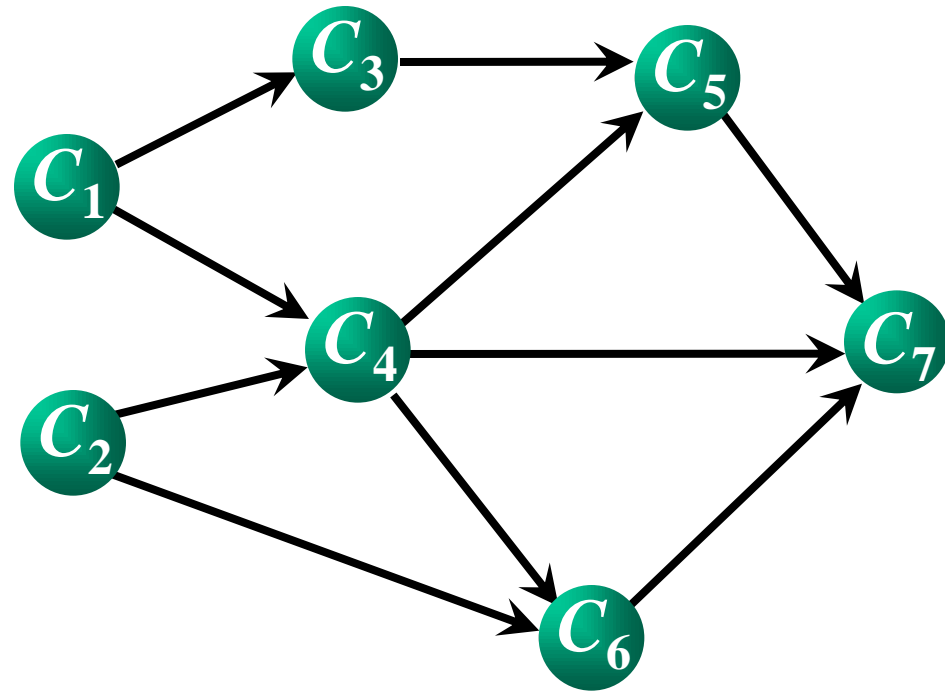


教学计划编排问题



要完成什么功能？如何表示课程之间的先修关系？抽象出的模型是什么？

编号	课程名称	先修课
C ₁	高等数学	无
C ₂	计算机导论	无
C ₃	离散数学	C ₁
C ₄	程序设计	C ₁ , C ₂
C ₅	数据结构	C ₃ , C ₄
C ₆	计算机原理	C ₂ , C ₄
C ₇	数据库原理	C ₄ , C ₅ , C ₆



第六章 图

本章的主要内容是：

6.1 图的基本概念

6.2 图的遍历

6.3 图的存储结构

6.4 图的连通性

6.5 图的应用举例

- 最小生成树

- 拓扑排序

6.1 图的基本概念

图(graph)是由顶点(vertex)的有穷非空集合和顶点之间边(edge)的集合组成，通常表示为：

$$\underline{G=(V, E)}$$

其中： G 表示一个图， V 是图 G 中顶点的集合， E 是图 G 中顶点之间边的集合。

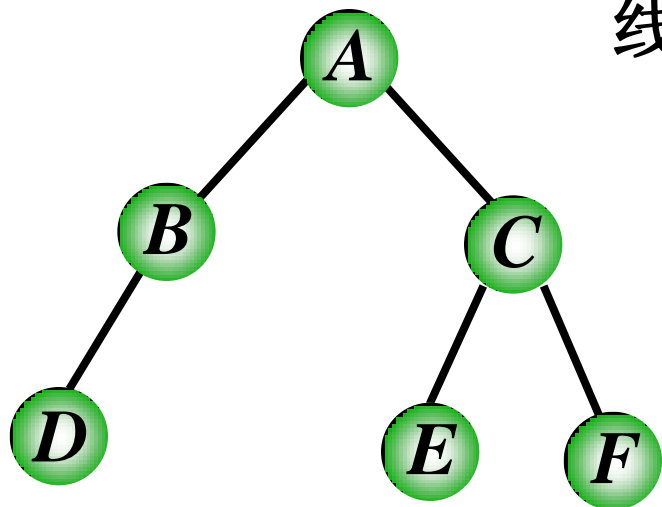
在线性表中，元素个数可以为零，称为空表；

在树中，结点个数可以为零，称为空树；

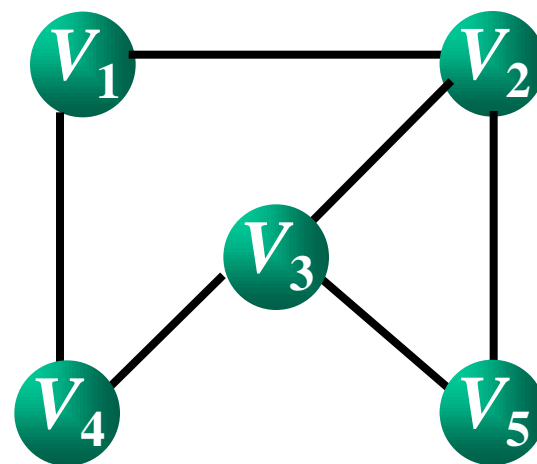
在图中，顶点个数不能为零，但可以没有边。



线性结构

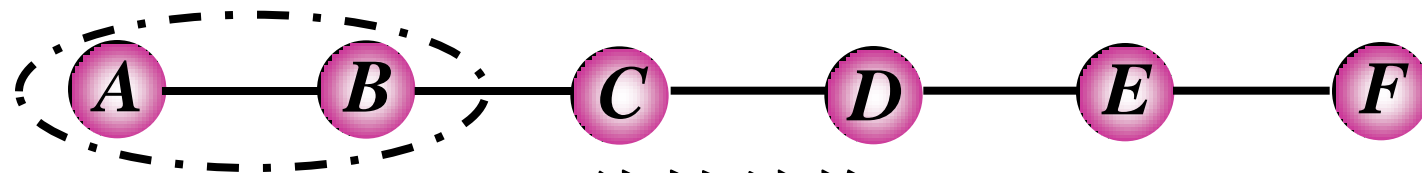


树结构

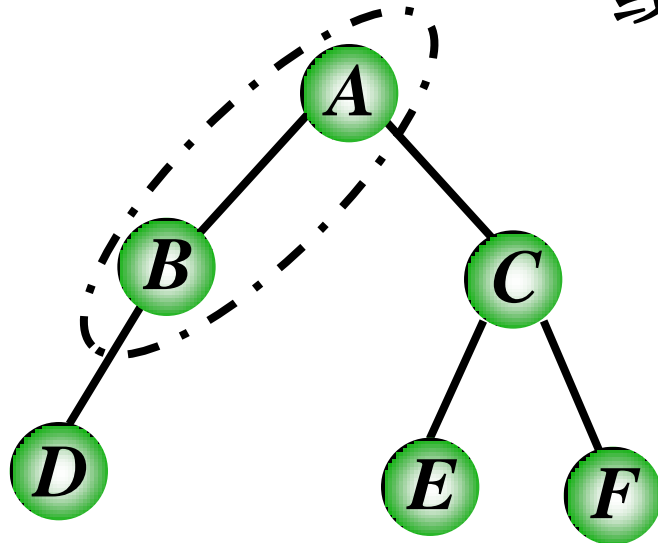


图结构

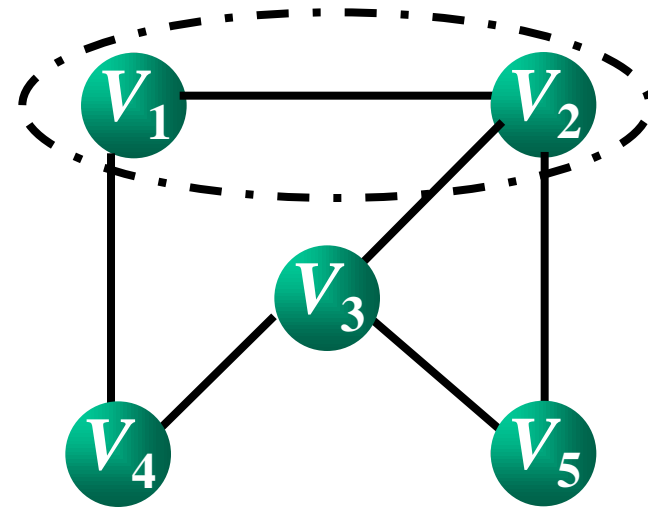
在线性结构中，数据元素之间仅具有线性关系；
在树结构中，结点之间具有层次关系；
在图结构中，任意两个顶点之间都可能有关系。



线性结构

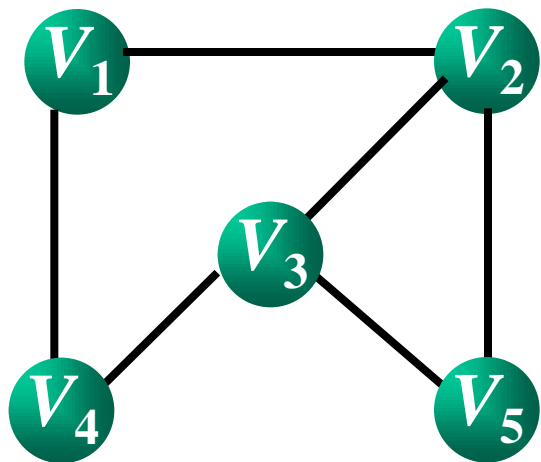


树结构



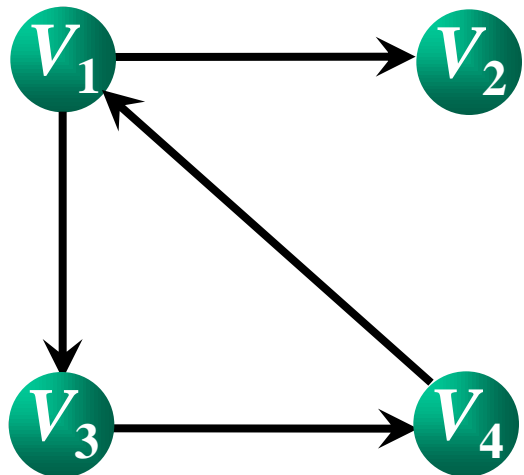
图结构

在线性结构中，元素之间的关系为**前驱**和**后继**；
在树结构中，结点之间的关系为**双亲**和**孩子**；
在图结构中，顶点之间的关系为**邻接**。



若顶点 v_i 和 v_j 之间的边没有方向，则称这条边为**无向边**，表示为 (v_i, v_j) 。

如果图的任意两个顶点之间的边都是无向边，则称该图为**无向图** (undirected graph)。

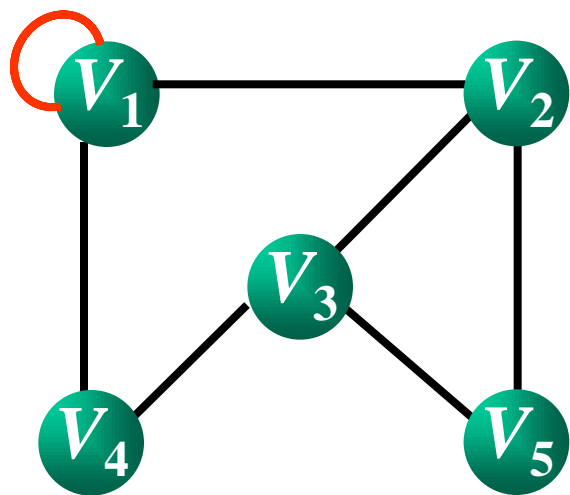


若从顶点 v_i 到 v_j 的边有方向，则称这条边为**有向边**，表示为 $\langle v_i, v_j \rangle$ 。

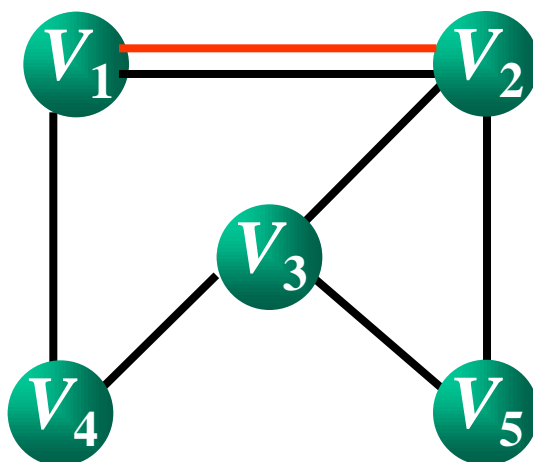
如果图的任意两个顶点之间的边都是有向边，则称该图为**有向图** (directed graph)。

图的基本术语

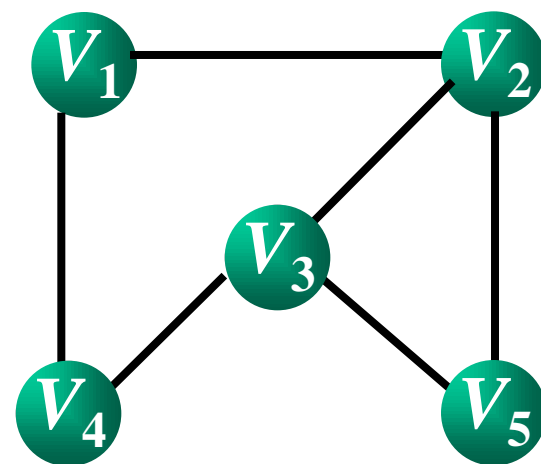
简单图(simple graph): 在图中, 若不存在顶点到其自身的边, 且同一条边不重复出现。



非简单图



非简单图



简单图

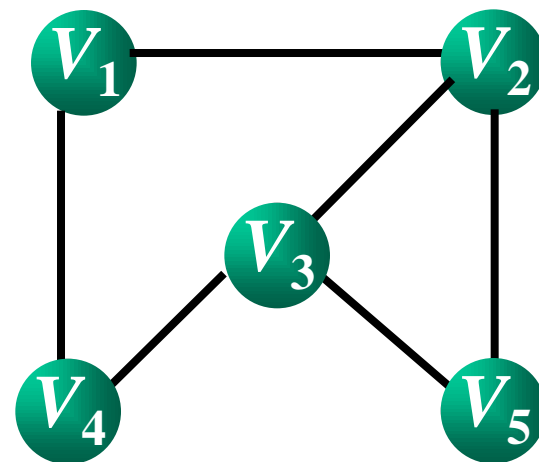
❖ 数据结构中讨论的都是简单图。

邻接(adjacent)、依附(adhere)

无向图中，对于任意两个顶点 v_i 和顶点 v_j ，若存在边 (v_i, v_j) ，则称顶点 v_i 和顶点 v_j 互为邻接点，同时称边 (v_i, v_j) 依附于顶点 v_i 和顶点 v_j 。

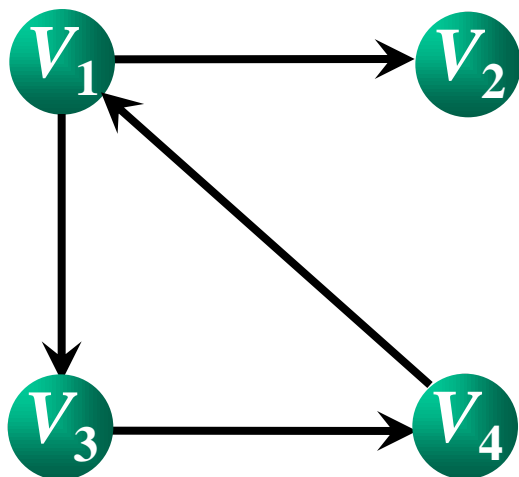
V_1 的邻接点: V_2, V_4

V_2 的邻接点: V_1, V_3, V_5



邻接、依附

有向图中，对于任意两个顶点 v_i 和顶点 v_j ，若存在弧 $\langle v_i, v_j \rangle$ ，则称顶点 v_i 邻接到顶点 v_j ，顶点 v_j 邻接自顶点 v_i ，同时称弧 $\langle v_i, v_j \rangle$ 依附于顶点 v_i 和顶点 v_j 。

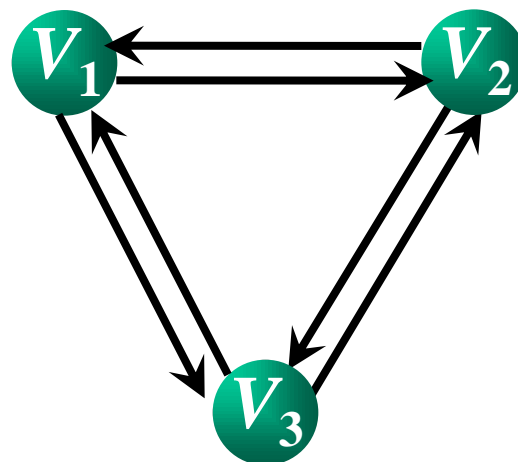
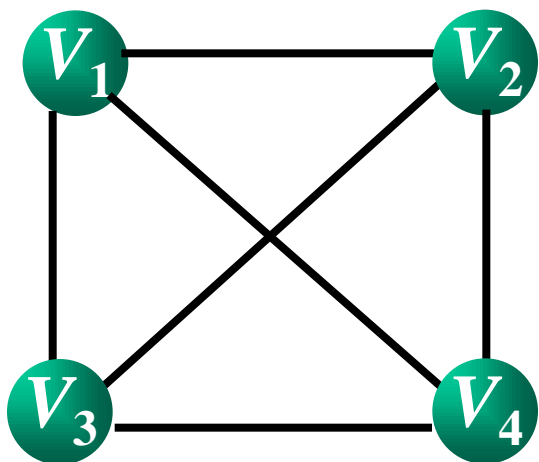


V_1 的邻接点: V_2 、 V_3

V_3 的邻接点: V_4

无向完全图(undirected complete graph): 在无向图中, 如果任意两个顶点之间都存在边, 则称该图为无向完全图。

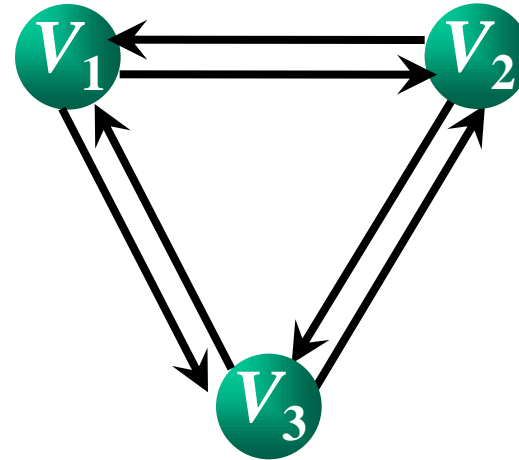
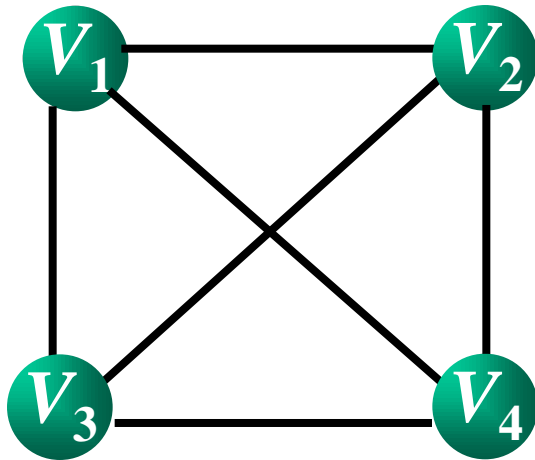
有向完全图(directed complete graph): 在有向图中, 如果任意两个顶点之间都存在方向相反的两条弧, 则称该图为有向完全图。





含有 n 个顶点的无向完全图有多少条边？

含有 n 个顶点的有向完全图有多少条弧？



含有 n 个顶点的无向完全图有 $n \times (n-1)/2$ 条边。

含有 n 个顶点的有向完全图有 $n \times (n-1)$ 条边。

稀疏图(sparse graph): 称边数很少的图为稀疏图;

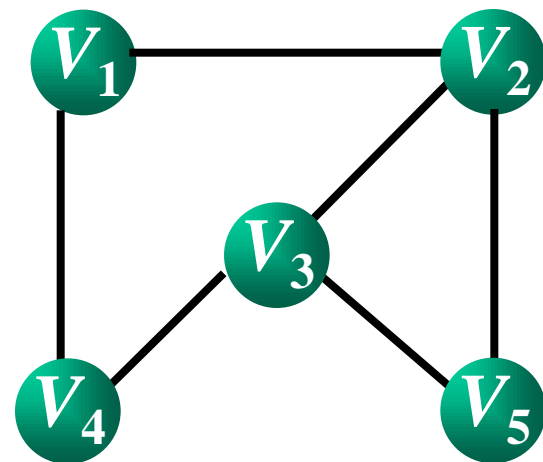
稠密图(dense graph): 称边数很多的图为稠密图。

顶点的度(degree): 在无向图中, 顶点 v 的**度**是指依附于该顶点的边数, 通常记为 $TD(v)$ 。

顶点的入度(in-degree): 在有向图中, 顶点 v 的**入度**是指以该顶点为弧头的弧的数目, 记为 $ID(v)$;

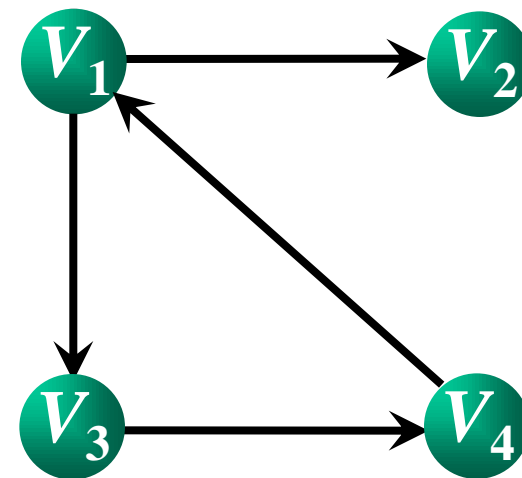
顶点的出度(out-degree): 在有向图中, 顶点 v 的**出度**是指以该顶点为弧尾的弧的数目, 记为 $OD(v)$ 。

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



在具有 n 个顶点、 e 条边的无向图 G 中，各顶点的度之和与边数之和的关系？

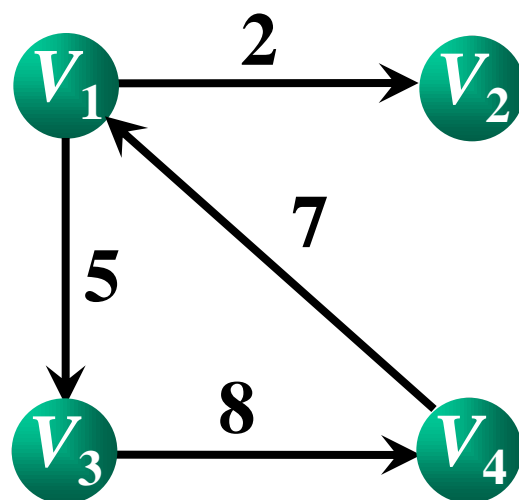
$$\sum_{i=1}^n ID(v_i) = \sum_{i=1}^n OD(v_i) = e$$



在具有 n 个顶点、 e 条边的有向图 G 中，各顶点的入度之和与各顶点的出度之和的关系？与边数之和的关系？

权 (weight): 是指对边赋予的有意义的数值量。

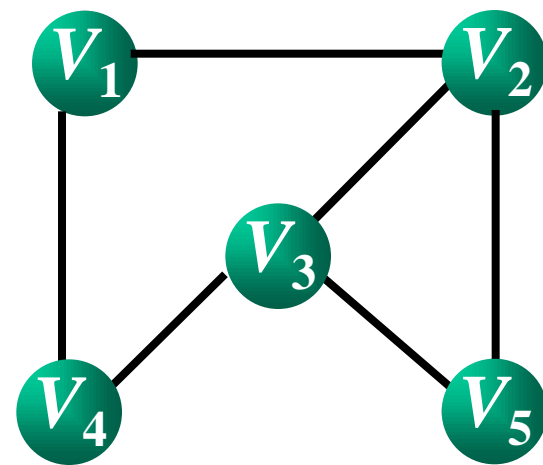
网 (network): 边上带权的图，也称网图。



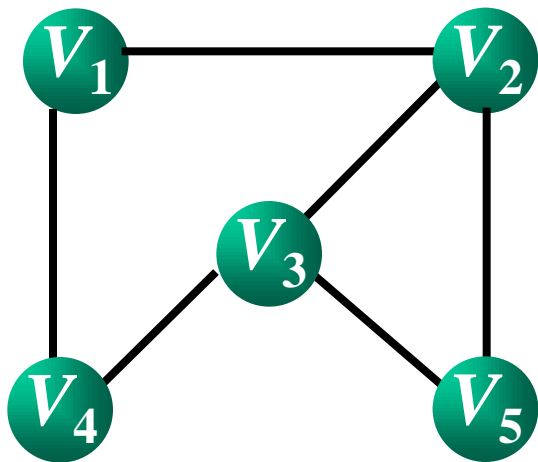
路径：在无向图 $G=(V, E)$ 中，从顶点 v_p 到顶点 v_q 之间的**路径**是一个顶点序列 $(v_p=v_{i0}, v_{i1}, v_{i2}, \dots, v_{im}=v_q)$ ，其中， $(v_{ij-1}, v_{ij}) \in E$ ($1 \leq j \leq m$)。若 G 是有向图，则路径也是有方向的，顶点序列满足 $\langle v_{ij-1}, v_{ij} \rangle \in E$ 。

V_1 到 V_4 的路径：
 $V_1 V_4$
 $V_1 V_2 V_3 V_4$
 $V_1 V_2 V_5 V_3 V_4$

❖ 一般情况下，图中的路径不惟一。



路径长度: { 非带权图——路径上边的个数
带权图——路径上各边的权之和

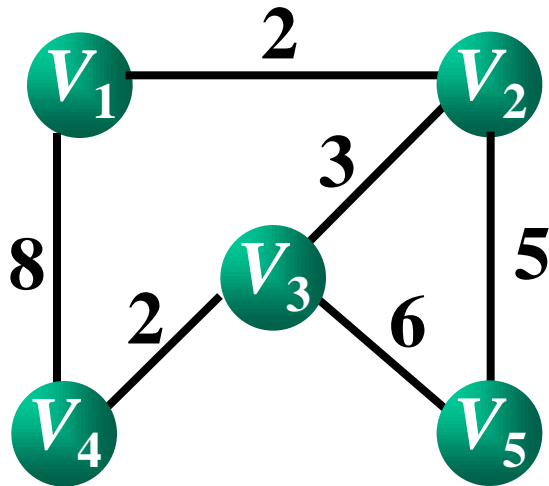


$V_1 V_4$: 长度为1

$V_1 V_2 V_3 V_4$: 长度为3

$V_1 V_2 V_5 V_3 V_4$: 长度为4

路径长度: { 非带权图——路径上边的个数
带权图——路径上各边的权之和



$V_1 V_4$: 长度为8

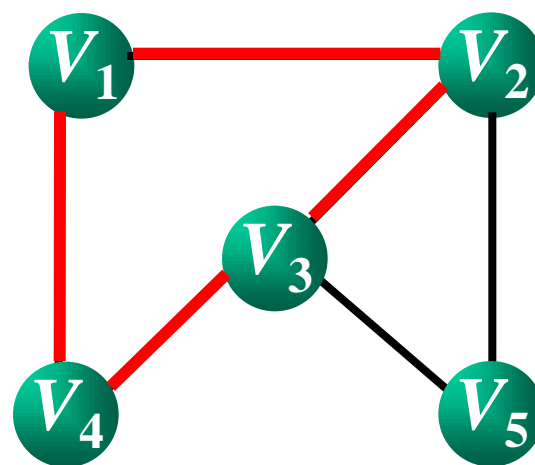
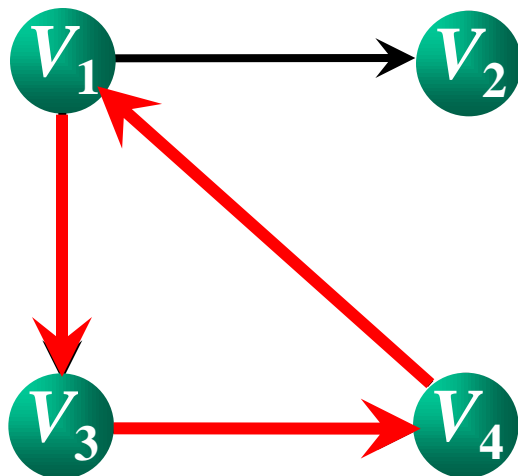
$V_1 V_2 V_3 V_4$: 长度为7

$V_1 V_2 V_5 V_3 V_4$: 长度为15

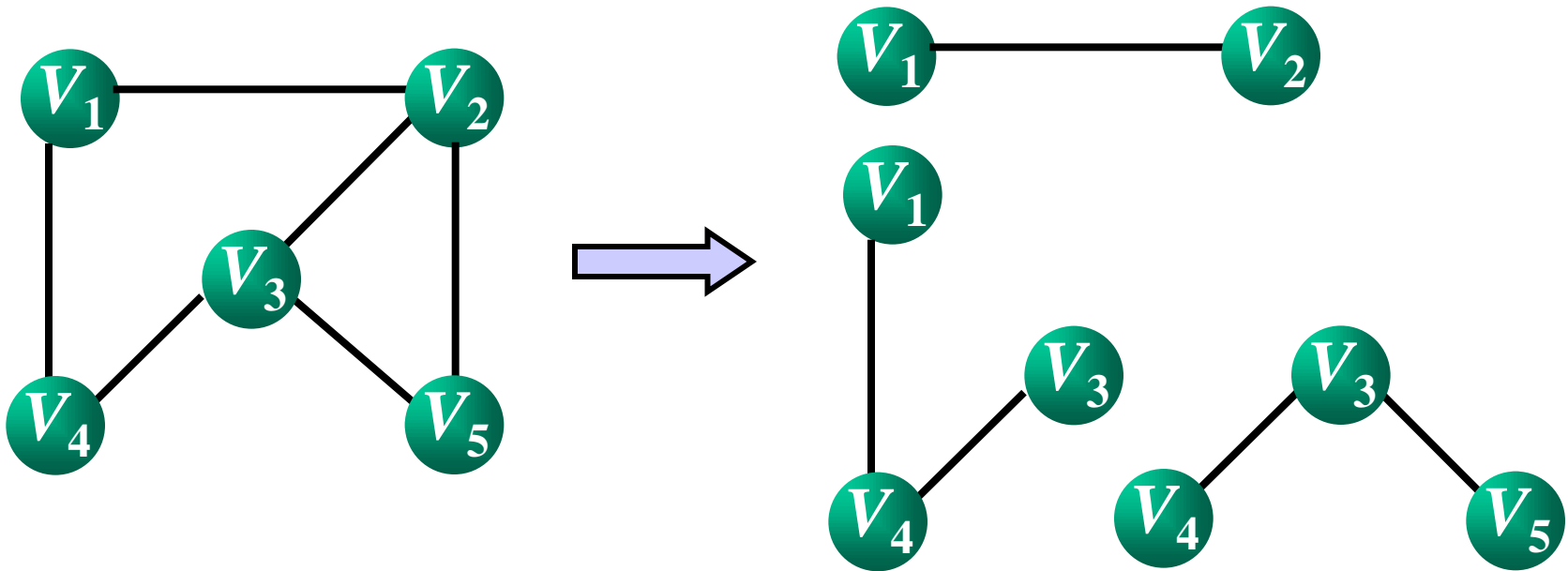
回路（环）： 第一个顶点和最后一个顶点相同的路径。

简单路径： 序列中顶点不重复出现的路径。

简单回路（简单环）(simple circuit)： 除了第一个顶点和最后一个顶点外，其余顶点不重复出现的回路。



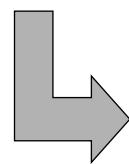
子图(subgraph): 若图 $G = (V, E)$ ， $G' = (V', E')$ ，
如果 $V' \subseteq V$ 且 $E' \subseteq E$ ，则称图 G' 是 G 的子图。



连通图(connected): 在无向图中, 如果从一个顶点 v_i 到另一个顶点 $v_j (i \neq j)$ 有路径, 则称顶点 v_i 和 v_j 是连通的。如果图中任意两个顶点都是连通的, 则称该图是连通图。

连通分量: 非连通图的极大连通子图称为连通分量。

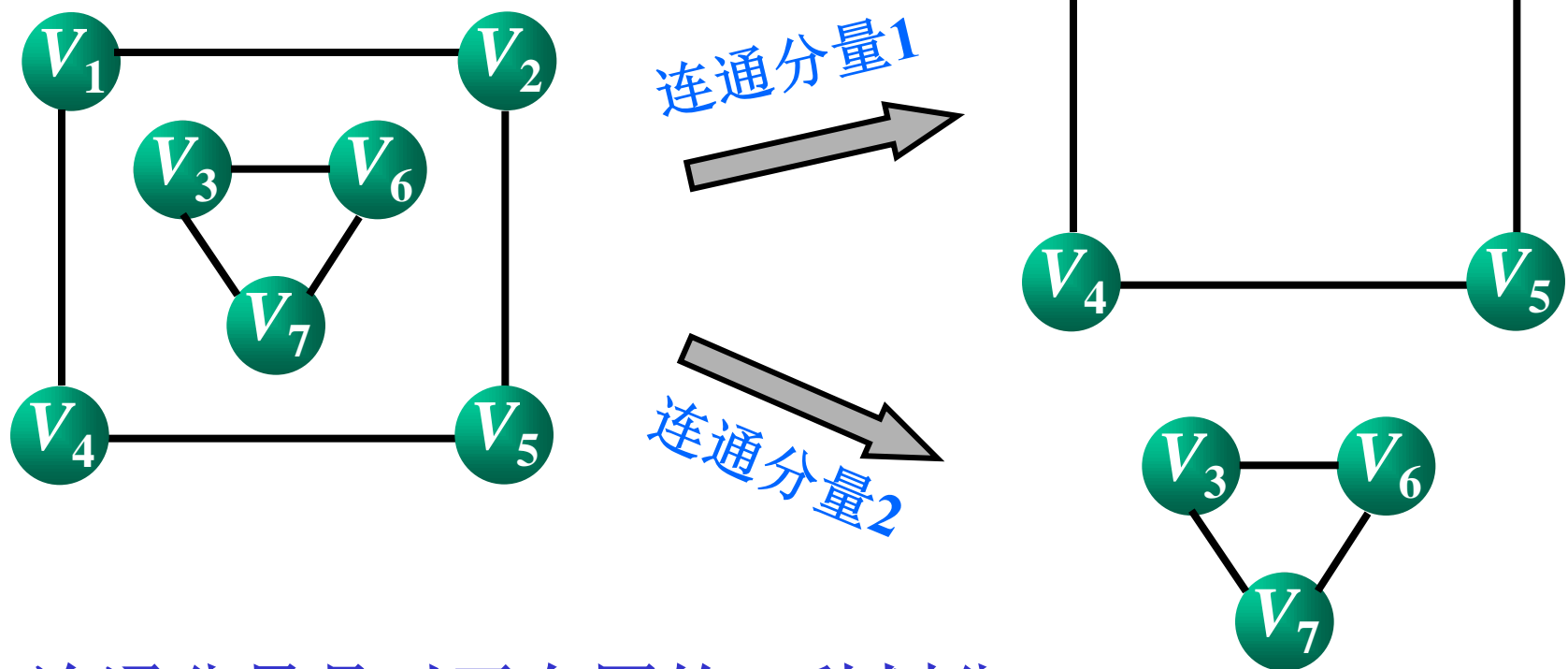
(connected component)



- 1. 含有极大**顶点**数;
- 2. 依附于这些顶点的所有**边**。



如何求得一个非连通图的连通分量?



