L14：图的概念存储遍历

吉林大学计算机学院
谷方明
fmgu2002@sina.com

学习目标

- 掌握图的概念和术语
- 掌握图的存储结构
- 掌握图的深度优先搜索
- 掌握图的广度优先搜索

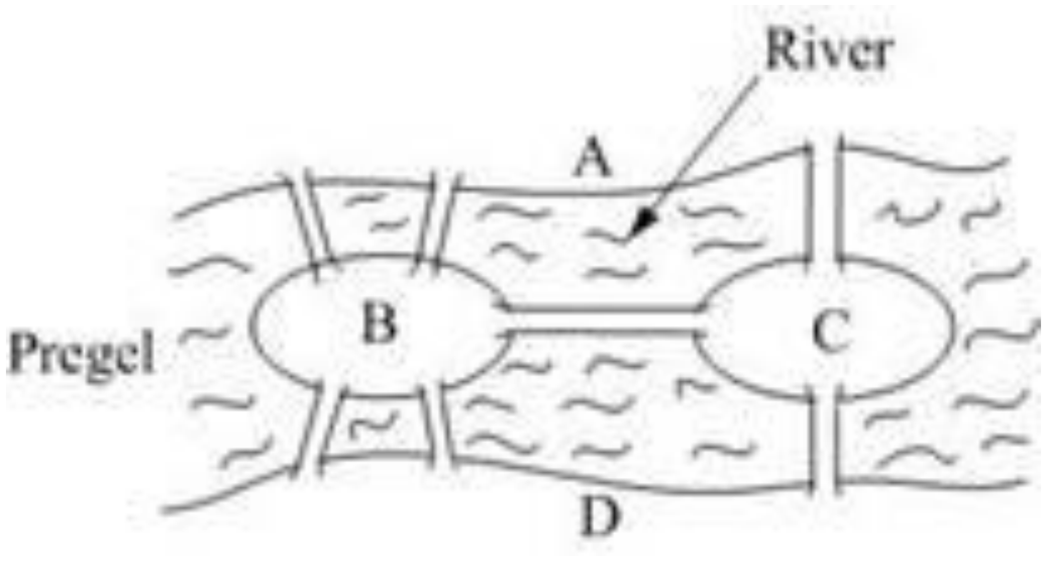
图（Graph）

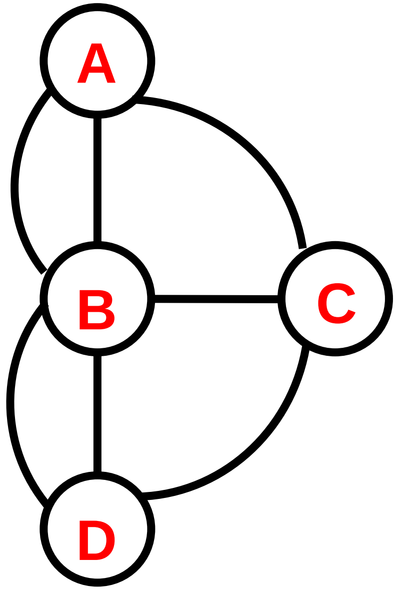
- 图是一种复杂的**非线性结构**。

- 在图结构中，结点的前趋和后继个数不加限制，结点之间的关系是任意的。

图论

- 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。



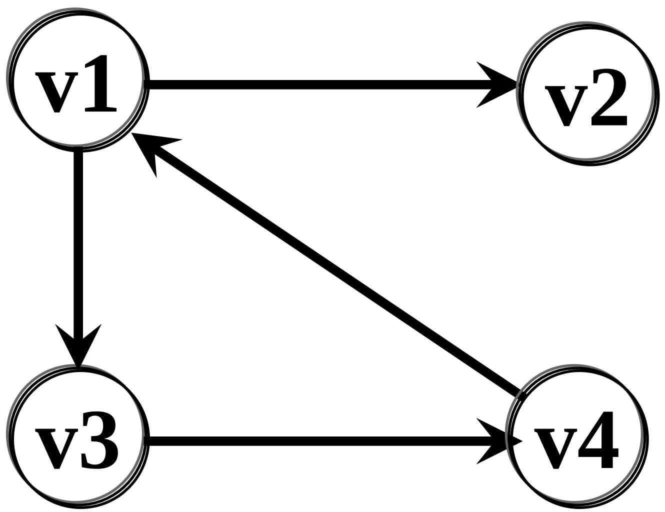


图的定义

- 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。

例：有向图

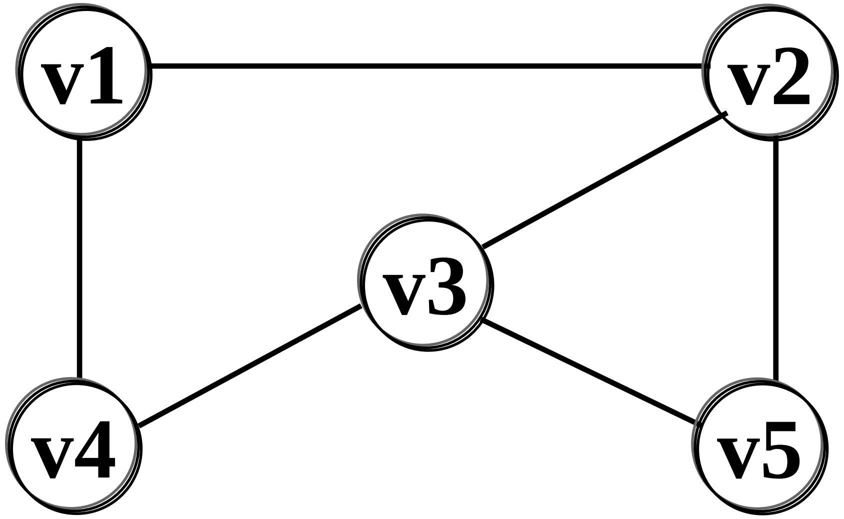


G＝(V， E)

V＝{ v1， v2， v3， v4 }

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

例：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) }

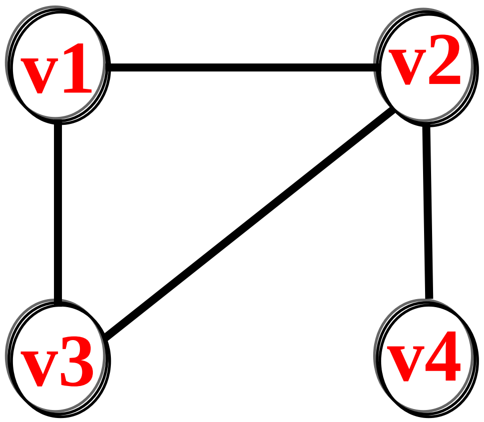
定义6.2 弧和边

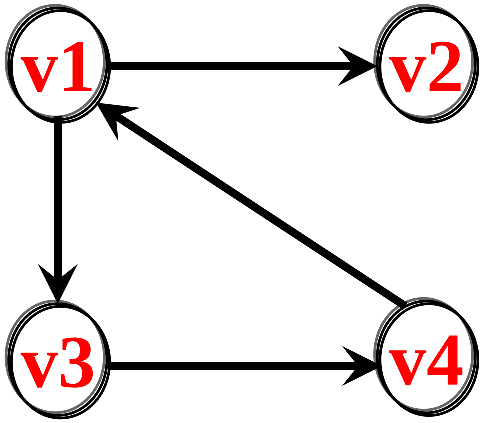
- 若 G = (V, E) 是有向图，则它的一条有向边是由 V 中两个顶点构成的有序对, 亦称为弧, 记为<u,v>, 其中 u 是边的始点，又称弧尾；v 是边的终点，又称弧头。

- 若G为无向图，则图中的边记为（ u, v ）。因为（u, v）是无序对，所以（ u, v ）和（ v, u ）代表同一条边。

定义6.3 邻接

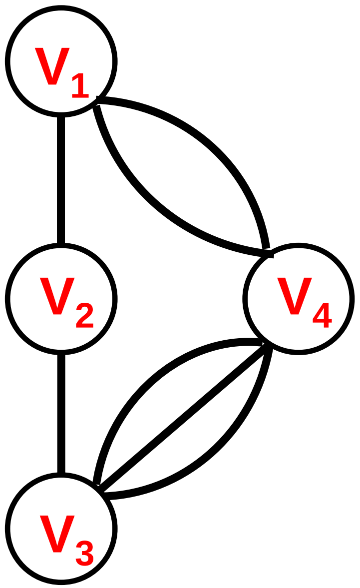
- 在无向图中，若顶点 u 和 v 间存在一条边(u, v), 则称 u, v 是相邻的，二者互为邻接顶点。
- 在有向图中，若存在一条边<u, v>，则称顶点 u 邻接到顶点 v，顶点 v 邻接自顶点 u.





定义6.4 简单图和多重图

- E 是边的集合，一般图中不会出现重复的边。重复的边简称为重边。
- 若一条边的两个顶点相同，则此边称作自环。
- 无重边和自环的图称为简单图，否则为多重图。



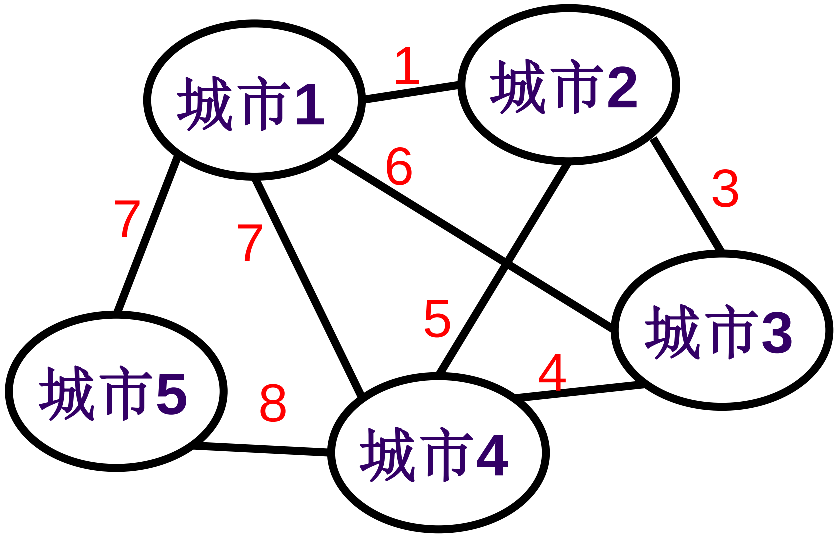
例1 函数调用图

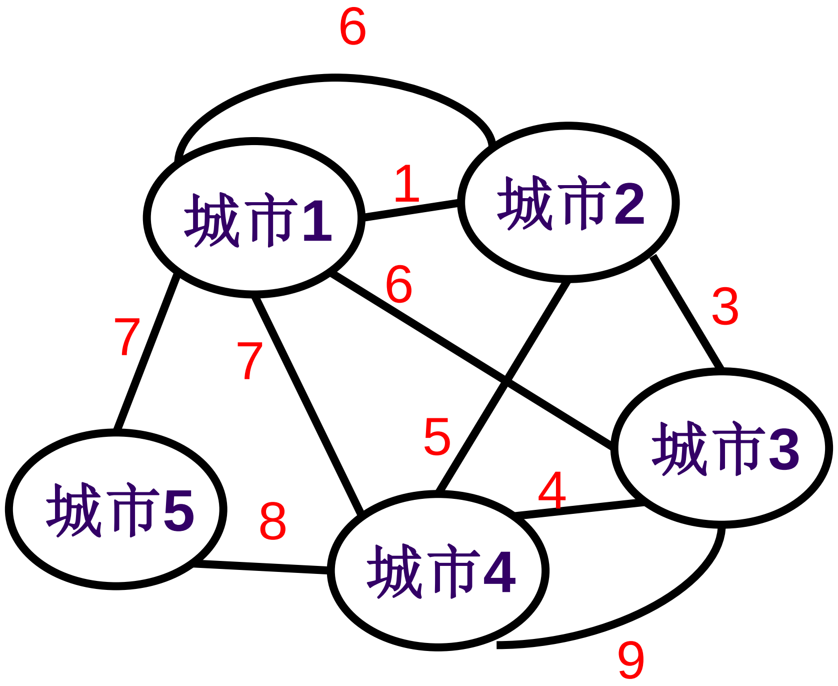
- C程序中的所有函数构成顶点集V，如果 函数 a 调用了 函数b，则定义一条从 a 指向 b 的有向边。按这种方式建立的图是有向图。

- 如果要建模直接递归函数，就会产生自环，就是多重图。

例2 公路规划

- 城市构成顶点集V，若城市 a 和 b 间有一条公路，则在 a 和 b 间连接一条边。如果道路是双向的，这种图就是无向图。如果允许两个城市间修建多条公路，这种图是就是多重图。



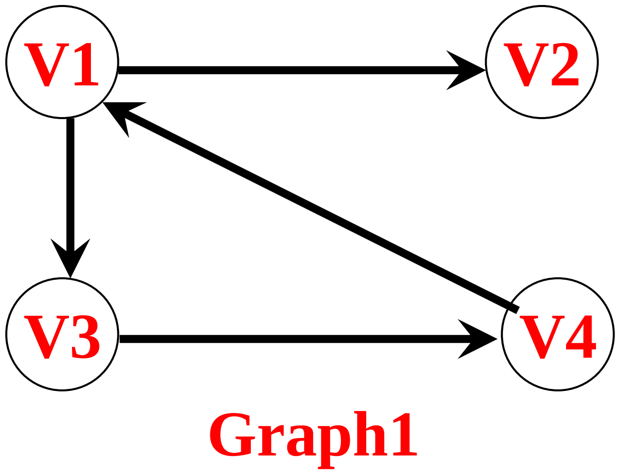


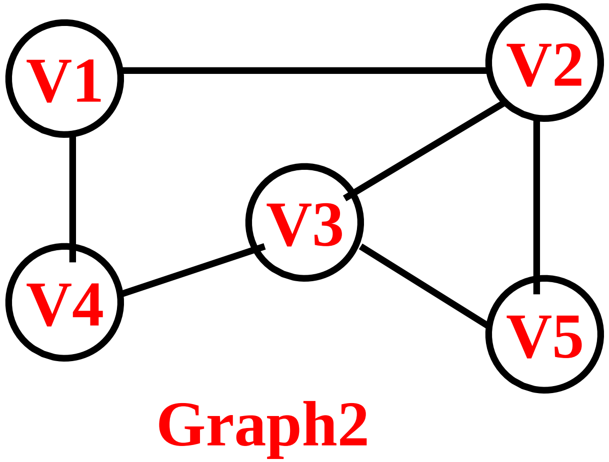
定义6.5 度

- 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.

