

第六章 图

本章的主要内容是：

图的逻辑结构

图的存储结构及实现

图的连通性

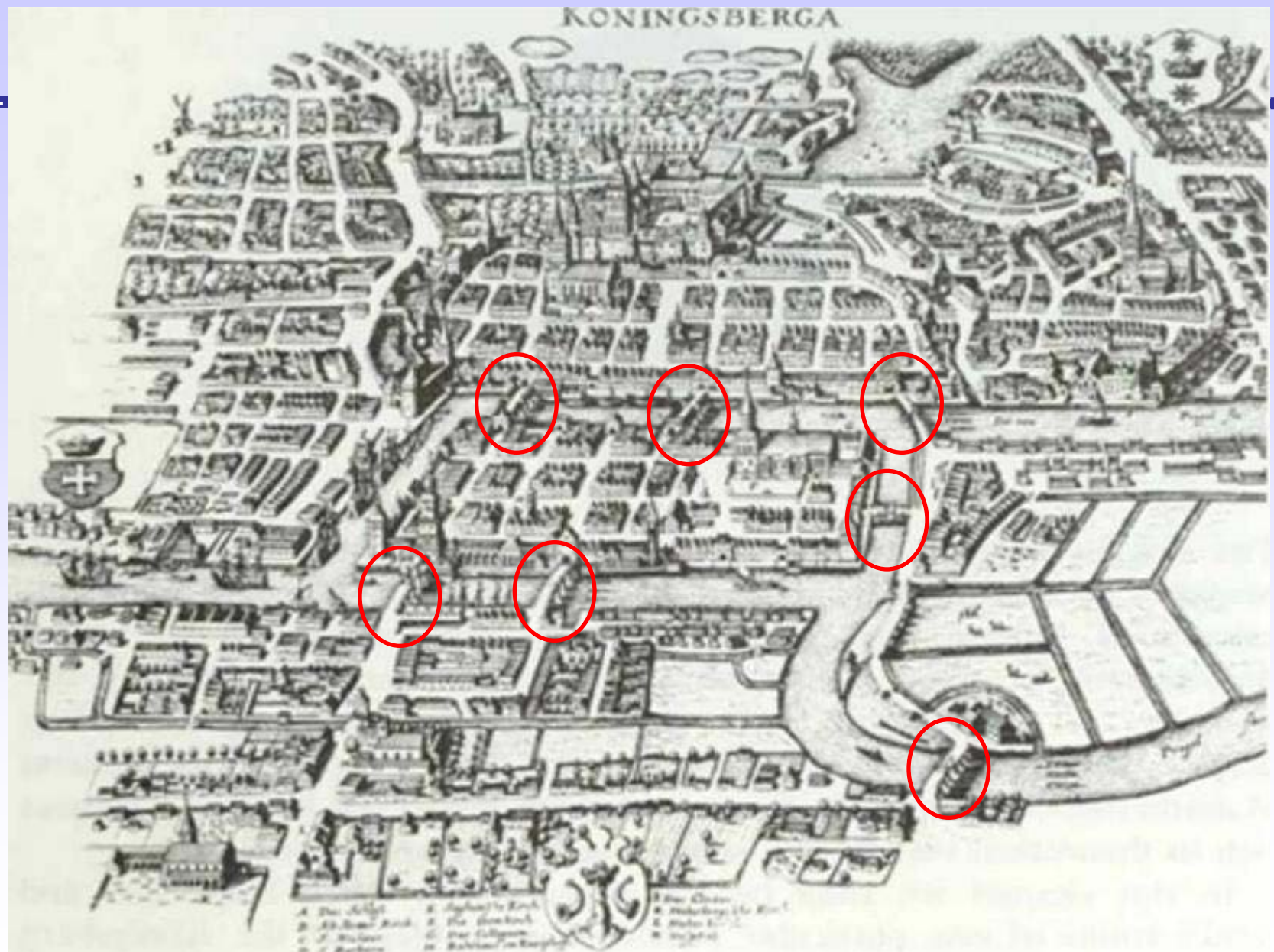
最小生成树

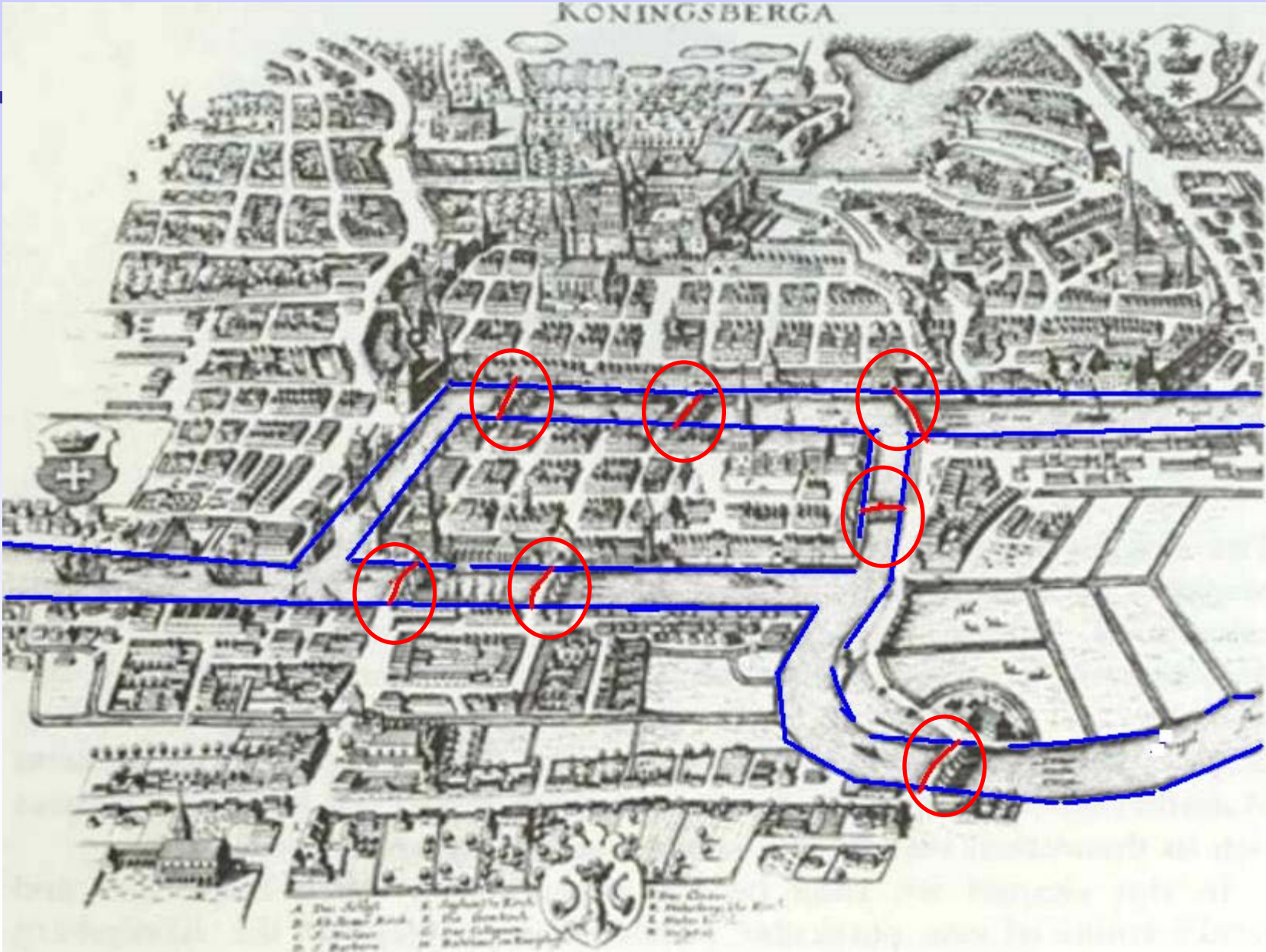
最短路径

AOV网与拓扑排序

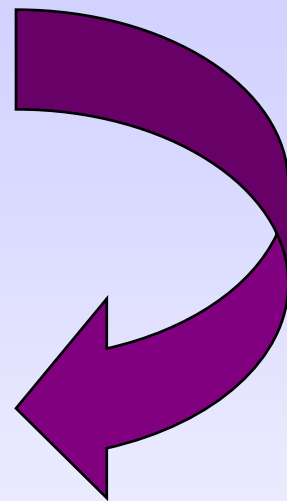
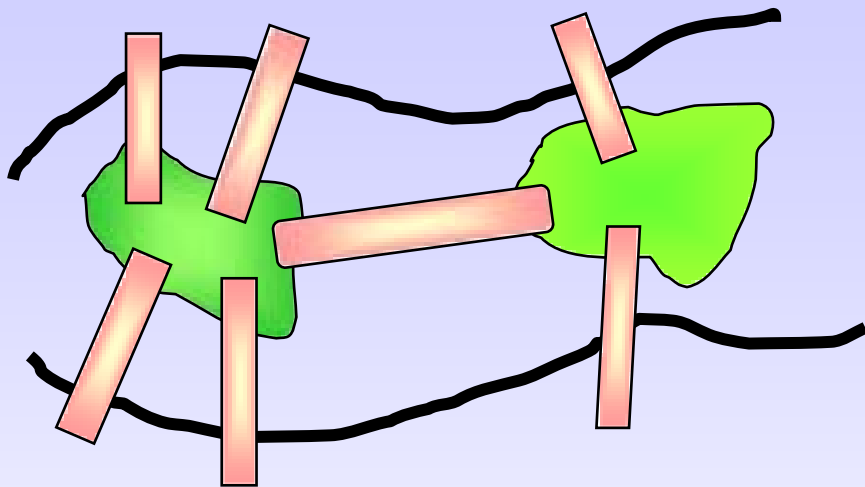
AOE网与关键路径

-
- 哥尼斯堡七桥问题 **Seven Bridges Problem**
——18世纪著名古典数学问题之一
 - Königsberg, Prussia(现 Kaliningrad加里宁格勒 Russia)





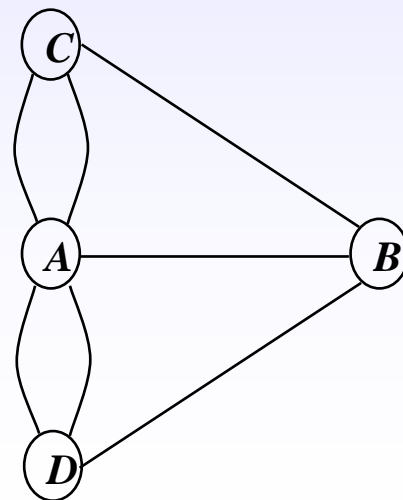
哥尼斯堡七桥问题



哥尼斯堡七桥问题：如何不重复地走完七桥后回到起点？

一笔画问题

如何将此图一笔画出？



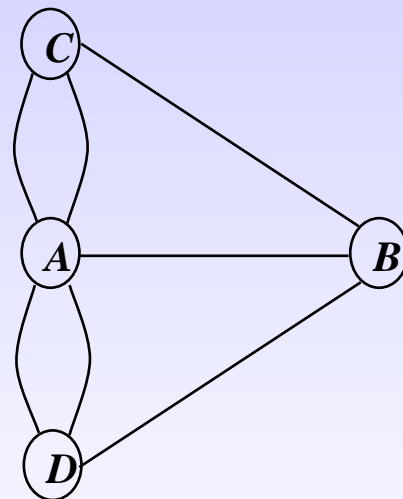
欧拉这种处理问题的方法标志着图论的诞生

欧拉的推理

凡是一笔画中出现的交点处，线一出一进总应该通过偶数条（偶点），只有作为起点和终点的两点才有可能通过奇数条（奇点）。

结论：凡是奇点个数多于两个的平面图都不能一笔画出。

七桥问题中有四个奇点，故不能一笔画出



图论——欧拉



欧拉 (L. Euler, 1707. 4. 15– 1783. 9. 18) 著名的数学家。生于瑞士的巴塞尔，卒于彼得堡。大部分时间在俄国和德国度过。他早年在数学天才贝努里赏识下开始学习数学，17岁获得硕士学位，毕业后研究数学，是数学史上最高产的作家。在世时发表论文700多篇，去世后还留下100多篇待发表。其论著几乎涉及**所有数学分支**。

欧拉在数学、物理、天文、建筑以至音乐、哲学方面都取得了辉煌的成就。

6.1 图的逻辑结构

图的定义

图是由顶点的有穷非空集合和顶点之间边的集合组成，通常表示为：

$$\underline{G=(V, E)} \quad (\text{Graph Vertex Edge})$$

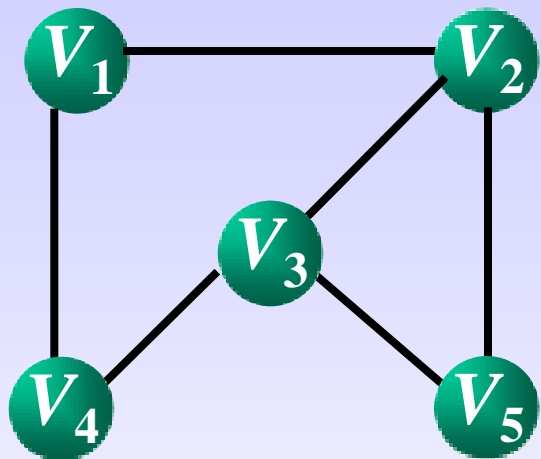
其中： G 表示一个图， V 是图 G 中顶点的集合， E 是图 G 中顶点之间边的集合。

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；

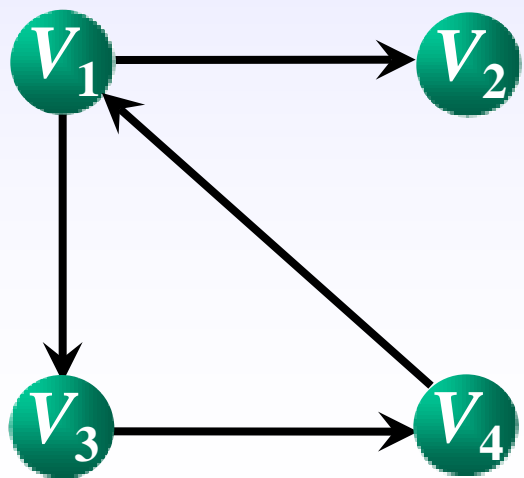
在图中，顶点个数不能为零，但可以没有边。

6.1 图的逻辑结构



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**。



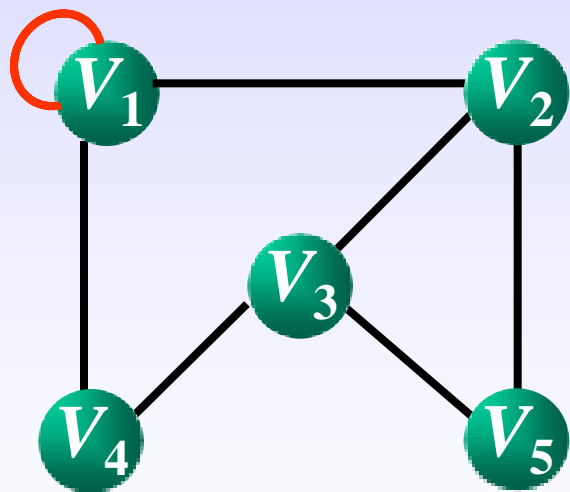
若从顶点 v_i 到 v_j 的边有方向，则称这条边为**有向边**，表示为 $\langle v_i, v_j \rangle$ 。

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**。

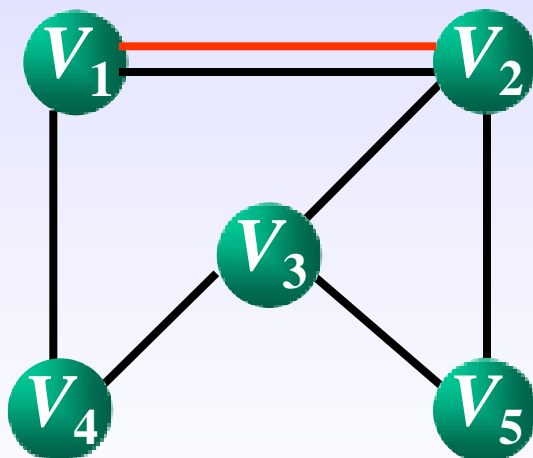
6.1 图的逻辑结构

图的基本术语

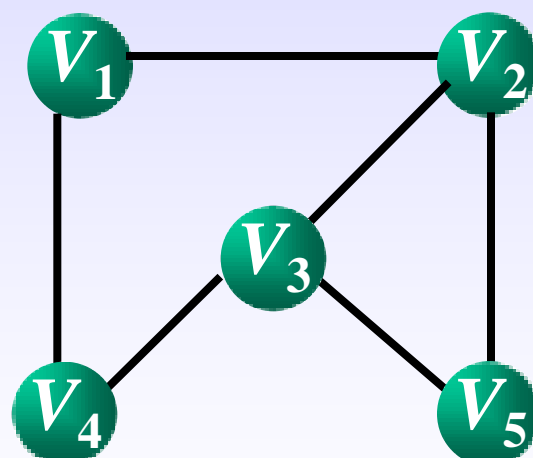
简单图：在图中，若不存在顶点到其自身的边，且同一条边不重复出现。



非简单图



非简单图



简单图

❖ 数据结构中讨论的都是简单图。

6.1 图的逻辑结构

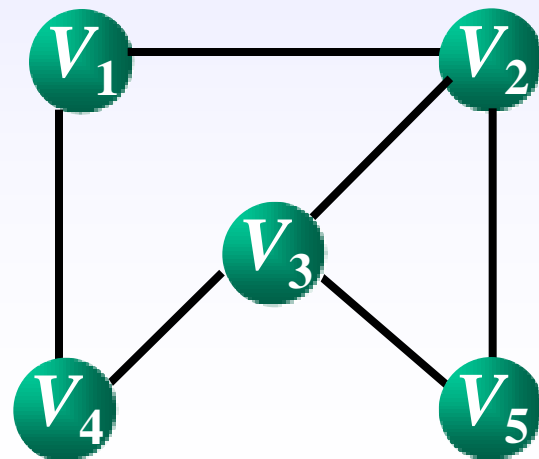
图的基本术语

邻接、依附

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

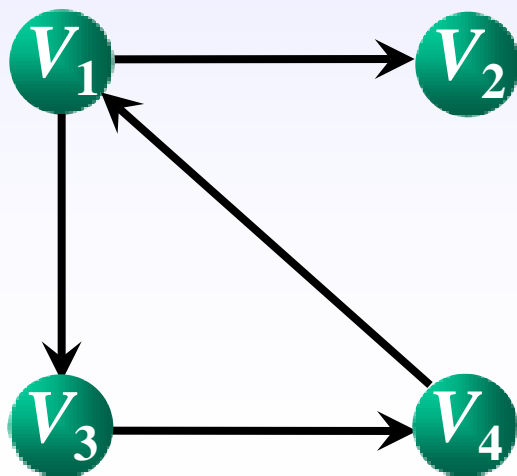


6.1 图的逻辑结构

图的基本术语

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 。



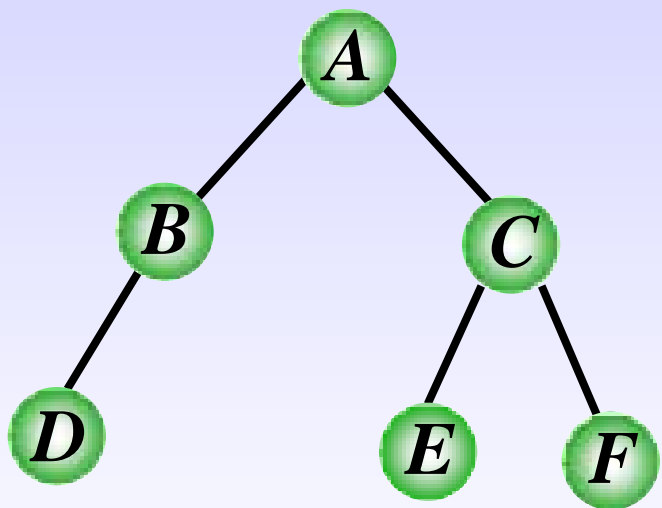
V_1 的邻接点: V_2 、 V_3

V_3 的邻接点: V_4

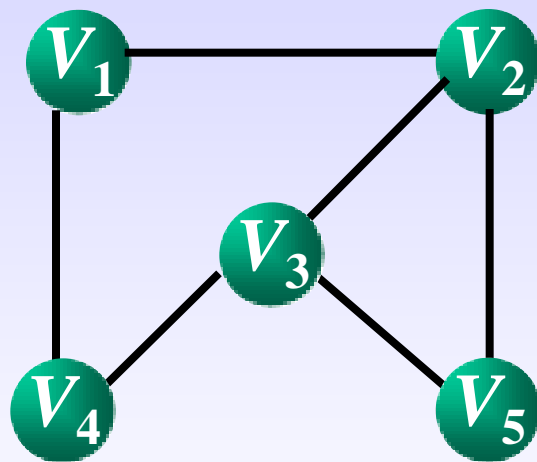
不同结构中逻辑关系的对比



线性结构



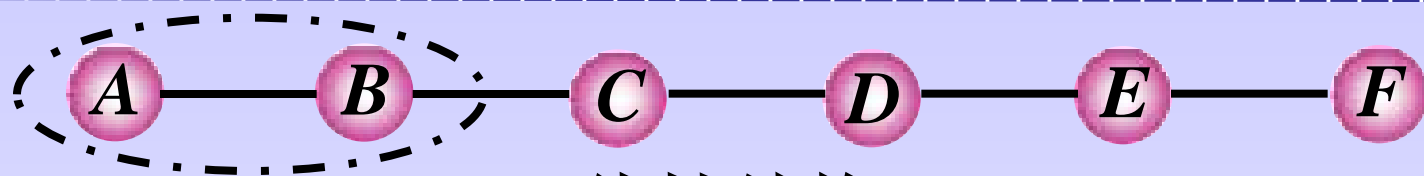
树结构



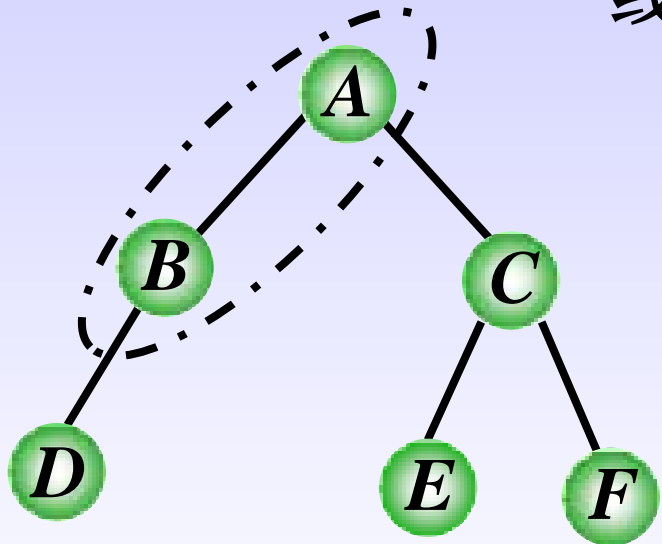
图结构

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。

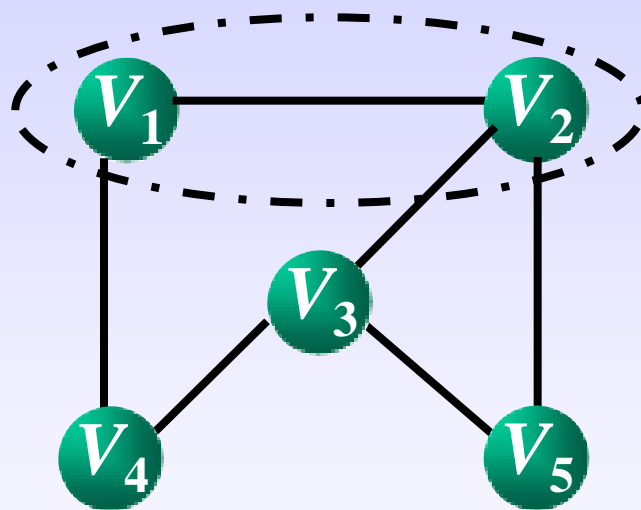
不同结构中逻辑关系的对比



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**。

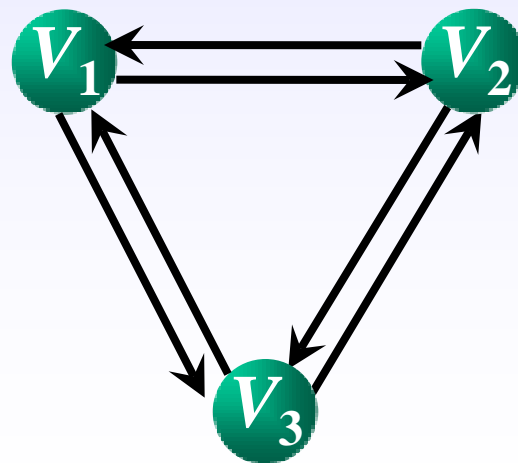
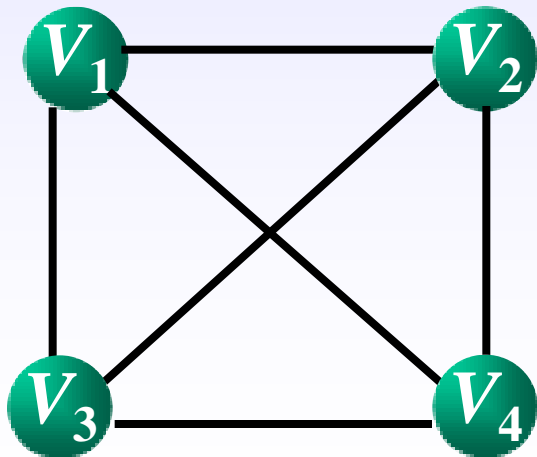
6.1 图的逻辑结构

图的基本术语

无向完全图：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：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。

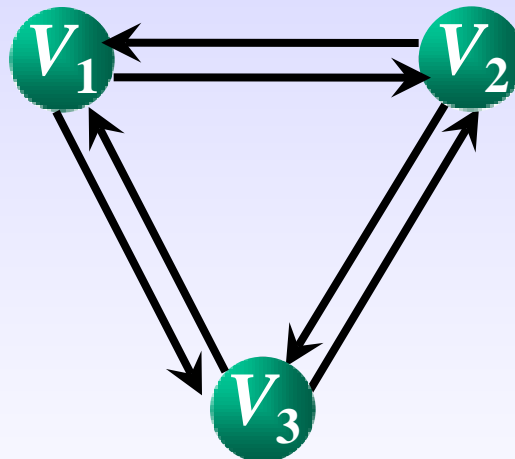
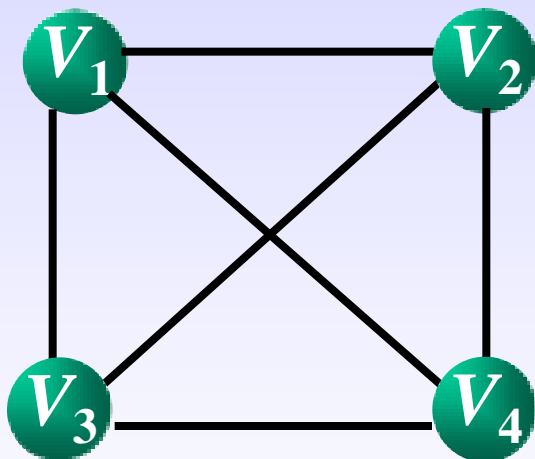
。



6.1 图的逻辑结构



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。

6.1 图的逻辑结构

图的基本术语

稀疏图：称边数很少的图为稀疏图；

稠密图：称边数很多的图为稠密图。

顶点的度：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 $TD(v)$ 。

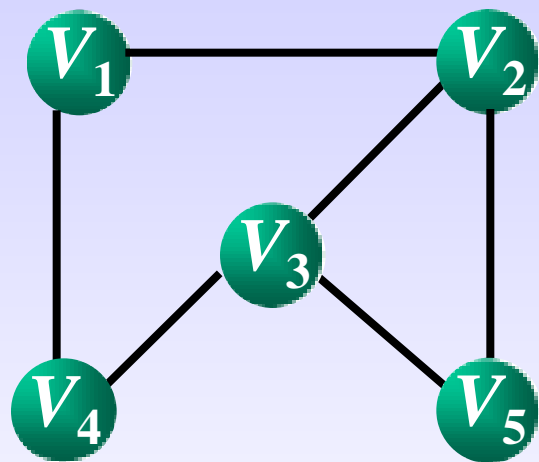
顶点的入度：在**有向图**中，顶点 v 的**入度**是指以该顶点为弧头的弧的数目，记为 $ID(v)$ ；

顶点的出度：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 $OD(v)$ 。

6.1 图的逻辑结构

图的基本术语

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

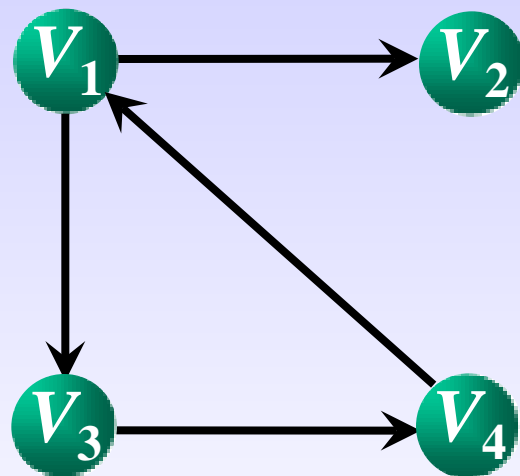


在具有 n 个顶点、 e 条边的无向图 G 中，各顶点的度之和与边数之和的关系？

6.1 图的逻辑结构

图的基本术语

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



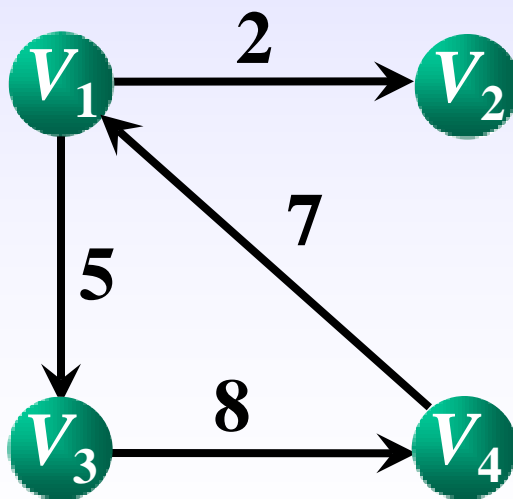
在具有 n 个顶点、 e 条边的有向图 G 中，各顶点的入度之和与各顶点的出度之和的关系？与边数之和的关系？

6.1 图的逻辑结构

图的基本术语

权：是指对边赋予的有意义的数值量。

网：边上带权的图，也称网图。



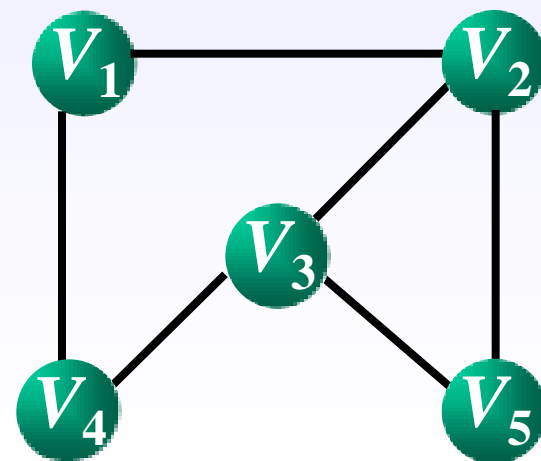
6.1 图的逻辑结构

图的基本术语

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：
 $V_1 V_4$
 $V_1 V_2 V_3 V_4$
 $V_1 V_2 V_5 V_3 V_4$

