

湖南科技大学课程教案

(章节、专题首页)

授课教师: 王志喜

职称: 副教授

单位: 计算机科学与工程学院

课程名称	算法设计与分析
章节、专题	图的遍历
教学目标及基本要求	熟悉图论的基本知识, 掌握基本的搜索和遍历技术, 以及二叉树、一般树和图的简单应用。
教学重点	图的基本知识, 二叉树的遍历, 一般树的遍历, 一般图的遍历。
教学难点	堆的操作与实现, 合并查找结构, 一般图的遍历, 连通图的双连通分支。
教学内容与时间分配	<p>(1) 图的基本知识 (2课时)</p> <p>(2) 二叉树的遍历 (0.75课时)</p> <p>(3) 堆的操作与实现 (0.6课时)</p> <p>(4) 一般树的遍历 (0.5课时)</p> <p>(5) 合并查找结构 (0.4课时)</p> <p>(6) 一般图的遍历 (0.75课时)</p> <p>(7) 连通图的双连通分支 (1课时)</p> <p>共计6.5课时。</p>
习题	第3.8.1节(基础知识题), 第3.8.2节(实验题)。

第3章 图的遍历

本章复习图论的基本知识，介绍基本的搜索和遍历技术（包括二叉树、一般树及图的遍历和搜索技术）以及二叉树、一般树和图的简单应用（包括堆的基本知识和基本操作、合并查找结构以及连通图的双连通分支）。

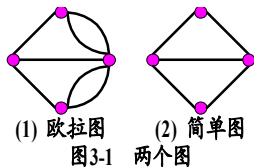
- 图的基本知识
- 二叉树的遍历
- 堆的操作与实现
- 一般树的遍历
- 合并查找结构
- 一般图的遍历
- 连通图的双连通分支

3.1 图的基本知识

3.1.1 图的基本概念

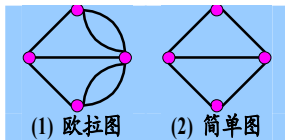
1. 图的定义

图是一个二元组 $G=(V,E)$ ，其中， V 是一个非空有限的顶点集合， E 是一个有限的边集合，每条边对应一个顶点对（如图3-1所示）。顶点数为 n 的图称为 n 阶图。



如果多条边对应同一个顶点对，则称这些边为一组重边。如果一条边对应的顶点对的两个顶点相同，则称该边为一个自圈。没有重边和自圈的图称为简单图。如图3-1(2)所示。简单图的边通常用对应的顶点对表示，表示形式形如 (x, y) 或 xy 。

本书只讨论简单图。



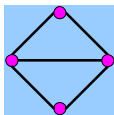
2. 边和顶点的关系

(1) 端点。一条边对应的2个顶点称为该边的端点。

(2) 关联。一条边对应2个顶点也可以说成该边关联或连接这2个顶点。

(3) 相邻。与同一条边关联的顶点称为相邻（邻接）顶点，与同一个顶点关联的边称为相邻（邻接）边。

(4) 度数。图中与顶点 v 关联的边数称为 v 的度数，记做 $d(v)$ 。度数为0的顶点称为孤立点。图 $G=(V,E)$ 中顶点的度数与边数满足关系 $\sum \{d(v): v \in V\} = 2|E|$ 。



3. 路与回路

使用图3-2(1)所示的图为例说明。

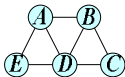
(1) 途径。途径（通路，链）是相邻顶点都对应边的顶点序列，途径的起点和终点是序列的起点和终点，途径的长度是途径包含的边数，如图3-2(2)所示。

(2) 路径。路径（路，基本路径）是无相同顶点的途径，如图3-2(3)所示。

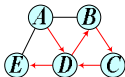
(3) 回路。回路（闭途径）是起点和终点相同的途径，如图3-2(4)所示。

(4) 圈。圈（环，基本回路）是除起点和终点外无其他相同顶点的回路，如图3-2(4)所示。

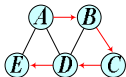
(5) Hamilton通路与Hamilton回路。Hamilton通路是经过图中每个顶点正好一次的路径，如图3-2(3)所示。Hamilton回路（Hamilton环，Hamilton圈）是经过图中每个顶点正好一次的回路，如图3-2(4)所示。



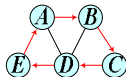
(1) 一个图



(2) 途径ADBCDE



(3) 路径ABCDE



(4) 回路ABCDEA

图3-2 路径

3.1.2 无向图

如果边对应的顶点对是无序对，即顶点对 xy 和 yx 对应同一条边，则称该边是无向边。如果图中的边都是无向边，则称该图是一个无向图。

1. 空图与完全图

(1) 空图。没有边的图称为空图，如图3-3中的 K_1 。

(2) 完全图。任何两个顶点都相邻的无向图称为完全图。 n 阶完全图有 n 个顶点，有 $n(n-1)/2$ 条边，通常记作 K_n 。如图3-3所示。

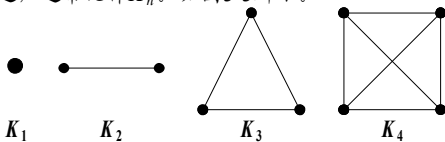


图3-3 1-4阶完全图

2. 无向图的连通性

(1) 可达性。在无向图中，顶点 u 可达顶点 v 是指存在一条以 u 为起点、 v 为终点的途径（或路径）。

(2) 连通图。称一个无向图是连通的，是指该图中任何两个顶点之间都是可达的。连通的无向图称为连通图，不是连通的无向图称为非连通图。例如，图3-4(1)中的无向图是连通图，而图3-4(2)中的无向图是非连通图。

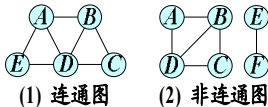


图3-4 无向图的连通性

3. 无向图的着色

(1) 着色。为一个无向图着色是指为该图的每一个顶点都指定一种颜色，使得任何两个相邻顶点的颜色都不相同。其中，任何一种只需使用 m 种颜色的着色方案都称为该图的一种 m -着色。如果一个无向图至少有一种 m -着色，则称该图是可 m 着色的。例如，图3-5所示的两个无向图都是可3着色的。

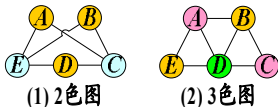


图3-5 无向图的着色

(2) 色数。如果一个无向图是可 m 着色的，但不是可 $(m-1)$ 着色的，则称 m 为该图的色数，该图是一个 m 色图。例如，图3-5给出了一个2色图和一个3色图。

3.1.3 子图

本小节介绍几种子图，以及一些与这几种子图相关的顶点集和边集。

1. 子图

(1) 子图。对于图 $G=(V,E)$ 和 $H=(V',E')$ ，如果 $V' \subseteq V$ ， $E' \subseteq E$ ，则称图 H 是图 G 的子图，图 G 是图 H 的母图。例如，在图3-6中，图 T 和图 C 都是图 G 的子图。

(2) 生成子图。由原图去掉一些边（顶点集不变）得到的子图称为原图的生成子图。例如，在图3-6中，图 T 是图 G 的一个生成子图。

(3) 导出子图。由原图去掉一些顶点及与这些顶点关联的边得到的子图称为原图的导出子图，通常将导出子图的顶点集 U 称为原图的导出子图 U 。例如，例如，在图3-6中，图 C 是图 G 的一个导出子图。

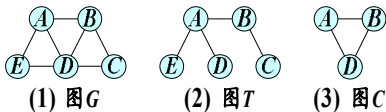


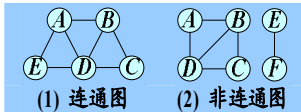
图3-6 生成子图和导出子图

2. 连通子图

如果无向图 G 的一个子图 H 是一个连通图，则称 H 是 G 的一个连通子图。

如果 G 的连通子图 H 不是 G 的其他连通子图的子图，则称 H 是 G 的一个连通分支（也称为极大连通子图）。

非连通图可以分成几个连通分支。例如，4-4(2)所示的非连通图有两个连通分支，顶点集分别是 $\{A, B, C, D\}$ 和 $\{E, F\}$ 。



3. 完全子图

如果无向图 G 的导出子图 U 是一个完全图，则称 U 是 G 的一个完全子图或 G 的一个团。

团的顶点数称为它的大小。大小为 k 的团称为 k 团。

G 的团 U 是 G 的极大团当且仅当 U 不包含在 G 的更大团中。在如图3-7所示的无向图 G 中， $\{A, B\}$ 是 G 的2团。这个团不是极大团，因为它包含在3团 $\{A, B, D\}$ 中。

G 的最大团是指 G 中顶点数最多的团。在如图3-7(1)所示的无向图 G 中， $\{A, B, D\}$ 、 $\{A, D, E\}$ 和 $\{B, C, D\}$ 都是 G 的最大团。

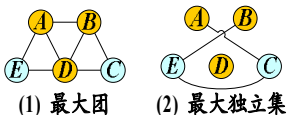


图3-7 最大团与最大独立集

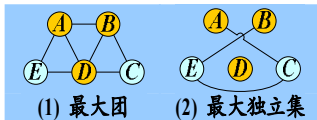
4. 空子图

如果无向图 G 的导出子图 U 是一个空图，则称 U 是 G 的一个空子图或 G 的一个独立集。

独立集的顶点数称为它的大小。大小为 k 的独立集称为 k 独立集。

G 的独立集 U 是 G 的极大独立集当且仅当 U 不包含在 G 的更大独立集中。在如图3-7所示的无向图 \bar{G} 中， $\{A, B\}$ 是 \bar{G} 的2独立集。这个独立集不是极大独立集，因为它包含在3独立集 $\{A, B, D\}$ 中。

