## 附录C 图论基础

*He offered a bet that we could name any person among earth’s one and a half billion inhabitants and through at most five acquaintances, one of which he knew personally, he could link to the chosen one.*

— Frigyes Karinthy, *Láncszemek*[[1]](#footnote-1)

*他打赌说：在地球上15亿茫茫人海中随意选出一位，只要通过至多五名中间人，就能和这位随意选出的陌生人产生联系。*

— Frigyes Karinthy, *Láncszemek*

The following presentation is loosely based on the first chapters of *Graph Theory* by Reinhard Diestel and *Digraphs* by Bang-Jensen and Gutin, and on the appendixes of *Introduction to Algorithms* by Cormen et al. (Note that the terminology and notation may differ between books; it is not completely standardized.) If you think it seems like there’s a lot to remember and understand, you probably needn’t worry. Yes, there may be many new words ahead, but most of the concepts are intuitive and straightforward, and their names usually make sense, making them easier to remember.

本附录的内容与以下章节有一定关联：Reinhard Diestel编写的第一章《图论》；Bang-Jensen和Gutin合著的第二章《*有向图*》以及Cormen et al的《算法介绍》的附录部分/

So … A *graph* is an abstract network, consisting of *nodes* (or *vertices*), connected by *edges* (or *arcs*). More formally, we define a graph as a pair of sets, *G* = (*V*, *E*), where the node set *V* is any finite set, and the edge set *E* is a set of (unordered) node pairs.[[2]](#footnote-2) We call this a graph *on V*. We sometimes also write *V*(*G*) and *E*(*G*), to indicate which graph the sets belong to.[[3]](#footnote-3) Graphs are usually illustrated with network drawings, like those in Figure C-1 (just ignore the gray

highlighting for now). For example, the graph called *G*1 in Figure C-1 can be represented by the node set *V* = {*a*,*b*,*c*,*d*,*e*,*f* } and the edge set *E* = {{*a*,*b*},{*b*,*c*},{*b*,*d*},{*d*,*e*}}.

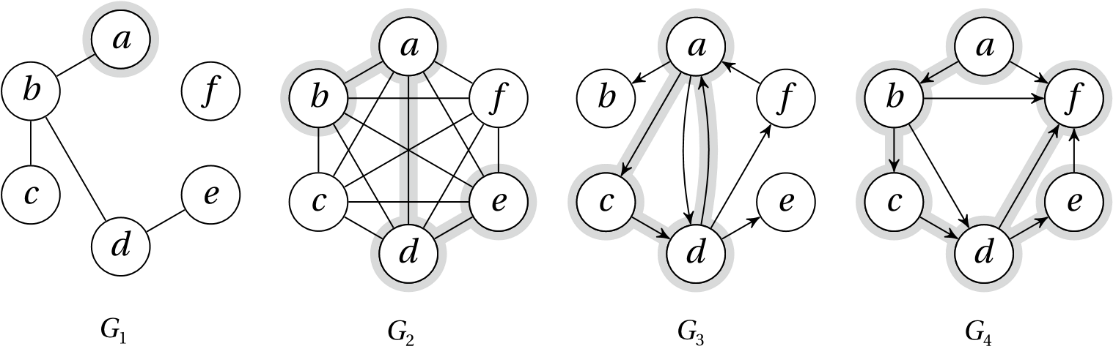
我们可以得出以下结论：图就是一种抽象网络，由节点（或顶点）组成，通过边（或者弧线）彼此连接。更规范地说，我们可以将图定义成一对集合*G* = (*V*, *E*)，节点集合V可以是任何有限集合，而边集E则是由（无序的）节点对组成的。我们将它称作图V。有时也写作*V*(*G*) 和*E*(*G*)来表明这个结合属于哪个集合。图通常由类似C-1（暂时忽略灰色高亮部分）这样的网络结构来表示。C-1中的图称为*G*1 ，可以表示成节点集*V* = {*a*,*b*,*c*,*d*,*e*,*f* }以及边集*E* = {{*a*,*b*},{*b*,*c*},{*b*,*d*},{*d*,*e*}}。

You don’t always have to be totally strict about distinguishing between the graph and its node and edge sets. For example, we might talk about a node *u* in a graph *G*, really meaning in *V*(*G*), or equivalently, an edge {*u*,*v*} in *G*, meaning in *E*(*G*).

