第 6 章　图

本章要讨论的“图”不是指图形、图像和数码照片，它是一种直观、简洁、优美的数学工具。它仅有两个构成要件顶点（vertex）和边（edge），但却能描述和表达自然界和人类社会生活中许多复杂的事物和关系。例如，人类一个群体中的“同学关系”和“认识关系”，计算机网络，web，城市交通网络，电力供应网络，城市自来水管网，神经网络等等，凡此种种都可以用图的形式来进行抽象表示。数据结构中用它来描述和表示多对多的复杂数据结构。目前图已经被广泛应用到语言学、逻辑学、物理、化学、通信、计算机科学、人工智能、数学的其它分支等众多学科和领域。

图结构相比前面介绍的其它数据结构更为复杂，线性结构中，结点之间是线性关系，一个结点最多只有一个直接前驱和一个直接后继；树型结构呈现明显的层次关系，每个结点最多只有一个双亲结点，但可以有多个孩子结点；而在图结构中，任意两个顶点之间都可能发生邻接关系，每个顶点都可能有多个直接前驱和多个直接后继。

本章先介绍图的定义和基本概念，然后讨论图结构在计算机中的两种最常用的存储形式。遍历算法是图结构最基本的运算，有两种遍历图的运算，即深度优先搜索遍历和广度优先搜索遍历，有关图结构上的许多运算都可借助于这两个运算的变化来实现。不仅如此，这也是软件设计中常用的经典的运算，许多其它结构上的运算也可借助这两个运算的思想来实现。

在其后所介绍的内容是图结构的应用，也就是将图应用于某一具体问题的描述及其求解实现，通过对这些问题的背景、模型抽象、求解方法的实现等的理解，可进一步掌握图结构运用于实际问题的实现，从而为应用于实际问题奠定基础。

6.1 图的定义和基本概念

考虑到部分非计算机专业学生没有先修图论课程，本节将多用一点笔墨来讨论图的基本概念，更多的内容请大家阅读专门的图论书籍。

如前所述，许多问题可描述为一组对象及其相互间的关系。此处所提及的关系为二元关系，即每一组关系中涉及到两个元素，例如，“A 和 B 是同学”，“A 认识 B”等。为便于描述，通常将这类具体领域的问题按如下方式抽象地描述为图结构。

6.1.1 图的定义

图（Graph）G 由两部分构成：顶点集合 V 和边（弧）的集合 E，

记作 G =（V，E）。其中：

顶点（vertex）用来表示和描述各种对象。数据结构中用顶点表示数据元素，相当于前面各章中所介绍的元素、结点等，在图中都表示为一个顶点。

例如：一个图的顶点集合 V= {v_1, v_2, v_3, v_4}。

边（edge）是顶点集 V 中的顶点对（偶对），表示两个顶点之间的关系。边可以加方向区分，这样边就可分为无向边和有向边（弧）。

无向边，简称**边**，由两个顶点的无序偶构成，用“（顶点 1，顶点 2）”形式表示，两个顶点位置可以互换。例：边 e_1=(v_1, v_2)，表示由顶点 v_1 和 v_2 构成一条边 e_1，交换顶点 v_1 和 v_2 的位置，即变为(v_2, v_1)，仍表示同一条边 e_1。

弧（arc），或有向边，由两个顶点的有序偶构成，用“<顶点 1，顶点 2>”形式表示，顶点位置不能互换，交换后将表示另一条弧。例：弧 e_2=<v_1, v_3>，表示由顶点 v_1 和 v_3 构成一条弧 e_2。如果交互顶点 v_1 和 v_3 的位置，即变为<v_3, v_1>，则表示另外一条弧。

有些教材规定无向边叫边，有向边叫弧，请读者阅读时注意。其中弧表示单向关系，而边表示相互关系，用离散数学中的术语来说，则分别表示为非对称关系和对称关系。如“A 和 B 是同学”是相互关系，而“A 认识 B”则是单向关系，因为 B 不一定认识 A。

图的图形化表示

上面定义的图（graph）是一种数学工具，而不是指图像或图形，这个工具只有顶点和边两种要素，但它却有着强大的形式化表达能力，在科学和工程上有许多的实际应用，是一种“简单、直观、优美、强大”的数学工具。图的图形化表示是指用图形方式来表示图的顶点和边。就是用图形上画的一个点，再加上顶点的名称表示图中的一个顶点；用图形上画出的连接两个顶点的连线表示图中的一条边；如果是弧（有向边）就在连接顶点的线条上加箭

头表示。

【例 6.1】已知图 $G_1=(V_1, E_1)$，其中：$V_1=\{v_1, v_2, v_3, v_4, v_5\}$，$E_1=\{(v_1,v_2), (v_1,v_4), (v_2,v_3), (v_3,v_4), (v_4,v_5)\}$，画出 G_1 的图形表示。

【解】从给出的条件可知图有 5 个顶点，5 条边。我们可以先画出 5 个顶点，再根据边集，在每条边关联的两个顶点之间画线条连接即可得到对应图形表示，如图 6-1 所示。

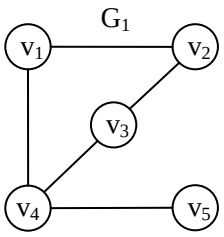


图 6-1 图 G_1 的图形化表示

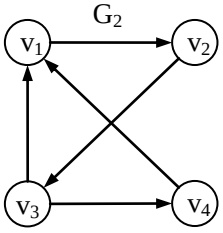


图 6-2 图 G_2 的图形化表示

【例 6.2】已知图 $G_2=(V_2,E_2)$，其中：$V_2=\{v_1, v_2, v_3, v_4\}$，$E_2=\{<v_1,v_2>, <v_2,v_3>, < v_3, v_1>, <v_3,v_4>, <v_4,v_1>\}$，画出图 G_2 的图形表示。

【解】由给出的条件可知 G_2 有 4 个顶点，5 条边，且每条边都是弧（有向边）。与上题一样先画出 4 个顶点，再根据边集，在每条边关联的两个顶点之间画线条连接，且给线条加上箭头表示边的方向，如图 6-2 所示。

我们在后面的介绍中经常会直接给出图的图形表示，以此表示一个图，而不再给出顶点集合和边的集合。

6.1.2 图的基本概念和术语

1. 无向图和有向图

无向图：每条边都是无向边。

有向图：每条边都是弧（有向边）。

混合图：既有有向边，又有无向边。（一般不讨论）

例：图 6-1 所示的 G_1 是一个无向图；图 6-2 所示的 G_2 是一个有向图。

2. 网络（带权图）

网（network）指边或弧上带有权值的图，也叫带权图。网可以是无向的，也可以是有向的。权值可以表示两个顶点之间的“距离”。计算机网络、通信网络、交通网络、供水网络等都可以抽象成这种网来表示。例：图 6-3 是一个网的实例。

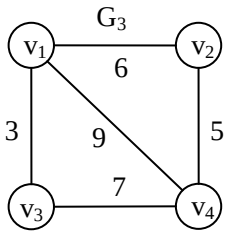
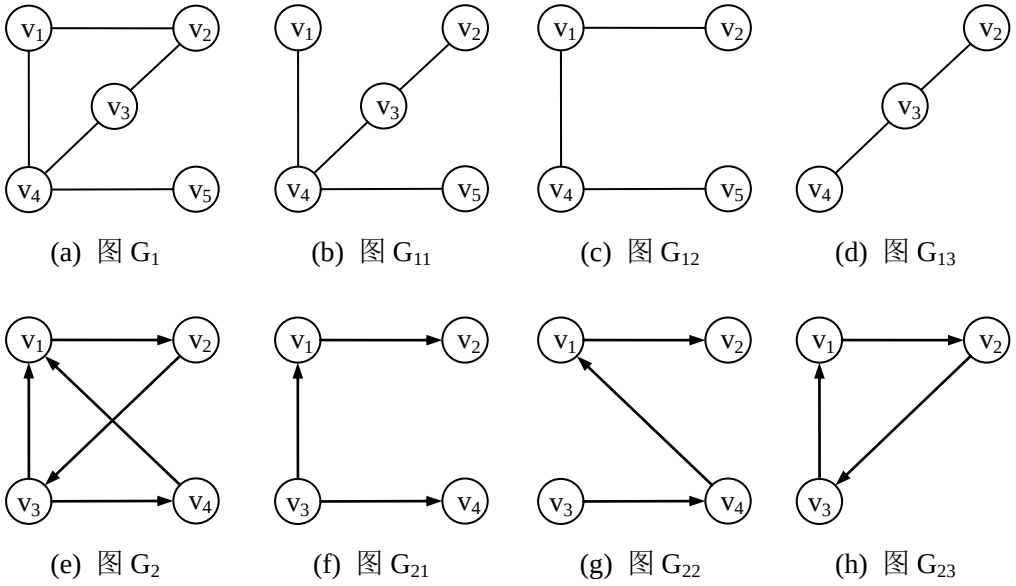


图 6-3 无向网示例

3. 子图

已知图 G=(V,E)，若另一个图 $G_1=(V_1，E_1)$是从图 G 中选取部分顶点和部分边（或弧）构成，即 $V_1⊆V$，$E_1⊆E$，则称 G_1 是 G 的子图。例：图 6-4 中，G_{11}、G_{12} 和 G_{13} 是 G_1 的子图；G_{21}、G_{22} 和 G_{23} 是图 G_2 的子图。



(a) 图 G_1　(b) 图 G_{11}　(c) 图 G_{12}　(d) 图 G_{13}

(e) 图 G_2　(f) 图 G_{21}　(g) 图 G_{22}　(h) 图 G_{23}

图 6-4 子图示意

4. 邻接(adjacent)

若两个顶点之间有边相连，则称这两个顶点邻接（相邻的）。

无向图 G=(V,E)，若顶点 u、w 之间有一条边 (u, w)∈E，则称顶点 u、w 互为邻接点。

有向图 G=(V,E)，若顶点 u、w 之间有一条弧 <u, w>∈E，则称 u 邻接 w，w 邻接自（于）u。

例：图 6-1 无向图 G_1 中，v_1 和 v_2、v_3 和 v_2、v_5 和 v_4 等互为邻接顶点。

图 6-2 有向图 G_2 中，v_1 邻接 v_2、v_3 邻接 v_1、v_2 邻接 v_3 等。

5. 顶点的度

度是指一个顶点关联的边的数量。对有向图还可以区分出入度和出度：

入度指射入顶点的边的数量。

出度指从顶点射出的边的数量。

那么，对有向图，**顶点的度=入度+出度**。

例：图 6-1 无向图 G_1，顶点 v_1、v_2 和 v_3 度为 2，v_4 度为 3，v_5 度为 1。

图 6-2 有向图 G_2 中，顶点 v_1 度为 3，入度 2，出度 1；v_2 度为 2，入度 1，出度 1；v_3 度为 3，入度 1，出度 2；v_4 度为 2，入度 1，出度 1。

由上面的讨论可知，度和边是相关的，由此我们可以推出以下结论：

无向图中：**图的边数=图的顶点度数之和 / 2**。

有向图中：**入度之和=出度之和**；

图的边数=入度之和=出度之和=图的顶点度数之和 / 2。

6. 路径和路径长度

路径（path）：通俗地说就是从图中一个顶点出发，途经图中一些边和顶点能到达另外一个顶点，把途经的顶点依次排列成一个序列叫做路径。（数学定义有点繁杂，不再给出）。对无向图，路径是双向可达的；对有向图，由于边的方向性，路径往往是单向可达的，可以区分出路径的**起点**和**终点**。

路径对图中的某些顶点或边可以多次重复走过。

路径长度：一条路径上经过的边数（或弧数）。

计算路径长度时，对重复途经的同一条边要重复计数，比如途经同一条边 3 次，长度就要加 3。

例：图 6-1 无向图 G_1 中，（v_1, v_2, v_3, v_4, v_5）是一条路径，长度为 4；（$v_3, v_2, v_1, v_4, v_3, v_2, v_1$）是一条路径，长度为 6，其中边$(v_1, v_2)$和$(v_2, v_3)$重复走过 2 次，长度中计数 4。

图 6-2 有向图 G_2 中，（v_1, v_2, v_3, v_4）是一条路径，长度为 3，起点 v_1，终点 v_4；（v_3, v_4, v_1）是一条路径，长度为 2，起点 v_3，终点 v_1；（$v_3, v_1, v_2, v_3, v_4, v_1, v_2$）是一条路径，长度为 6，起点 v_3，终点 v_2，边$< v_1, v_2>$重复走过 2 次，长度中计数 2。

7. 回路

路径上第一个顶点和最后一个顶点相同的闭合路径叫做**回路**，或叫**环**（loop）。回路中顶点或边也可以重复走过。

例：图 6-1 无向图 G_1 中，（v_1,v_2,v_3,v_4,v_1）是一个回路，长度 4；（$v_1,v_2,v_3,v_4,v_5,v_4,v_1$）是一个回路，长度 6。

图 6-2 有向图 G_2 中，（v_1,v_2,v_3,v_1）是一个回路，长度 3；（v_2,v_3,v_4,v_1,v_2）是一个回路，长度 4。

8. 简单路径

路径中途径的顶点不重复叫**简单路径**。

例：图 6-1 无向图 G_1 中，（v_1, v_2, v_3, v_4, v_5）、（v_1, v_4, v_3, v_2）等都是简单路径。

图 6-2 有向图 G_2 中，（v_1, v_2, v_3, v_4）、（v_3, v_4, v_1, v_2）等都是简单路径。

9. 简单回路

除了第一个顶点和最后一个顶点外，中间途经顶点不重复的闭合路径叫**简单回路**，或**简单环**。

例：图 6-1 无向图 G_1 中，（v_1, v_2, v_3, v_4, v_1）是一条简单回路。

图 6-2 有向图 G_2 中，（v_1, v_2, v_3, v_1）、（v_1, v_2, v_3, v_4, v_1）等都是简单回路。

10. 连通图

连通：图 G 中，如果从顶点 u 到顶点 w 有路径，则称 u 和 w 是连通的。

连通图（connected graph）：无向图 G 中，如果任意两个顶点之间都有路径，或是连通的，则称图 G 是连通图。

例：图 6-1 所示图 G_1 是一个连通图。

11. 连通分量

连通分量 (connected component)：无向图中分割出来的极大连通子图。非连通图可视为由若干连通分量（连通子图）组成。

例：图 6-5 图 G 是一个非连通图，由三个连通分量（连通子图）构成，每个连通分量都是一个连通图。

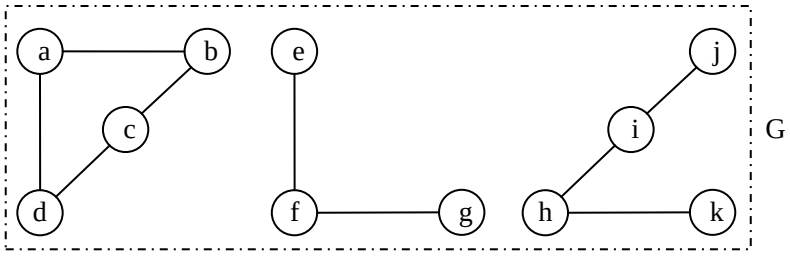


图 6-5 非连通图示例

【例 6.3】求图 6-6(a)所示图 G 的连通子图。

【解】这个图的顶点虽然画在一起，但细看就会发现它是一个非连通图，我们首先从中分割出一个极大连通分量 G_1，再在剩下部分中分割出第二个极大连通分量 G_2，最后剩下的单个顶点为一个连通分量 G_3。所以非连通图 G 由 3 个连通分量 G_1、G_2 和 G_3 组成，如图 6-6(b)所示。

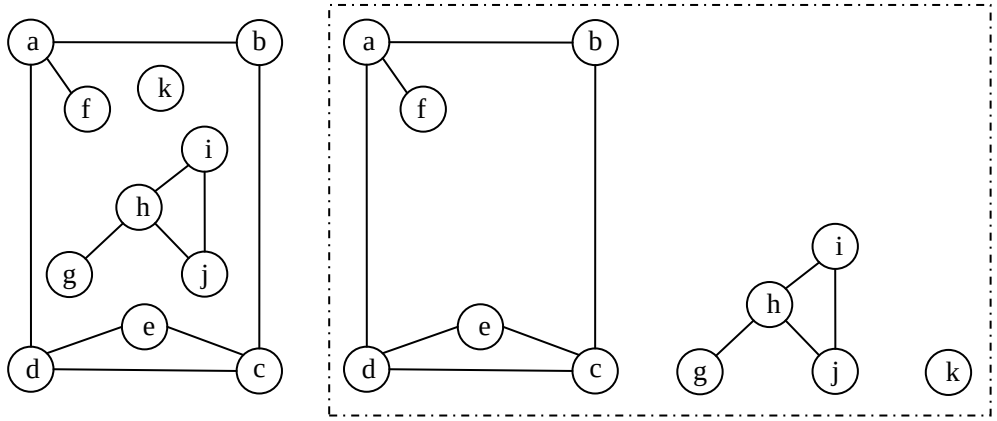


图 6-6 非连通图分割连通分量示例

12. 强连通图

强连通图 (enhance-connected graph)：有向图 G 中，若任意两个顶点之间都有路径，或连通的，则称 G 为强连通图。或：若有向图中任意两个顶点间可以互相到达，则称为强连

通图。

例：图 6-7 中图 G_1 和 G_2 都是强连通图。

n 个顶点的强连通图，至少要有 n 条边。

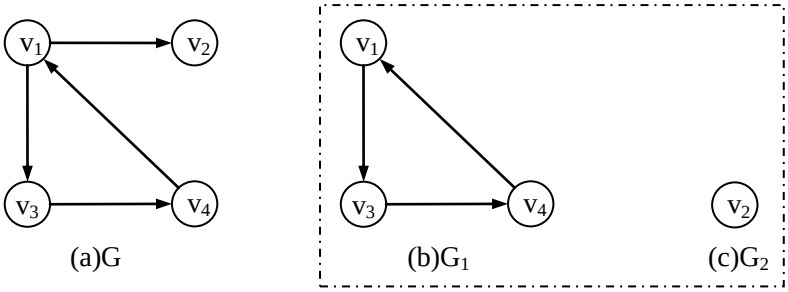
(a)G_1　　(b)G_2

图 6-7 强连通图实例

13. 强连通分量

强连通分量(enhance-connected component)：有向图中分割出来的极大连通子图。非强连通有向图由若干强连通分量构成。

例：图 6-8(a)所示有向图 G 是一个非强连通图，可分割出两个连通分量，如图 6-8(b)和图 6-8(c)。



(a)G　　(b)G_1　　(c)G_2

图 6-8 非强连通图分割连通分量实例

14. 无向完全图

若无向图 G 中任意两个顶点之间都有一条边相连，称其为**无向完全图**。

n 个顶点的无向完全图有 n(n-1)/2 条边。

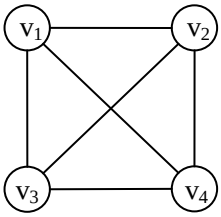
例：图 6-9(a)所示图 G_1 为 4 个顶点的无向完全图，共有 6 条边。

15. 有向完全图

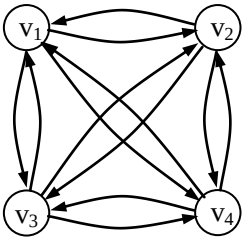
若有向图 G 中任意两个顶点之间都有一条弧（有向边）相连，称其为**有向完全图**。

n 个顶点的有向完全图有 n(n-1)条边。

例：图 6-9(b)所示图 G_2 为 4 个顶点的有向完全图，共有 12 条边。



(a) 4 个顶点的无向完全图



(b) 4 个顶点的有向完全图

图 6-9 完全图实例

16. （无向）树

若无向图连通并且无回路，则称为**（无向）树**。树还有如下几种等价的描述：

连通的无环图。

有 n-1 条边的连通图。

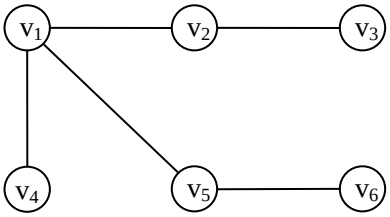
有最少边的连通图。

例：图 6-10(a)为一棵（无向）树。

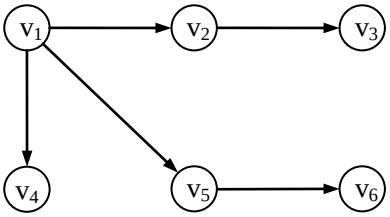
17. 有向树

有向树指仅有一个顶点入度为 0，其余顶点入度均为 1 的有向图。并称其中入度为 0 的顶点为其（有向）根。

例：图 6-10(b)为一棵有向树。



(a)一棵无向树



(b)一棵有向树

图 6-10 无向树和有向树实例

有的读者可能会想，第 5 章一章内容都在介绍树，这里怎么又出现了树呢？这里我们是从图论的角度给树下定义，可能您已注意到这个定义和第 5 章树的定义表述方式不同。从图的角度看树只是一种特殊的图，因为树的特殊性且在计算机领域有诸多重要应用，所以我们专门用很大篇幅专门讨论了树结构。

18. 连通图的生成树

生成树 (spanning tree)：一个 n 个顶点的连通图（或强连通图），其生成树是它的一个极小的连通子图，它含有图中的**全部顶点**，但只有足以构成一棵树的 **n-1 条边**。

如果在一棵生成树上添加一条边，必定构成一个环，因为这条添加的边使得它依附的那两个顶点之间有了 2 条路经。

一棵有 n 个顶点的生成树，有且仅有 n-1 条边。

如果一个图有 n 个顶点和小于 n-1 条边，则它一定是非连通图。

如果它多于 n-1 条边，则一定有环。

但是，有 n-1 条边的图，不一定是生成树。

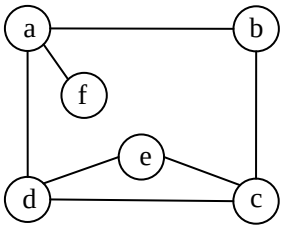
生成树是图论中一个重要的概念，在后续图的遍历算法和最小生成树等内容中都会涉及到生成树的问题。

例：图 6-11(b)是图 6-11(a)的一棵生成树。一个连通图往往可以产生多棵不同形态的生成树。

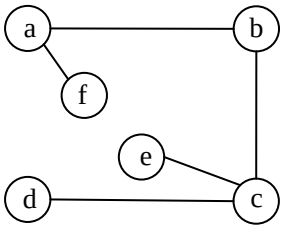
19. 非连通的生成森林 (spanning forest)

生成森林 (spanning forest)：一个非连通图的生成森林由若干棵互不相交的树组成，含有图中的**全部顶点**，但是只有足以构成若干棵不相交的树的边（弧）。

例：图 6-12(b)是图 6-12(a)的生成森林，有 2 可树构成。通常生成森林时可有多种生成方法，生成森林的形态也会不同。

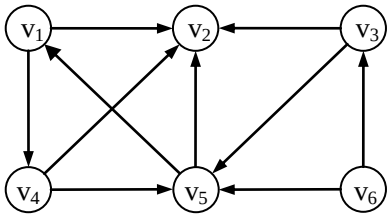


(a) 连通图 G

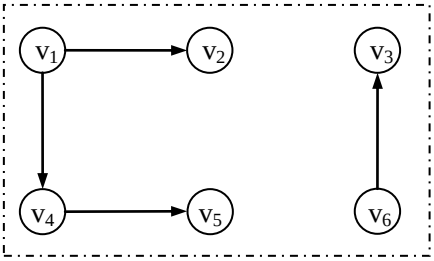


(b) 图 G 的一棵生成树

图 6-11 连通图的生成树示例



(a) 非连通图 G



(b) 图 G 的生成森林

图 6-12 非连通图的生成森林实例

6.1.3 图的顶点编号

图的各种算法在引用图的顶点时一般都用顶点的编号（序号）来代表顶点，而不是直接用顶点的元素值来区分。**所谓顶点编号就是对图中顶点从 1 开始进行顺序编号，每个顶点对应一个序号**。如图 6-13 所示，顶点 a 的编号为 3，b 为 1，另一个 b 为 4，c 为 2 等。6 个顶点，编号从 1 到 6。因为图中所有顶点地位是平等的，如无特殊要求可以按任意次序编号。这样做的原因可能有一下几点：首先，顶点元素值在不同问题中数据类型会不同，用编号代表顶点容易实现算法的通用性。其次，顶点的元素值可能会重复，比如图 6-13 中元素 b 就是重复的，使用编号可以解决重复问题。第三，图的很多算法中都会用到数组来保存顶点相

关信息，使用编号可以直接对应到数组的下标，一是可以随机访问，二是某些情况下还可以用来降低数组的维数，因为数组下标就可以代表顶点。假设有一个边数组 E[]，用邻接点来描述，存储图 6-13 中边，如果直接存储顶点值，比如边(a,b)，则数组元素就要 2 个分量。然而，用编号来存储用整型的一维数组就可以了，比如，边(a,b)可表示为 E[2]=1，边(b,c)可表示为 E[0]=2，类似，(c,b)为 E[1]=4，(b,d)为 E[3]=6，(d,a）为 E[5]=3 等。

