## **LAB-4** - 8 Puzzle with A*, IDDFS on a Graph

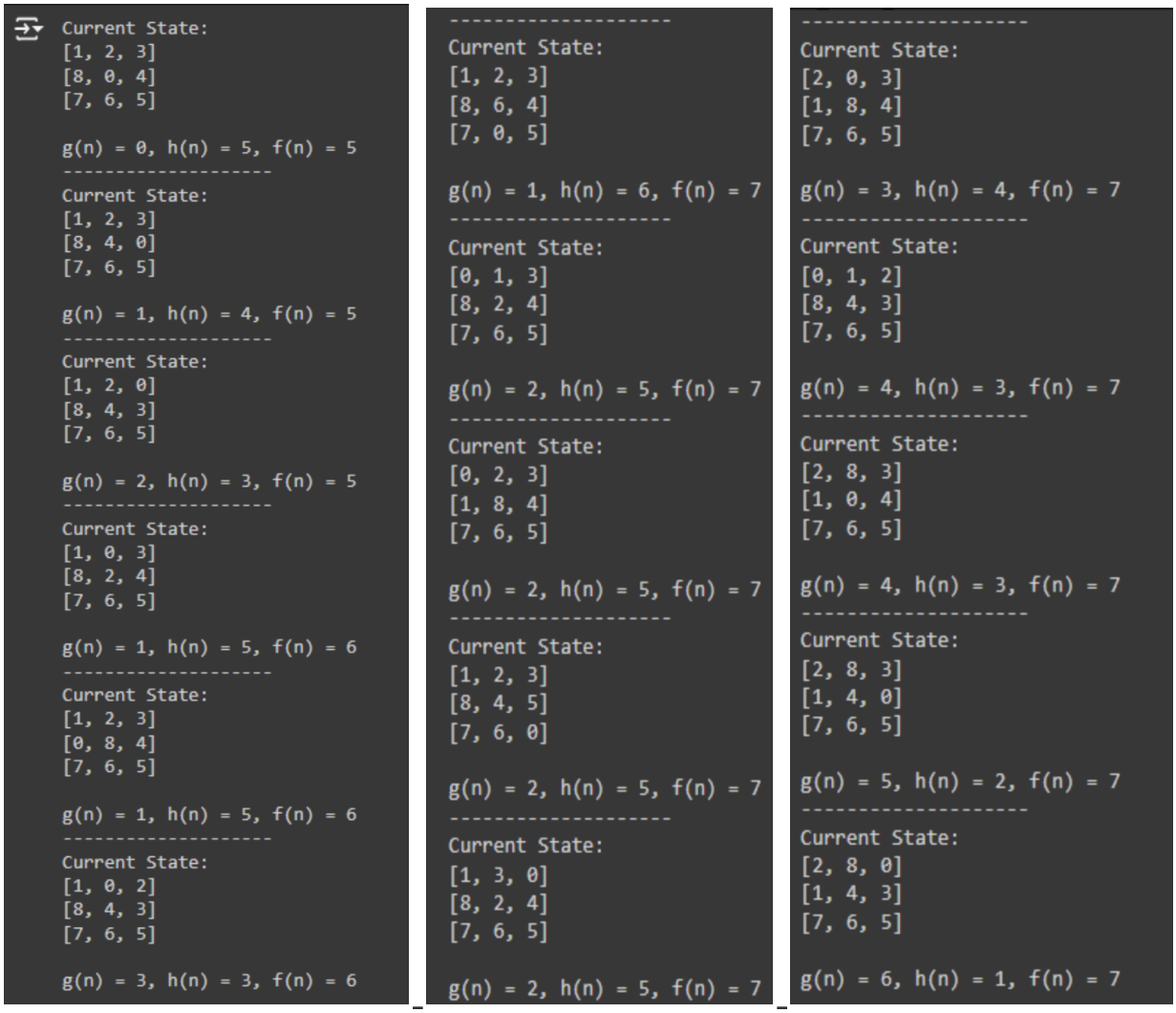
## **Code:** (8 Puzzle with A*)

