数据结构与算法

Data Structures and Algorithms

第四部分 图

回顾：图--2

1. 图的搜索算法

图的遍历 { 深度优先搜索DFS (Depth-First Search)
广度优先搜索BFS (Breadth-First Search)

2. 图与树的联系

深度优先生成树

广度优先生成树

最小生成树

算法一： **Prim （普里姆算法）**

算法二： **Kruskal （克鲁斯卡尔算法）**

（1）求最小生成树 —— Prim 算法

输入： 加权无向图（无向网）G=(V, E)，其中v=(1,2, …,n).

输出： G的最小生成树

步骤： 引入集合U和T。U为预备顶点集，T为树边集。
初值U={v_0}，T=∅。选择有最小权的边(u,v)，
使u∈U, v∈(V-U), 将v加入U,（u,v）加入T。
重复这一过程，直到 U=V。

```
void  Prim( G, T )
{     T = ∅ ;
      U = { v0 };
      while ( ( V – U )  != ∅ )
        {   设( u, v ) 是使u∈U与v∈(V-U)且权最小的边 ;
            T = T∪{ ( u, v ) } ;
            U = U ∪ { v };
        }
} ;
```

普里姆（Prim）算法如何实现呢？

引入辅助向量：

CloseST[] 和 LowCost，其中：

CloseST[*i*] 为 U 中的一个顶点

边 (*i* , CloseST[*i*]) 具有最小的权 LowCost[*i*]；

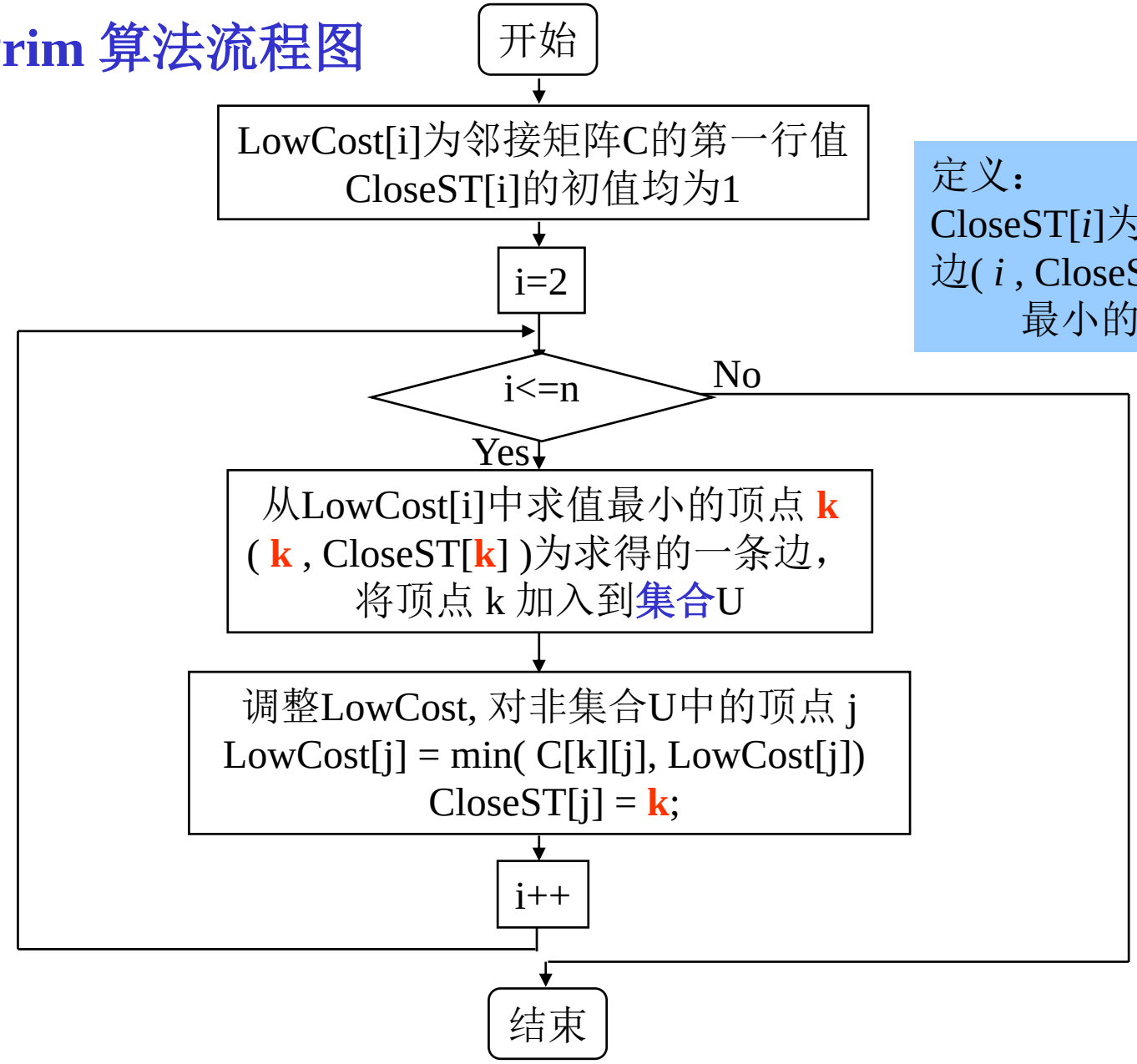
CloseST 和 LowCost 的初值是多少？

CloseST全为1；LowCost为邻接矩阵第一行。

集合如何实现？

若顶点 i ∈ U 则 LowCost[i] = INFINITY
否则 LowCost[i] = 0；

Prim 算法流程图

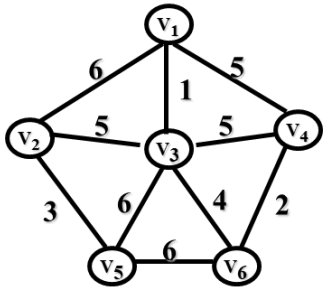


定义：
CloseST[*i*]为U中的一个顶点
边(*i* , CloseST[*i*]) 具有
最小的权 LowCost[*i*]

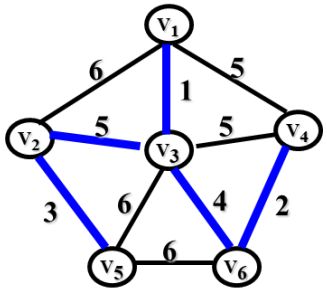
```
Void Prim( C )
Costtype C[n+1][n+1] ;
{ costtype LowCost[n+1];  int CloseST[n+1];  int i,j,k;  costtype min;
  for( i=2; i<=n; i++ )
    {   LowCost[i] = C[1][i];    CloseST[i] = 1;   }   // 赋初值, U中顶点都是1
  for( i = 2; i <= n; i++ )
    {    min = LowCost[i];
         k = i;
         for( j = 2; j <= n; j++ )
           if ( LowCost[j] < min )
             { min = LowCost[j] ;   k=j; } //求离U中某一顶点最近的顶点k
         Cout << "(" << k << "," << CloseST[k] << ")" << end1;
         LowCost[k] = INFINITY ;     //将k加入集合U
         for ( j = 2; j <= n; j++ )
             if ( C[k][j] < LowCost[j] && LowCost[j] != INFINITY )
               {   LowCost[j]=C[k][j];  CloseST[j]=k;   }  //调整
    }
}
```

CloseST[*i*]为U中的一个顶点
边(*i* , CloseST[*i*])具有最小的权LowCost[*i*]

【例4-3】



C	1	2	3	4	5	6
1	∞	6	1	5	∞	∞
2	6	∞	5	∞	3	∞
3	1	5	∞	5	6	4
4	5	∞	5	∞	∞	2
5	∞	3	6	∞	∞	6
6	∞	∞	4	2	6	∞



CloseST[*i*]为U中的一个顶点

边(*i* , CloseST[*i*])具有最小的权LowCost[*i*]

K=?
min(LowCost[])

LowCost[i]

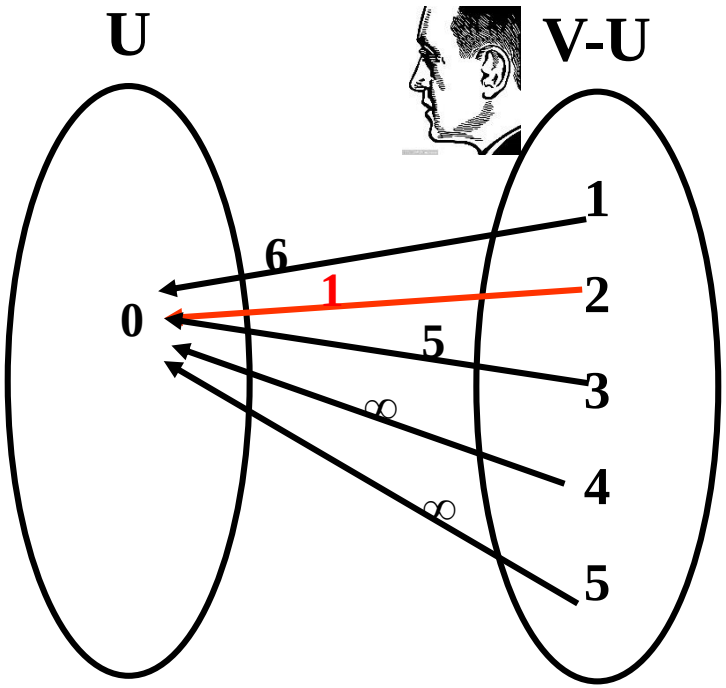
i=	1	2	3	4	5	6
1	--	6	1	5	∞	∞
2		5	∞	5	6	4
3		5	∞	2	6	∞
4		5	∞	∞	6	∞
5		∞	∞	∞	3	∞
6		∞	∞	∞	∞	∞

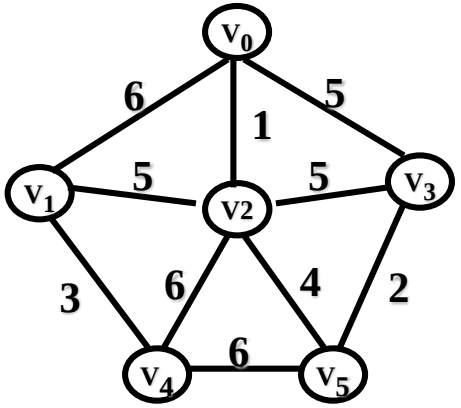
CloseST[i]

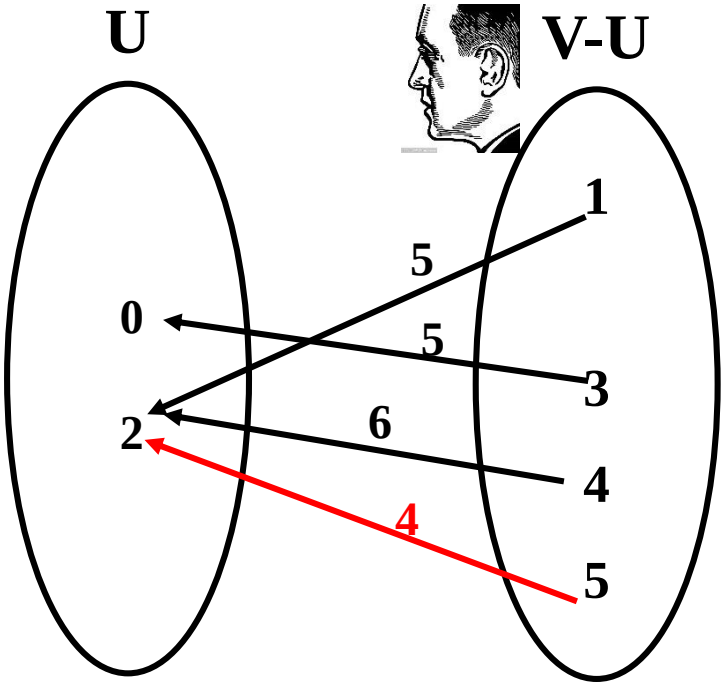
i=	1	2	3	4	5	6
1	--	1	1	1	1	1
2		3	1	1	3	3
3		3	1	6	3	3
4		3	1	6	3	3
5		3	1	6	2	3
6		3	1	6	2	3

打印边

	k	(k, CloseST[k])
→	3	(3, 1)
→	6	(6, 3)
→	4	(4, 6)
→	2	(2, 3)
→	5	(5, 2)