G 的最大独立集是指 G 中顶点数最多的独立集。在如图3-7(2)所示的无向图 \bar{G} 中， $\{A, B, D\}$ 、 $\{A, D, E\}$ 和 $\{B, C, D\}$ 都是 \bar{G} 的最大独立集。

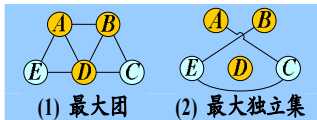


5. 顶点覆盖

设 V' 是无向图 G 的一个顶点子集，如果 G 的每一条边都有一个端点在 V' 中，则称 V' 是 G 的一个顶点覆盖。

顶点覆盖的顶点个数称为它的大小。大小为 k 的顶点覆盖称为 k 顶点覆盖，顶点数最少的顶点覆盖称为最小顶点覆盖。最小顶点覆盖的顶点数称为 G 的顶点覆盖数。

例如，对于图3-7(1)所示的无向图， $\{A, B, D\}$ 、 $\{A, C, D\}$ 和 $\{B, D, E\}$ 都是它的最小顶点覆盖。



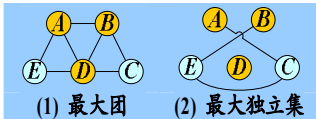
6. 各类子图之间的联系

为了方便描述各类子图之间的联系，这里首先给出补图的含义。

(1) 补图。对于无向图 $G=(V,E)$ 和 $\bar{G}=(V,\bar{E})$ ，如果 $(u,v) \in \bar{E}$ 当且仅当 $(u,v) \notin E$ ，则称 \bar{G} 是 G 的补图。例如，图3-7(2)中的图是图3-7(1)中的图的补图。

(2) 独立集、团和顶点覆盖的联系。对于 n 阶无向图 $G=(V,E)$ ，下列结论成立。

- U 是 G 的 k 团当且仅当 U 是补图 \bar{G} 的 k 独立集，并且， U 是 G 的最大团当且仅当 U 是补图 \bar{G} 的最大独立集。
- U 是 G 的 k 团当且仅当 $V-U$ 是补图 \bar{G} 的 $n-k$ 顶点覆盖，并且， U 是 G 的最大团当且仅当 $V-U$ 是补图 \bar{G} 的最小顶点覆盖（证明可参阅“11.5 NP完全性的证明举例”一节）。



3.1.4 有向图和加权图

1. 有向图的基本概念

(1) 定义。如果边对应的顶点对是有序对，即顶点对 xy 和 yx 对应不同的边，则称该边是有向边。如果图中的边都是有向边，则称该图是一个有向图。有向图实际上是在无向图的基础上进一步指定各个边的方向。如图3-8所示。

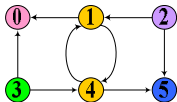
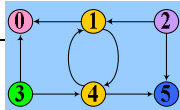


图3-8 有向图

【注】在有向图中，边、途径、回路、路径和圈等都是有序的。

(2) 有向图与无向图的关系。有向图和无向图可以相互转化。将一个无向图的每条边都规定方向后，即得到一个有向图，称为原图的一个定向图。将一个有向图的各条边的方向去掉，即得到一个无向图，称为原图的基础图。

(3) 有向图中顶点的度数。在有向图中，顶点 v 的出度是指由顶点 v 出发的边数，记作 $d^+(v)$ ，而入度则是指向顶点 v 的边数，记作 $d^-(v)$ 。在有向图中，顶点的度是该顶点的出度与入度之和，即 $d(v) = d^+(v) + d^-(v)$ 。



2. 有向图的连通性

(1) 连通。有向图是连通的是指其基础图是连通的。

(2) 可达。在有向图中，顶点 u 可达顶点 v 是指存在一条以 u 为起点、 v 为终点的途径或路径。

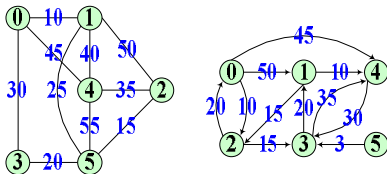
(3) 双向连通。如果有向图 D 中任意两个顶点都是可达的，则说有向图 D 是双向连通的或强连通的。例如，图3-8所示的有向图是连通的，但不是双向连通的，因为没有到达顶点3的路径。

注意，双向连通不是双连通。双连通是一个针对无向图的概念。

3. 加权图的含义

加权图(赋权图、带权图)是指对图的每条边 e 赋予一个值 $w(e)$ (如图3-9所示)。例如，架设连接各城镇的交通路网需要考虑各段线路的修建费用，在运输网络中需要考虑网络各段线路的容量。

加权图也可以分为有向加权图和无向加权图2种，如图3-9所示。



(1) 无向加权图

(2) 有向加权图

图3-9 加权图

4. 加权图中的路与回路

(1) 路的权值。在加权图中，一条路径的各边权值之和称为该路径的总权值、权值或长度。对于边权值非负的加权图，可以根据问题背景将路径的权值称为该路径的成本、代价、耗费或效益等。

(2) 最短路径。在加权图中，顶点 i 和 j 之间的最短路径是指从顶点 i 出发到顶点 j 的所有路径中总权值最小的那一条路径。

(3) 回路的权值。在加权图中，一条回路的各边权值之和称为该回路的总权值、权值或长度。对于边权值非负的加权图，可以根据问题背景将回路的权值称为该回路的成本、代价、耗费或效益等。

(4) 旅行商回路。加权图中任何一个Hamilton圈都称作该图的一个旅行（周游路线，旅行商回路）。边权值非负的加权图中耗费最小的旅行称为该图的最优旅行。

3.1.5 树和森林

1. 树的定义和相关术语

不含圈的连通图称为树。如图3-10所示。

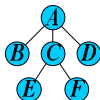


图3-10 树

通常在树中指定一个顶点作为树的根，与根相邻的顶点称为根的儿子，相应地，根称为这些儿子的双亲或父亲，双亲相同的顶点称为兄弟。如果在树中去掉根，则可以得到一些以原根的儿子为根的树，这些树称为原根的子树。

没有儿子的顶点称为叶子，非叶子顶点也称为分枝顶点，非根的分枝顶点称为内部顶点。

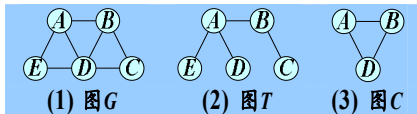
从根到一个顶点的路径包含的顶点数称为该顶点的级或层次。顶点层次的最大值称为树的高度或深度。

为了方便讨论，通常将空顶点集（没有顶点）称为空树，空树的深度规定为0。

2. 生成树

如果某连通图的一个生成子图是一棵树，则称这棵树为原图的生成树。例如，例如，在图3-6中，图 T 是图 G 的一颗生成树。

在树中，顶点数 n 和边数 m 满足 $m = n - 1$ 。生成树是其所在的无向图中边数最少的连通子图。因此，有 n 个顶点的连通图的边数至少是 $n - 1$ 。



3. 森林

森林是没有圈的无向图。森林的连通分支是树，也就是说，森林是由一些不相交的树组成的，如图3-11中的图H。非连通图没有生成树，但有生成森林，例如，在图3-11中，图H是图G的生成森林。如果非连通图G有 k 个连通分支，则G的边数至少是 $n-k$ 。

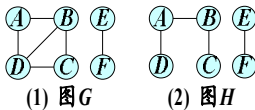
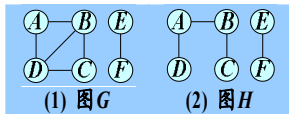


图3-11 生成森林

4. 相关性质

若无向图 G 的顶点数为 n ，边数为 m ，则下列结论成立。

- 若 G 是树，则 $m = n - 1$ 。
- 若 G 是连通图，而且满足 $m = n - 1$ ，则 G 是树。
- 若 G 不包含圈，而且满足 $m = n - 1$ ，则 G 是树。
- 若 G 是由 k 棵树构成的森林，则 $m = n - k$ 。
- 若 G 有 k 个连通分支，而且满足 $m = n - k$ ，则 G 是森林。



5. 加权图的生成树

(1) 加权树。加权树（赋权树、带权树）是指对树的每条边都赋予一个权值。

(2) 生成树的权值。在连通的加权图中，一棵生成树的各边权值之和称为该生成树的总权值（权值，成本，耗费，代价），其中，生成树中各边的权值就是原图中各边的权值。

(3) 最小生成树。在一个连通的加权图中，总权值最小的生成树称为该图的最小生成树。

3.1.6 二叉树

1. 相关定义

(1) 有序树。如果将树中每个顶点的各子树看成是从左到右有次序的（即不能互换），则称该树为有序树，否则称为无序树或自由树。

(2) 二叉树。每个顶点至多有两个儿子（分别称为左儿子和右儿子）的有序树称为二叉树。图3-12给出了2棵二叉树。

(3) 满二叉树。不在最大层上的每个顶点都有两个儿子的二叉树称为满二叉树。如图3-12(1)所示。

(4) 完全二叉树。不在层次最大的两层上的每个顶点都有两个儿子，且最大层的顶点只能在最左侧连续集中出现的二叉树称为完全二叉树。如图3-12(2)所示。

显然，完全二叉树可由相同高度的满二叉树连续删除最右侧的若干叶子得到。

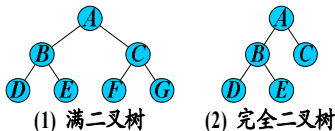


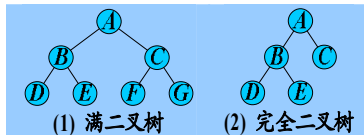
图3-12 二叉树

2. 基本性质

(1) 二叉树的第 i 层最多有 2^{i-1} 个顶点，满二叉树的第 i 层正好有 2^{i-1} 个顶点，其中 $i \geq 1$ 。

(2) 深度为 k 的二叉树最多有 $2^k - 1$ 个顶点和 2^{k-1} 个叶子，深度为 k 的满二叉树正好有 $2^k - 1$ 个顶点和 2^{k-1} 个叶子。

