第七章 图

中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

本章节目录

7.1图的基本概念

- 每个结点代表一个数据元素，数据元素间满足某种关系就认为对应的结点之间有一条边，认为他们是相邻的。
- 图和树一样，也是一种**非线性**的数据结构，但比树形结构更复杂。在图状结构中，任意两个结点之间都可能相关，即结点之间的邻接关系可以是任意的。
- 有向图和无向图

7.1.1图的抽象数据类型的定义

- 图(Graph)是由**非空的顶点集合**和一个描述顶点之间关系，即边（或者弧）的集合组成。

7.1.1 图的抽象数据类型的定义

ADT Graph{

数据对象：V是具有**相同特性**的数据元素的集合，称为**顶点集**。

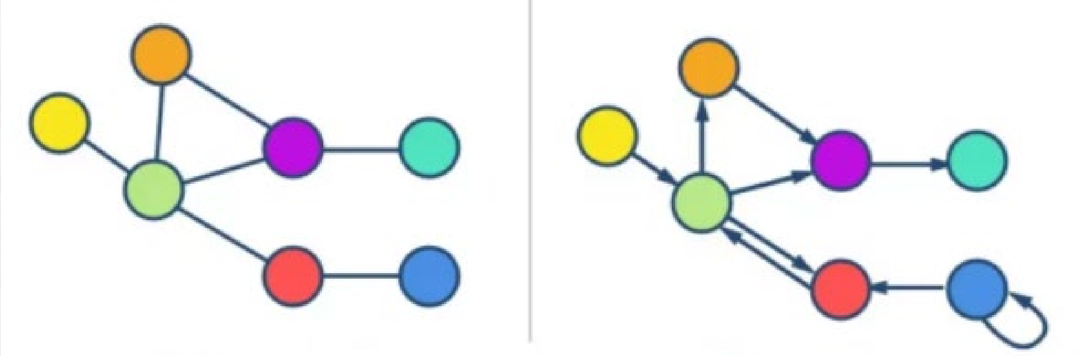
数据关系：R={E}， E={<vi,vj>|vi,vj∈V∧P(vi,vj)，<vi,vj>表示从vi到vj的弧，谓词P(vi,vj)定义了弧<vi,vj>的意义}

基本操作：产生图，销毁图，查找某个结点，插入一个结点，删除一个结点及对应的弧，插入弧，删除弧，遍历图

}

有向图？

7.1.2 图的基本术语



（1）顶点、弧、边

（2）有向图、对称图

（3）无向图

（4）无向完全图：任何两个顶点都有边

（5）有向完全图

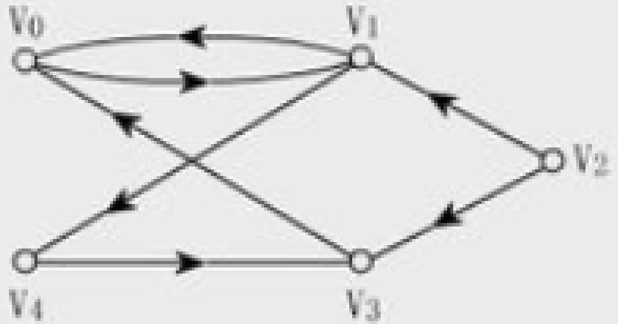
（6）稠密图：接近完全图的图

（7）稀疏图：边很少的图

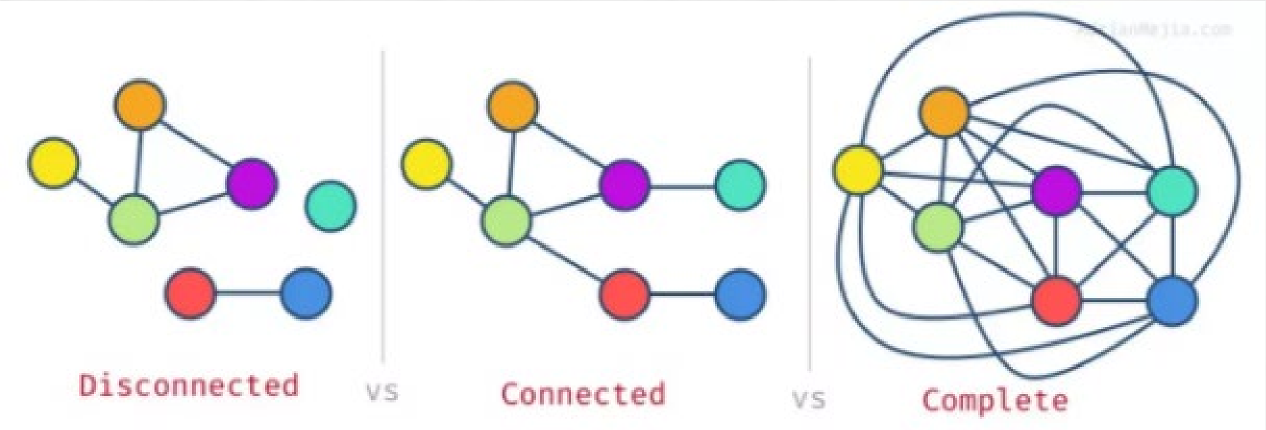
边数、顶点数和度间的关系？

（8）顶点的度：连接顶点的边数

（9）顶点的入度、出度：有向图



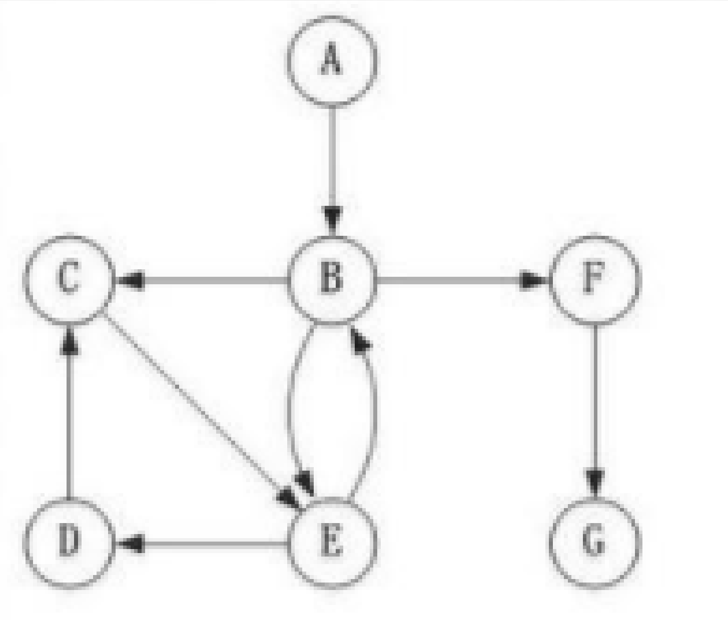
- 边的权、网图/网络
- 路径：顶点序列，相邻顶点有边
- 路径长度：边的数量
- 简单路径：顶点不重复的路径
- 回路：起止点相同
- 简单回路（除起止点外其他节点均不同）
- 子图 ：顶点子集+边子集
- 无向图：顶点连通、连通图、连通分量（无向图的极大连通子图）



●**有向图：**

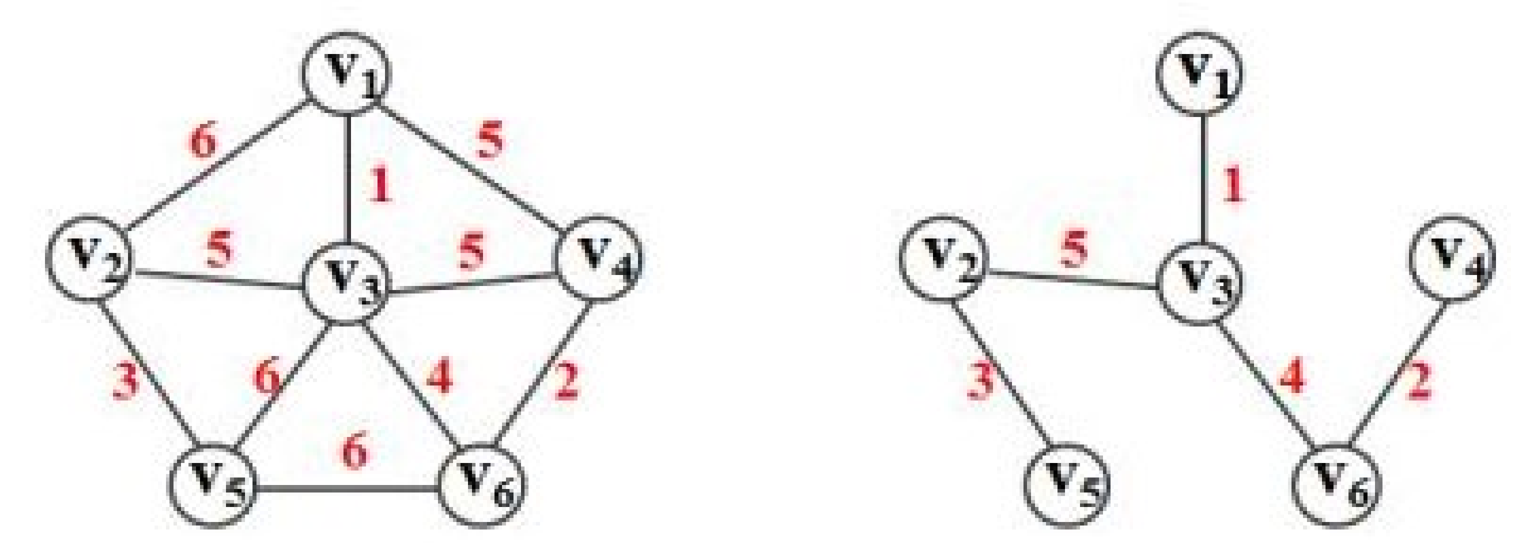
强连通图：任何两个顶点均有一条路径

强连通分量 （有向图的极大强连通子图）



- 生成树：包括所有结点的极小连通子图，具有n-1条边，若再增加一条边将产生回路，若去掉一条边则不连通。

- 一个图存在一棵生成树，它一定是连通的吗？



7.2图的存储结构

存储图中结点及其关系

◆**邻接矩阵：**两个数据类型

一维数组：存储图中顶点的信息

矩阵：表示图中各顶点之间的邻接关系

◆**邻接表：**两个数据类型

一维数组：存储图中顶点的信息

链表：表示图中各顶点之间的邻接关系

7.2.1邻接矩阵

◆用一维数组存储图中顶点的信息，用矩阵表示图中各顶点之间的邻接关系。

◆假设图G＝（V，E）有n个确定的顶点，即V＝$\{v_0,v_1,\ldots,v_{n-1}\}$，则表示G中各顶点相邻关系为一个n×n的矩阵A，矩阵的元素为：

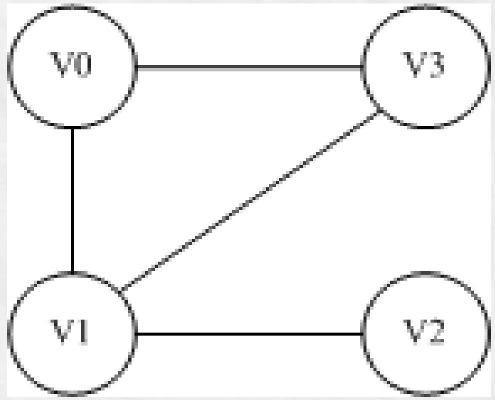
$$A[i][j]=\begin{cases}1，若（v_i, v_j）或<v_j, v_i>是E(G)中的边 \\ 0，若（v_i, v_j）或<v_j, v_i>不是E(G)中的边\end{cases}$$

若G是网图，则邻接矩阵可定义为：

$$A[i][j]=\begin{cases}w_{ij}，\quad 若（v_i,v_j）或<v_j,v_i>是E(G)中的边 \\ 0或\infty，若（v_i,v_j）或<v_j,v_i>不是E(G)中的边\end{cases}$$

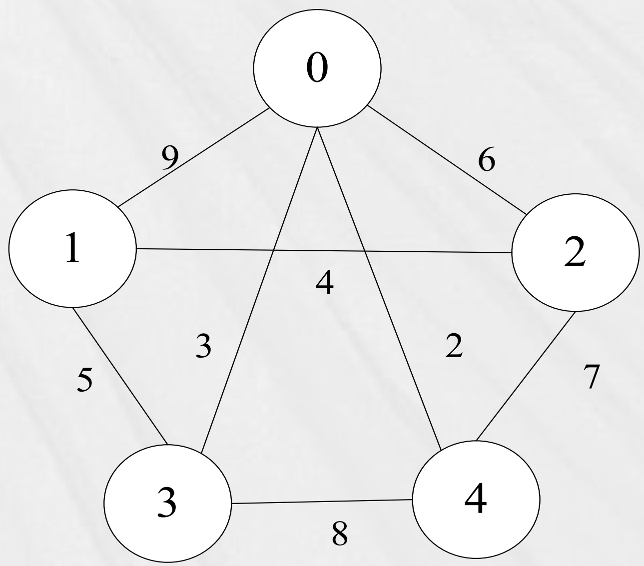
其中，w_{ij}表示边(v_i,v_j)或$<v_j,v_i>$上的权值；∞表示一个计算机允许的、大于所有边上权值的数。

【例1】用邻接矩阵表示法表示图



0	1	0	1
1	0	1	1
0	1	0	0
1	1	0	0

【例2】用邻接矩阵表示法表示网图



∞	9	6	3	2
9	∞	4	5	∞
6	4	∞	∞	7
3	5	∞	∞	8
2	∞	7	8	∞

- 无向图的邻接矩阵一定是一个对称矩阵。因此，在存放邻接矩阵时只需存放上（或下）三角矩阵的元素即可。
- 对于无向图，邻接矩阵的第i行（或第i列）非零元素（或非∞元素）的个数正好是第i个顶点的度TD(v_i)。
- 对于有向图，邻接矩阵的第i行（或第i列）非零元素（或非∞元素）的个数正好是第i个顶点的出度OD(v_i)（或入度ID(v_i)）。
- 用邻接矩阵方法存储图，很容易确定图中任意两个顶点之间是否有边相连；但是，要确定图中有多少条边，则必须按行、按列对每个元素进行检测，所花费的时间代价很大。

