浙江大学管理学院
SCHOOL OF MANAGEMENT
ZHEJIANG UNIVERSITY

第6章 图

AACSB ACCREDITED EFMD EQUIS ACCREDITED ASSOCIATION AMBA ACCREDITED CAMEA 中国高质量MBA教育认证 IQA CEEMAN Accredited

教学目标

1. 熟练掌握图的基本概念及相关术语和性质
2. 熟练掌握图的邻接矩阵和邻接表两种存储表示方法
3. 熟练掌握图的两种遍历方法DFS和BFS
4. 熟练掌握图的应用：最小生成树、最短路径、拓扑排序、关键路径

6.1 图的基本术语

1、顶点：图中的数据元素。

2、图的二元组定义：

G=(V，{R})

其中： V 为顶点的非空有穷集合

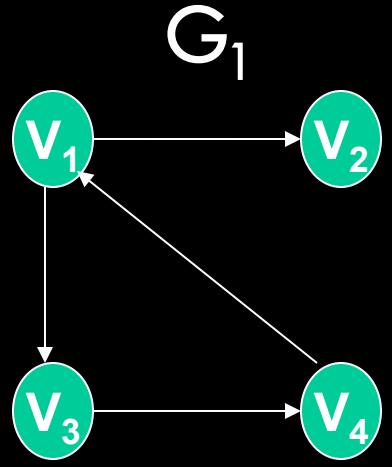
R 为顶点关系的集合

3、弧、有向图、边、无向图

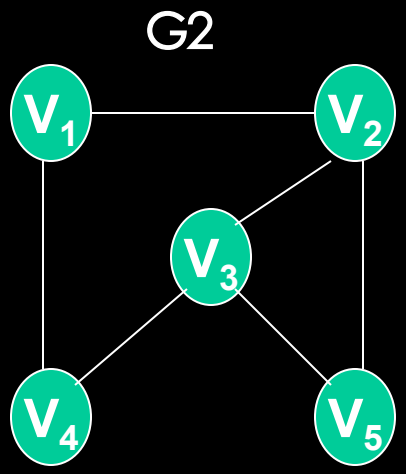
4、无向完全图：$e = C_n^2 = n(n-1)/2$

5、有向完全图： $e = P_n^2 = n(n-1)$

6、稀疏图、稠密图



有向图G1



无向图G2

7、权、网（带权的图称网。包括有向、无向）

8、子图：图G’是图G的一部分。即 对图：

G=(V,{R}) , G’=(V’,{R’})

满足 V’ ⊆ V AND R’ ⊆R ，

称G’为G的**子图**。

9、邻接点：对无向图G=(V,{E})，若(v,v’)∈E，

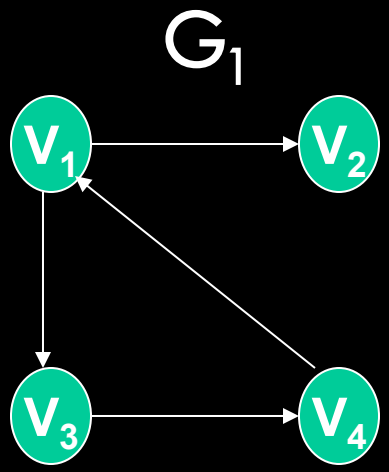
称v,v’为**邻接点**。

边(v,v’)**依附**于顶点v,v’或称**相关联**。

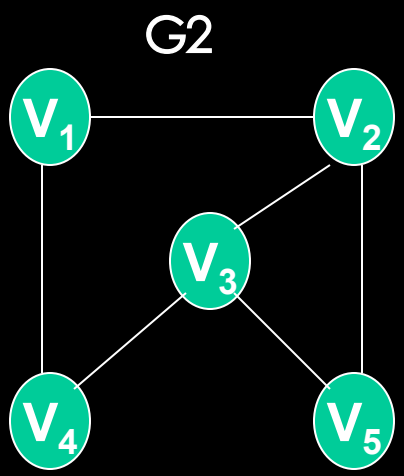
对于有向图G=(V,{E})，若弧<v,v’> ∈E，

称顶点v**邻接到**v’，或称v’**邻接自**v；

弧<v,v’>与顶点v,v’相关联。



有向图G1



无向图G2

10、度、入度、出度：

无向图顶点v的度：与v相关联的边的数目，记TD(v)。

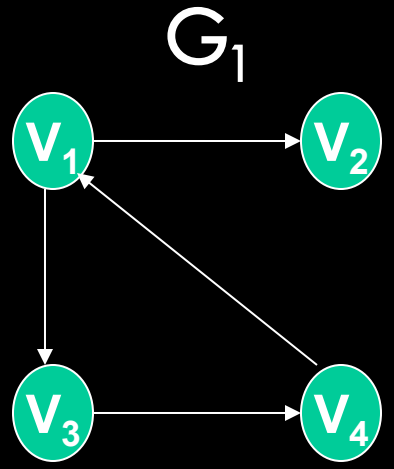
有向图顶点v的度：v的入度与出度之和。ID(v)+OD(V)

v 的入度是以v为头的弧的数目；

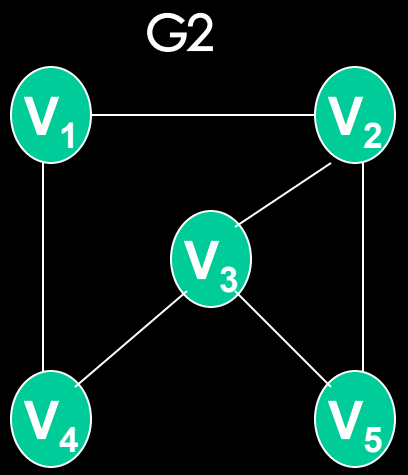
v 的出度是以v为尾的弧的数目。

对于有n个顶点，e条边或弧的图，满足：

$e = (TD(v_1)+ TD(v_2)+\ldots\ldots+ TD(v_n)) / 2$



有向图G1



无向图G2

11、路径、简单路径、路径长度：

从一个顶点到另一个顶点所经过的顶点序列称为路径。

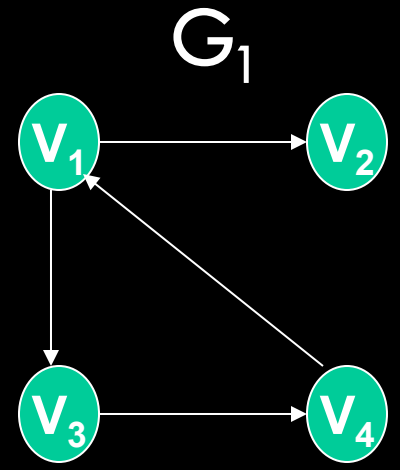
序列中顶点不重复出现的路径称为简单路径。

路径长度是路径上的边或弧的数目。

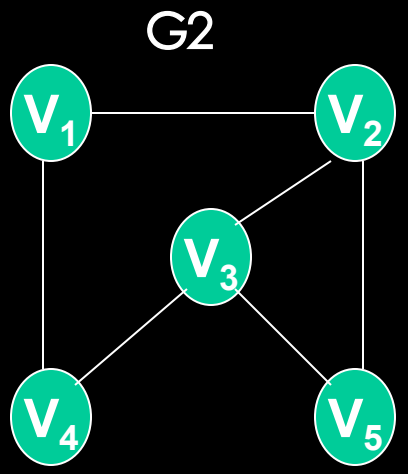
12、回路（环）、简单回路（简单环）：

第一个顶点和最后一个顶点相同的路径称为回路或环。

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路，称为简单回路或简单环。



有向图G1



无向图G2

对于无向图

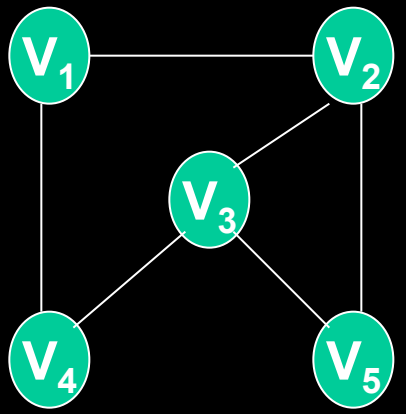
13、连通、连通图、连通分量：

若顶点v到顶点v’有路径，称v与v’是**连通**的。

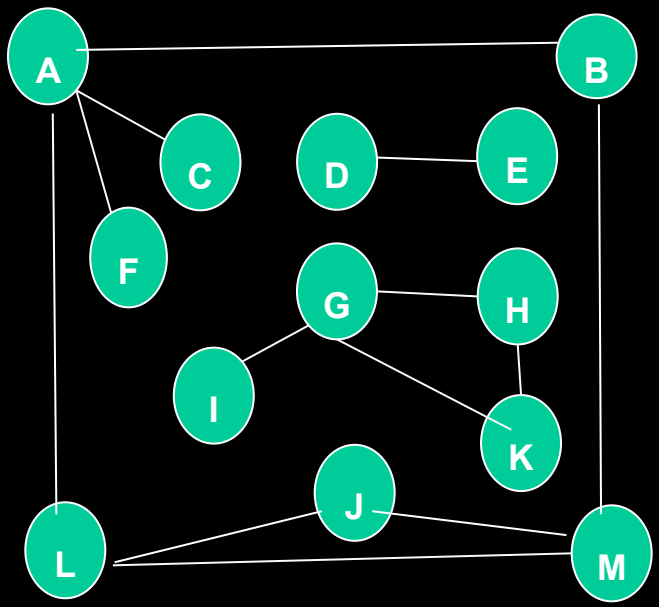
图中任意两个顶点都是连通的，称为**连通图**。

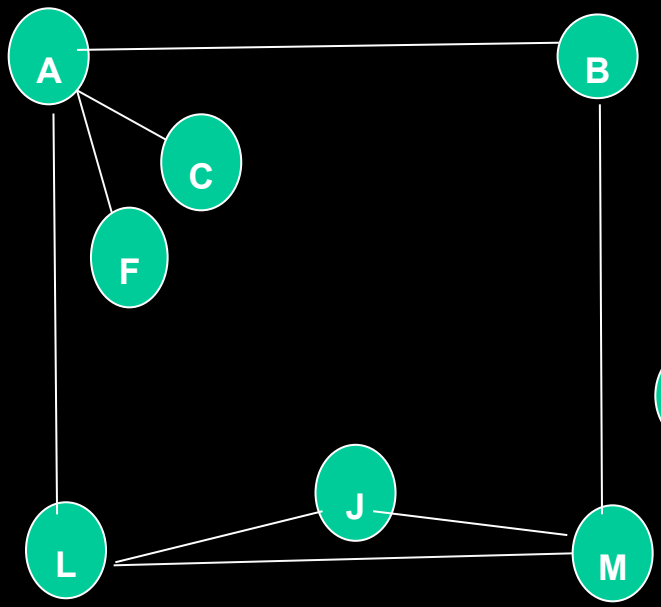
无向图中的**极大连通子图**称**连通分量**。

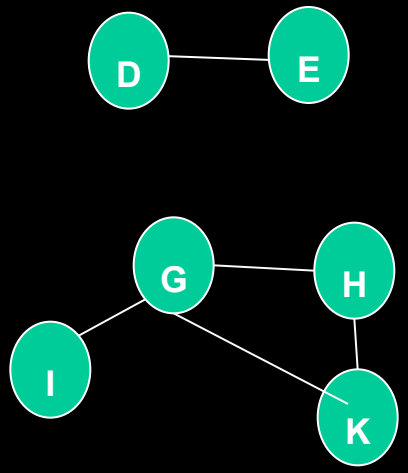
e.g. G2，G3(P159)



无向图G2







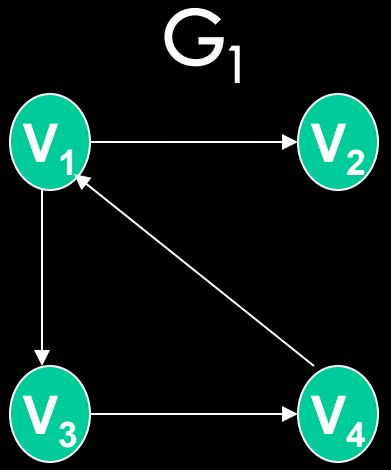
无向图G3

对于有向图

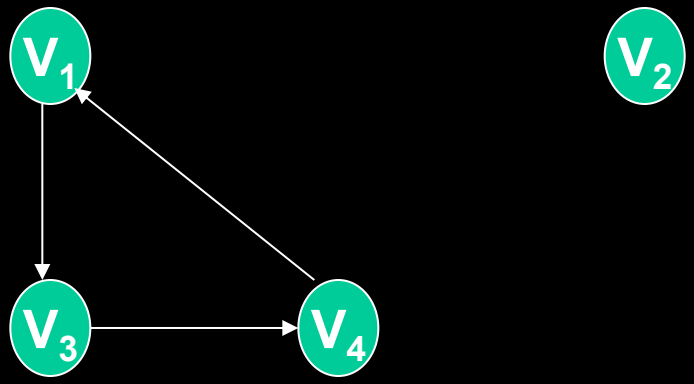
14、强连通图、强连通分量：

对有向图的任意两个顶点v_i、v_j，从v_i到v_j和从v_j到v_i都存在路径，称为**强连通图**。

有向图中的极大强连通子图称**强连通分量**。



有向图G1



2个强连通分量

15、生成树：

生成树必针对一个连通图。

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（n个顶点），但只有足以构成一棵树的 n-1条边。

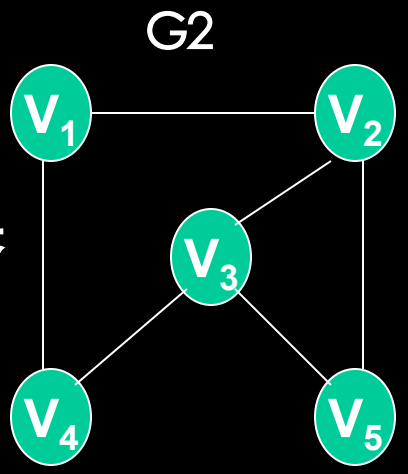
性质：

若在一棵生成树上添加一条边，则必定构成一个环。

一棵n个顶点的生成树有且仅有n-1条边，如果边小于n-1，则是非连通图。

思考：一个连通图只有一棵生成树吗?

