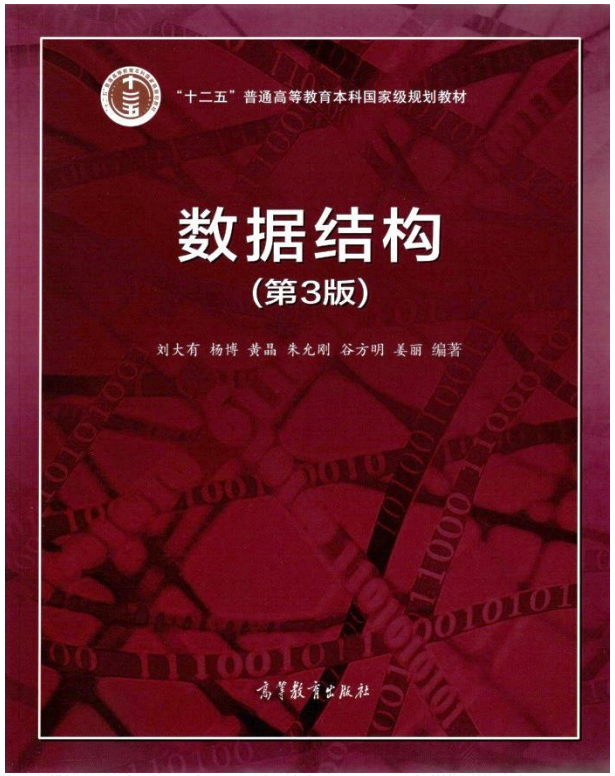


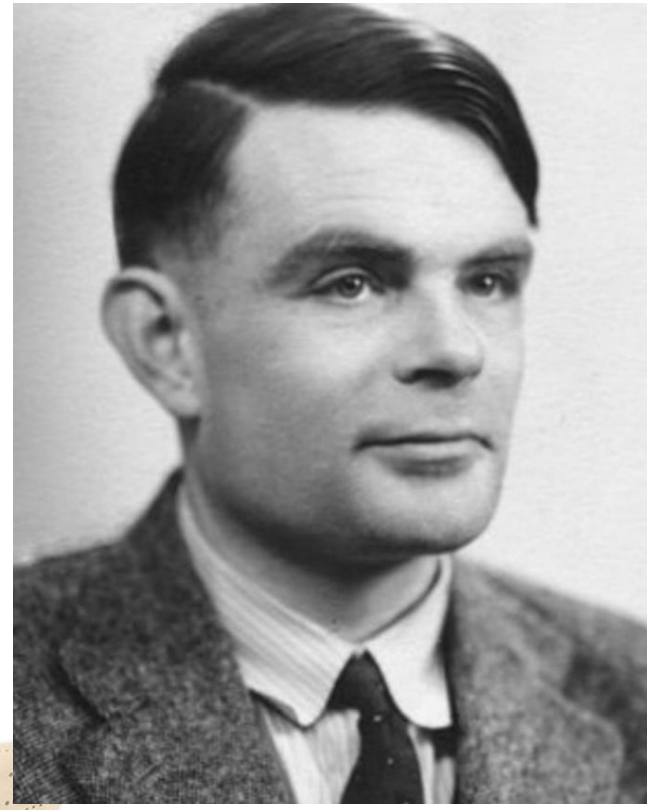
图的存储结构与遍历

- 图的概念简要回顾
- 图的存储结构
- 图的遍历
- 图遍历的应用

数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

Last updated on 2022.11



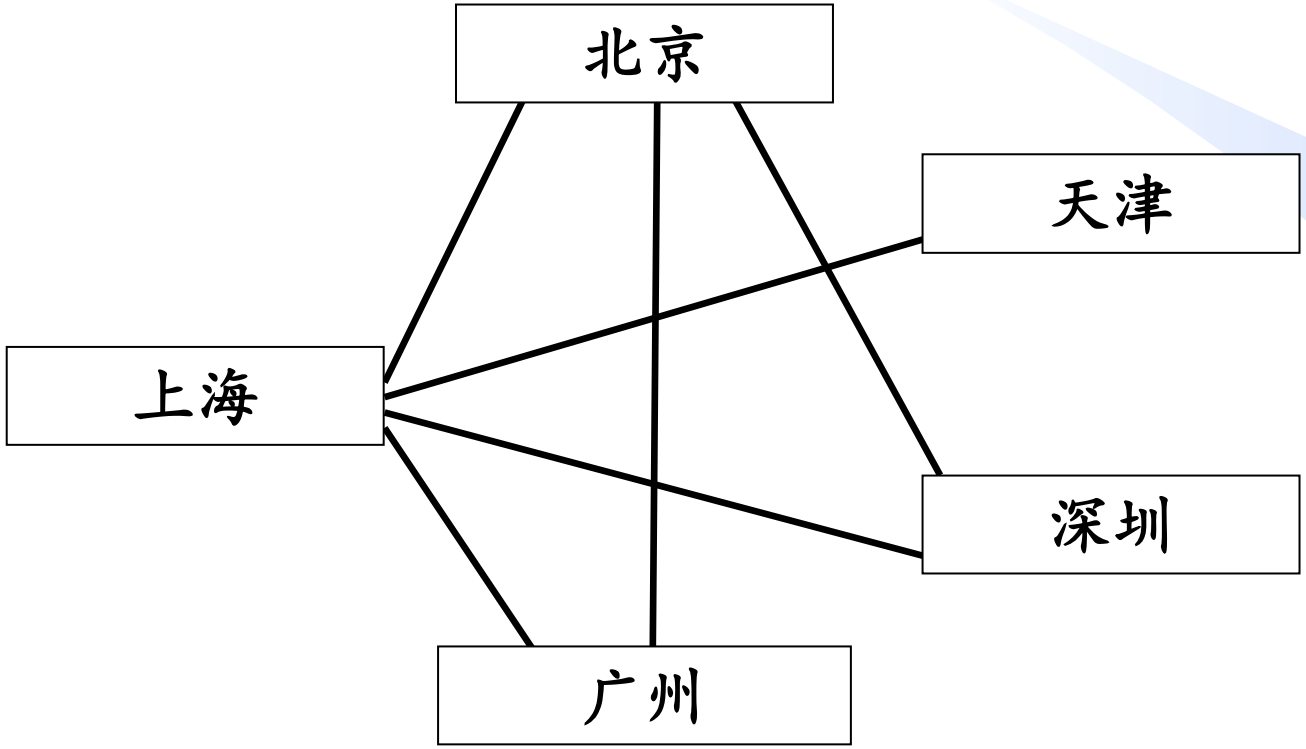
Programming is a skill best acquired by practice and example rather than from books.

— *Alan Turing*

计算机科学之父
英国皇家科学院院士
普林斯顿大学博士
曼彻斯特大学教授
剑桥大学研究员

- 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）的前驱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。

城市航线网



计算机网络

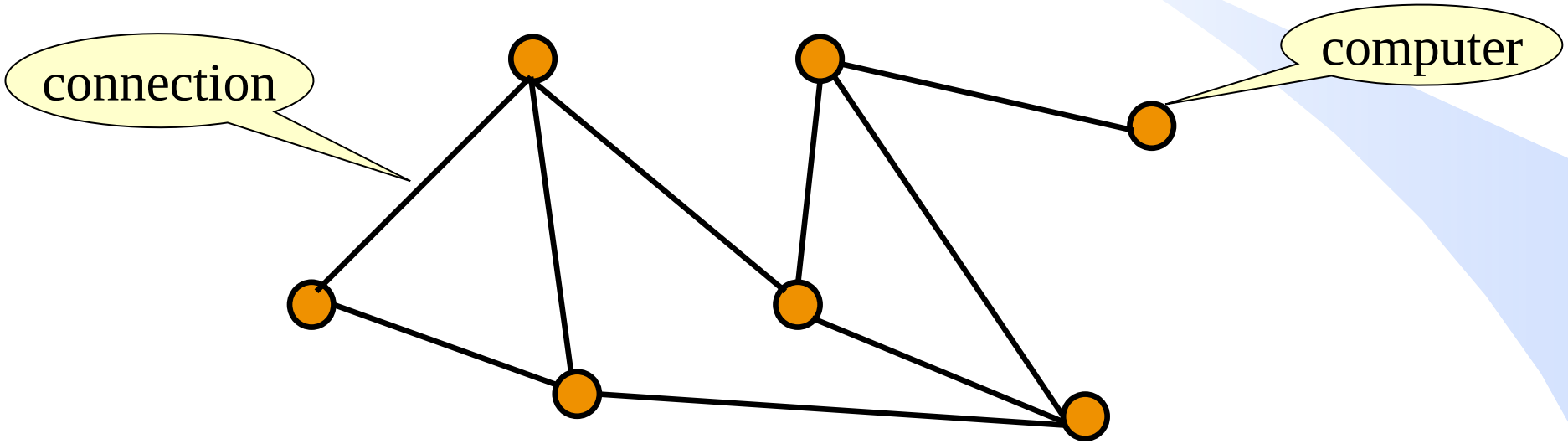


图 VS. 树

- 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

图的基本概念

Vertex

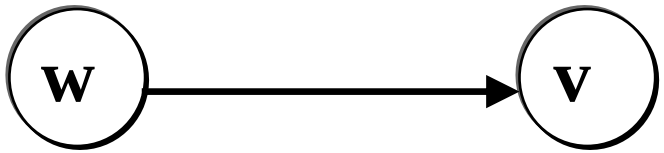
Edge

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

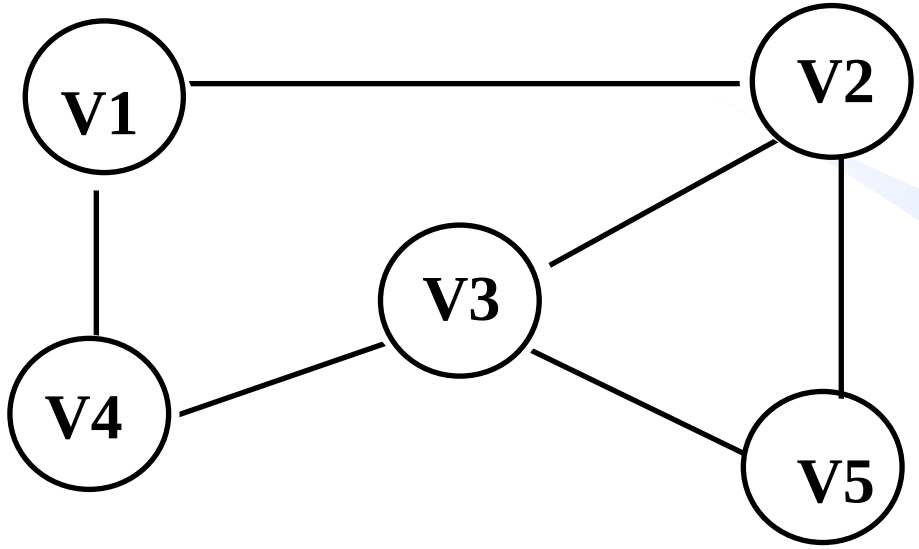
若图中的边限定为从一个顶点指向另一个顶点，则称此图为有向图。

若图中的边无方向性，则称之为无向图。

定义 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。



无向图

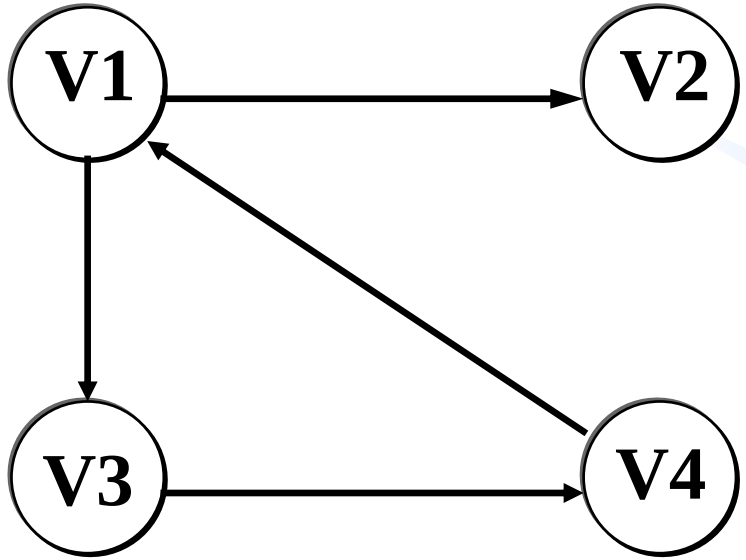


G=(V, E)

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

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

有向图



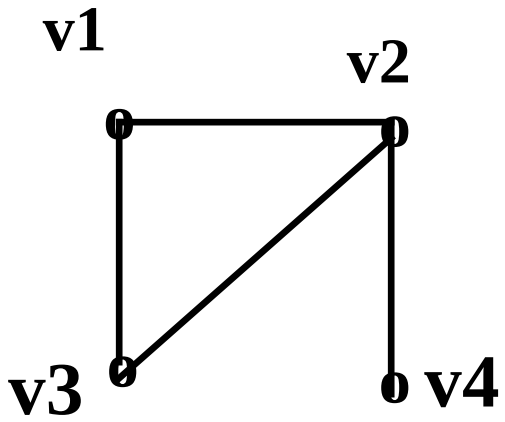
G=（V，E）

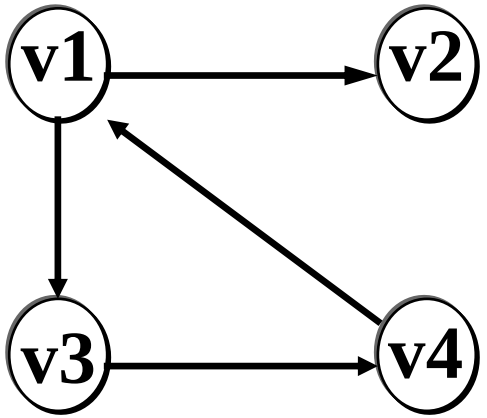
V={ V1，V2，V3，V4}

E={<V1,V2>, <V1,V3>, <V3,V4>, <V4,V1>}

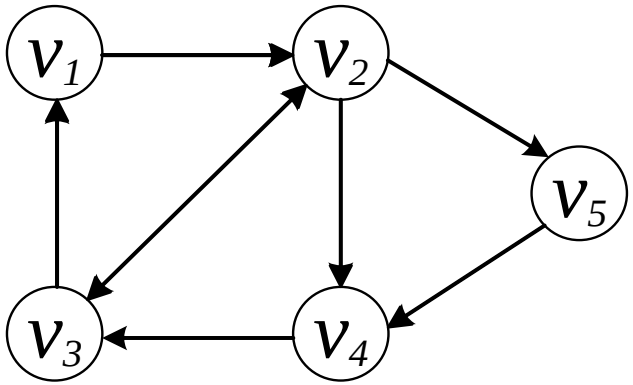
定义 在一个无向图中，若存在一条边(w, v)，则称w, v为此边的两个**端点**，它们是相邻的，并称它们互为**邻接顶点**。

在一个有向图中，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。

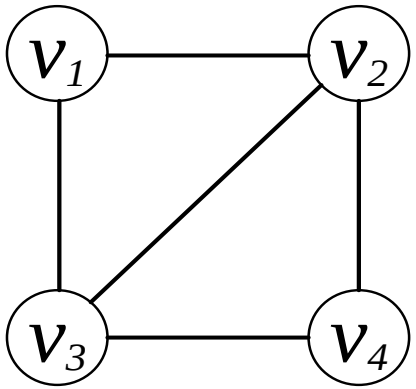




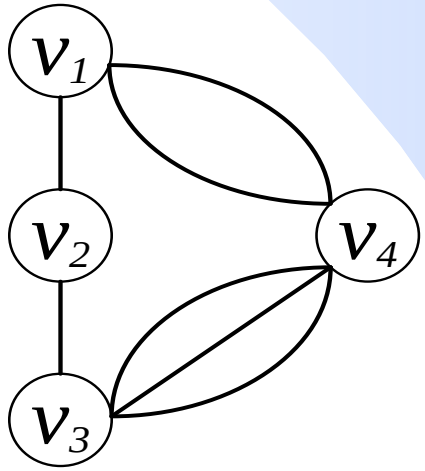
定义　由于E是边的集合，故一个图中不会多次出现一条边。若去掉此限制，则由此产生的结构称为多重图。图 (c)就是一个多重图。



(a)



(b)



(c)

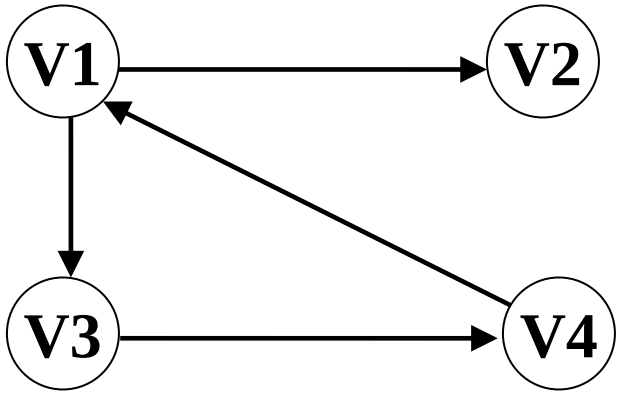
很多问题都可以抽象成一个图结构，考虑如下三个例子：

- 将电影界的所有演员构成顶点集V，其中两位演员*u*和*v*如果共同出演过至少一部影片，那么在*u*和*v*之间连接一条边。演员之间的这种合作关系看作对等关系。按照这种方式建立的图是无向图。
- 将多个城市构成顶点集V，如果城市a和城市b之间有一条高速公路，则在a和b之间连接一条边。允许在两个城市之间修建多条高速公路。按照这种方式建立的图是多重图。

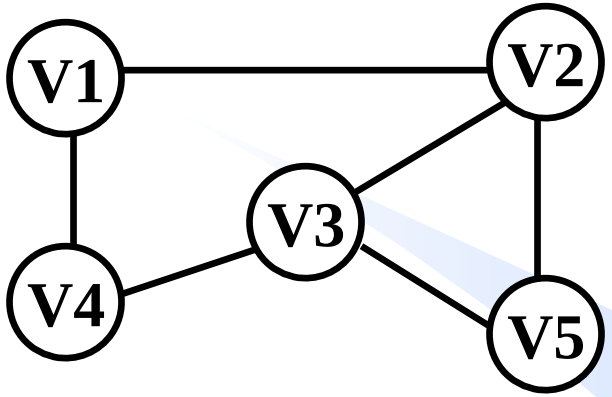
度

定义

- 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v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 ，顶点的度=入度+出度。



Graph1



Graph2

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

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

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

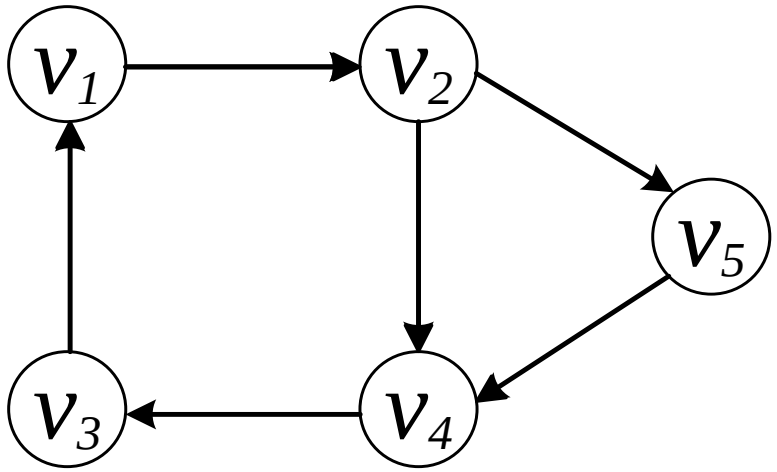
因为一条边关联两个顶点，而且使得这两个顶点的度数分别增加1。因此顶点的度数之和就是边的两倍。

