第5章 chapter 5

数组、特殊矩阵和广义表

本章介绍的数组与广义表可视为线性表的推广，其特点是数据元素仍然是一个表。本章讨论多维数组的逻辑结构和存储结构、特殊矩阵、矩阵的压缩存储、广义表的逻辑结构和存储结构等。

5.1 多维数组

5.1.1 数组的逻辑结构

数组是我们很熟悉的一种数据结构，它可以看作线性表的推广。数组作为一种数据结构，其特点是结构中的元素本身可以是具有某种结构的数据，但属于同一数据类型。例如一维数组可以看作一个线性表，二维数组可以看作“数据元素是一维数组”的一维数组，三维数组可以看作“数据元素是二维数组”的一维数组，以此类推。图 5.1 是一个 m 行 n 列的二维数组。

$$A=\begin{Bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{Bmatrix}$$

图 5.1　m 行 n 列的二维数组

数组是一个具有固定格式和数量的数据有序集，每一个数据元素有唯一的一组下标来标识，因此在数组上不能做插入、删除数据元素的操作。通常在各种高级语言中数组一旦被定义，每一维的大小及上下界都不能改变。在数组中通常做下面两种操作。

取值操作：给定一组下标，读其对应的数据元素。

赋值操作：给定一组下标，存储或修改与其相对应的数据元素。

我们着重研究二维和三维数组，因为它们的应用是广泛的，尤其是二维数组。

5.1.2 数组的内存映像

现在来讨论数组在计算机中的存储表示。通常，数组在内存 ϕ 被映像为向量，即用向量作为数组的一种存储结构，这是因为内存的地址空间是一维的，数组的行列固定后，通过一个映像函数，则可根据数组元素的下标得到它的存储地址。

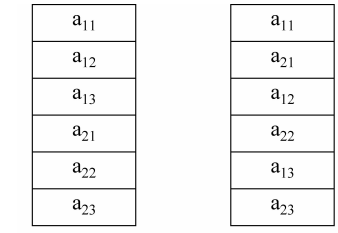
对于一维数组按下标顺序分配即可。

对于多维数组分配时，要把它的元素映像存储在一维存储器中，一般有两种存储方式，一是以行为主序（或先行后列）的顺序存放。如 BASIC、PASCAL、COBOL、C 等程序设计语言中用的是以行为主的顺序分配，即一行分配完了接着分配下一行，另一种是以列为主序（先列后行）的顺序存放，如 FORTRAN 语言中，用的是以列为主序的分配顺序，即一列一列地分配。以行为主序的分配规律是：最右边的下标先变化，即最右下标从小到大。循环一遍后，右边第二个下标再变化，……，从右向左，最后是左下标。以列为主序分配的规律恰好相反：最左边的下标先变化，即最左下标从小到大。循环一遍后，左边第二个下标再变化，……，从左向右，最后是右下标。

例如一个 2×3 二维数组，逻辑结构如图 5.2 所示。以行为主序的内存映像如图 5.3(a)所示。分配顺序为 a_{11}，a_{12}，a_{13}，a_{21}，a_{22}，a_{23}；以列为主序的分配顺序为 a_{11}，a_{21}，a_{12}，a_{22}，a_{13}，a_{23}。它的内存映像如图 5.3(b)所示。

