311. Sparse Matrix Multiplication

Medium | Topics | Hash Table | Matrix

Leetcode Link

Problem Description

The problem at hand requires us to perform the multiplication of two sparse matrices. A sparse matrix is a matrix that is comprised mostly of zero values. The input consists of two matrices `mat1` and `mat2` where `mat1` is of size $m \times k$ and `mat2` is of size $k \times n$. The task is to compute the matrix product of `mat1` and `mat2` and return the resulting matrix. It's assumed that the multiplication of the given inputs is always valid, which means that the number of columns in `mat1` is equal to the number of rows in `mat2`. The output should also be in the form of a sparse matrix where the spaces are optimized to store only non-zero values as much as possible.

Intuition

The intuitive approach to matrix multiplication involves three nested loops, iterating through each row of the first matrix and each column of the second matrix to calculate the values of the resulting matrix. However, since we are given a condition that these are sparse matrices, many of these calculations will involve multiplication by zero, which is unnecessary and can be skipped to save computation time and space.

The idea behind the solution approach is to preprocess each input matrix to filter out all the zero values and keep track of the non-zero values along with their positions. This preprocessing step helps to perform multiplication operations only when it is needed (i.e., only with the non-zero values).

The function `find()` in the provided solution converts a matrix into a list of lists, where each sublist contains tuples. Each tuple has two elements: the column index and the value of the non-zero element in the matrix. This effectively compresses the matrix to store only the necessary information needed for the multiplication.

Once we have these compressed representations `g1` and `g2` of `mat1` and `mat2`, respectively, we create an answer matrix `ans` initialized with zeros. For each row of `mat1`, we look at the non-zero values and their column indexes. For each of these non-zero values, we find the respective row (with matching column index) in `g2` and multiply the corresponding elements. The product is then added to the correct position in the answer matrix.

This approach significantly reduces the number of multiplication operations, particularly when the matrices have a lot of zeros, which is very common in applications that deal with sparse data.

Solution Approach

The solution follows a two-step approach: preprocessing the input matrices to extract the non-zero values with their positions, and then performing a modified multiplication that only operates with these non-zero values.

Preprocessing Step

The preprocessing function `find()` is a decisive optimization in the algorithm. It takes advantage of the matrix's sparsity to reduce the computational complexity. The function goes through each element of the given matrix `mat` and records the column index and the value of all non-zero elements in a new data structure which we refer to as `g`.

For every row `i` in `mat`, a list `g[i]` is created. For every non-zero element `x` in `row`, a tuple `(j, x)` is appended to `g[i]`, where `j` is the column index of `x`. This results in a list of lists where each sublist represents a row and contains only the relevant data for multiplication - that is, the column position and value of non-zero elements.

Matrix Multiplication Step

Once we have applied `f` to both `mat1` and `mat2`, obtaining `g1` and `g2`, we proceed to the actual multiplication.

1. We initialize the answer matrix `ans` with the correct dimensions $a \times c$ and fill it with zeros. To do this, we build a list of lists with `a` sublists each containing `c` zeros. `[[0] * c for _ in range(a)]` creates a list with `c` elements, each of which is a list of `z` zeros.
2. We then iterate through each row `i` of `mat1`. For each non-zero element `x` in this row (represented as a tuple `(k, x)` in `g1[i]`), we perform the next steps:
   - For every tuple `(j, y)` in `g2[k]`, which represents the non-zero elements of the `k`-th row in `mat2`, we multiply the value `x` from `mat1` with the value `y` from `mat2`, and accumulate the product into the appropriate cell in the answer matrix `ans[i][j]`.
3. This accumulator step, `ans[i][j] += x * y`, is the core of matrix multiplication. It adds up the product of corresponding elements from pairwise aligned rows of `mat1` and columns of `mat2`.
4. The nested loops over `g1[i]` and `g1[k]` ensure that we only consider terms that contribute to the final answer, avoiding the unnecessary operations by zero which is the central inefficiency in dense matrix multiplication.

