# 4.1 基本定义(cont.)

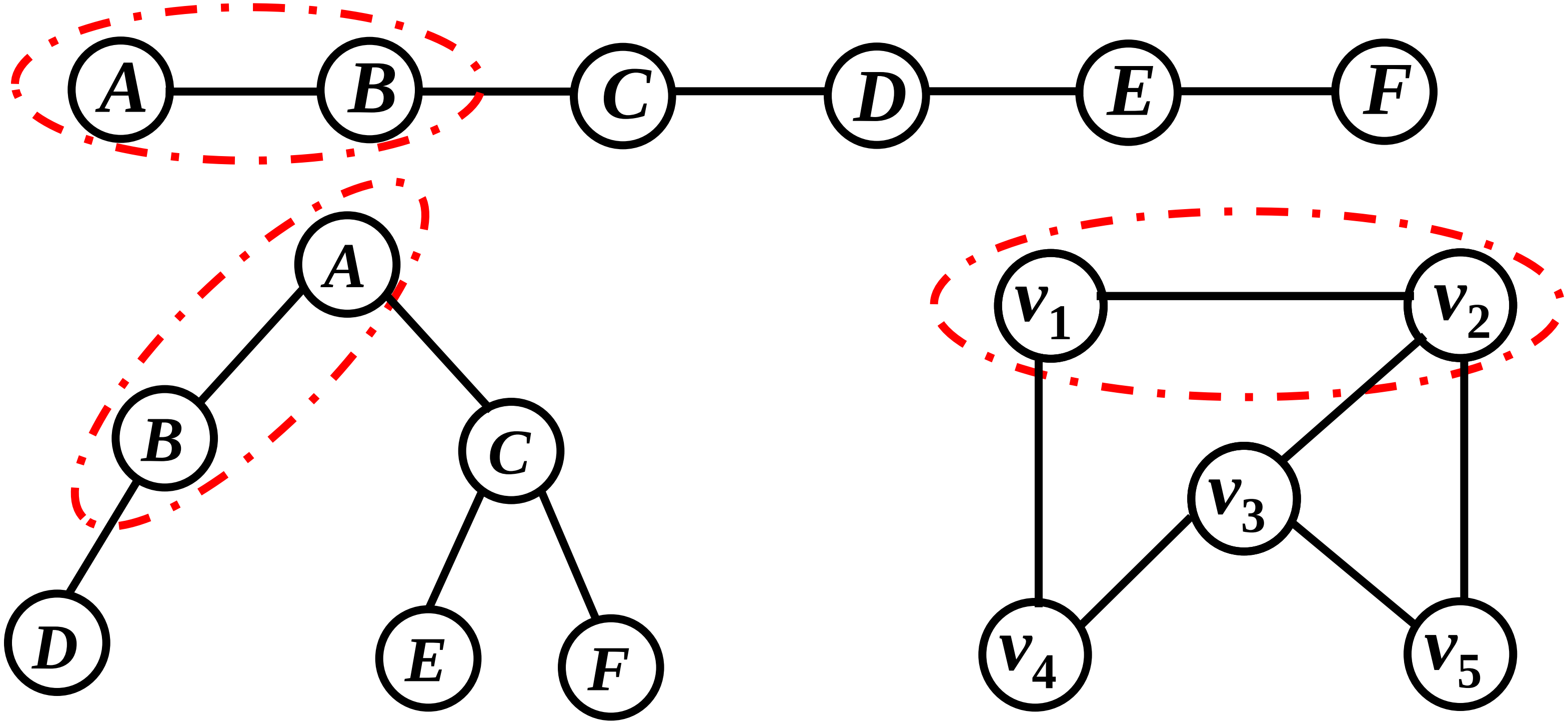
## 图的操作

设图G=(V,E)，图上定义的基本操作如下：

- NewNode ( G, v )：建立一个新顶点，V=V∪{v}
- DelNone ( G,  v )：删除顶点v以及与之相关联的所有边
- SetSucc ( G, v1, v2 ):增加一条边，E = E∪(v1,v2) ,V=V
- DelSucc ( G, v1, v2 )：删除边（v1,v2）,V不变
- Succ ( G, v )：求出v的所有直接后继结点
- Pred ( G, v)：求出v的所有直接前导结点
- IsEdge ( G, v1, v2 )：判断（v1,v2）∈E
- FirstAdjVex( G , v )：顶点v 的第一个邻接顶点
- NextAdjVex( G, v, w)：顶点v 的某个邻接点w的下一个邻接顶点。

# 4.1 基本定义(cont.)

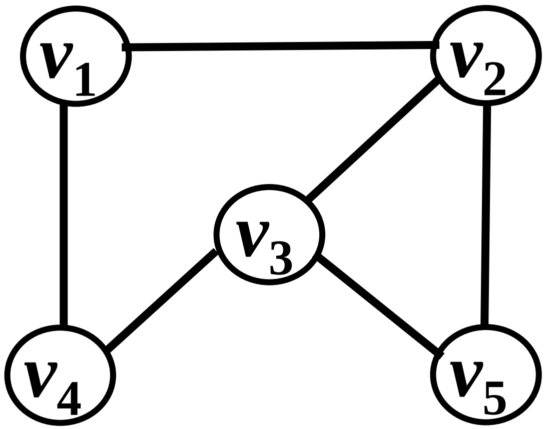
不同逻辑结构之间的比较

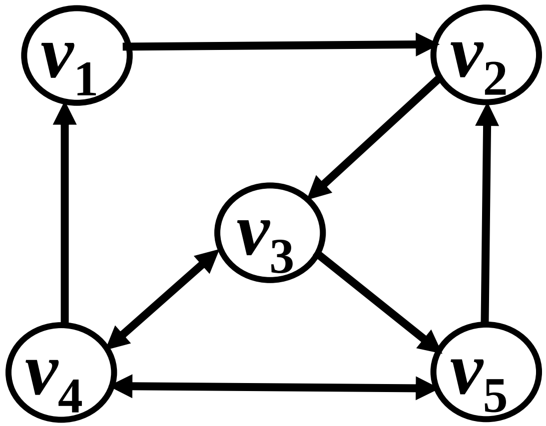



- 在线性结构中，数据元素之间仅具有**线性关系(1:1)**；
- 在树型结构中，结点之间具有**层次关系(1:*m*)**；
- 在图型结构中，任意两个顶点之间**都可能有关系(*m*:*n*)**。

# 4.2 图的存储结构

- 是否可以采用顺序存储结构存储图(一维数组）？
  - 图的特点：顶点之间的关系是*m:n*，即任何两个顶点之间都可能存在关系（边），无法通过存储位置表示这种任意的逻辑关系，所以，图无法采用顺序存储结构。
- 如何存储图？
  - 考虑图的定义，图是由顶点和边组成的；
  - 如何存储顶点、如何存储边----顶点之间的关系。



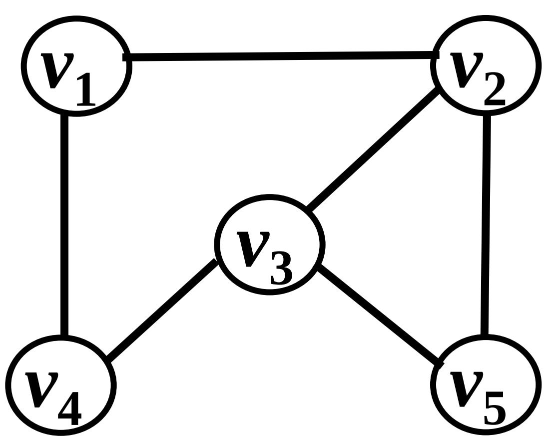



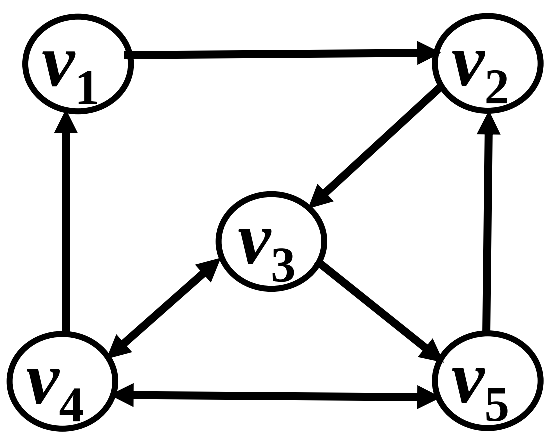
# 4.2 图的存储结构(cont.)

邻接矩阵 （*Adjacency Matrix*）表示（数组表示法）

- 基本思想：
  - 用一个一维数组存储图中顶点的信息，用一个二维数组（称为邻接矩阵）存储图中各顶点之间的邻接关系。
  - 假设图$G=(V, E)$有$n$个顶点，则邻接矩阵是一个$n\times n$的方阵，定义为：

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

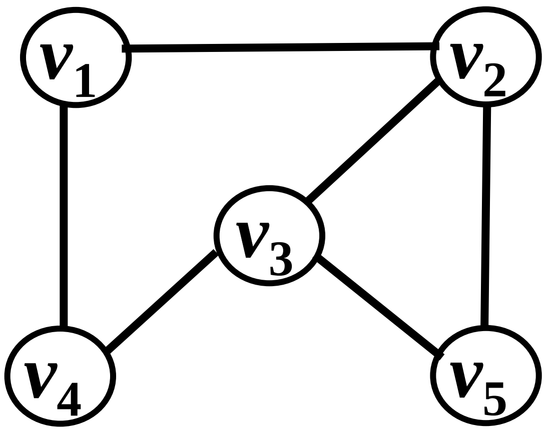


# 4.2 图的存储结构(cont.)

邻接矩阵 （*Adjacency Matrix*）表示（数组表示法）

- 无向图的邻接矩阵：




$$\text{vertex}=\begin{array}{|c|c|c|c|c|}\hline v_1 & v_2 & v_3 & v_4 & v_5 \\ \hline\end{array}$$

$$\text{edge} = \begin{array}{cc} \begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} & \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix} & \begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \end{array}$$

- 存储结构特点：
  - 主对角线为 0 且一定是对称矩阵；

问题：**1.** 如何求顶点$v_i$的度？

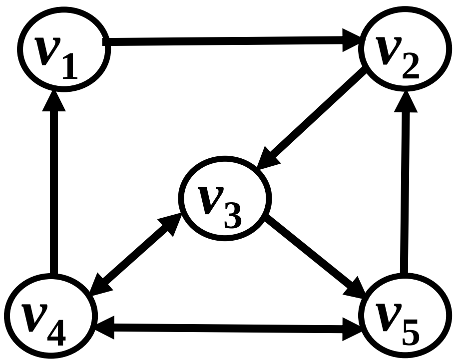
**2.**如何判断顶点 $v_i$和 $v_j$ 之间是否存在边？

**3.**如何求顶点 $v_i$的所有邻接点？

# 4.2 图的存储结构(cont.)

邻接矩阵 （*Adjacency Matrix*）表示（数组表示法）

- 有向图的邻接矩阵：




vertex= $\boxed{v_1 \mid v_2 \mid v_3 \mid v_4 \mid v_5}$

$$
\text{edge} = \begin{array}{c c}
\begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} & \\
\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} & \begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array}
\end{array}
$$

- 存储结构特点：
  - 有向图的邻接矩阵一定不对称吗？

问题：1.如何求顶点$v_i$的出度？

2. 如何判断顶点 $v_i$和 $v_j$ 之间是否存在有向边？

3.如何求邻接于顶点 $v_i$的所有顶点？

4.如何求邻接到顶点 $v_i$的所有顶点？

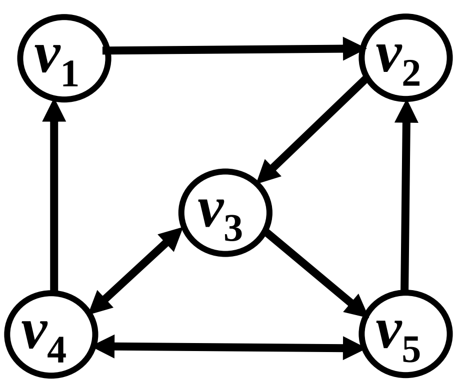
# 4.2 图的存储结构(cont.)

