第7章 图

引言

- 图（Graph）是一种比线性表和树更为复杂的数据结构
  - 结点之间的关系可以是任意的，不受限制，图中任意两个元素之间都可能相关。
  - 图有着更为广泛的应用，已经渗透到计算机、逻辑学、物理、化学、通信、甚至日常生活中，图论是计算机的重要理论分支。

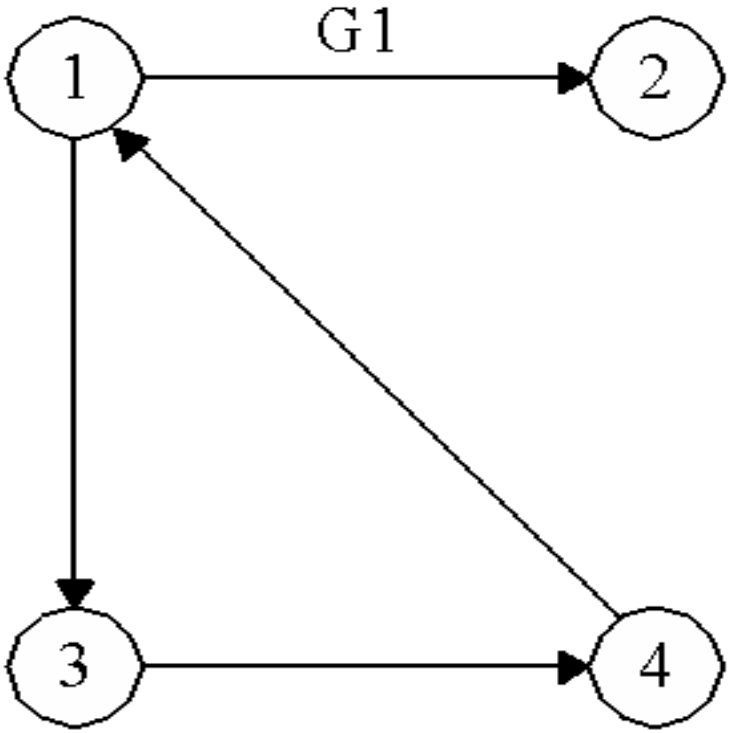
7.1 图的定义和术语

1. 图的定义

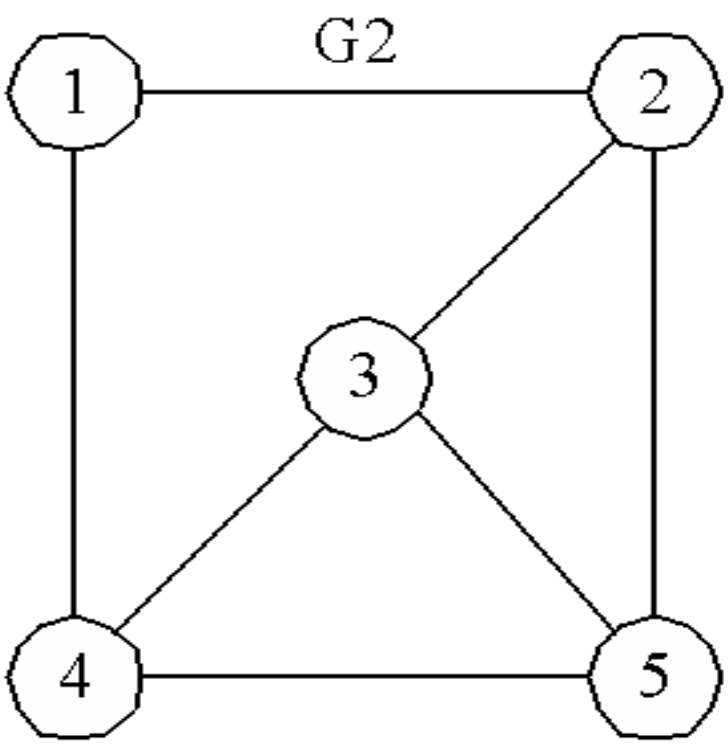
图(Graph)G由两个集合V(Vertex)和E(Edge)组成,记为G=(V,E),其中V是顶点的有限集合,记为V(G),E是连接V中两个不同顶点(顶点对)的边的有限集合,记为E(G)。

在图G中,如果代表边的顶点对是无序的,则称G为无向图,无向图中代表边的无序顶点对通常用圆括号括起来,用以表示一条无向边。

如果表示边的顶点对是有序的,则称G为有向图,在有向图中代表边的顶点对通常用尖括号括起来 。（弧）



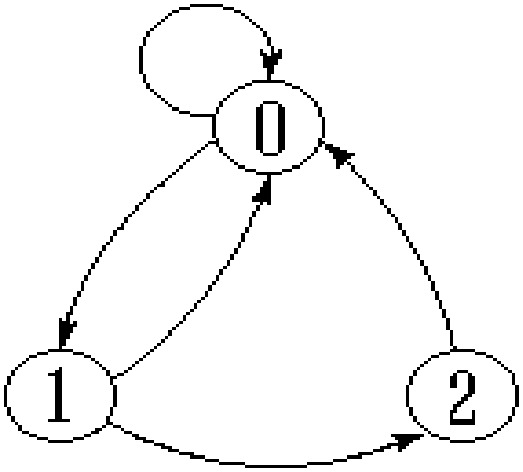
(a) G1是有向图



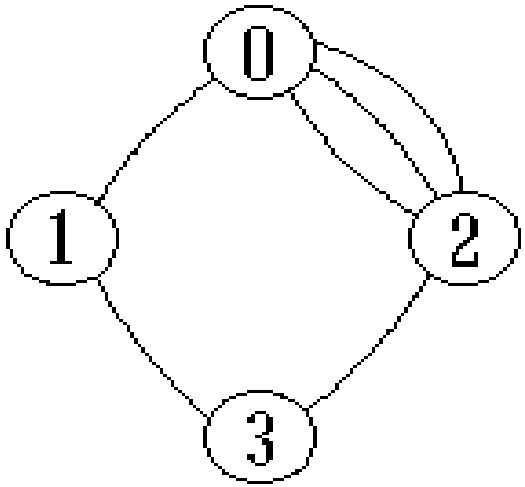
(b) G2是无向图

有向图、无向图示例

本章不予讨论的图



（a）带自身环的图



（b）多重图

2. 基本术语

●完全图、稀疏图与稠密图

n:图中顶点的个数；e:图中边或弧的数目。

无向图其边数e的取值范围是0~n(n-1)/2。

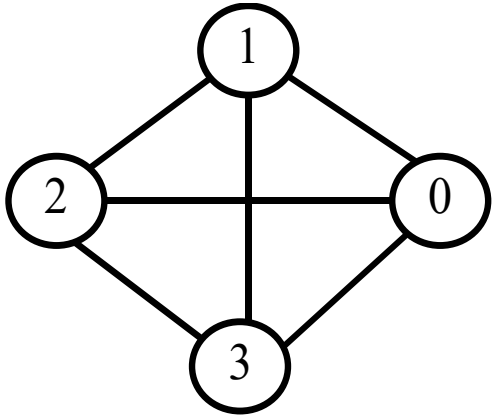
无向完全图：**有n(n-1)/2条边的无向图。**

有向图其边数e的取值范围是0~n(n-1)。

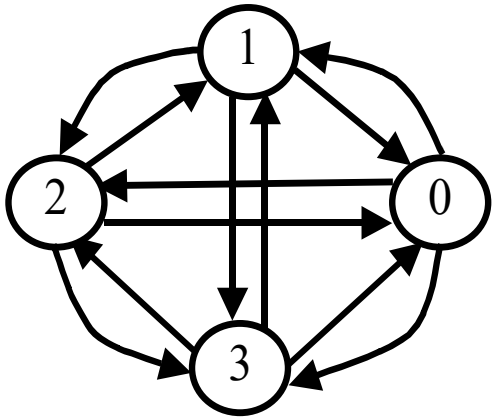
有向完全图：**有n(n-1)条边的有向图。**

稀疏图：**对于有很少条边的图(*e<nlogn*)，**

反之称为稠密图**。**



(a)



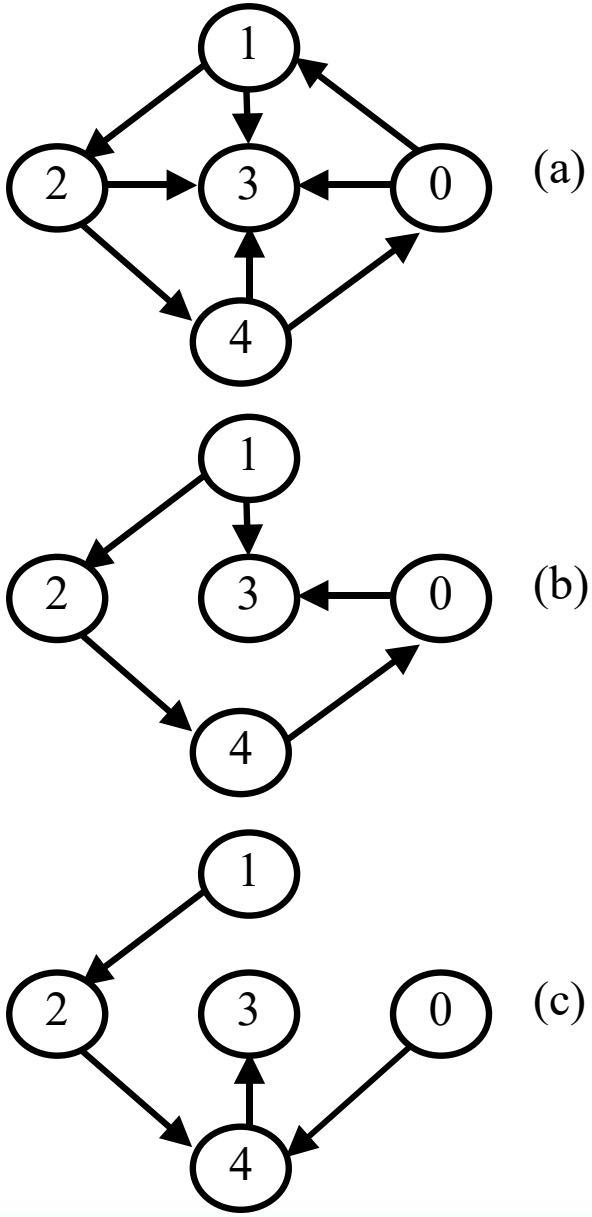
(b)

- 子图

有两个图G=(V, {E})和图G′=(V′, {E′}),

若V′⊆V且E′⊆E, 则称图G′为G的子图。

- 邻接点



●顶点的度、入度和出度

在无向图中,顶点所具有的边的数目称为该顶点的度。

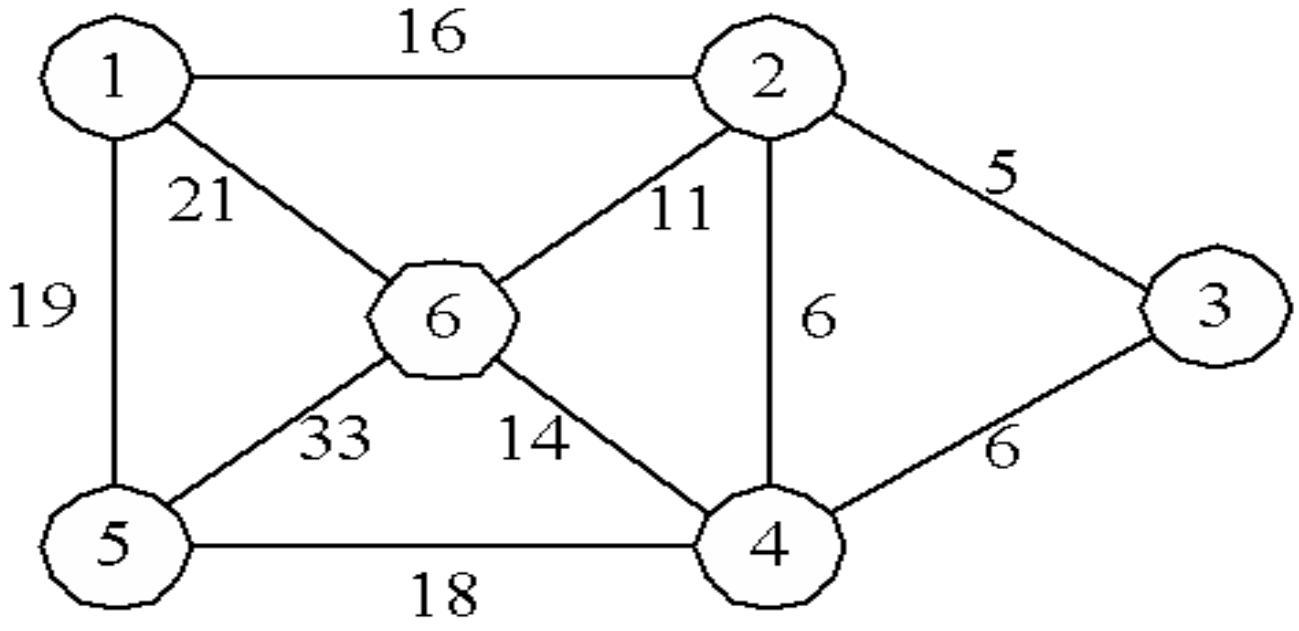
在有向图中,以顶点*v*为头的弧的数目,称为该顶点的入度。以顶点*v*为尾的弧的数目,称为该顶点的出度。一个顶点的入度与出度的和为该顶点的度。

一般地， 若图G中有n个顶点，e条边或弧，则图中顶点的度与边的关系如下：

$$e = \frac{\sum_{i=1}^{n} TD(v_i)}{2}$$

- 权与网

在实际应用中，有时图的边或弧上往往与具有一定意义的数有关，即每一条边都有与它相关的数，称为权，这些权可以表示从一个顶点到另一个顶点的距离或耗费等信息。我们将这种带权的图叫做赋权图或网。



●路径与回路

无向图G=(V,{E})中从顶点v到v′的路径是一个顶点序列$v_{i0},v_{i1},v_{i2},\cdots,v_{in}$，其中$(v_{ij-1}, v_{ij})\in E$, $1\le j\le n$。如果图G是有向图，则路径也是有向的，顶点序列应满足$<v_{ij-1}, v_{ij}>\in E$， $1\le j\le n$。

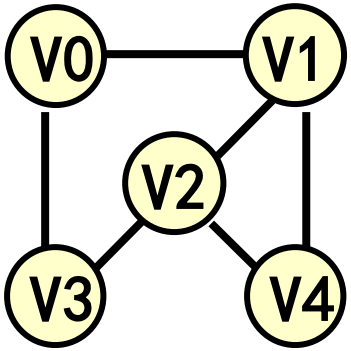
路径的长度: 路径上经过的弧或边的数目。

回路或环:

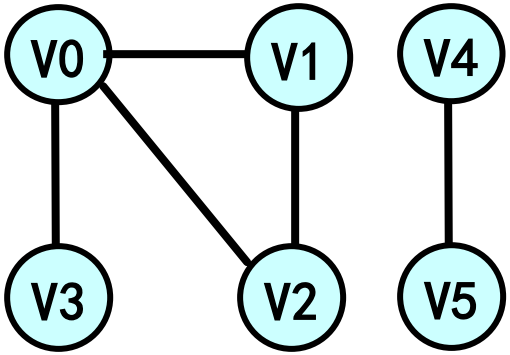
简单路径:表示路径的顶点序列中的顶点各不相同。

简单回路:除了第一个和最后一个顶点外，其余各顶点均不重复出现的回路。

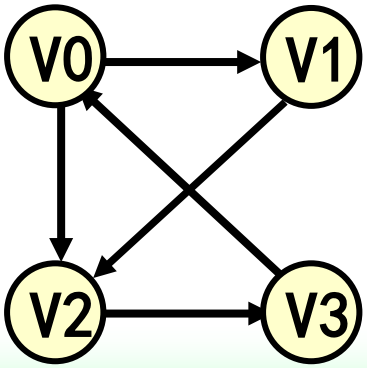
- 连通图、强连通图
  在无（有）向图G=<V, E>中，若对任何两个顶点u、v都存在从u到v的路径，则称G是连通图(强连通图)。



