**图的基础知识**

**一、图及图的存储**

<http://www.layz.net/LAOJ/suanfa/s9-1.html>

**1、概念**

图(Graph)是由顶点集合和一些顶点间的连线组成的数据结构。通常可以用G(V, E)来表示。其中**顶点集合(Vertext Set)**和**边的集合(Edge Set)**分别用V(G)和E(G)表示。V(G)中的元素称为顶点(vertex)，用u、v等符号表示。E(G)中的元素称为边(edge)，用e等符号表示。

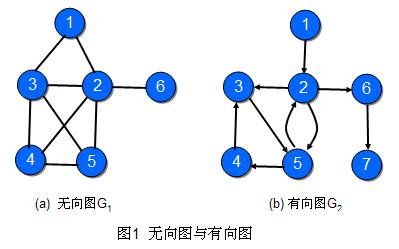
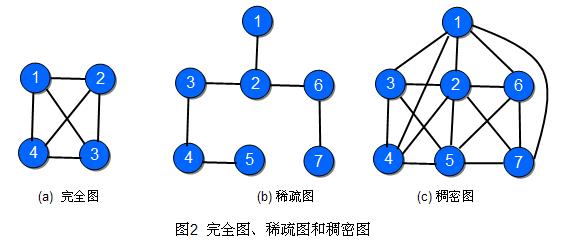


图1(a)所示的图可以表示为G1(V, E)。其中，顶点集合V={1,2,3,4,5,6}，边集合E={(1,2), (1,3), (2,3), (2,4), (2,5), (2,6), (3,4), (3,5), (4,5)}。这样的图叫**无向图**。边(u,v)和(v,u)是同一条边。

图1(b)所示的图是**有向图**。有向边<u,v>中u为起点，v为终点。

如果*无向图中任何一对顶点之间都有一条边，这样的图称为完全图*。在完全图中，顶点数m和边数n的关系为：**m=n×(n-1)/2**。

边的数目相对较少的图称为**稀疏图**。边的数目相对较多的图称为**稠密图**。



在无向图中，如果(u,v)是图中的一条无向边，则称顶点u和顶点v互为**邻接顶点**，或称(u,v)与顶点u和v相**关联**。

**顶点的度数：**一个顶点u的度数是与它相关联的边的数目，记为**deg(u)**。*有向图中，顶点的度数等于该顶点的出度与入度之和*。其中，顶点u的出度是以u为起始顶点的有向边（即从顶点u出发的有向边）的数目，记为**od(u)**。顶点u的入度是以u为终点的有向边（即进入到顶点u的有向边）的数目，记为**id(u)**。顶点u的度数：**deg(u)=od(u) + id(u)**。

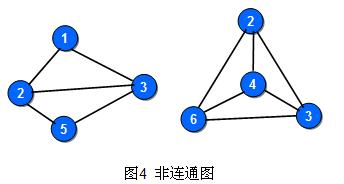
**定理：在无向图和有向图中，所有顶点度数总和，等于边数的两倍。**

这是因为，不管是有向图还是无向图，在统计所有顶点的度数之和时，每条边都被统计两次。为了方便起见，我们把度数为偶数的顶点称为**偶点**，把度数为奇数的顶点称为**奇点**。有以下推论：

**推论：每个图都有偶数个奇点。**

**2、图的连通性**

在无向图中，如果*从顶点u到v有路径*，则称为顶点u和v是**连通**的。*如果无向图中任意一对顶点都是连通的，则称此图为连通图*。相反，如果一个无向图不是连通图，则称为非连通图。

**（图有错）**

如果一个无向图不是连通的，则其**极大连通子图称为连通分量**。这里的极大是指子图中包含的顶点个数极大。例如图4（注意：**图有错**）所示的无向图就是非连通图。其中顶点1,2,3和5构成一个连通分量，顶点4,6,7和8构成另一个连通分量。

在有向图中，若对每一对顶点u和v，即存在从u到v的路径，也存在从v到u的路径，则称此图为**强连通图**。对于非强连通图，其*极大强连通子图称为其强连通分量*。

某些图的边具有与它相关的数，称为权值。这些权值可以表示一个顶点到另一个顶点的距离、花费的代价、所需的时间等等。如果一个图，其所有边都具有权值，则称为加权图，或者称为网络(net)。

**3、图的存储结构**

图的**存储结构**主要有四种：

★ 邻接矩阵(使用二维数组存储，不推荐使用)

★ 前向星

★ 邻接表

★ 链式前向星（**静态建表**）

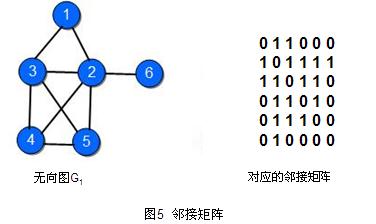
在编写程序中，后三种是我们经常采用的图的存储结构。

**1）邻接矩阵**

在邻接矩阵存储方法中，使用一个二维数组e[n][n]来表示，定义为：



例如图1(a)中，下图表示无向图G1及其邻接矩阵表示。**注意：**如果图中存在**自身环**（连接某个顶点自身的边）和**重边**，多条边的起点一样，终点也一样，也称为平行边的情况，则无法使用邻接矩阵存储。



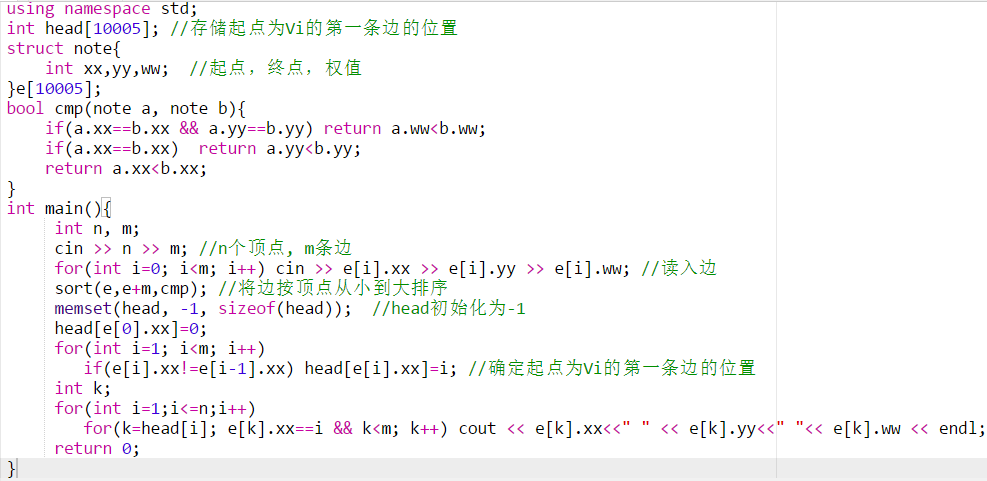
邻接矩阵是表示图的数据结构中最简单也是最常用的一种。

* **实现：**二维数组Map[MAXN][MAXN]，Map[i][j]表示点i到点j的距离。
* **初始化：**Map[i][i] = 0,Map[i][j] = INF(i!=j)，读入数据Map[i][j] = w。
* **时间复杂度：**初始化O(n^2)，建图需要O(m)，总时间复杂度O(n^2)。
* **优缺点：**简单直观，可直接查询点i和点j之间是否有边；遍历效率低，不能存储重编；初始化效率低；大图开销大，适合存储点少的稠密图。

