# 第六讲　图论初步

## § 6.1　引言

图论是运筹学的一个经典和重要的分支，它起源于欧拉（Euler）对七桥问题的抽象和论证。1936 年，匈牙利数学家柯尼希（König）出版了图论的第一部专著《有限图与无限图理论》，竖立了图论发展的第一座里程碑。此后，图论进入发展与突破的快车道，所研究的问题涉及经济管理、工业工程、交通运输、计算机科学与信息技术、通讯与网络技术等诸多领域。近几十年来，由于计算机技术和科学的飞速发展，大大地促进了图论研究和应用，图论的理论和方法已经渗透到物理、化学、通讯科学、建筑学、生物遗传学、心理学、经济学、社会学等学科中。

图论中所谓的“图”是指某类具体事物和这些事物之间的联系。如果我们用点表示这些具体事物，用连接两点的线段（直的或曲的）表示两个事物的特定的联系，就得到了描述这个“图”的几何形象。图论为任何一个包含了一种二元关系的离散系统提供了一个数学模型，借助于图论的概念、理论和方法，可以对该模型求解。

引例 6.1.1　最短路问题（SPP－shortest path problem）

一名货柜车司机奉命在最短的时间内将一车货物从甲地运往乙地。从甲地到乙地的公路网纵横交错，因此有多种行车路线，这名司机应选择哪条线路呢？假设货柜车的运行速度是恒定的，那么这一问题相当于需要找到一条从甲地到乙地的最短路。

引例 6.1.2　中国邮递员问题（CPP－chinese postman problem）

一名邮递员负责投递某个街区的邮件。如何为他（她）设计一条最短的投递路线（从邮局出发，经过投递区内每条街道至少一次，最后返回邮局）？由于这一问题是我国管梅谷教授 1960 年首先提出的，所以国际上称之为中国邮递员问题。

引例 6.1.3　旅行商问题（TSP－traveling salesman problem）

一名推销员准备前往若干城市推销产品。如何为他（她）设计一条最短的旅行路线（从驻地出发，经过每个城市恰好一次，最后返回驻地）？这一问题的研究历史十分悠久，通常称之为旅行商问题。

引例 6.1.4　指派问题（assignment problem）

一家公司经理准备安排 $N$ 名员工去完成 $N$ 项任务，每人一项。由于各员工的特点不同，不同的员工去完成同一项任务时所获得的回报是不同的。如何分配工作方案可以使总回报最大？

引例 6.1.5　运输问题（transportation problem）

某种原材料有 $M$ 个产地，现在需要将原材料从产地运往 $N$ 个使用这些原材料的工厂。假定 $M$ 个产地的产量和 $N$ 家工厂的需要量已知，单位产品从任一产地到任一工厂的运费已知，那么如何安排运输方案可以使总运输成本最低？

## § 6.2　图的基本概念

### § 6.2.1　图的定义

定义 6.2.1　称数学结构 $G = \{V(G), E(G), \psi_G\}$ 为一个图，其中 $V(G) = \{v_1, v_2, \ldots, v_n\}$ 称为图 $G$ 的**顶点集**（vertex set）或节点集（node set），$V(G)$ 中的每一个元素 $v_i$（$i = 1, 2, \ldots, n$）称为该图的一个**顶点**（vertex）或节点（node）；$E(G) = \{e_1, e_2, \ldots, e_m\}$ 称为图 $G$ 的**边集**（edge set），$E(G)$ 中的每一个元素 $e_k$（即 $V(G)$ 中某两个元素 $v_i, v_j$ 的无序对）记为 $e_k = (v_i, v_j)$ 或 $e_k = v_iv_j = e_k = v_jv_i$（$k = 1, 2, \ldots, m$），被称为该图的一条从 $v_i$ 到 $v_j$ 的**边**（edge）；$\psi_G$ 是从 $E(G)$ 到 $V(G)\times V(G)$ 的一个映射，它指定 $E(G)$ 中的每条边与 $V(G)$ 中的点组成的无序点对相对应。

若用小圆点表示点集 $V(G)$ 中的点，连线表示边集 $E(G)$ 中的边，则可用图形将图表示出来，称之为图的图形。我们常用图的图形代表图本身。

例 6.2.1　设 $V=\{v_1, v_2, v_3, v_4\}$，$E=\{e_1, e_2, e_3, e_4, e_5\}$，$\psi$：$E\to V\times V$ 定义为

$$\psi(e_1)=v_1v_2,\ \psi(e_2)=v_2v_3,\ \psi(e_3)=v_2v_3,\ \psi(e_4)=v_3v_4,\ \psi(e_5)=v_4v_4,$$

则 $G=\{V, E\}$ 是一个图，其图形如图 6.2.1 所示。

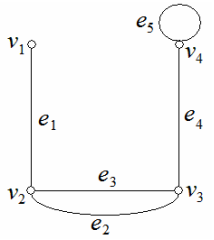



图 6.2.1

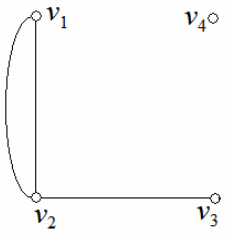



图 6.2.2

例 6.2.2　设 $V=\{v_1, v_2, v_3, v_4\}$，$E=\{v_1v_2, v_1v_2, v_2v_3\}$，则 $G=\{V, E\}$ 是一个图，其图形如图 6.2.2 所示。

定义 6.2.2　设 $e=uv$ 为图 $G$ 的一条边，我们称 $u$，$v$ 是 $e$ 的**端点**，$u$ 与 $v$ **相邻**，边 $e$ 与点 $u$（或 $v$）相**关联**；称 $u$ 是 $e$ 的起点，$v$ 是 $e$ 的终点。若两条边 $e_1$ 与 $e_2$ 有共同的端点，则称边 $e_1$ 与 $e_2$ 相邻；称有相同起点和终点的两条边为**重边**；称两端点均相同的边为**环**；称不与任何边相关联的点为**孤立点**。

无环且无重边的图称之为**简单图**。

例 6.2.1 和例 6.2.2 都不是简单图，因为例 6.2.1 中既含重边（$e_2$ 与 $e_3$）又含环（$e_5$），而例 6.2.2 中含重边（$v_1v_2$）。图 6.2.3 中给出的则是简单图。

任意两点均相邻的简单图称之为**完全图**。

$n$ 个点的完全图记为 $K_n$，图 6.2.4 中给出的分别是 $K_2$，$K_3$，$K_4$。

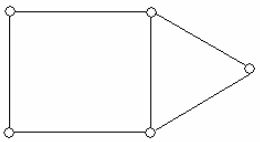


图 6.2.3

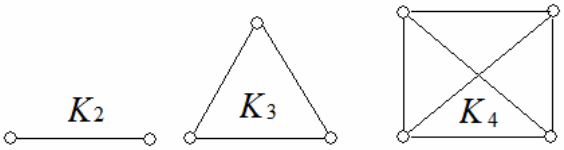



图 6.2.4

如果图 $G$ 的各条边都被赋予了方向，则称图 $G$ 为**有向图**。如果图 $G$ 的每条边 $e$ 都附有一个实数 $w(e)$，则称图 $G$ 为**赋权图**，实数 $w(e)$ 称为边 $e$ 的权（值）。

图 6.2.5 和图 6.2.6 分别给出了有向图和赋权图的例子；而图 6.2.7 则给出了有向赋权图的例子。

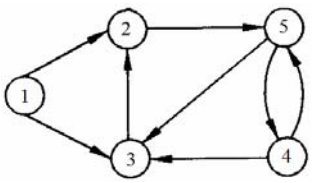



图 6.2.5

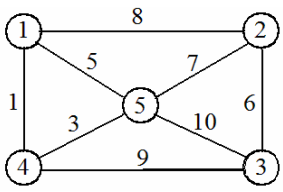



图 6.2.6

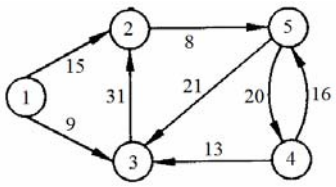



图 6.2.7

设 $v$ 为图 $G$ 的点，$G$ 中与 $v$ 相关联的边的条数（环计算两次）称为点 $v$ 的**度**，记为 $d_G(v)$，简记为 $d(v)$。

例如，在图 6.2.1 中，$d(v_1)=1$，$d(v_2)=d(v_3)=d(v_4)=3$；在图 6.2.2 中，$d(v_1)=2$，$d(v_2)=3$，$d(v_3)=1$，$d(v_4)=0$。