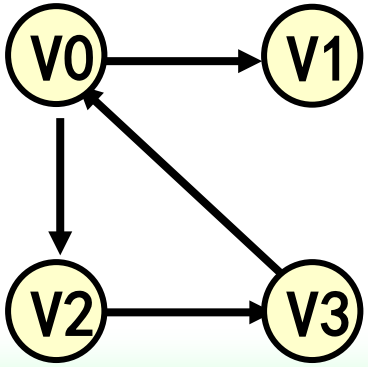
(a)连通图



(b)非连通图



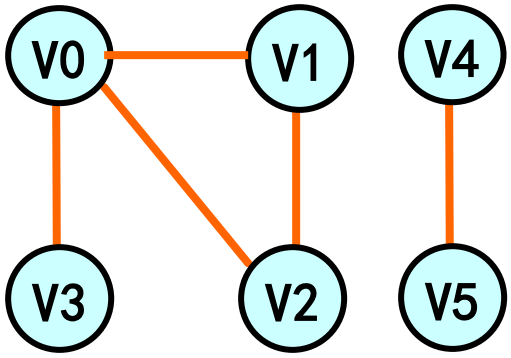
(c)强连通图



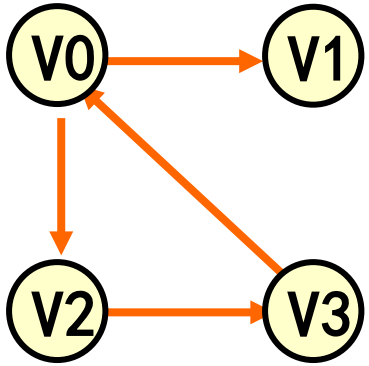
(d)非强连通图

连通分量:无向图中的极大连通子图。

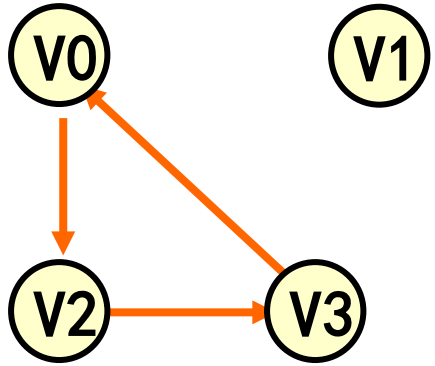
强连通分量: 有向图的极大强连通子图。



非连通图，有两个连通分量

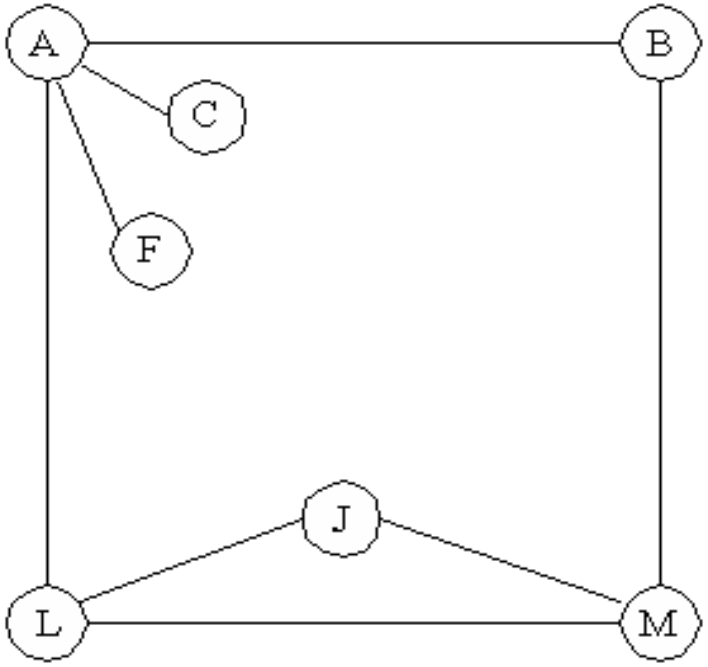


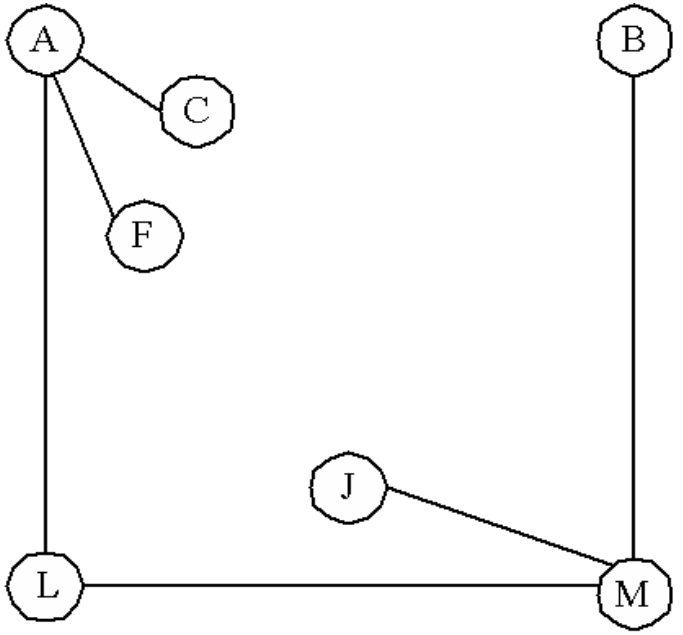
非强连通图



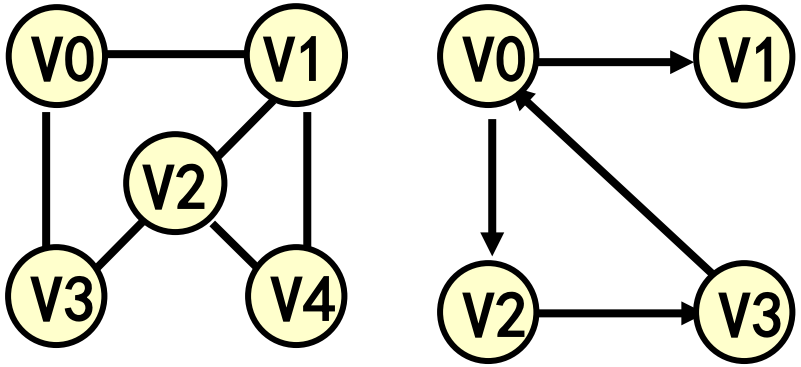
有两个强连通分量

●生成树: 连通图的生成树是一个极小连通子图，它含有图中全部顶点，但只有足以构成一棵树的n-1条边。





图的应用示例



- 例1 交通图（公路、铁路）
  - 顶点：地点
  - 边：连接地点的公路
  - 交通图中的有单行道双行道，分别用有向边、无向边表示；

- 例2　电路图
  - 顶点：元件
  - 边：连接元件之间的线路

- 例3 通讯线路图
  - 顶点：地点
  - 边：地点间的连线

- 例4 各种流程图
  - 如产品的生产流程图
  - 顶点：工序
  - 边：各道工序之间的顺序关系

7.2 图的存储结构

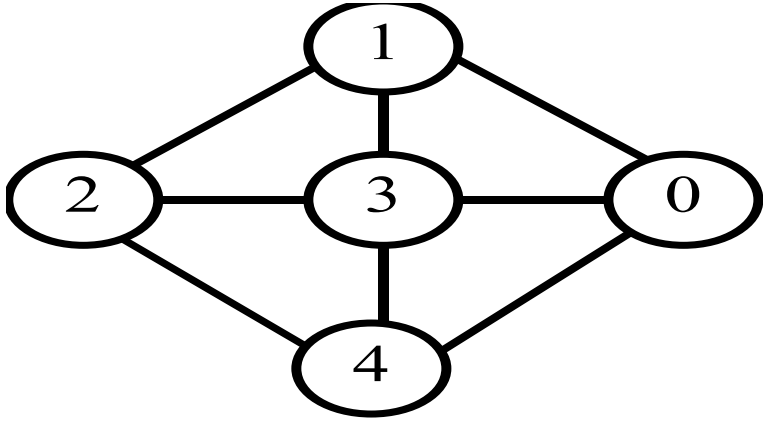
1. 邻接矩阵表示法（数组表示法）

- 图G是一个具有n个顶点的无权图，G的邻接矩阵是具有如下性质的n×n矩阵A：

$$A[i,j]=\begin{cases}1, & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\in E\\ 0, & \text{其它}\end{cases}$$

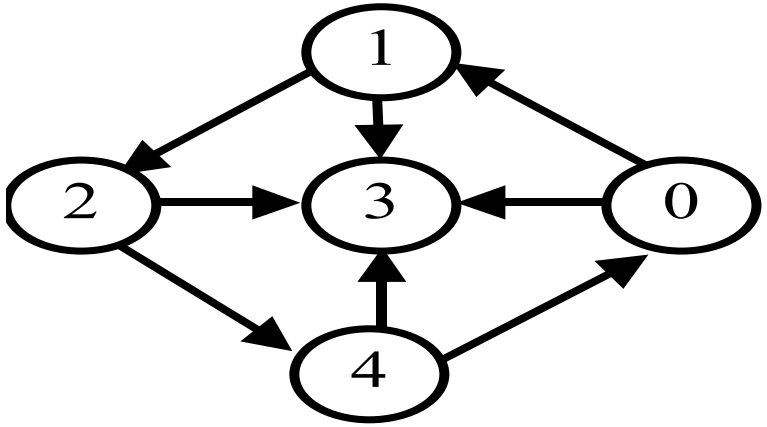
- 若图G是一个有n个顶点的网，则它的邻接矩阵是具有如下性质的n×n矩阵A：

$$A[i,j]=\begin{cases}w_{ij}, & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\in E\\ \infty, & \text{其它}\end{cases}$$



(a)

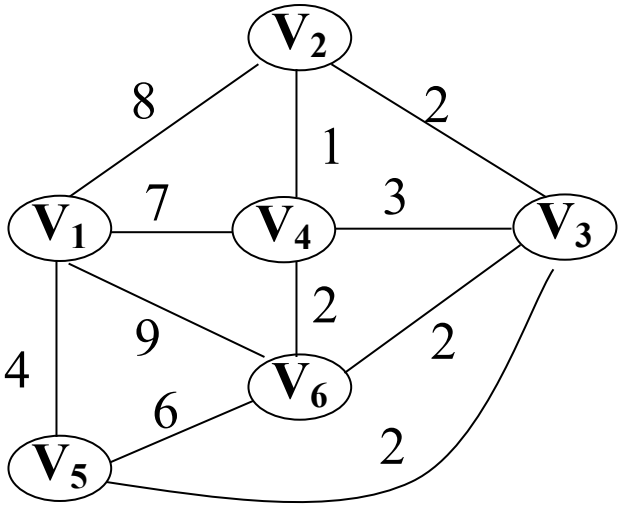
$$A1 = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$



(b)

$$A2 = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

- 网的数组表示法



	1	2	3	4	5	6
1	∞	8	∞	7	4	9
2	8	∞	2	1	∞	∞
3	∞	2	∞	3	2	2
4	7	1	3	∞	∞	2
5	4	∞	2	∞	∞	6
6	9	∞	2	2	6	∞

邻接矩阵表示法类型描述

```
#define  MAX_VERTEX_NUM   20          //最多顶点个数
#define  INFINITY   INT_MAX           //表示极大值∞
typedef  enum{DG,  DN,  UDG,  UDN}  GraphKind;
typedef  struct  ArcCell{
      VRType    adj;
      InfoType   *info;
} ArcCell,  AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];

typedef  struct{
     VertexType  vexs［MAX_VERTEX_NUM];  //顶点向量
     AdjMatrix   arcs;    //邻接矩阵
     int   vexnum,   arcnum;  //图的顶点数和弧数
     GraphKind  kind;   //图的种类标志
} MGraph;
```

邻接矩阵的特点如下：

(1) 图的邻接矩阵表示是唯一的。

(2) 无向图的邻接矩阵一定是一个对称矩阵。因此，按照压缩存储的思想，在具体存放邻接矩阵时只需存放上(或下)三角形阵的元素即可。

(3) 判断两顶点v、u是否为邻接点：只需判二维数组对应分量是否为1；

(4) 顶点不变，在图中增加、删除边：只需对二维数组对应分量赋值1或清0；

(5)不带权的有向图的邻接矩阵一般来说是一个稀疏矩阵,因此,当图的顶点较多时,可以采用三元组表的方法存储邻接矩阵。

(6) 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**vi**的度。

(7)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v_i的出度(或入度)。

(8)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。这是用邻接矩阵存储图的局限性。

2. 邻接表存储方法

图的邻接表存储方法是一种顺序分配与链式分配相结合的存储方法。在邻接表中，对图中每个顶点建立一个单链表，第i个单链表中的结点表示依附于顶点v_i的边(对有向图是以顶点v_i为尾的弧)。每个单链表上附设一个表头结点。表结点和表头结点的结构如下：

表头结点

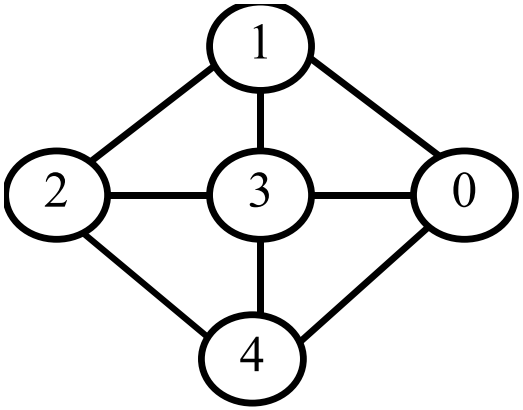
data	firstarc

表结点

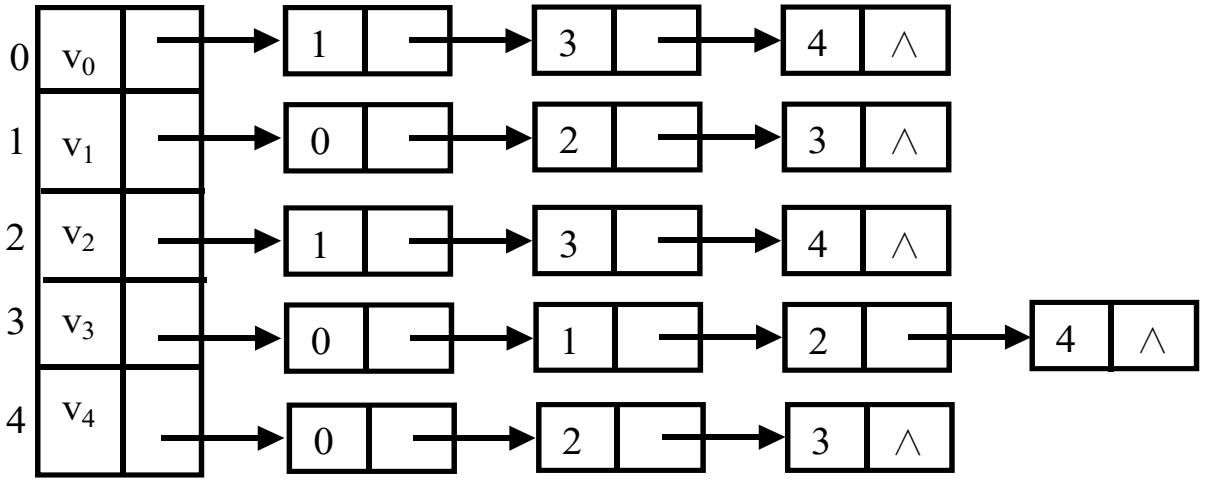
adjvex	nextarc	info

data存储顶点v_i的名称或其他信息; **firstarc**指向链表中第一个结点。

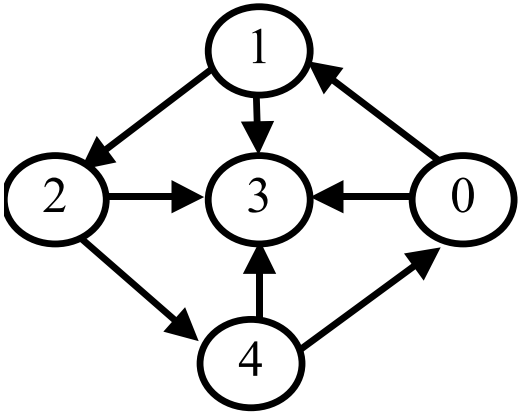
adjvex指示与顶点vi邻接的点在图中的位置;**nextarc**指示下一条边或弧的结点; **info**存储与边或弧相关的信息,如权值等。


1
2
3
0
4

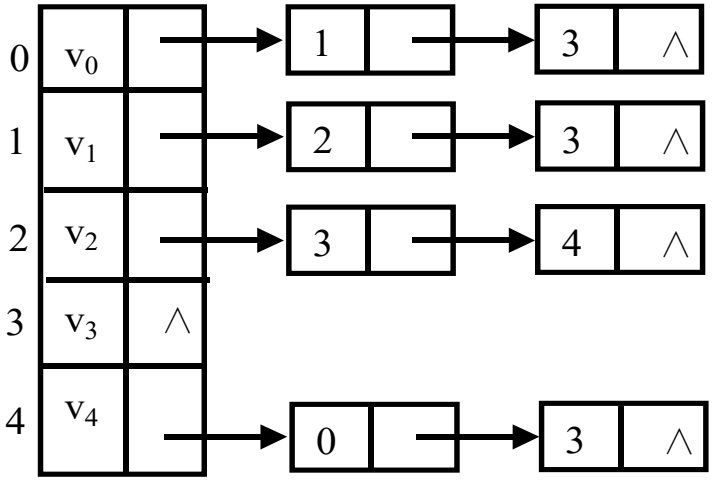
(a)


0
v0
1
3
4
1
v1
0
2
3
2
v2
1
3
4
3
v3
0
1
2
4
4
v4
0
2
3

(a)

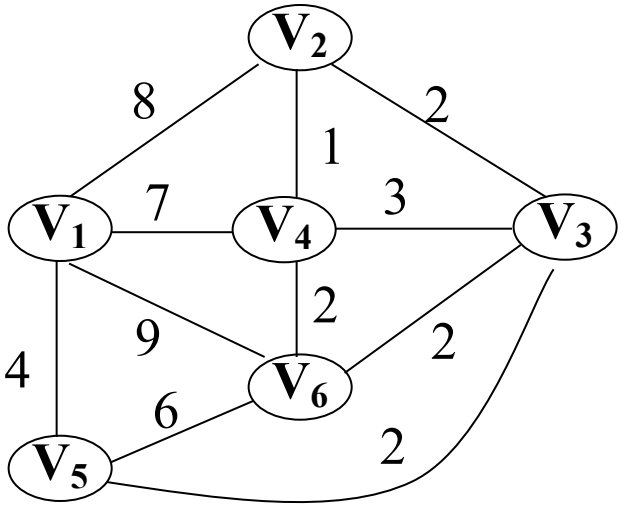

1
2
3
0
4

(b)


0
v0
1
3
1
v1
2
3
2
v2
3
4
3
v3
4
v4
0
3

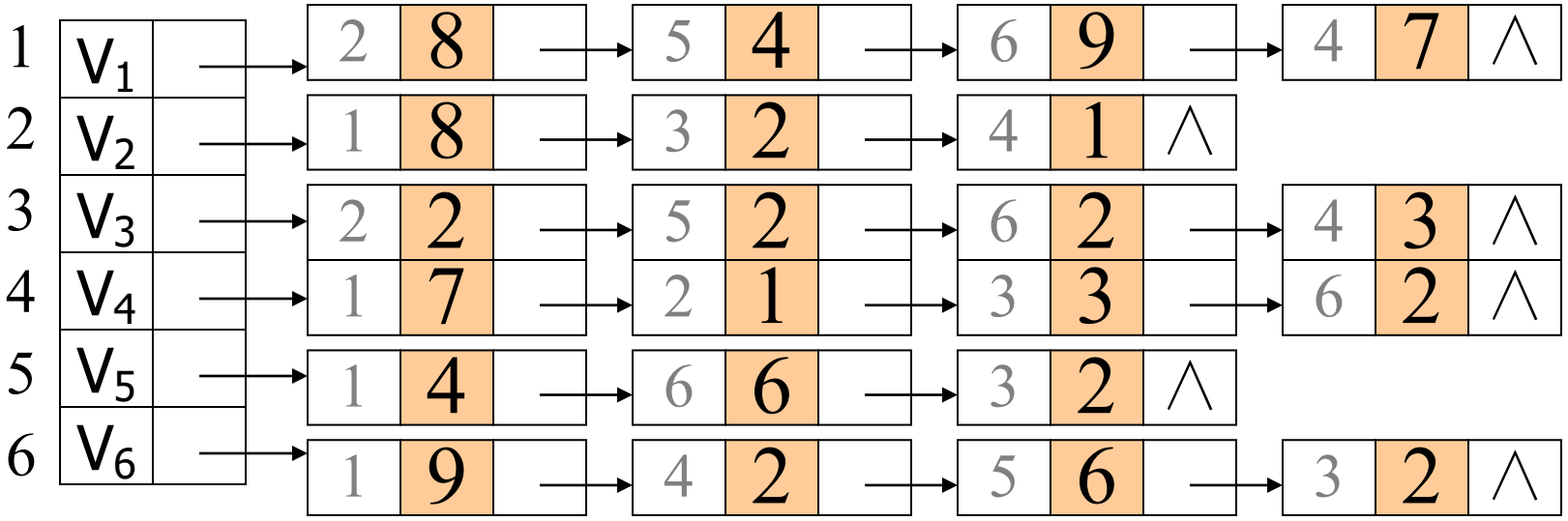
(b)

网的邻接链表表示



表头顶点的 邻接顶点编号	和边相关 的信息	指向下一个 邻接顶点的指针

(a) 表结点结构



(b) 邻接链表

邻接表存储结构的类型定义：

```
typedef struct ArcNode{//表结点结构类型
        int adjvex;                 //该弧(边)的终点位置
        struct ArcNode *nextarc;    //指向下一条弧的指针
        InfoType info;              //该弧的相关信息
} ArcNode;
typedef struct Vnode {//头结点的类型
        Vertex data;                //顶点信息
        ArcNode *firstarc;          //指向第一条弧
} VNode, AdjList[MAX_VERTEX_NUM];
typedef struct {//邻接表
      AdjList vertices;
      int vexnum, arcnum;           //图中顶点数n和边数e
      int kind;                     //图的类型
} ALGraph;
```

