



数据结构与算法

Data Structure and Algorithm

第7章 图



目 录

7.1 图的基本概念

7.2 图的存贮结构

7.3 图的遍历

7.4 最小生成树

7.5 最短路径

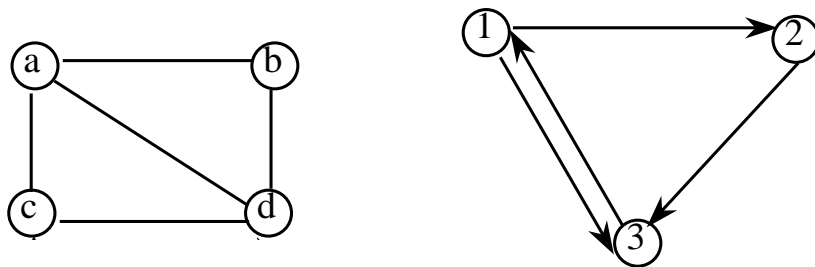
7.6 拓扑排序

7.1 图的基本概念

7.1.1 图的定义

图是由顶点集 V 和顶点间的关系集合 E （边的集合）组成的一种数据结构，可以用二元组定义为： $G=(V,E)$ 。

例如，对于图7-1所示的无向图 G_1 和有向图 G_2 ，它们的数据结构可以描述为： $G_1=(V_1,E_1)$ ，其中 $V_1=\{a,b,c,d\}$ ， $E_1=\{(a,b),(a,c),(a,d),(b,d),(c,d)\}$ ，而 $G_2=(V_2,E_2)$ ，其中 $V_2=\{1,2,3\}$ ， $E_2=\{<1,2>,<1,3>,<2,3>,<3,1>\}$ 。



(a) 无向图 G_1

(b) 有向图 G_2

图 7-1 无向图和有向图

7.1.2 图的基本术语

1. 有向图和无向图

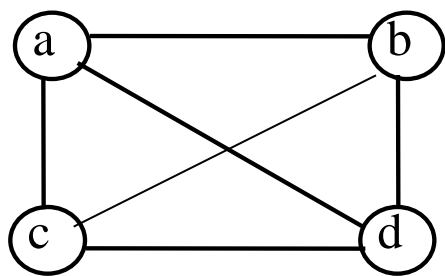
在图中，若用箭头标明了边是有方向性的，则称这样的图为有向图，否则称为无向图。如图7-1中， G_1 为无向图， G_2 为有向图。

$$\begin{cases} \text{无向图} & (x, y) \leftrightarrow (y, x) \\ \text{有向图} & \langle x, y \rangle \rightarrow \langle y, x \rangle \end{cases}$$

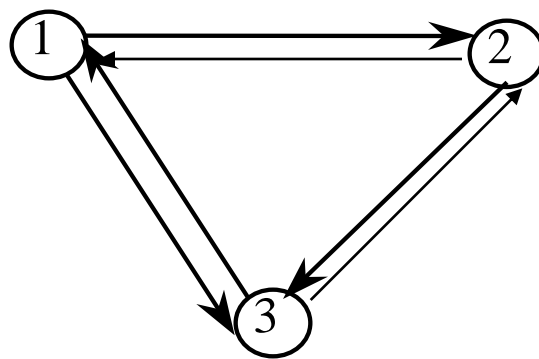


2. 完全图、稠密图、稀疏图

具有 n 个顶点， $n(n-1)/2$ 条边的图，称为完全无向图，具有 n 个顶点， $n(n-1)$ 条弧的有向图，称为完全有向图。完全无向图和完全有向图都称为完全图。



(a) 完全无向图 G_3



(b) 完全有向图 G_4

图 7-2 完全图



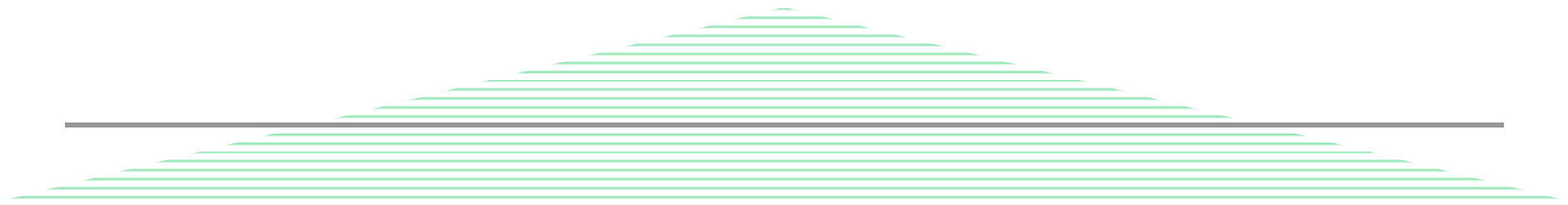
2. 完全图、稠密图、稀疏图

对于一般无向图，顶点数为 n ，边数为 e ，则 $0 \leq e \leq n(n-1)/2$ 。

对于一般有向图，顶点数为 n ，弧数为 e ，则 $0 \leq e \leq n(n-1)$ 。

当一个图接近完全图时，则称它为稠密图。

相反地，当一个图中含有较少的边或弧时，则称它为稀疏图。





7.1.2 无向图的基本术语

1. 邻接

在有向图中，存在边 (x, y) ，则称 x 和 y 是相邻的。

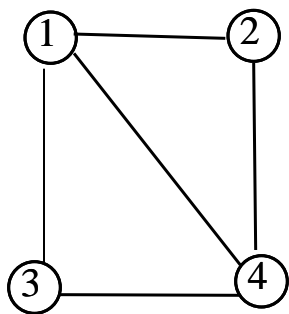
2. 依附：

有边 (x, y) ，则边依附在顶点 x 和 y 上，或者说 (x, y) 和顶点 x 、 y 相关联

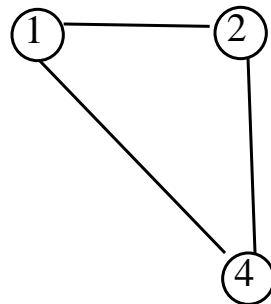
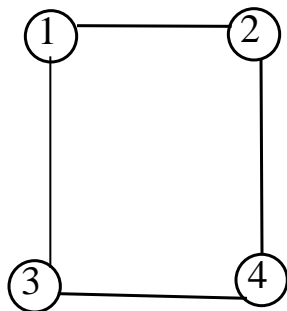
3. 子图

若有两个图 G_1 和 G_2 ， $G_1=(V_1, E_1)$ ， $G_2=(V_2, E_2)$ ，满足如下条件： $V_2 \subseteq V_1$ ， $E_2 \subseteq E_1$ ，即 V_2 为 V_1 的子集， E_2 为 E_1 的子集，称图 G_2 为图 G_1 的子图。

图和子图的示例具体见图7-3。



(a)图 G



(b)图 G 的两个子图

图 7-3 图与子图示意

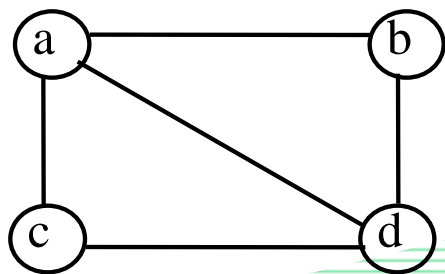
4. 路径、路径长度、简单路径、回路

从顶点x到y存在一条路径的一个顶点序列称为路径。

路径上经过的边的数目称为该路径的路径长度。

若一条路径上除起点和终点可以相同外，其余顶点均不相同，则称此路径为简单路径。

起点和终点相同的路径称为回路，简单路径组成的回路称为简单回路。



例如：

路径 (a, b, d)

路径长度 2

简单路径 (a,b,c)

非简单路径 (a, b, d, a, c)

回路 (a, b, d, a)

5. 连通、连通图、连通分量

在无向图中，若从顶点 i 到顶点 j 有路径，则称顶点 i 和顶点 j 是连通的。

若任意两个顶点都是连通的，则称此无向图为连通图，否则称为非连通图。

连通图和非连通图示例见图7-4。

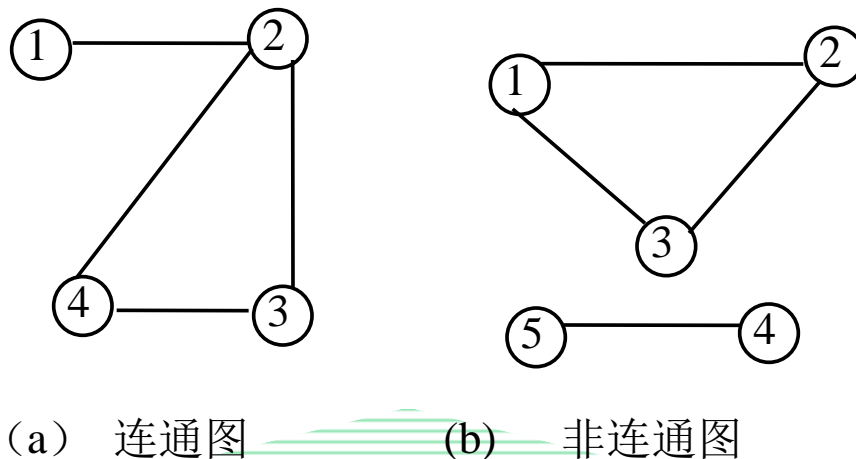


图 7-4 连通图和非连通图



无向图中，极大的连通子图为该图的连通分量。显然，任何连通图的连通分量只有一个，即它本身，而非连通图有多个连通分量。

对于图7-4 中的非连通图，它的连通分量见图7-6。

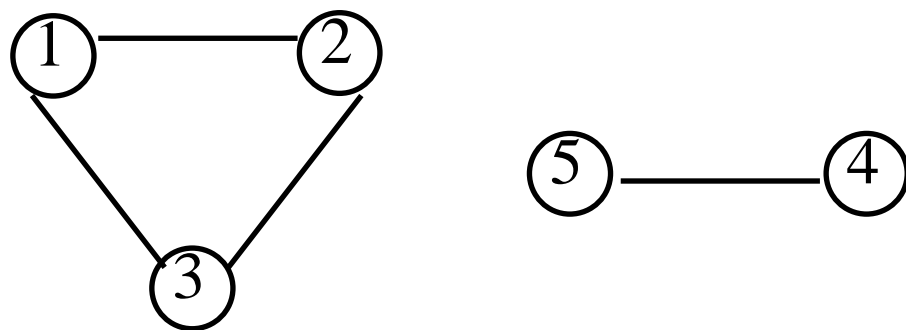


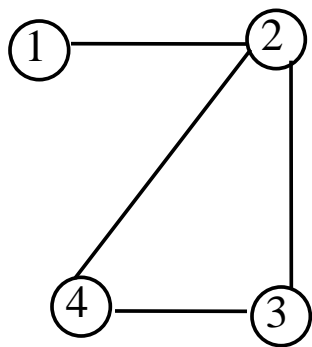
图 7-6 图 7-4(b)的连通分量

6. 生成树、生成森林

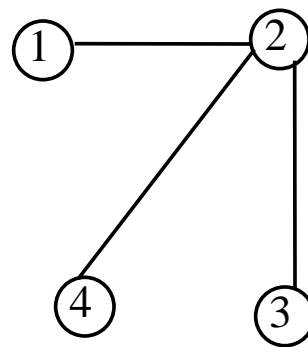
连通图的生成树是一个极小连通子图，它包含图中全部 n 个顶点和 $n-1$ 条不构成回路的边。

非连通图的生成树则组成一个生成森林。

若图中有 n 个顶点， m 个连通分量，则生成森林中有 $n-m$ 条边。



连通图

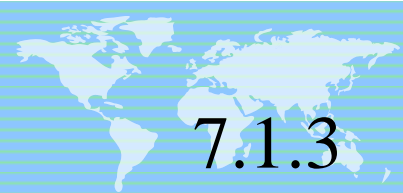


生成树



性质：

1. 添加一条边，必定构成一个环。
2. 一棵有 n 个顶点的生成树有且仅有 $n-1$ 条边。
 - 顶点 n 个，边数 $< n-1$ 非连通图
 - 顶点 n 个，边数 $> n-1$ 必有环
 - 有 $n-1$ 条边的图不一定是生成树



7.1.3 有向图的基本术语

1. 度、入度、出度

在图中，一个顶点依附的边或弧的数目，称为该顶点的度。

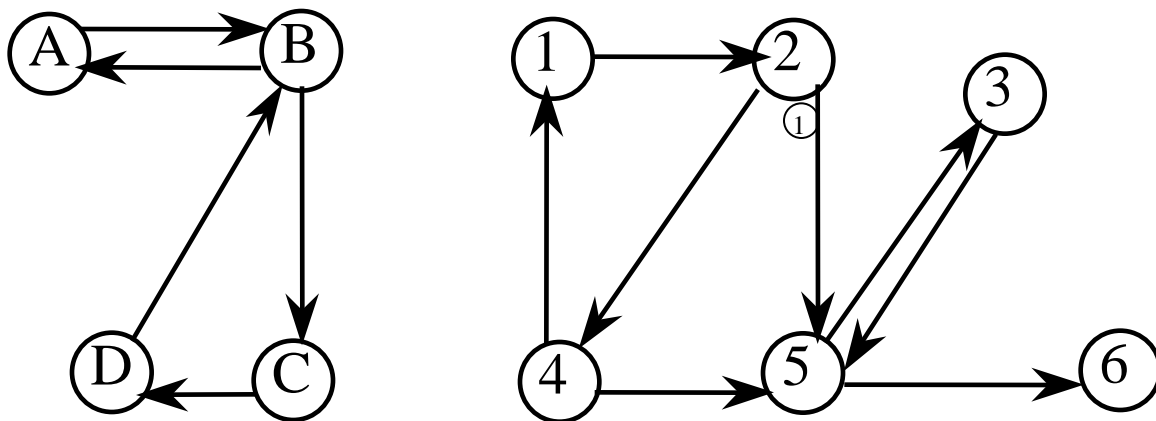
在有向图中，一个顶点依附的弧头数目，称为该顶点的入度；一个顶点依附的弧尾数目，称为该顶点的出度；某个顶点的入度和出度之和称为该顶点的度。

2. 强连通图、强连通分量

在有向图中，若从顶点 i 到顶点 j 有路径，则称从顶点 i 和顶点 j 是连通的。

若图中任意两个顶点都是连通的，则称此有向图为强连通图，否则称为非强连通图。

强连通图和非强连通图示例见图7-5。



(a) 强连通图 (b) 非强连通图

图 7-5 强连通图和非强连通图



有向图中，极大的强连通子图为该图的强连通分量。显然，任何强连通图的强连通分量只有一个，即它本身，而非强连通图有多个强连通分量。

对于图7-5 中的非强连通图，它的强连通分量见图7-7。

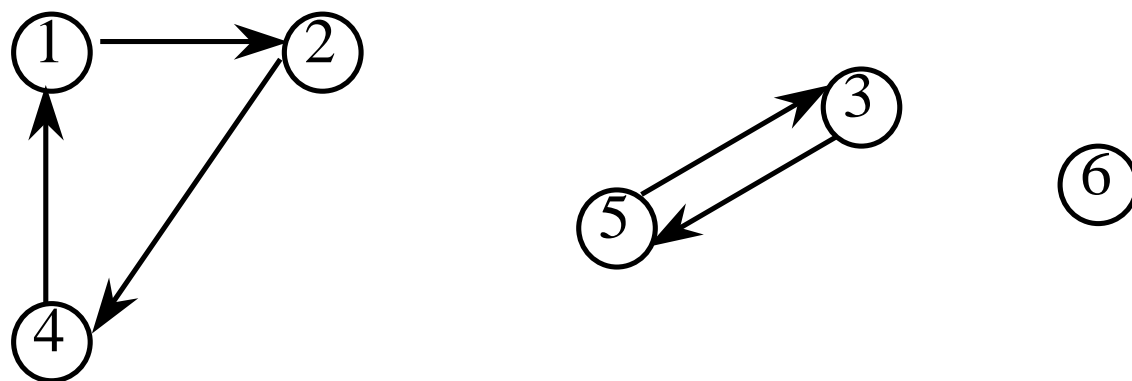


图 7-7 图 7-5(b)的强连通分量

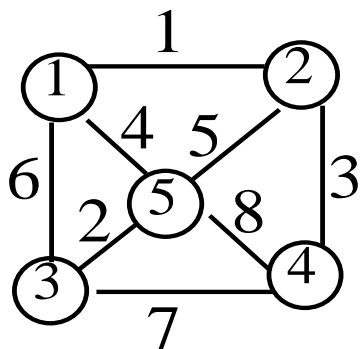
7.1.4 其他术语

1. 权

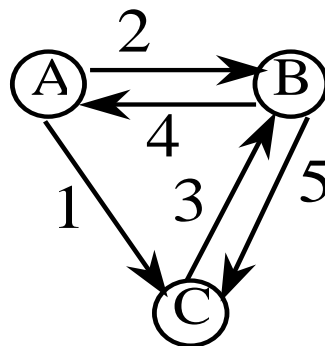
在图的边或弧中给出相关的数，称为权。权可以代表一个顶点到另一个顶点的距离，耗费等。

2. 网

带权图一般称为网。带权图的示例具体见图7-4。



(a) 无向网



(b) 有向网

图 7-3 无向带权图和有向带权图



7.2 图的存贮结构

7.2.1 邻接矩阵-----无、有向图、网

7.2.2 邻接表-----无、有向图、网

7.2.1 十字链表-----有向图

7.2.1 邻接多重表-----无向图

7.2.1 邻接矩阵

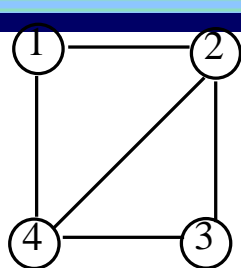
1. 图的邻接矩阵表示

在邻接矩阵表示中，除了存放顶点本身信息外，还用了一个矩阵表示各个顶点之间的关系。若 $(i,j) \in E(G)$ 或 $\langle i,j \rangle \in E(G)$, 则矩阵中第 i 行 第 j 列元素值为 1，否则为 0。

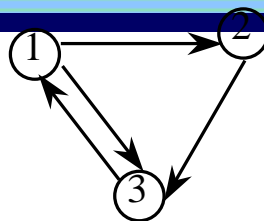
图的邻接矩阵定义为：

$$A[i][j] = \begin{cases} 1 & \text{若 } (i,j) \in E(G) \text{ 或 } \langle i,j \rangle \in E(G) \\ 0 & \text{其它情形} \end{cases}$$

例如, 对图7-8所示的无向图和有向图, 它们的邻接矩阵见图7-9。



(a) 无向图 G3



(b) 有向图 G4

图 7-8 无向图 G3 及有向图 G4

$$\begin{array}{c}
 \begin{array}{cccc}
 & 1 & 2 & 3 & 4 \\
 1 & \begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix} \\
 2 & \begin{bmatrix} 1 & 0 & 1 & 1 \end{bmatrix} \\
 3 & \begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix} \\
 4 & \begin{bmatrix} 1 & 1 & 1 & 0 \end{bmatrix}
 \end{array}
 \end{array}$$

(a) G3 的邻接矩阵

$$\begin{array}{c}
 \begin{array}{ccc}
 & 1 & 2 & 3 \\
 1 & \begin{bmatrix} 0 & 1 & 1 \end{bmatrix} \\
 2 & \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \\
 3 & \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}
 \end{array}
 \end{array}$$

(b) G4 的邻接矩阵

图 7-9 邻接矩阵表示



7.2.1 邻接矩阵

2. 从无向图的邻接矩阵可以得出如下结论

- (1) 矩阵是对称的;
- (2) 第 i 行或第 i 列1的个数为顶点 i 的度;
- (3) 矩阵中1的个数的一半为图中边的数目;
- (4) 很容易判断顶点 i 和顶点 j 之间是否有边相连(看矩阵中 i 行 j 列值是否为1)。



7.2.1 邻接矩阵

3. 从有向图的邻接矩阵可以得出如下结论

- (1) 矩阵不一定是对称的;
- (2) 第 i 行中1的个数为顶点 i 的出度;
- (3) 第 i 列中1的个数为顶点 i 的入度;
- (4) 矩阵中1的个数为图中弧的数目;
- (5) 很容易判断顶点 i 和顶点 j 是否有弧相连.

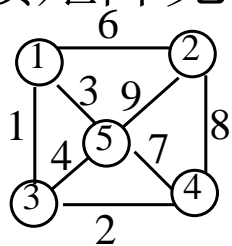
7.2.1 邻接矩阵

4. 网的邻接矩阵表示

类似地可以定义网的邻接矩阵为：

$$A[i][j] = \begin{cases} w_{ij} & \text{若 } (i,j) \in E(G) \text{ 或 } \langle i,j \rangle \in E(G) \\ 0 & \text{若 } i=j \\ \infty & \text{其它情形} \end{cases}$$

网及网的邻接矩阵见图7-10。



∞	6	1	∞	3
6	∞	∞	8	9
1	∞	∞	2	4
∞	8	2	∞	7
3	9	4	7	∞

(a) 网 G5

(b) 网 G5 的邻接矩阵示意图

图 7-10 网及邻接矩阵示意图

7.2.1 邻接矩阵

5. 储存表示

```
// ----- 图的数组(邻接矩阵)存储表示 -----
#define INFINITY INT_MAX           // 用整型最大值代替 $\infty$ 
#define MAX_VERTEX_NUM 20         // 最大顶点个数
typedef enum{DG,DN,AG,AN}GraphKind; // {有向图,有向网,无向图,无向网}
typedef struct ArcCell{
    VRType adj;           // VRType是顶点关系类型。对无权图,用1(是)或0(否)
                          // 表示相邻否; 对带权图,则为权值类型
    InfoType *info;       // 该弧相关信息的指针
}ArcCell,AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];

typedef struct {
    VertexType vexs[MAX_VERTEX_NUM]; // 顶点向量
    AdjMatrix arcs;                  // 邻接矩阵
    int vexnum,arcnum;               // 图的当前顶点数和弧数
    GraphKind kind;                  // 图的种类标志
}MGraph;
```




7.2.1 邻接矩阵

6. 图的邻接矩阵表示的优缺点

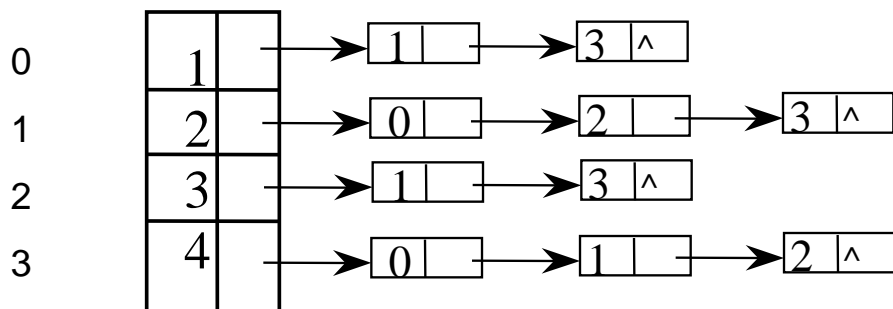
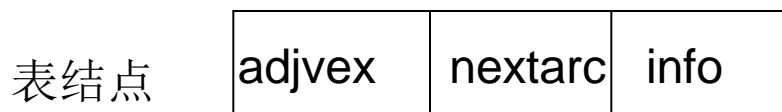
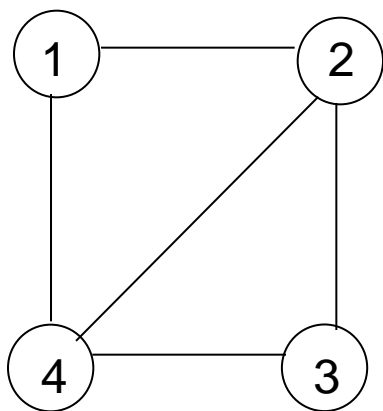
优点：容易判定任何两个顶点是否有边或弧相连

缺点：找到任一个顶点的邻结点，则需要搜索矩阵的一行或者一列链表，不及邻接表方便

7.2.2 邻接表

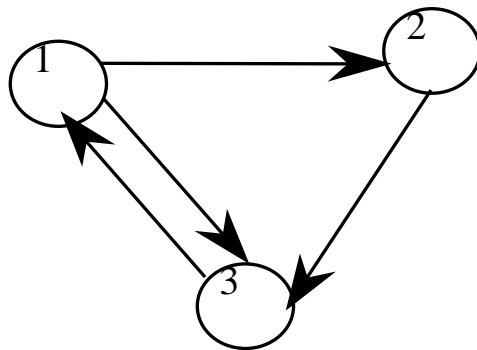
1. 图的邻接表表示

将每个结点的边用一个单链表链接起来，若干个结点可以得到若干个单链表，每个单链表都有一个头结点，所有头结点组成一个一维数组，称这样的链表为邻接表。

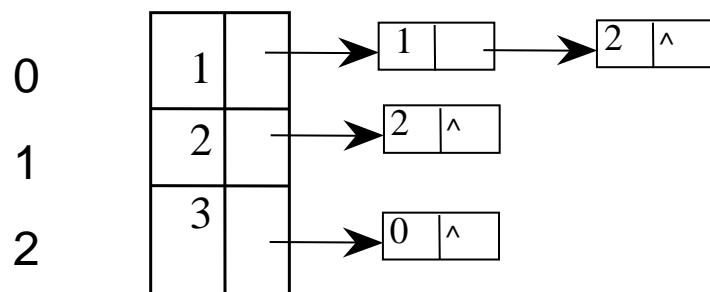


(a) 无向图 G3 的邻接表

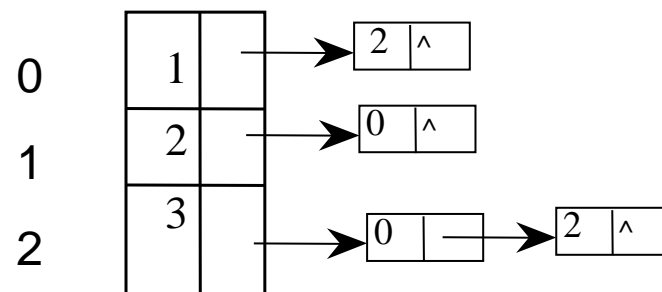
7.2.2 邻接表



(b) 有向图 G4



(b) 有向图 G4 的邻接表



(c) 有向图 G4 的逆邻接表

图 7-11 邻接表示例



7.2.2 邻接表

2. 从无向图的邻接表可以得到如下结论

- (1) 第 i 个链表中结点数目为顶点 i 的度;
- (2) 所有链表中结点数目的一半为图中边数;
- (3) 占用的存储单元数目为 $n+2e$ (n 个结点, e 条边)。