邻接矩阵 （*Adjacency Matrix*）表示（数组表示法）

- 存储结构定义：

> 假设图*G*有*n*个顶点e条边，则该图的存储需求为$O(n+n^2) = O(n^2)$，与边的条数e无关。

```
typedef struct {
    VertexData verlist [NumVertices];   //顶点表
    EdgeData edge[NumVertices][NumVertices];
        //邻接矩阵—边表, 可视为边之间的关系
    int n, e;  //图的顶点数与边数
} MTGraph;
```




vertex

| | | | | |
|---|---|---|---|---|
| $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ |

$$\begin{array}{c} \\ edge= \end{array}\begin{array}{c} \begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} \end{array}\begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array}$$

# 4.2 图的存储结构(cont.)

- 存储结构的建立----算法实现的步骤：

1.确定图的顶点个数n和边数e；

2.输入顶点信息存储在一维数组vertex中；

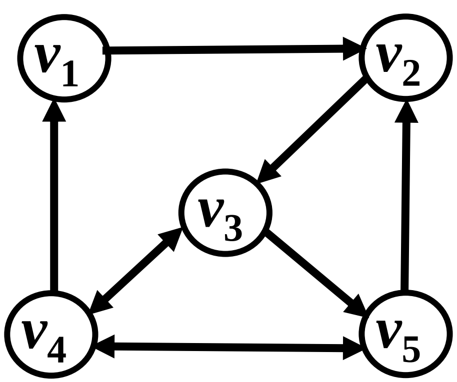
3.初始化邻接矩阵；

4.依次输入每条边存储在邻接矩阵edge中；

4.1 输入边依附的两个顶点的序号i, j；

4.2 将邻接矩阵的第i行第j列的元素值置为1；

4.3 将邻接矩阵的第j行第i列的元素值置为1。




vertex

| | | | | |
|---|---|---|---|---|
| $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ |

$$\text{edge}=\begin{array}{c} \begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array} \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array}$$

# 4.2 图的存储结构(cont.)

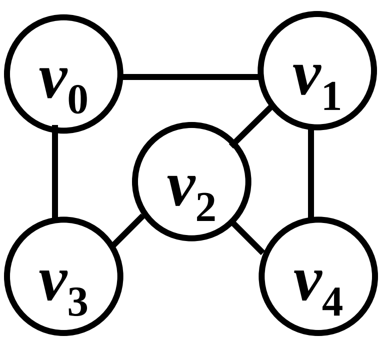
- 存储结构的建立算法的实现：

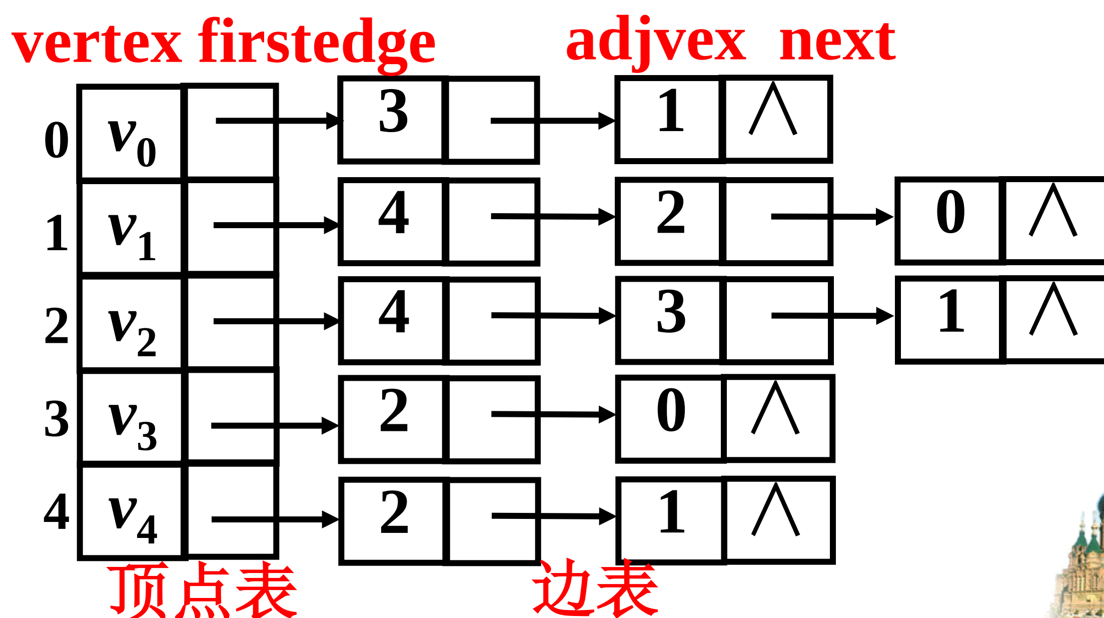
```
void  CreateMGragh (MTGragh *G) //建立图的邻接矩阵
{     int  i, j, k, w;
      cin>>G→n>>G→e;              //1.输入顶点数和边数
      for (i=0; i<G→n; i++)        //2.读入顶点信息，建立顶点表
           G→vertlist[i]=getchar( );
      for (i=0; i<G→n; i++)
           for (j=0;j<G→n;j++)
                G→edge[i][j]=0;        //3.邻接矩阵初始化
      for (k=0; k<G→e; k++) {      //4.读入e条边建立邻接矩阵
           cin>>i>>j>>w;              // 输入边（i,j）上的权值w
           G→ edge[i][j]=w; G→ edge[j][i]=w;
      }
} //时间复杂度：T = O( n+ n² +e) 。当e < <n, T = O( n² ) ?
```

# 4.2 图的存储结构(cont.)

## 邻接表（Adjacency List）表示

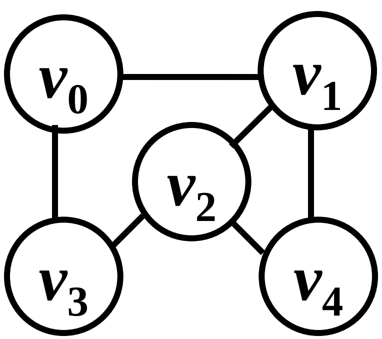
- 无向图的邻接表：
  - 对于无向图的每个顶点$v_i$，将所有与$v_i$相邻的顶点链成一个单链表，称为顶点$v_i$的边表（顶点$v_i$的邻接表）；
  - 再把所有边表的指针和存储顶点信息的一维数组构成顶点表。

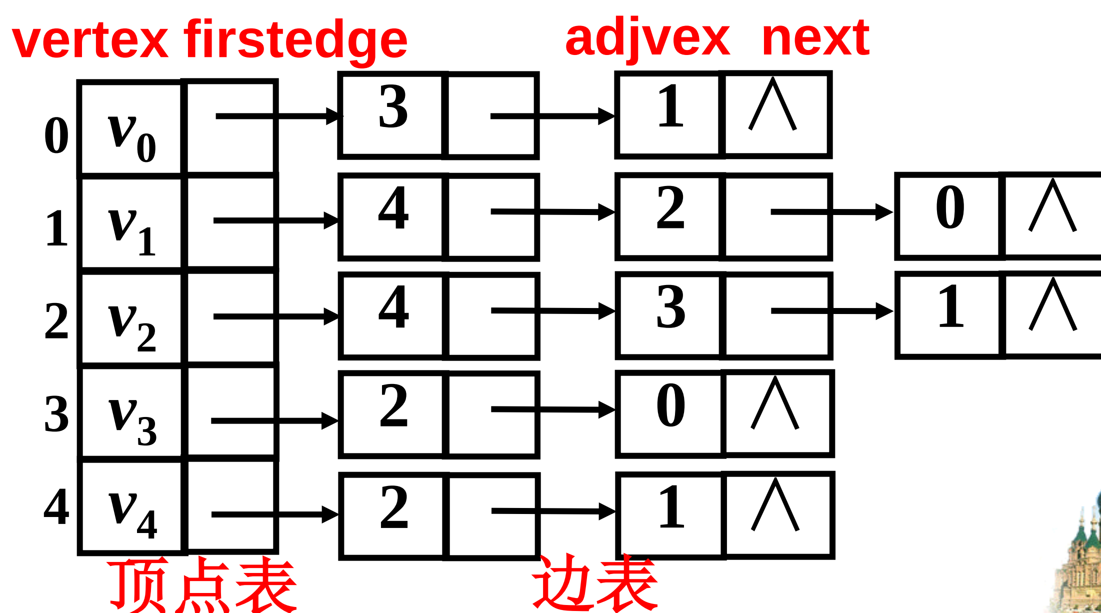


# 4.2 图的存储结构(cont.)

- 无向图的邻接表存储的特点:
  - 边表中的结点表示什么?
  - 如何求顶点 $v_i$的度?
  - 如何判断顶点$v_i$和顶点$v_j$之间是否存在边?
  - 如何求顶点 $v_i$的所有邻接点?
  - 空间需求O(n+2e)

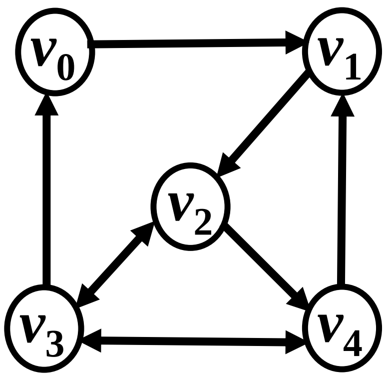


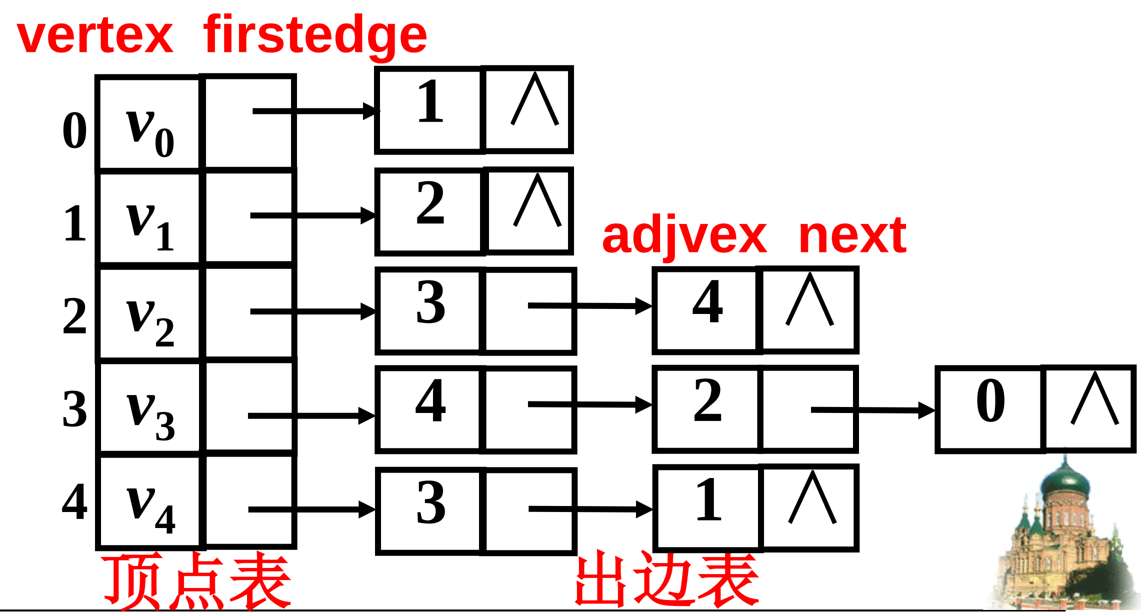



# 4.2 图的存储结构(cont.)

## 邻接表（Adjacency List）表示

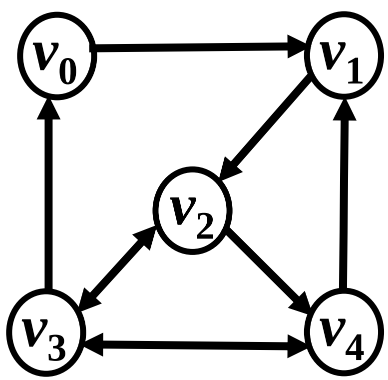
- 有向图的邻接表---正邻接表
  - 对于有向图的每个顶点$v_i$，将邻接于$v_i$的所有顶点链成一个单链表，称为顶点$v_i$的出边表；
  - 再把所有出边表的指针和存储顶点信息的一维数组构成顶点表。