**2）前向星**

前向星是一种*通过存储边的信息的方式存储图的数据结构*。它的构造方式非常简单，读入每条边的信息，将边存放在数组中，把***数组中的边按照起点顺序排序***，前向星就构造完成了。为了查询方便，经常会有一个**数组存储起点为Vi的第一条边的位置**。

前向星一种数据结构，以**储存边的方式来存储图**。构造方法如下：读入每条边的信息将边存放在数组中，然后把数组中的**边按照起点顺序排序**。这样，前向星就构造完了。通常用在点的数目太多，或两点之间有多条弧的时候。一般在别的数据结构不能使用的时候才考虑用前向星。*除了不能直接用起点终点定位以外，前向星几乎是完美的*。

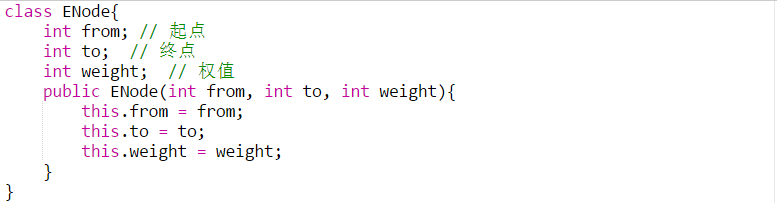


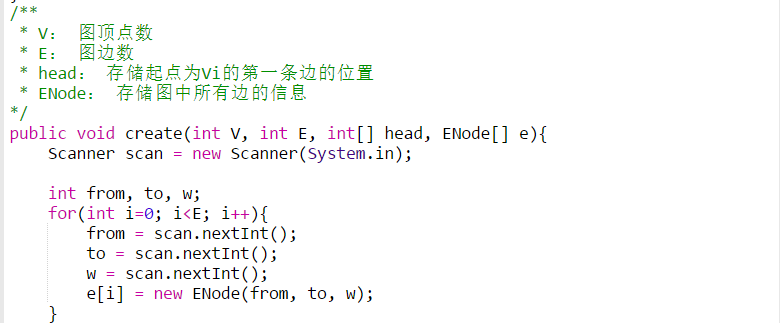
**优化：**

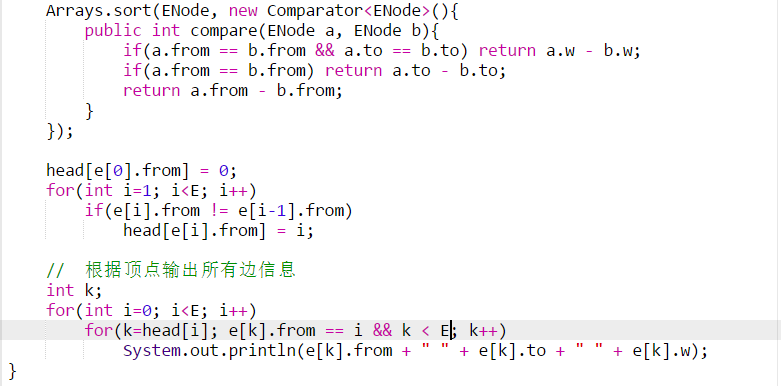
前向星不需要像邻接表那样用指针指向下一条边，还是挺方便的。但是，*由于前向星初始化需要快排一遍，相对邻接表要慢许多*。考虑到一般图论题点数都不会很大，所以可以改为采用**基数排序**的思想对前向星进行排序。

一开始读入时，先算出每个点出去的边有多少条，然后*计算出排序后每个点出去的第一条边位置应在哪里，最后把全部边扫一遍放到排序后应在的位置就好了*。

这样排序的话初始化的时间复杂度就降到了O(m)，总体时间并不会逊色于邻接表。



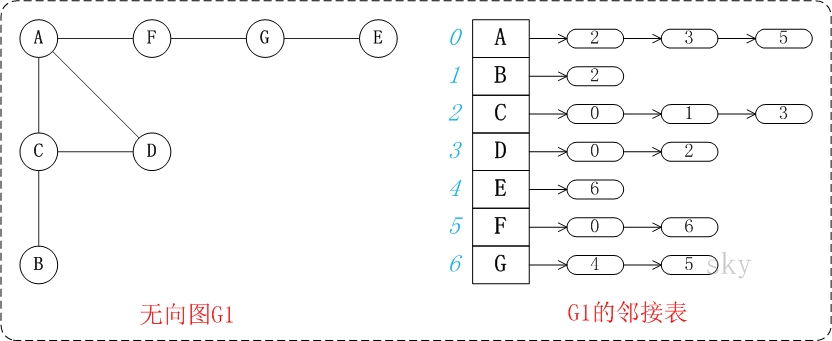




**3）邻接表**

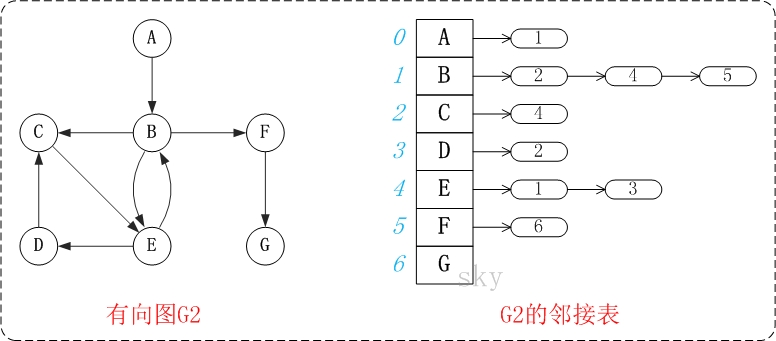
邻接表是一种**链式的存储结构**。对于图G中的每个顶点Vi，*所有邻接于Vi顶点Vj链成一个单链表，这个单链表称为顶点Vi的邻接表*。

邻接表中每个**表节点**有**三个属性**：其一，邻接点序号to，用以存放与顶点Vi 相邻接的顶点vj的序号j；其二，边上的权值；其三，为指针next，用来将邻接表的所有节点链在一起。另外，**为每个顶点Vi的邻接表设置一个具有两个属性的表头节点**：一个是顶点序号from，另一个是指向其**邻接表的指针first**，它是指向Vi的邻接表的第一个节点的指针。建立一个Vnode的数组就可以访问每个顶点的邻接表了。



上面的图G1包含了"A,B,C,D,E,F,G"共7个顶点，而且包含了"(A,C),(A,D),(A,F),(B,C),(C,D),(E,G),(F,G)"共7条边。

