第 5 章

图

本章重点

- 图的概念及性质。
- 图的遍历和应用。

图也是一种非线性结构。从历年试题来看，必须熟练掌握图的基本概念、性质和存储结构，本章介绍的知识点对很多考生而言都是难点，如深度优先遍历、广度优先遍历、最小生成树算法、最短路径算法和关键路径算法等，一定要掌握这些算法的基本思想和实现步骤，并能实现。

思维导图

本章内容的思维导图如图 5-0 所示。

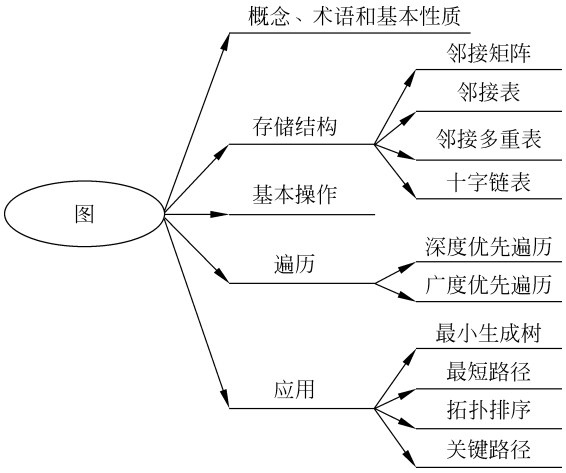


图 5-0　第 5 章内容思维导图

考纲内容

（一）图的基本概念

（二）图的存储及基本操作

1. 邻接矩阵

2. 邻接表

3. 邻接多重表、十字链表

（三）图的遍历

1. 深度优先搜索

2. 广度优先搜索

（四）图的基本应用

1. 最小(代价)生成树

2. 最短路径

3. 拓扑排序

4. 关键路径

5.1 图的基本概念

1. 图的定义

图的形式化定义为 $G=(V,\{VR\})$，其中，V 是图中数据元素的有穷非空集合，VR 是两个顶点间关系的集合，它可以是空集。通常将图中的数据元素称为**顶点**。

注意：V 是有穷非空集，VR 可以是空集，即图的顶点集不可为空，边集可为空，这意味着图不可以像线性表可以为空表，也不可以和树一样为空树。

2. 图的相关术语

1）无向图

给定图 $G=(V,\{VR\})$，若该图中每条边都是没有方向的，则称其为**无向图**。对于图 G 中顶点 v 和顶点 w 的关系可用无序对 (v,w) 表示，它是连接 v 和 w 的一条**边**。如图 5-1 所示的无向图，一共存在五条边 (V_1,V_2)，(V_1,V_3)，(V_2,V_4)，(V_1,V_5)，(V_5,V_4)。

2）有向图

给定图 $G=(V,\{VR\})$，若该图中每条边都是有方向的，则称其为**有向图**。对于图 G 中顶点 v 和顶点 w 的关系可用有序对 $<v,w>$ 表示，它是从 v 到 w 的一条**弧**，其中，v 称为**弧尾**或初始点，w 称为**弧头**或终端点。如图 5-2 所示的有向图，一共存在四条弧 $<V_1,V_2>$，$<V_1,V_3>$，$<V_3,V_4>$，$<V_4,V_1>$。

对于图 $G=(V,\{VR\})$，若无特别说明，在本章中约定：

(1) 若 $v_i,v_j\in V$，且 $<v_i,v_j>\in VR$ 或 $(v_i,v_j)\in VR$，则必有 $v_i\neq v_j$。

(2) 集合 VR 中的元素是彼此不同的，即若图 G 为有向图且 $<v_i,v_j>\in VR$，则不存在 $<v,w>\in VR$，使得 $v=v_i,w=v_j$；若图 G 为无向图且 $(v_i,v_j)\in VR$，则不存在 $(v,w)\in VR$，使得 $v=v_i,w=v_j$。

3）完全图

对于任一无向图，若其顶点的总数目为 n，边的总数目为 $e=\dfrac{n(n-1)}{2}$，则称其为**完全图**。如图 5-3 所示为一个完全图。

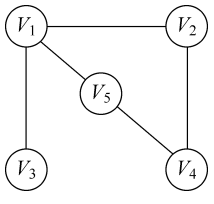


图 5-1 无向图

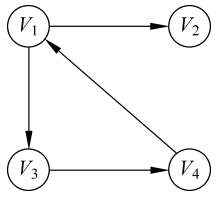


图 5-2 有向图

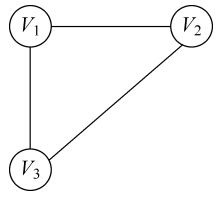


图 5-3 完全图

4）有向完全图

对于任一有向图，若其顶点的总数目为 n，边的总数目为 $e=n(n-1)$，则称其为**有向完全图**。如图 5-4 所示为一个有向完全图。

5）稀疏图和稠密图

对于具有 n 个顶点、e 条边或弧的图来说，若 e 很小（如 $e<n\log n$），则称其为**稀疏图**，反之称其为**稠密图**。

6）权和网

权：图中边或弧被赋予的某一数值称为权，它可以表示从一个顶点到另外一个顶点的距离或其他相关信息。

网：带权的图称为网。

7）稀疏网和稠密网

稀疏网：带权的稀疏图称为稀疏网。

稠密网：带权的稠密图称为稠密网。

8）子图

对于图 $G=(V,\{R\})$和图 $G'=(V',\{R'\})$，若 $V'\subseteq V$ 且 $R'\subseteq R$，则称 G'为 G 的**子图**。在图 5-5 中，如图 5-5(b)～图 5-5(d)所示为图 5-5(a)中无向图的子图。

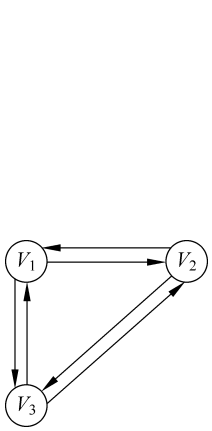


图 5-4　有向完全图

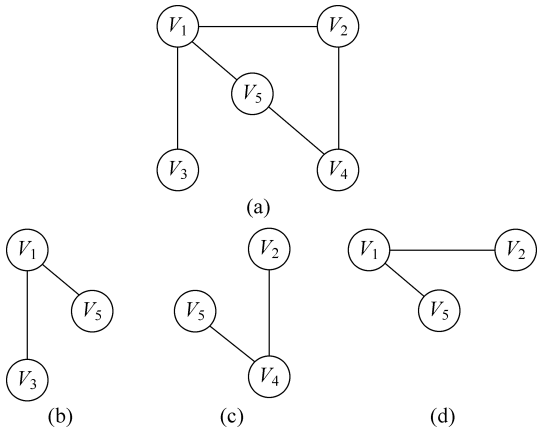


图 5-5　图与子图

9）邻接点

对于无向图 $G=(V,\{R\})$，若 $v\in V$，$w\in V$ 且$(v,w)\in R$，则称顶点 v 和顶点 w 互为**邻接点**，并称边(v,w)依附于顶点 v 和顶点 w，或称边(v,w)与顶点 v 和顶点 w 相关联。在图 5-6(a)中，顶点 V_1 和顶点 V_2 互为邻接点，边(V_1,V_2)与顶点 V_1 和顶点 V_2 相关联；由于顶点 V_1 和顶点 V_3 之间不存在边(V_1,V_3)，因此顶点 V_1 不是顶点 V_3 的邻接点，顶点 V_3 也不是顶点 V_1 的邻接点（即顶点 V_1 和顶点 V_3 不互为邻接点）。

对于有向图 $G=(V,\{R\})$，若 $v\in V$，$w\in V$ 且$<v,w>\in R$，则称顶点 v 邻接到顶点 w，顶点 w 邻接自顶点 v，弧$<v,w>$与顶点 v 和顶点 w

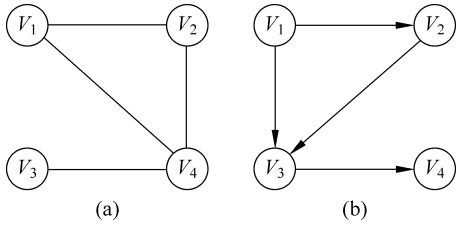


图 5-6　无向图和有向图示例 1

相关联。在图 5-6(b)中，顶点 V_1 邻接到顶点 V_2，顶点 V_2 邻接自顶点 V_1，弧$<V_1,V_2>$与顶点 V_1 和顶点 V_2 相关联；由于不存在弧$<V_2,V_1>$，因此顶点 V_2 不会邻接到顶点 V_1，顶点 V_1 也不会邻接自顶点 V_2。

10) 顶点的入度、出度和度

无向图中顶点的度：在无向图中，顶点 v 的度等于与该顶点相关联的边的数目，记为 TD(v)。如图 5-7(a)所示，顶点 V_2 的度 TD(V_2)=3。

有向图中顶点的入度、出度和度：在有向图中，某一顶点 v 的度等于该顶点的入度与出度之和，将其记为 TD(v)。其中，顶点 v 的入度(记为 ID(v))是以该顶点为弧头的弧的数目，顶点 v 的出度(记为 OD(v))是以该顶点为弧尾的弧的数目。如图 5-7(b)所示，由于顶点 V_4 的入度为 1，即 ID(V_4)=1，顶点 V_4 的出度为 2，即 OD(V_4)=2，因此顶点 V_4 的度 TD(V_4)=OD(V_4)+ID(V_4)=2+1=3。

11) 路径、简单路径和路径长度

路径：在图 $G=(V,\{R\})$中，顶点 v_1 到顶点 v_m 的路径是一个顶点序列($v_1,v_2,\cdots,v_i,v_j,\cdots,v_m$)，对于上述序列中任意两个相邻的顶点 v_i 和 v_j，若图 G 是无向图，则有(v_i,v_j)$\in R$；若图 G 是有向图，则有$<v_i,v_j>\in R$。

在如图 5-8(a)所示的无向图中，顶点 V_1 到顶点 V_4 的路径之一为(V_1,V_2,V_4)；在如图 5-8(b)所示的有向图中，顶点 V_1 到顶点 V_4 的路径之一为(V_1,V_2,V_4)。

