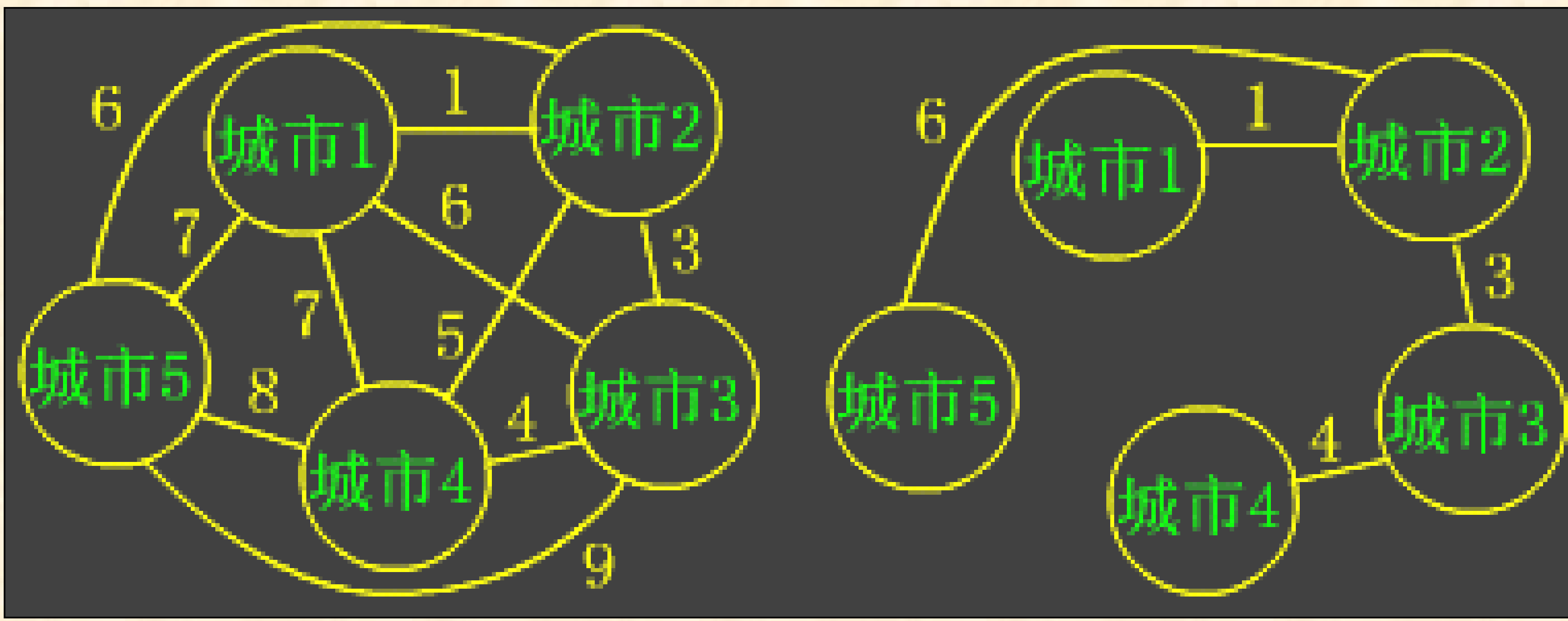
第七章 图

- **图（Graph）**是一种较线性表和树更为复杂的非线性结构。在图结构中，对结点（图中常称为顶点）的前趋和后继个数不加限制，即**结点之间的关系是任意的**。图中任意两个结点之间都可能相关。图状结构可以描述各种复杂的数据对象。
- 图的应用极为广泛，特别是近年来的迅速发展，已经渗透到诸如语言学、逻辑学、物理、化学、电讯工程、计算机科学以及数学的其它分支中。
- 图的出现最早可以追溯到1736年，著名的数学家欧拉使用它解决了经典的柯尼斯堡七桥难题。从此，有关图的理论形成了一个专门的数学分支——**图论.**

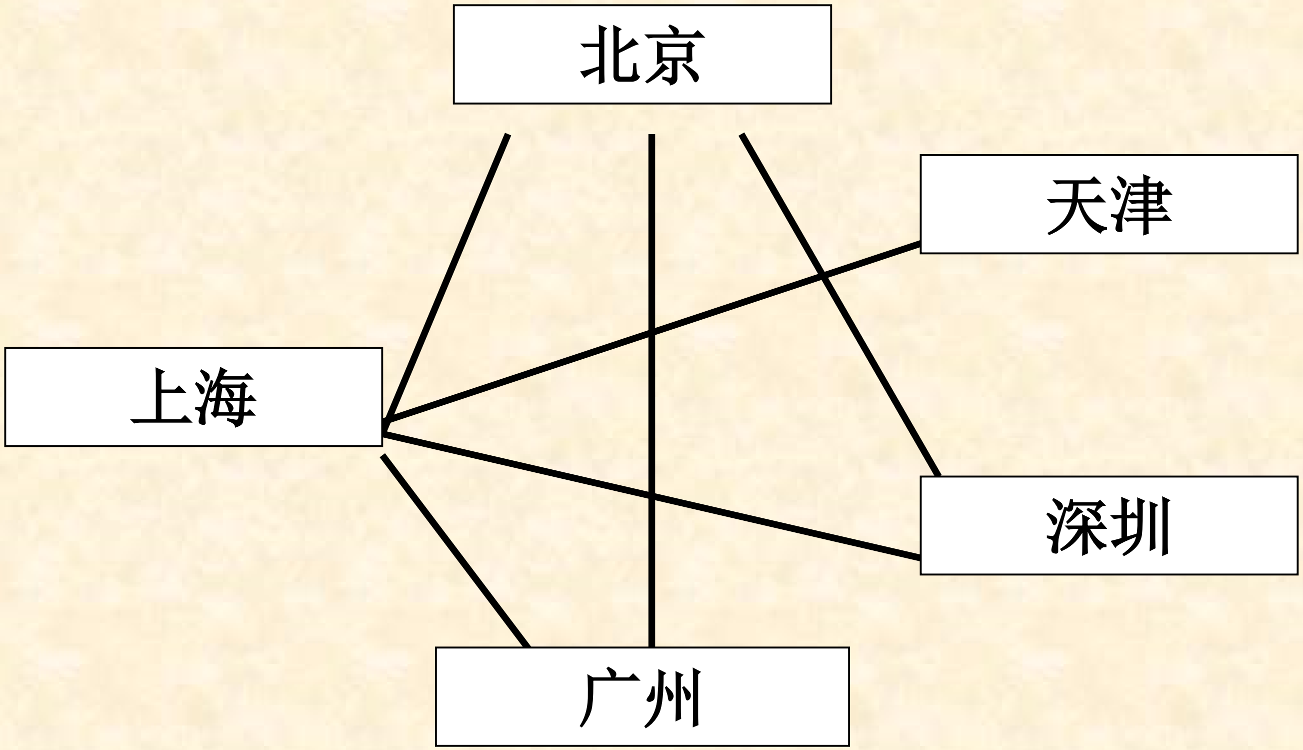
- 柯尼斯堡是18世纪初普鲁士的一个小镇，普雷格尔河流经此镇，共有7座桥横跨河上，把全镇连接起来。当时当地居民热衷于一项非常有趣的消遣活动：在星期六作一次走过所有七座桥的散步，每座桥只能经过一次而且起点与终点必须是同一地点，这就是**柯尼斯堡七桥问题**。
- 为了解决七桥问题，欧拉第一次提出了**“图”**的概念。欧拉用**点**表示岛和陆地，两点之间的**连线（边）**表示连接它们的桥，将河流、小岛和桥简化为一幅**图**。定义与顶点相连的边的数目为**顶点的度**，欧拉证明了如果这个问题有答案的话只有在每个顶点的度都是偶数的情况下才成立，而在七桥所形成的图中没有一个点具有偶数条边，因此**七桥问题不存在解**。

图状结构的实际背景

在城市之间建立**通讯网络**，使得其中任意两个城市之间都有**直接或间接**的通讯线路，假设已知每对城市之间通讯线路的造价，要求找出一个**造价最低**的通讯网络。



城市航线网



计算机网络

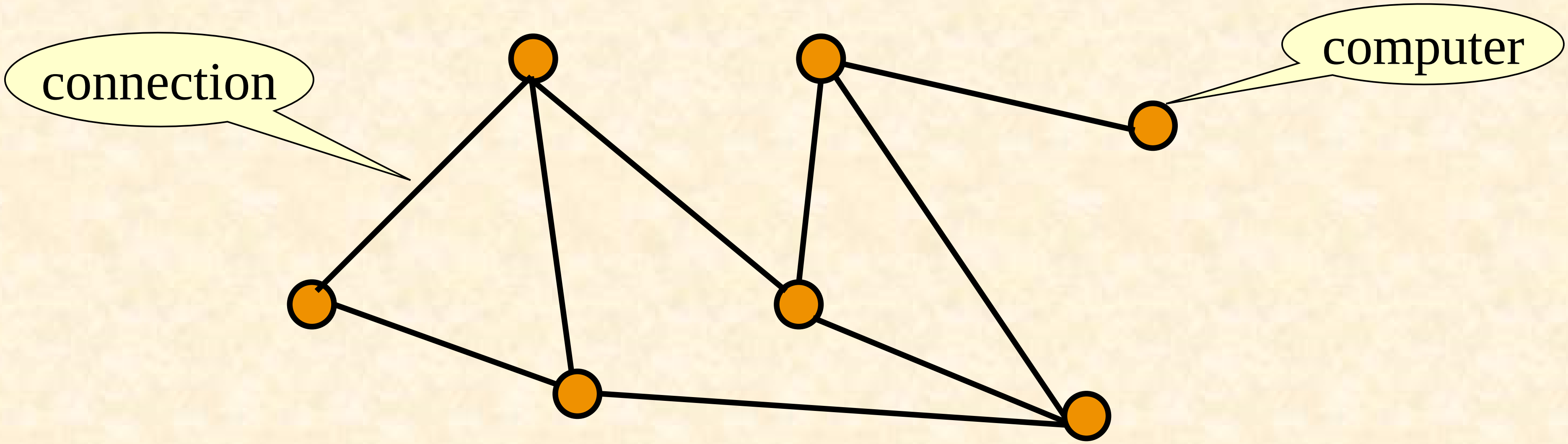


图 VS. 树

- 不一定具有一个根结点
- 没有明显的父子关系
- 从一个顶点到另一个顶点可能有多个（或0个）路径

第七章 图

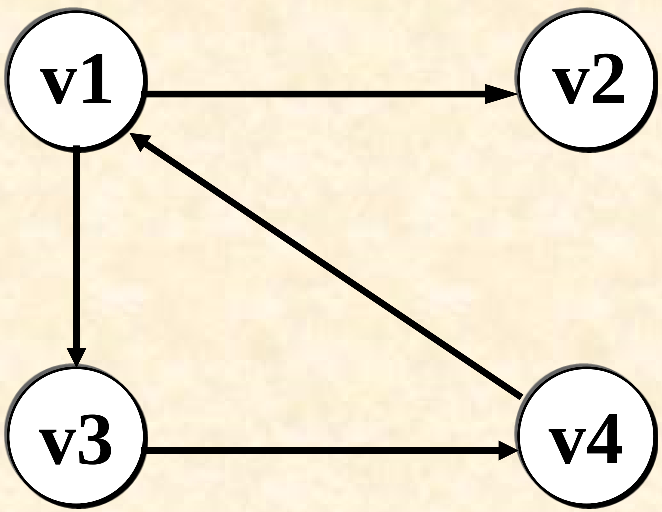
7.1 图的基本概念

定义7.1：图G由两个集合V和E组成，记为G=(V, E)；其中 V 是顶点的有限集合，E 是连接 V 中两个不同顶点的边的有限集合。通常，也将图G的顶点集和边集分别记为V(G)和E(G)。

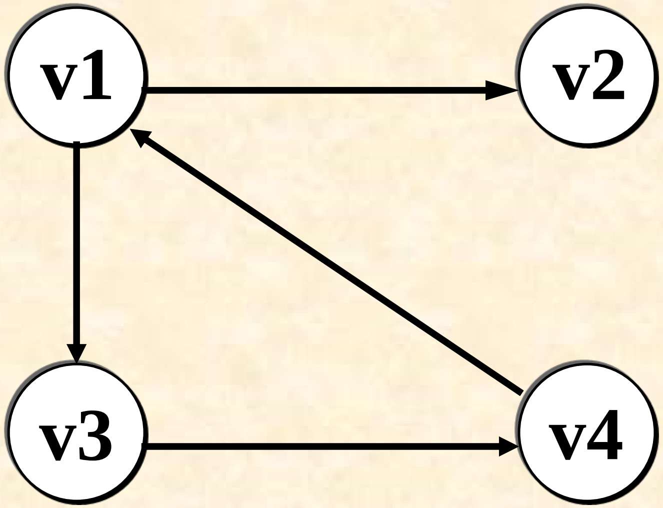
如果E中的顶点对是有序的，即E中的每条边都是有方向的，则称G为有向图。如果顶点对是无序对，则称G是无向图。

有向图

定义7.2　若G = (V, E)是有向图，则它的一条有向边是由V中两个顶点构成的有序对，亦称为弧，记为<w,v>，其中w是边的始点，又称弧尾；v是边的终点，又称弧头。



在有向图中，若存在一条边<w, v>，则称顶点w **邻接到**顶点v，顶点v**邻接自**顶点w.



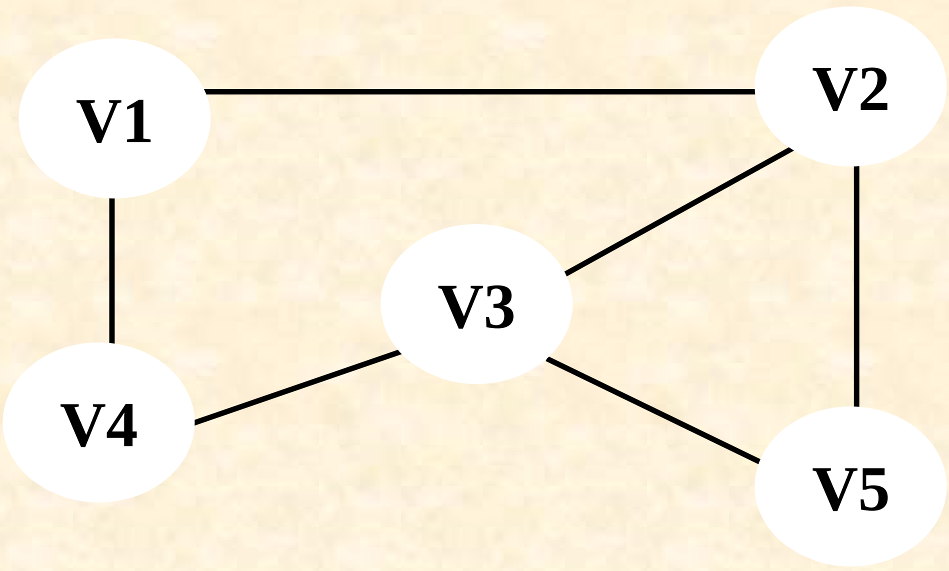
G=（V，E）

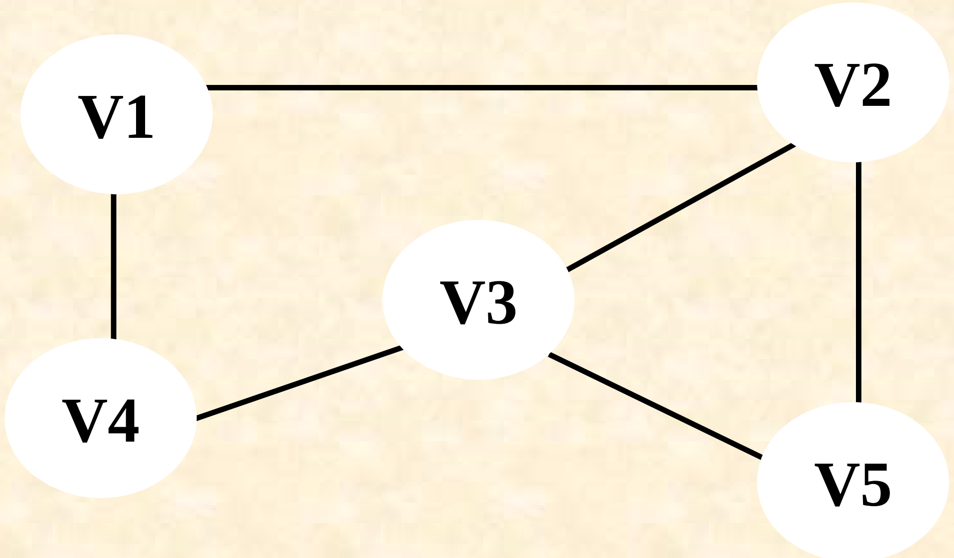
V={ v1，v2，v3，v4}

E={<v1,v2>,<v1,v3>,<v3,v4>,<v4,v1>}

无向图

定义7.3 在无向图中，若两个顶点w和v之间存在一条边(w, v)，则称w, v是相邻的，二者互为**邻接顶点**。



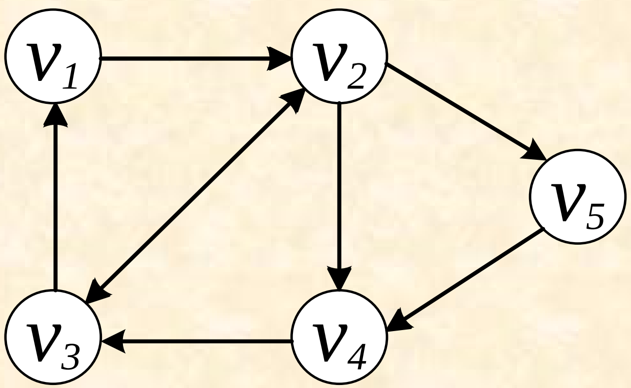


G=(V,E)

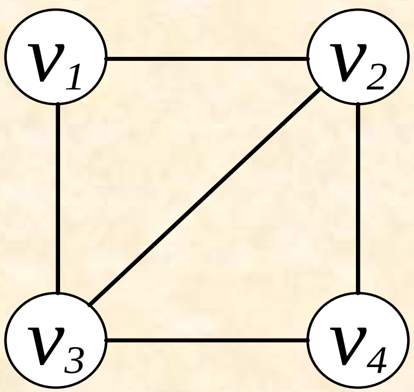
V={V1, V2, V3, V4 ,V5}

E={(V1, V4), (V1, V2), (V2, V3), (V2, V5),
(V3, V4), (V3, V5) }

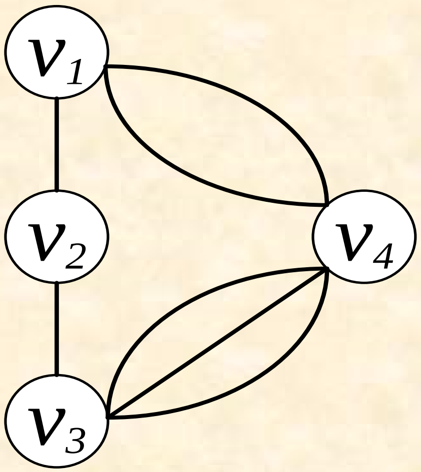
定义7.4 由于E是边的集合，故一个图中不会多次出现一条边。若去掉此限制，则由此产生的结构称为多重图。图 (c)就是一个多重图。



(a)



(b)

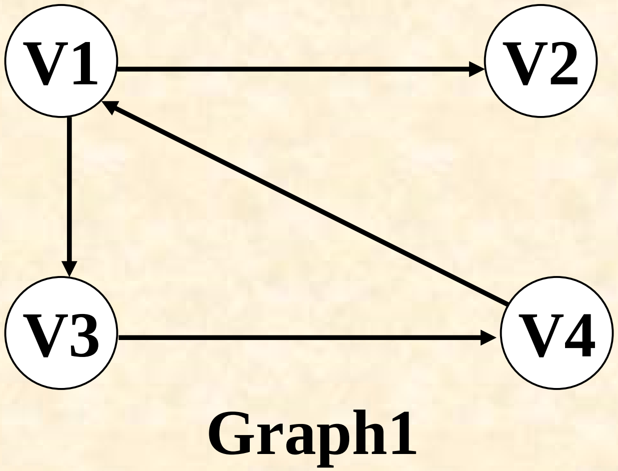


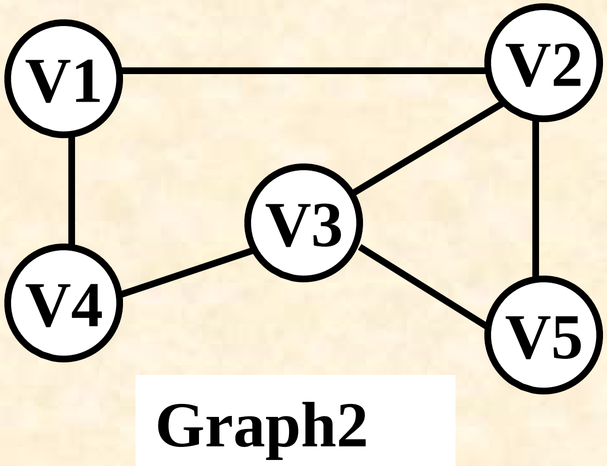
(c)

定义7.5 设G是无向图，$v \in V(G)$，E(G)中以v为端点的边的个数，称为顶点的**度**。若G是有向图，则v的**出度**是以v为始点的边的个数，v的**入度**是以v为终点的边的个数。

- 有向图中，以某顶点为弧头的弧的数目称为该顶点的**入度**。以某顶点为弧尾的弧的数目称为该顶点的**出度**。

顶点的度=入度+出度。





- **度: D(v)**
- **入度: ID(v)**
- **出度: OD(v)**
- **D(v)=ID(v)+OD(v)**

设图G(可以为有向或无向图)共有n个顶点，e条边，若顶点v_i的度数为$D(v_i)$，则

$$\sum_{i=0}^{n-1} D(v_i) = 2e$$

因为一条边关联两个顶点，而且使得这两个顶点的度数分别增加1。因此顶点的度数之和就是边的两倍。

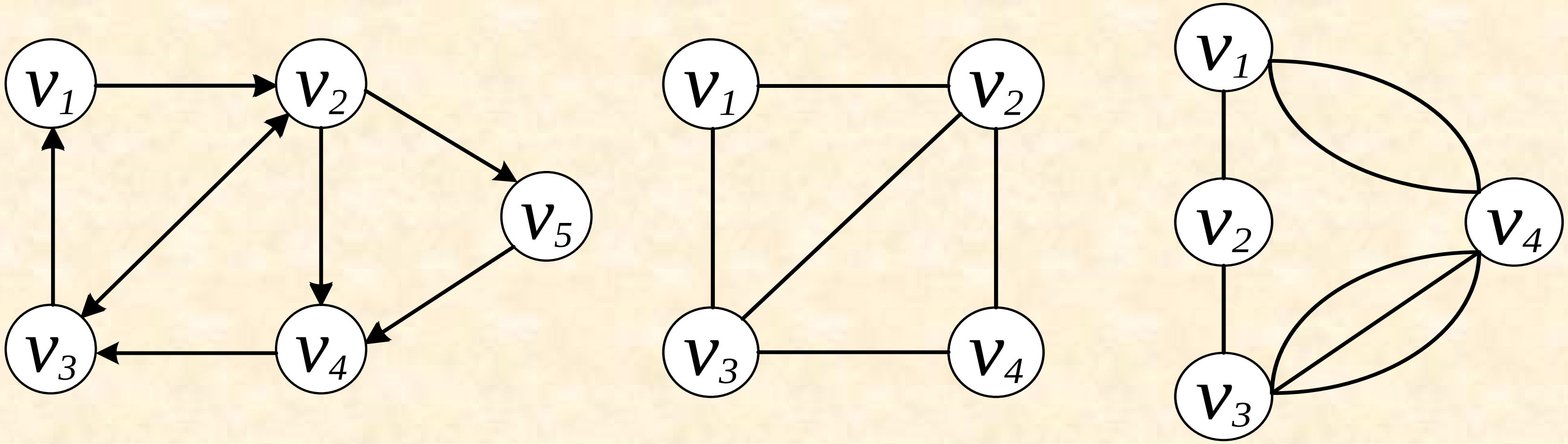
- **定义7.6** 设G是图，若存在一个顶点序列

$$v_p, v_1, v_2, \cdots, v_{q-1}, v_q$$

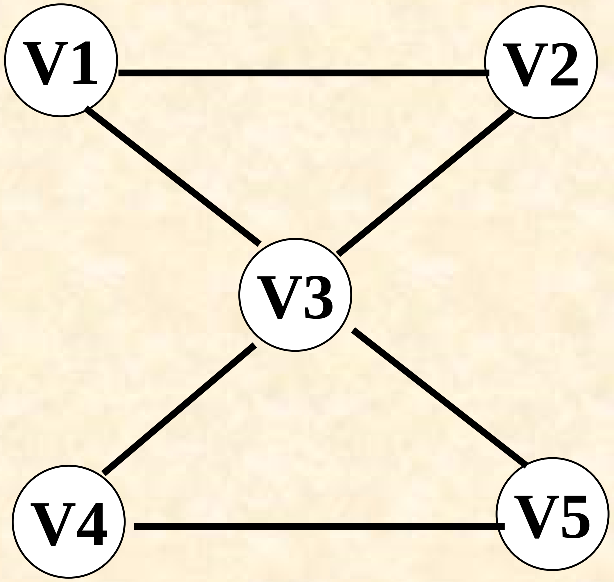
使得 $<v_p, v_1>, <v_1, v_2>, \cdots, <v_{q-1}, v_q>$ 或 $(v_p, v_1), (v_1, v_2), \cdots, (v_{q-1}, v_q)$ 属于E(G)，则称 v_p 到 v_q 存在一条路径，其中 v_p 称为起点，v_q 称为终点。

