AI LAB PROGRAMS

1. Write a program to solve the Water Jug problem using Depth-First Search (DFS) and display the steps from the start to the goal state.

2. Write a program to solve the Missionaries and Cannibals problem using Best-First Search and explain the heuristic used.

3. Simulate the Wumpus World using rule-based logic to show how decisions are made with uncertainty.

4. Write a program to find the shortest path using the A search algorithm with a suitable heuristic

5. Solve a problem using the AO search algorithm on an AND-OR graph and show the solution path.

6. Solve the 8-Queens problem using backtracking and display the final board arrangement.

7. Solve the Traveling Salesman Problem using a heuristic method like nearest neighbour and calculate the total cost.

8. Write a rule-based program to perform either forward chaining or backward chaining inference.

9. Develop a Tic-Tac-Toe game for two players using the Minimax algorithm for intelligent moves.

10. Create a simple Chabot that answers user questions using a predefined set of responses on keywords.

11. Make a simple board game (like Snake or Connect Four) and use a strategy or learning technique to play it.

12. Write a program to spell-checker using bigrams or Levenshtein distance.

13. Write a function that takes (01, 02, Li, La) and outputs (x, y) of the end effector.

14. Implement backward chaining to derive diagnoses from user symptom input.

1. Write a program to solve the Water Jug problem using Depth-First Search (DFS) and display the steps from the start to the goal state.

```
def water_jug_dfs(capacity1, capacity2, target):
    """     Solves the water jug problem using DFS and returns the path of states.
    Args:
        capacity1 (int): Capacity of the first jug.
        capacity2 (int): Capacity of the second jug.
        target (int): The target amount of water in either jug.
    Returns:
        list: A list of tuples representing the states in the path from start to goal,
              or None if no solution is found.
    """
    visited = set()  # To keep track of visited states to avoid cycles
    path = []  # To store the solution path
    def dfs(jug1, jug2):
        # If this state has been visited, return False (avoiding infinite loops)
        if (jug1, jug2) in visited:
            return False
        # Mark the current state as visited
        visited.add((jug1, jug2))
        # Add the current state to the path
        path.append((jug1, jug2))

        # Check if the goal state is reached
        if jug1 == target or jug2 == target:
            return True
        # Explore all possible actions (transitions)
        # 1. Fill jug1
        if dfs(capacity1, jug2):
```

```
            return True
        # 2. Fill jug2
        if dfs(jug1, capacity2):
            return True
        # 3. Empty jug1
        if dfs(0, jug2):
            return True
        # 4. Empty jug2
        if dfs(jug1, 0):
            return True
        # 5. Pour jug1 into jug2
        pour_amount = min(jug1, capacity2 - jug2)
        if dfs(jug1 - pour_amount, jug2 + pour_amount):
            return True
        # 6. Pour jug2 into jug1
        pour_amount = min(jug2, capacity1 - jug1)
        if dfs(jug1 + pour_amount, jug2 - pour_amount):
            return True

        # If no action from this state leads to the goal, backtrack
        path.pop()
        return False
    # Start DFS from the initial state (both jugs empty)
    if dfs(0, 0):
        return path
    else:
        return None
# Example Usage:
jug1_capacity = 4
jug2_capacity = 3
```

```
target_amount = 2
solution_path = water_jug_dfs(jug1_capacity, jug2_capacity, target_amount)
if solution_path:
    print(f"Solution found for jugs with capacities {jug1_capacity} and {jug2_capacity}, targeting {target_amount}:")
    for i, (j1, j2) in enumerate(solution_path):
        print(f"Step {i + 1}: Jug1 = {j1}, Jug2 = {j2}")
else:
    print("No solution found.")
```