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)

度和边的关系

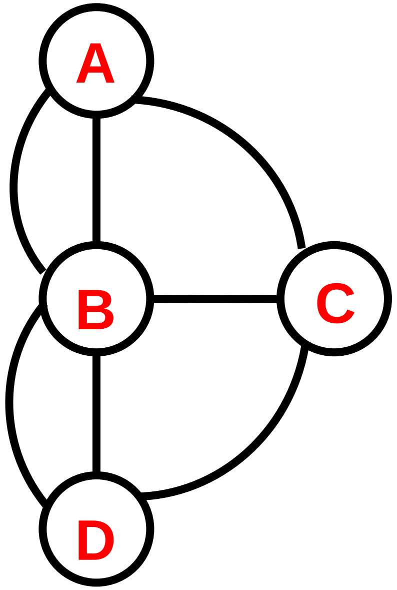
- 设图 **G**（有向或无向图）有 ***n*** 个顶点， ***e*** 条边，若顶点 v_i 的度数为 $D(v_i)$，则

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

- 因为一条边关联两个顶点，而且使这两个顶点的度数分别增加 **1.** 因此顶点的度数之和是边的两倍。

一笔画定理

- 奇点：度为奇数
- 偶点：度为偶数



- 由偶点组成的连通图一定可以一笔画成。
- 只有两个奇点的连通图一定可以一笔画成。

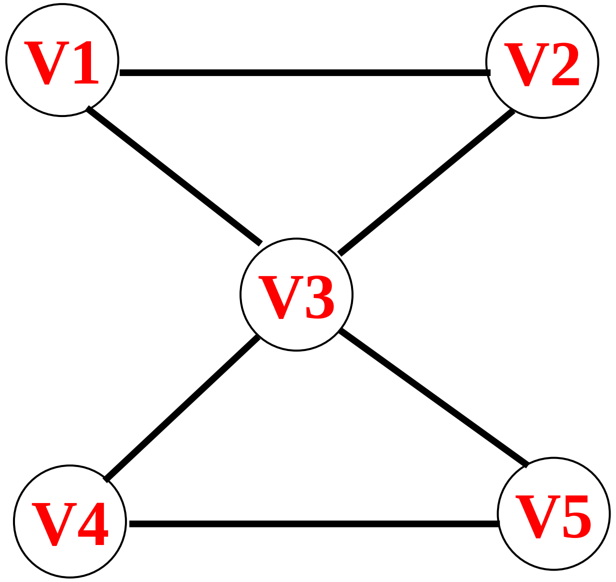
定义6.6 路

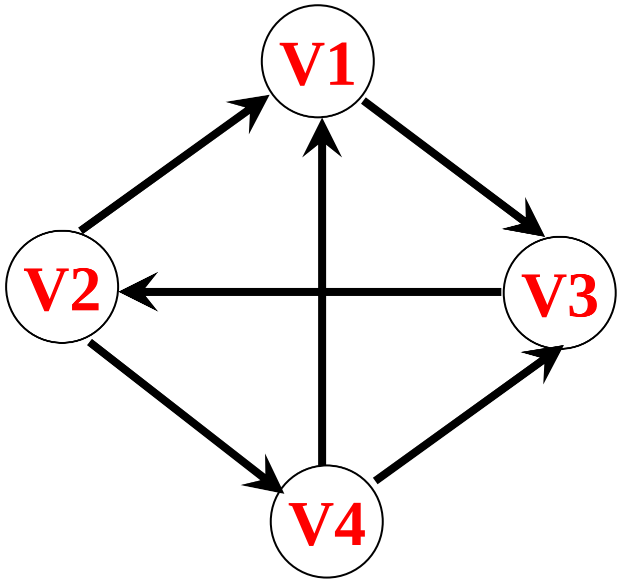
- 设G是图，若存在一个顶点序列

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

使得$<v_p , v_1>$, $<v_1 , v_2>$, …, $< v_{q-1} , v_q >$或 (v_p , v_1) , (v_1 , v_2) , …, (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。

- **路径长度：** 该路径上边的个数。
- **简单路径：** 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。简称为路径。
- **简单回路：** 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**。简称为回路





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

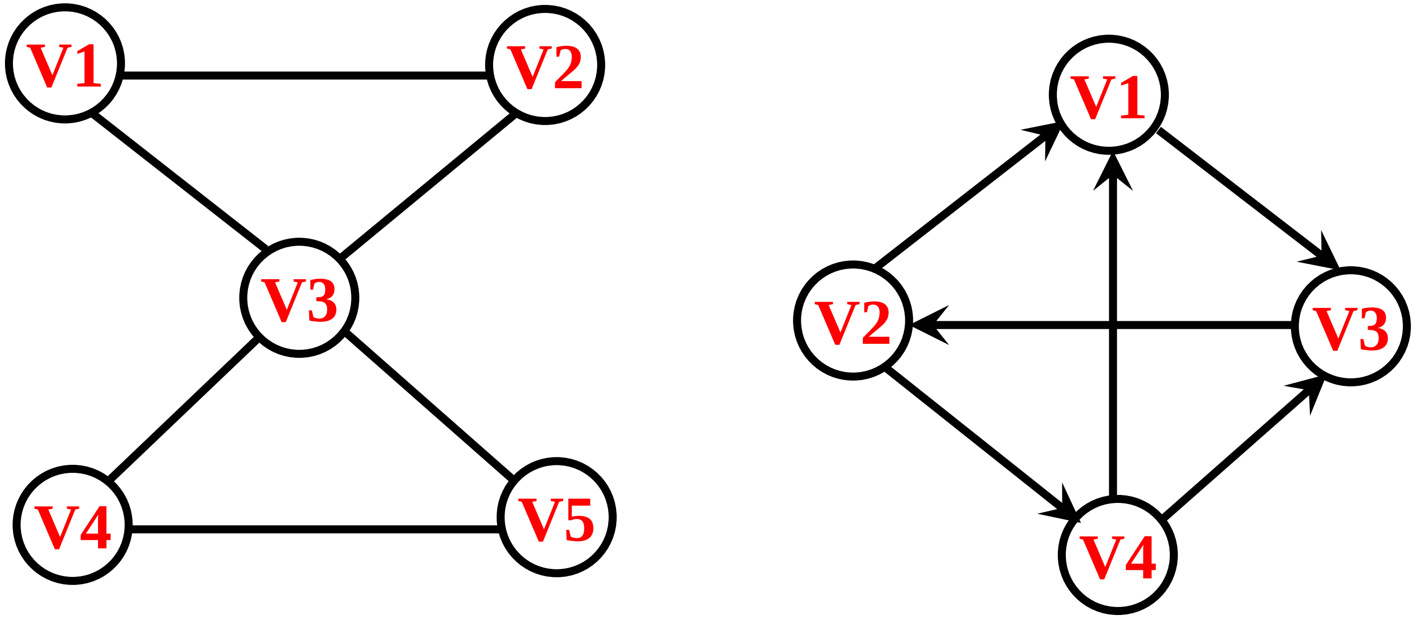
简单回路：v1 v3 v2 v1

例3 社会网

- 电影界的所有演员构成顶点集V, 其中若演员 *u* 和 *v* 共演过至少一部影片，那么在 *u* 和 *v* 之间就连接一条边。
- 演员间的这种合作关系是对等关系，按这种方式建立的图是无向图。
- Kevin Bacon距离

定义6.7 连通

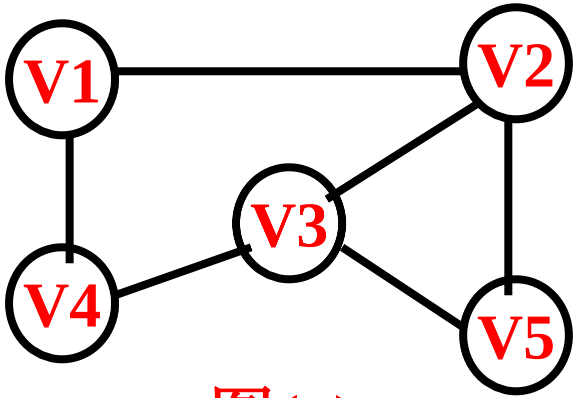
- 设G是图，若存在一条从顶点v_i到顶点v_j的路径，则称v_i与v_j连通(可及)。
- 若G为无向图，且V(G)中任意两顶点都连通, 则称G为连通图.
- 若G为有向图，且V(G)中任意两个顶点v_i和v_j, v_i与v_j以及 v_j与v_i均连通，则称G为强连通图。
- 若G为有向图，且V(G)中任意两个顶点v_i和v_j, v_i与v_j可及或 v_j与v_i可及，则称G为弱连通图。