有了编号后顶点元素可按编号顺序存储到一维数组中，需要用到元素值时，用编号获取即可。

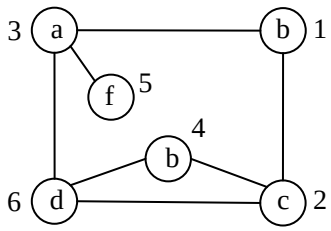


图 6-13 图的顶点编号示意图

6.2 图的存储结构

为了能在计算机上实现图结构的有关运算，首先要将图存储到计算机中。为此，要选择一种合适的存储结构以存储图的顶点的信息及相互的关系（即所有边或弧）。

已经提出的图的存储结构有好几种，其中用得最多的是邻接矩阵和邻接表这两种形式。下面讨论这两种结构。

6.2.1 邻接矩阵表示

邻接矩阵是表示图中顶点之间邻接关系的矩阵，即表示各顶点之间是否有边（弧）关系的矩阵。对有 n 个顶点的图来说，用 n×n 阶的邻接矩阵 A 表示，其中矩阵元素 A_{ij} 表示顶点 v_i 到 v_j 之间是否有边或弧。

1. 图的邻接矩阵表示

这里的图指无向图或有向图，不含网（带权图）。矩阵 A 中，若顶点 v_i 到 v_j 之间有边或弧连接，则 $A_{ij}=1$；否则 $A_{ij}=0$。即：

$$A_{ij}=\begin{cases}1 & (v_i,v_j)\in E \text{ 或 } <v_i,v_j>\in E\\ 0 & \text{其它}\end{cases}$$

例：图 6-1 G_1 的邻接矩阵如图 6-14(a)所示；图 6-2 G_2 的邻接矩阵如图 6-14(b)所示。

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array}\begin{array}{c} \begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

(a) 图 G_1 的邻接矩阵

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array}\begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

(b) 图 G_2 的邻接矩阵

图 6-14 图的邻接矩阵实例

从邻接矩阵我们可以得出如下一些结论：

无向图的邻接矩阵是对称的；第 i 行或 i 列“1”的个数就是顶点 v_i 的度；**图的边数=矩阵中“1”的个数/2**；

有向图因为边的方向性，邻接矩阵不一定对称；第 i 行“1”的个数是顶点 v_i 的出度，第 i 列“1”的个数是顶点 v_i 的入度；**图的边数=“1”的个数**。

2. 网的邻接矩阵表示

网中每个边上都带有权值，邻接矩阵简单的用“1”和“0”来表示就不能描述权值。所以我们要对图的邻接矩阵进行改造。因为网的每个边都有权值，我们就用这个权值作为邻接矩阵的元素；如果两个顶点之间没有边或弧，我们用无穷大来代替（也可以用 0 来代替，视具体使用情况而定）。设顶点 v_i 和 v_j 之间有边（弧），权值为 w_{ij}，则邻接矩阵元素 A_{ij} 为：

$$A_{ij}=\begin{cases} w_{ij} & (v_i,v_j)\in E \text{ 或 } <v_i,v_j>\in E \\ \infty & \text{其它} \end{cases}$$

例：图 6-3 所示网 G3 的邻接矩阵如图 6-15。

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array}\begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} \infty & 6 & 3 & 9 \\ 6 & \infty & \infty & 5 \\ 3 & \infty & \infty & 7 \\ 9 & 5 & 7 & \infty \end{pmatrix} \end{array}$$

图 6-15 网的邻接矩阵实例

网的邻接矩阵与图的邻接矩阵有类似的特性。无向网的邻接矩阵是对称的；从中数出有效元素个数（非∞），再除 2 即网的边数。有向网的邻接矩阵不一定对称；输出有效元素个数（非∞）即网的边数。

在实际实现时“∞”可以用计算机能接受的一个很大数来表示，只要能区分出有效的权值即可。

3.邻接矩阵存储结构描述

【邻接矩阵存储结构描述】

```
#define INF 65535                //定义无穷大，也可以是其它很大的数字
#define MaxVerNum   1000         //定义最大顶点个数，可根据需要定义最大顶点数
typedef char elementType;        //定义图中顶点的数据类型，这里不妨设为 char 类型
typedef int cellType;            //定义邻接矩阵中元素的数据类型，这里不妨设为 int 型
                                 //对无权图，1-相邻（有边），0-不相邻（无边）
                                 //对有权图，为边的权值，无边为无穷大。
typedef enum{UDG, UDN, DG, DN} GraphKind;
                                 //枚举图的类型--无向图，无向网，有向图，有向网
//*******************************************************//
//*  定义邻接矩阵表示的图结构。5 个分量组成：          *//
//*       data[]数组存储图中顶点数据元素                *//
//*       AdjMatrix[][]邻接矩阵                         *//
//*       VerNum 图中顶点个数                           *//
//*       ArcNum 图中边（弧）条数                       *//
//*       gKind 枚举图的类型                            *//
//*  考虑到名称的统一性，图类型名称定义为 Graph         *//
//*******************************************************//
typedef struct GraphAdjMatrix
{
        elementType Data[MaxVerNum];          //顶点数组，存放顶点元素的值
        cellType AdjMatrix[MaxVerNum][MaxVerNum];
                                              //邻接矩阵，元素类型为 cellType
        int VerNum;          //顶点数
        int ArcNum;          //弧（边）数
        GraphKind gKind;     //图的类型:0-无向图；1-无向网；2-有向图；3-有向网
                             //此项用以区分图的类型，为可选分量，可以取消。
                             //此项也可以直接定义为整型，而不用枚举定义。
} Graph;   //图的类型名
```

图的邻接矩阵表示的优点：非常直观，并且容易实现，编写算法也较简便，因而应用较广；根据矩阵元素 A_{ij}=1 或 0，便于判定两个顶点之间是否有边（弧）相连；计算顶点的度数，或有向图的入度、出度方便；计算图的边数算法简单等。

图的邻接矩阵表示的缺点：邻接矩阵事实上是一种顺序存储结构，具有顺序结构共有的缺点，比如：只能按最大空间需求申请内存空间、插入和删除顶点复杂等；空间复杂度高，n 个顶点的图，存储邻接矩阵需要 n^2 个单元，如果一个图的顶点数较多，但边（弧）数较少的话，邻接矩阵一样需要 n^2 个存储单元，就太浪费存储空间；统计图的边数算法虽然简单，用双重循环统计“1”的个数即可，但其时间复杂度为 $O(n^2)$。

因此，需要讨论另外的存储形式，下面的邻接表就是一种解决方法。

6.2.2 邻接表表示

邻接表是图的一种链式存储结构，其基本思想是这样的：给图中每个顶点，v_i，建立一个链表，链表中的结点保存 v_i 的邻接点，假设 v_i 到 v_j 有一条边（弧），那么就把 v_j 作为结点加入到链表中，这个链表描述了顶点 v_i 关联的边的信息，称为**边链表**；除了要描述每个顶点关联边的信息，还要描述整个图的顶点信息，我们用另一个表来存储图中所有顶点，但光存储顶点信息还不够，还需要把这个顶点与对应的边链表关联起来，不妨把这个表叫做**顶点表**。下面详细讨论顶点表和边链表的构成。

（1）顶点表

顶点表存储图中所有顶点信息，可以用顺序表（数组）存储也可以用链表存储，为了方便找到当前顶点对应的边链表，我们在每个顶点上附加一个指针，指向对应的边链表，即对应边链表的头指针。这样由顶点元素和边链表头指针构成顶点表的结点结构，构成如下：

图 6-16 顶点表结点结构

本书后面的描述中，顶点表采用数组（顺序表）实现。

（2）边链表

边链表存储顶点表中某个顶点关联边的信息，链表中结点可有三个域或两个域构成，为了通用性我们用三个域的结点结构，如图 6-17 所示：

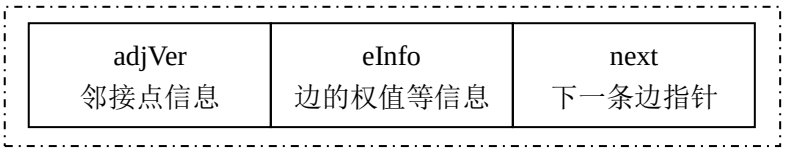


图 6-17 边链表结点结构

结点结构由两个数据域和一个指针域组成：

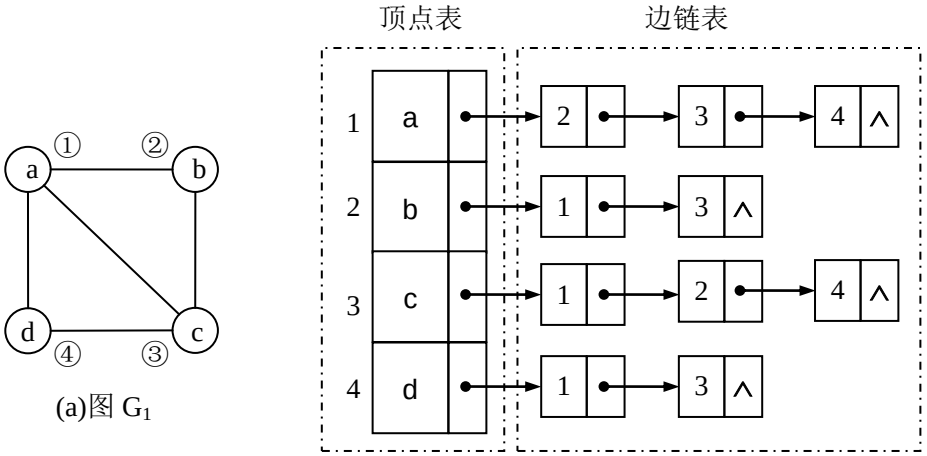
adjVer 域：存放邻接点信息，既可以直接存放邻接顶点的元素，也可以存放邻接点在顶点表中的编号。本书后面的邻接表存储结构描述，存储的就是邻接点在顶点表中的编号。

eInfo 域：这个域为可选项，对无向图和有向图这个域可以不要。但对网就必须要这个域，可以用来保存边的权值。

next 域：指向链表中下一个结点，对图来说即指向下一个邻接点，或下一条边。一个顶点关联边的次序没有规定，所以一条边链表中结点的位置是可以交换的。

下面举几个例子来说明邻接表的构成，以加深理解。

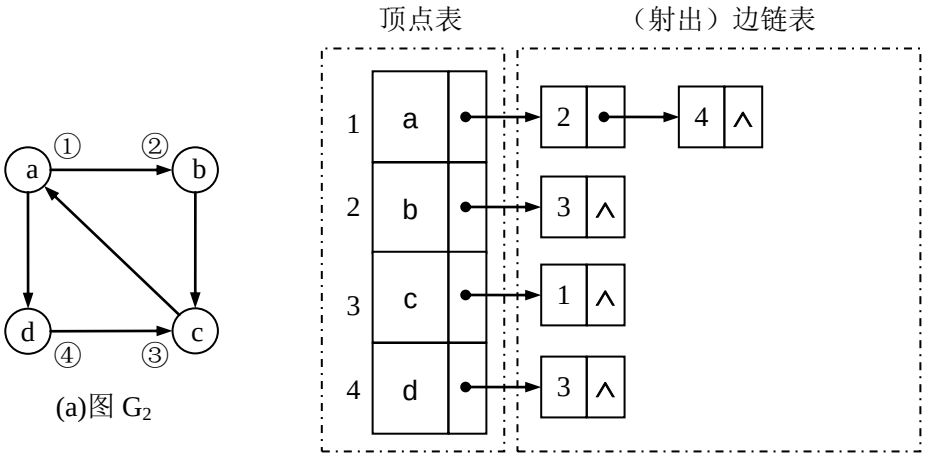
图 6-18(b)是图 6-18(a)所示无向图 G_1 的邻接表表示。在构建邻接表时一般先安排好顶点的次序；然后依次**给顶点编号**，如图 6-18(a)，顶点编号用加圆圈的数字表示，下同；然后按编号次序将顶点存储到顶点表，如图 6-18(b)；本题边链表结点没有 eInfo 域，只有一个邻接点信息域，存放的是顶点的编号。



(a)图 G_1　　(b)图 G_1 的邻接表

图 6-18 无向图邻接表实例

图 6-19 是一个有向图邻接链表的实例，边链表描述顶点关联的射出边信息。图 G_2 与图 6-18 中的图 G_1 顶点数相同，边数也相同，但有向图边链表的结点数只有无向图的一半。

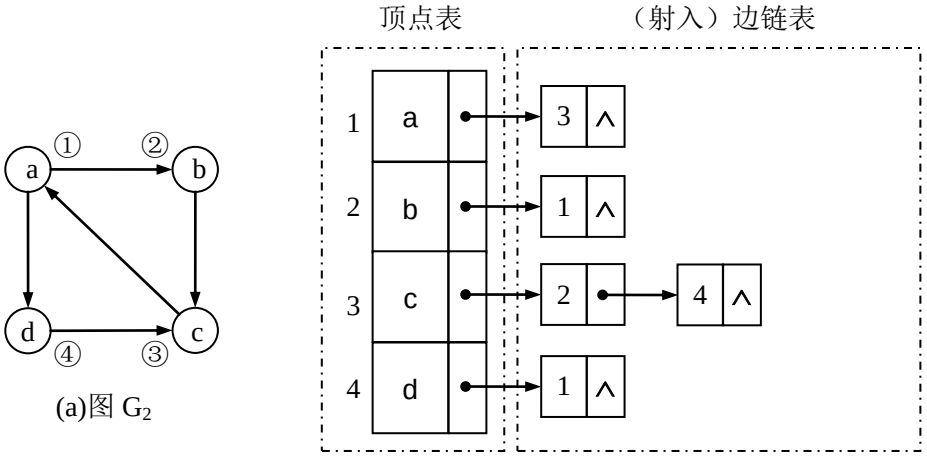


(a)图 G_2　　(b)图 G_2 的邻接表

图 6-19 有向图邻接表实例

逆邻接表：有向图的邻接表中，边链表描述的是一个顶点的射出边信息，如果我们把边链表改造为描述每个顶点**射入边**信息，那么这样的邻接表就叫做逆邻接表。

例图 6-20 中图 G**2** 的逆邻接表如图 6-20(b)所示。

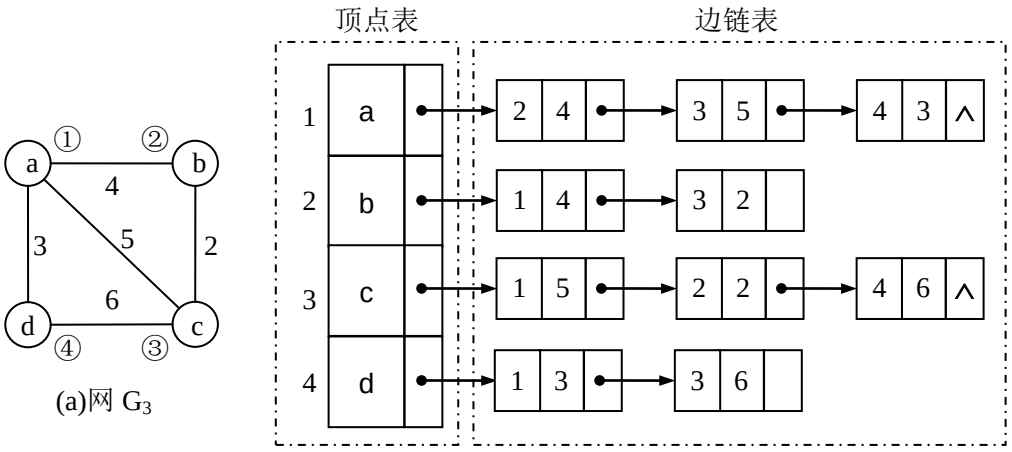


(a)图 G_2　　(b)图 G_2 的逆邻接表

图 6-20 有向图逆邻接表实例

有向图的邻接表和逆邻接表对算法求解的实现方面有差异。例如，如果要求一个顶点的出度，则在邻接表中和逆邻接表中的求解就有明显的不同：在邻接表中求出该顶点的邻接表的长度（结点个数）即可，而在逆邻接表中，需要搜索整个图的邻接表的各结点，因而其时间花费显然要多。反之，逆邻接表求顶点入度很方便，但求顶点的出度就比较麻烦。

图 6-21 所示为无向网 G3 的邻接表表示，这个实例中边链表结点就有 3 个域组成，其中 eInfo 域处于结点中间位置，存储边的权值。



(a)网 G_3　　(b)网 G_3 的邻接表

图 6-21 无向网邻接表实例

接下来我们来讨论邻接表的存储描述。这个结构相对复杂，但只要我们充分理解了上面几个实例，也不难想象出这个结构描述的核心内容。首先，我们需要一个结构体来描述边链表的结点结构，这个结构体应由 3 个分量构成，对应前面讨论的结点的三个域：adjVer、eInfo 和 next。且 eInfo 域对图可以省略，在网中才是必须的。其次，需要一个结构体来描述顶点表中结点的结构，这个结构应由 2 个分量构成，对应前面讨论的结点的两个域：data 和

firstEdge。最后需要一个结构体来描述图或网的整体结构。再加上一些辅助内容就可以给出邻接表结构描述了，描述如下：

【邻接表结构描述】

```
#define INF 65535                  //定义无穷大
#define MaxVerNum    1000          //定义最大顶点个数

typedef char elementType;          //定义图中顶点的数据类型
typedef int eInfoType;             //定义 eInfo 的数据类型，即权值的数据类型
                                   //枚举图的类型--无向图，无向网，有向图，有向网
typedef enum{UDG, UDN, DG, DN} GraphKind;
//---------------------------------------以上为辅助信息
typedef struct eNode    //定义边链表的结点结构
{
    int adjVer;              //邻接顶点信息，此处为顶点编号，从 1 开始
    eInfoType eInfo;         //边链表中表示边的相关信息，比如表的权值
    struct eNode* next;      //指向边链表中的下一个结点。
}EdgeNode;    //边链表结点类型

typedef struct vNode    //定义顶点表的结点结构
{
    elementType data;          //存放图中顶点的数据值
    EdgeNode* firstEdge;       //指向此顶点关联的第一条边的指针，即边链表的头指针
                               //注意：fristEdge 指针与边链表结点中的 next 指针类型相同
}VerNode;       //顶点表结点类型

typedef struct GraphAdjLinkList     //定义图的整体结构
{
    VerNode VerList[MaxVerNum];    //顶点表，此为数组（顺序表），存放顶点信息
                                   //数组的元素为 VerNode 结构类型
    int VerNum;        //顶点数
    int ArcNum;        //弧（边）数
    GraphKind gKind;   //图的类型:0-无向图；1-无向网；2-有向图；3-有向网
                       //此项用以区分图的类型，为可选分量，可以取消。
                       //此项也可以直接定义为整型，而不用枚举定义。
}Graph;    //图的类型名
```

从邻接表我们可以得出如下一些结论：

通过邻接表我们可以求出图的边数，对无向图和网我们计数所有边链表的结点数之和，除以 2 即是边数；对有向图和网通过邻接表或逆邻接表计数边链表结点之和即是边数。通过有向图的邻接表很容易求一个顶点的出度，计数对应边链表的结点个数即可；但求入度相对复杂；利用逆邻接表很容易求一个顶点的入度，但求出度相对复杂。

邻接表的优点：如果顶点表也采用链式结构存储，那么邻接表就可以动态申请内存，插入和删除顶点方便；便于求图的边数；顶点很多边很少的稀疏图空间效率较高，一个 n 个顶点，e 条边的图（网），如果是无向图（网），需要 n 个顶点表结点和 2e 个边链表结点，如果为有向图（网），需要 n 个顶点表结点和 e 个边链表结点。

邻接表的缺点：判断两个顶点之间是否有边（弧）相对复杂，比 v_i 和 v_j 之间是否有边，需要先在顶点表中找到 v_i 结点，根据其 firstEdge 指针找到对应的边链表，然后搜索 v_j 是否在此边链表上，在则有边（弧），不在则没有边（弧）；对有向图（网），邻接表求出度方便，但求入度麻烦，逆邻接表求入度方便，求出度麻烦。

6.2.3 图的创建和销毁

后面的内容中将介绍图的遍历等一些列算法和应用，如果要实现并体验这些算法，第一步就必须要学会创建图，如果不会创建图，其它算法的学习都只能停留在理论上。

图通常有较多顶点，如果是网的话还涉及到边的权值。如果用键盘交互式创建图，键盘输入繁琐，且极容易出错，浪费大量宝贵时间。如果图采用链式存储结构，创建图中途出错退出，还会造成内存泄漏。

这里介绍一种从文本文件读入图的数据创建图的方法，这样我们可以按照指定的格式，先从容地准备好数据，然后由程序自动读入数据来创建图。

1. 数据文件格式设计

这里数据用文本文件保存，文件扩展名可自行指定，比如 g8.grp，只要数据按文本文件格式读写即可。下面给出一种数据文件格式，其实读者可以自行设计图的数据文件格式。

① 标识行 1：Graph

标识这是一个图的数据文件，这一行也可以不要。

② 标识行 2：UDG、或 UDN、或 DG、或 DN

这一行用来标识此图是无向图(UDG)、无向网(UDN)、有向图(DG)、还是有向网(DN)。

③ 顶点行

这一行将图中所有顶点列出，顶点之间用空格进行分割。这些顶点数据读出后存放到图的顶点数组中。

例如，图 6-21(a)所示的图的顶点行数据为：a b c d。

图的各种算法都是用顶点的编号来引用顶点的，所以这一行顶点的排列顺序是很重要的，顶点的排列顺序决定了顶点的编号。比如上例中，顶点 a、b、c、d 对应的编号就为 1、2、3、4。

④ 边数据行

一条边一行，边的 2 个顶点之间用空格分割。如果是网，每一行再加边的权值，也以空格分割。如果是无向图和无向网，每条边会重复一次。

例如图 6-18(a)无向图的边的数据为：

```
a b
a c
a d
b a
```

b c
c a
c b
c d
d a
d c

图 6-21(a)无向网边的数据为：

a b 4
a c 5
a d 3
b a 4
b c 2
c a 5
c b 2
c d 6
d a 3
d c 6

⑤ 其它行

如果程序强大一点，还可以在文件中加注释行，允许出现空行等，当然这是非必须的。

举一个完整的图的数据文件的例子，对图 6-18(a)的无向图，完整的数据文件如下：

//文件可以加注释行，注释以“//”开始
//Graph 为图标志，否则判定格式不对
//标志行后，第一行为图的类型。UDG--无向图；UDN--无向网；DG--有向图；DN--有向网
//标志行后，第二行为顶点元素
//顶点行以下图的边或弧。用顶点表示，第一列为起始顶点；第二列为邻接点；在网中再增加一列表示权值。
//本图具有 4 个顶点 5 条边