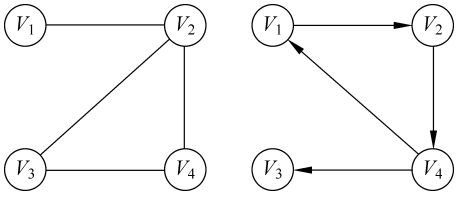


图 5-7 无向图和有向图示例 2

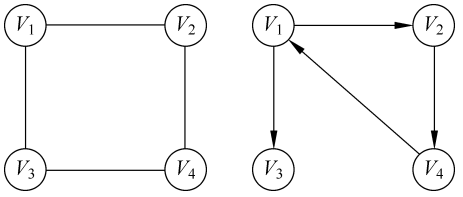


图 5-8 无向图和有向图示例 3

简单路径：给定一条路径，若该路径对应的序列中的顶点不重复出现，则称该路径为简单路径。在图 5-8(a)中，(V_1,V_2,V_4)为一条简单路径；而在图 5-8(b)中，(V_4,V_1,V_3)为一条简单路径。

路径长度：路径上边或弧的数目称为路径长度。在如图 5-8(a)所示的无向图中，顶点 V_1 到顶点 V_4 的路径为(V_1,V_2,V_4)，其长度是 2；而在如图 5-8(b)所示的有向图中，顶点 V_2 到顶点 V_3 的路径为(V_2,V_4,V_1,V_3)，其长度是 3。

12) 回路(环)和简单回路(简单环)

回路(环)：若某一路径中的第一个顶点和最后一个顶点相同，则称该路径为回路(或环)。在如图 5-9(a)所示的有向图中，路径(V_1,V_2,V_4,V_3,V_1)是一个回路；而在如图 5-9(b)所示的无向图中，路径(V_1,V_2,V_4,V_3,V_1)是一个回路。

简单回路(简单环)：在某一回路中，若除第一个顶点和最后一个顶点外，其余顶点均不重复，则称该回路为简单回路(或简单环)。

13) 连通图和连通分量

连通图：在无向图中，若从顶点 v 到顶点 v'有路径，则称 v 和 v'是连通的，若在该图中任意两个顶点间都是连通的，则称其为连通图。如图 5-10 所示的无向图是一个连通图。

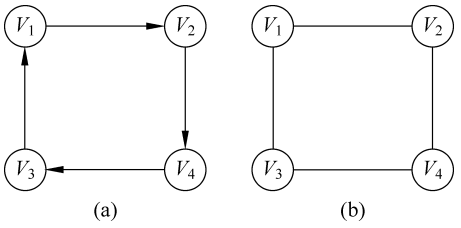


图 5-9 包含回路的有向图和无向图

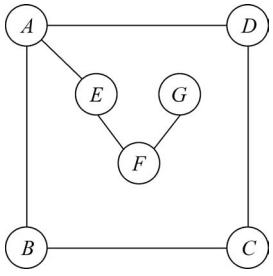


图 5-10 连通图

连通分量：连通分量即为无向图中的极大连通子图。如图 5-11(a)所示的无向图，其中包含 3 个连通分量，如图 5-11(b)所示。

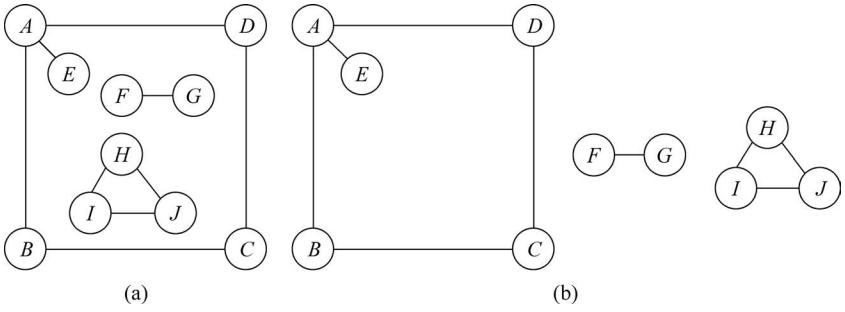


图 5-11 无向图及其连通分量

14) 强连通图和强连通分量

强连通图：在有向图中，若对于任意两顶点 v 和 v'，都存在从 v 到 v' 的路径和从 v' 到 v 的路径，则称这样的有向图为强连通图。如图 5-12 所示的有向图是一个强连通图。

强连通分量：强连通分量即为有向图中的极大强连通子图。如图 5-13(a)所示的有向图，其中包含两个强连通分量，如图 5-13(b)所示。

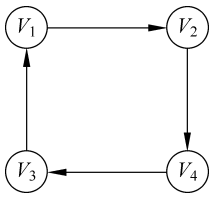


图 5-12 强连通图

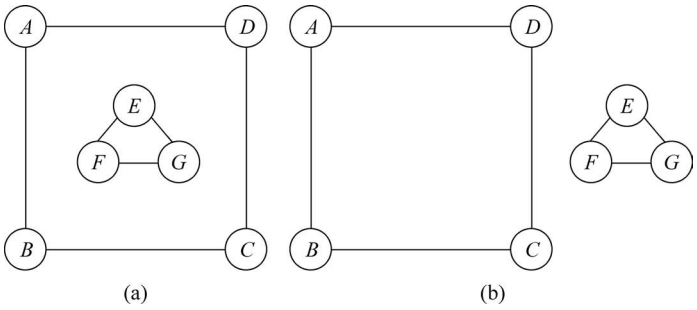


图 5-13 有向图及其强连通分量

15) 生成树和最小生成树

生成树：某一具有 n 个顶点的连通图的生成树是该图的极小连通子图，生成树包含这一连通图中的 n 个顶点和 $n-1$ 条边。如图 5-14(a)所示的连通图，它的一棵生成树如图 5-14(b)所示。

由生成树的定义可知，若某图有 n 个顶点和 $m(m<n-1)$ 条边，则该图是非连通的。

最小生成树：通常把各边带权的连通图称为连通网，在某一连通网的所有生成树中，对

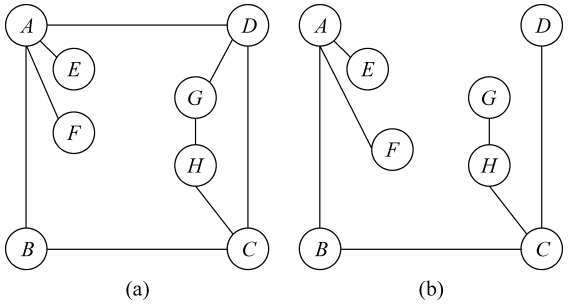


图 5-14　连通图及其生成树

其中每一棵生成树的各边权值求和，并找出权值之和最小的生成树，这一生成树被称为该连通网的最小生成树。如图 5-15(a)所示的连通网，其最小生成树如图 5-15(b)所示。

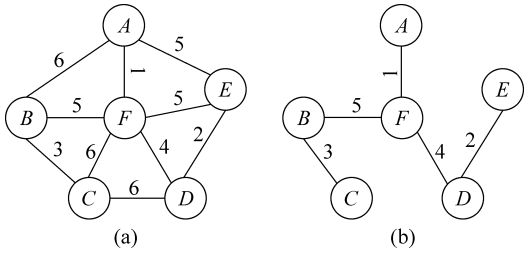


图 5-15　连通网及其最小生成树

非连通图的各连通分量的生成树组成的森林称为生成森林。

3. 图的性质

性质 1：若某图有度分别为 $TD(v_1)$，$TD(v_2)$，…，$TD(v_n)$的 n 个顶点 v_1，v_2，…，v_n 和 e 条边或弧，则有

$$e=\frac{1}{2}\sum_{i=1}^{n}TD(v_i)$$

性质 2：一棵有 n 个顶点的生成树有且仅有 $n-1$ 条边。

5.2　图的存储及基本操作

在存储图这一数据结构时，除了要考虑顶点本身如何存储，还要考虑如何存储图中顶点间的关系(即边或弧)。接下来简要介绍 4 种最为常见图的存储结构，分别为邻接矩阵法、邻接表法、十字链表法和邻接多重表法。

5.2.1　邻接矩阵法

1. 邻接矩阵的定义

在数组表示法中，需要使用一个数组存储图中顶点的信息，再使用另一个数组存储图中边或弧的信息，通常把后一个数组称为图的**邻接矩阵**。

在使用数组存储含有 $n(n>0)$个顶点的图 $G=\{V,\{E\}\}$时，将图中所有顶点存储在长

度为 n 的一维数组 Vertexs 中，并将图中边或弧的信息存储在 $n \times n$ 的二维数组（即邻接矩阵 Arcs）中。假设图 G 中顶点 v 和顶点 w 在数组 Vertexs 中的下标分别为 i 和 j，该图对应的邻接矩阵 Arcs 的定义如下。

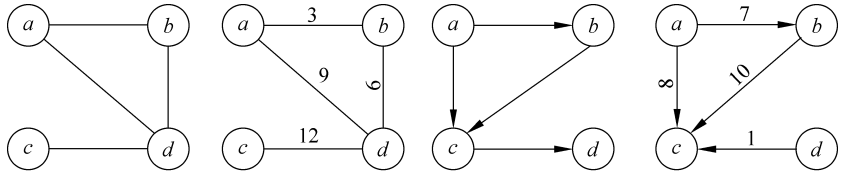
（1）若图 G 为有向图或无向图：

$$\text{Arcs}[i][j]=\begin{cases}1, & \text{若}(v,w)\text{或}<v,w>\in E\\ 0, & \text{其他}\end{cases}$$

（2）若图 G 为有向网或无向网：

$$\text{Arcs}[i][j]=\begin{cases}w_{ij}, & \text{若 } i\neq j \text{ 且}(v,w)\text{或}<v,w>\in E\text{，该边或弧的权值为 } w_{ij}\\ 0, & i=j\\ \infty, & \text{其他}\end{cases}$$

如图 5-16 所示，通常将其中无向图、无向网、有向图和有向网中所有顶点均存储在数组 Vertexs=[a，b，c，d]中，而这些图对应的邻接矩阵如图 5-17 所示。



