第二章　图与遍历算法

基本概念和术语

二叉树与遍历算法

双连通与网络可靠性

对策树

图的基本概念和术语

图是一个用（边）连结节点（顶点）的结构.

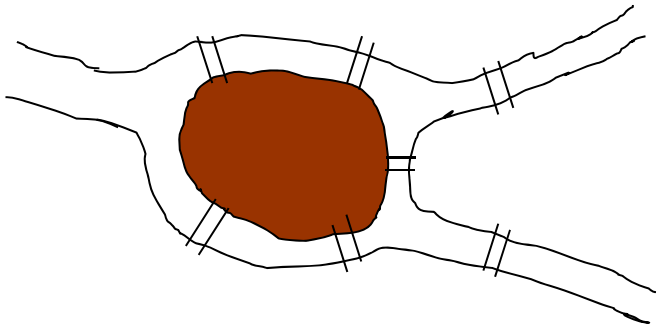
三元素：顶点集、边集、关联关系

$G=(V,E,I)$

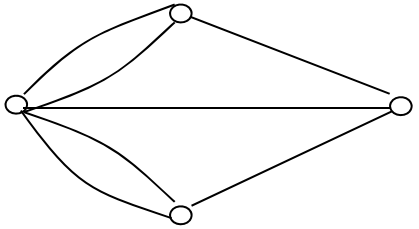
术语：顶点与边关联、边的端点、顶点的度、简单图、完全图、偶图、k-部图、途径、迹、路、圈、连通

Euler公式：

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



哥尼斯堡七桥



Euler图

图的邻接矩阵和关联矩阵

图G=(V,E,I)

$V=\{v_1,v_2,\cdots,v_n\}$ ， $E=\{e_1,e_2,\cdots,e_m\}$

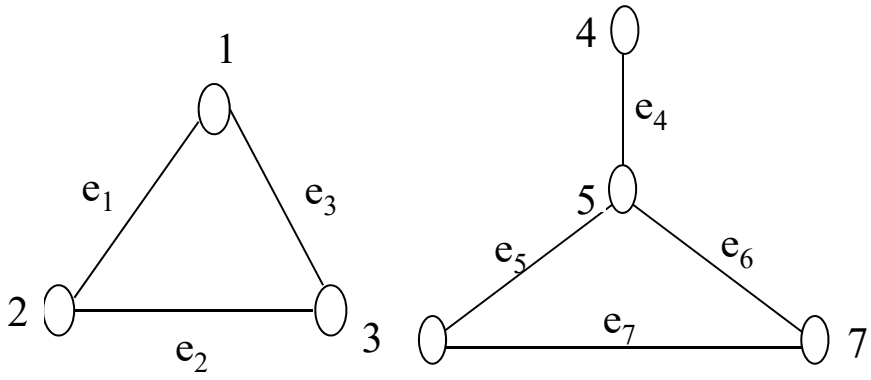
邻接矩阵 $A=(a_{ij})_{n\times n}$

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

关联矩阵 $M=(b_{ij})_{n\times m}$

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

右边的图不是连通的，有两个连通分支。它含有两个圈，不是树。



有7个顶点和7条边的图

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

上图的邻接矩阵与关联矩阵

树的性质

定义 不含圈的连通图称为**树**。对于连通图，适当去掉一些边后，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**。 森林是由若干棵树构成的图。

定理 如果G是具有n个顶点、m条边的图，则下列结论立：

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是连通图，则$m \geq n-1$;
5. 若G是由k棵树构成的森林，则$m = n-k$ ；
6. 若G有k个连通分支，而且满足$m = n-k$，则G是森林;
7. 若G有k个连通分支，则$m \geq n-k$。

有向图

有向图 $G=(V,E,I)$；

有向边 $e=(u,v)$ ；

出(入)度 $d^+(v)$ （$d^-(v)$ ）;

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

有向途径、有向迹、有向路，有向圈，双向连通（强连通）

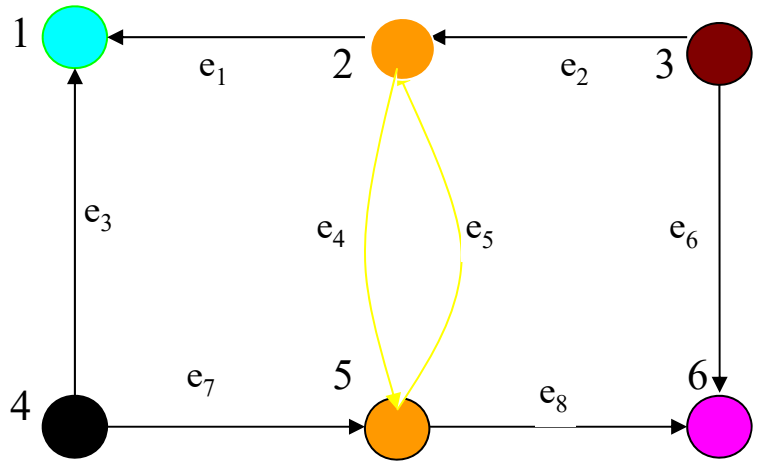
$V=\{v_1,v_2,\cdots,v_n\}$ $E=\{e_1,e_2,\cdots,e_m\}$

邻接矩阵 $A=(a_{ij})_{n\times n}$

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

关联矩阵 $M=(b_{ij})_{n\times m}$

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



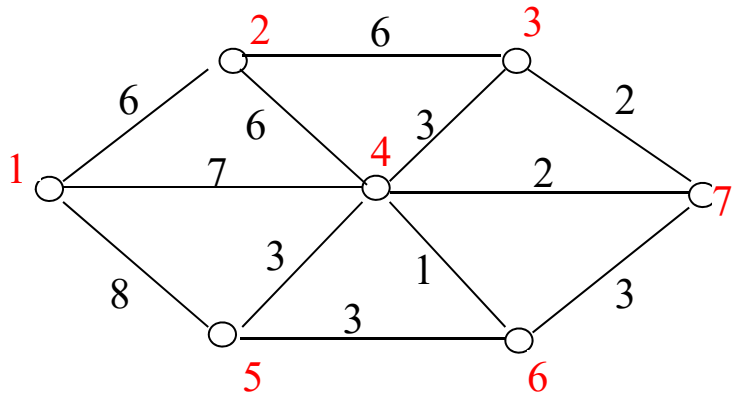
$$M=\begin{pmatrix}-1&0&-1&0&0&0&0&0\\1&-1&0&1&-1&0&0&0\\0&1&0&0&0&1&0&0\\0&0&1&0&0&0&1&0\\0&0&0&-1&1&0&-1&1\\0&0&0&0&0&-1&0&-1\end{pmatrix}$$

