Title : Missionaries and Cannibals Problem

Experiment No: 5 Date: / /2025

Aim: To find solution to Missionaries and Cannibals problem using DFS (Depth First Search)

Theory :

The Missionaries and Cannibals problem is a classic example of state-space search in artificial intelligence. The problem consists of three missionaries and three cannibals who need to cross a river using a boat that can carry at most two people. The challenge is to ensure that at no point on either bank do the cannibals outnumber the missionaries, as this would result in missionaries being eaten.

The problem can be solved using various search algorithms, such as Depth-First Search (DFS) and Breadth-First Search (BFS). The state of the problem is typically represented as (M, C, B), where M is the number of missionaries on the left bank, C is the number of cannibals on the left bank, and B represents the boat's position (L for left, R for right). The goal is to transition from the initial state (3,3,L) to the goal state (0,0,R), while obeying the constraints.

The solution involves generating valid state transitions and using a search algorithm to explore paths until the goal is reached.

Algorithm:

1. **Initialize the stack** with the starting state (3,3,L).
2. **Mark the initial state as visited** and store the parent-child relationship for path reconstruction.
3. **While the stack is not empty:**
   - Pop the top state from the stack.
   - If the popped state is the goal (0,0,R), reconstruct and store the solution path.
   - Otherwise, generate all possible valid states from the current state.
   - For each valid state:
     - If it has not been visited, push it onto the stack.
     - Mark it as visited.
     - Record its parent for path reconstruction.
4. **Repeat** until the stack is empty.
5. **Return the solution path** if found, otherwise indicate failure.

Code:

```
from collections import deque

class state:
    def __init__(self,m,c,direction):
        self.m=m
        self.c=c
        self.direction=direction

    def __eq__(self, other):
        return self.m==other.m and self.c==other.c and self.direction==other.direction

    def __hash__(self):
        """Allows the state to be used as a key in a dictionary."""
        return hash((self.m,self.c,self.direction))

    def __str__(self):
        return f"({self.m}, {self.c}, {self.direction})"

    def missionary1(self):
        if self.m >= 1 and self.direction == "L":
            return state(self.m - 1, self.c, "R")
        elif self.m <= 2 and self.direction == "R":
            return state(self.m + 1, self.c, "L")

    def missionary2(self):
        if self.m >= 2 and self.direction == "L":
            return state(self.m - 2, self.c, "R")
        elif self.m <= 1 and self.direction == "R":
            return state(self.m + 2, self.c, "L")

    def cannibal1(self):
        if self.c >= 1 and self.direction == "L":
            return state(self.m, self.c - 1, "R")
        elif self.c <= 2 and self.direction == "R":
            return state(self.m, self.c + 1, "L")

    def cannibal2(self):
        if self.c >= 2 and self.direction == "L":
            return state(self.m, self.c - 2, "R")
        elif self.c <= 1 and self.direction == "R":
            return state(self.m, self.c + 2, "L")

    def missionary1_cannibal1(self):
        if self.m >= 1 and self.c >= 1 and self.direction == "L":
            return state(self.m - 1, self.c - 1, "R")
        elif self.m <= 2 and self.c <= 2 and self.direction == "R":
            return state(self.m + 1, self.c + 1, "L")

    def is_valid(self):
        if self.m < 0 or self.c < 0 or self.m > 3 or self.c > 3:
            return False
        if (self.m > 0 and self.c > self.m) or (3 - self.m > 0 and (3 - self.c) > (3 - self.m)):
            return False
        return True

    def return_moves(self):
        moves = [
            self.missionary1(),
            self.cannibal1(),
            self.missionary2(),
            self.cannibal2(),
            self.missionary1_cannibal1()
        ]
        return [move for move in moves if move and move.is_valid()]

def MnC_dfs():
    start = state(3,3,"L")
    goal = state(0,0,"R")
    stack = deque([start])
    visited = set()
    visited.add(start)
    solutions = []
    parent = {start:None}
    while stack:
        current = stack.pop()

        # If goal is reached, reconstruct and store the solution path
        if current == goal:
            solutions.append(reconstruct_path(goal,parent))
            continue
        else:
            # Explore next valid states
            for next_state in current.return_moves():
                if next_state not in visited or next_state == goal:
                    stack.append(next_state)
                    visited.add(next_state)
                    parent[next_state]=current
    return solutions

def reconstruct_path(end_state,parent):
    path = []
    current = end_state
    while current is not None:
        path.append(current)
        current=parent[current]
    path.reverse()
    return path

sols = MnC_dfs()
if sols:
    print(f"{len(sols)} solutions found")
    for idx,sol in enumerate(sols,start=1):
        print(f"\nSolution {idx}:")
        print(" -> ".join(str(state) for state in sol))
else:
    print("No solution found")
```

Output:

```
2 solutions found

Solution 1:
(3, 3, L) -> (2, 2, R) -> (3, 2, L) -> (3, 0, R) -> (3, 1, L) -> (1, 1, R) -> (2, 2, L) -> (0, 2, R) -> (0, 3, L) ->
(0, 1, R) -> (0, 2, L) -> (0, 0, R)

Solution 2:
(3, 3, L) -> (2, 2, R) -> (3, 2, L) -> (3, 0, R) -> (3, 1, L) -> (1, 1, R) -> (2, 2, L) -> (0, 2, R) -> (0, 3, L) ->
(0, 1, R) -> (1, 1, L) -> (0, 0, R)
```

Conclusion:

The Missionaries and Cannibals problem demonstrates state-space search and constraint satisfaction in AI. Using DFS, we explore possible moves to find a valid solution while ensuring constraints are met. This problem highlights the importance of search algorithms in problem-solving. While DFS finds a solution, other methods like BFS or heuristic approaches can provide more efficient results.