V1
V2
V3
V4
V5
V1
V2
V3
V4

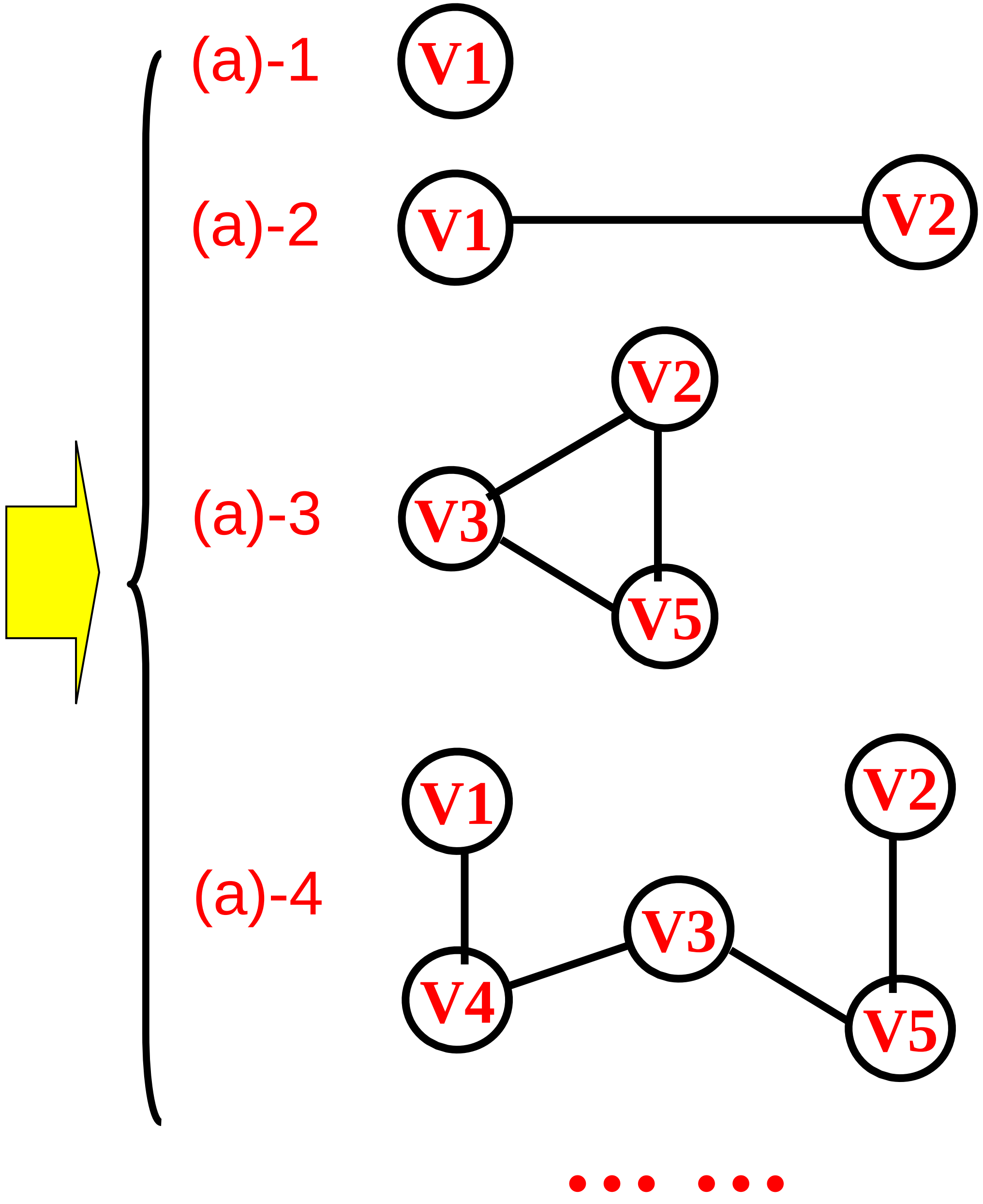
定义6.8 子图

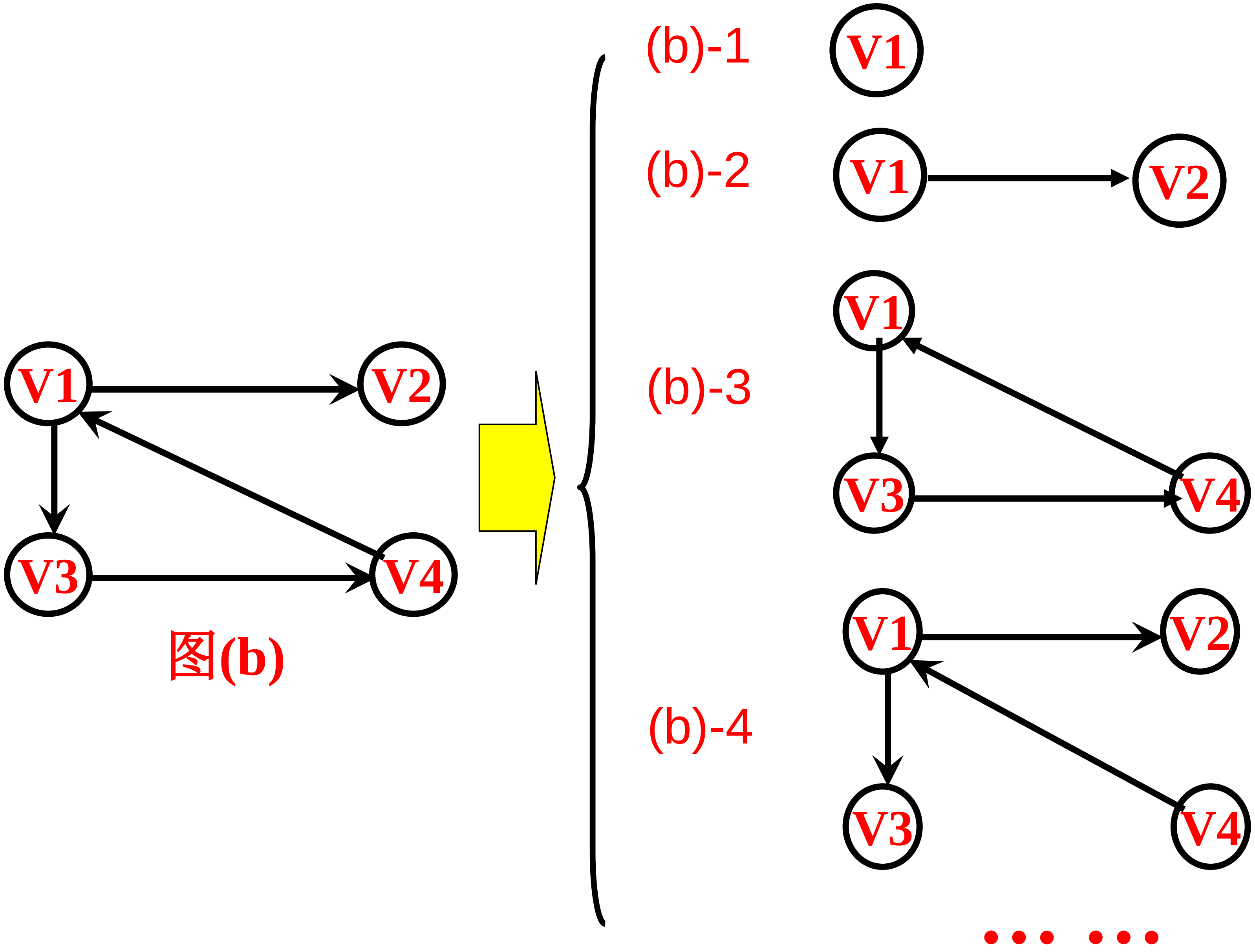
- 设G，H是图，如果$V(H) \subseteq V(G)$，$E(H) \subseteq E(G)$，则称 H 是 G 的子图，G 是 H 的母图。

- 如果 H 是 G 的子图，并且$V(H) = V(G)$，则称 H 为 G 的支撑子图（生成子图）。



图(a)



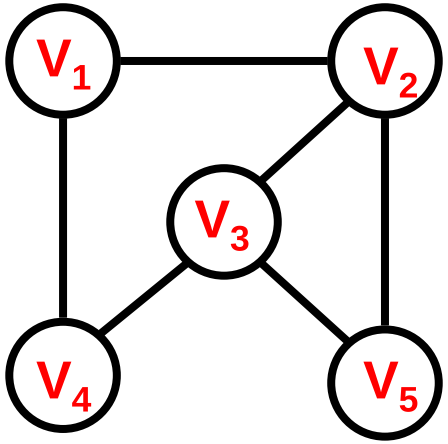

V1
V2
V3
V4
图(b)
(b)-1
V1
(b)-2
V1
V2
(b)-3
V1
V3
V4
(b)-4
V1
V2
V3
V4
… …

定义6.9 6.10连通分支

- 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<u>连通子图</u>。
  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<u>强连通子图</u>。

- 对于 G 的一个连通子图 G_K，若不存在G的另一个(强)连通子图G′, 使得$V(G_K) \subset V(G')$,则称G_K为G的<u>(强)连通分量（连通分支）</u>。

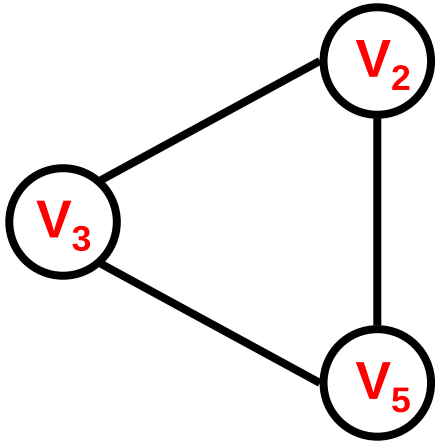
例：连通子图



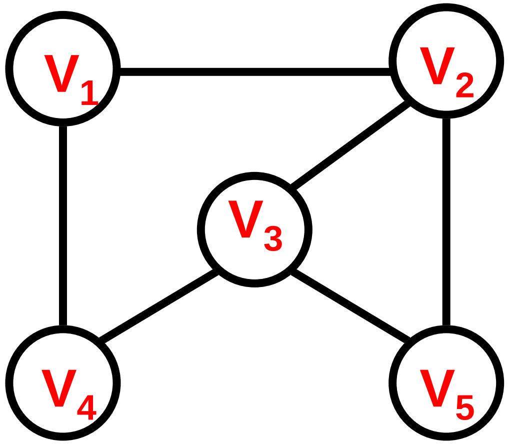
(a)

(b)

(c)



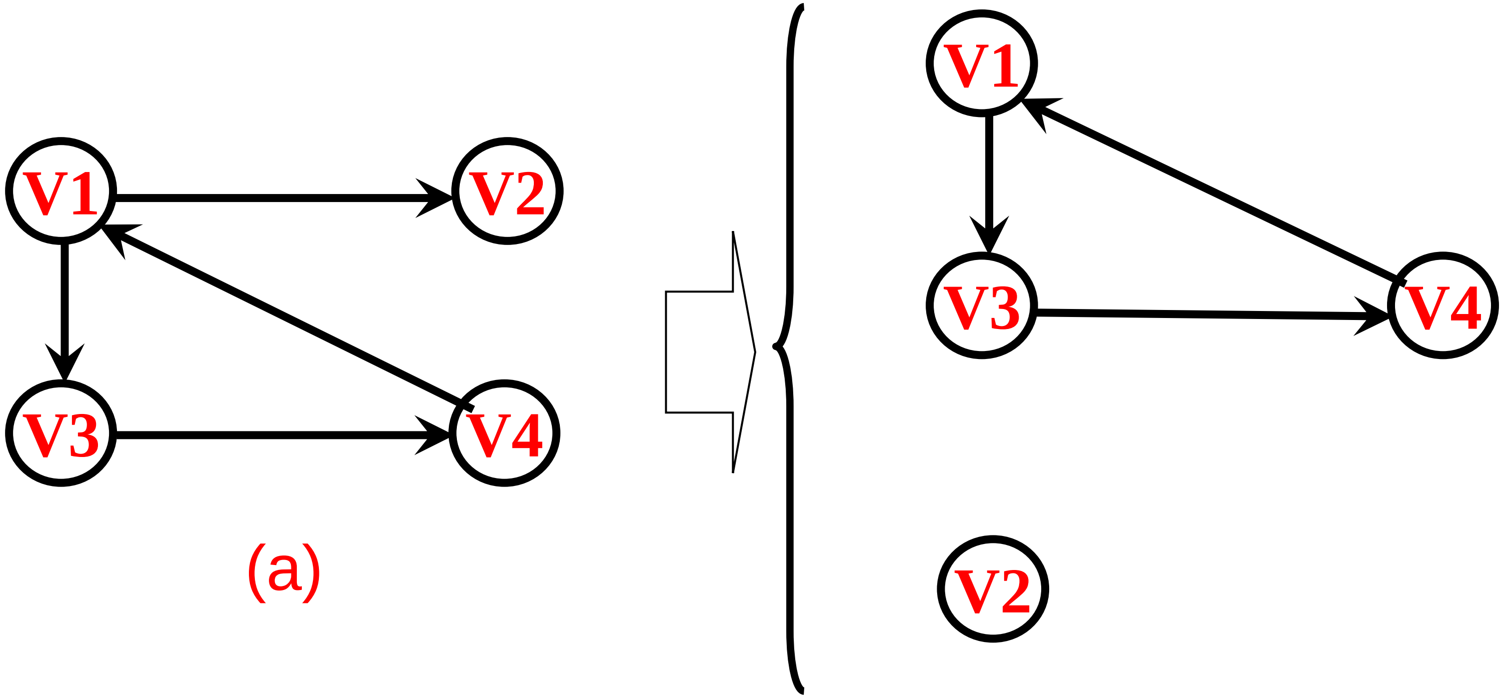
(d)