3. 从有向图的邻接表可以得到如下结论

- (1) 第 i 个链表中结点数目为顶点 i 的出度;
- (2) 所有链表中结点数目为图中弧数;
- (3) 占用的存储单元数目为 $n+e$ (n 个结点, e 条边)。

7.2.2 邻接表

4. 储存表示

```
// ----- 图的邻接表存储表示 -----  
#define MAX_VERTEX_NUM 20  
typedef enum{DG,DN,AG,AN} GraphKind; // {有向图,有向网,无向图,无向网}  
  
typedef struct ArcNode{  
    int          adjvex;          // 该弧所指向的顶点的位置  
    struct ArcNode *nextarc;      // 指向下一条弧的指针  
    InfoType      *info;          // 网的权值指针  
}ArcNode;  
typedef struct{  
    VertexType    data;           // 顶点信息  
    ArcNode       *firstarc;       // 指向第一条依附该顶点的弧的指针  
}VNode,AdjList[MAX_VERTEX_NUM]; // 头结点  
typedef struct{  
    AdjList        vertices;  
    int            vexnum,arcnum;  // 图的当前顶点数和弧数  
    GraphKind      kind;          // 图的种类标志  
}ALGraph;
```



7.2.2 邻接表

5. 图的邻接表表示的优缺点

优点：容易找到任一个顶点的邻结点

缺点：判定任何两个顶点是否有边或弧相连，则需要搜索链表，不及邻接矩阵方便



7.2.3 十字链表

1. 图的邻接表表示

将有向图的邻接表和逆逆邻接表结合起来的一种链表。

结点定义如下：

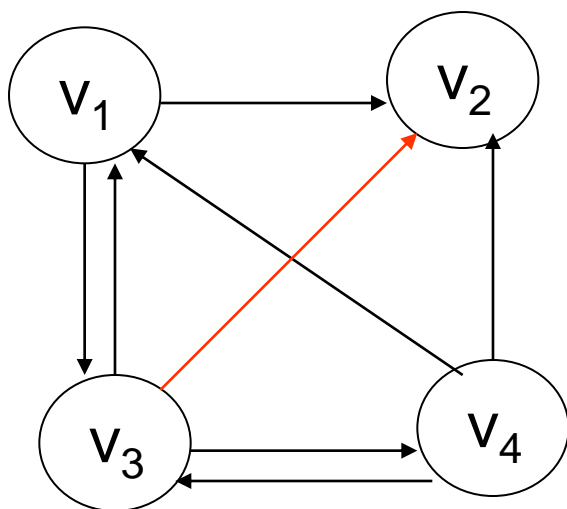
弧结点

tailvex	headvex	hlink	tlink	info
---------	---------	-------	-------	------

顶点结点

data	firstin	firstout
------	---------	----------

7.2.3 十字链表



请画出上图的十字链表存储结构



7.2.3 十字链表

优点：

1. 求顶点的出度、入度简单。
2. 建立十字链表的时间复杂度和建立邻接表相同 $O(n+e)$ 。



7.2.3 邻接多重表

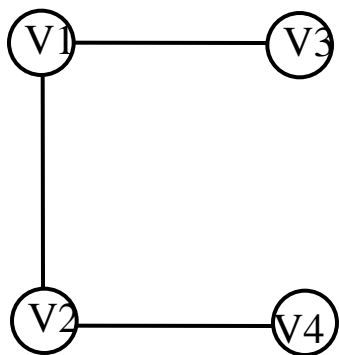
在邻接多重表中，每条边用一个结点表示，每个结点由五个域组成，其结点结构为：

边结点	mark	ivex	ilink	jvex	jlink	info
-----	------	------	-------	------	-------	------

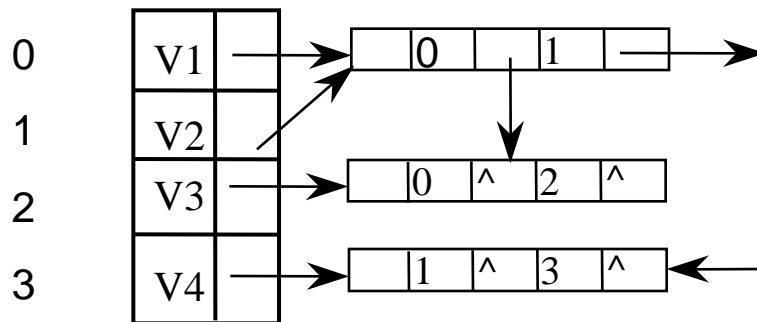
顶点结点	data	firstedge
------	------	-----------

7.2.3 邻接多重表

邻接多重表的形式见图7-12。



(a) 无向图 G6



(b) G6 的邻接多重表

图 7-12 邻接多重表示例



7.2.3 邻接多重表

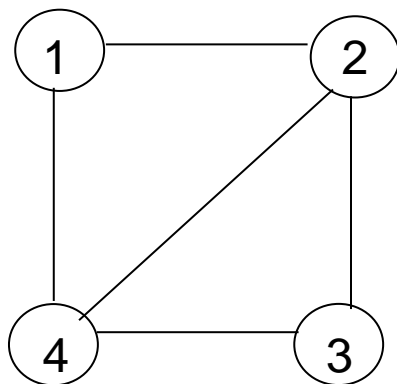
结论：

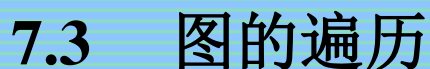
- 1.在邻接多重表中，所有依附于同一结点的边串联在同一链表中，由于每条边依附于两个结点，则每个边结点同时链接在两个链表中。
 - 2.对无向图而言，邻接多重表和邻接表的差别，仅在于同一条边在邻接表中用两个结点表示，而在邻接多重表中只有一个结点，因此除了在边结点中增加一个标志域外，邻接多重表所需的存储量和邻接表相同。
-



7.3 图的遍历

总结： 图的邻接矩阵和邻接表优点对比

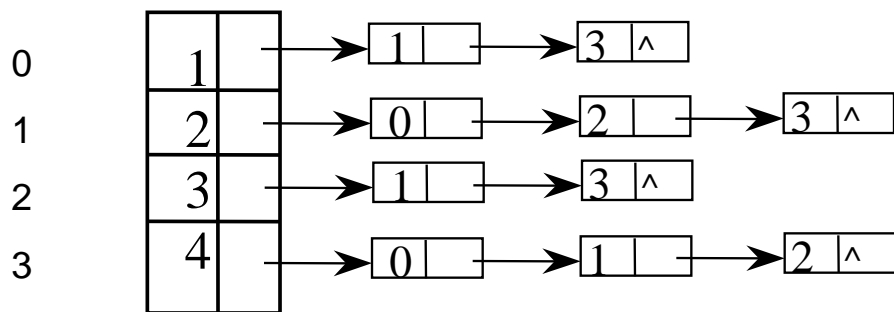




总结：图的邻接矩阵和邻接表优点对比

$$\begin{array}{ccccc}
 & 1 & 2 & 3 & 4 \\
 \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \left[\begin{array}{cccc} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{array} \right]
 \end{array}$$

(a) G3 的邻接矩阵



(b)无向图 G3 的邻接表

邻接矩阵优点：容易判定任何两个顶点是否有边或弧相连

邻接表优点：容易找到任一个顶点的邻结点



7.3 图的遍历

作业：

7.1

7.14



实验三：无向图邻接表的构造

构造一个无向图的邻接表，要求从键盘输入图的顶点数和图的边数，并显示所构造的邻接表。

实验步骤：

1. 构造一个无向图的邻接表
2. 屏幕输出

实验拓展：

1. 构建有向图的邻接表
2. 判断边是否存在
3. 求顶点的度数

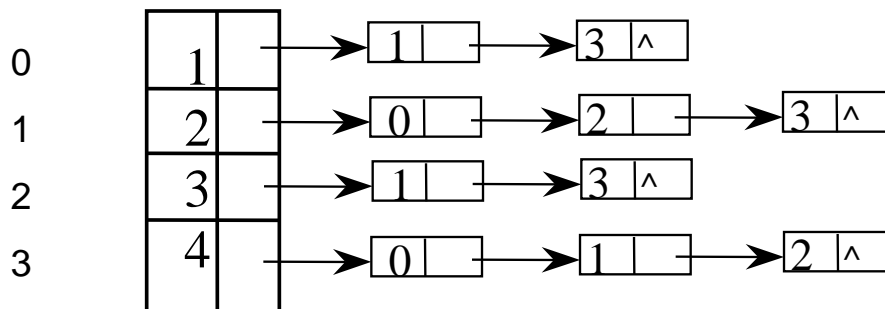
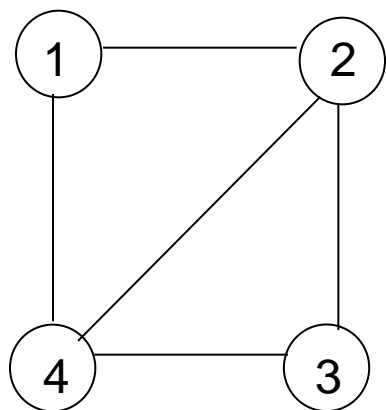
无向图邻接表例子：

头结点

data	firstarc
------	----------

表结点

adjvex	nextarc	info
--------	---------	------



(a)无向图 G3 的邻接表



结构体定义:

```
#define MAXVEX 30
typedef char VertexType;
typedef struct ArcNode {
    int adjvex;    /*邻接点序号*/
    VertexType data; /*邻接点信息*/
    struct ArcNode *next;
} ArcNode;

typedef struct VNode    {
    VertexType data;    /*结点信息*/
    ArcNode *firsrarc; /*指向下一个边结点*/
}VNode,AdjList[MAXVEX];
```



7.3 图的遍历

和树的遍历类似，图的遍历也是从某个顶点出发，沿着某条搜索路径对图中所有顶点仅各作一次访问。

若给定的图是连通图，则从图中任一顶点出发顺着边可以访问到该图中所有的顶点，但是，在图中有回路，从图中某一顶点出发访问图中其它顶点时，可能又会回到出发点，而图中可能还剩余有顶点没有访问到，因此，图的遍历较树的遍历更复杂。我们可以设置一个全局型标志数组visited来标志某个顶点是否被访问过，未访问的值为0，访问过的值为1。

。



7.3 图的遍历

7.3.1 深度优先搜索遍历

7.3.2 广度优先搜索遍历



7.3.1 深度优先搜索遍历(DFS Depth-First Search)

1. 深度优先搜索思想

深度优先搜索遍历类似于树的先序遍历(实质)。假定给定图G的初态是所有顶点均未被访问过，在G中任选一个顶点i作为遍历的初始点，则深度优先搜索遍历可定义如下：

- (1) 首先访问顶点i，并将其访问标记置为访问过，即 $visited[i]=1$ ；
- (2) 然后搜索与顶点i有边相连的下一个顶点j，若j未被访问过，则访问它，并将j的访问标记置为访问过， $visited[j]=1$ ，然后从j开始重复此过程，若j已访问，再看与i有边相连的其它顶点；
- (3) 若与i有边相连的顶点都被访问过，则退回到前一个访问顶点并重复刚才过程，直到图中所有顶点都被访问完止。

7.3.1 深度优先搜索遍历(DFS Depth-First Search)