邻接表的特点如下：

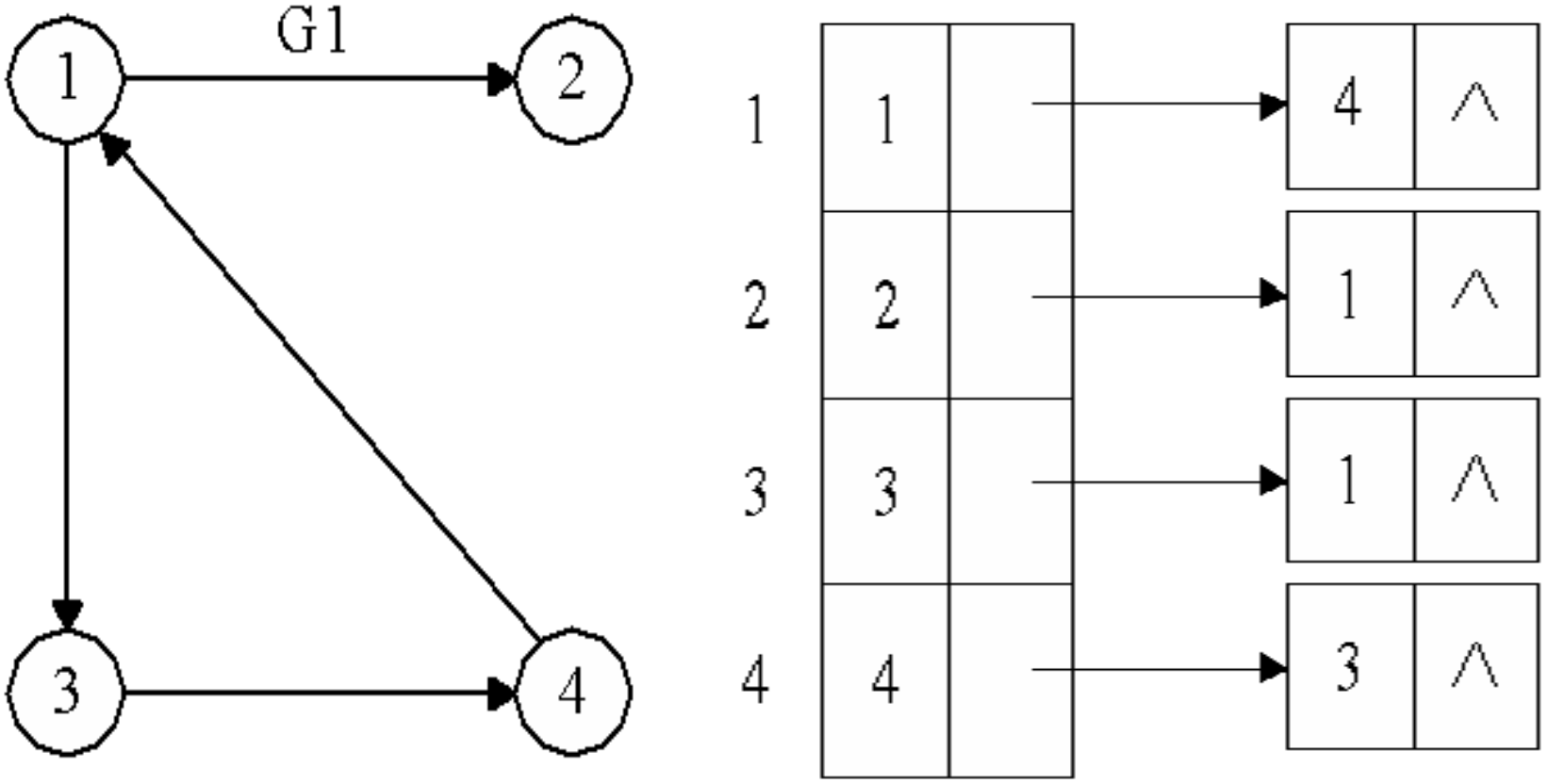
(1) 邻接表表示不唯一。这是因为在每个顶点对应的单链表中，各边结点的链接次序可以是任意的，取决于建立邻接表的算法以及边的输入次序。

(2) 对于有n个顶点和e条边的无向图，其邻接表有n个头结点和2e个表结点。显然，在总的边数小于n(n-1)/2的情况下，邻接表比邻接矩阵要节省空间。

(3) 对于无向图，邻接表的顶点v_i对应的第i个链表的表结点数目正好是顶点v_i的度。

(4) 对于有向图，邻接表的顶点v_i对应的第i个链表的表结点数目仅仅是v_i的出度。其入度为邻接表中所有adjvex域值为i的表结点数目。（逆邻接表）

图G1的逆邻接表表示法



3. 十字链表
4. 邻接多重表

十字链表

将有向图的邻接表和逆邻接表结合在一起，就得到了有向图的另一种链式存储结构——十字链表。

头结点

vexinfo ： 顶点的信息

firstin ： 第一条关联入弧结点

firstout ： 第一条关联出弧结点

表结点

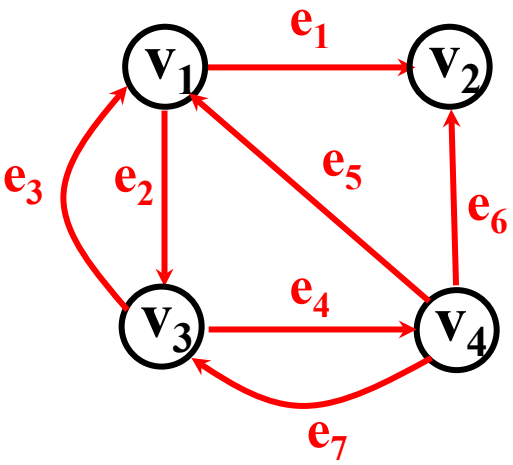
tailvex ： 弧尾顶点位置

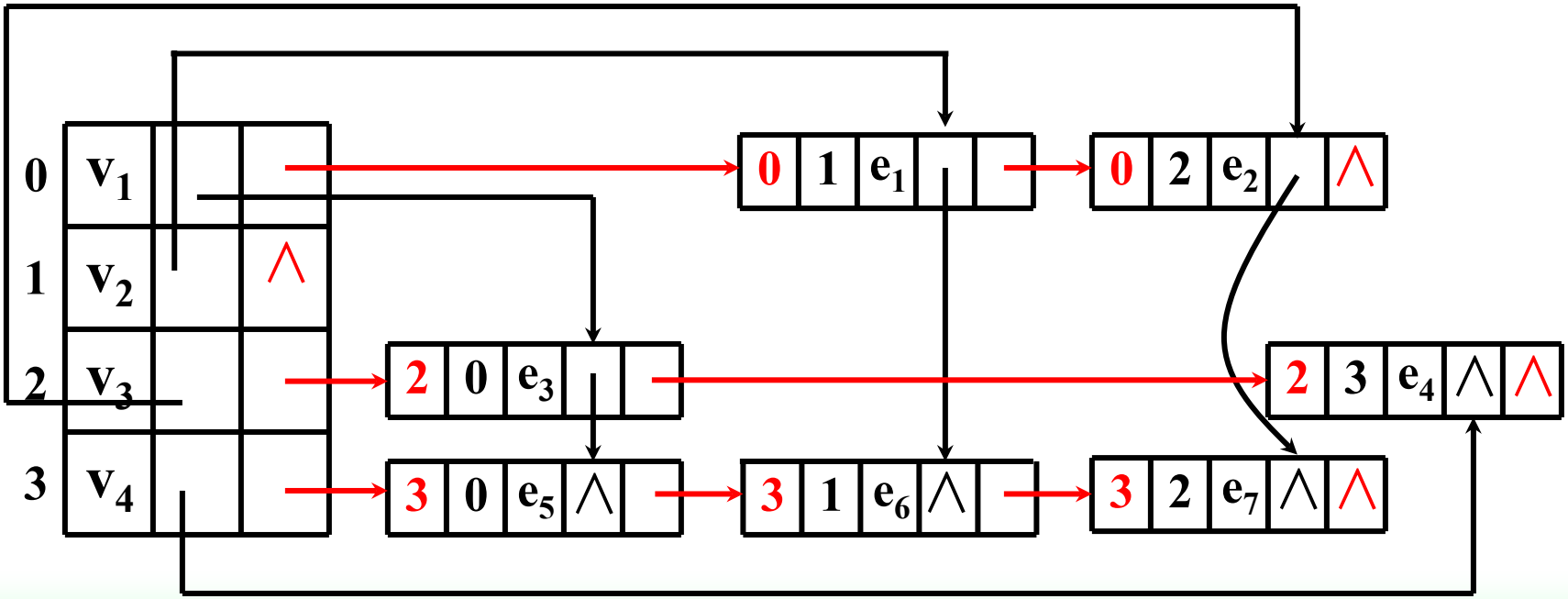
headvex ： 弧头顶点位置

arcinfo ： 弧的信息

tnext ： 弧尾相同的下一条弧

hnext ： 弧头相同的下一条弧


v1
v2
v3
v4
e1
e2
e3
e4
e5
e6
e7


0 v1
1 v2
2 v3
3 v4
0 1 e1
0 2 e2
2 0 e3
2 3 e4
3 0 e5
3 1 e6
3 2 e7

邻接多重表

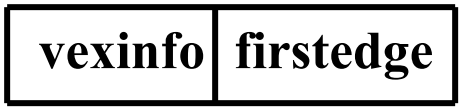
邻接表是无向图的一种很有效的存储结构，在邻接表中容易求得顶点和边的各种信息；

但在邻接表中，每一条边都有两个结点表示，因此在某些对边进行的操作(例如对搜索过的边做标记)中就需要对每一条边处理两遍；

故引入邻接多重表实现无向图的存储结构。

邻接多重表的结构与十字链表相似

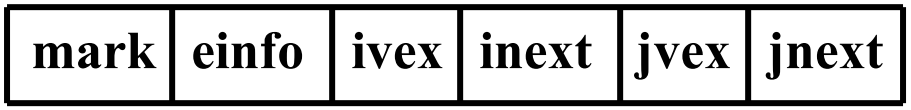
头结点



vexinfo ： 顶点的信息

firstedge ： 第一条关联边结点

表结点



mark ： 标志域，是否遍历过

einfo ： 边的信息

ivex ： 边的第一个顶点位置

inext ： 顶点 i 的下一条关联边

jvex ： 边的另一个顶点位置

jnext ： 顶点 j 的下一条关联边

作业：

7.1

7.3 图 的 遍 历

1. 图的遍历的概念

从给定图中任意指定的顶点(称为初始点)出发, 按照某种搜索方法沿着图的边访问图中的所有顶点, 使每个顶点仅被访问一次, 这个过程称为图的遍历。如果给定图是连通的无向图或者是强连通的有向图, 则遍历过程一次就能完成, 并可按访问的先后顺序得到由该图所有顶点组成的一个序列。

根据搜索方法的不同, 图的遍历方法有两种：一种叫做深度优先搜索法**DFS(Depth-First Search)**；另一种叫做广度优先搜索法**BFS (Breadth-First Search)**。

2. 深度优先搜索遍历的过程：

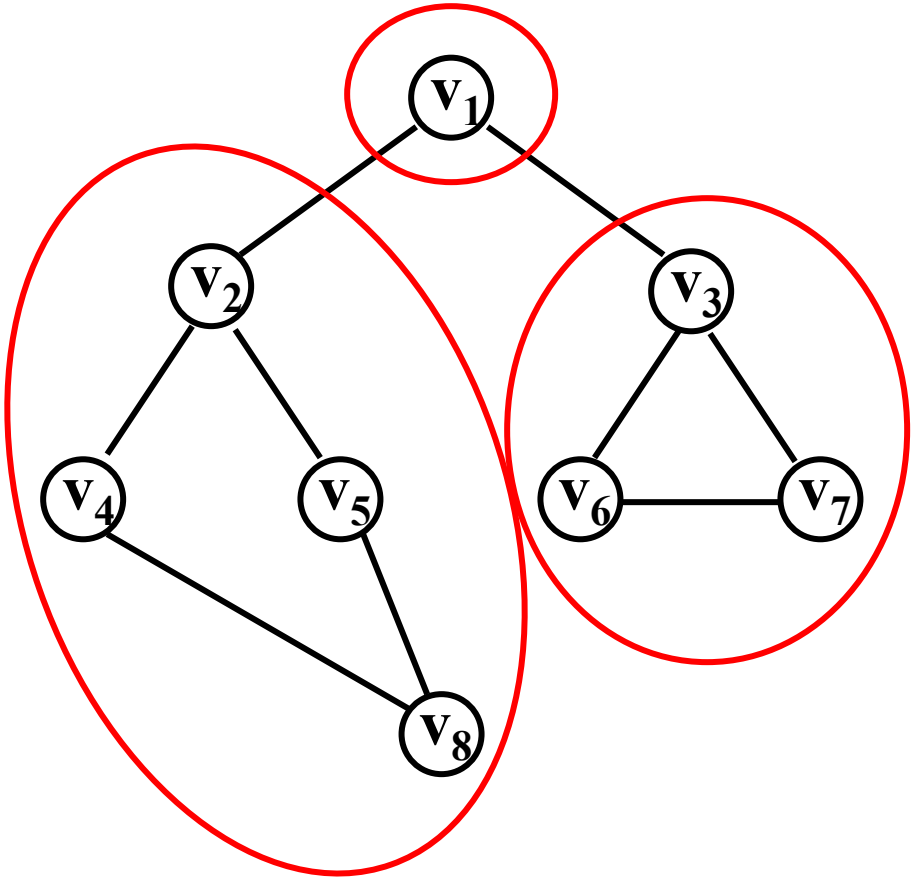
(1) 从图中某个初始顶点v出发, 首先访问初始顶点v；

(2) 然后选择一个与顶点v相邻且没被访问过的顶点w为初始顶点, 再从w出发进行深度优先搜索；

(3) 重复以上第(2)步，直到图中与当前顶点v邻接的所有顶点都被访问过为止。

显然, 这个遍历过程是个递归过程。

7.3.1 深度优先搜索



图可分为三部分:

基结点

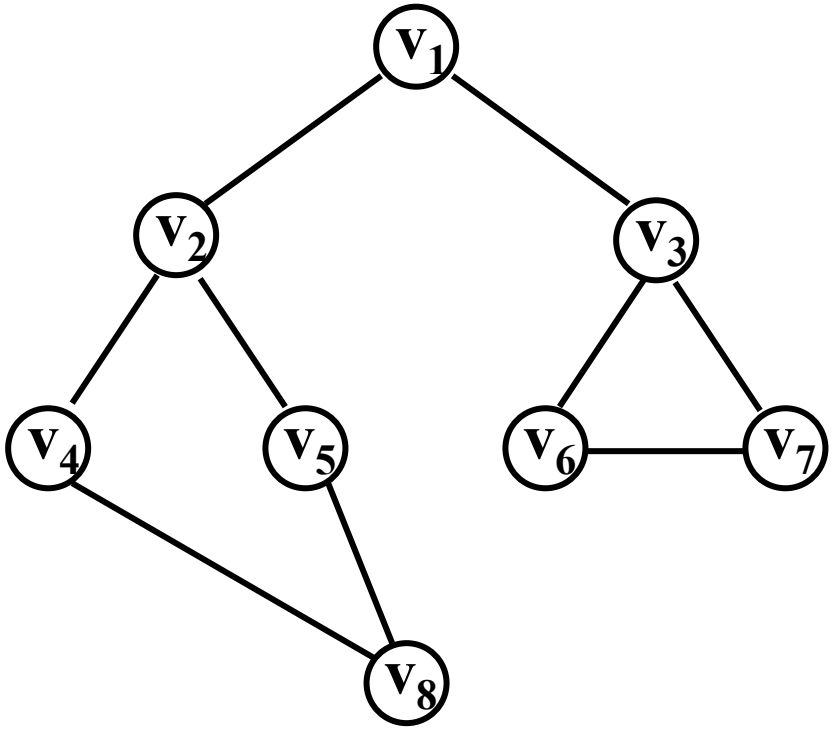
第一个邻接结点导出的子图

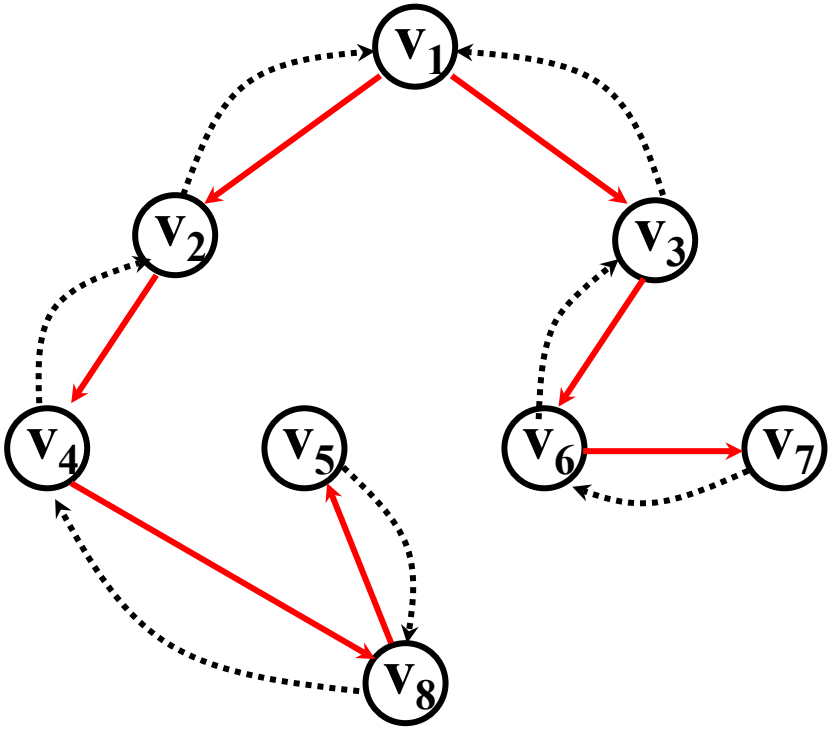
其它邻接顶点导出的子图

深度优先搜索是类似于树的一种先序遍历

深度优先搜索顺序: v_1 v_2 v_4 v_8 v_5 v_3 v_6 v_7

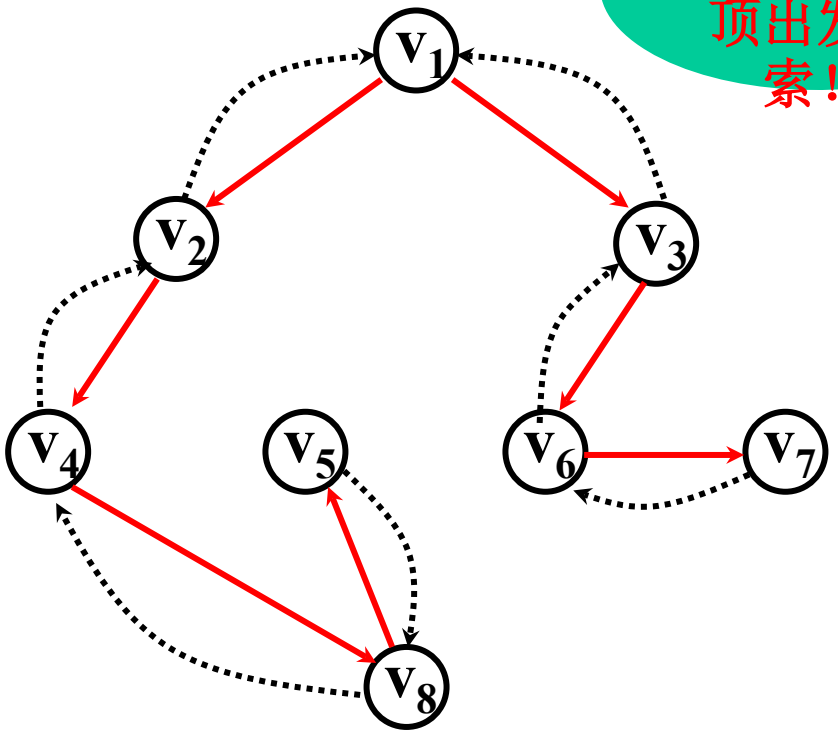
过程分析



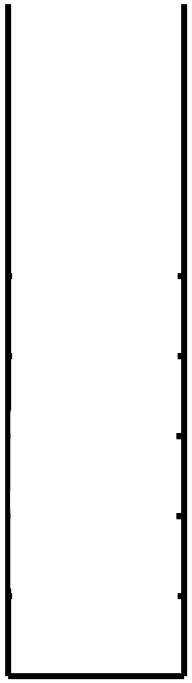


深度优先搜索顺序：　v_1 v_2 v_4 v_8 v_5 v_3 v_6 v_7

栈实现深度优先搜索



总是从栈顶出发搜索！

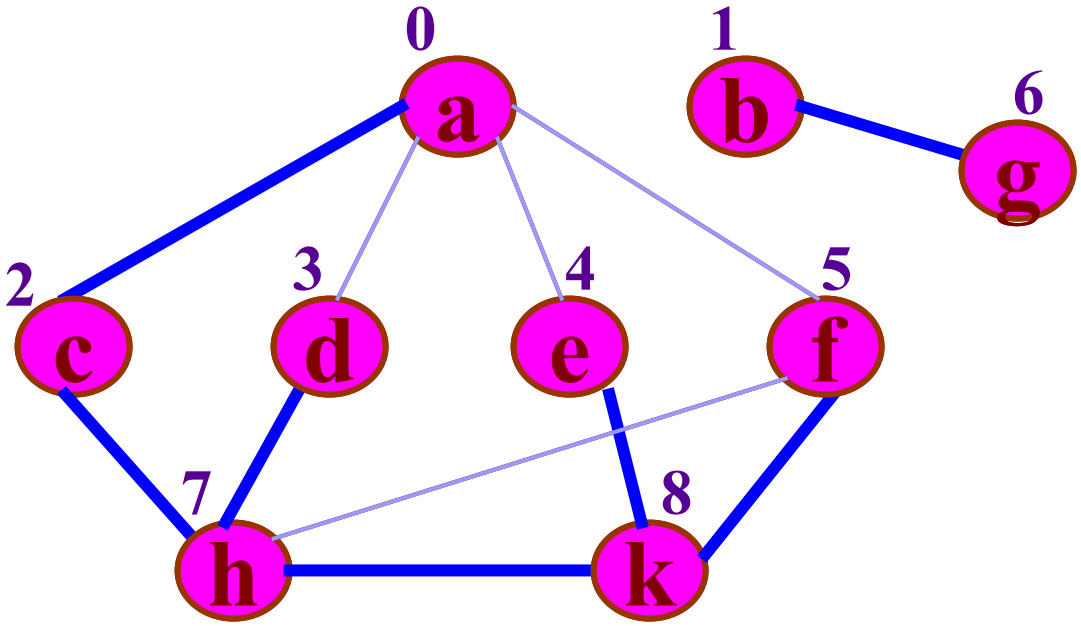


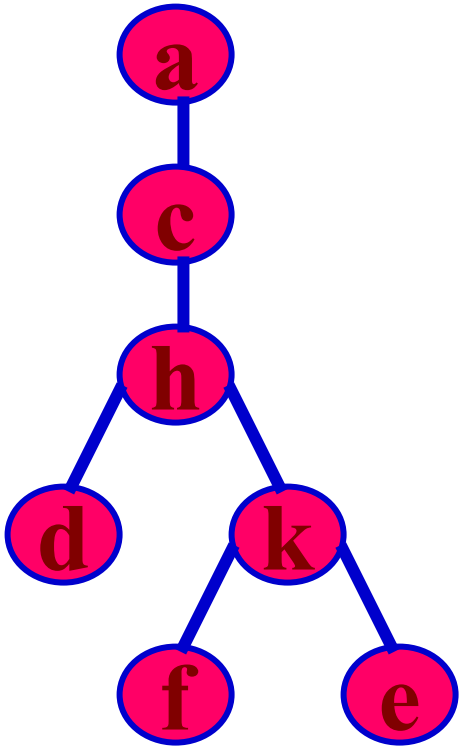
深度优先搜索顺序：　v_1　v_2　v_4　v_8　v_5　v_3　v_6　v_7

非连通图的深度优先搜索遍历

首先将图中每个顶点的访问标志设为 FALSE，之后搜索图中每个顶点，如果未被访问，则以该顶点为起始点，进行深度优先搜索遍历，否则继续检查下一顶点。

例如：





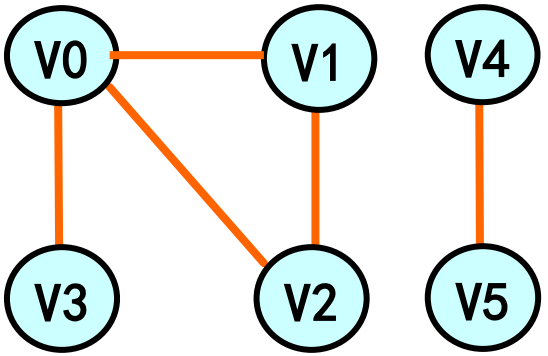
访问标志：

0	1	2	3	4	5	6	7	8
T	T	T	T	T	T	T	T	T

访问次序：

a	c	h	d	k	f	e	b	g

```
void DFSTraverse(Graph G) {
   for(v=0;v<G.vexnum;++v)
      visited[v] = false;
   for(v=0;v<G.vexnum;++v)
      if (visited[v]==false) DFS(G,v);
}//DFSTraverse
```




```
void DFS(Graph G,int v) {
   visited[v] = true;
   for(w为v的第一个邻接顶点; w存在 ; w取v的下一个邻接顶点)
      if (visited[w]==false) DFS(G,w
}//DFS
```

邻接链表表示：查找每个顶点的邻接点所需时间为O(e)，e为边(弧)数，算法时间复杂度为O(n+e)

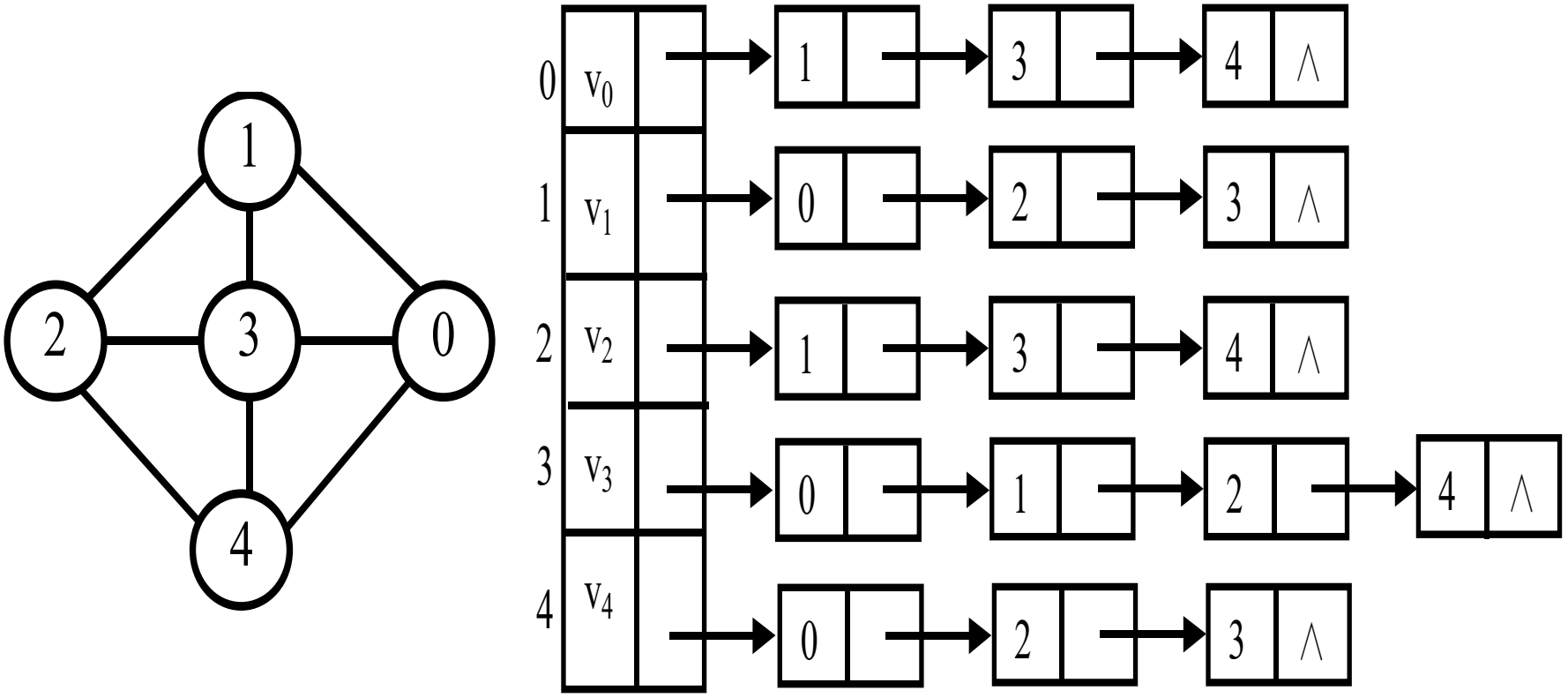
表示：查找每个顶点的邻接点时间为$O(n^2)$，n为顶点数，算间复杂度为$O(n^2)$

用邻接表方式实现深度优先搜索

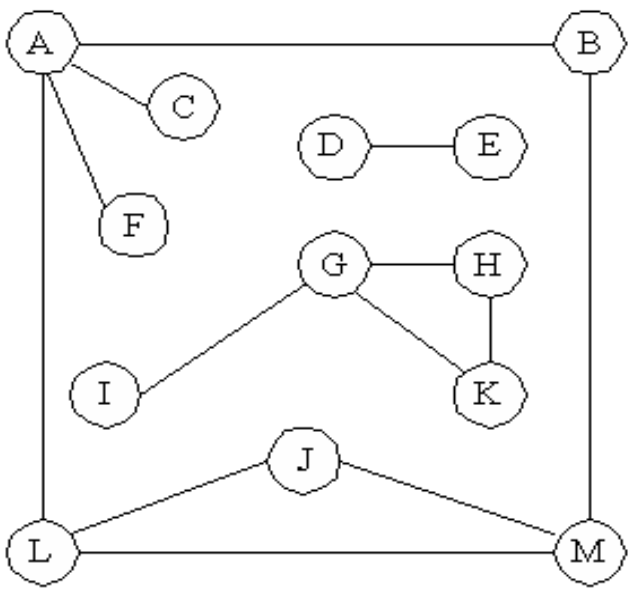
```
void DFS(ALGraph G, int v)
{    ArcNode *p;  visited[v]=1;     /*置已访问标记*/
     printf("%d  ",v);                  /*输出被访问顶点的编号*/
     p=G.vertices[v].firstarc;
          /*p指向顶点v的第一条弧的弧头结点*/
     while (p!=NULL) {
          if (visited[p->adjvex]==0)   DFS(G, p->adjvex);
               /*若p->adjvex顶点未访问, 递归访问它*/
          p=p->nextarc;
               /*p指向顶点v的下一条弧的弧头结点*/
     }
}
```

算法分析

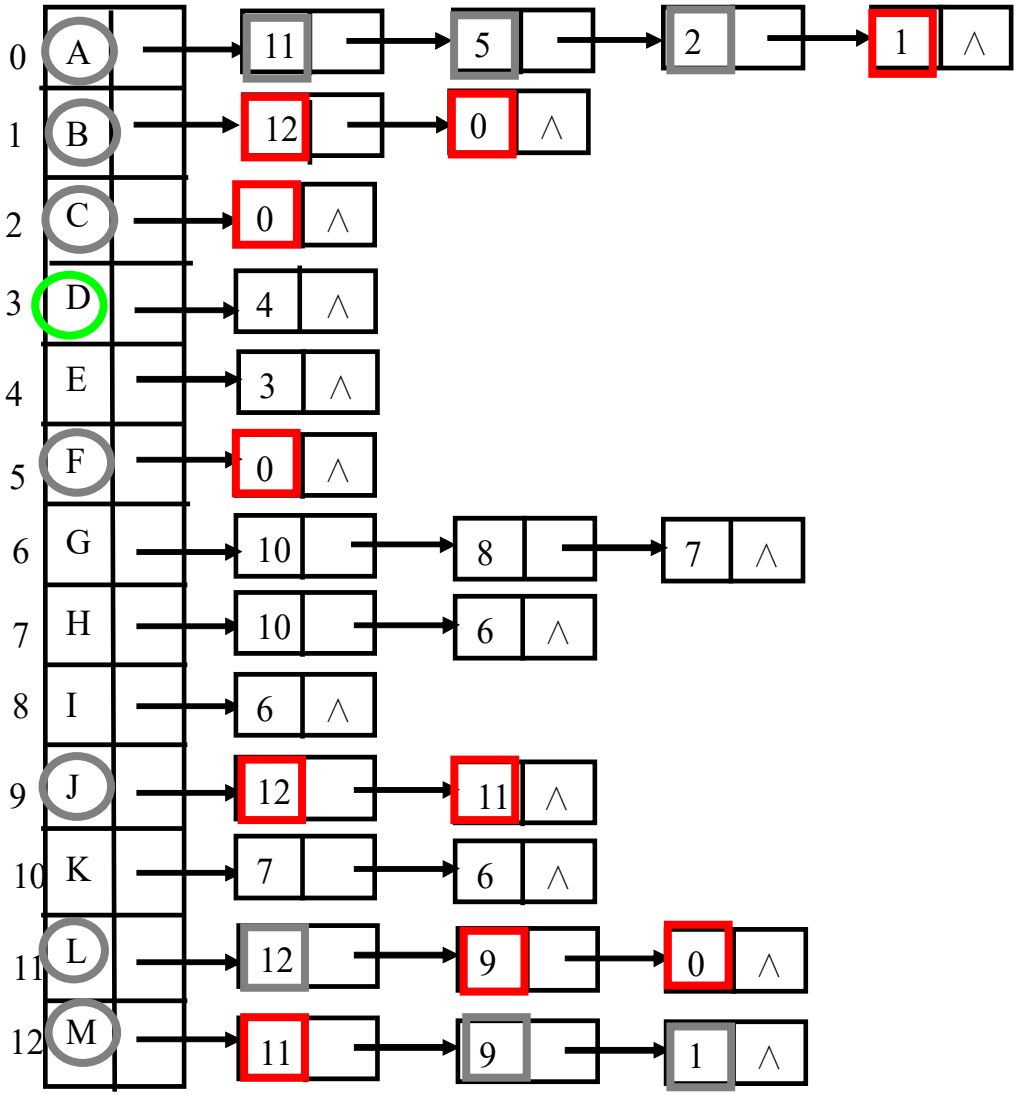
- 图中有 n 个顶点，e 条边。
- 如果用邻接表表示图，沿 *firsarc*链可以找到某个顶点 v 的所有邻接顶点 w。由于总共有 $2e$ 个边结点，所以扫描边的时间为$O(e)$。所以遍历图的时间复杂性为$O(n+e)$。
- 如果用邻接矩阵表示图，则查找每一个顶点的所有的边，所需时间为$O(n)$，则遍历图中所有的顶点所需的时间为$O(n^2)$。



从顶点2出发：2,1,0,3,4



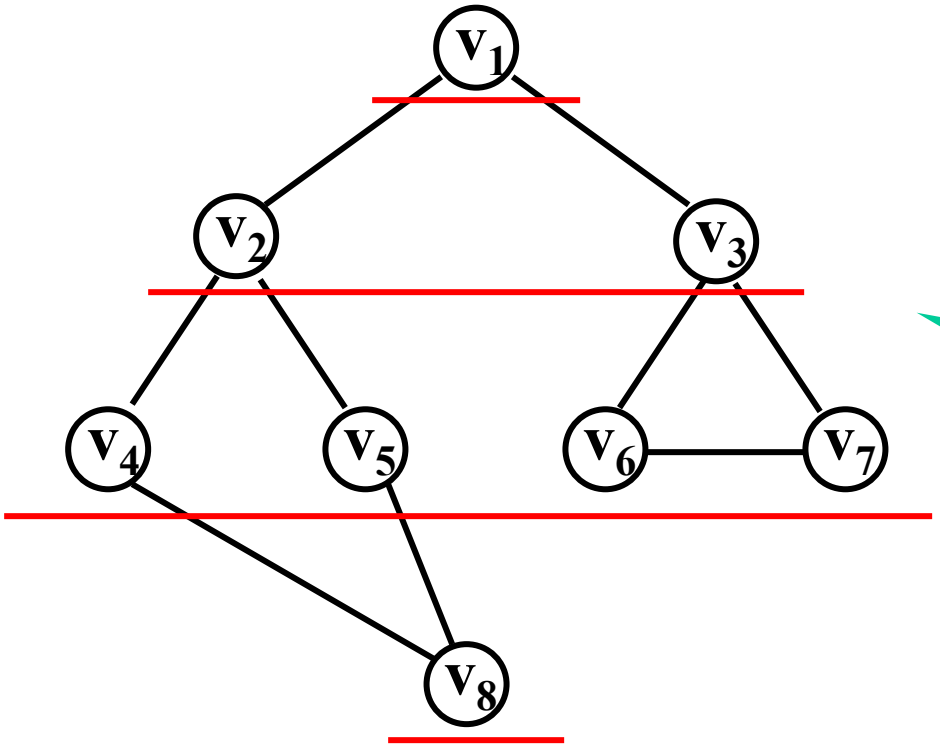
A L M J B F C;
D E;
G K H I;



广度优先搜索(BFS)

- 从图中某顶点vi出发：
  - ① 访问顶点vi ；
  - ② 访问vi 的所有未被访问的邻接点w_1 ，w_2 ， ...w_k ；
  - ③ 依次从这些邻接点（在步骤②中访问的顶点）出发，访问它们的所有未被访问的邻接点；依此类推，直到图中所有访问过的顶点的邻接点都被访问；
- 为实现③，需要保存在步骤②中访问的顶点，而且访问这些顶点的邻接点的顺序为：先保存的顶点，其邻接点先被访问。

