## 第5章 遍历：算法学中的万能钥匙

*若您目前正处于一条狭长的走廊中，这条走廊有数米之长，尽头是一扇门。此外，走廊中间还有一座拱门，这里的台阶通往下方。这时候，您会选择向前走向那扇门（转向5），还是沿着台阶往下爬呢（转向344）？*

*——摘自《Citadel of Chaos》  
Steve Jackson著*

在大多数时候，图结构都是一种非常强大的结构化思考（或数学）模型。如果您能用图的处理方式来规范化某个问题，即使该问题本身*看上去*并不像是个图问题，也能使您离解决问题更进一步。在众多图*算法*中，我们时常会用到一种非常实用的思考模型——如果愿意的话，甚至可以将其视为一把万能钥匙[[1]](#footnote-1)。这把万能钥匙就是*遍历（traversal）*：即对图中所有节点的探索及访问操作。当然，这里要说的不单是对*显式*图结构的遍历。例如我们可以思考一下，GIMP或Adobe Photoshop这类绘图程序究竟是如何将某种单色填充到指定区域内（进行所谓的填色操作）的，这就需要用到我们接下来要学习的东西（见习题5-4）。再比如，当我们要序列化某个复杂的数据结构时，势必要对该结构中所有的组成对象进行检查。这本身也是遍历。当我们需要列出文件系统中的所有文件或某部分文件，或管理软件包之间的依赖关系时，显然会涉及到更多的遍历。

但遍历还并不只有这些直接作用。它同时还是*其它*许多算法（譬如第9、10两章中的算法）的关键组成部分和基本原理。例如在第10章中，我们会遇到一个为*n*个人分配*n*个工作的问题。由于该问题中的每个人都只拥有其中一部分工作所需要的技能。因此那个算法的工作原理是先暂时将工作分配给某个人，一旦发现某项工作更适合于*其他人*，再重新分配。然后这次分配又可能引发下一次重新分配，最终形成一系列联动反应。并且如您所见，这种联动会在人与工作之间来回，形成一个锯齿状分布，从某个人的闲置状态开始，到其拥有合适的工作为止。这里用的是什么方法呢？猜对了，就是遍历。

接下来，我们将会分多个版本、从多个角度来阐述这一思路，以便尽可能地将其方方面面一并说清楚。这意味着我们将会介绍到两种著名的基本遍历策略：*深度优先搜索*（*depth-first search*）及*广度优先搜索*（*breadth-first search*），并通过构建一个稍微复杂一些的、以遍历技术为基础的算法，以找出相关图中的强连通分量。

我们可以基于某种基本归纳法，用遍历技术来建构某种层次的抽象。下面，我们看一个在一个图中寻找其连通分量的问题。可能您还记得，在第2章中我们曾经说过，如果一个图中的任何一个节点都有一条路径可以到达其它各个节点，那么它就是连通的。而现在我们所谓的连通分量就是指的目标图中最大（且独立）的连通子图。找出这种连通分量的方法之一就是从图中的某个部分开始，逐步扩大其连通子图的确认范围，直至它再也无法向外连通为止。那么。我们究竟如何确认我们已经完整地重构了这一部分呢？

对此，我们可以从下面这个问题入手：请您证明在一个连通图的节点序列*v*1, *v*2, … *vn*中，对于任何*i* = 1…*n*，其子图节点*v*1, … , *vi*也都是连通的。如果我们能证明这一点，并且找出其具体排序方式，我们就可以在其某个连通分量中访问到它所有的节点，并能了解到该访问操作会在什么时候完事。

那么应该怎么做呢？答案是归纳思想，我们需要从*i*-1中推导出*i*的情况。而现在我们所知道的是，该图前*i*-1个节点所组成的子图是连通的。那下一步呢？嗯，由于该图中任意一对节点之间都存在着路径，所以我们可以考虑前*i*-1个节点中的一个节点*u*，和剩余部分节点中的一个节点*v*。在这条从*u*到*v*的路径中，我们可以找到*在*我们目前所建构的部分中的最后一个节点，以及第一个*不在*这个部分中的节点。我们可以分别将其称作x和y。很显然，这两者之间也一定存在着一条边，所以将y加入到现有节点中，就会使得我们的连通分量持续扩张，由此证明了我们所要证明的东西。

