数据结构与算法

Data Structures and Algorithms

第四部分 图

4.3 图的搜索算法

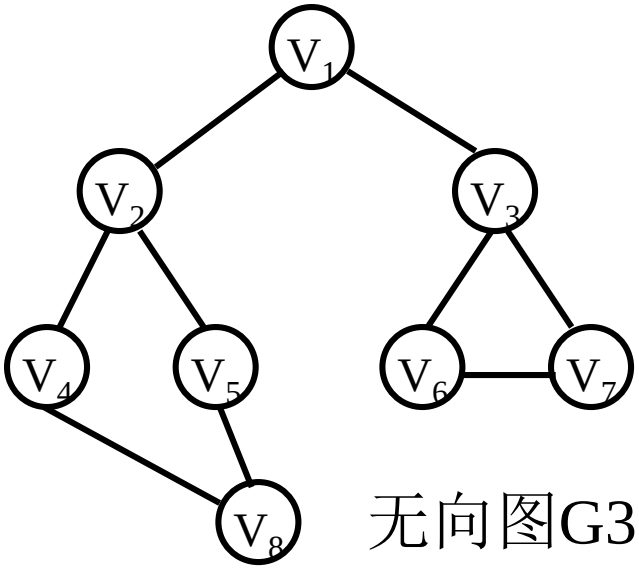
图的遍历 { 深度优先搜索DFS (Depth-First Search)
广度优先搜索BFS (Breadth-First Search)

遍历图注意问题：

- 确定遍历起点；
- 为保证非连通图的每一顶点都能被访问到，应轮换起点；
- 为避免顶点的重复访问，做访问标记。

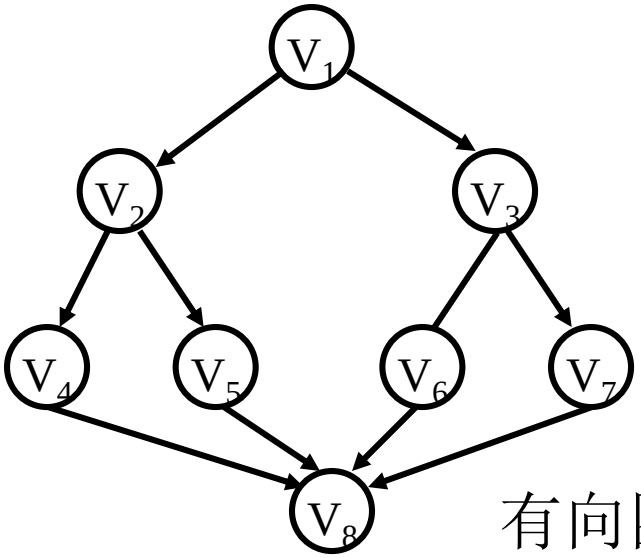
1、深度优先搜索DFS(Depth-First Search)

首先访问起点，然后依次访问与该起点相关联的每一个顶点，并以该关联顶点为新的起点，继续深度优先遍历。若图中还有未被访问的顶点，则换一个起点，继续深度优先遍历，直到所有的顶点都被访问。



无向图G3

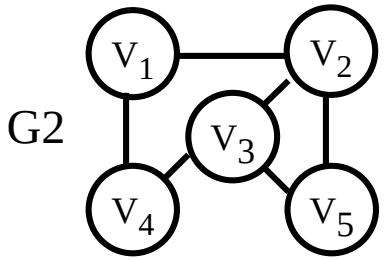
$V_1, V_3, V_6, V_7, V_2, V_4, V_8, V_5$



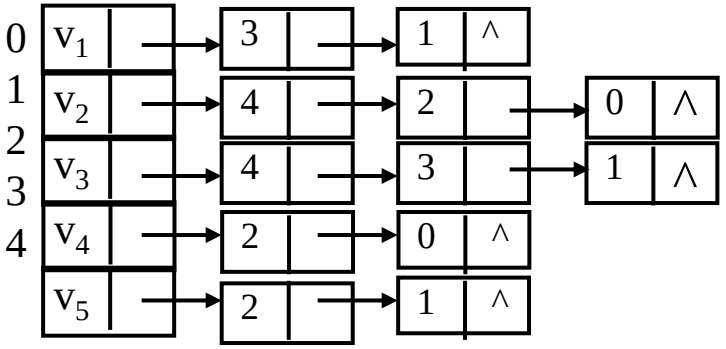
有向图G4

$V_1, V_2, V_4, V_8, V_5, V_3, V_6, V_7$

深度优先遍历：




```
Void   DFSTravers( GRAPH  G , v )
{
   For ( v = 0 ; v < G.vexnum ; ++v )  visited [ v ] = FALSE ;
   For ( v = 0 ; v < G.vexnum ; ++v )
         if ( !visited [ v ] )    DFS ( G , v );
}
```



无向图G2邻接表

```
Void  DFS(GRAPH  G , int   v )
{
   visited[ v ] = TRUE ;
   visitfunc ( v ) ;
   for ( w = FirstAdjVex ( G , v ); w; w = NextAdjVex( G , v , w );
       if ( !visited [w ] )     DFS( G , w );
}
```