有n-1条边的图一定是生成树吗?



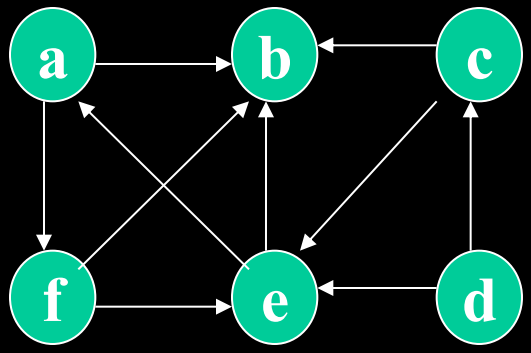
无向图G2

16、有向树:

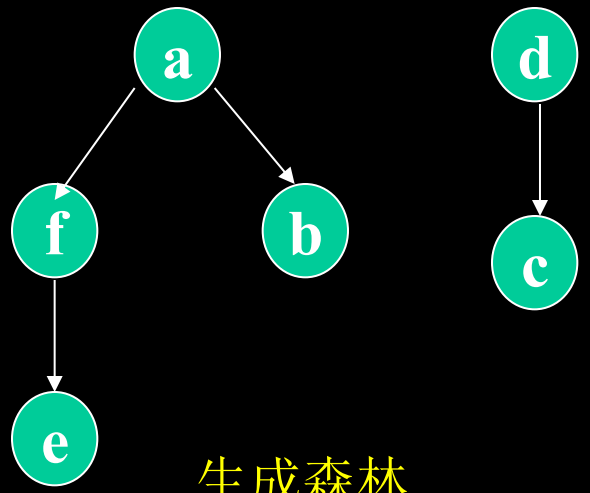
如果一个有向图满足以下条件：有一个顶点的入度为0，其余顶点的入度均为1，称为有向树。

17、生成森林:

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。

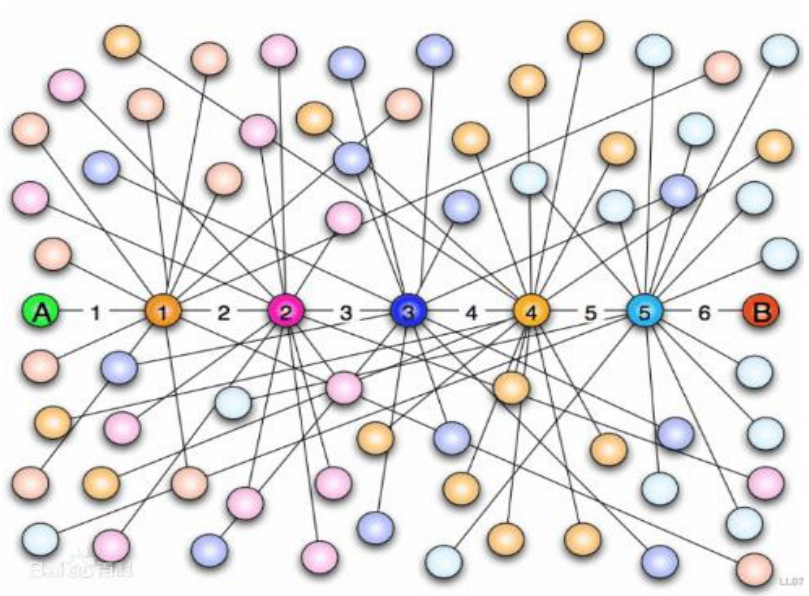


一个有向图



生成森林

6.2 案例引入-----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

图的ADT定义：p187

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

图的存储结构必须反映出图的一些基本特征：

* 图的种类 （有向图、有向网、无向图、无向网）
* 图的顶点数和图的弧数（边数）
* 弧（边）的信息 （邻接点、权、含义）
* 顶点的信息 （编号、名称、含义）

因此，设计图的存储结构必须对上述各项进行描述。

1、邻接矩阵（数组）表示

```
# define  INFINITY     INT_MAX    // 最大值∞,用于网的权值
# define MAX_VERTEX   20    //顶点个数
# typedef  enum {DG,DN,AG,AN}  GraphKind; //图的种类
typedef  struct  ArcCell {
  // 二维数组定义弧或边，对应的顶点编号为数组下标
      VRType          adj;   // 对无权图用0/1，如带权，则为权值( int )
      InfoType        *info;  // 弧或边的相关信息（ char )
} ArcCell, AdjMatrix[MAX_VERTEX] [MAX_VERTEX]

typedef struct {
  VertexType   vexs[MAX_VERTEX];    //顶点数组存放顶点信息
  AdjMatrix      arcs;                          //邻接矩阵
  int                vexnum,arcnum;          //图的顶点数与弧数
  GraphKind    kind;                          //图的种类
}  Mgraph;
```

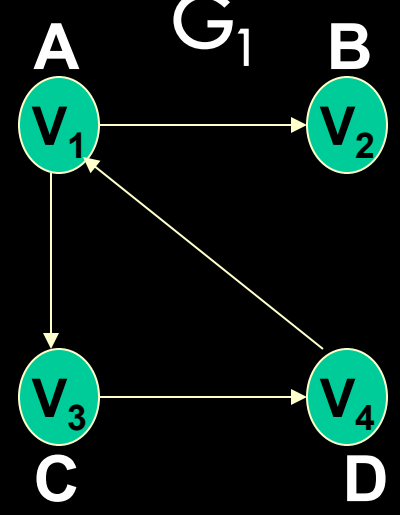
- 有n个顶点的图G=(V,E)的邻接矩阵为n阶方阵A

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

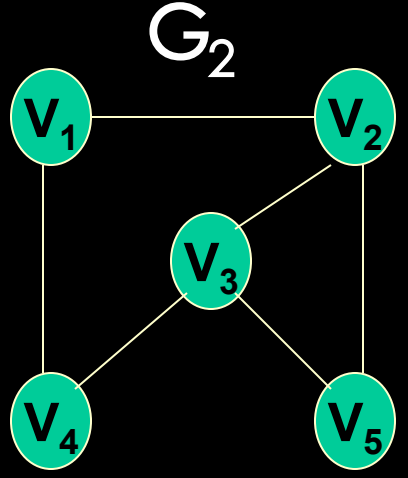
- 有n个顶点的网G=(V,E)的邻接矩阵可定义为

$$A[i][j]=\begin{cases}w_{ij} & \text{若}<v_i,v_j>\text{或}(v_i,v_j)\in E \\ \infty & \text{反之}\end{cases} \qquad 1\le i,j\le n$$

```
Mgraph  G1;      //定义图G1
G1.kind=DG;      // G1为有向图
G1.vexnum=4;  // G1的顶点数为4
G1.arcnum=4 ;   // G1的总弧数为4
 // 以下定义顶点信息，VertexType为一结构
G1.vexs[0].no=1; G1.vexs[0].data="A";
G1.vexs[1].no=2; G1.vexs[1].data="B"
G1.vexs[2].no=3; G1.vexs[2].data="C";
G1.vexs[3].no=4; G1.vexs[3].data="D";
// 以下定义弧
G1.arcs[ ][ ].adj ={ 0,  1,   1,   0,
                     0,   0,   0,   0,
                     0,   0,   0,   1,
                     1,   0,   0,   0 };
```

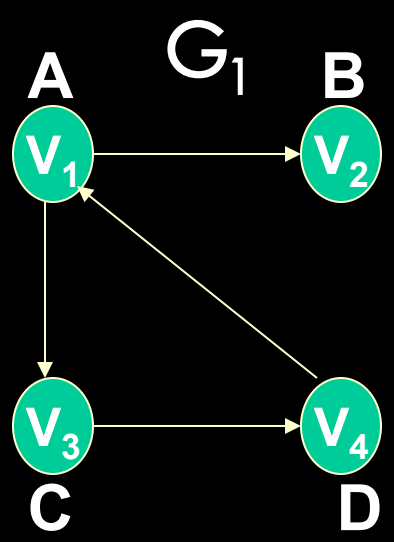


有向图G1

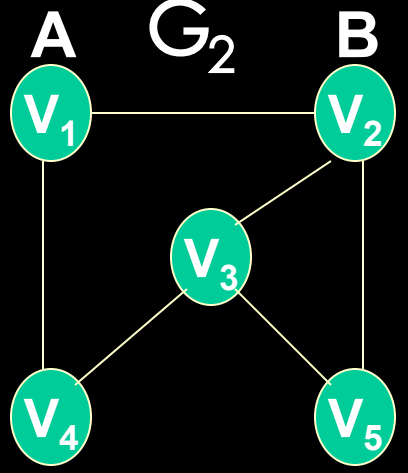


无向图G2

```
Mgraph  G2;      //定义图G2
G2.kind=AG;      // G2为无向图
G2.vexnum=5;  // G2的顶点数为5
G2.arcnum=6;   // G2的总弧数为6
 // 以下定义顶点信息，VertexType为一结构
G2.vexs[0].no=0; G2.vexs[0].data="A";
G2.vexs[1].no=1; G2.vexs[1].data="B"
G2.vexs[2].no=2; G2.vexs[2].data="C";
G2.vexs[3].no=3; G2.vexs[3].data="D";
// 以下定义弧（边）
G2.arcs[ ][ ].adj ={ 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 };
```



有向图G1



无向图G2

问题

如图采用邻接矩阵存储：

- 对于无向图，如何求解一个顶点的度？
- 对于有向图，如何求解一个顶点的入度？出度？
- 如何找某顶点的第一个邻接点**FirstAdjVex(G,u)**和下一个邻接点**NextAdjVex(G,v,w)**？