在这里我们希望读者明白的是，其实际结果的产生过程有多么地容易。它只是一个将连通节点添加到相关部分中，而且该节点应该是顺着一条边线而被发现的。这里很有趣的一点是，只要我们通过这种方式持续地往该连通分量添加新节点，最终会构建出一个*树结构*。这种树结构就是所谓的*遍历树*（*traversal tree*），而且是我们正在遍历的连通分量的生成树。（当然，对于有向图来说，它只能根据我们所能到达的节点来生成。）

而要想实现这个过程，我们就得需要以某种方式持续跟踪那些通过一条边线就能到达的“边沿”或“前沿”节点。如果从某个单一节点开始的话，其前沿就只有它的邻居节点。而当您开始向外扩展时，新访问到的节点的邻居节点就会成为新的边沿节点，而我们目前所访问到的节点就落在连通分量里了。换句话说，我们需要以某种形式维护一个边沿节点的集合，我们反复地从该集合排除掉我们访问过的节点，并向其中添加它们的邻居节点（除非这些节点本身已经在该列表中，或已经被访问到了）。这就构成了一个我们想要访问但还未完成访问的节点列表。有了这个列表，我们可以默认为那些*已经*被访问过的节点，就一定被检查过了。

下面，我们拿曾经玩过的《龙与地下城》[[2]](#footnote-2)这种老式的角色扮演类游戏（事实上也包括今天所流行的许多视频游戏）来举个例子。在图5-2中，我们可以很清晰地看到这类游戏的设计，这是一张典型的地牢地图[[3]](#footnote-3)，我们可以把其中的房间（及走廊）当作节点，而将门之间的距离当作边线。当然您会看到该图中存在着许多条边（门），但这些都不是真正的问题。此外，我们还在地图上标出了“您当前的位置”，以及一条用于指示目标的路线。

（图）

**图5-1.** 带有三个连通分量的无向图

（图）

**图5-2.**这是一张典型的角色扮演类游戏的地牢图的部分遍历结果，我们可以将其中的房间视为节点，而门之间的距离则视为边线。其遍历树将由您的运动轨迹来定义。其边沿节点集（即遍历队列）则将由相邻的房间组成，除此之外的其它部分（即阴影部分）的房间就都属于未探索区

请注意，这里有三种类型的房间：有一种房间是我们实际访问过的（即您的运行轨迹会穿过它们）；还有一种房间属于是我们所知道的，因为我们看到过它们的门；最后一种房间是我们所不知道的（即阴影部分）。其中，将未知房间与已访问房间区隔开来的前沿节点（当然）就是那些已知但尚未访问的房间。下面，我们来给出这种一般性遍历策略的简单实现，如清单5-1所示。[[4]](#footnote-4)

**清单5-1.**遍历一个表示为邻接集的图结构的连通分量

def walk(G, s, S=set()): # Walk the graph from node s

P, Q = dict(), set() # Predecessors + "to do" queue

P[s] = None # s has no predecessor

Q.add(s) # We plan on starting with s

while Q: # Still nodes to visit

u = Q.pop() # Pick one, arbitrarily

for v in G[u].difference(P, S): # New nodes?

Q.add(v) # We plan to visit them!

P[v] = u # Remember where we came from

return P # The traversal tree

|  |
| --- |
| **提示：**set类型的对象也可以让我们在其它类型上执行某些集合操作。例如在清单5-1中，我们可以在不同的方法中像使用set（有关该dict的键的集合）一样使用dict对象P。而且，这种工作方式也可以被运用在其它迭代类型（如list或deque）以及其它集合方法（如update()等）上。 |

