第 5 章 chapter 5

数组和广义表

数组是人们已经非常熟悉的一种数据类型，几乎所有的计算机高级程序设计语言都支持数组这种数据类型，它的特点是数组中的数据元素都具有相同的数据类型，不同的元素通过下标来区别。本章主要讨论数组的逻辑结构、几种特殊矩阵的压缩存储以及广义表的存储与操作。

【本章学习要求】

掌握：数组的基本概念、数组的存储结构特点以及数组元素存储地址的计算。

掌握：特殊矩阵的压缩存储技术，如对称矩阵、三角矩阵、对角矩阵等的压缩存储。

掌握：稀疏矩阵的压缩存储方法——三元组表和十字链表。

掌握：广义表的基本定义和概念，理解广义表的递归性。

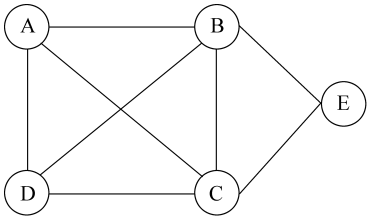
了解：广义表的存储特点和基本操作算法。

5.1 案 例 导 引

数组是程序设计中非常重要的存储形式，它可以将分散的数据按要求存放到一起，为数据的进一步处理提供方便，下面看几个案例。

【案例 5.1】 图的保存。

利用二维数组形式(常称为邻接矩阵，详见第 7 章)，可以很方便地将一个图的信息保存下来，如图 5.1(a)所示的无向图，用 1 表示有边相连，0 表示没有直接边相连，则对应的邻接矩阵如图 5.1(b)所示。



(a) 无向图

$$G=\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

(b) 对应的邻接矩阵

图 5.1 无向图及其对应的邻接矩阵

【案例 5.2】 图像的保存。

数字图像数据可以用矩阵来表示，因此可以采用矩阵理论和矩阵算法对数字图像进行分析和处理。由于数字图像可以表示为矩阵的形式，所以在计算机数字图像处理程序中，通常用二维数组来存放图像数据。

二维数组的行对应图像的高，二维数组的列对应图像的宽，二维数组的元素对应图像的像素，二维数组元素的值就是像素的灰度值。采用二维数组来存储数字图像，符合二维图像的行列特性，同时便于程序的寻址操作，使得计算机图像编程十分方便。

数字图像一般有二值图像、灰度图像、彩色图像之分。其中二值图像按名字来理解只有两个值，0 和 1，0 代表黑，1 代表白，或者说 0 表示背景，而 1 表示前景。其保存也相对简单，每个像素只需要 1b 就可以完整存储信息。灰度图像是二值图像的进化版本，是彩色图像的退化版，也就是灰度图像保存的信息没有彩色图像多，但比二值图像多，灰度图像只包含一个通道的信息，而彩色图像通常包含 3 个通道的信息。灰度图像是每个像素只有一个采样颜色的图像，这类图像通常显示为从最暗的黑色到最亮的白色的灰度，用于显示的灰度图像通常用每个采样像素 8b 的非线性尺度来保存，这样可以有 256 级灰度。彩色图像，每个像素通常是由红(R)、绿(G)、蓝(B)三个分量来表示的，每个分量值介于(0,255)。图 5.2(a)是原始人脸图像(32×32 像素)，图 5.2(b)和图 5.2(c)分别是对应的二值图像和灰度图像。图 5.3 所示是二值图像对应的矩阵，图 5.4 所示是灰度图像对应的矩阵。

(a) 原始图像　(b) 二值图像　(c) 灰度图像

图 5.2　原始图像及其转换

【案例 5.3】 图像卷积操作。

近年来，深度学习非常热门，而深度学习的网络结构要用到卷积操作。卷积的基本性质是将一个核与一个离散的单位脉冲进行卷积，在脉冲的位置上得到一个核的拷贝。在图像处理中，卷积操作是提取图像特征的一种方法，如图 5.5 所示。经过卷积操作的过滤器可以提取到图像的轮廓特征。