❖ 图的邻接矩阵存储表示

```
#define  MaxVertexNum  100         // 最大顶点数设为100
  typedef  char  VertexType;       // 顶点类型设为字符型
  typedef  int  EdgeType;          // 边的权值设为整型
  typedef  struct {
    VertexType  vexs[MaxVertexNum];  // 顶点表
    EdgeType   edges[MaxVertexNum][MaxVertexNum];
                                   // 邻接矩阵，即边表
    int  n, e;                     // 顶点数和边数
  }  Mgragh;                       // Maragh是以邻接矩阵存储的图类型
```

❖ 建立一个图的邻接矩阵存储的算法

```
void CreateMGraph(MGraph   *G)
 {
   int i, j, k;
   printf("请输入顶点数和边数\n");
   scanf("%d,%d",&(G->n),&(G->e));        // 输入顶点数和边数
   printf("请输入顶点信息\n");
   for (i=0;  i<G->n;  i++)    scanf("%c",  &(G->vexs[i]));
   for (i=0;i<G->n;i++)
        for (j=0;j<G->n;j++)   G->edges[i][j]=0;    // 初始化邻接矩阵
```

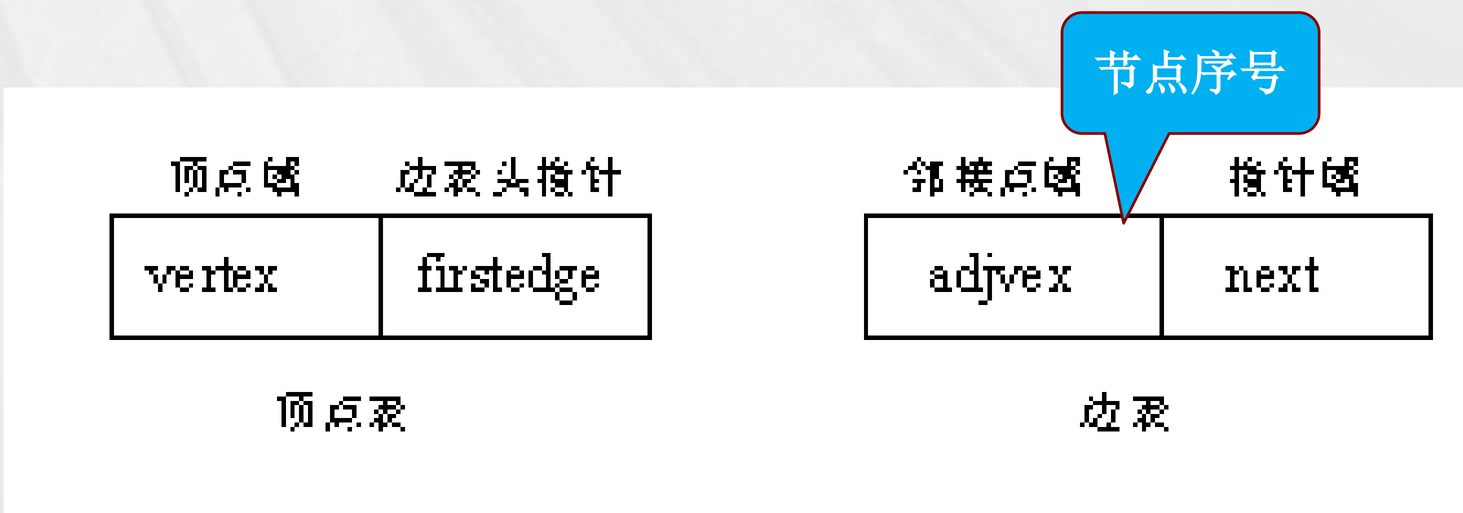
```
printf("请输入每条边对应的两个顶点的序号:\n");
 for (k=0;  k<G->e;  k++)
   {
        scanf(“%d, %d”,&i, &j);    // 输入e条边
        G->edges[i][j]=1;
      }
}//CreateMGraph
```

7.2.2 邻接表

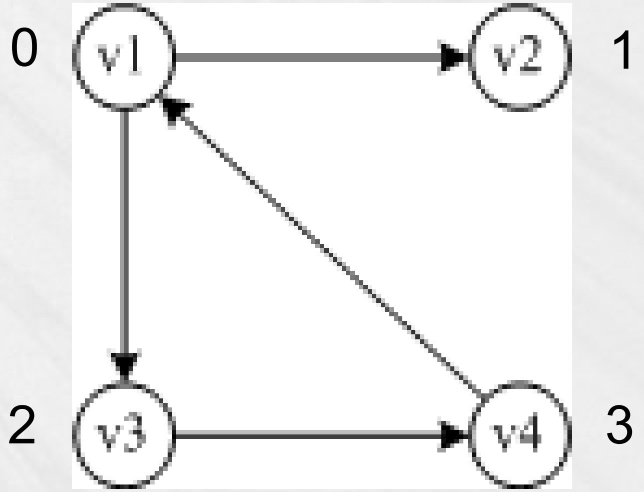
❖ **邻接表(Adjacency List)**是图的一种顺序存储与链式存储相结合的存储方法。

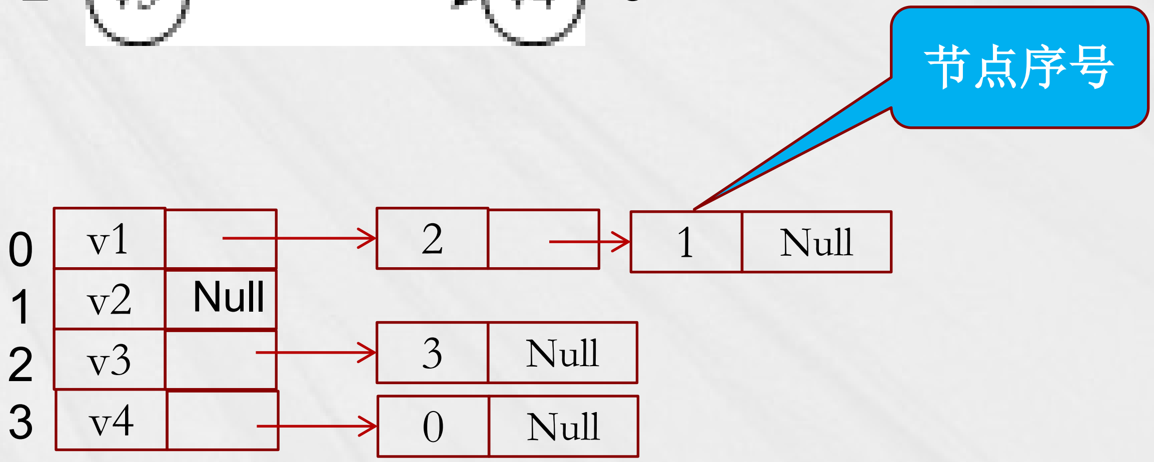
❖ 对于图G中的每个顶点v_i，将所有邻接于v_i的顶点v_j链成一个单链表，这个单链表就称为**顶点v_i的邻接表**，再将所有点的邻接表表头放到顶点表的对应数组元素的指针域中，就构成了图的邻接表。

❖ 在邻接表表示中有两种结点结构：



邻接表表示

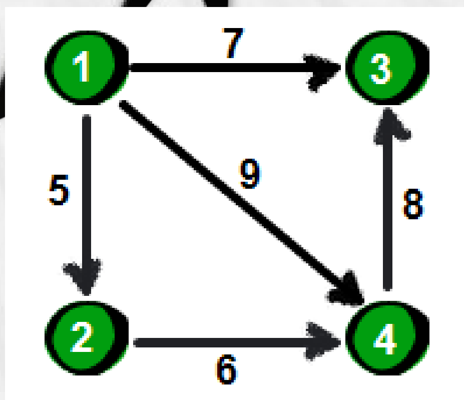


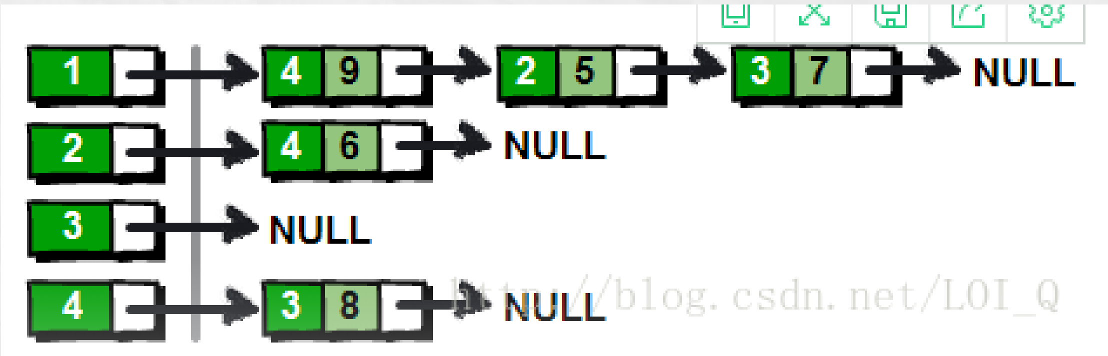


节点表

边表：邻接表

复杂的边表：邻接表

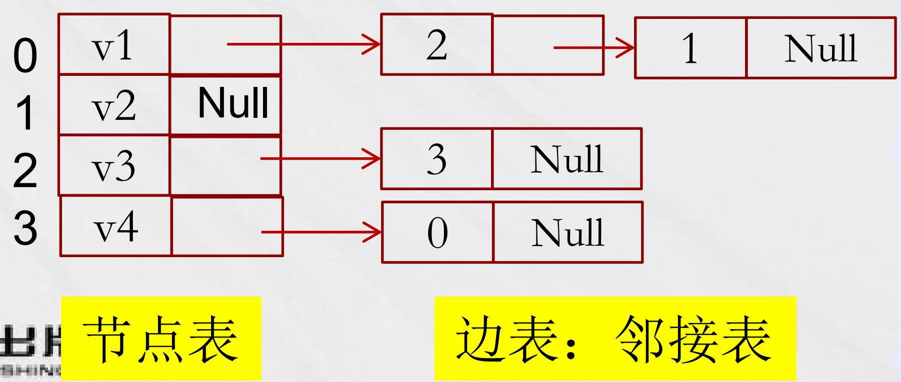




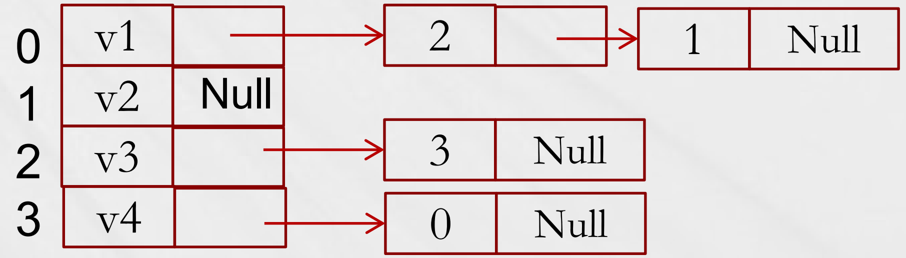
邻接顶点编号	边上的权值	指向下一个邻接顶点的指针

- 邻接表表示的形式描述如下：

```
#define MaxVerNum 100          // 最大顶点数为100

//邻接表—边表中的结点
typedef struct node
    {
      int    adjvex;                  // 邻接顶点域
      struct   node   * next;        // 指向下一个邻接点的指针域
     }    EdgeNode;
```



```
// 顶点表
typedef   struct vnode
 {
        VertexType    vertex;          // 顶点域
        EdgeNode    * firstedge;   // 邻接表头指针
      }     VertexNode;

 //定义邻接表

 typedef  struct  {

      VertexNode   adjlist[MaxVertexNum];      // 节点表

       int   n,   e;               // 顶点数和边数

      }   ALGraph;          // 以邻接表方式存储的图类型
```

