山东大学 计算机科学与技术 学院

数据结构与算法 课程实验报告

学号：202300130183	姓名： 宋浩宇	班级：23 级人工智能班
实验题目：2024 数据结构--数据/智能 实验 5 数组和矩阵		
实验学时：2	实验日期：2024/10/9	

实验目的：
掌握稀疏矩阵结构的描述及操作的实现。

软件开发工具：
1. visual studio code 2022（使用 C/C++、C/C++ Extension Pack、C/C++ Themes 插件）
2. mingw64 工具包

1. 实验内容
完成 2024 数据结构--数据/智能 实验 5 数组和矩阵 A：稀疏矩阵。
创建稀疏矩阵类（参照课本 MatrixTerm 三元组定义），采用行主顺序把稀疏矩阵非 0 元素映射到一维数组中，实现操作：两个稀疏矩阵相加、两个稀疏矩阵相乘、稀疏矩阵的转置、输出矩阵。
2. 数据结构与算法描述 （整体思路描述，所需要的数据结构与算法）
首先是读取和储存这个稀疏矩阵，我们使用三元组来存储，即行、列、值三个属性为一组描述一个非零的点，在输入时分为两种，一种是直接输入行列大小和这个三元组，另一种是输入矩阵的每一项，直接输入三元组比较简单，依次存储即可，输入矩阵每一项也只需要在每次输入一个值后检查一下这个值是否为 0，不为零则记录下当前所输入的行列数以及这个值，并存入三元组。我们的三元组使用优化后的实验 2 写的 arraylist 来存储，并创建一个结构体 point，它有三个成员，分别是 row、col、val，依次存储行列值。以上的内容也可以用于矩阵的重置操作。再说矩阵的输出，我们可以通过行数和列数来写一个循环，依次遍历每一个位置，我们可以通过每一个位置创建一个 point，其中 row、col 为在检查的位置，我们通过重载结构体 point 的==运算符，来确认这组索引是否存在于这个三元组表中，如果在则输出对应的值，如果不在则输出 0。在说稀疏矩阵的加法，相加的两项行列数不相等的时候会输出-1 并保留加号后项的值，对于可以正常计算的矩阵首先我们创建一个行列数与加法左右两项行列数相等的空的稀疏矩阵用于存储结果，我们依次遍历要加和的两个矩阵，先将其中一个矩阵中的所有 point 都复制进来，再遍历另一个矩阵中的每一个 point，检查当前矩阵的已有 point，检查其中是否有行列数与另一个矩阵中的处理项相等的项，如果有，则将处理项的值加给这个 point，如果不存在，则将这个 point 加入结果矩阵的三元组表。经过以上计算之后可以得到结果矩阵的三元组表。再是乘法，如果左矩阵的列数与右矩阵的行数不相等，则输出-1，并返回右矩阵，对于合法的数据，先创建一个左矩阵行数、右矩阵列数的空稀疏矩阵用于存储结果，首先对于一个矩阵的中的每一项，它与另一个矩阵中的哪些项会相乘、相乘后的结果会加到哪里都是确定的，因此对于左矩阵中的一个在 row 行 col 列的元素，他会与另一个矩阵中的每一列的第 col 行相乘，并加到第 row 行的每一列上，因此我们可以分别处理这个矩阵中的每一项，至此我们可以获取到一组三元表，我们通过重载三元组表的+运算符来完成这些三元组表的相加，即可得到乘法结果的稀疏矩阵的三元组表。转置是所有计算里最简单且快速的，只需要调换稀疏矩阵所存储的行数和列数，再将三元组表里的每一项的行数和列数交换即可。
3. 测试结果（测试输入，测试输出）
测试输入为：

7
1
5 5
2 1 0 0 0
0 0 -1 0 0
0 0 0 0 0
0 0 -1 0 0
0 0 0 0 0
3
5 5
4
2 2 5
3 5 8
4 4 2
5 3 4
4
2
5 5
3
1 1 8
2 4 4
3 5 2
4
5
4
输出为：
5 5
2 1 0 0 0
0 5 -1 0 0
0 0 0 0 8
0 0 -1 2 0
0 0 4 0 0
5 5
16 0 0 4 0
0 0 0 20 -2
0 0 0 0 0
0 0 0 0 -2
0 0 0 0 8
5 5
16 0 0 0 0
0 0 0 0 0
0 0 0 0 0
4 20 0 0 0
0 -2 0 -2 8

