2397. Maximum Rows Covered by Columns

Medium · Bit Manipulation · Array · Backtracking · Enumeration · Matrix   Leetcode Link

Problem Description

The problem provides us with a binary matrix where rows represent different items, and columns represent attributes that are either present (1) or absent (0). We are also given a number `numSelect` which indicates how many columns we can choose.

A row is said to be covered by the selection if for each "1" in the row, the corresponding column is part of the chosen columns, or if the row does not contain any "1"s. The goal is to select `numSelect` columns in such a way that the maximum number of rows are covered.

The challenge is to examine all possible combinations of columns that can be selected and then determine which combination covers the most rows.

Intuition

The intuition behind the solution is to use bit manipulation to represent the presence or absence of columns in a more computationally efficient manner. Each row of the matrix can be represented by a bitmask where the bit is set to 1 if the corresponding column contains a 1. Selecting `numSelect` columns is equivalent to creating a bitmask with `numSelect` number of 1s, representing the columns being selected.

The problem is then reduced to iterating over all possible combinations of selected columns (which are represented by bitmasks with exactly `numSelect` 1s). For each combination, we determine how many rows are covered by using a logical "AND" operation. A row is covered if, after the "AND" operation between the row's bitmask and the chosen columns' bitmask, the result is equal to the row's bitmask.

To count the number of 1s in a bitmask (or check if a bitmask has exactly `numSelect` 1s), the `bit_count()` function in Python is utilized. The `max` function is then used to keep track of the maximum number of rows that have been covered so far.

Through this approach using bitmasks and iteration, we can find the optimal selection of columns without explicitly checking each element of the subset in the original matrix, saving both time and space, and providing an optimized solution to the problem.

Solution Approach

The solution approach relies on bit manipulation and enumeration. Let's walk through the implementation as provided in the reference solution with particular emphasis on the algorithms, data structures, and patterns used:

1. We start by representing each row as a bitmask. This is done by iterating over each row in the original matrix and for every "1" encountered, a corresponding bit in the mask is set. This uses list comprehension along with the `reduce` function from `functools` and the `or_` bitwise operator from `operator`.

   ```python
   rows = []
   for row in matrix:
       mask = reduce(or_, (1 << j for j, x in enumerate(row) if x), 0)
       rows.append(mask)
   ```

2. Next, we want to enumerate all possible combinations of columns we can choose. As each column can be represented by a bit in a bitmask, we iterate over the range `0` to `2^n` (where `n` is the number of columns), using a `for` loop. Each number in this range corresponds to a potential combination of columns, with the bit at position j representing column j.

   ```python
   for mask in range(1 << len(matrix[0])):
       if mask.bit_count() != numSelect:
           continue
       # ... Check coverage for this combination
   ```

3. Within this loop, we use the `bit_count` method on the bitmask to check whether the number of columns chosen (`numSelect`) is equal to the number of 1s in the bitmask. If not, we skip to the next iteration:

   ```python
   if mask.bit_count() != numSelect:
       continue
   ```

4. For each valid combination of selected columns, we check how many rows are covered. This is done with a neat one-liner list comprehension by iterating over each row's bitmask and checking if the row's bitmask `AND` the selected column bitmask equals the row's bitmask, meaning the row is covered:

   ```python
   t = sum((x & mask) == x for x in rows)
   ```

5. Finally, we update our answer with the maximum value between the current answer and the number of covered rows for this column selection:

   ```python
   ans = max(ans, t)
   ```

6. Return the maximum number of rows covered after examining all combinations:

   ```python
   return ans
   ```

The overall pattern is an exhaustive search where we iterate over all possible selections of columns and, for each selection, we count how many rows are covered. This brute-force approach is made tractable by using bitmasks to efficiently represent and compare sets of columns and rows.

By compactly representing columns and rows as bits, we avoid dealing with actual column elements and row elements directly, which reduces computational complexity. However, since the number of possible combinations is `2^n`, this approach is still exponential in time complexity, but it's practical for small input sizes where n (the number of columns) is not too large.

The key data structures here:

- A list `rows` that holds the bitmask representation of each row.
- An integer `mask` that holds the bitmask representation of each possible column selection.
- An integer `ans` that keeps track of the maximum number of rows that can be covered.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider a binary matrix `matrix` and `numSelect` as follows:

```python
matrix = [
    [1, 0, 0],
    [0, 1, 1],
    [1, 1, 0]
]
numSelect = 2
```

Our task is to choose 2 columns that cover the maximum number of rows based on the given rules.