赋权图

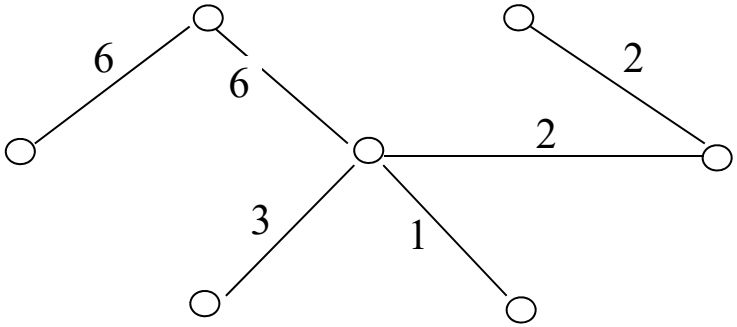
赋权图是将图的每个边都赋予一个权值，表示成本、效益值、容量、流量等。

赋权图的邻接矩阵的 (i, j) - 元素为连结顶点 i，j 的权值。当顶点 i，j 没有边连结时，可根据问题需要，取 0 或 ∞ 等。

$$A = \begin{pmatrix} \infty & 6 & \infty & 7 & 8 & \infty & \infty \\ 6 & \infty & 6 & 6 & \infty & \infty & \infty \\ \infty & 6 & \infty & 3 & \infty & \infty & 2 \\ 7 & 6 & 3 & \infty & 3 & 1 & 2 \\ 8 & \infty & \infty & 3 & \infty & 3 & \infty \\ \infty & \infty & \infty & 1 & 3 & \infty & 3 \\ \infty & \infty & 2 & 2 & \infty & 3 & \infty \end{pmatrix}$$

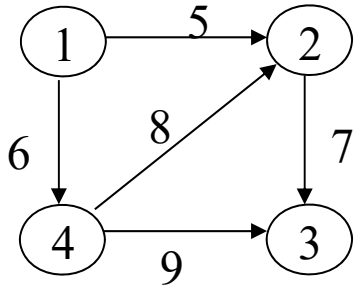


一个赋权图



上图的一个最优生成树

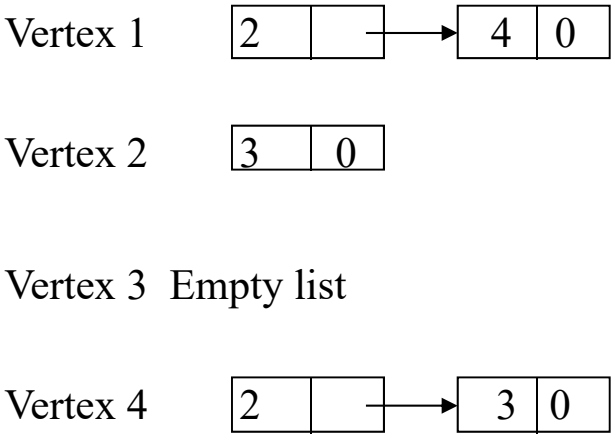
图的邻接链表



(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)



(c)

		HEAD	NEXT
Vertices	1		5
	2		7
	3		0
	4		8
Edges	5	2	6
	6	4	0
	7	3	0
	8	2	9
	9	3	0

(d)

图的遍历算法

有根树

- 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。递归地，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**；顶点的高度是以它为根的子树的高度；顶点的**深度**是从根顶点到它的路的长度。深度相同的顶点称为同层的。
- 有根树中，一个顶点的深度与高度的和不超过整个树的高度

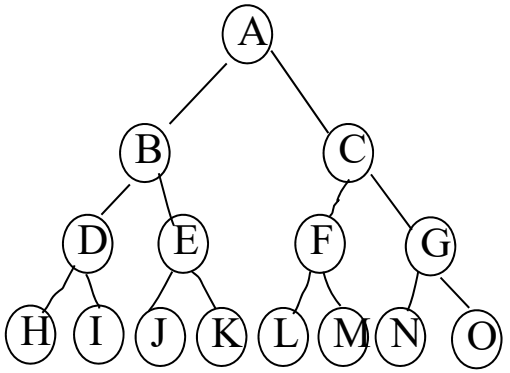
二叉树

二叉树：每个顶点至多有两个儿子的有根树。

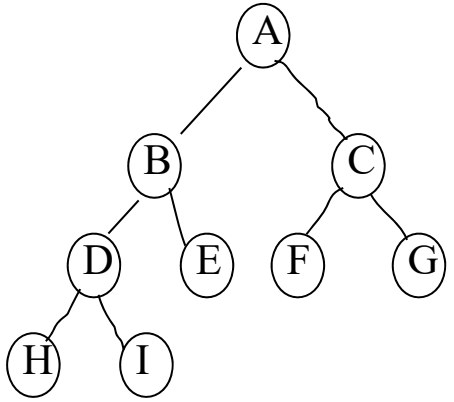
完整二叉树：叶顶点深度都相同，而且除了叶顶点以外，每个顶点都恰有两个儿子的二叉树。

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$1,2,\cdots,2^{k+1}-1$。

完全二叉树：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。



一棵完整的二叉树



从一棵高为h-1的完整二叉树中删掉标号为2^h-i，$1\le i\le k$的k个顶点

一棵全二叉树

有向树

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头；
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。