(a) 无向图 (b) 无向网 (c) 有向图 (d) 有向网

图 5-16 无向图、无向网、有向图和有向网

$$\text{Arcs}=\begin{bmatrix}0&1&0&1\\1&0&0&1\\0&0&0&1\\1&1&1&0\end{bmatrix}\quad \text{Arcs}=\begin{bmatrix}0&3&0&9\\3&0&0&6\\0&0&0&12\\9&6&12&0\end{bmatrix}\quad \text{Arcs}=\begin{bmatrix}0&1&1&0\\0&0&1&0\\0&0&0&1\\0&0&0&0\end{bmatrix}\quad \text{Arcs}=\begin{bmatrix}0&7&8&0\\0&0&10&0\\0&0&0&1\\0&0&0&0\end{bmatrix}$$

(a) 无向图的邻接矩阵 (b) 无向网的邻接矩阵 (c) 有向图的邻接矩阵 (d) 有向网的邻接矩阵

图 5-17 邻接矩阵

2. 邻接矩阵的特点

邻接矩阵的特点总结如下。

（1）由于创建邻接矩阵时，输入顶点的顺序可能不同，因此一个图的邻接矩阵并不是唯一的。

（2）对于含有 n 个顶点的图，无论图中包含多少条边或弧，其邻接矩阵一定是 $n \times n$ 的二维数组，因此邻接矩阵更适用于存储稠密图。

（3）无向图的邻接矩阵具有对称性，因此可采用压缩存储的方式，只对其上三角（或下三角）元素进行存储。

（4）对于无向图，若某一顶点 v 在一维数组 Vertexs 中的下标为 i，则该顶点的度为邻接矩阵第 $i+1$ 行中值为 1 的元素的总数目。

（5）对于有向图，若某一顶点 v 在一维数组 Vertexs 中的下标为 i，则该顶点的出度为邻接矩阵第 $i+1$ 行中值为 1 的元素的总数目，该顶点的入度为邻接矩阵第 $i+1$ 列中值为 1 的元素的总数目。

（6）在简单应用中，可直接用二维数组作为图的邻接矩阵（可忽略顶点信息）。

(7) 若邻接矩阵中的元素仅表示顶点之间的边是否存在时，可记为 0 或 1。

(8) 用邻接矩阵存储图，很容易确定图中任意两个顶点之间是否有边相连，但是确定图中边数时需按行和列进行检索，时间代价较大。

构造一个具有 n 个顶点、e 条边的无向网的时间复杂度为 $O(n^2+e\cdot n)$，其中，对邻接矩阵的初始化使用了 $O(n^2)$ 的时间。

5.2.2 邻接表法

1. 邻接表的定义

在使用邻接表存储图时，通常将图分为顶点和边两部分：第一部分为图中每一顶点及与该顶点相关联的第一条边或弧；第二部分为与某一顶点相关联的所有边或以某一顶点为弧尾的所有弧。

在实现顶点部分时，使用 data 域来存储图中每一个顶点的值，并使用 FirstArc 域来存储与该顶点相关联的第一条边或弧，这一部分通常使用数组来存储。在实现边部分时，每一条边或弧都存储在一个结点中，该结点由 adjacent 域、info 域和 NextArc 域组成，这些结点形成了若干个单链表。一般情况下，第一部分中数组每一维的 FirstArc 域均指向第二部分中某一单链表的第一个结点，该结点和当前 FirstArc 域对应的 data 域存储的顶点之间存在边或弧。

如图 5-18(a)所示的无向网，其对应的邻接表如图 5-18(b)所示。如图 5-18(a)所示的无向网中与顶点 A 相关联的边为(A,C)和(A,D)，它们的权值分别为 1 和 2，而在图 5-18(b)中，由于值为 C 的元素的下标为 2，因此边(A,C)对应的结点 adjacent 域值为 2，又因为该边的权值为 1，所以这一结点的 info 域值为 1。

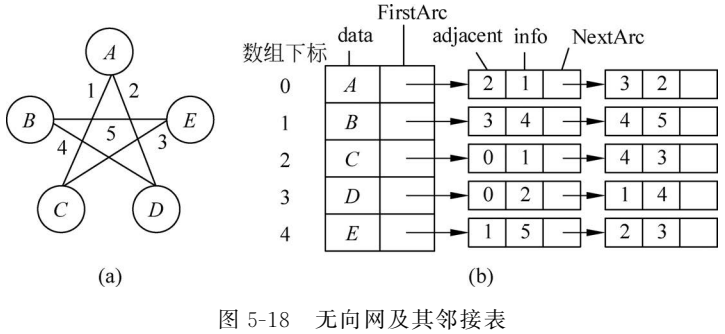


图 5-18 无向网及其邻接表

同理，由于值为 D 的元素的下标为 3，因此边(A,D)对应的结点 adjacent 域值为 3，又因为该边的权值为 2，所以这一结点的 info 域值为 2。

值为 A 的元素的 FirstArc 域指向由上述两个结点组成的单链表的第一个结点(图中所示为边(A,C)对应的结点)。

如图 5-19(a)所示的有向网，其对应的邻接表如图 5-19(b)所示。如图 5-19(a)所示的有向网中以顶点 B 为弧尾的弧为$<B,C>$和$<B,D>$，它们的权值分别为 5 和 3，而在图 5-19(b)中，由于值为 C 的元素的下标为 2，因此弧$<B,C>$对应的结点 adjacent 域值为 2，又因为该弧的权值为 5，所以这一结点的 info 域值为 5。

同理，由于值为 D 的元素的下标为 3，因此弧$<B,D>$对应的结点 adjacent 域值为 3，又

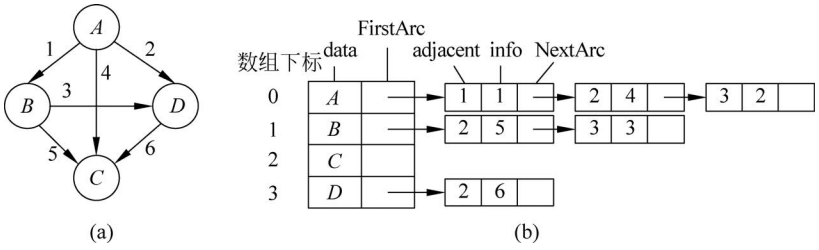


图 5-19 有向网及其邻接表

因为该弧的权值为 5，所以这一结点的 info 域值为 5。

值为 B 的元素的 FirstArc 域指向由上述两个结点组成的单链表的第一个结点(图中所示为弧$<B,C>$)。

2. 邻接表的实现

邻接表存储结构如图 5-20 所示。

```
#define MaxVertexSize 20

typedef struct ArcEdgeNode{
    int adjacentVex;                        //边或弧指向的顶点位置
    struct arcEdgeNode *nextArcEdge;        //指向下一条边或弧
    int info;                               //权重信息
}ArcEdgeNode;

typedef struct vnode
{
    int data;                               //顶点
    ArcEdgeNode *firstArcEdge;              //边或弧
}VertexNode, AdjList[MaxVertexSize];

typedef struct
{
    VertexNode vertices;                    //邻接表
    int n,e;                                //图的顶点数和边数或弧数
    int kind;                               //图的种类标志
}AdjacencyListGraph;                        //以邻接表方式存储的图
```

图 5-20 图的邻接表存储结构

3. 邻接表的特点

邻接表的特点如下。

(1) 将图中存储边或弧的结点通过不同的顺序链接起来会形成不同的单链表，也就是说，一个图的邻接表并不是唯一的，它取决于建立邻接表的算法及边的输入顺序。

(2) 若使用邻接表存储具有 e 条边的无向图，则需要 $2e$ 个结点存储该图中的边，而对于具有 e 条弧的有向图，则需要 e 个结点存储此图中的弧，这是因为无向图中的边在邻接表中出现了两次。

因此，对于 n 个顶点 e 条边的无向图，其存储空间为 $O(n+2e)$，而对于 n 个顶点 e 条弧的有向图，其存储储空间则为 $O(n+e)$。

(3) 对于具有 n 个顶点 e 条边或弧的稀疏图而言，若采用数组存储该图，则需要 n^2 个存储空间来存储图中所有的边或弧，而采用邻接表存储该图，则至多需要 $2e$ 个结点存储图

中所有的边或弧。由于稀疏图中的顶点数目远大于边数，即 $n \gg e$，因此可得 $n^2 \gg 2e$，所以对于稀疏图，采用邻接表存储更节省存储空间。

(4) 对于无向图，某一顶点的度为其对应链表中结点(边)的总数目。

(5) 对于有向图，若某一顶点在数组中的存储下标为 i，则该顶点的出度为其对应链表中结点(弧)的总数目，入度为邻接表中 adjacent 域内值为 i 的结点(弧)的总数目。

(6) 对于邻接表，给定某一顶点，查找其邻边只需遍历该顶点对应的边的邻接表，而若需判断两个顶点之间是否存在边时，则需要遍历这两个顶点对应的边的邻接表。

注意：在使用邻接表存储有向图时，计算图中某一顶点的出度很容易，但是在计算某一顶点的入度时，最坏情况下需要遍历整个邻接表。因此，有时为了方便计算有向图中某一顶点的入度，可以为该图建立一个**逆邻接表**。

如图 5-21(a)所示的有向图，其对应的逆邻接表如图 5-21(b)所示。如图 5-21(a)所示的有向图中以顶点 D 为弧头的弧有$<A,D>$和$<B,D>$；而在图 5-21(b)中，由于值为 A 的元素的下标为 0，因此弧$<A,D>$对应的结点 adjacent 域值为 0。同理，由于值为 B 的元素的下标为 1，因此弧$<B,D>$对应的结点 adjacent 域值为 1。

