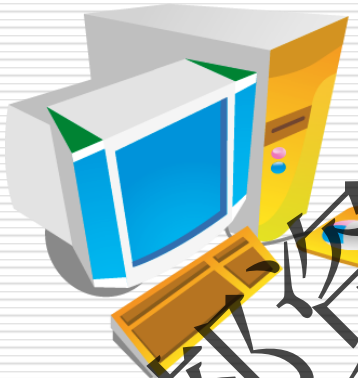
数据结构

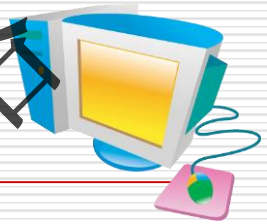


第六章　图

人工智能学院

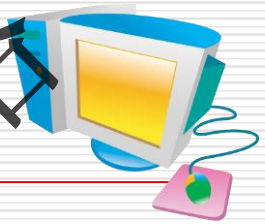
刘运

主要内容



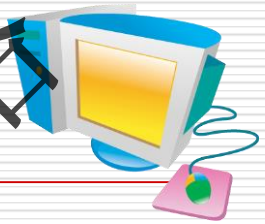
- 6.1 图的定义和基本术语
- 6.2 案例引入
- 6.3 图的类型定义
- 6.4 图的存储结构
- 6.5 图的遍历
- 6.6 图的应用
- 6.7 案例分析与实现

教学目标



- 掌握：图的基本概念及相关术语和性质；
- 熟练掌握：图的邻接矩阵和邻接表两种存储表示方法；
- 熟练掌握：图的两种遍历方法DFS和BFS；
- 熟练掌握：最短路径算法；
- 掌握：最小生成树的两种算法及拓扑排序算法；
- 了解：关键路径的算法思想

引例



- 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

6.1 图的定义

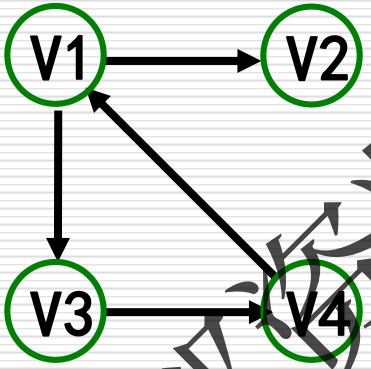
图(Graph)： G=(V,E)

V：顶点(数据元素)的有穷非空集合；

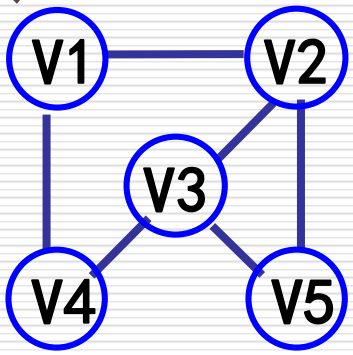
E：边的有穷集合。

有向图：每条边都是有方向的

无向图：每条边都是无方向的



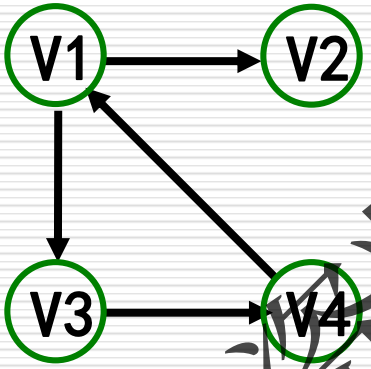
有向图G_1



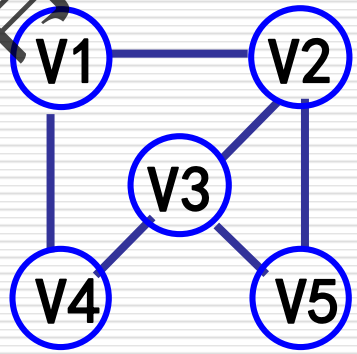
无向图G_2

6.1.1 图的定义

- 在有向图中，顶点对<x, y>是有序的，它称为从顶点x到顶点y的一条有向边。
- 在无向图中，顶点对(x, y)是无序的，它称为与顶点x和顶点y相关联的一条边。



有向图G_1

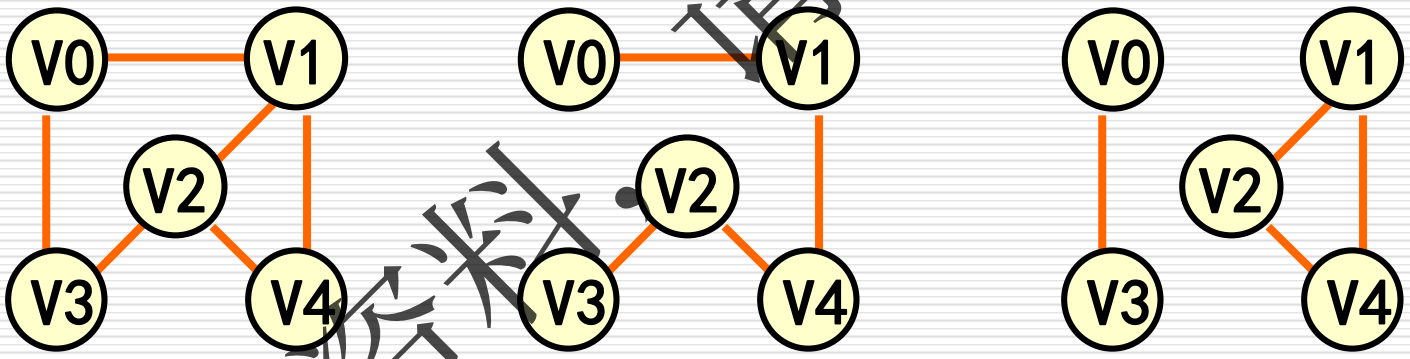


无向图G_2

6.1.1 图的基本术语

（1）子图：假设有两个图G=（V，{E}）、G’=（V’，{E’}），若V’⊆ V，E’ ⊆E，则称 G’是G的子图。

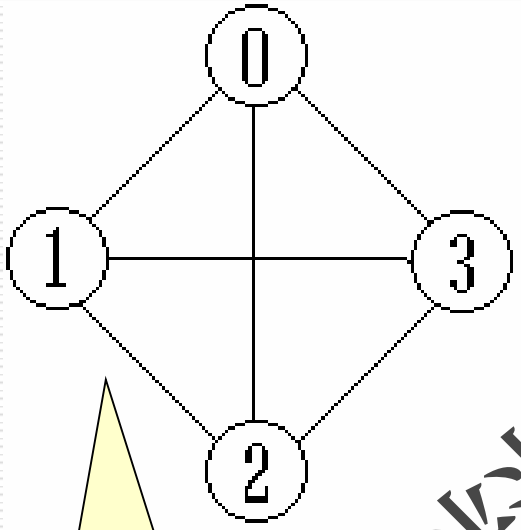
例：



(a)　　(b)　　(c)

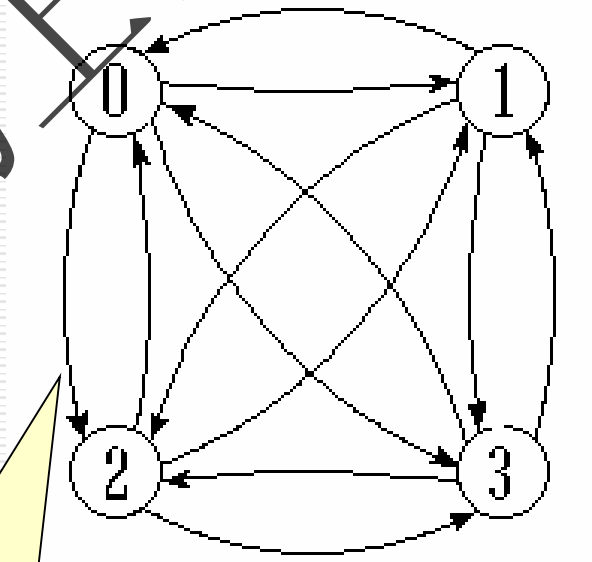
6.1.1 图的基本术语

（2）完全图：图中任意两个顶点都有一条边相连。



无向完全图

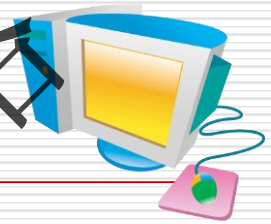
$n(n-1)/2$ 条边



有向完全图

$n(n-1)$ 条边

6.1.1 图的基本术语



（3）稀疏图和稠密图：有很少条边或弧（如$e<n\log_2 n$）的图称为稀疏图，反之称为稠密图。

（4）权和网：边上标出的具有某种含义的数值称为权，边/弧带权的图称为网。

（5）邻接点：有边/弧相连的两个顶点之间的关系。

存在(v_i, v_j)，则称v_i和v_j互为**邻接点**；

存在$<v_i, v_j>$，则称v_i**邻接到**v_j，v_j**邻接于**v_i

关联（依附）：边/弧与顶点之间的关系。

存在(v_i, v_j)/$<v_i, v_j>$，则称该边/弧关联于v_i和v_j

6.1.1 图的基本术语

（6）顶点的度、入度和出度

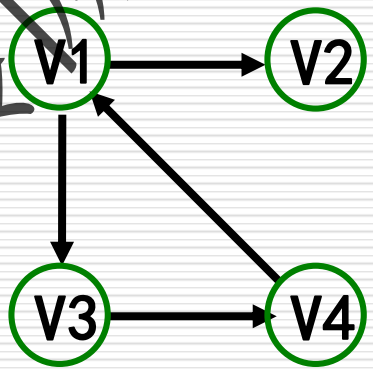
顶点V的度（TD）=与V相关联的边的数目

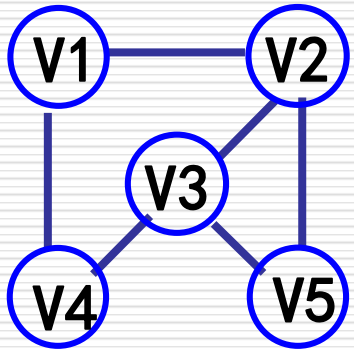
在有向图中：

顶点V的出度（OD）=以V为起点的有向边数

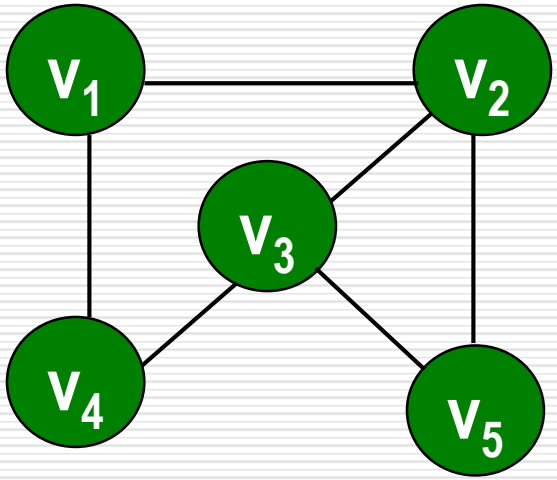
顶点V的入度（ID）=以V为终点的有向边数

顶点V的度（TD）=V的出度+V的入度





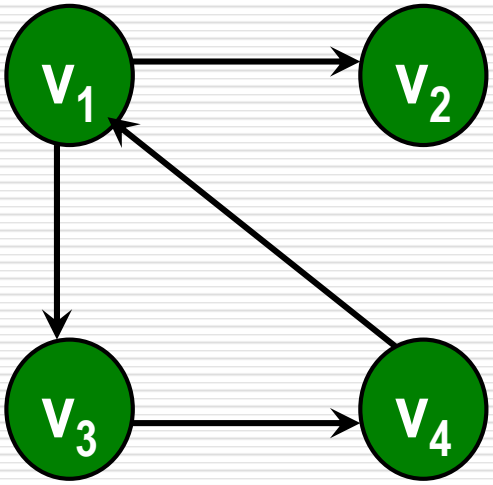
例1



无向图G_1

顶点	度
v_1	2
v_2	3
v_3	3
v_4	2
v_5	2

例2



有向图G_2

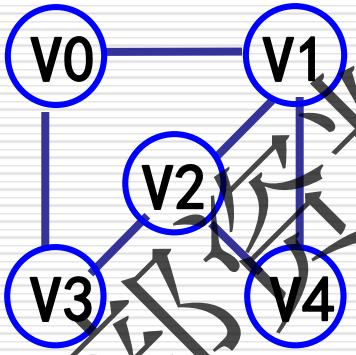
顶点	ID	OD	TD
v_1	1	2	3
v_2	1	0	1
v_3	1	1	2
v_4	1	1	2

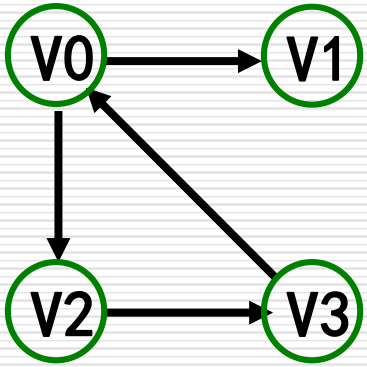
设图G的顶点数为n，边数为e

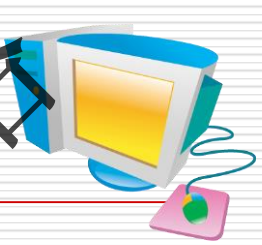
图的所有顶点的度数之和 = 2*e

（7）路径和路径长度：

在图G=<V,E>中，若有顶点序列$v_1,v_2,\ldots,v_k$,且$<v_i,v_{i+1}>\in E$(有向图)或$(v_i,v_{i+1})\in E$(无向图)，其中$i=1,2,\ldots k-1$,$v=v_1$,$u=v_k$,则称该序列是从顶点v到顶点u的路径；路径长度是指一条路径上经过的边或弧的数目。

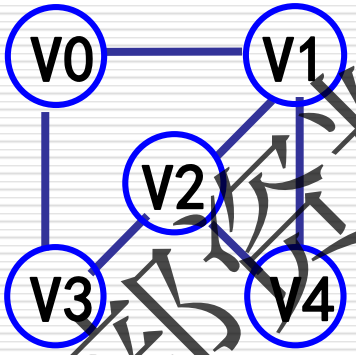


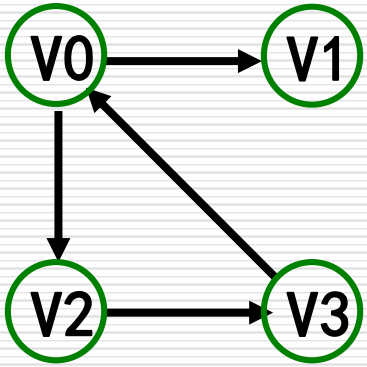




（8）回路或环：第一个顶点和最后一个顶点相同的路径称为回路或环。

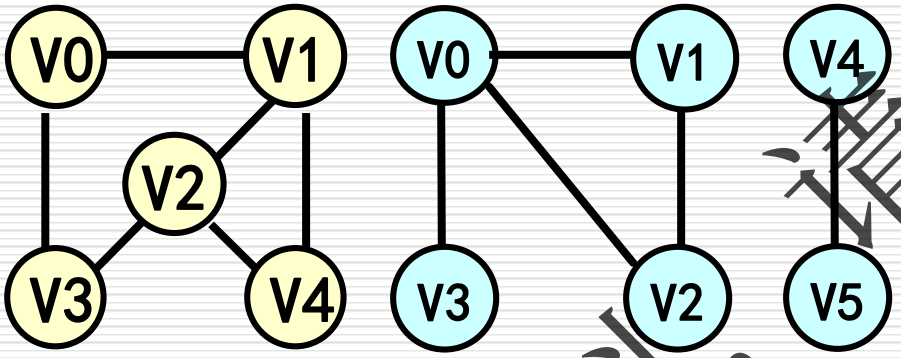
（9）简单路径、简单回路或简单环：序列中顶点为重复出现的路径称为简单路径。除了第一个顶点和最后一个顶点之外，其余顶点不重复出现的回程，称为简单回路或简单环。



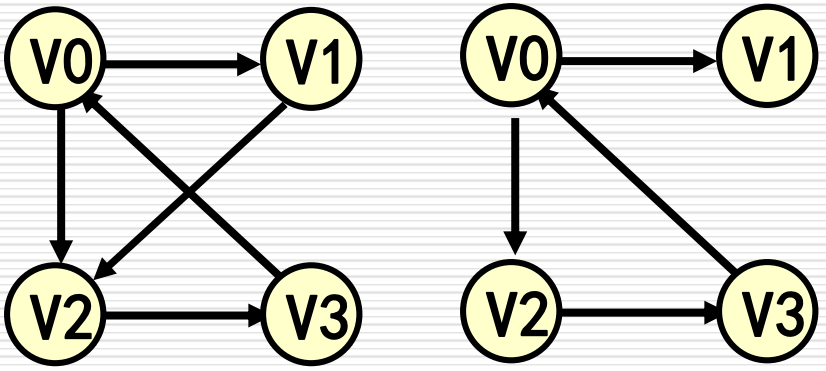


(10) 连通图、强连通图

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)。



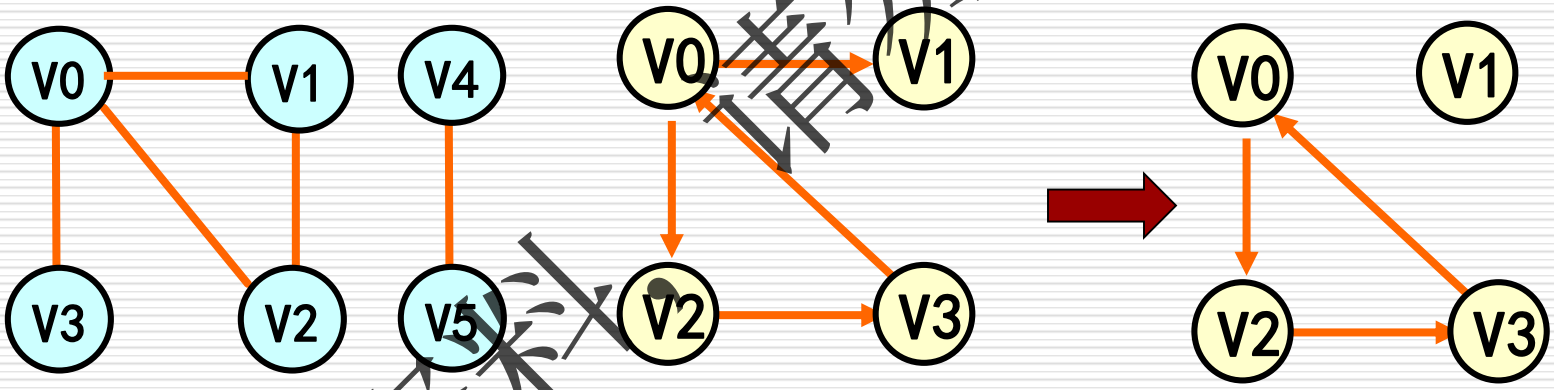
(a) 连通图　　(b) 非连通图



(c) 强连通图　　(d) 非强连通图

(11)连通分量、强连通图分量

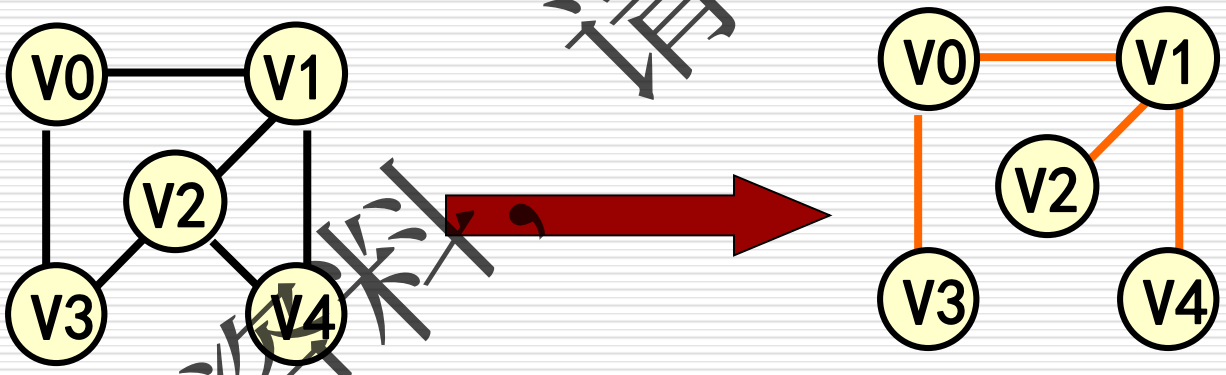
无（有）向图的极大连通子图称为其(强)连通分量。



非连通图，有两个连通分量　　非强连通图　　有两个强连通分量

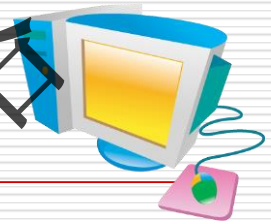
（12）连通图的生成树

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，但只有足以构成一棵树的n-1条边。



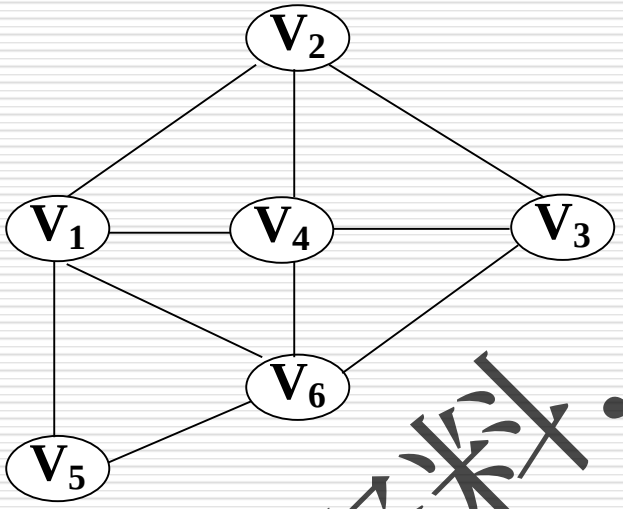
(a)连通图G1　　(b)连通图G1的一个生成树

生成树的有关性质

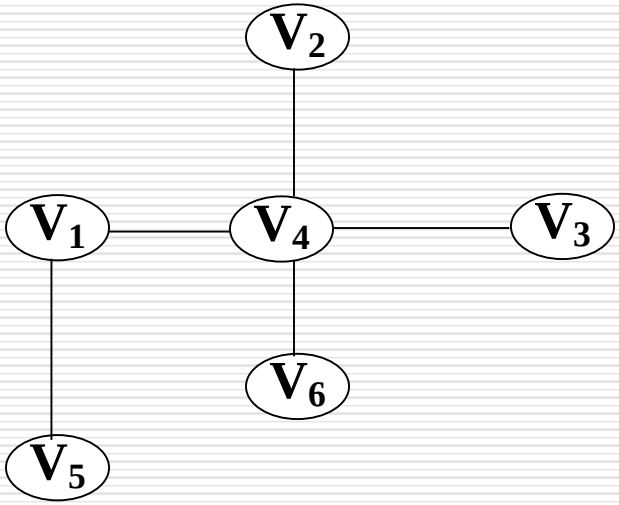


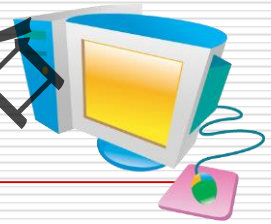
- ✓ 如果在一棵生成树上添加一条边，必定构成一个环。
- ✓ 一棵有n个顶点的生成树有且仅有n-1条边。
- ✓ 如果一个图有n个顶点和小于n-1条边，则是非连通图。
- ✓ 如果一个图有n个顶点和多于n-1条边，则一定有环。
- ✓ 有n-1条边的图不一定是生成树。
- ✓ 生成树可能不惟一。

无向图及其生成树



无向图G





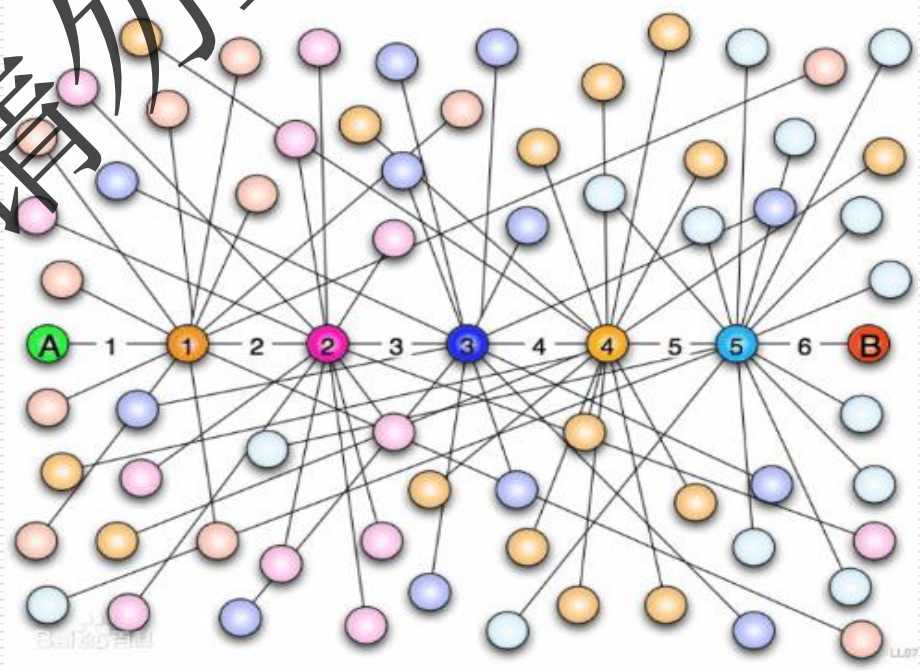
（13）有向树和生成森林

有一个顶点的入度为0，其余顶点的入度均为1的有向图称为有向树。一个有向图的生成森林由若干棵有向树组成，含有图中全部顶点，但只有足以构成若干棵不相交的有向树的弧。

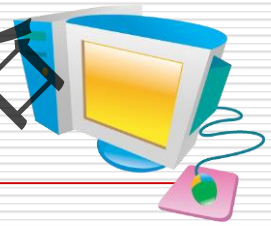
6.2 案例引入

案例6.1： 六度空间理论

你和任何一个陌生人之间所间隔的人不会超过6个，也就是说，最多通过6个中间人你就能够认识任何一个陌生人。



6.3 图的类型定义



ADT Graph{

数据对象V：V是具有相同特性的数据元素的集合，即顶点集

数据关系R：R＝{VR}

VR＝{<v,w>|v,w∈V 且P（v,w)}

基本操作P：

{ CreateGraph(&G,V,VR);

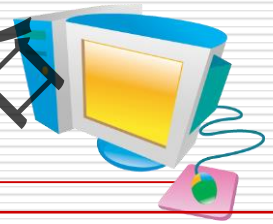
DFSTraverse(G);

BFSTraverse(G);

……

}ADT Graph

主要操作



CreateGraph(&G,V,VR)

初始条件：V是图的顶点集，VR是图中弧的集合。

操作结果：按V和VR的定义构造图G。

DFSTraverse(G)

初始条件：图G存在。

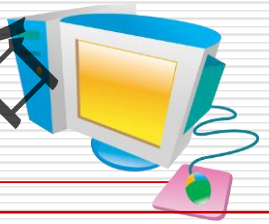
操作结果：对图进行深度优先遍历。

BFSTraverse(G)

初始条件：图G存在。

操作结果：对图进行广度优先遍历。

6.4 图的存储结构



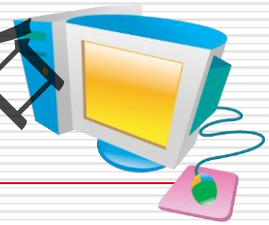
顺序存储结构：　　数组表示法（邻接矩阵）

链式存储结构：　　多重链表

- 邻接表
- 邻接多重表
- 十字链表

重点介绍：　邻接矩阵(数组)表示法
　　邻接表(链式)表示法

6.4.1 邻接矩阵表示法（数组表示法）



建立一个顶点表（记录各个顶点信息）和一个邻接矩阵（表示各个顶点之间关系）。

即一维数组存顶点，二维数组存关系。

设图 A = (*V*, *E*) 有 n 个顶点，则图的邻接矩阵是一个二维数组 A.Edge[n][n]，定义为：

$$\mathbf{A}.Edge[i][j] = \begin{cases} 1, & \text{如果} < i, j > \in E \text{ 或者 } (i, j) \in E \\ 0, & \text{否则} \end{cases}$$

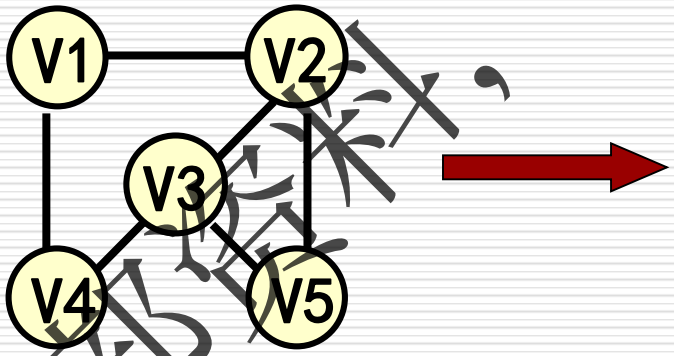
数组表示法的特点（无向图）

1）无向图的邻接矩阵是对称矩阵，同一条边表示了两次；

2）顶点v的度：等于二维数组对应行（或列）中值为1的元素个数；

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；

顶点表：（ v1 v2 v3 v4 v5 ）



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

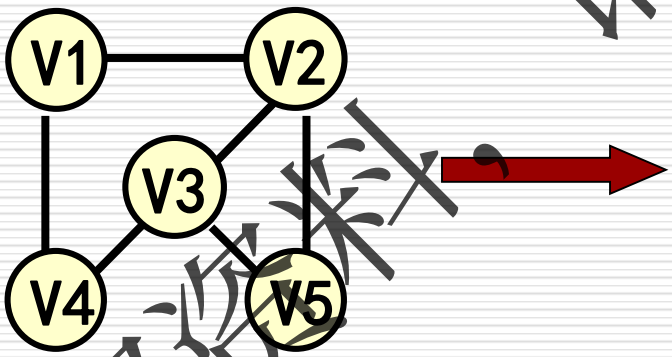
数组表示法的特点（无向图）

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）设图的顶点数为 n ，用有n个元素的一维数组存储图的顶点，用邻接矩阵表示边，则G占用的存储空间为：$n+n^2$；

图的存储空间占用量只与它的顶点数有关，与边数无关；

适用于边稠密的图；



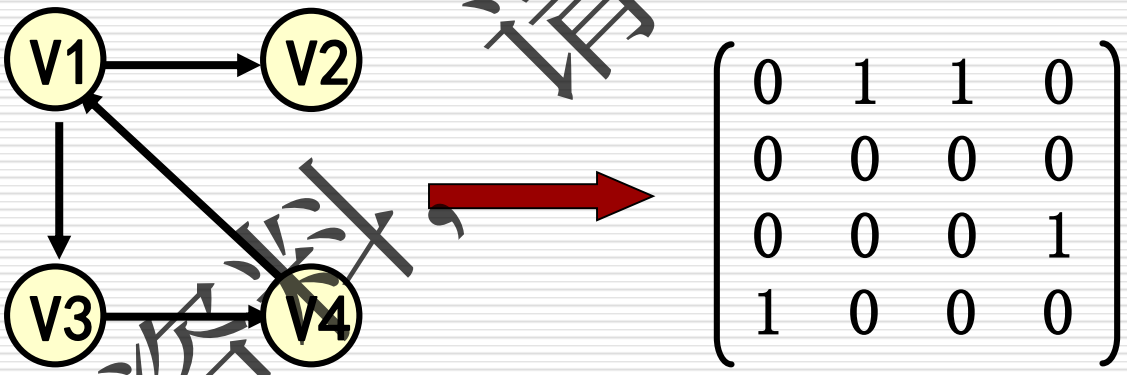
$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

数组表示法的特点（有向图）

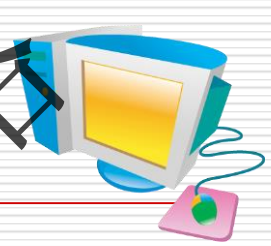
1）有向图的邻接矩阵不一定是对称的；

2) 顶点v的出度：等于二维数组对应行中值为1的元素个数；

3）顶点v的入度：等于二维数组对应列中值为1的元素个数；



邻接矩阵的存储表示



```
//用两个数组分别存储顶点表和邻接矩阵
#define MaxInt 32767          //表示极大值，即∞
#define MVNum 100             //最大顶点数
typedef char VerTexType;      //假设顶点的数据类型为字符型
typedef int ArcType;          //假设边的权值类型为整型
typedef struct{
  VerTexType vexs[MVNum];           //顶点表
  ArcType arcs[MVNum][MVNum];       //邻接矩阵
  int vexnum,arcnum;                //图的当前点数和边数
}AMGraph;
```

