AI LAB PROGRAMS

DEVANSHI SLATHIA

1BM22CS083

TIC-TAC-TOE Problem

```
import random

# Tic Tac Toe board
board = [' ' for _ in range(9)]

# Function to print the board
def print_board():
    for row in [board[i*3:(i+1)*3] for i in range(3)]:
        print('| ' + ' | '.join(row) + ' |')

# Function to check if a player has won
def check_winner(board, player):
    win_conditions = [(0, 1, 2), (3, 4, 5), (6, 7, 8),  # Horizontal
                      (0, 3, 6), (1, 4, 7), (2, 5, 8),  # Vertical
                      (0, 4, 8), (2, 4, 6)]  # Diagonal
    return any(board[a] == board[b] == board[c] == player for a, b, c in win_conditions)

# Function to check if the board is full
def is_board_full(board):
    return ' ' not in board

# Minimax algorithm for AI moves
def minimax(board, depth, is_maximizing):
    if check_winner(board, 'O'):
        return 1  # AI wins
    if check_winner(board, 'X'):
        return -1  # Player wins
```

```
    if is_board_full(board):
        return 0  # Tie

    if is_maximizing:
        best_score = -float('inf')
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'O'
                score = minimax(board, depth + 1, False)
                board[i] = ' '
                best_score = max(score, best_score)
        return best_score
    else:
        best_score = float('inf')
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'X'
                score = minimax(board, depth + 1, True)
                board[i] = ' '
                best_score = min(score, best_score)
        return best_score

# Function for AI to make its move
def ai_move():
    best_score = -float('inf')
    move = -1
    for i in range(9):
        if board[i] == ' ':
            board[i] = 'O'
            score = minimax(board, 0, False)
            board[i] = ' '
            if score > best_score:
                best_score = score
```

```
            move = i
    board[move] = 'O'

# Function for player to make a move
def player_move():
    while True:
        move = int(input("Enter your move (1-9): ")) - 1
        if board[move] == ' ':
            board[move] = 'X'
            break
        else:
            print("Invalid move. Try again.")

# Main game loop
def play_game():
    print("Welcome to Tic-Tac-Toe!")
    print_board()

    while True:
        # Player's move
        player_move()
        print_board()
        if check_winner(board, 'X'):
            print("You win!")
            break
        if is_board_full(board):
            print("It's a tie!")
            break

        # AI's move
        ai_move()
        print_board()
        if check_winner(board, 'O'):
```

```
            print("AI wins!")
            break
        if is_board_full(board):
            print("It's a tie!")
            break

play_game()
```

OUTPUT

```
Welcome to Tic-Tac-Toe!
|   |   |   |
|   |   |   |
|   |   |   |
Enter your move (1-9):  1
| X |   |   |
|   |   |   |
|   |   |   |
| X |   |   |
|   | O |   |
|   |   |   |
Enter your move (1-9):  7
| X |   |   |
|   | O |   |
| X |   |   |
| X |   |   |
| O | O |   |
| X |   |   |
```

```
Enter your move (1-9):  7
| X |   |   |
|   | O |   |
| X |   |   |
| X |   |   |
| O | O |   |
| X |   |   |
Enter your move (1-9):  6
| X |   |   |
| O | O | X |
| X |   |   |
| X | O |   |
| O | O | X |
| X |   |   |
Enter your move (1-9):  8
| X | O |   |
| O | O | X |
| X | X |   |
| X | O |   |
| O | O | X |
| X | X | O |
Enter your move (1-9):  3
| X | O | X |
| O | O | X |
| X | X | O |
It's a tie!
```