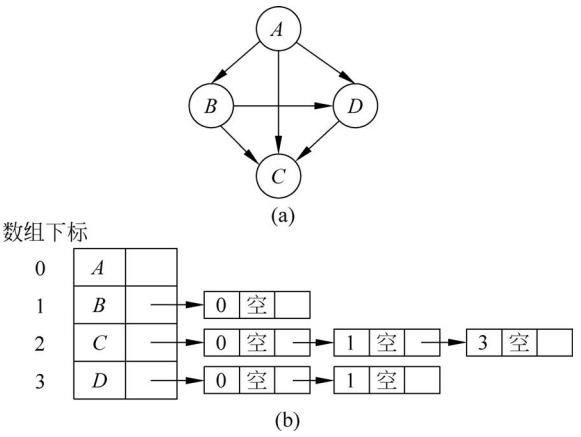


图 5-21 有向图及其逆邻接表

值为 D 的元素的 FirstArc 域指向由上述两个结点组成的单链表的第一个结点(图中所示为弧$<A,D>$)。

在建立邻接表或逆邻接表时，若输入的顶点信息为顶点的编号，则建立邻接表或逆邻接表的时间复杂度为 $O(n+e)$；否则，需要通过查找才能得到顶点在图中的位置，则时间复杂度为 $O(n \cdot e)$。

5.2.3 十字链表法

1. 十字链表的定义

十字链表通常用于存储有向图，可以将其看成邻接表和逆邻接表的结合。

在使用十字链表存储有向图时，可将它分为两部分：顶点结点部分和弧结点部分。在顶点结点部分，每一个顶点结点包含 data 域、FirstTailArc 域和 FirstHeadArc 域，如图 5-22 所示。其中，data 域存储顶点的值，FirstTailArc 指向以当前顶点为弧尾的第一条弧，

FirstHeadArc 指向以当前顶点为弧头的第一条弧。

data	FirstTailArc	FirstHeadArc

图 5-22 十字链表中的顶点结点

在弧结点部分，每一条弧结点包含 TailVertex 域、HeadVertex 域、NextTailArc 域、NextHeadArc 域和 info 域，如图 5-23 所示。其中，TailVertex 域存储当前弧的弧尾在数组中的下标，HeadVertex 域存储当前弧的弧头在数组中的下标，NextTailArc 指向与当前弧有相同弧尾的下一条弧，NextHeadArc 指向与当前弧有相同弧头的下一条弧，info 域存储当前弧的其他信息。

TailVertex	HeadVertex	NextTailArc	NextHeadArc	info

图 5-23 十字链表中的弧结点

如图 5-24(a)所示的有向图，其对应的十字链表如图 5-24(b)所示。

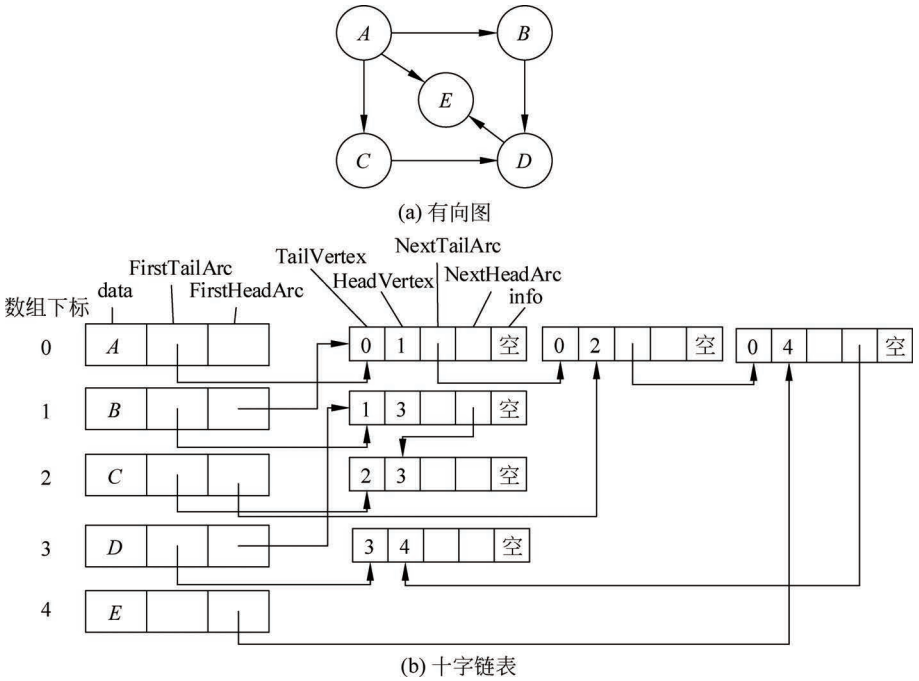


图 5-24 有向图及其十字链表

2. 十字链表的实现

十字链表的存储结构如图 5-25 所示。

3. 十字链表的特点

十字链表的特点如下。

(1) 十字链表所有的顶点结点通常存储在数组(即顺序结构)中，而所有的弧结点存储在单链表(即链式结构)中。

(2) 在这一存储结构中可以很容易地计算图中某一顶点的出度和入度。

```
#define MaxVertexSize 20

typedef struct ArcEdgeNode{
    int headVex,tailVex;                    //弧的尾和头顶点的位置
    struct ArcBox * headLink, * tailLink;   //弧头相同和弧尾相同的链域
    int * info;                             //弧相关信息的指针
}ArcBox;

typedef struct VexNode
{
    int data;                               //顶点
    ArcBox * firstIn, * firstOut;           //边或弧
}VertexNode;

typedef struct
{
     VertexNode hList[MaxVexterSize];       //表头向量
     int n,e;                               //图的顶点数和边数或弧数
}OrthogonalListGraph;                       //以十字链表方式存储的图
```

图 5-25　十字链表的存储结构

(3) 某一个图的十字链表结构不唯一，即存在多个十字链表结构均对应于同一个图的情况。

5.2.4　邻接多重表法

1. 邻接多重表的定义

在使用邻接表存储无向图时，无向图中的每一条边都对应两个结点，由于这两个结点均属于两个不同的单链表，这使得无向图中的某些操作(例如，在删除图中某一指定的边时，需要对邻接表中的两条单链表执行删除操作)变得复杂，此时可以采用邻接多重表来存储无向图以解决上述问题。

在使用邻接多重表存储无向图时，与使用十字链表存储有向图类似，也可将其分为两部分：顶点结点部分和边结点部分。在顶点结点部分，每一个顶点结点包含 data 域和 FirstEdge 域，如图 5-26 所示。其中，data 域存储顶点的值，FirstEdge 则指向与当前顶点相关联的第一条边。

data	FirstEdge

图 5-26　邻接多重表的顶点结点

在边结点部分，每一个边结点包含 mark 域、VertexOne 域、NextEdgeOne 域、VertexTwo 域、NextEdgeTwo 域和 info 域，如图 5-27 所示。其中，mark 域用于标记当前边是否被访问，VertexOne 域和 VertexTwo 域分别存储当前边的两个顶点在数组中的下标，NextEdgeOne 指向与 VertexOne 对应的顶点相关联的下一条边，NextEdgeTwo 指向与 VertexTwo 对应的顶点相关联的下一条边，info 域存储当前边的其他信息。

mark	VertexOne	NextEdgeOne	VertexTwo	NextEdgeTwo	info

图 5-27　邻接多重表的边结点

图 5-28(a)中的无向图的邻接多重表如图 5-28(b)所示。

2. 邻接多重表的实现

邻接多重表的存储结构如图 5-29 所示。

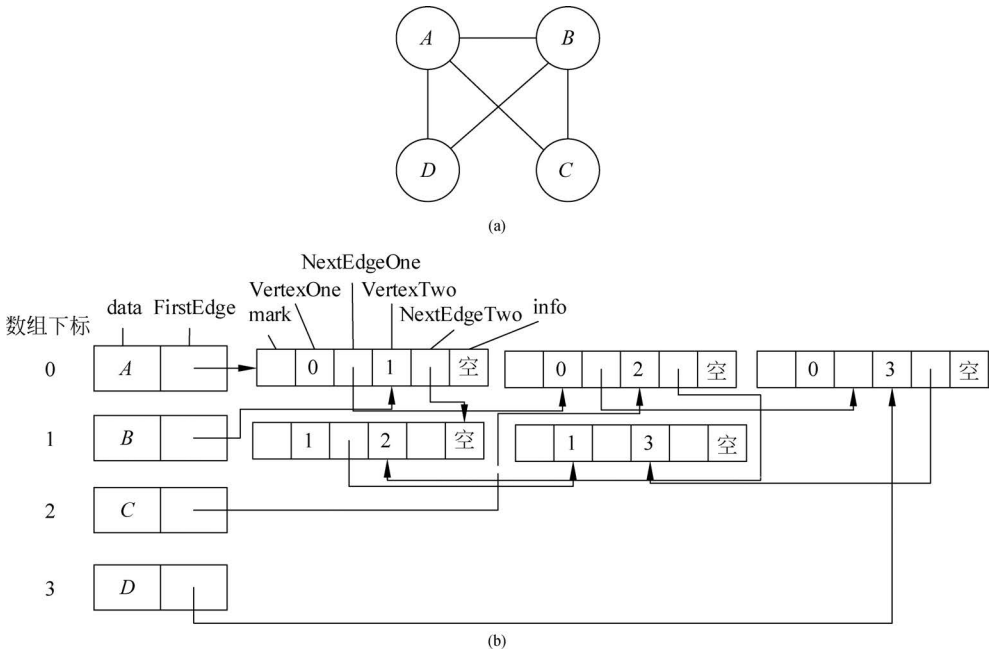


图 5-28 无向图及其邻接多重表

```
#define MaxVertexSize 20
typedef emnu{unvisited, visited} IsVisited;
typedef struct EdgeNode{
    IsVisited  IsVisitedMark;                    //访问标记
    int hVex,tVex;                               //该边依附的两个顶点的位置
    struct EdgeBox * hLink, * tLink;             //分别指向依附这两个顶点的下一条边
    int * info;                                  //相关信息的指针
} EdgeBox;

typedef struct VextexNode
{
    int data;                                    //顶点
    EdgeBox * firstEdge;                         //指向第一条依附该顶点的边
}VertexNode;

typedef struct
{
    VertexNode adjMulList[MaxVexterSize];
    int n,e;                                     //无向图的顶点数和边数或弧数
}AdjacencyMultiListGraph;                        //以邻接多重表方式存储的无向图
```