U
V-U
0
1
2
3
4
5
6
1
5
∞
∞


V0
V1
V2
V3
V4
V5
6
5
1
5
5
3
6
4
2
6


U
V-U
0
2
1
3
4
5
5
5
6
4

（2）求最小生成树 —— Kruskal 算法

问题出发点：为使生成树上边的权值之和达到最小，则应使生成树中每一条边的权值尽可能地小。

算法要点：

令 T = (V , E), (V=1,2,3,…,n), c是关于E中每条边的权函数

(1) T中每个顶点自身构成一个连通分量；

(2) 按照边的权不减的顺序，依次考查E中的每条边；//边进行排序

(3) 如果被考查的边连接不同的分量中的两个顶点，则合并两个分量；

(4) 如果被考查的边连接同一个分量中的顶点，则放弃，避免环路；

(5) T中连通分量逐渐减少；

当T中的连通分量的个数为1时，说明V中的全部顶点通过E中权最小的那些边，构成了一个没有环路的连通图T，即为最小生成树。

克鲁斯卡尔Kruskal算法的基本思想：

设G=(V,E)是连通网，用T来记录G上最小生成树边的集合。

(1) 对所有的边上权值进行一次从小到大的排序。

(2) 从G中取权值最小的边e：

--- 如果边e所关联的两个顶点不在T的同一个连通分量中，则将该边作为最小生成树的边加入T；

--- 如果边e所关联的两个顶点属于同一个连通分量，则舍弃此边，以免造成回路；

(3) 从G中删除边e；

(4) 重复(2)和(3)两个步骤，直到T中有n-1条边。

输入：连通图G=(V, E)，其中v=(1,2, …,n)，C是关于E中的每条弧的权。

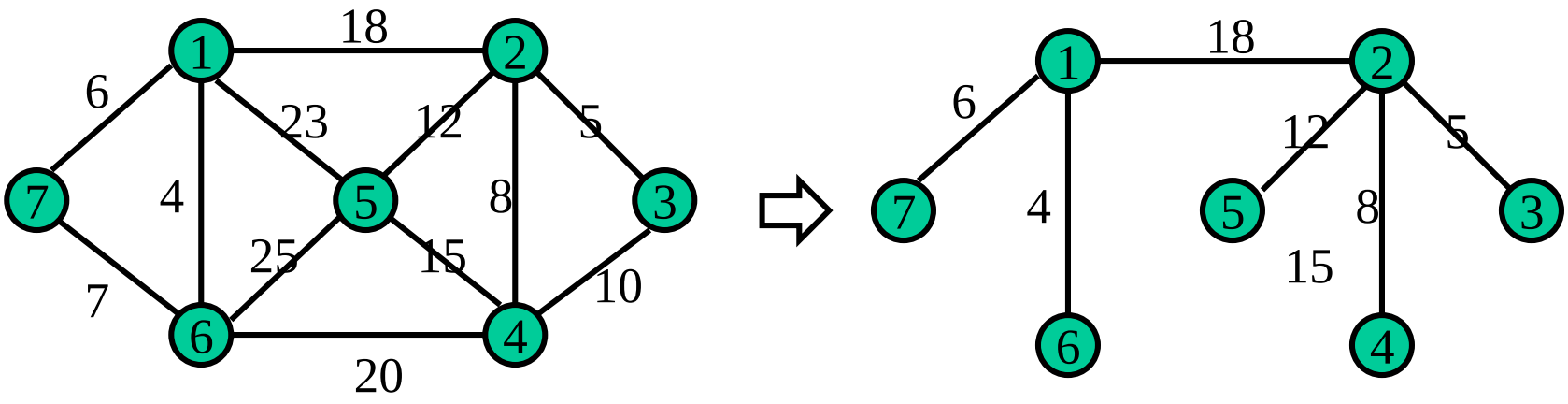
输出：G的最小生成树

```
Void   Kruskal ( V, T )
{    T = V ;
     ncomp = n ;    /*图中总结点个数*/
     while ( ncomp > 1 )
           {    从E中取出删除权最小的边（v, u） ;
                if ( v 和 u 属于T中不同的连通分量）
                  {   T = T∪{ ( v, u ) }
                      ncomp -- ;
                  }
           }
}
```

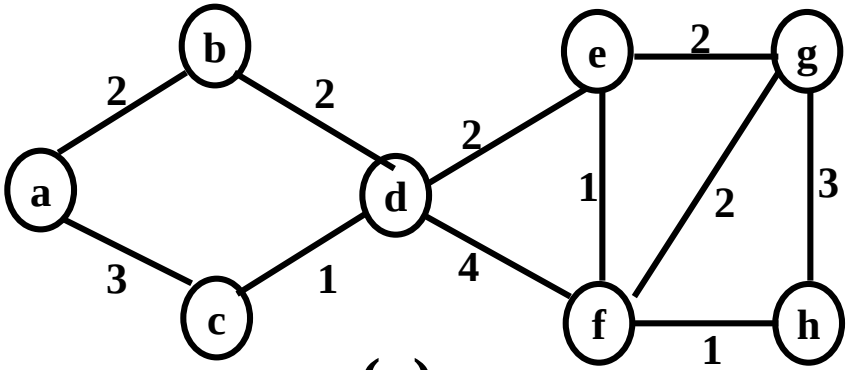
Prim算法与Kruskal算法的比较：

- 都是贪心算法；
- Kruskal算法在效率上总体上要比Prim算法快，因为Kruskal只需要对权重边做一次排序，而Prim算法则需要做多次排序；
- Prim算法是挨个找，而Kruskal是先排序再找；
- 稀疏图可以用Kruskal,因为Kruskal算法每次查找最短的边。 稠密图可以用Prim，因为它是每次加一个顶点，对边数多的适用。

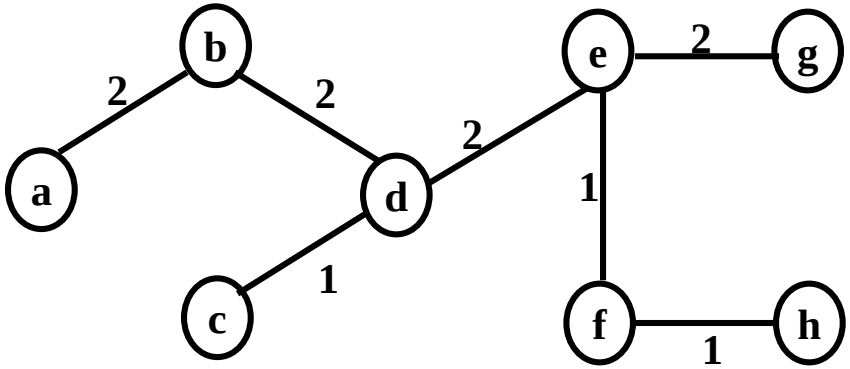
【例4-4】求最小生成树



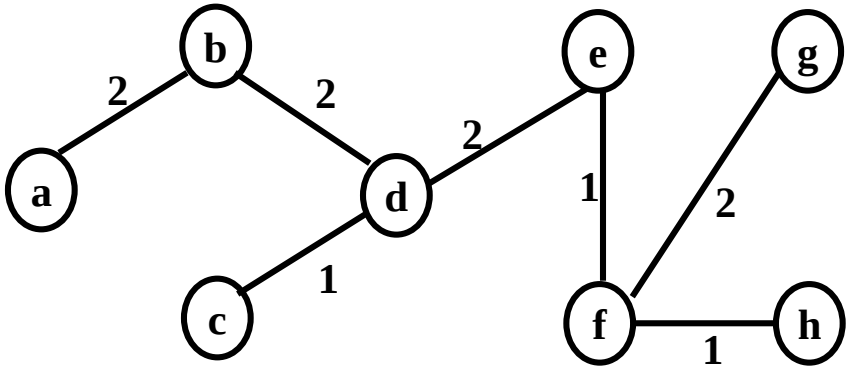
【例4-5】求最小生成树



(a)



(b)



(c)

4.5 无向图的双连通性(Biconnectivity)

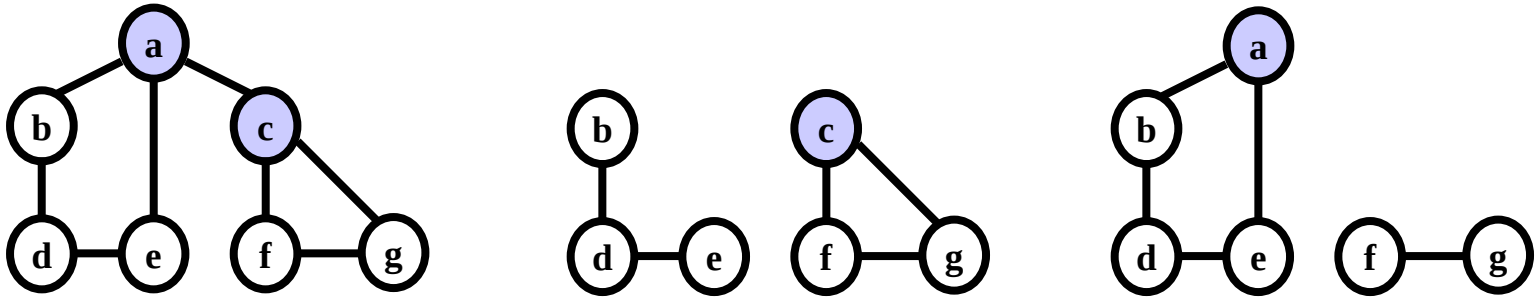
先深搜索和先深编号的作用:

通过是否遇到回退边，即可确定是否有环路。

4.5.1 无向图的双连通分量

设 G = (V, E)

【定义】 假若在删去顶点 *v* 以及和 *v* 相关联的边之后，将图的一个连同分量分割成两个或两个以上的连同分量，则称该结点为关节点。



(a)　　(b)　　(c)

【定义】 若对V中每个不同的三元组*v*,*w*,*a*；在*v*和*w*之间都存在一条不包含 *a* 的路，就说G是双连通的（Biconnected）

■ 双连通的无向图是连通的，但连通的无向图未必双连通。

■ 一个连通的无向图是双连通的，当且仅当它没有关节点。

◆*双连通图的研究意义*：如通讯网络。

【定义】 边*e1*和*e2*等价，若$e_1=e_2$或者有一条环路包含e_1又包含e_2，则称边e_1和e_2是等价的。

上述等价关系将E分成等价类$E_1,E_2,\ldots,E_k$,两条不同的边属于同一个类的充要条件是它们在同一个环路上。

【定义】 设V_i是E_i中各边所连接的点集（1≤i≤k），每个图$G_i=(V_i, E_i)$叫做G的一个双连通分量。

双连通分量的性质：

性质1： G_i是双连通的（1≤i≤k）；

性质2： 对所有的$i\neq j$, $V_i\cap V_j$最多包含一个点；

性质3： v是G的关节点，当且仅当$v\in V_i\cap V_j\ (i\neq j)$。

【例4-6】双连通分量

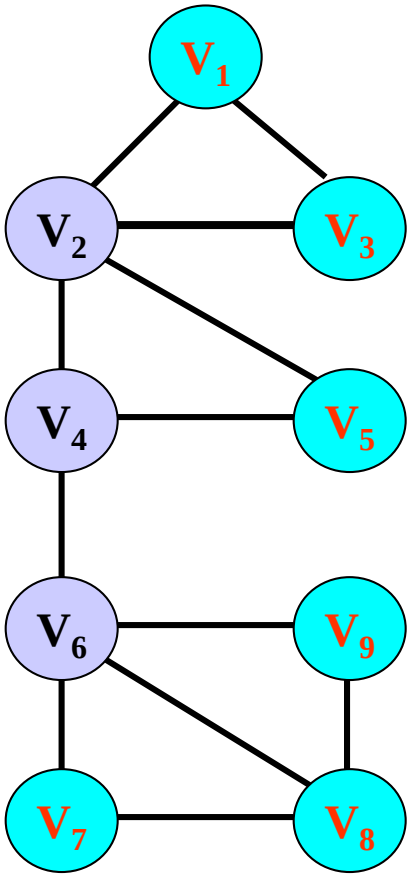
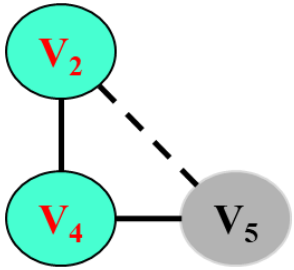
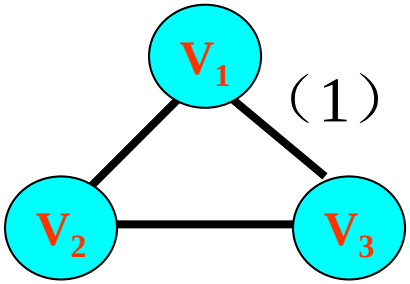
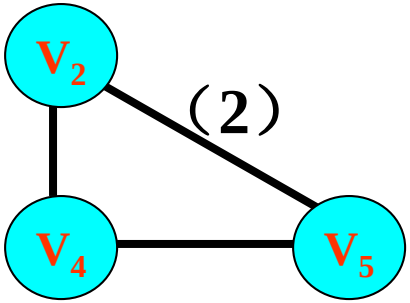


