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：201900161140 | 姓名： 张文浩 | | 班级： 19级人工智能 |
| 实验题目：数组和矩阵 | | | |
| 实验学时：2 | | 实验日期： 9.28 | |
| 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：  Vscode | | | |
| 1. 实验内容   **1、题目描述**：  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。  重置矩阵：操作1，即重置矩阵 P 的尺寸为 n 行 m 列,且随后按行优先顺序输入矩阵 P 的各个元素。  矩阵乘法：操作2，t 行非零元素已按行优先顺序给出，矩阵中非零元素的表示为 x y v，其中 x 表示行序号，y 表示列序号，v 表示非零元素值，行列序号从 1 开始。设输入的矩阵为Q，若PxQ运算合法,则将PxQ的结果矩阵赋给 P,若不合法,则将Q赋给P，同时输出-1。  矩阵加法：操作3，t 行非零元素已按行优先顺序给出，矩阵中非零元素的表示为 x y v，其中 x 表示行序号，y 表示列序号，v 表示非零元素值，行列序号从 1 开始。设输入的矩阵为 Q,若 P+Q 运算合法,则将 P+Q 的结果矩阵赋给 P,若不合法,则将 Q 赋给 P,同时输出 -1。  输出操作：操作4，设当前矩阵 P 的尺寸为 n 行 m 列,第一行输出矩阵 P 的行数和列数，随后 n 行按行优先顺序输出矩阵 P,每行 m 个数字,来表示当前的矩阵内容，每行数字之间用空格分隔。  转置操作：操作5，设当前矩阵 P 的尺寸为 n 行 m 列，将其转置为 m 行 n 列的矩阵，无需输出。  **输入输出格式**：  **输入**：  第一行一个w代表操作个数，接下来若干行是各个操作，其中保证第一个操作一定为重置矩阵。  **输出：**  当执行操作4时，输出矩阵P；当执行操作2或3时，若对应运算不合法，则输出-1。   1. 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）   定义一个三元组结构体matrixTerm，表示非零元素的行猎和数值。  定义稀疏矩阵的类sparseMatrix，其中成员变量包括，rows总行数，cols总列数，tot非零元素总个数，以及\*terms三元组结构体数组的头节点指针。在类中封装为实现题目要求操作的方法函数。包括构造函数、析构函数、init初始化/重置矩阵、multiply矩阵乘法、add矩阵加法，transpose矩阵转置、pushback插入元素  **init(int row, int col)：**  将对象中的rows行总数和cols列总数设置为输入的row和col，再根据输入的矩阵，依次向后设置三元组  **multiply(sparseMatrix &x)矩阵乘法函数**：  首先判断两个矩阵相乘的维度是否匹配，否则输出-1。这里要建立三个数组分别是      int numrow[maxx];               //numrow[i]表示x矩阵第i行中有多少个非零的数      int prerow[maxx];               //prerow[i]表示x矩阵前i-1行已经一共有多少个非零的数了  int lineans[maxx];              //暂存一行的结果  遍历被乘矩阵每一行，计算结果矩阵每一行，假设被乘矩阵有n列，则没计算一行阶输出结果的过程中就要维护一个大小为n的lineans数组，记录结果矩阵一行的结果。计算完一行后将这一行交给结果矩阵，再算下一行。  **add(sparseMatrix &x)矩阵加法函数**  首先判断两个矩阵相乘的维度是否匹配，否则输出-1。采用双指针算法，通过两个系数矩阵的三元组结构体中的row和col计算谁的位置靠前，如果a的位置比b的位置靠前，及说明a的这个位置上b矩阵没有元素，所以可以直接在结果矩阵这个位置上插入a矩阵的这个位置的元素。同理，如果b的位置比a的位置靠前，及说明b的这个位置上a矩阵没有元素，所以可以直接在结果矩阵这个位置上插入b矩阵的这个位置的元素。如果a和b两个指针在同一个位置，就说明a和b两个矩阵在这个位置上都有元素，就可以在结果矩阵中对应位置插入一个三元组，value数值为a和b两个对应位置的value相加即可。、  **insert(int pos, const matrixTerm &x)在指定位置插入三元组：**  要插入位置后面元素都向后挪一位，然后在空出来的位置上设置三元组的值为输入的x。  **transpose()转置函数：**  将转置后的矩阵存在矩阵res中，首先设置res的行数和列数，然后使线性表res.terms的元素个数等于被转置的元素个数。尽管这时的线性表res.terms还没有元素，但还是要令他的元素个数等于它最后具有的元素个数，这一步是必要的，因为这样就可以把转之后的矩阵元素存储到线性表res.terms的任意相应位置。否则，就要借助插入操作，每插入一个元素，增加一次线性表的元素个数。可是，当我们煮至一个稀疏矩阵时，原来的第0个元素可能成为转置后的第6个元素，而除非线性表的元素个数是6或更多，否则我们是不能把一个元素插到第6个位置上去的。因此，我们实质上是把线性表作为一个一维数组，一开始的元素个数等于它的最终的元素个数，然后给任意一个位置赋一个新值。  与稀疏矩阵快速乘法一样，为快速找到矩阵非零元素的位置，要维护两个数组，colSize和rowNext，因为矩阵转置之后的行是之前的列，所以与之前multify矩阵乘法中的两个数组不同的是，转置中的两个数组是对列进行操作的。colSize表示矩阵中每一列的非零元素的个数，rowNext[i]表示前i-1行有了多少个非零元素了  在拥有这两个数组之后就可以构建转置后的矩阵了，构建时主义插入结果矩阵的顺序按照行排序优先的顺序进行插入，即需要计算转置后矩阵中元素对应的下标是啥，具体实现方法见附录源代码。  **重载<<输出**  先输出矩阵的rows和cols信息，在用i和j从1开始遍历行和列，里面再用一个for循环从三元组结构体中寻找，如果对应的i和j有对应的三元组结构体，就在这个位置输出对应元素的数值，如果没有，就说明这个位置是0。   1. 测试结果（测试输入，测试输出）   C:\WINDOWS\TEMP\WeChat Files\220596131dbe137a35fb0ee923cf7d5.jpg   1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）   我感觉本次实验的难点是矩阵乘法和矩阵转置函数中的快速计算操作，要利用稀疏矩阵的性质，不能简单地用两个for循环遍历二维矩阵，因为稀疏矩阵的性质是矩阵中由大量的0元素，而且我们采用了三元组结构体的方式定义稀疏矩阵，可以充分利用三元组结构体的性质，加快乘法和转置操作。