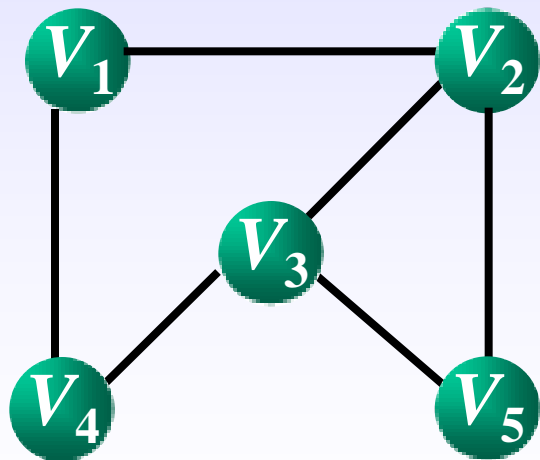
❖ 一般情况下，图中的路径不惟一。



6.1 图的逻辑结构

图的基本术语

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

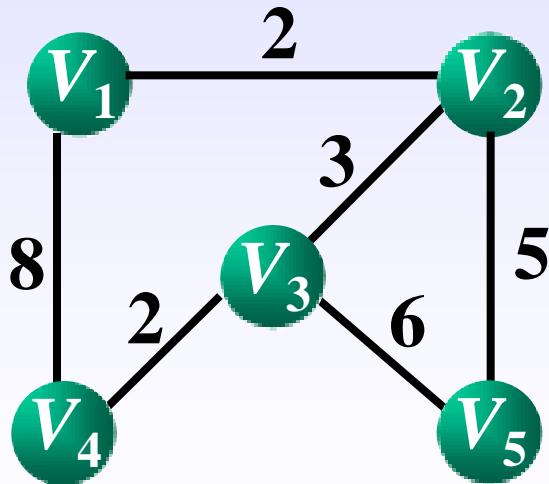
$V_1 V_2 V_3 V_4$: 长度为3

$V_1 V_2 V_5 V_3 V_4$: 长度为4

6.1 图的逻辑结构

图的基本术语

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

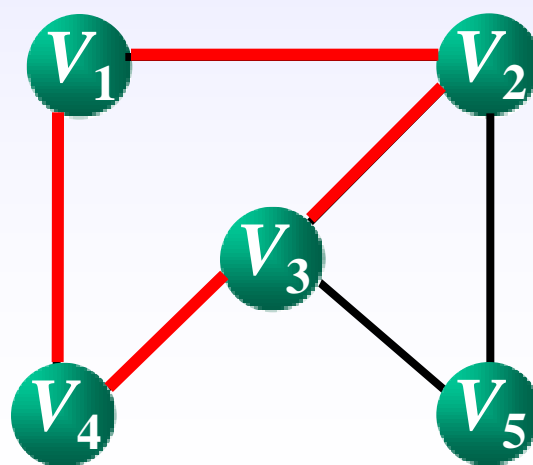
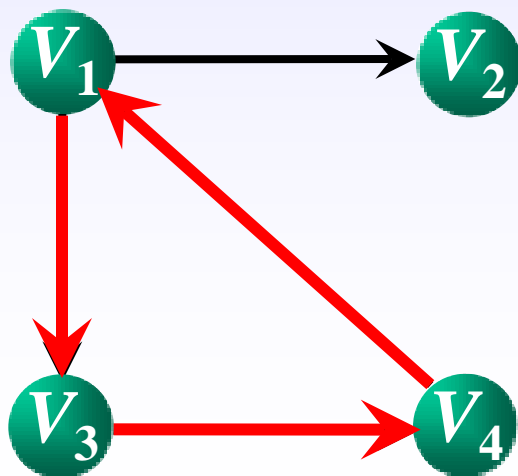
6.1 图的逻辑结构

图的基本术语

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。

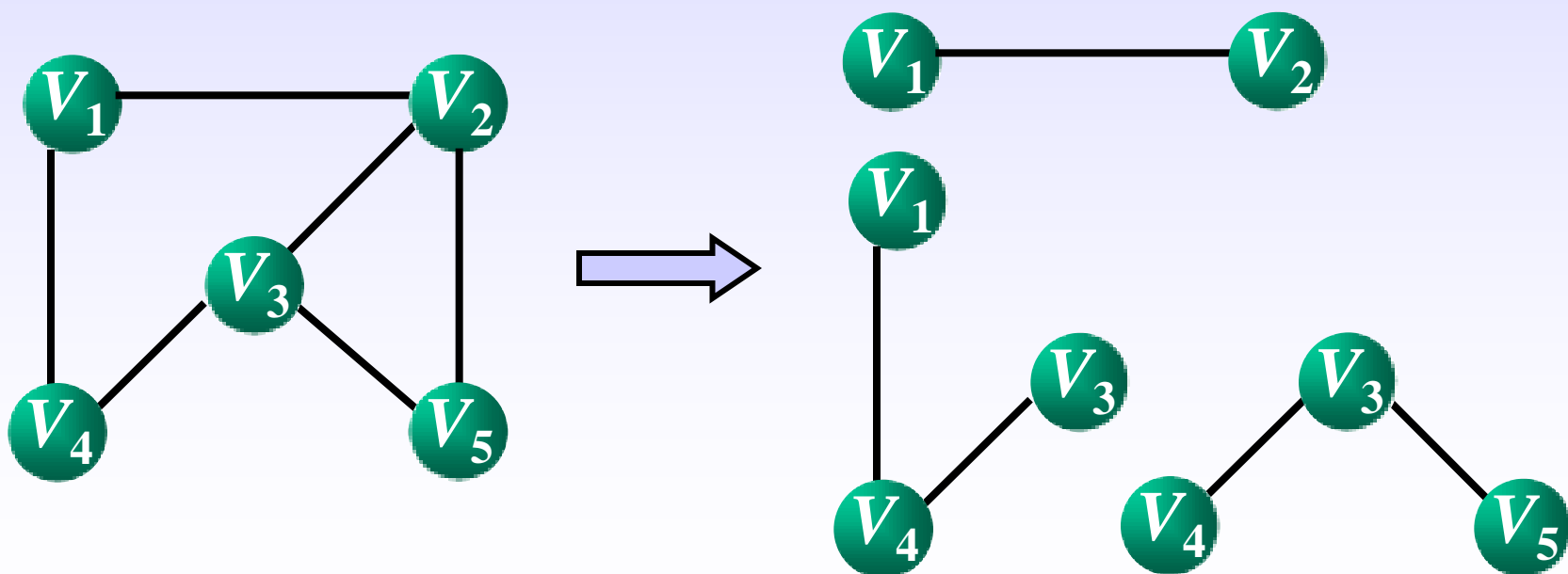
简单回路（简单环）：除了第一个顶点和最后一个顶点外，其余顶点不重复出现的回路。



6.1 图的逻辑结构

图的基本术语

子图：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。

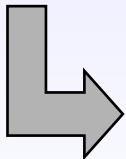


6.1 图的逻辑结构

图的基本术语

连通图：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。

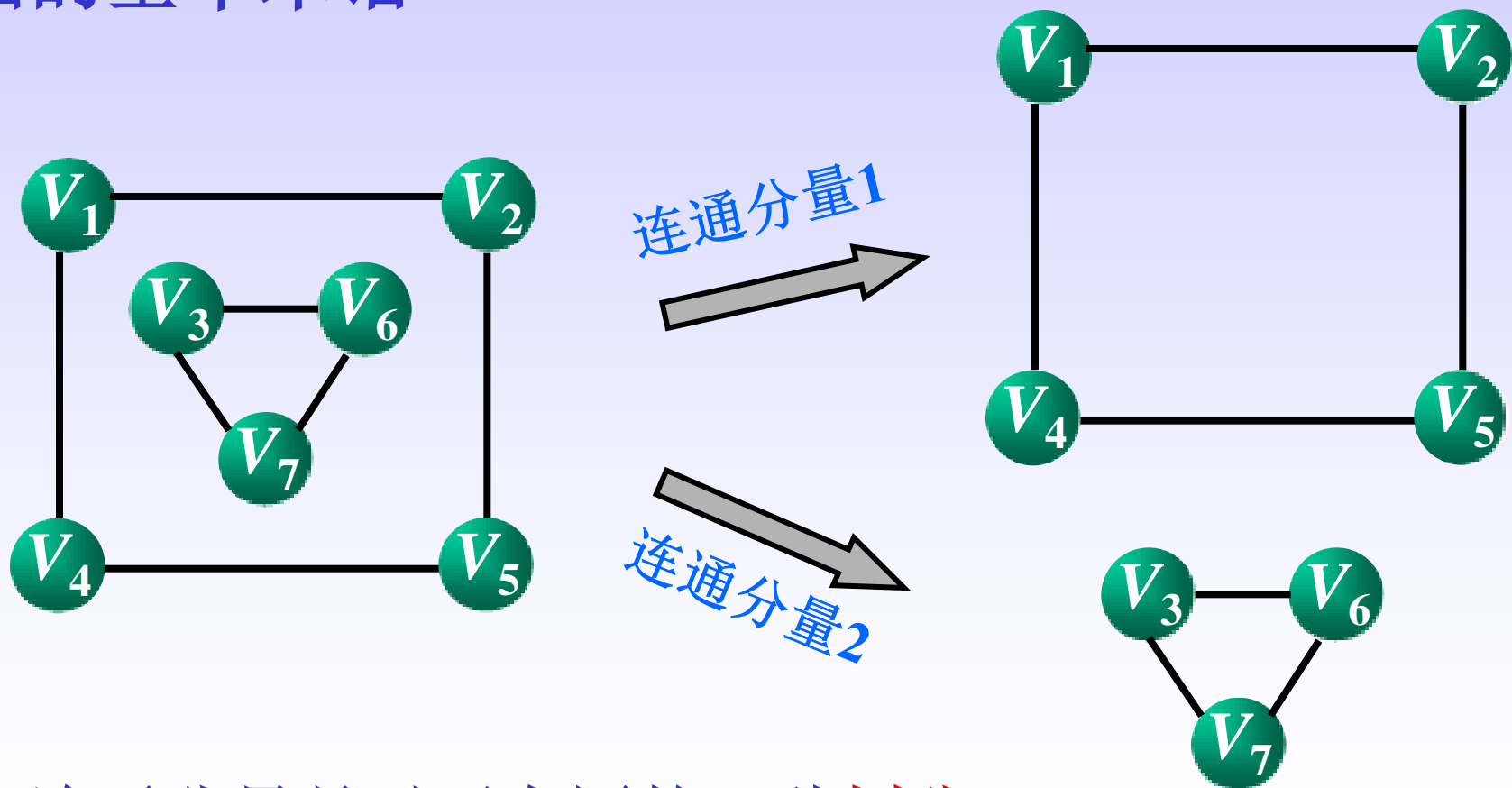
- 
- 1. 含有极大**顶点**数；
 - 2. 依附于这些顶点的所有**边**。



如何求得一个非连通图的连通分量？

6.1 图的逻辑结构

图的基本术语



❖ 连通分量是对无向图的一种划分。

6.1 图的逻辑结构

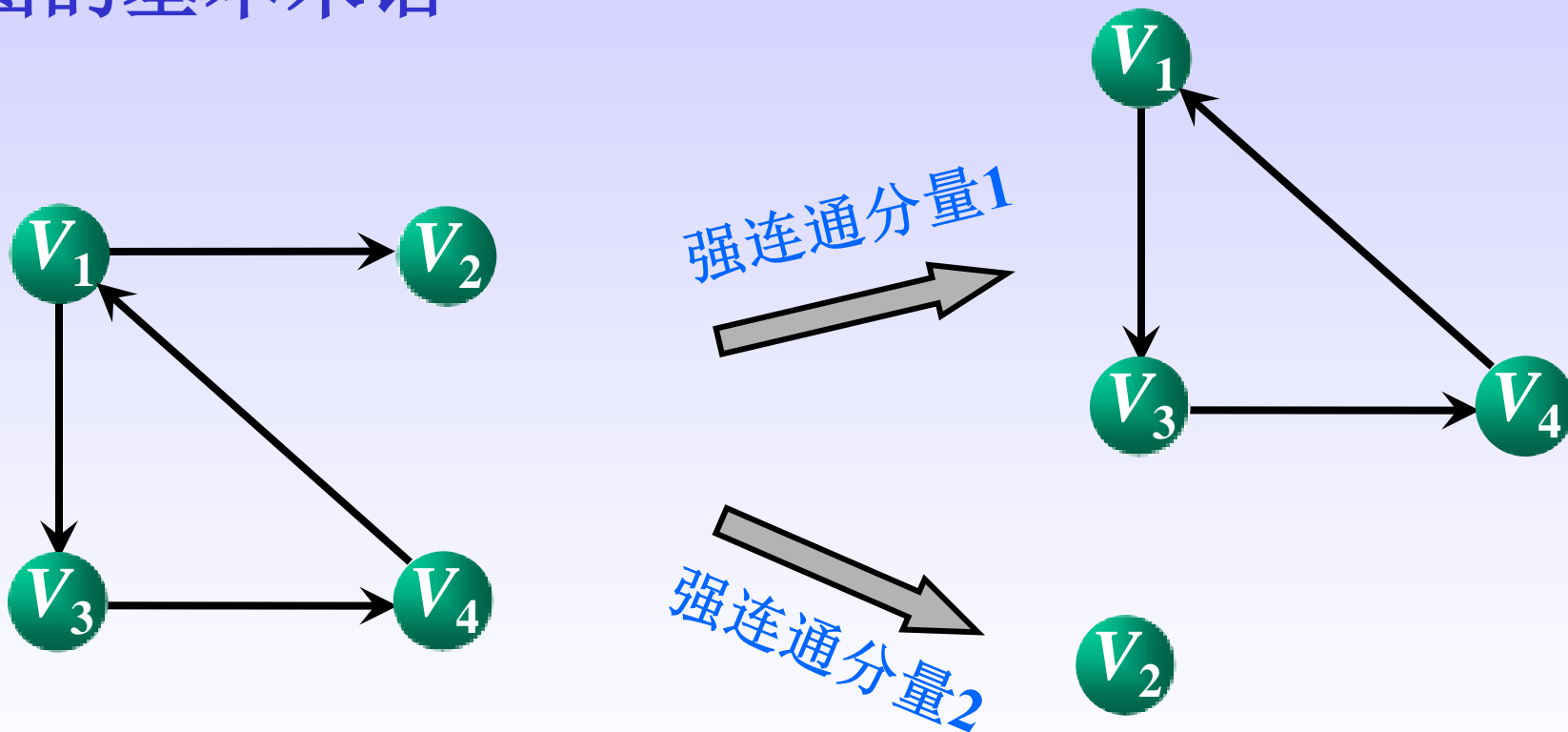
图的基本术语

强连通图：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。

6.1 图的逻辑结构

图的基本术语



6.1 图的逻辑结构

图的基本术语

生成树： 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。

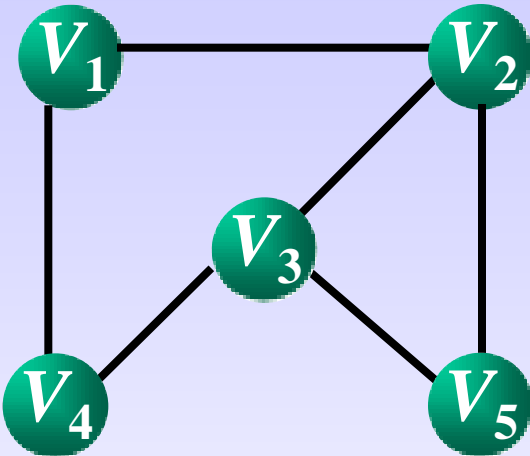
 含有 $n-1$ 条边 $\left\{ \begin{array}{l} \text{多——构成回路} \\ \text{少——不连通} \end{array} \right.$



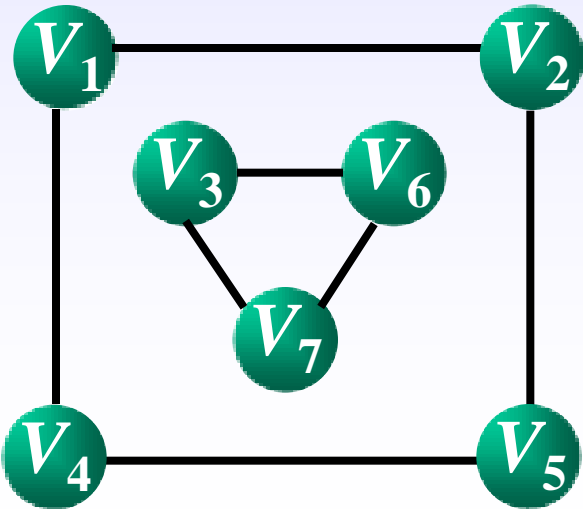
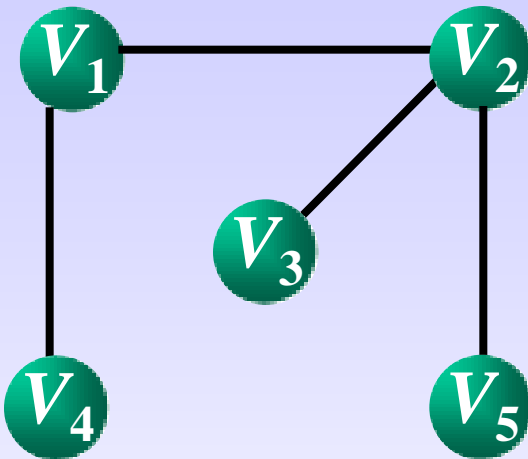
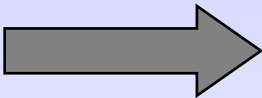
如何理解极小连通子图？

生成森林：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**。

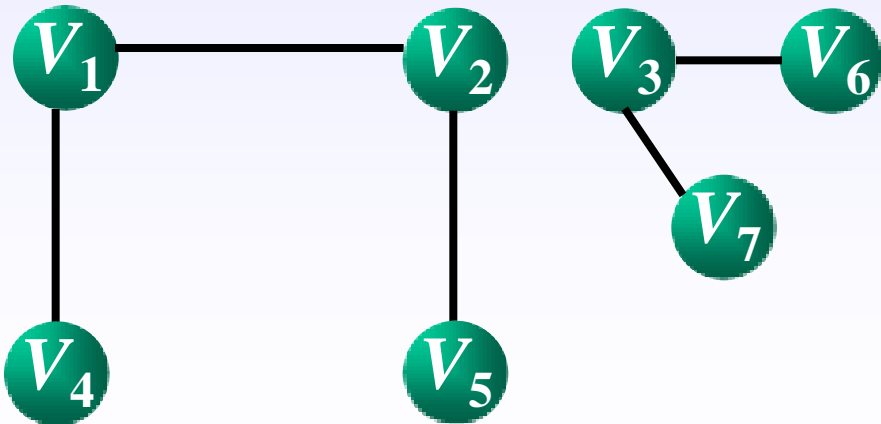
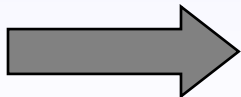
6.1 图的逻辑结构



生成树



生成森林



6.1 图的逻辑结构

图的抽象数据类型定义

ADT Graph

Data

顶点的有穷非空集合和边的集合

Operation

InitGraph

前置条件：图不存在

输入：无

功能：图的初始化

输出：无

后置条件：构造一个空的图

6.1 图的逻辑结构

DFSTraverse

前置条件：图已存在

输入：遍历的起始顶点 v

功能：从顶点 v 出发深度优先遍历图

输出：图中顶点的一个线性排列

后置条件：图保持不变

BFTTraverse

前置条件：图已存在

输入：遍历的起始顶点 v

功能：从顶点 v 出发广度优先遍历图

输出：图中顶点的一个线性排列

后置条件：图保持不变

6.1 图的逻辑结构

DestroyGraph

前置条件：图已存在

输入：无

功能：销毁图

输出：无

后置条件：释放图所占用的存储空间

GetVex

前置条件：图已存在

输入：顶点 v

功能：在图中查找顶点 v 的数据信息

输出：顶点 v 的数据信息

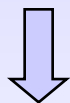
后置条件：图保持不变

endADT

6.1 图的逻辑结构

图的遍历操作

图的遍历是在从图中**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。

6.1 图的逻辑结构

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。

6.1 图的逻辑结构

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。

6.1 图的逻辑结构

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。

6.1 图的逻辑结构

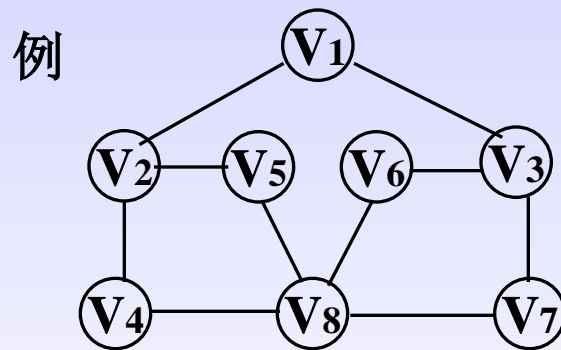
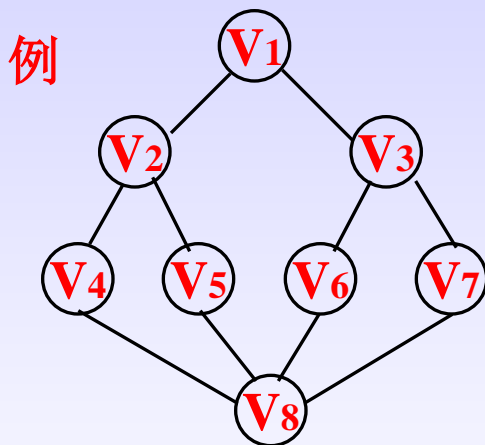
1. 深度优先遍历 (depth-first traverse)

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。

6.1 图的逻辑结构

1. 深度优先遍历 (depth-first traverse)

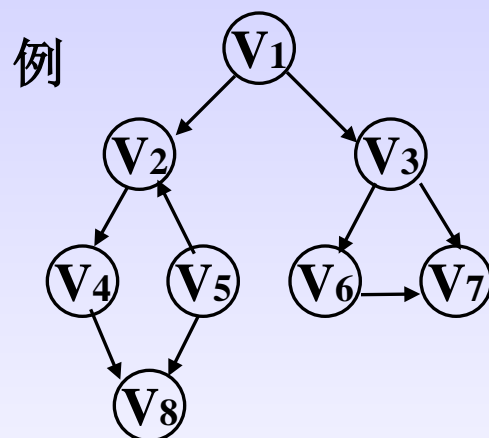


深度优先遍历: $V1 \Rightarrow V2 \Rightarrow V4 \Rightarrow V8 \Rightarrow V5 \Rightarrow V6 \Rightarrow V3 \Rightarrow V7$

深度优先遍历: $V1 \Rightarrow V2 \Rightarrow V4 \Rightarrow V8 \Rightarrow V5 \Rightarrow V6 \Rightarrow V3 \Rightarrow V7$

6.1 图的逻辑结构

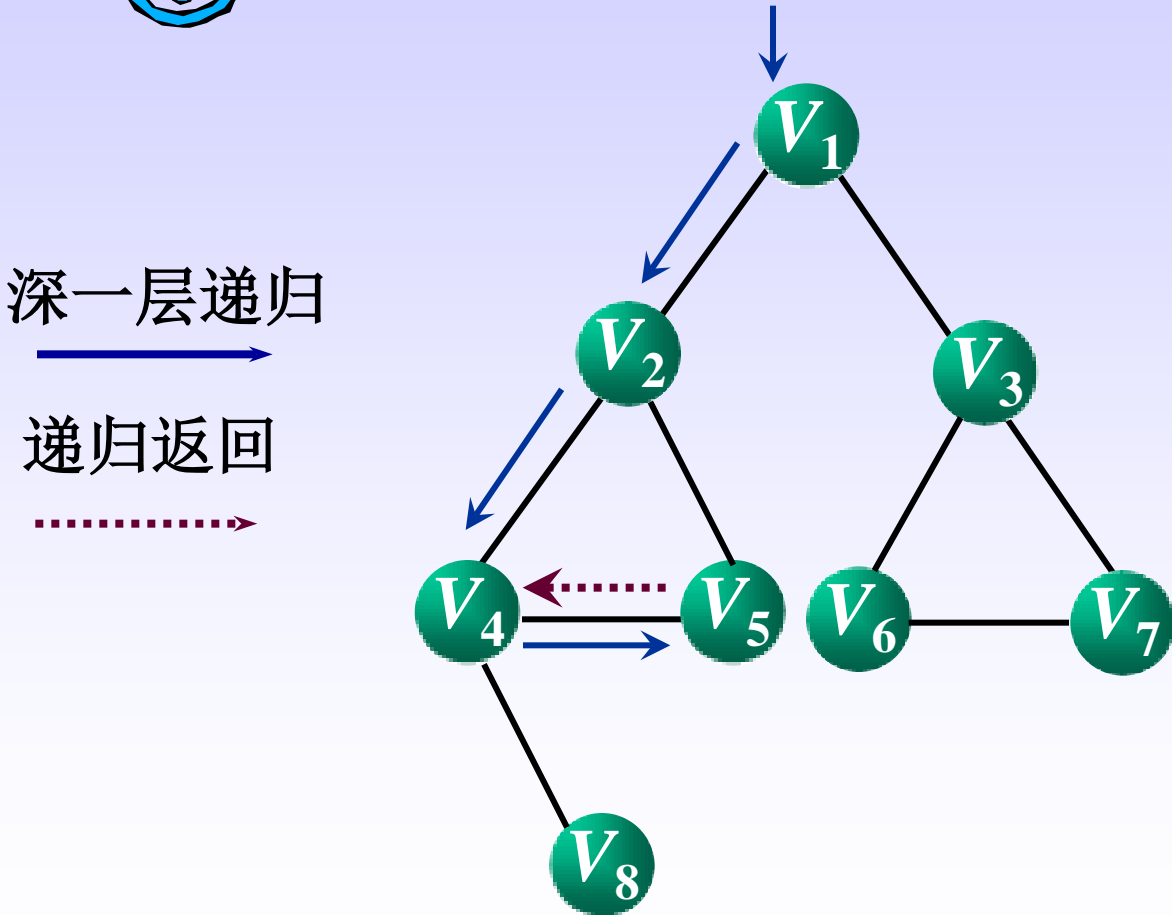
1. 深度优先遍历 (depth-first traverse)



深度优先遍历: $V1 \Rightarrow V2 \Rightarrow V4 \Rightarrow V8 \Rightarrow V3 \Rightarrow V6 \Rightarrow V7 \Rightarrow V5$

6.1 图的逻辑结构

① 深度优先遍历序列?入栈序列?出栈序列?



V_5
V_4
V_2
V_1

遍历序列: V_1 V_2 V_4 V_5

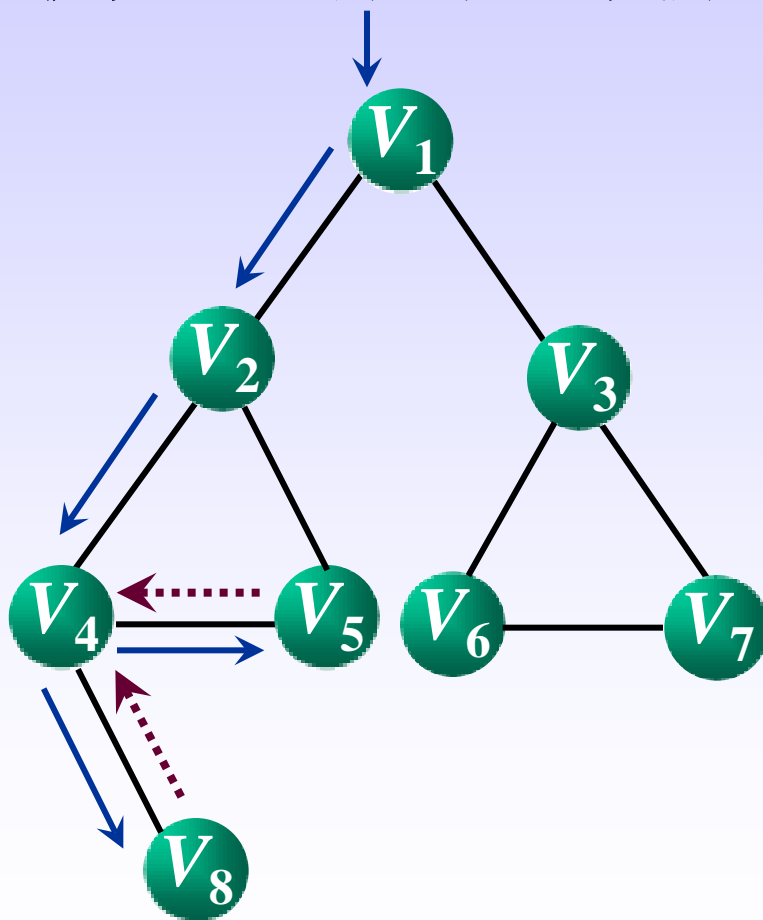
6.1 图的逻辑结构

① 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

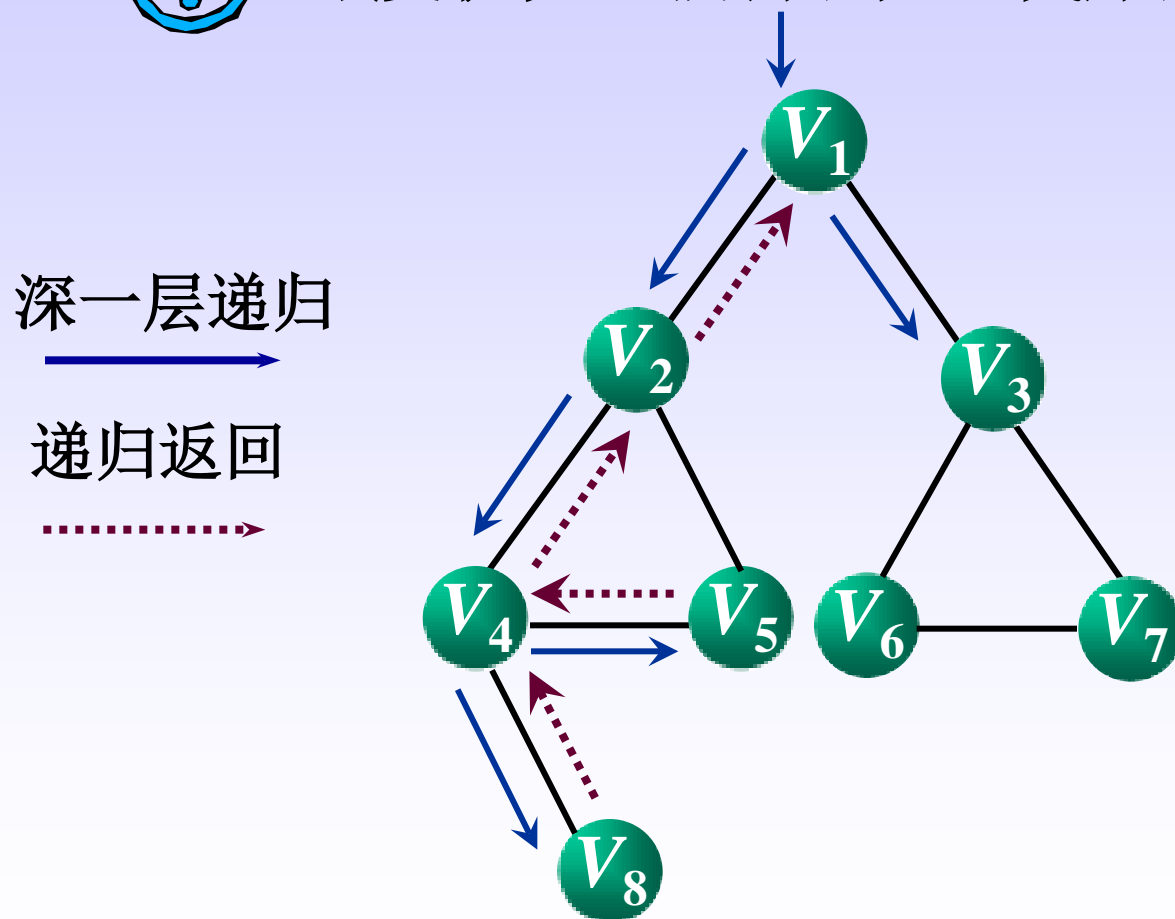


V_8
V_4
V_2
V_1

遍历序列: V_1 V_2 V_4 V_5 V_8

6.1 图的逻辑结构

① 深度优先遍历序列? 入栈序列? 出栈序列?