图 5-29 邻接多重表的存储结构

3. 邻接多重表的特点

邻接多重表的特点如下。

(1) 邻接多重表通常将所有的顶点结点存储在数组(即顺序结构)中,而所有的边结点存储在单链表(即链式结构)中。

(2) 在这一存储结构中,同一边只有一个结点。

(3) 某一个图的邻接多重表并不唯一,即存在多个邻接多重表均对应于同一个图的情况。

5.2.5 图的基本操作

表 5-1 为图的基本操作，在实现这些基本操作时，应该考虑基于上述介绍的 4 种基本存储结构时算法效率的问题。

表 5-1 图的基本操作

序号	基本操作的名称	基本操作的功能说明
1	CreateGraph(&Graph,Vertex,VR)	创建并初始化图 Graph
2	DestroyGraph(&Graph)	销毁图 Graph
3	LocateVertex(Graph,v)	判断 v 是否为图 Graph 中的顶点
4	GetVertex(Graph,v)	返回顶点 v 的值
5	SetVertex(&Graph,v,value)	令 value 为顶点 v 的值
6	GetFirstAdjacentVertex(Graph,v)	获取顶点 v 在图 Graph 中的第一个邻接点
7	GetNextAdjacentVertex(Graph,v,w)	返回顶点 v 的一个邻接点
8	InsertVertex(&Graph,v)	将 v 作为顶点添加到图 Graph 中
9	DeleteVertex(&Graph,v)	删除图 Graph 中的顶点 v 及其相关的弧或边
10	InsertArc(Graph,v,w)	若图 Graph 为有向图，则添加弧$<v,w>$；否则添加边(v,w)
11	DeleteArc(Graph,v,w)	若图 Graph 为有向图，则删除弧$<v,w>$；否则删除边(v,w)
12	DFSTraverse(Graph)	深度优先遍历图 Graph
13	BFSTraverse(Graph)	广度优先遍历图 Graph
14	VisitVertex(Vertex)	访问顶点 Vertex

5.3 图的遍历

图的遍历是指从图中某一顶点开始按指定方式访问图中每一个顶点，在执行图的遍历操作时，要求所有顶点均被访问且每一个顶点仅能被访问一次。图的遍历方式主要有深度优先遍历(Depth-First Search，DFS)和广度优先遍历(Breadth-First Search，BFS)，这两种方式均可被用于遍历无向图和有向图。

图的遍历算法极为重要，它是判断图是否连通、进行拓扑排序和求解关键路径等图的应用的基础。由于图中任一顶点都可能存在相邻的顶点，在实现图的遍历算法时，为了避免同一顶点被多次访问，可以使用辅助变量(如数组)来标记某一顶点是否被访问过。

5.3.1 深度优先遍历

图的深度优先遍历的递归过程的基本思想如下。

(1) 从图中某一顶点 v 开始，先访问顶点 v，若被访问的图是无向图，则依次以顶点 v 未被访问的邻接点为起点深度优先遍历图，直到所有与顶点 v 连通的顶点都被访问；若被访问的图是有向图，则依次以顶点 v 邻接到的未被访问的顶点为起点深度优先遍历图，直到从顶点 v 出发能到达的所有顶点都被访问。

(2) 若图中还有未被访问的顶点，则从中选择一个顶点并重复执行(1)，直到图中所有

顶点均被访问。

该算法的实现代码如图 5-30 所示。

```
#define MaxVisitedSize 20
int IsVisited[MaxVisitedSize];

void DFS(Graph G, int v)
{
    IsVisited[v] = 1;                                   //标记访问顶点 v
    VisitVertex(v);                                     //访问顶点 v
    for(w = GetFirstAdjacentVertex(G,v);w >= 0;w = GetNextAdjacentVertex(Graph,v,w))
        if(IsVisited[w] == 0)
            DFS(G,w);                                   //对于 v 尚未访问的邻接点 w 递归调用 DFS

}

void DFSTraverse(Graph G)                               //对图 G 做深度优先遍历
{
    for(v = 0;v < G.n;v++)
        IsVisited[v] = 0;                               //初始化标志数组
    for(v = 0;v < G.n;v++)
        if(IsVisited[v] == 0)                           //对尚未访问的顶点调用 DFS
            DFS(G,v);
}
```

图 5-30 深度优先遍历递归算法

对于含有 n 个顶点和 e 条边或弧的图，调用深度优先遍历递归算法对其进行深度优先遍历时，由于图中每一个顶点仅能被访问一次，因此对每一个顶点至多调用一次 DFS()方法，所以递归调用的总次数为 n，所需的时间为 $O(n)$，该算法运算时需要借助于一个递归工作栈，故空间复杂度为 $O(n)$。

考虑到深度优先遍历图的过程实质上是对每一个顶点查找其邻接点的过程，故所需时间与图的存储结构相关。当使用邻接表存储该图时，查找每一个顶点的邻接点所需的时间为 $O(e)$；而使用邻接矩阵存储该图时，查找每一个顶点的邻接点所需的时间为 $O(n)$。

综上可知，使用邻接表存储图时，深度优先遍历算法的时间复杂度为 $O(n+e)$；而使用邻接矩阵存储该图时，深度优先遍历算法的时间复杂度为 $O(n^2)$。

对于如图 5-31 所示的无向图，按照深度优先遍历递归算法对其进行深度优先遍历时，假设从顶点 A 开始，则一种可能被访问的顺序为 $A \rightarrow F \rightarrow E \rightarrow D \rightarrow C \rightarrow B \rightarrow G$。

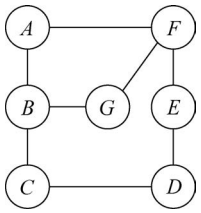


图 5-31 无向图

注意：

(1) 由于通常情况下图的邻接表并不是唯一的(无向图与边的输入次序有关，有向图与弧的输入次序有关)，因此深度优先遍历图时各顶点被访问的顺序可能不同，所得的遍历序列不唯一；若使用邻接矩阵存储图，则所得的遍历序列一定是唯一的，因为不论是无向图还是有向图，其邻接矩阵都是唯一的。

(2) 对于连通图执行深度优先遍历，会产生深度优先生成树，否则会产生深度优先生成森林。基于邻接矩阵的图，其深度优先生成树或森林是唯一的，而基于邻接表的图，其深度优先生成树或森林是不唯一的。

5.3.2 广度优先遍历

图的广度优先遍历的基本思想如下。

(1) 从图中某一顶点 v 开始，先访问顶点 v，若被访问的图是无向图，则依次访问顶点 v 未被访问的邻接点，再依次访问这些邻接点未被访问的邻接点，直到所有与顶点 v 连通的顶点都被访问；若被访问的图是有向图，则依次访问顶点 v 邻接到的所有未被访问的顶点，再依次访问这些顶点邻接到的未被访问的顶点，直到从顶点 v 出发能到达的所有顶点都被访问。

(2) 若图中还有未被访问的顶点，则从中选择一个顶点并重复执行(1)，直到图中所有顶点均被访问。

该算法的实现代码如图 5-32 所示。

```
#define MaxVisitedSize 20
int IsVisited[MaxVisitedSize];

void BFSTraverse(Graph G)                                   //对图 G 做广度优先遍历
{
    for(v = 0;v < G.n;v++)
        IsVisited[v] = 0;                                   //初始化标志数组
    InitQueue(Q);
    for(v = 0;v < G.n;v++)
    {
        if(IsVisited[v] == 0)                               //对尚未访问的顶点进行处理
        {
            IsVisited[v] = 1;                               //修改访问标志
            VisitVertex(v);                                 //访问顶点 v
            EnQueue(Q,v);
            while(IsQueueEmpty(Q) == 0)
            {
                DeQueue(Q,u);
                for(w = GetFirstAdjacentVertex(G,v);w >= 0;w = GetNextAdjacentVertex(Graph,v,w))
                {
                    if(IsVisited[w] == 0)                   //w 为 v 尚未访问的邻接点
                    {
                        IsVisited[w] = 1;
                        VisitVertex(w);
                        EnQueue(Q,w);
                    }//end if
                }//end for
            }//end while
        }
    }
}
```

图 5-32 广度优先遍历算法

如图 5-33 所示的无向图，按照上述算法对其进行广度优先遍历时，一种被访问的顺序为 $A \rightarrow H \rightarrow F \rightarrow B \rightarrow E \rightarrow C \rightarrow D \rightarrow G$。

注意：

(1) 由于通常情况下图的邻接表并不是唯一的(无向图与边的输入次序有关，有向图与弧的输入次序有关)，因此广度优先遍历图时各顶点被访问的顺序可能不同，所得的遍历序列不唯一；若使用邻接矩阵存储图，则所得的遍历序列一定是唯一的，因为不论是无向图，

还是有向图，其邻接矩阵都是唯一的。

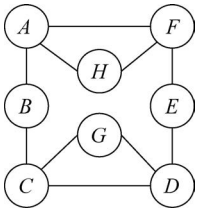


图 5-33 无向图

(2) 对于连通图执行广度优先遍历，会产生广度优先生成树，否则会产生广度优先生成森林。基于邻接矩阵的图，其广度优先生成树或森林是唯一的，而基于邻接表的图，其广度优先生成树或森林是不唯一的。

(3) 对于含有 n 个顶点和 e 条边或弧的图，当使用邻接表存储该图时，对其进行广度优先遍历和深度优先遍历所需的时间一样，即广度优先遍历算法的时间复杂度为 $O(n+e)$；而当使用邻接矩阵存储该图时，对其进行广度优先遍历和深度优先遍历所需的时间一样，即广度优先遍历算法的时间复杂度为 $O(n^2)$。

(4) 广度优先遍历与二叉树的层次遍历完全一致，可以认为前者是后者的扩展。

(5) 广度优先遍历算法可用于求解单源最短路径问题，因为它总是按距离由近到远来遍历每个顶点。

5.4 图的基本应用

5.4.1 最小生成树

在图的相关术语中已经介绍过，最小生成树是最小代价生成树的简称，它是指在 n 个顶点的连通网中构造一棵代价最小的生成树。由于各边的代价(权值)有可能相同，或不同边的代价之和可能相同，故最小生成树可能不唯一。构造最小生成树的算法大都利用以下性质：假设 $N=(V,\{E\})$是一个连通网，U 是顶点集 V 的一个非空子集，若(u,v)是一条最小权值的边，其中，$u\in U$，$v\in V-U$，则必存在一棵包含边(u,v)的最小生成树。接下来介绍最为常用的普里姆(Prim)算法和克鲁斯卡尔(Kruskal)算法。