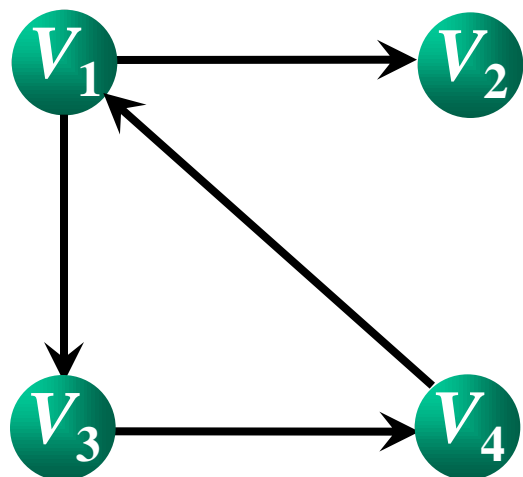
❖ 连通分量是对无向图的一种划分。

强连通图(strongly connected graph): 在有向图中, 对图中任意一对顶点 v_i 和 v_j ($i \neq j$), 若从顶点 v_i 到顶点 v_j 和从顶点 v_j 到顶点 v_i 均有路径, 则称该有向图是强连通图。

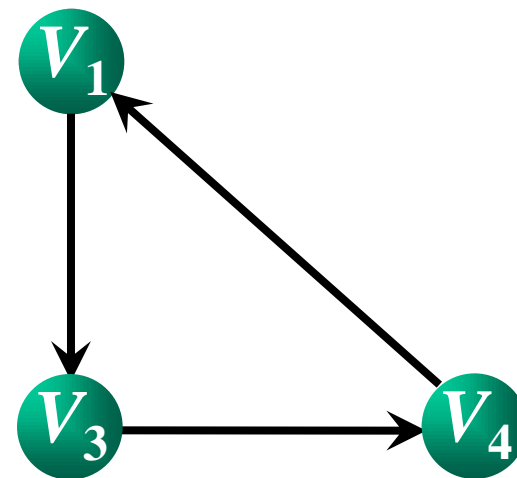
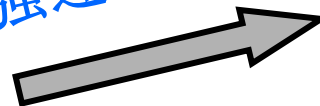
强连通分量: 非强连通图的极大强连通子图。



如何求得一个非强连通图的强连通分量?



强连通分量1

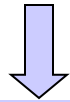


强连通分量2



6.2 图的遍历

图的遍历(graph traverse)是在从图中**某**一顶点出发，对图中所有顶点访问一次且仅**访问**一次。



抽象操作，可以是对结点进行的各种处理，这里简化为输出结点的数据。

图的遍历操作要解决的关键问题

① 在图中，如何选取遍历的起始顶点？

解决方案：从编号小的顶点开始。

- 在**线性表**中，数据元素在表中的编号就是元素在序列中的位置，因而其编号是唯一的；
- 在**树**中，将结点按层序编号，由于树具有层次性，因而其层序编号也是唯一的；
- 在**图**中，任何两个顶点之间都可能存在边，顶点是没有确定的先后次序的，所以，**顶点的编号不唯一**。
为了定义操作的方便，将图中的顶点按任意顺序排列起来，比如，按顶点的存储顺序。

图的遍历操作要解决的关键问题

② 从某个起点始可能到达不了所有其它顶点，怎么办？

解决方案：多次调用从某顶点出发遍历图的算法。

❖ 下面仅讨论从某顶点出发遍历图的算法。

图的遍历操作要解决的关键问题

③ 因图中可能存在回路，某些顶点可能会被重复访问，那么如何避免遍历不会因回路而陷入死循环？

解决方案：附设访问标志数组visited[n] 。

④ 在图中，一个顶点可以和其它多个顶点相连，当这样的顶点访问过后，如何选取下一个要访问的顶点？

解决方案：深度优先遍历和广度优先遍历。

1. 深度优先遍历(depth-first traverse)

基本思想：

- (1) 访问顶点 v ;
- (2) 从 v 的未被访问的邻接点中选取一个顶点 w , 从 w 出发进行深度优先遍历;
- (3) 重复上述两步, 直至图中所有和 v 有路径相通的顶点都被访问到。

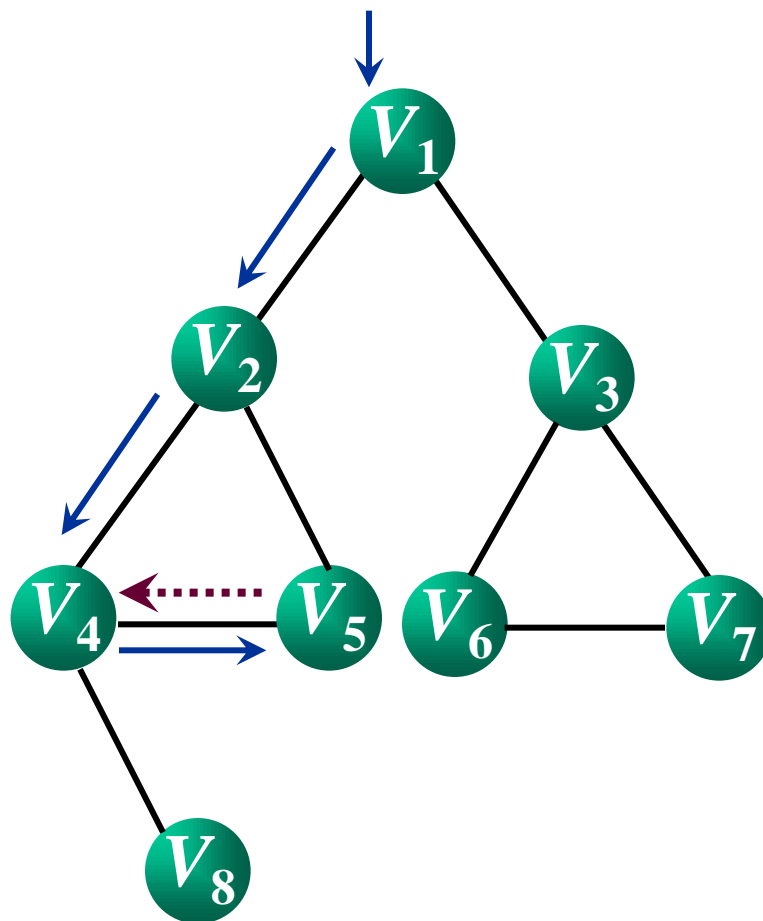


深度优先遍历序列? 入栈序列? 出栈序列?

深一层递归



递归返回



V_5
V_4
V_2
V_1

遍历序列: V_1 V_2 V_4 V_5

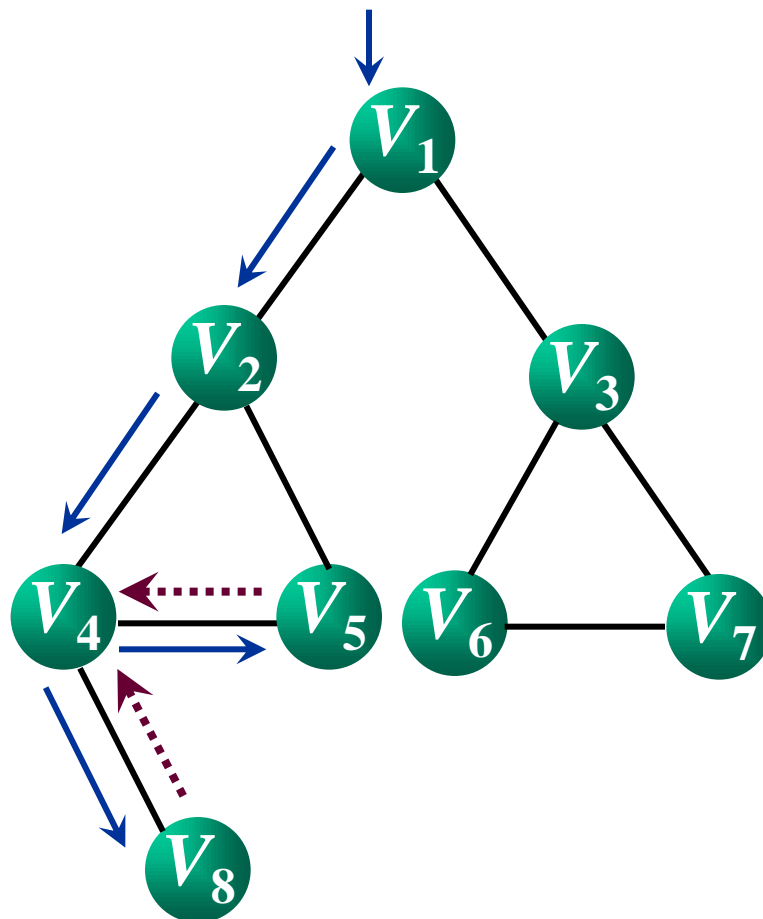


深度优先遍历序列? 入栈序列? 出栈序列?

深一层递归



递归返回



V_8
V_4
V_2
V_1

遍历序列: V_1 V_2 V_4 V_5 V_8

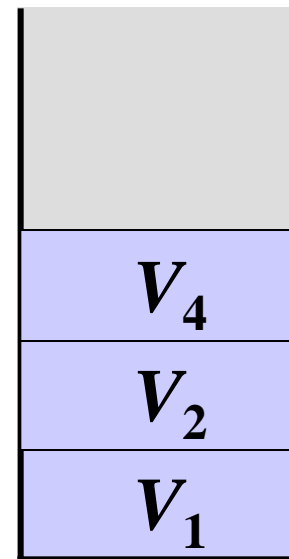
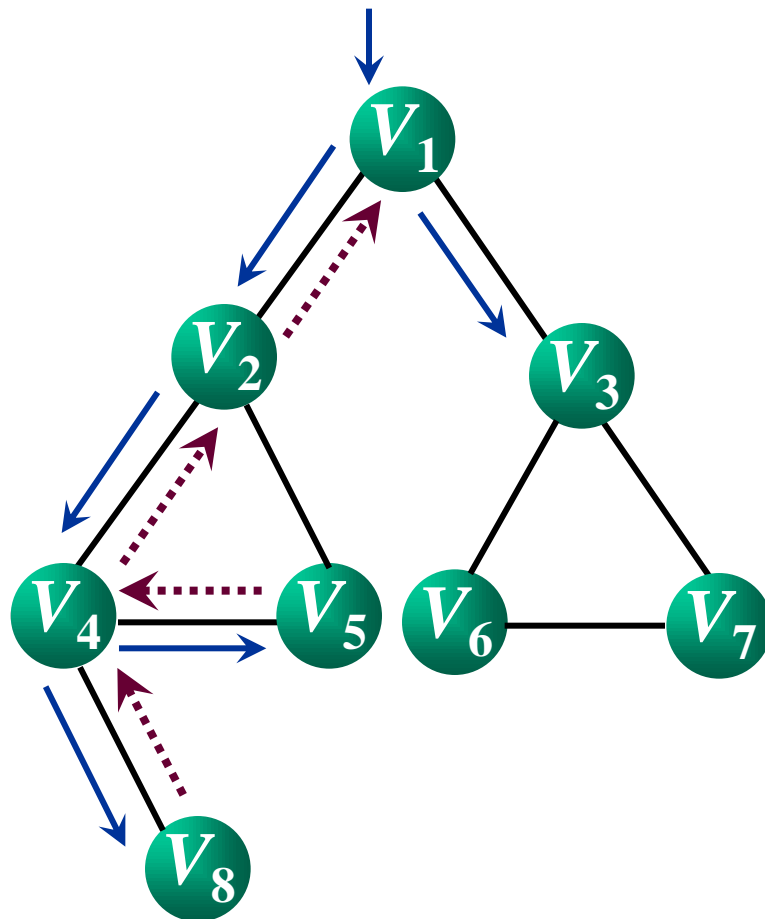


深度优先遍历序列？ 入栈序列？ 出栈序列？

深一层递归



递归返回



遍历序列： V_1 V_2 V_4 V_5 V_8

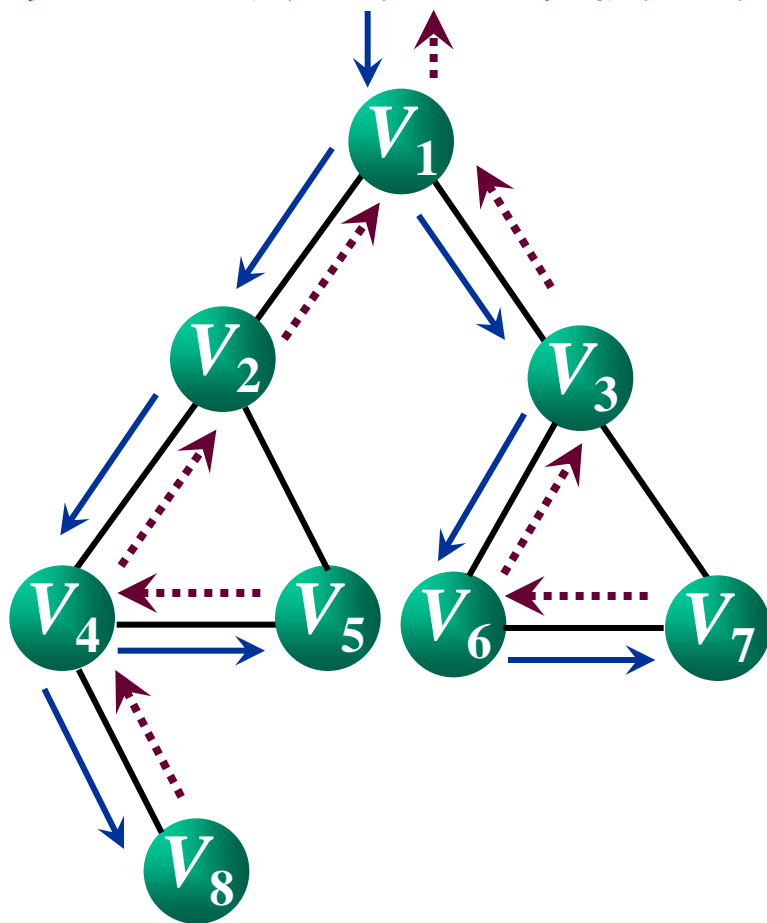


深度优先遍历序列? 入栈序列? 出栈序列?

深一层递归



递归返回



V_7
V_6
V_3
V_1

遍历序列: $V_1 V_2 V_4 V_5 V_8 V_3 V_6 V_7$

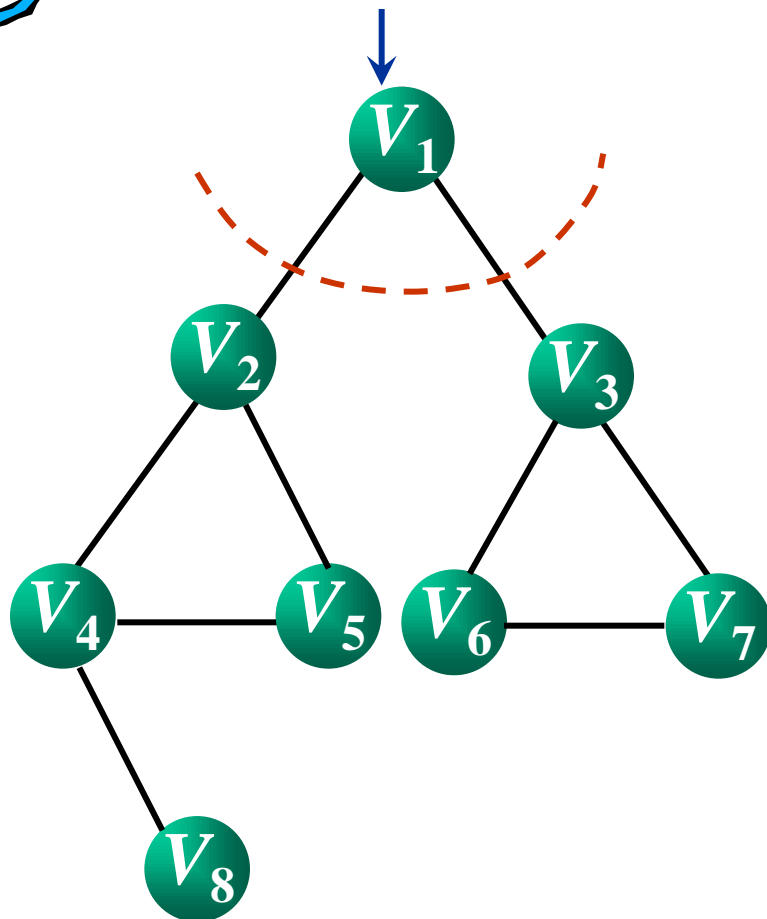
2. 广度优先遍历(breadth-first traverse)

基本思想:

- (1) 访问顶点 v ;
- (2) 依次访问 v 的各个未被访问的邻接点 v_1, v_2, \dots, v_k ;
- (3) 分别从 v_1, v_2, \dots, v_k 出发依次访问它们未被访问的邻接点, 并使“先被访问顶点的邻接点”先于“后被访问顶点的邻接点”被访问。直至图中所有与顶点 v 有路径相通的顶点都被访问到。



广度优先遍历序列? 入队序列? 出队序列?

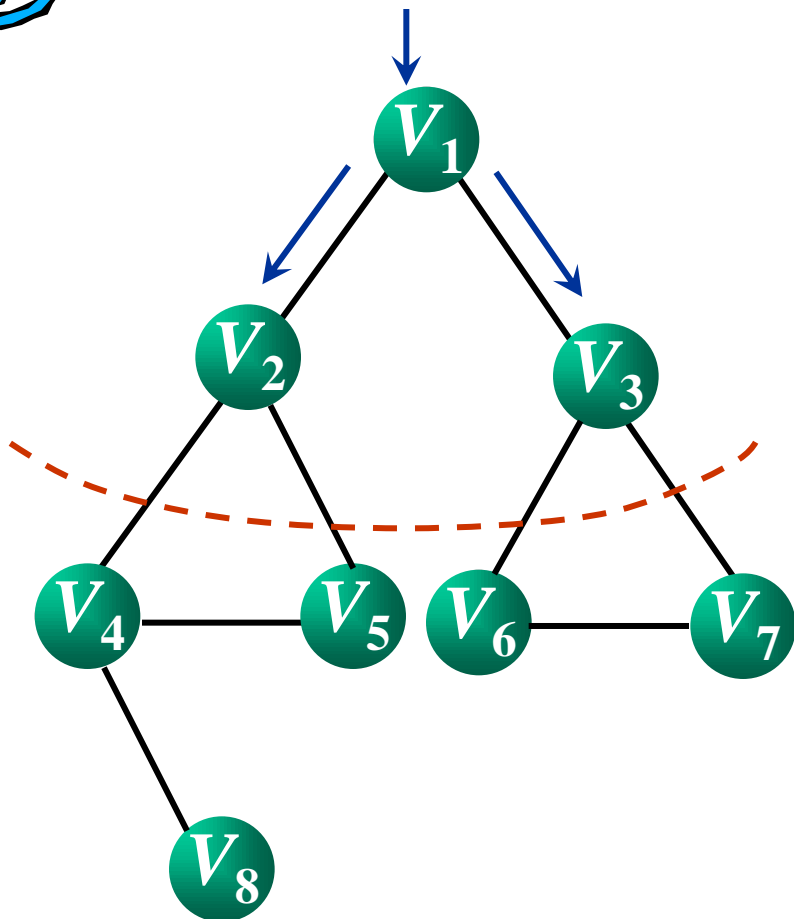


V_1

遍历序列: V_1



广度优先遍历序列? 入队序列? 出队序列?

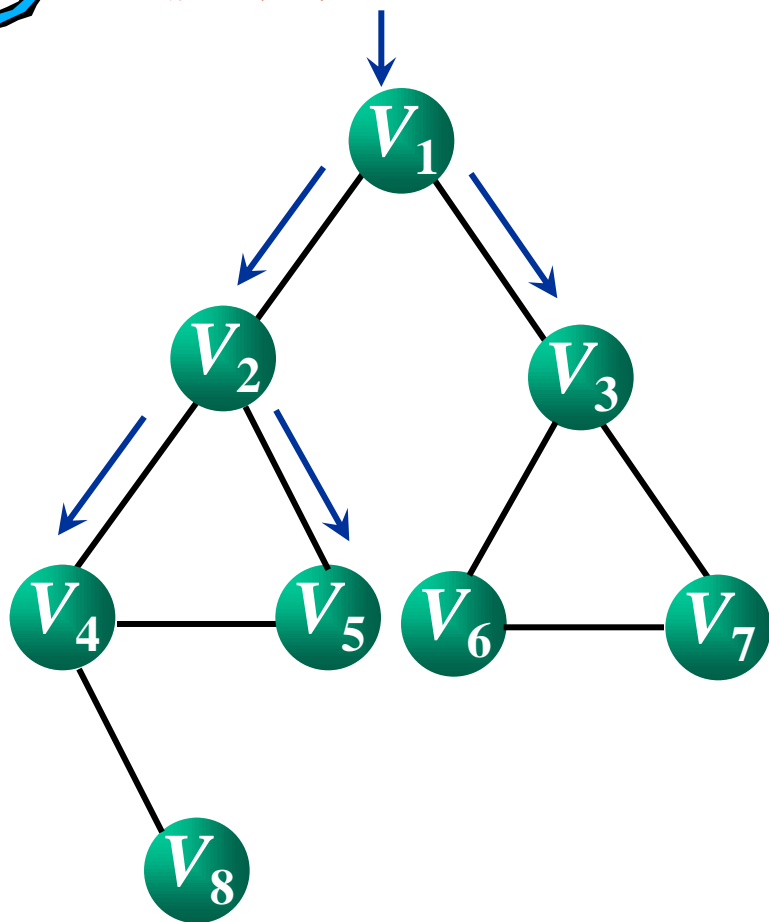


V_2 V_3

遍历序列: V_1 V_2 V_3



广度优先遍历序列? 入队序列? 出队序列?

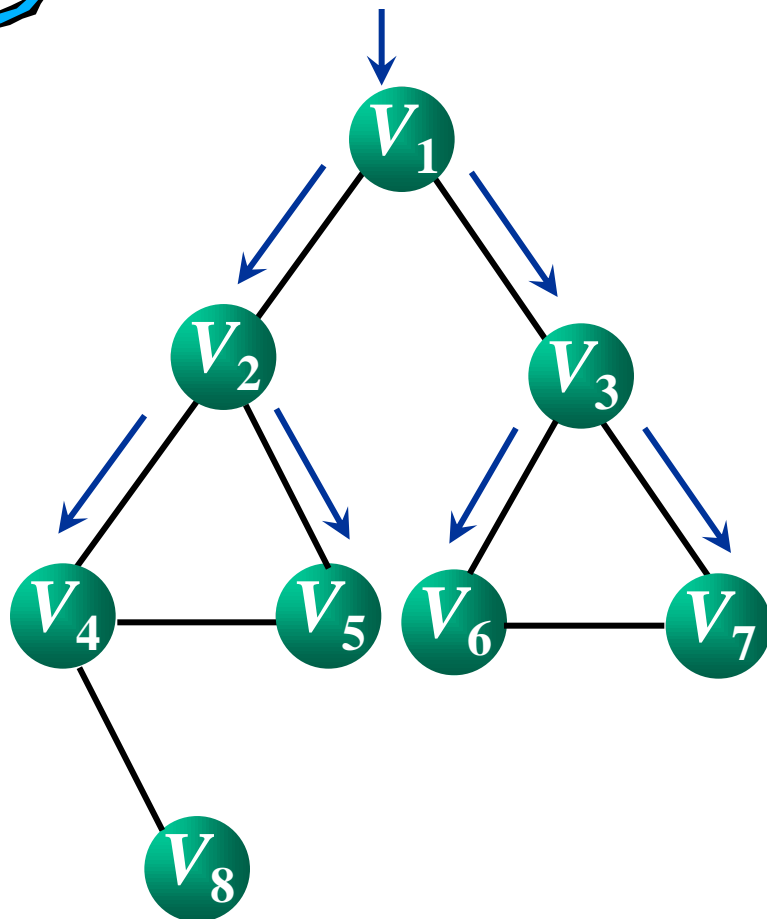


$V_3 V_4 V_5$

遍历序列: $V_1 V_2 V_3 V_4 V_5$



广度优先遍历序列? 入队序列? 出队序列?

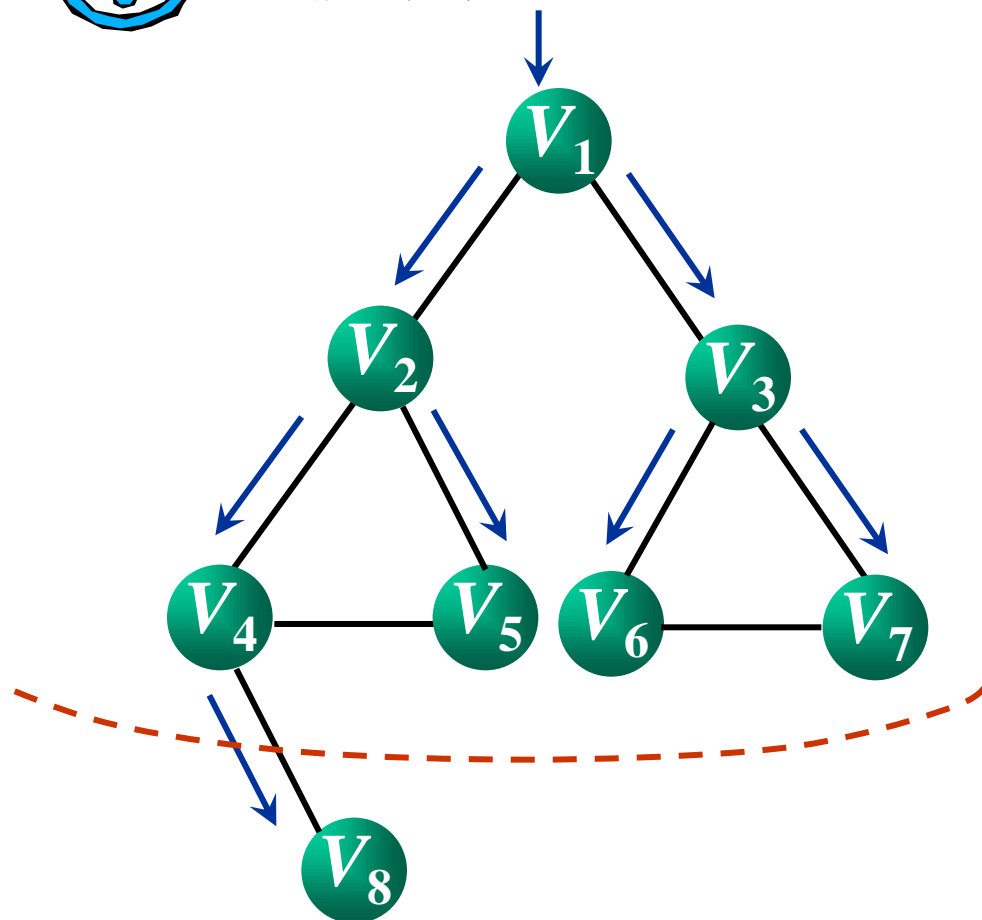


$V_4 V_5 V_6 V_7$

遍历序列: $V_1 V_2 V_3 V_4 V_5 V_6 V_7$



广度优先遍历序列? 入队序列? 出队序列?



$V_5 V_6 V_7 V_8$

遍历序列: $V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8$

6.3 图的存储结构

① 是否可以采用顺序存储结构存储图?

图的特点：顶点之间的关系是 $m:n$ ，即任何两个顶点之间都可能存在关系（边），无法通过存储位置表示这种任意的逻辑关系，~~所以，图无法采用顺序存储结构。~~

② 如何存储图?

考虑图的定义，图是由顶点和边组成的，分别考虑如何存储顶点、如何存储边。

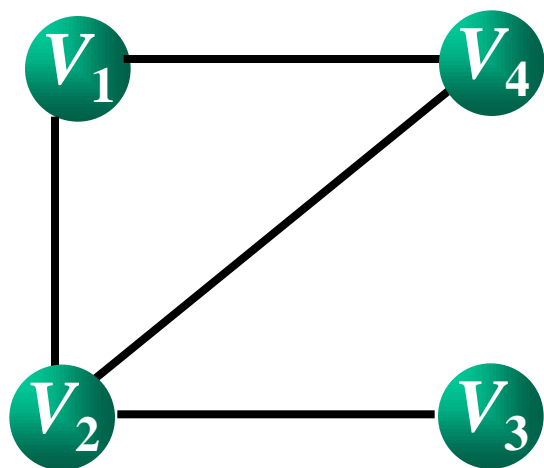
邻接矩阵（数组表示法）(adjacency matrix)

基本思想：用一个一维数组存储图中**顶点**的信息，用一个二维数组（称为邻接矩阵）存储图中各顶点之间的**邻接**关系。

假设图 $G=(V, E)$ 有 n 个顶点，则邻接矩阵是一个 $n \times n$ 的方阵，定义为：

$$\text{arc}[i][j] = \begin{cases} 1 & \text{若}(v_i, v_j) \in E \text{ (或 } \langle v_i, v_j \rangle \in E) \\ 0 & \text{其它} \end{cases}$$

无向图的邻接矩阵



vertex=

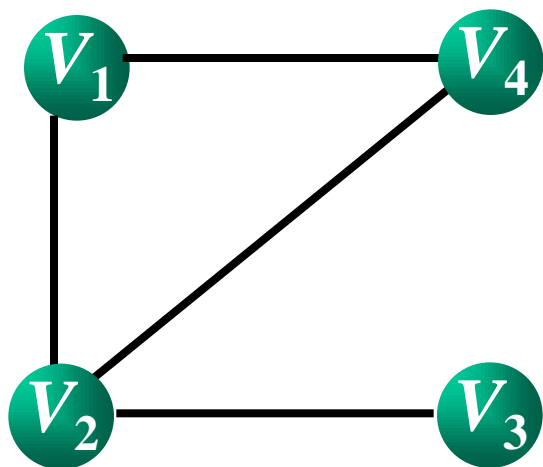
V_1	V_2	V_3	V_4
-------	-------	-------	-------

arc=

	V_1	V_2	V_3	V_4	
[0	1	0	1	V_1
	1	0	1	1	V_2
	0	1	0	0	V_3
	1	1	0	0	V_4

① 无向图的邻接矩阵的特点？

主对角线为 0 且一定是对称矩阵。



vertex=

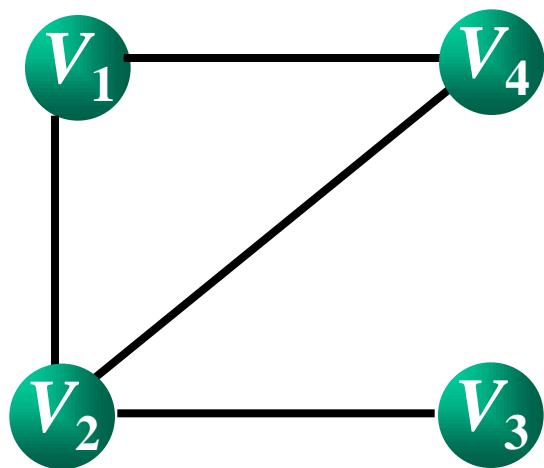
V_1	V_2	V_3	V_4
-------	-------	-------	-------

arc=

	V_1	V_2	V_3	V_4	
arc=	0	1	0	1	V_1
	1	0	1	1	V_2
	0	1	0	0	V_3
	1	1	0	0	V_4

① 如何求顶点 i 的度?

