

第6章



4.4节曾仿效古希腊英雄忒修斯，以栈等基本数据结构模拟线绳和粉笔，展示了试探回溯策略的应用技巧。实际上，这一技巧可进一步推广至更为一般性的场合，包括可以图结构描述的应用问题，从而导出一系列对应的图算法。

忒修斯取得成功的关键在于，借助线绳掌握迷宫内各通道之间的联接关系。在很多应用中，能否有效描述和利用这类信息，同样至关重要。一般地，这类信息往往可表述为定义于一组对象之间的二元关系，比如城市交通图中，联接于各公交站之间的街道，或者互联网中，联接于IP节点之间的路由，等等。尽管在某种程度上，第5章所介绍的树结构也可用以表示这种二元关系，但仅限于父、子节点之间。相互之间均可能存在二元关系的一组对象，从数据结构的角度分类，属于非线性结构(**non-linear structure**)。此类一般性的二元关系，属于图论(**Graph Theory**)的研究范畴。从算法的角度对此类结构的处理策略，与上一章相仿，也是通过遍历将其转化为半线性结构，进而借助树结构已有的处理方法和技巧，最终解决问题。

以下首先简要介绍图的基本概念和术语，已有相关基础的读者可直接跳过。接下来，介绍如何实现作为抽象数据类型的图结构，主要讨论邻接矩阵和邻接表两种实现方式。然后，从遍历的角度介绍将图转化为树的典型方法，包括广度优先搜索和深度优先搜索。进而，分别以拓扑排序和双连通域分解为例，介绍利用基本数据结构并基于遍历模式，设计图算法的主要方法。最后，从“数据结构决定遍历次序”的观点出发，将所有遍历算法概括并统一为最佳优先遍历这一模式。如此，我们不仅能够更加准确和深刻地理解不同图算法之间的共性与联系，更可以学会通过选择和改进数据结构，高效地设计并实现各种图算法——这也是本章的重点与精髓。

§ 6.1 概述

■ 图

图结构是描述和解决实际应用问题的一种基本而有力的工具。所谓的图(**graph**)，可定义为 $G = (V, E)$ 。其中，集合V中的元素称作顶点(**vertex**)；集合E中的元素分别对应于V中的某一对顶点(u, v)，表示它们之间存在某种关系，故亦称作边(**edge**)^①。一种直观显示图结构的方法是，用小圆圈或小方块代表顶点，用联接于其间的直线段或曲线弧表示对应的边。

从计算的需求出发，我们约定V和E均为有限集，通常将其规模分别记 $n = |V|$ 和 $e = |E|$ 。

■ 无向图、有向图及混合图

若边(u, v)所对应顶点u和v的次序无所谓，则称作无向边(**undirected edge**)，例如表示同学关系的边。反之若u和v不对等，则称(u, v)为有向边(**directed edge**)，例如描述企业与银行之间的借贷关系，或者程序之间的相互调用关系的边。

^① 在某些文献中，顶点也称作节点(**node**)，边亦称作弧(**arc**)，本章则统一称作顶点和边。

如此，无向边 (u, v) 也可记作 (v, u) ，而有向的 (u, v) 和 (v, u) 则不可混淆。这里约定，有向边 (u, v) 从 u 指向 v ，其中 u 称作该边的起点（origin）或尾顶点（tail），而 v 称作该边的终点（destination）或头顶点（head）。

若 E 中各边均无方向，则 G 称作无向图（undirected graph，简称undigraph）。例如在描述影视演员相互合作关系的图 G 中，若演员 u 和 v 若曾经共同出演过至少一部影片，则在他（她）们之间引入一条边 (u, v) 。反之，若 E 中只含有向边，则 G 称作有向图（directed graph，简称digraph）。例如在C++类的派生关系图中，从顶点 u 指向顶点 v 的有向边，意味着类 u 派生自类 v 。特别地，若 E 同时包含无向边和有向边，则 G 称作混合图（mixed graph）。例如在北京市内交通图中，有些道路是双行的，另一些是单行的，对应地可分别描述为无向边和有向边。

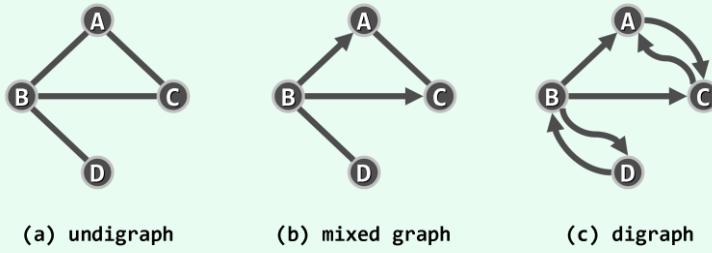


图6.1 (a)无向图、(b)混合图和(c)有向图

相对而言，有向图的通用性更强，因为无向图和混合图都可转化为有向图——如图6.1所示，每条无向边 (u, v) 都可等效地替换为对称的一对有向边 (u, v) 和 (v, u) 。因此，本章将主要针对有向图，介绍图结构及其算法的具体实现。

度

对于任何边 $e = (u, v)$ ，称顶点 u 和 v 彼此邻接（adjacent），互为邻居；而它们都与边 e 彼此关联（incident）。在无向图中，与顶点 v 关联的边数，称作 v 的度数（degree），记作 $\deg(v)$ 。以图6.1(a)为例，顶点{ A, B, C, D }的度数为{ 2, 3, 2, 1 }。

对于有向边 $e = (u, v)$ ， e 称作 u 的出边（outgoing edge）、 v 的入边（incoming edge）。 v 的出边总数称作其出度（out-degree），记作 $\text{outdeg}(v)$ ；入边总数称作其入度（in-degree），记作 $\text{indeg}(v)$ 。在图6.1(c)中，各顶点的出度为{ 1, 3, 1, 1 }，入度为{ 2, 1, 2, 1 }。

简单图

联接于同一顶点之间的边，称作自环（self-loop）。在某些特定的应用中，这类边可能的确具有意义——比如在城市交通图中，沿着某条街道，有可能不需经过任何交叉路口即可直接返回原处。不含任何自环的图称作简单图（simple graph），也是本书主要讨论的对象。

通路与环路

所谓路径或通路（path），就是由 $m + 1$ 个顶点与 m 条边交替而成的一个序列：

$$\pi = \{ v_0, e_1, v_1, e_2, v_2, \dots, e_m, v_m \}$$

且对任何 $0 < i \leq m$ 都有 $e_i = (v_{i-1}, v_i)$ 。也就是说，这些边依次地首尾相联。其中沿途边的总数 m ，亦称作通路的长度，记作 $|\pi| = m$ 。

为简化描述，也可依次给出通路沿途的各个顶点，而省略联接于其间的边，即表示为：

$$\pi = \{ v_0, v_1, v_2, \dots, v_m \}$$

图6.2(a)中的{ C, A, B, A, D }, 即是从顶点C到D的一条通路, 其长度为4。可见, 尽管通路上的边必须互异, 但顶点却可能重复。沿途顶点互异的通路, 称作简单通路 (simple path)。在图6.2(b)中, { C, A, D, B }即是从顶点C到B的一条简单通路, 其长度为3。

特别地, 对于长度 $m \geq 1$ 的通路 π , 若起止顶点相同 (即 $v_0 = v_m$), 则称作环路 (cycle), 其长度也取作沿途边的总数。图6.3(a)中, { C, A, B, A, D, B, C }即是一条环路, 其长度为6。反之, 不含任何环路的有向图, 称作有向无环图 (directed acyclic graph, DAG)。

同样, 尽管环路上的各边必须互异, 但顶点却也可能重复。反之若沿途除 $v_0 = v_m$ 外所有顶点均互异, 则称作简单环路 (simple cycle)。例如, 图6.3(b)中的{ C, A, B, C }即是一条简单环路, 其长度为3。特别地, 经过图中各边一次且恰好一次的环路, 称作欧拉环路 (Eulerian tour) ——当然, 其长度也恰好等于图中边的总数 e 。

图6.4(a)中的{ C, A, B, A, D, C, D, B, C }即是一条欧拉环路, 其长度为8。对偶地, 经过图中各顶点一次且恰好一次的环路, 称作哈密尔顿环路 (Hamiltonian tour), 其长度亦等于构成环路的边数。图6.4(b)中, { C, A, D, B, C }即是一条长度为4的哈密尔顿环路。

■ 带权网络

图不仅需要表示顶点之间是否存在某种关系, 有时还需要表示这一关系的具体细节。以铁路运输为例, 可以用顶点表示城市, 用顶点之间的联边, 表示对应的城市之间是否有客运铁路联接; 同时, 往往还需要记录各段铁路的长度、承运能力, 以及运输成本等信息。

为适应这类应用要求, 需通过一个权值函数, 为每一边 e 指定一个权重 (weight), 比如 $wt(e)$ 即为边 e 的权重。各边均带有权重的图, 称作带权图 (weighted graph) 或带权网络 (weighted network), 有时也简称网络 (network), 记作 $G(V, E, wt())$ 。

■ 复杂度

与其它算法一样, 图算法也需要就时间性能和空间性能, 进行分析和比较。相应地, 问题的输入规模, 也应该以顶点数与边数的总和 ($n + e$) 来度量。不难看出, 无论顶点多少, 边数都有可能为0。那么反过来, 在包含 n 个顶点的图中, 至多可能包含多少条边呢?

对于无向图, 每一对顶点至多贡献一条边, 故总共不超过 $n(n - 1)/2$ 条边, 且这个上界由完全图达到。对于有向图, 每一对顶点都可能贡献 (互逆的) 两条边, 因此至多可有 $n(n - 1)$ 条边。总而言之, 必有 $e = O(n^2)$ 。

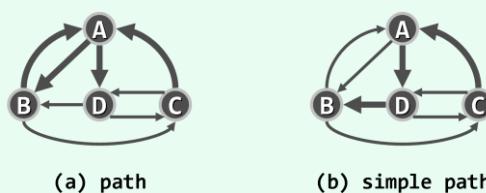


图6.2 通路与简单通路

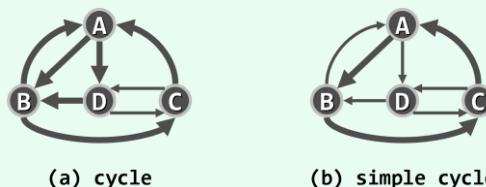


图6.3 环路与简单环路

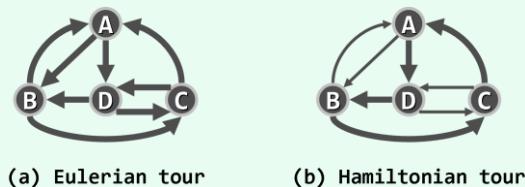


图6.4 欧拉环路与哈密尔顿环路

§ 6.2 抽象数据类型

6.2.1 操作接口

作为抽象数据类型，图支持的操作接口分为边和顶点两类，分列于表6.1和表6.2。

表6.1 图ADT支持的边操作接口

操作接口	功能描述
e()	边总数 E
exist(v, u)	判断联边(v, u)是否存在
insert(v, u)	引入从顶点v到u的联边
remove(v, u)	删除从顶点v到u的联边
type(v, u)	边在遍历树中所属的类型
edge(v, u)	边所对应的数据域
weight(v, u)	边的权重

表6.2 图ADT支持的顶点操作接口

操作接口	功能描述
n()	顶点总数 V
inser(v)	在顶点集V中插入新顶点v
remo e(v)	将顶点v从顶点集中删除
inDegree(v) outDegree(v)	顶点v的入度、出度
firstNbr(v)	顶点v的首个邻接顶点
nextNbr(v, u)	在v的邻接顶点中，u的后继
status(v)	顶点v的状态
dTime(v)、fTime(v)	顶点v的时间标签
parent(v)	顶点v在遍历树中的父节点
priority(v)	顶点v在遍历树中的权重

6.2.2 Graph模板类

代码6.1以抽象模板类的形式，给出了图ADT的具体定义。

```

1 typedef enum { UNDISCOVERED, DISCOVERED, VISITED } VStatus; //顶点状态
2 typedef enum { UNDETERMINED, TREE, CROSS, FORWARD, BACKWARD } EType; //边在遍历树中所属的类型
3
4 template <typename Tv, typename Te> //顶点类型、边类型
5 class Graph { //图Graph模板类
6 private:
7     void reset() { //所有顶点、边的辅助信息复位
8         for ( int i = 0; i < n; i++ ) { //所有顶点的
9             status ( i ) = UNDISCOVERED; dTime ( i ) = fTime ( i ) = -1; //状态，时间标签
10            parent ( i ) = -1; priority ( i ) = INT_MAX; // (在遍历树中的)父节点，优先级数
11            for ( int j = 0; j < n; j++ ) //所有边的
12                if ( exists ( i, j ) ) type ( i, j ) = UNDETERMINED; //类型
13        }
14    }
15    void BFS ( int, int& ); // (连通域) 广度优先搜索算法
16    void DFS ( int, int& ); // (连通域) 深度优先搜索算法
17    void BCC ( int, int&, Stack<int>& ); // (连通域) 基于DFS的双连通分量分解算法
18    bool TSort ( int, int&, Stack<Tv>* ); // (连通域) 基于DFS的拓扑排序算法
19    template <typename PU> void PFS ( int, PU ); // (连通域) 优先级搜索框架