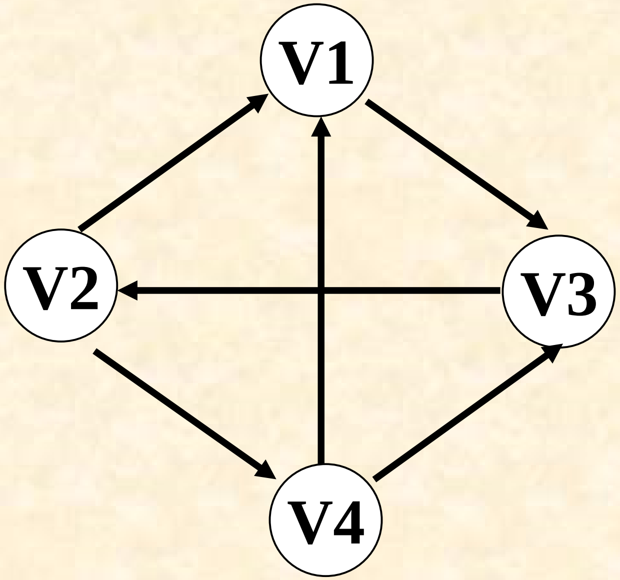
路径的长度是该路径上边的个数。如果一条路径上除了起点和终点可以相同外，再不能有相同的顶点，则称此路径为简单路径。如果一条简单路径的起点和终点相同，且路径长度大于等于2，则称之为简单回路。

- 图(a)中，*v*1到*v*3之间存在一条路径*v*1, *v*2, *v*5, *v*4, *v*3，同时这也是一条简单路径；*v*1, *v*2, *v*5, *v*4, *v*3, *v*1是一条简单回路。



(a) (b) (c)





路径：v1 v3 v4 v3 v5

简单路径：v1 v3 v5

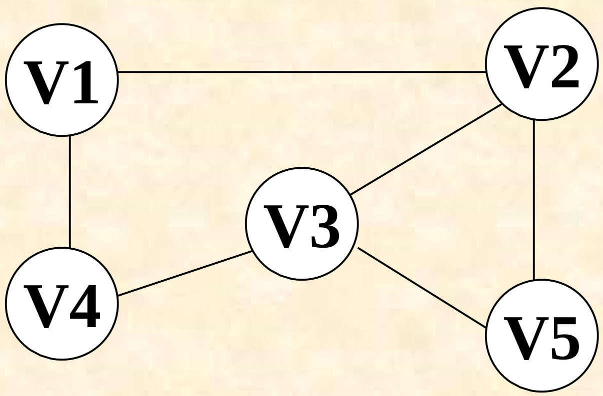
简单回路：v1 v2 v3 v1

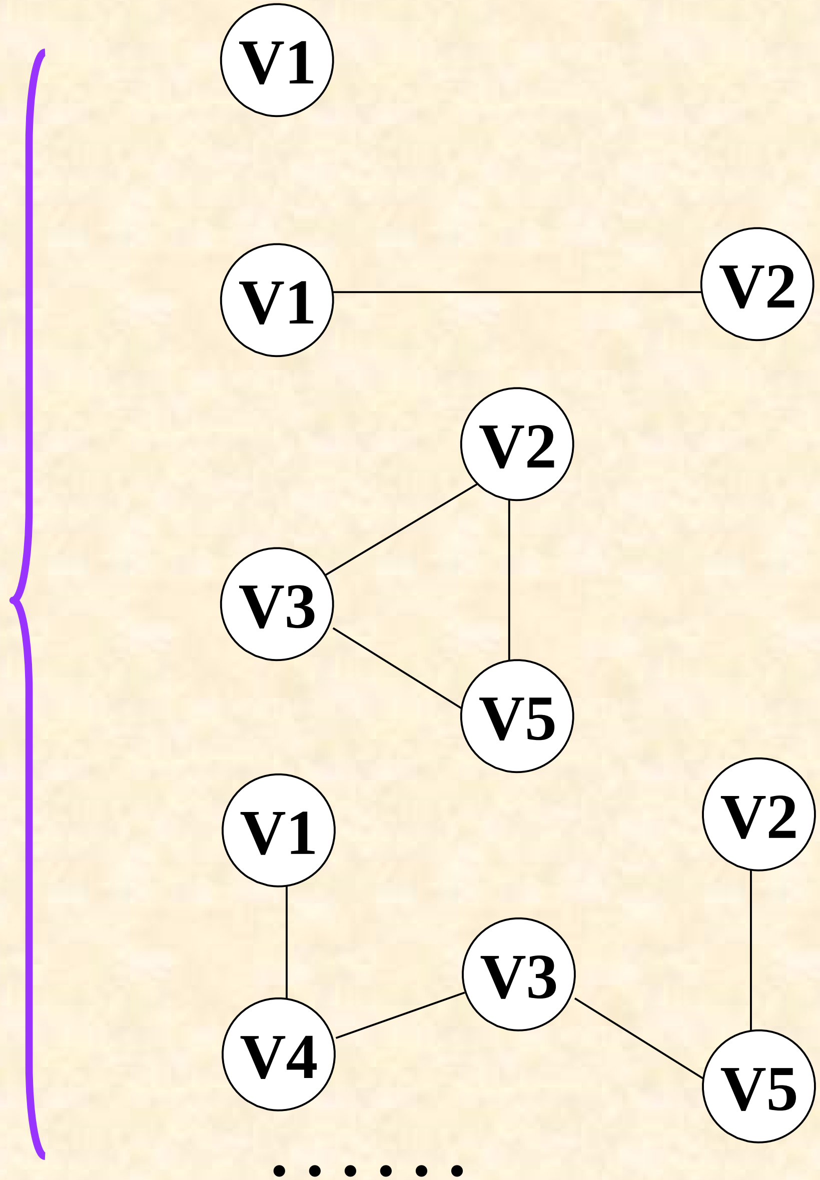
路径：v1 v3 v2 v4 v3 v2

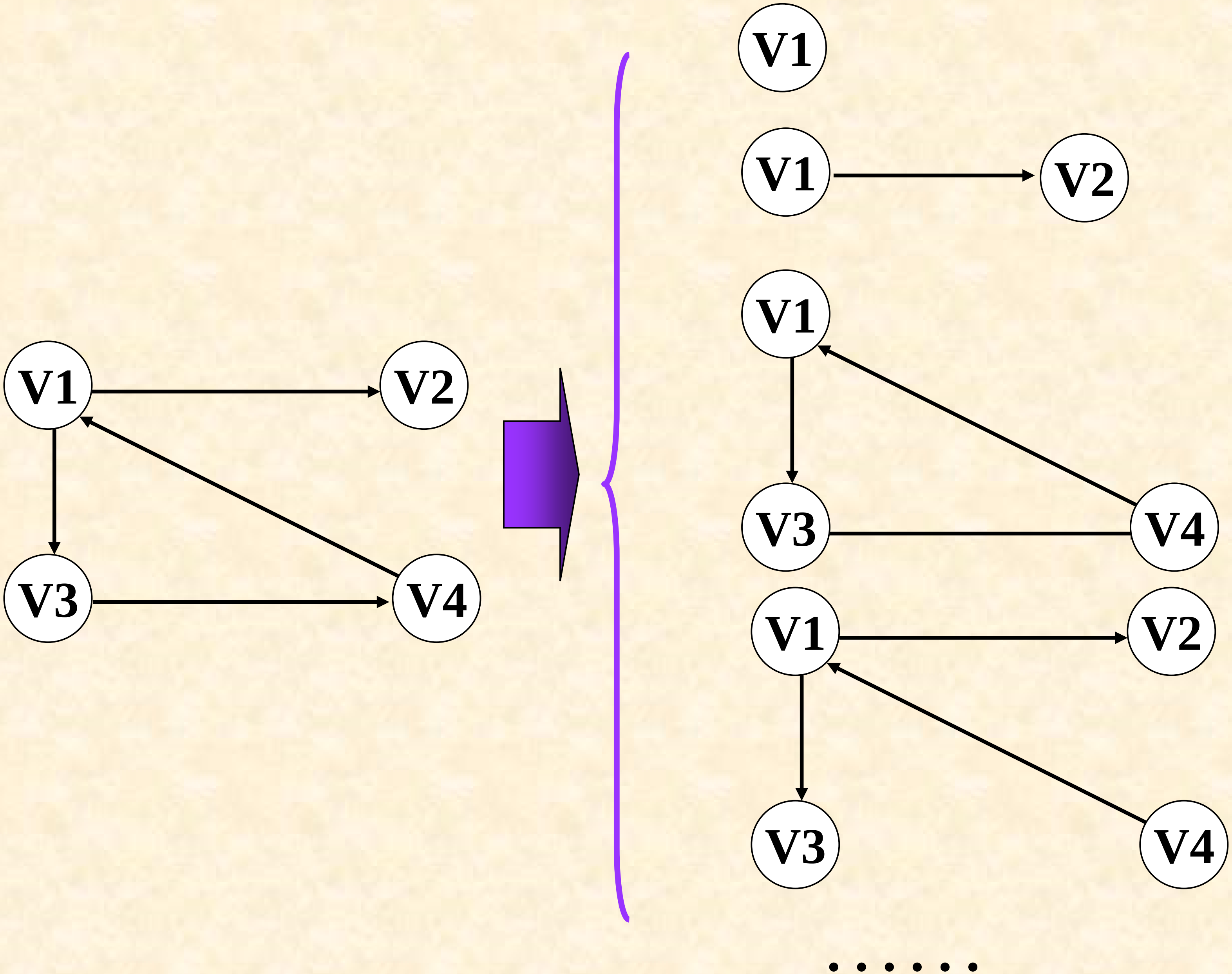
简单路径：v1 v3 v2

简单回路：v1 v3 v2 v1

定义7.7　设G，H是图，如果V(H) $\subseteq$ V(G)，E(H) $\subseteq$ E(G)，则称H是G的子图，G是H的母图。如果H是G的子图，并且V(H) = V(G)，则称H为G的支撑子图。

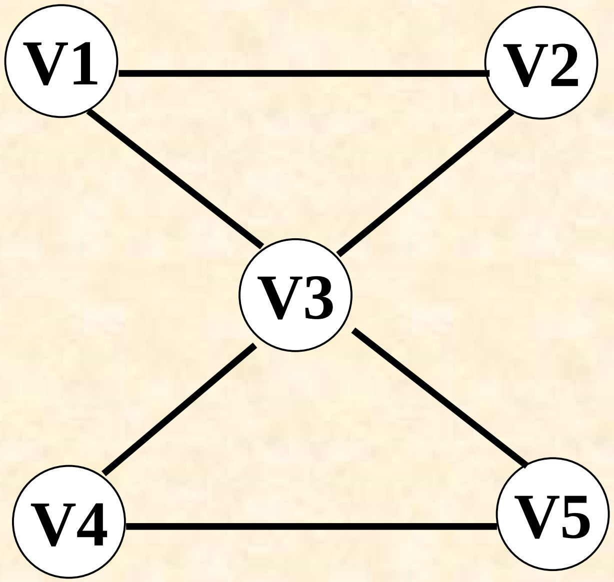

V1
V2
V3
V4
V5

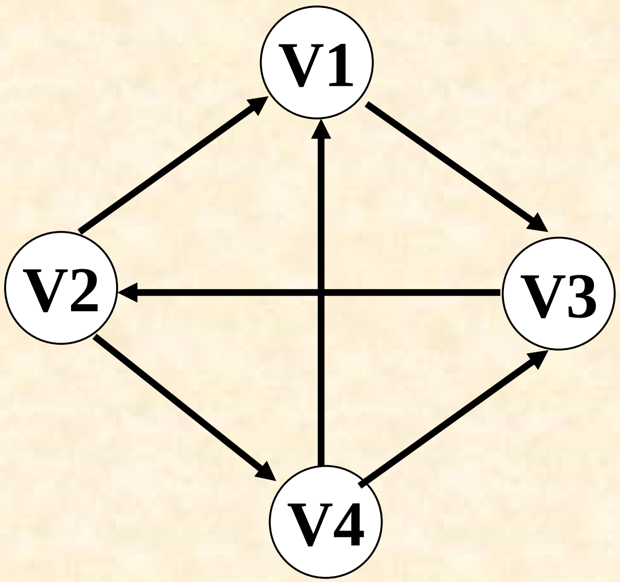

V1
V1
V2
V2
V3
V5
V1
V2
V3
V4
V5
……


V1
V2
V3
V4
V1
V1
V2
V1
V3
V4
V1
V2
V3
V4
……

◆定义7.8 设G是图，若存在一条从顶点v_i到顶点v_j的路径，则称v_i与v_j可及(连通)。若G为无向图，且V(G)中任意两顶点都可及，则称G为连通图。若G为有向图，且对于V(G)中任意两个顶点v_i和v_j， v_i与v_j可及，v_j与v_i也可及，则称G为强连通图。

◆也可以定义“弱连通图”的概念，即在任何顶点u和v之间，至少存在一条从u到v的路径或者存在一条从v到u的路径。

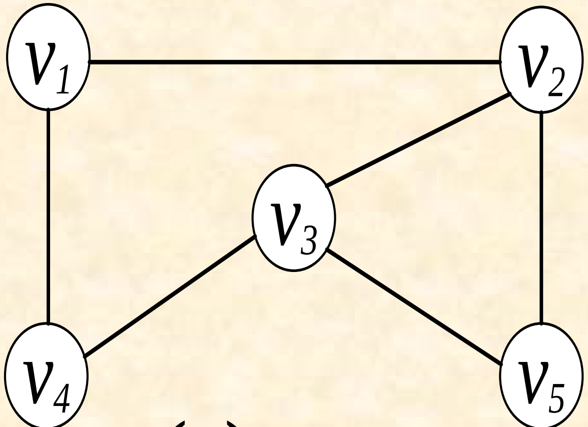

V1
V2
V3
V4
V5


V1
V2
V3
V4

◆**定义7.9** 设图G = (V，E)是无向（或有向）图，若G的子图G_K是一个（强）连通图，则称G_K 为G的（强）连通子图。

◆**定义7.10** 对于G的一个连通子图G_K，如果不存在G的另一个连通子图G'，使得$V(G_K) \subset V(G')$，则称G_K为G的<u>连通分量</u>。

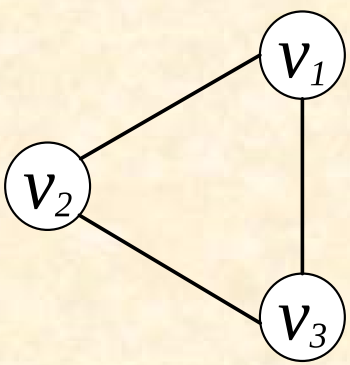
一个图的连通子图



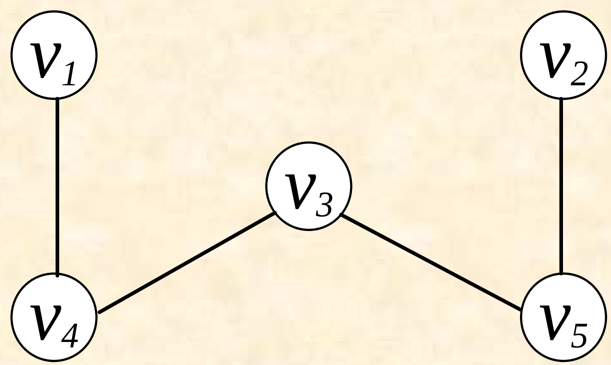
(a)

(b)

(c)



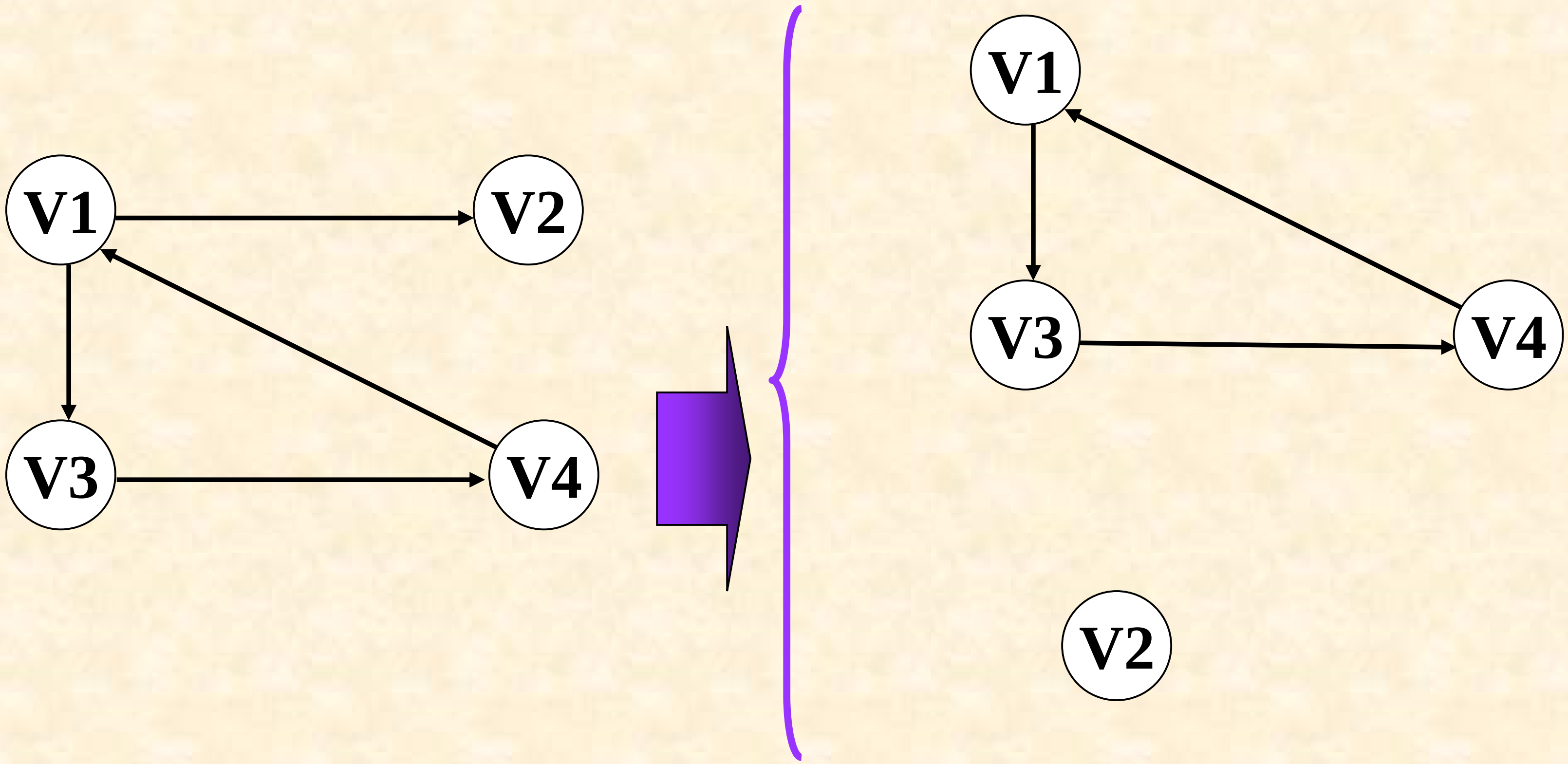
(d)



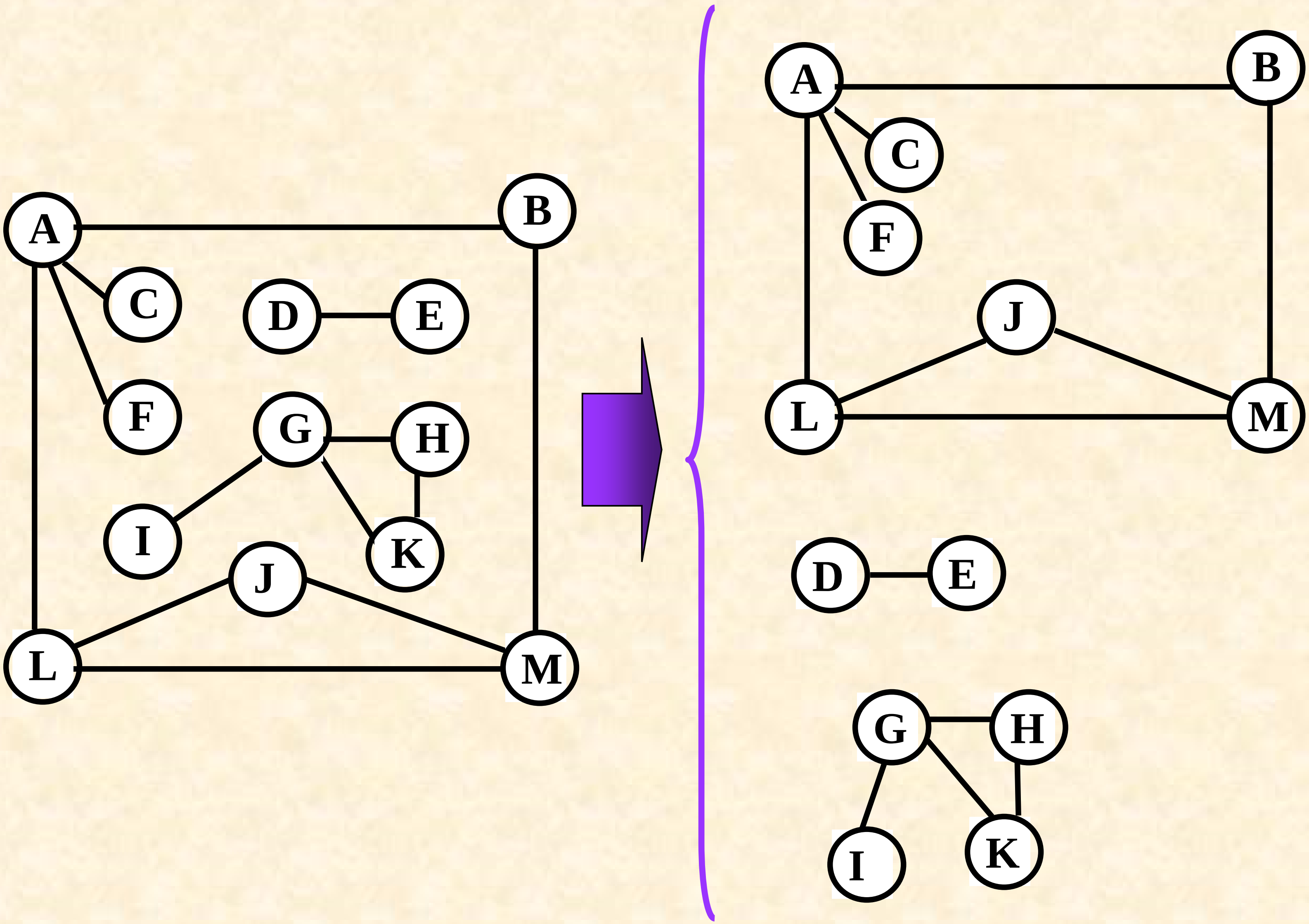
(e)

e是a的连通分量

连通分量



连通分量



- 有时候，图不仅要表示出元素之间是否存在某种**关系**，同时还需要表示与这一关系**相关的某些信息**。
- 例如在计算机网络对应的图中，顶点表示计算机，顶点之间的边表示计算机之间的通讯链路。实际中，为了管理计算机网络，我们需要这个图包含更多的信息，例如每条通讯链路的物理长度、成本和带宽等信息。为此，我们为传统图中的**每条边添加相应的数据域以记录所需要的信息**。

◆**定义7.11** 设G = (V, E)是图，若对图中的任意一条边l，都有实数$w(l)$与其对应，则称G为<u>权图</u>，记为G = (V, E, w)。记$w(u,v)$表示$w((u,v))$或$w(\langle u,v\rangle)$，规定：