1. First, we convert each row of the matrix to a bitmask representation, where each "1" present in the row corresponds to a set bit in the bitmask:
   - Row 1 bitmask = `100` (in binary) = `4` (in decimal)
   - Row 2 bitmask = `011` (in binary) = `3` (in decimal)
   - Row 3 bitmask = `110` (in binary) = `6` (in decimal)
2. The list of bitmasks corresponding to the rows will be `rows = [4, 3, 6]`.
3. Next, we need to generate all possible combinations of the columns that we can select and check if they cover the rows. Since `numSelect = 2`, we are looking for bitmasks with exactly 2 set bits. Our matrix has 3 columns, so we iterate over the range `0` to `2^3` or `0` to `8` in decimal.
4. The possible column combinations with 2 set bits are `110` (`6` in decimal), `101` (`5` in decimal) and `011` (`3` in decimal). These correspond to selecting columns `[1, 2]`, `[1, 3]`, and `[2, 3]`, respectively.
5. Now, check each valid column bitmask against all the row bitmasks. For each combination:
   - Consider `mask = 6` (selecting columns `[1, 2]`):
     - Row 1 (`4`): `(4 & 6) == 4` is `True` (covered)
     - Row 2 (`3`): `(3 & 6) == 3` is `True` (covered)
     - Row 3 (`6`): `(6 & 6) == 6` is `True` (covered)

     This selection covers all 3 rows.
   - Consider `mask = 5` (selecting columns `[1, 3]`):
     - Row 1 (`4`): `(4 & 5) == 4` is `True` (covered)
     - Row 2 (`3`): `(3 & 5) == 1` is `False` (not covered)
     - Row 3 (`6`): `(6 & 5) == 4` is `False` (not covered)

     This selection covers 1 row.
   - Consider `mask = 3` (selecting columns `[2, 3]`):
     - Row 1 (`4`): `(4 & 3) == 0` is `False` (not covered)
     - Row 2 (`3`): `(3 & 3) == 3` is `True` (covered)
     - Row 3 (`6`): `(6 & 3) == 2` is `False` (not covered)

     This selection covers 1 row.
6. The maximum number of rows covered by any selection is `3` (from the bitmask `6`, which represents selecting columns `[1, 2]`).
7. Therefore, the output for this example would be `3` as it is the maximum number of rows that can be covered by selecting 2 columns.

Python Solution

```python
from functools import reduce
from operator import or_

class Solution:
    def maximumRows(self, matrix, num_select):
        # Convert each row of the matrix to a bitmask where 1's represent columns
        # that have the value 1. This will allow us to easily compare which rows
        # can be entirely covered by selecting certain columns.
        row_masks = [reduce(or_, (1 << j for j, cell in enumerate(row) if cell), 0) for row in matrix]

        max_covered_rows = 0  # Initialize the max number of rows that can be covered

        # Iterate over all possible selections of columns as bitmasks.
        for col_mask in range(1 << len(matrix[0])):
            # Check if the number of selected columns matches the required `num_select`
            if bin(col_mask).count('1') != num_select:
                continue

            # Count the number of rows that can be entirely covered by the selected columns
            covered_row_count = sum(1 for row_mask in row_masks if (row_mask & col_mask) == row_mask)

            # Update the maximum number of rows that can be covered
            max_covered_rows = max(max_covered_rows, covered_row_count)

        return max_covered_rows