//下一行为图的标识行
Graph
//图的类型标识，此为无向图
UDG
//顶点元素数据
a b c d
//以下为边的数据，共 10 行数据，表示 5 条边
a b
a c
a d
b a

b c
c a
c b
c d
d a
d c

文件名不妨叫做 Gudg4.grp。

再举一个有向网的例子，对图 6-22 所示的有向网，完整的数据文件如下：

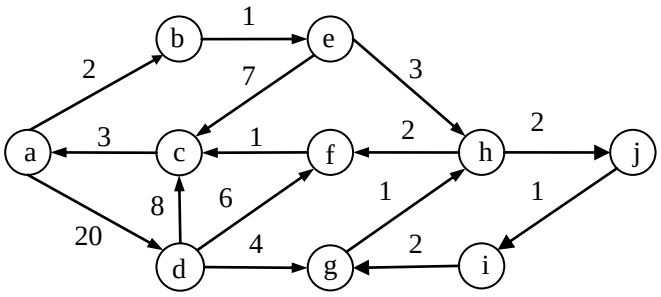


图 6-22 一个有向网实例

//标识为图数据

Graph

//标识有向网

DN

//顶点数据

a b c d e f g h i j

//以下为边数据，共 15 条边

a b 2

a d 20

b e 1

c a 3

d c 8

d f 6

d g 4

e c 7

e h 3

f c 1

g h 1

h f 2

h j 2

i g 2

j i 1

不妨设文件名为 Gdn10.grp

2. 从数据文件创建邻接矩阵表示的图

指定图的数据文件名，然后逐行读出数据并处理，自动创建邻接矩阵表示的图。本程序可以自动处理注释行和空行，程序实现如下：

```
//*****************************文件创建图*****************************//
//* 函数功能：从文本文件创建邻接矩阵表示的图                          *//
//* 入口参数  char fileName[]，文件名                                 *//
//* 出口参数：Graph &G，即创建的图                                    *//
//* 返 回 值：bool，true 创建成功；false 创建失败                     *//
//* 函 数 名：CreateGraphFromFile(char fileName[], Graph &G)          *//
//*******************************************************************//
int CreateGraphFromFile(char fileName[], Graph &G)
{
    FILE* pFile;              //定义文件指针
    char str[1000];           //存放读出一行文本的字符串
    char strTemp[10];         //判断是否注释行
    cellType eWeight;         //边的信息，常为边的权值
    GraphKind graphType;      //图类型枚举变量
    pFile=fopen(fileName,"r");
    if(!pFile)
    {
        printf("错误：文件%s 打开失败。\n",fileName);
        return false;
    }
    while(fgets(str,1000,pFile)!=NULL)
    {
        strLTrim(str);        //删除字符串左边空格，这是一个自定义的函数
        if (str[0]=='\n')     //空行，继续读取下一行
            continue;
        strncpy(strTemp,str,2);
        if(strstr(strTemp,"//")!=NULL)   //跳过注释行
            continue;
        else                              //非注释行、非空行，跳出循环
            break;
    }
    //循环结束，str 中应该已经是图的标识 Graph，判断标识是否正确
    if(strstr(str,"Graph")==NULL)
    {
        printf("错误：打开的文件格式错误！\n");
        fclose(pFile); //关闭文件
        return false;
```

```
}
//读取图的类型，跳过空行
while(fgets(str,1000,pFile)!=NULL)
{
    strLTrim(str);      //删除字符串左边空格，这是一个自定义函数
    if (str[0]=='\n')    //空行，继续读取下一行
        continue;
    strncpy(strTemp,str,2);
    if(strstr(strTemp,"//")!=NULL)   //注释行，跳过，继续读取下一行
        continue;
    else                              //非空行，也非注释行，即图的类型标识
        break;
}
//设置图的类型
if(strstr(str,"UDG"))
    graphType=UDG;      //无向图
else if(strstr(str,"UDN"))
    graphType=UDN;      //无向网
else if(strstr(str,"DG"))
    graphType=DG;       //有向图
else if(strstr(str,"DN"))
    graphType=DN;       //有向网
else
{
    printf("错误：读取图的类型标记失败！\n");
    fclose(pFile);          //关闭文件
    return false;
}
//读取顶点行数据到 str。跳过空行
while(fgets(str,1000,pFile)!=NULL)
{
    strLTrim(str);         //删除字符串左边空格，这是一个自定义函数
    if (str[0]=='\n')       //空行，继续读取下一行
        continue;
    strncpy(strTemp,str,2);
    if(strstr(strTemp,"//")!=NULL)   //注释行，跳过，继续读取下一行
        continue;
    else                              //非空行，也非注释行，即图的顶点元素行
        break;
}
```

```
   //顶点数据放入图的顶点数组
char* token=strtok(str," ");
int nNum=0;
while(token!=NULL)
{
      G.Data[nNum]=*token;
      token = strtok( NULL, " ");
      nNum++;
}
//图的邻接矩阵初始化
int nRow=0;       //矩阵行下标
int nCol=0;       //矩阵列下标
if(graphType==UDG || graphType==DG)
{
      for(nRow=0;nRow<nNum;nRow++)
            for(nCol=0;nCol<nNum;nCol++)
                  G.AdjMatrix[nRow][nCol]=0;
}
else
{
      for(nRow=0;nRow<nNum;nRow++)
            for(nCol=0;nCol<nNum;nCol++)
                  G.AdjMatrix[nRow][nCol]=INF;   //INF 表示无穷大
}
//循环读取边的数据到邻接矩阵
int edgeNum=0;              //边的数量
elementType Nf, Ns;         //边或弧的 2 个相邻顶点
while(fgets(str,1000,pFile)!=NULL)
{
      strLTrim(str);          //删除字符串左边空格，这是一个自定义函数
      if (str[0]=='\n')       //空行，继续读取下一行
            continue;
      strncpy(strTemp,str,2);
      if(strstr(strTemp,"//")!=NULL)   //注释行，跳过，继续读取下一行
            continue;
      char* token=strtok(str," ");   //以空格为分隔符，分割一行数据，写入邻接矩阵
      if(token==NULL)             //分割为空串，失败退出
      {
            printf("错误：读取图的边数据失败！\n");
            fclose(pFile);            //关闭文件
            return false;
```

```
    }
    Nf=*token;                          //获取边的第一个顶点
    token = strtok( NULL, " ");     //读取下一个子串，即第二个顶点
    if(token==NULL)                 //分割为空串，失败退出
    {
        printf("错误：读取图的边数据失败！\n");
        fclose(pFile);              //关闭文件
        return false;
    }
    Ns=*token;   //获取边的第二个顶点
                 //从第一个顶点获取行号
    for(nRow=0;nRow<nNum;nRow++)
    {
        if(G.Data[nRow]==Nf)   //从顶点列表找到第一个顶点的编号
            break;
    }
                 //从第二个顶点获取列号
    for(nCol=0;nCol<nNum;nCol++)
    {
        if(G.Data[nCol]==Ns)   //从顶点列表找到第二个顶点的编号
            break;
    }
    //如果为网，读取权值
    if(graphType==UDN || graphType==DN)
    {                       //读取下一个子串，即边的附加信息，常为边的权重
        token = strtok( NULL, " ");
        if(token==NULL)      //分割为空串，失败退出
        {
            printf("错误：读取图的边数据失败！\n");
            fclose(pFile);      //关闭文件
            return false;
        }
        eWeight=atoi(token);   //取得边的附加信息
    }
    if(graphType==UDN || graphType==DN)
        G.AdjMatrix[nRow][nCol]=eWeight;
            //如果为网，邻接矩阵中对应的边设置权值，否则置为 1
    else
        G.AdjMatrix[nRow][nCol]=1;
    edgeNum++;     //边数加 1
}
```

```
    G.VerNum=nNum;                  //图的顶点数
    if(graphType==UDG || graphType==UDN)
        G.ArcNum=edgeNum / 2;   //无向图或网的边数等于统计的数字除 2
    else
        G.ArcNum=edgeNum;
    G.gKind=graphType;              //图的类型
    fclose(pFile);                  //关闭文件
    return true;
  }
```

3. 从数据文件创建邻接表表示的图

程序实现如下：

```
//*****************************文件创建图*****************************//
//* 函数功能：从文本文件创建邻接表表示的图                              *//
//* 入口参数  char fileName[]，文件名                                  *//
//* 出口参数：Graph &G，即创建的图                                     *//
//* 返 回 值：bool，true 创建成功；false 创建失败                      *//
//* 函 数 名：CreateGraphFromFile(char fileName[], Graph &G)           *//
//* 备注：本函数使用的数据文件格式以边（顶点对）为基本数据             *//
//*************************************************************************//
int CreateGraphFromFile(char fileName[], Graph &G)
{
    FILE* pFile;              //定义文件指针
    char str[1000];           //存放读出一行文本的字符串
    char strTemp[10];         //判断是否注释行
    char* ss;
    int i=0, j=0;
    int edgeNum=0;            //边的数量
    eInfoType eWeight;        //边的信息，常为边的权值
    GraphKind graphType;   //图类型枚举变量
    pFile=fopen(fileName,"r");
    if(!pFile)
    {
        printf("错误：文件%s 打开失败。\n",fileName);
        return false;
    }
        while(fgets(str,1000,pFile)!=NULL)   //跳过空行和注释行
    {
        strLTrim(str);        //删除字符串左边空格，这是一个自定义函数
        if (str[0]=='\n')     //空行，继续读取下一行
            continue;
```

```
        strncpy(strTemp,str,2);
        if(strstr(strTemp,"//")!=NULL)       //跳过注释行
            continue;
        else                                 //非注释行、非空行，跳出循环
            break;
}
//循环结束，str 中应该已经是图的标识 Graph，判断标识是否正确
if(strstr(str,"Graph")==NULL)
{
        printf("错误：打开的文件格式错误！\n");
        fclose(pFile);                       //关闭文件
        return false;
}
//读取图的类型，跳过空行及注释行
while(fgets(str,1000,pFile)!=NULL)
{
        strLTrim(str);          //删除字符串左边空格，这是一个自定义函数
        if (str[0]=='\n')       //空行，继续读取下一行
            continue;
        strncpy(strTemp,str,2);
        if(strstr(strTemp,"//")!=NULL)   //注释行，跳过，继续读取下一行
            continue;
        else                             //非空行，也非注释行，即图的类型标识
            break;
}
//设置图的类型
if(strstr(str,"UDG"))
        graphType=UDG;       //无向图
else if(strstr(str,"UDN"))
        graphType=UDN;       //无向网
else if(strstr(str,"DG"))
        graphType=DG;        //有向图
else if(strstr(str,"DN"))
        graphType=DN;        //有向网
else
{
        printf("错误：读取图的类型标记失败！\n");
        fclose(pFile);           //关闭文件
        return false;
}
//读取顶点数据到 str。跳过空行
```

```
while(fgets(str,1000,pFile)!=NULL)
{
    strLTrim(str);        //删除字符串左边空格，这是一个自定义函数
    if (str[0]=='\n')     //空行，继续读取下一行
        continue;
    strncpy(strTemp,str,2);
    if(strstr(strTemp,"//")!=NULL)   //注释行，跳过，继续读取下一行
        continue;
    else                            //非空行，也非注释行，即图的顶点元素行
        break;
}
//顶点数据放入图的顶点数组
char* token=strtok(str," ");
int nNum=0;
while(token!=NULL)
{
    G.VerList[nNum].data=*token;
    G.VerList[nNum].firstEdge=NULL;
    token = strtok( NULL, " ");
    nNum++;
}
//循环读取边（顶点对）数据
int nRow=0;                 //矩阵行下标
int nCol=0;                 //矩阵列下标
EdgeNode* eR;               //边链表尾指针
EdgeNode* p;
elementType Nf, Ns;     //边或弧的 2 个相邻顶点
while(fgets(str,1000,pFile)!=NULL)
{
    strLTrim(str);        //删除字符串左边空格，这是一个自定义函数
    if (str[0]=='\n')     //空行，继续读取下一行
        continue;
    strncpy(strTemp,str,2);
    if(strstr(strTemp,"//")!=NULL)   //注释行，跳过，继续读取下一行
        continue;
    char* token=strtok(str," ");        //以空格为分隔符，分割一行数据
    if(token==NULL)                     //分割为空串，失败退出
    {
        printf("错误：读取图的边数据失败！\n");
        fclose(pFile);                  //关闭文件
        return false;
```

```
}
Nf=*token;                          //获取边的第一个顶点
token = strtok( NULL, " ");     //读取下一个子串，即第二个顶点
if(token==NULL)                 //分割为空串，失败退出
{
    printf("错误：读取图的边数据失败！\n");
    fclose(pFile);              //关闭文件
    return false;
}
Ns=*token;      //获取边的第二个顶点
                //从第一个顶点获取行号
for(nRow=0;nRow<nNum;nRow++)
{
    if(G.VerList[nRow].data==Nf)   //从顶点列表找到第一个顶点的编号
        break;
}
                //从第二个顶点获取列号
for(nCol=0;nCol<nNum;nCol++)
{
    if(G.VerList[nCol].data==Ns)   //从顶点列表找到第二个顶点的编号
        break;
}
//如果为网，读取权值
if(graphType==UDN || graphType==DN)
{               //读取下一个子串，即边的附加信息，常为边的权重
    token = strtok( NULL, " ");
    if(token==NULL)      //分割为空串，失败退出
    {
        printf("错误：读取图的边数据失败！\n");
        fclose(pFile);      //关闭文件
        return false;
    }
    eWeight=atoi(token);   //取得边的附加信息，即权值
}
eR=G.VerList[nRow].firstEdge;
while(eR!=NULL && eR->next!=NULL)
{
    eR=eR->next;            //后移边链表指针，直至尾节点
}
p=new EdgeNode;             //申请一个边链表结点
p->adjVer=nCol+1;           //顶点的编号，从 1 开始
```

```
            //边的附加信息（权值），对有权图保存权值，无权图为 1
            if(graphType==UDN || graphType==DN)
                p->eInfo=eWeight;
            else
                p->eInfo=1;
            p->next=NULL;
            if(G.VerList[nRow].firstEdge==NULL)
            {
                G.VerList[nRow].firstEdge=p;
                eR=p;
            }
            else
            {
                eR->next=p;
                eR=p;           //新的尾指针
            }
            edgeNum++;        //边数加 1
        }
        G.VerNum=nNum;      //图的顶点数
        if(graphType==UDG || graphType==UDN)
            G.ArcNum=edgeNum / 2;     //无向图或网的边数等于统计的数字除 2
        else
            G.ArcNum=edgeNum;
        G.gKind=graphType;                //图的类型
        fclose(pFile);                    //关闭文件
        return true;
    }
```

4. 图的销毁

以邻接矩阵为存储结构的图，因为使用矩阵存储图的数据，不存在销毁（释放内存）问题。但是以邻接表为存储结构的图，由于在创建图的过程中使用 malloc()函数或 new 操作符动态申请了内存，当这个图不再需要时，必须手工释放动态申请的内存，否则造成内存泄漏。下面给出一个销毁邻接表表示的图的程序。

```
    void DestroyGraph(Graph &G)
    {
        EdgeNode *p,*u;             //边链表结点指针
        int vID;
        for(vID=1; vID<=G.VerNum; vID++)   //循环删除每个顶点的边链表
        {
            p=G.VerList[vID-1].firstEdge;
            G.VerList[vID-1].firstEdge=NULL;
            while(p)                //循环删除当前顶点所有的关联边
```

```
            {
                u=p->next;      //u 指向下一个边结点
                delete(p);      //删除当前边结点
                p=u;
            }
        }
        p=NULL;
        u=NULL;
        G.VerNum=-1;   //标识图已经销毁
    }
```

6.3 图的遍历算法及其应用

与树的遍历类似，所谓图的遍历就是按照某种次序访问图中每个结点一次且仅一次。显然图的遍历要比树的遍历复杂，首先图中没有标志性的根结点，图中所有顶点地位平等，我们可以选择任一顶点开始遍历；其次，图中可能存在回路，访问过的顶点可能被重复访问，不加处理的话，遍历可能会在这个回路上反复“绕圈子”，导致算法死循环，所以我们要设法标记已经访问过的顶点，以免多次重复访问。

对图的遍历有两种基本的方法，即**深度优先搜索遍历（简称深度遍历）**和**广度优先搜索遍历（简称广度遍历）**，这是许多图算法的基础。这两种算法也是经典算法，许多问题的求解可以转化为这两种算法及其变形形式。因此，理解这两种算法有助于后续课程的学习。但是，这两个算法有一定的难度。为此，下面分层次介绍有关内容。

6.3.1 深度优先搜索遍历算法及其应用

深度优先搜索遍历（Depth_First Search traverse -- DFS）有点类似树的先序遍历，可看成是树的先序遍历的推广。

1. 基本深度遍历算法描述

假定图 G 是连通的，选定从顶点 v_0 出发深度优先搜索遍历算法 DFS(v_0)描述如下：

（1）访问 v_0；

（2）依次从 v_0 的各个未被访问的邻接点出发执行深度遍历（DFS）。

虽然只有短短的两句话，但却将遍历过程描述得清清楚楚，这个算法的描述是递归的。所谓“基本”是指这个算法只能遍历连通图，或一般图的一个连通分量。

下面以对图 6-23 所示的图 G_9 的遍历过程来说明其遍历的执行过程。设从顶点 v_0 出发深度优先搜索记为 DFS(v_0)。假定图中顶点已经编号，我们直接用编号代表顶点，并假定顶点 v_0 的编号为 1，如图 6-23 所示。则 DFS(v_0) 即 DFS(1)，下面讨论 DFS(1)的执行过程。为描述清晰起见，用实箭头表示其搜索过程，用虚箭头表示其返回（回溯）过程。

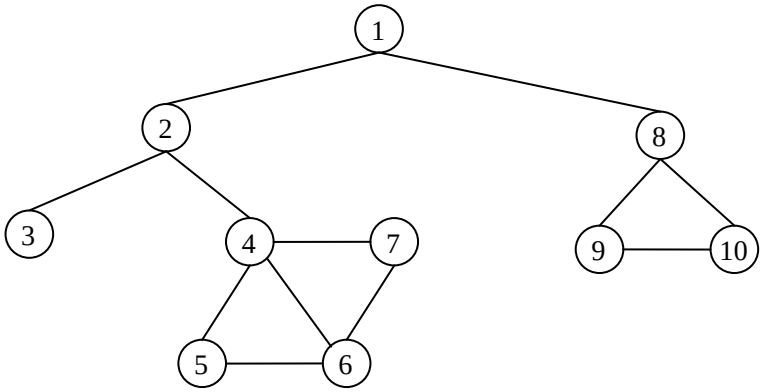


图 6-23 图 G_9

按算法描述，首先执行 DFS(1)包含如下两部分操作：

（1）访问顶点 1；

（2）依次从顶点 1 的未被访问的邻接点（2 和 8）出发执行深度遍历，即要依次执行 DFS(2)和 DFS(8)。首先执行 DFS（2）（假设顶点 2 作为顶点 1 的第一个邻接点）。

为清晰起见，在执行 DFS(v)时，在图上的顶点 v 附近标出 DFS(v)，在从顶点 v 转向其邻接点 w 执行 DFS(w)时，则在其边（弧）旁标记一个实箭头以示其执行线路，例如，图中顶点 1 到顶点 2 之间的边旁便有一条实箭头。（如图 6-24 所示）。

在执行 DFS(2)时，也同样包括两部分执行：

（1）访问顶点 2；

（2）依次从顶点 2 的未被访问的邻接点（有 3 和 4 这两个顶点，另一个邻接点 1 已经被访问）出发深度遍历，即要依次执行 DFS(3)和 DFS(4)。首先执行的是 DFS(3)。

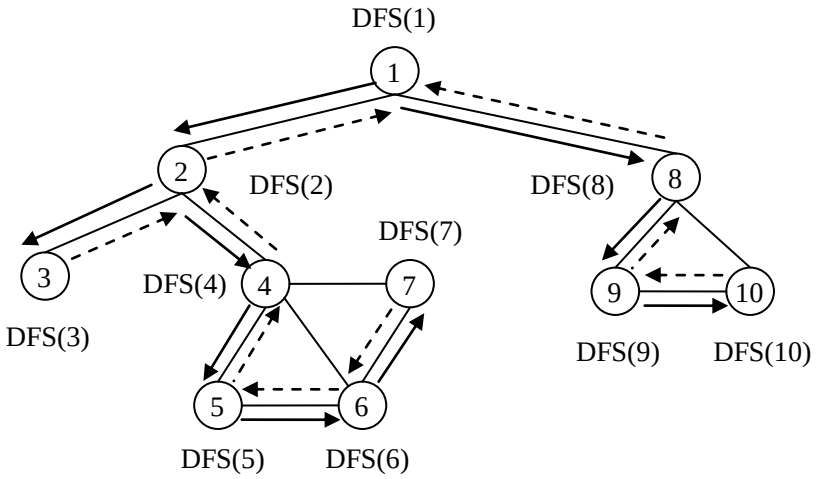


图 6-24 对图 G_9 的 DFS(1)的执行过程示意图

在执行 DFS（3）时，也同样包括两部分执行：

（1） 访问顶点 3；

（2） 依次从顶点 3 的未被访问的邻接点出发进行深度遍历。但由于现在已经不存在这样的顶点了，故 DFS（3）的执行到此结束。因此，应返回到调用层，即返回到

DFS(2)（用从 3 到 2 的虚箭头表示这一返回过程），以执行其中未完成的操作，即执行 DFS(4)（同样用从 2 到 4 的实箭头表示其调用过程）。

以下操作类似，下面简要叙述：

执行 DFS(4)，包括访问顶点 4 和从其邻接点 5（不妨设顶点 5 作为其第一个邻接点）出发深度遍历，即执行 DFS(5)；

执行 DFS(5)，包括访问顶点 5 和执行 DFS(6)；

执行 DFS(6)，包括访问顶点 6 和执行 DFS(7)；

执行 DFS(7)时，由于其邻接点全部都已被访问，因而其操作仅有访问顶点 7 这一项。然后返回到调用层，即返回到 DFS(6)中；DFS(6)由于也没有其它邻接点可访问，故也返回到 DFS(5)，类似地由 DFS(5)返回到 DFS(4)，从 DFS(4)返回到 DFS(2)，再返回到 DFS(1)中。此时，对 DFS(1)来说，由于还有顶点 1 的另一个邻接点 8 未被访问，故要执行 DFS(8)。因此，余下操作依次如下：

访问顶点 8；执行 DFS(9)。

执行 DFS（9），包括访问顶点 9 和执行 DFS(10)；

执行 DFS(10)，仅有访问顶点 10 这一个操作。然后返回到 DFS(9)中，再返回到 DFS(8)中，最后返回到 DFS(1)中。到此时，DFS(1)的执行过程结束。

整个执行过程如图 6-24 所示。所得到的**顶点访问序列**为 1,2,3,4,5,6,7,8,9,10。

深度遍历生成树（简称 DFS 生成树）：

对连通图或强连通图，如果将 DFS 算法访问过程中搜索顶点的箭头所对应的边（弧）连起来，则可构成一棵树称这棵树为**深度遍历生成树**（简称 DFS 生成树）。如果图是非连通，则会得到一个生成森林，每个连通分量对应一棵生成树。

由图 6-24 所描述的 DFS(1)生成树如图 6-25 所示。图的深度遍历得到的顶点访问序列对应 DFS 生成树的先序遍历序列，本例中皆为：1,2,3,4,5,6,7,8,9,10，即生成树的先序遍历反映了对原图的深度优先搜索遍历过程。

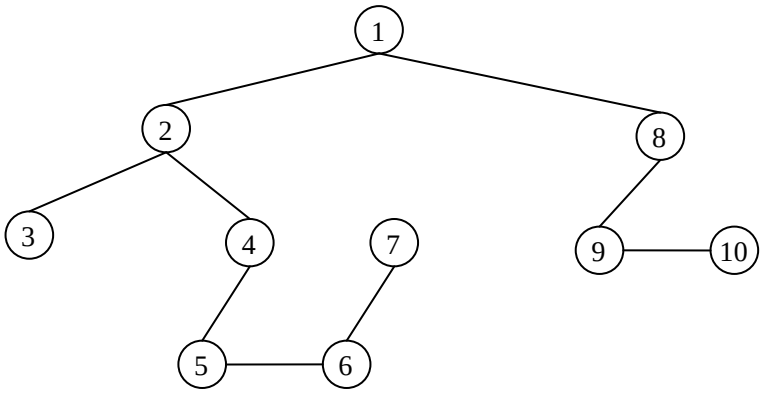


图 6-25 G_9 的 DFS(1)生成树

2. 深度优先搜索遍历算法

下面先讨论前述基本 DFS 算法的实现，再基于 DFS 算法讨论一般图的深度遍历算法，即对连通和非连通图都适用的算法。

（1）DFS 算法

由 DFS(v)的描述可知，算法需要分别实现以下内容：

①访问顶点的实现：大多场合是输出顶点的值，也可根据具体问题的需要来设计。

②DFS(v)的第二句描述为“依次从顶点 v 的未被访问的邻接点出发进行深度遍历”。这涉及到以下内容：

（a）顶点是否被访问的标识：为此，为每个顶点设置一个访问标志（设置为布尔型），未访问时，其值为 FALSE，被访问后设置为 TRUE，将各顶点的标志合在一起形成数组 visited[n+1]，后面代码中皆假定 visited[0]单元不用，使顶点编号与数组下标一致。

（b）顶点 v 的各邻接点的求解：很显然，一个顶点的邻接点的求解取决于图的存储结构。例如，在用邻接矩阵存储图时，顶点 v_i 的各邻接点在邻接矩阵的第 i 行中，因此可通过在该行中依次搜索非 0 元素（或非∞元素）来搜索所有邻接点；在用邻接表存储时，该顶点的邻接点全部在邻接表的第 i 个链表中，因此可通过依次取该链表中结点来实现所有邻接点的求解。为使算法不受图的具体存储结构的影响，同时也为算法更清晰，下面的讨论更倾向于用如下两个不依赖于特定存储结构的邻接点函数来实现这一求解：

firstAdj(G, v)：返回图 G 中顶点 v 的第一个邻接点。若不存在邻接点（编号），则返回 0；

nextAdj(G, v, w)：返回图 G 中顶点 v 的邻接点中处于 w 之后的那个邻接点。若不存在这样的邻接点（编号），则返回 0；

通过运用这两个函数，可依次求出一个顶点的所有邻接点。

（c）从邻接点出发深度遍历的实现：可通过调用 DFS 算法来实现，也就是说，DFS 算法是一个递归算法。

【DFS 算法描述】

```
void   DFS（graph G，int v）     //从编号为 v 的顶点出发对图 G 进行深度优先搜索遍历
{
      int w;
      visit(G,v);   visited[v]=TRUE;   //访问顶点 v，并设置其访问标志
       w=firstAdj(G,v);                //求出 v 的第一个邻接点，返回邻接点编号给 w
       while (w!=0)                    //当还存在邻接点时
       {
          if (visited[w]==FALSE)       //从没有访问过的邻接点出发继续深度遍历
                DFS(G, w);
          w=nextAdj(G,v,w);            //取下一个邻接点
      }
}
```

（2）一般图的（通用）深度优先搜索遍历算法

前面已经说过 DFS(G, v)只能从指定顶点 v 出发遍历连通图，或一个连通分量。如果图 G 是非连通的，只能遍历顶点 v 所在的连通分量，那么图 G 中就会剩下一些顶点未被访问。解决办法是 DFS(G,v)执行结束后，在未被访问的顶点中选一个再次执行 DFS，反复执行这个过程，直到所有顶点都被访问。上述过程每执行一次，就遍历一个连通分量。现在的问题是：需要选择多少次起点？显然，这取决于具体的图，有多少个连通分量就会选择多少次，因而只能在算法中通过条件加以判断来实现。显然，在启动遍历算法之前应初始化各顶点的访问标志为 FALSE。综上讨论，可得遍历图的完整算法如下：

