Implement BFS

05/8/25. BFS

Algorithm:

* Check the initial state with goal state, if matching break, else move the empty tile to all the possible positions [up, right, left, down].
* Now check all the new states with goal state.
* If any of the state match goal state, break, else move the empty tile to all the possible positions and gain new states.
* Check all the new states with goal state and repeat the process until the goal state is achieved.

~~Output:~~

Using BFS solve 8 puzzle without heuristic.

1/9/25.

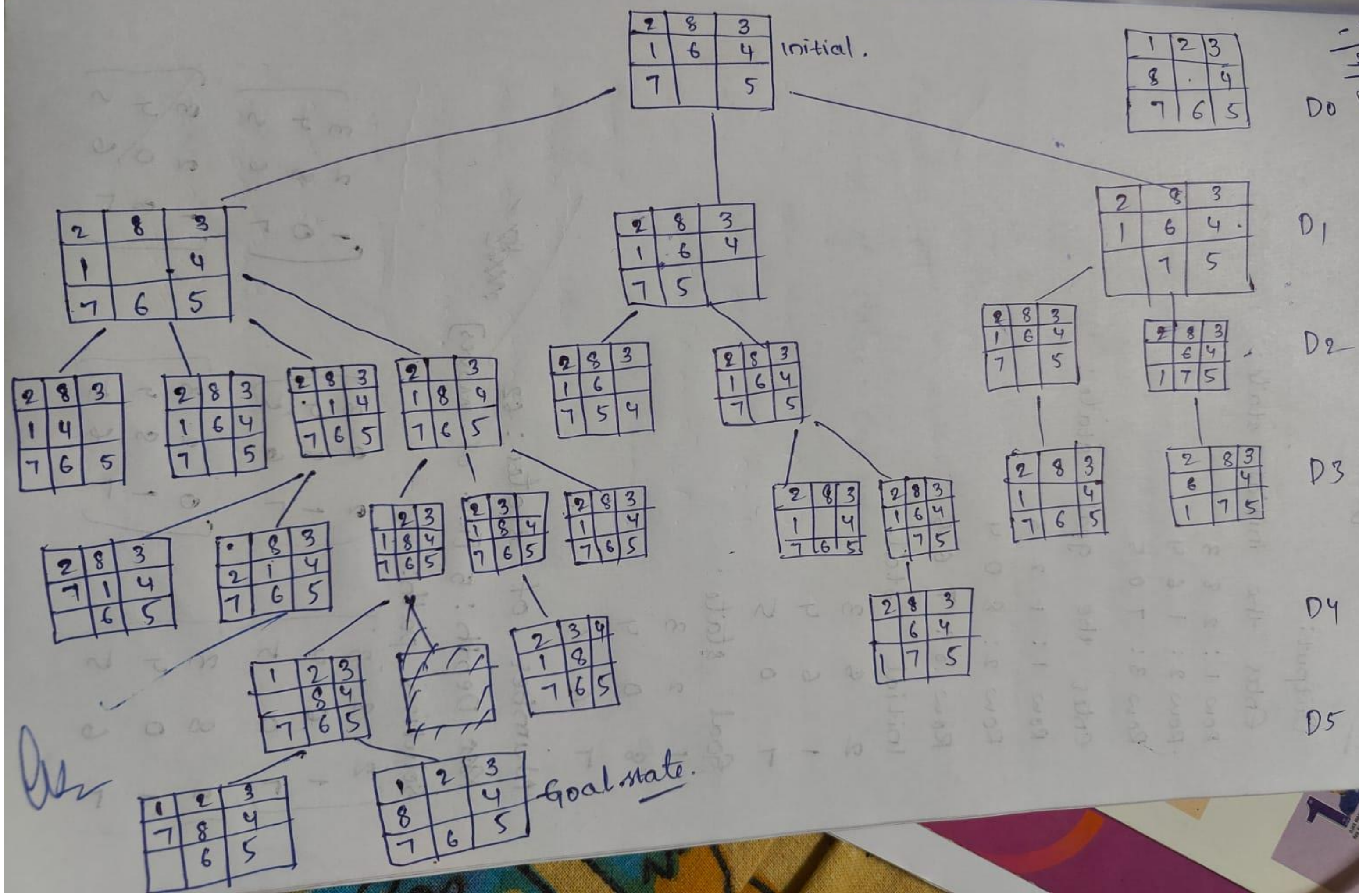
DO

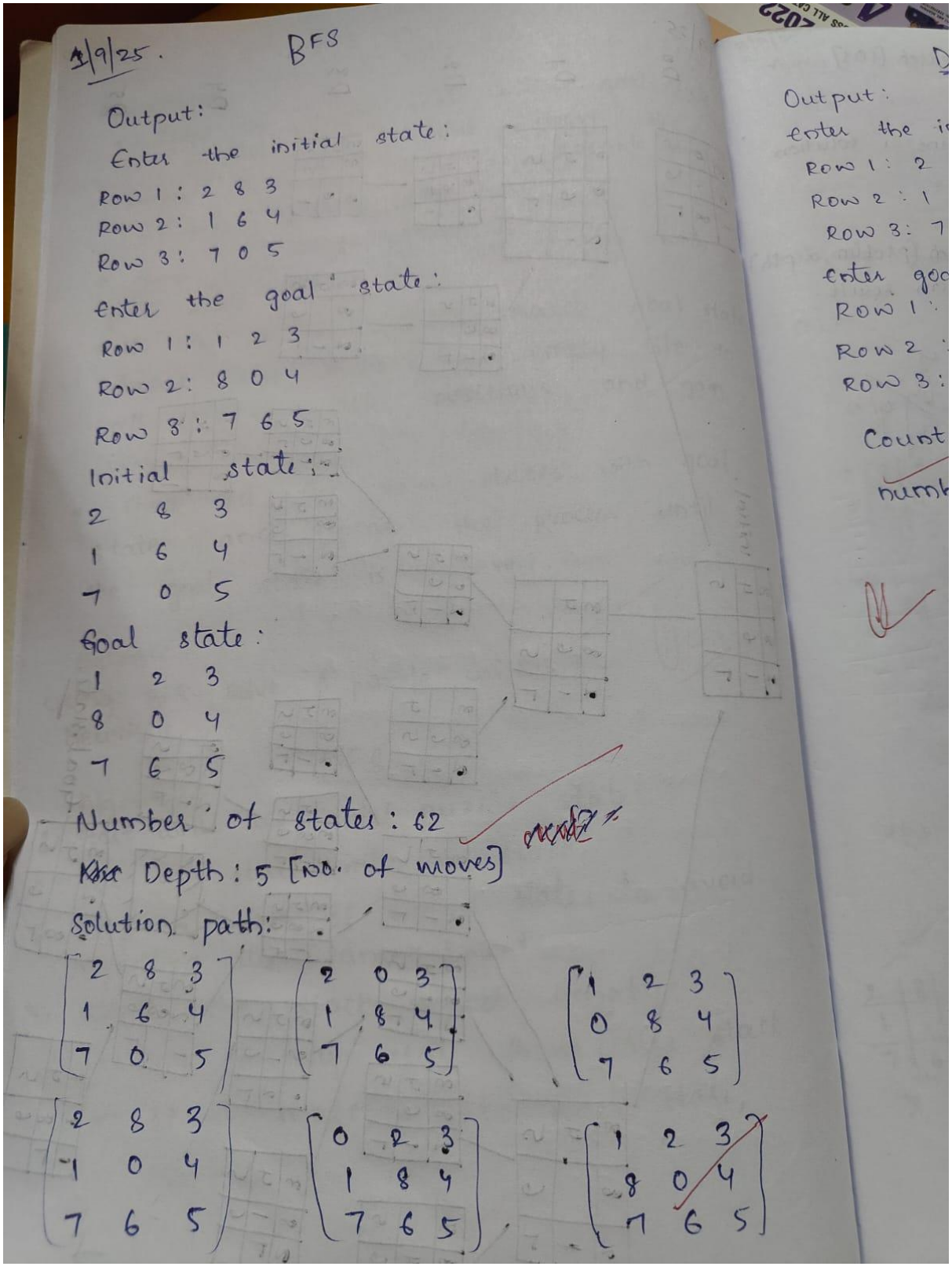
DFS.

Algorithm:

1. Start with the initial puzzle state.
2. Put this state into a stack.
3. Keep a set of visited states to avoid repeating the same board.
4. While the stack is not empty:
   → Take the top state from the stack.
   → If it matches the goal state, stop.

1/9/25.