图G2的深度优先遍历结果：V_1, V_4, V_3, V_5, V_2

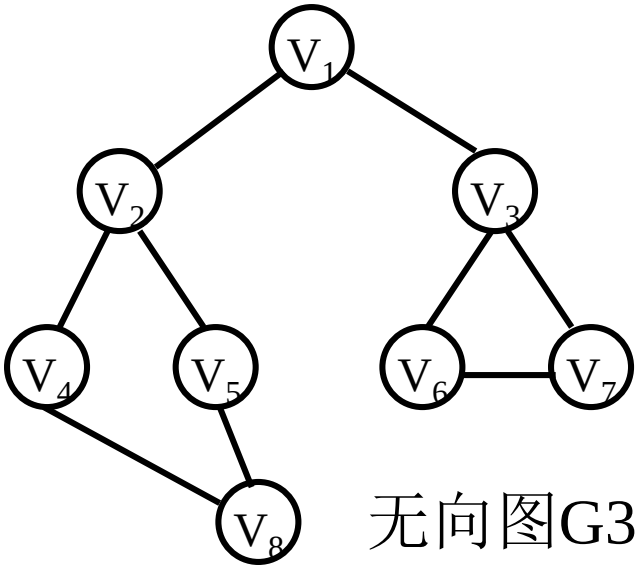
输入：L [v]表示无向图G的关于v的邻接表

输出：每个结点有先深编号的无向图和树边集 T

```
{
   T = ∅ ;    /*树边集开始为空*/
   count = 1 ; /*先深编号计数器*/
   for ( all v ∈ V )
   while ( there  exists a vertex  v ∈ V marked "new" )  dfs-search(v)
}
   void  dfs-search( v )
   {   dfn[ v ] = count ;                    /*对v编号*/
       count = count +1 ;
       mark  v "old"                         /*访问结点v*/
       for ( each  vertex w∈L[ v ] )
            if ( w is  marked "new"
               {   add ( v , w ) to T ;     /* ( v, w )是树边 */
                   dfs-search ( w ) ;       /* 递归搜索 */
                }
    }
```

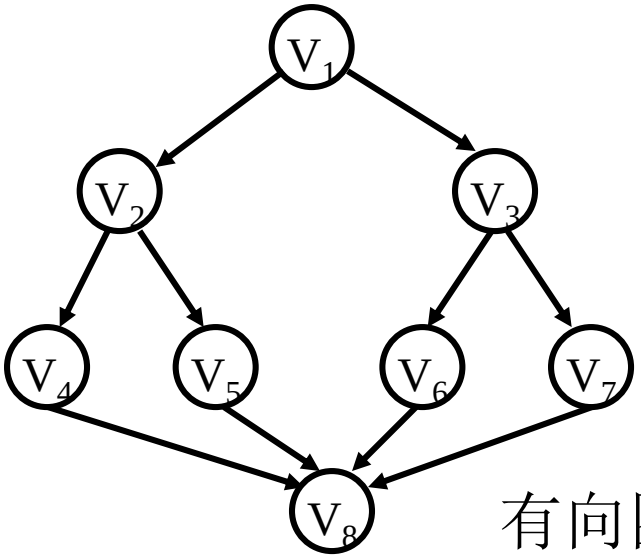
2、广度优先搜索BFS (Breadth-First Search)

首先访问起点，依次访问与该起点相关联的每一个邻接点，然后分别从这些邻接点出发访问它们的邻接点，并使“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”被访问，若图中还有未被访问的顶点，则换一个起点，继续广度优先遍历, 直到所有的顶点都被访问。



无向图G3

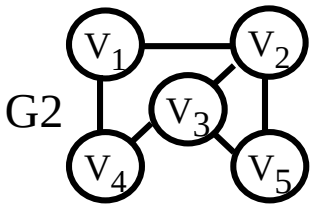
$V_1, V_2, V_3, V_4, V_5, V_6, V_7, V_8$

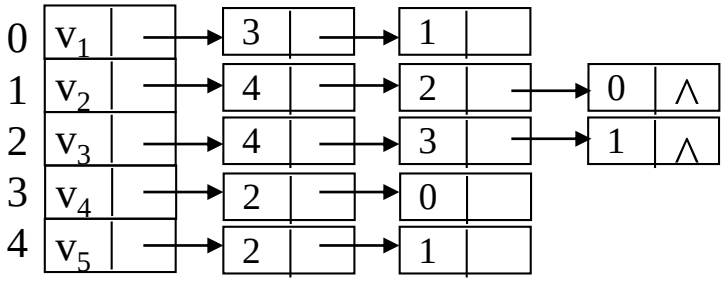


有向图G4

$V_1, V_2, V_3, V_4, V_5, V_6, V_7, V_8$

```
Void   BFSTravers( GRAPH  G , v )
{ For ( v = 0 ; v < G.vexnum ; ++v )  visited [ v ] = FALSE ;
   InitQueue( Q ) ;
   For ( v = 0 ; v < G.vexnum ; ++v )
     if ( !visited [ v ] )
       {  EnQueue ( Q , v );
          while ( !QueueEmpty ( Q ) )
                {  DeQueue ( Q , u ) ;
                   for ( w = FirstAdjVex ( G , u ); w;
                                          w = NextAdjVex( G , u , w )
                     if ( !visited [w ] ) {  visited[ w ] = TRUE ;
                                             visit( w ) ;
                                            EnQueue( Q , w );  }
                }
       }
}
```





无向图G2邻接表

图G2的广度优先遍历结果：V_1,V_4,V_2,V_3,V_5

输入：L [v]表示无向图G的关于v的邻接表

输出：每个结点有先广编号的无向图和树边集 T

```
Void   bfs-search ( v )
{  MakeNull ( Q ) ;
   bfn[ v ] = count ;                          /*先广编号*/
   count = count + 1 ;
   mark v "old"
   EnQueue ( v , Q ) ;                         /*v入队*/
   while ( !Empty( Q ) )
    {   v = Front ( Q ) ;
        DeQueue ( Q ) ;
        for ( each w∈L[ v ] )
            {  bfn[ w ] = count ;              /*先广编号*/
               count = count + 1 ;
               mark  w "old" ;
               EnQueue( w , Q ) ;
               Insert( (v , w ) , T ) ;   }    }  /*树边入队*/
}
```

图 的存储结构：　邻接矩阵 + 邻接表

图的遍历/搜索：　深度优先搜索（DFS）, dfn(*i*)

广度优先搜索（BFS）, bfn(*i*)

有向图十字链表（邻接表+逆邻接表）

无向图的邻接多重表 （邻接表压缩）

深度优先生成树　　广度优先生成树

顶点集S: $S[i]=\begin{cases}1 & \text{顶点 } i \in S\\ 0 & \text{顶点 } i \notin S\end{cases}$

*i*为顶点编号，$i=0..n-1$

边集T: $T[i][j]=\begin{cases}1 & \text{边 } (i,j) \in T\\ 0 & \text{边 } (i,j) \notin T\end{cases}$

*i,j*为顶点编号，$i,j=0..n-1$

思考题:

1、图的路径问题

(1)无向图两点之间是否有路径存在?

(2)有向图两点之间是否有路径存在?

(3)如果有路径,路径经过哪些顶点?

2、图的环路问题

(1)无向图是否存在环路?

(2)有向图是否存在环路?

(3)有几条环路?

(4)环路经过哪些点,环路轨迹是什么?

参考算法1-1：判断是否存在从u到v的路径，返回1或0

```
int  ExistPathDfs1(ALGraph G,int *visited,int u,int v)
{
   ArcNode *p;
   int w;
   if(u==v)     return 1;
   else
    {
        visited[u]=1;                    //访问标志
        for(p=G.vertices[u].firstarc;p;p=p->nextarc)
        {
            w=p->adjvex;
            if(!visited[w]&&ExistPathDfs1(G,visited,w,v))
            return 1;
         }//for
    }//else
    return(0);
}//ExistPathDfs1
```

参考算法1-2：判断u到v是否有通路,返回1或0

```
int ExistPathDfs2(ALGraph G,int *visited,int u,int v)
{    ArcNode *p;
     int w;
     int static flag=0;
     visited[u] = 1;                    //访问标志
     p = G.vertices[u].firstarc;  //第一个邻接点
     while(p!=NULL)
     {    w = p->adjvex;
          if(v==w)
          {   flag = 1;
              return (1);    }         // u和v有通路
           if(!visited[w])
          ExistPathDfs2(G,visited,w,v);
          p=p->nextarc;
      }//while
      if(!flag)  return(0) ;
}  //ExistPathDfs2
```

参考算法2：求u到v所有简单路径

```
int  FindAllPath(ALGraph G,int *visited,int *path,int u,int v,int k)
{   ArcNode *p;   int static paths=0;      //paths控制指输出第几条有效路径
    int n,i;
    path[k]=u;       visited[u]=1;
    if(u==v)
    {   if(path[1])
        {   if(!paths)  printf("找到如下路径：\n");
            paths++;
            printf("路径%d：  %d",paths,path[0]);
            for(i=1;path[i];i++)  printf("--%d",path[i]);   printf("\n");   }
     }
    else
        for(p=G.vertices[u].firstarc;p;p=p->nextarc)
        {  n=p->adjvex;
           if(!visited[n])
           FindAllPath(G,visited,path,n,v,k+1);      }
           for(i=1;i<=G.vexnum;i++)
           {     visited[i]=0;
                 path[i]=0;      }
    return(paths);
}
```

//path[0]为路径起点，从path[1]开始为路径中个顶点，若不存在路径，则从path[1]起均为0

调用：FindAllPath(G,visited,path,u,v,0))

参考算法3：判断图是否有回路存在

是否存在包含顶点u的回路

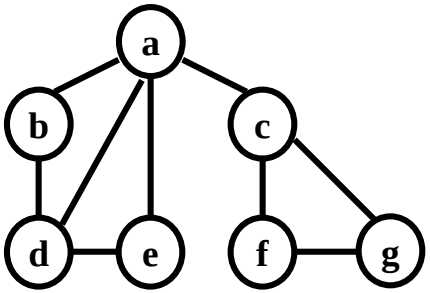
```
int Cycle(ALGraph G, int *visited, int u)
{   ArcNode *p;
    int w;
    int flag =0;
    visited[u]=1;
    p=G.vertices[u].firstarc;
    while(p&&!flag)
    {  w=p->adjvex;
       if(visited[w]!=1)  //w未访问过（0）或访问且其邻接点已访问完（2）
       {  if(!visited[w])  //w未访问过，从w开始继续dfs
          flag=Cycle(G,visited,w);  }
        else                    //顶点w已经访问过，并且其邻接点已经访问完
           flag=1;
       p=p->nextarc;
    }
    visited[u]=2;
    return(flag);
}
```

判断图是否有回路

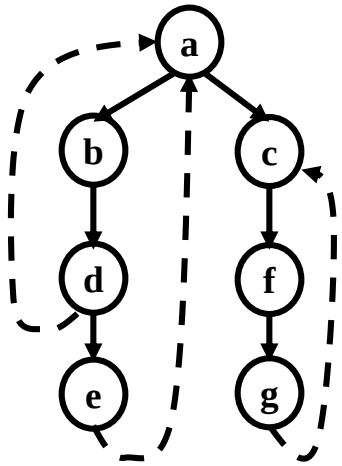
```
void IsCycle(ALGraph G)
{
    int visited[MAX_VERTEX_NUM],u, CycleFlag;
    for(u=1;u<=G.vexnum;u++)
        visited[u]=0;
    for(u=1;u<=G.vexnum;u++)
        if(!visited[u])
        {
            CycleFlag=Cycle(G,visited,u);
            if(CycleFlag)  break;
        }
    if(CycleFlag)
        printf("图中存在回路！\n");
    else
        printf("图中不存在回路！\n");
}
```

4.4 图与树的联系

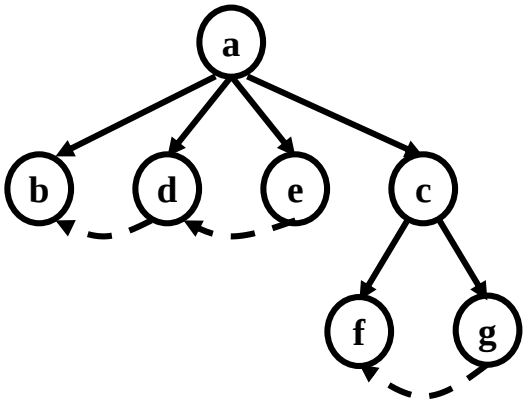
4.4.1 先深生成森林和先广生成森林



(a)



(b)



(c)

树边(编号从小到大) 与 非树边：回退边/横边(编号从大到小)
先深(广)搜索过程中，由树边和树边所连接的顶点组成的子图，称为图的先深(广)生成森林。

无向连通图通过先深(广)搜索只能得到一个先深(广)生成树。

非连通图则可得到多个生成树， 连通子图（连通分量）。

4.4.2 无向图与开放树

【定义】 连通而无环路的无向图称作开放树（Free Tree）。

开放树的性质：

（1）具有n≥1个顶点的开放树包含n-1条边；

（2）如果在开放树中任意加上一条边，便得到一条回路。

(证明见教材P133~134)

4.4.3 最小生成树

设G=(V, E)是一个连通图，E中每一条边(u, v)的权为C(u, v)，也叫做边长。图G的一株生成树(spanning tree)是连接V中所有结点的一株开放树。将生成树中所有边长之总和称为生成树的价(cost)。使这个价最小的生成树称为图G的最小生成树(minimum-cost spanning tree)。

MST性质

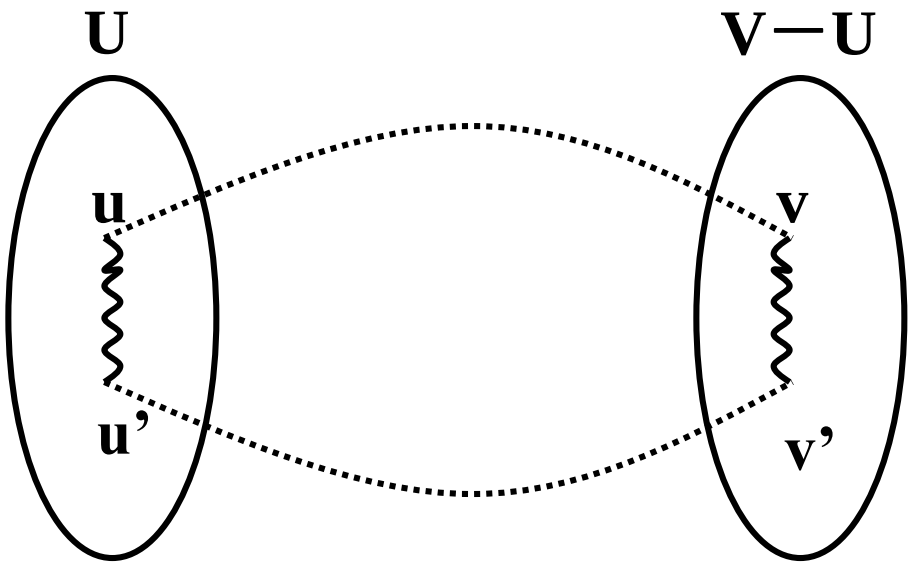
【描述1】设G=(V, E)是一个连通图，在E上定义一个权函数，且{ (V_1, T_1), (V_2, T_2), …, (V_k, T_k) } 是 G的任意生成森林。令:

$$T = \bigcup_{i=1}^{k} T_i \quad (k > 1)$$

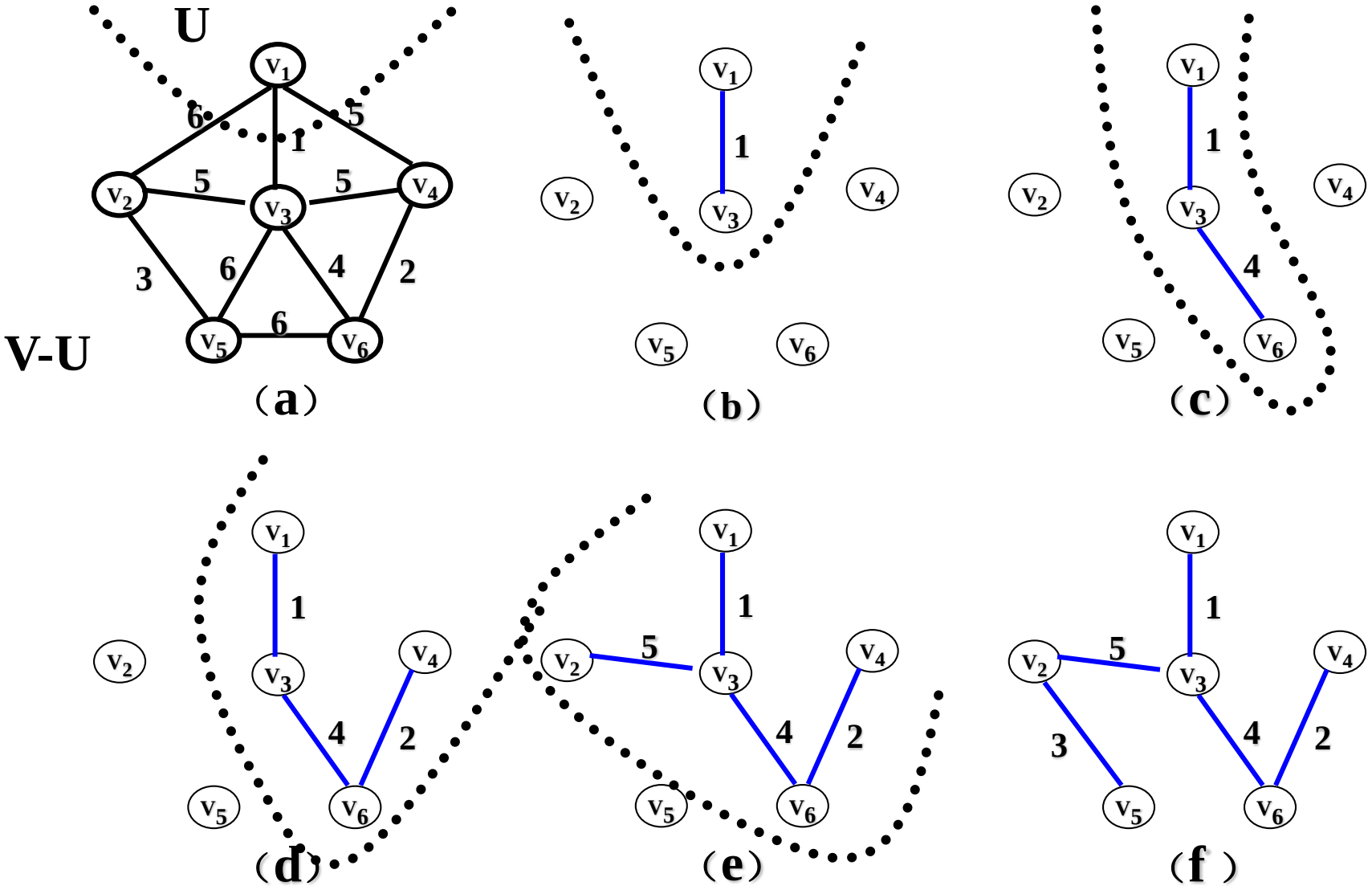
又设e=(v, w)是E-T中这样一条边，其权C[v,w]最小，而且v∈V_1和w∈V_1。则图G有一棵包含 T ∪{e} 的生成树，其价不大于包含T的任何生成树的价。

【描述2】假设N=(V, {E})是一个连通网，U是顶点V的一个非空子集。若(u, v)是一条具有最小权值（代价）的边，其中u∈U, v ∈V-U, 则必存在一棵包含边(u,v)的最小生成树。

MST 性质证明



假设网N的任何一棵最小生成树都不包含$(u,v)_{min}$，设 T 是连通网上的一棵最小生成树，当将边$(u,v)_{min}$加入到 T中时，由生成树的定义，T 中必包含一条$(u,v)_{min}$的回路。另一方面，由于 T 是生成树，则在T上必存在另一条边(u',v')，且u和u'、v和v'之间均有路径相同。删去边(u',v')便可消去上述回路，同时得到另一棵最小生成树T'。但因为$(u, v)_{min}$的代价不高于(u',v')，则T'的代价亦不高于T，T'是包含$(u, v)_{min}$的一棵最小生成树。



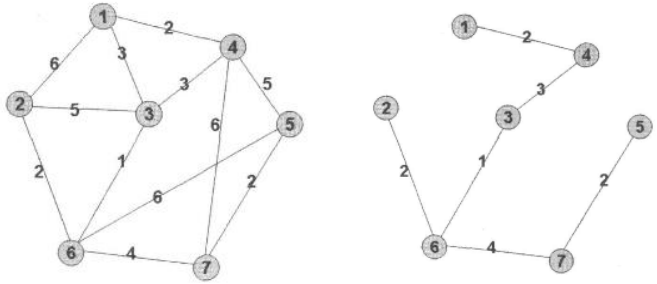
求最小生成树

很多关于最小成本的问题都可以通过求带权图的最小生成树来解决。

【问题1】高速公路问题：假设有N个城市，每条公路可以连接两个城市。目前原有的公路有m条，但是不能实现所有城市之间的连通，因此需要继续修建公路，在费用最低的原则下，实现N个城市的连通，还需要修建哪些条公路。由于修路的费用与公路的长短是成正比的，所以这个问题就可以转化成求修建哪几条公路能够实现所有城市的连通，同时满足所修公路总长最短。

【问题2】在n个城市间建立通信网络，需架设n-1条线路。求解如何以最低经济代价建设此通信网。

【问题3】某市区有七个住宅小区需要铺设天然气管道,各小区的位置及它们之间可修建管道路线与费用如图所示。现要设计一个管道铺设路线,要求天然气能输送到各个小区并且修建的总费用为最小,这就是求最小生成树的问题。