V_4
V_2
V_1

遍历序列: V_1 V_2 V_4 V_5 V_8

6.1 图的逻辑结构

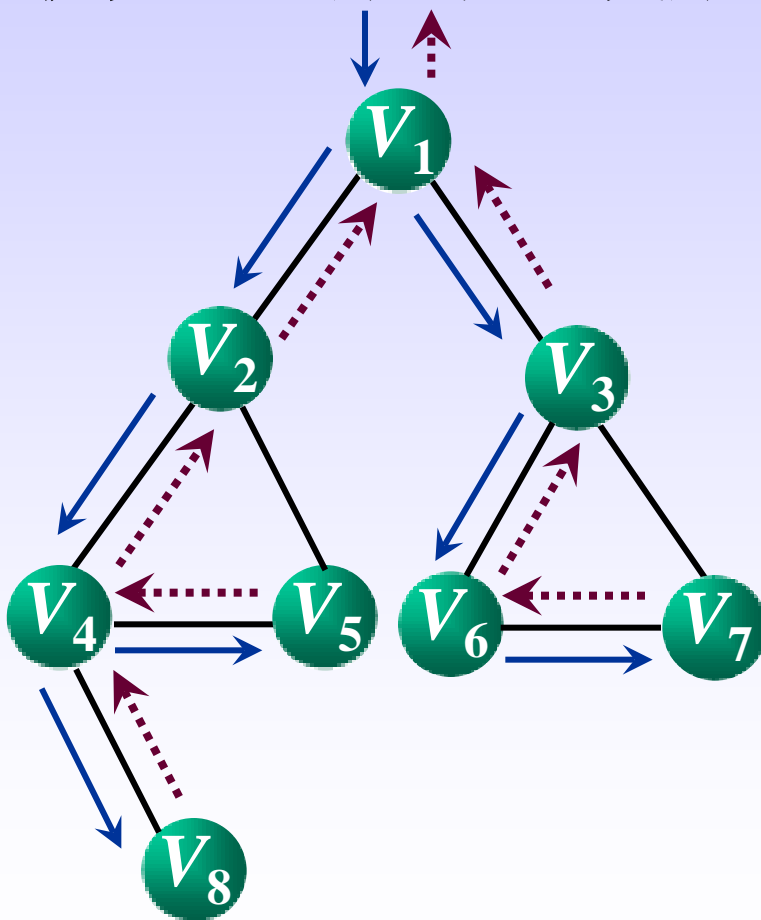


深度优先遍历序列? 入栈序列? 出栈序列?

深一层递归



递归返回



V_7
V_6
V_3
V_1

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

6.1 图的逻辑结构

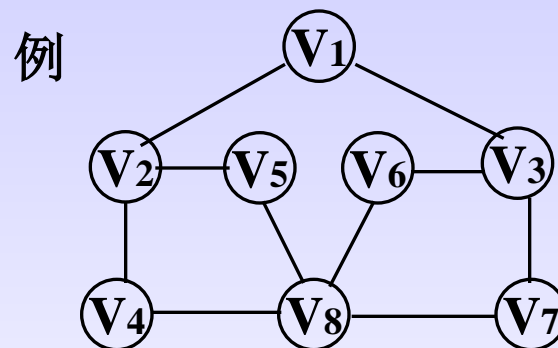
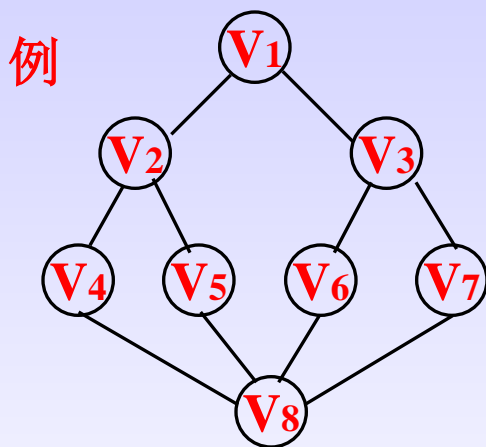
2. 广度优先遍历 (breadth-first traverse)

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。

6.1 图的逻辑结构

2. 广度优先遍历 (breadth-first traverse)

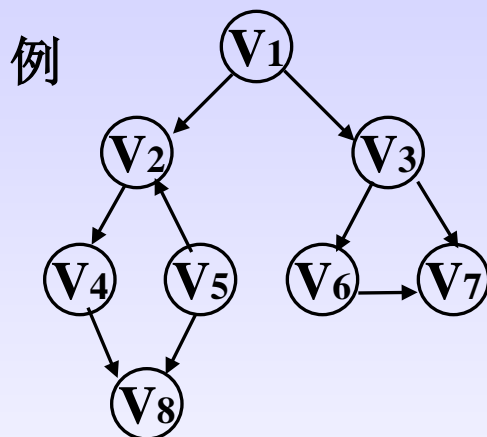


广度优先遍历: $V1 \Rightarrow V2 \Rightarrow V3 \Rightarrow V4 \Rightarrow V5 \Rightarrow V6 \Rightarrow V7 \Rightarrow V8$

广度优先遍历: $V1 \Rightarrow V2 \Rightarrow V3 \Rightarrow V4 \Rightarrow V5 \Rightarrow V6 \Rightarrow V7 \Rightarrow V8$

6.1 图的逻辑结构

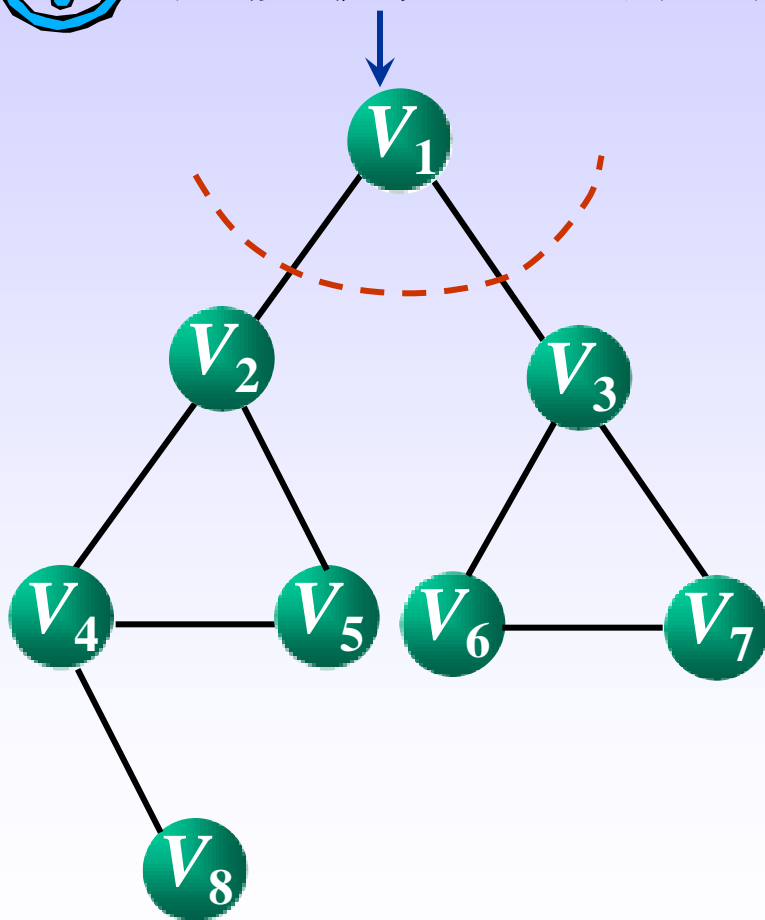
2. 广度优先遍历 (breadth-first traverse)



广度优先遍历: $V1 \Rightarrow V2 \Rightarrow V3 \Rightarrow V4 \Rightarrow V6 \Rightarrow V7 \Rightarrow V8 \Rightarrow V5$

6.1 图的逻辑结构

① 广度优先遍历序列? 入队序列? 出队序列?

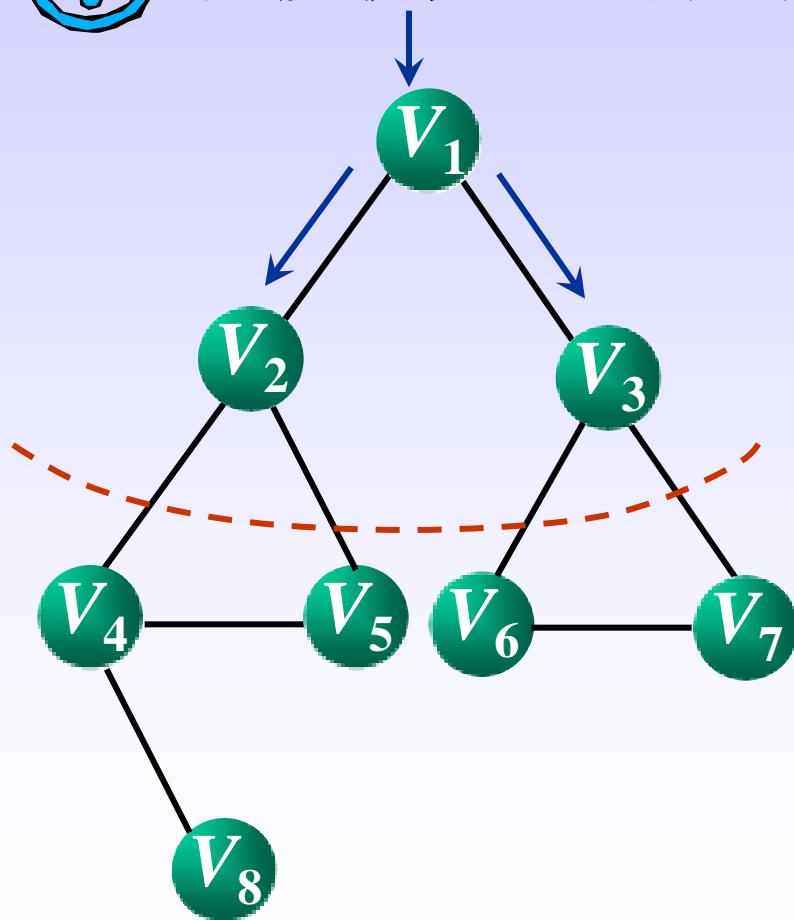


V_1

遍历序列: V_1

6.1 图的逻辑结构

① 广度优先遍历序列? 入队序列? 出队序列?

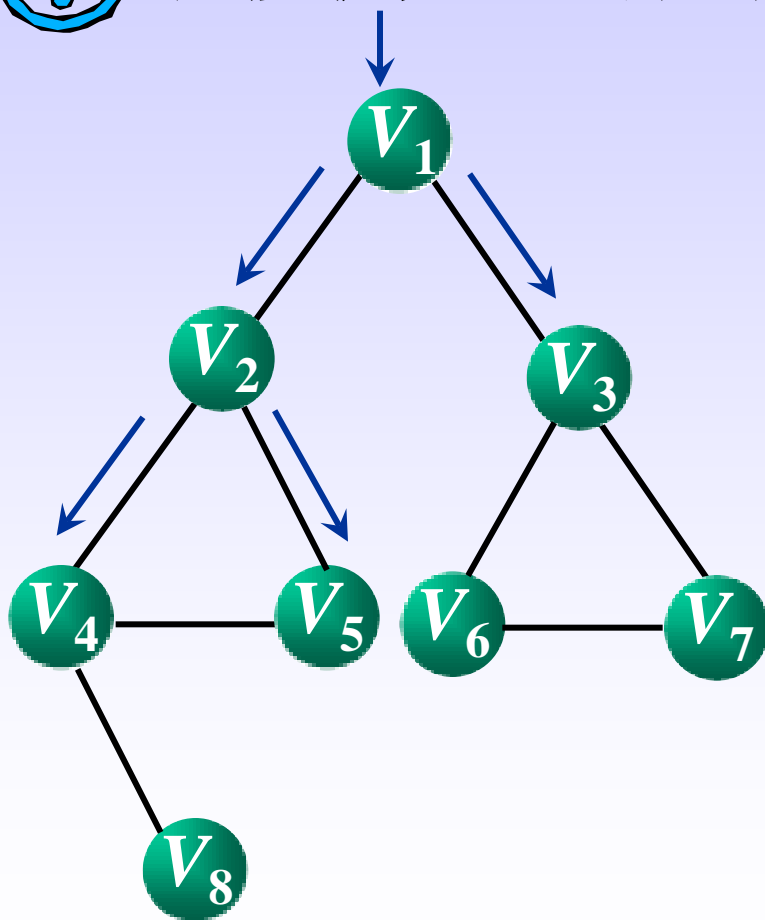


V_2 V_3

遍历序列: V_1 V_2 V_3

6.1 图的逻辑结构

① 广度优先遍历序列? 入队序列? 出队序列?

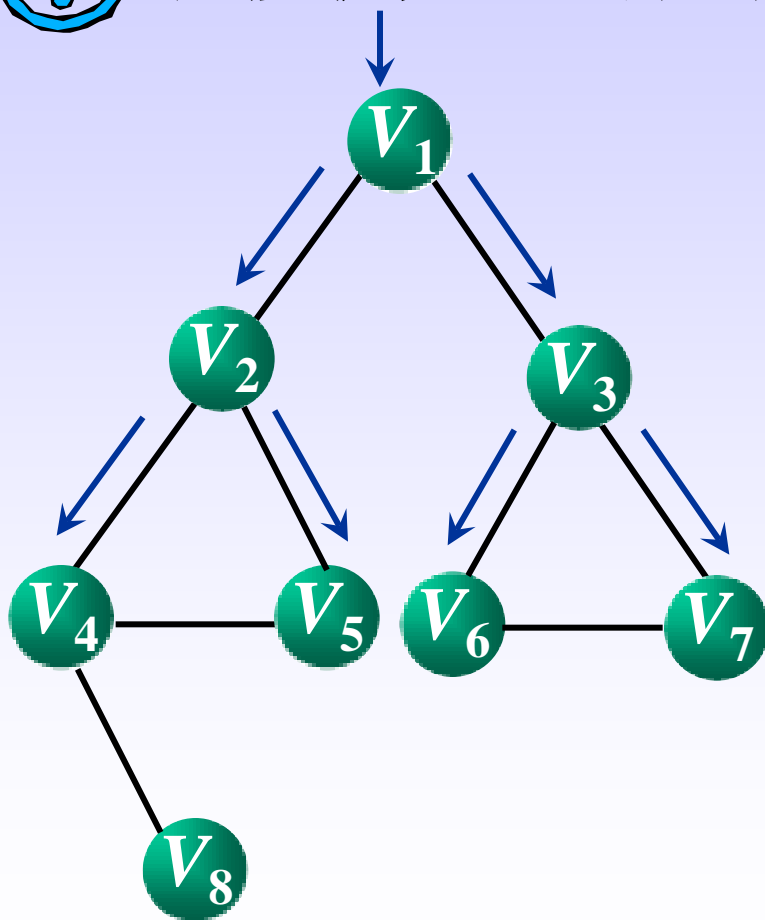


$V_3 \ V_4 \ V_5$

遍历序列: $V_1 \ V_2 \ V_3 \ V_4 \ V_5$

6.1 图的逻辑结构

① 广度优先遍历序列? 入队序列? 出队序列?

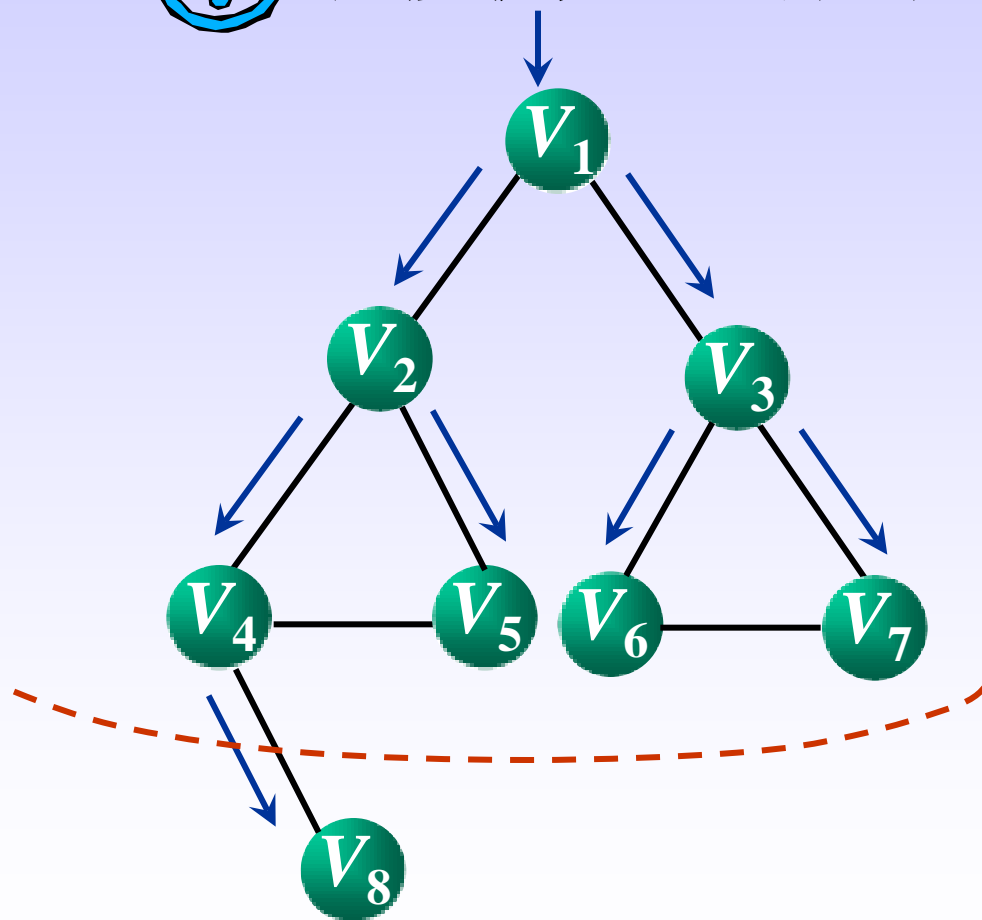


$V_4 V_5 V_6 V_7$

遍历序列: $V_1 V_2 V_3 V_4 V_5 V_6 V_7$

6.1 图的逻辑结构

① 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: $V_1 \ V_2 \ V_3 \ V_4 \ V_5 \ V_6 \ V_7 \ V_8$

6.2 图的存储结构及实现

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。

6.2 图的存储结构及实现

邻接矩阵 (Adjacency Matrix 数组表示法)

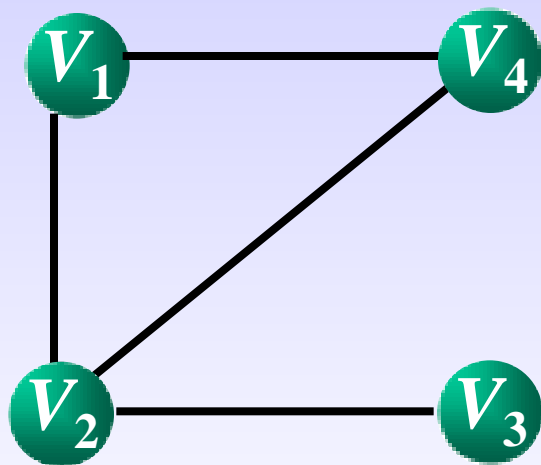
基本思想：用一个一维数组存储图中**顶点**的信息，
用一个二维数组（称为邻接矩阵）存储图中各顶点
之间的**邻接**关系。

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：

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

6.2 图的存储结构及实现

无向图的邻接矩阵



vertex=

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

arc=

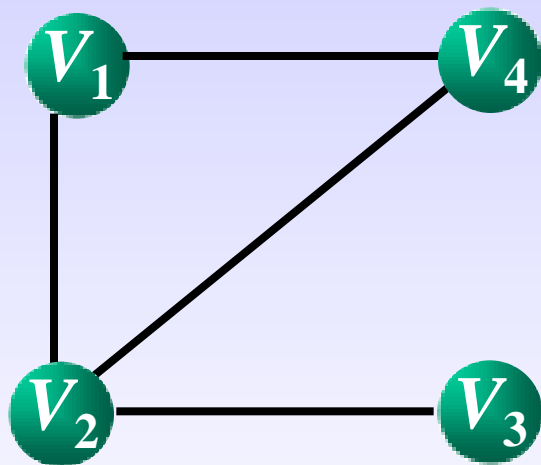
	V_1	V_2	V_3	V_4	
arc=	0	1	0	1	V_1
	1	0	1	1	V_2
	0	1	0	0	V_3
	1	1	0	0	V_4

① 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。

6.2 图的存储结构及实现

无向图的邻接矩阵



vertex=

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

arc=

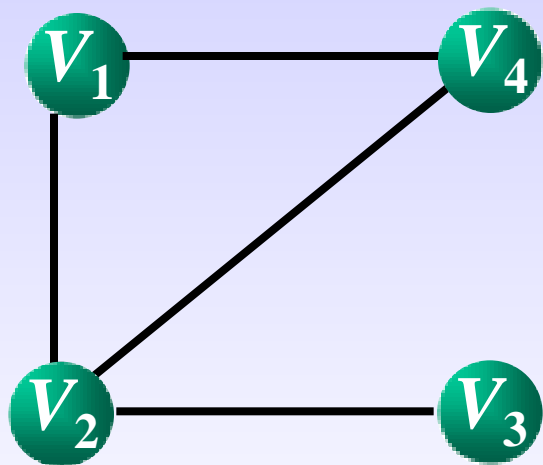
	V_1	V_2	V_3	V_4	
arc=	0	1	0	1	V_1
	1	0	1	1	V_2
	0	1	0	0	V_3
	1	1	0	0	V_4

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。

6.2 图的存储结构及实现

无向图的邻接矩阵



vertex=

V₁

V₂

V₃

V₄

arc=

0

1

0

1

V₁

1

0

1

1

V₂

0

1

0

0

V₃

1

1

0

0

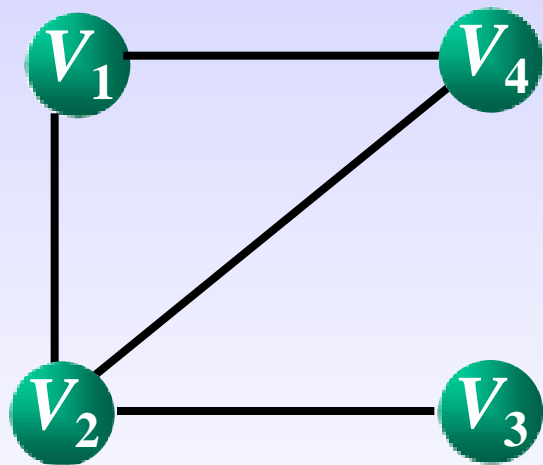
V₄

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。

6.2 图的存储结构及实现

无向图的邻接矩阵



vertex=

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

arc=

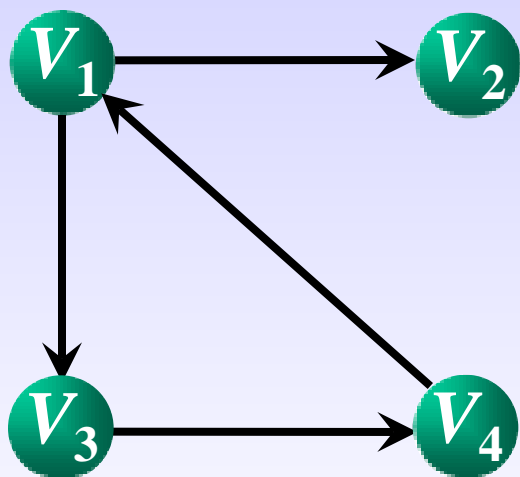
	V_1	V_2	V_3	V_4	
$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$					V_1
					V_2
					V_3
					V_4

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。

6.2 图的存储结构及实现

有向图的邻接矩阵



vertex=

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

arc=

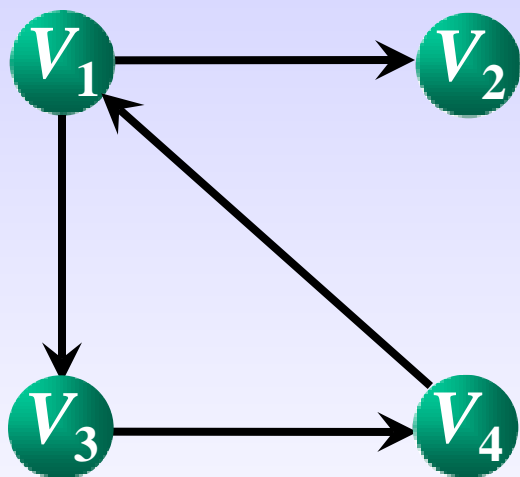
	V_1	V_2	V_3	V_4	
0	1	1	0		V_1
0	0	0	0		V_2
0	0	0	1		V_3
1	0	0	0		V_4

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。

6.2 图的存储结构及实现

有向图的邻接矩阵



vertex=

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

arc=

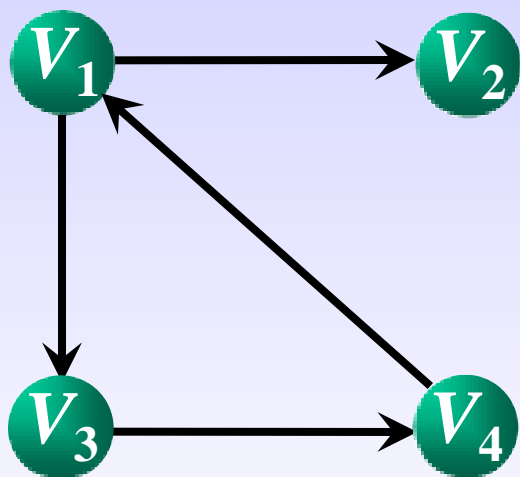
	V_1	V_2	V_3	V_4	
arc=	0	1	1	0	V_1
	0	0	0	0	V_2
	0	0	0	1	V_3
	1	0	0	0	V_4

① 如何求顶点 i 的出度？

邻接矩阵的第 i 行元素之和。

6.2 图的存储结构及实现

有向图的邻接矩阵



vertex=

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

arc=

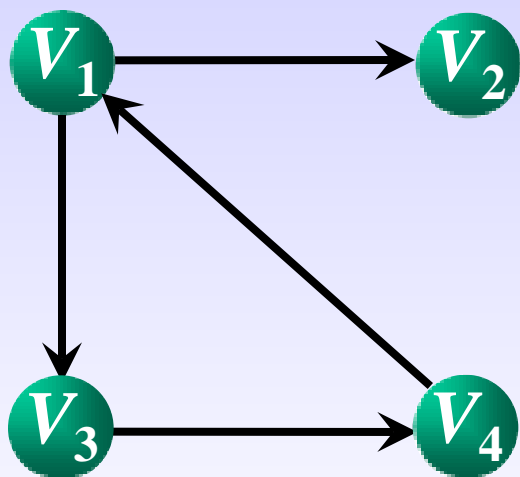
	V_1	V_2	V_3	V_4	
V_1	0	1	1	0	V_1
V_2	0	0	0	0	V_2
V_3	0	0	0	1	V_3
V_4	1	0	0	0	V_4

① 如何求顶点 i 的入度?

邻接矩阵的第 i 列元素之和。

6.2 图的存储结构及实现

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	
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？

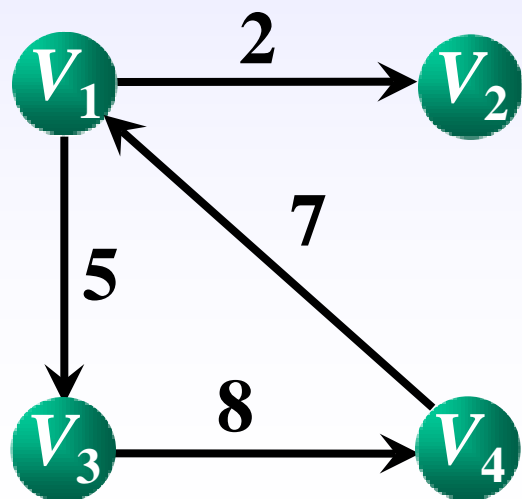
测试邻接矩阵中相应位置的元素 $\text{arc}[i][j]$ 是否为1。

6.2 图的存储结构及实现

网图的邻接矩阵

网图的邻接矩阵可定义为:

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



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

6.2 图的存储结构及实现

邻接矩阵存储无向图的类

```
const int MaxSize=10;
template <class T>
class Mgraph    // Adjacency Matrix
{
public:
    MGraph(T a[ ], int n, int e );
    ~MGraph( )
    void DFSTraverse(int v);
    void BFSTraverse(int v);
private:
    T vertex[MaxSize];
    int arc[MaxSize][MaxSize];
    int vertexNum, arcNum;
};
```

6.2 图的存储结构及实现

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;

6.2 图的存储结构及实现

邻接矩阵中图的基本操作——构造函数

```
template <class T>
MGraph::MGraph(T 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
    }
}
```

6.2 图的存储结构及实现

邻接矩阵中图的基本操作——深度优先遍历

```
template <class T>
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 );
}
```

访问标志数组visited[n]

——全局变量

6.2 图的存储结构及实现

邻接矩阵中图的基本操作——广度优先遍历

```
template <class T>
void MGraph::BFSTraverse (int v)
{
    front=rear=-1; //假设采用顺序队列且不会发生溢出
    cout<<vertex[v]; visited[v]=1; Q[++rear]=v;
    while (front!=rear)
    {
        v=Q[++front];
        for (j=0; j<vertexNum; j++)
            if (arc[v][j]==1 && visited[j]==0 ) {
                cout<<vertex[j]; visited[j]=1; Q[++rear]=j;
            }
    }
}
```

6.2 图的存储结构及实现

邻接表(Adjacency List)



图的邻接矩阵存储结构的时空复杂度？

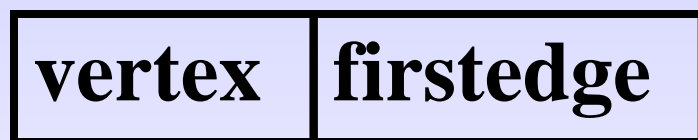
如果为稀疏图则会出现什么现象？

假设图 G 有 n 个顶点 e 条边，则存储该图需要 $O(n^2)$ 。

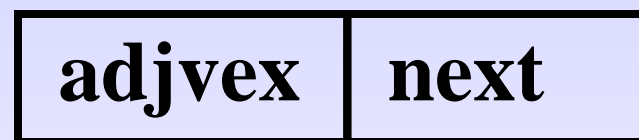
邻接表存储的基本思想：对于图的每个顶点 v_i ，将所有邻接于 v_i 的顶点链成一个单链表，称为顶点 v_i 的**边表**（对于有向图则称为出边表），所有边表的头指针和存储顶点信息的一维数组构成了**顶点表**。

6.2 图的存储结构及实现

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。

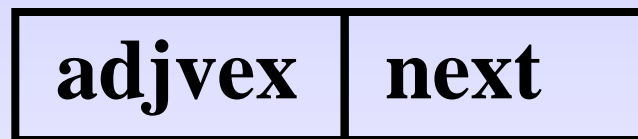
adjvex: 邻接点域，边的终点在顶点表中的下标。

next: 指针域，指向边表中的下一个结点。

6.2 图的存储结构及实现

定义邻接表的结点

```
struct ArcNode
{
    int adjvex;
    ArcNode *next;
};
template <class T>
struct VertexNode
{
    T vertex;
    ArcNode *firstedge;
};
```

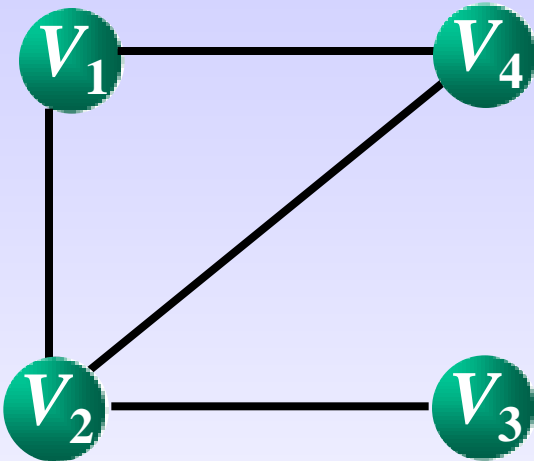


6.2 图的存储结构及实现

无向图的邻接表

❓ 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 $O(n+e)$ 。



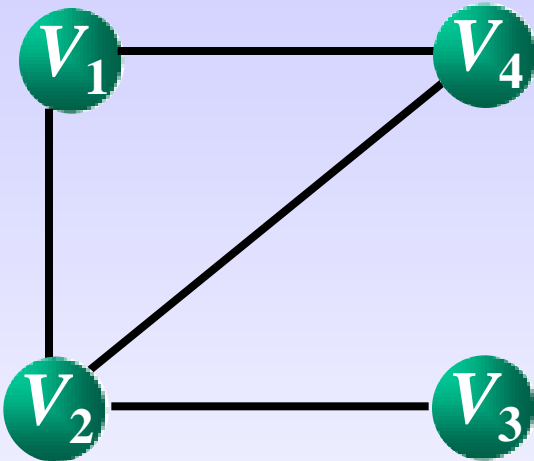
	vertex	firstedge				
0	V ₁	→	1	→	3	∧
1	V ₂	→	0	→	2	→ 3 ∧
2	V ₃	→	1	→	∧	
3	V ₄	→	0	→	1	∧

6.2 图的存储结构及实现

无向图的邻接表

① 如何求顶点 i 的度?