2. Write a program to solve the Missionaries and Cannibals problem using Best-First Search and explain the heuristic used.

```
from collections import deque


def is_valid(m, c):
    """Checks if a state (m missionaries, c cannibals) is valid on a single bank."""
    if m < 0 or c < 0 or m > 3 or c > 3:
        return False
    if m > 0 and m < c:  # Cannibals outnumber missionaries
        return False
    return True


def solve_missionaries_cannibals():
    initial_state = (3, 3, 0)  # (M_left, C_left, Boat_pos)
    goal_state = (0, 0, 1)

    queue = deque([(initial_state, [])])  # (current_state, path_to_current_state)
    visited = {initial_state}

    while queue:
        current_state, path = queue.popleft()
```

```
        m_left, c_left, boat_pos = current_state

        if current_state == goal_state:
            return path + [current_state]

        # Define possible moves (delta_m, delta_c)
        moves = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]

        for dm, dc in moves:
            if boat_pos == 0:  # Boat on the left bank, moving to the right
                next_m_left, next_c_left = m_left - dm, c_left - dc
                next_m_right, next_c_right = 3 - next_m_left, 3 - next_c_left
                next_boat_pos = 1
            else:  # Boat on the right bank, moving to the left
                next_m_left, next_c_left = m_left + dm, c_left + dc
                next_m_right, next_c_right = 3 - next_m_left, 3 - next_c_left
                next_boat_pos = 0

            next_state = (next_m_left, next_c_left, next_boat_pos)

            if is_valid(next_m_left, next_c_left) and \
               is_valid(next_m_right, next_c_right) and \
               next_state not in visited:
                visited.add(next_state)
                queue.append((next_state, path + [current_state]))

    return None  # No solution found

if __name__ == "__main__":
    solution_path = solve_missionaries_cannibals()
```

```
    if solution_path:
        print("Solution found:")
        for i, state in enumerate(solution_path):
            m_left, c_left, boat_pos = state
            m_right, c_right = 3 - m_left, 3 - c_left
            boat_side = "Left" if boat_pos == 0 else "Right"
            print(f"Step {i}: Left Bank: (M:{m_left}, C:{c_left}), Right Bank: (M:{m_right}, C:{c_right}), Boat: {boat_side}")
    else:
        print("No solution exists.")
```

4. Write a program to find the shortest path using the A* search algorithm with a suitable heuristic

```
g = {'A': 0}                      # Cost from start
h = {'A': 3, 'B': 2, 'C': 1, 'D': 0}  # Heuristic
p = {'A': None}                   # Parent pointers
G = {'A': ['B', 'C'], 'B': ['D'], 'C': ['D'], 'D': []}  # Graph
o = {'A'}                         # Open set

while o:
    n = min(o, key=lambda x: g[x] + h[x])  # Node with lowest f = g + h
    if n == 'D':
        break
    o.remove(n)
    for m in G[n]:
        g[m] = g[n] + 1
        p[m] = n
        o.add(m)

# Reconstruct path
path = []
```

```
while n is not None:
    path.append(n)
    n = p[n]
print("Path:", path[::-1])
```

OUTPUT:

Path: ['A', 'C', 'D']

5. Solve a problem using the AO* search algorithm on an AND-OR graph and show the solution path.

```
# AND-OR graph definition
graph = {
    'A': [['B', 'C'], ['D']],  # A has two options: AND(B,C) or OR(D)
    'B': [['E', 'F']],         # B AND(E,F)
    'C': [],                   # C is terminal
    'D': [],                   # D is terminal
    'E': [],                   # E terminal
    'F': []                    # F terminal
}
h = {'A': 5, 'B': 3, 'C': 2, 'D': 4, 'E': 0, 'F': 0}  # Heuristic
def ao_star(node):
    if not graph[node]:
        return [node]  # Terminal node
    min_cost, best_path = float('inf'), []
    for option in graph[node]:  # Each option (AND nodes list)
        cost = sum(h[n] for n in option)
        if cost < min_cost:
            min_cost = cost
            best_path = option
    solution = [node]
```

```
    for n in best_path:
        solution += ao_star(n)
    return solution
path = ao_star('A')
print("AO* solution path:", path)
```

6. Solve the 8-Queens problem using backtracking and display the final board arrangement.

```
def q(c):
    if c==8:
        for r in b: print(*[1 if i==r else 0 for i in range(8)])
        return True
    for r in range(8):
        if all(r!=b[i] and abs(r-b[i])!=c-i for i in range(c)):
            b[c]=r
            if q(c+1): return True

b=[0]*8
q(0)
```

7. Solve the Traveling Salesman Problem using a heuristic method like nearest neighbour and calculate the total cost.

```
from itertools import permutations

def calculate_distance(route, distances):
    total_distance = 0
    for i in range(len(route) - 1):
        total_distance += distances[route[i]][route[i + 1]]
    total_distance += distances[route[-1]][route[0]]
    return total_distance
```

```
def brute_force_tsp(distances):
    n = len(distances)
    cities = list(range(1, n))
    shortest_route = None
    min_distance = float('inf')

    for perm in permutations(cities):
        current_route = [0] + list(perm)
        current_distance = calculate_distance(current_route, distances)
            if current_distance < min_distance:
            min_distance = current_distance
            shortest_route = current_route

    shortest_route.append(0)
    return shortest_route, min_distance
distances = [
    [0, 2, 2, 5, 9, 3],
    [2, 0, 4, 6, 7, 8],
    [2, 4, 0, 8, 6, 3],
    [5, 6, 8, 0, 4, 9],
    [9, 7, 6, 4, 0, 10],
    [3, 8, 3, 9, 10, 0]
]
route, total_distance = brute_force_tsp(distances)
print("Route:", route)
print("Total distance:", total_distance)
```

