AI LAB

PROGRAM 1- Implement Tic –Tac –Toe Game.

Code with output

```
#progrm tic tac toe
board = {1: ' ', 2: ' ', 3: ' ',
         4: ' ', 5: ' ', 6: ' ',
         7: ' ', 8: ' ', 9: ' '}
print("tamanna rukhaya ,1bm22cs301")
def printBoard(board):
    print(board[1] + '|' + board[2] + '|' + board[3])
    print('-+-+-')
    print(board[4] + '|' + board[5] + '|' + board[6])
    print('-+-+-')
    print(board[7] + '|' + board[8] + '|' + board[9])
    print('\n')

def spaceFree(pos):
    return board[pos] == ' '

def checkWin():
    winning_combinations = [
        (1, 2, 3), (4, 5, 6), (7, 8, 9),  # Rows
        (1, 4, 7), (2, 5, 8), (3, 6, 9),  # Columns
        (1, 5, 9), (3, 5, 7)              # Diagonals
    ]
    for combo in winning_combinations:
        if board[combo[0]] == board[combo[1]] == board[combo[2]] != ' ':
            return True
    return False

def checkDraw():
    return all(board[key] != ' ' for key in board.keys())

def insertLetter(letter, position):
    if spaceFree(position):
        board[position] = letter
        printBoard(board)

        if checkDraw():
            print('Draw!')
        elif checkWin():
            print(f'{letter} wins!')
        return
```

```python
        print('Position taken, please pick a different position.')
        playerMove()

def playerMove():
    while True:
        try:
            position = int(input('Enter position for O (1-9): '))
            if position < 1 or position > 9:
                print("Invalid input. Please choose a position from 1 to 9.")
            else:
                insertLetter(player, position)
                break
        except ValueError:
            print("Invalid input. Please enter a number.")

def compMove():
    bestScore = -1000
    bestMove = 0
    for key in board.keys():
        if spaceFree(key):
            board[key] = bot
            score = minimax(board, False)
            board[key] = ' '
            if score > bestScore:
                bestScore = score
                bestMove = key
    insertLetter(bot, bestMove)

def minimax(board, isMaximizing):
    if checkWin():
        return 1 if isMaximizing else -1
    if checkDraw():
        return 0

    if isMaximizing:
        bestScore = -1000
        for key in board.keys():
            if spaceFree(key):
                board[key] = bot
                score = minimax(board, False)
                board[key] = ' '
                bestScore = max(bestScore, score)
        return bestScore
    else:
        bestScore = 1000
        for key in board.keys():
            if spaceFree(key):
                board[key] = player
```

```python
            score = minimax(board, True)
            board[key] = ' '
            bestScore = min(bestScore, score)
        return bestScore

# Main game loop
player = 'O'
bot = 'X'

print("Welcome to Tic-Tac-Toe!")
printBoard(board)

while not checkWin() and not checkDraw():
    playerMove()
    if not checkWin() and not checkDraw():
        compMove()

print("Game over!")
```

```
tamanna rukhaya ,1bm22cs301
Welcome to Tic-Tac-Toe!
 | | 
-+-+-
 | | 
-+-+-
 | | 


Enter position for O (1-9): 3
 | |O
-+-+-
 | | 
-+-+-
 | | 


 |X|O
-+-+-
 | | 
-+-+-
 | | 


Enter position for O (1-9): 6
 |X|O
-+-+-
 | |O
-+-+-
 | | 


X|X|O
-+-+-
 | |O
-+-+-
 | | 


Enter position for O (1-9): 9
X|X|O
-+-+-
 | |O
-+-+-
 | |O


O wins!
Game over!
```

PROGRAM 2- Implement vacuum cleaner agent.