```

```

20 public:
21 // 顶点
22     int n; //顶点总数
23     virtual int insert ( Tv const& ) = 0; //插入顶点，返回编号
24     virtual Tv remove ( int ) = 0; //删除顶点及其关联边，返回该顶点信息
25     virtual Tv& vertex ( int ) = 0; //顶点v的数据（该顶点的确存在）
26     virtual int inDegree ( int ) = 0; //顶点v的入度（该顶点的确存在）
27     virtual int outDegree ( int ) = 0; //顶点v的出度（该顶点的确存在）
28     virtual int firstNbr ( int ) = 0; //顶点v的第一个邻接顶点
29     virtual int nextNbr ( int, int ) = 0; //顶点v的（相对于顶点j的）下一邻接顶点
30     virtual VStatus& status ( int ) = 0; //顶点v的状态
31     virtual int& dTime ( int ) = 0; //顶点v的时间标签dTime
32     virtual int& fTime ( int ) = 0; //顶点v的时间标签fTime
33     virtual int& parent ( int ) = 0; //顶点v在遍历树中的父亲
34     virtual int& priority ( int ) = 0; //顶点v在遍历树中的优先级数
35 // 边：这里约定，无向边均统一转化为方向互逆的一对有向边，从而将无向图视作有向图的特例
36     int e; //边总数
37     virtual bool exists ( int, int ) = 0; //边(v, u)是否存在
38     virtual void insert ( Te const&, int, int, int ) = 0; //在顶点v和u之间插入权重为w的边e
39     virtual Te remove ( int, int ) = 0; //删除顶点v和u之间的边e，返回该边信息
40     virtual EType& type ( int, int ) = 0; //边(v, u)的类型
41     virtual Te& edge ( int, int ) = 0; //边(v, u)的数据（该边的确存在）
42     virtual int& weight ( int, int ) = 0; //边(v, u)的权重
43 // 算法
44     void bfs ( int ); //广度优先搜索算法
45     void dfs ( int ); //深度优先搜索算法
46     void bcc ( int ); //基于DFS的双连通分量分解算法
47     Stack<Tv>* tSort ( int ); //基于DFS的拓扑排序算法
48     void prim ( int ); //最小支撑树Prim算法
49     void dijkstra ( int ); //最短路径Dijkstra算法
50     template <typename PU> void pfs ( int, PU ); //优先级搜索框架
51 };

```

代码6.1 图ADT操作接口

仍为简化起见，这里直接开放了变量n和e。除以上所列的操作接口，这里还明确定义了顶点和边可能处于的若干状态，并通过内部接口reset()复位顶点和边的状态。

图的部分基本算法在此也以操作接口的形式供外部用户直接使用，比如广度优先搜索、深度优先搜索、双连通分量分解、最小支撑树、最短路径等。为求解更多的具体应用问题，读者可照此模式，独立地补充相应的算法。

就功能而言，这些算法均超脱于图结构的具体实现方式，借助统一的顶点和边ADT操作接口直接编写。尽管如此，正如以下即将看到的，图算法的时间、空间性能，却与图结构的具体实现方式紧密相关，在这方面的理解深度，也将反映和决定我们对图结构的驾驭与运用能力。

§ 6.3 邻接矩阵

6.3.1 原理

邻接矩阵（adjacency matrix）是图ADT最基本的实现方式，使用方阵 $A[n][n]$ 表示由 n 个顶点构成的图，其中每个单元，各自负责描述一对顶点之间可能存在的邻接关系，故此得名。

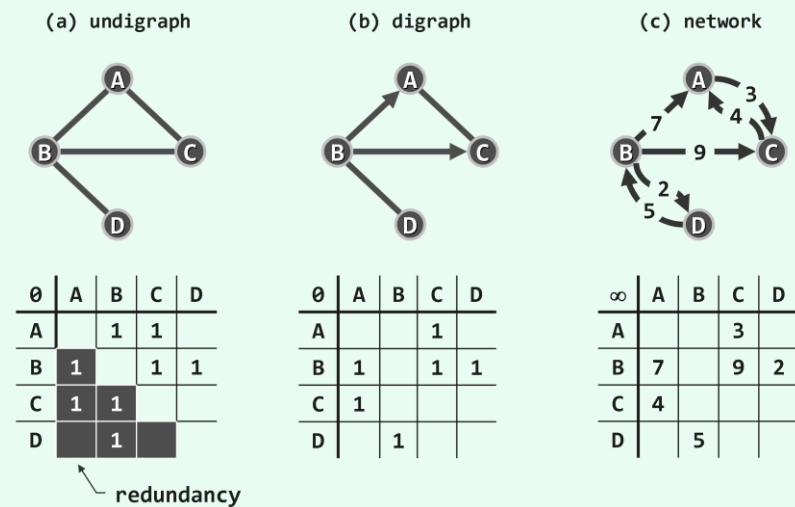


图6.5 邻接矩阵（空白单元对应的边不存在，其统一取值标注于矩阵最左上角）

对于无权图，存在（不存在）从顶点 u 到 v 的边，当且仅当 $A[u][v] = 1 (0)$ 。图6.5(a)和(b)即为无向图和有向图的邻接矩阵实例。

这一表示方式，不难推广至带权网络。此时如图(c)所示，矩阵各单元可从布尔型改为整型或浮点型，记录所对应边的权重。对于不存在的边，通常统一取值为 ∞ 或 0 。

6.3.2 实现

基于以上原理与构思实现的图结构如代码6.2所示。

```

1 #include "../Vector/Vector.h" //引入向量
2 #include "../Graph/Graph.h" //引入图ADT
3
4 template <typename Tv> struct Vertex { //顶点对象（为简化起见，并未严格封装）
5     Tv data; int inDegree, outDegree; VStatus status; //数据、出入度数、状态
6     int dTime, fTime; //时间标签
7     int parent; int priority; //在遍历树中的父节点、优先级数
8     Vertex ( Tv const& d = ( Tv ) 0 ) : //构造新顶点
9         data ( d ), inDegree ( 0 ), outDegree ( 0 ), status ( UNDISCOVERED ),
10        dTime ( -1 ), fTime ( -1 ), parent ( -1 ), priority ( INT_MAX ) {} //暂不考虑权重溢出
11    };
12
13 template <typename Te> struct Edge { //边对象（为简化起见，并未严格封装）
14     Te data; int weight; EType type; //数据、权重、类型
15     Edge ( Te const& d, int w ) : data ( d ), weight ( w ), type ( UNDETERMINED ) {} //构造
16    };
17
18 template <typename Tv, typename Te> //顶点类型、边类型

```

```

19 class GraphMatrix : public Graph<Tv, Te> { //基于向量，以邻接矩阵形式实现的图
20 private:
21     Vector< Vertex< Tv > > V; //顶点集（向量）
22     Vector< Vector< Edge< Te > * > > E; //边集（邻接矩阵）
23 public:
24     GraphMatrix() { n = e = 0; } //构造
25     ~GraphMatrix() { //析构
26         for ( int j = 0; j < n; j++ ) //所有动态创建的
27             for ( int k = 0; k < n; k++ ) //边记录
28                 delete E[j][k]; //逐条清除
29     }
30 // 顶点的基本操作：查询第i个顶点 ( 0 <= i < n )
31     virtual Tv& vertex ( int i ) { return V[i].data; } //数据
32     virtual int inDegree ( int i ) { return V[i].inDegree; } //入度
33     virtual int outDegree ( int i ) { return V[i].outDegree; } //出度
34     virtual int firstNbr ( int i ) { return nextNbr ( i, n ); } //首个邻接顶点
35     virtual int nextNbr ( int i, int j ) //相对于顶点j的下一邻接顶点（改用邻接表可提高效率）
36     { while ( ( -1 < j ) && ( !exists ( i, --j ) ) ); return j; } //逆向线性试探
37     virtual VStatus& status ( int i ) { return V[i].status; } //状态
38     virtual int& dTime ( int i ) { return V[i].dTime; } //时间标签dTime
39     virtual int& fTime ( int i ) { return V[i].fTime; } //时间标签fTime
40     virtual int& parent ( int i ) { return V[i].parent; } //在遍历树中的父亲
41     virtual int& priority ( int i ) { return V[i].priority; } //在遍历树中的优先级数
42 // 顶点的动态操作
43     virtual int insert ( Tv const& vertex ) { //插入顶点，返回编号
44         for ( int j = 0; j < n; j++ ) E[j].insert ( NULL ); n++; //各顶点预留一条潜在的关联边
45         E.insert ( Vector<Edge<Te>*> ( n, n, ( Edge<Te>* ) NULL ) ); //创建新顶点对应的边向量
46         return V.insert ( Vertex<Tv> ( vertex ) ); //顶点向量增加一个顶点
47     }
48     virtual Tv remove ( int i ) { //删除第i个顶点及其关联边 ( 0 <= i < n )
49         for ( int j = 0; j < n; j++ ) //所有出边
50             if ( exists ( i, j ) ) { delete E[i][j]; V[j].inDegree--; } //逐条删除
51         E.remove ( i ); n--; //删除第i行
52         Tv vBak = vertex ( i ); V.remove ( i ); //删除顶点i
53         for ( int j = 0; j < n; j++ ) //所有入边
54             if ( Edge<Te> * e = E[j].remove ( i ) ) { delete e; V[j].outDegree--; } //逐条删除
55         return vBak; //返回被删除顶点的信息
56     }
57 // 边的确认操作
58     virtual bool exists ( int i, int j ) //边(i, j)是否存在
59     { return ( 0 <= i ) && ( i < n ) && ( 0 <= j ) && ( j < n ) && E[i][j] != NULL; }
60 // 边的基本操作：查询顶点i与j之间的联边 ( 0 <= i, j < n 且 exists(i, j) )

```

```

61     virtual EType& type ( int i, int j ) { return E[i][j]->type; } //边(i, j)的类型
62     virtual Te& edge ( int i, int j ) { return E[i][j]->data; } //边(i, j)的数据
63     virtual int& weight ( int i, int j ) { return E[i][j]->weight; } //边(i, j)的权重
64 // 边的动态操作
65     virtual void insert ( Te const& edge, int w, int i, int j ) { //插入权重为w的边e = (i, j)
66         if ( exists ( i, j ) ) return; //确保该边尚不存在
67         E[i][j] = new Edge<Te> ( edge, w ); //创建新边
68         e++; V[i].outDegree++; V[j].inDegree++; //更新边计数与关联顶点的度数
69     }
70     virtual Te remove ( int i, int j ) { //删除顶点i和j之间的联边 (exists(i, j))
71         Te eBak = edge ( i, j ); delete E[i][j]; E[i][j] = NULL; //备份后删除边记录
72         e--; V[i].outDegree--; V[j].inDegree--; //更新边计数与关联顶点的度数
73         return eBak; //返回被删除边的信息
74     }
75 };

```

代码6.2 基于邻接矩阵实现的图结构

可见，这里利用第2章实现并封装的**Vector**结构，在内部将所有顶点组织为一个向量**V[]**；同时通过嵌套定义，将所有（潜在的）边组织为一个二维向量**E[][]**——亦即邻接矩阵。

每个顶点统一表示为**Vertex**对象，每条边统一表示为**Edge**对象。

边对象的属性**weight**统一简化为整型，既可用于表示无权图，亦可表示带权网络。

6.3.3 时间性能

按照代码6.2的实现方式，各顶点的编号可直接转换为其在邻接矩阵中对应的秩，从而使得图ADT中所有的静态操作接口，均只需 $O(1)$ 时间——这主要是得益于向量“循秩访问”的特长与优势。另外，边的静态和动态操作也仅需 $O(1)$ 时间——其代价是邻接矩阵的空间冗余。

然而，这种方法并非完美无缺。其不足主要体现在，顶点的动态操作接口均十分耗时。为了插入新的顶点，顶点集向量**V[]**需要添加一个元素；边集向量**E[][]**也需要增加一行，且每行都需要添加一个元素。顶点删除操作，亦与此类似。不难看出，这些恰恰也是向量结构固有的不足。

好在通常的算法中，顶点的动态操作远少于其它操作。而且，即便计入向量扩容的代价，就分摊意义而言，单次操作的耗时亦不过 $O(n)$ （习题[6-2]）。

6.3.4 空间性能

上述实现方式所用空间，主要消耗于邻接矩阵，亦即其中的二维边集向量**E[][]**。每个**Edge**对象虽需记录多项信息，但总体不过常数。根据2.4.4节的分析结论，**Vector**结构的装填因子始终不低于50%，故空间总量渐进地不超过 $O(n \times n) = O(n^2)$ 。

当然，对于无向图而言，仍有改进的余地。如图6.5(a)所示，无向图的邻接矩阵必为对称阵，其中除自环以外的每条边，都被重复地存放了两次。也就是说，近一半的单元都是冗余的。为消除这一缺点，可采用压缩存储等技巧，进一步提高空间利用率（习题[6-4]）。

§ 6.4 邻接表

6.4.1 原理

即便就有向图而言， $\Theta(n^2)$ 的空间亦有改进的余地。实际上，如此大的空间足以容纳所有潜在的边。然而实际应用所处理的图，所含的边通常远远少于 $\Theta(n^2)$ 。比如在平面图之类的稀疏图（sparse graph）中，边数渐进地不超过 $\Theta(n)$ ，仅与顶点总数大致相当（习题[6-3]）。

由此可见，邻接矩阵的空间效率之所以低，是因为其中大量单元所对应的边，通常并未在图中出现。因静态空间管理策略导致的此类问题，并非首次出现，比如此前的2.4节，就曾指出这类缺陷并试图改进。既然如此，为何不仿照3.1节的思路，将这里的向量替换为列表呢？

是的，按照这一思路，的确可以导出图结构的另一种表示与实现形式。

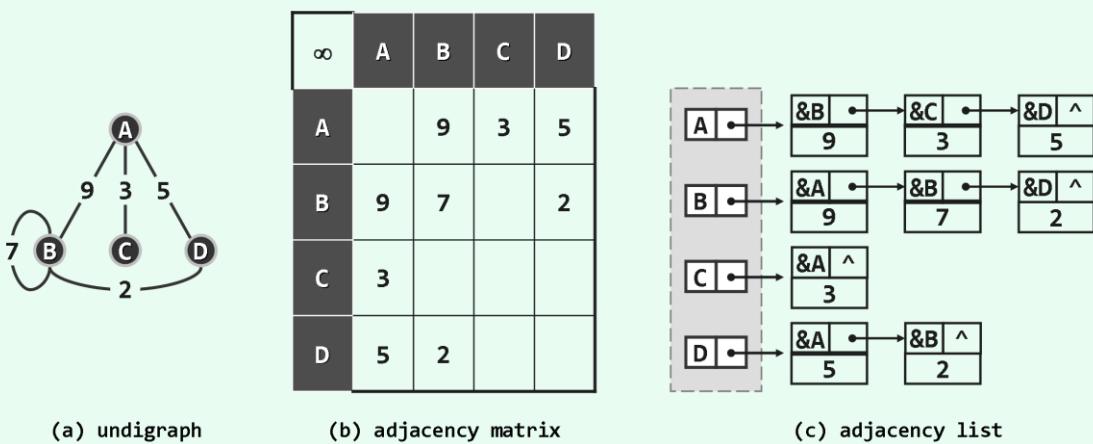


图6.6 以邻接表方式描述和实现图

以如图6.6(a)所示的无向图为例，只需将如图(b)所示的邻接矩阵，逐行地转换为如图(c)所示的一组列表，即可分别记录各顶点的关联边（或等价地，邻接顶点）。这些列表，也因此称作邻接表（adjacency list）。实际上，这种通用方法不难推广至有向图（习题[6-5]）。