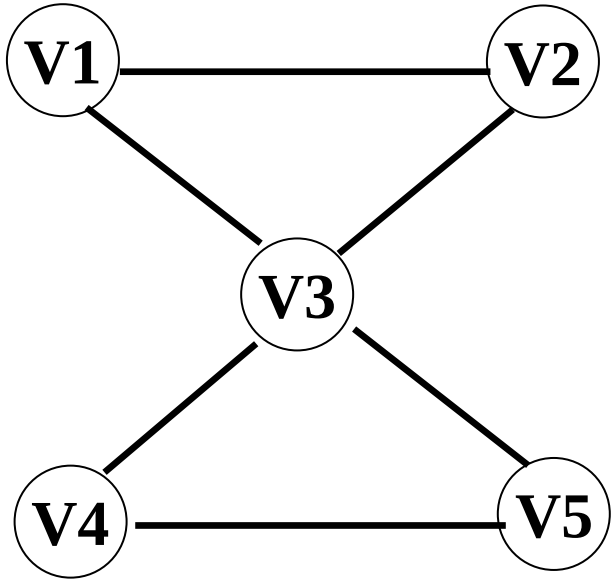
定义 设G是图，若存在一个顶点序列 $v_p, v_1, v_2, \cdots, v_{q-1}, v_q$ 使得 $<v_p, v_1>, <v_1, v_2>, \cdots, <v_{q-1}, v_q>$ 或 $(v_p, v_1), (v_1, v_2), \cdots, (v_{q-1}, v_q)$ 属于E(G)，则称v_p到v_q存在一条路径，其中v_p称为起点，v_q称为终点。

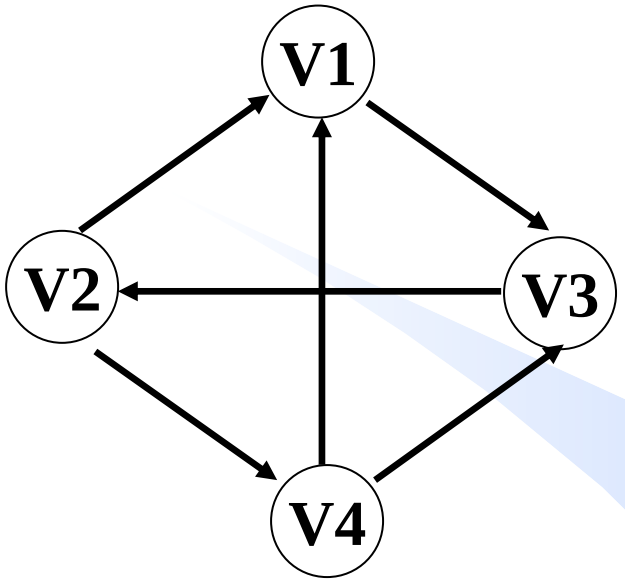
路径的长度是该路径上边的个数。

- 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。

- 下图中，v1到v3之间存在一条路径v1, v2, v5, v4, v3，同时这也是一条简单路径；v1, v2, v5, v4, v3, v1是一条简单回路。







路径：v1 v3 v4 v3 v5

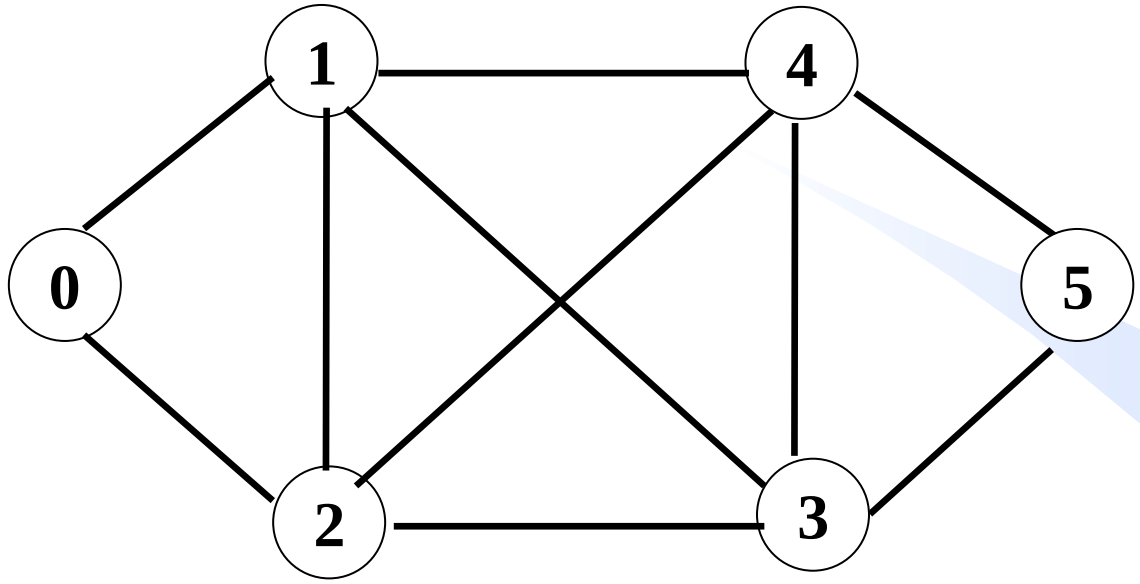
简单路径：v1 v3 v5

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



欧拉回路（一笔画）

每条边经过一次

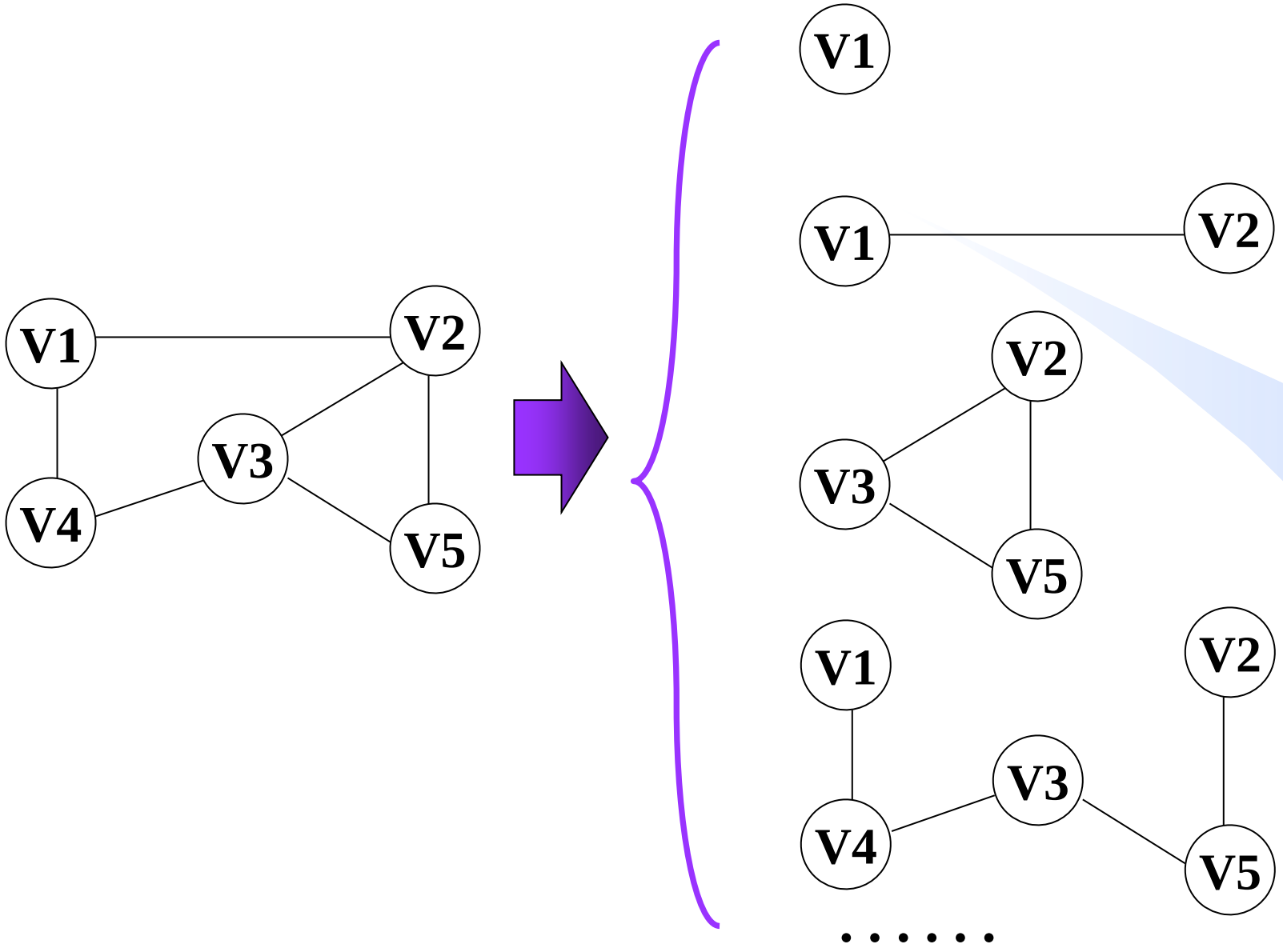
5-4-1-2-0-1-3-2-4-3-5

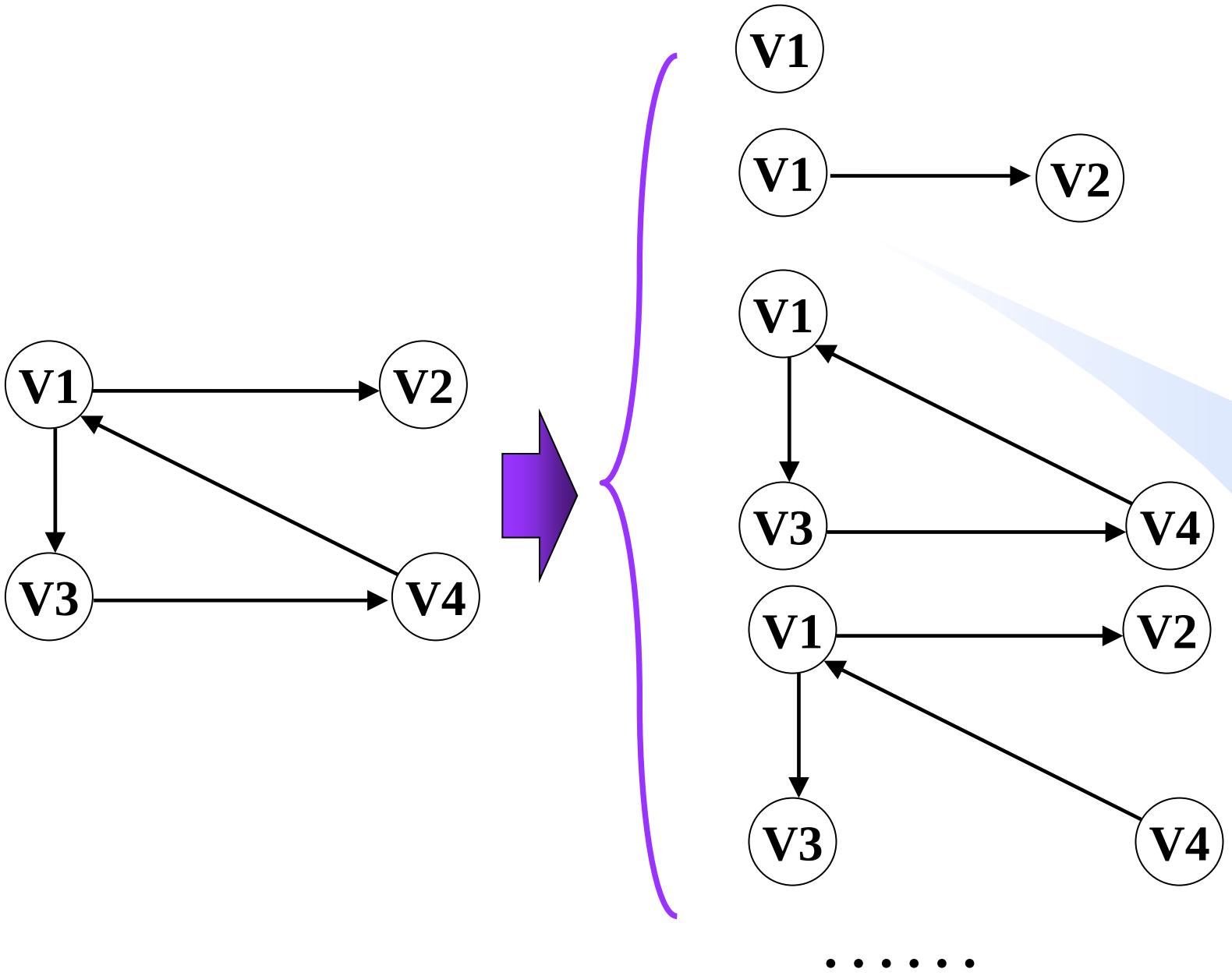
汉密尔顿回路

每个顶点经过一次

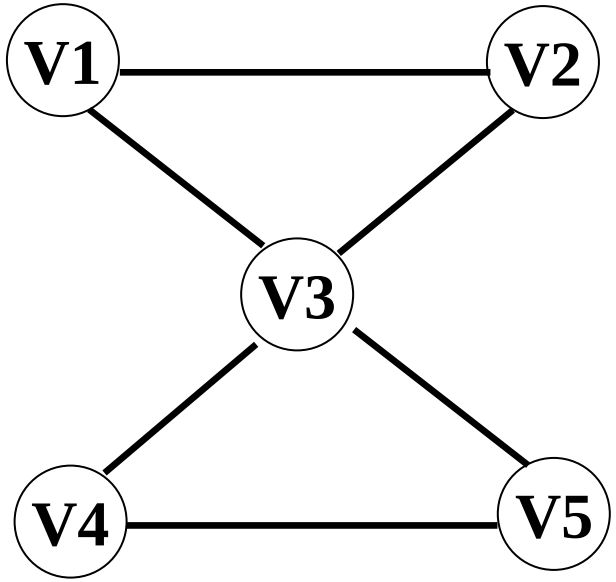
5-4-1-0-2-3-5

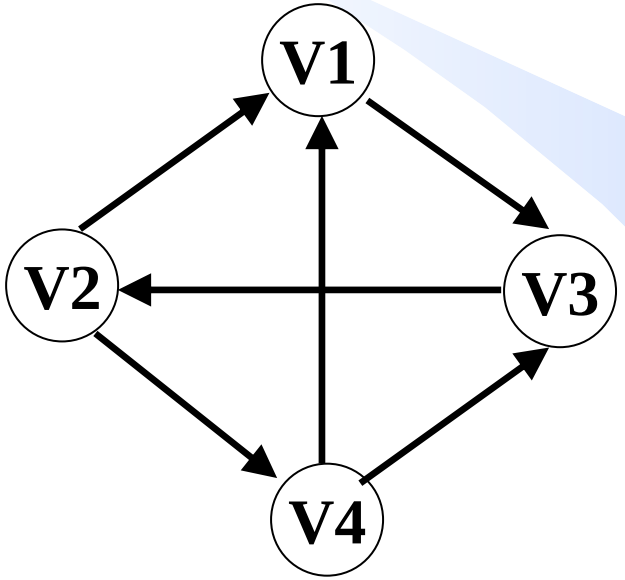
定义　设G，H是图，如果$V(H) \subseteq V(G)$，$E(H) \subseteq E(G)$，则称H是G的子图，G是H的母图。如果H是G的子图，并且$V(H) = V(G)$，则称H为G的支撑子图。


V1
V2
V3
V4
V5
V1
V2
V1
V2
V3
V5
V1
V2
V3
V4
V5
. . . . . .


V1
V2
V3
V4
V1
V1
V2
V1
V3
V4
V1
V2
V3
V4
. . . . . .

- **定义** 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)中任意两个不同的顶点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

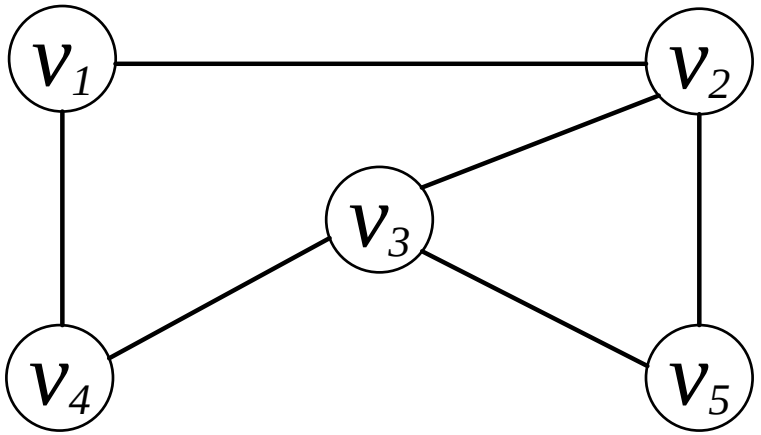




连通图　　强连通图

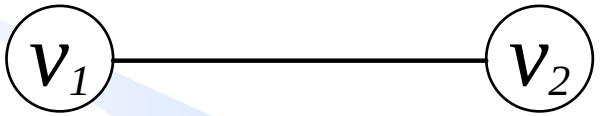
- 设图G = (V，E)是无向图，若G的子图G_K是一个连通图，则称G_K为G的连通子图；G的极大连通子图称为G的连通分量。
- 设图G = (V，E)是有向图，若G的子图G_K是一个强连通图，则称G_K 为G的强连通子图；G的极大强连通子图称为G的强连通分量。

一个图的连通子图不一定唯一

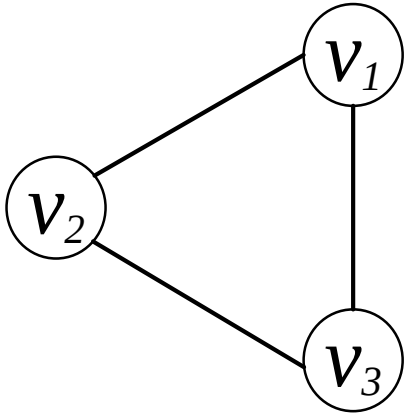


(a)

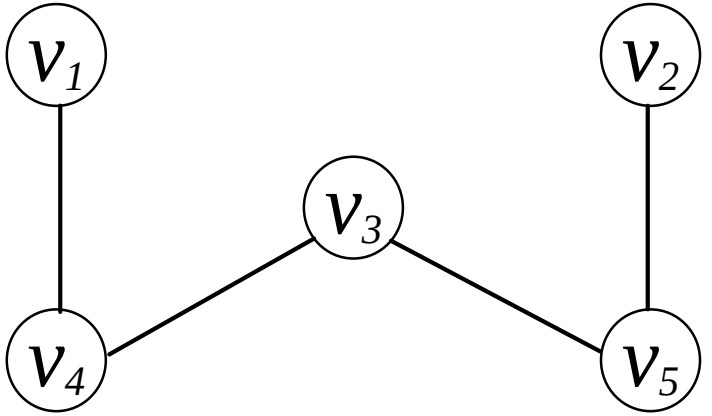
(b)



(c)

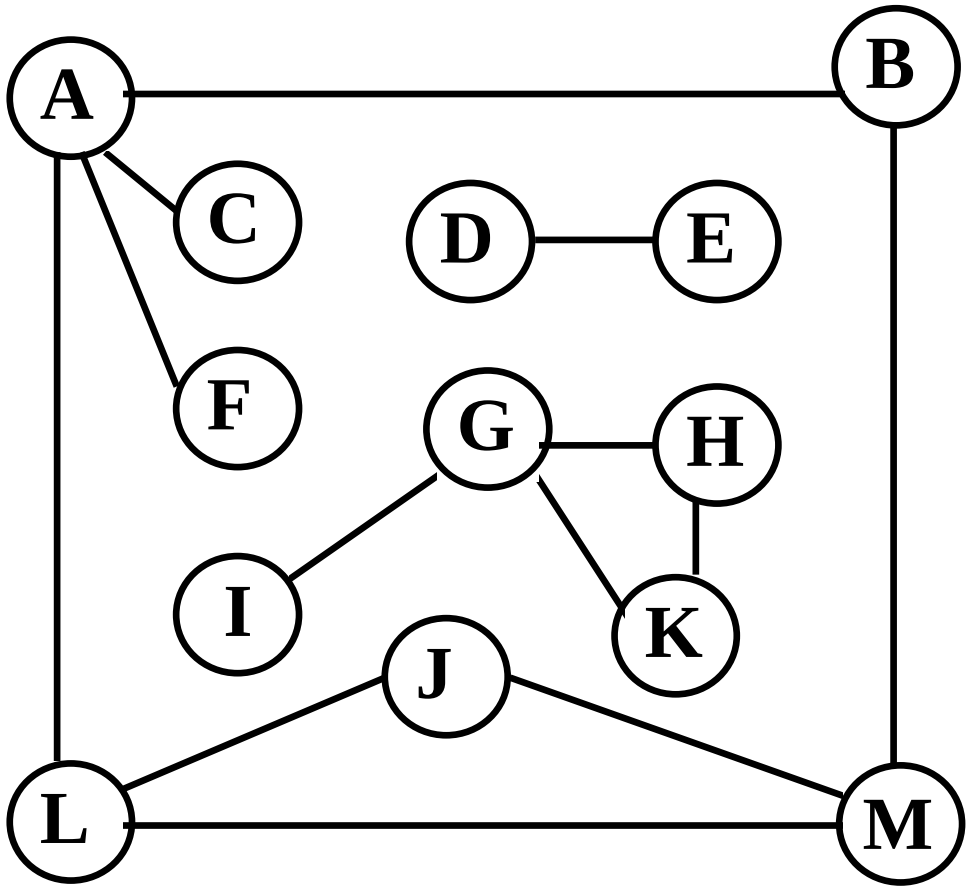


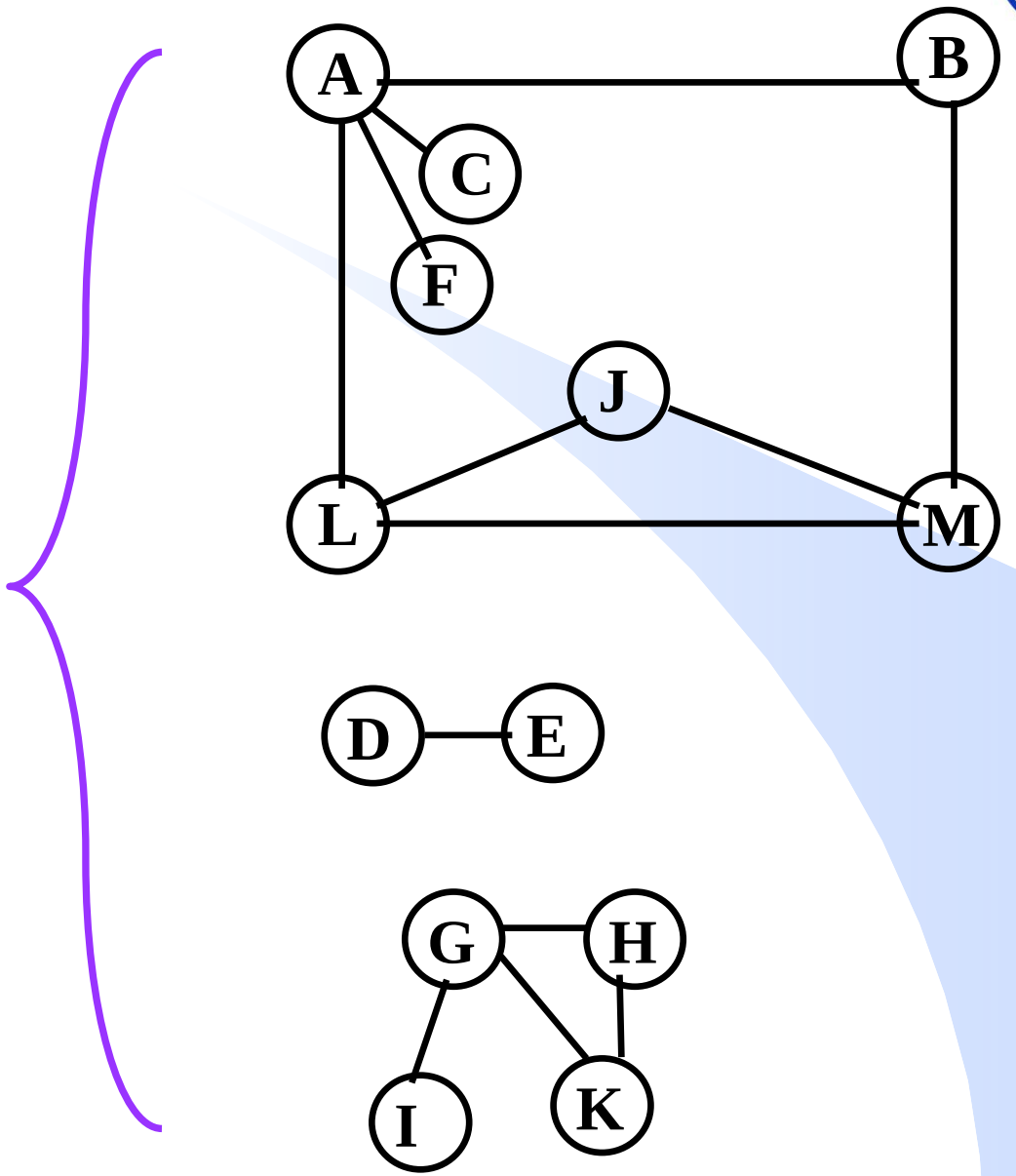
(d)



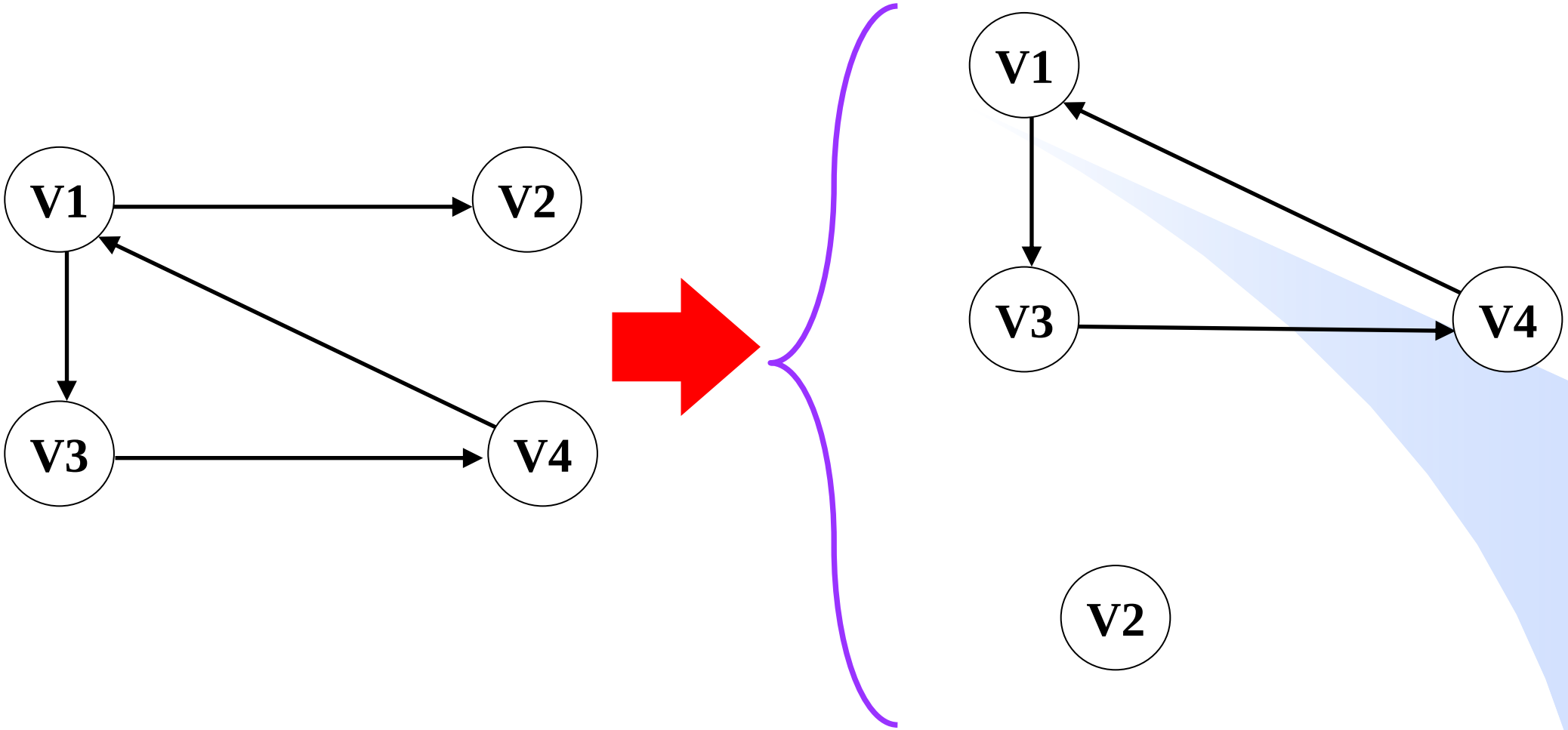
(e)

一个图的连通分量不一定唯一





一个有向图的强连通分量不一定唯一



无向图	有向图
连通图	强连通图
连通子图	强连通子图
连通分量	强连通分量

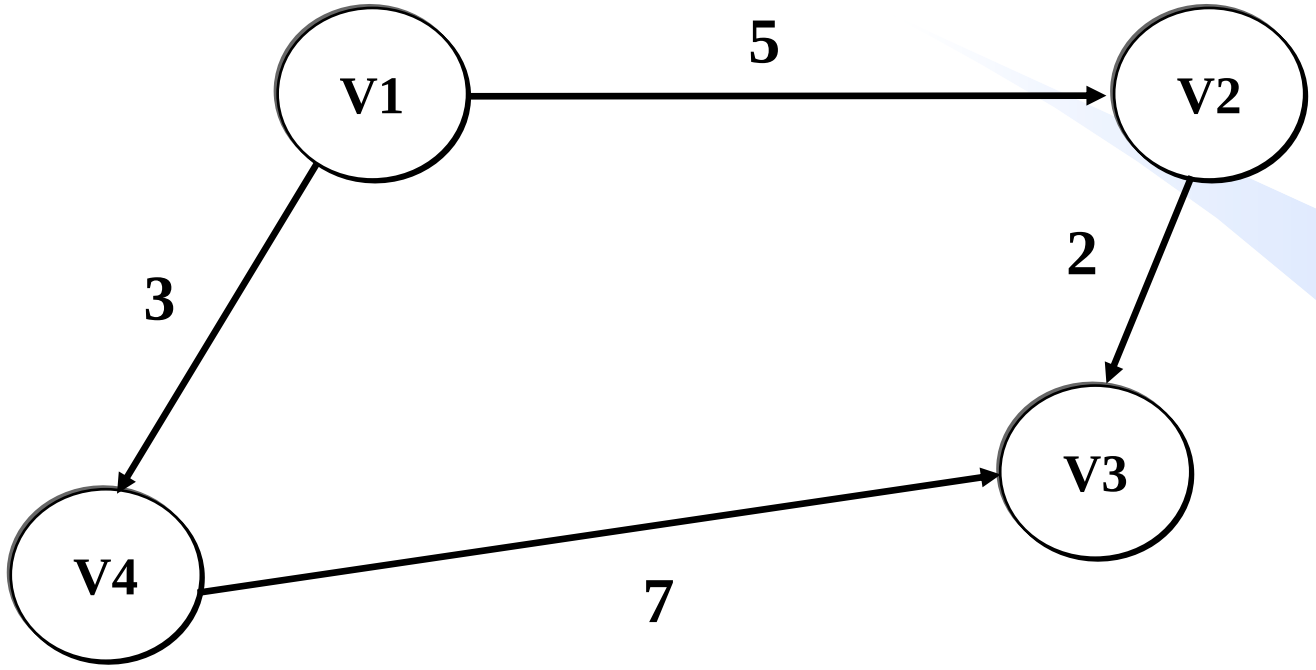
- 有时候，图不仅要表示出元素之间是否存在某种关系，同时还需要表示与这一关系相关的某些信息。
- 例如在计算机网络对应的图中，顶点表示计算机，顶点之间的边表示计算机之间的通讯链路。实际中，为了管理计算机网络，我们需要这个图包含更多的信息，例如每条通讯链路的物理长度、成本和带宽等信息。为此，为传统图中的每条边添加相应的数据域以记录所需要的信息。

定义 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为权图，记为G = (V, E, w)。记$w(u, v)$为以u, v为端点的边的权值，规定：

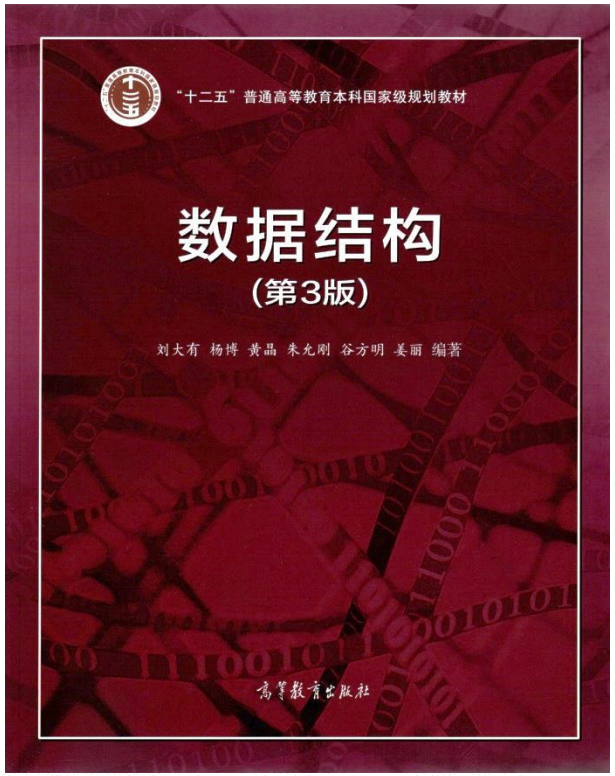
$\forall u \in V$, 有$w(u, u)=0$

$\forall u, v \in V$, 若顶点u和v之间不存在边，则则$w(u, v)=+\infty$

- **定义** 若$\sigma=(v_0,v_1,v_2,\cdots,v_k)$是权图G中的一条路径，则$|\sigma|=\sum_{i=1}^{k}w(v_{i-1},v_i)$称为路径$\sigma$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
V2
V3
V4
5
3
2
7

无向图	有向图
无向边	有向边（弧）
端点	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
连通子图	强连通子图
连通分量	强连通分量



图的存储结构与遍历

- 图的概念简要回顾
- **图的存储结构**
- 图的遍历
- 图遍历的应用

数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

Last updated on 2022.11

图的存储结构

- 邻接矩阵
- 邻接表

邻接矩阵

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

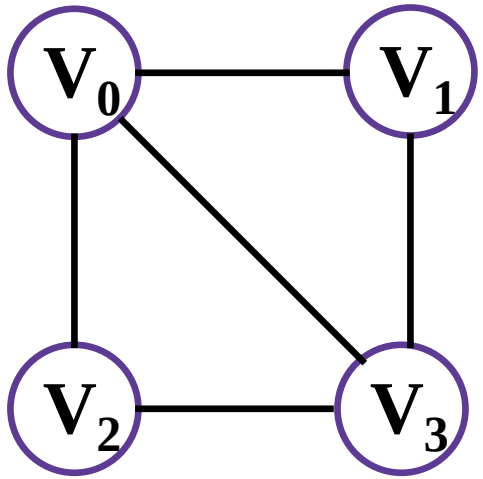
(1)若非权图，则：

- $a_{ii}=0$；
- $a_{ij}=1$，当$i\neq j$且v_i与v_j之间存在边；
- $a_{ij}=0$，当$i\neq j$且v_i与v_j之间不存在边。

(2)若权图，则：

- $a_{ii}=0$；
- $a_{ij}=$边的权值，当$i\neq j$且v_i与v_j之间存在边；
- $a_{ij}=\infty$，当$i\neq j$且v_i与v_j之间不存在边。