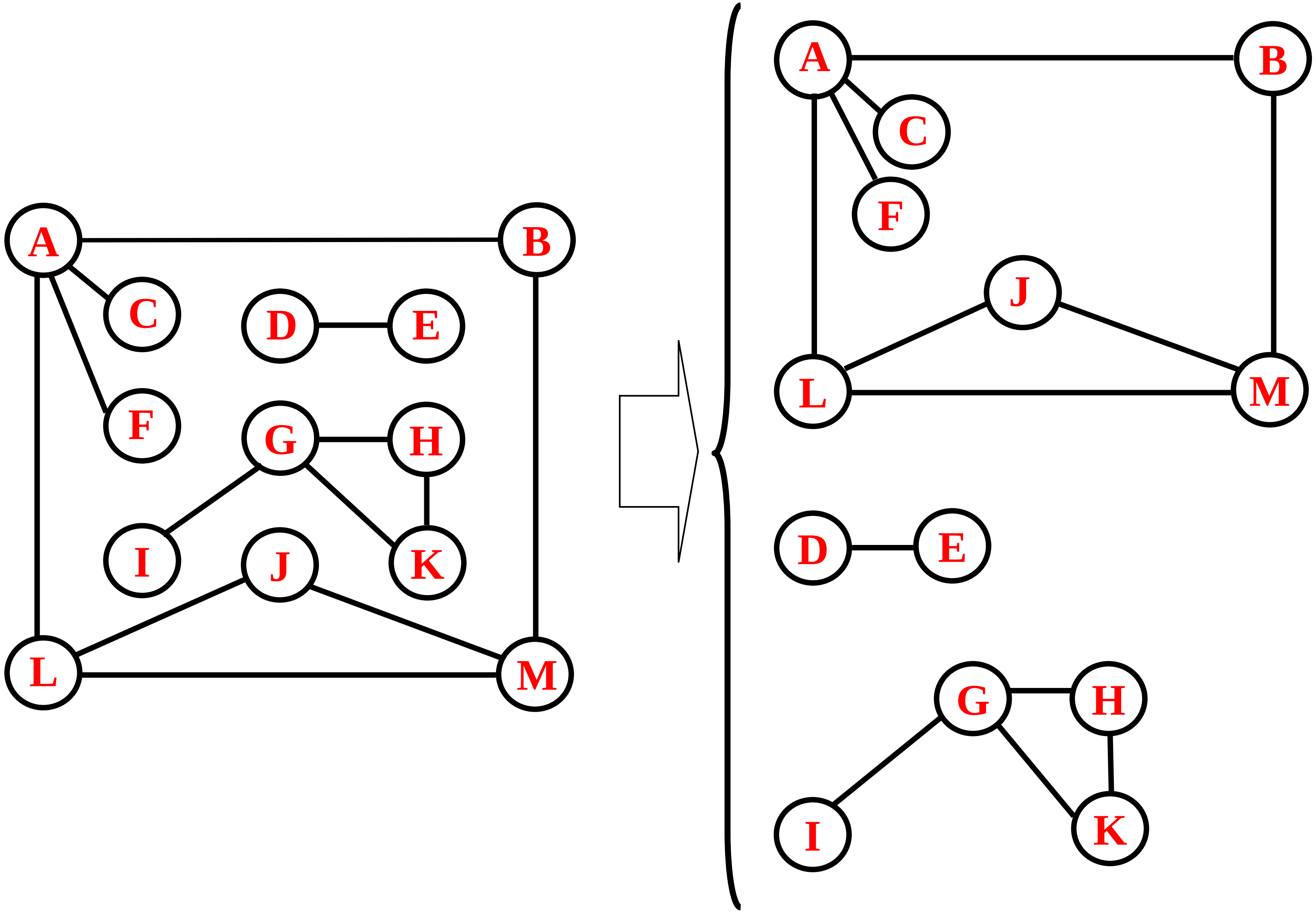
(e)

(e)是(a)的
连通分量

连通分量



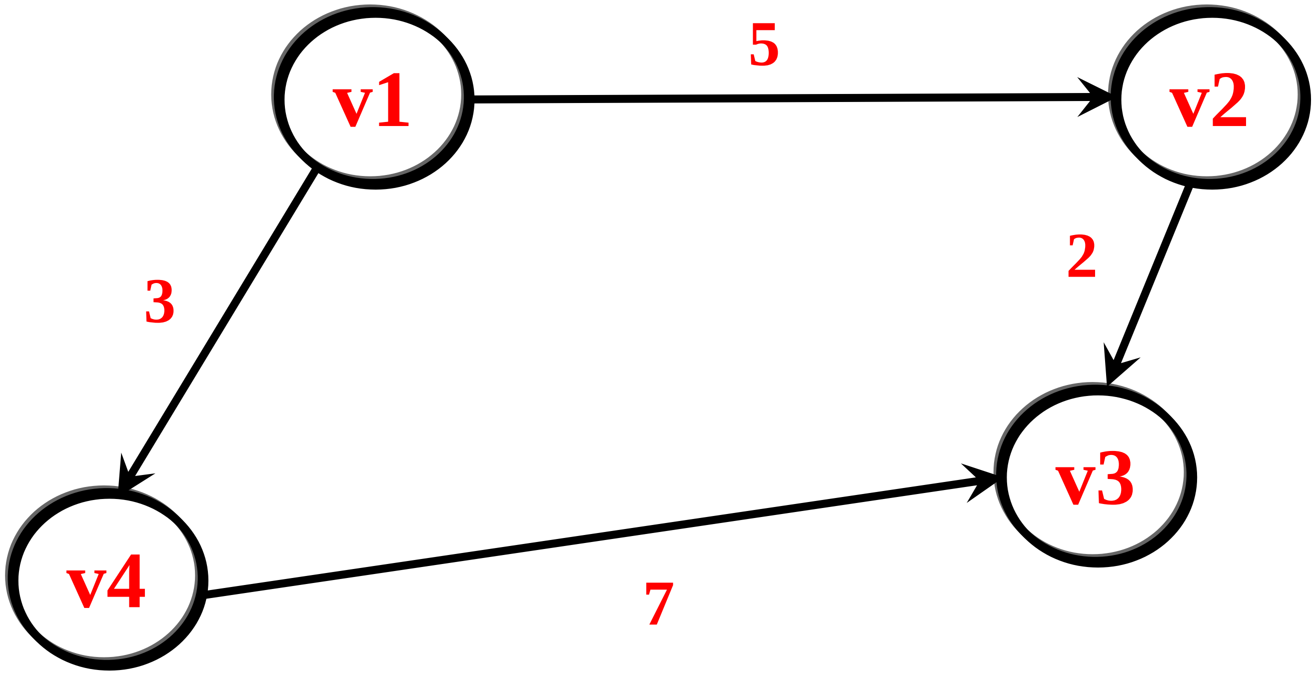
连通分量



定义6.11 6.12 网

- 设G = (V, E) 是图, 若对图的任一条边 *l*， 都有实数*w*(*l*)与其对应， 则称 G 为权图（网）， *w*(*l*)为(边)权。记为G = (V, E, *w*). 记 *w*(*u*,*v*) 表示 *w*((*u*,*v*)) 或 *w*(<*u*,*v*>)。权通常用来表示从一个顶点到另一个顶点的距离或费用

- 若$\sigma=(v_0, v_1, v_2, \ldots, v_k)$是权图 G 中的一条路径, 则称$|\sigma|=\Sigma_{i=1\ldots k}w(v_{i-1},v_i)$ 为加权路径σ的长度



v1
5
v2
3
2
v3
v4
7

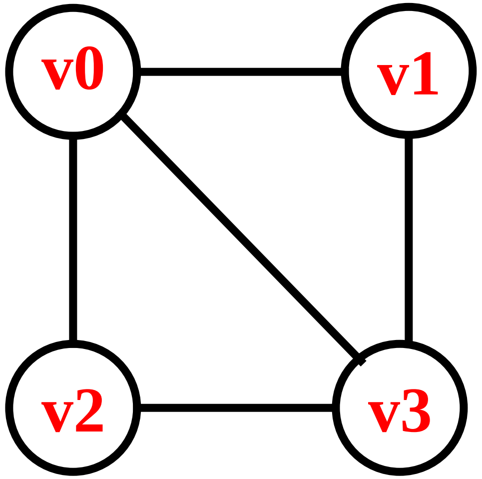
<table>
<tr><th>无向图</th><th colspan="2">有向图</th></tr>
<tr><td rowspan="2">边</td><td colspan="2">弧</td></tr>
<tr><td>弧头</td><td>弧尾</td></tr>
<tr><td>邻接</td><td>邻接到</td><td>邻接自</td></tr>
<tr><td>结点的度</td><td colspan="2">结点的出度、入度</td></tr>
<tr><td>连通图</td><td colspan="2">强连通图、弱连通图</td></tr>
</table>

图的存储

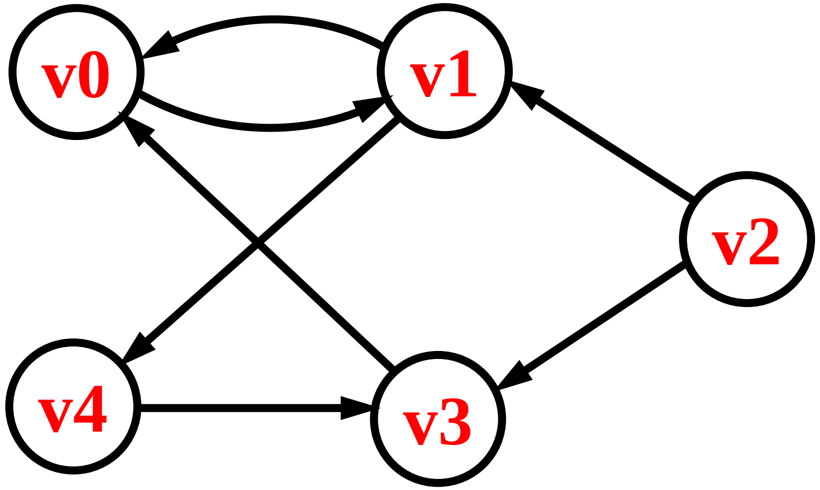
- 点
  - ✓ 只关注顶点的编号，可不存
  - ✓ 若存，可用线性表存储图的顶点集合$v_1,v_2,...v_n$。
- 边
  - ✓ 邻接矩阵
  - ✓ 邻接表
  - ✓ 边表
  - ✓ 潜表示

邻接矩阵

- 边用 $n \times n$ 矩阵 $A=(a_{ij})$ 表示，A 的定义如下:
- 非权图的邻接矩阵，则:
  $a_{ij} = 1$，当 $i \neq j$ 且 $\langle v_i, v_j \rangle$ 或 (v_i, v_j) 存在；
  $a_{ij} = 0$，否则



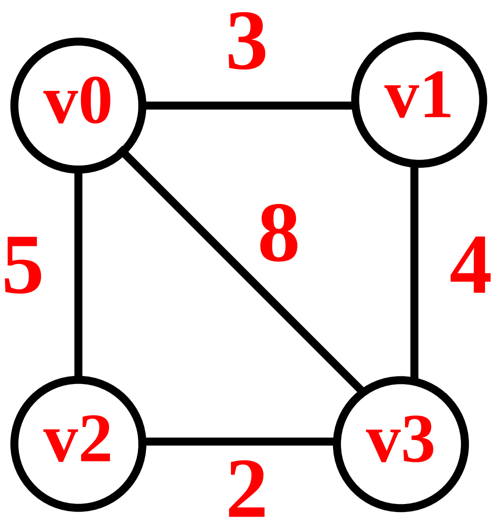
$$
\begin{array}{c c}
 & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}
\end{array}
$$



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

权图的邻接矩阵

- a_{ij} 为对应边$<v_i,v_j>$或(v_i,v_j)的权值，当 $i \neq j$ 且 $<v_i, v_j>$ 或 (v_i, v_j) 存在时。
- $a_{ii} = 0$；
- $a_{ij} = \infty$， 当 $i \neq j$ 且 $<v_i, v_j>$ 或 (v_i, v_j) 不存在时；



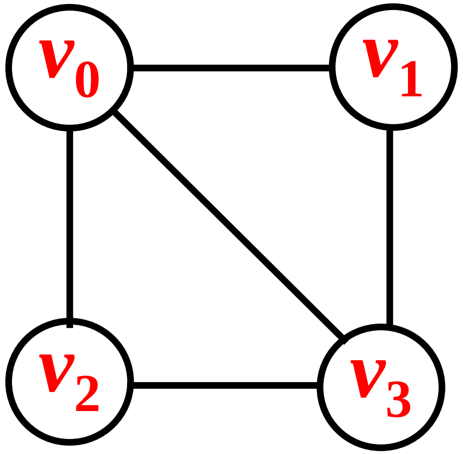
$$
\begin{array}{c c}
 & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} 0 & 3 & 5 & 8 \\ 3 & 0 & \infty & 4 \\ 5 & \infty & 0 & 2 \\ 8 & 4 & 2 & 0 \end{pmatrix}
\end{array}
$$

邻接矩阵小结

- **借助邻接矩阵, 很容易求出图中顶点的度.**
  - ✓ 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 行非零元素的个数为顶点v_i 的出度；第 j 列非零元素的个数为顶点 v_j 的入度.
- **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**

邻接表

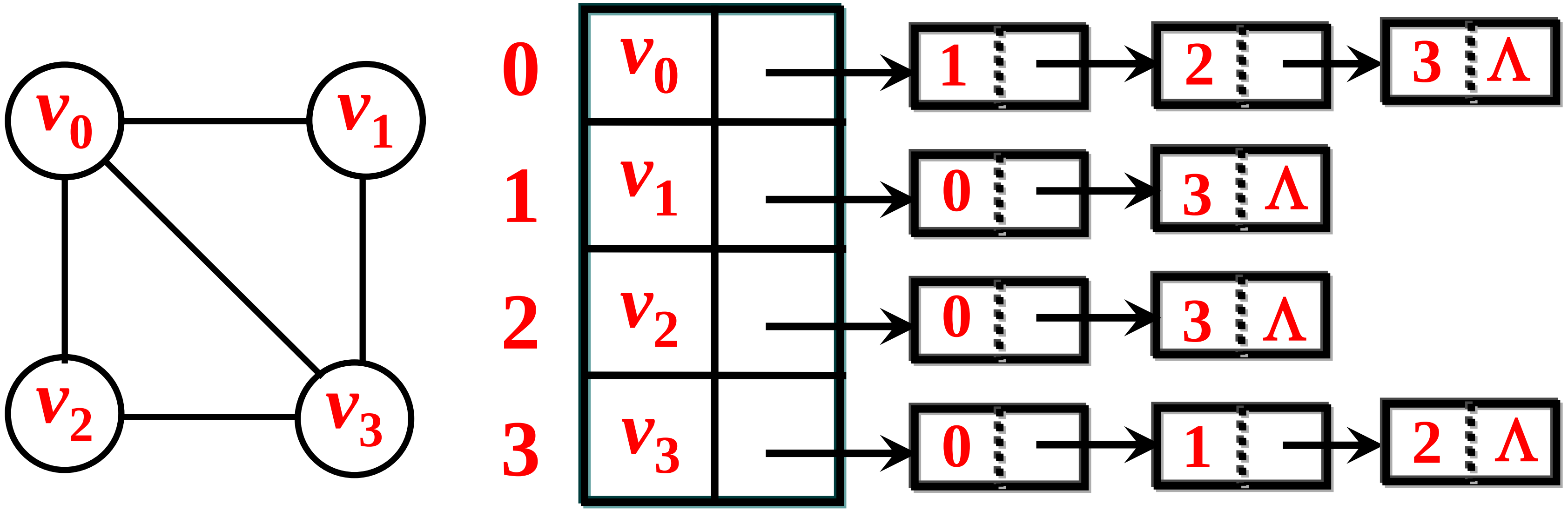
- 用线性表存储每个顶点发出的边
  - ✓ 定长数组A[n][d];
  - ✓ 可变长数组vecter;
  - ✓ 邻接链表



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

邻接链表

- 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。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链表。



顶点的结构

VerName	adjacent

非权图中边结点结构为（VerAdj , link）

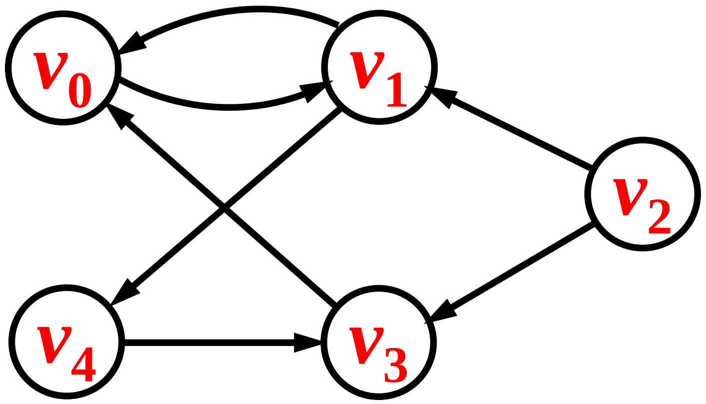
VerAdj	link

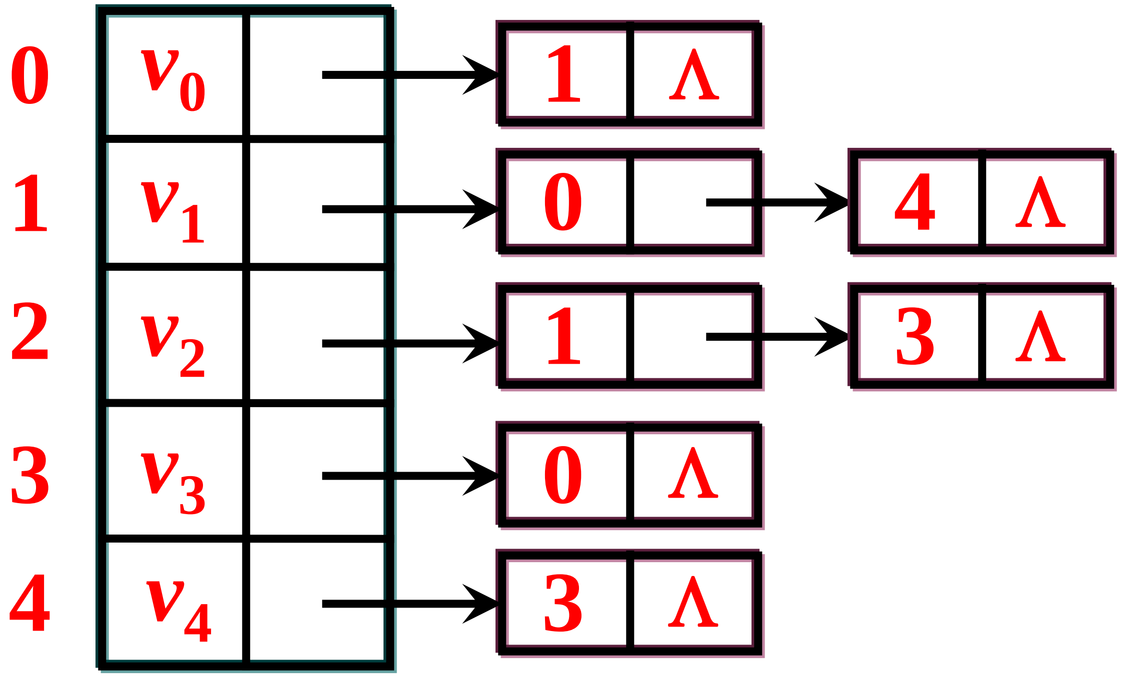
权图中边结点结构为（VerAdj , cost , link）

VerAdj	cost	link

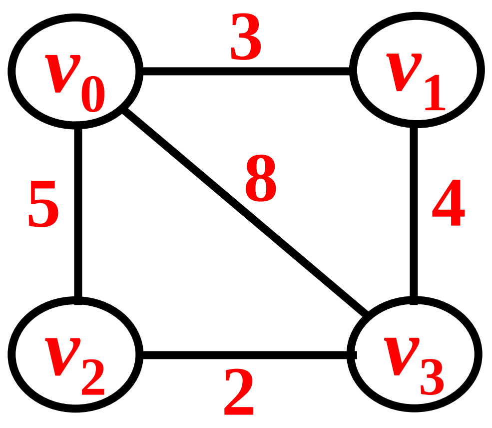
图的邻接链表结构

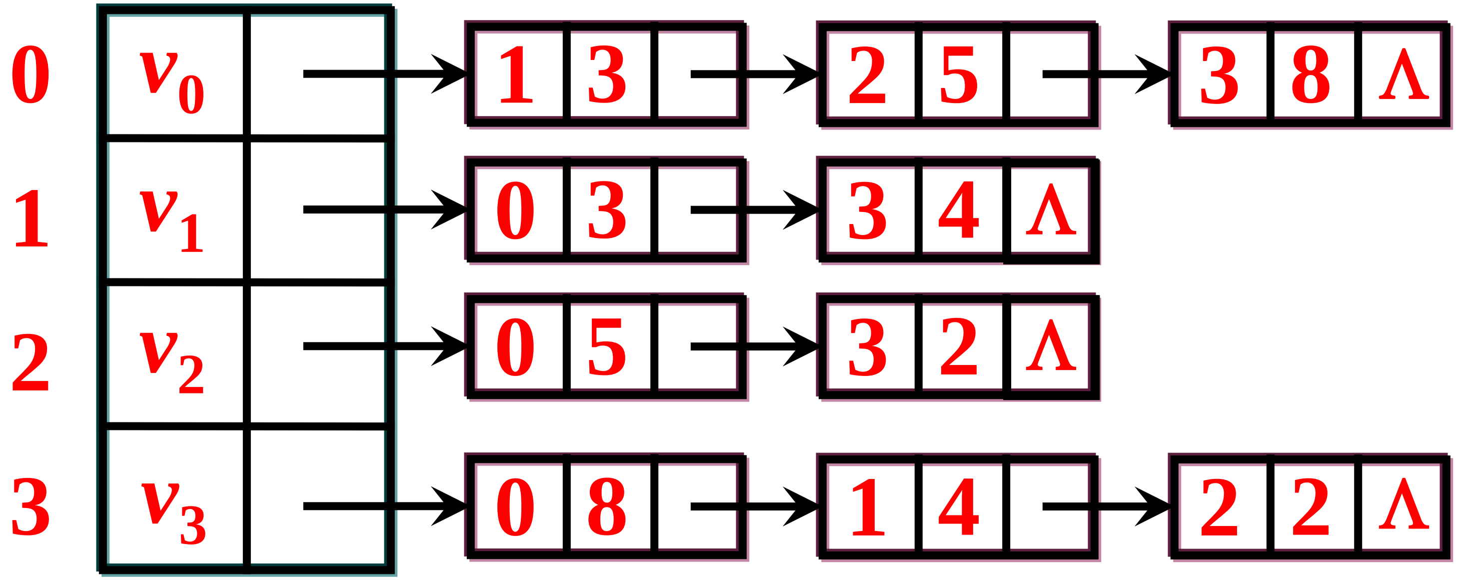
```
typedef struct edge  {    // 边结点的结构体
   int VerAdj ; // 邻接顶点序号，用自然数编号
   int cost ;    // 边的权值
   struct edge *link ;  // 指向下一个边结点的指针
} Edge;
struct Vertex {    //顶点表中结点的结构体
    int VerName;  // 顶点的名称
    Edge *adjacent; // 边链表头指针
} ;
struct Graph{
   Vertex* head;
};
```


v_0
v_1
v_2
v_3
v_4


0 v_0 1 Λ
1 v_1 0 4 Λ
2 v_2 1 3 Λ
3 v_3 0 Λ
4 v_4 3 Λ

权图的邻接表（无向图）



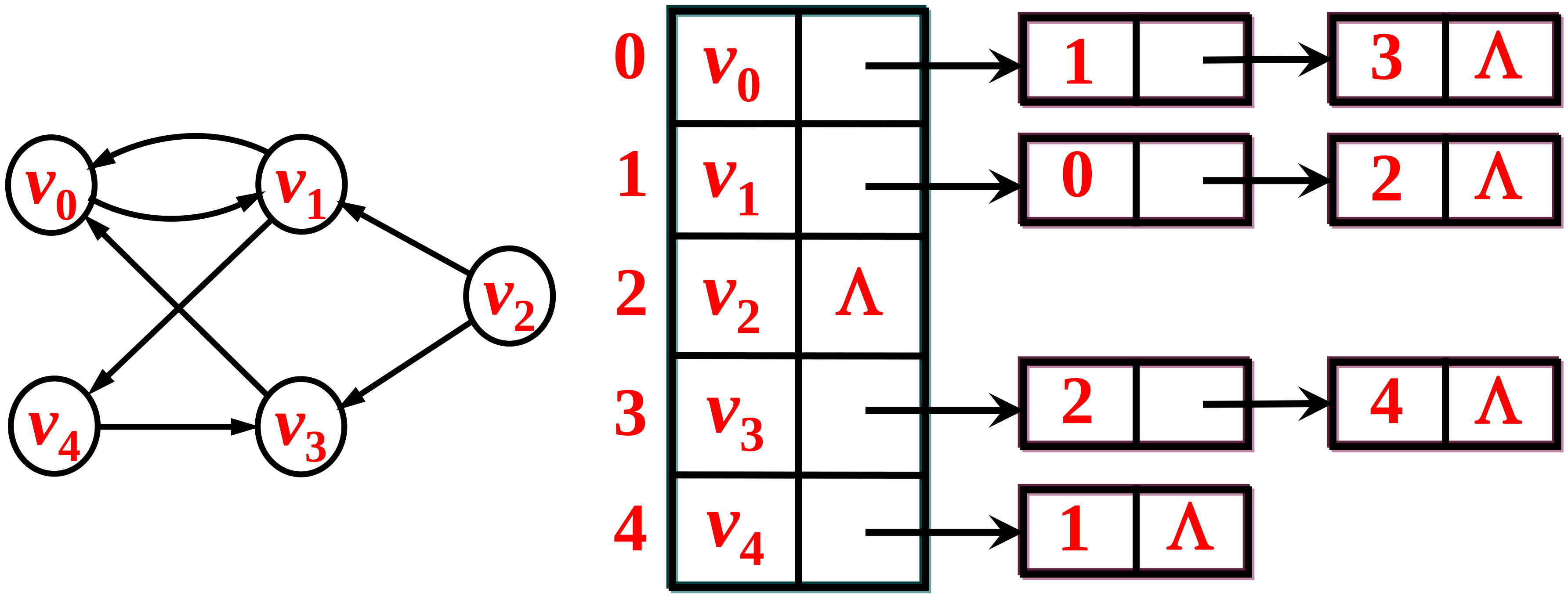


邻接表小结

- 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）**.**

逆邻接表

- 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。



十字链表和邻接多重表

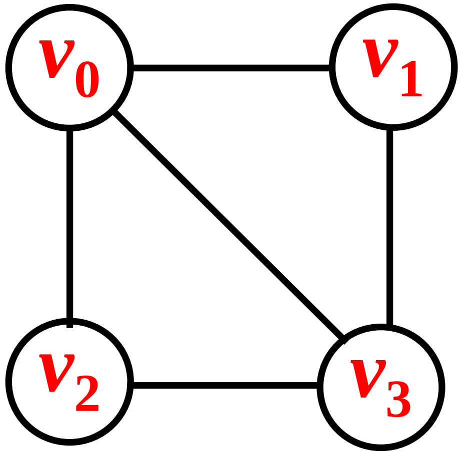
- **十字链表：** 解决**有向图**的邻接表不能同时方便表示出度和入度的问题。 **（知道即可）**

