

## 第 7 章 图结构

这里的图，有时也称网络，是指由若干个结点与若干条边构成的结构，其中每个结点可有多个前趋和多个后继，结点是一些具体对象的抽象，而边是对象间的关系的抽象。值得注意的是，与一般意义下的图不同，这里的图只涉及图的拓扑结构，与图的几何性质无关。这种意义下的图就是图论中讨论的图。但在图论中，重点讨论图的数学性质，而这里的重点是图结点间的关系以及图的存贮结构与基本操作的实现。

图是一种复杂的非线性结构，它有极强的表达能力，现实世界中许多问题均可用图结构描述。

### § 7.1 图的基本概念

#### § 7.1.1 图的概念

这里先给出关于图的几个概念。

##### 1. 图

图(Graph)是由若干个结点和若干条边构成的结构，每个结点具有任意多个前驱和后继。这种结构的含意为：

结点集合：结点代表对象

边集合：两结点之间的边表示它们代表的对象之间的关系

在具体应用中，结点与边要被赋予具体的含意。如结点代表城市，而边代表城市之间的行程。

形式化地讲，图是形为

$$G=(V, R)$$

的数据结构，其中，

$$V=\{x|x \text{ 属于数据对象}\}$$

$$R=\{VR\}$$

$$VR=\{\langle x,y \rangle \mid p(x,y) \wedge x \in V \wedge y \in V\}$$

这里， $p(x,y)$ 是  $V$  上的一个谓词， $p(x,y)$ 为真当且仅当  $x$  与  $y$  存在问题世界中的关系。

## 2 有向/无向图

若二元关系  $VR$  中的每个型如  $\langle x, y \rangle$  的成员中的  $x$  与  $y$  是次序无关的，则称该图无向图(undirected graph)，否则称为有向图(directed graph)。无向图也可以看作  $VR$  对称的图，即对任意  $x$  与  $y$ ，若有  $\langle x, y \rangle \in VR$ ，则必有  $\langle y, x \rangle \in VR$ 。对无向图，我们用  $(x, y)$  表示  $\langle x, y \rangle$ 。

## 3. 结点、边、弧

图中的数据元素称为**结点**（或**顶点**）(vertex/node/point)，有时为了强调，对有向图，称  $\langle x, y \rangle$  为**弧(arc)**， $x$  与  $y$  分别为**弧尾**与**弧头**，或**始点**与**终点**，称  $y$  为  $x$  的**出点/可达邻接点**，称  $x$  为  $y$  的**入点**，称  $\langle x, y \rangle$  为  $x$  的**出边/出弧**， $\langle x, y \rangle$  为  $y$  的**入边/入弧**。对无向图，泛称  $\langle x, y \rangle$  为**边(edge)**。

在讨论图中，为了说话方便，一般给结点编号，用它们的编号代表它们。结点编号一般用自然数。

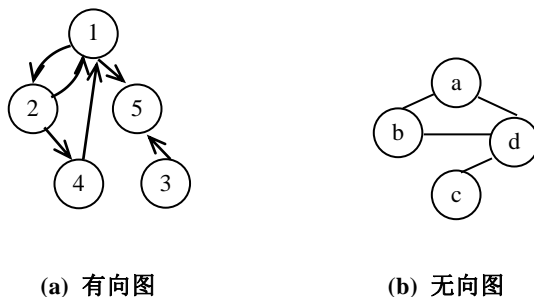


图 7-1 图示例

**例 7-1** 设有两个二元组  $G_1=(V_1, R_1)$  与  $G_2=(V_2, R_2)$ ，其中，

$$V_1=\{1, 2, 3, 4, 5\}$$

$$R_1=\{VR_1\}$$

$$VR_1=\{\langle 1, 2 \rangle, \langle 1, 5 \rangle, \langle 2, 1 \rangle, \langle 2, 4 \rangle, \langle 3, 5 \rangle, \langle 4, 1 \rangle\}$$

$$V_2=\{a, b, c, d\}$$

$$R_2=\{VR_2\}$$

$$VR_2=\{(a, b), (a, d), (b, d), (d, c)\}$$

显然，它们不满足线性结构、广义表和树的定义，而满足图的定义，其图示如图 7-1 所示。

## 4. 邻接、关联

对图中任意结点  $x$  与  $y$ ，若它们之间存在边（即有  $\langle x, y \rangle$ ，或  $\langle y, x \rangle$ ），则称  $x$  与  $y$  邻

接(Adjacent), 它们互为邻接点。同时称  $x$  或  $y$  与边  $\langle x, y \rangle$  (或  $\langle y, x \rangle$  或  $(x, y)$ ) 关联(Incident)。

### 5. 度

对任一结点  $x$ , 称以它为头的弧的个数为它的**入度(In degree)**; 称以它为尾的弧的个数为它的**出度(Out Degree)**; 称它的入度与出度之和为它的**度(Degree)**。对无向图, 无出度入度之分, 直接称与它关联的边的个数为它的**度**。

例如, 图 6-1(a)中的结点 1 的出度与入度都为 2, 结点 3 的出度入度分别为 1 和 0, (b)中的结点 a、b、d 的度均为 2, 而 c 的度为 1。

### 6. 权

权(Weight)与哈夫曼树中的概念相同, 即权是一个数值量, 是某些信息的数字化抽象。权分边权与结点权, 分别是边与结点的问题世界所关心的信息的数值化表示。

例如, 若结点代表城市, 则边权可代表城市之间的交通费用。

### 7. 路径(通路)

对有向图, 若存在弧序列

$$\langle x_0, x_1 \rangle, \langle x_1, x_2 \rangle, \dots, \langle x_{n-1}, x_n \rangle$$

且  $n \geq 1$ , 则称从  $x_0$  到  $x_n$  有通路(路径)(Path)。上列序列称为  $x_0$  到  $x_n$  的路径。该路径也可表示为

$$(x_0, x_1, \dots, x_n)$$

对无向图, 若有边序列

$$(x_0, x_1), (x_1, x_2), \dots, (x_{n-1}, x_n)$$

且  $n \geq 1$ , 则称  $x_0$  与  $x_n$  之间有路径(通路), 该路径可用上列边序列表示, 亦可用下列结点序列表示

$$(x_0, x_1, \dots, x_n)$$

路径中边的数目称为**路径长**。

若  $x_0 = x_n$ , 则称相应的路径为**回路/环路(loop)**。

若在路径  $(x_0, x_1, \dots, x_n)$  中, 除  $x_0$  与  $x_n$  外, 任意其它结点均不相同, 则称该路径为**简单路径(Simple Path)**。起点与终点重合的简单路径称为**简单回路(Simple Loop)**。

显然, 简单路径中不含回路。

### 8. 子图/生成图

若某图  $G'$  的结点集合与边集合分别是另一图  $G$  的结点集合和边集合的子集, 则称  $G'$  为  $G$  的**子图(Subgraph)**。

设有两个图  $G$  与  $G'$ , 若它们的结点集合相同, 但  $G'$  的边集合是  $G$  的边集合的子集, 则称  $G'$  是  $G$  的**生成图(Spanning Graph)**。

显然, 生成图一定是子图, 但反之未必。

## 9. 连通

若图中任意两结点  $x$  和  $y$ ，或  $x$  到  $y$  有通路，或  $y$  到  $x$  有通路，则称该图是（弱）**连通的**(Connected)；若  $x$  到  $y$  有通路，且  $y$  到  $x$  有通路，则称该图是**强连通的**(Strongly Connected)。若图中某子图是连通的，则称该子图为一个**连通分量**(Connected Component)。若  $G$  的某子图  $G'$ 是连通的，并且，若将  $G$  中任意其他结点及其任意相关联的边加入到  $G'$ ，都会导致  $G'$ 不连通，那么称  $G'$ 是一个**极大连通分量** (Maximum Connected Component)。显然极大连通分量不唯一。

## 10. 生成树

无回路的连通的无向图的生成图，称为该无向图的生成树(Spanning Tree)。

**定理 7-1** 生成树中边的个数为结点数减 1。

证：首先，可归纳证明，边的个数不能少于结点数减 1（否则图就不连通了）。另一方面，边数必不大于结点数减 1，因为若有  $n$  个结点，则有  $(n-1)$ 条边就使这  $n$  个结点连通了，若再添加一条边，则这条边关联的两结点间的路径就多于 1 条了，从而构成回路。综合这两个方面，就证得了本定理。

### § 7.1.2 图的基本操作

图是一种与具体应用密切相关的结构，有关它的基本操作也往往随应用不同而出入很大，这里仅作一个一般性介绍。

#### ①检索：

- 按结点（编号或内容）检索结点（位置）
- 按边的两顶点检索边

#### ②插入：

- 插入结点
- 插入边

#### ③删除：

- 删除结点及与其关联的边
- 删除边

#### ④求邻接点、关联边

#### ⑤求度（入度与出度）

#### ⑥求路径

#### ⑦图的遍历

#### ⑧求子图

#### ⑨求生成图

#### ⑩求连通分量

#### (11)求拓扑序列（有向图）

- (12) 求（最小）生成树
- (13) 求关键路径
- (14) 求最短路径

## § 7.2 图的对象抽象模型

这里，我们考虑仅从逻辑关系出发时，图结点和整个图对象应具有的性质。由于图结点的关系不象其他结构中结点那样有限制，所以它的抽象结构比较复杂。

为了后面的使用，首先定义一个枚举类型和常量：

```
enum TTraverseMode {DFS, WFS}; //定义枚举类型，表示图遍历方式
const CNST_NumNodes=100; //定义常量，表示最大结点个数。实际结点个数可以小于该数
```

### § 7.2.1 图结点抽象模型

图结点抽象模型重点是描述图结点的内容，隐蔽具体的图结点的结构。所以，图结点对象模型中，重点是关于结点信息的 Get 和 Set 类操作。

下面是图结点的描述：

```
template <class TElem> //结点内容的类型（可变类型）
struct TGraphNode0 //图结点类
{
    long no; //图结点的编号
    float weight; //结点的权
    TElem info; //图结点的内容

    virtual long GetNo() {return no;};
    virtual void SetNo(long mNo) {no = mNo;};
    virtual TElem& GetElem() {return info;};
    virtual TElem& SetElem(TElem &e){ info = e; return info;};
    virtual float GetWeight() {return weight;};
    virtual float SetWeight(TElem &w) { weight = w; return weight;};
};
```

该类的各基本操作都已在类定义中实现，所以已不属于抽象类，但其在整个图类中，扮演抽象类的作用。该类中主要成员说明如下：

GetNo(): 返回本结点的编号。

SetNo(long mNo): 将 mNo 做为本结点的编号值。

GetElem(): 返回本结点的值。

SetElem(TElem &e): 将 e 做为本结点的元素值。

GetWeight(): 返回本结点的权值。

SetWeight (TElem &w): 将 w 做为本结点的权值。

### § 7.2.2 图的边对象抽象模型

边实质上就是元素间的关系。在图中，边不仅仅表达的是逻辑关系，它本身也可以具有其他的属性，如权值。因此，有必要专门建立针对图的边的对象模型。

```
template <class TElem>
struct TGraphEdge0 //图边抽象类
{
    long startNo, endNo; //边的起点和终点
    float weight; //边权
    TElem info; //边的内容

    virtual TElem &GetElem() { return info; }; //读边内容
    virtual TElem &SetElem(TElem &e) { info = e; return info; }; //置边内容
    virtual float GetWeight() { return weight; }; //读边权
    virtual float SetWeight(TElem &w) { weight = w; return weight; }; //置边权

    virtual long GetStartNo() { return startNo; }; //读边的起点编号
    virtual long GetEndNo() { return endNo; }; //读边的终点编号

    virtual void SetStartNo(long mNo) { startNo = mNo; }; //置边的起点编号
    virtual void SetEndNo(long mNo) { endNo = mNo; }; //置边的终点编号
};

struct TDijkPath
{ //存放路径的数组的元素类型
    float len;
    long pre;
};
```

