北京郵電大學

数据结构实验报告

题目： 稀疏矩阵的加法和乘法

姓　　名：魏生辉

学　　院：计算机学院（国家示范性软件学院）

专　　业：计算机类

班　　级：2023211307

学　　号：2023211075

指导教师：杨震

2024年 10月

目录

1 需求分析

1.1 题目描述

输入两个稀疏矩阵 A 和 B，判断它们能否相加/相乘。
如果能，做矩阵相加/相乘运算，输出运算结果；如果不能，输出原因

1.2 输入描述

操作选择：

用户通过命令行输入操作选项：1：选择矩阵相加操作。
2：选择矩阵相乘操作。
0：退出程序。

矩阵输入（有效输入范围：1 <= 行数，列数 <= 100。）：

当用户选择矩阵相加或相乘时，需要输入两个矩阵的行数和列数，以及具体的矩阵数据：
矩阵A的行数与列数（格式：m n，其中m表示行数，n表示列数）。
矩阵A的每个元素（按照行顺序输入，包含零元素）。
矩阵B的行数与列数（格式：p q，其中p表示行数，q表示列数）。
矩阵B的每个元素（按照行顺序输入，包含零元素）。
稀疏矩阵的数据以二维矩阵的形式输入，每行数据之间以空格分隔。

范围说明

（矩阵有效输入范围：1 <= 行数，列数 <= 100。）
（元素有效输入范围：每个矩阵元素应为 -1000 <= 元素值 <= 1000）

1.3 输出描述

无效操作提示：

若用户输入的操作选项不在 0 到 2 范围内，程序会提示“无效的选项”。

矩阵相加：

如果矩阵A和矩阵B行数、列数相同，程序输出相加后的矩阵，按二维矩阵的形式打印每行元素，以空格分隔。

若行数或列数不同，程序输出“不可相加：行列数不相等”。

矩阵相乘：

如果矩阵A的列数等于矩阵B的行数，程序输出相乘后的矩阵，按二维矩阵的形式打印每行元素，以空格分隔。

若矩阵A的列数不等于矩阵B的行数，程序输出“不可相乘：A列数不等于B行数”。

程序退出：

当用户选择退出选项（0），程序输出“感谢您的使用”，并结束运行。

1.4 样例输入输出

1.4.1 样例输入输出 1

【输入】

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

【输出】

```
1 1
1 1
```

1.4.2 样例输入输出 2

【输入】

```
2
2 2
3 0
0 7
2 2
4 6
7 7
```

【输出】

```
12 18
49 49
```

1.4.3 样例输入输出 3

【输入】

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

【输出】

```
9 2
4 3
```

1.5 程序功能

该程序实现了对稀疏矩阵的基本运算功能，包括矩阵相加、相乘。具体功能如下：

矩阵相加：

程序实现了两个稀疏矩阵的相加运算。只有当两个矩阵的行数和列数相同时，程序才进行相加操作。相加过程中逐个比较两个矩阵中的非零元素，若行列位置相同则将对应元素相加，否则将元素直接添加到结果矩阵中。

矩阵相乘：

程序实现了两个稀疏矩阵的相乘运算。只有当矩阵A的列数与矩阵B的行数相等时，才允许进行矩阵乘法。矩阵相乘的结果为一个新矩阵，程序通过获取矩阵A的行和矩阵B的转置后的列，逐个计算每个位置的乘积和，并将非零结果存入结果矩阵。

2 概要设计

2.1 问题解决的思路

本程序的主要目的是对稀疏矩阵进行高效存储和运算。稀疏矩阵的大部分元素为零，因此直接存储所有元素会浪费空间。通过将非零元素单独存储并使用特定的算法进行矩阵运算，程序实现了空间和时间的优化。下面是解决问题的主要思路：

矩阵初始化：

当用户输入矩阵数据时，程序只将非零元素存储到稀疏矩阵结构中。在此过程中，通过遍历输入的二维矩阵，将每个非零元素的位置（行号和列号）及其值记录在 MatrixElement 数组中。同时统计非零元素的数量，以便后续操作。