- **邻接多重表：** 解决**无向图**的邻接表一条边存储两遍的问题。 **（知道即可）**

邻接矩阵 VS 邻接链表

	邻接矩阵	邻接表
存储表示	唯一	不唯一；取决于边的输入顺序和链接次序
空间复杂度	$O(n^2)$ 稠密图	O(n+e) 稀疏图
求顶点的度	无向图O(n) 有向图O(n)	无向图 O(n) 有向图 O(n)+O(e)
判定(vi,vj)	O(1)	O(n)
求边的数目	$O(n^2)$	O(n+e)

边表



	1	2	3	4	5	6	7	8	9	10
U	0	0	0	1	1	2	2	3	3	3
V	1	2	3	0	3	0	3	0	1	2

- 有序边表：引入be[N],en[N]，可作邻接表用

图的存储小结

- 邻接链表的静态链表版是图存储的前向星法

- 隐含表示：不显式存储边。如搬箱子游戏，将每个格子建模为一个点；每个格子可以上下左右去相邻格子，建模成边。知道(i,j)就知道了这4条边，此时不必存储边。

- 用什么方式存储图，取决于对图的操作。

图的遍历

- 从图的任一顶点出发，沿着边访遍图中所有顶点，且每个顶点仅访问一次，这一过程称作图的遍历 (**Graph Traversal**)。
- 图的遍历是图的一种基本操作，图的许多其它操作都是建立在遍历的基础之上。

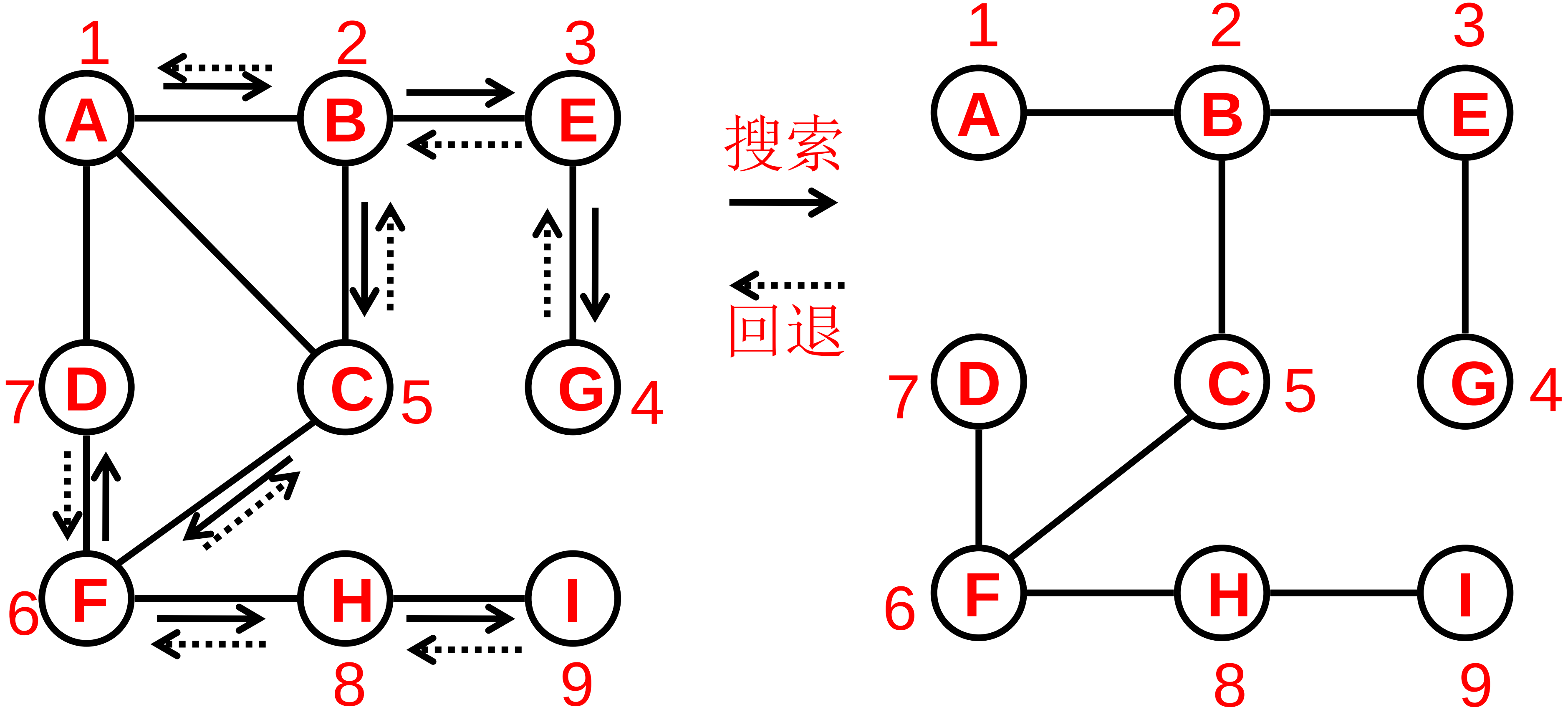
分析

- 出发结点：在图结构中，没有一个“自然”的首结点 。需用户指定。
- 重复访问：访问完某个顶点后可能沿着某些边又回到曾经访问过的顶点。为避免重复，用标识数组*visited*[ ]，初值为0，标识未访问。如果顶点 i 被访问，则置visited[i]为1.
- 访问策略：多个邻接顶点，如何选择下一个访问顶点。
  - ✓ 深度优先
  - ✓ 广度优先

1. 深度优先遍历

- 深度优先遍历又被称为深度优先搜索 (**Depth First Search**， **DFS**)
- 访问策略: 从图中某一起始顶点 **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 邻接且还没有访问过的任一顶点 w_2；... 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。此时，回溯到上一个被访问的顶点，看它是否还有其它没被访问的邻接顶点。若有，则访问该邻接顶点，进行与前述类似的访问；若没有，进一步回溯。

DFS示例



深度优先搜索过程　　　　深度优先搜索树

深度优先遍历递归算法(连通分支)

算法**DepthFirstSearch** (*v*) //或DFS(v)

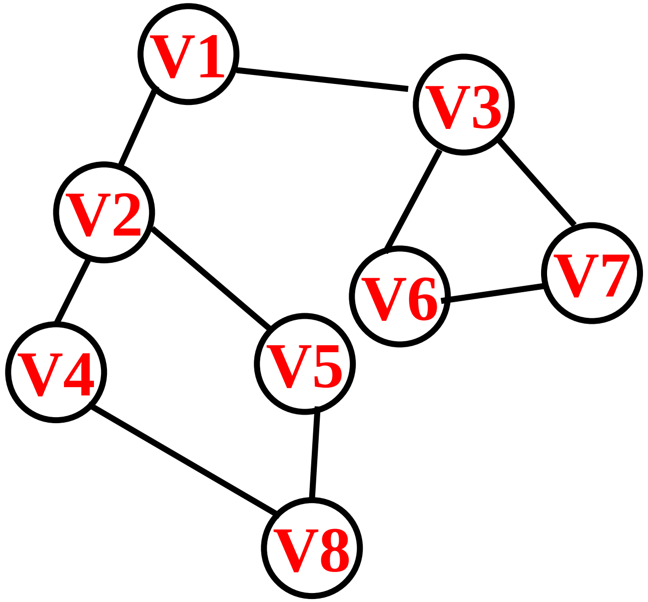
/* 深度优先遍历连通分支的递归算法,*visited*全局*/

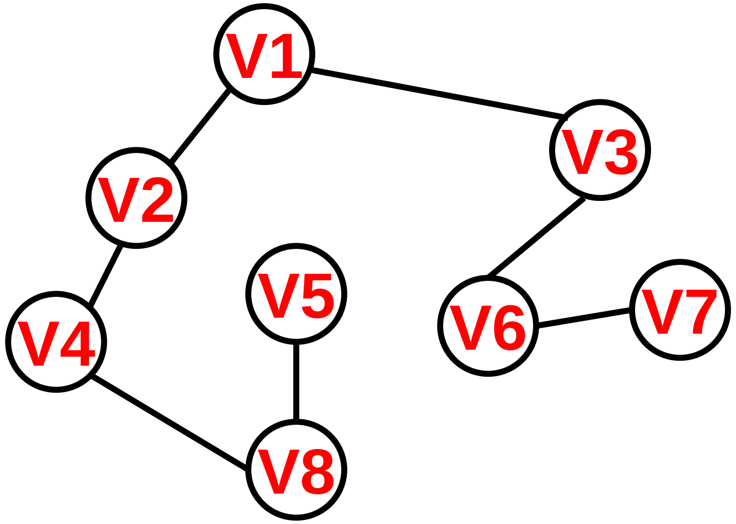
DFS1[初始化]

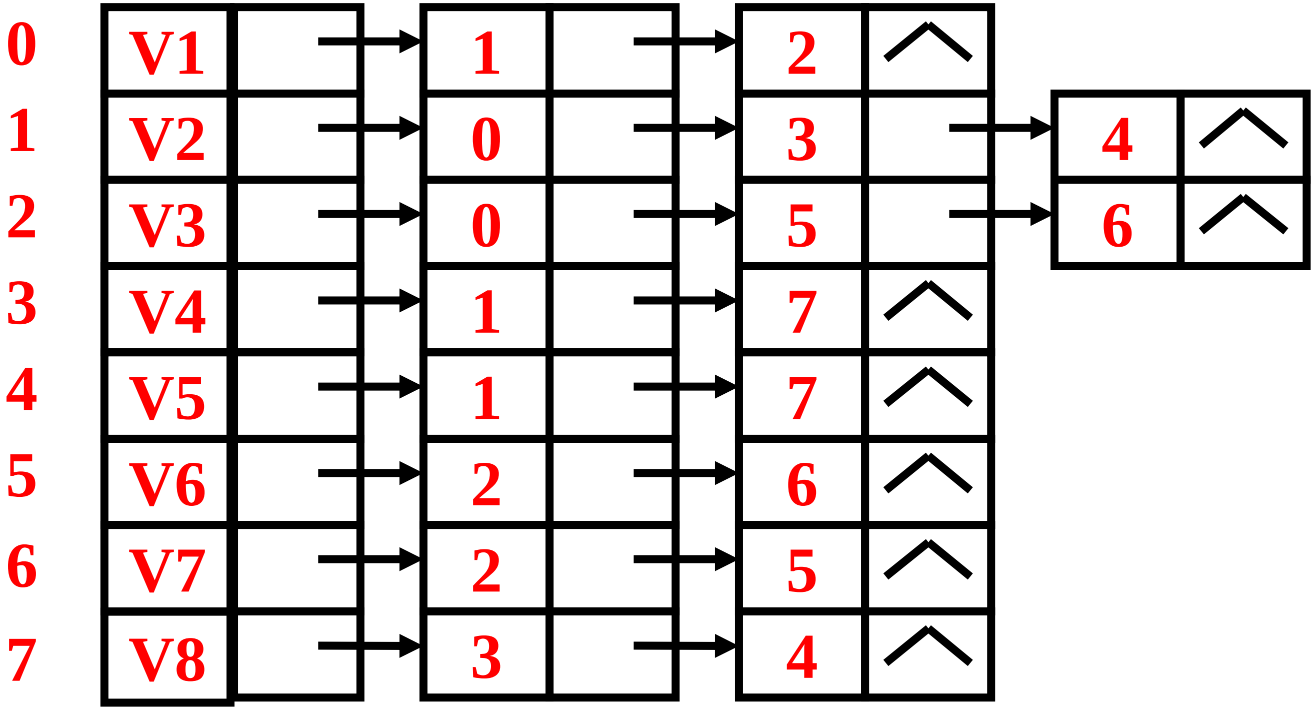
```
    visit( v ); //可以是输出、操作等
    visited[v] = 1;
```

DFS2[深度优先遍历]

```
    for( p =  Head[v] -> adjacent ; p ; p = p -> link )
        if (visited[ p -> VerAdj ] != 1 )
                DepthFirstSearch (p -> VerAdj);
```


V1
V3
V2
V6
V7
V4
V5
V8


V1
V3
V2
V5
V6
V7
V4
V8


0 V1 1 2
1 V2 0 3 4
2 V3 0 5 6
3 V4 1 7
4 V5 1 7
5 V6 2 6
6 V7 2 5
7 V8 3 4

DFS算法分析

- 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**。**DFS**每个结点只访问一次，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**)**。

定理6.1

- **DFS每次遍历一个连通分支**

  $\mathbf{V_s = \{ v \mid v \in V}$ **且** $\mathbf{s \rightarrow_{E^*} v \}}$

  $\mathbf{V_{dfs} = \{ v \mid v \in V}$ **且** $\mathbf{visited[v] = 1 \}}$

- 集合相等的证明方法
  - ✓ 下推上： 显然
  - ✓ 上推下： 数学归纳法.

