第5章 数组和稀疏矩阵

本章学习目标

- 理解数组的基本操作和存储方式；
- 掌握特殊矩阵的压缩存储方法；
- 掌握稀疏矩阵的表示方法；
- 掌握快速转置算法和乘法算法的实现方法。

本章重点介绍数组的概念及其抽象数据类型、特殊矩阵的压缩存储和稀疏矩阵的表示。

5.1 数组的概念与表示

数组是计算机运作中非常重要的概念，几乎每种编程语言都有数组。作为一种数据结构，数组可以看作线性表的推广，其特点是结构中的元素本身可以是原子类型，也可以是具有某种结构的数据，但所有元素都属于同一数据类型。

5.1.1 数组的概念

数组是有序排列的相同类型的数据元素的集合。

由上述定义可以得到数组具有如下两个特征。

(1) 数组中的元素是有序的。数组中的元素可以用数组名和下标指定，下标指出了该元素在数组中的顺序。

(2) 数组的元素都是相同类型的。数组中存储的所有元素必须是同一种数据类型，不能是多种数据类型的混合。

除了上述两个特征，通常还认为数组是一种静态的数据结构，即一旦定义了某个数组，它包含的元素个数(数组的大小)将不再改变，不能再增加和删除数组中的元素。可动态改变大小的动态数组不在本书的讨论范围内。因此，对数组的操作一般只有两类：

(1) 获得特定位置的元素值；

(2) 修改特定位置的元素值。

由此可以将数组看作是从下标集到元素集的一个映射。

如果下标依次指出了数组的第 1 个元素、第 2 个元素、……、第 n 个元素，即下标集为{1,2,…,n}，则这样的数组称为一维数组，该数组的长度为 n。在该一维数组中，除最后一个元素(第 n 个元素)外，每个元素都有唯一的一个后继元素。

如果要指定数组中的某个元素，需要两个下标分别指定该元素在数组中位置，这样的数组称为二维数组。二维数组中的所有元素形成了一个有行有列的矩形结构，要指定一个 m 行 n 列的二维数组中的某个元素，需要指定它所在的行下标和列下标，即下标集为{(1,1),(1,2),…,(m,n)}。通常，第 1 个下标为行下标，第 2 个下标为列下标，该数组在行这一维度的长度为 m，在列这一维度的长度为 n。在该二维数组中，除最后一列元素(列下标为 n 的元素)外，每个元素都有一个行相同但位于下一列的后继元素；同理，除最后一行元素(行下标为 m 的元素)外，每个元素都有一个列相同但位于下一行的后继元素。

多维数组可以由此推广得到，n 维数组需要 n 个下标确定数组元素在数组中的位置。

由此可以得到数组的抽象数据类型：

ADT Array{

数据对象：$D=\{a_{j_1j_2\cdots j_n} \mid n(>0)$是数组的维数，$j_i$ 是数组的第 i 维下标，$1\leqslant j_i\leqslant b_i$，$b_i$ 是数组的第 i 维的长度，$a_{j_1j_2\cdots j_n}\in ElemSet\}$

数据关系：$R=\{R1,R2,\cdots,Rn\}$

$Ri=\{<a_{j_1\cdots j_i\cdots j_n},a_{j_1\cdots j_i+1\cdots j_n}>\mid 1\leqslant j_k\leqslant b_k,1\leqslant k\leqslant n$ 且 $k\neq i,$ $1\leqslant j_i\leqslant b_i-1,a_{j_1\cdots j_i\cdots j_n},a_{j_1\cdots j_i+1\cdots j_n}\in D,i=1,2,\cdots,n\}$

基本操作：

InitArray(&A,n,b1,b2,…,bn)

初始条件：维数 n 和各维的长度合法

操作结果：构造相应的数组 A，并返回 TRUE

DestroyArray(&A)

初始条件：数组 A 存在

操作结果：销毁数组 A，并返回 TRUE

GetValue(A,&e,j1,j2,…,jn)