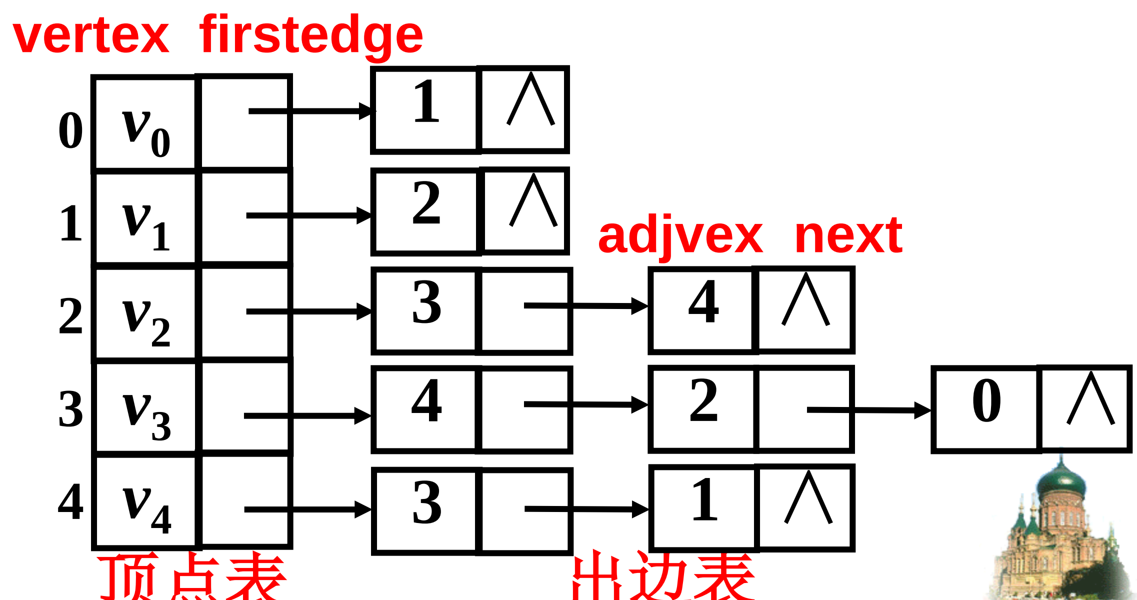


# 4.2 图的存储结构(cont.)

- 有向图的正邻接表的存储特点
  - 出边表中的结点表示什么？
  - 如何求顶点 $v_i$的出度？如何求顶点 $v_i$的入度？
  - 如何判断顶点 $v_i$和顶点$v_j$之间是否存在有向边?
  - 如何求邻接于顶点 $v_i$的所有顶点?
  - 如何求邻接到顶点 $v_i$的所有顶点?
  - 空间需求:O(n+e)

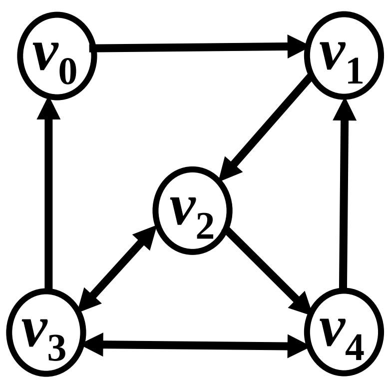


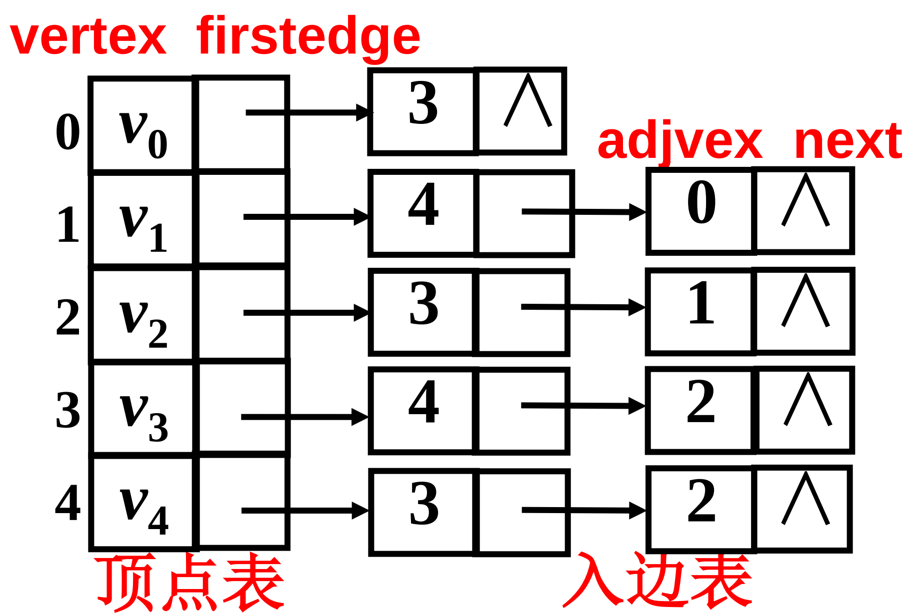



# 4.2 图的存储结构(cont.)

## 邻接表（Adjacency List）表示

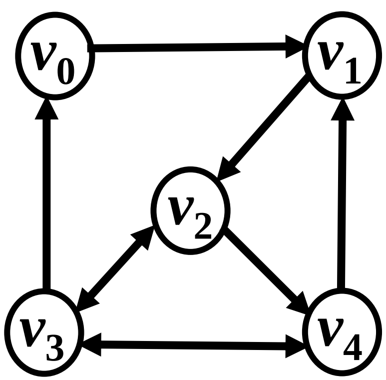
- 有向图的邻接表----逆邻接表
  - 对于有向图的每个顶点$v_i$，将邻接到$v_i$的所有顶点链成一个单链表，称为顶点$v_i$的入边表；
  - 再把所有入边表的指针和存储顶点信息的一维数组构成顶点表。

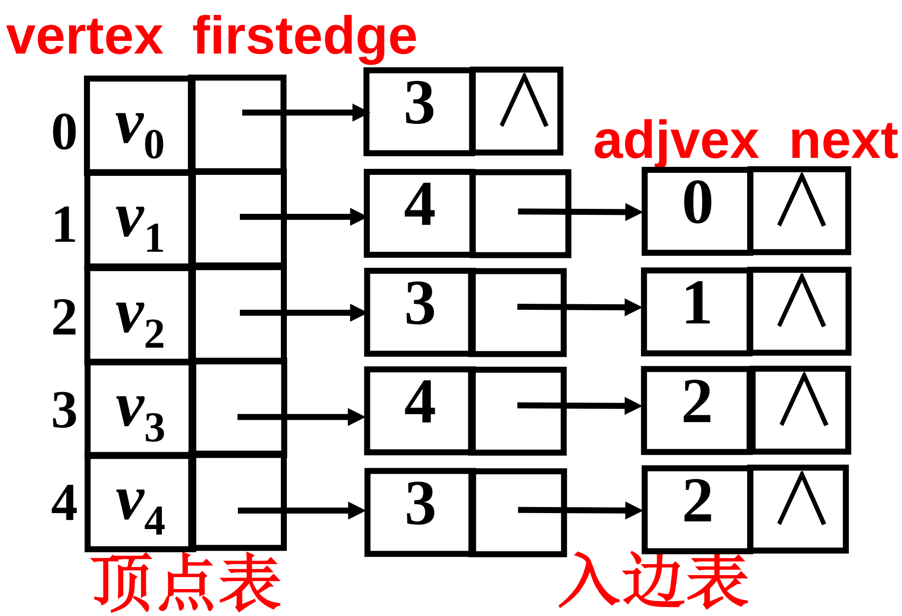


# 4.2 图的存储结构(cont.)

- 有向图的逆邻接表的存储特点
  - 出边表中的结点表示什么？
  - 如何求顶点 $v_i$的入度？如何求顶点 $v_i$的出度？
  - 如何判断顶点 $v_i$和顶点$v_j$之间是否存在有向边?
  - 如何求邻接到顶点 $v_i$的所有顶点?
  - 如何求邻接于顶点 $v_i$的所有顶点?
  - 空间需求:O(n+e)



# 4.2 图的存储结构(cont.)

- 邻接表存储结构的定义

```
typedef struct node {//边表结点
    int adjvex;          //邻接点域（下标）
    EdgeData cost;            //边上的权值
    struct node *next; //下一边链接指针
} EdgeNode;
typedef struct {            //顶点表结点
    VertexData  vertex; //顶点数据域
    EdgeNode * firstedge;//边链表头指针
} VertexNode;
typedef struct {        //图的邻接表
    VertexNode vexlist [NumVertices];
    int n, e;            //顶点个数与边数
} AdjGraph;
```

边表结点

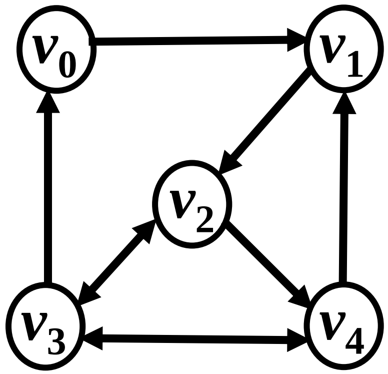
| adjvex | cost | next |
|---|---|---|

顶点表结点

| vertex | firstedge |
|---|---|

# 4.2 图的存储结构(cont.)

- 邻接表存储结构建立算法实现的步骤：




1. 确定图的顶点个数和边的个数；

2. 建立顶点表：

 2.1 输入顶点信息；

 2.2 初始化该顶点的边表；

3. 依次输入边的信息并存储在边表中；

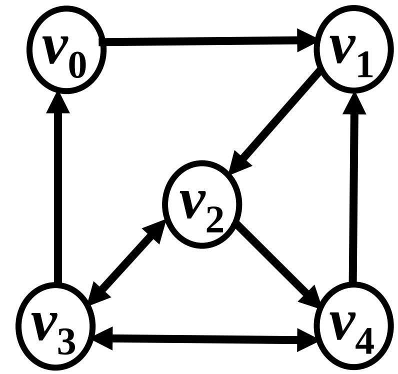
 3.1 输入边所依附的两个顶点的序号tail和head和权值w；

 3.2 生成邻接点序号为head的边表结点p；

 3.3 设置边表结点p；

 3.4 将结点p插入到第tail个边表的头部；

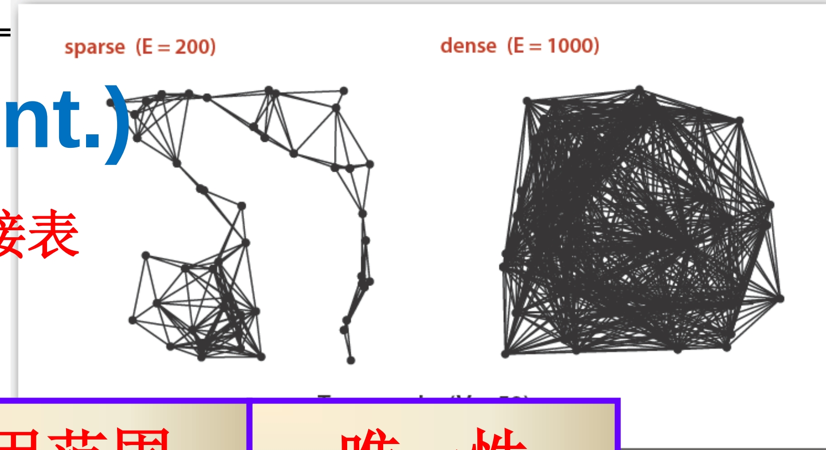
# 4.2 图的存储结构(cont.)




