第 1 章　图的基本概念及图的存储

图是一种重要的数学模型和数据结构，通常用来描述某些事物之间的某种特定关系，图可以对自然科学和社会科学中许多领域的问题进行恰当的描述或建模，因而在很多领域都有广泛的应用。本章介绍图论的一些基本概念及图的两种存储表示方法：邻接矩阵和邻接表。

1.1　基 本 概 念

图论知识的一个特点就是概念特别多，为了使读者尽快进入到图论算法层次，而不是停留在概念层面上，本节只介绍一些基本的概念，其他概念(如遍历、拓扑排序、网络流等)在其他章节里需要用的时候再补充介绍。

1.1.1　有向图与无向图

图(Graph)是由顶点集合和顶点间的二元关系集合(即边的集合或弧的集合)组成的数据结构，通常可以用 $G(V, E)$来表示。其中**顶点集合**(Vertext Set)和**边的集合**(Edge Set)分别用 $V(G)$和 $E(G)$表示。$V(G)$中的元素称为**顶点**(Vertex)，用 u、v 等符号表示；顶点个数称为图的**阶**(Order)，通常用 n 表示。$E(G)$中的元素称为**边**(Edge)，用 e 等符号表示；边的个数称为图的**边数**(Size)，通常用 m 表示。

例如，图 1.1(a)所示的图可以表示为 $G_1(V, E)$。其中，顶点集合 $V(G_1) = \{ 1, 2, 3, 4, 5, 6 \}$，集合中的元素为顶点(用序号代表，在其他图中，顶点集合中的元素也可以是其他标识顶点的符号，如字母 A、B、C 等)；边的集合为：

$$E(G_1) = \{ (1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (2, 6), (3, 4), (3, 5), (4, 5) \}。$$

在上述边的集合中，每个元素(u, v)为一对顶点构成的无序对(用圆括号括起来)，表示与顶点 u 和 v **相关联**的一条**无向边**(Undirected Edge)，这条边没有特定的方向，因此(u, v)与(v, u)是同一条的边。如果图中所有的边都没有方向性，这种图称为**无向图**(Undirected Graph)。

图 1.1(b)所示的图可以表示为 $G_2(V, E)$，其中顶点集合 $V(G_2) = \{ 1, 2, 3, 4, 5, 6, 7 \}$，集合中的元素也为顶点的序号；边的集合为：

$$E(G_2) = \{ <1, 2>, <2, 3>, <2, 5>, <2, 6>, <3, 5>, <4, 3>, <5, 2>, <5, 4>, <6, 7> \}。$$

在上述边的集合中，每个元素$<u, v>$为一对顶点构成的有序对(用尖括号括起来)，表示从顶点 u 到顶点 v 的**有向边**(Directed Edge)，其中 u 是这条有向边的**起始顶点**(Start Vertex)，简称**起点**，v 是这条有向边的**终止顶点**(End Vertex)，简称**终点**，这条边有特定的方向，由 u 指向 v，因此$<u, v>$与$<v, u>$是两条不同的边。例如，在图 1.1(b)有向图 G_2 中，$<2, 5>$和$<5, 2>$是两条不同的边。如果图中所有的边都是有方向性的，这种图称为**有向图**(Directed Graph)。

有向图中的边也可以称为**弧**(Arc)。有向图也可以表示成 $D(V, A)$，其中 A 为弧的集合。

有向图的**基图**(Ground Graph)：忽略有向图所有边的方向，得到的无向图称为该有向图的基图。例如，图 1.1(c)所示为图 1.1(b)中有向图 G_2 的基图。

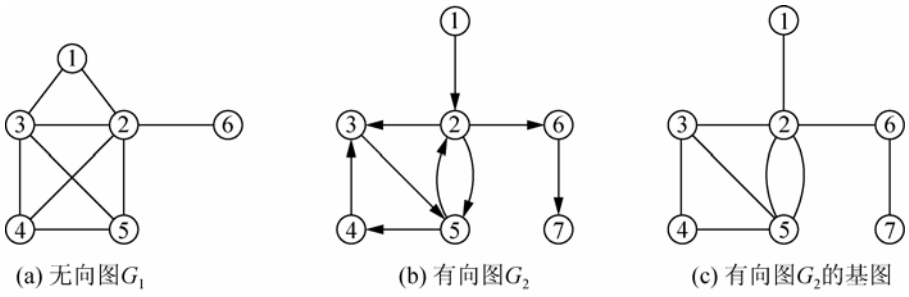


图 1.1　无向图与有向图

说明：如果一个图中某些边具有方向性，而其他边没有方向性，这种图可以称为**混合图**(Mixed Graph)；如无特殊说明，本书的讨论仅限于无向图和有向图，不包括混合图。

1.1.2　完全图、稀疏图、稠密图

许多图论算法的复杂度都与图中顶点个数 n 或边的数目 m 有关，甚至 m 与 $n\times(n-1)$之间的相对关系也会影响图论算法的选择。下面介绍几个与顶点个数、边的数目相关的概念。

完全图(Complete Graph)：如果无向图中任何一对顶点之间都有一条边，这种无向图称为完全图。在完全图中，阶数和边数存在关系式：$m = n\times(n-1)/2$。例如，图 1.2(a)所示的无向图就是完全图。阶为 n 的完全图用 K_n 表示。例如，图 1.2(a)所示的完全图为 4 阶完全图 K_4。

有向完全图(Directed Complete Graph)：如果有向图中任何一对顶点 u 和 v，都存在$<u, v>$和$<v, u>$两条有向边，这种有向图称为有向完全图。在有向完全图中，阶数和边数存在关系式：$m = n\times(n-1)$。例如，图 1.2(b)所示的有向图就是有向完全图。

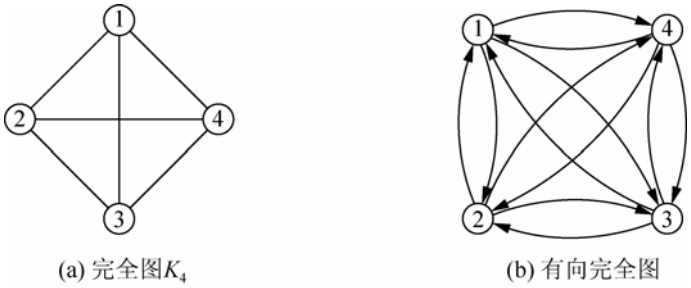


图 1.2　完全图与有向完全图

稀疏图(Sparse Graph)：边或弧的数目相对较少(远小于 $n\times(n-1)$)的图称为稀疏图。有的文献认为，边或弧的数目 $m<n\log(n)$的无向图或有向图，称为稀疏图。例如，图 1.3(a)所示的无向图可以称为稀疏图。

稠密图(Dense Graph)：边或弧的数目相对较多的图(接近于完全图或有向完全图)称为稠密图。例如，图 1.3(b)所示的无向图可以称为稠密图。

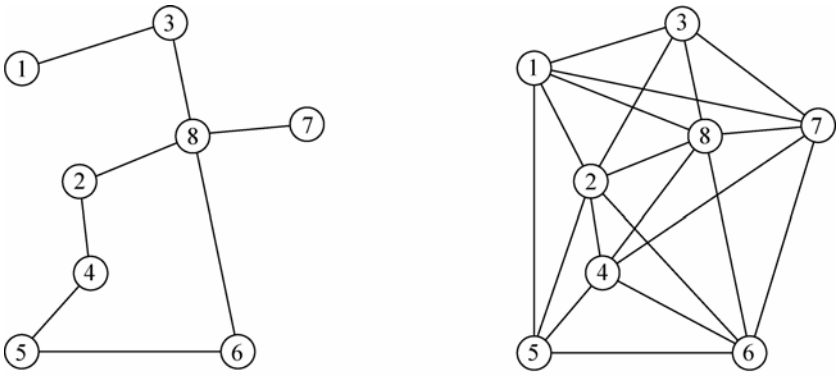


图 1.3　稀疏图与稠密图

平凡图(Trivial Graph)：只有一个顶点的图，即阶 $n=1$ 的图称为平凡图。相反，阶 $n>1$ 的图称为**非平凡图**(Nontrivial Graph)。

零图(Null Graph)：边的集合 $E(G)$为空的图，称为零图。

1.1.3　顶点与顶点、顶点与边的关系

在无向图和有向图中，顶点与顶点之间的关系，以及顶点与边的关系是通过“**邻接**(Adjacency)”这个概念来表示的。

在无向图 $G(V, E)$中，如果(u, v)是 $E(G)$中的元素，即(u, v)是图中的一条无向边，则称顶点 u 与顶点 v 互为**邻接顶点**(Adjacent Vertex)，边(u, v)**依附于**(Attach To)顶点 u 和 v，或称边(u, v)与顶点 u 和 v **相关联**(Incident)。此外，称有一个共同顶点的两条不同边为**邻接边**(Adjacent Edge)。

例如，在图 1.1(a)所示的无向图 G_1 中，与顶点 2 相邻接的顶点有 1, 3, 4, 5, 6，而依附于顶点 2 的边有(2, 1), (2, 3), (2, 4), (2, 5)和(2, 6)。

在有向图 $G(V, E)$中，如果$<u, v>$是 $E(G)$中的元素，即$<u, v>$是图中的一条有向边，则称顶点 u **邻接到**(Adjacent To)顶点 v，顶点 v **邻接自**(Adjacent From)顶点 u，边$<u, v>$与顶点 u 和 v **相关联**。

例如，在图 1.1(b)所示的有向图 G_2 中，顶点 2 分别邻接到顶点 3, 5, 6，邻接自顶点 1 和 5；有向边<2, 6>的顶点 2 邻接到顶点 6，顶点 6 邻接自顶点 2；顶点 2 分别与边<2, 3>, <2, 5>, <2, 6>, <1, 2>和<5, 2>相关联等。