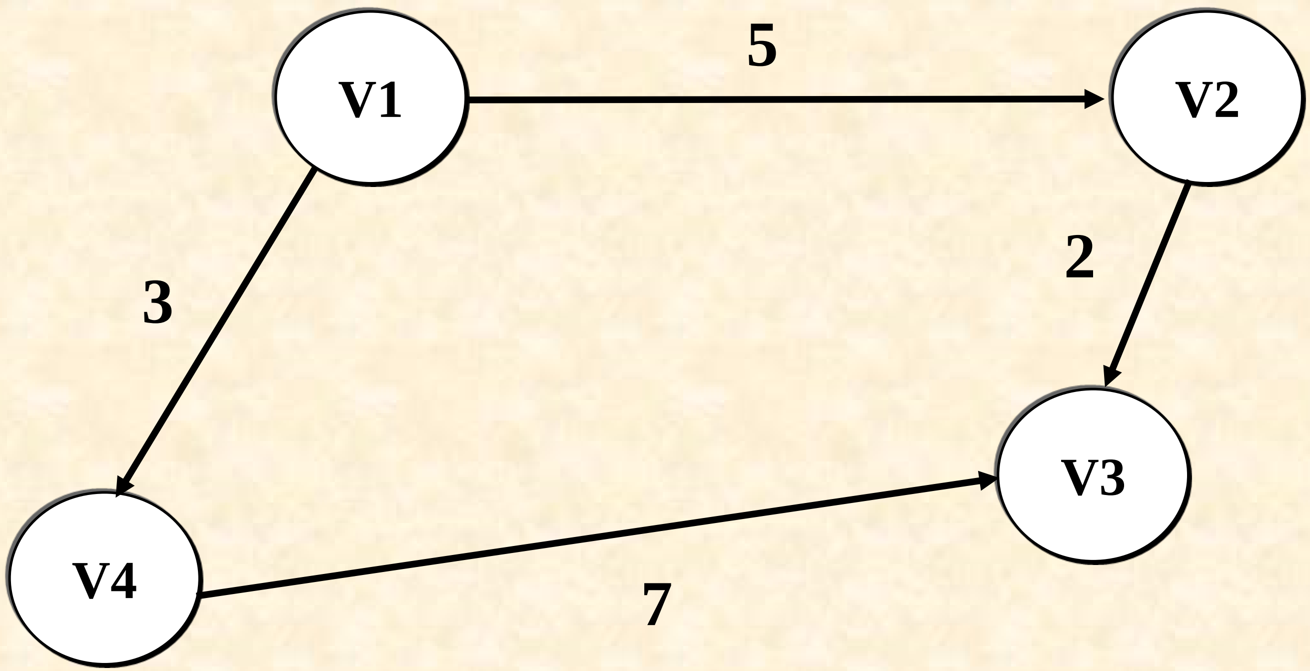
$\forall u\in V$, 有$w((u,u))=0$或$w(\langle u,u\rangle)=0$

$\forall u,v\in V$, 若$(u,v)\notin E(G)$或$\langle u,v\rangle\notin E(G)$

则$w((u,v))=+\infty$或$w(\langle u,v\rangle)=+\infty$

◆**定义7.12** 若$\sigma=(v_0,v_1,v_2,\cdots,v_k)$是权图G中的一条路径，则$|\sigma|=\sum_{i=1}^{k}w(v_{i-1},v_i)$称为<u>加权路径</u>$\sigma$的长度或权重。

◆权通常用来表示从一个顶点到另一个顶点的距离或费用。


V1
5
V2
2
3
V3
V4
7

无向图	有向图
端点	弧　弧头　弧尾
相邻的	邻接到　邻接自
度	出度　入度
连通图	强连通图

第七章 图

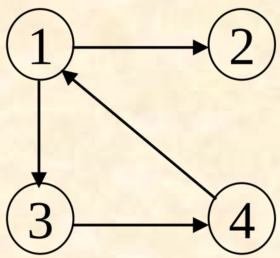
图的存储结构

- 多重链表
- 邻接矩阵
- 关联矩阵
- 邻接表(逆邻接表)

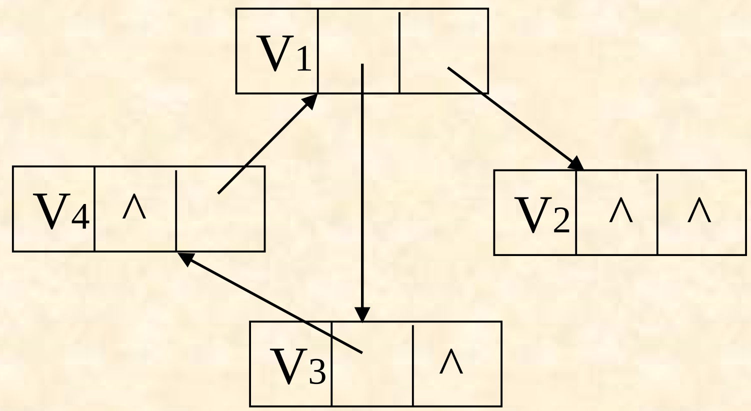
1、图的存储结构-多重链表

- 多重链表

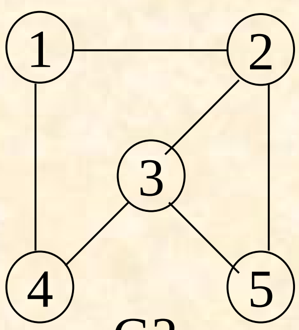
例



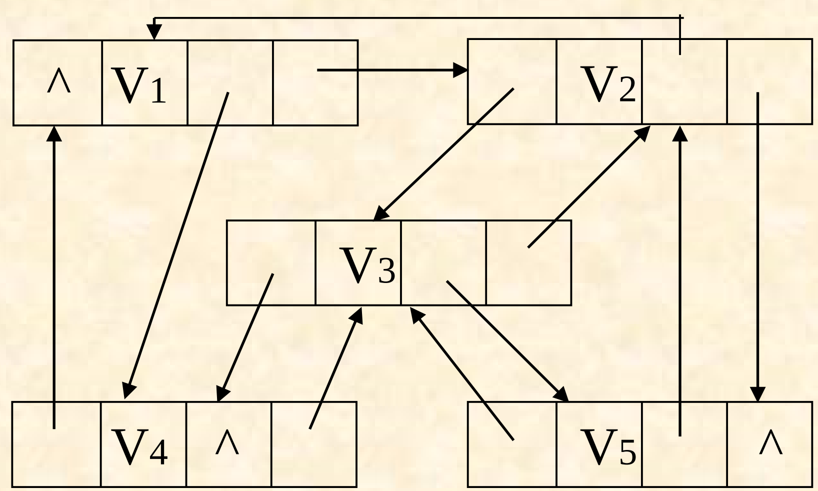
G1



例



G2



2、邻接矩阵

用顺序方式或链接方式存储图的顶点表$v_0,v_1,\ldots v_{n-1}$，图的边用n×n阶矩阵A =(a_{ij})表示，A的定义如下：

（a）若图为权图，a_{ij}对应边$<v_i,v_j>$的权值；

（b）若图为非权图，则

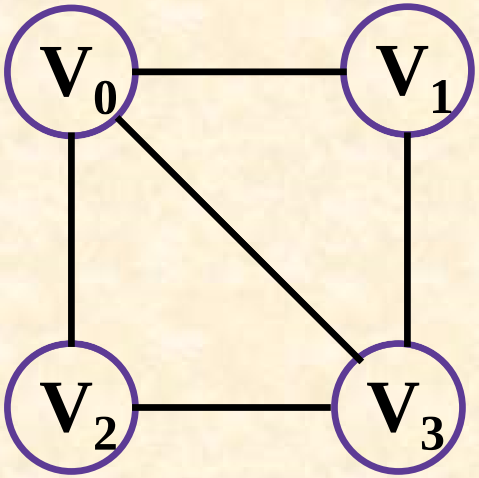
（1） $a_{ii}=0$；

（2） $a_{ij}=1$，当$i\neq j$且$<v_i,v_j>$或(v_i,v_j)存在时；

（3）$a_{ij}=0$，当$i\neq j$且$<v_i,v_j>$或(v_i,v_j)不存在时。

称矩阵A为图的邻接矩阵。

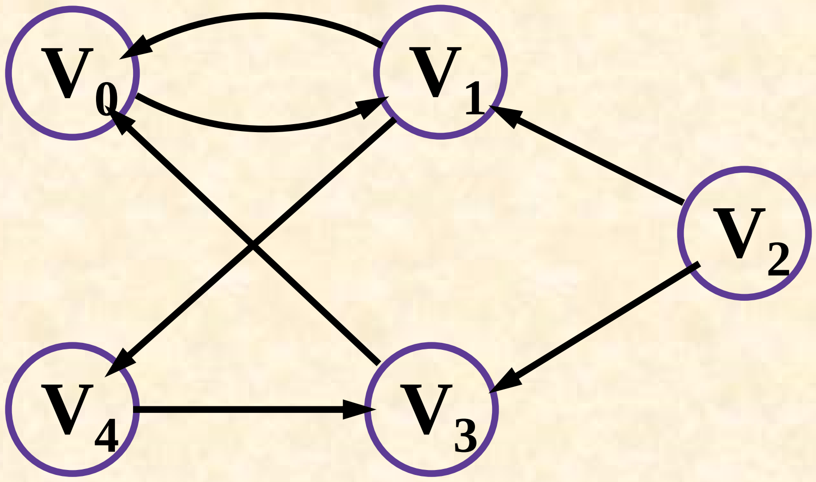
[例1]无向图的邻接矩阵



$$\begin{array}{c c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

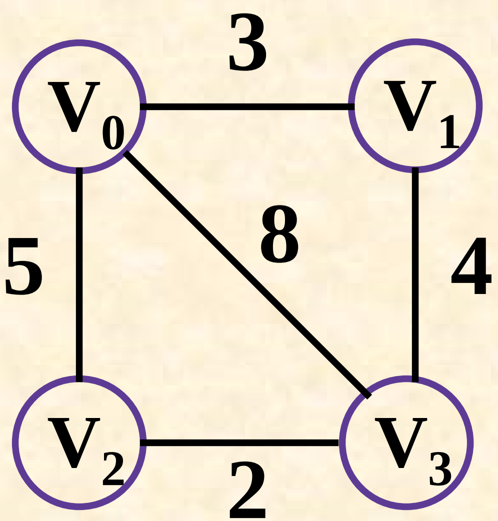
无向图的邻接矩阵是对称阵。

[例2]有向图的邻接矩阵



$$
\begin{array}{c}
 \\ 0 \\ 1 \\ 2 \\ 3 \\ 4
\end{array}
\begin{array}{c}
\begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\
\begin{pmatrix}
0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0
\end{pmatrix}
\end{array}
$$

[例3]权图的邻接矩阵



$$
\begin{array}{c}
\\
\begin{array}{c} \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \end{array}
\end{array}
\begin{array}{c}
\begin{array}{cccc} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} \end{array} \\
\begin{pmatrix}
\mathbf{0} & 3 & 5 & 8 \\
3 & \mathbf{0} & \infty & 4 \\
5 & \infty & \mathbf{0} & 2 \\
8 & 4 & 2 & \mathbf{0}
\end{pmatrix}
\end{array}
$$

特点：

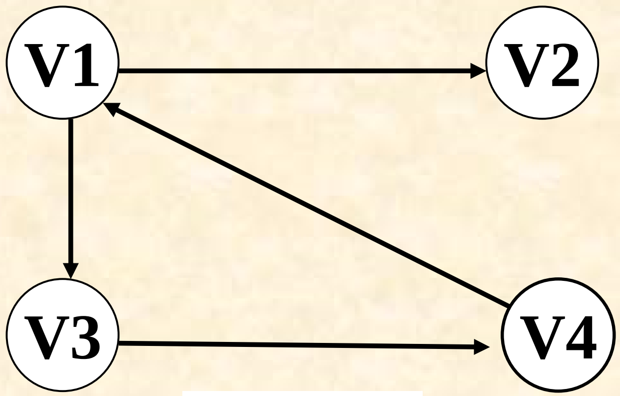
无向图的邻接矩阵对称，可压缩存储，有**n**个顶点的无向图需存储空间为**n(n+1)/2**

有向图邻接矩阵不一定对称，有**n**个顶点的有向图需存储空间为n^2

借助邻接矩阵，可以很容易地求出图中顶点的度。

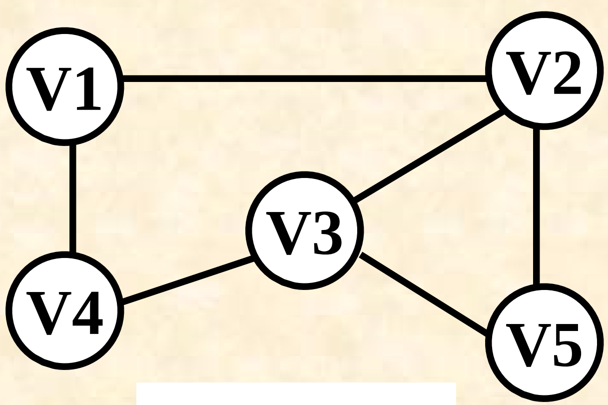
–无向图　邻接矩阵的第i行（或第i列）的非零元素的个数是顶点V_i的度。

–有向图　邻接矩阵第i行的非零元素的个数为顶点V_i的出度；第i列的非零元素的个数为顶点V_i的入度。



Graph1

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



Graph2

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

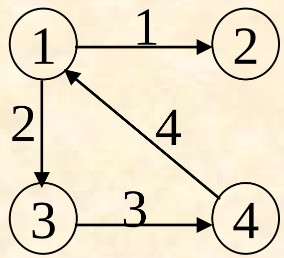
3、图的存储结构-关联矩阵

- 关联矩阵——表示顶点与边的关联关系的矩阵
  - 定义：设G=(V,E)是有n≥1个顶点，e≥0条边的图，G的关联矩阵A是具有以下性质的n×e阶矩阵

$$\text{有向图：} A[i,j]=\begin{cases}1, i\text{顶点与}j\text{边相连，且}i\text{为尾}\\0, i\text{顶点与}j\text{边不相连}\\-1, i\text{顶点与}j\text{边相连，且}i\text{为头}\end{cases}$$

$$\text{无向图：} A[i,j]=\begin{cases}1, \quad i\text{顶点与}j\text{边相连}\\0, \quad i\text{顶点与}j\text{边不相连}\end{cases}$$

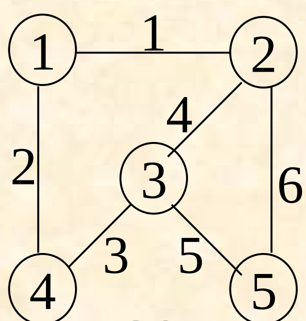
例



G1

$$
\begin{array}{c}
\\ ① \\ ② \\ ③ \\ ④
\end{array}
\begin{array}{c}
\begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\
\begin{bmatrix}
1 & 1 & 0 & -1 \\
-1 & 0 & 0 & 0 \\
0 & -1 & 1 & 0 \\
0 & 0 & -1 & 1
\end{bmatrix}
\end{array}
$$

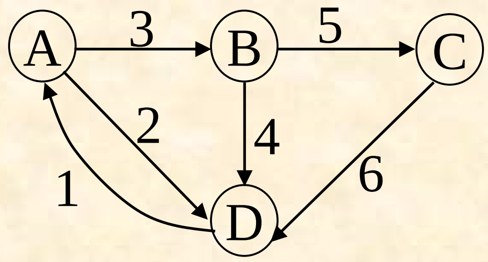
例



G2

$$
\begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
① & 1 & 1 & 0 & 0 & 0 & 0 \\
② & 1 & 0 & 0 & 1 & 0 & 1 \\
③ & 0 & 0 & 1 & 1 & 1 & 0 \\
④ & 0 & 1 & 1 & 0 & 0 & 0 \\
⑤ & 0 & 0 & 0 & 0 & 1 & 1
\end{array}
$$

例



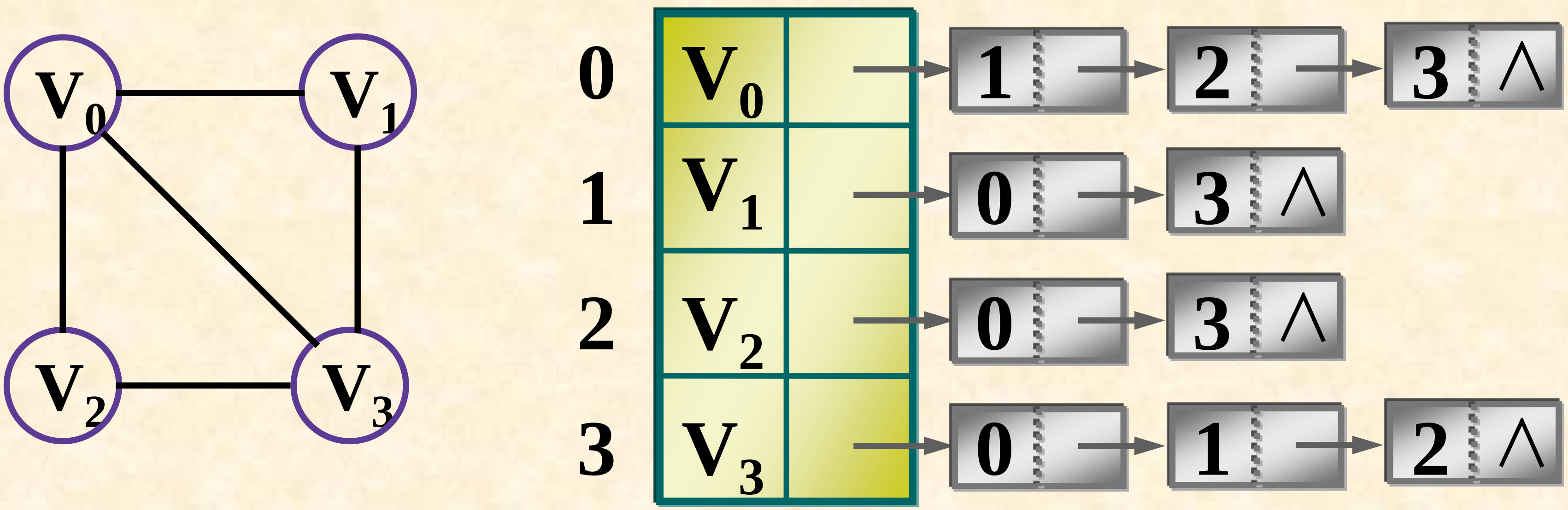
$$
\begin{array}{c c}
 & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \end{array} &
\begin{bmatrix}
-1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & -1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & -1 & 1 \\
1 & -1 & 0 & -1 & 0 & -1
\end{bmatrix}
\end{array}
$$

❑ 特点

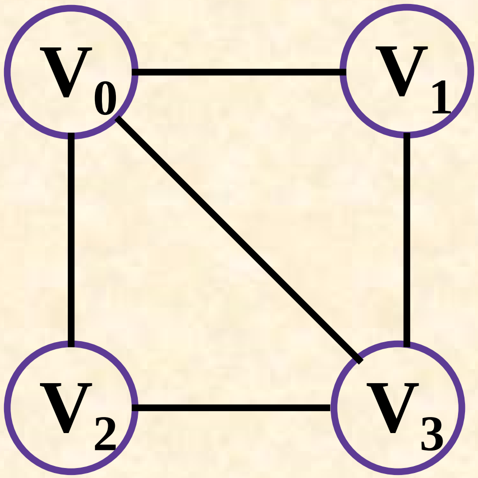
- 关联矩阵每列只有两个非零元素，是稀疏矩阵；n越大，零元素比率越大
- 无向图中顶点V_i的度$TD(V_i)$是关联矩阵A中第i行元素之和
- 有向图中，
  - ✧ 顶点V_i的出度是A中第i行中“1”的个数
  - ✧ 顶点V_i的入度是A中第i行中“-1”的个数

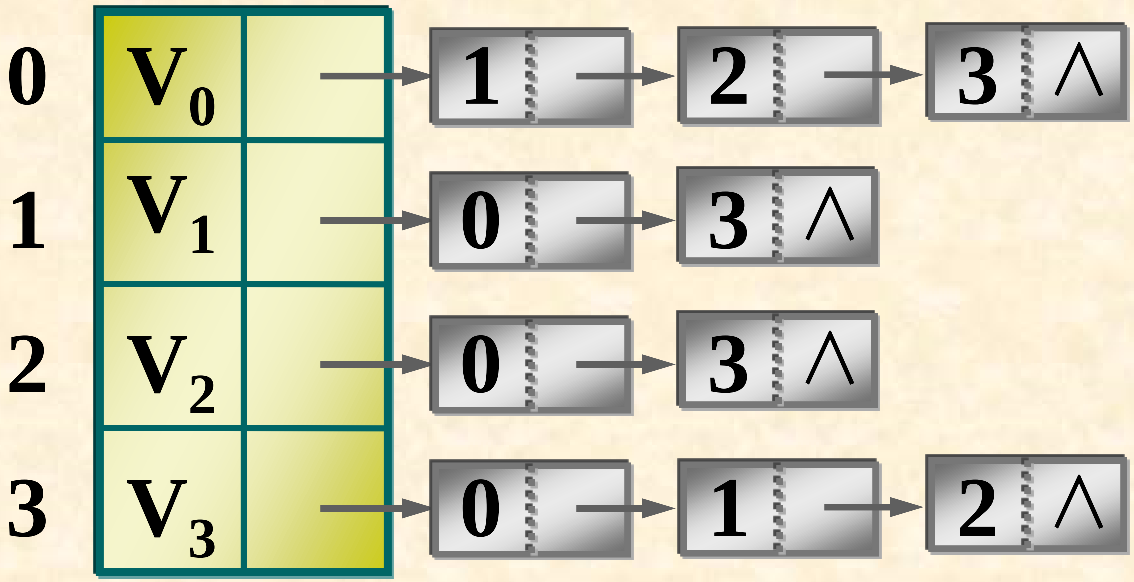
4、邻接表

邻接表是图的一种链式存储结构。对图的每个顶点建立一个单链表（n个顶点建立n个单链表），第i个单链表中的结点包含顶点V_i的所有邻接顶点。由顺序存储的顶点表和链接存储的边链表构成的图的存储结构被称为邻接表。