图 5.6 所示的卷积过程就是卷积核不停在原图上进行滑动(左上角开始)，每次滑动移动 1 格，然后再利用原图与卷积核上的数值进行计算得到缩略图矩阵(卷积特征)的数据。当卷积窗口滑动到某一位置时，窗口中的输入子数组与卷积核数组按元素相乘并求和，得到输出数组(卷积特征)中相应位置的元素。图 5.6 中的输出数组(卷积特征)的第一个元素(左上角)4 的计算过程为 1×1＋0×1＋1×1＋0×0＋1×1＋0×1＋1×0＋0×0＋1×1＝4。

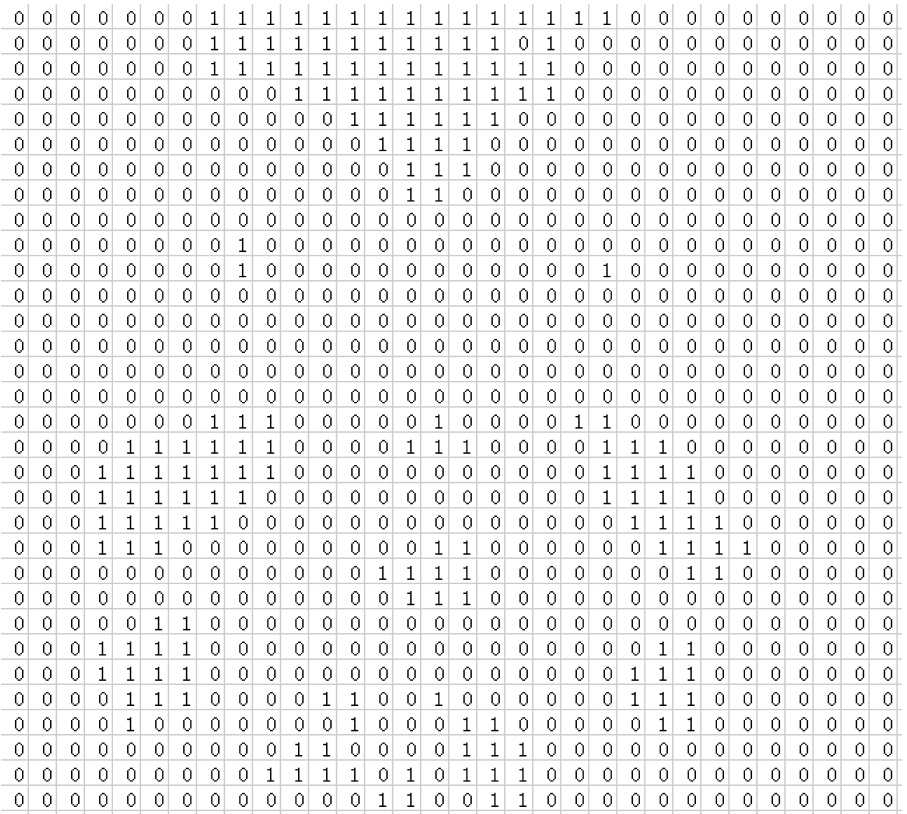


图 5.3 二值图像对应的矩阵

77	84	94	98	104	115	126	134	137	141	141	141	146	151	154	154	153	146	145	143	133	131	124	119	114	111	99	95	89	81	80	64
77	84	93	97	101	111	122	132	142	143	143	136	138	142	140	144	134	136	126	129	120	118	120	119	109	108	106	97	88	91	76	67
67	67	86	86	93	111	117	138	136	140	149	149	153	151	140	146	149	146	146	130	125	122	123	124	120	111	98	83	88	86	76	59
63	67	57	59	71	87	99	92	101	113	131	149	164	166	157	160	163	154	147	135	120	107	109	108	101	100	85	83	81	82	66	56
53	37	18	25	28	19	25	28	31	51	72	106	146	170	176	171	160	149	103	113	69	40	43	32	45	42	52	45	60	66	57	55
30	21	24	19	21	28	22	26	31	41	39	58	99	143	166	156	142	109	88	75	40	41	24	29	20	21	17	21	28	48	51	56
10	15	20	39	57	62	62	52	35	40	40	52	85	127	151	143	134	105	74	46	33	26	25	21	24	25	20	20	18	16	42	54
34	51	68	91	104	105	104	95	89	82	76	74	87	118	143	148	127	100	71	61	71	72	83	94	102	83	67	57	33	18	29	51
60	72	70	59	41	28	27	47	83	90	97	80	82	98	119	117	104	81	77	87	89	83	72	67	76	85	104	90	74	54	33	49
70	72	35	55	85	17	27	39	177	115	113	87	74	73	84	91	70	69	76	117	102	117	64	23	15	41	51	67	83	74	50	51
86	55	40	60	88	45	42	81	128	115	91	78	69	61	83	97	80	60	81	89	111	151	78	26	14	84	87	39	60	80	68	66
94	83	69	72	72	80	100	117	111	106	83	71	56	66	93	110	88	63	60	93	110	106	88	74	60	72	61	49	61	90	85	84
96	91	99	102	109	115	115	119	107	87	83	72	62	69	91	105	99	65	55	73	82	103	109	109	99	92	83	81	92	99	111	89
96	101	107	107	116	127	123	113	101	102	80	70	63	61	90	115	98	69	60	67	88	103	115	121	117	122	107	99	96	104	98	89
97	99	104	113	114	119	117	114	114	108	88	73	58	65	88	115	101	78	52	68	95	103	103	112	120	111	115	104	102	104	95	85
