AI Applications and Ethics

LAB – 4

Utkarsh Bhangale

20200802124

Aim: The aim of this practical is to design and implement a program that solves the Missionaries and Cannibals problem, a classic puzzle in artificial intelligence, using a depth first search (DFS) algorithm. The program aims to find a sequence of moves that safely transport all missionaries and cannibals from the left bank of a river to the right bank while adhering to specific constraints.

Software Required: Python compiler

Prerequisite: Problem Solving, Transition Operators, Heuristic Function

Objectives:

- Develop a Python program to model the Missionaries and Cannibals problem as a search problem.
- Implement a state representation that tracks the number of missionaries and cannibals on each river bank and the boat's position.
- Use depth-first search to find a sequence of valid moves that solve the problem.
- Enforce constraints to prevent cannibals from outnumbering missionaries on either bank.
- Demonstrate and verify the solution path satisfies all constraints.

Theory:

The problem involves transporting 3 missionaries and 3 cannibals across a river using a boat that can carry max 2 people at a time. The cannibals cannot outnumber the missionaries on either bank, or they will eat the missionaries. Depth-first search systematically explores possible states to find a safe solution.

Algorithm:

- Define initial state, goal state and constraints
- Use depth-first search to find a sequence of valid moves
- Iterate until goal state is reached, generating and applying valid moves

- Return sequence of moves as solution

Real-World Applications:

- Game AI for pathfinding and decision making
- Robotics for navigation planning
- Transportation/logistics optimization
- Emergency response planning
- Traffic control optimization

Code –

```
class State:
    def __init__(self, left_m, left_c, boat, right_m, right_c):
        self.left_m = left_m
        self.left_c = left_c
        self.boat = boat
        self.right_m = right_m
        self.right_c = right_c

    def is_valid(self):
        if self.left_m < self.left_c and self.left_m > 0:
            return False
        if self.right_m < self.right_c and self.right_m > 0:
            return False
        return True

    def is_goal(self):
        return self.left_m == 0 and self.left_c == 0

    def get_neighbors(self):
        neighbors = []

        if self.boat == 0:
            # boat on left bank
            for dm in range(3):
                for dc in range(3):
                    if 0 < dm + dc <= 2:
                        new_state = State(self.left_m - dm, self.left_c -
dc,
                                          1, self.right_m + dm, self.right_c +
dc)
                        if new_state.is_valid():
                            neighbors.append(new_state)

        else:
```

```
            # boat on right bank
            for dm in range(3):
                for dc in range(3):
                    if 0 < dm + dc <= 2:
                        new_state = State(self.left_m + dm, self.left_c + dc,
                                          0, self.right_m - dm, self.right_c -
dc)
                        if new_state.is_valid():
                            neighbors.append(new_state)

        return neighbors

def dfs_search():
    initial_state = State(3, 3, 0, 0, 0)
    goal_state = State(0, 0, 1, 3, 3)

    stack = [initial_state]
    visited = set()

    while stack:
        current = stack.pop()
        if current.is_goal():
            return current

        visited.add(current)

        for neighbor in current.get_neighbors():
            if neighbor not in visited:
                stack.append(neighbor)

    return None

def print_solution(state):
    if state is None:
        print("No solution found")
    else:
        print(state)

solution = dfs_search()
print_solution(solution)
```