SHREE SANKET

1BM22CS261

A) Depening Search Algorithm for 8Puzzle Problem

Code:

```
class PuzzleState:
    def __init__(self, board, empty_tile_pos, depth=0, path=[]):
        self.board = board
        self.empty_tile_pos = empty_tile_pos
        self.depth = depth
        self.path = path

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

    def generate_moves(self):
        row, col = self.empty_tile_pos
        moves = []
        directions = [(-1, 0, 'Up'), (1, 0, 'Down'), (0, -1, 'Left'), (0, 1, 'Right')]
        for dr, dc, move_name in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_board = self.board[:]
                new_board[row * 3 + col], new_board[new_row * 3 + new_col] = new_board[new_row * 3
+ new_col], new_board[row * 3 + col]
                new_path = self.path + [move_name]
                moves.append(PuzzleState(new_board, (new_row, new_col), self.depth + 1, new_path))
        return moves

    def display(self):
```

```
        for i in range(0, 9, 3):
            print(self.board[i:i + 3])
        print(f"Moves: {self.path}")
        print()

def iddfs(initial_state, goal, max_depth):
    for depth in range(max_depth + 1):
        print(f"Searching at depth: {depth}")
        found = dls(initial_state, goal, depth)
        if found:
            print(f"Goal found at depth: {found.depth}")
            found.display()
            return found
    print("Goal not found within max depth.")
    return None

def dls(state, goal, depth):
    if state.is_goal(goal):
        return state

    if depth <= 0:
        return None

    for move in state.generate_moves():
        print("Current state:")
        move.display()
        result = dls(move, goal, depth - 1)
        if result is not None:
            return result
    return None
```

```python
def main():

    initial_state_input = input("Enter initial state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0'): ")
    goal_state_input = input("Enter goal state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0'): ")
    max_depth = int(input("Enter maximum depth: "))

    initial_board = list(map(int, initial_state_input.split()))
    goal_board = list(map(int, goal_state_input.split()))
    empty_tile_pos = initial_board.index(0) // 3, initial_board.index(0) % 3

    initial_state = PuzzleState(initial_board, empty_tile_pos)

    solution = iddfs(initial_state, goal_board, max_depth)

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

OUTPUT:

```
Enter initial state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0'): 1 2 3 4 0 6 7 5 8
Enter goal state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0'): 1 2 3 4 5 6 7 8 0
Enter maximum depth: 3
Searching at depth: 0
Searching at depth: 1
Current state:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]
Moves: ['Up']

Current state:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
Moves: ['Down']

Current state:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
Moves: ['Left']

Current state:
[1, 2, 3]
[4, 6, 0]
[7, 5, 8]
Moves: ['Right']
```

```
Current state:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
Moves: ['Down']

Current state:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
Moves: ['Down', 'Up']

Current state:
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
Moves: ['Down', 'Left']

Current state:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Moves: ['Down', 'Right']

Goal found at depth: 2
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Moves: ['Down', 'Right']
```

B) N Queens Problem

Code:

```
import random


def calculate_cost(board):
    n = len(board)
    attacks = 0

    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j]:  # Same column
                attacks += 1
            if abs(board[i] - board[j]) == abs(i - j):  # Same diagonal
                attacks += 1

    return attacks


def get_neighbors(board):
```

```
    neighbors = []
    n = len(board)

    for col in range(n):
        for row in range(n):
            if row != board[col]:  # Only change the row of the queen
                new_board = board[:]
                new_board[col] = row
                neighbors.append(new_board)

    return neighbors

def hill_climb(board):
    current_cost = calculate_cost(board)
    print("Initial board configuration:")
    print_board(board, current_cost)

    iteration = 0
    while True:
        neighbors = get_neighbors(board)
        best_neighbor = None
        best_cost = current_cost

        for neighbor in neighbors:
            cost = calculate_cost(neighbor)
            if cost < best_cost:  # Looking for a lower cost
                best_cost = cost
                best_neighbor = neighbor

        if best_neighbor is None:  # No better neighbor found, we're done
            break
```

```
        board = best_neighbor
        current_cost = best_cost
        iteration += 1
        print(f"Iteration {iteration}:")
        print_board(board, current_cost)

    return board, current_cost

def print_board(board, cost):
    n = len(board)
    # Create an empty board
    display_board = [['.'] * n for _ in range(n)]

    # Place queens on the board
    for col in range(n):
        display_board[board[col]][col] = 'Q'

    # Print the board
    for row in range(n):
        print(' '.join(display_board[row]))
    print(f"Cost: {cost}\n")

if __name__ == "__main__":
    n = int(input("Enter the number of queens (N): "))  # User input for N
    initial_state = list(map(int, input(f"Enter the initial state (row numbers for each column, space-separated): ").split()))

    if len(initial_state) != n or any(r < 0 or r >= n for r in initial_state):
        print("Invalid initial state. Please ensure it has N elements with values from 0 to N-1.")
    else:
```

```
solution, cost = hill_climb(initial_state)

print(f"Final board configuration with cost {cost}:")

print_board(solution, cost)
```

OUTPUT:

```
Enter the number of queens (N): 5
Enter the initial state (row numbers for each column, space-separated): 1 0 1 0 1
Initial board configuration:
. Q . Q .
Q . Q . Q
. . . . .
. . . . .
. . . . .
Cost: 8

Iteration 1:
. Q . Q .
Q . . . Q
. . Q . .
. . . . .
. . . . .
Cost: 4

Iteration 2:
. Q . Q .
. . . . Q
Q . Q . .
. . . . .
. . . . .
Cost: 3
```

```
Iteration 3:
. Q . . .
. . . . Q
Q . Q . .
. . . . .
. . . Q .
Cost: 1

Iteration 4:
. Q . . .
. . . . Q
. . Q . .
Q . . . .
. . . Q .
Cost: 0

Final board configuration with cost 0:
. Q . . .
. . . . Q
. . Q . .
Q . . . .
. . . Q .
Cost: 0
```