Code with output

```
# program vaccum cleaner
def vacuum_world():

  goal_state = {'A': '0', 'B': '0'}
  cost = 0


  location_input = input("Enter location of vacuum (A/B): ").strip().upper()
  status_input = input(f"Enter status of {location_input} (0 for Clean, 1 for Dirty): ").strip()


  other_room = 'B' if location_input == 'A' else 'A'
  status_input_complement = input(f"Enter status of {other_room} (0 for Clean, 1 for Dirty): ").strip()

  print("Initial Location Condition:", goal_state)

  def clean_room(room):
    nonlocal cost
    goal_state[room] = '0'
    cost += 1
    print(f"Location {room} has been cleaned. Cost: {cost}")

  if location_input == 'A':
    if status_input == '1':
      print("Vacuum is placed in Location A.")
      print("Location A is Dirty.")
      clean_room('A')

    if status_input_complement == '1':
      print("Moving right to Location B.")
      cost += 1
      print(f"COST for moving RIGHT: {cost}")
      clean_room('B')
    else:
      print("Location B is already clean.")

  elif location_input == 'B':
    print("Vacuum is placed in Location B.")
    if status_input == '1':
      print("Location B is Dirty.")
      clean_room('B')

    if status_input_complement == '1':
      print("Moving left to Location A.")
      cost += 1
      print(f"COST for moving LEFT: {cost}")
```

```
      clean_room('A')
    else:
      print("Location A is already clean.")
  else:
    print("Invalid location input. Please enter A or B.")


  print("GOAL STATE:", goal_state)
  print("Performance Measurement: ", cost)
  print("tamanna rukhaya ,1bm22cs301")


vacuum_world()
```

```
Enter location of vacuum (A/B): a
Enter status of A (0 for Clean, 1 for Dirty): 1
Enter status of B (0 for Clean, 1 for Dirty): 0
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in Location A.
Location A is Dirty.
Location A has been cleaned. Cost: 1
Location B is already clean.
GOAL STATE: {'A': '0', 'B': '0'}
Performance Measurement:  1
tamanna rukhaya ,1bm22cs301
```

PROGRAM 3- Solve 8 puzzle problem using BFS and DFS.

Using BFS

Code with output

```
from collections import deque
print ("tamanna ,1BM22CS301")


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()
```

```
tamanna ,1BM22CS301
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, 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]
```

Using DFS

Code with output

```python
from collections import deque
print("tamanna ,1BM22CS301")


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.")


''
```

```
tamanna ,1BM22CS301
Enter the initial state of the puzzle:
Enter row 1 (space-separated numbers, use 0 for empty space): 1 3 5
Enter row 2 (space-separated numbers, use 0 for empty space): 2  6  8
Enter row 3 (space-separated numbers, use 0 for empty space): 0 7 4
Enter the goal state of the puzzle:
Enter row 1 (space-separated numbers, use 0 for empty space): 1 2 3
Enter row 2 (space-separated numbers, use 0 for empty space): 4 5 6
Enter row 3 (space-separated numbers, use 0 for empty space): 7  8 0
This puzzle is unsolvable.
''
```

PROGRAM 4- Solve 8 puzzle problem using A* algorithm

Using misplaced tiles

Code with output

```
import heapq
print("tamanna ,1BM22CS301")


# Define the goal state for the 8-puzzle
GOAL_STATE = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 0]
]


# Define the position moves (up, down, left, right)
MOVES = [
  (-1, 0),  # Up
  (1, 0),   # Down
  (0, -1),  # Left
  (0, 1)    # Right
]


class PuzzleNode:
  def __init__(self, state, parent=None, g=0, h=0):
    self.state = state
    self.parent = parent
    self.g = g  # Cost from start to current node
    self.h = h  # Heuristic cost to goal
    self.f = g + h  # Total cost


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

```
def misplaced_tiles(state):
    """Heuristic function that counts the number of misplaced tiles."""
    misplaced = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != GOAL_STATE[i][j]:
                misplaced += 1
    return misplaced


def get_zero_position(state):
    """Find the position of the zero (empty tile) in the puzzle."""
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None


def generate_successors(node):
    """Generate successors by moving the empty tile in all possible directions."""
    successors = []
    zero_x, zero_y = get_zero_position(node.state)


    for move_x, move_y in MOVES:
        new_x, new_y = zero_x + move_x, zero_y + move_y
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in node.state]
            new_state[zero_x][zero_y], new_state[new_x][new_y] = new_state[new_x][new_y],
new_state[zero_x][zero_y]
            h = misplaced_tiles(new_state)
            successors.append(PuzzleNode(new_state, parent=node, g=node.g + 1, h=h))
    return successors


def is_goal(state):
    """Check if the current state is the goal state."""
    return state == GOAL_STATE


def reconstruct_path(node):
    """Reconstruct the path from the start state to the goal state."""
    path = []
    while node:
        path.append(node.state)
        node = node.parent
```