```

Java Solution

```java
class Solution {
    public int maximumRows(int[][] matrix, int numSelect) {
        int rowCount = matrix.length; // total number of rows in the matrix
        int colCount = matrix[0].length; // total number of columns in the matrix
        int[] rowBitmasks = new int[rowCount]; // array to store the bitmask representation of each row

        // Convert every row in the matrix to their respective bitmask representation
        for (int rowIndex = 0; rowIndex < rowCount; ++rowIndex) {
            for (int colIndex = 0; colIndex < colCount; ++colIndex) {
                // If the cell contains a 1, update the bitmask for that row
                if (matrix[rowIndex][colIndex] == 1) {
                    rowBitmasks[rowIndex] |= 1 << colIndex;
                }
            }
        }

        int maxRowsCompleted = 0; // variable to store the maximum number of rows that can be completed

        // Iterate over all possible combinations of columns to select
        for (int mask = 1; mask < 1 << colCount; ++mask) {
            // Continue only if the bit count of 'mask' equals 'numSelect'
            if (Integer.bitCount(mask) != numSelect) {
                continue;
            }

            int completedRows = 0; // variable to count how many rows are "completed" with the current mask

            // Check each row to see if the selected columns can complete the row
            for (int rowBitmask : rowBitmasks) {
                // A row is considered complete if the columns selected by 'mask' cover all the 1s in the row
                if ((rowBitmask & mask) == rowBitmask) {
                    completedRows++; // increment the number of completed rows
                }
            }

            // Update the maximum number of rows that we can complete with the current combination
            maxRowsCompleted = Math.max(maxRowsCompleted, completedRows);
        }

        return maxRowsCompleted; // return the maximum number of rows that can be completed
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>
#include <cstring>

class Solution {
public:
    // Returns the maximum number of rows where all '1's are covered when choosing 'numSelect' columns
    int maximumRows(vector<vector<int>>& matrix, int numSelect) {
        int rowCount = matrix.size(); // Number of rows in the matrix
        int colCount = matrix[0].size(); // Number of columns in the matrix
        vector<int> rowMasks(rowCount, 0); // Vector to store the bitmask representation of each row

        // Convert each row of the matrix into a bitmask and store it in rowMasks
        for (int i = 0; i < rowCount; ++i) {
            for (int j = 0; j < colCount; ++j) {
                if (matrix[i][j]) {
                    rowMasks[i] |= 1 << j;
                }
            }
        }

        int maxRowsCovered = 0; // Variable to keep track of the maximum rows covered

        // Iterate through all possible combinations of selected columns
        for (int mask = 1; mask < (1 << colCount); ++mask) {
            // If the number of selected columns doesn't match 'numSelect', skip this combination
            if (__builtin_popcount(mask) != numSelect) {
                continue;
            }

            int currentCovered = 0; // Counter for the number of rows fully covered in this combination

            // Check each row to see if it is fully covered by the selected columns
            for (int rowMask : rowMasks) {
                // If the intersection of the row bitmask and selected columns equals the row bitmask,
                // it means all '1's in that row are covered
                if ((rowMask & mask) == rowMask) {
                    currentCovered++;
                }
            }

            // Update the maximum rows covered if the current configuration covers more rows
            maxRowsCovered = max(maxRowsCovered, currentCovered);
        }

        // Return the maximum number of rows that can be covered
        return maxRowsCovered;
    }
};

```

Typescript Solution

```typescript
function maximumRows(matrix: number[][], numSelect: number): number {
    const rowCount = matrix.length; // Number of rows in the matrix
    const colCount = matrix[0].length; // Number of columns in the matrix
    const rowMasks: number[] = new Array(rowCount).fill(0); // Array to store the bitmask representation of each row

    // Convert each row of the matrix into a bitmask and store it in rowMasks
    for (let i = 0; i < rowCount; ++i) {
        for (let j = 0; j < colCount; ++j) {
            if (matrix[i][j]) {
                rowMasks[i] |= 1 << j;
            }
        }
    }

    let maxRowsCovered = 0; // Variable to keep track of the maximum rows covered

    // Iterate through all possible combinations of selected columns
    for (let mask = 1; mask < (1 << colCount); ++mask) {
        // If the number of selected columns doesn't match 'numSelect', skip this combination
        if (popCount(mask) !== numSelect) {
            continue;
        }

        let currentCovered = 0; // Counter for the number of rows fully covered in this combination

        // Check each row to see if it is fully covered by the selected columns
        for (let rowMask of rowMasks) {
            // If the intersection between the row bitmask and selected columns equals the row bitmask,
            // it means all '1's in that row are covered
            if ((rowMask & mask) === rowMask) {
                currentCovered++;
            }
        }

        // Update the maximum rows covered if the current configuration covers more rows
        maxRowsCovered = Math.max(maxRowsCovered, currentCovered);
    }

    // Return the maximum number of rows that can be covered
    return maxRowsCovered;
}

// Helper function to count the number of '1's in binary representation of a number
function popCount(n: number): number {
    let count = 0;
    while (n) {
        count += n & 1;
        n >>>= 1;
    }
    return count;
}

```

Time and Space Complexity

The time complexity of the given code snippet primarily arises from the nested loops. The number of columns in the matrix is denoted as `C = len(matrix[0])` and the number of rows is `R = len(matrix)`. The for loop through every row has a complexity of `O(R)`, and within that loop, it iterates through all columns, incurring a complexity of `O(C)` resulting in a total of `O(R * C)` for this segment.

Additionally, the code iterates through every possible combination of columns (`1 << C` possibilities), with a complexity of `O(2^C)`. Within that loop, for every mask, we iterate over all the rows to perform bitmask comparisons, resulting in an `O(R)` complexity for this part.

Combining these, the overall time complexity is `O(R * C + R * 2^C)`, which simplifies to `O(R * (C + 2^C))`.

The space complexity of this code is `O(R)`. This arises due to the additional `rows` list storing a mask for each row. Each mask is an integer and there are `R` masks to store. The other variables used in the code are of constant size and therefore do not significantly add to the complexity.