1219. Path with Maximum Gold

Medium Array Backtracking Matrix

Leetcode Link

Problem Description

In this challenge, you are given a two-dimensional grid that represents a gold mine, with each cell containing a certain amount of gold or being empty (indicated by a `0`). The objective is to collect as much gold as possible with the following constraints:

- You can collect gold from any cell and start from there.
- You may move one step at a time in any of the four cardinal directions (left, right, up, or down).
- Once you leave a cell, you cannot visit it again; you collect the gold once only.
- You cannot step into a cell with zero gold.
- You can end your collection route at any cell containing gold.

Your task is to determine the maximum amount of gold that can be collected following these rules.

Intuition

To solve this problem, we need to explore every possible path that maximizes gold collection, which is a classic Depth-First Search (DFS) scenario. The intuition behind using DFS is that we want to explore all potential routes from each cell containing gold.

The process involves exploring a cell, collecting its gold, and then recursively checking all other paths from its adjacent cells. We keep track of the gold collected along each path and then backtracking, which means restoring the state before the DFS call, to ensure that we can explore other paths from the current cell.

Here's the approach:

1. Iterate through each cell in the grid.
2. If a cell has gold, initiate a DFS search from that cell.
3. Mark the cell as visited by setting its value to `0` to prevent revisiting, and collect the gold.
4. Explore all four directions recursively, summing up the gold from subsequent cells.
5. After exploring all directions from the current cell, backtrack by restoring the cell's gold value.
6. Keep track of the maximum gold collected during each DFS call.
7. Once all cells have been explored, return the maximum gold collected.

This approach exhaustively searches all possible paths and ensures that we find the maximum amount of gold that can be collected in the mine.

Solution Approach

The implementation uses Depth-First Search (DFS), a recursive algorithm that exhaustively searches through all possible paths in the grid. Here's a breakdown of how the DFS is implemented in the provided solution:

- A helper function `dfs(i, j)` is defined to perform the DFS search from the cell at row `i` and column `j`.
- The function first checks if the current cell `(i, j)` is out of bounds or if it contains `0` gold, in which case it returns `0` since no gold can be collected.
- If the cell contains gold, the function temporarily sets the gold amount to zero (`grid[i][j] = 0`). This serves as marking the cell as visited to avoid revisiting it in this path.
- The function then recursively explores the four adjacent cells (left, right, up, and down) by calling `dfs(i + a, j + b)` for each direction and adds the result to the current gold value. The `max()` function is used to choose the path that yields the maximum gold from these recursive calls.
- After exploring all directions, the function backtracks by resetting the gold value of the current cell (`grid[i][j] = v`) to ensure that future DFS calls can visit this cell from a different starting point.
- The main function iterates over every cell in the `grid` and calls `dfs(i, j)` if the cell contains gold, thereby initiating the DFS search from that cell.
- Each call to `dfs(i, j)` will return the maximum amount of gold that can be collected starting from that cell, and the overall maximum is updated accordingly.
- The built-in `max()` function is applied across all cells to find the single highest gold value collected.

The data structures used in this approach are:

- The input grid itself, which is a 2-dimensional list.
- Temporary variables for storing the current gold amount and for carrying out the DFS.

By using recursive DFS, the solution is able to explore all paths optimally without the need for additional data structures to track visited cells or paths, since the grid is modified in place and later restored during backtracking.

Example Walkthrough

Let's illustrate the solution approach using a small example of a gold mine grid:

```
1 4 2
3 0 0
```

We want to collect the maximum amount of gold following the given rules.

Here is a step-by-step walkthrough of the solution implemented with DFS:

1. We start our search with the cell at the top-left corner (0,0) which contains `1` gold. We initiate DFS from this cell.
2. We collect the `1` gold from this cell and mark it as visited by setting its value to `0`. Our grid now looks like this:

```
0 4 2
3 0 0
```

   The total gold collected is `1`.

3. From here, we can move right or down. Moving right, we find `4` gold. We collect it and mark the cell as visited, updating the grid:

```
1 0 0
3 0 0
```

   Now we have `4 + 2 = 6` gold.

4. Since we can't move into a cell with zero gold, we backtrack and restore the gold value of the cell (0,1). Our grid is back to:

```
1 4 2
3 0 0
```

   And we explore the next direction, which is down, finding `3` gold in cell (1,0).

5. We repeat the process for cell (1,0): collect the `3` gold, mark the cell as visited, and update the grid:

```
1 4 2
0 0 0
```

   The total gold now is `4 + 3 = 7`.