7.3.2 广度优先搜索



把图人为的分层，

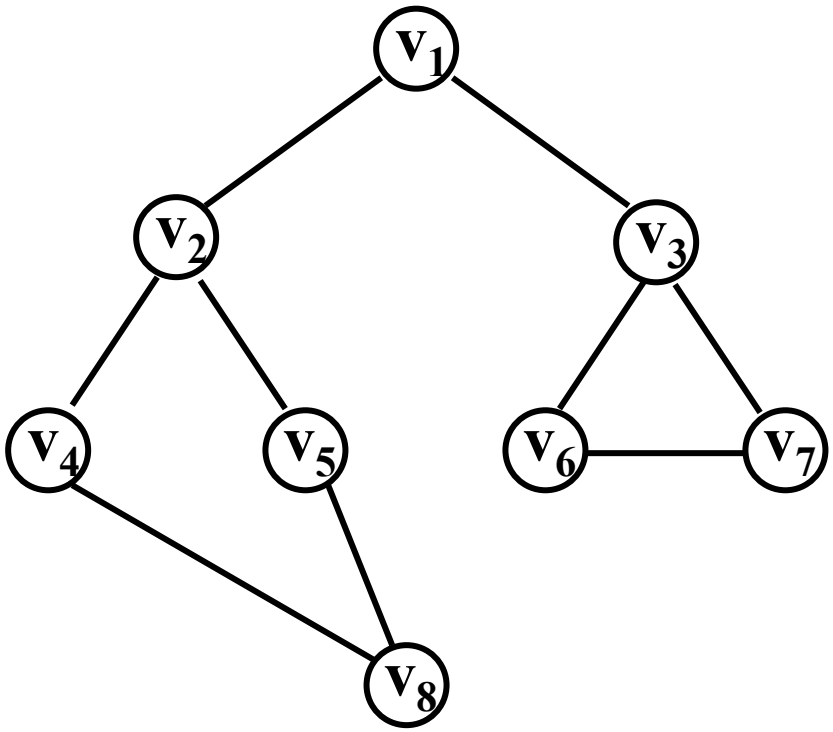
按层遍历。

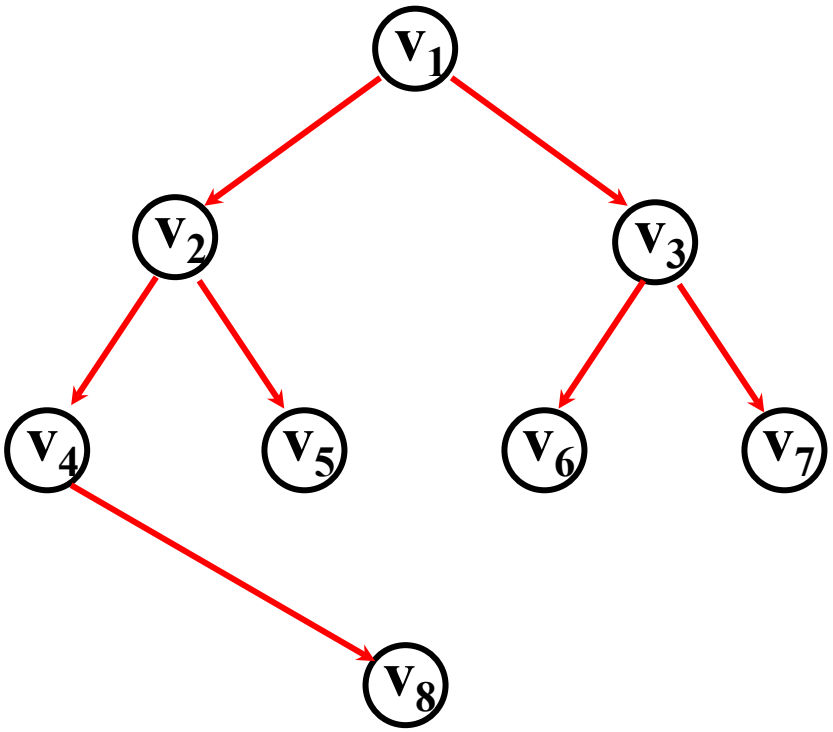
只有父辈结点
被访问后才会
访问子孙结点！

广度优先搜索类似于树的层次遍历，

广度优先搜索顺序： v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8

过程分析



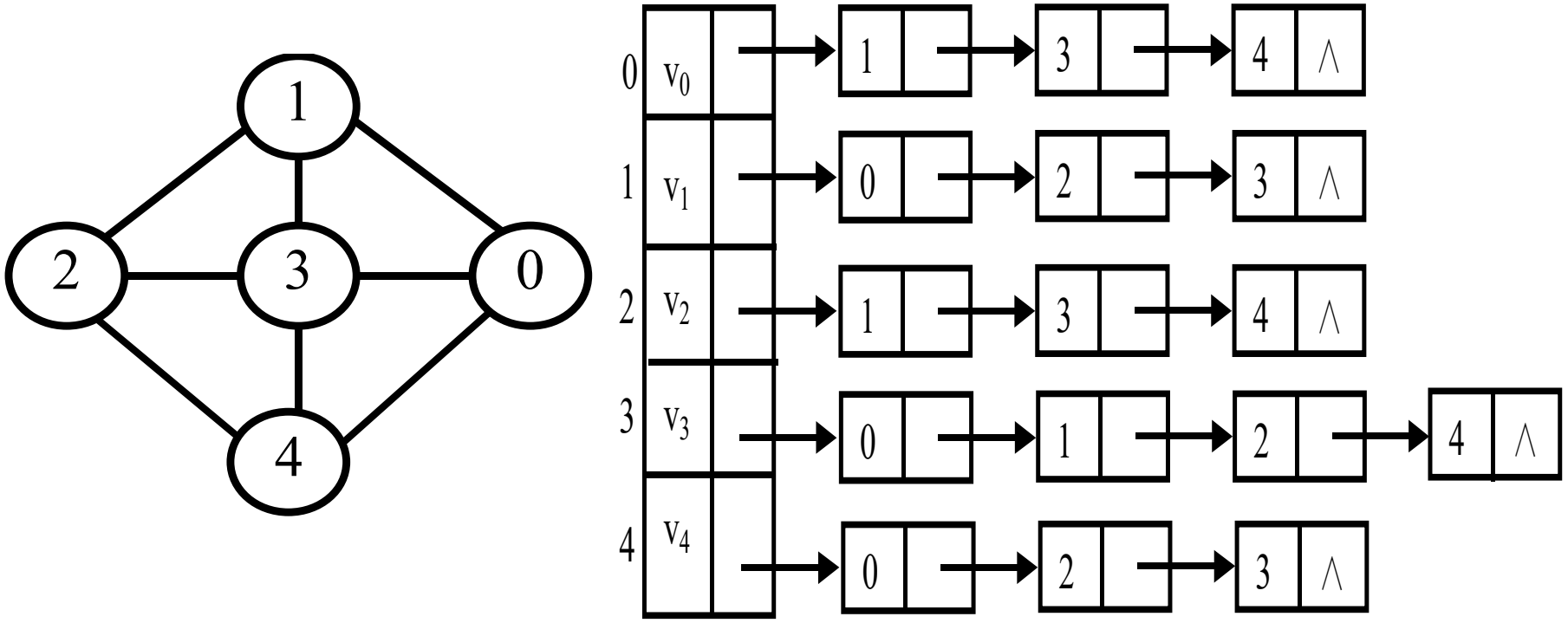


广度优先搜索顺序： v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8

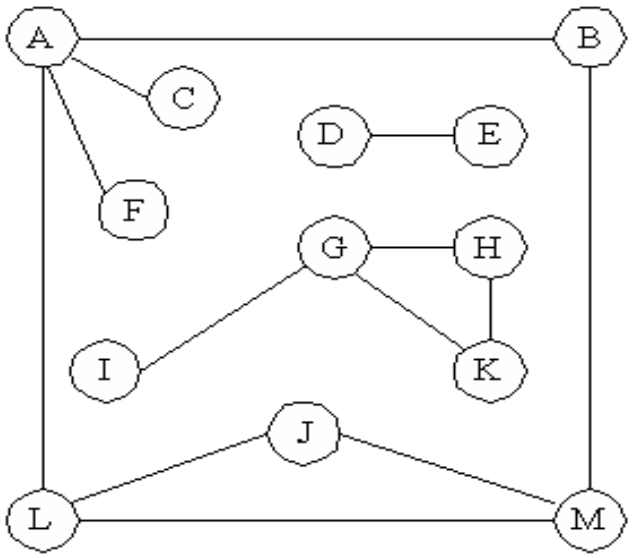
```
Void BFSTraverse(ALGraph G)
{ for (v=0; v<G.vexnum; ++v)  visited[v] = FALSE ; InitQueue(Q);
  for(v=0; v<G.vernum; ++v){
    if(!visited[v]){ visited[v]=1; printf("%d  ",v); EnQueue(Q,v);
      while(!QueueEmpty(Q)){
        DeQueue(Q,u); p=G.vertices[u].firstarc;
        while(p){
          if(!visited[p->adjvex]){
            visited[p->adjvex]=1; printf("%d  ", p->adjvex);
            EnQueue(Q, p->adjvex);} //if
          p=p->nextarc;} //while(p)
      }//while(!Queue…)
    }//if(!visite…)
  }//for
}
```

算法分析

- 图中每个顶点至多入队一次，因此外循环次数为n。
- 当图G采用邻接表方式存储，则当结点v出队后，内循环次数等于结点v的度。由于访问所有顶点的邻接点的总的时间复杂度为$O(d_0+d_1+d_2+...+d_{n-1})=O(e)$， 因此图采用邻接表方式存储，广度优先搜索算法的时间复杂度为O(n+e)；
- 当图G采用邻接矩阵方式存储，由于找每个顶点的邻接点时，内循环次数等于n，因此广度优先搜索算法的时间复杂度为$O(n^2)$。



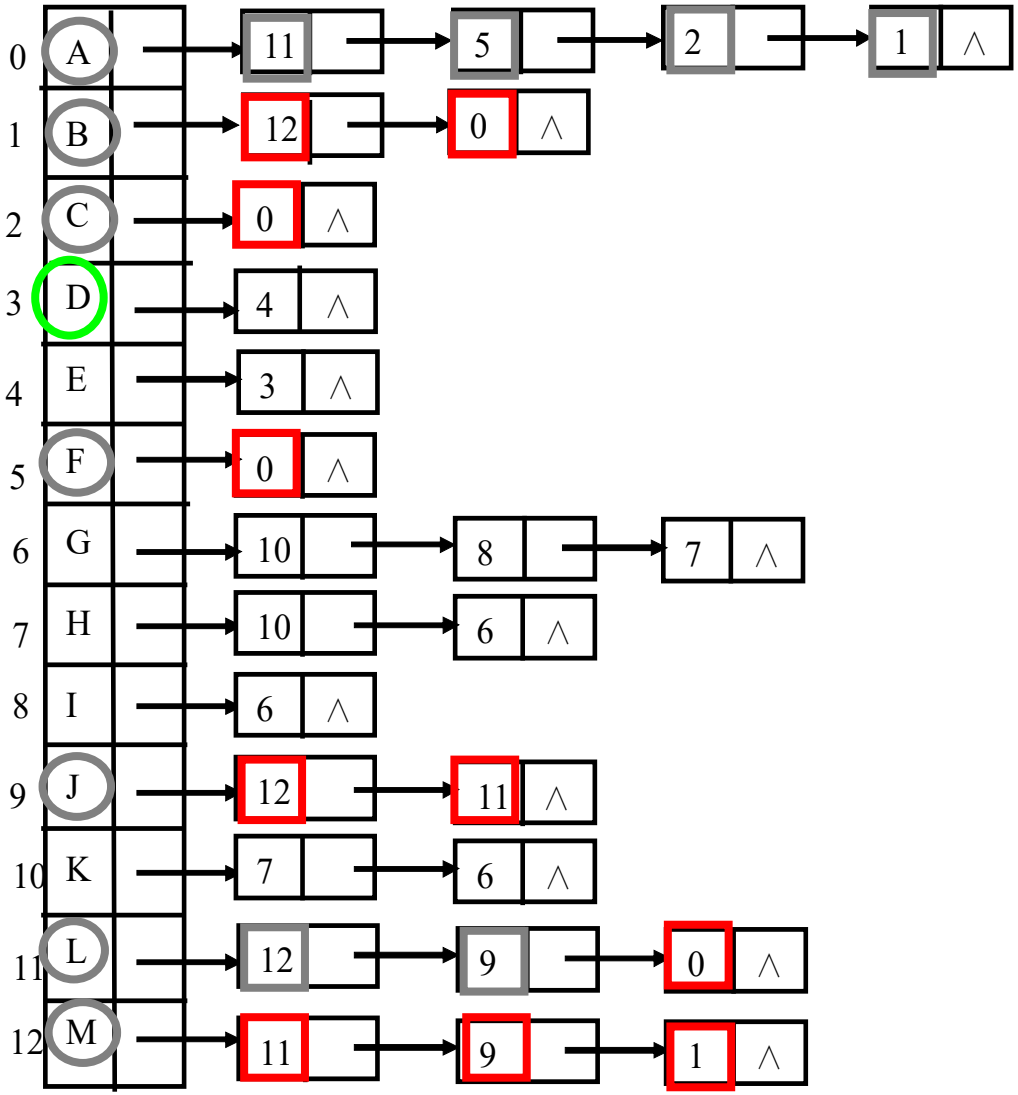
从顶点2出发：2,1,3,4,0,



A L F C B M J;
D E;
G K I H;

队列

~~A~~ ~~L~~ ~~F~~ ~~C~~ ~~B~~ ~~M~~ ~~J~~



7.4 图的连通性

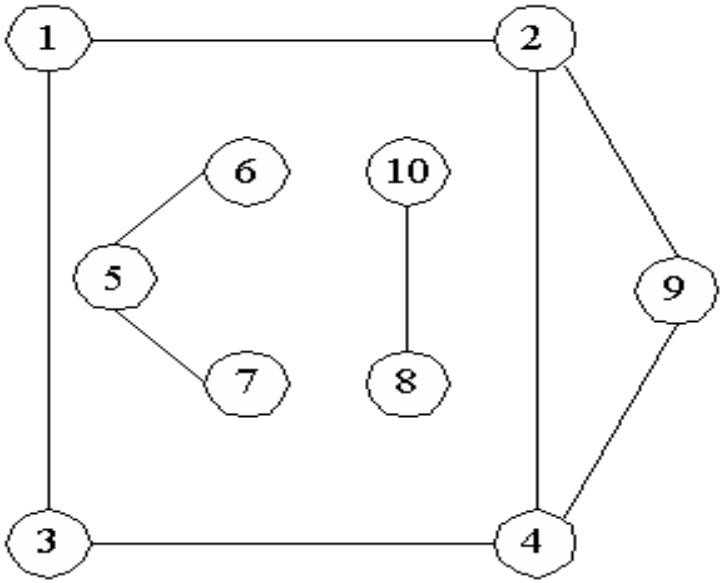
- 利用图的遍历运算求解图的连通性问题
  - 无向图是否连通、有几个连通分量，求解无向图的所有连通分量
    - 深度优先生成树、生成森林
    - 广度优先生成树、生成森林
  - 有向图是否是强连通、求解其强连通分量
- 求无向网的最小代价生成树

7.4 图的连通性问题

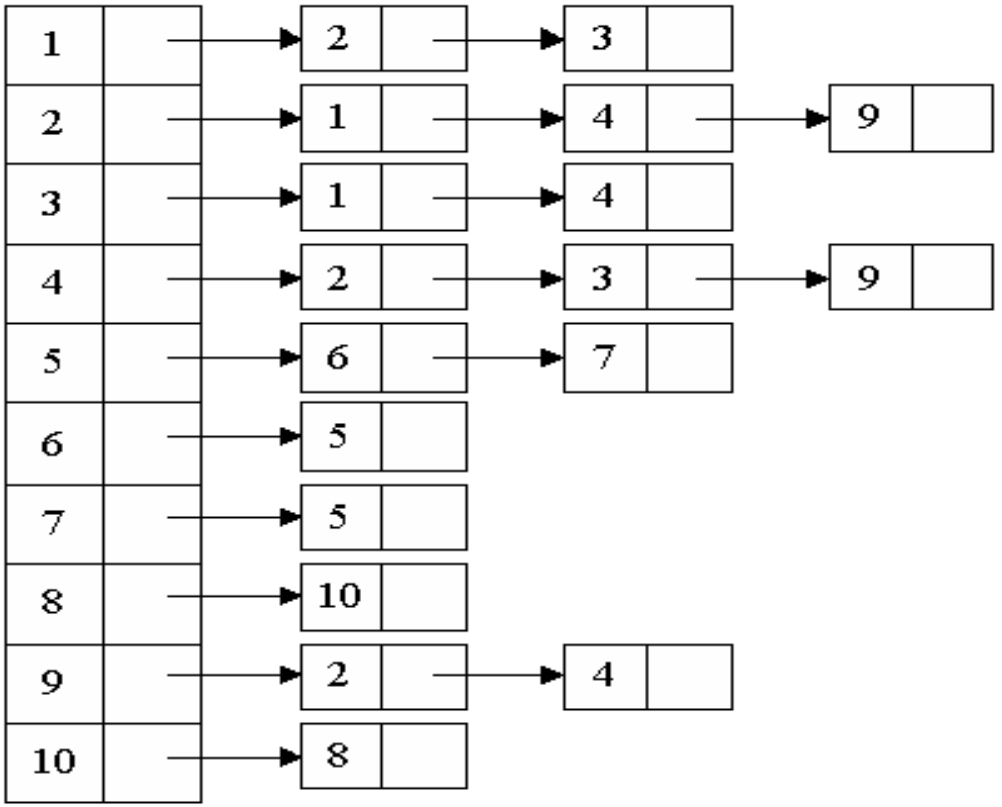
1. 无向图的连通分量

图遍历时，对于连通图，无论是广度优先搜索还是深度优先搜索，仅需要调用一次搜索过程，即从任一个顶点出发，便可以遍历图中的各个顶点。对于非连通图，则需要多次调用搜索过程，而每次调用得到的顶点访问序列恰为各连通分量中的顶点集。

```
j=0;
for(v=0; v < G.vernum; ++v)
    if(!visited[v]){
        DFS(G, v);
        j++;
    }
```

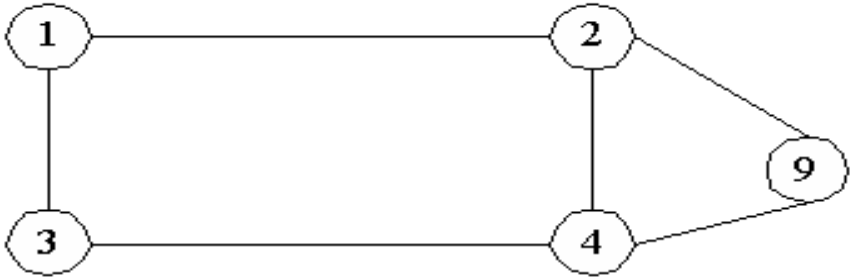


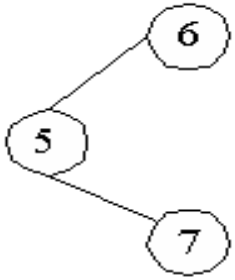
(a) 无向图G5



(b) G5的邻接表

1，2，4，3，9
5，6，7
8，10





(c) 无向图G5的三个连通分量

2. 无向图的生成树

一个连通图的生成树是一个极小连通子图,它含有图中全部顶点,但只有构成一棵树的(n-1)条边。

e<n-1 → 非连通图

e>n-1 → 有回路

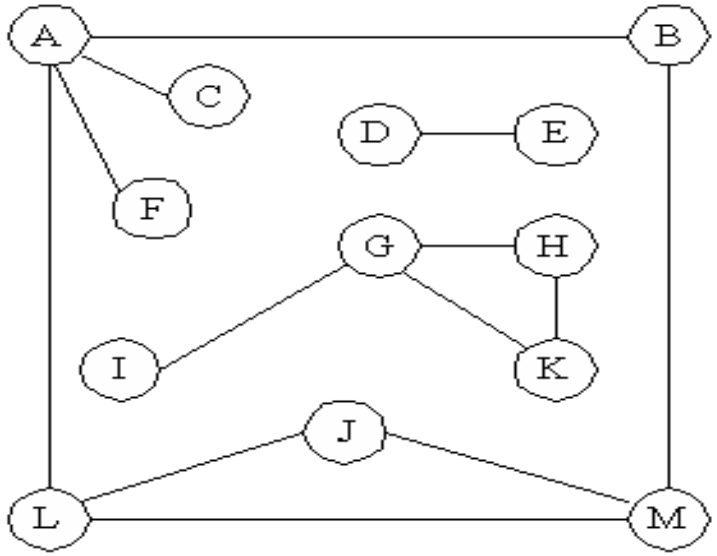
e=n-1 → 不一定都是图的生成树

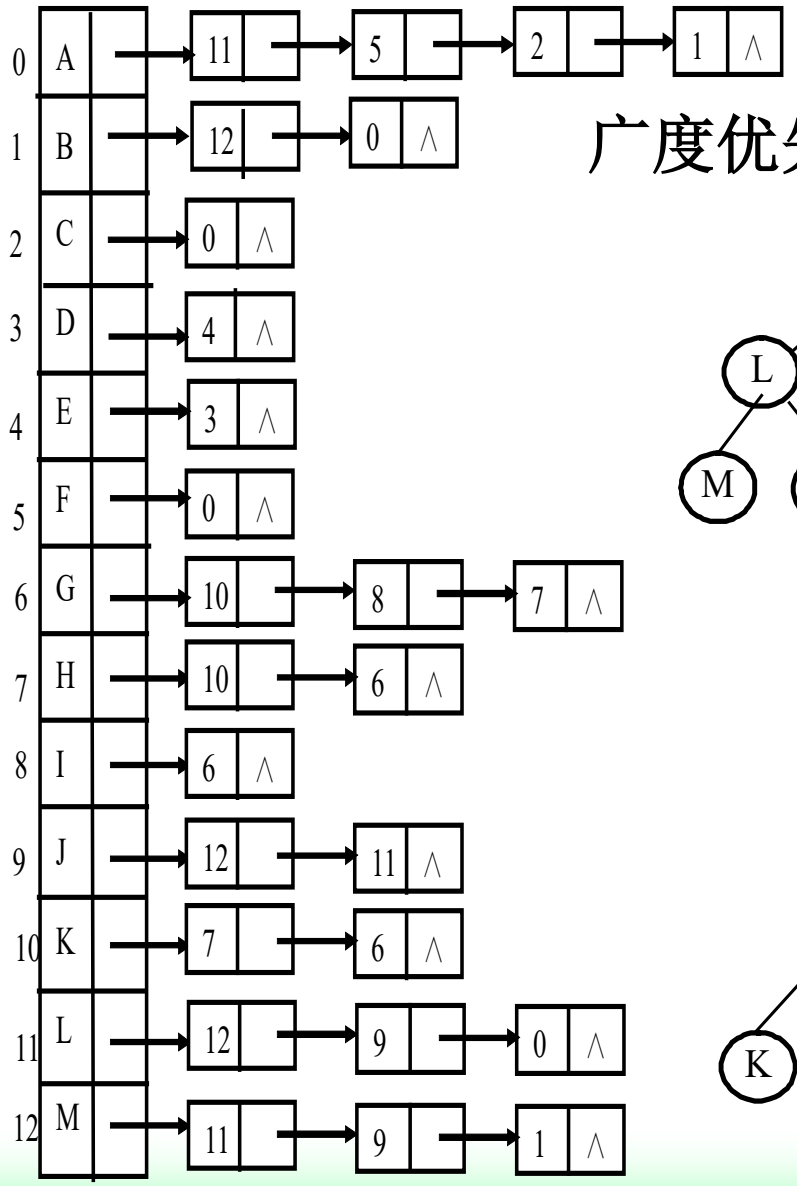
设G=(V,E)为连通图,则从图中任一顶点出发遍历图时,必定将E(G)分成两个集合T和B,其中T是遍历图过程中走过的边的集合,B是剩余的边的集合：$T\cap B=\Phi, T\cup B=E(G)$。显然,G'=(V,T)是G的极小连通子图,即G'是G的一棵生成树。