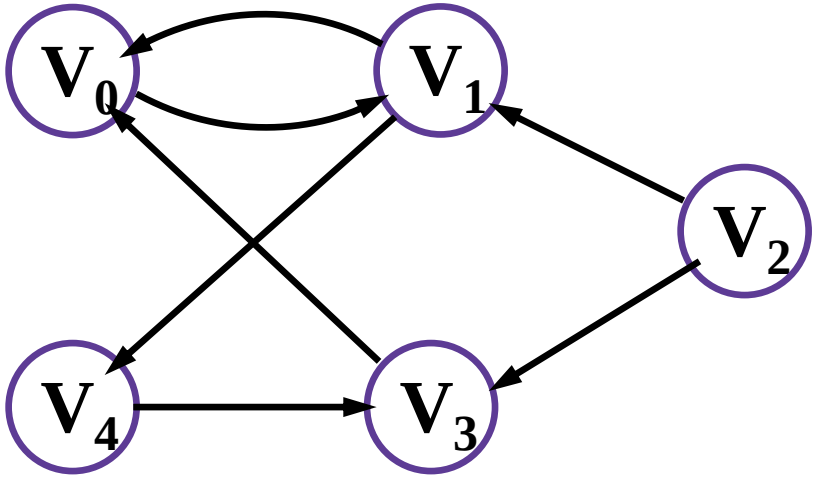
[例]无向图的邻接矩阵



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

无向图的邻接矩阵是对称矩阵。

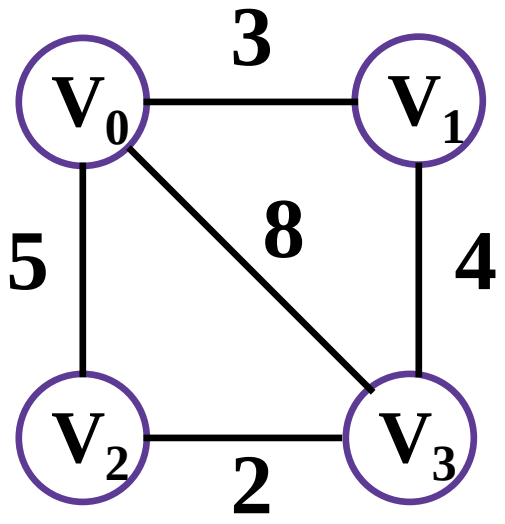
[例]有向图的邻接矩阵



A

	0	1	2	3	4
0	0	1	0	0	0
1	1	0	0	0	1
2	0	1	0	1	0
3	1	0	0	0	0
4	0	0	0	1	0

[例]权图的邻接矩阵



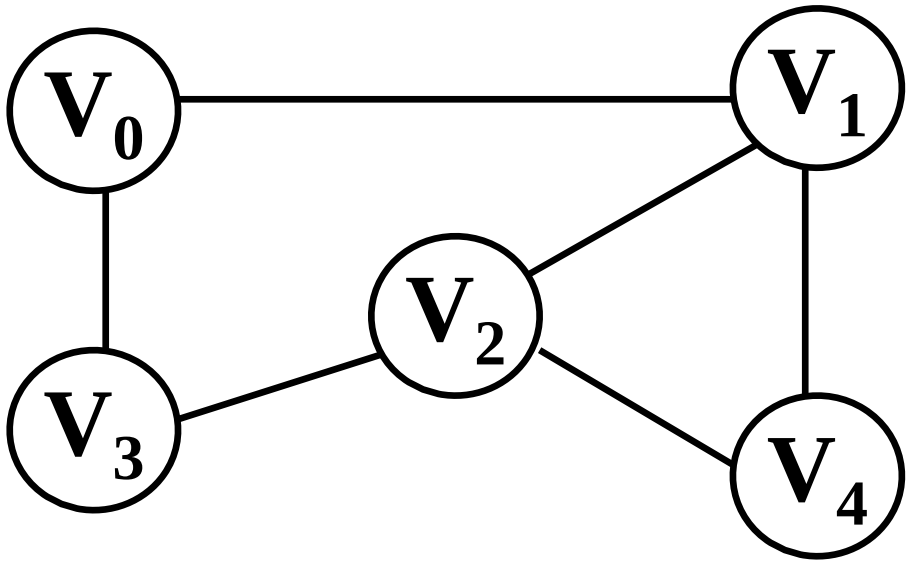
$$A=\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} \mathbf{0} & 3 & 5 & 8 \\ 3 & \mathbf{0} & \infty & 4 \\ 5 & \infty & \mathbf{0} & 2 \\ 8 & 4 & 2 & \mathbf{0} \end{pmatrix} \end{array}$$

特点：无向图的邻接矩阵是对称矩阵，有向图邻接矩阵不一定对称。

借助邻接矩阵可方便的求出图中顶点的度

无向无权图 矩阵的第*i*行（或第*i*列）的1的个数是**顶点V_i的度**。

第*i*行有一个1就意味着有一条以顶点V_i为端点的边



	0	1	2	3	4
0	0	1	0	1	0
1	1	0	1	0	1
2	0	1	0	1	1
3	1	0	1	0	0
4	0	1	1	0	0

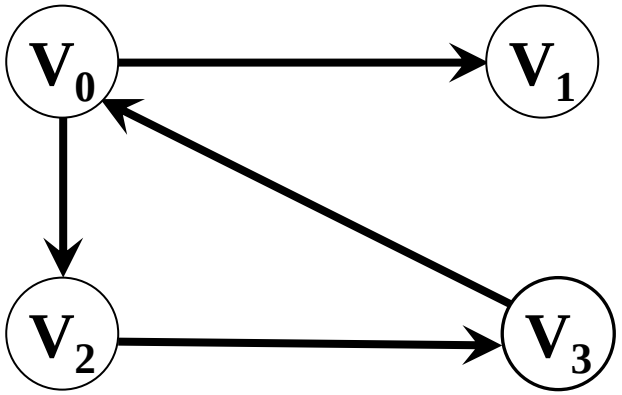
有向无权图

- 邻接矩阵第i行的1的个数为顶点V_i的出度

  第i行有一个1，就意味着有一条由V_i引出的边

- 第i列的1的个数为顶点V_i的入度。

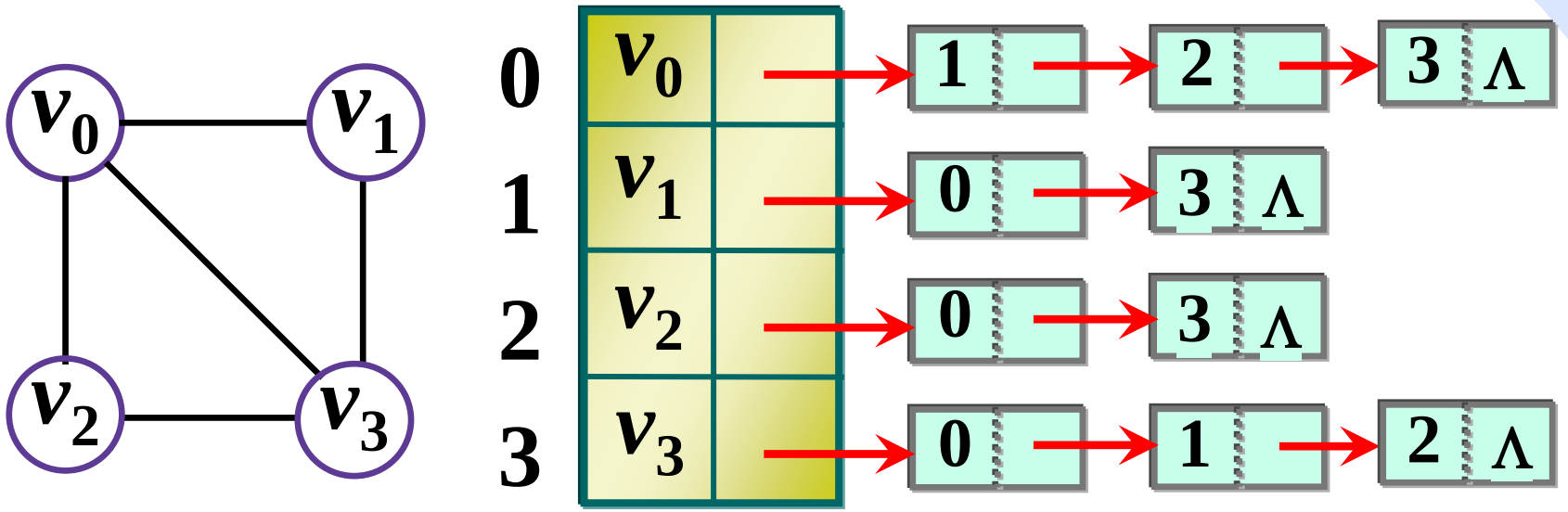
  第i列有一个1，就意味着有一条引入（指向）V_i的边



	0	1	2	3
0	0	1	1	0
1	0	0	0	0
2	0	0	0	1
3	1	0	0	0

2、邻接表

- 顺序存储顶点表。
- 对图的每个顶点建立一个单链表，第 i 个单链表中的结点包含顶点v_i的所有邻接顶点（边链表）。
- 由顺序存储的顶点表和链接存储的边链表构成的图存储结构被称为邻接表。



➢与顶点 v 邻接的所有顶点以某种次序组成的单链表称为顶点 v 的边链表。

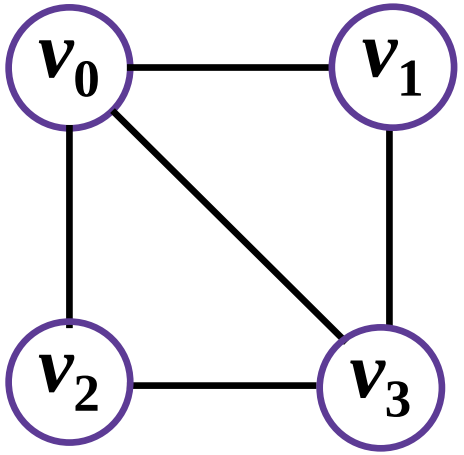
➢边链表的每一个结点叫做边结点，对于非权图和权图边结点结构分别为：

VerAdj	*link*

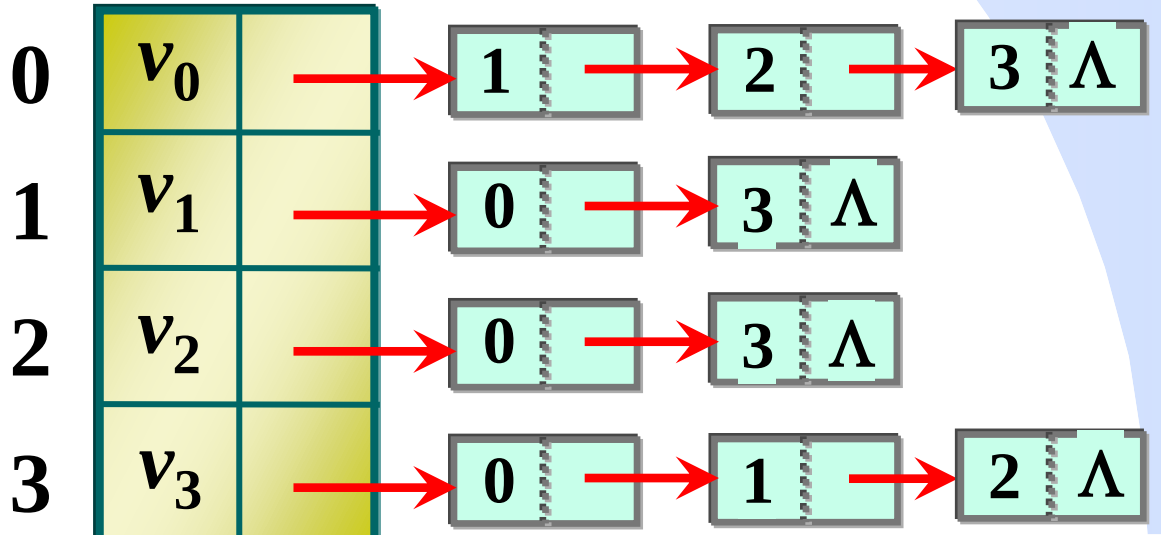
VerAdj	*cost*	*link*

其中，域*VerAdj*存放 v 的某个邻接顶点在顶点表中的下标；域*link*存放指向 v 的边链表中结点 *VerAdj* 的下一个结点的指针。域*cost* 存放边(*v*, VerAdj)或<*v*, VerAdj>的权值；

VerName	*adjacent*



顶点表



顶点的结构

VerName	*adjacent*

非权图中边结点结构为

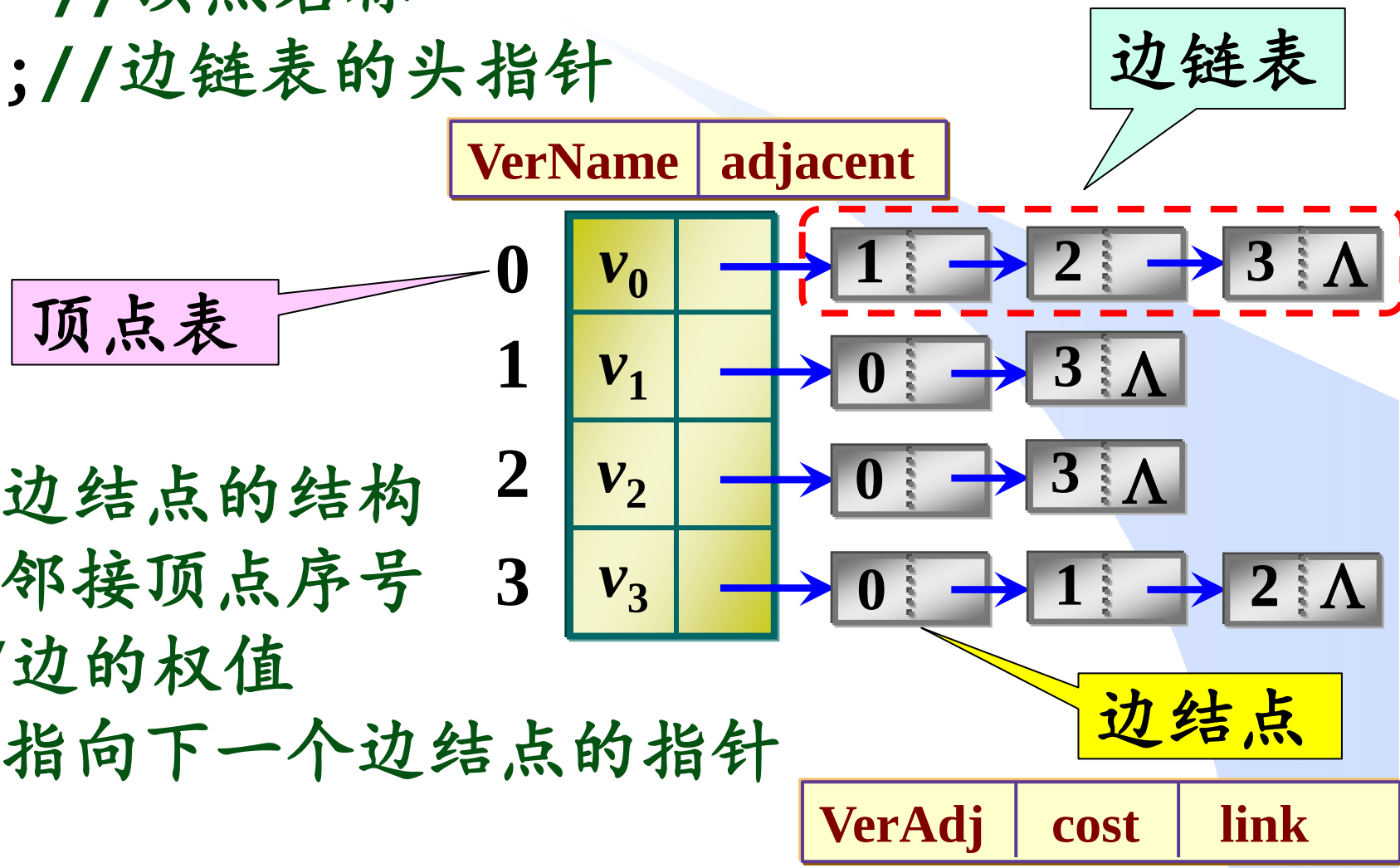
VerAdj	*link*

权图中边结点结构为

VerAdj	*cost*	*link*

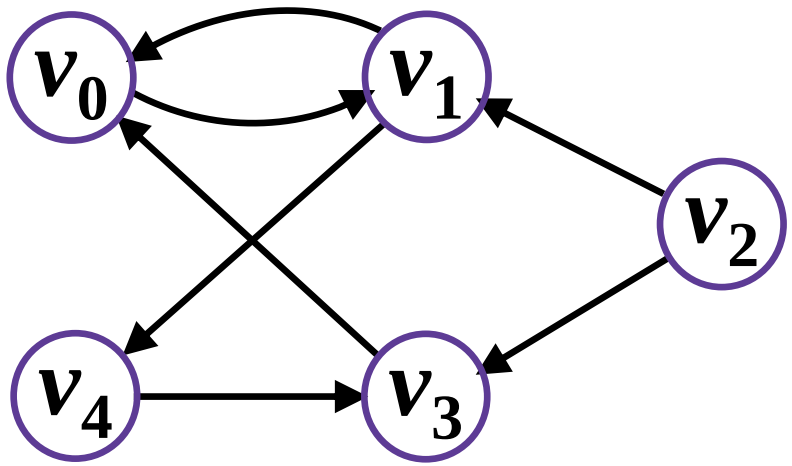
用邻接表存储图

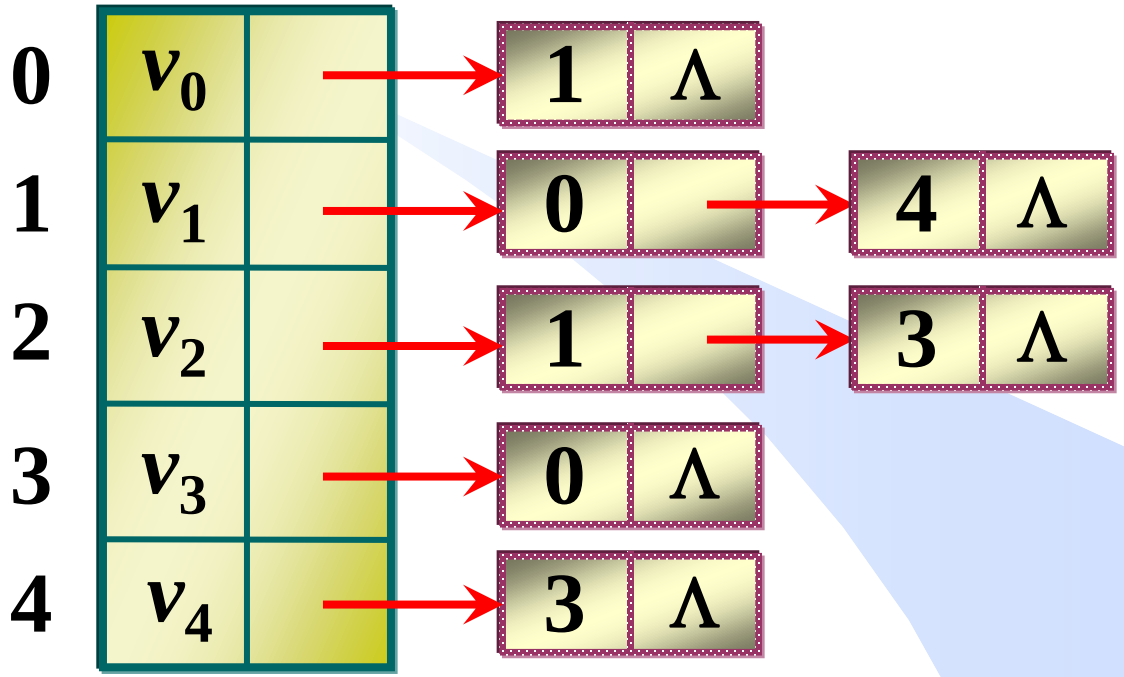
```
struct Vertex{     //顶点表中结点的结构
   int VerName;    //顶点名称
   Edge *adjacent;//边链表的头指针
};
```




```
struct Edge{   //边结点的结构
   int VerAdj; //邻接顶点序号
   int cost;   //边的权值
   Edge *link; //指向下一个边结点的指针
};
```