矩阵相加：

稀疏矩阵的相加需要判断两个矩阵是否具有相同的维度（行数和列数相等）。若满足相加条件，程序逐个比较两个矩阵的非零元素，按照行和列的位置顺序进行合并：如果两个矩阵在同一个位置都有非零元素，则相加其值。

如果某个位置只有一个矩阵有非零元素，则直接将该元素添加到结果矩阵中。

结果矩阵最终只包含相加后的非零元素。

矩阵相乘：

矩阵相乘的条件是矩阵A的列数必须等于矩阵B的行数。为提高乘法运算效率，程序先对矩阵B进行转置操作，以便通过获取矩阵A的行和矩阵B（转置后）的列进行逐元素相乘。具体步骤为：程序遍历矩阵A的每一行和矩阵B（转置后）的每一列，计算它们对应位置的元素乘积并求和。

只在求和结果非零的情况下，将结果存入结果矩阵，避免存储不必要的零元素。

2.1 数据结构的定义

①***MatrixElement*** 结构体

MatrixElement 结构体用于表示矩阵中的一个非零元素，包括该元素的值及其在矩阵中的位置（行号和列号）。

```
typedef struct MatrixElement {
  int value; // 非零元素的值
  int row;   // 非零元素所在的行号
  int col;   // 非零元素所在的列号
} MatrixElement;
```

value: 存储矩阵中该位置的非零元素的值。

row: 表示该非零元素在矩阵中的行号。

col: 表示该非零元素在矩阵中的列号。

通过 MatrixElement 结构体，程序只存储矩阵中的非零元素，并通过行号和列号来标识该元素的位置。

② ***SparseMatrix*** 结构体

SparseMatrix 结构体用于表示整个稀疏矩阵，包含矩阵的行数、列数、非零元素的数量以及一个用于存储所有非零元素的数组。

```
typedef struct SparseMatrix {
   int rows;             // 矩阵的行数
   int cols;             // 矩阵的列数
   int nonZeroCount;     // 矩阵中非零元素的数量
   MatrixElement* elements; // 存储非零元素的数组
} SparseMatrix;
```