4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

首先我们的算法可以正确的计算出结果，但不幸的是算法的效率比较低，在操作数比较多的

时候容易超时，为解决这个问题我们对计算的算法进行优化，首先是输入输出，本题会使用大量的输入输出，我们通过设置输入缓冲 ios::sync_with_stdio(false);来对输入输出过程进行提速，而矩阵输出的时候会涉及的多次对于矩阵的三元组表的查询操作，因此我们再为 arraylist 添加 self_quick_sort()方法和 binary_search()方法，即快速排序和二分查找，快速排序和二分查找算法都可以使用迭代器来实现，为完成快速排序的比较，我们再为 point 重载<和>运算符。我们在输出之前先执行一次 self_quick_sort()，这个方法会检查序列是否有序，无序则调用快速排序算法。在序列有序后，在使用二分查找来获取点的输出信息。对于乘法和加法，我们主要优化点在三元表的相加的运算符重载上，在这个里边也会用到大量的查询，因此我们在每次相加之前使用 self_quick_sort()方法进行一次两个三元组表的排序，再将两者相加，实际将以上这些优化都使用之后，速度依旧达不到题目的要求，我们继续优化两个三元组表的相加，我们先检查两个表的数据个数，将数据数较多的那个表进行排序，创建一个临时的与数据量较小的表的数据个数相等的 bool 数组，分别在排序后的表中查询这个数据量较小的表里的每一项，如果对应的位置在那个表里存在，则将 bool 数组中的对应索引位进行一个标记，并将那一项的值加到有序表中，查询完之后我们可以知道哪些点已经被加和过了，在这之后我们根据这个 bool 数组将未被加和的项插入这个三元组表，至此优化完成。本题通过，我们的算法成功在题目要求的时间内解决了问题。

5. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）

```
#include <iostream>
#include <stdio.h>
#define debug cout<<__LINE__<<": "<<__FUNCTION__<<endl;
using namespace std;

template<class T>
void my_swap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}


template<class T>
class my_iterator
{
    private:
        T* current;
        unsigned int index;
    public:
        my_iterator(T* current, unsigned int index = 0) : current(current), index(index) {}
        T* getCurrent(){ return current; }
        T& getData()  const { return *current; }
        unsigned int getIndex()  const{ return index; }
        void operator++() { current++; index++; }
        void operator--() { current--; index--; }
        void operator++(int) { current++; index++; }
```

```
29.           void operator--(int) { current--; index--; }
30.           bool operator==(const my_iterator& my_iterator) const { return index == m
    y_iterator.getIndex(); }
31.           bool operator!=(const my_iterator& my_iterator) const { return index != m
    y_iterator.getIndex(); }
32.           bool operator<(const my_iterator& my_iterator) const { return index < my_
    iterator.getIndex(); }
33.           bool operator>(const my_iterator& my_iterator) const { return index > my_
    iterator.getIndex(); }
34.           unsigned int getIndex() { return index; }
35.           void reset(T* current) { this->current = current; }
36.   };
37.
38.
39.
40.
41.   template<class T>
42.   void quick_sort(my_iterator<T> head, my_iterator<T> tail)
43.   {
44.       if (head == tail || tail < head) { return; }
45.       T pivot = *(head.getCurrent());
46.       my_iterator<T> i = head;
47.       my_iterator<T> j = tail;
48.       while (i < j)
49.       {
50.           while (*(j.getCurrent()) >= pivot && i < j)
51.           {
52.               j--;
53.           }
54.           if (i<j)
55.           {
56.               *(i.getCurrent()) = *(j.getCurrent());
57.           }
58.           while (*(i.getCurrent()) <= pivot && i < j)
59.           {
60.               i++;
61.           }
62.           if (i<j)
63.           {
64.               *(j.getCurrent()) = *(i.getCurrent());
65.           }
66.           if (i==j)
67.           {
68.               *(i.getCurrent()) = pivot;
69.           }
70.       }
```

```
71.         my_iterator<T> j2 = j;
72.         j2++;
73.         quick_sort(head, j);
74.         quick_sort(j2, tail);
75.     }
76.
77.     template<class T>
78.     class arraylist
79.     {
80.     private:
81.         T* datas;
82.         int count;
83.         int capacity;
84.         bool is_sorted;
85.         void copy(T* copyed, T* to, int acount = -1);
86.
87.     public:
88.         T& operator[](const unsigned int subscript) { return datas[subscript]; };
89.         const T& operator[](const unsigned int subscript) const { return datas[subscr
    ipt]; };
90.         int find_first(const T& target) const;
91.         void push_back(const T& element);
92.         void erase(const T& target);
93.         void clear();
94.         void double_capacity();
95.         int getCapacity() const{return capacity;};
96.         bool is_in(const T& target) const;
97.         int size() const { return count; };
98.         void self_quick_sort() { if (count == 0) { return; }quick_sort(my_iterator<T>
    (datas, 0), my_iterator<T>(datas + count - 1, count - 1)); is_sorted = true; };
99.         bool getSorted() const { return is_sorted; };
100.        arraylist();
101.        ~arraylist() { delete[] datas; };
102.        arraylist(const arraylist<T>& other);
103.        void operator=(const arraylist<T>& other);
104.    };
105.
106.    template<class T>
107.    void arraylist<T>::copy(T* copyed, T* to,int acount)
108.    {
109.        if (acount == -1)
110.        {
111.            acount = this->count;
112.        }
113.        for (size_t i = 0; i < acount; i++)
114.        {
```

```
115.         to[i] = copyed[i];
116.     }
117. }
118. 
119. template<class T>
120. void arraylist<T>::double_capacity()
121. {
122.     capacity *= 2;
123.     T* new_datas = new T[capacity];
124.     copy(datas, new_datas, count);
125.     delete[] datas;
126.     datas = new_datas;
127.     is_sorted = false;
128. }
129. 
130. 
131. 
132. template<class T>
133. void arraylist<T>::operator=(const arraylist<T>& other)
134. {
135.     capacity = other.getCapacity();
136.     count = other.size();
137.     datas = new T[capacity];
138.     copy(other.datas, datas, count);
139.     is_sorted = other.getSorted();
140. }
141. 
142. 
143. template<class T>
144. void arraylist<T>::push_back(const T& element)
145. {
146.     // if (count == 0)
147.     // {
148.     //     datas = new T[1];
149.     //     datas[0] = element;
150.     //     count = 1;
151.     //     capacity = 1;
152.     //     return;
153.     // }
154.     if (count + 1 > capacity)
155.     {
156.         double_capacity();
157.     }
158.     datas[count] = element;
159.     is_sorted = false;
160.     count++;
```

```
}


template<class T>
int arraylist<T>::find_first(const T& target) const
{
    if (is_sorted == false)
    {
        for (size_t i = 0; i < count; i++)
        {
            if (target == datas[i])
            {
                return i;
            }
        }
        return -1;
    }
    if (is_sorted == true)
    {
        int head = 0;
        int tail = count - 1;
        while (head <= tail)
        {
            int mid = head + ((tail - head) / 2);

            if (target == datas[mid])
            {
                return mid;
            }
            if (target < datas[mid])
            {
                tail = mid - 1;
            }
            if (target > datas[mid])
            {