如果简单图 $G$ 的每个顶点都有相同的度数 $d$，则称 $G$ 为 $d$ 次**正则图**。

完全图 $K_n$ 一定是 $n$ 次正则图，如图 6.2.4。

定理 6.2.1　设 $G$ 是具有 $n$ 个顶点 $m$ 条边的图，则顶点度数的总和等于边数的 2 倍，即

$$\sum_{i=1}^{n} d(v_i)=2m\ 。$$

定理 6.2.2　完全图 $K_n$ 的边数为 $m=n(n-1)/2$。

### § 6.2.2　图的矩阵表示

一个图除了可以用图形表示之外，还可用矩阵来表示。用矩阵表示图有利于计算机处理。

设图 $G = \{V, E\}$，$V = \{v_1, v_2, \ldots, v_n\}$，$E = \{e_1, e_2, \ldots, e_m\}$。

无向图的**关联矩阵** $M(G) = (m_{ij})$ 是一个 $n \times m$ 矩阵，其中

$$m_{ij} = \begin{cases} 1, & \text{若 } v_i \text{ 与 } e_j \text{ 相关联} \\ 0, & \text{若 } v_i \text{ 与 } e_j \text{ 不相关联} \end{cases};$$

有向图的**关联矩阵** $M(G) = (m_{ij})$ 是一个 $n \times m$ 矩阵，其中

$$m_{ij} = \begin{cases} 1, & \text{若 } v_i \text{ 是 } e_j \text{ 的起点} \\ -1, & \text{若 } v_i \text{ 是 } e_j \text{ 的终点} \\ 0, & \text{若 } v_i \text{ 与 } e_j \text{ 不相关联} \end{cases}。$$

无向图的**邻接矩阵** $A(G) = (a_{ij})$ 是一个 $n$ 阶方阵，其中 $a_{ij}$ 为连接点 $v_i$ 与点 $v_j$ 之间边的数目；有向图的**邻接矩阵** $A(G) = (a_{ij})$ 是一个 $n$ 阶方阵，其中 $a_{ij}$ 为从点 $v_i$ 与出发到点 $v_j$ 终止的边的数目；**简单赋权有向图的邻接矩阵** $A(G) = (a_{ij})$ 是一个 $n$ 阶方阵，其中

$$a_{ij} = \begin{cases} w, & \text{若 } v_i v_j \in E \text{ 且 } w \text{ 是它的权值} \\ 0, & \text{若 } i = j \\ \infty, & \text{若 } v_i v_j \notin E \end{cases}。$$

无向赋权图的**边权矩阵** $W(G) = (w_{ij})$ 是一个 $3 \times m$ 方阵，其中 $w_{1j}$ 为第 $j$ 条边的起点的标号，$w_{2j}$ 为第 $j$ 条边的终点的标号，$w_{3j}$ 为第 $j$ 条边的权值。

下面的矩阵是图 6.2.6 的边权矩阵：

$$W = \begin{pmatrix} 1 & 1 & 1 & 2 & 2 & 3 & 3 & 4 \\ 2 & 4 & 5 & 3 & 5 & 4 & 5 & 5 \\ 8 & 1 & 5 & 6 & 7 & 9 & 10 & 3 \end{pmatrix}。$$

### § 6.2.3　子图

定义　设 $G = \{V(G), E(G), \psi_G\}$ 与 $H = \{V(H), E(H), \psi_H\}$ 为两个图。若 $V(H) \subseteq V(G)$，$E(H) \subseteq E(G)$，且 $\psi_H$ 是 $\psi_G$ 在 $E(H)$ 上的限制，则称 $H$ 是 $G$ 的**子图**。若 $H$ 是 $G$ 的子图，且 $V(H) = V(G)$，则称 $H$ 是 $G$ 的**生成子图**。若 $V(H) \subseteq V(G)$，$V(H) \neq \varnothing$，且对于 $v_i, v_j \in V(H)$，$v_i v_j \in E(G) \Rightarrow v_i v_j \in E(H)$，则称 $H$ 是 $G$ 的**导出子图**。

图 6.2.8 中，$H_1$ 与 $H_2$ 均为 $G$ 的子图，其中 $H_2$ 是 $G$ 的生成子图，$H_1$ 是 $G$ 的导出子图。

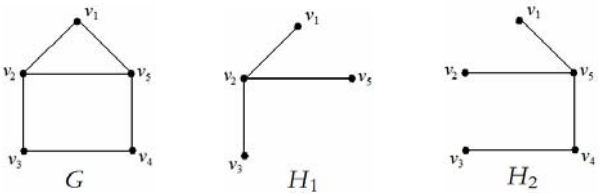



图 6.2.8

## § 6.3　最短路问题及其算法

### § 6.3.1　相关的概念

定义 6.3.1　设图 $G$ 不是赋权图。由图 $G$ 中点与边交替组成的序列

$$\Gamma = v_1 e_1 v_2 e_2 \ldots v_k e_k v_{k+1},$$

若满足 $e_i$ 的端点为 $v_i$ 与 $v_{i+1}$，$i = 1, 2, \ldots, k$，则称 Γ 为一条从起点 $v_1$ 到终点 $v_{k+1}$ 的长为 $k$ 的**通路**。

边不重复的通路称为**简单通路**；除起点与终点可以相同外，任意两点都不同的通路，称为**基本通路**，基本通路简称为**路**。显然，基本通路必为简单通路。

称起点与终点相同的通路为**回路**；边不重复的回路称为**简单回路**；起点与终点相同的长为正的基本通路称为**基本回路**，也称为**圈**。

如不引起混淆（如在简单图中），通路与回路均可用点序列来表达。

例 6.3.1　在图 6.3.1 中，取

$$\Gamma_1 = v_1v_2v_3，\ \Gamma_2 = v_1v_2v_3v_4v_2，\ \Gamma_3 = v_1v_2v_3v_2v_3v_4，$$

则 $\Gamma_1$，$\Gamma_2$，$\Gamma_3$ 分别是长为 2，4，5 的通路。其中 $\Gamma_1$ 与 $\Gamma_2$ 为简单通路，$\Gamma_1$ 为基本通路。

又取

$$C_1 = v_1v_2v_3v_4v_2v_5v_1，\ C_2 = v_1v_2v_5v_1，$$

则 $C_1$ 是长为 6 的简单回路，$C_2$ 是长为 3 的圈。

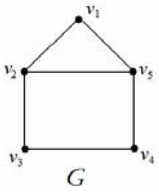


图 6.3.1

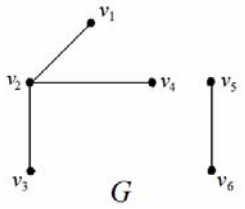


图 6.3.2

定义 6.3.2　任意两点之间均存在通路的图称为**连通图**，否则称为非连通图。非连通图中的连通子图，称为连通**分支**。

图 6.3.1 所示的图为连通图，而图 6.3.2 所示的图为非连通图，它含有两个连通分支。

定义 6.3.3　设图 $G$ 是赋权图，Γ 为 $G$ 中的一条路，则称 Γ 的各边权之和为路 Γ 的**长度**。对于 $G$ 的两个顶点 $u$ 和 $v$，从 $u$ 到 $v$ 的路一般不只一条，其中最短的一条称为从 $u$ 到 $v$ 的**最短路**；最短路的长称为从 $u$ 到 $v$ 的**距离**，记为 $d(u, v)$。

## § 6.3.2　固定起点到其余各点的最短路算法

寻求从一固定起点 $u_0$ 到其余各点的最短路的最有效算法之一是 Dijkstra（迪克斯特拉）算法，1959 年由 Dijkstra 提出。这个算法是一种迭代算法，它依据的是一个重要而明显的性质：**最短路是一条路，最短路上的任一子段也是最短路**。