91	97	100	106	116	118	121	119	116	120	104	63	55	71	100	114	107	77	55	76	107	112	113	109	107	111	105	104	107	103	90	82
85	91	103	115	122	124	127	130	139	142	91	57	69	97	125	143	125	101	61	64	131	131	118	115	111	109	108	104	106	105	89	75
87	93	107	124	132	133	137	144	154	142	83	74	80	115	158	187	147	98	77	69	124	155	140	128	122	114	113	107	106	101	86	72
91	99	114	131	141	144	151	158	148	133	97	80	85	81	127	125	110	73	87	83	113	158	155	141	133	121	119	112	107	97	84	68
91	103	117	128	139	149	155	155	133	125	114	100	78	89	85	87	88	82	81	91	118	146	162	155	140	127	124	114	106	95	83	65
94	106	119	128	138	145	143	136	122	116	110	101	98	106	114	121	110	84	96	94	96	126	145	158	143	134	127	114	103	93	83	61
101	107	118	130	135	129	118	111	97	96	95	97	105	124	121	132	134	114	93	87	100	108	120	139	142	139	128	114	100	94	81	58
102	103	112	125	125	112	102	100	91	105	100	112	123	136	138	132	139	124	110	96	94	91	101	116	131	135	124	115	101	96	78	56
103	102	110	121	122	112	108	112	106	106	103	100	112	125	132	133	130	120	123	116	107	100	104	98	117	126	117	115	102	97	75	53
94	96	106	127	126	131	131	86	69	69	71	74	76	81	79	83	83	82	86	98	97	104	121	109	119	117	114	99	102	91	70	48
91	94	105	131	135	138	144	116	74	64	67	63	67	65	65	61	59	56	57	57	61	89	121	130	129	122	105	93	97	81	70	42
92	96	106	131	142	142	143	124	102	97	94	99	104	109	92	95	101	86	81	76	93	125	136	136	132	124	106	93	92	72	59	27
85	96	107	127	137	131	128	122	114	109	127	132	129	127	126	131	122	118	119	110	113	127	129	133	132	123	108	96	91	74	46	16
83	93	102	118	128	121	115	116	117	114	121	123	130	123	116	126	132	130	126	121	121	122	123	129	129	119	98	91	83	66	36	14
67	87	101	109	119	121	114	110	123	127	136	141	123	120	120	127	133	136	128	121	115	110	116	125	121	115	89	87	67	45	34	15
44	72	96	103	108	115	115	115	122	132	137	132	135	127	129	127	137	142	134	122	115	114	121	125	112	105	88	79	52	39	45	16
52	57	75	94	107	110	107	110	115	125	123	124	124	131	134	125	122	136	138	126	118	118	117	113	104	88	81	60	38	46	57	13

