130. Surrounded Regions

Medium Depth-First Search Breadth-First Search Union Find Array **Matrix** Leetcode Link

Problem Description

In this problem, we are given a `m x n` matrix called `board`, composed of characters `'X'` and `'O'`. The goal is to identify all the regions formed by `'O'` that are completely surrounded by `'X'` in all four directions (up, down, left, and right) without any breaks. Once these regions are identified, we need to flip all the `'O'`s in those regions to `'X'`s. It's important to note that an 'O' on the boundary of the board isn't considered surrounded, and hence, the regions connected to an 'O' on the board boundary are safe and should not be flipped.

Intuition

The solution is based on the idea that an 'O' region would not be captured if it's connected directly or indirectly to an 'O' on the boundary of the board since it cannot be surrounded entirely by 'X's. To implement this, we apply depth-first search (DFS) to mark all 'O's that are connected to the boundary 'O's with a temporary marker (in this case, '.'). This multi-step approach ensures we only flip those 'O's into 'X's that are truly surrounded:

1. **Identify Boundary-Connected 'O's**: Iterate over the border rows and columns. For any 'O' found on the boundary, perform a DFS search to mark not only this 'O' but also any other 'O' connected to this one, directly or indirectly. We replace these 'O's temporarily with '.' to indicate they're safe from flipping.

2. **Capture Surrounded Regions**: After the DFS marking step, we go through the entire board to flip the remaining 'O's (those that are not marked with '.') into 'X's as they are surrounded by 'X's.

3. **Restore Boundary-Connected Regions**: Finally, we make another pass to convert all temporary markers '.' back into 'O's to restore the initial state for all the 'O's that were connected to the board boundary.

By taking this approach, we ensure to flip only those 'O's that are truly surrounded by 'X's, while preserving the boundary-connected 'O's.

Solution Approach

The solution implements a depth-first search (DFS) algorithm to modify the matrix in place. Here's a walk-through of the code and the algorithms/data structures/patterns used in the solution:

1. **Depth-First Search (DFS) Algorithm**: The `dfs` function is a classic implementation of the DFS algorithm. When it encounters an `'O'`, it changes that `'O'` to a temporary marker `'.'` to indicate that it has been visited and should not be flipped later. Then, it checks adjacent cells (in all four directions) and recursively calls itself to mark connected 'O's.

```python
def dfs(i, j):
    board[i][j] = '.'
    for a, b in [[0, -1], [0, 1], [1, 0], [-1, 0]]:
        x, y = i + a, j + b
        if 0 <= x < m and 0 <= y < n and board[x][y] == 'O':
            dfs(x, y)
```

2. **Boundary Cells Check**: We iterate over the matrix looking for `'O'`s on the boundary edges. For every `'O'` found, the `dfs` function is called to start the marking process.

```python
for i in range(m):
    for j in range(n):
        if board[i][j] == 'O' and (
            i == 0 or i == m - 1 or j == 0 or j == n - 1
        ):
            dfs(i, j)
```

3. **Flipping Surrounded 'O's Inside the Matrix**: After marking all the boundary-connected `'O'`s with `'.'`, we iterate over the matrix again. This time, we flip the unmarked `'O'`s to `'X'`s because they're surrounded by 'X's. The marked `'.'` cells retain information about boundary-connected 'O's and are not flipped.

```python
for i in range(m):
    for j in range(n):
        if board[i][j] == 'O':
            board[i][j] = 'X'
```

4. **Restoring the Boundary-Connected 'O's**: In the last iteration over the matrix, we revert our temporary markers `'.'` back to `'O'`s, restoring their original state as they were not supposed to be flipped.

```python
for i in range(m):
    for j in range(n):
        if board[i][j] == '.':
            board[i][j] = 'O'
```

In this implementation, no additional data structures are needed as we modify the board in place. The pattern used here mainly revolves around the DFS algorithm, which is a powerful tool for searching or traversing through an adjacency graph, or in this case, a matrix, especially when we are dealing with connected components.

The solution has linear complexity with respect to the number of cells in the matrix since each cell is visited at most twice. Once during the DFS marking stage and once during the flipping stage. This ensures an efficient solution to the problem.

Example Walkthrough

Let's consider a small `3x4` board as an example:

```
Board:
O X X X
X O O X
X X X X
```

Step 1: Identify Boundary-Connected 'O's

We find that the 'O' in the top left corner is on the boundary. Applying the DFS algorithm starting from this 'O', we change it to a '.' to mark it as visited and safe:

```
Board:
. X X X
X O O X
X X X X
```

Since there are no other 'O's directly connected to the boundary 'O's, our board remains the same after the first step.