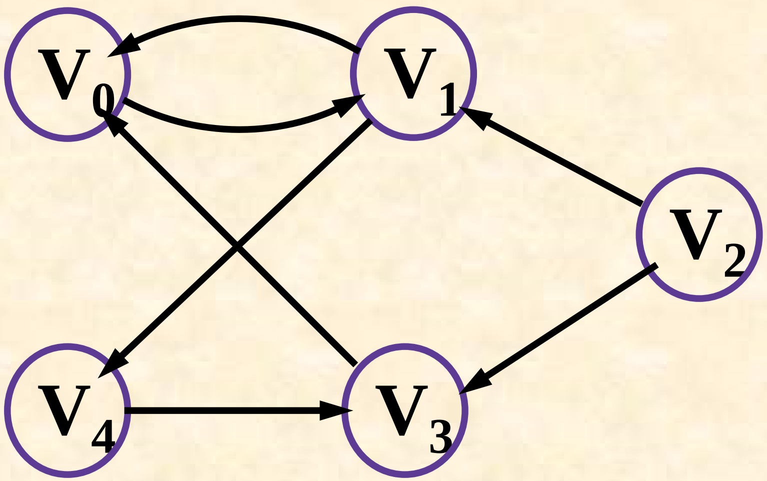
[例1]无向图的邻接表

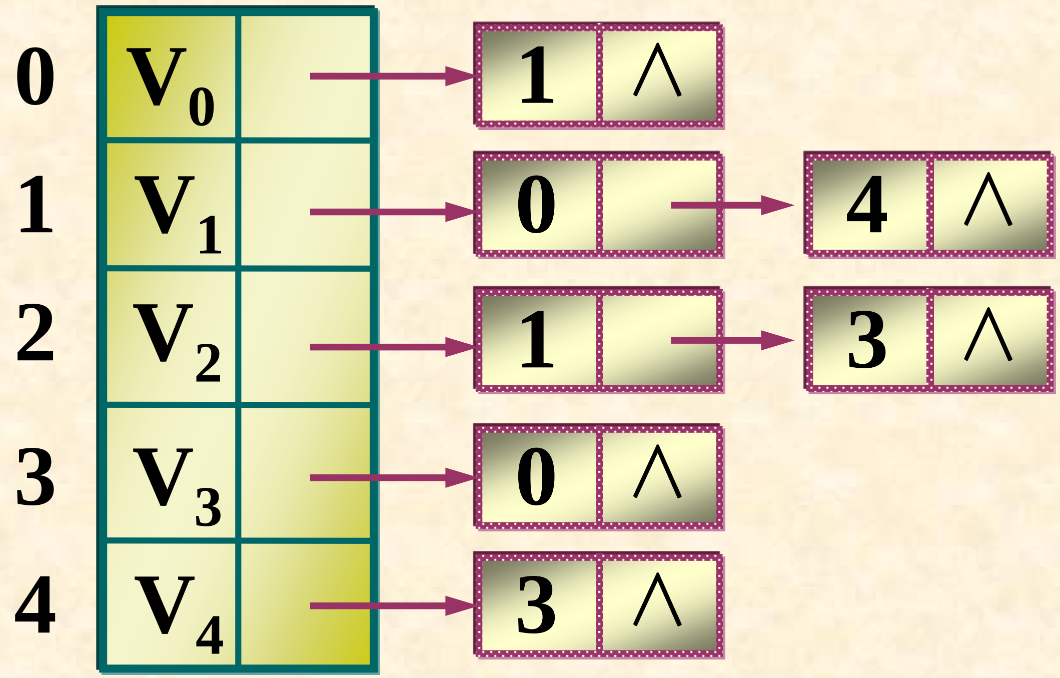




无向图中顶点V_i的度为第i个单链表中的结点数

[例2]有向图的邻接表

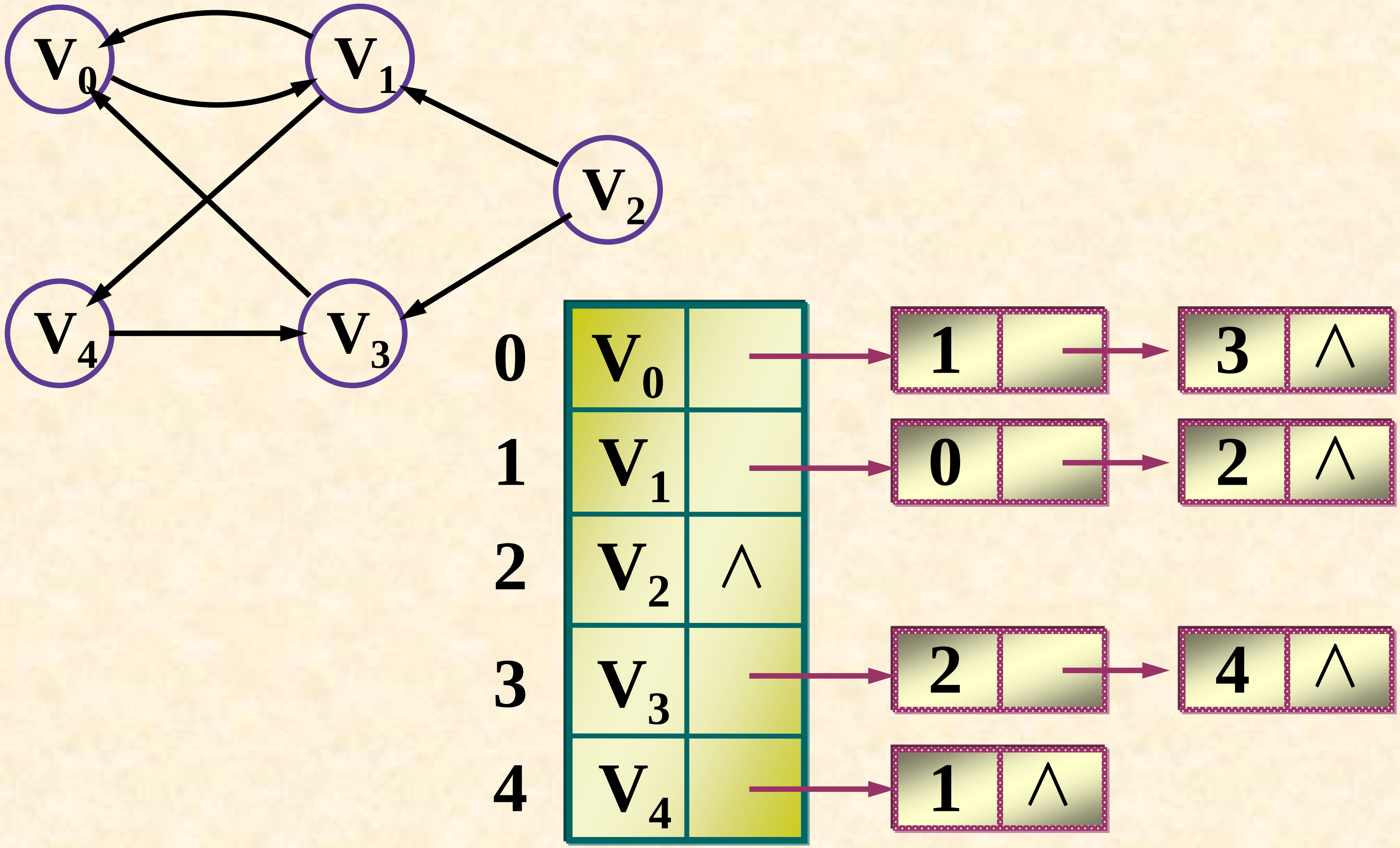




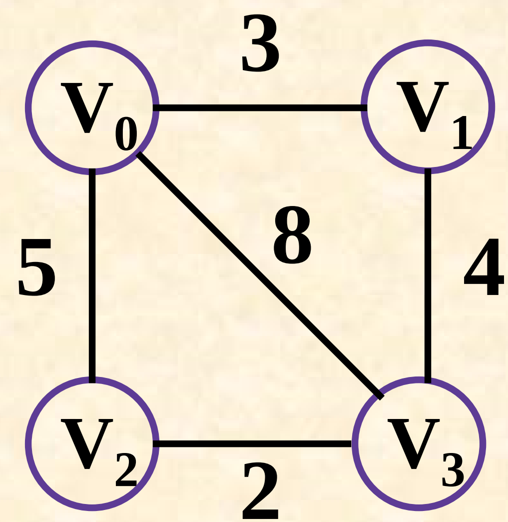
- 顶点V_i的出度为第i个单链表中的结点个数
- 顶点V_i的入度为整个单链表中邻接点域值是i的结点个数
- 逆邻接表：有向图中对每个结点建立以V_i为头的弧的单链表

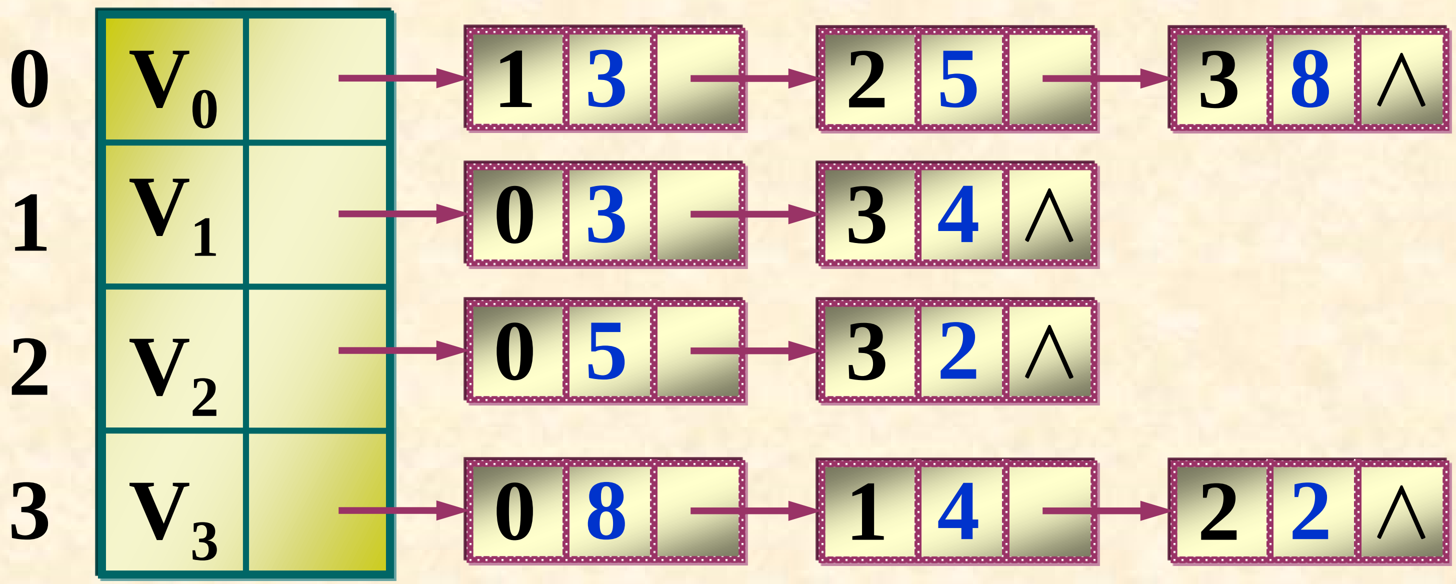
- 对于用邻接表存储的有向图，每条边只对应一个边结点；而对于用邻接表存储的无向图，每条边则对应两个边结点。
- 根据**邻接表**，可以统计出有向图中每个顶点的**出度**。但是，如果要统计顶点的入度，每统计一个顶点，就要遍历所有的边结点，其时间复杂度为**O(*e*)(*e***为图中边的个数)，从而统计所有顶点入度的时间复杂度为**O(*ne*)(*n***为图的顶点个数)。
- 建立**逆邻接表**(顶点的指向关系与邻接表恰好相反)，根据逆邻接表，很容易统计出图中每个顶点的**入度**。

[例3]有向图的逆邻接表



[例4] 权图的邻接表





邻接表数据结构定义

- 实现：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于顶点V_i的边（有向图中指以V_i为尾的弧）

```
#define MAX_VERTEX_NUM 20
typedef   struct   ArcNode {
    int  adjvex;        //邻接点域，存放与Vi邻接的点在表头数组中的位置
    struct  ArcNode  *nextarc;   //链域，指示下一条边或弧
    InfoType *info;
}ArcNode;
typedef   struct  VNode {   //表头接点
    VertexType  data;       //存放顶点信息
    ArcNode   *firstarc;    //指示第一个邻接点
}VNode, AdjList[MAX_VERTEX_NUM];
typedef struct {
    AdjList vertices;
    int vexnum, arcnum;
    int kind;     //图的种类标识
} ALGraph;
```

adjvex	nextarc	info

vexdata	firstarc

- 采用**邻接矩阵**还是用**邻接表**来存储图，要视对给定图实施的具体操作而定。
- 对于边很多的图(也称**稠密图**)，适于用**邻接矩阵**存储，因为占用的空间少。
- 而对于顶点多而边少的图(也称**稀疏图**)，若用邻接矩阵存储，对应的邻接矩阵将是一个稀疏矩阵，存储利用率很低。因此，顶点多而边少的图适于用**邻接表**存储。

有向图的十字链表表示法

```
#define MAX_VERTEX_NUM 20

typedef  struct  ArcBox {     //弧结点
    int  tailvex, headvex;        //弧尾、弧头在表头数组中位置
    struct ArcBox  *hlink, *tlink;  //分别指向弧头、狐尾相同的下一条弧
    InfoType *info;
}ArcBox;
```

tailvex	headvex	hlink	tlink	info

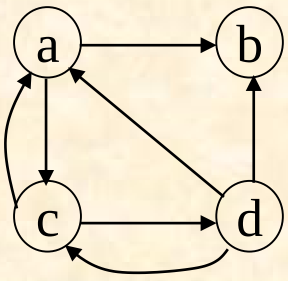
```
typedef  struct  VexNode {     //顶点结点
    VertexType data;  //存与顶点有关信息
    ArcBox *firstin, *firstout;  //分别指向该顶点第一条入弧和出弧
}VexNode;
```

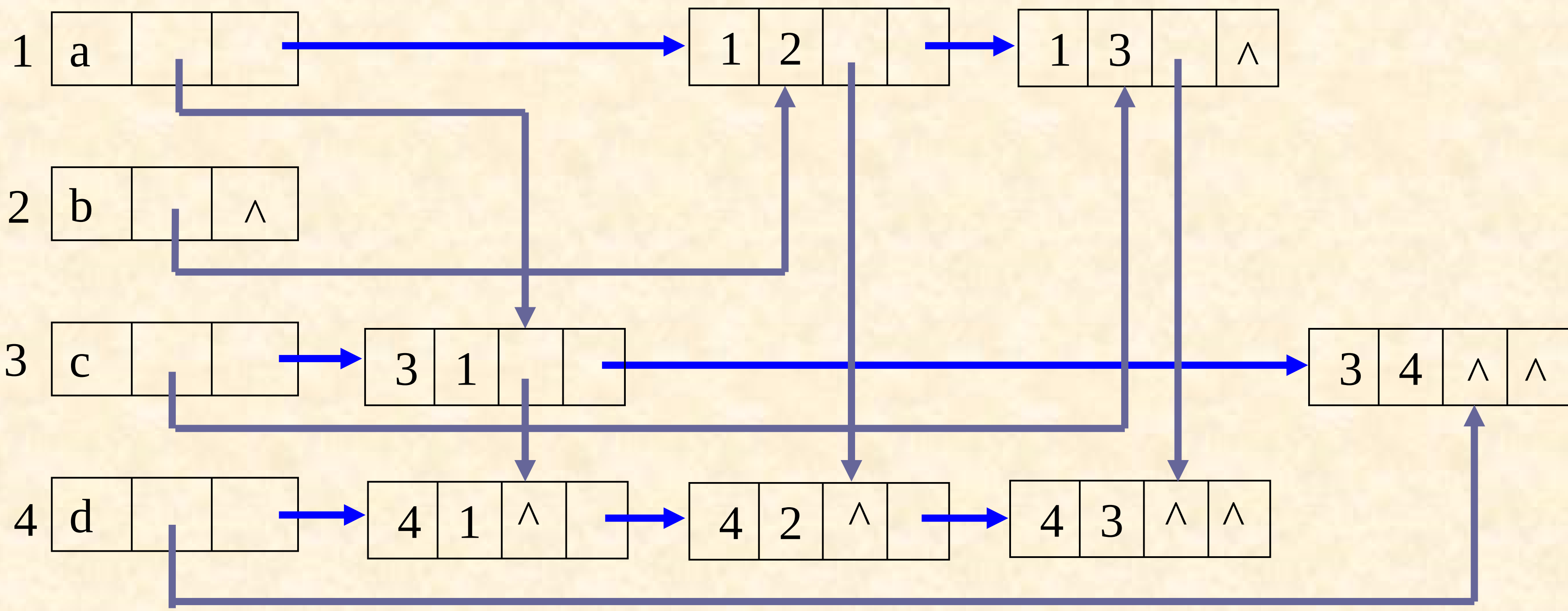
data	firstin	firstout

```
typedef struct {
    VexNode xlist[MAX_VERTEX_NUM];
    int vexnum, arcnum;
} OLGraph;
```

有向图的十字链表表示法

例





有向图的十字链表表示法

```
#define MAX_VERTEX_NUM 20

typedef  struct  ArcBox {     //弧结点
    int  tailvex, headvex;      //弧尾、弧头在表头数组中位置
    struct ArcBox  *hlink, *tlink; //分别指向弧头、狐尾相同的下一条弧
    InfoType *info;
}ArcBox;
```

tailvex	headvex	hlink	tlink	info

```
typedef  struct  VexNode {     //顶点结点
    VertexType data;  //存与顶点有关信息
    ArcBox *firstin, *firstout; //分别指向该顶点第一条入弧和出弧
}VexNode;
```

data	firstin	firstout

```
typedef struct {
    VexNode xlist[MAX_VERTEX_NUM];
    int vexnum, arcnum;
} OLGraph;
```

无向图的邻接多重表表示法

```
#define MAX_VERTEX_NUM 20
#define enum {unvisited, visited} VisitIf;
typedef   struct  EBox {        //边结点
    VisitIf  mark;     //标志域
    int  ivex, jvex;   //该边依附的两个顶点在表头数组中位置
    struct  EBox   *ilink,  *jlink;  //分别指向依附于ivex和jvex的下一条边
    InfoType *info;
}EBox;
```

mark	ivex	ilink	jvex	jlink

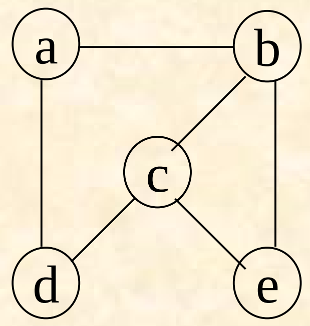
```
typedef   struct  VexNode {    //顶点结点
    VertexType data;   //存与顶点有关的信息
    EBox  *firstedge;  //指向第一条依附于该顶点的边
}VexBox;
```

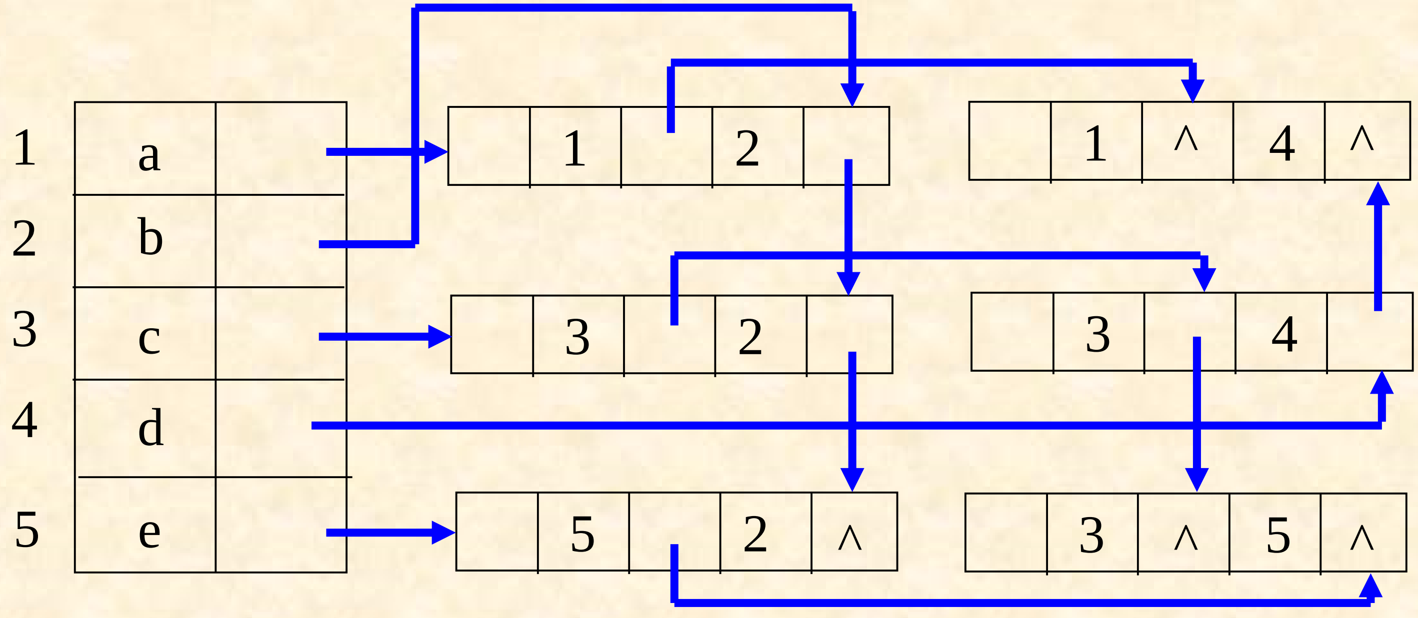
data	firstedge

```
typedef struct {
    VexBox adjmulist[MAX_VERTEX_NUM];
    int vexnum, edgenum;
} AMLGraph;
```

无向图的邻接多重表表示法

例





无向图的邻接多重表表示法

```
#define MAX_VERTEX_NUM 20
#define enum {unvisited, visited} VisitIf;
typedef   struct  EBox {         //边结点
    VisitIf  mark;      //标志域
    int  ivex, jvex;   //该边依附的两个顶点在表头数组中位置
    struct  EBox   *ilink,  *jlink;  //分别指向依附于ivex和jvex的下一条边
    InfoType *info;
}EBox;
```

mark	ivex	ilink	jvex	jlink

```
typedef   struct  VexNode {    //顶点结点
    VertexType data;   //存与顶点有关的信息
    EBox  *firstedge;  //指向第一条依附于该顶点的边
}VexBox;
```

data	firstedge

```
typedef struct {
    VexBox adjmulist[MAX_VERTEX_NUM];
    int vexnum, edgenum;
} AMLGraph;
```

第七章 图

◆从已给的连通图中**某一顶点出发**，沿着一些边**访遍图中所有顶点**，且使每个顶点仅**被访问一次**，就叫做**图的遍历 (Graph Traversal)**。

◆**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。

◆**为了避免重复访问**，可设置一个**标志**顶点是否被访问过的辅助数组 *visited*[ ]，它的初始状态为 0，在图的遍历过程中，一旦某一个顶点 *i* 被访问，就立即让 *visited*[*i*] 为 1，**防止它被多次访问**。

7.3.1 深度优先遍历

- 深度优先遍历又被称为**深度优先搜索** ***DFS*** **(Depth First Search)**

- **基本思想（递归定义）：**

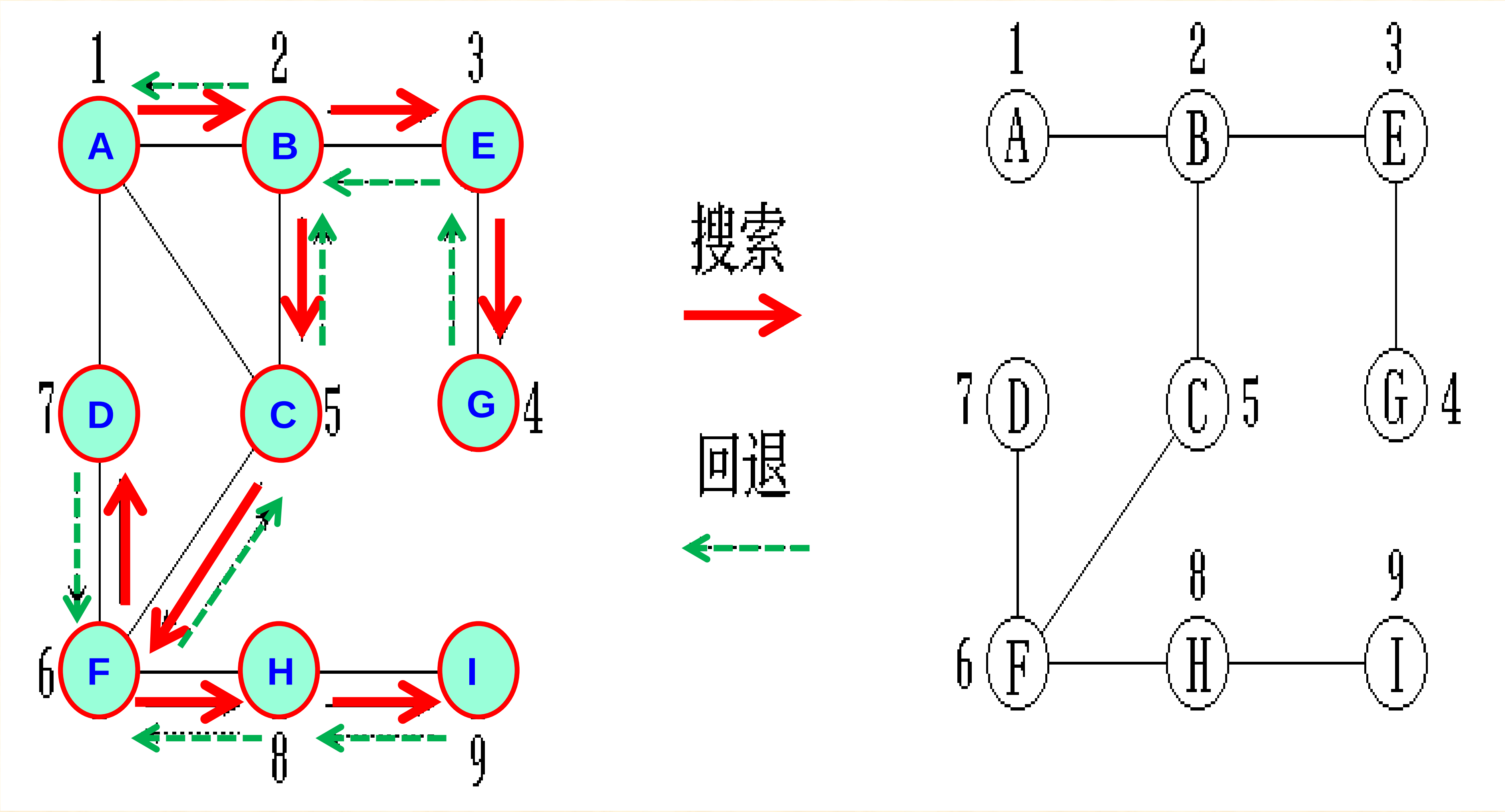
DFS从图的某一顶点**V0**出发，访问此顶点；然后依次从**V0**的未被访问的邻接点出发，**深度优先遍历图**，直至图中所有和**V0**相通的顶点都被访问到；若此时图中尚有顶点未被访问，**则另选图中一个未被访问的顶点作起点**，重复上述过程，直至图中所有顶点都被访问为止。