利用邻接矩阵构造图的算法描述：

1、输入图的顶点数、弧（边）数、是否含有弧（边）信息。

2、构造顶点信息：输入图的顶点名称。（顶点数组）

3、初始化邻接矩阵。

4、依次输入各条弧（边）：（构造邻接矩阵）

1、输入一条弧（边）所依附的两个顶点及权值；

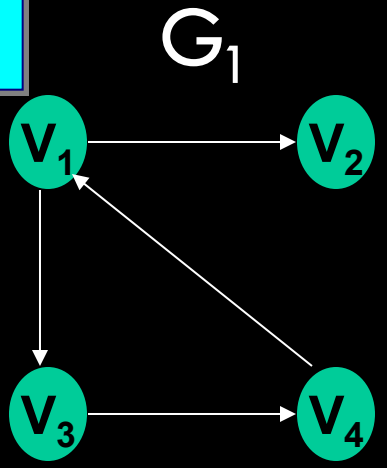
2、找出所输入顶点在顶点数组中的位置（数组下标 i,j）；

3、按 i ,j 修改邻接矩阵，如是无向的则对称修改；

4、输入弧的信息；

算法时间复杂度为 $O(\max(n^2, e*n))$

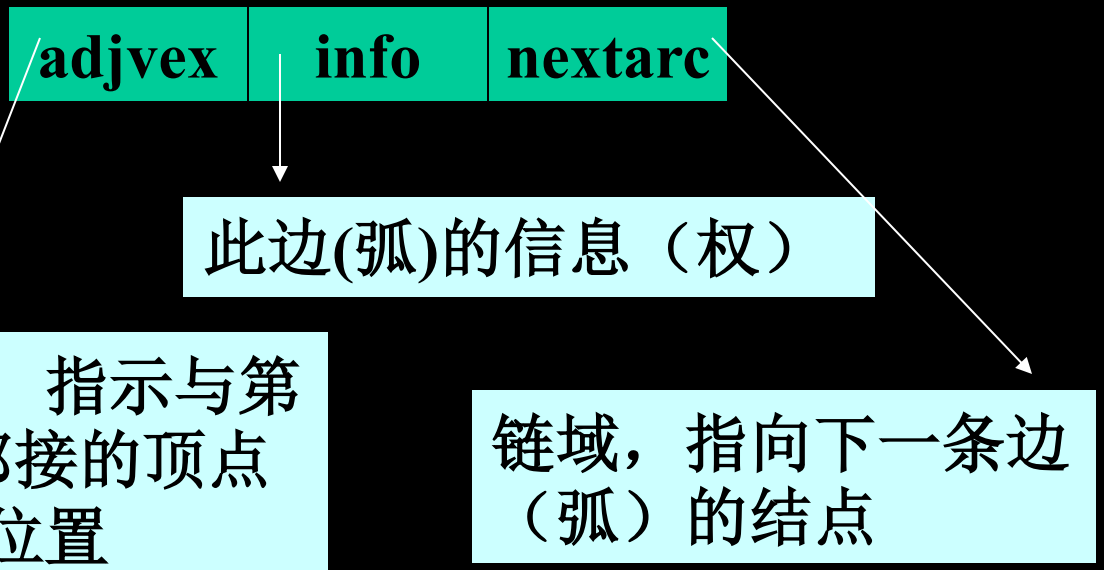
2、邻接表表示（图的链式存储结构）

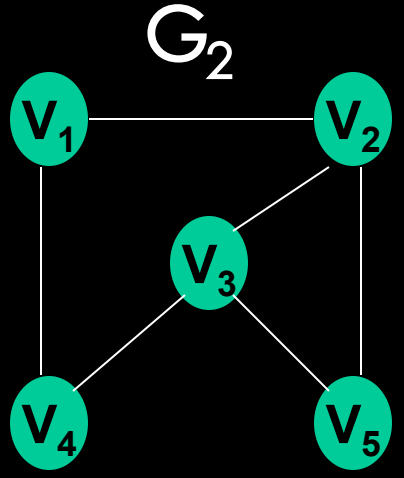


有向图G1

邻接表是图的一种链式存储结构，设计如下：

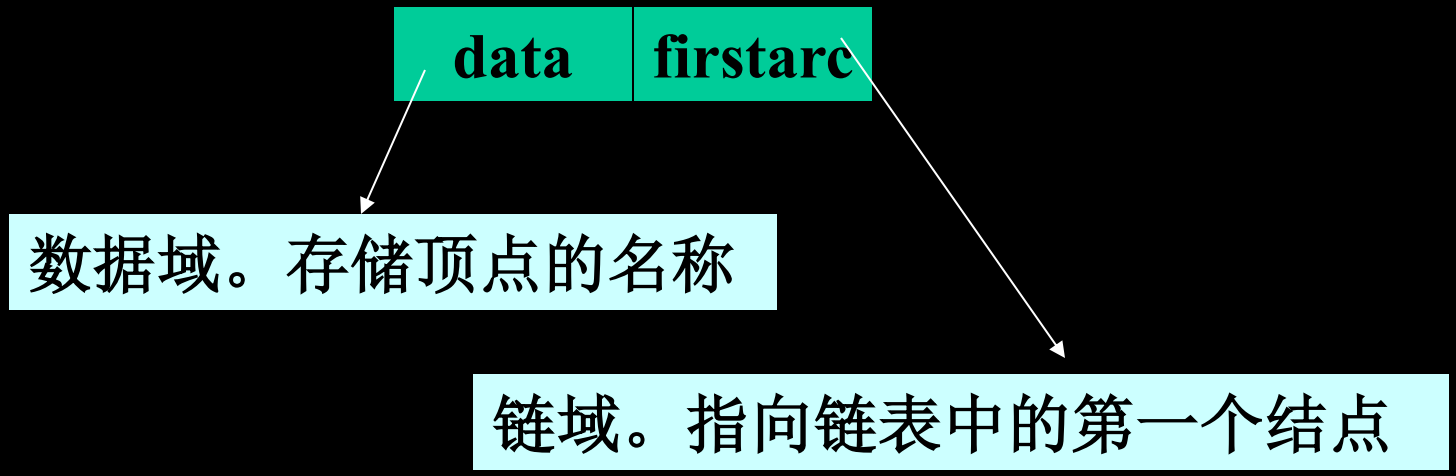
首先： 为图中每个顶点建立一个单链表，第i个单链表中的结点表示依附与第i个顶点的边（对有向图表示以第i个顶点为尾的弧）。每个结点结构为：



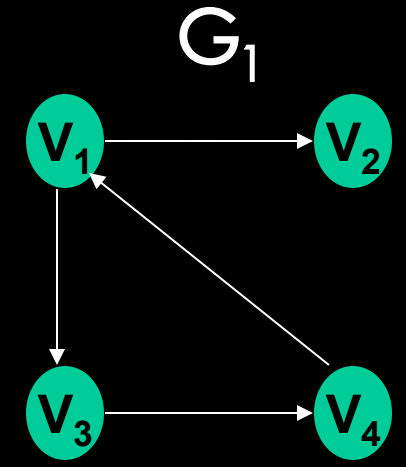


无向图G2

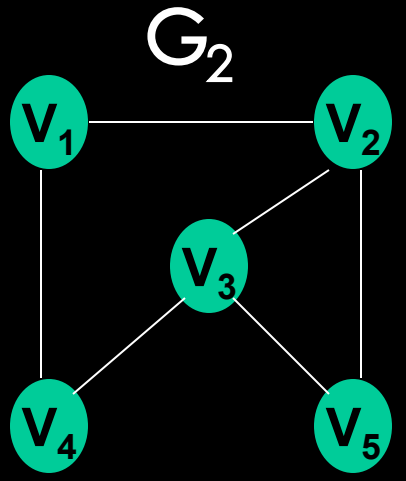
其次：为每个链表建立一个表头结点，其结构如下：



显然，对图中n个顶点的表头结点可以采用顺序存储结构。



有向图G1

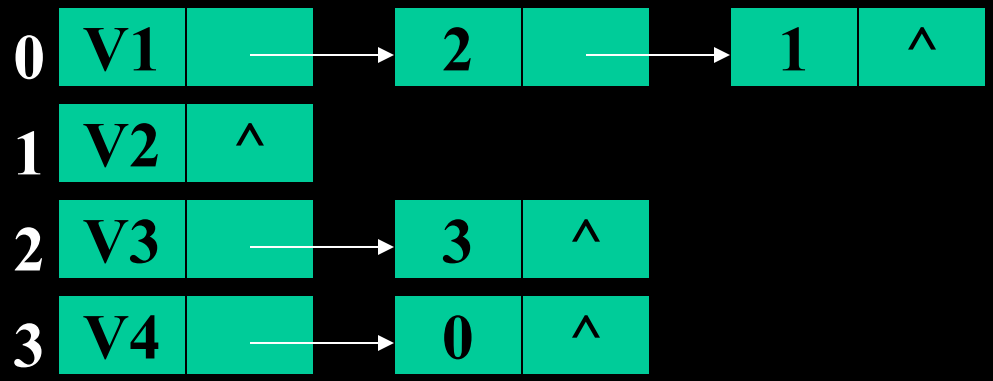


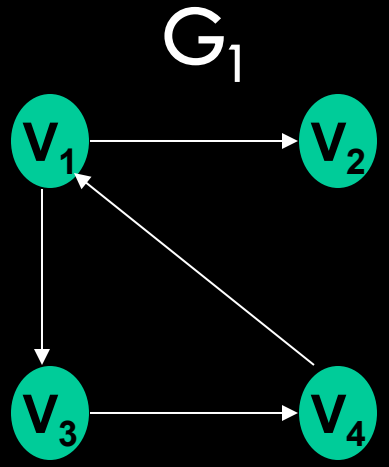
无向图G2

图的邻接表存储表示：

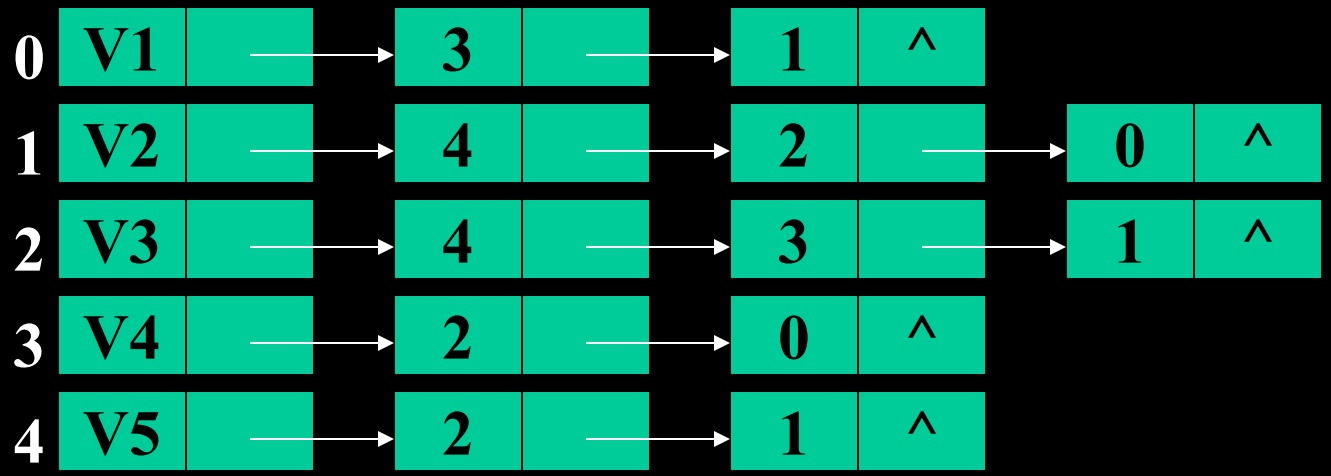
```
# define MAX_VERTEX   20    //顶点个数
# typedef  enum {DG,DN,AG,AN}  GraphKind; //图的种类
typedef  struct  ArcNode {  // 链表结点，实际上是存储图的边（弧）
      int                adjvex;            // 边（弧）所邻接的顶点的位置
      struct ArcNode     *nextarc;  // 指向下一条边（弧）
      InfoType       *info;                 //  弧的相关信息（ char,int ）
} ArcNode;
typedef struct VNode {      // 顶点数组
  VertexType     data;              //顶点信息
  ArcNode         *firstarc;        //指向第一个依附于该顶点的边（弧）
}  VNode,  AdjList[MAX_VERTEX];
typedef struct {
  AdjList         Vertices;    //顶点数组存放顶点信息
  int               vexnum,arcnum;          //图的顶点数与弧数
  GraphKind    kind;                          //图的种类
}  ALGraph;
```

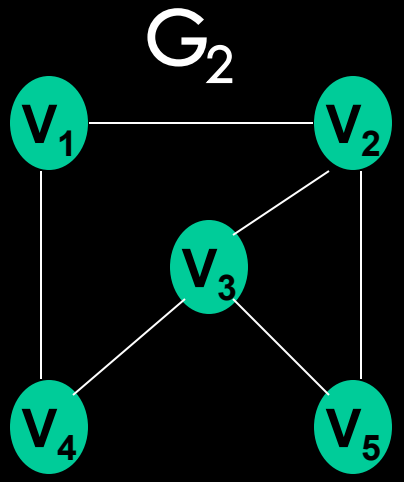
图G1和G2的邻接表





有向图G1





无向图G2

•逆邻接表的作用？

利用邻接表构造图的算法描述（算法6.2）

1、输入图的顶点数、弧（边）数。

2、输入顶点信息，并保存到顶点表G.vertices[].data中，并初始化每个顶点表头指针G.vertices[].firstac为NULL。

3、构造邻接表, 依次输入各条弧（边）：

1. 输入一条弧（边）所依附的两个顶点及权值；

2. 确定所输入顶点在顶点表中的位置（数组下标 i,j）；

3. 生成一个新的边结点，结点的adjvex域赋值为j，并将此结点插入到下标为i的链表中（头插）。

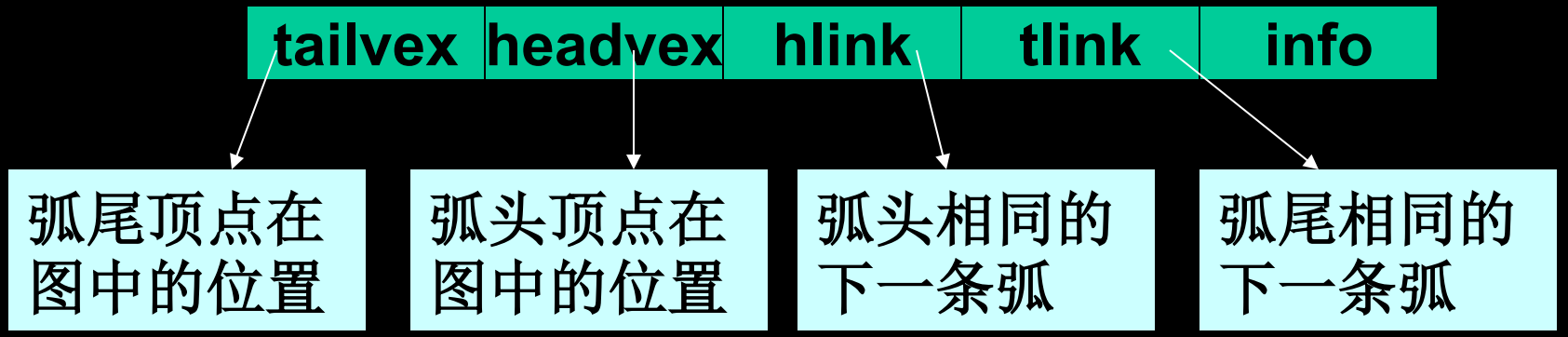
如果是无向图，则再生成一个新的边结点，结点的adjvex域赋值为i，并将此结点插入到下标为j的链表中（头插）。

算法时间复杂度为 O(e*n)