图 5.4 灰度图像对应的矩阵

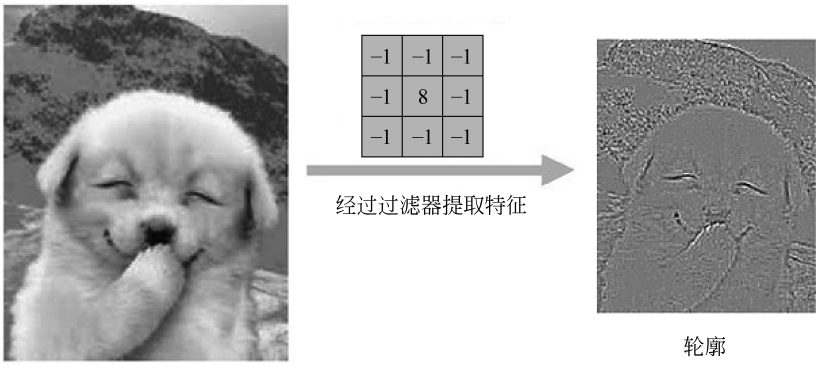


图 5.5　图像特征提取

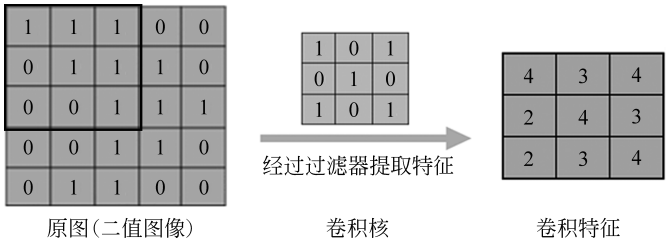


图 5.6　特征提取过程

【案例 5.4】　本科生导师制问题。

在高校的教学改革中，有很多学校实行了本科生导师制。一个班级的学生被分给几个老师，每个老师带领多个学生，如果老师还带研究生，那么研究生也可以直接负责本科生。

本科生导师制问题中的数据元素具有如下形式。

(1) 导师带研究生：(导师，((研究生 1，(本科生 1，…，本科生 m))，…))。

(2) 导师不带研究生：(导师，(本科生 1，…，本科生 m))。

例如：

(章老师，((李强，(王集山，武义军，刘秀))，张红卫，程昌义，姜和利))表示章老师指导 6 名本科生和一名叫李强的研究生，其中有 3 个本科生(王集山，武义军，刘秀)由研究生李强负责指导。

(李老师，(齐珊珊，黄凯，刘树发，陈海星))表示李老师指导 4 名本科生，没有指导研究生。

读者可以思考这两个数据信息表与前面所学的线性表有什么不同？如果不用这种信息表，利用线性表能把导师制中导师与本科生以及导师与研究生之间的逻辑关系表达出来吗？这种信息表就是本章要介绍的广义表，表中每个元素可以是类似线性表中的元素(原子项)，又可以是嵌套的线性表。这种数据结构在数据处理中有广泛的应用。

5.2 数 组

5.2.1 数组的定义

简单地讲，数组是由 $n(n\geqslant 1)$ 个相同类型数据元素 $a_0,a_1,\cdots,a_{n-1}$ 组成的有限序列，且该有限序列存储在一块地址连续的内存单元中，因而数组是顺序存储结构。

对于一个一维数组，一旦 a_0 的存储地址 $\mathrm{Loc}(a_0)$ 确定，每个数据元素的存储单元数 k 确定，则任一数据元素 a_i 的存储地址 $\mathrm{Loc}(a_i)$ 可由以下公式求出：

$$\mathrm{Loc}(a_i)=\mathrm{Loc}(a_0)+i\times k \quad (0\leqslant i<n) \tag{5.1}$$

