1.16 CONWAY'S GAME OF LIFE – NEXT STATE OF THE GRID

Question:

"The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970." The board is made up of an m x n grid of cells, where each cell has an initial state: live (represented by a 1) or dead (represented by a 0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules

Any live cell with fewer than two live neighbors dies as if caused by under-population.

Any live cell with two or three live neighbors lives on to the next generation.

Any live cell with more than three live neighbors dies, as if by over-population.

Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The next state is created by applying the above rules simultaneously to every cell in the current state, where births and deaths occur simultaneously. Given the current state of the m x n grid board, return the *next state*.

AIM:

To compute the next state of an m x n grid based on the rules of Conway's Game of Life, where each cell updates simultaneously depending on its neighbors.

ALGORITHM:

1. Create a copy of the board (since updates are simultaneous).
2. For each cell (i, j):
    - Count live neighbors (check 8 possible directions).
    - Apply the four rules to determine its next state.
3. Update the original board with computed values.
4. Return the updated board.

PROGRAM:

```
def average_neighbors(grid):
    rows, cols = len(grid), len(grid[0])
    result = [[0]*cols for _ in range(rows)]
    directions = [(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1), (1,0), (1,1)]

    for i in range(rows):
        for j in range(cols):
            total = 0
            count = 0
            for dx, dy in directions:
                ni, nj = i + dx, j + dy
                if 0 <= ni < rows and 0 <= nj < cols:
                    total += grid[ni][nj]
                    count += 1
            result[i][j] = total // count if count else 0
    return result

def run_average_neighbours():
    m = int(input("Enter number of rows: "))
    n = int(input("Enter number of columns: "))
    print("Enter grid row by row:")
    grid = [list(map(int, input().split())) for _ in range(m)]
    updated = average_neighbors(grid)
    print("Updated grid:")
    for row in updated:
        print(row)
run_average_neighbours()
```

Input:

Row=3

Column=3

Board = 1 1 1

1 0 1

1 1 1

Output:

```
Enter number of rows: 3
Enter number of columns: 3
Enter grid row by row:
1 1 1
1 0 1
1 1 1
Updated grid:
[0, 0, 0]
[0, 1, 0]
[0, 0, 0]
>>>
```

RESULT:

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

PERFORMANCE ANALYSIS:

- Time Complexity: O(m * n) → each cell is visited once, checking 8 neighbors.
- Space Complexity: O(m * n) if using a copy, or O(1) if updating in-place with state markers.