【通用深度遍历算法描述】

```
void    DFSTraverse (graph G )
{
    int    i;
    for (i=1; i<=n; i++)
        visited[i]=FALSE;                    //初始化各顶点的访问标志为 FALSE
    for (i=1; i<=n; i++)
        if (visited[i]==FALSE)               //循环选择未被访问的顶点 i，调用 DFS
            DFS(G,i);                        //每次循环遍历一个连通分量
}
```

这个算法对连通图（网）、非连通图（网）都是适用的，如果 G 是连通图在第一次调用 DFS(G, 1)时就访问了全部顶点，不会有第二次调用，所示可成为通用深度遍历算法。

3. 深度遍历算法的参考实现代码

上面给出的 DFS(G, v)算法中 firstAdj()和 nextAdj()两个函数没有给出具体实现，读者上机实验时需要自己完成这个工作。下面分别针对图的邻接矩阵表示和邻接表表示给出 DFS 算法的两种参考实现代码，供您上机时参考，在这两个实现代码中把 fristAdj()和 nextAdj()函数的功能直接在 DFS 算法中实现，取消了这两个函数，请读者阅读时注意。

（1）基于邻接矩阵的 DFS 实现

【基于邻接矩阵的 DFS 的一种实现代码】

```
//*************** 连通图或一个连通分量的 DFS ****************//
//* 函数功能：深度优先遍历连通图，或一个连通分量                *//
//* 入口参数  Graph G，待访问的图；int verID 起始顶点编号     *//
//* 出口参数：无                                              *//
//* 返 回 值：无                                              *//
//* 函 数 名：DFS(Graph &G, int verID)                        *//
//************************************************************//
void DFS(Graph &G, int verID)
{
    cout<<G.Data[verID-1]<<"\t";   //访问编号为 verID 的顶点，相当于 visit(G, verID);
    visited[verID]=TRUE;           //标记编号为 verID 的顶点已经访问
    for(int w=0;w<G.VerNum;w++)
    {
        //下面 if 语句前 2 个条件是为了确保顶点 w+1 与 verID 在一个连通分量上
        //写成“>=1”和“<INF”是为了通用性，适用于无向图（网）和有向图（网）
        //还有要注意顶点编号从 1 开始，数组下标从 0 开始，两者差 1。
      if( (G.AdjMatrix[verID-1][w]>=1) &&
          (G.AdjMatrix[verID-1][w]<INF) &&
          (!visited[w+1]) )
      {
          DFS(G,w+1);
      }
    }
```

```
}
```

（2）基于邻接表的 DFS 实现

【基于邻接表的 DFS 的一种实现】

```
//*************** 连通图或一个连通分量的 DFS ****************//
//* 函数功能：深度优先遍历连通图，或一个连通分量                *//
//* 入口参数  Graph G，待访问的图；int verID 起始顶点编号    *//
//* 出口参数：无                                            *//
//* 返 回 值：无                                            *//
//* 函 数 名：DFS(Graph &G, int verID)                      *//
//*********************************************************//
void DFS(Graph &G, int verID)
{
    cout<<G.Data[verID-1]<<"\t";  //访问编号为 verID 的顶点，相当于 visit(G, verID);
    visited[verID]=TRUE;          //标记编号为 verID 的顶点已经访问
    EdgeNode *p;
    p=G.VerList[verID-1].firstEdge;  //p 初始化为顶点 verID 的边链表的头指针
    while(p)
    {
        if(!visited[(p->adjVer)-1])
            DFS(G,p->adjVer);  //递归访问顶点 verID 的邻接点
        p=p->next;
    }
}
```

（3）通用遍历算法的一种实现

这个算法对邻接矩阵表示和邻接表表示都适用，算法中我们用函数返回值返回图 G 的连通分量数，且与上面描述不同的是这个算法也可以从指定顶点开始遍历。

【DFSTraverse 的一种实现代码】

```
//*******************     任意图的 DFS     *********************//
//* 函数功能：连通或非连通的 DFS 遍历                          *//
//* 入口参数  Graph G，待访问的图；int verID 起始顶点编号    *//
//* 出口参数：无                                            *//
//* 返 回 值：连通分量数                                      *//
//* 函 数 名：DFSTraverse(Graph &G, int verID)               *//
//*********************************************************//
int DFSTraverse(Graph &G, int verID)
{
    int vID;          //顶点编号
    int conNum=0;     //记录连通分量数
    for(vID=1;vID<=G.VerNum;vID++)  //访问标记数组初始化
        visited[vID]=false;
```

```
        DFS(G,verID);   //从指定的顶点，遍历指定的第一个连通分量
        conNum++;       //已经遍历一个连通分量，连通分量数加 1
        for(vID=1;vID<=G.VerNum;vID++)   //再依次遍历图中其它的连通分量
        {
            if(!visited[vID])
            {
                DFS(G,vID);
                conNum++;   //连通分量数加 1
            }
        }
        return conNum;
    }
```

4. 深度遍历算法的应用

下面讨论深度遍历算法的应用，通过例题的讨论，可加深对深度遍历算法的理解。

【例 6.4】设计算法以求解无向图 G 的连通分量的个数。

【分析】由前面分析可知，对图 G 来说，选择某一顶点 v 执行 DFS(G，v)，即可访问到 v 所在连通分量中的所有顶点，故为遍历整个图而选择起点的次数即是图 G 的连通分量数，而这可通过修改遍历整个图的算法 DFSTraverse 来实现：每调用一次 DFS 算法计数一次。我们上面给出的 DFSTraverse()实现代码就可以统计连通分量数。下面我们另外设计一个函数，不指定遍历的其实顶点，用函数值返回结果具体算法如下：

【算法描述】

```
int numOfCC(Graph G)
{       int i;    int k=0;
        for (i=1; i<=n; i++)
                visited[i]=FALSE;
        for (i=1; i<=n; i++)
             if (visited[i]==FALSE)
                 { k++;   DFS(G,i); }     //累计连通分量数
        return    k ;
}
```

这个函数中涉及到的 DFS 算法可直接照搬上面给出的 DFS 函数，故此处略。

类似的问题还有判断一个无向图是否连通图，有向图是否强联通图等。

【例 6.5】设计算法求出无向图 G 的边数，要求利用 DFS 算法。

【分析】如果明确给出了图 G 的存储结构是邻接矩阵或邻接表，则该问题的求解应该是容易的：可分别写出针对具体存储结构的算法。但能否在前面所讨论的遍历算法的基础上作适当修改，以得到不依赖于具体存储形式的算法？分析如下：

首先，在执行 DFS(v)时，搜索下一个访问顶点是从当前访问顶点 v 的邻接表中搜索的，因此，每搜索到一个邻接点即意味着一条以 v 为一个端点的边或弧，故应在 DFS 算法中将其数目计算进去。

其次，由于遍历算法保证每个顶点都要被访问一次，也就是都要作为起点调用 DFS 算法一次，因此，与每个顶点相关联的边都会被计算在内。由此而导致无向图中的每条边都要

被计算两次，故最后计算的边数的结果应该除以 2 才是实际的边数。综上所述得算法如下：

【算法描述】

```
void    DFS (Graph G, int v )   //改造原始的 DFS 算法，增加边计数功能
{   int   i;
    visited[v]=TRUE;          //直接设置访问标志，输出操作可省去。
    w=firstAdj(G,v);
    while (w!=0)
    {
        E++;                  //对边数计数，累计到全局变量 E 中
        if (visited[w]==FALSE)
            DFS(G,w);
        w=nextAdj(G,v,w);
    }
}
int   Enum (Graph G )       //考虑到非连通图，需要对每个连通分量中边计数
{   int   i;   static E=0;
    for (i=1;   i<=n;   i++)
      visited[i]=FALSE;                 //初始化顶点的访问标志
    for (i=1;   i<=n;   i++)
      if (visited[i]==FALSE;)
        DFS(G,i);
    return   E/2;       //返回边数
}
```

与上一例不同的是，本例对 DFS 算法作了修改，增加了边的计数功能。

6.3.2 广度优先搜索遍历算法及其应用

广度优先搜索遍历（Breadth_First Search Traverse -- BFS）算法是另一种典型的算法，是一种由近而远的层次遍历方法。

1. 基本广度优先搜索遍历算法描述

假定图 G 是连通的，选定从顶点 v_0 出发广度优先搜索遍历算法 BFS(v_0)描述如下：

（1）访问 v_0（可作为访问的第一层）。

（2）依次访问 v_0 的各邻接点 v_{11}，v_{12}，…，v_{1k}（可作为访问的第二层）。

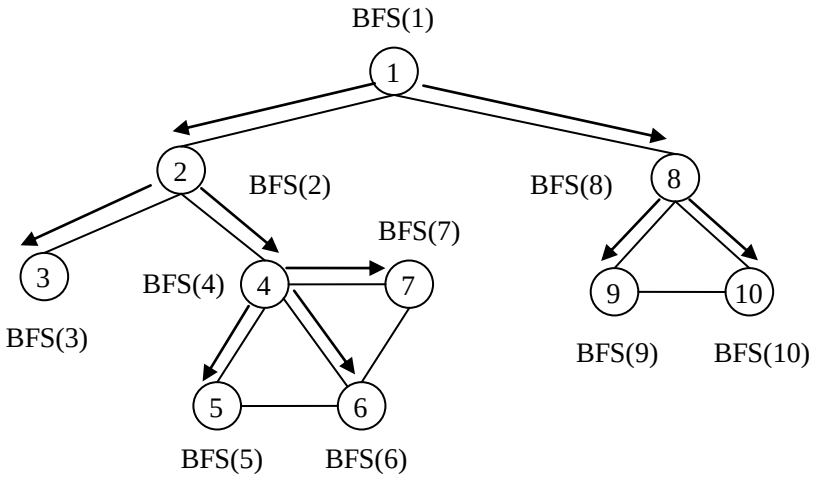
（3）假设最近一层的访问顶点依次为 v_{i1}，v_{i2}，…，v_{ik}，则依次访问 v_{i1}，v_{i2}，…，v_{ik} 的未被访问的邻接点。

（4）重复（3），直到找不到未被访问的邻接点为止。

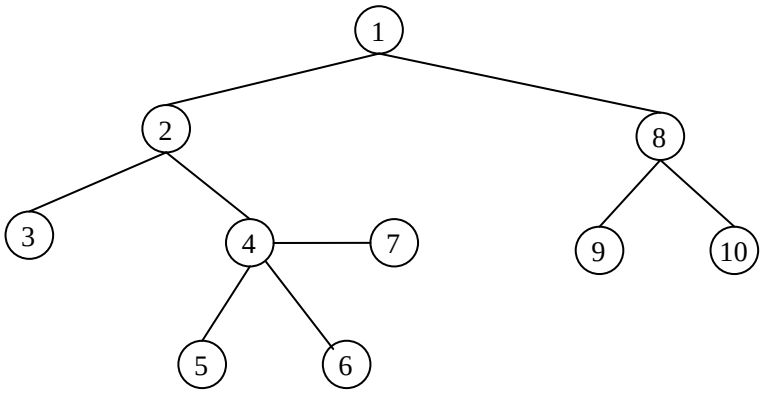
这一描述较为直观，应该容易理解的，故对其求解过程的实例只做简要叙述。另外，其中的操作（2）可作为操作（3）的特例，只不过为使初学者不至感到太突然而添加的，可省略。这个算法的描述也是递归的。所谓“基本”是指这个算法只能遍历连通图，或一般图的一个连通分量。

对图 G_9 从顶点 1 出发的广度遍历过程同样可用图形方式表示，如图 6-26(a)所示。其中，

仍用箭头表示搜索顶点的路线，同样可得到 BFS(1)生成树（如图 6-26(b)）。所不同的是访问次序是依层次方式进行的，其顶点**访问序列**为 1,2,8,3,4,9,10,5,6,7。



（a）对图 G_9 的 BFS(1)的执行过程示意图



（b）图 G_9 的 BFS(1)生成树

图 6-26 对图 G_9 的 BFS(1)的执行过程示意图

2. 广度遍历算法

广度优先搜索遍历算法也要分两个算法来讨论，下面我们先讨论基本 BFS 算法的实现，再基于 DFS 算法讨论一般图的广度遍历算法，即对连通和非连通图都适用的算法。

（1）BFS 算法

为实现基本广度遍历算法 BFS，作如下讨论：

① 与 DFS 类似，同样要设访问标志数组 visited。

② 为了能依次访问上一层次的访问序列中的各顶点的邻接点，需要设置一个结构来保存上一层次的顶点，即刚刚被访问过且其后继邻接点还未被访问的顶点，并且这一结构还要满足这样的条件：这一层中最先被访问的顶点，其后继也应被最先检测到。由此可知，这一结构应是队列。

③ 既然涉及到队列（不妨设为 Q），则需要在适当的情况下操作队列：

（a）初始化：开始时要设置队列 Q 为空，不妨用 initialQueue(Q)；

（b）入队：每访问一个顶点 v，除了访问操作、设置标志外，还要将其入队。在

此不妨设为 EnQueue(Q,v)。

（c）出队：从队列中删除出一个顶点 v，不妨用 outQueue(Q)），意味着要依次访问 v 的所有未被访问过的邻接点。为了求解其各邻接点，仍采用 DFS 算法中的方式。

综上所述，可将 BFS（v_0）可细化如下：

① initialQueue(Q)
② 访问 v_0（包括三个操作：访问 v_0，设置标志，入队）
③ 若队列 Q 为空，则结束 BFS（v_0），否则，转④
④ v=outQueue（Q）　　//从队列 Q 中取出队头元素，并出队
⑤ w=v 的第一邻接点　　//依次访问 v 的未被访问的邻接点
⑥ 若 w 未被访问过，则访问 w（同样包括访问 w，设置标志，入队这三个操作）
⑦ w＝v 的下一个邻接点，若不存在，则转③
⑧ 转⑥

由此得 BFS 算法如下：

【BFS 算法描述】

```
void BFS(Graph G, int v)      //从指定的编号为 v 的顶点出发广度遍历
{
    int w;  queue  Q;
    initialQueue(Q);      //初始化队列
    visite(v);  visited[v]=TRUE;  EnQueue(Q,v);  //访问 v、标记 v 已访问、v 入队
    while (!queueEmpty(Q))
    {
        v=outQueue(Q);
        w=firstAdj(G,v);
        while (w!=0)
        {
            if (!visited[w])
            {
                visite(w);  visited[w]=TRUE;  EnQueue(Q,w);
            }
            w=nextAdj(G,v);
        }
    }
}
```

注意这个**算法是非递归的**。还有，算法中引用顶点都是使用顶点的编号，顶点编号从 1 开始。

（2）一般图的（通用）广度优先搜索遍历算法

与 DFS 算法相似，BFS(G, v)只能从指定顶点 v 出发遍历连通图，或一个连通分量。使用深度遍历相同的处理方法，得到基于 BFS 的遍历整个图的算法如下：

【通用广度遍历算法描述】

```
void BFSTraverse(Graph G)
{
```

```
    int   i;
    for (i=1; i<=n; i++)
        visited[i]=FALSE;
    for (i=1; i<=n; i++)
        if(!visited[i])
          BFS(G,i);
}
```

3. 广度遍历算法的参考实现代码

与 DFS 类似，上面给出的 BFS(G, v)算法中 firstAdj()和 nextAdj()两个函数没有给出具体实现，读者上机实验时需要自己完成这个工作。下面分别针对图的邻接矩阵表示和邻接表表示参考实现代码，供您上机时参考，在这两个实现代码中也把 fristAdj()和 nextAdj()函数的功能直接在 BFS 算法中实现，取消了这两个函数，请读者阅读时注意。

（1）基于邻接矩阵的 BFS 实现

【基于邻接矩阵的一种 BFS 实现代码】

```
//*************** 连通图或一个连通分量的 BFS *****************//
//* 函数功能：广度优先遍历连通图，或一个连通分量              *//
//* 入口参数  Graph G，待访问的图；int verID 起始顶点编号    *//
//* 出口参数：无                                             *//
//* 返 回 值：无                                             *//
//* 函 数 名：BFS(Graph &G, int verID)                       *//
//***********************************************************//
void BFS(Graph &G, int verID)
{
    int u;
    seqQueue Q;   //定义一个循环顺序队列
    initQueue(&Q);                  //初始化队列
    cout<<G.Data[verID-1]<<"\t";   //访问编号为 verID 的顶点，相当于 visit(G, verID);
    visited[verID]=TRUE;            //标记编号为 verID 的顶点已经访问
    enQueue(&Q, verID);             //编号为 verID 的顶点入队
    while(!queueEmpty(Q))           //队列不空循环处理顶点
   {
        queueFront(Q, u);   //取队头元素到 u，即顶点编号为 u。
        outQueue(&Q);       //u 出队
        for(int w=0;w<G.VerNum;w++)
        {
            if((G.AdjMatrix[u-1][w]>=1) &&
               (G.AdjMatrix[u-1][w]<INF) &&
               (!visited[w+1]))
            {
                                    //访问编号为 w+1 的顶点，相当于 visit(G, w+1);
```

```
                cout<<G.Data[w]<<"\t";
                visited[w+1]=TRUE;          //标记编号为 w+1 的顶点已经访问
                enQueue(&Q,w+1);          //编号 w+1 的邻接点入队
            }
        }
    }
}
```

（2）基于邻接表的 BFS 实现

【基于邻接表的一种 BFS 实现代码】

```
//*************** 连通图或一个连通分量的 BFS *****************//
//* 函数功能：广度优先遍历连通图，或一个连通分量                *//
//* 入口参数  Graph G，待访问的图；int verID 起始顶点编号     *//
//* 出口参数：无                                              *//
//* 返 回 值：无                                              *//
//* 函 数 名：BFS(Graph &G, int verID)                        *//
//*************************************************************//
void BFS(Graph &G, int verID)
{
    int u;                  //顶点编号
    EdgeNode *p;          //边链表结点指针
    seqQueue Q;
    initQueue(&Q);     //初始化循环顺序队列

    cout<<G.VerList[verID-1].data<<"\t";
                                  //访问编号为 verID 的顶点，相当于 visit(G, verID);
    visited[verID]=TRUE;           //标记编号为 verID 的顶点已经访问
    enQueue(&Q, verID);             //编号为 verID 的顶点入队
    while(!queueEmpty(Q))           //队列不空循环处理顶点
    {
        queueFront(Q, u);           //取队头元素到 u，即顶点编号为 u。
        outQueue(&Q);               //u 出队
        p=G.VerList[u-1].firstEdge;   //获取当前边链表的头指针
        while(p)
        {
            if(!visited[p->adjVer])
            {
                cout<<G.VerList[p->adjVer -1].data<<"\t";
                        //访问编号为 p->adjVer 的顶点，相当于 visit(G, p->adjVer);
                visited[p->adjVer]=TRUE; //标记编号为 p->adjVer 的顶点已经访问
                enQueue(&Q,p->adjVer);       // 编号 p->adjVer 顶点入队
```

```
            }
            p=p->next;  //移动到下一条边，取下一个邻接点
        }
    }
}
```

（3）BFSTraverse 算法的一种实现

这个算法对邻接矩阵表示和邻接表表示都适用，算法中我们用函数返回值返回图 G 的连通分量数，且与上面描述不同的是这个算法也可以从指定顶点开始遍历。

【BFSTraverse 的一种实现代码】

```
//*******************     任意图的 BFS     *********************//
//* 函数功能：连通或非连通的 BFS 遍历                            *//
//* 入口参数  Graph G，待访问的图；int verID 起始顶点编号    *//
//* 出口参数：无                                                *//
//* 返 回 值：连通分量数                                        *//
//* 函 数 名：BFSTraverse(Graph &G, int verID)                  *//
//************************************************************//
int BFSTraverse(Graph &G, int verID)
{
    int vID;
    int conNum=0;  //记录连通分量数
    for(vID=1;vID<=G.VerNum;vID++)  //访问标记数组初始化
        visited[vID]=false;

    BFS(G,verID);  //从指定的顶点 verID 开始，遍历指定的第一个连通分量
    conNum++;       //连通分量数加 1
    for(vID=1;vID<=G.VerNum;vID++)  //再依次遍历图中其它的连通分量
    {
        if(!visited[vID])
        {
            BFS(G,vID);    //遍历 vID 所在的连通分量
            conNum++;      //连通分量数加 1
        }
    }
    return conNum;        //返回连通分量数
}
```

4．广度遍历算法应用实例

广度遍历算法是一个层次型的遍历，即由近而远地访问各个顶点，并且距起点 v_0 最短路径长度为 L 的顶点一定在 BFS(v_0)生成树的第 L+1 层上。因此，运用广度遍历算法的基本思想、方法和算法可实现许多问题的求解。下面通过实例讨论这一算法和应用。

【例 6. 6】已知一图的邻接表如图 6-27 所示，不用还原出原图，请执行 BFS(1)，构造其 BFS（1）生成树及其遍历序列。

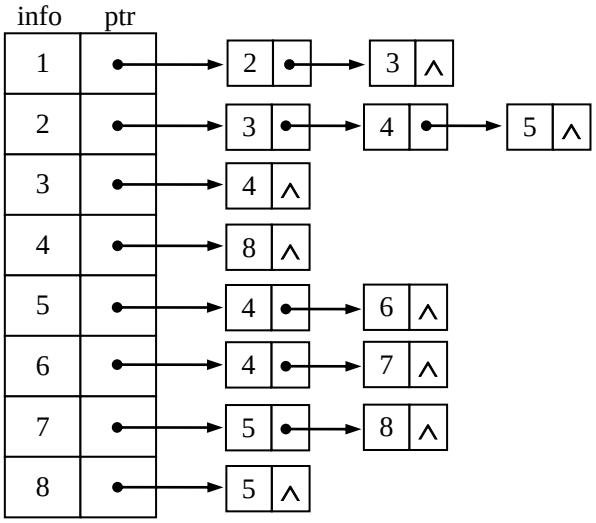
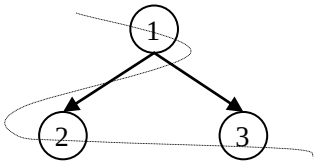


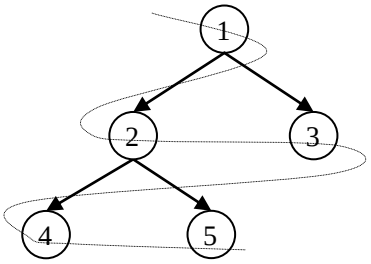
图 6-27 例 6-6 的邻接表图

【解】由 BFS 算法可知，执行 BFS(1)时，首先要访问顶点 1，然后依次访问其邻接点 2 和 3。由此可得到其局部生成树如图 6-28(a) 所示。

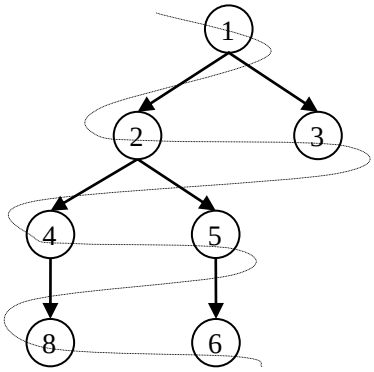
接着再依次访问这一访问层次中的顶点 2 的未被访问的邻接点 4 和 5（虽然顶点 3 也是其邻接点，但此前已经被访问过），以及顶点 3 的未被访问的邻接点（仅有顶点 4，但此时已经被访问过了，故实际上没有访问到），结果如图 6-28(b) 所示。



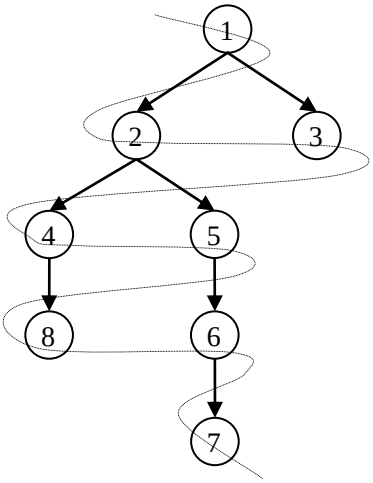
（a）访问顶点 1 的局部生成树



（b）访问顶点 2、3 的局部生成



（c）访问顶点 4、5 的局部生成



（d）最终生成树

图 6-28 例 6.6 的求解过程

下面再依次访问顶点 4 的未被访问的邻接点（仅有顶点 8）及顶点 5 的未被访问的邻接点 6（另一个邻接点 4 此前已经被访问），得结果如图 6-28(c) 所示。

下面再依次访问顶点 8 的未被访问得邻接点（已没有了），及顶点 6 的未被访问的邻接点（仅剩顶点 7），得顶点如图 6-28(d)所示。

由此可知，其访问顶点的序列为 1,2,3,4,5,8,6,7。

6.4 最小生成树

现实中的许多问题，表面上似乎有较大的差异，然而其实际的求解方法可能非常类似。因此，如果用合适的数学模型来表示这些问题，就可能会使这些问题变成几乎相同的问题，因而可借助一些成熟的方法来求解。本节及后面的几节讨论将图这种模型运用于实际问题的求解。