顶点 i 的边表中结点的个数。



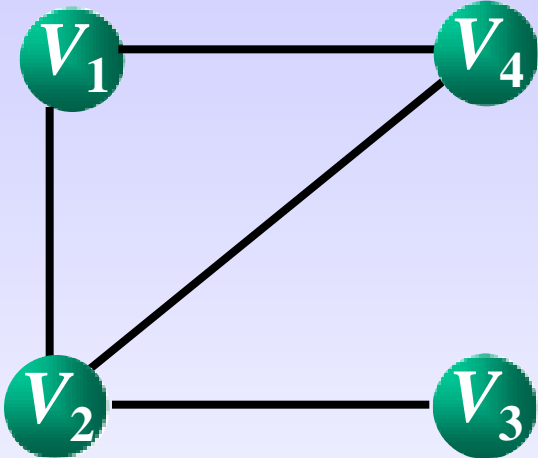
	vertex	firstedge					
0	V ₁	→	1	→	3	∧	
1	V ₂	→	0	→	2	→	3 ∧
2	V ₃	→	1	∧			
3	V ₄	→	0	→	1	∧	

6.2 图的存储结构及实现

无向图的邻接表

❓ 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	
0	V ₁	1	3
1	V ₂	0	2
2	V ₃	1	^
3	V ₄	0	1

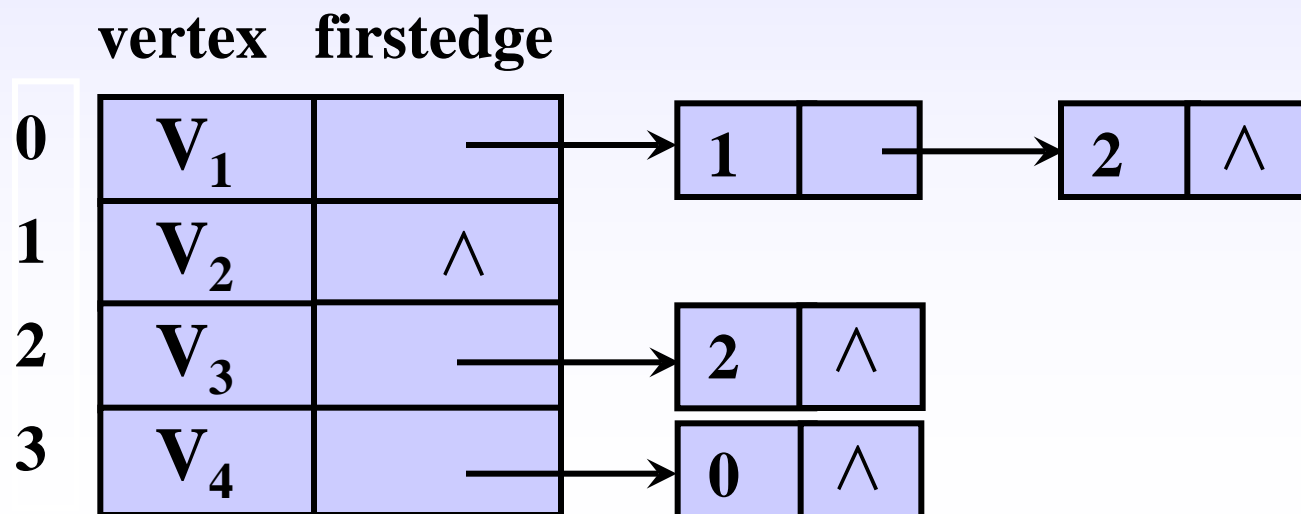
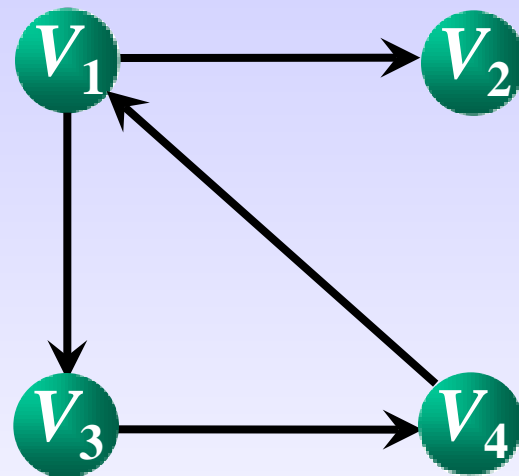
Detailed description: The diagram shows an adjacency list for an undirected graph with 4 vertices. The vertices are indexed 0 to 3. For each vertex, there is a box containing the vertex label and a pointer to the first edge. The edges are represented as a linked list of boxes, each containing an adjacent vertex index and a pointer to the next edge (or null, represented by ^).
- Vertex 0 (V1) points to edge 1, which points to edge 3.
- Vertex 1 (V2) points to edge 0, which points to edge 2, which points to edge 3.
- Vertex 2 (V3) points to edge 1.
- Vertex 3 (V4) points to edge 0, which points to edge 1.

6.2 图的存储结构及实现

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。

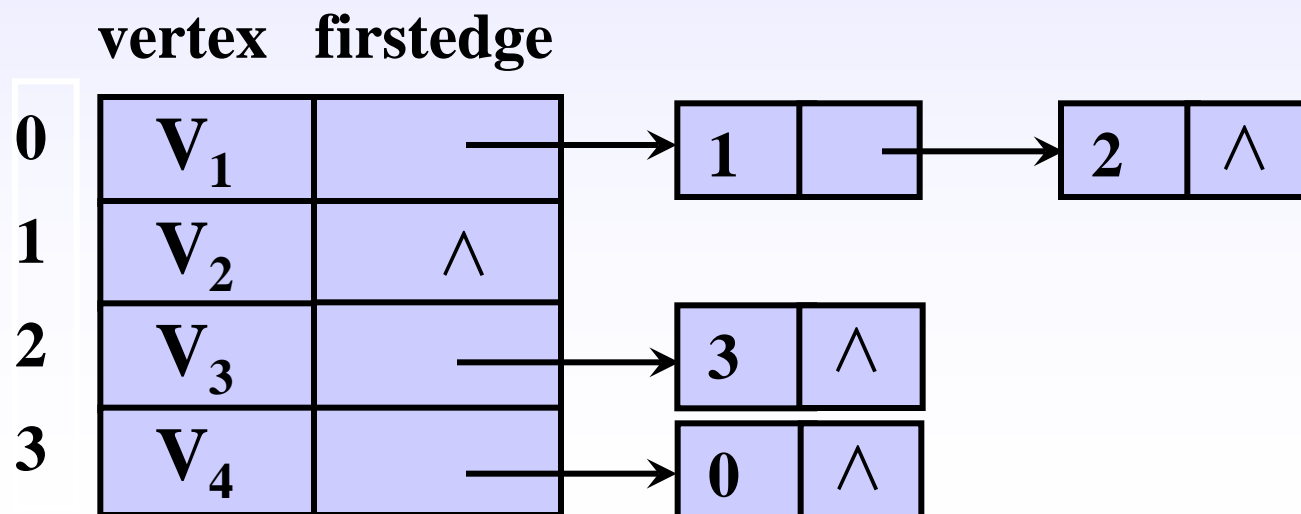
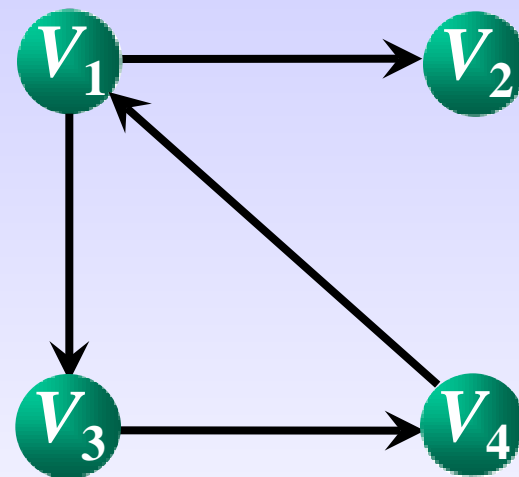


6.2 图的存储结构及实现

有向图的邻接表

② 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。

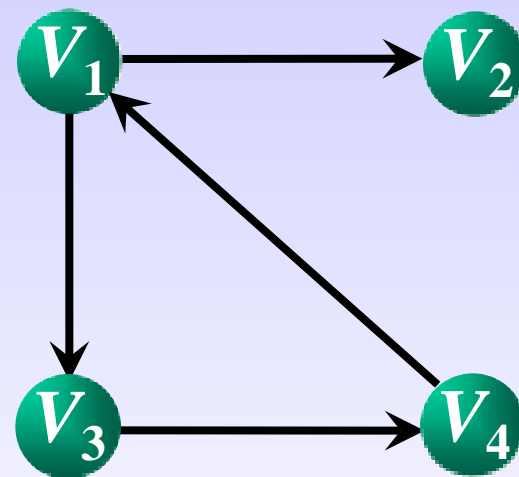


6.2 图的存储结构及实现

有向图的邻接表

① 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。

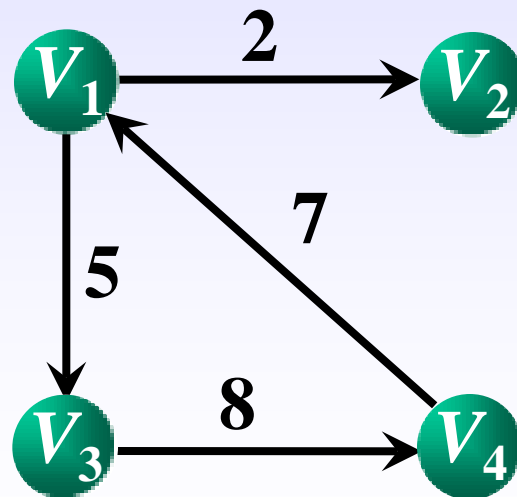
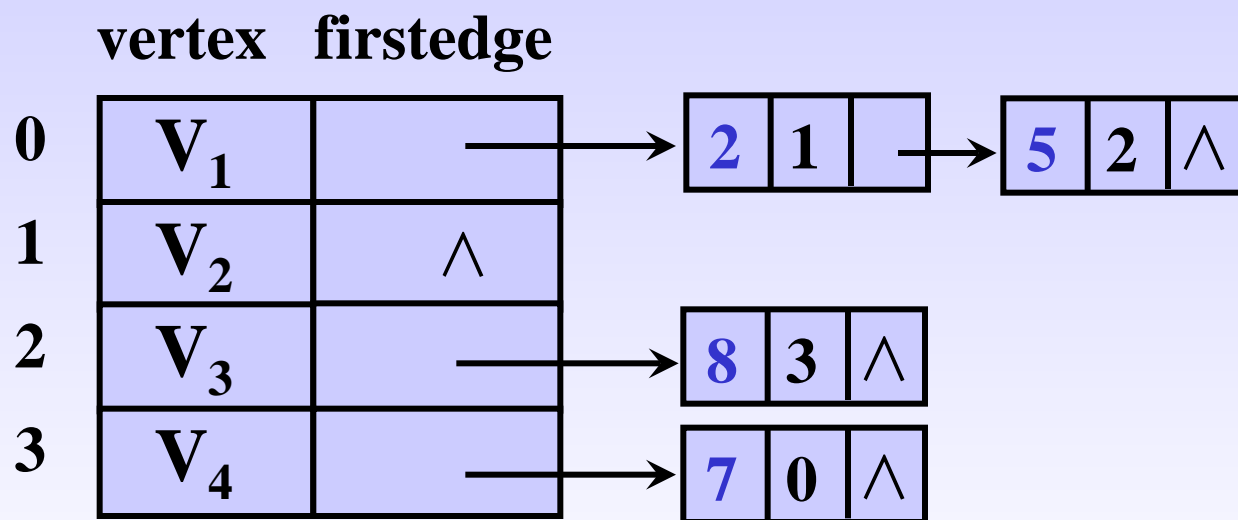


vertex firstedge

0	V ₁	→	1	→	2	∧
1	V ₂	∧				
2	V ₃	→	3	∧		
3	V ₄	→	0	∧		

6.2 图的存储结构及实现

网图的邻接表



6.2 图的存储结构及实现

邻接表存储有向图的类

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

adjvex	next
--------	------

```
struct ArcNode
{
    int adjvex;
    ArcNode *next;
};
template <class T>
struct VertexNode
{
    T vertex;
    ArcNode *firstedge;
};
```

vertex	firstedge
--------	-----------

6.2 图的存储结构及实现

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;

6.2 图的存储结构及实现

邻接表中图的基本操作——构造函数

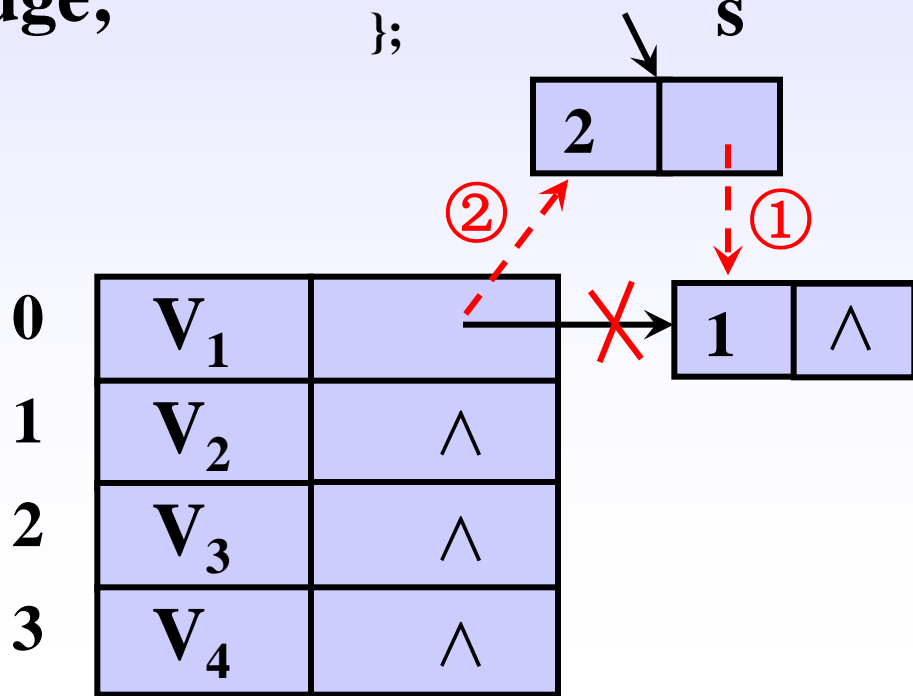
```
template <class T>
ALGraph::ALGraph(T a[ ], int n, int e)
{
    vertexNum=n; arcNum=e;
    for (i=0; i<vertexNum; i++)
        //输入顶点信息，初始化边表
        {
            adjlist[i].vertex=a[i];
            adjlist[i].firstedge=NULL;
        }
}
```

```
struct ArcNode
{
    int adjvex;
    ArcNode *next;
};
template <class T>
struct VertexNode
{
    T vertex;
    ArcNode *firstedge;
};
```

6.2 图的存储结构及实现

```
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;
}
```

```
struct ArcNode
{   int adjvex;
    ArcNode *next;
};
template <class T>
struct VertexNode
{
    T vertex;
    ArcNode *firstedge;
};
```



6.2 图的存储结构及实现

邻接表中图的基本操作——深度优先遍历

```
template <class T>
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;
    }
}
```

```
struct ArcNode
{
    int adjvex;
    ArcNode *next;
};
template <class T>
struct VertexNode
{
    T vertex;
    ArcNode *firstedge;
};
```

访问标志数组visited[n]

——全局变量

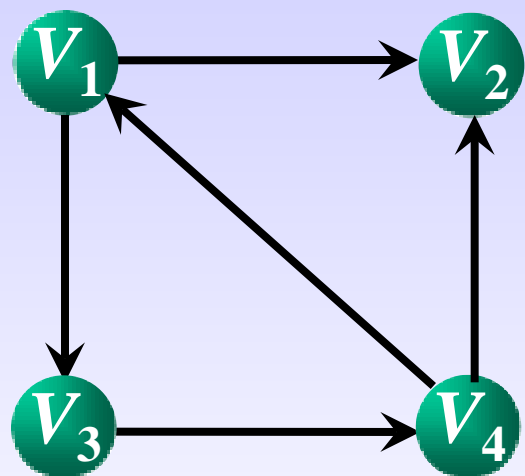
6.2 图的存储结构及实现

邻接表中图的基本操作——广度优先遍历

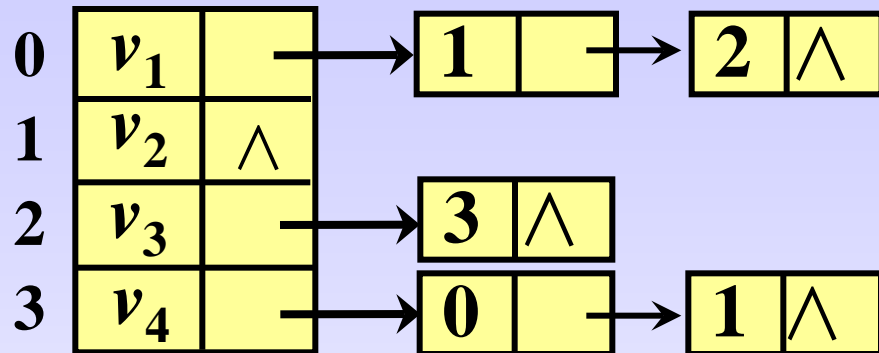
```
template <class T>
void ALGraph::BFSTraverse (int v)
{
    front=rear=-1;
    cout<<adjlist[v].vertex;   visited[v]=1;  Q[++rear]=v;
    while (front!=rear)
    {
        v=Q[++front];  p=adjlist[v].firstedge;
        while (p!=NULL)
        {
            j= p->adjvex;
            if (visited[j]==0) {
                cout<<adjlist[j].vertex;  visited[j]=1; Q[++rear]=j;
            }
            p=p->next;
        }
    }
}
```

6.2 图的存储结构及实现

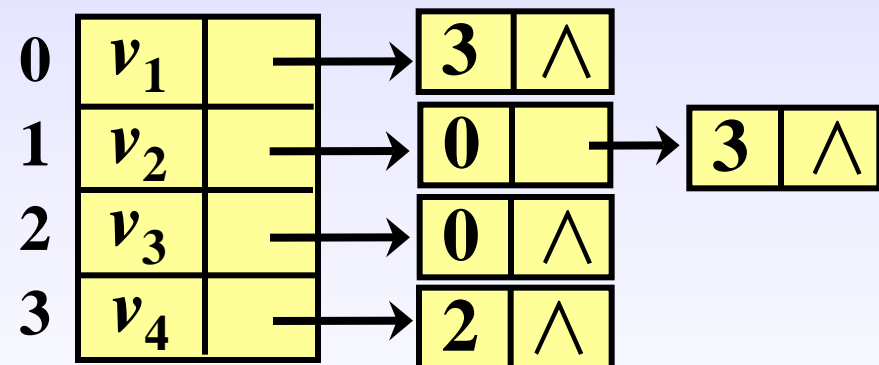
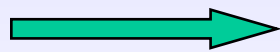
逆邻接表



邻接表



逆邻接表

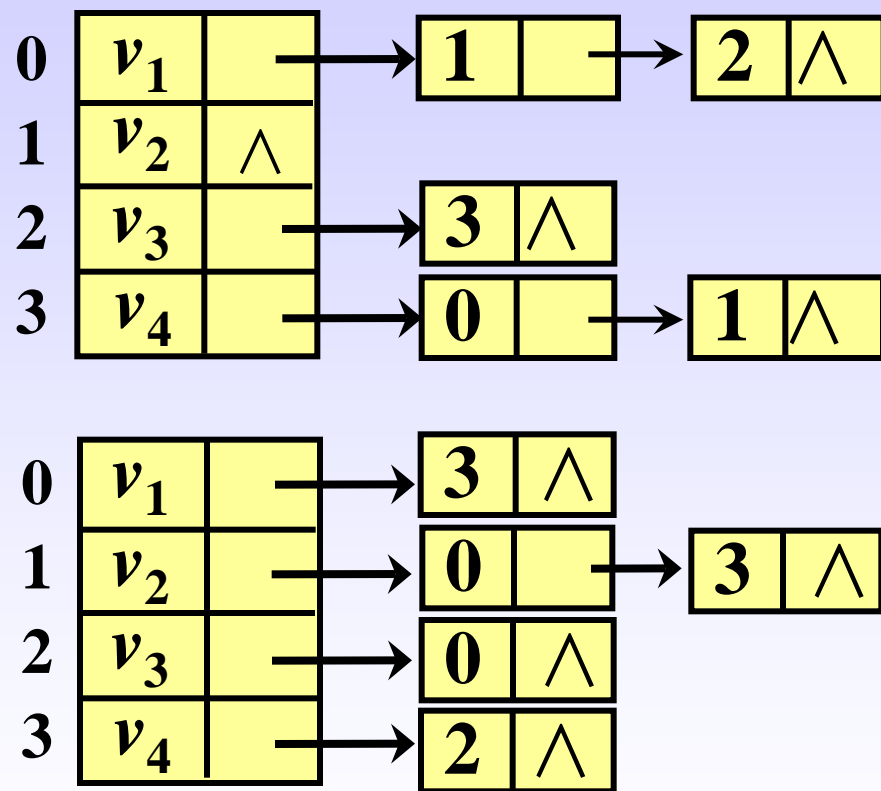
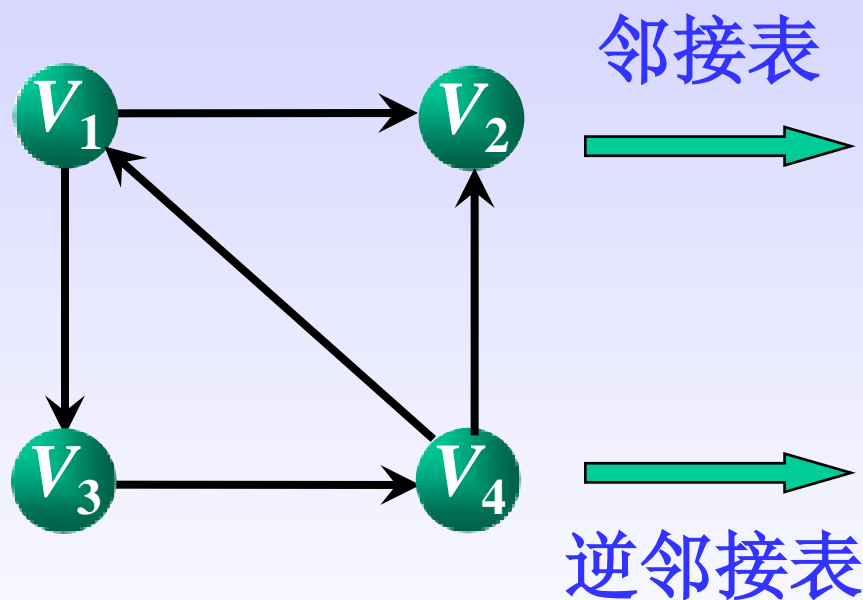


邻接表：有向图中对每个结点建立以 V_i 为尾的弧的单链表，即：出边表

逆邻接表：有向图中对每个结点建立以 V_i 为头的弧的单链表

6.2 图的存储结构及实现

十字链表



① 将邻接表与逆邻接表合二为一?为什么要合并?

6.2 图的存储结构及实现

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: 弧的起点在顶点表中的下标;

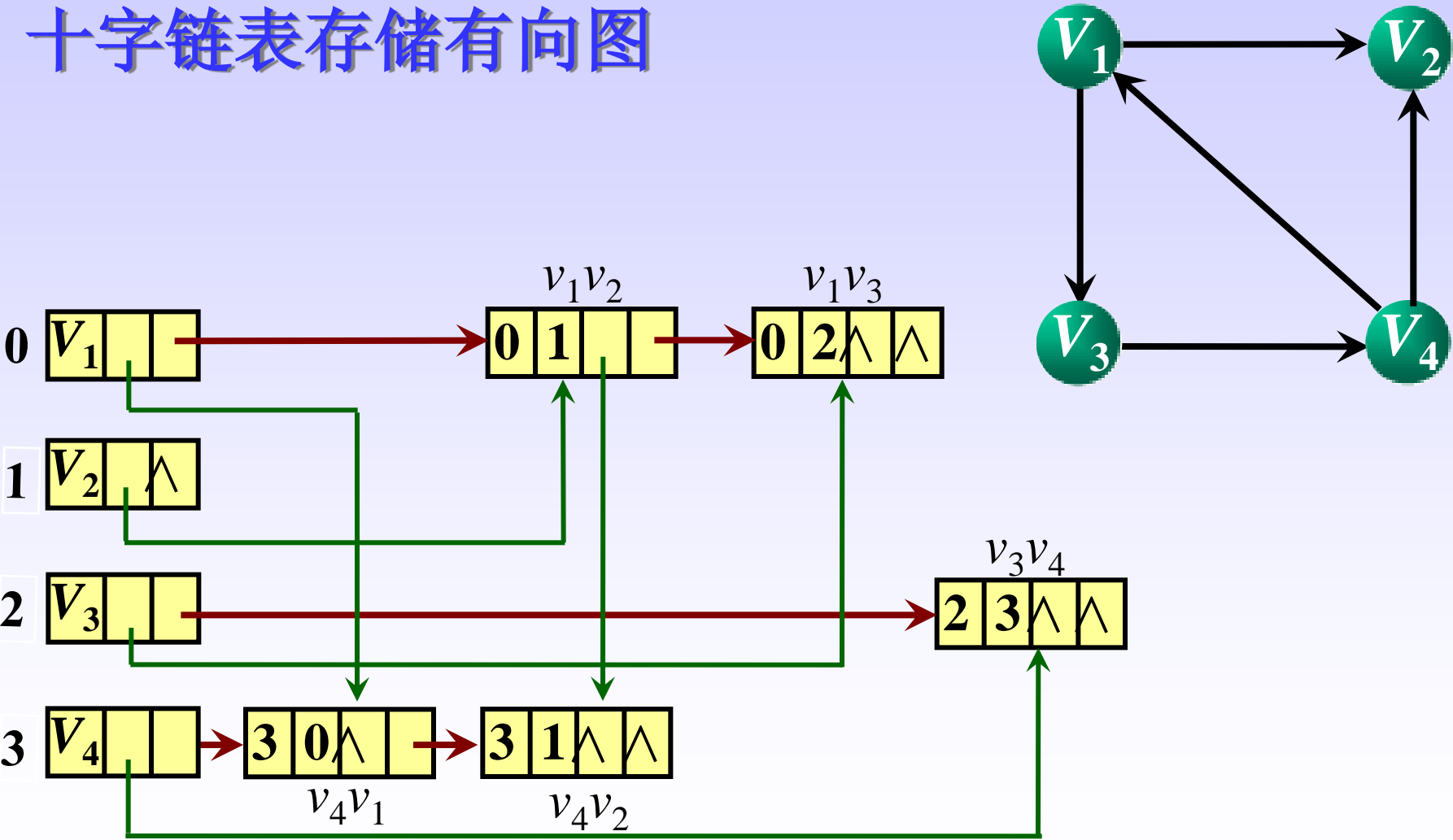
headvex: 弧的终点在顶点表中的下标;

headlink: 入边表指针域, 指向弧**头**相同的下一条弧;

taillink: 出边表指针域, 指向弧**尾**相同的下一条弧。

6.2 图的存储结构及实现

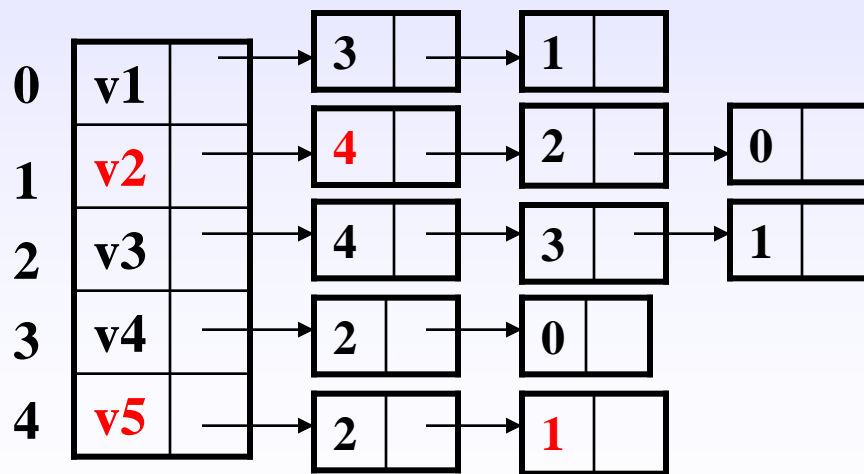
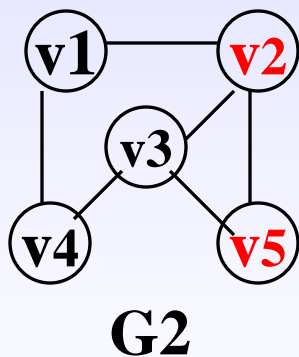
十字链表存储有向图



6.2 图的存储结构及实现

邻接多重表 (Adjacency Multilist)