rows: 矩阵的行数，表示矩阵有多少行。

cols: 矩阵的列数，表示矩阵有多少列。

nonZeroCount: 矩阵中非零元素的数量，即 elements 数组的有效元素个数。

elements: 这是一个 MatrixElement 数组，用于存储矩阵中的所有非零元素。每个元素都包含其值及所在位置的行号和列号。

2.3 主程序的流程

启动程序：

用户选择操作（矩阵相加、相乘或退出程序）。

矩阵输入：

用户输入矩阵A和矩阵B的行数、列数及其元素。

矩阵运算：

根据用户选择，程序调用相应的函数执行矩阵相加或矩阵相乘操作。

结果输出：

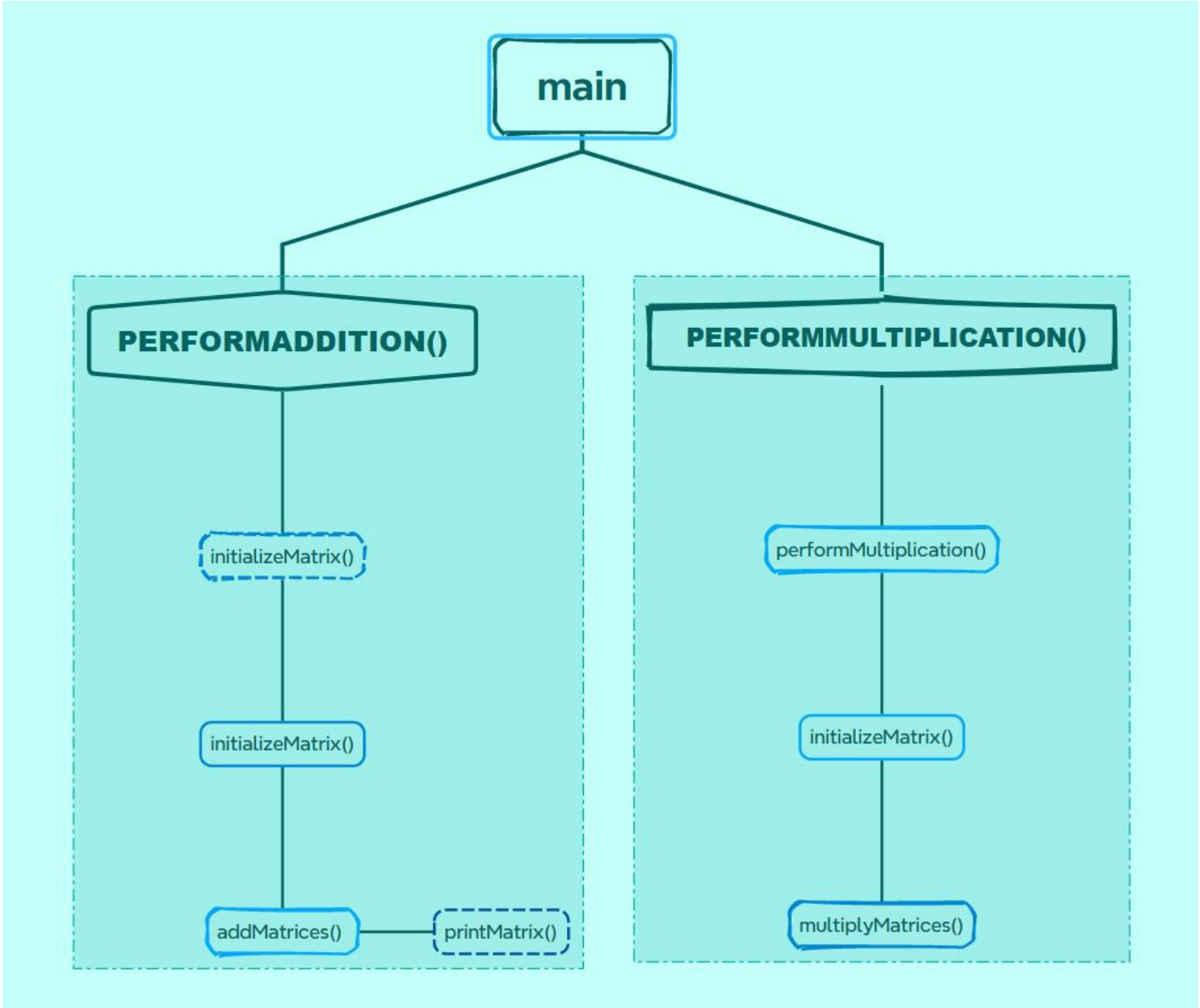
程序输出矩阵运算结果（加法或乘法），或提示不能进行运算（例如行列不匹配）。

循环操作：

用户可以继续选择操作，直到输入 0 退出程序。

2.4 各程序模块之间的层次关系

函数调用关系图下图 ：



3 详细设计

3.1 伪代码的设计

伪代码设计如下：

```
主程序 main()
        初始化 选项
        循环
             输出 "请选择功能"
             输出 "1. 矩阵相加"
             输出 "2. 矩阵相乘"
            输出 "0. 结束程序"
       输入 选项
     如果 选项 == 1
         执行 performAddition()
     否则如果 选项 == 2
         执行 performMultiplication()
     否则如果 选项 == 0
         输出 "感谢您的使用"
        退出程序
     否则
        输出 "无效的选项"
    结束条件
 结束循环
结束主程序
```

```
函数 performAddition()
    输入 行数A, 列数A
初始化 dataA 数组
    输入 矩阵A元素到 dataA
执行 initializeMatrix(行数A, 列数A, dataA) 得到 matrixA
    输入 行数B, 列数B
    初始化 dataB 数组
 输入 矩阵B元素到 dataB
 执行 initializeMatrix(行数B, 列数B, dataB) 得到 matrixB
 resultMatrix = addMatrices(matrixA, matrixB)
 如果 resultMatrix == NULL
     输出 "不可相加: 行列数不相等"
 否则
     输出 "和矩阵为:"
    执行 printMatrix(resultMatrix)
    释放 resultMatrix 的内存
 结束条件
  释放 matrixA 和 matrixB 的内存
结束函数
```