由深度优先遍历得到的生成树称为深度优先生成树；由广度优先遍历得到的生成树称为广度优先生成树。这样的生成树是由遍历时访问过的n个顶点和遍历时经历的n-1条边组成。

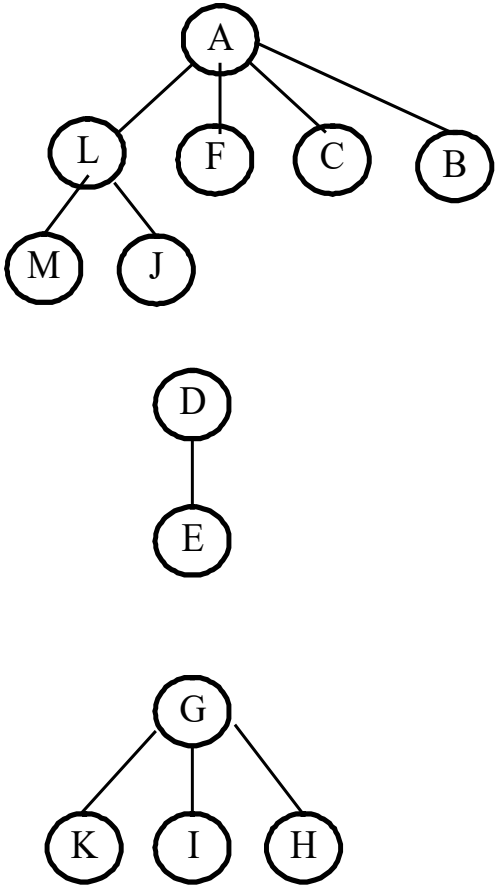
对于非连通图,每个连通分量中的顶点集和遍历时走过的边一起构成一棵生成树,各个连通分量的生成树组成非连通图的生成森林。

问题：如何建立无向图的深度优先生成森林或广度优先生成森林？

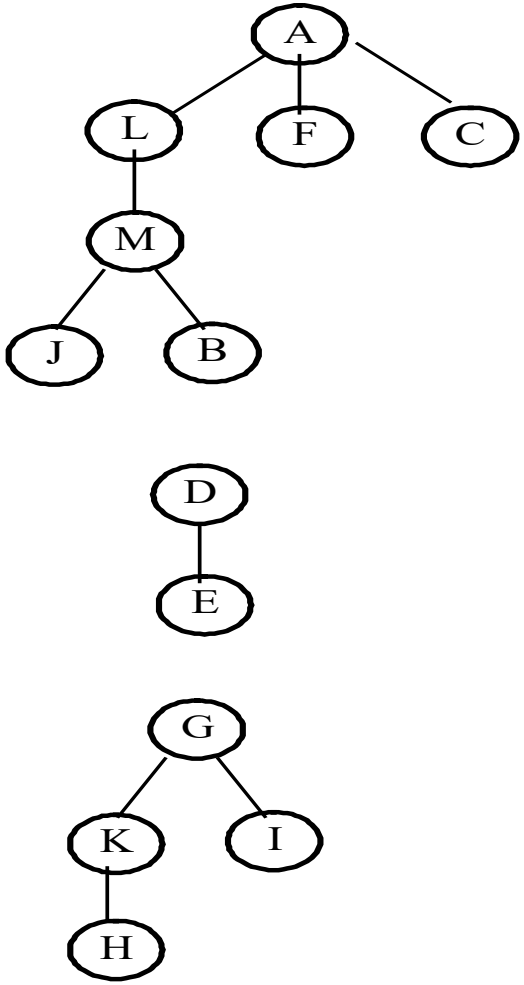




广度优先生成森林



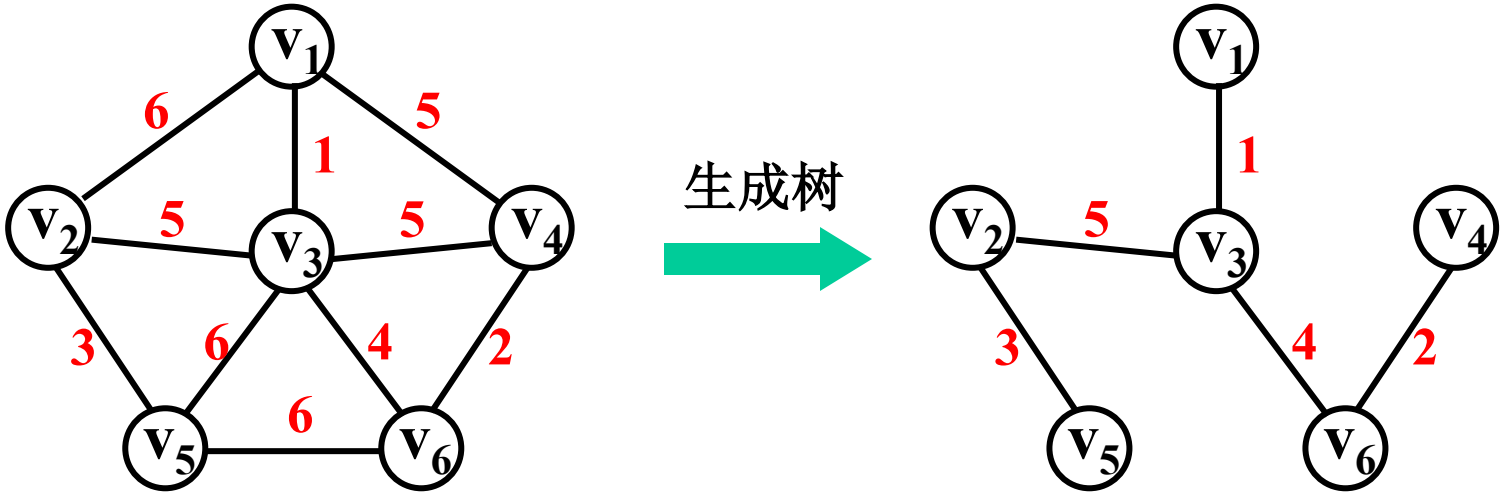
深度优先生成森林



3 最小生成树

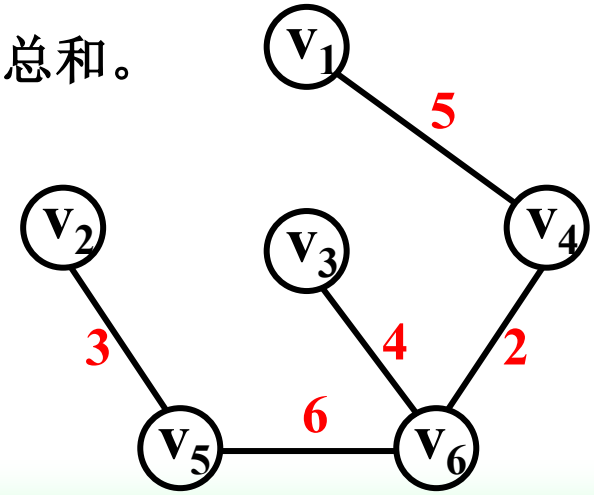
一个无向图可以对应多个生成树。

一个带权无向图(无向网)同样可以对应多个生成树。



一棵生成树的代价定义为树上各边的权之总和。

代价最小的生成树称为最小代价生成树（Minimum Cost Spanning Tree），简称为最小生成树(MST)。



Prim 算法　　1957年由Prim提出

思想：

N = (V , E) 是具有 n 个顶点的连通网，设 U 是最小生成树中顶点的集合，设 TE 是最小生成树中边的集合；

初始，$U = \{ u_1 \}$，$TE = \{ \}$，

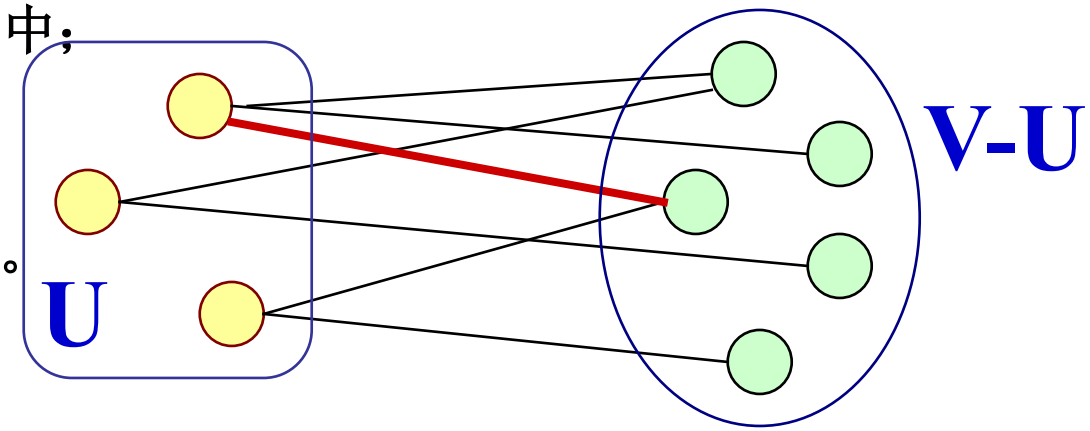
重复执行：

在所有 $u \in U$，$v \in V-U$ 的边 (u , v) 中寻找代价最小的边(u' , v')，并纳入集合 TE 中；

同时将 v' 纳入集合 U 中；

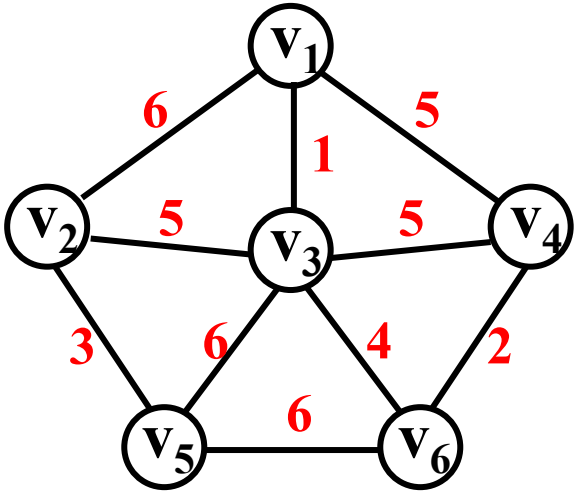
直至 U = V 为止。

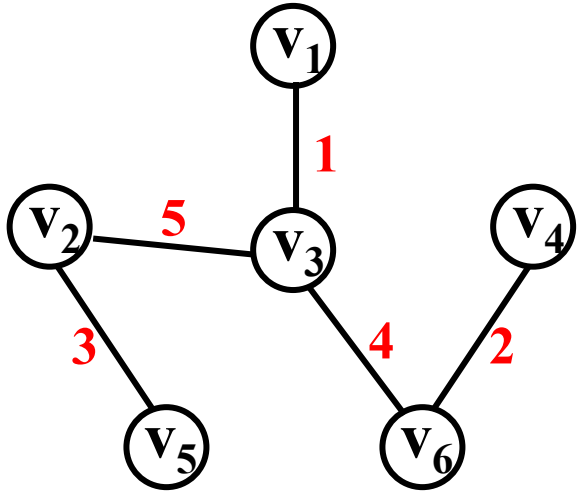
集合 TE 中必有 n−1 条边。



从一个平凡图开始，普利姆算法逐步增加U中的顶点， 可称为“加点法”。

例，





初始：$U=\{v_1\}$，$V-U=\{v_2,v_3,v_4,v_5,v_6\}$　　$TE=\{\}$

$U=\{v_1,v_3\}$，$V-U=\{v_2,v_4,v_5,v_6\}$　　$<v_1,v_3>$

$U=\{v_1,v_3,v_6\}$，$V-U=\{v_2,v_4,v_5\}$　　$<v_3,v_6>$

$U=\{v_1,v_3,v_4,v_6\}$，$V-U=\{v_2,v_5\}$　　$<v_6,v_4>$

$U=\{v_1,v_2,v_3,v_4,v_6\}$，$V-U=\{v_5\}$　　$<v_3,v_2>$

$U=\{v_1,v_2,v_3,v_4,v_5,v_6\}$，$V-U=\{\}$　　$<v_2,v_5>$

重点：边一定存在于 U 与 V−U 之间。

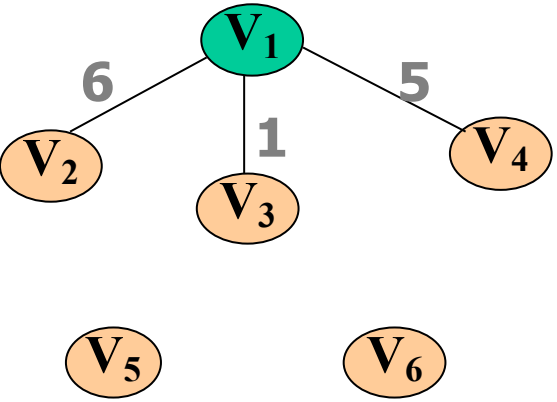
- **Prim算法的实现**
  - 顶点集合如何表示？
  - 最小边如何选择？
  - 一个顶点加入U集合如何表示？

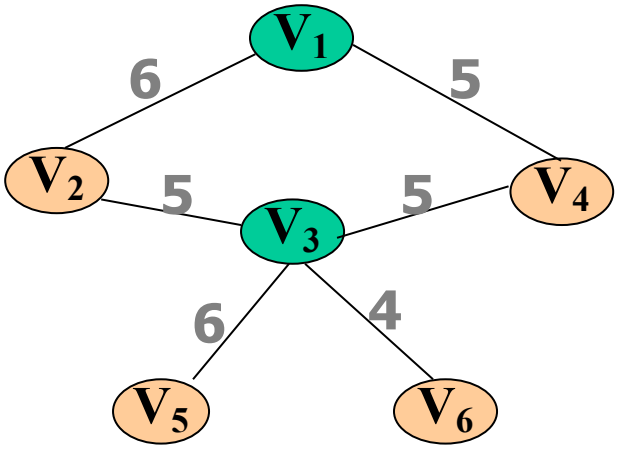
```
struct {//记录从U到V-U具有最小代价的边。
   VertexType  adjvex; //顶点名称
    int                lowcost;
} closedge[MAX_VERTEX_NUM];
```

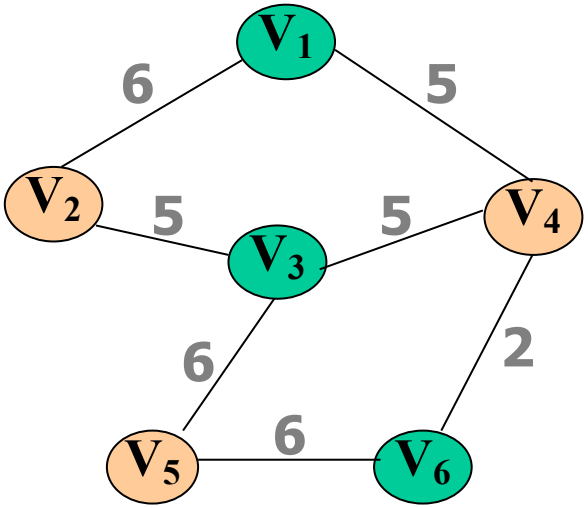
closedge[i].lowcost = Min({cost(u, v_i) | u∈U})

closedge[i].adjvex=k　顶点i与顶点k邻接,顶点k已经在U集合中

closedge[i].lowcost=0　顶点i加入U集合时







顶点i / closedge	2	3	4	5	6	U	V-U	k
adjvex lowcost	v1 6	v1 1	v1 5			{v1}	{v2,v3,v4,v5,v6}	3

closedge[2].adjvex=1 .lowcost=6

closedge[3].adjvex=1 .lowcost=1

closedge[4].adjvex=1 .lowcost=5

adjvex lowcost	v3 5	0	v1 5	v3 6	v3 4	{v1,v3}	{v2,v4,v5,v6}	6
adjvex lowcost	v3 5	0	v6 2	v3 6	0	{v1,v3,v6}	{v2,v4,v5 }	4

```
struct {
    int adjvex;
    double lowcost;
}closedge[MAX_VERTEX_NUM];
```

```
void MiniSpanTree_PRIM( MGrap
{   k = LocateVex(G, u);
    for(j=0; j<G.vexnum; ++j)
        if(j!=k)   closedge[j] = { u, G.arcs[k][j].adj };
    closedge[k].lowcost = 0;
    for(i=1; i<G.vexnum; ++i){
        k = minimun(closedge);
        printf(colsedge[k].adjvex, G.vexs[k]);
        closedge[k].lowcost = 0;
        for(j=0; j<G.vexnum; ++j)
            if(G.arcs[k][j].adj < closedge[j].lowcost)
                co
    }
}
```