邻接矩阵的第 i 行（或第 i 列）非零元素的个数。



vertex=

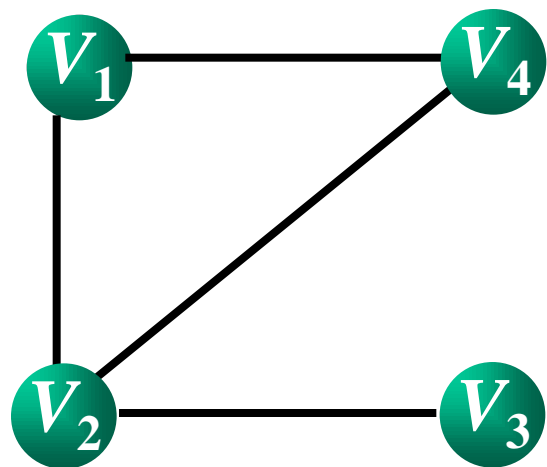
V_1	V_2	V_3	V_4
-------	-------	-------	-------

arc=

	V_1	V_2	V_3	V_4	
[0	1	0	1	V_1
	1	0	1	1	V_2
	0	1	0	0	V_3
	1	1	0	0	V_4

① 如何判断顶点 i 和 j 之间是否存在边?

测试邻接矩阵中相应位置的元素 $\text{arc}[i][j]$ 是否为1。



vertex=

V_1	V_2	V_3	V_4
-------	-------	-------	-------

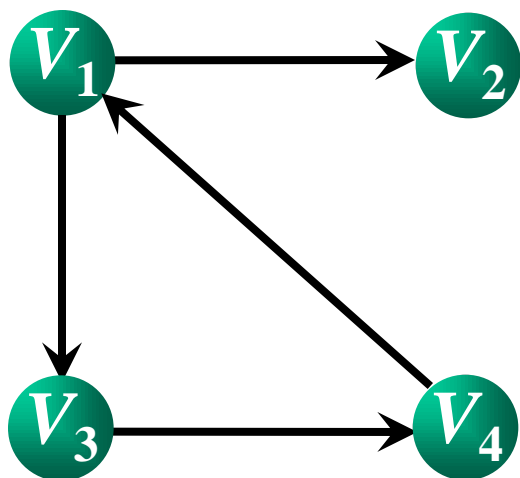
arc=

	V_1	V_2	V_3	V_4	
arc=	0	1	0	1	V_1
	1	0	1	1	V_2
	0	1	0	0	V_3
	1	1	0	0	V_4

① 如何求顶点 i 的所有邻接点？

将数组中第 i 行元素扫描一遍，若 $\text{arc}[i][j]$ 为 1，则顶点 j 为顶点 i 的邻接点。

有向图的邻接矩阵



vertex=

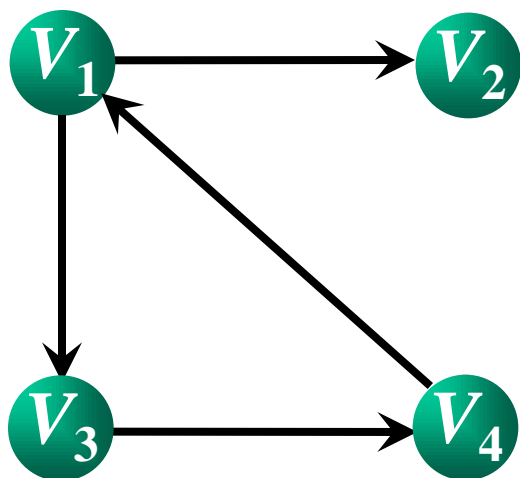
V_1	V_2	V_3	V_4
-------	-------	-------	-------

arc=

	V_1	V_2	V_3	V_4	
$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$					V_1
					V_2
					V_3
					V_4

① 有向图的邻接矩阵一定不对称吗？

不一定，例如有向完全图。



vertex=

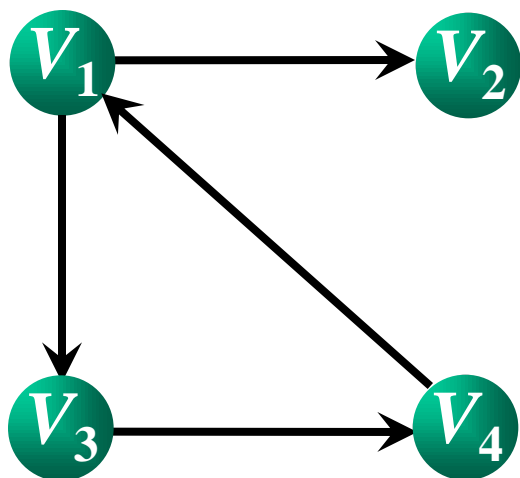
V_1	V_2	V_3	V_4
-------	-------	-------	-------

arc=

	V_1	V_2	V_3	V_4	
V_1	0	1	1	0	V_1
V_2	0	0	0	0	V_2
V_3	0	0	0	1	V_3
V_4	1	0	0	0	V_4

① 如何求顶点 i 的出度?

邻接矩阵的第 i 行元素之和。



vertex=

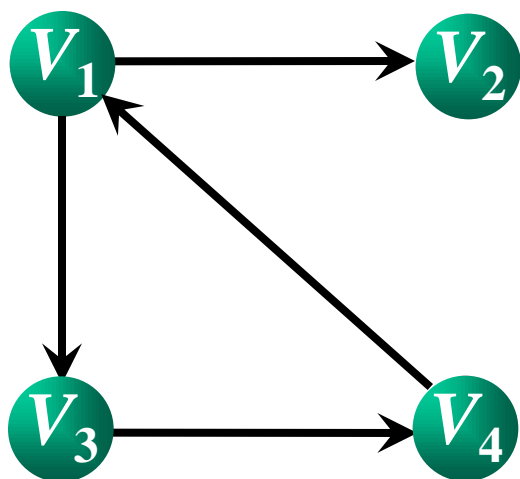
V_1	V_2	V_3	V_4
-------	-------	-------	-------

arc=

	V_1	V_2	V_3	V_4	
V_1	0	1	1	0	V_1
V_2	0	0	0	0	V_2
V_3	0	0	0	1	V_3
V_4	1	0	0	0	V_4

① 如何求顶点 i 的入度?

邻接矩阵的第 i 列元素之和。



vertex=

V_1	V_2	V_3	V_4
-------	-------	-------	-------

arc=

	V_1	V_2	V_3	V_4	
V_1	0	1	1	0	V_1
V_2	0	0	0	0	V_2
V_3	0	0	0	1	V_3
V_4	1	0	0	0	V_4

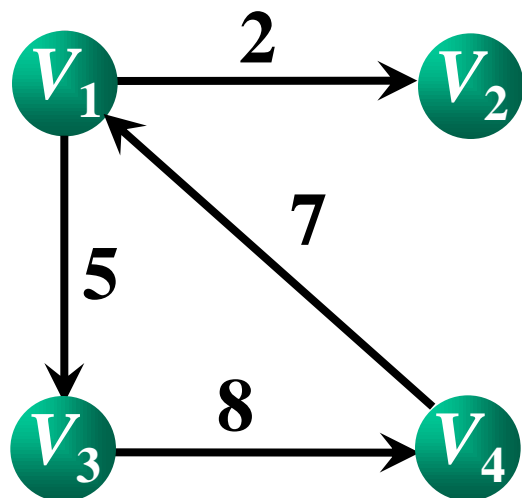
① 如何判断从顶点 i 到顶点 j 是否存在边?

测试邻接矩阵中相应位置的元素 $\text{arc}[i][j]$ 是否为1。

网图的邻接矩阵

网图的邻接矩阵可定义为：

$$\text{arc}[i][j] = \begin{cases} w_{ij} & \text{若}(v_i, v_j) \in E \text{ (或 } \langle v_i, v_j \rangle \in E) \\ 0 & \text{若 } i=j \\ \infty & \text{其他} \end{cases}$$



$$\text{arc} = \begin{bmatrix} 0 & 2 & 5 & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & 0 & 8 \\ 7 & \infty & \infty & 0 \end{bmatrix}$$

邻接矩阵存储无向图的类

```
const int MaxSize=100;
typedef char dataType;
class Mgraph
{
public:
    MGraph (dataType a[ ], int n, int e );
    ~MGraph ( )
    void DFSTraverse (int v);
    void BFSTraverse (int v);
private:
    dataType vertex[MaxSize];
    int arc[MaxSize][MaxSize];
    int vertexNum, arcNum;
};
```

邻接矩阵中图的基本操作——构造函数

1. 确定图的顶点个数和边的个数;
2. 输入顶点信息存储在一维数组`vertex`中;
3. 初始化邻接矩阵;
4. 依次输入每条边存储在邻接矩阵`arc`中;
 - 4.1 输入边依附的两个顶点的序号`i, j`;
 - 4.2 将邻接矩阵的第`i`行第`j`列的元素值置为1;
 - 4.3 将邻接矩阵的第`j`行第`i`列的元素值置为1;

邻接矩阵中图的基本操作——构造函数

```
MGraph::MGraph(dataType a[ ], int n, int e)
{
    vertexNum=n; arcNum=e;
    for (i=0; i<vertexNum; i++)
        vertex[i]=a[i];
    for (i=0; i<vertexNum; i++) //初始化邻接矩阵
        for (j=0; j<vertexNum; j++)
            arc[i][j]=0;
    for (k=0; k<arcNum; k++) //依次输入每一条边
    {
        cin>>i>>j; //边依附的两个顶点的序号
        arc[i][j]=1; arc[j][i]=1; //置有边标志
    }
}
```

邻接矩阵中图的基本操作——深度优先遍历

```
void MGraph::DFSTraverse(int v)
{
    cout<<vertex[v]; visited [v]=1;
    for (j=0; j<vertexNum; j++)
        if (arc[v][j]==1 && visited[j]==0)
            DFSTraverse( j );
}
```

visited在哪声明呢？

public:

int visited[100];

邻接矩阵中图的基本操作——广度优先遍历

```
void MGraph::BFSTraverse(int v)
{
    CirQueue queue;
    cout<<vertex[v]; visited[v]=1; queue.Enqueue(v);
    while (!queue.Empty()) {
        v=queue.DeQueue();
        for (j=0; j<vertexNum; j++)
            if (arc[v][j]==1 && visited[j]==0 ) {
                cout<<vertex[j];visited[j]=1;
                queue.Enqueue(j);
            }
    }
}
```

邻接表(adjacency list)

① 图的邻接矩阵存储结构的空间复杂度?

假设图 G 有 n 个顶点 e 条边，则存储该图需要 $O(n^2)$ 。

② 如果为稀疏图则会出现什么现象?

邻接表存储的基本思想：对于图的每个顶点 v_i ，将所有邻接于 v_i 的顶点链成一个单链表，称为顶点 v_i 的**边表**（对于有向图则称为出边表），所有边表的头指针和存储顶点信息的一维数组构成了**顶点表**。

邻接表有两种结点结构：顶点表结点和边表结点。



顶点表



边 表

vertex: 数据域，存放顶点信息。

firstedge: 指针域，指向边表中第一个结点。

adjvex: 邻接点域，边的终点在顶点表中的下标。

next: 指针域，指向边表中的下一个结点。

定义邻接表的结点

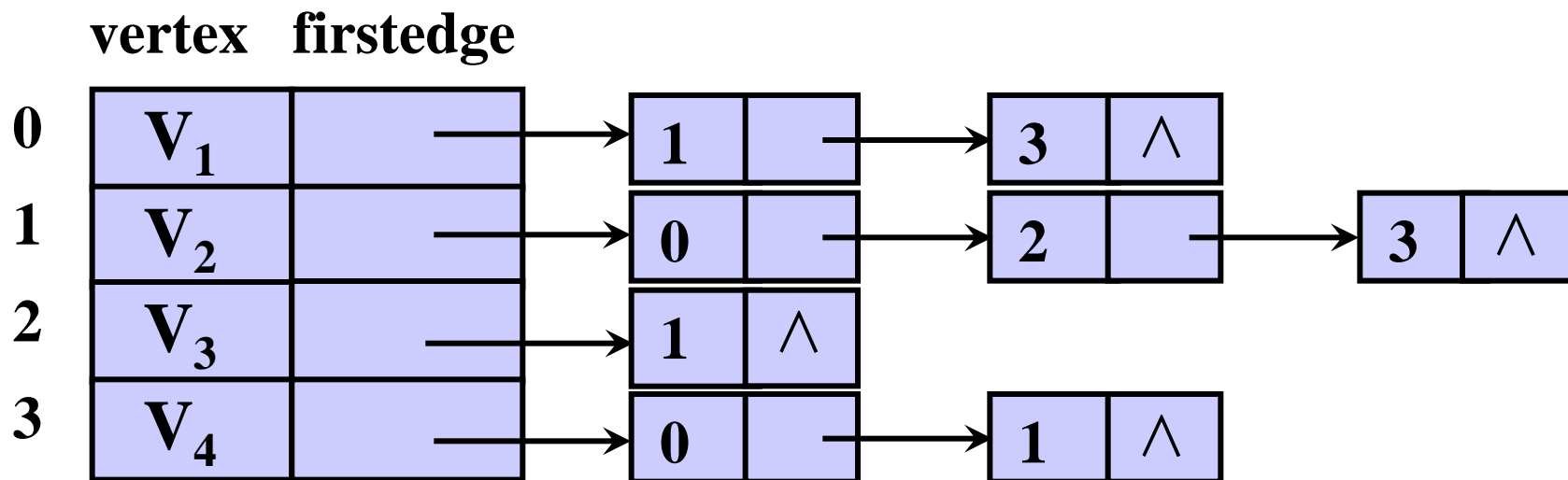
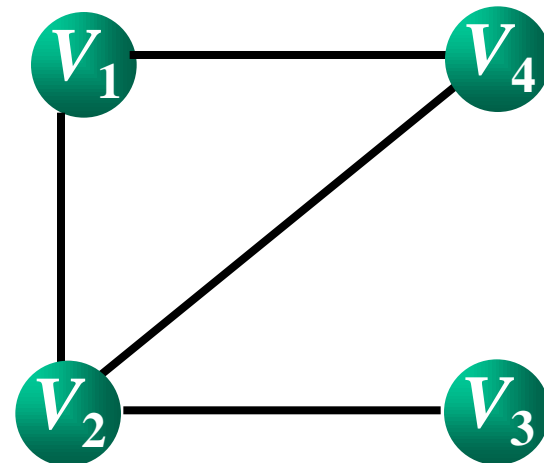
```
struct ArcNode
{
    int adjvex;
    ArcNode *next;
};
typedef char dataType;
struct VertexNode
{
    dataType vertex;
    ArcNode *firstedge;
};
```



无向图的邻接表

② 边表中的结点表示什么？

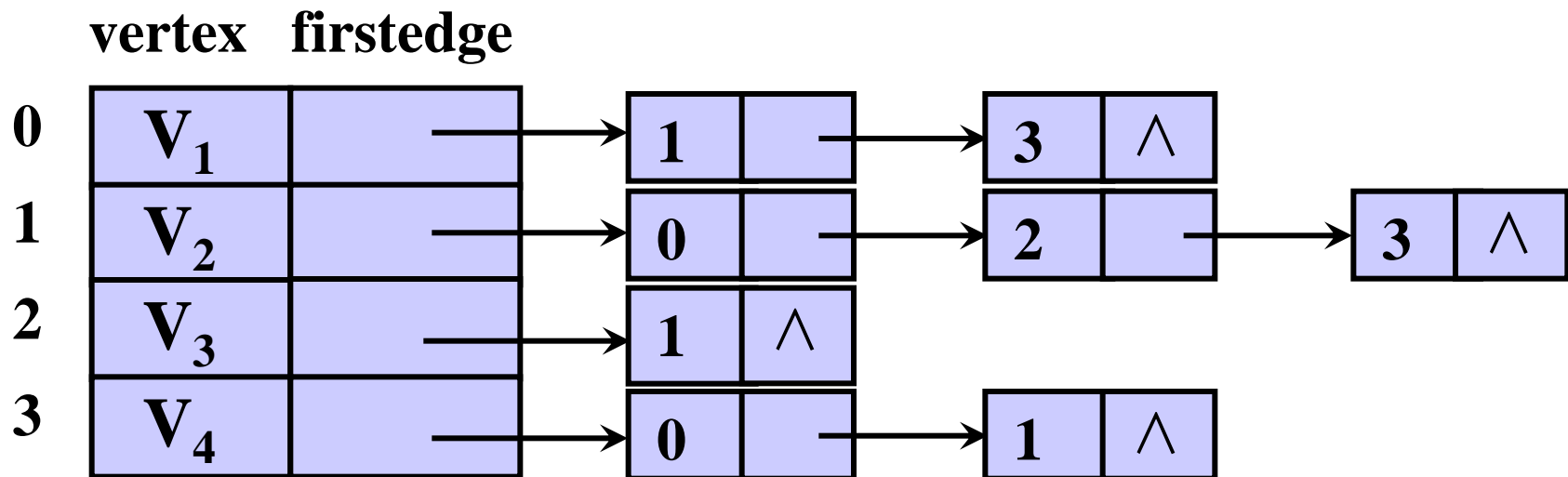
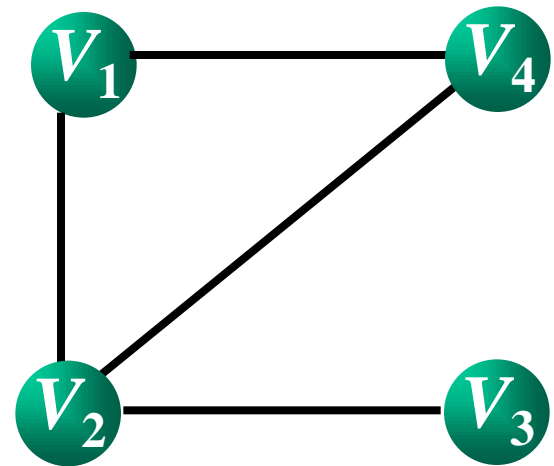
每个结点对应图中的一条边，
邻接表的空间复杂度为 $O(n+e)$ 。





如何求顶点 i 的度?

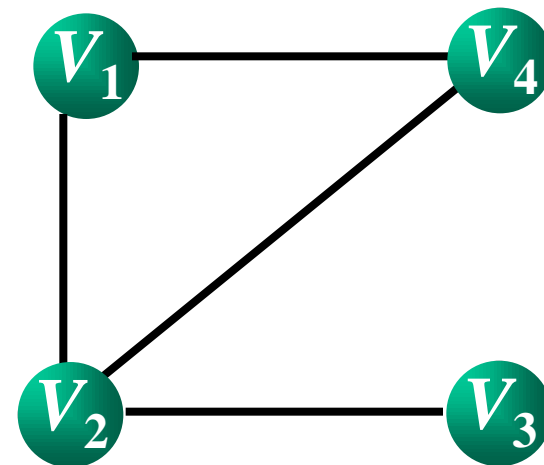
顶点 i 的边表中结点的个数。



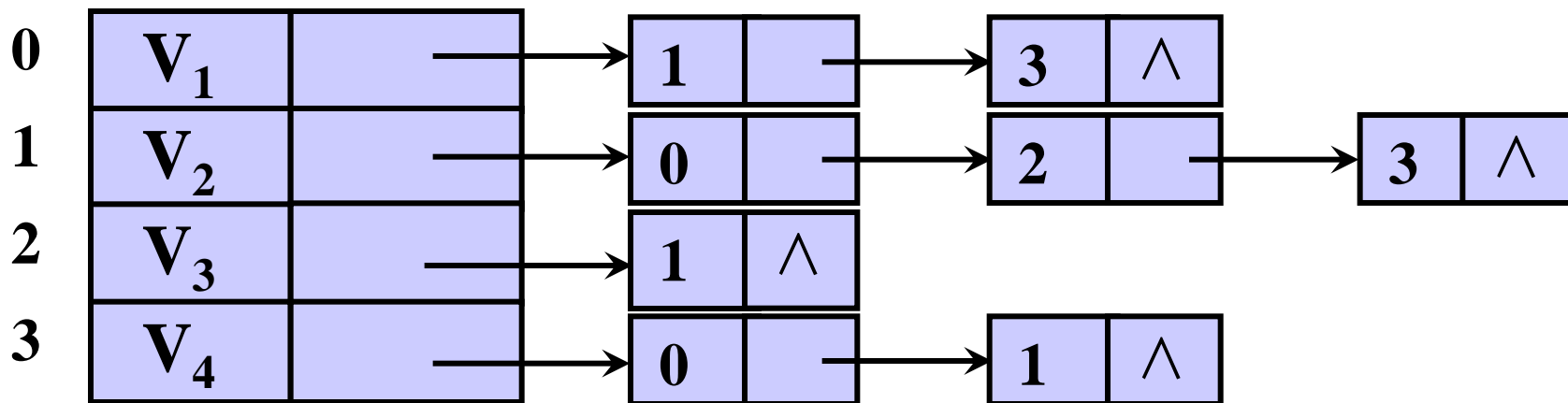


如何判断顶点 i 和顶点 j 之间是否存在边?

测试顶点 i 的边表中是否存在终点为 j 的结点。



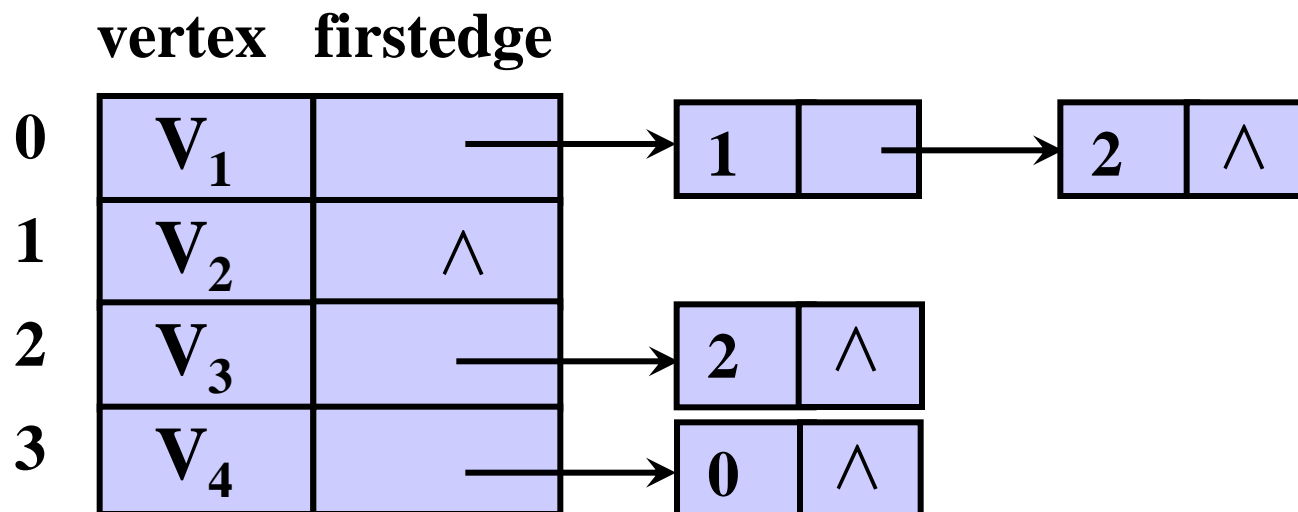
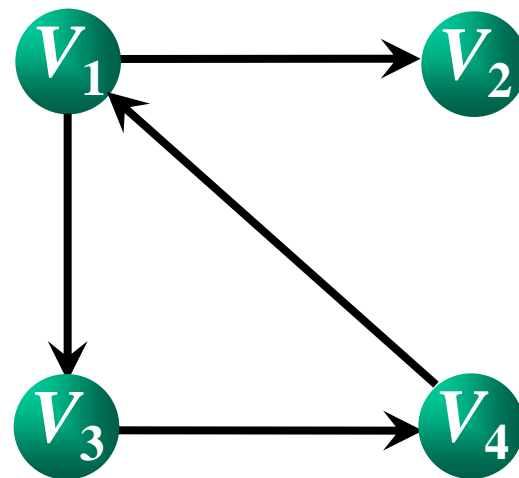
vertex firstedge



有向图的邻接表

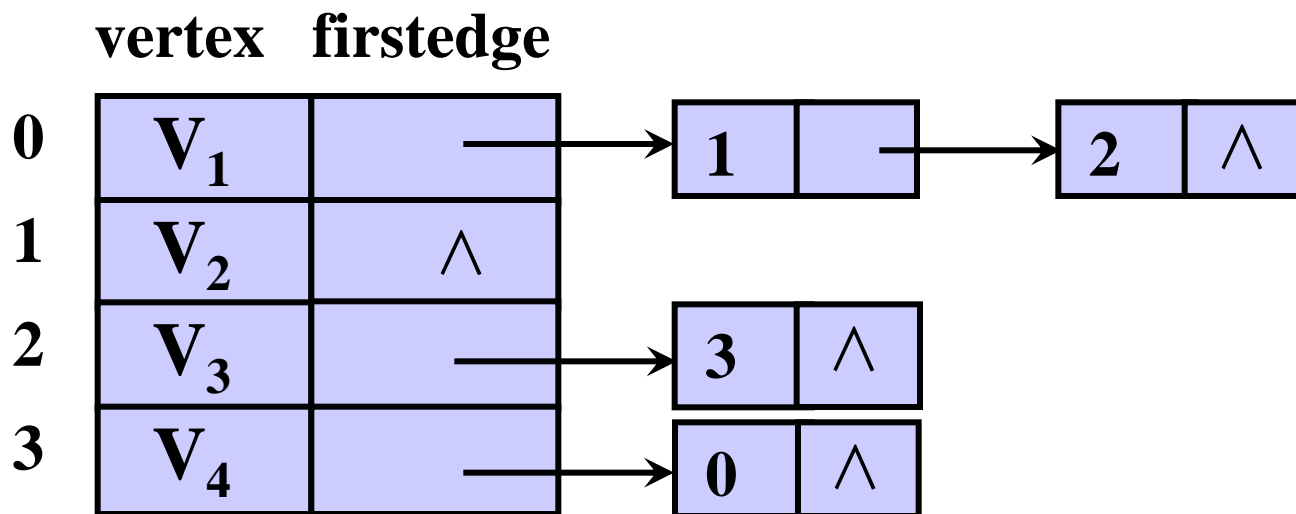
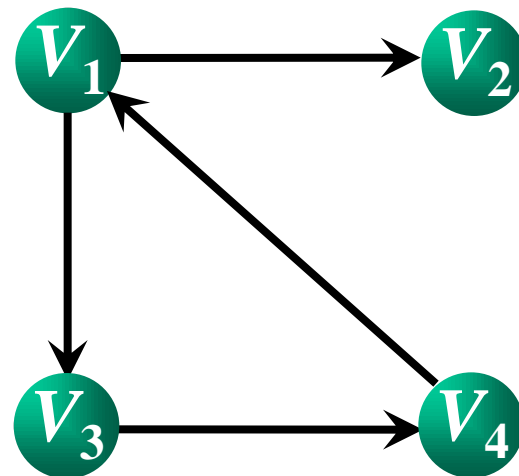
① 如何求顶点 i 的出度?

顶点 i 的出边表中结点的个数。



① 如何求顶点 i 的入度?

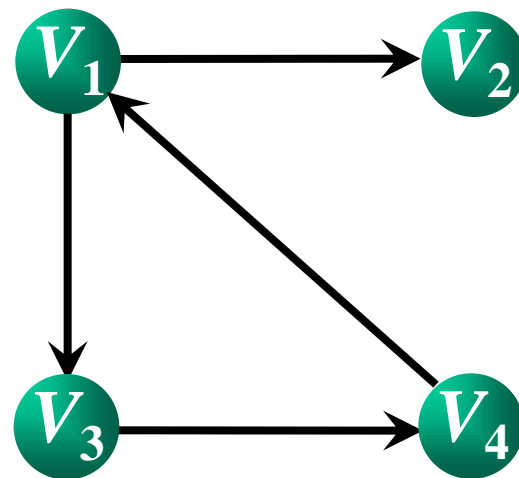
各顶点的出边表中以顶点 i 为终点的结点个数。



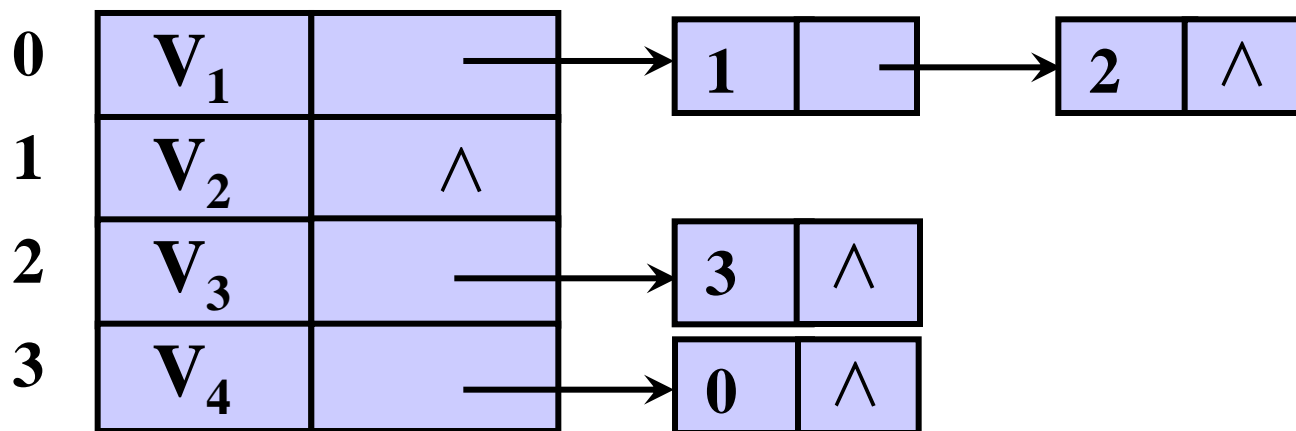


如何求顶点 i 的所有邻接点？

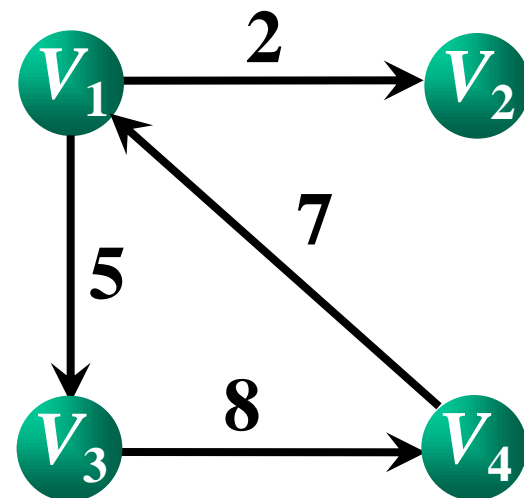
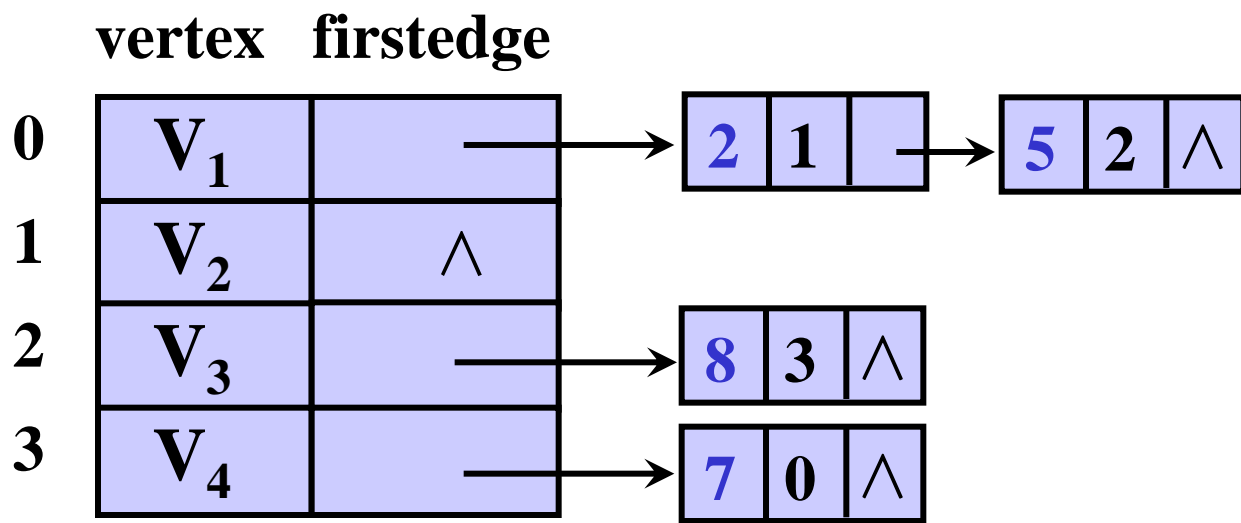
遍历顶点 i 的边表，该边表中的所有终点都是顶点 i 的邻接点。



vertex firstedge



网图的邻接表



邻接表存储有向图的类

```
const int MaxSize=10; //图的最大顶点数
typedef char dataType;
class ALGraph
{
public:
    ALGraph (dataType a[ ], int n, int e);
    ~ALGraph;
    void DFSTraverse (int v);
    void BFSTraverse (int v);
private:
    VertexNode adjlist[MaxSize];
    int vertexNum, arcNum;
};
```