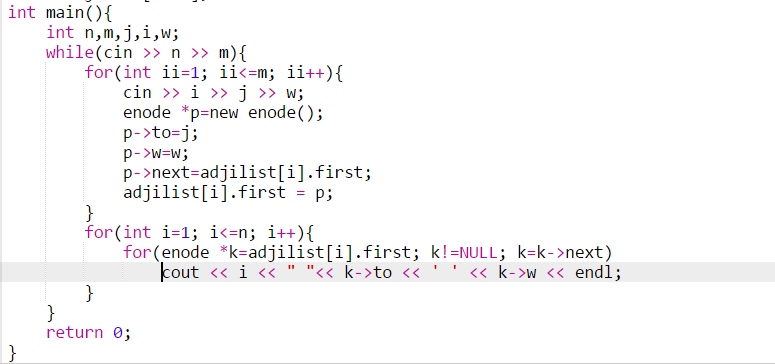
上图右边的矩阵是G1在内存中的邻接表示意图。**每一个顶点都包含一条链表**，该链表记录了**"该顶点的邻接点的序号"**。例如，第2个顶点(顶点C)包含的链表所包含的节点的数据分别是"0,1,3"；而这"0,1,3"分别对应"A,B,D"的序号，"A,B,D"都是C的邻接点。就是通过这种方式记录图的信息的。



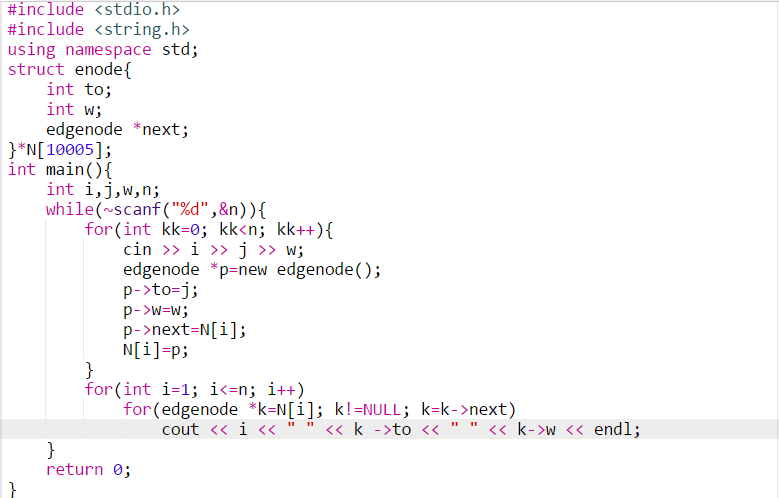
上面的图G2包含了"A,B,C,D,E,F,G"共7个顶点，而且包含了"<A,B>,<B,C>,<B,E>,<B,F>,<C,E>,<D,C>,<E,B>,<E,D>,<F,G>"共9条边。

上图右边的矩阵是G2在内存中的邻接表示意图。*每一个顶点都包含一条链表，该链表记录了"该顶点所对应的出边的另一个顶点的序号"*。例如，第1个顶点(顶点B)包含的链表所包含的节点的数据分别是"2,4,5"；而这"2,4,5"分别对应"C,E,F"的序号，"C,E,F"都属于B的出边的另一个顶点。





另外，还可以只用到一个结构体，接着**结构体数组定义为结构体指针数组**，代码如下：



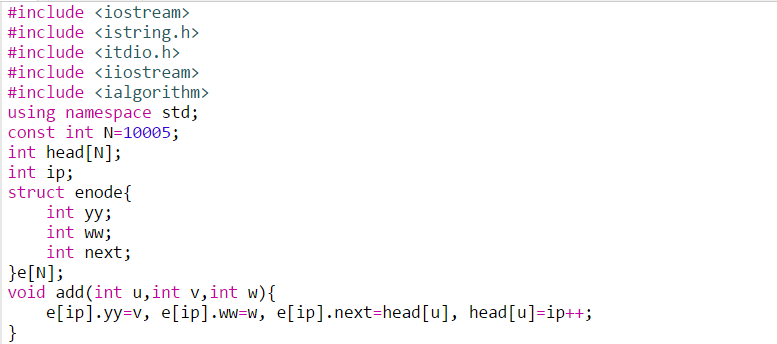
**4）链式前向星**

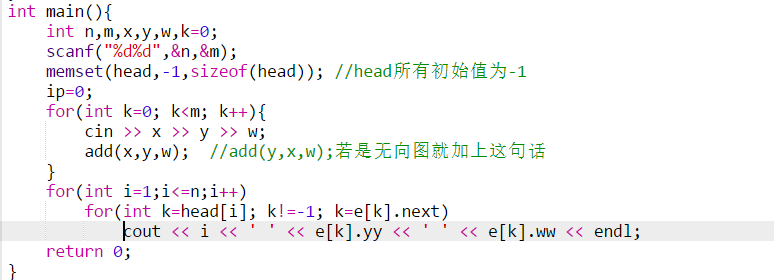
**数组模拟链表**的主要方式是*记录下一个节点在数组中的那个位置*。head数组存储描述vi边信息的链的起点在edge数组的位置。构造**链式前向星**就是*将新加入的节点链在对应链的最开始并修改head数组的对应位置的值*。

链式前向星的建图效率非常高，读入结束建图就结束，时间复杂度为O(m)，空间上使用了两个数组，所以空间复杂度为O(m+n)。

缺点：**不能用起点和终点来确定是否有边存在**。。。。

链式前向星又称为**邻接表的静态建表方式**，其最开始确实是基于前向星，是以提高前向星的构造效率为目的设计的存储方式，最终演变成了一个变形的邻接表这一数据结构。*链式前向星采用数组模拟链表的方式实现邻接表的功能*（**数组模拟链表**的主要方式就是记录下一个节点在数组的哪个位置），并且使用很少的额外空间，可以说是目前建图和遍历效率最高的存储方式了。





**程序说明：**

假如在上面的程序中，读入如下数据：

7 10

5 6 8

6 7 7

2 5 5

3 5 6

3 6 4

4 6 10

4 7 5

1 2 2

3 1 3

1 4 3

在**读入数据中**，从顶点3出发的边共有3条，依次为(3,5)、(3,6)和(3,1)。

* 读入边(3,5)时（说明：第4次读入），**e[3].yy=5**，表示编号为3的边的终点为5。
* e[3].ww=6，表示编号为3的边的权值为6。
* e[3].next=head[u]=head[3]=-1（初始为-1），表示*编号为3的边只有一条，没有上一条边*。
* 对应的head[3]=3，表示以顶点3为起点的第一条边的编号为3。
* 读入边(3,6)时（说明：第5次读入），e[4].yy=6，表示编号为3的边的终点为6。
* e[4].ww=4，表示编号为3的边的权值为4。
* e[4].next=head[u]=head[3]=3，表示编号为4的上一条边的编号为3。
* 对应的head[3]=4，表示以顶点3为起点的第一条边的编号为4。
* 读入边(3,1)时（说明：第8次读入），e[7].yy=1，表示编号为7的边的终点为1。
* e[7].ww=3，表示编号为3的边的权值为3。
* e[7].next=head[u]=head[3]=4，表示编号为7的上一条边的编号为4。
* 对应的head[3]=7，表示***以顶点3为起点的第一条边的编号为7***。
* 最终，head[3]=7