a_{11}	a_{12}	a_{13}
a_{21}	a_{22}	a_{23}

图 5.2 2×3 数组的逻辑状态



(a) 以行为主序　　(b) 以列为主序

图 5.3 2×3 数组的物理状态

设有 m×n 二维数组 A_{mn}，下面来看按元素的下标求其地址的计算方法。

以“以行为主序”的分配为例：设数组的基址为 $LOC(a_{11})$，每个数组元素占据 I 个地址单元，那么 a_{ij} 的物理地址可用一线性寻址函数计算：

$$LOC(a_{ij})=LOC(a_{11})+((i-1)\times n+j-1)*I$$

这是因为数组元素 a_{ij} 的前面有 i−1 行，每一行的元素个数为 n，在第 i 行中它的前面还有 j−1 个数组元素。

在 C 语言中，数组中每一维的下界定义为 0，则：

$$LOC(a_{ij})=LOC(a_{00})+(i*n+j)*I$$

推广到一般的二维数组：$A(c_1\cdots d_1)(c_2\cdots d_2)$，则 a_{ij} 的物理地址计算函数为：

$$LOC(a_{ij})=LOC(a_{c_1c_2})+(i-c_1)*(d_2-c_2+1)+(j-c_2)*I$$

同理对于三维数组 A_{mnp}，即 m×n×p 数组，对于数组元素 a_{ijk} 其物理地址为：

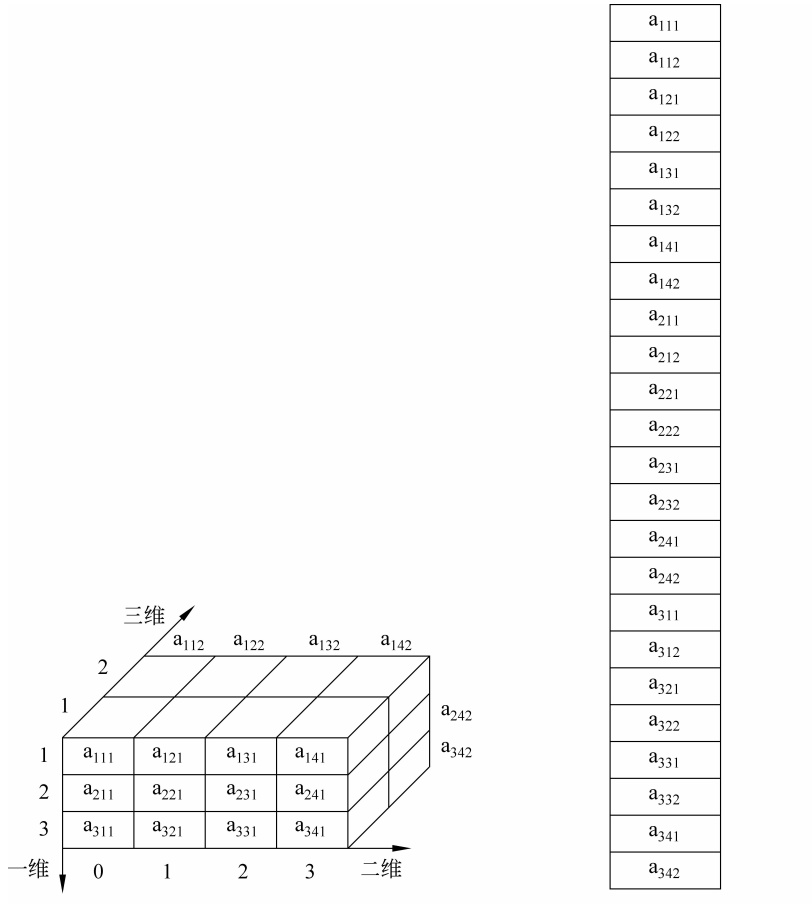
$$LOC(a_{ijk})=LOC(a_{111})+((i-1)\times n\times p+(j-1)*p+k-1)*I$$

推广到一般的三维数组：$A(c_1\cdots d_1)(c_2\cdots d_2)(c_3\cdots d_3)$，则 a_{ijk} 的物理地址为：

$$LOC(i,j)=LOC(a_{c_1c_2c_3})+((i-c_1)*(d_2-c_2+1)*(d_3-c_3+1)$$
$$+(j-c_2)*(d_3-c_3+1)+(k-c_3))*I$$

三维数组的逻辑结构和以行为主序的分配示意图如图 5.4 所示。

【例 5.1】 若矩阵 $A_{m\times n}$ 中存在某个元素 a_{ij} 满足：a_{ij} 是第 i 行中的最小值且是第 j 列中的最大值，则称该元素为矩阵 A 的一个鞍点。试编写一个算法，找出 A 中的所有鞍点。