下面先从一个实例开始本节的内容：假设在某地有一个煤气供应站点及 n-1 个生活小区，现在要给这些小区铺设煤气管道。请问应怎样铺设管道能使总造价最低（假设已知各小区之间是否可连接，以及相应的造价）。与此类似的问题还要许多。

可将这类问题转化为图的问题：将各小区分别表示为图的一个顶点，两点之间能连接就表示为一条边，连接的造价表示为边的权值。在这种表示下，原问题就变成了这样的问题：

从图中选取若干条边，将所有顶点连接起来，并且所选取的这些边的权值之和最小。

显然，所选取的边不会构成回路（否则，可去掉回路中的一条边使权值之和更小），因而构成了一棵树，称这样的树为**生成树**。由于这一生成树的权值之和最小，故称为**最小生成树**。构造最小生成树是本节的内容。

关于最小生成树的求解，有两种较为典型的的算法－－Prim 算法和 Kruskal 算法。下面分别介绍这两种算法的求解思想。

6.4.1 Prim 算法

下面分几个层次来讨论 Prim 算法，首先介绍算法的基本思想，然后以实例来详细描述算法的求解过程，在此基础上讨论算法设计的相关问题，最后给出 Prim 算法描述。

1. 求解方法

Prim 算法通常要指定一个起点，其求解思想非常简单：

首先将所指定的起点作为**已选顶点**，然后反复在满足如下条件的边中选择一条最小边，直到所有顶点成为已选顶点为止（选择 n-1 条边）：**一端已选，另一端未选。**

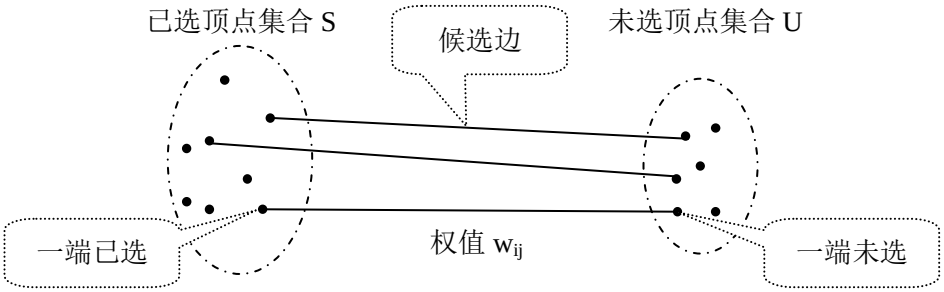


图 6-29 Prim 算法已选和未选顶点集示意图

为了更好说明问题，不妨引入几个符号，假定 S 为已选顶点集合，U 为未选顶点集合，WE 为候（待）选边的集合，TE 为已选表的集合（存放选定的符合条件的边）。初始化时 S 为空，U=V，WE 为空，TE 为空。然后把起点，比如 v_0，从 U 移到 S 中；则 S 中的 v0 与 U 中某些顶点有边（弧）相连，形成候选边集合，放入 WE 中，这些边即“一端已选、一端未选”；从 WE 中选择权值 w_{ij} 最小的一条边加入集合 TE，并将此边的另一端顶点移到 S 中。每次从 U 移动一个顶点到 S 后，要更新候选边集 WE。重复上述操作，直到所有顶点都从 U 移到 S，即 S=V，U 为空，TE 中保存的即是每次选择的边。如图 6-29 所示。如果一个连通网有 n 个顶点，则总共需要经过 n-1 次选择，算法结束时，TE 中应该保存了 n-1 条边，这些边即是最小生成树的边。

2. 求解实例

例如，对图 6-30 所示的连通网，用图形化方法模拟 Prim 算法的求解过程，设顶点 1 为起点。

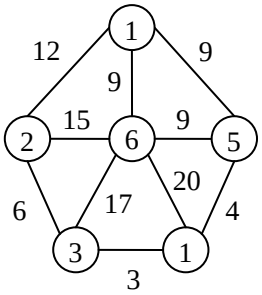


图 6-30 Prim 算法求解例图

Prim 算法求解的各步的中间结果如下：

初始化时 S={}，U={1,2,3,4,5,6}，WE={}，TE={}。

① 第一轮选择：顶点 1 从 U 移到 S，则 S={1}，U={2, 3, 4, 5, 6}；更新候选边集，此时满足“一端已选、一端未选”的候选边有 3 条，即 WE={(1, 2), (1, 5), (1, 6)}，其中权值最小的边是(1,6)和(1,5)（权值均为 9），不妨选**(1,6)**，将其从 WE 移到 TE。这样，顶点 6 就成为新的已选顶点。见图 6-31(a) 所示，原图上加虚线部分为候选边，已选顶点用黑色背景表示。此时有：

S={1}，U={2,3,4,5,6}，WE={(1,2),(1,5)}，TE={(1,6)}。

② 第二轮选择：顶点 6 移到 S，则 S={1,6}，U={2,3,4,5}；更新候选边集，WE={(1,2),(1,5),(6,2),(6,3),(6,4),(6,5)}，此时边(1,5)和(6,5)权值最小为 9，故均可入选，不妨选**(6,5)**，将其加入 TE。这样，顶点 5 成为新的已选顶点，如图 6-31(b) 所示。这时有：

S={1,6}，U={2,3,4,5}，WE={(1,2),(1,5),(6,2),(6,3),(6,4) }，TE={(1,6), (6,5)}。

③ 第三轮选择：顶点 5 移到 S，则 S={1, 6, 5}，U={2, 3, 4, }；更新候选边集，WE={(1,2),(6,2),(6,3),(6,4),(5,4)}，因为顶点 1 和 5 已选，WE 中候选边(1, 5)删去；其中边**(5,4)**权值最小为 4，故入选，使顶点 4 成为新的已选顶点。结果如图 6-31(c) 所示。这时有：

S={1,6,5}，U={2,3,4}，WE={(1,2),(6,2),(6,3),(6,4)}，TE={(1,6), (6,5), (5,4)}。

④ 第四轮选择：顶点 4 移入 S，则 S={1,6,5,4}，U={2,3}；更新候选边集，WE={(1,2),(6,2),(6,3),(4,3)}，因为顶点 6 和 4 已选，WE 中候选边(6, 4)删去；其中边（**4,3**）权值最小为 3，故入选，顶点 3 成为最新的已选顶点。结果如图 6-31(d) 所示。这时有：

S={1,6,5,4}，U={2,3}，WE={(1,2),(6,2),(6,3)}，TE={(1,6), (6,5), (5,4),(4,3)}。

⑤ 第五轮选择：顶点 3 移入 S，则 S={1,6,5,4,3}，U={2}；更新候选边集，WE={(1,2),(6,2),(3,2)}，因为顶点 6 和 3 已选，WE 中候选边(6, 3)删去；其中边（**3,2**）权值最小为 6，故入选，顶点 2 成为最新的已选顶点。结果如图 6-31(e) 所示。这时有：

S={1,6,5,4,3}，U={2}，WE={(1,2),(6,2)}，TE={(1,6), (6,5), (5,4),(4,3),(3,2)}。

⑥ 第六轮选择：顶点 2 移到 S 中，则 S={1,6,5,4,3,2}，U={}；更新候选边集，WE={}，因为顶点 1、2、6 均已选，候选边(1,2)，(6,2)删去。算法结束，这时有：

S={1,6,5,4,3,2}，U={}，WE={}，TE={(1,6), (6,5), (5,4),(4,3),(3,2)}。

此时 TE 中保存的就是最小生成树的 5 条边（n-1 条），这是一种总造价最低的结果。显然，这不是唯一的结果。

上述求解过程也可用表 6-1 所示的表格形式來描述。其中，表中每一行表示一次求解；

每一列对应一个顶点的求解状态；每个未选顶点对应的当前行中，列出了与所有已选顶点之间的边（即候选边）及其权值；将本次入选的边加上一个框；当一个顶点已被选择后其状态不再变化，余下各行用阴影表示，表示这个顶点不需再参与求解。

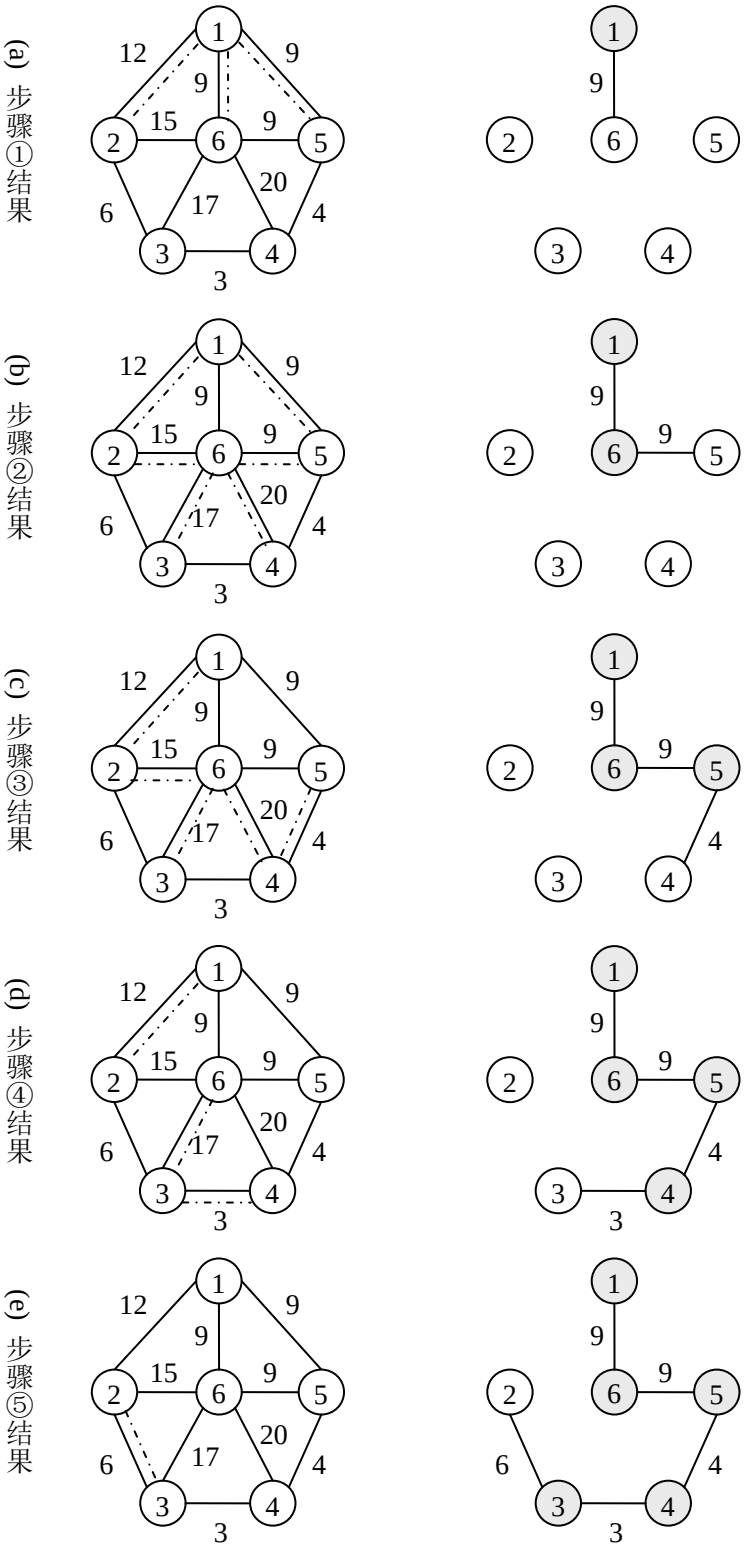


图 6-31 Prim 算法求解例图

表 6-1　Prim 算法求解过程的表格化描述

已选顶点集 U	顶点 1 候选边	顶点 2 候选边	顶点 3 候选边	顶点 4 候选边	顶点 5 候选边	顶点 6 候选边
{1}		（1,2）/12	（1,3）/∞	（1,4）/∞	（1,5）/9	（1,6）/9
{1,6}		（1,2）/12 （6,2）/15	（6,3）/17	（6,4）/20	（1,5）/9 （6,5）/9	
{1,6,5}		（1,2）/12 （6,2）/15	（6,3）/17	（6,4）/20 （5,4）/4		
{1,6,5,4}		（1,2）/12 （6,2）/15	（6,3）/17 （4,3）/3			
{1,6,5,4,3}		（1,2）/12 （6,2）/15 （3,2）/6				
{1,6,5,4,3,2}						

3. Prim 算法思想

对连通网 N=(V,E)，仍设 S 为已选顶点集，U 为未选顶点集，WE 为候选边集，TE 为已选边集，根据上面的讨论，我们得到 Prim 算法流程如下：

① 初始化：S 为空，U=V，WE 为空，TE 为空；

② 将起点从 U 移到 S；

③ 更新候选边集合 WE；

④ 如果集合 U 非空，重复执行下列操作：

（a）从候选边集 WE 中选择一条**权值最小的边**，比如(v,u)，加入到已选边集 TE 中。这里 v 为“已选一端”，u 为“未选一端”。如果最小权值的边有几条，任选其中一条；

（b）把（a）中所选边的另一端顶点，u，加入到已选顶点集合 S 中；

（c）更新候选边集合 WE；

循环结束后，整个算法结束，此时有：S=V，U 为空，WE 为空，TE 中为最小生成树的 n-1 条边。

4. 算法实现及讨论（*）

下面我们先讨论算法实现要解决的相关问题，然后给出一种 Prim 算法的实现方案。这部分内容难度较大，可作为选学内容。

为实现 Prim 算法，首先需要考虑解决如下的问题：

（1）已选顶点和未选顶点的标识问题

我们在前面的讨论和描述中使用了两个集合 S 和 U 来区分已选顶点和未选顶点，这只是为了叙述更清晰。实现时我们用一个 BOOL 类型（或 int 型）的**一维数组 visited[n]**就可以代替前面的 S 和 U 两个集合。数组的长度等于图的顶点数。数组的下标对应顶点（编号）。数组元素值为 TRUE 表示对应顶点已选；FALSE 表示顶点未选。

例如：对图 6-30 的例子，需要一个长度为 7 的 visited[]的数组来标识已选顶点和未选顶点，假定 visited[0]不用。比如对顶点 1，visited[1]=TRUE 表示顶点 1 已选，visited[1]=FALSE 标记未选；对顶点 4，visited[4]=TRUE 为顶点 4 已选，visited[4]=FALSE 为未选，等等。这里就看到了本章 6.3.1 小节里提到的用顶点编号来处理的好处—数组下标代表顶点，后面我们还会体会到这一点。

（2）候选边集的优化问题

从前面的实例可以看出，随着选择最小权值边的进行，候选边集不断更新，会出现这样的情况：未选顶点集 U 中的一个顶点与已选顶点集 S 中的多个顶点有边（弧）相连。比如图 6-31(b)，S={1,6}，U={2,3,4,5}，WE={(1,2),(1,5),(6,2),(6,3),(6,4),(6,5)}，U 中的顶点 2 与 S 中 1 和 6 分别有边相连，即(1,2)和(6,2)，分析一下就会知道，这两条边中只要留下权值最小的一条就可以了，即只要留下边(1,2)，剔除边(6,2)，因为每次从候选边集中都是选择权值最小的边，在选择邻接点为 2 的边时，不可能选中边(6,2)。同样，图 6-31(c)中，S={1, 6, 5}，U={2, 3, 4, }，WE={(1,2),(6,2),(6,3),(6,4),(5,4)}，U 中顶点 4 与 S 中的 6 和 5 有边相连，即(6, 4)和(5, 4)，只要保留权值最小的(5, 4)，剔除边(6, 4)。结论：**对 U 中每个顶点最多只需保留一条权值最小的边与 S 中某顶点相连，作为候选边**。由此可知候选边集 WE 中至多只会有 n-1 条候选边，我们在后面实现时，为方便处理 WE 的大小设置为 n。

（3）边的表示和标记

表示网的一条边需要 3 部分信息：2 个顶点和边的权值。我们仍用数组来保存边，顶点用编号代表。与前面讨论的 visited[]数组类似，这里我们把 2 个顶点中的一个顶点对应到数组下标，那么数组元素只要能表示出另一个顶点和边的权值即可，为此，我们需要定义一个辅助结构，由两个分量构成：另一个顶点编号和边的权重，结构描述如下：

```
typedef struct minEdgeType
{
    int v;                  //边中已选一端的顶点编号
    cellType eWeight;   //边的权值
} MinEdgeType;          //边数组的元素类型
```

这样，前面的候选边集 WE 和已选边集 TE 都可以定义为：

```
MinEdgeType WE[n];   //n 为顶点数
MinEdgeType TE[n];
```

WE 和 TE 的下标都代表顶点（编号），表示图 6-31(d)中的候选边{(1,2),(6,3),(4,3)}就变为： WE[1].v=1， WE[1].eWeight=12； WE[2].v=6， WE[2].eWeight=17； WE[2].v=4，WE[2].eWeight=3。已选边{(1,6),(6,5),(5,4)}就成为：TE[5].v=1，TE[5].eWeight=9；TE[4].v=6，TE[4].eWeight=9，TE[3].v=5，TE[3].eWeight=4。注意这里顶点编号与数组下标差 1。

用这种方式表示边，看上去怪怪的，比如边(6, 3)要表示为 WE[2]=6 或 TE[2]=6，但算法处理起来还是比较方便的，很多教材都是沿用这种表示方式，本书也是这样，希望读者认真体会和理解这种表示习惯，以免阅读程序和上机实现时感到一头雾水。

有了上述边的表示方法后，接下来还要解决已选边和候选边的标记问题。在前面的描述中我们一直使用候选边集合 WE 和已选边集合 TE。在使用上述表示法定义 WE[]和 TE[]时，我们发现他们的定义是完全相同的，那么能不能把这两个数组合并为一个数组呢？-- 回答是肯定的！我们在后面的实现中就使用一个数组 TE[]同时表示候选边集合和已选边集合。那么在一个数组 TE[]中怎样来区分候选边和已选边呢？这需要借助前面介绍的 visited[]数

组，两个数组配合就可以区分出候选边和已选边。不管是候选边，还是已选边，至少有一个顶点已经选择，我们**让 TE[i].v 固定表示已选顶点一端，数组下标对应边的另一端顶点 i+1，i+1 可能已选（已选边），也可能未选（候选边）**。另一顶点通过 visited[]数组来判定是已选 TRUE，还是未选 FALSE。如果为 visited[i]==TRUE，则这条边(TE[i].v, i+1)的 2 个顶点都已选，对应边为已选边；为 visited[i]==FALSE，则一端已选（TE[i].v），一端未选（i+1），(TE[i].v, i+1)为候选边。

举一个例子，判定边(k, i+1)为已选边还是未选边，k 固定为已选顶点。用上面表示方法表示即 TE[i].v=k，这里 k 固定为已选端顶点，我们只要判断顶点 i+1 是否已经选择即可。如果 visited[i]==TRUE，说明顶点 i+1 已选，(k, i+1)为已选边；若 visited[i]==FALSE，则顶点 i+1 未选，(k, i+1)为候选边。

这里我们再次看到了本章 6.3.1 小节里提到的用顶点编号来处理的好处—数组下标代表顶点，处理方便，降低数组维数。

（4）实现 Prim 算法需要的几个辅助函数

① BOOL HasEdge(Graph &G, int vBegin, int vEnd, eInfoType &eWeight)

函数功能：判断顶点 vBegin 和 vEnd 之间是否有边，函数值为 TRUE 表示两个顶点之间有边（弧），FALSE 表示两个顶点之间没有边（弧）相连。同时利用这个函数的参数 eWeight 返回两个顶点之间边的权值，没有边则 eWeight=∞。对于图的邻接矩阵表示相对简单，直接判断即可，甚至不需要这个函数；但对邻接表表示，就需要搜索顶点 vBegin 的边链表来完成。下面给出邻接表表示的实现：

```
BOOL   HasEdge(Graph &G, int vBegin, int vEnd, eInfoType &eWeight)
{
    EdgeNode *p;     //边链表结点指针
    int f=FALSE;     //是否有边的标记
    eWeight=INF;   //边的权值初始化为无穷大
    p=G.VerList[vBegin-1].firstEdge;   //取得 vBegin 的边链表的头指针
    while(p)
    {
        if( p->adjVer==vEnd )   //vEnd 是 vBegin 的连接点，有边
        {
            f=TRUE;   //vBegin 与 vEnd 之间有边，退出循环，返回 TRUE
            eWeight=p->eInfo;   //获取权值，用参数返回主调函数
            break;
        }
        p=p->next;   //搜索边链表中下一个结点
    }
    return f;
}
```

② void InitialTE(Graph &G, MinEdgeType TE[], int vID)

初始化边数组 TE，根据上述讨论，候选表集和已选边集用同一个数组 TE[]来保存，TE 的长度等于图（网）的顶点数 n，TE[]的类型为上面定义的辅助结构类型：MinEdgeType 类

型。这个函数也可以直接放在 Prim 算法中，但为了 Prim 算法的简洁，我们单独使用此函数来实现初始化。函数参数 vID 是算法开始的起始顶点编号。同时在初始化函数中把 vID 设置为已选顶点，并以此来初始化 TE[]中候选边集。 这个函数对邻接矩阵和邻接表表示一致，代码如下：

```
void InitialTE( Graph &G,   MinEdgeType TE[], int vID )
{
    int i;
    eInfoType eWeight;   //定义变量保存边的权值
    for(i=1;i<=G.VerNum;i++)
    {
        //初始化边数组 TE[]
        if(HasEdge(G, vID, i, eWeight))     //如果顶点 vID 与 i 之间有边
        {               //保存边，vID 为一个顶点，数组下标 i 代表另一个顶点，
            TE[i-1].v=vID;
            TE[i-1].eWeight=eWeight;   //保存边的权值
        }
        else          // vID 与 i 之间没有边，权值置为无穷大
            TE[i-1].eWeight=INF;
    }
}
```

③ int GetMinEdge(Graph &G, MinEdgeType TE[])

函数功能：从边集数组 TE[]中获取权值最小的候选边，即算法描述中选择操作。函数返回值为候选边另一端的顶点编号（即在未选顶点集 U 的那个顶点）。对图的两种表示代码相同，实现代码如下：

```
int GetMinEdge(Graph &G,   MinEdgeType TE[] )
{
    eInfoType eMin=INF;   //eMin 保存最小的权值，初始化为无穷大
    int i, j=0;
    for( i=1;i<=G.VerNum;i++ )
    {
        if( visited[i-1]==FALSE && TE[i-1].eWeight<eMin )
        {           //i 顶点未选，且权值比 eMin 小，
                    //暂选 i 为候选顶点，对应的边( TE[i-1].v,   i )为候选边。
            j=i;
            eMin=TE[i-1].eWeight;
        }
    }
    return j;   //j 为选中的权值最小的候选边的一个顶点，
                //对应的边( TE[j-1].v,   j )为选中的边
}
```

④ void UpdateTE(Graph &G, MinEdgeType TE[], int vID)

函数功能：当顶点 vID 被选中变为已选顶点后，更新候选边集合。代码如下：

```
void UpdateTE(Graph &G, MinEdgeType TE[], int vID)
{
    //对新选出的编号为 vID 的顶点（新加入集合 S 中），更新候选边集合
    int i,j;
    eInfoType eWeight;
    for(i=1;i<=G.VerNum;i++)
    {
        if(visited[i-1]==FALSE)  //如果顶点 i 为未选顶点，即 i 在集合 U 中
        {
            //检查 S 中 vID 与 U 中 i 之间是否相邻（有边）
            //检查 S 中的 vID 与 i 之间的边权值是否更小，若更小则更新(vID,i)权值
            if(HasEdge(G,vID,i, eWeight) && eWeight<TE[i-1].eWeight)
            {
                TE[i-1].v=vID;      //(vID,i)作为未选顶点 i 对应的候选边
                TE[i-1].eWeight=eWeight;  //更新(vID,i)的权值
            }
        }
    }
}
```

有了上面这些辅助函数后，下面隆重推出 Prim 算法，代码如下：

【Prim 算法描述】

```
void Prim( Graph &G,   int vID )    //从起始顶点 vID 开始 Prim 算法
{           //定义边数组，因为图的顶点数是变化的，
            //所以数组长度设为预定的最大值 MaxVerNum
    MinEdgeType TE[MaxVerNum];  //TE[i]的下标加 1，即 i+1 为选定边的终点
                                //TE[i].v 为选定边的起点
    int i;
    int   curID;                //当前选择顶点编号
    InitialTE( G, TE, vID );  //初始化边数组（候选边集和已选边集）
    visited[vID-1]=TRUE;    //标记 vID 为已选顶点，即进入已选集合 S

    for(i=1;i<G.VerNum;i++)  //循环选择 n-1 条边，产生最小生成树
    {           //从 TE 选择权值最小的候选边，返回未选一端的顶点编号到 curID
        curID=GetMinEdge( G, TE );
        visited[curID-1]=TRUE;  //标记 curID 已选，即进入 S 中
        UpdateTEt(G, TE, curID);  //选择 curID 后，更新候选边集
    }
```