**5）深度理解链式前向星**

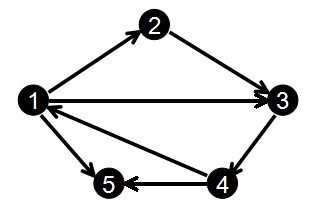
<http://blog.csdn.net/acdreamers/article/details/16902023>

前向星是一种**特殊的边集数组**，我们把边集数组中的每一条边按照起点从小到大排序,如果起点相同就按照终点从小到大排序，并记录下以某个点为起点的所有边在数组中的起始位置和存储长度，那么前向星就构造好了。

用len[i]来记录所有***以i为起点的边在数组中的存储长度***。

用head[i]记录***以i为边集在数组中的第一个存储位置***。

**举例分析如下：**



输入边的顺序为:

1 2

2 3

3 4

1 3

4 1

1 5

4 5

那么排完序后就得到:

编号 ： 1 2 3 4 5 6 7

起点u： 1 1 1 2 3 4 4

终点v： 2 3 5 3 4 1 5

得到:

head[1] = 1 len[1] = 3

head[2] = 4 len[2] = 1

head[3] = 5 len[3] = 1

head[4] = 6 len[4] = 2

但是利用前向星会有排序操作,如果用快排时间至少为O(nlog(n))

如果用**链式前向星，就可以避免排序**。我们建立边结构体为:

struct Edge{

int next;

int to;

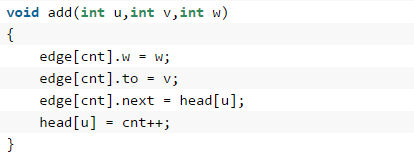
int w;

};

其中edge[i].to表示第i条边的终点，edge[i].next表示*与第i条边同起点的下一条边的存储位置*，edge[i].w为边权值.

另外还有一个数组head[]，它是用来***表示以i为起点的第一条边存储的位置***，实际上你会发现这里的*第一条边存储的位置其实在以i为起点的所有边的最后输入的那个编号*。

head[]数组一般初始化为-1，对于加边的add函数是这样的：



初始化cnt = 0,这样,现在我们还是按照上面的图和输入来模拟一下:

edge[0].to = 2; edge[0].next = -1; head[1] = 0;

edge[1].to = 3; edge[1].next = -1; head[2] = 1;

edge[2].to = 4; edge[2],next = -1; head[3] = 2;

edge[3].to = 3; edge[3].next = 0; head[1] = 3;

edge[4].to = 1; edge[4].next = -1; head[4] = 4;

edge[5].to = 5; edge[5].next = 3; head[1] = 5;

edge[6].to = 5; edge[6].next = 4; head[4] = 6;

很明显，head[i]保存的是以i为起点的所有边中编号最大的那个，而**把这个当作顶点i的第一条起始边的位置**。这样在遍历时是**倒着遍历的**，也就是说与输入顺序是相反的，不过这样不影响结果的正确性。比如以上图为例，以节点1为起点的边有3条，它们的编号分别是0、3、5，而head[1] = 5。

我们在遍历以u节点为起始位置的所有边的时候是这样的:

**for(int i=head[u];~i;i=edge[i].next)**

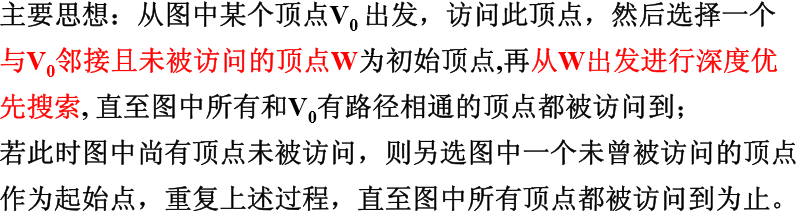
那么就是说先遍历编号为5的边，也就是head[1]，然后就是edge[5].next，也就是编号3的边，然后继续edge[3].next，也就是编号0的边，可以看出是**逆序**的。

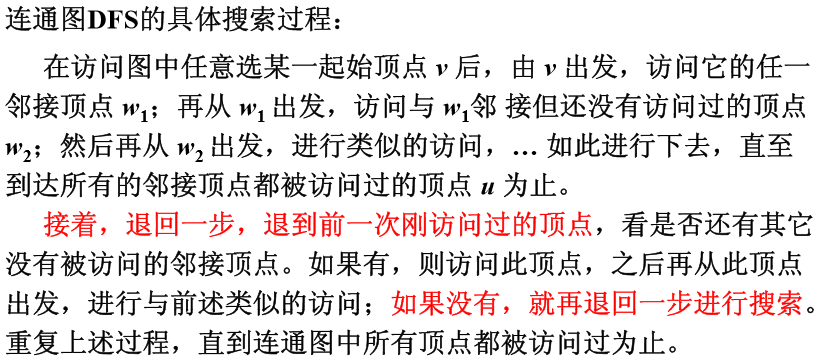
**二、图的遍历**

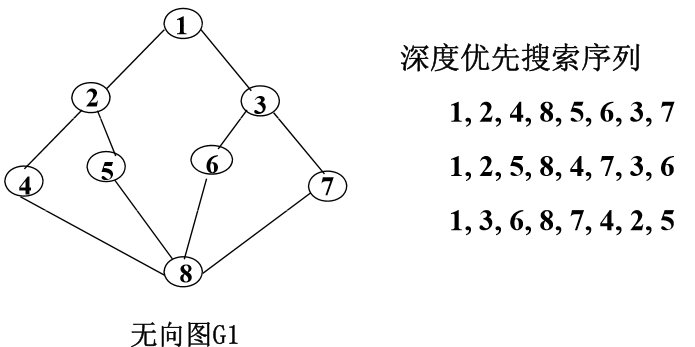
<http://www.jianshu.com/p/2469a4d9708e>

**1、深度优先遍历搜索DFS**

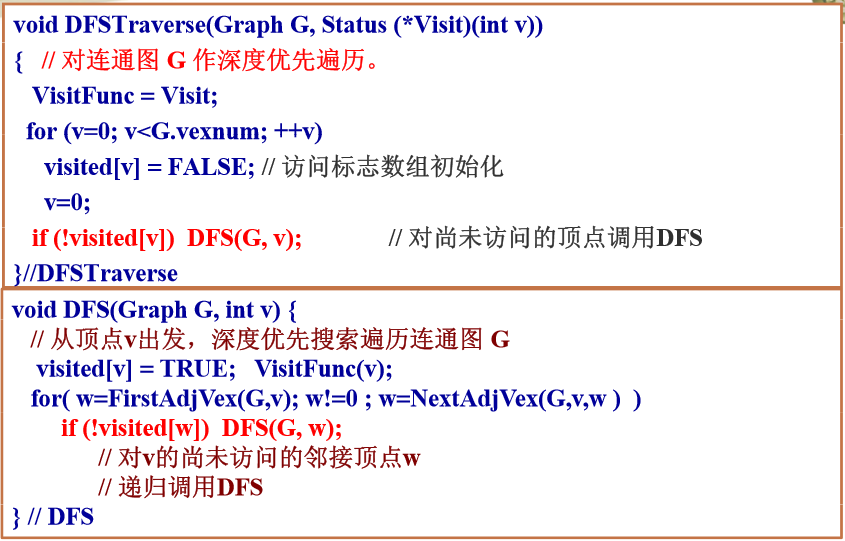
从图中某个顶点v出发，访问此顶点，然后*从v的未被访问的邻接点出发深度优先遍历图，直至图中所有和v有路径相通的顶点都被访问到*。



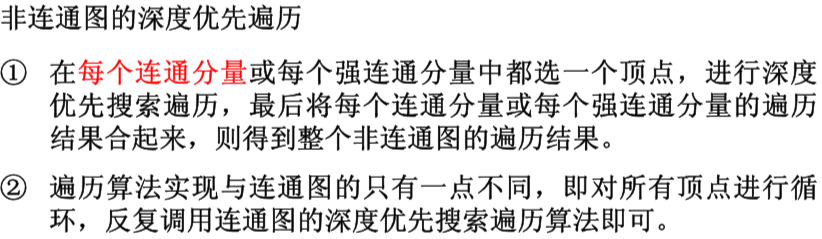


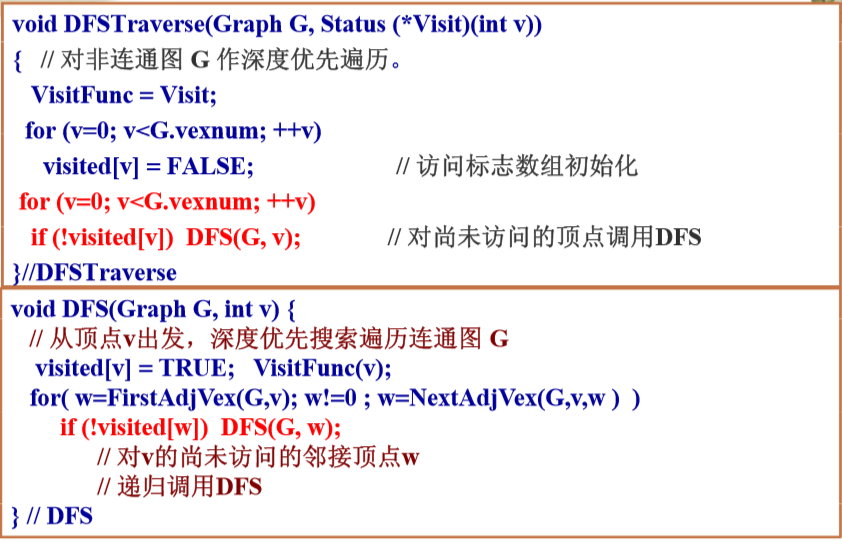


**1）连通图**



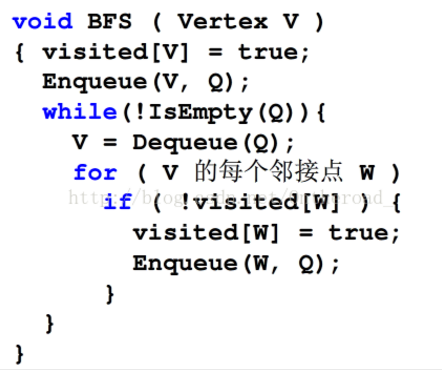
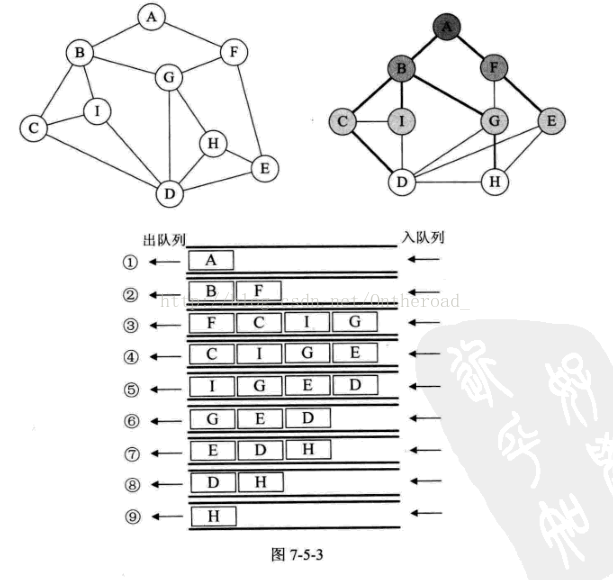
**2）非连通图**

