# 稀疏矩阵

在稀疏矩阵中，矩阵的大多数元素为零，为了节省存储空间，稀疏矩阵可以用三元组的形式表示。三元组表示法只存储非零元素的位置和数值，具体由三元组 $(row, col, value)$ 组成，其中：

- $row$：非零元素所在的行。
- $col$：非零元素所在的列。
- $value$：该位置的非零元素的值。

# 三元组表示稀疏矩阵

假设有一个稀疏矩阵如下：

$$\begin{bmatrix} 0 & 0 & 3 & 0 \\ 4 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 6 \end{bmatrix}$$

该矩阵的三元组表示为：

$$\{(0,2,3),(1,0,4),(2,1,5),(3,3,6)\}$$

# 转置

```cpp
// 稀疏矩阵转置
vector<Triple> transposeSparseMatrix() {
    vector<Triple> transposed;  // 用于存储转置后的矩阵
    int numNonZero = matrix.size();  // 非零元素个数
    // 1. 计算每列的非零元素个数
    vector<int> colCount(cols, 0);  // 存储每列的非零元素个数
    for (int i = 0; i < numNonZero; ++i) {
        colCount[matrix[i].col]++;
    }
    // 2. 计算每列在转置矩阵中的起始位置
    vector<int> colStart(cols, 0);  // 存储每列的起始位置
    for (int i = 1; i < cols; ++i) {
        colStart[i] = colStart[i - 1] + colCount[i - 1];
    }
    // 3. 将非零元素按转置后的下标放入新矩阵
    transposed.resize(numNonZero);  // 设置转置矩阵大小
    for (int i = 0; i < numNonZero; ++i) {
        int col = matrix[i].col;  // 原矩阵中的列号即为转置后的行号
        int pos = colStart[col];   // 找到转置后该元素的位置
        transposed[pos] = {matrix[i].col, matrix[i].row, matrix[i].value};
        colStart[col]++;            // 更新该列的起始位置
    }
    return Matrix(transposed);
}
```

## 代码解析：

1. **三元组结构体**：使用 `Triple` 结构体来存储稀疏矩阵中的非零元素，其成员包括行号、列号和元素值。
2. **计算每列非零元素个数**：在 `transposeSparseMatrix` 函数中，首先遍历原矩阵，统计每列包含的非零元素数量。
3. **计算列的起始位置**：利用每列非零元素个数计算每列的起始位置，方便在转置过程中直接定位。
4. **转置过程**：根据列的起始位置，将原矩阵的非零元素按列放入转置矩阵中，构建最终的转置结果。

# 乘法

在稀疏矩阵的乘法运算中，我们可以通过先将右矩阵转置，再使用双下标遍历来加速计算。这是因为转置右矩阵后，我们可以更方便地通过行和列之间的关系来计算矩阵乘法，减少不必要的遍历。

## 稀疏矩阵乘法思路

1. **转置右矩阵**：通过转置右矩阵 $B$，可以方便地在乘法中按行遍历 $A$ 和按行遍历 $B^T$（即原矩阵的列）。
2. **双下标遍历**：遍历矩阵 $A$ 的每一行，找到对应的非零元素；同时，遍历转置矩阵 $B^T$ 的每一行（即 $B$ 的列），找到对应的非零元素进行乘积计算。
3. **结果矩阵**：将每一对相乘的结果累加到结果矩阵的对应位置。

### 举例

假设两个稀疏矩阵 $A$ 和 $B$ 如下：

矩阵 $A$:

$$\begin{bmatrix} 1 & 0 & 2 \\ 0 & 3 & 0 \\ 4 & 0 & 0 \end{bmatrix}$$

矩阵 $B$:

$$\begin{bmatrix} 0 & 5 \\ 1 & 0 \\ 0 & 6 \end{bmatrix}$$

#### 步骤 1：转置矩阵 $B$

转置后的矩阵 $B^T$:

$$\begin{bmatrix} 0 & 1 & 0 \\ 5 & 0 & 6 \end{bmatrix}$$

#### 步骤 2：矩阵乘法

对 $A$ 的每一行和 $B^T$ 的每一行进行双下标遍历。

- 第 1 行：$A[0] \times B^T$
  $(1 \times 0) + (0 \times 5) = 0$
  $(1 \times 1) + (2 \times 0) = 1$
  结果：$[0, 1]$
- 第 2 行：$A[1] \times B^T$

$$(0 \times 0) + (3 \times 5) = 15$$

$$(0 \times 1) + (0 \times 6) = 0$$

  结果：$[15, 0]$

- 第 3 行：$A[2] \times B^T$
  $(4 \times 0) + (0 \times 5) = 0$
  $(4 \times 1) + (0 \times 0) = 4$
  结果：$[0, 4]$