采用邻接矩阵表示法创建无向网

```
4    5
A B C D
A B 500
A C 200
A D 150
B C 400
C D 600
```

算法思想：

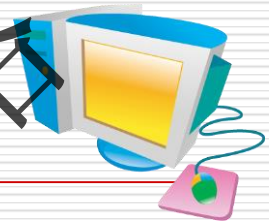
（1）输入总顶点数和总边数。

（2）依次输入点的信息存入顶点表中。

（3）初始化邻接矩阵，使每个权值初始化为极大值。

（4）构造邻接矩阵。

【算法描述】

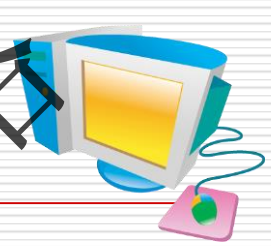


```
Status CreateUDN(AMGraph &G){
    //采用邻接矩阵表示法，创建无向网G
    cin>>G.vexnum>>G.arcnum;
     //输入总顶点数，总边数
    for(i = 0; i<G.vexnum; ++i)
      cin>>G.vexs[i];
     //依次输入点的信息
    for(i = 0; i<G.vexnum;++i)
     //初始化邻接矩阵，边的权值均置为极大值
       for(j = 0; j<G.vexnum;++j)
         G.arcs[i][j] = MaxInt;
```

【算法描述】

```
4    5
A B C D
A B 500
A C 200
A D 150
B C 400
C D 600
```

```
for(k = 0; k<G.arcnum;++k){//构造邻接矩
        cin>>v1>>v2>>w;
//输入一条边依附的顶点及权值
        i = LocateVex(G, v1);
        j = LocateVex(G, v2);
//确定v1和v2在G中的位置
        G.arcs[i][j] = w; //边<v1, v2>的权值置为w
        G.arcs[j][i] = G.arcs[i][j];
//置<v1, v2>的对称边<v2, v1>的权值为w
    }//for
    return OK;
}//CreateUDN
```



```
int LocateVex(MGraph G,VertexType u)
{//存在则返回u在顶点表中的下标;否则返回-1
  int i;
  for(i=0;i<G.vexnum;++i)
    if(u==G.vexs[i])
      return i;
  return -1;
}
```

对于n个顶点的图或网，时间效率=$O(n^2)$

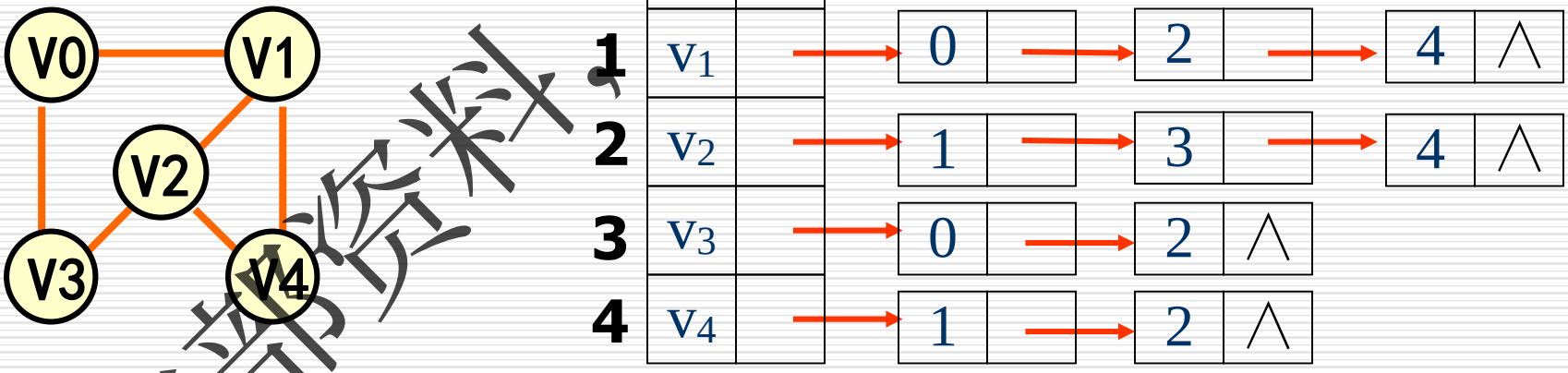
6.4.2 邻接表

顶点： 通常按编号顺序将顶点数据存储在一维数组中。

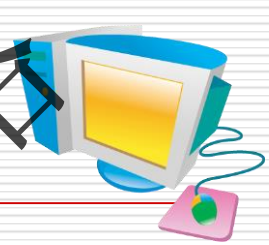
关联同一顶点的边： 用线性链表存储

例

该结点表示边（V0,V1），其中的1是V1的序号，即一维数组中的下标。

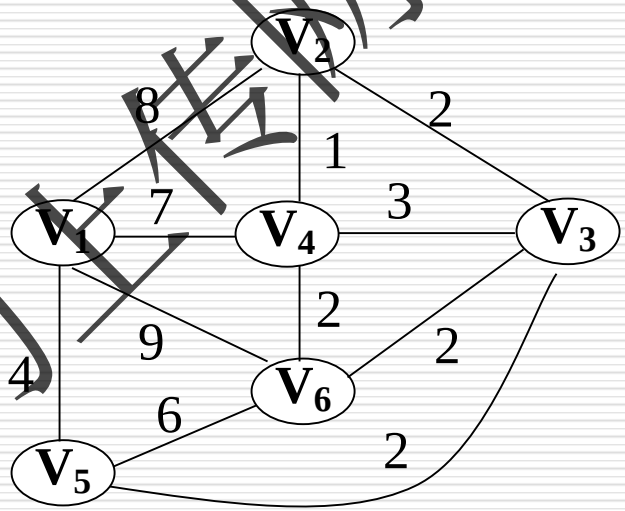


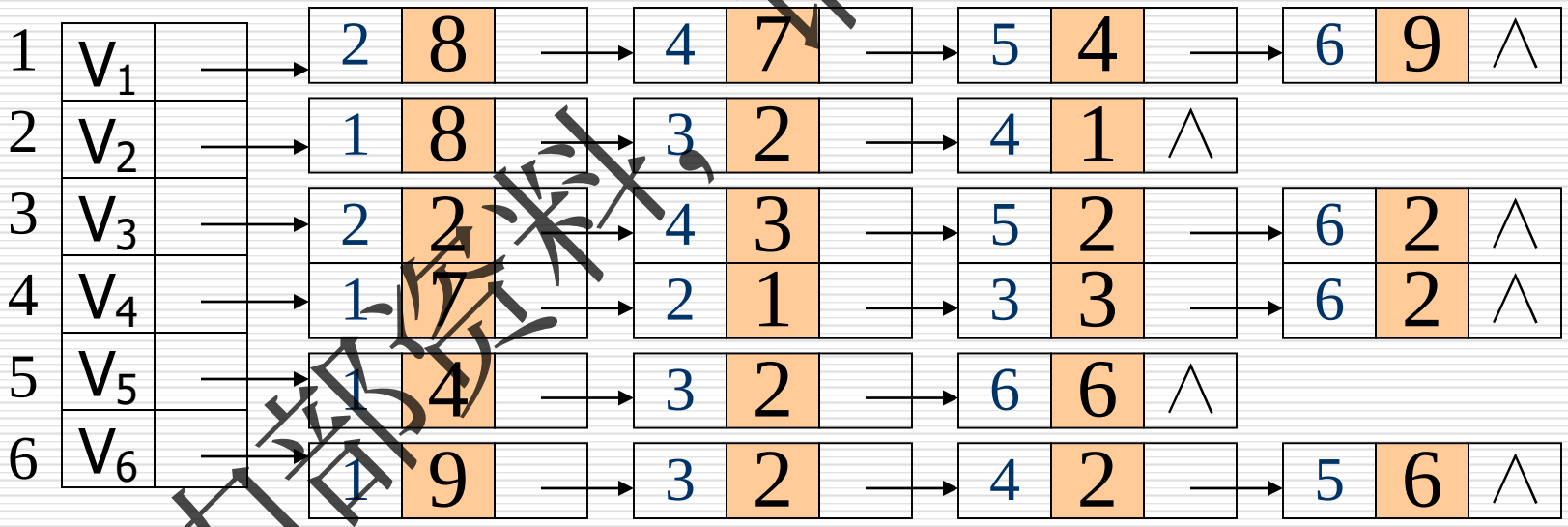
网的邻接链表表示



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) 边结点

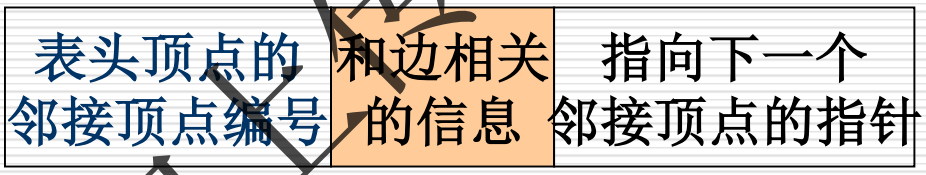




(b) 邻接链表

邻接表的类型定义

```
#define MVNum  100
 typedef struct ArcNode{
      int      adjvex;
      struct ArcNode *nextarc;
      OtherInfo  info;
 }ArcNode;
```

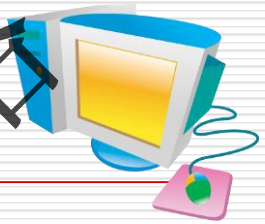


(a) 边结点

```
typedef struct  VNode{
     VertexType   data;
     ArcNode  *firstarc;
}VNode,AdjList[MVNum];
```

(b) 表头结点

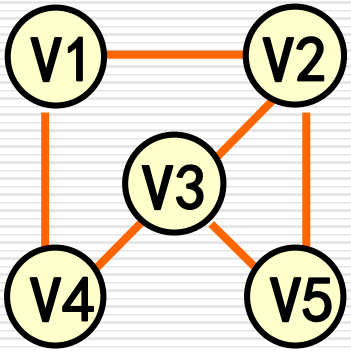
图的邻接表表示

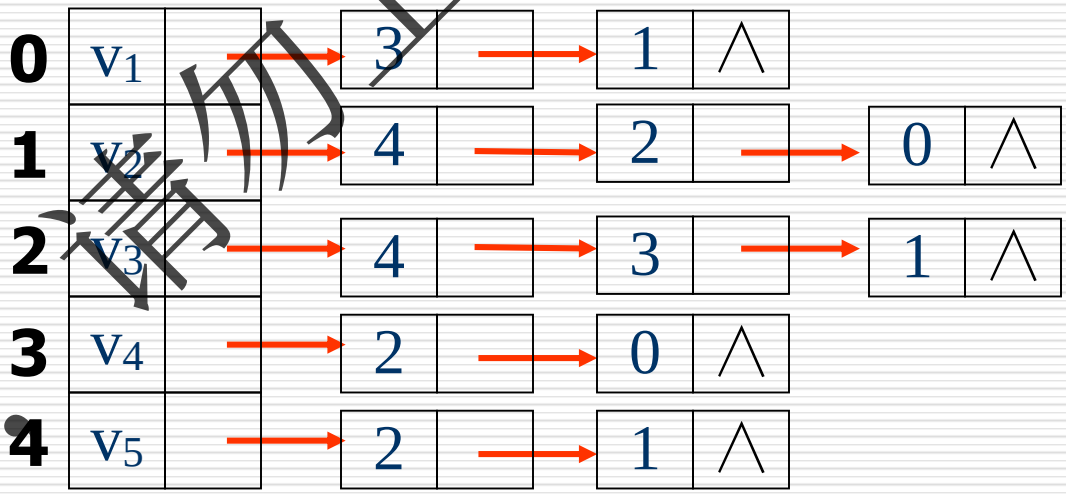


```
typedef struct {
      AdjList  vertices;
      int vexnum,arcnum;
}ALGraph;
```

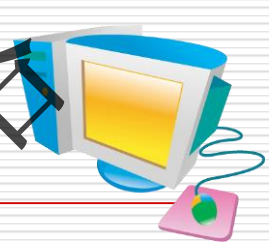
无向图的邻接表表示

G1的邻接链表



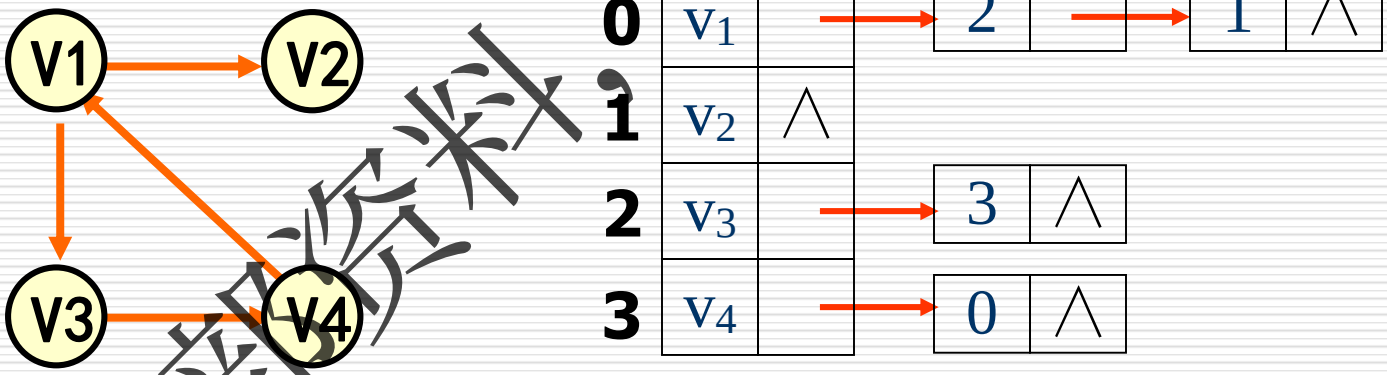


有向图的邻接表表示



顶点：通常按编号顺序将顶点数据存储在一维数组中以同一顶点为起点的弧：用线性链表存储

G2的邻接表：

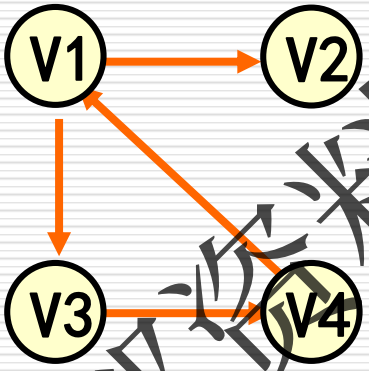


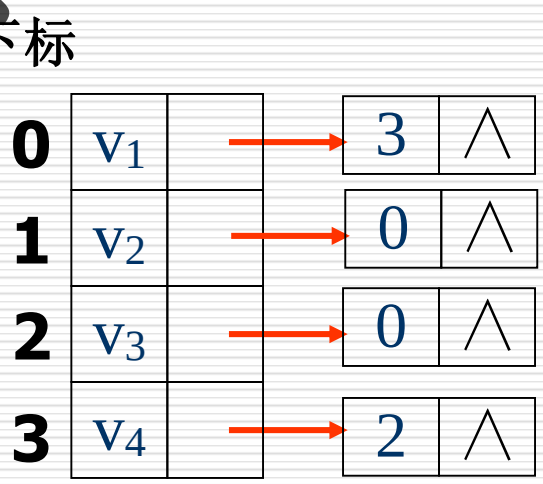
有向图的逆邻接表表示

顶点：通常按编号顺序将顶点数据存储在一维数组中。

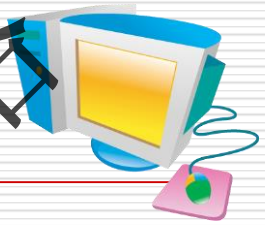
以同一顶点为终点的弧：用线性链表存储。

G1的逆邻接表：





采用邻接表表示法创建无向网



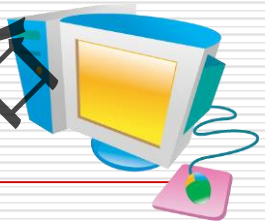
算法思想：

（1）输入总顶点数和总边数。

（2）依次输入点的信息存入顶点表中，使每个表头结点的指针域初始化为NULL。

（3）创建邻接表。

【算法描述】

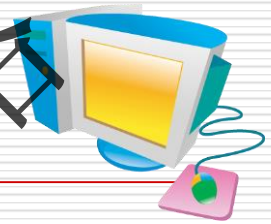


```
Status CreateUDG(ALGraph &G){
    //采用邻接表表示法，创建无向图G
   cin>>G.vexnum>>G.arcnum;
      //输入总顶点数，总边数
  for(i = 0; i<G.vexnum; ++i){
      //输入各点，构造表头结点表
   cin>> G.vertices[i].data;
      //输入顶点值
   G.vertices[i].firstarc=NULL;
      //初始化表头结点的指针域为NULL
 }//for
```

【算法描述】

```
for(k = 0; k<G.arcnum;++k){                    //输入各边，构造邻接表
    cin>>v1>>v2;                               //输入一条边依附的两个顶点
    i = LocateVex(G, v1);  j = LocateVex(G, v2);
    p1=new ArcNode;          //生成一个新的边结点*p1
      p1->adjvex=j;          //邻接点序号为j
      p1->nextarc= G.vertices[i].firstarc;
    G.vertices[i].firstarc=p1; //将新结点*p1插入顶点vi的边表头部
   p2=new ArcNode; //生成另一个对称的新的边结点*p2
      p2->adjvex=i;          //邻接点序号为i
      p2->nextarc= G.vertices[j].firstarc;
    G.vertices[j].firstarc=p2; //将新结点*p2插入顶点vj的边表头部
  }//for
  return OK;
}//CreateUDG
```