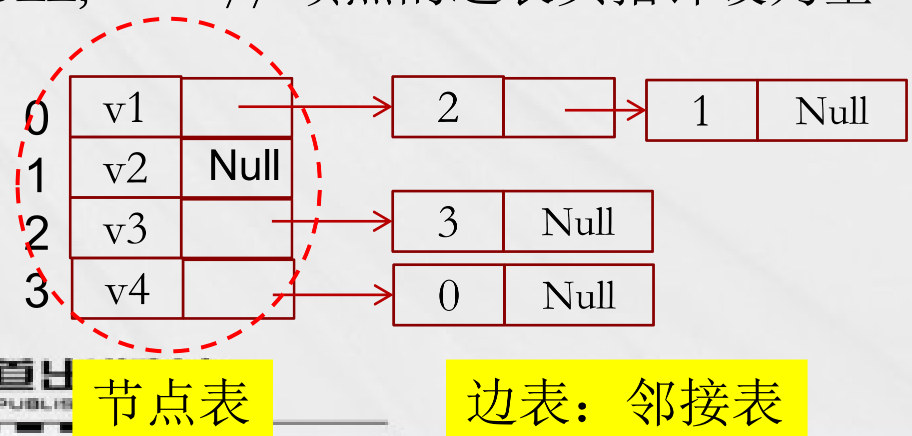


节点表

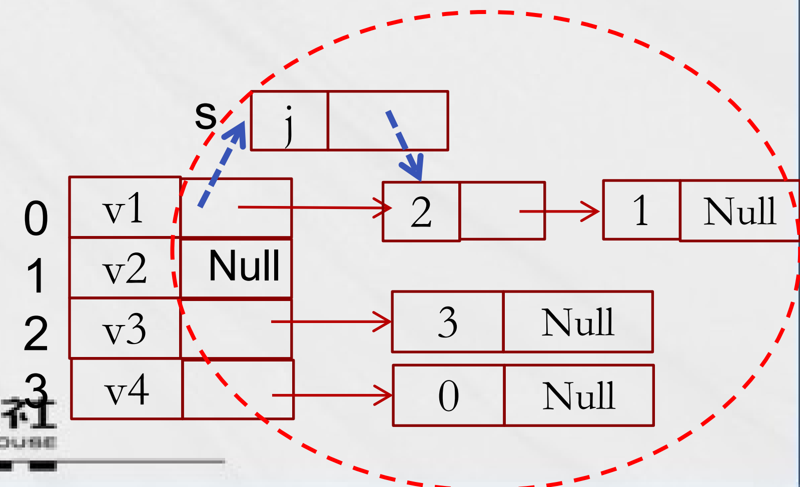
边表：邻接表

❖ 建立一个有向图的邻接表存储的算法：

```
void   CreateALGraph(ALGraph   *G)
  {   int i,j,k;
   EdgeNode   *s;
   printf("请输入顶点数和边数：\n");
   scanf("%d,%d",&(G->n),&(G->e));  //读入顶点数和边数
   printf("请输入顶点信息：\n");
   for (i=0; i<G->n; i++)              // 建立有n个顶点的顶点表
   { scanf("%c",&(G->adjlist[i].vertex));  // 读入顶点信息
    G->adjlist[i].firstedge=NULL;          // 顶点的边表头指针设为空
    }
```



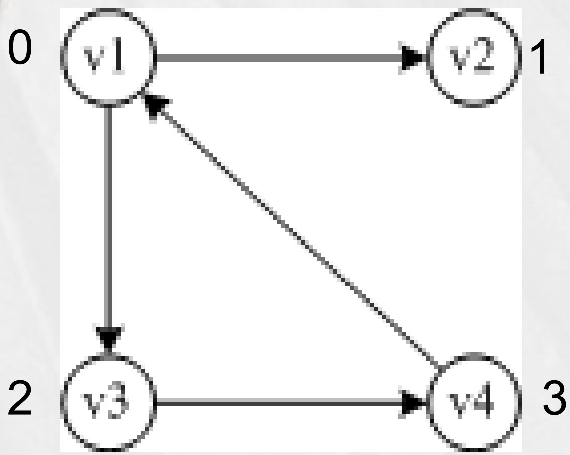
```
printf("请输入边的信息(输入格式为:i,j)：\n");
for (k=0;k<G->e;k++)           // 建立边表—顶点Vi的邻接表
{  scanf("%d,%d", &i, &j);
   s=(EdgeNode *)malloc(sizeof(EdgeNode));
                                       // 生成新边表结点s
  s->adjvex=j;                          // 邻接点序号为j
  s->next=G->adjlist[i].firstedge;
 G->adjlist[i].firstedge=s;
                    // 将新边表结点s插入到顶点Vi的边表头部
 }
}//CreateALGraph
```

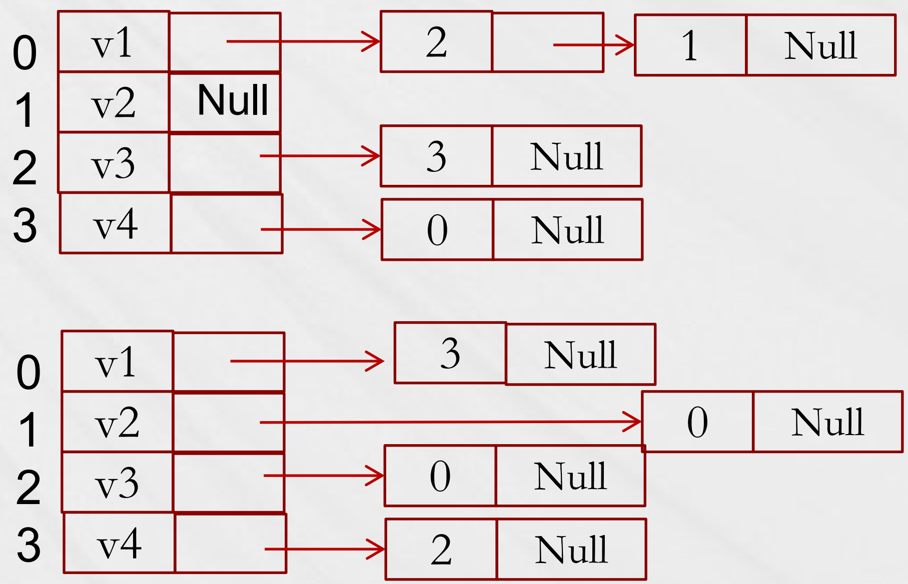


中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

- 若无向图中有n 个顶点、e条边，则它的邻接表需n个顶点表结点和2e个邻接表结点。 在稀疏情况下，比邻接矩阵节省存储。
- 时间复杂度：O(n+e)
- 有向图的逆邻接表：对每个顶点v_i 建立一个链接以v_i为终点的弧的链表。

【例2】有向图的邻接表和逆邻接表。





7.3 图的遍历

图的遍历可分为四类：

遍历完所有的边而不能有重复，即所谓“一笔画问题”或“欧拉路径”；

遍历完所有的顶点而没有重复，即所谓“哈密尔顿问题”。

遍历完所有的边而可以有重复，即所谓“中国邮递员问题”；

遍历完所有的顶点而可以重复，即所谓“旅行推销员问题”。

对于第一和第三类问题已经得到了完满的解决，而第二和第四类问题则只得到了部分解决。

7.3 图的遍历

- 图的遍历是指从图中的任一顶点出发，对图中的所有顶点访问一次且只访问一次。

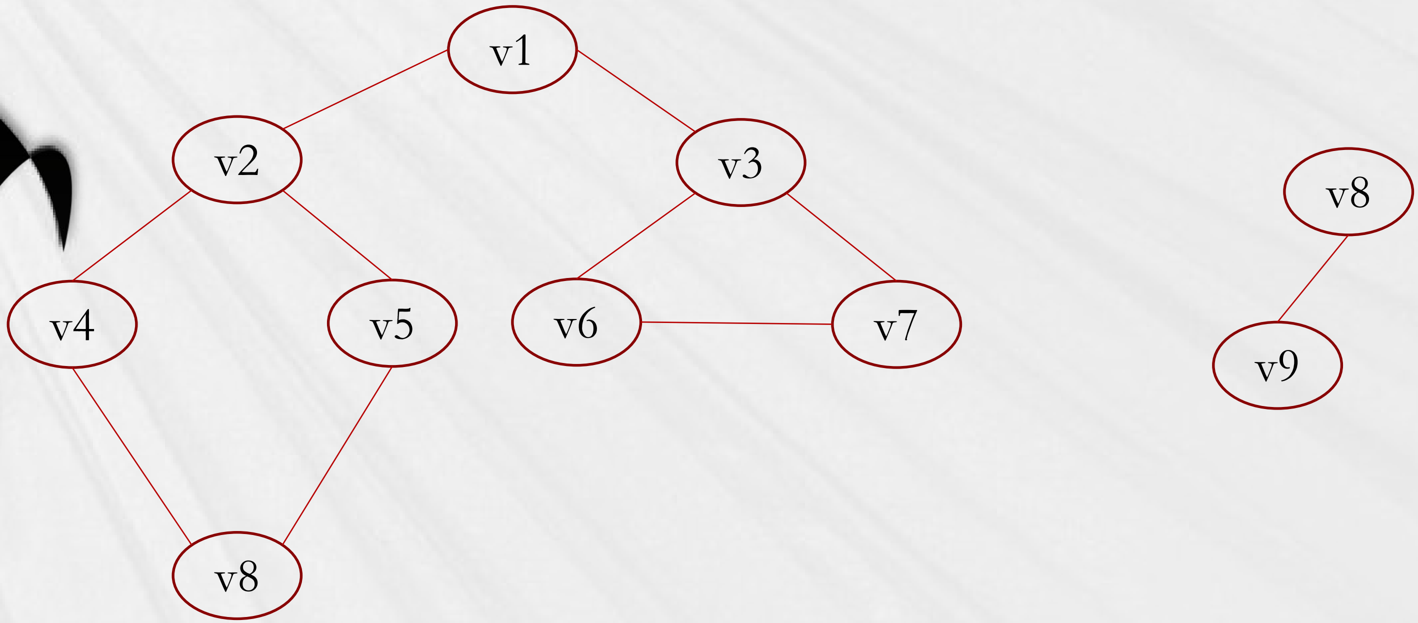
- **两种遍历方法：深度优先，广度优先**

7.3.1深度优先搜索------相当于树的先根遍历

从图中某个顶点v出发，访问此顶点，然后依次从v的未被访问的邻接顶点出发深度优先遍历图，直至图中所有和v有路径相通的顶点都被访问到；若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。

注意： 依次访问邻接顶点的邻接顶点，一直访问到没有邻接顶点，或者邻接顶点已被访问。

深度优先搜索

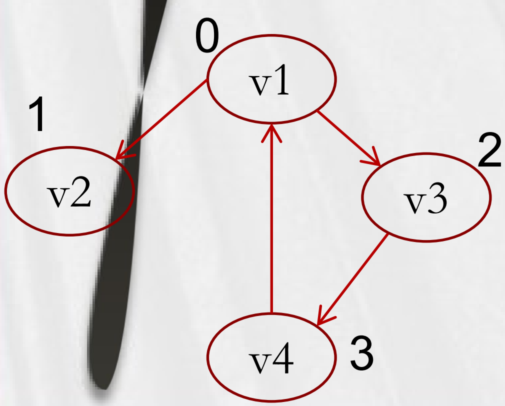


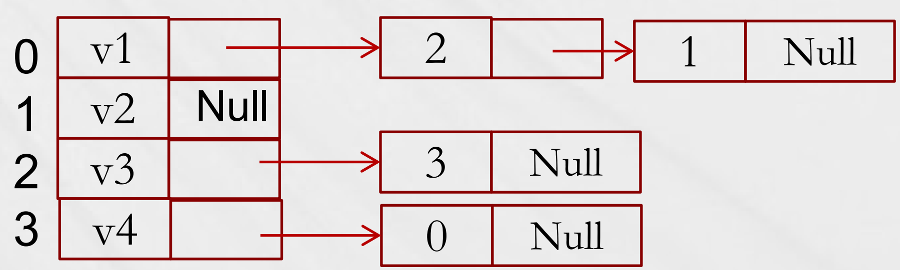
v1 v2 v4 v8 v5 v3 v6 v7   v1 v2 v4 v8 v5 v3 v6 v7v8v9

- 深度优先遍历以邻接表存储的图G：
- 全局数组visit[i]：标识节点i是否访问过

```
void  DFSTraverseAL(ALGraph  *G)
{   int  i;
   for (i=0; i<G->n; i++)
        visited[i]=FALSE;            // 初始化
   for (i=0; i<G->n; i++)
       if  ( ! visited[i])  DFSAL(G, i);   // vi未访问过，从vi开始搜索
 }//DFSTraveseAL
```

```
void DFSAL(ALGraph  *G,  int  i) // 以Vi为出发点对图G搜索
{  EdgeNode  *p;
   printf("visit vertex:V%c\n", G->adjlist[i].vertex); // 访问顶点Vi
   visited[i]=TRUE;              // 标记Vi已访问
   p=G->adjlist[i].firstedge;      // 取Vi边表的头指针
   while(  p  )                  // 依次搜索Vi的邻接点Vj
    {               // 若Vj尚未访问，则以Vj为出发点向纵深搜索
       if (!visited[p->adjvex])   DFSAL(G, p->adjvex);
       p=p->next;                 // 找Vi的下一个邻接点
     }
}//DFSAL
```





节点表

边表：邻接表

- 深度优先遍历的复杂度
- 遍历图的过程：查找邻接节点的过程
- 邻接矩阵存储的图G：O（n^2)
- 邻接表存储的图G： O（n+e)

7.3.2 广度优先搜索

类似于树的**层次遍历**。

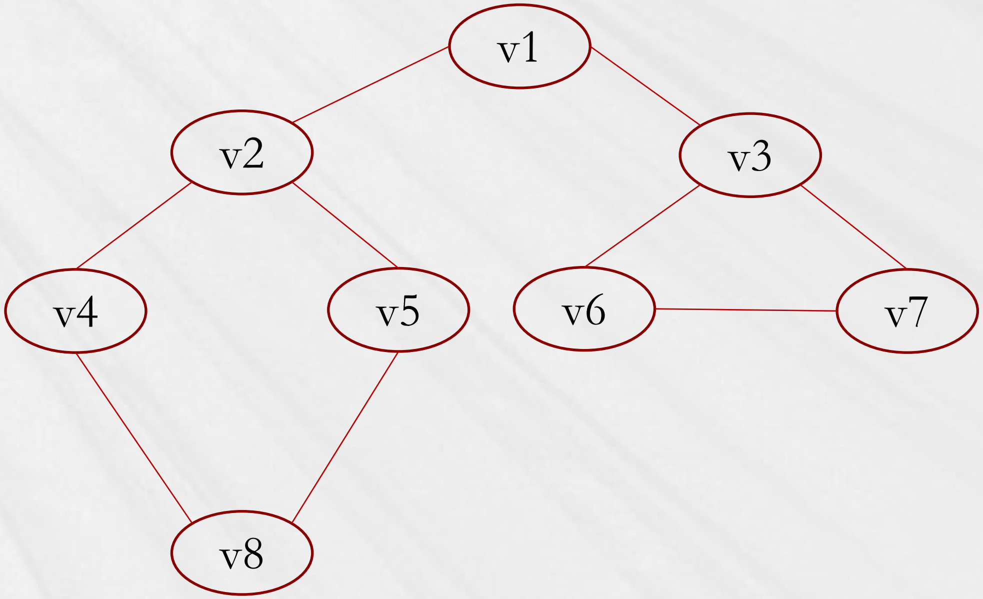
从图中某顶点v出发，在访问了v之后依次访问v的**所有未曾访问过**的**邻接点**，然后分别从这些邻接点出发依次访问它们的所有邻接点，直至图中所有已被访问的顶点的邻接点都被访问到。