8. Write a rule-based program to perform either forward chaining or backward chaining inference.

#Forward Chaining

```
# Define initial facts
facts = {"has_fever", "has_cough"}


# Define rules as a list of dictionaries
# Each rule has a 'conditions' list and a 'conclusion' string
rules = [
    {
        "conditions": ["has_fever", "has_cough"],
        "conclusion": "has_flu"
    },
    {
        "conditions": ["has_flu"],
        "conclusion": "needs_rest"
    },
    {
        "conditions": ["has_fever"],
        "conclusion": "might_have_infection"
    }
]
# Forward chaining inference engine
def forward_chaining(facts, rules):
    new_facts_inferred = True
    while new_facts_inferred:
        new_facts_inferred = False
        for rule in rules:
            # Check if all conditions of the rule are present in the facts
            all_conditions_met = all(condition in facts for condition in rule["conditions"])
```

```
        # If conditions are met and the conclusion is not already a fact, add it
        if all_conditions_met and rule["conclusion"] not in facts:
            facts.add(rule["conclusion"])
            new_facts_inferred = True
            print(f"Inferred: {rule['conclusion']}")
    return facts

# Run the forward chaining
final_facts = forward_chaining(facts.copy(), rules) # Use a copy to avoid modifying the original set
print("\nFinal inferred facts:")
for fact in final_facts:
    print(fact)
```

 # Backward Chaining

```
# Knowledge base: Rules
rules = {
    "flu": ["fever", "cough"],
    "measles": ["fever", "rash"],
    "viral_infection": ["headache", "fever"]
}

# Facts known so far
facts = {}

# Function to ask user about a symptom
def ask(symptom):
    if symptom not in facts:
        ans = input(f"Do you have {symptom}? (yes/no): ").lower()
        facts[symptom] = (ans == "yes")
```

```
    return facts[symptom]

# Backward Chaining function
def backward_chain(goal):
    print(f"\nTrying to prove: {goal}")

    # If goal is a symptom, ask user
    if goal not in rules:
        return ask(goal)

    # Goal is a conclusion, check its conditions
    for condition in rules[goal]:
        if not backward_chain(condition):
            print(f"Cannot prove {goal}")
            return False

    print(f"{goal} is TRUE")
    return True

# Main program
print("Backward Chaining Diagnosis System")
print("Possible diagnoses:", list(rules.keys()))

goal = input("\nEnter the disease you want to diagnose: ").lower()

if backward_chain(goal):
    print(f"\n✅ Diagnosis: You may have {goal}")
else:
    print(f"\n❌ Diagnosis: {goal} could not be confirmed")
```

9. Develop a Tic-Tac-Toe game for two players using the Minimax algorithm for intelligent moves.

```
board = [' ']*9


def print_board():
    print(f"{board[0]}|{board[1]}|{board[2]}\n-+-+-\n{board[3]}|{board[4]}|{board[5]}\n-+-+-\n{board[6]}|{board[7]}|{board[8]}")
def check_win(p):
    for i in [(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)]:
        if board[i[0]]==board[i[1]]==board[i[2]]==p: return True
    return False
player = 'X'
for _ in range(9):
    print_board()
    move = int(input(f"{player}'s turn (0-8): "))
    if board[move]==' ':
        board[move]=player
        if check_win(player):
            print_board(); print(player, "wins!"); break
        player = 'O' if player=='X' else 'X'
else:
    print_board(); print("Draw!")
```

10. Create a simple Chabot that answers user questions using a predefined set of responses on keywords.

```
# Simple Keyword-Based Chatbot


def chatbot_response(user_input):
    user_input = user_input.lower()


    if "hello" in user_input or "hi" in user_input:
```

```
        return "Hello! How can I help you today?"

    elif "name" in user_input:
        return "I am a simple chatbot."

    elif "how are you" in user_input:
        return "I am doing well. Thank you for asking!"

    elif "python" in user_input:
        return "Python is a popular programming language."

    elif "help" in user_input:
        return "I can answer simple questions based on keywords."

    elif "bye" in user_input or "exit" in user_input:
        return "Goodbye! Have a nice day."

    else:
        return "Sorry, I don't understand that."

print("Chatbot: Hello! Type 'bye' to exit.")

while True:
    user_input = input("You: ")

    response = chatbot_response(user_input)
    print("Chatbot:", response)

    if "bye" in user_input.lower() or "exit" in user_input.lower():
break
```

Output:

Chatbot: Hello! Type 'bye' to exit.