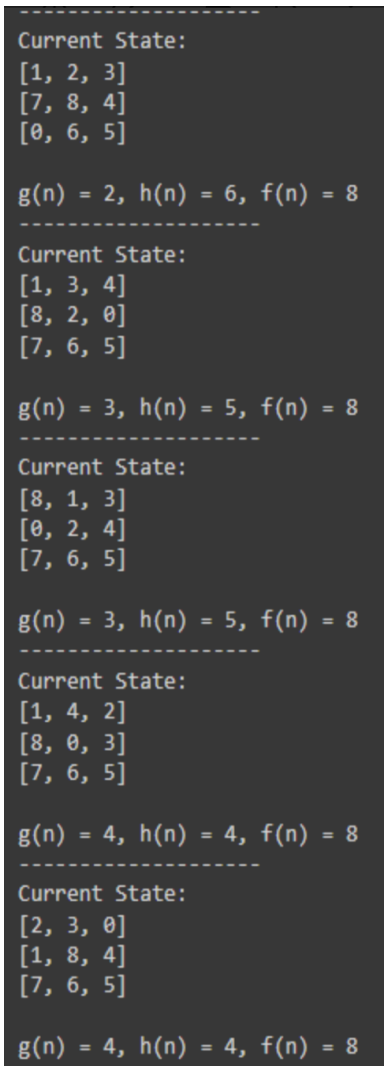
```
import heapq
# Goal state where blank (0) is the first tile
goal_state = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8]
]

# Helper functions
def flatten(puzzle):
    return [item for row in puzzle for item in row]

def find_blank(puzzle):
    for i in range(3):
        for j in range(3):
            if puzzle[i][j] == 0:
                return i, j

def misplaced_tiles(puzzle):
    flat_puzzle = flatten(puzzle)
    flat_goal = flatten(goal_state)
    return sum([1 for i in range(9) if flat_puzzle[i] != flat_goal[i] and flat_puzzle[i] != 0])

def generate_neighbors(puzzle):
    x, y = find_blank(puzzle)
    neighbors = []
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_puzzle = [row[:] for row in puzzle]
            new_puzzle[x][y], new_puzzle[nx][ny] = new_puzzle[nx][ny], new_puzzle[x][y]
            neighbors.append(new_puzzle)
    return neighbors
```

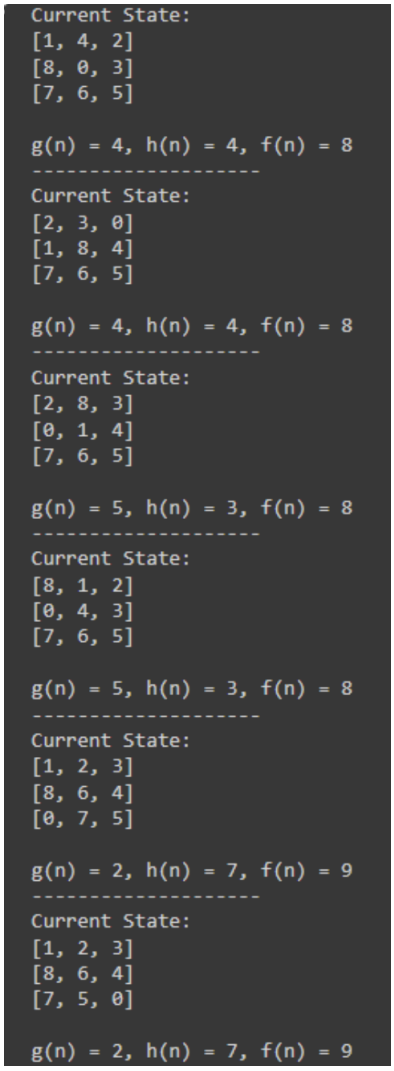
```
def is_goal(puzzle):
    return puzzle == goal_state

def print_puzzle(puzzle):
    for row in puzzle:
        print(row)
    print()

def a_star_misplaced_tiles(initial_state):
    # Priority queue (min-heap) and visited states
    frontier = []
    heapq.heappush(frontier, (misplaced_tiles(initial_state), 0, initial_state, []))
    visited = set()

    while frontier:
        f, g, current_state, path = heapq.heappop(frontier)

        # Print the current state
        print("Current State:")
        print_puzzle(current_state)
        h = misplaced_tiles(current_state)
        print(f"g(n) = {g}, h(n) = {h}, f(n) = {g + h}")
        print("-" * 20)

        if is_goal(current_state):
            print("Goal reached!")
            return path

        visited.add(tuple(flatten(current_state)))

        for neighbor in generate_neighbors(current_state):
            if tuple(flatten(neighbor)) not in visited:
                h = misplaced_tiles(neighbor)
                heapq.heappush(frontier, (g + 1 + h, g + 1, neighbor, path + [neighbor]))

    return None  # No solution found

# Initial puzzle state
initial_state = [
    [1, 2, 0],
```