6.4.2 复杂度

可见，邻接表所含列表数等于顶点总数n，每条边在其中仅存放一次（有向图）或两次（无向图），故空间总量为 $\Theta(n + e)$ ，与图自身的规模相当，较之邻接矩阵有很大改进。

当然，空间性能的这一改进，需以某些方面时间性能的降低为代价。比如，为判断顶点v到u的联边是否存在，`exists(v, u)`需在v对应的邻接表中顺序查找，共需 $\Theta(n)$ 时间。

与顶点相关操作接口，时间性能依然保持，甚至有所提高。比如，顶点的插入操作，可在 $\Theta(1)$ 而不是 $\Theta(n)$ 时间内完成。当然，顶点的删除操作，仍需遍历所有邻接表，共需 $\Theta(e)$ 时间。

尽管邻接表访问单条边的效率并不算高，却十分擅长于以批量方式，处理同一顶点的所有关联边。在以下图遍历等算法中，这是典型的处理流程和模式。比如，为枚举从顶点v发出的所有边，现在仅需 $\Theta(1 + \text{outDegree}(v))$ 而非 $\Theta(n)$ 时间。故总体而言，邻接表的效率较之邻接矩阵更高。因此，本章对以下各算法的复杂度分析，多以基于邻接表的实现方式为准。

§ 6.5 图遍历算法概述

图算法是个庞大的家族，其中大部分成员的主体框架，都可归结于图的遍历。与5.4节中树的遍历类似，图的遍历也需要访问所有顶点一次且仅一次；此外，图遍历同时还需要访问所有的边一次且仅一次——尽管对树而言这显而易见——并对边做分类，以便后续的处理。

实际上，无论采用何种策略和算法，图的遍历都可理解为，将非线性结构转化为半线性结构的过程。经遍历而确定的边类型中，最重要的一类即所谓的树边，它们与所有顶点共同构成了原图的一棵支撑树（森林），称作遍历树（*traversal tree*）。以遍历树为背景，其余各种类型的边，也能提供关于原图的重要信息，比如其中所含的环路等。

图中顶点之间可能存在多条通路，故为避免对顶点的重复访问，在遍历的过程中，通常还要动态地设置各顶点不同的状态，并随着遍历的进程不断地转换状态，直至最后的“访问完毕”。图的遍历更加强调对处于特定状态顶点的甄别与查找，故也称作图搜索（*graph search*）。

与树遍历一样，作为图算法基石的图搜索，本身也必须能够高效地实现。幸运的是，正如我们马上就会看到的，诸如深度优先、广度优先、最佳优先等基本而典型的图搜索，都可以在线性时间内完成。准确地，若顶点数和边数分别为 n 和 e ，则这些算法自身仅需 $O(n + e)$ 时间。既然图搜索需要访问所有的顶点和边，故这已经是我们所能期望的最优的结果。

§ 6.6 广度优先搜索

6.6.1 策略

各种图搜索之间的区别，体现为边分类结果的不同，以及所得遍历树（森林）的结构差异。其决定因素在于，搜索过程中的每一步迭代，将依照何种策略来选取下一接受访问的顶点。

通常，都是选取某个已访问到的顶点的邻居。同一顶点所有邻居之间的优先级，在多数遍历中不必讲究。因此，实质的差异应体现在，当有多个顶点已被访问到，应该优先从谁的邻居中选取下一顶点。比如，广度优先搜索（*breadth-first search, BFS*）采用的策略，可概括为：

越早被访问到的顶点，其邻居越优先被选用

于是，始自图中顶点 s 的BFS搜索，将首先访问顶点 s ；再依次访问 s 所有尚未访问到的邻居；再按后者被访问的先后次序，逐个访问它们的邻居；...；如此不断。在所有已访问到的顶点中，仍有邻居尚未访问者，构成所谓的波峰集（*frontier*）。于是，BFS搜索过程也可等效地理解为：

反复从波峰集中找到最早被访问到顶点 v ，若其邻居均已访问到，则将其逐出波峰集；否则，随意选出一个尚未访问到的邻居，并将其加入到波峰集中

不难发现，若将上述BFS策略应用于树结构，则效果等同于层次遍历（5.4.5节）——波峰集内顶点的深度始终相差不超过一，且波峰集总是优先在更浅的层次沿广度方向拓展。实际上，树层次遍历的这些特性，在一定程度上也适用于图的BFS搜索（习题[6-7]）。

由于每一步迭代都有一个顶点被访问，故至多迭代 $O(n)$ 步。另一方面，因为不会遗漏每个刚被访问顶点的任何邻居，故对于无向图必能覆盖 s 所属的连通分量（*connected component*），对于有向图必能覆盖以 s 为起点的可达分量（*reachable component*）。倘若还有来自其它连通分量或可达分量的顶点，则不妨从该顶点出发，重复上述过程。

6.6.2 实现

图的广度优先搜索算法，可实现如代码6.3所示。

```

1 template <typename Tv, typename Te> //广度优先搜索BFS算法(全图)
2 void Graph<Tv, Te>::bfs ( int s ) { //assert: 0 <= s < n
3     reset(); int clock = 0; int v = s; //初始化
4     do //逐一检查所有顶点
5         if ( UNDISCOVERED == status ( v ) ) //一旦遇到尚未发现的顶点
6             BFS ( v, clock ); //即从该顶点出发启动一次BFS
7     while ( s != ( v = ( ++v % n ) ) ); //按序号检查，故不漏不重
8 }
9
10 template <typename Tv, typename Te> //广度优先搜索BFS算法(单个连通域)
11 void Graph<Tv, Te>::BFS ( int v, int& clock ) { //assert: 0 <= v < n
12     Queue<int> Q; //引入辅助队列
13     status ( v ) = DISCOVERED; Q.enqueue ( v ); //初始化起点
14     while ( !Q.empty() ) { //在Q变空之前，不断
15         int v = Q.dequeue(); dTime ( v ) = ++clock; //取出队首顶点v
16         for ( int u = firstNbr ( v ); -1 < u; u = nextNbr ( v, u ) ) //枚举v的所有邻居u
17             if ( UNDISCOVERED == status ( u ) ) { //若u尚未被发现，则
18                 status ( u ) = DISCOVERED; Q.enqueue ( u ); //发现该顶点
19                 type ( v, u ) = TREE; parent ( u ) = v; //引入树边拓展支撑树
20             } else { //若u已被发现，或者甚至已访问完毕，则
21                 type ( v, u ) = CROSS; //将(v, u)归类于跨边
22             }
23         status ( v ) = VISITED; //至此，当前顶点访问完毕
24     }
25 }
```

代码6.3 BFS算法

算法的实质功能，由子算法BFS()完成。对该函数的反复调用，即可遍历所有连通或可达域。

仿照树的层次遍历，这里也借助队列Q，来保存已被发现，但尚未访问完毕的顶点。因此，任何顶点在进入该队列的同时，都被随即标记为DISCOVERED（已发现）状态。

BFS()的每一步迭代，都先从Q中取出当前的首顶点v；再逐一核对其各邻居u的状态并做相应处理；最后将顶点v置为VISITED（访问完毕）状态，即可进入下一步迭代。

若顶点u尚处于UNDISCOVERED（未发现）状态，则令其转为DISCOVERED状态，并随即加入队列Q。实际上，每次发现一个这样的顶点u，都意味着遍历树可从v到u拓展一条边。于是，将边(v, u)标记为树边(tree edge)，并按照遍历树中的承袭关系，将v记作u的父节点。

若顶点u已处于DISCOVERED状态（无向图），或者甚至处于VISITED状态（有向图），则意味着边(v, u)不属于遍历树，于是将该边归类为跨边(cross edge)（习题[6-11]）。

BFS()遍历结束后，所有访问过的顶点通过parent[]指针依次联接，从整体上给出了原图某一连通或可达域的一棵遍历树，称作广度优先搜索树，或简称BFS树(BFS tree)。

6.6.3 实例

图6.7给出了一个含8个顶点和11条边的有向图，起始于顶点S的BFS搜索过程。请留意观察辅助队列（下方）的演变，顶点状态的变化，边的分类与结果，以及BFS树的生长过程。

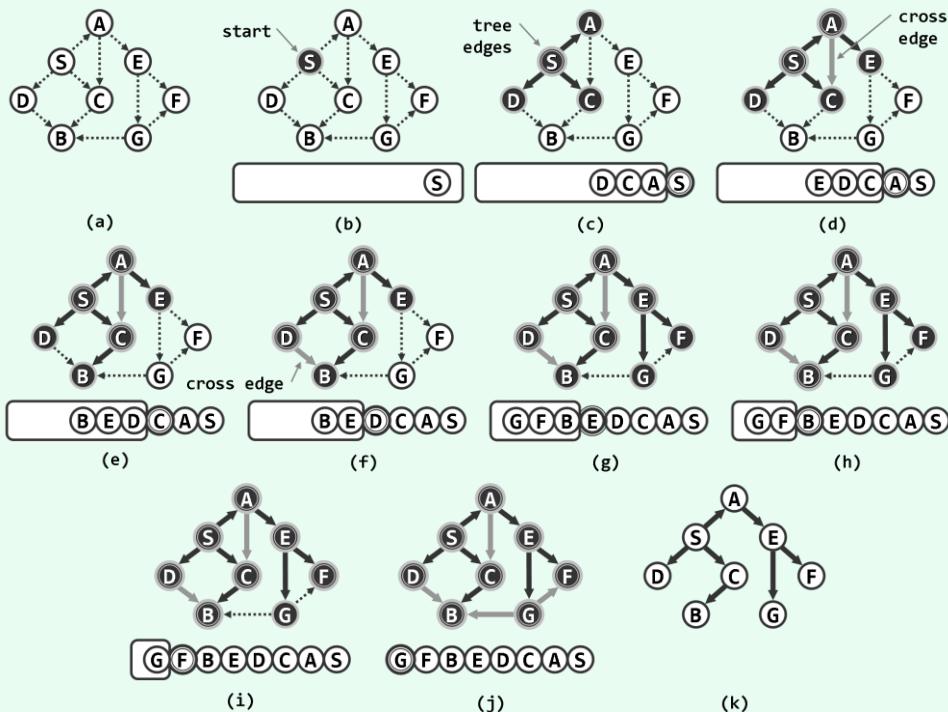


图6.7 广度优先搜索示例

不难看出， $\text{BFS}(s)$ 将覆盖起始顶点s所属的连通分量或可达分量，但无法抵达此外的顶点。而上层主函数 $\text{bfs}()$ 的作用，正在于处理多个连通分量或可达分量并存的情况。具体地，在逐个检查顶点的过程中，只要发现某一顶点尚未被发现，则意味着其所属的连通分量或可达分量尚未触及，故可从该顶点出发再次启动 $\text{BFS}()$ ，以遍历其所属的连通分量或可达分量。如此，各次 $\text{BFS}()$ 调用所得的BFS树构成一个森林，称作BFS森林（BFS forest）。

6.6.4 复杂度

除作为输入的图本身外，BFS搜索所使用的空间，主要消耗在用于维护顶点访问次序的辅助队列、用于记录顶点和边状态的标识位向量，累计 $O(n) + O(n) + O(e) = O(n + e)$ 。

时间方面，首先需花费 $O(n + e)$ 时间复位所有顶点和边的状态。不计对子函数 $\text{BFS}()$ 的调用， $\text{bfs}()$ 本身对所有顶点的枚举共需 $O(n)$ 时间。而在对 $\text{BFS}()$ 的所有调用中，每个顶点、每条边均只耗费 $O(1)$ 时间，累计 $O(n + e)$ 。综合起来，BFS搜索总体仅需 $O(n + e)$ 时间。

6.6.5 应用

基于BFS搜索，可有效地解决连通域分解（习题[6-6]）、最短路径（习题[6-8]）等问题。

§ 6.7 深度优先搜索

6.7.1 策略

深度优先搜索（Depth-First Search, DFS）选取下一项点的策略，可概括为：

优先选取最后一个被访问到的顶点的邻居

于是，以顶点 s 为基点的DFS搜索，将首先访问顶点 s ；再从 s 所有尚未访问到的邻居中任取其一，并以之为基点，递归地执行DFS搜索。故各顶点被访问到的次序，类似于树的先序遍历（5.4.2节）；而各顶点被访问完毕的次序，则类似于树的后序遍历（5.4.4节）。