                head = mid + 1;
            }
        }
    }
    return -1;

}


template<class T>
```

```
207. void arraylist<T>::clear()
208. {
209.     if (datas!= nullptr)
210.     {
211.         delete[] datas;
212.     }
213.     count = 0;
214.     capacity = 16;
215.     is_sorted = false;
216.     datas = new T[capacity];
217. 
218. }
219. 
220. 
221. template<class T>
222. void arraylist<T>::erase(const T& target)
223. {
224.     int index = find_first(target);
225. 
226.     T* new_datas = new T[count - 1];
227. 
228.     for (size_t i = 0; i < count; i++)
229.     {
230.         if (i < index)
231.         {
232.             new_datas[i] = datas[i];
233.         }
234.         if (i == index)
235.         {
236.             continue;
237.         }
238.         if (i > index)
239.         {
240.             new_datas[i - 1] = datas[i];
241.         }
242.     }
243. 
244.     delete[] datas;
245. 
246.     count--;
247. 
248.     datas = new_datas;
249. }
250. 
251. template<class T>
252. bool arraylist<T>::is_in(const T& target) const
```

```
253.  {
254.      int index = this->find_first(target);
255.
256.      if (index != -1)
257.      {
258.          return true;
259.      }
260.      if (index == -1)
261.      {
262.          return false;
263.      }
264.      return false;
265.  }
266.
267.  template<class T>
268.  arraylist<T>::arraylist()
269.  {
270.      datas = nullptr;
271.      count = 0;
272.      capacity = 16;
273.      is_sorted = false;
274.
275.      datas = new T[capacity];
276.
277.  }
278.
279.  template<class T>
280.  arraylist<T>::arraylist(const arraylist<T>& other)
281.  {
282.      capacity = other.getCapacity();
283.      count = other.size();
284.      datas = new T[capacity];
285.      copy(other.datas, datas, count);
286.  }
287.
288.
289.
290.  struct point
291.  {
292.      int row, col;
293.      int value;
294.  };
295.
296.  bool operator==(const point& a, const point& b)
297.  {
298.      if (a.row == b.row && a.col == b.col)
```

```
    {
        return true;
    }
    return false;
}

bool operator!=(const point& a, const point& b)
{
    return !(a == b);
}

bool operator<(const point& a, const point& b)
{
    if (a.row < b.row)
    {
        return true;
    }
    if (a.row == b.row && a.col < b.col)
    {
        return true;
    }
    return false;
}

bool operator>(const point& a, const point& b)
{
    return !(a < b || a == b);
}

bool operator<=(const point& a, const point& b)
{
    return a < b || a == b;
}

bool operator>=(const point& a, const point& b)
{
    return a > b || a == b;
}

void operator<<(const ostream& os, const point& a)
{
    cout << "(" << a.row << "," << a.col << "," << a.value << ")";
}


class SparseMatrix {
```

```
345. private:
346.     int row, col;
347.     int count;
348.     arraylist<point> values;
349. public:
350.     SparseMatrix(int r, int c);
351.     SparseMatrix(int r, int c, int count);
352.     SparseMatrix(const SparseMatrix& other);
353.     SparseMatrix();
354.     void print();
355.     void reset(int r, int c);
356.     int getRow() const { return row; }
357.     int getCol() const{ return col; }
358.     int getCount() const { return count; }
359.     void resetRow(int r) { row = r; }
360.     void resetCol(int c) { col = c; }
361.     void resetCount(int c) { count = c; }
362.     arraylist<point>& getValues() { return values; }
363.     const arraylist<point>& getValues() const { return values; }
364.     void transpose();
365.     SparseMatrix operator+(SparseMatrix& other);
366.     SparseMatrix operator*(SparseMatrix& other);
367. };
368. 
369. 
370. SparseMatrix::SparseMatrix(int r, int c)
371. {
372.     row = r;
373.     col = c;
374.     count = 0;
375.     values = arraylist<point>();
376. 
377.     for (size_t i = 0; i < row; i++)
378.     {
379.         for (size_t j = 0; j < col; j++)
380.         {
381.             int value;
382.             scanf("%d", &value);
383.             if (value != 0)
384.             {
385.                 point p;
386.                 p.row = i;
387.                 p.col = j;
388.                 p.value = value;
389.                 values.push_back(p);
390.                 count++;
```

```
            }
        }
    }
    this->values.self_quick_sort();
}

SparseMatrix::SparseMatrix()
{
    row = 0;
    col = 0;
    count = 0;
    values = arraylist<point>();
}

SparseMatrix::SparseMatrix(const SparseMatrix& other) :
    values(other.getValues())
{
    this->row = other.getRow();
    this->col = other.getCol();
    this->count = other.getCount();
}

SparseMatrix::SparseMatrix(int r, int c, int count)
{
    row = r;
    col = c;
    this->count = count;
    values = arraylist<point>();

    for (size_t i = 0; i < count; i++)
    {
        point p;
        scanf("%d %d %d", &p.row, &p.col, &p.value);
        p.row--;
        p.col--;
        values.push_back(p);
    }

}

SparseMatrix SparseMatrix::operator+(SparseMatrix& other)
{
    if (this->row != other.getRow() || this->col != other.getCol())
    {
        return other;
    }
```

```
437.     else
438.     {
439.         SparseMatrix result;
440.         result.resetRow(this->row);
441.         result.resetCol(this->col);
442.         result.resetCount(this->count);
443.         result.getValues() = this->values;
444.         const arraylist<point>& other_values = other.getValues();
445.         arraylist<point>& result_values = result.getValues();
446.         for (size_t i = 0; i < other.getCount(); i++)
447.         {
448.             point p = other_values[i];
449.             int index = result_values.find_first(p);
450.             if (index == -1)
451.             {
452.                 result_values.push_back(p);