(a) 一个3×4×2的三维数组的逻辑结构　　(b) 以行为主序的三维数组内存映像

图 5.4　三维数组示意图

基本思想：在矩阵 A 中求出每一行的最小值元素，然后判断该元素是否为其所在列中的最大值，是则打印出来，接着处理下一行。矩阵 A 用一个二维数组表示。

算法如下：

```
void saddle(int A[ ][ ],int m,int n)
    /* m,n 是矩阵 A 的行和列 */
{  int i,j,min;
     for(i=0;i<m;i++)                    /* 按行处理 */
{
    min=A[i][0]
        for(j=1; j<n; j++)
            if(A[i][j]<min)min=A[I][j];  /* 找第 I 行最小值 */
        for(j=0;j<n; j++)                /* 检测该行中的每一个最小值是否是鞍点 */
if(A[I][j]==min)
            {  k=j; p=0;
```

```
            while(p<m&& A[p][j]<min)
                p++;
            if( p>=m)printf("%d,%d,%d\n",i,k,min);
        }                                   /* if */
    }                                       /* for i */
}
```

算法的时间复杂度为 O(m×(n+m×n))。

5.2 特殊矩阵的压缩存储

对于一个矩阵结构显然用一个二维数组来表示是非常恰当的,但在有些情况下,例如常见的一些特殊矩阵,如三角矩阵、对称矩阵、带状矩阵、稀疏矩阵等,从节约存储空间的角度考虑,这种存储是不太合适的。下面从这一角度来考虑这些特殊矩阵的存储方法。

5.2.1 对称矩阵

对称矩阵的特点是:在一个 n 阶方阵中,有 $a_{ij}=a_{ji}$,其中 $1\leqslant i,j\leqslant n$,如图 5.5 所示是一个 5 阶对称矩阵。对称矩阵关于主对角线对称,因此只需存储上三角或下三角部分即可,例如只存储下三角中的元素 a_{ij},其特点是 $j\leqslant i$ 且 $1\leqslant i\leqslant n$,对于上三角中的元素 a_{ij},它和对应的 a_{ji} 相等,因此当访问的元素在上三角时,直接去访问和它对应的下三角元素即可,这样,原来需要 n×n 个存储单元,现在只需要 n(n+1)/2 个存储单元了,节约了 n(n−1)/2 个存储单元,当 n 较大时,这是可观的一部分存储资源。

$$A=\begin{bmatrix}3&6&4&7&8\\6&2&8&4&2\\4&8&1&6&9\\7&4&6&0&5\\8&2&9&5&7\end{bmatrix}$$

3	6	2	4	8	1	7	4	6	0	8	2	9	5	7

图 5.5 5 阶对称方阵及它的压缩存储

如何只存储下三角部分呢?对下三角部分以行为主序顺序存储到一个向量中去,在下三角中共有 n×(n+1)/2 个元素,因此,不失一般性,设存储到向量 $SA_{[n(n+1)/2]}$ 中,存储顺序可用图 5.6 示意,这样,原矩阵下三角中的某一个元素 a_{ij} 则具体对应一个 $SA_{[k]}$,下面的问题是要找到 k 与 i、j 之间的关系。

对于下三角中的元素 a_{ij},其特点是 $i\geqslant j$ 且 $1\leqslant i\leqslant n$,存储到 SA 中后,根据存储原则,它前面有 i−1 行,共有 1+2+…+i−1=i×(i−1)/2 个元素,而 a_{ij} 又是它所在的行中的第 j 个,所以在上面的排列顺序中,a_{ij} 是第 i×(i−1)/j 个元素,因此它在 SA 中的下标 k 与 i、j 的关系为:

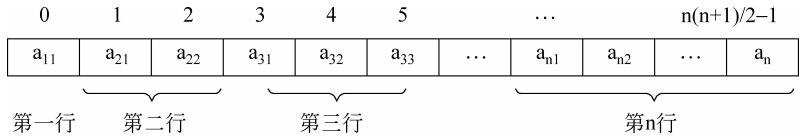


图 5.6　一般对称矩阵的压缩存储

$$k=i\times(i-1)/2+j-1 \quad (0\leqslant k<n\times(n+1)/2)$$

若 $i<j$，则 a_{ij} 是上三角中的元素，因为 $a_{ij}=a_{ji}$，这样，访问上三角中的元素 a_{ij} 时则去访问和它对应的下三角中的 a_{ij} 即可，因此将上式中的行列下标交换就是上三角的元素在 SA 中的对应关系：

$$k=j\times(j-1)/2+i-1 \quad (0\leqslant k<n\times(n+1)/2)$$

综上所述，对于对称矩阵中的任意元素 a_{ij}，若令 $I=\max(i,j)$，$J=\min(i,j)$，则将上面两个式子综合起来得到：$k=I*(I-1)/2+J-1$。

5.2.2　三角矩阵

形如图 5.7 所示的矩阵称为三角矩阵，其中 c 为某个常数。图 5.7(a)为下三角矩阵，主对角线以上均为同一个常数；图 5.7(b)为上三角矩阵，主对角线以下为同一个常数。下面讨论它们的压缩存储方法。

$$\begin{bmatrix} 3 & c & c & c & c \\ 6 & 2 & c & c & c \\ 4 & 8 & 1 & c & c \\ 7 & 4 & 6 & 0 & c \\ 8 & 2 & 9 & 5 & 7 \end{bmatrix} \qquad \begin{bmatrix} 3 & 4 & 8 & 1 & 0 \\ c & 2 & 9 & 4 & 6 \\ c & c & 1 & 5 & 7 \\ c & c & c & 0 & 8 \\ c & c & c & c & 7 \end{bmatrix}$$

(a) 下三角矩阵　　(b) 上三角矩阵

图 5.7　三角矩阵

1. 下三角矩阵

与对称矩阵类似，不同之处在于存完下三角中的元素之后，紧接着存储对角线上方的常量，因为是同一个常数，所以存一个即可，这样一共存储了 $n\times(n+1)+1$ 个元素，设存入向量：$SA_{[n\times(n+1)+1]}$ 中，如图 5.8 所示。这种的存储方式可节约 $n\times(n-1)-1$ 个存储单元，SA_k 与 a_{ij} 的对应关系为：

$$k=\begin{cases} i\times(i-1)/2+j-1 & \text{当 } i\geqslant j \\ n\times(n+1)/2-1 & \text{当 } i<j \end{cases}$$

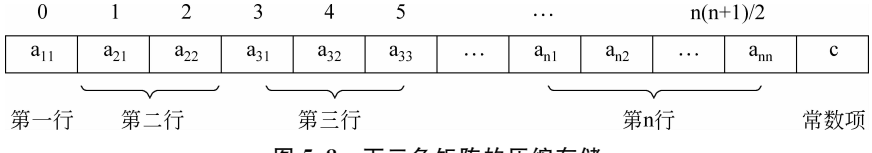


图 5.8　下三角矩阵的压缩存储

2. 上三角矩阵

对于上三角矩阵，存储思想与下三角类似，以行为主序顺序存储上三角部分，最后存储对角线下方的常量。对于第 1 行，存储 n 个元素，第 2 行存储 n－1 个元素，……，第 p 行存储(n－p＋1)个元素，如图 5.9 所示。a_{ij}的前面有 i－1 行，共存储：

$$n+(n-1)+\cdots+(n-i+1)=\sum_{p=1}^{i-1}(n-p)+1=(i-1)\times(2n-i+2)/2$$

个元素，而 a_{ij}是它所在的行中要存储的第(j－i＋1)个，所以，它是上三角存储顺序中的第(i－1)×(2n－i＋2)/2＋(j－i＋1)个，因此它在 SA 中的下标为：