在上面这段新代码中，有一对东西的作用并不是那么显而易见的。比如，参数S到底是什么？为什么我们要用字典类型（而不是集合类型）来跟踪自己所访问过的节点呢？目前看来，这个参数S似乎是完全没有用处，但当我们试图要找出其*强连通分量*时就会需要用到它（相关内容我们将会在本章临近结束时介绍）。基本上，这个参数所代表的是一个“禁区”——一个我们在遍历过程中没有访问到，但又被告知需要回避的节点集合。至于说到字典对象P，它的主要作用是表示已经访问过的*前趋节点*，每当我们往队列中添加新的节点时，都会同时设置其前趋节点。这样我们就能确保自己在必要时能知道应该去哪儿找它们。将这些前趋节点组合在一起，就形成了相应的遍历树。如果您并不关心这个树结构，当然也可以改用已访问节点的集合来做这件事（对此，我们将会在本章稍后的一些实现中介绍）。

|  |
| --- |
| 请注意：无论您是打算在节点被添加到相关队列的同时就将其添加到这种“已访问”集合中，还是以后再来添加，当我们将它们从队列中弹出（pop）的时候，这通常是无关紧要的。但是，如果你需要做一个“是否已被访问过……”的检查，那么就要注意及早添加了。对此，我们将会在本章中看到若干个这样的版本。 |

然而，该walk函数所遍历的只是单个连通分量（既定目标是个无向图）。如果想要找出该图*所有的*连通分量，我们就得要将其封装在一个涉及到所有节点的循环中，具体如清单5-2所示。

**清单5-2.**找出图的连通分量

def components(G): # The connected components

comp = []

seen = set() # Nodes we've already seen

for u in G: # Try every starting point

if u in seen: continue # Seen? Ignore it

C = walk(G, u) # Traverse component

seen.update(C) # Add keys of C to seen

comp.append(C) # Collect the components

return comp

在这里，walk函数返回的是一个已被访问过的前趋节点的映射集（即递归树），而我们将这些映射集收进了comp列表中（以代表连通分量）。另外，我们还用seen集合来确保自己不会遍历到之前连通分量中的节点。请注意，即便seen.update(C)是一个有关C的规模的线性操作，我们调用walk的工作量也基本是相同的。所以从渐进法的角度来说，它并没有给我们带来更多开销。总而言之，找出图中连通分量应该是一个Θ(*E* +*V*)时间的操作，因为该图中的各条边和节点都是必须要探索的。[[5]](#footnote-5)

尽管walk函数也并没有真正做完所有的事情。但从很多方面来说，我们都可以将这段简单的代码视为本章的基础骨架，以及（标题）将来学习并理解许多其它算法的万能钥匙。它很值得我们深入研究一番。我们可以试着在自己所选的图结构（如图5-1）上手动执行一下该算法，以便了解一下其探索图中所有连通分量的过程。这里有一点很重要，即尽管Q.pop()返回节点的顺序*并不重要*，整个连通分量的探索过程并不需要这种区分，但该顺序是walk函数行为定义的关键元素，通过调整这种顺序，我们可以获得一系列即插即用的实用算法。

