第二章 图与遍历算法

§1 图的基本概念和术语

- 无向图(undirected graph)

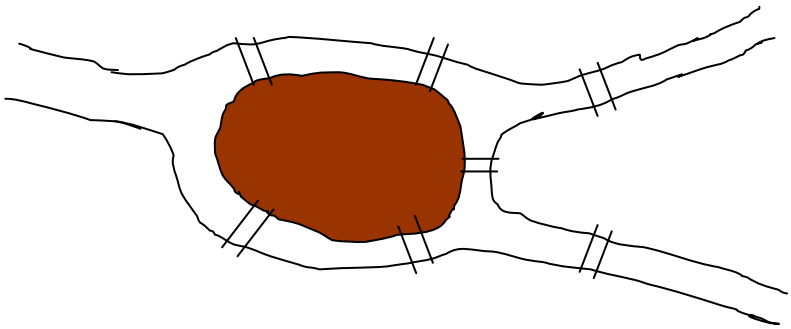


图 2-1-1 哥尼斯堡七桥

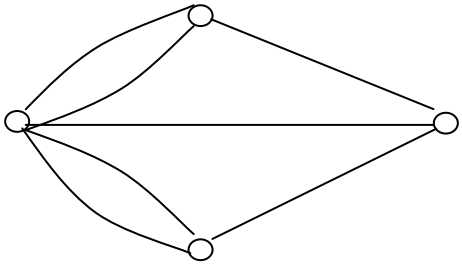


图 2-1-2 Euler 图

无向图，简称图，是一个用线（边）连接在一起的结点（顶点）的集合。严格地说，图是一个三元组 G=(V, E, I)，其中，V 是顶点的集合，E 是边的集合，而 I 是关联关系，它指明了 E 中的每条边与 V 中的每个顶点之间的关联关系：每条边必定连接两个而且只有两个顶点，它们称为该边的端点。有边相连的两个顶点称为相邻的。连接顶点 v 的边的条数称为 v 的度，记做 d(v). 图 G=(V, E, I)中顶点的度与边数有如下的 Euler 公式

$$\sum_{v\in V} d(v) = 2|E| \qquad (2.1.1)$$

由公式(2.1.1)可知，*图中奇度顶点的个数一定是偶数*。

没有重复边的图称为简单图，n 阶完全图是指具有 n 个顶点而且每两个顶点之间都有边连接的简单图，这样的图的边数为 n(n-1)/2.以下是 1～4 阶的完全图：

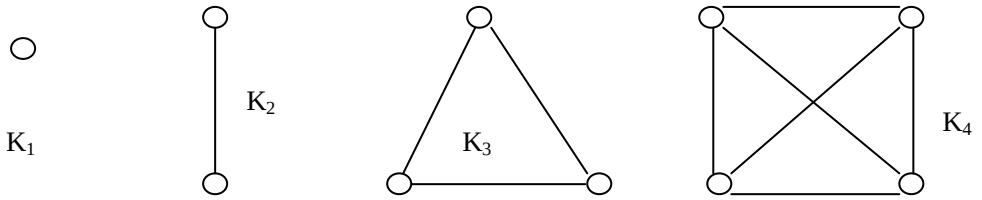


图 2-1-3 几个完全图

另一类特殊的图是偶图（也叫二部图），它的顶点集分成两部分 V_1，V_2，每部分中的顶点之间不相连（没有边连接）。

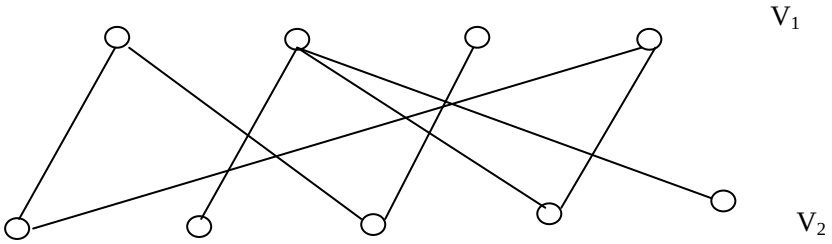


图 2-1-4　一个偶图

当然，k-部图，是指图的顶点集被分成k个部分，同部分的顶点之间没有边相连。

设图 G 的顶点集是$V=\{v_1,v_2,\cdots,v_n\}$，边集是$E=\{e_1,e_2,\cdots,e_m\}$，则顶点与顶点之间的邻接关系可以用如下矩阵表示：

$$\begin{array}{c} \\ v_1 \\ v_2 \\ \vdots \\ v_n \end{array}\begin{array}{c} \begin{array}{cccc} v_1 & v_2 & \cdots & v_n \end{array} \\ \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \end{array} = A$$

称为邻接矩阵，其中

$$a_{ij}=\begin{cases} 1 & \text{如果}(v_i,v_j)\text{是}G\text{的一条边} \\ 0 & \text{否则} \end{cases}$$

顶点与边之间的关联关系可以用如下矩阵表示：

$$\begin{array}{c} \\ v_1 \\ v_2 \\ \vdots \\ v_n \end{array} \begin{array}{c} \begin{array}{cccc} e_1 & e_2 & \cdots & e_m \end{array} \\ \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1} & b_{n1} & \cdots & b_{nm} \end{pmatrix} \end{array} = M$$

称为关联矩阵，其中

$$b_{ij} = \begin{cases} 1 & \text{如果 } v_i \text{ 是边 } e_j \text{的一个端点} \\ 0 & \text{否则} \end{cases}$$

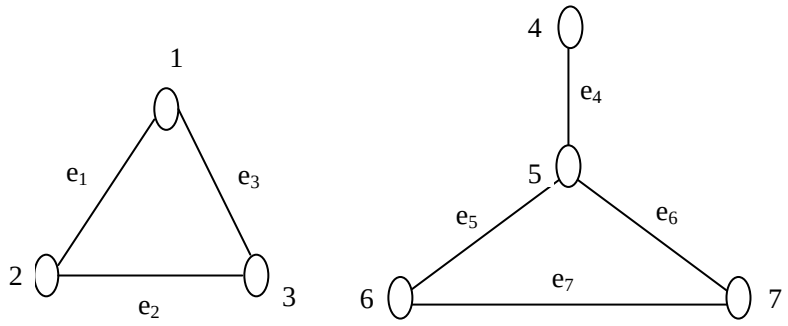


图 2-1-5 一个具有 7 个定点、7 条边的图

图 2-1-5 的邻接矩阵 A、关联矩阵 M 分别为：

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \quad M = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

图的另一个重要概念是路径，区分为途径、迹和路。

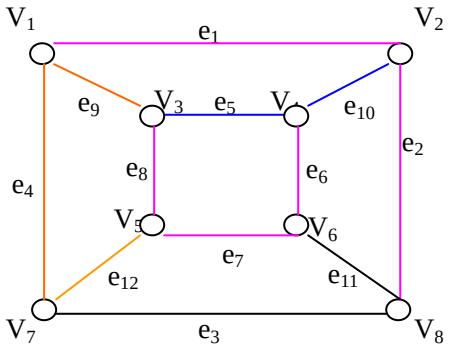
途径：顶点与边交叉出现的序列

$$v_0\ e_1\ v_1\ e_2\ v_2\ \cdots\ e_l\ v_l \qquad (2.1.2)$$

其中 e_l 的端点是 v_{i-1} 和 v_i 。**迹**是指边不重复的途径，而顶点不重复的途径称为**路**。起点和终点重合的途径称为闭途径，起点和终点重合的迹称为闭迹，顶点不重复的闭迹称为**圈**。没有圈的图称为**森林**。

如果存在一条以 u 为起点、v 为终点的途径，则说顶点 u 可达顶点 v。如果图 G 中任何两个顶点之间都是可达的，则说图 G 是连通的。如果图 G 不是连通的，则其必能分成几个连通分支。图 2-1-6 是连通的，而图 2-1-5 不是连通的，它有

两个连通分支。



一条途径：

$v_1e_1v_2e_{10}v_4e_5v_3e_9v_1e_1v_2e_2v_8$

一条迹：

$v_1e_1v_2e_{10}v_4e_5v_3e_9v_1e_4v_7$

一条路：

$v_1e_1v_2e_{10}v_4e_5v_3e_8v_5e_{12}v_7$

图 2-1-6　立方体

不含圈的连通图称为**树**。森林的每个连通分支都是树，也就是说，森林是由树组成的。对于连通图，适当去掉一些边后（包括去掉零条边），会得到一个不含圈、而且包含所有顶点的连通图，它是一棵树，称为原图的**生成树**。一棵具有 n 个顶点的树的边数恰好为 $n-1$。所以，一个具有 k 个连通分支的森林恰好有 $n-k$ 条边。一个具有 n 个顶点的连通图至少有 n-1 条边；一个具有 n 个顶点，k 个连通分支的图至少有 $n-k$ 条边.