（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

要点： 引入集合U和T。U准备结点集，T为树边集。
初值U={1}，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 = { 1 };
       while ( ( U – V )  != ∅ )
         {   设( u, v ) 是使u∈U与v∈(V-U)且权最小的边 ;
             T = T∪{ ( u, v ) } ;
             U = U ∪ { v };
         }
 } ;
```

引入辅助向量：

CloseST[] 和 LowCost，其中：

CloseST[*i*] 为 U 中的一个顶点

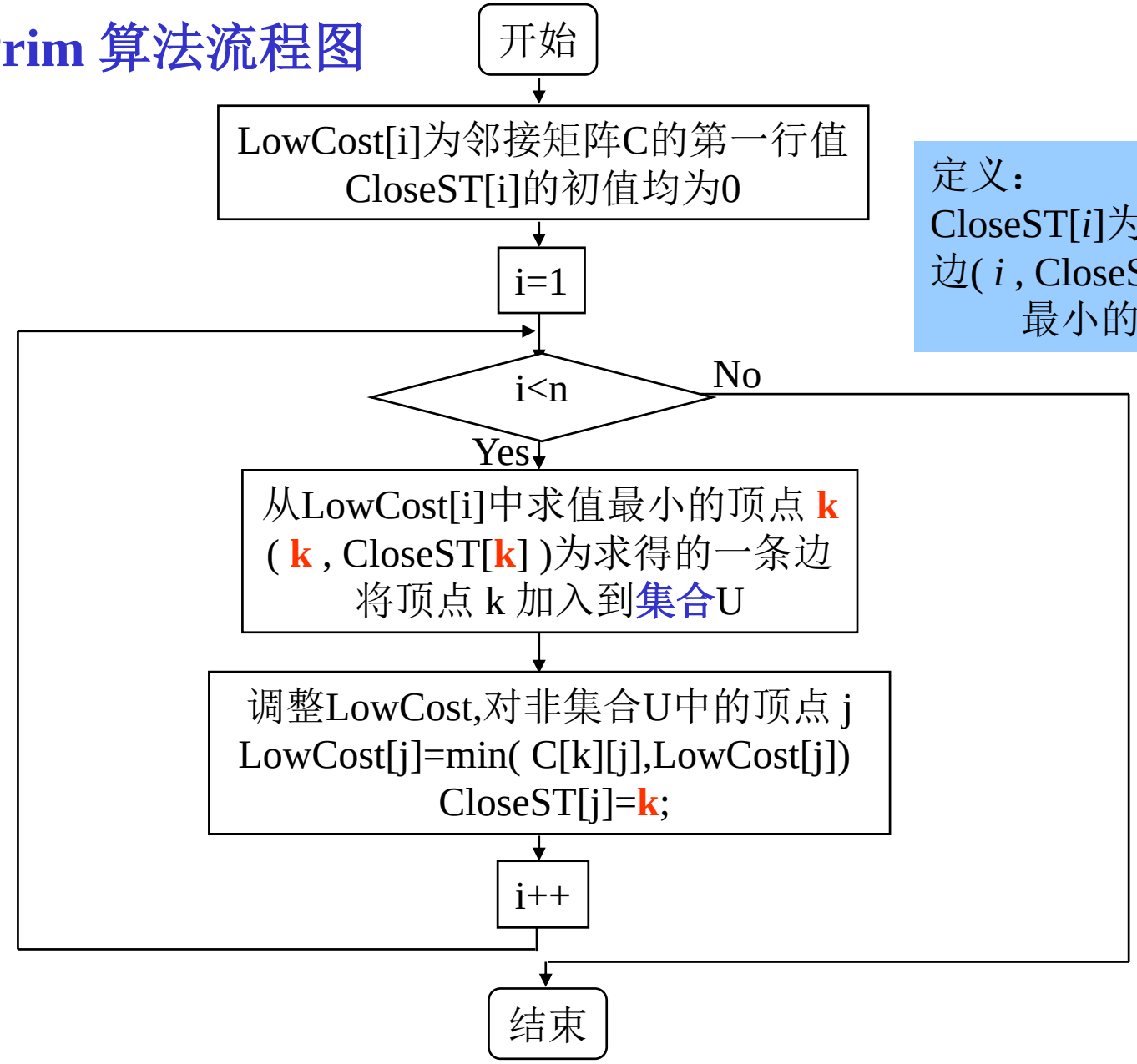
边 (*i* , CloseST[*i*]) 具有最小的权 LowCost[*i*]；

CloseST 和 LowCost 的初值是多少？

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

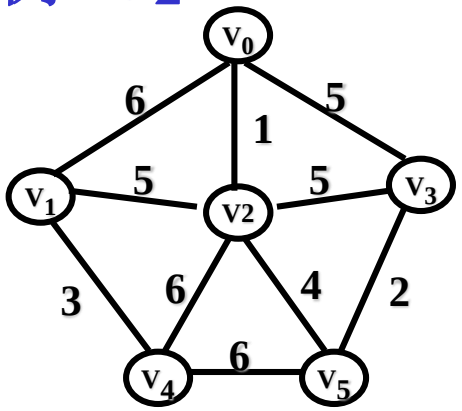


定义:
CloseST[*i*]为U中的一个顶点
边(*i* , CloseST[*i*]) 具有
最小的权 LowCost[*i*]

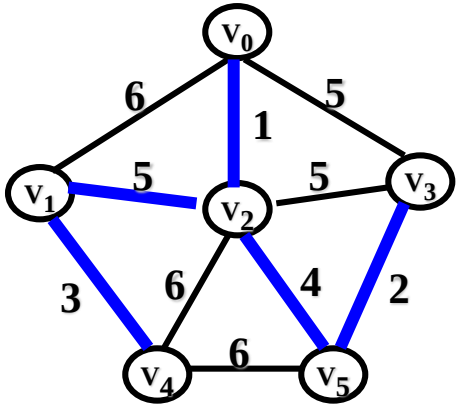
```
Void Prim( C )
Costtype C[n][n] ;
{ costtype LowCost[n];  int CloseST[n];  int i,j,k;  costtype min;
  for( i=1; i<n; i++ )
    {   LowCost[i] = C[0][i];     CloseST[i] = 0;   }   // 赋初值, U中顶点都是0
  for( i = 1; i < n; i++ )
    {    min = LowCost[i];
         k = i;
         for( j = 1; j < n; j++ )
           if ( LowCost[j] < min )
              { min = LowCost[j] ;    k=j; } //求离U中某一顶点最近的顶点
         Cout << "(" << k << "," << CloseST[k] << ")" << end1;
         LowCost[k] = INFINITY ;      //将k加入集合U
         for ( j = 1; 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	0	1	2	3	4	5
0	∞	6	1	5	∞	∞
1	6	∞	5	∞	3	∞
2	1	5	∞	5	6	4
3	5	∞	5	∞	∞	2
4	∞	3	6	∞	∞	6
5	∞	∞	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]