6.7.2 实现

深度优先遍历算法可实现如代码6.4所示。

```

1 template <typename Tv, typename Te> //深度优先搜索DFS算法(全图)
2 void Graph<Tv, Te>::dfs ( int s ) { //assert: 0 <= s < n
3     reset(); int clock = 0; int v = s; //初始化
4     do //逐一检查所有顶点
5         if ( UNDISCOVERED == status ( v ) ) //一旦遇到尚未发现的顶点
6             DFS ( v, clock ); //即从该顶点出发启动一次DFS
7     while ( s != ( v = ( ++v % n ) ) ); //按序号检查，故不漏不重
8 }
9
10 template <typename Tv, typename Te> //深度优先搜索DFS算法(单个连通域)
11 void Graph<Tv, Te>::DFS ( int v, int& clock ) { //assert: 0 <= v < n
12     dTime ( v ) = ++clock; status ( v ) = DISCOVERED; //发现当前顶点v
13     for ( int u = firstNbr ( v ); -1 < u; u = nextNbr ( v, u ) ) //枚举v的所有邻居u
14         switch ( status ( u ) ) { //并视其状态分别处理
15             case UNDISCOVERED: //u尚未发现，意味着支撑树可在此拓展
16                 type ( v, u ) = TREE; parent ( u ) = v; DFS ( u, clock ); break;
17             case DISCOVERED: //u已被发现但尚未访问完毕，应属被后代指向的祖先
18                 type ( v, u ) = BACKWARD; break;
19             default: //u已访问完毕(VISITED, 有向图)，则视承袭关系分为前向边或跨边
20                 type ( v, u ) = ( dTime ( v ) < dTime ( u ) ) ? FORWARD : CROSS; break;
21         }
22     status ( v ) = VISITED; fTime ( v ) = ++clock; //至此，当前顶点v方告访问完毕
23 }
```

代码6.4 DFS算法

算法的实质功能，由子算法DFS()递归地完成。每一递归实例中，都先将当前节点 v 标记为 DISCOVERED（已发现）状态，再逐一核对其各邻居 u 的状态并做相应处理。待其所有邻居均已处理完毕之后，将顶点 v 置为VISITED（访问完毕）状态，便可回溯。

若顶点 u 尚处于UNDISCOVERED（未发现）状态，则将边 (v, u) 归类为树边（tree edge），

并将 v 记作 u 的父节点。此后，便可将 u 作为当前顶点，继续递归地遍历。

若顶点 u 处于DISCOVERED状态，则意味着在此处发现一个有向环路。此时，在DFS遍历树中 u 必为 v 的祖先（习题[6-13]），故应将边 (v, u) 归类为后向边（back edge）。

这里为每个顶点 v 都记录了被发现的和访问完成的时刻，对应的时间区间 $[dTime(v), fTime(v)]$ 均称作 v 的活跃期（active duration）。实际上，任意顶点 v 和 u 之间是否存在祖先/后代的“血缘”关系，完全取决于二者的活跃期是否相互包含（习题[6-12]）。

对于有向图，顶点 u 还可能处于VISITED状态。此时，只要比对 v 与 u 的活跃期，即可判定在DFS树中 v 是否为 u 的祖先。若是，则边 (v, u) 应归类为前向边（forward edge）；否则，二者必然来自相互独立的两个分支，边 (v, u) 应归类为跨边（cross edge）。

DFS(s)返回后，所有访问过的顶点通过parent[]指针依次联接，从整体上给出了顶点 s 所属连通或可达分量的一棵遍历树，称作深度优先搜索树或DFS树（DFS tree）。与BFS搜索一样，此时若还有其它的连通或可达分量，则可以其中任何顶点为基点，再次启动DFS搜索。

最终，经各次DFS搜索生成的一系列DFS树，构成了DFS森林（DFS forest）。

6.7.3 实例

图6.8针对含7个顶点和10条边的某有向图，给出了DFS搜索的详细过程。请留意观察顶点时间标签的设置，顶点状态的演变，边的分类和结果，以及DFS树（森林）的生长过程。

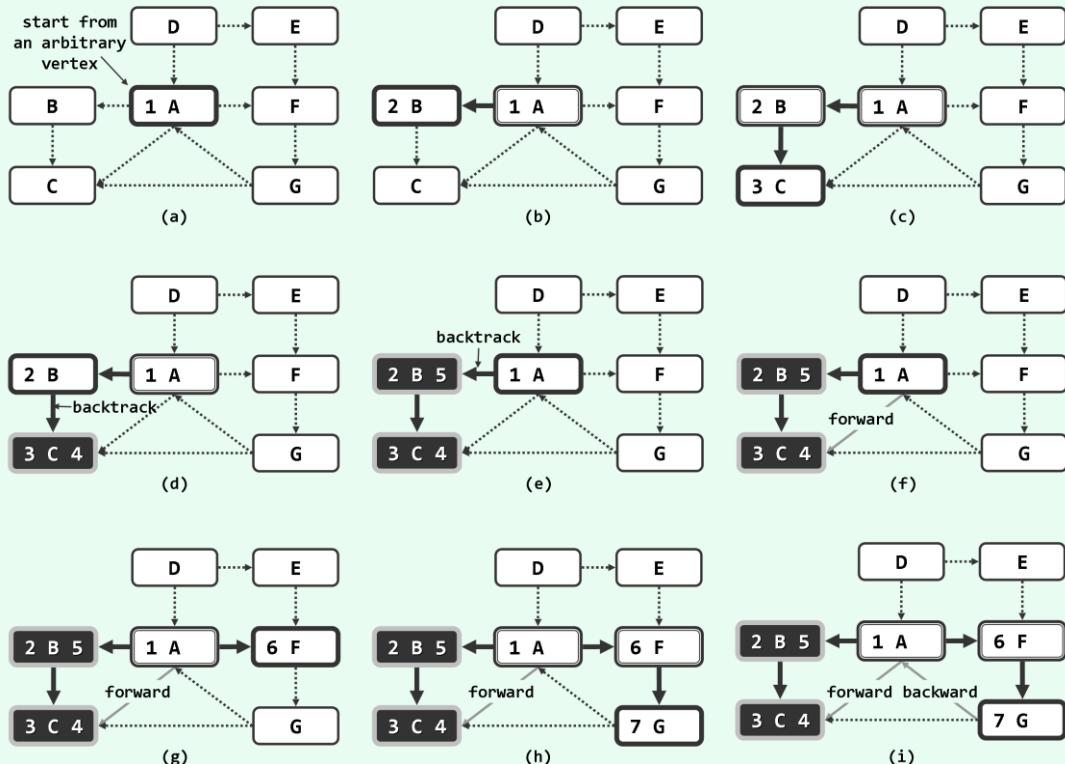


图6.8 深度优先搜索实例（粗边框白色，为当前顶点；细边框白色、双边框白色和黑色，分别为处于UNDISCOVERED、DISCOVERED和VISITED状态的顶点；dTime和fTime标签，分别标注于各顶点的左右）

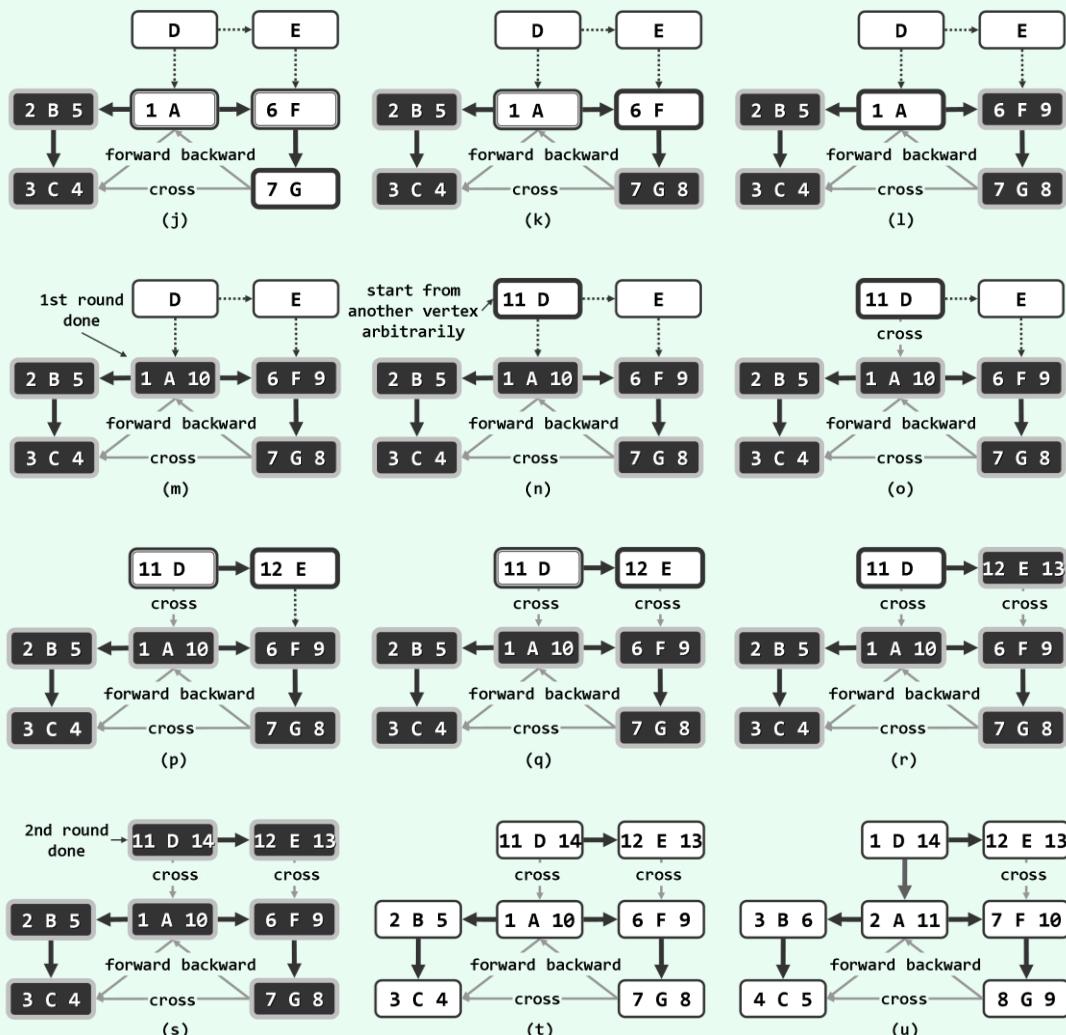


图6.8 深度优先搜索实例(续): (a~m)对应于DFS(A), (n~s)为随后的DFS(D)

最终结果如图(t)所示, 为包含两棵DFS树的一个DFS森林。可以看出, 选用不同的起始基点, 生成的DFS树(森林)也可能各异。如本例中, 若从D开始搜索, 则DFS森林可能如图(u)所示。

图6.9以时间为横坐标, 绘出了图6.8(u)中DFS树内各顶点的活跃期。可以清晰地看出, 活跃期相互包含的顶点, 在DFS树中都是“祖先-后代”关系(比如B之于C, 或者D之于F); 反之亦然。

这种对应关系并非偶然, 習此可以便捷地判定节点之间的承袭关系(习题[6-12])。故无论是对DFS搜索本身, 还是对基于DFS的各种算法而言, 时间标签都至关重要。

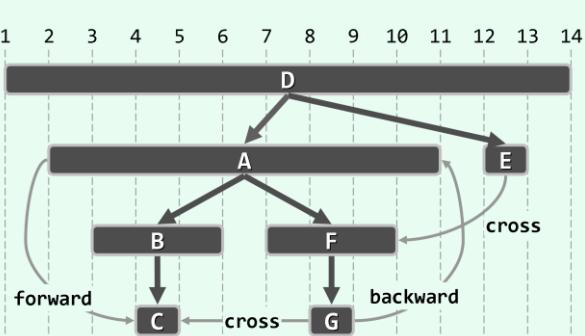


图6.9 活跃期与“祖先-后代”关系之间的对应关系

6.7.4 复杂度

除了原图本身，深度优先搜索算法所使用的空间，主要消耗于各顶点的时间标签和状态标记，以及各边的分类标记，二者累计不超过 $\mathcal{O}(n) + \mathcal{O}(e) = \mathcal{O}(n + e)$ 。当然，如采用以上代码6.4的直接递归实现方式，操作系统为维护运行栈还需耗费一定量的空间——尽管这部分增量在渐进意义下还不足以动摇以上结论。为此，不妨仿照5.4节的做法，通过显式地引入并维护一个栈结构，将DFS算法改写为迭代版本（习题[6-14]）。

时间方面，首先需要花费 $\mathcal{O}(n + e)$ 时间对所有顶点和边的状态复位。不计对子函数DFS()的调用，dfs()本身对所有顶点的枚举共需 $\mathcal{O}(n)$ 时间。不计DFS()之间相互的递归调用，每个顶点、每条边只在子函数DFS()的某一递归实例中耗费 $\mathcal{O}(1)$ 时间，故累计亦不过 $\mathcal{O}(n + e)$ 时间。综合而言，深度优先搜索算法也可在 $\mathcal{O}(n + e)$ 时间内完成。

6.7.5 应用

深度优先搜索无疑是最重要的图遍历算法。基于DFS的框架，可以导出和建立大量的图算法。以4.4节英雄忒修斯营救公主的故事为例，为寻找从迷宫入口（起始顶点）至公主所在位置（目标顶点）的通路，可将迷宫内不同位置之间的联接关系表示为一幅图，并将问题转化为起点和终点之间的可达性判定，从而可利用DFS算法便捷地加以解决。非但如此，一旦找到通路，则不仅可以顺利抵达终点与公主会合，还能沿这条通路安全返回。当然，与广度优先搜索一样，深度优先搜索也可用作连通分量的分解，或者有向无环图的判定。

下面仅以拓扑排序和双连通域分解为例，对DFS模式的应用做更为具体的介绍。

§ 6.8 拓扑排序

6.8.1 应用

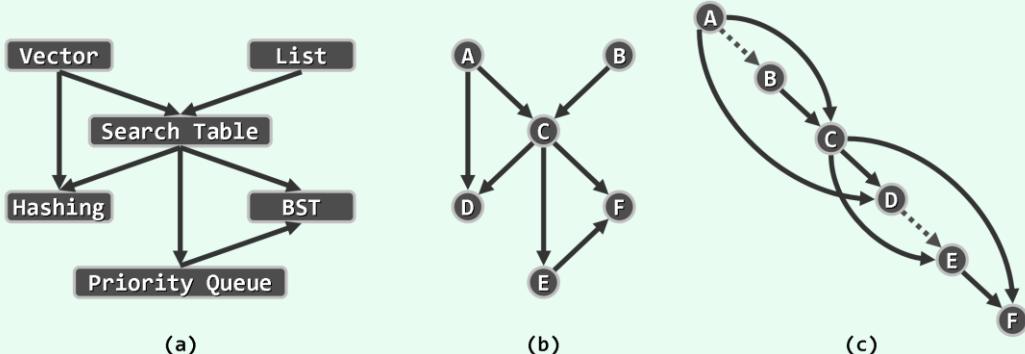


图6.10 拓扑排序

以教材的编写这一实际问题为例。首先，作者可借助有向图结构，整理出相关知识点之间的依赖关系。如图6.10(a)所示，因向量是散列表和查找表的基础知识点，故从Vector发出两条边分别指向Hashing和Search Table；同理，查找表是二叉搜索树的基础知识点，故也从前者引出一条边指向后者；...；诸如此类。那么，如何将这些知识点串联为一份教学计划，以保证在整个授课进程中，每堂课的基础知识点均在此前业已讲授呢？