Dijkstra 算法的基本思想是：按距 $u_0$ 从近到远为顺序，依次求得 $u_0$ 到图 $G$ 的各顶点的最短路和距离，直至顶点 $v_0$（或直至图 $G$ 的所有顶点）。

**Dijkstra 算法**

问题：设简单赋权图 $G = \{V, E\}$ 有 $n$ 个顶点，求 $G$ 中 $u_0$ 点到其它各点的距离及最短路。

为避免重复并保留每一步的计算信息，对$\forall v \in V$，定义两个标号：

$l(v)$——顶点 $v$ 的标号，表示从顶点 $u_0$ 到 $v$ 的一条路的权值；

$z(v)$——顶点 $v$ 的父节点标号，用以确定最短路的路线。

第一步　赋初值：令 $l(u_0) = 0$，对所有 $v \in V - \{u_0\}$，令 $l(v) = \infty$，$z(v) = u_0$；$S_0 = \{u_0\}$，$i = 0$。

第二步　若 $i = n - 1$，停止；否则令 $\overline{S}_i = V - S_i$，进行下一步。

第三步　更新标号：对每个 $v \in \overline{S}_i$，令

$$l(v) = \min_{u_i \in S_i}\{l(v),\ l(u_i) + w(u_i, v)\}；$$

如果 $l(v) > l(u_i) + w(u_iv)$，则 $z(v) = u_i$，否则 $z(v)$ 不变。

第四步　计算 $\min\limits_{v \in \overline{S}_i}\{l(v)\}$，并用 $u_{i+1}$ 记达到最小值的顶点，置 $S_{i+1} = S_i \cup \{u_{i+1}\}$，$i = i+1$，转第二步。

算法终止后，$u_0$ 到 $v$ 的距离由 $l(v)$ 的终值给出，从 $v$ 的父节点标号 $z(v)$ 追溯到 $u_0$，就得到 $u_0$ 到 $v$ 的最短路的路线。

例 6.3.2　求图 6.3.3 所示的图 $G$ 中 $v_1$ 到其余各顶点的最短路及其距离。

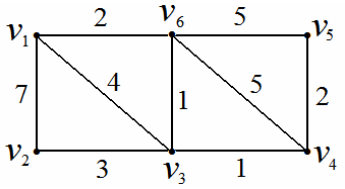


图 6.3.3

解：(1) 初始标号。$i = 1$。

$S_0 = \{v_1\}$，$v_1 = u_0$，参见图 6.3.4(a)。

(2) 用 $u_0 = v_1$ 对各顶点的标号进行更新。$i = 1$。

$\overline{S}_0 = \{v_2, v_3, v_4, v_5, v_6\}$，$\forall v \in \overline{S}_0$，由算法有：

$$l(v_2) = \{\infty, 0+7\} = 7，\ l(v_3) = \{\infty, 0+4\} = 4，$$
$$l(v_4) = \{\infty, 0+\infty\} = \infty，\ l(v_5) = \{\infty, 0+\infty\} = \infty，$$
$$l(v_6) = \{\infty, 0+2\} = 2。$$

由于 $v_2, v_3, v_6$ 的标号被 $v_1$ 更新，故这三个顶点的父节点为 $v_1$，即 $z(v_2) = z(v_3) = z(v_6) = v_1$，参见图 6.3.4(b)，数字边方框中的符号表示父节点。

又由于 $\min\limits_{v \in \overline{S}_0}\{l(v)\} = l(v_6) = 2$，故 $u_1 = v_6$，$S_1 = \{v_1, v_6\}$。参见图 6.3.4(c)。

(3) 用 $u_1 = v_6$ 对各顶点的标号进行更新。$i = 2$。

$\overline{S}_1 = \{v_2, v_3, v_4, v_5\}$，$\forall v \in \overline{S}_1$，由算法有：

$$l(v_2) = \{7, 2+\infty\} = 7，\ l(v_3) = \{4, 2+1\} = 3，\ l(v_4) = \{\infty, 2+5\} = 7，\ l(v_5) = \{\infty, 2+5\} = 7。$$

在此次迭代中，$v_2$ 的标号不变，$v_3$，$v_4$，$v_5$ 的标号被 $v_6$ 更新，故 $v_2$ 的父节点不变，$v_3$，$v_4$，$v_5$ 的父节点为 $v_6$，即 $z(v_2) = v_1$，$z(v_3) = z(v_4) = z(v_5) = v_6$。参见图 6.3.4(d)。

又由于 $\min\limits_{v \in \overline{S}_1}\{l(v)\} = l(v_3) = 3$，故 $u_2 = v_3$，$S_2 = \{v_1, v_6, v_3\}$。参见图 6.3.4(e)。

(4) 用 $u_2 = v_3$ 对各顶点的标号进行更新。$i = 3$。

$\overline{S}_2 = \{v_2, v_4, v_5\}$，$\forall v \in \overline{S}_2$，由算法有：

$$l(v_2) = \{7, 3+3\} = 6，\ l(v_4) = \{7, 3+1\} = 4，\ l(v_5) = \{7, 3+\infty\} = 7。$$

在此次迭代中，$v_5$ 的标号不变，$v_2$ 和 $v_4$ 的标号被 $v_3$ 更新，故 $v_5$ 的父节点不变，$v_2$ 和 $v_4$ 的父节点为 $v_3$，即 $z(v_5) = v_6$，$z(v_2) = z(v_4) = v_3$。参见图 6.3.4(f)。

又由于 $\min\limits_{v \in \overline{S}_2}\{l(v)\} = l(v_4) = 4$，故 $u_3 = v_4$，$S_3 = \{v_1, v_6, v_3, v_4\}$。参见图 6.3.4(g)。

(5) 用 $u_3 = v_4$ 对各顶点的标号进行更新。$i = 4$。

$\overline{S}_3 = \{v_2, v_5\}$，$\forall v \in \overline{S}_3$，由算法有：

$$l(v_2) = \{6, 4+\infty\} = 6，\ l(v_5) = \{7, 4+2\} = 6。$$

在此次迭代中，$v_2$ 的标号不变，$v_5$ 标号被 $v_4$ 更新，故 $v_2$ 的父节点不变，$v_5$ 的父节点为 $v_4$，即 $z(v_2) = v_3$，$z(v_5) = v_4$。参见图 6.3.4(h)。

又由于 $\min\limits_{v \in \overline{S}_3}\{l(v)\} = l(v_5) = 6$，故 $u_4 = v_5$，$S_4 = \{v_1, v_6, v_3, v_4, v_5\}$。参见图 6.3.4(i)。

(6) 用 $u_4 = v_5$ 对各顶点的标号进行更新。$i = 5$。

$\overline{S}_4 = \{v_2\}$，$\forall v \in \overline{S}_4 = \{v_2\}$，由算法有：$l(v_2) = \{6, 6+\infty\} = 6$。

在此次迭代中，$v_2$ 的标号不变，故 $v_2$ 的父节点不变，即 $z(v_2) = v_3$。参见图 6.3.4(i)。

又由于 $\min\limits_{v \in \overline{S}_4}\{l(v)\} = l(v_2) = 6$，故 $u_5 = v_2$，$S_5 = \{v_1, v_6, v_3, v_4, v_5, v_2\}$。参见图 6.3.4(j)。

(7) 由于 $i = 5 = n - 1$，停止。

**注**：① 迭代的终止条件也可使用 $S_{n-1} = \{v_1, \ldots, v_n\}$，或者 $\overline{S}_{n-1} = \varnothing$。

② 若寻找 $v_1$ 到某点 $v$ 的最短路的路由，则由 $v$ 开始追溯父节点直至 $v_1$。

例如，寻找 $v_1$ 到 $v_5$ 的最短路的路由，根据图 6.3.4(j)，从 $v_5$ 开始追溯父节点：$v_5$ 的父节点为 $v_4$，$v_4$ 的父节点为 $v_3$，$v_3$ 的父节点为 $v_6$，$v_6$ 的父节点为 $v_1$。于是 $v_1$ 到 $v_5$ 的最短路为：$v_1 \to v_6 \to v_3 \to v_4 \to v_5$。