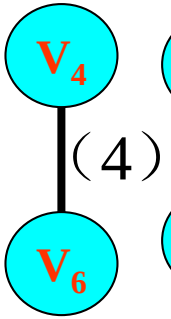
图 G

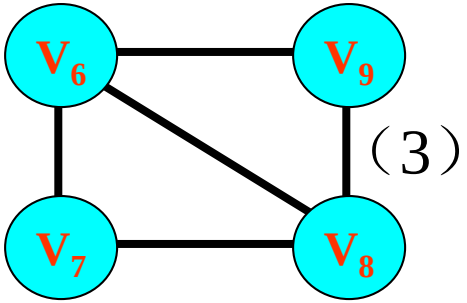
图G的双连通分量

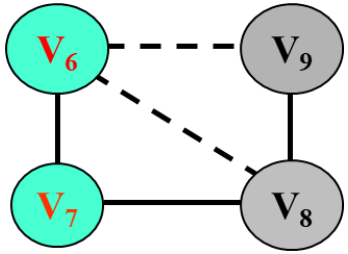


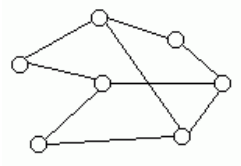




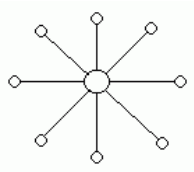




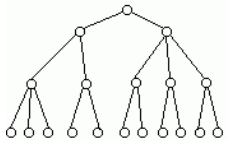




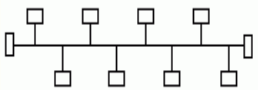
(1)网状网：
•结构：所形成的网络链路较多，形成的拓扑结构象网状。
•优点：线路冗余度大，网络可靠性高，任意两点间可直接通信；
•缺点：线路利用率低(N 值较大时传输链路数将很大)，网络成本高，另外网络的扩容也不方便，每增加一个节点，就需增加N条线路。
•适用场合：通常用于节点数目少，又有很高可靠性要求的场合。



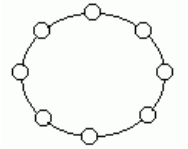
(2)星形网又称辐射网
•结构：星形结构由一个功能较强的转接中心S以及一些各自连到中心的从节点组成。
•优点：与网形网相比，降低了传输链路的成本，提高了线路的利用率
•缺点: 网络的可靠性差，一旦中心转接节点发生故障或转接能力不足时，全网的通信都会受到影响。
•适用场合：传输链路费用高于转接设备、可靠性要求又不高的场合，以降低建网成本。局域网常见



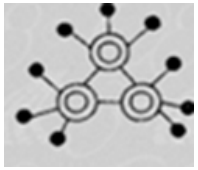
(3)树型结构
•分级结构。在树型结构的网络中，任意两个结点之间不产生回路，每条通路都支持双向传输。
•扩充方便、灵活，成本低，易推广
•适合于分主次或分等级的层次型管理系统。



(4)总线型网属于共享传输介质型网络
•结构：网中的所有节点都连至一个公共的总线上，任何时候只允许一个用户占用总线发送或接送数据。
•优点：需要的传输链路少，节点间通信无需转接节点，控制方式简单，增减节点也很方便；
•缺点：网络服务性能的稳定性差，节点数目不宜过多，网络覆盖范围也较小。
•适用场合：主要用于计算机局域网、电信接入网等网络中。局域网常见



(5)环形网
•结构：网中所有节点首尾相连，组成一个环。
•优点：是结构简单，容易实现，双向自愈环结构可以对网络进行自动保护；
•缺点：是节点数较多时转接时延无法控制，并且环形结构不好扩容。
•适用场合：目前主要用于计算机局域网、光纤接入网、城域网、光传输网等网络中。

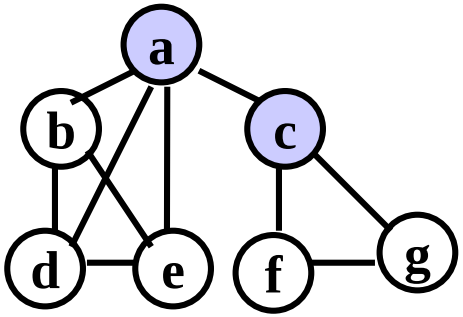


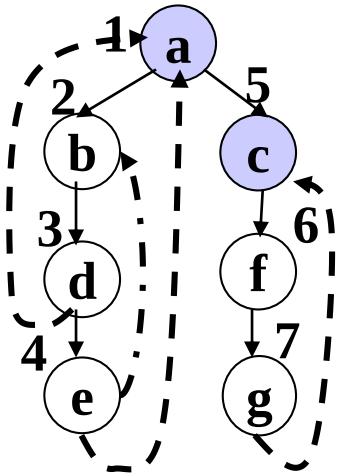
(6)复合型网
•结构：是由网状网和星形网复合而成的。它以星形网为基础，在业务量较大的转接交换中心之间采用网状网结构.
•优点：兼并了网状网和星形网的优点。整个网络结构比较经济，且稳定性较好。
•适用场合：规模较大的局域网和电信骨干网中广泛采用分级的复合型网络结构。

4.5.2 求关节点—对图进行一次先深搜索便可求出所有的关节点

由先深生成树可得出两类关节点的特性：

- 若生成树的根有两株或两株以上子树，则此根结点必为关节点（第一类关节点）。因图中不存在连接不同子树中顶点的边，因此，若删去根顶点，生成树变成生成森林。
- 若生成树中非叶顶点v，其某株子树的根和子树中的其它结点均没有指向v 的祖先的回退边，则v是关节点（第二类关节点）。 因为删去v，则其子树和图的其它部分被分割开来。





树边：编号小⟶大

回退边：编号大⟶小

定义 low[v]：设对连通图G＝（V,E）进行先深搜索的先深编号为dfn[v]，产生的先深生成树为S＝（V,T），B是回退边之集。对每个顶点*v*，low[*v*]定义如下：

$$low[v]=\min\left\{dfn[v],dfn[w],low[y]\ \middle|\ \begin{array}{l}(v,w)\in B，\ w\text{ 是顶点}v\text{ 在先深生成树上有回退边连接的祖先结点；}\\(v,y)\in T，\ y\text{ 是顶点}v\text{ 在先深生成树上的孩子顶点}\end{array}\right\}$$

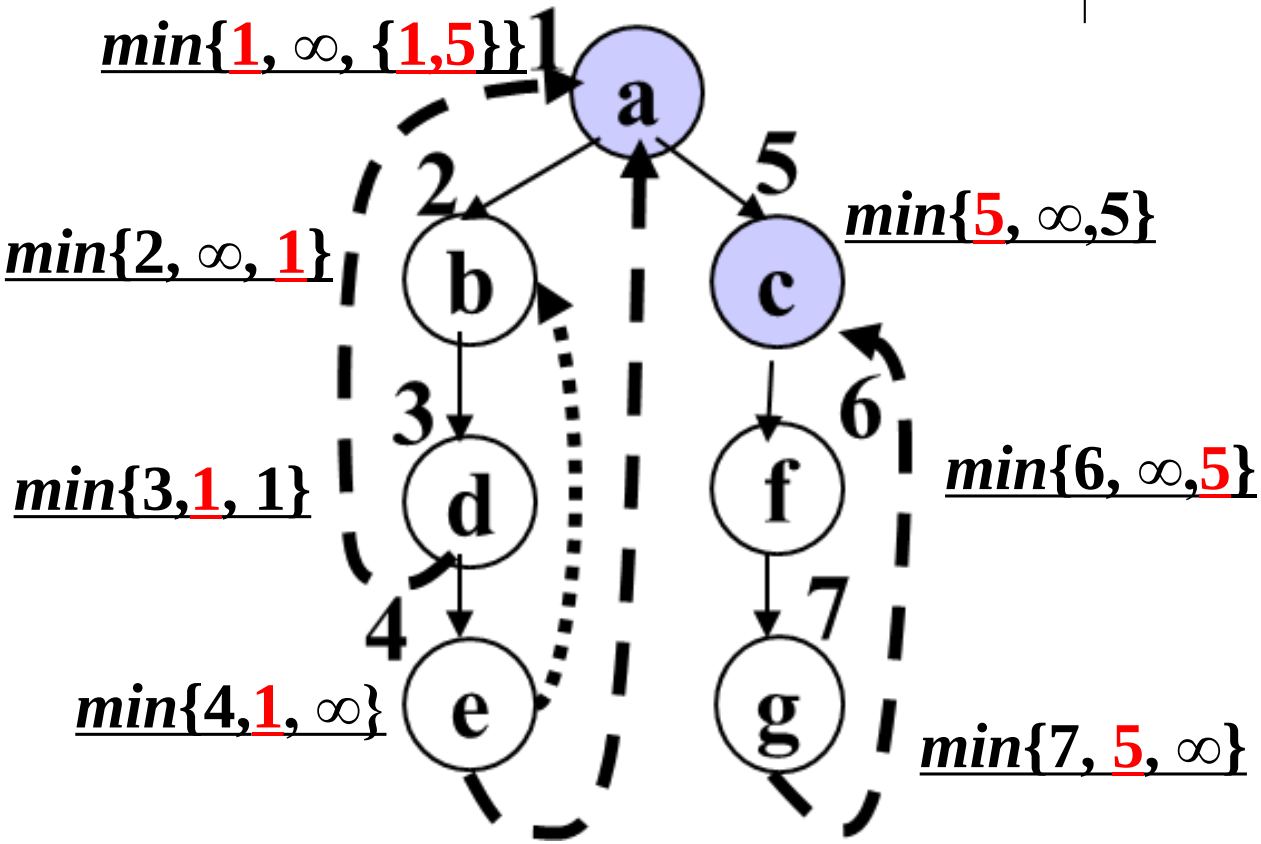
Low[*v*]取顶点*v*和*w*的深度优先编号的较小者，其中的*w*是从*v*点沿着零条或多条树边到*v*的后代*x*，之后沿着任意一条回退边(*x*,*w*)所能达到的任何顶点。

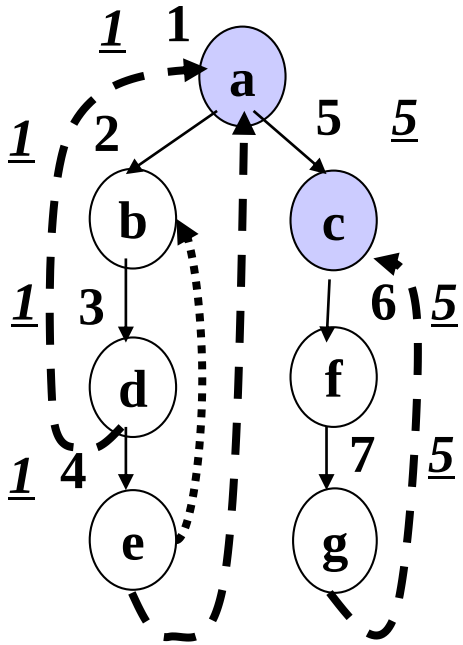
后根遍历实现如下求值：

$$low[v]=\min\begin{cases} dfn[v] \\ dfn[w] \\ low[y] \end{cases}\quad \begin{array}{l}(v,w)\in B，\ w\text{ 是顶点}v\text{ 在先深生成树上有回退边连接的祖先结点；}\\ (v,y)\in T，\ y\text{ 是顶点}v\text{ 在先深生成树上的孩子顶点}\end{array}$$

dfn

Min{}





求无向图的双连通分量算法步骤：

输入：连通的无向图G=(V, E)。L[*v*]表示关于 *v* 的邻接表。
输出：G的所有双连通分量，每个连通分量由一序列的边组成。

算法要点：

(1)计算先深编号:对图进行先深搜索，计算每个结点v的先深编号dfn[*v*]，形成先深生成树S＝（V,T）。

(2)计算low[*v*]:在先深生成树上按后根顺序进行计算每个顶点 *v* 的low[*v*]， low[*v*]取下述三个结点中的最小者：

1) dfn[*v*]: *v* 的先深编号；
2) dfn[*w*]:凡是有回退边（*v*,*w*）的任何结点 *w*；
3) low[*y*]: 对*v*的任何儿子(树边) *y* 。

(3)求关节点:

1)树根是关节点，当且仅当它有两个或两个以上的儿子（第一类关节点）；

2)非树根结点 *v* 是关节点当且仅当 *v* 有某个儿子 *y* ，使 low[*y*]≥dfn[*v*]（第二类关节点）。

```
Void searchB(v)
{  (1)  make v "old" ;
   (2)  dfn[v]=count  ;
   (3)  count++;
   (4)  low[v]=dfn[v] ;
   (5)  for ( each w ∈ L[v] )
   (6)      if(w is marked"new")
   (7)       { add(v,w) to T ;
   (8)         father[w]=v;      // w 是 v 的儿子
   (9)         searchB(w);
   (10)          if(low[w]>=dfn[v])
                    A biconnected component has been found ;
   (11)         low[v]=min(low[v],low[w]);   }
   (12)    else if (w is not father[v] )   // (v ,w ) 是回退边
   (13)         low[v]=min(low[v],dfn[w]);
}
```

调用过程：

T=¢;

count=1;

for(all v∈V) make v "new" ;

searchB(v_0); //v0为任意顶点

理解即可!

4.6 有向图的搜索

DFS 和 BFS 搜索在有向图和无向图中的区别？

有向图搜索： 树边、向前边、回退边、和横边。

(1) 若dfn[*v*]<dfn[*w*]，则(*v*,*w*)是树边或向前边；

此时，visited[*v*]="old",visited[*w*]="new",(*v*,*w*)为 树边；

visited[*v*]="old",visited[*w*]="old",(*v*,*w*) 为 向前边。

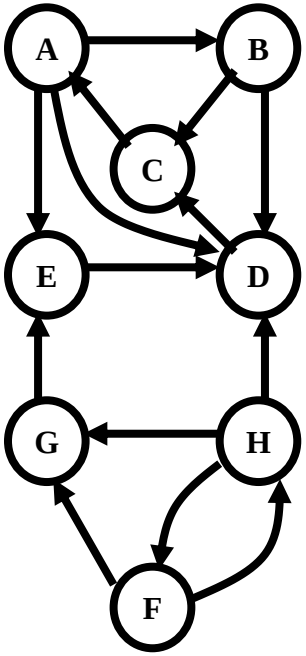
(2) 若dfn[*v*]>dfn[*w*]，则(*v*,*w*)是回退边或横边；

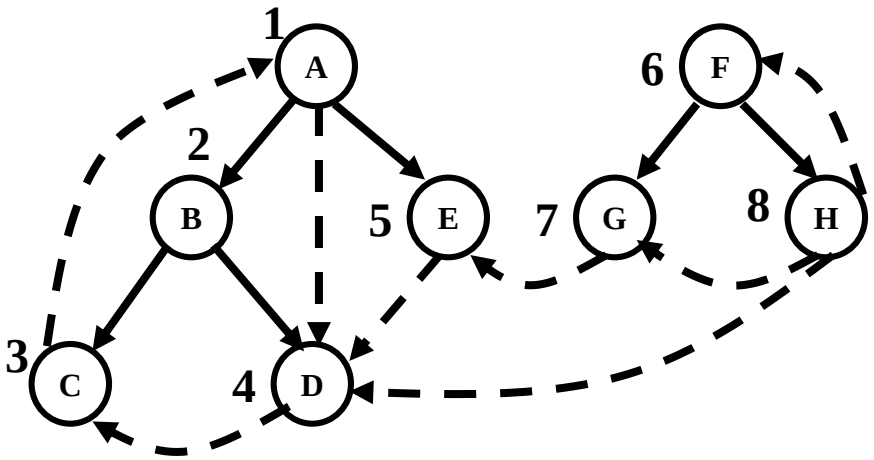
当产生树边(i,j) 时，同时记下j的父亲：father[j]=i,于是对图中任一条边(*v*,*w*)，当visited[*v*]="old",visited[*w*]="old"且dfn[*v*]>dfn[*w*]时，由结点v沿着树边向上(father中)查找*w*（可能直到根）；

若找到*w*,则(*v*,*w*)是回退边，否则是横边。

【例4-7】生成森林

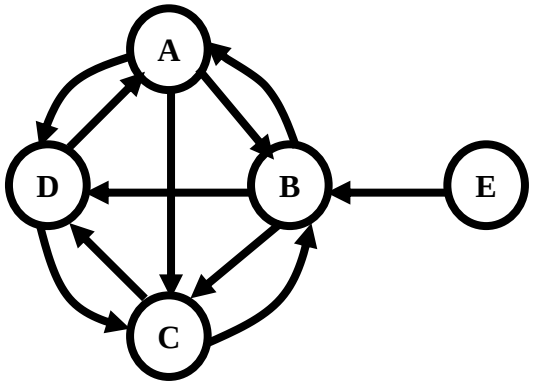
(a)

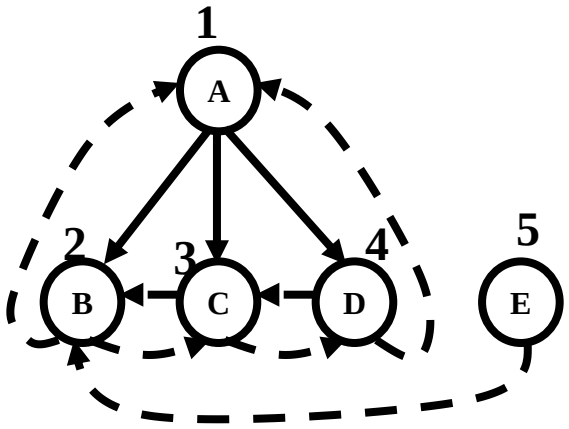




图(a)先深生成森林

(b)





图(b)先广生成森林

4.7 强连通性

【定义】设G =（V, E）是一个有向图，称顶点v,$w \in V$是等价的，要么$v = w$；要么从顶点 v 到 w 有一条有向路，并且从顶点 w 到 v 也有一条有向路。

上述等价关系将V分成若干个等价类V_1，V_2, … , V_r

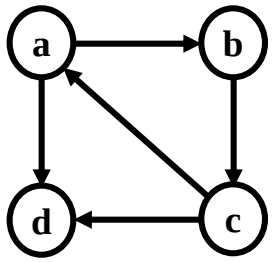
【定义】设 E_i（$1 \le i \le r$）是头、尾均在V_i 中的边集，则：

$G_i = (V_i, E_i)$ 称为 G 的一个强连通分量，简称强分量。

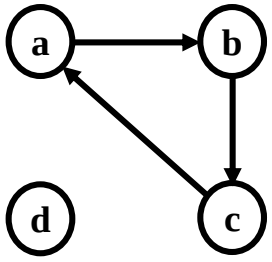
强连通图：只有一个强分量的有向图称为强连通图。

■有向图的强连通分量是满足下列要求的最大子集：

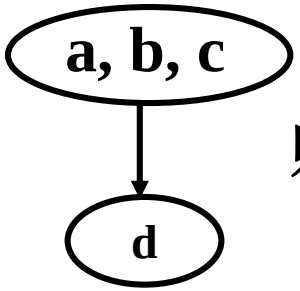
对任意两个顶点 x 和 y ，都存在一条有向路从 x 到 y ，也存在一条有向路从 y 到 x。



(a)



(b)



归约图

分支横边：不在任何强连通分支（量）中的边，如：$a \rightarrow d, c \rightarrow d$

【注】每个结点都是在某个强连通分支中出现，但有些边可能不在任何强分支中。

归约图：用强分量代表顶点，用分支横边代表有向边的图称为原图的归约图。

显然，归约图是一个不存在环路的有向图，它表示了强分量之间的连通性。

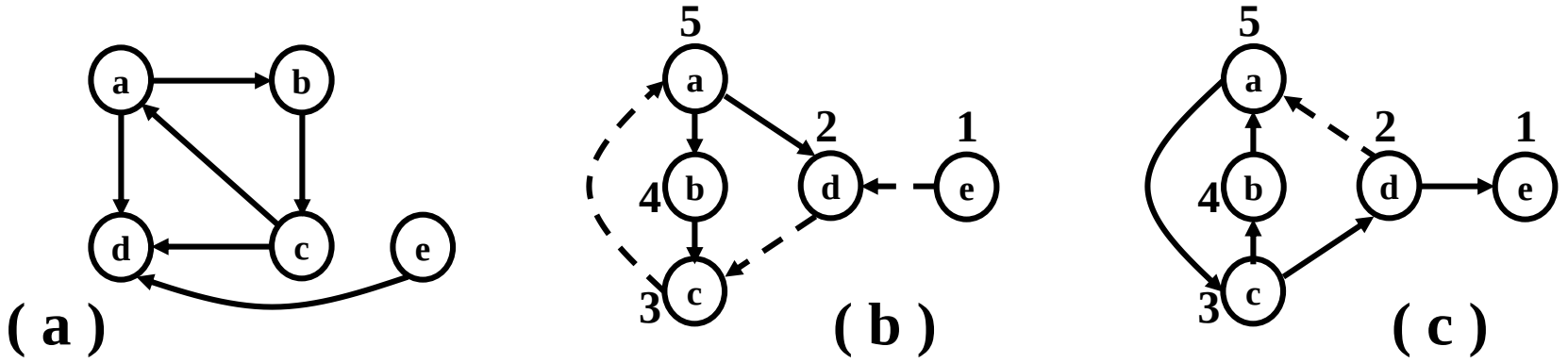
求强连通图算法步骤：

(1)对有向图G进行DFS并按树的逆先根顺序对顶点编号；

(2)将G中的每条边取反方向，构造一个新的有向图G_r；

(3)根据(1)的编号，从编号最大顶点对图G_r进行一次DFS搜索，凡是经过树边（(1) 中的分支横边除外）能到达的所有顶点，都形成一个DFS生成树；若本次搜索没有达到所有顶点，则下次DFS从余下顶点中编号最大的顶点开始；

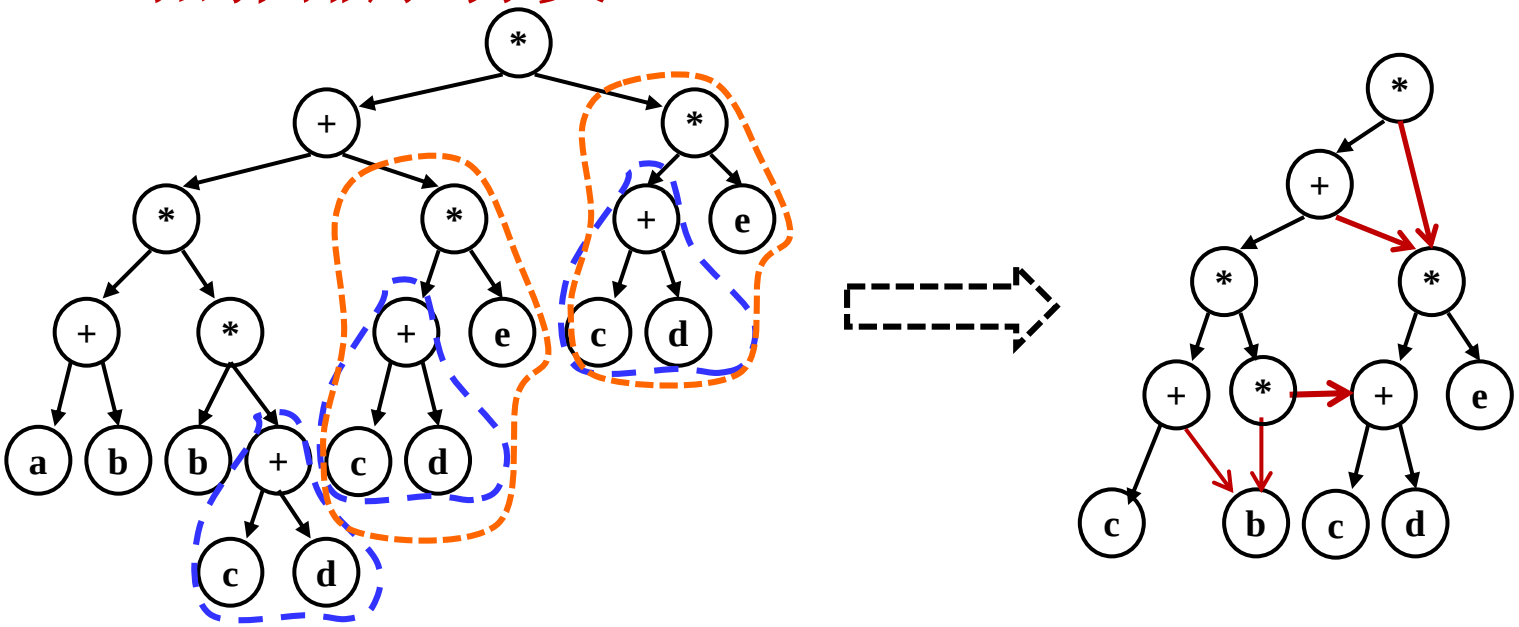
(4)在G_r的DFS生成深林中，每棵树对应与G的一个强连通分量。



求强连通图的实例

4.8 拓扑排序/分类

有向无环图及其应用



中缀：((a+b)*(b*(c+d))+(c+d)*e)*((c+d)*e)

给定一个无环路有向图G=(V,E)，各结点的编号为v=(1,2，…,n)。要求对每一个结点 i 重新进行编号，使得若 i 是 j 的前导，则有新的编号label[i]<label[j]。换而言之，拓扑分类是将无环路有向图排成一个线性序列，使当从结点 i 到结点 j 存在一条弧，则在线性序列中，将 i 排在 j 的前面。