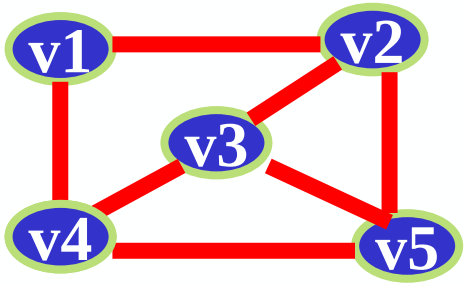
邻接表表示法的特点



优点：空间效率高，容易寻找顶点的邻接点；

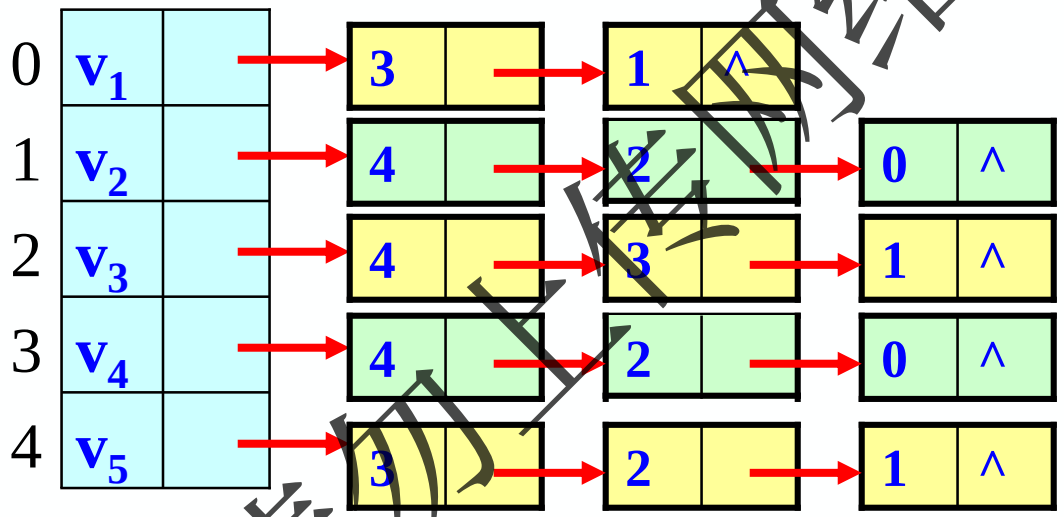
缺点：判断两顶点间是否有边或弧，需搜索两结点对应的单链表，没有邻接矩阵方便。

邻接矩阵与邻接表表示法的关系



（ v1 v2 v3 v4 v5 ）

0	1	0	1	0	v1
1	0	1	0	1	v2
0	1	0	1	1	v3
1	0	1	0	1	v4
0	1	1	1	0	v5



1. 联系：邻接表中每个链表对应于邻接矩阵中的一行，链表中结点个数等于一行中非零元素的个数。

邻接矩阵与邻接表表示法的关系

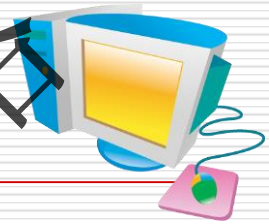
2. 区别：

① 对于任一确定的无向图，邻接矩阵是唯一的（行列号与顶点编号一致），但邻接表不唯一（链接次序与顶点编号无关）。

② 邻接矩阵的空间复杂度为$O(n^2)$，而邻接表的空间复杂度为$O(n+e)$。

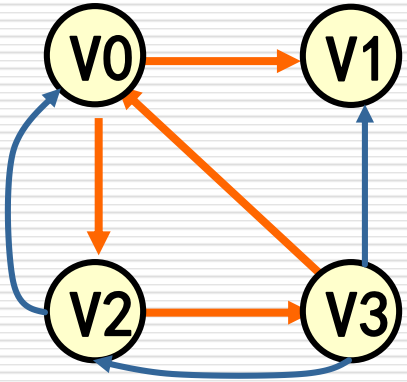
3. 用途：邻接矩阵多用于稠密图；而邻接表多用于稀疏图

6.4.3 有向图的十字链表表示



❖将有向图的邻接表和逆邻接表结合起来得到的链表。

❖在十字链表中，顶点结点存储数据元素，弧结点存储弧及其上的信息。



顶点结点

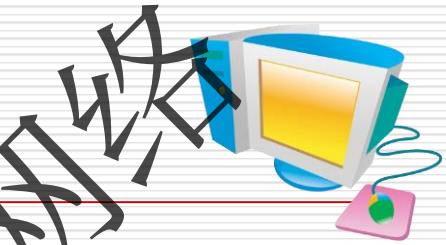
data	firstin	firstout

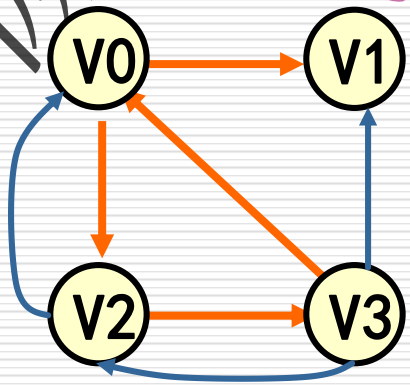
<vi,vj>

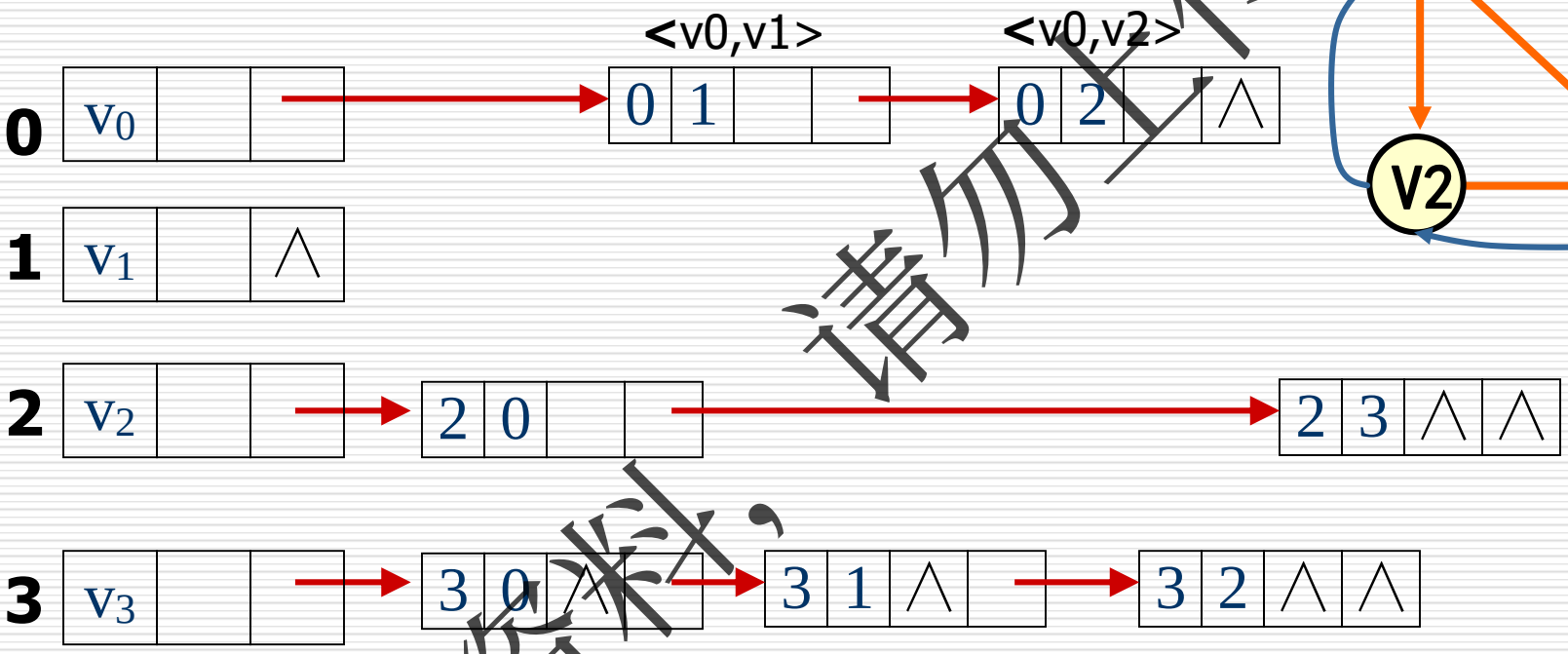
弧结点

tailvex	headvex	hlink	tlink	info

6.4.3 十字链表（用于有向图）

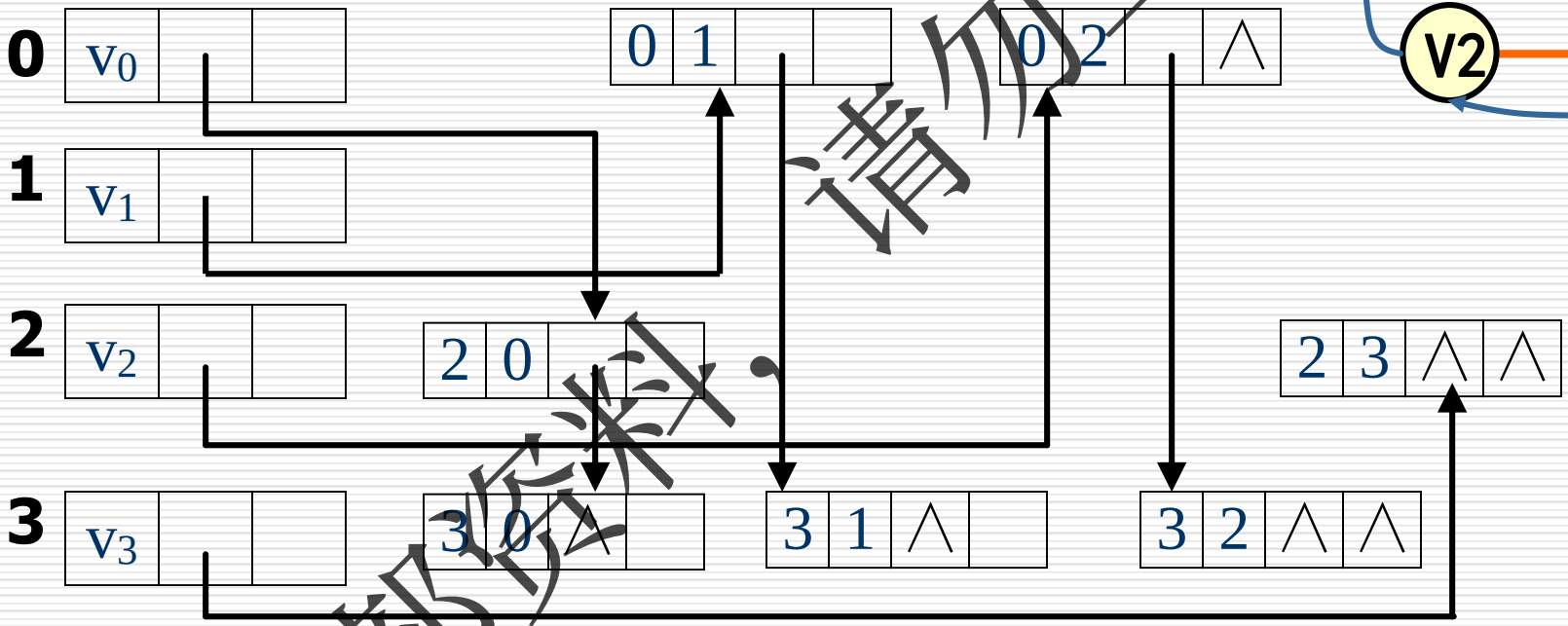


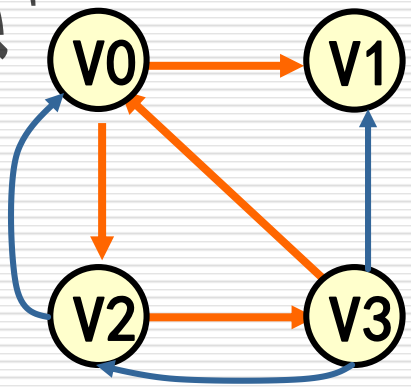




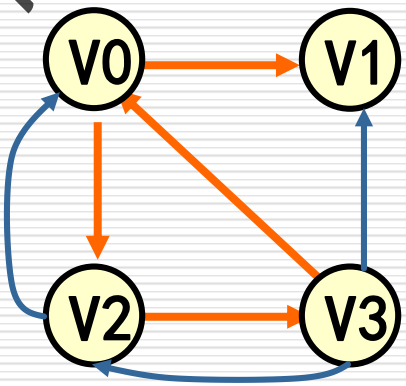
邻接链表

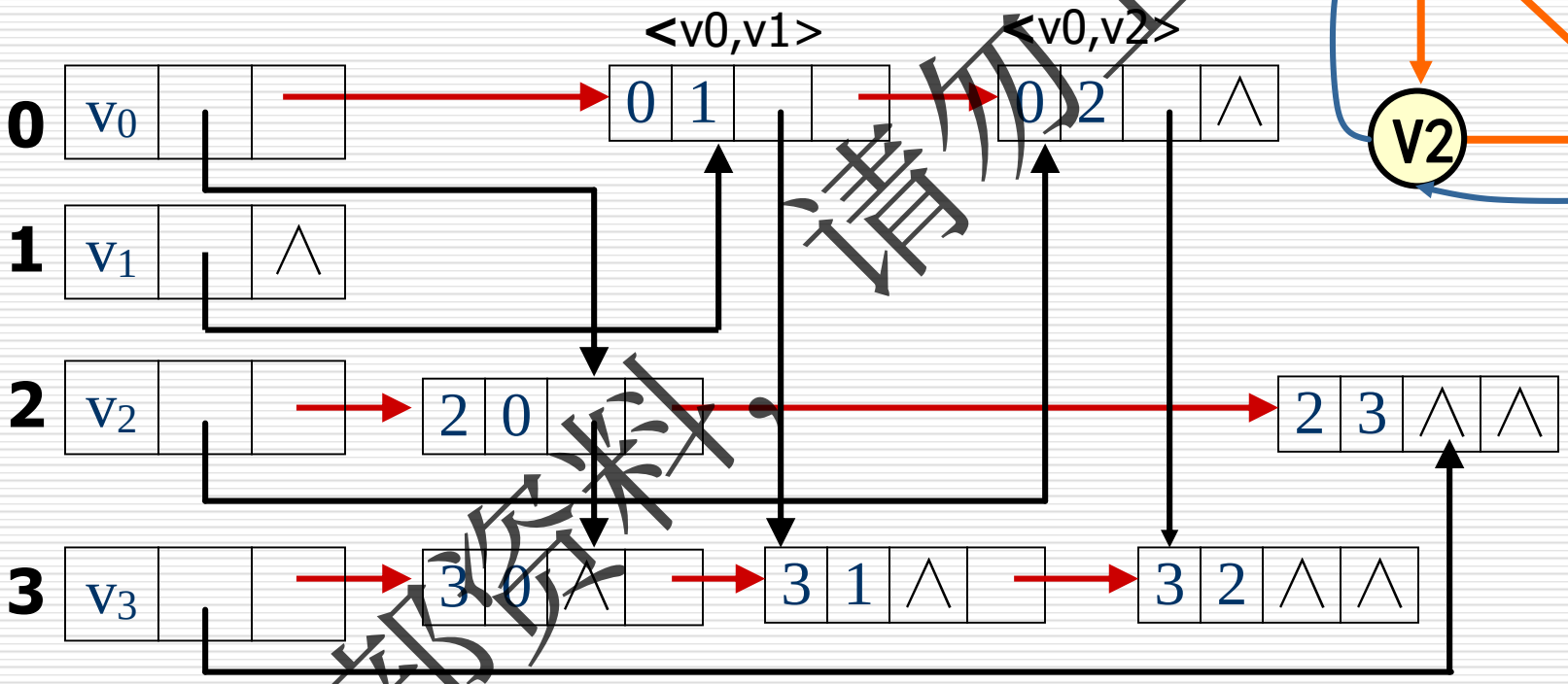
逆邻接链表



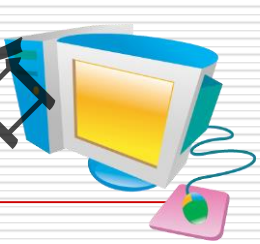


十字链表





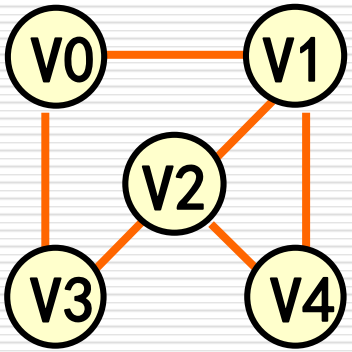
6.4.4 邻接多重表（用于无向图）

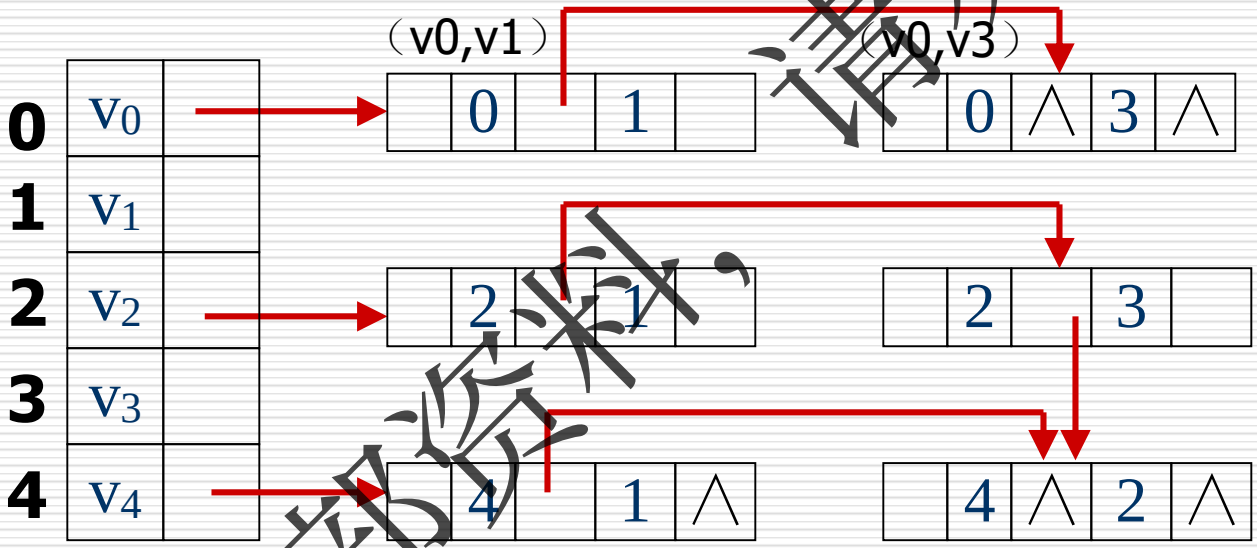


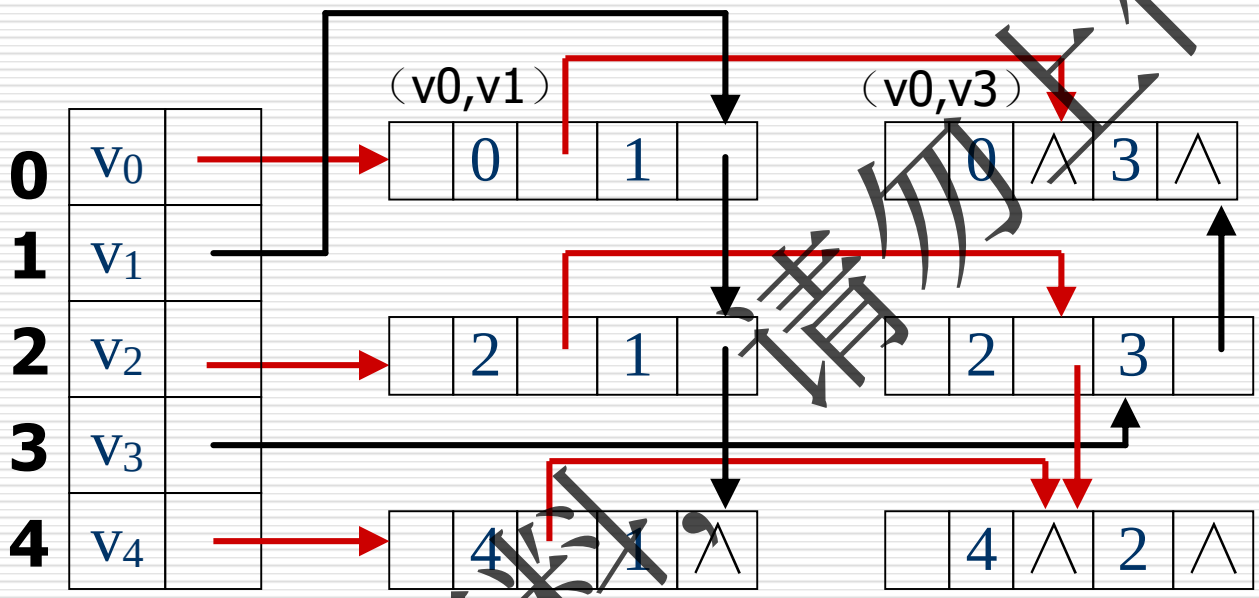
❖无向图的邻接多重表表示中，每条边只表示一次。

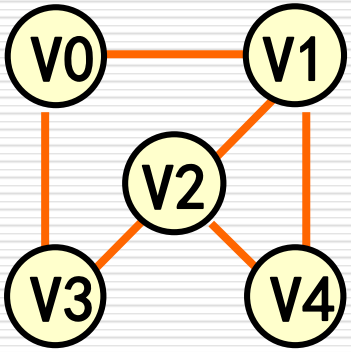
(vi,vj)

mark	ivex	ilink	jvex	jlink

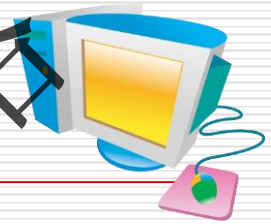






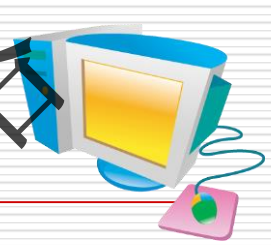


6.5 图的遍历



- 从图的某个顶点出发，访问图中的所有顶点，且使每个顶点仅被访问一次。这一过程叫做图的遍历。
- 图的遍历操作是求解图的连通性问题、拓扑排序等问题的基础。
- 遍历方法:深度优先遍历和广度优先遍历

深度优先遍历图的基本思想



从图中某顶点v出发：

(1)访问顶点v；

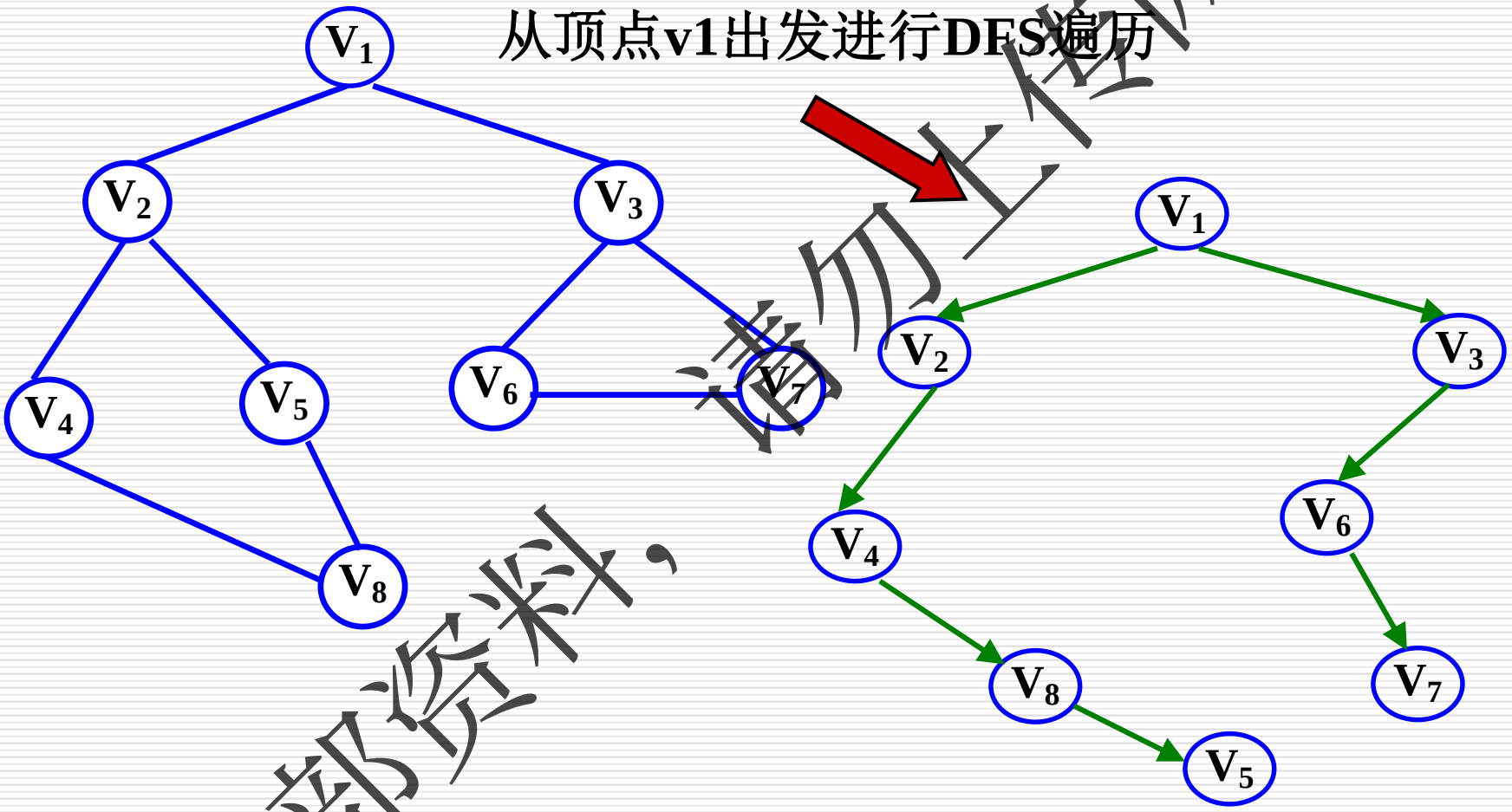
(2)依次从v的未被访问的邻接点出发，对图进行深度优先遍历；直至图中和v有路径相通的顶点都被访问；

(3)若此时图中尚有顶点未被访问，则从一个未被访问的顶点出发，重新进行深度优先遍历，直到图中所有顶点均被访问过为止。

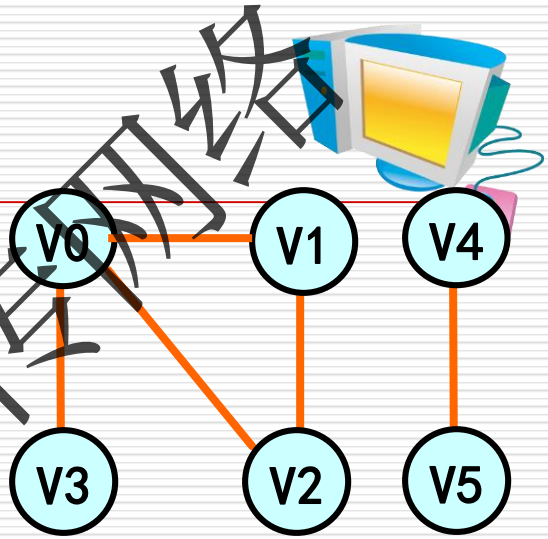
深度优先遍历过程是递归的，在遍历过程中，若某个顶点的所有邻接顶点均被访问过，则需要回溯。

6.5.1 深度优先搜索(DFS)

从顶点v1出发进行DFS遍历

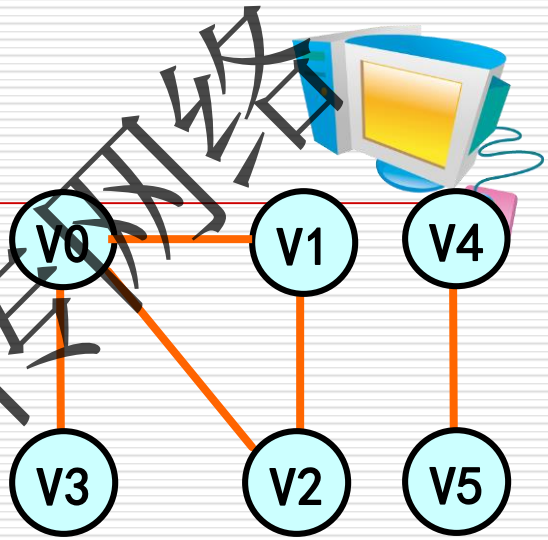


算法6.3 深度优先搜索遍历连通图



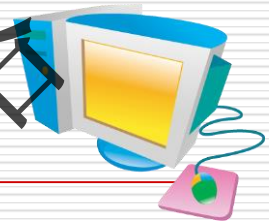

```
Bool visited[Mvnum];
//访问标志数组，其初值为“false”
void DFS(Graph G, int v)
{//从第v个顶点出发递归地深度优先遍历图G
   cout<<v, visited[v] = true;//访问第v个顶点
   for(w为v的第一个邻接顶点; w存在；w取v的下一个邻接顶点)
      if (!visited[w]) DFS(G, w);
}//DFS
```

算法6.4 深度优先搜索遍历非连通图



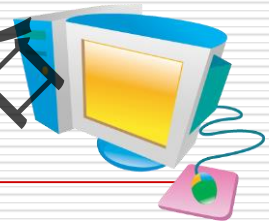

```
void DFSTraverse(Graph G) {
   for(v=0;v<G.vexnum;++v)
       visited[v] = false;
   for(v=0;v<G.vexnum;++v)
       if (!visited[v]) DFS(G,v);
  //对尚未访问的顶点调用DFS
}//DFSTraverse
```

算法6.5 采用邻接矩阵表示图的深度优先搜索遍历



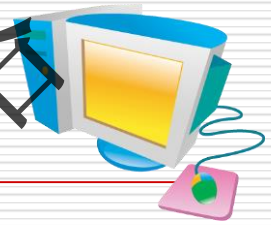
```
void DFS_AM(AMGraph G, int v){          //图G为邻接矩
   阵类型
  cout<<v;  visited[v] = true;          //访问第v个顶点
  for(w = 0; w< G.vexnum; w++)  //依次检查邻接矩阵v所在的行
      if((G.arcs[v][w]!=0)&& (!visited[w]))
           DFS_AM(G, w);
    //w是v的邻接点，如果w未访问，则递归调用DFS
}
```

算法6.6 采用邻接表表示图的深度优先搜索遍历



```
void DFS_AL(ALGraph G, int v){  //图G为邻接表类型
  cout<<v;  visited[v] = true;  //访问第v个顶点
  p= G.vertices[v].firstarc;
  //p指向v的边链表的第一个边结点
 while(p!=NULL){              //边结点非空
  w=p->adjvex;                //表示w是v的邻接点
  if(!visited[w])  DFS_AL(G, w);
//如果w未访问，则递归调用DFS
  p=p->nextarc;               //p指向下一个边结点
 }
}
```

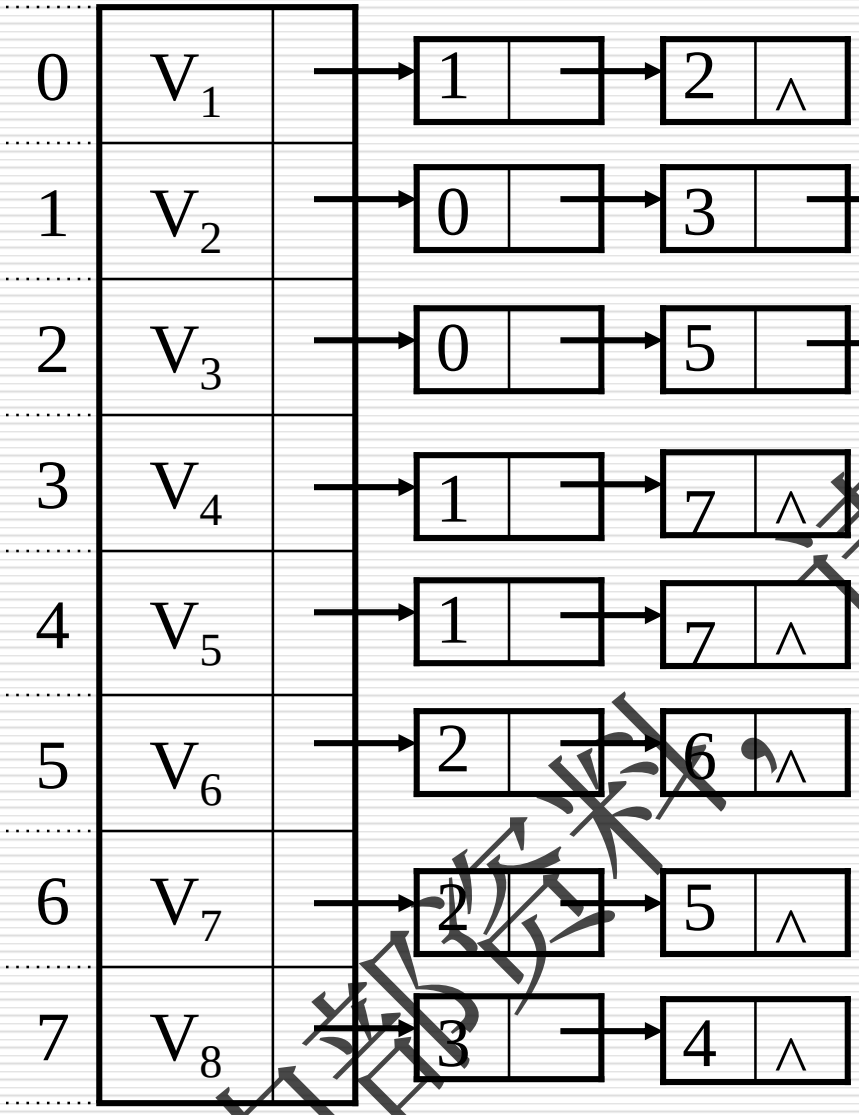
DFS算法效率分析

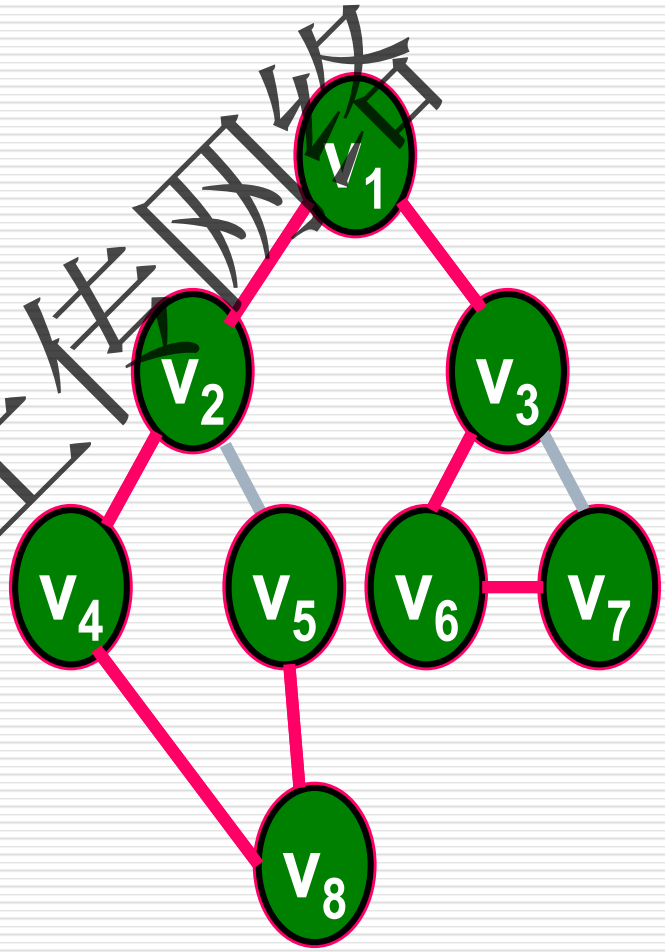


- 遍历图的过程实际上是对每个顶点查找其邻接点的过程。用邻接矩阵表示图时，查找每个顶点的邻接点的时间复杂度为$O(n^2)$。
- 用邻接表来表示图，查找每个邻接点的时间复杂度为$O(e)$，加上访问n个头结点的时间，故以邻接表做存储结构时，深度优先搜索遍历图的时间复杂度为$O(n+e)$。

结论：
稠密图适于在邻接矩阵上进行深度优先遍历；
稀疏图适于在邻接表上进行深度优先遍历。

示例及分析1

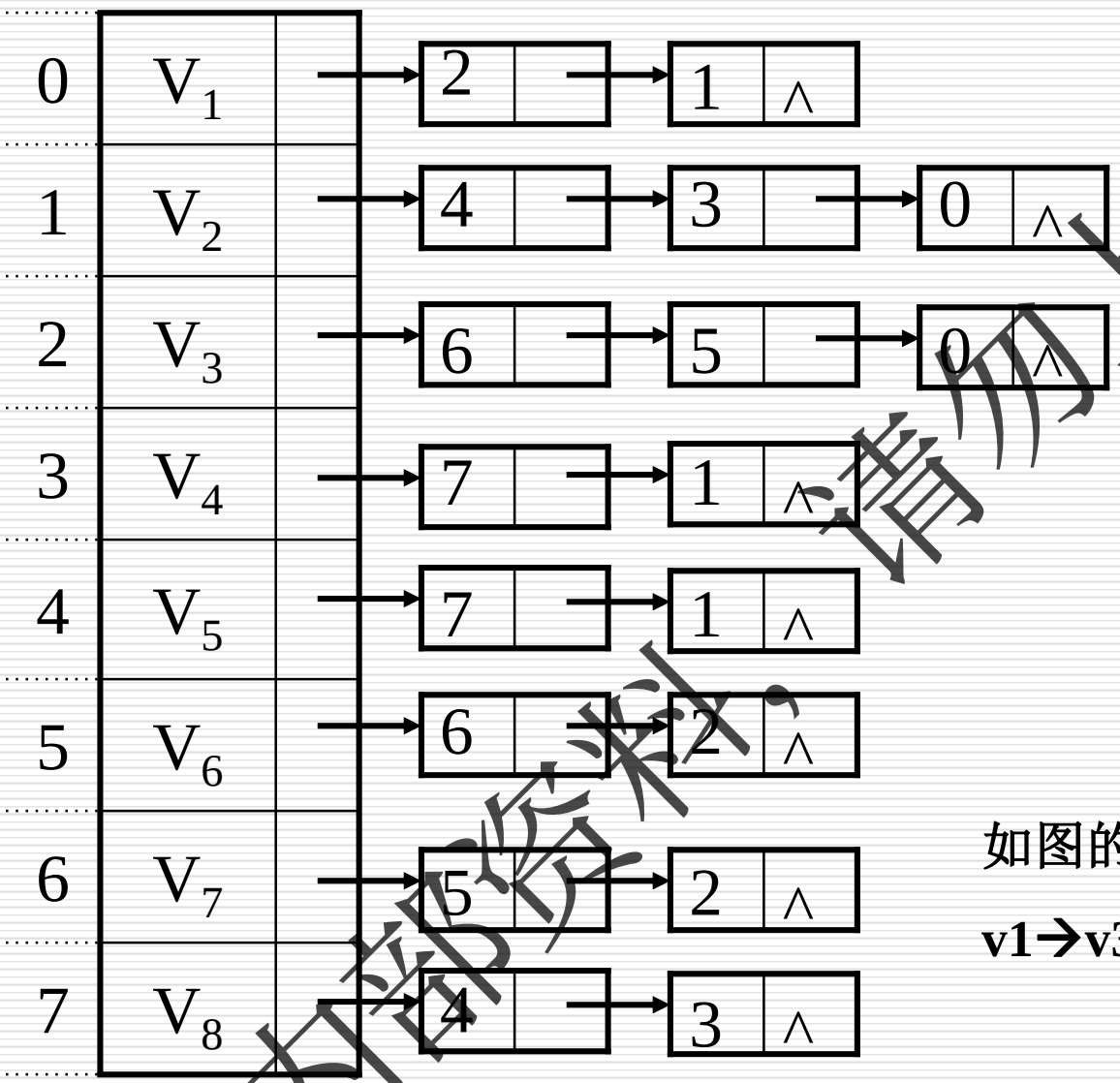


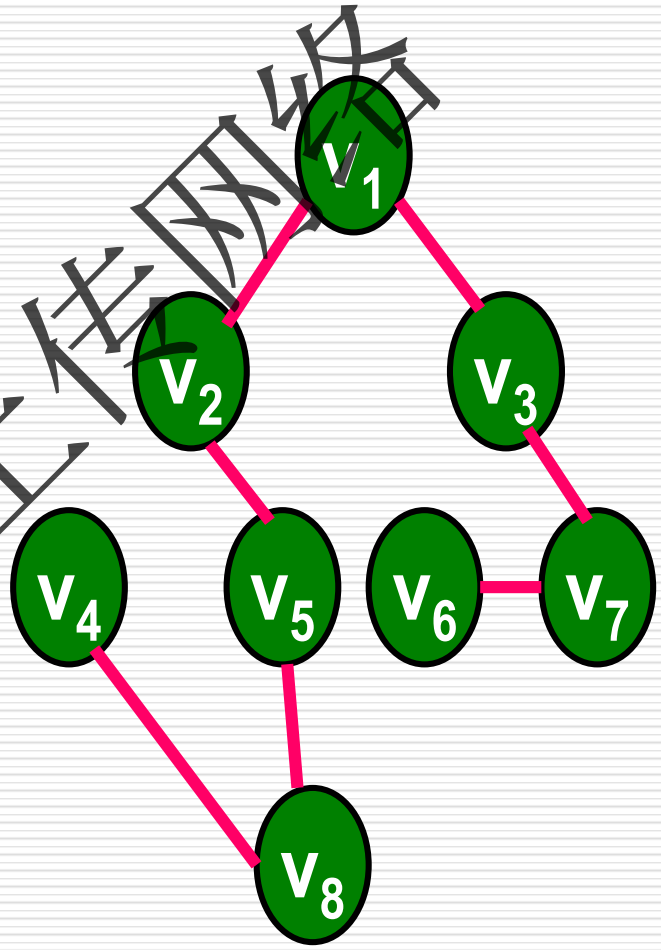


如图的深度优先搜索序列为：

v1→v2→v4→v8→v5→v3→v6→v7

示例及分析2





如图的深度优先搜索序列为：

v1→v3→v7→v6→v2→v5→v8→v4

6.5.2 广度优先搜索BFS

广度优先遍历图的方法

- 从图中某顶点v出发：

  ① 访问顶点v ；

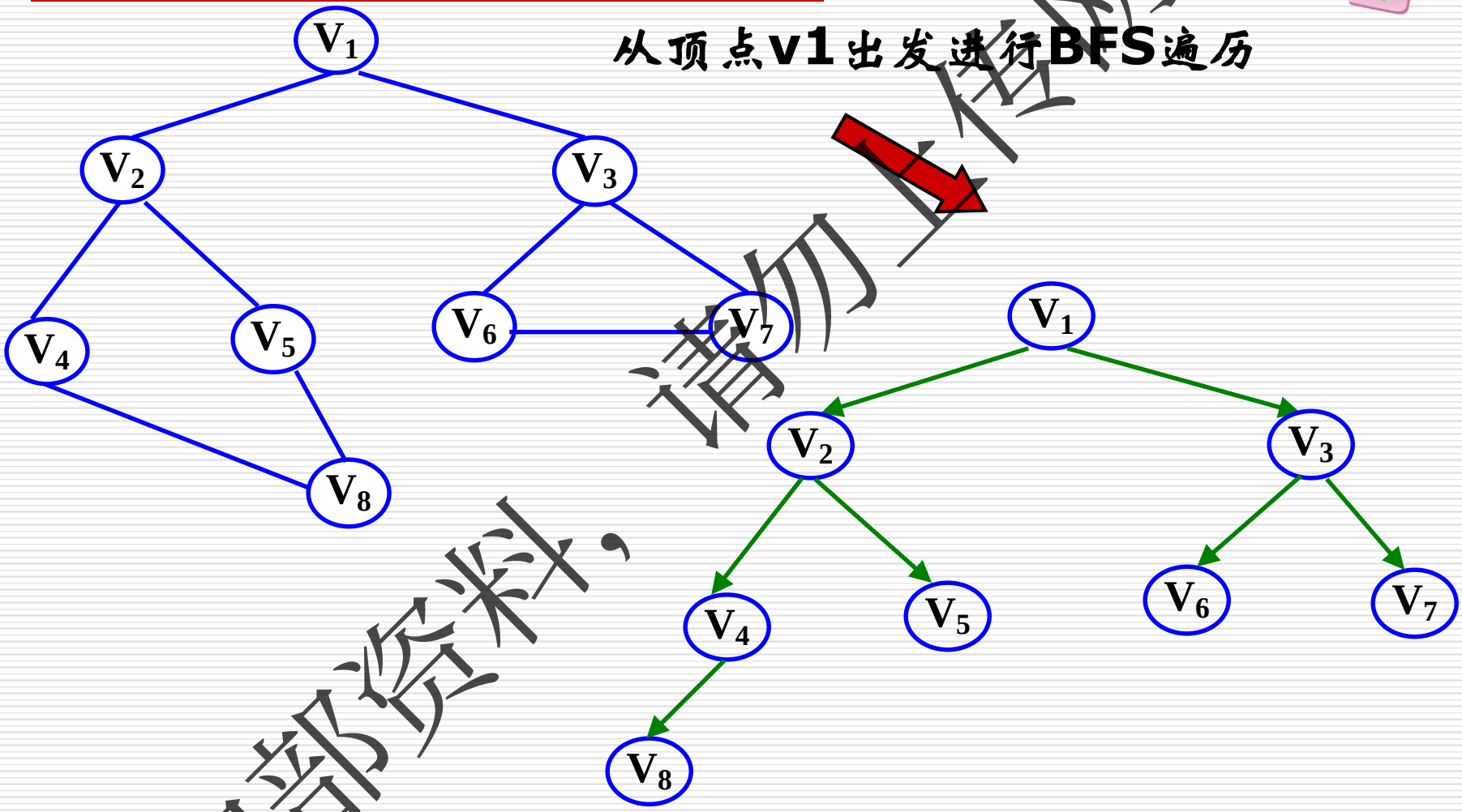
  ② 访问v的所有未被访问的邻接点w_1 ，w_2 ， …w_k ；

  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；

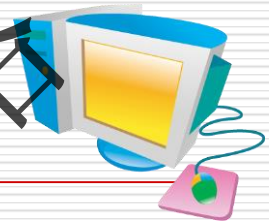
- 为实现③，需要保存在步骤②中访问的顶点，而且访问这些顶点的邻接点的顺序为：先保存的顶点，其邻接点先被访问。

6.5.2 广度优先搜索BFS

从顶点v1出发进行BFS遍历



算法6.7 广度优先搜索遍历连通图



算法步骤：

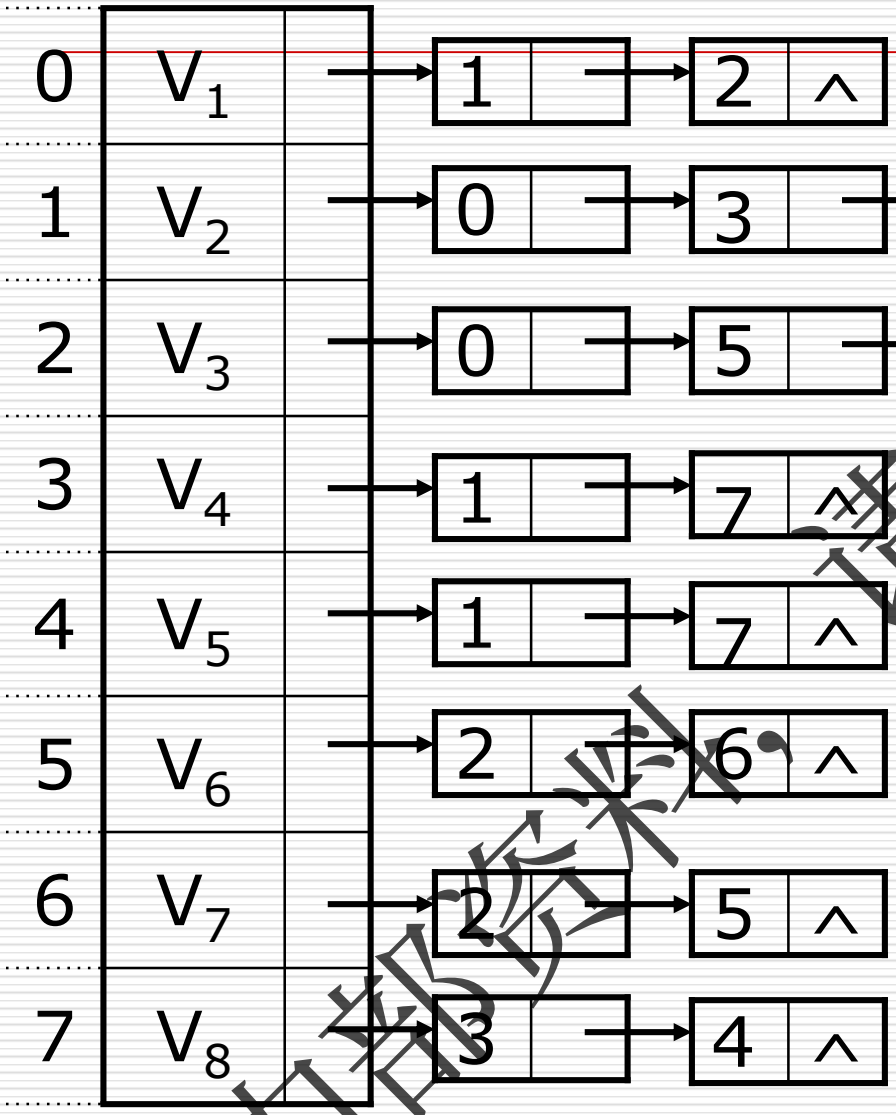
(1)从图中某个顶点v出发，访问v，并置visited[v]的值为true，然后将v进队。