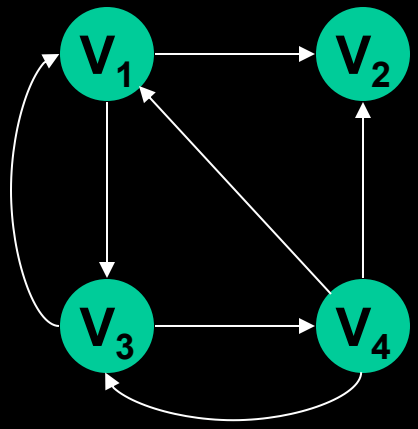
3、十字链表表示

十字链表是有向图的一种链式存储结构。设计如下：

对有向图的每一条弧建立一个结点，结构如下：

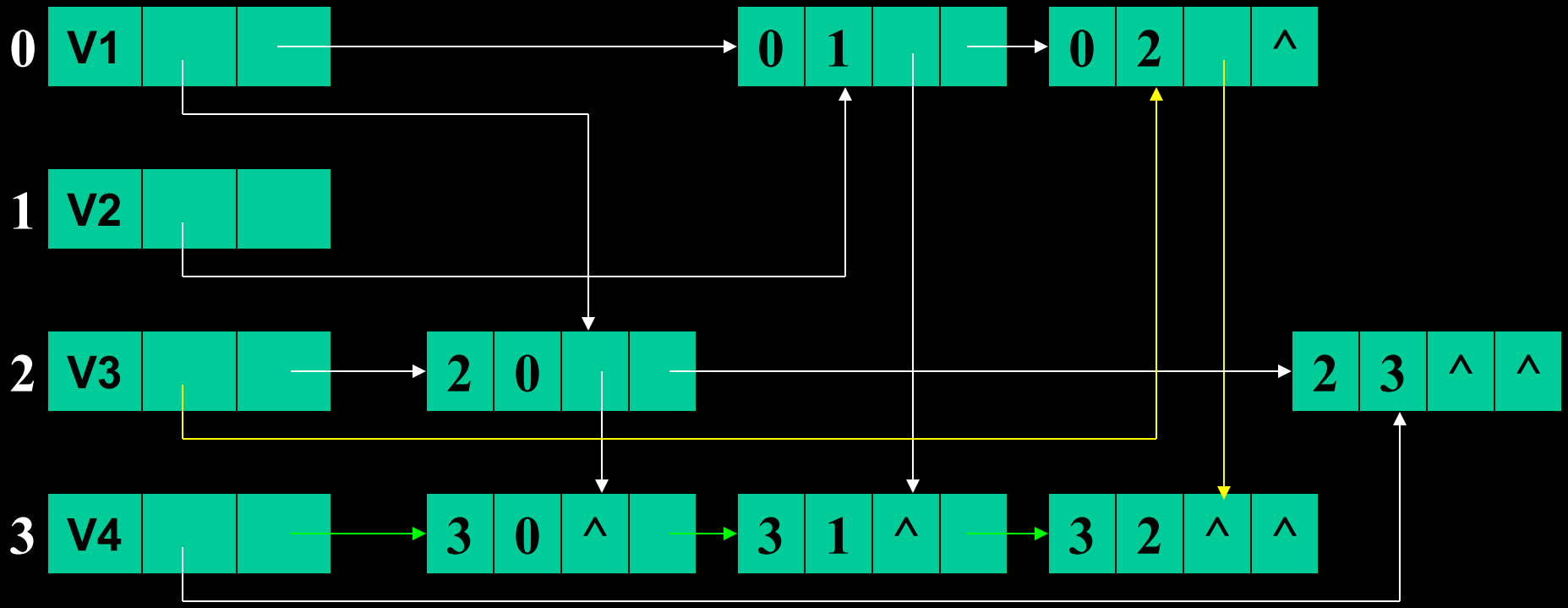


对有向图的每一个顶点建立一个结点，结构如下：



邻接矩阵：

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



有向图的十字链表存储表示：

```
# define MAX_VERTEX   20    //顶点个数
typedef  struct  ArcBox {  // 弧结点
        int                  tailvex , headvex;      // 弧的尾和头顶点的位置
        struct ArcBox        * hlink, * tlink;     // 弧头弧尾相同的指针域
        InfoType             * info;               //  弧的相关信息（ char,int ）
 } ArcBox;

 typedef struct VexNode {                  // 顶点结点
   VertexType     data;                    //顶点信息(char)
   ArcBox         *firstin, *firstout ;    //指向该顶点第一条入弧和出弧
 }  VexNode;

 typedef struct {
   VexNode        xList[MAX_VERTEX];    //顶点数组
   int            vexnum,arcnum;        //图的顶点数与弧数
}  ALGraph;
```

利用十字链表构造图的算法描述：

1、输入图的顶点数、弧（边）数、是否含有弧（边）信息。

2、构造顶点结点信息：（顶点数组）

输入图的顶点名称、初始化顶点的firstin、firstout指针。

3、依次输入各条弧（边）：（构造十字链表）

1、输入一条弧（边）所依附的两个顶点；

2、找出所输入顶点在顶点数组中的位置（数组下标 i,j）；

3、分配一个弧结点p，对p的各域进行赋值；

4、修改弧头顶点和弧尾顶点结点的相应域；即

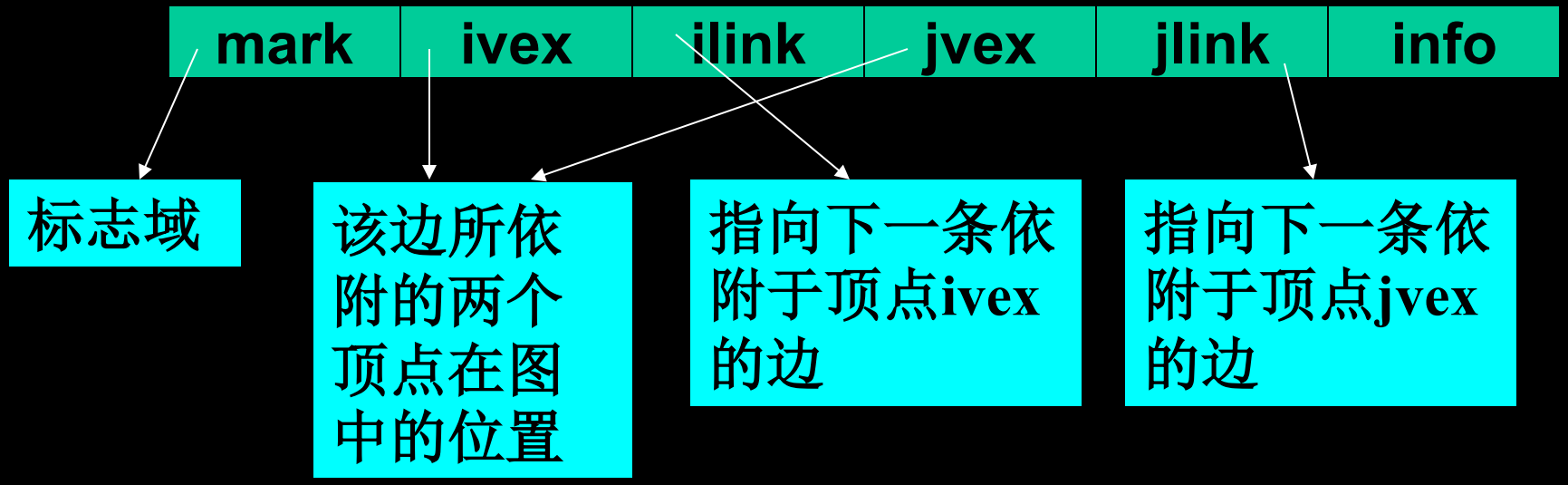
G.xlist[j].firstin=G.xlist[i].firstout=p

算法时间复杂度为 O(n+e)

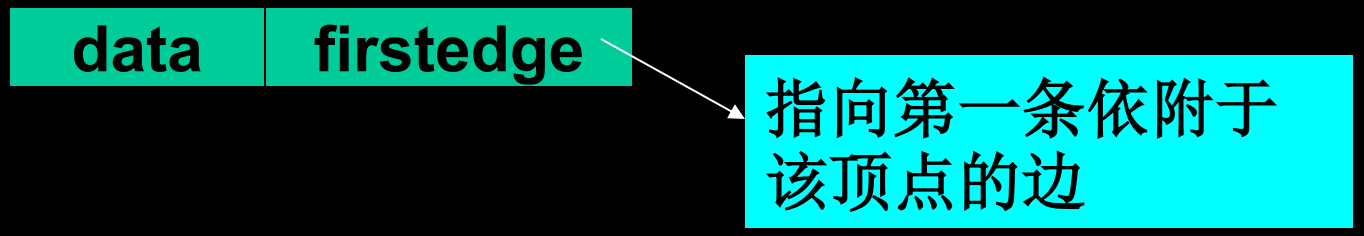
4、邻接多重表表示

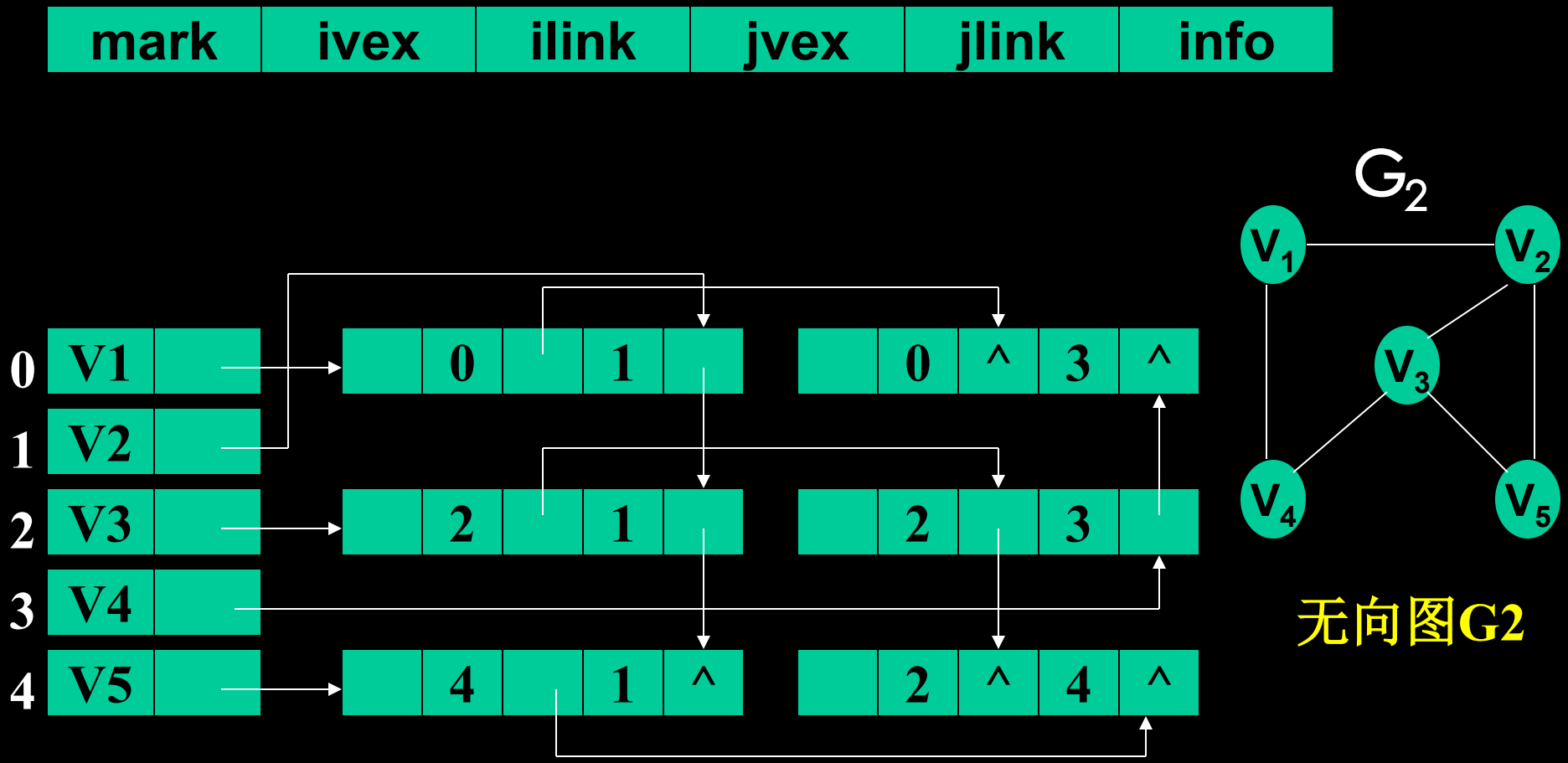
邻接多重表是无向图的一种链式存储结构。设计如下：

对无向图的每一条边建立一个结点，结构如下：



对无向图的每一个顶点建立一个结点，结构如下：





从图中可知：所有依附于同一顶点的边串联在同一链表中；

每一条边结点同时链接在两个链表中；

无向图的邻接多重表存储表示：

```
# define MAX_VERTEX   20    //顶点个数
typedef   emnu  { unvisited , visited }  VisitIf;
typedef  struct  EBox {  // 边结点
      VisitIf              mark;            // 访问标记
      int                  ivex , jvex;     // 边所依附的两个顶点位置
      struct EBox      * ilink, * jlink;  // 分别指向依附该顶点下一条边
      InfoType           * info;            // 边的相关信息 ( char,int )
} EBox;
typedef struct VexBox {                     // 顶点结点
  VertexType     data;                      // 顶点信息(char)
  EBox          *firstedge ;                // 指向依附于该顶点第一条边
}  VexBox;
typedef struct {
  VexBox        adjmulist[MAX_VERTEX];    //顶点数组
  int             vexnum,arcnum;            //图的顶点数与弧数
}  ALGraph;
```

6.5 图的遍历

从图的某个顶点出发访遍图中其余顶点，且使每个顶点仅被访问一次。

图的遍历是处理图的算法的基础。

由于图中的任意一个顶点都可能和其他顶点相邻接，所以在访问了某个顶点后，可能又会回到该顶点上。故必须记下每个已被访问过的顶点，可以设立一个数组visited[0..n-1]，其初始值为 0 ，一旦访问了顶点 V_i ，则置visited[i]=1 。