```python
    return path[::-1]


def a_star(start_state):
    """A* algorithm to solve the 8-puzzle problem."""
    start_node = PuzzleNode(start_state, g=0, h=misplaced_tiles(start_state))
    open_list = []
    closed_set = set()


    heapq.heappush(open_list, start_node)


    while open_list:
        current_node = heapq.heappop(open_list)


        if is_goal(current_node.state):
            return reconstruct_path(current_node)


        closed_set.add(tuple(map(tuple, current_node.state)))


        for successor in generate_successors(current_node):
            if tuple(map(tuple, successor.state)) in closed_set:
                continue
            heapq.heappush(open_list, successor)


    return None


def get_user_input():
    """Get a valid 8-puzzle input state from the user."""
    print("Enter your 8-puzzle configuration (0 represents the empty tile):")
    state = []
    values = set()


    for i in range(3):
        row = input(f"Enter row {i+1} (space-separated numbers between 0 and 8): ").split()
        if len(row) != 3:
            print("Each row must have exactly 3 numbers. Please try again.")
            return None


        row = [int(x) for x in row]
```

```
        if not all(0 <= x <= 8 for x in row):
            print("Values must be between 0 and 8. Please try again.")
            return None


        state.append(row)
        values.update(row)


    if values != set(range(9)):
        print("All numbers from 0 to 8 must be present exactly once. Please try again.")
        return None


    return state


# Main function
def main():
    start_state = None
    while start_state is None:
        start_state = get_user_input()


    solution = a_star(start_state)


    # Print the solution steps
    if solution:
        print("Solution found in", len(solution) - 1, "moves:")
        for step in solution:
            for row in step:
                print(row)
            print()
    else:
        print("No solution found.")


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

```
tamanna ,1BM22CS301
Enter your 8-puzzle configuration (0 represents the empty tile):
Enter row 1 (space-separated numbers between 0 and 8): 1 2 3
Enter row 2 (space-separated numbers between 0 and 8): 4 5 6
Enter row 3 (space-separated numbers between 0 and 8): 7 0 8
Solution found in 1 moves:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

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

Using Manhattan distance

Code with output

```python
# manhattan distance
import heapq
print("tamanna ,1BM22CS301")


# Define the goal state for the 8-puzzle
GOAL_STATE = [
    [2, 8, 1],
    [0, 4, 3],
    [7, 6, 5]



]


# Define the position moves (up, down, left, right)
MOVES = [
   (-1, 0),  # Up
   (1, 0),   # Down
   (0, -1),  # Left
   (0, 1)    # Right
]


class PuzzleNode:
   def __init__(self, state, parent=None, g=0, h=0):
      self.state = state
      self.parent = parent
      self.g = g  # Cost from start to current node
      self.h = h  # Heuristic cost to goal (Manhattan distance)
      self.f = g + h  # Total cost
```

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


def manhattan_distance(state):
    """Heuristic function that calculates the Manhattan distance for each tile."""
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:  # Skip the empty tile
                # Calculate goal position for this value
                goal_x, goal_y = (value - 1) // 3, (value - 1) % 3
                # Add the Manhattan distance for this tile
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance


def get_zero_position(state):
    """Find the position of the zero (empty tile) in the puzzle."""
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None


def generate_successors(node):
    """Generate successors by moving the empty tile in all possible directions."""
    successors = []
    zero_x, zero_y = get_zero_position(node.state)


    for move_x, move_y in MOVES:
        new_x, new_y = zero_x + move_x, zero_y + move_y
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in node.state]
            new_state[zero_x][zero_y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[zero_x][zero_y]
            h = manhattan_distance(new_state)
            successors.append(PuzzleNode(new_state, parent=node, g=node.g + 1, h=h))
    return successors


def is_goal(state):
    """Check if the current state is the goal state."""
    return state == GOAL_STATE
```

```python
def reconstruct_path(node):
    """Reconstruct the path from the start state to the goal state."""
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return path[::-1]


def a_star(start_state):
    """A* algorithm to solve the 8-puzzle problem."""
    start_node = PuzzleNode(start_state, g=0, h=manhattan_distance(start_state))
    open_list = []
    closed_set = set()


    heapq.heappush(open_list, start_node)


    while open_list:
        current_node = heapq.heappop(open_list)


        if is_goal(current_node.state):
            return reconstruct_path(current_node)


        closed_set.add(tuple(map(tuple, current_node.state)))


        for successor in generate_successors(current_node):
            if tuple(map(tuple, successor.state)) in closed_set:
                continue
            heapq.heappush(open_list, successor)


    return None


def get_user_input():
    """Get a valid 8-puzzle input state from the user."""
    print("Enter your 8-puzzle configuration (0 represents the empty tile):")
    state = []
    values = set()


    for i in range(3):
        row = input(f"Enter row {i+1} (space-separated numbers between 0 and 8): ").split()
```

```python
        if len(row) != 3:
            print("Each row must have exactly 3 numbers. Please try again.")
            return None

        row = [int(x) for x in row]

        if not all(0 <= x <= 8 for x in row):
            print("Values must be between 0 and 8. Please try again.")
            return None

        state.append(row)
        values.update(row)

    if values != set(range(9)):
        print("All numbers from 0 to 8 must be present exactly once. Please try again.")
        return None

    return state

# Main function
def main():
    start_state = None
    while start_state is None:
        start_state = get_user_input()

    solution = a_star(start_state)

    # Print the solution steps
    if solution:
        print("Solution found in", len(solution) - 1, "moves:")
        for step in solution:
            for row in step:
                print(row)
            print()
    else:
        print("No solution found.")

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