对于二维数组，可将其转化为一维数组来考虑。例如图 5.7 为一个 m 行 n 列的二维数组，可以看成一个线性表

$$A=(b_0,b_1,\cdots,b_{n-1})$$

其中每个数据元素 $b_i=(a_{0i},a_{1i},\cdots,a_{m-1,i})(0\leqslant i<n)$。

$$A_{m\times n}=\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & a_{m-1,2} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

图 5.7 二维数组示例

显然，二维数组同样满足数组的定义。一个二维数组可以看作是每个数据元素都是相同类型的一维数组的一维数组。以此类推，一个三维数组可以看作是一个每个数据元素都是相同类型的二维数组的一维数组，等等。

因此，数组具有以下特点：

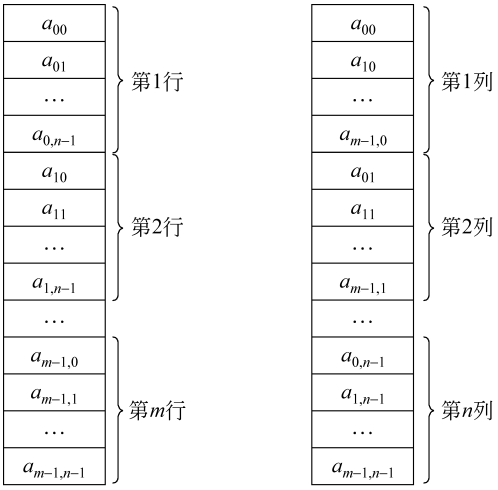
(1) 数组中的数据元素具有相同的数据类型。

(2) 数组是一种随机存储结构，可以根据给定的一组下标直接访问对应的数组元素。

(3) 一旦建立了数组，则数组中的数据元素个数和元素之间的关系就不再发生变化。

5.2.2 数组的内存映像

一维数组是用内存中一段连续的存储空间进行存储的，它的存储结构关系为式(5.1)。由于计算机的内存结构是一维的，因此用一维内存来表示多维数组，就必须按某种次序将数组元素排成一个序列，然后将这个线性序列存放在存储器中。对于二维数组，其存储可按行或列的次序用一组连续存储单元存放数组中的数组元素。如在 C、PASCAL、BASIC 等多数程序语言中，采用的是按行序为主序的存储结构，图 5.7 所示的二维数组可表示为图 5.8(a)，即先存储第 1 行，然后紧接着存储第 2 行，最后存储第 m 行。而在 FORTRAN 等少数程序语言中，采用的是以列序为主序的存储方式，图 5.7 所示的二维数组可表示为图 5.8(b)，即先存储第 1 列，然后紧接着存储第 2 列，最后存储第 n 列。



(a) 以行序为主序　　(b) 以列序为主序

图 5.8　二维数组的两种存储形式

在一个以行序为主序的计算机系统中，当二维数组第一个数据元素 a_{00} 的存储地址为 $\mathrm{Loc}(a_{00})$，假定每个数据元素占 k 个存储单元，则该二维数组中任一数据元素的存储地址可由下式确定：

$$\mathrm{Loc}(a_{ij}) = \mathrm{Loc}(a_{00}) + (i \times n + j) \times k \tag{5.2}$$

同理，可计算出更高维数组的数据元素存储位置的计算公式。

5.3　特殊矩阵的压缩存储

矩阵运算是许多科学和工程计算问题中常常遇到的问题，在用高级程序设计语言编制程序求解矩阵问题时，一般都是用二维数组来存储矩阵元素。在实际应用中，常常出现有许多值相同的元素或有许多零元素，且分布有一定规律的矩阵，一般称为特殊矩阵。为了节省存储空间，可以对这类特殊矩阵进行压缩存储，即多个相同的非零元素只分配一个存储空间；对零元素不分配空间。本节讨论这些特殊矩阵的压缩存储。