若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。

注意：访问v的**所有未曾访问过**的邻接点，然后再分别访问这些邻接顶点的邻接顶点。

数据结构:队列的使用

广度优先搜索

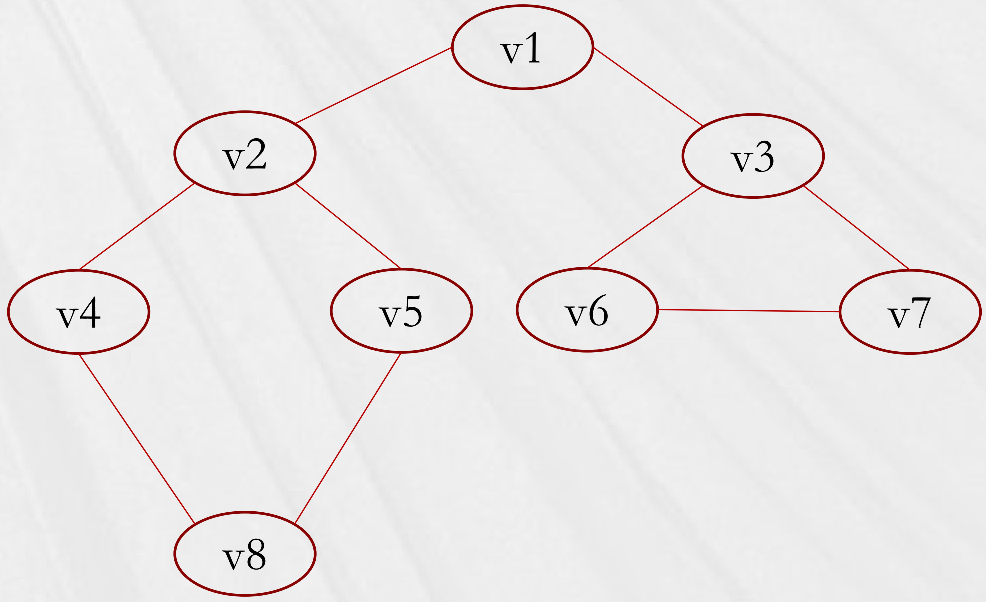


v1 v2 v3 v4 v5 v6 v7 v8

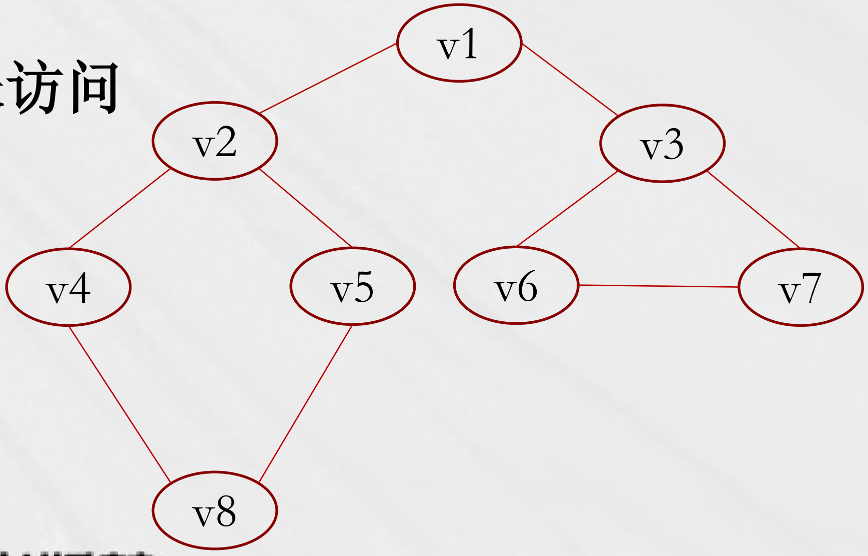
广度优先搜索算法

1）设置队列，存储要遍历的各个节点，出队列时访问

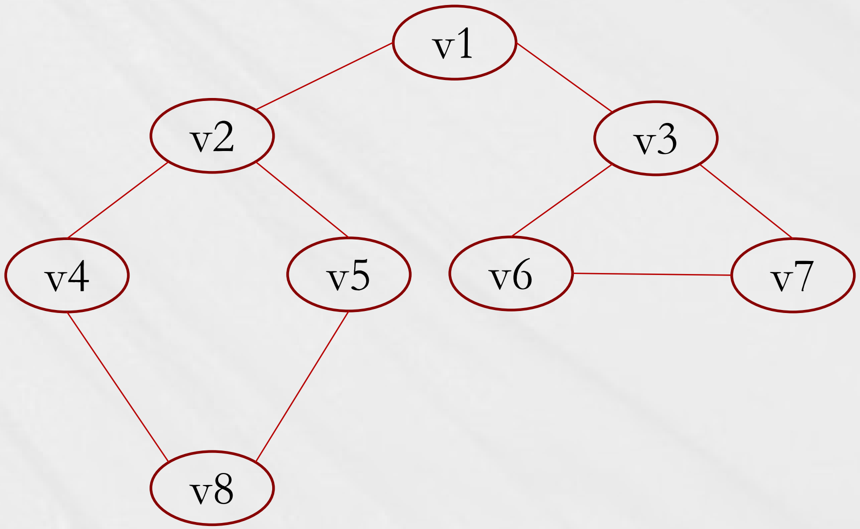
2）设置一个数组visited[i]，标识节点是否被访问



```
void  BFSTraverse(Graph G,  int v)
  {// 按广度优先非递归遍历图G。使用辅助队列Q和访问
  标志数组visited[ ]
 int   i, u, w, visited[G.n];
 for (i=0; i<G.n; ++i)
       visited[i]=FALSE;
  InitQueue(Q);                // 置空队列Q
  if  (!visited[v])   // v尚未访问
   {
     EnQueue(Q, v);
```

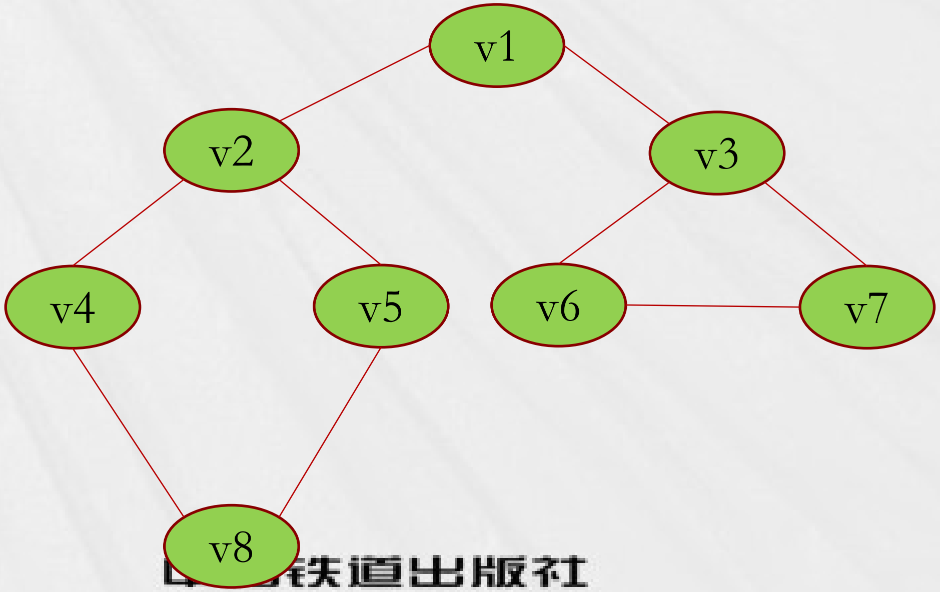


```
while ( !QueueEmpty(Q) )
   { DeQueue(Q, u);       // 队头元素出队并置为u
    visit(u);  // 访问u
    visited[u]=TRUE;
    for(w=FistAdjVex(G,u); w ; w=NextAdjVex(G,u))
          if (!visited[w])  EnQueue(Q, w);
     }//while
  }//if
}//BFSTraverse
```



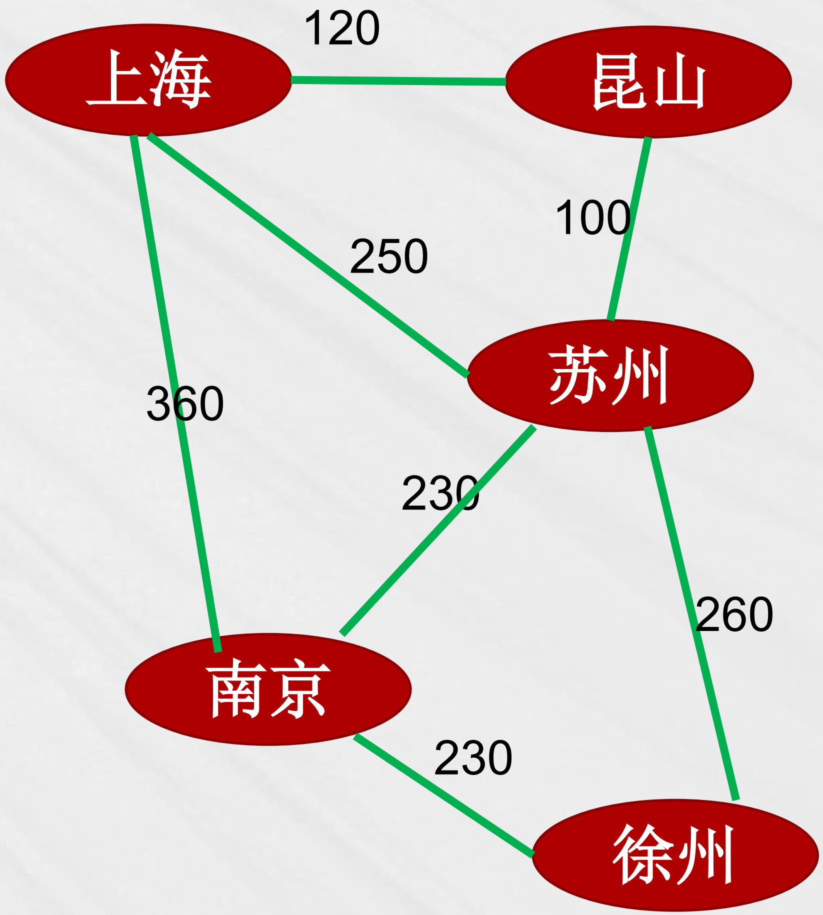
7.4最小生成树

- **生成树**：一个**连通图**G=(V, E)， 树**T=(V, E')**称为G的生成树，若E'⊆E。
- 由连通图可以生成多种不同的树
- 对于有n个顶点的无向连通图，无论其生成树的形态如何，所有**生成树**中都有且仅有**n－1条边**。



中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

- 如果无向连通图是一个网图，那么，它的所有生成树中必有一棵边的权值总和最小的生成树，我们称这棵生成树为最小生成树。
- 城市间铺设光纤
- 如何生成最小生成树？



7.4.1 普里姆算法

假设G＝（V，E）为一网图，其中V为网图中所有顶点的集合，E为网图中所有带权边的集合。

设置两个新的集合U和T

集合U：用于存放G的最小生成树中的顶点

集合T：存放G的最小生成树中的边

顶点集合U的初值为U＝{u_1}（假设构造最小生成树时，从顶点u_1出发，**u_1**为根节点）

边集合T的初值为T＝{ }

U

V/U

Prim算法的思想：

从所有u∈U，v∈V－U的边中，选取具有最小权值的边（u，v），将顶点v加入集合U中，将边（u，v）加入集合T中，如此不断重复，直到U＝V时，最小生成树构造完毕，这时集合T中包含了最小生成树的所有边。

Prim算法的思想：

1.U={u1}， T={ }

2.while(U≠V) do

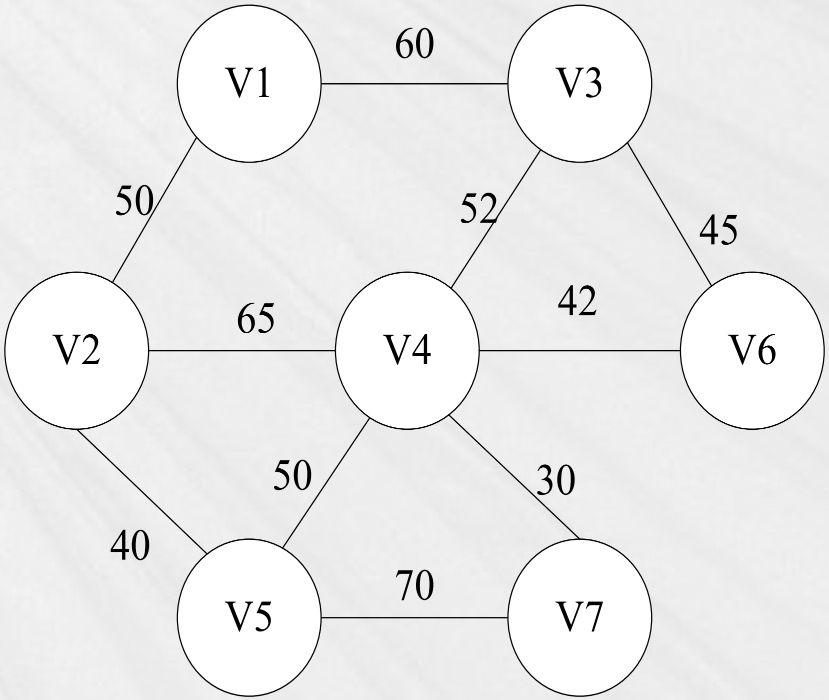
$(u,v)=\min\{w_{uv}, u\in U, v\in V-U\}$