若将图6.10(a)抽象为图(b), 则不难看出, 图(c)就是一份可行的教材目录和授课计划。实际上, 许多应用问题, 都可转化为和描述为这一标准形式: 给定描述某一实际应用(图(a))的有向图(图(b)), 如何在与该图“相容”的前提下, 将所有顶点排成一个线性序列(图(c))。

此处的“相容”, 准确的含义是: 每一顶点都不会通过边, 指向其在此序列中的前驱顶点。这样的一个线性序列, 称作原有向图的一个拓扑排序(**topological sorting**)。

6.8.2 有向无环图

那么, 拓扑排序是否必然存在? 若存在, 又是否唯一? 这两个问题都不难回答。

在图6.10(c)中, 顶点A和B互换之后依然是一个拓扑排序, 故知同一有向图的拓扑排序未必唯一。又若在图(b)中引入一条从顶点F指向B的边, 使顶点B、C和F构成一个有向环路, 则无论如何也不可能得到一个“相容”的线性序列, 故拓扑排序也未必存在。

反之, 不含环路的有向图——有向无环图——一定存在拓扑排序吗? 答案是肯定的。

有向无环图的拓扑排序必然存在; 反之亦然。这是因为, 有向无环图对应于偏序关系, 而拓扑排序则对应于全序关系。在顶点数目有限时, 与任一偏序相容的全序必然存在。

实际上, 在任一有限偏序集中, 必有极值元素(尽管未必唯一); 相应地, 任一有向无环图, 也必包含入度为零的顶点。否则, 每个顶点都至少有一条入边, 这意味着图中包含环路。

于是, 只要将入度为0的顶点m(及其关联边)从图G中取出, 则剩余的G'依然是有向无环图, 故其拓扑排序也必然存在。从递归的角度看, 一旦得到了G'的拓扑排序, 只需将m作为最大顶点插入, 即可得到G的拓扑排序。如此, 我们已经得到了一个拓扑排序的算法(习题[6-18])。

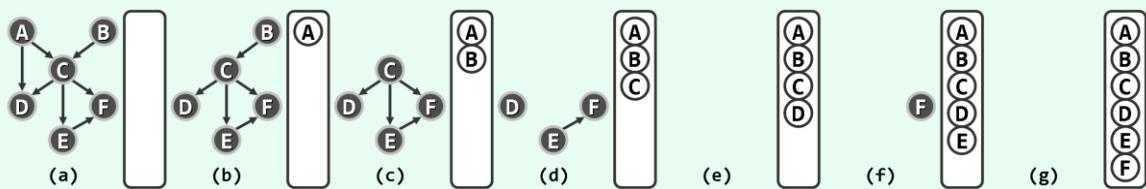


图6.11 利用“DAG必有零入度顶点”的特性, 实现拓扑排序

图6.11给出了该算法的一个实例。以下, 将转而从BFS搜索入手, 给出另一拓扑排序算法。

6.8.3 算法

不妨将关注点, 转至与极大顶点相对称的极小顶点。

同理, 有限偏序集中也必然存在极小元素(同样, 未必唯一)。该元素作为顶点, 出度必然为零——比如图6.10(b)中的顶点D和F。而在对有向无环图的DFS搜索中, 首先因访问完成而转换至VISITED状态的顶点m, 也必然具有这一性质; 反之亦然。

进一步地, 根据DFS搜索的特性, 顶点m(及其关联边)对此后的搜索过程将不起任何作用。于是, 下一转换至VISITED状态的顶点可等效地理解为是, 从图中剔除顶点m(及其关联边)之后的出度为零者——在拓扑排序中, 该顶点应为顶点m的前驱。由此可见, DFS搜索过程中各顶点被标记为VISITED的次序, 恰好(按逆序)给出了原图的一个拓扑排序。

此外, DFS搜索善于检测环路的特性, 恰好可以用来判别输入是否为有向无环图。具体地, 搜索过程中一旦发现后向边, 即可终止算法并报告“因非DAG而无法拓扑排序”。

6.8.4 实现

基于DFS搜索框架的拓扑排序算法，可实现如代码6.5所示。

```

1 template <typename Tv, typename Te> //基于DFS的拓扑排序算法
2 Stack<Tv>* Graph<Tv, Te>::tSort ( int s ) { //assert: 0 <= s < n
3     reset(); int clock = 0; int v = s;
4     Stack<Tv>* S = new Stack<Tv>; //用栈记录排序顶点
5     do {
6         if ( UNDISCOVERED == status ( v ) )
7             if ( !TSort ( v, clock, S ) ) { //clock并非必需
8                 while ( !S->empty() ) //任一连通域（亦即整图）非DAG
9                     S->pop(); break; //则不必继续计算，故直接返回
10            }
11    } while ( s != ( v = ( ++v % n ) ) );
12    return S; //若输入为DAG，则S内各顶点自顶向底排序；否则（不存在拓扑排序），S空
13 }
14
15 template <typename Tv, typename Te> //基于DFS的拓扑排序算法（单趟）
16 bool Graph<Tv, Te>::TSort ( int v, int& clock, Stack<Tv>* S ) { //assert: 0 <= v < n
17     dTime ( v ) = ++clock; status ( v ) = DISCOVERED; //发现顶点v
18     for ( int u = firstNbr ( v ); -1 < u; u = nextNbr ( v, u ) ) //枚举v的所有邻居u
19         switch ( status ( u ) ) { //并视u的状态分别处理
20             case UNDISCOVERED:
21                 parent ( u ) = v; type ( v, u ) = TREE;
22                 if ( !TSort ( u, clock, S ) ) //从顶点u处出发深入搜索
23                     return false; //若u及其后代不能拓扑排序（则全图亦必如此），故返回并报告
24                 break;
25             case DISCOVERED:
26                 type ( v, u ) = BACKWARD; //一旦发现后向边（非DAG），则
27                 return false; //不必深入，故返回并报告
28             default: //VISITED (digraphs only)
29                 type ( v, u ) = ( dTime ( v ) < dTime ( u ) ) ? FORWARD : CROSS;
30                 break;
31         }
32     status ( v ) = VISITED; S->push ( vertex ( v ) ); //顶点被标记为VISITED时，随即入栈
33     return true; //u及其后代可以拓扑排序
34 }
```

代码6.5 基于DFS搜索框架实现拓扑排序算法

相对于标准的DFS搜索算法，这里增设了一个栈结构。一旦某个顶点被标记为VISITED状态，便随即令其入栈。如此，当搜索终止时，所有顶点即按照被访问完毕的次序——亦即拓扑排序的次序——在栈中自顶而下排列。

6.8.5 实例

图6.12以含6个顶点和7条边的有向无环图为例，给出了以上算法的执行过程。共分三步迭代，分别对应于起始于顶点C、B和A的三趟DFS搜索。请留意观察，各顶点的入栈次序。

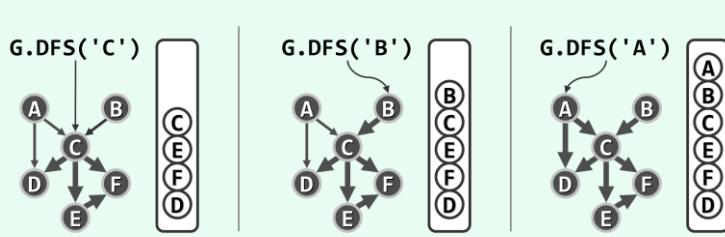


图6.12 基于DFS搜索的拓扑排序实例

另外，对照图6.11中的结果可见，因多个极大、极小元素（入度、出度为零顶点）并存而导致拓扑排序的不唯一性并未消除，而是转由该算法对每趟DFS起点的选择策略决定。

6.8.6 复杂度

这里仅额外引入的栈，规模不超过顶点总数 $O(n)$ 。总体而言，空间复杂度与基本的深度优先搜索算法同样，仍为 $O(n + e)$ 。该算法的递归跟踪过程与标准DFS搜索完全一致，且各递归实例自身的执行时间依然保持为 $O(1)$ ，故总体运行时间仍为 $O(n + e)$ 。

为与基本的DFS搜索算法做对比，代码6.5保留了代码6.4的通用框架，但并非所有操作都与拓扑排序直接相关。因此通过精简代码，还可进一步地优化（习题[6-19]）。

§ 6.9 *双连通域分解

6.9.1 关节点与双连通域

考查无向图G。若删除顶点v后G所包含的连通域增多，则v称作切割节点（cut vertex）或关节点（articulation point）。如图6.13中的C即是一个关节点——它的删除将导致连通域增加两块。反之，不含任何关节点的图称作双连通图。任一无向图都可视作由若干个极大的双连通子图组合而成，这样的每一子图都称作原图的一个双连通域（bi-connected component）。例如图6.14(a)中的无向图，可分解为如图(b)所示的三个双连通域。

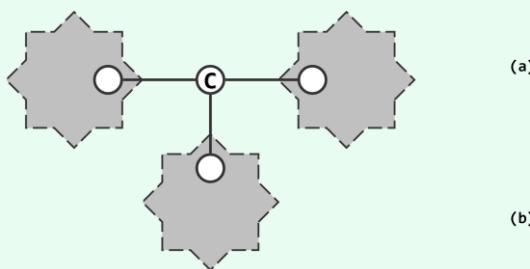


图6.13 关节点

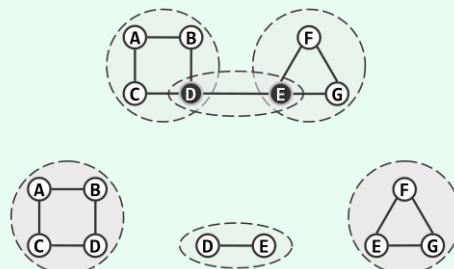


图6.14 双连通域

较之其它顶点，关节点更为重要。在网络系统中它们对应于网关，决定子网之间能否连通。在航空系统中，某些机场的损坏，将同时切断其它机场之间的交通。故在资源总量有限的前提下，找出关节点并重点予以保障，是提高系统整体稳定性和鲁棒性的基本策略。

6.9.2 蛮力算法

那么，如何才能找出图中的关节点呢？

由其定义，可直接导出蛮力算法大致如下：首先，通过BFS或DFS搜索统计出图G所含连通域的数目；然后逐一枚举每个顶点v，暂时将其从图G中删去，并再次通过搜索统计出图G\{v}所含连通域的数目。于是，顶点v是关节点，当且仅当图G\{v}包含的连通域多于图G。

这一算法需执行n趟搜索，耗时 $O(n(n + e))$ ，如此低的效率无法令人满意。

以下将介绍基于DFS搜索的另一算法，它不仅效率更高，而且可同时对原图做双连通域分解。

6.9.3 可行算法

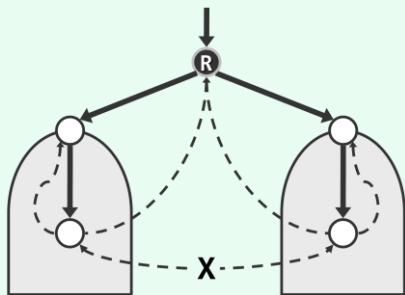


图6.15 DFS树根节点是关节点，当且仅当它拥有多个分支

经DFS搜索生成的DFS树，表面上看似乎“丢失”了原图的一些信息，但实际上就某种意义而言，依然可以提供足够多的信息。

比如，DFS树中的叶节点，绝不可能是原图中的关节点——此类顶点的删除既不致影响DFS树的连通性，也不致影响原图的连通性。此外，DFS树的根节点若至少拥有两个分支，则必是一个关节点。如图6.15所示，在原无向图中，根节点R的不同分支之间不可能通过跨边相联，R是它们之间唯一的枢纽。反之，若根节点仅有一个分支，则与叶节点同理，它也不可能成为关节点。

那么，又该如何甄别一般的内部节点是否为关节点呢？

考查图6.16中的内部节点C。若节点C的移除导致其某一棵（比如以D为根的）真子树与其真祖先（比如A）之间无法连通，则C必为关节点。反之，若C的所有真子树都能（如以E为根的子树那样）与C的某一真祖先连通，则C就不可能是关节点。

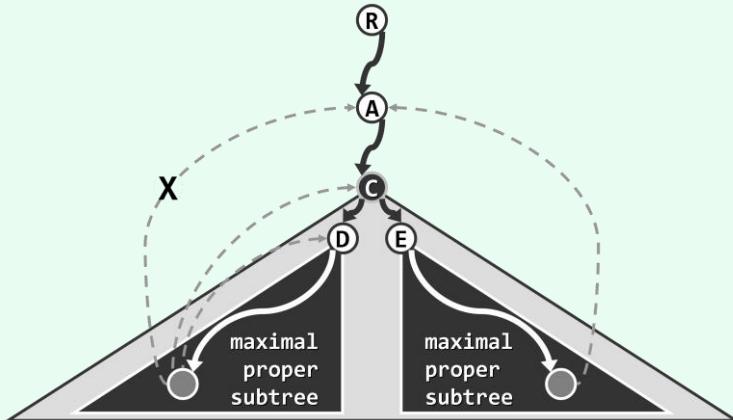


图6.16 内部节点C是关节点，当且仅当C的某棵极大真子树不（经后向边）联接到C的真祖先

当然，在原无向图的DFS树中，C的真子树只可能通过后向边与C的真祖先连通。因此，只要在DFS搜索过程记录并更新各顶点v所能（经由后向边）连通的最高祖先（highest connected ancestor, HCA）hca[v]，即可及时认定关节点，并报告对应的双连通域。