与结点类 TGraphNode0，该类的成员函数也都实现，在语法上不属于抽象类。

### § 7.2.3 图抽象对象模型

图抽象模型重点是定义针对图的整体结构的操作。根据前面分析，图的整体方面的操作主要有遍历、查找、求路径、求生成树、求拓扑序列（有向图）、求关键路径、求最短路径、求最小生成树等。下面是具体的抽象模型。

```

template <class TElem, class TEdgeElem> //用到两个可变类型，分别是图结点内容和边内容
class TGraph0 //图抽象类
{
public:
    long numNodes;

    virtual long GetNodeNo(TGraphNode0<TElem> *pNode);
    virtual TGraphNode0<TElem> *GetNodeAddress(long nodeNo);

    virtual TGraphNode0<TElem> *GetSucc(TGraphNode0<TElem> *pNode,
                                        long sucNo)=0;
    virtual TGraphNode0<TElem> *GetPrior(TGraphNode0<TElem> *pNode,
                                        long preNo)=0;
    virtual TGraphNode0<TElem> *SetSucc(TGraphNode0<TElem> *pNode,
                                        long sucNo, TGraphNode0* pNode)=0;
    virtual TGraphNode0<TElem> *SetPrior(TGraphNode0<TElem> *pNode,
                                        long preNo, TGraphNode0* pNode)=0;
    virtual TGraphEdge0<TEdgeElem> *GetSuccEdge(TGraphNode0<TElem>*pNode,
                                                long sucNo)=0;
    virtual TGraphEdge0<TEdgeElem> *GetPriorEdge(TGraphNode0<TElem> *pNode,
                                                long preNo)=0;

    virtual long GetInDegree(TGraphNode0<TElem> *pNode)=0;
    virtual long GetOutDegree(TGraphNode0<TElem> *pNode)=0;

    virtual long Cluster(long v0, long *nos, TTraverseMode tm)=0;
    virtual long GetNumNodes() {return numNodes;};

    virtual TGraphNode0<TElem> *AddNode(long mNo, TElem &nodeInfo) =0;
    virtual TGraphEdge0<TEdgeElem> *AddEdge(long startNo, long endNo,
                                            TEdgeElem *pee=NULL)=0;
    virtual TGraphNode0<TElem> *DeleteNode(long mNo) =0;
    virtual TGraphEdge0<TElem> *DeleteEdge(long startNo, long endNo) =0;

    virtual long Cluster(TGraphNode0<TElem>* pStartNode, TElem **e,
                        TTraverseMode tm)=0;
    virtual long Cluster(TGraphNode0<TElem>* pStartNode,

```

```

TGraphNode0<TElem> ** Nodes, TTraverseMode tm)=0;

virtual TGraphNode0<TElem> *Locate(long nNo)=0;
virtual TGraphEdge0<TEdgeElem> *Locate(long startNo, long endNo)=0;
virtual TGraphNode0<TElem> *Locate(TElem &e,
                                TTraverseMode tm,long sn=1)=0;
virtual long GetPath(long startNo, long endNo, long sn, long *pathNodeNo) =0;
virtual long GetPath(long startNo, long endNo, long sn,
                    TGraphNode0<TElem> ** pathNode) =0;
virtual long GetPathShortest(long startNo, TDijkPath *path) =0;
virtual long GetPathShortest(long **path) =0;
};

```

其中主要成员说明如下。

**GetNodeNo(TGraphNode0<TElem> \*pNode):** 返回地址为pNode的结点的编号。

**GetNodeAddress(long nodeNo):** 返回编号为 nodeNo 的结点的地址。

有了这两个函数，就可以自由地由结点的编号，求得其地址，或反过来。

**GetSucc(long nodeNo, long sucNo):** 返回编号为 nodeNo 的结点的第 sucNo 个后继结点的地址。

**GetPrior(long nodeNo, long preNo):** 返回返回编号为 nodeNo 的结点的第 preNo 个前驱结点的地址。

**SetSucc(long nodeNo, long sucNo, TGraphNode\* pNode):** 将编号为 nodeNo 的结点设置为本结点的第 sucNo 个后继。

**SetProir(long nodeNo, long preNo, TGraphNode\* pNode):** 将编号为 nodeNo 的结点设置为本结点的第 preNo 个前驱。

**GetSuccEdge(long nodeNo, long sucNo):** 返回编号为 nodeNo 的结点的第 sucNo 个结点的地址。:

和 **GetPriorEdge(long nodeNo, long sucNo)**。

**GetInDegree(long nodeNo):** 返回编号为 nodeNo 的结点的入度值。

**GetOutDegree(long nodeNo):** 返回编号为 nodeNo 的结点的出度值。

**AddNode(mNo, nodeInfo):** 给图加入一个结点。结点的编号为 mNo，值为 nodeInfo。

**AddEdge(startNo, endNo, pee):** 给图增加一条边。边的起点和终点分别为 startNo 和 endNo，边的内容为 pee 所指内容，其为空时表示暂不设置边内容。当 startNo 或 endNo 不存在时，该函数还负责加入相应的结点。

**DeleteNode(mNo) :** 删除编号为 mNo 的结点，并将其值存储在一个临时的缓冲区，返回该缓冲区的地址。

**DeleteEdge(startNo, endNo):** 删除两端点的编号分别为 startNo 和 endNo 的边，并将



其值存储在一个临时的缓冲区，返回该缓冲区的地址。

**Cluster(long v0,TElem \*\*e,TTraverseMode tm):** 串行化聚集：从结点 v0 出发，按 tm 方式遍历图，将遍历到的结点的内容的地址依次存于 e 中，并返回所遍历到的结点的个数。

**Cluster(long v0, long \*nos, TTraverseMode tm):** 与上面的函数类似，不同的是将遍历到的结点的编号存于 nos 中。

**Locate(long startNo, long endNo):** 边定位：返回两端点编号分别为 startNo 和 endNo 的边的指针。

**Locate(TElem &e, TTraverseMode tm,long sn=1):** 结点内容定位：返回结点内容为 e 的（在遍历方式 tm 下的）第 sn 个结点的指针。sn>0 时表示正数（第一次遍历到的内容为 e 的结点的对应 sn 为 1，余递增），sn<0 时表示倒数（最后遍历到的内容为 e 的结点的对应 sn 为-1，余为-2,...）。

**GetPath(long startNo, long endNo, long sn, long \*pathNodeNo) :** 求路径：求从 startNo（编号）到 endNo(编号)的第 sn 条路径，并将其上结点依次存入 pathNodeNo 中。这里，所谓路径的次序（第 sn 条），是按从 startNo 到 endNo 的深度优先次序。sn 的含义同前。

**GetPath(long startNo, long endNo, long sn, TGraphNode0<TElem>\*\* pathNode) :** 与上面的函数类似，不同的是将路径上的结点的指针存入 pathNode。

**GetPathShortest(long startNo, TDijkPath \*path):** 生成单源最短路径（从 startNo 到其余各点的最短路径）。

**GetPathShortest(long \*\*path) :** 生成多源最短路径（每对结点讲间的最短路径）。

关于这些抽象操作的实现，要在介绍了图的具体存储结构后才能进行。

## § 7.3 图的存贮结构

因为在图中，任意一元素都可能有多多个前趋与多个后继，所以无法通过元素的存贮次序反映元素间的关系，而必须显式地存贮结点之间的关系（即边）的信息。一般有两类方法：

- 用边的集合表示图；
- 用链接关系表示图。

邻接矩阵法就属于第一类，而邻接表、十字链表及邻接多重表等属于第二类。

### § 7.3.1 邻接矩阵法

#### (一) 存贮方法

邻接矩阵(Adjacency Matrix)这是一种用边的集合表示图的方法。其中边集合用一个

二维数组表示，数组每个元素表示一条边，它的两个下标分别表示边的两个端点的编号。结点编号要用连续的自然数，使得能与数组下标对应。

具体地讲，若用二维数组  $A$  表示图，则  $A[i][j]$  表示边  $\langle i, j \rangle$  或  $(i, j)$ 。边一般要用记录表示，以描述与边相关的各种信息。前面我们给出了边的抽象定义  $TGraphEdge0$ ，对邻接矩阵， $TGraphEdge0$  包含一般需求下的内容，故邻接矩阵的边的定义可以是  $TGraphEdge0$  的直接派生（未增加新成员）：

```
template <class TEdgeInfo>
struct TGraphEdgeMatrix: public TGraphEdge0 <TEdgeInfo>
{ public:
//邻接矩阵边， TGraphEdge0 中的函数均已实现，故这里不重载时不需给出。
};
```

邻接矩阵结点的定义也类似：

```
template <class TElem>
struct TGraphNodeMatrix : public TGraphNode0<TElem> //邻接矩阵结点抽象类
{
// TGraphNode0 中的函数均已实现，故这里不重载时不需给出。
//下面可以声明 TGraphNodeMatrix 中特有的函数
};
```

在这种情况下，图类可定义为：

```
template <class TElem, class TEdgeElem> //用到两个可变类型，分别是图结点内容和边内容
class TGraphMatrix: public TGraph0<TElem, TEdgeElem>
{ //邻接矩阵类
public:
    TGraphEdgeMatrix<TEdgeElem> edges[][CNST_NumNodes]; //存放图边的矩阵
    TGraphNodeMatrix<TElem> nodes[CNST_NumNodes]; //存放图结点的一维数组

    //下面是其他成员的声明，同 TGraph0，此处略(实际上机时不可略)
};
```

如果图结点个数变化很大，则边矩阵可以考虑用动态存储区（动态（可调）二维数组）。由于标准的 C/C++ 不直接支持该功能，所以需要 `new` 申请一块连续空间，然后将其封装在一个类中，设置二维数组访问接口。对结点数组，也要类似处理。具体实现，留作练习。

显然，对邻接矩阵，边描述  $TGraphEdgeMatrix$  中的起点和终点信息是冗余的，所以可以适当简化，例如，可以只用一个数表示边，即

$$\text{edges}[i][j] = \begin{cases} \text{边 } \langle i, j \rangle \text{ 的权: 若 } \langle i, j \rangle \text{ 是边} \\ \text{特殊标志} & \text{若 } \langle i, j \rangle \text{ 不是边} \end{cases}$$

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	0	1	0
3	0	0	0	0	1
4	1	0	0	0	0
5	0	0	0	0	0

	a	b	c	d
1	0	1	0	1
2	1	0	0	1
3	0	0	0	0
4	1	1	0	0

图 7-2 图 7-1 所示图的邻接矩阵

若不考虑边的权, 则  $\text{edges}[i][j]$  只表示边  $\langle i, j \rangle$  是否存在, 即  $i$  到  $j$  无边时  $\text{edges}[i][j] = 0$  或  $\infty$ , 否则  $\text{edges}[i][j] = 1$ 。

上列情况同样适合于无向图, 此时, 在上面的说明中, 用  $(i, j)$  代替  $\langle i, j \rangle$ 。

对无向图, 邻接矩阵是个对称矩阵, 可按对称矩阵的压缩存储方法存储。

**例 7-2** 对图 7-1 所示的有向图与无向图, 它们的邻接矩阵如图所示 (不考虑边权, 设 a,b,c,d 的编号为 1, 2, 3, 4)

这种存储方式的空间复杂度正比于图的结点个数的平方, 所以, 当图中结点很多但边却很少时, 采用这种结构会造成很大的浪费。不过, 这种存储结构没有链指针之类的附加开销。