(3) 含 n 个顶点的完全二叉树的深度不超过 $\log(2n) = \log(n) + 1$ 。因为深度为 k 的完全二叉树至少比深度为 $k-1$ 的满二叉树至少多一个叶子，所以它的顶点数 n 不会少于 2^{k-1} ，从而由 $2^{k-1} \leq n$ 可得 $k \leq \log(2n)$ 。



3.1.7 图的表示方法

这里只介绍图的邻接矩阵表示方法。树和二叉树具有比邻接矩阵更方便的表示方法，这些表示方法将在后续部分介绍。

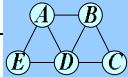
1. 无向图的邻接矩阵

设无向图 G 的顶点集 $V = \{v_1, v_2, \dots, v_n\}$ ，则顶点与顶点之间的邻接关系可以表示成邻接矩阵

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

其中

$$a_{ij} = \begin{cases} 1 & v_i v_j \text{ 是边} \\ 0 & \text{否则} \end{cases}$$



例如，图3-4(1)所示的无向图对应的邻接矩阵为

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

显然，无向图的邻接矩阵一定是对称的。

2. 有向图的邻接矩阵

有向图的邻接矩阵不一定对称。设有向图 D 的顶点集是 $V = \{v_1, v_2, \dots, v_n\}$ ，则顶点与顶点之间的邻接关系可以表示成邻接矩阵

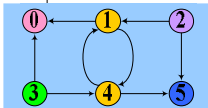
$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

其中

$$a_{ij} = \begin{cases} 1 & v_i v_j \text{ 是有向边} \\ 0 & \text{否则} \end{cases}$$

例如，图3-8所示的有向图对应的邻接矩阵为

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$



3. 无向加权图的邻接矩阵

设无向加权图 G 的顶点集是 $V = \{v_1, v_2, \dots, v_n\}$, 则 G 的邻接矩阵为

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

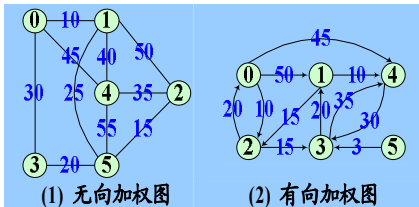
其中

$$a_{ij} = \begin{cases} \text{权值} & v_i v_j \text{ 是边} \\ \infty & v_i v_j \text{ 不是边} \\ 0 & v_i = v_j \end{cases}$$

例如，图3-9(1)所示的无向加权图对应的邻接矩阵为

$$A = \begin{pmatrix} 0 & 10 & \infty & 30 & 45 & \infty \\ 10 & 0 & 50 & \infty & 40 & 25 \\ \infty & 50 & 0 & \infty & 35 & 15 \\ 30 & \infty & \infty & 0 & \infty & 20 \\ 45 & 40 & 35 & \infty & 0 & 55 \\ \infty & 25 & 15 & 20 & 55 & 0 \end{pmatrix}$$

显然，无向加权图的邻接矩阵一定是对称的。



4. 有向加权图的邻接矩阵

设有向加权图 D 的顶点集是 $V = \{v_1, v_2, \dots, v_n\}$, 则 D 的邻接矩阵为

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

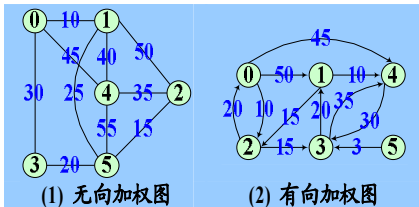
其中

$$a_{ij} = \begin{cases} \text{权值} & v_i v_j \text{ 是有向边} \\ \infty & v_i v_j \text{ 不是有向边} \\ 0 & v_i = v_j \end{cases}$$

例如，图3-9(2)所示的有向加权图对应的邻接矩阵为

$$A = \begin{pmatrix} 0 & 50 & 10 & \infty & 45 & \infty \\ \infty & 0 & 15 & \infty & 10 & \infty \\ 20 & \infty & 0 & 15 & \infty & \infty \\ \infty & 20 & \infty & 0 & 35 & \infty \\ \infty & \infty & \infty & 30 & 0 & \infty \\ \infty & \infty & \infty & 3 & \infty & 0 \end{pmatrix}$$

显然，有向加权图的邻接矩阵不一定是对称的。



3.2 二叉树的遍历

本节介绍二叉树的搜索算法，下一节介绍一个二叉树的应用实例（使用二叉树实现堆）。

3.2.1 二叉树的深度优先遍历

二叉树的深度优先遍历可以分为先根次序遍历、中根次序遍历、后根次序遍历等三种方式。

1. 二叉树的先根次序遍历

- (1) 若 T 为空，则返回。
- (2) 访问 T 的根。
- (3) 按二叉树的先根次序依次遍历 T 的左子树和右子树。

图3-13演示了先根次序遍历二叉树的执行过程及结果。

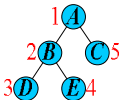


图3-13 二叉树的先根次序遍历

2. 二叉树的后根次序遍历

- (1) 若 T 为空，则返回。
- (2) 按二叉树的后根次序依次遍历 T 的左子树和右子树。
- (3) 访问 T 的根。

图3-14演示了后根次序遍历二叉树的执行过程及结果。

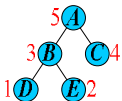


图3-14 二叉树的后根次序遍历

3. 二叉树的中根次序遍历

- (1) 若 T 为空，则返回。
- (2) 按二叉树的中根次序遍历 T 的左子树。
- (3) 访问 T 的根。
- (4) 按二叉树的中根次序遍历 T 的右子树。

图3-15演示了中根次序遍历二叉树的执行过程及结果。

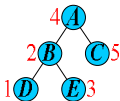


图3-15 二叉树的中根次序遍历

3.2.2 二叉树的宽度优先遍历

宽度优先遍历就是按层次遍历。

- (1) 若 T 非空，则访问 T 的根并将根加入队列。
- (2) 若队列非空，取出队首元素 x 。
- (3) 访问 x 的左右儿子并将左右儿子加入队列。
- (4) 重复(2) ~ (3)，直到队列为空。

图3-16演示了宽度优先遍历二叉树的执行过程及结果。

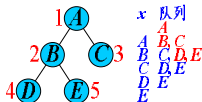
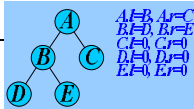


图3-16 二叉树的宽度优先遍历



3.2.3 二叉树遍历算法的C++实现

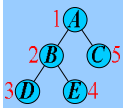
1. 二叉树的C++表示

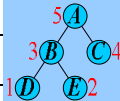
```
#include "algorithm.h"

template<class T> // 二叉树结点
struct BtNode
{
    T data;
    BtNode *left = 0, *right = 0; // 左子树, 右子树
};
```

2. 二叉树的先根次序遍历

```
template<class T, class Func> // 先根遍历
void PreOrder(BtNode<T> *x, Func Visit)
{
    if(x == 0) return;
    Visit(x); // 访问根
    PreOrder(x->left, Visit); // 遍历左子树
    PreOrder(x->right, Visit); // 遍历右子树
} // 耗时O(n)
```



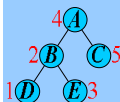


3. 二叉树的后根次序遍历

```
template<class T, class Func> // 后根遍历
void PostOrder(BtNode<T> *x, Func Visit)
{
    if(x == 0) return;
    PostOrder(x->left, Visit); // 遍历左子树
    PostOrder(x->right, Visit); // 遍历右子树
    Visit(x); // 访问根
} // 耗时O(n)
```

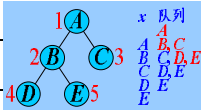
4. 二叉树的中根次序遍历

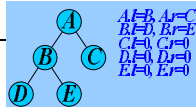
```
template<class T, class Func> // 中根遍历
void InOrder(BtNode<T> *x, Func Visit)
{
    if(x == 0) return;
    InOrder(x->left, Visit); // 遍历左子树
    Visit(x); // 访问根
    InOrder(x->right, Visit); // 遍历右子树
} // 耗时O(n)
```



5. 二叉树的宽度优先遍历

```
template<class T, class Func> // 宽度优先遍历
void LevelOrder(BtNode<T> *x, Func Visit)
{
    queue<BtNode<T>*> Q;
    if(x != 0) Visit(x), Q.push(x); // 访问根并将根加入队列
    while(not Q.empty())
    {
        x = Q.front(), Q.pop();
        // 访问x的左右儿子并将左右儿子加入队列
        auto left = x->left, right = x->right;
        if(left != 0) Visit(left), Q.push(left);
        if(right != 0) Visit(right), Q.push(right);
    }
} // 耗时O(n)
```