T=T+(u,v)

U=U+{v}

3.end

【例3】 按照Prim方法，从顶点1出发，求最小生成树。



U＝{v1}
T={ }

U＝{v1,v2}
T={<v1,v2>}

U＝{v1,v2,v5}
T={<v1,v2>,<v2,v5>}

U＝{v1,v2,v5,V4}
T＝{<v1,v2>,<v2,v5>,<v5,v4> }

U＝{v1,v2,v5,V4,v7}
T＝{<v1,v2>,<v2,v5>,<v5,v4> ,<v4,v7>}

两个辅助一维数组lowcost[ ]和closevertex[ ]

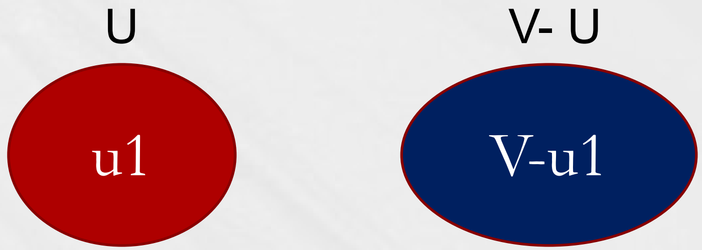
lowcost[ ]用来保存集合V－U中各顶点与集合U中各顶点构成的边中具有最小权值的边的**权值**，lowcost[i]=56，表示V－U中顶点 **i** 到U中某个顶点k的最小边长为56。 closevertex[i]=k。

closevertex[ ]用来保存与集合U-V中的顶点具有最小边、且位于U中的**顶点**，存最小生成树。$uk \in U$， $ui \in V-U$

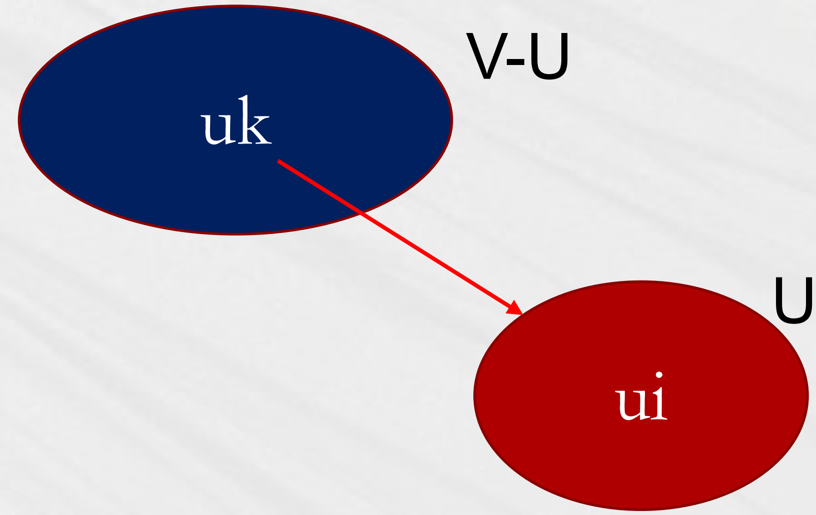
lowcost[i]=0，表示**i**顶点在集合U中。

假设初始状态时：

U＝{u1}(u1为出发的顶点)，这时有lowcost[0]=0，它表示顶点u1已加入集合U中，数组lowcost[ ]的其它各分量的值是{V-u1}到 顶点u1所构成的直接边的权值。closevertex[i]=0;

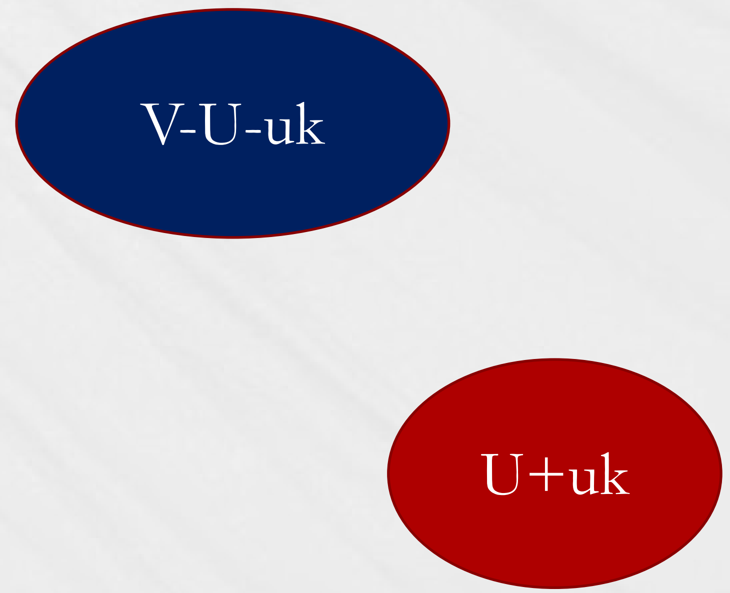


然后不断选取权值最小的边（uk，ui）（ui∈U，uk∈V－U），每选取一条边，就将lowcost（k）置为0，表示顶点uk已加入集合U中。
closevertex[k]=i，记录边。



由于顶点uk从集合V－U进入集合U后，这两个集合的内容发生了变化， V－U-uk中的顶点ui到达 uk的边长也许比原先还要短，就需依据具体情况更新数组lowcost[i]和closevertex[i]=k。

最后closevertex[ ]中即为所建立的最小生成树。



```
void Prim（int gm[ ][MAXNODE]，int n，int closevertex[ ]）
{    int lowcost[100],mincost;
      int i,j,k;
for (i=1;i<n;i++)
  {    lowcost[i]=gm[0][i];      closevertex[i]=0;          }
lowcost[0]=0;              //从序号为0的顶点出发生成最小生成树
closevertex[0]=0;
for (i=1;i<n;i++)            //寻找当前最小权值的边的顶点
 {    mincost=MAXCOST;     //MAXCOST为一个极大的常量值
       j=1;   k=1;
      while (j<n)
        { if (lowcost[ j ]<mincost && lowcost[ j ]!=0)
                       {  mincost=lowcost[ j ];        k=j;       }
             j++;     }
   printf("顶点:%d,%d,边的权值＝%d\n",k, closevertex[k],mincost);
   lowcost[k]=0;
   for (j=1; j<n; j++)   //修改其它顶点的边的权值和最小生成树顶点序号
     if (gm[k][j] < lowcost[ j ] && lowcost[j] != 0)
          {   lowcost[ j ]=gm[ k ][ j ];     closevertex[ j ]=k;        }
 }
}
```

V-U

U={u0}

U+uk

V-U-uk

7.4.2 克鲁斯卡尔算法

Kruskal算法是一种按照网中**边的权值递增的顺序**构造最小生成树的方法。设无向连通网为G＝（V，E），令G的最小生成树为T，其初态为T＝（V，{ }）。

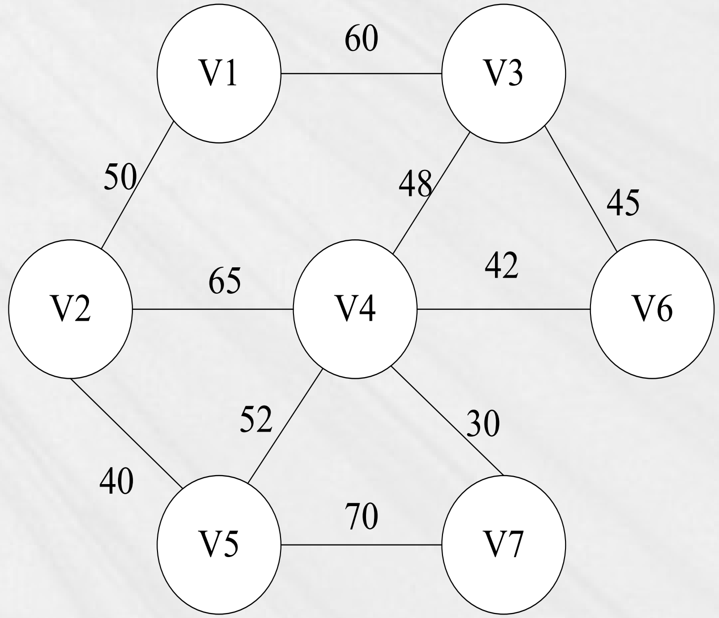
基本思想：按照边的权值由小到大的顺序排序，考察G的边集E中的各条边。

若被考察的边的两个顶点**属于T的两个不同的连通分量**，则将此边作为最小生成树的边加入到T中，同时把两个连通分量连接为一个连通分量;

若被考察边的两个顶点**属于同一个连通分量**，则舍去此边，以免造成回路;

如此重复，当T中的连通分量个数为1时，此连通分量便为G的一棵最小生成树。

【例4】 按照Kruskal方法构造最小生成树 。



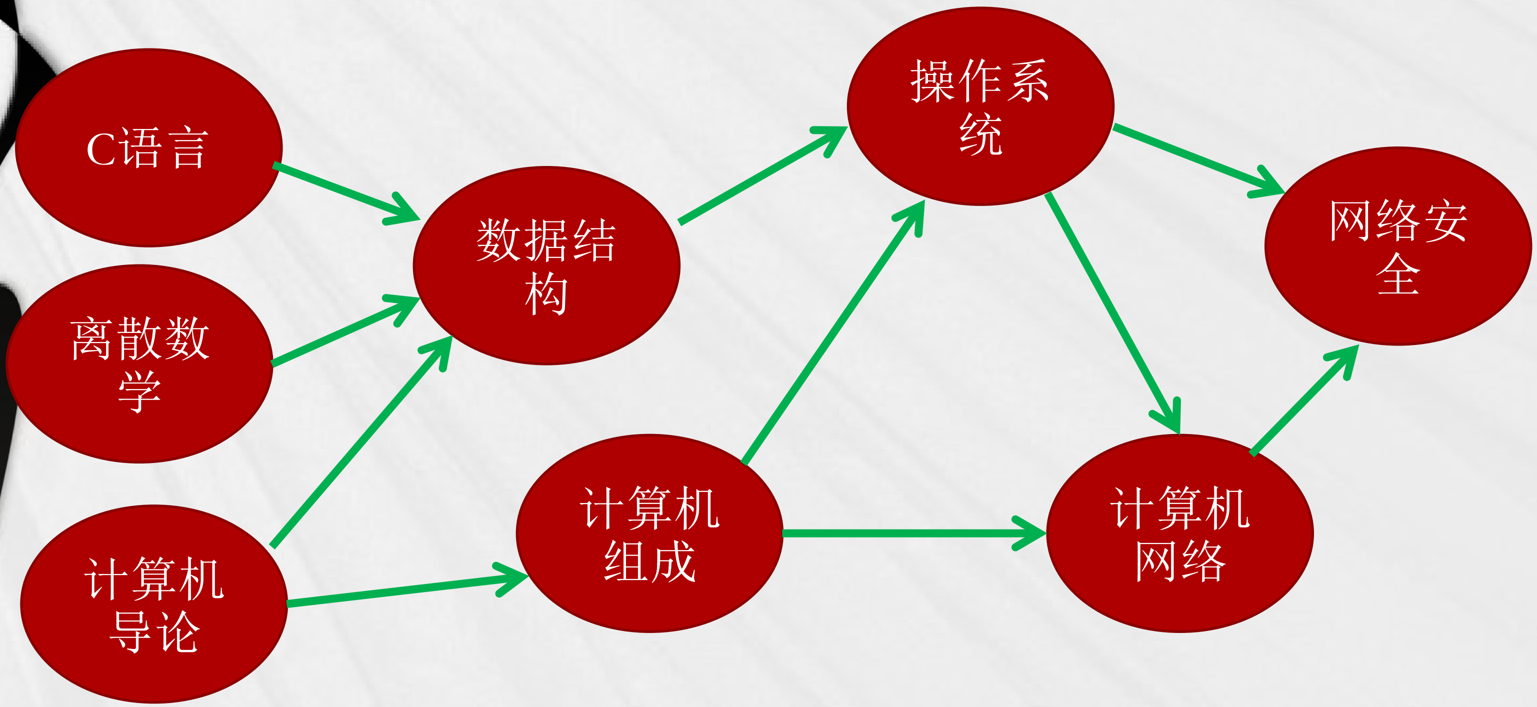
<v4,v7> <v2,v5> <v4,v6> <v3,v6> <v3,v4><v1,v2> <v4,v5> <v1,v3> <v2,v4> <v5,v7>

T＝{<v4,v7> } T＝{<v4,v7> ,<v2,v5>} T＝{<v4,v7> ,<v2,v5>,<v4,v6> }

T＝{<v4,v7> ,<v2,v5>,<v4,v6>,<v3,v6> }

T＝{<v4,v7> ,<v2,v5>,<v4,v6>,<v3,v6>,<v1,v2> }

7.5 拓扑排序



偏序关系：自反的、非对称的、传递的（大于等于）

中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

7.5 拓扑排序

1. AOV网 （Activity on Vertex network）

- 工程或者流程的某个阶段：**活动**
- 以有向图中的**顶点**来表示**活动**，**有向边**表示活动之间的**优先关系，这样的图称为AOV网**。
- **前驱、后继**：顶点Vi到顶点Vj有一条有向路径
- **直接前驱、直接后继：有向边<Vi,Vj>**
- AOV网中的弧表示了活动之间存在的制约关系。如课程的先后修读关系。