5.3.1　对称矩阵

在一个 n 阶方阵 $\boldsymbol{A}$ 中，若所有元素满足下述性质：

$$a_{ij} = a_{ji} \quad 0 \leqslant i, j \leqslant n-1$$

则称 $\boldsymbol{A}$ 为对称矩阵。

由于对称矩阵中的元素关于主对角线对称，因而只要存储矩阵中上三角或下三角中的元素，让每两个对称的元素共享一个存储空间，这样就可以将 n^2 个元素压缩存储到 $n(n+1)/2$ 个元素的空间中，能节约近一半的存储空间。假定按“行优先顺序”存储主对角线（包括对角线）以下的元素。

假设以一维数组 sa[$n(n+1)/2$]作为 n 阶对称矩阵 $\boldsymbol{A}$ 的存储结构，则 $\boldsymbol{A}$ 中任一元素 a_{ij} 和 sa[k]之间存在如下对应关系：

$$k=\begin{cases}\dfrac{i(i+1)}{2}+j & i\geqslant j\\ \dfrac{j(j+1)}{2}+i & i<j\end{cases} \tag{5.3}$$

由此，称一维数组 sa[$n(n+1)/2$]为 n 阶对称矩阵 $\boldsymbol{A}$ 的存储结构。其存储对应关系如图 5.9 所示。

k	0	1	2	3	…	$n(n-1)/2$	…	$n(n+1)/2-1$
sa[k]	a_{00}	a_{10}	a_{11}	a_{20}	…	$a_{n-1,0}$	…	$a_{n-1,n-1}$

图 5.9 对称矩阵的压缩存储

5.3.2 三角矩阵

以主对角线划分，三角矩阵有上三角和下三角两种。所谓 n 阶下(上)三角矩阵是指矩阵的上(下)三角(不包括主对角线)中的元素均为常数或零的 n 阶方阵。可以采用和对称矩阵类似的压缩存储方法来存储。三角矩阵中的重复元素 c 可共享一个存储空间，其余的元素正好有 $n(n+1)/2$ 个，可以用一维数组 sa[$n(n+1)/2+1$]作为 n 阶下(上)三角矩阵 $\boldsymbol{A}$ 的存储结构，其中常量 c 存放在数组的最后一个单元中，当 $\boldsymbol{A}$ 为下三角矩阵时，任一元素 a_{ij} 和 sa[k]之间存在式(5.4)的对应关系。

$$k=\begin{cases}\dfrac{i(i+1)}{2}+j & i\geqslant j\\ \dfrac{n(n+1)}{2} & i<j\end{cases} \tag{5.4}$$

5.3.3 稀疏矩阵

什么是稀疏矩阵？简单说，设矩阵 $\boldsymbol{A}_{mn}$ 中有 s 个非零元素，若 s 远远小于矩阵元素的总数(即 $s<<m\times n$)，则称 $\boldsymbol{A}$ 为稀疏矩阵。令 $e=s/(m\times n)$，称 e 为矩阵的稀疏因子。当用数组存储稀疏矩阵中元素时，仅有少部分的空间被利用，造成空间浪费。为节省存储空间，可以采用一种压缩的存储方法来表示稀疏矩阵的内容。由于非零元素的分布一般是没有规律的，因此在存储非零元素的同时，还必须同时记下元素所在的行和列的位置(row,col)。由于 a_{00} 位于矩阵的第 1 行第 1 列，因此，稀疏矩阵 $\boldsymbol{A}$ 中的任一非零元素 a_{ij} 可由一个三元组($i+1$,$j+1$,a_{ij})唯一确定。

1. 三元组表

假设非零元素的三元组是以按行优先的顺序排列，一个稀疏矩阵就可转换成用一个对应的线性顺序表来表示，其中每个元素由一个上述的三元组构成，该线性表称为三元组表，记为(i,j,v)。其类型说明如下：

```
#define MAXSIZE  1000
typedef struct {
      int  i, j;                    /*非零元素的行、列号*/
      DataType  v;                  /*非零元素的值*/
}triple;
typedef struct {
       triple  data[MAXSIZE];       /*非零元素的三元组表*/
       int  m,n,t;                  /*稀疏矩阵的行数、列数和非零元素的个数*/
}tripletable;
```