再如，$v_1$ 到 $v_2$ 的最短路为：$v_1 \to v_6 \to v_3 \to v_2$。

③ 若求 $v_1$ 到某点 $v$ 的距离，则直接由 $l(v)$ 的终值确定。

例如，根据图 6.3.4(j)，由于 $l(v_5) = 6$，故 $v_1$ 到 $v_5$ 的距离为 6。再如，由于 $l(v_2) = 6$，故 $v_1$ 到 $v_2$ 的距离也为 6。

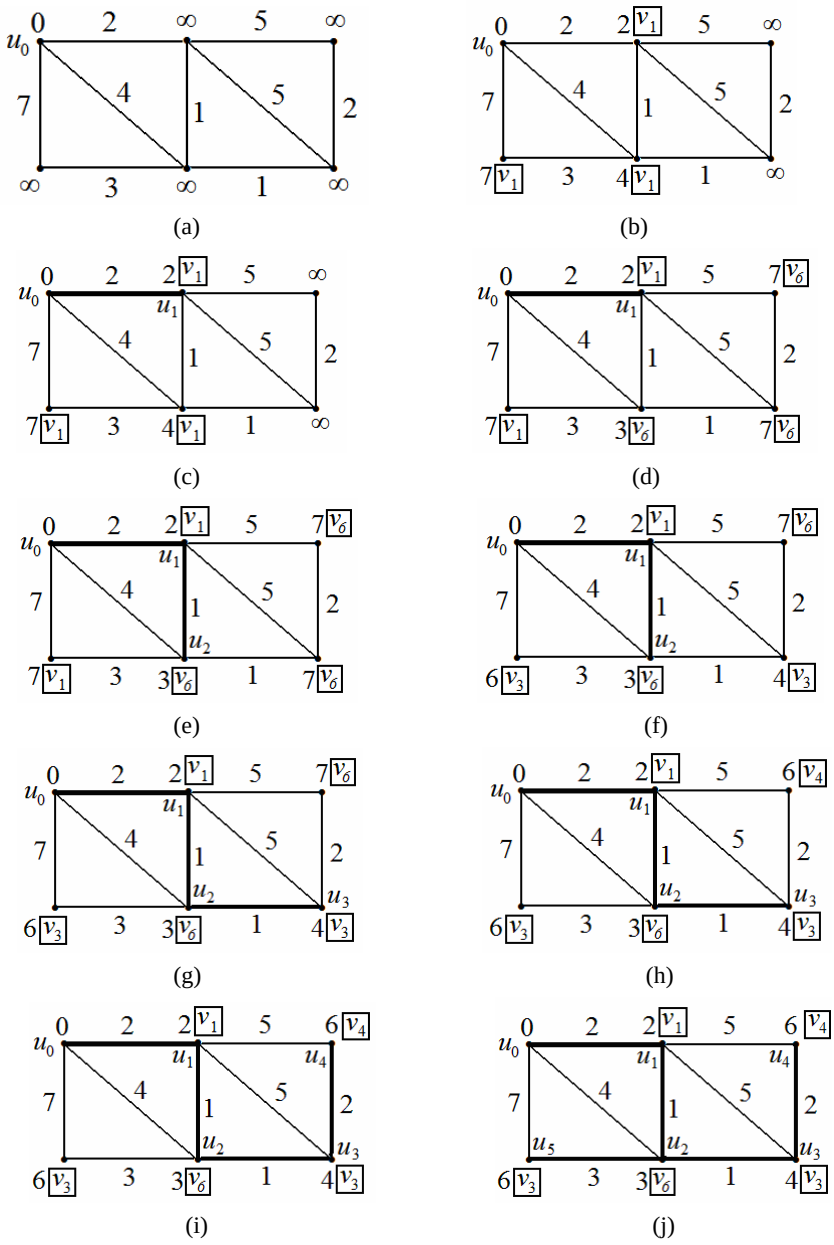



图 6.3.4

## § 6.3.3　每对顶点间的最短路算法

寻求赋权图中各对顶点之间最短路，显然可以调用 Dijkstra 算法。具体方法是：每次以不同的顶点作为起点，用 Dijkstra 算法求出从该起点到其余顶点的最短路径，反复执行 $n$ 次这样的操作，就可得到每对顶点之间的最短路。但这样做需要大量重复计算，效率不高。R. W. Floyd（弗洛伊德）另辟蹊径，提出了比这更好的算法，操作方式与 Dijkstra 算法截然不同。

Floyd 算法的基本思想是：从图的带权邻接矩阵 $A = [a(i, j)]_{n\times n}$ 开始，在 $A$ 中用插入顶点的方法依次构造出 $n$ 个矩阵 $D^{(1)}$、$D^{(2)}$、…、$D^{(n)}$，使最后得到的矩阵 $D^{(n)}$ 成为**图的距离矩阵**，即**矩阵 $D^{(n)}$ 的 $i$ 行 $j$ 列元素便是 $i$ 号顶点到 $j$ 号顶点的距离**。构造 $D^{(i)}$ 的同时，也引入一个**路由矩阵** $P^{(i)}$ 来记录两点间的最短路径。

构造矩阵 $D^{(k)}$，$k = 1, 2, \ldots, n$，采用如下的递推公式：

$D^{(0)} = [d_{ij}^{(0)}]_{n\times n} = A$：是带权邻接矩阵，$d_{ij}^{(0)}$ 表示从 $v_i$ 到 $v_j$ 的、中间不插入任何点的路径，即边 $v_iv_j$ 的权值；

$D^{(1)} = [d_{ij}^{(1)}]_{n\times n}$，其中 $d_{ij}^{(1)} = \min\{ d_{ij}^{(0)}, d_{i1}^{(0)} + d_{1j}^{(0)} \}$：$d_{ij}^{(1)}$ 表示从 $v_i$ 到 $v_j$ 的、中间最多只允许 $v_1$ 作为插入点的路径中最短路的长度；

$D^{(2)} = [d_{ij}^{(2)}]_{n\times n}$，其中 $d_{ij}^{(2)} = \min\{ d_{ij}^{(1)}, d_{i2}^{(1)} + d_{2j}^{(1)} \}$：$d_{ij}^{(2)}$ 表示从 $v_i$ 到 $v_j$ 的、中间最多只允许 $v_1$ 和 $v_2$ 作为插入点的路径中最短路的长度；

… …

$D^{(n)} = [d_{ij}^{(n)}]_{n\times n}$，其中 $d_{ij}^{(n)} = \min\{ d_{ij}^{(n-1)}, d_{i,n-1}^{(n-1)} + d_{n-1,j}^{(n-1)} \}$：$d_{ij}^{(n)}$ 表示从 $v_i$ 到 $v_j$ 的、中间最多只允许 $v_1, v_2, \ldots, v_{n-1}$ 作为插入点的路径中最短路的长度，即 $v_i$ 和 $v_j$ 之间的距离。

建立路由矩阵 $P^{(k)}$，$k = 1, 2, \ldots, n$，采用如下的递推公式：

$P^{(0)} = [p_{ij}^{(0)}]_{n\times n}$：$p_{ij}^{(0)}$ 表示从 $v_i$ 到 $v_j$ 的要经过点 $v_j$；

每求得一个 $D^{(k)}$ 时，按下列方式产生相应的新的 $P^{(k)} = [p_{ij}^{(k)}]_{n\times n}$，其中

$$p_{ij}^{(k)} = \begin{cases} k, & \text{若 } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ p_{ij}^{(k-1)}, & \text{否则} \end{cases} 。$$

即当 $v_k$ 被插入到 $v_i$ 与 $v_j$ 之间的最短路时，被记录在 $P^{(k)}$ 中。依次求 $D^{(n)}$ 时求得 $P^{(n)}$，可由 $P^{(k)}$ 来查找任何两点之间最短路的路由。

**Floyd 算法**

问题：设简单赋权图 $G = \{V, E\}$ 有 $n$ 个顶点，求 $G$ 中任意两点 $v_i$ 和 $v_j$ 之间的距离及最短路。

输入带权邻接矩阵 $A = [a(i, j)]_{n\times n}$。

第一步　赋初值：对所有 $i$ 和 $j$，$d(i, j) = a(i, j)$；当 $a(i, j) = \infty$ 时，path $(i, j) = 0$，否则 path $(i, j) = j$，$k = 1$。