1/9/25. BFS
Output:
Enter the initial state:
Row 1: 2 8 3
Row 2: 1 6 4
Row 3: 7 0 5
Enter the goal state:
Row 1: 1 2 3
Row 2: 8 0 4
Row 3: 7 6 5
Initial state:
2 8 3
1 6 4
7 0 5
Goal state:
1 2 3
8 0 4
7 6 5
Number of states: 62
Depth: 5 [No. of moves]
Solution path:
Output:
Enter the
Row 1: 2
Row 2: 1
Row 3: 7
Enter goa
Row 1:
Row 2
Row 3:
Count
numb

Output:

```
Enter the INITIAL 8-puzzle board configuration (use 0 for blank):
Row 1 (3 numbers space-separated): 2 8 3
Row 2 (3 numbers space-separated): 1 6 4
Row 3 (3 numbers space-separated): 7 0 5

Enter the GOAL 8-puzzle board configuration (use 0 for blank):
Row 1 (3 numbers space-separated): 1 2 3
Row 2 (3 numbers space-separated): 8 0 4
Row 3 (3 numbers space-separated): 7 6 5

Initial State:
2 8 3
1 6 4
7 0 5

Goal State:
1 2 3
8 0 4
7 6 5


 Goal reached!
 Number of states explored: 62
  Number of moves: 5

Solution path:
2 8 3
1 6 4
7 0 5

2 8 3
1 0 4
7 6 5

2 0 3
1 8 4
7 6 5
```

```
2 0 3
1 8 4
7 6 5

0 2 3
1 8 4
7 6 5

1 2 3
0 8 4
7 6 5

1 2 3
8 0 4
7 6 5

Sareddy Poojya Sree
1BM23CS303
```

