3. Solve 8 puzzle problem using BFS and DFS.

USING BFS

Code:

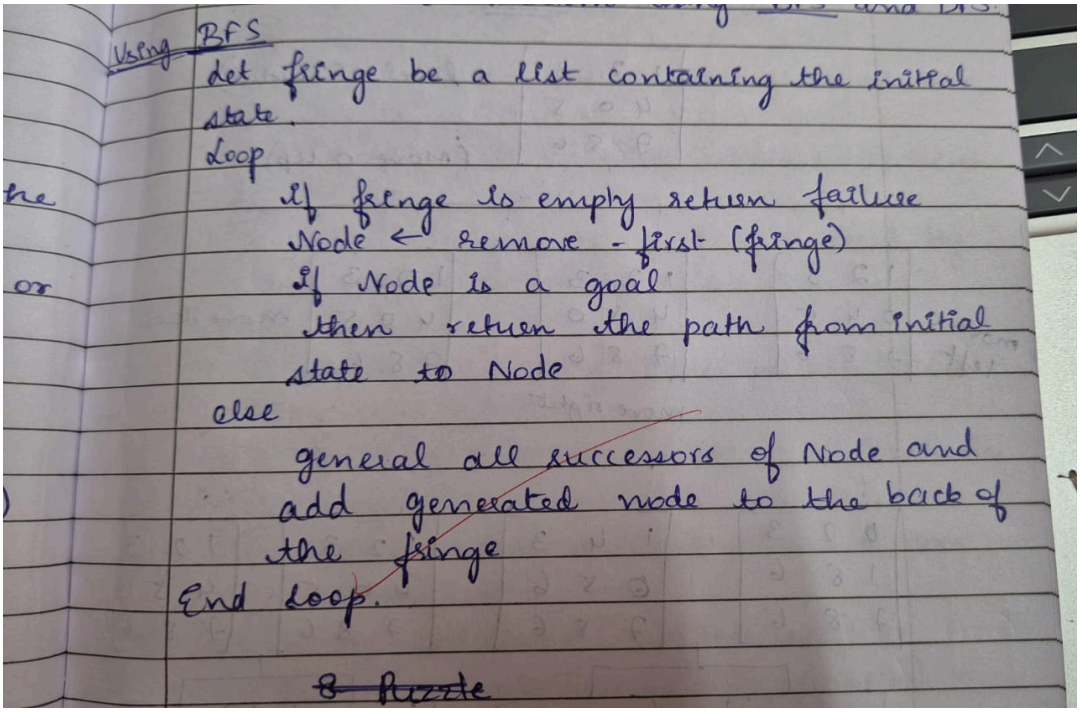
```
from collections import deque
print("Sonal,1BM22CS286")

class PuzzleState:
    def __init__(self, board, zero_position, path=[]):
        self.board = board
        self.zero_position = zero_position
        self.path = path

    def is_goal(self):
        return self.board == [1, 2, 3, 4, 5, 6, 7, 8, 0]

    def get_possible_moves(self):
        moves = []
        row, col = self.zero_position
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left, Up

        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_board = self.board[:]
                # Swap zero with the adjacent tile
                new_board[row * 3 + col], new_board[new_row * 3 + new_col] = new_board[new_row *
    3 + new_col], new_board[row * 3 + col]
                moves.append(PuzzleState(new_board, (new_row, new_col), self.path + [new_board]))

        return moves

def bfs(initial_state):
    queue = deque([initial_state])
    visited = set()

    while queue:
        current_state = queue.popleft()

        # Show the current board
        print("Current Board State:")
        print_board(current_state.board)
        print()
```