第二步　更新 $d(i, j)$，path $(i, j)$：对所有 $i$ 和 $j$，若 $d(i, k) + d(k, j) \geq d(i, j)$，则转第三步；否则 $d(i, j) = d(i, k) + d(k, j)$，path $(i, j)$ = path $(i, k)$，$k = k + 1$，继续执行第三步。

第三步　重复第二步直到 $k = n + 1$。

在此：

① $d(i, j)$：$d_{ij}^{(k)}$ 。

② path $(i, j)$：$p_{ij}^{(k)}$；对应于 $d_{ij}^{(k)}$ 的路径上 $i$ 的后继点，最终的取值为 $i$ 到 $j$ 的最短路径上 $i$ 的后继点。例如，若

$$\text{path} = \begin{bmatrix} 1 & 2 & 2 & 2 & 2 \\ 5 & 2 & 5 & 5 & 5 \\ 4 & 2 & 3 & 4 & 4 \\ 5 & 5 & 5 & 4 & 5 \\ 1 & 1 & 3 & 3 & 5 \end{bmatrix}$$

则顶点 1 到顶点 3 的最短路径为 1→2→5→3。这是因为：path(1, 3) = 2，意味着顶点 1 的后继点为 2；又 path(2, 3) = 5，意味着顶点 2 的后继点为 5；同理，因 path(5, 3) = 3，从而顶点 5 的后继点为 3。故 1→2→5→3 便是顶点 1 到顶点 3 的最短路径。

## § 6.3.4　最短路算法的 Matlab 实现

根据 Dijkstra 算法和 Floyd 算法的步骤，[4]和[5]中分别给出了相应的 Matlab 程序。

**Dijkstra 算法的 Matlab 实现**

```
% Dijkstra 算法
% 输入带权邻接矩阵 w
n = size(w,1);
```

```
w1 = w(1,:);

%  赋初值
for i = 1:n
    l(i) = w1(i);
    z(i) = 1;
end
s = [];
s(1) = 1;
u = s(1);
k = 1;
l;
z;

while k < n
    %  更新 l(v)和 z(v)
    for i = 1:n
        for j = 1:k
            if i~=s(j)
                if l(i) > l(u)+w(u,i)
                    l(i) = l(u)+w(u,i);
                    z(i) = u;
                end
            end
        end
    end
    l;
    z;

    %  求 v*
    ll = l;
    for i = 1:n
        for j = 1:k
            if i~=s(j)
                ll(i) = ll(i);
            else
                ll(i) = inf;
            end
        end
    end

    lv = inf;
    for i = 1:n
        if ll(i) < lv
            lv = ll(i);
            v = i;
        end
    end
```

```
        lv;
        v;

        s(k+1) = v;
        k = k+1;
        u = s(k);

    end
    l
    z
```

**Floyd 算法的 Matlab 实现**

```
% Floyd 算法
%  输入带权邻接矩阵 a
n = size(a,1)
D = a;
path = zeros(n,n);
for i = 1:n
    for j = 1:n
        if D(i,j)~=inf
            path(i,j) = j;
        end
    end
end
for k = 1:n
    for i = 1:n
        for j = 1:n
            if D(i,k)+D(k,j) < D(i,j)
                D(i,j) = D(i,k)+D(k,j);
                path(i,j) = path(i,k);
            end
        end
    end
end
D
path
```

例 6.3.3　分别用 Dijkstra 算法和 Floyd 算法的 Matlab 程序求解例 6.3.2。

解：输入带权邻接矩阵

$$\begin{pmatrix} 0 & 7 & 4 & \inf & \inf & 2 \\ 7 & 0 & 3 & \inf & \inf & \inf \\ 4 & 3 & 0 & 1 & \inf & 1 \\ \inf & \inf & 1 & 0 & 2 & 5 \\ \inf & \inf & \inf & 2 & 0 & 5 \\ 2 & \inf & 1 & 5 & 5 & 0 \end{pmatrix}。$$

(1) 运行 Dijkstra 算法的程序，输出结果为

l = [0　6　3　4　6　2]；z = [1　3　6　3　4　1]。

于是，由 l 和 z 可以得到 $v_1$ 到其余各点的距离和最短路的路由。

例如，$v_1$ 到 $v_5$ 的距离由 l(5) = 6 给出；其最短路的路由根据 z(5) = 4，z(4) = 3，z(3) = 6，z(6) = 1 得到：$v_1 \to v_6 \to v_3 \to v_4 \to v_5$。

(2) 运行 Floyd 算法的程序，输出结果为

D = [ 0 6 3 4 6 2
6 0 3 4 6 4
3 3 0 1 3 1
4 4 1 0 2 2
6 6 3 2 0 4
2 4 1 2 4 0 ]

path = [ 1 6 6 6 6 6
3 2 3 3 3 3
6 2 3 4 4 6
3 3 3 4 5 3
4 4 4 4 5 4
1 3 3 3 3 6 ]

例如，$v_1$ 到 $v_5$ 的距离由 D(1, 5) = 6 给出；其最短路的路由根据 path(1, 5) = 6，path(6, 5) = 3，path(3, 5) = 4，path(4, 5) = 5 得到：$v_1 \to v_6 \to v_3 \to v_4 \to v_5$。

例 6.3.4　对于图 6.3.5 给出的有向图，用 Dijkstra 算法的 Matlab 程序求出节点 1 到节点 4 的最短路，用 Floyd 算法的 Matlab 程序求出任意两点间的最短路。

解：输入带权邻接矩阵

$$\begin{pmatrix} 0 & 50 & inf & inf & inf \\ inf & 0 & inf & inf & 80 \\ inf & 30 & 0 & 20 & inf \\ inf & inf & inf & 0 & 70 \\ 65 & inf & 100 & inf & 0 \end{pmatrix}。$$

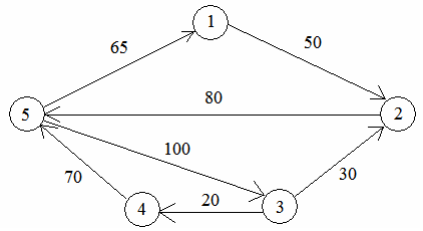


图 6.3.5

(1) 运行 Dijkstra 算法的程序，输出结果为

l = [0　50　230　250　130]；z = [1　1　5　3　2]。

于是，由 l 和 z 可以得到节点 1 到其余各点的距离和最短路的路由。

例如，节点 1 到节点 4 的距离由 l(4) = 250 给出；其最短路的路由根据 z(4) = 3，z(3) = 5，z(5) = 2，z(2) = 1 得到：1→2→5→3→4。

(2) 运行 Floyd 算法的程序，输出结果为

D = [ 0 50 230 250 130
145 0 180 200 80
155 30 0 20 90
135 185 170 0 70
65 115 100 120 0 ]

path = [ 1 2 2 2 2
5 2 5 5 5
4 2 3 4 4
5 5 5 4 5
1 1 3 3 5 ]

例如，节点 1 到节点 4 的距离由 D(1, 4) = 250 给出；其最短路的路由根据 path(1, 4) = 2，path(2, 4) = 5，path(5, 4) = 3，path(3, 4) = 4 得到：1→2→5→3→4。