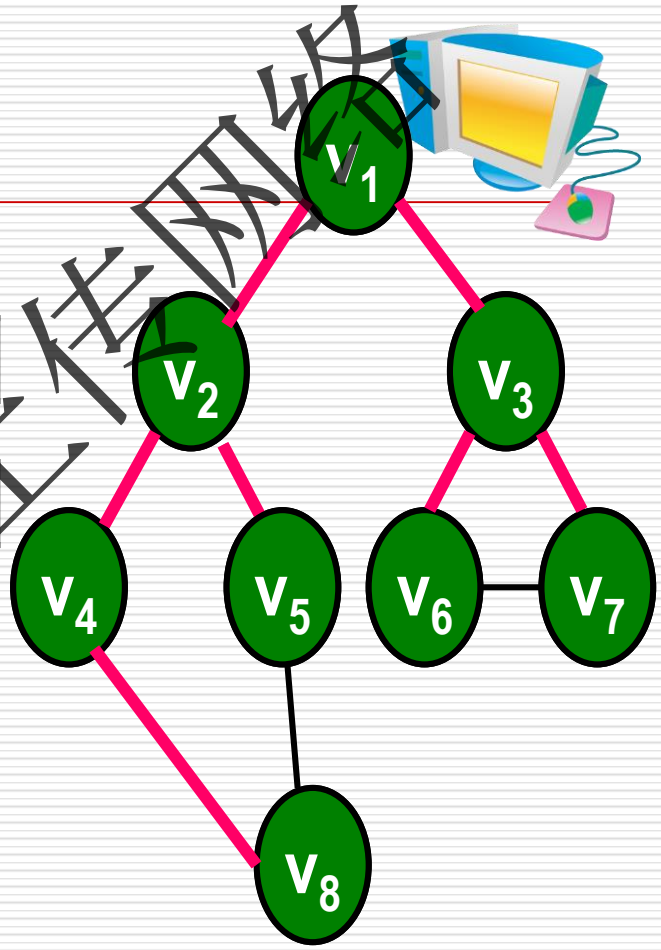
(2)只要队列不空，则重复下述操作：

- 队头顶点u出队；
- 依次检查u的所有邻接点w，如果visited[w]的值为false，则访问w，并置visited[w]的值为true，然后将w进队。

```
void BFS (Graph G, int v){//广度优先遍历连通图G
  cout<<v;  visited[v]=true;
  InitQueue(Q);      EnQueue(Q,v);
  while (!QueueEmpty(Q)) {
            DeQueue(Q,u);
    for(w取u的第一个邻接顶点; w存在 ; w取u的下一个邻接顶点)
                  if (!visited[w] ) {
                    cout<<w; visited[w]=true;
                    EnQueue(Q,w); }
    }//while
  } //BFS
```

示例及分析

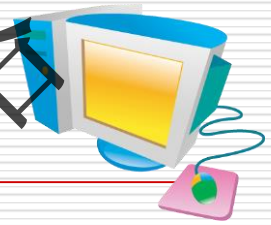




如图的广度优先搜索序列为：

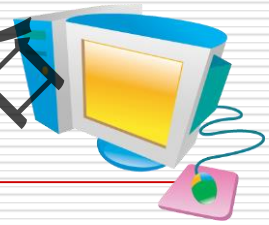
$v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow v_7 \rightarrow v_8$

BFS算法效率分析



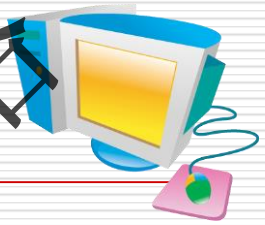
- 如果使用邻接矩阵，则BFS对于每一个被访问到的顶点，都要循环检测矩阵中的整整一行（ n 个元素），总的时间代价为$O(n^2)$。
- 用邻接表来表示图，虽然有 2e 个表结点，但只需扫描 e 个结点即可完成遍历，加上访问 n个头结点的时间，时间复杂度为O(n+e)。

DFS与BFS算法效率比较



- 时间复杂度只与存储结构（邻接矩阵或邻接表）有关，而与搜索路径无关。用邻接矩阵存储时，算法时间复杂度为$O(n^2)$；用邻接矩阵存储时，算法时间复杂度为O(n+e)。

6.6 图的应用

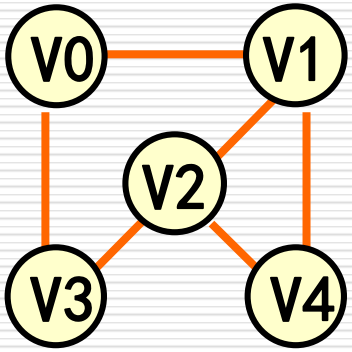


- 最小生成树
- 最短路径
- 拓扑排序
- 关键路径

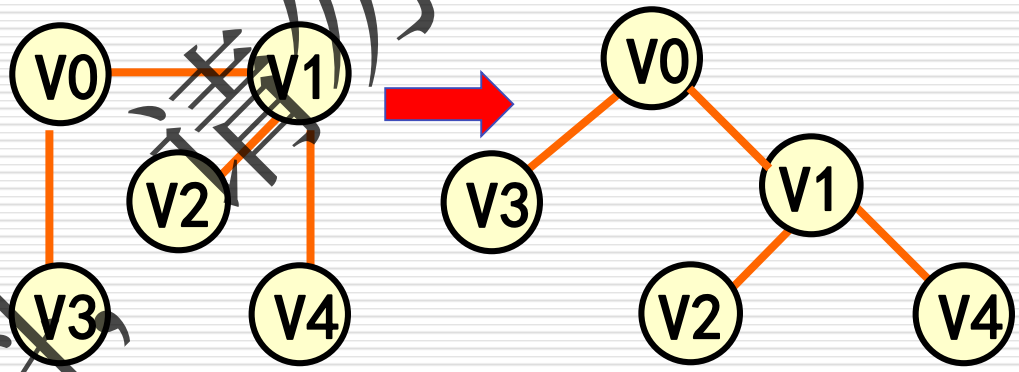
6.6.1 最小生成树

极小连通子图：该子图是G 的连通子图，在该子图中删除任何一条边，子图不再连通。

生成树：包含图G所有顶点的极小连通子图（n-1条边）。

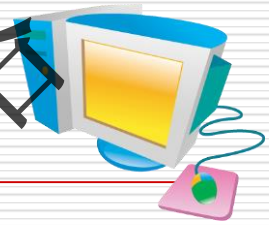


连通图 G1



G1的生成树

求最小生成树



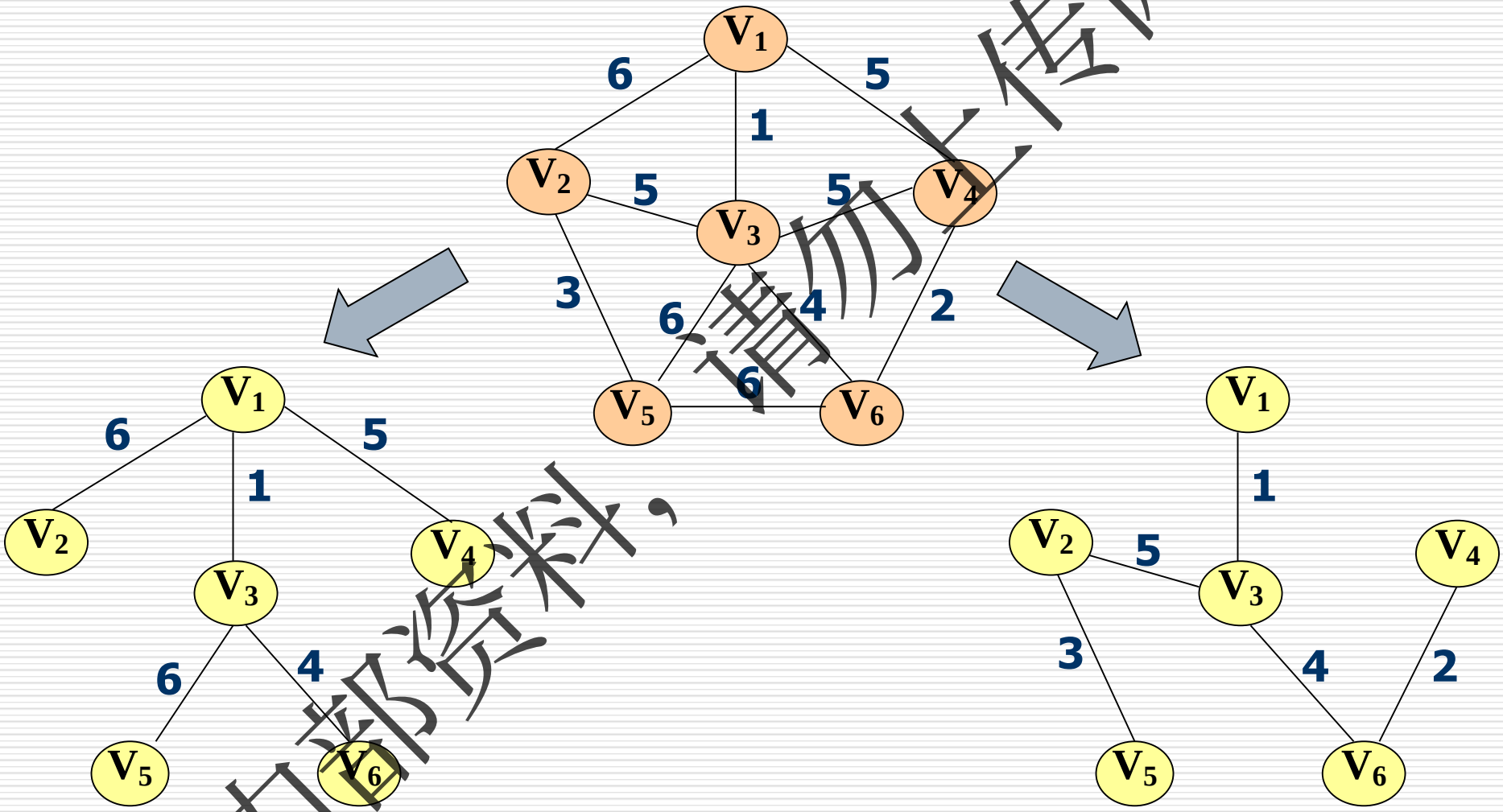
首先明确：

- 使用不同的遍历图的方法，可以得到不同的生成树
- 从不同的顶点出发，也可能得到不同的生成树。
- 按照生成树的定义，n 个顶点的连通网络的生成树有 n 个顶点、$n-1$ 条边。

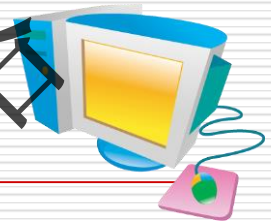
目标：

在网的多个生成树中，寻找一个各边权值之和最小的生成树

生成树的代价等于其边上的权值之和。

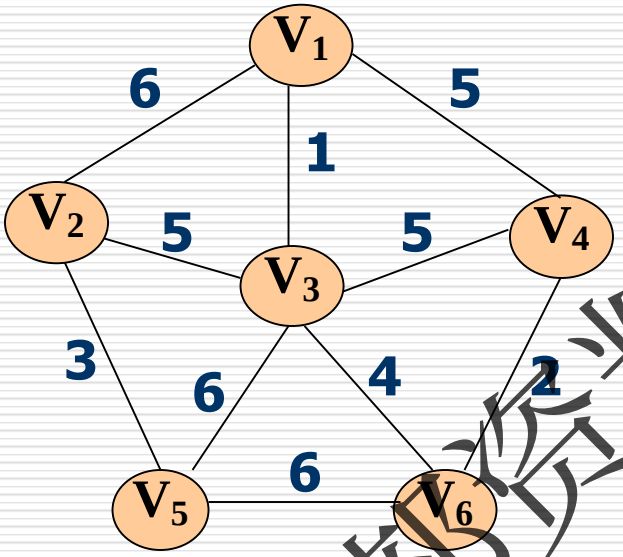


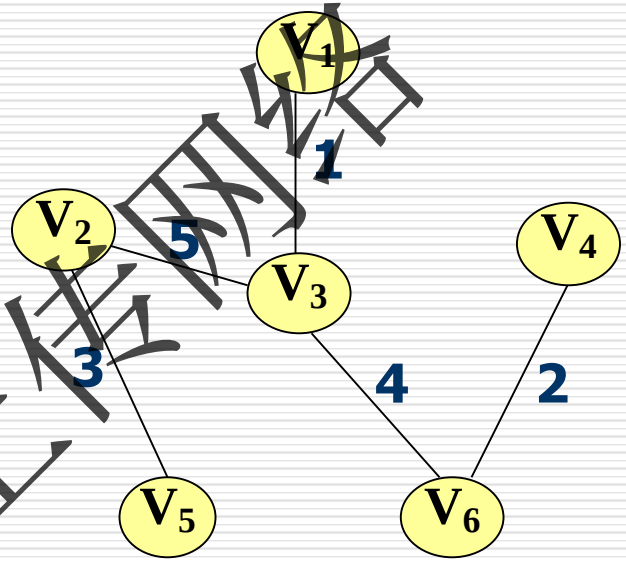
普里姆(Prim)算法



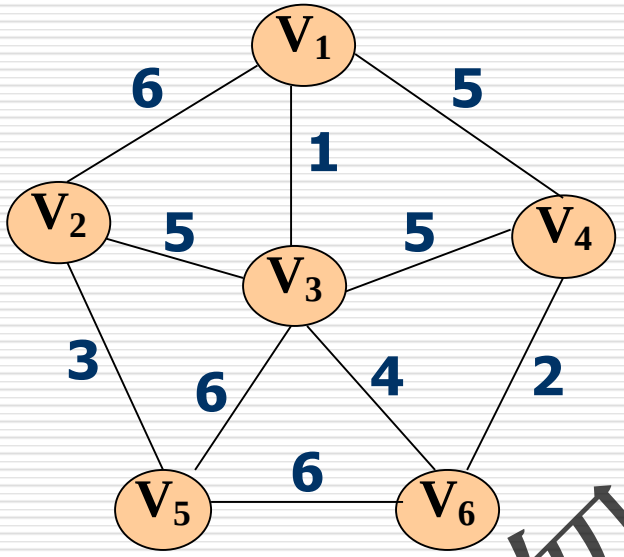
- 假设N=(V，E)是连通网，TE是N上最小生成树中边的集合。
- 算法从U={u_0}($u_0 \in V$)，TE={}开始，重复执行下述操作：
  - 在所有$u \in U$，$v \in V-U$的边(u，v)中找一条代价最小的边(u_0,v_0),将其并入集合TE，同时将v_0并入U集合。
  - 当U=V则结束，此时TE中必有n-1条边，则T=(V，{TE})为N的最小生成树。
- 普里姆算法构造最小生成树的过程是从一个顶点U={u_0}作初态，不断寻找与U中顶点相邻且代价最小的边的另一个顶点，扩充到U集合直至U=V为止。

普里姆算法求最小生成树

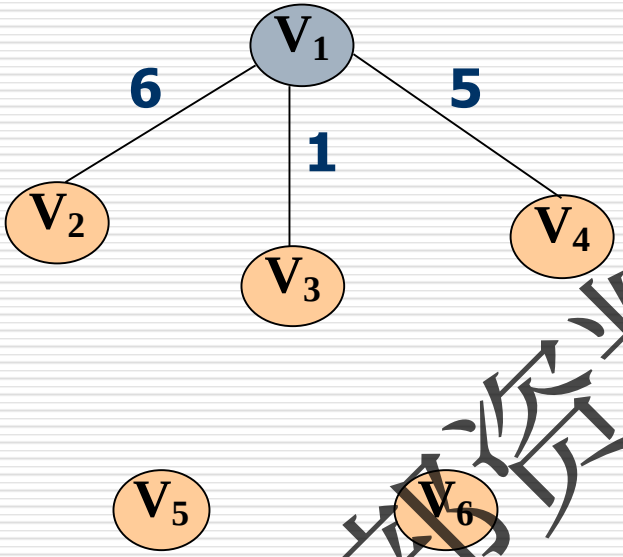




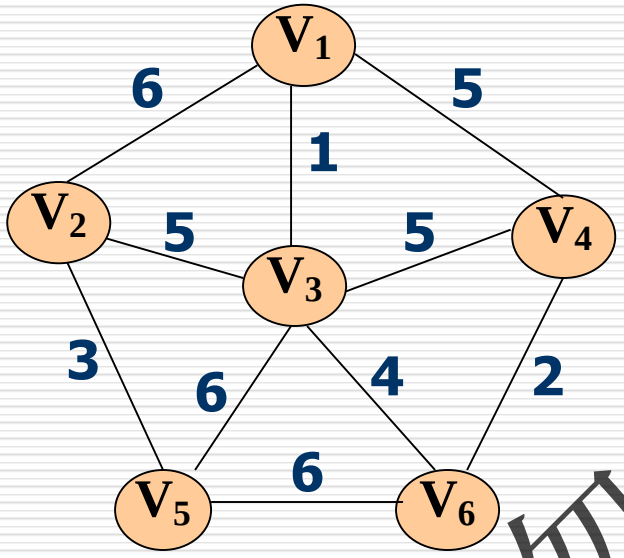
步骤	U	V-U
(0)	$\{V_1\}$	$\{V_2, V_3, V_4, V_5, V_6\}$
(1)	$\{V_1, V_3\}$	$\{V_2, V_4, V_5, V_6\}$
(2)	$\{V_1, V_3, V_6\}$	$\{V_2, V_4, V_5\}$
(3)	$\{V_1, V_3, V_6, V_4\}$	$\{V_2, V_5\}$
(4)	$\{V_1, V_3, V_6, V_4, V_2\}$	$\{V_5\}$
(5)	$\{V_1, V_3, V_6, V_4, V_2, V_5\}$	$\{\}$



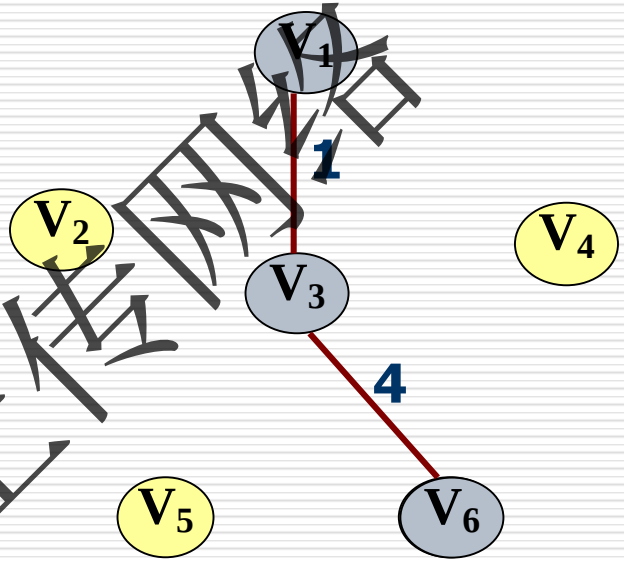
步骤1

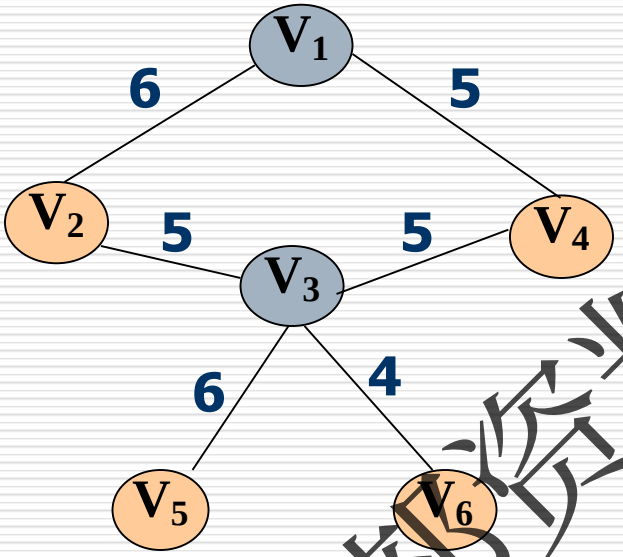


步骤	U	V-U
(0)	$\{V_1\}$	$\{V_2, V_3, V_4, V_5, V_6\}$
(1)	$\{V_1, V_3\}$	$\{V_2, V_4, V_5, V_6\}$



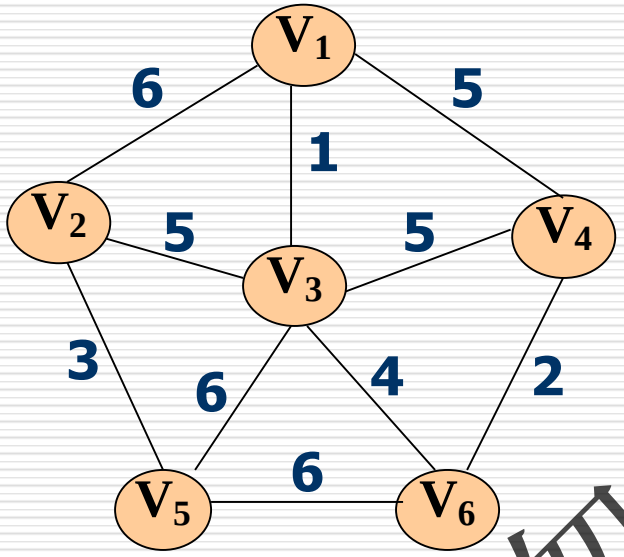
步骤2

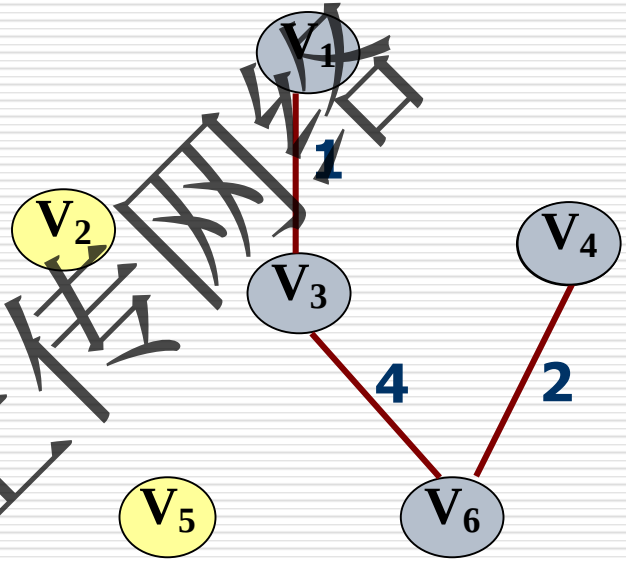


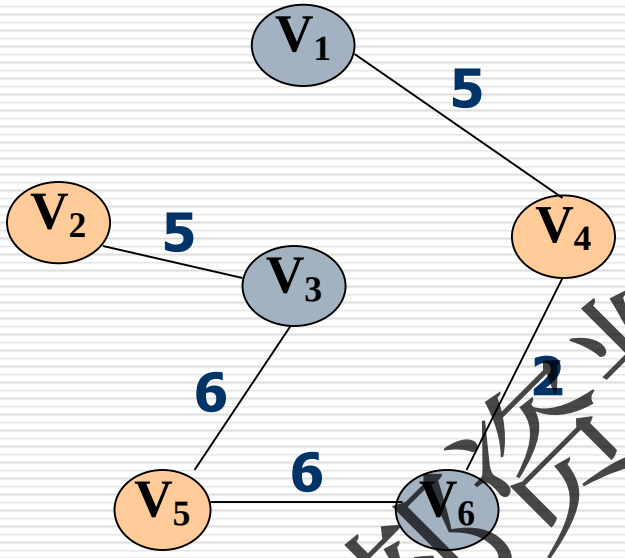


步骤	U	V-U
(0)	$\{V_1\}$	$\{V_2, V_3, V_4, V_5, V_6\}$
(1)	$\{V_1, V_3\}$	$\{V_2, V_4, V_5, V_6\}$
(2)	$\{V_1, V_3, V_6\}$	$\{V_2, V_4, V_5\}$

步骤3

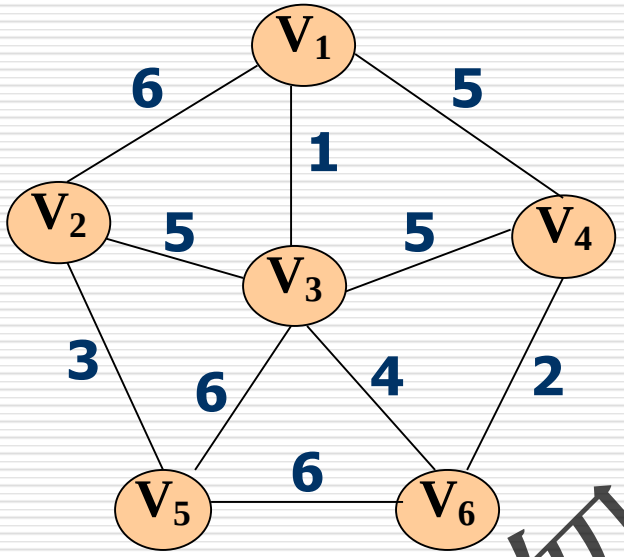


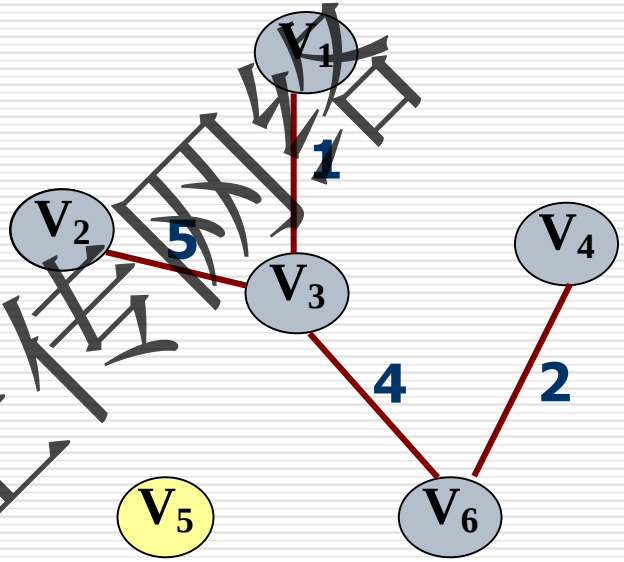


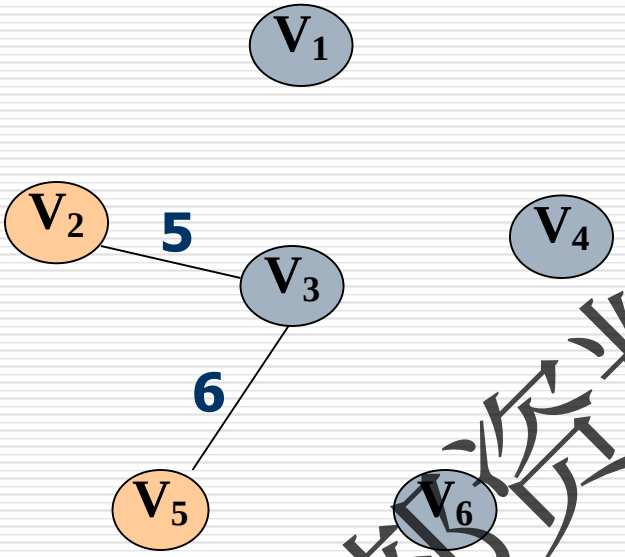


步骤	U	V-U
(0)	$\{V_1\}$	$\{V_2, V_3, V_4, V_5, V_6\}$
(1)	$\{V_1, V_3\}$	$\{V_2, V_4, V_5, V_6\}$
(2)	$\{V_1, V_3, V_6\}$	$\{V_2, V_4, V_5\}$
(3)	$\{V_1, V_3, V_6, V_4\}$	$\{V_2, V_5\}$

步骤4

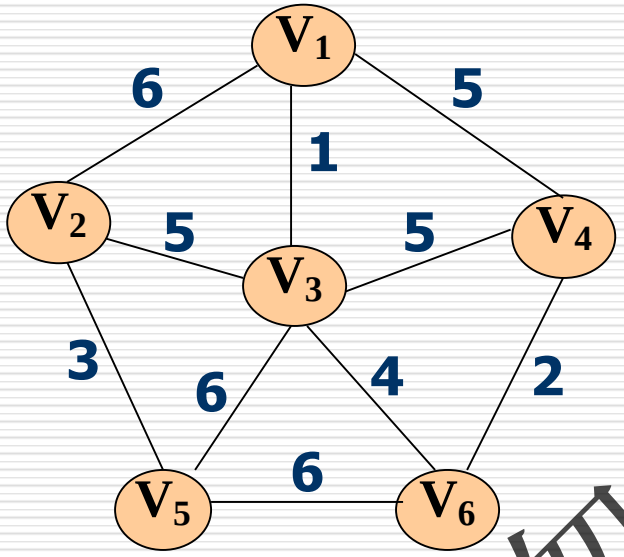


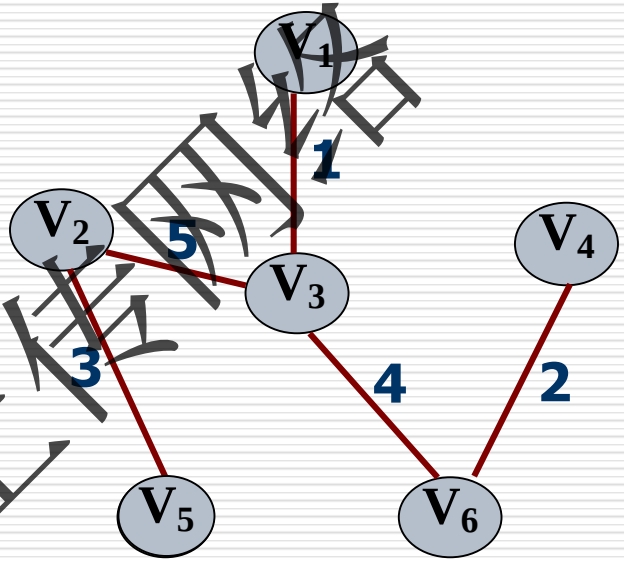


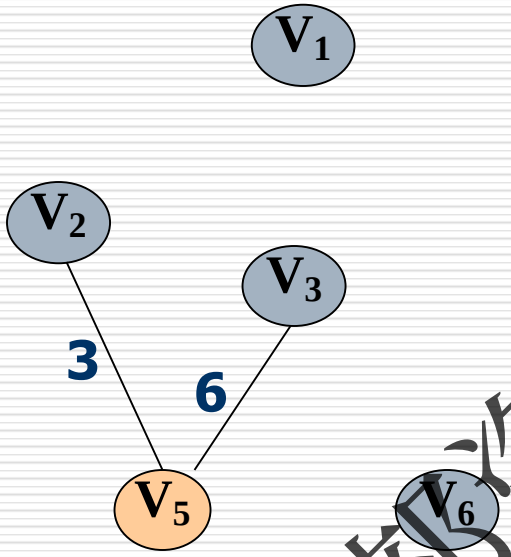


步骤	U	V-U
(0)	$\{V_1\}$	$\{V_2, V_3, V_4, V_5, V_6\}$
(1)	$\{V_1, V_3\}$	$\{V_2, V_4, V_5, V_6\}$
(2)	$\{V_1, V_3, V_6\}$	$\{V_2, V_4, V_5\}$
(3)	$\{V_1, V_3, V_6, V_4\}$	$\{V_2, V_5\}$
(4)	$\{V_1, V_3, V_6, V_4, V_2\}$	$\{V_5\}$

步骤5

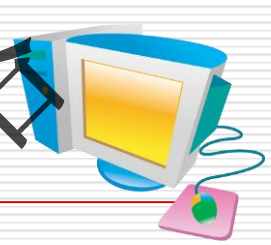






步骤	U	V-U
(0)	$\{V_1\}$	$\{V_2, V_3, V_4, V_5, V_6\}$
(1)	$\{V_1, V_3\}$	$\{V_2, V_4, V_5, V_6\}$
(2)	$\{V_1, V_3, V_6\}$	$\{V_2, V_4, V_5\}$
(3)	$\{V_1, V_3, V_6, V_4\}$	$\{V_2, V_5\}$
(4)	$\{V_1, V_3, V_6, V_4, V_2\}$	$\{V_5\}$
(5)	$\{V_1, V_3, V_6, V_4, V_2, V_5\}$	$\{\}$

