1. 8-puzzle A* implementation using Number of Misplaced tiles

```
import heapq

class Node:
    def __init__(self, name, parent=None, g=0, h=0):
        self.name = name          # The name of the node
        self.parent = parent      # Parent node
        self.g = g                # Cost from start to this node
        self.h = h                # Heuristic cost to goal
        self.f = g + h            # Total cost

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

def heuristic(node, goal):
    # Example heuristic (Manhattan distance for grid)
    return abs(node[0] - goal[0]) + abs(node[1] - goal[1])

def a_star_search(start, goal, neighbors):
    open_list = []
    closed_set = set()

    start_node = Node(start, None, 0, heuristic(start, goal))
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)

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

        closed_set.add(current_node.name)

        for neighbor in neighbors(current_node.name):
            if neighbor in closed_set:
                continue

            g_cost = current_node.g + 1  # Assuming uniform cost for neighbors
            h_cost = heuristic(neighbor, goal)
            neighbor_node = Node(neighbor, current_node, g_cost, h_cost)
```

```
            if any(open_node.name == neighbor and open_node.g <= g_cost for open_node in
open_list):
                continue

            heapq.heappush(open_list, neighbor_node)

    return None  # No path found

# Example usage:
def get_neighbors(node):
    # Define neighbors for a grid or graph
    neighbors_map = {
        (0, 0): [(0, 1), (1, 0)],
        (0, 1): [(0, 0), (0, 2), (1, 1)],
        (1, 0): [(0, 0), (1, 1), (2, 0)],
        (1, 1): [(0, 1), (1, 0), (1, 2)],
        (0, 2): [(0, 1)],
        (2, 0): [(1, 0)],
        # Add more neighbors as needed
    }
    return neighbors_map.get(node, [])

start = (0, 0)
goal = (1, 1)
path = a_star_search(start, goal, get_neighbors)
print("Path found:", path)
```

15/10/24
Tuesday

Lab - 3

(a) For 8-puzzle A* Implementation, to calculate. f(n), consider two Calc.

(i) g(n): Depth of the node, h(n): Number of misplaced tiles.

5	4	
6	1	8
7	3	2

Initial

	1	2
3	4	5
6	7	8

Goal state

Algorithm

Function init (board, zero_pos, moves=0):
    SET self.board = board
    SET self.zero_pos = zero_pos
    SET self.moves = moves
    SET self.heuristic := Calculate_misplaced_tiles()

Function calculate misplaced_tiles():
    SET - goal_state = [[1,2,3], [8,0,4], [7,6,5]]
RETURN Count of tiles no in goal_state

FUNCTION get_neighbors():
    SET neighbors = [ ]
FOR each direction in [up, Down, Left, Right]:
CALCULATE new position (new_x, new_y)
    IF new position is valid:
CREATE new_board by swapping zero with neighbor
ADD (new_board, new position) to neighbors

RETURN neighbors

FUNCTION __lt__(other)
RETURN (self.moves + self.heuristic) < (other.moves + other.heuristic)
~~RETUR~~
FUNCTION __repr__():
RETUN String representation of self.board, moves, and heuristic

FUNCTION a_star(initial_board):
FIND initial zero position
CREATE initial_state with PuzzleState

INITIALIZE priority_queue as Empty
PUSH initial state onto priority_queue
INITIALIZE visited as empty set

WHILE priority_queue is not Empty:
POP current_state from priority_queue
PRINT "Exploring state:", current_state

IF current_state is goal state:
PRINT "Reached goal in moves"
RETURN

ADD current_state to visited

FOR Each neighbor in current_state.get_neighbors():

PUSH neighbor-state onto priority-queue

PRINT "No Solution Found"

MAIN :

SET initial-board $\begin{bmatrix} 2 & 8 & 3 \\ 1 & 6 & 4 \\ 0 & 7 & 5 \end{bmatrix}$

CALL a-star (initial-board)

Output :

Exploring state : $\begin{bmatrix} 2 & 8 & 3 \\ 1 & 6 & 4 \\ 0 & 7 & 5 \end{bmatrix}$

(moves : 0, Heuristic : 6)

Exploring state : $\begin{bmatrix} 2 & 8 & 3 \\ 1 & 6 & 4 \\ 7 & 0 & 5 \end{bmatrix}$

(moves : 1, Heuristic : 5)

Exploring state : $\begin{bmatrix} 2 & 8 & 3 \\ 1 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix}$

(moves : 2, Heuristic : 3)

Exploring state : $\begin{bmatrix} 2 & 8 & 3 \\ 0 & 6 & 4 \\ 1 & 7 & 5 \end{bmatrix}$

(moves : 1, Heuristic : 6)

Exploring state : $\begin{bmatrix} 2 & 0 & 3 \\ 1 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix}$

(moves : 3, Heuristic : 4)

Exploring state : $\begin{bmatrix} 2 & 8 & 3 \\ \end{bmatrix}$

Exploring state : $\begin{bmatrix} 0 & 2 & 3 \\ 1 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix}$

(moves : 4 , Heuristic : 3)

Exploring state : $\begin{bmatrix} 2 & 8 & 3 \\ 0 & 1 & 4 \\ 7 & 6 & 5 \end{bmatrix}$

(moves : 3 , Heuristic : 4)

Exploring state : $\begin{bmatrix} 1 & 2 & 3 \\ 0 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix}$

(moves : 5 , Heuristic : 2)

Exploring state : $\begin{bmatrix} 1 & 2 & 3 \\ 8 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix}$

(moves : 6 , Heuristic : 0)