$$k=(i-1)\times(2n-i+2)/2+j-i$$

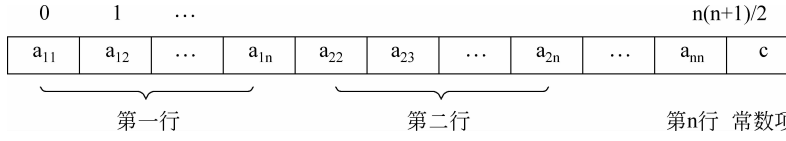


图 5.9 上三角矩阵的压缩存储

综上，SA_k 与 a_{ij}的对应关系为：

$$k=\begin{cases}(i-1)\times(2n-i+2)/2+j-1 & \text{当 } i\geqslant j\\ n\times(n+1)/2-1 & \text{当 } i<j\end{cases}$$

5.2.3 带状矩阵

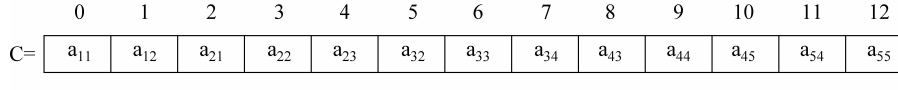
n 阶矩阵 A 称为带状矩阵，如果存在最小正数 m，满足当|i－j|≥m 时，$a_{ij}=0$，这时称 w＝2n－1 为矩阵 A 的带宽。如图 5.10(a)是一个 w＝3(m＝2)的带状矩阵。带状矩阵也称为对角矩阵。由图 5.10(a)可看出，在这种矩阵中，所有非零元素都集中在主对角线的带状区域中，即除了主对角线和它的上下方若干条对角线的元素外，所有其他元素都为零(或同一个常数 c)。

$$A=\begin{bmatrix}a_{11}&a_{12}&0&0&0\\a_{21}&a_{22}&a_{23}&0&0\\0&a_{32}&a_{33}&a_{34}&0\\0&0&a_{43}&a_{44}&a_{45}\\0&0&0&a_{54}&a_{55}\end{bmatrix}$$

(a) w=3的5阶带状矩阵

$$B=\begin{bmatrix}a_{11}&a_{12}&0\\a_{21}&a_{22}&a_{23}\\a_{32}&a_{33}&a_{34}\\a_{43}&a_{44}&a_{45}\\a_{54}&a_{55}&0\end{bmatrix}$$

(b) 压缩为5×3的矩阵



(c) 压缩为向量

图 5.10 带状矩阵及压缩存储

带状矩阵 A 也可以采用压缩存储。一种压缩方法是将 A 压缩到一个 n 行 w 列的二维数组 B 中，如图 5.10(b)所示，当某行非零的个数小于带宽 w 时，先存放非零元素后补零，那么 a_{ij} 映射为 b_{ij}，映射关系为：

$$i' = i$$

$$j = \begin{cases} j & (\text{当 } i \leqslant m) \\ j - i + m & (\text{当 } i > m) \end{cases}$$

另一种压缩方法是将带状矩阵压缩到向量 C 中去，按以行为主序，顺序的存储其非零元素，如图 5.10(c)所示，按其压缩规律，找到相应的映像函数。

如当 w=3 时，映像函数为：

$$K = 2 \times i + j - 3$$

5.3 稀疏矩阵

设 m×n 矩阵中有 t 个非零元素且 $t \ll m \times n$，这样的矩阵称为稀疏矩阵。很多科学管理及工程计算中，常会遇到阶数很高的大型稀疏矩阵。如果按常规分配方法，顺序分配在计算机内，那将是相当浪费内存的。为此提出另一种存储方法，仅仅存放非零元素。但对于这类矩阵，通常零元素分布没有规律，为了能找到相应的元素，所以仅存储非零元素的值是不够的，还要记下它所在的行和列。于是采取如下方法：将非零元素所在的行、列以及它的值构成一个三元组(i,j,v)，然后再按某种规律存储这些三元组，这种方法可以节约存储空间。下面讨论稀疏矩阵的压缩存储方法。