- 是**无向图**的另一种链式存储结构。
- 虽然邻接表是无向图的一种很有效的存储结构，在邻接表中容易求得顶点和边的各种信息。



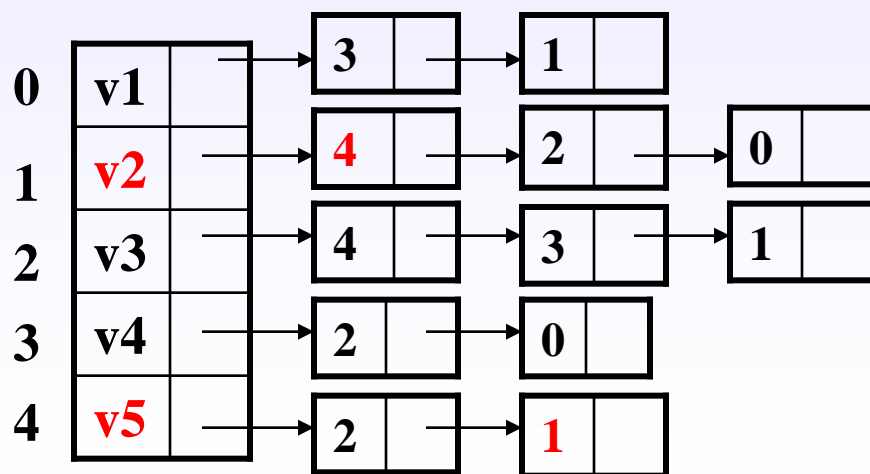
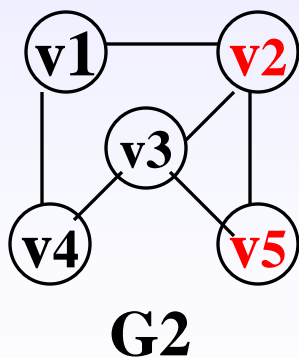
(b) G2的邻接表

6.2 图的存储结构及实现

邻接多重表 (Adjacency Multilist)

- 在邻接表中每一条边(v_i, v_j)有两个结点，分别在第*i*个和第*j*个链表中，这给某些图的操作带来不便。如对已被搜索过的边作记号或删除一条边等，此时需要找到表示同一条边的两个结点。

因此，在进行这一类操作的无向图的问题中采用**邻接多重表**作存储结构更为适宜。



(b)G2的邻接表

6.2 图的存储结构及实现

邻接多重表的结构

- ✎ 每一条边用一个结点表示，它由如下所示的六个域组成：
- ❖ **mark** 为标志域，用以标记该条边是否被搜索过；
 - ❖ **ivex** 和 **jvex** 为该边依附的两个顶点在图中的位置；
 - ❖ **ilink** 指向下一条依附于顶点 **ivex** 的边；
 - ❖ **jlink** 指向下一条依附于顶点 **jvex** 的边，
 - ❖ **info** 为指向和边相关的各种信息的指针域。
- ✎ 每一个顶点用一个结点表示，它由如下所示的两个域组成：
- ❖ **data** 域存储和该顶点相关的信息，
 - ❖ **firstedge** 域指示第一条依附于该顶点的边。

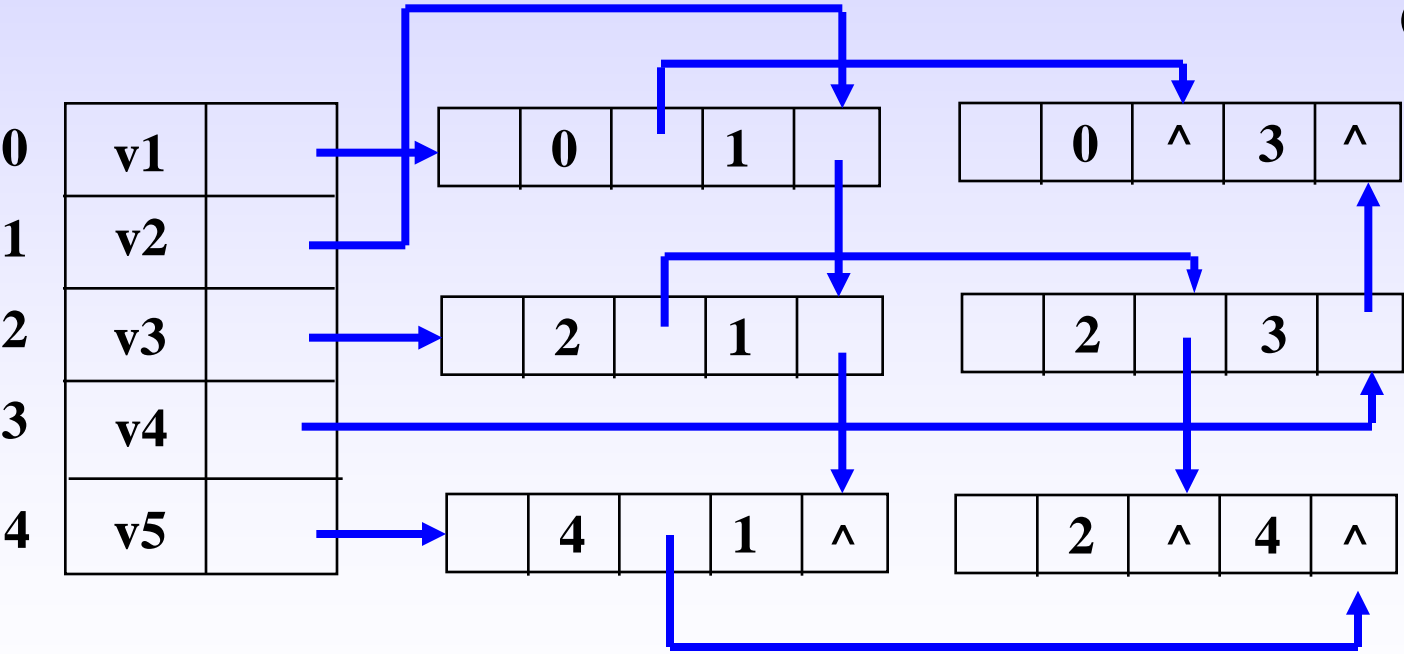
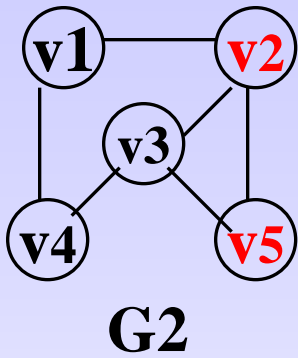
mark	ivex	ilink	jvex	jlink	info
------	------	-------	------	-------	------

data	firstedge
------	-----------

6.2 图的存储结构及实现

邻接多重表的结构

例如，无向图G2的邻接多重表:



6.2 图的存储结构及实现

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 (边的排列)

6.3 最小生成树

无向图的连通性

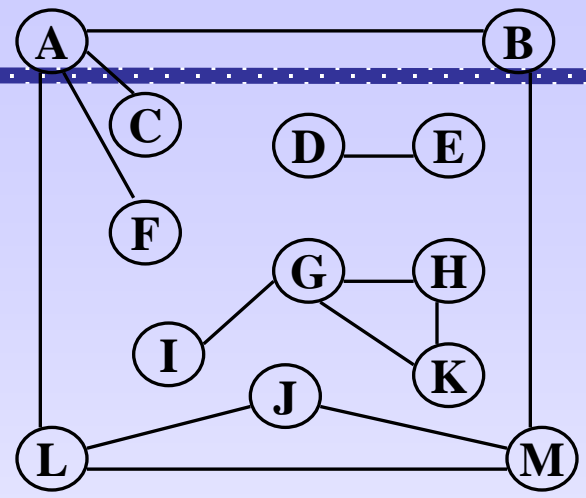
要想判定一个无向图是否为连通图，或有几个连通分量，通过对无向图遍历即可得到结果。

➤ 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。

图G3的邻接表

0	A	→	11	→	5	→	2	→	1	
1	B	→	12	→	0					
2	C	→	0							
3	D	→	4							
4	E	→	3							
5	F	→	0							
6	G	→	10	→	8	→	7			
7	H	→	10	→	6					
8	I	→	6							
9	J	→	12	→	11					
10	K	→	7	→	6					
11	L	→	12	→	9	→	0			
12	M	→	11	→	9	→	1			



G3

非连通图**G3**按照所示邻接表进行深度优先搜索遍历，三次调用DFS过程：

- 从顶点A出发得到的顶点访问序列为：**ALMJBFC**
- 从顶点D出发得到的顶点访问序列为：**DE**
- 从顶点G出发得到的顶点访问序列为：**GKHI**

这三个顶点集分别加上所有依附于这些顶点的边，构成非连通图G3的三个**连通分量**。

6.3 最小生成树

求无向图的连通分量

1. **count=0;** //遍历算法调用的次数
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";

6.3 最小生成树

生成树

设 $E(G)$ 为连通子图 G 中所有边的集合,

• 则从图中任一顶点出发遍历图时, 必定将 $E(G)$ 分成两个集合: $T(G)$ 和 $B(G)$, 其中 $T(G)$ 是遍历过程中历经的边的集合。

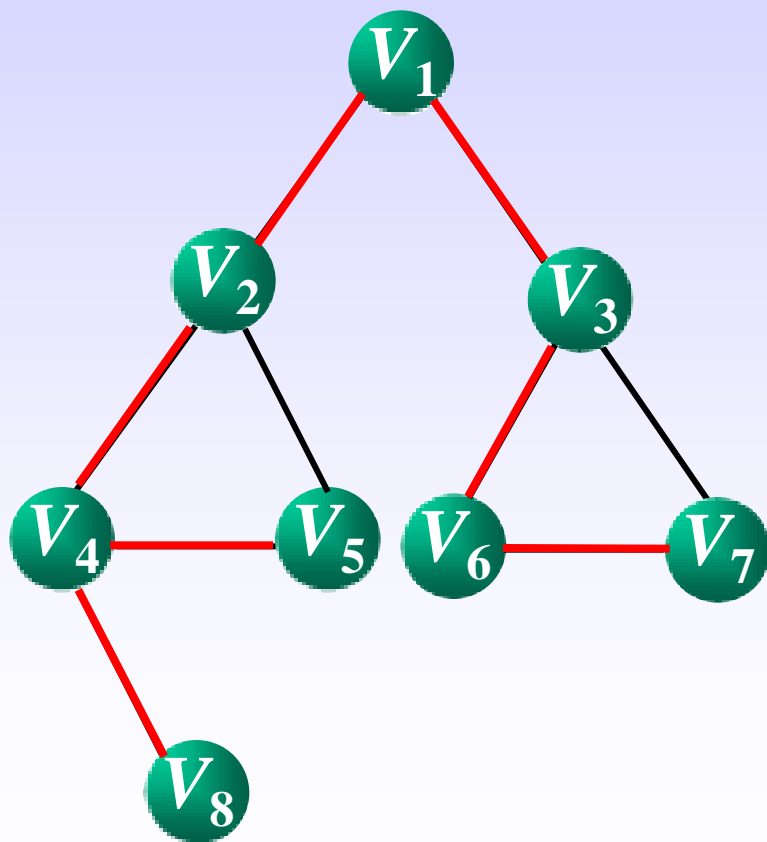
• $T(G)$ 和图 G 中所有顶点一起构成连通图 G 的极小连通子图, 按照定义 (生成树: n 个顶点的连通图 G 的生成树是包含 G 中全部顶点的一个极小连通子图。), 它是连通图的一棵生成树, 并且称

- 由深度优先搜索得到的为深度优先生成树;
- 由广度优先搜索得到的为广度优先生成树

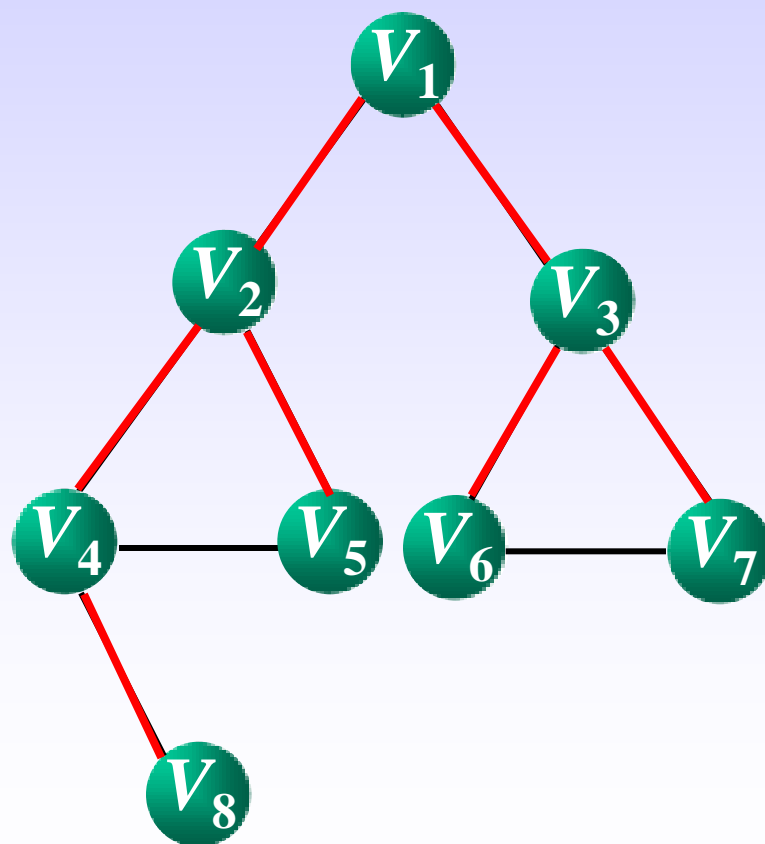
(生成树不唯一)

6.3 最小生成树

生成树



(a) 深度优先生成树



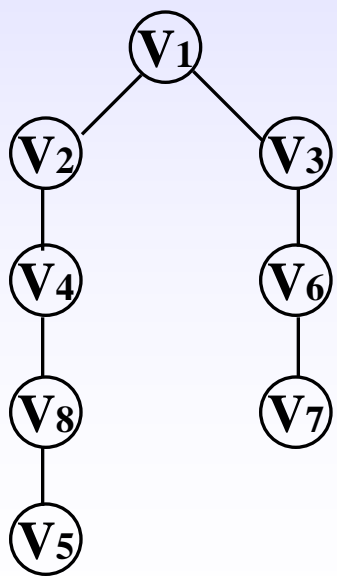
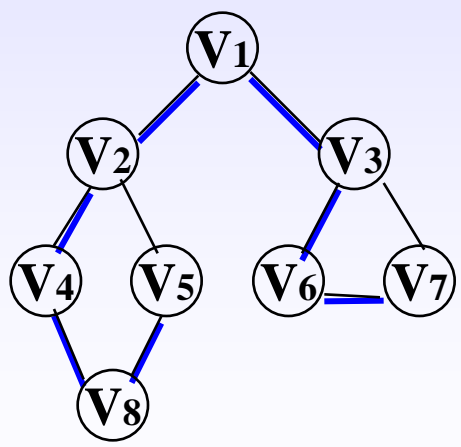
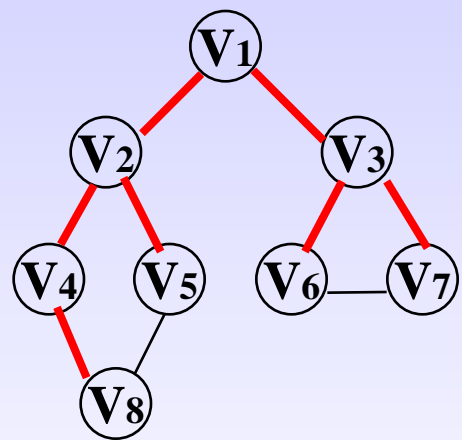
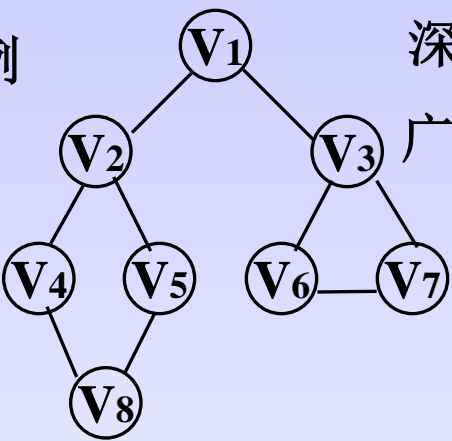
(b) 广度优先生成树

6.3 最小生成树

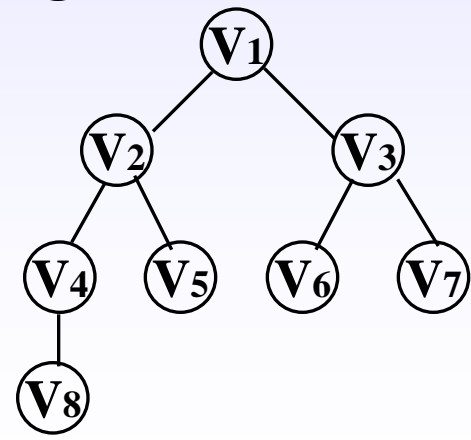
例

深度优先遍历: $V1 \Rightarrow V2 \Rightarrow V4 \Rightarrow V8 \Rightarrow V5 \Rightarrow V3 \Rightarrow V6 \Rightarrow V7$

广度优先遍历: $V1 \Rightarrow V2 \Rightarrow V3 \Rightarrow V4 \Rightarrow V5 \Rightarrow V6 \Rightarrow V7 \Rightarrow V8$



深度优先生成树



广度优先生成树

6.3 最小生成树

说明: 一个连通图

❖ 可以有**许多棵不同的**生成树

❖ 所有生成树具有以下**共同特点**:

- ♣ 生成树的顶点个数与图的顶点个数相同

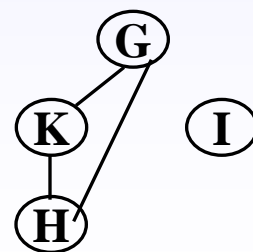
- ♣ 生成树是图的极小连通子图

- ♣ 一个有 n 个顶点的连通图的生成树有 $n-1$ 条边

- ♣ 生成树中任意**两个**顶点间的路径是**唯一**的

- ♣ 在生成树中再加一条边必然形成回路

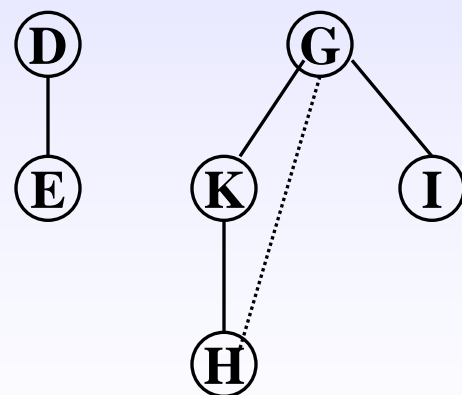
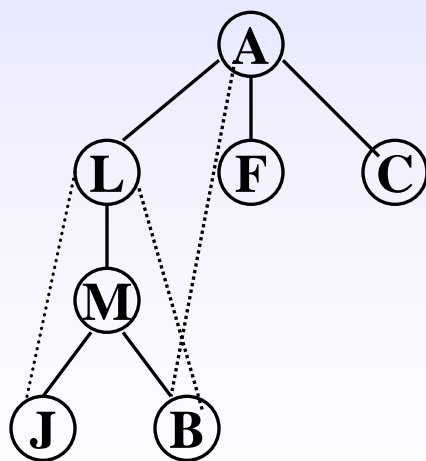
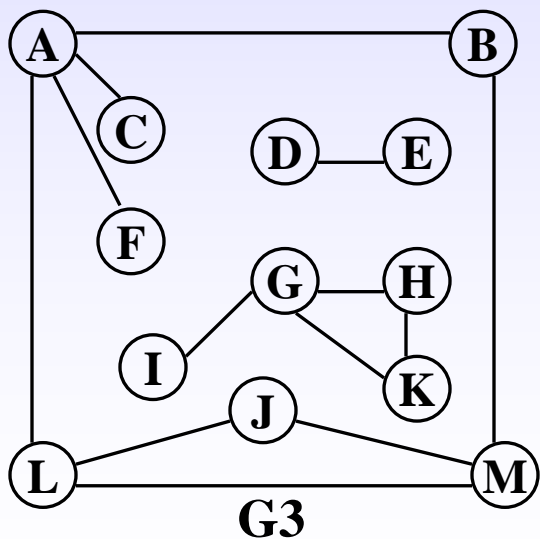
❖ 含 n 个顶点 $n-1$ 条边的图不一定是生成树



6.3 最小生成树

对于非连通图

每个连通分量中的顶点集和遍历时走过的边一起构成**若干棵生成树**，这些连通分量的生成树组成**非连通图的生成森林**。例如，图示为G3的深度优先生成森林，它由三棵深度优先生成树组成。



G3的深度优先生成森林

应用举例——最小生成树

最小生成树(Minimum Spanning Tree)

生成树的代价： 设 $G=(V, E)$ 是一个无向连通网，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**。

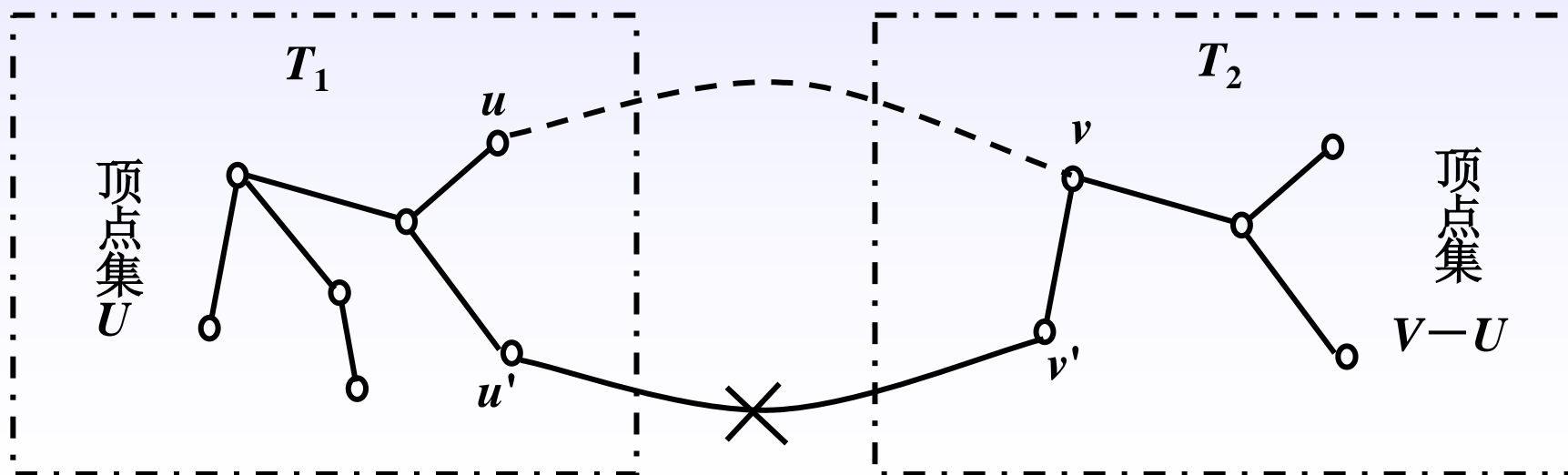
最小生成树的概念可以应用到许多实际问题中。

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？

应用举例——最小生成树

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。



应用举例——最小生成树

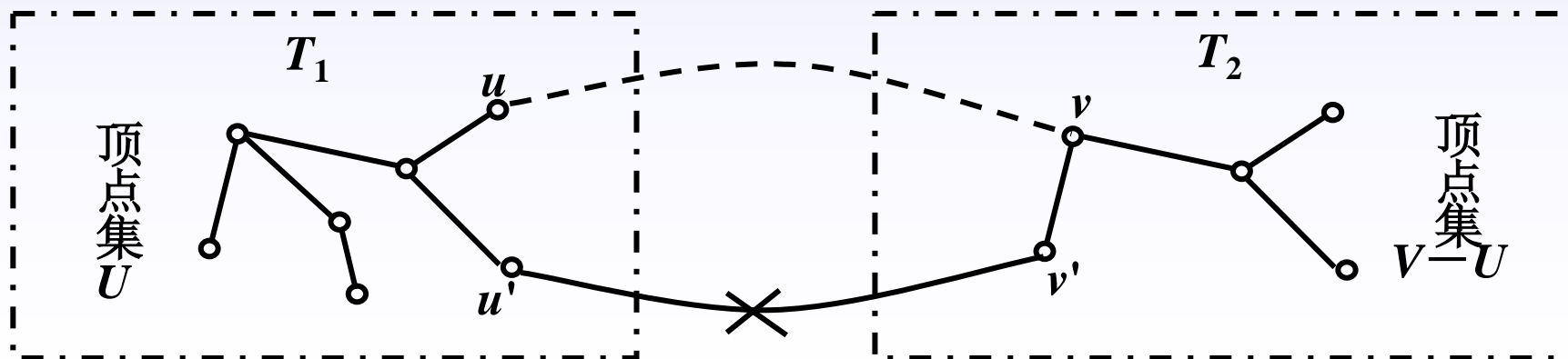
MST性质 用反证法证明:

假设网 N 的任何一棵最小生成树都不包含 (u, v) 。

设 T 是连通网上的一棵最小生成树,

当将边 (u, v) 加入到 T 中时, 由生成树的定义, T 中必存在一条包含 (u, v) 的回路。

另一方面, 由于 T 是生成树, 则在 T 上必存在另一条边 (u', v') , 其中 $u' \in U$, $v' \in V-U$, 且 u 和 u' 之间, v 和 v' 之间均有路径相通。删去边 (u', v') , 便可消除上述回路, 同时得到另一棵生成树 T' 。因为 (u, v) 的代价不高于 (u', v') , 则 T' 的代价亦不高于 T , T' 是包含 (u, v) 的一棵最小生成树。由此和假设矛盾。



应用举例——最小生成树

两个利用MST性质构造最小生成树的算法:

普里姆(Prim)算法

克鲁斯卡尔(Kruskal)算法

应用举例——最小生成树

普里姆 (Prim) 算法

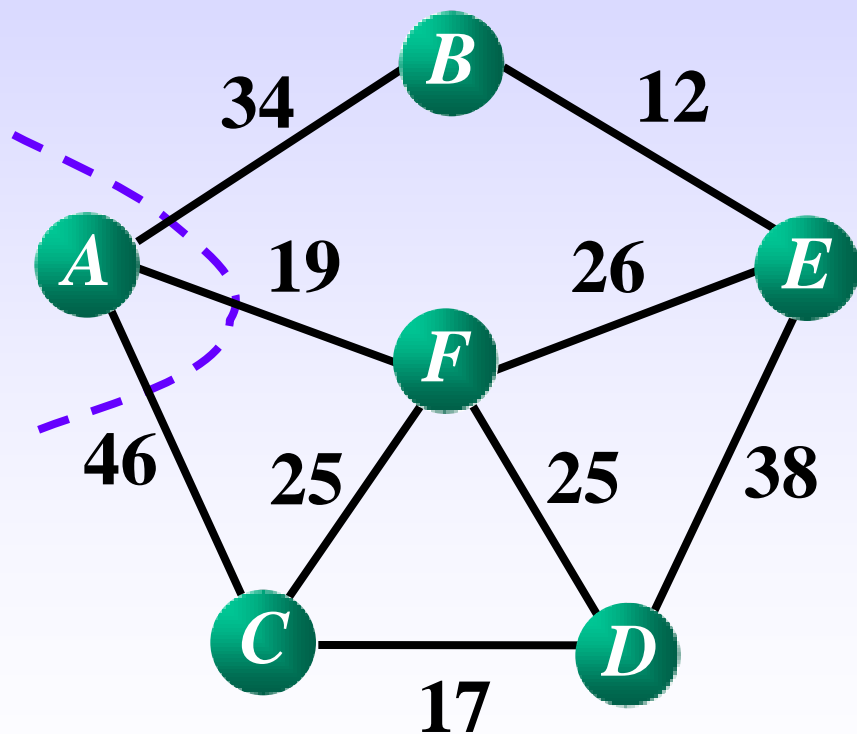
基本思想： 设 $G=(V, E)$ 是具有 n 个顶点的连通网， $T=(U, TE)$ 是 G 的最小生成树， T 的**初始状态**为 $U=\{u_0\}$ ($u_0 \in V$)， $TE=\{ \}$ ，重复执行下述操作：在所有 $u \in U$ ， $v \in V-U$ 的边中找一条代价最小的边 (u, v) 并入集合 TE ，同时 v 并入 U ，直至 $U=V$ 。