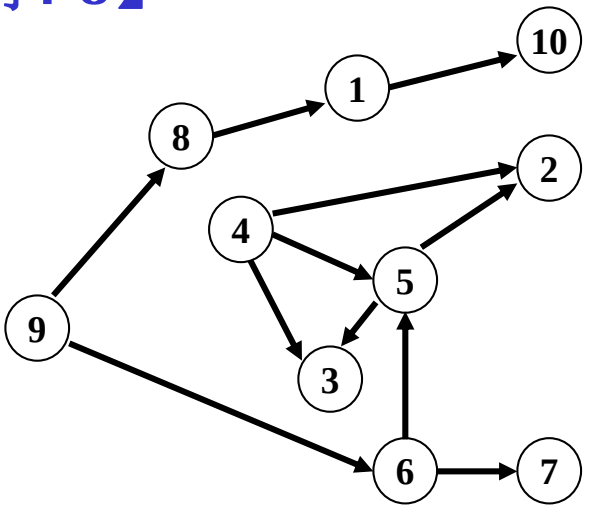
【问题1】日常工作中，可能会将项目拆分成A、B、C、D四个子部分来完成，但A依赖于B和D，C依赖于D。为了计算这个项目进行的顺序，可对这个关系集进行拓扑排序，得出一个线性的序列，则排在前面的任务就是需要先完成的任务。

【问题2】有n个士兵（1≤n≤26），编号依次为A、B、C，… 队列训练时，指挥官要把一些士兵从高到矮依次排成一行。但现在指挥官不能直接获得每个人的身高信息，只能获得任意两个人“p1比p2高”这样的比较结果:

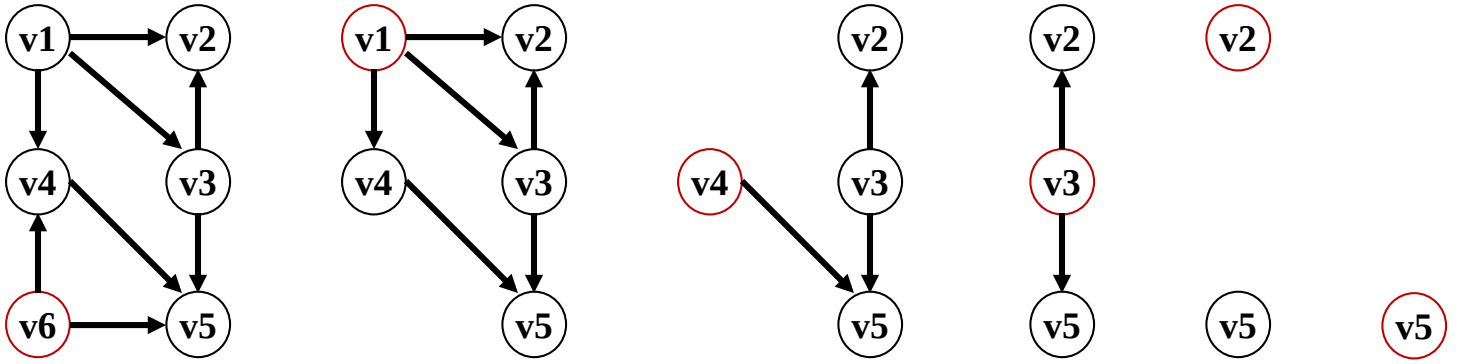
（p1，p2∈{‘A’, … ，'Z'}），记为p1>p2。

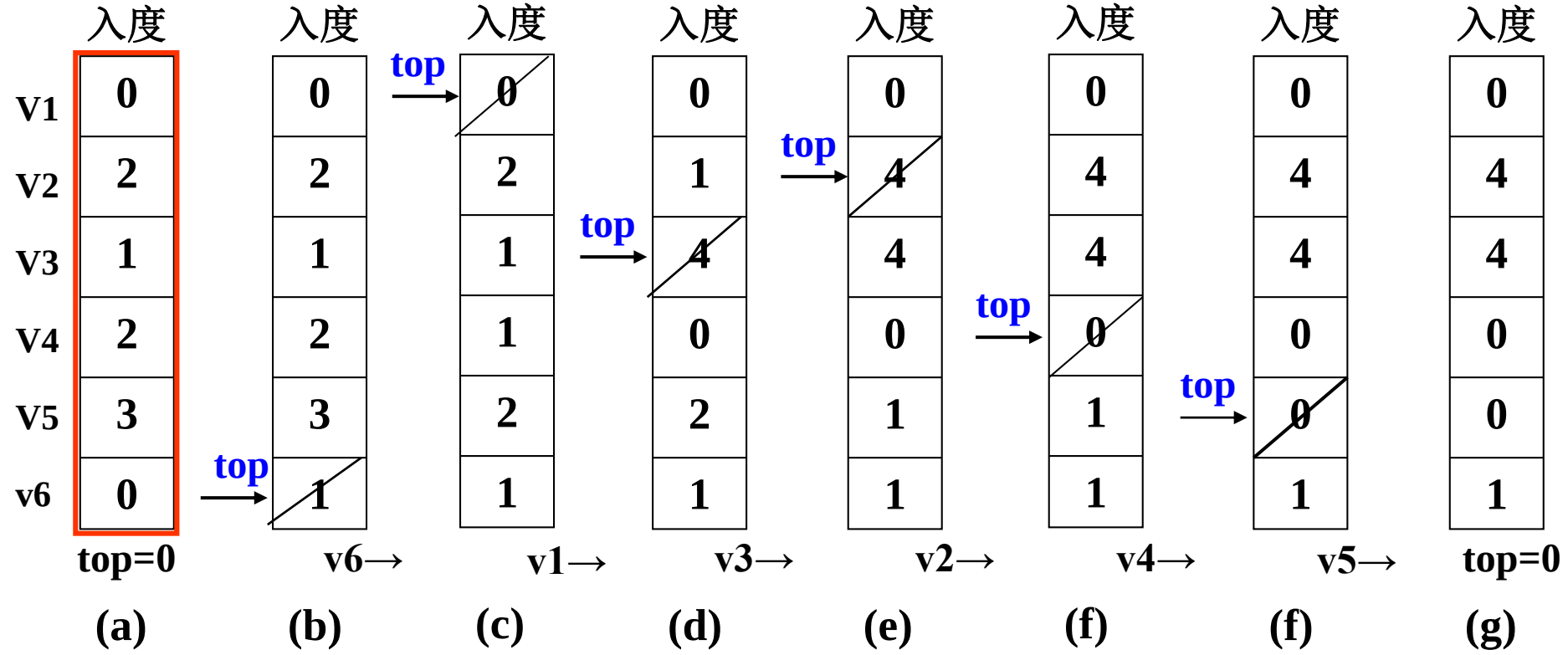
课程代号	课程名称	先修课代号
1	计算机原理	8
2	编译原理	4,5
3	操作系统	4,5
4	程序设计	无
5	数据结构	4,6
6	离散数学	9
7	形式语言	6
8	电路基础	9
9	高等数学	无
10	计算机网络	1

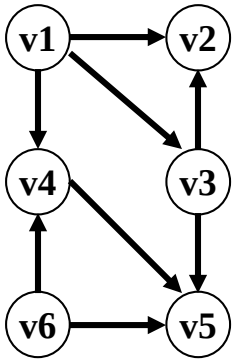
【例4-8】

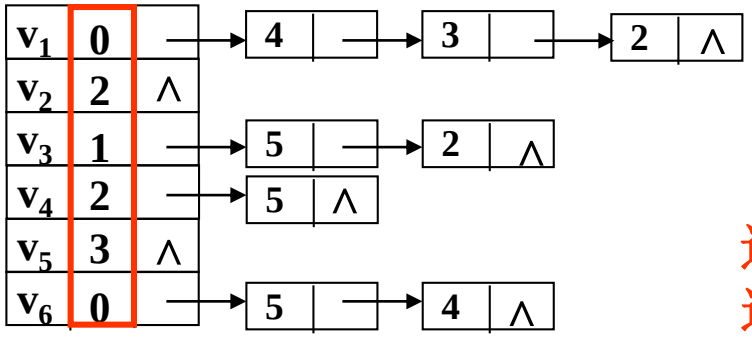


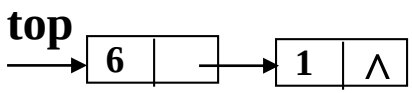
可输出结点的入度为零，删去该结点，并将与该顶点相邻的顶点的入度减1。







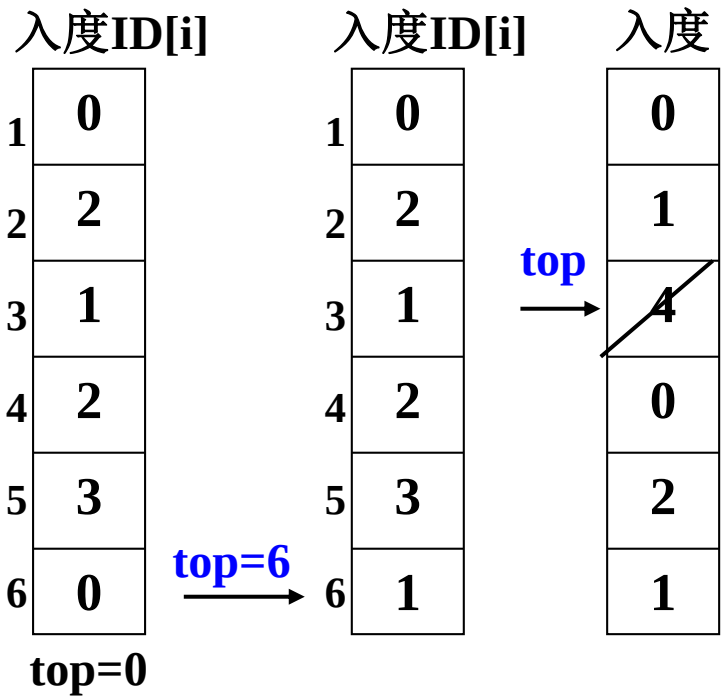


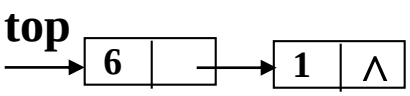


链栈的初始状态

进栈：ID[i]=top; top=i;

退栈：i=top; top=ID[top]





进栈：ID[i]=top;
　　top=i;
退栈：i=top;
　　top=ID[top]

算法主要步骤：

```
计算每个顶点i的入度ID[i];
for ( i=1; i<=n ; i++ )  //初始化链栈;
    if ( ID[i] == 0)
        {  ID[i] = top;  top = i ;  }
for ( i=1 ; i<=n ; i++ )
{  输出top指针指向的顶点;
    k = top;
    top = ID[top];
    顶点 k 的每一个邻接点 j 的入度 -1
        if  (  ID[j] == 0  )
            {  ID[j] = top;
                top = j;      }
}
```

拓扑分类算法—应用栈

```
Status   Topologicalsort( ALGRAPH  G )
{   FindInDegree( G, InDegree ) ;
    MakeNull( S ) ;
    for( v=0; v<n ; ++v )
        if ( !InDegree[v] )  push( v, S ) ;
    count = 0 ;
    while ( !Empty( S ) )
        {  v = Pop ( S ) ; printf( v );  ++count ;
           for( 邻接于 v 的每个顶点 w )
               {
                  if( !(--InDegree[w])) push(S, w) ;
               }
         }
     if ( count < n )  cout<<“图中有环路” ;
     else return OK;
}
```

拓扑分类算法—应用队列

```
Status   Topologicalsort( L )
{   QUEUE  Q ;
    MakeNull( Q ) ;
    for( v=1; v<=n ; ++v )
        if ( InDegree[v] = 0 )  EnQueue( v, Q ) ;
    nodes = 0 ;
    while ( !Empty( Q ) )
        {  v = Front(Q) ;
           DeQueue( Q ) ;
           cout << v ;   nodes ++ ;
           for( 邻接于 v 的每个顶点 w )
             if( !(--InDegree[w])) EnQueue(w,Q) ;
         }
     if ( nodes < n )  cout<<“图中有环路” ;
}
```

拓扑分类算法— DFS遍历生成拓扑序列

```
Void topodfs  ( v )
{   Push( v ,S ) ;
    mark[v]=TRUE;
    for ( L[v] 中的每一个顶点w)  do
        if ( mark[w] = FALSE )
              topodfs ( w ) ;
    printf ( top( S ) ) ;
    POP ( S ) ;
}
```

