\_index % 3

                distance += abs(current\_row - goal\_row) + abs(current\_col - goal\_col)

        return distance

    def \_\_lt\_\_(self, other):

        return True

def a\_star(start, goal):

    open\_list = []

    closed\_set = set()

    start\_state = PuzzleState(start)

    heappush(open\_list, (start\_state.manhattan\_distance(goal), start\_state))

    while open\_list:

        \_, current = heappop(open\_list)

        if current.is\_goal(goal):

            return reconstruct\_path(current)

        closed\_set.add(current.board)

        for neighbor in current.get\_moves():

            if neighbor.board in closed\_set:

                continue

            cost = neighbor.depth + neighbor.manhattan\_distance(goal)

            heappush(open\_list, (cost, neighbor))

    return None

def reconstruct\_path(state):

    path = []

    while state.parent is not None:

        path.append(state.move)

        state = state.parent

    path.reverse()

    return path

if \_\_name\_\_ == "\_\_main\_\_":

    start = (2, 8, 3,

             1, 6, 4,

             7, 0, 5)

    goal = (1, 2, 3,

            8, 0, 4,

            7, 6, 5)

    print("Sinchana Hemanth (1BM23CS330)")

    solution = a\_star(start, goal)

    if solution:

        print(f"Solution found in {len(solution)} moves: {solution}")

    else:

        print("No solution found.")

OUTPUT:

