第四章　图的连通性

4.1　连通性的基本概念和定义

图的连通性这个概念十分重要，它的直观意义就是“连成一片”。当然，如果把连通图定义为“连成一片的图”就不太好，因为如果再追问一句：“什么叫连成一片？”恐怕就不好回答了。因此必须给图的连通性下一个准确的定义：

在无向图 G 中，如果从顶点 V 到顶点 V' 有路径，则称 V 和 V' 是连通的。如果对于图中任意两个顶点 $V_i, V_j \in V$，V_i 和 V_j 都是连通的，则称是连通图。

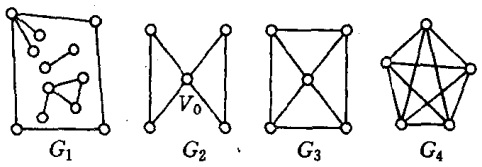


图　4-1

图 4-1 中的 G_2, G_3, G_4 是连通图，而 G_1 则是非连通图，但 G_1 有 3 个连通分量，如图 4-2 所示。所谓连通分量指的是无向图中的极大连通子图。

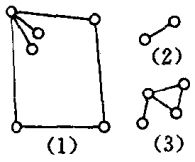


图　4-2

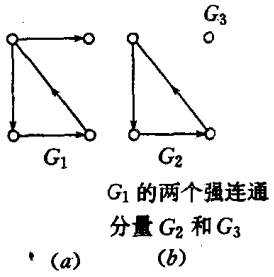


图　4-3

在有向图 G 中，如果对于每一对 $V_i, V_j \in V$，$V_i \neq V_j$，从 V_i 到 V_j 和从 V_j 到 V_i 都存在路径，则称 G 是强连通图。有向图中的极大强连通子图称作有向图的强连通分量。

例如图 4-3(a)中的 G_1 不是强连通图，但它有两个强连通分量，如图 4-3(b)所示。

一个连通图的生成树是一个极小连通子图，它含图的全部顶点，但只有足以构成一棵树的 $n-1$ 条边。图 4-4 是图 4-2 所示 G_1 中最大的连通分量的一棵树。如果在一棵生成树上添加一条边必构成一环，因为这条边使得它依附的那两个顶点之间有了第 2 条路径。一棵有 n 个顶点的生成树有且仅有 $n-1$ 条边。如果一个图有 n 个顶点而小于 $n-1$ 条边，则是非连通图，如果它多于 $n-1$ 条边，则一定有环。但是，有 $n-1$ 条边的图不一定是生成

树。

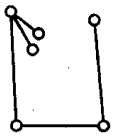


图4.2所示G_1的最大连通分量的一棵生成树

图 4-4

对于任意连通图来说，是否连通的程度都一样？不是。例如图 4-1 中的 G_2、G_3、G_4 都是连通图，但是连通的程度大不一样。G_2 删除一边后仍连通，但它有一个顶点 V_0，删去 V_0 和与其关联的各边后，G_2 就不连通了；G_3 只删除一条边或一个顶点后仍连通，而 G_4 是一个具有 5 个顶点的完全图，删除任意 3 个顶点或 3 条边都不能使其破碎。为精确刻画连通的程度，我们引入两个量：边连通度和顶连通度。

一、顶连通度 $K(G)$

V'是连通图 G 的一个顶点子集。在 G 中删去 V' 及与 V' 相关联的边后图不连通，则称 V' 是 G 的割顶集。最小割顶集中顶点的个数，记成 $K(G)$，叫做 G 的连通度。规定 K(完全图)＝顶点数－1；K(不连通图)＝K(平凡图)＝0。$K(G)=1$ 时，割顶集中的那个顶点叫做割顶。没有割顶的图叫做块，G 中成块的极大子图叫做 G 的块。

例如图 4-1 中，$K(G_2)=1$，$K(G_3)=3$，$K(G_4)=5-1=4$；G_3 与 G_4 是块；G_2 中有两个三角形块。

二、边连通度 $K'(G)$

E'是连通图 G 的一个边子集。在 G 中删去 E' 中的边后图不连通，则称 E' 是 G 的桥集。若 G 中已无桥集 E''，使得 $|E''|<|E'|$，则称 $|E'|$ 为 G 的边连通度，记成 $K'(G)$。$|E'|=1$ 时，E' 中的边叫做桥。规定 K'(不连通图)＝0，K'(完全图)＝顶点数－1。

例如图 4-1 中，$K'(G_2)=2$，$K'(G_3)=3$，$K'(G_4)=5-1=4$。G_2，G_3，G_4 中无桥。

对于任意一个连通图 G，在计算出上述两个量后，我们可以给该图下一个定义：

$K(G)\geqslant M$，G 叫做 M 连通图；$K'(G)\geqslant M$ 时，G 称为 M 边连通图。

例如图 4-1 中，G_2 是 1 连通图，2 边连通图，当然也是 1 边连通图；G_3 是 3 连通图，当然也是 1，2 连通图，G_3 是 3 边连通图，当然也是 1，2 边连通图；G_4 是 4 连通图，当然也是 1，2，3 连通图，G_4 是 4 边连通图，当然也是 1，2，3 边连通图。

M 连通图，当 $M>1$ 时，也是 $M-1$ 连通图。

M 边连通图，当 $M>1$ 时，也是 $M-1$ 边连通图。

下面给出连通图 G 的两个特征：

1. $K(G)\leqslant K'(G)\leqslant G$ 的顶点度数的最小值 &

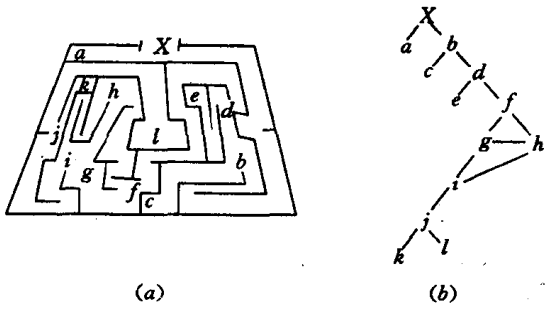
2. 顶点数大于 2 的 2 连通图的充分必要条件是任两个顶点在一个圈上。

由于连通度和边连通图的计算涉及网络流知识，因此我们将在第六章作专门介绍。这一章里我们给出求割顶、块以及极大强连通子图的算法，并给出两个实际应用。在此之前，我们先介绍一个关键算法——深度优先搜索(dfs)。它在解图的连通性问题中扮演了极其重要的角色，而且其思想方法还渗透到其它许多图论算法的设计中。

4.2 深度优先搜索(dfs)

一、什么是深度优先搜索

为了理解深度优先搜索的思路，让我们追溯到1690年修筑的威廉王迷宫，它至今保存完好(见图4-5(*a*))。



(*a*) (*b*)

图 4-5

迷宫法则：任务是从迷宫入口处出发，每个走廊都要搜索，最后再从入口出来。为了不兜圈子，需要记住哪些走廊已经走过，沿着未走过的通道尽可能远地走下去，走到死胡同或那里已无未走过的走廊可选时，沿原路返回，到达一个路口，发现可通往一条未走过的走廊可选时，沿这一未走过的走廊尽可能远地走下去……，最后即可搜索遍全部走廊和厅室，再由入口处出迷宫。

我们把图4-5(*a*)中字母表示的厅室看作是顶点。如果两个厅室之间通有走廊，则在代表它们的两个顶点之间连一条边，两个厅室之间没有通道就不连。这样形成了一个图(图4-5(*b*))。

迷宫法则粗略地描述出深度优先搜索的算法轮廓。现在我们针对抽象出的无向图再作一番讨论。为了简略起见，先看连通图。对于非连通图，则只要在每个连通分量中用一次dfs算法便可遍历图的每个顶点。

先任取一个顶点作为根，记为U，现认为U顶点已访问过(在迷宫问题中，取入口X作为根)。再在U的邻接顶点中，任选W顶点，形成关联边(U,W)，且对这条边定方向为U到W。此时边(U,W)称为“已检查”，且送入树枝边集合中，U称为W的父亲点，记为$U=\text{father}(W)$。

一般情况下当访问某顶点X时，有两种可能：

1. 如果X顶点的所有关联边被检查过了，则回溯至X的父亲顶点，从$\text{father}(X)$再继续搜索，此时X顶点已无未用过的关联边；

2. 如存在X顶点未用过的关联边，则任选其中一边(X,Y)，对其定向为从X到Y，此时说边(X,Y)正等待检查。

检查结果有两种情况：

1. 如Y顶点从未被访问过，则顺着(X,Y)边访问Y，下一步从Y开始继续搜索。此时

(X,Y)是前向边，且顶点 $X=\text{father}(Y)$，图上用实线表出；

2. 如 Y 顶点已被访问过，则再选 X 顶点未用过的关联边．此时(X,Y)是后向边，用虚线表出。

此时，(X,Y)的归属已明确，(X,Y)就检查完毕。在 dfs 搜索期间，根据顶点 X 被首次访问的次序安排了不同的整数 dfn(X)，如 $\text{dfn}(X)=i$，则 X 是第 i 个首次被访问的顶点，$\text{dfn}(X)$称为 X 的深度优先搜索序数，当搜索返回根时，连通图的所有顶点都访问完毕。现在图中的边都定了方向，且分成前向边(实线)组成的有向树以及后向边(虚线)，这个有向生成树称为 dfs 树。

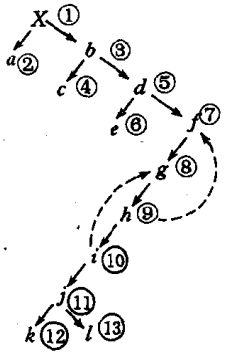
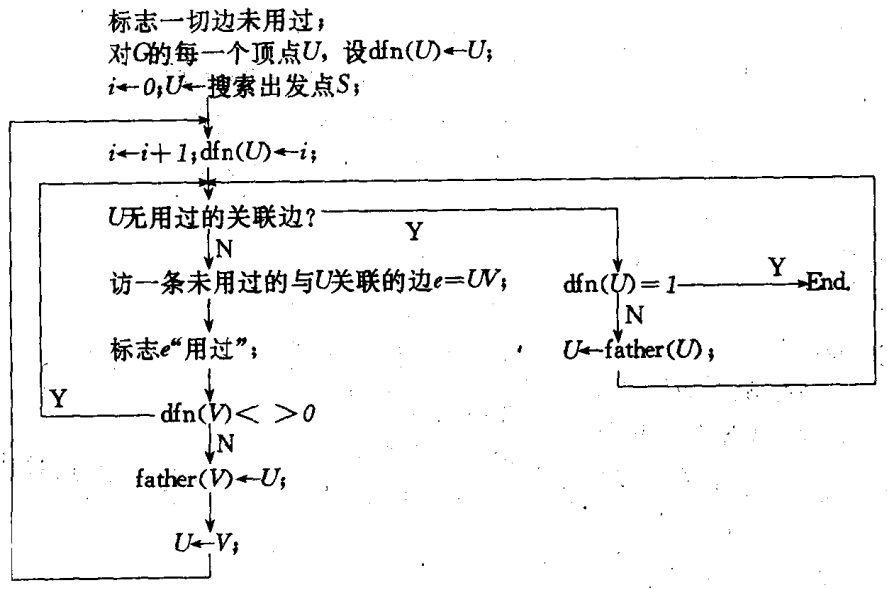


图 4-6

图 4-6 给出图 4-5(b)的 dfs 搜索过程。每个顶点旁○内的数字为 dfn 值。

显然 dfs 算法类似树的前序遍历。

二、无向图 G 的 dfs 算法分析



为了便于在以后各章节中深入开发上述算法，再引进一些术语。现在 G 从无向图改造成有向图，且每顶点有 dfn 值，前向边组成 dfs 树，(若非连通图，则对每一个连通分量进行一次 dfs 搜索，结果产生一个由若干 dfs 树组成的森林)。如果从 U 到 W 有树边组成的有向路，则称 U 和 W 有直系关系，U 称为 W 的祖先点，W 称为 U 的后代点。如(U,W)是树中一条边，U 又确切地称为 W 的父亲点，W 也可称为 U 的儿子点。一个顶点可有多个儿子点，顶点 U 和它的所有后代点组成关于树中以 U 为根的子树。

如果 U 和 W 不存在直系上下关系，则说它们有旁系关系。在这种情况下，$\text{dfn}(U)<\text{dfn}(W)$，则说 U 在 W 的左边，或 W 在 U 的右边。连接旁系之间的边叫横叉边。在 dfs 搜索中，G 不产生横叉边。或者说：如(U,W)是无向图 G 的一条边，则任何一种 dfs 树(取不同的根可有不同的树)，U 或是 W 的祖先点，或是 W 的后代点。

为什么？请读者自行分析。

下面讨论有向图的 dfs。

有向图的 dfs 本质上近似无向图，区别在于搜索只能顺边的方向去进行（有向边，简称弧）。有向图中的弧根据被检查情况可有 4 种，一条未检查的弧(U,W)可按如下 4 种情况，分类归划成（其中 2.，3.，4. 所述的三种情况中，设 W 已访问过）：

1. W 是未访问过的点，(U,W)归入树枝弧（见图 4-7(a)）

2. W 是 U 的已形成的 dfs 森林中直系后代点，则(U,W)称为前向弧（见图 4-7(b)）。无论(U,W)是树枝弧还是前向弧，$\mathrm{dfn}(W)>\mathrm{dfn}(U)$。细分一下，树枝边总使搜索导向一个新的（未访问）点，且 $\mathrm{dfn}(U)+1=\mathrm{dfn}(W)$。

3. W 是 U 的已形成的 dfs 森林中直系祖先点，则(U,W)称为后向弧（见图 4-7(c)）。

4. U 和 W 在已形成的 dfs 森林中没有直系上下关系，并且有 $\mathrm{dfn}(W)<\mathrm{dfn}(U)$，则称$(U,W)$是横叉弧。注意没有 $\mathrm{dfn}(W)>\mathrm{dfn}(U)$这种类型的横叉弧。因为 dfs 搜索只能从左到右逐条树枝地进行（见图 4-7(d)）。

无论(U,W)是后向弧（W 的所有关联边未完全检查）还是横叉弧（W 的所有关联边完全检查），$\mathrm{dfn}(W)>\mathrm{dfn}(U)$。

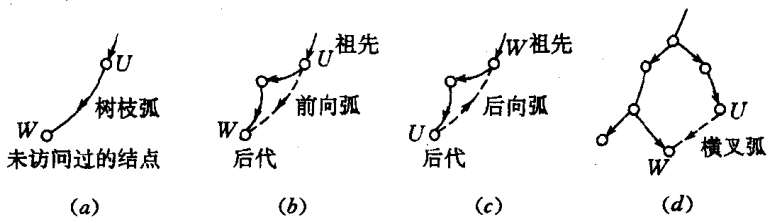


图 4-7

例如图 4-8 所示的有向图经 dfs 后，弧划分为：

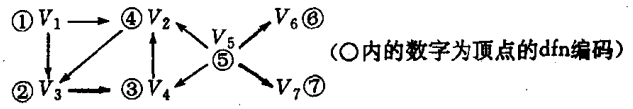
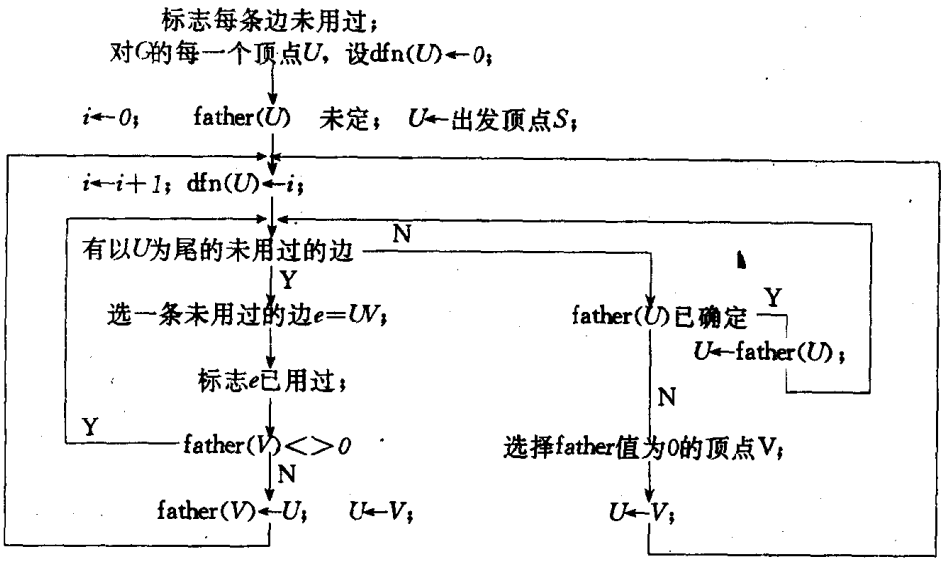


图 4-8

$<V_1V_3>$，$<V_3V_4>$，$<V_4V_2>$，$<V_5V_6>$，$<V_5V_7>$是树枝弧，组成 dfs 森林（用粗线表示）；$<V_2V_3>$是后向弧；$<V_1V_2>$是前向弧；$<V_5V_2>$，$<V_5V_4>$是横叉弧。

下面给出有向图 G 的 dfs 算法：



三、无向图的 dfs 程序

由于有向图的 dfs 本质上近似无向图，况且这个算法将在 4.4 节求极大强连通子图中还会使用到，因此这里不再作介绍。

```
program dfs_p94;

  const
    maxn          =  30;

  type
    node          =  record
                       k,f : 1..maxn { 顶点类型： DFS 编码，父指针 }
                     end;
    ghtype        =  array [1..maxn,1..maxn] of integer;
    ltype         =  array [1..maxn] of node;

  var
    g             :  ghtype;     { 邻接矩阵 }
    n             :  integer;    { 顶点数 }
    f             :  text;       { 文件变量 }
    l             :  ltype;      { 顶点序列 }

  procedure read_graph;
    var
      str : string;
      i,j : integer;
    begin
      write('Graph file = ');   { 读入外部文件名并与文件变量连接 }
      readln(str);
      assign(f,str);
      reset(f);
      readln(f,n);   { 读入结点数 }
```

```
    for i:=1 to n do { 读入图矩阵 }
      for j:=1 to n do read(f,g[i,j]);
    close(f); { 关闭文件 }
  end;

procedure DFS(s:integer);
  var
    i,v,u:integer;
  begin
    fillchar(l,sizeof(l),0); { l数组清零 }
    i:=0;v:=s; { s作为第一个访问结点，即深度优先树的根 }
    repeat
      repeat
        inc(i);l[v].k:=i; { 设置v顶点的dfn编码 }
        repeat
          u:=1;
          while (u<=n) and (g[v,u]<=0) do inc(u);
          { 选一条未用过的与v关联的边<v,u> }
          g[v,u]:=-g[v,u];g[u,v]:=-g[u,v]; { 标志<u,v>用过 }
        until (u=n+1) or (l[u].k=0);
        { 直至与v关联的所有边都使用过或者<v,u>的顶点u未曾编码为止 }
        if u<>n+1                    { 若<v,u>的顶点u未曾编码,则 }
          then begin
                 writeln(v,'——',u);
                   { 打印边<v,u>,设置u的父亲结点为v,从u出发搜索 }
                 l[u].f:=v;v:=u
               end
      until u=n+1; { 直至与v关联的边都使用过 }
      if l[v].k<>1 then v:=l[v].f { v不是根结点,回溯至v的父亲结点 }
    until l[v].k=1 { 直至回溯至根结点 }
  end;
begin
  read_graph; { 输入图 }
  DFS(1)      { 从顶点1出发,作深度优先搜索 }
end.
```