1.1.4　顶点的度数及度序列

1. 与顶点度数有关的概念

顶点的度数(Degree)：一个顶点 u 的度数是与它相关联的边的数目，记作 deg(u)。例如，在图 1.1(a)所示的无向图 G_1 中，顶点 2 的度数为 5，顶点 5 的度数为 3 等。

在有向图中，顶点的度数等于该顶点的出度与入度之和。其中，顶点 u 的**出度**(Outdegree)是以 u 为起始顶点的有向边(即从顶点 u 出发的有向边)的数目，记作 od(u)；顶点 u 的**入度**(Indegree)是以 u 为终点的有向边(即进入到顶点 u 的有向边)的数目，记作 id(u)。顶点 u 的

度数：deg(u) = od(u) + id(u)。例如，在图 1.1(b)所示的有向图 G_2 中，顶点 2 的出度为 3，入度为 2，度数为：3 + 2 = 5。

在无向图和有向图中，边数 m 和所有顶点度数总和都存在如下关系。

定理 1.1 在无向图和有向图中，所有顶点度数总和，等于边数的两倍，即：

$$m = \frac{1}{2}\left\{\sum_{i=1}^{n}\deg(u_i)\right\} \tag{1-1}$$

这是因为，不管是有向图还是无向图，在统计所有顶点度数总和时，每条边都统计了两次。

偶点与奇点：为方便起见，把度数为偶数的顶点称为**偶点**(Even Vertex)，把度数为奇数的顶点称为**奇点**(Odd Vertex)。

推论 1.1 每个图都有偶数个奇点。

孤立顶点(Isolated Vertex)：度数为 0 的顶点，称为孤立顶点。孤立顶点不与其他任何顶点邻接。

叶(Leaf)：度数为 1 的顶点，称为叶顶点，也称**叶顶点**(Leaf Vertex)或**端点**(End Vertex)。其他顶点称为**非叶顶点**。

图 G 的**最小度**(Minimum Degree)：图 G 所有顶点的最小的度数，记为 $\delta(G)$。

图 G 的**最大度**(Maximum Degree)：图 G 所有顶点的最大的度数，记为 $\Delta(G)$。

例如，图 1.1(a)所示的无向图没有孤立顶点，顶点 6 为叶顶点；$\delta(G) = 1$，$\Delta(G) = 4$；等。

2. 度序列与 Havel-Hakimi 定理

度序列(Degree Sequence)：若把图 G 所有顶点的度数排成一个序列 s，则称 s 为图 G 的度序列。例如，图 1.1(a)所示的无向图 G_1 的度序列为

s: 2, 5, 4, 3, 3, 1；或 s': 1, 2, 3, 3, 4, 5；或 s'': 5, 4, 3, 3, 2, 1。

其中序列 s 是按顶点序号排序的，序列 s'是按度数非减顺序排列的，序列 s''是按度数非增顺序排列的。给定一个图，确定它的度序列很简单，但是其逆问题并不容易，即给定一个由非负整数组成的有限序列 s，判断 s 是否是某个图的度序列。

序列是**可图的**(Graphic)：一个非负整数组成的有限序列如果是某个无向图的度序列，则称该序列是可图的。判定一个序列是否是可图的，有以下 Havel-Hakimi 定理。

定理 1.2(Havel-Hakimi 定理) 由非负整数组成的非增序列 s：$d_1, d_2, \cdots, d_n (n \geqslant 2, d_1 \geqslant 1)$ 是可图的，当且仅当序列

$$s_1: d_2 - 1, d_3 - 1, \cdots, d_{d1+1} - 1, d_{d1+2}, \cdots, d_n$$

是可图的。序列 s_1 中有 $n-1$ 个非负整数，s 序列中 d_1 后的前 d_1 个度数(即 $d_2 \sim d_{d1+1}$)减 1 后构成 s_1 中的前 d_1 个数。

例如，判断序列 s: 7, 7, 4, 3, 3, 3, 2, 1 是否是可图的。删除序列 s 的首项 7，对其后的 7 项每项减 1，得到：6, 3, 2, 2, 2, 1, 0。继续删除序列的首项 6，对其后的 6 项每项减 1，得到：2, 1, 1, 1, 0, −1，到这一步出现了负数。由于图中不可能存在负度数的顶点，因此该序列不是可图的。

再举一个例子，判断序列 s: 5, 4, 3, 3, 2, 2, 2, 1, 1, 1 是否是可图的。删除序列 s 的首项 5，对其后的 5 项每项减 1，得到：3, 2, 2, 1, 1, 2, 1, 1, 1，重新排序后为：3, 2, 2, 2, 1, 1, 1, 1, 1。继续删除序列的首项 3，对其后的 3 项每项减 1，得到：1, 1, 1, 1, 1, 1, 1, 1。如此再陆续得

到序列：1, 1, 1, 1, 1, 1, 0；1, 1, 1, 1, 0, 0；1, 1, 0, 0, 0；0, 0, 0, 0。由此可判定该序列是可图的。

Havel-Hakimi 定理实际上给出了根据一个序列 s 构造图(或判定 s 不是可图的)的方法：把序列 s 按照非增顺序排序以后，其顺序为 $d_1, d_2, \cdots, d_n$；度数最大的顶点设为 v_1，将它与度数次大的前 d_1 个顶点之间连边，然后这个顶点就可以不管了，即在序列中删除首项 d_1，并把后面的 d_1 个度数减 1；再把剩下的序列重新按非增顺序排序，按照上述过程连边；……；直到建出完整的图，或出现负度数等明显不合理的情况为止。

例如，对序列 s: 3, 3, 2, 2, 1, 1 构造图，设度数从大到小的 6 个顶点为 v_1～v_6。首先 v_1 与 v_2、v_3、v_4 连一条边，如图 1.4(a)所示；剩下的序列为 2, 1, 1, 1, 1。如果后面 4 个 1 对应顶点 v_3、v_4、v_5、v_6，则应该在 v_2 与 v_3、v_2 与 v_4 之间连边，最后在 v_5 与 v_6 之间连边，如图 1.4(b)所示。如果后面 4 个 1 对应顶点 v_5、v_6、v_3、v_4，则应该在 v_2 与 v_5、v_2 与 v_6 之间连边，最后在 v_3 与 v_4 之间连边，如图 1.4(c)所示。可见，由同一个可图的序列构造出来的图不一定是唯一的。

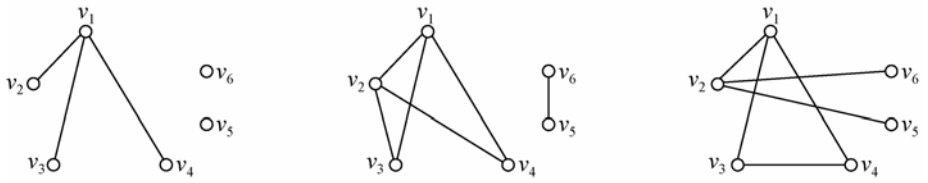


图 1.4　根据度序列构造图

利用 Havel-Hakimi 定理判断一个序列是否是可图的，其程序实现详见例 1.2。

1.1.5　二部图与完全二部图

二部图(Bipartite Graph)：设无向图为 $G(V, E)$，它的顶点集合 V 包含两个没有公共元素的子集：$X = \{ x_1, x_2, \cdots, x_s \}$和 $Y = \{ y_1, y_2, \cdots, y_t \}$，元素个数分别为 s 和 t；并且 x_i 与 x_j 之间$(1 \leqslant i, j \leqslant s)$、$y_l$ 与 y_r 之间$(1 \leqslant l, r \leqslant t)$没有边连接，则称 G 为二部图，有的文献也称为**二分图**。

例如，图 1.5(a)所示的无向图就是一个二部图。

完全二部图(Complete Bipartite Graph)：在二部图 G 中，如果顶点集合 X 中每个顶点 x_i 与顶点集合 Y 中每个顶点 y_l 都有边相连，则称 G 为完全二部图，记为 $K_{s,t}$，s 和 t 分别为集合 X 和集合 Y 中的顶点个数。在完全二部图 $K_{s,t}$ 中一共有 $s \times t$ 条边。

例如，如图 1.5(b)所示的 $K_{2,3}$ 和图 1.5(c)所示的 $K_{3,3}$ 都是完全二部图。

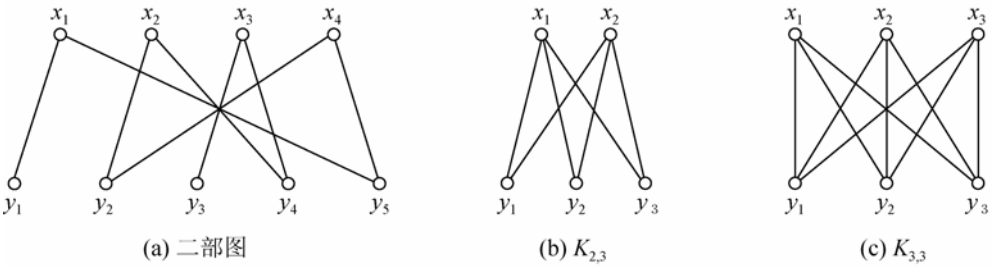
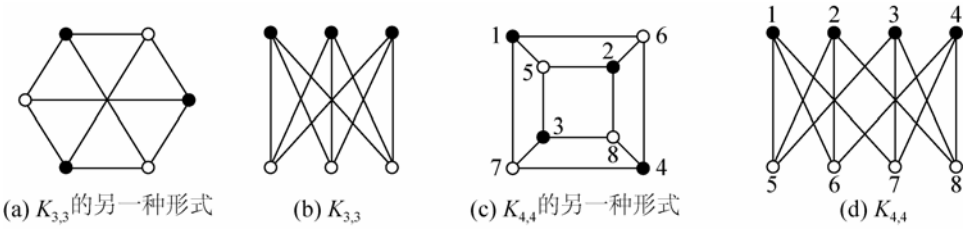


