Lab0 实验报告

学号：521021910952

姓名：杜心敏

1. 数据结构设计

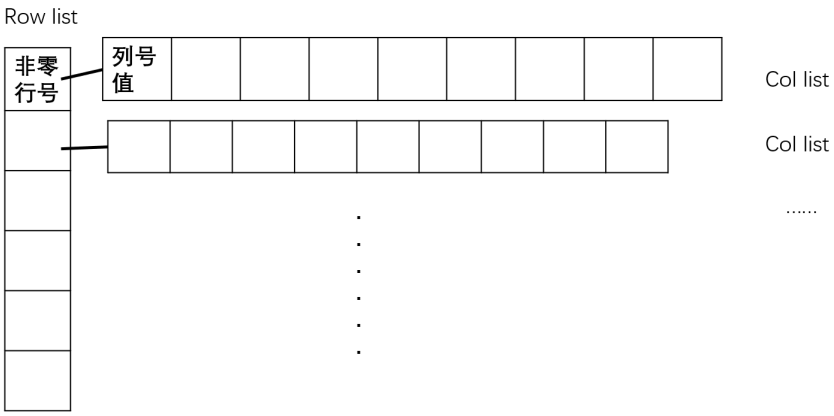

图1 数据结构

总体结构类似`HashMap`的形式，但由于输入有序，且无需查询操作，所以与hash表不同，非零行号连续存放，与其下标没有关系。由于不涉及插入、删除操作，没有采取链式存储，选择`vector`存储（数组也可）。

结构：

- `Col_value`：保存非零列号`col`，数值`val`
- `Row_list`：保存非零行号`row`，以及该行的一条数据记录`vector< Col_value* >`

```
struct Col_value{
    int col;
    int val;
};

struct Row_list{
    int row;
    vector<Col_value*> list;
};

vector<Row_list*> Matrix;
```

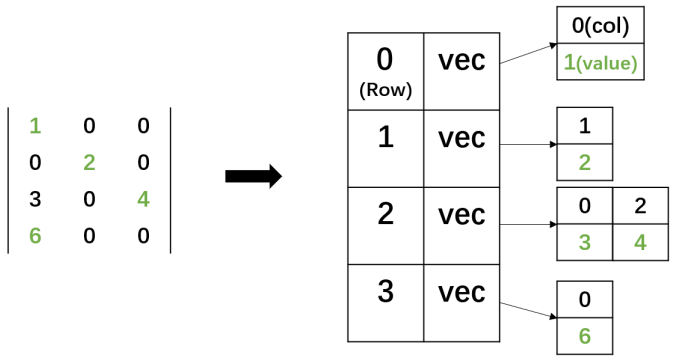

图2 存储示例

存储示例如上。非零行号按序存在连续空间上，同时存储该行上所有非零列号和对应的数值（连续空间）。

2.时空复杂度分析

存储的时空复杂度

存储单个$M \times N$，非零元素占比（稀疏程度）p

- 普通二维数组存储法：$O(M \times N)$
- 该结构：$O(pM \times N)$

矩阵乘法时空复杂度

> 设左矩阵大小为M x N，右矩阵大小为 N x K，总体稀疏程度p

本次矩阵乘法实现如下：

```
for left_row := LeftMatrix.begin()  to  LeftMatrix.end()
    while left != left_row.end() && right_row != RightMatrix.end() do
        if left.col == right.row  then
            for right := right_row.begin() to right_row.end()
                tmp[right.col] += left * right
            left++
            right_row++
        if left.col < right.row  then  left++
        if left.col < right.row  then  right_row++

    for i := 0 to ans.col
        if tmp[i] != 0  then
            ans_Row.push_back(Col_value(i, tmp[i]))
```

1. 取出左矩阵的一行`left_row`
2. 同时遍历该行`left_row`中的元素的**列号**，以及右矩阵的**行号**
3. 当左矩阵元素列号`left_row`等于右矩阵某一行的行号`right.row`时，用左矩阵该**元素**`left`依次乘右矩阵对应行的每一个元素，存放在**暂存区**`tmp`数组，下标为**右矩阵元素的列号**`right.col`。
4. 左矩阵一行`left_row`遍历完后，`tmp`数组就是乘积矩阵的一行，行号为左矩阵该行的行号`left.row`