定理 2.1.1 如果 G 是具有 n 个顶点、m 条边的图，则下列结论成立：

1. 若 G 是树， 则 m = n-1；
2. 若 G 是连通图，而且满足 m = n-1，则 G 是树；
3. 若 G 不包含圈，而且满足 m = n-1，则 G 是树；
4. 若 G 是由 k 棵树构成的森林，则 m = n-k ；
5. 若 G 有 k 个连通分支，而且满足 m = n-k，则 G 是森林。

由图 G 的部分顶点和部分边按照它们在 G 中的关联关系构成的图称为 G 的子图，如果子图 H 的顶点集恰是 G 的顶点集，则 H 称为 G 的生成子图。可见，当 G 是连通图时，它的生成树是一种特殊的生成子图，它是 G 的既连通又边数最少的一个生成子图。一个连通图的生成树不是唯一的。

- **有向图**

描述单行道系统就不能用无向图，因为它不能指明各条路的方向。所谓有向图实际上是在无向图的基础上进一步指定各条边的方向。在有向图中每一条有向边可以用一对顶点表示：$e=(u,v)$，u 为始点，v 为终点，这条边是由顶点 u 指向顶点 v 的。

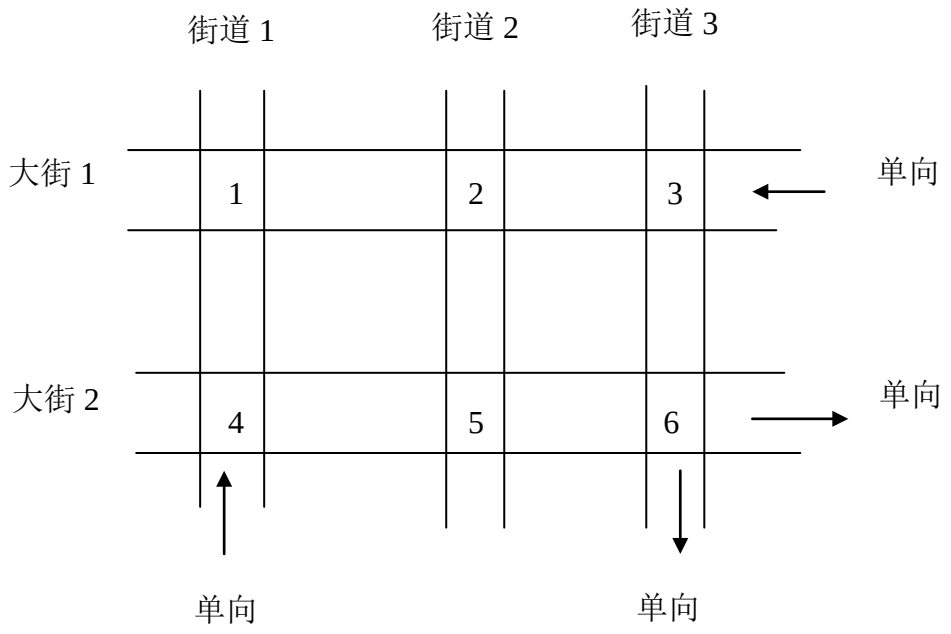


图 2-1-7 具有单行线的交通示意图

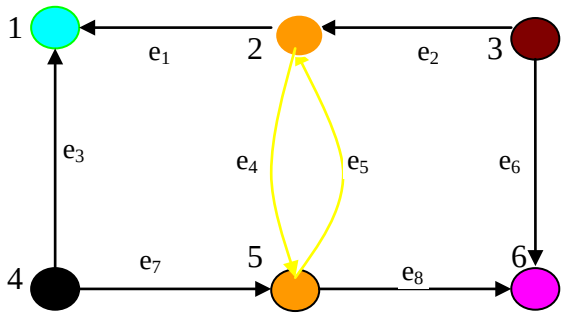


图 2-1-8
一个有向图及其双向连通分支

指向顶点 v 的有向边的条数称为顶点 v 的入度，记做 $d^-(v)$；而由顶点 u 出发的有向边的条数称为顶点 u 的出度，记做 $d^+(u)$。一个顶点 w 的出度与入度之和称为该顶点的度，记做 $d(w)$，即 $d(w)=d^+(w)+d^-(w)$。有向图 $G=(V,E,I)$ 的顶点的度和边数之间有如下的关系：

$$|E|=\sum_{v\in V}d^-(v)=\sum_{v\in V}d^+(v) \qquad (2.1.3)$$

类似地，有向图的表示也可以用邻接矩阵和关联矩阵，但是为指明边的方向，只用 0,1 两个元素是不够的。设有向图 G 的顶点集和边集分别是 $V=\{v_1,v_2,\cdots,v_n\}$ 和 $E=\{e_1,e_2,\cdots,e_m\}$，则邻接矩阵定义如下：

$$\begin{array}{c} \\ v_1 \\ v_2 \\ \vdots \\ v_n \end{array}\begin{array}{c} \begin{array}{cccc} v_1 & v_2 & \cdots & v_n \end{array} \\ \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \end{array} = A$$

其中

$$a_{ij} = \begin{cases} 1 & \text{如果}(v_i, v_j)\text{是}G\text{的一条有向边} \\ 0 & \text{否则} \end{cases}$$

关联矩阵定义如下：

$$\begin{array}{c} \\ v_1 \\ v_2 \\ \vdots \\ v_n \end{array}\begin{array}{c} \begin{array}{cccc} e_1 & e_2 & \cdots & e_m \end{array} \\ \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1} & b_{n1} & \cdots & b_{nm} \end{pmatrix} \end{array} = M$$

其中

$$b_{ij} = \begin{cases} 1 & \text{如果 } v_i \text{ 是有向边 } e_j\text{的始点} \\ -1 & \text{如果 } v_i \text{ 是有向边 } e_j\text{的终点} \\ 0 & \text{其它} \end{cases}$$

如，图 2-1-8 的关联矩阵为

$$M = \begin{pmatrix} -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 \end{pmatrix}$$

每个列上恰有一个 1 和一个 -1,代表一条有向边的始点和终点。

在有向图中，许多概念都可以通过无向图的相关概念加“有向”二字得到，如：有向边、有向途径、有向迹、有向路、有向圈，等等。有向图和无向图可以相互转化：将一个无向图的每条边都规定方向后，即得到有向图，其称为原无向图的一个定向图；将一个有向图的各条有向边的方向去掉，即得到一个无向图，其称为原有向图的基础图。

有向图中也有一些概念不能由无向图通过简单地附加“有向”一词而得到。如，连通，有向图 D 说是连通的是指其基础图是连通的。如果 D 中任意两个顶点都是相互有向可达的，则说有向图 D 是双向连通的（或叫强连通）。这里，顶点 u 可达顶点 v（有向可达）是指存在一条以 u 为起点、v 为终点的有向路。这里

的起点、终点不能互为替换，即，u 可达 v 不能保证 v 也可达 u。

有向图 2-1-8 就是连通的，但不是双向连通的，因为从任何顶点出发，都没有到达顶点 3 的有向路。不是双向连通的有向图可以分解成几个双向连通分支。图 2-1-8 共有 5 个双向连通分支，分别用不同的颜色标出。

- **赋权图**

一般的赋权图是指对图的每条边 e 赋予一个实数值 W(e)。如架设连接各城镇的交通路网，需考虑各段线路的修建费用；在运输网络中要考虑网络各段线路的容量，等等。

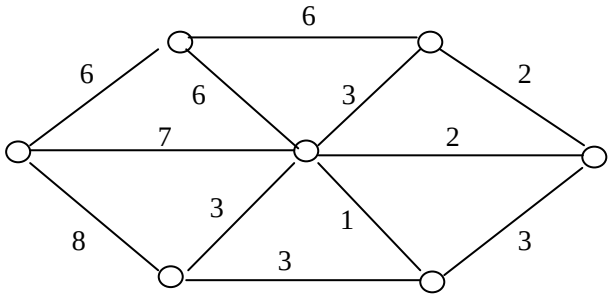
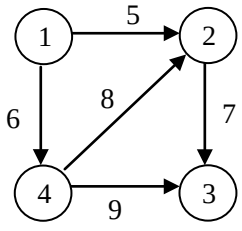


图 2-1-9　一个交通路网

注意，描述一个赋权图时，如果顶点v_i,v_j之间有一条边连接，而且权值为a，则其邻接矩阵中的(i,j)元素为a（而不再是 1），其余元素统一取为 0 或一个充分大的数，视问题而定。

- **图的邻接链表表示**

除了邻接矩阵表示和关联矩阵表示，在数据结构上也常常采用邻接链表的方法表示一个图。这是把邻接于每个顶点的点做成一个表连接到这个顶点上。图 2-1-10 给出了一个有向图(a)和它的邻接链表，其中，(b)是图 G 的邻接矩阵表示；(c)是各点的邻接表；(d)给出各点之间链接起来的结构。