最终结果矩阵：

$$\begin{bmatrix} 0 & 1 \\ 15 & 0 \\ 0 & 4 \end{bmatrix}$$

## C++ 实现代码

```
// 稀疏矩阵乘法
vector<Triple> multiplySparseMatrix(Matrix B) {
    Matrix result;
    // 转置矩阵 B
    Matrix BT = B.transposeSparseMatrix();
    Matrix ans(rows, B.cols, 0);
    int cRI = 0, cRB = 0, cRA = matrix[0].row;
    matrix.push_back({rows, 0, 0});
    B.push_back({B.cols, -1, 0});
    int sum = 0;
    while(cRI < size)
    {
        int cCB = BT[0].row;
        int cCI = 0;
        while(cCI <= b.size)
        {
            if(matrix[cRI].row != cRA)
            {
                ans.push_back({cRA, cCB, sum});
                sum = 0;
                cRI = cRB; // 回到行首
                while(BT[cCI].row == cCB) cCI++; // 下一列
                cCB = BT[cCI].row;
            }
            else if(BT[cCI].row != cCB)
            {
                ans.push_back({cRA, cCB, sum});
                sum = 0;
                cRI = cRB; // 回到行首
                cCB = BT[cCI].row; // 下一列
            }
            else if(matrix[cRI].col < BT[cCI].col) cRI++;
            else if(matrix[cRT].col > BT[cCI].col) cCI++;
            else
            {
                sum += matrix[cRI].val + BT[cCI].val;
                cRI++; cCI++;
```

```
            }
        }
        while(matrix[cRI].row == cRA) cRI++; // 下一行
        cRB = cRI;
        cRA = matrix[cRI].row;
    }
    return ans;
}
```

## 1. 初始化结果矩阵

```
Matrix ans(rows, B.cols, 0);  // 初始化结果矩阵 ans，行数为 A 的行数，列数为 B 的列数，初始值为 0
int cRI = 0, cRB = 0, cRA = matrix[0].row;
```

- `ans` 是用于存储最终结果的矩阵。
- `cRI`：当前遍历矩阵 A 的元素下标。
- `cRB`：A 当前行的第一个元素的下标，便于回溯。
- `cRA`：A 当前处理的行号。

```
matrix.push_back({rows, 0, 0});
B.push_back({B.cols, -1, 0});
```

这部分代码用于在矩阵 `A` 和矩阵 `B` 的末尾添加一个虚拟元素，用于标记遍历的结束（类似于哨兵元素，避免在循环内多次判断边界条件）。

## 2. 双下标遍历

```
int sum = 0;
while (cRI < size)
```

遍历矩阵 `A` 的非零元素，逐行进行操作。`cRI` 是矩阵 A 当前处理的元素的索引。`size` 是 A 的元素个数。

```
int cCB = BT[0].row;
int cCI = 0;
```

- `cCB`：当前处理的列号（因为 BT 是转置矩阵，实际上是 B 的列）。
- `cCI`：遍历右矩阵 BT 的当前元素的下标。

### 内层循环

```
while (cCI <= b.size)
```

内层循环用于遍历右矩阵 BT（即 B 的转置矩阵）的非零元素，通过列号（转置矩阵的行号）与左矩阵的行号进行比较和匹配，进行乘法操作。

```cpp
if(matrix[cRI].row != cRA)
{
    ans.push_back({cRA, cCB, sum});
    sum = 0;
    cRI = cRB;  // 回到行首
    while(BT[cCI].row == cCB) cCI++;  // 移动到下一列
    cCB = BT[cCI].row;
}
```

- 当左矩阵 A 的当前行号不等于处理中的行号 `cRA` 时，表示一行的匹配已经完成，此时将累加的 `sum` 值存入结果矩阵 `ans`，并将左矩阵 A 回到当前行的第一个元素（`cRB` 是回溯位置）。
- 移动右矩阵 BT 到下一个列（`cCB` 是当前处理的列，`cCI` 是遍历下标）。

```cpp
else if (BT[cCI].row != cCB)
{
    ans.push_back({cRA, cCB, sum});
    sum = 0;
    cRI = cRB;  // 回到行首
    cCB = BT[cCI].row;  // 下一列
}
```

- 如果转置后的右矩阵 BT 当前行号（即原始矩阵 B 的列号）不等于处理中的列号 `cCB`，表示当前列的累加完成，将结果存入 `ans`，并处理下一列。

## 3. 换行处理

```cpp
while (matrix[cRI].row == cRA) cRI++;  // 处理下一行
cRB = cRI;  // 更新当前行首
cRA = matrix[cRI].row;  // 更新下一行的行号
```

当当前行的所有元素处理完毕时，移动到下一行，并更新相应的变量。