图 1.5　二部图与完全二部图

观察图 1.6(a)和图 1.6(c)所示两个图，表面上看起来这两个图都不是二部图。但仔细观察，发现图 1.6(a)中 3 个黑色顶点互不相邻，3 个白色顶点也互不相邻，每个黑色顶点都与 3 个白色顶点相邻，因此图 1.6(a)实际上也是 $K_{3,3}$，如图 1.6(b)所示。同样，图 1.6(c)中 4 个黑色顶点互不相邻，4 个白色顶点也互不相邻，对这 8 个顶点进行编号后，重新画成图 1.6(d)所示的图，发现图 1.6(c)实际上也是一个二部图。



(a) $K_{3,3}$的另一种形式　(b) $K_{3,3}$　(c) $K_{4,4}$的另一种形式　(d) $K_{4,4}$

图 1.6　二部图的判定

一个图是否为二部图，可由下面的定理判别。

定理 1.3　一个无向图 G 是二部图当且仅当 G 中无奇数长度的回路(回路及路径长度的概念请参考 1.1.8 节)。

由定理 1.3 可判定图 1.6(a)和图 1.6(c)都是二部图。

1.1.6　图的同构

从图 1.6 可知，有些图之间看起来差别很大，比如图 1.6(a)和图 1.6(b)、图 1.6(c)和图 1.6(d)，但经过改画后，它们实际上是同一个图。

又如，图 1.7(a)和图 1.7(b)两个图表面上看差别也很大，但是对图 1.7(b)按照图中的顺序给每个顶点编号后发现，这两个图实际上也是同一个图。

(a) 第1种编号方式　(b) 第2种编号方式

图 1.7　图的同构

图的**同构**(Isomorphism)：设有两个图 G_1 和 G_2，如果这两个图区别仅在于图的画法与(或)顶点的标号方式，则称它们是同构的。意思就是说这两个图是同一个图。关于同构的严格定义，请参考其他教材。

1.1.7　子图与生成树

设有两个图 $G(V, E)$和 $G'(V', E')$，如果 $V'\subseteq V$，且 $E'\subseteq E$，则称图 G'是图 G 的**子图**(Subgraph)。例如，图 1.8(a)、图 1.8(b)所示的无向图都是图 1.1(a)所示的无向图 G_1 的子图，而图 1.8(c)、图 1.8(d)所示的有向图都是图 1.1(b)所示的有向图 G_2 的子图。

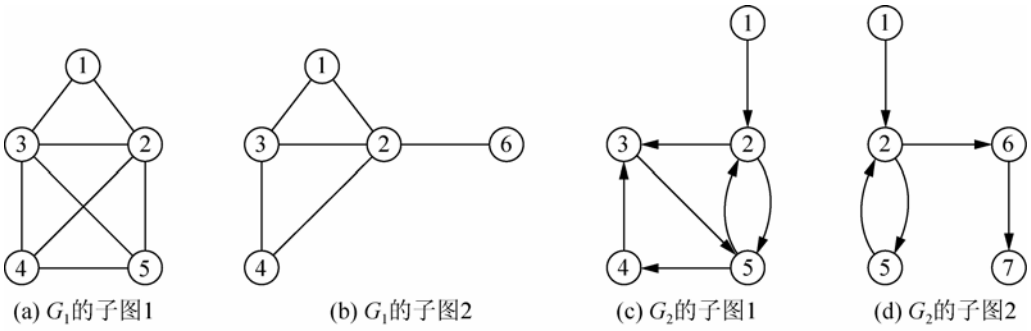


图 1.8　子图

生成树(Spanning Tree)：一个无向连通图(连通图的概念详见 1.1.9 节)的生成树是它的包含所有顶点的极小连通子图，这里所谓的极小就是边的数目极小。如果图中有 n 个顶点，则生成树有 $n-1$ 条边。一个无向连通图可能有多个生成树。

例如，图 1.1(a)所示的无向图 G_1 的一个生成树如图 1.9(a)所示。为了更形象地表示这个生成树，在图 1.9(b)中把它画成了以顶点 1 为根结点的树，在图 1.9(c)中把它画成了以顶点 3 为根结点的树。

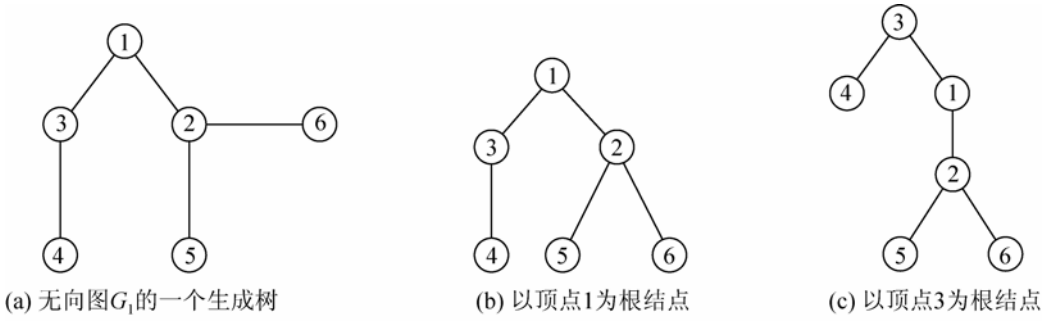


图 1.9　无向图的生成树

观察图 1.10，其中图 1.10(b)和图 1.10(c)都是图 1.10(a)的子图，这两个子图的顶点集相同，为 $V'=\{2,3,4,5\}$，但边集不相同。图 1.10(b)保留了原图中 V'内各顶点间的边，而在图 1.10(c)中，原图的边(3, 5)和(3, 2)被去掉了。因此有必要进一步讨论子图。

设图 $G'(V', E')$是图 $G(V, E)$的子图，且对于 V'中的任意两个顶点 u 和 v，只要(u, v)是 G 中的边，则一定是 G'中的边，此时称图 G'为**由顶点集合 V'诱导的 G 的子图**(Subgraph of G Induced By V')，简称为**顶点诱导子图**(Vertex-Induced Subgraph)，记为 $G[V']$。根据定义，在图 1.10 中，图 1.10(b)是由 $V'=\{2,3,4,5\}$诱导的子图，图 1.10(c)和图 1.10(d)都不是顶点诱导子图。

类似地，对于图 G 的一个非空的边集合 E'，**由边集合 E'诱导的 G 的子图**是以 E'作为边集，以至少与 E'中一条边关联的那些顶点构成顶点集 V'，这个子图 $G'(V', E')$称为是 G 的一个**边诱导子图**(Edge-Induced Subgraph)，记为 $G[E']$。根据定义，在图 1.10 中，图 1.10(b)、图 1.10(c)和图 1.10(d)都是边诱导子图。

说明：由于边必须依附于顶点而存在，所以对于“某条边属于子图，但该边某个顶点不属于子图”的情形，是没有意义的，本书对这种子图不作进一步的讨论。

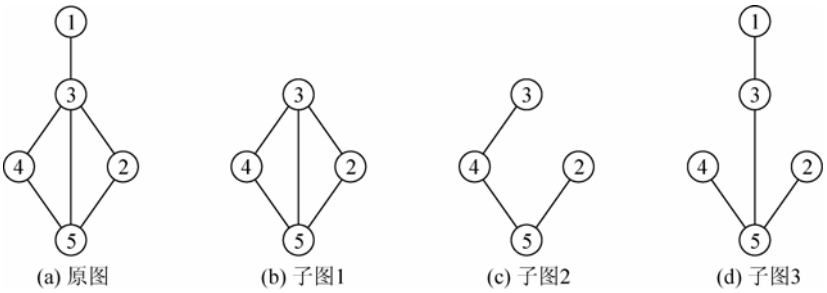


图 1.10　子图与诱导子图

1.1.8　路径

路径是图论中一个很重要的概念。在图 $G(V, E)$中，若从顶点 v_i 出发，沿着一些边经过一些顶点 $v_{p1}, v_{p2}, \cdots, v_{pm}$，到达顶点 v_j，则称顶点序列$(v_i, v_{p1}, v_{p2}, \cdots, v_{pm}, v_j)$为从顶点 v_i 到顶点 v_j 的一条**路径**(Path)，或称为**通路**，其中$(v_i, v_{p1}), (v_{p1}, v_{p2}), \cdots, (v_{pm}, v_j)$为图 G 中的边。如果 G 是有向图，则$<v_i, v_{p1}>, <v_{p1}, v_{p2}>, \cdots, <v_{pm}, v_j>$为图 G 中的有向边。

路径长度(Length)：路径中边的数目通常称为路径的长度。

例如，在图 1.1(a)所示的无向图 G_1 中，顶点序列(1, 2, 5, 4)是从顶点 1 到顶点 4 的路径，路径长度为 3，其中(1,2), (2,5), (5,4)都是图 G_1 中的边；另外，顶点序列(1, 3, 4)也是从顶点 1 到顶点 4 的路径，路径长度为 2。

在图 1.1(b)所示的有向图 G_2 中，顶点序列(3, 5, 2, 6)是从顶点 3 到顶点 6 的路径，路径长度为 3，其中<3,5>, <5,2>, <2,6>都是图 G_2 中的有向边；而从顶点 7 到顶点 1 没有路径。

简单路径(Simple Path)：若路径上各顶点 $v_i, v_{p1}, v_{p2}, \cdots, v_{pm}, v_j$ 均互相不重复，则这样的路径称为简单路径。例如，在图 1.1(a)所示的无向图 G_1 中，路径(1, 2, 5, 4)就是一条简单路径。