This algorithm maintains an efficient computation by only considering the relevant (non-zero) elements in the multiplication, significantly speeding it up for sparse matrices. The use of list comprehensions and tuple unpacking in Python contributes to the readability and conciseness of the code, while the use of a list of lists as a data structure allows for fast access and update of individual elements.

Example Walkthrough

Let's walk through a simple example to illustrate the solution approach. Suppose we have the following two sparse matrices `mat1` and `mat2`.

`mat1` (2×3 matrix):

```
1  [1, 0, 0]
2  [0, 0, 1]
```

`mat2` (3×2 matrix):

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

The expected result of the multiplication will be a 2×2 matrix. Now, let's use the solution approach to calculate this.

Preprocessing Step

First, we preprocess `mat1` and `mat2` into their sparse representations `g1` and `g2` using function `f(mat)`. Non-zero values and their column indices are kept.

Result of `f(mat1)` is `g1` (list of lists of tuples):

```
1  [
2    [0, 1],   // Elements are `1` at column `0` and `2` at column `1`
3    [2]       // Second row's 0-th index, non-zero element's column index
4  ]           // Only element in third row is `1` at column `2`
5  ]
```

Result of `f(mat2)` is `g2` (list of lists of tuples):

```
1  [
2    [0, 1], [1, 2],   // Elements are `1` at column `0` and `2` at column `1`
3    [1]               // Second row's 0-th index, non-zero element's column index
4    [1, 4]            // Only element in third row is `1` at column `1`
5  ]
```

Matrix Multiplication Step

Next, we perform the matrix multiplication:

1. Initialize the answer matrix `ans` filled with zeroes. In this case, an `2x2` matrix would look like:

```
1  ans = [
2    [0, 0],
3    [0, 0]
4  ]
```

2. For each row `i` in `g1`, and for each tuple `(k, x)` in `g1[i]`, we do the following:
   - If `i` is 0 (first row in `mat1`), we process `g1[0]`, which is `[0, 1]`. There's only one non-zero value `1` at column `0`. We find all non-zero elements in row `0` of `g2` and multiply:
     - We multiply `1` (from `g1[0][0]`) * `1` (from `g2[0][0]`) and add it to `ans[0][0]`.
     - We multiply `1` (from `g1[0][0]`) * `2` (from `g2[0][1]`) and add it to `ans[0][1]`.
   - If `i` is 1 (second row in `mat1`), we process `g1[1]`, which is `[2, 3]`. There's only one non-zero value `3` at column `2`. We find all non-zero elements in row `2` of `g2` and multiply:
     - We multiply `3` (from `g1[1][0]`) * `4` (from `g2[2][1]`) and add it to `ans[1][1]`.

The `ans` matrix now contains the result of the multiplication:

```
1  ans = [
2    [1, 2],
3    [0, 12]
4  ]
```

This walk-through illustrates how the algorithm efficiently performs multiplication by directly accessing the non-zero elements and their positions, avoiding unnecessary multiplications with zero, which would not contribute to the result.

Python Solution

```python
# Import type hints for better code readability
from typing import List

class Solution:
    def multiply(self, mat1: List[List[int]], mat2: List[List[int]]) -> List[List[int]]:
        # Helper function to create a sparse matrix representation
        def create_sparse_matrix(matrix: List[List[int]]) -> List[List[Tuple]]:
            # Initialize a list to store the sparse representation
            sparse_matrix = [[] for _ in range(len(matrix))]
            # Iterate through the matrix to populate non-zero values along with their column indexes
            for row_index, row in enumerate(matrix):
                for col_index, value in enumerate(row):
                    if value != 0:
                        # Append a tuple containing the column index and the value if non-zero
                        sparse_matrix[row_index].append((col_index, value))
            return sparse_matrix

        # Create the sparse matrix representations for both input matrices
        sparse_mat1 = create_sparse_matrix(mat1)
        sparse_mat2 = create_sparse_matrix(mat2)

        # Get the dimensions for the resulting matrix
        m, n = len(mat1), len(mat2[0])
        # Initialize the resulting matrix with zeros
        result_matrix = [[0] * n for _ in range(m)]

        # Iterate through each row of mat1
        for i in range(m):
            # Iterate through the sparse representation of the row from mat1
            for col_index_mat1, value_mat1 in sparse_mat1[i]:
                # Multiply with corresponding elements from mat2 through the corresponding row in mat2
                for col_index_mat2, value_mat2 in sparse_mat2[col_index_mat1]:
                    # Multiply and add to the resulting matrix
                    result_matrix[i][col_index_mat2] += value_mat1 * value_mat2

        # Return the resulting matrix after multiplication
        return result_matrix
```