(a)

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

(b)

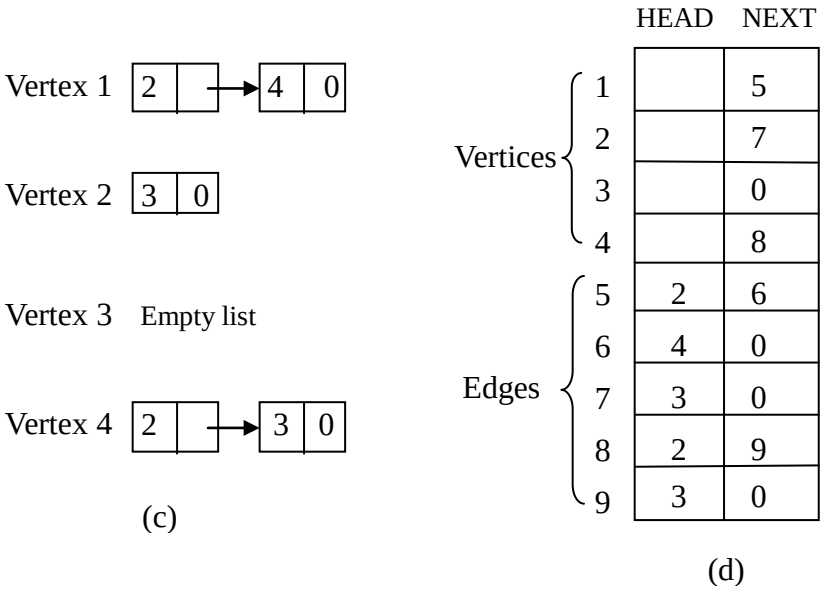


图 2-1-10 G 及其邻接链表

§2 图的遍历(搜索)算法

- **有根树与有向树**

树是一个没有圈的连通图。如果指定树的一个顶点为根，则这棵树称为有根树。在有根树中，邻接根的顶点称为根的儿子，而根称为这些儿子的父亲。对于不是根的顶点，除了它的父亲之外其它与之邻接的顶点都称为该顶点的儿子，该顶点也自然称为它的这些儿子的父亲。没有儿子的顶点称为叶顶点。

从根到每一个叶顶点都有一条唯一的路。这些路的最长者的长度称为该树的高度；不是根的顶点v不通过其父顶点而能到达的所有顶点同v一起生成一棵以v为根的子树，这棵子树的高度称为顶点v在原树中的高度。树的根到每个顶点v都有一条唯一的路，这条路的长度称为顶点v在树中的深度。如在图 2-2-1 的二叉树中，树的高度为 4，顶点 D,E 的深度都是 2；顶点 G 的高度为 1，深度为 3。可见，树中每个顶点的高度与深度的和不超过树的高度。

二叉树是这样一棵有根树，它的每个顶点至多有两个儿子。叶顶点深度都相同，而且除了叶顶点以外，每个顶点都恰有两个儿子的有根树称为完整的二叉树。高度为k的完整二叉树恰有$2^{k+1}-1$个顶点，它的所有顶点按照从上到下、从左到右的顺序可以编号为：$1,2,\cdots,2^{k+1}-1$。从这样编号的完整二叉树中的某一位置开始，删掉后面编号的所有顶点及与之关联的边所得到的二叉树称为一棵完全二叉树。

h 表示二叉树的高度

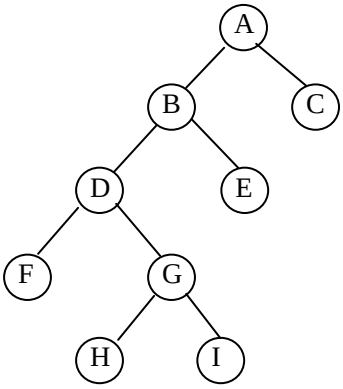


图 2-2-1 一棵二叉树

从一棵完整的二叉树中删掉标号为 2^h-i，$1\le i\le k$ 的 k 个顶点

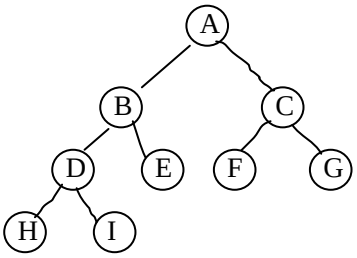


图 2-2-2 一棵完全二叉树

从上到下、从左到右，将完整二叉树上的顶点编号为 $1,2,\ldots,2^{h+1}-1$

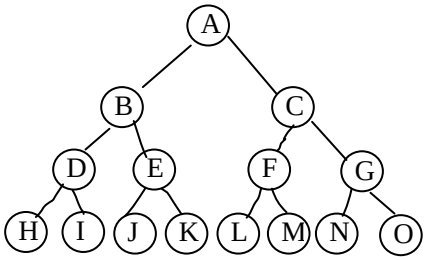


图 2-2-3 一个完整的二叉树

一个二叉树通常用两个数组 Lson 和 Rson 表示。假设二叉树的顶点标为 $1,2,\cdots,n$，则 Lson[i]=j 表示顶点 j 是顶点 i 的左儿子；同样，Rson[i]=j 表示顶点 j 是顶点 i 的右儿子。一个完全的二叉树可以用一个数组来表示：设 T 是一棵高度为 k 的完全二叉树，则数组的第一个元素是该树的根，第二个元素是根的左儿子，第三个元素是根的右儿子。一般地，第 i 个元素的左儿子是数组的第 $2i$ 个元素，第 i 个元素的右儿子是数组的第 $2i+1$ 个元素。反之，数组中第 $j(>1)$ 个元素的父亲是数组中的第 $\lfloor j/2 \rfloor$ 个元素。

一棵有向树是满足下述条件的无圈有向图：

1．有一个称为根的顶点，它不是任何有向边的终点；

2．除了根以外，其余每个顶点的入度均为 1；

3．从根到每个顶点有一条有向路。

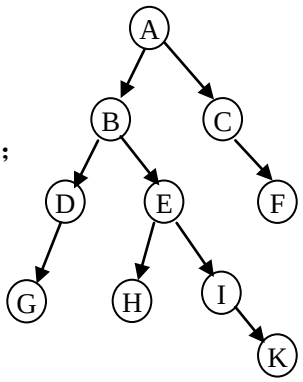


图 2-2-4 有向树

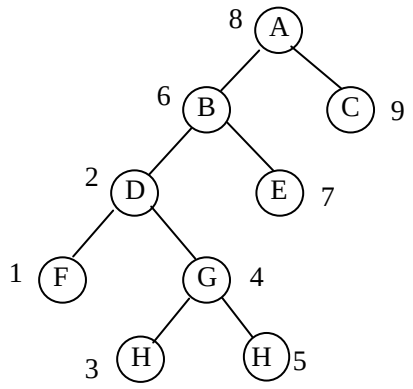
图 2-2-4 是一棵高度为 4 的有向树。对于有向树有类似于有根树的全部术语。

- **二叉树的搜索**

二叉树的搜索主要有先根次序搜索、中根次序搜索和后根次序搜索，各种遍历算法的伪代码写出如下，并在其后给出了搜索过程示意图。其中，Lson(T)表示以树 T 的左儿子为根的子树；Rson(T)表示以树 T 的右儿子为根的子树。

程序 2-2-1　二叉树的中根次序遍历算法伪代码

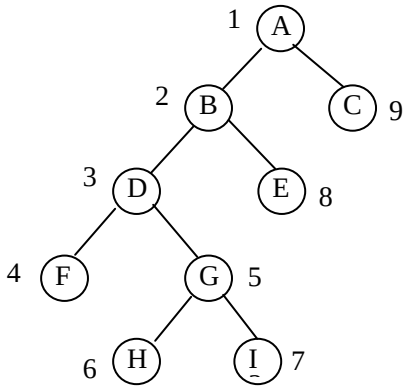
```
InOder(T) // T 是一棵二叉树，T 的每个顶点有三个信息段：
          //Lson,Data,Rson
   if T≠0 then
      InOrder(Lson(T));
      Visit(T);
      InOrder(Rson(T));
    end{if}
end{InOrder}
```



二叉树的中根次序遍历

程序 2-2-2　二叉树的先根次序遍历算法伪代码

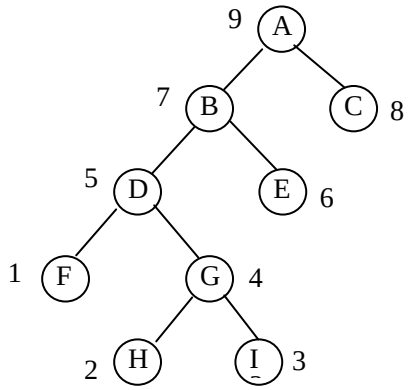
```
PreOder(T) // T 是一棵二叉树，T 的每个顶点有三个信息段：
          //Lson,Data,Rson
   if T≠0 then
      Visit(T);
      PreOrder(Lson(T));
      PreOrder(Rson(T));
    end{if}
end{PreOrder}
```