初始条件：n 维数组 A 存在，各下标 j_i 合法

操作结果：用 e 返回数组 A 中由 j1,…,jn 指定的元素的值

SetValue(&A,e,j1,j2,…,jn)

初始条件：n 维数组 A 存在，各下标 j_i 合法

操作结果：将数组 A 中由 j1,…,jn 指定的元素的值置为 e

} **ADT** Array

在上述数组抽象数据类型的描述中，数组中的每个数据元素都对应了一组下标(j_1,…,j_n)，每个下标的取值范围是 $1\leqslant j_i\leqslant b_i$，其中 i=1,2,…,n。

在上述数据关系中，R_i 描述的是第 i 维的后继关系，即数据元素 $a_{j_1\cdots j_i\cdots j_n}$ 在第 i 维的后继元素是 $a_{j_1\cdots j_i+1\cdots j_n}$。仅就单个关系而言，仍是线性关系。

举例来说，当 n=1 时，n 维数组就退化为一维数组，即定长的线性表，假设它的长度为 k，则它的唯一后继关系 $R=\{<a_1,a_2>,<a_2,a_3>,\cdots,<a_{k-1},a_k>|a_i\in D,i=1,2,\cdots,k\}$。

当 n=2 时就是二维数组，假设它第一维的长度为 m，第二维的长度为 n，则它的两个数据关系分别为：行方向上的后继关系 $R_1=\{<a_{ij},a_{i,j+1}>|1\leqslant j\leqslant n-1,a_{ij},a_{i,j+1}\in D,i=1,2,\cdots,m\}$；列方向上的后继关系 $R_2=\{<a_{ij},a_{i+1,j}>|1\leqslant i\leqslant m-1,a_{ij},a_{i+1,j}\in D,j=1,2,\cdots,n\}$。

抽象地讲，n(n>1)维数组也可以看作一个特殊的一维数组，该一维数组的每个元素本身是一个 n−1 维数组。如 n=2 时，如果把二维数组看作一个特殊的一维数组，则该一维数组中的每个元素也是一个一维数组。

例如，下面是以 m 行 n 列矩阵形式表示的二维数组 $A_{m\times n}$。

$$A_{m\times n}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

数组 $A_{m\times n}$可以看作一个由行向量构成的一维数组

$$\begin{aligned} A_{m\times n}&=(a_1,a_2,\cdots,a_m)\\ &=((a_{11},a_{12},\cdots,a_{1n}),(a_{21},a_{22},\cdots,a_{2n}),\cdots(a_{m1},a_{m2},\cdots,a_{mn})) \end{aligned}$$

其中，每个元素 a_i 对应 $A_{m\times n}$的每一行，即 $a_i=(a_{i1},a_{i2},\cdots,a_{in}),1\leqslant i\leqslant m$。

数组 $A_{m\times n}$也可以看作一个由列向量构成的一维数组

$$A_{m\times n}=(a_1,a_2,\cdots,a_n)=\left[\begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix}\begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix}\cdots\begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix}\right]$$

其中，每个元素 a_j 对应 A 的每一列，即 $a_j=(a_{1j},a_{2j},\cdots,a_{mj}),1\leqslant j\leqslant n$。

数组作为一个抽象数据类型，用户可以根据应用的需要增加其他必要的运算，如数组的整体赋值运算等，但不应当违背数组是静态数据结构的特性。

本节定义的数组的抽象数据类型 Array 可以用 C 语言的数组定义机制实现。

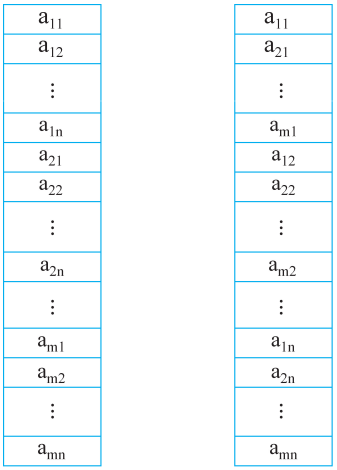
5.1.2　数组的顺序表示

由于数组是一种静态的数据结构且数组元素有序，因此采用顺序存储结构表示数组是非常自然的。对于一维数组，只需要按顺序依次分配数组元素的存储单元即可；但是对于多维数组，当数组元素在一维结构的存储单元中连续存储时，就会产生存储的次序问题。