你并不需要总是严格分辨图和它的节点及边集之间的区别。比如，当我们提到图G中的节点*u时，我们真正想表达的是V*(*G*)中的节点，类似的，图G中的边 {*u*,*v*} 实际上指的是 *E*(*G*)中的边。

■ **Note** It is common to use the sets *V* and *E* directly in asymptotic expressions, such as Q(*V* +*E* ), to indicate linearity in the graph size. In these cases, the sets should be interpreted as their *cardinalities* (that is, sizes), and a more correct expression would be Q(|*V* | + |*E* |), where | · | is the cardinality operator.

■ **注意** 直接在渐进表达式中使用*V* 和*E是很常见的，比如*Q(*V* +*E* )，这表示图的线性长度。在这种情况下，集合可以通过它们的模来表示（即长度），更确切的表达式应该是Q(|*V* | + |*E* |)，| · |表示取模运算。



***Figure C-1.*** *Various types of graphs and digraphs*

***图 C-1.****各式各样的图和有向图*

The basic graph definition gives us what is often called an *undirected* graph, which has a close relative: the *directed* graph, or *digraph*. The only difference is that the edges are no longer *unordered* pairs but *ordered*: An edge between nodes *u* and *v* is now either an edge (*u*,*v*) from *u* to *v* or an edge (*v*,*u*) from *v* to *u*. In other words, in a digraph *G*, *E*(*G*) is a relation over *V*(*G*). The graphs *G*3 and *G*4 in Figure C-1 are digraphs where the edge directions are indicated by arrowheads. Note that *G*3 has what is called *antiparallel* edges between *a* and *d*, that is, edges going both ways. This is OK because (*a*,*d*) and (*d*,*a*) are different. Parallel edges, though (that is, the same edge, repeated) are not allowed—neither in graphs nor digraphs. (This follows from the fact that the edges form a set.) Note also that an undirected graph cannot have an edge between a node and itself, and even though this is possible in a digraph (so-called *self-loops*), the convention is to disallow it.

最基本的图的定义通常被称为“无向图”，与之接近的是：“有向图”。它们两者唯一的区别在于有向图中组成边的点是由指向性的：连接节点*u* 和 *v的边要么是从u* 指向 *v* 的边(*u*,*v*)，要么是从*v* 指向 *u的边*(*v*,*u*)。也就是说，在有向图*G中*, *E*(*G*) 中的边是*V*(*G*)中节点的有序对。C-1中的图*G*3和*G*4 都是有向图，边的方向通过箭头来表示。请注意*G*3 包含了连接*a* 和*d* 的反向线，即两条方向相反的线。在是允许的，因为(*a*,*d*) 和 (*d*,*a*) 是完全不同的两条线。而平行边（同样的边），则无论在有向图还是无向图中都是不允许的。 (这遵循了边是一个集合这条准则) 。另外，无相图中的边不允许连接节点和它自己本身。这一点在有向图中则是允许的（称为自闭环）。至于原因？规则定义如此。

■ **Note** There are other relatives of the humble graph that *do* permit such things as parallel edges and self-loops. If we construct our network structure so that we can have multiple edges (that is, the edges now form a *multiset* ), and self-loops, we call it a (possibly directed) *pseudograph*. A pseudograph without self-loops is simply a *multigraph*. There are also more exotic versions, such as the *hypergraph*, where each edge can have multiple nodes.

■ **注意** 有些图的要求不那么严格，允许诸如平行边以及自闭环之类的概念。如果我们允许网络图结构中有相同的边（也就是说边集现在成为了允许重复的多重集合）以及自闭环，那么我们就可以把它称作伪图 (也可能是有向的)*。没有自闭环的伪图*可以简化为多重图。还有其它许多稀奇古怪的版本，比如“超图”，每条边都能有超过两个的节点。

Even though graphs and digraphs are quite different beasts, many of the principles and algorithms we deal with work just as well on either kind. Therefore, it is common to sometimes use the term *graph* in a more general sense, covering both directed and undirected graphs. Note also that in many contexts (such as when *traversing* or “moving around in” a graph), an undirected graph can be simulated by a directed one, by replacing each undirected edge with a pair of antiparallel directed edges. This is often done when actually implementing graphs as data structures (discussed in more detail in Chapter 2). If it is clear whether an edge is directed or undirected or if it doesn’t matter much, I’ll sometimes write *uv* instead of {*u*,*v*} or (*u*,*v*).

虽然无向图和有向图是截然不同的两种事物，但它们的许多原理和算法是相通的。也就是说，在大部分场合中，“图”这个概念是非常宽泛的，既包括了无向图，也包括了有向图。值得注意的是：在许多上下文中（比如说遍历一张图），通过将每条无向边替换成一组反向边，无向图可以用有向图来模拟。通过数据结构来实现图结构时，这样的做法是很常见的（第二章中将深入探讨这方面的问题）。当我们非常清楚边是否有向时，或者有向与否无关紧要时，我们也可以把{*u*,*v*} or (*u*,*v*)写作*uv。*

An edge is *incident on* its two nodes, called its *end nodes*. That is, *uv* is incident on *u* and *v*. If the edge is directed, we say that it *leaves* (or is *incident from*) *u* and that it *enters* (or is *incident to*) *v*. We call *u* and *v* its *tail* and *head*, respectively. If there is an edge *uv* in an undirected graph, the nodes *u* and *v* are *adjacent* and are called *neighbors*. The set of neighbors of a node *v*, also known as the *neighborhood* of *v*, is sometimes written as *N*(*v*). For example, the neighborhood *N*(*b*) of *b* in *G*1 is {*a*,*c*,*d*}. If all nodes are pairwise adjacent, the graph is called *complete* (see *G*2 in Figure C-1). For a directed graph, the edge *uv* means that *v* is *adjacent to u*, but the converse is true only if we also have an antiparallel edge *vu*. (In other words, the nodes adjacent to *u* are those we can “reach” from *u* by following the edges incident from it in the right direction.)

边连接着它的两个节点，这两个节点也称为它的端点。也就是说，*uv连接u* 和*v。如果某条边是有向的，那么我们就称该边从u* 出发，指向 *v。u和v分别称为此边的尾和头。如果uv是无向的，那么u和v就是相邻的，彼此互为相邻节点。所有v的相邻节点组成的集合，称为v的邻点集，记为N(v)。比如说，图G1中b节点的邻点集合N(b)就是{a,c,d}。如果任意两个节点都互为邻点，那么这张图就称为完全图（C-1中的G2就符合这样的条件）。对于有向图而言，边uv表示v为u的邻点，但反过来未必成立，当且仅当它的反向边vu也存在时我们才能说u是v的邻点。 (也就是说，u的邻点必须满足从u出发存在一条有向边能够指向这些节点)*

The number of (undirected) edges incident on a node *v* (that is, the size of *N*(*v*)) is called its *degree*, often written *d*(*v*). For example, in *G*1 (Figure C-1), the node *b* has a degree of 3, while *f* has a degree of 0. (Zero-degree nodes are called *isolated*.) For directed graphs we can split this number into the *in-degree* (the number of incoming edges) and *out-degree* (the number of outgoing edges). We can also partition the neighborhood of a node into an *in*-*neighborhood*, sometimes called *parents*, and an *out-neighborhood*, or *children*.

节点*v的（无向）边数（也就是N*(*v*)中的元素数量*），称为v的度，通常写作d*(*v*)。以C-1中的*G*1 *为例，节点b的度数为3，而f的度数为0（度数为0的节点也称作孤点）。对于有向图而言，度数就得分为入度（指向该点的边）以及出度（从该点出发的边）。推而广之，我们也能把邻点分为入邻点，即父节点，以及出邻点，即子节点。*

One graph can be *part of* another. We say that a graph *H* = (*W*,*F*) is a *subgraph* of *G* = (*V*, *E*) or, conversely, that *G* is a *supergraph* of *H*, if *W* is a subset of *V* and *F* is a subset of *E*. That is, we can get *H* from *G* by (maybe) removing some nodes and edges. In Figure C-1, the highlighted nodes and edges indicate some example subgraphs that will be discussed in more detail in the following. If *H* is a subgraph of *G*, we often say that *G contains H*. We say that *H spans G* if *W* = *V*. That is, a *spanning* subgraph is one that covers all the nodes of the original graph (such as the one in graph *G*4 in Figure C-1).

图也可以包含于另一张图。当*W* 为*V* 的子集并且 *F* 是 *E的子集时*.我们将图*H* = (*W*,*F*)称为图*G* = (*V*, *E*)的子图或者反过来G是H的超图。也就是说，我们可以通过删除图G中的某些节点和边来得到H。在图C-1中，高亮显示的节点和边预示着我们即将讨论的子图概念。如果H是G的子图，我们通常称为G包含H。如果*W* =*V，那么H和G就是共轭的。也就是说，某张图的共轭图包含了该图的所有节点（比如C-1中的图G*4*）。*

Paths are a special kind of graphs that are primarily of interest when they occur as subgraphs. A *path* is often identified by an sequence of (distinct) nodes, such as *v*1, *v*2, … , *vn*, with edges (only) between pairs of successive nodes: *E* = {*v*1*v*2, *v*2*v*3, … , *vn*–1*vn*}. Note that in a directed graph, a path has to follow the directions of the edges; that is, all the edges in a path point forward. The *length* of a path is simply its edge count. We say that this is a path between *v*1 to *vn* (or, in the directed case, from *v*1 to *vn*). In the sample graph *G*2, the highlighted subgraph is a path between *b* and *e*, for example, of length 3. If a path *P*1 is a subgraph of another path *P*2, we say that *P*1 is a *subpath* of *P*2. For example, the paths *b*, *a*, *d* and *a*, *d*, *e* in *G*2 are both subpaths of *b*, *a*, *d*, *e*.

路径图是一种特殊的图，它主要以子图的形式出现。路径图通常由一系列互不重复的节点，比如*v*1, *v*2, … , *vn*组成，它的边连接着一组组连续的节点：*E* = {*v*1*v*2, *v*2*v*3, … , *vn*–1*vn*}。请注意，在有向图中，路径图必须跟随边的方向；也就是说，有向图中的每条边都必须标明方向。路径的长度正是边的数量。我们将它称为*v*1 至 *vn*之间的路径（如果是有向图则为从*v*1 指向*vn*的路径）.G2中高亮部分子图就是*b* 和 *e之间的路径，长度为3.如果路径P*1 是另一条路径 *P*2的子图，那么我们就称*P*1 为 *P*2的子路径。比如，G2中的路径*b*, *a*, *d* 和路径 *a*, *d*, *e* 都是路径*b*, *a*, *d*, *e*的子路径。

A close relative of the path is the *cycle*. A cycle is constructed by connecting the last node of a path to the first, as illustrated by the (directed) cycle through *a*, *b*, and *c* in *G*3 (Figure C-1). The length of a cycle is also the number of edges it contains. Just like paths, cycles must follow the directions of the edges.

与路径相似的概念是环。在路径的终点上添加一条指向起点的边就能构成环了（C-1的图*G*3中*a*, *b*, *c就构成了一个有向环*）。环的长度同样为它所包含的边数。和路径相似，环也必须跟随边的指向。

■ **Note** These definitions do not allow paths to cross themselves, that is, to contain cycles as subgraphs. A more general path-like notion, often called a *walk*, is simply an alternating sequence of nodes and edges (that is, not a graph in itself), which would allow nodes and edges to be visited multiple times and, in particular, would permit us to “walk in cycles.” The equivalent to a cycle is a *closed walk*, which starts and ends on the same node. To distinguish a path without cycles from a general walk, the term *simple path* is sometimes used.