- **基本思想（非递归定义）：**

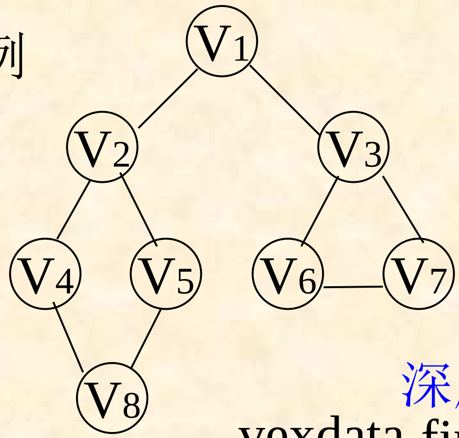
DFS 在访问图中某一起始顶点 v 后，**由 v 出发**，访问它的任一邻接顶点 w_1；**再从w_1出发**，访问与 w_1邻接但还没有访问过的顶点 w_2；**然后再从w_2出发**，进行类似的访问，… 如此进行下去，直至到达所有的邻接顶点都被访问过的顶点 u 为止。**接着，退回一步，退到前一次刚访问过的顶点**，看是否还有其它没有被访问的邻接顶点。如果有，则访问此顶点，之后再从此顶点出发，进行与前述类似的访问；**如果没有，就再退回一步进行搜索**。重复上述过程，直到连通图中所有顶点都被访问过为止。

深度优先搜索*DFS* (Depth First Search)

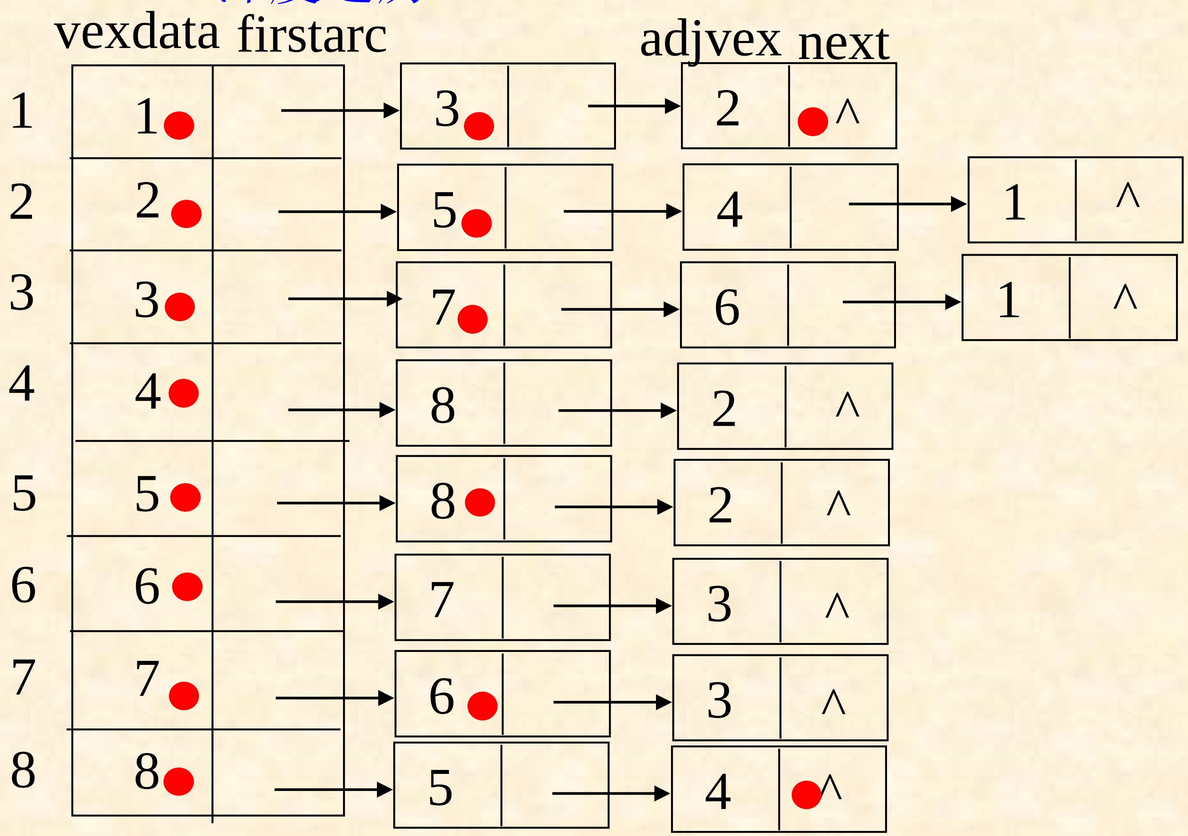
- 深度优先搜索的示例



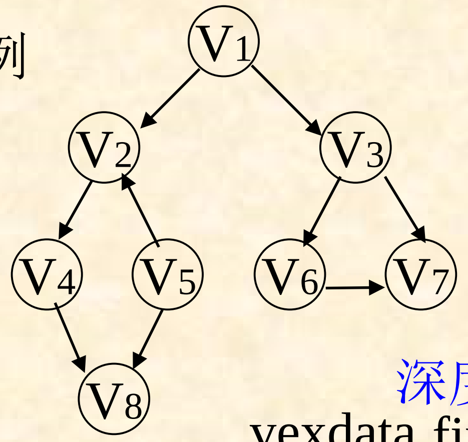
例



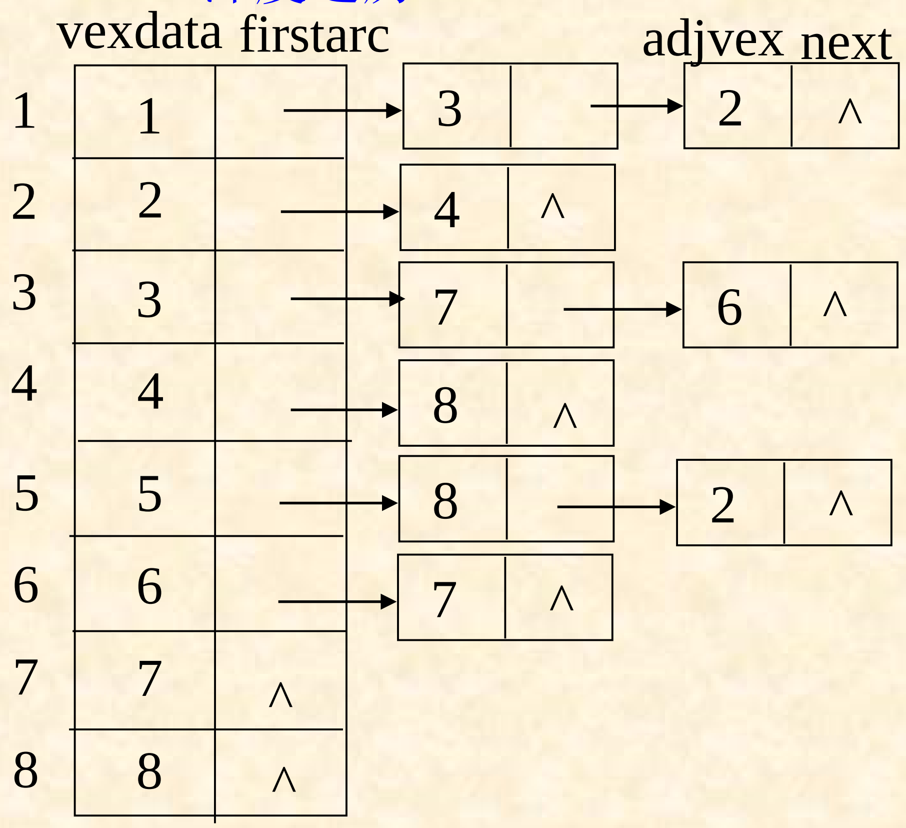
深度遍历：V1⇒V3 ⇒V7 ⇒V6 ⇒V2 ⇒V5 ⇒V8 ⇒V4



例



深度遍历：V1⇒V3 ⇒V7 ⇒V6 ⇒V2 ⇒V4 ⇒V8 ⇒V5



1. 递归算法

```
bool visited[MAX_VERTEX_NUM];  // 访问标志数组
Status (* VisitFunc)(int v);    // 函数变量
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
    int w;
    visited[v] = true;   VisitFunc(v);  // 访问第v个顶点
    for (w=FirstAdjVex(G, v);  w!=-1;  w=NextAdjVex(G, v, w))
        if (!visited[w])   // 对v的尚未访问的邻接顶点w递归调用DFS
            DFS(G, w);
}
void DFSTraverse(Graph G, Status (*Visit)(int v)) { // 对图G作深度优先遍历
    int v;
    VisitFunc = Visit; // 使用全局变量VisitFunc，使DFS不必设函数指针参数
    for (v=0; v<G.vexnum; ++v) visited[v] = false; // 访问标志数组初始化
    for (v=0; v<G.vexnum; ++v)
        if (!visited[v]) DFS(G, v);        // 对尚未访问的顶点调用DFS
}
```

2. 迭代算法

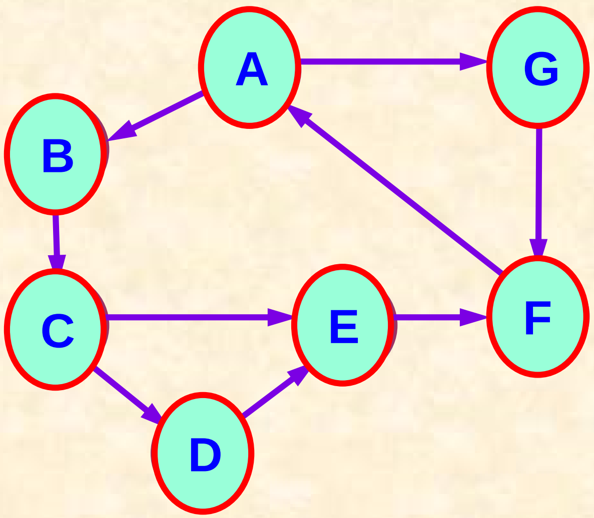
可以利用**堆栈**实现**深度**优先遍历的**非递归算法**。

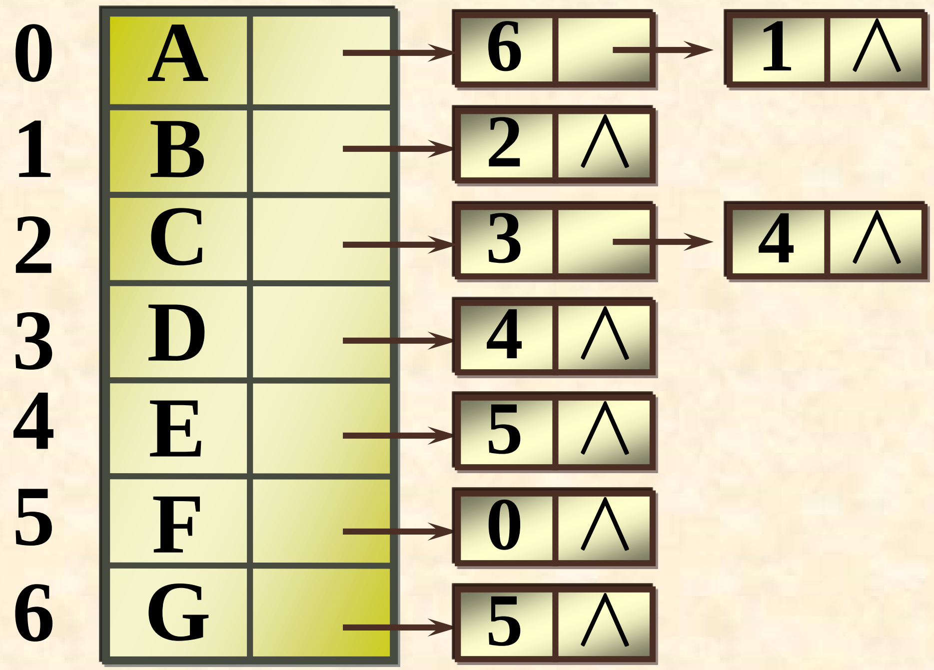
堆栈中存放已访问结点的未被访问的邻接顶点，每次弹出栈顶元素时，如其未被访问，则访问该顶点，并**检查当前顶点的边链表，将其未被访问的邻接顶点入栈**，循环进行。

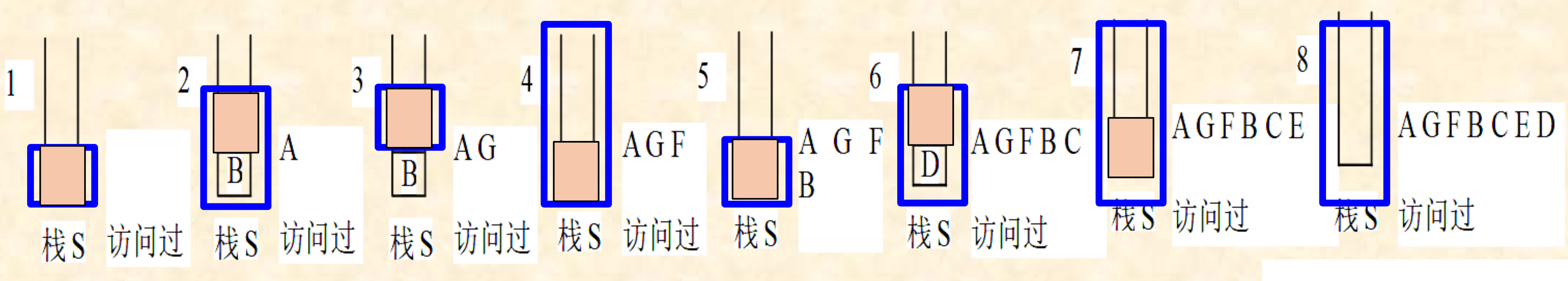
首先将所有顶点的visited[]值置为0, 初始顶点压入堆栈;

① 检测堆栈是否为空。若堆栈为空，则迭代结束；否则，从**栈顶弹出**一个顶点v;

② 如果v未被访问过，则访问v，将visited[v]值更新为1，然后根据v的邻接顶点表，将v的未被访问的**邻接顶点压入栈**，执行步骤①。


A
G
B
C
E
F
D


0 A 6 1 ∧
1 B 2 ∧
2 C 3 4 ∧
3 D 4 ∧
4 E 5 ∧
5 F 0 ∧
6 G 5 ∧


1
栈S 访问过
2
B
栈S
A
访问过
3
B
栈S
A G
访问过
4
栈S
A G F
访问过
5
栈S
A G F
B
6
D
栈S
A G F B C
访问过
7
栈S
A G F B C E
访问过
8
栈S
A G F B C E D
访问过

算法DFS (*Head*, *v* , *visited. visited*)

/* 图的深度优先遍历的非递归算法*/

DFS1[初始化]

CREATS(*S*) . /*创建堆栈 *S* */

FOR *i* = 1 TO *n* DO *visited*[*i*] ← 0.

***S*⇐ *v*. /* 将*v*压入栈中 */**

DFS2[利用堆栈S深度优先遍历图]

WHILE NOT(ISEMTS(*S*)) DO /* 当*S*不空时 */

(*v*⇐ *S*. /*弹出堆栈顶元素 */

IF *visited*[*v*] = 0 THEN

(PRINT(*v*) . *visited*[*v*] ← 1.

***p* ← adjacent(*Head*[*v*]) .**

WHILE *p*≠Λ DO

(IF *visited*[VerAdj(*p*)] = 0 THEN

***S* ⇐ VerAdj(*p*) .**

***p* ← link(*p*).))) ▌**

算法分析

- 图中有 n 个顶点，e 条边。
- 如果用邻接表表示图，沿顶点的**adjacent**可以找到某个顶点v 的所有邻接顶点w。由于总共有$2e$个边结点，所以扫描边的时间为$O(e)$。而且对所有顶点递归访问1次，所以遍历图的时间复杂性为$O(n+e)$。
- 如果用邻接矩阵表示图，则查找每一个顶点的所有的边，所需时间为$O(n)$，则遍历图中所有的顶点所需的时间为$O(n^2)$。

非连通图需要多次调用深度优先遍历算法

```
For i=0 to n-1 DO
    visited[i] ← 0.
For j=0 to n-1 DO
    IF visited[j]=0 THEN
        DepthFirstSearch (v[j], visited)
```

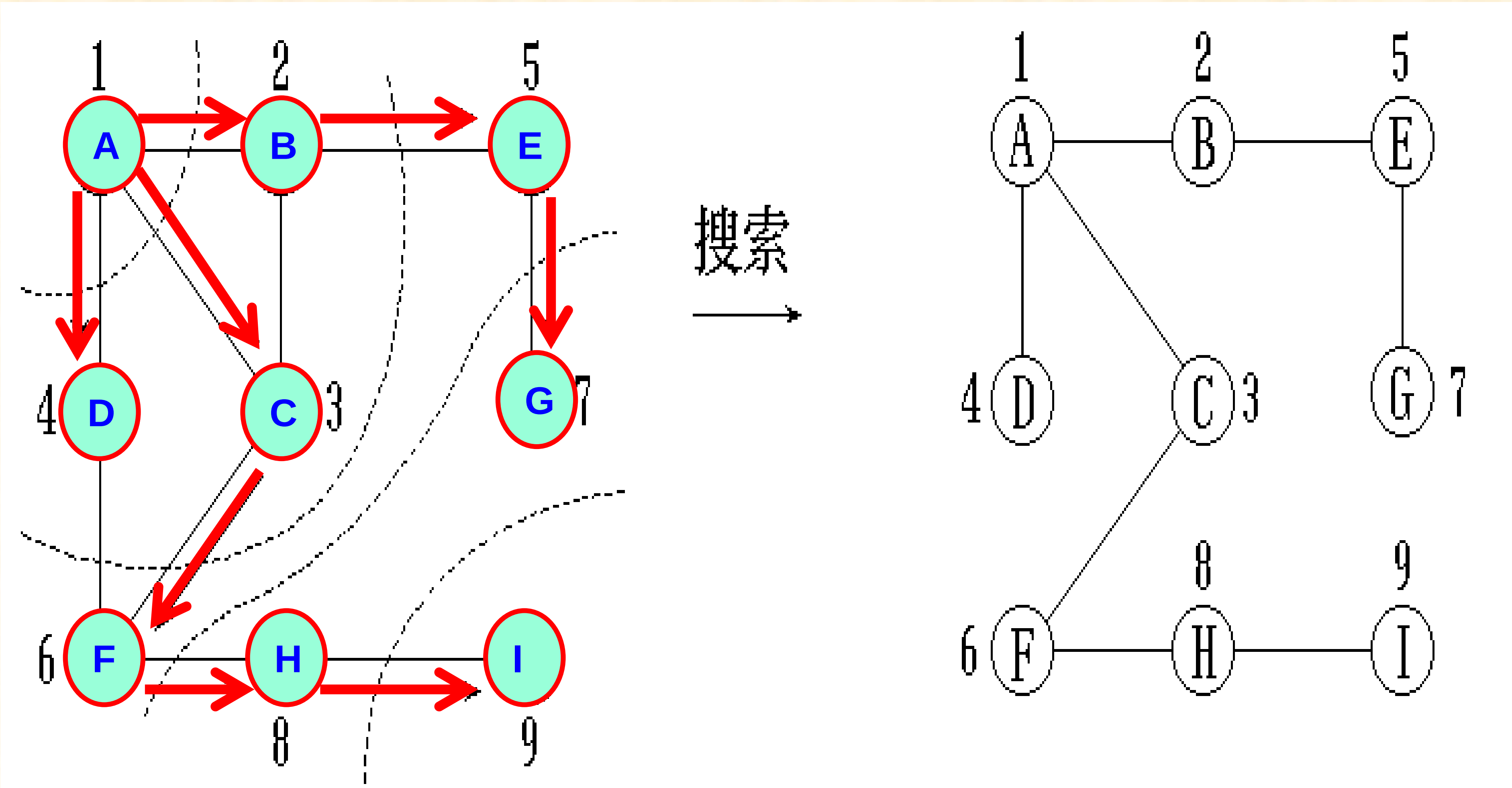
7.3.2 广度优先遍历*BFS*

● 基本思想:

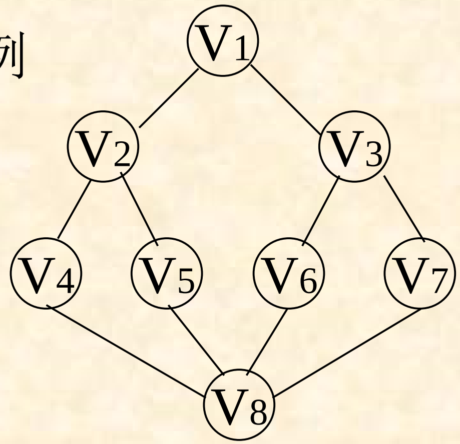
*BFS*首先**访问初始点顶点**v_0，之后**依次访问**与v_0**邻接的全部顶点**w_1，w_2，…，w_k。然后，**再顺次访问**与w_1，w_2，…，w_k**邻接的尚未访问的全部顶点**，再从这些被访问过的顶点出发，逐个访问与它们邻接的尚未访问过的全部顶点。依此类推，直到连通图中的所有顶点全部访问完为止。

广度优先搜索*BFS* (Breadth First Search)

- 广度优先搜索的示例

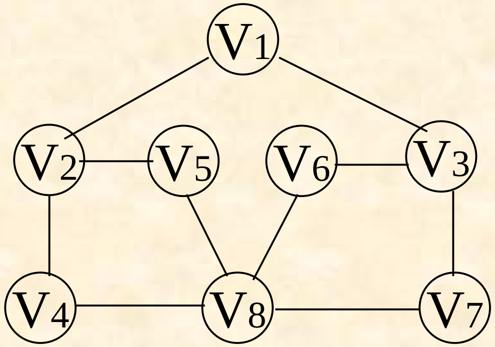


例



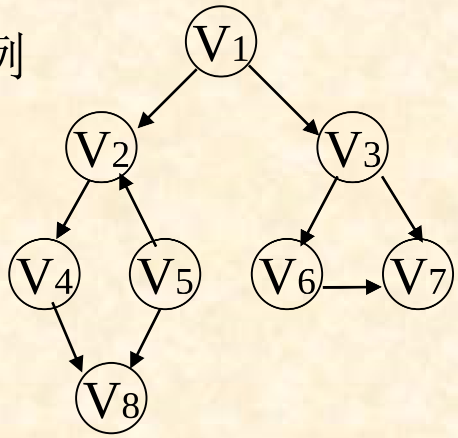
广度遍历：V1⇒ V2 ⇒V3 ⇒ V4 ⇒V5 ⇒V6 ⇒V7 ⇒V8

例



广度遍历：V1⇒ V2 ⇒V3 ⇒ V4 ⇒V5 ⇒V6 ⇒V7 ⇒V8

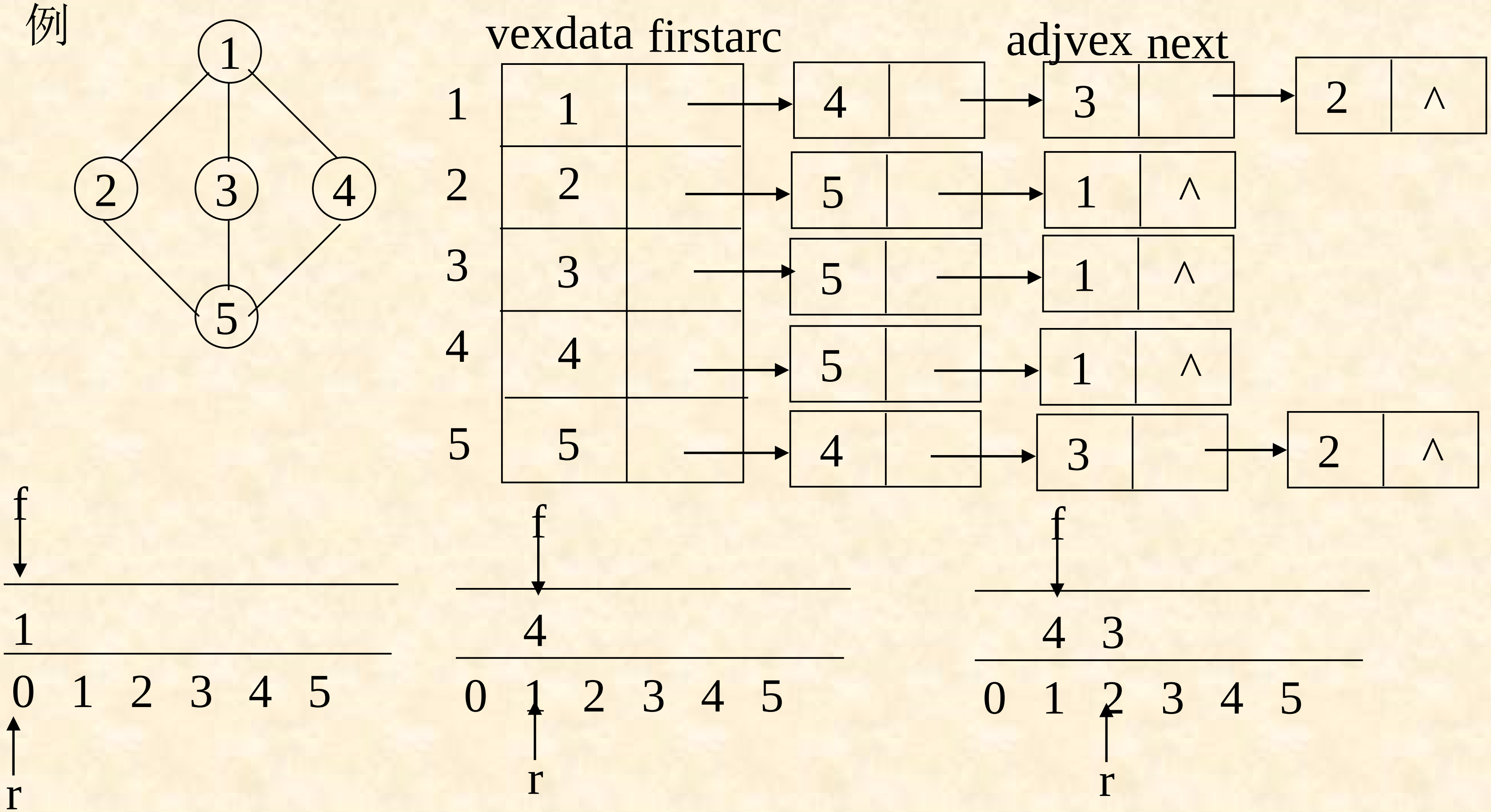
例



广度遍历：V1⇒ V2 ⇒V3 ⇒ V4 ⇒V6 ⇒V7 ⇒V8 ⇒V5

- 广度优先搜索**类似于树的层次遍历**，是一种分层的搜索过程，每向前走一步可能访问一批顶点，不像深度优先搜索那样有回退的情况。因此，**广度优先搜索不是一个递归的过程**，其算法也不是递归的。
- 为了实现**逐层访问**，算法中使用一个**队列**，以便于向下一层访问。
- 与深度优先搜索过程一样，为**避免重复**访问，需要一个**辅助数组visited[ ]**。