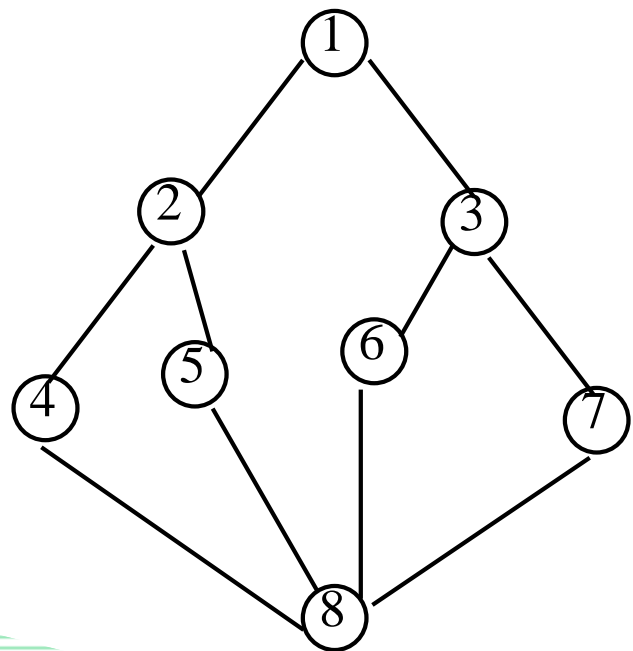
例如，对图7-13所示无向图G7，从顶点1出发的深度优先搜索遍历序列可有多种，下面仅给出三种，其它可作类似分析。

在无向图G7中，从顶点1出发的深度优先搜索遍历序列列举三种为：

1, 2, 4, 8, 5, 6, 3, 7

1, 2, 5, 8, 4, 7, 3, 6

1, 3, 6, 8, 7, 4, 2, 5



7-13 无向图 G7



7.3.1 深度优先搜索遍历(DFS Depth-First Search)

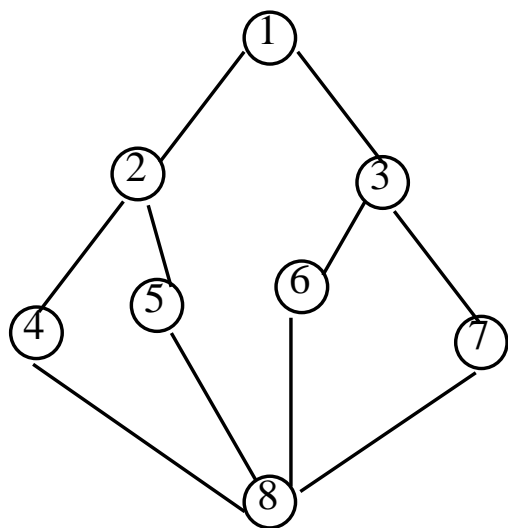
问题：

- 遍历结果不唯一！

7.3.1 深度优先搜索遍历(DFS Depth-First Search)

2. 深度优先搜索（邻接表实现图）

仍以图7-13中无向图G7 为例，来说明算法的实现，G7 的邻接表见图7-16，



7-13 无向图 G7

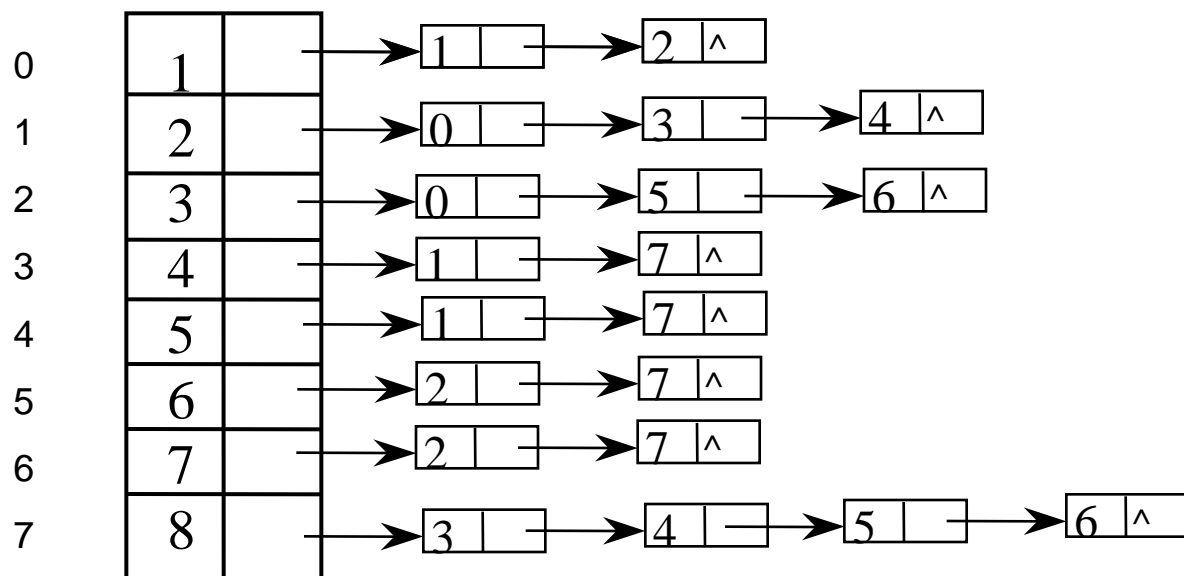
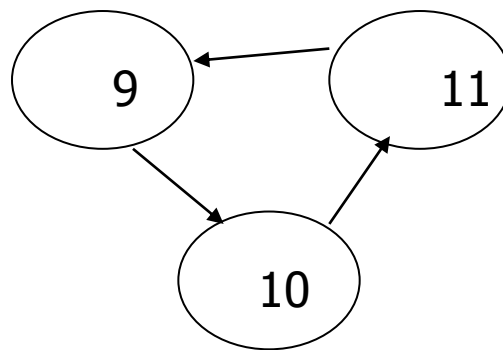


图 7-16 G 7 的邻接表

7.3.1 深度优先搜索遍历(DFS Depth-First Search)

问题:

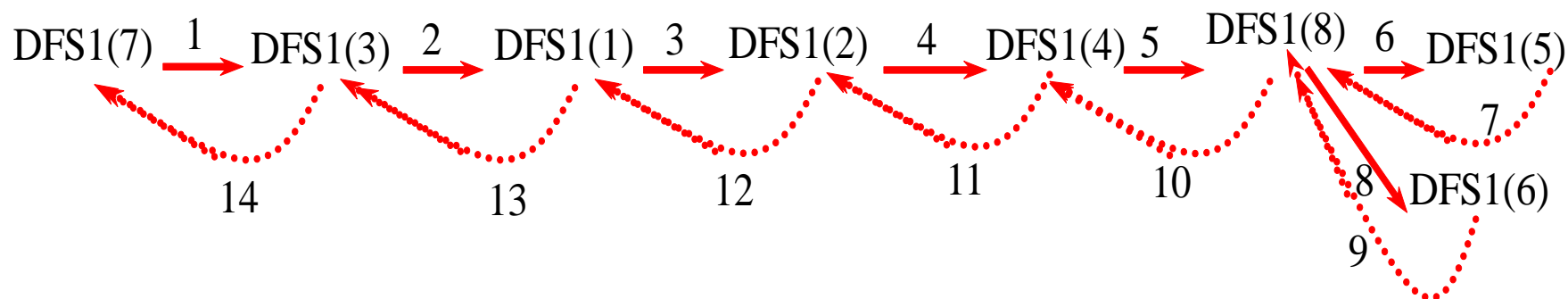
■ 非连通图可以吗?



7.3.1 深度优先搜索遍历(DFS Depth-First Search)

用刚才算法及图7-16，可以描述从顶点7出发的深度优先搜索遍历示意图，见图7-17，其中实线表示下一层递归，虚线表示递归返回，箭头旁边数字表示调用的步骤。

于是，从顶点7出发的深度优先搜索遍历序列，从图7-17中可得出为 7, 3, 1, 2, 4, 8, 5, 6。从其它顶点出发的深度优先搜索序列，请读者自己写出。



7-17 邻接表深度优先搜索示意图

7.3.1 深度优先搜索遍历(DFS Depth-First Search)

算法7.5描述为下面形式:

```
Boolean visited[MAX];  
status (*VisitFunc)(int v);
```

```
void DFSTraverse( Graph G,  
                Status(*visit)( int v)){  
    //对图G作深度优先遍历  
    VisitFunc=Visit;  
    for(v=0;v<=G.vexnum;++v)  
        visited[v]=FLASE;  
    for( v=0;v<G.vexnum; ++v)  
        if( !visited[v]) DFS( G,v);  
} //DFSTraverse
```

```
void DFS( Gragh G, int v){  
    //从第V个顶点出发递归的深度优先遍历图G  
    Visted[v]=TRUE; VisitFunc(V);  
    for( w=FirstAdjVex(G,v) ;  
        w; w=NextAdjVex ( G,v,w))  
        if( !visited[ w]) DFS( G,w);  
} //DFS
```



7.3.1 深度优先搜索遍历(DFS Depth-First Search)

思考：

- 如果存储改为邻接矩阵，相比时间效率哪个更好？



7.3.2 广度优先搜索遍历(BFS,Breadth-first Search)

1. 广度优先搜索的思想

广度优先搜索遍历类似于树的按层次遍历(实质)。设图G的初态是所有顶点均未访问，在G 中任选一顶点i作为初始点，则广度优先搜索的基本思想是：

- (1) 首先访问顶点i，并将其访问标志置为已被访问，即 $visited[i]=1$ ；
- (2) 接着依次访问与顶点i有边相连的所有顶点 W_1, W_2, \dots, W_t ；
- (3) 然后再重复按顺序访问与 W_1, W_2, \dots, W_t 有边相连又未曾访问过的顶点；

依此类推，直到图中所有顶点都被访问完为止。

7.3.2 广度优先搜索遍历(BFS,Breadth-first Search)

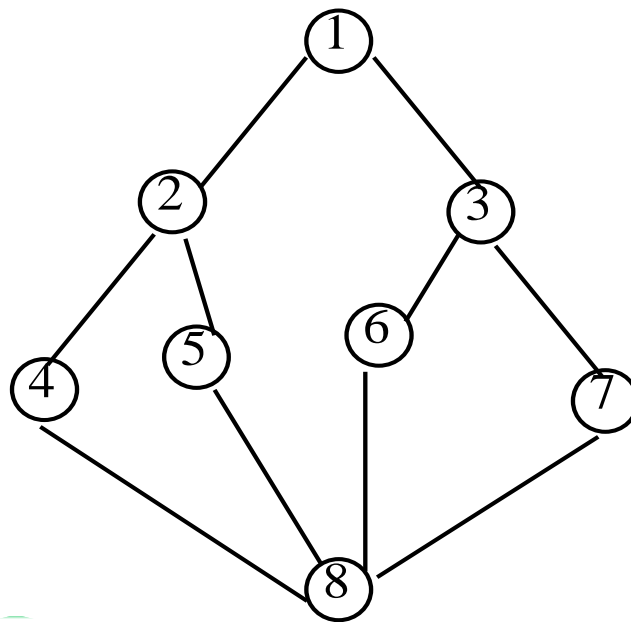
例如，对图7-13所示无向图G7，从顶点1出发的广度优先搜索遍历序列可有多种，下面仅给出三种，其它可作类似分析。

在无向图G7中，从顶点1出发的广度优先搜索遍历序列举三种为：

1, 2, 3, 4, 5, 6, 7, 8

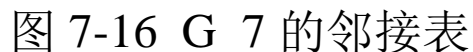
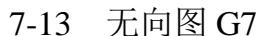
1, 3, 2, 7, 6, 5, 4, 8

1, 2, 3, 5, 4, 7, 6, 8



7-13 无向图 G7

7.3.2 广



7.3.2 广度优先搜索遍历(BFS,Breadth-first Search)

算法描述为下面形式:

```
Boolean visited[MAX];
Status (*visitFunc)(int v);

void BFSTraverse( Graph G,
                 Status(*visit)( int v)){

//对图G作深度优先遍历
    VisitFunc =visit;
    for(v=0;v<=G.vexnum; ++v)
        visited[v]= FLASE;
    for( v=0;v<G.vexnum; ++v)
        if( !visited[v])  BFS( G, v);
} //BFSTraverse
```

```
void BFS(Graph G,int v){
//从v出发广度优先遍历图G，非递归算法实现，访问操作入队列时进行

    visited[ v]=True;  VisitFunc( v);
    InitalQueue( Q);
    EnQueue( Q, v);
    While( !QueueEmpty( Q )){
        DeQueue( Q, v);
        for( w=FirstAdjVex( G, v) ;
            w; w=NextAdjVex ( G, v, w))
            if( !visited[ w])
            {  visited[ w ]=TRUE;
               visit( w );
               EnQueue( Q,w)
            } //if
        } //while
    } //BFS
```