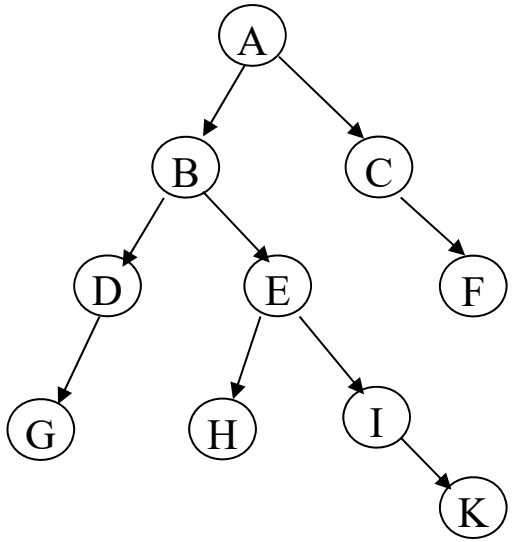
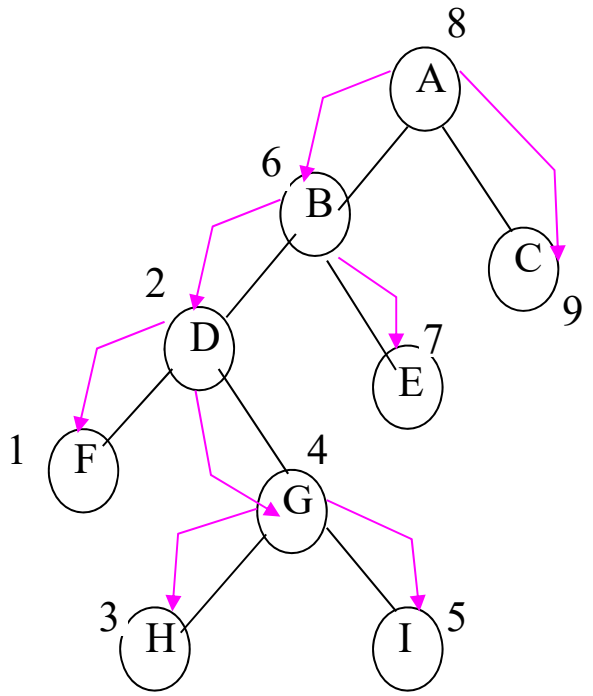


图 2-2-4 有向树

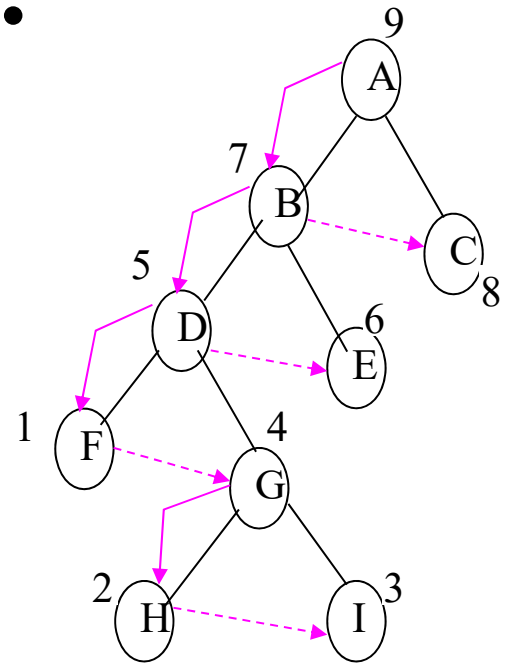
二叉树的搜索：先根、中根、后根

二叉树的中根次序遍历算法

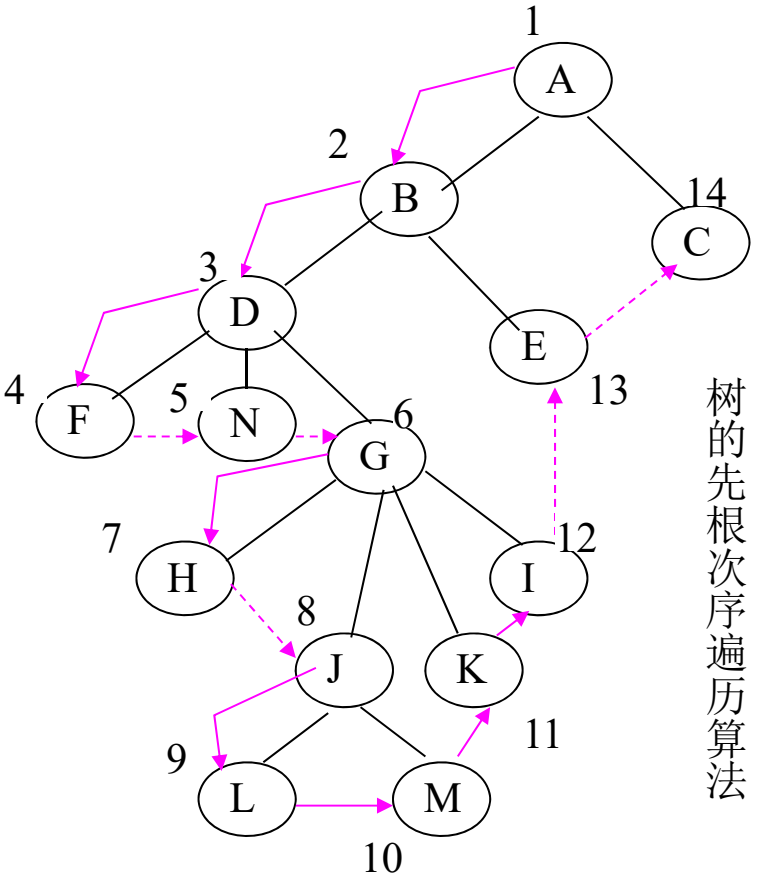
```
InOder(T) // T是一棵二叉树，
    //T的每个顶点有三个信息
    //段：Lson,Data, Rson
  if T≠0 then
    InOrder(Lson(T));
    Visit(T);
    InOrder(Rson(T));
  end{if}
end{InOrder}
```