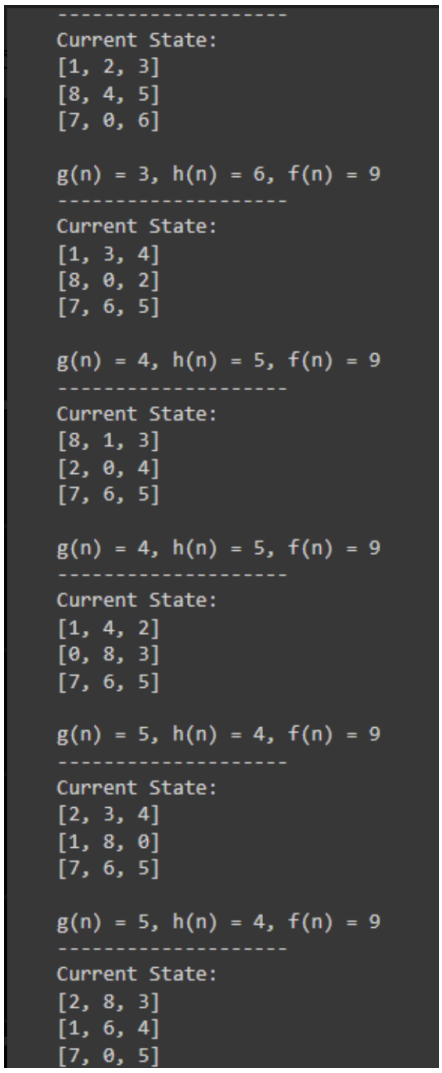
```
    [3, 4, 5],
    [6, 7, 8]
]

solution = a_star_misplaced_tiles(initial_state)
if solution:
    print("Solution found!")
else:
    print("No solution found.")
```

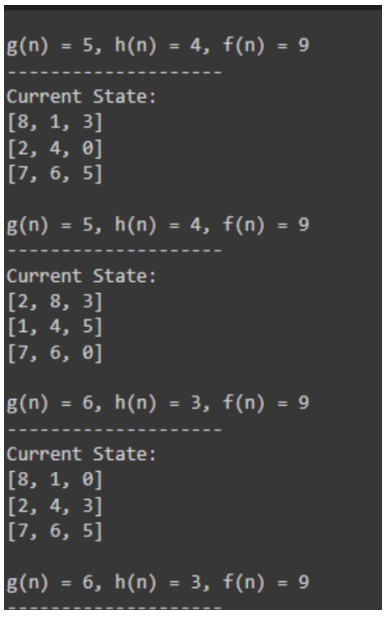
## Output:

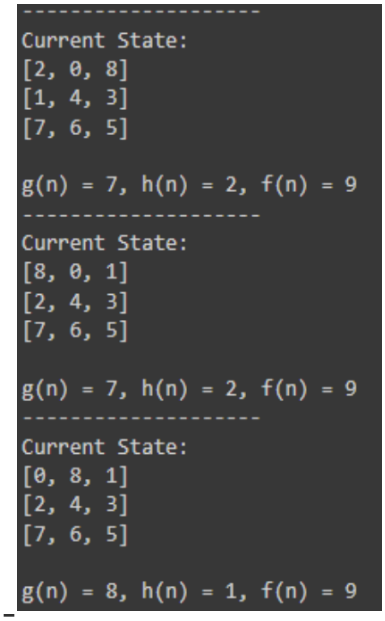


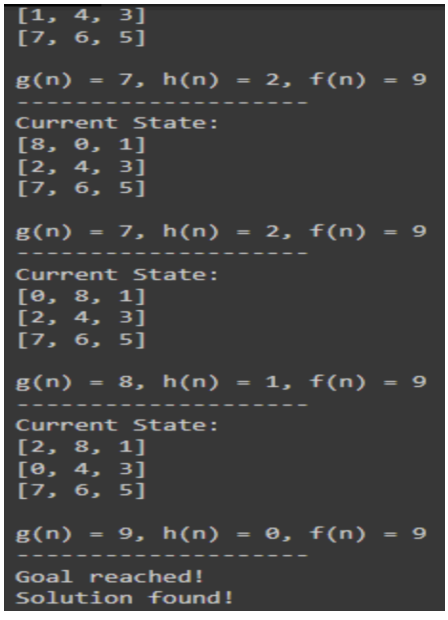

Current State:
[1, 2, 3]
[7, 8, 4]
[0, 6, 5]
g(n) = 2, h(n) = 6, f(n) = 8
Current State:
[1, 3, 4]
[8, 2, 0]
[7, 6, 5]
g(n) = 3, h(n) = 5, f(n) = 8
Current State:
[8, 1, 3]
[0, 2, 4]
[7, 6, 5]
g(n) = 3, h(n) = 5, f(n) = 8
Current State:
[1, 4, 2]
[8, 0, 3]
[7, 6, 5]
g(n) = 4, h(n) = 4, f(n) = 8
Current State:
[2, 3, 0]
[1, 8, 4]
[7, 6, 5]
g(n) = 4, h(n) = 4, f(n) = 8


Current State:
[1, 4, 2]
[8, 0, 3]
[7, 6, 5]
g(n) = 4, h(n) = 4, f(n) = 8
Current State:
[2, 3, 0]
[1, 8, 4]
[7, 6, 5]
g(n) = 4, h(n) = 4, f(n) = 8
Current State:
[2, 8, 3]
[0, 1, 4]
[7, 6, 5]
g(n) = 5, h(n) = 3, f(n) = 8
Current State:
[8, 1, 2]
[0, 4, 3]
[7, 6, 5]
g(n) = 5, h(n) = 3, f(n) = 8
Current State:
[1, 2, 3]
[8, 6, 4]
[0, 7, 5]
g(n) = 2, h(n) = 7, f(n) = 9
Current State:
[1, 2, 3]
[8, 6, 4]
[7, 5, 0]
g(n) = 2, h(n) = 7, f(n) = 9


Current State:
[1, 2, 3]
[8, 4, 5]
[7, 0, 6]
g(n) = 3, h(n) = 6, f(n) = 9
Current State:
[1, 3, 4]
[8, 0, 2]
[7, 6, 5]
g(n) = 4, h(n) = 5, f(n) = 9
Current State:
[8, 1, 3]
[2, 0, 4]
[7, 6, 5]
g(n) = 4, h(n) = 5, f(n) = 9
Current State:
[1, 4, 2]
[0, 8, 3]
[7, 6, 5]
g(n) = 5, h(n) = 4, f(n) = 9
Current State:
[2, 3, 4]
[1, 8, 0]
[7, 6, 5]
g(n) = 5, h(n) = 4, f(n) = 9
Current State:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]