二叉树的先根次序遍历

程序 2-2-3　二叉树的后根次序遍历算法伪代码

```
PostOder(T) // T 是一棵二叉树，T 的每个顶点有三个信息段：
            //Lson,Data,Rson
      if T≠0 then
         PostOrder(Lson(T));
         PostOrder(Rson(T));
         Visit(T);
       end{if}
  end{PostOrder}
```



二叉树的后根次序遍历

- **一般树的遍历算法**

我们可以将二叉树的父与子之间的关系推广到一般树上，这样每个父亲的儿子可以有许多个，而且可以排出顺序。于是，关于二叉树的后两种遍历算法完全可以移置到一般的树上来，而中根优先次序遍历算法对于多根的情况不好确定根居于哪个位置，所以，不宜照搬。

树的先根次序遍历算法：

i. 若 T 为空，则返回；

ii. 访问 T 的根；

iii. 按树的先根次序遍历 T 的第一棵子树；

iv. 按树的先根次序依次遍历 T 的其余子树。

树的后根次序遍历算法：

v. 若 T 为空，则返回；

vi. 按树的后根次序遍历 T 的第一棵子树；

vii. 按树的后根次序依次遍历 T 的其余子树；

viii. 访问 T 的根。

- **一般图的遍历**

无论是二叉树还是一般的树，由于其不含有圈，所以属于同根的各个子树之间是相互独立的，遍历过程是对各个子树的分别遍历和对根遍历以及把这些遍历有机地组合起来。无论是什么顺序搜索，都不会出现重叠和死循环现象。一般的图没有这种独立性，所以上述方法不能施行。但是，上述方法的思想可以借鉴，于是产生了深度优先搜索方法和宽度优先搜索方法。

问题：*在一个给定的图 G=(V,E)中是否存在一条起于顶点 v 而终于顶点 u 的路径？遍历与某一起点 v 有路相通的所有顶点。*

- **宽度优先搜索算法（BFS）**

开始：起点 v 和一个空的待访队列 Q。

将 v 标记为已访问的顶点，然后开始检查其所有邻点，把其中未被访问的顶点依次放在待检查队列 Q 的尾部。用队列 Q 的首元素 u 替换 v（并从队列 Q 中去掉首元素 u），重复以上过程，直到队列 Q 空为止。

程序 2-2-4 由一点出发的宽度优先搜索算法伪代码

```
proc BFS(v) //宽度优先搜索 G，从顶点 v 开始执行，数组 visited 标示各
    //顶点被访问的序数；数组 s 将用来标示各顶点是否曾被放进待检查队
    //列，是则过标记为 1，否则标记为 0；计数器 count 计数到目前为止已
    //经被访问的顶点个数，其初始化为在 v 之前已经被访问的顶点个数
 1. AddQ（v，Q）；//首先访问 v,将 Q 初始化为只含有一个元素 v 的队列
 2. while Q 非空 do
 3.    u:=DelHead(Q); count:=count+1; visited[u]:=count;
 4.      for 邻接于 u 的所有顶点 w  do
 5.          if s[w]=0 then
```

```
6.              AddQ(w,Q); //将 w 放于队列 Q 之尾
7.              s[w]:=1;
8.            end{if}
9.         end{for}
10.      end{while}
11.  end{BFS}
```

这里调用了两个函数：AddQ(w,Q)是将 w 放于队列 Q 之尾；DelHead(Q)是从队列 Q 取第一个元素，并将其从 Q 中删除。

定理 2.2.1 图 G 的宽度优先搜索算法能够访问 G 中由 v 可能到达的所有顶点。如果记 $t(\nu,\varepsilon)$和 $s(\nu,\varepsilon)$为算法 BFS 在任意一个具有ν个顶点和ε条边的连通图 G 上所花的最大时间和最大空间。则当 G 由邻接矩阵表示时有：

$$t(\nu,\varepsilon)=\Theta(\nu^2),\quad s(\nu,\varepsilon)=\Theta(\nu)$$

当 G 由邻接链表表示时；

$$t(\nu,\varepsilon)=\Theta(\nu+\varepsilon),\ s(\nu,\varepsilon)=\Theta(\nu);$$

证明：除结点 v 外，只有当结点 w 满足 s[w]=0 时才被加到队列上，因此每个结点至多有一次被放到队列上。需要的队列空间至多是 ν-1；visited 数组变量所需要的空间为 ν；其余变量所用的空间为$O(1)$，所以 $s(\nu,\varepsilon)=\Theta(\nu)$。

如果使用邻接链表，语句 4 的 for 循环要做 d(u)次，而语句 2～11 的 while 循环需要做 ν 次，因而整个循环做$\sum_{u\in V}d(u)=2\varepsilon$次$O(1)$操作，又 visited、s 和 count 的赋值都需要 ν 次操作，因而 $t(\nu,\varepsilon)=\Theta(\nu+\varepsilon)$。

如果采用邻接矩阵，则语句 2～11 的 while 循环总共需要做 ν^2次$O(1)$操作，visited、s 和 count 的赋值都需要 ν 次操作，因而 $t(\nu,\varepsilon)=\Theta(\nu^2)$。 证毕

由定理 2.2.1 可知，宽度优先搜索算法能够遍历图 G 的包含 v 的连通分支中的所有顶点。对于不连通的图，可以通过在每个连通分支中选出一个顶点作为起点，应用宽度搜索算法于每个连通分支，即可遍历该图的所有顶点。

程序 2-2-5 图的宽度优先遍历算法伪代码

```
proc BFT(G, ν) //count、s 同 BFS 中的说明，branch 是统计图 G 的
  //连通分支数
  count:=0; branch:=0;
  for i to n do
    s[i]:=0;  //将所有的顶点标记为未被访问
  end{for}
```

```
  for i to v do
    if s[i]=0 then
       BFS(i); branch:=branch+1;
    end{if}
  end{for}
end{BFT}
```

关于 BFT 算法的时间和空间复杂性与 BFS 同样估计（注意空间的差别）。

如果 G 是连通图，则 G 有生成树。注意到 BFS 算法中，由 4～8 行，将所有邻接于顶点 u 但未被访问的顶点 w 添加到待检测队列中。如果在添加 w 的同时将边(u,w)收集起来，那么算法结束时，所有这些边将形成图 G 的一棵生成树。称为图 G 的宽度优先生成树。为此，在 BFS 算法的第 1 行增加语句 T:={}，在第 7 行增加语句 T:=T∪{(u,w)}即可。

- **图的深度优先搜索**

深度优先搜索是沿着顶点的邻点一直搜索下去，直到当前被搜索的顶点不再有未被访问的邻点为止，此时，从当前被搜索的顶点原路返回到在它之前被访问的顶点，并以此顶点作为当前被搜索顶点。继续这样的过程，直至不能执行为止。

程序 2-2-6 图的深度优先搜索算法伪代码

```
proc DFS(v) //访问由 v 到达的所有顶点,计数器 count 已经初始化为 1;
    //数组 visited 标示各顶点被访问的序数，其元素已经初始化为 0。
1.  visited(v):=count;
2.  for 邻接于 v 的每个顶点 w  do
3.    if visited(w)=0 then
4.       count:=cout+1;
5.       DFS(w);
6.    end{if}
7.  end{for}
8. end{DFS}
```