5.3.1 稀疏矩阵的三元组表存储

将三元组按行优先的顺序，同一行中列号从小到大的规律排列成一个线性表，称为三元组表，采用顺序存储方法存储该表。图 5.11 所示的稀疏矩阵对应的三元组表如图 5.12 所示。显然，要唯一地表示一个稀疏矩阵，还需要在存储三元组表的同时存储该矩阵的行、列，为了运算方便，矩阵的非零元素的个数也同时存储。这种存储的思想实现如下：

$$A=\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

图 5.11 稀疏矩阵

	i	j	v
1	1	1	15
2	1	4	22
3	1	6	−15
4	2	2	11
5	2	3	3
6	3	4	6
7	5	1	91

图 5.12 三元组表

```
define SMAX 1024                          /* 一个足够大的数 */
typedef struct
{
  int i,j;                                /* 非零元素的行、列 */
datatype v;                               /* 非零元素值 */
} SPNode;                                 /* 三元组类型 */
typedef struct
{
int mu,nu,tu;                             /* 矩阵的行列及非零元素的个数 */
SPNode data[SMAX];                        /* 三元组表 */
}SPMatrix;                                /* 三元组表的存储类型 */
```

这样的存储方法确实节约了存储空间，但矩阵的运算从算法上可能变得复杂些。下面我们讨论这种存储方式的稀疏矩阵的两种运算：转置和相乘。

1. 稀疏矩阵的转置

设 SPMatrix A；表示一个 m×n 的稀疏矩阵，其转置 B 则是一个 n×m 的稀疏矩阵，因此也有 SPMatrix B。

A 的行、列转化成 B 的列、行。

将 A.data 中每一个三元组的行列交换后转化成 B.data 中。

看上去以上两点完成之后，似乎完成了 B，但并没有。因为前面规定三元组是按一行一行且每行中的元素是按列号从小到大的规律存放的，因此 B 也必须按此规律存放，A 的转置 B 如图 5.13 所示，图 5.14 是它对应的三元组存储，就是说，在 A 的三元组存储基础上得到 B 的三元组存储(为了运算方便，矩阵的行列都从 1 算起，三元组表 data 也从 1 单元用起)。

$$B=\begin{bmatrix} 15 & 0 & 0 & 0 & 91 & 0 \\ 0 & 11 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 6 & 0 & 0 \\ 22 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

图 5.13 A 的转置 B

	i	j	v
1	1	1	15
2	1	5	91
3	2	2	11
4	3	2	3
5	4	1	22
6	4	3	6
7	6	1	−15

图 5.14 B 的三元组表

算法思路：

(1) A 的行、列转化成 B 的列、行；

(2) 在 A.data 中依次找第一列的、第二列的、直到最后一列，并将找到的每个三元组的行、列交换后顺序存储到 B.data 中即可。

算法如下：

```
void TransM1(SPMatrix * A)
{
SPMatrix * B;
int p,q,col;
B=malloc(sizeof(SPMatrix));                    /* 申请存储空间 */
B->mu=A->nu;B->nu=A->mu;B->tu=A->tu;
    /* 稀疏矩阵的行、列、元素个数 */
if(B->tu>0)                                    /* 有非零元素则转换 */
{  q=0;
for(col=1;col<=(A->nu);col++)                  /* 按 A 的列序转换 */
    for(p=1;p<=(A->tu);p++)                    /* 扫描整个三元组表 */
        if (A->data[p]j==col)
    {  B->data[q].i=A->data[p].j;
       B->data[q].j=A->data[p].i;
       B->data[p].v=A->data[p].v;
       q++;}                                   /* if */
}                                              /* if(B->tu>0) */
    return B;                                  /* 返回的是转置矩阵的指针 */
}                                              /* TranM1 */
```

算法 5.1　稀疏矩阵转置

分析该算法，其时间主要耗费在 col 和 p 的二重循环上，所以时间复杂度为 O(n×t)，(设 m、n 是原矩阵的行、列，t 是稀疏矩阵的非零元素个数)，显然当非零元素的个数 t 和 m×n 同数量级时，算法的时间复杂度为 $O(m\times n^2)$，和通常存储方式下矩阵转置算法相比，可能节约了一定量的存储空间，但算法的时间性能更差一些。

