4.17 MOUSE AND CAT GAME ON AN UNDIRECTED GRAPH

Question:

A game on an undirected graph is played by two players, Mouse and Cat, who alternate turns. The graph is given as follows: graph[a] is a list of all nodes b such that ab is an edge of the graph. The mouse starts at node 1 and goes first, the cat starts at node 2 and goes second, and there is a hole at node 0. During each player's turn, they must travel along one edge of the graph that meets where they are. For example, if the Mouse is at node 1, it must travel to any node in graph[1]. Additionally, it is not allowed for the Cat to travel to the Hole (node 0).Then, the game can end in three ways:

- If ever the Cat occupies the same node as the Mouse, the Cat wins.
- If ever the Mouse reaches the Hole, the Mouse wins.
- If ever a position is repeated (i.e., the players are in the same position as a previous turn, and it is the same player's turn to move), the game is a draw.

Given a graph, and assuming both players play optimally, return

- 1 if the mouse wins the game,
- 2 if the cat wins the game, or
- 0 if the game is a draw.

AIM

To implement a recursive simulation with memorization to determine the outcome of the Mouse vs. Cat game, assuming both players play optimally.

ALGORITHM

1. Define a 3D memoization table dp[turn][mouse][cat] to store outcomes.
2. Base cases:
    - If mouse == 0: Mouse wins
    - If mouse == cat: Cat wins
    - If turn == 2n: Draw (cycle detected)

3. Recursively simulate all valid moves for the current player.

4. For Mouse's turn:
    - Try all moves from graph[mouse]
    - If any move leads to Mouse win, return 1
    - If all moves lead to Cat win, return 2
    - Otherwise, return 0
5. For Cat's turn:
    - Try all moves from graph[cat] except node 0
    - If any move leads to Cat win, return 2
    - If all moves lead to Mouse win, return 1
    - Otherwise, return 0

PROGRAM

```
from collections import deque
def cat_mouse_game(graph):
    n = len(graph)
    DRAW, MOUSE_WIN, CAT_WIN = 0, 1, 2
    dp = [[[DRAW] * 3 for _ in range(n)] for _ in range(n)]
    queue = deque()

    for i in range(n):
        dp[0][i][0] = MOUSE_WIN/
        dp[i][i][0] = CAT_WIN
        queue.append((0, i, 0))
        queue.append((i, i, 0))

    def parents(m, c, t):
        if t == 0:
            for pm in graph[m]:
                yield (pm, c, 1)
        else:
            for pc in graph[c]:
                if pc != 0:
                    yield (m, pc, 0)

    while queue:
        m, c, t = queue.popleft()
        result = dp[m][c][t]
        for pm, pc, pt in parents(m, c, t):
            if dp[pm][pc][pt] != DRAW:
                continue
            if result == MOUSE_WIN and pt == 0:
                dp[pm][pc][pt] = MOUSE_WIN
                queue.append((pm, pc, pt))
            elif result == CAT_WIN and pt == 1:
                dp[pm][pc][pt] = CAT_WIN
                queue.append((pm, pc, pt))
            else:
                if pt == 0:
                    if all(dp[nm][pc][1] == CAT_WIN for nm in graph[pm]):
                        dp[pm][pc][pt] = CAT_WIN
                        queue.append((pm, pc, pt))
                else:
                    if all(dp[pm][nc][0] == MOUSE_WIN for nc in graph[pc] if nc != 0):
                        dp[pm][pc][pt] = MOUSE_WIN
                        queue.append((pm, pc, pt))

    return dp[1][2][0]

graph = eval(input("Enter graph as adjacency list: "))
print("Game result:", cat_mouse_game(graph))
```

Input:

Enter graph as adjacency list: [[1,3],[0],[3],[0,2]]

Output:

```
Enter graph as adjacency list: [[1,3],[0],[3],[0,2]]
Game result: 0
>>>
```

RESULT:

Thus the program is successfully executed and the output is verified.

PERFORMANCE ANALYSIS:

- Time Complexity: $O(n^3)$, due to 3D memorization
- Space Complexity: $O(n^3)$, for storing game states