回路(Circuit)：若路径上第一个顶点 v_i 与最后一个顶点 v_j 重合，则称这样的路径为回路。例如，在图 1.1 中，图 G_1 中的路径(2, 3, 4, 5, 2)和图 G_2 中的路径(5, 4, 3, 5)都是回路。回路也称为**环**(Loop)。

简单回路(Simple Circuit)：除第一个和最后一个顶点外，没有顶点重复的回路称为简单回路。简单回路也称为**圈**(Cycle)。长度为奇数的圈称为**奇圈**(Odd Cycle)，长度为偶数的圈称为**偶圈**(Even Cycle)。

1.1.9　连通性

连通性也是图论中一个很重要的概念。在无向图中，若从顶点 u 到 v 有路径，则称顶点 u 和 v 是**连通**(Connected)的。如果无向图中任意一对顶点都是连通的，则称此图是**连通图**(Connected Graph)；相反，如果一个无向图不是连通图，则称为**非连通图**(Disconnected Graph)。

如果一个无向图不是连通的，则其极大连通子图称为**连通分量**(Connected Component)，这里所谓的极大是指子图中包含的顶点个数极大。

例如，图 1.1(a)所示的无向图 G_1 就是一个连通图。在图 G_1 中，如果去掉边(2, 6)，则剩下的图就是非连通的，且包含两个连通分量，一个是由顶点 1、2、3、4、5 组成的连通

分量，另一个是由顶点 6 构成的连通分量。

又如，图 1.11 所示的无向图也是非连通图。其中顶点 1、2、3 和 5 构成一个连通分量，顶点 4、6、7 和 8 构成另一个连通分量。

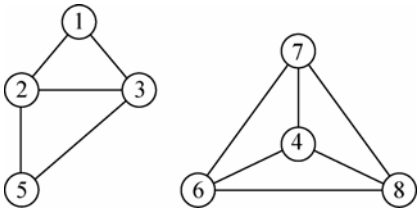


图 1.11　非连通图

在有向图中，若对每一对顶点 u 和 v，既存在从 u 到 v 的路径，也存在从 v 到 u 的路径，则称此有向图为**强连通图**(Strongly Connected Digraph)。例如，图 1.12(a)和图 1.2(b)所示的有向图就是强连通图。

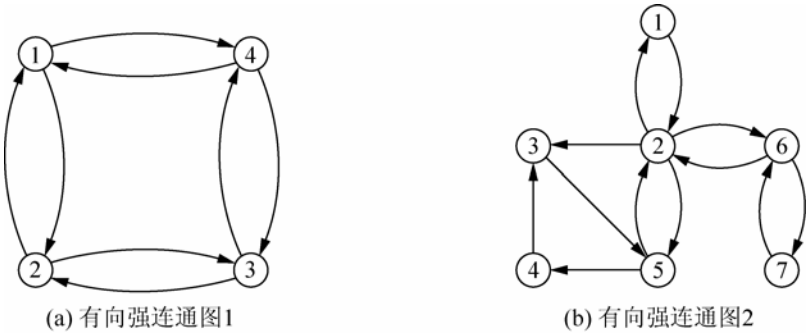


图 1.12　强连通图

对于非强连通图，其极大强连通子图称为其**强连通分量**(Strongly Connected Component)。例如，图 1.13(a)所示的有向图 G_2 就是非强连通图，它包含 3 个强连通分量，如图 1.13(b)所示。其中，顶点 2、3、4、5 构成一个强连通分量，在这个子图中，每一对顶点 u 和 v，既存在从 u 到 v 的路径，也存在从 v 到 u 的路径；顶点 1、6、8 也构成一个强连通分量，顶点 7 自成一个强连通分量。

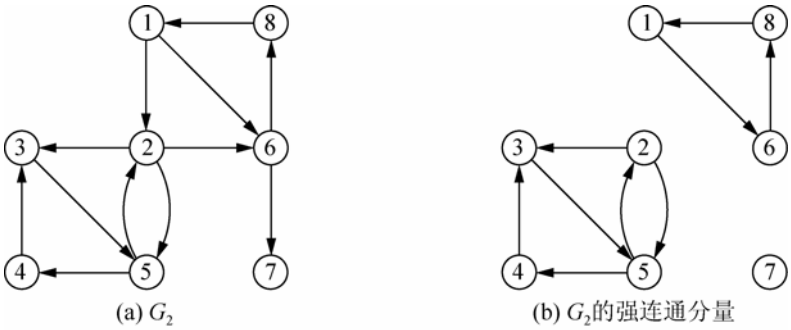


图 1.13　有向图的强连通分量

1.1.10 权值、有向网与无向网

权值(Weight)：某些图的边具有与它相关的数，称为权值。这些权值可以表示从一个顶点到另一个顶点的距离、花费的代价、所需的时间等。如果一个图，其所有边都具有权值，则称为**加权图**(Weighted Graph)，或者称为**网络**(Net)。根据网络中的边是否具有方向性，又可以分为**有向网**(Directed Net)和**无向网**(Undirected Net)。网络也可以用 $G(V, E)$表示，其中边的集合 E 中每个元素包含 3 个分量：边的两个顶点和权值。

例如，图 1.14(a)所示的无向网可表示为 $G_1(V, E)$，其中顶点集合 $V(G_1) = \{1, 2, 3, 4, 5, 6, 7\}$；边的集合为：

$$E(G_1) = \{ (1, 2, 28), (1, 6, 10), (2, 3, 16), (2, 7, 14), (3, 4, 12), (4, 5, 22), (4, 7, 18), (5, 6, 25), (5, 7, 24) \}。$$

在边的集合中，每个元素的第 3 个分量表示该边的权值。

如图 1.14(b)所示的有向网可以表示为 $G_2(V, E)$，其中顶点集合 $V(G_1) = \{1, 2, 3, 4, 5, 6, 7\}$；边的集合为：

$$E(G_2) = \{ <1, 2, 12>, <2, 4, 85>, <3, 2, 43>, <4, 3, 65>, <5, 1, 58>, <5, 2, 90>, <5, 6, 19>, <5, 7, 70>, <6, 4, 24>, <7, 6, 50> \}。$$

同样在边的集合中，每个元素的第 3 个分量也表示该边的权值。

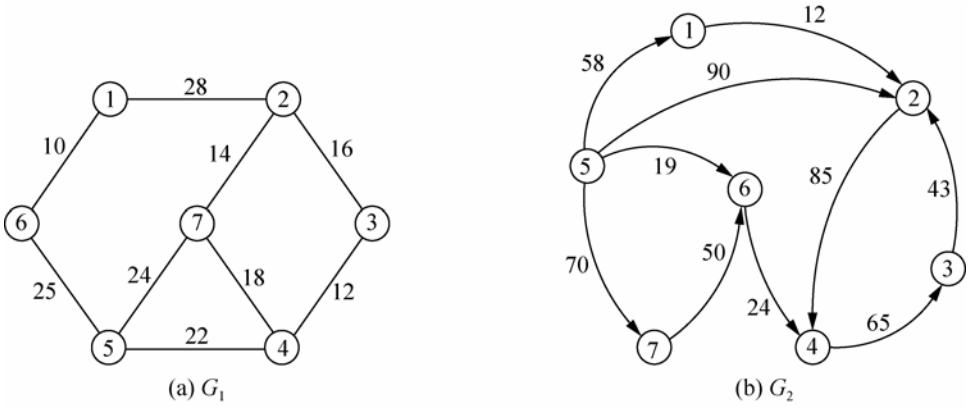


图 1.14 无向网与有向网

1.2 图的存储表示

图的存储表示方法有很多种，常用的有 3 种：邻接矩阵(Adjacency Matrix)、邻接表(Adjacency List)、邻接多重表(Adjacency Multilists)。本节只介绍前面两种。

1.2.1 邻接矩阵

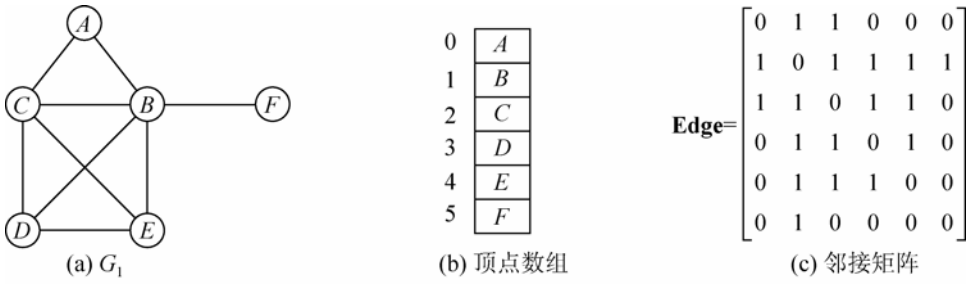
1. 有向图和无向图的邻接矩阵

在邻接矩阵存储方法中，除了一个记录各个顶点信息的**顶点数组**外，还有一个表示各个顶点之间关系的矩阵，称为**邻接矩阵**。设 $G(V, E)$是一个具有 n 个顶点的图，则图的邻接

矩阵是一个 $n \times n$ 的二维数组，在本书中用 **Edge**[n][n]表示，它的定义为：

$$\mathbf{Edge}[i][j]=\begin{cases}1 & \text{如果} <i,j>\in E, \text{或} (i,j)\in E \\ 0 & \text{否则}\end{cases} \tag{1-2}$$