- 最坏的结果是左矩阵非零元素全部**遍历一遍**，右矩阵遍历了`left.row`次。

- 额外空间：用于暂存累加值的`tmp`数组，长度为`right.col`。
- 运算复杂度：$O(pM \times N + pM \times N \times K) = O(pM \times N \times K)$

普通矩阵乘法：

- 无需额外空间
- 运算复杂度：$O(M \times N \times K)$

3.实验探究

表1 实验数据记录

矩阵规模(M,N,K)	稀疏程度	普通算法耗时	优化算法耗时	普通算法空间消耗	优化算法空间消耗
(57,31,404)	0.59%	0.004	0	37319	710
(57,31,404)	1.22%	0.004	0.001	37319	1229
(57,31,404)	4.43%	0.008	0.004	37319	6491
(57,31,404)	8.98%	0.024	0.015	37319	19976
(57,31,404)	13.21%	0.033	0.030	37319	37823
(57,31,404)	19.96%	0.056	0.050	37319	58244
(57,31,404)	57.5%	0.109	0.089	37319	94139
(57,31,404)	79.1%	0.074	0.076	37319	103421
(125,240,540)	0.49%	0.121	0.002	227100	4098
(125,240,540)	0.98%	0.129	0.006	227100	10086
(125,240,540)	5.04%	0.163	0.091	227100	117285
(125,240,540)	10.01%	0.237	0.187	227100	125805960
(1000,2000,3000)	0.93%	64.586	1.74	11×10^6	1.86×10^6
(1000,2000,3000)	4.98%	71.181	8.693	11×10^6	10.1×10^6
(25,12,42)	22.14%	0.001	0.001	1854	1842
(35,101,52)	15.19%	0.007	0.008	10607	8968

1. 其中稀疏程度计算方法为：$\frac{\text{非零元素个数}}{MN+NK}$
2. 元素大小范围：[0,100]
3. 耗时使用`time.h`中的`clock()`函数计时，单位为秒
4. 空间消耗以一个`int`为单位，例如优化算法中三元组占3个单位，包括新构建矩阵的空间。