一般树上的搜索

- 

二叉树的后根次序遍历算法



树的先根次序遍历算法

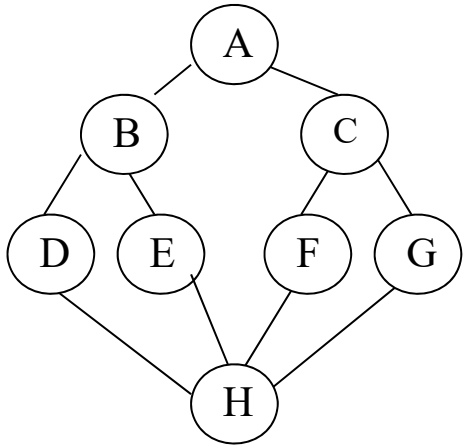
图的宽度优先搜索算法

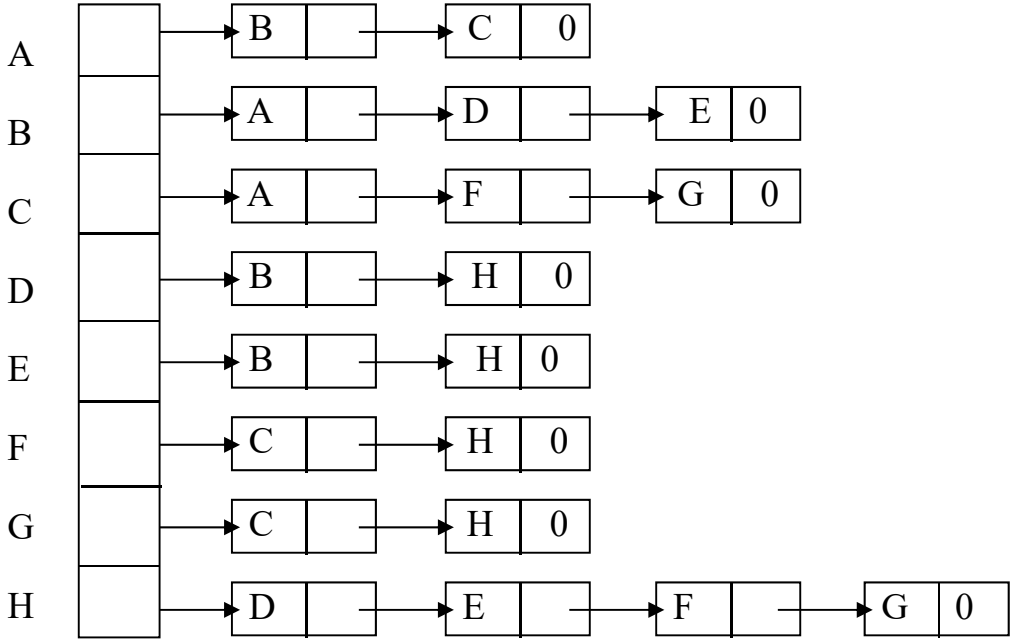
BFS(v) //宽度优先搜索G，从顶点v开始执行，数组Visited标记各顶点被
//访问的次序，其分量已经初始化为0。数组s将用来标示各顶点是否被

- //检索过，是则标记为1，否则标记为0；计数器count已经初始化为0。
- 1. AddQ(v,Q)； S[v]:=1; // 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);
- 4. count:=count+1;
- 5. visited[u]:=count;
- 6. **for** 邻接于u的所有顶点w **do**
- 7. **if** S[w]=0 **then**
- 8. AddQ(w,Q); //将w放于队列Q之尾
- 9. S[w]:=1;
- 10. **end{if}**
- 11. **end{for}**
- 12. **end{while}**
- 13. **end{BFS}**

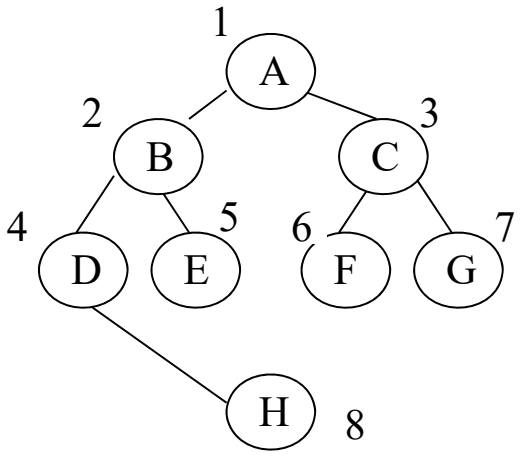
ET:={ };

ET:=ET union { (u,w)};

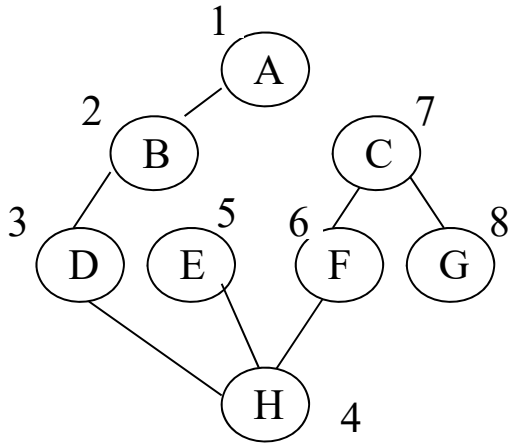




图G和它的邻接链表



图G的宽度优先搜索树



图G的深度优先搜索树

图的深度优先搜索

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次序，其元
//素已经初始化为0。

ET初始化为空集。

- 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:=count+1;
- 5. DFS(w);
- 6. **end{if}**
- 7. **end{for}**
- 8. **end{DFS}**

ET:=ET union { (u,w)};

算法BFS的复杂性分析

空间复杂度为 $S(n,m)=O(n)$ ；

时间复杂度，当G用邻接矩阵表示时：$T(m,n)=O(n^2)$；

当G由邻接链表表示时，$T(n,m)=O(n+m)$。

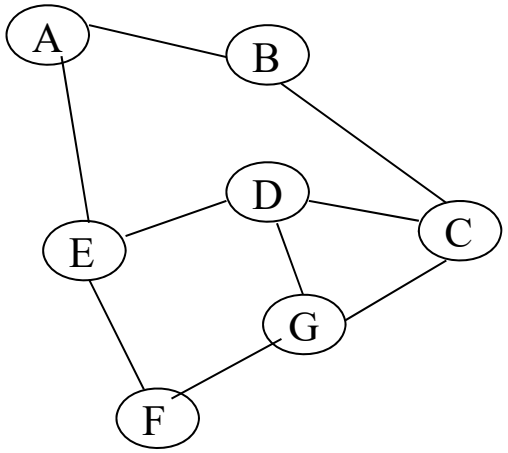
除结点v外，只有当结点w满足Visited(w)=0时才被加到队列上，然后，Visited(w)的值马上被修改增加1.因此每个结点有一次机会被放到队列上。需要的队列空间至多是n-1，其余变量所用的空间为O(1)，S(n,m)=O(n)。 在极端情况下，v邻接于全部其他n-1个结点，此时队列需要n-1的空间。又Visited需要Θ(n)的空间，所以S(n,m)=Θ(n)。