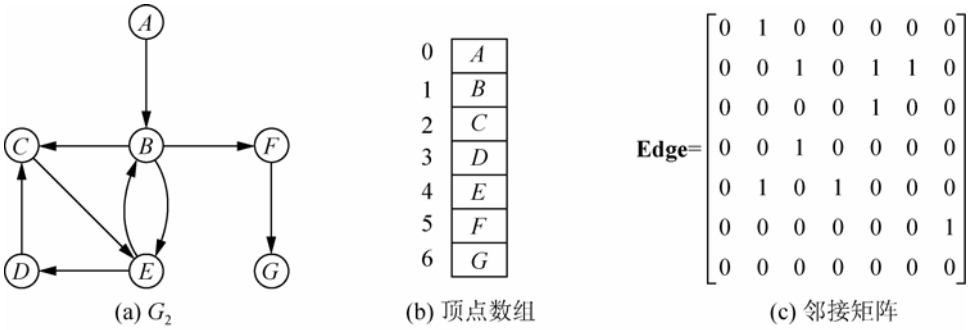
例如，图 1.15 给出了图 1.1(a)中的无向图 $G_1(V, E)$及其邻接矩阵表示。在图 1.15 中，为了表示顶点信息，特意将顶点的标号用字母 *A*、*B*、*C*、*D*、*E* 和 *F* 表示，各顶点的信息存储在顶点数组中，如图 1.15(b)所示，注意在 C/C++语言中，数组元素下标是从 0 开始计起的。G_1 的邻接矩阵如图 1.15(c)所示，从图中可以看出，无向图的邻接矩阵是沿主对角线对称的。



(a) G_1　　(b) 顶点数组　　(c) 邻接矩阵

图 1.15　无向图的邻接矩阵表示

又如，图 1.16 给出了图 1.1(b)中的有向图 $G_2(V, E)$及其邻接矩阵表示。同样，为了表示顶点信息，在图 1.16 中特意将顶点的标号用字母 *A*、*B*、*C*、*D*、*E*、*F* 和 *G* 表示，各顶点的信息存储在顶点数组中，如图 1.16(b)所示。G_2 的邻接矩阵如图 1.16(c)所示，从该图中可以看出，有向图的邻接矩阵不一定是沿主对角线对称的。



(a) G_2　　(b) 顶点数组　　(c) 邻接矩阵

图 1.16　有向图的邻接矩阵表示

注意：如果图中存在**自身环**(Self Loop，连接某个顶点自身的边)和**重边**(Multiple Edge)，多条边的起点一样，终点也一样，也称为**平行边(**Parallel Edge)的情形，则无法用邻接矩阵存储。

从图的邻接矩阵可以获得什么信息？对无向图的邻接矩阵来说，如果 **Edge**[i][j] = 1，则表示顶点 i 和顶点 j 之间有一条边。因此，邻接矩阵 **Edge** 第 i 行所有元素中元素值为 1 的个数表示顶点 i 的度数，第 i 列所有元素中元素值为 1 的个数也表示顶点 i 的度数，即：

$$\deg(i)=\sum_{j=0}^{n-1}\mathbf{Edge}[i][j]=\sum_{j=0}^{n-1}\mathbf{Edge}[j][i] \tag{1-3}$$

而对有向图的邻接矩阵来说，如果 **Edge**[*i*][*j*] = 1，则表示存在从顶点 *i* 到顶点 *j* 有一条有向边，*i* 是起点，*j* 是终点。因此，邻接矩阵 **Edge** 第 *i* 行所有元素中元素值为 1 的个数表示顶点 *i* 的出度，第 *i* 列所有元素中元素值为 1 的个数表示顶点 *i* 的入度，即：

$$\mathrm{od}(i)=\sum_{j=0}^{n-1}\mathbf{Edge}[i][j],\ \mathrm{id}(i)=\sum_{j=0}^{n-1}\mathbf{Edge}[j][i] \tag{1-4}$$

说明：

(1) 由于在 C/C++语言中，数组元素下标是从 0 开始计起的，而图的顶点序号通常是从 1 开始计起的，所以数组元素下标 *i* 与第 *i*+1 个顶点对应。在本书中，为避免繁琐，如无特殊说明，都隐含这种对应关系。例如，例 1.1 的分析中提到邻接矩阵中第 *i* 行与第 *i*+1 个顶点对应。

(2) 邻接矩阵经过其他运算如乘方后，还可以获得更多的信息，具体请参见相关图论书籍，本书不讨论这些运算。

例 1.1 用邻接矩阵存储有向图，并输出各顶点的出入和入度。

输入描述：

输入文件中包含多个测试数据，每个测试数据描述了一个无权有向图。每个测试数据的第一行为两个正整数 *n* 和 *m*，1≤*n*≤100，1≤*m*≤500，分别表示该有向图的顶点数目和边数，顶点的序号从 1 开始计起。接下来有 *m* 行，每行为两个正整数，用空格隔开，分别表示一条边的起点和终点。每条边出现一次且仅一次，图中不存在自身环和重边。输入文件最后一行为 0 0，表示输入数据结束。

输出描述：

对输入文件中的每个有向图，输出两行：第 1 行为 *n* 个正整数，表示每个顶点的出度；第 2 行也为 *n* 个正整数，表示每个顶点的入度。每两个正整数之间用一个空格隔开，每行的最后一个正整数之后没有空格。

样例输入：

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

样例输出：

```
1 3 1 1 2 1 0
0 2 2 1 2 1 1
```

分析：

在程序中可以使用一个二维数组 **Edge** 存储表示邻接矩阵。输入文件中顶点的序号是从 1 开始计起的，所以在将有向边<*u*, *v*>存储表示到邻接矩阵 **Edge** 时，需要将元素 **Edge**[*u*−1][*v*−1]的值置为 1。

本题中的有向图都是无权图，邻接矩阵中每个元素要么为 1，要么为 0。第 *i*+1 个顶点

的出度等于邻接矩阵中第 i 行所有元素中元素值为 1 的个数，把第 i 行所有元素值累加起来，得到的结果也是该顶点的出度。同理，在计算第 i+1 个顶点的入度时，也只需将第 i 列所有元素值累加起来即可。

题目要求在输出 n 个顶点的出度(和入度)时，每两个正整数之间用一个空格隔开，最后一个正整数之后没有空格。可以采取的策略是：输出第 0 个顶点的出度时前面没有空格，输出后面 n−1 个顶点的出度时都先输出一个空格。

代码如下：

```
#define MAXN 100                          //顶点个数最大值
int Edge[MAXN][MAXN];                     //邻接矩阵
int main( )
{
    int i, j;                             //循环变量
    int n, m;                             //顶点个数、边数
    int u, v;                             //边的起点和终点
    int od, id;                           //顶点的出度和入度
    while( 1 )
    {
        scanf( "%d%d", &n, &m );          //读入顶点个数 n 和边数 m
        if( n==0 && m==0 )  break;        //输入数据结束
        memset( Edge, 0, sizeof(Edge) );
        for( i=1; i<=m; i++ )
        {
            scanf( "%d%d", &u, &v );      //读入边的起点和终点
            Edge[u-1][v-1]=1;             //构造邻接矩阵
        }
        for( i=0; i<n; i++ )              //求各顶点的出度
        {
            od=0;
            for( j=0; j<n; j++ )  od+=Edge[i][j];  //累加第 i 行
            if(i==0)  printf( "%d", od );
            else  printf( " %d", od );
        }
        printf( "\n" );
        for( i=0; i<n; i++ )                          //求各顶点的入度
        {
            id=0;
            for( j=0; j<n; j++ )  id += Edge[j][i];//累加第 i 列
            if(i==0)  printf( "%d", id );
            else  printf( " %d", id );
        }
        printf( "\n" );
    }
    return 0;
}
```

例 1.2　青蛙的邻居(Frogs' Neighborhood)

题目来源：

POJ Monthly-2004.05.15, POJ1659

题目描述：

未名湖附近共有 n 个大小湖泊 $L_1, L_2, \cdots, L_n$(其中包括未名湖)，每个湖泊 L_i 里住着一只青蛙 $F_i(1 \leqslant i \leqslant n)$。如果湖泊 L_i 和 L_j 之间有水路相连，则青蛙 F_i 和 F_j 互称为邻居。现在已知每只青蛙的邻居数目为 $x_1, x_2, \cdots x_n$，请给出每两个湖泊之间的相连关系。

输入描述：

第一行是测试数据的组数 $t(0 \leqslant t \leqslant 20)$。每组数据包括两行，第一行是整数 $n(2 \leqslant n \leqslant 10)$，第二行是 n 个整数，$x_1, x_2, \cdots, x_n(0 \leqslant x_i < n)$。

输出描述：

对输入的每组测试数据，如果不存在可能的相连关系，输出"NO"；否则输出"YES"，并用 $n \times n$ 的矩阵表示湖泊间的相邻关系，即如果湖泊 i 与湖泊 j 之间有水路相连，则第 i 行的第 j 个数字为 1，否则为 0。每两个数字之间输出一个空格。如果存在多种可能，只需给出一种符合条件的情形。相邻两组测试数据之间输出一个空行。

样例输入：

```
2
7
4 3 1 5 4 2 1
6
4 3 1 4 2 0
```

样例输出：

```
YES
0 1 1 1 1 0 0
1 0 0 1 1 0 0
1 0 0 0 0 0 0
1 1 0 0 1 1 1
1 1 0 1 0 1 0
0 0 0 1 1 0 0
0 0 0 1 0 0 0

NO
```

分析：

本题的意思实际上是给定一个非负整数序列，问是不是一个可图的序列，也就是说能不能根据这个序列构造一个图。这需要根据 Havel-Hakimi 定理(定理 1.2)中的方法来构图，并在构图中判断是否出现了不合理的情形。有以下两种不合理的情形。