}

算法结束后，最小生成树的边保存在 TE[]数组中。使用 TE[]数组要注意，我们定义数组时使用的是最大长度 MaxverNum，而 TE[]有效数据范围在 0—n-1 之间，n 为图的顶点数，即 n=G.VerNum。TE[]数组中有 n 条有效数据，但生成树只需要 n-1 条边，**其中 TE[vID-1] 这条数据无效**，即不是生成树上的边，vID 为起始顶点。

【算法分析】

Prim 算法的执行时间取决于 for 循环的时间，在这个 for 循环中调用了 2 个函数 GetMinEdge( )和UpdateTEt()，它们的时间复杂度为 O(n)，所以，Prim 算法的时间复杂度为：$O(n^2)$。且**执行时间为连通网 N 的边数无关，适合于求解边数相对较多（较稠密）的网的生成树**。

6.4.2 Kruskal 算法

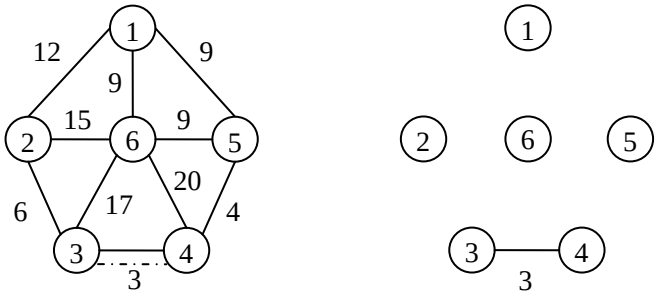
1. 求解方法：

Kruskal 算法与 Prim 算法的求解思路不同。Kruskal 算法的基本思想是：反复在图中未选边中选出一条权值最小的边，前提是**和已选边不构成回路，**直到选出 n-1 条边构成生成树。因为每次都是选择权值最小的边，所以得到的是最小生成树。

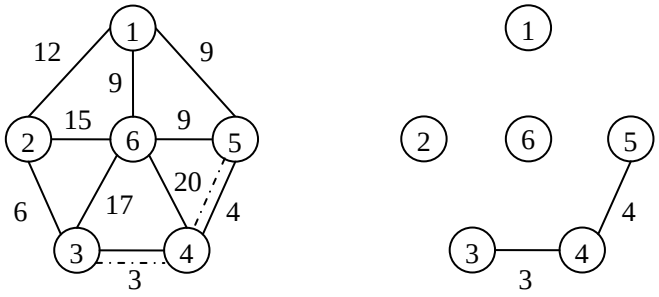
2. 求解实例

例如，对前面图 6-30 所示的连通网，用图形化方法模拟 Kruskal 算法的求解过程。

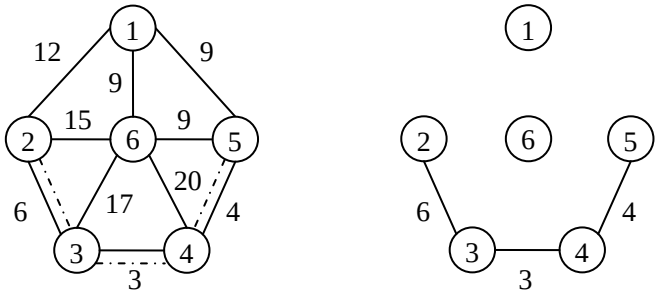
Kruskal 算法求解的各步的中间结果如图 6-32 所示：



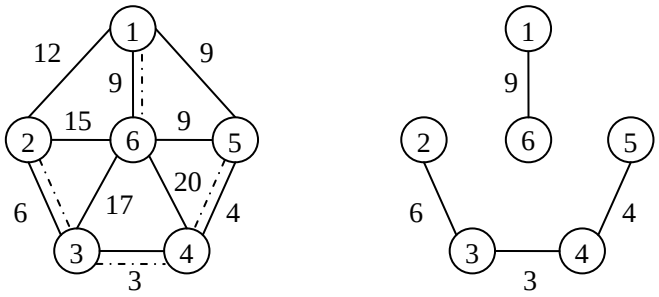
(a) 选择最小边(3,4)，权值 3



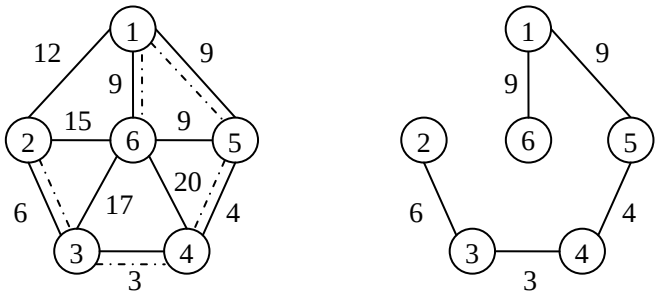
（b）选择最小边(4,5)，权值 4



(c) 选择最小边(2,3)，权值 6



(d) 最小边有 3 条(1,6)、(1,5)和(6,5)，权值 9，不妨选(1,6)



(e) 最小边有 2 条 (1,5)和(6,5)，权值 9，不妨选(1,5)。结束

图 6-32 Kruskal 算法求解例图

3. 算法思想

对连通网 N=(V, E)，假定初始时构造一个图 T，T 的顶点集为 V，边集为空，即 T=(V, Φ)，可见初始时 T 为 n 个连通分量的非连通图，或 n 棵子树的森林，每个连通分量（子树）只有一个顶点。算法思想简要描述如下：

T=(V, Φ); //初始化生成图 T，用此存放生成树。

while(T 的边数 < n-1)

{

从 N=(V, E)的边集中选取当前最短边（u,v）；

```
        标记 E 中边(u,v)已选，不可再用;
        if( 边(u,v)与 T 中已有边不构成回路)
            将边(u,v)加入到树 T 中;
    }
```

算法结束时，T 成为一个顶点集为 V，n-1 条边的连通图，即连通网 N=(V,E)的生成树，且因为每次都是选择最小权值边，所以 T 为最小生成树。

4. 算法实现及讨论（*）

下面我们先讨论算法实现要解决的相关问题，然后给出一种 Kruskal 算法的实现方案。

（1）边的存储和处理问题

① 边的存储

一般我们用数组来存储边，边的信息包括 3 个部分：2 个顶点和边的权值，为此我们需要定义一个具有 3 个分量的结构体来描述一条边，结构定义如下：

```
typedef struct edgetype
{
    int vBegin;              //边的起始顶点编号，从 1 开始
    int vEnd;                //边的另一顶点编号，从 1 开始
    eInfoType eWeight;   //边的权值
}EdgeType;          //边数组的类型
```

有了这个结构体，我们就可以定义边数组：

```
EdgeType edges[MaxEdgeNum];
```

由于不能预测一个图的边数，我们定义时只能按预定的最大边数值来定义数组长度。

② 边的读取

有了边数组后，我们就可以从原连通网中读取边的信息，并保存的 edges[]数组中。对图的不同表示，读取的边的代码不同，但流程类似，时间复杂度都是 $O(n^2)$。下面给出图的邻接表表示读取边的代码：

```
void GetEdges(Graph &G, EdgeType edges[])
{
    int i;
    int k=0;              //作为边数组的下标
    EdgeNode *p;      //边链表的头指针
    for(i=1;i<=G.VerNum;i++)
    {
        p=G.VerList[i-1].firstEdge;     //读取当前顶点的边链表的头指针
        while(p)
        {                                   //由顶点数据获取顶点编号
            edges[k].vBegin=LocateVertex(G, G.VerList[i-1].data);
            edges[k].vEnd=p->adjVer;   //读取另一顶点编号到 vEnd
            edges[k].eWeight=p->eInfo;
            p=p->next;
            k++;
        }
```

```
    }
}
```

③ 最小边的获取

一种方法是对 edges[]数组按边的权值进行递增排序，这要用到后面介绍的排序算法。还可以在上面②中读取边的同时进行递增插入，读取边结束得到的 edges[]就是递增有序的。edges[]递增有序的话，我们从最小权值逐次往后面读取边即可获得最小边。

另一种方法是我们写一个函数，用循环方法每次从 edges[]中获得最小边。这种方法需要借助一个标记数组，比如 edgeUsed[]，为 BOOL 类型，对已经读取的最小边标记为 TRUE，不可再读取，值为 FALSE 的边是可用的边。下面给出读取最小边的代码：

```
EdgeType GetMinEdge(Graph &G, EdgeType edges[], int edgeUsed[], int &n)
{               //函数值返回读取的最小边
                //n 为返回的最小边在 edges[]数组中的下标
    EdgeType minEdge;
    cellType wMin=INF;   //保存最小权值
    int i;
    int M;   //控制循环次数
    if(G.gKind==UDG || G.gKind==UDN)
        M=G.ArcNum*2;
          //无向网（图），因为对称性，邻接矩阵或邻接表中有效数据是边数的 2 倍
    else
        M=G.ArcNum;      //有向图，有向网中，M 即为图的边数
    for(i=0; i<M ;i++)
    {
        if(edgeUsed[i]==FALSE && edges[i].eWeight<wMin)
        {           //对未使用，且权值较小的边，暂定位最小边，更新相关数组
            wMin=edges[i].eWeight;
            minEdge.eWeight=edges[i].eWeight;;
            minEdge.vBegin=edges[i].vBegin;
            minEdge.vEnd=edges[i].vEnd;
            n=i;
        }
    }
    return minEdge;   //返回取得的最小边
}
```

④ 生成树的边的保存

一种方法是利用上面的边数组 edges[]，把没有用的边删除，剩下的边即为最小生成树的边，但数组的删除操作是耗时的操作，时间性能不好。也可以通过标记方法来标记 edges[]中哪些边是生成树的边。

本书使用一个类型与 edges[]相同的数组来存放生成树的边，即：

EdgeType treeEdges[MaxVerNum];　　//存放生成树中的边信息，n-1 条

（2）判定选择的边是否与已选边形成回路问题

这既是算法重点也是难点问题，需要特殊而巧妙的方法来处理。从前面的分析和讨论我们知道，Kruskal 算法开始时要构建一个没有边的图 T=(V,Φ)，然后逐步选择 n-1 条边加入到 T 中形成最小生成树。所以算法开始时，T 有 n 个连通分量或 n 棵子树，以后每加入一条边都要连接 2 个连通分量（子树），使这 2 个两个连通分量（子树）合并为一个连通分量（子树），T 的连通分量（子树）数减 1。加入 n-1 条边后 T 成为只有一个连通分量的生成树（连通且没回路）。可见**要加入 T 中的边的 2 个顶点应该处于 T 的 2 个不同连通分量（子树）上**。如果边的 2 个顶点落到 T 的同一连通分量上，必然形成回路，而不能成为树。这样通过边的 2 个顶点是否处于同一连通分量，即可判定选择的边是否和已选边形成回路。那么怎么判定呢？这可以通过给 T 中的连通分量编号来完成，具体做法如下：

① 初始时，T 有 n 个连通分量，需要 n 个编号，我们把顶点的编号就作为每个连通分量的编号；

② 选择边时判断是否形成回路：如果边的 2 个顶点的连通分量编号相同，则会形成回路，说明这条边不可用，舍去。

③ 选择的边的 2 个顶点的连通分量编号不同，说明这条边可用，加入 T 中。此边加入 T 后，将把 T 中 2 个连通分量（子树）连接为一个连通分量（子树），因此要把此分量（子树）上所有顶点的连通分量编号更新为同一编号。可选择原来 2 个编号中的一个作为新的编号。如果统一选择较小编号作为新编号，最终生成树的连通编号为 1；如果统一选择较大的编号作为新编号，最终生成树的连通编号将为 n，两种选择皆可以。

最后我们给出 Kruskal 算法的一种实现方案，代码描述如下：

【Kruskal 算法描述】

```
void Kruskal(Graph &G)
{
    int conVerID[MaxVerNum];   //顶点的连通分量编号数组
    EdgeType edges[MaxVerNum*MaxVerNum];   //存放图的所有边信息
    EdgeType treeEdges[MaxVerNum];              //存放生成树中的边信息，n-1 条
    int edgeUsed[MaxVerNum*MaxVerNum];
        //辅助数组，标记 edges[]中的边是否已经用过。1--已用过，0--未用过
        //也可以用排序算法对 edges[]进行排序来完成这个工作
    EdgeType minEdge;   //保存最小边
    int i,j;
    int k=0;
    int conID;                  //边的终止顶点 vEnd 的连通分量的编号
    int n;                      //返回的最小边在 edges[]数组中的下标
    GetEdges( G, edges );       //获取图所有边的信息，存入数组 edges[]
                                //初始化可用边标记数组 edgeUsed []--可用排序算法取代
    int M;                      //循环次数
    if(G.gKind==UDG ||G.gKind==UDN)
        M=G.ArcNum*2; //因为无向图（网），有效数据是边数的 2 倍，所以乘 2
    else
```

```
        M=G.ArcNum;

    for(i=0; i<M; i++)
        edgeUsed[i]=FALSE;   //标记所有边都可用
    for(i=1;i<=G.VerNum;i++)   //初始化连通分量编号。
    {
        conVerID[i-1]=i;          //连通分量编号=顶点编号。注意编号与数组下标差 1
    }
    for(i=1; i<G.VerNum; i++)   //取出 n-1 条边，构成生成树
    {
        minEdge=GetMinEdge(G, edges, edgeUsed, n );   //获取一条最小边
        while(conVerID[minEdge.vBegin-1]==conVerID[minEdge.vEnd-1])
        {                          //如果 minEdge 会形成回路
            edgeUsed[n]=1;   //标记此最小边不可用
            minEdge=GetMinEdge( G, edges, edgeUsed, n ); //继续取下一条最小边
        }
            //到此取得了一条可用最小边，加入生成树中
        treeEdges[i-1]=minEdge;                  //可用最小边加入生成树的边数组
        conID=conVerID[minEdge.vEnd-1];     //取得最小边的终点编号
            //conID=conVerID[minEdge.vBegin-1];
        for(j=1;j<=G.VerNum;j++)                 //合并连通编号到最小编号
        {          //所有连通分量编号为 conID 的顶点，
                   //连通分量编号都改为最小边起始顶点的连通号
            if(conVerID[j-1]==conID)
            {
                 conVerID[j-1]=conVerID[minEdge.vBegin-1];
            }
        }
        edgeUsed[n]=TRUE;     //标记当前选择的边已经用过。
    }
}
```

【算法分析】

Kruskal 算法的时间复杂度分析比较困难。假设连通网 N=(V,E)的顶点数为 n，边数为 e。首先从图中读取边的信息，即执行 GetEdges()函数，的时间复杂度为 $O(n^2)$，对邻接表表示可能稍微好一点；其次，选择 n-1 条最小边，按本书给出的算法的时间复杂度为 O(n*e)。如果通过对边数组 edges[]排序方法求解，最好的排序时间性能为 $O(e\log_2 e)$，但排序中会涉及数组 edges[]的多次读写操作，实际也是相当耗时的，还要加上对 n-1 条边选择操作的时间。对比 Prim 算法，**Kruskal 算法时间与给定连通网的边数 e 相关，Kruskal 算法更适用于边数相对较少（较稀疏）的连通网**。

6.5 最短路径

求解两点之间的最短路径的问题也可转换为图的应用问题。这类问题包含两个典型的问题：

（1）从单个顶点到其余各顶点之间的最短路径。

（2）各顶点之间的最短路径。

针对这两个问题，下面分别介绍 Dijkstra 算法和 Floyd 算法。

6.5.1 从一个顶点到其余各个顶点的最短路径—Dijkstra 算法

1. 算法求解思想及实例

从网中一个指定的顶点到其余各顶点之间的最短路径的求解方法是由 Dijkstra 提出的，其**求解思想**是**按最短路径长度不减的次序求解各顶点的解**，即按由近到远的次序递推求解各顶点的解。

这一提法似乎有些矛盾：既然我们要求解的是到各顶点的最短路径，那么又如何先知道其长度呢？下面我们来对此进行分析，为了叙述方便，我们假设一个带权图 G(V, E)，V 为图的顶点集合，一个集合 S 保存已经求出最短路径的顶点，则未解顶点集合为 V-S。在此，不妨假设起点为 v_0，其编号为 vID。引入辅助数组 path[ ] 用来保存全图的最短路径信息，数组的下标对应顶点的编号，数组的值表示当前顶点的直接前驱的编号，例，path[3]=4，表示编号 3 顶点的直接前驱是编号为 4 的顶点。引入数组 dist[ ] 用来保存全图顶点到指定起点 v_0 的最短距离值，数组下标对应顶点编号，数组的元素值即为距离值，例，dist[4]=18，表示编号为 4 的顶点距离指定顶点 v_0 的最短距离值为 18。

事实上，其求解方法是由部分已知的距 v_0 近的顶点逐渐向远的顶点推进求解的，具体求解方法如下：

（1）初始化：将起点 vID 加入已解集合 S 中。若顶点 vID 与目标顶点 i 相邻，即存在边(vID, i)，则 dist[i]初始化为边(vID, i)的权值；否则 dist[i]初始化为无穷大。将其作为 v_0 到 v_i 的最短路径（当然这只是暂时的）存放到 path[v]中，将其权值作为对应的路径长度存放到 dist[v]中。

（2）从未解顶点中选择一个 dist 值最小的顶点 v，则当前的 path[v]和 dist[v]就是顶点 v 的最终解（从而使 v 成为已解顶点）。

（3）由于某些顶点经过 v 可能会（使得从 v_0 到该顶点）更近一些，因此，应修改这些顶点的路径及其长度的值，即要修改其 path 和 dist 的值。（显然，这些顶点既可能是 v 的直接后继，也可能是 v 的间接后继，但我们只需修改其直接后继的 path 和 dist 的值）

（4）重复（2）（3），直到所有顶点求解完毕。

【例 6.7】求解图 6-33 所示图中从顶点 1 到其余各顶点之间的最短路径。

【解】为直观起见，下面依次画出求解过程中的各步的图的状态及其相应的路径和长度，如表 6-2 所示。其中采用了以下一些约定和符号：

图中的实线表示已确定的解，虚线表示待确定的部分，用黑色背景表示已解顶点。到顶点的最短路径长度标注在顶点边上。

用两个数组 path 和 dist 分别表示从顶点 1 到其余各顶点的最短路径及其长度。

路径及其长度中的已解顶点用框框住，修改前后的值也相应地表示出来。

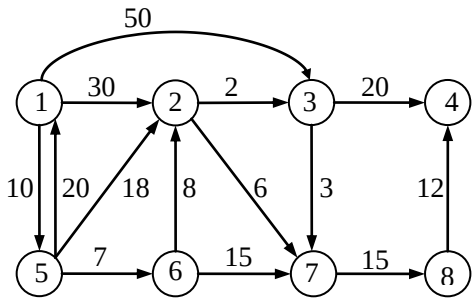
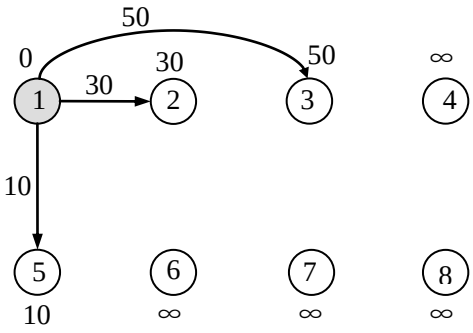
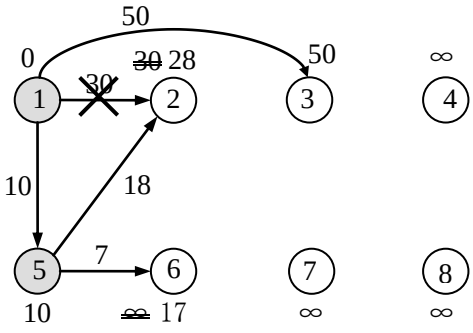
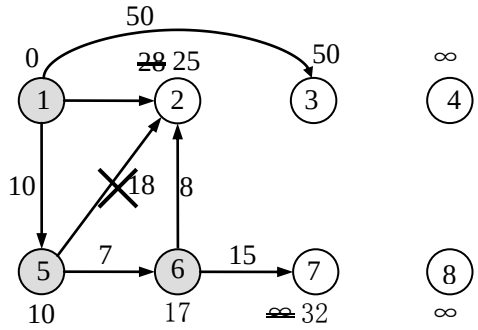
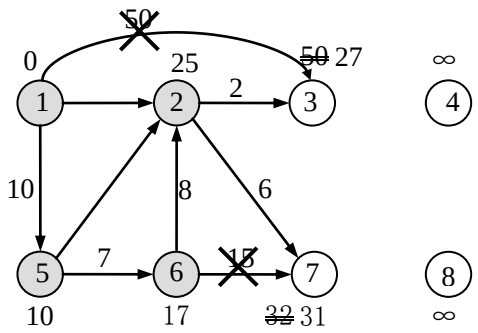
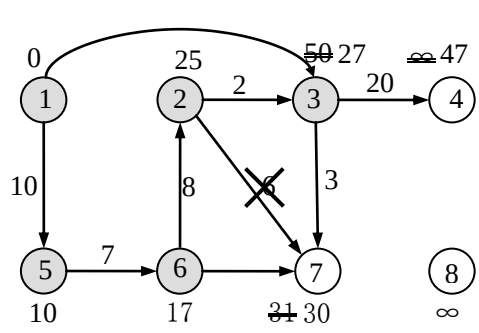
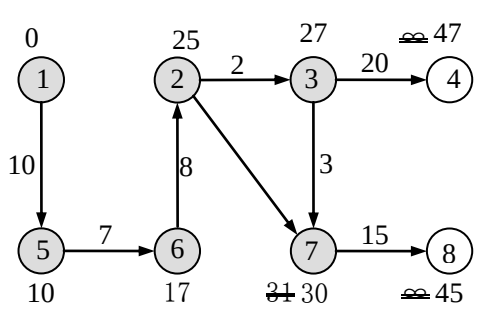
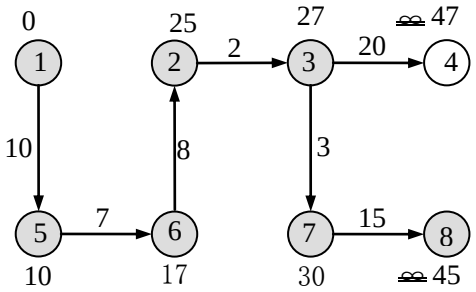
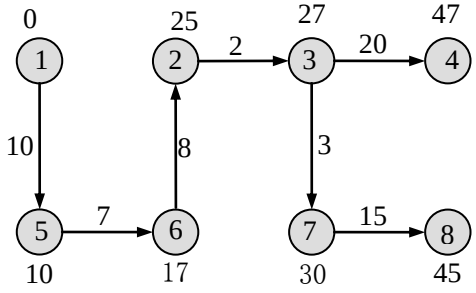


图 6-33 例 6.7 图

表 6-2 Dijkstra 算法求解流程示意表

序号及操作	图状态	各点的最短路径 path	相应长度 dist
第一步：标出顶点 1 直接可达的顶点的路径及其长度。		1 **()** 2 **(1,2)** 3 **(1,3)** 4 **()** 5 **(1,5)** 6 **()** 7 **()** 8 **()**	1 **0** 2 **30** 3 **50** 4 ∞ 5 **10** 6 ∞ 7 ∞ 8 ∞
第二步：顶点 5 最近，故选顶点 5。 修改顶点 5 的直接后继的路径及其长度。		1 **()** 2 ***(1,2)*(1,5,2)** 3 **(1,3)** 4 **()** 5 **(1,5)** 6 ***()*(1,5,6)** 7 **()** 8 **()**	1 **0** 2 ***30*28** 3 **50** 4 ∞ 5 **10** 6 ~~∞~~**17** 7 ∞ 8 ∞

第三步：顶点 6 最近，故选顶点 6。 修改顶点 6 的直接后继的路径及其长度。		1 **()** 2 ***(1,5,2)*****(1,5,6,2)** 3 **(1,3)** 4 **()** 5 **(1,5)** 6 **(1,5,6)** 7 ***θ*** **(1,5,6,7)** 8 **()**	1 **0** 2 ***28*****25** 3 **50** 4 **∞** 5 **10** 6 **17** 7 ~~∞~~32 8 **∞**
第四步：顶点 2 最近，故选顶点 2。 修改顶点 2 的直接后继的路径及其长度。		1 **()** 2 **(1,5,6,2)** 3 ***(1,3)*** **(1,5,6,2,3)** 4 **()** 5 **(1,5)** 6 **(1,5,6)** 7 ***(1567)*****(15627)** 8 **()**	1 **0** 2 **25** 3 ***50*****27** 4 **∞** 5 **10** 6 **17** 7 ~~*32*~~31 8 **∞**
第五步：顶点 3 最近，故选顶点 3。 修改顶点 3 的直接后继的路径及其长度。		1 **()** 2 **(1,5,6,2)** 3 **(1,5,6,2,3)** 4 ***θ*** **(1,5,6,2,3,4)** 5 **(1,5)** 6 **(1,5,6)** 7 ***(15627)*****(156237)** 8 **()**	1 **0** 2 **25** 3 **27** 4 ~~∞~~47 5 **10** 6 **17** 7 ~~*31*~~30 8 **32**
第六步：顶点 7 最近，故选顶点 7。 修改顶点 7 的直接后继的路径及其长度。		1 **()** 2 **(1,5,6,2)** 3 **(1,5,6,2,3)** 4 **(1,5,6,2,3,4)** 5 **(1,5)** 6 **(1,5,6)** 7 **(156237)** 8 ***θ*** **(1,5,6,2,3,7,8)**	1 **0** 2 **25** 3 **27** 4 **47** 5 **10** 6 **17** 7 **30** 8 ~~∞~~45