图的遍历常用的有两种：

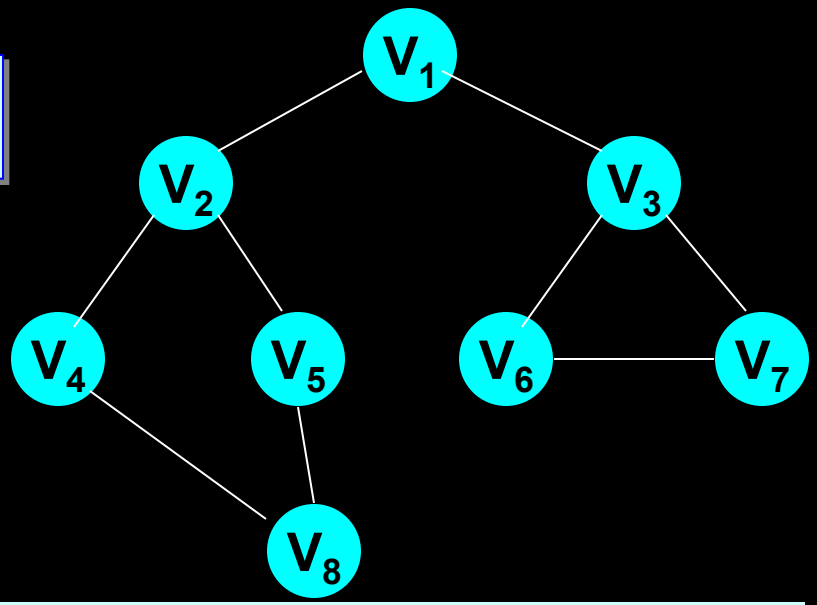
深度优先搜索、广度优先搜索。

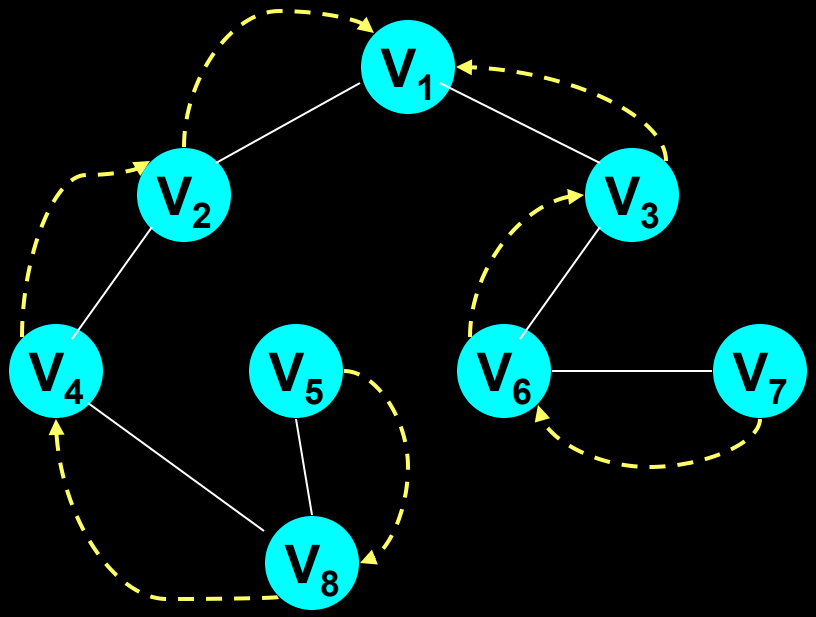
图的深度优先搜索：

设初始状态为图中所有顶点均未被访问，则遍历过程为：从图中某个顶点v_0出发，访问此顶点；然后选择一个与v_0相邻接但未被访问过的顶点v_i进行访问；再从v_i出发选择一个与v_i相邻且未被访问过的顶点v_j进行访问，依次类推。

一旦当前访问的顶点的所有邻接点都已被访问，则进行回溯，退回到前一个顶点，找此顶点其他一个相邻的未被访问的顶点，再继续遍历，直到图中所有和v_0有路径相通的顶点都被访问到；若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点为起始点，重复上述过程，直到图中所有顶点都被访问过。

深度优先搜索遍历过程：





以 邻接表 表示的图存储结构

1 | v1 => 2 -> 3 ^
2 | v2 => 4 -> 5 -> 1 ^
3 | v3 => 6 -> 7 -> 1 ^
4 | v4 => 2 -> 8 ^
5 | v5 => 2 -> 8 ^
6 | v6 => 3 -> 7 ^
7 | v7 => 6 -> 3 ^
8 | v8 => 4 -> 5 ^

顶点的访问序列为： $v_1, v_2, v_4, v_8, v_5, v_3, v_6, v_7$

深度优先搜索遍历算法：

```
int  visited[MAX];     //访问标志

void  DFSTraverse(Graph  G, int  v)  {
    for (v=0; v<G.vexnum; v++)  visited[v]=0;  //初始化数组
    for (v=0; v<G.vexnum; v++)
        if (!visited[v])   DFS(G,v);      //对未访问的顶点调用DFS
}
void  DFS(Graph  G , int v)  {  //从v出发递归调用深度优先遍历
    cout<<v; visited[v]=1; //访问第v个顶点，并标记已访问
    for  (w=FirstAdjVex(G,v); w ; w=NextAdjVex(G,v,w))
        if ( !visited[w])   DFS(G,w);
            //对v的尚未访问的邻接顶点w递归调用DFS
}
```

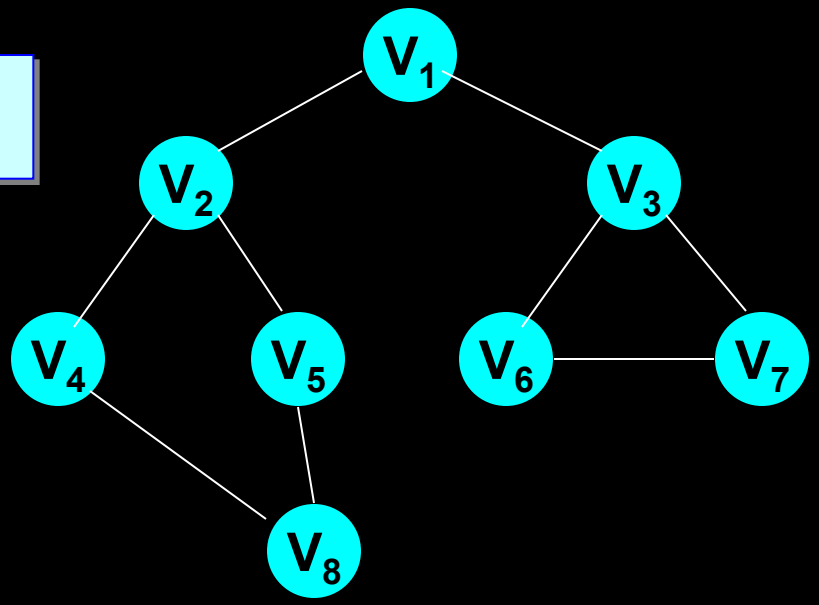
图的广度优先搜索：

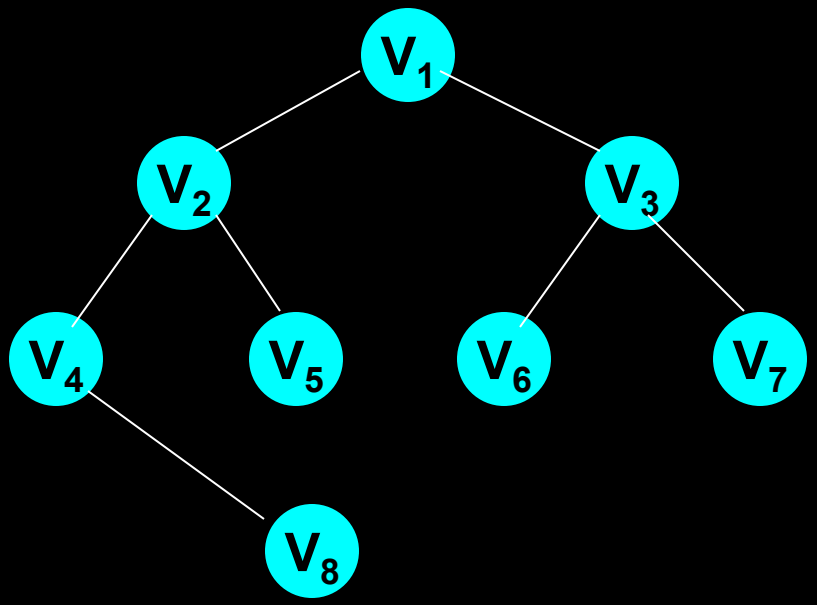
设初始状态为图中所有顶点均未被访问，则遍历过程为：从图中某个顶点v_i出发，访问此顶点；然后依次访问v_i的所有未被访问过的邻接点v_{i1}、v_{i2}、…、v_{ij}，再按照v_{i1}、v_{i2}、…、v_{ij}的次序，访问v_{ix}的所有未被访问的邻接点，直至所有已被访问的顶点的邻接点都被访问到。

一旦图中尚有顶点未被访问，则选择一个未访问的顶点作为起始点，重复上面过程。直到图中所有顶点都被访问过。

实质：以一个顶点v为起始点，由近至远，依次访问和v有路径相通且长度为1、2、3、…，的顶点。

广度优先搜索遍历过程：





以 邻接表 表示的图存储结构

1 | v1 => 2 -> 3 ^
2 | v2 => 4 -> 5 -> 1 ^
3 | v3 => 6 -> 7 -> 1 ^
4 | v4 => 2 -> 8 ^
5 | v5 => 2 -> 8 ^
6 | v6 => 3 -> 7 ^
7 | v7 => 6 -> 3 ^
8 | v8 => 4 -> 5 ^

顶点的访问序列为： $v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$

广度优先搜索遍历算法：

```
int  visited[MAX];     //访问标志
void  BFSTraverse(Graph  G, int  v)  {
  // 广度优先非递归遍历图G。使用辅助队列Q和标志数组visited。
   for (v=0; v<G.vexnum; v++)  visited[v]=0;  //初始化数组
   InitQueue(Q);              // 辅助队列 Q 置空
   for (v=0; v<G.vexnum; v++)
      if (!visited[v])   {
         cout<<v; visited[v]=1;EnQueue(Q,v);   // 顶点v入队列
         while  ( ! QueueEmpty(Q))   {
            DeQueue(Q,u);        //队头元素出队并赋值给u
            for  (w=FirstAdjVex(G,u); w ; w=NextAdjVex(G,u,w))
               if ( ! visited[w])  {     // w为u的尚未访问的邻接点
                  cout<<w; visited[w]=1; EnQueue(Q,w);  }  //if
          }    // while
       }    // if
}   // BFSTraverse
```

无向图的连通分量和生成树

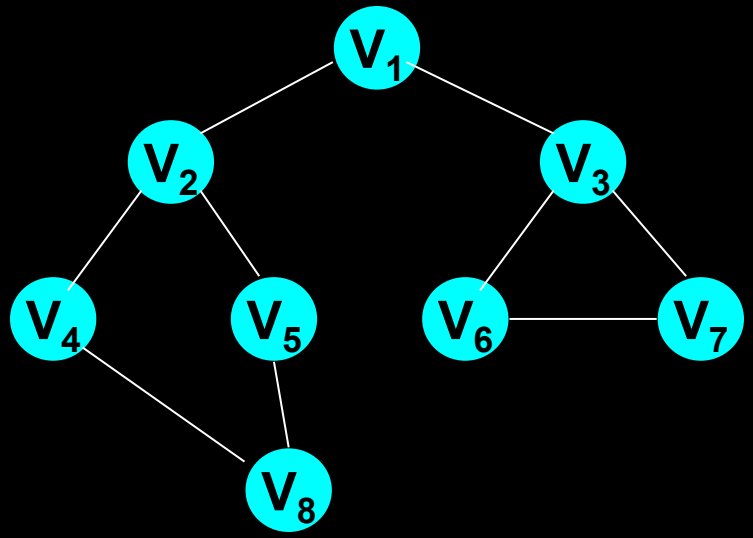
由图的遍历过程可知：

1、对连通图，只要从某一顶点出发进行搜索，就可访问图中的所有顶点；

2、对非连通图，需要从多个顶点出发进行搜索；每次从某个顶点出发进行搜索，所得到的顶点序列是图的一个连通分量的顶点集，若加上所有依附于这些顶点的边，即可构成图的一个连通分量。

3、设E(G)为一连通图的所有边的集合，则从任一顶点出发遍历时，必可将E(G)划分成两个子集设为T(G)和B(G)，其中T(G)为遍历过程中经过的边的集合，B(G)为剩余边的集合，则T(G)和图的所有顶点将构成一个连通图 G的极小连通子图，该子图为连通图G的一棵生成树（深度优先生成树、广度优先生成树）。

4、相应地、对于非连通图，遍历将产生多个连通分量，每个连通分量中的顶点和遍历时经过的边将构成多棵生成树；这些生成树组成非连通图的生成森林。



6.6 图的应用

6.6.1 最小生成树

问题：

在n个城市（电脑）间建立通信网，如何使得费用最省？

设以连通网来表示n个城市以及城市之间的通信路线，其中网的顶点表示城市，边表示两个城市间的线路，边的权值表示线路相应的代价。显然、对于 n 个顶点的连通网最多可以设置n(n-1)/2条边,而连通n个城市只需要n-1条线路，则问题可以转化为对n个顶点的连通网使用最少的边且总耗费最少并使得n个顶点相互连通，即如何从这些可能的线路中选择n-1条以使总耗费最少。

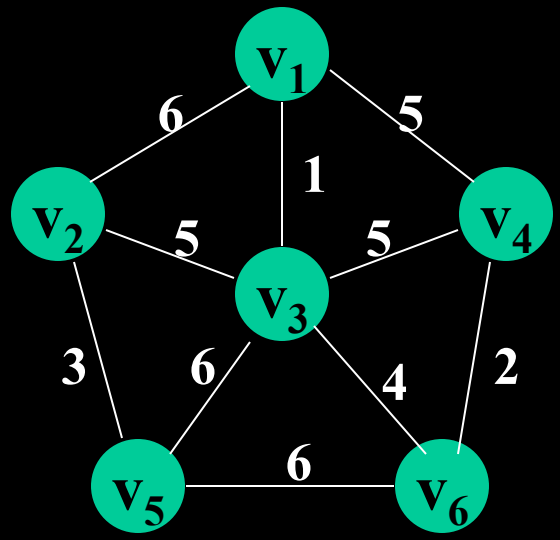
对于n个顶点的连通网可以建立多颗不同的生成树，每一棵生成树都可以是一个通信网，但总的耗费不同。如何选择（构造）一棵总耗费最少的生成树？

