529. Minesweeper

Medium | Depth-First Search | Breadth-First Search | Array | Matrix | Leetcode Link

Problem Description

The Minesweeper game consists of a board with mines ('M') and empty squares ('E'). The objective is to reveal all squares without triggering a mine. When a player clicks a square:

1. If a mine is revealed ('M'), the game ends and the mine becomes an 'X'.
2. If an empty square ('E') without adjacent mines is revealed, it becomes a 'B', representing a blank space, and its adjacent unrevealed squares should also be revealed recursively.
3. If an empty square ('E') with adjacent mines is revealed, it shows a digit ('1' to '8') representing the number of mines in the adjacent squares.
4. The board state is returned when no more squares can be revealed.

The task is to implement the logic that simulates this part of Minesweeper game behavior.

Intuition

To solve this problem, we use Depth-First Search (DFS) algorithm, where we start from the clicked position and explore adjacent squares. Here's a step-by-step breakdown:

1. Check the square at the clicked position:
   - If it's a mine ('M'), change it to an 'X'.
   - If it's an empty square ('E'), perform the following:
     - Count the number of mines in the adjacent squares.
     - If there are adjacent mines, update the square with the mine count.
     - If there are no adjacent mines, update the square to a 'B' and recursively reveal adjacent 'E' squares.
2. The base case for recursive DFS is when the current square is not an 'E' or when it is out of bounds, in which case we just return.
3. By following the steps above, we ensure that we reveal only the necessary squares to either show the number of adjacent mines or to expand the empty regions ('E' to 'B').
4. The recursion naturally takes care of revealing all connected empty squares ('E' to 'B') and stopping when it encounters numbered squares or the edge of the board.

This DFS approach encapsulates the essence of Minesweeper's revealing mechanic, carefully unveiling the board according to the game's rules.

Solution Approach

The implementation of the solution is a straightforward application of Depth-First Search (DFS). Here are the details of the implementation steps, corresponding to the given Python code:

1. The nested `dfs` function is defined to perform a DFS starting from a particular cell on the board.
2. The main logic for the DFS method is as follows:
   - First, we count the number of mines (`'M'`) around the current square `(i, j)`. This is done in a nested loop where we iterate over all adjacent cells including diagonals.
   - If there is at least one mine adjacent to `(i, j)`, we update the current square with the number of mines (`cnt`).
   - If there are no adjacent mines, we update the current square to `'B'`. We then recursively call `dfs` on all adjacent unrevealed squares (`'E'`) to continue revealing the board.
3. Before DFS is initiated, we first check the cell at the click coordinates. If the clicked cell is a mine (`'M'`), we update it to `'X'` to indicate the game is over. Otherwise, we call our `dfs` function with the click coordinates.
4. The `dfs` function is designed to stop the recursive calls when it encounters a cell that is not `'E'`, which handles both the case when revealing `'B'` squares, stopping at the border, and stopping at numbered squares.
5. The `dfs` function also ensures that we never go out of bounds of the board by checking whether the coordinates `(x, y)` are within the range `[0, m)` for rows and `[0, n)` for columns, where `m` and `n` are the lengths of the board's rows and columns respectively.
6. The `updateBoard` function then returns the updated board once all the possible and necessary cells are revealed based on the initial click.

This implementation ensures that all recursive calls to reveal cells only act on valid, in-bound, and appropriate cells, preventing unnecessary work and ensuring that the board state is mutated correctly according to Minesweeper rules.

Example Walkthrough

Let's go through a small example to illustrate the solution approach. Suppose we have a Minesweeper board like this, and the player clicks on the cell at row 1, column 2 (indexes are zero-based):

```
[['E', 'E', 'E', 'E'],
 ['E', 'E', 'E', 'M'],
 ['E', 'M', 'E', 'E'],
 ['E', 'E', 'M', 'E']]
```

We will follow the solution steps for our DFS approach using this board:

1. Initialize – We note the player's click on cell (1, 2), which is 'E'.
2. DFS Function Call – We check whether the clicked cell is 'M' or 'E'. Since it's 'E', we invoke our `dfs` function here.
3. Count Mines – We look at all the adjacent cells around (1, 2). The adjacent cells contain one mine at (2, 1). Therefore, the mine count `cnt` is 1.
4. Update Cell – Since there's one mine adjacent to (1, 2), we update the board as follows, replacing 'E' with '1' to indicate the number of adjacent mines:

```
[['E', 'E', 'E', 'E'],
 ['E', 'E', '1', 'M'],
 ['E', 'M', 'E', 'E'],
 ['E', 'E', 'M', 'E']]
```

5. Adjacent Squares – Since cell (1, 2) is next to only one mine, we don't perform recursive calls on its neighbors as per standard Minesweeper rules.
6. Final Board – The `updateBoard` function returns the board since we cannot reveal any further cells based on the click position.