第七步：选顶点8。顶点8的直接后继的路径及其长度未作修改。		1 **()** 2 **(1,5,6,2)** 3 **(1,5,6,2,3)** 4 **(1,5,6,2,3,4)** 5 **(1,5)** 6 **(1,5,6)** 7 **(156237)** 8 **(1,5,6,2,3,7,8)**	1 **0** 2 **25** 3 **27** 4 **47** 5 **10** 6 **17** 7 **30** 8 **45**
第八步：选顶点4。全部求解完毕		1 **()** 2 **(1,5,6,2)** 3 **(1,5,6,2,3)** 4 **(1,5,6,2,3,4)** 5 **(1,5)** 6 **(1,5,6)** 7 **(156237)** 8 **(1,5,6,2,3,7,8)**	1 **0** 2 **25** 3 **27** 4 **47** 5 **10** 6 **17** 7 **30** 8 **45**

【说明】由例可知，用图形化方式来表示到各顶点的最短路径较为直观，因此，建议读者在平常练习及考试时，可将这种形式的图（事实上，在许多情况下可以说是一棵生成树）作为结果的一部分。

需要清楚的是求出了顶点 v_0 和 v_i 之间的最短路径后，中间途经的所有顶点都在相应的最短路径上。例如，图 6-33 中，顶点 1 到顶点 8 的最短路径是 1->5->6->2->3->7->8，途经的顶点 5、6、2、3、7，也在顶点 1 到相应顶点的最短路径上，即 1->5->6->2->3->7 是 1 到 7 的最短路径；1->5->6->2->3 是 1 到 3 的最短路径，等等。

2．算法实现的讨论

下面讨论最短路径算法的实现。由算法描述可知，为实现这一算法，需要解决以下问题：

（1）记录从起点到各顶点的最短路径及其长度。

（2）表示各顶点是否已经求解的标志。

（3）搜索下一个求解的顶点，即在未解顶点中搜索出（距离起点）最近的顶点。

（4）修改所搜索出的顶点的后继的最短路径及其长度。

下面依次简要讨论上述各问题的实现。

（1）求解结果的表示和存储：如前所述，引入两个一维数组 path[]和 dist[]。path[]数组存储最短路径信息，其下标对应网中的顶点编号，元素值表示下标对应顶点的直接前驱顶点，例，path[3]=4，表示顶点 3 的直接前驱顶点为 4。Dist[]数组存指定顶点到各个目标顶点的距离值，数组下标也为顶点编号，例，dist[3]=18，表示指定顶点 v_0 到编号为 3 的顶点的最短

距离值为 18。（注：一般情况下数组下标和顶点编号差 1，数组下标往往从 0 开始，顶点编号从 1 开始，算法实现时需要注意。）

（2）标记各顶点是否已经求解：可引入一个一维标记数组 solved[]，数组下标对应顶点编号，数组元素值为 1，表示已求出最短路径，0 表示尚未求解，例，solved[3]=1，solved[4]=0 表示编号为 3 的顶点已经求解，编号 4 的顶点尚未求解。

（3）搜索下一个求解顶点 v：就是在未解顶点中搜索出 dist 值最小的一个顶点。

（4）修改所搜索出的顶点 v 的直接后继顶点的最短路径及其长度。

【Dijkstra 算法—基于邻接矩阵】

```
//**************  Dijkstra 算法--基于邻接矩阵 ***************//
//* 函数功能：给定顶点，求解此点与网中其它顶点的最短路径     *//
//* 入口参数：Graph G，待访问的网（图）                      *//
//*           int vID，指定顶点的编号                        *//
//* 出口参数：int path[]，返回最短路径信息                   *//
//*           int dist[]，返回最短距离值                     *//
//* 返 回 值：无                                             *//
//************************************************************//
void Dijkstra( Graph &G, int path[], int dist[], int vID )
{
        //数组 path[]保存最短路径信息，并返回到调用程序
        //数组 dist[]保存最短路径距离值，并返回到调用程序
        //vID 为指定的起始顶点编号
    int solved[MaxVerNum];  //标记顶点是否已经求出最短路径。1--已求解，0--未求解
    int i, j, v;
    cellType minDist;    //最短距离，cellType 为自定义的邻接矩阵中元素的数据类型
        //初始化集合 S 和距离向量
    for(i=1;i<=G.VerNum;i++)
    {
        solved[i-1]=0;          //标记所有顶点均未求解
        dist[i-1]=G.AdjMatrix[vID-1][i-1];
        if(dist[i-1]!=INF)
            path[i-1]=vID;   //第 i 顶点的前驱为 vID
        else
            path[i-1]=-1;    //当前顶点 i 无前驱
    }
    solved[vID-1]=1; //标记顶点 vID 已求解
    dist[vID-1]=0;    //vID 到自身的距离为 0
    path[vID-1]=-1;   //vID 为起始顶点，无前驱
        //依次找出其它 n-1 个顶点加入已求解集合 S
    for(i=1; i<G.VerNum; i++)
    {
```

```
        minDist=INF;
            //在未解顶点中寻找距 vID 距离最近的顶点，编号保存到 v。
        for(j=1;j<=G.VerNum;j++)
        {
            if(solved[j-1]==0 && dist[j-1]<minDist)   //j 目前尚在 V-S 中，为未解顶点
            {
                v=j;
                minDist=dist[j-1];
            }
        }
        if(minDist==INF)
            return;
                                //输出本次选择的顶点距离
        cout<<"选择顶点："<<G.Data[v-1]<<"--距离："<<minDist<<endl;
        solved[v-1]=1;          //顶点 v 已找到最短距离，标记为已解顶点
        //对选中的顶点 v，修改未解顶点集 V-S 中，v 的直接后继到顶点 vID 的距离
        for(j=1; j<=G.VerNum; j++)
        {
            if(solved[j-1]==0 && (minDist+G.AdjMatrix[v-1][j-1])<dist[j-1])
            {
                                    //更新顶点 j 到顶点 vID 的最短距离。
                dist[j-1]=minDist+G.AdjMatrix[v-1][j-1];
                path[j-1]=v;    //更新顶点 j 的直接前驱为顶点 v
            }
        }
    }
}
```

【Dijkstra 算法—基于邻接表】

```
void Dijkstra(Graph &G, int path[], int dist[], int vID)
{
    int solved[MaxVerNum];
    int i, j, v;
    eInfoType minDist;   //保存最短距离值，eInfoType 为自定义的边的权值类型
    EdgeNode *p;         //指向边链表结点的指针，EdgeNode 为边链表结点结构类型

        //初始化已求解集合 S，距离数组 dist[]，路径数组 path[]
    for(i=1;i<=G.VerNum;i++)
    {
        solved[i-1]=0;   //所有顶点均未处理
        dist[i-1]=INF;   //所有顶点初始距离置为无穷大（INF）
```

```
        path[i-1]=-1;     //所有顶点的前驱置为-1，即无前驱
}
        //处理顶点 vID
solved[vID-1]=1;   //标记 vID 已经处理
dist[vID-1]=0;
path[vID-1]=-1;
        //从邻接表初始化 dist[]和 path[]
p=G.VerList[vID-1].firstEdge;   //顶点 vID 的边链表头指针
while(p)
{
        v=p->adjVer;            //取得顶点 vID 的邻接顶点编号
        dist[v-1]=p->eInfo;   //取得 vID 与 v 之间边的权值，赋给 dist[v-1]
        path[v-1]=vID;        //顶点 v 的前驱为 vID
        p=p->next;
}
        //依次找出余下 n-1 个顶点加入已求解集合 S 中
for(i=1;i<G.VerNum;i++)
{
        minDist=INF;
            //在未解顶点中寻找距 vID 距离最近的顶点，编号保存到 v。
        for(j=1;j<=G.VerNum;j++)
        {
            if(solved[j-1]==0 && dist[j-1]<minDist)
            {
                v=j;      //j 为未解顶点集 V-S 中候选的距离 vID 最近的顶点
                minDist=dist[j-1];
            }
        }
            //已解顶点集 S 与未解顶点集 V-S 没有相邻的顶点，算法退出
        if(minDist==INF)
            return;
            //输出本次选择的顶点距离
        cout<<"选择顶点："<<G.VerList[v-1].data<<"--距离："<<minDist<<endl;
        solved[v-1]=1;   //标记顶点 v 以找到最短距离，加入集合 S 中

        //对选中的顶点 v，更新集合 V-S 中所有与 v 邻接的顶点距离 vID 的距离
        p=G.VerList[v-1].firstEdge;   //取得 v 的边链表的头指针
        while(p)
        {
            j=p->adjVer;   //取得 v 的邻接顶点编号，保存到 j
            if(solved[j-1]==0 && minDist+p->eInfo<dist[j-1])
```

```
                {
                    dist[j-1]=minDist+p->eInfo;   //更新顶点 j 到顶点 vID 的最小距离
                    path[j-1]=v;   //j 的前驱改为顶点 v
                }
                p=p->next;
            }
        }
    }
```

【Dijkstra 算法的输出】

调用上面给出的 Dijkstra 算法，返回最短路径信息 path 和最短距离值 dist。输出各个目标顶点到指定顶点 vID 的最短距离非常简单，直接输出 dist[i]即可。要输出指定顶点 vID 到目标顶点 i 之间的最短路径，即最短路径经过的顶点序列则稍微复杂一点，因为 path[i]给出的是全图最短路径上顶点 i 的直接前驱。我们要输出 vID 到 i 最短路径的顶点序列，可以从 i 开始，求出其直接前驱 vPre=path[i]，再循环求出其前驱的前驱 vPre=path[vPre]，直到 vID（没有前驱）。在此过程中还需要一个临时结构保存 vID 到 i 途径的顶点，可以借助栈或数组来保存顶点信息。下面给出一段借助数组保存顶点信息，输出最短路径的代码。

```
    void PrintDijkstra(Graph &G, int path[], int dist[], int vID )
    {
        int sPath[MaxVerNum];   //数组 sPath[]保存 vID 到目标顶点 i 的最短路径顶点
        int vPre;       //前驱结点编号
        int top=-1;   //保存最短路径上的顶点个数，以控制输出最短路径
        int i, j;

        for(i=1; i<=G.VerNum; i++)
        {
            cout<<G.Data[vID-1]<<" to "<<G.Data[i-1];
            if(dist[i-1]==INF)
                cout<<"  无可达路径。"<<endl;
            else
            {
                cout<<"  最短距离："<<dist[i-1]<<endl;
                cout<<"                路径：";
            }

            top++;
            sPath[top]=i;       //sPath[0]保存目标顶点编号 i
            vPre=path[i-1];   //取得顶点 i 的直接前驱编号，赋给 vPre
                //从第 i 个顶点，迭代求前驱顶点，直到 vID，保存最短路径到 sPath[]
            while(vPre!=-1)
            {
```

```
            top++;
            sPath[top]=vPre;
            vPre=path[vPre-1];
        }
            //如果最短路径存在，依次打印从 vID 到 i 顶点的最短路径顶点序列
        if(dist[i-1]!=INF)
        {
            For (j=top; j>=0; j-- )   //sPath[top]为指定的起始顶点 vID
            {
                cout<<G.Data[sPath[j]-1]<<"    ";
            }
        }

        top=-1;   //初始化 top，以处理下一个目标顶点
        cout<<endl;
    }
}
```

【思考问题】本节我们讨论了带权图求解最短的算法，那么如何改造这些算法来求带权图中两个顶点之间的最长路径呢？

【算法分析】

上面算法中，主循环执行 n-1 次，每次循环都要在未解顶点循环找出距离起始顶点最近的顶点，执行 n 次，所以 Dijkstra 算法时间复杂度：$O(n^2)$。

有时我们要求解的问题是指定起始顶点到指定目标顶点的距离，但只要使用 Dijkstra 算法，时间复杂度仍为：$O(n^2)$。

6.5.2 每一对顶点之间的最短路径—Floyd 算法

每一对顶点之间的最短路径指：对给定的连通网 N=(V,E)，求出网中任意一对顶点之间的距离。我们前面学习了 Dijkstra 算法，我们最容易想到的解决方法可能是循环使用 Dijkstra 算法，指定网中每个顶点作为起始顶点，这个方法的确是可行的，代码如下：

```
for( int i=1; i<=G.VerNum; i++)
{
    Dijkstra( G,  path,  dist,  i  );  //循环调用 Dijkstra 算法，i 作为起始顶点
}
```

这个算法的时间复杂度为 $O(n^3)$。

下面我们介绍解决这个问题另一种解决方案：Floyd 算法。

1. Floyd 算法思想

Floyd 算法使用一个二维的顶点之间的距离矩阵，设为 D，数组下标为顶点编号，数组元素值即为下标对应顶点之间的最短距离值，简单见我们用 D_{ij} 表示顶点 i 和 j 之间的最短距离。我们将距离矩阵 D 初始化为连通网的邻接矩阵，记为 $D^{(0)}$，此时，如果两个顶点 i、j

之间有边（弧）(i,j)，则 $D^{(0)}_{ij}$ 即为边(i,j)的权值，否则为无穷大。然后尝试用顶点 1 作为中间跳点，看看从 i 先到顶点 1，再从 1 到顶点 j，路径距离是否变短，如果变短，即 $D^{(0)}_{ij}$〉$D^{(0)}_{i1}+D^{(0)}_{1j}$，则选择顶点 1 为中间跳点，用 $D^{(0)}_{i1}+D^{(0)}_{1j}$ 去更新 $D^{(0)}_{ij}$；否则不变。更新结束得到新的距离矩阵 $D^{(1)}$。接着再用顶点 2 作跳点，更新距离矩阵得到 $D^{(2)}$。按编号依次尝试图中的每个顶点作为中间跳点，更新距离矩阵，直到最后一个编号顶点 n，得到最终的距离矩阵 $D^{(n)}$，存放的就是图中每对顶点之间的最短距离。求解过程距离矩阵的递推序列是：

$D^{(0)}$，$D^{(1)}$，$D^{(2)}$，... ，$D^{(n-1)}$，$D^{(n)}$。

下面我们讨论一下一般情况：假定我们已经得到 $D^{(m-1)}$，我们现在用顶点 m 作跳点，递推得到 $D^{(m)}$。如图 6-34 所示：

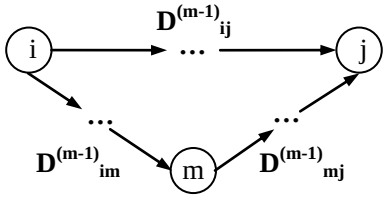


图 6-34　m 作为跳点示意

顶点 m-1 作为跳点更新距离矩阵为 $D^{(m-1)}$，其中，i 到 j 的最短距离为 $D^{(m-1)}_{ij}$，i 到 m 的最短距离为 $D^{(m-1)}_{im}$，m 到 j 的最短为 $D^{(m-1)}_{mj}$。现在尝试以顶点 m 为跳点的话，要判断 i 经 m 到达 j 的距离与不经过 m 的距离是否变短。分为下面两种情况：

① i->m->j 的距离比 i->j 距离更短

即：$D^{(m-1)}_{im}+D^{(m-1)}_{mj}<D^{(m-1)}_{ij}$，接受 m 最为跳点，即 i->m->j 为当前最短路径，距离更新为：$D^{(m)}_{ij}=D^{(m-1)}_{im}+D^{(m-1)}_{mj}$ 。

② i->m->j 的距离比 i->j 距离更长或不变

即：$D^{(m-1)}_{im}+D^{(m-1)}_{mj}>=D^{(m-1)}_{ij}$，则不接受 m 最为跳点，维持原来的 i->j 最短路径不变，距离更新为：$D^{(m)}_{ij}=D^{(m-1)}_{ij}$ 。

留意一下就会注意到，我们在尝试跳点时，是按顶点编号依次尝试的，所以当尝试顶点 m 作为跳点时，i 到 j 的路径上，除了 i 和 j 外，中间途经的顶点的编号都小于 m。

为了保存两个顶点之间的最短路径信息，我们还需要一个二维数组 path，其详细构成后面介绍，下面我们简单描述一下 Floyd 算法流程：

距离矩阵 D 初始化为连通网的邻接矩阵；

初始化路径矩阵 path；

```
for( m=1; m<=n; m++)
{
     以 m 为跳点，更新距离矩阵 D；
    更新路径矩阵 path；
}
```

2. Floyd 算法实现及讨论

① 二维距离矩阵

前面已经详细讨论，在后面实现中我们使用 dist[][]数组作为距离矩阵，dist[i][j]

为顶点 i 和 j 之间的最短距离。还有实现时距离数组递推更新在同一个数组上完成，因为得到 $D^{(m)}$时，$D^{(m-1)}$就没用了。

② 二维路径数组

两个顶点之间最短路径中间途径的顶点数量是不同的，所以要保存最短路径上途径的顶点是有一定难度的，需要特别的设计。我们使用一个二维的路径数组 path[][]，其中**数组元素 path[i][j]保存的是 i 到 j 最短路径终点 j 的直接前驱顶点的编号**(i 到 j 路径上顶点序列中 j 的直接前驱，即 j 的前一个顶点)。

初始化时，如果 i 和 j 之间有边（弧），那么 i 就是 j 的前驱，即 path[i][j]=i；如果 i 和 j 之间没有边(弧)相连，或 i==j，则初始化为-1，标记没有前驱顶点，写成 path[i][j]=-1 及 path[i][i]=-1。

当尝试顶点 m 作为跳点时，**如果接受 m 为跳点，则用 m 到 j 路径上 j 的前驱去更新原来 i 到 j 路径上 j 的前驱，即 path[i][j]=path[m][j]**；否则，不接受 m 为跳点，j 的前驱不变。

算法结束时，由 path 求取的两个顶点 i 和 j 之间最短路径上顶点序列的方法：

首先由 path[i][j]可以求得 j 的前驱，不妨设编号为 x_1；有了 x_1，我们又可以通过 path[i][x_1]求出路径上 x_1的前驱，不妨设为 x_2；这个过程持续下去，直到路径某个顶点，不妨设为 x_k，的前驱为 i，即 path[i][x_k]==i。这样就求出了 i 到 j 路径上的所有顶点。简单地说就是：通过求 j 的前驱，再求前驱的前驱，持续下去直到 i。

下面给出 Floyd 算法实现，本算法是基于图的邻接矩阵表示的。因为算法是通过邻接矩阵的递推更新完成的，所以必须要有邻接矩阵。如果图以邻接表表示，算法的不同之处只在于邻接矩阵的获取。算法描述如下：

【Floyd 算法描述】

```
void Floyd(Graph &G, cellType dist[MaxVerNum][MaxVerNum], int path[MaxVerNum][MaxVerNum])
{
    int i,j,m;
    for(i=1;i<=G.VerNum;i++)  //初始化距离矩阵和路径矩阵
    {
        for(j=1;j<=G.VerNum;j++)
        {             //距离矩阵初始化为邻接矩阵
            dist[i-1][j-1]=G.AdjMatrix[i-1][j-1];
                    //初始化路径矩阵，路径矩阵元素 path[i-1][j-1]中
                    //保存编号 j 顶点的前驱的顶点编号
                    //如果 i,j 之间存在边(弧)，则 j 的前驱为 i。否则前驱置为-1
            if( i!=j && G.AdjMatrix[i-1][j-1]<INF)
                path[i-1][j-1]=i;
            else
                path[i-1][j-1]=-1;
        }
    }
```

```
            //下面是 Floyd 算法的核心 -- 三重 for 循环
        for(m=1; m<=G.VerNum; m++)  //注意外层循环必须为中转跳点选择循环
        {
            for(i=1; i<=G.VerNum; i++)
            {
                for(j=1; j<=G.VerNum;j++)
                {         //m 作为跳点，i、j 之间距离变小，接收 m 作为中转点
                    if(i!=j && dist[i-1][m-1]+dist[m-1][j-1]<dist[i-1][j-1])
                    {                       //更新最短距离
                        dist[i-1][j-1]=dist[i-1][m-1]+dist[m-1][j-1];
                                            //更新路径，以 m->j 路径 j 的前驱更新
                                            //原来 i->j 路径 j 的前驱
                        path[i-1][j-1]=path[m-1][j-1];
                    }
                }
            }
        }
    }
```

【算法分析】

从算法中的三重 for 循环直接可以看出算法的时间复杂度为：$O(n^3)$。

6.6 有向无环图

有向无环图（Directed Acycline Graph，DAG）即不存在回路的有向图。这是工程领域应用较多的一种图结构。由于一个工程通常包含多个子工程（或叫做活动），而且子工程之间存在这制约关系，因此，在工程应用领域，最关心的问题有一下两类：

一个工程能否顺利进行？即所包含的子工程之间是否存在相互制约，而导致工程不能顺利进行。

一个工程至少需要多长时间？其中哪些活动是影响工程成败的关键？

针对上述问题，常规的做法是采用图结构来描述工程，从而形成更一般形式的问题求解方法。下面给出针对这两类问题的求解算法：拓扑排序和关键路径求解算法。

6.6.1 拓扑排序

1. 问题描述

拓扑排序（Topological Sort）是从工程领域中抽象出来的问题求解方法。一般来说，一个工程大多由若干项子工程（或活动，后面就用活动来代替子工程）组成，各项活动之间存在一定的制约关系。

例如，将大学的专业学习作为一个为期几年的工程，其中所计划的各门课程及教学环节

分别作为一项活动。很显然，各教学环节之间存在先后次序关系。例如，在计算机专业的学习过程中，学习《数据结构》课程需要有《高级语言程序设计》课程的知识，否则就很难进行。正因如此，需要制定合理的专业教学计划。仔细观察和体会，你会发现在生活和工作中有很多存在这种制约关系的实例。

然而，有些工程中存在的制约关系使得工程难以正常进行下去，例如，一共工程中有 A、B、C 三件事要做，但相互间存在这样的制约关系：A 完成之后才能到 B，B 完成之后才能到 C，C 完成之后才能到 A。很显然，这种制约关系将导致工程无法进行。

因此，对于工程中的这类问题，我们感兴趣的是：一个工程能否顺利进行下去？也就是说，**工程中的各活动间的制约关系是否会导致工程不能正常进行**？

为求解此类问题，首先用图来表示工程：**用顶点表示活动；用弧表示活动间的制约关系**。

例如，前面所描述的 A、B 和 C 间的制约关系如图 6-35 所示：

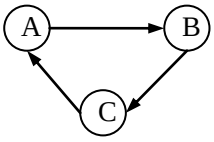


图 6-35 A、B、C 对应关系图

称这种图为 AOV 网（Activity On Vertex）。在 AOV 网这种结构中，判断工程能否顺利进行的问题就变成了判断 **AOV 网中是否存在有向回路**的问题。接下来的问题是：如何判断 AOV 网中是否存在有向回路？

对此问题，可能有读者会想到用深度优先搜索遍历的方法：若从某顶点出发按深度遍历方式遍历能绕回来，则可断定存在有向回路。这种方法显然存在不足：若所选择的起点不在回路中，就不能绕回来。

对这一问题的求解是通过产生满足如下条件的（包含所有顶点的）顶点序列来实现的：

若图（即 AOV **网中**）中顶点 V_i 到顶点 V_j 之间存在路径，则在序列中顶点 V_i 领先于顶点 V_j。

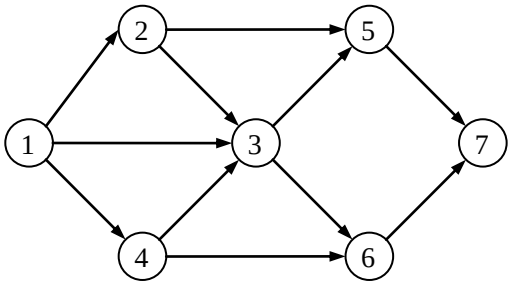
称满足上述条件的顶点序列为**拓扑序列**，称产生这一序列的过程为**拓扑排序**。也就是说，判断 AOV 网中是否存在有向回路的问题变成了拓扑排序的问题：如果拓扑排序能输出所有顶点，则说明不存在回路，否则就存在回路。拓扑排序如何进行？方法如下：

① 找出一个入度为 0 的顶点 V，输出（作为序列中的第一个元素）。

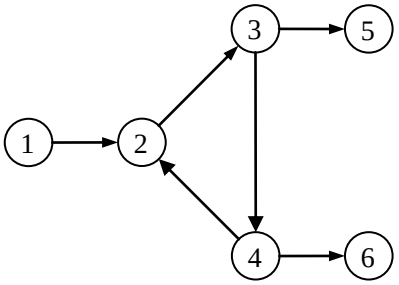
② 删除顶点 V 及其相关的弧（因而使其后继顶点的入度减 1，并可能出现新的入度为 0 的顶点）。