数据分析

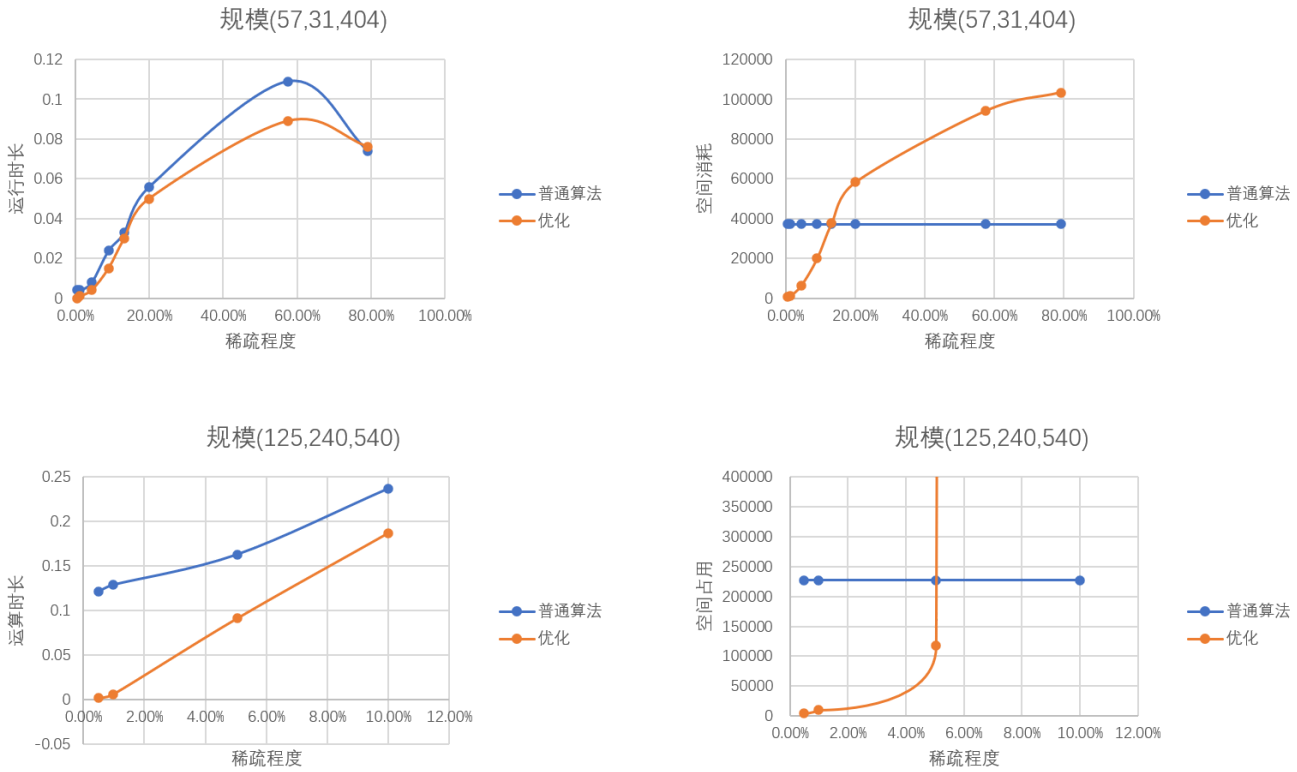

图3 时空消耗对比

- 普通算法：由于每次都是遍历全部，运算速度随p的变化相对较小（图1由于时间较短，曲线起伏略明显）；空间消耗不随p变化。
- 优化后：运算速度随p变化较大，但由于减少了遍历次数，算法速度较快；

  空间消耗受p影响较大。p较小时，有明显优势。分别在10%和5%左右空间消耗程度激增，失去优势。

2. 比较不同规模，算法失效的p的临界值

> 注：由于减少遍历次数，优化后算法运算速度普遍快于普通算法，此处“失效”由空间消耗程度衡量。源数据在表1中。

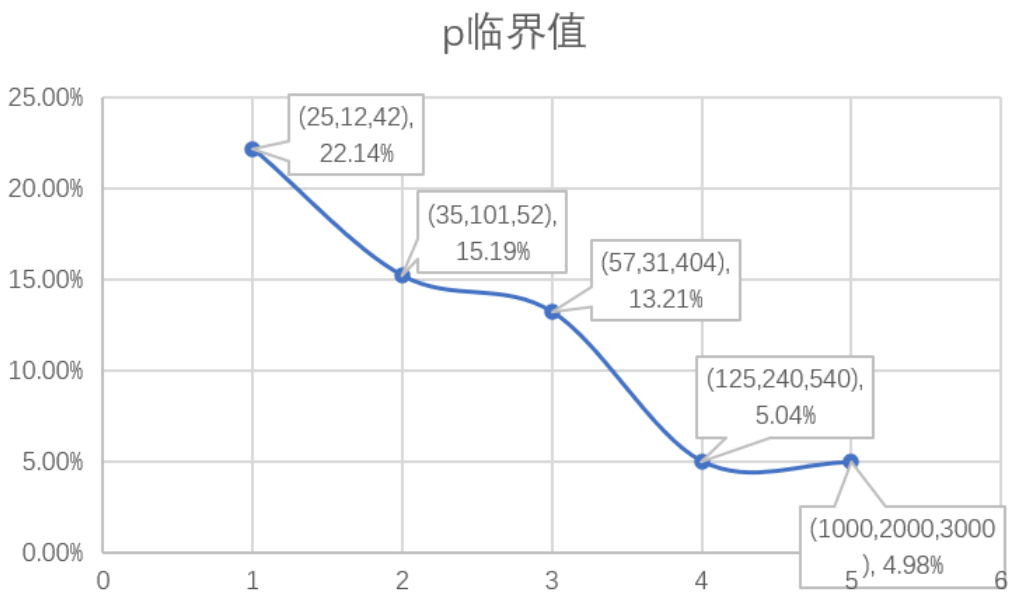

图4 不同规模下p临界值

实验结论

- 程序简便程度：在p较大、元素数据较小的时候，使用普通的矩阵算法较为方便，且空间存储量较小。
- 空间优势：当矩阵的行列在10^3数量级以内的，p的临界值在5%左右，稀疏度大于5%SparseMatrix没有优势。
- 时间优势：由于优化后的算法减少了遍历次数，普遍比原算法快。

4. 实验反思

1. 实验中衡量空间消耗程度，仅仅用存储的`int`型数据的数量衡量，没有考虑结构体中`vector`指针以及`vector`实现时额外的空间消耗。
2. 矩阵规模过大且p较高时，输出文件过大导致出现错误。所以没有对更大规模的矩阵进行p的临界值实验。