关键：是如何找到连接 U 和 $V-U$ 的最短边。

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。

应用举例——最小生成树

Prim算法



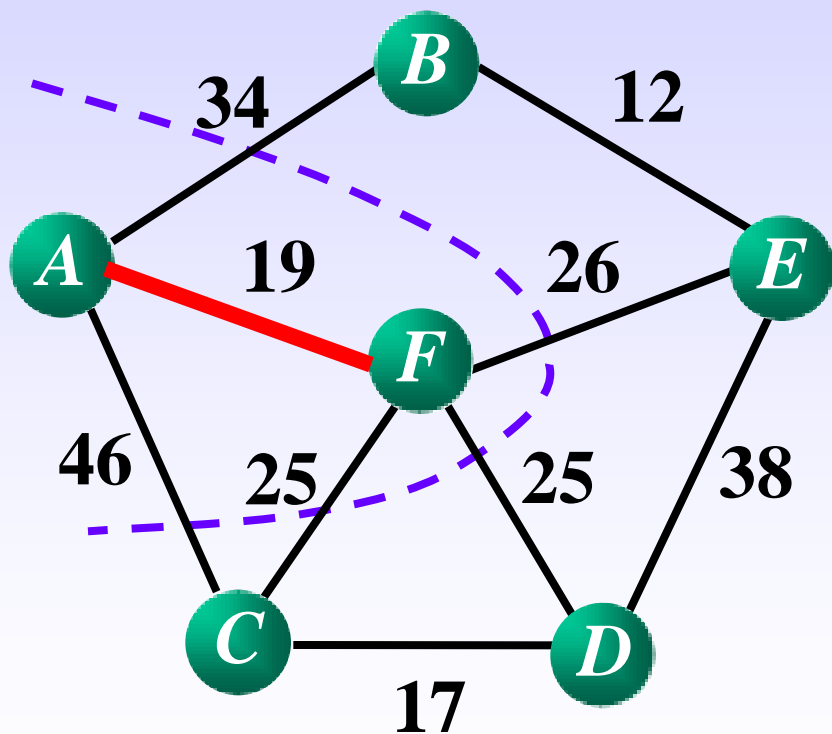
$U = \{A\}$

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

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

应用举例——最小生成树

Prim算法



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

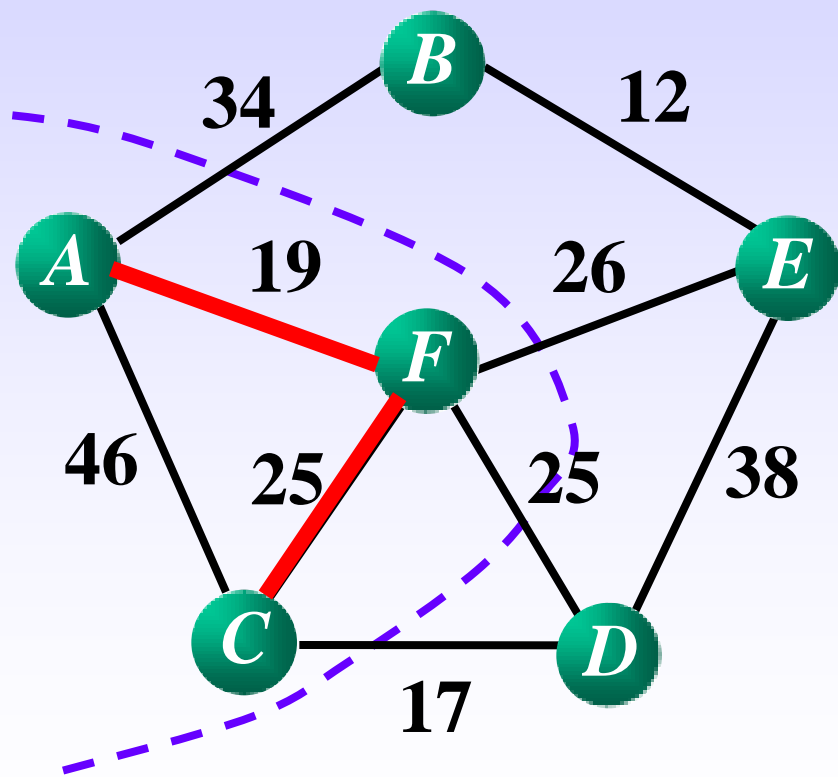
$U=\{A, F\}$

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

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

应用举例——最小生成树

Prim算法



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

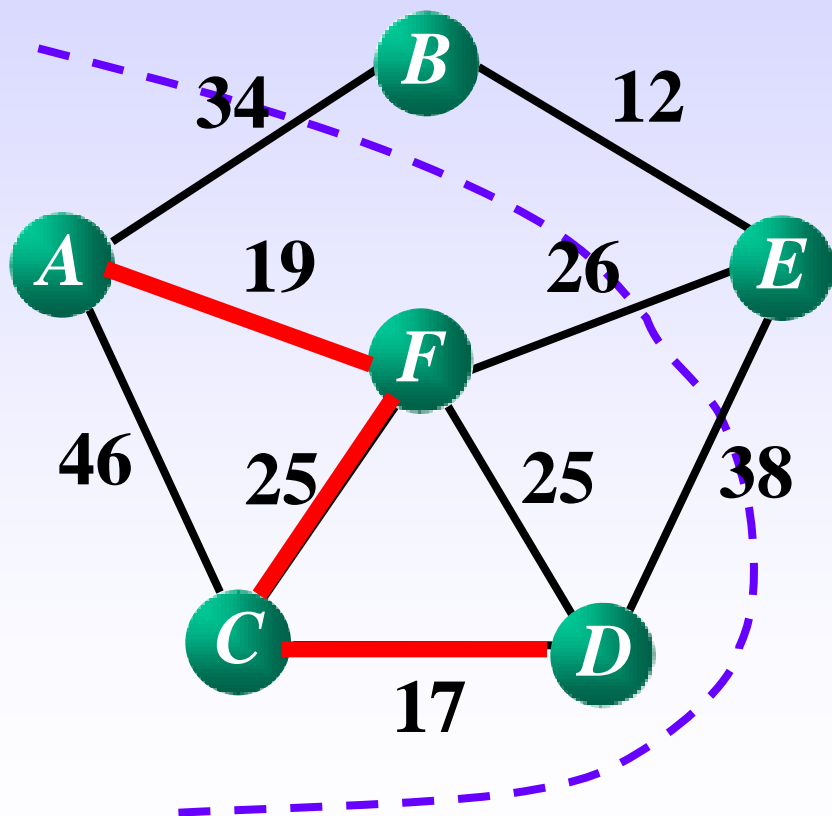
$U=\{A, F, C\}$

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

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

应用举例——最小生成树

Prim算法



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

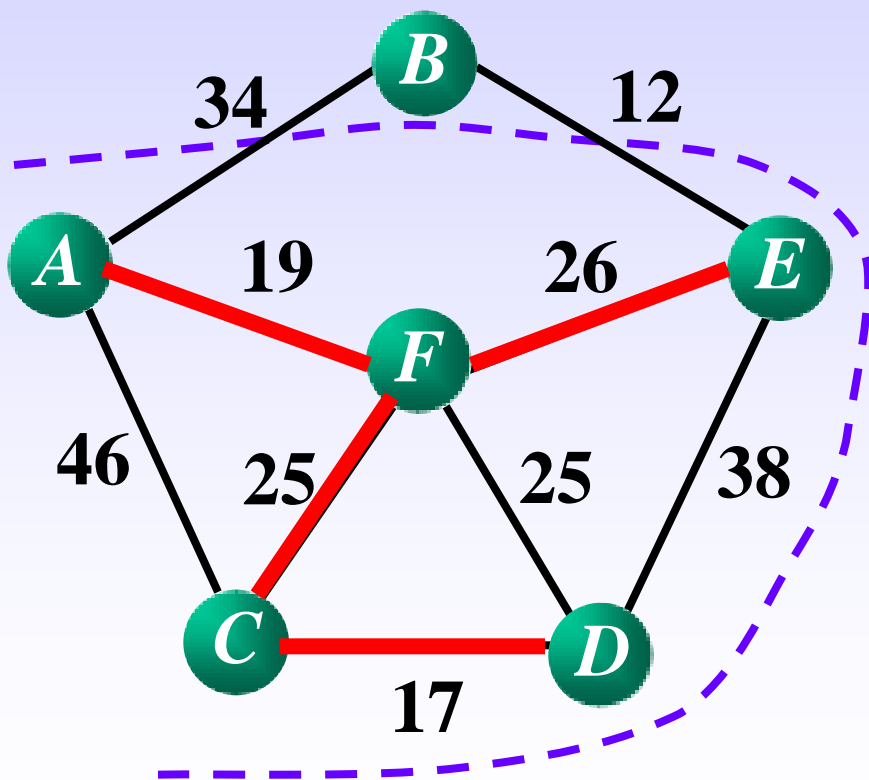
$U=\{A, F, C, D\}$

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

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

应用举例——最小生成树

Prim算法



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

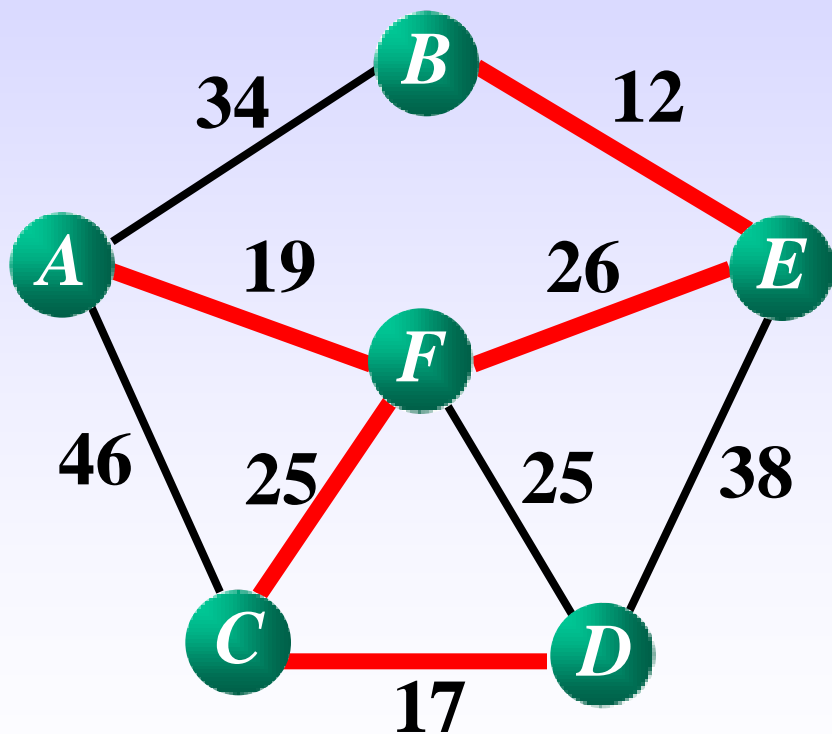
$U=\{A, F, C, D, E\}$

$V-U=\{B\}$

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

应用举例——最小生成树

Prim算法



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

$V - U = \{ \}$

应用举例——最小生成树

数据结构设计

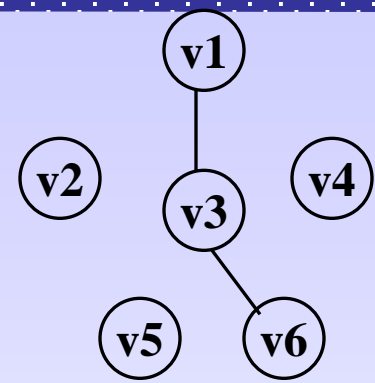
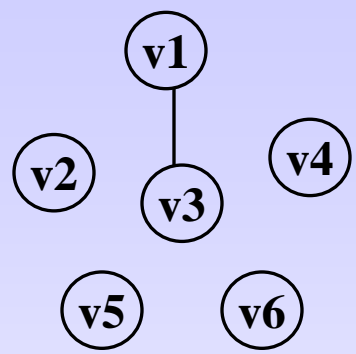
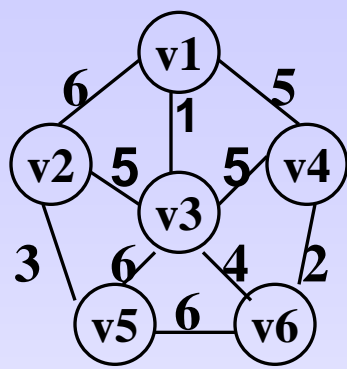
数组 **lowcost[n]**: 用来保存集合 $V-U$ 中各顶点与集合 U 中顶点最短边的权值, $\text{lowcost}[v]=0$ 表示顶点 v 已加入最小生成树中;

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 $v_k \in U$

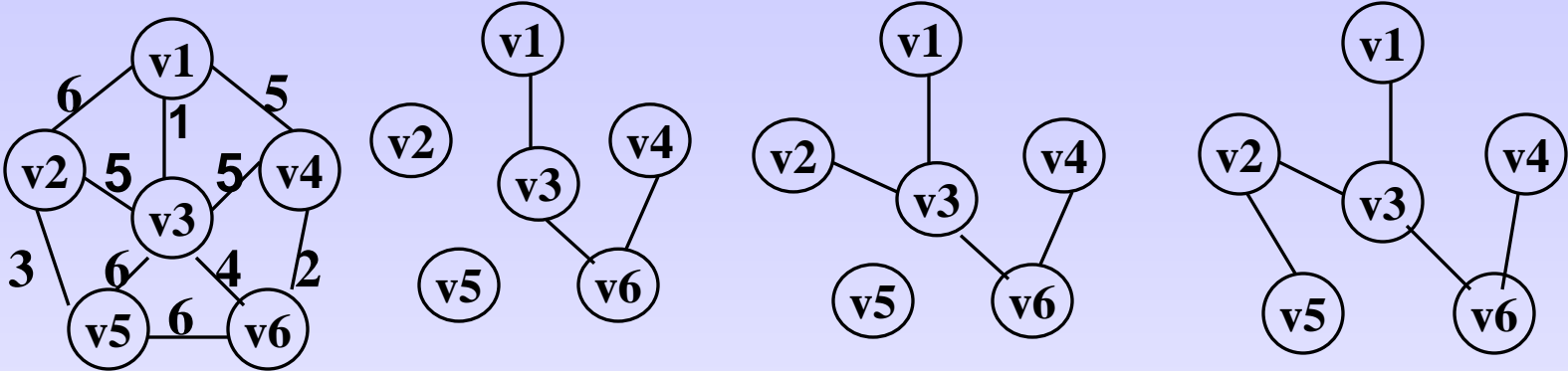
应用举例——最小生成树



<div>i \ shortEdge</div>	1	2	3	4	5	U	V-U	k
	v2	v3	v4	v5	v6			
Adjvex lowcost	v1 6	v1 1	v1 5			{v1}	{v2,v3,v4, v5,v6}	2
Adjvex lowcost	v3 5		v1 5	v3 6	v3 4	{v1,v3}	{v2,v4,v5, v6}	5
Adjvex lowcost	v3 5		v6 2	v3 6		{v1,v3,v6}	{v2,v4,v5}	3

普里姆算法构造最小生成树的过程

应用举例——最小生成树



<div>i \ shortEdge</div>	1	2	3	4	5	U	V-U	k
	v2	v3	v4	v5	v6			
Adjvex	v3			v3		{v1,v3,v6,v4}	{v2,v5}	1
lowcost	5	0	0	6	0			
Adjvex				v2		{v1,v3,v6,v4,v2}	{v5}	4
lowcost	0	0	0	3	0			
Adjvex						{v1,v3,v6,v4,v2,v5}	{ }	
lowcost	0	0	0	0	0			

普里姆算法构造最小生成树的过程 (续)

应用举例——最小生成树

Prim算法——伪代码

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;

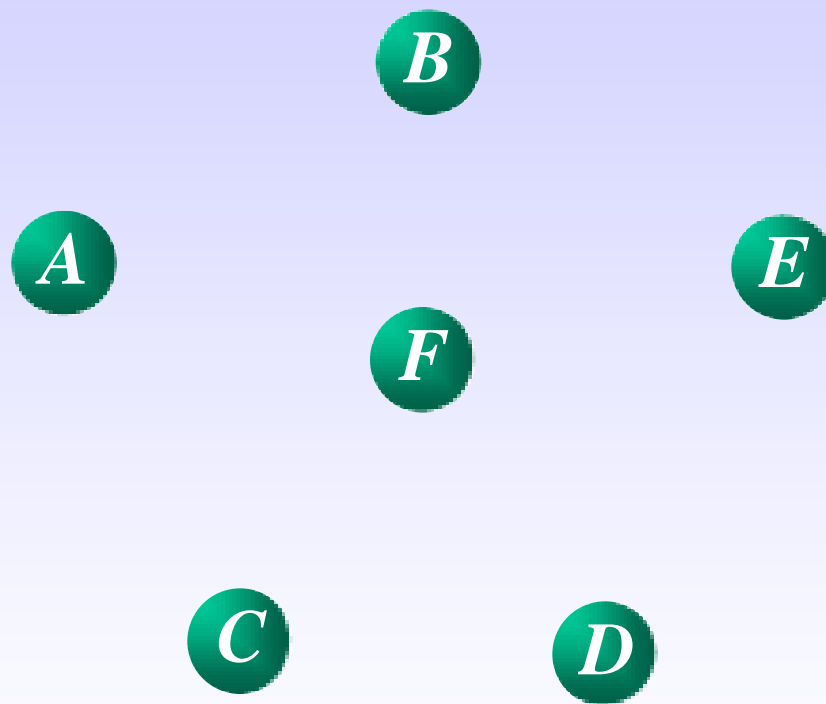
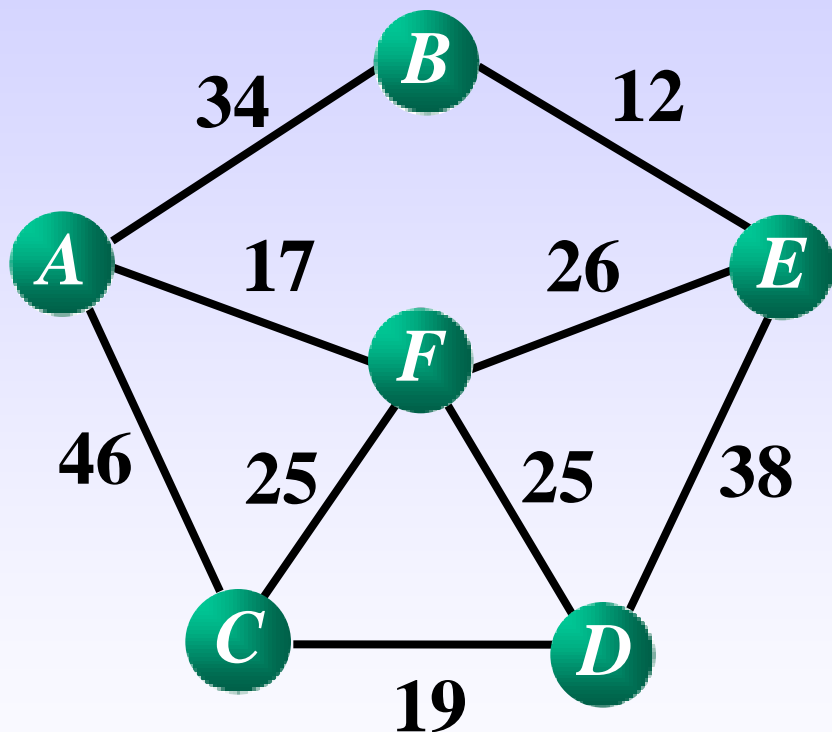
应用举例——最小生成树

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 $U=V$ ， $TE=\{ \}$ ，图 G 中每个顶点自成一个连通分量。按照边的权值由小到大的顺序，在 G 的边集 E 中选择代价最小的边，

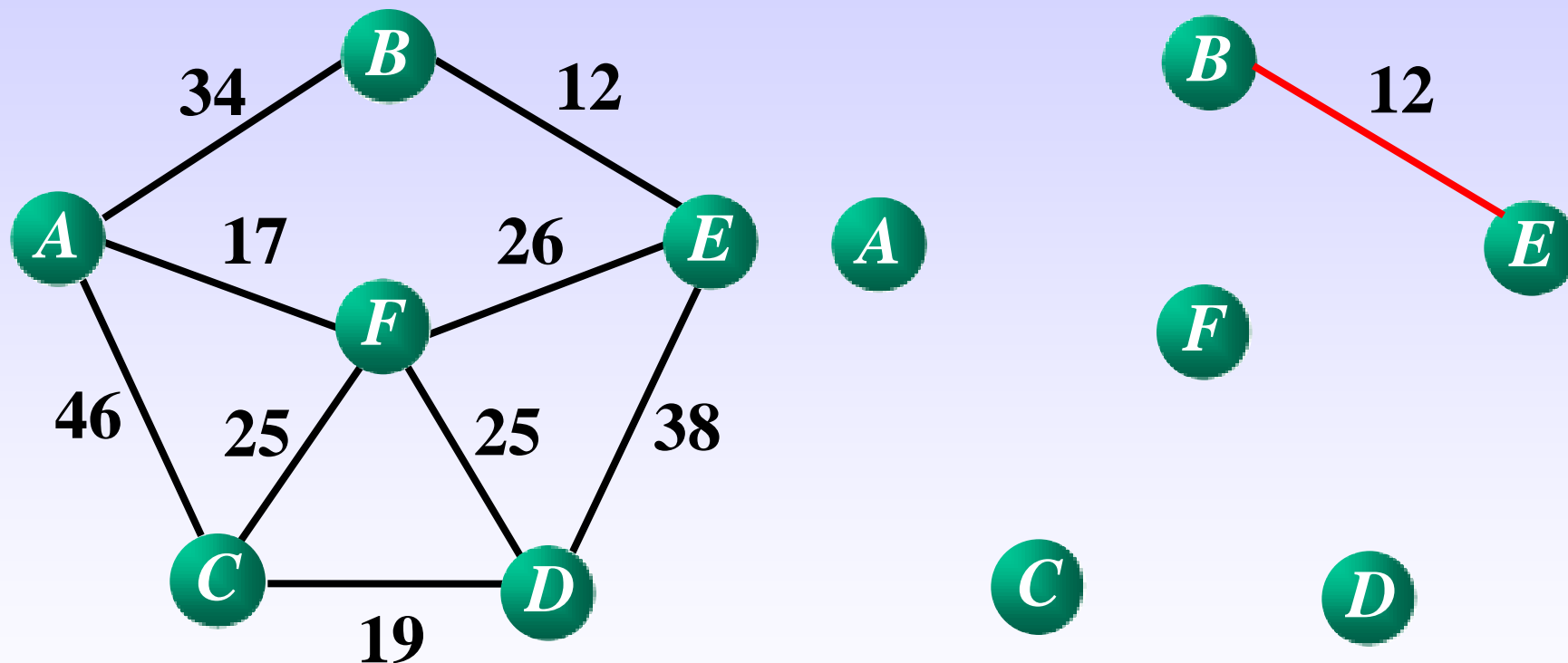
- 若该边依附的顶点属于 T 中不同的连通分量上，则将此边作为最小生成树的边加入到 T 中，
- 否则舍去此边而选择下一条代价最小的边。
- 依此类推，直至所有顶点都在同一连通分量上。此连通分量便为 G 的一棵最小生成树。

应用举例——最小生成树



连通分量 = {A}, {B}, {C}, {D}, {E}, {F}

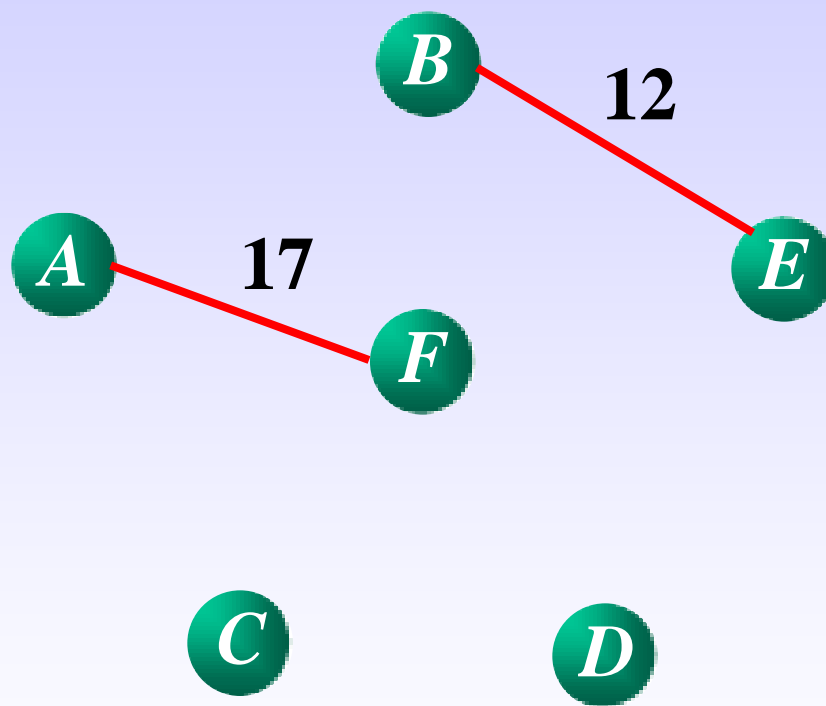
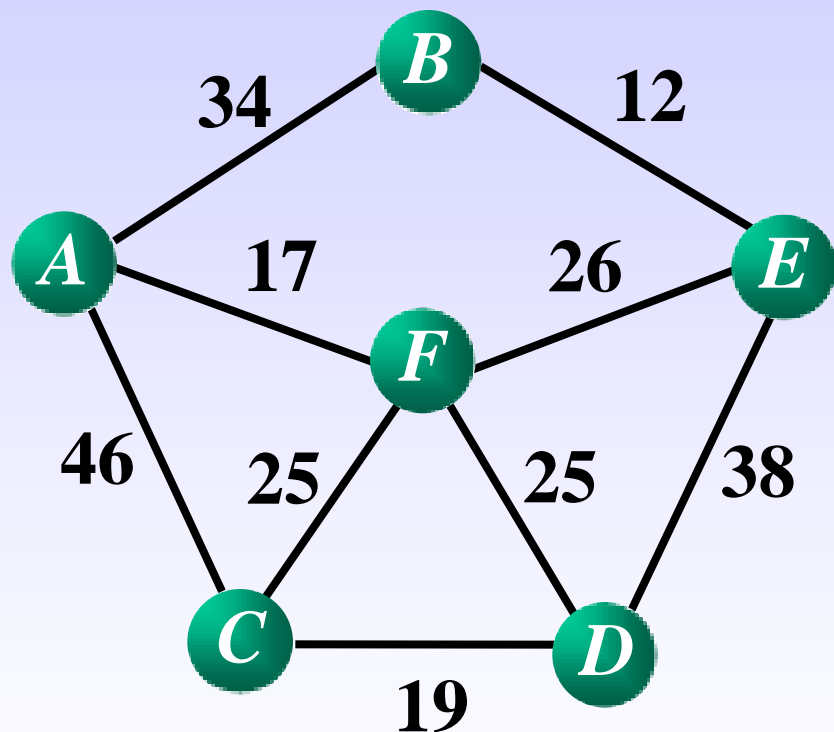
应用举例——最小生成树



连通分量 = {A}, {B}, {C}, {D}, {E}, {F}

连通分量 = {A}, {B, E}, {C}, {D}, {F}

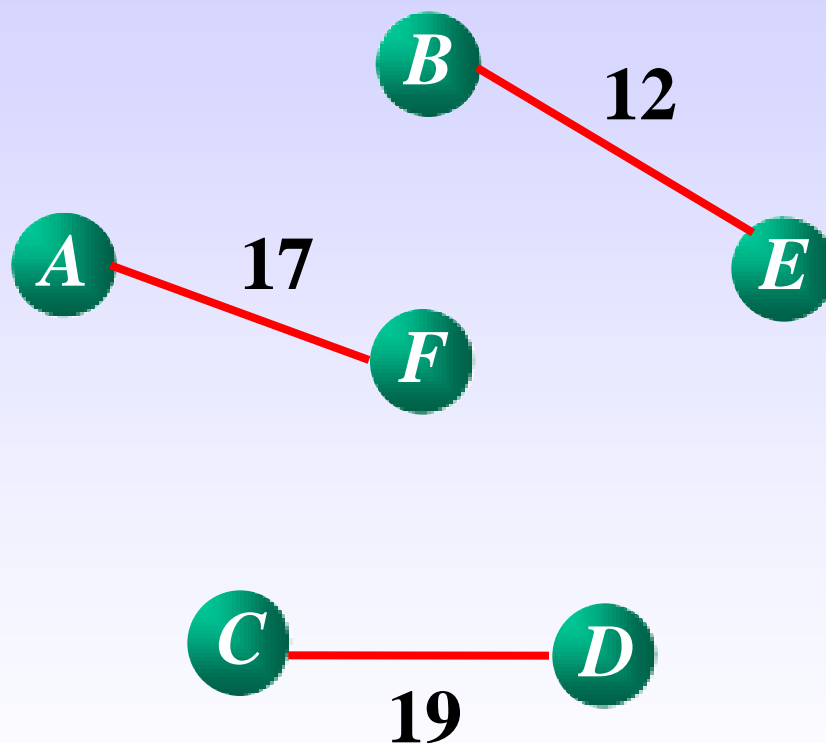
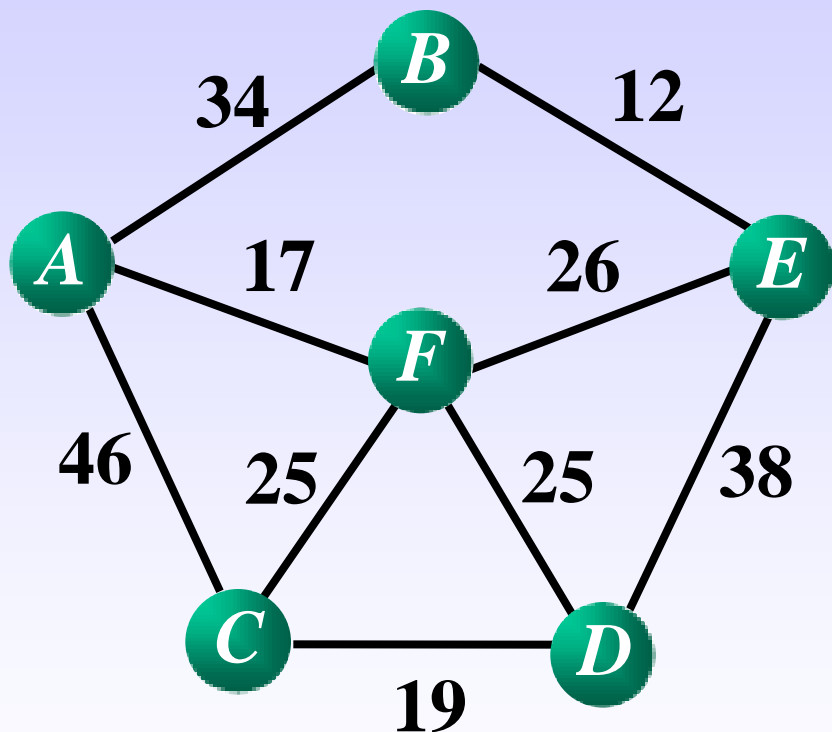
应用举例——最小生成树



连通分量 = {A}, {B, E}, {C}, {D}, {F}

连通分量 = {A, F}, {B, E}, {C}, {D}

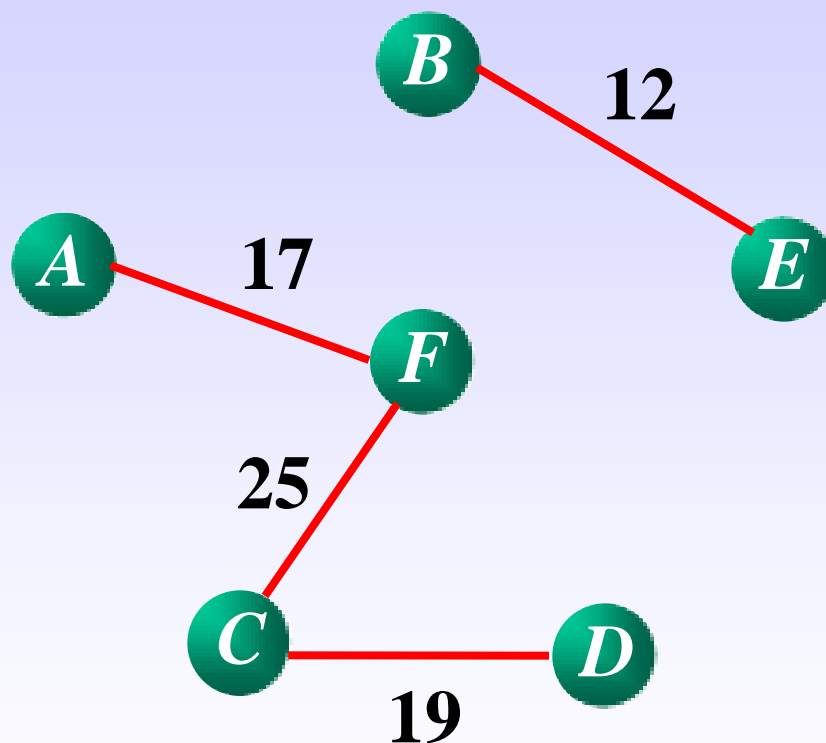
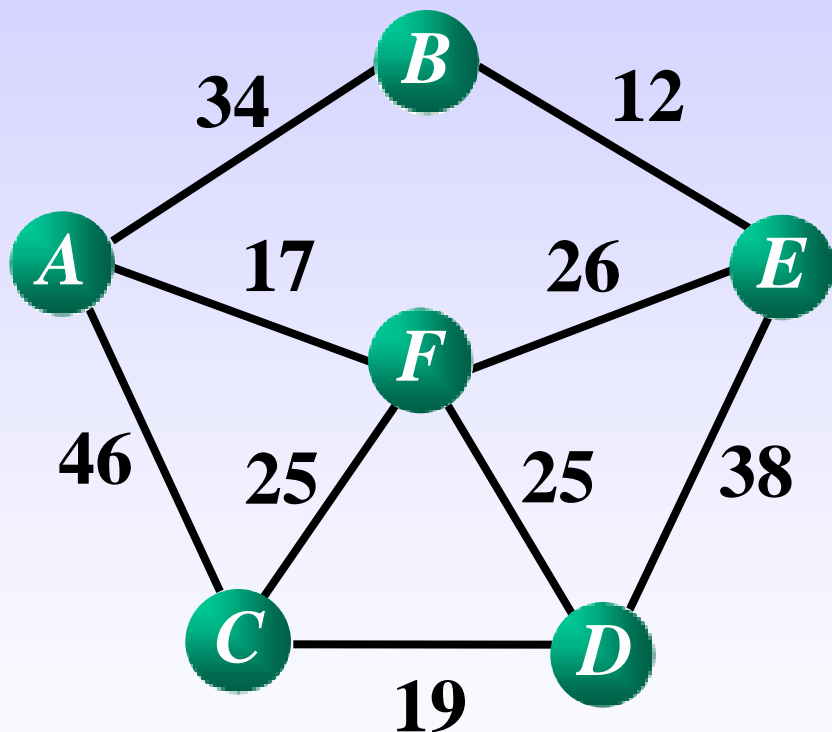
应用举例——最小生成树