7.3 图的遍历

作业：

7.3


7.4

7.5



7.4 最小生成树（无向图遍历的应用）

假设现在某个通信部门有一个大型建设项目，需要在 n 个城市之间建立通信网络，也就是任两个城市间有通信线路相通。这时会考虑如何在最节省经费的前提下建立这个通信网？



7.4 最小生成树（无向图遍历的应用）

7.4.1 基本概念

1. 定义

无向图的边具有不同的权值，每条边的加权值之和为最小的生成树称做最小成本生成树。

在建立最小成本生成树时，以最少成本为原则，必须满足下列条件：

- 1) 只能使用图中的边
- 2) 只能使用 $n-1$ 条边
- 3) 所使用的边不能产生一个环路



7.4 最小生成树（无向图遍历的应用）

2. 分类

7.4.2 普里姆算法

7.4.3 克鲁斯卡尔算法



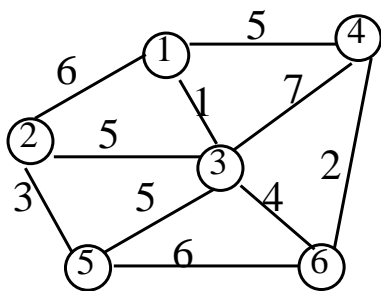
7.4.2 普里姆算法 (适合于边稠密的网)

1. 普里姆(prim)算法思想

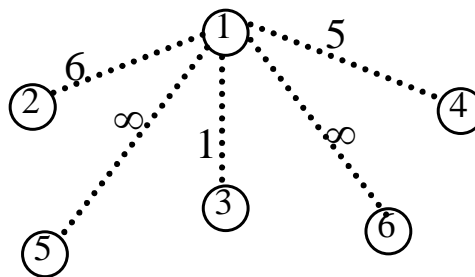
- (1) 从任一顶点开始，找出其权值最小的一条边
- (2) 由此两点向外再找一条权值最小的边连接起来，此权值次小的边必须与刚才相连接的顶点相连
- (3) 利用规则 (2) 将所有顶点相连，但不造成环路

7.4.2 普里姆算法 (适合于边稠密的网)

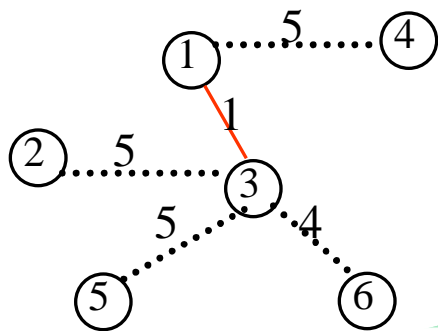
例如：假设开始顶点就选为顶点1，故首先有 $U=\{1\}$ ， $W=\{2, 3, 4, 5, 6\}$



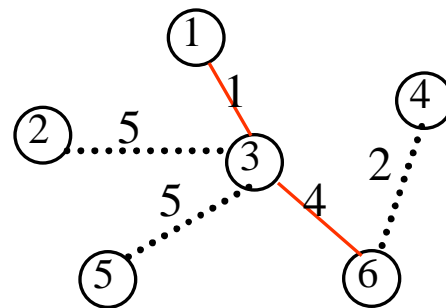
(a) 无向网



(b) $u=\{1\}$ $w=\{2,3,4,5,6\}$



(c) $u=\{1,3\}$ $w=\{2,4,5,6\}$

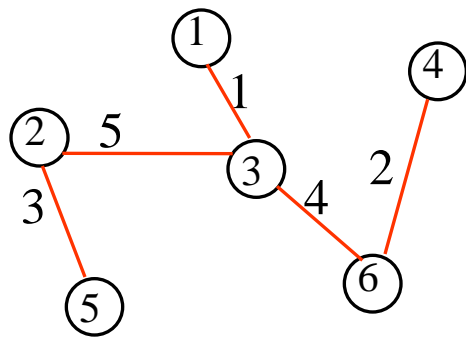


(d) $u=\{1,3,6\}$ $w=\{2,4,5\}$

7.4.2 普里姆算法 (适合于边稠密的网)



(e) $u=\{1,3,6,4\}$ $w=\{2,5\}$ (f) $u=\{1,3,6,4,2\}$ $w=\{5\}$



(g) $u=\{1,3,6,4,2,5\}$ $w=\{ \}$

图 7-20 prim 方法构造最小生成树的过程



7.4.2 普里姆算法 (适合于边稠密的网)

问题:

- 图的存储结构?
- ✓ 邻接矩阵
- 操作过程怎样来表示?
 - 选择最小的边
 - 更新到剩余点的最小值

✓ 一维closedge

```
struct{  
    VertexType adjvex;  
    VRType    lowcost;  
}closedge[MAX_VERTEX_NUM];
```




7.4.2 普里姆算法 (适合于边稠密的网)


	1	2	3	4	5	6
1	~	6	1	5	~	~
2	6	~	5	~	3	~
3	1	5	~	7	5	4
4	5	~	7	~	~	2
5	~	3	5	~	~	6
6	~	~	4	2	6	~

7.4.2 普里姆算法 (适合于边稠密的网)

closed edge i	0 1 2 3 4 5	u	v-u	k
adjvex lowcost	1 1 1 1 1 1 0 6 1 5 ∞ ∞ 1 5 ∞ 7 5 4	{ 1 }	{2, 3 , 4, 5, 6}	2
adjvex lowcost	1 3 1 1 3 3 0 5 0 5 5 4 ∞ ∞ 4 2 6 ∞	{1, 3}	{2, 4, 5, 6 }	5
adjvex lowcost	1 3 1 6 3 3 0 5 0 2 5 0 5 ∞ 7 ∞ ∞ 2	{1, 3, 6}	{2, 4 , 5}	3
adjvex lowcost	1 3 1 6 3 3 0 5 0 0 5 0 6 ∞ 5 ∞ 3 ∞	{1, 3, 6, 4}	{ 2 , 5}	1
adjvex lowcost	1 3 1 6 2 3 0 0 0 0 3 0 ∞ 3 5 ∞ ∞ 6	{1, 3, 6, 4, 2}	{ 5 }	4
adjvex lowcost	1 3 1 6 2 3 0 0 0 0 0 0	{1, 3, 6, 4, 2, 5}	{ }	

7.4.2 普里姆算法 (适合于边稠密的网)

closed edge i	0 1 2 3 4 5	u	v-u	k
adjvex lowcost	1 1 1 1 1 1 0 6 1 5 ∞ ∞ 1 5 ∞ 7 5 4	{ 1 }	{2, 3 , 4, 5, 6 }	2
adjvex lowcost	1 3 1 1 3 3 0 5 0 5 5 4 ∞ ∞ 4 2 6 ∞	{1, 3}	{2, 4, 5, 6 }	5
adjvex lowcost	1 3 1 6 3 3 0 5 0 2 5 0 5 ∞ 7 ∞ ∞ 2	{1, 3, 6}	{2, 4 , 5}	3
adjvex lowcost	1 3 1 6 3 3 0 5 0 0 5 0 6 ∞ 5 ∞ 3 ∞	{1, 3, 6, 4}	{ 2 , 5}	1
adjvex lowcost	1 3 1 6 2 3 0 0 0 0 3 0 ∞ 3 5 ∞ ∞ 6	{1, 3, 6, 4, 2}	{ 5 }	4
adjvex lowcost	1 3 1 6 2 3 0 0 0 0 0 0	{1, 3, 6, 4, 2, 5}	{ }	



7.4.2 普里姆算法 (适合于边稠密的网)

2. 普里姆(prim)算法

```
void MiniSpanTree_PRIM(MGraph G,VertexType u){  
    // 用普里姆算法从第u个顶点出发构造网G的最小生成树T，输出T的各条边  
    //记录从顶点集U到V-U的代价最小的边的辅助数组定义;  
    //struct{  
        //  VertexType adjvex;  
        //  VRType    lowcost;  
    //}closedge[MAX_VERTEX_NUM];  
    k=LocateVex(G,u); //定位顶点u的在邻接矩阵中的位序  
    for( j=0;j<G.vexnum;++j ) // 辅助数组初始化  
        if( j!=k ) closedge[j] = { u, G.arcs[k][j].adj } ; //{adjvex,lowcost}  
    closedge[k].lowcost=0;      // 初始,U={u}
```



7.4.2 普里姆算法 (适合于边稠密的网)

2. 普里姆(prim)算法

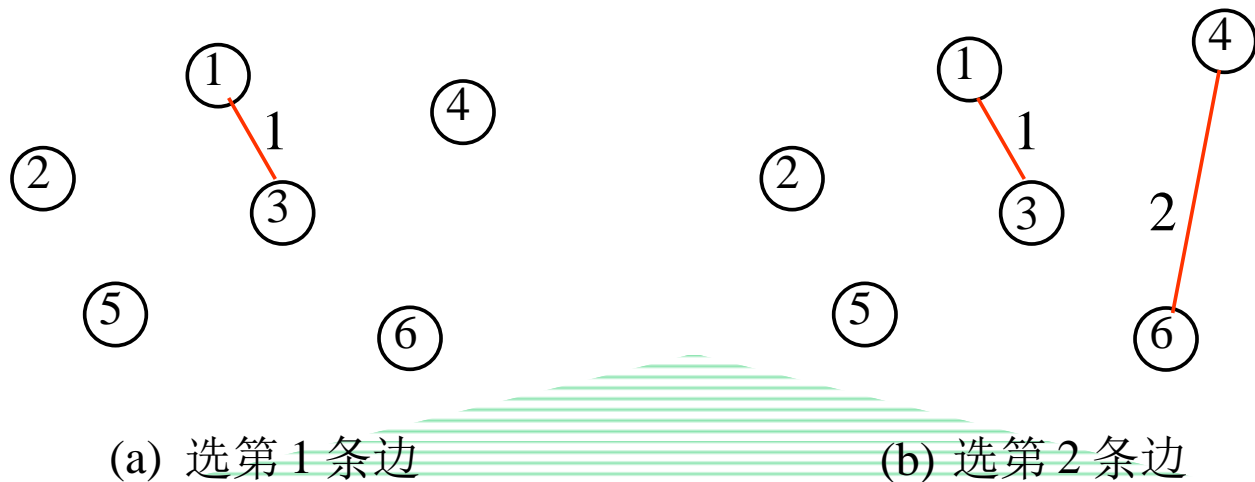
```
for(i=1;i<G.vexnum;++i) {    // 选择其余G.vexnum-1个顶点
    k=minimum(closededge,G); //求出T的下一个结点: 第K顶点
    //此时closededge[k].lowcost =
    //      MIN{closededge[vi].lowcost | closededge[vi].lowcost>0,vi∈ V-U}
    printf(closededge[k].adjvex, G.vexs[k]); //输出生成树的边
    closededge[k].lowcost=0;                // 第K顶点并入U集
    for( j=0;j<G.vexnum;++j )
        if( G.arcs[k][j].adj < closededge[j].lowcost )
            //新顶点并入U集后重新选择最小边
            closededge[j] = {G.vexs[k], G.arcs[k][j].adj };
    }
} MiniSpanTree_PRIM
```

7.4.3 克鲁斯卡尔 (kruskal) 算法 (适合于边稀疏的网)

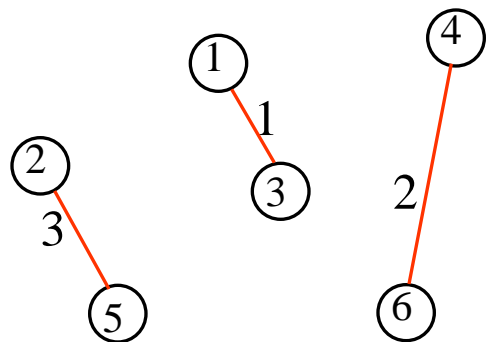
1. 克鲁斯卡尔算法基本思想

- (1) 将整个图所有边的权值依小到大列成表
- (2) 由权值最小的边开始进行连接, 若连接结果不会造成环路则成立, 否则不予采用。

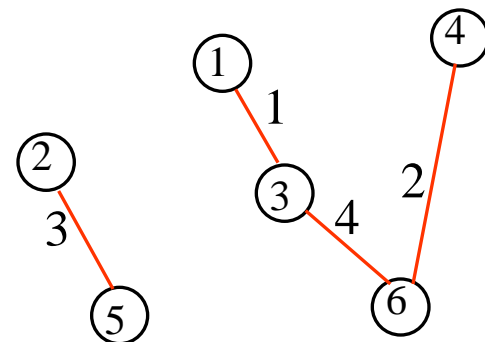
例如, 对图7-20 (a) 中无向网, 用克鲁斯卡尔算法求最小生成树的过程见图7-22。



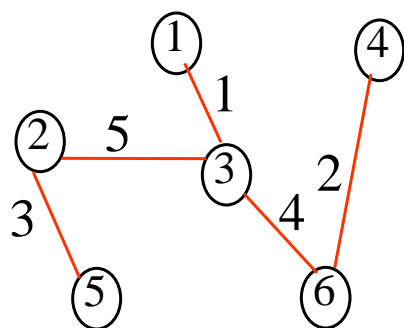
7.4.3 克鲁斯卡尔 (kruskal) 算法 (适合于边稀疏的网)