6. 测试程序^{*}

使用图3-13~图3-16中使用的二叉树。

```
int main()
{
    auto Visit = [](BtNode<char> *x) // 访问结点x
    {
        cout << x->data << ' ';
    };

    BtNode<char> a{'A'}, b{'B'}, c{'C'}, d{'D'}, e{'E'};
    // 建立一棵以a为根的二叉树
    a.left = &b, a.right = &c, b.left = &d, b.right = &e;

    cout << "先根次序: ";
    PreOrder(&a, Visit), cout << endl; // A B D E C
    cout << "中根次序: ";
    InOrder(&a, Visit), cout << endl; // D B E A C
    cout << "后根次序: ";
    PostOrder(&a, Visit), cout << endl; // D E B C A
    cout << "宽度优先遍历: ";
    LevelOrder(&a, Visit), cout << endl; // A B C D E
}
```

3.2.4 复杂性分析

1. 结论

假设树 T 有 n 个顶点, 用 $t(n)$ 和 $s(n)$ 分别表示这些遍历算法中的任意一个算法所需要的最大时间和空间。如果访问一个顶点所需要的时间和空间都是 $\Theta(1)$, 则 $t(n) = O(n)$, $s(n) = O(n)$ 。

下面分别针对前述4种算法证明该结论。

2. 宽度优先遍历

对于宽度优先遍历, 因为每个顶点正好进入队列一次, 容易看出, 结论成立。

```
template<class T, class Func> // 宽度优先遍历
void LevelOrder(BtNode<T> *x, Func Visit)
{
    queue<BtNode<T>*> Q;
    if(x != 0) Visit(x), Q.push(x);
    while(not Q.empty())
    {
        x = Q.front(), Q.pop();
        auto left = x->left, right = x->right;
        if(left != 0) Visit(left), Q.push(left);
        if(right != 0) Visit(right), Q.push(right);
    }
} // 耗时O(n)
```

3. 深度优先遍历算法

三种深度优先遍历算法可以统一考虑。这里首先证明 $s(n) = O(n)$ ，其次证明 $t(n) = O(n)$ 。

(1) 证明 $s(n) = O(n)$ 。只有为了保存递归调用时那些局部变量的值时，才需要附加递归栈空间。如果 T 的深度为 d ，则这个空间显然是 $\Theta(d)$ 。对于一棵 n 个顶点的二叉树，显然有 $d \leq n$ ，从而 $s(n) = O(n)$ 。

```
template<class T, class Func>
void PreOrder(BtNode<T> *x, Func Visit)
{ if(x == 0) return;
  Visit(x);
  PreOrder(x->left, Visit);
  PreOrder(x->right, Visit);
}
```

(2)证明 $t(n) = O(n)$ 。只需证明存在 $c > 0$ ，使得当 n 充分大时，有 $t(n) \leq cn$ 。

遍历算法做的工作由在递归一级做的工作（耗时可由常数 b 限界）和递归调用做的工作组成。容易看出，当 T 的左子树顶点数为 n_1 时，

$$t(n) = \max_{n_1} \{ t(n_1) + t(n - n_1 - 1) + b \}。$$

下面使用数学归纳法证明 $t(n) \leq cn$ ，其中常数 $c \geq b$ 。

由 $t(0) \leq b$ 和 $t(1) \leq b$ 可知，当 $n = 1$ 时，
 $t(n) \leq cn$ 成立。

假定当 $1 \leq n < m$ 时， $t(n) \leq cn$ 成立，
 现在证明当 $n = m$ 时， $t(n) \leq cn$ 也成立。

设 T 是一棵 m 个顶点的树， m_1 是 T 的左子树的顶点数，于是

$$\begin{aligned} t(m) &= \max_{m_1} \{ t(m_1) + t(m - m_1 - 1) + b \} \\ &\leq \max_{m_1} \{ cm_1 + c(m - m_1 - 1) + b \} \\ &= \max_{m_1} \{ cm + b - c \} \\ &\leq cm \end{aligned}$$

```
template<class T, class Func>
void PreOrder(BtNode<T> *x, Func Visit)
{ if(x == 0) return;
  Visit(x);
  PreOrder(x->left, Visit);
  PreOrder(x->right, Visit);
}
```

3.3 堆的操作与实现^{*}

3.3.1 数据结构

1. 二叉树的顺序表示

用一个一维数组 $\text{buf}[n]$ 来表示二叉树（可使用 std::vector 表示一维数组），其中 n 是允许存入二叉树中元素的最大数目。根元素存放在 $\text{buf}[0]$ 中，第 i 号元素的左右儿子分别存放在 $\text{buf}[2*i+1]$ 和 $\text{buf}[2*i+2]$ 中。如图3-17所示。

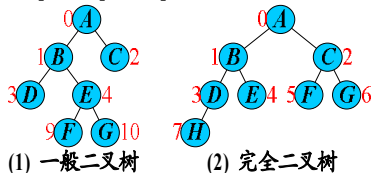


图3-17 二叉树的顺序表示

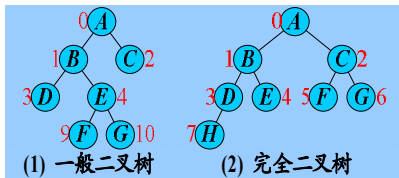
显然，如果二叉树中的元素在该数组中正好是从第0号位置开始连续存放的，则该二叉树是一棵完全二叉树。

2. 二叉树顺序表示的优缺点

显然，二叉树的顺序表示可能会造成数组元素不能连续使用，有很多未用空间。例如，对于图3-17(1)所示的二叉树，第5, 6, 7, 8号位置都是未用空间。但是，这种表示方法能够直接访问每个元素的双亲和儿子，第 i 号元素的双亲位于 $\lfloor (i-1)/2 \rfloor$ 位置，儿子分别位于 $2i+1$ 和 $2i+2$ 位置。

3. 堆的含义

堆分为最小堆和最大堆2种。最小堆是一种元素能够比较大小的完全二叉树，其中每个元素均不大于它的儿子。最大堆是一种元素能够比较大小的完全二叉树，其中每个元素均不小于它的儿子。显然，在最小堆中，根元素是最小的元素，在最大堆中，根元素是最大的元素。



3.3.2 主要操作

堆的基本操作主要是插入元素和删除元素，也可称为入队和出队。本小节以最小堆为例说明堆的基本操作和实现方法，最大堆的基本操作和实现方法可对应完成。

2. 删除元素

只能删除根元素。如图3-19所示，从根元素开始，根据下层元素与原最后元素的比较结果，将较小的元素逐步上移，直到无需上移元素（已经找到原最后元素的插入位置）。

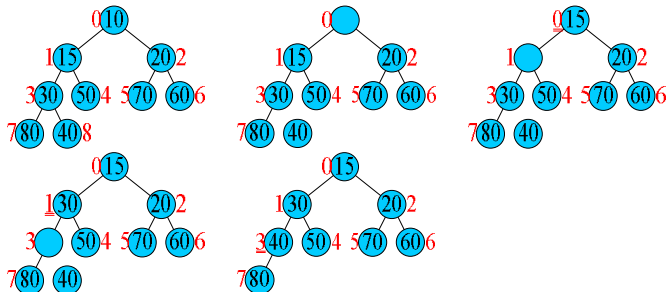


图3-19 删除堆的根元素

该过程耗时 $O(\log n)$ ， n 是删除前元素个数。这是因为有 n 个顶点的完全二叉树的深度不超过 $\log(2n)$ 。

3.3.3 堆的C++实现

这里以最小堆MinHeap为例给出C++ STL中priority_queue类模板的一个简化实现，保存在文件MinHeap.h中。

1. 预处理

```
// minheap.h  
#pragma once  
#include <vector>
```

2. 数据成员

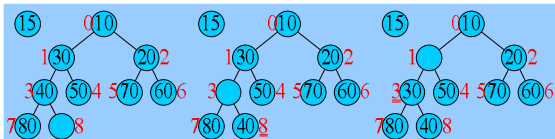
```
template<class T>  
class MinHeap  
{   std::vector<T> buf; // 一维数组
```

3. 访问元素

```
public:  
    const T &top() const // 只读，堆中元素不允许改写  
    {   return buf[0];  
    }
```

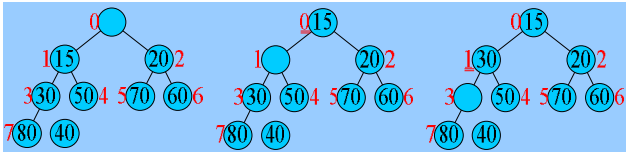
4. 插入元素

```
void push(const T &v)
{   int sz = buf.size(); // 原大小
    buf.resize(sz + 1); // 增加一个元素
    int i = sz; // 当前结点，代表空位，初始为新元素
    while(i > 0) // 寻找第一个小于v的元素
    {   int up = (i - 1) / 2; // 上层结点
        if(buf[up] < v) break; // 已经找到插入位置，不再移动元素
        buf[i] = buf[up], i = up; // 否则，大元素下移，空位上移
    }
    buf[i] = v; // 插入v
} // 耗时O(log(n))
```