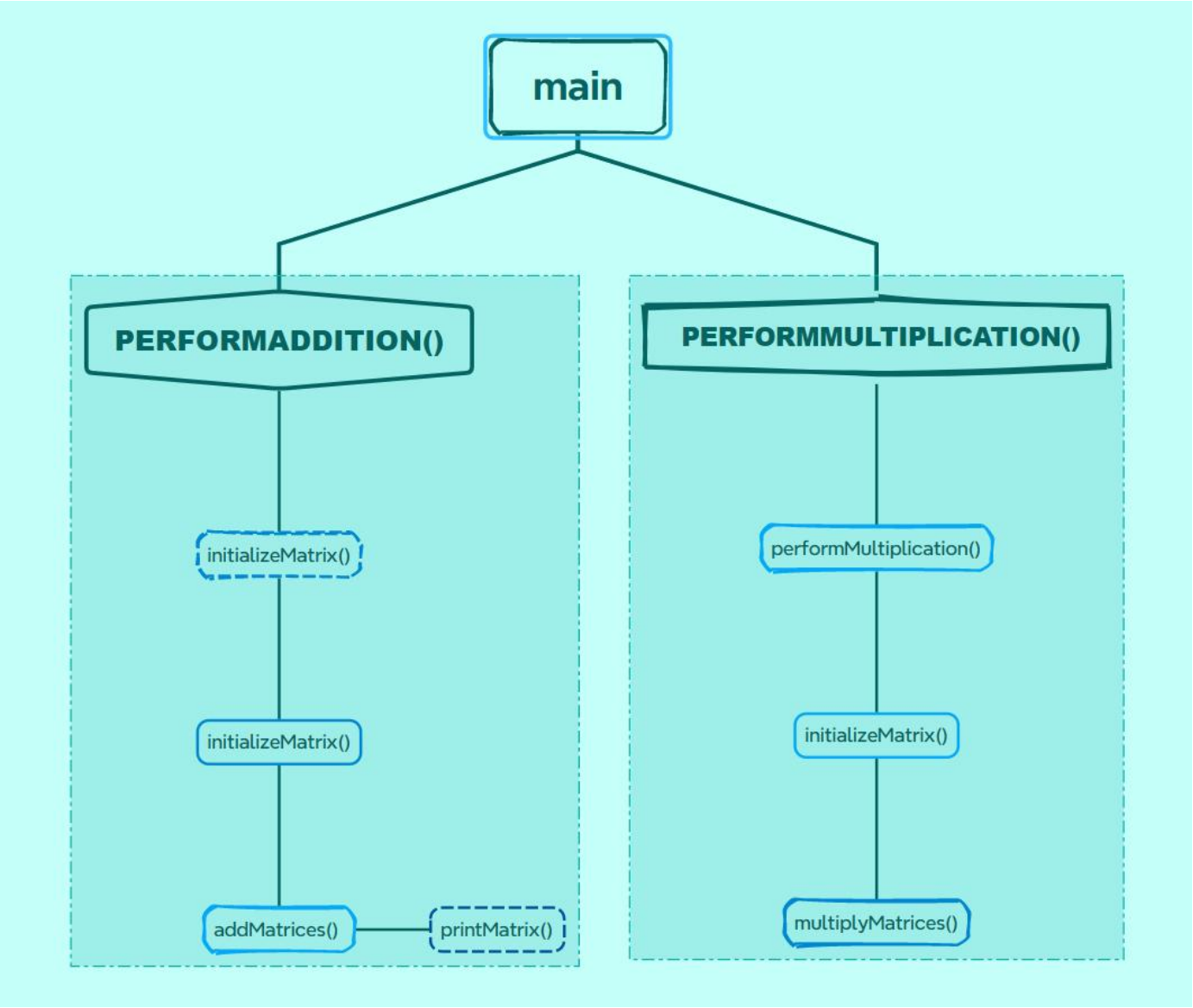
```
函数 performMultiplication()
输入 行数A, 列数A
初始化 dataA 数组
输入 矩阵A元素到 dataA
执行 initializeMatrix(行数A, 列数A, dataA) 得到 matrixA

输入 行数B, 列数B
初始化 dataB 数组
输入 矩阵B元素到 dataB
执行 initializeMatrix(行数B, 列数B, dataB) 得到 matrixB
   resultMatrix = multiplyMatrices(matrixA, matrixB)
如果 resultMatrix == NULL
     输出 "不可相乘: A列数不等于B行数"
否则
    输出 "乘积矩阵为:"
    执行 printMatrix(resultMatrix)
    释放 resultMatrix 的内存
结束条件
   释放 matrixA 和 matrixB 的内存
结束函数
```

```
函数 initializeMatrix(行数, 列数, data[])
创建 SparseMatrix 结构体
初始化 rows, cols, nonZeroCount
遍历 data 数组
    如果 data[i] != 0
        创建 MatrixElement 元素
        记录元素的值、行号、列号
        将元素存入 elements 数组
        增加 nonZeroCount
    结束条件
返回 SparseMatrix
结束函数
函数 addMatrices(matrixA, matrixB)
如果 matrixA 和 matrixB 的维度不匹配
    返回 NULL
创建 resultMatrix 结构体
初始化指针 ptrA, ptrB, ptrResult
遍历 matrixA 和 matrixB 的非零元素
    根据行列号相加或选择非零元素
更新 resultMatrix 的非零元素数量
返回 resultMatrix
结束函数
```

```
  函数 multiplyMatrices(matrixA, matrixB)
 如果 matrixA 的列数 != matrixB 的行数
    返回 NULL
 创建 resultMatrix 结构体
执行 matrixB 的转置操作
 遍历 matrixA 和转置后的 matrixB 计算乘积
 返回 resultMatrix
  结束函数
```

```
函数 printMatrix(matrix)
 遍历每一行和每一列
    根据非零元素的位置打印元素或打印 0
  结束函数
```