[例]有向图的邻接表



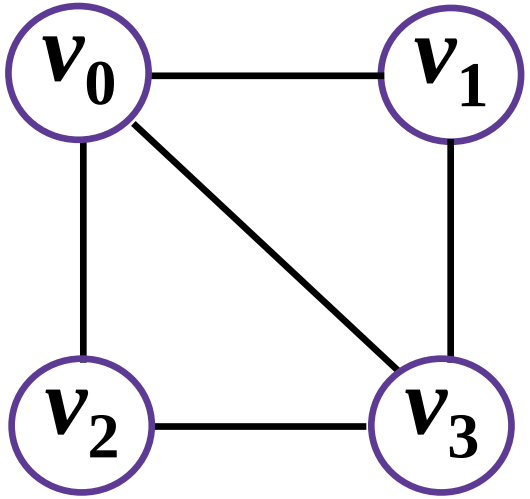


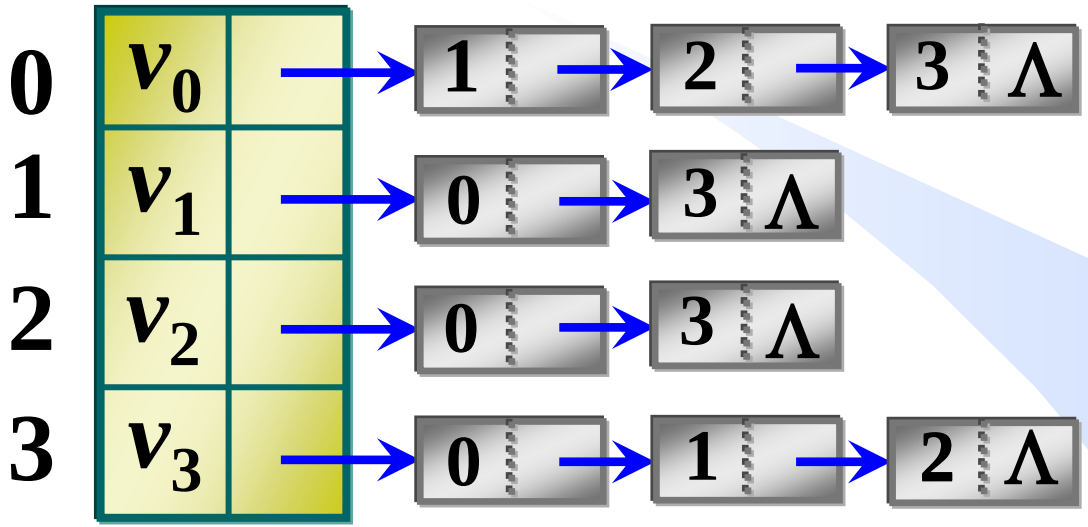
边结点的个数 = e

e为图中边的条数

1个边结点 ⇔ 1条边

[例]无向图的邻接表



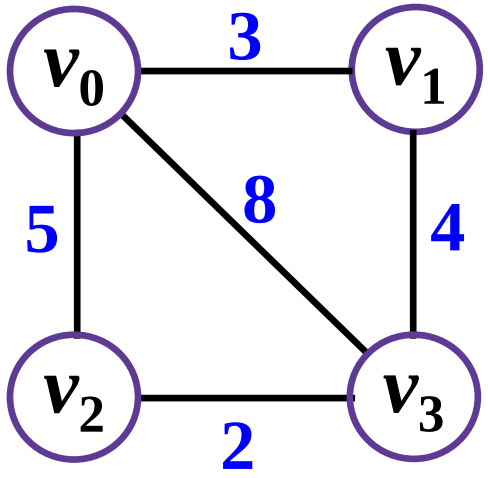


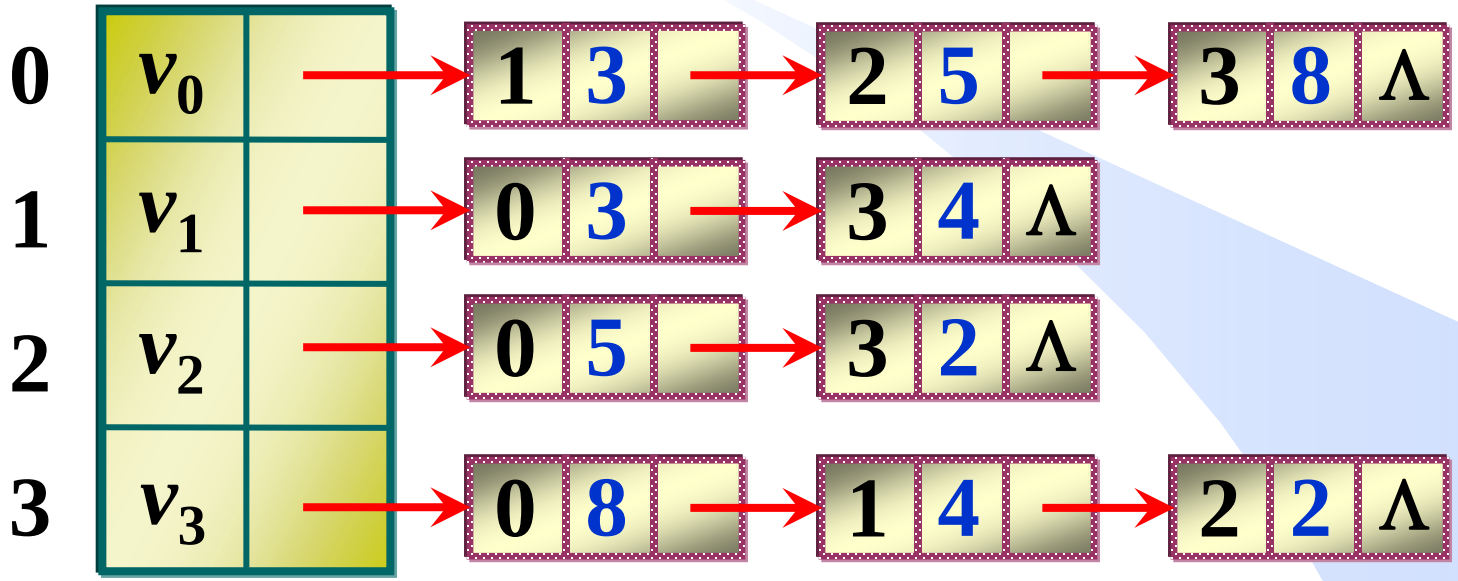
边结点的个数=2e

e为图中边的条数

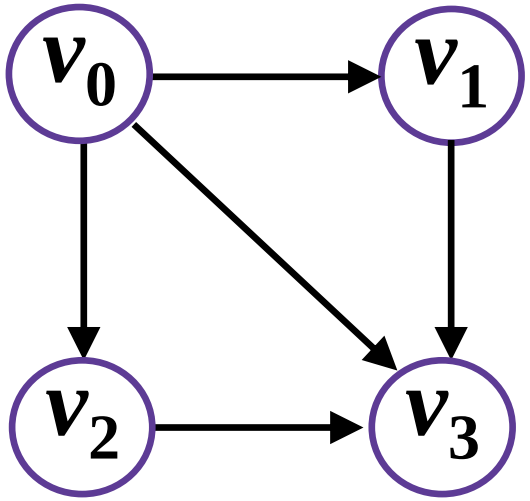
2个边结点 ⇔ 1条边

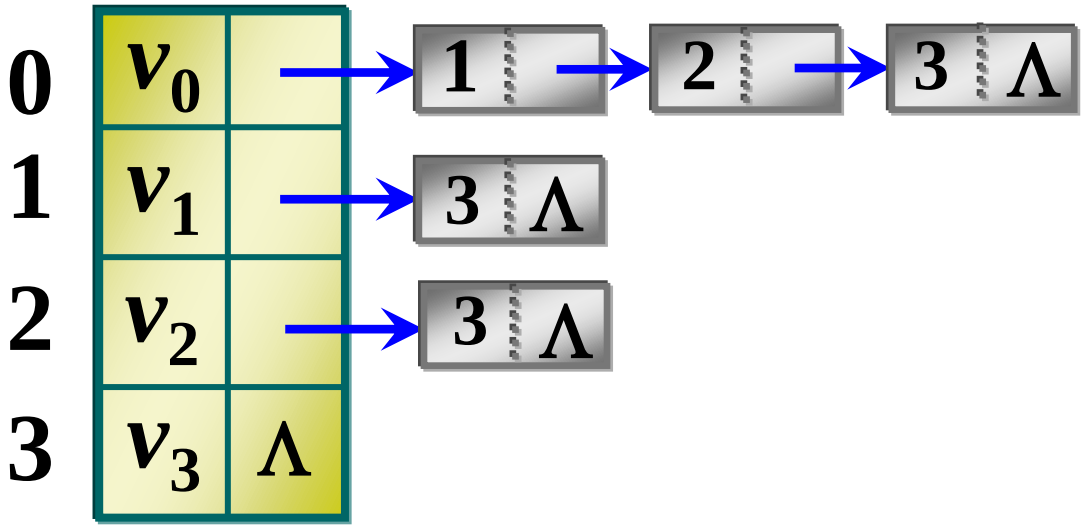
[例] 带权图的邻接表





链式前向星



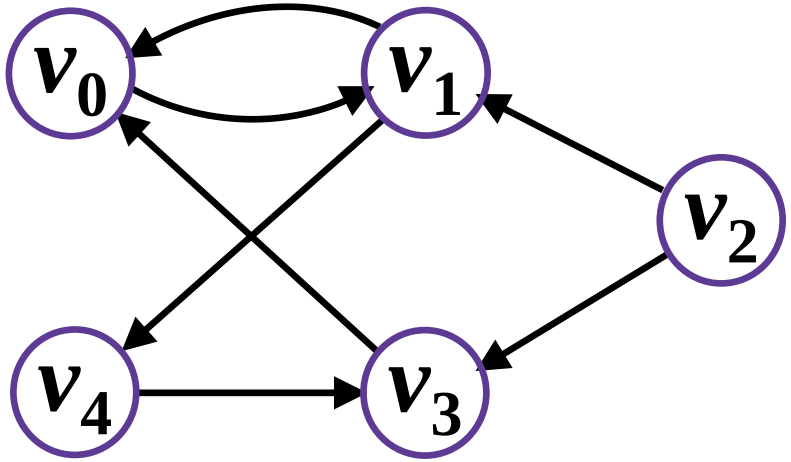


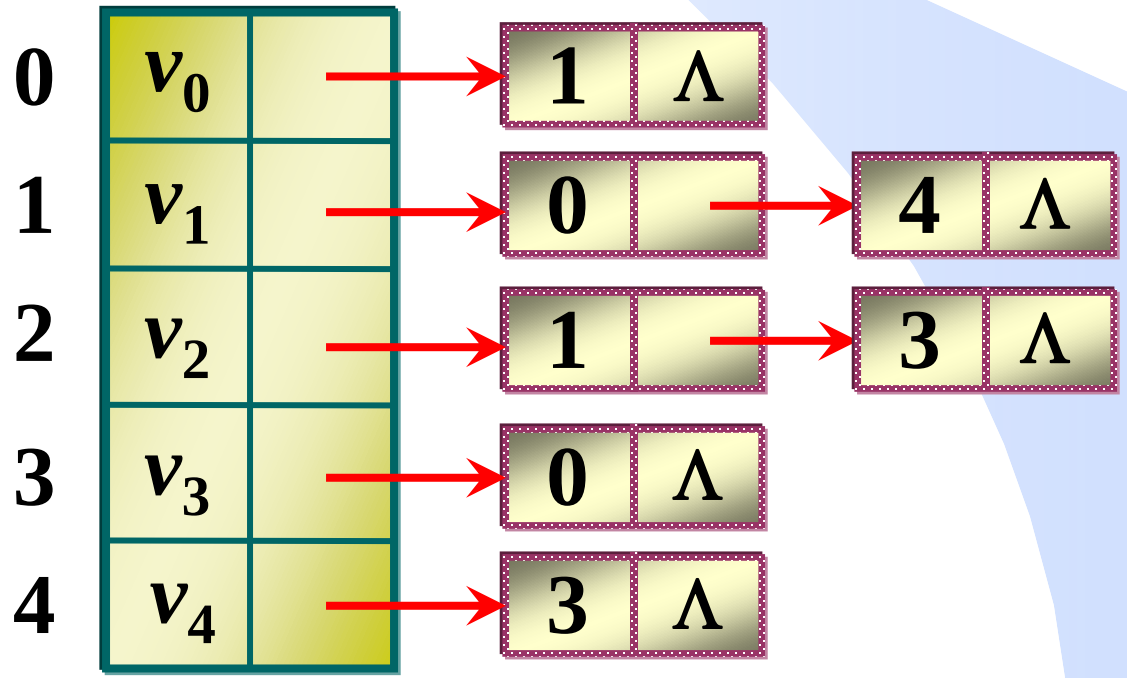
	head
0	0
1	3
2	4
3	-1

	to	next
0	1	1
1	2	2
2	3	-1
3	3	-1
4	3	-1

把边结点组织成静态链表
用数组下标模拟指针
程序设计竞赛中广泛使用

- 根据邻接表，可统计出有向图中每个顶点的出度。
- 但是，如果要统计一个顶点的入度，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）
- 有无更高效的方法？
  时间为$O(n+e)$?



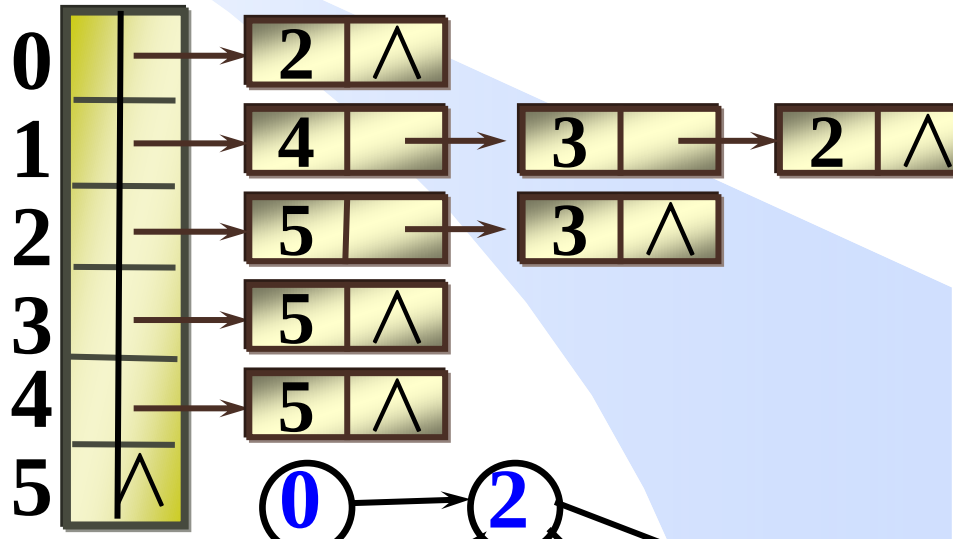


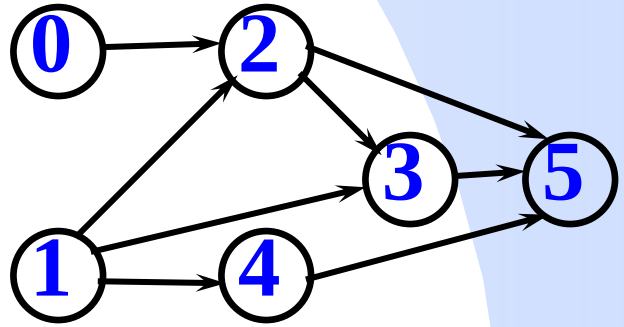
VerName	adjacent

VerAdj	link

```
void InDegree(Vertex Head[],int n, int count[]) {
    for(int i=0;i<n;i++) count[i]=0;
    for(int i=0;i<n;i++){//用i扫描每个顶点
        Edge* p=Head[i].adjacent;
        while(p!=NULL){
            int k=p->VerAdj;
            count[k]++;
            p = p->link;
        }
    }
}
```

用 *p* 扫描每个顶点对应的边结点（邻接顶点）

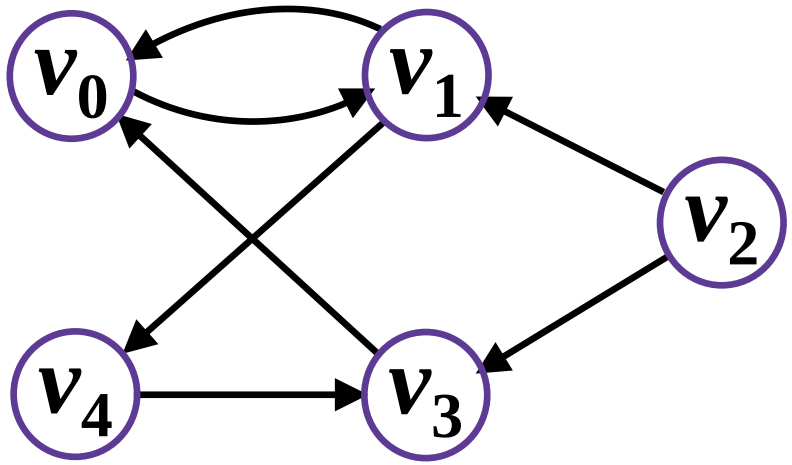




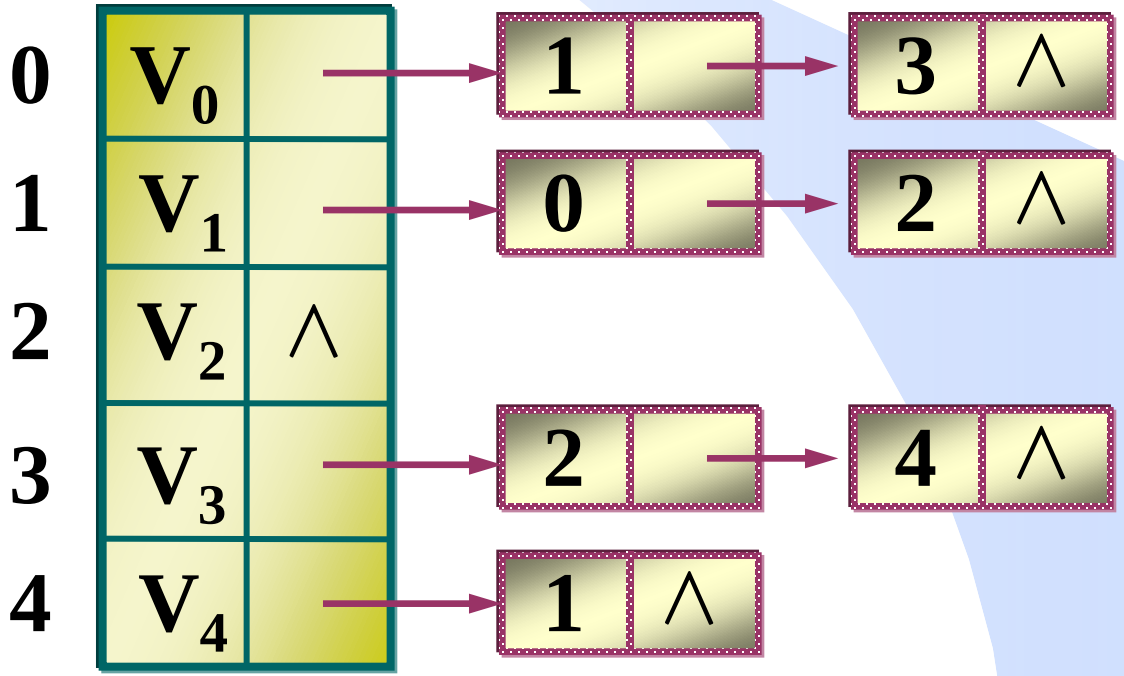
	0	1	2	3	4	5
count	0	0	2	2	1	3

```
for(Edge* p=Head[i].adjacent; p!=NULL; p=p->link )
```