## (二) 算法实现的考虑

下面我们考虑在邻接矩阵存储方式下, 图的基本操作的基本实现方法。

### 1. 结点、边的增删

对邻接矩阵, 结点或边的增删的算法较简单, 但一般要移动元素。

- 删除边: 删除一条边比较简单, 只需将对应元素置无边标志;
- 删除点: 删除一个结点, 要将与它关联的边一同删除, 即删除结点在矩阵中对应的行与列。
- 增加边: 为现有两结点间增加一条边, 只需将对应矩阵元素置边信息。
- 增加点: 增加一个新结点, 要在矩阵中相应位置插入一行一列。
- 增加新结点边: 即插入带新结点的边, 先插入新结点, 然后置边信息。

## 2. 度的计算

- 出度：结点  $i$  的出度等于  $i$  所对应的行上的非特殊元素的个数。
- 入度：结点  $i$  的入度等于  $i$  所对应的列上的非特殊元素的个数。

## 3. 找邻接点/判别是否有边

对邻接矩阵，判别是否有边是直接的，即要知结点  $i$  到  $j$  是否有边，只要测试元素  $A[i][j]$  即可。至于找邻接点，则是个依次判别是否有边的问题。

## 4. 找路径

要找出结点  $x$  到结点  $y$  之间的路径，即  $\text{path}(x,y)$ ，按下列伪码描述进行：

```
Path(x, y)
{
    if (x==y) return x; //所找的路径为(x)
    if (若 x 到 y 有边) return (x, y); //所找的路径为(x,y)
    for (x 的每个直接可达点 u)
    {
        if (u 尚未试探过)
            if (Path(u,y)存在) return (x,u)+Path(u, y);
    }
    return 0; //空路径
}
```

该算法并未涉及到图的具体存贮结构，故可适合于任意存贮结构。对不同的存贮结构，只是判别是否有边及找邻接点的方法不同。

## § 7.3.2 邻接表

### (一) 存贮方法

图的邻接表(Adjacency List)存贮法与树的邻接表存贮法类似，其具体方法为：

- 为图中每个结点  $x$  建立一个线性链表，称  $x$  的出边链表。
- 对无向图， $x$  的出边链表的每个元素，分别存放  $x$  的各邻接点边的信息。
- 对有向图， $x$  的出边链表的每个元素，分别存放以  $x$  为始点的各个弧的信息。
- 线性链表的结点结构为：

边终点编号	边内容	链指针
-------	-----	-----

■为每个图结点，设立一个如下形式的结点：

结点编号	结点内容	链头指针
------	------	------

■各结点可以存储在一个一维数组（称为头数组）中（每个数组元素分别对应一个图结点），也可以增设一个链指针，使各结点链在一起，形成另一个单链表。

例如，图 7-1 (a)所示的图的邻接表如图 7-3 所示。这里，假定图结点存储在数组中。

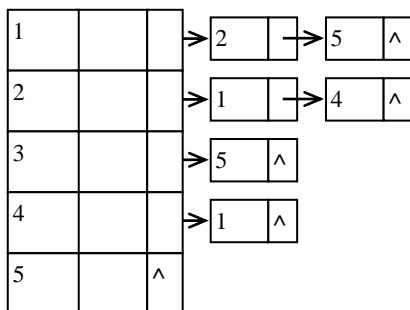


图 7-3 图 7-1(a)对应的邻接表  
(链表结点中的起点编号未标出)

如果图中有  $n$  个结点， $e$  条边，则邻接表需  $n$  个头数组元素、 $e$  个表结点。显然，当边数较少时，该存储方式的存储利用率更高。这与邻接矩阵的情况正好相反。

🔊 邻接矩阵主要面向有向图。对无向图，为了处理方便，将  $(x, y)$  看作是两条边  $\langle x, y \rangle$  和  $\langle y, x \rangle$ ，因此，每条边在出边链表中对应两个链结点。如果为了节省存储空间、方便处理一致性问题，则对每条边，可只设置一个链结点。但为了处理上方便，还需要设置其他一些信息，这就是在后面将要介绍的多重邻接表的方法。

## (二) 存储结构的高级语言描述

邻接矩阵下的图边，应该是前面介绍的图边抽象类型 `TGraphEdge0` 的派生（对象）类，在数据成员方面，它只需在 `TGraphEdge0` 的基础上增加指向下条出边的指针。具体的 C++ 描述为：

```
template <class TEdgeInfo>
struct TGraphEdgeAL: public TGraphEdge0 <TEdgeInfo>
{ // 邻接表边
public:
    TGraphEdgeAL *next; // 链指针，指向与本对象邻接的下条出边
    // 下面是成员函数的声明，同 TGraphEdge0，此处不重载，故不给出
};
```

邻接表下的图结点，也应该是前面介绍的图结点抽象类型 `TGraphNode0` 的派生（对象）类，在数据成员方面，它只需在 `TGraphNode0` 的基础上增加指向相应的出边链表的头结点的指针，具体的 C++ 描述为：

```
template <class TElem, class TEdgeElem>
struct TGraphNodeAL : public TGraphNode0 <TElem>
{ //邻接表结点
public:
    GraphEdgeAL<TEdgeElem> * firstOutEdge; //本结点的第一条出边的指针
    //面是其他成员的声明，同 TGraphNode0，此处不重载，故不给出
};
```

接下来就可以定义具体的邻接表图类 `TGraphAL` 了。该类是 `TGraph0` 的派生类，代表着整个图，所以，它的数据成员的设置，应该能使得在已知它的情况下，访问到邻接表，这需要在 `TGraphAL` 设置指向邻接表的指针。具体的定义如下：

```
template <class TElem, class TEdgeElem> //用到两个可变类型，分别是图结点内容和边内容
class TGraphAL: public TGraph0<TElem, TEdgeElem>
{ //邻接表类
public:
    TGraphNodeAL<TElem, TEdgeElem> *nodes; //存放图结点的一维数组（头数组）

    TGraphAL(){nodes=NULL; numNodes=0;}; //构造函数
    TGraphAL(long mNumNodes); //构造函数，只创建头数组
    TGraphAL(TGraphEdgeRec *edgeSet, long n);
        //构此造函数根据边集 edgeSet 创建邻接矩阵
    ~TGraphAL(); //析构函数，负责释放邻接表内结点

    virtual long Cluster(long v0, long *nos, TTraverseMode tm);
        //面是其他成员的声明，同 TGraph0，此处略(实际上机时不可略)
};
```

### (三) 算法实现的考虑

邻接表下的图的操作的实现，与邻接矩阵下不同，下面进行讨论。

#### 1. 求邻接点、关联边

求以  $x$  为始点（尾）的边时，只需搜索  $x$  的出边链表（线性链表）；求以  $x$  为终点（头）的结点时，则要搜索各结点的出边链表，检查其中是否有以  $x$  为终点的链表结点，

若  $i$  的线性链表中有终点为  $x$  的结点，则  $\langle i, x \rangle$  就是一条边，其中  $i$  为始点。

## 2. 结点、边的增删

• 删除一条边：对有向图，删除边  $\langle x, y \rangle$  时，从  $x$  的出边链表中找到终点为  $y$  的结点，并删除之；对无向图，删除边  $(x, y)$  时，先在  $x$  的出边链表中找到终点为  $y$  的结点，并删除之，然后在  $y$  的出边链表中找到终点为  $x$  的结点并删除之。

• 增加新结点：在头数组中增加一个元素即可。

• 增加边：设要增加边  $\langle x, y \rangle$ ，若  $x$  与  $y$  是已存在结点，则在  $x$  的线性链表中增加一个结点，置其终点为  $y$ ；对无向图，还要按同样方法增加  $\langle y, x \rangle$ 。若  $x$  或  $y$  是新结点，则先在头数组中增加对应的元素，然后按上法将边  $\langle x, y \rangle$  加到链表中。

• 删除一个结点：是上述几项操作的复合。先删除与待删结点关联的各个边，然后再在头数组中删除该结点。

## 3. 求度

• 出度：结点  $x$  的出度等于  $x$  的出边链表中结点个数。

• 入度：结点  $x$  的入度，等于各结点的出边链表以  $x$  为终点的链结点的个数。故求入度时要扫描整个邻接表。

## 4. 路径

基本思想同邻接矩阵。

## 5. 串行化问题

串行化问题是指如何将内存图结构转化（输出）为线性表示（表达式表示），或者如何将线性表示还原为内存结构（逆串行化）。该问题对图结构的持久保存（磁盘文件保存）十分重要。对邻接矩阵，串行化问题是数组的存储问题，很简单。

对邻接表，要实现串行化，首先要确定结构的线性表示方式。我们这里将边集做为图的线性表示。例如，图 7-1(a)对应的边集为

$\{\langle 1, 2 \rangle, \langle 1, 5 \rangle, \langle 2, 1 \rangle, \langle 2, 4 \rangle, \langle 3, 5 \rangle, \langle 4, 1 \rangle\}$

下面我们给出基于这种线性表示的串行化程序。该程序也可供 TGraphAL 的构造函数调用，实现根据边集创建邻接表。

```
template <class TElem, class TEdgeElem>
TGraphNodeAL<TElem, TEdgeElem> *EdgeSetToAL(TGraphEdgeRec *edgeSet,
                                              long n, long &numN)
{
    //由边集合 edgeSet（一维数组）创建图的邻接表。
    //返回所创建的邻接表的头数组指针，并将图结点个数返回到 numN 中
    //邻接表的边（链表）结点类型为 TGraphEdgeAL
    //结点（头数组元素）类型为 TGraphNodeAL,它们在头文件中定义
    long i,j;
```

```
numN=0;
for (i=0; i<n; i++) //统计 edgeSet 中编号值的种数，即图结点的个数
{
    for (j=0; j<i; j++)
        if (i==j) continue;
    else
        if (edgeSet[i].startNo==edgeSet[j].startNo || edgeSet[i].startNo==edgeSet[j].endNo)
            break;
    if (j==i) numN++;

    for (j=0; j<i; j++)
        if (i==j) continue;
    else
        if (edgeSet[i].endNo==edgeSet[j].startNo || edgeSet[i].endNo==edgeSet[j].endNo)
            break;
    if (j==i) numN++;
} //for
if (numN<1) return NULL;

TGraphEdgeAL<TEdgeElem> *p;
TGraphNodeAL<TElem, TEdgeElem> *head;
head = new TGraphNodeAL<TElem, TEdgeElem>[numN];

for (i=0; i<numN; i++)
{
    head[i].firstOutEdge = NULL;
    head[i].no=i;
}

for (i=0; i<n; i++)
{ //每次生成一条边
    p = new TGraphEdgeAL<TEdgeElem> ; //生成一个邻接表结点 p
    p->startNo = edgeSet[i].startNo;
    p->endNo = edgeSet[i].endNo;
    p->weight = edgeSet[i].weight;

    p->next = head[edgeSet[i].startNo].firstOutEdge; //将 p 插入对应链表的最前面
```



```

    head[edgeSet[i].startNo].firstOutEdge = p;
}
return head;
}

```

至于逆串行化，也比较简单，留作练习。

🔊 有时为了操作方便，对图采用逆邻接表存贮法。所谓**逆邻接表**，是指将有向图中的各弧的方向掉转后所对应的邻接表。对逆邻接表求入度与出度的方法，正好与邻接表情况相反。

### § 7.3.3 十字链表（有向图）\*

#### （一）存贮方法

图中任一结点均可能有若干个前趋和后继。如果将某结点的各前趋视为与它同“行”的元素，而将它的各后继视为与它同“列”的元素，则可沿用矩阵的十字链表存贮法存贮图。这就是图的十字链表存贮法的基本思想。

通常，图的十字链表的“行”链表与“列”链表不使用循环结构。在图的十字链表中， $i$ “行”对应的线性链表中的元素是以  $i$  为始点（尾）的边，而  $i$ “列”对应的线性链表中的结点是以  $i$  为终点（头）的边。具体地讲，图的十字链表存贮规则为：

■ 对于图中的每条边，用如下结构的一个结点描述（称为**边结点**）：

边信息	起点	终点	射入链	射出链
-----	----	----	-----	-----

其中，

边信息：描述对应边的有关信息，如内容、权等。

起点：边的起点的编号。

终点：边的终点的编号。

射入链：指向终点相同的下条边

射出链：指向起点相同的下条边

■ 对于每个图结点，设一个**顶点结点**，其结构为：

结点编号	结点信息	射入链头	射出链头
------	------	------	------

其中：

结点编号：存放结点的编号；

结点信息：存放结点的有关信息；

射入链头：指向该结点的射入链（即以该结点为终点的弧构成的链）中的第一个结点；

射出链头：指向该结点的射出链（即以该结点为始点的弧构成的链）中的第一个结点；

■为了快速访问，可设一个一维数组（称为头数组），存放上述的各顶点结点。也可在顶点结点中增设一个链域，使各顶点结点链为一条链。

图 7-4 给出了一个有向图的十字链表。

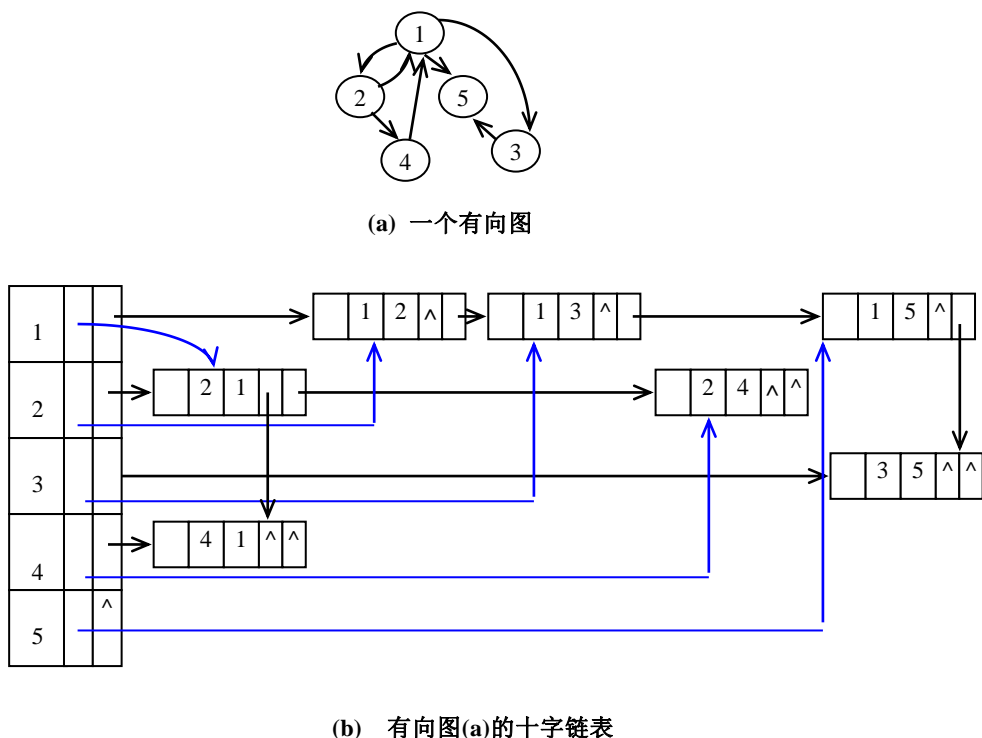


图 7-4 有向图十字链表示例

## (二) 高级语言描述：

有向图的十字链表存储结构的主体是上面介绍的边结点和顶点结点，它们都分别是前面介绍的抽象边和抽象结点的派生（对象）类。

十字链表的边结点，是在前面给出的抽象结点 TGraphEdge0 的基础上，增加“射入链”和“射出链”信息而成，具体的 C++描述如下：

```
template <class TEdgeInfo>
```

```
struct TGraphEdgeCrossLink: public TGraphEdge0 <TEdgeInfo>
{
    GraphEdge *nextOut; //指向下一条射出边
    TGraphEdge *nextIn; //指向下一条射入边
    //下面是其他成员的声明，同 TGraphEdge0，此处不重载，故不给出
}
```

十字链表的顶点结点，是在前面给出的抽象结点 TGraphNode0 的基础上，增加指向相应的第一条射入边和第一条射出边的指针而成，具体的 C++描述如下：

```
template <class TElem>
struct TGraphNodeCrossLink: public TGraphNode0 <TElem>
{
    TGraphEdge *firstOut; //指向第一条射出边
    TGraphEdge *firstIn; //指向第一条射入边
    //下面是其他成员的声明，同 TGraphNode0，此处不重载，故不给出
}
```

至于代表整个图的类，是从 TGraph0 派生而来，并增加头数组作为成员，以便访问整个十字链表。

```
const int CNST_NumNodes;
template <class TElem, class TEdgeElem> //用到两个可变类型，分别是图结点内容和边内容
class TGraphCrossLink: public TGraph0<TElem, TEdgeElem>
{
public:
    TGraphNodeCrossLink <TElem> nodes[CNST_NumNodes] //存放图结点的一维数组
    //下面是其他成员的声明，同 TGraph0，此略（但实际上机时不可省略）
}
```

### (三) 结构与算法实现的考虑

图的十字链表是对图的邻接表的一种扩展，与邻接表相同的是，它也是对每条边设立一个链表结点。但不同的是，它的链表结点增设了一个用于将终点相同的弧链接在一起的链指针。所以对十字链表的操作与对邻接表的操作类似。只是要注意对新增链的利用（带来了方便性）和处理（带来了复杂性）。

十字链表实质上是邻接表与逆邻接表的结合，它同时具备这两种表的方便性，对任一结点，既可容易找出以它为始点的各边（点），也可方便找出以它为终点的各边（点）。

由于十字链表中每个链表结点（图中的边对应的结点）有两个链域，所以它的存储效率要比邻接表低。

如果将图的邻接矩阵用矩阵的十字链表表示，则可发现，图的十字链表是图的邻接矩阵的十字链表的变形。二者的主要差别是：①前者的链表结点不需按起点与终点（相当于行号与列号）的大小排列；②前者的“行”与“列”链表一般不需构成循环结构。

### § 7.3.4 邻接多重表（无向图）\*

邻接多重表是面向无向图的一种链式存储结构，可视为由有向图的十字链表演变而来。

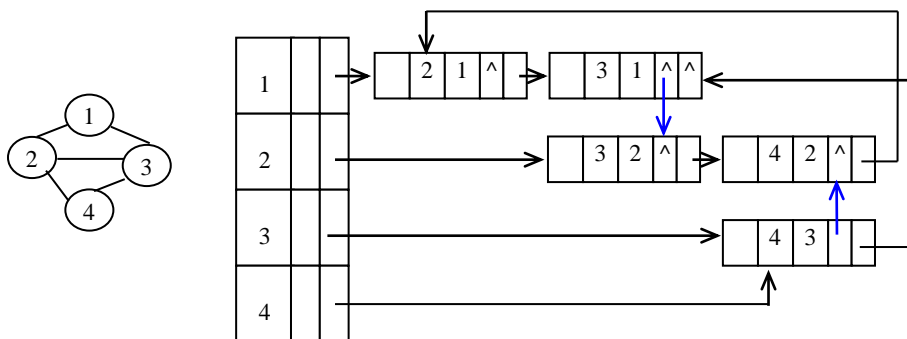


图 7-5 无向图邻接多重表示例

对于无向图，若要按邻接表或十字链表的方法存储，则图中每条边对应两个链表结点（边 $(x,y)$ 相当于 $\langle x,y \rangle$ 与 $\langle y,x \rangle$ ），这不仅造成存储浪费，而且给图的操作维护带来麻烦。要解决此问题，就必须限制每个图边对应一个链表结点，将此限制应用于图的十字链表存储法上，就导出了图的邻接多重表存储法。下面给出邻接多重表的规则。

■图中每条边 $(x,y)$ 设一个边结点，格式为

边信息	起点 x	终点 y	link1	link2
-----	------	------	-------	-------

这里：

边信息：存储有关边 $(x,y)$ 的信息；

起点 x、终点 y：边 $(x,y)$ 的两个端点的编号；

link1、link2：分别为关联于 x 和 y 的下条边的指针。即 link1 指向关联于结点 x 的下条边，而 link2 指向关联于结点 y 的下条边。

■每个图结点设立一个如下形式的结点（顶点结点）：

结点编号	结点信息	首边指针
------	------	------

这里：

结点编号：图结点的编号；

结点信息：存储有关图结点的信息；

首边指针：指向关联于该结点的第一条边的边结点。

■为了快速查找顶点结点，可设一个一维数组，存放各顶点结点。也可在顶点结点中增设链域，将各顶点结点链接在一起。

图 7-5 给出了一个无向图的邻接多重表的例子。

容易看出，因为对无向图，每条边的起点与终点没有指定，所以邻接多重表在形式上是不唯一的。

在这种结构中，每一结点都对应一条链（称为关联边链），链中结点是与它关联的各个边。有时，为了处理方便，使链中结点按边的另一端点排序。图 7-5 所示的邻接多重表中，结点的关联边链中结点按关联边的邻接点升序排列。

邻接多重表的高级语言描述及基本操作的实现均与十字链表类似。此处不再赘述。

## § 7.4 图的遍历

从这节起，我们介绍图的一些重要操作的实现，包括遍历、拓扑排序、关键路径等。另有一些重要操作，如最短路径问题、最小生成树问题，由于主要难点在于算法，所以我们安排在后面的算法设计章节中介绍。

图的遍历与树的遍历一样具有十分重要的作用，图的许多其他操作（算法）都与遍历相关。

### § 7.4.1 概述

与树类似，也可对图进行遍历。图的遍历的含意是，从图中某结点出发，按某既定方式访问图中各个可访问到的结点，使每个可访问到的结点恰被访问一次。

图的遍历方式有两种：深度优先与广度优先方式，分别对应于树的先根遍历与层序遍历。

树中不存在回路，但图中可能有回路。因此，当沿回路进行扫描时，一个结点可能被扫描到多次，可能导致死循环。为了避免这种情形，在遍历中，应为每个结点设立一个访问标志，每扫描到一个结点，要检查它的访问标志，若标志为“未访问”，则按正常方式对其进行处理（如访问或转到它的邻接点等），否则放过它，扫描下一个结点。

访问标志的设置有两种方法：①在描述图结的记录中增设一个访问标志位。这种方法的优点是，访问“访问标志”的方法与访问结点的方法一致。如果访问标志需要与图结构同生命期，则这种方法比较合适。但是，若访问标志要重复使用，就必须先重新初始化访问标志。如果图结点的存储不利于顺序访问，这往往也是个遍历问题！②另设一个“访问数组”，令它的每个元素对应于一个图结点访问标志。这种方法的访问标志很容易多次初始化。

从图中某一结点出发，一趟只能遍历到它所在的极大连通分量中的结点，要想遍历

到图中各结点，需进行多趟遍历（每趟遍历一个极大连通分量）。该过程可描述为：

```
for (图中每个结点 v)
    if (v 尚未被访问过)
        从 v 出发遍历该图;
```

## § 7.4.2 深度优先遍历

### (一) 遍历规则

从图中某结点  $v_0$  出发，**深度优先遍历**（DFS: Depth First Search）图的规则为：

- 访问  $v_0$ ;
- 对  $v_0$  的各个出点  $v_{01}, v_{02}, \dots, v_{0m}$ ，每次从它们中按一定方式（也可任选）选取一个未被访问过的结点，从该结点出发按深度优先遍历方式遍历。

显然，因为我们没有规定对出点的遍历次序，所以，图的深度优先遍历结果一般不唯一。

例如，对图 7-6 给出的有向图与无向图，一些遍历结果（结点访问次序）为：

左图：从 1 出发：1, 2, 4, 5；或 1, 5, 2, 4

从 2 出发：2, 1, 5, 4；或 2, 4, 1, 5

右图：从 a 出发：a, b, c, d；或 a, b, d, c；... ..