3.2 函数的调用关系图



4 调试分析报告

4.1 调试过程中遇到的问题和思考

稀疏矩阵存储设计的挑战：

问题：稀疏矩阵中的大多数元素为零，存储所有元素不仅浪费空间，还会导致在矩阵运算时大量不必要的零元素参与计算，降低效率。

思考：为了提高存储和计算效率，我决定只存储非零元素，并记录这些元素的具体行列位置。这样可以减少对空间的浪费，同时在进行矩阵运算时仅处理有效数据，避免零元素的冗余操作。

稀疏矩阵转置问题：

问题：在矩阵乘法中，矩阵B的列需要与矩阵A的行相匹配。因此在计算时需要对矩阵B进行转置操作，使其列与矩阵A的行对齐。

思考：为了高效处理矩阵B的转置，我设计了一个矩阵转置函数，在该函数中通过将行列位置互换的方式重新组织矩阵B的非零元素，并维护它们在转置后的位置顺序。

4.2 设计实现的回顾讨论

矩阵相加时行列数匹配的实现：

回顾：在执行矩阵相加前增加了一个检查，确保两个矩阵的行列数一致。这不仅避免了无效计算，还提升了程序的鲁棒性。当用户输入的矩阵不符合要求时，程序会给予友好的提示信息。

讨论：在调试过程中，这一设计使得程序能够更好地处理用户输入错误，并且避免潜在的运算错误或程序崩溃。这也为后续扩展更多矩阵操作奠定了基础。

矩阵相乘时行列数检查与操作的实现：

回顾：为了保证矩阵乘法的有效性，我在矩阵相乘之前检查矩阵A的列数是否等于矩阵B的行数。如果不满足，程序会输出错误提示。这一实现确保了矩阵乘法的正确性。

讨论：该检查机制的引入，增强了程序的健壮性，防止了由于不匹配的矩阵输入而导致的计算错误。同时，我在调试时发现，列数和行数的严格匹配是确保矩阵乘法逻辑正确运行的关键。