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 {
  VerTexType adjvex;
  ArcType lowcost;
}closedge[MVNum];
```

顶点i与顶点k邻接

顶点k已经在U集合中

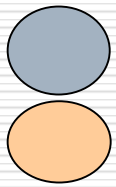
closedge[i].adjvex=vk

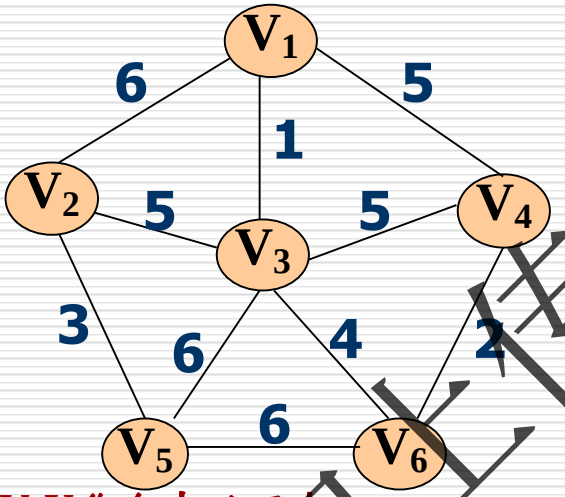
closedge[i].lowcost=0

顶点i加入U集合时

U集合的成员：

V-U集合的成员：





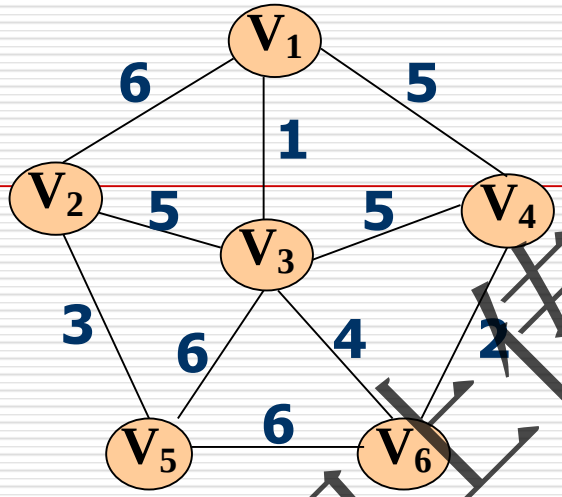
- 当U集合中加入一个新顶点时，V-U集合中的顶点到U的最小代价边可能会更新

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

closedge[1].adjvex=v1 .lowcost=6

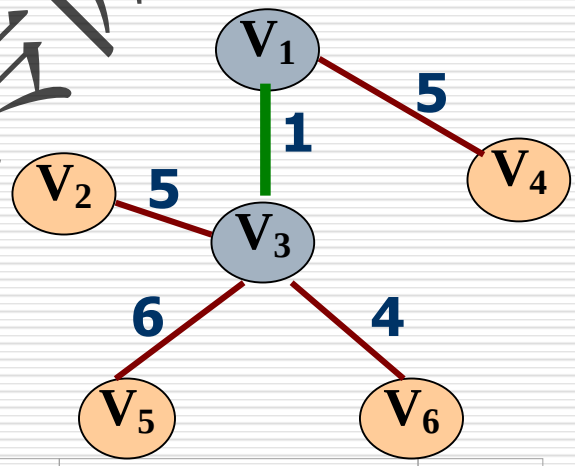
closedge[2].adjvex=v1 .lowcost=1

closedge[3].adjvex=v1 .lowcost=5



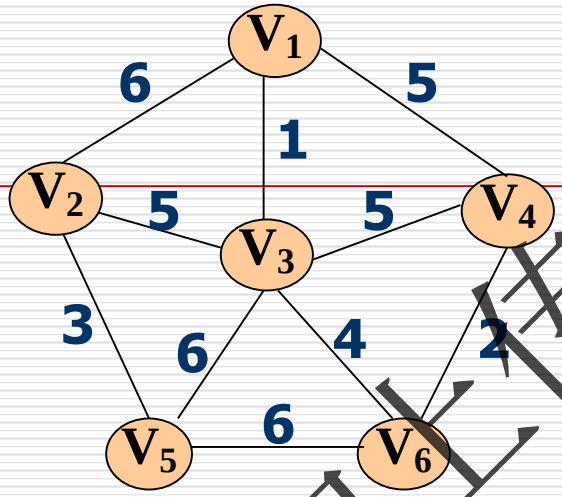
U集合的成员：

V-U集合的成员：



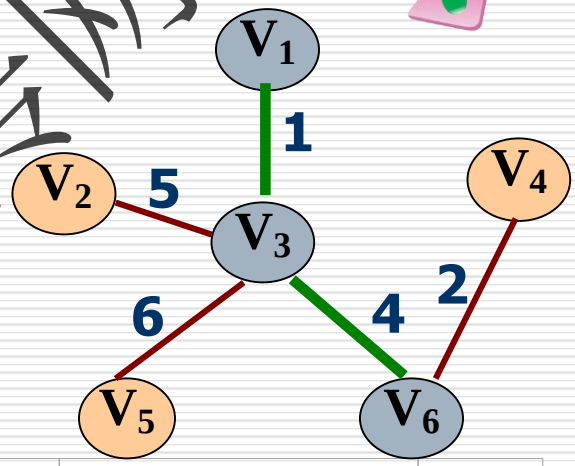
- 当U集合中加入一个新顶点时，V-U集合中的顶点到U的最小代价边可能会更新

顶点i / closedge	1	2	3	4	5	U	V-U	k
adjvex	v1	v1	v1			{v1}	{v2,v3,v4,v5,v6}	2
lowcost	6	1	5					
adjvex	v3	0	v1	v3	v3	{v1,v3}	{v2,v4,v5,v6}	5
lowcost	5		5	6	4			



U集合的成员：

V-U集合的成员：

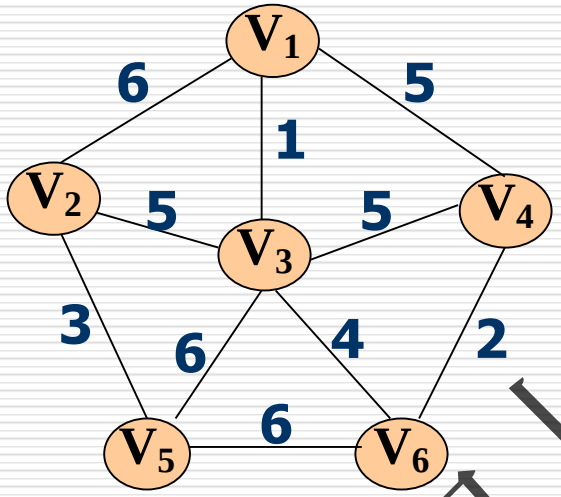


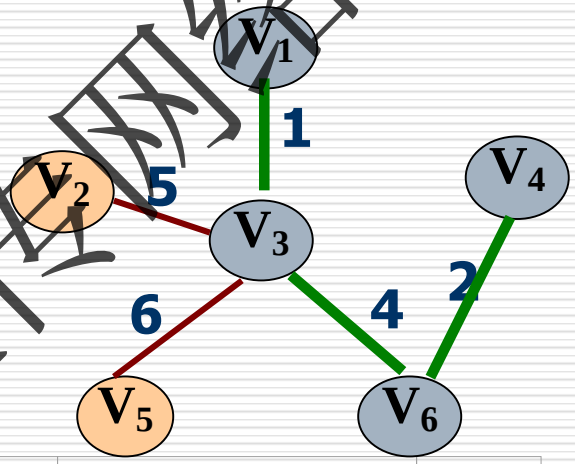
- 当U集合中加入一个新顶点时，V-U集合中的顶点到U的最小代价边可能会更新

顶点i / closedge	1	2	3	4	5	U	V-U	k
adjvex	v1	v1	v1			{v1}	{v2,v3,v4,v5,v6}	2
lowcost	6	1	5					
adjvex	v3		v1	v3	v3	{v1,v3}	{v2,v4,v5,v6}	5
lowcost	5	0	5	6	4			
adjvex	v3		v6	v3		{v1,v3,v6}	{v2,v4,v5 }	3
lowcost	5	0	2	6	0			

U集合的成员：

V-U集合的成员：

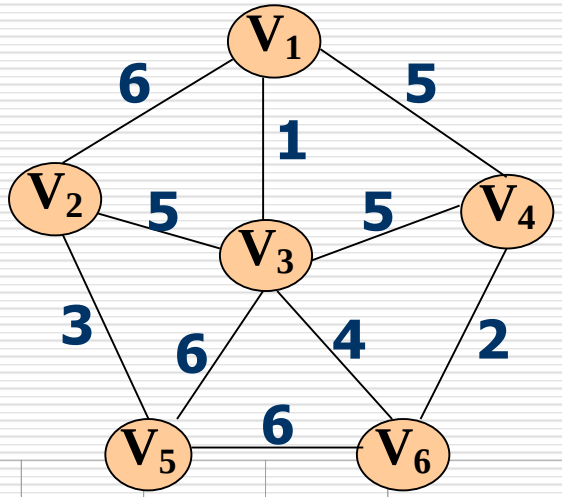


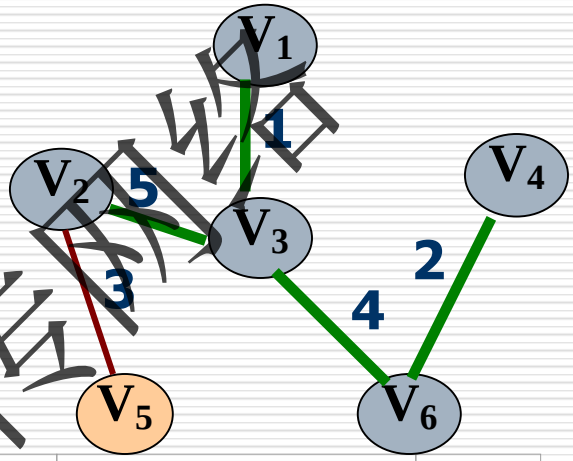


顶点i / closedge	1	2	3	4	5	U	V-U	k
adjvex lowcost	v1 6	v1 1	v1 5			{v1}	{v2,v3,v4,v5,v6}	2
adjvex lowcost	v3 5	0	v1 5	v3 6	v3 4	{v1,v3}	{v2,v4,v5,v6}	5
adjvex lowcost	v3 5	0	v6 2	v3 6	0	{v1,v3,v6}	{v2,v4,v5 }	3
adjvex lowcost	v3 5	0	0	v3 6	0	{v1,v3,v6,v4}	{v2,v5 }	1

U集合的成员：

V-U集合的成员：

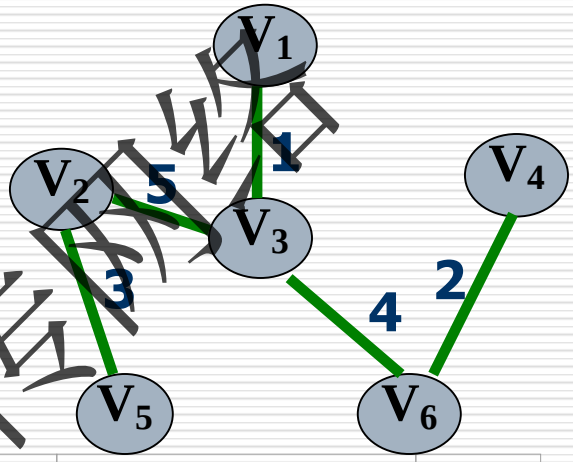




顶点i / closedge	1	2	3	4	5	U	V-U	k
adjvex lowcost	v1 6	v1 1	v1 5			{v1}	{v2,v3,v4,v5,v6}	2
adjvex lowcost	v3 5	0	v1 5	v3 6	v3 4	{v1,v3}	{v2,v4,v5,v6}	5
adjvex lowcost	v3 5	0	v6 2	v3 6	0	{v1,v3,v6}	{v2,v4,v5 }	3
adjvex lowcost	v3 5	0	0	v3 6	0	{v1,v3,v6,v4}	{v2,v5 }	1
adjvex lowcost	0	0	0	v2 3	0	{v1,v3,v6,v4,v2}	{v5 }	4

U集合的成员：

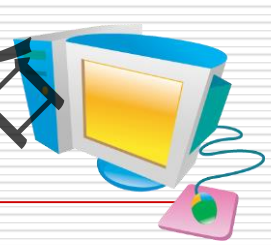
V-U集合的成员：



顶点i / closedge	1	2	3	4	5	U	V-U	k
adjvex lowcost	v1 6	v1 1	v1 5			{v1}	{v2,v3,v4,v5,v6}	2
adjvex lowcost	v3 5	0	v1 5	v3 6	v3 4	{v1,v3}	{v2,v4,v5,v6}	5
adjvex lowcost	v3 5	0	v6 2	v3 6	0	{v1,v3,v6}	{v2,v4,v5 }	3
adjvex lowcost	v3 5	0	0	v3 6	0	{v1,v3,v6,v4}	{v2,v5 }	1
adjvex lowcost	0	0	0	v2 3	0	{v1,v3,v6,v4,v2}	{v5 }	4
adjvex lowcost	0	0	0	0	0	{v1,v3,v6,v4,v2,v5}	{}	

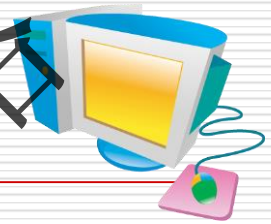
```
void MiniSpanTree_Prim (AMGraph G, VertexType u){
  k=LocateVex(G,u);
 for(j = 0; j < G.vexnum; ++j)  //辅助数组初始化
     if ( j != k ) closedge[j] = {u, G.arcs[k][j]};
  closedge[k].lowcost = 0;        //初始,U={u}
  for(i = 1; i < G.vexnum; ++i) {
      k = Min (closedge);  //求生成树的下一个顶点k
      cout << closedge[k].adjvex << G.vexs[k]; //输出生成树的边
      closedge[k].lowcost = 0;      //顶点k并入U集合
     for(j = 0; j < G.vexnum; ++j)
         if (G.arcs[k][j] < closedge[j].lowcost)
              closedge[j] = {G.vexs[k], G.arcs[k][j]};
  }//for
}// MiniSpanTree_Prim
```

Prim算法分析



Prim算法的时间复杂度是O（n^2），与网中边数无关，因此适用于求边稠密的网的最小生成树。

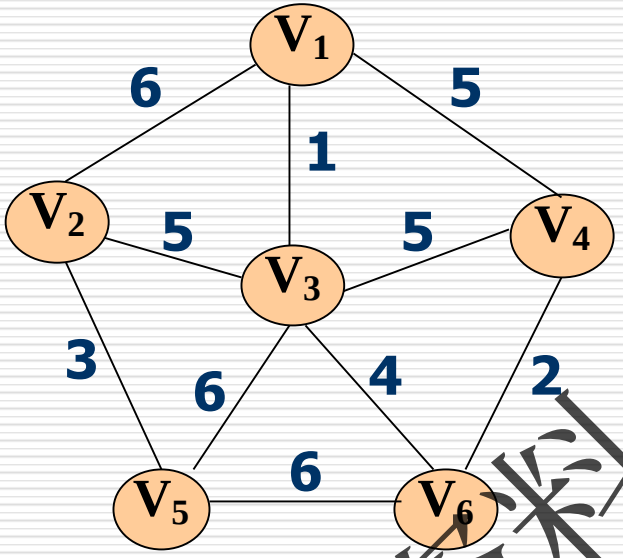
克鲁斯卡尔(Kruskal)算法

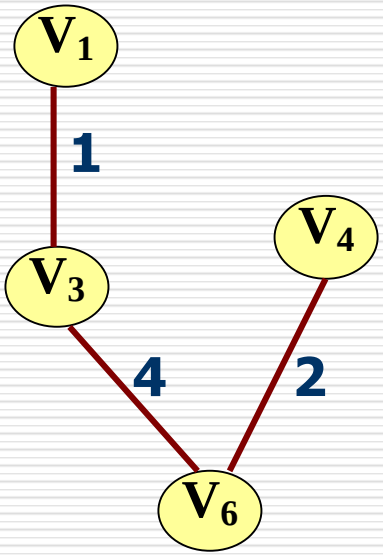


- 假设连通网N=(V，E)，则令最小生成树的初始状态为只有n个顶点而无边的非连通图T=(V，{})，图中每个顶点自成一个连通分量。

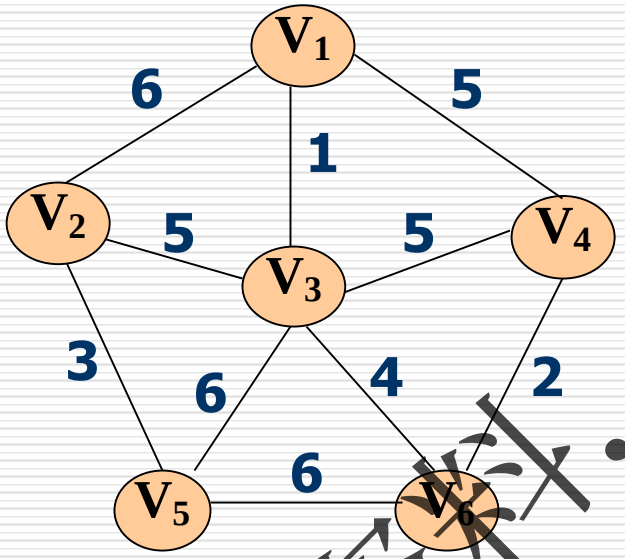
- 在E中选择代价最小的边，若该边依附的顶点落在T中不同的连通分量上，则将此边加入到T中，否则舍去此边而选择下一条代价最小的边。依次类推，直至T中所有顶点都在同一连通分量上为止。

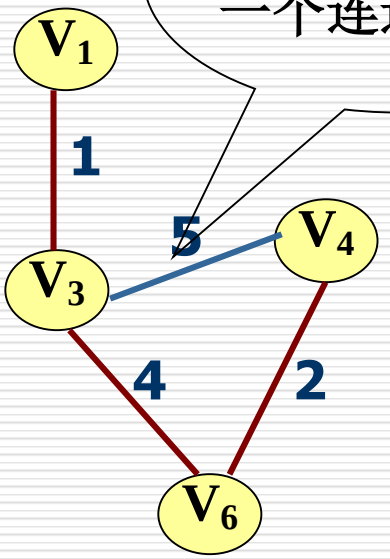
克鲁斯卡尔算法求最小生成树





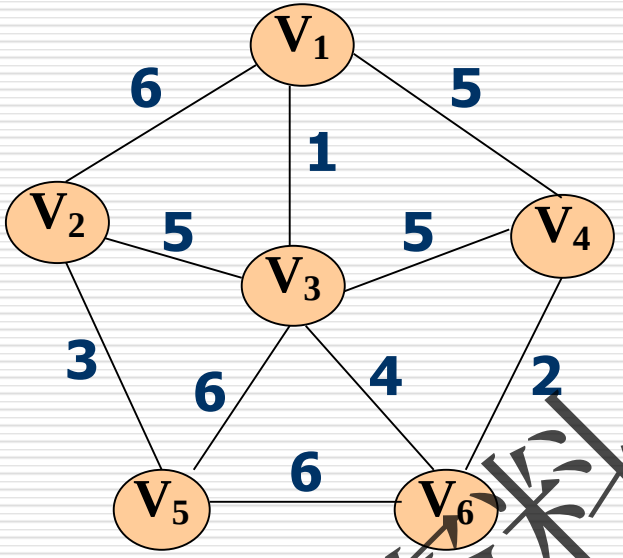
克鲁斯卡尔算法求最小生成树

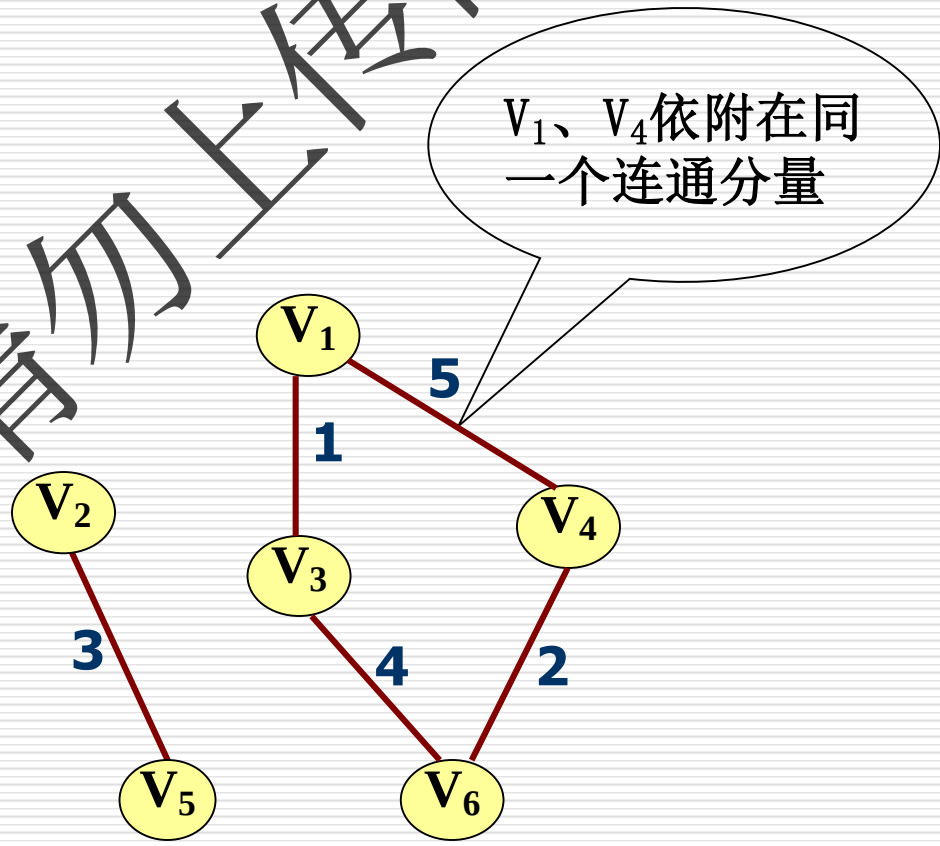




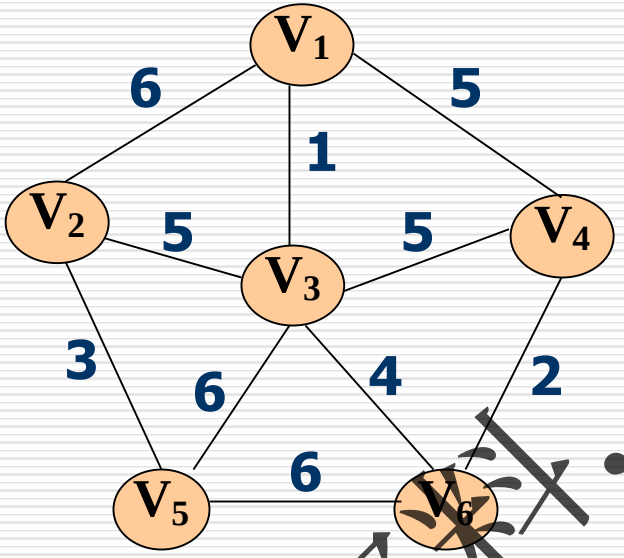
V_3、V_4依附在同一个连通分量

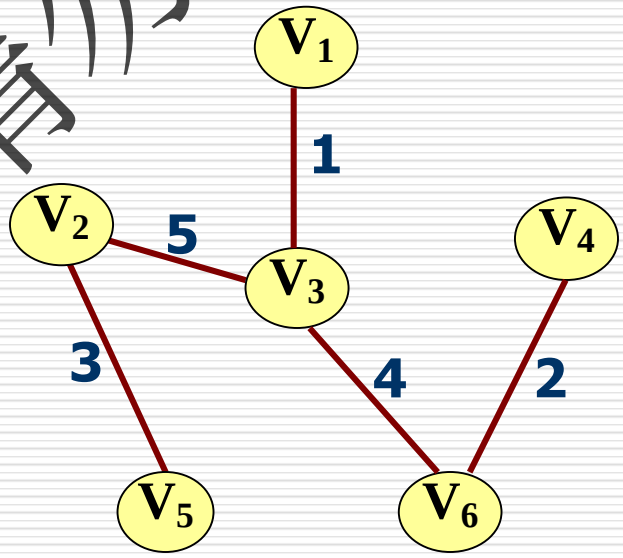
克鲁斯卡尔算法求最小生成树



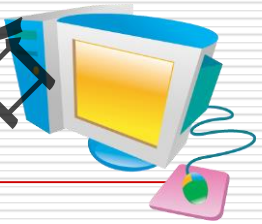


克鲁斯卡尔算法求最小生成树



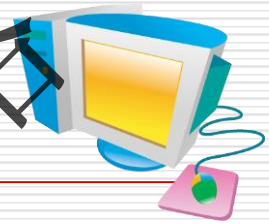


克鲁斯卡尔(Kruskal)算法



- 从上述过程可知，实现克鲁斯卡尔(Kruskal)算法时，要解决以下两个问题：
  - 如何选择代价最小的边；
  - 如何判定边所关联的两个顶点是否在同一个连通分量中

克鲁斯卡尔(Kruskal)算法实现

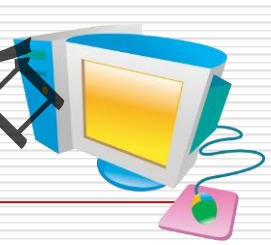


存储结构

（1）结构体数组Edge：存储边的信息，包括边的两个顶点信息和权值：