■ **注意** 根据定义，路径不能穿越自身，也就是说，路径中不能包含带有环的子图。有一个类似于路径而又更宽泛的概念——称作路，它由节点和边交替而成（也就是说它本身并不是一张图），其中每个节点和每条边都能被多次经过，允许“环绕”。环实际上和封闭的路是相同的，其首尾节点重合。为了将无环的路径与通常的路相区分，又是也会用“简单路径”这样的术语。

A common generalization of the concepts discussed so far is the introduction of *edge weights* (or *costs* or *lengths*). Each edge *e* = *uv* is assigned a real number, *w*(*e*), sometimes written *w*(*u*,*v*), usually representing some form of cost associated with the edge. For example, if the nodes are geographic locations, the weights may represent driving distances in a road network. The weight *w*(*G*) of a graph *G* is simply the sum of *w*(*e*) for all edges *e* in *G*. We can then generalize the concept of path and cycle length to *w*(*P*) and *w*(*C*) for a path *P* and cycle *C*, respectively. The original definitions correspond to the case where each edge has a weight of 1. The *distance* between two nodes is the length of the shortest path between them. (Finding such shortest paths is dealt with extensively in the book, primarily in Chapter 9.)

关于边的权重（或者称为开销，长度等），也是目前经常讨论的概念。每一条边*e* = *uv都有与之对应的w*(*e*)，有时也写作*w*(*u*,*v*)，表示这条边所对应的某些开销。比如说，当节点代表某些物理地点时，权重就是路网中的开车距离。整个图G的权重*w*(*G*)则为每一条边*e的权重w*(*e*)之和。推而广之，如果将这个概念应用到路径和环上，那么路径*P* 的权重就是*w*(*P*)，环C的权重就是*w*(*C*) 。两个节点之间的距离指的是连接这两点最短路径的长度。（寻找最短路径的方法将在本书第9章中介绍）。

A graph is *connected* if it contains a path between every pair of nodes. We say that a digraph is connected if the so-called *underlying undirected graph* (that is, the graph that results if we ignore edge directions) is connected. In Figure C-1, the only graph that is not connected is *G*1. The maximal subgraphs of a graph that *are* connected are called its *connected components*. In Figure C-1, *G*1 has two connected components, while the others have only one (each), because the graphs themselves are connected.

如果一张图中任意两点之间都存在路径，那么它就是连通的。如果一张有向图忽略了边的方向之后是连通的，那么我们就称之为基础的无向连通图。C-1中只有图*G*1是唯一的非连通图。图中最大的连通子图称为它的连通分支。C-1中的图*G*1有两个连通分支，而其它图都只有一个连通分支，因为这些图本身就是连通的。

■ **Note** The term *maximal*, as it is used here, means that something cannot be extended and still have a given property. For example, a connected component is maximal in the sense that it is not a subgraph of a *larger* graph (one with more nodes or edges) that is also connected.

■ **注意** 术语“最大的”指的是在仍然满足给定特性的情况下，无法再进行扩展了。比如说，连通分支不能成为任何更大的连通图（包含更多节点或边）的子图。

One family of graphs in particular is given a lot of attention, in computer science and elsewhere: graphs that do not contain cycles, or *acyclic graphs*. Acyclic graphs come in both directed and undirected variants, and these two versions have rather different properties. Let’s focus on the undirected kind first.

在计算机科学以及许多其它领域中，有一种图是深受关注的：不包含环的图，也称为无环图。无环图既可以是有向的也可以是无向的，这两种图大相径庭。让我们先来看看无向的。

Another term for an undirected, acyclic graph is *forest*, and the connected components of a forest are called *trees*.

无向的无环图也称为森林，它的连通分支则称为树。

In other words, a tree is a connected forest (that is, a forest consisting of a single connected component). For example, *G*1 is a forest with two trees. In a tree, nodes with a degree of one are called *leaves* (or *external nodes*),[[4]](#footnote-4) while all others are called *internal* nodes. The larger tree in *G*1, for example, has three leaves and two internal nodes. The smaller tree consists of only an internal node, although talking about leaves and internal nodes may not make much sense with fewer than three nodes.