VACCUM CLEANER PROBLEM

```
def vacuum():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter Location of Vacuum (A or B): ").strip().upper()
    status_input = input(f"Enter status of {location_input} (0 for clean, 1 for dirty): ").strip()
    status_input_complement = input("Enter status of the other room (0 for clean, 1 for dirty): ").strip()


    print("Initial Location Condition: " + str(goal_state))



    if location_input == 'A':
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")
            goal_state['A'] = '0'
```

```
            cost += 1
            print("Cost for CLEANING A: " + str(cost))
            print("Location A has been Cleaned.")

        if status_input_complement == '1':
            print("Location B is Dirty.")
            print("Moving right to Location B.")
            cost += 1
            print("Cost for moving RIGHT: " + str(cost))
            goal_state['B'] = '0'
            cost += 1
            print("Cost for SUCK: " + str(cost))
            print("Location B has been Cleaned.")
        else:
            print("No action. 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.")
            goal_state['B'] = '0'
            cost += 1
            print("Cost for CLEANING B: " + str(cost))
            print("Location B has been Cleaned.")

        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving LEFT to Location A.")
            cost += 1
            print("Cost for moving LEFT: " + str(cost))
            goal_state['A'] = '0'
            cost += 1
```

```
        print("Cost for SUCK: " + str(cost))
        print("Location A has been Cleaned.")
    else:
        print("No action. Location A is already clean.")


print("GOAL STATE: " + str(goal_state))
print("Performance Measurement: " + str(cost))


vacuum()
```

OUTPUT

```
Enter Location of Vacuum (A or B):  A
Enter status of A (0 for clean, 1 for dirty):  0
Enter status of the other room (0 for clean, 1 for dirty):
1
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location B is Dirty.
Moving right to Location B.
Cost for moving RIGHT: 1
Cost for SUCK: 2
Location B has been Cleaned.
GOAL STATE: {'A': '0', 'B': '0'}
Performance Measurement: 2
```

---

8 SQUARE PUZZLE

```
from collections import deque


class PuzzleState:
    def __init__(self, board, zero_position, moves):
        self.board = board
        self.zero_position = zero_position
        self.moves = moves

    def get_possible_moves(self):
        moves = []
        row, col = self.zero_position
```

```
        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:
                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]
                moves.append((new_board, (new_row, new_col)))
        return moves

def bfs(start_board):
    initial_zero_position = start_board.index(0)
    initial_state = PuzzleState(start_board, (initial_zero_position // 3, initial_zero_position % 3), [])

    goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
    visited = set()
    queue = deque([initial_state])

    while queue:
        current_state = queue.popleft()

        if current_state.board == goal_state:
            return current_state.moves

        visited.add(tuple(current_state.board))

        for new_board, new_zero_position in current_state.get_possible_moves():
            if tuple(new_board) not in visited:
                queue.append(PuzzleState(new_board, new_zero_position, current_state.moves + [new_board]))

    return None

def print_board(board):
```

```
    for i in range(3):
        print(board[i*3:(i+1)*3])

def main():
    print("Enter the initial board state (0 for the empty space) as a list of 9 numbers:")
    print("For example: [1, 2, 3, 4, 5, 6, 0, 7, 8]")
    user_input = input()

    try:
        start_board = list(map(int, user_input.strip('[]').split(',')))
        if len(start_board) != 9 or set(start_board) != set(range(9)):
            raise ValueError("Invalid input. Please enter 9 unique numbers from 0 to 8.")

        print("Initial board:")
        print_board(start_board)

        solution = bfs(start_board)

        if solution is not None:
            print("\nSolution found! Steps to reach the goal state:")
            for step in solution:
                print_board(step)
                print()
        else:
            print("\nNo solution found for the given board.")

    except Exception as e:
        print("Error:", e)

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

OUTPUT

```
Enter the initial board state (0 for the empty space) as a
list of 9 numbers:
For example: [1, 2, 3, 4, 5, 6, 0, 7, 8]
 1,4,5,3,2,0,6,8,7
Initial board:
[1, 4, 5]
[3, 2, 0]
[6, 8, 7]

Solution found! Steps to reach the goal state:
[1, 4, 0]
[3, 2, 5]
[6, 8, 7]


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

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


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

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


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

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


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

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

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

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

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

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

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

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

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

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

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

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

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

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