764. Largest Plus Sign

Medium Array Dynamic Programming

Leetcode Link

Problem Description

You are tasked with finding the largest '+' shape that can be formed in a binary grid, where '1's represent a potential part of the '+' and '0's represent a space where part of a '+' cannot exist. The grid is `n x n` in size, and originally it's filled with '1's. However, there are some coordinates provided in the `mines` array which represent where '0s' will be, thus creating obstacles for forming the '+'. The order of the plus sign is defined as the length from its center to its edge, which means the total size will be related to the order as `order*2 - 1`. The goal is to return the highest order of a '+' sign found in this grid. If no '+' can be formed, the function should return 0.

Intuition

When tasked with finding the largest axis-aligned plus sign of '1's in a modified binary grid, we need to think about what defines a plus sign. For a plus sign to exist at any location, there must be uninterrupted sequences of '1's extending to the left, right, up, and down from that center '1'.

The intuition behind finding the largest plus sign involves assessing each potential center for how far in each of the four directions (up, down, left, and right) we can go before hitting a '0' or the edge of the grid.

We use dynamic programming to avoid redundant calculations. A matrix `dp` of the same dimensions as `grid` is created. Each element in `dp` will eventually represent the largest order of a plus sign that can have that cell as its center.

We initialize `dp` with 'n', representing the maximum possible order for a plus sign in an `n x n` grid. We then iterate through the indices in the `mines` array, setting the corresponding `dp` cell to '0', indicating an obstacle.

Next, we make four passes through the grid for each cell `(i, j)`:

- From left to right: if there's no obstacle, we increase the count `left`, otherwise we start the count over. We then set `dp[i][j]` to the minimum of its current value (which will be at least `left`).
- From right to left, following the same logic with the variable `right`.
- From top to bottom and bottom to top: the same process follows for the variables `up`, and `down`.

After this process, we know for any cell in `dp`, it holds the minimum possible extension in all four directions, effectively giving us the order of the largest plus sign centered at that cell.

Finally, we look for the maximum value in the `dp` array. This value is the order of the largest axis-aligned plus sign of '1's in the grid.

Solution Approach

To implement the solution for finding the largest plus sign, we utilize the dynamic programming technique that involves creating a 2D array called `dp` that serves as a table to store intermediate results. This is crucial because it will help in reducing the complexity by storing the achievable order of a plus sign at each (i, j) position, thus avoiding recomputation.

Here's the step-by-step approach:

1. Initialize the `dp` matrix, with `n` rows and `n` columns, setting every cell to `n`. This pre-population assumes the plus sign can extend to the edge of the grid in every direction from any cell.

2. Iterate over the `mines` list, which contains the positions that must be set to '0'. The `mines` list modifies the `dp` matrix by setting `dp[x][y] = 0` for every `[x, y]` pair, showing that no plus sign can be centered at these 'mined' cells.

3. To calculate the order of the largest plus sign that can be centered at each cell, iterate over each row and column four times, once for each direction: left, right, up, and down.

   - For left and rightward checks, we loop through each row, once from left to right and then in the reverse direction. If `dp[i][j]` is not zero, we increment `left` or `right` as appropriate, showing that the arm of the plus can extend in that direction.

   - Similarly, for upwards and downwards, we loop through each column in both top-to-bottom and bottom-to-top directions, incrementing `up` or `down` if `dp[j][i]` is not zero.

   - At each step, we update the `dp` matrix by assigning the minimum value amongst the current `dp` value and the 'left', 'right', 'up', or 'down' counters.

This process ensures that `dp[i][j]` holds the order of the largest plus sign for which the cell (i, j) can be the center, considering the constraints in all four directions.

4. After the conclusion of the loops, scan through the `dp` matrix to find the maximum value. This value is the order of the largest plus sign that can be found in the grid.

The specific data structure used here is a 2D list (list of lists) for the `dp` table, enabling direct access to each cell for its up, down, left, right sequence calculations.

In terms of the algorithm, the solution employs an efficient linear scan of rows and columns, taking `O(n^2)` time — a significant optimization over brute force approaches that would require `O(n^3)` or greater. It effectively combines the principles of dynamic programming with those of matrix traversal, effectively reducing the problem to simple arithmetic and comparisons.

Example Walkthrough

Let's say we have a 3×3 grid (`n = 3`) and the `mines` array is `[[1, 1]]`, meaning there will be a '0' in the center of the grid.