- 邻接表存储结构建立算法的实现：

```
void CreateGraph (AdjGraph G)
{  cin >> G.n >> G.e;                       //1.输入顶点个数和边数
   for ( int i = 0; i < G.n; i++) {         //2.建立顶点表
      cin >> G.vexlist[i].vertex;           //2.1输入顶点信息
      G.vexlist[i].firstedge = NULL; }      //2.2边表置为空表
   for ( i = 0; i < G.e; i++) {             //3.逐条边输入,建立边表
      cin >> tail >> head >> weight;        //3.1输入
      EdgeNode * p = new EdgeNode;          //3.2建立边结点
      p→adjvex = head;   p→cost = weight;   //3.3设置边结点
      p→next = G.vexlist[tail].firstedge;   //3.4链入第 tail 号链表的前端
      G.vexlist[tail].firstedge = p;
      p = new EdgeNode;
      p→adjvex = tail;   p→cost = weight;
      p→next = G.vexlist[head].firstedge; //链入第 head 号链表的前端
     G.vexlist[head].firstedge = p; }
} //时间复杂度：O(2e+n)
```

# 4.2 图的存储结构(cont.)




- 图的存储结构的比较——邻接矩阵和邻接表

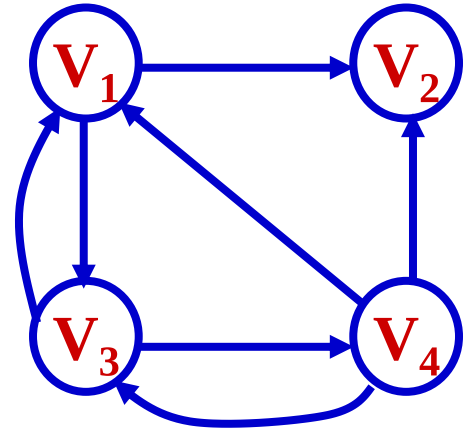
| | 空间性能 | 时间性能 | 适用范围 | 唯一性 |
|---|---|---|---|---|
| 邻接矩阵 | O（$n^2$） | O（$n^2$） | 稠密图 | 唯一？ |
| 邻接表 | O（n+e） | O（n+e） | 稀疏图 | 不唯一？ |

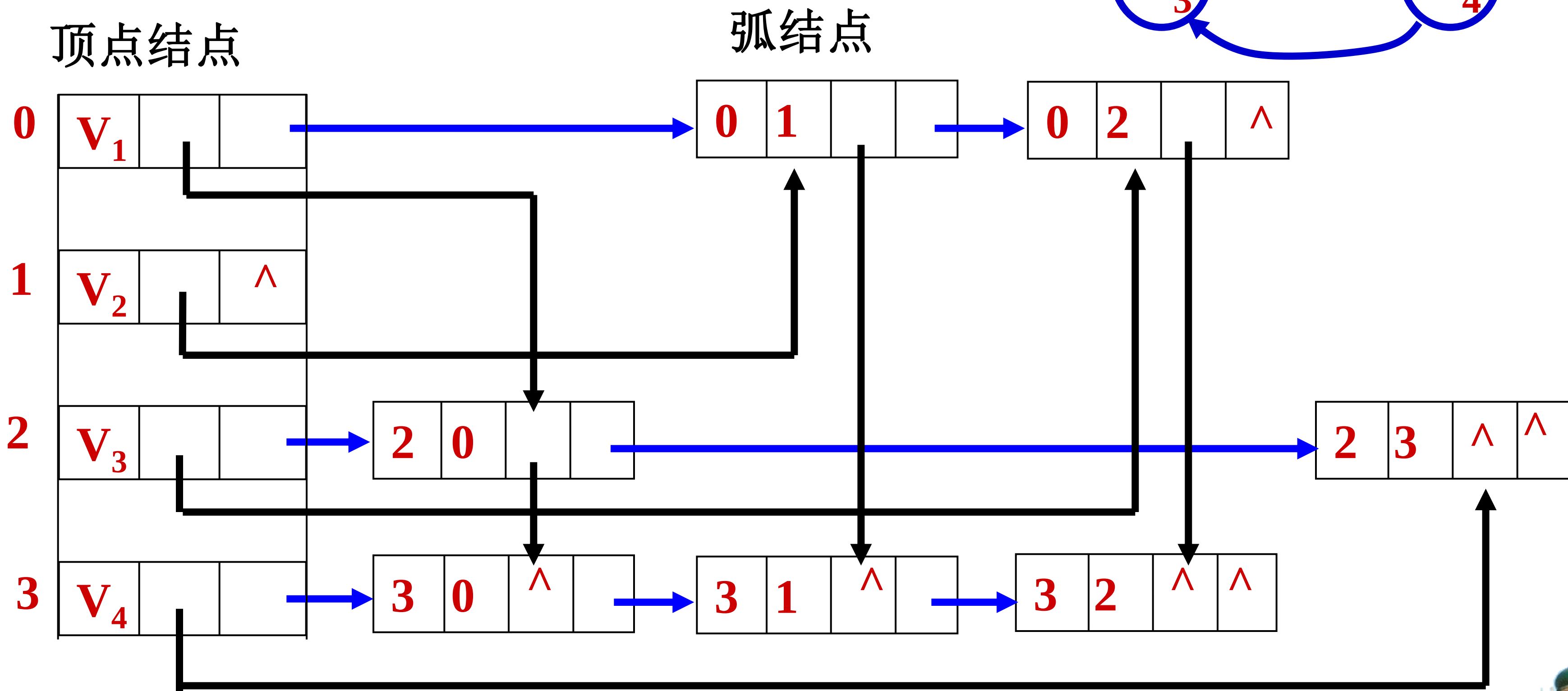
## ◆ 十字链表(有向图)

- jlink:指向j的边；ilink:i发出的边

| ivex | jvex | jlink | ilink | info |
|---|---|---|---|---|

| data | firstin | firstout |
|---|---|---|

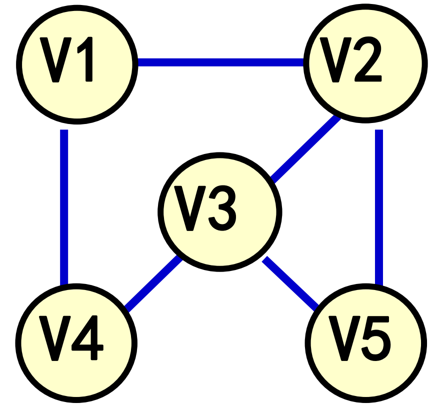


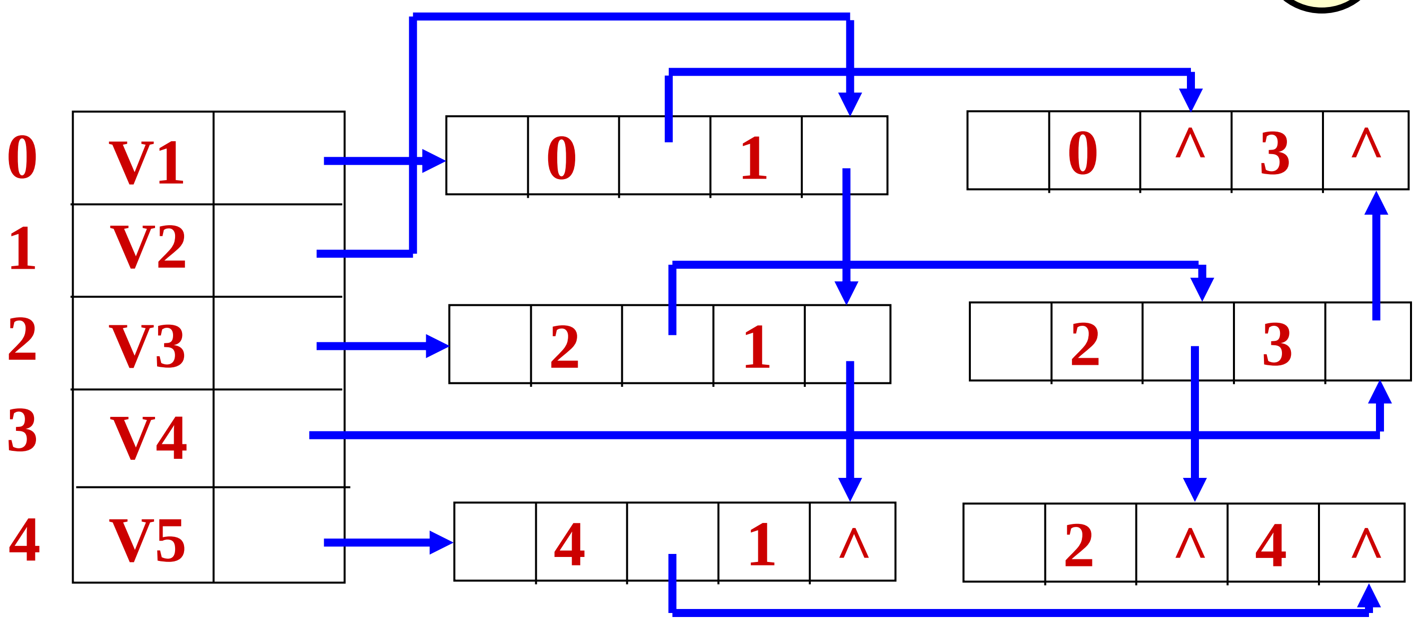



## ◆ 邻接多重表(无向图)

| mark | ivex | ilink | jvex | jlink | info |
|---|---|---|---|---|---|

| data | firstedge |
|---|---|



# 4.3 图的搜索（遍历）


**John Edward Hopcroft**

**Robert Endre Tarjan**


## 1986年图灵奖获得者

约翰·霍普克洛夫特1939年生于西雅图。美国国家科学院和工程院院士、康奈尔大学智能机器人实验室主任。1962和1964年获斯坦福大学硕士和博士学位。先后在普林斯顿大学、斯坦福大学等工作，也曾任职于一些科学研究机构如NSF和NRC。著作很多如《算法设计与分析基础》《 数据结构与算法》《自动机理论、语言和计算导论》《形式语言及其与自动机的关系》

罗伯特·塔扬普林斯顿大学计算机科学系教授，1948年4月30日生于加利福尼亚州。1969年本科毕业，进入斯坦福大学研究生院，1972年获得博士学位。平面图测试的高效算法；合并-搜索问题；“分摊”算法的概念；八字形树；持久性数据结构

# 4.3 图的搜索（遍历） (cont.)

- 图的遍历（图的搜索）
  - 从图中某一顶点出发，对图中所有顶点访问一次且仅访问一次。
  - 访问：抽象操作，可以是对结点进行的各种处理
- 图结构的复杂性
  - 在线性表中，数据元素在表中的编号就是元素在序列中的位置，因而其编号是唯一的；
  - 在树结构中，将结点按层序编号，由于树具有层次性，因而其层序编号也是唯一的；
  - 在图结构中，任何两个顶点之间都可能存在边，顶点是没有确定的先后次序的，所以，顶点的编号不唯一。

# 4.3 图的搜索（遍历）(cont.)

- 图的遍历要解决的关键问题
  - 在图中，如何选取遍历的起始顶点？
    - 解决办法：从编号小的顶点(任取一顶点，适合编程）开始 。
  - 从某个起点始可能到达不了所有其它顶点，怎么办？
    - 解决办法：多次调用遍历图（起点选没有用过的）的算法。
  - 图中可能存在回路，且图的任一顶点都可能与其它顶点"相通"，在访问完某个顶点之后可能会沿着某些边又回到了曾经访问过的顶点。如何避免某些顶点可能会被重复访问？
    - 解决办法：设访问标志数组visited[n] 。
  - 在图中，一个顶点可以和其它多个顶点相连，当这样的顶点访问过后，如何选取下一个要访问的顶点？
    - 解决办法：深度优先搜索（Depth First Search）和广度优先搜索（Breadth First Search）。