例

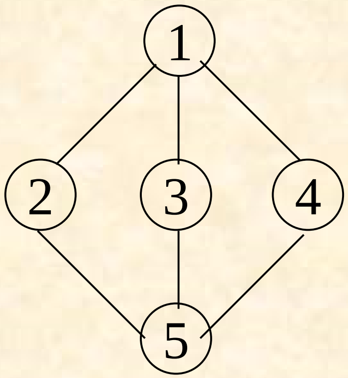


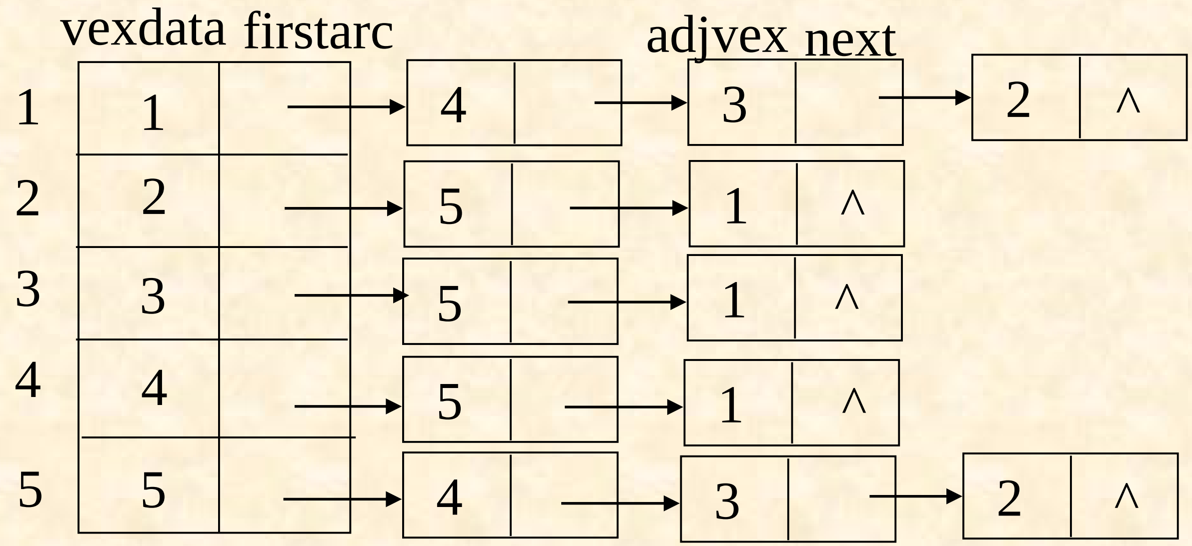
遍历序列：1

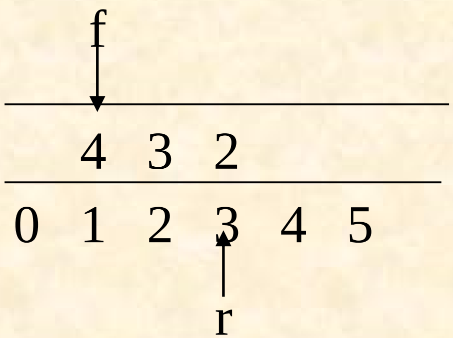
遍历序列：1 4

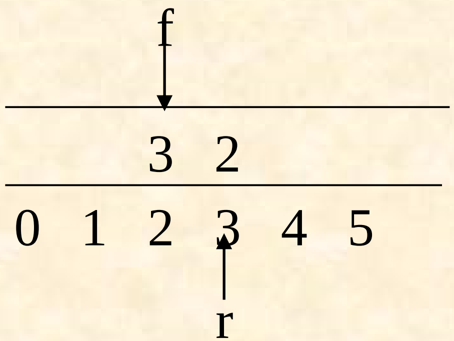
遍历序列：1 4 3

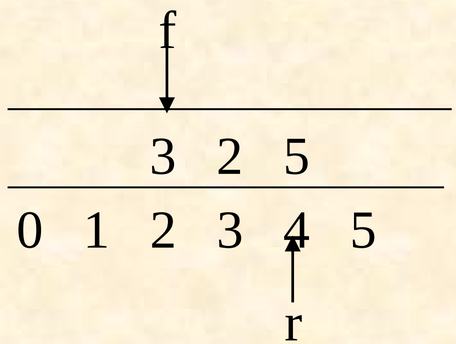
例









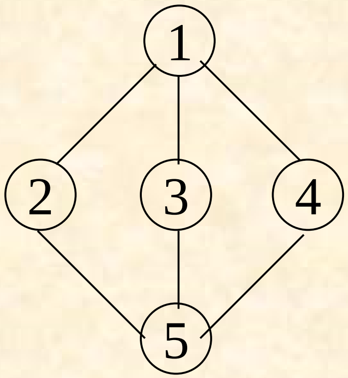


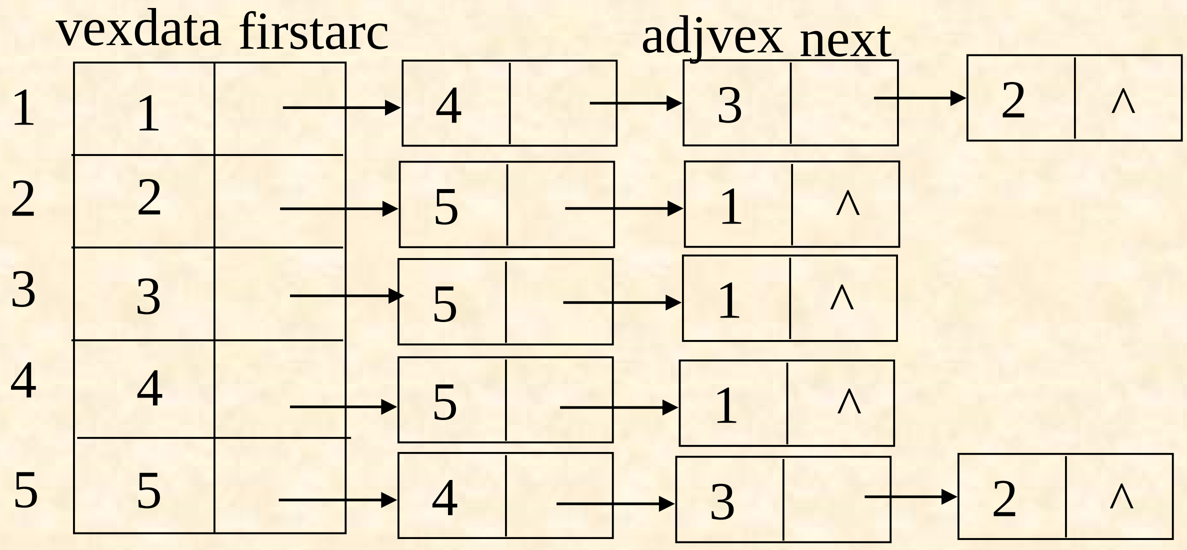
遍历序列：1 4 3 2

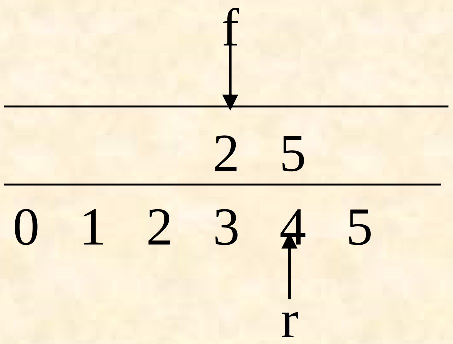
遍历序列：1 4 3 2

遍历序列：1 4 3 2 5

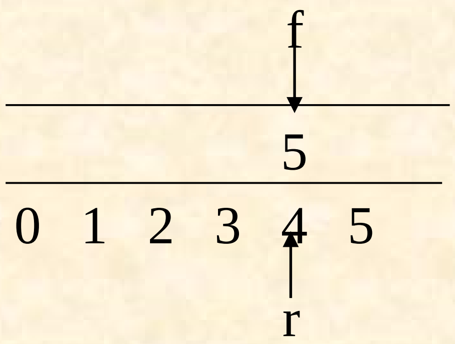
例



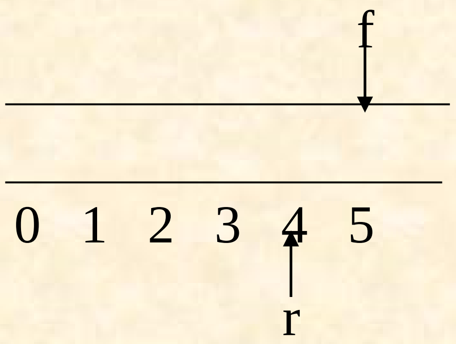




遍历序列：1 4 3 2 5



遍历序列：1 4 3 2 5



遍历序列：1 4 3 2 5

```
void BFSTraverse(Graph G, Status (*Visit)(int v )) {// 按广度优先非递归遍历
图G。使用辅助队列Q和访问标志数组visited。
    QElemType v,w;
    queue Q;
    QElemType u;
    for (v=0; v<G.vexnum; ++v) visited[v] = FALSE;
    InitQueue(Q);                    // 置空的辅助队列Q
    for (v=0;  v<G.vexnum;  ++v)
        if (!visited[v]) {           // v尚未访问
            visited[v] = TRUE;  Visit(v); // 访问v
            EnQueue(Q, v);               // v入队列
            while (!QueueEmpty(Q)) {
                DeQueue(Q, u);           // 队头元素出队并置为u
                for (w=FirstAdjVex(G, u);  w>=0;  w=NextAdjVex(G, u, w))
                    if (!visited[w]) {       // u的尚未访问的邻接顶点w入队列Q
                        visited[w] = TRUE;  Visit(w);
                        EnQueue(Q, w);
                    }//if
            }//while
        }//if
} // BFSTraverse
```

算法分析

- 如果使用邻接表表示图，则循环的总时间代价为 $d_0 + d_1 + \ldots + d_{n-1} = O(e)$，其中的 d_i 是顶点 i 的度。总的时间复杂度为$O(n+e)$。
- 如果使用邻接矩阵，则对于每一个被访问的顶点，循环要检测矩阵中的 n 个元素，总的时间代价为$O(n^2)$。

- 图的深度优先—树的先根遍历—回溯法（试探法）
- 图的广度优先—树的层次遍历—分支限界法

进行问题搜索时，哪种方法更好？

深度优先

优点：存储空间少；

缺点：会面临**“钻牛角尖”**的问题，有时找不到解；

宽度优先

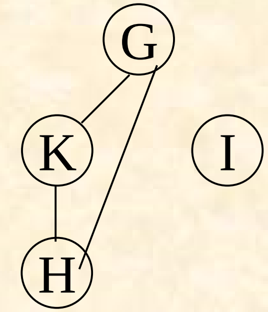
优点：只要存在解，则一定能找到；

缺点：经常会面临**组合爆炸**问题。

第七章 图

图的生成树

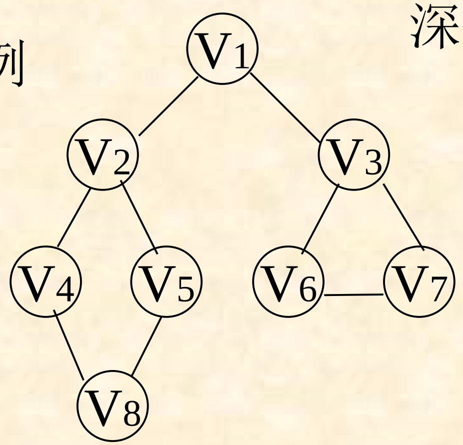
- ❑ 所有顶点均由边连接在一起，但不存在回路的图**深度优先生成树**与**广度优先生成树**
- ❑ 生成森林：非连通图每个连通分量的生成树一起组成非连通图的~
- ✧ 说明　　**一个图可以有许多棵不同的生成树**
  - ■ 所有生成树具有以下共同特点：
    - ✧生成树的顶点个数与图的顶点个数**相同**
    - ✧生成树是图的**极小连通子图**
    - ✧一个有n个顶点的连通图的生成树有**n-1条边**
    - ✧生成树中任意两个顶点间的路径是**唯一的**
    - ✧在生成树中再加一条边必然形成**回路**
  - ■ 含n个顶点n-1条边的图不一定是生成树

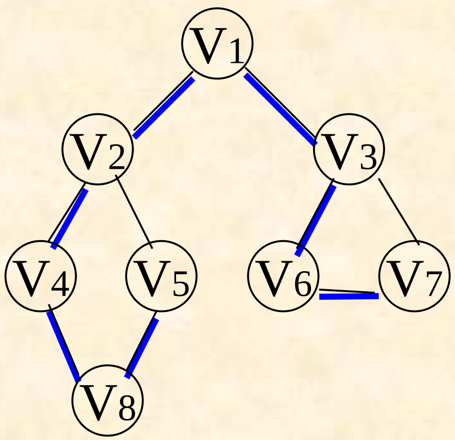


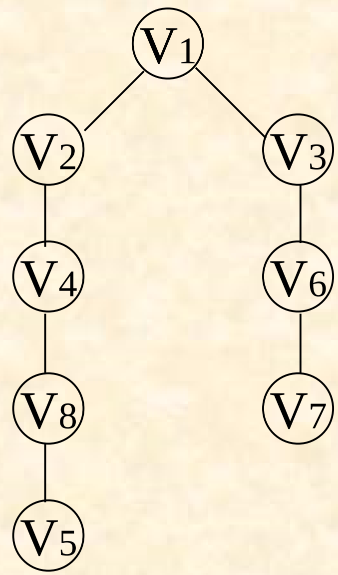
广度遍历：V1⇒ V2 ⇒V3 ⇒ V4 ⇒V5 ⇒V6 ⇒V7 ⇒V8

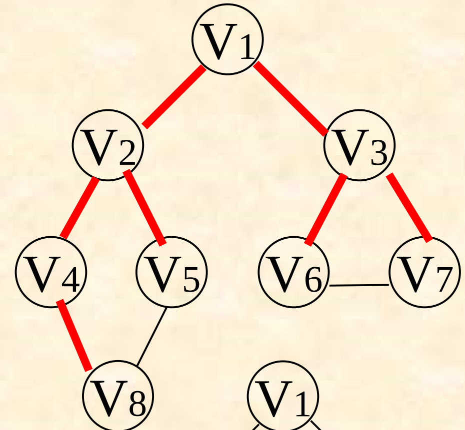
深度遍历：V1⇒ V2 ⇒V4 ⇒ V8 ⇒V5 ⇒V3 ⇒V6 ⇒V7

例

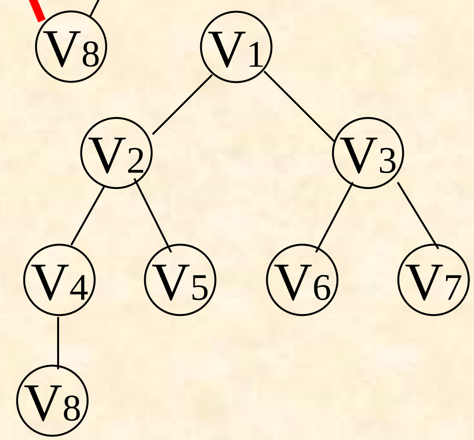






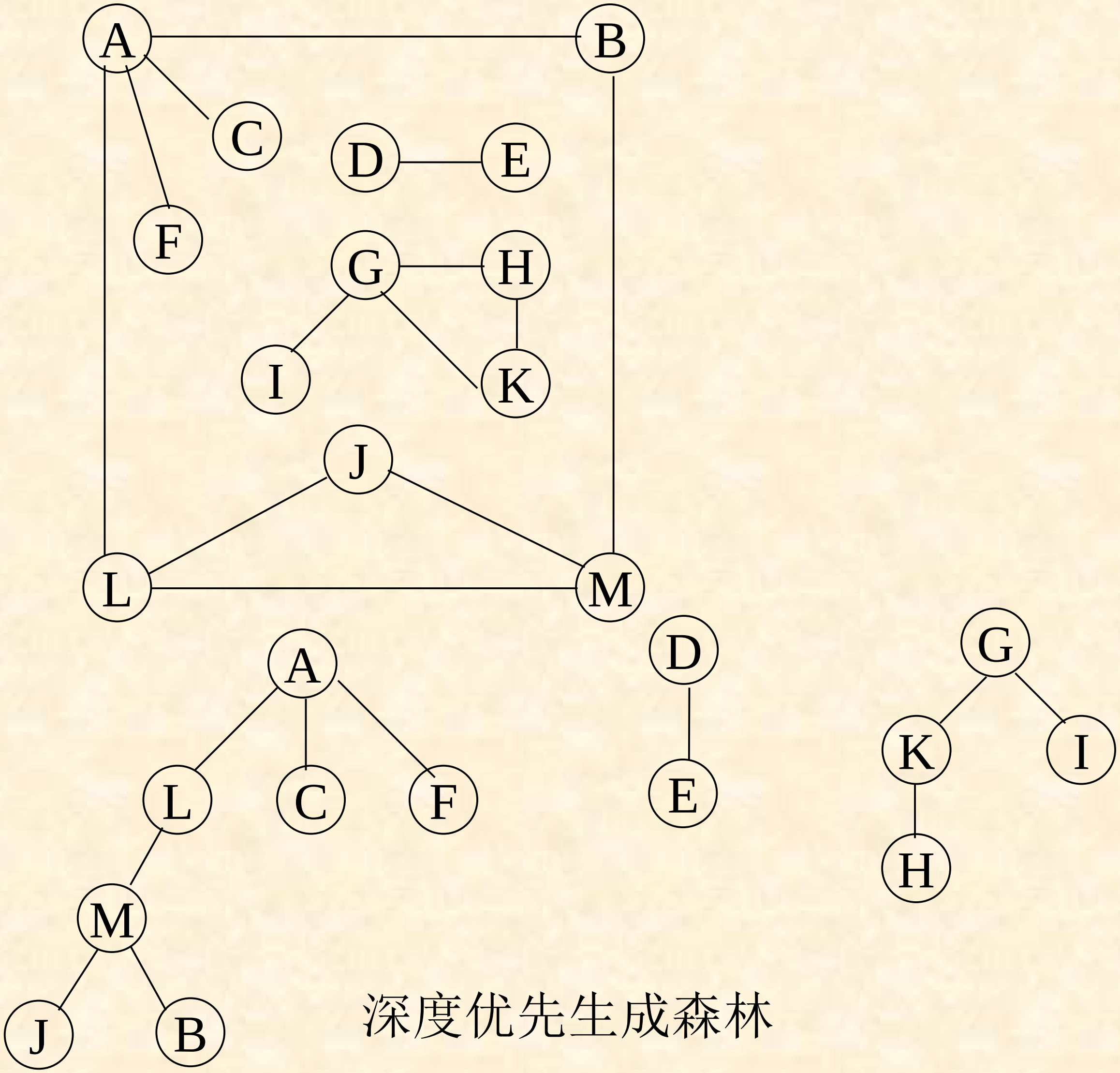


深度优先生成树



广度优先生成树

例



深度优先生成森林

最小生成树——基本概念

对于一个无向网络——无向加权连通图N=(V,E,C)(C表示该图为权图)，其顶点个数为|V|=n，图中边的个数为|E|，我们可以从它的|E|条边中选出n-1条边，使之满足

（1）这n-1条边和图的n个顶点构成一个连通图。

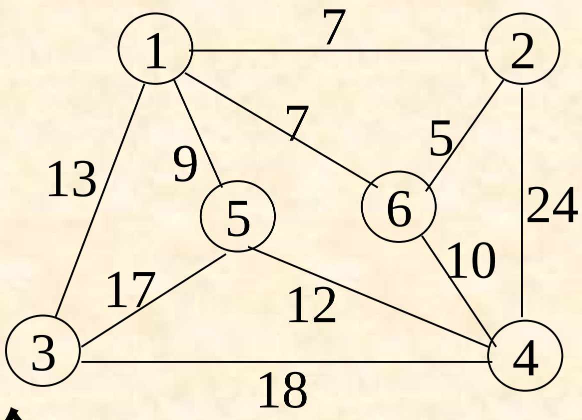
（2）该连通图的代价是所有满足条件(1)的连通图的代价的最小值。

这样的连通图被称为网络的**最小生成树**(Minimum-cost Spanning Tree)。

最小支撑树的性质

- 最小支撑树中没有回路
  - ❖ 若**MST** 的边集中有回路，显然可通过去掉回路中某条边而得到花销更小的**MST**
- 最小支撑树是一棵有**|V| - 1**条边的树
- 最小支撑树
  - ❖ 满足最小支撑树要求的边集所构成的树支撑起了所有的顶点(即把它们联接起来了)
  - ❖ 此边集的代价最小

最小生成树应用



❑ 问题提出

要在n个城市间建立通信联络网，

顶点——表示城市

权——城市间建立通信线路所需花费代价

希望找到一棵生成树，它的每条边上的权值之和（即建立该通信网所需花费的总代价）最小——最小代价生成树。

❑ 问题分析

n个城市间，最多可设置n(n-1)/2条线路

n个城市间建立通信网，只需n-1条线路

问题转化为：如何在可能的线路中选择n-1条，能把所有城市（顶点）均连起来，且总耗费（各边权值之和）最小。

最小生成树——普里姆(Prim)算法

1、普里姆(Prim)算法(逐点加入)

设N=(V,E,C)为连通网，TE是N的最小支撑树MST的**边的集合**，U为MST**顶点集**。

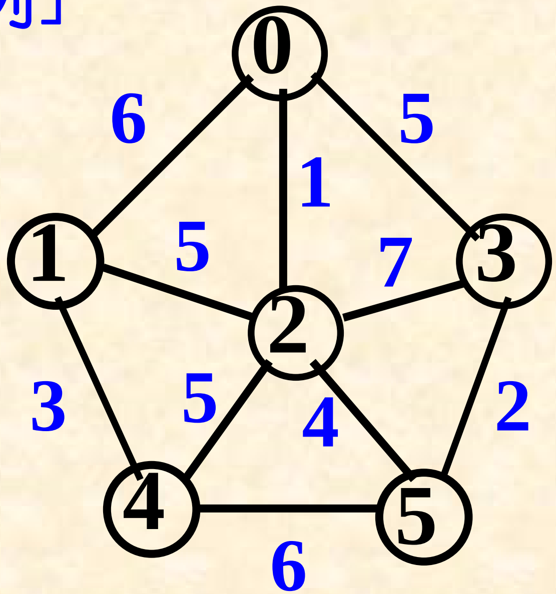
① 初始设$U=\{u_o\}(u_o\in V)$, $TE=\Phi$；

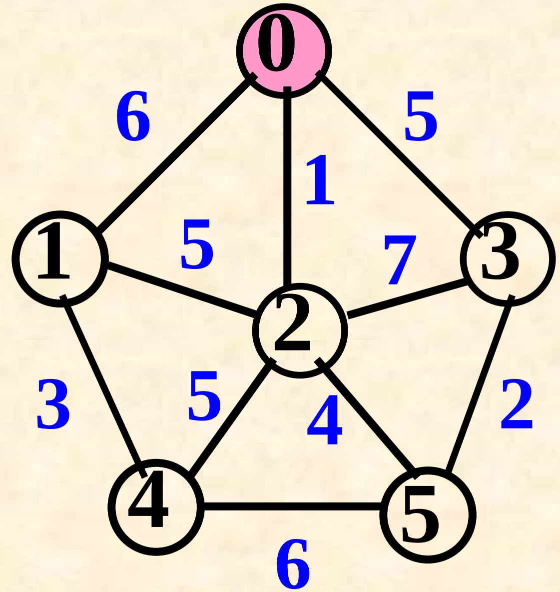
② 找到满足

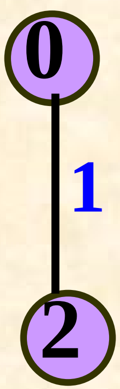
$weight(u,v)=min\{weight(u_1,v_1)|u_1\in U, v_1\in V-U\}$，的边，把它**并入**TE，同时v**并入**U;