453.                 result_values.self_quick_sort();
454.                 result.resetCount(result.getCount() + 1);
455.             }
456.             else
457.             {
458.                 result_values[index].value += other_values[i].value;
459.             }
460.         }
461.         return result;
462.     }
463. }
464.
465. SparseMatrix SparseMatrix::operator*(SparseMatrix& other)
466. {
467.     if (this->col != other.getRow())
468.     {
469.         return other;
470.     }
471.     else
472.     {
473.         SparseMatrix result;
474.         result.resetRow(this->row);
475.         result.resetCol(other.getCol());
476.         result.resetCount(0);
477.         const arraylist<point>& other_values = other.getValues();
478.         arraylist<point>& result_values = result.getValues();
479.         //todo 稀疏矩阵简易乘法算法
480.         for (size_t i = 0; i < count; i++)
481.         {
482.             point p = values[i];
```

```
483.                point new_p;
484.                new_p.row = p.row;
485.                for (size_t j = 0;j < other.getCount();j++)
486.                {
487.                    point q = other_values[j];
488.                    if (p.col == q.row)
489.                    {
490.                        new_p.col = q.col;
491.                        new_p.value = p.value * q.value;
492.                        int index = result_values.find_first(new_p);
493.                        if (index != -1)
494.                        {
495.                            result_values[index].value += new_p.value;
496.                        }
497.                        else
498.                        {
499.                            result_values.push_back(new_p);
500.                            result_values.self_quick_sort();
501.                            result.resetCount(result.getCount() + 1);
502.                        }
503.                    }
504.                }
505.            }
506.        return result;
507.        }
508.    }
509.
510.    void SparseMatrix::transpose()
511.    {
512.        my_swap(this->row, this->col);
513.        for (size_t i = 0; i < count; i++)
514.        {
515.            my_swap(values[i].row, values[i].col);
516.        }
517.    }
518.
519.    void print(const SparseMatrix& matrix)
520.    {
521.        static int tem_matrix[501][501];
522.        printf("%d %d\n", matrix.getRow(), matrix.getCol());
523.        const arraylist<point>& values = matrix.getValues();
524.        for (size_t i = 0; i < matrix.getCount(); i++)
525.        {
526.            tem_matrix[values[i].row][values[i].col] = values[i].value;
527.        }
528.        for (size_t i = 0; i < matrix.getRow(); i++)
```

```
529.     {
530.         for (size_t j = 0; j < matrix.getCol(); j++)
531.         {
532.             printf("%d ", tem_matrix[i][j]);
533.             tem_matrix[i][j] = 0;
534.         }
535.         printf("\n");
536.     }
537. }
538. 
539. 
540. void SparseMatrix::print()
541. {
542.     printf("%d %d\n", row, col);
543.     for (size_t i = 0; i < row; i++)
544.     {
545.         for (size_t j = 0; j < col; j++)
546.         {
547.             point p;
548.             p.row = i;
549.             p.col = j;
550.             int index = values.find_first(p);
551.             if (index != -1)
552.             {
553.                 printf("%d ", values[index].value);
554.             }
555.             else
556.             {
557.                 printf("0 ");
558.             }
559.         }
560.         printf("\n");
561.     }
562. }
563. 
564. void SparseMatrix::reset(int r, int c)
565. {
566.     row = r;
567.     col = c;
568.     count = 0;
569. 
570.     values.clear();
571. 
572.     for (size_t i = 0; i < row; i++)
573.     {
574.         for (size_t j = 0; j < col; j++)
```

```
575.         {
576.             int value;
577.             scanf("%d", &value);
578.             if (value != 0)
579.             {
580.                 point p;
581.                 p.row = i;
582.                 p.col = j;
583.                 p.value = value;
584.                 values.push_back(p);
585.                 count++;
586.             }
587.         }
588.     }
589. 
590. }
591. 
592. class Solution
593. {
594. public:
595.     void solute();
596.     void test();
597. };
598. 
599. void Solution::test()
600. {
601. 
602.     int row, col;
603.     cin >> row >> col;
604.     SparseMatrix matrix(row, col);
605.     matrix.print();
606.     matrix.transpose();
607.     matrix.print();
608. }
609. 
610. void Solution::solute()
611. {
612. 
613.     int n;
614.     scanf("%d", &n);
615.     int operation;
616.     scanf("%d", &operation);
617.     int row, col;
618.     scanf("%d %d", &row, &col);
619.     SparseMatrix matrix(row, col);
620.     for (int i = 1;i < n;i++)
```

```
    {
        scanf("%d", &operation);
        if (operation == 1)
        {
            int row, col;
            scanf("%d %d", &row, &col);
            matrix.reset(row, col);
        }
        else if (operation == 2)
        {
            int row, col, count;
            scanf("%d %d %d", &row, &col, &count);
            SparseMatrix matrix1(row, col, count);
            matrix = matrix*matrix1;
        }
        else if (operation == 3)
        {
            int row, col, count;
            scanf("%d %d %d", &row, &col, &count);
            SparseMatrix matrix1(row, col, count);
            matrix = matrix + matrix1;
        }
        else if (operation == 4)
        {
            matrix.print();
        }
        else if (operation == 5)
        {
            matrix.transpose();
        }

    }
}

int main()
{
    Solution solution;
    // solution.test();
    solution.solute();
    return 0;
}
```