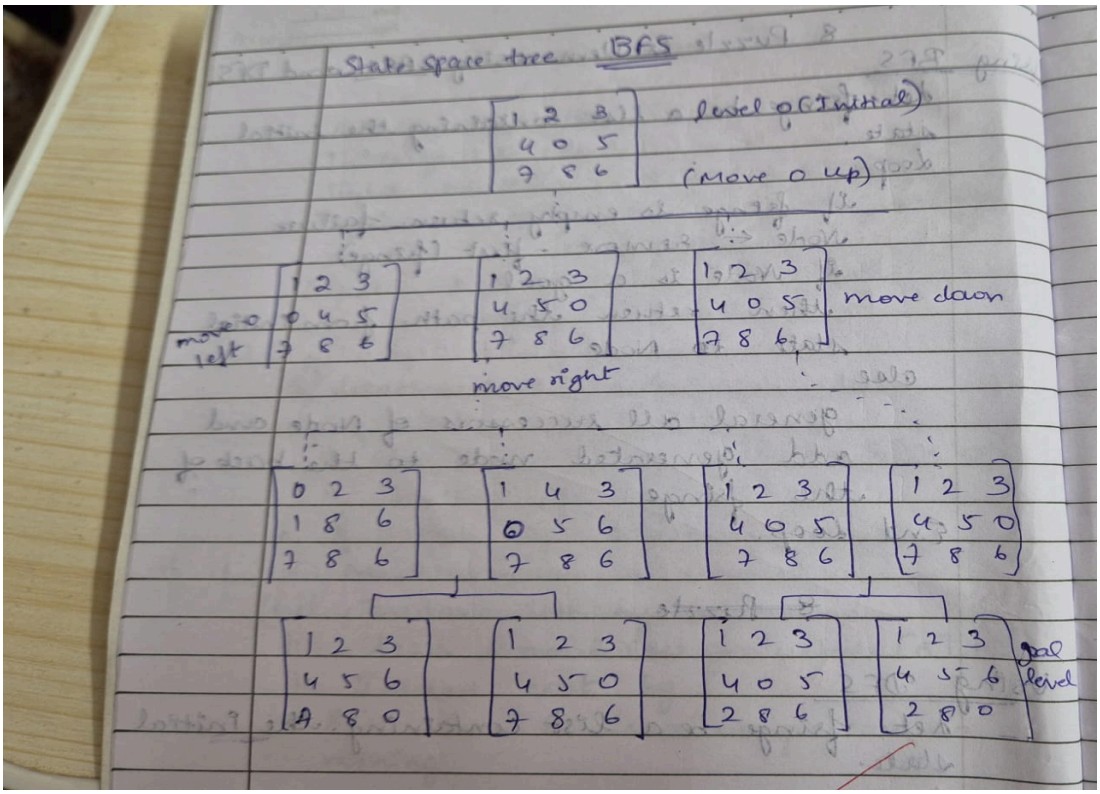
```
        if current_state.is_goal():
            return current_state.path

        visited.add(tuple(current_state.board))

        for next_state in current_state.get_possible_moves():
            if tuple(next_state.board) not in visited:
                queue.append(next_state)

    return None

def print_board(board):
    for i in range(3):
        print(board[i * 3:i * 3 + 3])

def main():
    print("Enter the initial state of the 8-puzzle (use 0 for the blank tile, e.g., '1 2 3 4 5 6 7 8 0'): ")
    user_input = input()
    initial_board = list(map(int, user_input.split()))

    if len(initial_board) != 9 or set(initial_board) != set(range(9)):
        print("Invalid input! Please enter 9 numbers from 0 to 8.")
        return

    zero_position = initial_board.index(0)
    initial_state = PuzzleState(initial_board, (zero_position // 3, zero_position % 3))

    solution_path = bfs(initial_state)

    if solution_path is None:
        print("No solution found.")
    else:
        print("Solution found in", len(solution_path), "steps.")
        for step in solution_path:
            print_board(step)
            print()

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

Output :

```
Sonal,1BM22CS286
Enter the initial state of the 8-puzzle (use 0 for the blank tile, e.g., '1 2 3 4 5 6 7 8 0'):
1 2 3 4 0 6 7 5 8
Current Board State:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Current Board State:
[1, 2, 3]
[4, 6, 0]
[7, 5, 8]

Current Board State:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Current Board State:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]