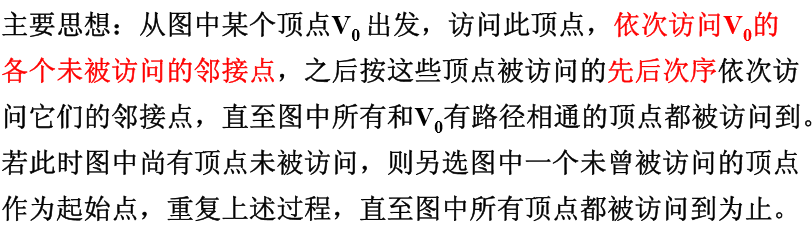




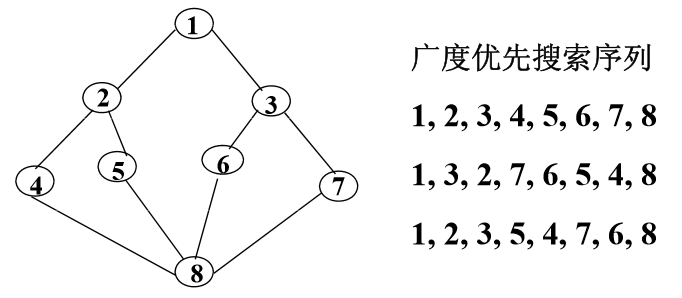
**2、广度优先遍历搜索BFS**

类似于**树的层次遍历**

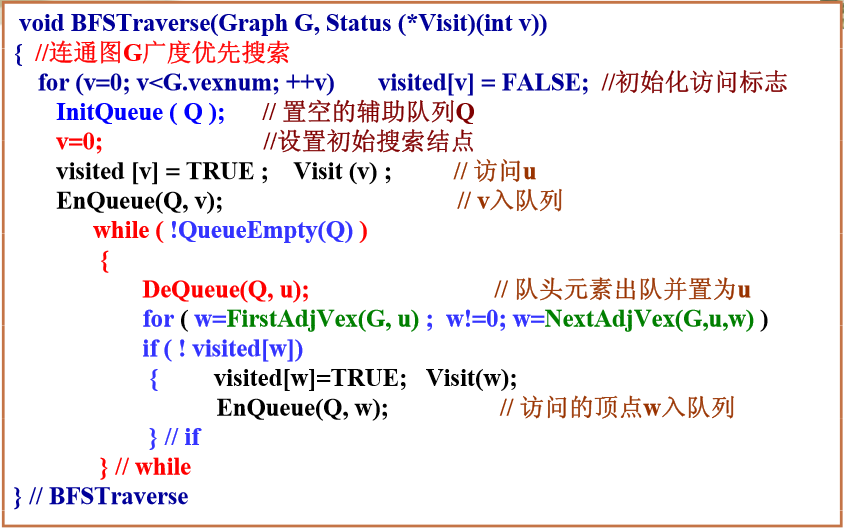




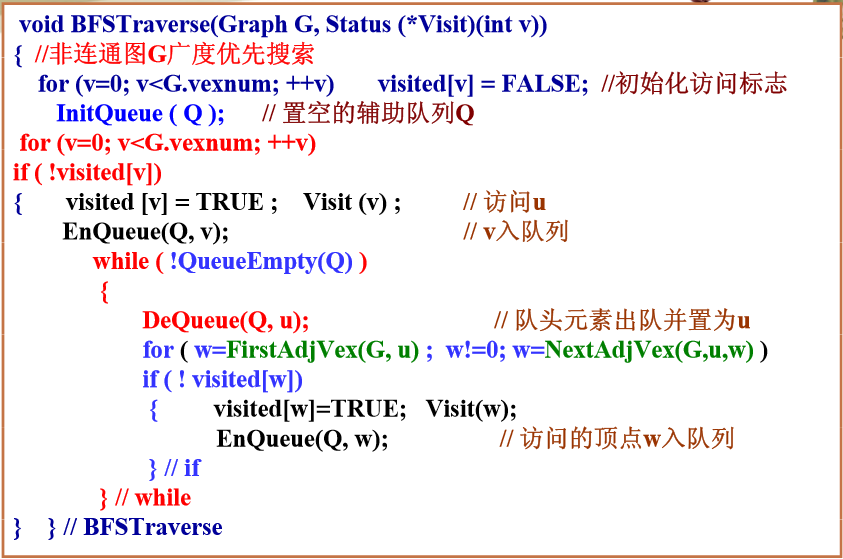
**广度优先搜索是一种分层的搜索过程**，每向前走一步可能访问一批顶点，不像深度优先搜索那样有回退的情况。因此，广度优先搜索不是一个递归的过程，其算法也不是递归的。



**1）连通图**



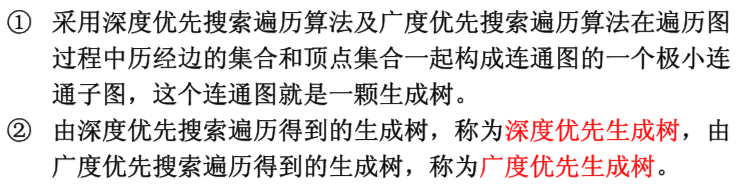
**2）非连通图**

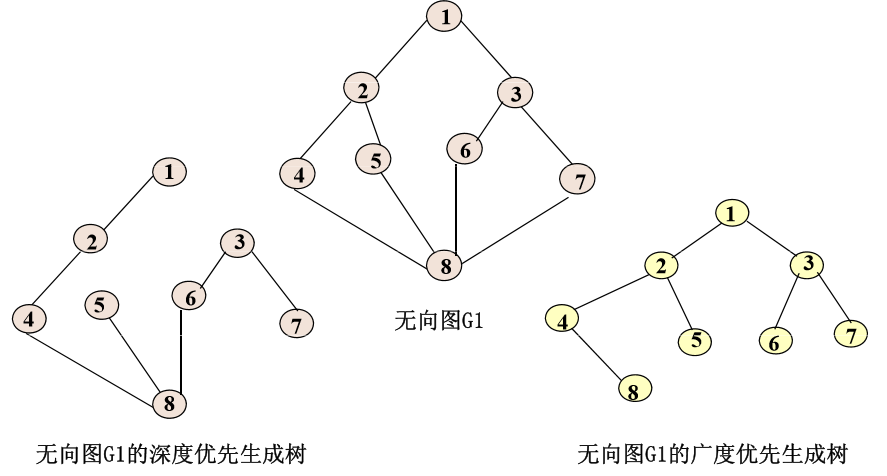


**三、无向图的连通分量和生成树**

**1、连通生成树**

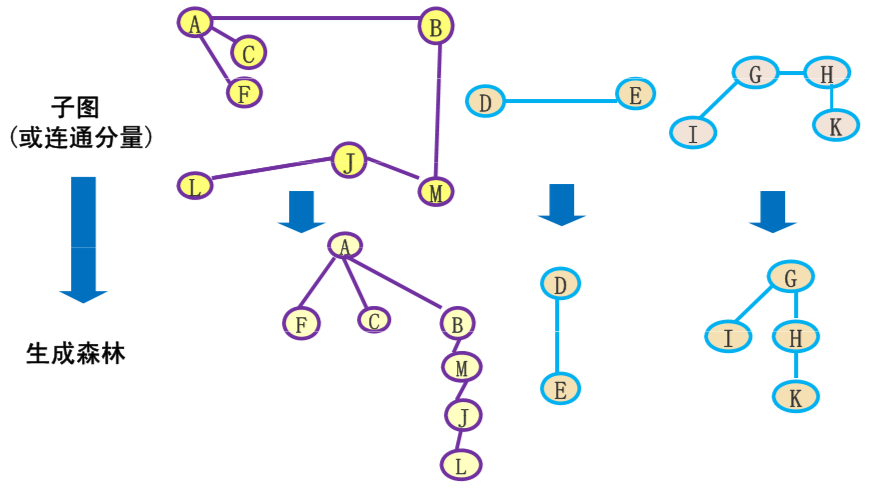
一个连通图的生成树是一个极小连通子图，它含有图中全部顶点，但只有构成一棵树的n-1条边。



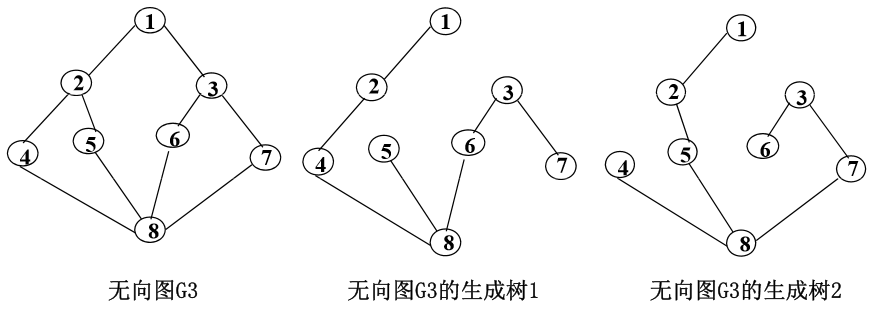


**2、非连通生成树**

当无向图为非连通图时，从图中某一顶点除法，利用深度优先搜索算法或广度优先搜索算法不可能遍历到图中的所有顶点，只能访问到该顶点所在的极大连通子图的所有顶点，该极大连通子图称为无向图的一个**连通分量**。