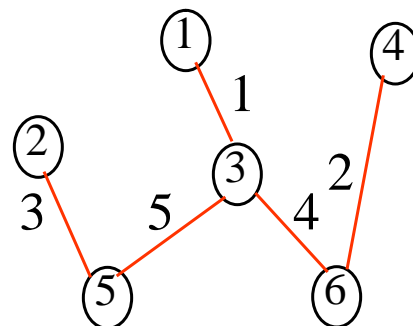
(c) 选第 3 条边



(d) 选第 4 条边



或者



(e) 第5条边不能选 (1, 4), 只能选 (2, 3) 或 (5, 3)

图7-22 克鲁斯卡尔方法求最小生成树的过程



7.4 最小生成树（无向图遍历的应用）

作业：

7.7

7.5.1 拓扑排序----有向无环图的应用

一个无环的有向图称作有向无环图（**Directed Acycling Graph**），简称**DAG**图，它在工程计划和管理方面有着广泛而重要的应用。判断一个工程是否能够有效进行，即对应于有向无环图的拓扑排序。

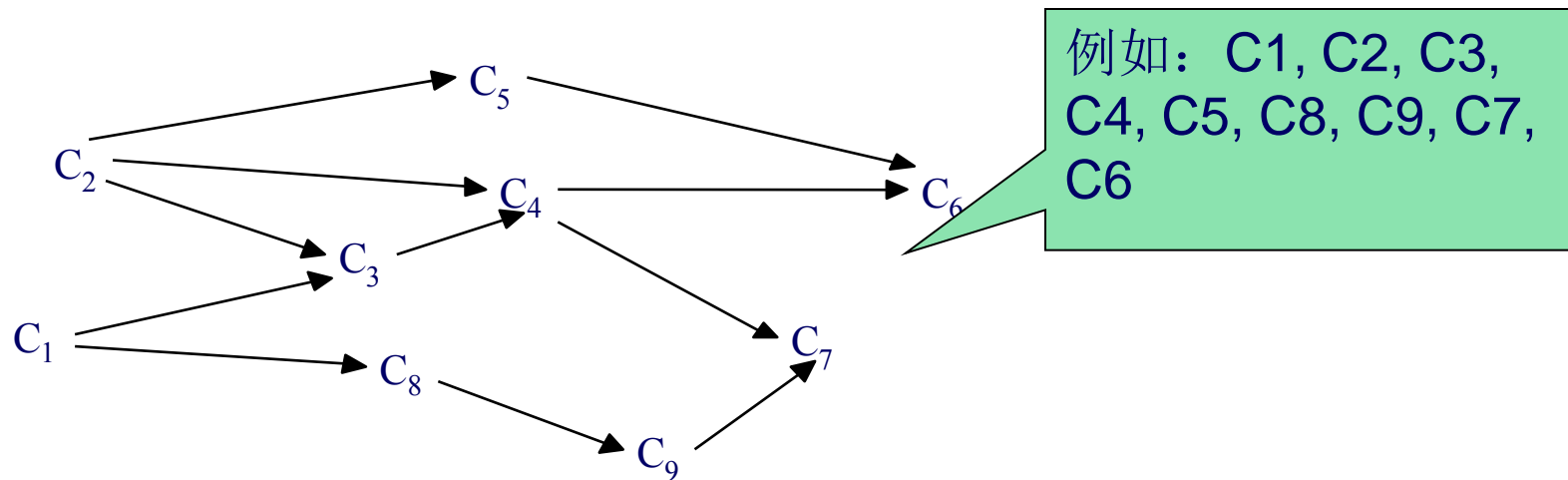
课程编号	课程名称	先修课程
C ₁	高等数学	无
C ₂	程序设计基础	无
C ₃	离散数学	C ₁ , C ₂
C ₄	数据结构	C ₂ , C ₃
C ₅	算法语言	C ₂
C ₆	编译技术	C ₄ , C ₅
C ₇	操作系统	C ₄ , C ₉
C ₈	普通物理	C ₁
C ₉	计算机原理	C ₈

软件专业必须学习的课程表

7.5.1 拓扑排序----基本概念

用顶点表示活动，用弧表示活动间的优先关系的有向图，称为顶点表示活动的网（**Activity On Vertex Network**），简称为**AOV-网**。

拓扑排序（**Topological Sorting**）用来分析AOV网络，将图中活动的优先次序以线性方式列出来。

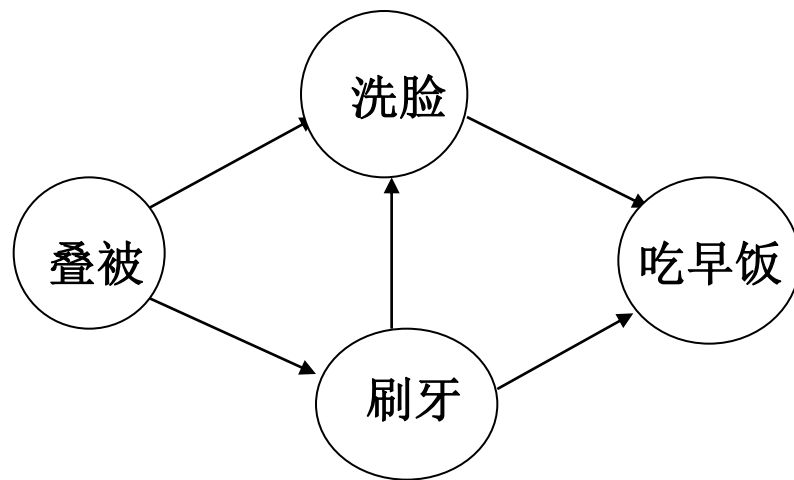


课程之间优先关系的有向无环图

7.5.1 拓扑排序----举例说明：早上工程

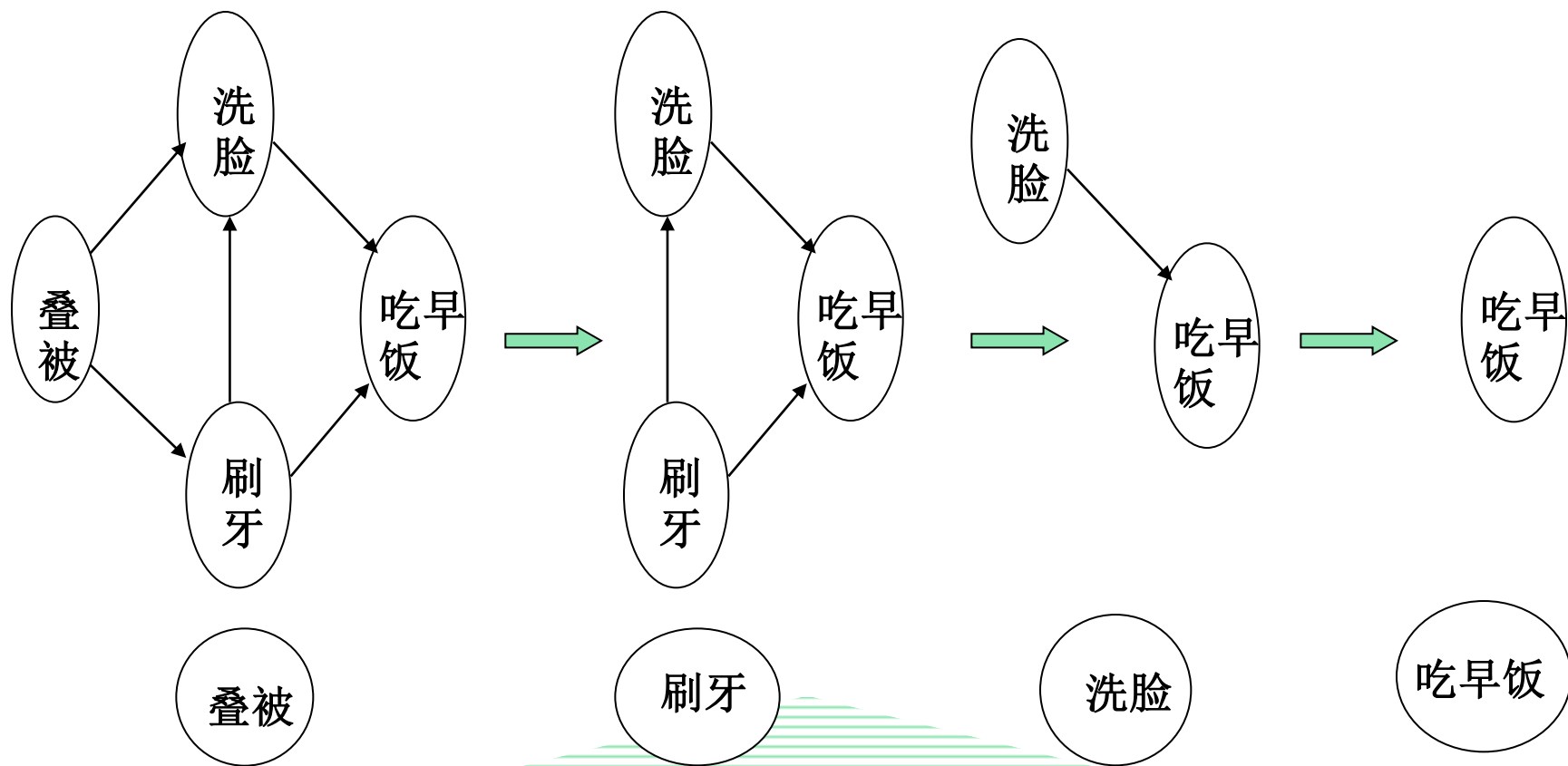
由四个子工程组成：

- 叠被
- 刷牙
- 洗脸
- 吃早饭



早上工程

7.5.1 拓扑排序----举例说明：早上工程



7.5.1 拓扑排序----举例说明：早上工程

规律：

1.被选入顶点的之前活动都已完成

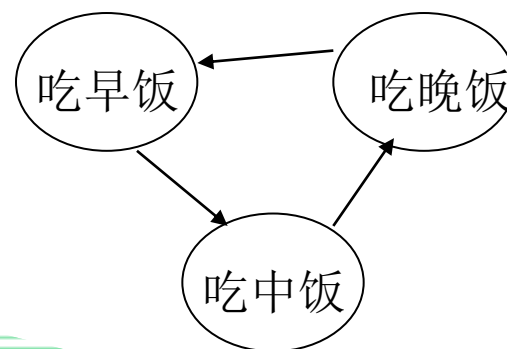
描述：顶点无前驱顶点，入度（indegree）为0

2.把与被选入顶点相关的先后关系都去掉

描述：从AOV网中删除此顶点及该顶点发出来的所有有向边；

3.并不是所有有向图都可以排出线性序列

描述：有环路不能排出拓扑序列



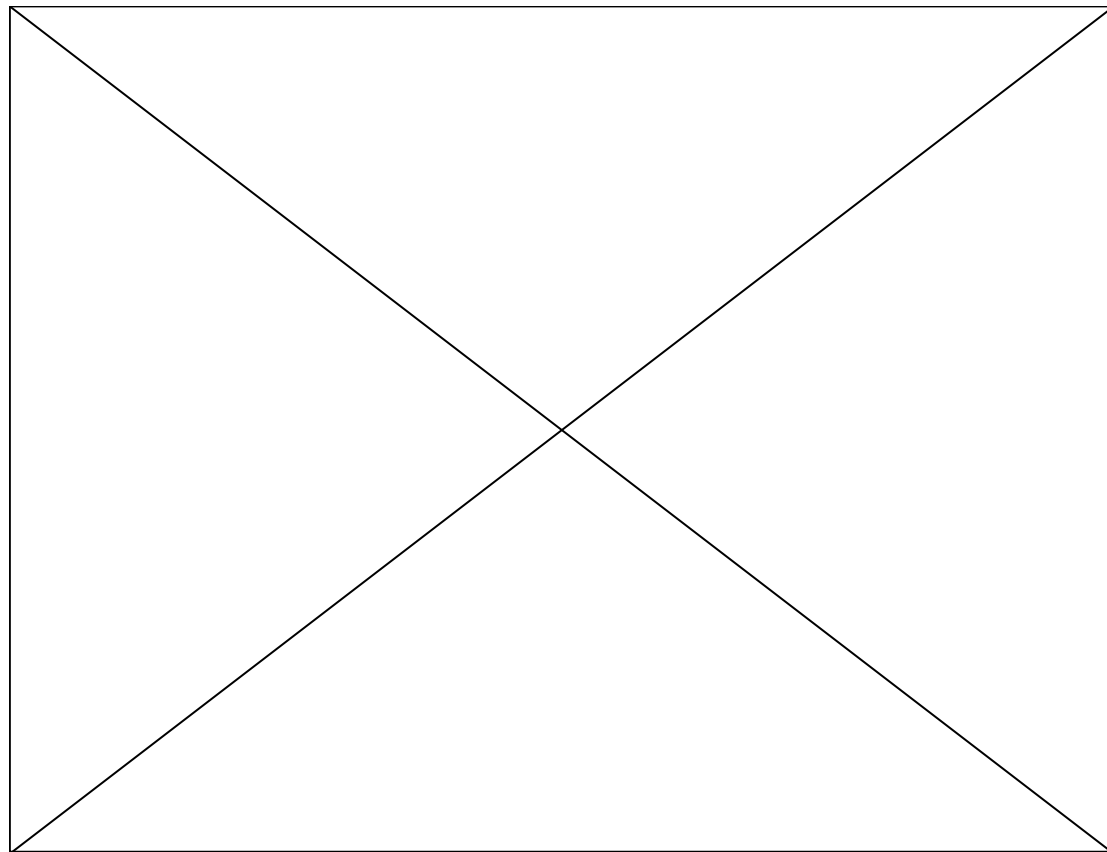
7.5.1 拓扑排序----基本思想

基本思想如下：

- (1) 在AOV网中选一个入度为0的顶点且输出之；
- (2) 从AOV网中删除此顶点及该顶点发出来的所有有向边；
- (3) 重复(1)、(2)两步，直到AOV网中所有顶点都被输出或网中不存在入度为0的顶点。