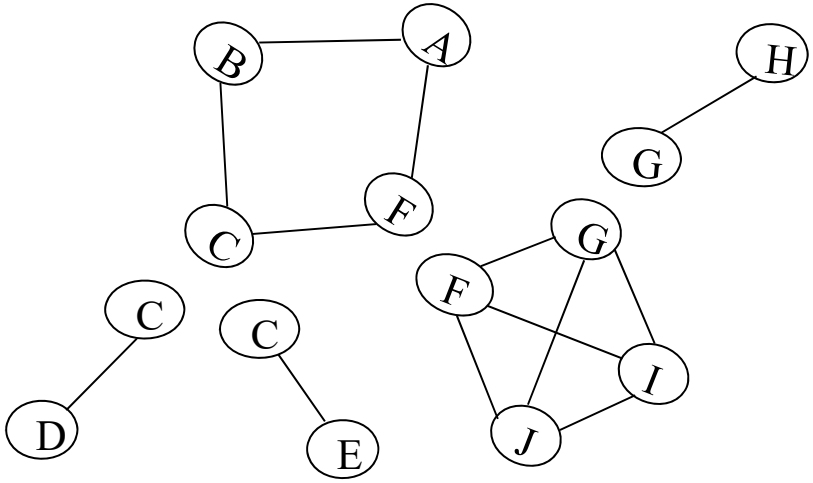
如果使用邻接链表，语句4的for循环要做d(u)次，而语句3～11的while循环需要做n次，因而整个循环做∑d(u)=2m次O(1)操作，又Visited的赋值需要n次操作，因而T(n,m)=Θ(n+m)。

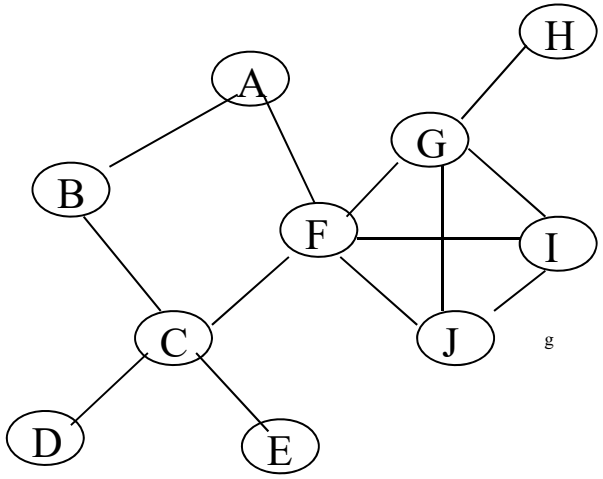
如果采用邻接矩阵，则语句3～12的while循环总共需要做n^2次O(1)操作，Visited赋值需要n操作，因而T(n,m)=Θ(n^2)。