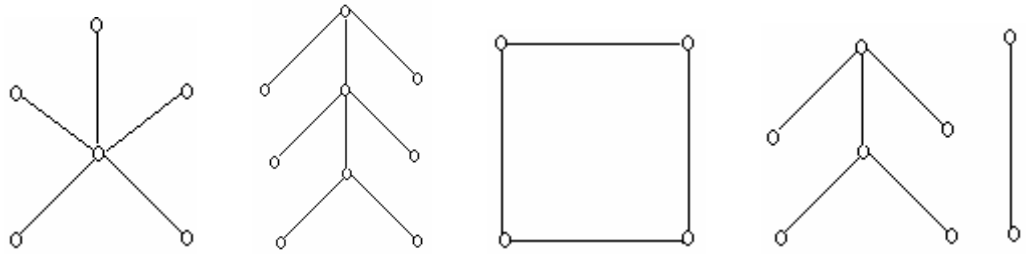
# § 6.4　树及其算法

## § 6.4.1　相关的概念

树（tree）在图论中是相当重要的一类图，它非常类似于自然界中的树。树的性质非常好，应用相当广泛[1]。

定义 6.4.1　无圈的连通图称为**树**。

例如，图 6.4.1 给出的 $G_1$ 和 $G_2$ 是树，但 $G_3$ 和 $G_4$ 则不是树。



$G_1$　　$G_2$　　$G_3$（有圈）　　$G_4$（不连通）

图 6.4.1

定理 6.4.1　设 $G$ 是具有 $n$ 个点 $m$ 条边的图，则以下关于树的命题等价。

(1) $G$ 是树；

(2) $G$ 中任意两个不同点之间存在唯一的路；

(3) $G$ 连通，删去任一条边均不连通；

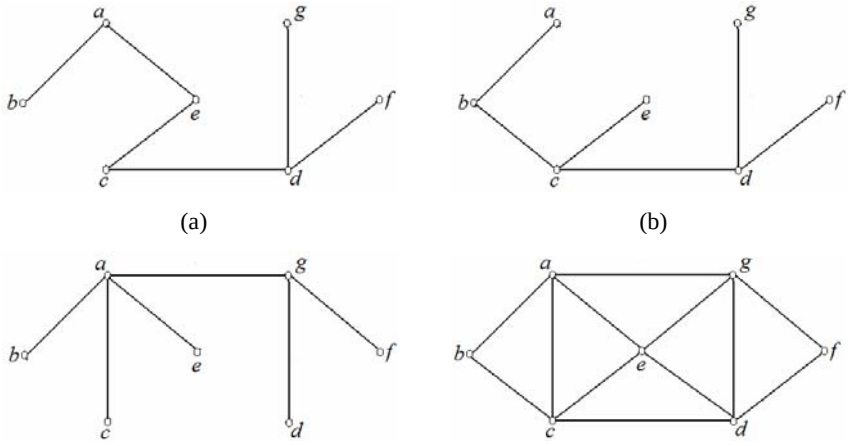
(4) $G$ 连通，且 $n = m + 1$；

(5) $G$ 无圈，且 $n = m + 1$；

(6) $G$ 无圈，添加任一条边可得唯一的圈。

定义 6.4.2　若图 $G$ 的生成子图 $H$ 是树，则称 $H$ 为 $G$ 的**生成树**或**支撑树**。

一般来讲，一个图的生成树不唯一。例如，在图 6.4.2 中，(a)、(b)、(c) 均是 (d) 的生成树。




(a)　　(b)

(c)　　(d)

图 6.4.2

定理 6.4.2　连通图的生成树一定存在。

证明　给定连通图 $G$，若 $G$ 无圈，则 $G$ 本身就是自己的生成树。若 $G$ 有圈，则任取 $G$ 中一个圈 $C$，记删去 $C$ 中一条边后所得之图为 $G'$。显然 $G'$ 中圈 $C$ 已经不存在，但 $G'$ 仍然连通。若 $G'$ 中还有圈，再重复以上过程，直到得到一个无圈的连通图 $H$。易知 $H$ 是 $G$ 的生成树。证毕。

定理 6.4.2 的证明方法也是求生成树的一种方法，称为“破圈法”。

定义 6.4.3　在赋权图 $G$ 中，边权之和最小（大）的生成树称为 $G$ 的**最小（大）生成树**。

## § 6.4.2　最小生成树算法

一个简单连通图只要不是树，其生成树就不唯一，而且非常多。一般地，$n$ 个顶点地完全图，其不同地生成树个数为 $n^{n-2}$。因而，寻求一个给定赋权图的最小生成树，一般是不能用穷举法的。例如，30 个顶点的完全图有 $30^{28}$ 个生成树，$30^{28}$ 有 42 位，即使用最现代的计算机，在我们的有生之年也是无法穷举的。所以，穷举法求最小生成树是无效的算法，必须寻求有效的算法。

在求最小生成树的有效算法中，最著名的两个是 Kruskal（克罗斯克尔）算法和 Prim（普瑞姆）算法，其迭代过程都是基于贪婪法来设计的。

1．求最小生成树的 Kruskal 算法

**Kruskal 算法的直观描述**

假设 $T_0$ 是赋权图 $G$ 的最小生成树，$T_0$ 中的边和顶点均涂成红色，初始时 $G$ 中的边均为白色。

① 将所有顶点涂成红色；

② 在白色边中挑选一条权值最小的边，使其与红色边不形成圈，将该白色边涂红；

③ 重复②直到有 $n-1$ 条红色边，这 $n-1$ 条红色边便构成最小生成树 $T_0$ 的边集合。

例如，对于图 6.4.3(a) 给出的赋权图，按照上述的步骤，容易求出其最小生成树。其中 (b)、(c) 分别是第一步和第二步，(d) 是最后结果。

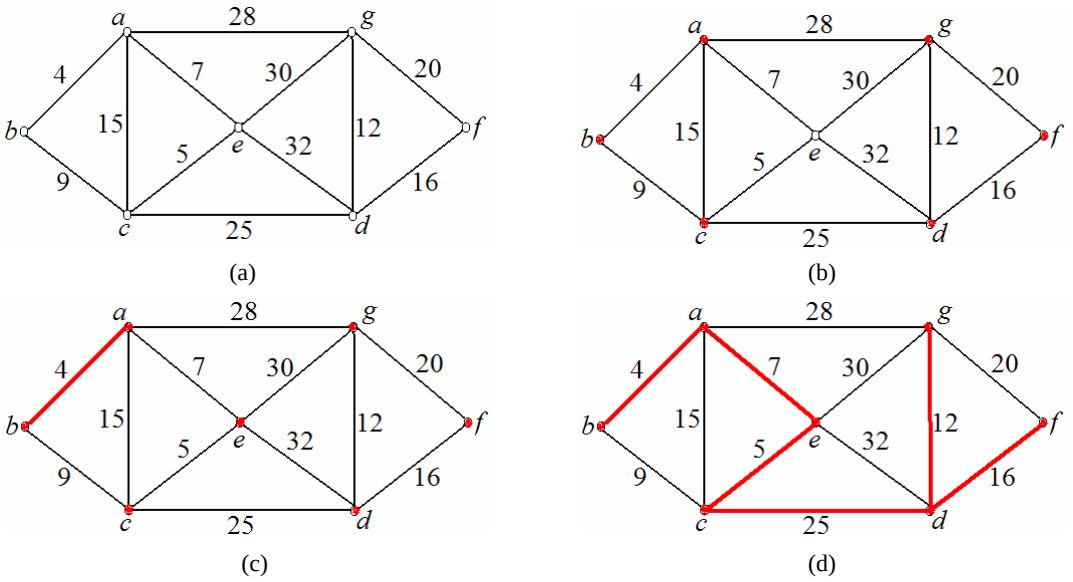



图 6.4.3

**Kruskal 算法的基本思想**

设 $T_0$ 和 $C(T_0)$ 分别为图 $G$ 的最小生成树的边集及其权值，初始状态均为空，算法结束时 $T_0$ 包含最小生成树的所有边，$C(T_0)$ 表示最小生成树的权值。令 $VS$ 是一个不相交的节点的集合，初始状态时 $VS = \{\{v_1\}, \{v_2\}, \ldots, \{v_n\}\}$。算法的主要步骤是每次从边集 $E$ 中选出一条未处理的有最小权值的边 $(u, v)$ 进行分析，如果 $u$ 和 $v$ 同属于 $VS$ 的一个元素集，则将边 $(u, v)$ 删除，如果 $u$ 和 $v$ 分属于 $VS$ 的两个元素集，则将边 $(u, v)$ 加入到 $T_0$ 中，并将这两个元素集合并为一个元素集，然后再从 $E$ 中另选权值最小的边进行处理，直至找到一棵最小生成树为止。

**Kruskal 算法的步骤**

第一步　$T_0 \leftarrow \varnothing$，$C(T_0) \leftarrow 0$，$VS \leftarrow \varnothing$，将 $E$ 中的边按权值从小到大排成序列 $Q$。

第二步　对所有 $v \in V$，$VS \leftarrow \{v\}$，即 $VS = \{\{v_1\}, \{v_2\}, \ldots, \{v_n\}\}$。