邻接表中图的基本操作——构造函数

1. 确定图的顶点个数和边的个数;
2. 输入顶点信息, 初始化该顶点的边表;
3. 依次输入边的信息并存储在边表中;
 - 3.1 输入边所依附的两个顶点的序号 i 和 j ;
 - 3.2 生成邻接点序号为 j 的边表结点 s ;
 - 3.3 将结点 s 插入到第 i 个边表的头部;

邻接表中图的基本操作——构造函数

```
ALGraph::ALGraph(dataType a[ ], int n, int e)  
{  
    vertexNum=n; arcNum=e;  
    for (i=0; i<vertexNum; i++)  
        //输入顶点信息，初始化边表  
        {  
            adjlist[i].vertex=a[i];  
            adjlist[i].firstedge=NULL;  
        }  
}
```

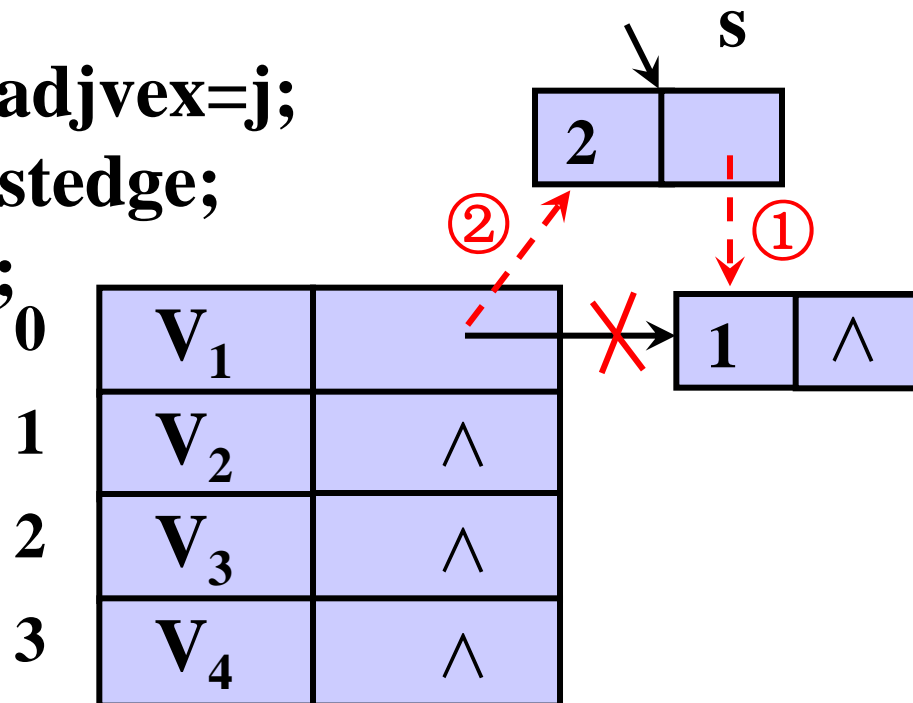
0	V_1	\wedge
1	V_2	\wedge
2	V_3	\wedge
3	V_4	\wedge

邻接表中图的基本操作——构造函数

```

for (k=0; k<arcNum; k++)
//输入边的信息存储在边表中
{
    cin>>i>>j;
    s=new ArcNode; s->adjvex=j;
    s->next=adjlist[i].firstedge;
    adjlist[i].firstedge=s;
}
}

```



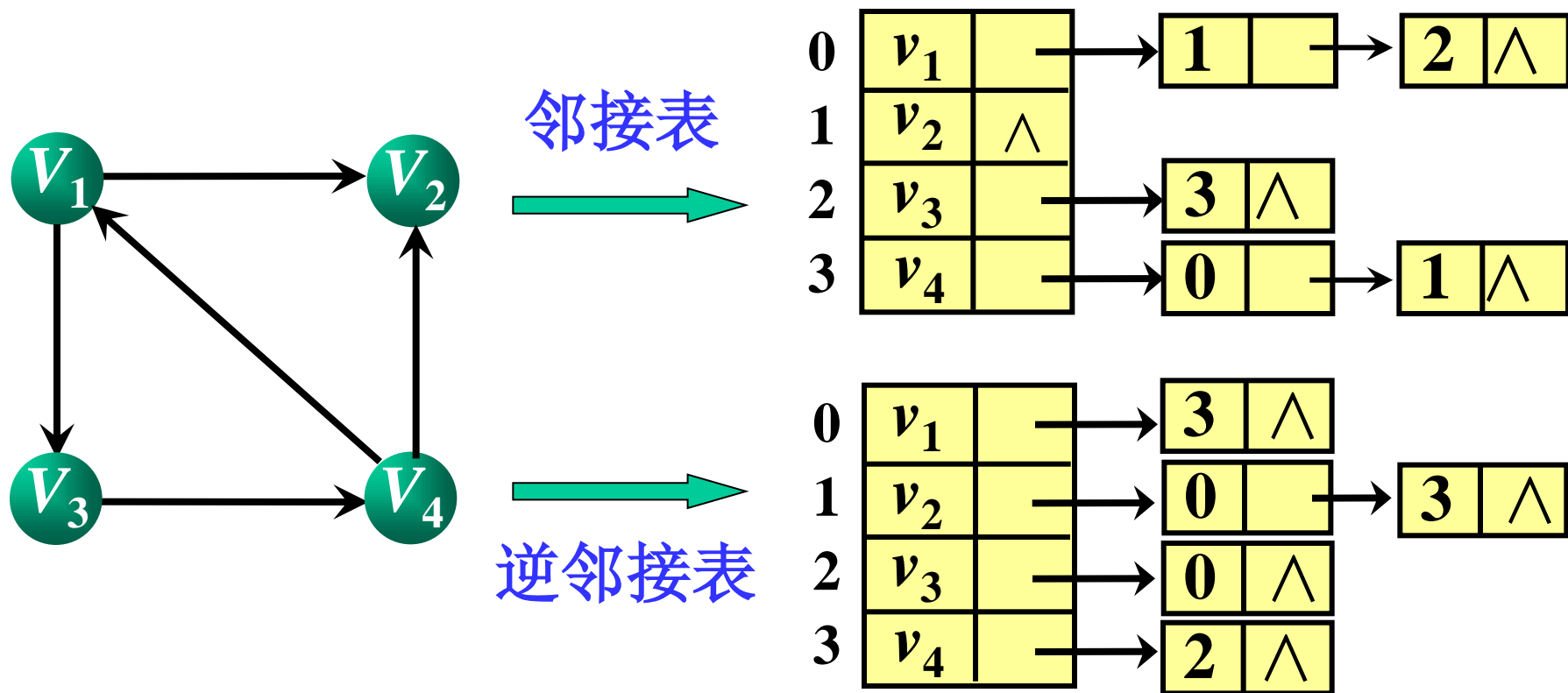
邻接表中图的基本操作——深度优先遍历

```
void ALGraph::DFSTraverse(int v)
{
    cout<<adjlist[v].vertex; visited[v]=1;
    p=adjlist[v].firstedge;
    while (p!=NULL)
    {
        j=p->adjvex;
        if (visited[j]==0) DFSTraverse(j);
        p=p->next;
    }
}
```

邻接表中图的基本操作——广度优先遍历

```
void ALGraph::BFSTraverse(int v)
{
    CirQueue queue;
    cout<<adjlist[v].vertex; visited[v]=1; queue.Enqueue(v);
    while (!queue.Empty()) {
        v=queue.DeQueue(); p=adjlist[v].firstedge;
        while (p!=NULL) {
            j= p->adjvex;
            if (visited[j]==0) {
                cout<<adjlist[j].vertex; visited[j]=1;
                queue.Enqueue(j);
            }
            p=p->next;
        }
    }
}
```

十字链表(orthogonal list) (也叫正交表)



① 将邻接表与逆邻接表合二为一?

② 为什么要合并?

十字链表的结点结构

vertex	firstin	firstout	tailvex	headvex	headlink	taillink
--------	---------	----------	---------	---------	----------	----------

顶点表结点

边表结点

vertex: 数据域，存放顶点信息；

firstin: 入边表头指针；

firstout: 出边表头指针；

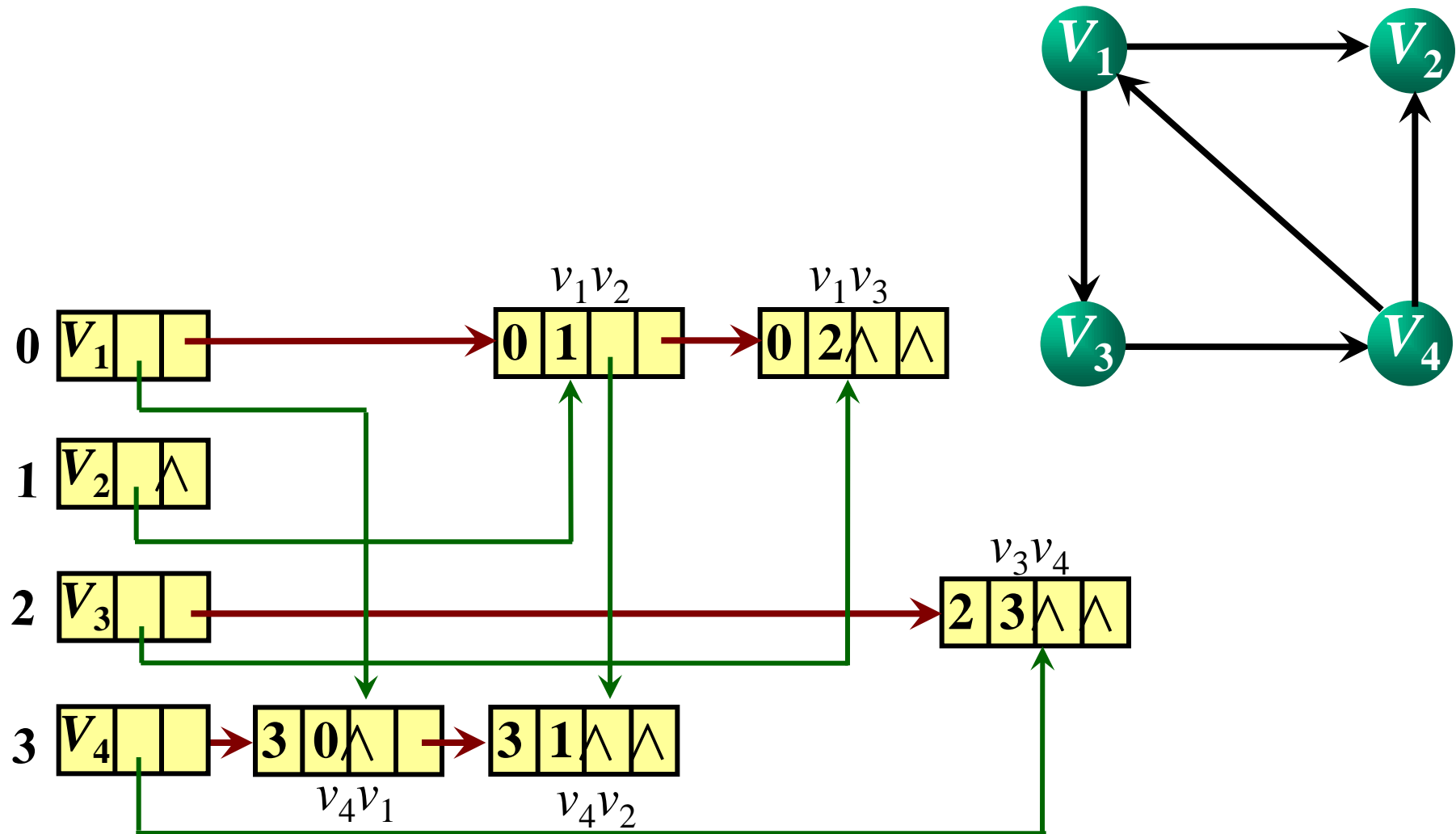
tailvex: 有向边（或弧）的起点在顶点表中的下标；

headvex: 有向边（或弧）的终点在顶点表中的下标；

headlink: 入边表指针域；

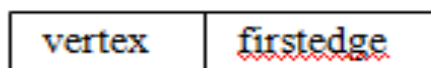
taillink: 出边表指针域。

十字链表举例

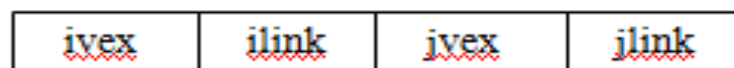


邻接多重表 (adjacency multilist)

- 对于无向图，当用邻接表表示时，为了解决重复存储问题，可以采用“邻接多重表”。

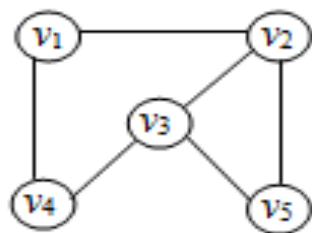


(a) 顶点表结点

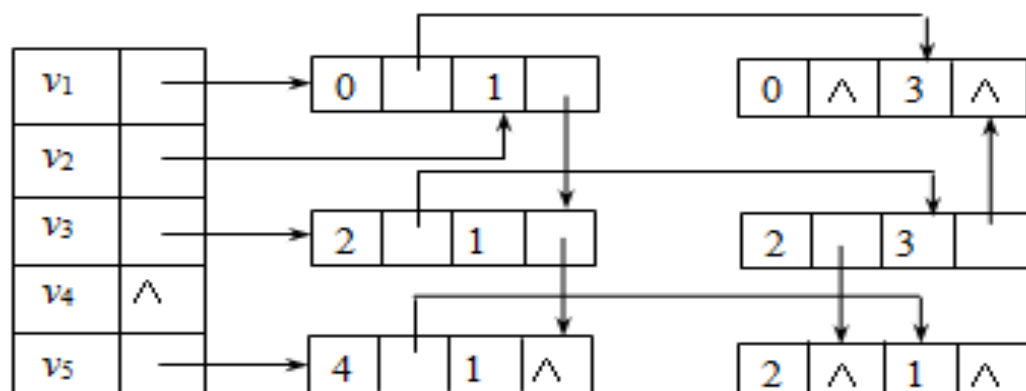


(b) 边表结点

邻接多重表的结点结构



(a) 无向图



(b) 邻接多重表存储示意图

无向图的邻接多重表存储示意图

图的存储结构的比较：邻接矩阵和邻接表

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

6.4 图的连通性

1-无向图的连通性

① 如何判断一个无向图是否是连通图，或有几个连通分量？

通过对无向图遍历即可得到结果。

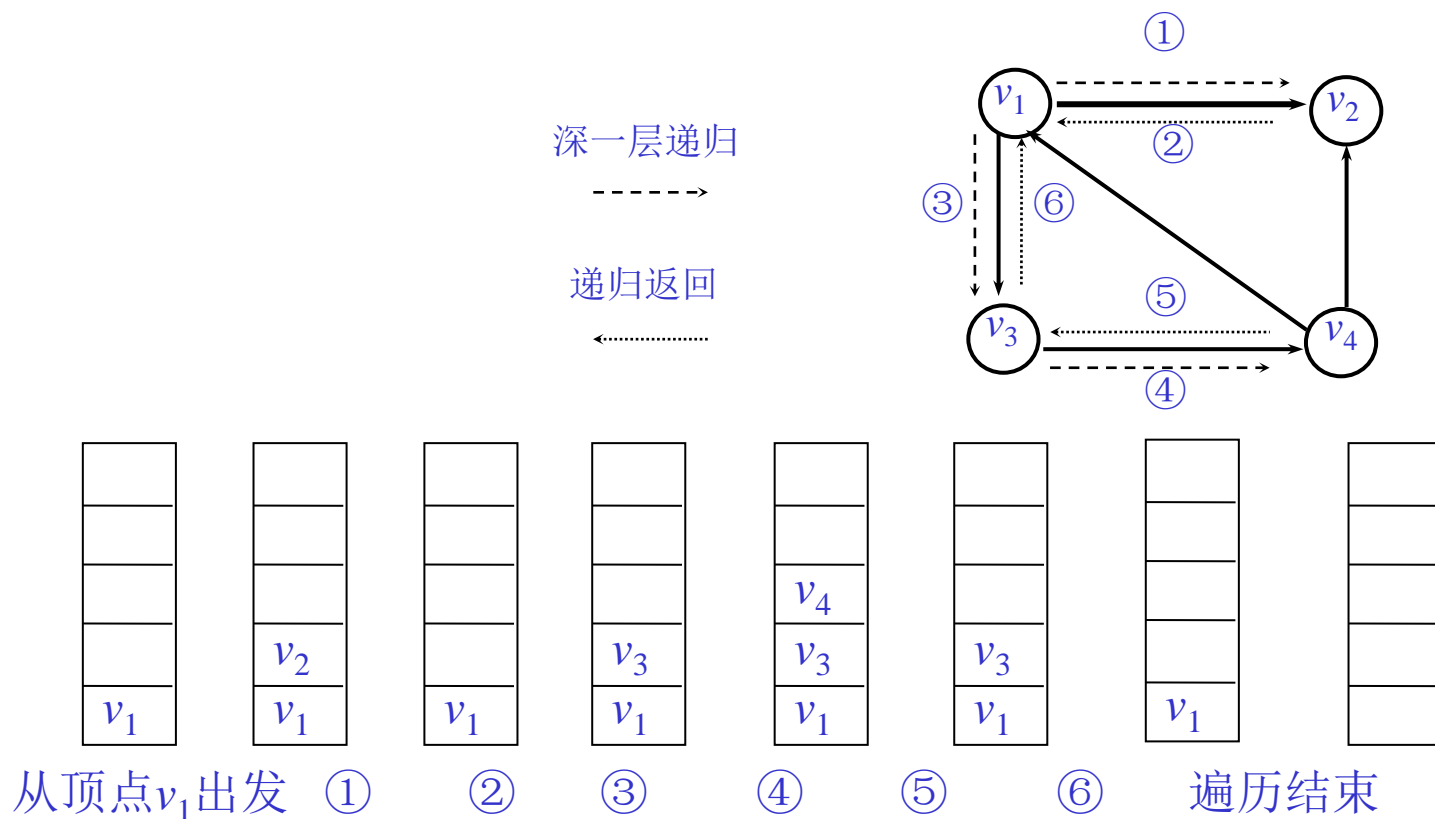
➤ **连通图**：仅需从图中任一顶点出发，进行深度优先搜索（或广度优先搜索），便可访问到图中所有顶点。

➤ **非连通图**：需从多个顶点出发进行搜索，而每一次从一个新的起始点出发进行搜索过程中得到的顶点访问序列恰为其各个连通分量中的顶点集。

判定一个无向图是否连通，或有几个连通分量

1. **count=0;**
2. **for** (图中每个顶点v)
 - 2.1 **if** (v尚未被访问过)
 - 2.1.1 **count++;**
 - 2.1.2 从v出发遍历该图;
3. **if** (count==1) **cout**<<"图是连通的";
else cout<<"图中有"<<count<<"个连通分量";

2-有向图的连通性



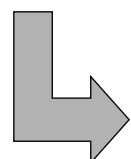
求强连通分量：出栈顶点的序列是 $v_2v_4v_3v_1$ ，再从顶点 v_1 出发做逆向深度优先遍历，得到两个顶点集 $\{v_1v_4v_3\}$ 和 $\{v_2\}$ 。

2-有向图的连通性（续）

- (1) 从某顶点出发进行深度优先遍历，并按其所有邻接点都访问（即出栈）的顺序将顶点排列起来。
- (2) 从最后完成访问的顶点出发，沿着以该顶点为头的弧作逆向的深度优先遍历。若不能访问到所有顶点，则从余下的顶点中最后访问的那个顶点出发，继续作逆向的深度优先遍历，直至有向图中所有顶点都被访问到为止。
- (3) 每一次逆向深度优先遍历所访问到的顶点集便是该有向图的一个强连通分量的顶点集。若仅做一次逆向深度优先遍历就能访问到图的所有顶点，则该有向图是强连通图。

6.5 图的应用举例: (1) 最小生成树

生成树(spanning tree): n 个顶点的连通图 G 的生成树是包含 G 中全部顶点的一个极小连通子图。

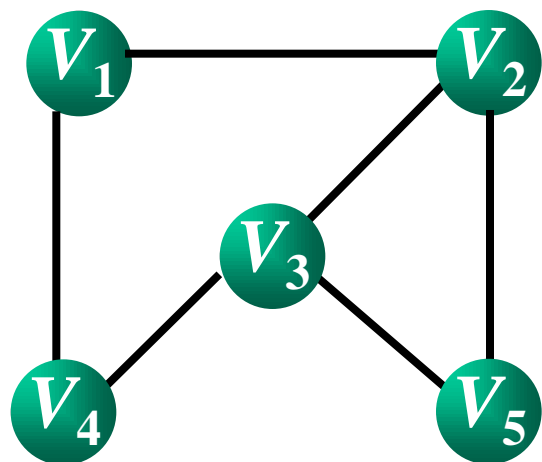
 含有 $n-1$ 条边 $\left\{ \begin{array}{l} \text{多: 构成回路} \\ \text{少: 不连通} \end{array} \right.$



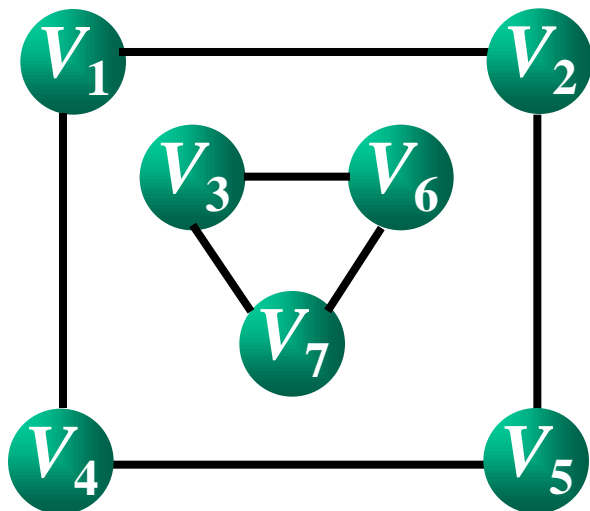
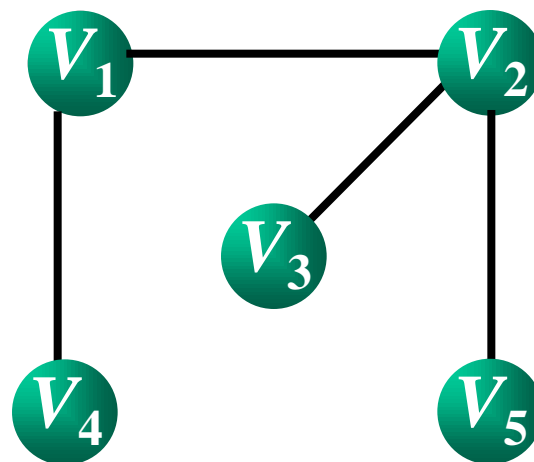
如何理解极小连通子图?

生成森林(spanning forest): 在非连通图中, 由每个连通分量都可以得到一棵生成树, 这些连通分量的生成树就组成了一个非连通图的生成森林。

生成树和生成森林



生成树



生成森林

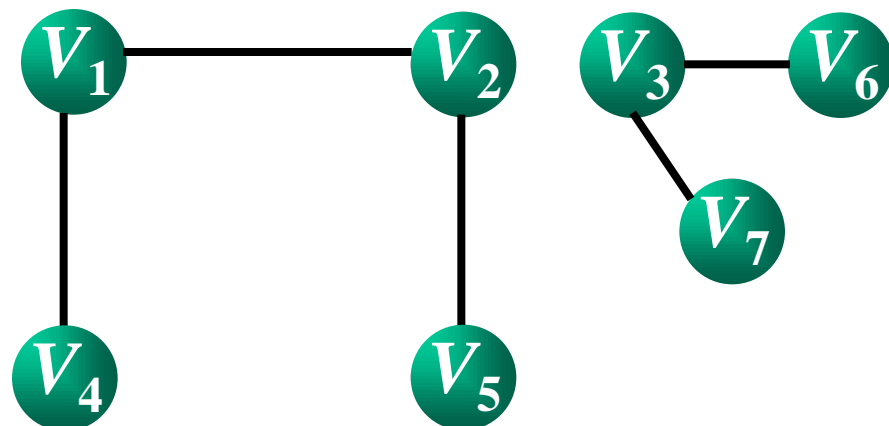
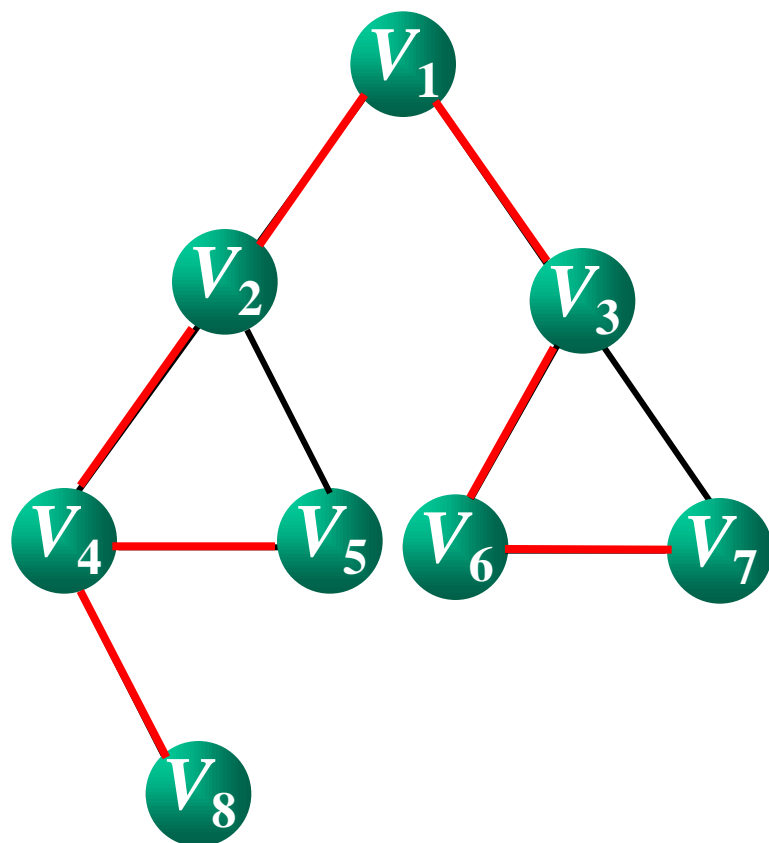
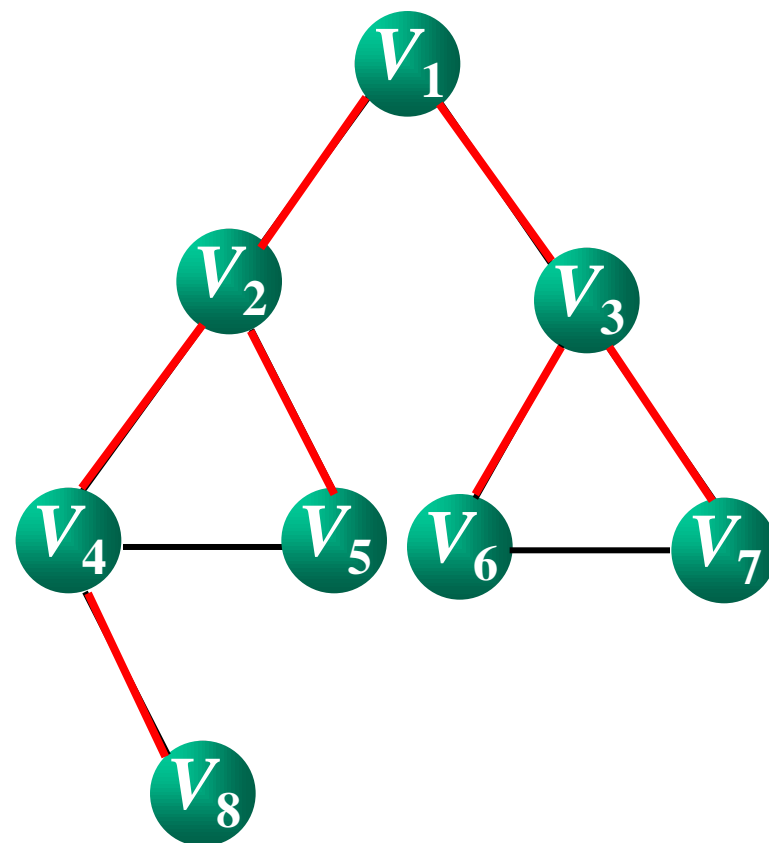


图 \rightarrow 生成树?