2-连通与网络可靠性

定义 连通图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。







右下图的5个2-连通分支

2-连通化

- 添加边的算法

E1： **for**每个割点u **do**

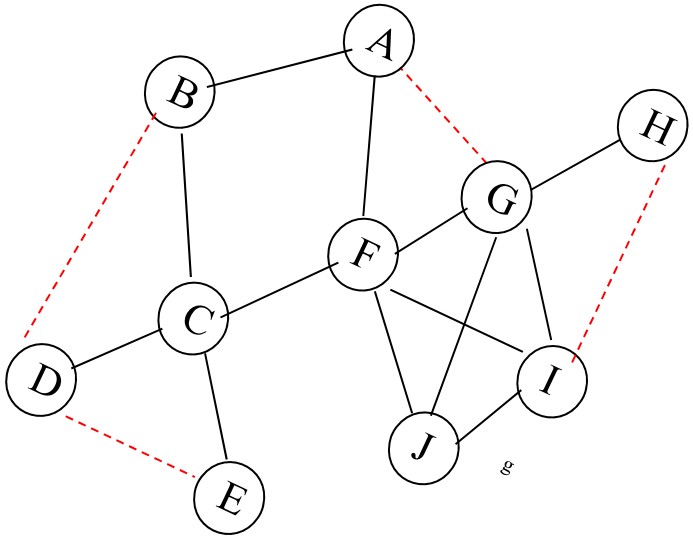
E2： 设B1，B2，…，Bk，是包含割点u的全部2-连通分支；

E3： 设 Vi是Bi的一个顶点，且Vi≠u，1≤i≤k。

E4： 将边(Vi, Vi+1)添加到G，1≤i≤k-1。

E5：**end{for}**

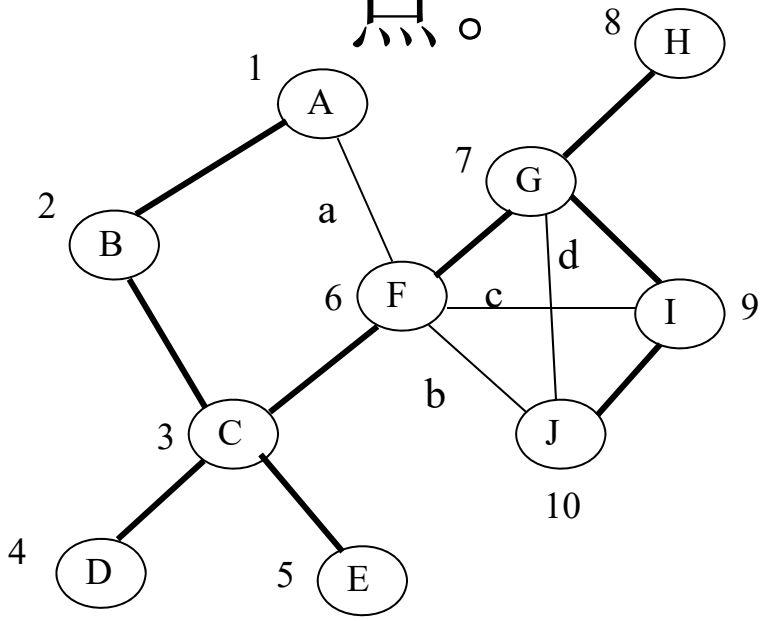
-



图G 添加边使成为2-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。

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

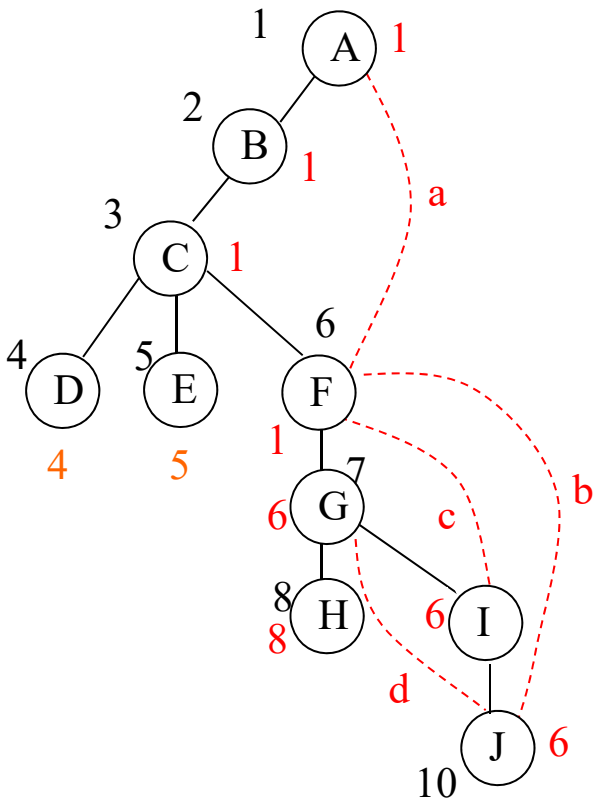
图G及其深度优先生成树**T**

采用深度优先算法得到深度优先搜索树,并且给每个顶点赋值深索数DFN(v)。

分析割点特征

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）关联着一条余边e，e的另一个端点是u的某个祖先；
4. 叶顶点不能是割点。

- 根据性质1，3，4，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。



深度优先树的分层

最低深索数 L(u)

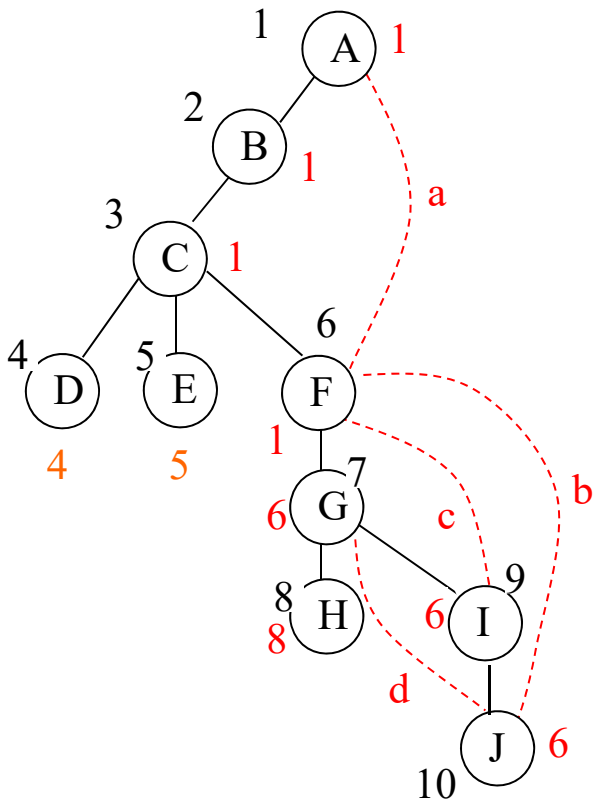
L(u):=min{ DFN(u),

min{L(w)|w是u的儿子},

min{DFN(x)|(u,x) 是T的余边}}

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)≥DFN(u), 对u的某个儿子w成立。

求法：采用深度优先搜索结合递归算法，逐层深入确定各点的深索数和最低深索数。



红色数字表示最低深索数

计算DFN和L的算法伪代码

```
DFNL(u,v)  //一个深度优先搜索算法，u是开始顶点。在深度
    //优先搜索树中， 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}
```