1. Prim 算法

假设 $G=\{V,\{E\}\}$是含有 n 个顶点的连通网，使用 Prim 算法构造其最小生成树 $T=\{U,\{\text{TE}\}\}$的基本思想如下。

(1) 指定连通网 G 中某一顶点 w 作为构造最小生成树的起点，并令 $U=\{w\}$，TE=$\{\}$。

(2) 在所有 $u\in U$，$v\in V-U$ 的边中，找到具有最小权值的一条边$(u,v)\in E$，将 v 并入 U，并将边(u,v)并入 TE。

(3) 重复执行(2)，直到 $U=V$，此时最小生成树包含 $n-1$ 条边。

在实现构造最小生成树时，可采用邻接矩阵 Arcs 存储图。当 $i=j$ 时，Arcs[i][j] =0；当 $i\neq j$ 时，若下标为 i 和 j 的顶点之间存在边且该边权值为 w，则 Arcs[i][j]=Arcs[j][i]=w，否则 Arcs[i][j]=Arcs[j][i]=∞。此外，还需要一个辅助数组 CloseEdge，用于存储从 U 到 $V-U$ 中权值最小的边。对于 $V-U$ 中的任一顶点 v 都对应数组中的一个分量 CloseEdge[i]，它包含两部分，一部分用于存储与顶点 v 相关联的边 edge 的权值；另一部分用于存储边 edge 中属于 U 的顶点的下标。edge 是所有边(v,w)($w\in U$)中权值最小的边，若边 edge 为组成最小生成树的边，则将该部分的值置为 0，即将顶点 v 并入 U，并更新 $V-U$ 中顶点对应分量的值。最后还需要一个列表 arc 来存储最小生成树的边。

假定网中共包含 n 个顶点，则初始化 CloseEdge 的循环语句的频度为 n，而构造最小生

成树的循环语句的频度为 $n-1$。又因为构造最小生成树时需要获取权值最小的边和更新 CloseEdge 中边的长度，它们对应的执行语句的频度为 $n-1$ 和 n。因此，Prim 算法的时间复杂度为 $O(n^2)$。由于该算法的执行时间只与图中顶点的总数目有关，而与边的总数目无关，因此它更适用于稠密网求最小生成树。

以 A 为起点，由上述 Prim 算法构造最小生成树的过程如图 5-34 所示。

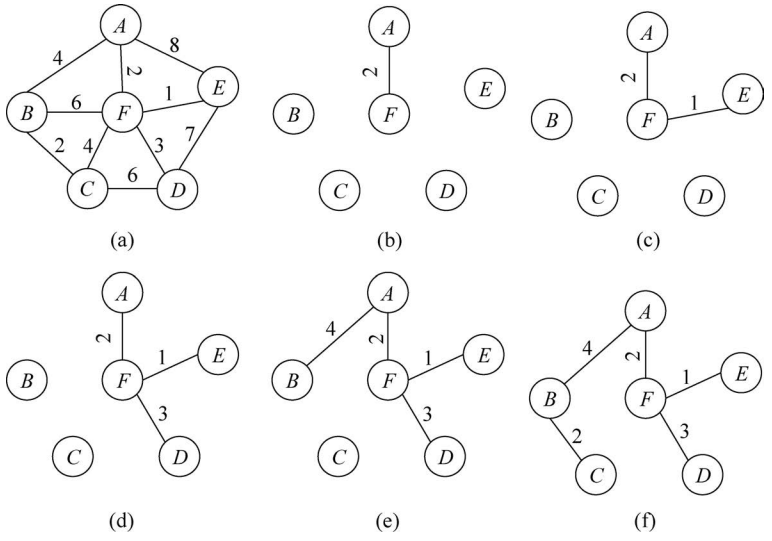


图 5-34 Prim 算法构造最小生成树

注意：在通常情况下，对于某一图而言，选择不同的起点构造最小生成树，其过程不同。

2. Kruskal 算法

假设 $G=\{V,\{E\}\}$ 是含有 n 个顶点的连通网，使用 Kruskal 算法构造其最小生成树 $T=\{U,\{\mathrm{TE}\}\}$ 的基本思想如下。

(1) 将连通网 G 中所有的边存入集合 Edges，并使它们按权值的升序排列，同时令 $U=V$，TE=｛｝，由于此时 TE 为空，最小生成树 T 中每一个顶点都自成一个连通分量。

(2) 依次访问 Edges 中的边，若当前被访问的边的两个顶点属于不同的连通分量，则将该边并入 TE，并标记两个顶点所在的连通分量为同一连通分量；否则将该边从 Edges 中删除。

(3) 重复执行(2)，直到最小生成树 T 的所有顶点均属于同一连通分量，此时 Edges 中的边与组成最小生成树 T 的边相同，这些边组成了集合 TE。

Kruskal 算法的时间复杂度为 $O(e^2)$，也就是说，该算法的执行时间与图中边的总数目有关，而与顶点的总数目无关，因此它更适用于稀疏网求最小生成树。事实上，可对上述 Kruskal 算法进行如下改进，即用堆来存储连通网中的边，并采用更加合适的数据类型来描述生成树。此时 Kruskal 算法的时间复杂度为 $O(e\log e)$。

以 A 为起点，由上述 Kruskal 算法构造最小生成树的过程如图 5-35 所示。

注意：对图中所有边按权值的升序排列时，由于采用的排序算法不同，权值相等的边的顺序有可能不同，所以构造图的最小生成树的过程可能不同。

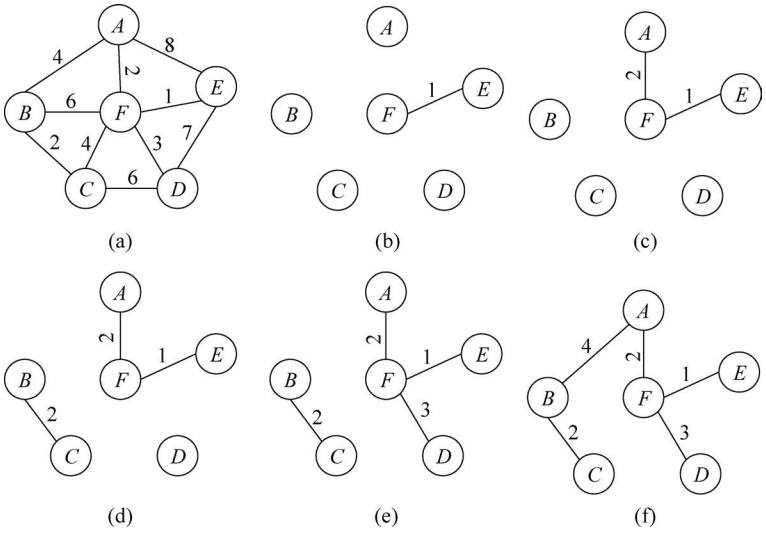


图 5-35 Kruskal 算法构造最小生成树

5.4.2 最短路径

对于带权图而言，可以将从某一个顶点到其余任意一个顶点的一条路径所经过边上的权值之和定义为该路径的带权路径长度，通常将带权路径长度最短的那条路径称为最短路径。最短路径的问题一般可分为两大类：从某一顶点到其余各顶点的最短路径和每对顶点间的最短路径。前者可以通过 Dijkstra 算法求解，后者可以通过 Floyd 算法求解。

1. 从某一顶点到其余各顶点的最短路径

Dijkstra 算法可用于求解图中某一顶点到其余各顶点的最短路径。假设 $G=\{V,\{E\}\}$ 是含有 n 个顶点的有向网，以该图中顶点 v 为源点，使用 Dijkstra 算法求顶点 v 到图中其余各顶点的最短路径的基本思想如下。

(1) 使用集合 S 记录已求得最短路径的终点，初始时 $S=\{v\}$。

(2) 选择一条长度最小的最短路径，该路径的终点 $w\in V-S$，将 w 并入 S，并将该最短路径的长度记为 D_w。

(3) 对于 $V-S$ 中任一顶点 s，将源点到顶点 s 的最短路径长度记为 D_s，并将顶点 w 到顶点 s 的弧的权值记为 D_{ws}，若 $D_w+D_{ws}<D_s$，则将源点到顶点 s 的最短路径的长度修改为 D_w+D_{ws}。

(4) 重复执行(2)和(3)，直到 $S=V$。

为了实现 Dijkstra 算法，可以使用邻接矩阵 Arcs 存储有向网，当 $i=j$ 时，Arcs[i][j] $=0$；当 $i\neq j$ 时，若下标为 i 的顶点到下标为 j 的顶点有弧且该弧的权值为 w，则 Arcs[i][j]$=w$，否则 Arcs[i][j]$=\infty$。使用列表 Dist 存储源点到每一个终点的最短路径的长度，并使用列表 Path 存储每一条最短路径中倒数第二个顶点的下标，通过对 Path 的处理可以得到从源点到每一个终点完整的最短路径，最后使用集合 flag 记录每一个顶点是否已经求得最短路径。

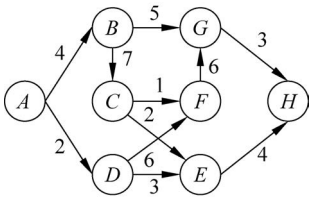


图 5-36　有向图

对于某一包含 n 个顶点的有向图，Dijkstra 算法共执行 $n-1$ 次，每一次的执行时间为 $O(n)$。因此该算法的时间复杂度为 $O(n^2)$。若使用带权的邻接表存储，算法的时间复杂度仍为 $O(n^2)$。