关于其它类型的图遍历，您也可以参考图5-3与图5-4。（如果还想了解更多例子，还可以参考下面的专栏内容）

（图）

**图5-3.**科尼斯堡（今加里宁格勒）的桥梁图。选自《Recreations Mathematiques》第１卷（Lucas 1891年著，第22页）[[6]](#footnote-6)

（图）

**图5-4.**这是一个十二面体，目标是沿着图中的边线，访问它的每一个顶点各一次。该图选自《Recreations Mathematiques》第2卷（Lucas 1896年著，第205页）

|  |
| --- |
| **加里宁格勒的跳岛** |
| **听说过科尼斯堡（如今的加里宁格勒）的那个七桥问题吗？在1736年，瑞士数学家Leonhard Euler[[7]](#footnote-7)曾在这里遇到过一个难题，该城中的许多居民一直都希望能解决它。这个问题是这样的：人们希望无论从城中哪一点出发，都能一次性地将城中的七座桥全部走一遍，并回到原点（这七座桥的具体布局如图5-3所示）。为了解开这个难题，Euler决定先抽象掉一些细节，然后……就开创了图论这门学科。这个谜题似乎是一个不错的开局，不是吗？**  **或许您已经注意到的了。图5-3其实是一个河岸与岛屿的多重图结构。例如A、B之间及A、C之间都有两条边。但它们并不会对问题本身产生影响。（我们可以轻松地在这些边的中间位置制造出一些假想的岛屿，以获得一个普通的图结构。）**  **Euler最终证明了的是，我们能在一个（多重）图结构中一次性访问到其所有的边，并返回原点，当且仅当该图为连通图，且其各节点的度为偶数时。对于这种*闭合式遍历路径*（在该路径上，该图的节点是可以被多次访问的），我们称之为*欧拉环路（Euler tour）*或*欧拉回路（Euler circuit）*，并称这样的图结构为*欧拉图（Eulerian）*[[8]](#footnote-8)（相信我们已经可以轻松地看出，科尼斯堡七桥并不是一个欧拉图，它所有节点的度都是奇数。）**  **从以上论述中不难看出，图的连通性以及其各节点的度为偶数是解决这个问题的*必要条件*（非连通性显然是个障碍，而如果其节点度为奇数的话，也必然会让我们的环路止步于某处）。而有些不太明显的是，它们同时也是解决问题的*充分条件*。关于这一点，我们可以用归纳法来证明它（这是一个大惊喜，不是吗？），但对于相关的归纳参数，我们得小心仔细一些。如果直接从移除图中的节点或边入手，我们化简出来的问题可能就不是欧拉图了，而且我们的归纳前提也会变得不可用。其实在这个问题中，我们不必担心图的连通性。如果化简后的图是不连通的，我们也可以将归纳前提运用到其各个连通分量中去。但偶数度方面又该如何保证呢？**  **我们被允许访问同一个节点任意多次，所以我们所要移除（或者说“用掉”）的东西，是一组边的集合。而如果我们从自己访问的那些节点之上移除了偶数数量的边，就能应用我们的归纳前提。所以在这种情况下，我们选择的一种方式是移除一些闭合式遍历操作中所经过的边（不一定要访问所有的节点）。接下来的问题是，欧拉图中是否总是能存在这样的闭合式遍历操作？如果我们从某些节点*u*出发，所进入的每个节点的度都由偶数变成了奇数，就可以安心地离开它们。只要我们所走过的边没有被重复经过，就最终一定会回到*u*节点。**  **现在我们将归纳前提设定为：在一个节点度为偶数，且边数少于E的连通图中，我们都能一次性地闭合式遍历到其每一条边。我们从E中边开始，移除掉任意的闭合式遍历路径中的所有边。然后，我们就会得到一个或多个部分，这些部分也分别是欧拉图，可以继续适用于归纳前提。最后一步是合并我们在这些部分中获得的欧拉环路。由于我们的原始图结构是连通的，所以被移除的封闭式遍历路径也必然能用来连通这些个部分。因此，该问题的最终解决方案应该就是我们所合并的这些遍历路径，这条路径就是“绕道”走过每个部分的欧拉环路的结果。**  **换而言之，判断某个图结构是否属于欧拉图，并找出其中的欧拉环路其实很简单（见习题5-2）。但欧拉环路引出了一个相对更为复杂的问题：哈密顿环路问题（Hamilton cycle）。**  **哈密顿环路问题是由爱尔兰数学家William Rowan Hamilton爵士所命名（他还有一些别的头衔[[9]](#footnote-9)），该问题最初出自一个（名为*The Icosian Game*的）游戏，该游戏的目的是要一次性地访问到一个正十二面体（其由12个正多面体组成，也可简写为12d）上的所有顶点，并返回原点（如图5-4所示）。在更一般的情况下，一个哈密顿环路是一个包含整个图结构中所有节点的子图（恰好形成一次性经过的环路）。我相信您肯定已经看出来了。科尼斯堡七桥就属于哈密顿图（也就是说，它存在一个哈密顿环路）。而要证明这个正十二面体是一个哈密顿图是有点困难的。事实上，要在一般性图结构中找出哈密顿路径是一个非常非常困难的问题，人们至今也还没找到解决该问题的有效算法（详见第11章）。考虑到欧拉问题和哈密尔顿问题是挺类似的，这难度上的巨大差异则让人感到怪异，您不觉得吗？** |