(1) 某次对剩下序列排序后，最大的度数(设为 d_1)超过了剩下的顶点数。

(2) 对最大度数后面的 d_1 个度数各减 1 后，出现了负数。

一旦出现了以上两种情形之一，即可判定该序列不是可图的。

如果一个序列是可图的，本题还要求输出构造得到的图的邻接矩阵，实现思路如下。

(1) 为了确保顶点序号与输入时的度数顺序一致，特意声明了一个 vertex 结构体，包含了顶点的度和序号两个成员。

(2) 每次对剩下的顶点按度数从大到小的顺序排序后，设最前面的顶点(即当前度数最大的顶点)序号为 i、度数为 d_1，对后面 d_1 个顶点每个顶点(序号设为 j)度数减 1，并连边，即在邻接矩阵 **Edge** 中设置 **Edge**[i][j]和 **Edge**[j][i]为 1。

代码如下：

```
#define N 15
struct vertex
{
    int degree; //顶点的度
```

```
    int index;  //顶点的序号
}v[N];
int cmp( const void *a, const void *b )
{
    return ((vertex*)b)->degree-((vertex*)a)->degree;
}
int main( )
{
    int r, k, p, q;         //循环变量
    int i, j;               //顶点序号(用于确定图中边的两个顶点)
    int d1;                 //对剩下序列排序后第 1 个顶点(度数最大的顶点)的度数
    int T, n;               //测试数据个数，湖泊个数
    int Edge[N][N], flag;   //邻接矩阵，是否存在合理相邻关系的标志
    scanf( "%d", &T );
    while( T-- )
    {
        scanf( "%d", &n );
        for( i=0; i<n; i++ )
        {
            scanf( "%d", &v[i].degree );
            v[i].index=i;           //按输入顺序给每个湖泊编号
        }
        memset( Edge, 0, sizeof(Edge) );
        flag=1;
        for( k=0; k<n&&flag; k++ )
        {
            //对 v 数组后 n-k 个元素按非递增顺序排序
            qsort( v+k, n-k, sizeof(vertex), cmp );
            i=v[k].index;           //第 k 个顶点的序号
            d1=v[k].degree;
            if( d1>n-k-1 )  flag=0;
            for( r=1; r<=d1&&flag; r++ )
            {
                j=v[k+r].index;     //后边 d1 个顶点中每个顶点的序号
                if( v[k+r].degree<=0 )  flag=0;
                v[k+r].degree--;
                Edge[i][j]=Edge[j][i]=1;
            }
        }
        if( flag )
        {
            puts( "YES" );
            for( p=0; p<n; p++ )
            {
                for( q=0; q<n; q++ )
                {
                    if(q)  printf( " " );
                    printf( "%d", Edge[p][q] );
                }
                puts( "" );     //换行
            }
```

```
            }
            else  puts( "NO" );
            if(T)  puts( "" );      //换行
        }
        return 0;
    }
```

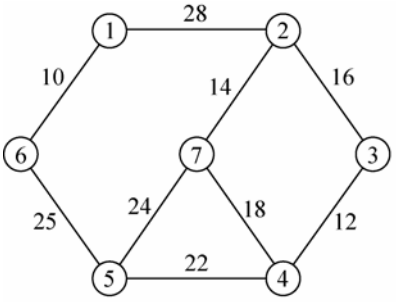
2. 有向网和无向网的邻接矩阵

对于网络(即带权值的图)，邻接矩阵的定义为：

$$\mathbf{Edge}[i][j]=\begin{cases} W(i,j) & \text{如果} i\ !=j\text{，且} <i,j>\in E(\text{或}(i,j)\in E) \\ \infty & \text{如果} i\ !=j\text{，且} <i,j>\notin E(\text{或}(i,j)\notin E) \\ 0 & \text{对角线上的位置，即} i=j \end{cases} \tag{1-5}$$

在编程实现时，可以用一个比较大的常量表示无穷大∞。

图 1.17 给出了图 1.14(a)中的无向网 $G_1(V, E)$及其邻接矩阵表示。在无向网的邻接矩阵中，如果 0<**Edge**[*i*][*j*]<∞，则顶点 *i* 和顶点 *j* 之间有一条无向边，其权值为 **Edge**[*i*][*j*]。从图中可以看出，无向网的邻接矩阵也是沿主对角线对称的。



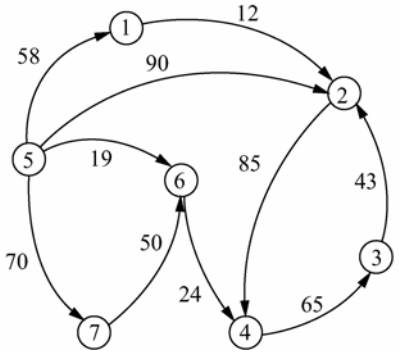
(a) G_1

$$\mathbf{Edge}=\begin{bmatrix} 0 & 28 & \infty & \infty & \infty & 10 & \infty \\ 28 & 0 & 16 & \infty & \infty & \infty & 14 \\ \infty & 16 & 0 & 12 & \infty & \infty & \infty \\ \infty & \infty & 12 & 0 & 22 & \infty & 18 \\ \infty & \infty & \infty & 22 & 0 & 25 & 24 \\ 10 & \infty & \infty & \infty & 25 & 0 & \infty \\ \infty & 14 & \infty & 18 & 24 & \infty & 0 \end{bmatrix}$$

(b) 邻接矩阵

图 1.17　无向网的邻接矩阵表示

图 1.18 给出了图 1.14(b)中的有向网 $G_2(V, E)$及其邻接矩阵表示。在有向网的邻接矩阵中，如果 0<**Edge**[*i*][*j*]<∞，则从顶点 *i* 到顶点 *j* 有一条有向边，其权值为 **Edge**[*i*][*j*]。从图中可以看出，有向网的邻接矩阵不一定是沿对主角线对称的。



$$\mathbf{Edge}=\begin{bmatrix} 0 & 12 & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & 85 & \infty & \infty & \infty \\ \infty & 43 & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & 65 & 0 & \infty & \infty & \infty \\ 58 & 90 & \infty & \infty & 0 & 19 & 70 \\ \infty & \infty & \infty & 24 & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & 50 & 0 \end{bmatrix}$$

图 1.18　有向网的邻接矩阵表示

3. 关于邻接矩阵的进一步说明

在求解 ACM/ICPC 实际题目时，有时并不严格按照公式(1-2)或(1-5)来定义邻接矩阵。例如，有时为了处理方便的需要，可以将有向网(或无向网)的邻接矩阵中对角线元素也定义成+∞。

1.2.2　邻接表

尽管 ACM/ICPC 中绝大多数图论题目在求解时可以采用邻接矩阵存储图，但由于邻接矩阵无法存储带自身环(或重边)的图，所以有时不得不采用邻接表来存储图。另外，当图的边数(相对于邻接矩阵中的元素个数，即 $n \times n$)较少时，使用邻接矩阵存储会浪费较多的存储空间，而用邻接表存储可以节省存储空间。

所谓**邻接表**(Adjacency list)，就是把从同一个顶点发出的边连接在同一个称为**边链表**的单链表中。边链表的每个结点代表一条边，称为**边结点**。每个边结点有 2 个域：该边终点的序号，以及指向下一个边结点的指针。在邻接表中，还需要一个用于存储顶点信息的顶点数组。

例如，图 1.19(a)所示的有向图对应的邻接表如图 1.19(b)所示。在**顶点数组**中，每个元素有两个成员：一个成员用来存储顶点信息；另一个成员为该顶点的边链表的表头指针，指向该顶点的边链表。如果没有从某个顶点发出的边，则该顶点没有边链表，因此表头指针为空，如图 1.19(b)中的顶点 *G*。在该图中，如果指针为空，则用符号“∧”表示。

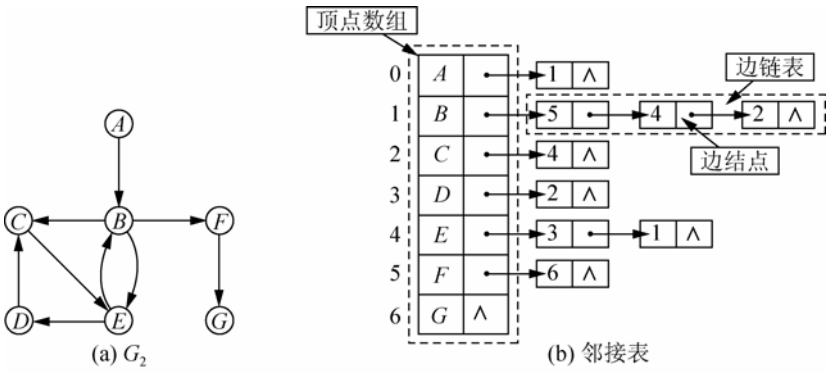


图 1.19　有向图的邻接表(出边表)

在邻接表每个顶点的边链表中，各边结点所表示的边都是从该顶点发出的边，因此这种邻接表也称为**出边表**。采用邻接表存储图时，求顶点的出度很方便，只需要统计每个顶点的边链表中边结点的个数即可，但在求顶点的入度时就比较麻烦。

在图 1.19(b)中，顶点 *B* 的边链表有 3 个边结点，分别表示边<*B*, *F*>、<*B*, *E*>和<*B*, *C*>，因此顶点 *B* 的出度为 3；顶点 *C* 的边链表中只有 1 个边结点，表示边<*C*, *E*>，因此顶点 *C* 的出度为 1。

如果需要统计各顶点的入度，可以采用逆邻接表存储表示图。所谓**逆邻接表**，也称为**入边表**，就是把进入同一个顶点的边链接在同一个边链表中。

例如，图 1.20(a)所示的有向图对应的逆邻接表如图 1.20(b)所示。在图 1.20(b)中，顶点 *B* 的边链表有 2 个边结点，分别表示边<*E*, *B*>和<*A*, *B*>，因此顶点 *B* 的入度为 2；顶点 *C* 的