对于连通图，深度优先算法也能产生一棵生成树，称为深度优先搜索树。读者可以在算法中添加语句使算法同时获得深度优先搜索树。

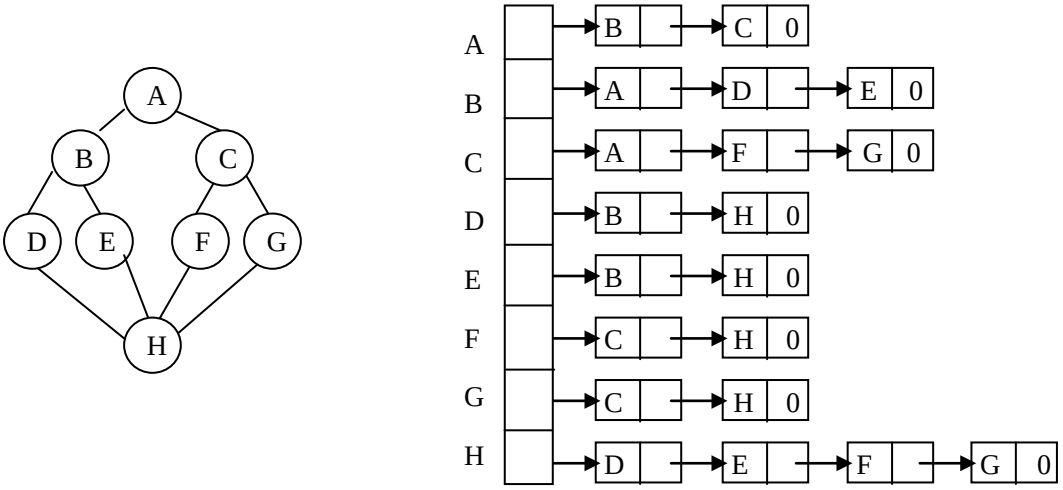


图 G 及其邻接链表

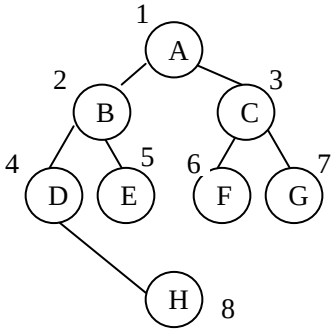


图 G 的宽度优先搜索树

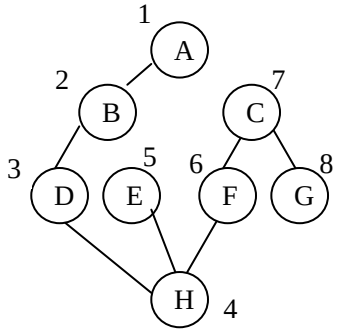


图 G 的深度优先搜索树

比较宽度优先和深度优先搜索算法，发现它们有很大不同。在 BFS 中，当搜索到某个顶点时,就要同时依次将该顶点的所有相邻顶点访问完;在 DFS 中，当搜索到某个顶点时，继续纵深搜索与该顶点相邻的其它一个未搜索过的顶点；前者用一个队列实现，而后者可以用一个栈来实现。当然，前者也可以用一个栈来实现，称为 D-搜索。然 D-搜索已经不同于 BFS 了。

§3 双连通与网络可靠性

通信网络的抽象模型可以是一个无向图:顶点代表通信站,边代表通信线路。图 2-3-1 和图 2-3-2 是两个通信网络示意图。直观可知，图 2-3-1 的通信网络可靠性较高,因为即使有一个网站出现问题,其它网站之间的通信仍能够继续进行。而图 2-3-2 所示的网络则不能，只要网站 F 发生故障，位于其左右两部分的网站之间就无法连通了。在图 2-3-2 中，象 F 这样的顶点称为割点。

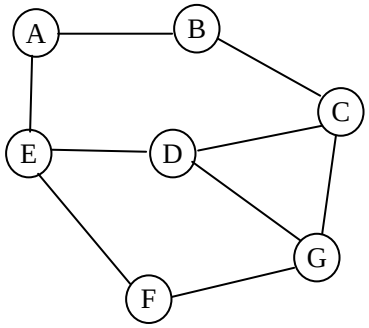


图 2-3-1

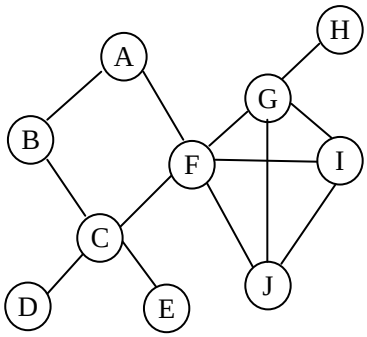


图 2-3-2

定义 连通无向图 G 中的顶点 v 称为割点，如果在 G 中去掉 v 及其关联的边，剩下的图就不再连通。

没有割点的连通图称为 2-连通的（也称为块）。图 G 中极大的 2-连通子图称为 G 的一个 2-连通分支。在图 2-3-2 中除了 F 以外，C 和 G 也都是割点。这个图有 5 个 2-连通分支（参见图 2-3-3）。图 2-3-1 是 2-连通的。

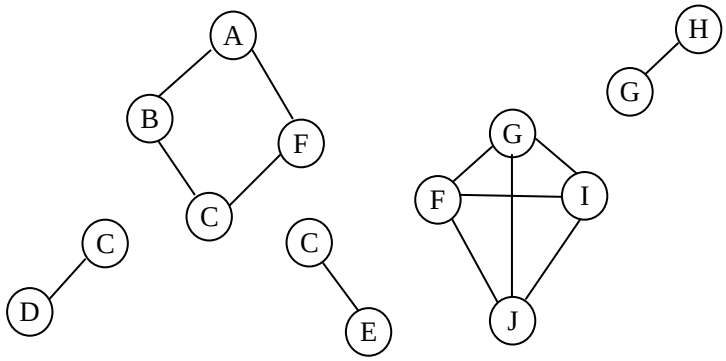


图 2-3-3 图 2-3-2 的 5 个 2-连通分支

就通信网络而言，当然希望没有割点。如果现有的网络有割点，则设法增加一些线路（当然希望尽量少的增加），使之成为 2-连通的。

添加边的算法：

E1： for 每个割点 u do

E2： 设 B_1，B_2，…，B_k，是包含割点 u 的全部 2-连通分支

E3： 设 V_i 是 B_i 的一个顶点，且 $V_i \neq u$，$1 \leqslant i \leqslant k$。

E4： 将边(V_i, V_{i+1})添加到 G，$1 \leqslant i \leqslant k-1$。

E5：endfor

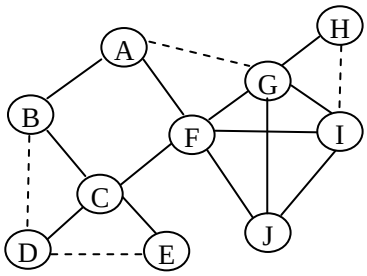


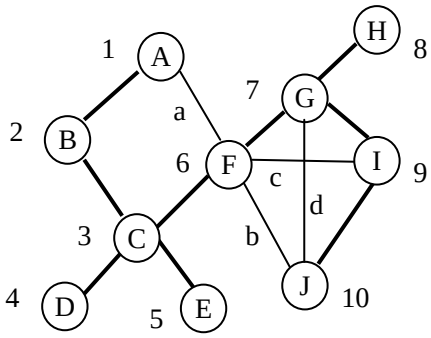
图 2-3-4 添加边使成为二连通图

问题：设计算法测试一个连通图是否 2-连通；若不是，算法将识别出割点。

解决方法：以深度优先遍历算法为基础，加以割点识别步骤。

当采用深度优先遍历算法时，顶点 v 被访问的序数称为 v 的深索数，记做 DFN(v).

假定图的邻接链表是按照顺时针方向的顺序构造的



a,b,c,d 是图 G 关于这棵生成树的余**边**

图 2-3-5 图 G 及其深度优先生成树

按照深度优先生成树将图中的顶点分层，使得上层是下层的祖先，而同层之间是兄弟关系：

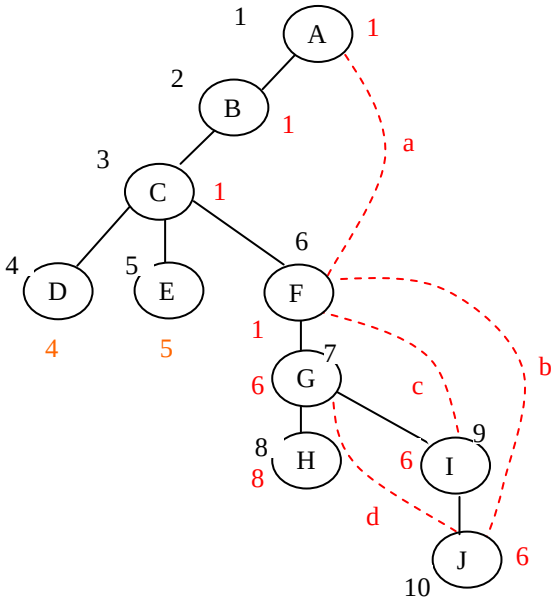


图 2-3-6 深度优先树的分层

1．关于深度优先生成树 T，图 G 的每一条边(u,v)的两个端点 u、v 之间，

或 u 是 v 的祖先，或 v 是 u 的祖先，即不是平辈关系。

2．树 T 的根是图 G 的割点当且仅当其在 T 中至少有两儿子；

3．既不是根也不是叶的顶点 u 不是 G 的割点当且仅当 u 在 T 中的每个儿子 w 都至少有一个子孙（或 w 本身）关联着一条边 e，e 的另一个端点是 u 的某个祖先（e 一定是树 T 的余边）；