对如图 5-36 所示的有向图，以顶点 A 为源点，按照上述算法思想求源点到图中其余各顶点的最短路径的结果如表 5-2 所示。

表 5-2　算法的某一次执行结果

源　点	终　点	最短路径	路径长度
A	B	A,B	4
	C	A,B,C	11
	D	A,D	2
	E	A,D,E	5
	F	A,D,F	8
	G	A,B,G	9
	H	A,D,E,H	9

注意：若图中有负权值时，Dijkstra 算法并不一定适用。读者可以思考为什么。

2. 每对顶点间的最短路径

假设 $G=\{V,\{E\}\}$ 是含有 n 个顶点的有向网，通过 Dijkstra 算法可以求得图中每一对顶点间的最短路径，即依次以图中每一个顶点作为源点，执行 Dijkstra 算法。除此之外，还可以使用 Floyd 算法求图中每一对顶点间的最短路径，其基本思想如下。

(1) 对于图 G 中任意两个顶点 v 和 w，将顶点 v 到顶点 w 的最短路径的长度记为 D_{vw}，并依次判断其余各顶点是否为这两个顶点间最短路径上的顶点，具体判断过程如下。

对于除了顶点 v 和顶点 w 的任一顶点 u，将顶点 v 到顶点 u 的最短路径的长度记为 D_{vu}，并将顶点 u 到顶点 w 的最短路径的长度记为 D_{uw}，若 $D_{vu}+D_{uw}<D_{vw}$，则将 D_{vw} 的值修改为 $D_{vu}+D_{uw}$，即当前所得顶点 v 到顶点 w 的最短路径经过顶点 u。

(2) 重复执行(1)，直到图中每一对顶点间的最短路径都被求出。

为了实现 Floyd 算法，可使用邻接矩阵 Arcs 存储有向网，当 $i=j$ 时，Arcs[i][j] $=0$，当 $i\neq j$ 时，若下标为 i 的顶点到下标为 j 的顶点有弧且该弧的权值为 w，则 Arcs[i][j]$=w$，否则 Arcs[i][j]$=\infty$。通常使用二维数组 Dist 存储每一对顶点间的最短路径的长度，并使用二维数组 Path 存储每一条最短路径中倒数第二个顶点的下标，通过对 Path 的处理可以得到每一对顶点间完整的最短路径。Floyd 算法的时间复杂度为 $O(n^3)$(n 为图中顶点的总数目)。

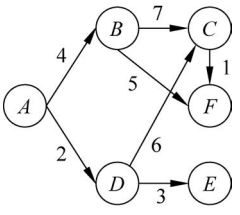


图 5-37　有向图

对如图 5-37 所示的有向图，按照上述算法思路求各顶点间的最短路径的结果如表 5-3 所示。

表 5-3 算法的某一次执行结果

源 点	终 点	最短路径	路径长度
A	B	A,B	4
	C	A,D,C	8
	D	A,D	2
	E	A,D,E	5
	F	A,B,F	9
B	C	B,C	7
	F	B,F	5
C	F	C,F	1
D	C	D,C	6
	E	D,E	3
	F	D,C,F	7

Floyd 算法也适用于带权无向图，此时可将无向图中任意顶点 v 和顶点 w 的无序对 (v,w) 看成有向图中顶点 v 和顶点 w 的有序对 $<v,w>$ 和 $<w,v>$，即无序对 (v,w) 的权值与有序对 $<v,w>$ 和 $<w,v>$ 的权值相等。在边权值为非负时，将每一个顶点作为源点执行 Dijkstra 算法也可以解决每对顶点之间的最短路径问题，其时间复杂度为 $O(n^3)$（n 为图中顶点的总数目）。

注意：Floyd 算法允许图中有带负权值的边，但不允许有包含带负权值的边组成的回路。读者可以思考为什么。

5.4.3 拓扑排序

拓扑排序是指构造拓扑序列的过程。对于一个包含 n 个顶点的 AOV 网（Activity On Vertex Network，即用顶点表示活动，用弧表示活动间的优先关系的有向图），假定将这 n 个顶点排列成一个线性序列 $S=v_1,v_2,\cdots,v_n$，如果该 AOV 网中存在从顶点 v_i 到顶点 v_j 的路径，那么在序列 S 中 v_i 必定出现在 v_j 之前，此时可将序列 S 称为该 AOV 网的拓扑序列。通常一个 AOV 网的拓扑序列并不唯一。

假设图 G 是一个包含 n 个顶点的有向图，对其进行拓扑排序的基本思想如下。

(1) 在图 G 中选择一个入度为 0 的顶点，并将其值输出。

(2) 将(1)中的顶点和以该顶点为弧尾的弧均从图 G 中删除。

(3) 重复执行(1)和(2)，直到所有顶点的值均被输出，或当前图中已经不存在入度为 0 的顶点（图 G 中包含回路）。

根据上述拓扑排序的基本思想，如图 5-38 所示的 AOV 网的拓扑序列的求解过程如图 5-39 所示，最终输出的拓扑序列为 C_1,C_2,C_3,C_4,C_6,C_5。

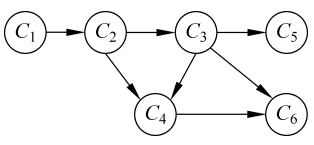


图 5-38 一个 AOV 网

对于含有 n 个顶点和 e 条弧的有向图，按上述算法思路进行拓扑排序时，可知计算各顶点入度的时间复杂度为 $O(e)$，并且建立存储入度为 0 的顶点的栈的时间复杂度为 $O(n)$。若该图中不包含回路，则每一个顶点进栈一次，出栈一

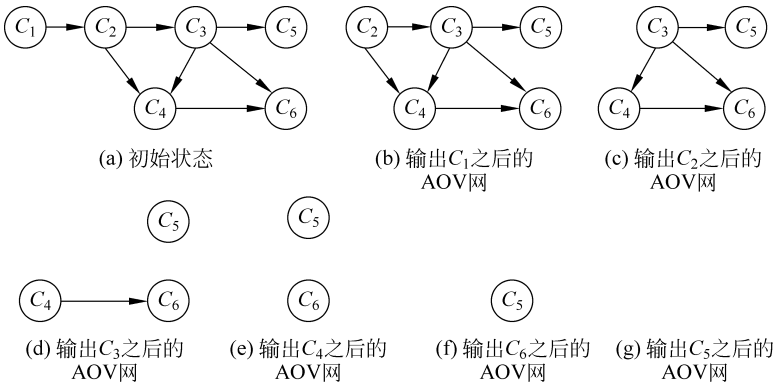


图 5-39　构造拓扑序列的过程

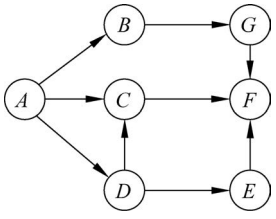


图 5-40　有向图

次，入度减 1 的操作共执行 e 次，经分析可知（可参考 DFS 算法），拓扑排序的时间复杂度为 $O(n+e)$。

如图 5-40 所示的有向图，按照上述算法对其进行拓扑排序的结果为：$ABGDCEF$。

注意：通常从入度为 0 的顶点开始或继续拓扑排序；若一个顶点有多个直接后继，则拓扑排序的结果通常不唯一；对于邻接矩阵是三角矩阵的图，存在拓扑序列，反之则不一定。此外，对于一个 AOV 网 G，还可以进行逆拓扑排序，具体步骤如下。

（1）在图 G 中选择一个出度为 0 的顶点，并将其值输出。

（2）将（1）中的顶点和以该顶点为弧头的弧均从 G 中删除。

（3）重复执行（1）和（2），直到所有顶点的值均被输出，或当前图中已经不存在出度为 0 的顶点（图 G 中包含回路）。

逆拓扑排序所得的序列称为逆向拓扑序列或逆拓扑序列。

5.4.4　关键路径

在 AOE 网（Activity On Edge，即弧表示活动、权值表示活动持续的时间、顶点表示事件的有向网）中，通过对源点到汇点的每一路径上的所有活动持续总时间（即各活动持续时间之和）进行计算，从而获得持续总时间最长的路径，通常把这一路径称为关键路径。

在 AOE 网中，仅有一个入度为 0 的顶点，称为源点，它表示整个工程的开始；仅有一个出度为 0 的顶点，称为汇点，它表示整个工程的结束。AOE 网必定是无环的，只有在某个顶点代表的事件发生后，以该顶点为弧尾的各有向边所代表的活动才能开始；只有以某顶点为弧头的各有向边所代表的活动都已经结束时，该顶点所代表的事件才能发生。

对于含有 n 个顶点的 AOE 网，E_i 表示该网中的某一事件，而 A_i 表示该网中的某一活动。在求该网的关键路径时，假设开始点是 E_1，从 E_1 到 E_i 的最长路径长度称为 E_1 的最早发生时间 EventEarly(i)，它决定了所有以 E_i 为弧尾的活动的最早开始时间，与某一事件的最早发生时间对应的是该事件的最晚发生时间 EventLate(i)。可以使用 ActivityEarly(i)表示活动 A_i 的最早开始时间，并使用 ActivityLate(i)来表示一个活动的最晚开始时间，这一时间是在不推迟整个项目完成的前提下，活动 A_i 最迟必须开始的时间。两者之差

ActivityLate(i)－ActivityEarly(i)表示完成活动 A_i 的时间余量，若活动 A_i 推迟开始或延迟完成的时间在该时间余量范围内，都不会影响整个项目的工期，并将时间余量为 0 的活动称为关键活动。显然，关键路径上的所有活动均为关键活动，因此提前完成非关键活动并不能加快项目的进度。

在如图 5-41 所示 AOE 网中，关键路径为(E_1，E_2，E_3，E_4，E_{11}，E_8，E_9，E_{10})。由此可知，关键活动为 A_1，A_2，A_3，A_{10}，A_{11}，A_{12}，A_{13}。关键活动 A_{10} 的最早发生时间是 30，若将其推迟，则会拖延整个项目的工期；而对于非关键活动 A_4 的最早发生时间也是 30，但若将其推迟 2 天后才开始，并不会影响整个项目的工期。因此，在项目管理时获取关键路径可以帮助识别项目中的关键活动，通过提高这些关键活动的执行效率，从而缩短整个项目的工期。