6. Now, we cannot move any further from cell (1,0), so we finish this path and record that we've collected `7` gold as our current maximum.
7. After that, we backtrack, restoring the grid, and continue the process for the remaining cells. The grid is restored to its previous state:

```
1 4 2
3 0 0
```

   Since there's only one other cell with gold we haven't started from (1,0), we move there and repeat the DFS process.

8. We try moving from cell (1,0) with `3` gold, but since the adjacent cells with gold have already been visited in previous iterations, there's no further gold to collect.
9. Our record shows the maximum gold collected was `7` gold from the path that started at (0,0), moved to (1,0), and then to (0,1).

So, the maximum amount of gold that can be collected in this example, following the DFS approach we outlined in the solution, is `7`.

Python Solution

```python
class Solution:
    def get_maximum_gold(self, grid):
        """
        Performs a depth-first search to find the maximum gold that can
        be collected starting from any point in the grid.

        :param grid: List[List[int]] representing the grid of cells with gold.
        :return: int representing the maximum amount of gold that can be collected.
        """

        def dfs(x, y):
            """
            Depth-first search helper function starting from cell (x, y).
            The idea is to try all possible paths and get the maximum gold.

            :param x: int representing the x-coordinate of the current cell.
            :param y: int representing the y-coordinate of the current cell.
            :return: int representing the collected gold amount via the path.
            """
            # Check if the current position is out of bounds or has no gold
            if not (0 <= x < rows and 0 <= y < cols and grid[x][y]):
                return 0

            # Store the gold at the current position and mark this cell as visited by setting the gold to 0
            original_gold = grid[x][y]
            grid[x][y] = 0

            # Calculate the gold gathered in all four directions
            total_gold = max(
                dfs(x + dx, y + dy) for dx, dy in directions
            )

            # Backtrack and reset the gold to original since we might visit this cell again via a different path
            grid[x][y] = original_gold
            return total_gold

        # Initialize the number of rows and columns of the grid
        rows, cols = len(grid), len(grid[0])

        # Define all four possible directions to move on the grid, which are right, left, up, and down
        directions = (0, 1), (0, -1), (-1, 0), (1, 0)

        # Using a grid comprehension, perform depth-first search starting from each cell
        # and return the maximum gold found in all the paths.
        return max(dfs(x, y) for x in range(rows) for y in range(cols))

# The modified class and function names now follow PEP 8 naming conventions.
```

Java Solution

```java
class Solution {
    private int[][] grid; // Holds the grid with gold amounts
    private int rows;     // Number of rows in the grid
    private int cols;     // Number of columns in the grid

    // Computes the maximum gold that can be collected
    public int getMaximumGold(int[][] grid) {
        // Initialize rows and cols based on the input grid size
        rows = grid.length;
        cols = grid[0].length;
        this.grid = grid; // Assign the grid
        int maxGold = 0;  // Initialize maximum gold collected

        // Iterate over all cells of the grid
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // Update maxGold with the maximum of the current maxGold or the gold collected by DFS from this cell
                maxGold = Math.max(maxGold, dfs(i, j));
            }
        }
        return maxGold; // Return the maximum gold collected
    }

    // Helper method for depth-first search
    private int dfs(int i, int j) {
        // Base case: if the cell is out of the grid bounds or has 0 gold, return 0
        if (i < 0 || i >= rows || j < 0 || j >= cols || grid[i][j] == 0) {
            return 0;
        }
        // Store gold value of the current cell
        int gold = grid[i][j];
        // Mark the current cell as visited by setting gold to 0
        grid[i][j] = 0;

        // Array to facilitate iteration over the adjacent cells (up, right, down, left)
        int[] directions = {-1, 0, 1, 0, -1};
        int maxGold = 0; // Initialize max gold collected from this cell

        // Iterate over all adjacent cells
        for (int k = 0; k < 4; ++k) {
            // Calculate the gold collected by DFS of the adjacent cells and update maxGold accordingly
            maxGold = Math.max(maxGold, gold + dfs(i + directions[k], j + directions[k + 1]));
        }
        // Backtrack: reset the value of the cell to the original gold amount
        grid[i][j] = gold;
        // Return the maximum gold that can be collected from this cell
        return maxGold;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm>  // include algorithm library for using max function

class Solution {
public:
    // Directions array representing the 4 possible movements: up, right, down, left.
    std::vector<int> directions = {-1, 0, 1, 0, -1};

    // Main function to call for getting the maximum gold.
    int getMaximumGold(std::vector<std::vector<int>>& grid) {
        int maxGold = 0;

        // Iterate over each cell in the grid.
        for (int i = 0; i < grid.size(); ++i) {
            for (int j = 0; j < grid[0].size(); ++j) {
                // Update maxGold by calculating starting from the current cell.
                maxGold = std::max(maxGold, dfs(i, j, grid));
            }
        }

        return maxGold;
    }

private:
    // Helper DFS function to explore the grid for gold collection.
    int dfs(int x, int y, std::vector<std::vector<int>>& grid) {
        // Boundary check and if cell with zero gold check
        if (x < 0 || x >= grid.size() || y < 0 || y >= grid[0].size() || grid[x][y] == 0) {
            return 0;
        }

        int gold = grid[x][y]; // Store the gold in the current cell.
        grid[x][y] = 0;        // Mark the current cell as visited by setting it to 0.

        int maxGoldFromHere = 0;

        // Explore all 4 adjacent cells.
        for (int k = 0; k < 4; ++k) {
            // Recursively collect the gold by going in one direction.
            // Update maxGoldFromHere with the max gold collected.
            int nextGold = dfs(x + directions[k], y + directions[k + 1], grid);
            maxGoldFromHere = std::max(maxGoldFromHere, gold + nextGold);
        }

        grid[x][y] = gold;  // Reset the cell to its original gold value.

        return maxGoldFromHere;  // Return the maximum gold obtained starting from (x, y).
    }
};
```