```
tamanna ,1BM22CS301
Enter your 8-puzzle configuration (0 represents the empty tile):
Enter row 1 (space-separated numbers between 0 and 8): 1 3 2
Enter row 2 (space-separated numbers between 0 and 8): 4 6
Each row must have exactly 3 numbers. Please try again.
Enter your 8-puzzle configuration (0 represents the empty tile):
Enter row 1 (space-separated numbers between 0 and 8): 2 8 1
Enter row 2 (space-separated numbers between 0 and 8): 0 4 3
Enter row 3 (space-separated numbers between 0 and 8): 6 5 7
Solution found in 14 moves:
[2, 8, 1]
[0, 4, 3]
[6, 5, 7]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

PROGRAM 5- N queens using Hill climbing algorithm

Code with output

```
from random import randint


# Function to print the board
def printBoard(board, N):
  for i in range(N):
    print(" ".join(map(str, board[i])))
  print("-" * (2 * N - 1))


# Function to calculate the objective value (number of attacking queens)
def calculateObjective(board, state, N):
  attacking = 0
  for i in range(N):
    row = state[i]

    # Check row conflicts (queens can't share the same row)
    for j in range(i + 1, N):
      if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
        attacking += 1
  return attacking


# Function to generate the board from the state (which is an array of row positions)
def generateBoard(board, state, N):
  for i in range(N):
    for j in range(N):
      board[i][j] = 0
  for i in range(N):
    board[state[i]][i] = 1


# Function to get the best neighbor (state with the minimum conflicts)
def getNeighbour(board, state, N):
  opState = state[:]
  opBoard = [[0] * N for _ in range(N)]
  generateBoard(opBoard, opState, N)
  opObjective = calculateObjective(opBoard, opState, N)

  # Try moving each queen to a different row and check if it improves the objective
  for i in range(N):
    original_row = state[i]
    for new_row in range(N):
      if new_row != original_row:
        state[i] = new_row
```

```python
            generateBoard(board, state, N)
            tempObjective = calculateObjective(board, state, N)
            if tempObjective < opObjective:
                opObjective = tempObjective
                opState = state[:]
        state[i] = original_row
    generateBoard(board, opState, N)
    return opState, opObjective


# Hill climbing algorithm
def hillClimbing(N, initial_state):
    board = [[0] * N for _ in range(N)]
    state = initial_state[:]
    generateBoard(board, state, N)

    iteration = 0
    while True:
        print(f"Iteration {iteration}:")
        print(f"Current State: {state}")
        currentObjective = calculateObjective(board, state, N)
        print(f"Objective Value: {currentObjective}")
        printBoard(board, N)

        nextState, nextObjective = getNeighbour(board, state, N)

        # Break if we reach an optimal solution (no conflicts)
        if nextObjective == 0:
            print("Final Solution:")
            printBoard(board, N)
            break

        # If stuck in a local optimum, pick a random neighboring state
        if nextObjective >= currentObjective:
            print("Stuck in local optimum. Jumping to a random neighbor...")
            # Randomly pick a new position for a queen in a column
            state[randint(0, N - 1)] = randint(0, N - 1)
            generateBoard(board, state, N)
        else:
            state = nextState  # Move to the next better state

        iteration += 1


# Main code to accept user input
if __name__ == "__main__":
    print("tamanna, 1bm22cs301")  # Print your name

    N = int(input("Enter the size of the board (e.g., 8 for 8-Queens problem): "))
```

```python
print(f"Enter the initial positions of queens for each column (0 to {N-1}):")
initial_state = list(map(int, input().split()))

# Validate the input (ensure it has the correct size and values)
if len(initial_state) != N or any(pos < 0 or pos >= N for pos in initial_state):
    print("Invalid input. Please ensure each queen position is within the board size.")
else:
    hillClimbing(N, initial_state)
```

```
tamanna, 1bm22cs301
Enter the size of the board (e.g., 8 for 8-Queens problem): 8
Enter the initial positions of queens for each column (0 to 7): 0 4 7 5 2 6 3 1