Java Solution

```java
class Solution {
    // The main method to perform matrix multiplication
    public int[][] multiply(int[][] mat1, int[][] mat2) {
        int m = mat1.length, n = mat2[0].length; // Determine the resulting matrix dimensions
        int[][] result = new int[m][n]; // Initialize the resulting matrix

        // Convert the matrices to lists containing only non-zero elements
        List<int[]>[] nonZeroValuesMat1 = convertToNonZeros(mat1);
        List<int[]>[] nonZeroValuesMat2 = convertToNonZeros(mat2);

        // Iterate through each row of the first matrix
        for (int i = 0; i < m; i++) {
            // For each non-zero element in the current row
            for (int[] pair1 : nonZeroValuesMat1[i]) {
                int columnIndexMat1 = pair1[0], valueMat1 = pair1[1];
                // Multiply with corresponding non-zero elements in second matrix
                for (int[] pair2 : nonZeroValuesMat2[columnIndexMat1]) {
                    int columnIndexMat2 = pair2[0], valueMat2 = pair2[1];
                    result[i][columnIndexMat2] += valueMat1 * valueMat2; // Update the result matrix
                }
            }
        }
        return result;
    }

    // Convert a matrix to list of non-zero elements
    private List<int[]>[] convertToNonZeros(int[][] matrix) {
        List<int[]>[] nonZeroValues = new List[matrix.length];
        for (int i = 0; i < matrix.length; i++) {
            nonZeroValues[i] = new ArrayList<>();
        }

        // Initialize the list of arrays for each row
        Arrays.setAll(nonZeroValues, i -> new ArrayList<>());

        // Collect non-zero elements in the matrix
        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[0].length; ++j) {
                if (matrix[i][j] != 0) {
                    // For each non-zero element, add an array containing the column index and the value
                    nonZeroValues[i].add(new int[] {j, matrix[i][j]});
                }
            }
        }
        return nonZeroValues;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <utility> // Include for std::pair

class Solution {
public:
    // Multiplies two matrices represented as 2D vectors.
    std::vector<std::vector<int>> multiply(std::vector<std::vector<int>>& mat1, std::vector<std::vector<int>>& mat2) {
        int rows = mat1.size(), cols = mat2[0].size(); // Number of rows in the result
        int commonDim = mat1[0].size(); // Number of columns in mat1/rows in mat2
        std::vector<std::vector<int>> result(rows, std::vector<int>(cols));

        // Convert the matrices to a sparse representation for faster computation
        auto sparseMat1 = convertToSparse(mat1);
        auto sparseMat2 = convertToSparse(mat2);

        // Perform multiplication using the sparse representation
        for (int i = 0; i < rows; ++i) {
            for (auto& [k, value1] : sparseMat1[i]) {
                for (auto& [j, value2] : sparseMat2[k]) {
                    result[i][j] += value1 * value2;
                }
            }
        }

        return result;
    }

    // Converts a matrix to a sparse representation to optimize multiplication of sparse matrices.
    std::vector<std::vector<std::pair<int, int>>> convertToSparse(std::vector<std::vector<int>>& matrix) {
        int rows = matrix.size(); // Number of rows in the matrix
        int cols = matrix[0].size(); // Number of columns in the matrix
        std::vector<std::vector<std::pair<int, int>>> sparseRepresentation(rows);

        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                if (matrix[i][j] != 0) { // Filter out zero values for sparse representation
                    sparseRepresentation[i].emplace_back(j, matrix[i][j]);
                }
            }
        }

        return sparseRepresentation;
    }
};
```