换言之，一棵树就是连通的森林（即：由单个连通分支组成的森林）。比如，*G*1就是包含了两棵树的森林。在一棵树中，度数为1的节点称为叶节点（或者外部节点），而其它的就称为内部节点。*G*1两棵树中较大的那棵有三个叶节点和两个内部节点。较小的那棵则只有一个内部节点，事实上这棵树只有三个以下节点，讨论它的叶节点和内部节点实在没有太大意义。

■ **Note** Graphs with 0 or 1 nodes are called *trivial* and tend to make definitions trickier than necessary. In many cases, we simply ignore these cases, but sometimes it may be important to remember them. They can be quite useful as a starting point for induction, for example (covered in detail in Chapter 4).

■ **注意** 仅有0个或1个节点的图称为平凡图，它们的作用只是让定义变得更加复杂，事实上并没有太大必要。通常情况下，我们就直接略过这些情况，不过有时还是得记住这些概念。使用归纳法时，它们就能体现出作用了（将在第4章中讨论）。

Trees have several interesting and important properties, some of which are dealt with in relation to specific topics throughout the book. I’ll give you a few right here, though. Let *T* be an undirected graph with *n* nodes. Then the following statements are equivalent (Exercise 2-9 asks you to show that this is, indeed, the case):

树有一些有趣而非常重要的特性，它们将始终贯穿于本书中。以下就是实例：假设T为一个拥有n个节点的无向图。那么以下命题实际上是等价的（练习2-9会证明这一切）

1. *T* is a tree (that is, it is acyclic and connected).

*T*是一棵树（也就是说无环而连通）

1. *T* is acyclic and has *n*–1 edges.

*T*无环且有n-1条边

1. *T* is connected and has *n*–1 edges.

*T是连通的并且有n-1条边*

1. Any two nodes are connected by exactly one path.

任意两个节点之间有且只有一条路径

1. *T* is acyclic, but adding any new edge to it will create a cycle.

*T是无环的，但是任意添加一条新边都会产生环*

1. *T* is connected, but removing any edge yields two connected components.

*T是连通的，但是任意删除一条边都会将它分成两个连通分支*

In other words, any one of these statements of *T*, on its own, characterizes it as well as any of the others.

也就是说，以上关于T的命题中的任意一条都和其他命题是等价的。

If someone tells you that there is exactly one path between any pair of nodes in *T*, for example, you immediately know that it is connected and has *n*–1 edges and that it has no cycles.

如果有人说T中任意两点之间有且只有一条路径，那么你应该立刻推导出它是连通的，有n-1条边，并且没有环。

Quite often, we anchor our tree by choosing a *root node* (or simply *root*). The result is

Quite often, we anchor our tree by choosing a *root node* (or simply *root*). The result is called a *rooted tree*, as opposed to the *free* trees we’ve been looking at so far. (If it is clear from the context whether a tree has a root or not, I will simply use the unqualified term *tree* in both the rooted and free case.) Singling out a node like this lets us define the notions of *up* and *down*. Paradoxically, computer scientists (and graph theorists in general) tend to place the root at the top and the leaves at the bottom. (We probably should get out more …). For any node, *up* is in the direction of the root (along the single path between the node and the root). *Down* is then any other direction (automatically in the direction of the leaves). Note that in a rooted tree, the root is considered an internal node, *not* a leaf, *even if it happens to have a degree of one*.

通常情况下，我们在画出树之前会给它找一个根节点（简称根）。这样一来，它就是一颗有根树，与之相对的是自由树。（如果上下文足够清晰能够区分树是否含有根节点，那么树就能同时指代有根树和自由树。）挑选出了根节点之后我们就能定义“上”和“下”这样的概念。不过，在计算机科学中（更进一步说在整个图论中），越来越多地把根节点置于顶端而把叶节点放在底部，这与定义正好完全相反（这样的情况还有很多）。对于任何节点而言，指向根节点的方向是“向上”的（沿着节点到根的路径）。反过来则是“向下的”（指向叶节点）。请记住，在有根树中，根节点视为内部节点而非节点，即使这棵树的度数正好是1。

Having properly oriented ourselves, we now define the *depth* of a node as its distance from the root, while its *height* is the length of longest downward path to any leaf. The height of the tree then is simply the height of the root. For example, consider the larger tree in *G*1 in Figure C-1 and let *a* (highlighted) be the root. The height of the tree is then 3, while the depth of, say, *c* and *d* is 2. A *level* consists of all nodes that have the same depth. (In this case, level 0 consists of *a*, level 1 of *b*, level 2 of *c* and *d*, and level *3* of *e*.)