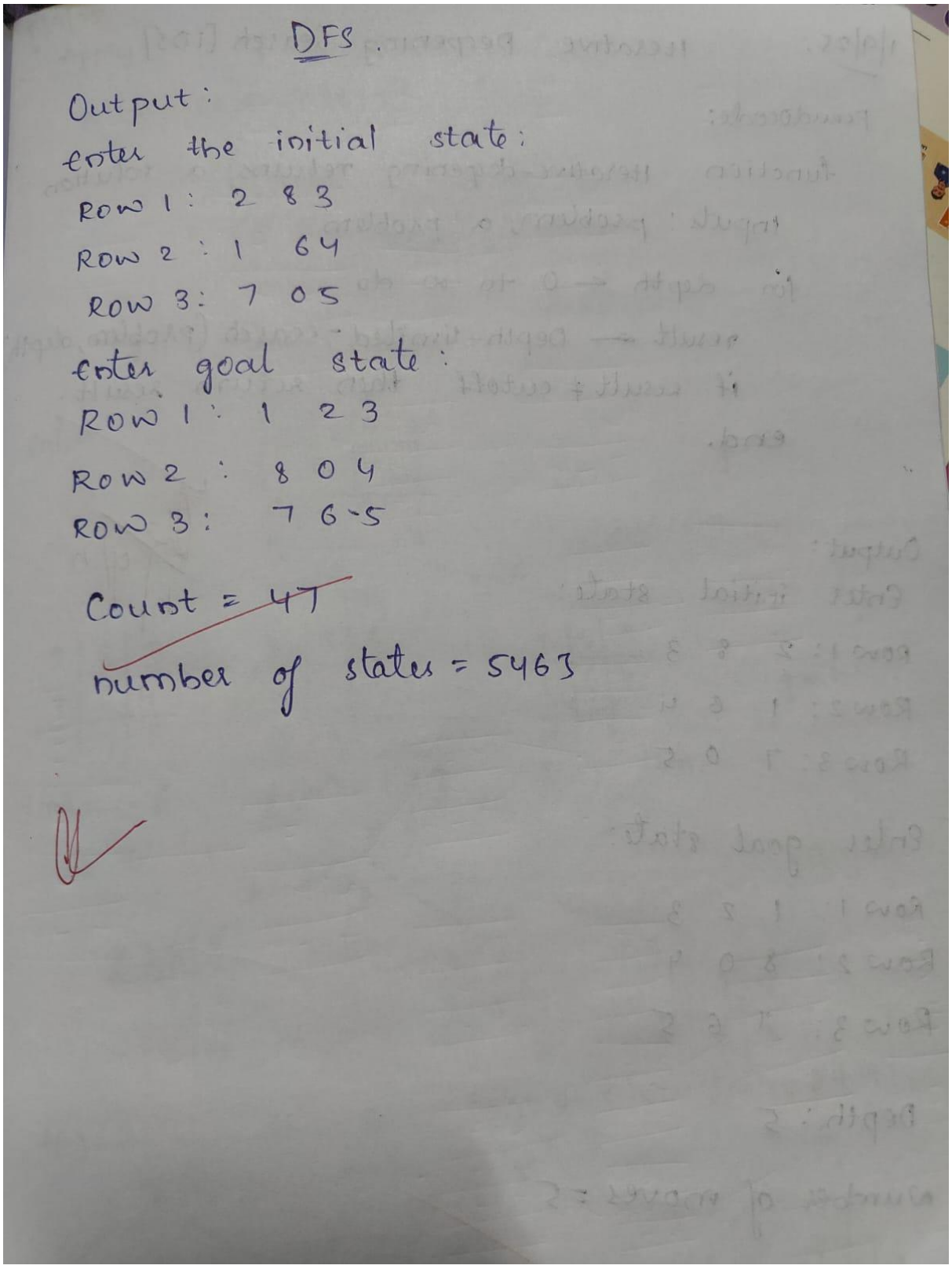
Code:

```
from collections import deque

# Function to display puzzle state
def print_state(state):
    for i in range(0, 9, 3):
        print(" ".join(state[i:i+3]))
    print()

# Generate neighbors
def get_neighbors(state):
    neighbors = []
```

```
    index = state.index("0")  # blank position
    row, col = divmod(index, 3)

    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # up, down, left, right

    for dr, dc in moves:
        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
            state_list = list(state)
            # swap blank
            state_list[index], state_list[new_index] = state_list[new_index], state_list[index]
            neighbors.append("".join(state_list))
    return neighbors

# BFS solver
def bfs(start, goal):
    visited = set()
    queue = deque([(start, [])])  # (state, path)
    visited.add(start)
    state_count = 1

    while queue:
        state, path = queue.popleft()

        if state == goal:
            print("Number of states explored:", state_count)
            print("  Number of moves:", len(path))
            print("\nSolution path:")
            for s in path + [state]:
                print_state(s)
            return path

        for neighbor in get_neighbors(state):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, path + [state]))
                state_count += 1

    print("No solution found.")
    return None

# -------------------------
# MAIN PROGRAM
# -------------------------
print("Enter the INITIAL 8-puzzle board configuration (use 0 for blank):")
```

```
initial_board = []
for i in range(3):
    row = input(f"Row {i+1} (3 numbers space-separated): ").split()
    initial_board.extend(row)

print("\nEnter the GOAL 8-puzzle board configuration (use 0 for blank):")
goal_board = []
for i in range(3):
    row = input(f"Row {i+1} (3 numbers space-separated): ").split()
    goal_board.extend(row)

start_state = "".join(initial_board)
goal_state = "".join(goal_board)

print("\nInitial State:")
print_state(start_state)

print("Goal State:")
print_state(goal_state)

bfs(start_state, goal_state)
```

Implement DFS



Output:

```
Enter the INITIAL 8-puzzle board configuration (use 0 for blank):
Row 1 (3 numbers space-separated): 2 8 3
Row 2 (3 numbers space-separated): 1 6 4
Row 3 (3 numbers space-separated): 7 0 5

Enter the GOAL 8-puzzle board configuration (use 0 for blank):
Row 1 (3 numbers space-separated): 1 2 3
Row 2 (3 numbers space-separated): 8 0 4
Row 3 (3 numbers space-separated): 7 6 5

Initial State:
2 8 3
1 6 4
7 0 5

Goal State:
1 2 3
8 0 4
7 6 5

count: 47

Solution path:
2 8 3
1 6 4
7 0 5
```

```
1 2 3
8 6 4
0 7 5

1 2 3
8 6 4
7 0 5

1 2 3
8 0 4
7 6 5

Sareddy Poojya Sree
1BM23CS303
```