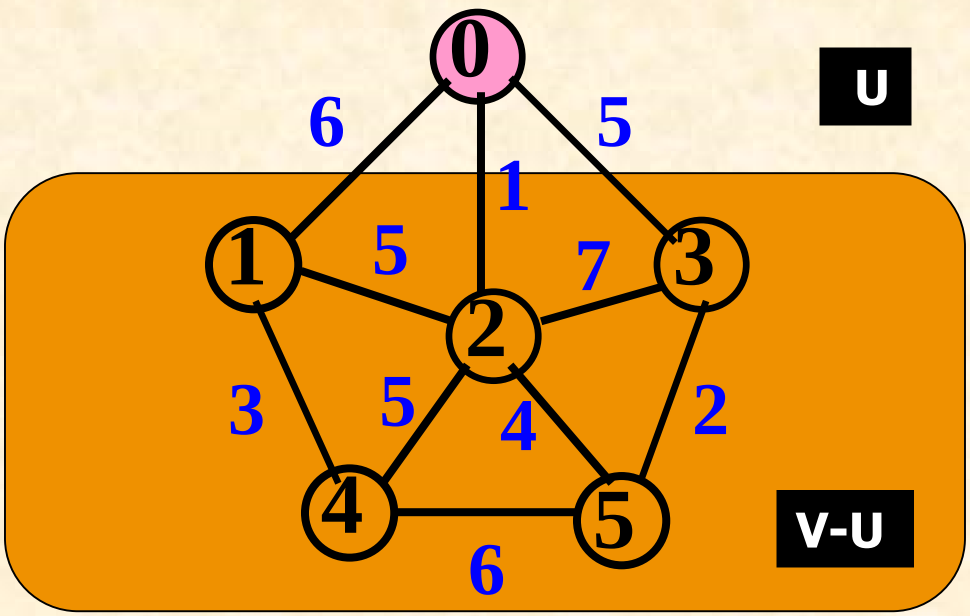
③ 反复执行② ,**直至** V=U，则T=(V,{TE})为N的**最小生成树**,算法结束。

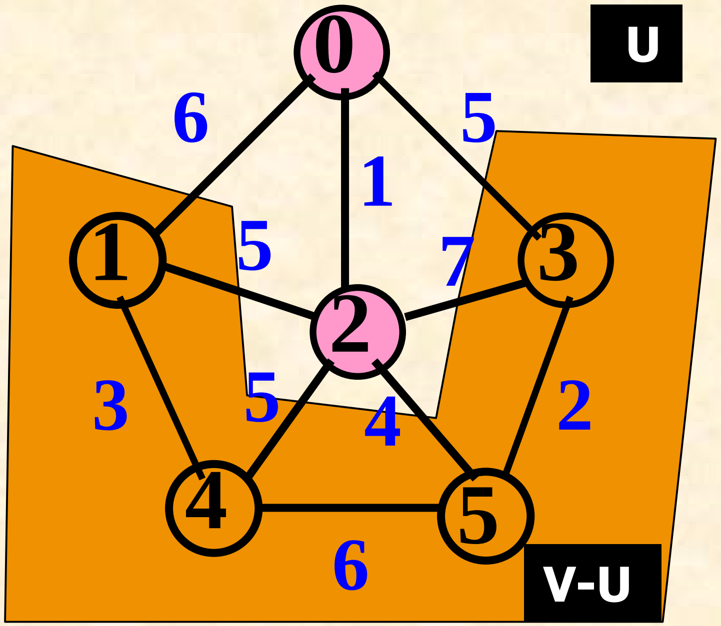
[例]

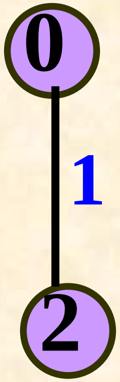


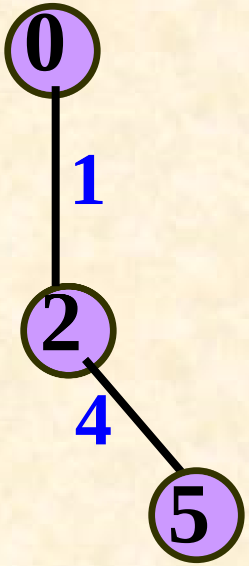


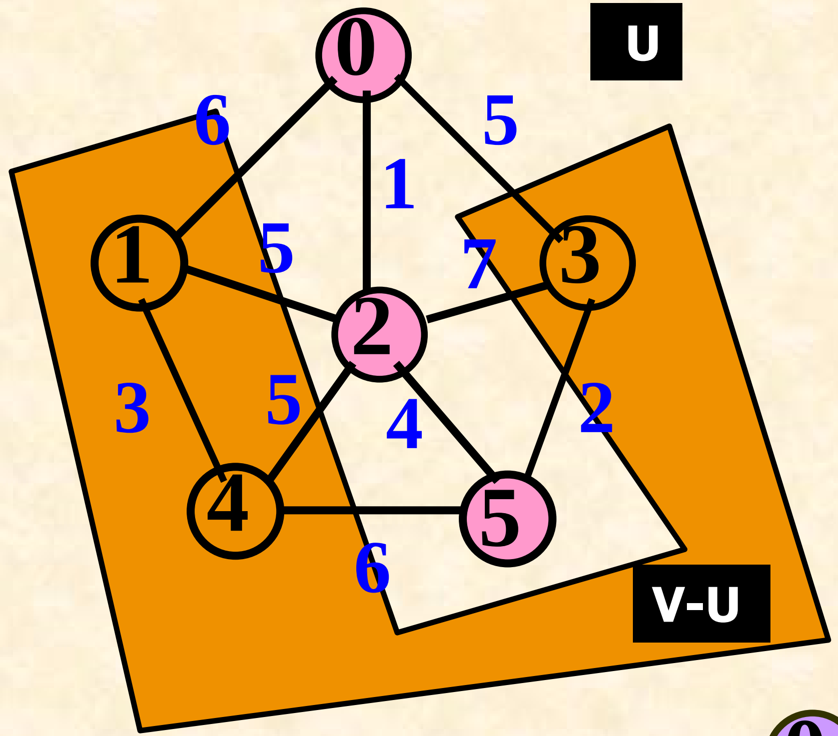


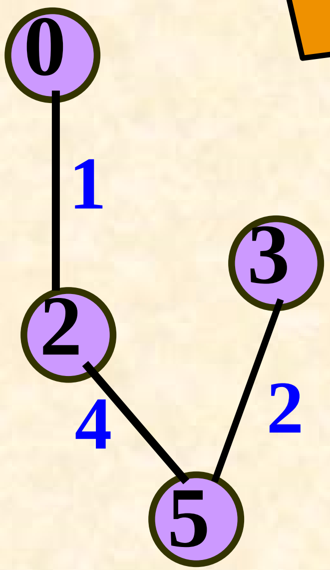


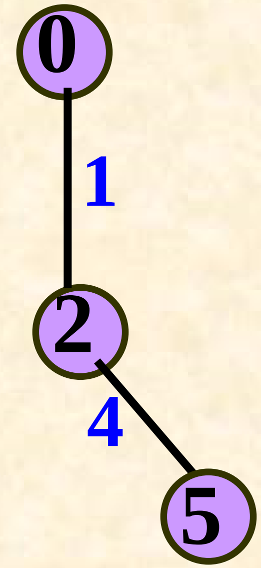

U
0
1
2
3
4
5
6
5
1
5
7
3
5
4
2
6
V-U

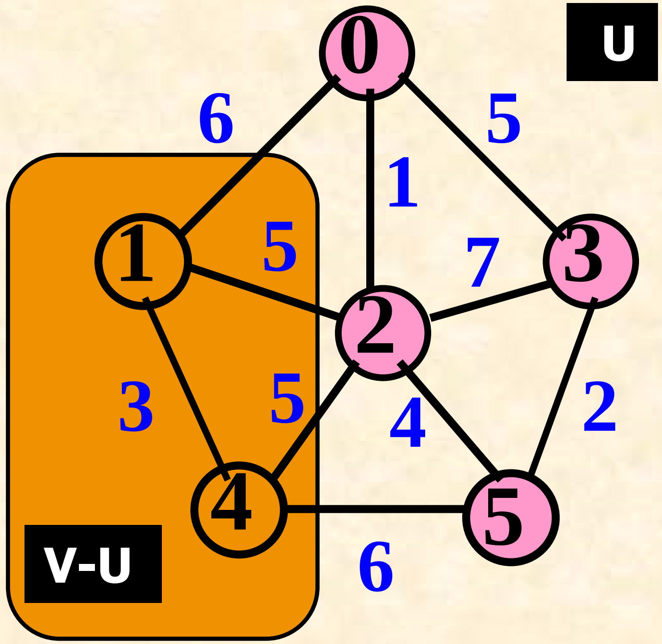

0
1
2

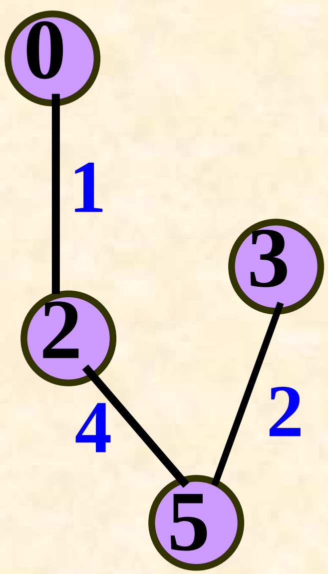

0
1
2
4
5

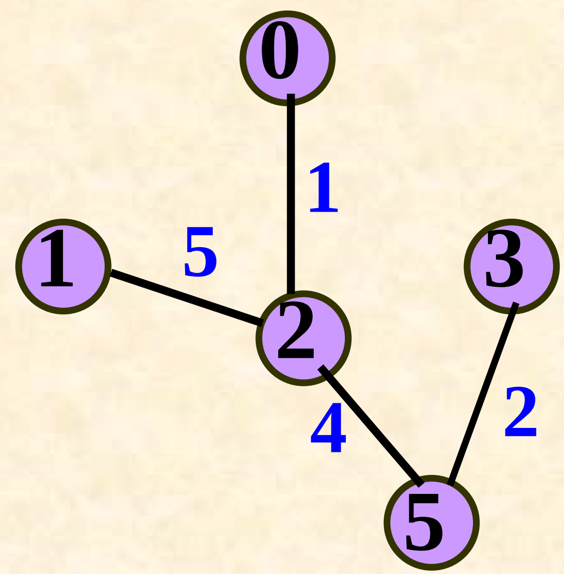

U
V-U

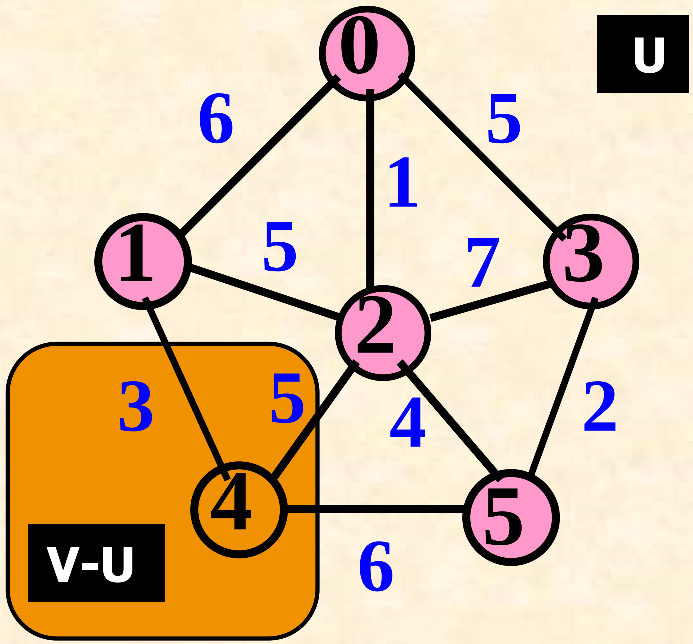


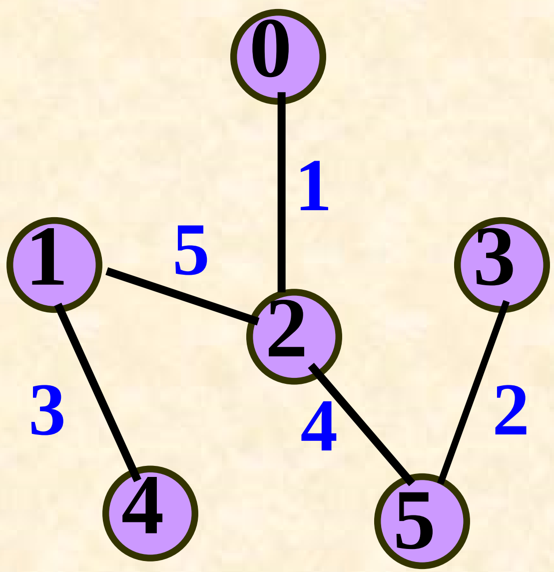


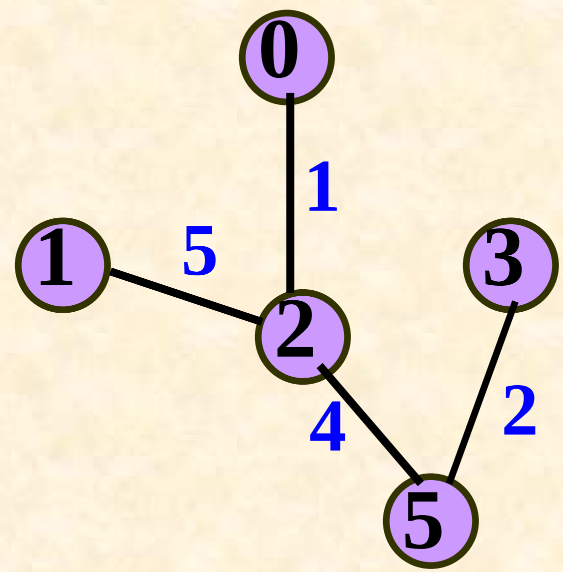

U
V-U
0
1
2
3
4
5
6
5
1
5
7
3
5
4
2
6


0
2
3
5
1
4
2


0
1
2
3
5
1
5
4
2


U
0
6
5
1
1
5
7
3
2
3
5
4
2
V-U
4
5
6


0
1
1
5
3
2
3
2
4
4
5


0
1
1
5
3
2
4
2
5

假设用邻接矩阵存储图。普里姆算法的实现需要增设**两个辅助数组*closedge[n]***和***TE[n-1]***.

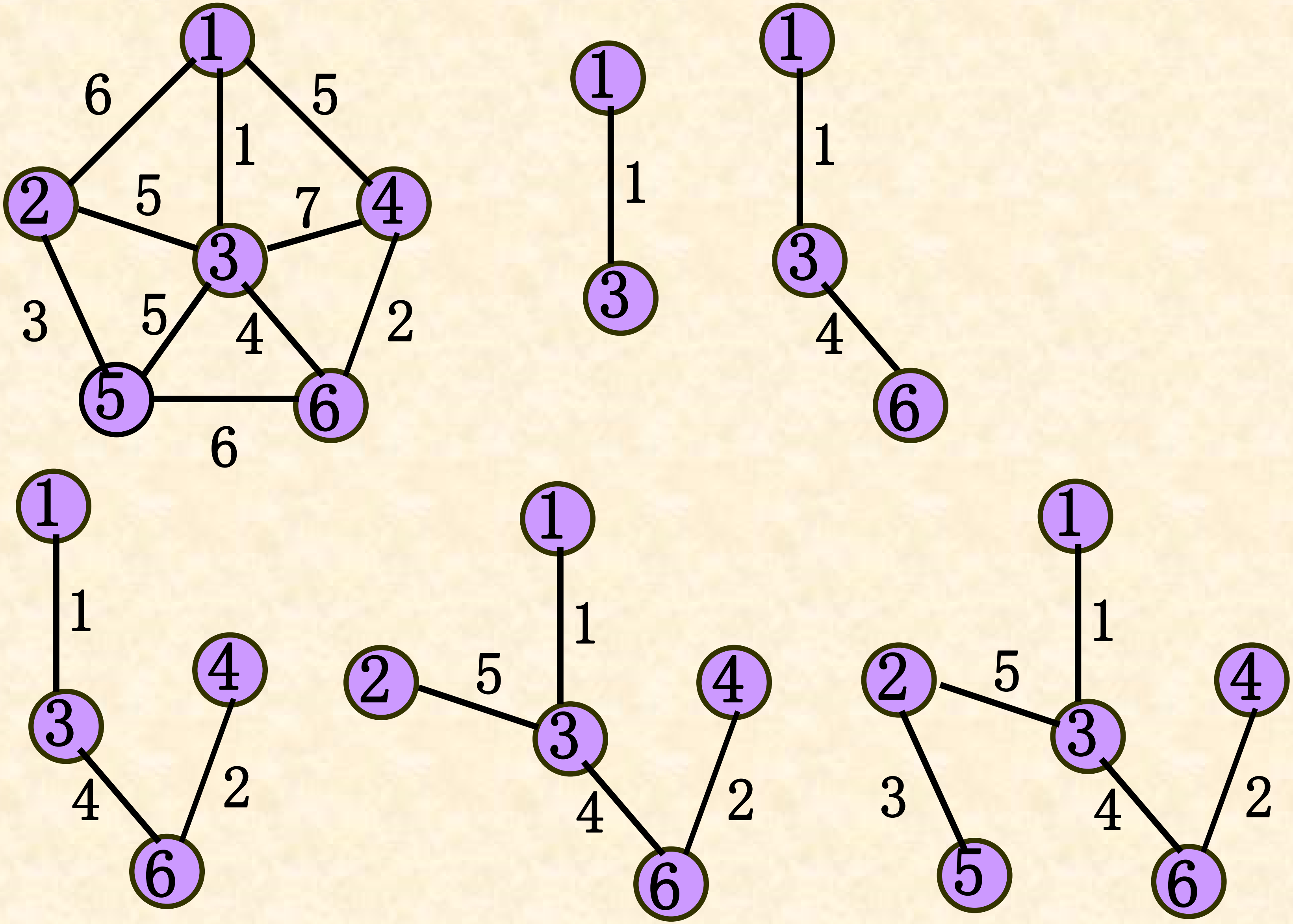
closedge[n]的每个数组元素由两个域构成：***Lowcost***和***Vex***，其定义如下：

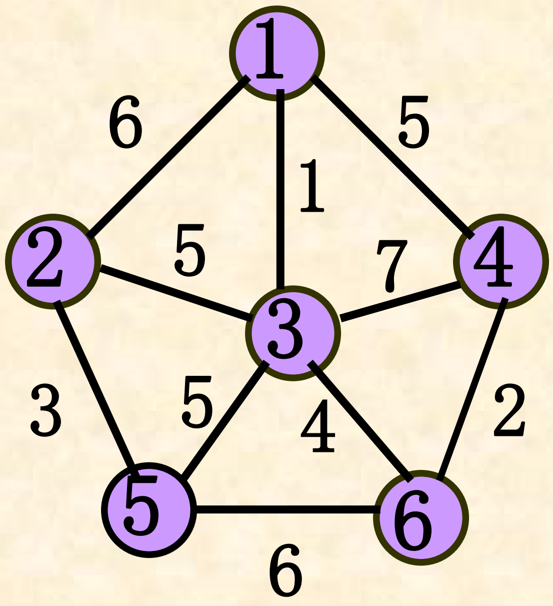
如果$v \notin U$，则 $closedge[v].Lowcost = \min\{weight(u,v) \mid u \in U\}$

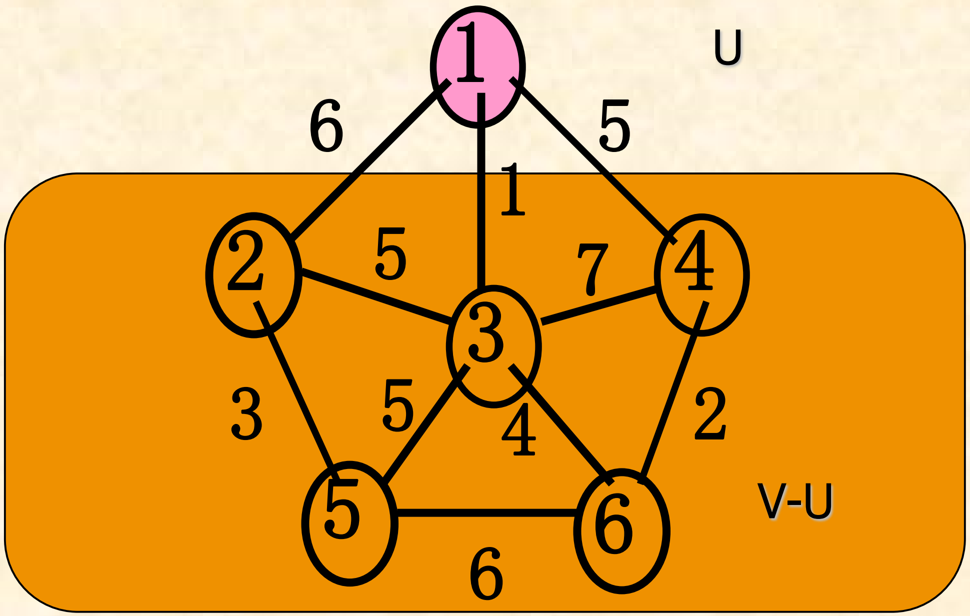
而***closedge[v].Vex***存储的是该边**依附在*U* 中的顶点*u***.

如果$v \in U$，则 $closedge[v].Lowcost = 0, closedge[v].Vex = -1$

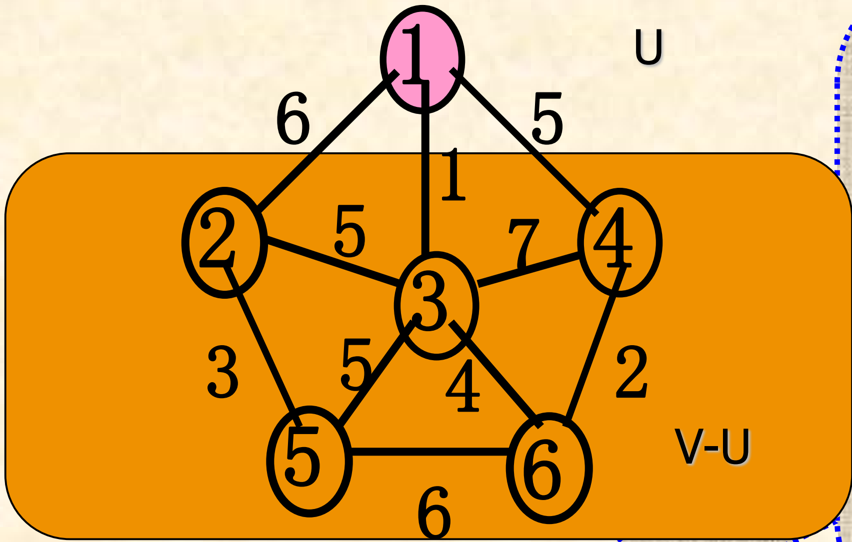
数组*TE[n-1]*是最小支撑树的边集合，每个数组元素***TE[i]***表示一条边，***TE[i]***由三个域***head***、***tail***和***cost***构成，它们分别存放边的始点、终点和权值。


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





V closedge	1	2	3	4	5	6	U	V-U
Vex Lowcost	-1 0	1 6	1 1	1 5	1 max	1 max	{1}	{2 ,3 ,4 ,5,6}



Prim:

FOR *i* = 1 TO *n* DO

(*Lowcost* (*closedge*[*i*])←*edge*[1][*i*].

Vex (*closedge*[*i*]) ← 1)

***Vex* (*closedge*[1]) ← -1.**

***count* ← 1.**

V / closedge	1	2	3	4	5	6	U	V-U
Vex	-1	1	1	1	1	1	{1}	{2 ,3 ,4 ,5,6}
Lowcost	0	6	1	5	max	max		

$v \leftarrow 0$. // 求当前权值最小的边和该边的终点v

$min \leftarrow max$.