6.9.4 实现

根据以上分析，基于DFS搜索框架的双连通域分解算法，可实现如代码6.6所示。

```

1 template <typename Tv, typename Te> void Graph<Tv, Te>::bcc ( int s ) { //基于DFS的BCC分解算法
2     reset(); int clock = 0; int v = s; Stack<int> S; //栈S用以记录已访问的顶点
3     do
4         if ( UNDISCOVERED == status ( v ) ) { //一旦发现未发现的顶点（新连通分量）
5             BCC ( v, clock, S ); //即从该顶点出发启动一次BCC
6             S.pop(); //遍历返回后，弹出栈中最后一个顶点——当前连通域的起点
7         }
8     while ( s != ( v = ( ++v % n ) ) );
9 }

10 #define hca(x) (fTime(x)) //利用此处闲置的fTime[]充当hca[]
11 template <typename Tv, typename Te> //顶点类型、边类型
12 void Graph<Tv, Te>::BCC ( int v, int& clock, Stack<int>& S ) { //assert: 0 <= v < n
13     hca ( v ) = dTime ( v ) = ++clock; status ( v ) = DISCOVERED; S.push ( v ); //v被发现并入栈
14     for ( int u = firstNbr ( v ); -1 < u; u = nextNbr ( v, u ) ) //枚举v的所有邻居u
15         switch ( status ( u ) ) { //并视u的状态分别处理
16             case UNDISCOVERED:
17                 parent ( u ) = v; type ( v, u ) = TREE; BCC ( u, clock, S ); //从顶点u处深入
18                 if ( hca ( u ) < dTime ( v ) ) //遍历返回后，若发现u（通过后向边）可指向v的真祖先
19                     hca ( v ) = min ( hca ( v ), hca ( u ) ); //则v亦必如此
20                 else { //否则，以v为关节点（u以下即是一个BCC，且其中顶点此时正集中于栈S的顶部）
21                     while ( v != S.pop() ); //依次弹出当前BCC中的节点，亦可根据实际需求转存至其它结构
22                     S.push ( v ); //最后一个顶点（关节点）重新入栈——总计至多两次
23                 }
24                 break;
25             case DISCOVERED:
26                 type ( v, u ) = BACKWARD; //标记(v, u)，并按照“越小越高”的准则
27                 if ( u != parent ( v ) ) hca ( v ) = min ( hca ( v ), dTime ( u ) ); //更新hca[v]
28                 break;
29             default: //VISITED (digraphs only)
30                 type ( v, u ) = ( dTime ( v ) < dTime ( u ) ) ? FORWARD : CROSS;
31                 break;
32             }
33         status ( v ) = VISITED; //对v的访问结束
34     }
35 #undef hca

```

代码6.6 基于DFS搜索框架实现双连通域分解算法

由于处理的是无向图，故DFS搜索在顶点v的孩子u处返回之后，通过比较hca[u]与dTime[v]的大小，即可判断v是否关节点。

这里将闲置的fTime[]用作hca[]。故若hca[u] ≥ dTime[v]，则说明u及其后代无法通过

后向边与 v 的真祖先连通，故 v 为关节点。既然栈 S 存有搜索过的顶点，与该关节点相对应的双连通域内的顶点，此时都应集中存放于 S 顶部，故可依次弹出这些顶点。 v 本身必然最后弹出，作为多个连通域的联接枢纽，它应重新进栈。

反之若 $hca[u] < dTime[v]$ ，则意味着 u 可经由后向边连通至 v 的真祖先。果真如此，则这一性质对 v 同样适用，故有必要将 $hca[v]$ ，更新为 $hca[v]$ 与 $hca[u]$ 之间的更小者。

当然，每遇到一条后向边(v, u)，也需要及时地将 $hca[v]$ ，更新为 $hca[v]$ 与 $dTime[u]$ 之间的更小者，以保证 $hca[v]$ 能够始终记录顶点 v 可经由后向边向上连通的最高祖先。

同样地，为便于与基本的DFS搜索算法相对比，代码6.6也保留了代码6.4的通用框架。因此，通过清理与双连通域分解无关的操作并精简代码，也可降低时间复杂度的常系数（习题[6-20]）。

6.9.5 实例

图6.17以一个包含10个顶点和12条边的无向图为例，详细给出了以上算法的完整计算过程。

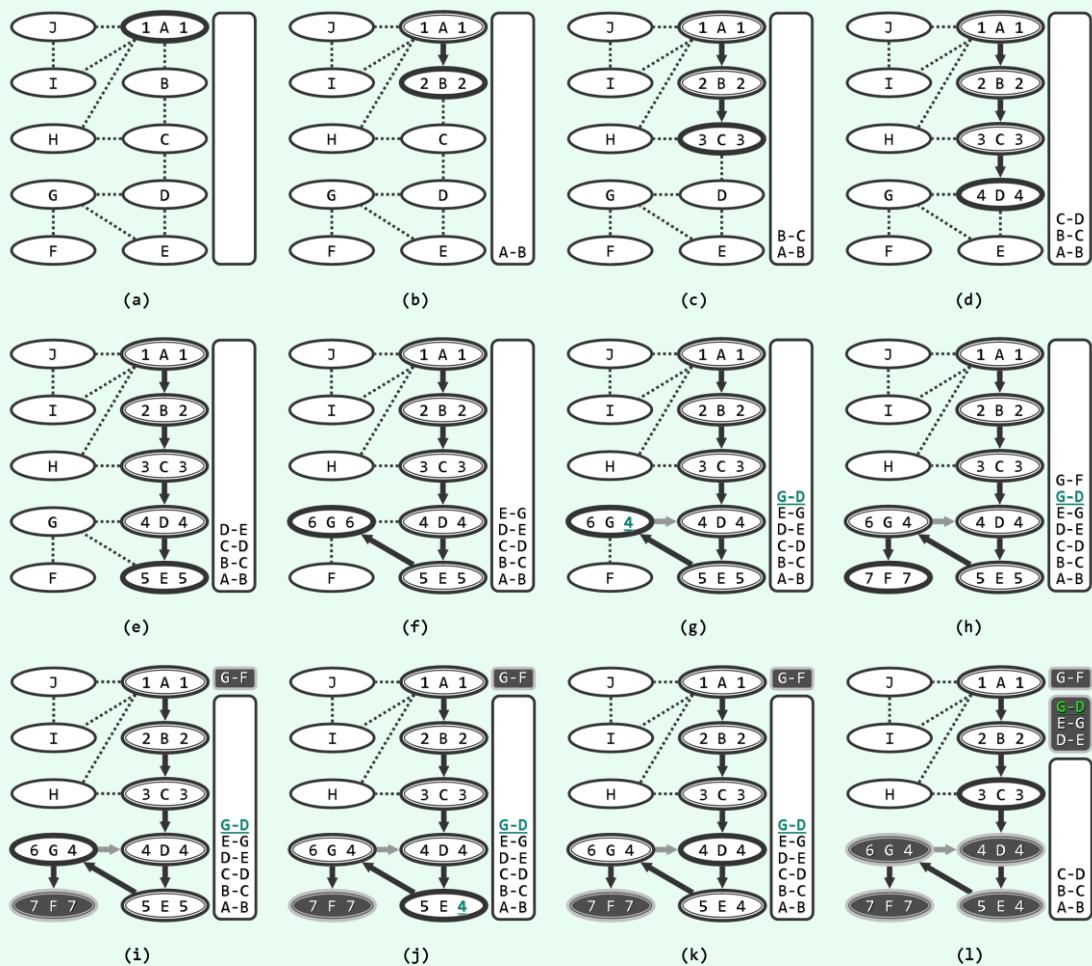


图6.17 基于DFS搜索的双连通域分解实例

（细边框白色、双边框白色和黑色分别示意处于UNDISCOVERED、DISCOVERED和VISITED状态的顶点，粗边框白色示意当前顶点； $dTime$ 和 $fTime(hca)$ 标签分别标注于各顶点的左、右）

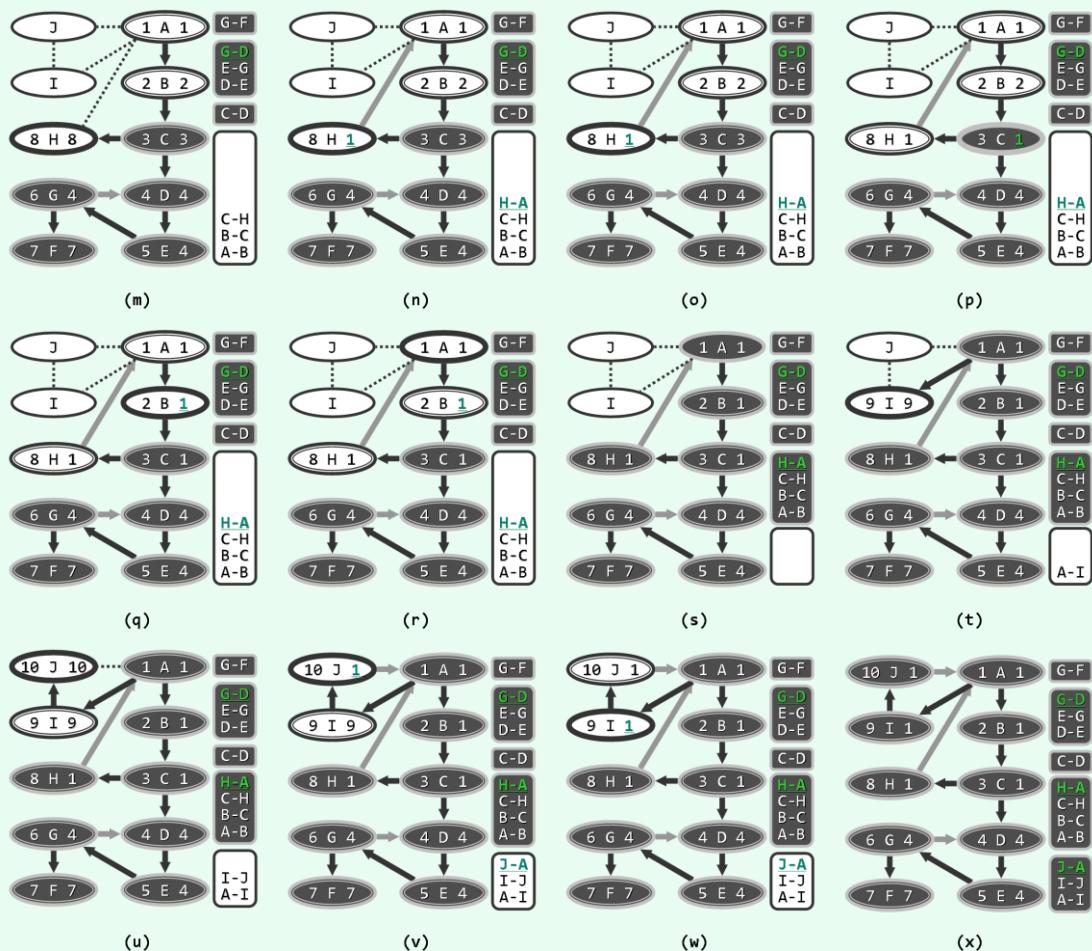


图6.17 基于DFS搜索的双连通域分解实例（续）

6.9.6 复杂度

与基本的DFS搜索算法相比，这里只增加了一个规模 $\mathcal{O}(n)$ 的辅助栈，故整体空间复杂度仍为 $\mathcal{O}(n + e)$ 。时间方面，尽管同一顶点 v 可能多次入栈，但每一次重复入栈都对应于某一新发现的双连通域，与之对应地必有至少另一顶点出栈且不再入栈。因此，这类重复入栈操作不会超过 n 次，入栈操作累计不超过 $2n$ 次，故算法的整体运行时间依然是 $\mathcal{O}(n + e)$ 。

§ 6.10 优先级搜索

6.10.1 优先级与优先级数

以上图搜索应用虽各具特点，但其基本框架却颇为相似。总体而言，都需通过迭代逐一发现各顶点，将其纳入遍历树中并做相应处理，同时根据应用问题的需求，适时给出解答。各算法在功能上的差异，主要体现为每一步迭代中对新顶点的选取策略不同。比如，BFS搜索会优先考查更早被发现的顶点，而DFS搜索则恰好相反，会优先考查最后被发现的顶点。

每一种选取策略都等效于，给所有顶点赋予不同的优先级，而且随着算法的推进不断调整；而每一步迭代所选取的顶点，都是当时的优先级最高者。按照这种理解，包括BFS和DFS在内的几乎所有图搜索，都可纳入统一的框架。鉴于优先级在其中所扮演的关键角色，故亦称作优先级搜索（priority-first search, PFS），或最佳优先搜索（best-first search, BFS）。

为落实以上理解与构思，图ADT（表6.2和代码6.1）提供了priority()接口，以支持对顶点优先级数（priority number）的读取和修改。在实际应用中，引导优化方向的指标，往往对应于某种有限的资源或成本（如光纤长度、通讯带宽、机票价格等），故这里不妨约定优先级数越大（小）顶点的优先级越低（高）。相应地，在算法的初始化阶段（如代码6.1中的reset()），通常都将顶点的优先级数统一置为最大（比如对于int类型，可采用INT_MAX）——或等价地，优先级最低。