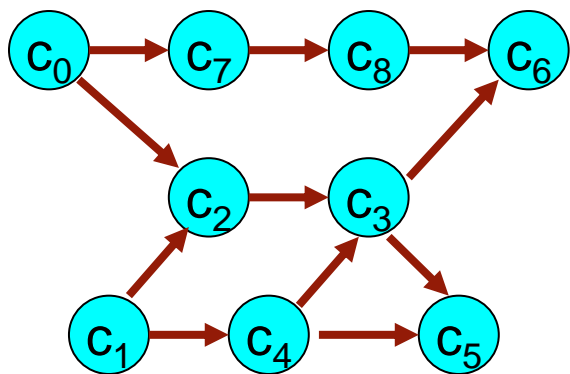
7.5.1 拓扑排序----规则应用演示



结果不唯一，
可能序列还有
很多

7.5.1 拓扑排序----储存方式的选择

有向图可选的存储方式有邻接表 (Adjacency List), 邻接矩阵 (Adjacency Matrix), 十字链表 (Orthogonal List), 由于有向图的存储形式的不同, 拓扑排序算法的实现也不同, 此处基于邻接矩阵表示的存储结构。



0	0	C ₀		→	2	→	7
1	0	C ₁		→	2	→	4
2	2	C ₂		→	3		
3	2	C ₃		→	5	→	6
4	1	C ₄		→	3	→	5
5	2	C ₅					
6	2	C ₆					
7	1	C ₇		→	8		
8	1	C ₈		→	6		

图的邻接表



7.5.1 拓扑排序----基于邻接表的存储结构的具体实现

■ 查找入度为零的顶点即为没有前驱的顶点。

附设一个存放各顶点入度的数组 $\text{indegree} []$ ，于是有无前驱的顶点，即为查找 $\text{indegree} [i]$ 为零的顶点。

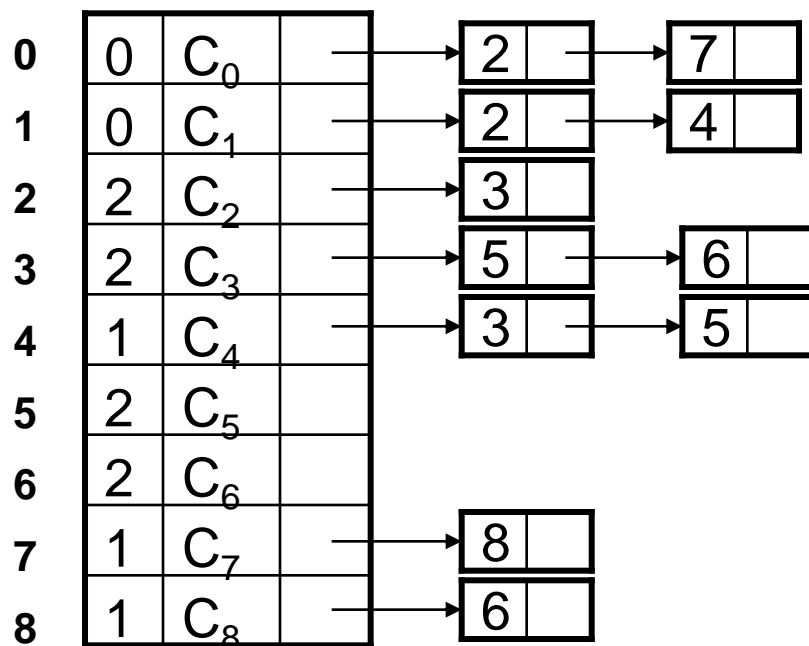
■ 删除以 i 为起点的所有弧。

既为对链在顶点 i 后面的所有邻接顶点 k ，将对应的 $\text{indegree} [k]$ 减1。

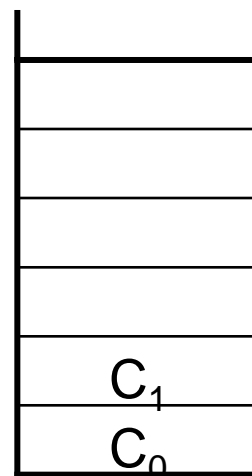
■ 避免重复检测入度为零的顶点。

可以设栈或队列。若为栈，若某一顶点的入度减为0，则将它入栈。每当输出某一入度为0的顶点时，便将它从栈中删除。

7.5.1 拓扑排序----基于邻接表的存储结构的具体实现

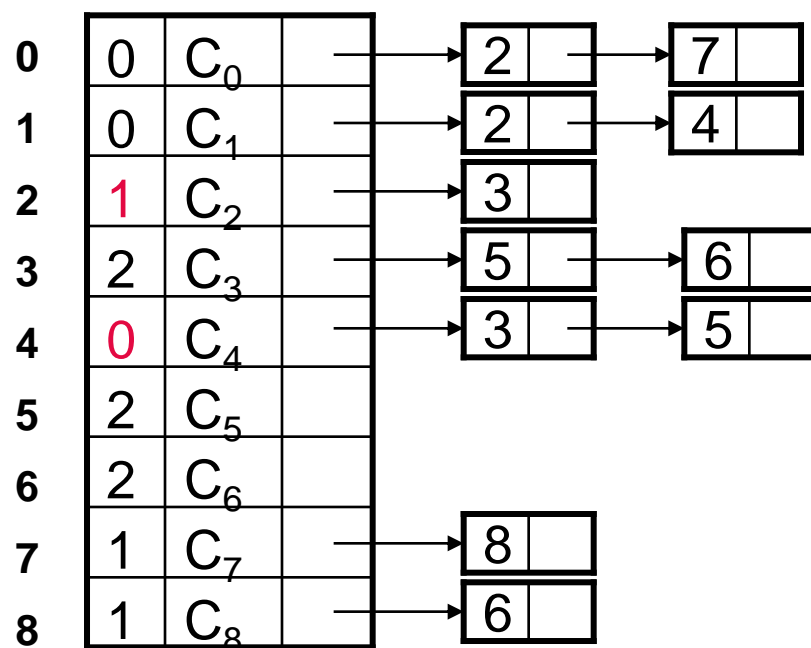


图的邻接表 (1)

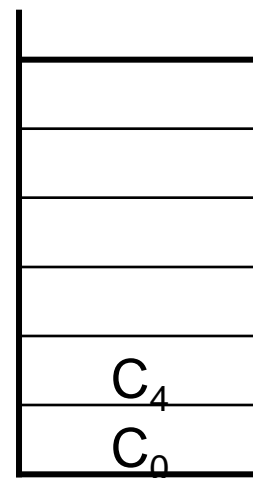


栈 (1) 弹出 C_1

7.5.1 拓扑排序----基于邻接表的存储结构的具体实现

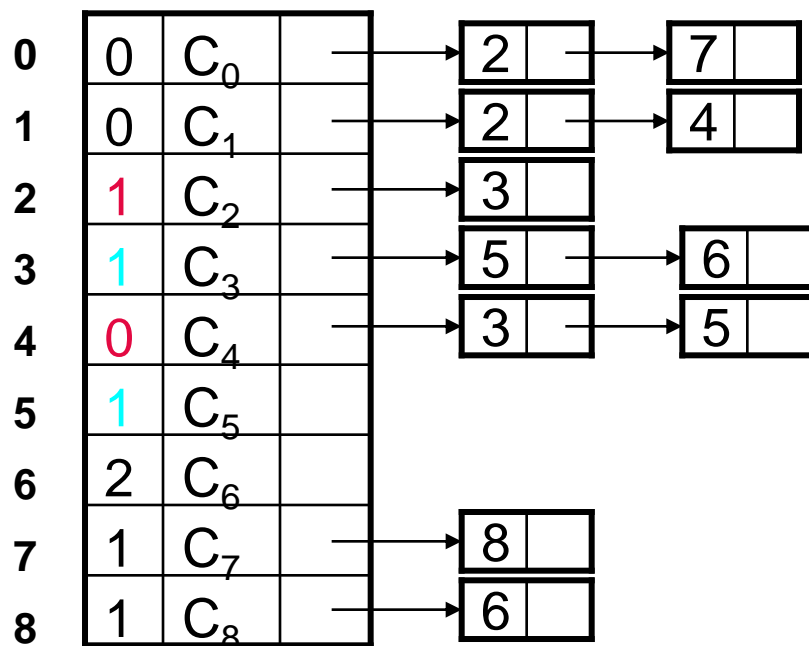


图的邻接表 (2)

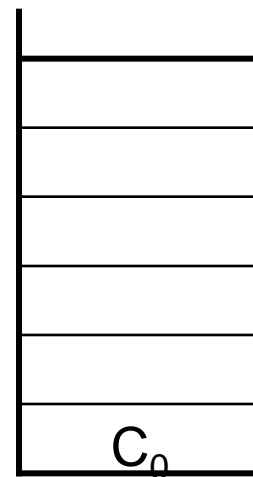


栈 (2) 弹出 C_4

7.5.1 拓扑排序----基于邻接表的存储结构的具体实现



图的邻接表 (3)



栈 (3) 弹出 C_0

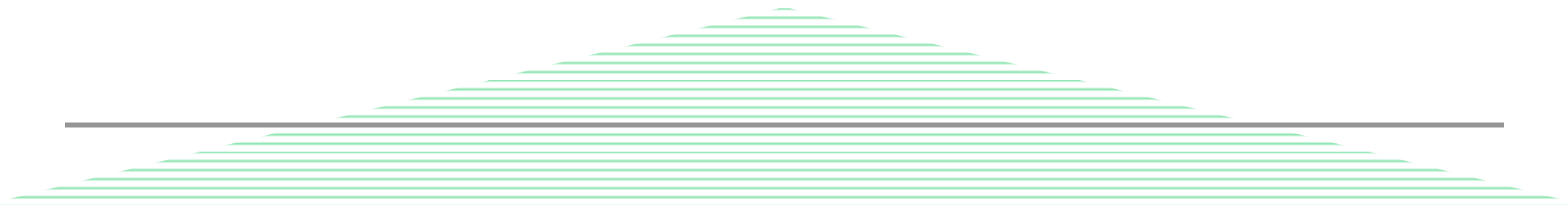
完整的拓扑序列为: $C_1, C_4, C_0, C_7, C_8, C_2, C_3, C_6, C_5$ 。



7.5.1 拓扑排序----基于邻接表的存储结构的具体实现

基于邻接表的拓扑排序算法描述如下：

其中A为有向图G的邻接表：

- (1) 初始化存放各顶点入度的数组`indegree []`，入度为0的顶点入栈
 - (2) 弹出栈顶顶点i，在A中对链在顶点i后面的所有邻接顶点k，将对应的`indegree [k]`减1，若`indegree [k]`为0，则k入栈。
 - (3) 当栈不为空，重复(2)，直到所有顶点都被输出,则算法结束，否则若不完全输出，存在环路。
- 

7.5.1 拓扑排序----基于邻接表的拓扑排序算法

```
Status TopologicalSort(ALGraph G){
    FindInDegree(G, indegree); //对各顶点求入度  indegree[0..vernum-1]
    initStack(S);
    for( i=0; i < G.vexnum; i ++ )
        if( ! indegree[i] ) Push(S,i); //入度为0者进栈
    count = 0;
    while( !StackEmpty(s) ){
        pop(s,i); printf(i, G.vertices[i].data); ++count;
        for(p=G.vertices[i].firstarc; p; p=p->nextarc){
            k=p->adjvex;    if(!( --indegree[k] )) Push(S, k);
        }//for
    }//while
    if(count<G.vexnum) return ERROR; //有向图有回路
    else return OK;
}//TopologicalSort
```

时间： $O(n+e)$




7.5.1 拓扑排序----总结

利用无前驱的顶点优先的拓扑排序算法求得**AOV**网的一个拓扑序列，成功得判断出一个由若干小工程组成的大工程是否能够顺利进行，也是学习无向图另一种应用----关键路径的基础。还有以下问题值得思考题：