具体思想是构造两个数组，第一个数组记录每一行/列有多少个非零的元素，第二个数组记录当前行之前已经有了多少个非零的元素，这样利用的这两个矩阵就可以快速定位到三元组结构体数组中对应的三元组元素。  稀疏矩阵类实现的难点在于乘法。由于缺少矩阵完整的结构，需要其他的数组辅助储存一些数据以便于乘法的执行。在最开始编写乘法的算法时我采用了填充矩阵之后相乘的方式，但很快发现这种计算方式效率过低。于是采用了另一种方法：利用另一个数组存储每行第一个非零元素在类中数组的下标，这样便可以很快地定位到每一行的非零元素。之后的乘法根据这个元素所在的列数，存放到记录运算结果的数组中，累计这些结果的和就可获得该位置在进行矩阵乘法运算之后的数值大小。  另外，对于类似本次实验这样输入结果与输出结果不分离的形式，可以利用freopen函数将输出保存到txt文件中，便于调试时结果的查看。Visual studio中对应函数为freopen\_s。   1. 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>  #define maxx 200000  using namespace std;  struct matrixTerm  {      int row, col, value;      matrixTerm &operator=(matrixTerm &x)      {          row = x.row;          col = x.col;          value = x.value;          return \*this;      }  };  class sparseMatrix  {      friend ostream &operator<<(ostream &out, sparseMatrix &x); //输出矩阵  public:      sparseMatrix(int row, int col);              //构造函数      ~sparseMatrix() { delete[] terms; }          //析构函数      sparseMatrix &operator=(sparseMatrix &x);    //重载=操作符      void init(int row, int col);                 //初始化/重置矩阵      void multiply(sparseMatrix &x);              //乘法函数      void add(sparseMatrix &x);                   //加法函数      void transpose();                            //转置函数      void insert(int index, const matrixTerm &x); //插入函数      void pushback(int x, int y, int z);          //插入函数      void output();  private:      int rows, cols, tot;      matrixTerm \*terms;  };  //构造函数  sparseMatrix::sparseMatrix(int row, int col)  {      terms = new matrixTerm[maxx];      rows = row;      cols = col;      tot = 0;  }  //重载=操作符  sparseMatrix &sparseMatrix::operator=(sparseMatrix &x)  {      rows = x.rows;      cols = x.cols;      tot = x.tot;      for (int i = 0; i < tot; i++)      {          terms[i] = x.terms[i];      }      return \*this;  }  //重载<<输出  ostream &operator<<(ostream &out, sparseMatrix &x)  {      cout << x.rows << " " << x.cols << endl;      for (int i = 1; i < x.rows + 1; i++)      {          for (int j = 1; j < x.cols + 1; j++)          {              bool f = true;              for (int k = 0; k < x.tot; k++)              {                  if (x.terms[k].row == i && x.terms[k].col == j)                  {                      cout << x.terms[k].value << " ";                      f = false;                  }              }              if (f)                  cout << 0 << " ";          }          cout << endl;      }      return out;  }  void sparseMatrix::output()  {      int cnt=0;      cout << rows << " " << cols << endl;      for (int i = 1; i < rows + 1; i++)      {          for (int j = 1; j < cols + 1; j++)          {              if (cnt < tot && terms[cnt].row == i && terms[cnt].col == j)              { //如果此时该位置存在元素 输出即可                  cout << terms[cnt].value << ' ';                  cnt++;              }              else                  cout << 0 << ' ';          }          cout << endl;      }  }  //初始化/重置矩阵  void sparseMatrix::init(int row, int col)  {      rows = row;      cols = col;      tot = 0;      for (int i = 1; i <= row; i++)      {          for (int j = 1; j <= col; j++)          {              int temp;              cin >> temp;              if (temp != 0)              {                  terms[tot].row = i;                  terms[tot].col = j;                  terms[tot].value = temp;                  tot++;              }          }      }  }  //矩阵乘法函数  void sparseMatrix::multiply(sparseMatrix &x)  {      if (cols != x.rows)      { //维度不匹配，矩阵无法相乘          \*this = x;          cout << -1 << endl;          return;      }      sparseMatrix res(rows, x.cols); //结果矩阵      int numrow[maxx];               //numrow[i]表示x矩阵第i行中有多少个非零的数      int prerow[maxx];               //prerow[i]表示x矩阵前i-1行已经一共有多少个非零的数了      int lineans[maxx];              //暂存一行的结果      for (int i = 1; i <= x.rows; i++)      {          numrow[i] = 0; //初始化为0      }      for (int i = 0; i < x.tot; i++)      {          if (x.terms[i].value != 0)          {              numrow[x.terms[i].row]++; //计算这一行有几个非零数          }      }      prerow[1] = 0;      for (int i = 2; i <= x.rows; i++)      {          prerow[i] = prerow[i - 1] + numrow[i - 1]; //计算prerow数组      }      int a = 0;      for (int i = 1; i <= rows && a < tot; i++)      {          for (int j = 1; j <= x.cols; j++)          {              lineans[j] = 0;          }          while (a < tot && terms[a].row == i)          {              int col\_temp = terms[a].col; //被乘矩阵的每一列              if (numrow[col\_temp] != 0)   //如果乘矩阵的对应行中有非零的数              {                  for (int b = prerow[col\_temp]; b < prerow[col\_temp] + numrow[col\_temp]; b++)                  {                      lineans[x.terms[b].col] += terms[a].value \* x.terms[b].value;                  }              }              a++;          }          for (int j = 1; j <= x.cols; j++)          {              if (lineans[j] != 0)              {                  res.terms[res.tot].value = lineans[j];                  res.terms[res.tot].row = i;                  res.terms[res.tot].col = j;                  res.tot++;              }          }      }      \*this = res;  }  //矩阵加法函数  void sparseMatrix::add(sparseMatrix &x)  {      sparseMatrix res(rows, cols); //结果矩阵      if (rows != x.rows || cols != x.cols)      {          \*this = x;          cout << -1 << endl;          return;      }      int restot = 0;   //结果系数矩阵中有多少个数      int a = 0, b = 0; //a,b分别遍历被加数和加数,下面类似于双指针算法      while (a != tot && b != x.tot)      {          int aa = terms[a].row \* cols + terms[a].col;          int bb = x.terms[b].row \* cols + x.terms[b].col;          //看看a和b的位置谁靠前          if (aa < bb)          {              res.insert(restot++, terms[a]); //如果a<b说明这个位置上b是0，结果直接是a              a++;          }          else if (aa > bb)          {              res.insert(restot++, x.terms[b]); //如果a>b说明这个位置上a是0，结果直接是b              b++;          }          else if (aa == bb)          {              if (terms[a].value + x.terms[b].value != 0)              {                  matrixTerm temp;                  