4.3 求割顶和块

一、什么是割顶和块

分析一个由无向图表示的通讯网，顶点看作是通讯站，无向边代表两个通讯站之间可以互相通讯。问：

1. 若其中一个站点出现故障，是否会影响系统正常工作；

2. 这个通讯网可以分成哪几个含站点尽可能多的子网，这些子网内的任一站点出现故障，都不会影响子网工作。

有了连通性的基础知识后，我们不难得出，问题1所指的那个“牵一发动全身”的站点，称为割点。在删去这个顶点以及和它相连的各边后，会将图的一个连通分割成两个或

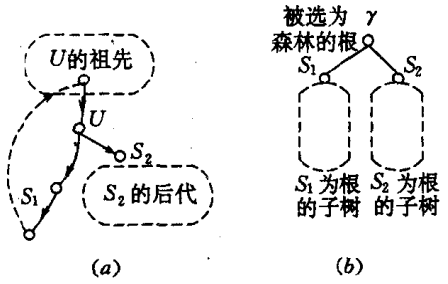
两个以上的连通分量，影响系统正常工作；而问题 2 所指的那些内部任一站点出现故障都不会影响其通讯的最大子网，指的是没有割点的连通子图。这个子图中的任何一对顶点之间至少存在两条不相交的路径，或者说要使这个子网不连通，至少要两个站点同时发生故障。这种二连通分支也称为块。

显然各个块之间的关系或者互不连接，或者通过割顶连接。这割顶可以属于不同的块，也可以两个块共有一个割顶。因此无向图寻找块，关键是找出割顶。下面有两个基本事实：

对一个给定的无向图，实施 dfs 搜索得到所有顶点的 dfn 值及 dfs 树（或森林）后，

1. 如 U 不是根，U 成为割顶当且仅当存在 U 的某一个儿子顶点 S，从 S 或 S 的后代点到 U 的祖先点之间不存在后向边（见图 4-9(a)）；

2. 如 γ 被选为根，则 γ 成为割顶当且仅当它有不止一个儿子点（见图4-9(b)）。



(a)　　(b)

图 4-9

由图 4-9(a)可以看出，虽然 U 的以 S_1 为根的子树中有后向边返回 U 的祖先，但是 U 的某儿子 S_2 或 S_2 的后代没有返回 U 的祖先，因此由基本事实 1 得出 U 是割点。

由图 4-9(b)可以看出，根 γ 存在两棵分别以 S_1 和 S_2 为根的子树，这两棵子树间没有横叉边（无向图不存在横叉边），因此由基本事实 2 得出 γ 是割顶。

上述两个基本事实是构思算法的精华，为此引入一种顶点 U 的标号函数 LOW(U)

$$\mathrm{LOW}(U)=\min(dfn(U),\mathrm{LOW}(S),dfn(W))$$

其中：S 是 U 的一个儿子，(U,W)是后向边。

显然 LOW(U)值恰是 U 或 U 的后代所能追溯到的最早（序号小）的祖先点序号。一般约定，顶点自身也认为自己是祖先点。所以有可能 LOW(U)=dfn(U)或 LOW(U)=dfn(W)。

利用标号函数 LOW，我们可以将基本事实 1 重新描述成：

顶点 $U\neq\gamma$ 作为 G 的割顶当且仅当 U 有一个儿子 S，使得 LOW(S)$\geqslant dfn(U)$，即 S 和 S 的后代不会追溯到比 U 更早的祖先点。

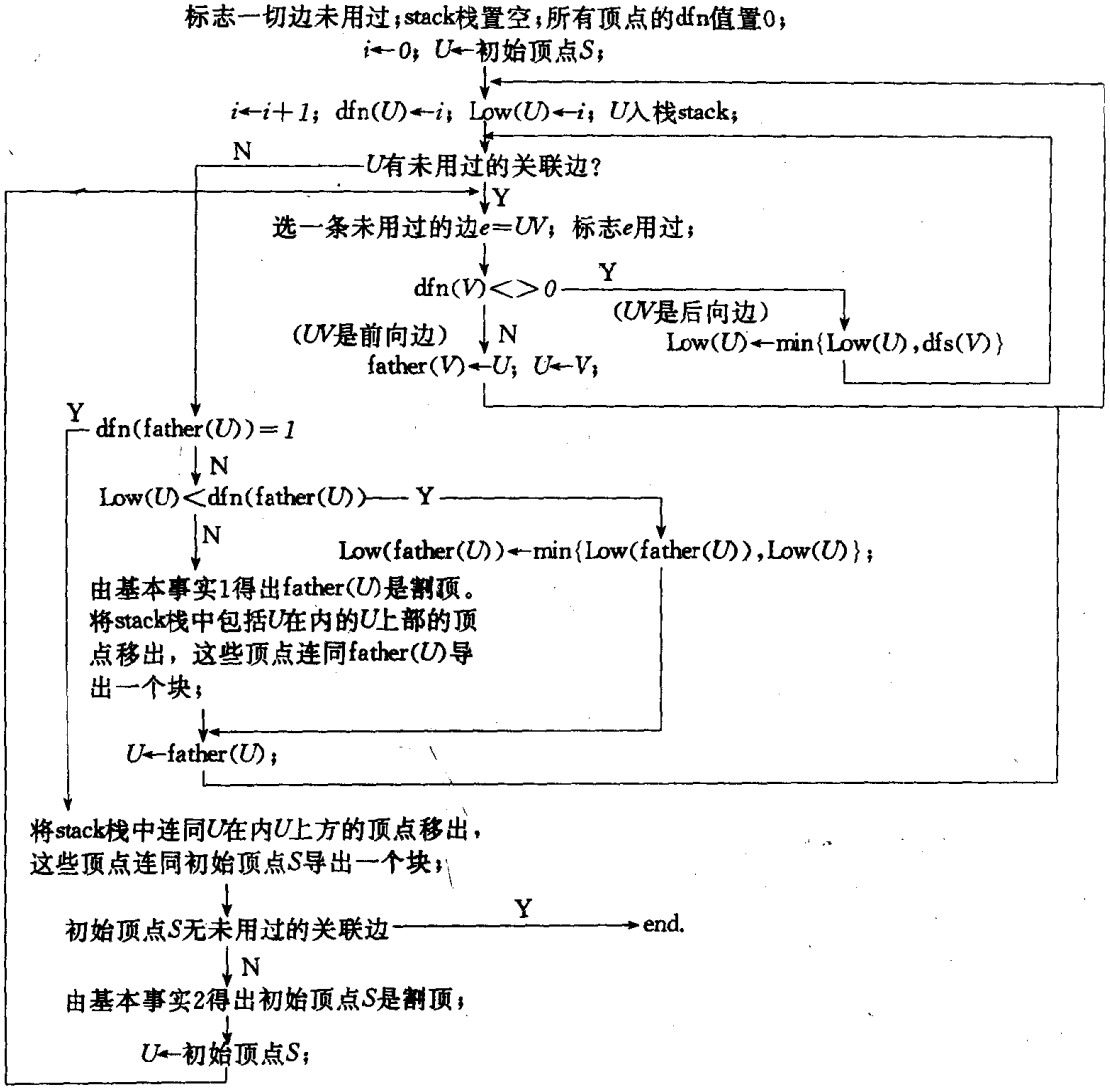
LOW(U)值的计算步骤如下：

$$\mathrm{LOW}(U)=\begin{cases}\text{—}\mathrm{dfn}(U)\\ \quad U\text{ 在 dfs 过程中首次被访问；}\\ \text{—}\min(\mathrm{LOW}(U),\mathrm{dfn}(W))\\ \quad \text{检查后向边}(U,W)\text{时；}\\ \text{—}\min(\mathrm{LOW}(U),\mathrm{LOW}(S))\\ \quad U\text{ 的儿子 }S\text{ 的关联边全部被检查时。}\end{cases}$$

在算法执行中，对任何顶点 U 计算 LOW(U)值是不断修改的，只有当以 U 为根的 dfs 子树和后代的 LOW 值、dfn 值产生后才停止。

二、求割顶和块的算法

根据前面的分析求割顶和块的算法如下面的逻辑图所表示。



例如图 4-10(*a*)

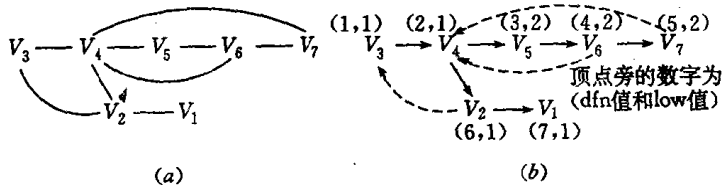


图 4-10

从 V_3 出发，按上述算法进行搜索，得出 dfs 树和各顶点 dfn 值、LOW 值(见图 4-10 (*b*))。由于$<V_4V_5>$是前向边，且 dfn(V_4)=2>1，LOW (V_5)=2，又 LOW(V_4)=2，由基本事实 1 得出 V_4 是割顶，并导出$\{V_4,V_3,V_2\}$和$\{V_4,V_5,V_6,V_7\}$两个块。