g(n) = 5, h(n) = 4, f(n) = 9
Current State:
[8, 1, 3]
[2, 4, 0]
[7, 6, 5]
g(n) = 5, h(n) = 4, f(n) = 9
Current State:
[2, 8, 3]
[1, 4, 5]
[7, 6, 0]
g(n) = 6, h(n) = 3, f(n) = 9
Current State:
[8, 1, 0]
[2, 4, 3]
[7, 6, 5]
g(n) = 6, h(n) = 3, f(n) = 9


Current State:
[2, 0, 8]
[1, 4, 3]
[7, 6, 5]
g(n) = 7, h(n) = 2, f(n) = 9
Current State:
[8, 0, 1]
[2, 4, 3]
[7, 6, 5]
g(n) = 7, h(n) = 2, f(n) = 9
Current State:
[0, 8, 1]
[2, 4, 3]
[7, 6, 5]
g(n) = 8, h(n) = 1, f(n) = 9


[1, 4, 3]
[7, 6, 5]
g(n) = 7, h(n) = 2, f(n) = 9
Current State:
[8, 0, 1]
[2, 4, 3]
[7, 6, 5]
g(n) = 7, h(n) = 2, f(n) = 9
Current State:
[0, 8, 1]
[2, 4, 3]
[7, 6, 5]
g(n) = 8, h(n) = 1, f(n) = 9
Current State:
[2, 8, 1]
[0, 4, 3]
[7, 6, 5]
g(n) = 9, h(n) = 0, f(n) = 9
Goal reached!
Solution found!

## **Code:** (IDDFS on a Graph )

```
class Graph:
    def __init__(self):
        self.adjacency_list = {}

    def add_edge(self, u, v):
        if u not in self.adjacency_list:
            self.adjacency_list[u] = []
        self.adjacency_list[u].append(v)

    def depth_limited_dfs(self, node, goal, limit, visited):
        if limit < 0:
            return False
        if node == goal:
            return True

        visited.add(node)
        for neighbor in self.adjacency_list.get(node, []):
            if neighbor not in visited:
                if self.depth_limited_dfs(neighbor, goal, limit - 1, visited):
                    return True
        visited.remove(node)  # Allow revisiting for the next iteration
        return False

    def iddfs(self, start, goal, max_depth):
        for depth in range(max_depth + 1):
            visited = set()
            if self.depth_limited_dfs(start, goal, depth, visited):
                return True
        return False

def main():
    graph = Graph()

    # Input number of edges
    num_edges = int(input("Enter the number of edges: "))

    # Input edges
    for _ in range(num_edges):
```

```
        edge = input("Enter an edge (format: A B): ").split()
        graph.add_edge(edge[0], edge[1])

    start_node = input("Enter the start node: ")
    goal_node = input("Enter the goal node: ")
    max_depth = int(input("Enter the maximum depth for IDDFS: "))

    if graph.iddfs(start_node, goal_node, max_depth):
        print(f"Goal node {goal_node} found!")
    else:
        print(f"Goal node {goal_node} not found within depth {max_depth}.")

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

## Output:

```
Enter the number of edges: 14
Enter an edge (format: A B): y p
Enter an edge (format: A B): y x
Enter an edge (format: A B): p r
Enter an edge (format: A B): p s
Enter an edge (format: A B): x f
Enter an edge (format: A B): x h
Enter an edge (format: A B): r b
Enter an edge (format: A B): r c
Enter an edge (format: A B): s X
Enter an edge (format: A B): s z
Enter an edge (format: A B): f u
Enter an edge (format: A B): f e
Enter an edge (format: A B): h l
Enter an edge (format: A B): h w
Enter the start node: y
Enter the goal node: f
Enter the maximum depth for IDDFS: 3
Goal node f found!
```