This updated board represents the state after the single click, adhering to the rules of Minesweeper where numbered cells should not reveal their neighbors. If the initial click was on a different 'E' cell, especially one not adjacent to any mines, the recursive `dfs` would take effect, revealing a larger area by turning 'E' cells to 'B' and uncovering all interconnected blank spaces.

Python Solution

```python
from typing import List

class Solution:
    def updateBoard(self, board: List[List[str]], click: List[int]) -> List[List[str]]:
        # The recursive function to reveal the board starting from the clicked cell.
        def reveal(i: int, j: int):
            # Count mines around the current cell
            mine_count = 0
            for x in range(max(i - 1, 0), min(i + 2, rows)):  # Limits the range on the board
                for y in range(max(j - 1, 0), min(j + 2, columns)):  # Limits the range on the board
                    if board[x][y] == "M":
                        mine_count += 1

            # If there are mines around the cell, update with mine count
            if mine_count > 0:
                board[i][j] = str(mine_count)
            else:
                # Otherwise, set the cell to "B" for blank and reveal surrounding cells
                board[i][j] = "B"
                for x in range(max(i - 1, 0), min(i + 2, rows)):
                    for y in range(max(j - 1, 0), min(j + 2, columns)):
                        if board[x][y] == "E":
                            reveal(x, y)

        # Get the size of the board
        rows, columns = len(board), len(board[0])

        # The clicked position
        click_row, click_col = click

        # If the clicked cell contains a mine, game over
        if board[click_row][click_col] == "M":
            board[click_row][click_col] = "X"
        else:
            # Start revealing from the clicked cell
            reveal(click_row, click_col)

        # Return the updated board
        return board

```

Java Solution

```java
class Solution {
    private char[][] gameBoard; // Renamed board to gameBoard for clarity
    private int rows;           // Renamed m to rows for clarity
    private int cols;           // Renamed n to cols for clarity

    // Function updates the game board when a user clicks on a cell (i.e., uncovers the cell)
    public char[][] updateBoard(char[][] board, int[] click) {
        rows = board.length;        // Total number of rows in the board
        cols = board[0].length;     // Total number of columns in the board
        gameBoard = board;          // Assign board to gameBoard for internal use

        int clickRow = click[0], clickCol = click[1]; // Row and column indexes of the click position

        // Check if the clicked cell contains a mine
        if (gameBoard[clickRow][clickCol] == 'M') {
            gameBoard[clickRow][clickCol] = 'X'; // If there's a mine, mark the cell as 'X'
        } else {
            // If the clicked cell does not contain a mine, perform Depth First Search (DFS) from this cell
            dfs(clickRow, clickCol);
        }
        return gameBoard; // Return the updated board
    }

    // Performs Depth First Search (DFS) to reveal cells
    private void dfs(int row, int col) {
        int mineCount = 0; // Counter for adjacent mines

        // Iterate through the adjacent cells
        for (int x = row - 1; x <= row + 1; ++x) {
            for (int y = col - 1; y <= col + 1; ++y) {
                // Check if the adjacent cell is within the board and if it contains a mine
                if (x >= 0 && x < rows && y >= 0 && y < cols && gameBoard[x][y] == 'M') {
                    mineCount++; // Increment the mine counter
                }
            }
        }

        if (mineCount > 0) {
            // If there are mines found around the cell, display the mine count
            gameBoard[row][col] = (char) (mineCount + '0');
        } else {
            // If there are no mines around the cell, mark the cell as 'B' for blank
            gameBoard[row][col] = 'B';
            // Recursively reveal adjacent cells that are not mines and are unrevealed ('E')
            for (int x = row - 1; x <= row + 1; ++x) {
                for (int y = col - 1; y <= col + 1; ++y) {
                    if (x >= 0 && x < rows && y >= 0 && y < cols && gameBoard[x][y] == 'E') {
                        dfs(x, y); // Recursive call to dfs
                    }
                }
            }
        }
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    vector<vector<char>> updateBoard(vector<vector<char>>& board, vector<int>& click) {
        int rows = board.size(), cols = board[0].size();
        int clickRow = click[0], clickCol = click[1];

        // Using lambda function to perform Depth-First Search (DFS)
        function<void(int, int)> dfs = [&](int row, int col) {
            int mineCount = 0;

            // Count mines in adjacent cells
            for (int x = row - 1; x <= row + 1; ++x) {
                for (int y = col - 1; y <= col + 1; ++y) {
                    if (x >= 0 && x < rows && y >= 0 && y < cols && board[x][y] == 'M') {
                        ++mineCount;
                    }
                }
            }

            // If mines exist around the cell, update the cell with the mine count. Otherwise, mark the cell as 'B' and continue DF
            if (mineCount > 0) {
                board[row][col] = mineCount + '0';
            } else {
                board[row][col] = 'B';
                for (int x = row - 1; x <= row + 1; ++x) {
                    for (int y = col - 1; y <= col + 1; ++y) {
                        if (x >= 0 && x < rows && y >= 0 && y < cols && board[x][y] == 'E') {
                            dfs(x, y);
                        }
                    }
                }
            }
        };

        // Handle the initial click
        if (board[clickRow][clickCol] == 'M') {
            // If the clicked cell contains a mine, game over
            board[clickRow][clickCol] = 'X';
        } else {
            // If the clicked cell is empty ('E'), perform DFS to reveal cells
            dfs(clickRow, clickCol);
        }

        return board;
    }
};