DFS迭代算法——问题分解

1. 将visited[]用0初始化；初始顶点v压入栈, visited[v] =1。
2. 循环处理：

若堆栈为空，则迭代结束；

否则，从栈顶弹出一个顶点v；访问v ；然后将v的所有未被访问的邻接顶点压栈，并将其visited更新为1。

注：教材更正，入栈打标志

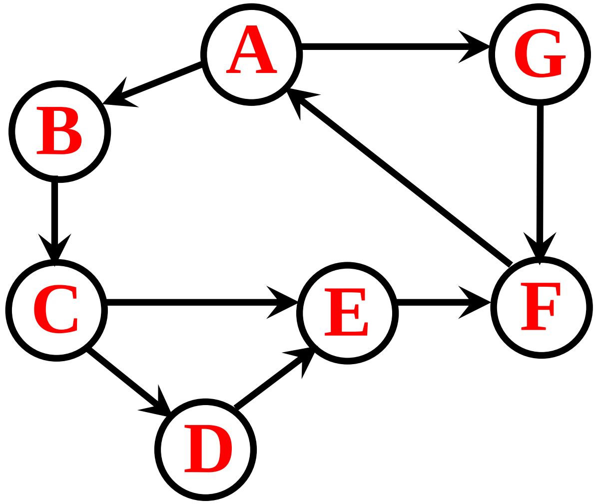
迭代算法NRDFS (v)

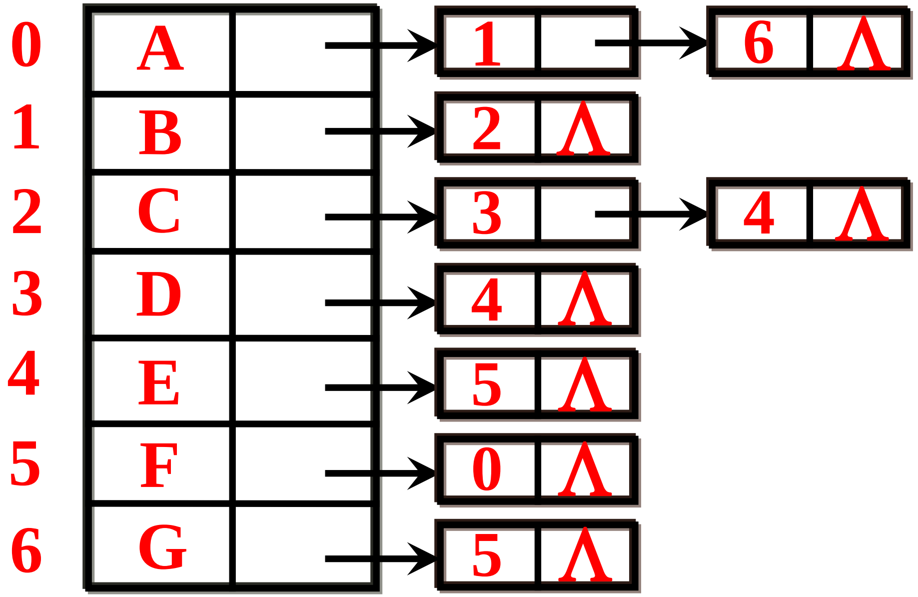
NRDFS1. [初始化]

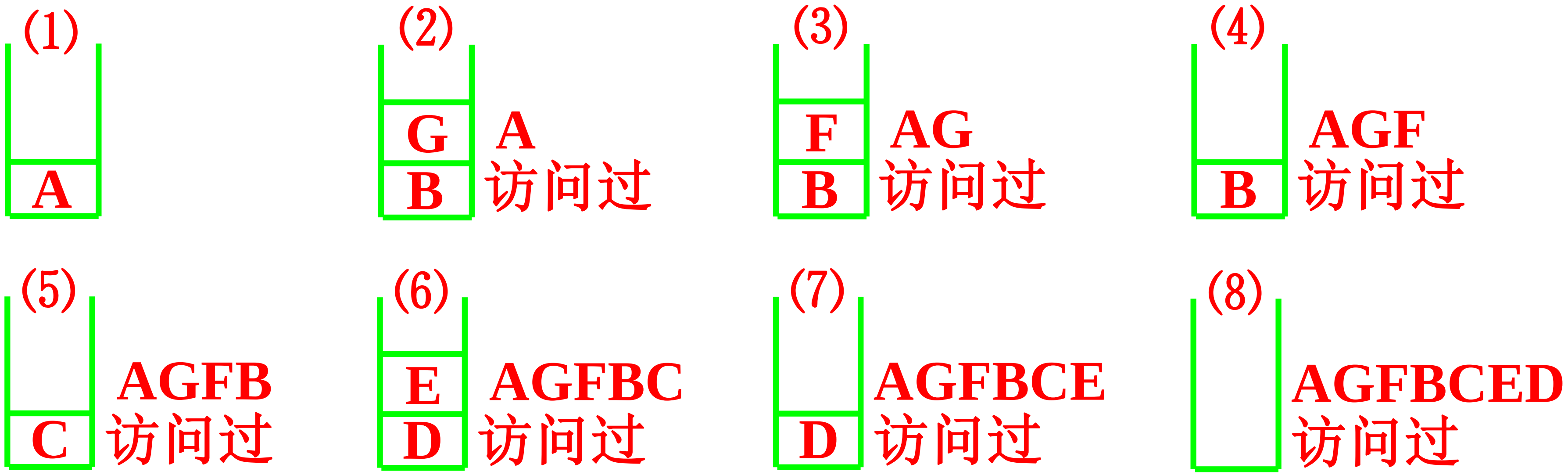
```
for( i=1 ; i<=n ; i++ ) visited[i] = 0;
CREATESTACK(S) ;
 S.push(v). visited[v]=1.
```

NRDFS2. [用*S* 深度优先遍历]

```
while ( ! S.empty()){
   v = S.pop();
   cout<<v;
   for( p=Head[v]->adjacent ;  p ; p = p -> link ）
      if(visited [ p ->VerAdj] ==0 ) {
           S.push( p->VerAdj); visited[v]=1;}
 }
```

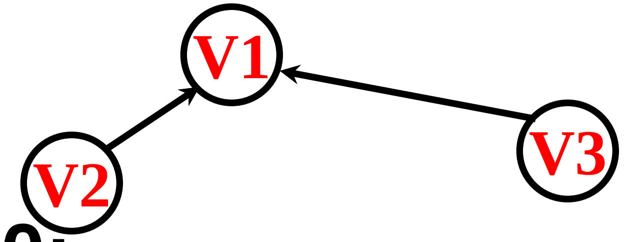

A
G
B
C
E
F
D


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


(1)
A
(2)
G
B
A
访问过
(3)
F
B
AG
访问过
(4)
B
AGF
访问过
(5)
C
AGFB
访问过
(6)
E
D
AGFBC
访问过
(7)
D
AGFBCE
访问过
(8)
AGFBCED
访问过

图的深度优先遍历算法

- 对于非连通图，需要多次调用**DFS或NRDFS**算法
- 算法GDFS()



GDFS1.[初始化]

```
    for ( i=1; i<=n; i++ ) visited[i] = 0;
```

GDFS2.[遍历多个连通分量]

```
  for ( i=1; i<=n; i++ )
     if ( visited[j] == 0 )
        DepthFirstSearch ( i)
```

课后思考

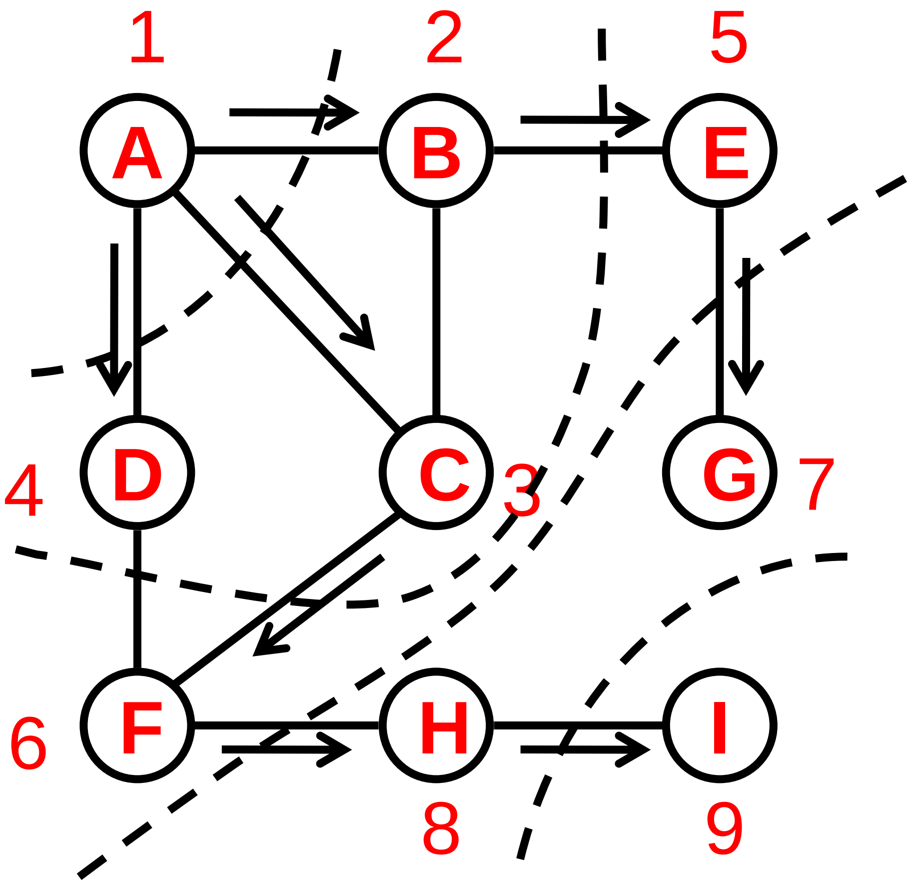
- 深度优先遍历的迭代算法如何定制访问顺序?
- 深度优先遍历可以先遍历诸邻接结点、再访问自身吗？如果可以，怎么处理?

2. 广度优先遍历

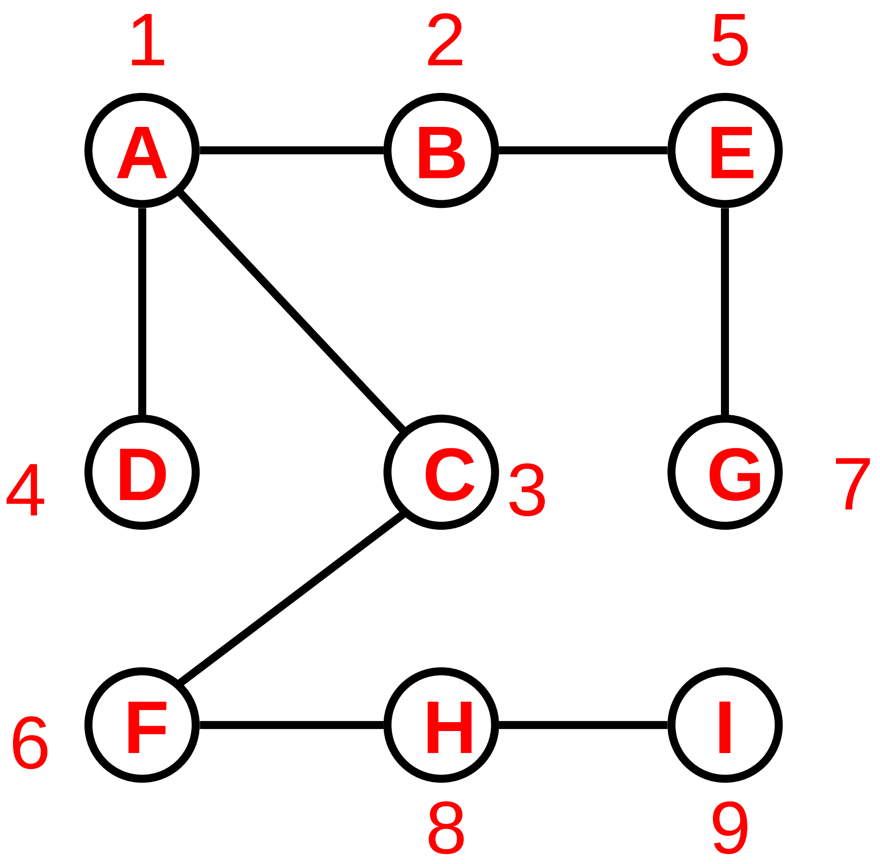
- 广度优先遍历又称为广度优先搜索 (Breadth First Search, BFS)

- 访问策略：首先访问初始点顶点v0，之后依次访问与v0邻接的全部顶点w1，w2，…，wk。然后，再顺次访问与w1，w2，…，wk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。