You: Hi

Chatbot: Hello! How can I help you today?

You: What is Python?

Chatbot: Python is a popular programming language.

You: bye

Chatbot: Goodbye! Have a nice day.

12. Write a program to spell-checker using bigrams or Levenshtein distance.

```
def bigrams(w): return [w[i:i+2] for i in range(len(w)-1)]
def similarity(a,b):
    x,y=bigrams(a),bigrams(b)
    return len(set(x)&set(y))/len(set(x)|set(y))

dict_words=["apple","banana","orange","grapes"]
word=input("Enter word: ")
print(max(dict_words,key=lambda w:similarity(word,w)))
```

13. Write a function that takes (01, 02, Li, La) and outputs (x, y) of the end effector.

```
import math
def forward_kinematics_2dof(q1, q2, l1, l2):
    """
    Calculates the end-effector (x, y) coordinates for a 2-DOF planar robot arm.

    Args:
        q1 (float): Angle of the first joint in radians.
        q2 (float): Angle of the second joint in radians (relative to the first link).
        l1 (float): Length of the first link.
```

```
        l2 (float): Length of the second link.

    Returns:
        tuple: A tuple containing the (x, y) coordinates of the end-effector.
    """
    x = l1 * math.cos(q1) + l2 * math.cos(q1 + q2)
    y = l1 * math.sin(q1) + l2 * math.sin(q1 + q2)
    return x, y

if __name__ == '__main__':
    # Example usage:
    q1_rad = math.pi / 4  # 45 degrees
    q2_rad = math.pi / 2  # 90 degrees
    link1_length = 1.0
    link2_length = 0.8

    end_effector_x, end_effector_y = forward_kinematics_2dof(q1_rad, q2_rad, link1_length,
link2_length)

    print(f"End-effector X coordinate: {end_effector_x:.2f}")
    print(f"End-effector Y coordinate: {end_effector_y:.2f}")

    # Another example
    q1_rad_2 = 0
    q2_rad_2 = 0
    link1_length_2 = 1.0
    link2_length_2 = 1.0

    end_effector_x_2, end_effector_y_2 = forward_kinematics_2dof(q1_rad_2, q2_rad_2,
link1_length_2, link2_length_2)
    print(f"\nEnd-effector X coordinate (second example): {end_effector_x_2:.2f}")
```

```
    print(f"End-effector Y coordinate (second example): {end_effector_y_2:.2f}")
```

OUTPUT:

End-effector X coordinate: 0.14

End-effector Y coordinate: 1.27

End-effector X coordinate (second example): 2.00

End-effector Y coordinate (second example): 0.00

14. Implement backward chaining to derive diagnoses from user symptom input.

```
rules = {
    "flu": ["fever", "cough"],
    "measles": ["fever", "rash"],
    "meningitis": ["headache", "stiff_neck"]
}
facts = {}  # stores user responses (symptom: True/False)
def ask_user(symptom):
    """Ask the user whether a symptom is present"""
    if symptom not in facts:
        answer = input(f"Do you have {symptom}? (yes/no): ").lower()
        facts[symptom] = True if answer == "yes" else False
    return facts[symptom]
def backward_chaining(goal, visited=None):
    if visited is None:
        visited = set()

    if goal in visited:
        return False
```

```
    visited.add(goal)
    # If goal is a symptom, ask the user
    if goal not in rules:
        return ask_user(goal)

    print(f"\n🔍 Checking if {goal.upper()} can be diagnosed...")

    # Check all conditions required for this goal
    for condition in rules[goal]:
        if not backward_chaining(condition, visited):
            return False

    return True
# Main Program
print("=== Medical Diagnosis System (Backward Chaining) ===")
print("Available diseases:", list(rules.keys()))

goal = input("\nEnter disease to diagnose: ").lower()

if goal in rules:
    if backward_chaining(goal):
        print(f"\n✅ Diagnosis Result: {goal.upper()} is CONFIRMED")
    else:
        print(f"\n❌ Diagnosis Result: {goal.upper()} is NOT confirmed")
else:
    print("Invalid disease selected").
```