4．叶顶点不能是割点。

根据性质 3，4，深度优先生成树 T 的非根顶点 u 是 G 的割点当且仅当 u 至少有一个儿子 w，w 及其子孙都不与 u 的任何祖先相邻。

注意到 u 的深索数一定小于其子孙的深索数，所以深索数 DFN 并不能反映一个顶点是否是割点的情况。为此，我们递归地定义各个顶点 u 的最低深索数 L(u)：

定义 顶点 u 的最低深索数 L(u)定义为

$$L(u):=\min\{DFN(u),\ \min\{DFN(x)\mid(u,x)\text{是 T 的余边}\},\ \min\{L(w)\mid w\text{是 u 的儿子}\}\ \}$$

可见，如果 $L(u)\neq DFN(u)$，则必定 $L(u)<DFN(u)$，因而 L(u)是 u 通过一条子孙路径(至多后跟一条 T 的余边)所可能到达的顶点的最低深索数。图 2-3-6 中，红色数字表示了各顶点的最低深索数。

结论：如果 u 不是深度优先生成树的根，则 u 是图 G 的割点当且仅当 u 有某个儿子 w，w 的最低深索数不小于 u 的深索数，即

$$L(w)\geqslant DFN(u),\ \text{for some son w of u}$$

看来要想识别图 G 的所有割点，需先获得深度优先树 T，然后按后根优先次序遍历 T，计算出各个顶点的最低深索数。但是，从函数 L 的定义可知这两步工作可以同时完成。

程序 2-3-1 计算 DFN 和 L 的算法伪代码

```
proc DFNL(u,v) //一个深度优先搜索算法，u 是开始顶点。在深度优先
      //树中，若 u 有父亲，则 v 即是。数组 DFN 初始化为 0，变量
      //num 初始化为 1，n 是图 G 的顶点数。
      global  DFN[n], L[n], num, n
1.      DFN(u):=num; L(u):=num; num:=num+1;
2.      for 每个邻接于 u 的顶点 w  do
3.        if DFN(w)=0 then
4.            DFNL(w,u); //还未访问 w
5.            L(u):=min(L(u),L(w));
6.        else
7.          if w≠v then L(u):=min(L(u),DFN(w)); end{if}
```

```
8.        end{if}
9.      end{for}
10. end{DFNL}
```

为了得到图 G 的 2-连通分支，需对 DFNL 作一些修改。注意到，在第 4 行调用 DFNL 时，如果出现情况：

$$L(w) \geqslant DFN(u),$$

就可以断定 u 或是 T 的根，或是图 G 的割点。不论那种情况，都将边(u,w)以及对 DFNL 这次调用期间遇到的所有树边和余边加在一起（除了包含在子树 w 中其它 2-连通分支中的边以外），就构成图 G 的一个 2-连通分支。鉴于此，将 DFNL 做如下修改：

```
// 引进一个存放边的全程栈 S；
// 在 2 到 3 行之间加：
    2.1  if v≠w and DFN(w)<DFN(u) then
    2.2     将(u,w)加到 S 的顶部；
    2.3   end{if}
// 在 4 到 5 行之间加下列语句：
    4.1  if L(w)≥DFN(u) then
    4.2    print（’new biconnected component’）;
    4.3    loop
    4.4      从栈 S 的顶部删去一条边；
    4.5      设这条边是(x,y)
    4.6      print(“(”, x,“,”, y,“)”);
    4.7    until ((x,y)=(u,w) or(x,y)=(w,u));
    4.8  end{if}
```

如果 G 是有 n 个顶点 m 条边的连通图，且 G 用邻接链表表示，那么 DFN 的计算时间是 O(n+m).一旦算出 L[1:n],G 的割点就能在 O(n)时间识别出来，因此，识别全部割点总的时间不超过 O(n+m).

当 DFNL 增加了上述语句之后，其计算时间仍然是 O(n+m).

定理 2.3.1 设 G 是至少有两个顶点的连通图，则算法 DFNL 增加了语句 2.1-2.3 和 4.1-4.8 的算法能正确生成 G 的全部 2-连通分支。

证明：略。

§4 对 策 树

在一盘棋中，对弈各方都要根据当前的局势，分析和预见以后可能出现的局面，决定自己要采取的各种对策，以争取最好的结果。本节我们将用树的模型来描述对弈局势，并给出产生对策树的算法。先来分析拾火柴棍游戏。

在盘面上放 n 支火柴，由弈者 A 和 B 两人参加比赛。规则：两人轮流从盘上取走 1，2 或 3 支火柴，拿走盘中最后一支火柴者为负。

以盘中剩下的火柴数来表示该时刻的棋局。棋局序列 C_1，C_2，…，C_k 称为有效（棋局）序列，如果：

- C_1 是开始棋局；
- C_i 不是终止棋局，i＝1，2，…，k-1；
- 由 C_i 走到 C_{i+1} 按下述规则：若 i 是奇数，则 A 走一合法步骤；若 i 是偶数，则 B 走一合法步骤。

以 C_k 为终局的一个有效棋局序列 C_1，C_2，…，C_k 是此游戏的一盘战例。有限次博弈游戏的所有可能的实际战例可以用一棵树来表示（参见附页“对策树 1”）。

在圈 B 标志的地方表示 B 取胜；在方块 A 标志的地方表示 A 取胜。从图中可以看出，只要 A 第一步取 1 根火柴，则不论 B 如何应着，A 都有取胜的应着。否则，A 不保证取胜。那么怎样确定 A 的棋着呢？是否有一般的规律？对策树在决定采取什么对策，即确定弈者下一步应走哪步棋上是很有用的。事实上，走哪一步使自己获胜的机会最大，可以用一个估价函数 E(X)来评价，它是棋局 X 对于弈者价值大小的估计。设 E(X)是弈者 A 的估价函数，若棋局 X 能使 A 有较大的获胜机会，则 E(X)的值就高，若棋局 X 使得 A 有较大的失败可能，则 E(X)的值就低。

那些使得 A 获胜的终止棋局或不管 B 如何应着都保证 A 能获胜的棋局，则 E(X)取最大值。而对于保证 B 取胜的棋局，E(X)则取最小值

对于其对策树顶点较少的博弈游戏，例如 n=6 的拾火柴棍游戏，可以采用先对终局定义 E(X)，而后逐步确定各个棋局 X 的价值 V(X)的方法给出 A 在各步走棋参考。在 n＝6 的拾火柴棍游戏的对策树中，终止顶点是叶顶点，我们给出如下估值：

$$E(X)=\begin{cases}1 & \text{若X对于A是胜局}\\ -1 & \text{若X对于A是负局}\end{cases} \qquad (2.4.1)$$

而对于其它顶点，需要给出对于 A 来说能够取胜的价值。例如，若已经知道顶点 b，c，d 的价值，则父顶点 a 的价值应当取它们的价值的最大值，当顶点 a 代表的棋局处应该 A 走棋时。因为从棋局 a 出发，A 下一着棋的走法应当导致其得胜可能性最大的下一步棋局。一般地，若 X 不是叶点，其有儿子 X_1，X_2，…，X_d，

则定义 X 的价值为

$$V(X)=\begin{cases}\max\limits_{1\le i\le d}\{V(X_i)\} & \text{若X是方形顶点}\\ \min\limits_{1\le i\le d}\{V(X_i)\} & \text{若X是圆形顶点}\end{cases} \qquad (2.4.2)$$

如此计算出对策树 1 各顶点的价值后，就很容易看出 A 要（想取胜）在其所处的各个棋局上应该采取的对策了。从图中可以看出 A 有三条必胜的路线。用(2.4.2)确定各个顶点价值的过程称为最大最小过程。

实际上，对策树 1 恰好列出了 n = 6 时，拾火柴棍游戏所有可能的棋局。从对策树根顶点出发到达叶顶点的每条路径恰是一个战例。但是对于较大规模的博弈，一般很难列出所有可能的棋局序列。比如，国际象棋，它的完整的对策的顶点数，据估计，将达到 10^{100}，即使用一台每秒能生成 10^{11} 个顶点，也需要 10^{80} 年以上的时间才能生成完整的对策树。所以，对于具有大规模对策树的博弈，不是采取考察其完整对策树的办法来确定弈者的对策。这种情况下，通常采用向前预测几步才决定走一步的策略，实际上是假想一组局面。这组局面也可以用部分对策树表示出来（参看文档“对策树 2”）。用估价函数估定这样的对策树（实际是子树）的叶顶点的值，然后根据公式(2.4.2)逐一确定其它顶点的价值。最后确定下一步该走的棋着。使用产生数级对策树确定下一步棋着的方法所导致的棋局的质量将取决于这两名弈者所采用的估价函数的功能和通过最大最小过程来确定当前棋局的价值 V(X)所使用算法的好坏。

因为 A、B 两弈者走棋总是交替进行的，在写递归算法时经常要区分 A、B，以确定是取最大值还是取最小值。为克服这个缺点，只要改变 B 者值的符号就可以。不妨假定弈者 A 是一台计算机， A 为了确定下一着棋，其应该有一个算法计算所有 V(X)。为产生一个递归程序，将公式(2.4.2)中的取最小部分也改成取最大，V(X)改为 V′(X)：

$$V'(X)=\begin{cases}e(X) & \text{若X是所生成子树的叶顶点}\\ \max\limits_{1\le i\le d}\{-V'(X_i)\} & \text{若X不是所生成子树的叶顶点}\end{cases} \qquad (2.4.3)$$