```
//辅助数组Edge的定义
typedef struct
{  VerTexType Head;     //边的始点
   VerTexType  Tail;    //边的终点
   ArcType lowcost;     //边上的权值
} Edge[arcnum];
```

克鲁斯卡尔(Kruskal)算法实现



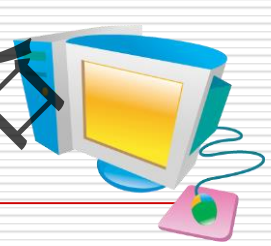
存储结构

（2）Vexset[i]:标识各个顶点所属的连通分量。对每个顶点vi ∈V，在辅助数组中存在一个相应元素Vexset[i]表示该顶点所在的连通分量。初始时Vexset[i]=i，表示各顶点自成一个连通分量。

//辅助数组Vexset的定义

typedef int Vexset[MVNum];

克鲁斯卡尔(Kruskal)算法步骤



（1）将数组Edge中的元素按权值从小到大排序

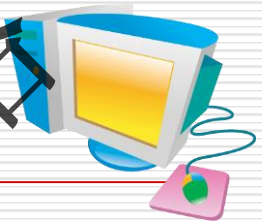
（2）依次查看数组Edge中的边，循环执行以下操作：

- 依次从排好序的数组Edge中选出一条边（v_1,v_2）；
- 在Vexset中分别查找v1和v2所在的连通分量vs1的vs2，进行判断：

➢ 如果vs1和vs2不等，表明所选的两个顶点分属不同的连通分量，输出此边，并合并vs1和vs2两个连通分量；

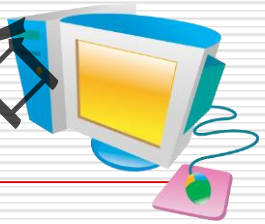
➢ 如果vs1和vs2相等，表明所选的两个顶点属于同一个连通分量，舍云此边而选择下一条权值最小的边。

克鲁斯卡尔(Kruskal)算法描述



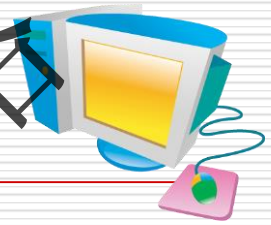
```
void MiniSpan_Kruskal(AMGraph G)
{ Sort(Edge);
   for(i=0; i<G.vexnum; ++i)
      Vexset[i]=i;
   for(i=0; i<G.arcnum; ++i)
  {    v1=LocateVex(G,Edge[i].Head);
       v2=LocateVex(G,Edge[i].Tail);
       vs1=Vexset[v1];
       vs2=Vexset[v2];
```

克鲁斯卡尔(Kruskal)算法描述



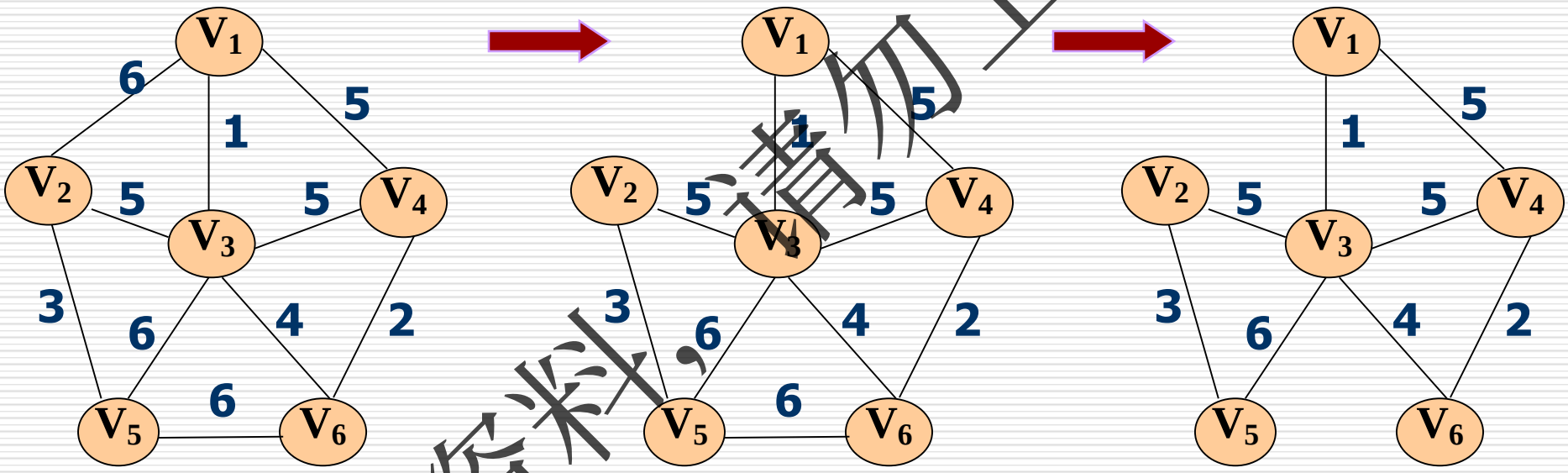
```
    if(vs1!=vs2)
    {
        cout<<Edge[i].Head<<Edge[i].Tail;
        for(j=0;j<G.vexnum; ++j)
          if(Vexset[j]==vs2) Vexset[j]=vs1;
           //集合编号为vs2的都改为vs1
    }
  }
}
```

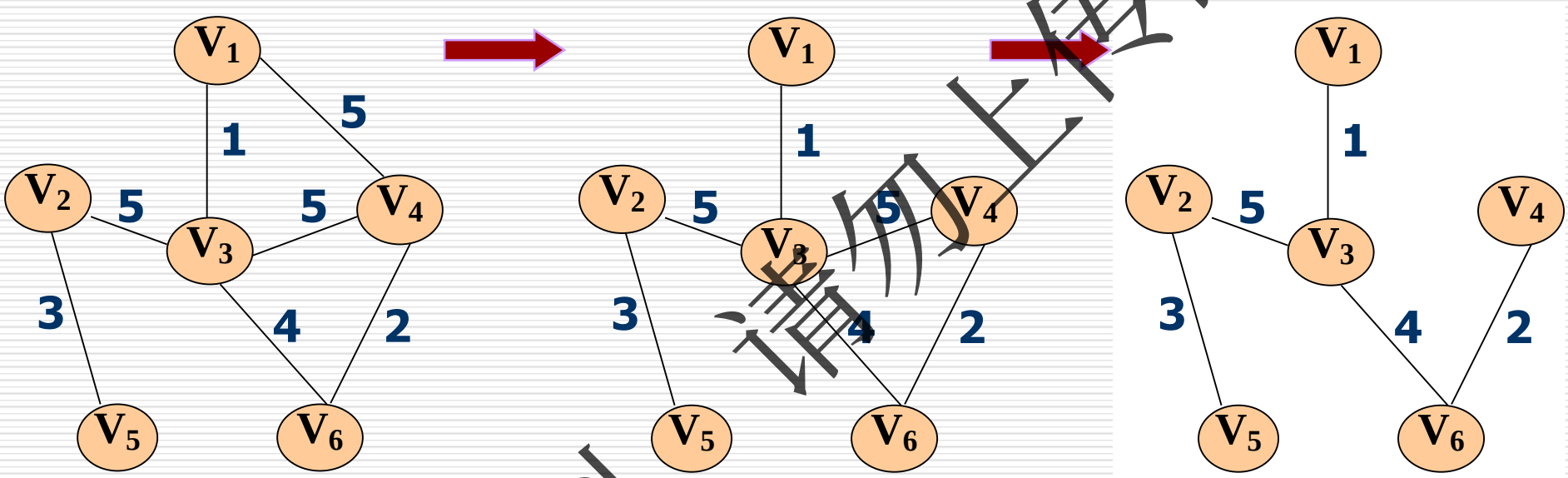
Kruskal算法分析



Kruskal算法的时间复杂度是O（$elog_2e$），与网中边数相关，因此适用于求边稀疏的网的最小生成树。

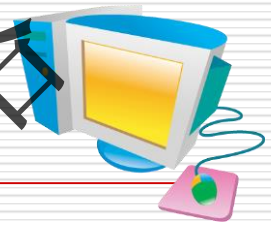
破圈法求最小代价生成树





破圈法思路简单，但实现时比普里姆算法和克鲁斯卡尔算法复杂。

6.6.2 最短路径



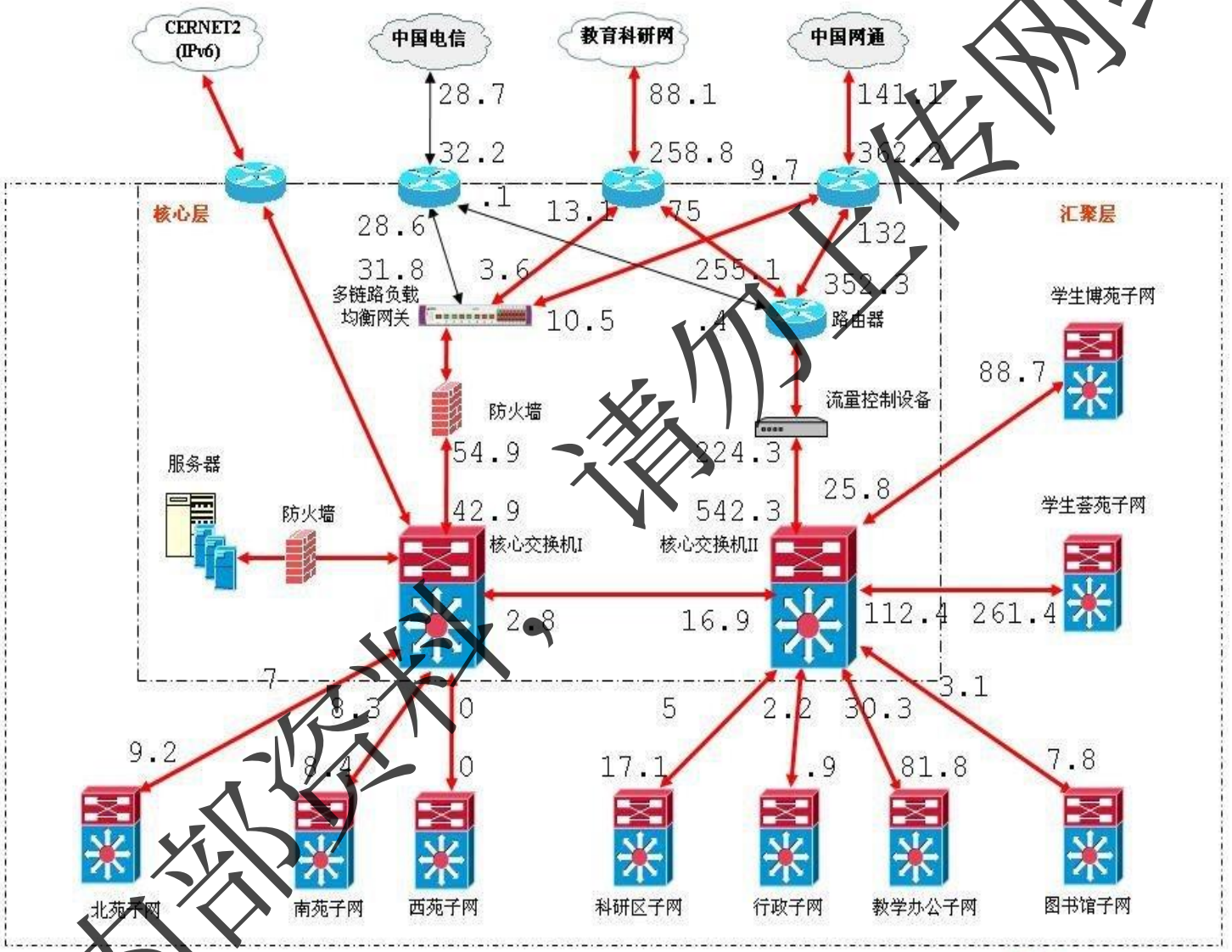
1. 问题的提出

- 交通咨询系统、通讯网、计算机网络需要寻找两结点间最短路径
- 交通咨询系统：A地点到B地点的最短路径
- 计算机网：A到B沿最短路径传送

2. 最短路径及其长度

- 路径长度：路径上的边数
  路径上边的权值之和
- 最短路径：两结点间权值之和最小的路径

最短路算法典型应用--计算机网络路由



最短路算法典型应用—机器人探路

最短路算法典型应用—游戏开发

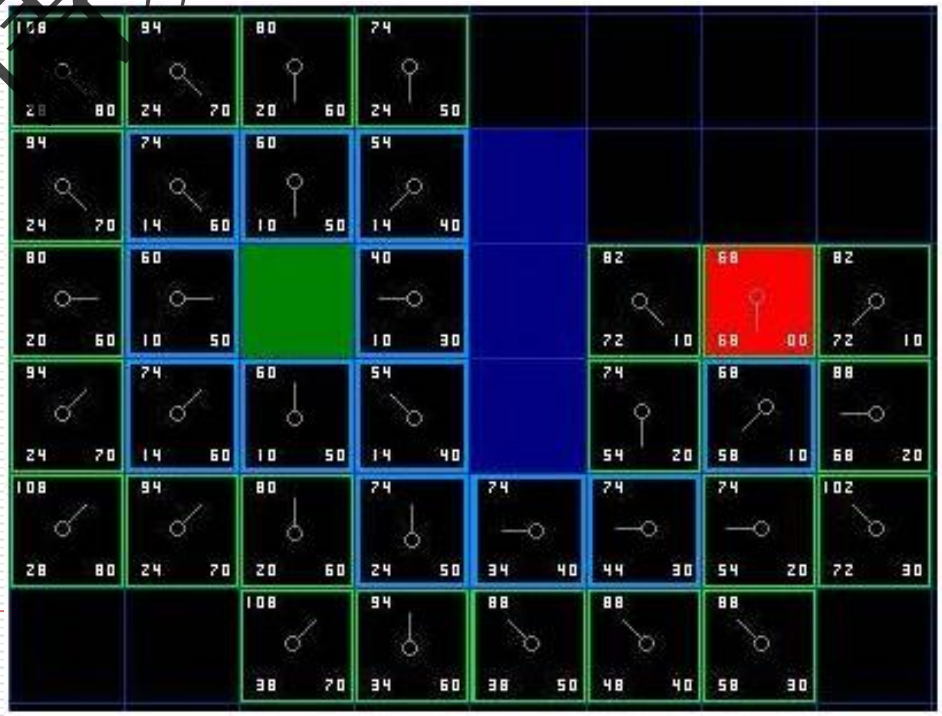
Dijistra算法的改进—A*算法（静态环境）

+估价值

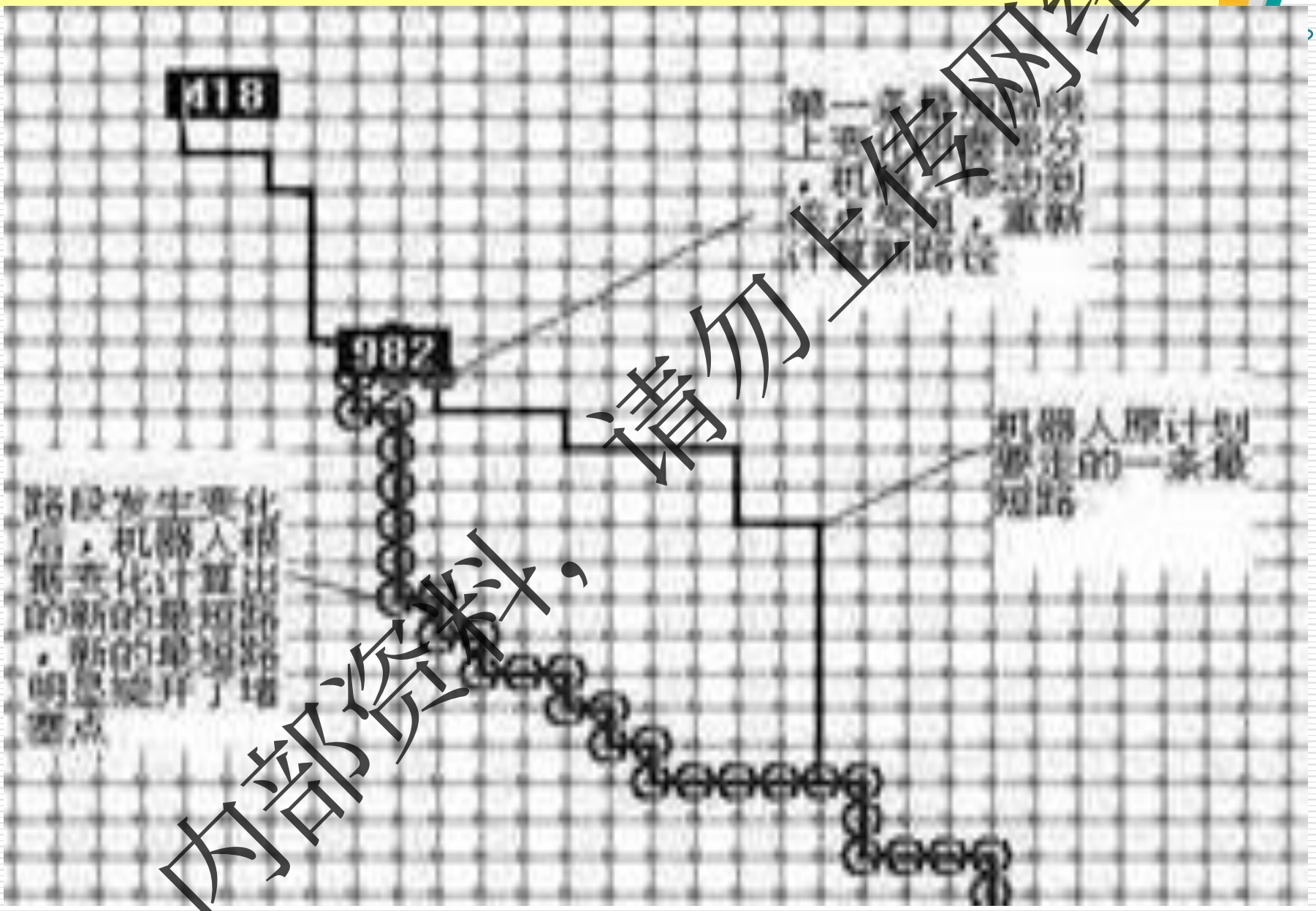
Dijistra算法 → A*算法

估价值=0

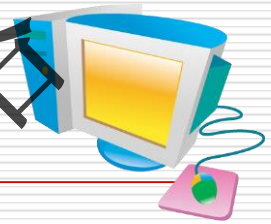
静态环境求解最短路
最有效的方法



Dijistra算法的改进—D*算法（动态环境）



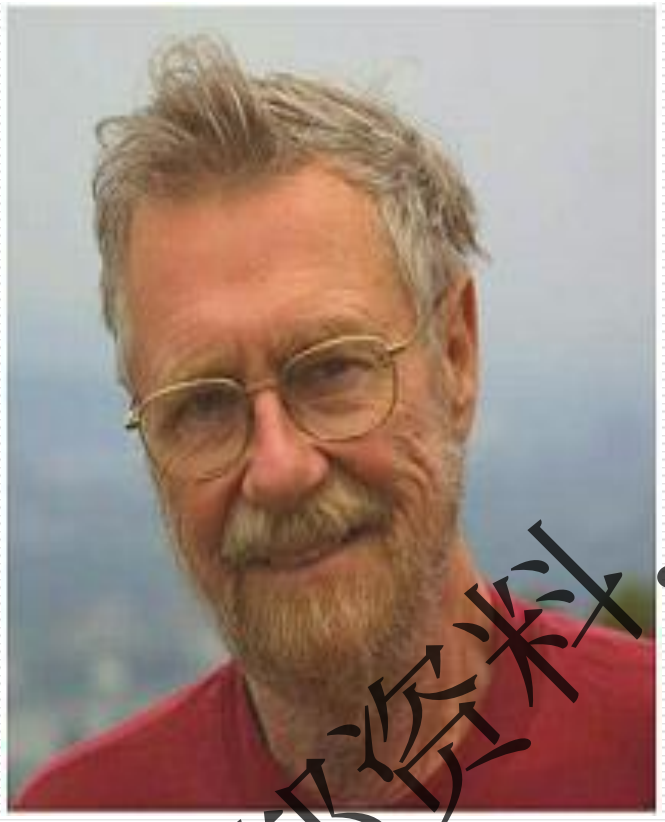
两种经典的最短路径算法



（1）单源点最短路径—Dijkstra（迪杰斯特拉）算法

（2）所有顶点间的最短路径—Floyd（弗洛伊德）算法

Dijkstra（迪杰斯特拉）算法



迪杰斯特拉

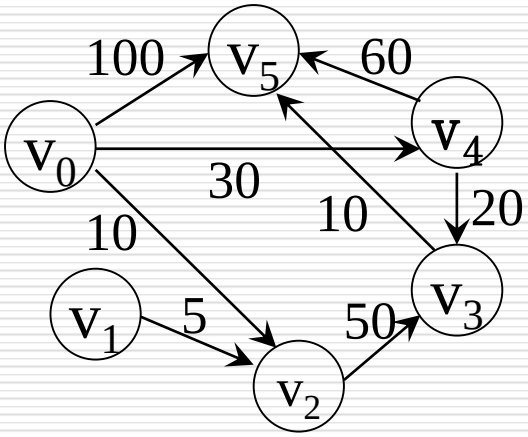
迪杰斯特拉，荷兰计算机科学家，1972年图灵奖得主，于1959 年提出迪杰斯特拉算法。

Floyd（弗洛伊德）算法

罗伯特·弗洛伊德

罗伯特·弗洛伊德（1936－2001），计算机科学家，斯坦福大学教授，1978年获图灵奖，1962年提出弗洛伊德算法。

例

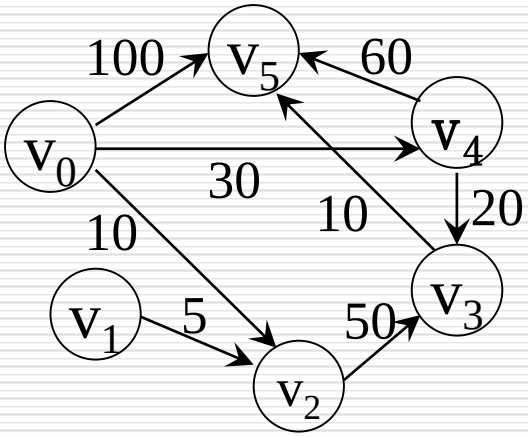


前提： 边上权值非负

过程： 按路径长度递增的次序产生最短路径。

源　点	终　点	最短路径	路径长度
v_0	v_2	(v_0, v_2)	10
	v_4	(v_0, v_4)	30
	v_3	(v_0, v_4, v_3)	50
	v_5	(v_0, v_4, v_3, v_5)	60
	v_1	无	∞

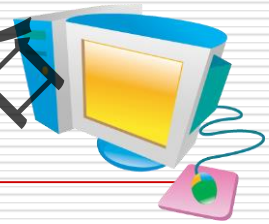
例



	0	1	2	3	4	5
0	∞	∞	10	∞	30	100
1	∞	∞	5	∞	∞	∞
2	∞	∞	∞	50	∞	∞
3	∞	∞	∞	∞	∞	10
4	∞	∞	∞	20	∞	60
5	∞	∞	∞	∞	∞	∞

源　点	终　点	最短路径	路径长度
v_0	v_2	(v_0, v_2)	10
	v_4	(v_0, v_4)	30
	v_3	(v_0, v_4, v_3)	50
	v_5	(v_0, v_4, v_3, v_5)	60
	v_1	无	∞

迪杰斯特拉算法的求解过程



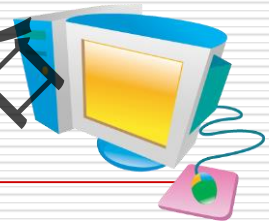
对于网N=(V,E),将N中的顶点分成两组：

第一组S：已求出的最短路径的顶点集合（初始时只包含源点v_0）。

第二组V-S：尚未求出最短路径的顶点集合（初始时为V-{v_0}）。

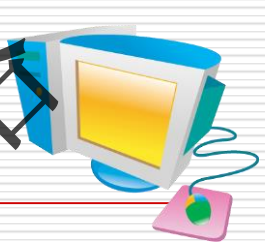
算法将按各顶点与v_0间最短路径长度递增的次序，逐个将V-S中的顶点加入到集合S中去。在这个过程中，总保持从v_0到集合S中各顶点的路径长度始终不大于到集合V-S中各顶点的路径长度。

存储结构（顶点个数为n）



- 主：邻接矩阵G.arc[n][n] (或者邻接表)
- 辅：
  - 数组S[n]：记录相应顶点是否已确定最短路径
  - 数组D[n]：记录源点到相应顶点路径长度
  - 数组Path[n]：记录相应顶点的前驱顶点

算法思想



（1）初始化：

将源点v0加到S中，即S[v0] = true；
将v0到各个终点的最短路径长度初始化为权值，即D[i] = G.arcs[v0][vi]，(vi∈V – S)；
如果v0和顶点vi之间有弧，则将vi的前驱置为v0，即Path[i] = v0，否则Path[i] = –1。

	0	1	2	3	4	5
0	∞	∞	10	∞	30	100
1	∞	∞	5	∞	∞	∞
2	∞	∞	∞	50	∞	∞
3	∞	∞	∞	∞	∞	10
4	∞	∞	∞	20	∞	60
5	∞	∞	∞	∞	∞	∞

	$v=0$	$v=1$	$v=2$	$v=3$	$v=4$	$v=5$
S	true	false	false	false	false	false
D	0	∞	10	∞	30	100
$Path$	−1	−1	0	−1	0	0

算法思想

（2） 循环n-1次，执行以下操作：

①选择下一条最短路径的终点vk，使得

D[k] = Min{D[i]|vi∈V－S}

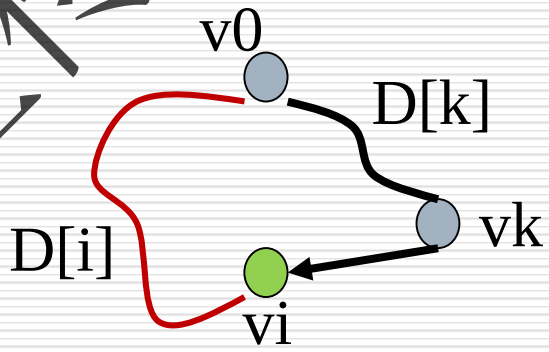
②将vk加到S中，即S[vk] = true。

③ 更新从v0出发到集合V－S上任一顶点的最短路径的长度，同时更改vi的前驱为vk。

若S[i]=false 且

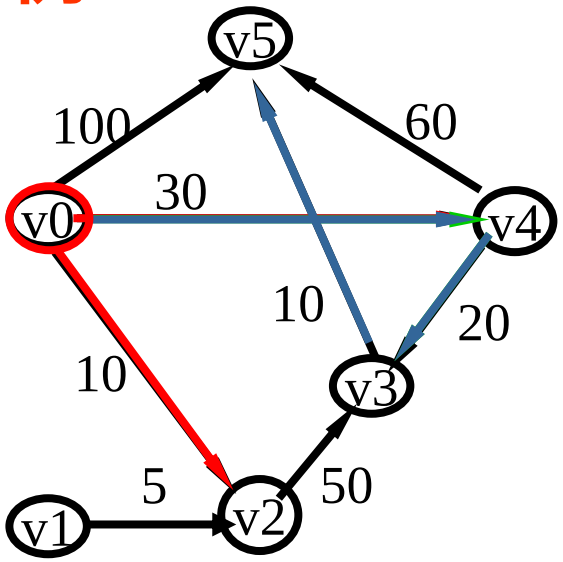
D[k]+G.arcs[k][i]<D[i]，则

D[i]=D[k]+ G.arcs[k][i]; Path [i]=k;。



【算法执行过程】

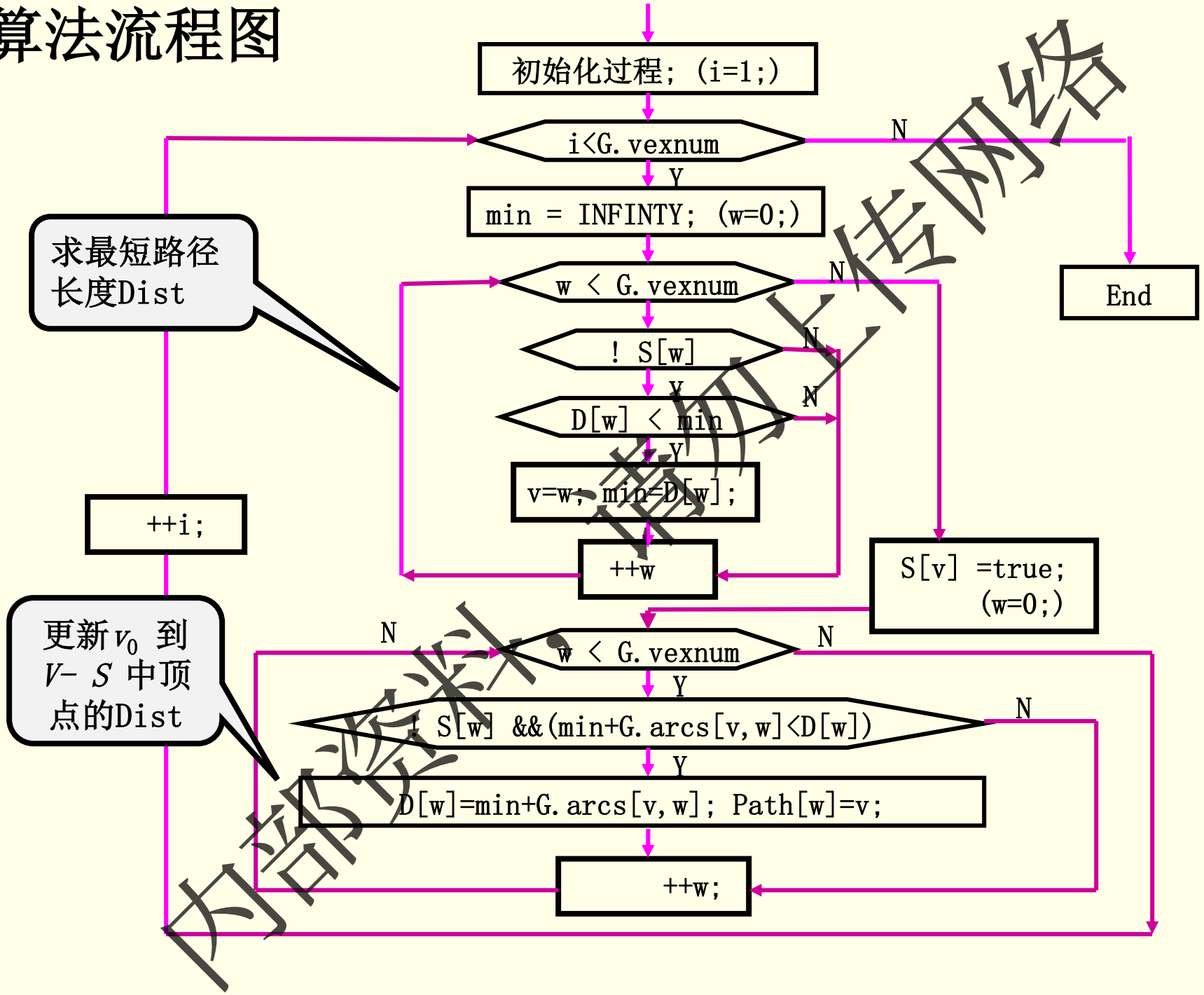
例



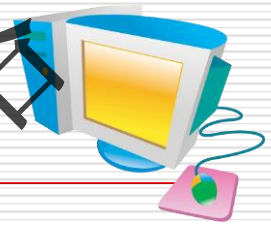
	0	1	2	3	4	5
0	∞	∞	10	∞	30	100
1	∞	∞	5	∞	∞	∞
2	∞	∞	∞	50	∞	∞
3	∞	∞	∞	∞	∞	10
4	∞	∞	∞	20	∞	60
5	∞	∞	∞	∞	∞	∞

终点	i=1	i=2	i=3	i=4
v_1	∞	∞	∞	∞
v_2	10 $\{v_0,v_2\}$			10 $\{v_0,v_2\}$
v_3	∞	60 $\{v_0,v_2,v_3\}$	50 $\{v_0,v_4,v_3\}$	50 $\{v_0,v_4,v_3\}$
v_4	30 $\{v_0,v_4\}$	30 $\{v_0,v_4\}$		30 $\{v_0,v_4\}$
v_5	100 $\{v_0,v_5\}$	100 $\{v_0,v_5\}$	90 $\{v_0,v_4,v_5\}$	60 $\{v_0,v_4,v_3,v_5\}$
v_k	v_2	v_4	v_3	v_5
Path		Path[3]=2	Path[3]=4 Path[5]=4	Path[5]=3
S	$\{v_0,v_2\}$	$\{v_0,v_2,v_4\}$	$\{v_0,v_2,v_4,v_3\}$	$\{v_0,v_2,v_4,v_3,v_5\}$

算法流程图



【算法描述】



```
void ShortestPath_DIJ(AMGraph G, int v0){
 //用Dijkstra算法求有向网G的v0顶点到其余顶点的最短路径
 n=G.vexnum;                          //n为G中顶点的个数
 for(v = 0; v<n; ++v){                //n个顶点依次初始化
   S[v] = false;                      //S初始为空集
   D[v] = G.arcs[v0][v]; //v0到各终点的最短路径长度初始化
   if(D[v]< MaxInt)  Path [v]=v0; //v0和v之间有弧，将v的
前驱置为v0
    else Path [v]=-1; //无弧，则将v的前驱置为-1
   }//for
   S[v0]=true;                        //将v0加入S
   D[v0]=0;                          //源点到源点的距离为0
```

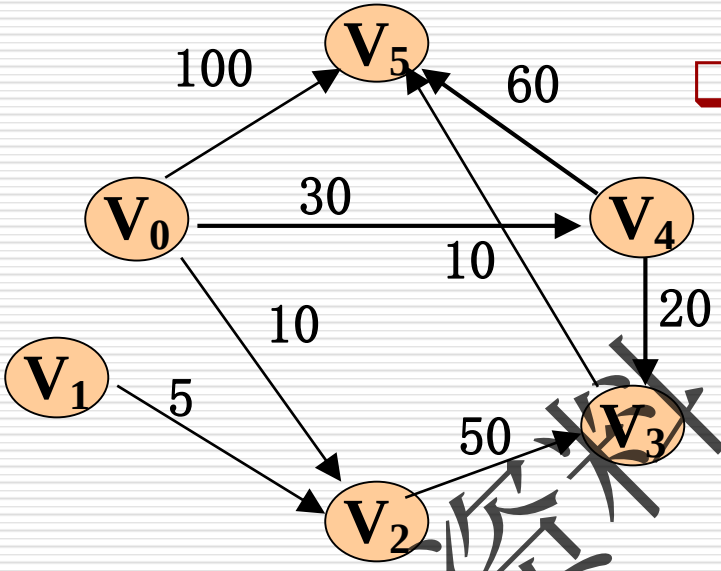
【算法描述】

```
//开始主循环，每次求得v0到某个顶点v的最短路径
   for(i=1;i<n; ++i){        //对其余n-1个顶点，依次进行计算
      min= MaxInt;
      for(w=0;w<n; ++w)
        if(!S[w]&&D[w]<min)
          {v=w; min=D[w];} //选择一条当前的最短路径，终点为v
      S[v]=true;                          //将v加入S
      for(w=0;w<n; ++w) //更新从v0出发到集合V-S上所有顶
   点的最短路径长度
      if(!S[w]&&(D[v]+G.arcs[v][w]<D[w])){
          D[w]=D[v]+G.arcs[v][w];        //更新D[w]
          Path [w]=v;                   //更改w的前驱为v
      }//if
   }//for
}//ShortestPath_DIJ
```

时间复杂度：$O(n^2)$

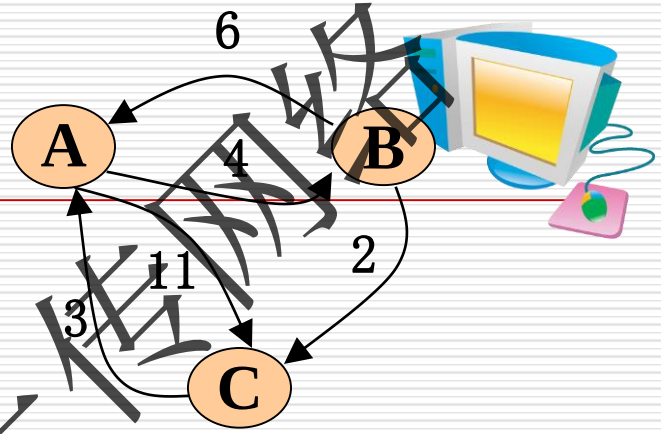
2. 每对顶点之间的最短路径

- 一种方法：每次以一个顶点为源点，重复执行迪杰斯特拉算法n次，求得每一对顶点之间的最短路径。



- 另一种方法：弗洛伊德算法(1962)
  - 以邻接矩阵作为图的存储结构

Floyd算法求最短路径



$$\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

(a) 路径长度

加入顶点A

$$\begin{bmatrix} & AB & AC \\ BA & & BC \\ CA & & \end{bmatrix}$$

(b) 路径

$$\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

(a) 路径长度

加入顶点B

$$\begin{bmatrix} & AB & AC \\ BA & & BC \\ CA & CAB & \end{bmatrix}$$

(b) 路径

$$\begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

(a) 路径长度

$$\begin{bmatrix} & AB & ABC \\ BA & & BC \\ CA & CAB & \end{bmatrix}$$

(b) 路径

$$\begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

(a) 路径长度

$$\begin{bmatrix} & AB & ABC \\ BA & & BC \\ CA & CAB & \end{bmatrix}$$

(b) 路径

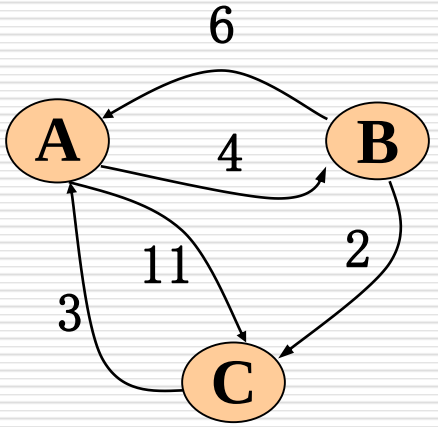
加入顶点C

$$\begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

(a) 路径长度

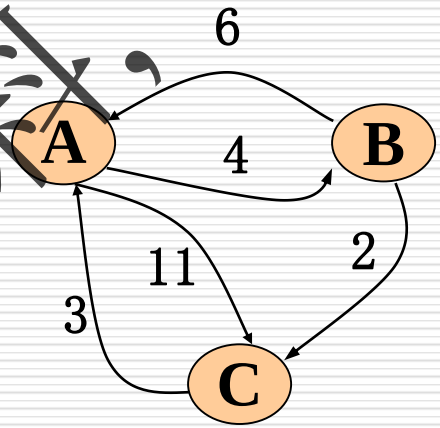
$$\begin{bmatrix} & AB & ABC \\ BCA & & BC \\ CA & CAB & \end{bmatrix}$$

(b) 路径



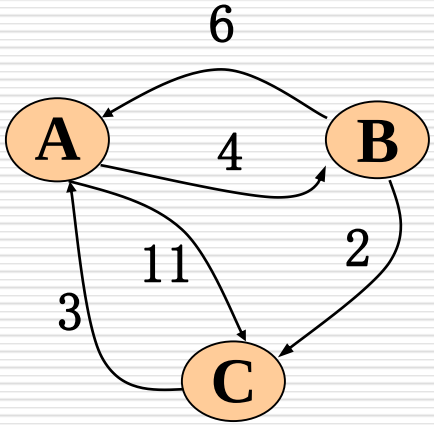
Floyd算法

- D[v][w]表示顶点v到顶点w的最短路径长度
  - 初值为顶点v到顶点w的弧上的权值
- P[v][w]表示顶点v到顶点w的最短路径
  - P[v][w][u]=true，u是顶点v到顶点w的最短路径上的顶点
  - P[v][w][u]=false，u不是顶点v到顶点w的最短路径上的顶点



Floyd算法

```
for(v=0;v<G.vexnum;v++) {      //初始化
    for(w=0;w<G.vexnum;w++) {
        D[v][w] = G.arcs[v][w];
        for(u=0; u<G.vexnum;u++)
            P[v][w][u] = false;
        if (D[v][w]<INFINITY) {
            P[v][w][v] = true;
            P[v][w][w] = true;
        }//if
    }//for
}//for
```