# 4.3 图的搜索（遍历）(cont.)

- 深度优先遍历----类似于树结构的先序遍历

设图G的初态是所有顶点都“未访问过（False）”，在G中任选一个顶点 v 为初始出发点(源点)，则深度优先搜索可定义为:

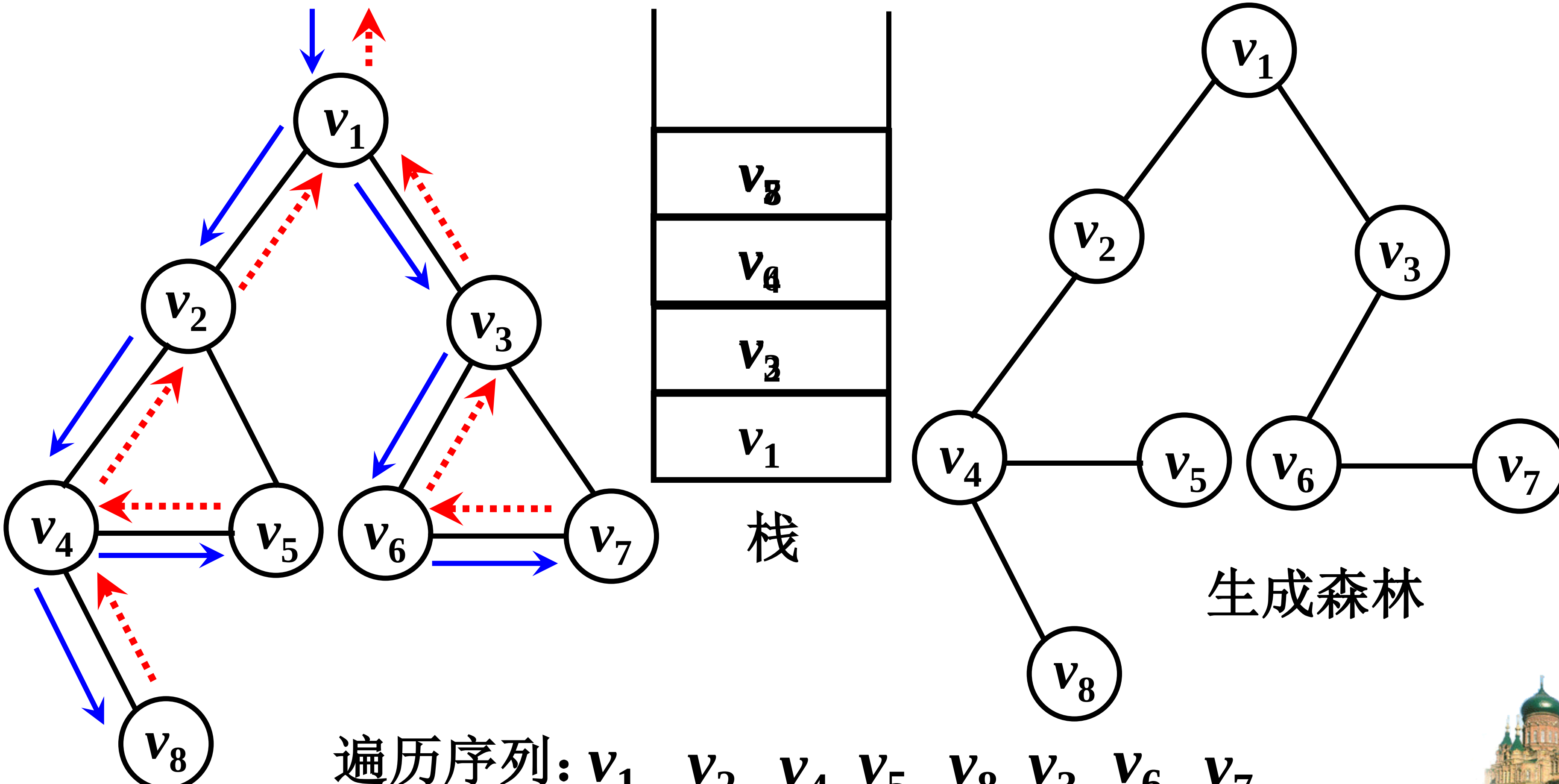
①首先访问出发点 v，并将其标记为“访问过（True）”；

②然后，从 v 出发，依次考察与 v 相邻的顶点 w；若 w “未访问过（False）”，则以 w 为新的出发点递归地进行深度优先搜索，直到图中所有与源点 v 有路径相通的顶点（亦称从源点可到达的顶点）均被访问为止；

③若此时图中仍有未被访问过的顶点，则另选一个“未访问过”的顶点作为新的搜索起点，重复上述过程，直到图中所有顶点都被访问过为止。

# 4.3 图的搜索（遍历）(cont.)

- 深度优先遍历示例
  - 深度优先遍历序列?入栈序列?出栈序列?




遍历序列：$v_1$ $v_2$ $v_4$ $v_5$ $v_8$ $v_3$ $v_6$ $v_7$

# 4.3 图的搜索（遍历）(cont.)

- 深度优先遍历特点：
  - 是递归的定义，是尽可能对纵深方向上进行搜索，故称先深或深度优先搜索。
- 先深或深度优先编号。
  - 搜索过程中，根据访问顺序给顶点进行的编号，称为先深或深度优先编号。
- 先深序列或**DFS**序列：
  - 先深搜索过程中，根据访问顺序得到的顶点序列，称为先深序列或**DFS**序列。
- 生成森林（树）：
  - 有原图的所有顶点和搜索过程中所经过的边构成的子图。
- 先深搜索结果不唯一
  - 即图的**DFS**序列、先深编号和生成森林不唯一。

# 4.3 图的搜索（遍历）(cont.)

- 深度优先遍历主算法：

```
bool visited[NumVertices];  //访问标记数组是全局变量
int dfn[NumVertices];  //顶点的先深编号
void DFSTraverse (AdjGraph G)     //主算法
// 先深搜索----邻接表表示的图G；而以邻接矩阵表示G时，算法完全相同

{   int i , count = 1;
    for ( int i = 0; i < G.n; i++ )
        visited [i] =False;   //标志数组初始化
    for ( int i = 0; i < G.n; i++ )
       if ( ! visited[i] )
            DFSX ( G, i ); //从顶点 i 出发的一次搜索，BFSX(G, i )
}
```

# 4.3 图的搜索（遍历） (cont.)

- 从一个顶点出发的一次深度优先遍历算法：
  - 实现步骤：

1. 访问顶点v; visited[v]=1;
2. w=顶点v的第一个邻接点；
3. while (w存在)
   3.1 if (w未被访问) 从顶点w出发递归执行该算法;
   3.2 w=顶点v的下一个邻接点;

# 4.3 图的搜索（遍历）(cont.)

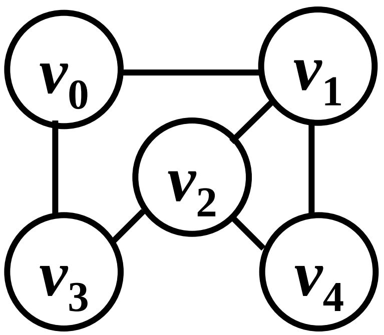
- 从一个顶点出发的一次深度优先遍历算法：

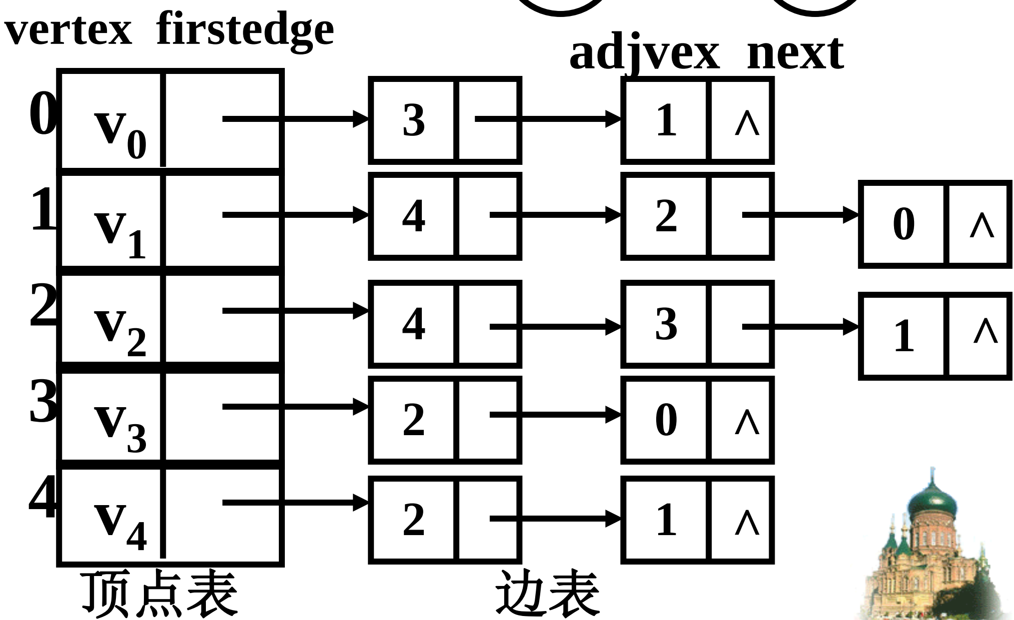
```
void DFS1 (AdjGraph* G, int i)
//以vi为出发点时对邻接表表示的图G进行先深搜索
{     EdgeNode *p;
      cout<<G→vexlist[i].vertex;     //访问顶点vi;
      visited[i]=True;               //标记vi已访问
      dfn[i]=count++;                //对vi进行编号
      p=G→vexlist[i].firstedge;      //取vi边表的头指针
      while( p ) {   //依次搜索vi的邻接点vj, 这里j=p->adjvex
          if( !visited[ p→adjvex ] )  //若vj尚未访问
              DFS1(G, p→adjvex);  //则以vj为出发点先深搜索
          p=p→next;
      }
} //DFS1
```

# 4.3 图的搜索（遍历） (cont.)

- 从一个顶点出发的一次深度优先遍历算法：

```
void DFS1 (AdjGraph* G, int i)
//以vi为出发点时对邻接表表示的图G进行先深搜索
{     EdgeNode *p;
      cout<<G→vexlist[i].vertex;
      visited[i]=True;
      dfn[i]=count++;
      p=G→vexlist[i].firstedge;
      while( p ) {
           if( !visited[ p→adjvex ] )
                DFS1(G, p→adjvex);
           p=p→next;
      }
} //DFS1
```



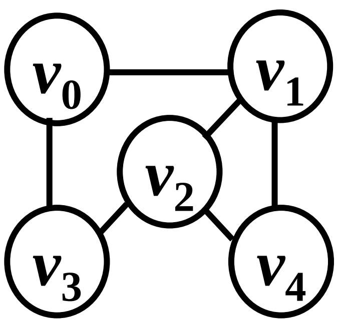



# 4.3 图的搜索（遍历） (cont.)

## 从一个顶点出发的一次深度优先遍历算法:

```
void DFS2(MTGraph *G, int i)
//以vi为出发点对邻接矩阵表示的图G进行深度优先搜索
{   int j;
    cout<<G→vexlist[i];      //访问定点vi
    visit[i]=True;           //标记vi已访问
    dfn[i]=count;            //对vi进行编号
    count ++;                //下一个顶点的编号
    for( j=0; j<G→n; j++ ) //依次搜索vi的邻接点
        if ( (G→edge[i][j] == 1)&&！ visited[j] ) //若vj尚未访问
            DFS2( G, j );
}//DFS2
```

$$
\begin{array}{c}
 \\ v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4
\end{array}
\begin{array}{c}
\begin{array}{ccccc} v_0 & v_1 & v_2 & v_3 & v_4 \end{array} \\
\begin{pmatrix}
0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0
\end{pmatrix}
\end{array}
$$