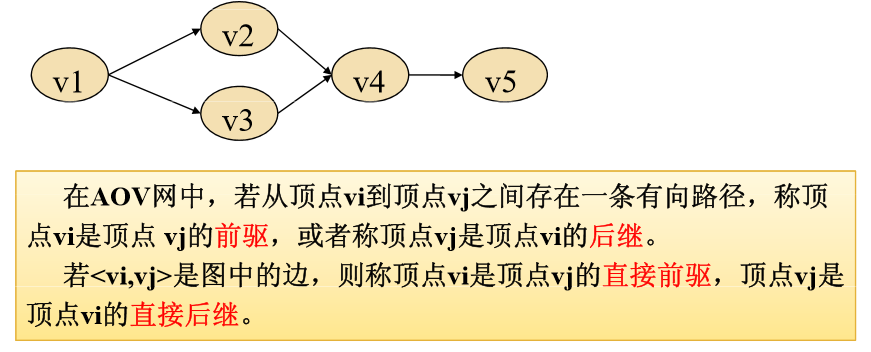


使用不同的遍历图的方法，可以得到不同的生成树；从不同的顶点出发，也可能得到不同的生成树。



**四、有向无环图**

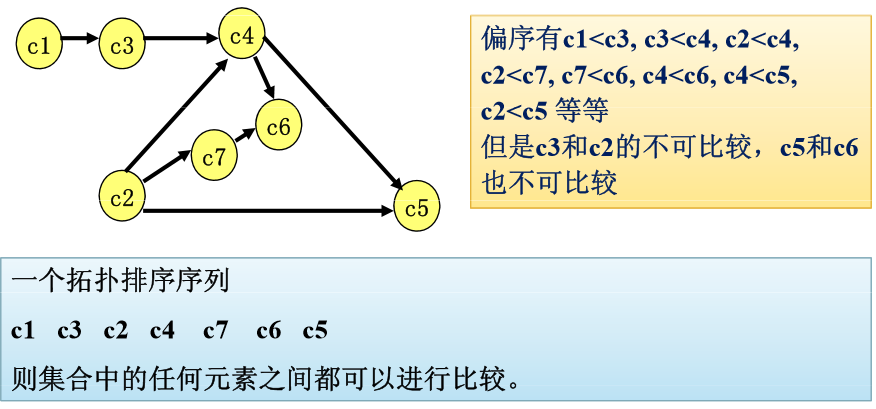
在一个有向图中，若用顶点表示活动，有向边表示活动间的先后关系，称该有向图叫做顶点表示活动的网络，简称AOV网。



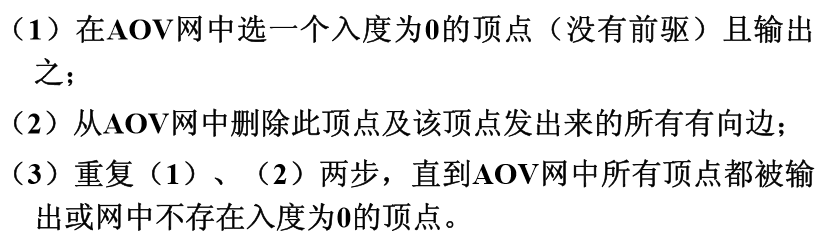
**1、拓扑排序**

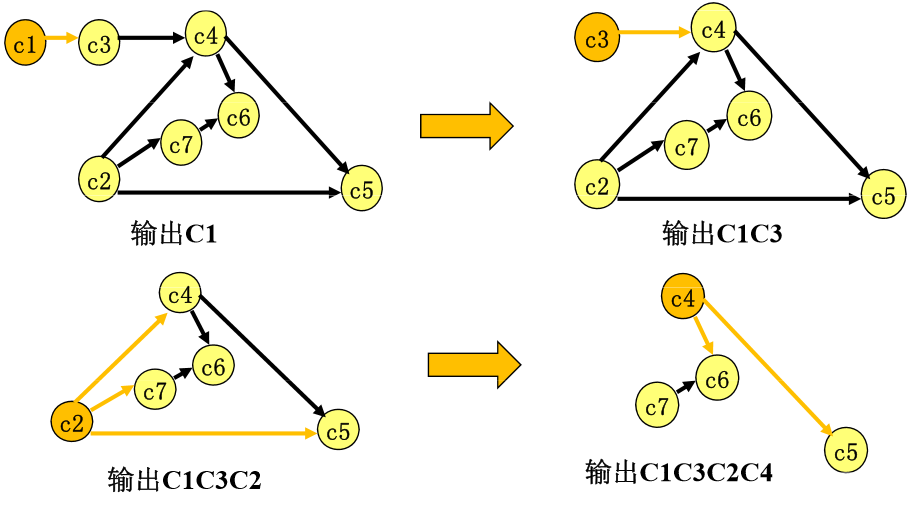
按照有向图给出的次序关系，将图中顶点排成一个线性序列，对于有向图中没有限定次序关系的顶点，则可以人为加上任意的次序关系。由此所得顶点的线性序列成为拓扑有序序列。

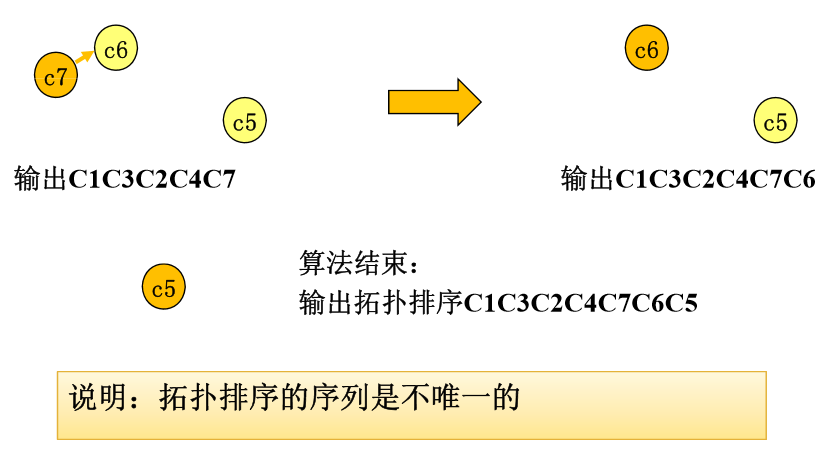
**拓扑排序：**由某个集合上的一个**偏序（集合中仅有部分成员之间可以比较）**得到该集合上的一个**全序（集合中全体成员之间均可以比较）**的操作称为拓扑排序。