三、求割顶和块的程序

```
Program QIU_KUI_HE_GE_DING;

const
    maxn        =  30;

type
  node          =  record
                     k,l : 1..maxn { 顶点类型: dfn 编码,low 编码 }
                   end;
  ghtype        =  array [1..maxn,1..maxn] of integer; { 邻接矩阵类型 }
  ltype         =  array [1..maxn] of node;

var
  g             :  ghtype; { 邻接矩阵 }
  n,p,x,i       :  integer;{ 顶点数,栈指针,根的子树个数,辅助变量 }
  f             :  text;   { 文件变量 }
  l             :  ltype;  { 顶点序列 }
  st            :  array [1..maxn] of integer; { 栈 }
  k             :  set of 1..maxn; { 割顶集 }

function min(a,b : integer) : integer; { 返回 a,b 间的小者 }
  begin
    if a>=b then min :=b
            else min :=a
  end;

procedure push(a : integer); { a 入栈 st }
  begin
    inc(p);st[p] :=a
  end;

function pop : integer; { 返回 st 的栈顶元素,退栈 }
  begin
    pop :=st[p];dec(p)
  end;

procedure read_graph;
  var
    str : string;
    i,j : integer;
  begin
    write('Graph file = '); { 输入外部文件名并与文件变量连接 }
    readln(str);
    assign(f,str);
    reset(f);
    readln(f,n); { 读入顶点数 }
    for i :=1 to n do  { 读入图矩阵 }
```

```
    for j:=1 to n do read(f,g[i,j]);
  close(f);
  fillchar(l,sizeof(l),0);{ l数组初始化 }
  p:=0;push(1);              { 顶点1入栈 }
  i:=1;l[1].k:=i;l[1].l:=i;{ 顶点1首次被访问 dfs(1)=low(1)=1 }
  k:=[];x:=0{ 割顶集置空 }
end;

procedure algorithm(v:integer);
  var
    u:integer;
  begin
    for u:=1 to n do
      if g[v,u]>0{ 若<v,u>边存在,则分析 }
        then if l[u].k=0{ 若u首次被访问 }
               then begin
                      inc(i);l[u].k:=i;l[u].l:=i;push(u);
                      { 置u的dfs和low值;u入栈 }
                      algorithm(u);
                      { 从u结点出发,递归求解 }
                      l[v].l:=min(l[v].l,l[u].l);
                      { v的儿子u完全扫描毕,置v的low值 }
                      if l[u].l>=l[v].k
                        { 从u及u的后代不会追溯到比v更早的祖先点 }
                        then begin
                               if (v<>1) and not (v in k)
                      { 若v不是根且不属割顶集,则打印割顶v,v进入割顶集 }
                                 then begin
                                        writeln('[',v,']');
                                        k:=k+[v]
                                      end;
                               if v=1 then inc(x);
                      { 若v是根,计算v根下的子树个数 }
                               while st[p]<>v do write(pop:3);
                      { 从栈中依次取出至v的各顶点,产生一个连通分支 }
                               writeln(v:3)
                             end
                      end
               else l[v].l:=min(l[v].l,l[u].k)
                    { <v,u>是后向边,u重复访问,则置v的low值 }
  end;

begin
  read_graph;  { 输入图 }
  algorithm(1);{ 递归求解割顶集与块 }
  if x>1 then write('[',1,']');{ 若根不止有一个儿子,则根作为割顶打印 }
end.
```

4.4 求极大强连通子图

一、什么是极大强连通子图

让我们先看一个例子。图 4-11 画了一个有向图，它的每一个顶点代表某篮球队的一个队员，它的弧代表的意思是如果有一条从 V_i 出发而指向 V_j 的弧(i,j)，就表示队员 V_i 能够通知 V_j。

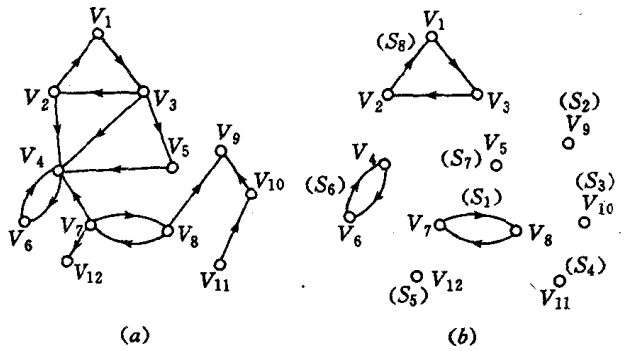


图 4-11

问这张有向图至少能划分几个最大子图，这些子图上的所有队员能相互通知(包括子图仅含一个队员，表示自己通知自己)。

有了连通性的基本知识后，我们不难得出，上述问题要求的最大子图即为极大强连通子图，在这个子图上的任一对顶点可以互相可达。按这个要求划分子图，图 4-11(a)的有向图共有 8 个强连通分支，它们分别是由

$$S_1=\{V_7,V_8\} \qquad S_2=\{V_9\}$$
$$S_3=\{V_{10}\} \qquad S_4=\{V_{11}\}$$
$$S_5=\{V_{12}\} \qquad S_6=\{V_4,V_6\}$$
$$S_7=\{V_5\} \qquad S_8=\{V_1,V_2,V_3\}$$

生成的(见图 4-11(b))。

那么，如何在有向图中寻找任意两个顶点都可用有向路双向连通的最大子图呢?还是采用 dfs 算法。因为每一个强连通子图从 dfs 树看，都是以 dfn 值最小的顶点为根的子树，这个根亦被称作强分支的根。

为了在 dfs 森林的基础上找强分支子树的根，我们引入了顶点 U 的标值函数 LOWLINK(U)：

$$\text{LOWLINK}(U)=\min\{\text{dfn}(U),\text{dfn}(W)\}$$

其中 W 是从 U 或 U 的后代点出发用一条后向弧或横叉弧所能到达的同一强连通分支中的顶点。

由此可以看出，LOWLINK(U)正是 U 所处的强分支中从 U 出发先用树枝弧、前向弧，最后用后向弧或横叉弧到达的 dfn 最小的顶点的序值。而对于强分支的根 r_i 显然有

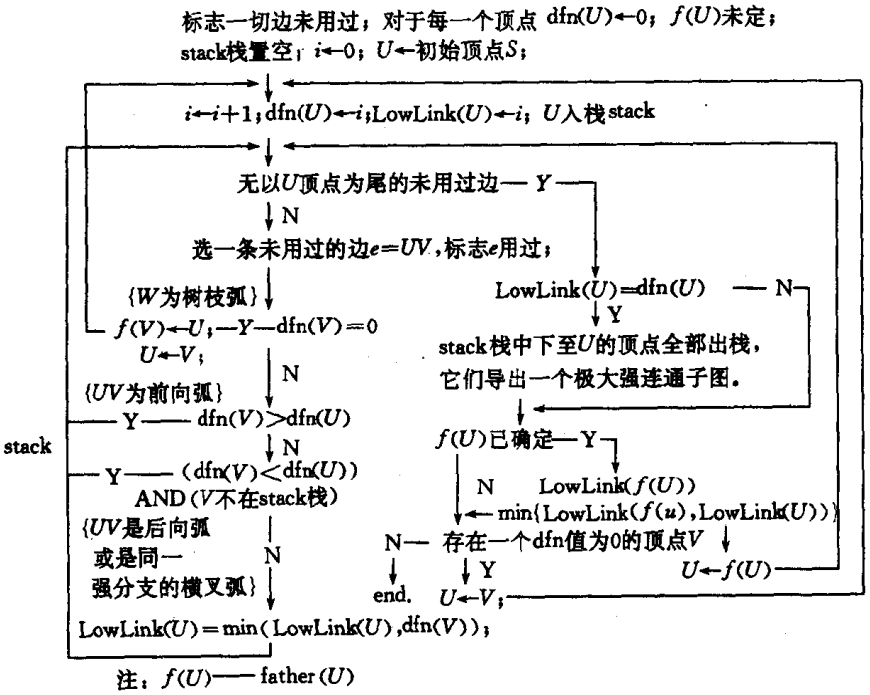
LOWLINK(r_i)=dfn(r_i)。因此当深度优先搜索从一个使 dfn 编码=LOWLINK 值的顶点 U 返回时，从树中移去根为 U 的所有顶点。每一个这种顶点的集合是一个强分支，即极大强连通子图。

算法的主要思路是用逐步迭代计算出 LOWLINK(U)值。

$$\text{LOWLINK}(U)=\begin{cases}\text{dfn}(U) & \text{第一次访问顶点 } U \text{ 时；}\\ \min(\text{LOWLINK}(U),\text{dfn}(W)) & \text{后向弧}(U,W)\text{被检查时；}\\ \min(\text{LOWLINK}(U),\text{dfn}(W)) & \text{处于同一强分支的横叉弧}(U,W)\text{被检查时；}\\ \min(\text{LOWLINK}(U),\text{LOW}\mathit{LINK}(S)) & \text{检查了 } U \text{ 的儿子 } S \text{ 的所有关联边后返回顶点 } U \text{ 时；}\end{cases}$$

二、求极大强连通分支的算法

下面介绍求极大强连通分支的算法。



三、求极大强连通分支的程序

```
program qiu_ji_da_qiang_liang_tong_de_suan_fa;

  const
    maxn          =  30;
```

```
type
  node        =  record
                   k,l : 1..maxn { 顶点的 dfs 和 lowlink 值 }
                 end;
  ghtype      =  array [1..maxn,1..maxn] of integer;
  ltype       =  array [1..maxn] of node;

var
  g           :  ghtype;  { 邻接矩阵 }
  n,p,i,x     :  integer; { 顶点数,栈指针,辅助变量,子树个数 }
  f           :  text;    { 文件 }
  l           :  ltype;   { 顶点序列 }
  st          :  array [1..maxn] of integer;
                 {存储当前强连通分支顶点的栈 }

procedure read_graph;
  var
    str : string;
    i,j : integer;
  begin
    write('Graph file = '); { 读入外部文件名,并与文件变量连接 }
    readln(str);
    assign(f,str);
    reset(f);
    readln(f,n); { 读入顶点数 }
    for i := 1 to n do
      for j := 1 to n do read(f,g[i,j]); { 读入邻接矩阵 }
    close(f);
    fillchar(l,sizeof(l),0); { 顶点信息初始化 }
    p := 1;
  end;

function min(a,b : integer) : integer; { 返回 a,b 间的小者 }
  begin
    if a>=b then min := b
            else min := a
  end;

function check(a : integer) : boolean;
{ st 栈中存在 a 返回 false,否则返回 true }
  var
    i : integer;
```

```
  begin
    check:=true;
    for i:=1 to p do if a=st[i] then exit;
    check:=false
  end;

procedure require(v:integer);{ 求以 v 为根的强连通分支 }
  var
    u:integer;
  begin
    inc(i);l[v].k:=i;l[v].l:=i;   {置 v 结点的 lowlink 和 dfs 值}
    inc(p);st[p]:=v;              {v 结点入栈}
    for u:=1 to n do
      if g[v,u]>0                 { 若<v,u>存在 }
        then if l[u].k=0          { 若 u 未检查,即<v,u>是前向弧 }
               then begin
                      require(u); { 从 u 出发,递归求解 }
                      l[v].l:=min(l[v].l,l[u].l);
                      { v 的儿子 u 被检查完毕,置 v 的 lowlink 值 }
                    end
               else if (l[u].k<l[v].k) and check(u)
{ <v,u>是后向弧或横向弧、u 被检查过且 u 与 v 处于同一强分支中,则修改 v 的 lowlink 值 }
                      then l[v].l:=min(l[v].l,l[u].k);
      if l[v].l=l[v].k { 若关联于 u 的全部弧都检查且 v 是强连通分支中的根 }
        then begin
               repeat { st 栈中至 v 的顶点相继出栈,组成一个强连通分支 }
                 write(st[p]:3);dec(p)
               until st[p+1]=v;
               writeln
             end
    end;

  procedure proceed;
    var
      x:integer;
    begin
      i:=0;p:=0;
      for x:=1 to n do if l[x].k=0 then require(x)
      { 对每一个未访问的顶点,求以它为树根的强连通分支 }
    end;

  begin
```

```
  read_graph;{ 输入图 }
  proceed     { 计算输出所有强连通分支 }
end.
```