边链表中也有 2 个边结点，分别表示边<*D*, *C*>和<B, *C*>，因此顶点 *C* 的入度也为 2。

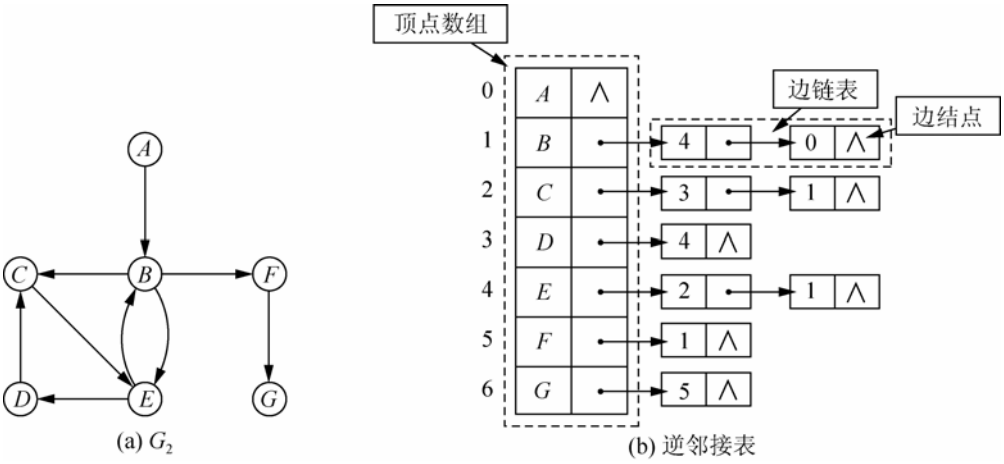


图 1.20　有向图的逆邻接表(入边表)

因为无向图中的边没有方向性，所以无向图的邻接表没有入边表和出边表之分。在无向图的邻接表中，与顶点 *v* 相关联的边都链接到该顶点的边链表中。无向图的每条边在邻接表里出现两次。例如，图 1.21(a)所示的无向图对应的邻接表如图 1.21(b)所示。

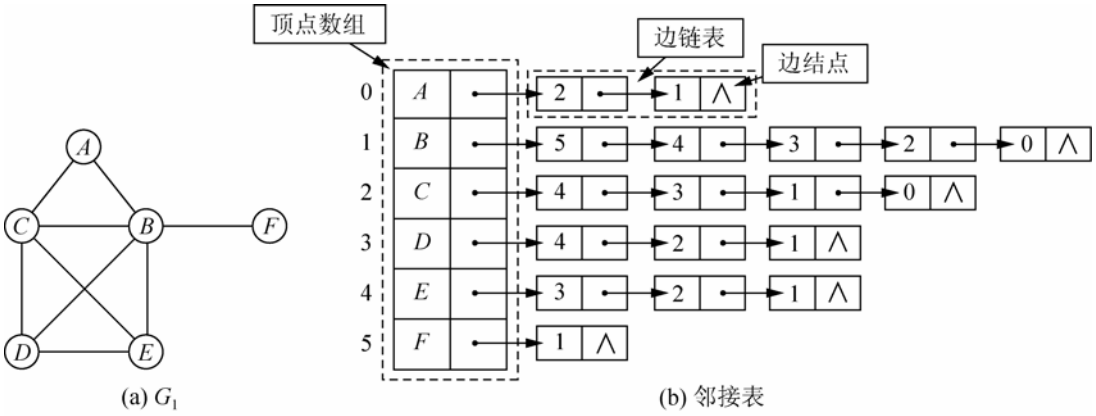


图 1.21　无向图的邻接表

在图 1.21(b)中，顶点 *B* 的边链表中有 5 个边结点，分别表示边(*B*, *F*)、(*B*, *E*)、(*B*, *D*)、(*B*, *C*)和(*B*, *A*)，因此顶点 *B* 的度为 5；顶点 *C* 的边链表中有 4 个边结点，分别表示边(*C*, *E*)、(*C*, *D*)、(*C*, *B*)和(*C*, *A*)，因此顶点 *C* 的度为 4。边(*B*, *C*)分别出现在顶点 *B* 和顶点 *C* 的边链表中。

说明：如果用邻接表存储有向网或无向网，则在边结点中还应增加一个成员，用于存储边的权值。

接下来以有向图为例介绍邻接表的实现方法。为了方便求解顶点的出度和入度，在实现时，把出边表和入边表同时包含在图的邻接表结构中。

有向图的邻接表用一个结构体 LGraph 存储表示，其中包含 3 个成员：顶点数组 vertexs，顶点数 vexnum 和边的数目 arcnum，其中顶点数组 vertexs 中每个元素都是 VNode 结构体

变量。VNode 结构体变量存储图中每个顶点，它包含 3 个成员：顶点信息、出边表的表头指针和入边表的表头指针，其中后面两个成员都是 ArcNode 结构体类型的指针。ArcNode 结构体存储边链表中的边结点，它包含两个成员：边的另一个邻接点的序号，以及指向下一个边结点的指针。

上述提及的 3 个结构体声明如下。

```
#define MAXN 100
struct ArcNode              //边结点
{
    int adjvex;             //有向边的另一个邻接点的序号
    ArcNode *nextarc;       //指向下一个边结点的指针
};
struct VNode                //顶点
{
    int data;               //顶点信息
    ArcNode *head1;         //出边表的表头指针
    ArcNode *head2;         //入边表的表头指针
};
struct LGraph               //图的邻接表存储结构
{
    VNode vertexs[MAXN];    //顶点数组
    int vexnum, arcnum;     //顶点数和边(弧)数
};
LGraph lg;                  //图(邻接表存储)
```

声明了有向图的邻接表结构体 LGraph 后，构造有向图 *G* 可以采取全局函数 CreateLG( ) 来实现，代码如下。在 CreateLG( )函数中，约定：构造邻接表时，先输入顶点个数和边数，然后按“起点/终点”的格式输入每条有向边，详见例 1.3 的输入/输出描述；注意在输入数据时，顶点序号从 1 开始计起，而在存储时顶点序号从 0 开始计起。

```
void CreateLG( )        //采用邻接表存储表示，构造有向图 G
{
    int i=0;            //循环变量
    ArcNode *pi;        //用来构造边链表的边结点指针
    int v1,v2;          //有向边的两个顶点
    lg.vexnum=lg.arcnum=0;
    scanf( "%d%d", &lg.vexnum, &lg.arcnum );   //首先输入顶点个数和边数
    for( i=0; i<lg.vexnum; i++ )               //初始化表头指针为空
        lg.vertexs[i].head1=lg.vertexs[i].head2=NULL;
    for( i=0; i<lg.arcnum; i++ )
    {
        scanf( "%d%d", &v1, &v2);              //输入一条边的起点和终点
        v1--;  v2--;
        pi=new ArcNode;                        //假定有足够空间
        pi->adjvex=v2;
        pi->nextarc=lg.vertexs[v1].head1;      //插入链表
        lg.vertexs[v1].head1=pi;
        pi=new ArcNode;                        //假定有足够空间
```

```
            pi->adjvex=v1;
            pi->nextarc=lg.vertexs[v2].head2;      //插入链表
            lg.vertexs[v2].head2=pi;
        }                                          //end of for
    }                                              //end of CreateLG
```

构造有向图邻接表(出边表)的过程，可以用图 1.22 所示的过程来表示。在读入边“2 3”后，申请一个边结点，并链入到顶点 1 的表头指针后面。再读入边“2 5”，又申请一个边结点，再插入到顶点 1 的表头指针后面。如图 1.22(b)所示，在顶点 1 表头指针与边结点<1, 2>之间插入了一个边结点<1, 4>。请注意输入数据中顶点序号是从 1 开始计起的，在处理时先减 1 再构造边结点。

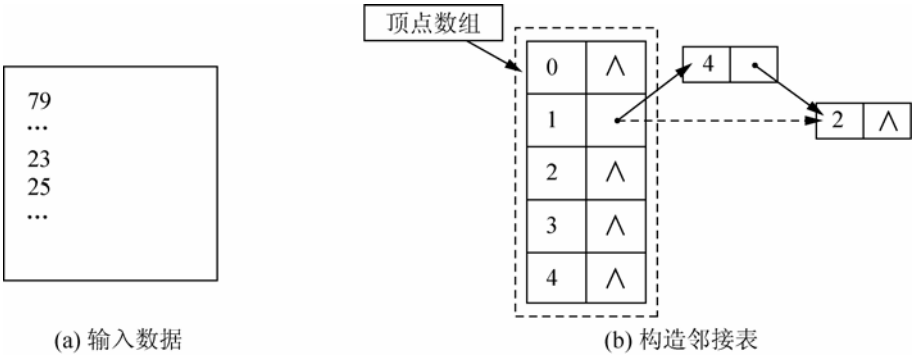


图 1.22　构造有向图邻接表的过程

使用完有向图 *G* 的邻接表后，应该释放图 *G* 各顶点的出边表和入边表中所有边结点所占的存储空间，可以使用全局函数 DeleteLG 实现，代码详见例 1.3。

以上声明的邻接表及定义的全局函数使用方法详见例 1.3。

例 1.3　用邻接表存储有向图，并输出各顶点的出度和入度。

输入描述：

输入文件中包含多个测试数据，每个测试数据描述了一个无权有向图。每个测试数据的第一行为两个正整数 *n* 和 *m*，1≤*n*≤100，1≤*m*≤500，分别表示该有向图的顶点数目和边数，顶点的序号从 1 开始计起。接下来有 *m* 行，每行为两个正整数，用空格隔开，分别表示一条边的起点和终点。每条边出现一次且仅一次，图中不存在自身环和重边。输入文件最后一行为 0 0，表示输入数据结束。