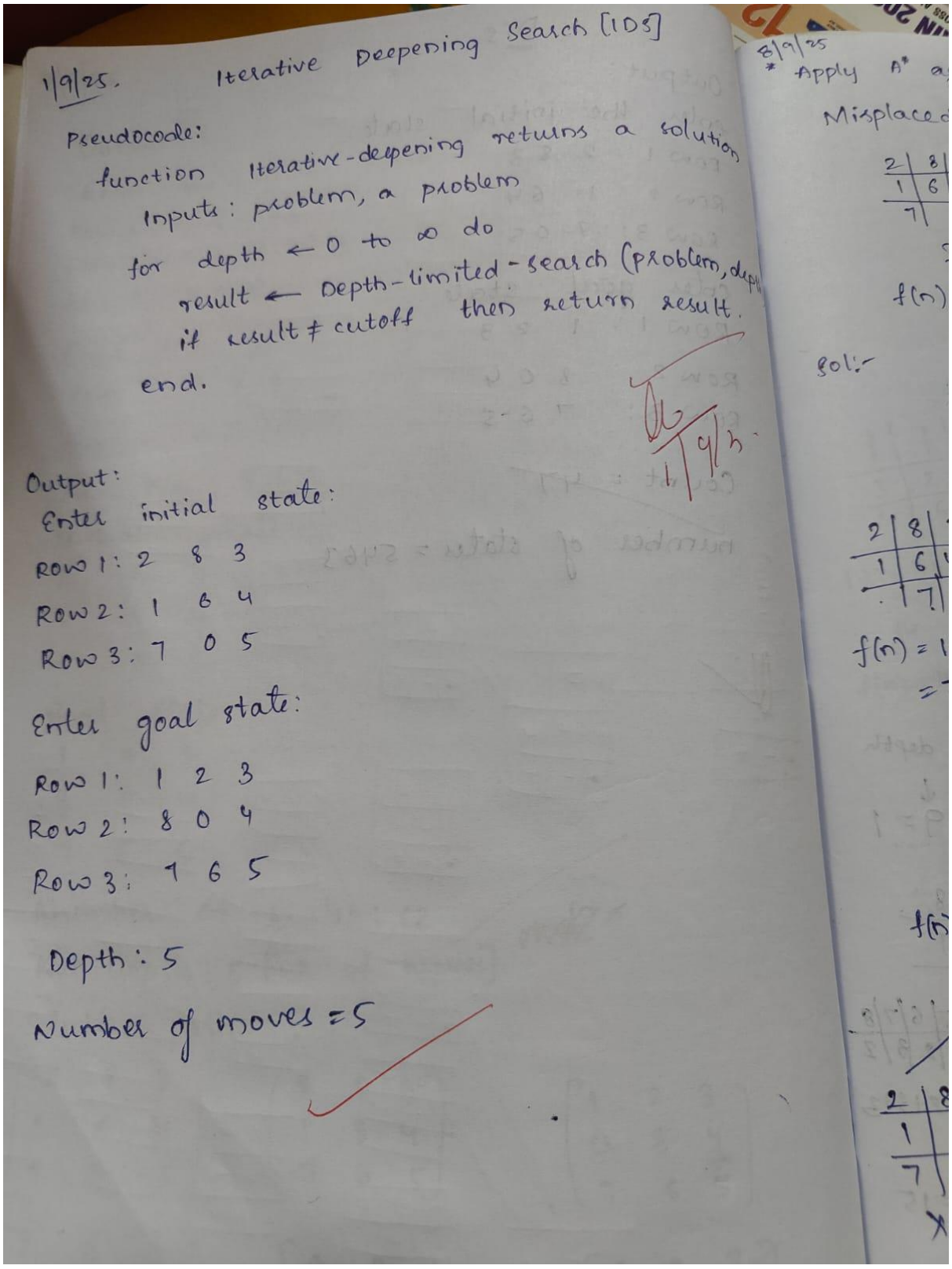
Code:

```
from collections import deque
```

```python
# Function to display puzzle state
def print_state(state):
    for i in range(0, 9, 3):
        print(" ".join(state[i:i+3]))
    print()

# Generate neighbors
def get_neighbors(state):
    neighbors = []
    index = state.index("0")  # blank position
    row, col = divmod(index, 3)

    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # up, down, left, right

    for dr, dc in moves:
        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
            state_list = list(state)
            # swap blank
            state_list[index], state_list[new_index] = state_list[new_index], state_list[index]
            neighbors.append("".join(state_list))
    return neighbors

# DFS solver (recursive)
def dfs(start, goal, visited=None, path=None, state_count=[0], max_depth=50):
    if visited is None:
        visited = set()
    if path is None:
        path = []

    visited.add(start)
    state_count[0] += 1

    if start == goal:

        #print("Number of states explored:", state_count[0])
        print("count:", len(path))
        print("\nSolution path:")
        for s in path + [start]:
            print_state(s)
        return path

    if len(path) >= max_depth:  # prevent infinite recursion
        return None

    for neighbor in get_neighbors(start):
        if neighbor not in visited:
            result = dfs(neighbor, goal, visited, path + [start], state_count, max_depth)
```

```
            if result is not None:
                return result

    return None

# -------------------------
# MAIN PROGRAM
# -------------------------
print("Enter the INITIAL 8-puzzle board configuration (use 0 for blank):")
initial_board = []
for i in range(3):
    row = input(f"Row {i+1} (3 numbers space-separated): ").split()
    initial_board.extend(row)

print("\nEnter the GOAL 8-puzzle board configuration (use 0 for blank):")
goal_board = []
for i in range(3):
    row = input(f"Row {i+1} (3 numbers space-separated): ").split()
    goal_board.extend(row)

start_state = "".join(initial_board)
goal_state = "".join(goal_board)

print("\nInitial State:")
print_state(start_state)

print("Goal State:")
print_state(goal_state)

dfs(start_state, goal_state)

print("Sareddy Poojya Sree")
print("1BM23CS303")
```