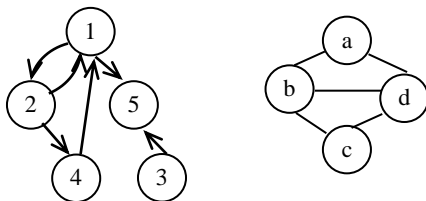


图 7-6 一个有向图（左）和无向图

### (二) 递归实现方法

#### 1. 一般算法

下面考虑深度优先遍历的递归实现的一般方法（存储结构无关）。

图的深度优先遍历与二叉树的前序遍历相似。不同之处有：①二叉树每个结点至多有两个可达邻接点（左右儿子），而图的可达邻接点数目不定；②对二叉树，沿可达邻接点搜索时不会发生回绕，但对图，若不加特别控制，就有可能回绕。

下面是图的深度优先遍历递归算法的一般性描述。如果要另设一个数组作为访问标

志，则该数组要在递归过程（函数）之外初始化为“未访问”。

long DFS(图 g, 结点 v0)

{ //图深度优先遍历递归算法。从结点 v0 出发，深度优先遍历图 g，返回访问到的结点总数

int nNodes; //寄存访问到的结点数目

访问 v0;

为 v0 置已访问标志;

nNodes=1;

求出 v0 的第 1 个可达邻接点 v;

while (v 存在)

{

if (v 未被访问过) nNodes=nNodes+DFS(g,v);

求出 v0 的下个可达邻接点 v;

}

return nNodes;

}

该算法的正确性是显然的，因为它完全按照深度优先遍历的递归定义进行。

该算法的伪码描述与具体的数据结构无关。下面的程序分别给出了针对邻接矩阵与邻接表的算法模型。

## 2. 邻接矩阵实现

这里我们为了突出主题、简化问题，假定图是用一般的邻接矩阵存储，邻接矩阵用简单的二维数组表示（静态），用 0 和 1 分别表示无边和有边。图结点用自然数编号。

long DFS1(int g[][CNST\_NumNodes], long n, long v0, char \*visited,

long \*resu, long &top )

{//深度优先遍历图（递归）。图 g 为邻接矩阵，结点编号为 0~n. 返回实际遍历到的结点数目

//visited 是访问标志数组，调用本函数前，应为其分配空间并初始化为全 0(未访问)

//resu 为一维数组，用于存放所遍历到的结点的编号，调用本函数前，应为其分配空间

long nNodes, i;

nNodes=1;

resu[top++]=v0; //将访问到的结点依次存于 resu[]

visited[v0]=1; //为 v0 置已访问标志

for (i=0; i<n; i++)

{//依次从 v0 的各个出点出发，深度优先遍历图

if (g[v0][i]!=0) //若<v0, i>是边

if (!visited[i]) //若结点 i 未被访问过

```

        nNodes = nNodes+DFS1(g, n, i, visited,resu); //从 i 起深度优先遍历图
    }
    return nNodes;
}

```

上面函数中的参数 `visited` 和 `top` 实质上是中间变量，只是为了避免在递归调用时重新初始化而放在参数表中，造成使用的不方便，为此，做个包装程序：

```

long DFS1(int g[][CNST_NumNodes], long n, long v0, long *resu )
{
    char *visited;
    long top=0;

    visited = new char[n];
    for (long i=0; i<n; i++) visited[i]=0;

    long num=DFS1( g, n, v0, visited, resu, top );
    delete visited;
    return num;
}

```

⚠️如果不想让 `visited` 或 `top` 做为函数参数，也可以在函数中将其定义为 `static` 型量。但是，这样的程序是不可再入的，即函数再次被调用时，`static` 型的量也不重新初始化，造成错误！

## 2. 邻接表实现

下面给出图 `g` 是邻接表时，深度优先遍历的实现。

```

template <class TElem, class TEdgeElem>
long DFS2(TGraphNodeAL<TElem, TEdgeElem> *nodes,long n,
          long v0, char *visited, long *resu,long &top)
{ //深度优先遍历用邻接表表示的图。nodes 是邻接表的头数组，n 为结点个数（编号为 0~n）。
  //v0 为遍历的起点。返回实际遍历到的结点的数目。
  //visited 是访问标志数组，调用本函数前，应为其分配空间并初始化为全 0(未访问)
  //resu 为一维数组，用于存放所遍历到的结点的编号，调用本函数前，应为其分配空间

  long nNodes, i;
  TGraphEdgeAL<TEdgeElem> *p;

  nNodes=1;
  resu[top++]=v0; //将访问到的结点依次存于 resu[]

```



```

visited[v0]=1; //置 v0 为"已访问"标志
p = nodes[v0].firstOutEdge; //求出 v0 的第一个出点 p
while (p!=NULL)
{
    if (!visited[p->endNo] ) //若 p 未访问，则从 p 出发深度优先遍历
        nNodes = nNodes+DFS2(nodes, n, p->endNo, visited, resu,top);
    p = p->next; //令 p 指向 v0 的下个出点
}
return nNodes;
}

```

与邻接矩阵的情况类似，也可以对该程序“包装”，以隐蔽 visited 和 top。

### (三) 非递归实现方法

#### 1. 一般方法

下面考虑深度优先遍历的非递归实现的一般方法（存储结构无关）。

图的深度优先遍历的非递归实现，仍然与二叉树的前序遍历非递归实现相似。不同之处有：①二叉树每个结点至多有两个可达邻接点（左右儿子），而图的可达邻接点数目不定，因此，当结点重新出现在栈顶时，不能一定出栈，而是要根据它的各出点是否都已被访问过来决定（是则出栈）；②对二叉树，沿可达邻接点搜索时不会发生回绕，但对图，若不加特别控制，就有可能回绕。

下面是图的深度优先遍历非递归算法的一般性描述。

long DFS\_NR(图 g, 结点 v0)

{ //图深度优先遍历非递归算法。从结点 v0 出发，深度优先遍历图 g，返回访问到的结点总数

int nNodes; //寄存访问到的结点数

访问 v0;

为 v0 置已访问标志;

v0 进栈 S;

nNodes=1;

求出 v0 的第 1 个可达邻接点 v;

while (栈 S 不空)

{

v = 栈 S 顶部元素;

求 v 的下个未访问过的出点 i;

访问 i;

为 i 置已访问标志;

i 进栈 S;

nNodes++;

```

        if (v 已无未被访问过的出点) 出栈;
    }
    return  nNodes;
}

```

上面的伪码描述与具体的数据结构无关。下面的程序分别给出了针对邻接矩阵与邻接表的算法模型。

## 2. 邻接矩阵实现

```

long DFS1_NR(int g[][CNST_NumNodes], long n, long v0, long *resu )
{ //深度优先遍历图(非递归)。图 g 为邻接矩阵，结点编号为 0~n. 返回实际遍历到的结点数目
  //resu 为一维数组，用于存放所遍历到的结点的编号，调用本函数前，应为其分配空间

  long nNodes, i, v;
  long top;
  char *visited;
  long *s;

  visited = new char[n];
  for (i=0; i<n; i++) visited[i]=0;

  s = new long[n+1];
  top=0;

  nNodes=0;
  resu[nNodes++]=v0; //将访问到的结点依次存于 resu[]
  visited[v0]=1; //为 v0 置已访问标志

  top++; s[top]=v0;
  while (top!=0)
  {
    v=s[top];
    for (i=0; i<n; i++) //寻找 v 的下个未访问的邻接点
      if (g[v][i]!=0) //若<v, i>是边
        if (!visited[i]) //若结点 i 未被访问过
        {
          resu[nNodes++]=i; //将访问到的结点依次存于 resu[]
          visited[i]=1; //为 i 置已访问标志
          top++; s[top]=i; //i 进栈
        }
      }
    top--;
  }
}

```

```

        break;
    }
    if (i==n) top--; //若 v 已无未访问到的出点，则将其退栈
} //while
return nNodes;
}

```

至于邻接表的实现，这里就不给出具体程序了，留作练习。

### § 7.4.3 深度优先遍历的性质

深度优先遍历有许多重要而有趣的性质，利用它们可以获得有关图的更多的信息。我们这里作一简单介绍。

#### (一) 深度优先生成树与单源路径

在深度优先遍历中，如果将每次“前进”（纵深）路过的（将被访问的）结点和边都记录下来，就得到一个子图，该子图是以出发点为根的树，称为**深度优先生成树**。

如果从图的多个结点出发才能遍历到所有结点，则图的深度优先遍历树有多棵，从而构成森林，称为**深度优先生成森林**。

显然，由图得到深度优先生成树，相当于对图“层次”化，使图中每个结点都有一个层次号。

此外，从  $v_0$  出发深度优先遍历树，同时也产生  $v_0$  到各结点的路径。例如，图 7-7(a) 的出发点为 1 的深度优先生成树如图 7-7(b)。

#### (二) 时间戳

在遍历中，对每个结点  $v$ ，定义从第一次“发现”（即第一次遇到，开始遍历）它的时刻为它的**发现时间**，记为  $S(v)$ ，定义遍历完  $v$  的时刻为  $v$  的**完成时间**，记为  $E(v)$ 。这两种时间都称  $v$  的**时间戳**。一般情况下，用遍历中“路过”（包括回退）的结点数表示时间。图 7-7(b)中，结点旁边的数字“ $a/b$ ”表示对应结点的开始时间和完成时间分别为  $a$  和  $b$ （针对(a)的从 1 出发的深度优先遍历）。

时间戳的差  $E(v)-S(v)$  可用在推算深度优先遍历的进行情况，做为遍历的启发信息，指导遍历算法尽快发现目标。

#### (三) 遍历括号

某结点  $v$  的深度优先**遍历括号**定义为：

$$(v \ X_1 \ X_2 \ \dots \ X_m \ v)$$

这里,  $X_i$  为  $v$  的出点中, 可从  $v$  出发直接访问到的各结点的遍历括号。  $i=1, \dots, m, m \geq 0$ 。

例如, 图 7-7(b) 的结点 1 的遍历括号为:

(1 (2 (4 4) 2) (5 (3 3) (6 6) 5) 1)