4.5 求最小点基

一、基本概念

还是把 4.4 节求极大强连通子图中的图 4-11(*a*)看作是某篮球队的通讯图。现在，我们对这张图提两个问题：

1. 教练想要通知全体队员都来练球，请你帮教练考虑一下，他至少要通知几个队员（然后由这些队员转告其他队员），才能使所有队员都被通知到；

2. 教练通知队员 U_i 时必须付出一定的代价 A_i（例如 A_i 可以代表教练给 V_i 打电话所需的时间），请你再帮教练考虑一下，如何以最小的代价使所有队员都得到通知。

图 4-12 给出了一种方案：教练直接通知 V_1，V_6，V_7，V_{11}，然后由他们转告其他队员，全部队员就都能接到通知了。但是，是不是至少通知 4 个队员呢？能不能再少通知几个呢？

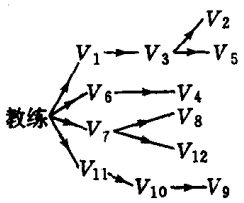


图 4-12

大家试一下就会知道，通知 3 个人就够了，例如可以只通知 V_1，V_7 和 V_{11}，而再少就不可能了（想想看，为什么）。

上面这个例子很简单，通过简单的分析、试验就可以知道至少要通知三个人。但是如果遇到一个类似的问题，而图中的顶点很多，仅仅靠试验就不行了。必须要有系统的方法。这一节的主要目的就是介绍解决这类问题的一种方法。

首先，让我们把要解决的问题明确一下。在有向图中，如果存在一条从顶点 V_i 到 V_j 的有向路，就称 V_i 是 V_j 的前代，V_j 是 V_i 的后代。我们从这个概念出发考虑问题。因为如果教练通知了 V_i，那么所有 V_i 的后代 V_j 就都可以间接得到通知了，而要保证所有队员都得到通知，由教练直接通知的队员集合 B 必须具备下述性质："对于任意一个队员 V_j，一定存在 B 中的一个 V_i，使得 V_i 是 V_j 的前代"。或者说，B 的所有后代包括了 G 的所有顶点。这个 B 集合就是所谓的点基：

设 $G=(V,E)$ 是一个有向图，B 是若干个顶点组成的 V 的子集。如果对于任意的 $V_j \in V$，都存在一个 $V_i \in B$，使得 V_i 是 V_j 的前代，则称 B 是一个点基。

例如按刚才讲的两个方案 $B_1=\{V_1,V_6,V_7,V_{11}\}$ 及 $B_2=\{V_1,V_7,V_{11}\}$ 都是点基。

前面讲的两个问题，实际上是求两个特殊要求的点基。问题 1 可以归结为求一个包含顶点数最少的点基（即最小点基），而问题 2 又可以归结为另一类点基：

设图 G 的每一顶点 V_i 都对应一个非负数 A_i（即 V_i 的权），现在要求一个点基，使得它所包含的顶点对应的 A_i 之和最小（即权最小的点基）。

注意，如果令每一个顶点 V_i 的权 A_i 都等于 1，那么权最小的点基就是最小点基。因此权最小点基是最小点基的一个特例。

那么如何求上述两个点基问题呢？在讲算法之前，我们还要介绍一个概念：

设[S_i]是有向图 G 的一个强连通分支，如果在 G 中，不存在终点属于[S_i]而起点不属于[S_i]的弧，就称[S_i]为最高的强连通分支。

用这个定义判断篮球队通讯图的 8 个强连通分支(图 4-11(b))，[S_1]、[S_4]和[S_8]是最高的，而其它 5 个都不是最高的(见图 4-13)。

在什么意义上讲，具有上述特征的强连通分支[S_i]是最高的呢？由最高强连通分支的定义可以看出，该分支与图 G 其它部分相连的所有弧是向外伸展的，即 V_j 属于[S_i]而 V_k 不属于[S_i]，那末 V_j 不会是 V_k 的后代。针对教练下达通知的问题来说，如果 V_j 是一个属于最高强连通分支[S_i]的队员，那么任何不属于[S_i]的队员 V_k 是无法通知到他的。因此教练至少要直接通知[S_i]中的一个队员。

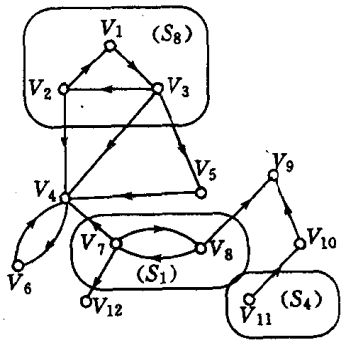


图 4-13

这一下就把求最小点基的算法由来说得再清楚不过了。如果顶点集合 B 是一个点基，那么每个最高强连通分支的[S_i]至少有一个顶点要属于 B。最小点基 B 中的顶点个数即为图 G 中最高强连通分支的分支个数。从每个最高强连通分支中任取一个顶点就可以组成这个最小点基 B。

至于求权最小点基的算法，从表面上看似乎比求最小点基复杂，其实不然，这两个问题的复杂程度是差不多的。要求一个权最小的点基，也需要进行 3 个步骤：

步骤 1　找出图 $G=(V,A)$的所有强连通分支；

步骤 2　从强连通分支中找出所有最高的强连通分支；

步骤 3　从每一个最高的强连通分支中取一个权最小的顶点，组成的顶点集 B 就是一个 G 的权最小的点基。

其中步骤 1,2 和求最小点基一样，而步骤 3 中，两个点基问题“从每一个最高强连通分支中取一个顶点”的方式不一样，求最小点基是“任取”，权最小点基是要求取的顶点权最小。

二、求最小点基的算法

求最小点基问题必须求极大强连通子图。4.4 节中给出了 dfs 算法求解的过程。求极大强连通子图在图论中有很大应用价值。为了便于读者今后更好地开发这一算法，我们在这里给出第二种解法。在求最小点基程序中，我们使用新方法求解。

从极大强连通分支的概念出发，很容易得出一种解法：

首先任取一个顶点 V_i，然后把所有与 V_i 互相可达的顶点 V_j 都找出来，这些顶点的集合 S_1(注意 V_i 也属于 S_1)生成的子图[S_1](就是 S_1 和所有起点和终点都属于 S_1 的弧组成的那个子图)就是包含 V_i 的强连通分支，然后再取一个不属于 S_1 的顶点 V_j，再求出与 V_j 互相可达的顶点集合 S_2，再生成一个强连通分支[S_2]，接下去再取一个既不属于 S_1 又不属于 S_2 的顶点 V_k……，最后就可以把所有强连通分支都求出来了，图的每一个顶点都属于一个强连通分支。

那么，如何从一个不属于目前任何强连通分支的顶点 V_i 出发，扩展出下一个强连通分支[S_k]呢？

如果存在一个顶点 V_j 与 V_i 互相可达，那么 V_j 即是 V_i 的后代(因为 V_i 可达 V_j)，又是 V_i 的前代(因为 V_j 可达 V_i)。因此，要求所有与 V_i 互相可达的顶点集合 S_k，只要先求出 V_i 的所有后代的集合 R，再求出 V_i 的所有前代的集合 P。然后，找出所有既属于 R 又属于 P 的顶点，这些顶点就组成了集合 S_k。

为了求出 V_i 所有后代的集合 R(所有前代的集合 P)，也可以采用一种标号法，在这种标号法中，一旦能确定一个顶点是 V_i 的后代(前代)，就可给它一个标号+(−)。具体计算时，首先给 V_i 以标号+(−)，这是因为 V_i 可达自己，因此 V_i 本身也是 V_i 的后代(前代)，然后逐步扩大已标号点的范围。办法是：如果发现一个顶点 V_k 已标号，说明它是 V_i 的后代(前代)，即存在一条从 V_i 到 V_k(从 V_k 到 V_i)的有向路。而与 V_k 相连接的是一条以 V_k 为起点的弧$<k,j>$(以 V_k 为终点的弧$<j,k>$)，这时如果 V_j 还没有得到标号，就可以给 V_j 以标号+(−)。因为从 V_i 到 V_k 的有向路(从 V_k 到 V_i 的有向路)上接上弧$<k,j>$($<j,k>$)就是一条 V_i 到 V_j 的有向路(V_j 到 V_i 的有向路)。因此 V_j 也是 V_i 的后代(前代)。这样不断地扩大已标号顶点的范围，直至无法扩大为止，这时所有已标号的顶点就恰好是 V_i 的后代集合 R(前代集合 P)了。