即构造连通网的最小代价（树上各边的代价之和）生成树（Minimum Cost Spanning Tree，简称最小生成树）。

MST的性质：

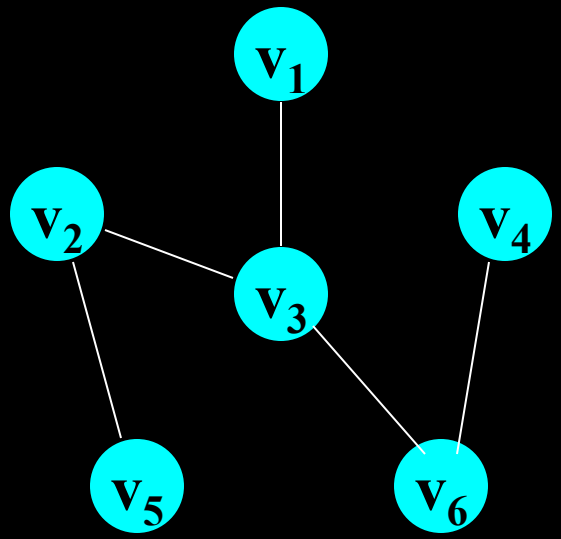
设N=(V , {E})是一个连通网，U是顶点集V的一个非空子集。若 (u , v) 是一条具有最小权值（代价）的边，其中 u∈U，v∈V – U，则必然存在一棵包含边 (u , v)的最小生成树。

普里姆（Prim）算法



设N = (V , {E})是连通图；U 为连通图N的最小生成树的顶点的集合，TE为最小生成树边的集合。

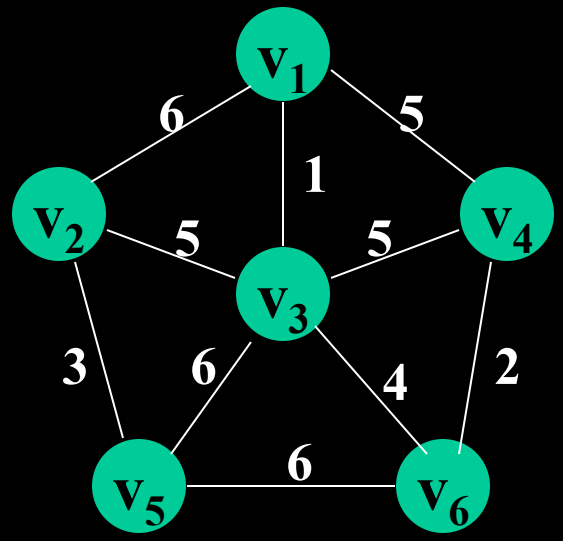
算法从U={ u_0 | u_0 ∈V}，TE={} 开始，重复执行以下操作：在所有u ∈U，v ∈V-U的边(u , v) ∈E中找一条代价最小的边(u_0 , v_0)并入集合TE，同时 v_0 并入集合U，直至U=V为止。



为实现普里姆算法，为了减少查找最小代价的边的过程，附设一个数组closedge[ ]，用以纪录从U到V-U具有最小代价的边，对每个顶点vi ∈V-U ，在辅助数组中存在一个相应分量closedge [i-1]，它包括两个域：lowcost和adjvex，则 closedge [i-1]. lowcost中存放的是边（u , v_i）（u ∈ U）上最小的权值， closedge [i-1]. adjvex中存放的是该边依附的在U中的顶点。一旦顶点vi并入集合U，则closedge [i-1]. lowcost的值改为0。

```
void MiniSpanTree_PRIM(MGraph G, VertexType u){
  struct{
        VertexType adjvex;
        VRType     lowcost;
   }closedge[MAX_VERTEX_NUM];      //定义辅助数组
   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].adj};
   closedge[k].lowcost=0;          //初始, U={u}
   for(i=1; i<G.vexnum; ++i){   //选择其余G.vexnum-1个顶点
       k=mininum(closedge);        //求权值最小的顶点
       printf(closedge[k].adjvex, G.vexs[k]); //输出生成树的边
       closedge[k].lowcost=0;      //第k顶点并入 U 集
       for(j=0; j<G.vexnum; ++j)
           if(G.arcs[k][j].adj < closedge[j].lowcost)
               //新顶点并入 U 集后,重新选择最小代价的边
                closedge[j]={G.vexs[k],G.arcs[k][j].adj};
       }
}//MiniSpanTree        T(n)=O(n²)
```

1



U={v_1}　V-U={v_2,v_3,v_4,v_5,v_6}

Closedge[0].lowcost=0

Closedge[1].lowcost=6　　adjvex=v_1

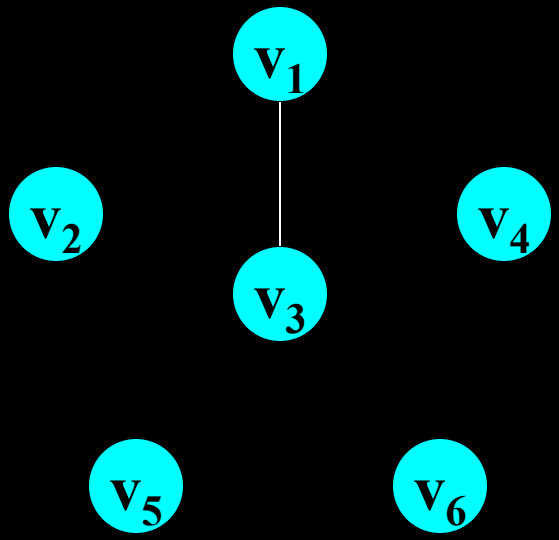
Closedge[2].lowcost=1　　adjvex=v_1

Closedge[3].lowcost=5　　adjvex=v_1

Closedge[4].lowcost=

Closedge[5].lowcost=

2



U={v_1,v_3}　V-U={v_2, v_4,v_5,v_6}

Closedge[0].lowcost=0

Closedge[1].lowcost=6 - 5　　adjvex=v_3

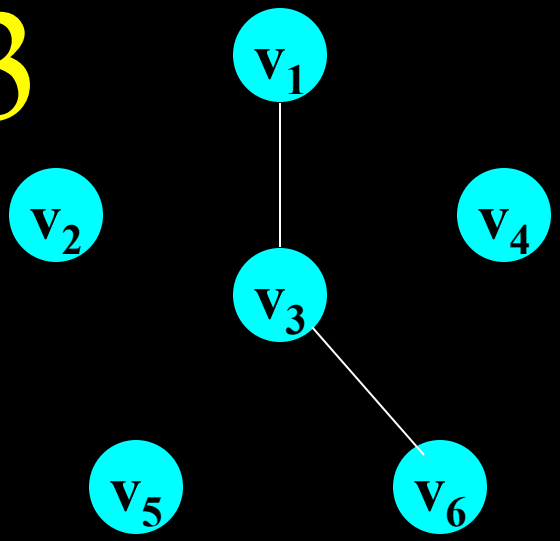
Closedge[2].lowcost=1 - 0　　adjvex=v_1

Closedge[3].lowcost=5　　adjvex=v_1

Closedge[4].lowcost= - 6　　adjvex=v_3

Closedge[5].lowcost= - 4　　adjvex=v_3

3



U={v_1,v_3 ,v_6} V-U={v_2, v_4,v_5}

Closedge[0].lowcost=0

Closedge[1].lowcost=5 adjvex=v_3

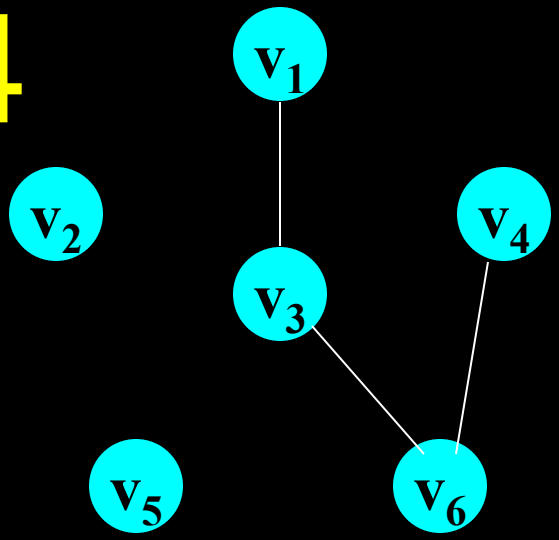
Closedge[2].lowcost=0 adjvex=v_1

Closedge[3].lowcost=5 - 2 adjvex=v_6

Closedge[4].lowcost=6 adjvex=v_3

Closedge[5].lowcost=4 - 0 adjvex=v_3

4



U={v_1,v_3 ,v_6 , v_4} V-U={v_2,v_5}

Closedge[0].lowcost=0

Closedge[1].lowcost=5 adjvex=v_3

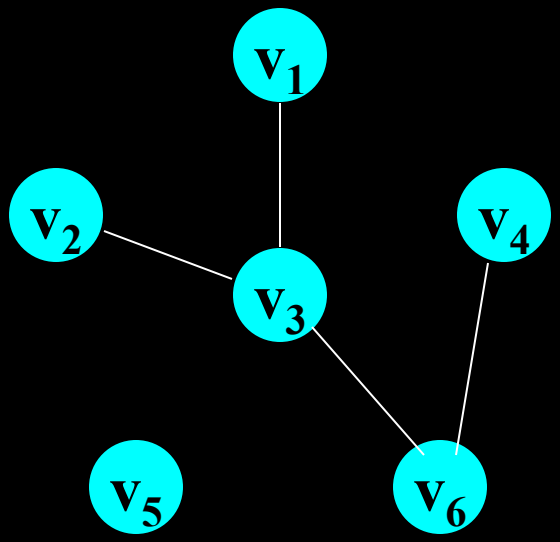
Closedge[2].lowcost=0 adjvex=v_1

Closedge[3].lowcost=2 - 0 adjvex=v_6

Closedge[4].lowcost=6 adjvex=v_3

Closedge[5].lowcost=0 adjvex=v_3

5



U={v_1,v_3 ,v_6 , v_4 ,v_2} V-U={v_5}

Closedge[0].lowcost=0

Closedge[1].lowcost=5 - 0 adjvex=v_3

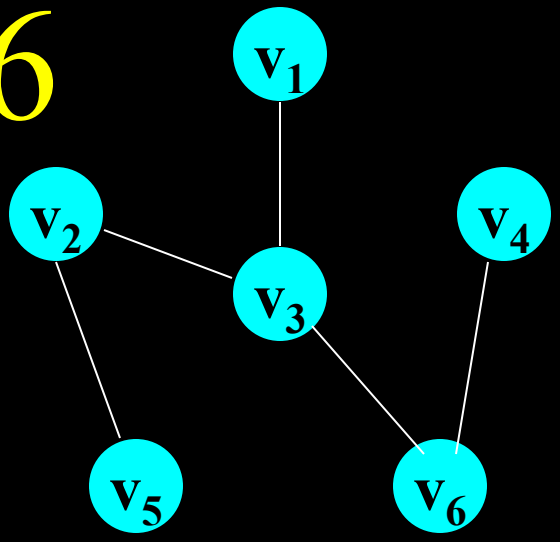
Closedge[2].lowcost=0 adjvex=v_1

Closedge[3].lowcost=0 adjvex=v_6

Closedge[4].lowcost=6 - 3 adjvex=v_2

Closedge[5].lowcost=0 adjvex=v_3

6



U={v_1,v_3 ,v_6 , v_4 ,v_2 ,v_5} V-U={}

Closedge[0].lowcost=0

Closedge[1].lowcost=0 adjvex=v_3

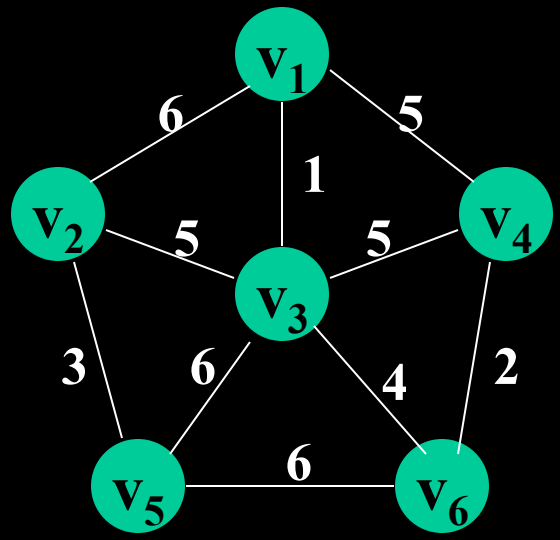
Closedge[2].lowcost=0 adjvex=v_1

Closedge[3].lowcost=0 adjvex=v_6

Closedge[4].lowcost=6 - 0 adjvex=v_2

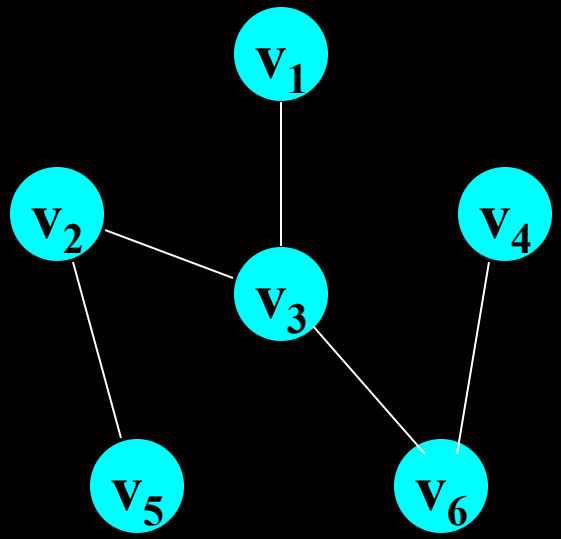
Closedge[5].lowcost=0 adjvex=v_3

克鲁斯卡尔算法



设N = (V , {E})是连通图；则连通图 N 的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。