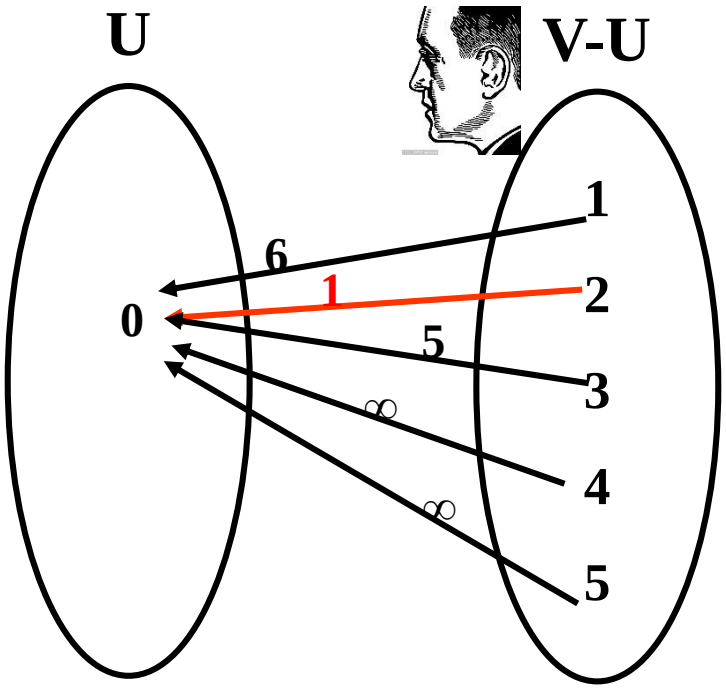
i =	0	1	2	3	4	5
0	--	6	1	5	∞	∞
1		5	∞	5	6	4
2		5	∞	2	6	∞
3		5	∞	∞	6	∞
4		∞	∞	∞	3	∞
5		∞	∞	∞	∞	∞

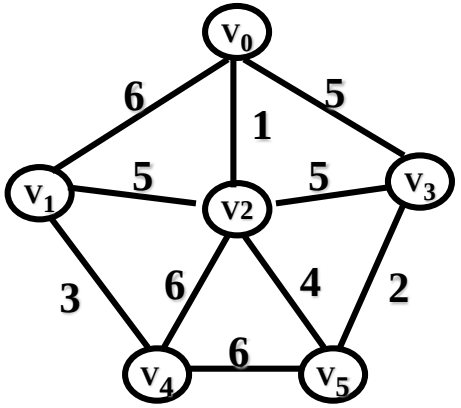
CloseST[i]

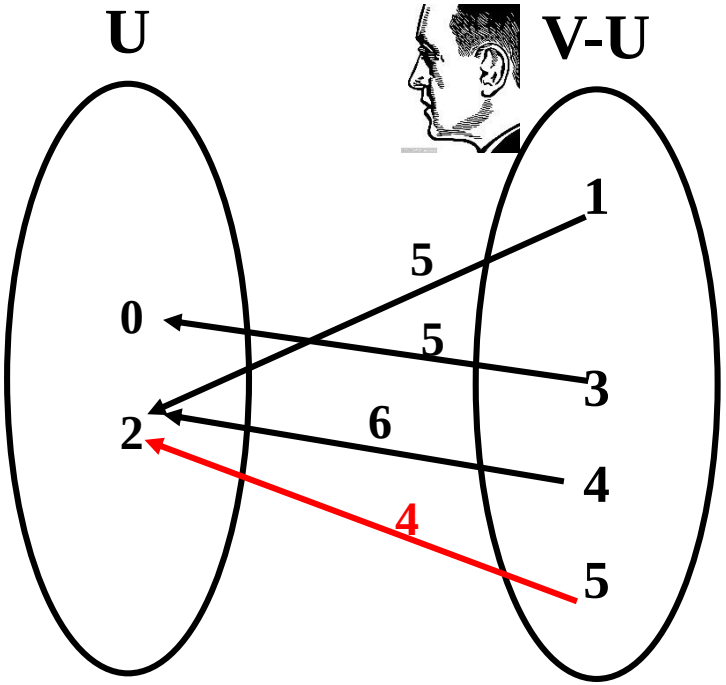
i =	0	1	2	3	4	5
0	--	1	1	1	1	1
1		3	1	1	3	3
2		3	1	6	3	3
3		3	1	6	3	3
4		3	1	6	2	3
5		3	1	6	2	2

打印边

	K	(K , CloseST[K])
0	K	(K , CloseST[K])
1	3	(3 , 1)
2	6	(6 , 3)
3	4	(4 , 6)
4	2	(2 , 3)
5	5	(5 , 2)