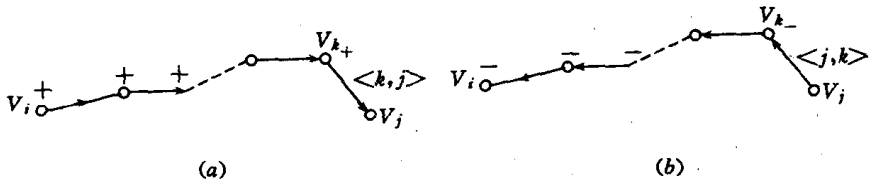


图 4-14

当一个图中顶点和弧很多时(见图 4-14)，为了使标号过程有条不紊，从而避免重复，通常在考虑一个有标号的顶点 V_k 时，应该把所有以 V_k 为尾的弧$<k,j>$(V_k 为头的弧$<j,k>$)都考察一遍。如果$<k,j>$的终点 V_j($<j,k>$的起点 V_j)还没有标号，就给它标上号。考察完以后，就可以认为顶点 V_k 已经“检查”过了，今后就不必再考虑它了，因为它再也不会使别的顶点得到标号了。接下来再取一个已标号而没有“检查”过的顶点来考虑……，如果发现所有标号的顶点都已经被检查过了，那么很显然，已经不可能再扩大已标号点的范围了。

在讲解标号法的过程中，( )里的内容对应于求前代集合 P。

把上面讲的总结起来，就可以得到下面的求顶点 V_i 的所有后代的集合 R 和所有前代的集合 P 的计算方法了：

1. 求 V_i 所有后代的集合 R

步骤 1. 给 V_i 以标号+，V_i 是已标号而未检查的顶点；

步骤 2. 看看是否有已标号而未检查的顶点，如果没有，计算结束，所有已标号的顶点组成集合 R。如果有，任取一个这样的顶点 V_k，转入步骤 3；

步骤 3. 找出以 V_k 为起点的所有弧$<k,j>$，一条一条地考虑这些弧，如果$<k,j>$的

终点 V_j 没有标号，就给 V_j 以标号＋，V_j 成为已标号而未检查的顶点。这些弧考虑过以后，V_k 成为已检查过的，作一个检查标志，转回步骤 2。

2. 求顶点 V_i 的所有前代的集合 P

采用与求 R 集合完全相似的算法。步骤 1 和 2 是一样的。唯一的差别是步骤 3。本来在取定了 V_k 以后，考虑的是以 V_k 为起点的弧$<k,j>$，现在则应该改为考虑以 V_k 为终点弧$<j,k>$，如果这条弧的起点 V_j 还没有标号，就应该给它以标号。另外，为了在一张图上同时求 V_i 的所有前代与后代，那么在求前代时，可以用'－'作为标号。

很显然，在求出集合 R 和 P 以后，既标有＋又标有－的顶点就是与 V_i 互相可达的顶点了，这些顶点组成了一个强连通分支的顶点集合 S_k。

三、求最小点基的程序

```
program dian_ji;
  uses crt;
  const maxn = 100;
  type  list      = array[1..maxn] of boolean;
  var   tu        : array[1..maxn] of list;  { 图的邻接矩阵 }
        s         : list;          { 记录顶点 i 是否已属于某强连通分支 }
        n         : integer;       { 顶点数 }
  procedure init;
    var i,j,e,k : integer;
      begin
        clrscr;
        repeat write('n='); { 输入顶点数 }
               readln(n);
        until (n>0) and (n<=maxn);
        for i:=1 to n do  { 邻接矩阵初始化 }
          for j:=1 to n do
            tu[i,j]:=false;
        write('e=');
        readln(e);
        for k:=1 to e do { 读入邻接矩阵 }
          begin
            read(i);
            readln(j);
            tu[i,j]:=true;
          end;
      end;
  procedure make_s_i(x:integer;var v_i:list); { 构造顶点 x 所属的强连通分支 v_i }
    var  shun,                          { x 的所有后代的集合 }
         ni,                            { x 的所有前代的集合 }
         b          : list;             { 记录当前顶点是否检查过 }
         more       : boolean;          { 前代(后代)集合未产生标志 }
         i,j        : integer;
    begin
      for i:=1 to n do { 后代集合，前代集合以及检查标志初始化 }
        begin
```

```
        shun[i]:=false;
        ni[i]:=false;
        b[i]:=false;
      end;
    shun[x]:=true; ni[x]:=true; {设 x 在所属的强连通分支上}
    repeat more:=false; { 求出 x 的所有后代集合,这些后代设检查标志 }
        for i:=1 to n do
          if (shun[i]) and (not b[i])
          then begin
                 for j:=1 to n do
                   if (tu[i,j]) and (not shun[j])
                   then shun[j]:=true;
                 more:=true;
                 b[i]:=true;
               end;
    until not more;
    for i:=1 to n do { 检查标志初始化 }
      b[i]:=false;
    repeat more:=false; { 求出 x 的所有前代集合,这些前代设检查标志 }
        for i:=1 to n do
          if ni[i] and (not b[i])
          then begin
                 for j:=1 to n do
                   if (tu[j,i]) and (not ni[j])
                     then ni[j]:=true;
                   more:=true;
                   b[i]:=true;
               end;
    until not more;
    for i:=1 to n do { 求 x 所属的强连通分支上的所有顶点 }
      v_i[i]:=shun[i] and ni[i];
  end;
function highest(s_i:list):boolean;
  { 搜索所有弧,若不存在终点属于 s_i 而起点不属于 s_i 的弧,      }
  { 则强连通分支 s_i 是最高强连通分支,返回 true;否则返回 false }
  var i,j : integer;
    begin
      highest:=false;
      for i:=1 to n do
        for j:=1 to n do
          if (tu[i,j]) and (not s_i[i]) and (s_i[j])
          then exit;
      highest:=true;
    end;
procedure main;
  var s_i : list;          { 当前顶点 i 属于的强连通分支 }
    i,j : integer;
    begin
      for i:=1 to n do { 设所有的顶点不在强连通分支上 }
```

```
        s[i]:=false;
      for i:=1 to n do
        if not s[i] then { 顶点i目前不属于任何强连通分支 }
          begin
            make_s_i(i,s_i);     { 构造顶点i所属的强连通分支s_i }
            if highest(s_i)              { 若s_i是最高强连通分支 }
              then writeln('take ',i); { 则i属于最小点基,输出i }
            for j:=1 to n do
              s[j]:=s[j] or s_i[j];
            { s_i归入已生成的强连通分支 }

          end;
    end;
begin
  init; { 输入图 }
  main; { 计算和输出最小点基 }
end.
```

4.6 可靠通讯网的构作

一、问题及其分析

构作一个具有 n 个通讯站的有线通讯网，使得敌人至少炸坏我 m 个($m<n$)通讯站后，才能中断其余的通讯站的彼此通话。显然有两个要求是必要的：一是不怕被敌人炸坏站的数目要多，二是整个造价要低。这个实际问题的数学模型如下：

G 是加权连通无向图，m 是给定的自然数，求 G 的最小权的 m 连通生成子图。当 $m=1$ 时，它就是 3.1 节所讲的求无向图的最小生成树；当 $m>1$ 时是尚未解决的难题之一。

我们将原来的问题作两条限制后就可以解决：

1. G 是完全图；

2. 每边的权皆为 1。

因为当每条通讯线(无论其长短)都造价一样时，要满足第 1 个要求，最理想的情况无非是原来每两个通讯站之间皆能彼此通话。但这样做造价太高，必须在满足第 1 个要求的前提下尽可能多地减小通讯线数，以达到两个要求同时满足。换句图论的话说，当 G 为 n 个顶点的完全图且每边权皆为 1 时，求一个 G 的边数最小的 m 连通子图。

令 $f(m,n)$表示 n 个顶点的 m 连通子图中边数的最小值($m<n$)。由 G 的 n 个顶点的次数和$=2\times G$ 的边数，$K\leqslant K'\leqslant G$ 的顶点度数的最小值(K 和 K'为 G 的连通度和边连通度)，得出：

$$f(m,n)\geqslant\{m\ n/2\}$$

由上可见，构作一个由最少边数连接 n 个顶点的 m 连通图。这个图的边数恰为$\{mn/2\}$，此图记成 $H_{m,n}$。

我们根据 m 和 n 的奇偶性作图。

1. 当 m 是偶数($m=2r$)

$H_{2r,n}$以$\{0,1,\cdots,n-1\}$为顶点集合。当 $i-r\leqslant j\leqslant i+r$ 时，在顶点 i 与 j 之间连一条

边。这里加法在 mod n 意义下进行。例如 4-15(a)是构作的 $H_{4,8}$图。

2. 当 m 是奇数($m=2r+1$),先构作 $H_{2r,n}$图,然后再根据 n 的奇偶性添边:

n 是偶数:

对 $1\leqslant i\leqslant n/2$ 的 i,在 i 与 $i+n/2$ 间加上一条边得 $H_{2r+1,n}$。例如 4-15(b)是构作的 $H_{5,8}$图。

n 是奇数:

在顶点 0 与顶点$(n-1)/2$、顶点 0 与顶点$(n+1)/2$ 之间加上两条边,在顶点 i 与顶点 $i+(n+1)/2$ 间加上边,其中 $1\leqslant i\leqslant(n-1)/2$,则得到 $H_{2r+1,n}$。例如 4-15(c)是构作的 $H_{5,9}$图。