遍历括号实质上是广义表, 它完全描述了深度优先遍历的过程及深度优先遍历生成树的结构, 可做为深度优先遍历生成树的串行化表达式。

遍历括号和时间戳存在某些有用的关系, 限于篇幅, 此处就不讨论了。

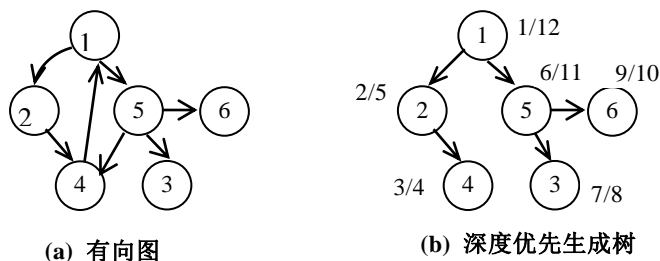


图 7-7 深度优先遍历性质说明

#### § 7.4.4 广度优先遍历

##### (一) 图的广度优先分层

图的广度优先分层与图的广度优先遍历密切相关。另外, 在许多其他问题中, 也涉及到图的广度优先分层。

图的广度优先分层就是要识别出图中每个结点属于的层次, 即给每个结点编一个层次号。但是, 图本身是非层次结构, 所以, 一般也无层次而言。然而, 我们若只从某些关系/角度考虑问题, 则就可对图分层了。

分层时, 应先确定一个或几个参考点, 将它们的层号指定为起始层 (第 1 层)。下面给出以结点  $v_0$  为参考点的图的**广度优先分层**的定义 (非过程化), 这里用  $\text{Level}(v)$  表示结点  $v$  的广度优先分层的层号:

■ 令  $\text{Level}(v_0) = 1$

■ 对任意的  $v \neq v_0$ , 若存在  $v_0$  到  $v$  的通路, 则令

$$\text{Level}(v) = 1 + \text{MIN}_u \{ \text{Level}(u) \mid \langle u, v \rangle \text{ 是图的边} \}$$

否则令  $\text{Level}(v) = \infty$

这个定义是说明性的 (非过程化), 上式中引用的  $\text{Level}(u)$  是指  $u$  的最终层号, 若要直接按此定义推算层号, 则可能是一个多次迭代的过程 (迭代前先令各结点的层号为  $\infty$ )。

例 7-3 对图 7-1, 广度优先分层情况为:

左图: 从 1 出发分层:

层号为 1 的结点: 1

层号为 2 的结点: 2, 5

层号为 3 的结点: 4

层号为  $\infty$  的结点: 3

左图: 从 2 出发分层:

层号为 1 的结点: 2

层号为 2 的结点: 1, 4

层号为 3 的结点: 5

层号为  $\infty$  的结点: 3

右图: 从 a 出发分层:

层号为 1 的结点: a

层号为 2 的结点: b, d

层号为 3 的结点: c

## (二) 图的广度优先遍历方法

从结点  $v_0$  出发, 广度优先遍历 (Breadth/Width First Search) 图的方法是, 按从  $v_0$  出发, 对图的广度优先分层的层次号的大小次序访问结点, 即先访问第一层上结点, 然后访问第二层上结点, ... 等等, 同层上结点可按邻接点次序或任意。

例如, 图 7-1 中的两个图的一些广度优先遍历次序如下。

左图: 从 1 出发广度优先遍历结果: 1, 2, 5, 4

左图: 从 2 出发广度优先遍历: 2, 1, 4, 5

右图: 从 a 出发广度优先遍历: a, b, d, c

从另一角度看, 从  $v$  出发广度优先遍历, 是先访问  $v$ , 然后, 对任意结点  $u$ , 在访问了  $u$  之后, 对  $u$  的各可达点的访问, 按距  $u$  的距离 (边数) 大小次序进行。

显然, 若图的结点是无序的 (即邻接点无次序关系), 则广度优先遍历次序也不是唯一的, 但层次关系不颠倒。

## (三) 算法实现

### 1. 一般方法

对于深度优先遍历, 用递归方法描述是件自然的事, 但广度优先遍历不然, 使用递归描述反而会使问题复杂化, 所以我们这里只讲非递归描述法

 对于基于递归的语言, 如 Prolog, 递归描述是不可绕过的。

广度优先遍历是一种分层处理, 对这种分层处理, 使用队列是自然的。我们设立一个队列, 任何时刻, 均保证它满足下列条件:


- 队中元素是已访问过的结点的可达邻接点;

- 队中元素是尚未被访问过的；
- 队中元素按它们所处的层次的先后排列。

这样，我们就可不断地每次从队中取出一个元素并访问之，然后再将该元素的尚未被访问过的邻接点进队，直至队空。该方法可描述如下：

```
long BFS(图 g, 结点 v0)
{ //在图 g 中从 v0 出发按广度优先遍历方式遍历 g，返回遍历到的结点数目
  long nNodes=0;
  初始化队 Q;
  if (v0 存在)
  { v0 入 Q;
    置 v0 为已访问标志;
  }

  while (Q 不空)
  {
    队 Q 头元素出队并送 v;
    访问 v;
    nNodes++; //对已访问元素计数
    求出 v 的第一个可达邻接点 w;
    while (w 存在)
    {
      if (w 尚未被访问过)
      { w 入 Q;
        置 w 为已访问标志;
      }
      求 v 的下个可达邻接点 w;
    }
  }
  return nNodes;
}
```

 请思考，(1)如果上面的程序中不使用队列，而所用栈，那么是否正确？为什么？(2)为什么结点一入队就置已访问标志？

上面的算法描述是一般性的，并未涉及到具体的存贮结构。下面给出针对具体存储结构算法。

## 2. 邻接矩阵实现

设图用邻接矩阵表示，则它的广度优先遍历算法如下。

```
long BFS1_NR(int g[][CNST_NumNodes], long n, long v0, long *nos)
```

```

{ //广度优先遍历（邻接矩阵）：从 v0 出发遍历用邻接矩阵表示的图 g（共 n 个结点）
//将访问到的结点的编号存入 nos(其必须在外面分配 n 个 long 型空间)，返回遍历到的结点数目
long nNodes=0;
long v, w,i;
char *visited;
TQueueSqu<long> Q(n+1);

visited = new char[n];
for (i=0; i<n; i++) visited[i] = 0; //初始化访问标志数组

if (v0>=0 && v0<n)
    { Q.QPush(v0); visited[v0]=1; }
while (!Q.IsEmpty())
{
    v = Q.QPop();
    nos[nNodes]= v; //访问结点 v
    nNodes++;
    for (w=0; w<n; w++) //找 v 的各未访问的出点
        if (g[v][w]!=0)
            if (!visited[w])
                { Q.QPush(w); //v 的各未访问的出点进栈
                  visited[w]=1;
                }
} //while
delete [] visited;
return nNodes;
}

```

## 2. 邻接表实现

针对邻接表的算法为：

```

template <class TElem, class TEdgeElem>
long BFS2_NR(TGraphNodeAL<TElem, TEdgeElem> *nodes, long n,
             long v0, long *nos)
{ //广度优先遍历（邻接表）：从 v0 出发遍历用邻接表表示的图 g（共 n 个结点）
//将访问到的结点的编号存入 nos(其必须在外面分配 n 个 long 型空间)，返回遍历到的结点数目
long nNodes=0;
TGraphEdge *p;
char *visited;

```

```

TQueuesSqu<long> Q(n+1);

visited = new char[n];
for (i=0; i<n; i++) visited[i] = 0; //初始化访问标志数组

if (v0>=0 && v0<=n) Q.QPush(v0);
while (!Q.Empty())
{
    v = Q.QPop();
    nos[nNodes]= v; //访问结点 v
    visited[v]=1;
    nNodes++;
    p = g[v]. firstOutEdge;
    while (p!=NULL)
    {
        if (!visited[p->endNo])
        { Q.QPush(p->endNo);
          visited[p->endNo]=1;
        }
        p = p->next
    }//while
} //while
delete [] visited;
return nNodes;
}

```

### § 7.4.5 广度优先遍历的性质

与深度优先遍历类似，广度优先遍历也有许多有用的特性。

#### (一) 广度优先生成树

在广度优先遍历中，如果将每次“前进”（纵深）路过的（将被访问的）结点和边都记录下来，就得到一个子图，该子图为以出发点为根的树，称为**广度优先生成树**。这种情况与深度优先遍历类似。

类似地，也可以给广度优先生成树结点定义时间戳。



## (二) 最短路径

显然，从  $v_0$  出发广度优先遍历图，将得到  $v_0$  到它的各个可达点的路径。我们这里定义路径上的边的数目为路径长度。与深度优先遍历不同，广度优先遍历得到的  $v_0$  到各点的路径是最短路径（未考虑边权）。关于结论的严格的证明，这里就不给出了。

## § 7.5 拓扑排序

拓扑排序是定义在有向图上的一种操作，目的是根据结点间的关系求得结点的一个线性排列（这点与遍历操作的目的类似）。这种操作在有关工程进度/次序规划之类问题中，有着大量应用。一般的大型工程，都可以划分为多个工序/步骤/子工程，这些工序有的可独立进行，但大多数和其他工序关联，即某工序的进行，要等到其他一些工序的完成才能开始。这类问题都可归结到拓扑排序。

### § 7.5.1 拓扑序列与 AOV 网

对有向图  $G$ ，若它的一个结点序列

$$LS: v_1, v_2, \dots, v_n$$

满足这样的条件： $\langle v_i, v_j \rangle$  是  $G$  的边时，在  $LS$  中  $v_i$  位于  $v_j$  之前，否则在  $LS$  中  $v_i$  与  $v_j$  的前后次序任意，则称  $LS$  为图  $G$  的一个**拓扑序列**，求拓扑序列的操作称为**拓扑排序**（Topological Sorting）。

从代数结构角度来看，拓扑排序是由某集合上的一个偏序关系得到一个全序关系的操作。若将集合中结点作为图结点，将它上的偏序关系作为图的边，则任一偏序关系均可表示为一个有向图。

显然，某一图的拓扑序列可能不唯一。

拓扑排序主要用在一类称为**AOV 网**（Activity on Vertex networks）的有向图中。AOV 网是给有向图的结点与边赋予一定的语义的图，具体地：

结点——代表活动。如工程中的工序/子任务、状态等

边 ——代表活动之间的进行次序。边  $\langle i, j \rangle$  表示活动  $i$  先于  $j$  进行。

AOV 网常用来表达流程图，如一个工程中各子任务间的流程图、产品生产加工流程图、程序流程图、信息（数据）流程图、活动安排流程图等。

例 7-4 学生的选课次序就是一个 AOV 网应用问题。假定某计算机专业的主要课程如图 7-8(左)所示，则对应的 AOV 网如图 7-8(右)所示。

图 7-8(右)可有多个拓扑序列，下面给出了它的两个不同的拓扑序列：

1 2 3 4 6 10 5 9 7 8 11

1 2 3 4 6 10 5 7 9 8 11

如果图中两个结点之间均没有通路，那么称它们是**可并行的**。在一个拓扑序列中，如果某子串(序列中某一段)中的各结点之间均是可并行的，则称该段为一个**并行段(组)**。一个可并行组包含了所有的可并行结点，则称该组为**最大并行段(组)**。找出最大并行组的工作成为**并行识别**。

课程编号	课程名称	先行课
1	高等数学	-
2	离散数学	-
3	高级语言	-
4	数据结构	3, 2
5	操作系统	4, 6
6	计算机组成原理	2
7	数据库原理	5
8	编译原理	4, 7
9	计算机网络	5, 1
10	人工智能原理	1, 2, 3
11	软件工程	7, 9