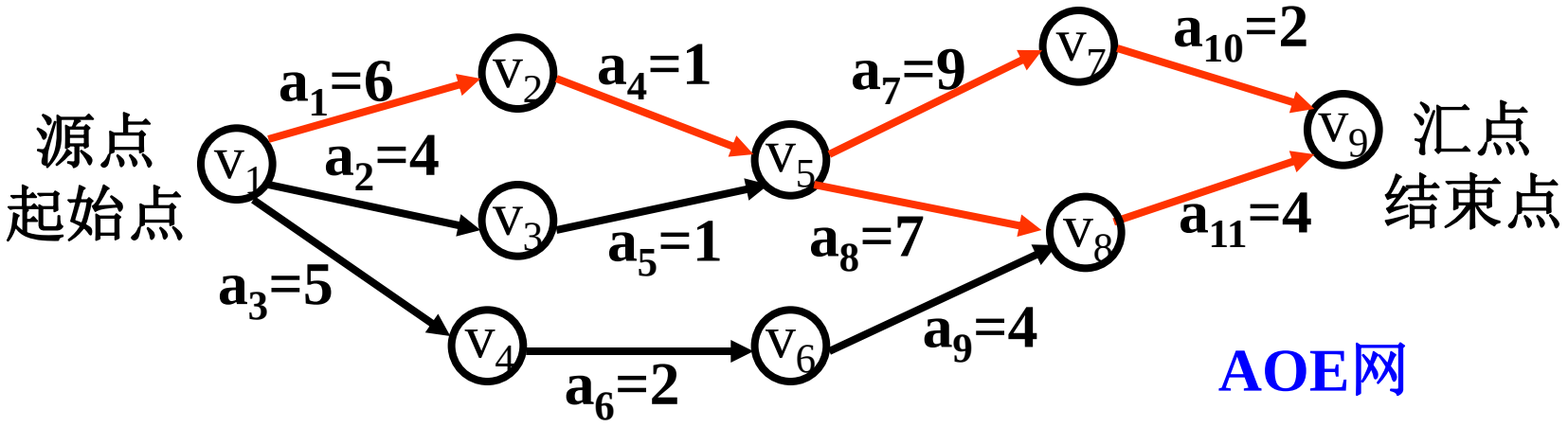
思想：借助栈，在DFS中，把第一次遇到的顶点入栈，到达某一顶点递归返回，从栈中弹出顶点并输出。

```
Void  dfs-topo ( GRAPH  L )
{  MakeNull( S );
   for( u=1;u<=n;u++)
       make[u]=FALSE;
   for( u=1;u<=n;u++)
       if ( !mark[u] )
                topodfs( u ) ;
}
```

4.9 关键路径

AOE网（Activity On Edge Network）

在带权的有向图中，用结点表示事件，用边表示活动，边上权表示活动的开销（如持续时间），则称此有向图为边表示活动的网络,简称AOE网。

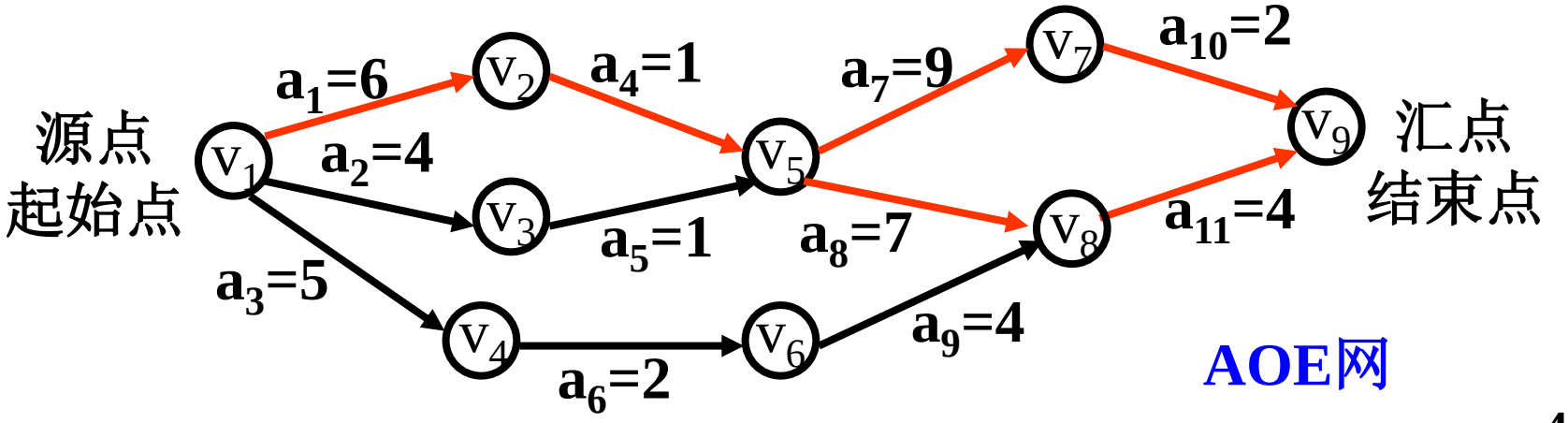


AOE网的性质：

◆只有在某个顶点所代表的事件发生后，从该顶点出发的各有向边代表的活动才能开始；

◆只有在进入某一顶点的各有向边代表的活动已经结束，该顶点所代表的事件才能发生；

◆表示实际工程计划的AOE网应该是无环的，并且存在唯一的入度为0的开始顶点（源点）和唯一的出度为0的结束点（汇点）。

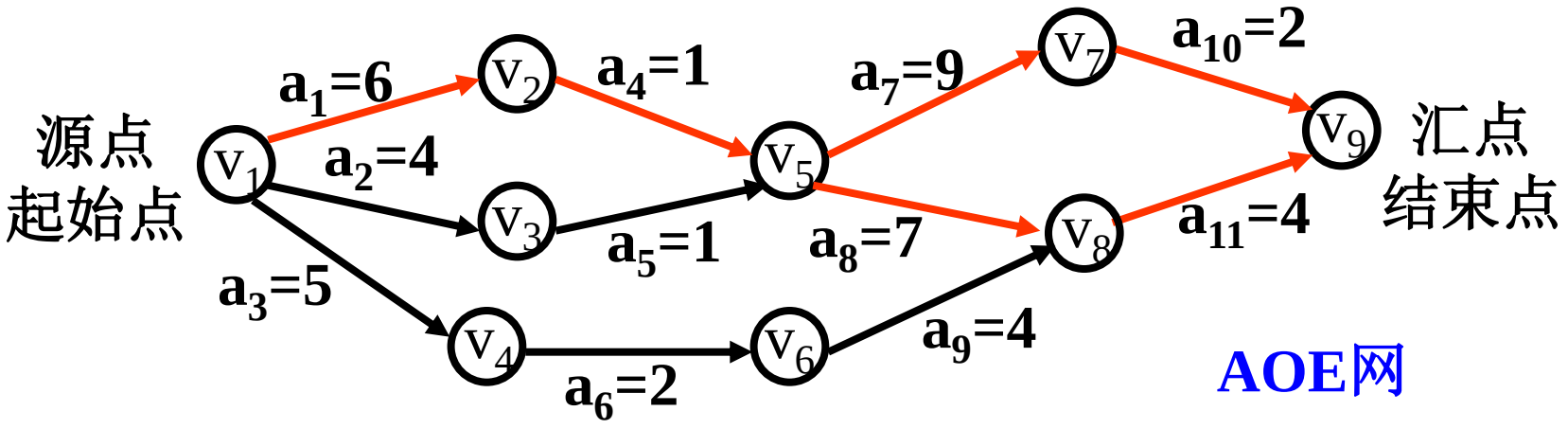


AOE网研究的主要问题：

如果用AOE 网表示一项工程，那么仅仅考虑各个子工程之间的优先关系还不够，更多地是关心整个工程完成的最短时间是多少，哪些活动的延迟将影响整个工程进度,而加速这些活动能否提高整个工程的效率,因此AOE网有待研究的问题是:

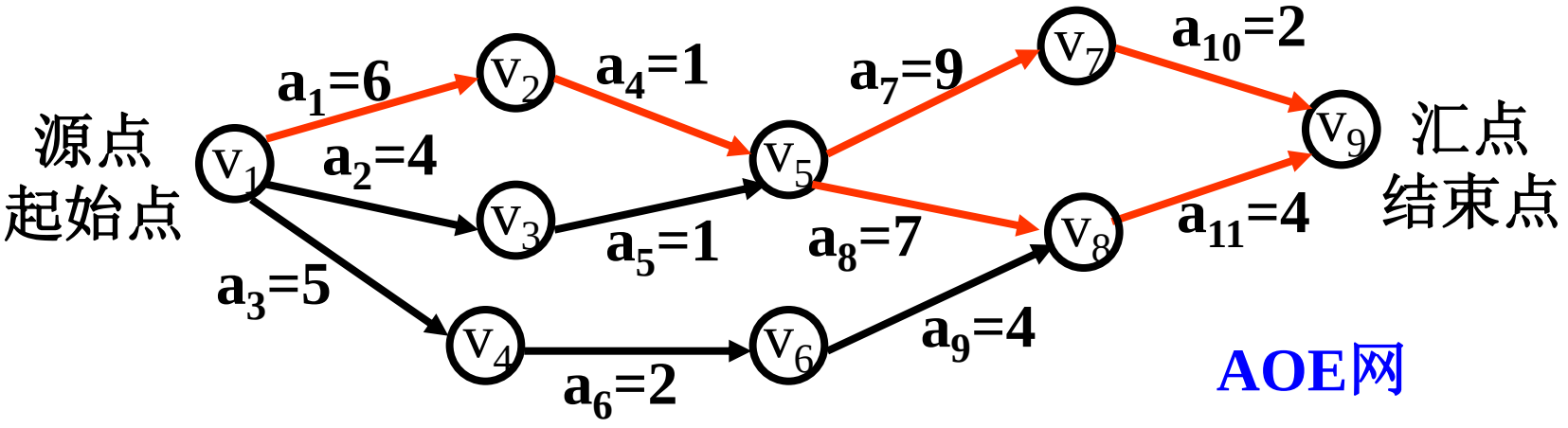
(1) 完成整个工程至少需要多少时间？

(2) 哪些活动是影响工程进度的关键活动？



路径长度、关键路径、关键活动：

- 路径长度：是指从源点到汇点路径上所有活动的持续时间之和。
- 关键路径：在AOE网中，由于有些活动可以并行，所以完成工程的最短时间是从源点到汇点的最大路径长度。因此，把从源点到汇点具有最大长度的路径称为关键路径。
- 一个AOE中，关键路径可能不只一条。
- 关键活动：关键路径上的活动称为关键活动。



关键路径和关键活动性质分析：（与计算关键活动有关的量）

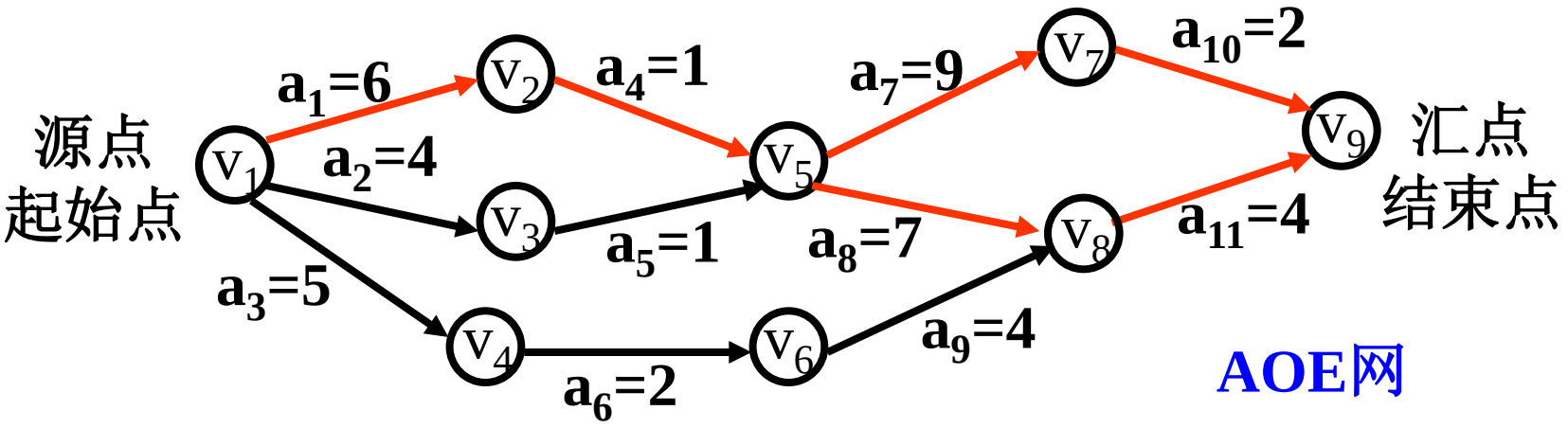
① 事件V_j的最早可能发生时间$VE(j)$

是从源点V_1到顶点V_j的最长路径长度。

②活动a_i的最早可能开始时间 $E(k)$

设活动a_i在边< V_j, V_k>上, 则$E(i)$也是从源点V_1到顶点V_j的最长路径长度。这是因为事件V_j发生表明以V_j为起点的所有活动a_i可以立即开始。因此,