2. 拓扑排序

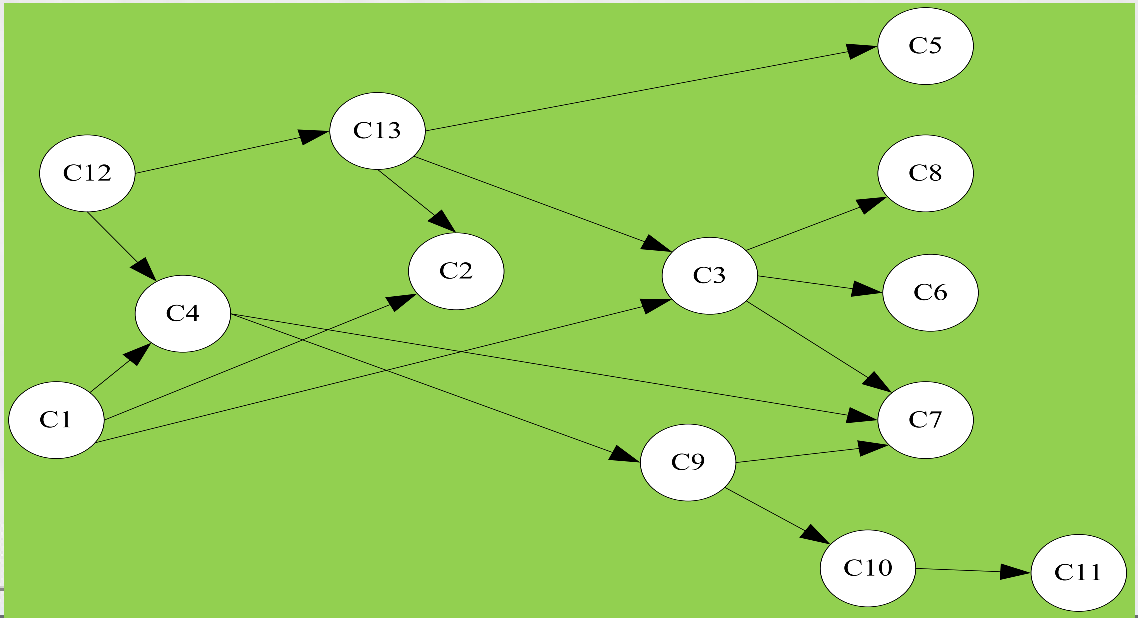
- AOV网所代表的一项工程中活动的集合是一个**偏序集合**。为了保证该项工程得以顺利完成，必须保证AOV网中不出现回路；否则，意味着某项活动应以自身作为能否开展的先决条件，这是荒谬的。如何判断AOV网是否合理呢(判断是否存在回路）？

❖ **测试AOV网是否具有回路**的方法，就是在AOV网的偏序集合下**构造一个线性序列**，该线性序列具有以下性质：

①在AOV网中，若顶点i 优先于顶点j ，则在线性序列中顶点i仍然优先于顶点j;

②对于网中原来没有优先关系的顶点 ，如图7-17中的C_1与C_{13} ，在线性序列中也建立一个先后关系，或者顶点i优先于顶点j ，或者顶点j 优先于i。

满足上述性质的线性序列称为**拓扑有序序列**，构造拓扑有序序列的过程称为**拓扑排序**。



3. 拓扑排序算法

对AOV网进行拓扑排序的方法和步骤是：

①从AOV网中选择一个没有前驱的顶点（该顶点的入度为0）并且输出它；

②从网中删去该顶点，并且删去从该顶点发出的全部有向边；

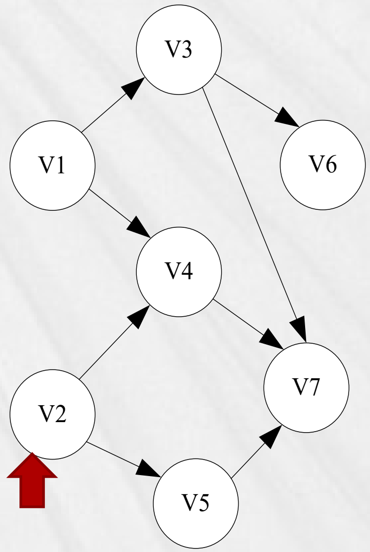
③重复上述两步，直到剩余的网中**不再存在没有前驱的顶点**为止。

❖ **操作的结果有两种：**

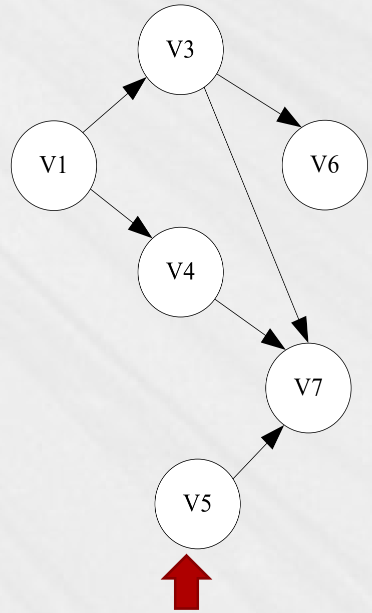
①网中全部顶点都被输出，这说明网中不存在有向回路；

②网中顶点未被全部输出，剩余的顶点均不是无前驱的顶点，这说明网中存在有向回路。

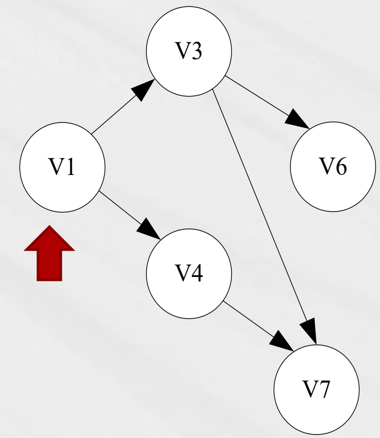
【例5】 在一个AOV网上进行拓扑排序 。



V2



V2，V5



V2，V5，V1

V2，V5，V1，V4

V2，V5，V1，V4，V3

7.6 关键路径

1. AOE网（Activity on edge network）

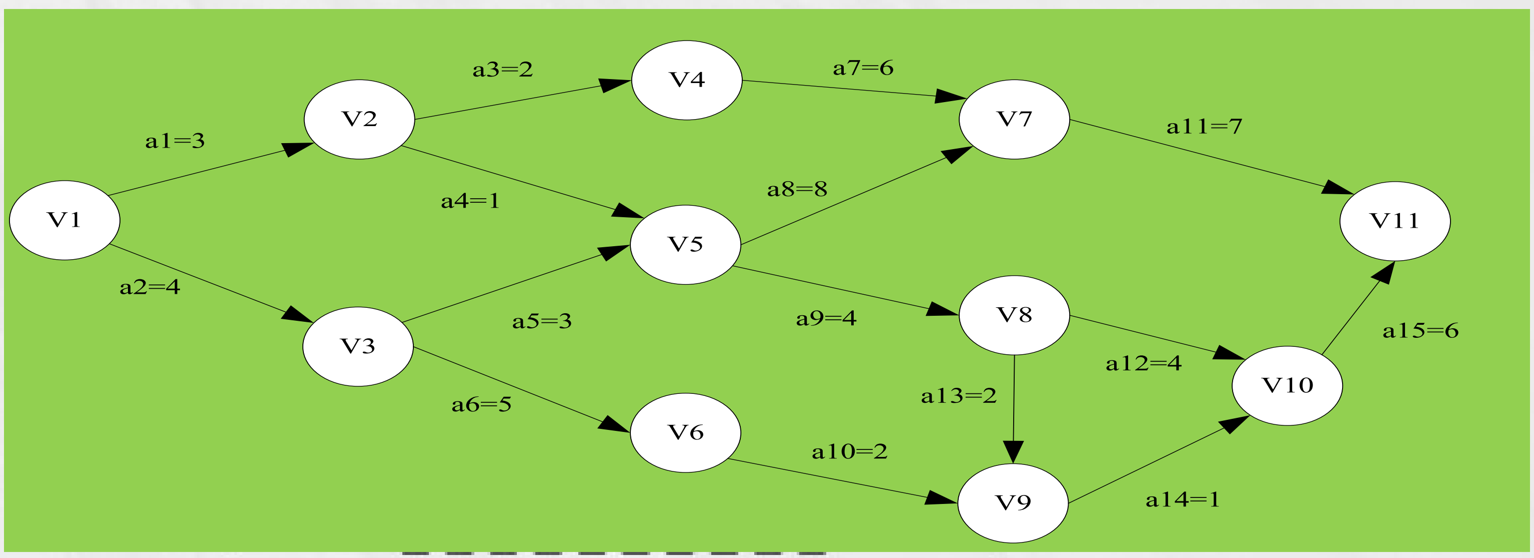
若**在带权的有向图**中，以**顶点表示事件**，以**有向边表示活动**，边上的权值表示活动的开销（如该活动持续的时间），则此带权的有向图称为**AOE网**。

AOE网的性质：

（1）只有某个顶点所代表的事件发生后，从该顶点出发的有向边所代表的活动才开始。

（2）只有进入某顶点的各条有向边所代表的活动都已结束，该顶点所代表的事件才能发生。

工程



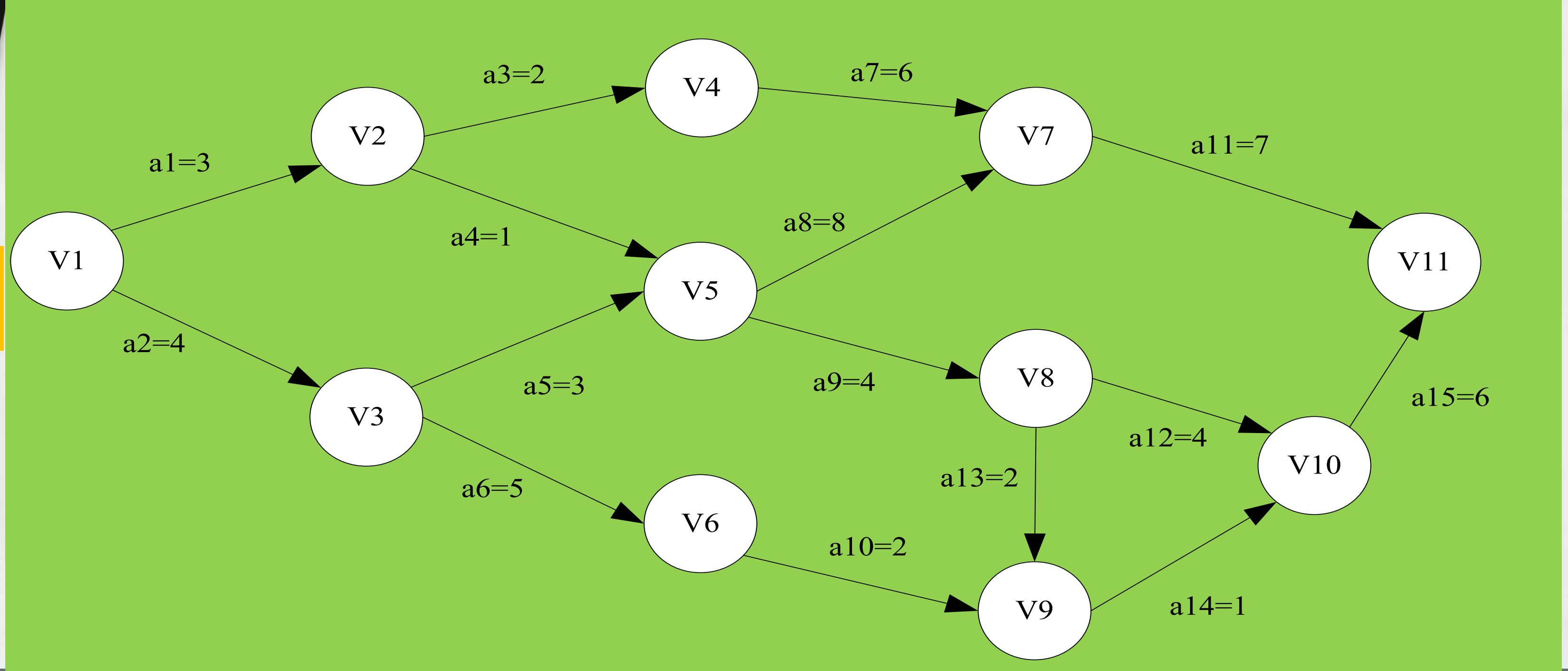
7.6 关 键 路 径

2. 关键路径

由于AOE网中的某些活动能够同时进行，所以完成整个工程所必须花费的时间应该为源点到终点的最大路径长度。

具有最大路径长度的路径称为**关键路径**。关键路径上的活动称为**关键活动**。关键路径长度是整个工程所需的最短工期。

工程



3. 关键路径的确定

（1）事件的最早发生时间ve[k]

源点V1到Vi的最长路径的长度叫做Vi的最早发生时间。

ve[k]是指从源点到顶点Vk的最大路径长度代表的时间

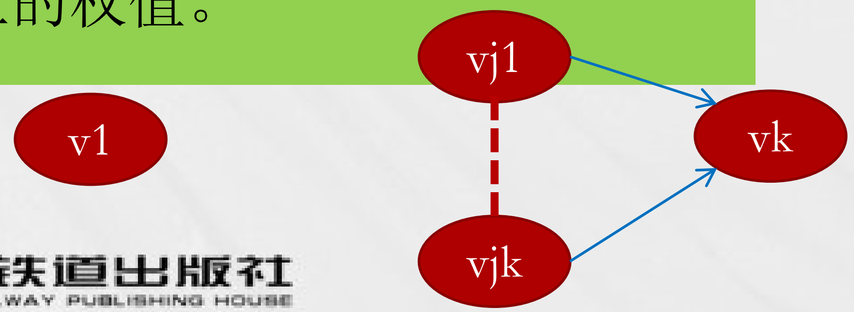
只有进入v_k的所有活动<Vj,Vk>发生后Vk才能开始

计算Vk发生的最早时间的方法如下：

$ve[l]=0$

$ve[k]=Max\{ve[j]+dut(<v_j,v_k>)\} \quad <v_j,v_k>\in p[k]$

- 其中，p[k]表示所有到达vk的有向边的集合；
- $dut(<v_j, v_k>)$为有向边$< v_j, v_k >$上的权值。



（2）事件的最迟发生时间vl[k]

vl[k]是指在不推迟整个工期的前提下，事件v_k允许的最晚发生时间。

vl[k] 的计算方法如下：

vl[n]=ve[n]

$vl[k]=Min\{vl[j]-dut(<v_k-v_j>)\} \quad <v_k-v_j> \in s[k]$

❖ 其中，s[k]为所有从v_k发出的有向边的集合。

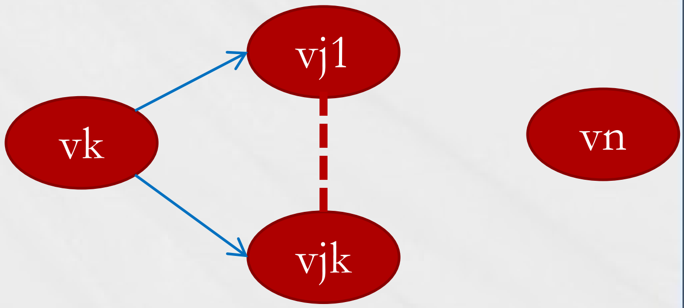
（3）活动a_i的最早开始时间e[i]

e[i]=ve[k]

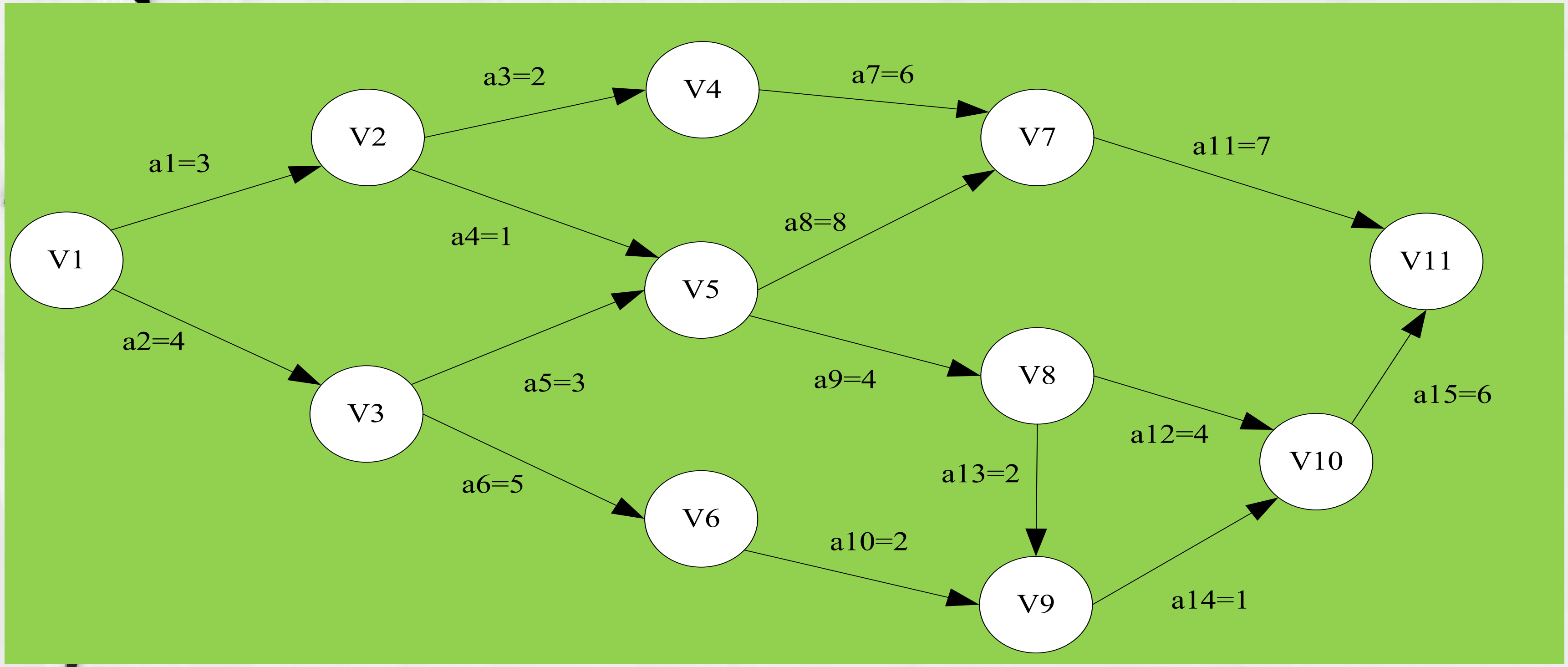
（4）<vk,vj>弧表示活动a_i， a_i最晚开始时间l[i]要保证vj不被拖后

$l[i]=vl[j]-dut(<v_k,v_j>)$

◆l[i]=e[i]的活动就是关键活动。



【例6】确定AOE网的关键活动和关键路径。



中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

7.7 最短路径

问题：

（1）城市为节点，公路为边，长度为边的权值，城市间找最短路径

（2）对于非网图，最短路径指两个节点间边最少的路径

两种最短路径问题：

单源顶点到其他顶点的最短路径

每对顶点之间的最短路径

7.7 最短路径

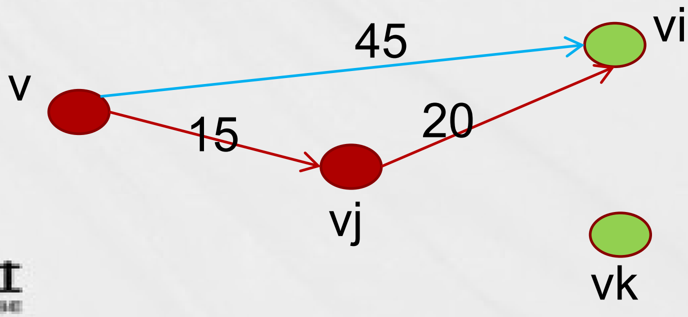
7.7.1 单源点最短路径

给定带权有向图G＝（V，E）和**源点v**∈V，求从v 到G中其余各顶点的最短路径。

◆**Dijkstra算法**的基本思想：

1.首先，引入一个数组D，它的每分量 D[i]表示当前找到的从**起始点（即源点v）**到**顶点vi**的最短路径的长度。

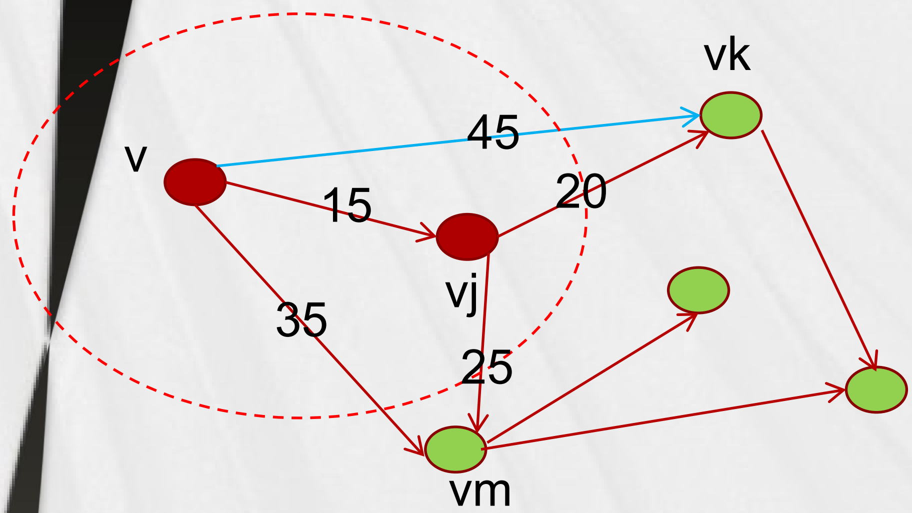
2.D的初始状态为：若从v到vi有弧（即从v到vi存在连接边），则D[i]为弧上的权值（即为从v到vi的边的权值）；否则置D[i]为∞。



◆Dijkstra算法的基本思想：

3.找最短的D[j]，即为源点v到vj的最短路径；调整与vj有弧的顶点的D[k]值；

假设该次最短路径的终点是vk，则可想而知，这条路径要么是(v,vk)，或者是(v,vj,vk)。它的长度或者是从v到vk的弧上的权值，或者是D[j]加上从vj到vk的弧上的权值。



D[k]=min{D[k], D[j] + (vj,vk)}

调整Vj的所有邻接顶点的D[ ]

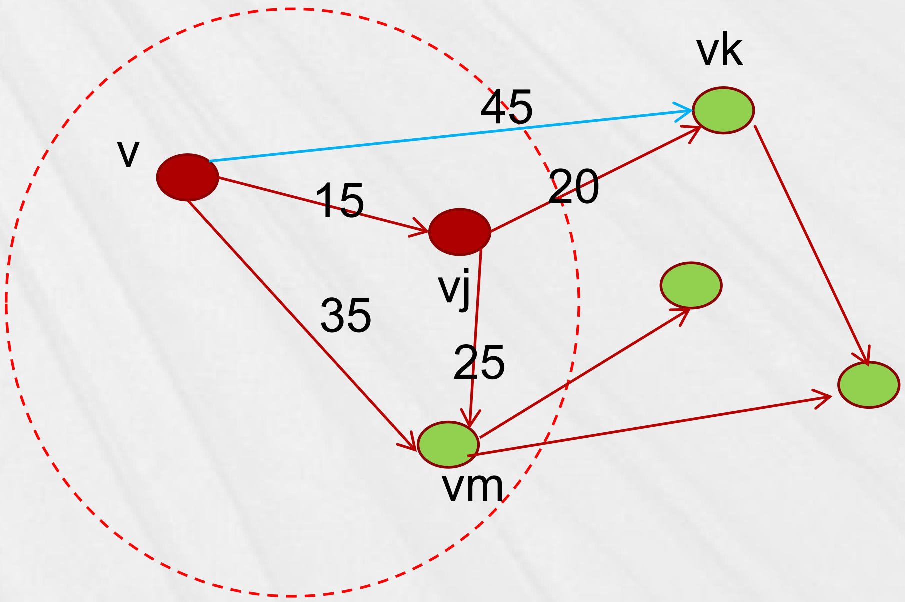
找到最小的D[x]

假设是D[m]，即Vm

Vm加入红圈----》next page

7.7 最短路径

◆Dijkstra算法的基本思想：



调整与vm有弧的顶点的D[k]值；

①用带权的**邻接矩阵edges** 来表示带权有向图，edges[i][j]表示弧 $\langle v_i, v_j \rangle$ 上的权值 。若 $\langle v_i, v_j \rangle$ 不存在，则置edges[i][j]为∞。

S为已找到从v出发的最短路径的终点的集合，初始状态为空集。

从v出发到图上其余各顶点（终点）v_i可能达到最短路径长度的**初值**为：

D[i]= edges[Locate_Vex(G,v)][i]　　$v_i \in V$

②选择v_j，使得　$D[j]=Min\{ D[i] \mid v_i \in V-S \}$

v_j就是当前求得的一条从v出发的最短路径的终点。令

$S=S \cup \{j\}$

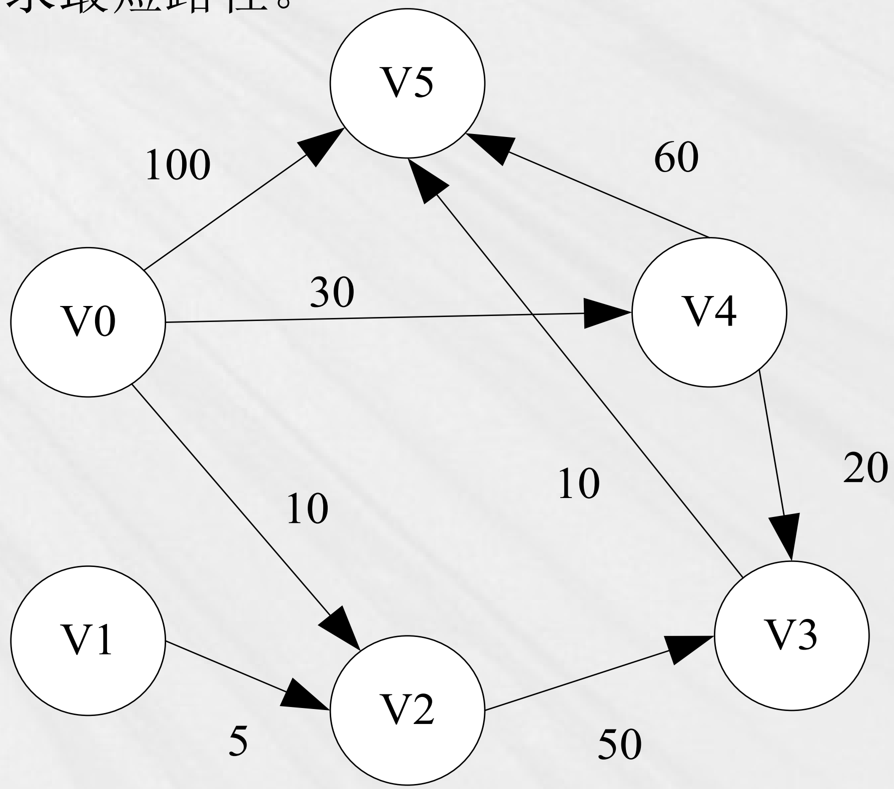
集合V-S：剩下的顶点

③修改vj到**集合V-S**上有弧的任一**顶点v_k**最短路径长度。如果 D[j]+edges[j][k]<D[k],则修改D[k]为

D[k]=D[j]+edges[j][k].