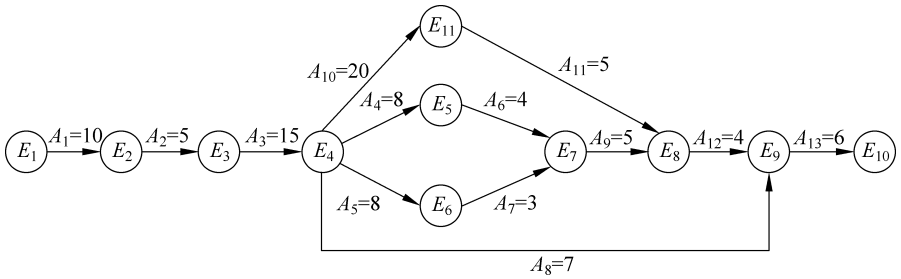


图 5-41 AOE 网

为了获取关键路径，需求出 AOE 网中每一个事件和活动的最早开始时间和最晚开始时间，它们的关系如下。

(1) 令 EventEarly(1)＝0，则有

$$\text{EventEarly}(i)=\text{Max}\{\ \text{EventEarly}(j)+W_{<j,i>}\}$$

$$<j,i>\in T \quad \text{且} \quad i=2,3,\cdots,n$$

其中，$<j,i>$是由 E_j 和 E_i 组成的弧，T 是所有以 E_i 为弧头的弧的集合，$W_{<j,i>}$表示弧$<j,i>$的权值。

(2) 令 EventLate(n)＝EventEarly(n)，则有

$$\text{EventLate}(i)=\text{Min}\{\text{EventLate}(j)-W_{<i,j>}\}$$

$$<i,j>\in S \text{ 且 } i=n-1,\cdots,1$$

其中，$<i,j>$是由 E_i 和 E_j 组成的弧，S 是所有以 E_i 为弧尾的弧的集合，$W_{<i,j>}$表示弧$<i,j>$的权值。

可以按照拓扑序列和逆向拓扑序列求得 EventEarly(i)和 EventLate(i)。

(3) 若 A_i 由弧$<j,k>$表示，该弧的权值为 $W_{<j,k>}$，则有

$$\text{ActivityEarly}(i)=\text{EventEarly}(j)$$

$$\text{ActivityLate}(i)=\text{EventLate}(k)-W_{<j,k>}$$

假设对于含有 n 个顶点的 AOE 网 G，对其求关键路径的算法思想如下。

(1) 对该网进行拓扑排序，求得每一个顶点对应的 EventEarly(i)。若拓扑序列的数目小于 n，则说明该网包含回路，因此不能求得关键活动；否则执行(2)。

(2) 按照逆向拓扑序列求得每一个顶点的 EventLate(i)。

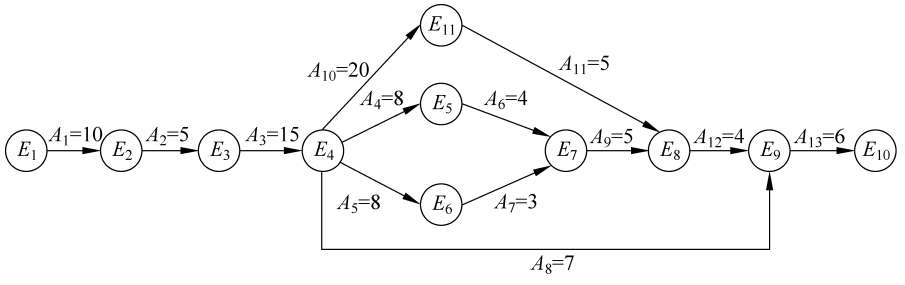
（3）通过每一个事件的 EventEarly(i)和 EventLate(i)求得每一个活动的 ActivityEarly(i)和 ActivityLate(i)。

（4）判断每一个活动的 ActivityEarly(i)和 ActivityLate(i)是否相等，若相等，则该弧为关键活动。

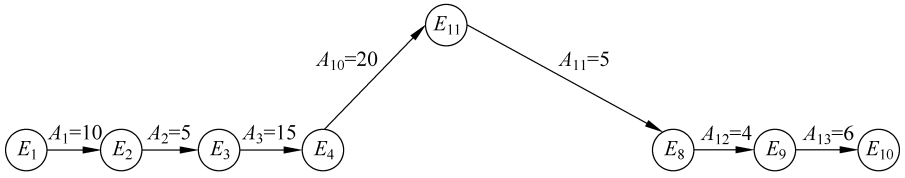
（5）所有关键活动构成的路径即为关键路径。

注意：关键路径上所有活动持续总时间即为完成项目需要的最少时间。

上述关键路径算法的时间复杂度为 $O(n+e)$（其中，n 为图中顶点的总数目，e 为图中边的总数目）。按照上述算法对如图 5-42(a)所示的 AOE 网求关键路径时，所有顶点和活动的开始时间如表 5-4 所示，最终得到的关键路径如图 5-42(b)所示。



(a)



(b)

图 5-42　AOE 网及其关键路径

从图 5-42 中可以看出，完成当前项目至少需要 10+5+15+20+5+4+6=65 天，而影响整个项目工期的关键任务是 A_1，A_2，A_3，A_{10}，A_{11}，A_{12}，A_{13}。

表 5-4　AOE 网中顶点和活动的开始时间

顶点	EventEarly	EventLate	活动	ActivityEarly	ActivityLate	ActivityLate－ActivityEarly
E_1	0	0	A_1	0	0	0
E_2	10	10	A_2	10	10	0
E_3	15	15	A_3	15	15	0
E_4	30	30	A_4	30	38	8
E_5	38	46	A_5	30	39	9
E_6	38	47	A_6	38	46	8
E_7	42	50	A_7	38	47	9
E_8	55	55	A_8	30	52	22
E_9	59	59	A_9	42	50	8
E_{10}	65	65	A_{10}	30	30	0
E_{11}	50	50	A_{11}	50	50	0
			A_{12}	55	55	0
			A_{13}	59	59	0

影响关键活动的因素有很多，任一活动持续时间的改变都可能会使关键路径中的关键活动发生变化。例如，对于如图 5-43(a)所示的 AOE 网，其关键路径如图 5-43(b)所示。

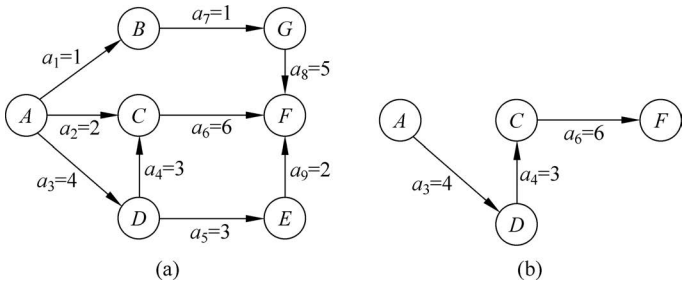


图 5-43 AOE 网及其关键路径

对于如图 5-43(a)所示的 AOE 网，若将 a_9 的持续时间改为如图 5-44(a)所示的 7，则该网的关键活动则为 a_3、a_5、a_9，如图 5-44(b)所示。

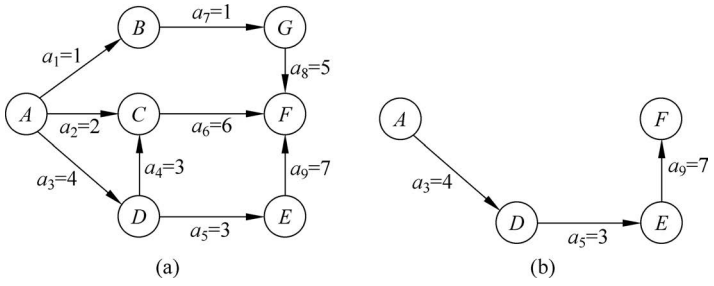


图 5-44 $a_9=7$ 时的 AOE 网及其关键路径

对于如图 5-43(a)所示的 AOE 网，若将 a_9 的持续时间改为 6，则该网的关键活动为 a_3、a_4、a_5、a_6、a_9，它们构成两条关键路径(A,D,C,F)和(A,D,E,F)，如图 5-45(b)所示。在这种情况下，若想缩短该网对应项目的完成工期，应提高所有关键路径中的关键活动的工效。

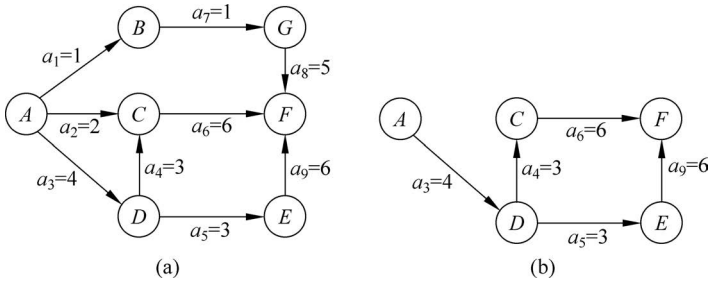


图 5-45 $a_9=6$ 时的 AOE 网及其关键路径

注意：关键路径上的所有活动都是关键活动，可以通过加快关键活动来缩短整个工期，但这样也可能会让关键活动变成非关键活动。此外，对于关键路径不唯一的 AOE 网，仅提高某一条关键路径上的活动速度并不能缩短工期，要加快所有关键路径上的关键活动才可能缩短工期。

小结

本章内容是历年考试的重点，主观题和客观题都有，后者居多。考生务必要掌握图的基本概念、存储结构和遍历等主要以客观题形式考查的内容，同时还要掌握图的基本应用，如最小生成树、最短路径、拓扑排序和关键路径等既能以客观题又能以主观题形式考查的知识点。图的邻接矩阵或邻接表存储、有向图的十字链表存储、无向图的邻接多重表存储也要引起重视。

考生复习时先从基本概念入手，然后是图的存储结构，至少要熟练掌握邻接矩阵存储，然后再掌握图的遍历算法思想和实现，最后才是图的基本应用。对于最小生成树、最短路径、拓扑排序和关键路径的算法思想千万不能死记硬背，而是要全面理解。反复研习历年真题中相关的试题可以帮助考生加深对这些知识的理解。