289. Game of Life

Medium Array Matrix Simulation

Leetcode Link

Problem Description

The "Game of Life" is a fascinating cellular automaton created by mathematician John Horton Conway. In this game, we're presented with a grid of cells, each of which can either be alive (denoted by '1') or dead (indicated by '0'). The game progresses in steps, or generations, with each cell's fate in the next generation determined by its current state and the states of its eight neighboring cells. The challenge is to implement an algorithm that takes an `m x n` grid representing the current state of a board and produces the next state of the board based on the following rules:

1. Any living cell with fewer than two live neighbors dies, as if by underpopulation.
2. Any living cell with two or three live neighbors lives on to the next generation.
3. Any living cell with more than three live neighbors dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The game's rules are applied to every cell in the board simultaneously. This means that when calculating the next state, the original state must remain unchanged until all cells have been considered.

Intuition

The solution requires us to update the board to its next state without using additional space to store an intermediate state, which can be tricky because the update of one cell should not affect the update of another cell.

A clever approach to solving this problem within the constraints is to define intermediate states:

- Let state `2` represent a cell that was alive but will die in the next state.
- Let state `-1` represent a cell that was dead but will become alive in the next state.

This way, we can encode the current and next state of a cell within the same grid. A positive value indicates the cell is currently alive, and its specific integer value indicates whether it will stay alive or die in the next state. Similarly, a negative or zero value indicates the cell is currently dead, and the exact value determines if it will remain dead or come to life.

The algorithm proceeds in two passes over the grid:

1. In the first pass, we calculate the next state for each cell without updating the board to the final state. Instead, we use the intermediate states `2` and `-1` if a cell's state is going to change. We use the variable `live` to count the number of living neighbors around each cell considering these transitions.

2. In the second pass, we update cells in state `2` to `0` (dead) and cells in state `-1` to `1` (alive), thus applying the transitions and achieving the next state of the board.

This approach leverages the fact that the state information is stored in-place with clear rules for interpretation, allowing us to determine a cell's original and next state without additional memory usage.

Solution Approach

The solution is implemented in two major steps: firstly detecting changes that will happen in the next state, and secondly applying these changes to realize the next state. This process happens in-place within the original grid to avoid extra space usage, thanks to the in-place marking strategy explained in the intuition part.

1. The first part of the solution involves iterating over all cells in the grid to determine their fate according to the Game of Life rules. While doing that, we keep track of the cell's state change using temporary markers.

   To determine the fate of each cell, we need to count its living neighbors. We create a nested loop to check all neighboring cells. If a neighbor is marked with a positive value, we know it's currently alive. Since we're also interested in cells that are about to change state from living to dead, we slightly modify the neighbor count logic. We start with the count at `-board[i][j]`, which helps us avoid counting the cell itself if it's alive.

   We use an `if` condition with a range to ensure we stay within the bounds of the grid without wrapping around. For each live neighbor, we increment the `live` counter. After counting neighbors, we apply the Game of Life rules:

   - If a cell is alive (`board[i][j] > 0`) but has fewer than two or more than three live neighbors (`live < 2` or `live > 3`), it is marked as `2`, signifying it will die.
   - Conversely, a dead cell (indicated by `board[i][j] == 0`) with exactly three live neighbors (`live == 3`) is marked as `-1`, signifying it will become alive.
2. Once the first pass is complete, all cells on the board will either have their original values (indicating no change) or will be marked with `2` (if they are to die) or `-1` (if they are to become alive). Therefore, in the second part, we make another pass over the grid to finalize the state transitions.

   - If a cell is marked `2`, this means it was alive but is about to die, so we set it to `0`.
   - If a cell is marked `-1`, it was dead and is about to come to life, so we set it to `1`.

By treating the states `2` and `-1` as temporary placeholders, the algorithm can keep track of both the current and future states of each cell without needing additional space for storing the grid's state. This efficient handling of the states demonstrates a common strategy in algorithm design called in-place computation, which is particularly useful for saving memory and often necessary when dealing with constraints such as constant space complexities.

Example Walkthrough

Let's consider a 3×3 grid to illustrate the solution approach:

```
Current state of the grid:
1 1 1
0 0 0
1 0 1
```

We will use the rules provided and apply the first phase of the solution. Let's see how we can determine the next state for each cell.