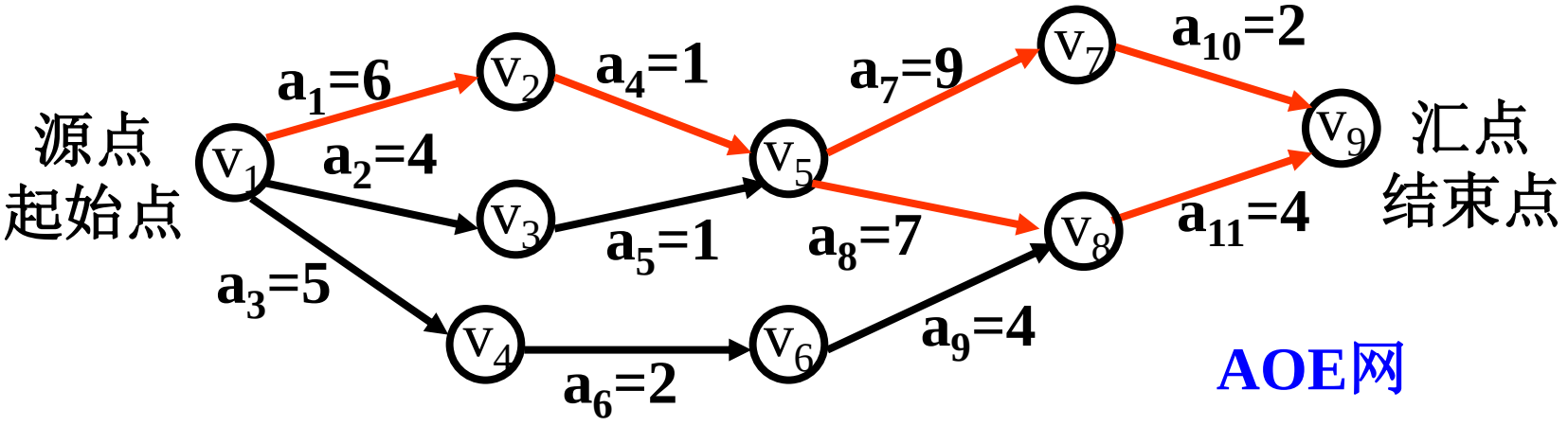
$$E(i) = VE(j) \text{ ……..(1)}$$



③事件V_k的最迟发生时间*VL(k)*

是在保证汇点V_n在*VE(n)*时刻完成的前提下，事件V_k的允许的最迟开始时间。

在不推迟工期的情况下，一个事件 最迟发生时间*VL(k)*应该等于汇点的最早发生时间*VE(n)*减去从V_k到V_n的最大路径长度。



④ 活动a_i的最迟允许开始时间 *L(i)*

是指在不会引起工期延误的前提下，活动a_i允许的最迟开始时间。

因为事件V_k发生表明以V_k为终点的入边所表示的所有活动均已完成，所以事件V_k的最迟发生时间*VL(k)*也是所有以V_k为终点的入边<V_j, V_k>所表示的活动a_i可以最迟完成时间。

显然，为不推迟工期，活动a_i的最迟开始时间*L(i)*应该是a_i的最迟完成时间*VL(k)*减去a_i的持续时间，即：

$$L(i) = VL(k) - ACT[j][k] \quad \text{.....(2)}$$

其中, ACT[j][k]是活动a_i的持续时间（< V_j, V_k> 上的权）。

⑤时间余量 $L(i) - E(i)$

$L(i) - E(i)$表示活动 a_k 的最早可能开始时间和最迟允许开始时间的时间余量。

关键路径上的活动都满足：$L(i) = E(i)$ ……..(3)

$L(i) = E(i)$表示活动是没有时间余量的关键活动。

由上述分析知，为找出关键活动，需要求各个活动的$E(i)$与$L(i)$，以判别一个活动a_i是否 满足$L(i) = E(i)$。 $E(i)$和$L(i)$可有公式 (1)和(2)。而$VE(k)$ 和$VL(k)$可由拓扑分类算法得到。

利用拓扑分类算法求关键路径和关键活动。

利用拓扑分类算法求关键路径和关键活动：

◆ **Step1**(前进阶段)：从源点V_1出发，令$VE(1) = 0$，按拓扑序列次序求出其余各顶点事件的最早发生时间：

$$VE(k) = \max_{j \in T}\{ VE(j)+ACT[j][k] \}$$

其中T是以顶点V_k为尾的所有边的头顶点的集合($2 \le k \le n$)

如果网中有回路，不能求出关键路径则算法中止；否则转**Step2**。

◆ **Step2**(回退阶段)：从汇点V_n出发，令$VL(n) = VE(n)$，按逆拓扑有序求其余各顶点的最晚发生时间：

$$VL(j) = \min_{k \in S}\{ VL(k)-ACT[j][k] \}$$

其中S是以顶点V_j为头的所有边的尾顶点的集合($2 \le j \le n-1$)。

◆ **Step3**：

求每一项活动a_i的最早开始时间：

$$E(i) = VE(j)$$

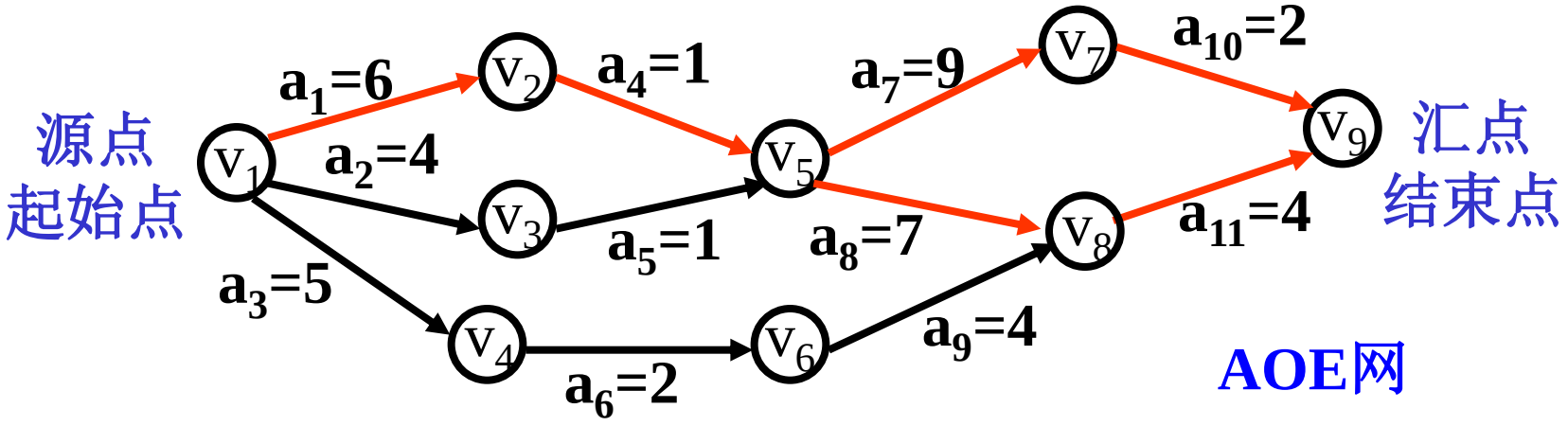
最晚开始时间：

$$L(i) = VL(k) - ACT[j][k]$$

若某条边满足$E(i) = L(i)$，则它是关键活动。

- ✓ 为了简化算法，可以在求关键路径之前已经对各顶点实现拓扑排序，并按拓扑有序的顺序对各顶点重新进行编号。
- ✓ 不是任意一个关键活动的加速一定能使整个工程提前。
- ✓ 想使整个工程提前，要考虑各个关键路径上所有关键活动。

【例4-9】关键路径1



事件	*ve*	*vl*	*活动*	*e(i)*	*l(i)*	*l(i)-e(i)*
v_1	0	0	a1	0	0	0
v_2	6	6	a2	0	2	2
v_3	4	6	a3	0	3	3
v_4	5	8	a4	6	6	0
v_5	7	7	a5	4	6	2
v_6	7	10	a6	5	8	3
v_7	16	16	a7	7	7	0
v_8	14	14	a8	7	7	0
v_9	18	18	a9	7	10	3
			a10	16	16	0
			a11	14	14	0

注：

Ve(i)：事件最早可能发生时间

源点到达该事件的最长路经

Vl(i)：事件最迟发生时间

$VE(n) - maxL_{ik}$

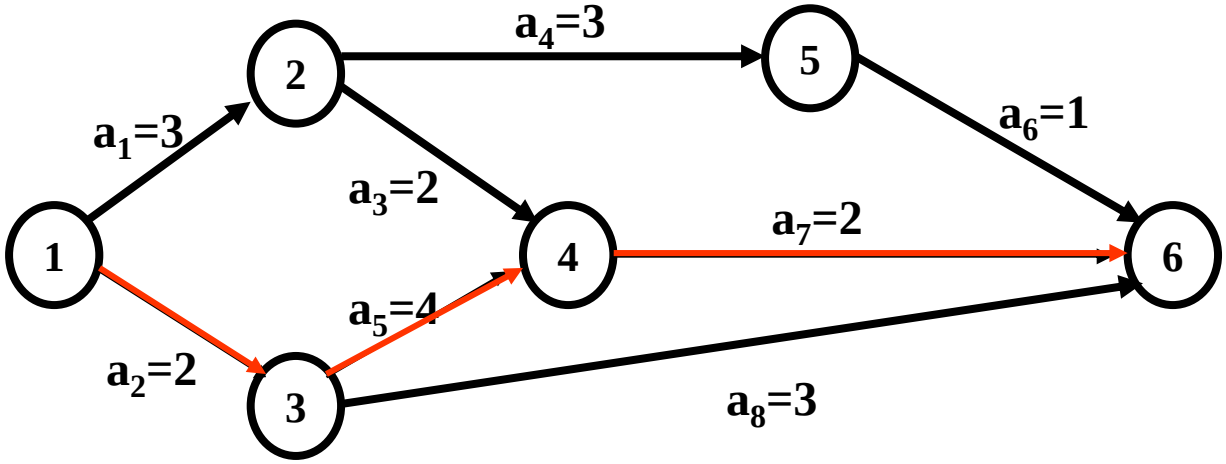
e(i)：活动最早可能开始时间

Ve(活动起点)

l(i)：活动最迟允许开始时间

Vl(活动终点) - a_i

【例4-10】关键路径2



顶点	Ve(i)	Vl(i)	活动	e(i)	l(i)	l(i)-e(i)
1	0	0	a1	0	1	1
2	3	4	a2	0	0	0
3	2	2	a3	3	4	1
4	6	6	a4	3	4	1
5	6	7	a5	2	2	0
6	8	8	a6	6	7	1
			a7	6	6	0
			a8	2	5	3

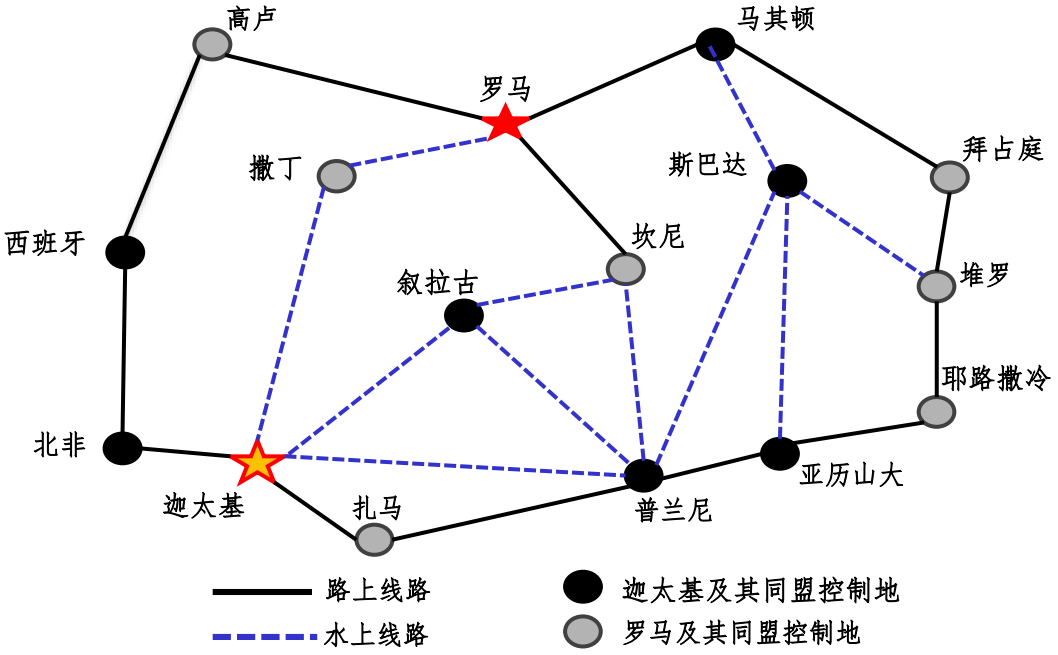
注：

事件最早可能发生时间**Ve(i)**
源点到达该事件的最长路经
事件最迟发生时间**Vl(i)**
$VE(n)-maxL_{ik}$
活动最早可能开始时间 **e(i)**
Ve(活动起点)
活动最迟允许开始时间 **l(i)**
Vl(活动终点)-a_i