给出图G的各个2-连通分支

- 引进一个存放边的全程栈S；
- 在2到3行之间加下列语句：

2.1 **if** w≠v 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}**

对 策 树

- 取火柴游戏：

  规则：两人轮流从盘子上取走1，2或3支火柴为合法步骤；

  评判：拿走盘中最后一支火柴者为负。

- 对弈棋局：以盘中剩下的火柴数来表示该时刻的棋局。棋局序列C_0，C_1，…，C_{k-1},C_k称为有效序列，如果：

  1. C_0是开始棋局；
  2. C_i不是终止棋局，i＝0,1，2，…，k-1；
  3. 由C_i走到C_{i+1}是按下述规则进行的：

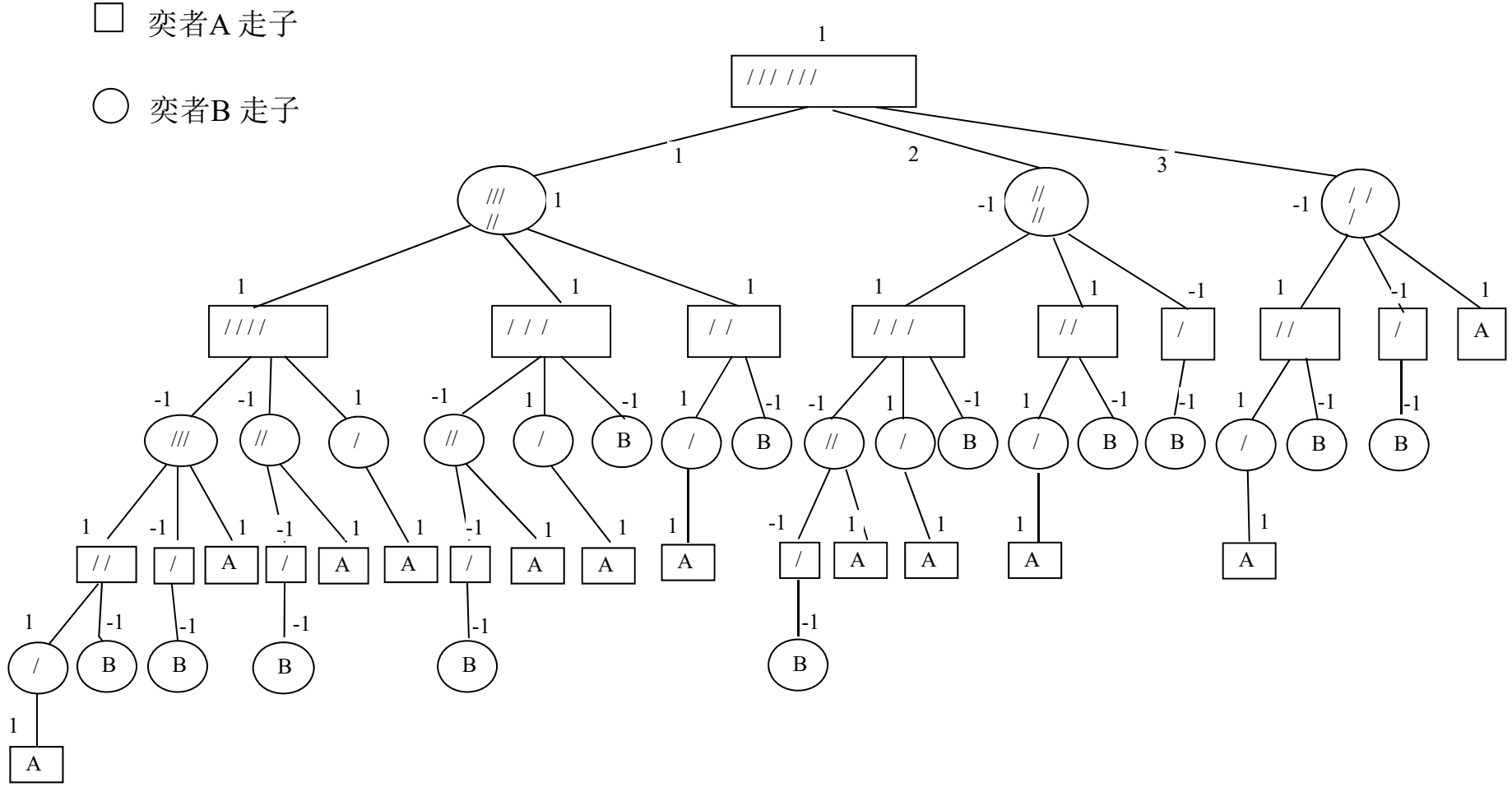
     若i是偶数，则A走一合法步骤；

     若i是奇数，则B走一合法步骤。
  4. 以C_k为终局.

一个有效棋局序列C_0，C_1，…，C_k是此游戏的一盘战例.

取火柴游戏所有可能的战例

X: 节点，代表一种棋局，V(X)表示该局下奕者A的胜负估计。



各种棋局下奕者胜算估值

奕者A,B,站在A的角度估计棋局X下，奕者A的胜率

- 代表棋局的顶点X是对策树的叶顶点时的胜率

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

- 代表棋局的顶点X不是对策树的叶顶点时的胜率计算

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

- 为了不要总是区别A走棋还B走棋，用V′(X)替换V(X)

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

- 若X是A走棋的位置，则e(X)=E(X);若是B走棋，则e(X)=-E(X)