1. First Pass (Detection Phase):

- For the cell at position (0, 0), which is alive (1), it has three live neighbors. According to the rules, a living cell with two or three live neighbors lives on. Therefore, this cell does not change.
- For the cell at position (0, 1), which is alive (1), it has three live neighbors. This cell also lives on, so no change.
- For the cell at position (0, 2), which is alive (1), it has two live neighbors. This cell continues to live, resulting in no change.
- The cell at position (1, 0) is dead (0), and it has two live neighbors. It remains dead, as it doesn't meet the reproduction rule (exactly three live neighbors required).
- The cell at position (1, 1) is dead (0) and has four live neighbors. It continues to be dead because it doesn't have exactly three live neighbors to become alive.
- The cell at position (1, 2) is dead (0), with three live neighbors. According to the rules, a dead cell with three live neighbors becomes alive. Therefore, this cell is marked as `-1` (an intermediate state showing it will be alive in the next state).

For the corner cells at positions (2, 0) and (2, 2), both being alive with only one living neighbor, they will die due to underpopulation. Thus, these cells are marked as `2`.

```
Intermediate state of the grid after the first pass:
1  1  1
0 -1  0
2  0  2
```

2. Second Pass (Update Phase):

Now we go through the grid again and apply the second part of the algorithm to update the cells.

- Cells at positions (0, 0), (0, 1), (0, 2), and (1, 0) retain the same state since they weren't assigned any intermediate state, staying at `1` and `0` respectively.
- The cell at position (1, 1) remains dead, so no change.
- The cell at position (1, 2) was marked as `-1` and is set to `1` to finalize its transition to a live cell.
- Cells at positions (2, 0) and (2, 2) were marked as `2` (meaning they should die) and are now set to `0`.

The final state of the grid, representing the next state, is now:

```
Next state of the grid after the second pass:
1 1 1
0 1 0
0 0 0
```

This example demonstrates how each cell's next state is determined using intermediate states to facilitate in-place updates, maintaining the integrity of the game's rules without additional space.

Python Solution

```python
class Solution:
    def gameOfLife(self, board: List[List[int]]) -> None:
        """
        Do not return anything, modify 'board' in-place instead.
        This function computes the next state of the Game of Life board.

        Rules for the Game of Life:
        1. Any live cell with fewer than two live neighbors dies, as if caused by underpopulation.
        2. Any live cell with two or three live neighbors lives on to the next generation.
        3. Any live cell with more than three live neighbors dies, as if by overpopulation.
        4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.
        """

        # Get the dimensions of the board
        rows, cols = len(board), len(board[0])

        # Iterate over each cell of the board to compute the next state
        for row in range(rows):
            for col in range(cols):
                live_neighbors = -board[row][col]  # Start with the negated cell state to offset the current cell count later
                # Examine all the neighbor cells in the surrounding 3x3 area
                for x in range(row - 1, row + 2):
                    for y in range(col - 1, col + 2):
                        # Check if the neighbor is within bounds and is alive
                        if 0 <= x < rows and 0 <= y < cols and board[x][y] > 0:
                            live_neighbors += 1
                # Apply the rules of the Game of Life for the next state of the cell
                # If the cell is alive and has too few or too many neighbors, it becomes 'dead' for the next state
                if board[row][col] and (live_neighbors < 2 or live_neighbors > 3):
                    board[row][col] = 2  # Mark this cell for death
                # If the cell is dead and has exactly 3 live neighbors, it becomes 'alive' for the next state
                if board[row][col] == 0 and live_neighbors == 3:
                    board[row][col] = -1  # Mark this cell for life

        # Finalize the next state of the board
        for row in range(rows):
            for col in range(cols):
                # Set the cell state to 'dead' if it was marked for death
                if board[row][col] == 2:
                    board[row][col] = 0
                # Set the cell state to 'alive' if it was marked for life
                elif board[row][col] == -1:
                    board[row][col] = 1
```