# 4.3 图的搜索（遍历）(cont.)

- 广度优先遍历----类似于树结构的层序遍历

设图G的初态是所有顶点都“未访问过（False）”，在G中任选一个顶点 v 为源点，则广度优先搜索可定义为:

①首先访问出发点 v，并将其标记为“访问过（True）”；

②接着依次访问所有与 v 相邻的顶点$w_1$，$w_2 \ldots w_t$;

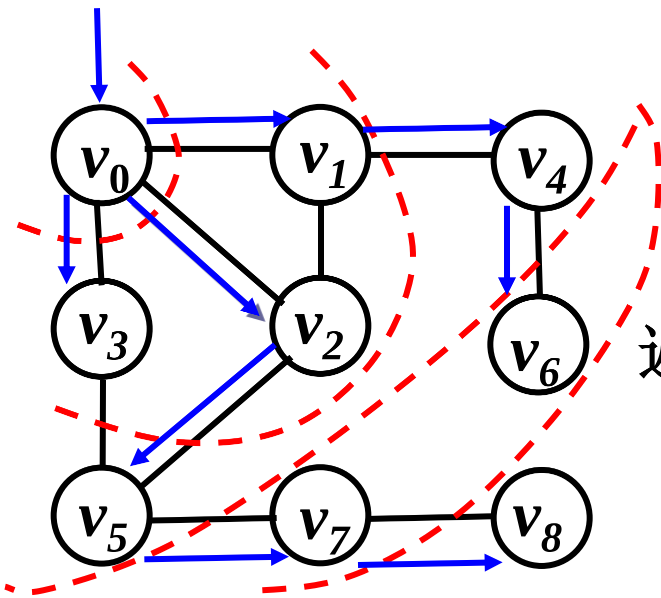
③然后依次访问与$w_1$，$w_2 \ldots w_t$相邻的所有未访问的顶点;

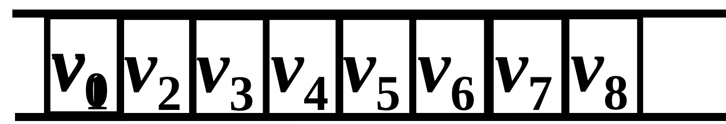
④依次类推，直至图中所有与源点v有路相通的顶点都已访问过为止；

⑤此时，从 v 开始的搜索结束，若G是连通的，则遍历完成；否则在G中另选一个尚未访问的顶点作为新源点，继续上述搜索过程，直到G中的所有顶点均已访问为止。

# 4.3 图的搜索（遍历）(cont.)

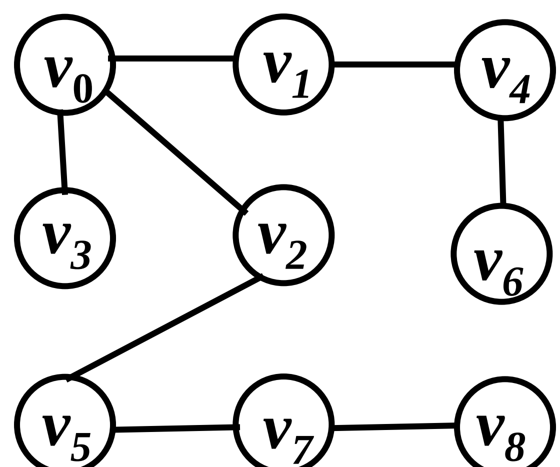
- 广度优先遍历示例
- 广度优先遍历序列?入队序列?出队序列?




遍历序列: $v_0$ $v_1$ $v_2$ $v_3$ $v_4$ $v_5$ $v_6$ $v_7$ $v_8$

生成森林



# 4.3 图的搜索（遍历）(cont.)

- 广度优先遍历特点：
  - 尽可能横向上进行搜索，并使“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”被访问，故称先广搜索或广度优先搜索。
- 先广或广度优先编号：
  - 搜索过程中，根据访问顺序给顶点进行的编号，称为先广或广度优先编号
- 先广序列或BFS序列：
  - 先广搜索过程中，根据访问顺序得到的顶点序列，称为先广序列或BFS序列。
- 生成森林（树）：
  - 有原图的所有顶点和搜索过程中所经过的边构成的子图。
- 先广搜索结果不唯一：
  - 即图的BFS序列、先广编号和生成森林不唯一。

# 4.3 图的搜索（遍历） (cont.)

- 广度优先遍历主算法：

```
bool visited[NumVertices]; //访问标记数组是全局变量
int dfn[NumVertices]; //顶点的先深编号
void BFSTraverse (AdjGraph G)     //主算法
//* 先广搜索一邻接表表示的图G；而以邻接矩阵表示G时，算法完
    全相同
{   int i , count = 1;
    for ( int i = 0; i < G.n; i++ )
        visited [i] =False;   //标志数组初始化
    for ( int i = 0; i < G.n; i++ )
        if ( ! visited[i] )
            BFSX ( G, i ); //从顶点 i 出发的一次搜索，DFSX (G, i )
}
```

# 4.3 图的搜索（遍历）(cont.)

- 从一个顶点出发的一次广度优先遍历算法：
  - 实现步骤：

1 . 初始化队列Q;

2. 访问顶点v; visited [v]=1; 顶点v入队Q;

3. while (队列Q非空)

   3.1 v=队列Q的队头元素出队;

   3.2 w=顶点v的第一个邻接点;

   3.3 while (w存在)

      3.3.1 如果w 未被访问，则

         访问顶点w; visited[w]=1; 顶点w入队列Q;

      3.3.2 w=顶点v的下一个邻接点;

# 4.3 图的搜索（遍历）(cont.)

```
void BFS1 (AdjGraph *G, int k)//这里没有进行先广编号
{    int i; EdgeNode *p; Queue Q;   MakeNull(Q);
     cout << G→vexlist[ k ] .vertex;   visited[ k ] = True;
     EnQueue (k, Q);                          //进队列
     while ( ! Empty (Q) ) {                    //队空搜索结束
         i=DeQueue(Q);                        //vi出队
         p =G→vexlist[ i ].firstedge;          //取vi的边表头指针
         while ( p ) {                          //若vi的邻接点 vj (j= p→adjvex)存在,依次搜索
             if ( !visited[ p→adjvex ]) {  //若vj未访问过
                 cout << G→vexlist[ p→adjvex ].vertex;   //访问vj
                 visited[ p→adjvex ]=True;              //给vj作访问过标记
                 EnQueue ( p→adjvex , Q );              //访问过的vj入队
             }
             p = p→next;                        //找vi的下一个邻接点
         } /          / 重复检测 vi的所有邻接顶点
     }                         //外层循环，判队列空否
}//以vk为出发点时对用邻接表表示的图G进行先广搜索
```

# 4.3 图的搜索（遍历）(cont.)

```
void BFS2 (MTGraph *G, int k) //这里没有进行先广编号
{    int i , j; Queue Q;   MakeNull(Q);
     cout << G→vexlist[ k ];   //访问vk
     visited[ k ] = True; //给vk作访问过标记
     EnQueue (k, Q);   // vk进队列
     while ( ! Empty (Q) ) {  //队空时搜索结束
          i=DeQueue(Q); //vi出队
          for(j=0; j<G→n; j++) {   //依次搜索vi的邻接点 vj
               if ( G→edge[ i ][ j ] ==1 && !visited[ j ]) { //若vj未访问过
                    cout << G→vexlist[ j ];//访问vj
                    visited[ j ]=True;  //给vj作访问过标记
                    EnQueue ( j , Q ); //访问过的vj入队
               }
          } //重复检测 vi的所有邻接顶点
     } //外层循环，判队列空否
} // 以vk为出发点时对用邻接矩阵表示的图G进行先广搜索
```

# 无向图（的搜索）及其应用

- 无向图连通性判定

- 不连通：若干个生成树

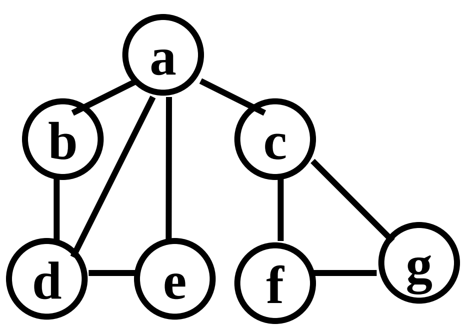
  求连通分量个数；

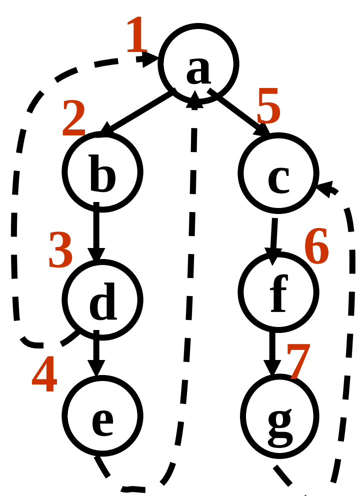
  求出每个连通分量；

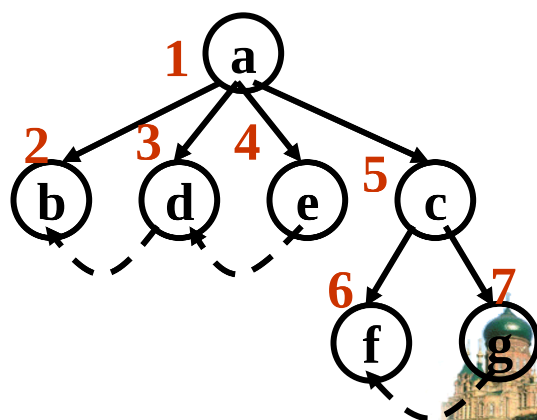
- 连通：一棵生成树

  判断是否有环路；

先深生成森林和先广生成森林



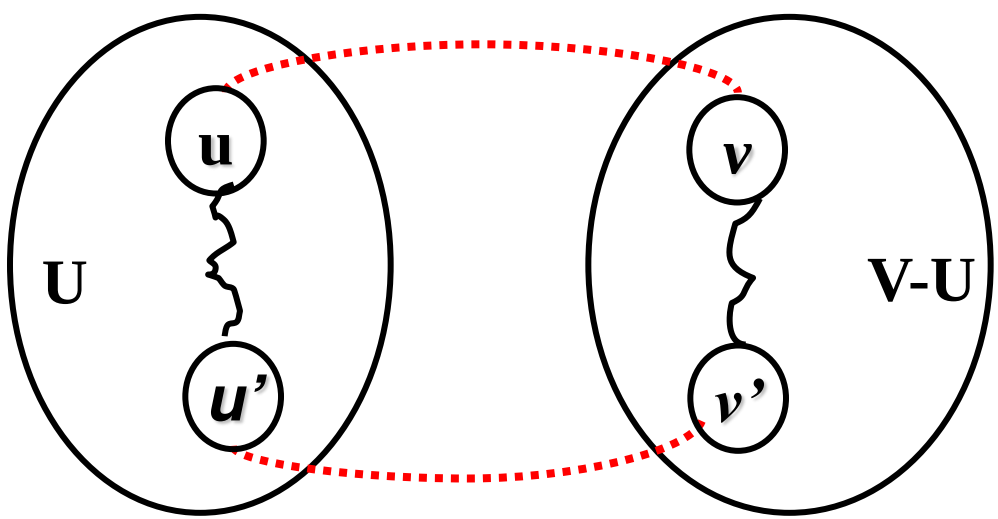
# 4.4.3 最小生成树算法

- 生成树的代价
  - 设G=（V, E）是一个无向连通网， E中每一条边（u, v）上的权值c（u, v），称为（u, v）的边长。
  - 图G的生成树上各边的权值（边长）之和称为该生成树的代价。
- 最小生成树(Minimum-Cost Spanning Tree，MST)
  - 在图G所有生成树中，代价最小的生成树称为最小生成树
- 最小生成树的概念可以应用到许多实际问题中。
  - 例如，在*n*个教室之间建造局域网络，至少要架设*n*-1条通信线路，而每两个教室之间的距离可能不同，从而架设通信线路的造价就是不一样的，那么如何设计才能使得总造价最小？