下面以矩阵的转置为例，说明在这种压缩存储结构上如何实现矩阵的运算。

一个 $m\times n$ 的矩阵 $\boldsymbol{A}$，它的转置 $\boldsymbol{B}$ 是一个 $n\times m$ 的矩阵，且 $a[i][j]=b[j][i]$，即 $\boldsymbol{A}$ 的行是 $\boldsymbol{B}$ 的列，$\boldsymbol{A}$ 的列是 $\boldsymbol{B}$ 的行。例如图5.10(a)中稀疏矩阵 $\boldsymbol{A}$ 及其转置矩阵 $\boldsymbol{B}$ 可用图5.10(b)所示的三元组表示。

$$\boldsymbol{A}_{4\times5}=\begin{bmatrix}0&7&0&4&0\\3&0&0&0&0\\0&0&0&0&1\\0&6&0&0&0\end{bmatrix}\quad \boldsymbol{B}_{5\times4}=\begin{bmatrix}0&3&0&0\\7&0&0&6\\0&0&0&0\\4&0&0&0\\0&0&1&0\end{bmatrix}$$

(a) 稀疏矩阵$\boldsymbol{A}$及其转置矩阵$\boldsymbol{B}$

i	j	v
1	2	7
1	4	4
2	1	3
3	5	1
4	2	6

a.data

i	j	v
1	2	3
2	1	7
2	4	6
4	1	4
5	3	1

b.data

(b) 稀疏矩阵$\boldsymbol{A}$和$\boldsymbol{B}$的三元组表示

图 5.10　稀疏矩阵的三元组表示

将 $\boldsymbol{A}$ 转置为 $\boldsymbol{B}$，就是将 $\boldsymbol{A}$ 的三元组表 a.data 置换为 $\boldsymbol{B}$ 的三元组表 b.data，如果只是简单地交换 a.data 中 i 和 j 的内容，那么得到的 b.data 将是一个按列优先顺序存储的稀疏矩阵 $\boldsymbol{B}$，要得到按行优先顺序存储的 b.data，就必须重新排列三元组的顺序。有如下两种方法来进行处理。

(1) 第一种方法(跳着找，顺着存)。

由于 $\boldsymbol{A}$ 的列是 $\boldsymbol{B}$ 的行，因此按 a.data 的列序转置，所得到的转置矩阵 $\boldsymbol{B}$ 的三元组表 b.data 必定是按行优先存放的。按这种方法设计的算法，其基本思想是对 $\boldsymbol{A}$ 中的每一列 col(1≤col≤n)，通过从头至尾扫描三元表 a.data，找出所有列号等于 col 的那些三元组，将它们的行号和列号互换后依次放入 b.data 中，即可得到 $\boldsymbol{B}$ 的按行优先的压缩存储表示。

具体算法如下。

【算法 5.1】

```
tripletable transmatrix(tripletable a)
{  /*将稀疏矩阵 a 转置，结果通过函数名返回*/
   tripletable b;
    int p,q,col;
```

```
    b.m=a.n;                    /*矩阵b的行数等于矩阵a的列数*/
    b.n=a.m;                    /*矩阵b的列数等于矩阵a的行数*/
    b.t=a.t;                    /*矩阵b的非零元素数等于矩阵a的非零元素数*/
    if(b.t)                     /*把a中每一个非零元素转换到b中相应位置*/
    {   q=0;
        for(col=1;col<=a.n;col++)           /*按列号扫描*/
        for(p=0;p<a.t;p++)                  /*在数据中找列号为col的三元组*/
            if(a.data[p].j==col)
            {   b.data[q].i=col;                    /*新三元组的行号/*
                b.data[q].j=a.data[p].i;            /*新三元组的列号/*
                b.data[q].v=a.data[p].v;            /*新三元组的值/*
                q++;
            }
    }
    return(b);
}
```