Typescript Solution

```typescript
/**
 * Retrieves the maximum amount of gold by visiting cells in the grid.
 * Each cell can be visited only once, and we must avoid cells with 0 gold.
 *
 * @param {number[][]} grid - The grid of cells containing gold amounts.
 * @return {number} - The maximum amount of gold that can be collected.
 */
function getMaximumGold(grid: number[][]): number {
    const rowCount: number = grid.length;
    const colCount: number = grid[0].length;

    // The depth-first search function explores all possible paths from the current cell.
    function dfs(row: number, col: number): number {
        // Boundary check and gold check, return 0 if it's out of bounds or cell is empty
        if (row < 0 || row >= rowCount || col < 0 || col >= colCount || grid[row][col] === 0) {
            return 0;
        }

        // Temporarily set the current cell to 0 to mark as visited
        const currentGold: number = grid[row][col];
        grid[row][col] = 0;

        let maxGoldFromHere: number = 0;

        // Directions: up, right, down, left
        const directions: number[] = [-1, 0, 1, 0, -1];

        // Explore in all four directions
        for (let k: number = 0; k < 4; ++k) {
            const nextRow: number = row + directions[k];
            const nextCol: number = col + directions[k + 1];

            // Accumulate the max gold along this path
            maxGoldFromHere = Math.max(maxGoldFromHere, currentGold + dfs(nextRow, nextCol));
        }

        // Reset the current cell's value after exploring all paths
        grid[row][col] = currentGold;

        return maxGoldFromHere;
    }

    // Initialize the answer to 0; it will store the maximum gold found
    let maxGold: number = 0;

    // Start from each cell in the grid to find the maximum gold
    for (let row: number = 0; row < rowCount; ++row) {
        for (let col: number = 0; col < colCount; ++col) {
            maxGold = Math.max(maxGold, dfs(row, col));
        }
    }

    return maxGold;
}
```

Time and Space Complexity

The time complexity of the provided code is calculated based on several factors:

- The algorithm performs a depth-first search (DFS) starting from each cell that has a positive gold value.
- Each DFS explores four possible directions to go next (except when on the border or if a cell is already visited).

Let's denote `m` as the number of rows and `n` as the number of columns in the `grid`. In the worst case, the entire grid could be filled with positive gold values, requiring an exhaustive search from each cell.

For each DFS, the maximum depth could be `min(m*n, 4^(m+n))` if the path is allowed to traverse every cell. However, due to path restrictions (not revisiting a cell with no gold), the maximum depth will be less than or equal to `m*n`. Each time we explore a cell, there are at most 4 directions to explore next.

Therefore, an upper bound of the time complexity is `O(4^(m+n))`, which is exponential due to the recursive exploration with branching.

For space complexity:

- The space used by the call stack during the recursive DFS would be equal to the maximum depth of the recursion which, in the worst case, is `O(m*n)` since we might have to traverse the entire grid if it is all filled with gold.
- We are also modifying the input grid to mark the cells as visited by temporarily setting `grid[i][j] = 0` and resetting it back to its original value before returning from the DFS. This in-place modification means we don't use extra space proportional to the grid size (except the implicit recursion stack).

Thus, the space complexity is `O(m*n)`.