```
for(u=0; u<G.vexnum; u++) {
    for(v=0;v<G.vexnum;v++) {
        for(w=0;w<G.vexnum;w++) {
            if (D[v][u]+D[u][w])<D[v][w] {
                D[v][w] = D[v][u]+D[u][w];
                    for(i=0;  i<G.vexnum;  i++)//更新v到
    w的最短路径
                        p[v][w][i] = p[v][u][i] || p[u][w][i];
            }//if
        }//for w
    }//for v
}//for u
```

算法的时间复杂度为$O(n^3)$

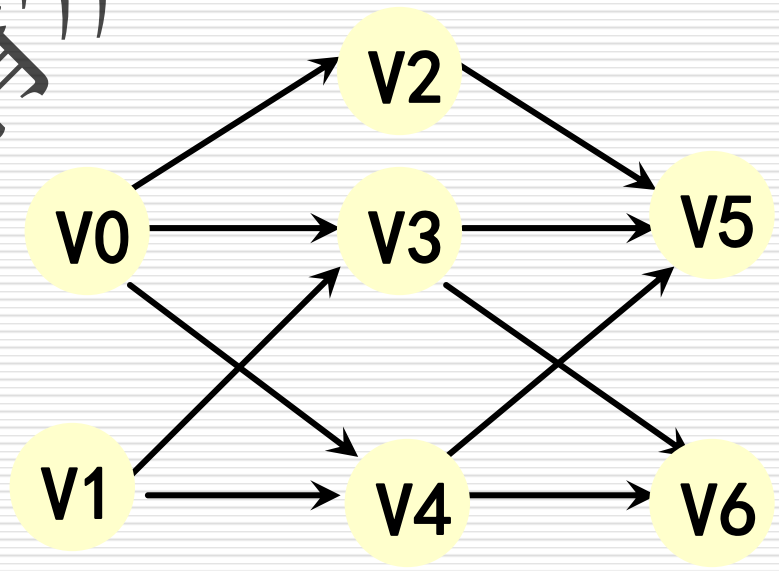
6.6.3 拓扑排序

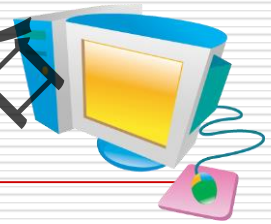
1. AOV网

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(Activity On Vertex net)。

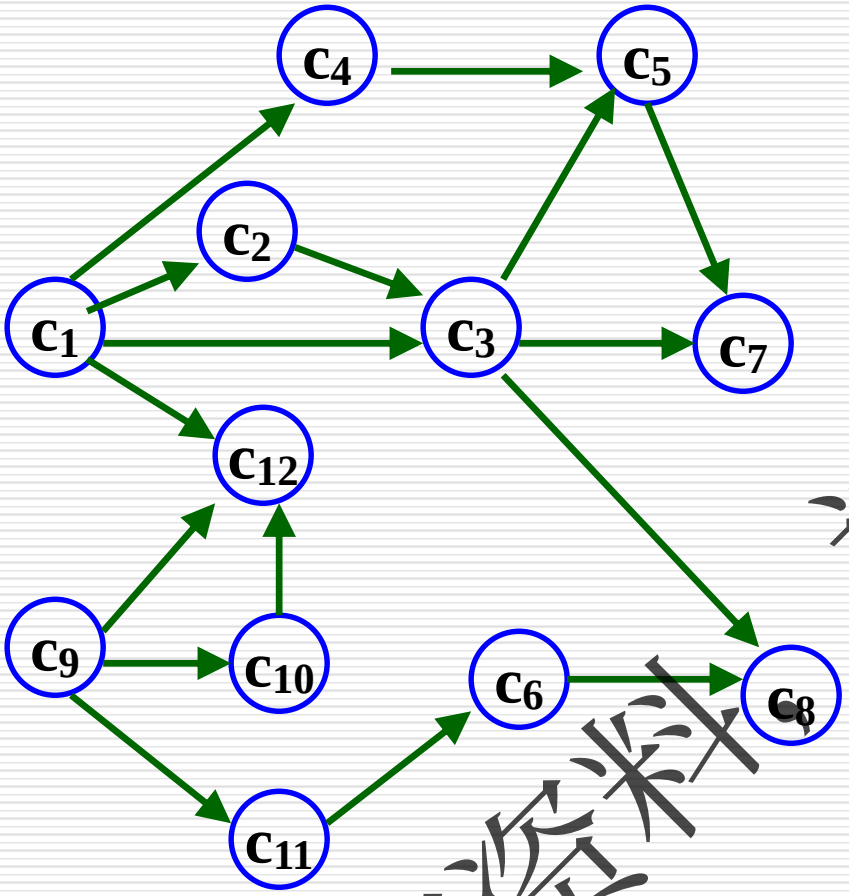
应用： 工程流程、生产过程中各道工序的流程等

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。





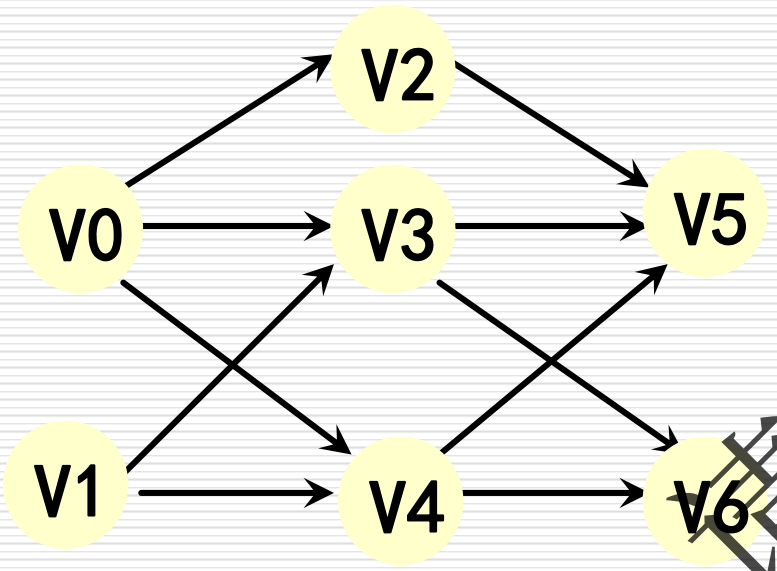
- 对工程问题，人们至少会关心如下两类问题：
  1) 工程能否顺序进行，即工程流程是否“合理”
  2) 完成整项工程至少需要多少时间，哪些子工程是影响工程进度的关键子工程
- 为求解工程流程是否“合理”，通常用AOV网表示工程流程

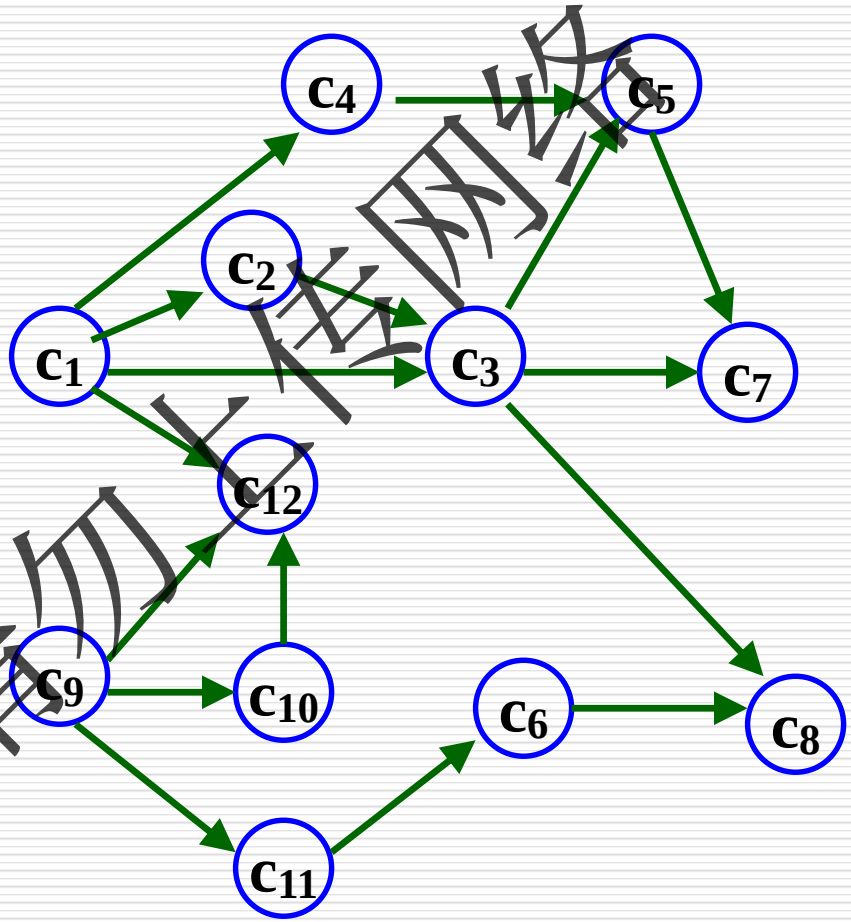


如何安排施工计划？

如何安排教学计划？

课程编号	课程名称	先修课程
C_1	程序设计基础	无
C_2	离散数学	C_1
C_3	数据结构	C_1, C_2
C_4	汇编语言	C_1
C_5	算法设计与分析	C_3, C_4
C_6	计算机体系结构	C_{11}
C_7	编译原理	C_3, C_5
C_8	操作系统原理	C_3, C_6
C_9	高等数学	无
C_{10}	线性代数	C_9
C_{11}	普通物理	C_9
C_{12}	数值分析	C_9, C_{10}, C_{11}



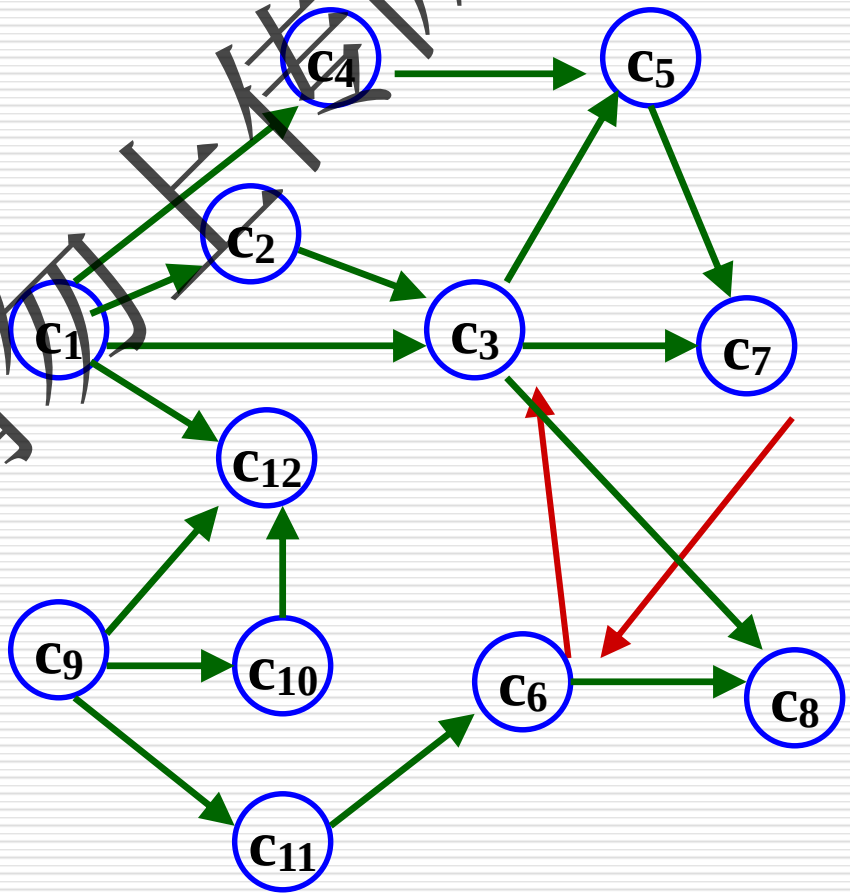


- 一个可行的施工计划： V0, V1, V2, V4, V3, V6, V5
- 一个可行的学习计划为：

  $C_1, C_9, C_4, C_2, C_{10}, C_{11}, C_{12}, C_3, C_6, C_5, C_7, C_8$

- **特点：**若在有向图中有弧<v,u>，则称顶点v是顶点u 的前驱，那么施工计划中顶点v 也排在u之前。也称u是v的后继。

- 一个AOV网不应该存在环，因为存在环意味着某项活动的进行应该以本活动的完成作为先决条件，这肯定是荒谬的。



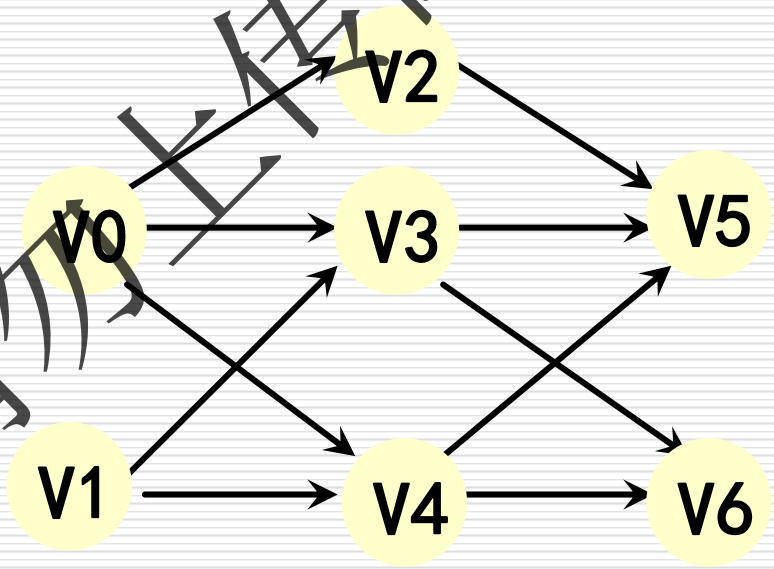
一个可行的学习计划：

$C_1, C_9, C_4, C_2, C_{10}, C_{11}, C_{12}, C_3, C_6, C_5, C_7, C_8$

2. 拓扑排序方法

拓扑排序： 将有向图中的顶点排成一个拓扑有序序列。

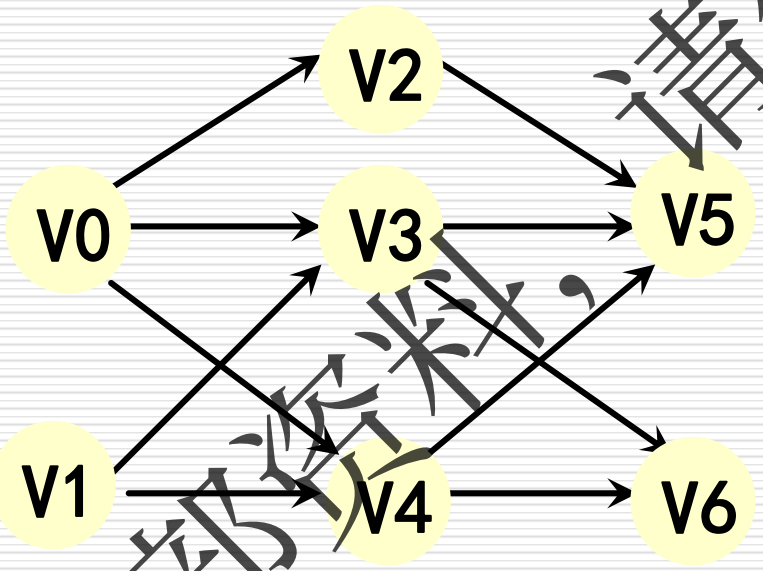
拓扑序列：有向图D的一个顶点序列称作一个拓扑序列。如果该序列中任两顶点v 、u ，若在D中v是u前驱，则在序列中v也是u前驱。



拓扑序列： V0, V1, V2, V4, V3, V6, V5

拓扑排序方法

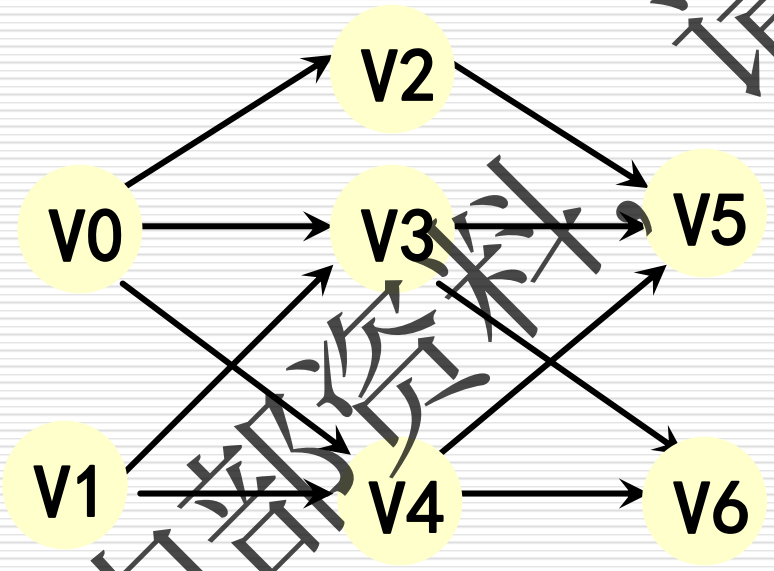
1) 在有向图中选一个无前驱的顶点v，输出之；
2) 从有向图中删除v及以v为尾的弧；
3) 重复1)、2)，直接全部输出全部顶点或有向图中不存在无前驱的结点时为止。



如何在计算机上实现
对有向图的拓扑排序？

拓扑排序方法

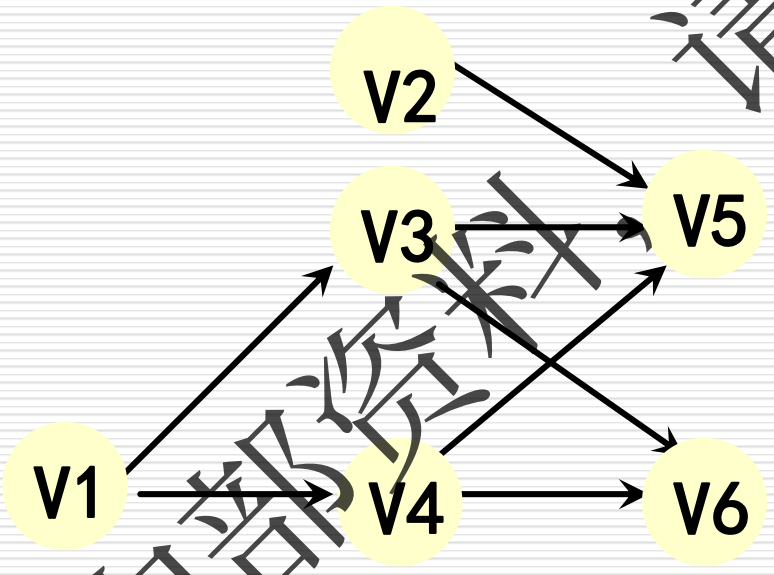
1) 在有向图中选一个无前驱的顶点v，输出之；
2) 从有向图中删除v及以v为尾的弧；
3) 重复1)、2)，直接全部输出全部顶点或有向图中不存在无前驱的结点时为止。



V0

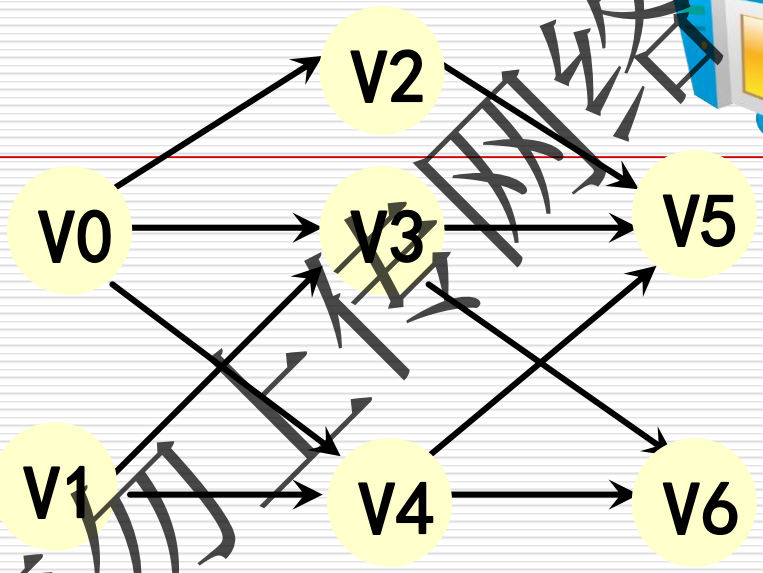
拓扑排序方法

1) 在有向图中选一个无前驱的顶点v，输出之；
2) 从有向图中删除v及以v为尾的弧；
3) 重复1)、2)，直接全部输出全部顶点或有向图中不存在无前驱的结点时为止。



V0 V1 V2 V3 V4 V5 V6

拓扑排序方法

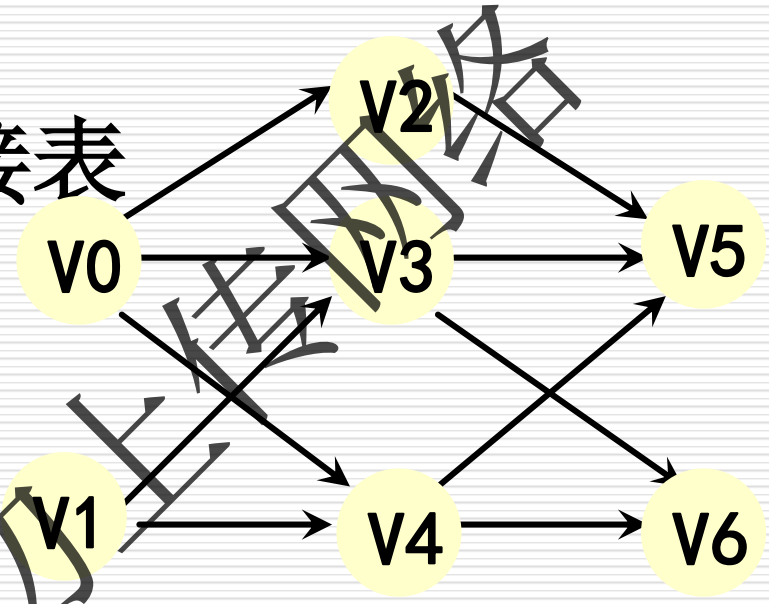


入度为0的顶点

1) 在有向图中选一个无前驱的顶点v，输出之；
2) 从有向图中删除v及以v为尾的弧；
3) 重复1)、2)，直接全部输出全部顶点或有向图中不存在无前驱的结点时为止。

v的邻接顶点入度减1

回顾：有向图的邻接表



顶点数据	指向第一个邻接顶点的指针
data	firstarc

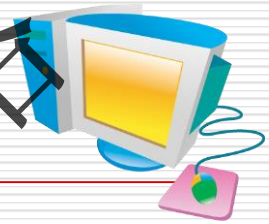
下标

0	v_0		2		3		4	∧
1	v_1		3		4	∧		
2	v_2		5	∧				
3	v_3		5		6	∧		
4	v_4		5		6	∧		
5	v_5	∧						
6	v_6	∧						

adjvex	weight	nextarc
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

表结点结构

3. 拓扑排序算法



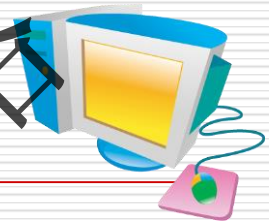
辅助数据结构：

（1）一维数组indegree[i]：存放各顶点入度；

（2）栈S：暂存所有入度为0的顶点；

（3）一维数组topo[i]，记录拓扑序列的顶点序号。

算法步骤



（1）求出各顶点的入度存入数组indegree中，并将入度为0的顶点入栈。

（2）while（栈非空）

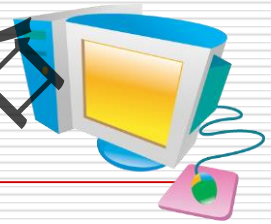
{将栈顶顶点vi 弹出并保存在拓扑序列数组topo中；

检查vi的出边，将每条出边<vi,vk>终点vk的入度减1，若vk的入度变为0，则将vk入栈；

}

3、若输出的顶点数小于AVO-网的顶点个数，则输出“有回路”；否则拓扑排序正常结束。

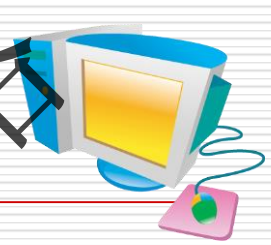
求有向图中顶点的入度



以邻接表为存储结构，求有向图中顶点的入度。

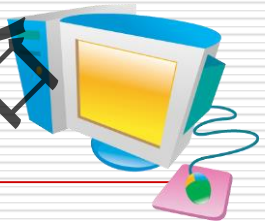
```
int indegree[G.vexnum]={0}; //临时存储各顶点的入度
for(i=0; i<G.vexnum; i++) //扫描邻接表，计算各顶点的入度
    for(p=G.vertices[i].firstarc; p; p=p->nextarc)
        indegree[p->adjvex]++;
```

算法描述



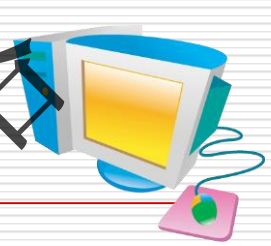
```
Status TopologicalSort(ALGraph G, int topo[ ])

int Indegree[G.vexnum]={0};    //保存各个顶点的入度
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]++;

InitStack(S);                          //栈初始化为空
for(i=0;  i<G.vexnum;  i++)
if (Indegree[i]==0)  Push(S,i);//入度为0者进栈

m = 0;                                 //对输出顶点计数
```

算法描述

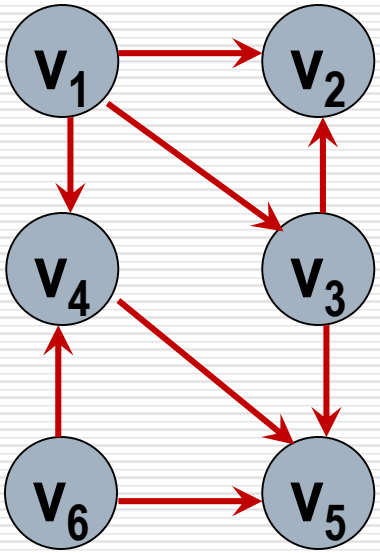