5. 删除元素

```
void pop()
{
    if(buf.empty()) return; // 空的
    const T &v = buf.back(); // 原尾部元素
    int i = 0; // 当前结点, 代表空位, 初始为根
    for(;;) // 寻找原最后元素的插入位置, 即寻找第一个大于v的元素
    {
        int ch = 2 * i + 1, ch2 = ch + 1; // 左儿子, 右儿子
        if(ch2 >= buf.size()) break; // 已经超出原尾部(越界)
        if(buf[ch2] < buf[ch]) ch = ch2; // 选取较小的儿子
        if(buf[ch] > v) break; // 已经找到插入位置, 不再移动元素
        buf[i] = buf[ch], i = ch; // 否则, 小元素上移, 空位下移
    }
    buf[i] = v; // 插入原尾部元素
    buf.pop_back(); // 减少一个元素
} // 耗时O(log(n))
```



6. 空堆判断

```
bool empty() const  
{ return buf.empty();  
}  
};
```


3.4 一般树的遍历

本节介绍一般树的搜索算法，下一节介绍一个一般树的应用实例（使用树结构实现不相交集合并与查找）。

3.4.1 一般树的遍历算法

可以将二叉树的父子关系推广到树上。每个父亲的儿子可以有多个，而且可以排出顺序。于是，关于二叉树的几种遍历算法完全可以移植到一般的树上来。

1. 树的先根次序遍历

(1) 若 T 为空，则返回。

(2) 访问 T 的根。

(3) 按树的先根次序依次遍历 T 的所有子树。

图3-20演示了先根次序遍历树的执行过程及结果。

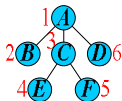


图3-20 树的先根次序遍历

2. 树的后根次序遍历

- (1) 若 T 为空，则返回。
- (2) 按树的后根次序依次遍历 T 的所有子树。
- (3) 访问 T 的根。

图3-21演示了后根次序遍历树的执行过程及结果。

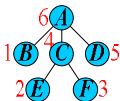


图3-21 树的后根次序遍历

3. 树的中根次序遍历

- (1) 若 T 为空，则返回。
- (2) 按树的中根次序遍历 T 的第一棵子树。
- (3) 访问 T 的根。
- (4) 按树的中根次序依次遍历 T 的其余子树。

图3-22演示了中根次序遍历树的执行过程及结果。

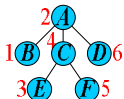


图3-22 树的中根次序遍历

4. 树的宽度优先遍历

- (1) 若 T 非空，则访问 T 的根并将根加入队列。
- (2) 若队列非空，取出队首元素 x 。
- (3) 访问 x 的所有儿子并将所有儿子加入队列。
- (4) 重复(2)~(3)，直到队列为空。

图3-23演示了使用宽度优先方式遍历一棵树的执行过程及结果。

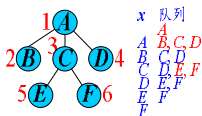
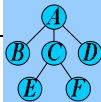


图3-23 树的宽度优先遍历

3.4.2 树遍历算法的C++实现



$A.f=B, A.r=F$
 $E.f=0, E.r=C$
 $C.f=E, C.r=D$
 $D.f=0, D.r=0$
 $E.f=0, E.r=F$
 $F.f=0, F.r=0$

1. 树的C++表示

使用孩子兄弟的表示方法。

```

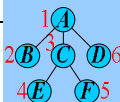
#include "algorithm.h"

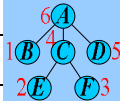
template<class T> // 树结点类
struct TreeNode
{
    T data;
    TreeNode *first = 0, *next = 0; // 第一棵子树, 下一棵树(兄弟)
};
    
```

2. 树的先根次序遍历

```

template<class T, class Func> //
void PreOrder(TreeNode<T> *x, Func Visit)
{
    if(x == 0) return;
    Visit(x); // 访问根
    for(auto w = x->first; w != 0; w = w->next)
        PreOrder(w, Visit); // 依次遍历所有子树
} // 耗时O(n)
    
```





3. 树的后根次序遍历

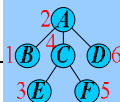
```

template<class T, class Func> //
void PostOrder(TreeNode<T> *x, Func Visit)
{
    if(x == 0) return;
    for(auto w = x->first; w != 0; w = w->next)
        PostOrder(w, Visit); // 依次遍历所有子树
    Visit(x); // 访问根
} // 耗时O(n)
    
```

4. 树的中根次序遍历

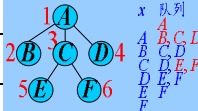
```

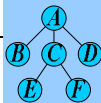
template<class T, class Func> //
void InOrder(TreeNode<T> *x, Func Visit)
{
    if(x == 0) return;
    auto w = x->first; // 第一棵子树
    InOrder(w, Visit); // 遍历第一棵子树
    Visit(x); // 访问根
    if(w == 0) return; // 叶子
    for(w = w->next; w != 0; w = w->next)
        InOrder(w, Visit); // 依次遍历其余子树
} // 耗时O(n)
    
```



5. 树的宽度优先遍历

```
template<class T, class Func> //
void LevelOrder(TreeNode<T> *x, Func Visit)
{
    queue<TreeNode<T>*> Q;
    if(x != 0) Visit(x), Q.push(x); // 访问根并将根加入队列
    while(not Q.empty())
    {
        x = Q.front(), Q.pop();
        // 访问x的所有儿子并将所有儿子加入队列
        for(x = x->first; x != 0; x = x->next)
            Visit(x), Q.push(x);
    }
} // 耗时O(n)
```





A.f=B, A.r=F
 B.f=0, B.r=C
 C.f=E, C.r=D
 D.f=0, D.r=0
 E.f=0, E.r=F
 F.f=0, F.r=0

6. 测试程序^{*}

使用图3-20~图3-23中使用的树。

```

int main()
{
    auto Visit = [](TreeNode<char> *x) // 访问结点*x
    {
        cout << x->data << ' ';
    };

    TreeNode<char> a{'A'}, b{'B'}, c{'C'}, d{'D'}, e{'E'}, f{'F'};
    // 建立一棵以A为根的树
    a.first = &b, b.next = &c, c.first = &e, c.next = &d, e.next = &f;

    cout << "先根次序: ";
    PreOrder(&a, Visit), cout << endl; // A B C E F D
    cout << "中根次序: ";
    InOrder(&a, Visit), cout << endl; // B A E C F D
    cout << "后根次序: ";
    PostOrder(&a, Visit), cout << endl; // B E F C D A
    cout << "宽度优先遍历: ";
    LevelOrder(&a, Visit), cout << endl; // A B C D E F
}
  
```

3.5 合并查找结构^{*}

也称为并查集，是一种树型的数据结构，用于处理一些不相交集合并及查找问题。合并查找结构中的每一个集合用一棵树表示，用根代表该集合。

3.5.1 数据结构

将每个集合包含的所有元素从0开始统一编号，用一个数组记录每个元素的双亲，根的双亲是一个负数，代表这棵树的深度。例如，对于图3-24左侧所示的3个集合，相应的双亲数组如图3-24下侧所示。

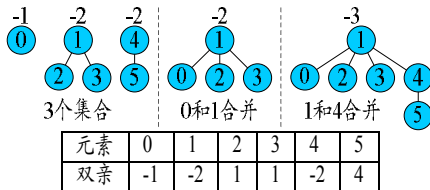


图3-24 合并查找结构

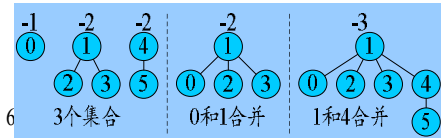
3.5.2 主要操作

1. 初始化

把每个元素作为一个单元素集合，即每个顶点都是根，深度为1，所以每个顶点的双亲均为-1。

2. 查找

查找元素所在的集合，即元素所在树的根。例如，在图3-24左侧所示的3个集合中，元素1、2和3所在的集合都是集合1。



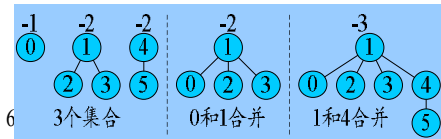
3. 合并

将两个元素所在的集合合并为一个集合。通常，在合并之前，应先判断两个元素是否属于同一集合，这可用查找操作实现。

合并操作可以根据两棵树的深度分为下列两种情况。

(1) 待合并的两棵树深度不相同。将深度较小的那颗树的根作为深度较大的那颗树的根的儿子。例如，图3-24中部所示的合并操作就实现了集合0和1的合并。

(2) 待合并的两棵树深度相同。将其中一颗树的根作为另一颗树的根的儿子。例如，图3-24右侧所示的合并操作就实现了集合1和4的合并。



4. 复杂性分析

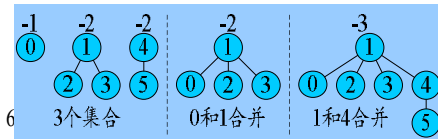
容易看出，初始化耗时 $O(n)$ 。

对于查找与合并操作，设 d 为合并算法产生的有 n 个顶点的树的深度。容易看出，查找与合并操作均耗时 $O(d)$ ，下列引理说明 $d \leq \log n + 1$ ，所以查找与合并操作均耗时 $O(\log n)$ 。