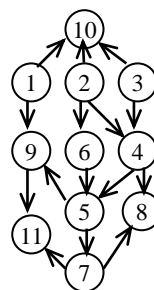


图 7-8 课程表(左)对应的 AOV 网(右)

并行组具有重要的实际意义。位于同一组的各结点代表的活动可以同时进行，以缩短整个事务的完成时间。例如，对选课问题，如果采用串行修课方式，则可完全按拓扑序列进行，但若要在一段时间内同时修多门课，这就需识别出哪些课可同时修，这是一个并行识别问题。同一并行组内的课程可以同时修学。例如，在图 7-8 中，可并行的课程组有(1,2,3)、(4,6,10)、(5)、(9,7)、(8,11)。但注意，这些组之间还是有序的(这几个组的次序为它们的出现次序)。

并不是所有的有向图都可以进行拓扑排序。显然，若图中每个结点都有前驱，或者每个结点都有后继，则该图是不可拓扑排序的。

## § 7.5.2 拓扑排序算法与实现

### (一) 基本方法

这里, 我们给出一种拓扑排序算法。该算法是简单而直观的, 实质上属于广度优先遍历, 因此称为广度优先拓扑排序算法。该算法包含下列几个步骤:

[1] 从有向图中找一个没有前趋的结点  $v$ , 若  $v$  不存在, 则表明不可进行拓扑排序(图中有环路), 结束(不完全成功);

[2] 将  $v$  输出;

[3] 将  $v$  从图中删除, 同时删除关联于  $v$  的所有的边

[4] 若图中全部结点均已输出, 则结束(成功), 否则转[1]继续进行。

此方法的正确性是显然的。

通过对此方法做适当扩充, 就可在求拓扑序列的过程中标识出可并行活动(结点)。具体做法是, 初始时将所有无前趋结点标为一组可并行结点。然后, 每次执行步骤[3]后, 将新产生的无前趋结点标为新的一组可并行结点。为了将同组并行结点连续排列, 在步骤[1]中应优先选取并行组号与上次选择结点的并行组号相同的结点(若有的话)。

🔗 对拓扑排序, 还有一种算法, 称为深度优先拓扑排序。它的基本思想是先输出无后继的结点, 即对每个结点  $v$ , 检查先一次递归地求出  $v$  的每个后继的拓扑序列(各序列连接在一起), 然后将  $v$  插入到该拓扑序列的最前面。关于具体的程序实现, 留作练习。

### (二) 数据结构的选取

为了方便拓扑排序的实现, 需对前面介绍过的图的存贮结构做些扩充。单纯从进行拓扑排序来讲, 图采用邻接表较为方便, 所以这里只考虑邻接表的情况。对其它存贮结构的情况, 读者可做为练习。

从前面给出的拓扑排序基本方法看, 拓扑排序涉及的基本操作有:

- 判别某结点是否有前趋;
- 删除一个无前趋结点及其关联的边。

为此, 我们为每个结点设一个入度域。结点入度大于 0 时表示有前趋, 否则无前趋。对于删除, 其真正目的也不是删除, 而是为了通知出点: 对应的一个前驱已处理完。因此, 删除某结点时, 只需对该结点的各直接可达邻接点(出点)的入度分别减 1。

为了方便找到尚未输出的无前趋结点, 将它们统一存放在一个栈中。每次执行入度减 1(即删除一边)操作后, 若有结点入度变为 0, 则将它放入栈中。每次选择无前趋结点时, 也是从栈中提取。

为此, 邻接表的图结点的定义中需设置入度域, 我们将前面给出的 TGraphNodeAL 定义修改为:

```
template <class TElem, class TEdgeElem>
```

```

struct TGraphNodeAL : public TGraphNode0 <TElem>
{ //邻接表结点
public:
    GraphEdgeAL<TEdgeElem> * firstOutEdge; //本结点的第一条出边的指针
    long inDegree; //入度
    //面是其他成员的声明，同 TGraphNode0，此处不重载，故不给出
};

```

### (三) 算法实现

根据上面的讨论，可给出拓扑排序算法的粗略描述：

```

long TopoSort(图 g)
{
    long cnt=0;
    for (每个结点 v)
        if (v 入度为 0) v 进栈;
    while (栈不空)
    {
        从栈中弹出一个元素送 v;
        输出 v; cnt++;
        求出 v 的第 1 个直接可达邻接点 w;
        while (w 存在)
        {
            将 w 的入度域减 1;
            if (w 的入度域为 0) w 进栈;
            求出 v 的下个直接可达邻接点 w;
        } //while
    } //while
    return cnt;
}

```

下面是具体的 C++ 程序。在该程序中，为了节省存储，没有单独设置栈，而是利用结点数组 g 的入度域做栈，具体的说明，见下小节。

```

template <class TElem, class TEdgeElem>
long TopoSort(TGraphNodeAL<TElem, TEdgeElem> *g, long n, long *resu)
{ //对图 g 求拓扑序列，存入 resu(结点编号)，n 为图结点数目。
    //返回所求得的结点数。若返回值小于 n，则表示未能完全拓扑排序。
    long top, cnt, i;

```

```

TGraphEdgeAL<TEdgeElem> *p;

top=0;
cnt=0;
for (i=0; i<=n; i++)
if (g[i].inDegree == 0)
    { g[i].inDegree = top;    top = i; } //结点 i 进栈。用 g 的入度域做为栈。

while (top>0)
{
    i = top; top = g[top].inDegree; //出栈，栈顶元素送 i
    resu[cnt]=i; k++;
    p=g[i].firstOutEdge;
    while ( p!=NULL )
    {
        i=p->endNo;
        g[i].inDegree--;
        if (g[i].inDegree==0)
        {
            g[i].inDegree = top;    top = i; //结点 i 进栈
        }
        p=p->next;
    } //while (p!=...
}
return cnt;
}

```

☞ 我们这里是为了方便才使用栈。但使用栈不能识别并行成分。事实上，拓扑排序可以使用队列，这样也可以方便地识别并行成分。

💡 如果入度域是做为公用的数据成员，则它的值应该一直保持正确。但该程序运行后，入度域的值被改变了，故该程序有副作用！避免的方法是设立临时结构存入度值。

分析此算法的时间复杂度。假定图的结点数与边数分别为  $n$  与  $m$ 。初始化操作是扫描  $n$  个结点，将入度为 0 者进栈，时间复杂度为  $O(n)$ ，此后是两个嵌套着的 `while` 循环，直接根据循环条件分析循环体进行的次数比较困难，故换个角度进行分析。首先看进栈操作（及输出操作），因在无环情况下每个结点恰好进栈一次，故此操作有时间复杂度  $O(n)$ 。其次分析入度减 1 操作。在无环情况下，当排序结束时，每个边恰好被逻辑删除一次，故入度减 1 操作的时间复杂度为  $O(m)$ 。因这几种操作是拓扑排序算法的主体。

故整个算法的时间复杂度为  $O(n+m)$ 。

#### (四) 静态链栈的使用

在上面给出的程序中，为了节省空间，我们没有单独设立栈，而利用图结点（其存放在称为头数组的数组中）中的入度域构造一个链式栈。

在拓扑排序程序中，逻辑上栈存放当前入度为 0 且尚未输出的结点，而这些结点的入度域此时对该程序已无用，所以可用它们形成一个链，将它们链接在一条链中，从而起到栈的作用。

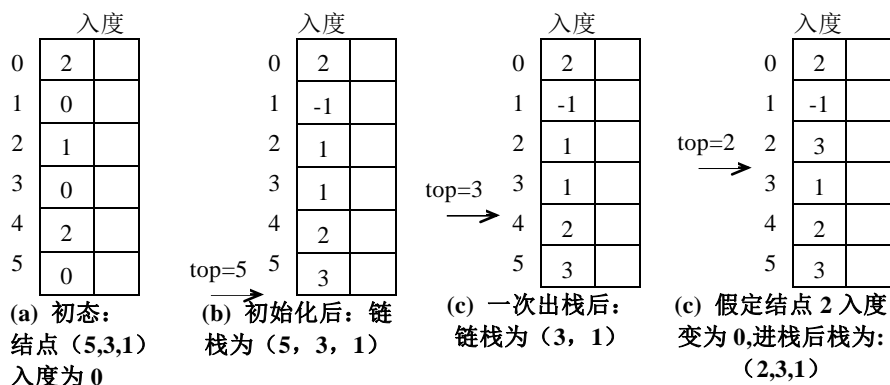


图 7-9 静态链式栈操作

具体实施方法是，对入度为 0 的结点，令它的入度域的值为下一个入度为 0 的结点在头指针数组中的序号（下标），最后一个入度为 0 的结点的入度域置特殊标志（如 -1）（充当栈底），最先一个入度为 0 的结点的位置（在数组中的下标）用一个指示器指示（如 top），充当栈顶指示器。

这种栈实质上是一种静态链式栈。这里给出的方法，不仅对拓扑排序本身有意义，更重要的是，它代表着一种方法。

下面给出具体的进栈和出栈操作方法。设 top 中存放栈顶元素对应的数组下标，i 是某个图结点的编号（与图数组下标兼容）。

##### 1. 进栈：结点 i 进栈：

```
g[i].inDegree = top;
top = i;
```

##### 2. 出栈：栈顶元素送 i：

```
i = top;
top = g[top].inDegree ;
```

图 7-9 给出了这种栈的进栈和出栈操作的例子。

也可以类似地实现静态链队。

§ 7.6 AOE 网与关键路径

求关键路径是对边加权的有向图的一种操作。AOE(Activity On Edge network)网与 AOV 网类似，也是一种被赋予了抽象语义的有向图，是许多实际问题的模型。

§ 7.6.1 AOE 网与关键路径的概念

(一) AOE 网

前面介绍过，对图结构，在实际应用时可给结点或边赋予某种描述量——权，这里我们考虑边带权的有向图。如果将有向图的结点与边按下列所述赋予抽象意义，则该种有向图就称为 AOE 网。

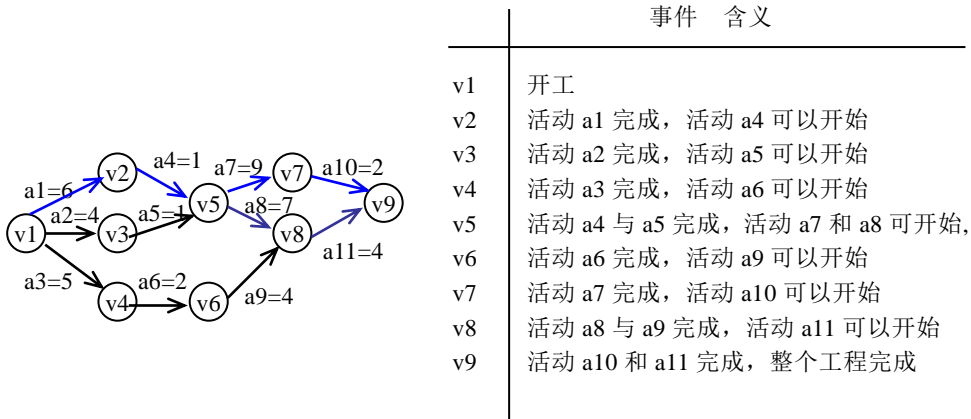


图 7-10 AOE 网举例

- 边（弧）：代表活动（操作）；
  - 边权：代表活动的持续时间。记边  $a_k=\langle i, j \rangle$  的权为  $\text{len}(a_k)$  或  $\text{len}(i, j)$ ；
  - 结点：代表事件（状态）。它表示它的各入边代表的活动均已完成，而它的出边代表的活动可以开始。
- 事实上，某结点代表的事件是它的各入边代表的活动的共同作用结果，同时也是它