有向图的逆邻接表



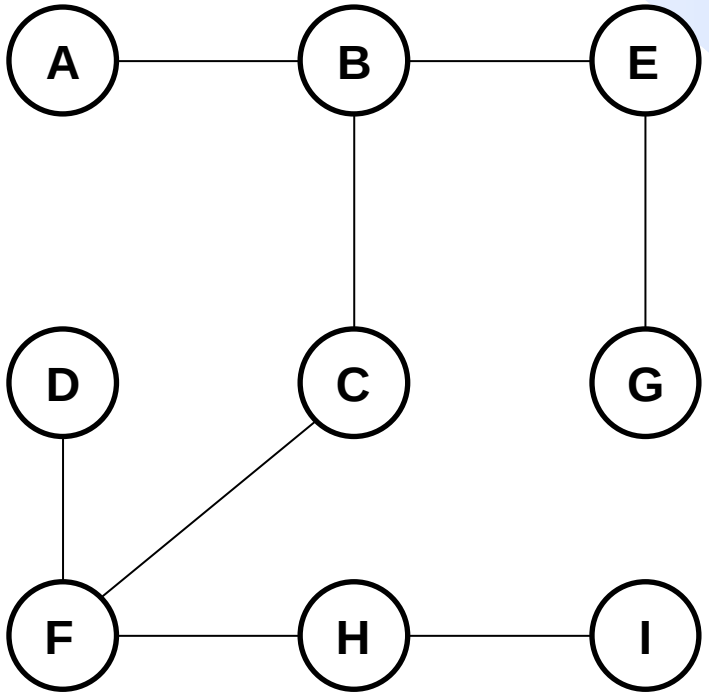
在有向图的逆邻接表中，第 i 个边链表链接的边都是指向顶点 i 的边。



对有向图建立逆邻接表（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。

无向图 vs. 有向图

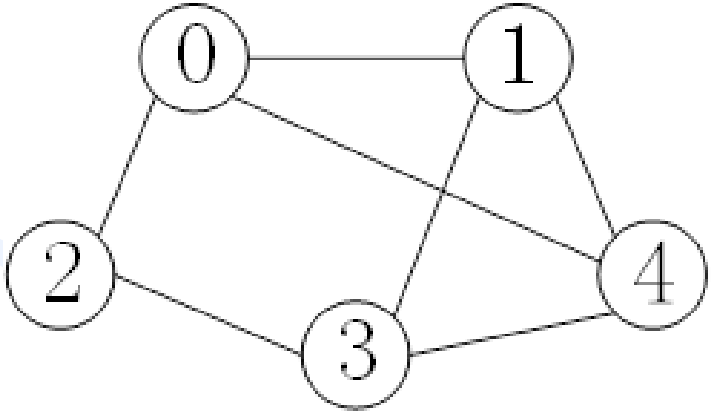
- 社交网络：好友关系
- QQ、微信：无向图
- 微博：有向图



邻接矩阵 vs. 邻接链表

- 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。
- 对边很多的图（也称稠密图），适于用邻接矩阵存储，因占用的空间少。另一好处是可以将图中很多运算转换成矩阵运算，方便计算。
- 对顶点多边少的图（也称稀疏图，如微信有几亿用户，每个用户好友一般三五百个），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。

已知有5个顶点的图G如右图所示，请回答下列问题：【考研题全国卷】



（1）写出图G的邻接矩阵A；

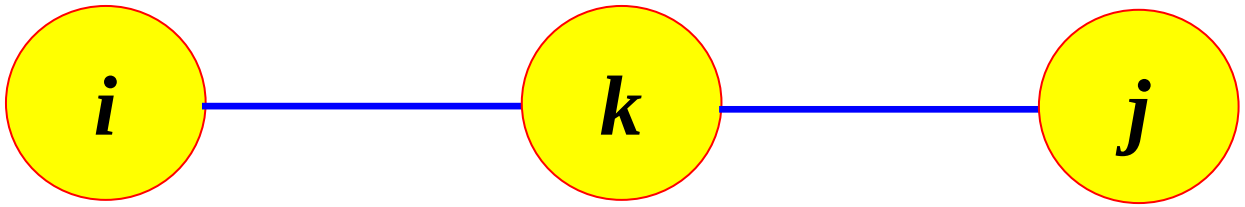
（2）求A^2，矩阵元素$A^2_{i,j}$的含义是什么；

顶点V_i到V_j的长度为2的路径条数。

（3）A^m（$m \le n$，n为图中顶点个数）非零元素的含义是什么。

顶点V_i到V_j的长度为m的路径条数。

$$A^2_{i,j} = \sum_{k=0}^{n-1} A_{i,k} \times A_{k,j}$$



例题

已知无向图G有16条边，其中度为4的顶点有3个，度为3的顶点有4个，其他顶点的度均小于3，则图G所含的顶点个数至少是_____。【考研题全国卷】

A. 10　　B. 11　　C. 13　　D.15

所有顶点度之和=2e，设顶点数为x

$$32 \quad \leq \quad 4*3+3*4+2*(x-7)$$

$x \geq 11$, 故选B

例题

- 具有n个顶点的无向图，当至少有___条边时可确保它一定是一个连通图。【吉林大学考研题】

答：n-1个顶点的完全图再加1条边，$C^2_{n-1}+1$，
故(n-1)(n-2)/2+1

- 无向图G包含7个顶点，要保证G在任何情况下都是连通的，则需要边数最少是___条。【考研题全国卷】

A. 6　　B. 15　　C. 16　　D. 21

例题

G是一个具有36条边的无向非连通图，则G的顶点数至少为_____。【北京大学、吉林大学考研题】

A. 12　　B. 11　　C. 10　　D. 9

无向图有n个顶点，则边的个数 $\leq C^2_n = n(n-1)/2$，即 $36 \leq n(n-1)/2$，即 $72 \leq n(n-1)$，得 $n \geq 9$。对于连通图，n至少为9，对于非连通图，n至少为10。

故选C

课下思考

对于无向图G =（V,E），下列选项中正确的是_____。
【2022年考研题全国卷，2分】

A. 当|V| > |E|时，G一定是连通的

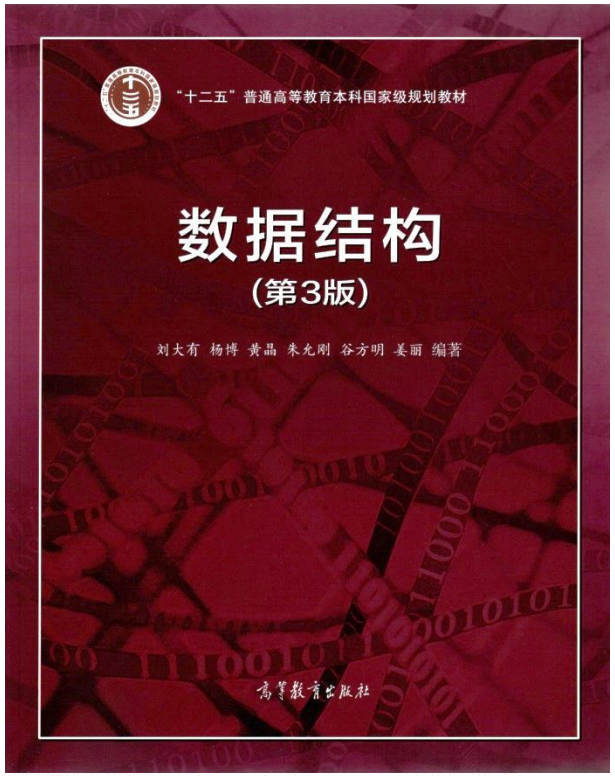
B. 当|V| < |E|时，G一定是连通的

C. 当|V| = |E| - 1时，G一定是不连通的

D. 当|V| > |E| + 1时，G一定是不连通的

课下思考

已知无向连通图G由顶点集V和边集E组成，|E|>0，当G中度为奇数的顶点个数为0或2时，G存在包含所有边且长度为|E|的路径（称为EL路径），设G采用邻接矩阵存储，请设计算法，判断G是否存在EL路径。【2021年考研题全国卷】



图的存储结构与遍历

- 图的概念简要回顾
- 图的存储结构
- **图的遍历**
- 图遍历的应用

数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

Last updated on 2022.11

图的遍历

- 从图的某顶点出发，访问图中所有顶点，且使每个顶点恰被访问一次的过程被称为图遍历。
- 图中可能存在回路，且图的任一顶点都可能与其它顶点连通，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，可用一个辅助数组*vis*[ ]记录顶点是否被访问过，数组各元素初始值为0。在遍历过程中，一旦某一个顶点 *i* 被访问，就置 *vis*[*i*] 为 1。

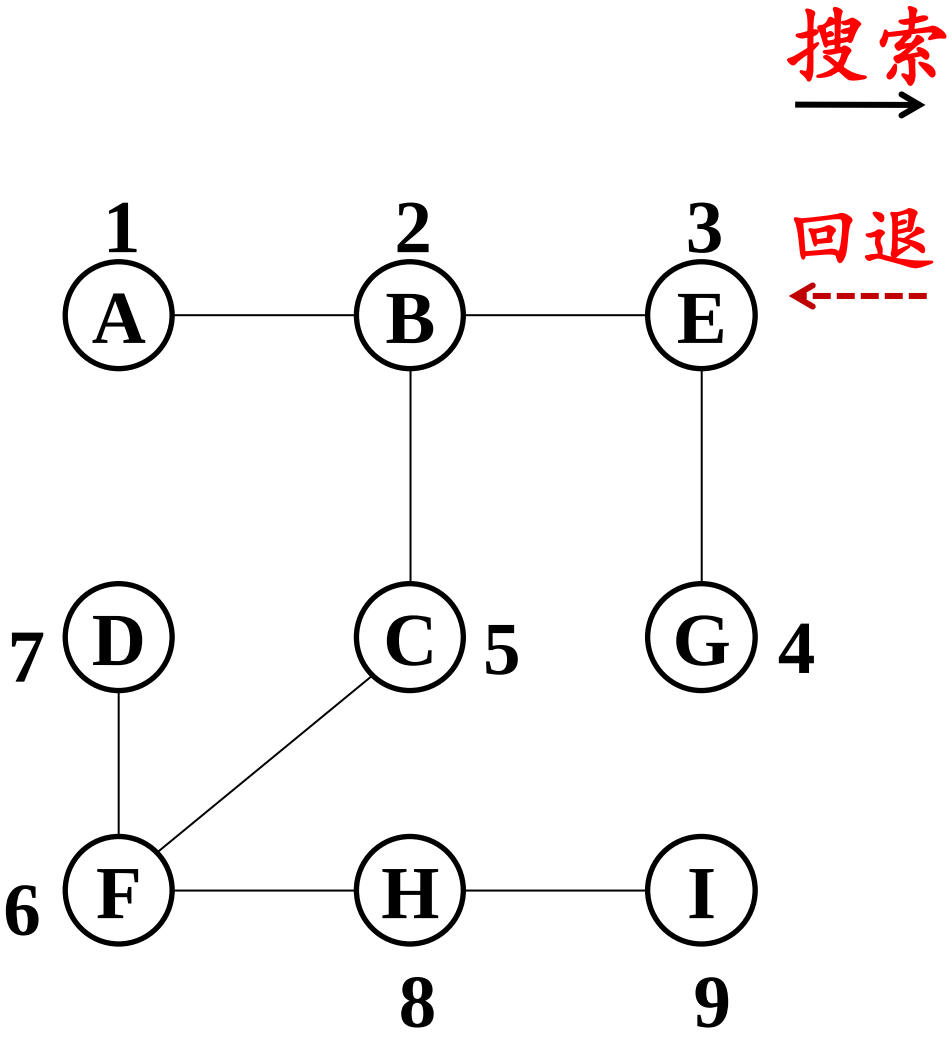
深度优先遍历/搜索 DFS (Depth First Search, DFS)

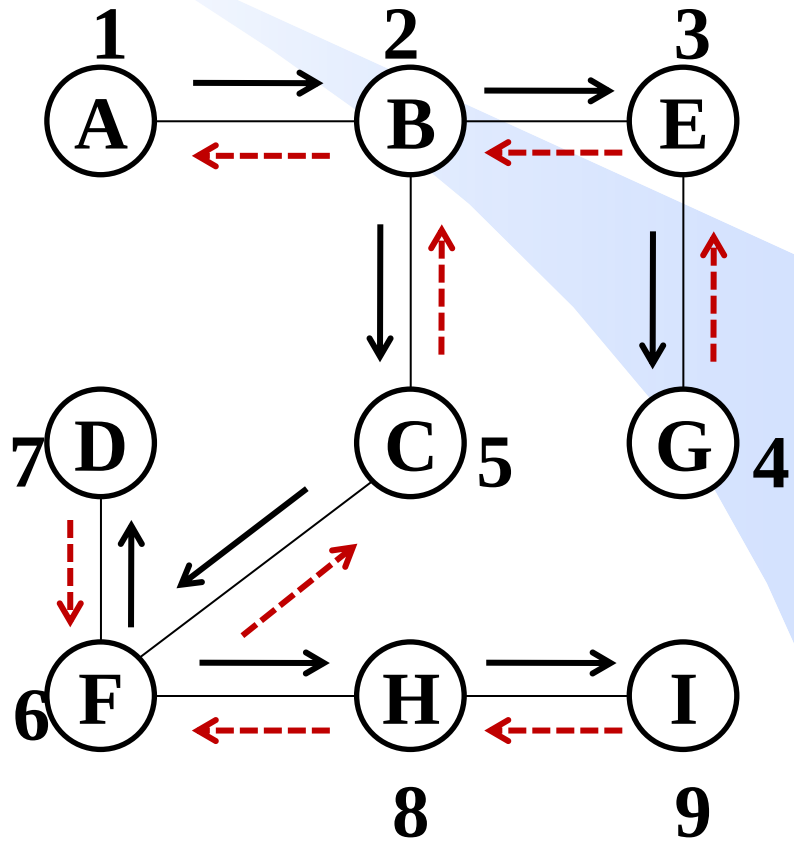
基本思想:

- *DFS* 在访问图中某一起始顶点 v_0 后，由 v_0 出发，访问它的任一邻接顶点 v_1；再从 v_1 出发，访问v_1的一个未曾访问过的邻接顶点 v_2；然后再从 v_2 出发，进行类似的访问，… 如此进行下去，直至到达一个顶点，它不再有未访问的邻接顶点。
- 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，直到图中所有顶点都被访问过为止。

图深度优先遍历的次序不唯一

深度优先遍历/搜索 DFS (Depth First Search, DFS)

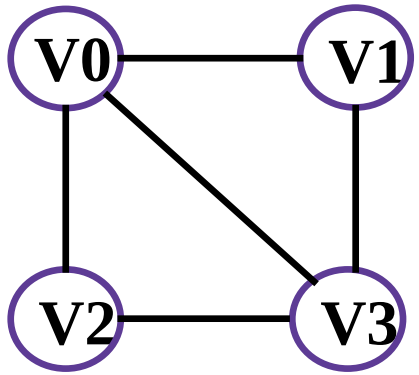


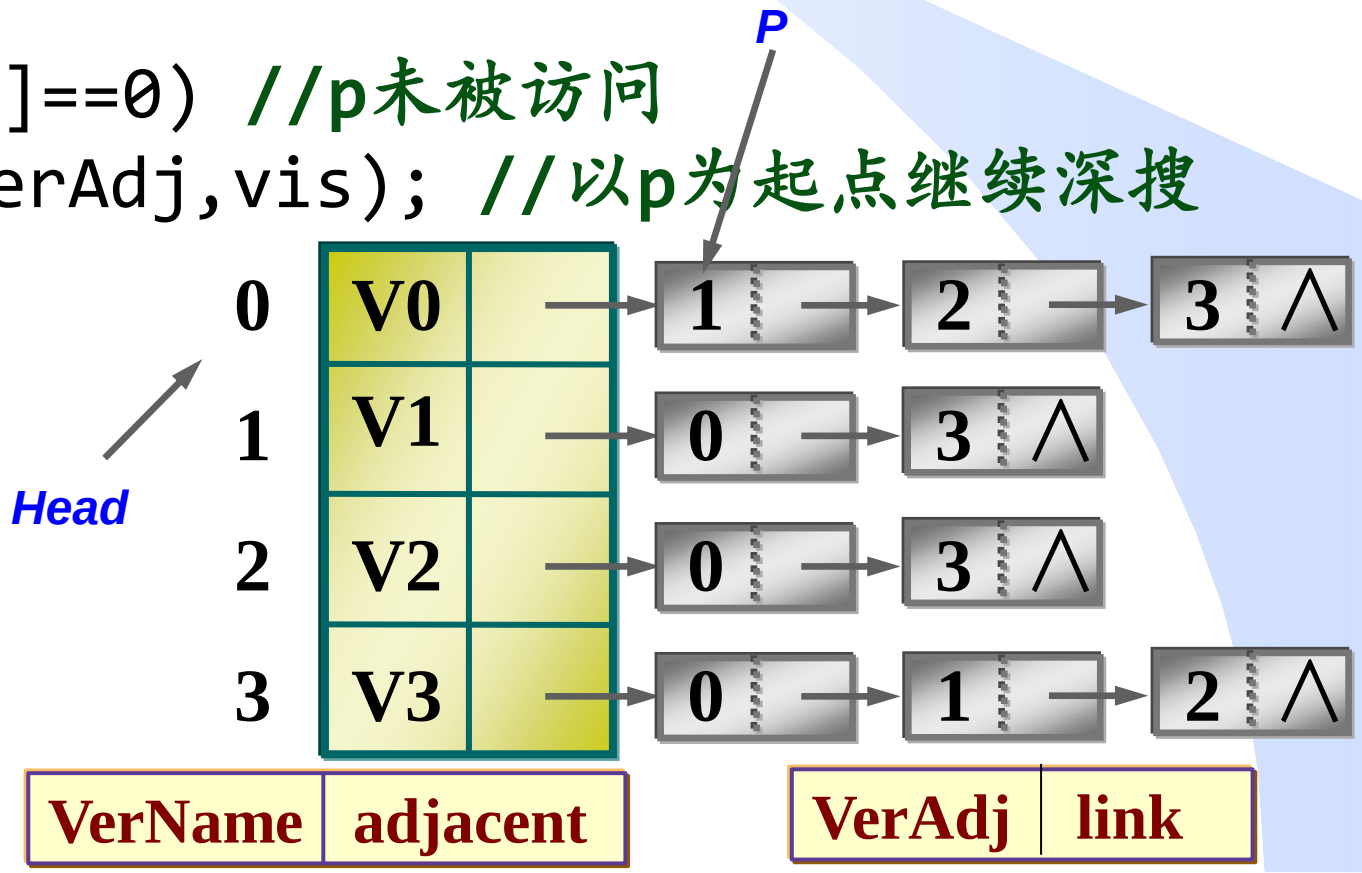


递归算法

回溯：可行则进，不行则换，换不了则退

```
void DFS(Vertex*Head, int v, int vis[]){ //vis初值为0
    //以v为起点进行深度优先搜索
    printf("%d ",v); vis[v]=1;   //访问顶点v
    Edge* p= Head[v].adjacent;   //考察v的所有邻接顶点
    while(p!=NULL){
        if(vis[p->VerAdj]==0) //p未被访问
            DFS(Head,p->VerAdj,vis); //以p为起点继续深搜
        p=p->link;
    }
}
```