连通分量 = {A, F}, {B, E}, {C}, {D}

连通分量 = {A, F}, {B, E}, {C, D}

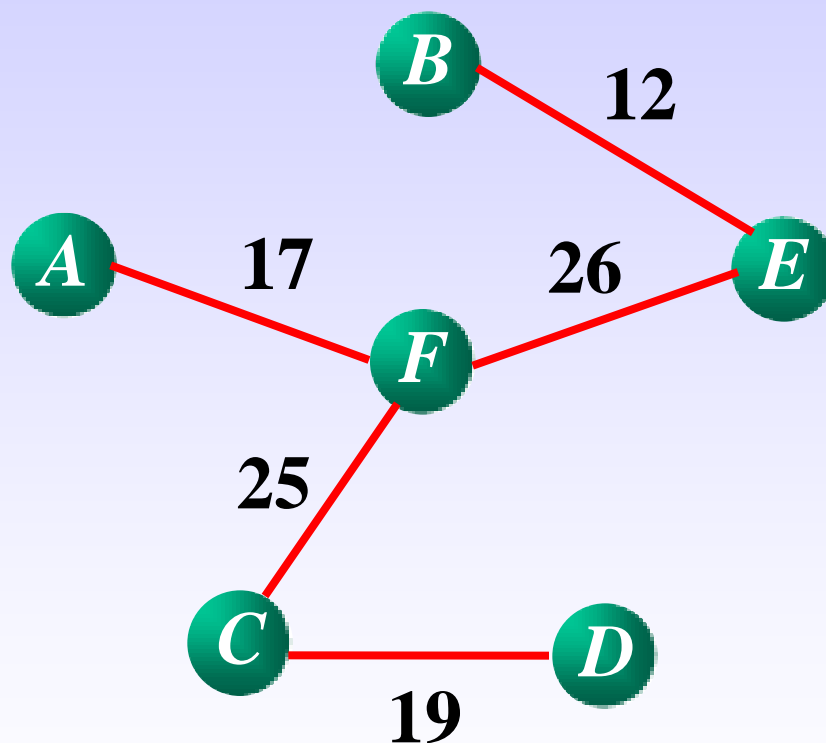
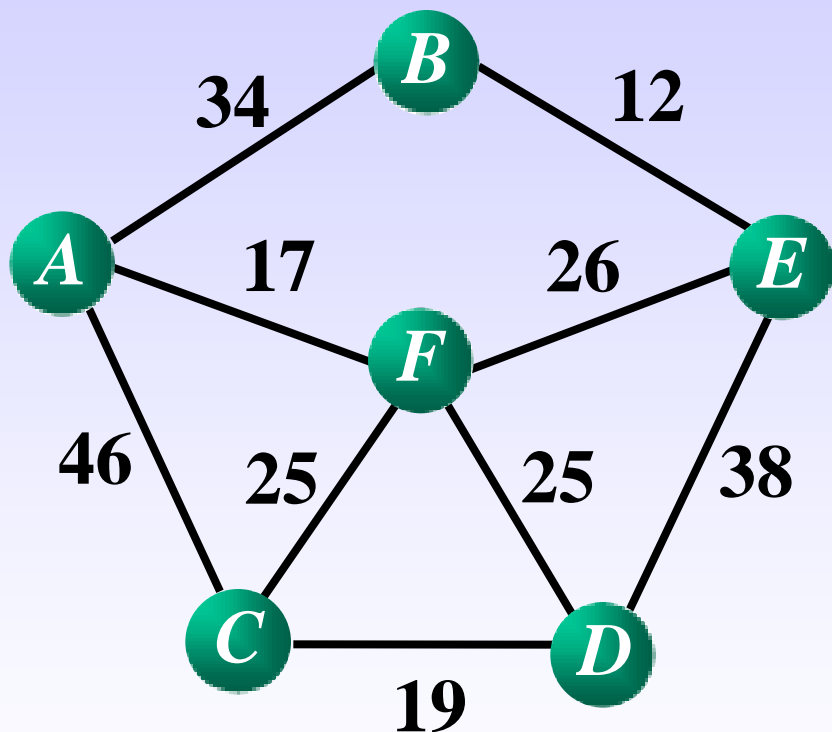
应用举例——最小生成树



连通分量 = {A, F}, {B, E}, {C, D}

连通分量 = {A, F, C, D}, {B, E}

应用举例——最小生成树

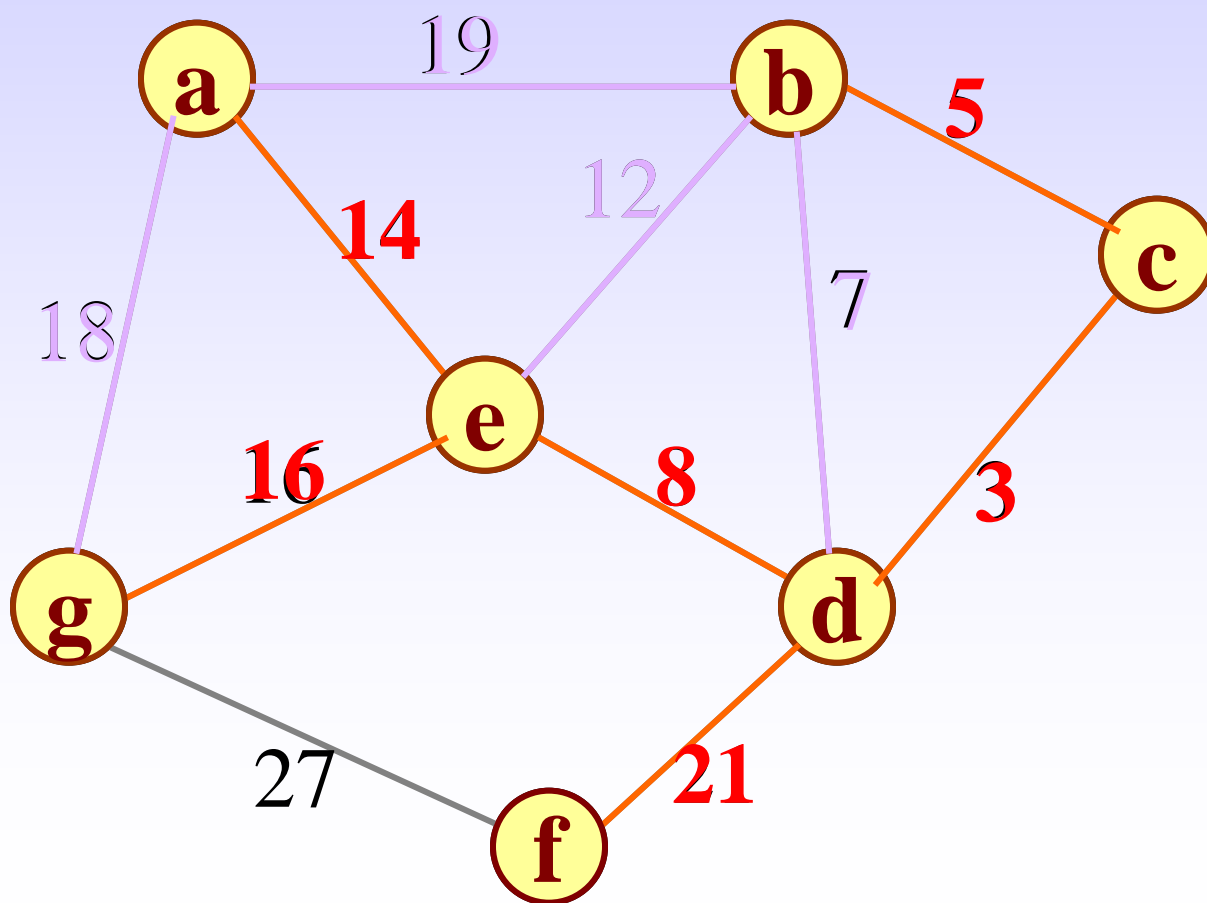


连通分量 = {A, F, C, D}, {B, E}

连通分量 = {A, F, C, D, B, E}

应用举例——最小生成树

Kruskal算法构造最小生成树



应用举例——最小生成树

Kruskal算法——伪代码

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;

应用举例——最小生成树

比较两种算法

算法名	普里姆算法	克鲁斯卡尔算法
-----	-------	---------

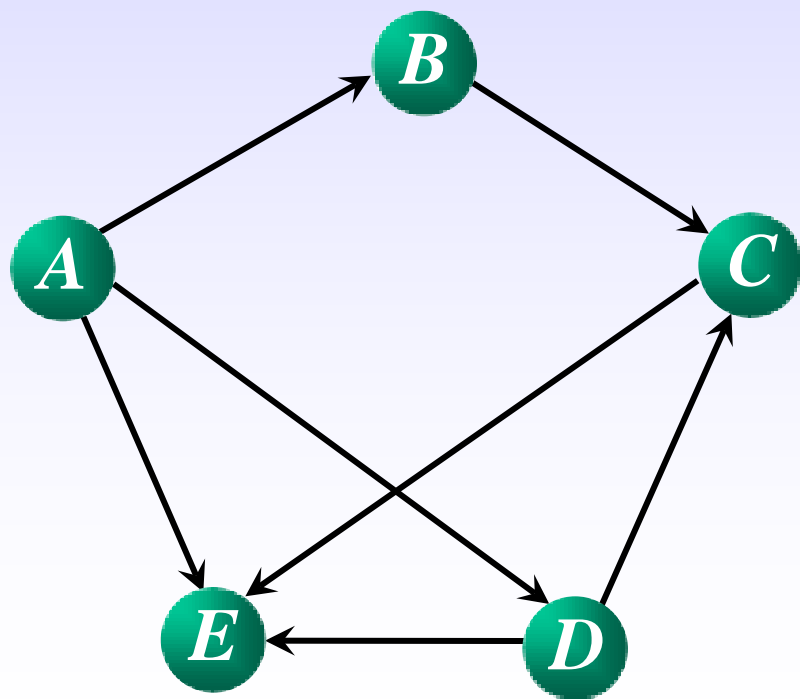
时间复杂度	$O(n^2)$	$O(e \log_2 e)$
-------	----------	-----------------

适应范围	稠密图	稀疏图
------	-----	-----

6.4 最短路径

最短路径

在非网图中，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

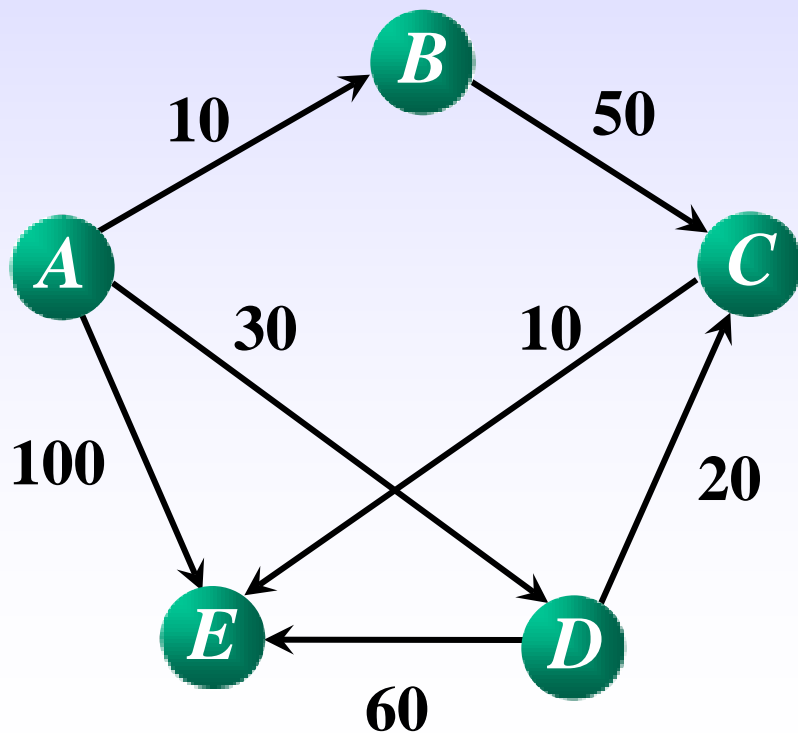
ADCE: 3

ABCE: 3

6.4 最短路径

最短路径

在网图中，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

6.4 最短路径

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。

6.4 最短路径

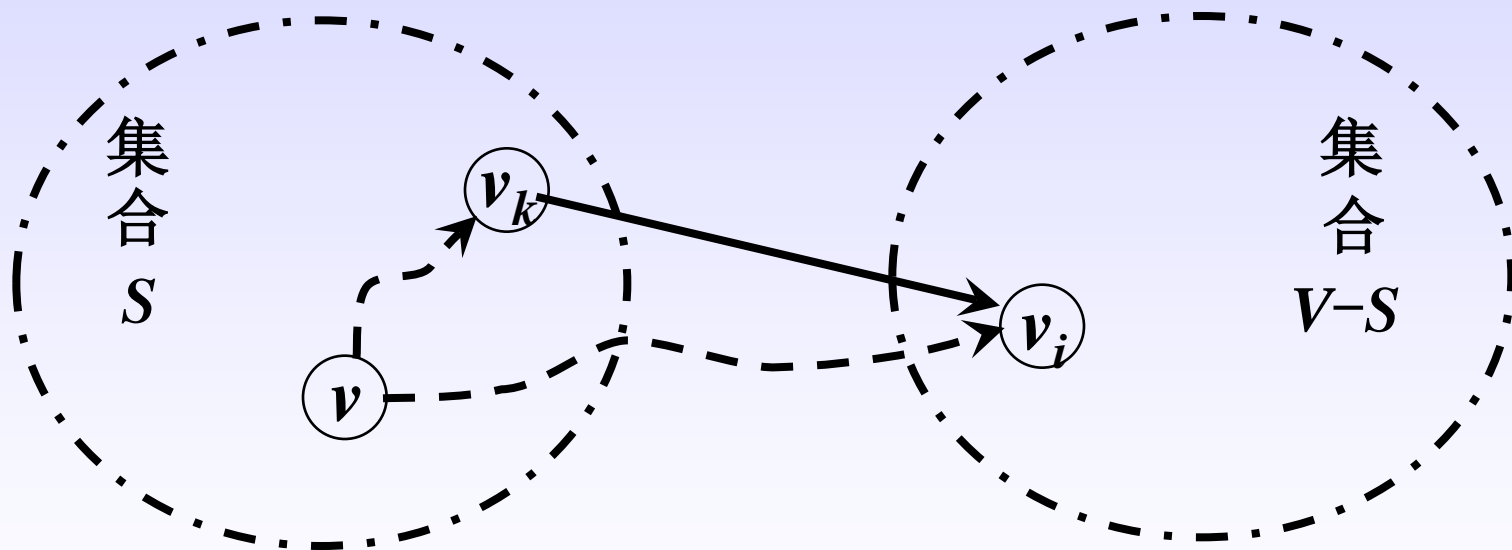
Dijkstra算法

基本思想： 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 。

- 初使时令 $S=\{v\}$, $T=\{\text{其余顶点}\}$ ， T 中顶点对应的距离值：
 - 若存在 $\langle v, v_i \rangle$ ，为 $\langle v, v_i \rangle$ 弧上的权值
 - 若不存在 $\langle v, v_i \rangle$ ，为 ∞
- 从 T 中选取一个其距离值为最小的顶点 v_k ，加入 S
- 对 T 中顶点的距离值进行修改：若加进 v_k 作中间顶点，从 v 到 v_i 的距离值比不加 v_k 的路径要短，则修改此距离值
- 重复上述步骤，直到 S 中包含所有顶点，即 $S=V$ 为止。

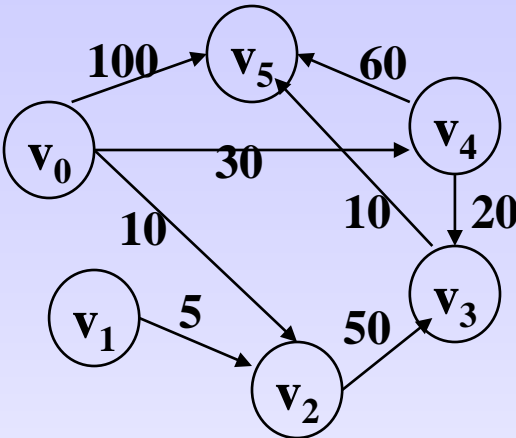
6.4 最短路径

Dijkstra算法的基本思想



6.4 最短路径

迪杰斯特拉
算法示例：



∞	∞	10	∞	30	100
∞	∞	5	∞	∞	∞
∞	50	∞	∞		
∞	∞	∞	∞	∞	10
∞	∞	∞	20	∞	60
∞	∞	∞	∞	∞	∞

终点	从v ₀ 到各终点的距离值和最短路径的求解过程				
v ₁	∞	∞	∞	∞	∞
v ₂	10, (v ₀ ,v ₂)				
v ₃	∞	60, (v ₀ ,v ₂ ,v ₃)	50, (v ₀ ,v ₄ ,v ₃)		
v ₄	30, (v ₀ ,v ₄)	30, (v ₀ ,v ₄)			
v ₅	100,(v ₀ ,v ₅)	100,(v ₀ ,v ₅)	90, (v ₀ ,v ₄ ,v ₅)	60, (v ₀ ,v ₄ ,v ₃ ,v ₅)	
v _k	v ₂	v ₄	v ₃	v ₅	
S	{v ₀ ,v ₂ }	{v ₀ ,v ₂ ,v ₄ }	{v ₀ ,v ₂ ,v ₃ ,v ₄ }	{v ₀ ,v ₂ ,v ₃ ,v ₄ ,v ₅ }	

6.4 最短路径

设计数据结构：

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 。

6.4 最短路径

Dijkstra算法——伪代码

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[j]和path[j];
 - 2.3 修改数组dist和path;
 - 2.4 将顶点 v_k 添加到数组s中;

6.4 最短路径

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。

6.4 最短路径

Floyd算法基本思想：假设求从顶点 v_i 到 v_j 的最短路径。

如果从 v_i 到 v_j 有弧，则从 v_i 到 v_j 存在一条长度为 $\text{arcs}[i][j]$ 的路径，该路径不一定是最短路径，尚需进行 n 次试探：

在路径上增加一个顶点 v_0 ，考虑路径 (v_i, v_0, v_j) 是否存在(即判别弧 (v_i, v_0) 和 (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,取长度较短者为从 v_i 到 v_j 的中间顶点的序号不大于 0 的最短路径。

在路径上增加一个顶点 v_1 ，如果 (v_i, \dots, v_1) 和 (v_1, \dots, v_j) 分别是当前找到的中间顶点的序号不大于 0 的最短路径，那么 $(v_i, \dots, v_1, \dots, v_j)$ 就有可能是从 v_i 到 v_j 的中间顶点的序号不大于 1 的最短路径。

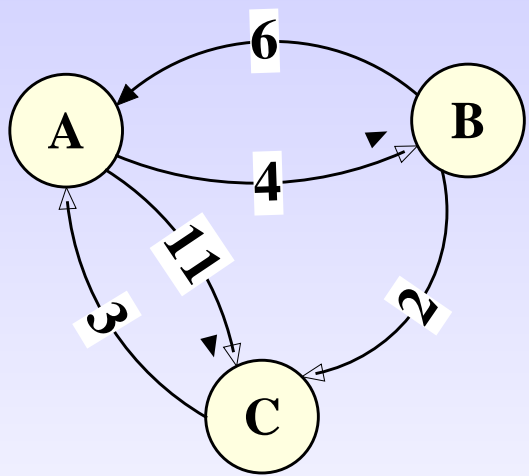
将它和已经得到的从 v_i 到 v_j 中间顶点序号不大于 0 的最短路径相比较，从中选出中间顶点的序号不大于 1 的最短路径

在路径上再增加一个顶点 v_2 ，继续进行试探。

依次类推，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。

6.4 最短路径

Floyd算法例



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

路径:

	AB	AC
BA		BC
CA		

加入A:
$$\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

路径:

	AB	AC
BA		BC
CA	CAB	

加入B:
$$\begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

路径:

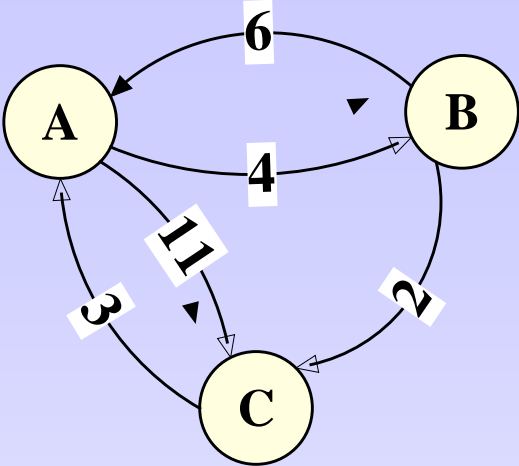
	AB	ABC
BA		BC
CA	CAB	

加入C:
$$\begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

路径:

	AB	ABC
BCA		BC
CA	CAB	

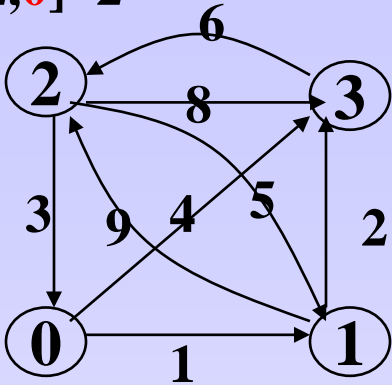
Floyd算法求解过程



	D ⁻¹				D ⁰				D ¹				D ²		
	0	1	2		0	1	2		0	1	2		0	1	2
0	0	4	11	0	0	4	11	0	0	4	6	0	0	4	6
1	6	0	2	1	6	0	2	1	6	0	2	1	5	0	2
2	3	∞	0	2	3	7	0	2	3	7	0	2	3	7	0
	Path ⁻¹				Path ⁰				Path ¹				Path ²		
	0	1	2		0	1	2		0	1	2		0	1	2
0	-1	0	0	0	-1	0	0	0	-1	0	1	0	-1	0	1
1	1	-1	1	1	1	-1	1	1	1	-1	1	1	2	-1	1
2	2	-1	-1	2	2	0	-1	2	2	0	-1	2	2	0	-1

Path⁽³⁾ [1,0]=3 Path⁽³⁾ [3,0]=2 Path⁽³⁾ [2,0]=2
1→3→2→0

Floyd算法求解过程



	0	1	2	3
0	0	1	∞	4
1	∞	0	9	2
2	3	5	0	8
3	∞	∞	6	0

	D ⁽⁻¹⁾				D ⁽⁰⁾				D ⁽¹⁾				D ⁽²⁾				D ⁽³⁾			
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	0	1	∞	4	0	1	∞	4	0	1	10	3	0	1	10	3	0	1	9	3
1	∞	0	9	2	∞	0	9	2	∞	0	9	2	12	0	9	2	11	0	8	2
2	3	5	0	8	3	4	0	7	3	4	0	6	3	4	0	6	3	4	0	6
3	∞	∞	6	0	∞	∞	6	0	∞	∞	6	0	9	10	6	0	9	10	6	0
	Path ⁽⁻¹⁾				Path ⁽⁰⁾				Path ⁽¹⁾				Path ⁽²⁾				Path ⁽³⁾			
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	-1	0	-1	0	-1	0	-1	0	-1	0	1	1	-1	0	1	1	-1	0	3	1
1	-1	-1	1	1	-1	-1	1	1	-1	-1	1	1	2	-1	1	1	3	-1	3	1
2	2	2	-1	2	2	0	-1	0	2	0	-1	1	2	0	-1	1	2	0	-1	1
3	-1	-1	3	-1	-1	-1	3	-1	-1	-1	3	-1	2	2	3	-1	2	2	3	-1

6.4 最短路径

设计数据结构

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 $\text{path}[i][j] = "v_i v_j"$ 。

6.4 最短路径

Floyd算法——C++描述

```
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]="";
        }
}
```

6.4 最短路径

Floyd算法——C++描述

```
for (k=0; k<G.vertexNum; k++)  
    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];  
            }  
}
```

6.5 有向无环图及其应用

AOV网

① 什么是工程？工程有什么共性？

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**，简称**AOV网**。

② AOV网中出现回路意味着什么？

6.5 有向无环图及其应用

AOV网

① 什么是工程？工程有什么共性？

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，简称AOV网。

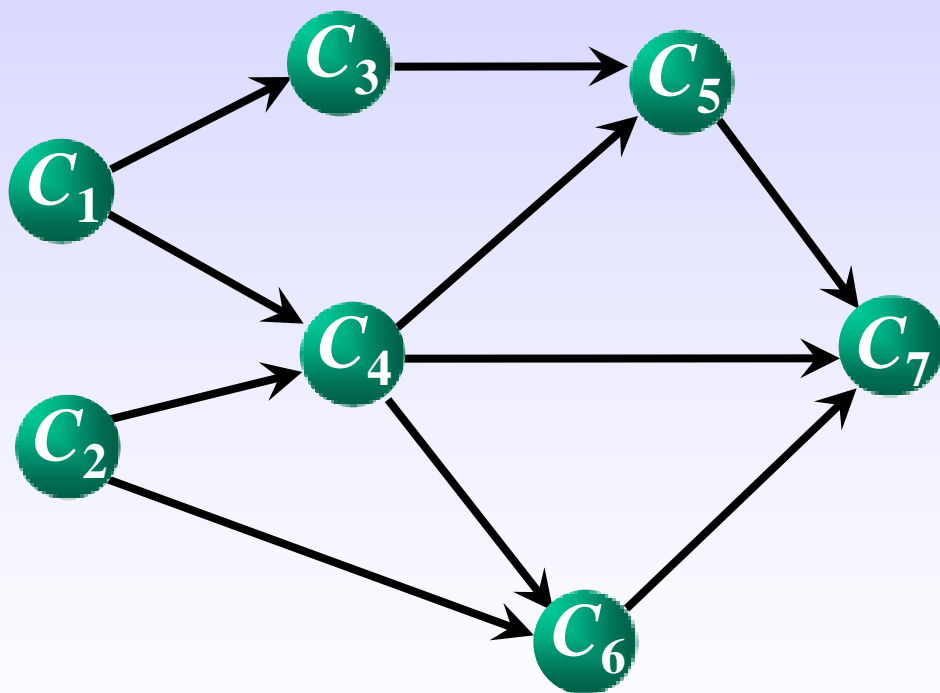
AOV网特点：

- 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。

6.5 有向无环图及其应用

AOV网

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



6.5 有向无环图及其应用

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。

拓扑排序： 对一个有向图构造拓扑序列的过程称为拓扑排序。

拓扑序列使得AOV网中所有应存在的前驱和后继关系都能得到满足。

6.5 有向无环图及其应用

拓扑排序

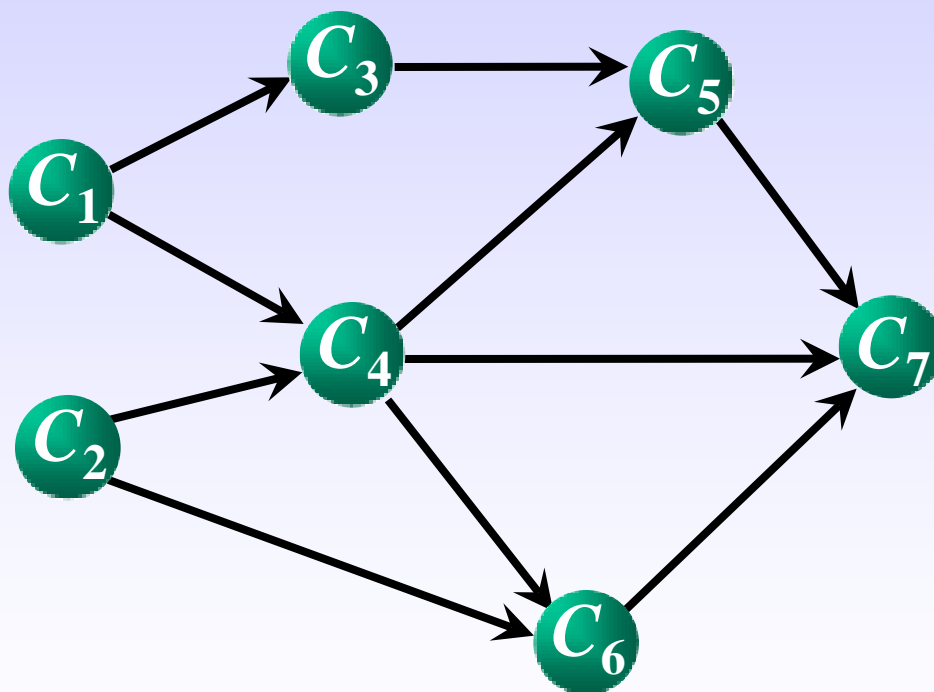
基本思想:

- (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?