(a) 深度优先生成树



(b) 广度优先生成树

生成树

结论：

- ✓ 由深度优先遍历得到的为深度优先生成树，
由广度优先遍历得到的为广度优先生成树。
- ✓ 一个连通图的生成树可能不唯一，由不同的遍历次序、从不同顶点出发进行遍历都会得到不同的生成树。
- ✓ 对于非连通图，通过图的遍历，将得到的是生成森林。

最小生成树(Minimal Spanning Tree, MST)

生成树的代价： 设 $G = (V, E)$ 是一个无向连通网，其生成树上各边的权值之和称为该**生成树的代价**。

最小生成树： 在图 G 所有生成树中，代价最小的生成树称为**最小生成树**。

最小生成树的概念可以应用到许多实际问题中。

例： 在 n 个城市之间建造通信网络，至少要架设 $n-1$ 条通信线路，而每两个城市之间架设通信线路的造价是不一样的，那么如何设计才能使得总造价最小？

普里姆 (Prim) 算法 :

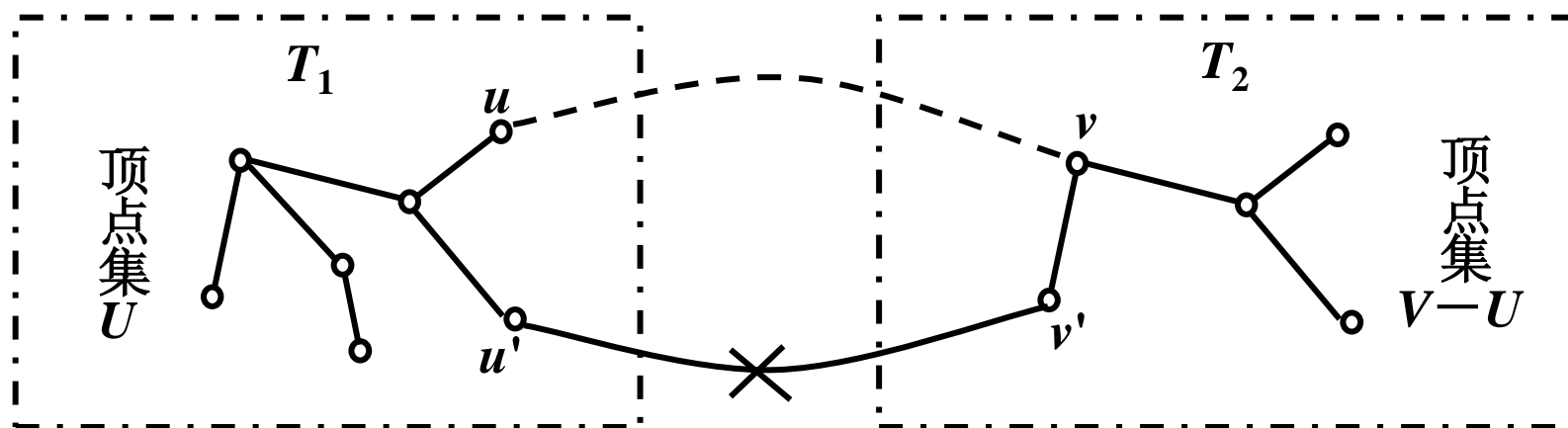
基本思想： 设 $G=(V, E)$ 是具有 n 个顶点的连通网， $T=(U, TE)$ 是 G 的最小生成树， **T 的初始状态为 $U=\{u_0\}$ ($u_0 \in V$)， $TE=\{\}$** ，重复执行下述操作：

在所有 $u \in U$ ， $v \in V-U$ 的边中**找一条代价最小的边 (u, v)** 并入集合 TE ，同时 v 并入 U ，直至 $U=V$ 。

Prim算法的依据:

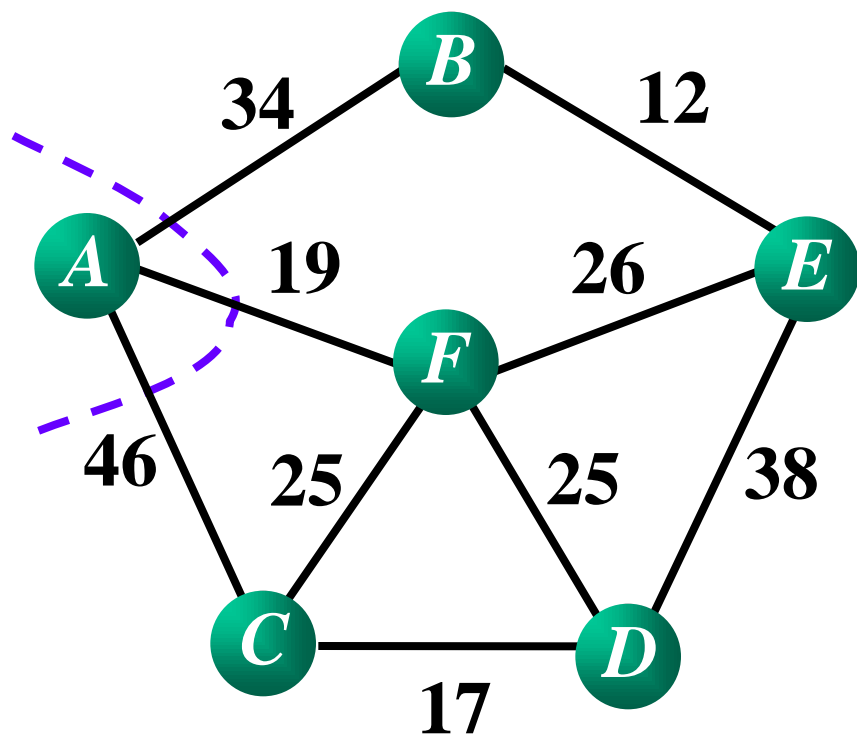
MST性质:

假设 $G=(V, E)$ 是一个无向连通网， U 是顶点集 V 的一个非空子集。若 (u, v) 是一条具有最小权值的边，其中 $u \in U$ ， $v \in V-U$ ，则必存在一棵包含边 (u, v) 的最小生成树。



证明：假设 G 中任何一棵最小生成树都不包含边 (u, v) 。

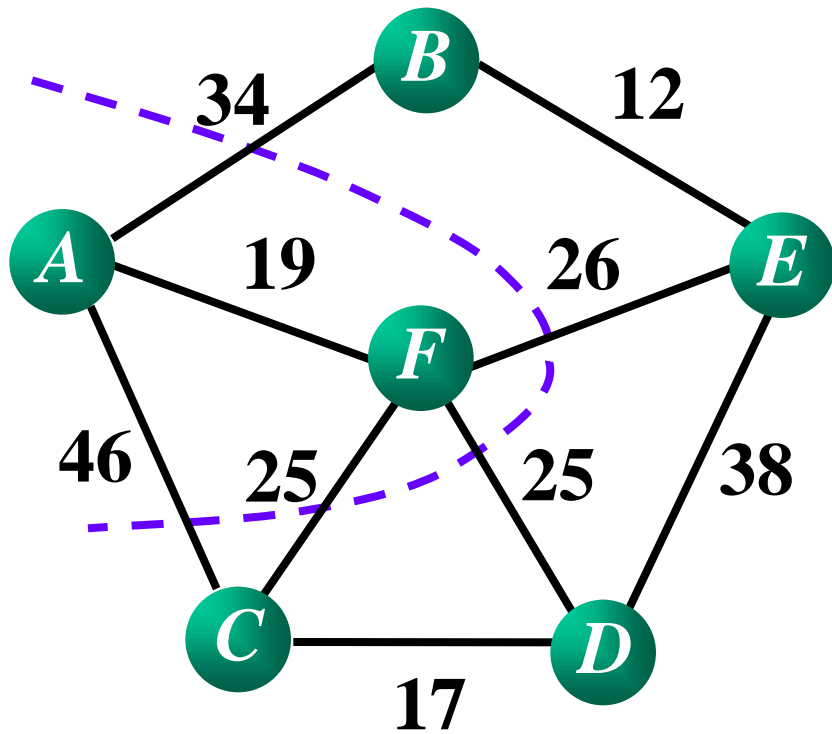
Prim算法示例:



$$U=\{A\}$$

$$V-U=\{B, C, D, E, F\}$$

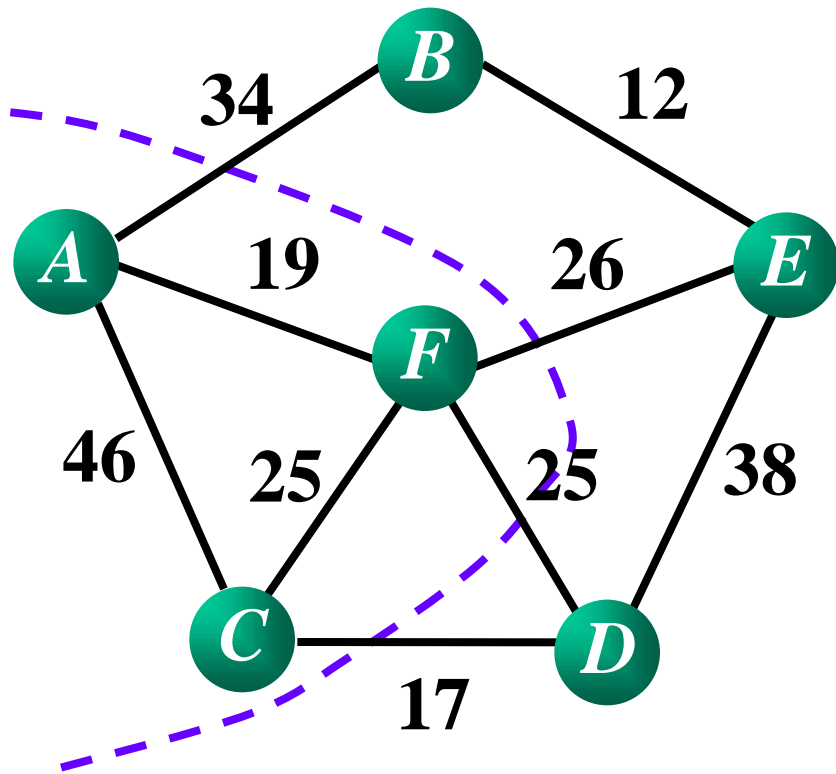
$$\text{cost}=\{(A, B)34, (A, C)46, (A, D)\infty, (A, E)\infty, (A, F)19\}$$



$U=\{A, F\}$

$V-U=\{B, C, D, E\}$

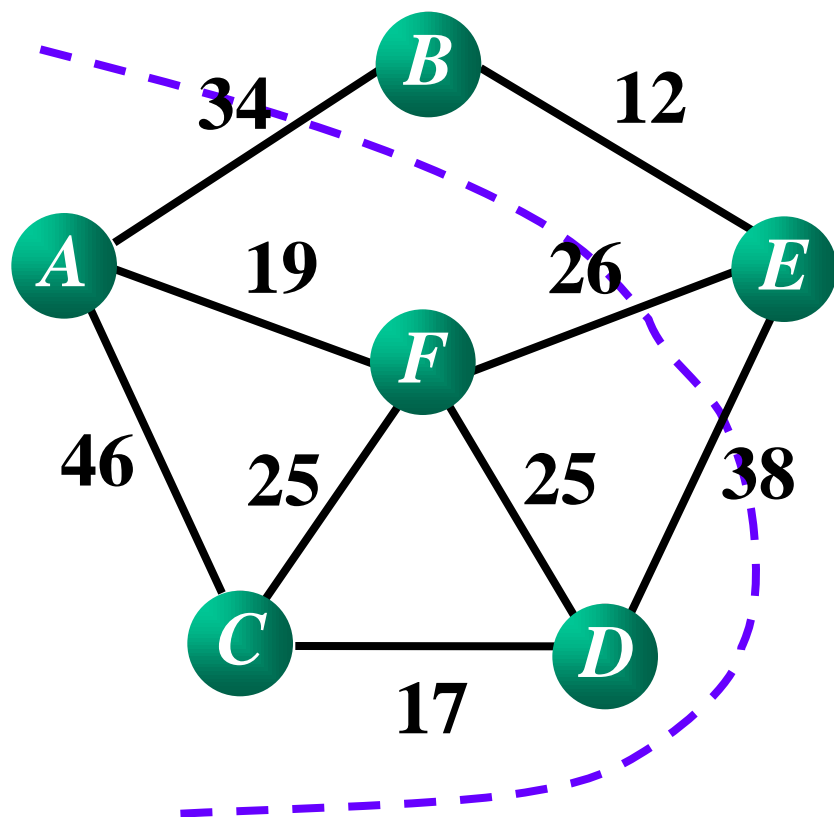
$\text{cost}=\{(A, B)34, (F, C)25, (F, D)25, (F, E)26\}$



$U = \{A, F, C\}$

$V - U = \{B, D, E\}$

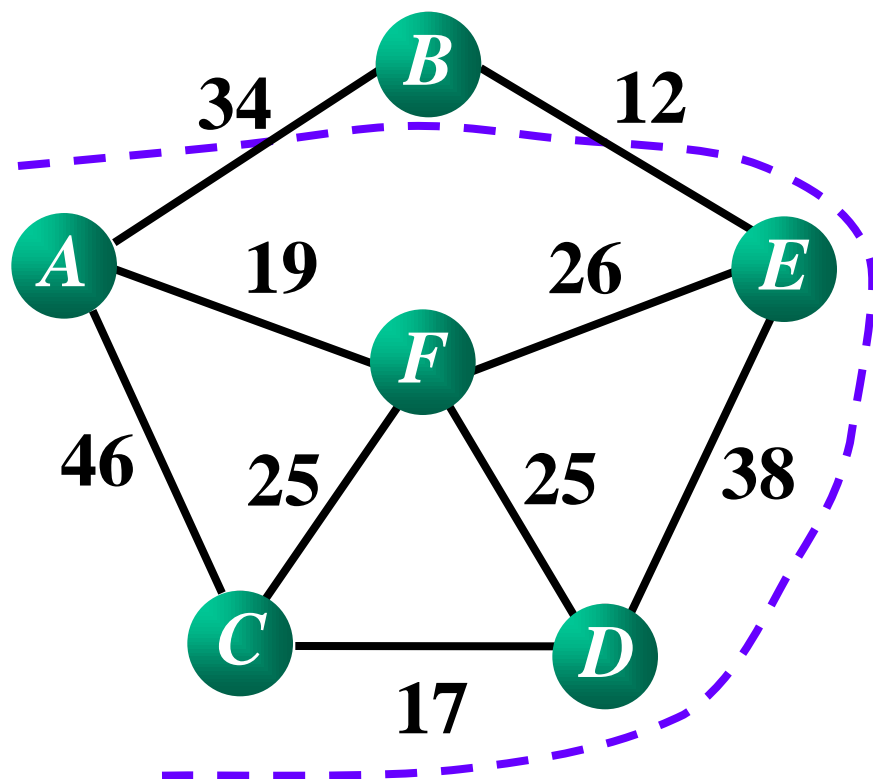
$\text{cost} = \{(A, B) 34, (C, D) 17, (F, E) 26\}$



$U = \{A, F, C, D\}$

$V - U = \{B, E\}$

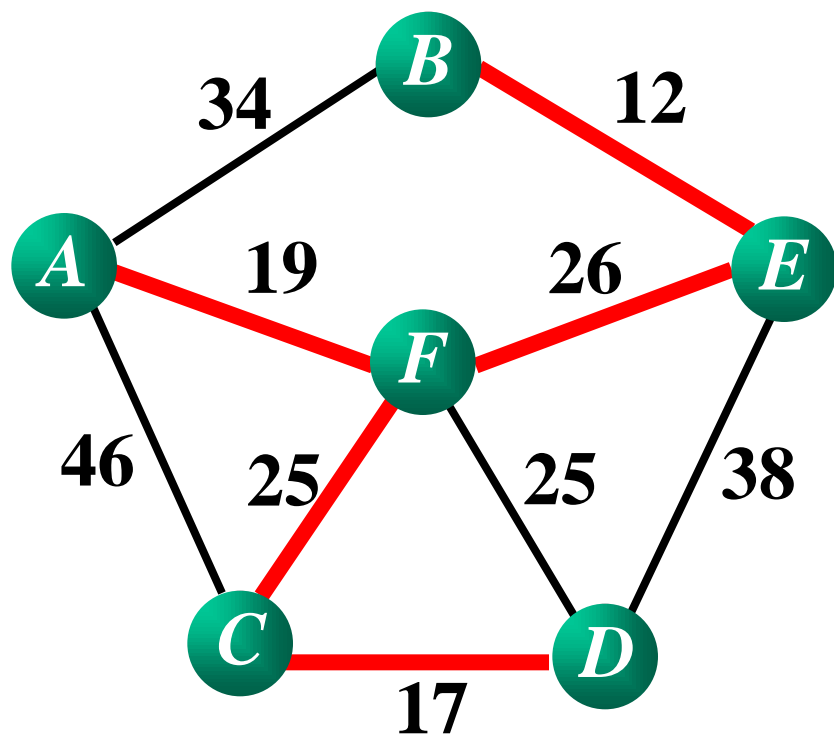
$\text{cost} = \{(A, B)34, (F, E)26\}$



$U = \{A, F, C, D, E\}$

$V - U = \{B\}$

$\text{cost} = \{(E, B) 12\}$



$U = \{A, F, C, D, E, B\}$

$V - U = \{ \}$

普里姆（Prim）算法实现：

关键:是如何找到连接 U 和 $V-U$ 的最短边来扩充生成树。

利用MST性质，可以用下述方法构造**候选最短边集**：

对应 $V-U$ 中的每个顶点，保留从该顶点到 U 中的各顶点的最短边。

相应的数据结构设计:

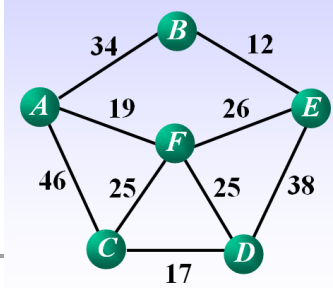
数组**lowcost**[n]: 用来保存集合 $V-U$ 中各顶点与集合 U 中顶点最短边的权值, **lowcost**[v]=0表示顶点v已加入最小生成树中;

数组**adjvex**[n]: 用来保存依附于该边 (集合 $V-U$ 中各顶点与集合 U 中顶点的最短边) 在集合 U 中的顶点。

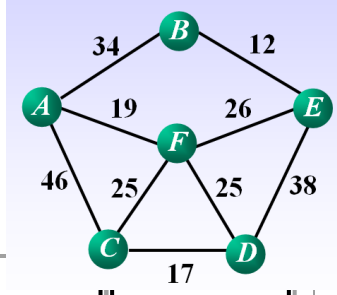
④ 如何用数组**lowcost**和**adjvex**表示候选最短边集?

$$\begin{cases} \text{lowcost}[i]=w \\ \text{adjvex}[i]=k \end{cases}$$
 表示顶点 v_i 和顶点 v_k 之间的权值为 w ,
 其中: $v_i \in V-U$ 且 $v_k \in U$

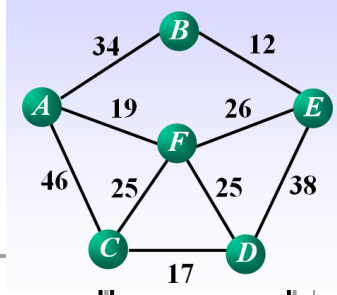
Prim算法最小生成树生成过程:



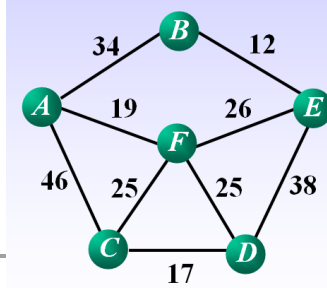
<div>i</div> <div>数组</div>	B (i=1)	C (i=2)	D (i=3)	E (i=4)	F (i=5)	U	V-U	输出
adjvex	A	A	A	A	A			
lowcost	34	46	∞	∞	19	{A}	{B, C, D, E, F}	(A F) 19



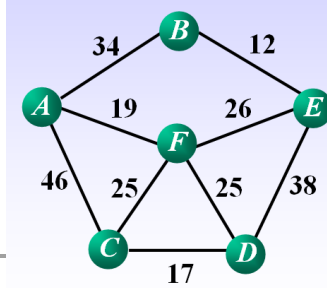
<div><div>i</div><div>数组</div></div>	B (i=1)	C (i=2)	D (i=3)	E (i=4)	F (i=5)	U	V-U	输出
adjvex	A	A	A	A	A			
lowcost	34	46	∞	∞	19	{A}	{B, C, D, E, F}	(A F) 19
adjvex	A	F	F	F				
lowcost	34	25	25	26		{A, F}	{B, C, D, E}	(F C) 25



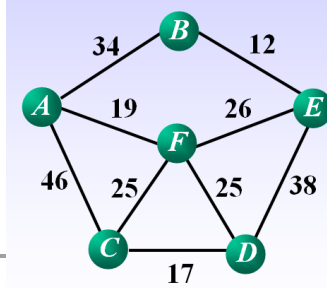
<div><div>i</div><div>数组</div></div>	B (i=1)	C (i=2)	D (i=3)	E (i=4)	F (i=5)	U	V-U	输出
adjvex lowcost	A 34	A 46	A ∞	A ∞	A 19	{A}	{B, C, D, E, F}	(A F) 19
adjvex lowcost	A 34	F 25	F 25	F 26		{A, F}	{B, C, D, E}	(F C) 25
adjvex lowcost	A 34		C 17	F 26		{A, F, C}	{B, D, E}	(C D) 17



<div> <div>i</div> <div>数组</div> </div>	B (i=1)	C (i=2)	D (i=3)	E (i=4)	F (i=5)	U	V-U	输出
adjvex lowcost	A 34	A 46	A ∞	A ∞	A 19	{A}	{B, C, D, E, F}	(A F) 19
adjvex lowcost	A 34	F 25	F 25	F 26		{A, F}	{B, C, D, E}	(F C) 25
adjvex lowcost	A 34		C 17	F 26		{A, F, C}	{B, D, E}	(C D) 17
adjvex lowcost	A 34			F 26		{A, F, C, D}	{B, E}	(F E) 26



<div>数组 \ i</div>	B (i=1)	C (i=2)	D (i=3)	E (i=4)	F (i=5)	U	V-U	输出
adjvex lowcost	A 34	A 46	A ∞	A ∞	A 19	{A}	{B, C, D, E, F}	(A F) 19
adjvex lowcost	A 34	F 25	F 25	F 26		{A, F}	{B, C, D, E}	(F C) 25
adjvex lowcost	A 34		C 17	F 26		{A, F, C}	{B, D, E}	(C D) 17
adjvex lowcost	A 34			F 26		{A, F, C, D}	{B, E}	(F E) 26
adjvex lowcost	E 12					{A, F, C, D, E}	{B}	(E B) 12



<div>数组 \ i</div>	B (i=1)	C (i=2)	D (i=3)	E (i=4)	F (i=5)	U	V-U	输出
adjvex lowcost	A 34	A 46	A ∞	A ∞	A 19	{A}	{B, C, D, E, F}	(A F) 19
adjvex lowcost	A 34	F 25	F 25	F 26		{A, F}	{B, C, D, E}	(F C) 25
adjvex lowcost	A 34		C 17	F 26		{A, F, C}	{B, D, E}	(C D) 17
adjvex lowcost	A 34			F 26		{A, F, C, D}	{B, E}	(F E) 26
adjvex lowcost	E 12					{A, F, C, D, E}	{B}	(E B) 12
adjvex lowcost						{A, F, C, D, E, B}	{ }	

Prim算法——伪代码描述:

1. 初始化两个辅助数组lowcost和adjvex;
2. 输出顶点 u_0 , 将顶点 u_0 加入集合U中;
3. 重复执行下列操作 $n-1$ 次
 - 3.1 在lowcost中选取最短边, 取adjvex中对应的顶点序号k;
 - 3.2 输出顶点k和对应的权值;
 - 3.3 将顶点k加入集合U中;
 - 3.4 调整数组lowcost和adjvex;

```
void Prim(MGraph G)
{
    for (i=1; i<G.vertexNum; i++)
        //初始化两个辅助数组lowcost和adjvex
        {
            lowcost[i]=G.arc[0][i];
            adjvex[i]=0;
        }
    lowcost[0]=0; //将顶点0加入集合U中
```

```
for (i=1; i<G.vertexNum; i++)
{
    //在lowcost中寻找最短边的顶点k
    k=MinEdge(lowcost, G.vertexNum);
    //输出加入TE中的边
    cout<<"("<<k<<adjvex[k]<<")"<<lowcost[k];
    lowcost[k]=0;        //将顶点v加入集合U中
    for (j=1; j<G.vertexNum; j++)
        //调整数组lowcost和adjvex
        if G.arc[k][j]<lowcost[j] {
            lowcost[j]=G.arc[k][j];
            adjvex[j]=k; }
    }//for
} //end
```

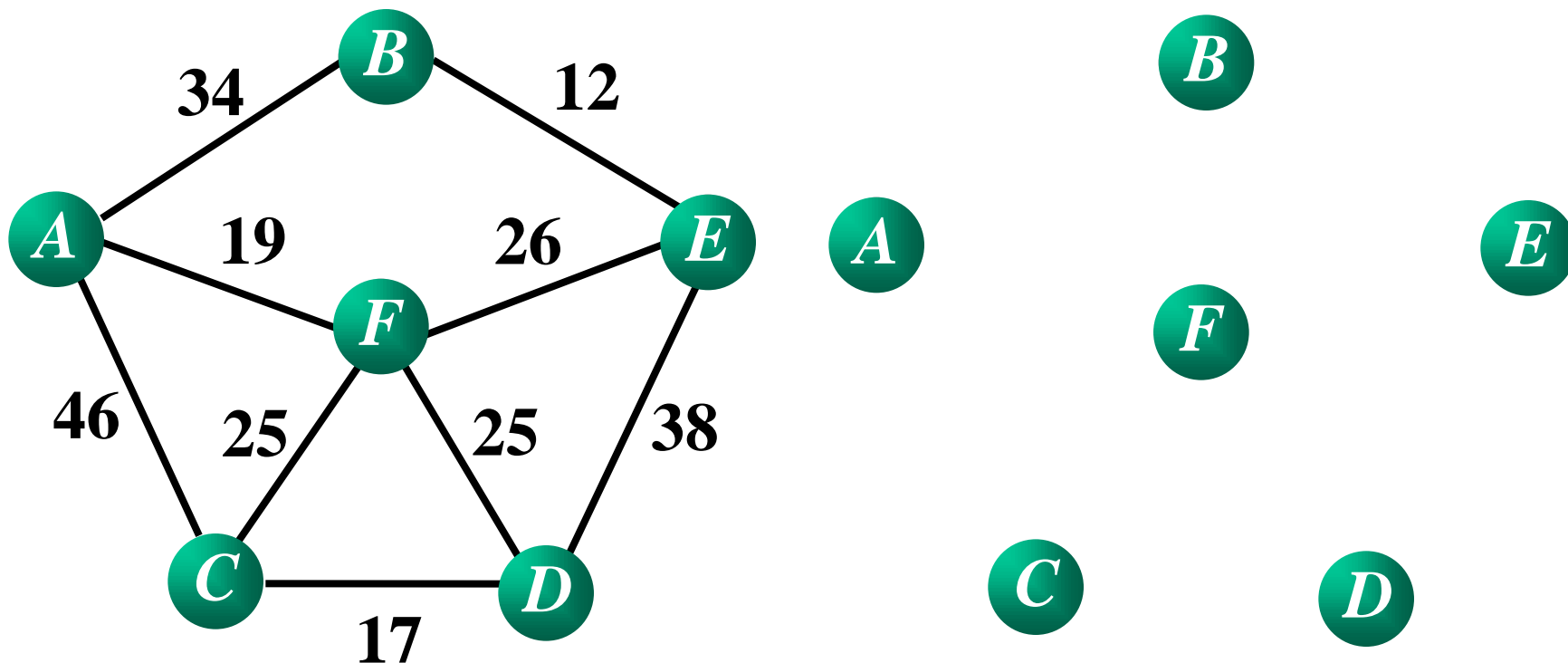

克鲁斯卡尔 (Kruskal) 算法:

基本思想: 设无向连通网为 $G=(V, E)$, 令 G 的最小生成树为 $T=(U, TE)$, 其初态为 $U=V$, $TE=\{ \}$, 然后按照边的权值由小到大的顺序, 考察 G 的边集 E 中的各条边。

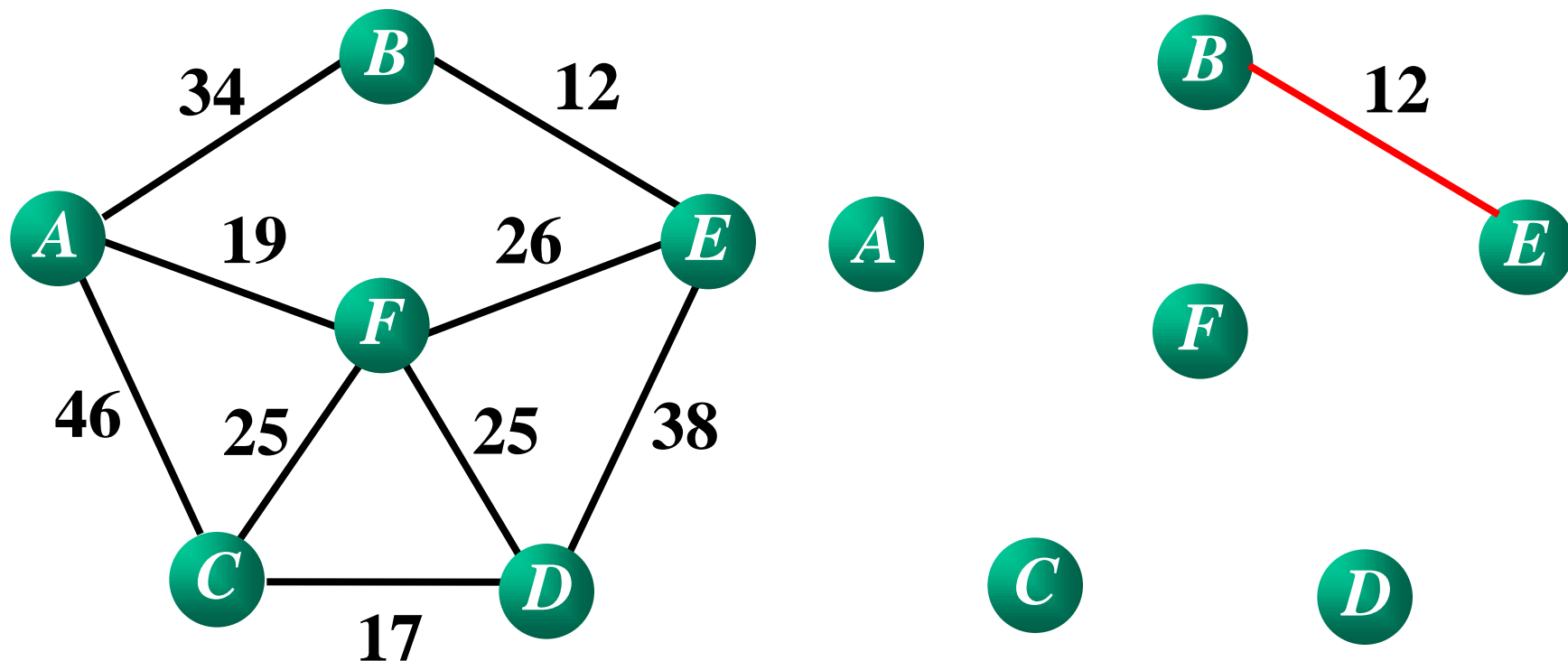
若被考察的边的两个顶点属于 T 的两个不同的连通分量, 则将此边作为最小生成树的边加入到 T 中, 同时把两个连通分量连接为一个连通分量;

若被考察边的两个顶点属于同一个连通分量, 则舍去此边, 以免造成回路, 如此下去, 当 T 中的连通分量个数为1时, 此连通分量便为 G 的一棵最小生成树。

克鲁斯卡尔算法示例：

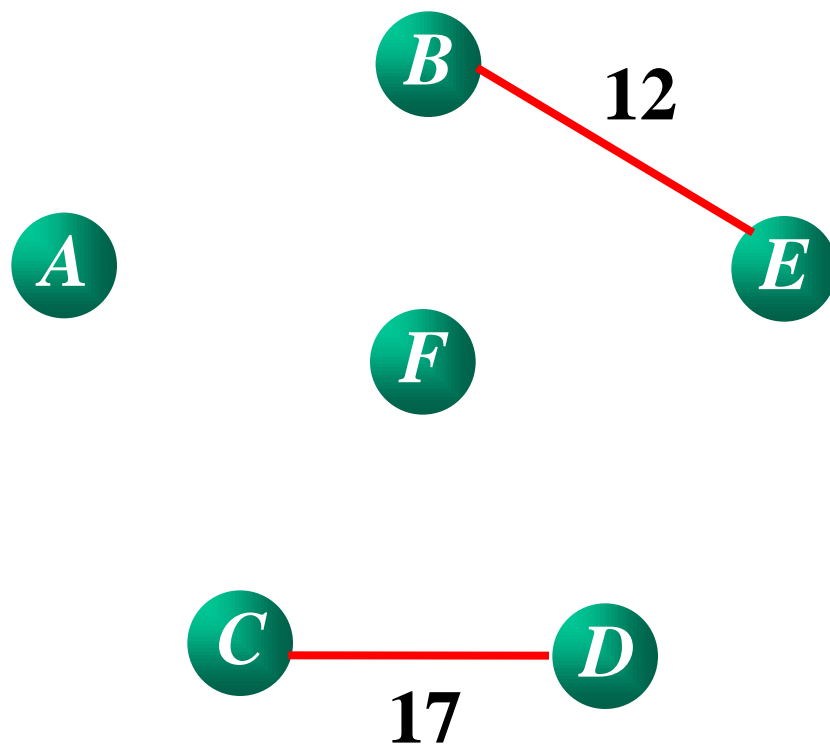
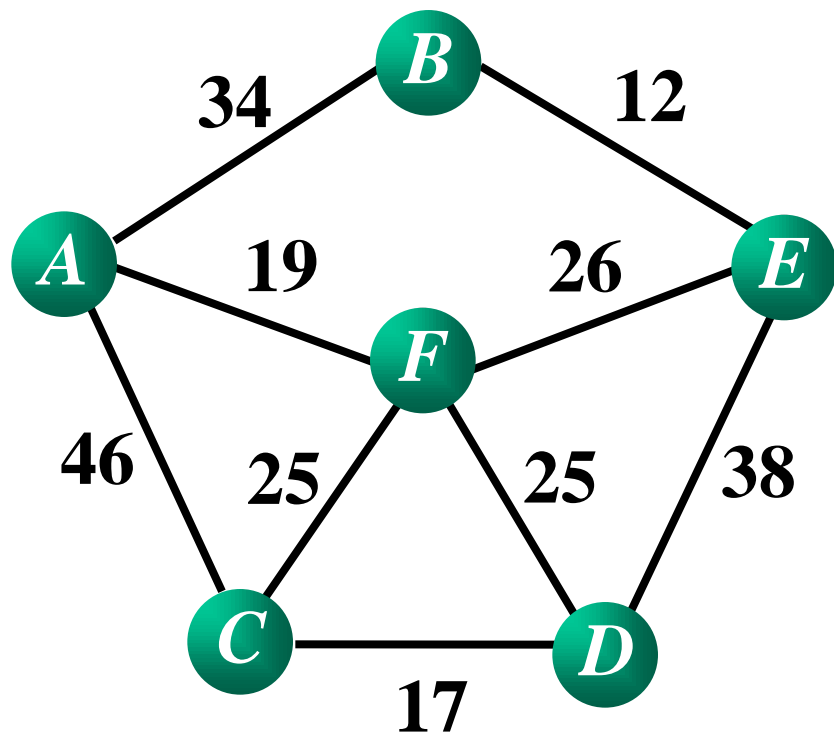