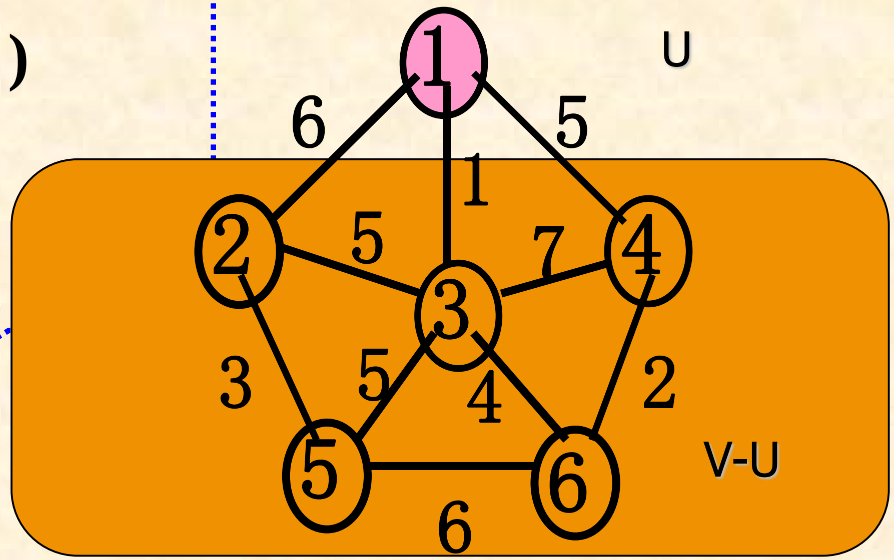
FOR j = 1 TO n DO

IF (vex ($closedge[j]$) ≠ -1 AND

$Lowcost(closedge[j]) < min$)

($v \leftarrow j$.

$min \leftarrow Lowcost\ (closedge[j])$)



V / closedge	1	2	3	4	5	6	U	V-U
Vex Lowcost	-1 0	1 6	1 1	1 5	1 max	1 max	{1}	{2 ,3 ,4 ,5 ,6}

If *v≠0* **THEN** // *v*加入*U*中

(***head***(***TE***[***count***]) ← ***Vex*** (***closedge***[***v***]) .

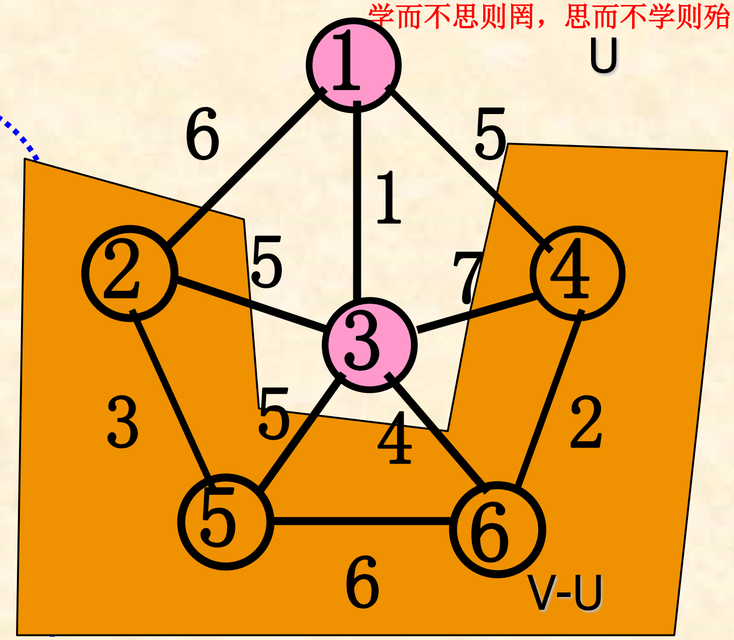
tail (***TE***[***count***]) ← ***v***.

cost (***TE***[***count***]) ← ***Lowcost***(***closedge***[***v***]) .

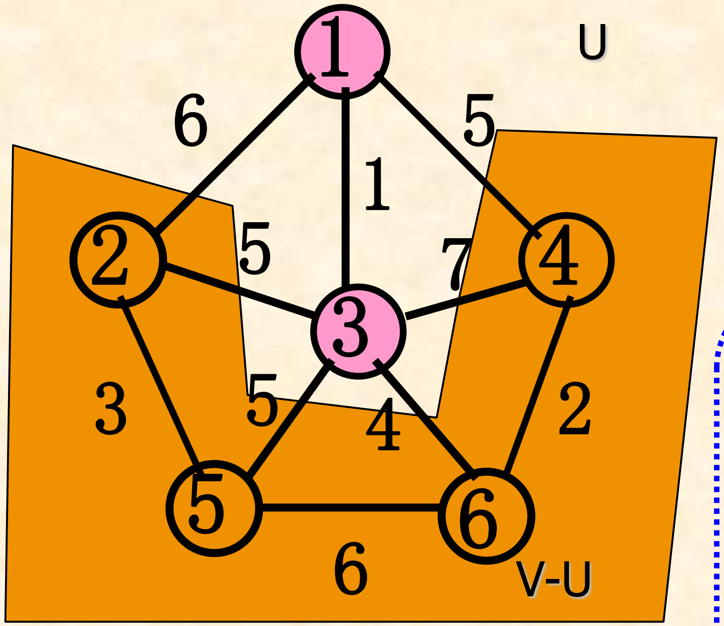
count ← ***count*** +**1.** // 计数器加1

Lowcost (***closedge***[***v***]) ← **0.** // 修改域值

Vex(***closedge***[***v***]) ← **-1.** // 顶点v进入集合U



V / closedge	1	2	3	4	5	6	U	V-U
Vex Lowcost	-1 0	1 6	-1 0	1 5	1 max	1 max	{1,3}	{2, 4 ,5 ,6}



FOR j **= 1 TO** n **DO** //修改某些顶点的值

IF(*Vex*(*closedge*[*j*]) ≠ -1 AND

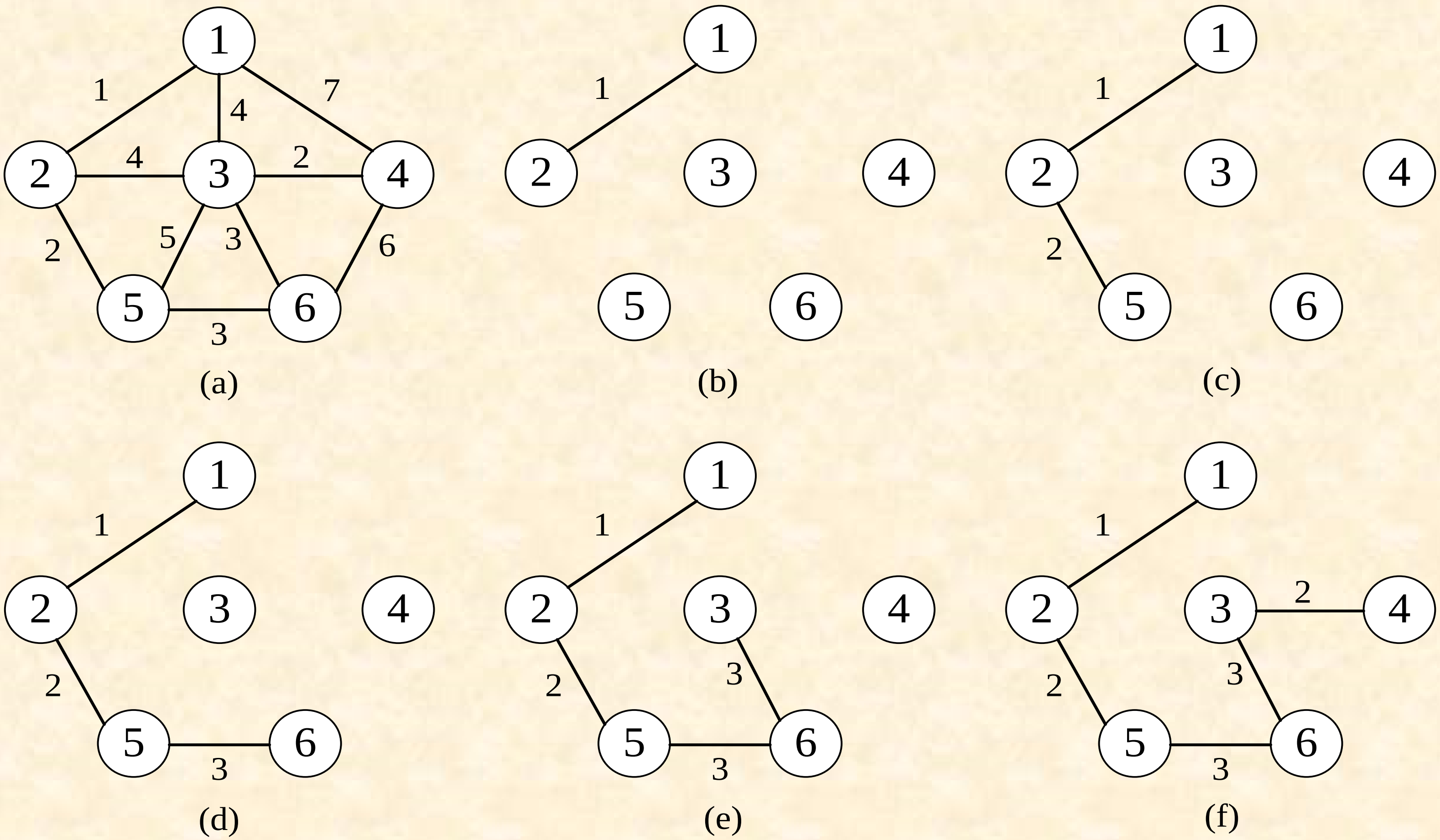
***edge*[*v*][*j*] < *Lowcost*(*closedge*[*j*])) THEN**

(*Lowcost*(*closedge*[*j*]) ← *edge*[*v*][*j*] .

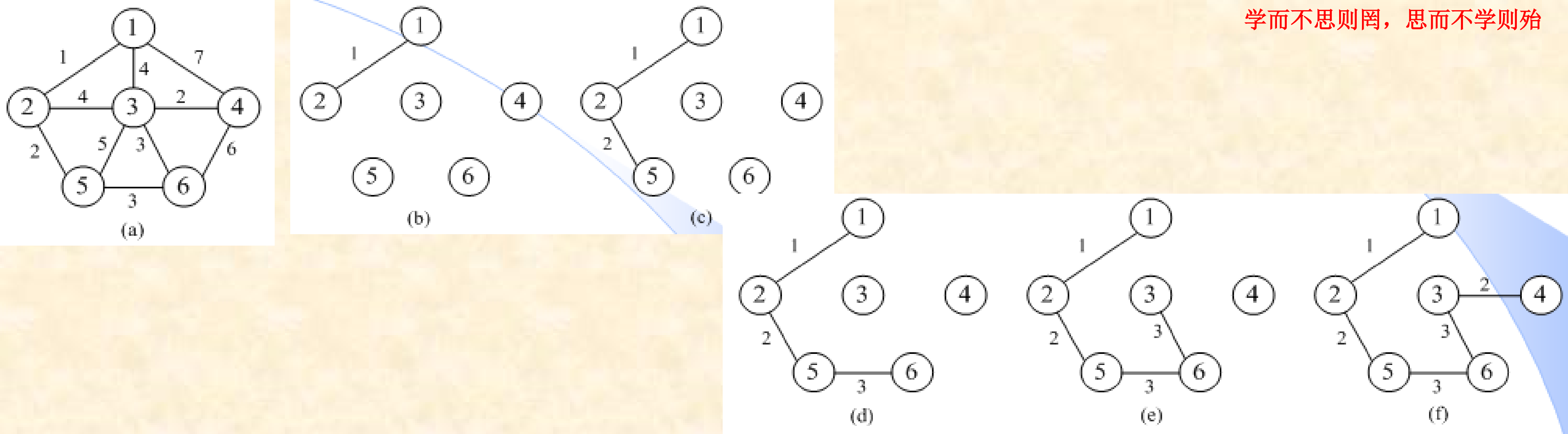
***Vex*(*closedge*[*j*]) ← *v*.))))**

V / closedge	1	2	3	4	5	6	U	V-U
Vex Lowcost	-1 0	**3** **5**	-1 0	1 5	**3** **5**	**3** **4**	{1,3}	{2,4 ,5,6 }

```
VOID MiniSpanTree_PRIM(MGraph G, VertexType u)
{  // 算法7.9
    int i, j, k;
    k = LocateVex ( G, u );
    for ( j=0; j<G.vexnum; ++j ) {     // 辅助数组初始化
        if (j!=k)
            { closedge[j].adjvex=u;  closedge[j].lowcost=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 = minimum(closedge);      // 求出T的下一个结点：第k顶点
        // 此时closedge[k].lowcost =
        // MIN{ closedge[vi].lowcost | closedge[vi].lowcost>0, vi∈V-U }
        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后重新选择最小边
                closedge[j].adjvex=G.vexs[k];
                closedge[j].lowcost=G.arcs[k][j].adj;
            }
    }
} // MiniSpanTree
```



(a) (b) (c) (d) (e) (f)



V closedge	2	3	4	5	6	U	V-U
Vex Lowcost	1 1	1 4	1 7	1 max	1 max	{1}	{2 ,3 ,4 ,5 ,6}
Vex Lowcost	-1 0	1 4	1 7	2 2	1 max	{1 ,2}	{3 ,4 ,5 ,6}
Vex Lowcost	-1 0	1 4	1 7	-1 0	5 3	{1 ,2 ,5}	{3 ,4 ,6}
Vex Lowcost	-1 0	6 3	6 6	-1 0	-1 0	{1 ,2 ,5 ,6}	{3 ,4}
Vex Lowcost	-1 0	-1 0	3 2	-1 0	-1 0	{1 ,2 ,3 ,5 ,6}	{4}
Vex Lowcost	-1 0	-1 0	-1 0	-1 0	-1 0	{1 ,2 ,3 ,4 ,5 ,6}	∅

2、克鲁斯卡尔(Kruskar)算法 （逐边加入）

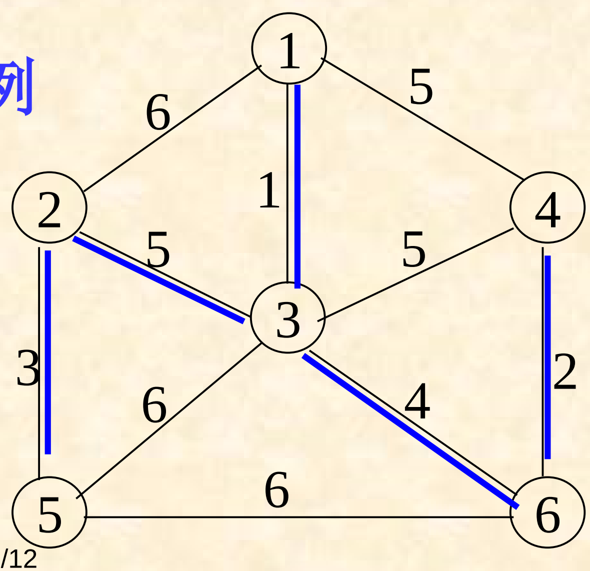
设连通网N=(V,E,C)，T为N的最小支撑树。初始时T={V,Φ}，即T中没有边，只有n个顶点，也就是n个连通分量。

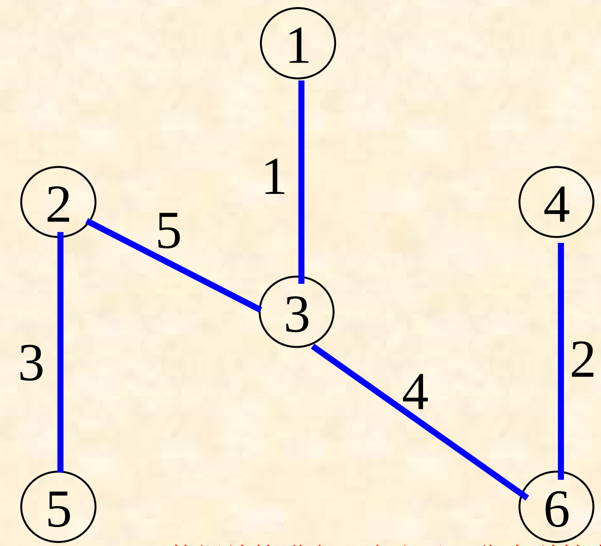
①在E中选择权值最小的边，并将此边从E中删除。

②如果此边的两个顶点在T的不同的连通分量中，则将此边加入到T中，从而导致T中减少一个连通分量；

如果此边的两个顶点在同一连通分量中，则重复执行① ②，直至T中仅剩一个连通分量时，终止操作。

例





克鲁斯卡尔算法实现：

顶点结点：

```
typedef   struct
{    int data;    //顶点信息
     int  jihe;
}VEX;
```

边结点：

```
typedef   struct
{    int   vexh, vext;  //边依附的两顶点
     int  weight;       //边的权值
     int  flag;         //标志域
}EDGE;
```

1）用**顶点数组**和**边数组**存放顶点和边信息

2）初始时，令每个顶点的**jihe互不相同**；每个边的**flag为0**

3）选出**权值最小**且flag为0的边

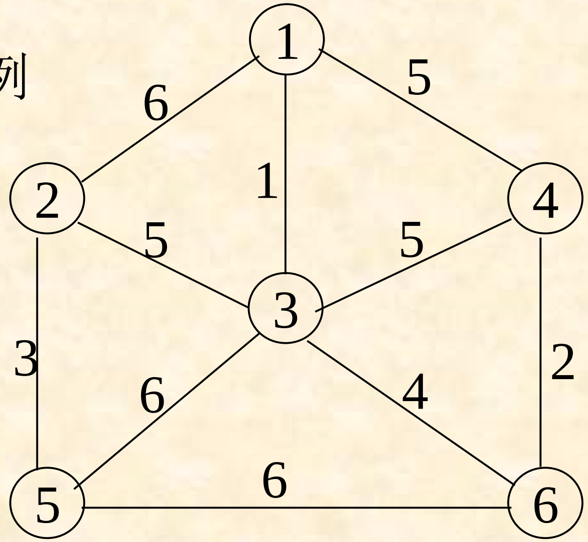
4）**若**该边依附的两个顶点的**jihe值不同**，即**非连通**，则令该边的**flag=1**，**选中该边**；再**令**该边依附的两顶点的jihe以及两集合中所有顶点的**jihe 相同**；若该边依附的两个顶点的jihe值相同，即**连通**，**则令该边的flag=2**，即**舍去该边**

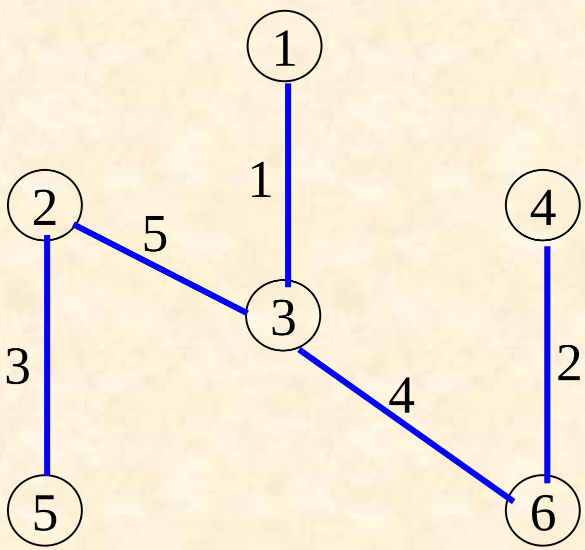
5）重复上述步骤，直到选出n-1条边为止

✧**算法描述：**

Ch6_30.txt

例



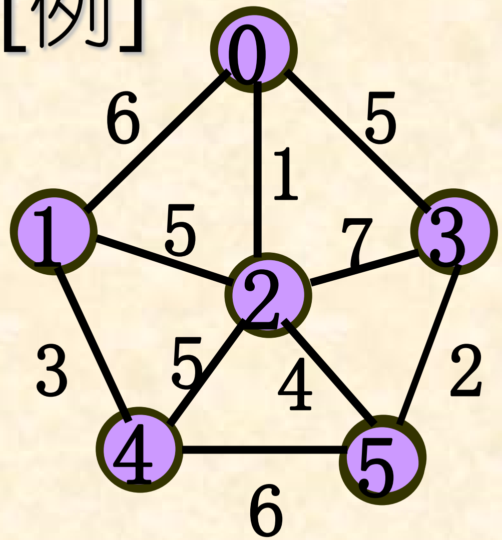


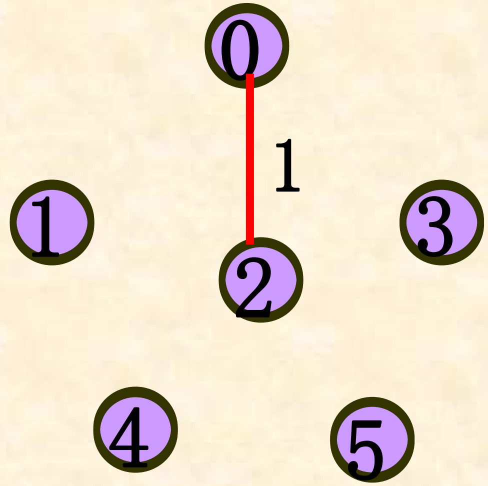
	data	jihe
1	1	1 2
2	2	2
3	3	3 1 2
4	4	4 1 2
5	5	5 2
6	6	6 4 1 2

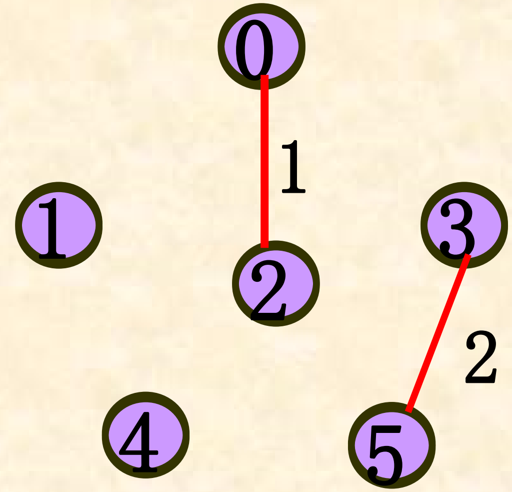
	vexh	vext	weight	flag
0	1	2	6	0
1	1	3	1	0 1
2	1	4	5	0 2
3	2	3	5	0 1
4	2	5	3	0 1
5	3	4	5	0
6	3	5	6	0
7	3	6	4	0 1
8	4	6	2	0 1
9	5	6	6	0

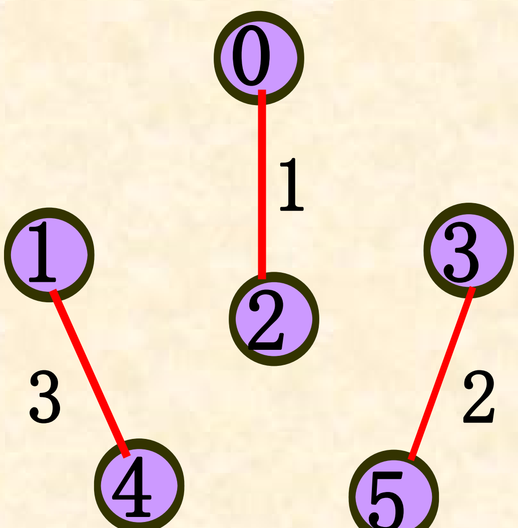
Ch6_30.c

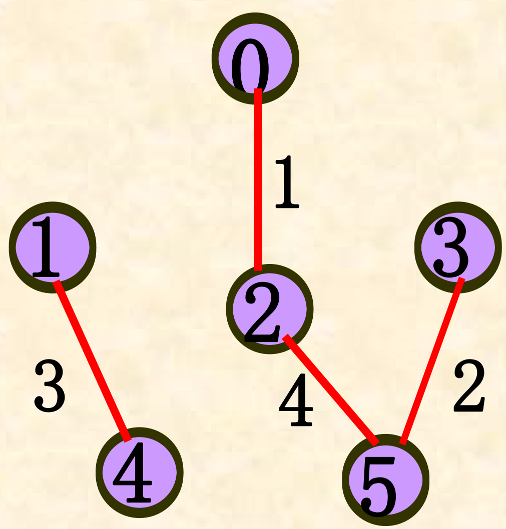
[例]

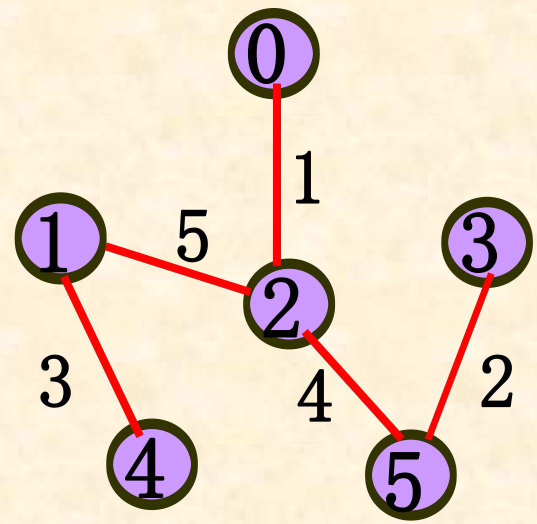












普里姆(Prim)算法的时间复杂性为O(n^2)，算法适用于求边稠密网的最小支撑树。

克鲁斯卡尔(Kruskar)算法正好相反，它适用于求边稀疏网的最小支撑树，它的时间复杂性为O(eloge).