④重复操作②、③共n-1次。由此求得从v 到图上其余各顶点的最短路径，是依**路径长度递增**的序列。

【例7】 求最短路径。



从v0出发 S={ } V-S={1，2，3，4，5}

初始化

顶点	1	2	3	4	5
D[i]初值	65535	10	65535	30	100

找最小

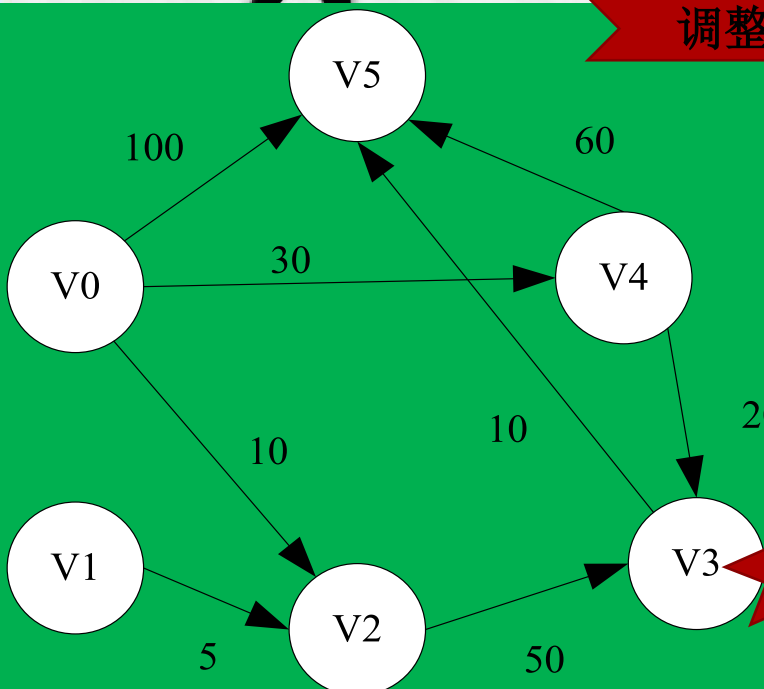
从v0出发　S={ }　V-S={1，2，3，4，5}

顶点	1	2	3	4	5
D[i]值	65535	10	65535	30	100

调整

从v0出发　S={2}　V-S={1，3，4，5}

顶点	1	2	3	4	5
D[i]值	65535	10	60	30	100



找最小

从v0出发　S={2}　V-S={1，3，4，5}

顶点	1	2	3	4	5
D[i]值	65535	10	60	30	100

调整

从v0出发　S={2,4}　V-S={1，3，5}

顶点	1	2	3	4	5
D[i]值	65535	10	50	30	90

找最小

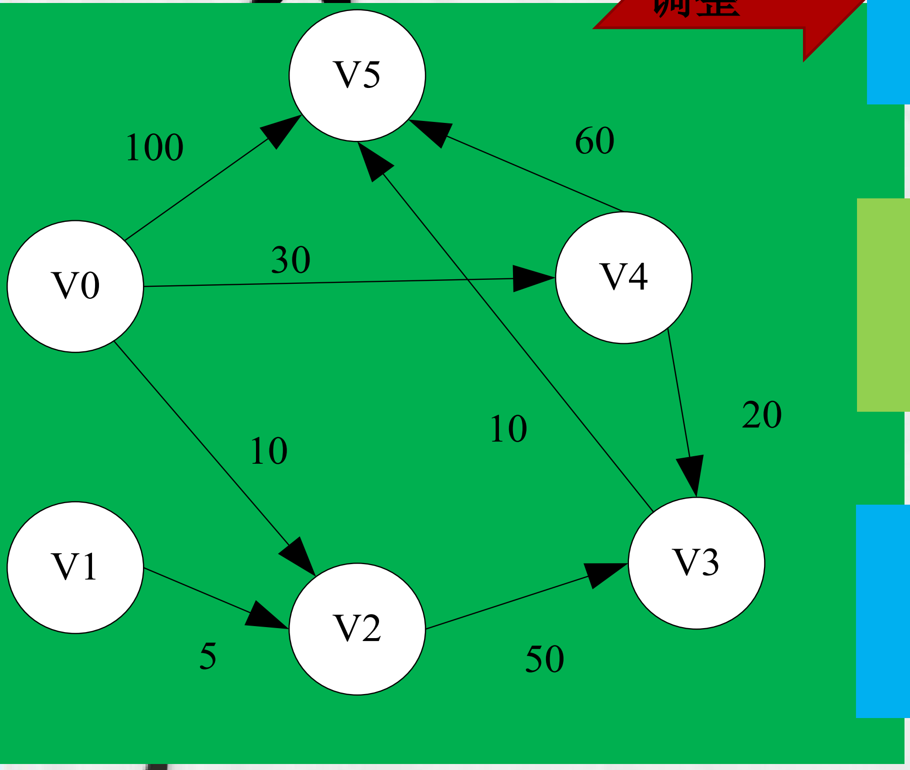
从v0出发 S={2, 4} V-S={1， 3， 5}

顶点	1	2	3	4	5
	65535	10	50	30	90

调整

从v0出发 S={2, 3， 4} V-S={1， 5}

顶点	1	2	3	4	5
	65535	10	50	30	60



从v0出发 S={2, 3， 4} V-S={1， 5}

顶点	1	2	3	4	5
	65535	10	50	30	60

从v0出发 S={2, 3， 4， 5} V-S={1}

顶点	1	2	3	4	5
	65535	10	50	30	60

从v0出发 S={2, 3， 4， 5， 1} V-S={ }

顶点	1	2	3	4	5
	65535	10	50	30	60

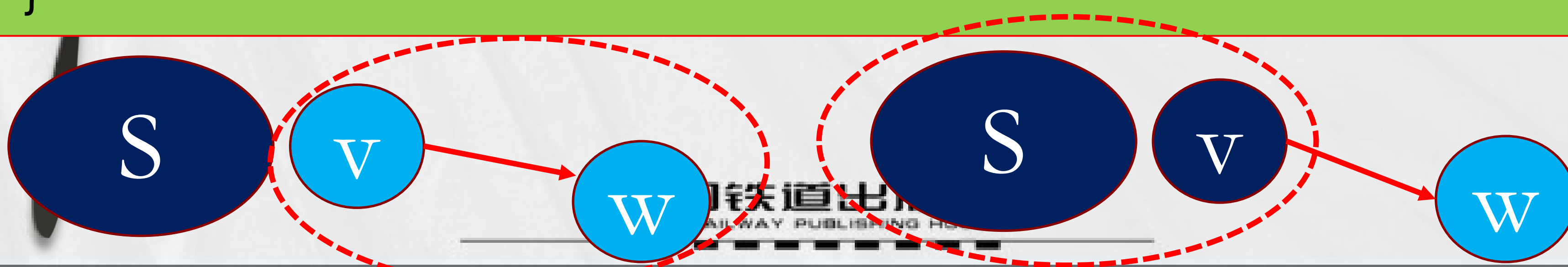
// P[v][ ]：存储从顶点v0到顶点v的最短路径上的顶点，若P[v][w]为TRUE，则w是从v0到 v当前求得最短路径上的顶点。

//路径长度 D[v]：存储从顶点v0到顶点v的最短路径长度

//final[v] 为TRUE当且仅当v∈S，即已经求得从v0到v的最短路径

//常量INFINITY为边上权值可能的最大值

```
void ShortestPath_1(Mgraph G, int v0, PathMatrix *p,  ShortPathTable *D)
{
      for (v=0;v<G.vexnum;++v)
          {   fianl[v]=FALSE;   D[v]=G.edges[v0][v];
              for (w=0; w<G.vexnum; ++w)    P[v][w]=FALSE;  //设空路径
             if (D[v]<INFINITY)     P[v][v0]=TRUE;       }

       D[v0]=0; final[v0]=TRUE;          //初始化，v0顶点属于S集
       for(i=1; i<G.vexnum; ++i)          //其余G.vexnum-1个顶点
        {  min=INFINITY;                  //min为当前所知离v0顶点的最近距离
           for (w=0;w<G.vexnum;++w)
              if ( ! final[w]&& (D[w]<min)) {  v=w;   min=D[w];  } //w顶点在V－S中
           final[v]=TRUE                       //离v0顶点最近的v加入S集合
           for(w=0;w>G.vexnum;++w)                       //更新当前最短路径
              if ( ! final[w] && (min+G.edges[v][w]<D[w]))   //修改D[w]和P[w], w∈V-S
                {  D[w]=min+G.edges[v][w];
                   P[w]=P[v];   P[w][v]=TRUE;      }          //P[w]=P[v]+P[
     }
```



7.7.2 每对顶点之间的最短路径

❖ Floyd算法基本思想：

假设求从顶点v_i到v_j的最短路径。如果从v_i到v_j有弧，则从v_i到vj存在一条长度为edges[i][j]的路径，该路径不一定是最短路径，尚需进行n次试探。

首先考虑路径（v_i, v_0, v_j）是否存在，即判别弧（v_i, v_0）和(v_0, v_j)是否存在。如果存在，则比较（v_i, v_j）和（v_i, v_0, v_j）的路径长度取长度较短者为从v_i到v_j 的中间顶点的序号不大于0的最短路径。

假如在路径上再增加一个顶点v1，也就是说，如果（ v_i, …, v_1）和（ v_1, …, v_j）分别是当前找到的中间顶点的序号不大于0的最短路径，那么（ v_i, …, v_1, … , v_j） 就有可能是从v_i到v_j的中间顶点的序号不大于1的最短路径。

将它和已经得到的从v_i到v_j中间顶点序号不大于0的最短路径相比较，从中选出中间顶点的序号不大于1的最短路径之后，再增加一个顶点v_2，继续进行试探。依次类推。

- 定义一个n阶方阵序列。

$D^{(-1)}$，$D^{(0)}$，$D^{(1)}$，…，$D^{(k)}$，$D^{(n-1)}$

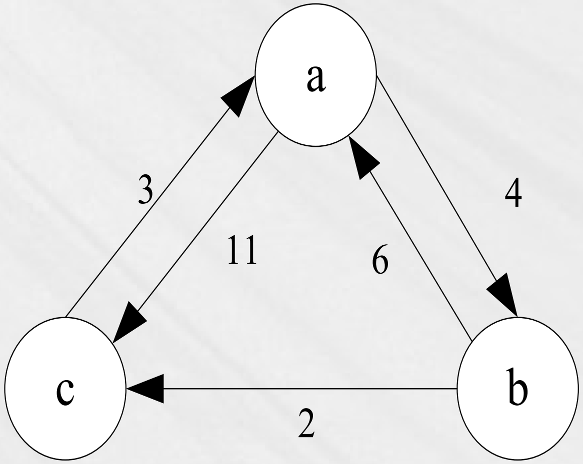
其中，$D^{(-1)}[i][j]=edges[i][j]$

$D^{(k)}[i][j]=Min\{D^{(k-1)}[i][j], D^{(k-1)}[i][k]+D^{(k-1)}[k][j]\}$，

$0 \leqq k \leqq n-1$

- $D^{(1)}[i][j]$是从v_i到v_j的中间顶点的序号不大于1的最短路径的长度；
- $D^{(k)}[i][j]$ 是从v_i到v_j的中间顶点的个数不大于k的最短路径的长度；$D^{(n-1)}[i][j]$ 就是从v_i到v_j的最短路径的长度。
- 求任意两顶点间的最短路径的算法（参见教材）。

【例8】用Floyd算法求有向网中每对顶点之间的最短路径。



本章小结

❖图是一种比树形结构更复杂的非线性数据结构。在图状结构中，任意两个结点之间都可能相关，即结点之间的邻接关系可以是任意的。

❖图通常分为无向图和有向图，图在计算机中存储时一般采用邻接矩阵和邻接表。

❖对于图的遍历有深度优先搜索和广度优先搜索，在求最小生成树时，一般采用普里姆算法和克鲁斯卡尔算法。在图的实际应用中，有求拓扑排序序列、计算关键路径和最短路径。

本章习题

1. 已知一个无向图的顶点集为：{a，b，c，d，e}，其邻接矩阵如下图所示，试：

（1）画出该图。

（2）画出它的邻接表。

（3）写出从顶点a出发按深度优先搜索进行遍历的结点序列。

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

本章习题

2. 已知网的邻接矩阵如下图所示，试：

- 画出该图。
- 计算出该图每个顶点的入度和出度。
- 画出它的一棵最小生成树。

$$\begin{pmatrix} 0 & 8 & 10 & 11 & 0 \\ 8 & 0 & 3 & 0 & 13 \\ 10 & 3 & 0 & 4 & 0 \\ 11 & 0 & 4 & 0 & 7 \\ 0 & 13 & 0 & 7 & 0 \end{pmatrix}$$

本章习题

3. 已知一个图的顶点集V和边集E分别为：

V={0，1，2，3，4，5，6，7}

E={<0,2>,<1,3>,<1,4>,<2,4>,<2,5>,<3,6>,<3,7>,<4,7>,<4,8>,<5,7>,<6,7>,<7,8>}

若采用邻接表存储它，并且每个顶点邻接表中的边结点都是按照顶点序号从小到大的次序链接的，试按照拓扑排序算法，写出得到的拓扑序列。

本章习题

4. 求出如图7.26中顶点1到其余各顶点的最短路径。