Java Solution

```java
class Solution {
    public void gameOfLife(int[][] board) {
        // numRows and numCols hold the dimensions of the board
        int numRows = board.length;
        int numCols = board[0].length;

        // Traverse through every cell of the board
        for (int row = 0; row < numRows; ++row) {
            for (int col = 0; col < numCols; ++col) {
                // Count live neighbors, initializing at -board[row][col] to offset self-counting if alive
                int liveNeighbors = -board[row][col];
                for (int i = row - 1; i <= row + 1; ++i) {
                    for (int j = col - 1; j <= col + 1; ++j) {
                        // Check if neighbor is within bounds and alive
                        if (i >= 0 && i < numRows && j >= 0 && j < numCols && board[i][j] > 0) {
                            liveNeighbors++;
                        }
                    }
                }

                // Apply the Game of Life rules to determine next state:

                // Rule 1 or Rule 3: Any live cell with fewer than two live neighbors
                // or with more than three live neighbors dies (set to 2 for temporary state)
                if (board[row][col] == 1 && (liveNeighbors < 2 || liveNeighbors > 3)) {
                    board[row][col] = 2;
                }

                // Rule 4: Any dead cell with exactly three live neighbors becomes a live cell
                // (set to -1 for temporary state)
                if (board[row][col] == 0 && liveNeighbors == 3) {
                    board[row][col] = -1;
                }
            }
        }

        // Re-traverse the board to rewrite the temporary states to final states
        for (int row = 0; row < numRows; ++row) {
            for (int col = 0; col < numCols; ++col) {
                // A value of 2 means the cell was previously alive and now is dead
                if (board[row][col] == 2) {
                    board[row][col] = 0;
                }
                // A value of -1 means the cell was previously dead and now is alive
                else if (board[row][col] == -1) {
                    board[row][col] = 1;
                }
            }
        }
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function that simulates the Game of Life on the given board.
    void gameOfLife(vector<vector<int>>& board) {
        int rows = board.size();    // Number of rows in the board
        int cols = board[0].size(); // Number of columns in the board

        // Iterate over each cell in the board
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                int liveNeighbors = -board[i][j]; // Initialize live neighbors count

                // Iterate over all the neighbors of the current cell
                for (int x = i - 1; x <= i + 1; ++x) {
                    for (int y = j - 1; y <= j + 1; ++y) {
                        // Check if the neighbor is within the board boundaries and is alive
                        if (x >= 0 && x < rows && y >= 0 && y < cols && board[x][y] > 0) {
                            ++liveNeighbors;
                        }
                    }
                }

                // Apply the Game of Life rules for living cells
                if (board[i][j] == 1 && (liveNeighbors < 2 || liveNeighbors > 3)) {
                    board[i][j] = 2; // Mark for death
                }
                // Apply the Game of Life rules for dead cells
                if (board[i][j] == 0 && liveNeighbors == 3) {
                    board[i][j] = -1; // Mark for life
                }
            }
        }

        // Update the board with the new state
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // If the cell was marked for death, make it dead
                if (board[i][j] == 2) {
                    board[i][j] = 0;
                } else if (board[i][j] == -1) { // If the cell was marked for life, make it alive
                    board[i][j] = 1;
                }
            }
        }
    }
};
```

Typescript Solution

```typescript
/**
 * Apply the Game of Life rules to the board in place.
 * @param board - The 2D array representing the Game of Life board, where 1 is a live cell and 0 is a dead cell.
 */
function gameOfLife(board: number[][]): void {
    const rows = board.length;
    const cols = board[0].length;

    // Iterate over each cell to apply the Game of Life rules
    for (let row = 0; row < rows; ++row) {
        for (let col = 0; col < cols; ++col) {
            // Initialize live neighbor count; the cell itself is counted and will be subtracted later if alive
            let liveNeighbors = -board[row][col];

            // Check all 8 neighbors of the current cell
            for (let x = row - 1; x <= row + 1; ++x) {
                for (let y = col - 1; y <= col + 1; ++y) {
                    // Check if neighbor coordinates are within the board bounds
                    if (x >= 0 && x < rows && y >= 0 && y < cols && board[x][y] > 0) {
                        // Increment live neighbor count if neighbor is alive
                        ++liveNeighbors;
                    }
                }
            }

            // Apply the Game of Life rules:
            // 1. Any live cell with fewer than two or more than three live neighbors dies
            if (board[row][col] === 1 && (liveNeighbors < 2 || liveNeighbors > 3)) {
                board[row][col] = 2; // Mark the cell to become dead
            }
            // 2. Any dead cell with exactly three live neighbors becomes a live cell
            if (board[row][col] === 0 && liveNeighbors === 3) {
                board[row][col] = -1; // Mark the cell to become alive
            }
        }
    }

    // Finalize the board state by changing marked cells to their new states
    for (let row = 0; row < rows; ++row) {
        for (let col = 0; col < cols; ++col) {
            if (board[row][col] === 2) {
                board[row][col] = 0; // Dead cells become 0
            }
            if (board[row][col] === -1) {
                board[row][col] = 1; // New live cells become 1
            }
        }
    }
}
```

Time and Space Complexity

// The time complexity of the code is `O(m * n)`, where `m` is the number of rows and `n` is the number of columns in the `board`. This is because there are two nested loops that iterate through each cell of the `board` once.

// The space complexity of the code is `O(1)` since it modifies the `board` in place without using any additional space proportional to the size of the input. All changes are made directly on the input `board`, and only a fixed number of extra variables are used.