4.10 单源最短路径

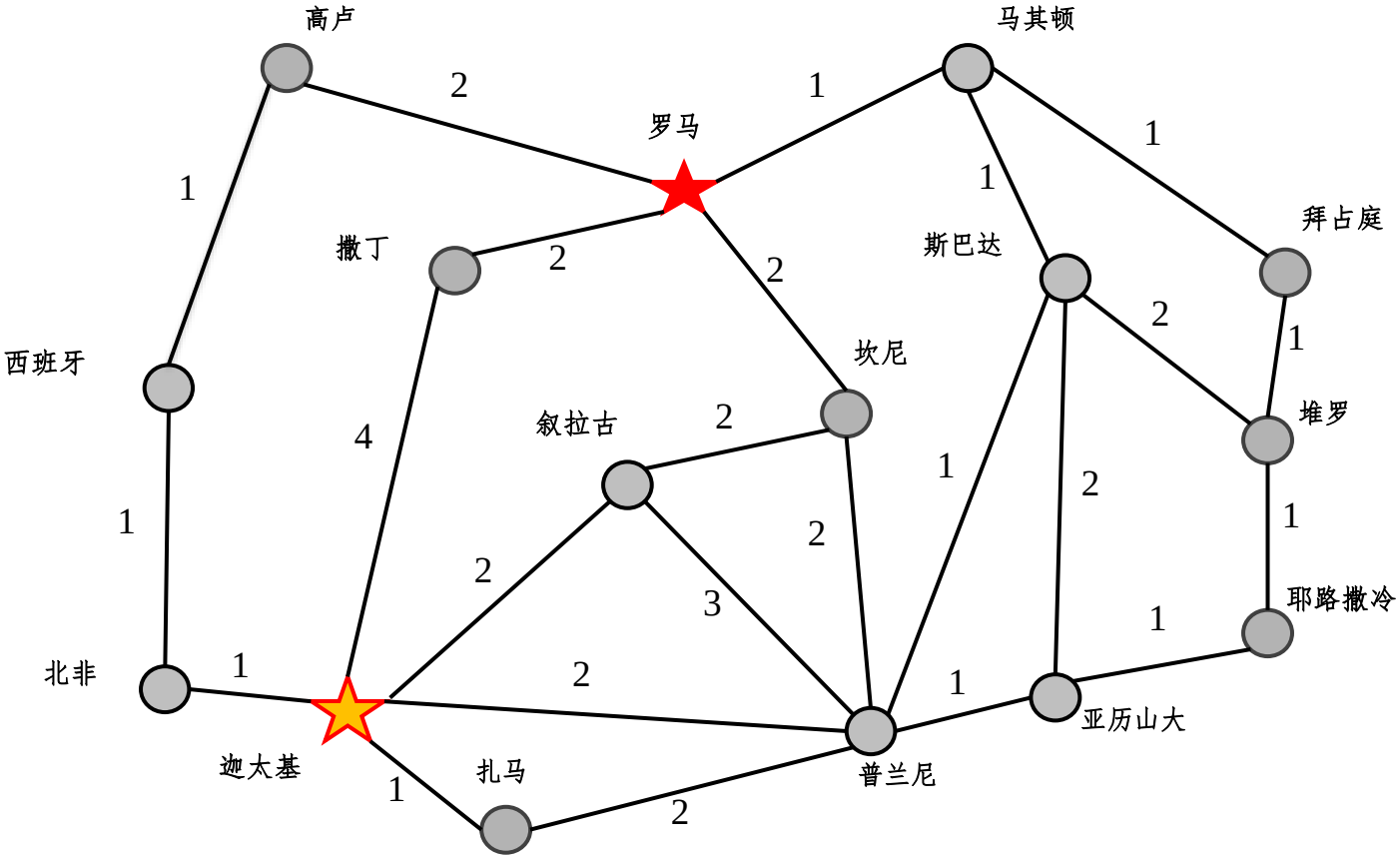
汉尼拔·巴卡（Hannibal Barca）；
公元前247年—前183年；
北非古国迦太基名将，军事家；
是欧洲历史最伟大四大军事统帅之一；
誓言终身与古罗马为敌；
被誉为战略之父。

公元前218年，
迦太基卓越的军事统帅汉尼拔决定进军罗马，准备最后解决罗马问题。选择行军路线成为首要问题。



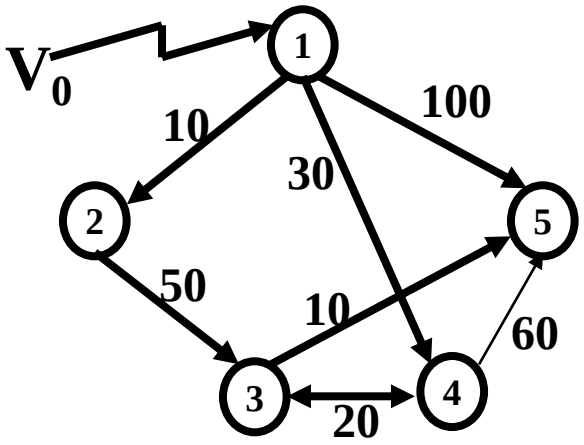
罗马帝国的海陆交通示意图

卢浮宫汉尼拔雕像



罗马帝国的海陆交通示意图—时间抽象版

4.10 单源最短路径



$$C\begin{pmatrix} \infty & 10 & \infty & 30 & 100 \\ \infty & \infty & 50 & \infty & \infty \\ \infty & \infty & \infty & 20 & 10 \\ \infty & \infty & 20 & \infty & 60 \\ \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

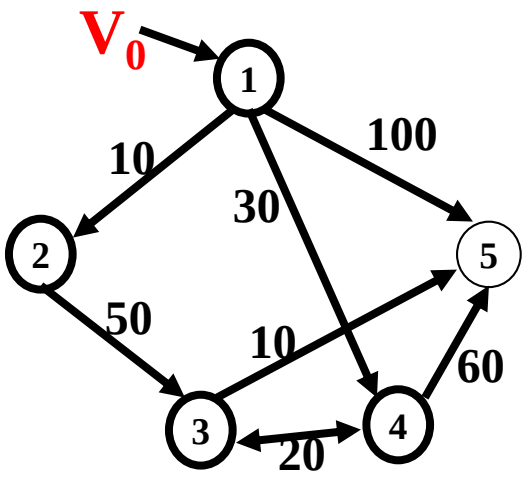
D

D[1]	D[2]	D[3]	D[4]	D[5]
∞	10	∞	30	100

集合　S = { 1 }　　{1, 2, 3, 4, 5 }

Dijkstra算法基本思想：

- 集合S的 初值为S={1}；
- D为各顶点当前最小路径
- 从V-S中选择顶点w，使D[w]的值最小并将 w加入集合 S，则w的最短路径已求出；
- 调整其他各结点的当前最小路径
  D[k]=min{D[k], D[w]+C[w][k]}
- 直到S中包含所有顶点。



算法的逐步求精过程：

循环	S	w	D[2]	D[3]	D[4]	D[5]
初态	{1}	-	10	∞	30	100
1	{1,2}	2	10	60	30	100
2	{1,2,4}	4	10	50	30	90
3	{1,2,4,3}	3	10	50	30	60
4	{1,2,4,3,5}	5	10	50	30	60

Dijkstra算法框架：

```
Void  Dijkstra( G )
{ S = { 1 } ;
  for( i=2; i<=n; i++ )
          D[i] = C[1][i];
  for( i=2; i<=n; i++ )
  {  从V-S中选出一个顶点w,使D[w]最小；
     S = S + {w} ;
     for ( V-S中的每一个顶点v )
     D[v]=min( D[v],  D[w]+C[w][v] );
   }
}
```

Dijkstra算法：

```
Void  Dijkstra(GRAPH  G, costtype D[MAXVEX+1] )
{  int  S[MAXVEX+1] ;
   for ( i=1 ; i<=n; i++ )
        {  D[i]=G[1][i] ;  S[i]=FALSE ;  }
  S [1]= TRUE ;
  for( i=2; i<=n; i++)
  {  w=mincost ( D, S ) ;
     S[w]=TRUE ;
     for ( v=2 ; v<= n ; n++ )
          if ( S[v]!=TRUE )
             {   sum=D[w] + G[w][v] ;
                 if (sum < D[v] )   D[v] = sum ; }
  }
}
```

最小路径
经过哪些点？

```
int  mincost ( D, S )
{
    temp = INFINITY ;
    w = 2 ;
    for ( i=2 ; i<=n ; i++ )
         if ( !S[i] && D[i]<temp)
            {  temp = D[i] ;
               w = i ;
            }
     return  w ;
}
```

Dijkstra算法（带路径）：

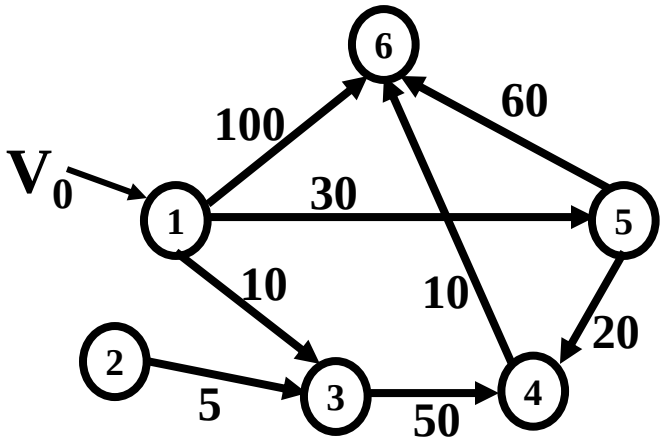
```
Void  Dijkstra(GRAPH  G, costtype D[MAXVEX+1] )
{  int  S[MAXVEX+1],  P[MAXVEX+1];
    for ( i=1 ; i<=n; i++ )
        {  D[i]=G[1][i] ;  S[i]=FALSE ; P[i]=1 ; }
   S [1]= TRUE ;
   for( i=2; i<=n; i++)
   {  w=mincost ( D, S ) ;
      S[w]=TRUE ;
      for ( v=2 ; v<= n ; n++ )
          if ( S[v]!=TRUE )
             {   sum=D[w] + G[w][v] ;
                 if (sum < D[v] )  {  D[v] = sum ;   p[v]=w; }
             }
   }
}
```

P

P[1]	P[2]	P[3]	P[4]	P[5]
1	1	4	1	3

```
void DisplayPath(int *P, int v)
{   //源点到v的最短路径
    if(P[v]!=1)
    {
       DisplayPath(P,P[v]);
       printf("%d--",P[v]);
     }
}
```

【例4-11】



$$C=\begin{pmatrix} \infty & \infty & 10 & \infty & 30 & 100 \\ \infty & \infty & 5 & \infty & \infty & \infty \\ \infty & \infty & \infty & 50 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & 20 & \infty & 60 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

D

D[1]	D[2]	D[3]	D[4]	D[5]	D[6]
∞	∞	10	∞	30	100

循环	S	w	D[2]	D[3]	D[4]	D[5]	D[6]
初态	{1}	-	∞	**10**	∞	30	100
1	{1,3}	3	∞	10	60	**30**	100
2	{1,3,5}	5	∞	10	**50**	30	90
3	{1,3,5,4}	4	∞	10	50	30	**60**
4	{1,3,5,4,6}	6	**∞**	10	50	30	60
5	{1,3,5,4,6,2}	2	∞	10	50	30	60

P[1]	1
P[2]	1
P[3]	1
P[4]	5
P[5]	1
P[6]	4

Prim 与 Dijkstra 算法对比：

区别	Prim算法构造最小生成树	Dijkstra算法构造单源最短路径
图类型	无向连通网	有向连通网
起始点	任选一个结点	有一个确定的起点（源点），其余为终点
连通顶点	连通所有顶点，且总造价最低	源点到各终点的两顶点间的最短路径
加入集合外顶点的修改方式	if (C[k][j] < LowCost[j] && LowCost[j] != INFINITY) { LowCost[j]=C[k][j]; CloseST[j]=k; }	sum=D[w] + G[w][v] ; if (sum < D[v]) { D[v] = sum ; p[v]=w; }
记录路径数组	无	有一个P数组记录从源点到各终点的路线
构成结果图	所构造的连通网的权值之和最小	一定是源点到终点的两路径权值最短
重复次数	重复n-1次	重复n-2次