Step 2: Capture Surrounded Regions

Now, we scan through the rest of the board. The 'O's in the middle are surrounded by 'X's and there is no DFS path from any boundary 'O' to them. So we flip these 'O's to 'X's:

```
Board:
. X X X
X X X X
X X X X
```

Step 3: Restore Boundary-Connected Regions

Finally, we need to revert the '.' back to 'O', since it was marked only for the purpose of identification and not flipping:

```
Board:
O X X X
X X X X
X X X X
```

Our final board shows the 'O's in the middle flipped to 'X's, while the 'O' on the boundary remains intact. This concludes our walkthrough of how the depth-first search algorithm can be used to solve the problem of flipping all 'O's surrounded by 'X's on a board, while leaving the boundary-connected 'O's untouched.

Python Solution

```python
from typing import List  # Importing the List type from typing module for type hinting

class Solution:
    def solve(self, board: List[List[str]]) -> None:
        """
        The function modifies the input 2D board (List of List of strings) in-place.
        If an 'O' region is not surrounded by 'X' on the board's edges, it is flipped to 'X'.
        'O' regions that are on the edges or connected to an edge 'O' region remain as 'O'.
        """

        def depth_first_search(i: int, j: int) -> None:
            """Helper function to perform depth-first search and mark connected 'O's with a placeholder."""
            board[i][j] = '.'  # Temporary marking to keep track of visited and edge-connected 'O's
            directions = [[0, -1], [0, 1], [1, 0], [-1, 0]]  # Directions for exploring up, down, left, and right

            # Explore adjacent cells in all 4 directions
            for direction in directions:
                x, y = i + direction[0], j + direction[1]
                if 0 <= x < rows and 0 <= y < cols and board[x][y] == 'O':
                    depth_first_search(x, y)

        # Board dimensions
        rows, cols = len(board), len(board[0])

        # First, traverse the border cells to find 'O's connected to borders
        for i in range(rows):
            for j in range(cols):
                is_border_cell = i in (0, rows - 1) or j in (0, cols - 1)
                if board[i][j] == 'O' and is_border_cell:
                    depth_first_search(i, j)

        # Then, flip all the 'O' regions that are not on the border to 'X'
        # and convert the previously marked border-connected 'O's back to 'O'
        for i in range(rows):
            for j in range(cols):
                if board[i][j] == 'O':
                    board[i][j] = 'X'
                elif board[i][j] == '.':
                    board[i][j] = 'O'

```

Java Solution

```java
class Solution {
    private char[][] board; // Member variable to hold the input board
    private int rows; // Number of rows in the board
    private int cols; // Number of columns in the board

    // Main function that solves the board by replacing all 'O' not surrounded by 'X' with 'X'
    public void solve(char[][] board) {
        rows = board.length; // Set the number of rows
        cols = board[0].length; // Set the number of columns
        this.board = board; // Initialize the board member variable

        // Explore all 'O' on the borders, any 'O' connected to them should not be flipped
        // hence temporarily mark them with '.'
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                // Condition to check if it's on the border and if it's an 'O'
                if ((i == 0 || i == rows - 1 || j == 0 || j == cols - 1) && board[i][j] == 'O') {
                    depthFirstSearch(i, j); // Call DFS to mark the connected 'O's
                }
            }
        }

        // Flip all remaining 'O' to 'X' and back all '.' to 'O'.
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                // If it was marked '.', it's safe to flip it back to 'O'
                if (board[i][j] == '.') {
                    board[i][j] = 'O';
                }
                // If it's still an 'O', it should be flipped to 'X' as it is not connected to a border
                else if (board[i][j] == 'O') {
                    board[i][j] = 'X';
                }
            }
        }
    }

    // Depth-first search function to find all the 'O's connected to a border 'O'
    private void depthFirstSearch(int row, int col) {
        board[row][col] = '.'; // Mark the cell as visited by replacing 'O' with '.'
        int[] directions = {-1, 0, 1, 0, -1}; // Directions to move in the matrix
        for (int k = 0; k < 4; k++) { // Loop through possible directions (up, right, down, left)
            int nextRow = row + directions[k];
            int nextCol = col + directions[k + 1];
            // Check bounds and if the next cell is 'O', continue DFS
            if (nextRow >= 0 && nextRow < rows && nextCol >= 0 && nextCol < cols && board[nextRow][nextCol] == 'O') {
                depthFirstSearch(nextRow, nextCol); // Recursive call for connected 'O's
            }
        }
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Main function to solve the board game
    void solve(vector<vector<char>>& board) {
        int rows = board.size();                // Number of rows in board
        int cols = board[0].size();             // Number of columns in board

        // Traverse the boundary of the board and perform DFS for 'O'
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // Check if we're on the boundary of the board and the current cell is 'O'
                bool isBoundary = i == 0 || i == rows - 1 || j == 0 || j == cols - 1;
                if (isBoundary && board[i][j] == 'O') {
                    dfs(board, i, j);  // Call DFS to mark connected 'O's
                }
            }
        }

        // Flip all remaining 'O's to 'X's, then flip marked '.' back to 'O'
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                if (board[i][j] == '.') {
                    board[i][j] = 'O';  // Restore temporarily marked 'O's
                } else if (board[i][j] == 'O') {
                    board[i][j] = 'X';  // Flip untouched 'O's to 'X's
                }
            }
        }
    }

    // Helper DFS function to mark connected 'O's starting from (i, j)
    void dfs(vector<vector<char>>& board, int i, int j) {
        board[i][j] = '.';  // Temporarily mark this 'O' to avoid repeated processing

        // All possible directions to move (up, right, down, left)
        vector<int> dirs = {-1, 0, 1, 0, -1};

        // Traverse all directions
        for (int k = 0; k < 4; ++k) {
            int x = i + dirs[k];          // Next row index
            int y = j + dirs[k + 1];      // Next column index

            // Check if (x, y) is in bounds and is 'O'
            bool inBounds = (x >= 0 && x < rows && y >= 0 && y < cols);
            if (inBounds && board[x][y] == 'O') {
                dfs(board, x, y);  // Continue DFS for this 'O'
            }
        }
    }
};