```
while(! StackEmpty(S))
{   Pop(S,i);        topo[m]=i;        ++m;
    p=G.vertices[i].firstarc;
    while(p!=NULL)
 {        k=p->adjvex; //vk为vi的邻接点
          --indgree[k];
          if (indegree[k]==0)  Push(S, k);
          p=p->nextarc;
    }//while
}//while
    if  (m<G.vexnum)  return ERROR;
    else return OK;
}
```

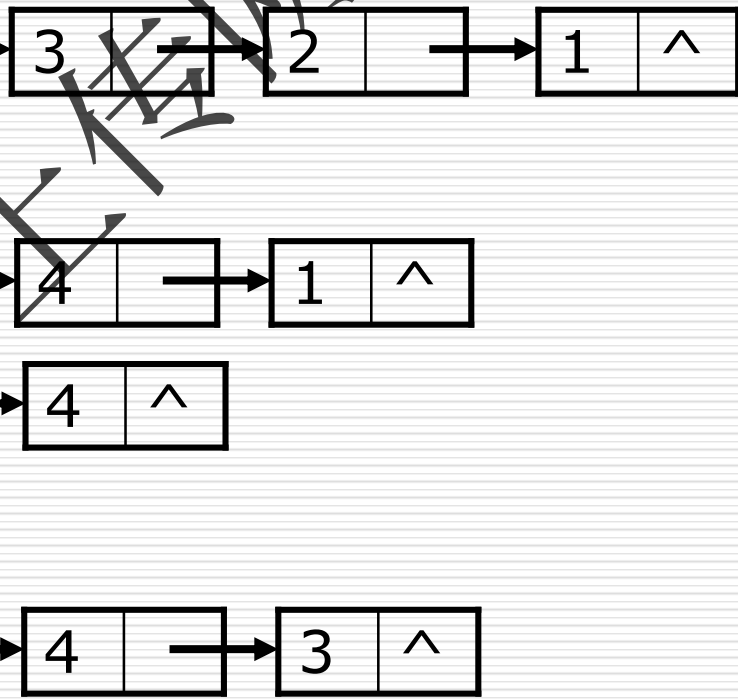
算法分析

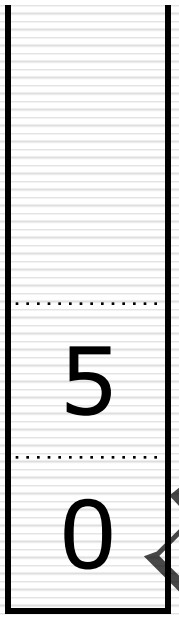


设AOV网有n个顶点，e条边。初始建立入度为0 的顶点栈，要检查所有顶点一次，执行时间为O(n)；排序中，若AOV网无回路，则每个顶点入、出栈各一次，每个边表结点被检查一次，执行时间为O(n+e)，所以总的时间复杂度为O(n+e)。

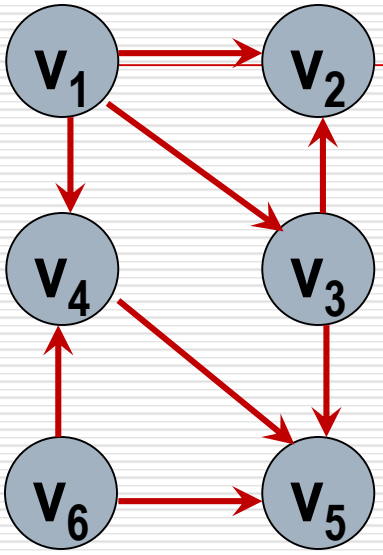


下标	入度	data	firstarc
0	0	v_1	
1	2	v_2	^
2	1	v_3	
3	2	v_4	
4	2	v_5	^
5	0	v_6	

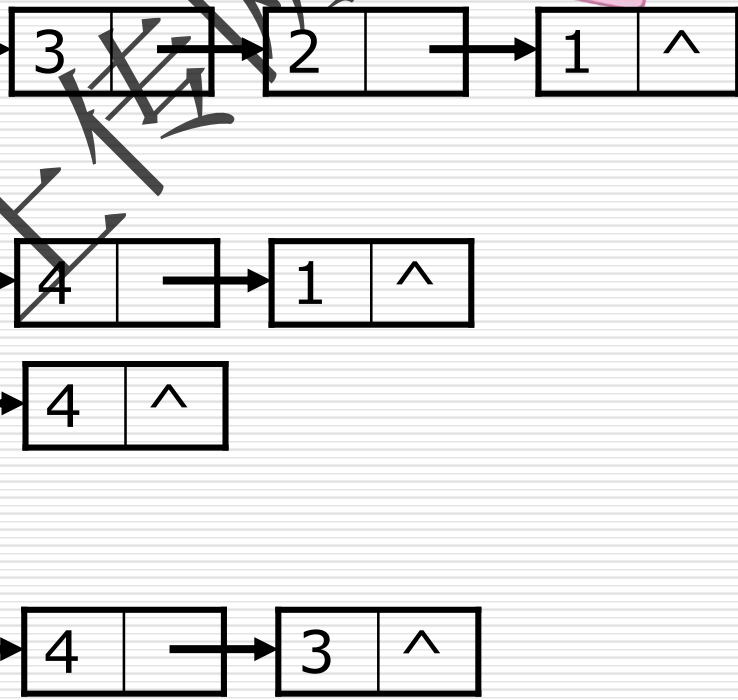


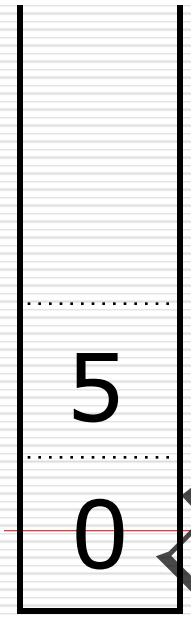


拓扑排序序列产生过程



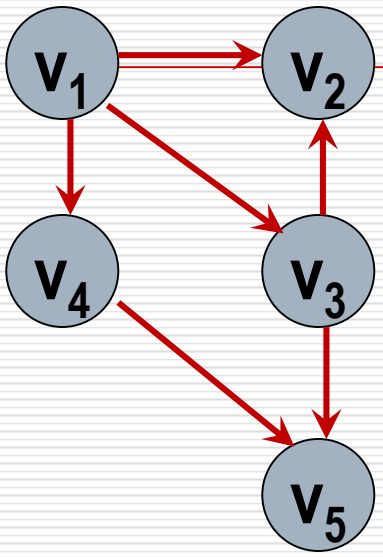
下标	入度	data	firstarc
0	0	v_1	
1	2	v_2	^
2	1	v_3	
3	1	v_4	
4	1	v_5	^
5	0	v_6	



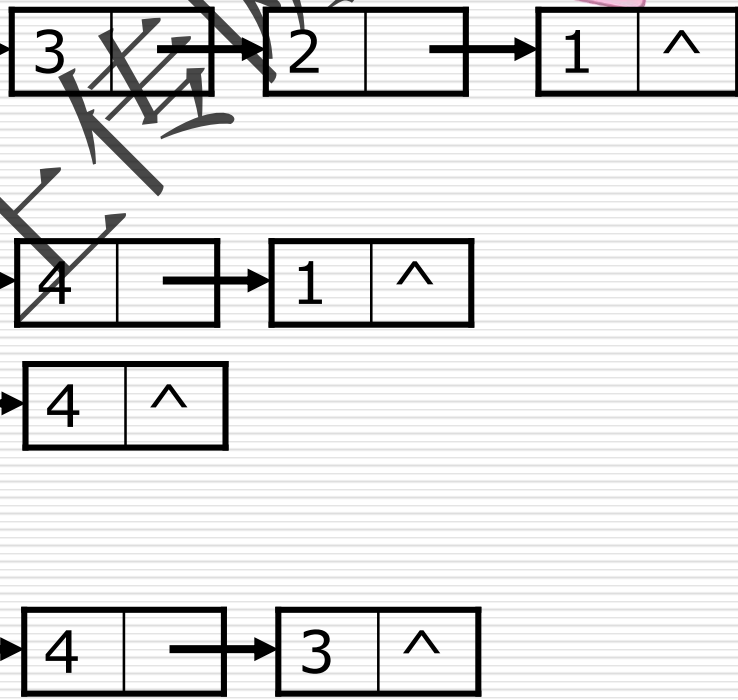


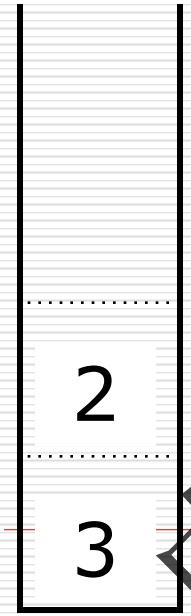
输出：v6

拓扑排序序列产生过程



下标	入度	data	firstarc
0	0	v_1	
1	1	v_2	^
2	0	v_3	
3	0	v_4	
4	1	v_5	^
5	0	v_6	





输出：v6

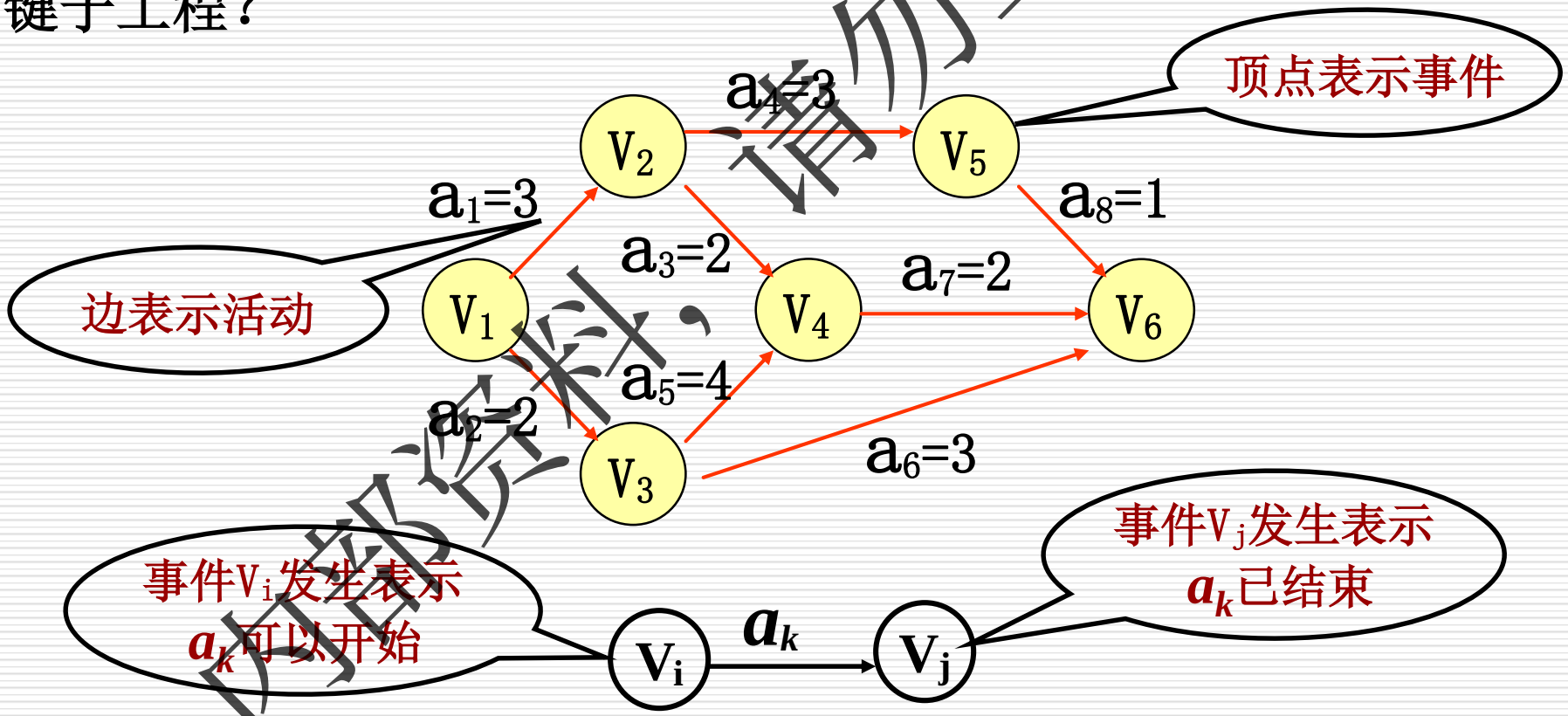
输出：v_1

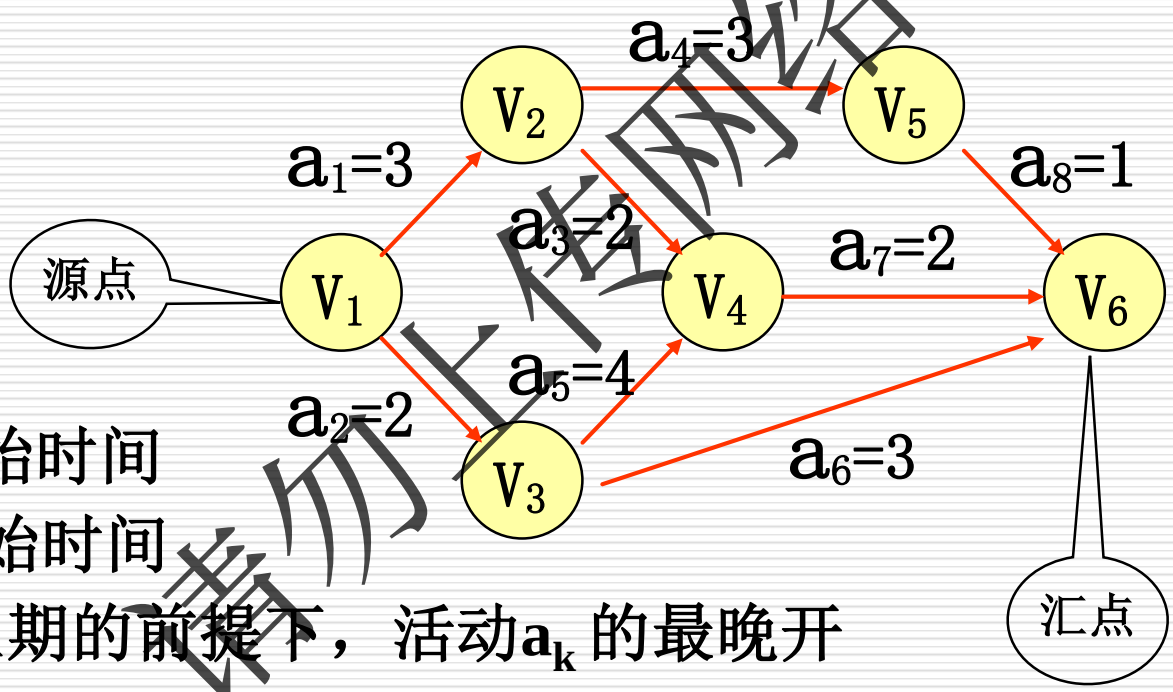
拓扑排序序列产生过程

6.6.4 AOE网(Activity On Edge net)

对工程人们关心两类问题：

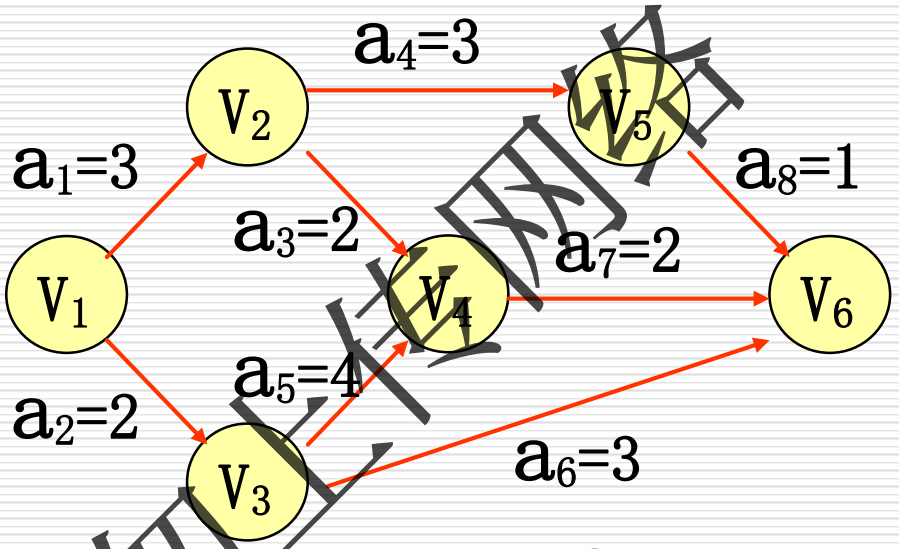
1）工程能否顺序进行，即工程流程是否“合理”(AOV网)
2）完成整项工程至少需要多少时间，影响工程进度的是哪些关键子工程？





- V_1开始事件(入度为0),
- V_6结束事件(出度为0)
- **e[k]: 活动a_k的最早开始时间**
- **l[k]: 活动a_k 的最晚开始时间**
  - 指在不推迟整个工期的前提下，活动a_k 的最晚开始时间
- **ve[j]: 事件v_j的最早发生时间**
- **vl[j]: 事件v_j的最迟发生时间**
  - 指在不推迟整个工期的前提下，事件v_j 的最晚发生时间

活动a_k 的延迟时间l[k]-e[k]

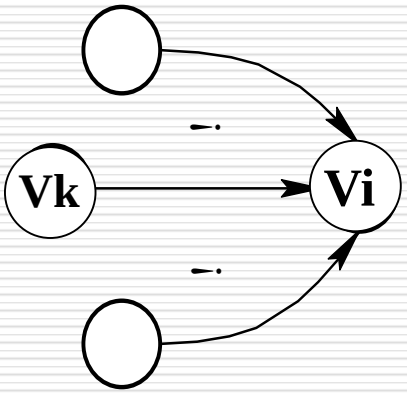


1. 事件v_j的最早发生时间ve[j]

ve[0]=0

ve[i]=Max{ve(k)+dut(<V_k,Vi>)}

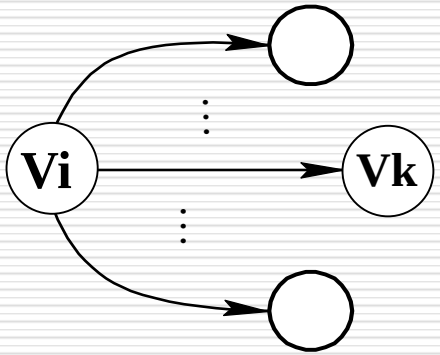
- ve[i]等于从源点到顶点v_i的最长路径的长度

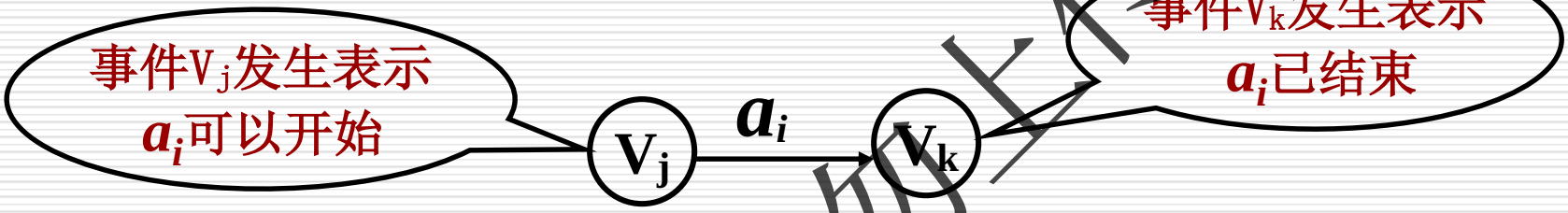


2. 事件v_i的最晚发生时间vl[i]

vl[n-1]=ve[n-1]

vl[i]=Min{vl(k)-dut(<V_i,V_k>)}



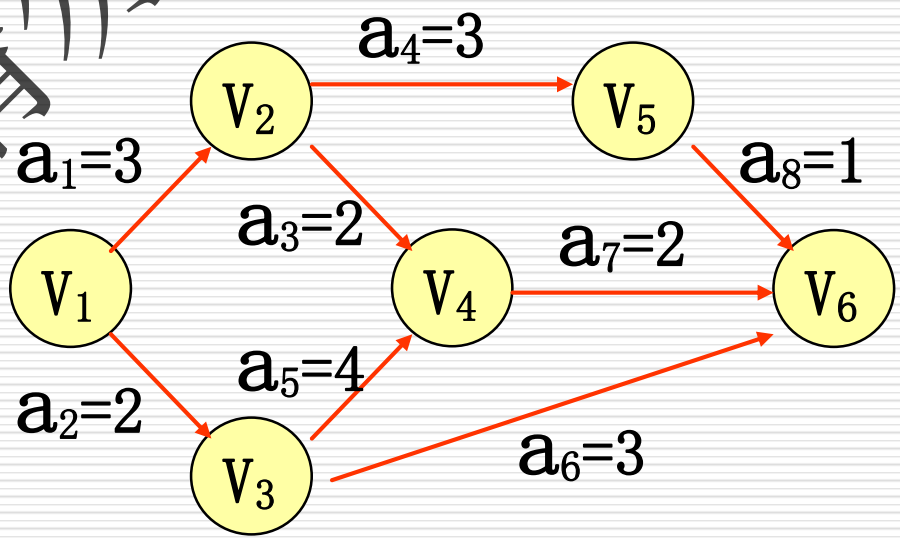


3. 活动a_i的最早开始时间e[i]

   e[i]=ve[j]

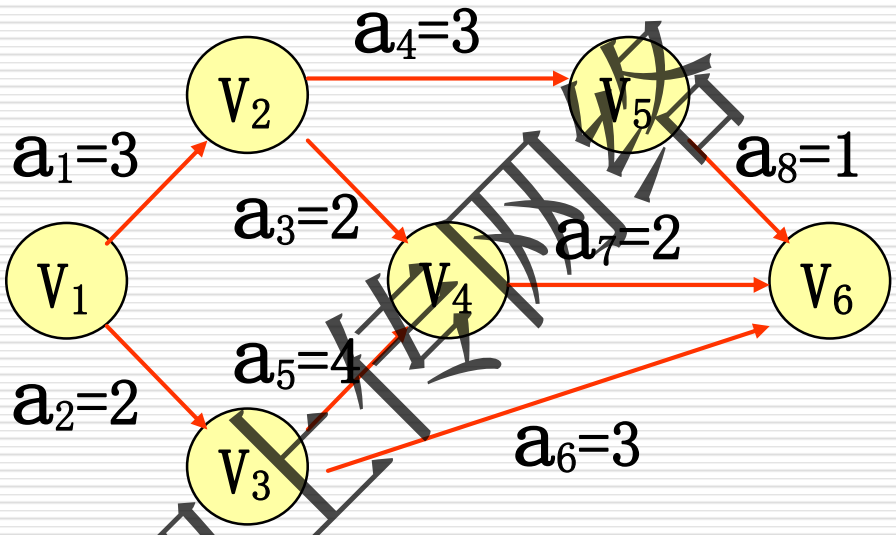
4. 活动a_i的最晚开始时间l[i]

   l[i]=vl[k]-dut(<V_j,V_k>)



$e[i]=ve[j]$

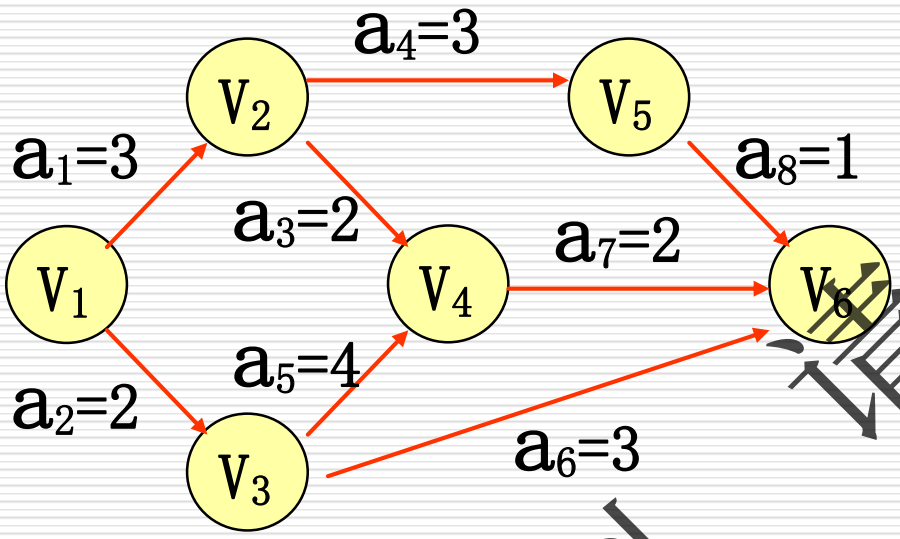
$l[i]=vl[k] - dut(<V_j,V_k>)$



事件	最早 发生时间	最晚 发生时间
V_1	0	0
V_2	3	4
V_3	2	2
V_4	6	6
V_5	6	7
V_6	8	8

活动	最早 开始时间	最晚 开始时间
a_1	0	1
a_2	0	0
a_3	3	4
a_4	3	4
a_5	2	2
a_6	2	5
a_7	6	6
a_8	6	7

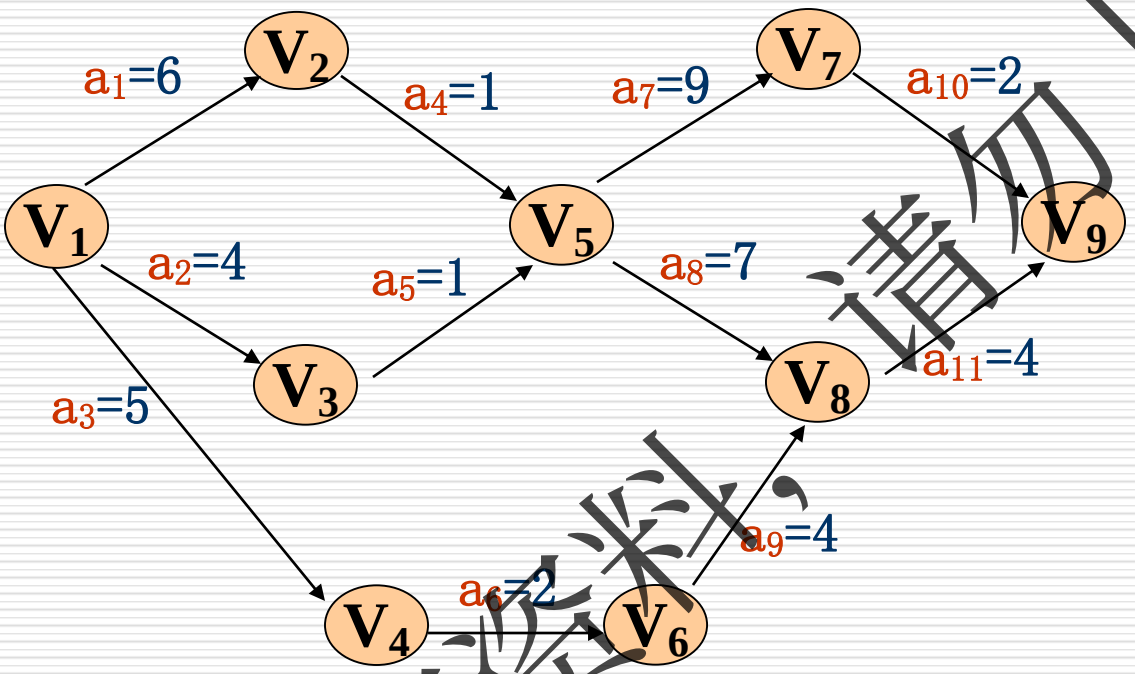
7.5.2关键活动和关键路径



活动	最早开始时间	最晚开始时间	延迟时间
a_1	0	1	1
a_2	0	0	0
a_3	3	4	1
a_4	3	4	1
a_5	2	2	0
a_6	2	5	3
a_7	6	6	0
a_8	6	7	1

- 若活动a_k的延迟时间等于0，则称之为关键活动。
- 从源点到汇点的所有路径中，长度最长的路径叫做关键路径。

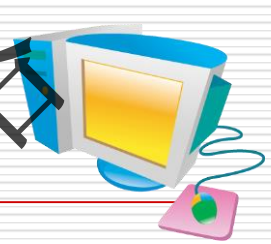
例：求下面AOE网的关键活动和关键路径



事件	最早发生时间	最晚发生时间
V_1	0	0
V_2	6	6
V_3	4	6
V_4	5	8
V_5	7	7
V_6	7	10
V_7	16	16
V_8	14	14
V_9	18	18

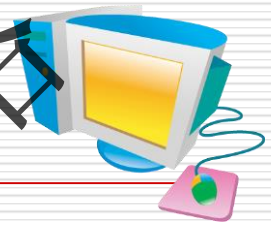
拓扑序列：V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8 V_9

求关键路径



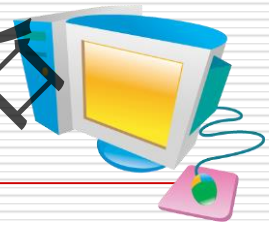
- 步骤1：输入e条弧<j,k>，建立AOE网的存储结构，顶点数为n，顶点从0开始编号，设源点为v_0、汇点为v_{n-1}。
- 步骤2：从源点v_0出发，令ve[0]=0，按拓扑有序求其余各顶点事件的最早发生时间ve[i](1≤i≤n-1)。如果得到的拓扑有序序列中顶点个数小于网中的顶点数，则说明网中存在环，不能求关键路径，算法终止；否则执行步骤3；
- 步骤3：从汇点v_{n-1}出发，令vl[n-1]=ve[n-1]，按逆拓扑有序求其余各顶点事件的最迟发生时间vl[i](0≤i≤n-2)；
- 步骤4：根据各顶点的ve、vl值，求每个活动ai的最早开始时间e(i)和最迟开始时间l(i)，若某个活动ai满足条件e(i)=l(i)，则为关键活动。

算法6.12 拓扑排序算法

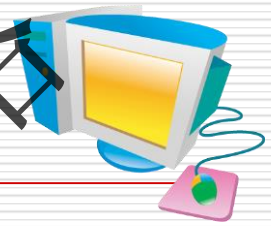


```
Status TopologicalSort(ALGraph G, int topo[ ])

int Indegree[G.vexnum]={0};    //保存各个顶点的入度
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]++;

 InitStack(S);                          //栈初始化为空
 for(i=0;  i<G.vexnum;  i++)
 if (Indegree[i]==0)  Push(S,i);//入度为0者进栈

   m = 0;                               //对输出顶点计数
```

```
while(! StackEmpty(S))
{   Pop(S,i);       copo[m]=i;       ++m;
   p=G.vertices[i].firstarc;
   while(p!=NULL)
 {        k=p->adjvex; //vk为vi的邻接点
         --indgree[k];
         if (indegree[k]==0)   Push(S, k);
         p=p->nextarc;
    }//while
}//while
    if  (m<G.vexnum) return ERROR;
    else return OK;
}
```

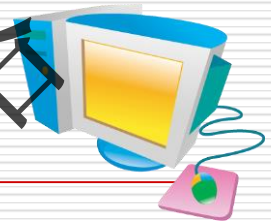


算法6.13 关键路径算法



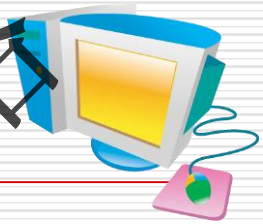
```
void CriticalPath (ALGraph G)
{
  //有向网G采用邻接表存储结构，输出G的各项关键活动
  if(!TopologicalSort( G, topo)) return ERROR;
//调用拓扑排序算法，使拓扑序列保存在topo中，若调用失败，则存在有向环

    n=G.vexnum;                    //n为顶点个数
    for(i=0;i<n;i++)
          ve[i]=0;    //初始化，给每个事件的最早发生时间置初值0
```

（按拓扑次序求每个事件的最早发生时间）



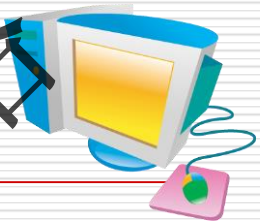
```
for(i=0; i<n; i++)
{  k=topo[i];  //取得拓扑序列中的顶点序号k
   p = G.vertices[k].firstarc; //p指向k的第一个邻接点
   while(p!=NULL)
      {   j= p->adjvex;          //j为邻接顶点的序号
        if (ve[j]<ve[k] + p->weight)
            ve[j] = ve[k] + p->weight;
//更新顶点j的最早发生时间ve[j]
          p=p->nextarc;      //p指向k的下一个邻接点
        }//while
    }//for
```

（按拓扑逆次序求每个事件的最迟发生时间）



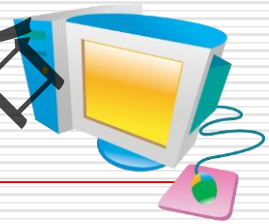
```
for(i=0;i<n;i++)            vl[i]=ve[n-1];
for(i=n-1; i>=0; i--)
{  k=topo[i];  //取得拓扑序列中的顶点序号k
   p = G.vertices[k].firstarc; //p指向k的第一个邻接点
   while(p!=NULL)
     {   j= p->adjvex;          //j为邻接顶点的序号
       if (vl[k]>vl[j] - p->weight)
           vl[k] = vl[j] - p->weight;
//更新顶点k的最迟发生时间ve[j]
        p=p->nextarc;      //p指向k的下一个邻接点
       }//while
    }//for
```

（判断每一活动是否为关键活动）



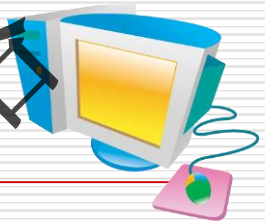
```
for(i=0;i<n;i++)
 {//每次循环针对 vi为活动开始点的所有活动
   p = G.vertices[i].firstarc;//p指向i的第一个邻接点
   while(p!=NULL)
   {   j= p->adjvex;       //j为邻接顶点的序号
       e=ve[i];   //计算活动<vi,vj>的最早开始时间
       l=vl[j]- p->weight; //计算活动<vi,vj>的最迟开始时间
       if(e==l) //若为关键活动，则输出<vi,vj>
           cout<<G.vertices[i].data<<G.vertices[j].data;
       p=p->nextarc;       //p指向i的下一个邻接点
     }//while
  }//for
}
```

算法分析：



- 若有向图用邻接链表表示，查找每个顶点的邻接点所需时间为O(e)，e为图中的边(弧)数，算法的时间复杂度为O(n+e)
- 关键路径可能不唯一；
- 影响关键路径的因素是多方面的，关键活动的速度提高是有限的。

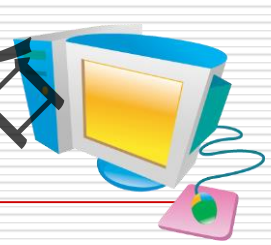
考研真题选讲



【2011年计算联考真题】
下列关于图的叙述中，正确的是（C）。
I. 回路是简单路径
II. 存储稀疏图，用邻接矩阵与邻接表更省空间
III. 若有向图中存在拓扑序列，则该图不存在回路
A. 仅II　　B. 仅I、II　　C. 仅III　　D. 仅I、III

【2009年计算联考真题】
下列关于无向连通图特性的叙述中，正确的是（A）。
I. 所有顶点的度数之和是偶数
II. 边数大于顶点个数减1
III. 至少有一个顶点的度为1
A. 只有I　　B. 只有II　　C. I和II　　D. I和III

考研真题选讲



【2010年计算联考真题】

若无向图G=(V,E)中含有7个顶点，要保证图G在任何情况下都是连通的，则需要的边数最少是（ C ）。

A. 6　　B. 15　　C. 16　　D. 21

【2013年计算联考真题】

设图的邻接矩阵A如下所示，各点的度依次是（ C ）。

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

A. 1,2,1,2　　B. 2,2,1,1　　C. 3,4,2,3　　D. 4,4,2,2

考研真题选讲

【2017年计算联考真题】

已知无向图G含有16条边，其中度为4的顶点个数为3，度为3的顶点个数为4，其他顶点的度均小于3. 图G所含的顶点个数至少是（ B ）。

A. 10　　B. 11　　C. 13　　D. 15

【2013年计算联考真题】

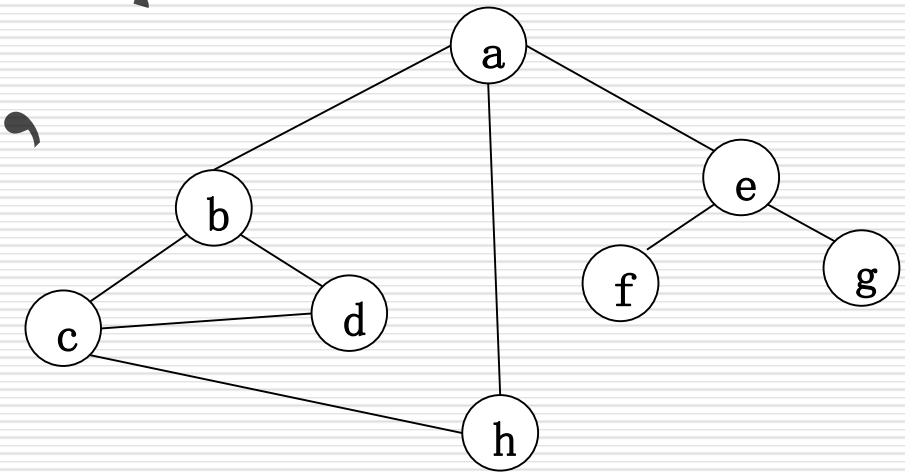
若对如下无向图进行遍历，则下列选项中，不是广度优先遍历序列的是（ D ）。

A. h, c, a, b, d, e, g, f

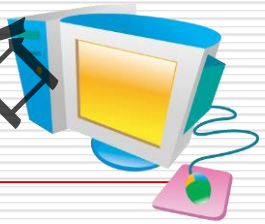
B. e, a, f, g, b, h, c, d

C. d, b, c, a, h, e, f, g

D. a, b, c, d, h, e, f, g



考研真题选讲



【2010年计算联考真题】

对有n个顶点、e条边且使用邻接表存储的有向图进行广度优先遍历，其算法的时间复杂度是（C）。

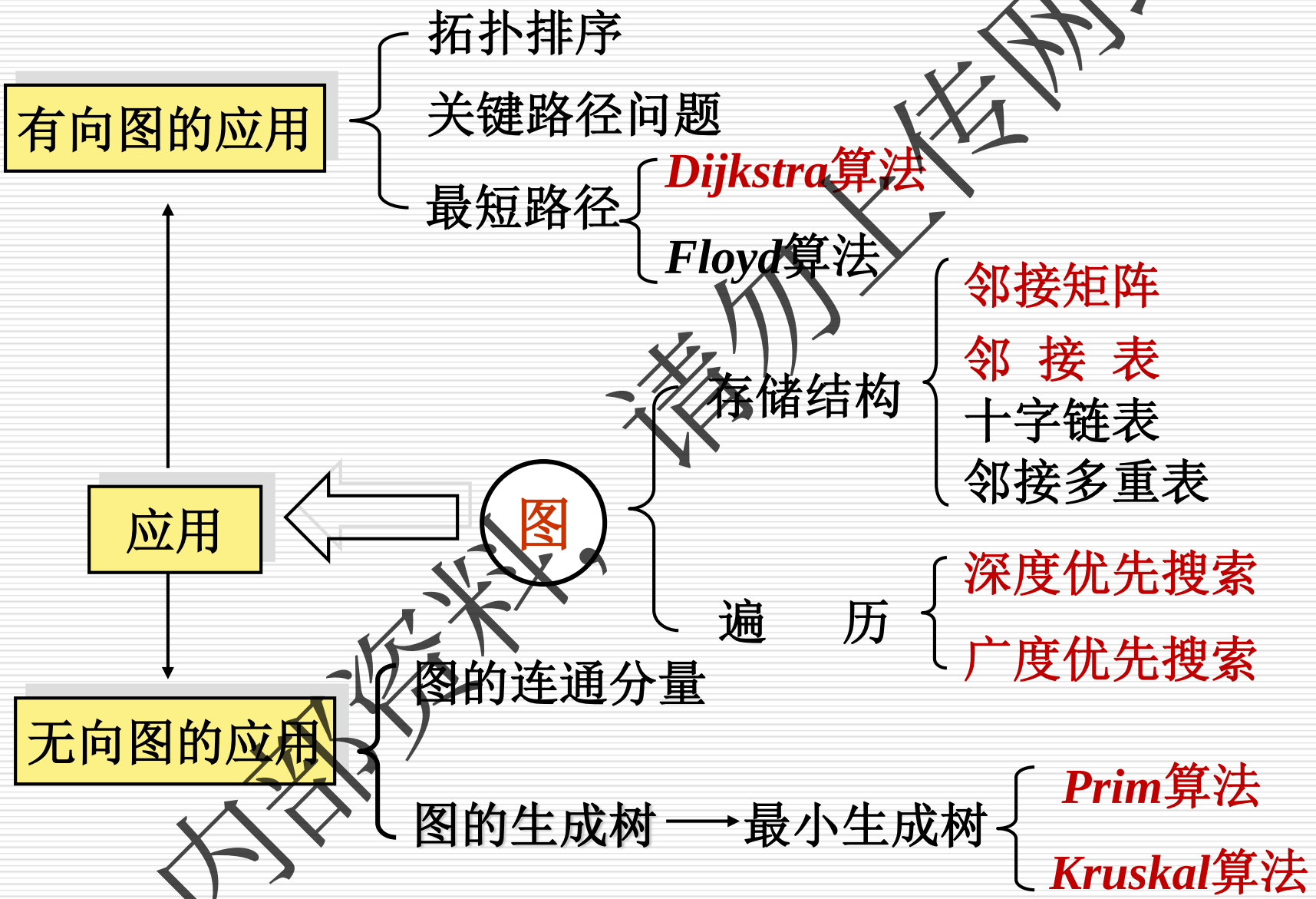
A．O(n)　　B．O(e)　　C．O(n+e)　　D．O(n*e)

【2015年计算联考真题】

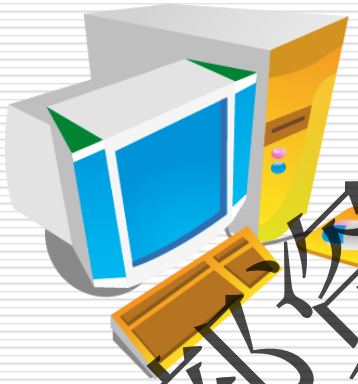
设有向图G=(V,E)，顶点集V={V0,V1,V2,V3}，边集E={<V0,V1>,<V0,V2>,<V0,V3>,<V1,V3>}，若从顶点V0开始对图进行深度优先遍历，则可能得到的不同遍历序列个数是（D）。

A．2　　B．3　　C．4　　D．5

6.8本章小结



第六章　结束



Thank you!