4.3 算法复杂度分析

4.3.1时间复杂度

初始化矩阵：

复杂度：O(m * n)

说明：遍历所有矩阵元素，找到非零元素。m 是矩阵的行数，n 是列数。

矩阵相加：

复杂度：O(kA + kB)

说明：遍历两个矩阵的非零元素进行相加，kA 和 kB 分别是矩阵A和B的非零元素个数。

矩阵相乘：

复杂度：O(kA * kB)

说明：对矩阵A的非零元素和矩阵B转置后的非零元素进行乘法操作。kA 和 kB 是两个矩阵的非零元素个数。

矩阵转置：

复杂度：O(k)

说明：遍历矩阵所有非零元素，交换行和列的位置，k 是非零元素个数。

4.3.2空间复杂度

初始化矩阵：

复杂度：O(k)

说明：只存储非零元素，k 是非零元素的个数。

矩阵相加：

复杂度：O(kA + kB)

说明：结果矩阵的非零元素个数最多为 kA + kB。

矩阵相乘：

复杂度：O(kC)

说明：乘法结果矩阵的非零元素个数为 kC。

矩阵转置：

复杂度：O(k)

说明：转置矩阵所需的空间与原矩阵的非零元素个数相同。

4.4 改进设想的经验和体会

4.4.1 改进 1-使用哈希表优化时间

在原有的矩阵乘法中，每一行的元素与另一矩阵的每一列进行逐一相乘。当矩阵稀疏时，很多乘法操作会涉及零元素，导致不必要的计算。

为了优化矩阵乘法，我们可以通过使用哈希表来存储矩阵的非零元素，以便快速查找并避免无效计算。哈希表的键可以是行号或列号，值则是该行或列的非零元素。这样在进行矩阵乘法时，我们只需要计算有意义的非零元素的乘积。

改动代码如下：

```
#include <unordered_map>
// 修改后的矩阵乘法函数
SparseMatrix* multiplyMatrices(SparseMatrix* matrixA, SparseMatrix* matrixB) {
    if (matrixA->cols != matrixB->rows)
        return NULL;
    SparseMatrix* resultMatrix = (SparseMatrix*)malloc(sizeof(SparseMatrix));
    resultMatrix->rows = matrixA->rows;
    resultMatrix->cols = matrixB->cols;
    resultMatrix->elements = (MatrixElement*)malloc(sizeof(MatrixElement) * resultMatrix->rows * resultMatrix->cols);
    std::unordered_map<int, std::unordered_map<int, int>> hashB;
    for (int i = 0; i < matrixB->nonZeroCount; i++) {
        int row = matrixB->elements[i].row;
        int col = matrixB->elements[i].col;
        hashB[col][row] = matrixB->elements[i].value;  // 列为键，行和值为键值对
    }
    int count = 0;
    for (int i = 0; i < matrixA->nonZeroCount; i++) {
        int rowA = matrixA->elements[i].row;
        int colA = matrixA->elements[i].col;
        int valueA = matrixA->elements[i].value;
        if (hashB.find(colA) != hashB.end()) {
            for (auto& elementB : hashB[colA]) {
                int rowB = elementB.first;
                int valueB = elementB.second;
                int resultValue = valueA * valueB;
                if (resultValue != 0) {
                    resultMatrix->elements[count].row = rowA;
                    resultMatrix->elements[count].col = rowB;
                    resultMatrix->elements[count].value = resultValue;
                    count++;
                }
            }
        }
    }

    resultMatrix->nonZeroCount = count;
    return resultMatrix;
```