输出描述：

对输入文件中的每个有向图，输出两行：第 1 行为 *n* 个正整数，表示每个顶点的出度；第 2 行也为 *n* 个正整数，表示每个顶点的入度。每两个正整数之间用一个空格隔开，每行的最后一个正整数之后没有空格。

样例输入：

```
4 7
1 4
2 1
2 2
2 3
```

样例输出：

```
1 4 0 2
1 2 3 1
```

```
2 3
4 2
4 3
0 0
```

分析：

用邻接表存储图，可以表示重边和自身环的情形。例如，样例输入中第 2 个测试数据所描述的有向图及对应的邻接表如图 1.23 所示，该有向图既包含重边，又包含自身环。在图 1.23(a)中，有向边<2, 2>为自身环，对应到图 1.23(b)所示的邻接表中，顶点 1 的入边表和出边表都有一个边结点，其 adjvex 分量均为 1。另外，在图 1.23(a)中，<2, 3>和<2, 3>为重边，对应到图 1.23(b)，在顶点 1 的出边表中，有两个边结点的 adjvex 分量均为 2；以及在顶点 2 的入边表中，有两个边结点的 adjvex 分量均为 1。

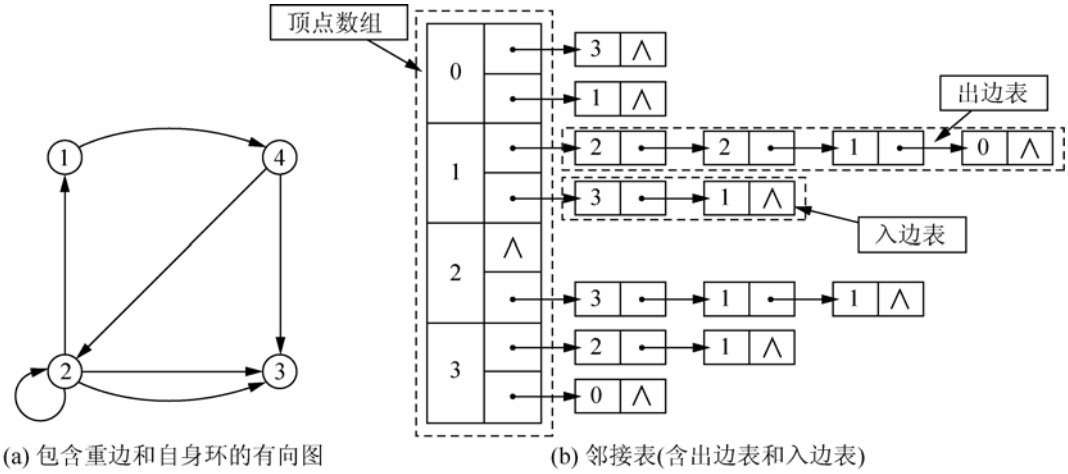


图 1.23　包含重边和自身环的有向图的邻接表

统计第 *i* 个顶点的出度的方法是在其出边表中统计边结点的个数，统计第 *i* 个顶点的入度的方法是在其入边表中统计边结点的个数。

注意，为了实现“当读入的顶点个数 *n* 为 0 时，输入结束，退出 main 函数”，下面的代码将设置邻接表的 vexnum 成员和 arcnum 成员，以及读入顶点个数的处理代码由 CreateLG 函数移至 main 函数的 while 循环中。

代码如下：

```
#define MAXN 100
struct ArcNode              //边结点
{
    int adjvex;             //有向边的另一个邻接点的序号
    ArcNode *nextarc;       //指向下一个边结点的指针
};
struct VNode                //顶点
{
    int data;               //顶点信息
    ArcNode *head1;         //出边表的表头指针
    ArcNode *head2;         //入边表的表头指针
```

```
};
struct LGraph               //图的邻接表存储结构
{
    VNode vertexs[MAXN];    //顶点数组
    int vexnum, arcnum;     //顶点数和边数
};
LGraph lg;  //图(邻接表存储)
void CreateLG( ) //采用邻接表存储表示，构造有向图 G
{
    int i=0;                        //循环变量
    ArcNode *pi;                    //用来构造边链表的边结点指针
    int v1, v2;                     //有向边的两个顶点
    for( i=0; i<lg.vexnum; i++ )    //初始化表头指针为空
        lg.vertexs[i].head1=lg.vertexs[i].head2=NULL;
    for( i=0; i<lg.arcnum; i++ )
    {
        scanf( "%d%d", &v1, &v2);           //输入一条边的起点和终点
        v1--;  v2--;
        pi=new ArcNode;                     //假定有足够空间
        pi->adjvex=v2;
        pi->nextarc=lg.vertexs[v1].head1;   //插入链表
        lg.vertexs[v1].head1=pi;
        pi=new ArcNode;                     //假定有足够空间
        pi->adjvex=v1;
        pi->nextarc=lg.vertexs[v2].head2;   //插入链表
        lg.vertexs[v2].head2=pi;
    }//end of for
}//end of CreateLG
//释放图 G 邻接表各顶点的边链表中的所有边结点所占的存储空间
void DeleteLG( )
{
    int i;          //循环变量
    ArcNode *pi;    //用来指向边链表中各边结点的指针
    for( i=0; i<lg.vexnum; i++ )
    {
        pi=lg.vertexs[i].head1;
        //释放第 i 个顶点出边表各边结点所占的存储空间
        while( pi!=NULL )
        {
            lg.vertexs[i].head1=pi->nextarc;
            delete pi;
            pi=lg.vertexs[i].head1;
        }
        pi=lg.vertexs[i].head2;
        //释放第 i 个顶点入边表各边结点所占的存储空间
        while( pi!=NULL )
        {
            lg.vertexs[i].head2=pi->nextarc;
            delete pi;
            pi=lg.vertexs[i].head2;
        }
```

```
        }
    }
    int main( )
    {
        int i;          //循环变量
        int id, od;     //顶点的入度和出度
        ArcNode *pi;    //用来遍历边链表的边结点指针
        while( 1 )
        {
            lg.vexnum=lg.arcnum=0;
            //首先输入顶点个数和边数
            scanf( "%d%d", &lg.vexnum, &lg.arcnum );
            if( lg.vexnum==0 )  break;      //输入数据结束
            CreateLG( );                    //构造有向图的邻接表结构
            for( i=0; i<lg.vexnum; i++ )    //统计各顶点出度并输出
            {
                od=0;
                pi=lg.vertexs[i].head1;
                while( pi!=NULL )
                {
            od++;
            pi=pi->nextarc;
                }
                if(i==0) printf( "%d", od );
                else printf( " %d", od );
            }
            printf( "\n" );
            for( i=0; i<lg.vexnum; i++ )    //统计各顶点入度并输出
            {
                id=0;
                pi=lg.vertexs[i].head2;
                while( pi!=NULL )
                {
                    id++;
                    pi=pi->nextarc;
                }
                if(i==0) printf( "%d", id );
                else printf( " %d", id );
            }
            printf( "\n" );
            DeleteLG( );    //释放
        }
        return 0;
    }
```

3. 关于邻接表的进一步说明

在求解 ACM/ICPC 题目时，有时并不需要严格按照 1.2.2 节中的形式来定义邻接表结构，可以根据题目的要求进行简化。例如，如果无向图(或有向图)中有 n 个顶点，其序号为 0～n，那么邻接表结构就可以简化成由 n 个表头指针所组成的数组，详见例 2.6、2.9、

5.6 等例题中的代码。

1.2.3 关于邻接矩阵和邻接表的进一步讨论

1. 存储方式对算法复杂度的影响

本书后续章节会介绍图论里很多算法，存储方式的选择对这些算法的时间复杂度和空间复杂度有直接影响。(假设图中有 n 个顶点，m 条边。)

时间复杂度：邻接表里直接存储了边的信息，浏览完所有的边，对有向图来说，时间复杂度是 $O(m)$，对无向图时间复杂度是 $O(2\times m)$。而邻接矩阵是间接存储边，浏览完所有的边，复杂度是 $O(n^2)$。

空间复杂度：邻接表里除了存储 m 条边所对应的边结点外，还需要一个顶点数组，存储各顶点的顶点信息及各边链表的表头指针，总的空间复杂度为 $O(n+m)$(或 $O(n+2m)$)；而用邻接矩阵存储图，需要 $n\times n$ 规模的存储单元，其空间复杂度为 $O(n^2)$。当边的数目相对于 $n\times n$ 比较小时，邻接矩阵里存储了较多的无用信息，用邻接表可以节省较多的存储空间。

2. 在求解问题时可以灵活地存储表示图

在求解实际问题时，有时并不需要严格采用邻接矩阵或邻接表来存储图。例如，当图中顶点个数确定以后(这里假设顶点序号是连续的)，图的结构就唯一地取决于边的信息，因此可以把每条边的信息(起点、终点、权值等)存储到一个数组里，在针对该图进行某种处理时只需要访问边的数组中每个元素即可。对于一些可以直接针对边进行处理的算法，如 4.2 节介绍的 Bellman-Ford 算法，可以采用这种存储方式来实现，详细方法见例 4.13 等例子。

练　习

1.1　编程实现：利用邻接矩阵存储一个有向图，并实现邻接矩阵的平方运算，并观察和分析平方运算后邻接矩阵中元素值的含义。

1.2　请模仿 1.2.2 节邻接表的实现方法，以及例 1.3 利用邻接表存储有向图，实现统计有向图各顶点的出度和入度的方法，编程实现：用邻接表存储一个无向图，并统计各顶点的度。注意：在构造无向图的邻接表时，每条边要分别链接到两个顶点的边链表中。