算法 5.1 的效率低的原因是算法要从 A 的三元组表中寻找第一列、第二列、……，要反复搜索 A 表，若能直接确定 A 中每一三元组在 B 中的位置，则对 A 的三元组表扫描一次即可。这是可以做到的，因为 A 中第一列的第一个非零元素一定存储在 B. data[1]，如果还知道第一列的非零元素的个数，那么第二列的第一个非零元素在 B. data 中的位置便等于第一列的第一个非零元素在 B. data 中的位置加上第一列的非零元素的个数，以此类推，因为 A 中三元组的存放顺序是先行后列，对同一行来说，必定先遇到列号小的元素，这样只需扫描一遍 A. data 即可。

根据这个想法，需引入两个向量来实现：num[n+1]和 cpot[n+1]，num[col]表示矩阵 A 中第 col 列的非零元素个数(为了方便均从 1 单元用起)，cpot[col]初始表表示矩阵 A 中的第 col 列的非零元素的个数(为了方便均从 1 单元用起)，cpot[col]初始值表示矩阵 A 中的第 col 列的第一个非零元素在 B. data 中的位置。于是 cpot 的初始值：

```
cpot[1]=1;
cpot[col]=cpot[col-1]+num[col-1];  2≤col≤n
```

例如对于图 5.11 所示的矩阵 A 的 num 和 cpot 的值如图 5.15 所示。

依次扫描 A.data，当扫描到 col 列元素时，直接将其存放在 B.data 的 cpot[col]位置上，cpot[col]加 1，cpot[col]中始终是下一个 col 列元素在 B.data 中的位置。

col	1	2	3	4	5	6
num[col]	2	1	1	2	0	1
cpot[col]	1	3	4	5	7	7

图 5.15 矩阵 A 的 num 与 cpot 值

下面按以上思路改进转置算法如下：

```
SPMatrix * TransM2(SPMatrix * A)
{  SPMatrix * B;
   int i,j,k;
   int num[n+1],cpot[n+1];
   B=malloc(sizeof(SPMatrix));                    /* 申请存储空间 */
   B->mu=A->nu;B->nu=A->mu;B->tu-=A->tu;          /* 稀疏矩阵的行、列元素个数 */
     if(B->tu>0)                                  /* 有非零元素则转换 */
     {
     for(i=1;i<=A->nu;i++)num[j]=0;
       for(i=1;i<=A->tu;i++)                      /* 求矩阵 A 中每一列非零元素个数 */
           {
               j=A->data[i],j;
                   num[j]++;
           }
           cpot[1]=1;        /* 求矩阵 A 中每一列第一个非零元素在 B.data 中的位置 */
           for(i=2;i<=A->nu;i++)
               cpot[i]=cpot[]i-1]+num[i-1];
           for(i=1;i<=A->tu;i++)                  /* 扫描三元组表 */
               {
                   j=A->data[i],j;                /* 当前三元组的列号 */
                   k=cpot[j];                     /* 当前三元组在 B.data 中的位置 */
                   B->data[k],i=A->data[i],j;
                   B->data[k],j=A->data[i],i;
                   B->data[k],v=A->data[i],v;
                   cpot[j]++;
               }                                  /* for i */
           }                                      /* if (B->tu>0) */
   return B;                                      /* 返回的是转置矩阵的指针 */
}                                                 /* TransM2 */
```

算法 5.2 稀疏矩阵转置的改进算法

分析这个算法的时间复杂度，这个算法中有 4 个循环，分别执行 n、t、n−1、t 次，在每个循环中，每次迭代时间是一个常量，因此总的计算量是 O(n+t)。当然它所需要的存储空间比前一个算法多了两个向量。

2. 稀疏矩阵的乘积

已知稀疏矩阵 A($m_1 \times n_1$)和($m_2 \times n_2$)，求乘积 C($m_1 \times n_2$)。