- 基于邻接矩阵的存储结构的具体实现
- 无后继的顶点优先拓扑排序方法

作业：7.9

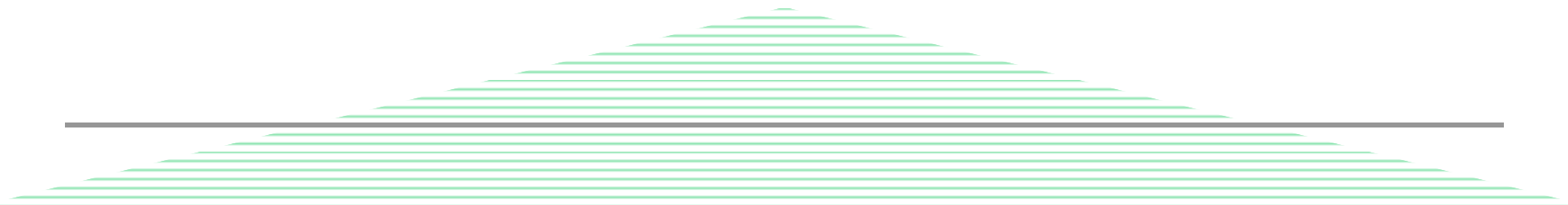





7.6最短路径

交通网络中常常提出这样的问题：从甲地到乙地之间是否有公路连通？在有多条通路的情况下，哪一条路最短？交通网络可用带权图来表示。顶点表示城市名称，边表示两个城市有路连通，边上权值可表示两城市之间的距离、交通费或途中所花费的时间等。求两个顶点之间的最短路径，不是指路径上边数之和最少，而是指路径上各边的权值之和最小。

三种常用的最短路径问题：

- 由某个固定接点到另一个结点的最短路径（一对一）
 - 由某个固定接点到其他各结点的最短路径（一对多）
 - 由各结点到其他各结点的最短路径（多对多）
- 



1. 单源点最短路径

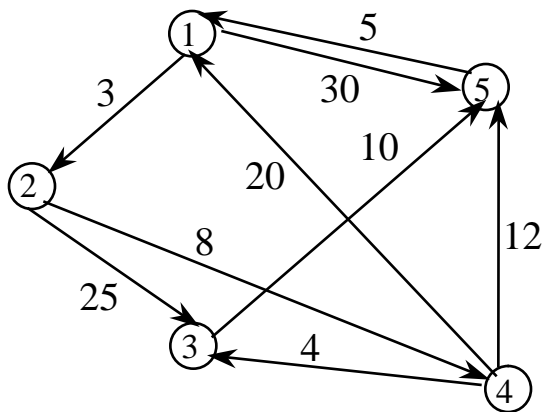
单源点最短路径是指：给定一个出发点(单源点)和一个有向网 $G=(V, E)$,求出源点到其它各顶点之间的最短路径。

迪杰斯特拉(Dijkstra)在做了大量观察后,首先提出了按路径长度递增序产生各顶点的最短路径算法,我们称之为迪杰斯特拉算法。

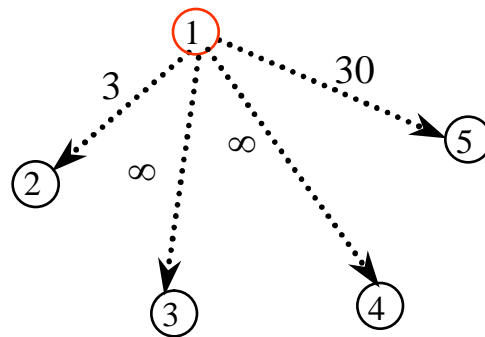
2. 迪杰斯特拉算法的基本思想

算法的基本思想是:设置并逐步扩充一个集合 S , 存放已求出其最短路径的顶点, 则尚未确定最短路径的顶点集合是 $V-S$, 其中 V 为网中所有顶点集合。按最短路径长度递增的顺序逐个以 $V-S$ 中的顶点加到 S 中, 直到 S 中包含全部顶点, 而 $V-S$ 为空。

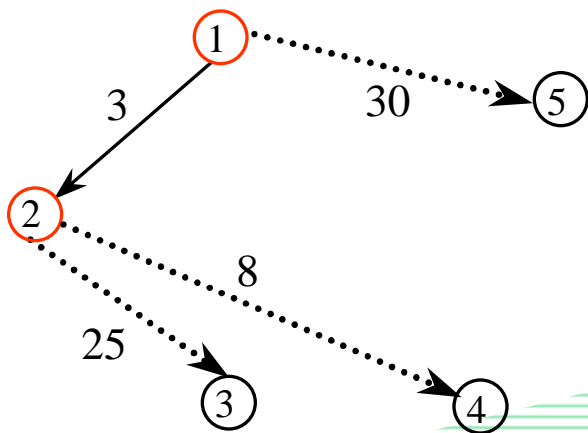
迪杰斯特拉算法的求解过程



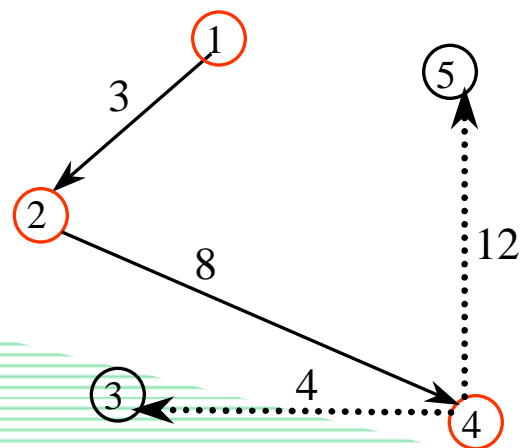
(a) 一个有向网点



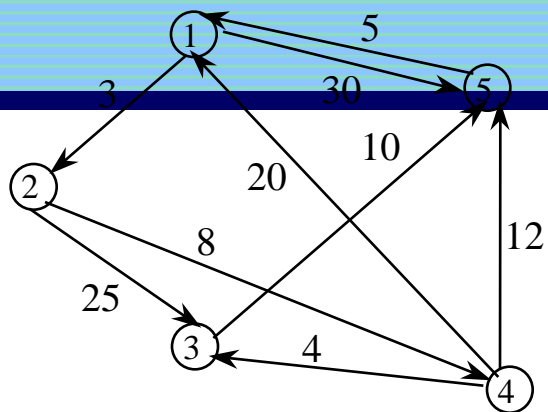
(b) 源点 1 到其它顶点的初始距离



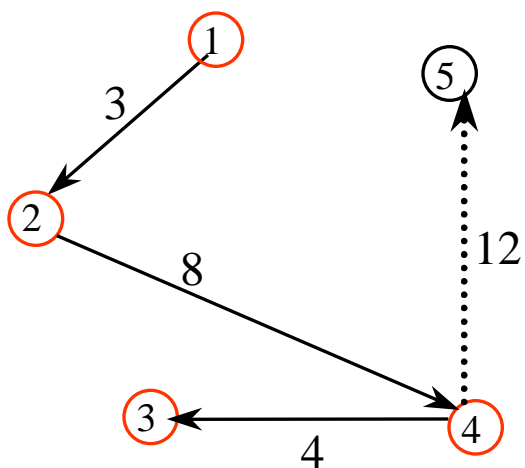
(c) 第一次求得的结果



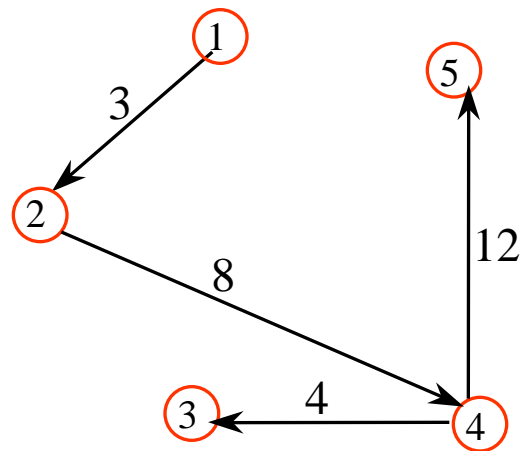
(d) 第二次求得的结果



(a) 一个有向网点



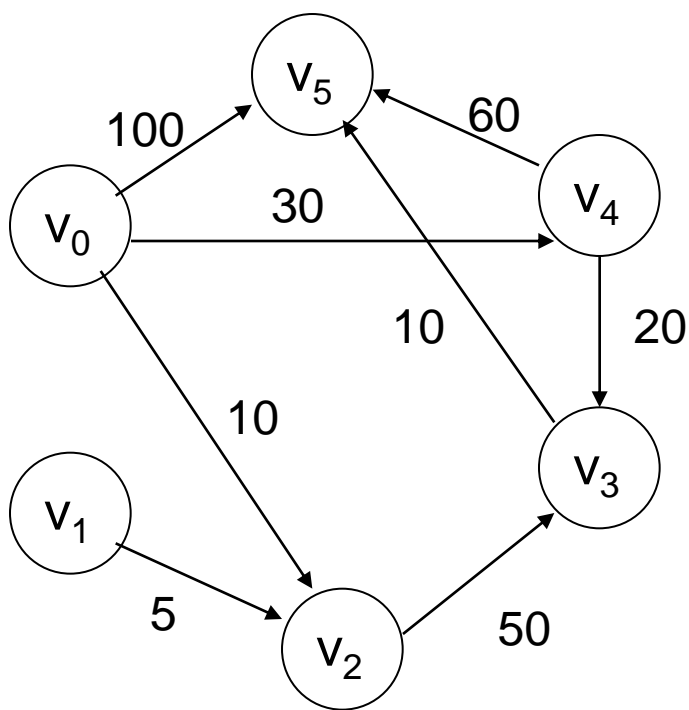
(e) 第三次求得的结果



(f) 第四次求得的结果

图 7-27 迪杰斯特拉算法求最短路径过程及结果

举例:求 V_0 到其余各点的最短路径



三个辅助向量:

1. $D[i] = \begin{cases} \langle v_0, v_j \rangle \text{的权值} & \langle v_0, v_j \rangle \in E \\ \infty & \text{否则} \end{cases}$

2. $p[i]$ 代表一个集合, 表示当前找到的, 从 v_0 到 v_i 的最短路径

3. $Final[]$ 已经求到最短路径的结点的集合



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

D[] P[]						Final	V
[0]	[1]	[2]	[3]	[4]	[5]		
∞ ()	∞ ()	10✓ (v0,v2)	∞ ()	30 (v0,v4)	∞ (v0,v5)	[v0]	
∞	∞		60 (v0,v2,v3)	30✓ (v0,v4)	100 (v0,v5)	[v0,v2]	2
∞	∞		50✓ (v0,v4,v3)		90 (v0,v4,v5)	[v0,v2,v4]	4
∞	∞				60✓ (v0,v4,v3,v5)	[v0,v2,v4, v3]	3
∞	∞					[v0,v2,v4, v3,v5]	5



3. 迪杰斯特拉算法

```
void ShortestPath_DLJ(Mgraph G,int v0,PathMatrix *p, ShortPathTable *D){  
    //用Dijkstra算法求有向网G的v0顶点到其余顶点v的最短路径P[v]及其路径长度D[v]  
    //若P[v][w]为TRUE, 则w是从v0到 v当前求得最短路径上的顶点  
    //final[v] 为TRUE当且仅当 $v \in S$ , , 即已经求得从v0到v的最短路径  
    //常量INFINITY为边上权值可能的最大值  
    for (v=0;v<G.vexnum;++v){  
        fianl[v]=FALSE; D[v]=G.edges[v0][v];  
        for (w=0; w<G.vexnum; ++w) P[v][w]=FALSE; /*设空路径*/  
        if (D[v]<INFINITY) {P[v][v0]=TRUE; P[v][w]=TRUE;}  
    }  
    D[v0]=0; final[v0]=TRUE;          /*初始化, v0顶点属于S集*/
```

3. 迪杰斯特拉算法

```
/*开始主循环，每次求得v0到某个v 顶点的最短路径，并加v到S集*/
for(i=1; i<G.vexnum; ++i)      /*其余G.vexnum-1个顶点*/
{min=INFINITY;                /*min为当前所知离v0顶点的最近距离*/
  for (w=0;w<G.vexnum;++w)
    if (!final[w])             /*w顶点在V-S中*/
      if (D[w]<min) {v=w; min=D[w];}
  final[v]=TRUE                /*离v0顶点最近的v加入S集合*/
  for(w=0;w>G.vexnum;++w)      /*更新当前最短路径*/
    if (!final[w]&&(min+G.edges[v][w]<D[w])) /*修改D[w]和P[w],w∈V-S*/
      { D[w]=min+G.edges[v][w];
        P[w]=P[v]; P[w][v]=TRUE;          /*P[w]=P[v]+P[w]*/
      }
}
}
}/*ShortestPath._1*/
```




作业：

7.11