U
V-U
0
6
1
5
∞
∞
1
2
3
4
5


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


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

（2）求最小生成树 —— Kruskal 算法

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；

(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。

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。

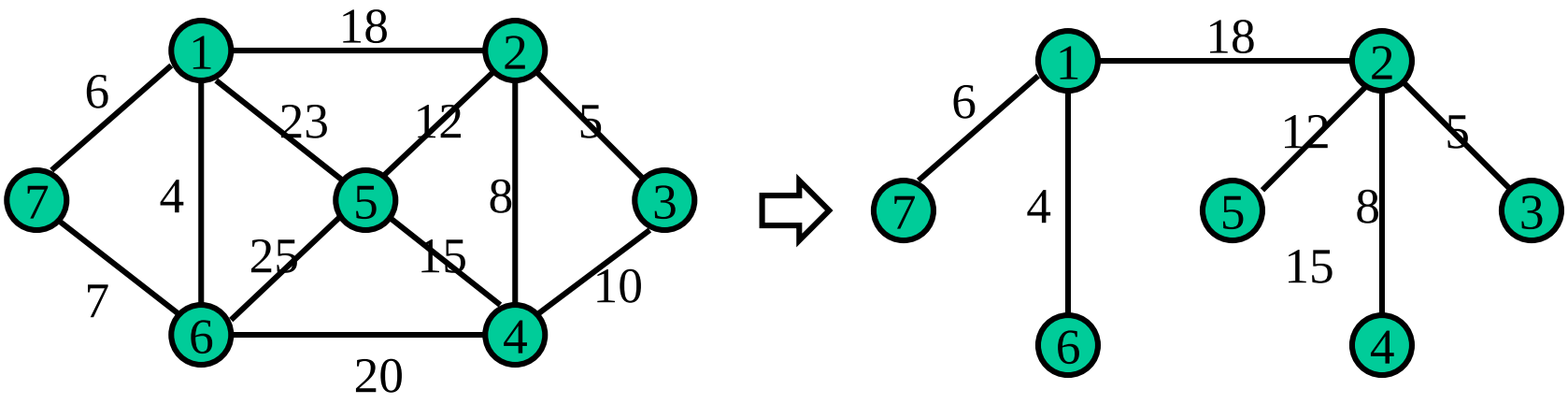
输出：G的最小生成树

```
Void   Kruskal ( V, T )
{    T = V ;
     ncomp = n ;    /*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 -- ;
                  }
           }
}
```

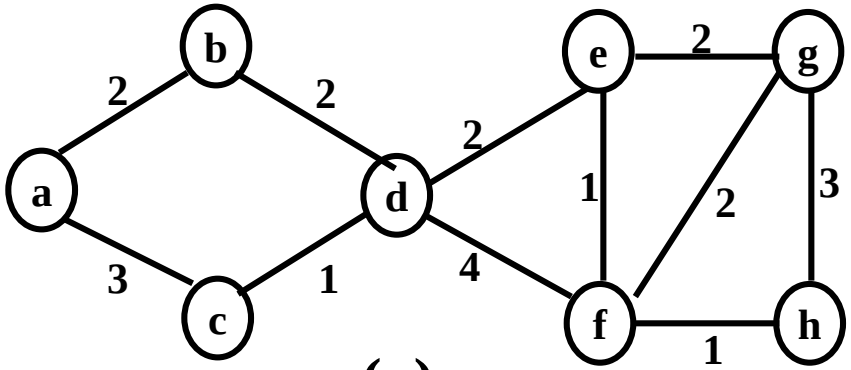
Prim算法与Kruskal算法的比较：

- 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。

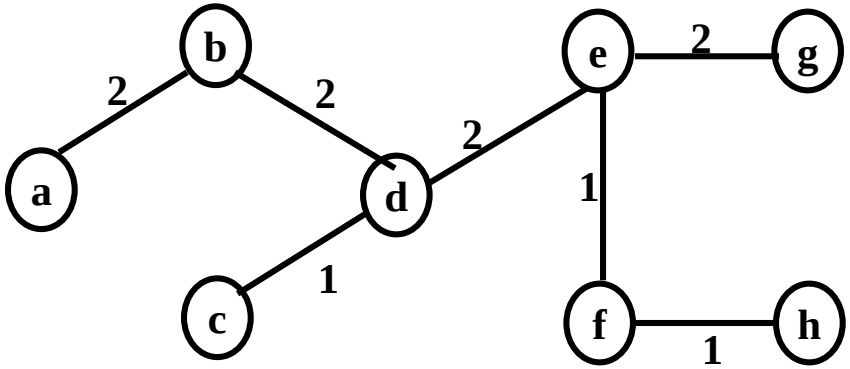
【例4-4】求最小生成树



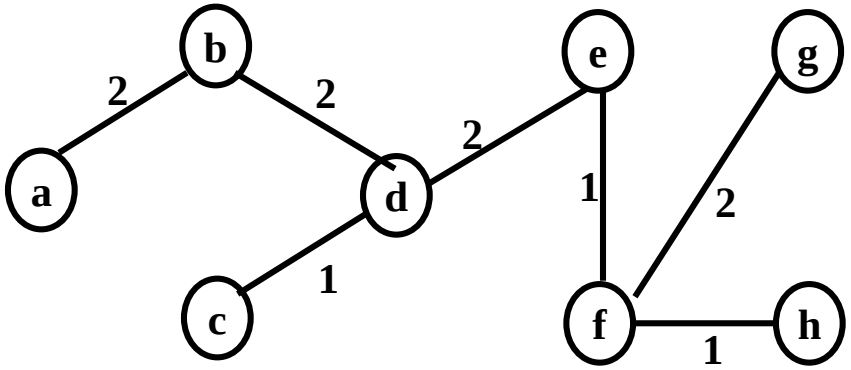
【例4-5】求最小生成树



(a)



(b)



(c)