- 新顶点并入U，重新计算代价最小的边

Prim()算法中有两重for循环, 所以时间复杂度为$O(n^2)$。

与网中的边数无关，适用于求边稠密的网的最小生成树。

Kruskal 算法

● Kruskal于1956年提出

思想：

考虑问题的出发点：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。

N = (V , E) 是 n 顶点的连通网，设 E 是连通网中边的集合；

构造最小生成树 N' = (V , TE)，TE 是最小生成树中边的集合，初始 TE = {} ；

重复执行：

选取 E 中权值最小的边 (u , v) ，

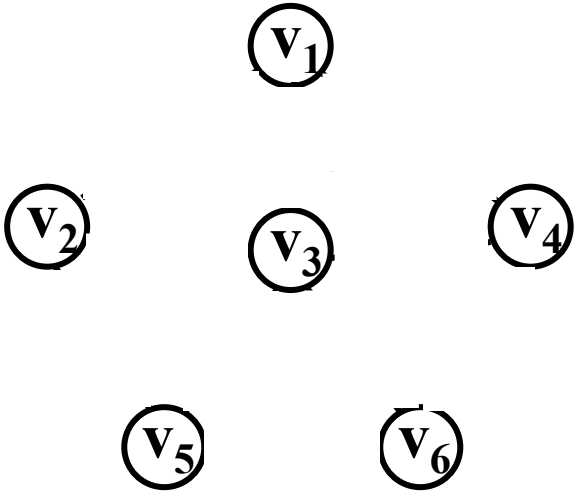
判断边 (u , v) 与 TE 中的边是否构成回路 ？

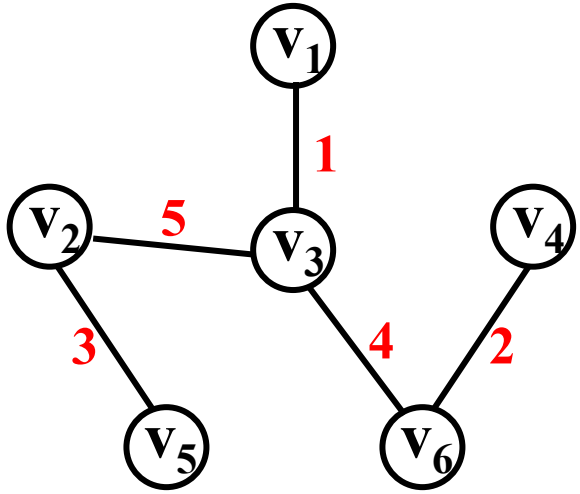
否，将边 (u , v) 纳入 TE 中，并从 E 中删除边 (u , v) ；

u 和 v 一定不在同一个连通分量中

直至 E 为空 ；

例，





当前权值最小边 (v_5 , v_6)

初始 TE = { }

< v_1 , v_3 >

< v_4 , v_6 >

< v_2 , v_5 >

< v_3 , v_6 >

< v_2 , v_3 >

从一个零图开始，克鲁斯卡尔算法逐步增加生成树的边，与普里姆算法相比，可称为“加边法”。

为了简便, 在实现克鲁斯卡尔算法Kruskal()时, 参数E存放图G中的所有边, 假设它们是按权值从小到大的顺序排列的。n为图G的顶点个数, e为图G的边数。

```
typedef struct {
    int u;     /*边的起始顶点*/
    int v;     /*边的终止顶点*/
    int w;     /*边的权值*/
} Edge;
```

```
void Kruskal(Edge E[],  int n)
{    int i,j,k; int vset[MAXV];
     for (i=0;i<n;i++)  vset[i]=i;
     k=1; j=0;
     while (k<n) { /*生成的边数小于n时循环*/
          if (vset[E[j].u] != vset[E[j].v]) {
               printf("(%d,%d):%d\n",E[j].u,E[j].v,E[j].w);
               k++; sv=vset[E[j].v];su=vset[E[j].u];
               for (i=0;i<n;i++)      /*两个集合统一编号*/
                    if (vset[i]==sv)
                         vset[i]=su;
          }
          j++;     /*扫描下一条边*/
     }
}
```

完整的克鲁斯卡尔算法应包括对边按权值递增排序，上述算法假设边已排序的情况下，时间复杂度为$O(n^2)$。

如果给定的带权连通无向图G有e条边, n个顶点，采用堆排序(在第10章中介绍)对边按权值递增排序，并且用6.5节的等价类判断连通性，则克鲁斯卡尔算法的时间复杂度降为**O(eloge)**。由于它与n无关，只与e有关，所以说克鲁斯卡尔算法适合于求边稀疏的网的最小生成树。

作业：

7.5　　7.7

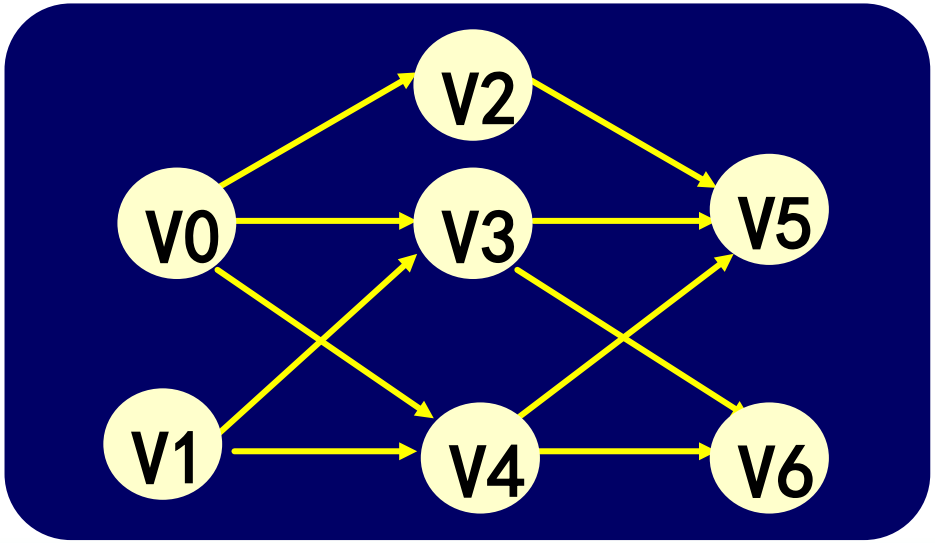
7.5 有向无环图DAG

- AOV网、AOE网
- 拓扑排序
- 关键路径

❑ AOV网(Activity On Vertex net)

☞用顶点表示活动，边表示活动的顺序关系的有向图称为AOV网.

例：某工程可分为7个子工程，若用顶点表示子工程（也称活动）， 用弧表示子工程间的顺序关系，工程的施工流程可用如右的AOV网表示。



➢工程能否顺序进行，即工程流程是否“合理”？

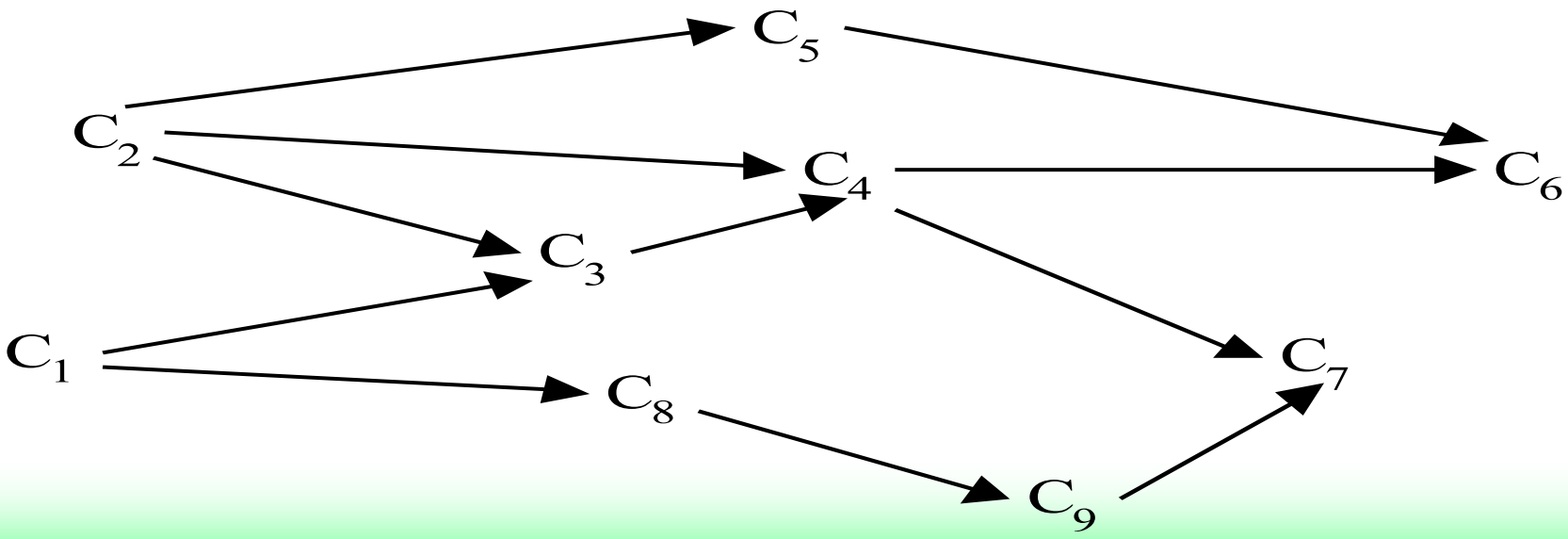
7.5 有向无环图及其应用

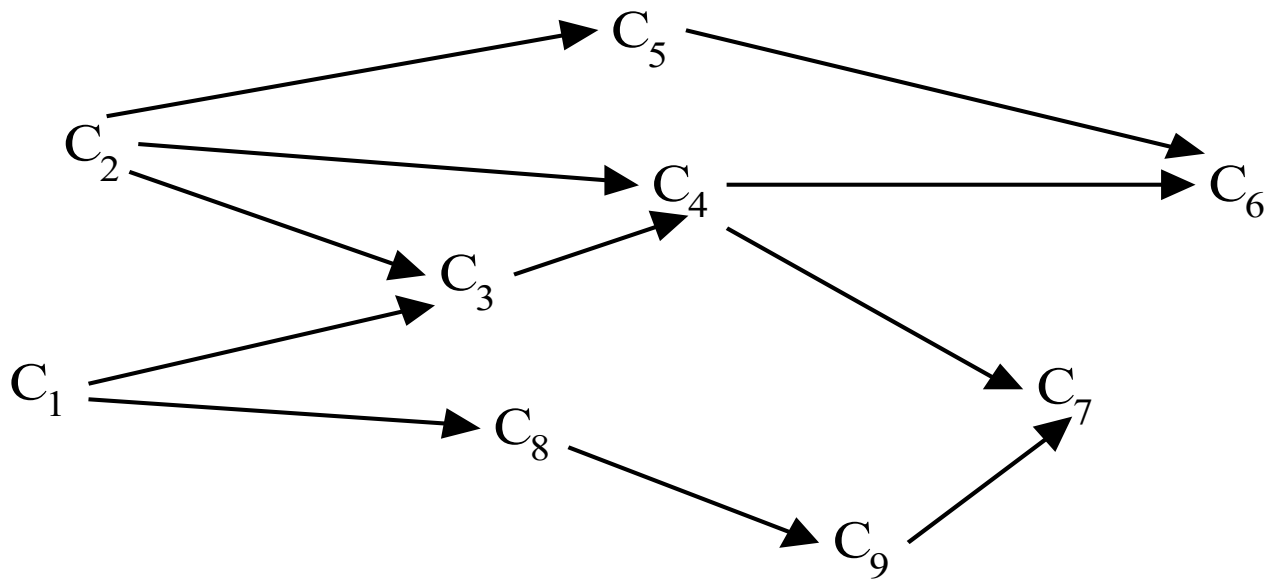
1. 拓扑排序（Topological Sort）

设G=(V,E)是一个具有n个顶点的有向图,V中顶点序列$v_1,v_2,\ldots,v_n$称为一个拓扑(有序)序列,当且仅当该顶点序列满足下列条件：若$<v_i,v_j>$是图中的弧(即从顶点v_i到v_j有一条路径),则在序列中顶点v_i必须排在顶点v_j之前。

在一个有向图中找一个拓扑序列的过程称为拓扑排序。

课程编号	课程名称	先修课程
C_1	高等数学	无
C_2	程序设计基础	无
C_3	离散数学	C_1，C_2
C_4	数据结构	C_2，C_3
C_5	算法语言	C_2
C_6	编译技术	C_4，C_5
C_7	操作系统	C_4，C_9
C_8	普通物理	C_1
C_9	计算机原理	C_8





拓扑序列： $C_1, C_2, C_3, C_4, C_5, C_8, C_9, C_7, C_6$ 。

拓扑序列： $C_1, C_2, C_3, C_8, C_4, C_5, C_9, C_7, C_6$ 。

用顶点表示活动，用弧表示活动间的优先关系的有向图，称为顶点表示活动的网(Activity On Vertex Network)，简称为**AOV-网**。

如何进行拓扑排序？

方法一：（从图中顶点的入度考虑）

① 从有向图中选择一个没有前驱(即入度为0)的顶点并且输出它。

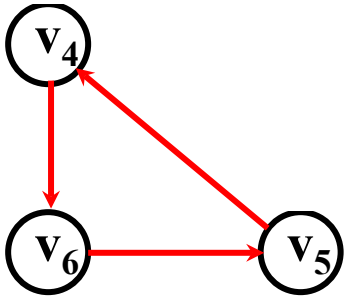
② 从网中删去该顶点和所有以它为尾的弧；

③ 重复上述两步,直到图全部顶点输出；或当前图中不再存在没有前驱的顶点。

例，

拓扑排序　v_1 v_6 v_4 v_3 v_2 v_5

例，



拓扑排序　v_1　v_3　v_2

存在环

方法二：（从图中顶点的出度考虑，得到逆拓扑序列）

① 从有向图中选择一个出度为0的顶点并且输出它。

② 从网中删去该顶点和所有以它为头的弧；

③ 重复上述两步,直到图全部顶点输出；或当前图中不再存在出度为0的顶点。

方法三：当有向图中无环时，利用深度优先遍历进行拓扑排序

从某点出发进行DFS遍历时，最先退出DFS函数的顶点即出度为0的顶点，是拓扑序列中最后一个顶点。按退出DFS函数的先后记录下来的顶点序列即为逆拓扑序列。

```
Status  TopologicalSort( ALGraph G)
{     int St[MAXV], top=-1;  /*栈St的指针为top*/
      FindInDegree(G, indegree); //indegree顶点入度
      for (i=0; i<G.vexnum; i++)
          if (! indegree[i])  {   top++; St[top]=i; }
      count = 0;
      while (top>-1) { /*栈不为空时循环*/
           i=St[top];  top--;
           printf("%d ",G.vertices[i].data);  ++count;
           for( p=G.vertices[i].firstarc; p; p=p->nextarc){
                k = p->adjvex;
                if ( !(--indegree[k])) {  top++;  St[top]=k;  }
           }
      }
      if(count<G.vexnum)  return ERROR;   else return OK;
}
```

```
void FindInDegree( ALGraph G,   int *indegree)
{  int i;    ArcNode *p;
   for(i=0; i<G.vexnum; i++)     indegree[i] = 0;
   for(i=0;  i<G.vexnum;  i++)  //扫描邻接表，计算各顶点的入度
     for(p=G.vertices[i].firstarc;  p;  p=p->nextarc)
           Indegree[p->adjvex]++;
   }
}
```

- 分析此拓扑排序算法可知，如果AOV网络有n个顶点，e条边，求个顶点的入度的时间复杂度为O(e)，建立零入度顶点栈所需要的时间是O(n)。在拓扑排序的过程中在，有向图有n个顶点，每个顶点进一次栈，出一次栈，共输出n次。顶点入度减一的运算共执行了e次。所以总的时间复杂度为O(n+e)。

2. 关键路径

有向图在工程计划和经营管理中有着广泛的应用。 通常用有向图来表示工程计划时有两种方法：

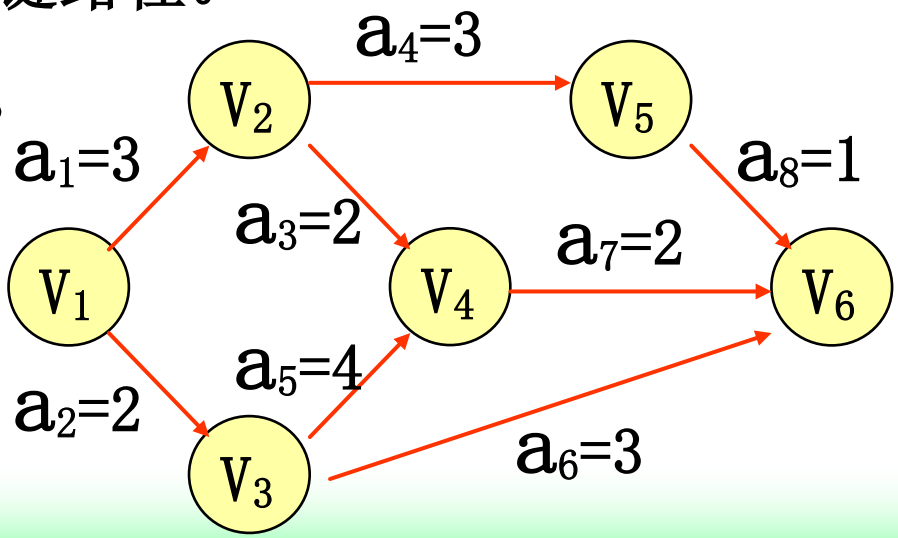
- 用顶点表示活动， 用有向弧表示活动间的优先关系， 即上节所讨论的AOV-网。
- 用顶点表示事件， 用弧表示活动， 弧的权值表示活动所需要的时间。带权的有向无环图叫做边表示活动的网（Activity On Edge Network）, 简称AOE-网。
- 事件：表示在它之前的活动已经完成，在它之后的活动可以开始。

- **AOE-网有待解决的问题：**

  ①哪些活动是影响工程进度的关键活动？

  ②至少需要多长时间能完成整个工程？

- 源点：在AOE网中存在唯一的、入度为零的顶点；
- 汇点：存在唯一的、出度为零的顶点。
- 关键路径：从源点到汇点的最长路径的长度即为完成整个工程任务所需的时间，该路径叫做关键路径。
- 关键活动：关键路径上的活动。



- 关键活动的改进可以改变工程进度；
- 关键活动速度提高有限度；只有在不改变关键路径的情况下，提高关键活动的速度才有效；
- 若有多条关键路径，则要改变进度必须同时提高各条关键路径上的速度。

定义几个与计算关键活动有关的量：

- 事件V_j 的最早发生时间*ve(j)*
  是从源点V_0到顶点V_j 的最长路径长度。
- 事件V_j 的最迟发生时间*vl(j)*
  是在保证汇点V_{n-1} 在*ve(n-1)*时刻完成的前提下，事件V_j 的允许的最迟开始时间。
- 活动a_i 的最早开始时间*e(i)*
  设活动a_i 在弧$<V_j, V_k>$上，则*e(i)*是从源点V_0到顶点V_j 的最长路径长度。因此，*e(i) = ve(j)*。
- 活动a_i 的最迟开始时间*l(i)*
  *l(i)*是在不会引起时间延误的前提下，该活动允许的最迟开始时间。 *l(i)= vl(k) - dur(<j, k>)*。其中，*dur(<j, k>)*是完成a_i所需的时间。

- 时间余量 $l(i) - e(i)$

  表示活动a_i 的最早开始时间和最迟开始时间的时间余量。

  $l(i) == e(i)$表示活动a_i 是没有时间余量的关键活动。

- 为找出关键活动, 需要求各个活动的 $e(i)$ 与 $l(i)$，以判别是否 $l(i) == e(i)$。
- 为求得$e(i)$与 $l(i)$，需要先求得从源点V_0到各个顶点V_j的 $ve(j)$和 $vl(j)$。

- 从$ve(0) = 0$开始，向前递推

$$ve(j) = \max_i \{ ve(i) + dur(<V_i, V_j>) \},$$

$$<V_i, V_j> \in T, \quad j = 1, 2, \ldots, n\text{-}1$$

其中T是所有以V_j为头的弧的集合。

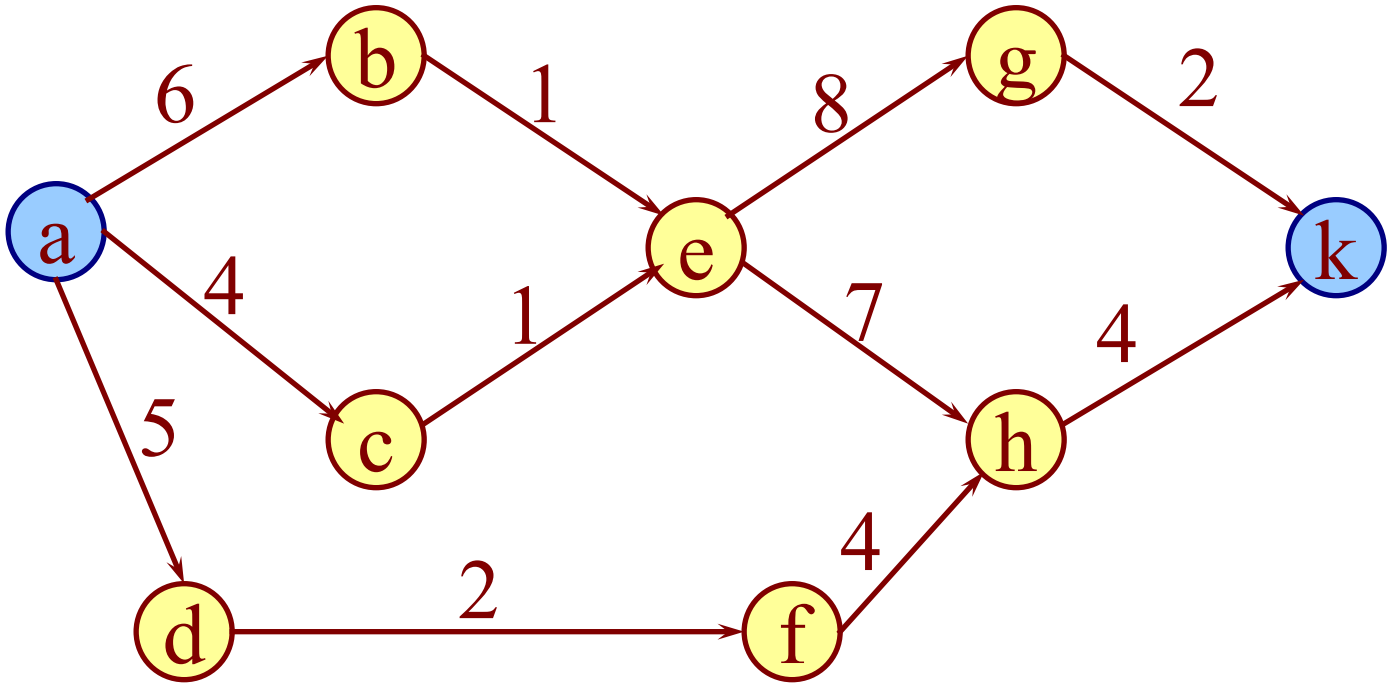
- 从$vl(n\text{-}1) = ve(n\text{-}1)$开始，反向递推

$$vl(i) = \min_j \{ vl(j) - dur(<V_i, V_j>) \},$$

$$<V_i, V_j> \in S, i = n\text{-}2, n\text{-}3, \ldots, 0$$

其中S是所有以V_i为尾的弧的集合。

- $e(i) = ve(j)$, $l(i) = vl(k) - dur(<j, k>)$



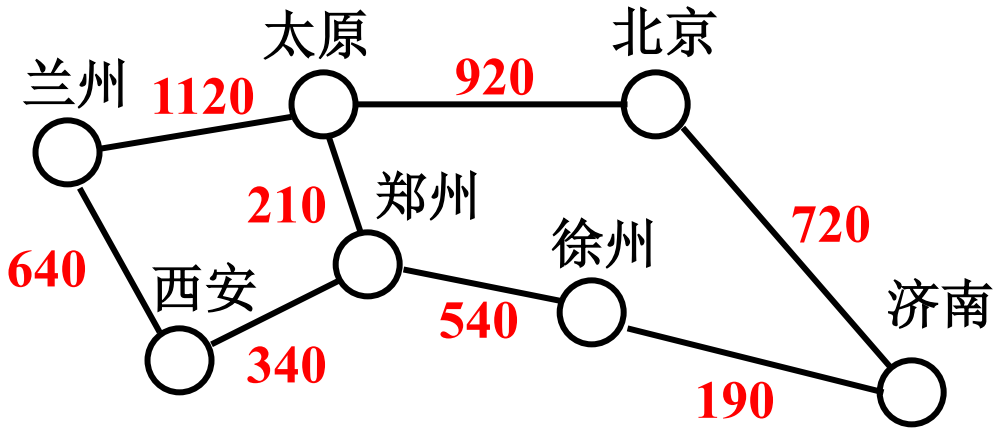
	a	b	c	d	e	f	g	h	k
ve	0	6	4	5	7	7	15	14	18
vl	0	6	6	8	7	10	16	14	18

拓扑有序序列：a - d - f - c - b - e - h - g - k

	a	b	c	d	e	f	g	h	k
ve	0	6	4	5	7	7	15	14	18
vl	0	6	6	8	7	10	16	14	18

	ab	ac	ad	be	ce	df	eg	eh	fh	gk	hk
权	6	4	5	1	1	2	8	7	4	2	4
e	0	0	0	6	4	5	7	7	7	15	14
l	0	2	3	6	6	8	8	7	10	16	14
	✓			✓				✓			✓

7.6 最短路径



旅客希望停靠站越少越好，则应选择

济南——北京——太原——兰州

旅客考虑的是旅程越短越好，

济南——徐州——郑州——西安——兰州

带权图的最短路径计算问题

通常在实际中，航运、铁路、船行都具有有向性，故我们以带权有向图为例介绍最短路径算法。

带权无向图的最短路径算法也通用。

从单个源点到其余各顶点的最短路径算法。

从一个顶点到其余各顶点的最短路径

问题：给定一个带权有向图G与源点v,求从v到G中其他顶点的最短路径, 并限定各边上的权值大于或等于0。

迪杰斯特拉(Dijkstra)算法思想：

贪心算法(局部最优)，按路径长度递增的次序产生最短路径。

贪心算法：利用局部最优来计算全局最优。

利用已得到的顶点的最短路径来计算其它顶点的最短路径。

- 路径长度最短的**最短路径**的特点:

  在这条路径上，必定只含一条弧，并且这条弧的权值最小。

- 下一条路径长度次短的**最短路径**的特点:

  它只可能有两种情况:或者是直接从源点到该点(只含一条弧)； 或者是，从源点经过顶点v_1，再到达该顶点(由两条弧组成)。

- 其余最短路径的特点：

  它或者是直接从源点到该点(只含一条弧)； 或者是，从源点经过已求得最短路径的顶点，再到达该顶点。

- 采用迪杰斯特拉(Dijkstra)算法求解

  Dijkstra提出按路径长度的递增次序，逐步产生最短路径的算法。

- 引入一个辅助数组D。它的每一个分量$D[i]$表示当前找到的从源点v_0到终点v_i的最短路径的长度。初始状态：
  - 若从源点v_0到顶点v_i有边，则$D[i]$为该边上的权值；
  - 若从源点v_0到顶点v_i没有边，则$D[i]$为$+\infty$。
- 一般情况下，假设S是已求得的最短路径的终点的集合，则可证明：下一条最短路径必然是从v_0出发，中间只经过S中的顶点便可到达的那些顶点v_x ($v_x \in V\text{-}S$)的路径中的一条。
- 每次求得一条最短路径之后，其终点v_k加入集合S，然后对所有的$v_i \in V\text{-}S$，修改其$D[i]$值。

一般情况，假设 S 为已求得最短路径的终点的集合，则有：下一条最短路径(设终点为 x) 或者是弧 (v_0 , x) ，或者是 v_0 出发中间只经过 S 中的顶点而最后到达顶点 x 的路径。

反证法:

假设下一条最短路径上有一个顶点不在 S 中，不妨设 v’；

则必存在一条终点为 v’ 的最短路径，其长度比该路径短；

可这是不可能的，因为我们是按照路径长度递增的次序来依次产生最短路径，即长度比该路径短的所有路径都已产生；

矛盾。

- **Dijkstra算法可描述如下：**

①初始化： $S \leftarrow \{ v0 \}$;

$D[j] \leftarrow arcs[0][j]$, $j = 1, 2, \ldots, n\text{-}1$;

②求出最短路径的长度：

$D[k] \leftarrow \min\{ D[i] \}$, $i \in V\text{-}S$;

$S \leftarrow S \cup \{ k \}$;

③修改：

$D[i] \leftarrow \min\{ D[i], D[k]+arcs[k][i] \}$,

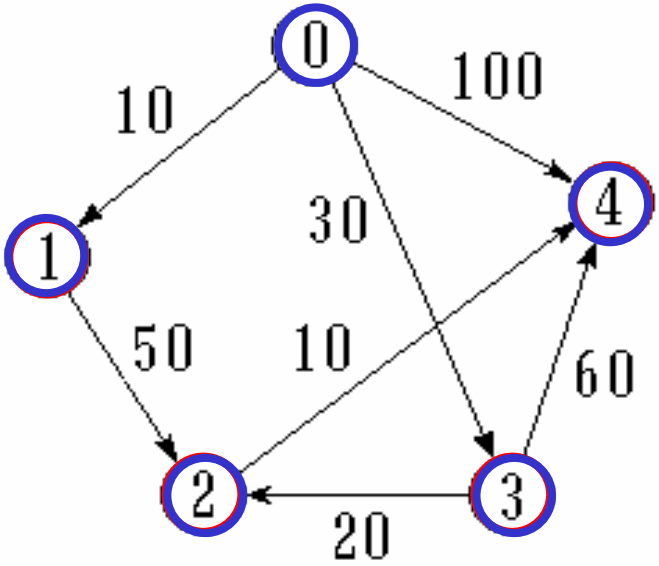
对于每一个 $i \in V\text{-}S$;

④判断：若$S = V$, 则算法结束，否则转② 。

D[n]: 从源点到其余顶点的最短路径长度；

F[n]: 已找到最短路径的顶点，属于S集or属于V-S集；

P[n]: 记录已找到的路径。P[i]记录路径上v_i的前驱。



如何从表中读取源点0 到终点v 的最短路径？举顶点4为例

path[4] = 2 →path[2] = 3 →

path[3] = 0，反过来排列，得到路径0, 3, 2, 4，这就是源点0到终点4的最短路径。

选取终点	顶点 1			顶点 2			顶点 3			顶点 4		
	F[1]	*D*[1]	*P*[1]	*F*[2]	*D*[2]	*P*[2]	*F*[3]	*D*[3]	*P*[3]	*F*[4]	*D*[4]	*P*[4]
0	0	10	0	0	∞	-1	0	30	0	0	100	0
1	1	10	0	0	60	1	0	30	0	0	100	0
3	1	10	0	0	50	3	1	30	0	0	90	3
2	1	10	0	1	50	3	1	30	0	0	60	2
4	1	10	0	1	50	3	1	30	0	1	60	2

```
void Dijkstra(MGraph G)
{   int D[MAXV], P[MAXV], F[MAXV]; F[0] = 1;
    for (i=1; i<G.vernum; i++) {
       D[i] = G.arcs[0][i]; F[i]=0;
       if (D[i]<INT_MAX)  P[i]=0;
       else  P[i]=-1;
    }
    for(i=1; i<G.vernum; i++){
       …..
    }
    Dispath(D, P, F, G.vernum, 0);
}
```

```
for(i=1; i<G.vernum; i++){
    min = INT_MAX;
    for(j=1; j<G.vernum; j++)
        if( !F[j])
            if( D[j] < min){ w = j; min = D[j]; }
    F[w] = 1;
    for(j=1; j<G.vernum; j++){
       if(!F[j]&&((D[w]+G.arcs[w][j]) < D[j])){
            D[j] = D[w]+G.arcs[w][j];
            P[j] = w;
   }
}
```

```
void Ppath(int *path,int i,int v0) /*前向递归查找路径上的顶点*/
{
    k=path[i];
    if (k==v0)  return;    /*找到了起点则返回*/
    Ppath(path, k, v0);     /*找k顶点的前一个顶点*/
    printf("%d,",k);      /*输出k顶点*/
}
```

```
void Dispath(int *dist,int *path,int *final,int n,int v0)
{
    for (i=0;i<n;i++){
        if (final[i]==1) {
            printf(“从%d到%d的最短路径长度为:
                    %d\t路径为:",v0,i,dist[i]);
            printf("%d,",v0);          /*输出路径上的起点*/
            Ppath(path,i,v0);          /*输出路径上的中间点*/
            printf("%d\n",i);          /*输出路径上的终点*/
        }
        else  printf("从%d到%d不存在路径\n",v0,i);
    }
}
```

2. 每对顶点之间的最短路径

◆ 问题的提法：已知一个各边权值均大于0的带权有向图，对每一对顶点 $v_i \neq v_j$，要求求出v_i 与v_j之间的最短路径和最短路径长度。

◆ 弗洛伊德(Floyd)算法的基本思想：

定义一个n阶方阵序列：

$$D^{(-1)},\ D^{(0)},\ \ldots,\ D^{(n-1)}.$$

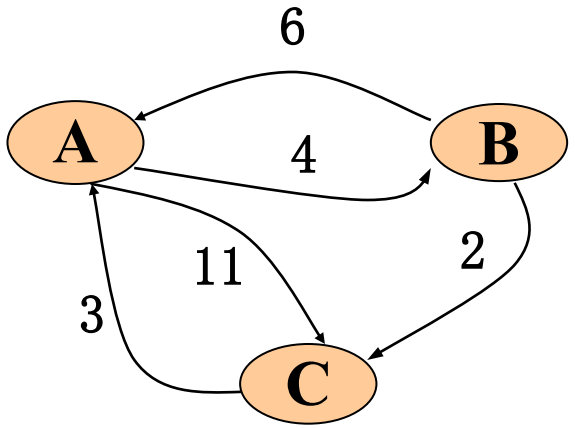
其中 $D^{(-1)}[i][j] = arcs[i][j]$；

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

$$k = 0,1,\ldots, n\text{-}1$$

$D^{(0)}[i][j]$是从顶点v_i到v_j，中间顶点是v_0的最短路径的长度，$D^{(k)}[i][j]$是从顶点v_i到v_j，中间顶点的序号不大于k的最短路径的长度，$D^{(n-1)}[i][j]$是从顶点v_i到v_j的最短路径长度。

Floyd算法求最短路径



	A	B	C
A	∞	4	11
B	6	∞	2
C	3	∞	∞

(a) 路径长度

加入顶点A

	A	B	C
A	∞	4	11
B	6	∞	2
C	3	7	∞

(a) 路径长度

加入顶点B

	A	B	C
A	∞	4	6
B	6	∞	2
C	3	7	∞

(a) 路径长度

	A	B	C
A		AB	AC
B	BA		BC
C	CA		

(b) 路径

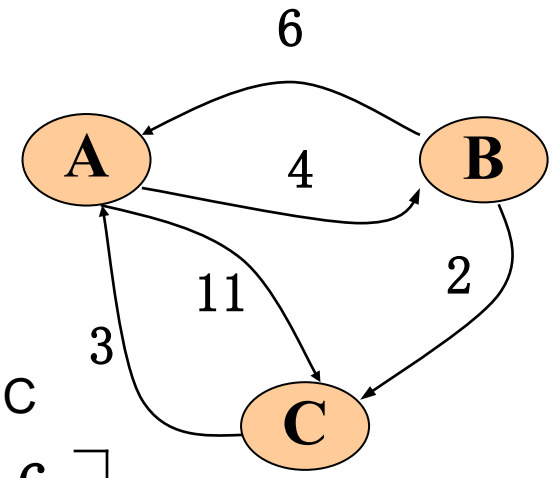
	A	B	C
A		AB	AC
B	BA		BC
C	CA	CAB	

(b) 路径

	A	B	C
A		AB	ABC
B	BA		BC
C	CA	CAB	

(b) 路径

Floyd算法求最短路径



	A	B	C
A	∞	4	6
B	6	∞	2
C	3	7	∞

(a) 路径长度

	A	B	C
A	∞	4	6
B	5	∞	2
C	3	7	∞

(a) 路径长度

加入顶点C

	A	B	C
A		AB	ABC
B	BA		BC
C	CA	CAB	

(b) 路径

	A	B	C
A		AB	ABC
B	BCA		BC
C	CA	CAB	

(b) 路径

	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(a) 路径长度

加入顶点A

	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

(a) 路径长度

加入顶点B

	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

(a) 路径长度

	1	2	3
1		AB	AC
2	BA		BC
3	CA		

(b) 路径

	1	2	3
1		AB	AC
2	BA		BC
3	CA	CAB	

(b) 路径

	1	2	3
1		AB	ABC
2	BA		BC
3	CA	CAB	

(b) 路径

1	1	1
2	2	2
3	-1	3

1	1	1
2	2	2
3	1	3

1	1	2
2	2	2
3	1	3

	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

(a) 路径长度

	1	2	3
1		AB	ABC
2	BA		BC
3	CA	CAB	

(b) 路径

	1	1	2
	2	2	2
	3	1	3

加入顶点C

	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

(a) 路径长度

	1	2	3
1		AB	ABC
2	BCA		BC
3	CA	CAB	

(b) 路径

	1	1	2
	3	2	2
	3	1	3

A→B：4

P[1][2]=1

A→C：6

P[1][3]=2

P[1][2]=1

C←B ←A

B→A：5

P[2][1]=3

P[2][3]=2

A←C ←B

C→B：7

P[3][2]=1

P[3][1]=3

B←A ←C

```
void ShortestPath_Floyd ( MGraph G )
{
   int path[NumVertices][NumVertices];
   for ( i = 0; i < G.vexnum; i++ )    //矩阵D与path初始化
      for ( j = 0; j < G.vexnum; j++ ) {
         D[i][j] = G.arcs[i][j];
         if ( D[i][j] < MAXINT )  path[i][j] = i;
         else path[i][j] = -1;          // i 到 j 无路径
      }
   for (k = 0; k < G.vexnum; k++ )     //产生D(k)及path(k)
      for ( i = 0; i < G.vexnum; i++ )
         for ( j = 0; j < G.vexnum; j++ )
            if (D[i][j] > D[i][k] + D[k][j] ) {
               D[i][j] = D[i][k] + D[k][j];
               path[i][j] = path[k][j];
            }     //缩短路径长度, 绕过 k 到 j
}
```

本章小结

本章基本学习要点如下：

(1)掌握图的相关概念,包括图、有向图、无向图、完全图、子图、连通图、度、入度、出度、简单回路和环等定义。

(2)重点掌握图的各种存储结构,包括邻接矩阵和邻接表。

(3)重点掌握图的基本运算,包括创建图、输出图、深度优先遍历、广度优先遍历算法等。

(4)掌握图的其他运算,包括最小生成树、最短路径、拓扑排序等算法。

(5)灵活运用图这种数据结构解决一些综合应用问题。

作业：

7.11