- 构造最小生成树的准则
  - 必须使用且仅使用该连通图中的$n$-1 条边连接结图中的 $n$ 个顶点；
  - 不能使用产生回路的边；
  - 各边上的权值的总和达到最小。

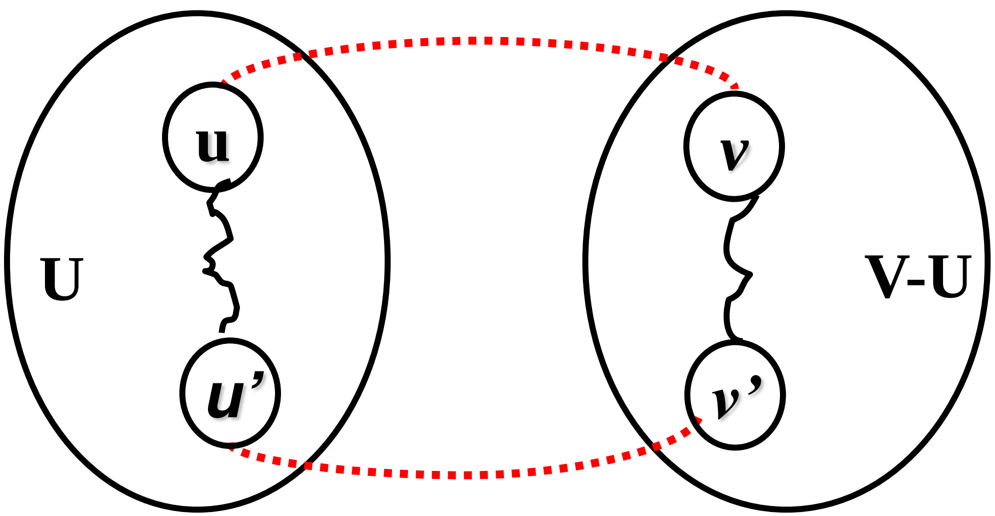
# 4.4 最小生成树算法(cont.)

- 最小生成树的性质
  - 假设G =（V，E）是一个连通网，U是顶点V的一个非空子集。若（u，v）是一条具有最小权值（代价）的边，其中u∈U，v∈V-U，则必存在一棵包含边（u，v）的最小生成树。
  - 此性质保证了Prim和Kruskal贪心算法的正确性

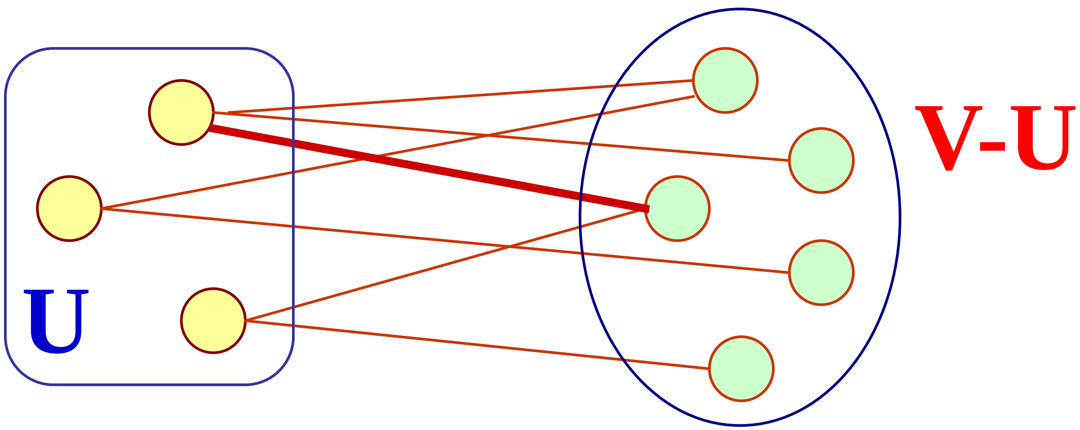


# 4.4 最小生成树算法(cont.)

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

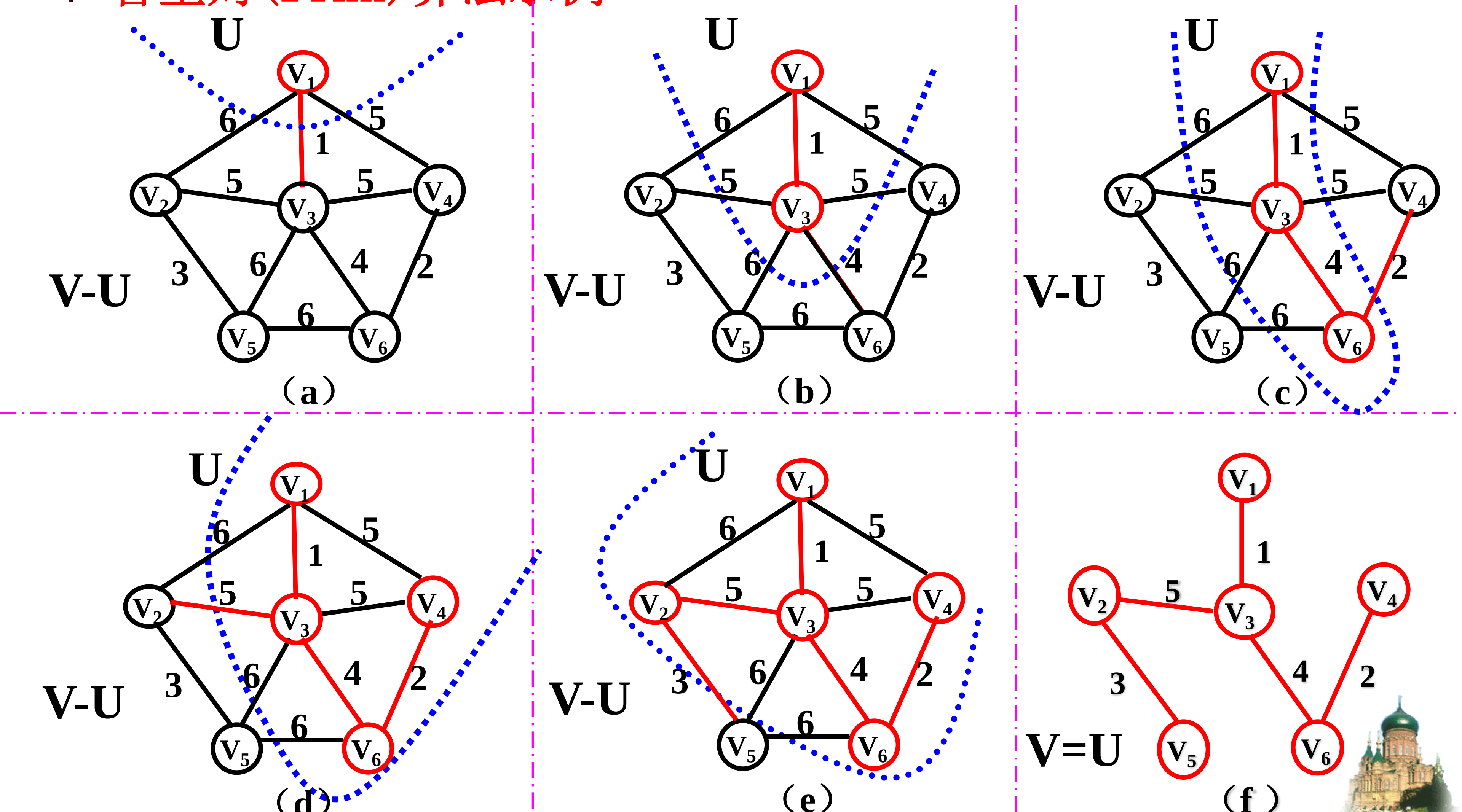


在生成树的构造过程中，图中n个顶点分属两个集合：已落在生成树上的顶点集U和尚未落在生成树上的顶点集V-U，则应在所有连通U中顶点和V-U中顶点的边中选取权值最小的边。



# 4.4 最小生成树算法(cont.)

- 普里姆(Prim)算法示例



# 4.4 最小生成树算法(cont.)

- 普里姆（Prim）算法
  - 基本思想

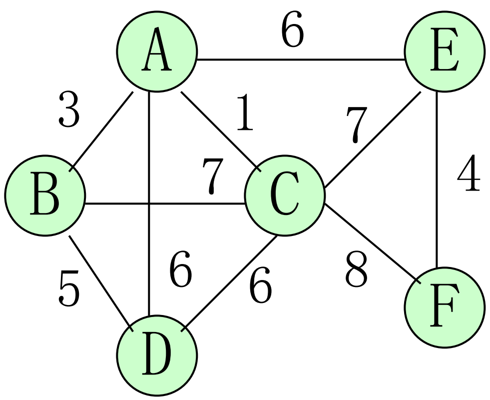
① 首先从集合V中任取一顶点(如顶点$v_0$)放入集合U中。这时U={$v_0$}，边集TE={ }

② 然后找出权值最小的边(*u*，*v*)，且$u \in$U，$v \in$(V-U)，将边加入TE，并将顶点*v*加入集合U

③ 重复上述操作直到U=V为止。这时TE中有n-1条边，T=(U，TE)就是G的一棵最小生成树

  - 如何找到连接U和V-U的最短边
    - 利用MST性质，可以用下述方法构造候选最短边集：对于V-U中的每个顶点，保存从该顶点到U中的各顶点的最短边。

# Prim算法思想：




图G

$$
\begin{array}{c}
 \\ A \\ B \\ C \\ D \\ E \\ F
\end{array}
\begin{array}{c}
\begin{array}{cccccc} A & B & C & D & E & F \end{array} \\
\left(\begin{array}{cccccc}
0 & 3 & 1 & 6 & 6 & \infty \\
3 & 0 & 7 & 5 & \infty & \infty \\
1 & 7 & 0 & 6 & 7 & 8 \\
6 & 5 & 6 & 0 & \infty & \infty \\
6 & \infty & 7 & \infty & 0 & 4 \\
\infty & \infty & 8 & \infty & 4 & 0
\end{array}\right)
\end{array}
$$

初始化

黄色表示U的顶点，其他为V-U的顶点

A

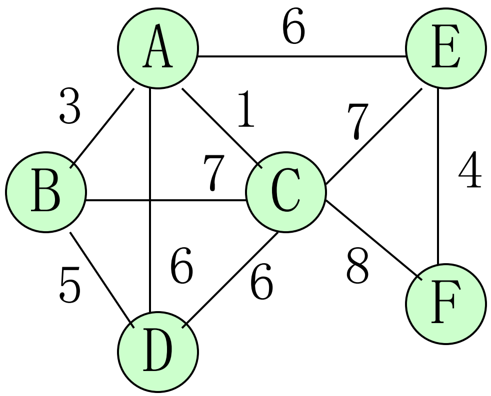
V-U中各顶点到U的最短直接路径：

| 0 | 3 | 1 | 6 | 6 | ∞ |
|---|---|---|---|---|---|
| A | B | C | D | E | F |

相邻顶点：

| | | | | | |
|---|---|---|---|---|---|
| A | A | A | A | A | A |

Prim算法:

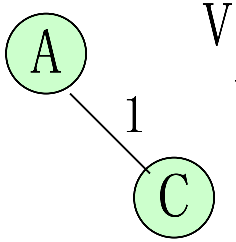



图G

$$
\begin{array}{c|cccccc}
 & A & B & C & D & E & F \\
\hline
A & 0 & 3 & 1 & 6 & 6 & \infty \\
B & 3 & 0 & 7 & 5 & \infty & \infty \\
C & 1 & 7 & 0 & 6 & 7 & 8 \\
D & 6 & 5 & 6 & 0 & \infty & \infty \\
E & 6 & \infty & 7 & \infty & 0 & 4 \\
F & \infty & \infty & 8 & \infty & 4 & 0
\end{array}
$$