Implement Iterative Deepening DFS



1/9/25. Iterative Deepening Search [IDS]

Pseudocode:

function Iterative-deepening returns a solution
inputs: problem, a problem
for depth ← 0 to ∞ do
result ← Depth-limited-search (problem, depth)
if result ≠ cutoff then return result.
end.

Output:
Enter initial state:
Row 1: 2 8 3
Row 2: 1 6 4
Row 3: 7 0 5

Enter goal state:
Row 1: 1 2 3
Row 2: 8 0 4
Row 3: 7 6 5

Depth: 5

Number of moves = 5

Output:

```
Enter initial state (9 numbers, use 0 for blank):
2 8 3 1 6 4 7 0 5
Enter goal state (9 numbers, use 0 for blank):
1 2 3 8 0 4 7 6 5

Solution found at depth: 5
Number of moves: 5

Steps:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

(2, 8, 3)
(1, 0, 4)
(7, 6, 5)

(2, 0, 3)
(1, 8, 4)
(7, 6, 5)

(0, 2, 3)
(1, 8, 4)
(7, 6, 5)

(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

Sareddy Poojya Sree
1BM23CS303
```


Code:

```
def get_neighbors(state):
    neighbors = []
    blank = state.index(0)
    x, y = divmod(blank, 3)

    moves = [(-1,0), (1,0), (0,-1), (0,1)]  # up, down, left, right
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_blank = nx*3 + ny
            new_state = list(state)
            new_state[blank], new_state[new_blank] = new_state[new_blank], new_state[blank]
            neighbors.append(tuple(new_state))
    return neighbors

# Depth Limited Search (recursive)
def depth_limited_search(state, goal, limit, path, visited):
    if state == goal:
        return path
```

```
    if limit == 0:
        return None

    visited.add(state)
    for neighbor in get_neighbors(state):
        if neighbor not in visited:
            result = depth_limited_search(neighbor, goal, limit - 1, path + [neighbor], visited)
            if result is not None:
                return result
    return None

# Iterative Deepening Search
def iterative_deepening_search(initial, goal):
    depth = 0
    while True:
        visited = set()
        result = depth_limited_search(initial, goal, depth, [initial], visited)
        if result is not None:
            return result, depth
        depth += 1

# Print puzzle in 3x3 grid
def print_state(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()

# Main
if __name__ == "__main__":
    print("Enter initial state (9 numbers, use 0 for blank):")
    initial = tuple(map(int, input().split()))
    print("Enter goal state (9 numbers, use 0 for blank):")
    goal = tuple(map(int, input().split()))

    path, depth = iterative_deepening_search(initial, goal)

    print("\nSolution found at depth:", depth)
    print("Number of moves:", len(path)-1)
    print("\nSteps:")
    for step in path:
        print_state(step)
print("Sareddy Poojya Sree\n1BM23CS303")
```