对于任意二维数组 $A_{m\times n}$，可以从行和列两个方向把它看作一个特殊的一维数组，相应的，二维数组就有按行序为主序和按列序为主序两种存储方式。在实际的高级语言中，不同的语言可能采用不同的存储结构，例如 C 语言采用按行序为主序的存储方式，而 FORTRAN 则是采用按列序为主序的存储方式。

对于二维数组 $A_{m\times n}$，按行序为主序的存储方式和按列序为主序的存储方式如图 5.1

所示。



(a) 按行序为主序　　(b) 按列序为主序

图 5.1　二维数组的两种存储方式

对于二维数组 $A_{m\times n}$，只要给出了首元素 a_{11} 的存储地址，并且知道每个元素所占的存储空间，就可以计算出任意元素 a_{ij} 的地址。

假设二维数组 $A_{m\times n}$ 的每个元素占 L 个存储单元，首元素 a_{11} 的地址为 Loc(1,1)。对于按行序为主序的存储方式，元素 a_{ij} 排在第 i 行、第 j 列，它前面的第 i−1 行有 n×(i−1) 个元素，在第 i 行 a_{ij} 前面还有 j−1 个元素。由此得到 a_{ij} 地址的计算公式为

$$Loc(i,j)=Loc(1,1)+(n\times(i-1)+(j-1))\times L \tag{5.1}$$

对于按列序为主序的存储方式，也可以得到类似的计算公式，在此不再赘述。

对于任意三维数组 $B_{m\times n\times p}$，同样可以把它看作一个特殊的一维数组

$$B_{m\times n\times p}=(b_1,b_2,\cdots,b_m)$$

其中，每个元素 b_i 对应一个 n×p 的二维数组。因此，以第一维的顺序为主序，然后依次考虑第二维的顺序和第三维的顺序，存储三维数组 $B_{m\times n\times p}$ 存储序列为

$$b_{111},b_{112},\cdots,b_{11p},b_{121},b_{122},\cdots,b_{12p},\cdots,b_{1n1},b_{1n2},\cdots,b_{1np},$$
$$b_{211},b_{212},\cdots,b_{21p},b_{221},b_{222},\cdots,b_{22p},\cdots,b_{2n1},b_{2n2},\cdots,b_{2np},\cdots,$$
$$b_{m11},b_{m12},\cdots,b_{m1p},b_{m21},b_{m22},\cdots,b_{m2p},\cdots,b_{mn1},b_{mn2},\cdots,b_{mnp}.$$

同样，给出首元素 b_{111} 的存储地址，并且知道每个元素所占的存储空间，就可以计算出任意元素 b_{ijk} 的地址。

假设三维数组 $B_{m\times n\times p}$ 的每个元素占 L 个存储单元，首元素 b_{111} 的地址为 Loc(1,1,1)。按上面的存储方式，元素 b_{ijk} 与 b_{i11} 在同一个二维数组中，由公式(5.1)

$$Loc(i,j,k)=Loc(i,1,1)+(p\times(j-1)+(k-1))\times L$$

而 b_{i11} 前面有 i−1 个 n×p 的二维数组，因此

$$Loc(i,1,1)=Loc(1,1,1)+(n\times p\times(i-1))\times L$$

由此可得 b_{ijk} 地址的计算公式为

$$Loc(i,j,k)=Loc(i,1,1)+(n\times p\times(i-1)+p\times(j-1)+(k-1))\times L \quad (5.2)$$