的各出边代表的活动的启动条件。

AOE 网中没有入边的结点称为始点，没有出边的结点称为终点。

AOE 一般用来描述工程进度，结点表示工程进展中的状态，边表示子任务。图 7-10 就是一个 AOE 网，它可以看作是一个具有 11 项子任务和 9 个状态的假想工程的进度图。

## (二) AOE 网的操作

针对 AOE 网的操作一般有下列几种：

- 关键路径 CPM(Critical Path Method)。这种操作最早用于维修与建筑行业中工期进度估算。
- 性能估计与复审 PERT(Performance Evaluation and Review Technique)：该项操作最初是为了研制北极星式导弹系统而引入的。
- 资源分配与多工程调度 RAMPS(Resource Allocation and Multi-Project Scheduling)

## (三) 关键路径的若干基本概念

下面的阐述中，设 AOE 网的起点为  $v_0$  终点为  $v_n$ 。

### 1. 关键路径

AOE 网中，从事件  $i$  到  $j$  的路径中，加权长度最大者称为  $i$  到  $j$  的**关键路径** (Critical Path)，记为  $cp(i, j)$ 。特别地，始点  $0$  到终点  $n$  的关键路径  $cp(0, n)$  是整个 AOE 的关键路径。

显然，关键路径决定着 AOE 网的工期，关键路径的长度就是 AOE 网代表的工程所需的最小工期。

### 2. 事件最早/晚发生时间

**事件  $v_i$  的最早发生时间**  $ve(i)$  定义为：从始点到  $v_i$  的最长（加权）路径长度，即  $cp(0, i)$

**事件  $v_i$  的最晚发生时间**  $vl(i)$  定义为：在不拖延整个工期的条件下， $v_i$  的可能的最晚发生时间。即  $vl(i) = ve(n) - cp(i, n)$

### 3. 活动最早/晚开始时间

活动  $a_k = \langle v_i, v_j \rangle$  的**最早开始时间**  $e(k)$ ：等于事件  $v_i$  的最早发生时间，即

$$e(k) = ve(i) = cp(0, i)$$

活动  $a_k = \langle v_i, v_j \rangle$  的**最晚开始时间**  $l(k)$  定义为：在不拖延整个工期的条件下，该活动的允许的最迟开始时间，即

$$l(k) = vl(j) - len(i, j)$$

这里， $vl(j)$  是事件  $j$  的允许的最晚发生时间， $len(i, j)$  是  $a_k$  的权。



活动  $a_k$  的**最大可利用时间**：定义为  $l(k)-e(k)$

#### 4. 关键活动

若活动  $a_k$  的最大可利用时间等于 0（即  $l(k)=e(k)$ ），则称  $a_k$  为**关键活动**，否则为**非关键活动**。

显然，关键活动的延期，会使整个工程延期。但非关键活动不然，只要它的延期量不超过它的最大可利用时间，就不会影响整个工期。

关键路径的概念，也可以用这里的关键活动定义，即有下面的：

**定理 7-2** 某路径为关键路径的充分必要条件是，它上的各活动均为关键活动。

证：设  $v_0$  与  $v_n$  分别为 AOE 网的始点与终点。用  $\text{len}(P)$  表示路径/边  $P$  的加权长度，先证必要性。设  $\text{cp}(0, n)$  是一条关键路径， $a_k=\langle i, j \rangle$  是它上的任一个活动。

由前面的定义知，

$$\text{len}(\text{cp}(0, n)) = \text{ve}(n)$$

$$\text{len}(\text{cp}(0, n)) = \text{len}(\text{cp}(0, i)) + \text{len}(a_k) + \text{len}(\text{cp}(j, n))$$

故

$$\text{len}(\text{cp}(0, i)) + \text{len}(a_k) + \text{len}(\text{cp}(j, n)) = \text{ve}(n),$$

即

$$\begin{aligned} \text{len}(\text{cp}(0, i)) &= \text{ve}(n) - \text{len}(\text{cp}(j, n)) - \text{len}(a_k) \\ &= \text{vl}(j) - \text{len}(a_k) \\ &= l(k) \end{aligned}$$

又

$$\text{len}(0, i) = \text{ve}(i) = e(k)$$

从而

$$e(k)=l(k)$$

这表明，活动  $a_k$  是关键活动。必要性得证。

再考查充分性。现假定某路径  $p(0, n)$  上各活动均为关键活动，但  $p(0, n)$  不是关键路径。这样就有，

$$\text{len}(p(0, n)) < \text{ve}(n)$$

从  $p(0, n)$  上任取一活动  $a_k = \langle i, j \rangle$  考查之，显然

$$\text{len}(p(0, n)) = \text{len}(p(0, i)) + \text{len}(a_k) + \text{len}(p(j, n))$$

故

$$\text{len}(p(0, i)) + \text{len}(a_k) + \text{len}(p(j, n)) < \text{ve}(n)$$

即

$$\text{len}(p(0, i)) < \text{ve}(n) - \text{len}(p(j, n)) - \text{len}(a_k) = \text{ve}(j) - \text{len}(a_k) = l(k)$$

但

$$\text{len}(p(0, i)) = \text{ve}(i) = e(k)$$

从而

$$e(k) < l(k)$$

这与  $a_k$  是关键活动矛盾，充分性亦得证。

### § 7.6.2 关键路径的识别

进行关键路径分析，目的是寻找合理的资源调配方案（这里的资源是指能使活动进行的人力或物力），使 AOE 网代表的工程尽快完成。为此，需先识别关键路径。只有缩短关键路径上的活动（关键活动）才有可能缩短整个工期。如果存在多条关键路径，只缩短部分关键路径上的活动还不够，要能使各关键路径同时缩短才有效。在这种情况下，识别关键路径的公共点就变得很重要，因为改变这些公共点的工期，可同时缩短多条关键路径，好钢用在了刀刃上。这些问题都是以识别关键路径为基础的，所以我们重点考虑如何识别关键路径。

#### (一) 基本算法

关键路径算法是一种典型的动态规划法，这点在学了后面的算法设计方法后就会看到。下面就来介绍该算法。

设图  $G=(V, E)$  是个 AOE 网，结点编号为  $1, 2, \dots, n$ ，其中结点 1 与  $n$  分别为始点和终点， $a_k = \langle i, j \rangle \in E$  是  $G$  的一个活动。

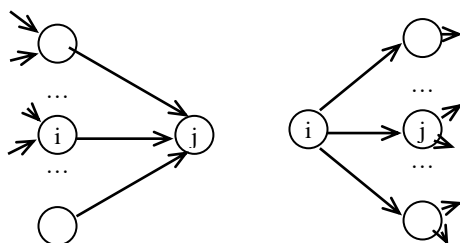


图 7-11 计算  $ve$  和  $vl$  的示意

根据前面给出的定义，可推出活动的最早及最晚发生时间的计算方法：

$$e(k) = ve(i)$$

$$l(k) = ve(j) - len(i, j)$$

结点的最早发生时间的计算，需按拓扑次序递推：

$$ve(1) = 0$$

$$ve(j) = \text{MAX}\{ ve(i) + len(i, j) \} \quad \text{对所有 } \langle i, j \rangle \in E \text{ 的 } i$$

结点的最晚发生时间的计算，需按逆拓扑次序递推：

$$vl(n) = ve(n)$$

$$vl(i) = \text{MIN}\{ vl(j) - len(i, j) \} \quad \text{对所有 } \langle i, j \rangle \in E \text{ 的 } j$$

关于  $ve$  与  $vl$  的求法, 可参阅图 7-11。

这种计算方法, 依赖于拓扑排序, 即计算  $ve(j)$  前, 应已求得  $j$  的各前趋结点的  $ve$  值, 而计算  $vl(i)$  前, 应已求得  $i$  的各后继结点的  $vl$  值。 $ve$  的计算可在拓扑排序过程中进行, 即在每输出一个结点  $i$  后, 在删除  $i$  的每个出边  $\langle i, j \rangle$  (即入度减 1) 的同时, 执行

```
if (  $ve[i] + len(i, j)$  ) >  $ve[j]$  )
     $ve[j] = ve[i] + len(i, j)$ 
```

实际上, 该操作对  $i$  的每个后继  $j$  分别进行一次。因此对程序作少量扩充即可求得  $ve$ 。

$vl$  的值可按类似的方法在逆拓扑排序过程 (即直接按与拓扑序列相反的次序输出结点的过程) 中求得, 但一般不必专门这样进行。事实上, 通过逆方向使用拓扑序列即可递推出各  $vl$  的值, 假定拓扑序列是  $topoSeq$ , 则  $vl$  的值的求法为 (结点编号为  $1 \sim n$ )。

```
for (  $k=1$ ;  $k \leq n$ ;  $k++$  )  $vl[k] = ve[n]$ ;    //初始化
for (  $k=n$ ;  $k \geq 1$ ;  $k--$  )
{
     $i = topoSeq[k]$ ;
     $j = (i \text{ 的第 } 1 \text{ 个出点})$ ;
    while (  $j$  存在 )
    {
        if (  $vl[j] - len(i, j) < vl[i]$  )
             $vl[i] = vl[j] - len(i, j)$ ;
         $j = (i \text{ 的下一个出点})$ ;
    }
}
```

具体的程序实现, 留作练习。

## 本章小结

图的基本特征是图中元素的前驱与后继的个数都没有限制。图是最复杂的非线性结构, 它的表达能力也最强, 前面介绍的线性结构、树等均是它的特例。

图的存储方式一般有两类: 用边的集合表示图和用链接关系表示图。其中, 边的集合方式有邻接矩阵, 链接方式有邻接表、十字链表和邻接多重表等。

图结构的接口主要有: 求某结点的各入点/出点, 求某结点的各入边/出边、遍历 (深度优先和广度优先)。图比较复杂, 还有一些较大的但可以抽象化的操作, 如拓扑排序、

关键路径、最小生成树、最短路径等。其中后两种我们安排在最后一章介绍。事实上，对图的分析，还有许多方面（如网络流量问题、连通性问题、图匹配问题等），这些内容传统地集中在称为网络分析的教程中。

## 习 题

1. 对图 7-1，分别写出从结点 1 出发的深度和广度优先遍历结果。
2. 编写求图的每个结点的层号的程序。
3. 针对图的十字链表存储结构，编写深度优先遍历图的程序。
4. 编写图的邻接表的逆串行化（根据图结构，输出图的边集）程序。
5. 编写程序，生成图的深度优先树。
6. 编写程序，生成图的广度优先树。
7. 改进深度优先遍历程序，使它能同时为每个结点生成对应的深度优先树中的层次号。
8. 改进深度优先遍历程序，使它能同时求出出发点到其他各点的路径及其长度。
9. 改进深度优先遍历程序，使它能同时求出各结点的时间戳。
10. 改进深度优先遍历程序，使它能同时求出相应的遍历括号。
11. 改进广度优先遍历程序，使它能同时求出各结点的层号。
12. 设图用邻接矩阵存储，编写相应的拓扑排序程序。
13. 修改拓扑排序程序，使它使用“静态链队”（参照静态链栈的方法）。
14. 修改拓扑排序程序，使它能同时标识出各并行组。
15. 对图 7-10，分别求各事件的最早和最晚发生时间、各活动的最早和最晚开始时间。
16. 编写计算各结点的最早和最晚发生时间、各活动的最早和最晚开始时间的程序。
17. 编写求关键路径的程序。
18. 编写程序实现深度优先拓扑排序。