**拓扑排序算法步骤：**

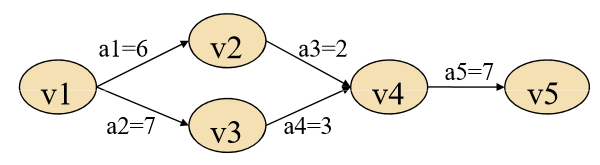


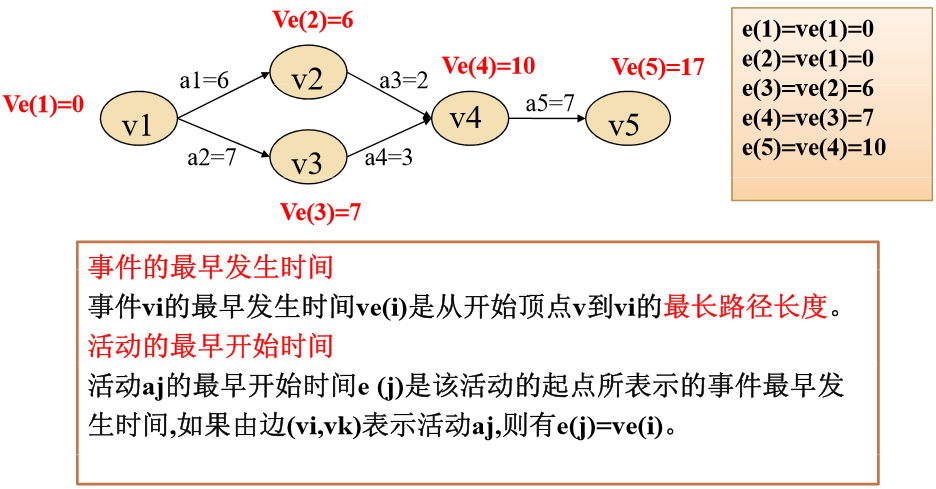


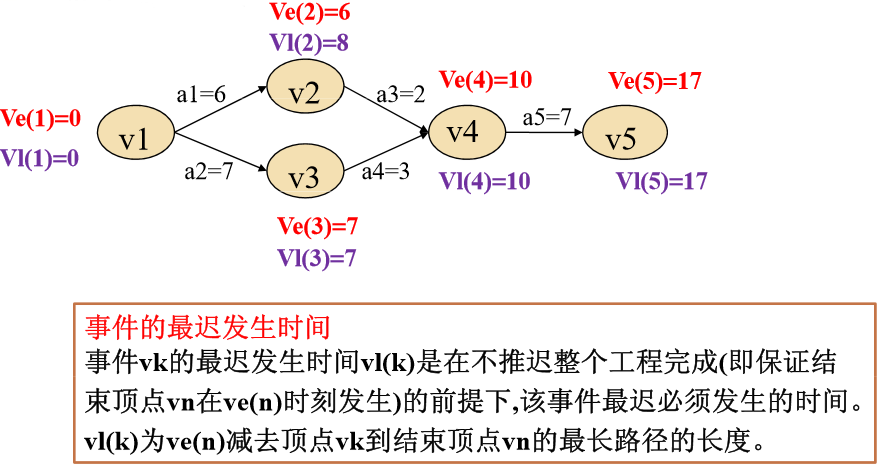


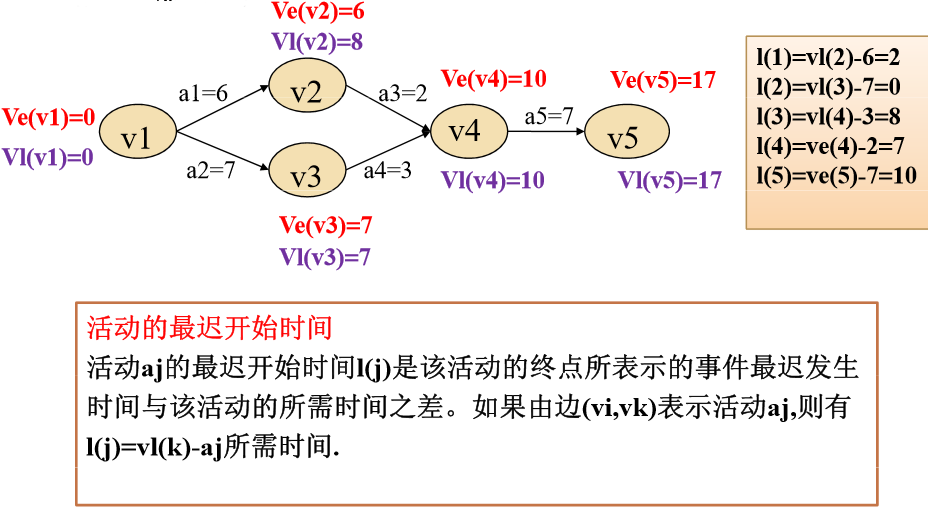
**2、关键路径**

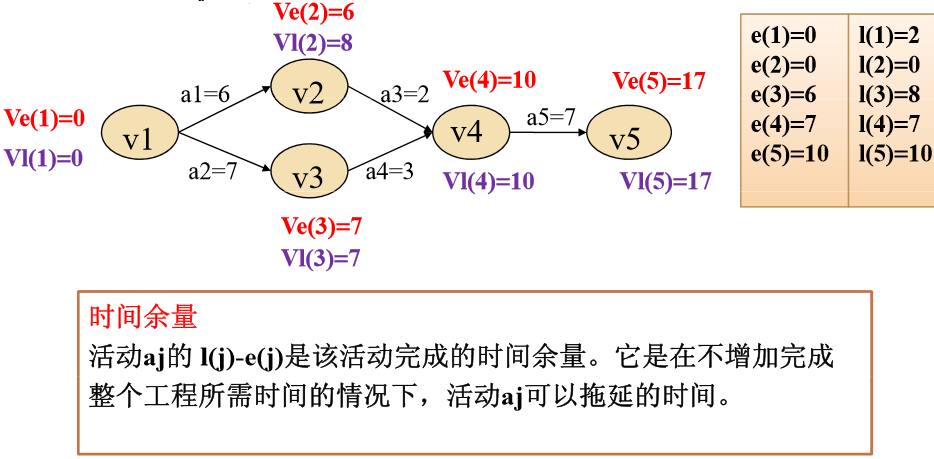
在一个有向图中，若用***顶点表示事件，弧表示活动，权表示活动持续时间***，称该有向图叫做边表示活动的网络，简称为AOE网。

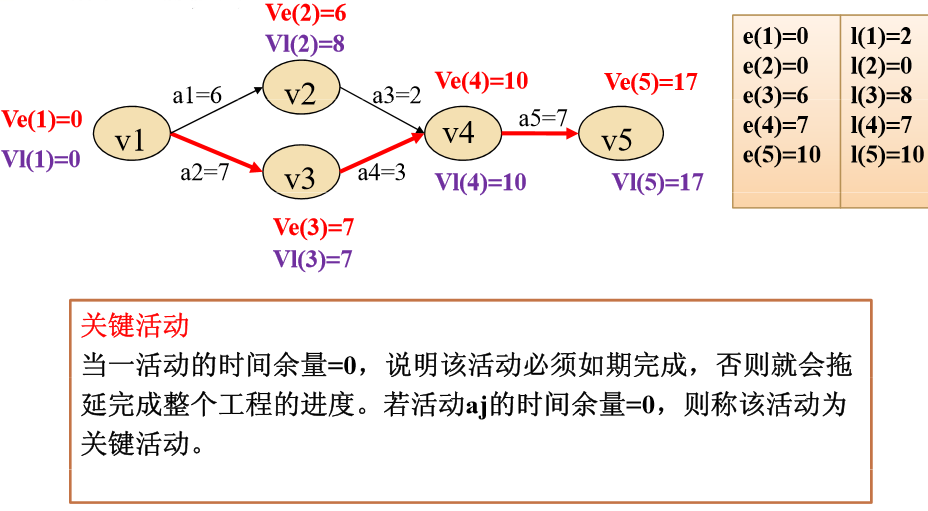












拓扑排序算法演示：