比较大小

| | | | | | |
|---|---|---|---|---|---|
| 0 | 3 | 1 | 6 | 6 | ∞ |



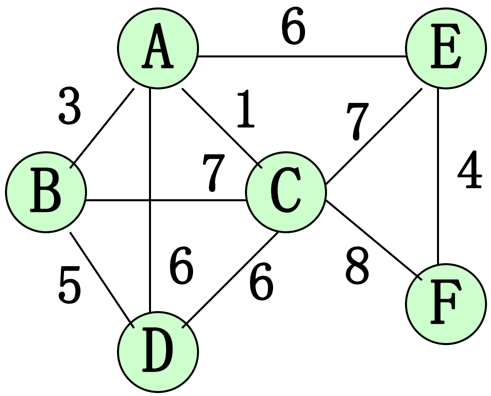

V-U中各顶点到顶点集U的最短直接路径:

| A | B | C | D | E | F |
|---|---|---|---|---|---|
| 0 | 3 | 1 | 6 | 6 | 8 |

相邻顶点:

| | | | | | |
|---|---|---|---|---|---|
| A | A | A | A | A | A |
| A | A | A | A | A | C |

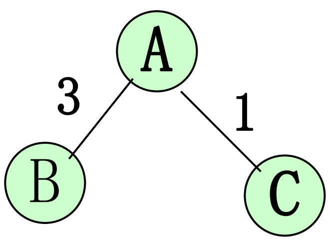
# Prim算法：




图G

$$
\begin{array}{c}
 \\ A \\ B \\ C \\ D \\ E \\ F
\end{array}
\begin{array}{c}
\begin{array}{cccccc} A & B & C & D & E & F \end{array} \\
\left(\begin{array}{cccccc}
0 & 3 & 1 & 6 & 6 & \infty \\
3 & 0 & 7 & 5 & \infty & \infty \\
1 & 7 & 0 & 6 & 7 & 8 \\
6 & 5 & 6 & 0 & \infty & \infty \\
6 & \infty & 7 & \infty & 0 & 4 \\
\infty & \infty & 8 & \infty & 4 & 0
\end{array}\right)
\end{array}
$$

比较大小




V-U中各顶点到顶点集U的最短直接路径

| | | | | | |
|---|---|---|---|---|---|
| 0 | 3 | 1 | 6 | 6 | 8 |

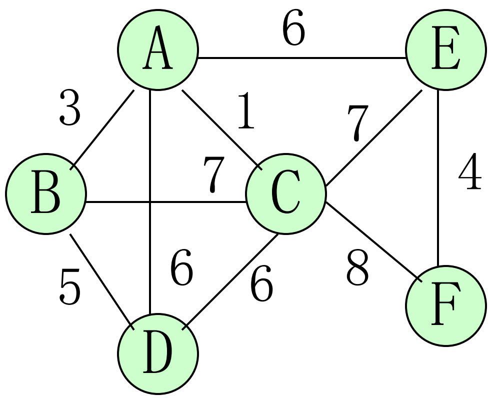
| | | | | | |
|---|---|---|---|---|---|
| 0 | 3 | 1 | 5 | 6 | 8 |

相邻顶点：

| A | B | C | D | E | F |
|---|---|---|---|---|---|
| A | A | A | A | A | C |

| | | | | | |
|---|---|---|---|---|---|
| A | A | A | B | A | C |

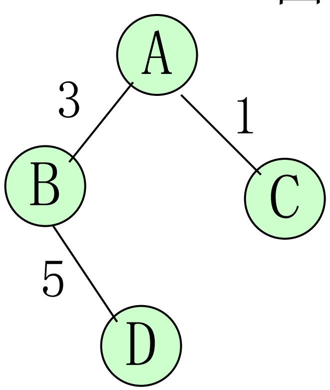
# Prime算法:




图G

$$
\begin{array}{c}
\\ A \\ B \\ C \\ D \\ E \\ F
\end{array}
\begin{array}{c}
\begin{array}{cccccc} A & B & C & D & E & F \end{array} \\
\left(\begin{array}{cccccc}
0 & 3 & 1 & 6 & 6 & \infty \\
3 & 0 & 7 & 5 & \infty & \infty \\
1 & 7 & 0 & 6 & 7 & 8 \\
6 & 5 & 6 & 0 & \infty & \infty \\
6 & \infty & 7 & \infty & 0 & 4 \\
\infty & \infty & 8 & \infty & 4 & 0
\end{array}\right)
\end{array}
$$

比较大小



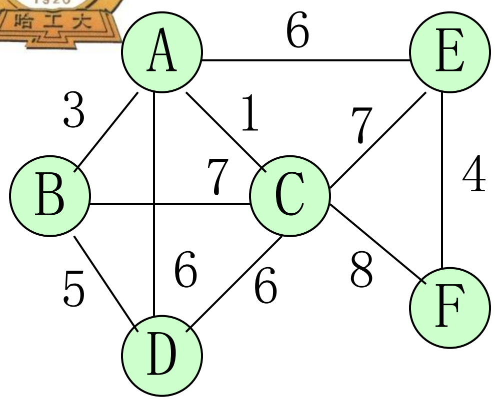

V-U中各顶点到顶点集U的最短直接路径

| | | | | | |
|---|---|---|---|---|---|
| 0 | 3 | 1 | 5 | 6 | 8 |
| 0 | 3 | 1 | 5 | 6 | 8 |
| A | B | C | D | E | F |

相邻顶点:

| | | | | | |
|---|---|---|---|---|---|
| A | A | A | B | A | C |
| A | A | A | B | A | C |

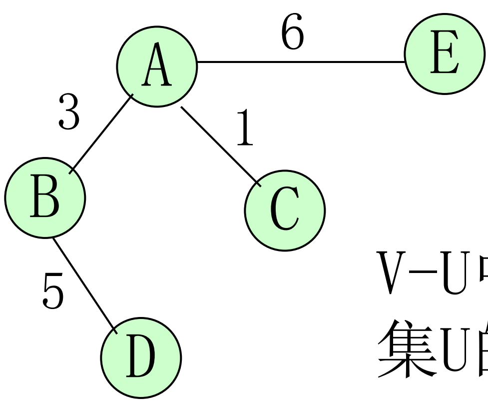
# Prime算法:




图G

$$
\begin{array}{c|c}
 & \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \\ E \\ F \end{array} &
\begin{pmatrix}
0 & 3 & 1 & 6 & 6 & \infty \\
3 & 0 & 7 & 5 & \infty & \infty \\
1 & 7 & 0 & 6 & 7 & 8 \\
6 & 5 & 6 & 0 & \infty & \infty \\
6 & \infty & 7 & \infty & 0 & 4 \\
\infty & \infty & 8 & \infty & 4 & 0
\end{pmatrix}
\end{array}
$$

比较大小



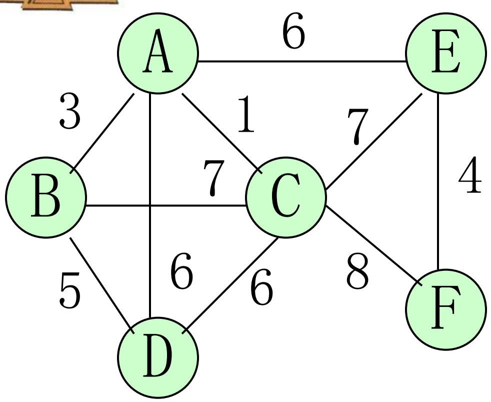

V-U中各顶点到顶点集U的最短直接路径

| | | | | | |
|---|---|---|---|---|---|
| 0 | 3 | 1 | 5 | 6 | 8 |
| 0 | 3 | 1 | 5 | 6 | 4 |

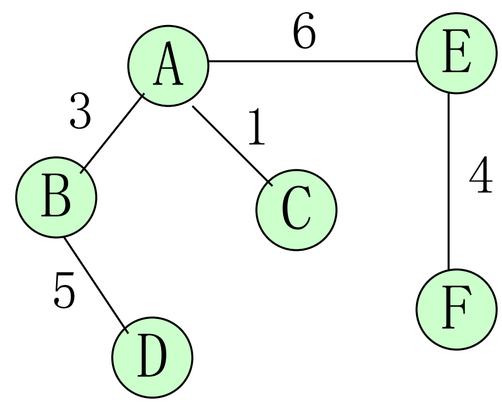
相邻顶点:

| A | B | C | D | E | F |
|---|---|---|---|---|---|
| A | A | A | B | A | C |
| A | A | A | B | A | E |

# Prime算法：




图G




| | | | | | |
|---|---|---|---|---|---|
| 0 | 3 | 1 | 5 | 6 | 4 |

U 和 V-U最短路径：

| A | B | C | D | E | F |
|---|---|---|---|---|---|
| 0 | 3 | 1 | 5 | 6 | 4 |

| | | | | | |
|---|---|---|---|---|---|
| A | A | A | B | A | E |

相邻顶点：

| | | | | | |
|---|---|---|---|---|---|
| A | A | A | B | A | E |

# 4.4 最小生成树算法(cont.)

## 普里姆（Prim）算法的实现

### 数据结构

- 数组LOWCOST[n]：用来保存集合V-U中各顶点与集合U中顶点最短边的权值，LOWCOST[v]=infinity表示顶点$v$已加入最小生成树中；
- 数组CLOSEST[n] ：用来保存依附于该边的（集合V-U中各顶点与集合U中顶点的最短边）在集合U中的顶点。

### 如何用数组LOWCOST[n]和CLOSEST[n]表示候选最短边集？

- LOWCOST[$i$]=$w$
- CLOSEST[$i$]=$k$

表示顶点$v_i$和顶点$v_k$之间的权值为$w$，其中：$v_i \in$ V-U 且$v_k \in$ U

### 如何更新？

$$\begin{cases} \text{LOWCOST[j]} = \min\{\text{cost}(v_k, v_j) \mid v_j \in U, \text{LOWCOST[j]}\} \\ \text{CLOSEST[j]} = k \end{cases}$$

# 4.4 最小生成树算法(cont.)

- 实现步骤：

1. 初始化两个辅助数组LOWCOST和CLOSEST；
2. 输出顶点$v_0$，将顶点$v_0$加入集合U中；
3. 重复执行下列操作n-1次

   3.1 在LOWCOST中选取最短边，取CLOSEST中对应的顶点序号k；

   3.2 输出顶点k和对应的权值；

   3.3 将顶点k加入集合U中；

   3.4 调整数组LOWCOST和CLOSEST；

$$\begin{cases} \mathbf{LOWCOST[j]=min\ \{\ cost\ (}v_k,\ v_j\mathbf{)\ |\ }v_j\mathbf{\in U,\ LOWCOST[j]\}} \\ \mathbf{CLOSEST[j]=}k \end{cases}$$

# 4.4 最小生成树算法(cont.)

## 普里姆（Prim）算法的实现

```
void  Prim(Costtype  C[n+1][n+1]  )
{ costtype  LOWCOST[n+1];   int  CLOSEST[n+1];   int i,j,k;  costtype  min;
      for( i=2; i<=n; i++ )
      {    LOWCOST[i] = C[1][i];      CLOSEST[i] = 1;    }
      for(  i = 2;  i <= n; i++ )
      {      min = LOWCOST[i];
              k = i;
              for(  j = 2;   j <= n;  j++ )
                      if ( LOWCOST[j] < min )
                       {  min = LOWCOST[j] ;     k=j;  }
              cout << “(”  << k << “,” << CLOSEST[k] <<  “)”  <<  end1;
              LOWCOST[k] = infinity ;
              for ( j = 2;  j <= n;  j++ )
                    if  ( C[k][j] <  LOWCOST[j] && LOWCOST[j] < infinity )
                    {     LOWCOST[j]=C[k][j];   CLOSEST[j]=k;     }
      }
} /* 时间复杂度：O( |V|² )
```

- 算法分析
- 分析Prim算法，该算法由两个并列的循环组成，第一个循环次数为vex_num(即顶点的个数n)；第二个循环，外层循环的次数为n-1，内层的循环次数为n。所以总体来说，Prim的时间复杂度为O（$n^2$），并且该算法与图中边数的多少无关，所以该算法适合于求边稠密的图的最小生成树。