推广到一般情况，对于 n 维数组 $C_{b_1b_2\cdots b_n}$，如果以第一维的顺序为主序，然后依次考虑第二、三、…、n 维的顺序，存储 n 维数组 C。若已知首元素 $c_{11\cdots1}$ 的存储地址为 Loc(1,1,…,1)，并且知道每个元素所占的存储空间 L，可得任意元素 $C_{j_1j_2\cdots j_n}$ 地址的计算公式为

$$\begin{aligned}Loc(j_1,j_2,\cdots,j_n)=Loc(1,1,\cdots,1)+(&b_2\times\cdots\times b_{n-1}\times b_n\times(j_1-1)+b_3\times\cdots\\&\times b_{n-1}\times b_n\times(j_2-1)+\cdots+b_n\times(j_{n-1}-1)+(j_n-1))\times L\end{aligned}$$

该公式可简写为

$$Loc(j_1,j_2,\cdots,j_n)=Loc(1,1,\cdots,1)+\Big(\sum_{i=1}^{n-1}(j_i-1)\prod_{k=i+1}^{n}b_k+(j_n-1)\Big)\times L \quad (5.3)$$

注意：以上讨论的前提是数组每维的下标均从 1 开始。在实际应用中，C 语言允许下标下界为 0，而 Pascal 允许下标下界为负整数、0 或正整数。在这种情况下，地址计算公式就与前面的讨论有所不同了，仿照前面的推导过程，读者不难推导出这类数组中任意元素的存储地址的计算公式。

5.1.3　特殊矩阵的压缩存储

5.1.2 节介绍的数组的顺序表示方法适用于具有完整结构的数组，即数组元素的各下标在各自独立的范围内（n 维数组对 $1\leqslant k\leqslant n$，$1\leqslant j_k\leqslant b_k$）改变，并且大部分元素值不为 0。但也有许多情况不适用，例如，对于对称矩阵，上述表示方法将会有大约一半的存储空间是多余的。因此，为了节省存储空间，可以对这些特殊矩阵进行压缩存储。这些特殊矩阵的共同特点是其元素的分布有明显的规律性，从而可将其压缩到一维数组中，并找到每个元素在一维数组中的对应位置。

下面对三种特殊矩阵加以讨论。

1. 下三角矩阵

对于一个 n 阶矩阵 $A_{n\times n}$，若当 i<j 时有 $a_{ij}=0$，则称此矩阵为下三角矩阵，下三角矩阵的形式如下。

$$A_{n\times n}=\begin{bmatrix}a_{11}&0&\cdots&0&\cdots&0\\a_{21}&a_{22}&\cdots&0&\cdots&0\\\vdots&\vdots&\ddots&\vdots&\vdots&0\\a_{i1}&a_{i2}&\cdots&a_{ii}&\cdots&0\\\vdots&\vdots&\vdots&\vdots&\ddots&0\\a_{n1}&a_{n2}&\cdots&a_{ni}&\cdots&a_{nn}\end{bmatrix}$$

对于下三角矩阵的压缩存储，只存储下三角中的元素，其余的 0 元素则不存储。

对于下三角矩阵 $A_{n\times n}$，下三角中共有 n(n+1)/2 个元素，若按行序为主序的原则，可以将矩阵下三角中的所有元素压缩存储到一个长度为 n(n+1)/2 的存储空间中，如图 5.2 所示。

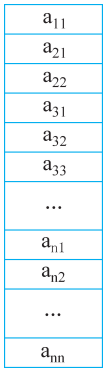


图 5.2　下三角矩阵的存储

若已知首元素 a_{11} 的存储地址为 Loc(1,1)，并且知道每个元素所占的存储空间 L，$A_{n\times n}$ 下三角中任意元素 a_{ij} 前面的 i−1 行共有