6.10.2 基本框架

按照上述思路，优先级搜索算法的框架可具体实现如代码6.7所示。

```

1 template <typename Tv, typename Te> template <typename PU> //优先级搜索(全图)
2 void Graph<Tv, Te>::pfs ( int s, PU prioUpdater ) { //assert: 0 <= s < n
3     reset(); int v = s; //初始化
4     do //逐一检查所有顶点
5         if ( UNDISCOVERED == status ( v ) ) //一旦遇到尚未发现的顶点
6             PFS ( v, prioUpdater ); //即从该顶点出发启动一次PFS
7     while ( s != ( v = ( ++v % n ) ) ); //按序号检查，故不漏不重
8 }
9
10 template <typename Tv, typename Te> template <typename PU> //顶点类型、边类型、优先级更新器
11 void Graph<Tv, Te>::PFS ( int s, PU prioUpdater ) { //优先级搜索(单个连通域)
12     priority ( s ) = 0; status ( s ) = VISITED; parent ( s ) = -1; //初始化，起点s加至PFS树中
13     while ( 1 ) { //将下一顶点和边加至PFS树中
14         for ( int w = firstNbr ( s ); -1 < w; w = nextNbr ( s, w ) ) //枚举s的所有邻居w
15             prioUpdater ( this, s, w ); //更新顶点w的优先级及其父顶点
16         for ( int shortest = INT_MAX, w = 0; w < n; w++ )
17             if ( UNDISCOVERED == status ( w ) ) //从尚未加入遍历树的顶点中
18                 if ( shortest > priority ( w ) ) //选出下一个
19                     { shortest = priority ( w ); s = w; } //优先级最高的顶点s
20         if ( VISITED == status ( s ) ) break; //直至所有顶点均已加入
21         status ( s ) = VISITED; type ( parent ( s ), s ) = TREE; //将s及与其父的联边加入遍历树
22     }
23 } //通过定义具体的优先级更新策略prioUpdater，即可实现不同的算法功能

```

代码6.7 优先级搜索算法框架

可见，PFS搜索的基本过程和功能与常规的图搜索算法一样，也是以迭代方式逐步引入顶点和边，最终构造出一棵遍历树（或者遍历森林）。如上所述，每次都是引入当前优先级最高（优先级数最小）的顶点s，然后按照不同的策略更新其邻接顶点的优先级数。

这里借助函数对象**prioUpdater**, 使算法设计者得以根据不同的问题需求, 简明地描述和实现对应的更新策略。具体地, 只需重新定义**prioUpdater**对象即可, 而不必重复实现公共部分。比如, 此前的BFS搜索和DFS搜索都可按照此模式统一实现(习题[6-21])。

下面, 以最小支撑树和最短路径这两个经典的图算法为例, 深入介绍这一框架的具体应用。

6.10.3 复杂度

PFS搜索由两重循环构成, 其中内层循环又含并列的两个循环。若采用邻接表实现方式, 同时假定**prioUpdater()**只需常数时间, 则前一内循环的累计时间应取决于所有顶点的出度总和, 即 $\mathcal{O}(e)$; 后一内循环固定迭代 n 次, 累计 $\mathcal{O}(n^2)$ 时间。两项合计总体复杂度为 $\mathcal{O}(n^2)$ 。

实际上, 借助稍后第10章将要介绍的优先级队列等结构, PFS搜索的效率还有进一步提高的余地(习题[10-16]和[10-17])。

§ 6.11 最小支撑树

6.11.1 支撑树

如图6.18所示, 连通图G的某一无环连通子图T若覆盖G中所有的顶点, 则称作G的一棵支撑树或生成树(spanning tree)。

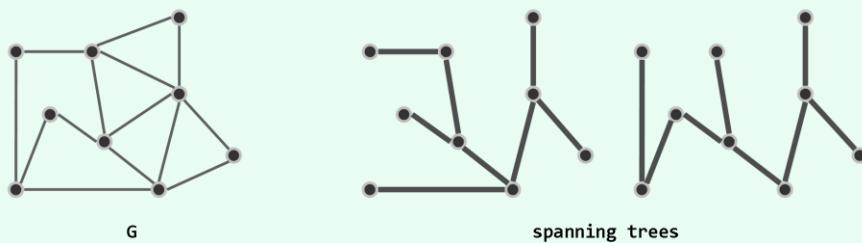


图6.18 支撑树

就保留原图中边的数目而言, 支撑树既是“禁止环路”前提下的极大子图, 也是“保持连通”前提下的最小子图。在实际应用中, 原图往往对应于由一组可能相互联接(边)的成员(顶点)构成的系统, 而支撑树则对应于该系统最经济的联接方案。确切地, 尽管同一幅图可能有多棵支撑树, 但由于其中的顶点总数均为n, 故其采用的边数也均为n - 1。

6.11.2 最小支撑树

若图G为一带权网络, 则每一棵支撑树的成本(cost)即为其所采用各边权重的总和。在G的所有支撑树中, 成本最低者称作最小支撑树(minimum spanning tree, MST)。

聚类分析、网络架构设计、VLSI布线设计等诸多实际应用问题, 都可转化并描述为最小支撑树的构造问题。在这些应用中, 边的权重大多对应于某种可量化的成本, 因此作为对应优化问题的基本模型, 最小支撑树的价值不言而喻。另外, 最小支撑树构造算法也可为一些NP问题提供足够快速、足够接近的近似解法(习题[6-22])。正因为受到来自众多应用和理论领域的需求推动, 最小支撑树的构造算法也发展得较为成熟。

6.11.3 歧义性

尽管同一带权网络通常都有多棵支撑树，但总数毕竟有限，故必有最低的总体成本。然而，总体成本最低的支撑树却未必唯一。以包含三个顶点的完全图为例，若三条边的权重相等，则其中任意两条边都构成总体成本最低的一棵支撑树。

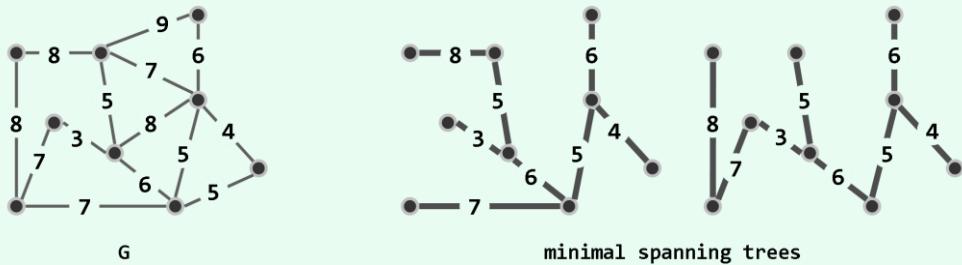


图6.19 极小支撑树与最小支撑树

更一般的例子如图6.19所示，对应于左侧的带权网络，有两棵支撑树的总体成本均达到最低（44）。故严格说来，此类支撑树应称作极小支撑树（minimal spanning tree）。当然，通过强制附加某种次序即可消除这种歧义性（习题[6-23]），故不妨仍称之为最小支撑树。

6.11.4 蛮力算法

由最小支撑树的定义，可直接导出蛮力算法大致如下：逐一考查 G 的所有支撑树并统计其成本，从而挑选出其中的最低者。然而根据Cayley公式，由 n 个互异顶点组成的完全图共有 n^{n-2} 棵支撑树，即便忽略掉构造所有支撑树所需的成本，仅为更新最低成本的记录就需要 $\mathcal{O}(n^{n-2})$ 时间。

事实上基于PFS搜索框架，并采用适当的顶点优先级更新策略，即可得出如下高效的最小支撑树构造算法。

6.11.5 Prim算法

为更好地理解这一算法的原理，以下先从最小支撑树的性质入手。为简化起见，不妨假定各边的权重互异。实际上，为将最小支撑树的以下性质及其构造算法的正确性等结论推广到允许多边等权的退化情况，还需补充更为严格的分析与证明（习题[6-25]、[6-26]和[6-27]）。

■ 割与极短跨越边

图 $G = (V; E)$ 中，顶点集 V 的任一非平凡子集 U 及其补集 $V \setminus U$ 都构成 G 的一个割（cut），记作 $(U : V \setminus U)$ 。若边 uv 满足 $u \in U$ 且 $v \notin U$ ，则称作该割的一条跨越边（crossing edge）。因此类边联接于 V 及其补集之间，故亦形象地称作该割的一座桥（bridge）。

Prim算法^②的正确性基于以下事实：最小支撑树总是会采用联接每一割的最短跨越边。

否则，如图6.20(a)所示假设 uv 是割 $(U : V \setminus U)$ 的最短跨越边，而最小支撑树 T 并未采用该边。于是由树的连通性，如图(b)所示在 T 中必有至少另一跨边 st 联接该割（有可能 $s = u$ 或 $t = v$ ）。

^② 由R. C. Prim于1956年发明

v , 尽管二者不能同时成立)。同样由树的连通性, T 中必有分别联接于 u 和 s 、 v 和 t 之间的两条通路。由于树是极大的无环图, 故倘若将边 uv 加至 T 中, 则如图(c)所示, 必然出现穿过 u 、 v 、 t 和 s 的唯一环路。接下来, 只要再删除边 st , 则该环路必然随之消失。

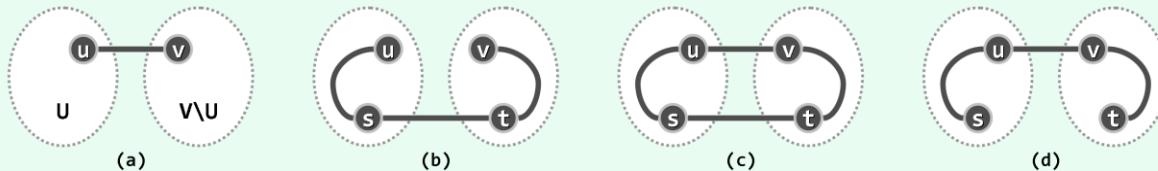


图6.20 最小支撑树总是会采用联接每一割的最短跨越边

经过如此的一出一入, 若设 T 转换为 T' , 则 T' 依然是连通图, 且所含边数与 T 相同均为 $n - 1$ 。这就意味着, T' 也是原图的一棵支撑树。就结构而言, T' 与 T 的差异仅在于边 uv 和边 st , 故二者的成本之差就是这两条边的权重之差。不难看出, 边 st 的权重必然大于身为最短跨越边的 uv , 故 T' 的总成本低于 T ——这与 T 总体权重最小的前提矛盾。

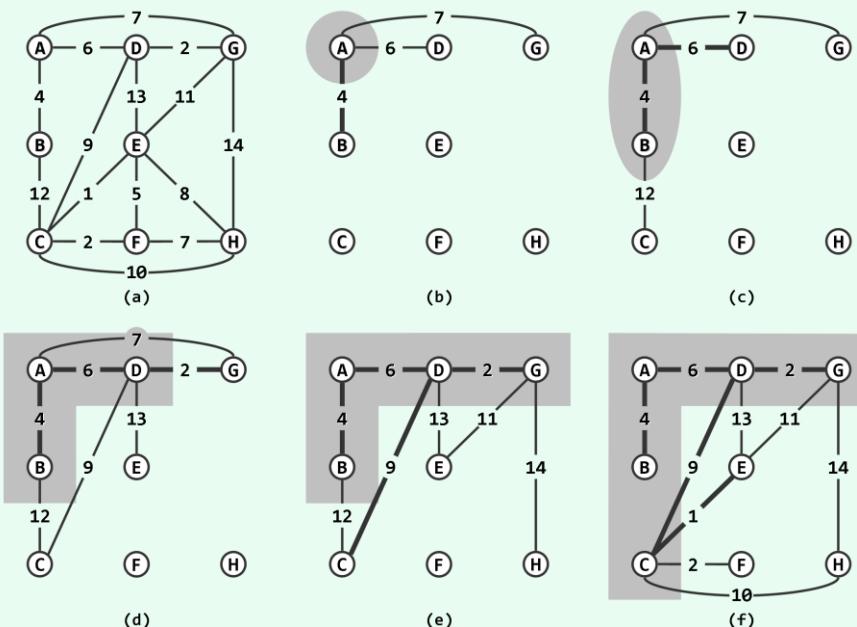
注意, 以上性质并不意味着同一割仅能为最小支撑树贡献一条跨越边(习题[6-17])。

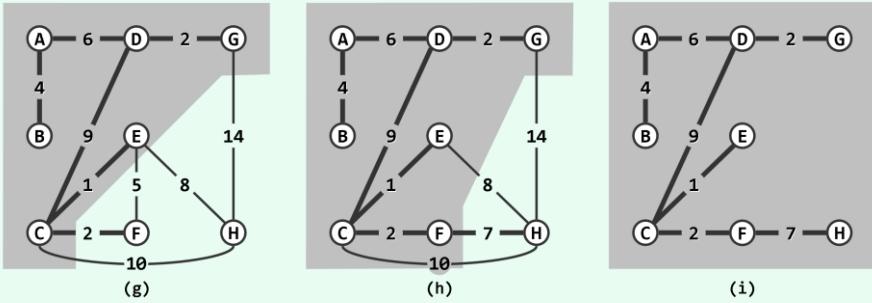
■ 贪心迭代

由以上性质, 可基于贪心策略导出一个迭代式算法。每一步迭代之前, 假设已经得到最小支撑树 T 的一棵子树 $T_k = (V_k; E_k)$, 其中 V_k 包含 k 个顶点, E_k 包含 $k - 1$ 条边。于是, 若将 V_k 及其补集视作原图的一个割, 则在找到该割的最短跨越边 $e_k = (v_k, u_k)$ ($v_k \in V_k$ 且 $u_k \notin V_k$) 之后, 即可将 T_k 扩展为一棵更大的子树 $T_{k+1} = (V_{k+1}; E_{k+1})$, 其中 $V_{k+1} = V_k \cup \{u_k\}$, $E_{k+1} = E_k \cup \{e_k\}$ 。最初的 T_1 不含边而仅含单个顶点, 故可从原图的顶点中任意选取。

■ 实例

图6.21以一个含8个顶点和15条边的无向图 G (图(a))为例, 给出了Prim算法的执行过程。



图6.21 Prim算法示例（阴影区域示意不断扩展的子树 T_k ，粗线示意树边）

首先如图(b)所示,任选一个顶点A作为初始的子树 $T_1 = (\{A\}; \emptyset)$ 。此时, T_1 所对应的割共有AB、AD和AG三条跨越边,故选取其中最短者AB,如图(c)所示将 T_1 扩充至 $T_2 = (\{A, B\}; \{AB\})$ 。此时, T_2 所对应的割共有BC、AD和AG三条跨越边,依然选取其中最短者AD,如图(d)所示将 T_2 扩充至 $T_3 = (\{A, B, D\}; \{AB, AD\})$ 。