正确定义了指向之后，我们能够将节点与根节点之间的距离定义成它的深度，而高度则是它到叶节点之间的最长路径长度。整棵树的高度正是根节点的高度。对于C-1的*G*1中较大的那棵树，假设*a（高亮显示）为根节点，那么这棵树的高度为3，c* 和 *d的深度是2。所有包含相同深度节点的集合称为层。（本例中，第0层包含a，第1层包含b，第2层由c和d构成，第3层则只有e）。*

These directions also allow us to define other relationships, using rather intuitive terms from family trees (with the odd twist that we have only single parents). Your neighbor on the level above (that is, closer to the root) is your *parent*, while your neighbors on the level below are your *children*.[[5]](#footnote-5) (The root, of course, has no parent, and the leaves, no children.) More generally, any node you can reach by going upward is an *ancestor*, while any node you can reach by going down is a *descendant*. The tree spanning a node *v* and all its descendants is called the *subtree rooted at v*.

有了方向的概念之后，我们还能通过非常直观的家族树来定义许多其它关系（多个子节点共享同一个父节点）。某节点的上一层邻节点（离根节点更近的）称为它的父节点，而某节点的下一层邻点则称为它的子节点（当然了，根节点没有父节点，而叶节点没有子节点）。推而广之，向上的路径中的节点称为祖先节点，向下路径中的节点称为后代节点。包含节点*v以及它所有后代节点的树称为以v为根的子树。*

■ **Note** As opposed to subgraphs in general, the term *subtree* usually does *not* apply to all subgraphs that happen to be trees—especially not if we are talking about rooted trees.

■ **注意** 与子图的概念不同，子树通常并不指代所有恰好为树的子图——特别是在讨论有根树时。

Other similar terms generally have their obvious meanings. For example, *siblings* are nodes with a common parent. Sometimes siblings are *ordered* so that we can talk about the “first child” or “next sibling” of a node, for example. In this case, the tree is called an *ordered tree*.

其它类似的术语通常都明确的含义。比如说：兄弟节点指的是那些共享同一个父节点的节点。有时兄弟节点是有序的，我们能指定某个节点的“第一个子节点”或者“下一个兄弟节点”。在这种情况下，这棵树就称为有序树。

As explained in Chapter 5, many algorithms are based on *traversal*, exploring graphs systematically, from some initial starting point (a start node). Although the way graphs are explored may differ, they have something in common. As long as they traverse the entire graph, they all give rise to *spanning trees*.[[6]](#footnote-6) (Spanning trees are simply spanning subgraphs that happen to be trees.) The spanning tree resulting from a traversal, called the *traversal tree*, is rooted at the starting node. The details of how this works will be revisited when dealing with the individual algorithms, but graph *G*4 in Figure C-1 illustrates the concept. The highlighted subgraph is such a traversal tree, rooted at *a*. Note that all paths from *a* to the other nodes in the tree follow the edge directions; this is always the case for traversal trees in digraphs.

第5章中将会提及：许多算都法基于遍历，它通过某个起始点（从某个节点开始）系统性地探索整张图。虽然探索方式各不相同，但有些概念是共通的。在遍历整个图的过程中，生成树的数量会逐渐增加（生成树就是那些恰好为树的生成子图）。通过遍历产生的生成树称为遍历树，它的根节点就是起始节点。更进一步的细节将会在遍历算法中展开介绍，不过C-1中的*G*4很好地诠释了这一概念。高亮部分的子图就是一棵以*a*为根节点的遍历树。请注意所有从*a到其它节点的路径都跟随了边的方向；大部分情况下有向图的遍历树都是如此。*

■ **Note** A digraph whose underlying graph is a rooted tree and where all the directed edges point away from the root (that is, all nodes can be reached by directed paths from the root) is called an *arborescence*, even though I’ll mostly talk about such graphs simply as trees. In other words, traversal in a digraph really gives you a *traversal arborescence*. The term *oriented tree* is used about both rooted (undirected) trees and arborescences because the edges of a rooted tree have an implicit direction away from the root.