Current Board State:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]

 Current Board State:
 [1, 0, 3]
 [4, 2, 6]
 [7, 5, 8]

 Current Board State:
 [1, 2, 3]
 [4, 6, 8]
 [7, 5, 0]

 Current Board State:
 [1, 2, 0]
 [4, 6, 3]
 [7, 5, 8]

 Current Board State:
 [1, 2, 3]
 [4, 5, 6]
 [7, 8, 0]

 Solution found in 2 steps.
 [1, 2, 3]
 [4, 5, 6]
 [7, 0, 8]

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

Observation book screenshots:


Using BFS
Let fringe be a list containing the initial state.
Loop
If fringe is empty return failure
Node ← remove-first (fringe)
If Node is a goal
then return the path from initial state to Node
else
general all successors of Node and add generated node to the back of the fringe
End loop.
8 Puzzle


State space tree BFS
level 0 (Initial)
(move 0 up)
move left
move right
move down
goal level

USING DFS

Code :

```
from collections import deque
print("Sonal,1BM22CS286")

def get_user_input(prompt):
    board = []
    print(prompt)
    for i in range(3):
        row = list(map(int, input(f"Enter row {i+1} (space-separated numbers, use 0 for empty
      space): ").split()))
        board.append(row)
    return board

def is_solvable(board):
    flattened_board = [tile for row in board for tile in row if tile != 0]
    inversions = 0
    for i in range(len(flattened_board)):
        for j in range(i + 1, len(flattened_board)):
            if flattened_board[i] > flattened_board[j]:
                inversions += 1
    return inversions % 2 == 0

class PuzzleState:
    def __init__(self, board, moves=0, previous=None):
        self.board = board
        self.empty_tile = self.find_empty_tile()
        self.moves = moves
        self.previous = previous

    def find_empty_tile(self):
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return (i, j)

    def is_goal(self, goal_state):
        return self.board == goal_state

    def get_possible_moves(self):
        row, col = self.empty_tile
        possible_moves = []
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]  # down, up, right, left

        for dr, dc in directions:
```

```
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                # Make the move
                new_board = [row[:] for row in self.board]  # Deep copy
                new_board[row][col], new_board[new_row][new_col] =
        new_board[new_row][new_col], new_board[row][col]
                possible_moves.append(PuzzleState(new_board, self.moves + 1, self))

        return possible_moves

def dfs(initial_state, goal_state):
    stack = [initial_state]
    visited = set()

    while stack:
        current_state = stack.pop()

        if current_state.is_goal(goal_state):
            return current_state

        # Convert board to a tuple for the visited set
        state_tuple = tuple(tuple(row) for row in current_state.board)

        if state_tuple not in visited:
            visited.add(state_tuple)
            for next_state in current_state.get_possible_moves():
                stack.append(next_state)

    return None  # No solution found

def print_solution(solution):
    path = []
    while solution:
        path.append(solution.board)
        solution = solution.previous
    for state in reversed(path):
        for row in state:
            print(row)
        print()

if __name__ == "__main__":
    # Get user input for initial and goal states
    initial_board = get_user_input("Enter the initial state of the puzzle:")
    goal_board = get_user_input("Enter the goal state of the puzzle:")
```

```
if is_solvable(initial_board):
    initial_state = PuzzleState(initial_board)
    solution = dfs(initial_state, goal_board)

    if solution:
        print("Solution found in", solution.moves, "moves:")
        print_solution(solution)
    else:
        print("No solution found.")
else:
    print("This puzzle is unsolvable.")
```

Output:

```
Streaming output truncated to the last 5000 lines.
[2, 5, 6]
[0, 3, 1]
[4, 7, 8]

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

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

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

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

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

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

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

Observation book screenshots:

Using DFS

Let fringe be a list containing the initial state.

Loop

if fringe is empty return failure

Node ← remove-first (fringe)

If Node is a goal

then return the path from initial state to Node

else

general all successors of Node and add generated node to the front of fringe.

End loop.