连通分量 = {A}, {B}, {C}, {D}, {E}, {F}



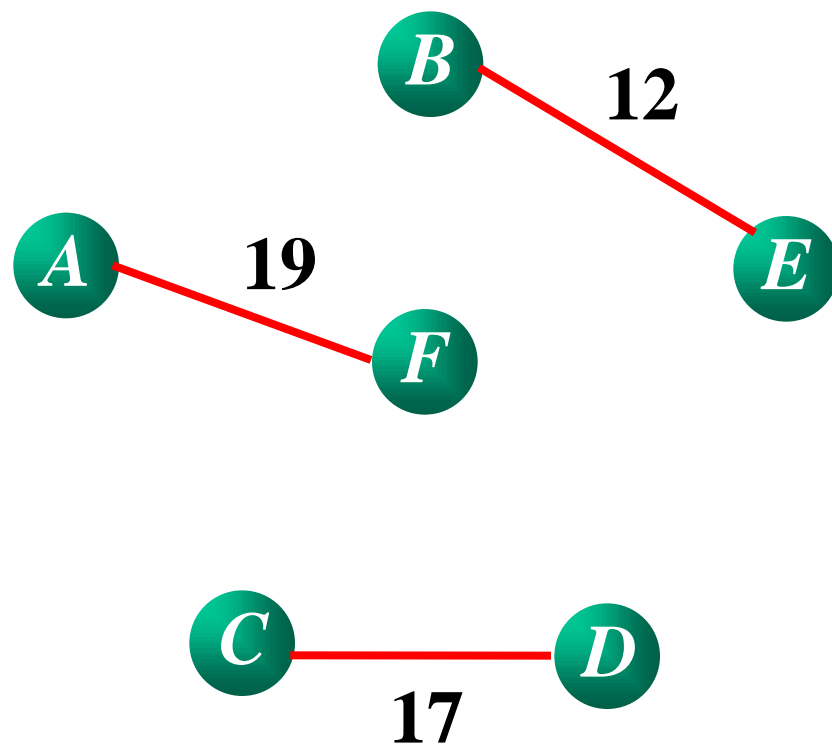
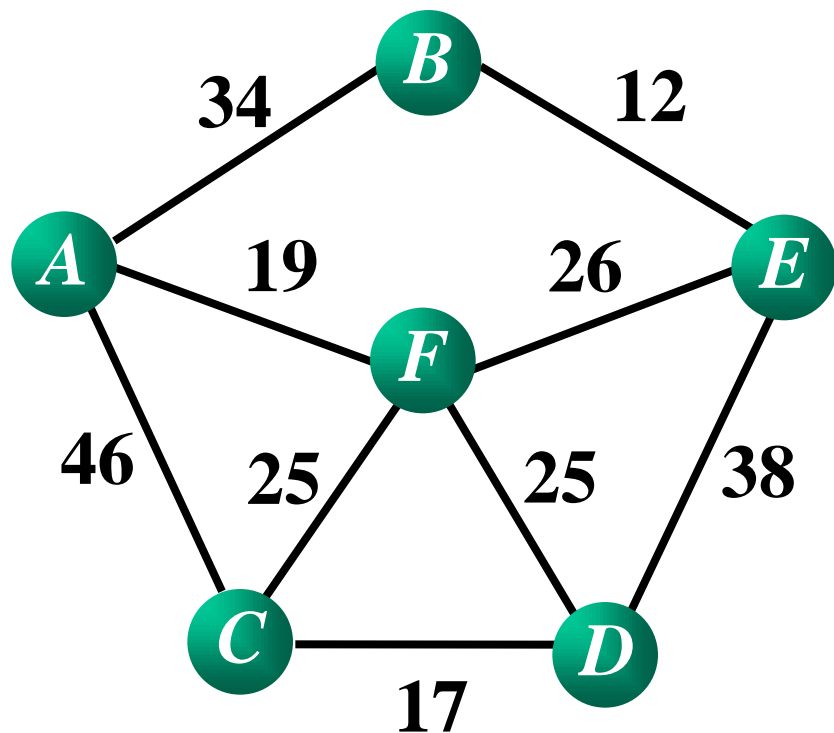
连通分量 = {A}, {B}, {C}, {D}, {E}, {F}

连通分量 = {A}, {B, E}, {C}, {D}, {F}



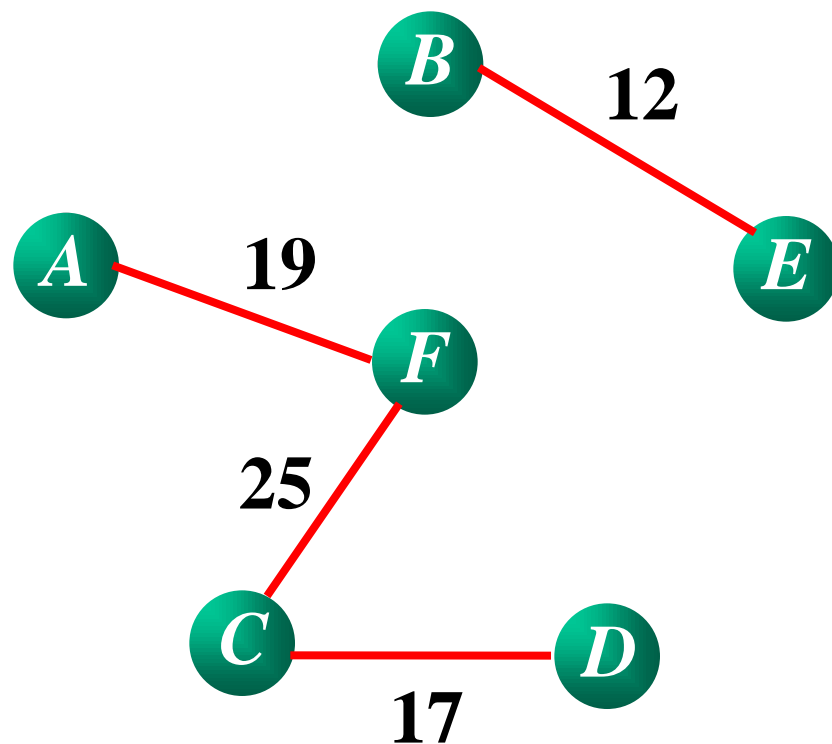
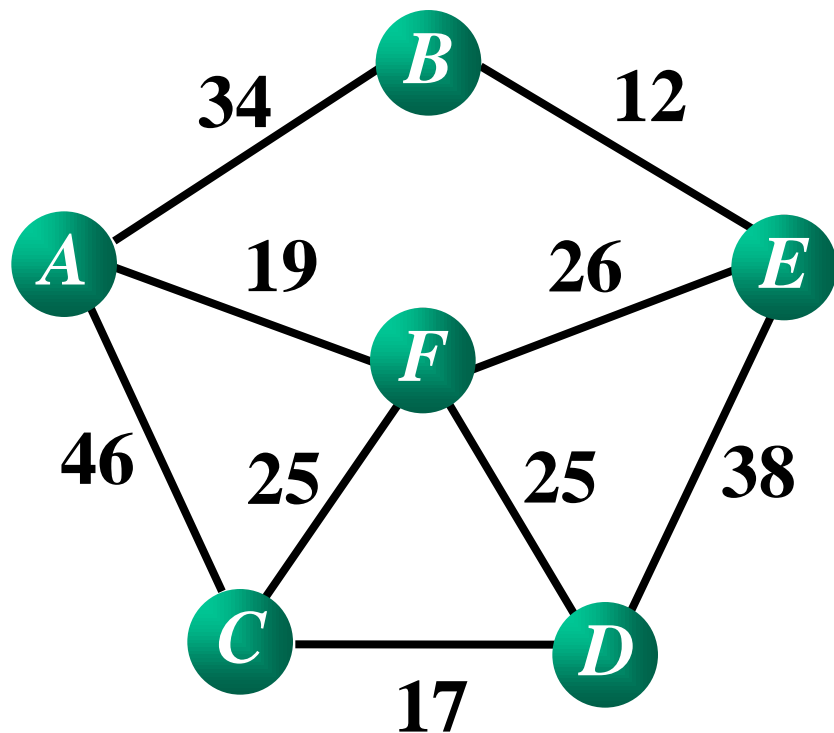
连通分量 = {A}, {F}, {B, E}, {C}, {D}

连通分量 = {A}, {F}, {B, E}, {C, D}



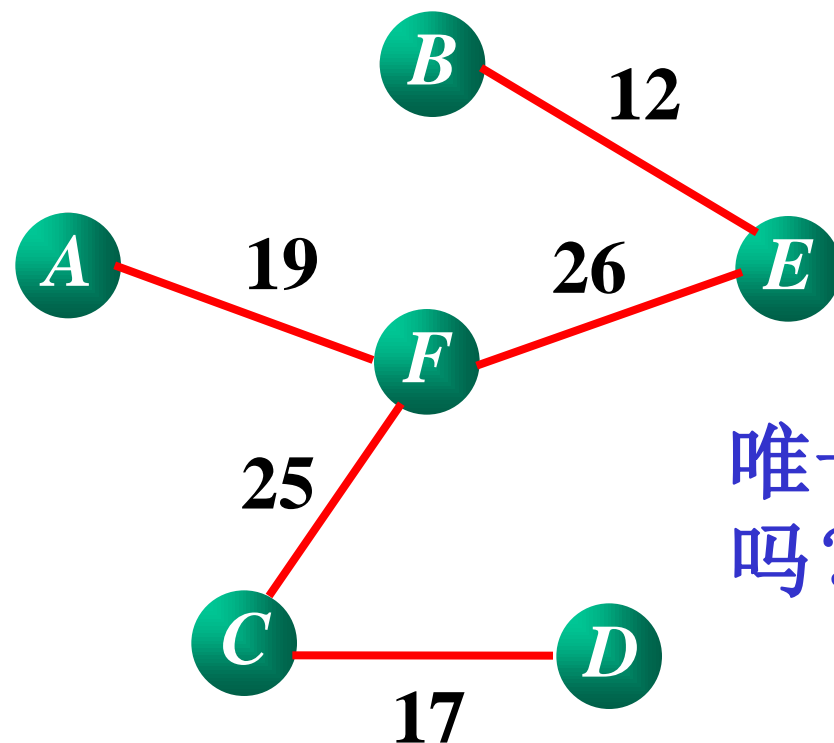
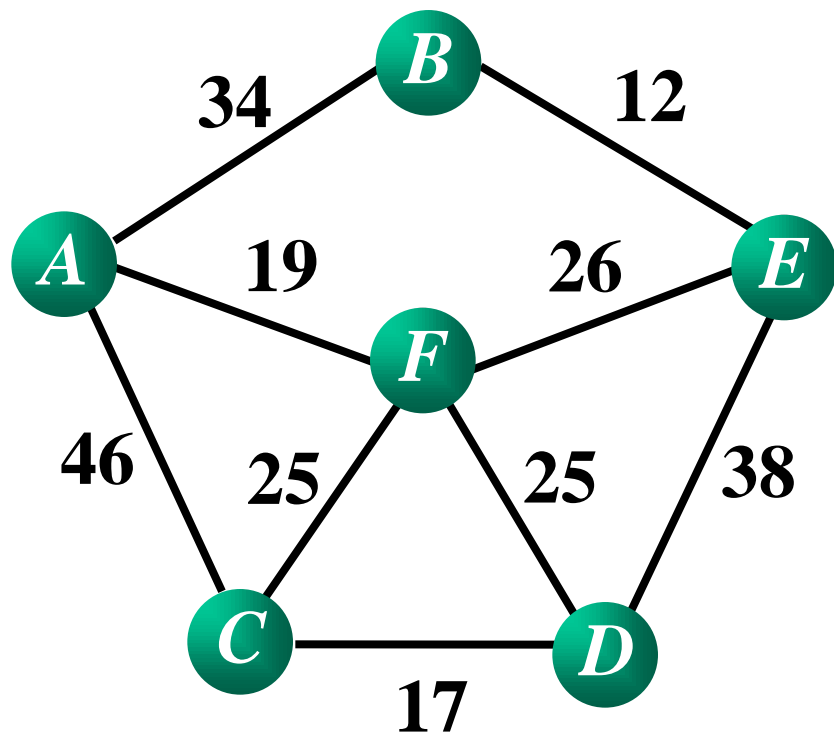
连通分量 = {A}, {F}, {B, E}, {C, D}

连通分量 = {A, F}, {B, E}, {C, D}



连通分量 = {A, F}, {B, E}, {C, D}

连通分量 = {A, F, C, D}, {B, E}



唯一吗？

连通分量 = {A, F, C, D}, {B, E}

连通分量 = {A, F, C, D, B, E}

Kruskal算法——伪代码描述:

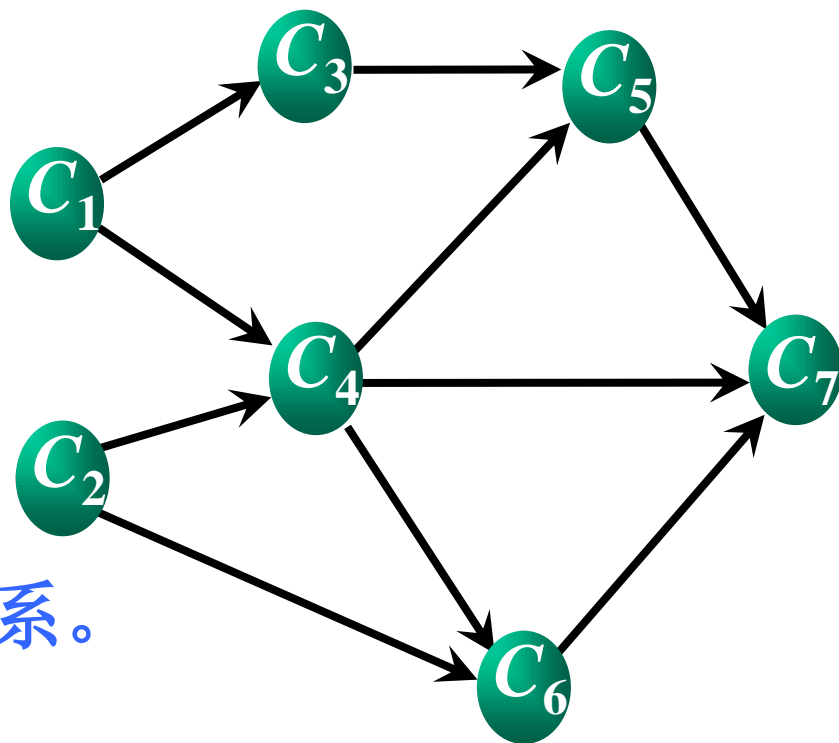
1. 初始化: $U=V$; $TE=\{ \}$;
2. 循环 直到T中的连通分量个数为1
 - 2.1 在E中寻找最短边 (u,v) ;
 - 2.2 如果顶点 u 、 v 位于T的两个不同连通分量, 则
 - 2.2.1 将边 (u,v) 并入TE;
 - 2.2.2 将这两个连通分量合为一个;
 - 2.3 在E中标记边 (u,v) , 使得 (u,v) 不参加后续最短边的选取。

Prim算法与Kruskal算法比较:

	空间性能	时间性能	适用范围	唯一性
普里姆	$O(n^2)$	$O(n^2)$	稠密网	不唯一
克鲁斯卡尔	$O(n+e)$	$O(e \log_2 e)$	稀疏网	不唯一

6.5 图的应用举例：（2）拓扑排序

考虑如下图：



表示 课程之间的先修关系。

对应的模型是 有向图。

① 在这个图中出现回路意味着什么？

AOV网:

在一个表示工程的有向图中，用顶点表示活动，用弧表示活动之间的优先关系，称这样的有向图为**顶点表示活动的网(activity on vertex network)**，简称**AOV网**。

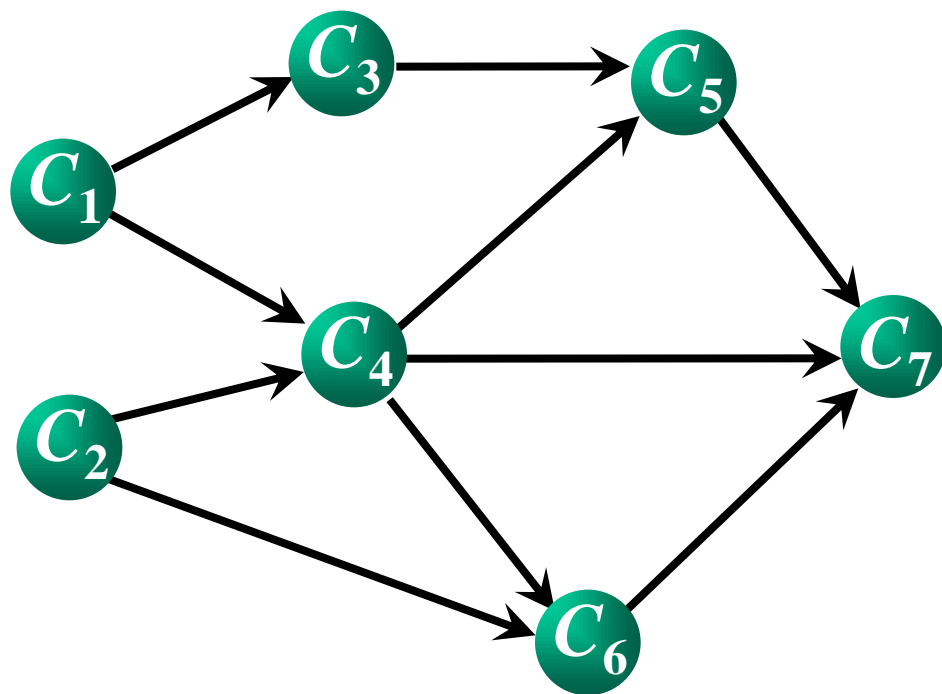
① AOV网中出现回路意味着什么？

AOV网的特点:

1. AOV网中的弧表示活动之间存在的某种制约关系。
2. AOV网中不能出现回路。

例如：教学计划编排问题

编号	课程名称	先修课
C_1	高等数学	无
C_2	计算机导论	无
C_3	离散数学	C_1
C_4	程序设计	C_1, C_2
C_5	数据结构	C_3, C_4
C_6	计算机原理	C_2, C_4
C_7	数据库原理	C_4, C_5, C_6



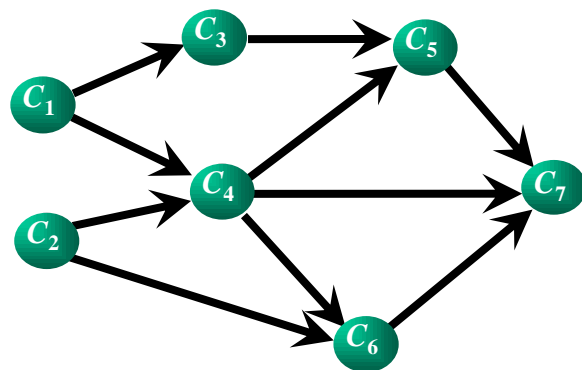
那么，如何判断有没有回路呢？ 拓扑排序

拓扑序列： 设 $G=(V, E)$ 是一个具有 n 个顶点的有向图， V 中的顶点序列 v_1, v_2, \dots, v_n 称为一个**拓扑序列**，当且仅当满足下列条件：若从顶点 v_i 到 v_j 有一条路径，则在顶点序列中顶点 v_i 必在顶点 v_j 之前。

那么，拓扑序列唯一吗？ 例如：

$C_1 C_2 C_4 C_3 C_5 C_6 C_7$

$C_1 C_3 C_2 C_4 C_5 C_6 C_7$

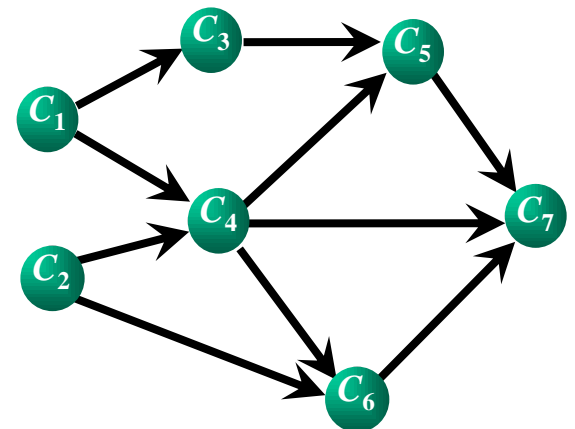


所谓**拓扑排序** (topological sort) :

对一个有向图构造**拓扑序列**的过程称为拓扑排序 。

拓扑排序基本思想:

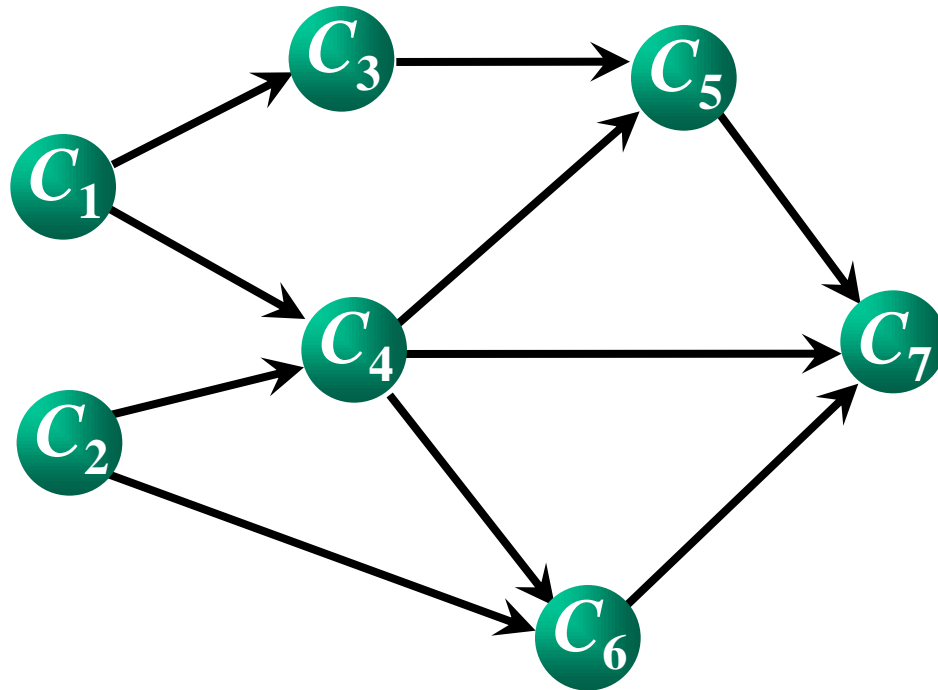
- (1) 从AOV网中选择一个没有前驱的顶点并且输出;
- (2) 从AOV网中删去该顶点, 并且删去所有以该顶点为尾的弧;
- (3) 重复上述两步, 直到全部顶点都被输出, 或AOV网中不存在没有前驱的顶点。



① 拓扑排序的结果说明了什么？

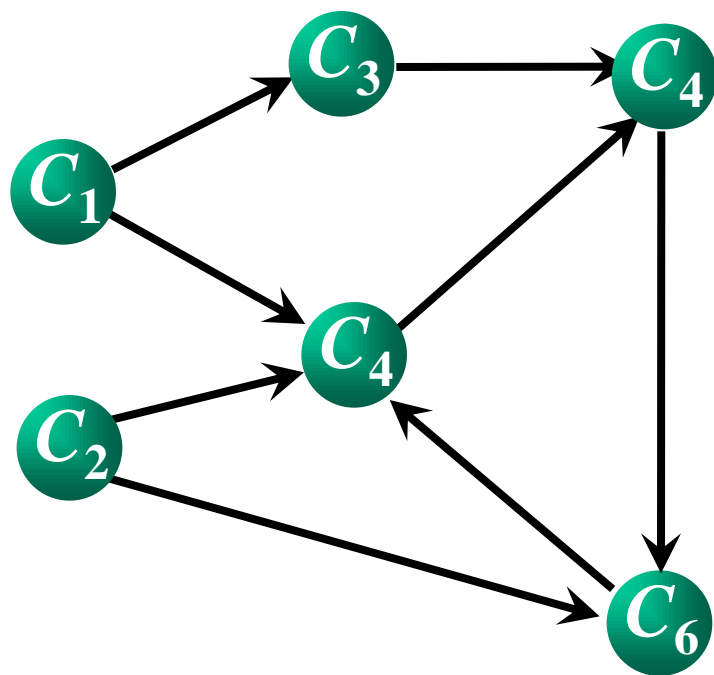
在AOV网中全部顶点都输出，则AOV网中不存在回路；
若AOV网中顶点未被全部输出，则AOV网中存在回路。

拓扑排序过程:



拓扑序列: $C_1, C_2, C_3, C_4, C_5, C_6, C_7$

拓扑排序过程:



说明AOV网中
存在回路。

拓扑序列: $C_1, C_2, C_3,$

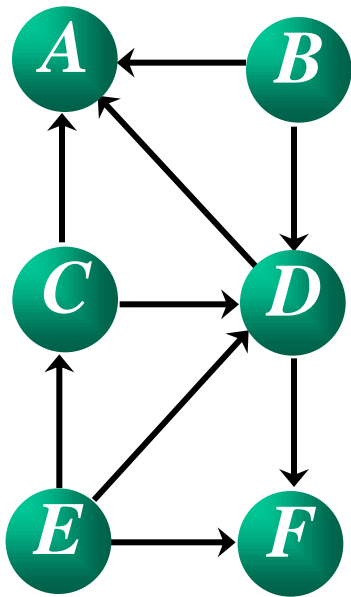
相应的数据结构设计：

1. 图的存储结构：采用邻接表存储，并在顶点表中增加一个入度域。

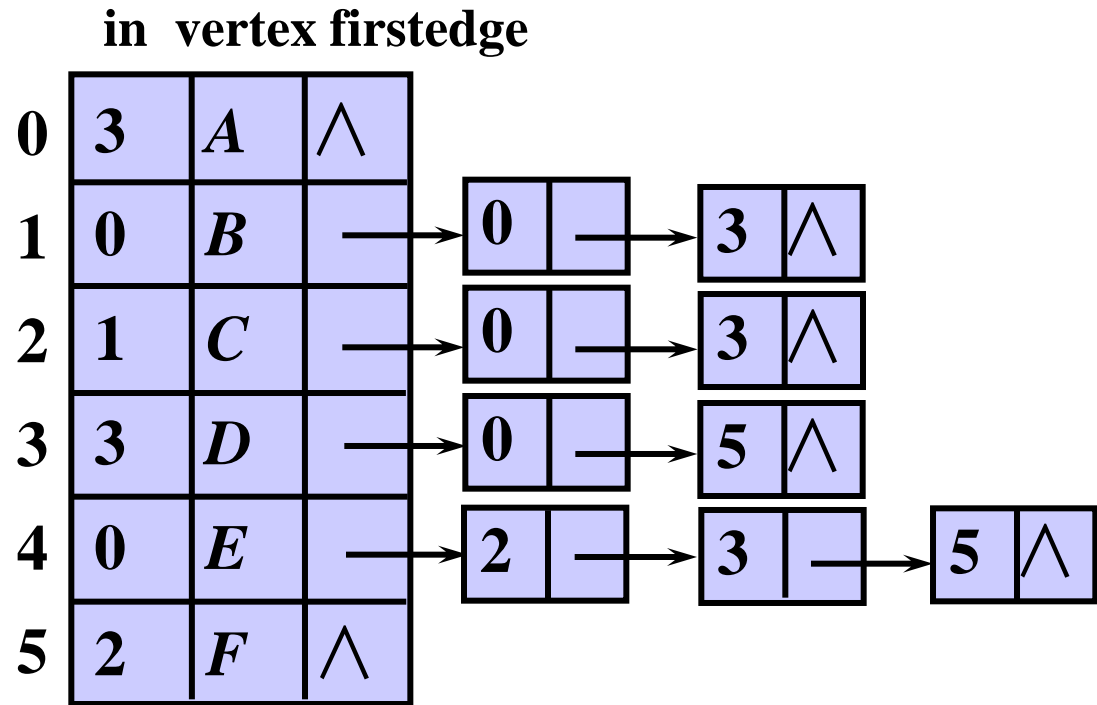
in	vertex	firstedge
-----------	---------------	------------------

顶点表结点

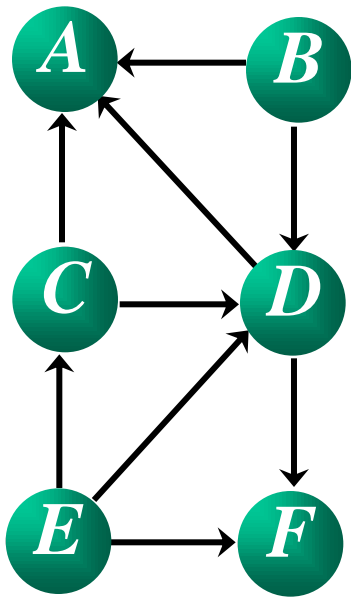
2. 栈S：存储所有无前驱的顶点。



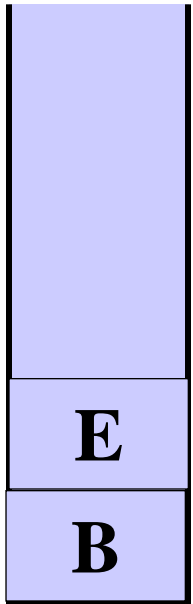
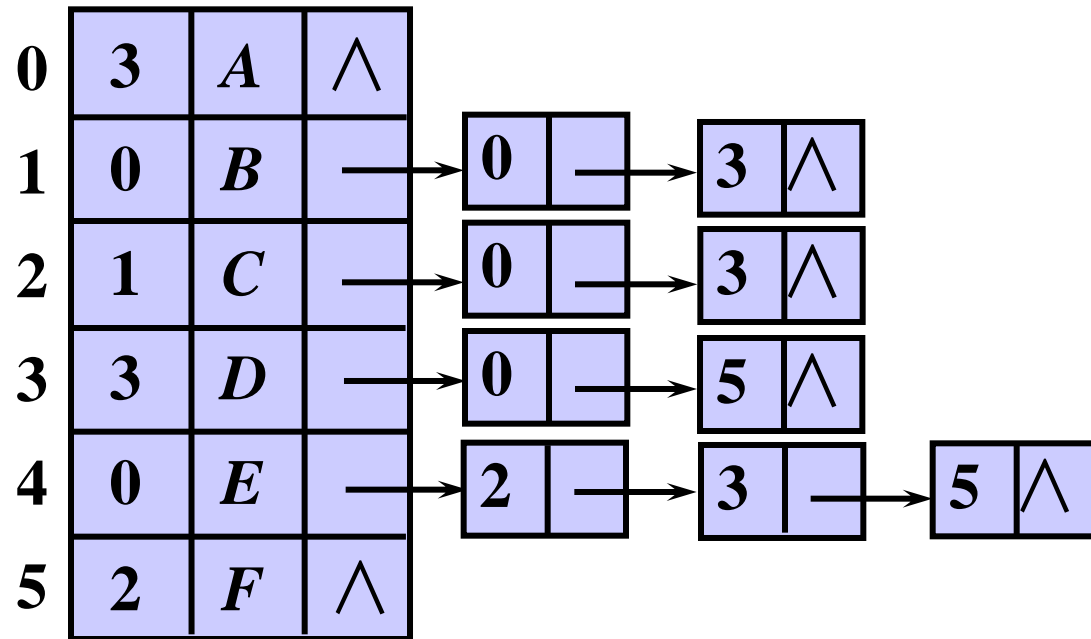
(a) 一个AOV网

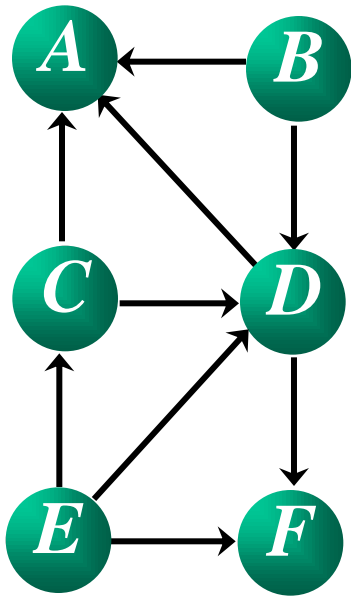


(b) AOV网的邻接表存储

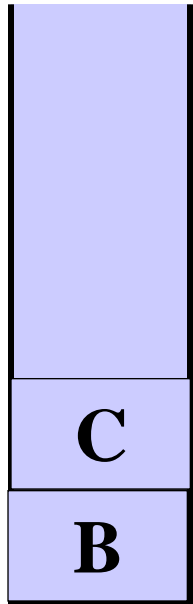
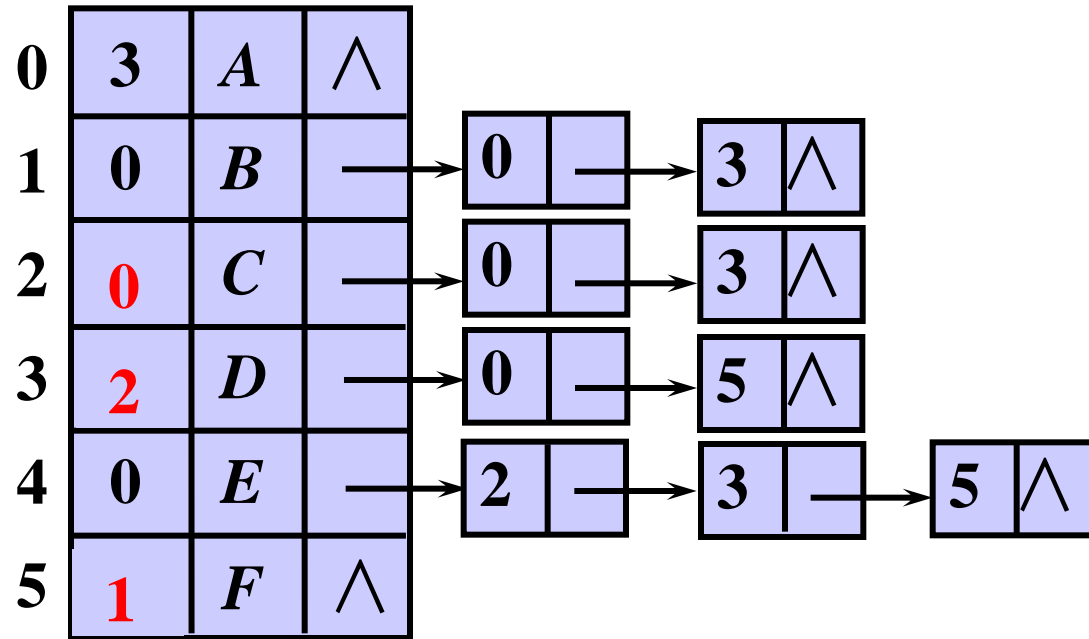