5 用户使用说明

使用 gcc 编译生成可执行文件。

```
gcc -o main -std=c11 main.c matrix.c
```

执行可执行文件：

```
./main
```

在 Windows cmd 下：

```
main
```

6 测试结果

6.1 测试1-基础测试

【输入】

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

【输出】

```
 1 1
 1 1
```

【输入】

```
2
2 2
3 0
0 7
2 2
4 6
7 7
```

【输出】

```
12 18
49 49
```

【输入】

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

```
0 1
4 0
```

【输出】

```
9 2
4 3
```

6.2 测试2-边界条件测试

【输入】

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

【输出】

```
不可相加：行列数不相等
```

【输入】

```
2
2 2
0 0
0 0
2 2
0 0
0 0
```

【输出】

```
乘积矩阵为：
0 0
0 0
```

6.3 测试3-大样本测试

【输入】

```
2
8 8
282 0 708 0 449 0 0 39
685 79 0 0 385 0 638 0
745 0 244 0 658 795 0 0
923 985 0 0 0 0 152 0
602 167 0 0 0 143 0 0
568 233 35 0 0 0 0 0
0 0 284 0 310 23 0 625
542 0 303 293 286 0 387 510
8 8
578 781 779 0 0 0 0 0
```

```
0 531 0 940 106 464 0 735
346 141 287 273 0 746 804 665
917 0 0 0 0 0 409 788
0 0 176 784 0 383 13 879
927 881 899 146 118 0 0 709
42 84 0 254 0 0 90 0
552 0 0 955 33 33 33 11
```

【输出】

```
乘积矩阵为:
429492 320070 501898 582545 1287 701422 576356 865920
422726 630526 601375 538152 8374 184111 62425 396480
1251999 1316644 1480896 698554 93810 434038 204730 1304297
539878 1256666 719017 964508 104410 457040 13680 723975
480517 684822 597515 177858 34576 77488 0 224132
340414 572266 452517 228575 24698 134222 28140 194530
464585 60307 156745 920805 23339 351219 252991 484532
984569 498533 559515 892291 16830 352406 418827 689383
```

7 可视化-优化耗时

我们对4.4中的改进思路补充，使用以下代码比较两种算法的运行时间，和计算优化时间。

```
// 调用原始的乘法函数
start_time = clock();
SparseMatrix* resultOriginal = multiplyMatricesOriginal(matrixA, matrixB);
end_time = clock();
double time_original = (double)(end_time - start_time) / CLOCKS_PER_SEC;
printf("原始乘法耗时: %f 秒\n", time_original);

// 调用使用哈希表的乘法函数
start_time = clock();
SparseMatrix* resultOptimized = multiplyMatrices(matrixA, matrixB);
end_time = clock();
double time_optimized = (double)(end_time - start_time) / CLOCKS_PER_SEC;
printf("优化后乘法耗时: %f 秒\n", time_optimized);
```

使用6.3 的大样本测试进行测试，

结果如下：

```
685 79 0 0 385 0 638 0
745 0 244 0 658 795 0 0
923 985 0 0 0 0 152 0
602 167 0 0 0 143 0 0
568 233 35 0 0 0 0 0
0 0 284 0 310 23 0 625
542 0 303 293 286 0 387 510
请输入矩阵B行数和列数: 8 8
请输入矩阵B:
578 781 779 0 0 0 0 0
0 531 0 940 106 464 0 735
346 141 287 273 0 746 804 665
917 0 0 0 0 0 409 788
0 0 176 784 0 383 13 879
927 881 899 146 118 0 0 709
42 84 0 254 0 0 90 0
552 0 0 955 33 33 33 11
优化后乘法耗时: 0.000012秒
乘积矩阵为:
714528 76259 326703 289326 158889 978539 11844 170910
457629 82759 761101 889087 8294 750024 92826 411879
620686 438628 920868 683165 347429 987615 31290 459189
1302779 523035 580451 908559 1976 1723406 135186 514512
478383 155029 338517 552034 54769 705181 39312 337023
537542 123723 239426 520856 6160 763274 43428 313536
221236 502907 514291 492500 608168 735021 0 17864
549313 680586 1004780 1057177 736361 1212947 132016 606818
```

在这个样本下优化了0.000012秒。