从E中选择一条代价最小的边，若该边依附的顶点落在T中两个不同的连通分量上，则将此边加入到T中，否则舍去此边选择下一条代价最小的边。依次类推，直到T中所有顶点都在同一个连通分量上为止。



思考：

普里姆算法与克鲁斯卡尔算法有何实质区别？

6.6.2 最短路径

1、有向图的最短路径与有向网的最短路径；

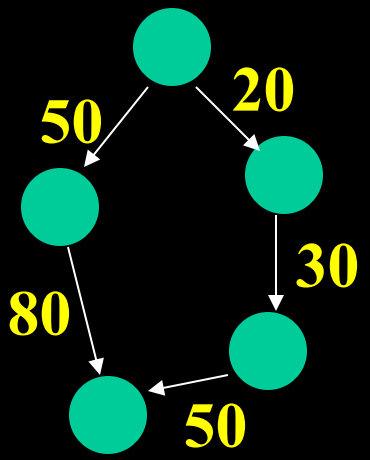
说明：不带权图的最短路径问题是带权图的最短路径问题的一个特例，可将图视为每条弧的权值均为1的带权图。

2、最短路径：从一个顶点（源点）到另一个顶点（终点）路径上所经过的弧的权值之和（路经长度）最小的那条路径。

3、两种最常见的最短路径问题：

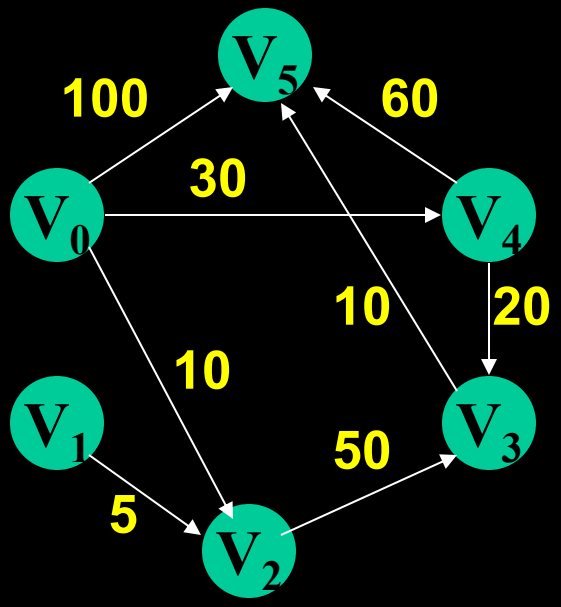
从某个源点到其余各顶点的最短路径

每对顶点间的最短路径



7.6.1 从某个源点到其余各顶点的最短路径

对于一个有n个顶点和e条弧的图G，从某一顶点（Vi）到其余任一顶点Vj 的最短路径，可能是弧 <Vi , Vj>，或是经过 k 个（1≤k≤ n-2）中间顶点和 k+1 条弧所形成的路径。



V0 → V2	(V0,V2)	10
V0 → V3	(V0,V4,V3)	50
V0 → V4	(V0,V4)	30
V0 → V5	(V0,V4,V3,V5)	60

迪杰斯特拉算法

基本思想:

按照从源点到其余每一顶点的最短路径长度递增的次序依次求出从源点到各顶点的最短路径及长度，每次求出从源点到一个终点m的最短路径后，都要以该顶点m作为新考虑的中间点，对那些尚未求出最短路径的终点的当前最短路径做相应的修改，当进行n-1次后结束。

算法需要设立一个集合S，用以保存已求得最短路径的终点，其初值为只有一个元素，即源点。

设立一个数组D[ ]，其每个分量D[j]保存当前从源点到该顶点 j 的最短路经长度。其初值为若源点到该终点有弧，则为弧的权值，否则置为∞。

另外，再设立一个指针数组path[n]，path[j]指向一个单链表，保存相应于D[j]的从源点到顶点 j 的最短路径（即顶点序列）。初值为空。

Dijkstra算法

① 初始化： $S \leftarrow \{ v_0 \}$;

D[j] ← G.arcs[0][j]， j = 1, 2, …, n-1;

path[j] ← <0, j>; // n为图中顶点个数

② 求出最短路径的长度：

D[k] ← min { D[i] }, $i \in V- S$;

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

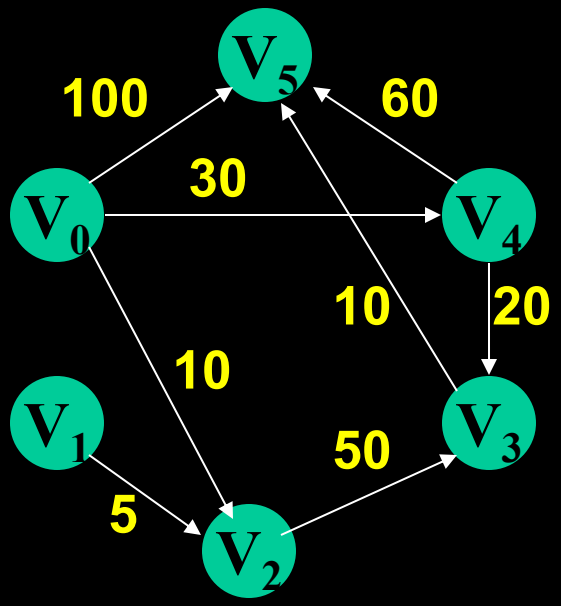
③ 修改：

D[i] ← min{ D[i], D[k] + G.arcs[k][i] },

若 D[i] 修改了，则 path[i] ← path[k] 加上 i

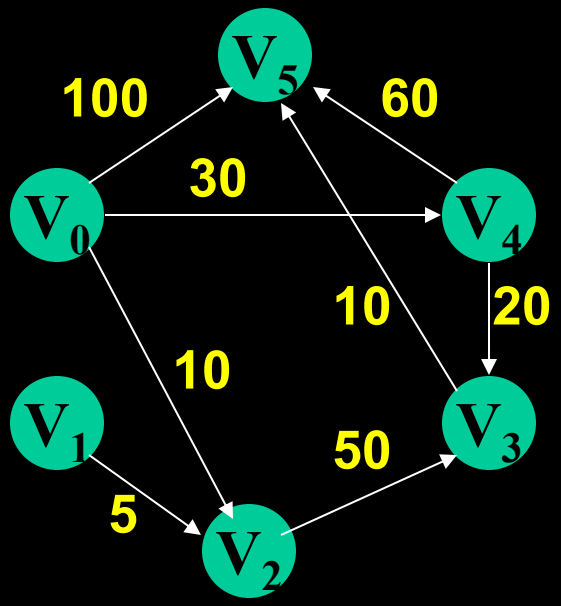
对于每一个 $i \in V- S$;

④ 判断：若 S = V, 则算法结束，否则转 ②。



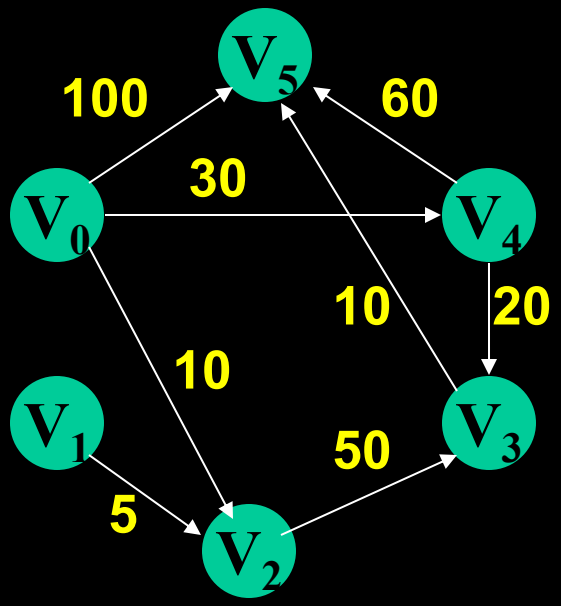
$$\begin{pmatrix} \infty & \infty & 10 & \infty & 30 & 100 \\ \infty & \infty & 5 & \infty & \infty & \infty \\ \infty & \infty & \infty & 50 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & 20 & \infty & 60 \\ \infty & \infty & \infty & \infty & \infty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

	0	1	2	3	4	5
S	1	0	0	0	0	0
D	0	∞	10	∞	30	100
P			v0,v2		v0,v4	v0,v5



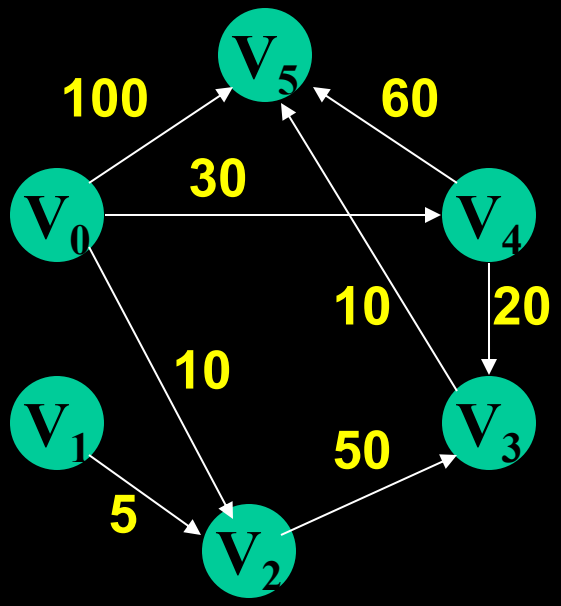
$$\begin{pmatrix} \infty & \infty & 10 & \infty & 30 & 100 \\ \infty & \infty & 5 & \infty & \infty & \infty \\ \infty & \infty & \infty & 50 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & 20 & \infty & 60 \\ \infty & \infty & \infty & \infty & \infty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

	0	1	2	3	4	5
S	1	0	1	0	0	0
D	0	∞	10	60	30	100
P			v_0,v_2	v_0,v_2,v_3	v_0,v_4	v_0,v_5



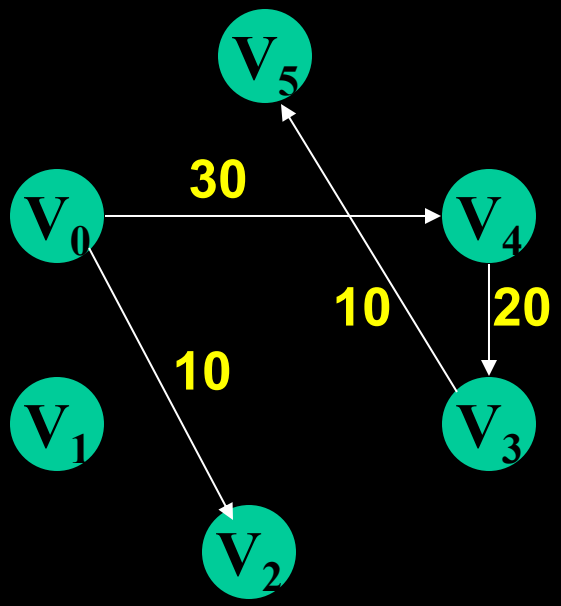
$$\begin{pmatrix} \infty & \infty & 10 & \infty & 30 & 100 \\ \infty & \infty & 5 & \infty & \infty & \infty \\ \infty & \infty & \infty & 50 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & 20 & \infty & 60 \\ \infty & \infty & \infty & \infty & \infty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

	0	1	2	3	4	5
S	1	0	1	0	1	0
D	0	∞	10	50	30	90
P			v_0,v_2	v_0,v_4,v_3	v_0,v_4	$v_0,v_4\ v_5$



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

	0	1	2	3	4	5
S	1	0	1	1	1	0
D	0	∞	10	50	30	60
P			v_0,v_2	v_0,v_4,v_3	v_0,v_4	$v_0v_4\ v_3\ v_5$


V5
30
V0
V4
10
20
10
V1
V3
V2

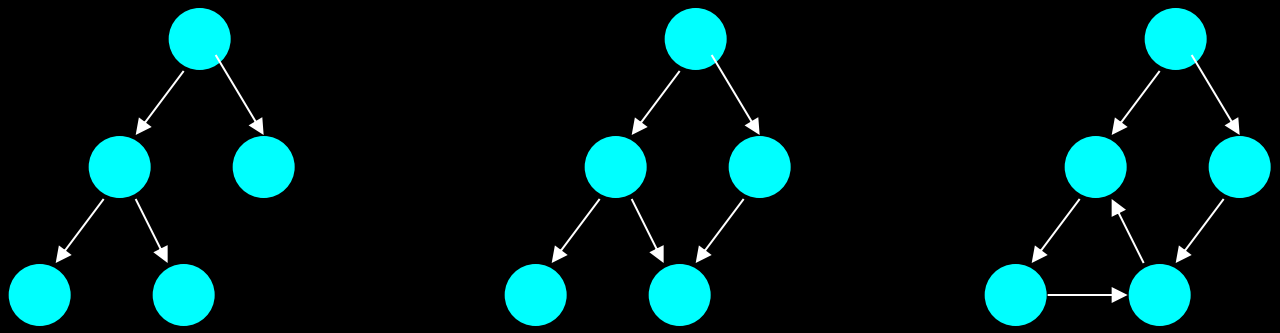
终点	从v0到各终点的D值和最短路径的求解过程				
	i=1	i=2	i=3	i=4	i=5
v1	∞	∞	∞	∞	∞ 无
v2	**10** **(v0,v2)**				
v3	∞	60 (v0,v2,v3)	**50** **(v0,v4,v3)**		
v4	30 (v0,v4)	**30** **(v0,v4)**			
v5	100 (v0,v5)	100 (v0,v5)	90 (v0,v4.v5)	**60** **(v0,v4,v3,v5)**	
v_j	v2	v4	v3	v5	
S	{v0,v2}	{v0,v2,v4}	{v0,v2,v3,v4}	{v0,v2,v3,v4,v5}	