1+2+3+…+(i−1)个元素，在第 i 行前面共有 j−1 个元素。由此得到 a_{ij} 地址的计算公式为

$$Loc(i,j)=Loc(1,1)+(i\times(i-1)/2+(j-1))\times L,i\geqslant j \tag{5.4}$$

2. 上三角矩阵

对于一个 n 阶矩阵 $A_{n\times n}$，若当 i>j 时有 $a_{ij}=0$，则称此矩阵为上三角矩阵，上三角矩阵的形式如下。

$$A_{n\times n}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1i} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2i} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{ii} & \cdots & a_{in} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & \cdots & a_{nn} \end{bmatrix}$$

对于上三角矩阵的压缩存储，同下三角矩阵类似，只存储上三角中的元素，对于其余的零元素则不存。

同样，若按行序为主序的原则，可以将矩阵上三角中的所有元素压缩存储到一个长度为 n(n+1)/2 的存储空间中，按以下序列存储：

$$a_{11},a_{12},\cdots,a_{1n},a_{22},a_{23},\cdots,a_{2n},\cdots,a_{nn}$$

因此，可得 a_{ij} 地址的计算公式为

$$Loc(i,j)=Loc(1,1)+((2n-i+2)\times(i-1)/2+(j-i))\times L,i\leqslant j \tag{5.5}$$

3. 对称矩阵

若矩阵中的所有元素均满足 $a_{ij}=a_{ji}$，则称此矩阵为对称矩阵。对称矩阵的形式如下。

$$A_{n\times n}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{12} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{bmatrix}$$

对于对称矩阵，因其元素满足 $a_{ij}=a_{ji}$，可以为每一对相等的元素分配一个存储空间，即只存下三角(或上三角)矩阵，从而将 n^2 个元素压缩到 n(n+1)/2 个存储空间中。

5.2 稀疏矩阵

在许多科学与工程应用中经常会遇到这样一种矩阵：矩阵规模很大，但其中大多数元素的值为零，只有少部分元素的值非零，并且这些非零元素在矩阵中的分布没有规律性，这种矩阵称为稀疏矩阵。稀疏矩阵经常出现在大规模集成电路设计、电力输配系统、图像处理、城市规划等应用领域。

例如，式(5.6)表示的矩阵 M 及其转置矩阵。

$$M=\begin{bmatrix}32&0&0&17&0&0\\0&0&5&9&0&0\\0&0&0&0&0&0\\0&13&0&0&0&0\\41&0&0&0&0&0\\0&0&0&0&12&0\end{bmatrix},\quad M^{T}=\begin{bmatrix}32&0&0&0&41&0\\0&0&0&13&0&0\\0&5&0&0&0&0\\17&9&0&0&0&0\\0&0&0&0&0&12\\0&0&0&0&0&0\end{bmatrix}\tag{5.6}$$

在上述矩阵 M 及其转置矩阵中，在 36 个元素中只有 7 个非零元素，显然是个稀疏矩阵。但非零元素要少到多少才称为稀疏矩阵并无确切定义，这是人们直观进行分辨的概念。

由此，可以得到稀疏矩阵的抽象数据类型：

ADT SMatrix{
　数据对象：D = {a_{ij} | 1≤i≤m,1≤j≤n,
　　a_{ij}∈ElemSet,m 和 n 分别是矩阵的行数和列数 }
　数据关系：R = {Row,Col}
　　Row = {<a_{ij},$a_{i,j+1}$> | 对 1≤j≤n-1,a_{ij},$a_{i,j+1}$∈D,1≤i≤m }
　　Col = {<a_{ij},$a_{i+1,j}$> | 对 1≤i≤m-1,a_{ij},$a_{i+1,j}$∈D,1≤j≤n }
　基本操作：
　CreateSMatrix(&M,n,n)
　操作结果：创建一个 m 行 n 列的稀疏矩阵
　DestroySMatrix(&M)
　初始条件：稀疏矩阵 M 存在
　操作结果：销毁稀疏矩阵 M
　PrintSMatrix(M)
　初始条件：稀疏矩阵 M 存在
　操作结果：输出稀疏矩阵 M
　CopySMatrix(M,&T)
　初始条件：稀疏矩阵 M 存在
　操作结果：将稀疏矩阵 M 复制得到稀疏矩阵 T
　Transpose(M,&T)
　初始条件：稀疏矩阵 M 存在
　操作结果：求 M 的转置矩阵 T
　Multiply(M,N,&P)
　初始条件：稀疏矩阵 M 的列数等于稀疏矩阵 N 的行数
　操作结果：求矩阵 M 与 N 的乘积 P
} **ADT** SMatrix