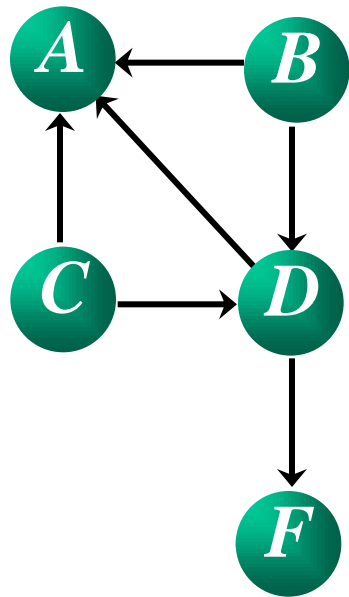
in vertex firstedge



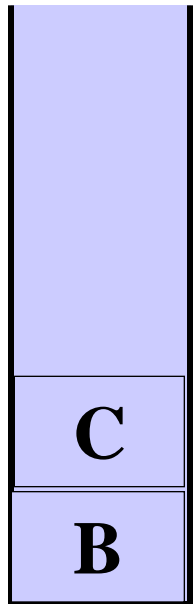
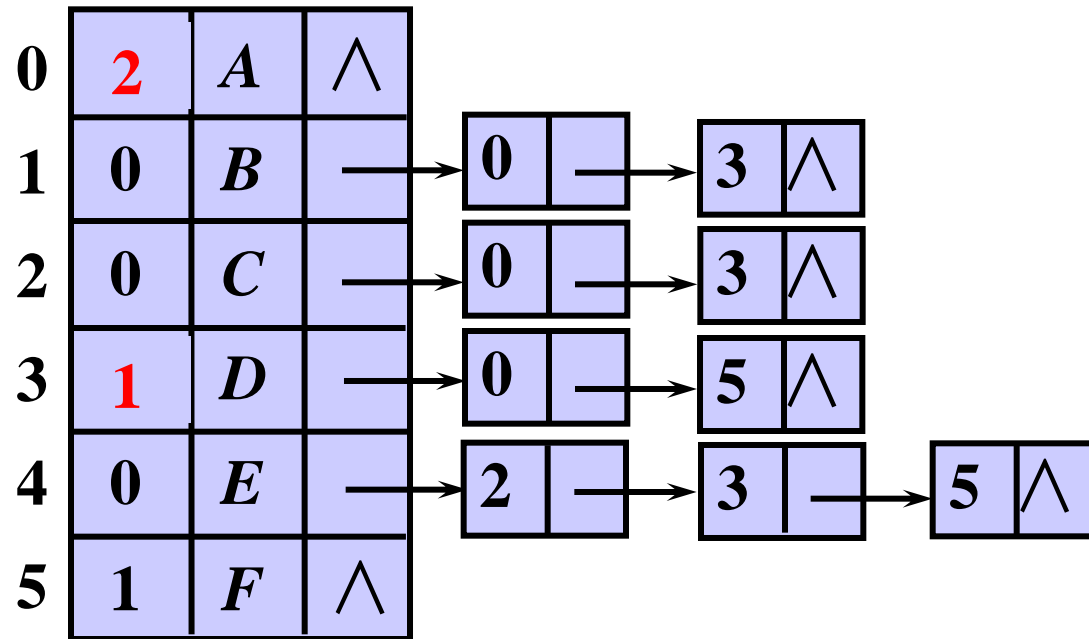


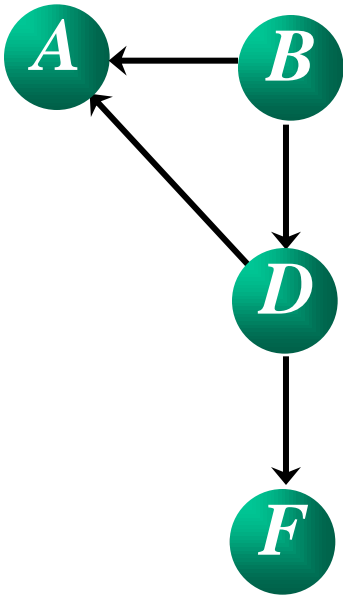
in vertex firstedge





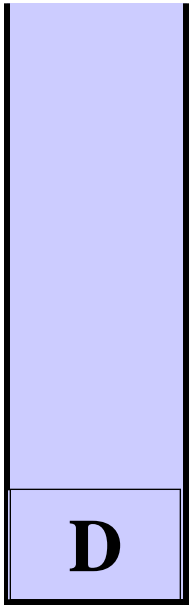
in vertex firstedge

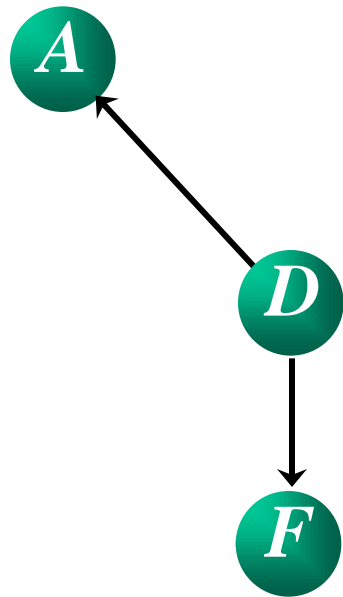




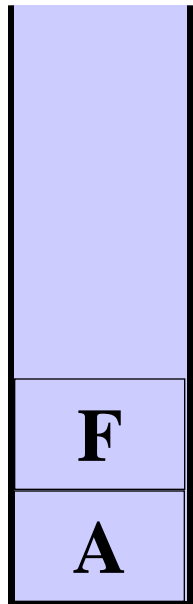
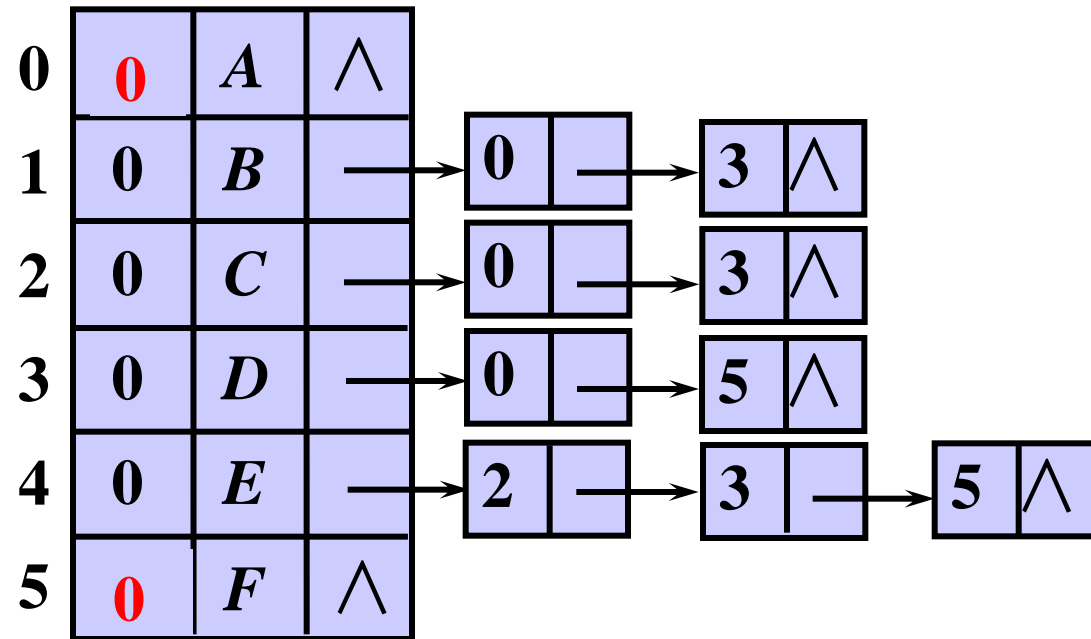
in vertex firstedge

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



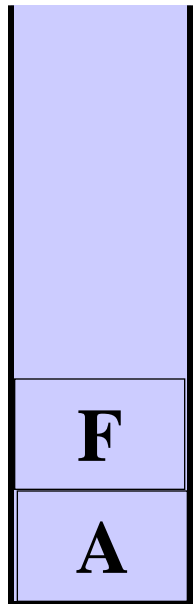
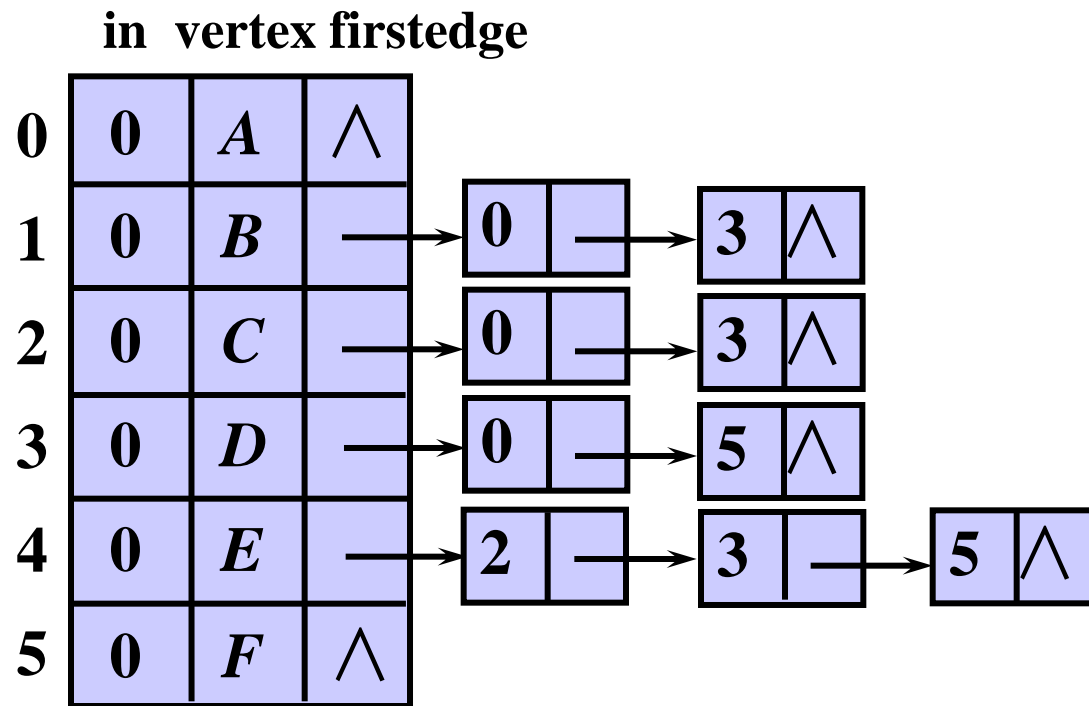


in vertex firstedge



A

F



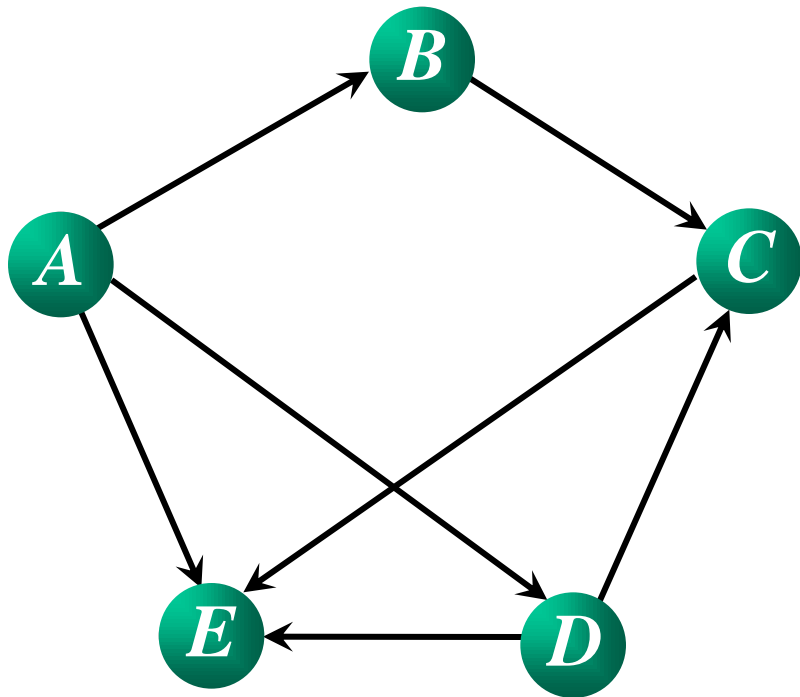
拓扑排序算法——伪代码描述：

1. 栈S初始化；累加器count初始化；
2. 扫描顶点表，将没有前驱的顶点压栈；
3. 当栈S非空时循环
 - 3.1 v_j =退出栈顶元素；输出 v_j ；累加器count加1；
 - 3.2 将顶点 v_j 的各个邻接点的入度减1；
 - 3.3 将新的入度为0的顶点入栈；
4. if (count<vertexNum) 输出有回路信息；

6.5 图的应用举例：（3）最短路径

最短路径(shortest path):

在一个图中，最短路径是指两顶点之间经历的边数最少的路径。



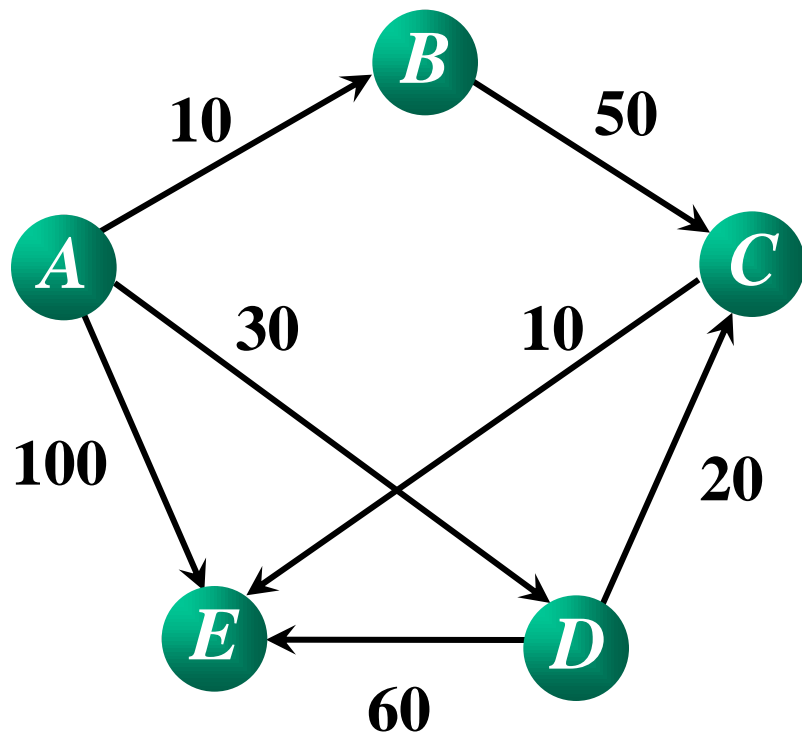
AE: 1

ADE: 2

ADCE: 3

ABCE: 3

在一个网中，最短路径是指两顶点之间经历的边上权值之和最短的路径。



AE: 100

ADE: 90

ADCE: 60

ABCE: 70

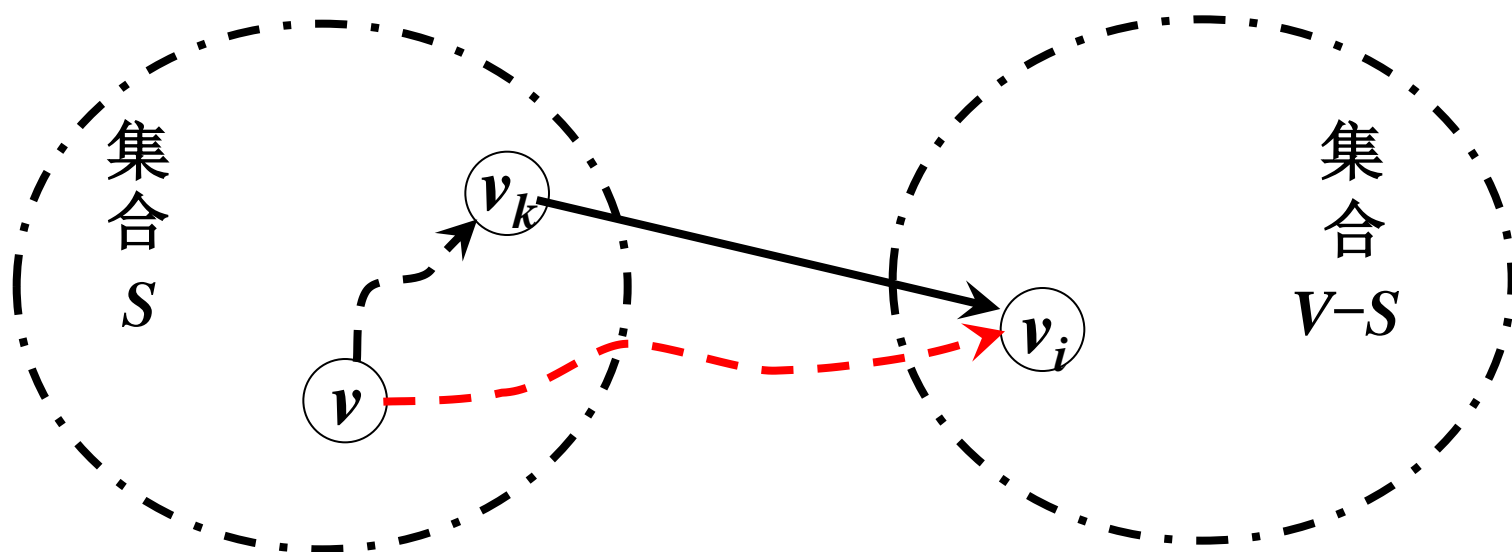
1、单源点最短路径问题

问题描述：给定带权有向图 $G=(V, E)$ 和源点 $v \in V$ ，求从 v 到 G 中其余各顶点的最短路径。

应用实例——计算机网络传输的问题：怎样找到一种最经济的方式，从一台计算机向网上所有其它计算机发送一条消息。

迪杰斯特拉（**Dijkstra**）提出了一个按路径长度递增的次序产生最短路径的算法——**Dijkstra**算法。

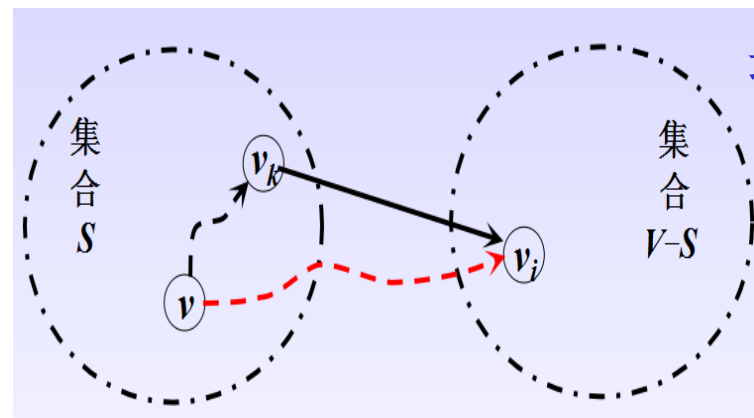
Dijkstra算法的基本思想:



两个规律:

- 最短路径是按路径长度递增的次序产生的;
- 最短路径上只可能经过已经产生终点的顶点。

Dijkstra算法的基本思想 (续) :

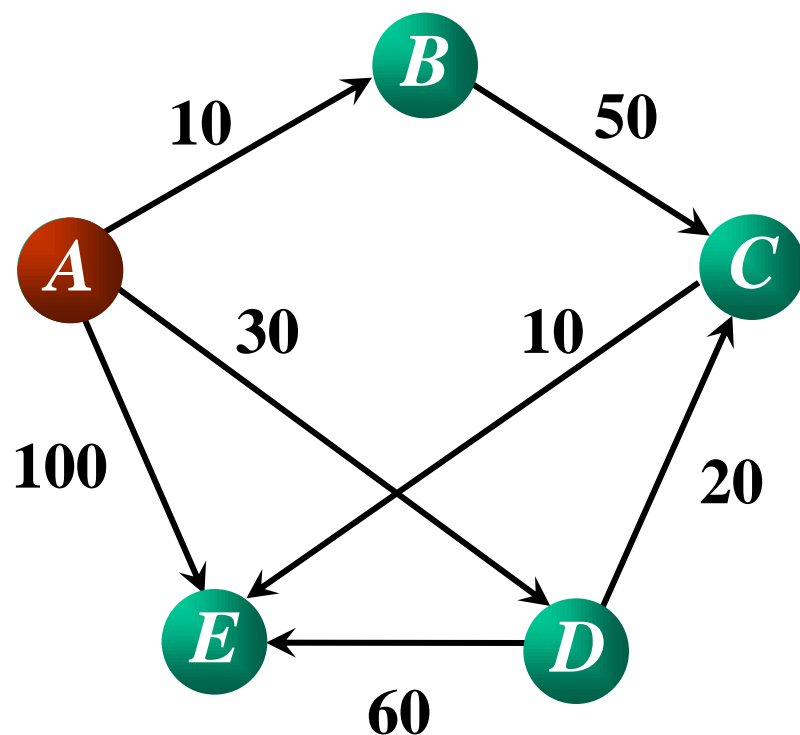


设置一个集合 S 存放已经找到最短路径的顶点， S 的初始状态只包含源点 v ，对 $v_i \in V-S$ ，**假设从源点 v 到 v_i 的有向边为最短路径。**

以后每求得一条最短路径 v, \dots, v_k ，就将 v_k 加入集合 S 中，并将路径 v, \dots, v_k, v_i 与原来的假设相比较，取路径长度较小者为最短路径。

重复上述过程，直到集合 V 中全部顶点加入到集合 S 中。

Dijkstra算法示例:



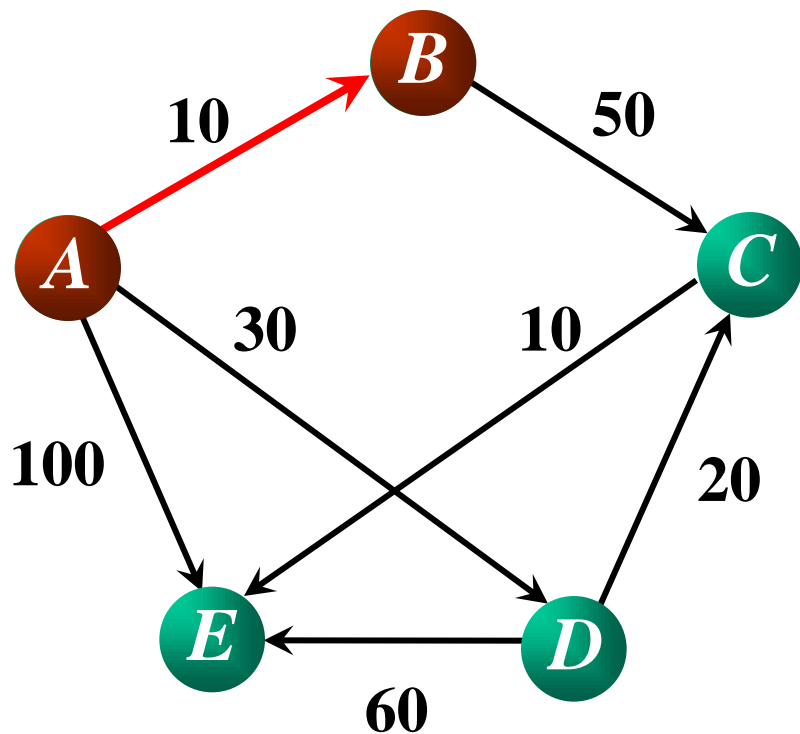
$S=\{A\}$

$A \rightarrow B: (A, B)10$

$A \rightarrow C: (A, C)\infty$

$A \rightarrow D: (A, D)30$

$A \rightarrow E: (A, E)100$



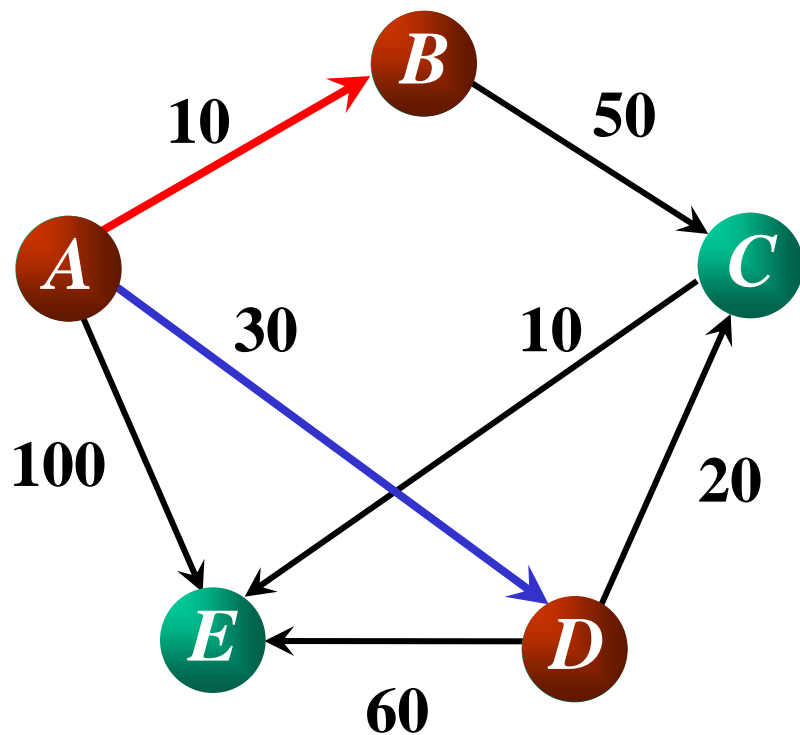
$S=\{A, B\}$

$A \rightarrow B: (A, B) 10$

$A \rightarrow C: (A, B, C) 60$

$A \rightarrow D: (A, D) 30$

$A \rightarrow E: (A, E) 100$



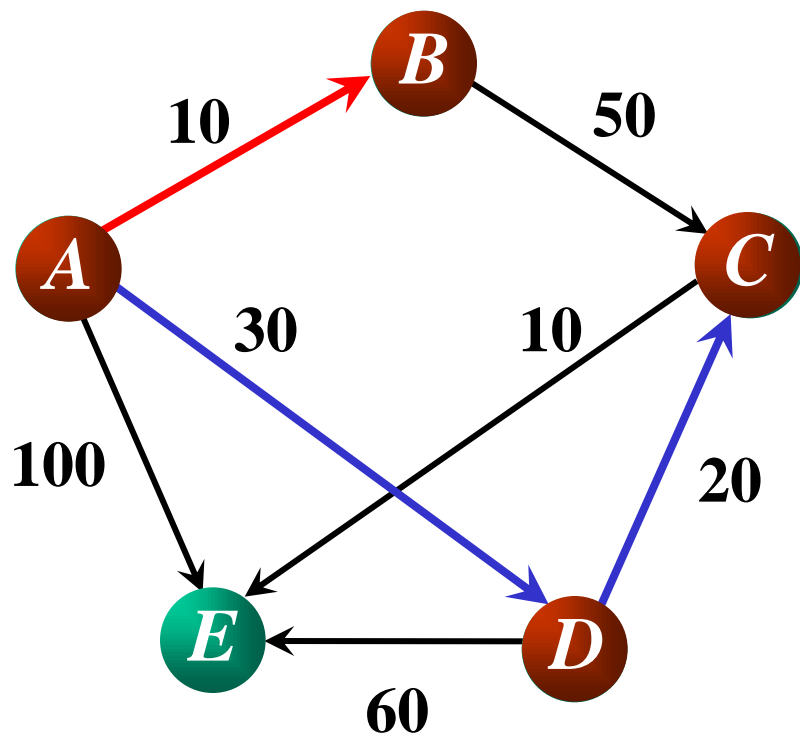
$S=\{A, B, D\}$

$A \rightarrow B: (A, B) 10$

$A \rightarrow C: (A, D, C) 50$

$A \rightarrow D: (A, D) 30$

$A \rightarrow E: (A, D, E) 90$



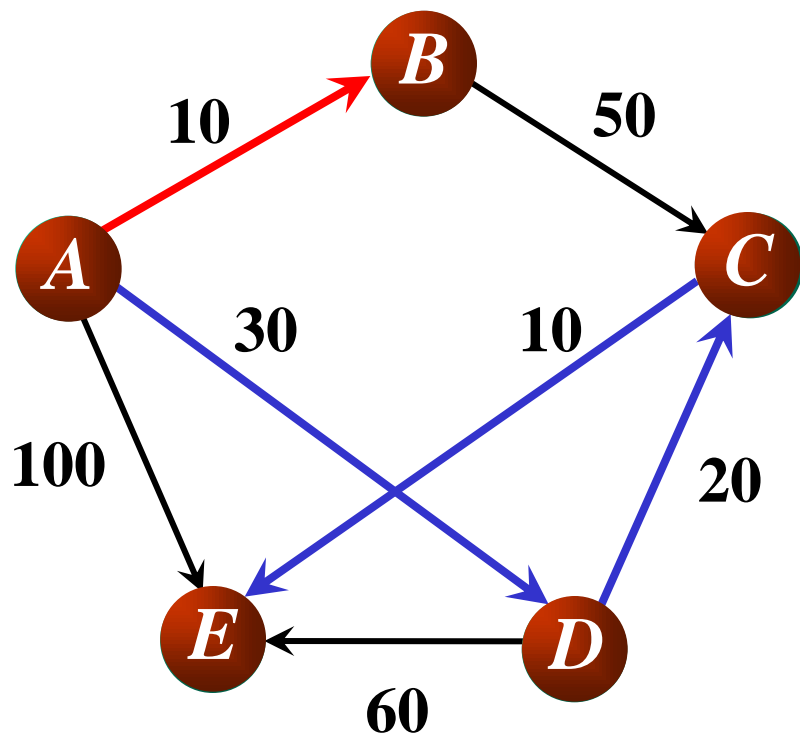
$S=\{A, B, D, C\}$

$A \rightarrow B: (A, B) 10$

$A \rightarrow C: (A, D, C) 50$

$A \rightarrow D: (A, D) 30$

$A \rightarrow E: (A, D, C, E) 60$



$S=\{A, B, D, C, E\}$

$A \rightarrow B: (A, B) 10$

$A \rightarrow C: (A, D, C) 50$

$A \rightarrow D: (A, D) 30$

$A \rightarrow E: (A, D, C, E) 60$

相应的数据结构设计：

图的存储结构：带权的邻接矩阵存储结构

数组dist[n]：每个分量dist[i]表示当前所找到的从始点 v 到终点 v_i 的最短路径的长度。初态为：若从 v 到 v_i 有弧，则dist[i]为弧上权值；否则置dist[i]为 ∞ 。

数组path[n]：path[i]是一个字符串，表示当前所找到的从始点 v 到终点 v_i 的最短路径。初态为：若从 v 到 v_i 有弧，则path[i]为 vv_i ；否则置path[i]空串。

数组s[n]：存放源点和已经生成的终点，其初态为只有一个源点 v 。

Dijkstra算法——伪代码描述:

1. 初始化数组dist、path和s;
2. while (s中的元素个数<n)
 - 2.1 在dist[n]中求最小值, 其下标为k;
 - 2.2 输出dist[k]和path[k];
 - 2.3 修改数组dist和path;
 - 2.4 将顶点 v_k 添加到数组s中。


```
void Dijkstra(MGraph G, int v)
{
    for (i=0; i<G.vertexNum; i++)
        //初始化dist[n]、 path[n]
        {
            dist[i]=G.arc[v][i];
            if (dist[i]!=∞) path[i]=G.vertex[v]+G.vertex[i];
            else path[i]="";
        }
    s[0]=G.vertex[v];    //初始化集合S
    dist[v]=0;           //标记顶点v为源点
    num=1;
}
```

```
while (num<G.vertexNum)
```

```
    //当数组s中的顶点数小于图的顶点数时循环
```

```
{ k=0;
```

```
    for (i=1; i<G.vertexNum; i++) //在dist中找最小元素
```

```
        if ((dist[i]<dist[k]) && dist[i]!=0 ) k=i;
```

```
    cout<<dist[k]<<path[k];
```

```
    s[num++]= G.vertex[k]; //将新生成的终点加入集合S
```

```
    dist[k]=0;    //置顶点vk为已生成终点标记
```

```
    for (i=0; i<G.vertexNum; i++) //修改数组dist和path
```

```
        if (dist[i]>dist[k]+G.arc[k][i]) {
```

```
            dist[i]=dist[k]+G.arc[k][i];
```

```
            path[i]=path[k]+G.vertex[i];
```

```
        }
```

```
    }//while
```

```
}//end
```

```
while (num<G.vertexNum)
```

```
    //当数组s中的顶点数小于图的顶点数时循环
```

```
{ k=0;
```

```
  for (i=1; i<G.vertexNum; i++) //在dist中找最小元素
```

```
    if ((dist[i]<dist[k]) && dist[i]!=0 ) k=i;
```

```
  cout<<dist[k]<<path[k];
```

```
  s[num++]= G.vertex[k]; //将新生成的终点加入集合S
```

```
  for (i=0; i<G.vertexNum; i++) //修改数组dist和path
```

```
    if (dist[i]>dist[k]+G.arc[k][i]) {
```

```
      dist[i]=dist[k]+G.arc[k][i];
```

```
      path[i]=path[k]+G.vertex[i];
```

```
    }
```