6.5 有向无环图及其应用

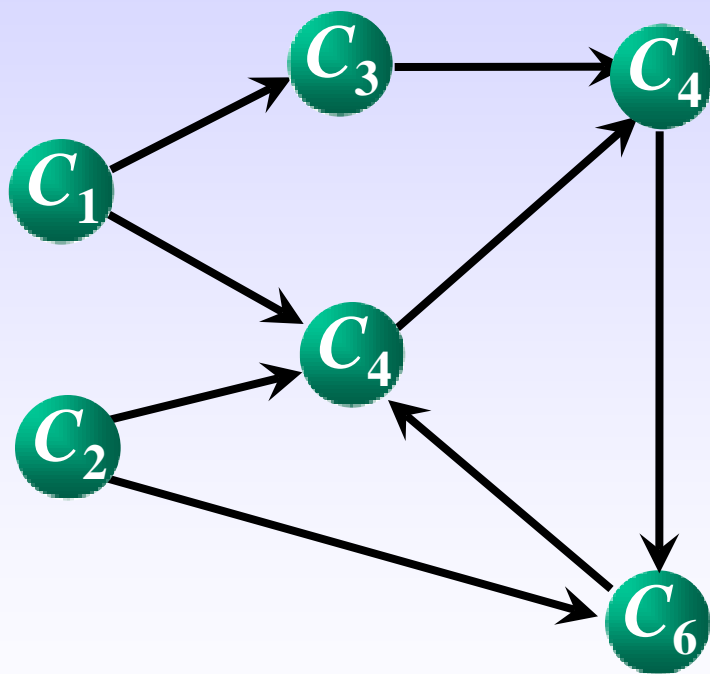
拓扑排序



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

6.5 有向无环图及其应用

拓扑排序



说明AOV网中
存在回路。

拓扑序列: $C_1, C_2, C_3,$

6.5 有向无环图及其应用

设计数据结构

1. 图的存储结构：采用邻接表存储，在顶点表中增加一个入度域。

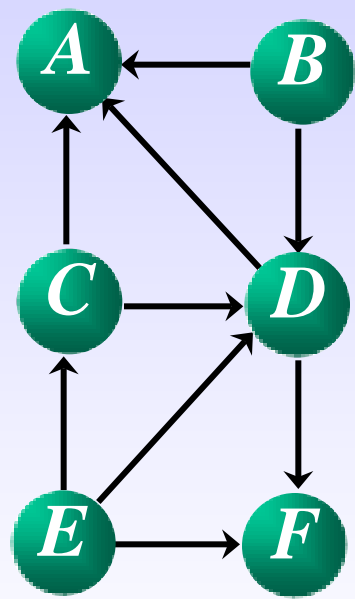
in	vertex	firstedge
-----------	---------------	------------------

顶点表结点

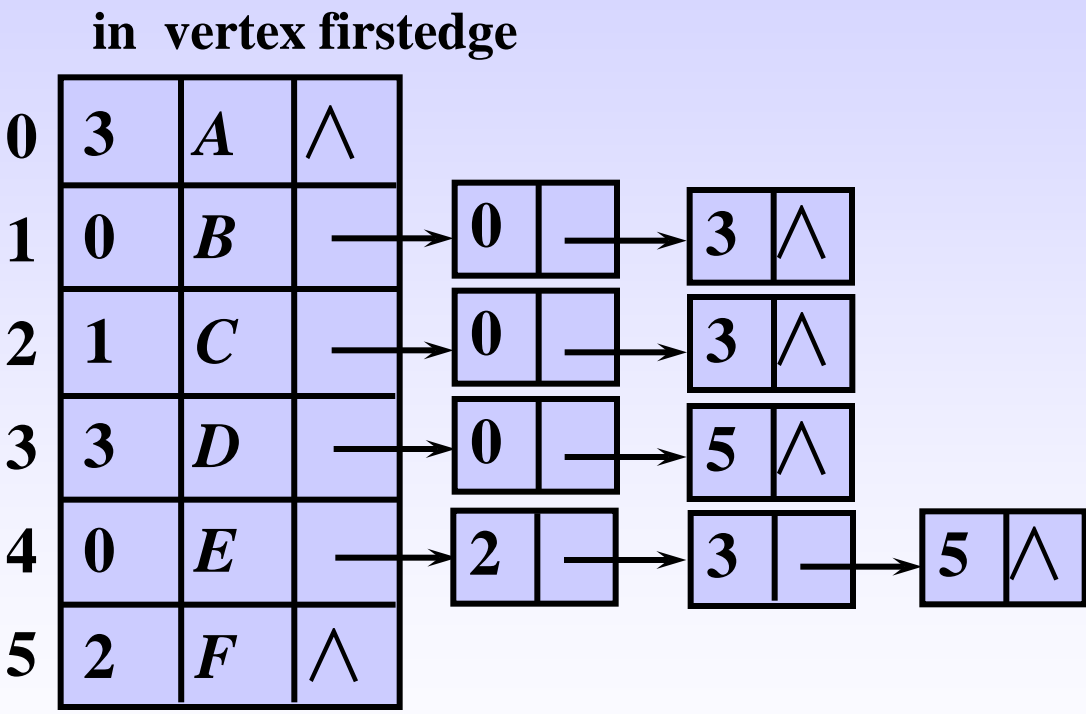
2. 栈S：存储所有无前驱的顶点。

6.5 有向无环图及其应用

拓扑排序



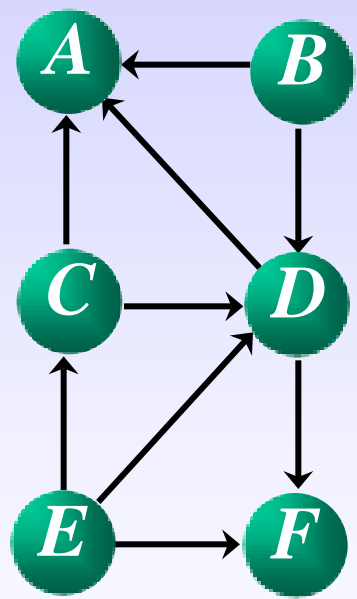
(a) 一个AOV网



(b) AOV网的邻接表存储

6.5 有向无环图及其应用

拓扑排序



in vertex firstedge

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

0

→

3

^

0

→

3

^

0

→

5

^

2

→

3

→

5

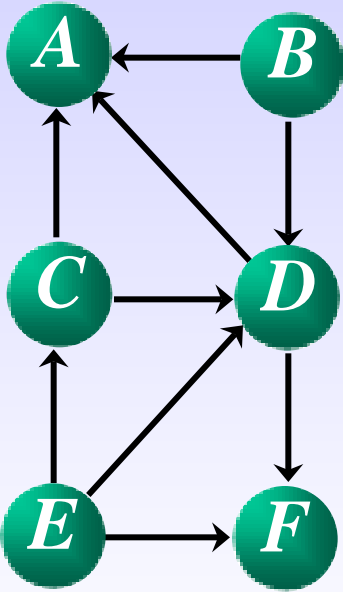
^

E

B

6.5 有向无环图及其应用

拓扑排序



in vertex firstedge

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

0

→

3

^

0

→

3

^

0

→

5

^

2

→

3

→

5

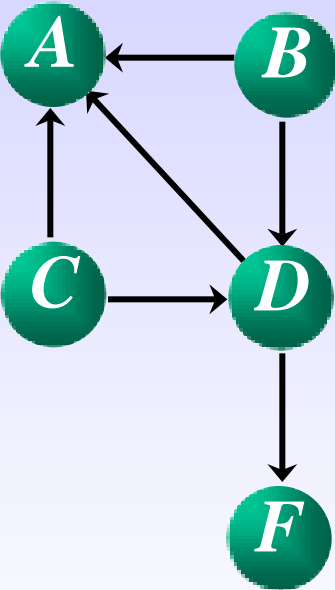
^

C

B

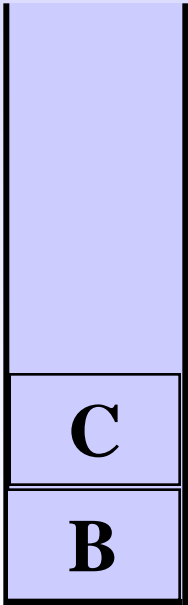
6.5 有向无环图及其应用

拓扑排序



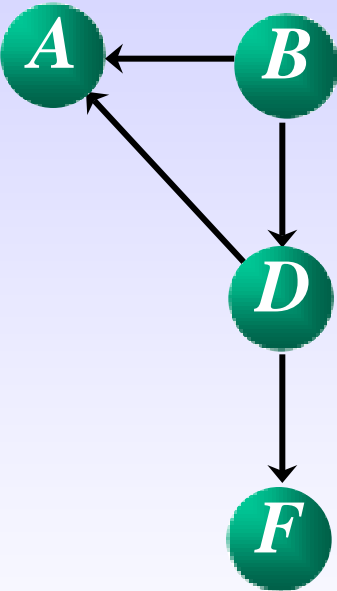
in vertex firstedge

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



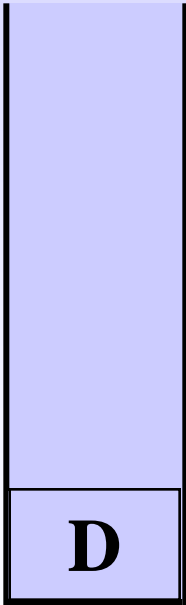
6.5 有向无环图及其应用

拓扑排序



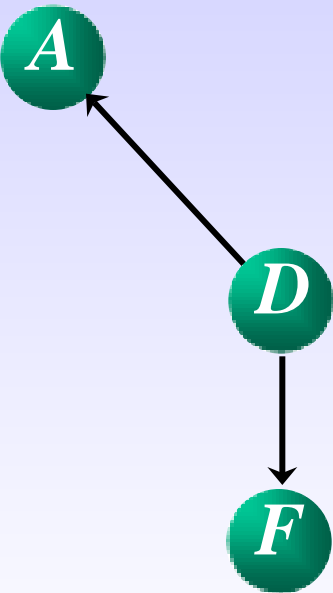
in vertex firstedge

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	^



6.5 有向无环图及其应用

拓扑排序



in vertex firstedge

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

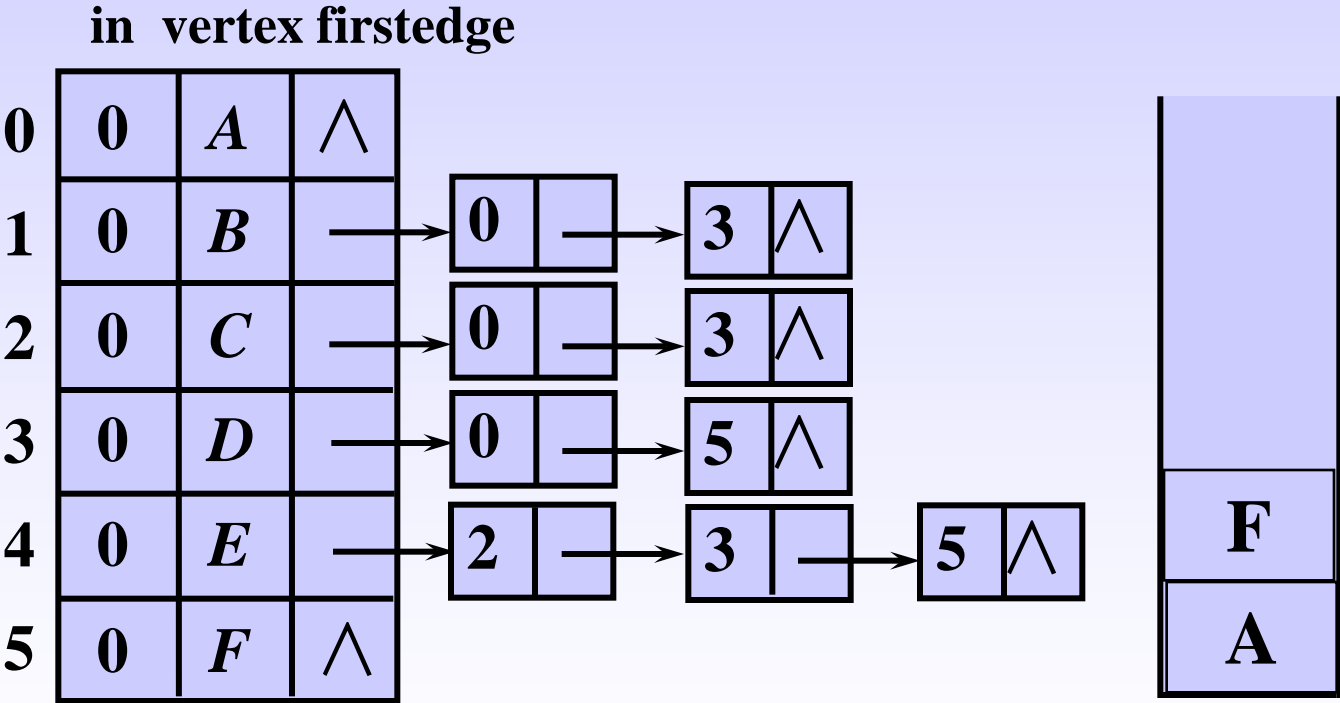
F
A

6.5 有向无环图及其应用

拓扑排序

A

F



6.5 有向无环图及其应用

拓扑排序算法——伪代码

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 有向无环图及其应用

拓扑排序算法

```
template<class T>
void ALGraph<T>::TopSort()
{
    int top= -1, count=0, S[MaxSize] ; //采用顺序栈S并初始化，累加器初始化；
    int k;int j;
    for (int i=0; i< vertexNum; i++) //扫描顶点表，将入度为0的顶点压栈；
        if (adjlist[i].in==0) S[++top]=i;
```

6.5 有向无环图及其应用

拓扑排序算法(续)

```
while (top!= -1 )           //当图中还有入度为0的顶点时循环
{
    j=S[top--];             //从栈中取出一个入度为0的顶点
    cout<<adjlist[j].vertex; count++;
    ArcNode * p=adjlist[j].firstedge; //扫描顶点表, 找出顶点j的所有出边
    while (p!=NULL)
    {
        k=p->adjvex;
        adjlist[k].in--;    //将入度减1, 如果为0, 则将该顶点入栈
        if (adjlist[k].in==0) S[++top]=k;
        p=p->next;
    }
} //while
if (count<vertexNum ) cout<<"有回路";
}
```

AOE网与关键路径

AOE网

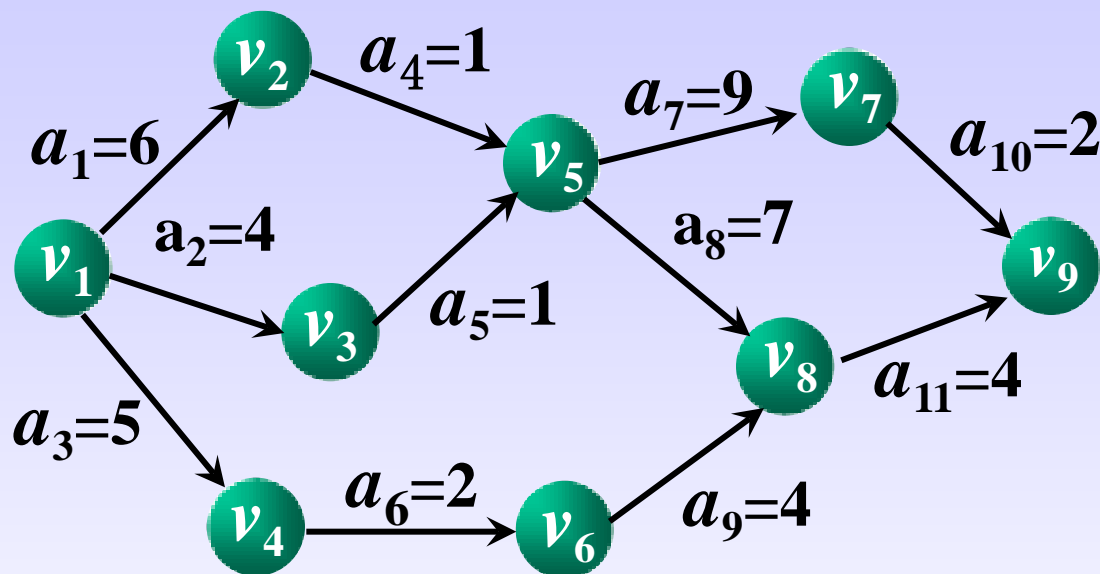
AOE网：在一个表示工程的带权有向图中，用顶点表示事件，用**有向边表示活动**，边上的**权值表示活动的持续时间**，称这样的有向图叫做**边表示活动的网**，简称**AOE网**。AOE网中没有入边的顶点称为**始点**（或源点），没有出边的顶点称为**终点**（或汇点）。

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网与关键路径

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网与关键路径

AOE网

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**。

AOE网与关键路径

关键路径

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。

AOE网与关键路径

关键路径

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：

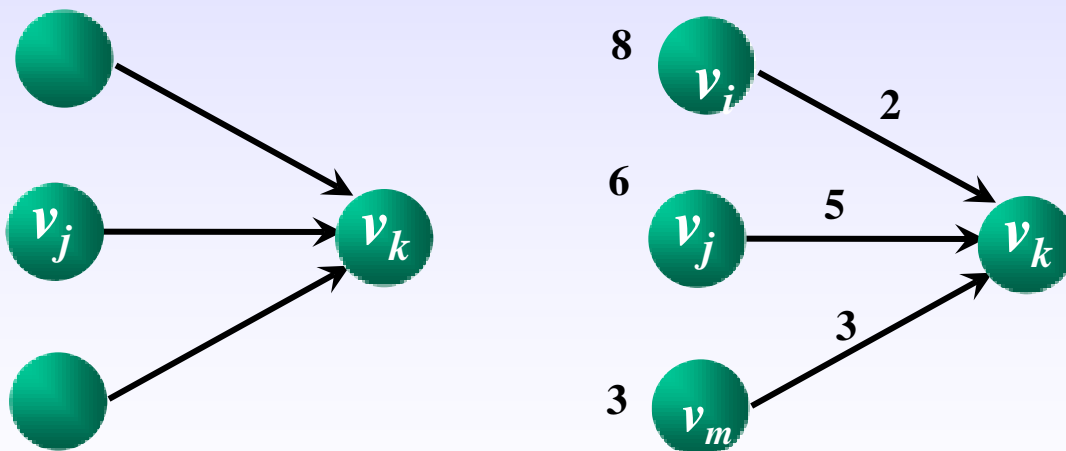
- (1) 事件的最早发生时间 $ve[k]$
- (2) 事件的最迟发生时间 $vl[k]$
- (3) 活动的最早开始时间 $e[i]$
- (4) 活动的最晚开始时间 $l[i]$

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。

AOE网与关键路径

(1) 事件的最早发生时间 $ve[k]$

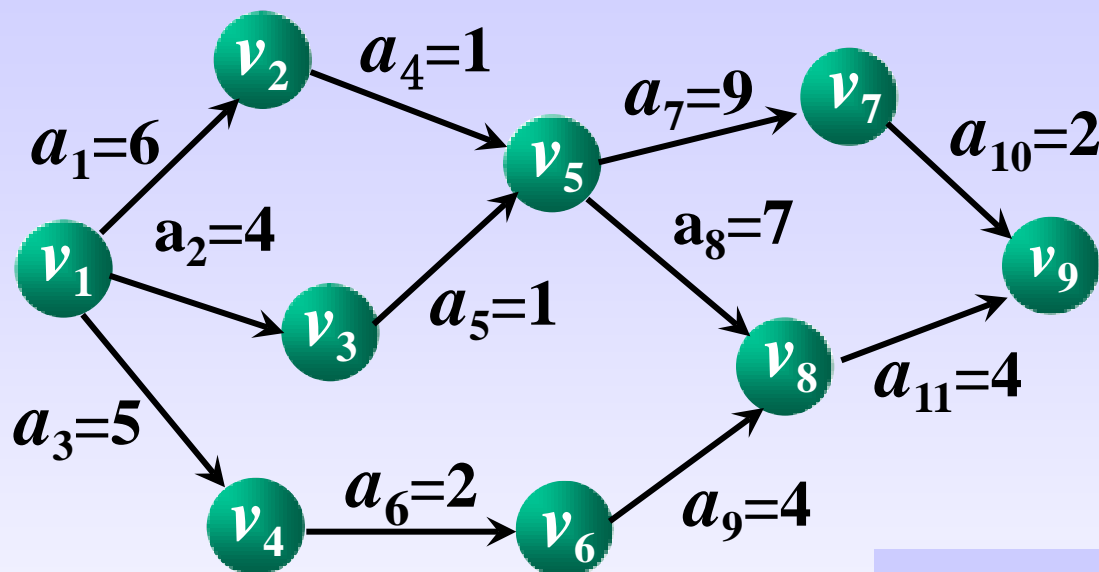
$ve[k]$ 是指从始点开始到顶点 v_k 的最大路径长度。这个长度决定了所有从顶点 v_k 发出的活动能够开工的最早时间。



$$\begin{cases} ve[1]=0 \\ ve[k]=\max\{ve[j]+\text{len}\langle v_j, v_k\rangle\} \quad (\langle v_j, v_k\rangle \in p[k]) \end{cases}$$

$p[k]$ 表示所有到达 v_k 的有向边的集合

AOE网与关键路径



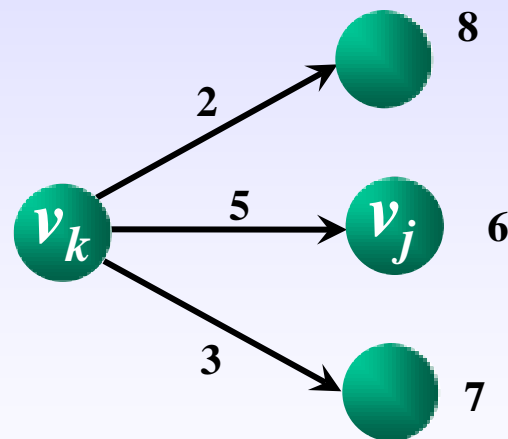
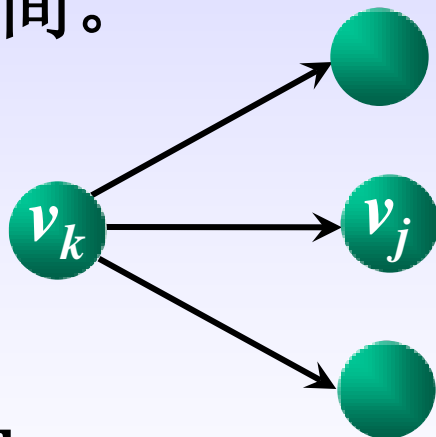
$$ve[k] = \max\{ve[j] + \text{len}\langle v_j, v_k \rangle\}$$

	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

AOE网与关键路径

(2) 事件的最迟发生时间 $vl[k]$

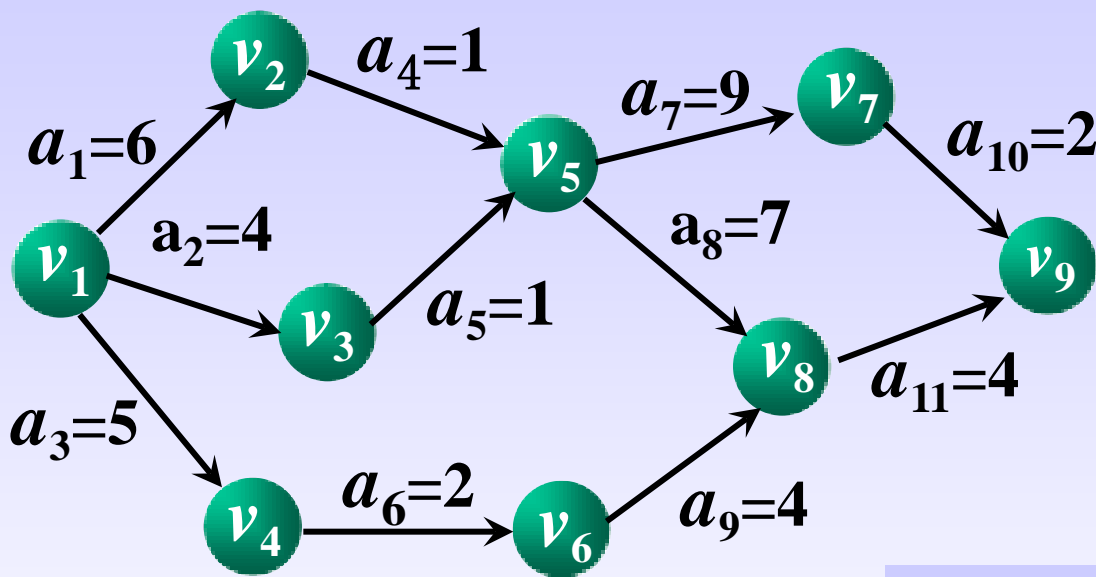
$vl[k]$ 是指在不推迟整个工期的前提下,事件 v_k 允许的最晚发生时间。



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

$s[k]$ 为所有从 v_k 发出的有向边的集合

AOE网与关键路径



$$vl[k]=\min\{vl[j]-len<v_k, v_j>\}$$

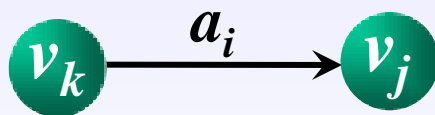
	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
ve[k]	0	6	4	5	7	7	16	14	18
vl[k]	0	6	6	8	7	10	16	14	18

AOE网与关键路径

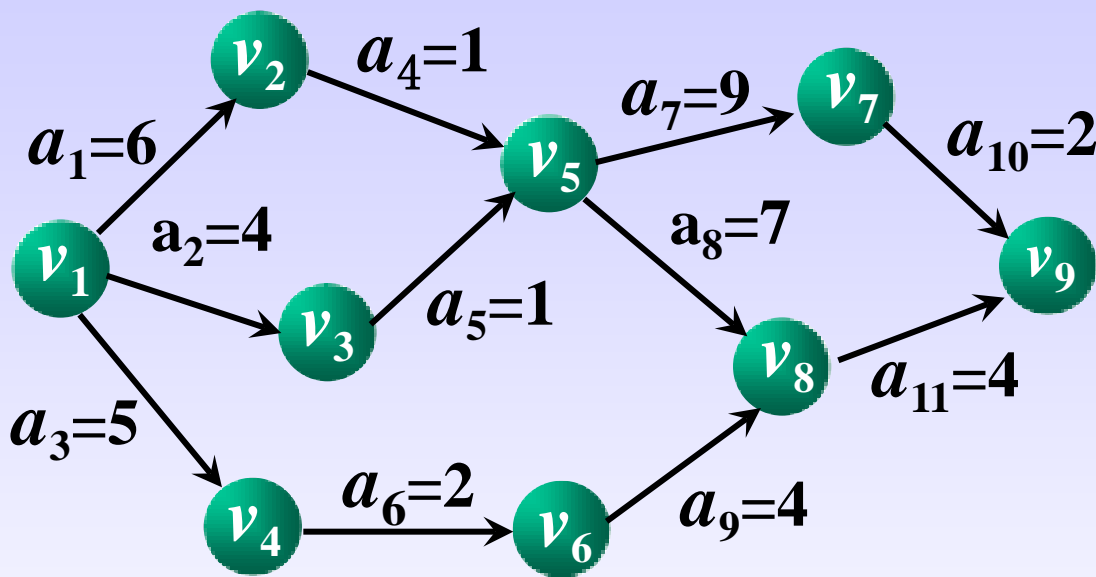
(3) 活动的最早开始时间 $e[i]$

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:

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



AOE网与关键路径



$e[i]=ve[k]$

	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

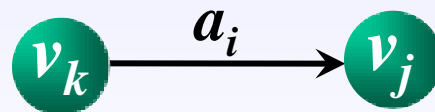
AOE网与关键路径

(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。

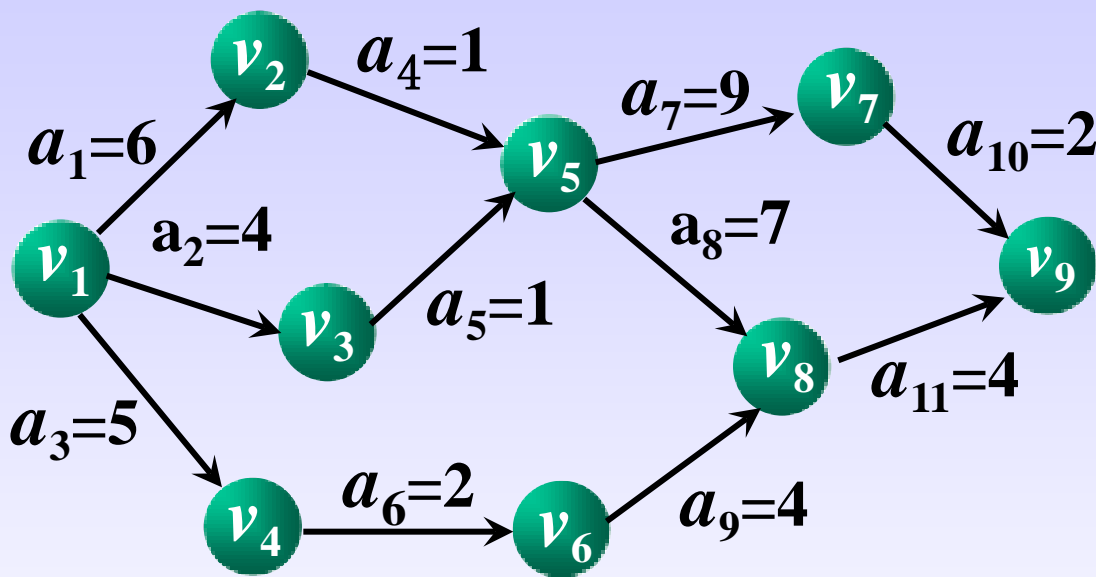
若 a_i 由弧 $\langle v_k, v_j \rangle$ 表示，则 a_i 的最晚开始时间要保证事件 v_j 的最迟发生时间不拖后。因此，有：

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



	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

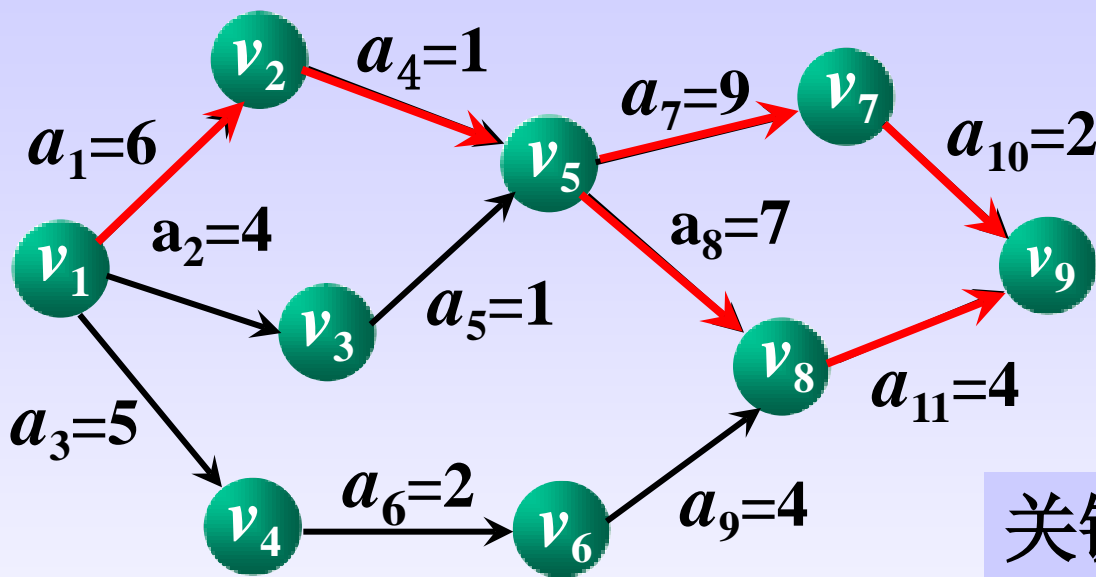
AOE网与关键路径



$$l[i]=vl[j]-len<v_k, v_j>$$

	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

AOE网与关键路径

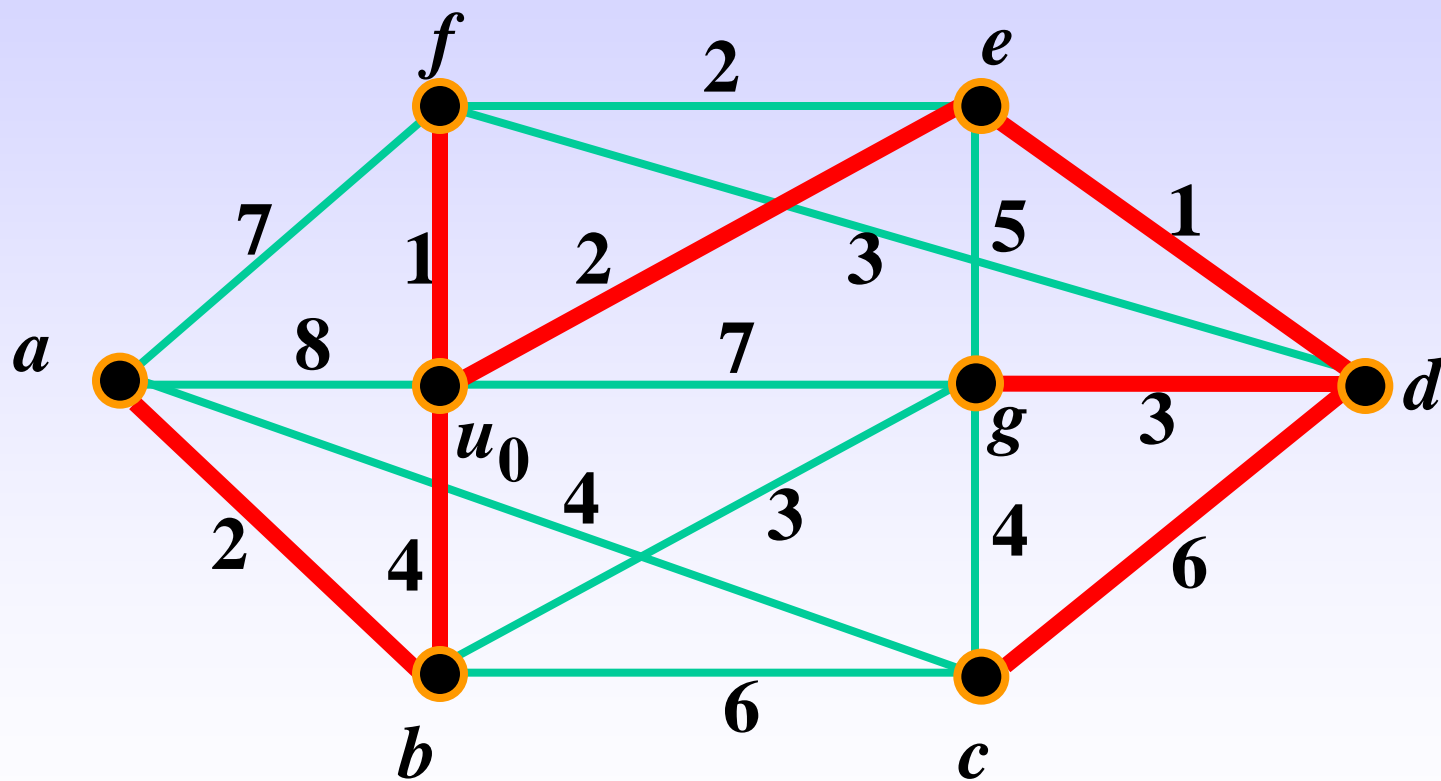


关键活动： $l[i]=e[i]$ 的活动

	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

AOE网与关键路径

课堂练习：求 u_0 到其他各顶点的最短路径。



Dijkstra算法：按路径长度递增的次序产生最短路径。

AOE网与关键路径

	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								