Here's a walkthrough of applying the solution step-by-step to this grid:

1. We initialize the `dp` matrix to represent the maximum possible order of the plus sign in a 3×3 grid:

```
dp = [[3, 3, 3],
      [3, 3, 3],
      [3, 3, 3]]
```

2. We then apply the `mines`: since we have a mine at `[1, 1]`, we set `dp[1][1]` to `0`, indicating no plus can be centered here.

```
dp = [[3, 3, 3],
      [3, 0, 3],
      [3, 3, 3]]
```

3. Next, we make passes to calculate the left, right, up, and down order for each cell:

   - **Left to right pass**: We count the number of consecutive '1's from the left for each cell in a row and update the `dp` value.

     After the left to right pass:

     ```
     dp = [[1, 2, 3],
           [1, 0, 3],
           [1, 2, 3]]
     ```

   - **Right to left pass**: Similar to above, but from the right.

     After the right to left pass:

     ```
     dp = [[1, 2, 3],
           [1, 0, 1],
           [1, 2, 3]]
     ```

     (We only update if the value of the current cell is greater than the count.)

   - **Top to bottom pass**: We do the same vertically.

     After top to bottom pass:

     ```
     dp = [[1, 1, 1],
           [1, 0, 1],
           [1, 2, 1]]
     ```

   - **Bottom to top pass**: And finally, from bottom to top.

     After bottom to top pass:

     ```
     dp = [[1, 1, 1],
           [1, 0, 1],
           [1, 1, 1]]
     ```

4. Now `dp` contains the order of the largest '+' that can be formed with each cell as its center. We scan through `dp` to find the maximum value which will give us the order of the largest '+'. In our example:

   - The maximum value in `dp` is `1`, which means the largest possible plus sign order is `1`, and the size of this plus sign would be `1*2 - 1 = 1`. Hence, the largest plus sign we can form has arms of length `1`.

To summarize, for this 3×3 grid with a mine at the center, the largest '+' sign we can form has an order of `1`, extending a single cell in each direction from its center.

Python Solution

```python
from typing import List

class Solution:
    def orderOfLargestPlusSign(self, n: int, mines: List[List[int]]) -> int:
        # Initialize a 2D DP array with n rows and columns filled with value n.
        # This represents the maximum size of the plus sign at each cell assuming no mines are present.
        dp = [[n] * n for _ in range(n)]

        # Set each mine location in the DP array to 0 because these locations cannot be part of a plus sign.
        for mine in mines:
            x, y = mine
            dp[x][y] = 0

        # Calculate the order of the largest plus sign considering mines.
        for i in range(n):
            # Initialize counts for the left, right, up, and down arms of the plus sign.
            left = right = up = down = 0
            # Iterate over rows and columns in both the forward and reverse direction.
            for j, k in zip(range(n), reversed(range(n))):
                # If there's no mine, increment counts; else reset to 0.
                left = left + 1 if dp[i][j] else 0
                right = right + 1 if dp[i][k] else 0
                up = up + 1 if dp[j][i] else 0
                down = down + 1 if dp[k][i] else 0

                # Update the DP array with the minimum arm length seen so far for each direction.
                # This ensures that the plus sign is only as long as the shortest arm length.
                dp[i][j] = min(dp[i][j], left)
                dp[i][k] = min(dp[i][k], right)
                dp[j][i] = min(dp[j][i], up)
                dp[k][i] = min(dp[k][i], down)

        # Find the largest value in the DP array, which corresponds to the order of the largest plus sign.
        return max(max(row) for row in dp)