第三步　如果 $|VS| = 1$，输出 $T_0$ 和 $C(T_0)$，停止。否则进行下一步。

第四步　从 $Q$ 中取出权值最小的边 $(u, v)$，并从 $Q$ 中删除 $(u, v)$。

第五步　如果 $u$，$v$ 在 $VS$ 的元素集 $V_1$、$V_2$ 中且 $V_1 = V_2$，则转第四步。否则进行下一步。

第六步　$T_0 \leftarrow T_0 \cup \{(u, v)\}$，$V \leftarrow V_1 \cup V_2$，$C(T_0) \leftarrow C(T_0) + w(u, v)$，转第三步。

例 6.4.1　用 Kruskal 算法求图 6.4.3(a) 给出的赋权图的最小生成树。

解：将图的边按照权值从小到大进行排列：

| 边 | $(a, b)$ | $(c, e)$ | $(a, e)$ | $(b, c)$ | $(d, g)$ | $(a, c)$ | $(d, f)$ | $(f, g)$ | $(c, d)$ | $(a, g)$ | $(e, g)$ | $(d, e)$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 权 | 4 | 5 | 7 | 9 | 12 | 15 | 16 | 20 | 25 | 28 | 30 | 32 |

操作 Kruskal 算法，迭代 9 步完成最小生成树的寻找。操作过程的各个步骤列于表 6.4.1，结果显示于图 6.4.4。

表 6.4.1

| 步骤 | 选出边 $e$ | $w(e)$ | 操作 | $VS$ | $T_0$ | $C(T_0)$ |
|---|---|---|---|---|---|---|
| 1 | $(a, b)$ | 4 | 加到 $T_0$ 中 | $\{\{a, b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}\}$ | $\{(a, b)\}$ | 4 |
| 2 | $(c, e)$ | 5 | 加到 $T_0$ 中 | $\{\{a, b\}, \{c, e\}, \{d\}, \{f\}, \{g\}\}$ | $\{(a, b), (c, e)\}$ | 9 |
| 3 | $(a, e)$ | 7 | 加到 $T_0$ 中 | $\{\{a, b, c, e\}, \{d\}, \{f\}, \{g\}\}$ | $\{(a, b), (c, e), (a, e)\}$ | 16 |
| 4 | $(b, c)$ | 9 | 删除 | $\{\{a, b, c, e\}, \{d\}, \{f\}, \{g\}\}$ | $\{(a, b), (c, e), (a, e)\}$ | 16 |
| 5 | $(d, g)$ | 12 | 加到 $T_0$ 中 | $\{\{a, b, c, e\}, \{d, g\}, \{f\}\}$ | $\{(a, b), (c, e), (a, e), (d, g)\}$ | 28 |
| 6 | $(a, c)$ | 15 | 删除 | $\{\{a, b, c, e\}, \{d, g\}, \{f\}\}$ | $\{(a, b), (c, e), (a, e), (d, g)\}$ | 28 |
| 7 | $(d, f)$ | 16 | 加到 $T_0$ 中 | $\{\{a, b, c, e\}, \{d, g, f\}\}$ | $\{(a, b), (c, e), (a, e), (d, g), (d, f)\}$ | 44 |
| 8 | $(f, g)$ | 20 | 删除 | $\{\{a, b, c, e\}, \{d, g, f\}\}$ | $\{(a, b), (c, e), (a, e), (d, g), (d, f), (c, d)\}$ | 44 |
| 9 | $(c, d)$ | 25 | 加到 $T_0$ 中 | $\{\{a, b, c, e, d, g, f\}\}$ | $\{(a, b), (c, e), (a, e), (d, g), (d, f), (c, d)\}$ | 69 |

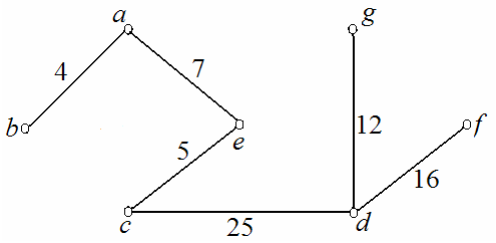


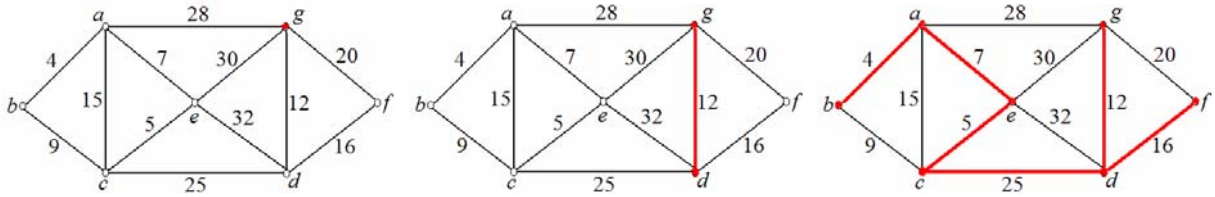

图 6.4.4

2．求最小生成树的 Prim 算法

**Prim 算法的直观描述**

假设 $T_0$ 是赋权图 $G$ 的最小生成树。任选一个顶点将其涂红，其余顶点为白点；在一个端点为红色，另一个端点为白色的边中，找一条权最小的边涂红，把该边的白端点也涂成红色；如此，每次将一条边和一个顶点涂成红色，直到所有顶点都成红色为止。最终的红色边便构成最小生成树 $T_0$ 的边集合。

例如，对于图 6.4.3(a) 给出的赋权图，按照上面的描述，容易求出其最小生成树。其中图 6.4.5(a)、(b) 分别是第一步和第二步，(c) 是最后结果。




(a) (b) (c)

图 6.4.5

**Prim 算法的基本思想**

设 $T_0$ 和 $C(T_0)$ 分别为图 $G$ 的最小生成树的边集及其权值，初始状态均为空，算法结束时 $T_0$ 包含最小生成树的所有边，$C(T_0)$ 表示最小生成树的权值。先指定一个顶点为初始访问点，记做 $v_0$，将 $v_0$ 加到“通过点”的集合 $V'$中，然后找出跨接在“通过点”集合 $V'$与“未通过点”集合 $V-V'$之间权最小的边 $e$ 作为“通过边”加入 $T_0$ 中，并将 $e$ 在 $V-V'$中的端点转到 $V'$中。重复上述过程直至 $V'=V$ 为止。

**Kruskal 算法的步骤**

第一步　$T_0 \leftarrow \varnothing$，$C(T_0) \leftarrow 0$，$V' \leftarrow \{v_0\}$。

第二步　对每一个点 $v \in V-V'$，$L(v) \leftarrow c(v, v_0)$（如果 $(v, v_0) \notin E$，则 $c(v, v_0) = \infty$）。

第三步　如果 $V'=V$，输出 $T_0$，$C(T_0)$，停止。否则进行下一步。

第四步　在 $V-V'$中找一点 $u$，使

$$L(u) = \min\{L(v) \mid v \in V-V'\},$$

并将 $V'$中与 $u$ 邻接的点记为 $w$，$e=(w, u)$。

第五步　$T_0 \leftarrow T_0 \cup \{e\}$，$C(T_0) \leftarrow C(T_0) + C(e)$，$V' \leftarrow V' \cup \{v_0\}$。

第六步　对所有 $v \in V - V'$，如 $c(v, u) < L(v)$，则 $L(v) \leftarrow c(v, u)$，否则 $L(v)$ 不变。