5.2.1　稀疏矩阵的三元组表示

由于没有特殊矩阵中非零元素分布的规律性，稀疏矩阵的压缩存储要比特殊矩阵复杂。为了节省稀疏矩阵的存储空间，按照压缩存储概念，只需要存储稀疏矩阵中的非零元素。但是，由于非零元素的分布没有规律性，为了保证实现矩阵的各种运算，除了需要存

储非零元素的值外，还必须同时记录它所在的行和列，这样一个三元组(i,j,a_{ij})便能唯一确定矩阵中的一个非零元素，其中 a_{ij} 表示矩阵第 i 行第 j 列非零元素的值。下面 7 个三元组表示了式(5.6)中矩阵 M 的 7 个非零元素：

(1,1,32),(1,4,17),(2,3,5),(2,4,9),(4,2,13),(5,1,41),(6,5,12)

若以某种方式(如上，按行序为主序的顺序)将 7 个三元组排列起来，则形成的表就能唯一确定稀疏矩阵，从而得到稀疏矩阵的一种压缩存储方式，即三元组顺序表，其结构描述如下：

```
//稀疏矩阵的三元组顺序表存储结构
#define MAXSIZE 1000                          //非零元素的个数最多为 1000
typedef struct
{
  int  row,col;                               //该非零元素的行下标和列下标
  ElemType  data;                             //该非零元素的值
} Triple;
typedef struct
{
  Triple   data[MAXSIZE+1];                   //非零元素的三元组顺序表,data[0]未用
  int      m,n,len;                           //矩阵的行数、列数和非零元素的个数
}SMatrix;
```

在此，data 域中表示非零元素的三元组是按行序为主序的顺序排列的。

表 5.1 表示了稀疏矩阵 M 中 data 域的排列情况。

表 5.1 矩阵 M 的三元组顺序表表示

row	col	data
1	1	32
1	4	17
2	3	5
2	4	9
4	2	13
5	1	41
6	5	12

在矩阵足够稀疏的情况下，这种表示方法对存储空间的需求量比通常的方法少得多。例如式(5.6)中的矩阵 M 若用三元组顺序表表示，在每个元素占一个存储单元的情况下，则只需要 21 个存储单元；若直接用二维数组按通常办法表示，则需要 36 个单元。矩阵越大，越稀疏，其优越性越明显。

下面以稀疏矩阵的转置运算和乘法运算为例，介绍采用稀疏矩阵三元组顺序表表示的实现方法。

1. 转置运算

矩阵的转置。通过变换元素的位置，把位于(row，col)位置上的元素换到(col，row)位置上，得到了一个新的矩阵。也就是说，把元素的行列互换。如式(5.6)中矩阵 M 及其转置矩阵。一般地，一个 m×n 的矩阵的转置矩阵就是 n×m 的矩阵。

采用矩阵的正常存储方式时，实现矩阵转置的经典算法如算法 5.1 所示。

【算法 5.1】

```
Transpose(ElemType src[n][m],ElemType dst[m][n])
{
    //src 和 dst 分别为转置前的源矩阵和转置后的结果矩阵(用二维数组表示)
    int i, j;
    for (i=0; i<m; i++)
    for (j=0; j< n; j++)
    dst[j][i]=src[i][j];
}                                                    //Transpose
```