【引理】 若 T 是一棵由合并算法产生的树，且 T 的顶点数为 n ，则 T 的深度 $d \leq \log n + 1$ 。

【证明】 当 $n=1$ 时引理显然成立。假设当 $n < m$ 时引理成立。

当 $n=m$ 时，设 T 是一棵由合并算法产生的树，且 T 的顶点数为 m 。考虑最后一次合并操作。若合并前两棵子树的深度不相同，则 T 的深度与深度较大的子树相同，引理成立。否则， T 的深度 d 比子树的深度 h 多1。对于顶点较少的子树，其顶点数 $v \leq m/2$ ，由归纳假设可知 $h \leq \log v + 1 = \log(2v) \leq \log m$ ，所以 $d = h + 1 \leq \log m + 1$ 。引理成立。



3.5.3 合并查找结构的C++程序

下面给出合并查找结构的C++程序，保存在文件UnionFind.hpp中。

1. 数据成员

```
// UnionFind.hpp, C++ 11
#pragma once
#include <vector>

class UnionFind
{
    std::vector<int> parent; // 双亲数组，负数代表根和深度
```

2. 初始化

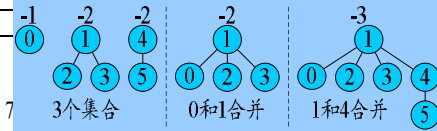
```
public:
    // 初始化，每个集合(类别，树)一个元素，即各元素的双亲值均为-1
    UnionFind(int n = 10): parent(n, -1) {}
```

3. 查找

```
int Find(int e) const // e所在树的根
{
    if(parent[e] < 0) return e; // e是根
    else return Find(parent[e]); // e不是根
}
```

4. 合并

```
void Union(int i, int j) // 合并i和j所在的树
{
    i = Find(i), j = Find(j); // i和j所在树的根
    if(i == j) return; // 无需合并
    if(-parent[i] < -parent[j]) // i深度较小, i成为j的子树
        parent[i] = j;
    else if(-parent[j] < -parent[i]) // j深度较小, j成为i的子树
        parent[j] = i;
    else // j成为i的子树, i深度增加
        parent[j] = i, --parent[i];
};
```



3.6 一般图的遍历

无论是二叉树还是一般树，由于不含圈，所以各个子树之间是相互独立的。因此遍历过程是对各个子树的分别遍历和对根遍历，以及把这些遍历有机地组合起来。一般的图没有这种独立性，所以上述方法不能直接推广到一般的图。但是，上述方法的思想可以借鉴，于是产生了深度优先搜索和宽度优先搜索两种方法。

本节依次介绍连通图的宽度优先搜索、一般图的宽度优先遍历、连通图的深度优先搜索和一般图的深度优先遍历。然后在下一节介绍一个深度优先搜索的应用实例（连通图的双连通分支）。

3.6.1 连通图的宽度优先搜索（BFS）

1. 算法思路

- (1) 准备一个空队列。
- (2) 访问起点 v ，将 v 打上已访问标记后放入队列。
- (3) 取出队列的队首元素 u ，搜索所有与 u 相邻的顶点。如果 w 与 u 相邻且未访问，则访问 w ，将 w 打上已访问标记后放入队列。
- (4) 重复 (3)，直到队列为空。

图3-25演示了BFS的执行过程及结果。

	v	队列	生成树
		A	
	A	B, D, E	AB, AD, AE
	B	D, E, C	BC
	D	E, C	
	E	C	

图3-25 连通图的宽度优先搜索

2. 算法描述

```
#include "algorithm.h"

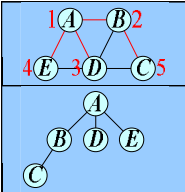
template<class Fun> // 搜索一个连通分支
void BFS(Matrix<bool> &G, int v, vector<bool> &Visited, Fun Visit)
{ // 邻接矩阵, 开始顶点, 访问标记(已经初始化为0), 访问函数
  int n = G.rows();
  queue<int> Q; // 创建空队列
  Visited[v] = 1, Visit(v), Q.push(v); // v标记, 访问, 入队
  while(not Q.empty())
  { v = Q.front(), Q.pop(); // 取出队首元素v
    for(int w = 0; w < n; ++w)
      if(not Visited[w] and G(v, w) == 1) // w与v相邻且未访问
        Visited[w] = 1, Visit(w), Q.push(w); // w标记, 访问, 入队
  }
}
```

3. 复杂性分析

易知，如果连通图 G 有 n 个顶点，用邻接矩阵表示，则BFS在图 G 上的时间复杂性为 $O(n^2)$ ，空间复杂性为 $O(n)$ 。

4. 宽度优先生成树

如果 G 是连通图，则 G 有生成树。注意到BFS算法中，while循环将所有与 u 相邻的未访问顶点 w 添加到队列中。如果在添加 w 的同时将边 (u, w) 收集起来，那么在算法结束时，所有这些边形成了图 G 的一棵生成树，称为图 G 的宽度优先生成树。如图3-25右侧所示。

	v	队列	生成树
		A	
	A	B, D, E	AB, AD, AE
	B	D, E, C	BC
	D	E, C	
	E	C	
	C		

3.6.2 一般图的宽度优先遍历（BFT）

1. 算法描述

对于非连通图，可以通过在每个连通分支中选取一个起点，应用宽度优先搜索遍历该图的所有顶点。如图3-26所示。

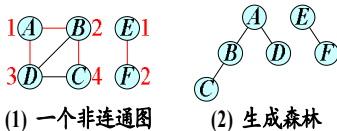
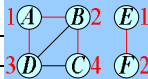
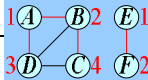


图3-26 一般图的宽度优先遍历

算法描述如下。



```
template<class Fun> // 一般图的宽度优先遍历
void BFT(Matrix<bool> &G, Fun Visit) // 邻接矩阵, 访问函数
{
    int n = G.rows();
    vector<bool> Visited(n, 0); // 访问标记, 初始化为0
    for(int v = 0; v < n; ++v)
        if(not Visited[v]) // 一个新连通分支
            BFS(G, v, Visited, Visit), cout << endl;
} // 耗时O(n^2)
```



2. 测试程序^{*}

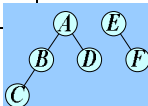
使用图3-26中的图。

```
int main()
{
    auto Visit = [](int v) // 访问函数
    {
        cout << char('A' + v) << '\n';
    };

    Matrix<bool> G =
    {
        {0, 1, 0, 1, 0, 0},
        {1, 0, 1, 1, 0, 0},
        {0, 1, 0, 1, 0, 0},
        {1, 1, 1, 0, 0, 0},
        {0, 0, 0, 0, 0, 1},
        {0, 0, 0, 0, 1, 0}
    };

    BFT(G, Visit); // 一般图的宽度优先遍历
    cout << endl;
}
```

A B D C
E F



3.6.3 连通图的深度优先搜索（DFS）

1. 算法思路

如图3-27所示。从顶点 v 出发，访问 v 并打上已访问标记。以 v 作为当前扩展顶点。先访问 v 的第一个未访问的相邻顶点，将这个顶点打上已访问标记，并取该顶点作为当前扩展顶点。如果当前扩展顶点 u 没有未访问的相邻顶点，则以 u 的前任扩展顶点为当前扩展顶点。这个过程一直持续到没有未访问顶点为止。

	访问	顶点栈	生成树
		A	
	A	A, B	AB
	B	A, B, C	BC
	C	A, B, C, D	CD
	D	A, B, C, D, E	DE
	E	A, B, C, D	
		A, B, C	
		A, B	
		A	

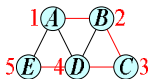


图3-27 连通图的深度优先遍历

2. 算法描述

```
#include "algorithm.h"

template<class Fun> // 搜索一个连通分支
void DFS(Matrix<bool> &G, int v, vector<bool> &Visited, Fun Visit)
{ // 邻接矩阵, 开始顶点, 访问标记(已经初始化为0), 访问函数
  int n = G.rows();
  Visited[v] = 1; // 标记v
  Visit(v); // 访问v(先根顺序)
  for(int w = 0; w < n; ++w)
    if(not Visited[w] and G(v, w) == 1) // w与v相邻且未访问
      DFS(G, w, Visited, Visit); // 访问子树w
  // Visit(v); // 访问v(后根顺序)
}
```

3. 复杂性

易知, 如果连通图 G 有 n 个顶点, 用邻接矩阵表示, 则DFS在图 G 上的时间复杂性为 $O(n^2)$, 空间复杂性为 $O(n)$ 。

4. 深度优先生成树

同样，如果在访问 w 的同时将边 (u, w) 收集起来，那么在算法结束时，所有这些边形成了连通图 G 的一棵生成树，称为图 G 的深度优先生成树。如图3-27右侧所示。

	访问	顶点栈	生成树
		A	
	A	A, B	AB
	B	A, B, C	BC
	C	A, B, C, D	CD
	D	A, B, C, D, E	DE
	E	A, B, C, D	
		A, B, C	
		A, B	
		A	

3.6.4 一般图的深度优先遍历（DFT）

1. 算法描述

如图3-28所示，对于非连通图，可以通过在每个连通分支中选取一个起点，应用深度优先搜索遍历该图的所有顶点。

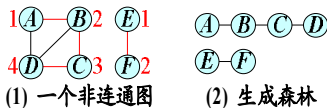
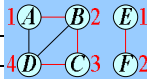
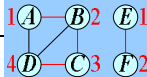