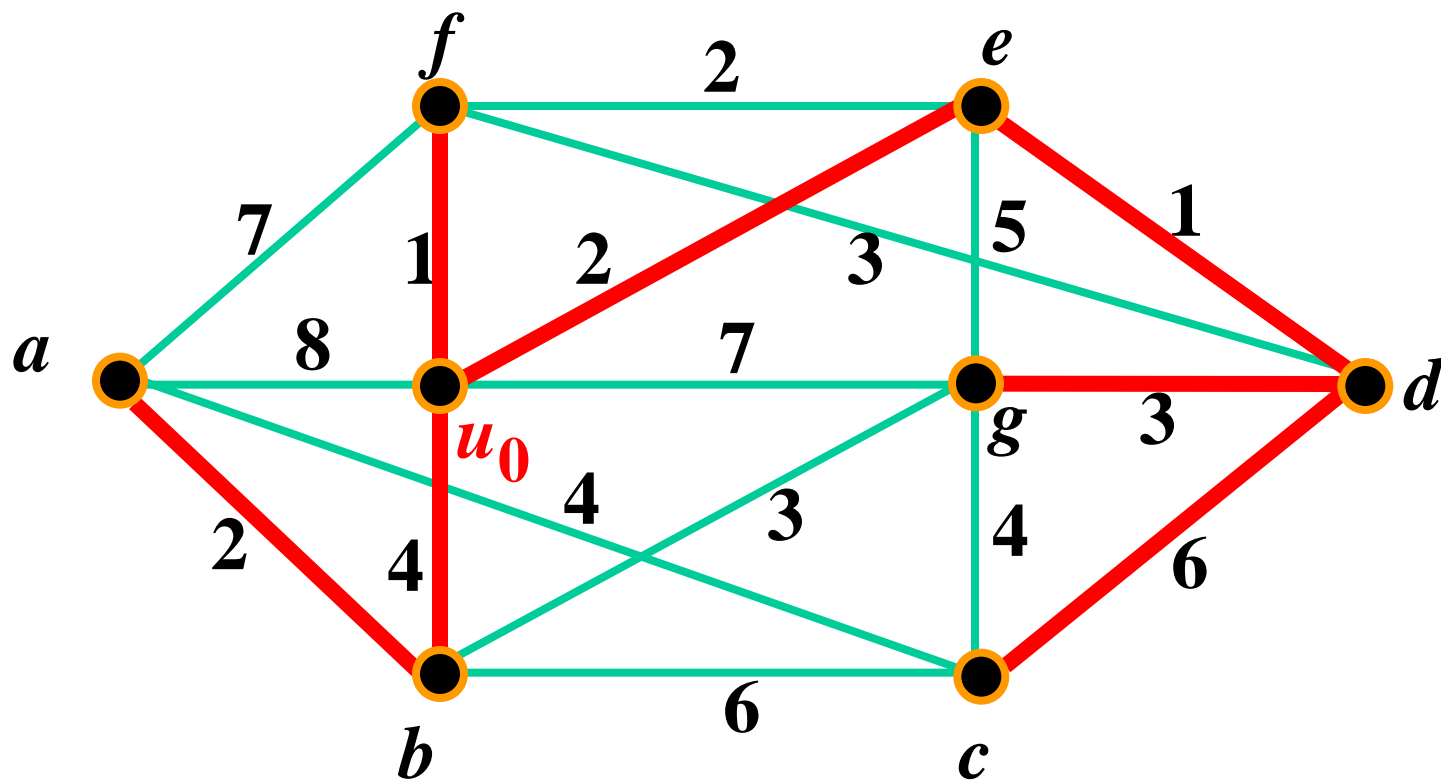
```
    dist[k]=0;      //置顶点vk为已生成终点标记
```

```
  }//while
```

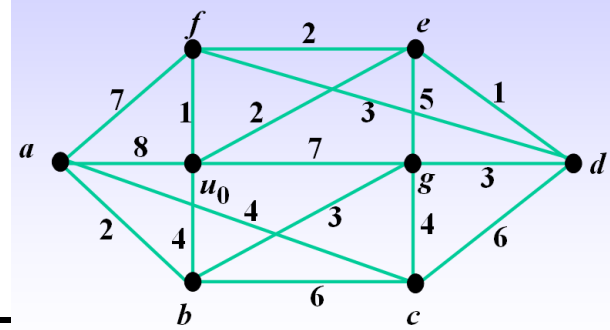
```
}//end
```

课堂练习

求 u_0 到其他各顶点的最短路径。



课堂练习



	S	T	l(a)	l(b)	l(c)	l(d)	l(e)	l(f)	l(g)
1	u_0	abcdefg	8	4	∞	∞	2	1	7
2	$u_0 f$	abcdeg	8	4	∞	4	2		7
3	$u_0 fe$	abcdg	8	4	∞	3			7
4	$u_0 fed$	abcg	8	4	9				6
5	$u_0 fedb$	acg	6		9				6
6	$u_0 fedba$	cg			9				6
7	$u_0 fedbag$	c			9				
8	$u_0 fedbagc$								

2、每一对顶点之间的最短路径

问题描述：给定带权有向图 $G=(V, E)$ ，对任意顶点 $v_i, v_j \in V (i \neq j)$ ，求顶点 v_i 到顶点 v_j 的最短路径。

解决办法1：每次以一个顶点为源点，调用Dijkstra算法 n 次。显然，时间复杂度为 $O(n^3)$ 。

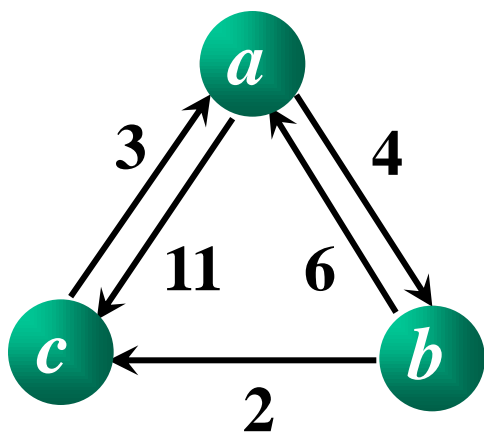
解决办法2：弗洛伊德提出的求每一对顶点之间的最短路径算法——Floyd算法，其时间复杂度也是 $O(n^3)$ ，但形式上要简单些。

Floyd算法的基本思想:

对于从 v_i 到 v_j 的弧, 进行 n 次试探: 首先考虑路径 v_i, v_0, v_j 是否存在, 如果存在, 则比较 v_i, v_j 和 v_i, v_0, v_j 的路径长度, 取较短者为从 v_i 到 v_j 的中间顶点的序号不大于0的最短路径。

在路径上再增加一个顶点 v_1 , 依此类推, 在经过 n 次比较后, 最后求得的必是从顶点 v_i 到顶点 v_j 的最短路径。

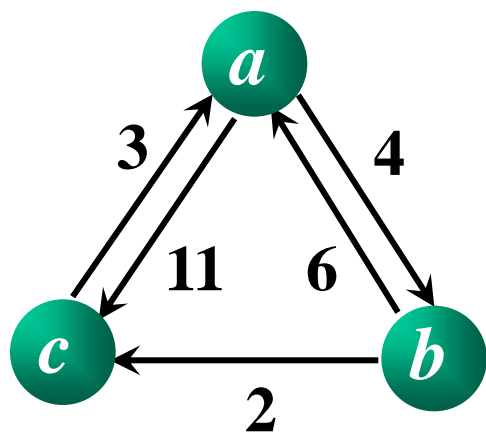
Floyd算法示例:



有向网图

$$\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

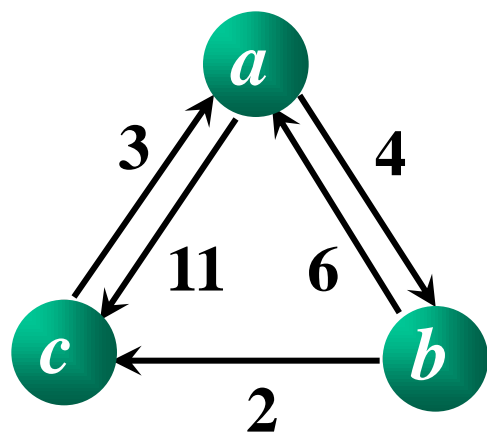
邻接矩阵



初始化

$$\text{dist}_{-1} = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{pmatrix}$$

$$\text{path}_{-1} = \begin{pmatrix} & ab & ac \\ ba & & \\ ca & & \end{pmatrix}$$



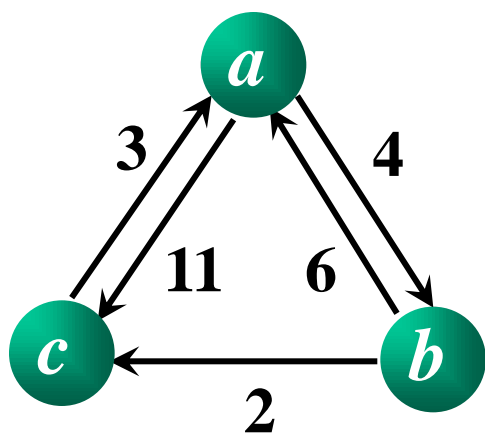
第1次迭代

$$\text{dist}_{-1} = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{pmatrix}$$

$$\text{path}_{-1} = \begin{pmatrix} & ab & ac \\ ba & & \\ ca & & \end{pmatrix}$$

$$\text{dist}_0 = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$$

$$\text{path}_0 = \begin{pmatrix} & ab & ac \\ ba & & bc \\ ca & cab & \end{pmatrix}$$



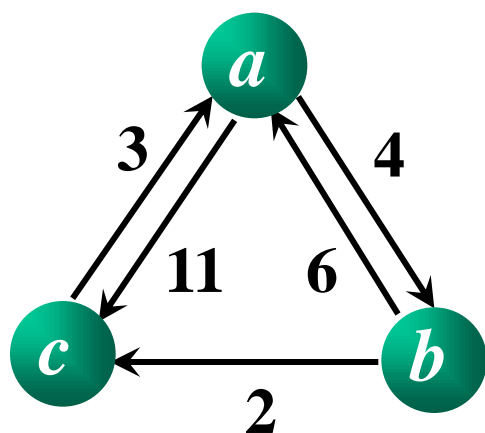
第2次迭代

$$\text{dist}_0 = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$$

$$\text{dist}_1 = \begin{pmatrix} 0 & 4 & \mathbf{6} \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$$

$$\text{path}_0 = \begin{pmatrix} & ab & ac \\ ba & & bc \\ ca & cab & \end{pmatrix}$$

$$\text{path}_1 = \begin{pmatrix} & ab & \mathbf{abc} \\ ba & & bc \\ ca & cab & \end{pmatrix}$$



第3次迭代

$$\text{dist}_1 = \begin{pmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$$

$$\text{dist}_2 = \begin{pmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$$

$$\text{path}_1 = \begin{pmatrix} & ab & abc \\ ba & & bc \\ ca & cab & \end{pmatrix}$$

$$\text{path}_2 = \begin{pmatrix} & ab & abc \\ bca & & bc \\ ca & cab & \end{pmatrix}$$

相应的数据结构设计:

图的存储结构: 带权的邻接矩阵存储结构

数组 $\text{dist}[n][n]$: 存放在迭代过程中求得的最短路径长度。

迭代公式:

$$\begin{cases} \text{dist}_1[i][j] = \text{arc}[i][j] \\ \text{dist}_k[i][j] = \min\{\text{dist}_{k-1}[i][j], \text{dist}_{k-1}[i][k] + \text{dist}_{k-1}[k][j]\} \\ 0 \leq k \leq n-1 \end{cases}$$

数组 $\text{path}[n][n]$: 存放从 v_i 到 v_j 的最短路径, 初始为 $\text{path}[i][j] = "v_i v_j"$ 。

Floyd算法:

```
void Floyd (MGraph G)  
{  
    for (i=0; i<G.vertexNum; i++)  
        for (j=0; j<G.vertexNum; j++)  
            {  
                dist[i][j]=G.arc[i][j];  
                if (dist[i][j]!=∞)  
                    path[i][j]=G.vertex[i]+G.vertex[j];  
                else path[i][j]='';  
            }
```

```
for (k=0; k<G.vertexNum; k++) //进行n次试探
```

```
for (i=0; i<G.vertexNum; i++)
```

```
for (j=0; j<G.vertexNum; j++)
```

```
if (dist[i][k]+dist[k][j]<dist[i][j]) {
```

```
    dist[i][j]=dist[i][k]+dist[k][j];
```

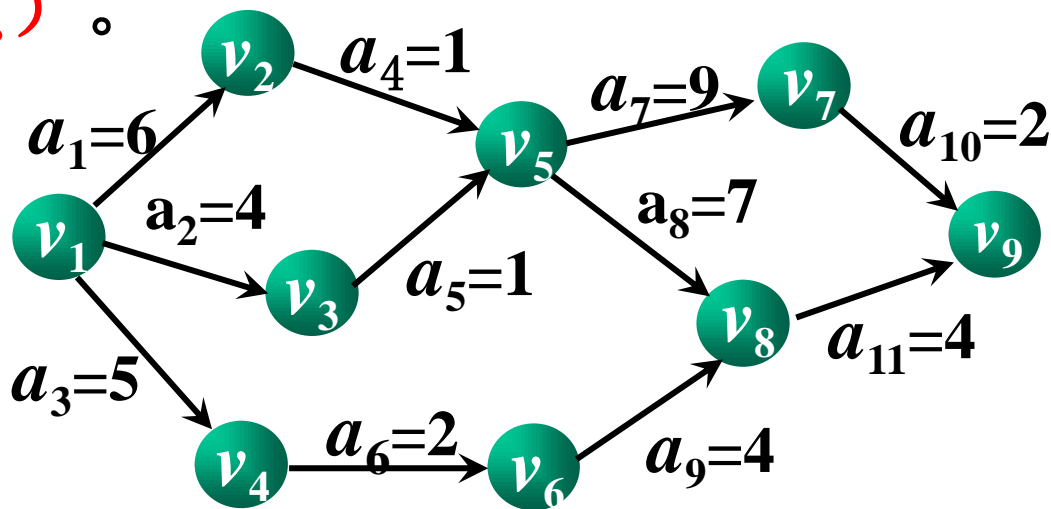
```
    path[i][j]=path[i][k]+path[k][j];
```

```
}
```

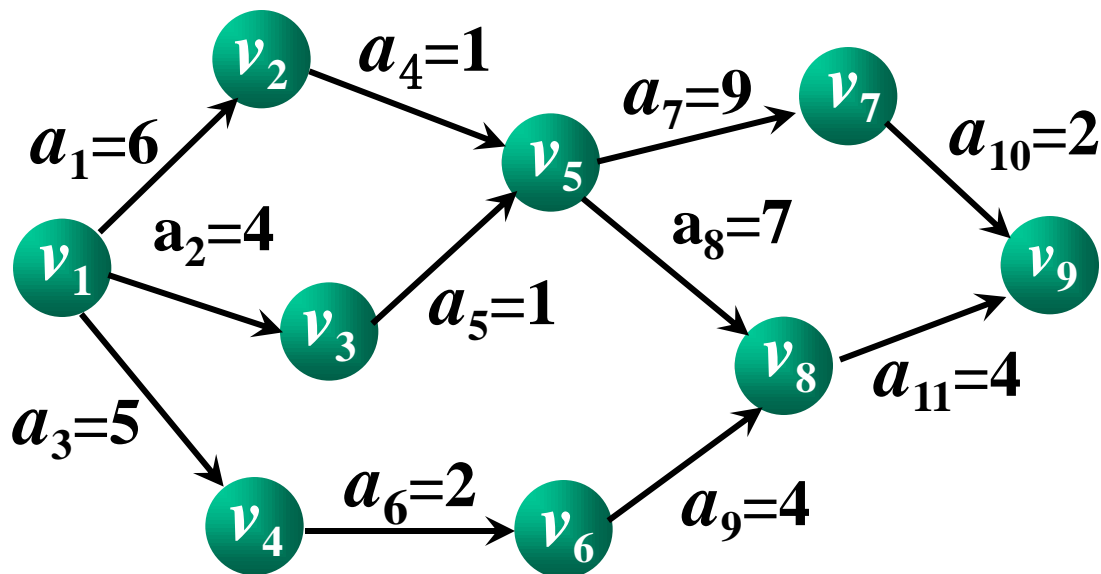
```
}//end
```

6.5 图的应用举例：（4）关键路径

AOE网：在一个表示工程的带权有向图中，用顶点表示事件，用有向边表示活动，边上的权值表示活动的持续时间，称这样的有向图叫做**边表示活动的网（activity on edge network）**，简称**AOE网**。AOE网中没有入边的顶点称为**始点（或源点）**，没有出边的顶点称为**终点（或汇点）**。



AOE网:



事件	事件含义
v_1	开工
v_2	活动 a_1 完成, 活动 a_4 可以开始
v_3	活动 a_2 完成, 活动 a_5 可以开始
...
v_9	活动 a_{10} 和 a_{11} 完成, 整个工程完成

AOE网的性质:

- (1) 只有在某顶点所代表的事件发生后，从该顶点出发的各活动才能开始；
- (2) 只有在进入某顶点的各活动都结束，该顶点所代表的事件才能发生。

AOE网可以回答下列问题：

1. 完成整个工程至少需要多少时间？
2. 为缩短完成工程所需的时间, 应当加快哪些活动？

从始点到终点的路径可能不止一条，只有各条路径上所有活动都完成了，整个工程才算完成。

因此，完成整个工程所需的最短时间取决于从始点到终点的**最长路径长度**，

即这条路径上所有活动的持续时间之和。这条路径长度最长的路径就叫做**关键路径(critical path)**。

关键路径：在AOE网中，从始点到终点具有**最大路径长度**（该路径上的各个活动所持续的时间之和）的路径称为关键路径。

关键活动：关键路径上的活动称为关键活动。

由于AOE网中的某些活动能够同时进行，故完成整个工程所必须花费的时间应该为始点到终点的**最大路径长度**。关键路径长度是整个工程所需的**最短工期**。

要找出关键路径，必须找出关键活动，即不按期完成就会影响整个工程完成的活动。

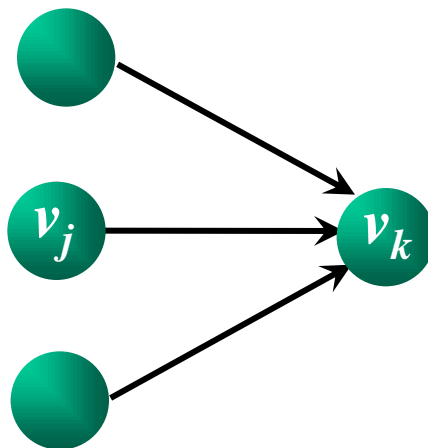
首先计算以下与关键活动有关的量：

- (1) 事件的最早发生时间 $ve[k]$
- (2) 事件的最迟发生时间 $vl[k]$
- (3) 活动的最早开始时间 $e[i]$
- (4) 活动的最晚开始时间 $l[i]$

最后计算各个活动的时间余量 $l[k] - e[k]$ ，时间余量为0者即为关键活动。

(1) 事件的最早发生时间 $ve[k]$

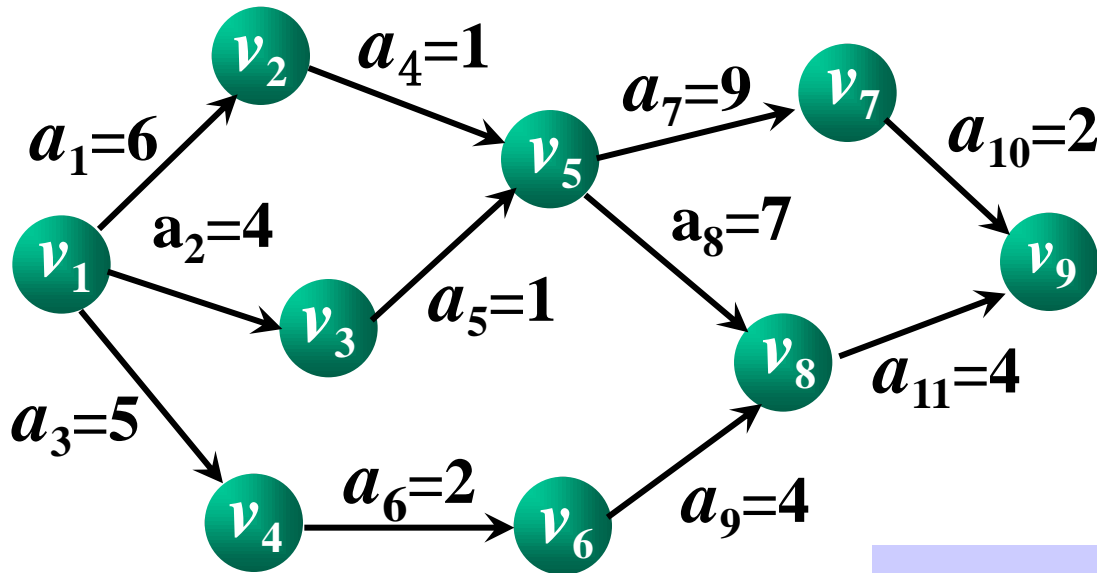
$ve[k]$ 是指从始点开始到顶点 v_k 的最大路径长度。这个长度决定了所有从顶点 v_k 发出的活动能够开工的最早时间。



$$\begin{cases} ve[1]=0 \end{cases}$$

$$\begin{cases} ve[k]=\max\{ve[j]+\text{len}\langle v_j, v_k\rangle\} \quad (\langle v_j, v_k\rangle \in p[k]) \end{cases}$$

$p[k]$ 表示所有到达 v_k 的有向边的集合

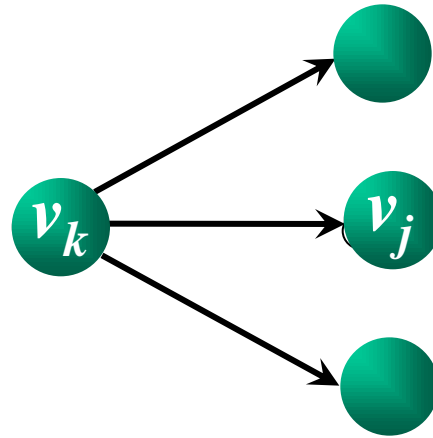


$$ve[k] = \max\{ve[j] + \text{len}\langle v_j, v_k \rangle\}$$

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
ve[k]	0	6	4	5	7	7	16	14	18

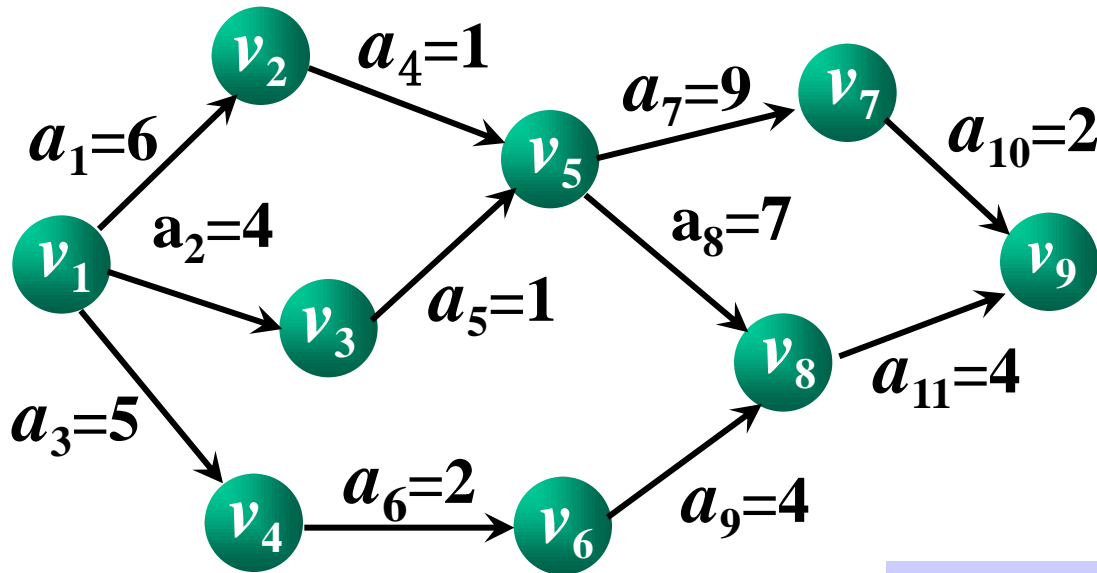
(2) 事件的最迟发生时间 $vl[k]$

$vl[k]$ 是指在不推迟整个工期的前提下,事件 v_k 允许的最晚发生时间。



$$\begin{cases} vl[n]=ve[n] \\ vl[k]=\min\{vl[j]-len\langle v_k, v_j \rangle\} \quad (\langle v_k, v_j \rangle \in s[k]) \end{cases}$$

$s[k]$ 为所有从 v_k 发出的有向边的集合



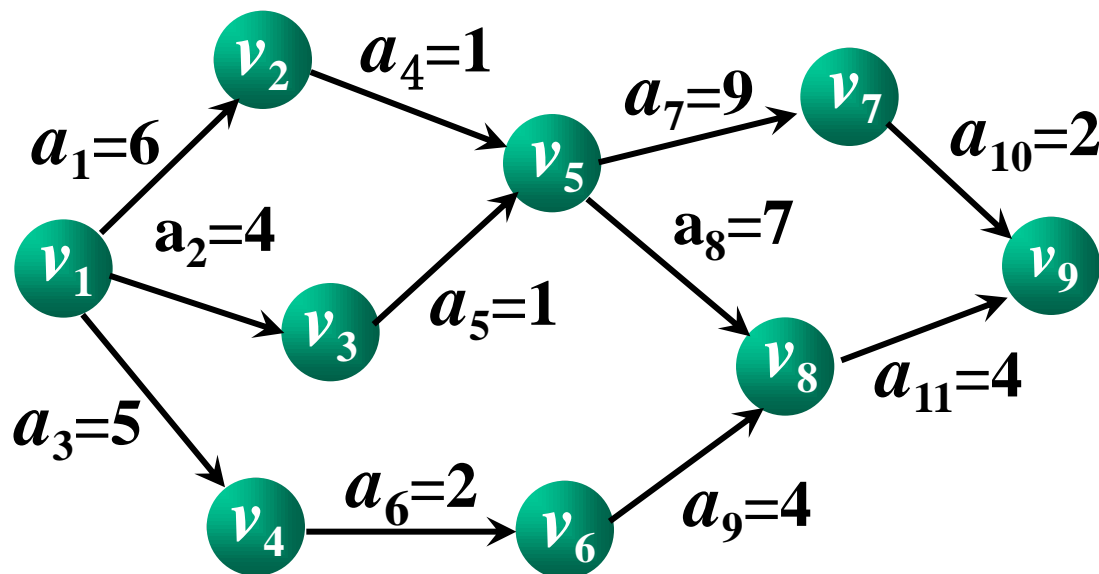
$$vl[k] = \min\{vl[j] - \text{len}\langle v_k, v_j \rangle\}$$

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
ve[k]	0	6	4	5	7	7	16	14	18
vl[k]	0	6	6	8	7	10	16	14	18

(3) 活动的最早开始时间 $e[i]$

若活动 a_i 是由弧 $\langle v_k, v_j \rangle$ 表示，则活动 a_i 的最早开始时间应等于事件 v_k 的最早发生时间。因此，有：

$$e[i] = ve[k]$$



$$e[i] = ve[k]$$

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
ve[k]	0	6	4	5	7	7	16	14	18

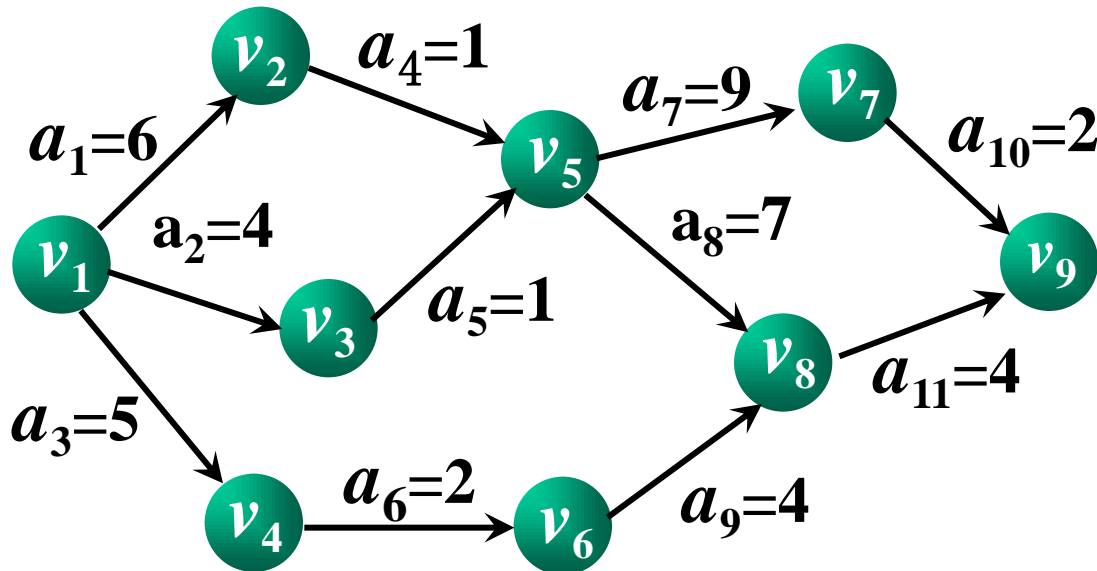
	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
e[i]	0	0	0	6	4	5	7	7	7	16	14

(4) 活动的最晚开始时间 $l[i]$

活动 a_i 的最晚开始时间是指，在不推迟整个工期的前提下， a_i 必须开始的最晚时间。

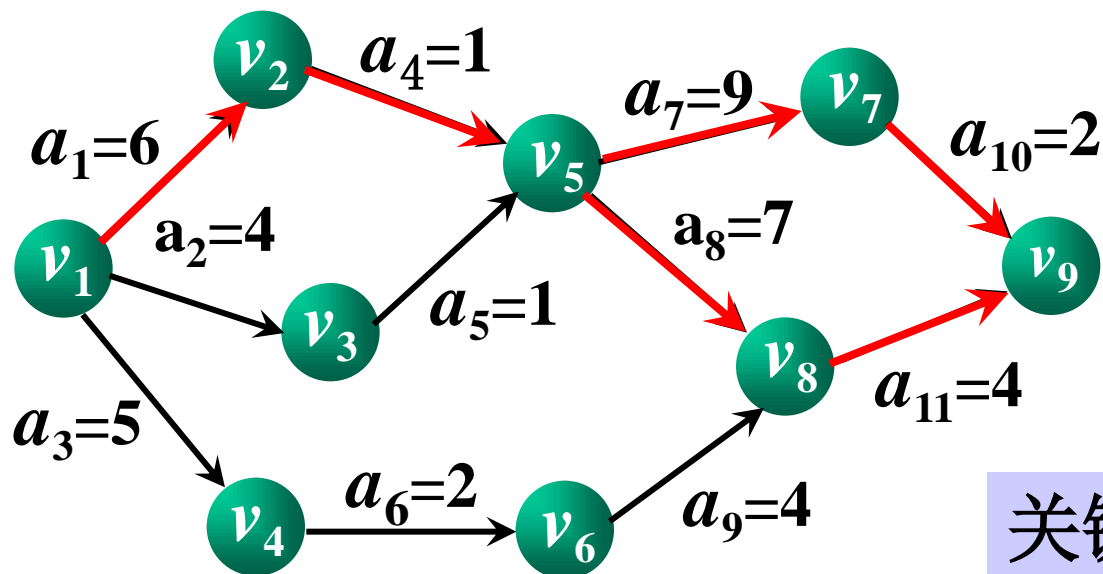
若 a_i 由弧 $\langle v_k, v_j \rangle$ 表示，则 a_i 的最晚开始时间要保证事件 v_j 的最迟发生时间不拖后。因此，有：

$$l[i] = vl[j] - \text{len}\langle v_k, v_j \rangle$$



$$l[i] = vl[j] - \text{len}\langle v_k, v_j \rangle$$

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9		
$vl[k]$	0	6	6	8	7	10	16	14	18		
	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
$l[i]$	0	2	3	6	6	8	7	7	10	16	14



关键活动： $l[i]=e[i]$ 的活动

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
$e[i]$	0	0	0	6	4	5	7	7	7	16	14
$l[i]$	0	2	3	6	6	8	7	7	10	16	14

本章作业

习题6 (P184) : $4(1)(2)(4)(5)(6), 5(3)(6)$