6.6.3 有向无环图及其应用

无环的有向图称为有向无环图 (Directed Acycline Graph), 简称DAG图。是一类较有向树更一般的特殊有向图。

有向无环图的应用？

- 描述含有公共子式的表达式；
- 描述工程或系统的进行过程：施工图、生产流程图、排课；(顺利进行、最短时间)



有向树　　有向无环图　　有向图

拓扑排序

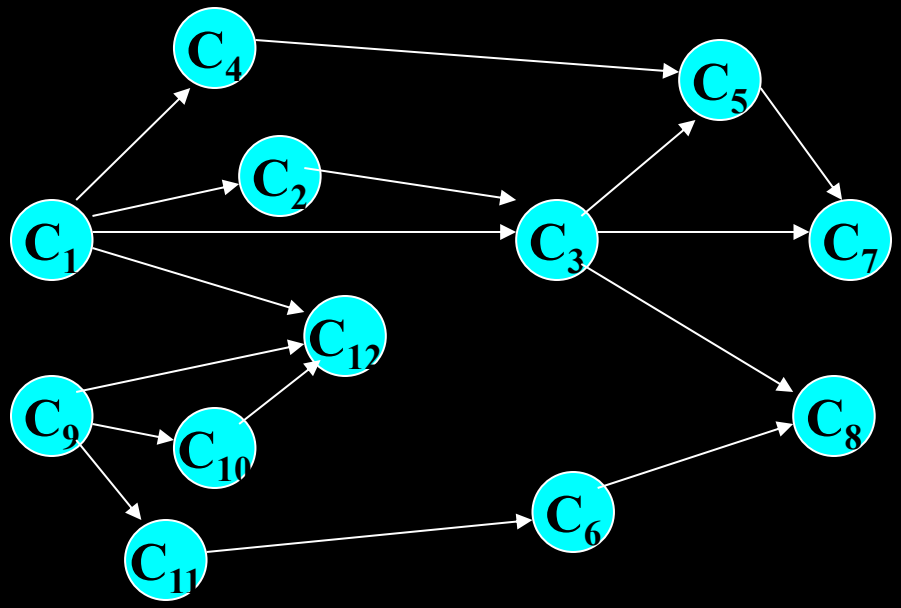
AOV-网：（Activity On Vertex Network）

用顶点表示活动（Activity），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。

如：p180 课程的学习。

显然，在AOV-网中不应该出现有向环，即不可能存在一条回路。如果这样，则所有活动可排列成一个线性序列，使得每个活动的所有前驱活动都排列在该活动的前面。则称此序列为拓扑序列；由AOV-网构造拓扑序列的过程则称作拓扑排序。

如果对给定的AOV网要检测网中是否存在环，则检测的方法就是对有向图构造其顶点的拓扑序列，若网中所有顶点都在其拓扑序列中，则该AOV网必定不存在环。



1 C1 , C2 , C3 , C4 , C5 , C7 , C9 , C10 , C11 , C6 , C12 , C8

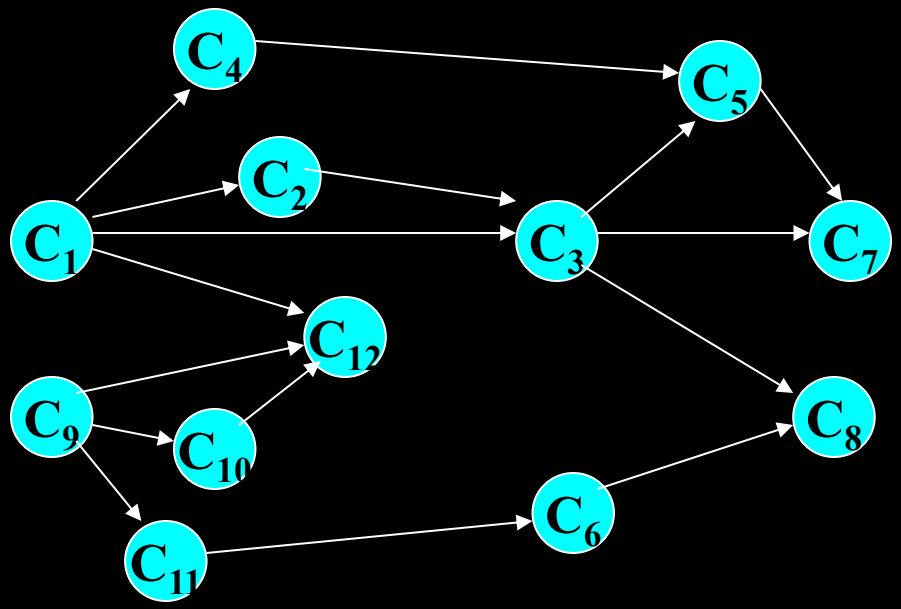
2 C9 , C10 , C11 , C6 , C1 , C12 , C4 , C2 , C3 , C5 , C7 , C8

那么，如何从AOV网构造拓扑序列？

1、在有向图中选择一个没有前驱的顶点且输出；

2、从图中删除该顶点和所有以它为尾的弧；

3、重复上述两步，直到全部顶点均已输出，或者图中没有无前驱的顶点为止（意味着存在环）。



C1 , C2 , C3 , C4 , C5 , C7 , C9 , C10 , C11 , C6 , C12 , C8

拓扑排序算法实现

设以**邻接表**为有向图的存储结构，且在其头结点中增加一个**存放该顶点入度的数据域 indegree**。则 indegree=0 的顶点即为没有前驱的顶点。此时，为删除入度为 0 的顶点为尾的弧，只需**将这些弧的弧头顶点的入度减 1**。另一方面，为避免重复检测入度为 0 的顶点，设立一个辅助栈存储入度为 0 的顶点编号。**算法描述如下：**

1、查邻接表中入度为 0 的顶点，并入栈；

2、当栈非空时，进行拓扑排序：

① 栈顶顶点v_i 出栈，并输出；

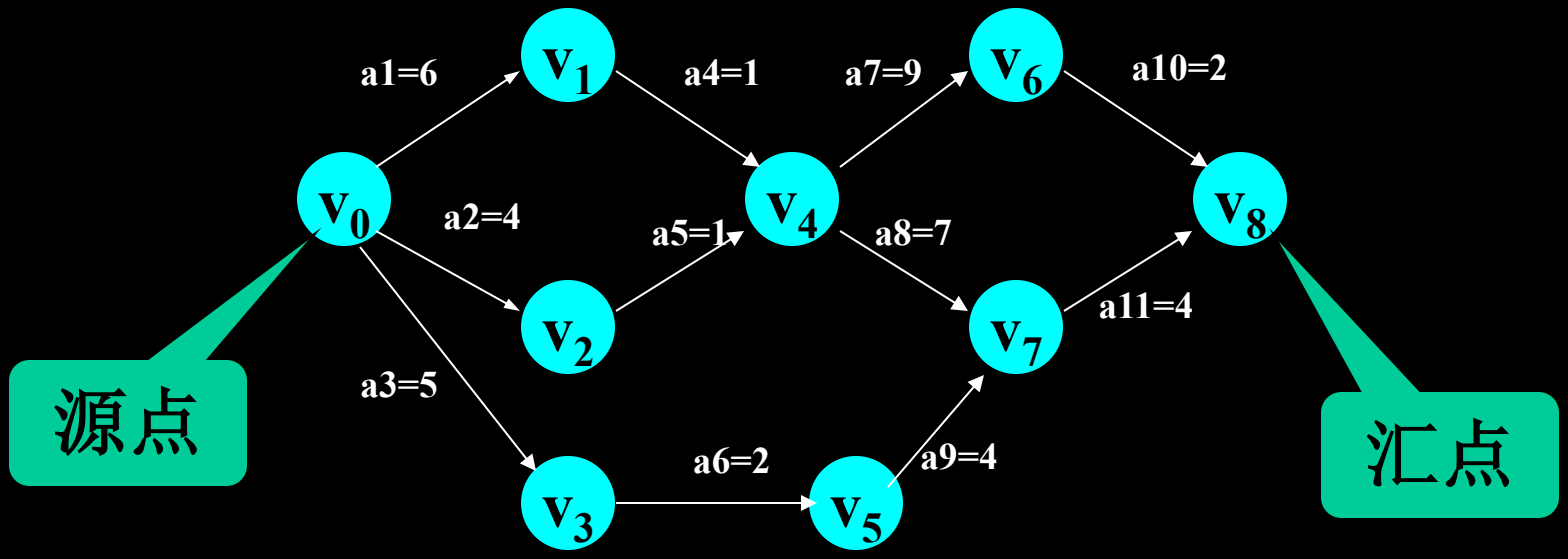
② 在邻接表中查找 v_i 的全部直接后继v_j，将v_j 的入度减 1，并将入度减至 0 的顶点入栈；

3、当栈空时，若输出的顶点数不足有向图的顶点数，则说明存在环，否则拓扑排序完成。

关键路径

AOE-网：（Activity On Edge Network）

边（edge）表示活动（Activity）的网，是一个带权的有向无环图。其中，顶点表示事件（Event），弧表示活动，权表示活动持续的时间。AOE网常用于估算工程完成时间。



利用AOE网可解决的问题：

1、工程是否可行？

2、完成整项工程至少需要多少时间？

3、哪些活动是影响工程进度的关键？

关键路径：从源点到汇点的路径长度最长的路径。

事件 v_j 的最早发生时间：从源点(v_0)到 v_j 的最长路经长度。记为 ve(j)。显然这个时间决定了所有以v_j 为尾的弧所表示的活动 a_i 的最早开始时间，记为 e(i)，e(i)=ve(j)；

事件 v_j 的最迟发生时间：在不推迟整个工程完成的前提下，事件 v_j 最迟必须发生的时间。记为 vl(j)。显然这个时间决定了所有以v_j 为头的弧所表示的活动 a_i 的最迟开始时间，记为 l(i)，l(i)=vl(j) - dut(a_i)；(活动 a_i 的持续时间)

关键活动：若活动 a_i 的 l(i) = e(i)，称 a_i 为关键活动。

显然，关键路径上的活动均为关键活动。分析关键路径的目的就是辨别哪些是关键活动，以便通过提高关键活动的效率来缩短整个工期。

要辨别哪些是关键活动就是找满足 l(i) = e(i) 这个条件的活动。而为了求得AOE网中活动的 l(i) 和 e(i)，首先应求得事件的最早发生时间 ve(j) 和 最迟发生时间 vl(j)。

- ve(j)=max{ve(k)+dut(<k,j>)}　　$<k , j>\in T , 2\le j\le n$

  其中： T为所有以j为头的弧的集合。

- vl(j)=min{vl(k) - dut(<j,k>)}　　$<j , k>\in S , 1\le j\le n-1$

  其中： S为所有以j为尾的弧的集合。

由此可见：求ve(i)和vl(i)必须分别按拓扑有序和逆拓扑有序下进行计算。则求关键路径的算法描述如下：

1、输入e条弧<j,k>，建立AOE网的存储结构；

2、从源点v_0出发，令ve[0]=0，按拓扑有序求其余各顶点的最早发生时间ve[i] (1 ≤i≤n-1)；如果得到拓扑有序序列中顶点个数小于网中顶点数n ，则说明存在环，算法中止；否则执行步骤 3 。

3、从汇点v_n出发，令vl[n-1]=ve[n-1]，按逆拓扑有序求其余各顶点的最迟发生时间vl[i] (n-2≥i≥1)；

4、根据各顶点的ve和vl值，求每条弧s的最早开始时间e(s)和最迟开始时间l(s)。若弧满足e(s)=l(s)，则该弧为关键活动。

按拓扑有序计算得到每个事件的最早发生时间：

ve(0)=0

Ve(1)=ve(0)+dut(<0,1>)=0+6=6

Ve(2)= ve(0)+dut(<0,2>)=0+4=4

Ve(3)= ve(0)+dut(<0,3>)=0+5=5

ve(4)=max{ve(1)+dut(<1,4>),ve(2)+dut(<2,4>)}=max{6+1,4+1}=7

Ve(5)=ve(3)+dut(<3,5>)=5+2=7

Ve(6)=ve(4)+dut(<4,6>)=7+9=16

Ve(7)=max{ve(4)+dut(<4,7>),ve(5)+dut(<5,7>)}=14

Ve(8)=max{ve(6)+dut(<6,8>),ve(7)+dut(<7,8>)}=18

按逆拓扑序列计算得到每个事件的最迟发生时间：

Vl(8)=ve(8)=18

Vl(7)=vl(8)-dut(<7,8>)=18-4=14

Vl(6)=vl(8)-dut(<6,8>)=18-2=16

Vl(5)=vl(7)-dut(<5,7>)=14-4=10

Vl(4)=min{vl(7)-dut(<4,7>),vl(6)-dut(4,6)}=min{14-7,16-9}=7

Vl(3)=vl(5)-dut(<3,5>)=10-2=8

Vl(2)=vl(4)-dut(<2,4>)=7-1=6

Vl(1)=vl(4)-dut(<1,4>)=7-1=6

Vl(0)=min{vl(1)-dut(<0,1>),vl(2)-dut(<0,2>),vl(3)-dut(<0,3>)}=0

顶点	ve	vl	活动	e	l	l-e
v0	0	0	a1	0	0	0
v1	6	6	a2	0	2	2
v2	4	6	a3	0	3	3
v3	5	8	a4	6	6	0
v4	7	7	a5	4	6	2
v5	7	10	a6	5	8	3
v6	16	16	a7	7	7	0
v7	14	14	a8	7	7	0
v8	18	18	a9	7	10	3
			a10	16	16	0
			a11	14	14	0

关键活动为：a1,a4,a7,a8,a10,a11.

构成两条关键路径：（v0,v1,v4,v7,v8)和(v0,v1,v4,v6,v8)