第七步　转第三步。

例 6.4.2　用 Prim 算法求图 6.4.3(a) 给出的赋权图的最小生成树。

解：为简便起见，将操作 Prim 算法的步骤列于表 6.4.2，结果参加图 6.4.4。

表 6.4.2

| 步骤 | $u$ | $L(b)$ | $L(c)$ | $L(d)$ | $L(e)$ | $L(f)$ | $L(g)$ | $e$ | $V'$ | $T_0$ | $C(T_0)$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | $a$ | 4 | 15 | ∞ | 7 | ∞ | 28 | | $\{a\}$ | ∅ | 0 |
| 2 | $b$ | — | 9 | ∞ | 7 | ∞ | 28 | $(a,b)$ | $\{a, b\}$ | $\{(a, b)\}$ | 4 |
| 3 | $e$ | — | 5 | 32 | — | ∞ | 28 | $(a,e)$ | $\{a, b, e\}$ | $\{(a, b), (a, e)\}$ | 11 |
| 4 | $c$ | — | — | 25 | — | ∞ | 28 | $(c,e)$ | $\{a, b, e, c\}$ | $\{(a, b), (a, e), (c, e)\}$ | 16 |
| 5 | $d$ | — | — | — | — | 16 | 12 | $(c,d)$ | $\{a, b, e, c, d\}$ | $\{(a, b), (a, e), (c, e), (c, d)\}$ | 41 |
| 6 | $g$ | — | — | — | — | 16 | — | $(d,g)$ | $\{a, b, e, c, d, g\}$ | $\{(a, b), (a, e), (c, e), (c, d), (d, g)\}$ | 53 |
| 7 | $f$ | — | — | — | — | — | — | $(d, f)$ | $\{a, b, e, c, d, g, f\}$ | $\{(a, b), (a, e), (c, e), (c, d), (d, g), (d, f)\}$ | 69 |

## § 6.4.3　最小生成树算法的 Matlab 实现

根据 Kruskal 算法和 Prim 算法的步骤，许多人都编写了相应的 Matlab 程序。在此选用[5]中给出的 Matlab 程序。

**Kruskal 算法的 Matlab 实现**

```
% Kruskal 算法
%  输入边权矩阵 b
b = []
[B,i] = sortrows(b',3);   %  按边权（b 的第三行）从小到大重新排列矩阵 b
B = B';
m = size(b,2);
n = ？;   %  输入顶点的数目
t = 1:n;
T = []; k = 0; c = 0;
for i = 1:m
    if t(B(1,i))~=t(B(2,i))   %  判断第 i 条边是否与树中的边形成圈
        k = k+1;
        T(1:2,k) = B(1:2,i);
        c = c+B(3,i);
        tmin = min(t(B(1,i)),t(B(2,i)));
        tmax = max(t(B(1,i)),t(B(2,i)));
        for j = 1:n
            if t(j)==tmax
                t(j) = tmin;
            end
        end
    end
    if k==n-1
        break;
    end
end
T, c
```

**Prim 算法的 Matlab 实现**

```
% Prim 算法
%  输带权邻接矩阵 a
a = []
n = size(a,1);
T = []; c = 0; v = 1; sb = 2:n;    % 1 是第一个红色点，sb 是白点集
%  构造初始候选边集
for j = 2:n
    b(1,j-1) = 1;
    b(2,j-1) = j;
    b(3,j-1) = a(1,j);
end
while size(T,2) < n-1
    [Y,i] = min(b(3,:));    %  在候选边集中找最短边
    T(:,size(T,2)+1) = b(:,i);
    c = c+b(3,i);
    v = b(2,i);      % v 是新红点
    temp = find(sb==b(2,i));
    sb(temp) = [];
    b(:,i) = [];
    %  调整候选边集
    for j = 1:length(sb)    % length(sb)  返回 sb 的行数和列数中最大的
        d = a(v,b(2,j));
        if d < b(3,j)
            b(1,j) = v;
            b(3,j) = d;
        end
    end
end
T, c
```

例 6.4.3　用分别 Kruskal 算法和 Prim 算法的 Matlab 程序，求图 6.4.3(a) 给出的赋权图的最小生成树。

解：(1) 用 Kruskal 算法

输入带权邻接矩阵

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 2 & 3 & 3 & 4 & 4 & 4 & 5 & 6 \\ 2 & 3 & 5 & 7 & 3 & 4 & 5 & 5 & 6 & 7 & 7 & 7 \\ 4 & 15 & 7 & 28 & 9 & 25 & 5 & 32 & 16 & 12 & 30 & 20 \end{pmatrix}。$$

运行 Kruskal 算法的程序，输出结果为

```
T =
     1     3     1     4     4     3
     2     5     5     7     6     4
c =   69
```

根据输出结果画出的最小生成树与图 6.4.4 相同。

(2) 用 Prim 算法

输入带权邻接矩阵

$$\begin{pmatrix} 0 & 4 & 15 & \text{inf} & 7 & \text{inf} & 28 \\ 4 & 0 & 9 & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 15 & 9 & 0 & 25 & 5 & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & 25 & 0 & 32 & 16 & 12 \\ 7 & \text{inf} & 5 & 32 & 0 & \text{inf} & 30 \\ \text{inf} & \text{inf} & \text{inf} & 16 & \text{inf} & 0 & 20 \\ 28 & \text{inf} & \text{inf} & 12 & 30 & 20 & 0 \end{pmatrix}。$$

运行 Prim 算法的程序，输出结果为

```
T =
     1     1     5     3     4     4
     2     5     3     4     7     6
     4     7     5    25    12    16
c =   69
```

根据输出结果画出的最小生成树与图 6.4.4 相同。

### § 6.4.4　关于最小生成树算法的注释

最小生成树的 Kruskal 算法是 1956 年由 Kruskal 提出的。随后在 1957 年，领导贝尔实验室数学研究室的 Prim 得到了他的算法。

Kruskal 算法的时间复杂性以 $O(m\log_2 m)$ 为界，当边数较多或是一个完全图时，$m \approx (n-1)^2$，则时间复杂性近似于 $O(n^2\log_2 n)$。而 Prim 算法的时间复杂性为 $O(n^2)$，所以，如果图的连通度较高（最高为完全图），以 Prim 算法较好，如果图的连通度较低，特别当 $m \approx O(n)$ 时，则 Kruskal 算法更合适。

在实际应用中，还会遇到求一个赋权图的最大生成树的问题。比如，某图的边权代表的是利润或效益，则最终的问题很可能就是求其最大生成树。事实上，从上面两个算法可以看出，只要边权的选择改为从大到小，求最小生成树的算就可以用来求最大生成树了。

## 实验作业

**设备更新策略**。某工厂的某台机器可连续工作 4 年，决策者每年年初都要决定机器是否需要更新。若购置新的，就要支付一定的购置费用；若继续使用，则要支付一定的维修与运行费用，而且随着机器的使用年限费用逐年增多。计划期（4 年）中每年年初的购置价格及各个年限内维修与运行费用由下表给出，试制订今后 4 年的机器更新计划，使总的支付费用最少。

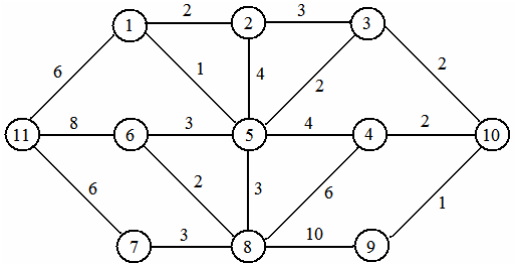
| 第 $i$ 年初 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 购置费（万元） | 2.5 | 2.6 | 2.8 | 3.1 |
| 使用年限 | 1 | 2 | 3 | 4 |
| 每年的维修与运行费（万元） | 1 | 1.5 | 2 | 4 |

又如果已知不同役龄机器年末的处理价格如下表所示，那末在这计划期内机器的最优更新计划又会怎样？

| | 第 1 年末 | 第 2 年末 | 第 3 年末 | 第 4 年末 |
|---|---|---|---|---|
| 机器处理价（万元） | 2.0 | 1.6 | 1.3 | 1.1 |

**最大容量路径**。在一个计算机通讯网络中，某一计算机欲呼叫另一台计算机并进行数据传输。若传输数据量很大，又要求了传输速度，则通常需要沿容量最大的路径进行数据传输。

假设该通讯网络对应于下面的无向图 $G$，其上每条边的权值代表容量（带宽），即通过该边的最大容量。求两个给定节点之间容量最大的路径。




**参考文献：**

[1] 王树禾，图论，科学出版社，2004。
[2] 杜端甫，运筹图论，北京航空航天大学出版社，1990。
[3] Robert Sedgewick 著，林琪译，C＋＋算法——图算法，清华大学出版社，2003。
[4] 赵静，但琦，数学建模与数学实验，高等教育出版社，2000。
[5] 傅鹏等，数学实验，科学出版社，2000。