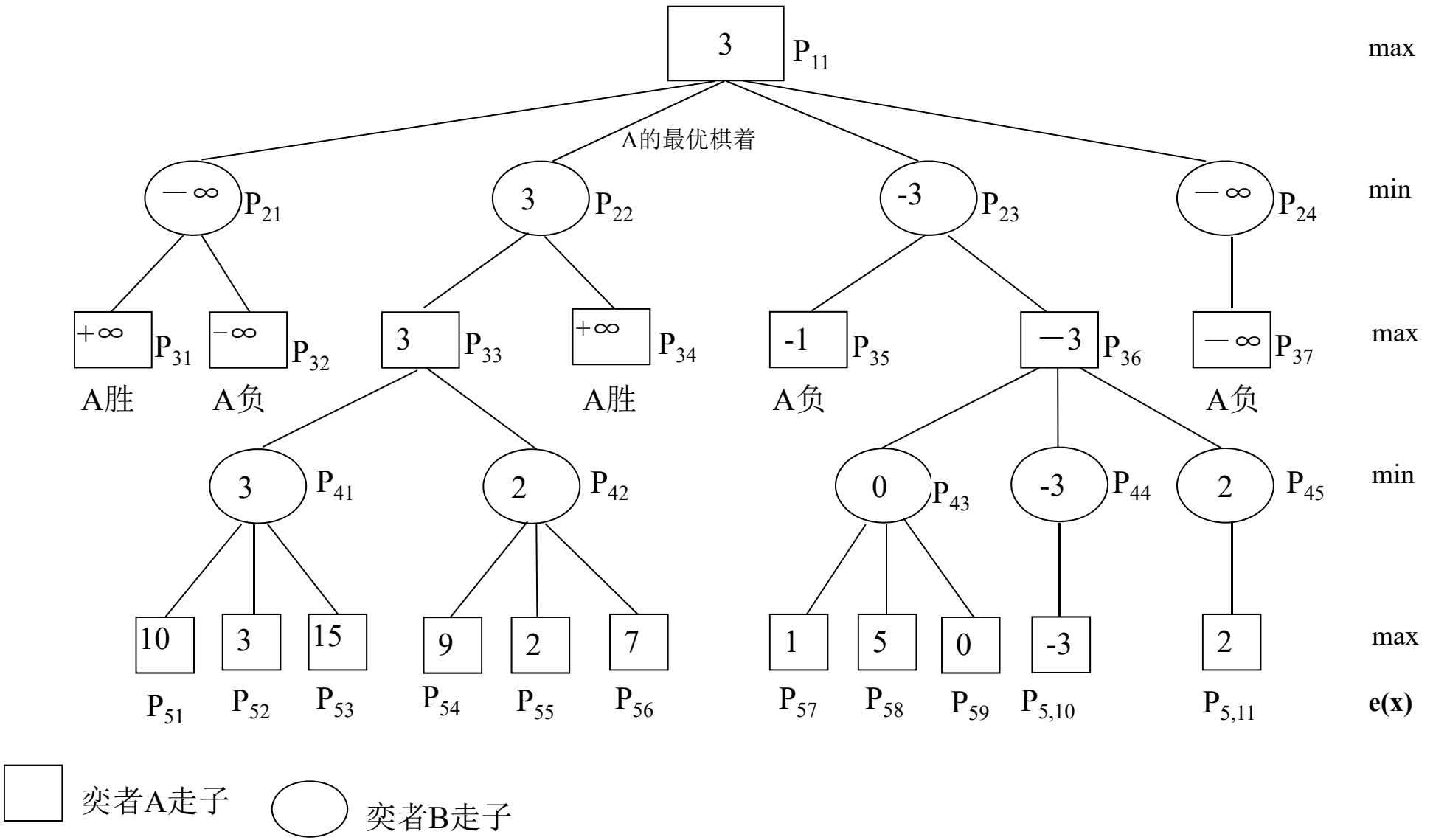
对策树的后根次序求值算法

VE(X,h)//通过至多向前看 h 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X_1，X_2，…，X_d.
- **if** X是终局或 h=0 **then** return(e(X)) **end{if}**
- ans:= -VE(X_1， h-1)；//遍历第一棵子树
- **for** i **from** 2 **to** d **do**
- ans:=max(ans,-VE(X_i， h-1))；
- **end{for}**
- return(ans);

end{VE}

一盘假想博弈游戏的部分对策树



$\alpha-\beta$ 截断规则

设Y是X的父亲，X有儿子 X_1，X_2，…，X_d，
则一旦知道

$V'(X_k) \le \mu(Y)$,

就不必再计算以

X_{k+1}，…，X_d

为根的子树的顶点的价值。

其中，$\mu(Y)$是当前所知道的V′(Y) 的最大可能值:

$$\mu(Y) \le V'(Y)$$

$$\mu(Y)=\begin{cases}\alpha(Y), \text{当Y取最大值时;}\\ \beta(Y), \text{当Y取最小值时。}\end{cases}$$

使用截断规则的后根次序求值算法

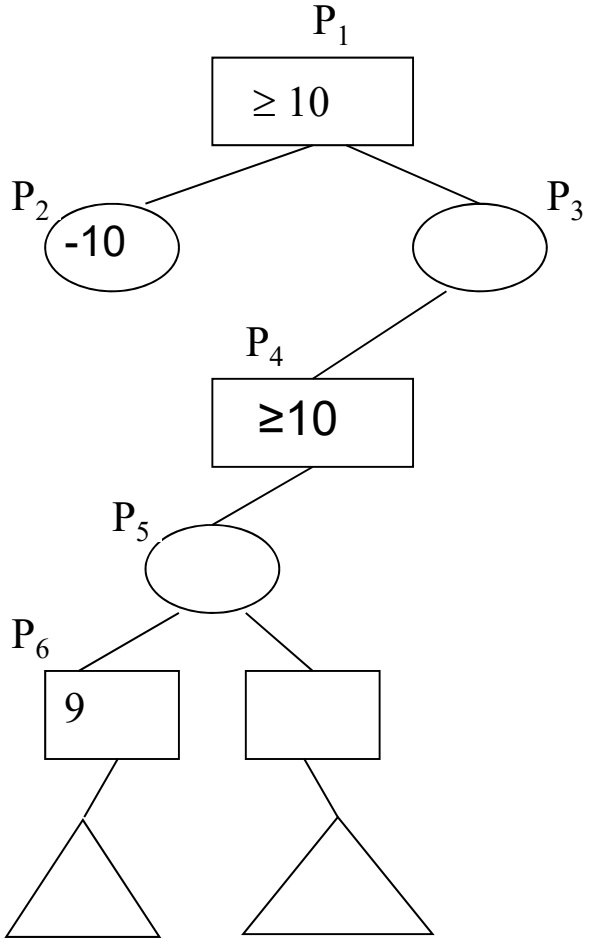
```
VEB(X, h,D)//通过至多向前看 h 着棋，使用
  //截断规则和公式计算V′(X)，弈者A的
  //估价函数e(X)假定由任一不是终局的棋局
  // X开始，此棋局的合法棋着只允许将棋局
  // X转换成棋局X1, X2, … , Xd
 if X是终局或 h =0 then return(e(X)) end{if}
 ans:=-VEB(X1, h-1,∞);
 for i from 2 to d do //使用截断规则
   if ans ≥ D then return(ans) end{if}
   ans:= max(ans,-VEB(Xi，h-1，-ans));
 end{for}
   return(ans);
end{VEB}
```

$\alpha-\beta$ 截断算法

VLB(X，h，LB，D) // LB是V′(X)的一个下界。

- //通过至多向前看 h着棋，使用截断规则计
  //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
  //X_1，X_2，…，X_d.
- **if** X是终局或=0 **then** return(e(X)) **end{if}**
- ans: = LB；
- **for** i **to** d **do**
- //使用截断规则
- **if** ans ≥ D **then** return（ans）**end{if}**
- ans: = max(ans,-VLB(X_i，h-1,-D,-ans))；
- **end{for}**
- return(ans);

end{VLB}



比较VEB和VLB的执行情况

- 于图中假想策树，使用算法VLB比使用算法VEB产生更大的截断。调用VEB时，假定最初的调用为VEB(P_1,h,∞),其中，h是这棵树的高度。在检查了P_1的左子树之后，知道P_1处估值的一个下界值10,并且相继生成节点P_3,P_4,P_5和P_6。此后，$V'(P_6)$确定为9，至此，知道P_5处估值的一个下界值-9。算法接下去要计算节点P_7处的估值。调用算法VLB时，假定最初的调用为VLB(P_1, h,-∞，∞), 在检查了P_1的左子树之后，知道P_1处估值的一个下界值10,因而知道P_4处的估值应该不小于10。根据算法，在检索节点P_5时所知道的P_4处估值的最好下界还是10，因而，P_6处的估值也应该不小于10。但是，在检查得知P_6处的估值为9时，算法就结束了，节点P_7不必生成。