BFS示例



广度优先搜索过程

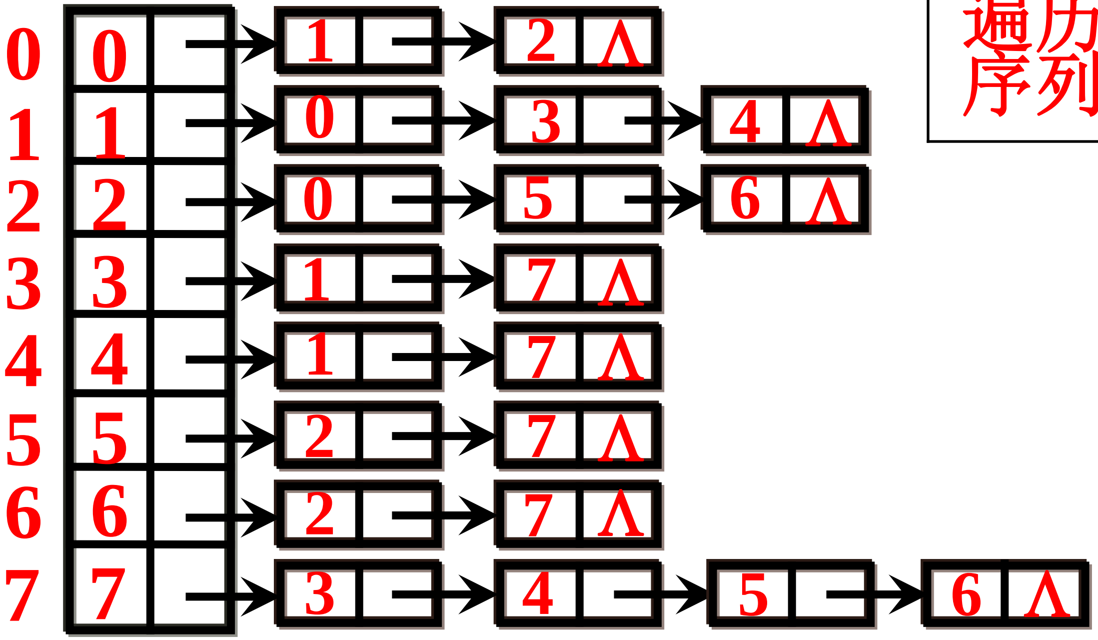


广度优先搜索树

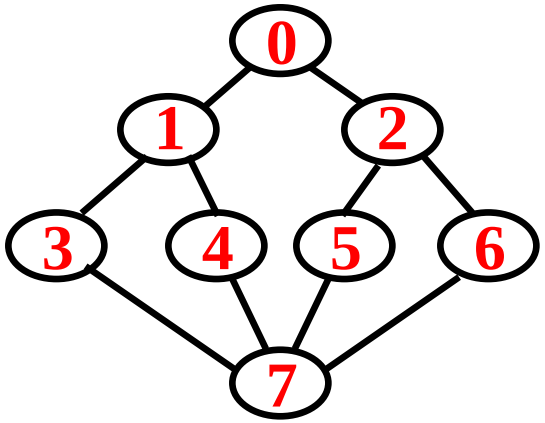
实现分析

- 广度优先搜索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。
- 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。

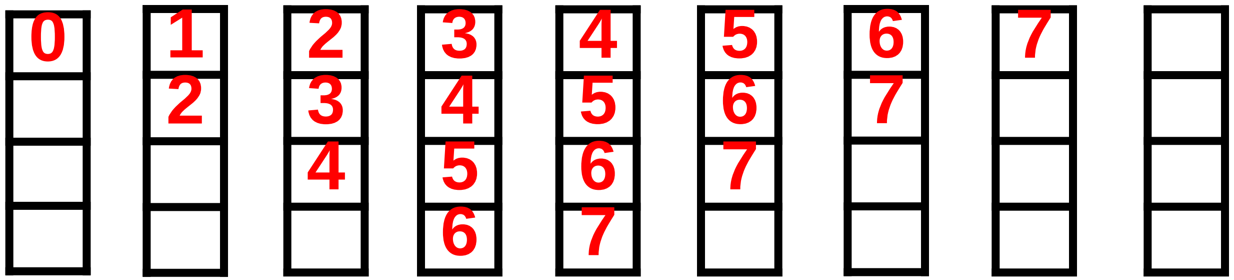
```
算法BFS (v)
 /*连通分支的广度优先遍历算法 */
BFS1[初始化]
    for ( i =1 ; i <= n ; i++ ) visited[i] = 0;
     CREATQuene Q;
     Q.insert(v); visited[v] = 1;
BFS2[广度优先遍历]
    while(! Q.empty()) {   /* 当队列不空时 */
        v=Q.delete()；       /* 出队 */
        cout<<v;
        for ( p = Head[v]->adjacent ; p ; p = p->link ) .
          if ( visited [ p -> VerAdj ] == 0 ){
            Q.insert ( p -> VerAdj) ; visited[p -> VerAdj] = 1;
          }
```



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。



BFS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$O(n^2)$。
- BFS只能遍历一个连通分支。如果遍历非连通图，需要多次调用BFS。与深度优先类似。

n皇后问题

- 在国际象棋中，皇后能攻击她所在行、列或对角线上的任何一个棋子。

- n皇后问题要求在 $n \times n$ 棋盘上放置 n 个皇后，使得没有哪个皇后能攻击其他的皇后

例：4皇后问题

Q	x	x	x
x	x	Q	x
x	x	x	x
x		x	x

Q	x	x	x
x	x	x	Q
x	Q	x	x
x	x	x	x

自上而下，从左到右

第3行不能放，回溯

第4行不能放回溯

X	Q	X	X
X	X	X	Q
Q	X	X	X
X	X	Q	X

算法思想

- 已知成功放置k-1个皇后

- 在第 *k*行放置第 *k*个皇后
  - ✓ 如果$k > n$，输出解，算法结束；
  - ✓ 如果不存在成功的摆放位置，回溯到 *k-1*；否则，选一个位置放置 *k* 个皇后，然后进行下一步：在第k+1行放置第k+1个皇后（递归）

N皇后问题递归算法描述

算法 NQueen (*k*)

/*递归求解n皇后问题。用一个大小为 n 的全局数组 b[ ] 记录皇后放置的位置，b[k]为第 k 个皇后所在的列(第 k 行)*/

NQ1. [递归出口]

```
    if ( k > n ){
        for(i=1;i<=n;i++) cout<<b[i]<<endl; exit(0);
    }
```

NQ2. [放置第k个皇后，递归]

```
    for(i=1;i<=n;i++)
        if(check(k,i)) { b[k] = i; NQueen(k+1);}
```

N皇后问题攻击检测

算法check(k,i)

/*判断第k个皇后可否放在第k行i列，返回值1或0*/

C1. [检查列和对角线]

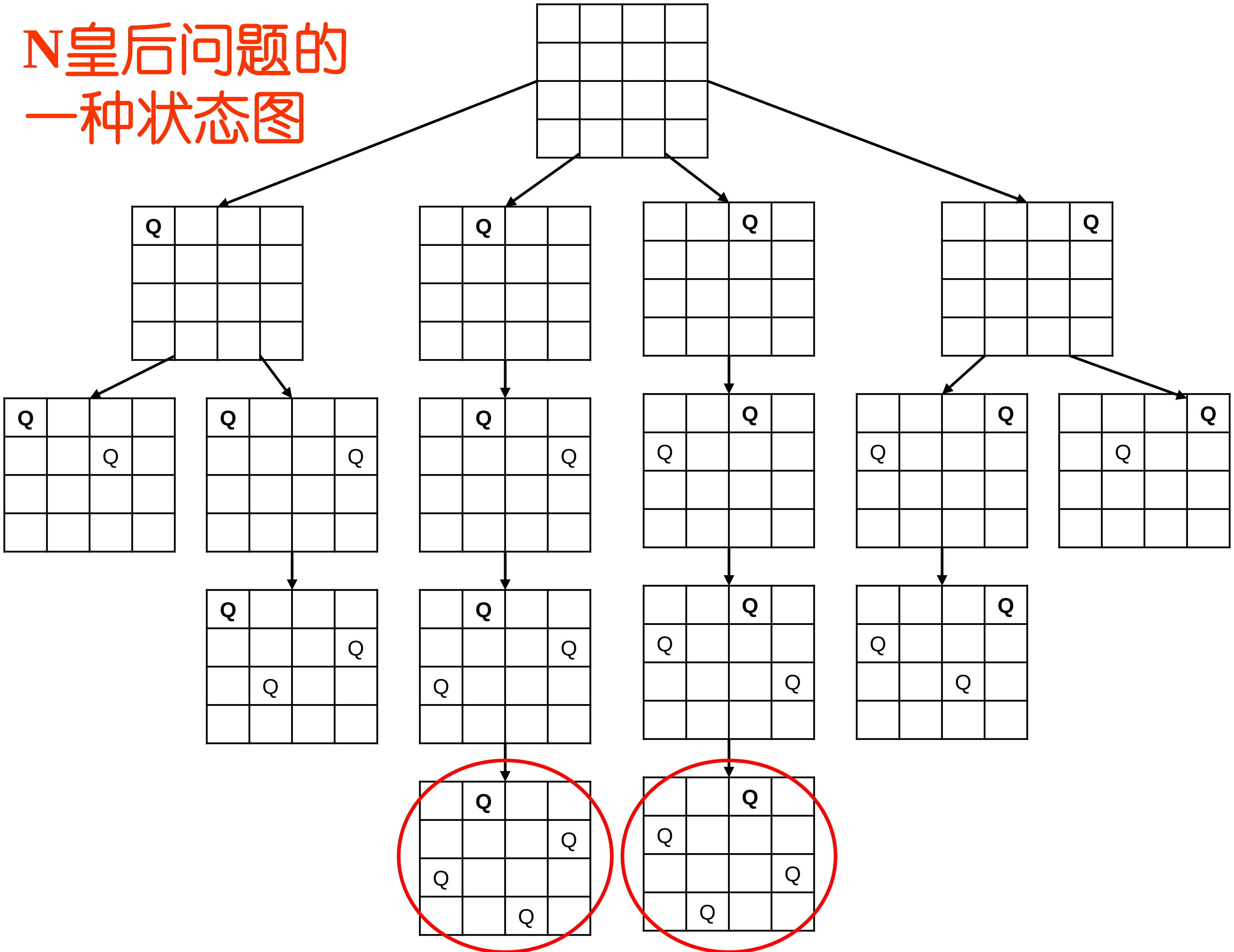
```
for(j=1;j<k;j++)
    if(b[j]==i || abs(k-j)==abs(i-b[j])) return 0;
return 1;
```

/*设有两个皇后分别被放置 (i , j)和(k , l)位置上，那么仅当 $i–j = k–l$ 或 $i+j = k+l$ 时，它们才在同一条斜角线上。将这两个等式分别变换成 $j–l = i–k$ 或 $j–l = k–i$。由此知，当且仅当 $|j–l| = |i–k|$ 时，两个皇后在同一条斜角线上.*/

回溯法-以4皇后为例

- 状态图
  - 每个结点是当前解的状态
  - 每条边代表状态转移(状态的扩展)
  - 初始状态
  - 目标状态(终止状态)
- 回溯法是状态图（隐含图）上的一种深度优先搜索方法

N皇后问题的一种状态图



回溯法的框架

```
void dfs(当前状态S){
    if(当前状态S==目标状态T){
        print();//输出、计数、更新最优；//搜索万能
    }
    else{
      for  each S的可扩展状态Si  do
          if (Si满足约束条件) {
              置当前状态为Si;
              dfs  (Si);
              恢复当前状态为S;
          }
    }
}
```

回溯法的设计要素

- 定义状态
  - ✓ 描述问题求解过程中每一步的状况（可行解空间）；
- 边界条件
  - ✓ 在什么情况下不再递归下去(目标状态，找到解);
- 状态扩展：
  - ✓ 可扩展状态集合（构造解）
- 约束条件：
  - ✓ 扩展出的状态应满足什么条件（剪枝）；

以n皇后问题为例

- 可行解空间：n 皇后问题的解可表为一个 n 元组 ($x_1, \ldots, x_n$)，这里 x_i 系指将第 i 个皇后放在第 i 行的列数且$1 \le x_i \le n$.
- 边界条件：k > n
- 可扩展状态：for(i=1;i<=n;i++)
  - ✓ 第k个位置可能放的列号
- 剪枝：! check(k,i)

回溯法的特点

- 回溯法就是通过递归实现枚举的方法
  - ✓ 通过递归调用逐步构造解
  - ✓ 通过回溯搜索所有解
- 回溯法任何时候只保存一条解路径
- 回溯法时间效率取决于隐含图的结点数。隐含图的结点数取决于状态的定义、扩展规则和剪枝技术。一般来说，隐含图的结点数非常多，通常是问题规模的指数级别。因此，回溯法一般用于问题规模较小时。

第6章 任务

- **慕课**
  - ✓ 在线学习/预习 第 6 章 视频

- **作业**
  - ✓ P213：6-1，6-2，6-13， 6-14，6-15，6-16，6-18
  - ✓ 在线提交