temp.row = terms[a].row;                  temp.col = terms[a].col;                  temp.value = terms[a].value + x.terms[b].value;                  res.insert(restot++, temp);              }              a++;              b++;          }      }      for (; a < tot; a++)      {          res.insert(restot++, terms[a]);      }      for (; b < x.tot; b++)      {          res.insert(restot++, x.terms[b]);      }      \*this = res;  }  void sparseMatrix::insert(int pos, const matrixTerm &x)  {      copy(terms + pos, terms + tot, terms + pos + 1); //先把要插入位置后面元素向后挪一位      //copy\_backward(terms + pos, terms + tot, terms + tot + 1);      terms[pos].row = x.row;      terms[pos].col = x.col;      terms[pos].value = x.value;      tot++;  }  void sparseMatrix::transpose()  {      sparseMatrix res(cols, rows);      //设置转置矩阵特征      res.tot = tot;      res.rows = cols;      res.cols = rows;      //初始化以实现转置      int \*colSize = new int[cols + 1];      int \*rowNext = new int[cols + 1];      //寻找\*this中每一列的项的数目      for (int i = 1; i <= cols; i++)      {          colSize[i] = 0;      }      for (int i = 0; i < tot; i++)      {          colSize[terms[i].col]++;      }      rowNext[1] = 0;      for (int i = 2; i <= cols; i++)      {          rowNext[i] = rowNext[i - 1] + colSize[i - 1];      }      for (int i = 0; i < tot; i++)      {          int j = rowNext[terms[i].col]++;          res.terms[j].row = terms[i].col;          res.terms[j].col = terms[i].row;          res.terms[j].value = terms[i].value;      }      \*this = res;  }  //插入函数  void sparseMatrix::pushback(int x, int y, int z)  {      matrixTerm temp;      temp.row = x;      temp.col = y;      temp.value = z;      terms[tot] = temp;      tot++;  }  int main()  {      ios::sync\_with\_stdio(false);      int w;      cin >> w;      int row, col, op, t, x, y, z;      sparseMatrix a(0, 0);      while (w--)      {          cin >> op;          if (op == 1)          {              cin >> row >> col;              a.init(row, col);          }          else if (op == 2)          {              cin >> row >> col;              cin >> t;              sparseMatrix b(row, col);              for (int i = 0; i < t; i++)              {                  cin >> x >> y >> z;                  b.pushback(x, y, z);              }              a.multiply(b);          }          else if (op == 3)          {              cin >> row >> col;              cin >> t;              sparseMatrix b(row, col);              for (int i = 0; i < t; i++)              {                  cin >> x >> y >> z;                  b.pushback(x, y, z);              }              a.add(b);          }          else if (op == 4)          {              //cout << a;              a.output();          }          else if (op == 5)          {              a.transpose();          }      }      //system("pause");      return 0;  }  // //样例输入  // 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  // //样例输出  // 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 | | | |