上述算法主要工作是在 p 和 col 的两重循环中完成的，故算法的时间复杂度为 $O(a.n \times a.t)$，即与矩阵的列数和非零元的个数的乘积成正比。而一般传统矩阵的转置算法为

```
for(col=0;col<n;++col)
    for(row=0;row<m;++row)
        b[col][row]=a[row][col];
```

其时间复杂度为 $O(n \times m)$。当非零元素的个数 t 和 $m \times n$ 同数量级时，算法 transmatrix 的时间复杂度为 $O(m \times n^2)$，因此上述稀疏矩阵转置算法的时间大于非压缩存储的矩阵转置的时间。三元组顺序表虽然节省了存储空间，但时间复杂度比一般矩阵转置的算法大，同时还有可能增加算法的难度。因此，此算法仅适用于 $t << m \times n$ 的情况。

(2) 第二种方法(顺着找，跳着存)。

第一种方法中重复比较的次数比较多，为了节省时间，需要确定矩阵 ***A*** 中每一列第一个非零元素在 ***B*** 中应存储的位置，为了确定这个位置，在转置前应求得矩阵 ***A*** 中的每列非零元素的个数。其算法思想为对 ***A*** 扫描一次，按 ***A*** 第二列提供的列号一次确定位置装入 ***B*** 的三元组中。具体实施方法如下：一遍扫描先确定三元组的位置关系，二次扫描由位置关系装入三元组。可见，位置关系是此种算法的关键。

为此需要附设两个一维数组 num 和 pot，num[j]表示矩阵 ***A*** 中的第 j 列非零元素个数，pot[j]表示 ***A*** 矩阵中第 j 列下一个非零元素在 ***B*** 中应存放的位置(初值为该列第一个非零元素在 ***B*** 中应存放的位置)。显然有：

$$\text{pot}[1] = 0$$

$$\text{pot}[j] = \text{pot}[j-1] + \text{num}[j-1] \quad 2 \leqslant j \leqslant a.n$$

例如，矩阵 ***A*** 的 num 和 pot 的数组元素值如表 5.1 所示。

表 5.1　矩阵 A 的向量 num 和 pot 的值

j	1	2	3	4	5
num[j]	1	2	0	1	1
pot[j]	0	1	3	3	4

快速转置算法如下。

【算法 5.2】

```
tripletable fasttranstri(tripletable a)
{  /*将稀疏矩阵 a 做快速转置,结果通过函数名返回*/
    tripletable b;
      int p,q,col,k;
      int num[a.n+1], pot[a.n+1];          /*建立辅助数组*/
      b.m=a.n; b.n=a.m; b.t=a.t;
      if(b.t)
       {   for(col=1;col<=a.n;++col)     /*对数组 num 初始化*/
           num[col]=0;
           for(k=0;k<a.t;++k)             /*计算 a 中每一列含非零元素的个数*/
             ++num[a.data[k].j];
             pot[1]=0;          /*计算 a 中第 col 列中第一个非零元素在 b 中的序号*/
           for(col=2;col<=a.n;++col)
             pot[col]=pot[col-1]+num[col-1];
           for(p=0;p<a.t;++p)/*把 a 中每一个非零元素插入 b 中的相应位置*/
         {    col=a.data[p].j;
                q=pot[col];
              b.data[q].i=a.data[p].j;
                b.data[q].j=a.data[p].i;
              b.data[q].v=a.data[p].v;
                ++pot[col];
         }
       }
       return(b);
}
```

该算法虽然多用了两个辅助向量空间，但它的时间复杂度为 $O(a.n+a.t)$，比第一种方法要好。

2. 十字链表存储

三元组表是用顺序方法来存储稀疏矩阵中的非零元素，当非零元素的位置或个数经常变化时，三元组表就不适合做稀疏矩阵的存储结构。例如，两矩阵做加操作时，会改变非零元素的个数，如用三元组表表示矩阵时，元素的插入和删除会导致大量的结点移动。此时，采用链式存储结构更为合适。