■ **注意** 如果有向图中的基础图是有根树，所有的边都从根节点指向其它节点（也就是说所有节点都可以通过从根节点出发的路径访问到），那么它就被称为有向树形图，不过我更愿意将这样的图简称为树。换言之，图的遍历过程中会产生遍历有向树形图。有向树这个术语同时适用于有根数（无向）以及有向树形图，因为有根数的边总是从根节点指向其它节点。

Terminology fatigue setting in yet? Cheer up—only one graph concept left. As mentioned, directed graphs can be acyclic, just as undirected graphs can. The interesting thing is that these graphs don’t generally look much like forests of directed trees. Because the underlying undirected graph can be as cyclic as it wants, a *directed acyclic* *graph*, or DAG, can have an arbitrary structure (see Exercise 2-11), as long as the edges point in the right directions—that is, they are pointing so that no *directed* cycles exist. An example of this can be seen in sample graph *G*4.

是否对没完没了的术语感到厌倦了？请打起精神来——现在只剩最后一个概念了！正如先前介绍的那样，有向图可以不含环，无向图也是这样。有趣的是这样的图看上去并不像森林或者有向树。原因是基础无向图完全可以包含任意数量的环，而一个有向图，只要每条边都指向合适的方向，就可以形成有向无环图（简称DAG），它的结构可以是任意的（参阅练习2-11）——也就是说，通过指定合适的方向，就能避免有向环的产生。*G*4就是一个很好的实例。

DAGs are quite natural as representations for dependencies because cyclic dependencies are generally impossible (or, at least, undesirable). For example, the nodes might be college courses, and an edge (u,v) would indicate that course u was a prerequisite for v. Sorting out such dependencies is the topic of the section on topological sorting in Chaptr 5. DAGs also form the basis for the technique of dynamic programming, discussed in Chapter 8.

有向无环图能够很好地模拟依赖关系，因为循环依赖通常是不可能的（至少来说是不符合需求的）。比如：节点所代表的可能是大学课程，边(u,v)则表示课程u是课程v的先决条件。第5章中的拓扑排序部分设计了这方面的讨论。第8章中会提到有向无环图也是动态编程技术的根本所在。

1. As quoted by Albert-László Barabási in his book *Linked: TheNew Science of Networks* (Basic Books, 2002).

   正如Albert-László Barabási在他的著作《*Linked: TheNew Science of Networks* (Basic Books, 2002).》中引用的那样 [↑](#footnote-ref-1)
2. You probably didn’t even think of it as an issue, but you can assume that *V* and *E* don’t overlap.

   你甚至可能没有考虑过这有没有问题，不过你还是可以假设V和E不会重合。 [↑](#footnote-ref-2)
3. The functions would still be called *V* and *E*, even if we give the sets other names. For example, for a graph *H* = (*W*,*F*), we would have *V*(*H*) = *W* and *E*(*H*) = *F*.

   即使有别的名称，函数还是可以定义成V和E，。比如，对于*H* = (*W*,*F*)这样的图，我们还是可以定义*V*(*H*) = *W* and *E*(*H*) = *F*. [↑](#footnote-ref-3)
4. As explained later, though, the root is not considered a leaf. Also, for a graph consisting of only two connected nodes, calling them both leaves sometimes doesn’t make sense.

   我们稍后将会详细说明，根节点不会被认作叶节点。同样的，对于只有两个连通节点的图，将两个节点都称为叶节点也没有任何意义。 [↑](#footnote-ref-4)
5. Note that this is the same terminology as for the in- and out-neighborhoods in digraphs. The two concepts coincide once we start orienting the tree edges. [↑](#footnote-ref-5)
6. Thisis true only if all nodes can be reached from the start node. Otherwise, the traversal may have to restart in several places, resulting in a *spanning forest*. Each component of the spanning forest will then have its own root.

   当且仅当所有节点都能通过起始节点访问到时才成立。否则，必须换一个起始节点重新开始新的遍历，结果会导致生成森林的产生。每一个生成森林的分支都有它自己的根节点。 [↑](#footnote-ref-6)