③ 重复①、②，直到找不到入度为 0 的顶点为止。

经过上述操作之后，若所有顶点都被输出了，则说明 AOV 网中不存在回路，否则存在回路。例如，可求出图 6-36 所示的 AOV 网的一个拓扑序列为 1，2，4，3，5，6，7，因而不存在回路。显然，其拓扑序列不唯一，你能否求解出其所有的拓扑序列？



（a）无回路的 AOV 网



（b）有回路的 AOV 网

图 6-36 AOV 网示例

2. 拓扑排序方法及实现

下面讨论拓扑排序方法的实现：

① 由于求解方法中涉及到“入度”，因而需要保存各顶点的入度，为此，不妨采用一个入度数组 ind。为简便起见，假设 ind 中的各元素的值已经设置好了。

② 为实现 ① 中的“寻找入度为 0 的顶点”的操作，有两种典型的方法：

（a）在 ind 数组中搜索－－这种方法不理想：一方面要花费较多的搜索时间，另一方面还要区分顶点是否已经被输出。

（b）将入度为 0 并且未输出的顶点放在一个结构中，需要时就直接从中取出，而不必搜索，从而节省搜索时间。这样，当出现新的入度为 0 的顶点时，就需要将其存放进来。符合这一要求的结构有栈、队列等线性结构，经典方法是用栈，因而本书也用栈。为此，需要为栈配套相应的操作集合，如置空、判断栈空、入栈、出栈等。

③ 步骤 ② 中“删除”顶点的实现：有些初学者看到这一操作便想着如何在存储结构上实现删除顶点的操作，这不方便，同时也没有必要。完整分析拓扑排序的方法发现，确定一个顶点是否能被输出的条件是其入度是否为 0，“删除顶点”的目的并非是真的要将其去掉，其真实目的是为了使其后继顶点少一个前趋（即入度减 1）。由此可知，删除顶点的实现可通过将其所有后继顶点的入度减 1 来实现。

综合上述讨论，可得到拓扑排序方法的细化描述：

① 初始化空栈 S。

② 将 AOV 网中所有入度为 0 的顶点压入栈 S 中。

③ 若 S 不空，则 V=POP(S)，并输出 V。

④ 将 V 的每个后继的入度减 1，若其中某个后继的入度变成了 0，则将其压入栈 S 中。

⑤ 转③。

按这一方法，可得到图 6-36(a)所示 AOV 网的拓扑序列为 1，4，2，3，6，5，7（设每个顶点的邻接点均按从小到大的次序排列）。其执行过程中各步的入度数组、栈的状态和后继操作如图 6-37 所示。其中，为醒目起见，将入度数组中发生变化的入度值框住。输出序列就是出栈的序列。

初始状态		第二步		第三步		第四步	
入度	栈	入度	栈	入度	栈	入度	栈

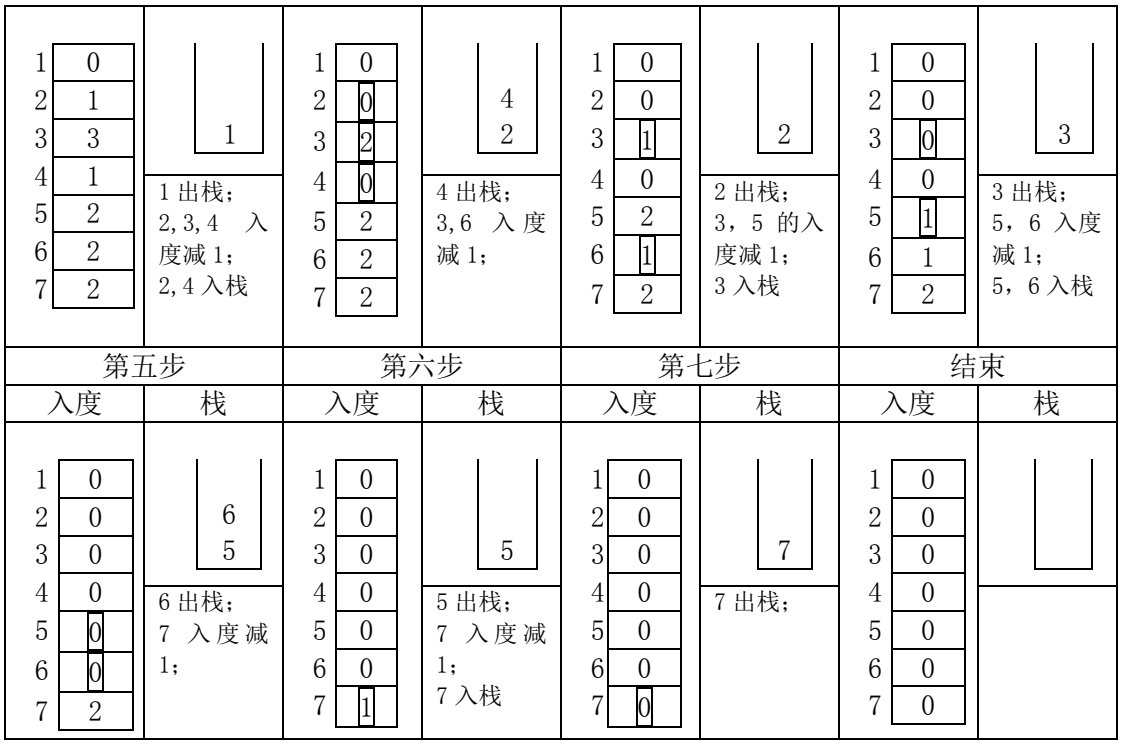


图 6-37 拓扑排序求解过程示例

3. 拓扑排序算法实现

下面给出完整的拓扑排序算法，给出的算法是借助栈实现的。

【算法框架】

```
int TopologicalSort(Graph &G)
{                   //产生图 G 的拓扑序列，并给出是否存在有向回路的判断
    Stack S;   //定义一个栈，保存入度为 0 的顶点
                //求图中各个顶点的入度，存入入度数组 ind
    GetInDegrees(G, ind);
    initStack(S);                    //初始化栈
    for(i=1;i<=G.VerNum;i++)     //入度为 0 的顶点入栈
    {
        if(inds[i-1]==0)
            pushStack(S,i);
    }
    int vCount=0;                 //定义变量 vCount 用于记录输出的顶点数
    while(!stackEmpty(S))
    {
        popStack(S,v);          //从栈顶弹出一个入度为 0 的顶点编号到 v
        visit(v);                //访问取出的顶点 v
        vCount++;                    //已处理顶点（入度为 0）数加 1
        w=firstAdj(G,v);             //对 v 的邻接（后继）顶点的入度减 1
```

```
        while(w!=0)
        {
            ind[w-1]--;                 //v 的邻接顶点 w 的入度减 1
            if(inds[w-1]==0)            //顶点 w 的入度已经为 0，入栈
                pushStack(S,w);
            w=nextAdj(G,v,w);           //取得 v 的下一个邻接（后继）顶点
        }
    }

    if(vCount==n)   //n 为图的顶点数
        return 1;   //返回无回路标记
    else
        return 0;   //有回路，不能产生拓扑序列
}
```

【基于邻接矩阵的一种实现】

```
                                        //拓扑排序算法—基于邻接矩阵表示，使用栈。
int TopologicalSortS(Graph &G, int topoList[])
{                                       //topoList[]数组用于存放拓扑序列
    int inds[MaxVerNum];                //定义顶点入度数组
    seqStack S;                         //定义一个顺序栈，保存入度为 0 的顶点
    int i;
    int v;                              //顶点编号，从 1 开始
    int vCount=0;                       //记录顶点入度为 0 的顶点数
    initStack(S);                       //初始化栈
    for(i=0;i<G.VerNum;i++)             //入度数组初始化
        inds[i]=0;
    for(i=1;i<G.VerNum;i++)             //拓扑序列数组初始化
        topoList[i-1]=-1;               //初始化顶点编号为-1
    GetInDegrees(G, inds);              //从邻接矩阵获取图中各个顶点的初始入度
    for(i=1;i<=G.VerNum;i++)            //入度为 0 的顶点入栈
    {
        if(inds[i-1]==0)
            pushStack(S,i);
    }
    while(!stackEmpty(S))
    {
        popStack(S,v);                  //从栈顶弹出一个入度为 0 的顶点编号到 v
        topoList[vCount]=v;             //当前入度为 0 顶点 v，加入拓扑序列
        vCount++;                       //已处理顶点（入度为 0）数加 1
        for(i=1;i<=G.VerNum;i++)        //与 v 邻接的顶点的入度减 1
```

```
        {
            if(G.AdjMatrix[v-1][i-1]>=1 && G.AdjMatrix[v-1][i-1]<INF && inds[i-1]>0)
            {
                inds[i-1]--;              //与 v 邻接的顶点 i 的入度减 1
                if(inds[i-1]==0)          //顶点 i 的入度已经为 0，入栈
                    pushStack(S,i);
            }
        }
    }
    if(vCount==G.VerNum)                  //G.VerNum 为图的顶点数
        return 1;   //返回无回路标记
    else
        return 0;   //有回路，不能产生拓扑序列
}
```

【基于邻接表的一种实现】

```
                                          //拓扑排序算法—基于邻接表表示，使用栈。
int TopologicalSortS(Graph &G, int topoList[])
{                                         //topoList[]数组保存拓扑排序序列
    int inds[MaxVerNum];                  //保存图中各个顶点的入度
    seqStack S;                           //定义栈，保存入度为 0 的顶点
    int i;
    int v;                                //保存顶点编号，编号从 1 开始
    int vCount=0;                         //记录入度为 0 的顶点数
    EdgeNode *p;                          //边链表结点指针
    initStack(S);                         //初始化栈
    for(i=1;i<=G.VerNum;i++)              //初始化数组 inds[]和 topoList[]
    {
        inds[i-1]=0;                      //每个顶点入度初始化为 0
        topoList[i-1]=-1;                 //拓扑序列初始化为-1
    }
    GetInDegrees(G,inds);                 //从邻接表读取顶点的初始入度
    for(i=1;i<=G.VerNum;i++)              //入度为 0 的顶点编号入栈
    {
        if(inds[i-1]==0)
            pushStack(S,i);
    }
    while(!stackEmpty(S))                 //依次弹出入度为 0 的顶点，将其邻接点入度减 1
    {
        popStack(S,v);                    //弹出一个入度 0 顶点到 v
        topoList[vCount]=v;               //顶点 v 存入拓扑序列
```

```
            vCount++;                    //入度为 0 顶点数加 1
            p=G.VerList[v-1].firstEdge;  //与 v 邻接的顶点入度减 1
            while(p)
            {
                v=p->adjVer;             //依次取出邻接点
                inds[v-1]--;             //邻接顶点入度减 1
                if(inds[v-1]==0)         //如果入度减 1 后变为 0，顶点 v 入栈
                    pushStack(S,v);
                p=p->next;
            }
        }

        if(vCount==G.VerNum)             // G.VerNum 为图的顶点个数
            return 1;                    //拓扑排序成功
        else
            return 0;                    //存在回路，拓扑排序失败。
    }
```

【算法分析】

由算法可知，整个算法要循环 n 次以输出每个顶点，其中在每一次循环中，每个顶点无需搜索。在输出每个顶点后，要对其所有的邻接（后继）顶点的入度减 1，因此，搜索其邻接顶点是花费时间最多的部分，并且所需的时间与深度优先搜索遍历算法类似，取决于存储结构：

① 若采用邻接矩阵存储图，算法的时间复杂度为 $O(n^2)$。

② 若采用邻接表存储图，算法的时间复杂度为 O(n+e)。

【思考问题】

① 不使用栈，使用队列如何实现拓扑排序算法？

② 既不使用栈，又不使用队列，如何实现算法？

6.6.2 关键路径

1. 问题描述

关键路径也是从工程领域抽象出来的一类问题，这类问题的求解常用于工程完成所需时间的估算，包括**完成整个工程需要多少时间？其中哪些子工程是影响工程进度的关键？**

为求解此类问题，采用如下形式的图结构来表示工程：**用弧（有向边）表示活动（子工程）；用弧的权值表示活动的持续时间；用弧 2 端的顶点分别表示活动的开始和结束，叫做事件（即工程中的瞬间行）。**

我们称这样的有向图为 **AOE 网**（Activity On Edge）。显然，一个能正常进行的工程所对应的 AOE 网是一个有向无环图。由于整个工程通常有一个唯一的开始时间和结束时间，因此，相应地，在 AOE 网中分别对应一个顶点，称开始点（即入度为 0 的顶点）为**源点**，

称结束点（出度为 0 的顶点）为**汇点**。

例如，在图 6-38 所示的 AOE 网中，有 15 个活动（子工程），分别编号为 a_1~a_{15}；每个活动所需的持续时间标注在相应的活动上；有 10 个事件，即 10 个顶点，顶点（事件）编号为 1~10。其中有一些活动对应相同的起点（即开始事件）或终点（结束事件），例如，活动 a_5 和 a_6 对应相同的开始事件 v_3（顶点③），活动 a_9 和 a_{10} 对应相同的结束事件 v_8（顶点⑧）。事件 v_1（顶点①）表示整个工程的开始，即为源点，事件 v_{10}（顶点⑩）表示整个工程的结束，即为汇点。其它顶点均表示其前面的所有活动都完成之后，其后续的各个活动可以开始，比如顶点⑤（事件 v_5）表示其前面的活动 a_4 和 a_5 已经完成，其后续的活动 a_8 和 a_9 可以开始。

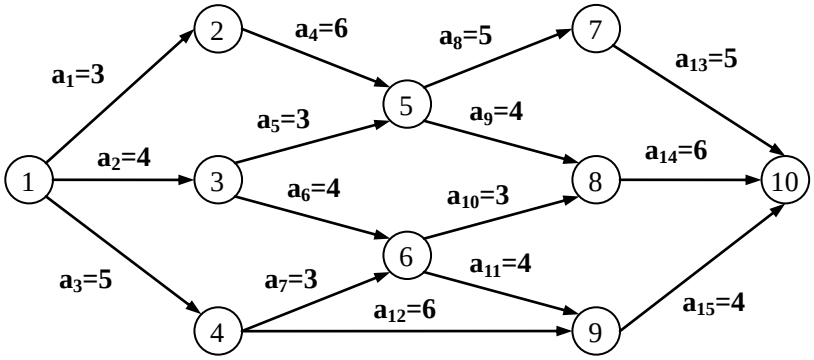


图 6-38 AOE 网示例

由图 6-38 可知，在 AOE 网中，有些活动之间只能依次（串行）地进行。例如，只有在 a_4 和 a_5 完成之后才能是顶点⑤对应事件发生，从而可以开始执行活动 a_8 和 a_9，还有些活动之间可以并行，如活动 a_1、a_2 和 a_3 之间就可以并行。因此，整个工程所需要的最少时间不是所有活动（子工程）所需时间的累积，而应是从源点到汇点之间的工期最长的一条路径。称这条最长的路径为关键路径（Critical Path）。由于从源点到汇点的有向路径可能不止一条，因此，需要采用有效的方法来计算。下面讨论这一问题。

2. 关键路径求解方法及描述

（1）工程所需最少时间的计算

关键路径的求解与工程所需最少时间是相关的，讨论如下。

工程所需最少时间的计算：若将源点对应事件的发生时间定为 0，则工程所需最少时间等于汇点对应事件的最早发生时间的值。显然，要使该事件发生，需要在以其为终点的所有活动都完成之后，而这些活动的实施需要在其所对应的起点事件发生之后。因此，汇点事件的最早时间依赖于其前驱顶点的最早发生时间。依此类推，每个顶点事件的最早发生时间都依赖于其前驱顶点事件的最早发生时间。由此可知，问题变成了按拓扑次序求解各顶点事件最早发生时间的问题。为此，需要解决一下问题。

① 为每个顶点设置一个对应事件的最早发生时间，合在一起构成一个数组 E[n+1]。

② 计算顶点 v 的 E[v]值：如前所述，每个顶点事件要在其前面所有活动都完成之后才能发生，因此，各 E[v]的值应是其每个前驱顶点 vPre 的最早发生时间与 vPre 和 v 两点之间活动的持续时间的和的最大值，即：

$$E[v]=\max\{\ E[vPre]+dur[vPre,v]\ \}$$

其中，vPre 代表 v 的各个前驱顶点，dur[vPre,v]代表弧<vPre,v>所对应活动的持续时

间。

③ 所有顶点的最早发生时间的计算方法：由于需要按拓扑次序求解各个顶点事件的最早发生时间，因而可将各顶点的 E[v]的计算嵌入到拓扑排序算法中。由于拓扑排序算法中对每个当前顶点都是往后搜索各个邻接点的，因此，对 E[v]的计算也采用这一次序，即在拓扑排序算法中，对当前顶点 v，每当找到一个后继顶点 w，就要计算 E[w]的值。因此，各顶点 w 的 E[w]值的计算要在其每个前驱顶点都作为当前顶点对其计算之后才能确定（此时，w 的入度变为 0），在此之前只能是暂时的。E[w]的计算公式如下：

```
E[w]=max{ E[w], E[v]+dur[v,w] }
```

其中，w 是 v 的后继顶点。

显然，各顶点 v 的 E[v]的初始值应设置为 0 比较合适。

将实现这一计算的下面语句嵌入到拓扑排序算法中的处理每个后继顶点的内层循环中，即可实现个顶点的 E[v]的计算（因而不再给出完整算法）：

```
if(E[v]+dur[v,w]>E[w])
        E[w]=E[v]+dur[v,w];
```

（2）关键路径的求解

下面讨论关键路径的求解。如前所述，关键路径是从源点到汇点的最长的路径（可能有多条），关键路径上的任一活动的耽搁都会直接影响到整个工程的进度。因此，关键路径事实上就是由所有不能耽搁的活动所组成的的路径。因此，求关键路径可通过求出所有不能耽搁的活动来实现，也就是要求出所有最早开始时间和最迟开始时间相同的活动，因而涉及相关各顶点事件的最早发生时间和最迟发生时间的计算。为此，需要实现一下问题的求解。

① 为每个顶点设置一个对应事件的最迟发生时间，合在一起构成一个数组 L[n+1]。

② 顶点 v 的 L[v]值的计算：如前所述，汇点对应事件的最早发生时间就是整个工程所需的最少时间，为不耽误工期，需要限定汇点事件的最迟发生时间与最迟发生时间相同。为此，汇点的各前驱顶点的最迟发生时间不能影响到汇点的最迟发生时间。由此可知，各 L[v]的值应是以不影响其每个后继顶点事件的发生为条件，从而可得计算公式如下：

```
L[v]=min{ L[vSuc]-dur[v,vSuc]}
```

其中，vSuc 代表 v 的后继顶点，dur[v,vSuc]代表弧<v,vSuc>所对应活动的持续时间的。

③ 所有顶点对应事件的最迟发生时间的计算方法：由于是按逆拓扑次序求解各顶点事件最迟发生时间的，因此可将各顶点的 L[v]的计算嵌入到逆驼鹿排序算法（采用逆邻接链表表示图，并按出度来求解即可仍采用拓扑排序算法）中。类似地，各顶点 v 的 L[v]的计算要在其每个后继顶点都作为当前顶点对其计算后才能最终确定（此时，v 的出度为 0），在此之前只能是暂时的。各顶点的最迟发生时间的计算变成对当前顶点 v 求其前驱 w 的 L[w]的计算，即：

```
L[w]=min{ L[w],L[v]-dur[w,v] }
```

其中，w 代表 v 的各个前驱顶点。

显然，各顶点 v 的 L[v]的初值应设置为不小于汇点事件的最迟发生时间的值。

将实施这一计算的下面的语句嵌入到逆拓扑排序算法中的处理每个后继顶点(此处事实上对应前驱顶点）的内存循环中即可实现各顶点的 L[v]的计算（同样不再给出完整算法），即：

```
if( L[v]-dur[w,v] < L[W] )
```

```
L[w]=L[v]-dur[w,v];
```

【算法分析】

关键路径的求解算法的时间复杂度取决于所采用的拓扑排序算法的时间复杂度。

3. 求解实例

【例 6.8】求解图 6-38 所示 AOE 网的关键路径。

【解】按照求解方法，首先需要分别求解最早发生时间和最迟发生时间，然后再确定关键路径。下面分别讨论。

① 最早发生时间的求解

为清晰起见，将求解过程以表 6-3 所示的表格形式给出，其中每一行代表一个求解步骤，每一列代表一个顶点的求解状态，每一单元格中的数据为此顶点当前最早发生时间和入度，形为：**最早发生时间/入度**，不变化时不给出。当入度变为 0 后，因不会再变化，故其下面均用阴影来指示。表中的“输出顶点”是按照拓扑排序次序进行的。

表 6-3 图 6-38AOE 网最早发生时间求解过程

步骤＼顶点	1	2	3	4	5	6	7	8	9	10
初始状态	0/0	0/1	0/1	0/1	0/2	0/2	0/1	0/2	0/2	0/3
输出 1		3/0	4/0	5/0						
输出 4						8/1			11/1	
输出 3					7/1	8/0				
输出 6								11/1	12/0	
输出 9										16/2
输出 2					9/0					
输出 5							14/0	13/0		
输出 7										19/1
输出 8										19/0
输出 10										

由表 6-3 可知，整个工程最少的工期为 19 个时间单位。

② 最迟发生时间的求解

求解过程类似最早发生时间的求解，可用表 6-4 形式给出，单元格中数据为：**最迟发生时间/出度**。

表 6-4 图 6-38AOE 网最迟发生时间求解过程

步骤＼顶点	1	2	3	4	5	6	7	8	9	10
初始状态	19/3	19/1	19/2	19/2	19/2	19/2	19/1	19/2	19/2	19/0
输出 10							14/0	13/0	15/0	
输出 8					9/1	10/1				
输出 7					9/0					
输出 5		3/0	6/1							

输出 3	0/2									
输出 9				9/1		10/0				
输出 6			6/0	7/0						
输出 3	0/1									
输出 4	0/0									
输出 1										

③ 关键路径的求解

为求解关键路径，将各顶点的最早发生时间和最迟发生时间在 AOE 网上各顶点上方和下方标出，如图 6-39 所示。将其中最早和最迟发生时间相同的活动连接起来便得到关键路径，如图中粗线所示。由图 6-39 可知，AOE 网中有 2 条关键路径：（1,2,5,7,10）和（1,2,5,8,10），关键活动包括：a_1、a_4、a_8、a_9、a_{13} 和 a_{14}。

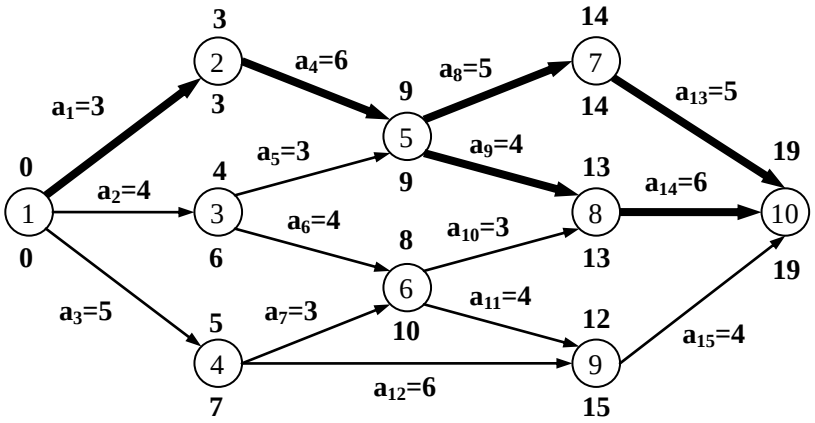


图 5-39 例 6.8 AOE 网关键路径求解示意图

本章小结

图是从实际问题中抽象出的一种模型，是一种复杂的结构，其中每个元素（在此称为顶点）可能有多个前驱和多个后继。图结构有许多相关的概念，包括：有向图、无向图、网络（带权图）、有向完全图、无向完全图、子图、邻接点、度、入度、出度、路径、回路、简单路径、简单回路、连通图、连通分量、强连通图、树、有向树、生成树等。

图结构最常用的存储形式是邻接矩阵和邻接表。邻接矩阵简单、直观，便于编程实现，不足之处是在图中边（弧）较少时，较浪费空间。邻接表存储能节省存储空间，但编程实现的难度要大一些。

深度遍历和广度遍历算法是图的基本运算，也是最重要的运算。深度遍历算法在选择下一个访问顶点时是根据深度越大越优先的原则进行的，其算法以递归形式给出，简捷直观，但有一定难度。广度遍历算法是典型的层次遍历算法，需要用队列保存有关信息。实例说明了这两个算法的基本应用。

最小生成树是图的应用之一，有 Prim 算法和 Kruskal 算法两种求解方法。Prim 算法的求解思想使得在求解过程中所选择的所有边是相连接的，其时间复杂度不受图的存储结构的影响，为 $O(n^2)$。Kruskal 算法的求解思想使得在求解过程中所选择的所有边可能不相连。

拓扑排序是面向工程问题的求解，主要通过判断是否存在有向回路来判断工程是否能顺利进行。算法的时间复杂度取决于图的存储结构。

最短路径算法是图结构的另外一个重要应用。