图3-28 一般图的深度优先遍历

算法描述如下。



```
template<class Fun> // 一般图的深度优先遍历
void DFT(Matrix<bool> &G, Fun Visit) // 邻接矩阵, 访问函数
{
    int n = G.rows();
    vector<bool> Visited(n, 0); // 访问标记, 初始化为0
    for(int v = 0; v < n; ++v)
        if(not Visited[v]) // 一个新连通分支
            DFS(G, v, Visited, Visit), cout << endl;
} // 耗时O(n^2)
```



2. 测试程序^{*}

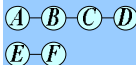
使用图3-28中的图。

```
int main()
{
    auto Visit = [](int v) // 访问函数
    {
        cout << char('A' + v) << '\n';
    };

    Matrix<bool> G =
    {
        {0, 1, 0, 1, 0, 0},
        {1, 0, 1, 1, 0, 0},
        {0, 1, 0, 1, 0, 0},
        {1, 1, 1, 0, 0, 0},
        {0, 0, 0, 0, 0, 1},
        {0, 0, 0, 0, 1, 0}
    };

    DFT(G, Visit); // 一般图的深度优先遍历
    cout << endl;
}
```

A B C D
E F



3.7 连通图的双连通分支^{*}

通信网络的抽象模型是一个连通图，其中，顶点代表通信站，边代表通信线路。图3-29和图3-30代表两个通信网络。

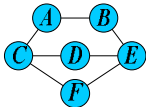


图3-29 双连通图

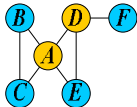


图3-30 非双连通图

容易看出，图3-29所示的通信网络可靠性较高，因为即使有一个网站出现问题，其他网站之间的通信仍然可以继续。而图3-30所示的网络则不能，只要网站A发生故障，A左右两侧的网站之间就无法连通了。

3.7.1 有关定义

【定义】如果在连通图 G 中去掉某个顶点 v 及其关联的边，剩下的图不再连通，则称 v 为 G 的割点（articulation point，也称为关键点和关节点等）。没有割点的连通图称为双连通的（biconnected graph，也称为重连通、2-连通和块等）。连通图 G 中极大的双连通子图称为 G 的一个双连通分支。

在图3-30中，除了 A 以外， D 也是割点。这个连通图有3个双连通分支（如图3-31所示）。图3-29中的图是双连通的。

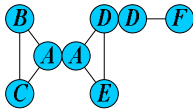


图3-31 图3-30的3个双连通分支

就通信网络而言，当然希望没有割点。如果现有网络存在割点，则设法增加一些线路（尽量少增加），使它成为双连通的。

3.7.2 添加边的算法

对每个割点 u

{ 设 B_1, B_2, \dots, B_k 是包含割点 u 的全部双连通分支;

在每个分支 B_i 中任意找一顶点 V_i , 使 $V_i \neq u, 1 \leq i \leq k$;

将 (V_i, V_{i+1}) 添加到 G , $1 \leq i < k$;

}

图3-32演示了添加边的算法。

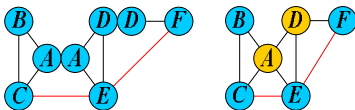


图3-32 添加边的算法

3.7.3 深度优先树

1. 割点识别问题

【问题】设计一个算法测试一个连通图是否双连通，若不是，则设法找出割点。

【解决方法】在深度优先遍历算法中加入割点识别步骤。

为了方便描述，首先给出下述几点规定。

(1) 深度优先树。由深度优先遍历算法获得的深度优先生成树。

(2) 树边和余边。原图中已选进深度优先树的边称为树边，未选进深度优先树的边称为余边。

(3) 顶点的层次。原图中的顶点 v 在深度优先树中的层次记做 $\text{Level}(v)$ 。

(4) 双亲和祖先。在深度优先树中，规定根的双亲为根自己，一个顶点的祖先不包含该顶点的双亲。

图3-33给出了图3-30的深度优先树及余边。

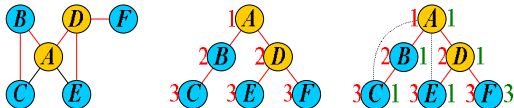
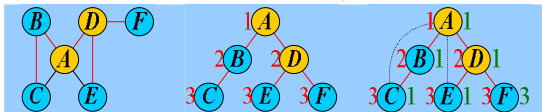


图3-33 图3-30的深度优先树及余边

2. 深度优先树的性质

- (1) 连通图 G 中每一条边的两个端点在深度优先树中不可能是兄弟关系。
- (2) 若顶点 u 的双亲是 v ，则原图中的边 (u, w) 是深度优先树的余边当且仅当 w 是 u 的祖先，即 $\text{Level}(w) < \text{Level}(v)$ 。
- (3) 深度优先树 T 的根是连通图 G 的割点当且仅当它在 T 中至少有两个儿子。
- (4) 深度优先树 T 的非根顶点 u 是连通图 G 的割点当且仅当 u （在 T 中）至少有一棵子树的任何顶点都不与 u 的任何祖先（在 G 中）相邻。



3. 最小层次

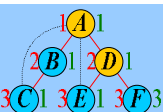
为了方便判断一个顶点是否是割点, 定义顶点 u 的最小层次为

$$L(u) = \min \left\{ \begin{array}{l} \text{Level}(u), \\ \min\{L(w) \mid w \text{ 是 } u \text{ 的儿子}\}, \\ \min\{\text{Level}(w) \mid (u, w) \text{ 是 } T \text{ 的余边}\} \end{array} \right\}$$

如果 $L(u) \neq \text{Level}(u)$, 则 $L(u)$ 是 u 通过一条子孙路径且至多后跟一条 T 的余边所可能到达的顶点的最小层次。

由深度优先树的性质可知, 顶点 u 是连通图的割点当且仅当 u 至少有一个儿子 w 满足 $L(w) \geq \text{Level}(u)$ 。

要想识别连通图 G 的所有割点, 需先获得深度优先树 T , 然后按后根优先次序遍历 T 计算出各个顶点的最小层次。从函数 L 的定义可知, 这两步工作可以同时完成。



深度优先树 T 的非根顶点 u 是 G 的割点当且仅当 u (在 T 中) 至少有一棵子树的任何顶点都不与 u 的任何祖先 (在 G 中) 相邻。

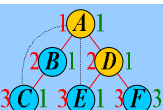
4. 最小层次的计算方法

(1) 变量的声明及初始化。

```
#include "Algorithm.h"
auto LowLevel(const Matrix<bool> &G)
{ // G是图的邻接矩阵
  int n = G.rows(); // 顶点数
  vector<int> Level(n, 0); // 记录各顶点的层次，初始为0
  vector<int> L(n); // 记录各顶点的最小层次
```

(2) 深度优先遍历的描述。

```
function<int(int, int)>
LowLevel = [&](int u, int v) // v是深度优先树中u的双亲
{
    Level[u] = Level[v] + 1; // u的层次
    L[u] = Level[u]; // u的最小层次
    for(int w = 0; w < n; ++w)
    {
        if(G(u, w) == 0) continue; // 对u的每个邻接顶点w
        if(Level[w] == 0) // w未访问(w是u的儿子), (u, w)是树边
        {
            L[w] = LowLevel(w, u); // 计算w的最小层次
            L[u] = min(L[u], L[w]); // 更新L[u]
        }
        else if(Level[w] < Level[v]) // w是u的祖先, (u, w)是余边
            L[u] = min(L[u], Level[w]); // 更新L[u]
    }
    return L[u];
};
```



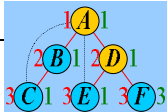
$$L(u) = \min \left\{ \begin{array}{l} \text{Level}(u), \\ \min\{L(w) \mid w \text{ 是 } u \text{ 的儿子}\}, \\ \min\{\text{Level}(w) \mid (u, w) \text{ 是 } T \text{ 的余边}\} \end{array} \right\}$$

(3) 深度优先遍历的调用。

```
LowLevel(0, 0); // 从0开始访问, 根的双亲是自己  
return pair{Level, L}; // 成组的返回值(优先数, 最小优先数)  
} // 耗时 $O(n^2)$ 
```

(4) 复杂性分析。如果 G 是有 n 个顶点 e 条边的连通图，用邻接矩阵表示，那么计算最小层次的时间为 $O(n^2)$ 。这是因为该算法与连通图的深度优先搜索相比，就相当于指定了一个常数时间的访问函数。

```
function<void(int, int)>  
DFNL = [&](int u, int v)  
{  
    L[u] = DFN[u] = k, ++k;  
    for(int w = 0; w < n; ++w)  
    {  
        if(G(u, w) == 0) continue;  
        if(DFN[w] == 0)  
        {  
            DFNL(w, u);  
            L[u] = min(L[u], L[w]);  
        }  
        else if(DFN[w] < DFN[v])  
            L[u] = min(L[u], DFN[w]);  
    }  
};
```



3.7.4 识别割点和生成双连通分支

为了得到连通图 G 的双连通分支，需对LowLevel作一些修改。

注意到，在递归调用LowLevel时，如果出现 $L(w) \geq \text{Level}(u)$ ，就可以断定 u 要么是 T 的根，要么是连通图 G 的割点。不论哪种情况，都将边 (u, w) 和对LowLevel这次调用期间遇到的所有树边和余边加在一起，就可以构成连通图 G 的一个双连通分支。

如果引入一个存放边的栈St，将边 (u, w) 和对LowLevel这次调用期间遇到的所有树边和余边都放到栈中，则从栈顶到 (u, w) 的所有边就可以构成连通图 G 的一个双连通分支。