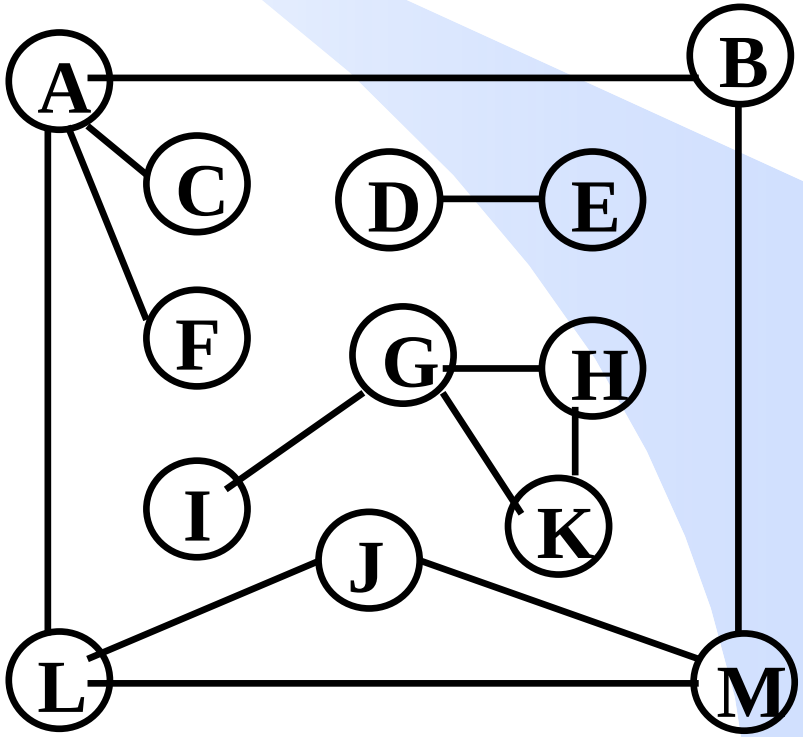
递归算法

```
void DFS(Vertex* Head, int v, int vis[]){
    //以v为起点进行深度优先搜索，vis初值为0
    printf("%d ",v); vis[v]=1;     //访问顶点v
    for(Edge* p=Head[v].adjacent;p!=NULL;p=p->link)
            //通过p循环扫描v的所有邻接顶点
            if(vis[p->VerAdj]==0) //考察v的邻接顶点p
                DFS(Head,p->VerAdj,vis);
}
```

一次DFS只能遍历一个连通分量

非连通图需要多次调用DFS算法

```
for(int i=0;i<n;i++) //数组初始化
    vis[i] = 0;
//以每个顶点为起点，试探是否能深搜
for(int i=0;i<n;i++)
   if(vis[i]==0)
      DFS(Head, i, vis);
```



图DFS的非递归算法

顶点弹栈时访问

① 将所有顶点的vis[ ]值置为0, 初始顶点v_0入栈；

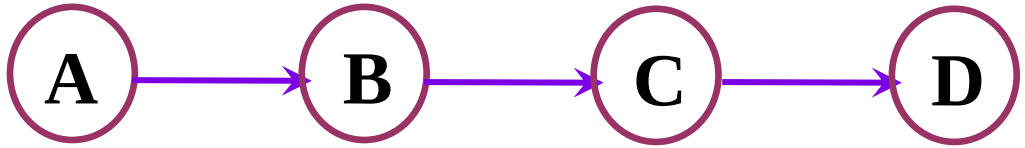
② 检测栈是否为空，若栈为空，则算法结束；

③ 从栈顶弹出一个顶点v，如果v未被访问过则：

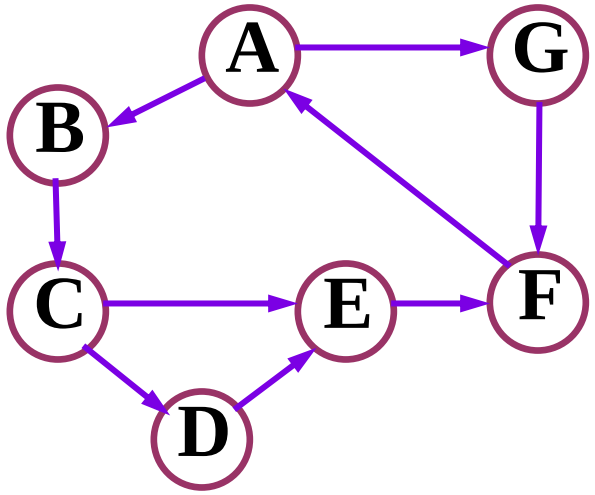
访问v，并将vis[v]值更新为1；

将v的未被访问的邻接顶点入栈；

④ 执行步骤②。



顶点弹栈时访问



- 基本思想：

① 将所有顶点的visited[]值置为0, 初始顶点v_0入栈;

② 检测堆栈是否为空，若堆栈为空，则迭代结束;

③ 从栈顶弹出一个顶点v，如果v未被访问过则:

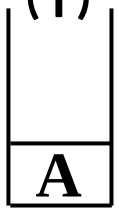
访问v，并将visited[v]值更新为1;

将v的未被访问的邻居入栈;

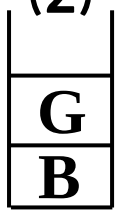
④ 执行步骤②。

顶点进栈时已确保未被访问，出栈时为何还判断是否被访问过？

(1)

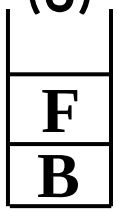


(2)



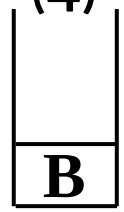
A 访问过

(3)



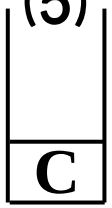
AG 访问过

(4)



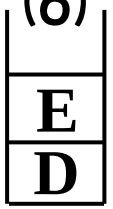
AGF 访问过

(5)



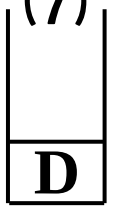
AGFB 访问过

(6)



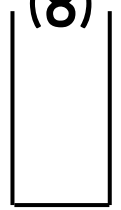
AGFBC 访问过

(7)

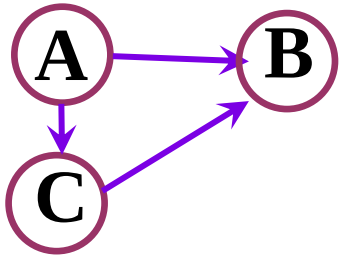


AGFBCE 访问过

(8)



AGFBCED 访问过



- 基本思想：

① 将所有顶点的visited[]值置为0, 初始顶点v_0入栈;

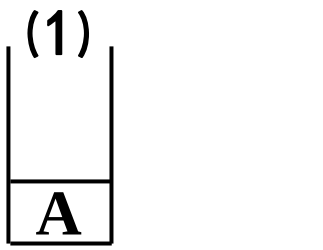
② 检测堆栈是否为空，若堆栈为空，则迭代结束;

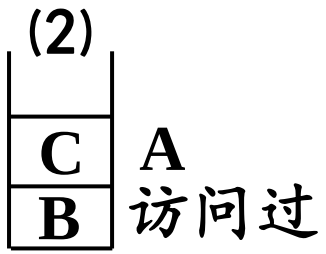
③ 从栈顶弹出一个顶点v，如果v未被访问过则:

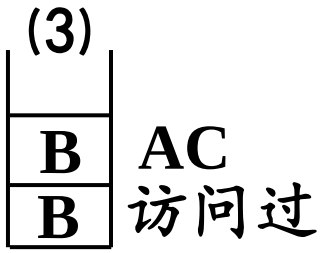
访问v，并将visited[v]值更新为1;

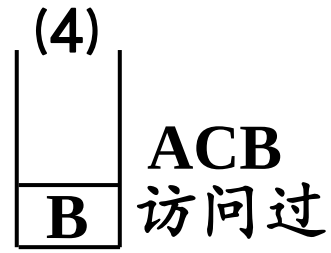
将v的未被访问的邻居入栈;

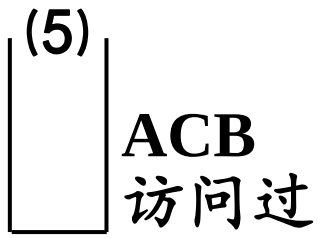
④ 执行步骤②。



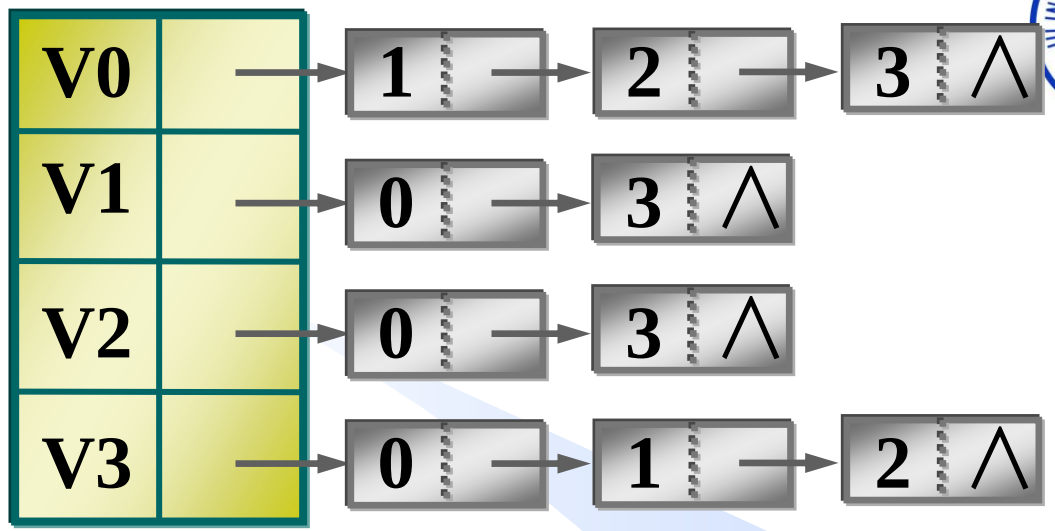








```
算法DFS (Head, v, n, vis)
/* 图的深度优先遍历的非递归算法*/
   CREATS(S) .  //创建栈 S
   FOR i = 0 TO n-1 DO  vis[i] ← 0.
   S⇐ v.  // 将v压入栈中
   WHILE NOT EMPTY(S) DO
   (    v ⇐ S.
        IF vis[v] = 0 THEN
         (      PRINT(v) .
                vis[v] ← 1.
                p ← adjacent(Head[v]) .
                WHILE p≠Λ DO   //找v的未被访问的邻接顶点压栈
               (      IF vis[VerAdj(p)] = 0 THEN S ⇐ VerAdj(p) .
                      p ← link(p).
               )
         )
   ) ▌
```



while循环本质：扫描每个顶点的边结点，即图中所有边结点，故时间O(*e*)

每个顶点被访问一次，共*n*个顶点，故时间O(*n*)

时间复杂度为O(*n*+*e*)

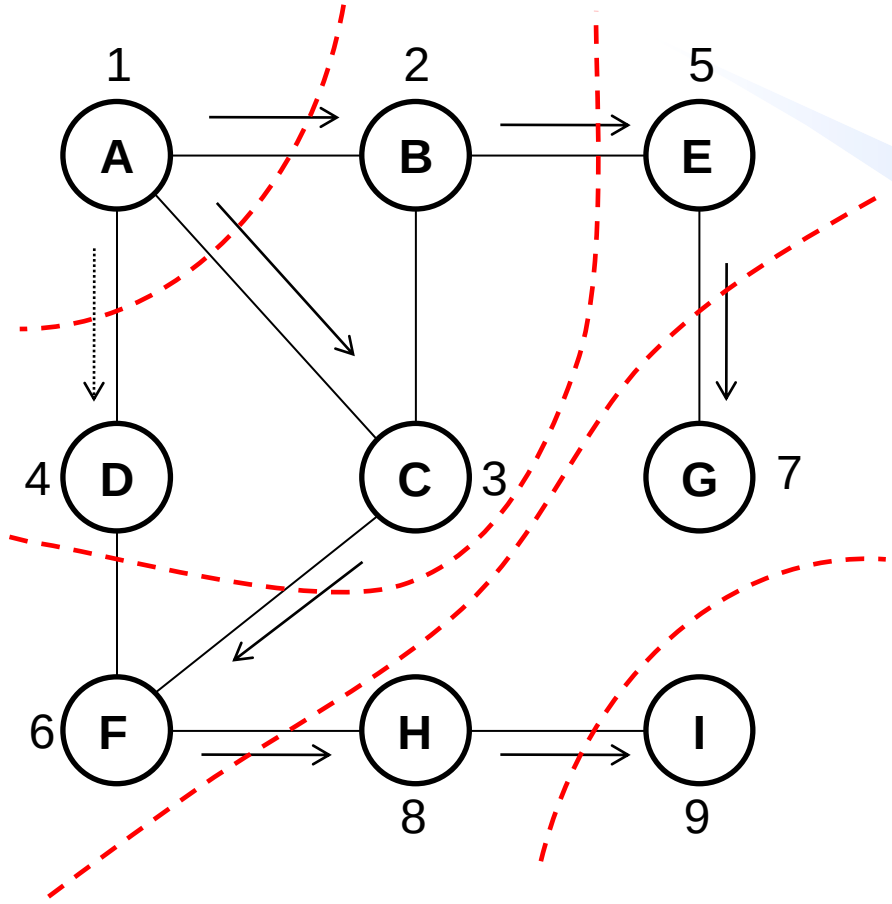
广度优先遍历

基本思想：

- ✓ 访问初始点顶点v_0；
- ✓ 依次访问v_0的邻接顶点w_1，w_2，…，w_k；
- ✓ 然后，再依次访问与w_1，w_2，…，w_k邻接的尚未访问的顶点；
- ✓ 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……
- ✓ 依此往复，直至连通图中的所有顶点全部访问完。

广度优先搜索 (Breadth First Search, *BFS*)

地毯式搜索
层层推进



广度优先搜索 (Breadth First Search, *BFS*)

- 图的广度优先遍历类似于树的层次遍历，是一种分层的搜索过程，每前进一层可能访问一批顶点，不像深度优先搜索那样有时需要回溯。因此，广度优先搜索不是一个递归过程，也不需要递归算法。
- 为了实现逐层访问，算法中使用一个队列，记忆还未被处理的顶点，用以确定正确的访问次序。
- 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 *vis*[ ]，标识一个顶点是否被访问过。

广度优先搜索 (Breadth First Search, *BFS*)

基本思想:

① 将所有顶点的vis[ ]值置为0，访问起点v，置vis[*v*]=1，*v*入队；

② 检测队列是否为空，若队列为空，则算法结束；

③ 出队一个顶点*v*，考察其每个邻接顶点*w*:

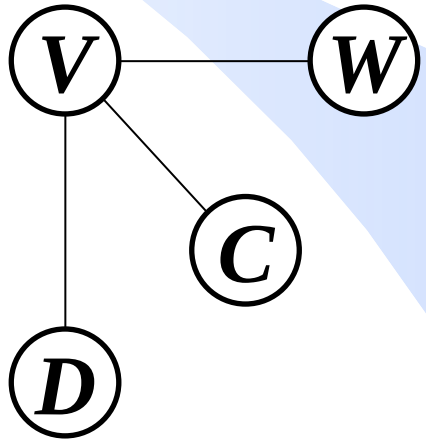
如果*w*未被访问过，则

访问*w*;

将vis[*w*]值更新为1;

将*w*入队;

④ 执行步骤②。



顶点入队时访问

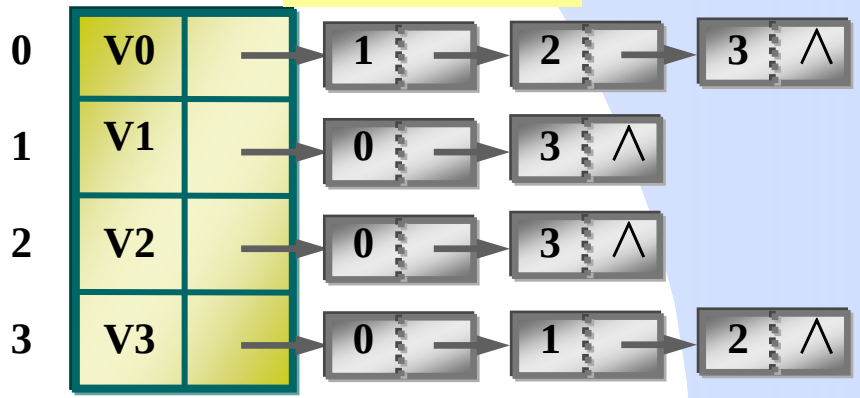
```
void BFS(Vertex* Head, int v, int n, int vis[])
    Queue Q;  //创建队列Q, 队列需预先实现
    for(int i=0; i<n; i++) vis[i]=0;
    printf("%d ",v); vis[v]=1;//访问点v
    Q.Enqueue(v);              //起点v入队
    while(!Q.Empty()){
        v=Q.Dequeue();     //出队一个点
        for(Edge* p=Head[v].adjacent; p!=NULL; p=p->link)
            if(vis[p->VerAdj]==0){ //考察v的邻接顶点p
                printf("%d ", p->VerAdj);
                vis[p->VerAdj] = 1;
                Q.Enqueue(p->VerAdj);
            } //end if
    } //end while
} //end BFS
```

while循环本质：扫描每个顶点的边结点，即图中所有边结点，故时间O(*e*)

每个顶点被访问一次，共*n*个顶点，故时间O(*n*)

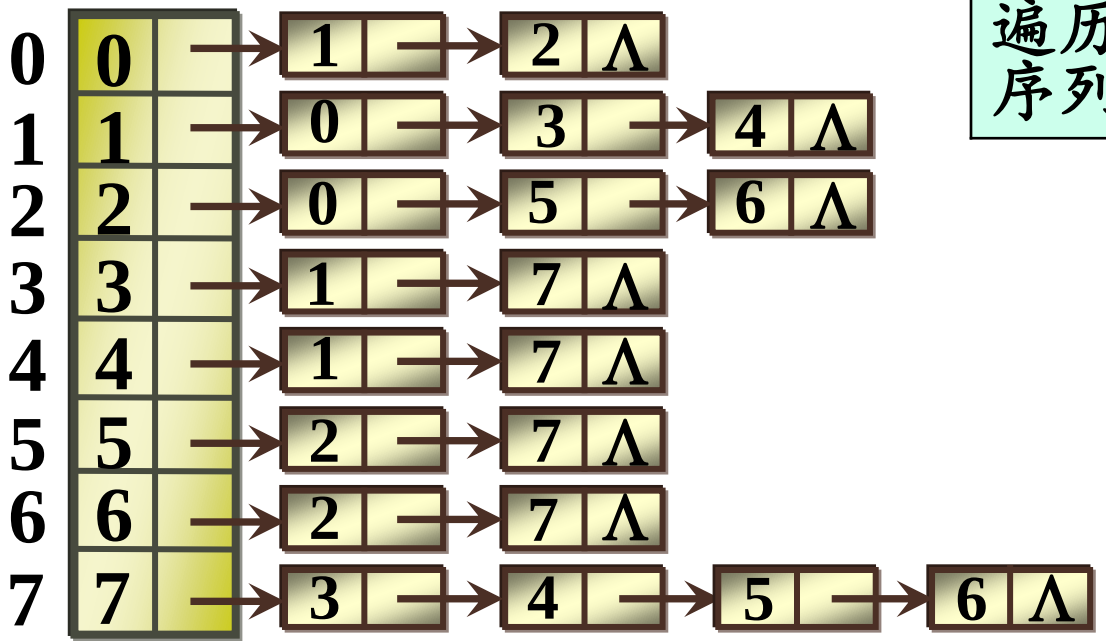
时间复杂度为O(*n*+*e*)

邻接表

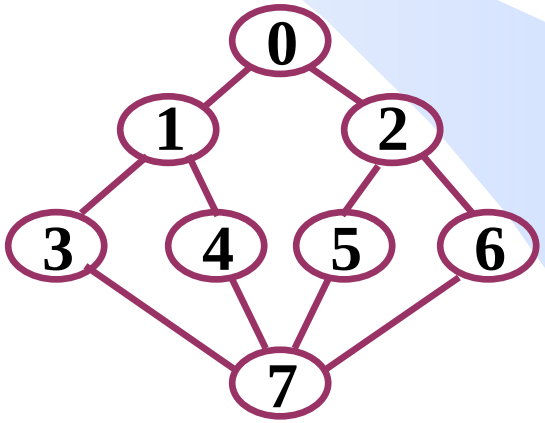


顶点入队时访问

```
while(!Q.Empty()){
    v=Q.Dequeue();  //出队一个点
    for(Edge* p=Head[v].adjacent; p!=NULL; p=p->link)
        if(vis[p->VerAdj]==0){ //考察v的邻接顶点p
            printf("%d ", p->VerAdj); vis[p->VerAdj]=1;Q.Enqueue(p->VerAdj);
        }
}
```