Iteration 0:
Current State: [0, 4, 7, 5, 2, 6, 3, 1]
Objective Value: 8
1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0
```

```
Stuck in local optimum. Jumping to a random neighbor...
Iteration 1:
Current State: [0, 4, 7, 5, 2, 6, 3, 1]
Objective Value: 8
...
Iteration 2:
Current State: [1, 4, 7, 5, 2, 6, 3, 1]
Objective Value: 8
...
Iteration 3:
Current State: [1, 3, 7, 5, 2, 6, 3, 1]
Objective Value: 7
...
Iteration 4:
Current State: [1, 3, 7, 5, 2, 6, 3, 1]
Objective Value: 5
...
Final Solution:
1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0
```

PROGRAM 6- N queens using Simulated Annealing

Code with output

```
import random
import math


def create_board_from_input(n, positions):
  """Create a board based on user input positions for each column."""
  return positions
```

```python
def calculate_conflicts(board):
    """Calculate the number of conflicts on the board."""
    n = len(board)
    conflicts = 0
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                conflicts += 1
    return conflicts


def get_neighbors(board):
    """Generate all neighboring boards by changing the position of one queen."""
    n = len(board)
    neighbors = []
    for i in range(n):
        for j in range(n):
            if board[i] != j:
                neighbor = list(board)
                neighbor[i] = j
                neighbors.append(neighbor)
    return neighbors


def simulated_annealing(n, initial_temperature, cooling_rate, initial_state):
    """Perform simulated annealing to solve the n-queens problem."""
    current_board = initial_state
    current_conflicts = calculate_conflicts(current_board)
    best_board = list(current_board)
    best_conflicts = current_conflicts
    temperature = initial_temperature
    iterations = 0  # Counter for iterations


    while temperature > 1:
        iterations += 1  # Increment the iteration count
        neighbors = get_neighbors(current_board)
        neighbor = random.choice(neighbors)
        neighbor_conflicts = calculate_conflicts(neighbor)


        delta = neighbor_conflicts - current_conflicts


        if delta < 0 or random.random() < math.exp(-delta / temperature):
            current_board = neighbor
            current_conflicts = neighbor_conflicts
```

```
        if current_conflicts < best_conflicts:
            best_board = list(current_board)
            best_conflicts = current_conflicts


        temperature *= cooling_rate


    return best_board, best_conflicts, iterations


# Main code to accept user input
if __name__ == "__main__":
    n = int(input("Enter the size of the board (number of queens): "))
    initial_temperature = float(input("Enter the initial temperature (e.g., 100): "))
    cooling_rate = float(input("Enter the cooling rate (e.g., 0.99): "))


    print(f"Enter the initial positions of queens for each column (values between 0 and {n-1}, one value per
column):")
    initial_state = []
    for i in range(n):
        pos = int(input(f"Position for column {i + 1} (0 to {n-1}): "))
        if 0 <= pos < n:
            initial_state.append(pos)
        else:
            print(f"Invalid input for column {i + 1}, please enter a number between 0 and {n-1}.")
            break
    else:
        solution, conflicts, iterations = simulated_annealing(n, initial_temperature, cooling_rate,
initial_state)


        print("\nSolution:")
        for i in range(n):
            line = ""
            for j in range(n):
                if j == solution[i]:
                    line += "Q "
                else:
                    line += ". "
            print(line)


        print("\nConflicts:", conflicts)
        print(f"Iterations: {iterations}")
```

```
Enter the size of the board (number of queens): 4
Enter the initial temperature (e.g., 100): 50
Enter the cooling rate (e.g., 0.99): 0.22
Enter the initial positions of queens for each column (values between 0 and 3, one value per column):
Position for column 1 (0 to 3): 2
Position for column 2 (0 to 3): 1
Position for column 3 (0 to 3): 3
Position for column 4 (0 to 3): 0

Solution:
. . Q .
. Q . .
. . . Q
Q . . .

Conflicts: 1
Iterations: 3
```