```

Typescript Solution

```typescript
function updateBoard(board: string[][], click: number[]): string[][] {
    const rowCount = board.length; // Number of rows in the board
    const colCount = board[0].length; // Number of columns in the board
    const [rowClicked, colClicked] = click; // Destructure click into row and column indices

    // Deep-First Search function to reveal the board
    const revealBoard = (row: number, col: number) => {
        let mineCount = 0;
        // Count the number of mines surrounding the current cell
        for (let x = row - 1; x <= row + 1; ++x) {
            for (let y = col - 1; y <= col + 1; ++y) {
                if (x >= 0 && x < rowCount && y >= 0 && y < colCount && board[x][y] === 'M') {
                    mineCount++;
                }
            }
        }

        if (mineCount > 0) { // If mines are found around the cell, update the cell with the mine count
            board[row][col] = mineCount.toString();
        } else { // If no mines are found, mark the cell as 'B' and continue the search
            board[row][col] = 'B';
            for (let x = row - 1; x <= row + 1; ++x) {
                for (let y = col - 1; y <= col + 1; ++y) {
                    // Recursively reveal the board for surrounding cells
                    if (x >= 0 && x < rowCount && y >= 0 && y < colCount && board[x][y] === 'E') {
                        revealBoard(x, y);
                    }
                }
            }
        }
    };

    // Click processing
    if (board[rowClicked][colClicked] === 'M') {
        // If the click is on a mine, game is over. Mark the clicked mine with 'X'
        board[rowClicked][colClicked] = 'X';
    } else {
        // If the click is not on a mine, reveal the board from the clicked spot
        revealBoard(rowClicked, colClicked);
    }

    return board; // Return the updated board state
}

```

Time and Space Complexity

The given Python code is a solution for the Minesweeper game where a player clicks on a cell on the board and depending on the board cell content, different actions may be taken.

Time Complexity:

The time complexity is determined by the number of times `dfs` is invoked and how many cells it inspects each time. In the worst-case scenario, `dfs` performs a Depth-First Search on the entire board starting from the clicked cell.

- Each cell is visited at most once due to the fact that once a cell is visited it changes from `"E"` (empty) to `"B"` (blank) or it becomes a digit representing the number of adjacent mines.
- For each cell visited, `dfs` inspects up to 8 adjacent cells.

Given a board of size `m x n`, the time complexity is thus `O(m * n)` since in the worst-case scenario, we might end up visiting each cell once.

Space Complexity:

The space complexity of this DFS solution is primarily determined by the recursion stack used by the `dfs` function.

- In the worst-case scenario, the DFS could recurse to a depth where the entire board is traversed, making the maximum recursion depth `m * n` in a situation where all cells are empty ("E") and connected, requiring the algorithm to visit every cell before hitting a base case.

The space complexity is hence `O(m * n)` due to the recursion stack in the worst-case scenario.

When looking at auxiliary space (excluding space for inputs and outputs), the space complexity depends on the number of recursive calls placed on the stack, which remains `O(m * n)`.