遍历序列	0	1	2	3	4	5	6	7



队列 Q 的变化，上面是队头，下面是队尾。

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

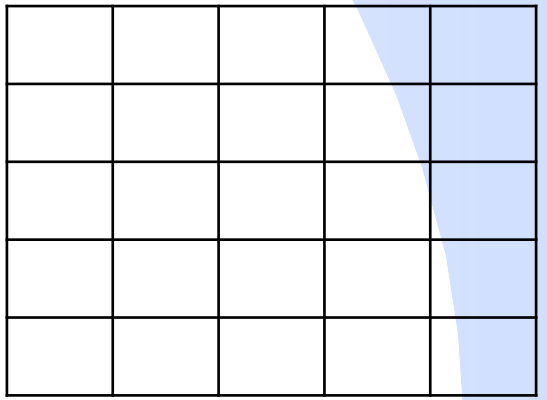
```
void BFS(int** A, int v, int n, int vis[])//邻接矩阵存图
    Queue Q;  //创建队列Q
    for(int i=0; i<n; i++) vis[i]=0;
    printf("%d ",v); vis[v]=1;//访问点v
    Q.Enqueue(v);  //起点v入队
    while(!Q.Empty()){
        v=Q.Dequeue();    //出队一个点
        for(int i=0; i<n; i++)
            if(A[v][i]==1 && vis[i]==0){ //考察v的邻接顶点
                printf("%d ", i);
                vis[i] = 1;
                Q.Enqueue(i);
            } //end if
    } //end while
} //end BFS
```

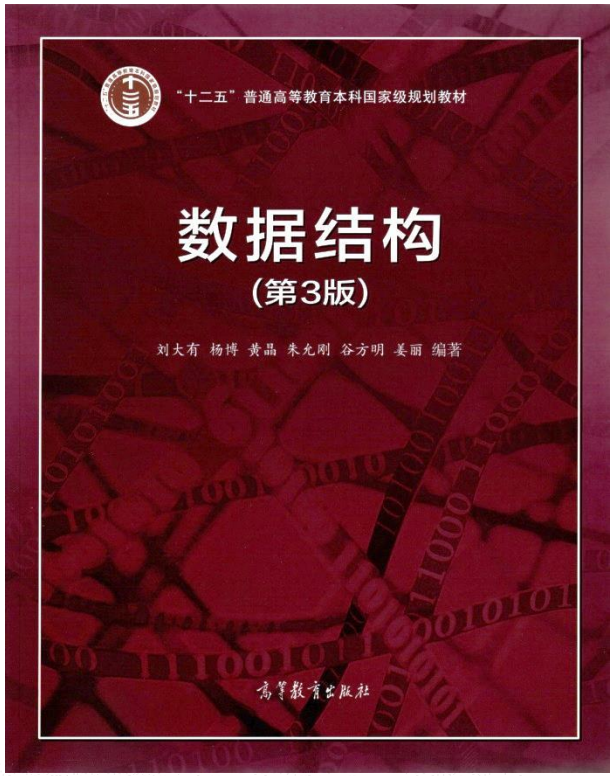
while循环扫描所有n个顶点的邻接顶点，时间$O(n^2)$

for循环：扫描顶点 i 的邻接顶点需遍历矩阵第 i 行，时间$O(n)$

时间复杂度为$O(n^2)$

邻接矩阵





图的存储结构与遍历

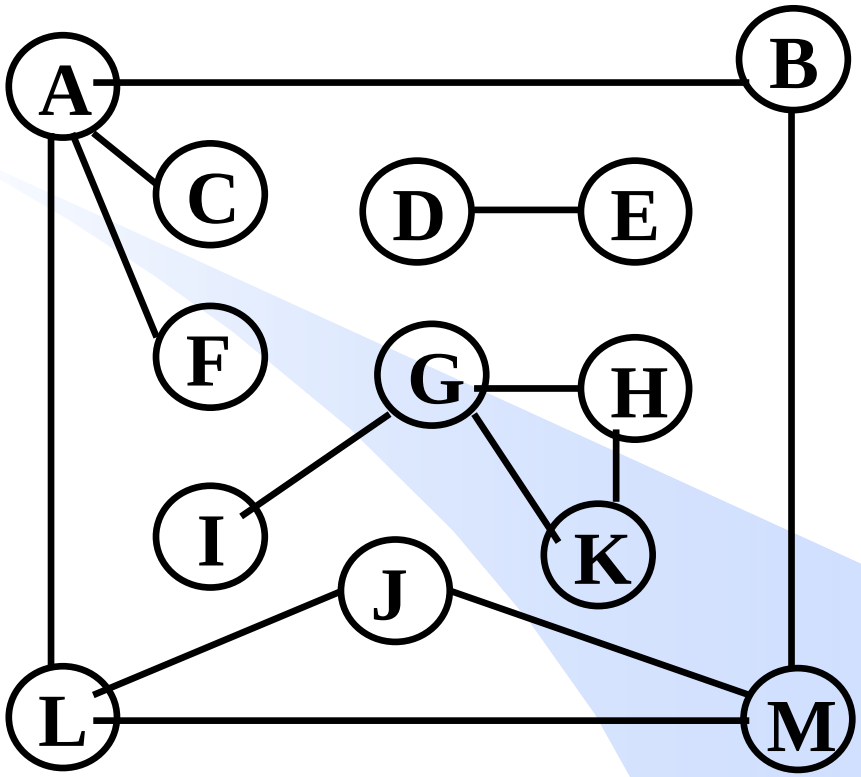
- 图的概念简要回顾
- 图的存储结构
- 图的遍历
- **图遍历的应用举例**

数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

Last updated on 2022.11

判断无向图是否连通及连通分量数目【谷歌、微软、苹果、英特尔面试题】



方案1：DFS

```
for(int i=0;i<n;i++)vis[i]=0;
int cnt=0; //连通分量数目计数器
for(int i=0;i<n;i++)
    if(vis[i]==0){
        DFS(Head, i, vis);
        cnt++; //每遍历一个连通分量，计数器加1
    }
```

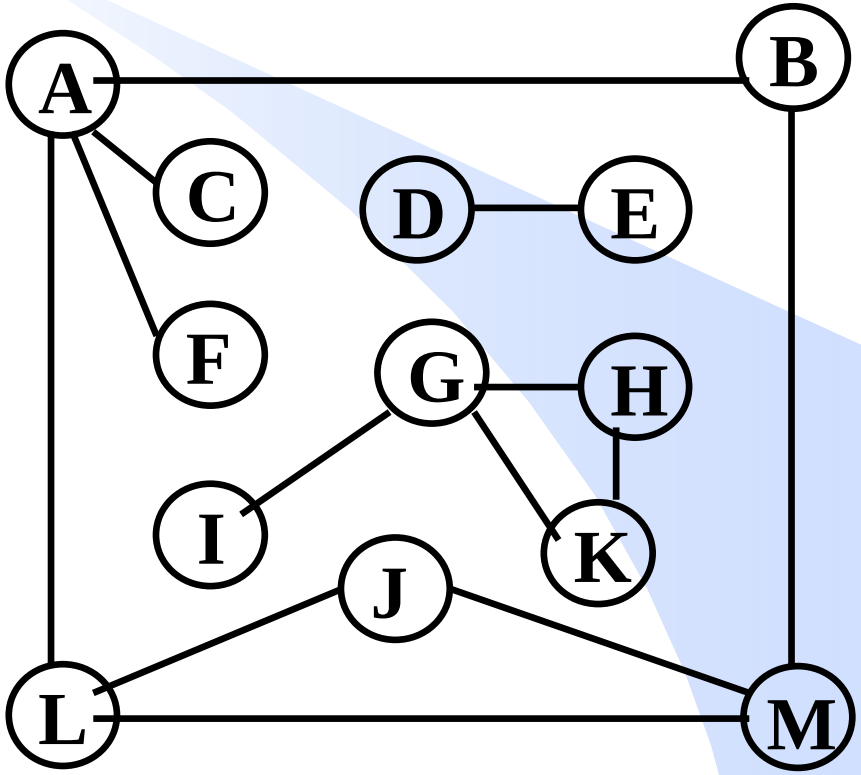
每调用一次DFS，遍历一个连通分量
cnt等于1即为连通图

判断无向图是否连通及连通分量数目【谷歌、微软、苹果、英特尔面试题】

方案2：并查集

连通分量⇔集合

每次读入一条边(x,y)，Union(x,y);
扫描完所有边后：集合数即连通分量数



判断图中顶点u到v是否存在路径【华为、谷歌、微软、苹果、亚马逊面试题】

- 以u为起点遍历，看遍历过程中是否经过v

```
bool DFS(Vertex* Head, int u, int v, int vis[]){
    vis[u]=1;  //访问顶点u
    if(u==v) return true;
    for(Edge* p=Head[v].adjacent; p!=NULL; p=p->link)
        if(vis[p->VerAdj]==0) //考察v的邻接顶点
            if(DFS(Head,p->VerAdj,v,vis)==true) return true;
    return false;
}
```

课下思考

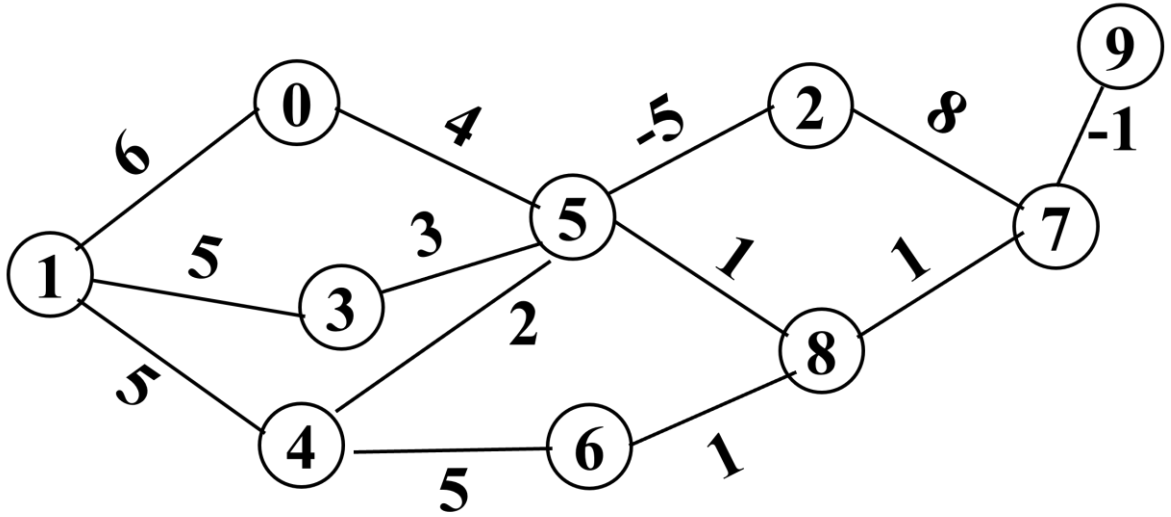
- 给定n个顶点的有向无环图，输出顶点0到顶点n-1的所有路径
  【字节跳动、阿里、谷歌、微软、苹果面试题】

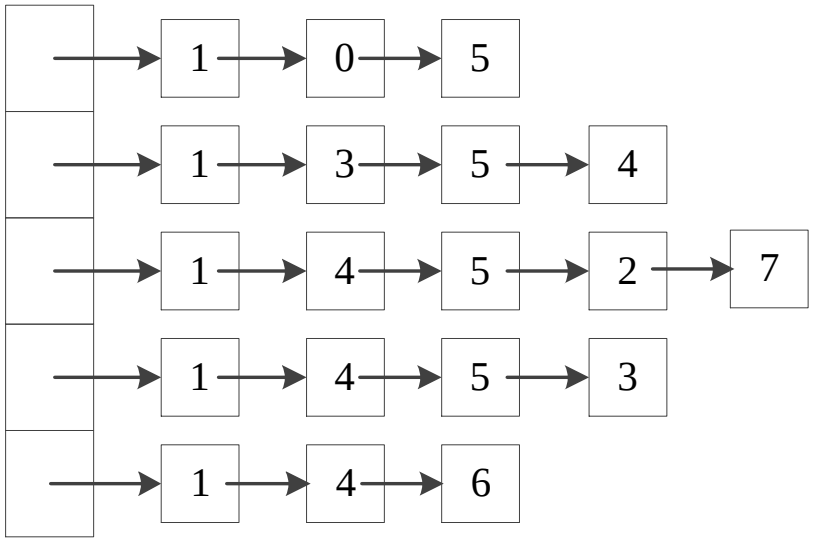
一个带权图G以邻接表作为存储结构，顶点编号为0至n-1，边权值可正可负。请设计一个算法，找出从指定顶点u出发的、不经过顶点$v(v \neq u)$的、包含总顶点数不超过m ($m \geq 2$)的、长度等于L的所有简单路径，并将每条路径保存在一个单链表中，所有单链表的头指针存入一个指针数组。即设计实现如下函数：

```
int FindPath(Vertex Head[ ], int n, int u, int v, int L, int m, ListNode* PathList[ ])
```

其中Head为图G的顶点表，n、u、v、L、m含义如题目所述，PathList为存储各路径链表头指针的数组，函数返回值为满足条件的路径条数。

例如对于左下图，若u=1, v=8, L=10, m=5，则算法得到右下图所示的结果。





课下思考

（1）给出算法的基本设计思想。

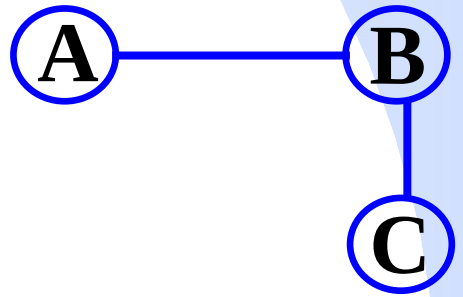
（2）根据设计思想，采用ADL、C/C++、Java等语言描述算法，关键之处给出注释。

（3）给出算法的时间复杂度。【2020级期末考试题，15分】

判断无向图中是否有环

（1）深度优先遍历，遍历过程中遇到之前访问过的顶点（不能是当前访问点的前驱），即有环

```
bool DFS(Vertex* Head, int v, int vis[], int pre){
    //以v为起点DFS，pre为v之前访问的结点
    vis[v]=1;  //访问顶点v
    for(Edge* p=Head[v].adjacent;p!=NULL;p=p->link)
       if(vis[p->VerAdj]==0) //考察v的邻接顶点
       { if(DFS(Head,p->VerAdj,vis,v)==true) return true;}
       else if(p->VerAdj!=pre) return true;
       //p之前被访问过，且不是pre的邻接顶点
    return false;
}
```

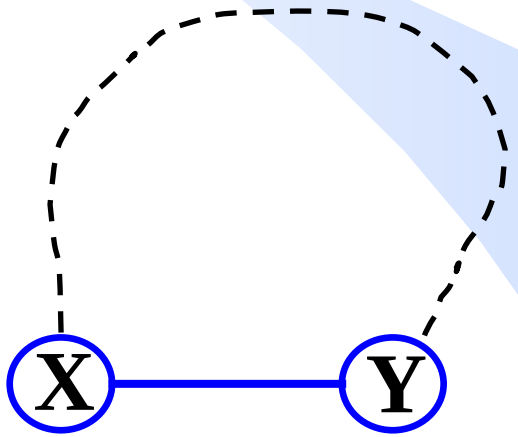


判断无向图中是否有环

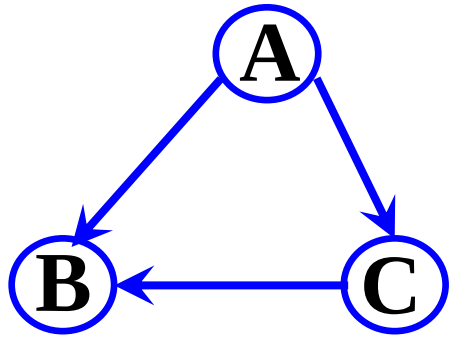
（2）并查集（连通关系⇔等价关系）

每次读入一条边(x,y)

```
if(Find(x)==Find(y))
    return true;
else Union(x, y);
```



判断有向图中是否有环



对于已访问的点，应区分是否在当前正在探索的路径上

判断有向图中是否有环

有向图：在深度优先遍历过程中，一个顶点会经历3种状态：

- *vis*[*i*]=0，顶点*i*尚未被遍历到。
- *vis*[*i*]=1，顶点*i*已经被遍历到，但对于它的遍历尚未结束。该顶点还有若干邻接顶点尚未遍历，当前算法正在递归地深入探索该顶点的某一邻接顶点。顶点*i*在当前探索的路径上。
- *vis*[*i*]=2，顶点*i*的所有邻接顶点已完成遍历，其自身的遍历也已结束。

在DFS过程中：

✓当前正在探索的路线（路径）上的点，vis值是1；

✓已经探索完的路线（路径）上的点，vis值是2；

✓还没探索的路线（路径）上的点，vis值是0.

判断有向图中是否有环

在DFS过程中，一个顶点的vis值按照0、1、2的顺序变化。

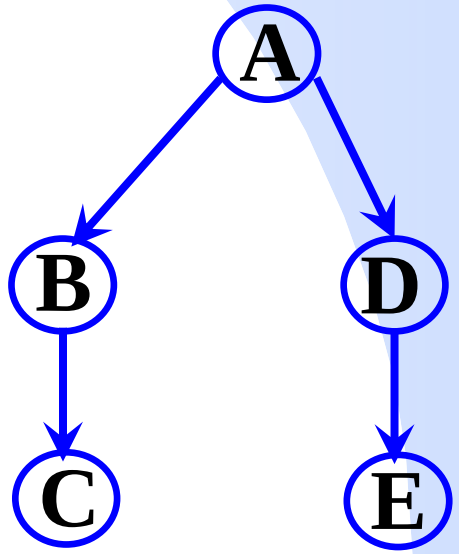
✓遍历开始时每个顶点vis值都是0;

✓当一个顶点刚被发现时，其vis值置为1，算法递归地遍历其邻接顶点;

✓当一个顶点所有邻居都完成遍历，该顶点vis值变为2，其自身也完成遍历

```
void DFS(Vertex* Head,int v,int vis[]){//vis初值为0
    printf("%d ",v); vis[v]=1;     //访问顶点v
    for(Edge*p=Head[v].adjacent;p!=NULL;p=p->link)
      if(vis[p->VerAdj]==0)
            DFS(Head, p->VerAdj, vis);
    vis[v]=2;
}
```

有环：DFS过程中遇到已访问过的且vis值为1的点



判断有向图中是否有环

```
bool HasCircle(Vertex* Head, int v, int vis[]){
    vis[v]=1;//访问顶点v
    for(Edge*p=Head[v].adjacent;p!=NULL;p=p->link){
       int k=p->VerAdj;
       if(vis[k]==0)
          if(HasCircle(Head,k,vis)==true) return true;
       if(vis[k]==1) return true; //考察v的邻接顶点
    }
    vis[v]=2;
    return false;
}
```