其中，X_1，X_2,---，X_d 是 X 的所有儿子顶点。当 X 是叶顶点时，若 X 是 A 走棋的位置，则 e(X)=E(X);若是 B 走棋的位置，则 e(X)=-E(X)。通过对以 X 为根顶点、高为 l 的对策树的后根次序遍历，可以产生求取 V′(X)的递归算法。

程序 2-4-1 对策树的后根次序求值算法

```
VE(X, l)//通过至多向前看l着棋计算V′(X)，弈者A的估价
  //函数是e(X)。假定由任一不是终局的棋局X开始，此棋局的
  //合法棋着只允许将棋局X转换成棋局X1，X2，…，Xd.
  if X是终局或l=0 then return(e(X)) end{if}
```

```
        ans:= -VE(X1, l-1); //遍历第一棵子树
        for i from 2 to d do
          ans:=max(ans,-VE(Xi, l-1));
        end{for}
    return(ans);
end{VE}
```

只要将偶数级顶点的价值改变一下符号，文档“对策树 2”中给出的各棋局的价值即是调用算法 VE 的计算结果，此时，X=P_{11}，l=4.各棋局的值按如下次序依次确定：

P_{31}，P_{32}，P_{21}，P_{51}，P_{52}，P_{53}，P_{41}，P_{54}，P_{55}，P_{56}，P_{42}，P_{33}，…，P_{37}，P_{24}，P_{11}.

算法 VE 的目标是求取在棋局 P_{11} 情况下对于弈者 A 的 $V(P_{11})$值，以决定 A 下一步要采取的决策。但是，从这个例子的计算过程来看，为达此目标不必计算上述所有棋局的估价值。比如，当知道 $V(P_{41})$=3 后，就知道 $V(P_{33})\geqslant 3$。因为 P_{33} 是求最大值点，所以，一旦知道 $V(P_{42})\leqslant 3$，就没有必要继续求以 P_{42} 为根子树其它顶点的估价值了(因为 P_{42} 是求最小值点，只要 P_{42} 有一个儿子结点的估价值不超过 3，就有 $V(P_{42})\leqslant 3$，所以，知道 $V(P_{42})\leqslant 3$ 可能是已经计算出 P_{42} 的某些儿子结点处的估价值)。如果给求最大值点 Y 赋一个值α(Y)，代表到目前为止所知道的该结点最大估值，那么，在求其儿子结点 X 的估价值时就有了参考，可以省掉不必要的计算。同样地，如果给求最小值点 Y 赋一个值 $\beta(Y)$，代表到目前为止所知道该结点最小估值，那么，在求其儿子结点 X 的估价值时就有了参考，可以省掉不必要的计算。比如，一旦判断出 $V(P_{36})\geqslant -1$，则不必计算以 P_{36} 的其它儿子结点为根的子树的顶点的价值。设 Y 是 X 的父亲结点，X 有儿子结点 X_1，X_2，…，X_d，上述减少计算的思路可以形成如下的规则：

- ✧ 若 Y 是取最大值的顶点，则一旦知道 $V(X_k)\leq\alpha(Y)$ 就不必再计算以 X_{k+1}，…，X_d 为根的子树的任何顶点的估价值；
- ✧ 若 Y 是取最小值的顶点，则一旦知道 $V(X_k)\geq\beta(Y)$ 就不必再计算以 X_{k+1}，…，X_d 为根的子树的任何顶点的估价值；

称之为 $\alpha-\beta$ 截断规则。采用 $\alpha-\beta$ 截断规则可以改进算法 VE。在算法 VE 中，各结点的估价值是用公式(2.4.3)计算的 V′，所以，每个结点都看作是求最大值点。于是，在定义一个结点的 B 值是该结点迄今最大可能值的情况下，$\alpha-\beta$ 截

断规则是：对于任一结点 X，设 B 是该结点父亲结点的 B 值，而且，D=-B，那么，如果 X 的估价值不小于 D 时，则可以停止生成 X 的其他儿子结点。

程序 2-4-2 使用$\alpha-\beta$截断规则的后根次序求值算法

```
VEB(X, l,D)//通过至多向前看l着棋，使用α－β截断规则和
    // 公式(2.4.3)计算 V′(X)，弈者 A 的估价函数是 e(X)。假定
    // 由任一不是终局的棋局 X 开始，此棋局的合法棋着只允许
    // 将棋局 X 转换成棋局 X1，X2，…，Xd.
    if X 是终局或l=0  then  return(e(X))  end{if}
    ans:=-VEB(X1，l-1，∞)；// V′(X)到目前可能的最大值
    for  i  from  2  to  d  do
      if  ans ≥ D then return（ans）end{if}//使用α－β截断规则
      ans:= max(ans,-VEB(Xi，l-1，-ans))；
    end{for}
  return(ans);
end{VEB}
```

算法 VEB 还可以进一步改进，即造成更大的截断。这只需要观察孙子结点的值达到多大才能影响到爷爷结点的值即可得到线索。仍采用公式(2.4.3)来计算 V′(X)，用$\mu(Y)$表示到迄今所知结点 Y 的最大的可能估价值，并设 X 是 Y 的儿子结点，X 的儿子结点 X_1，X_2，…，X_d ，要想结点 X 的值对 Y 的值产生影响，就必须要求 X 的每个儿子结点 X_i 满足：$V'(X_i)>\mu(Y)$。因为，若某个 $V'(X_i)\leqslant\mu(Y)$，

$$V'(Y)\geq\max\{\mu(Y),-V'(X\ )\},\text{而}V'(X)=\max_{1\leq j\leq d}\{-V'(X_j)\}\geq-V'(X_i)\geq-\mu(Y)$$

从而－V′(X) $\leqslant\mu(Y)$，所以，V′(Y)不受 V′(X)的影响。这样，我们可以给每个结点赋一个下界值 LB，只要判断出该结点的估价值不超过这个下界，就可以停止计算其父亲结点的估价值。这产生了下面的改进算法：

程序 2-4-3　$\alpha-\beta$截断算法

```
AB(X，l，LB，D) // LB 是 V′(X)的一个下界。//通过至多向前看l着棋，
    //使用α－β截断规则和公式(2.4.3) 计算 V′(X)，弈者 A 的估价
```

```
//函数是 e(X)。假定由任一不是终局的棋局 X 开始，此棋局的合法
//棋着只允许将棋局 X 转换成棋局 X1，X2，…，Xd.
if X 是终局或l=0 then return(e(X)) end{if}
ans = LB;
for i from 1 to d do
  if ans ≥ D then return（ans）end{if} //使用α－β截断规则
  ans = max(ans,-AB(Xi，l-1，-D，-ans));
end{for}
return(ans);
end{AB}
```

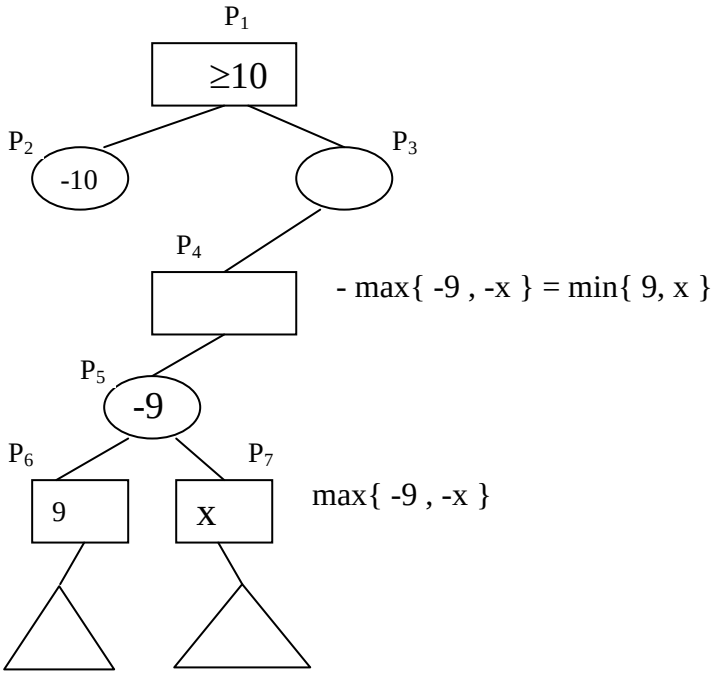


图 2-4-1 一棵假想的对策树

对图 2-4-1 这棵对策树调用算法 VEB(P_1,l,∞)和调用算法 AB(P_1,l,-∞，∞)实际需要计算价值的结点数是不一样的。使用算法 VEB 需要计算 V′(P_7)， 但使用算法 AB 时不必。因为，当我们知道 V′(P_2)=-10 后， 就知道 10 是 V′(P_1)至少是 10，因而知道了 10 是 V′(P_4)的一个下界，进而知道 10 也是 V′(P_6)的下界。但是 V′(P_6)=9<10，所以计算 V′(P_5)的进程停止，即不需计算 V′(P_7).