Typescript Solution

```typescript
function multiply(mat1: number[][], mat2: number[][]): number[][] {
    // Get the dimensions required for the resulting matrix
    const numRowsMat1 = mat1.length; // Number of rows in mat1
    const numColsMat1 = mat1[0].length; // Number of columns in mat1

    // Initialize the resulting matrix with zeroes
    const resultMatrix: number[][] = Array.from(
        { length: numRowsMat1 },
        () => Array.from({ length: numColsMat2 }, () => 0)
    );

    // Function to filter out all zero values and keep only non-zero values with their column index
    const filterZeroes = (matrix: number[][]): number[][][] => {
        // Map each row to a list of [column index, value] tuples for the given matrix
        return matrix.map(row => row.reduce((nonZeroes, value, colIndex) => {
            if (value !== 0) nonZeroes.push([colIndex, value]);
            return nonZeroes;
        }, []));
    };

    // Get non-zero values for both matrices
    const filteredMat1 = filterZeroes(mat1);
    const filteredMat2 = filterZeroes(mat2);

    // Perform matrix multiplication using the sparse representations
    for (let i = 0; i < numRowsMat1; ++i) {
        for (const [k, valueMat1] of filteredMat1[i]) {
            for (const [j, valueMat2] of filteredMat2[k]) {
                resultMatrix[i][j] += valueMat1 * valueMat2;
            }
        }
    }

    // Return the resulting matrix after multiplication
    return resultMatrix;
}
```

Time and Space Complexity

Time Complexity

To analyze time complexity, let's examine the code step by step.

1. **Function `f(mat)` Sparse Matrix Representation** - This function converts a regular matrix to a sparse representation by recording non-zero elements' coordinates and their values. If `mat` has dimensions $r \times c$ with `t` non-zero elements, then this function would take $O(r * c)$ time, assuming the worst case where every element needs to be checked.
2. **Conversion to Sparse Representation** - $g1 = f(mat1)$ is called for the first matrix having dimensions $m \times k$ (assuming n rows and k columns), and $g2 = f(mat2)$ for the second matrix having dimensions $k \times n$. The total time for these calls is $O(m * k) + O(k * n) = O(m * k + k * n)$.
3. **Matrix Multiplication** - The nested loops for matrix multiplication iterate over m rows of `mat1`, k sparse columns of `mat1`, and n sparse columns of `mat2`. The number of iterations would depend on the number of non-zero elements in the sparse representations of `mat1` and `mat2`. If `nz1` and `nz2` are the number of non-zero entries in `mat1` and `mat2` respectively, the time complexity for the multiplication part is $O(nz1 * nz * O(1))$ because for every non-zero element in `g1[i]`, we need to iterate over potentially up to n elements in `g2[k]`. However, in a truly sparse scenario, not every element in `g1[i]` correlates to a non-zero product, so this is an upper bound.

The overall time complexity would be $O(m * k + k * n + nz1 * n * nz2)$ which is typically represented as $O(m * k + k * n)$ if we are looking at worst-case scenario considering dense matrices, where `nz1` approaches $m * k$ and `nz2` approaches $k * n$.

Space Complexity

For space complexity, we have:

1. **Sparse Representation** - Storing the sparse representation of both matrices, which might take up to $O(m * k + k * n)$ space in the worst case (each original matrix is dense), but in practice would be $O(nz1 + nz2)$ for sparse matrices with `nz1` and `nz2` non-zero entries respectively.
2. **Output Matrix** `ans` - Initializing a result matrix of dimensions $m \times n$, thus requiring space complexity of $O(m * n)$.

The space complexity is dominated by the larger of the two factors: output matrix and sparse representations. Therefore, the space complexity is $O(m * n + nz1 + nz2)$. In worst-case scenario with dense matrices, this would simplify to $O(m * n)$.