很显然，该算法执行的基本操作就是变换元素的位置，把位于第 i 行第 j 列的元素换到第 j 行第 i 列的位置上，这样的变换操作需要对矩阵中每个元素进行，因此转置运算的时间复杂度为 O(m×n)。如果源矩阵 src 是一个稀疏矩阵，则结果矩阵 dst 也是一个稀疏矩阵。而对于三元组顺序表表示的稀疏矩阵 src，若三元组顺序表中的某一行为(i，j，x)，则结果矩阵 dst 的三元组顺序表中应有(j，i，x)，但如果仅仅是简单地对矩阵 src 的行列进行交换，所得到矩阵 dst 的三元组顺序表将由源矩阵的按行序进行存储变为按列序进行存储，为了保证 dst 的三元组顺序表仍按行序为主序存储，需要对行列互换后的三元组顺序表按行序进行重新排序。

表 5.2 表示了式(5.6)中稀疏矩阵 M 的转置矩阵 M^T 的三元组顺序表表示。

表 5.2　矩阵 M 的转置 M^T 的三元组顺序表表示

row	col	data
1	1	32
1	5	41
2	4	13
3	2	5
4	1	17
4	2	9
5	6	12

为了保证转置后矩阵的三元组顺序表仍按行序为主序存储，使用稀疏矩阵的三元组顺序表存储结构需要按照 col 从小到大的顺序完成元素位置的变换。显然，为了找到 M 中每列的所有非零元素，需要对其 data 域从第 1 行起整个扫描一遍。具体算法描述如算

法 5.2 所示。

【算法 5.2】

```
void Transpose (SMatrix src,  Smatrix* dst)
{
    //把矩阵 src 转置到 dst 所指的矩阵中,矩阵用三元组顺序表存储结构
    int    i ,  j,  k ;
    dst->m= src.n ;  dst ->n= src.m ;  dst ->len= src.len ;
    if(dst->len>0)
    {
        j=1;
        for(k=1;  k<= src.n;  k++)
        for(i=1;  i<= src.len;  i++)
        if(src.data[i].col==k)
        {
            dst->data[j].row = src.data[i].col;
            dst->data[j].col = src.data[i].row;
            dst->data[j].data = src.data[i].data;
            j++;
        }
    }
}                                                    //Transpose
```

算法 5.2 中设置了一个计数器 j,用于指向 src 当前元素转置后应放入 dst 所指的三元组顺序表中的位置。处理完 src 中的一个元素后,j 加 1,j 的初值为 1。

算法使用了一个二重循环:外循环用来控制扫描的次数,每执行一次外循环体,dst 所指的矩阵相当于排好了一行;内循环则用来扫描 src 三元组顺序表中的每一行,并判断表中每行相应的元素在该轮中是否需要转换。算法的时间耗费主要在此二重循环中,其时间复杂度为 O(src.n×src.len)。在最坏的情况下,当非零元素个数 src.len=src.m×src.n 时,算法的时间复杂度为 O(src.n×src.n^2),这比直接采用矩阵转置的经典算法的时间复杂度 O(src.n×src.n)要差。

不难看出,此算法之所以耗费时间,问题在于其每形成转置矩阵的一行,都必须从头到尾扫描 src.data 中的每一行。能否在实施转置时只对 src.data 扫描一次,就能得到按行序为主序存储的 dst->data 呢?为此提出了一种快速转置算法,它依赖于以下条件。

(1) 待转置矩阵 src 每一列中非零元素的个数(转置后 dst 指向矩阵的每一行中非零元素的个数)。

(2) 待转置矩阵 src 每一列中第一个非零元素(转置后 dst 指向矩阵的每一行中第一个非零元素)在 dst->data 中的正确位置。

为此,需要设两个数组 num[]和 position[],其中 num[col]用来存放待转置矩阵 src 第 col 列中非零元素的个数(转置后 dst 指向矩阵第 col 行非零元素的个数),position[col]用来存放待转置矩阵 src 第 col 列(转置后 dst 指向矩阵中第 col 行)中第一个非零元