```

Typescript Solution

```typescript
/**
 * Solves the "Surrounded Regions" problem on a 2D board inplace by flipping surrounded 'O's to 'X's.
 * Only 'O's on the border or connected to an 'O' on the border will remain as 'O'.
 * @param board A 2D array of strings representing the board
 */
function solve(board: string[][]): void {
    // Depth-first search to mark 'O's connected to the borders as temporary '.'
    function depthFirstSearch(row: number, col: number) {
        board[row][col] = '.';
        const directions = [-1, 0, 1, 0, -1]; // Directions array to simplify the exploration of adjacent cells

        // Explore all four directions
        for (let k = 0; k < 4; ++k) {
            const newRow = row + directions[k];
            const newCol = col + directions[k + 1];

            // If the new cell has 'O' and is within the bounds, continue the DFS
            if (newRow >= 0 && newRow < numRows && newCol >= 0 && newCol < numCols && board[newRow][newCol] == 'O') {
                depthFirstSearch(newRow, newCol);
            }
        }
    }

    const numRows = board.length;
    const numCols = board[0].length;

    // Start DFS from 'O's on the border
    for (let row = 0; row < numRows; ++row) {
        for (let col = 0; col < numCols; ++col) {
            const isBorderCell = row == 0 || row == numRows - 1 || col == 0 || col == numCols - 1;
            if (isBorderCell && board[row][col] == 'O') {
                depthFirstSearch(row, col);
            }
        }
    }

    // Post-process to complete the flip of surrounded regions
    for (let row = 0; row < numRows; ++row) {
        for (let col = 0; col < numCols; ++col) {
            if (board[row][col] == '.') {
                board[row][col] = 'O';                          // Restore temporary '.' to 'O'
            } else if (board[row][col] == 'O') {
                board[row][col] = 'X';                          // Flip completely surrounded 'O's to 'X'
            }
        }
    }
}

```

Time and Space Complexity

Time Complexity

The time complexity of this algorithm is `O(M*N)`, where `M` is the number of rows in the `board` and `N` is the number of columns. This is because the algorithm must visit every cell in the grid to perform the depth-first search (DFS) and the subsequent replacement of 'O's with 'X's.

DFS is invoked on cells that are on the border of the grid and are marked with 'O'. In the worst case, all border cells could be 'O's and the DFS would travel into every reachable 'O' from the border which can potentially cover the whole grid. Hence, the complexity considering the entire DFS calls collectively is linear with the number of cells, `O(M*N)`.

After the DFS, the algorithm goes through all cells to replace remaining 'O's with 'X's and convert all temporary '.' markers back to 'O's. This is a straightforward double for-loop over the cells, which again results in a complexity of `O(M*N)`.

Putting this together, despite having nested loops, the DFS does not visit cells more than once. Overall, the time complexity is `O(M*N)` as each cell is entered only once by the DFS.

Space Complexity

The space complexity of the algorithm is also `O(M*N)` in the worst-case scenario, due to the recursive nature of DFS. If the grid is filled with 'O's, the call stack for the DFS can grow as large as the number of cells in the grid before backtracking. This would be the case in a grid where one 'O' is connected to all other 'O's, thus the entire grid would be traversed without backtracking.

It is important to note that the space complexity assumes that the depth of the recursion stack is considered part of the space complexity. If the space used by the recursion stack is not considered, then the space complexity would be `O(1)` since the algorithm modifies the input grid in place and does not utilize additional space proportional to the input size.