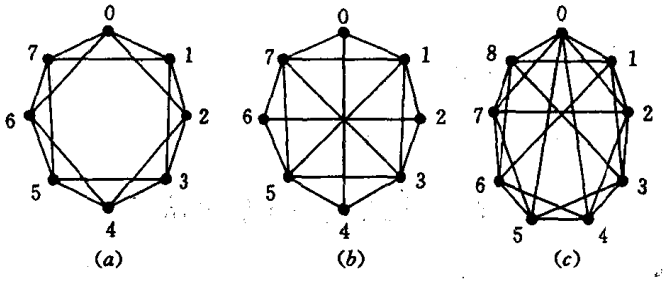


图 4-15

二、求 $H_{m,n}$图的程序

下面是求 $H_{m,n}$图的程序。

```
program harary_p53;


  var
    m,n,i,j      :   integer;

  procedure init;
    begin.
      repeat
        write('m n = ');readln(m,n) { 读入连通数和顶点数 }
      until (m<n)
    end;

  function change(a,n : integer) : integer; { 返回 a 除以 n 的余数 }
    begin
      if a<0
         then change := (n+a) mod n { 逆时针取余 }
         else change :=a mod n      { 顺时针取余 }
    end;

  procedure A(r,n : integer);
```

```
{ 以{0..n-1}为顶点集合,当 i-r≤j(在 mod n 定义下进行)≤i+r 时 }
{ 在顶点 i 与 j 之间连一条边 }
  begin
    for i:=0 to n-1 do
      for j:=i-r to i+r do
        if change(j,n)>i then writeln(i:2,'——',change(j,n):2)
  end;

procedure pp;
  begin
    if not odd(m) { m 是偶数,构作 H2r,n }
       then A(m div 2,n)
       else if not odd(n) { m 是奇数,n 是偶数 }
                then begin
                       A((m-1) div 2,n); { 先构作 H2r,n }
                       for i:=1 to n div 2 do writeln(i:2,'——',i+n div 2);
                       { 在 i 与 i+n/2 间加一条边(1≤i≤n/2),得 H2r+1,n }
                     end
                else begin { m 是奇数,n 是奇数 }
                       A((m-1) div 2,n); { 先构作 H2r,n }
              { 在顶点 0 与顶点(n-1)/2 之间,顶点 0 与顶点(n+1)/2 之间加两边 }
                       writeln(0:2,'——',(n-1) div 2:2);
                       writeln(0:2,'——',(n+1) div 2:2);
                       for i:=1 to (n-1) div 2-1 do
                       { 连接顶点 i 与顶点 i+(n+1)/2(1≤i≤(n-1)/2),得 H2r+1,n }
                         writeln(i:2,'——',i+(n+1) div 2);
                     end
  end;

begin
  init; { 读入连通数 m 和顶点数 n }
  pp    { 构作可靠通讯网并输出 }
end.
```

第五章　支配集与独立集

5.1　求支配集

一、问题及其分析

要在 $V_1,V_2,\cdots,V_n$ 这 N 个城镇建立一个通迅系统，为此从这 n 个城镇中选定几座城镇，在那里建立中心台站，要求它们与其它各城镇相邻，同时为降低造价，要使中心台站数目最少，有时还会提其它要求。例如在造价最低的条件下，需要造两套(或更多套)通讯中心，以备一套出故障时启用另一套。

例如图 5-1 看作是一个通讯系统。$V_1,V_2,\cdots,V_6$ 是一些城镇，仅当两城之间有直通通讯线时，相应的两顶点连一条边。在讲这个问题的数学模型之前，先讲一个定义：

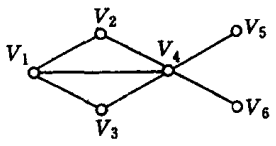


图　5-1

D 是图 G 的一个顶点子集，对于 G 的任一顶点 U，要么 U 是 D 集合的一个顶点元素，要么与 D 中的一个顶点相邻，那么 D 称为图 G 的一个支配集。若在 D 集中去除任何元素后 D 不再是支配集，则支配集 D 是极小支配集。称 G 的所有支配集中顶点个数最少的支配集为最小支配集 D_0，记 $\gamma(G)=D_0$ 中的顶点个数，称作 G 的支配数。

由上述定义可知，凡最小支配集一定是极小支配集；任何一个支配集以一个极小支配集为其子集；G 所含的极小支配集可能有两个以上，而且其顶点数也可以不一致，但支配数 $\gamma(G)$ 是唯一的。

显然中心台站的选址问题，实际上是求一个能与 G 中其它顶点相邻的顶点集合，且这个顶点集合所含的顶点元素必须最少，即求图的最小支配集。若建两套，则从一切极小支配集 $D_1,D_2,\cdots,D_d$ 中选取 D_m 和 D_n，使得 $D_m\cap D_n=\varphi$(φ 为支配数，愈小愈好)，即其中一套出故障时不影响另一套工作。并且 $|D_m\cup D_n|=\min\{|D_i|+|D_j|\,|\,1\leqslant i,j\leqslant d,D_i\cap D_j=\varphi\}$，即两套互不干扰的通讯中心所选定的城镇数最少。

例如在图 5-1 所示的 6 个城镇中建中心台站，方案有 $\{V_1,V_5\},\{V_1,V_6\},\{V_4\},\{V_2,V_3,V_5\},\{V_2,V_3,V_6\}$。这些顶点集合为极小支配集。若建一套通讯中心，只需建立在 V_4 城；若建两套，则 V_4 建一套，$\{V_1,V_5\}$ 或者 $\{V_1,V_6\}$ 建第 2 套。

有许多实际问题可以转化成上述这种数学模型来处理。

支配集有哪些性质呢？

1. 若 G 中无零次顶点($d(v)=0$)，则存在一个支配集 D，使得 G 中除 D 外的所有顶点也组成一个支配集；

2. 若 G 中无零次顶点($d(v)=0$)，D_1 为极小支配集，则 G 中除 D_1 外的所有顶点组成一个支配集。

求所有极小支配集的计算，包括 5.2 节中的求所有极小覆盖集的计算都是采用逻辑

运算的。

设 X,Y,Z 三条指令：

规定："要么执行 X，要么执行 Y" 记作 $X+Y$（和运算）

"X 与 Y 同时执行" 记作 XY （与运算）

上述逻辑运算有以下运算定律：

1. 交换律

$X+Y=Y+X$；$XY=YX$

2. 结合律

$(X+Y)+Z=X+(Y+Z)$；$(XY)Z=X(YZ)$

3. 分配律

$X(Y+Z)=XY+XZ$；$(Y+Z)X=XY+XZ$

4. 吸收律

$X+X=X$；$XX=X$；$X+XY=X$

上述定律尤其是吸收律，在求所有极小支配集和极小覆盖集的两个公式中用处颇大。求所有极小支配集的公式：

$$\varphi(V_1,V_2,\cdots,V_n)=\prod_{i=1}^{n}\left(V_i+\sum_{U\in N(V_i)}U\right)$$

一个顶点同与它相邻的所有顶点进行加法运算组成一个因子项，几个因子项再连乘。连乘过程中根据上述运算规律展开成积之和形式。每一积项给出一个极小支配集，所有积项给出了一切极小支配集，其中最小者为最小支配集。

例如求图 5-1 通讯网的一切极小支配集及支配数。

$$\begin{aligned}&\varphi(V_1,V_2,V_3,V_4,V_5,V_6)\\&=(V_1+V_2+V_3+V_4)(V_2+V_1+V_4)(V_3+V_1+V_4)(V_4+V_1+V_2+V_3+V_5+V_6)\\&\quad(V_5+V_4+V_6)(V_6+V_4+V_5)\\&=(1+2+3+4)(2+1+4)(3+1+4)(4+1+2+3+5+6)(5+4+6)(6+4+5)\\&=15+16+4+235+236\end{aligned}$$

故得所有极小支配集如下：

$\{V_1,V_5\},\{V_1,V_6\},\{V_4\},\{V_2,V_3,V_5\},\{V_2,V_3,V_6\}$

$\gamma(G)=1$

二、求极小支配集和支配数的程序

下面根据公式和运算定律，给出求所有极小支配集和支配数的程序。

```
Program ji_xiao_zhi_pei_ji;

const
  maxn        =  30;

type
  ghtype      =  array [1..maxn,1..maxn] of integer;
```

```
  settype      =  set of 1..maxn;
  ltype        =  array [1..maxn] of settype;

var
  g            :  ghtype;     { 邻接矩阵 }
  n,l          :  integer;    { 顶点数,极小支配集的个数 }
  f            :  text;       { 文件变量 }
  lt           :  ltype;      { 极小支配集 }

procedure read_graph;
  var
    str : string;
    i,j : integer;
  begin
    write('Graph file = '); { 输入文件名,并与文件变量连接 }
    readln(str);
    assign(f,str);
    reset(f);
    readln(f,n);          { 输入顶点数 }
    for i := 1 to n do
      for j := 1 to n do read(f,g[i,j]); { 输入图的邻接矩阵 }
    close(f);
  end;

procedure reduce(s : settype);
{ 根据吸收律求 s+lt[1]+lt[2]+…+lt[l]的结果 }
  var
    i,j : integer;
  begin
    i := 1;
    while i<=l do { 检查所有支配集 }
      begin
        if s * lt[i]=lt[i] then exit;
        { 若第 i 个支配集是 s 的子集,则退出过程 }
        if s * lt[i]=s
        { 若 s 是第 i 个支配集的子集,则删除该极小支配集 }
          then begin
                 for j := i+1 to l do lt[j-1] := lt[j];
                 dec(l)
               end
          else inc(i)
        { 否则检查下一个支配集,即删除所有含 s 子集的支配集 }
        { 直至某支配集作为 s 的子集或所有支配集与 s 非子集关系为止 }
      end;
    inc(l);lt[l] := s { s 作为最后一个支配集 }
  end;

procedure think;
  var
```

```
    tl,i,j,k : integer;
    t : ltype;
  begin
    l : =0;
    for i : =1 to n do if (i=1) or (g[1,i]>0) then reduce([i]);
    { 建立 v1+ ∑ u,各项存入 lt }
      u∈N(v1)
    for i : =2 to n do
      begin
        t : =lt;tl : =l; { 暂存所有支配集 lt 和支配集个数 }
        l : =0;
        for j : =1 to n do
        { 求(v2+ ∑ u)(v3+ ∑ u)…(vi+ ∑ u)的所有乘积项 lt[1]…lt[l] }
            u∈N(v2)   u∈N(v3)      u∈N(vi)
          if (i=j) or (g[i,j]>0)
            then for k : =1 to tl do
                   reduce(t[k]+[j])
      end;
    lt : =t;l : =tl { 求出 L 个极小支配集 lt }
  end;

procedure print; { 打印 l 个极小支配集中的所有元素 }
  var
    i,j : integer;
  begin
    for i : =1 to l do
      begin
        write(i : 3);
        for j : =1 to n do if j in lt[i] then write(j : 3);
        writeln
      end
  end;

begin
  read_graph;  { 输入图 }
  think;       { 计算极小支配集 }
  print        { 输出结果 }
end.
```