1. 边和双连通分支的表示

(1) 边的表示。

```
struct Edge { int u, v; }; // 用顶点对表示边
// 比较两无向边是否相等
bool operator==(const Edge &e, const Edge &f)
{   return e.u == f.u and e.v == f.v or e.u == f.v and e.v == f.u;
}
```

(2) 双连通分支的表示。

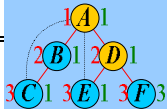
```
// 每个双连通分支定义为一个边子集
typedef vector<Edge> BiConnected_t;
```

2. 算法描述

(1) 变量的声明及初始化。

```
auto BiConnected(const Matrix<bool> &G)
{ // G是图的邻接矩阵
  int n = G.rows(); // 顶点数
  vector<int> Level(n, 0); // 记录各顶点的层次, 初始为0
  vector<int> L(n); // 记录各顶点的最小层次
  stack<Edge> St; // 用于存放边(树边和余边)
  vector<BiConnected_t> Components; // 双连通分支数组
```

(2) 深度优先遍历的描述。



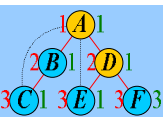
```
function<int(int, int)>
LowLevel = [&](int u, int v) // v是深度优先树中u的双亲
{
    Level[u] = Level[v] + 1; // u的层次
    L[u] = Level[u]; // u的最小层次
    for(int w = 0; w < n; ++w)
    {
        if(G(u, w) == 0) continue; // 对u的每个邻接顶点w
        if(Level[w] == 0) // w未访问(w是u的儿子), (u, w)是树边
        {
            St.push({u, w}); // 树边入栈
            L[w] = LowLevel(w, u); // 计算w的最小层次
            if(L[w] >= Level[u]) // 有新的双连通分支
            {
                BiConnected t C; // 新分支初始为空集
                for(;;)
                {
                    auto e = St.top(); St.pop(); // 取出新分支的当前边
                    C.push back(e); // 在新分支中加入当前边
                    if(e == Edge{u, w}) break; // 直到(u, w)
                }
                Components.push back(C); // 新分支加入到分支数组
            }
            L[u] = min(L[u], L[w]); // 更新L[u]
        }
    }
}
```

$$L(u) = \min \begin{cases} \text{Level}(u), \\ \min\{L(w) \mid w \text{ 是 } u \text{ 的儿子}\}, \\ \min\{\text{Level}(w) \mid (u, w) \text{ 是 } T \text{ 的余边}\} \end{cases}$$

```
else if(Level[w] < Level[v]) // w是u的祖先, (u, w)是余边
{ St.push({u, w}); // 余边入栈
  L[u] = min(L[u], Level[w]); // 更新L[u]
}
}
return L[u];
};
```

(3) 深度优先遍历的调用。

```
LowLevel(0, 0); // 从0开始访问, 根的双亲是自己
return Components;
} // 耗时O(n^2)
```



$$L(u) = \min \left\{ \begin{array}{l} \text{Level}(u), \\ \min\{L(w) \mid w \text{ 是 } u \text{ 的儿子}\}, \\ \min\{\text{Level}(w) \mid (u, w) \text{ 是 } T \text{ 的余边}\} \end{array} \right\}$$

3. 复杂性分析

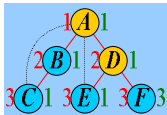
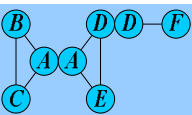
如果 G 是有 n 个顶点 e 条边的连通图，用邻接矩阵表示，那么计算最小层次的时间为 $O(n^2)$ 。一旦计算出所有顶点的最小层次， G 的一个顶点是否是割点就能在 $O(n)$ 内时间识别出来（执行 $O(n)$ 次“if($L[w] \geq \text{Level}[u]$)”），因此，识别全部割点的总时间为 $O(n^2)$ 。生成全部双连通分支的时间为 $O(e)$ （每条边的进栈和出栈等操作都只有一次），因为 $e \leq n^2$ ，所以BiConnected的计算时间为 $O(n^2)$ 。

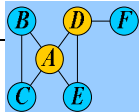
4. 实例分析

对于图3-30所示的连通图，执行过程和结果如表3-1所示。

表3-1 双连通分支实例分析

顶点	层次	邻点	顶点栈	边栈	双连通分支	最小层次
			A			
A	1	B	AB	AB		
B	2	C	ABC	AB, BC		
C	3	A	AB	AB, BC, CA		$L(C)=1$
B			A		CA, BC, AB	$L(B)=1 \geq \text{Level}(A)$
A		D	AD	AD		
D	2	E	ADE	AD, DE		
E	3	A	AD	AD, DE, EA		$L(E)=1$
D	3	F	ADF	AD, DE, EA, DF		
F			A	AD, DE, EA	DF	$L(F)=3 \geq \text{Level}(D)$
D			A		EA, DE, AD	$L(D)=1 \geq \text{Level}(A)$
A						$L(A)=1$





5. 测试程序^{*}

使用4-30所示的图。

```

// 用大写字母的方式输出一边
ostream &operator<<(ostream &out, const Edge &e)
{
    return out << (char)(e.u + 'A') << (char)(e.v + 'A');
}

int main()
{
    Matrix<bool> G = // 邻接矩阵
    {
        {0, 1, 1, 1, 1, 0},
        {1, 0, 1, 0, 0, 0},
        {1, 1, 0, 0, 0, 0},
        {1, 0, 0, 0, 1, 1},
        {1, 0, 0, 1, 0, 0},
        {0, 0, 0, 1, 0, 0},
    };
    auto Components = BiConnected(G);
    cout << to_string(Components, Components.size(), "\t") << endl;
}

```

CA BC AB

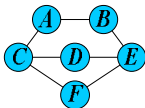
DF

EA DE AD

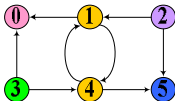
3.8 练习题

3.8.1 基础训练题

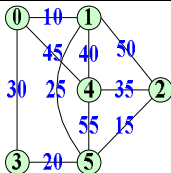
1. 请写出下列无向图的邻接矩阵。



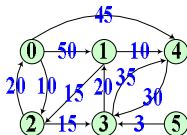
2. 请写出下列有向图的邻接矩阵。



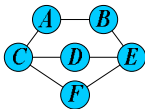
3. 请写出下列无向加权图的邻接矩阵。



4. 请写出下列有向加权图的邻接矩阵。



5. 使用宽度优先搜索方法遍历下列无向图，请写出搜索过程中访问队列中的元素。



6. 在合并查找结构中，若 T 是一棵由合并算法产生的树，且 T 的顶点数为 n ，则 T 的深度 $d \leq \log n + 1$ 。请证明该结论。
7. 请使用一种程序设计语言描述二叉树的4种遍历方法。
8. 请使用一种程序设计语言描述一般树的4种遍历方法（树使用儿子兄弟的方法表示）。
9. 请使用一种程序设计语言描述连通图的宽度优先搜索算法（图使用邻接矩阵表示）。
10. 请使用一种程序设计语言描述连通图的深度优先搜索算法（图使用邻接矩阵表示）。
11. 请使用一种程序设计语言描述能生成连通图的全部双连通分支的深度优先搜索算法（图使用邻接矩阵表示）。
12. 假设用一个 $n \times n$ 的数组来描述一个有向图的邻接矩阵，完成下列工作。
 - (1) 编写一个函数以确定某个顶点的出度，函数的复杂性应为 $\Theta(n)$ 。
 - (2) 编写一个函数以确定图中边的数目，函数的复杂性应为 $\Theta(n^2)$ 。
 - (3) 编写一个删除边 (i, j) 的函数，并确定代码的复杂性。

3.8.2 实验题

1. 请使用一种程序设计语言描述合并查找结构的查找操作和合并操作(合并查找结构使用整数数组表示, 每个元素的初始值均为-1)。
2. 请使用一种程序设计语言改写二叉树的一种深度优先遍历算法, 要求能够计算出每个顶点的深度。
3. 请使用一种程序设计语言改写一般树的宽度优先遍历算法, 要求能够计算出每个顶点的层次。
4. 请使用一种程序设计语言改写图的宽度优先遍历算法(图使用邻接矩阵表示), 要求能够输出图的每个连通分支(输出连通分支包含的顶点)。
5. 请使用一种程序设计语言改写图的深度优先遍历算法(图使用邻接矩阵表示), 要求能够输出图的每个连通分支(输出连通分支包含的顶点)。
6. 请使用一种程序设计语言改写连通图的深度优先遍历算法, 要求能够计算出每个顶点在深度优先生成树中的深度(图使用邻接矩阵表示)。
7. 请使用一种程序设计语言改写连通图的宽度优先遍历算法, 要求能够计算出每个顶点在深度优先生成树中的层次(图使用邻接矩阵表示)。

3.8.3 进阶训练题

1. 请设计三个算法，分别用于判断一棵二叉树是否是满二叉树、平衡二叉树（不在层次最大的两层上的每个顶点都有两个儿子）和完全二叉树。
2. 参照图的宽度优先遍历算法、深度优先遍历算法以及求连通图的双连通分支的算法，为有向图设计一个输出所有双向连通分支（输出双向连通分支包含的顶点）的算法。
3. 请使用一种程序设计语言给出合并查找结构的一种实现方法，要求至少给出查找操作和合并操作的实现，且查找操作和合并操作的时间复杂度分别为 $O(1)$ 和 $O(n)$ ，其中 n 是合并查找结构的元素数。