PROGRAM 7-Unification in First Order Logic

Code with output

```python
#program unification in First Order Logic

def unify(x1, x2):
    """
    Unify two expressions (x1 and x2) based on the given unification algorithm.
    Returns a substitution set (SUBST) or FAILURE if unification is not possible.
    """
    if is_variable_or_constant(x1) or is_variable_or_constant(x2):
        if x1 == x2:
            return []
        elif is_variable(x1):
            if occurs_check(x1, x2):
                return "FAILURE"
            else:
                return [(x2, x1)]
        elif is_variable(x2):
            if occurs_check(x2, x1):
                return "FAILURE"
            else:
                return [(x1, x2)]
        else:
            return "FAILURE"

    if not is_same_predicate(x1, x2):
        return "FAILURE"

    if len(x1) != len(x2):
        return "FAILURE"
```

```python
    subst = []

    for i in range(len(x1)):
        s = unify(x1[i], x2[i])
        if s == "FAILURE":
            return "FAILURE"
        elif s:
            subst.extend(s)
            apply_substitution(s, x1[i+1:])
            apply_substitution(s, x2[i+1:])

    return subst

def is_variable_or_constant(expr):
    """Check if the expression is a variable or a constant."""
    return isinstance(expr, str) and expr.isalnum()

def is_variable(expr):
    """Check if the expression is a variable."""
    return isinstance(expr, str) and expr.islower()

def occurs_check(var, expr):
    """Check if the variable occurs in the expression."""
    if var == expr:
        return True
    elif isinstance(expr, (list, tuple)):
        return any(occurs_check(var, sub_expr) for sub_expr in expr)
    return False

def is_same_predicate(x1, x2):
    """Check if the initial predicate symbols of x1 and x2 are the same."""
    if isinstance(x1, (list, tuple)) and isinstance(x2, (list, tuple)):
        return x1[0] == x2[0]
    return False

def apply_substitution(subst, expr):
    """Apply the substitution set to the given expression."""
    for old, new in subst:
        if expr == old:
            return new
        elif isinstance(expr, (list, tuple)):
            return [apply_substitution(subst, sub_expr) for sub_expr in expr]
    return expr

def parse_input(expr):
    """Parse user input into a list or tuple representing the predicate."""
    try:
        return eval(expr)
```

```python
    except Exception as e:
        print(f"Error in input format: {e}")
        return None

print("Enter two expressions to unify. Use list/tuple format.")
print("Example: ['P', 'x', 'a'] represents P(x, a)")

expr1_input = input("Enter the first expression: ")
expr2_input = input("Enter the second expression: ")

expr1 = parse_input(expr1_input)
expr2 = parse_input(expr2_input)

if expr1 is not None and expr2 is not None:
    result = unify(expr1, expr2)
    print("Unification Result:", result)
else:
    print("Invalid input format. Please try again.")

print("tamanna  - 1BM22CS301")
```

```
Enter two expressions to unify. Use list/tuple format.
Example: ['P', 'x', 'a'] represents P(x, a)
Enter the first expression: 'p' , 'x', 'a'
Enter the second expression: 'p', 'b', 'a'
Unification Result: [('b', 'x')]
tamanna  - 1BM22CS301
```