5.2 求独立集

一、问题及其分析

$S=\{S_1,S_2,\cdots,S_n\}$为信息传送的基本信号集合。

可以把信号S_i设想成汉字或拉丁字母。统计规律表明哪些信号与哪些信号易于发生错乱。例如输入S_i,输出应该是S_i^*(1≤i≤5)。但由于干扰发生了错乱,S_1可能和S_2错乱,S_1还可能和S_5错乱等。例如已知错乱可能性如图 5-2(a)所示。

为了确切地从输出信号得知输入信号,我们不能选用$S_1,S_2,\cdots,S_n$中的每一信号,只

能从中选一部分用于输出。那么选哪一部分信号作输入源呢？我们作另外一张图，顶点表示信号，若信号源 S_i 可能输出 S_j，则在 S_i 与 S_j 之间连一条边，如图 5-2(b)。为了使可用于输出的信号最多，我们在图 5-2(b)上求这样一个顶点集合 I，I 中任两个顶点不相邻且这些独立的顶点数最多。例如可以选$\{S_1, S_3\}$作为输出信号；或选$\{S_1, S_4\}$或$\{S_2, S_4\}$或$\{S_2, S_5\}$或$\{S_3, S_5\}$做输出信号。这个数学模型就是所谓的求最大独立集问题。

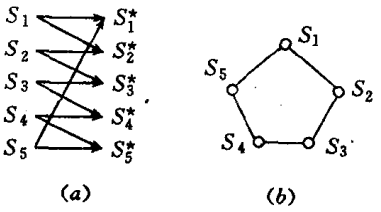


图 5-2

在实际生活中，求最大独立集的应用实例很多，如 8 皇后问题。我们将棋盘的每个方格看作一个顶点，放置的皇后所在格与她能攻击的格看成是有边连接，从而组成 64 个顶点的图。若要设计一个算法，放置尽可能少的皇后，使她们控制棋盘上全部格，这显然是求最小支配集问题；再设计一个算法，要放置尽可能多的皇后，而相互不攻击地共处，这个问题就是求最大独立集问题了。

在讲求最大独立集的定义之前，我们先引出与独立集密切相关的覆盖集的概念：K 是 G 图的一个顶点子集且 G 的每一边至少有一个端点属于 K，则称 K 是 G 的一个覆盖。这里覆盖一词的含义是顶点覆盖全体边。若在 K 集中去除任一顶点后将不再是覆盖，则称 K 为极小覆盖。在图 G 的所有极小覆盖集中含顶点数最少的极小覆盖称作最小覆盖。用 $\alpha(G)$表示最小覆盖的顶点数，又称 G 的覆盖数。

例如图 5-3 中的黑色顶点是一个极小覆盖，同时也是一个最小覆盖，$\alpha(G)=4$，$K=\{V_1, V_3, V_4, V_6\}$。下面给出独立集的概念：

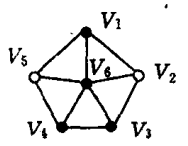


图 5-3

I 是 G 的一个顶点子集，I 中任两个顶点不相邻，则称 I 是图 G 的一个独立集。若独立集 I 中增加任一个除 I 集外的顶点后 I 不是独立集，则称 I 是极大独立集。在图 G 的所有极大独立集中含顶点数最多的极大独立集称作最大独立集。用 $\beta(G)$表示最大独立集的顶点数，又称 G 的独立数。

那么独立集又有什么性质呢？我们从独立集、覆盖集、支配集之间关系中论述：

1. 一个独立集是极大独立集，当且仅当它是一个支配集；

2. I 是独立集，当且仅当 G 中除 I 集外的所有顶点是一个覆盖集；

I 是极大(最大)独立集，当且仅当 G 中除 I 集外的所有顶点是一个极小(最小)覆盖集，即

$$\alpha(G) + \beta(G) = G \text{ 的顶点数}$$

3. 极大独立集必为极小支配集；但是极小支配集未必是极大独立集。

例如一个含边数为 4 的圈上的两个相邻顶点是极小支配集，但不能为极大独立集，连独立集也不是。

例如图 5-3 中$\{V_1, V_3, V_4, V_6\}$是一个最小覆盖集，$V-K=\{V_2, V_5\}$是一个最大独立集同时又是一个极小支配集。注意图 5-3 中的最小支配集是$\{V_6\}$不是独立集。

由于极大独立集与极小覆盖集有互补性，我们这里仅给出求所有极小覆盖集的计算

公式，读者可以由此推出极大独立集：

$$\varphi(V_1,V_2,\cdots,V_n)=\prod_{i=1}^{n}\left(V_i+\prod_{U\in N(V_i)}U\right)$$

某顶点的所有相邻顶点进行积运算后再与该顶点进行和运算，组成一个因子项。几个因子项连乘，并根据逻辑运算定律展开成积之和形式。每一积项给出一个极小覆盖集，所有积项给出了一切极小覆盖集，其中最小者为最小覆盖集。

例如求图 5-4 中的一切极小覆盖集和覆盖数 $\alpha(G)$ 以及一切极大独立集和独立数 $\beta(G)$。

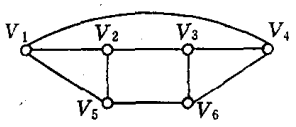


图 5-4

$\varphi(V_1,V_2,\cdots,V_6)$

$=(1+246)(2+136)(3+246)(4+135)(5+346)(6+125)$

$=2456+1356+1346+2346+1245+1235$

即得一切极小覆盖集如下

$$\{V_2,V_4,V_5,V_6\},\{V_1,V_3,V_5,V_6\},$$
$$\{V_1,V_3,V_4,V_6\},\{V_2,V_3,V_4,V_6\},$$
$$\{V_1,V_2,V_4,V_5\},\{V_1,V_2,V_3,V_5\}。$$
$$\alpha(G)=4$$

由于 G 的除极小覆盖集外的顶点可组成极大独立集，由此可以得出一切极大独立集为

$$V-\{V_2,V_4,V_5,V_6\}=\{V_1,V_3\};$$
$$V-\{V_1,V_3,V_5,V_6\}=\{V_2,V_4\};$$
$$V-\{V_1,V_3,V_4,V_6\}=\{V_2,V_5\};$$
$$V-\{V_2,V_3,V_4,V_6\}=\{V_1,V_5\};$$
$$V-\{V_1,V_2,V_4,V_5\}=\{V_3,V_6\};$$
$$V-\{V_1,V_2,V_3,V_5\}=\{V_4,V_6\}。$$
$$\beta(G)=2$$

可见求最大独立集，首要的是求极小覆盖集。

二、求所有极小覆盖集的程序

下面我们给出求所有极小覆盖集的程序。

```
program ji_xiao_fu_gai_ji;

const
  maxn        =  30;

type
  ghtype      =  array [1..maxn,1..maxn] of integer;
  settype     =  set of 1..maxn;
  ltype       =  array [1..maxn] of settype;

var
  g           :  ghtype;     { 邻接矩阵 }
```

```
  n,l          :  integer;    { 顶点数,极小覆盖集个数 }
  f            :  text;       { 文件变量 }
  lt           :  ltype;      { 极小覆盖集 lt[i]—第 i 个极小覆盖集 1≤i≤l }
procedure read_graph;
  var
    str : string;
    i,j : integer;
  begin
    write('Graph file = ');{输入文件名并与文件变量连接 }
    readln(str);
    assign(f,str);
    reset(f);
    readln(f,n); { 输入顶点数和图矩阵 }
    for i := 1 to n do
      for j := 1 to n do read(f,g[i,j]);
    close(f);
  end;

procedure reduce(s : settype); { 根据吸收律求 s+lt[1]+…+lt[l]的结果 }
  var
    i,j : integer;
  begin
    i := 1;
    while i<=l do
      begin
        if s * lt[i]=lt[i] then exit;
        if s * lt[i]=s
           then begin
                  for j := i+1 to l do lt[j-1] := lt[j];
                  dec(l)
                end
           else inc(i)
      end;
    inc(l);lt[l] := s
  end;

procedure think;
  var
    tl,i,j,k : integer;
    t : ltype;
    s : settype;
  begin
    lt[1] := [1];lt[2] := [];l := 2;
    for i := 2 to n do if g[1,i]>0 then lt[2] := lt[2]+[i]; { 建立 v1 + ∏ u }
                                                            u∈N(v1)
      for i := 2 to n do { 求 ∏ (vi + ∏ u) }
                            i=1   u∈N(vi)
```

```
      begin
        t := lt; tl := l;
        l := 0; s := [];
        for j := 1 to n do if g[i,j]>0 then s := s+[j];
        { 求所有与顶点 i 相连的顶点集合 s }
        for k := 1 to tl do
          begin
          reduce(t[k]+s); reduce(t[k]+[i])
            { 根据分配律求(t[1]+…+t[l])(s+[i])的所有乘积项 }
          end
      end;
      lt := t; l := tl
  end;

procedure print;
  var
    i,j : integer;
  begin
    for i := 1 to l do { 打印每个乘积项(极小覆盖集)的元素 }
      begin
        write(i : 3);
        for j := 1 to n do if j in lt[i] then write(j : 3);
        writeln
      end
  end;

begin
  read_graph;  { 输入图 }
  think;       { 计算极小覆盖集 }
  print        { 输出结果 }
end.
```