如此反复,直至最终如图(i)所示得到:

$T_8 = (\{A, B, D, G, C, E, F, H\}; \{AB, AD, DG, DC, CE, CF, FH\})$
此即原图的最小支撑树。

可以证明,即便出现多条极短跨越边共存的退化情况,以上方法依然可行(习题[6-27])。

■ 实现

以上Prim算法完全可以纳入6.10.2节的优先级搜索算法框架。为此,每次由 T_k 扩充至 T_{k+1} 时,可以将 V_k 之外每个顶点 u 到 V_k 的距离视作 u 的优先级数。如此,每一最短跨越边 e_k 对应的顶点 u_k 都会因拥有最小的优先级数(即最高的优先级)而自然地被选中。

```

1 template <typename Tv, typename Te> struct PrimPU { //针对Prim算法的顶点优先级更新器
2     virtual void operator() ( Graph<Tv, Te>* g, int uk, int v ) {
3         if ( UNDISCOVERED == g->status ( v ) ) //对于uk每一尚未被发现的邻接顶点v
4             if ( g->priority ( v ) > g->weight ( uk, v ) ) { //按Prim策略做松弛
5                 g->priority ( v ) = g->weight ( uk, v ); //更新优先级(数)
6                 g->parent ( v ) = uk; //更新父节点
7             }
8     }
9 };

```

代码6.8 Prim算法的顶点优先级更新器

那么, u_k 和 e_k 加入 T_k 之后,应如何快速更新 V_{k+1} 以外顶点的优先级数呢?实际上,与 u_k 互不关联的顶点都无需考虑,故只需遍历 u_k 的每一邻居 v ,若边 u_kv 的权重小于 v 当前的优先级数,则将后者更新为前者。这一思路可具体落实为如代码6.8所示的优先级更新器。

■ 复杂度

不难看出,以上顶点优先级更新器只需常数的运行时间,故由6.10.3节对优先级搜索算法性能的分析结论,以上Prim算法的时间复杂度为 $O(n^2)$ 。作为PFS搜索的特例,Prim算法的效率也可借助优先级队列进一步提高(习题[10-16]和[10-17])。

§ 6.12 最短路径

若以带权图来表示真实的通讯、交通、物流或社交网络，则各边的权重可能代表信道成本、交通运输费用或交往程度。此时我们经常关心的一类问题（习题[6-8]），可以概括为：

给定带权网络 $G = (V, E)$ ，以及源点（source） $s \in V$ ，对于所有的其它顶点 v ，
 s 到 v 的最短通路有多长？该通路由哪些边构成？

6.12.1 最短路径树

■ 单调性

如图6.22所示，设顶点 s 到 v 的最短路径为 ρ 。于是对于该路径上的任一顶点 u ，若其在 ρ 上对应的前缀为 σ ，则 σ 也必是 s 到 u 的最短路径（之一）。否则，若从 s 到 u 还有另一严格更短的路径 τ ，则易见 ρ 不可能是 s 到 v 的最短路径。

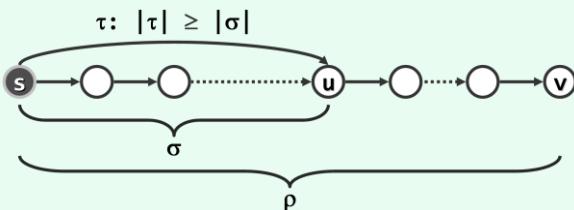


图6.22 最短路径的任一前缀也是最短路径

■ 歧义性

较之最小支撑树，最短路径的歧义性更难处理。首先，即便各边权重互异，从 s 到 v 的最短路径也未必唯一（习题[6-31]）。另外，当存在非正权重的边，并导致某个环路的总权值非正时，最短路径甚至无从定义。因此以下不妨假定，带权网络 G 内各边权重均大于零。

■ 无环性

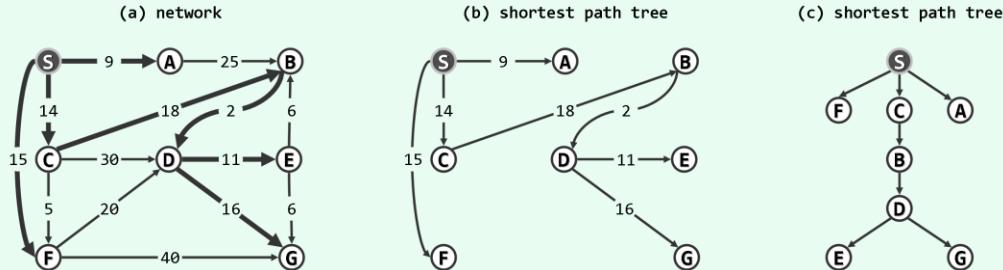


图6.23 有向带权图(a)，及其最短路径树(b)和(c)

在如图6.23(a)所示的任意带权网络中，考查从源点到其余顶点的最短路径（若有多条，任选其一）。于是由以上单调性，这些路径的并集必然不含任何（有向）回路。这就意味着，它们应如图(b)和图(c)所示，构成所谓的最短路径树（shortest-path tree）。

6.12.2 Dijkstra[®]算法

■ 最短路径子树序列

将顶点 u_i 到起点 s 的距离记作： $d_i = \text{dist}(s, u_i)$ ， $1 \leq i \leq n$ 。不妨设 d_i 按非降序排列，即 $d_i \leq d_j$ 当且仅当 $i \leq j$ 。于是与 s 自身相对应地必有： $u_1 = s$ 。

^③ E. W. Dijkstra (1930/05/11-2002/08/06)，杰出的计算机科学家，1972年图灵奖得主

在从最短路径树 T 中删除顶点 $\{u_{k+1}, u_{k+2}, \dots, u_n\}$ 及其关联各边之后，将残存的子图记作 T_k 。于是 $T_n = T$ ， T_1 仅含单个顶点 s 。实际上， T_k 必为一棵树。为验证这一点，只需归纳证明 T_k 是连通的。为从 T_{k+1} 转到 T_k 而删除的顶点 u_{k+1} ，在 T_{k+1} 中必是叶节点。而根据最短路径的单调性，作为 T_{k+1} 中距离最远的顶点， u_{k+1} 不可能拥有后代。

于是，如上定义的子树 $\{T_1, T_2, \dots, T_n\}$ ，便构成一个最短路径子树序列。

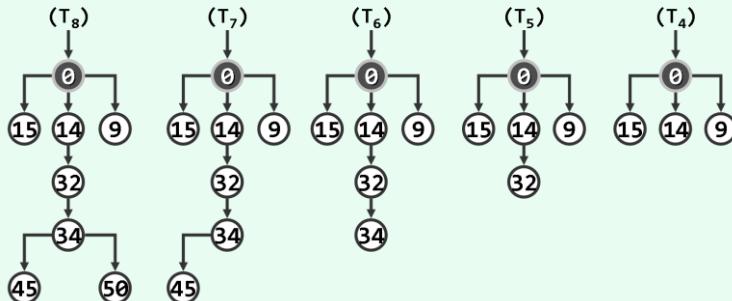


图6.24 最短路径子树序列

仍以图6.23中的最短路径树为例，最后五棵最短路径子树，如图6.24所示。

为便于相互比对，其中每个顶点都注有其到 s 的距离。可见，只需从 T_{k+1} 中删除距离最远的顶点 u_{k+1} ，即可将 T_{k+1} 转换至 T_k 。

■ 贪心迭代

颠倒上述思路可知，只要能够确定 u_{k+1} ，便可反过来将 T_k 扩展为 T_{k+1} 。如此，便可按照到 s 距离的非降次序，逐一确定各个顶点 $\{u_1, u_2, \dots, u_n\}$ ，同时得到各棵最短路径子树，并得到最终的最短路径树 $T = T_n$ 。现在，问题的关键就在于：

如何才能高效地找到 u_{k+1} ？

实际上，由最短路径子树序列的上述性质，每一个顶点 u_{k+1} 都是在 T_k 之外，距离 s 最近者。若将此距离作为各顶点的优先级数，则与最小支撑树的Prim算法类似，每次将 u_{k+1} 加入 T_k 并将其拓展至 T_{k+1} 后，需要且只需要更新那些仍在 T_{k+1} 之外，且与 T_{k+1} 关联的顶点的优先级数。

可见，该算法与Prim算法仅有处差异：考虑的是 u_{k+1} 到 s 的距离，而不再是其到 T_k 的距离。

■ 实现

与Prim算法一样，Dijkstra算法也可纳入此前6.10.2节的优先级搜索算法框架。

为此，每次由 T_k 扩展至 T_{k+1} 时，可将 V_k 之外各顶点 u 到 V_k 的距离看作 u 的优先级数（若 u 与 V_k 内顶点均无联边，则优先级数设为 $+\infty$ ）。如此，每一最短跨边 e_k 所对应的顶点 u_k ，都会因拥有最小的优先级数（或等价地，最高的优先级）而被选中。

```

1 template <typename Tv, typename Te> struct DijkstraPU { //针对Dijkstra算法的顶点优先级更新器
2     virtual void operator() ( Graph<Tv, Te>* g, int uk, int v ) {
3         if ( UNDISCOVERED == g->status ( v ) ) //对于uk每一尚未被发现的邻接顶点v，按Dijkstra策略
4             if ( g->priority ( v ) > g->priority ( uk ) + g->weight ( uk, v ) ) { //做松弛
5                 g->priority ( v ) = g->priority ( uk ) + g->weight ( uk, v ); //更新优先级(数)
6                 g->parent ( v ) = uk; //并同时更新父节点
7             }
8     }
9 };
  
```

代码6.9 Dijkstra算法的顶点优先级更新器

唯一需要专门处理的是，在 u_k 和 e_k 加入 T_k 之后，应如何快速地更新 V_{k+1} 以外顶点的优先级数。实际上，只有与 u_k 邻接的那些顶点，才有可能在此后降低优先级数。因此与Prim算法一样，也可遍历 u_k 的每一个邻居 v ，只要边 u_kv 的权重加上 u_k 的优先级数，小于 v 当前的优先级数，即可将后者更新为前者。具体地，这一思路可落实为如代码6.9所示的优先级更新器。

■ 实例

依然以图6.21(a)中无向图为例，一次Dijkstra算法的完整执行过程如图6.25所示。

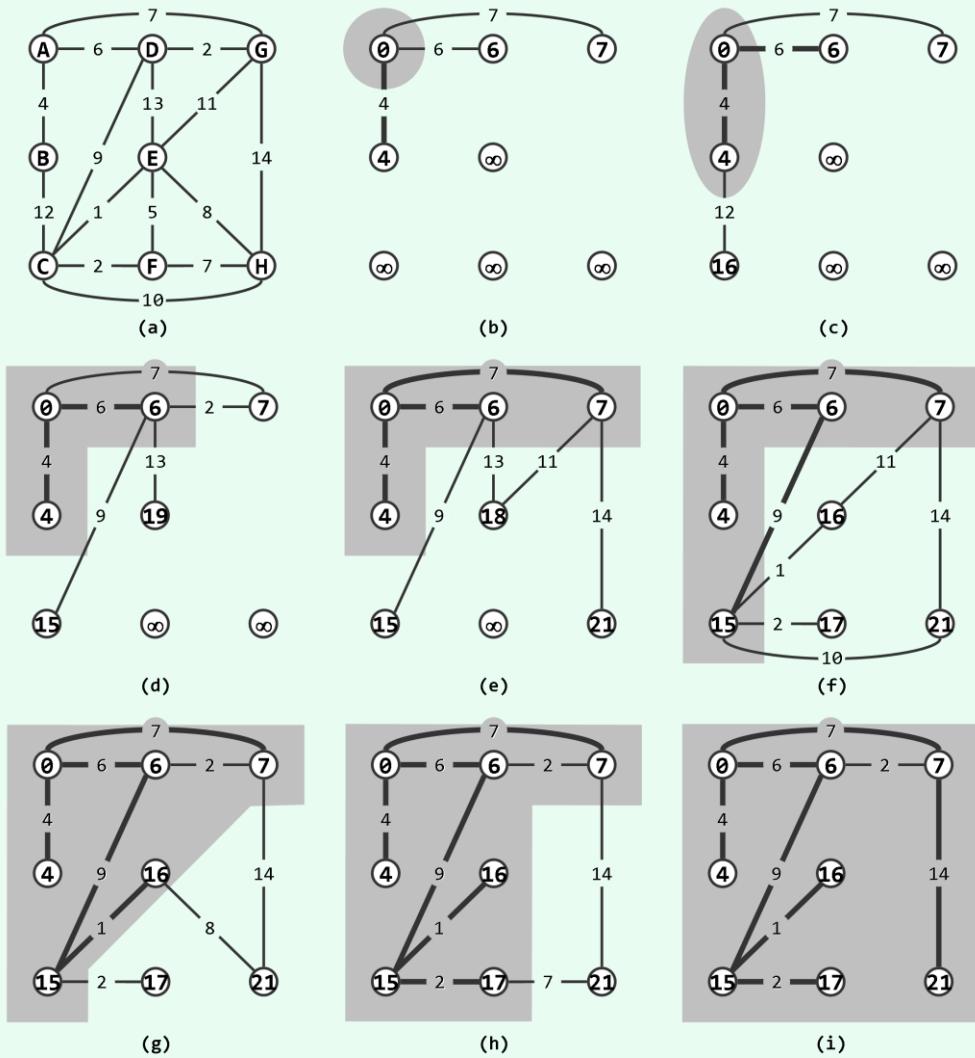


图6.25 Dijkstra算法示例（阴影区域示意不断扩展的子树 T_k ，粗线示意树边）

■ 复杂度

不难看出，以上顶点优先级更新器只需常数运行时间。同样根据6.10.3节对PFS搜索性能的分析结论，Dijkstra算法这一实现版本的时间复杂度为 $O(n^2)$ 。

作为PFS搜索的特例，Dijkstra算法的效率也可借助优先级队列进一步提高（习题[10-16]和[10-17]）。