习题 二

1.证明下列结论：

1）在一个无向图中，如果每个顶点的度大于等于 2，则该图一定含有圈；

2）在一个有向图 D 中，如果每个顶点的出度都大于等于 1，则该图一定含有一个有向圈。

2.设 D 是至少有三个顶点的连通有向图。如果 D 中包含有向的 Euler 环游（即是通过 D 中每条有向边恰好一次的闭迹），则 D 中每一顶点的出度和入度相等。反之，如果 D 中每一顶点的出度与入度都相等，则 D 一定包含有向的 Euler 环游。这两个结论是正确的吗？请说明理由。如果 G 是至少有三个顶点的无向图，则 G 包含 Euler 环游的条件是什么？

3. 设 G 是具有 n 个顶点和 m 条边的无向图，如果 G 是连通的，而且满足 m = n-1，证明 G 是树。

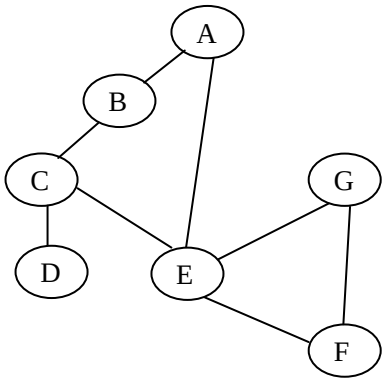
4. 假设用一个 n×n 的数组来描述一个有向图的 n×n 邻接矩阵，完成下面工作：

1） 编写一个函数以确定顶点的出度，函数的复杂性应为$\Theta(n)$；

2） 编写一个函数以确定图中边的数目，函数的复杂性应为$\Theta(n^2)$；

3） 编写一个函数删除边(i, j)，并确定代码的复杂性。

5.下面的无向图以邻接链表存储，而且在关于每个顶点的链表中与该顶点相邻的顶点是按照字母顺序排列的。试以此图为例描述讲义中算法 DFNL 的执行过程。



一个无向图 G

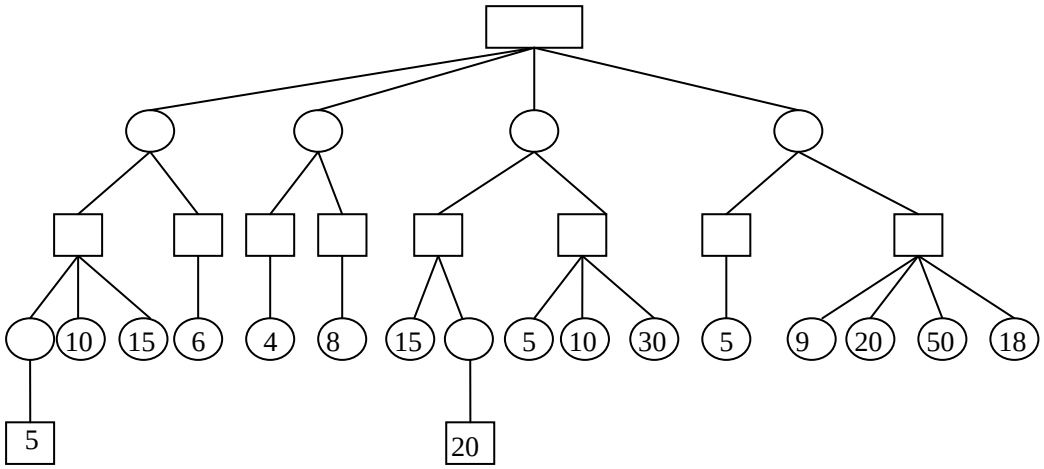
6.对图的另一种检索方法是 D-Search。该方法与 BFS 的不同之处在于将队列换成栈，即下一个要检测的结点是最新加到未检测结点表的那个结点。

1）写一个 D-Search 算法；

2）证明由结点 v 开始的 D-Search 能够访问 v 可到达的所有结点；

3）你的算法的时、空复杂度是什么？

7.考虑下面这棵假想的对策树：



1） 使用最大最小方法(2-4-2)式获取各结点的值；

2） 弈者 A 为获胜应该什么棋着？

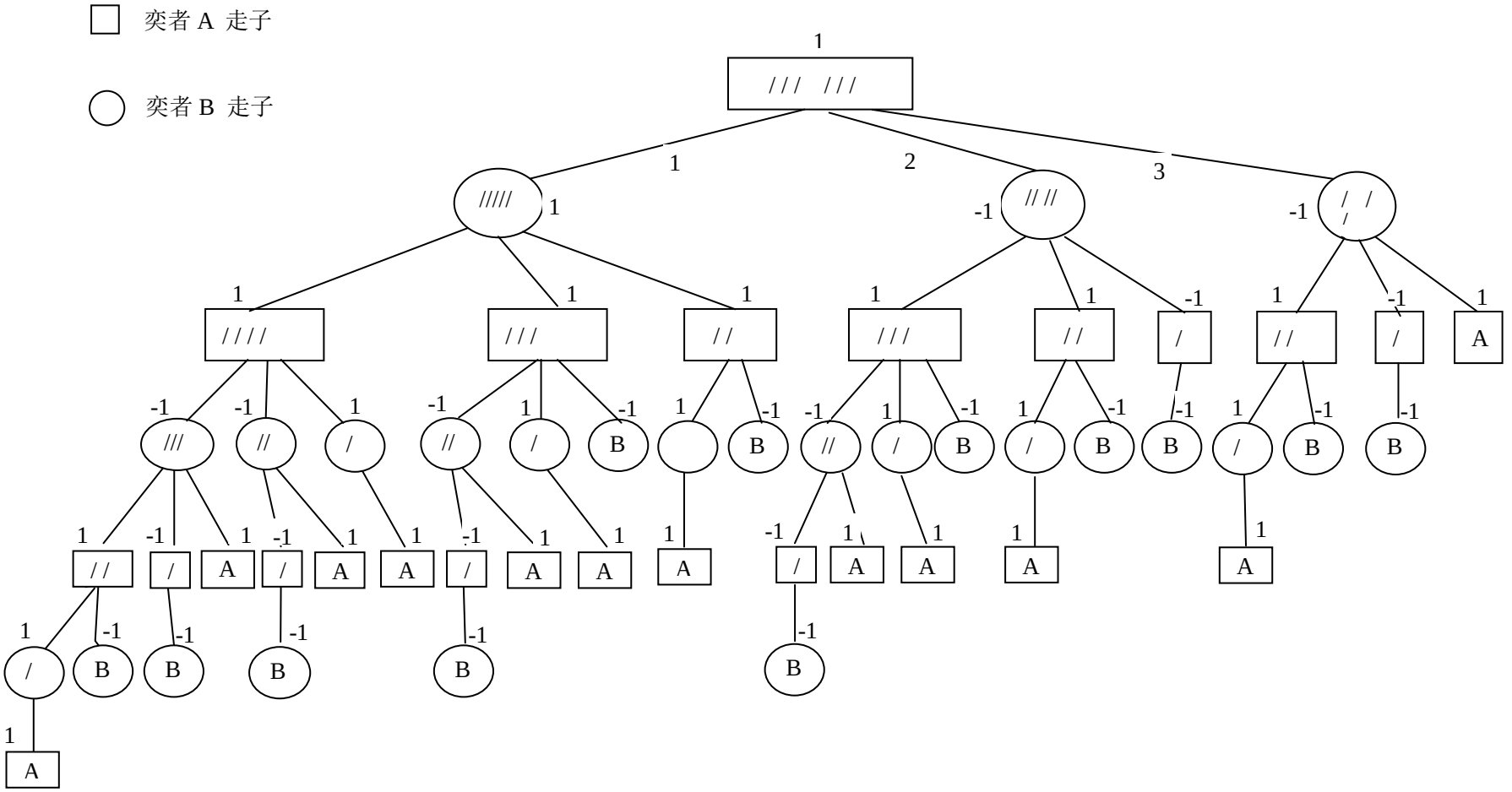
3） 列出算法 VEB 计算这棵对策树结点的值时各结点被计算的顺序；

4） 对树中每个结点 X，用(2-4-3)式计算 V(X)；

5） 在取 X＝根，$l=10, LB=-\infty, D=\infty$ 的情况下，用算法 AB 计算此树的根的值期间，这棵树的哪些结点没有计算？

对策树 1

n=6 时的拾火柴游戏状态树图



对策树 2

一盘假想博弈游戏的部分对策树