```

Java Solution

```java
import java.util.Arrays;

public class Solution {

    public int orderOfLargestPlusSign(int n, int[][] mines) {
        // Create a DP array and initialize each cell with the maximum size n
        int[][] dp = new int[n][n];
        for (int[] row : dp) {
            Arrays.fill(row, n);
        }

        // Set cells that are mines to 0
        for (int[] mine : mines) {
            dp[mine[0]][mine[1]] = 0;
        }

        // Iterate over the matrix to find the maximal order of the plus sign at each point
        for (int i = 0; i < n; ++i) {
            int left = 0, right = 0, up = 0, down = 0;
            for (int j = 0, k = n - 1; j < n; ++j, --k) {
                // Count continuous ones from the left to right in a row
                left = dp[i][j] > 0 ? left + 1 : 0;
                // Count continuous ones from the right to left in a row
                right = dp[i][k] > 0 ? right + 1 : 0;
                // Count continuous ones from top to bottom in a column
                up = dp[j][i] > 0 ? up + 1 : 0;
                // Count continuous ones from bottom to top in a column
                down = dp[k][i] > 0 ? down + 1 : 0;

                // Update DP for the arms length of the plus sign, consider previous computations too
                dp[i][j] = Math.min(dp[i][j], left);
                dp[i][k] = Math.min(dp[i][k], right);
                dp[j][i] = Math.min(dp[j][i], up);
                dp[k][i] = Math.min(dp[k][i], down);
            }
        }

        // Find the largest plus sign order from the DP array
        int maxOrder = Arrays.stream(dp)
            .flatMapToInt(Arrays::stream)
            .max()
            .getAsInt();

        return maxOrder;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // This function calculates the order of the largest plus sign that can be found
    // in a square binary grid. Mines are represented by zeros in the grid.
    int orderOfLargestPlusSign(int n, vector<vector<int>>& mines) {
        // Initialize a dp (dynamic programming) matrix filled with 'n' which is the maximum possible arm length
        vector<vector<int>> dpMatrix(n, vector<int>(n, n));

        // Setting up the mined locations in the dp matrix to 0
        for (auto& mine : mines) {
            dpMatrix[mine[0]][mine[1]] = 0;
        }

        // Iterate over the matrix to calculate the arm length of the plus sign in all directions
        for (int i = 0; i < n; ++i) {
            int left = 0, right = 0, up = 0, down = 0;
            for (int j = 0, k = n - 1; j < n; ++j, --k) {
                // Calculate continuous arm length for left and right directions
                left = (dpMatrix[i][j] != 0) ? left + 1 : 0;
                right = (dpMatrix[i][k] != 0) ? right + 1 : 0;

                // Calculate continuous arm length for up and down directions
                up = (dpMatrix[j][i] != 0) ? up + 1 : 0;
                down = (dpMatrix[k][i] != 0) ? down + 1 : 0;

                // The minimum of the 4 directions should be taken for the true arm length
                dpMatrix[i][j] = min(dpMatrix[i][j], left);
                dpMatrix[i][k] = min(dpMatrix[i][k], right);
                dpMatrix[j][i] = min(dpMatrix[j][i], up);
                dpMatrix[k][i] = min(dpMatrix[k][i], down);
            }
        }

        // Find the maximum arm length from the dp matrix, which is the order of the largest plus sign
        int maxOrder = 0;
        for (auto& row : dpMatrix) {
            maxOrder = max(maxOrder, *max_element(row.begin(), row.end()));
        }
        return maxOrder;
    }
};

```

Typescript Solution

```typescript
function orderOfLargestPlusSign(n: number, mines: number[][]): number {
    // Initialize a dp (dynamic programming) matrix filled with 'n', the maximum possible arm length
    const dpMatrix: number[][] = Array.from({ length: n }, () => Array(n).fill(n));

    // Setting up the mined locations in the dp matrix to 0
    mines.forEach(mine => {
        dpMatrix[mine[0]][mine[1]] = 0;
    });

    // Iterate over the matrix to calculate the arm length of plus signs in all directions
    for (let i = 0; i < n; i++) {
        let left = 0, right = 0, up = 0, down = 0;
        for (let j = 0, k = n - 1; j < n; j++, k--) {
            // Calculate continuous arm length for left and right directions
            left = (dpMatrix[i][j] !== 0) ? left + 1 : 0;
            right = (dpMatrix[i][k] !== 0) ? right + 1 : 0;

            // Calculate continuous arm length for up and down directions
            up = (dpMatrix[j][i] !== 0) ? up + 1 : 0;
            down = (dpMatrix[k][i] !== 0) ? down + 1 : 0;

            // The minimum of the 4 directions should be taken for the true arm length
            dpMatrix[i][j] = Math.min(dpMatrix[i][j], left);
            dpMatrix[i][k] = Math.min(dpMatrix[i][k], right);
            dpMatrix[j][i] = Math.min(dpMatrix[j][i], up);
            dpMatrix[k][i] = Math.min(dpMatrix[k][i], down);
        }
    }

    // Find the maximum arm length from the dp matrix, which is the order of the largest plus sign
    let maxOrder = 0;
    dpMatrix.forEach((row) => {
        maxOrder = Math.max(maxOrder, Math.max(...row));
    });
    return maxOrder;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed by breaking down each part of the code:

1. Initializing the `dp` matrix with the size `n x n` takes `O(n^2)` time.
2. The first loop to set the mines takes `O(m)` time, where `m` is the number of mines.
3. The nested loops to calculate the `left`, `right`, `up`, and `down` values take `O(n^2)` time. For each of the `n` rows and `n` columns, the loop runs `n` times.
4. The final nested loops also run in `O(n^2)` time, since it involves traversing the `dp` matrix again to calculate the minimum for each position and also to find the maximum order for plus sign at the end.

Overall, the time complexity is dominated by the nested loops, so the final time complexity is `O(n^2)`.

Space Complexity

The space complexity is determined by the extra space used by the algorithm, which is:

1. The `dp` matrix that requires `O(n^2)` space.
2. Four variables `left`, `right`, `up`, and `down` are used but they only add `O(1)` space.

Thus, the space complexity of the algorithm is `O(n^2)`.