### 公园漫步

1887年深秋的一天，一位法国电信工程师散步时路过了一个保存完好的花园式迷宫，于是就进去转了转，顺便看看已经发黄的树叶。他走过迷宫中的一个个通道和路口，认出了一些植物，但同时他也发现自己一直在原地转圈。作为一个思考者，这位工程师开始思考应该如何避免这样的错误，并尽可能地找到走出去的最好方式。他记得小时候有人告诉过自己，只要在每一个路口都一直坚持向左走，就能最终找到出口。但我们很容易就能看出，这种简单的策略是行不通的。如果真的一直向左走下去，在找到出口之前就会回到起点。这会使他陷入一个无限循环。这不行，他得另想办法。而当这位工程师最终摸索着走出迷宫时，他获得了某种闪亮的洞察力。于是他急忙赶回家，在笔记本上记录下了他的解决方案。

好吧，我承认这位工程师的实际经历可能并不是这个样子，上面说的一切，甚至包括具体年份[[10]](#footnote-10)都是我虚构的。但事实部分是在19世纪80年代末，一位名叫Trémaux的电信工程师发明了一种用来遍历迷宫的算法。我很快将会讨论这个算法。但现在，我们得先来探讨一下这个“一直向左走”的策略（通常也被称之为*左手规则*），看看它的工作方式——以及它何时是不适用的。

#### 不允许出现环路

下面我们以图5-5中的迷宫为例来思考一下这个问题。如您所见，左图的迷宫中没有环路，其基本结构是一棵树，具体如右图所示，在这种情况下，“保持单手扶墙走”的策略当然是切实可行的[[11]](#footnote-11)。了解该策略工作原理的方式之一就是，您会观察到该迷宫其实始终只有一面内墙（也就是说，如果我们为其贴上墙纸，它将会是一条连续不断的长带）。从外围来看，只要在不允许出现环路的情况下，我们所画的任何隔断都只能将路导向一个确定的地方，可见这对于左手规则来说根本不成问题。顺着这种遍历策略，我们会探索到图中所有的节点，而且每条通道都会经过两次（每个方向各一次）。

左手规则本身实际上就是为了让一个人利用它在迷宫中漫步，并只根据局部信息而设计的。为了真正掌握这方面的内容，我们可暂且先放弃当前的角度，并转而去制定一个同样的*递归性*策略[[12]](#footnote-12)。只要我们熟悉递归思想，就很容易通过这种制定来证明相关算法的正确性，并且还是最简单的那种递归算法。下面就是它的一个最基本实现（假设这棵树是用我们标准的图结构来表述的），具体如清单5-3所示。

**清单5-3.**递归的树遍历算法

def tree\_walk(T, r): # Traverse T from root r

for u in T[r]: # For each child...

tree\_walk(T, u) # ... traverse its subtree

（图）

**图5-5.**这一个树结构图，表示的是一个迷宫图以及覆盖其上的传统图结构

从迷宫比喻的角度来说就是，当我们正处于某个十字路口，既可以向左也可以向右时，先选择遍历迷宫的左半部分，然后下一次再来走右半部分。很显然（也许也需要用到一点归纳法），这种遍历策略最终会涵盖整个迷宫。值得注意的是，在每个路口径直往前走的动作是这里唯一被切实描述的行为。即当我们所要遍历的子树的根节点在*u*节点上时，我们会先走到*u*节点上，然后开始从这里经过所有新的道路再走出来。最终，我们还是得返回全树的根节点*r*。对于这种回归的运行轨迹，我们称之为*回溯*（*backtracking*），该操作通常会被隐藏在递归操作中。每次当递归调用返回时，其实都会自动返回到调用发起的那个点上。（您怎么看这里的回溯操作与左手规则之间的关联呢？）

下面，我们假设有人在该迷宫的某面墙上开了个洞，使对应的图结构中出现了一个环路。例如可能被打穿的是对死胡同e来说在北边的那面墙。那么，如果我们从e点出发去北面，只要愿意的话，我们还是可以一直向左走,，但永远都不可能再遍历整个迷宫了——因为我们一直在转圈[[13]](#footnote-13)。这在我们遍历一般性图结构时会成为一个问题[[14]](#footnote-14)。清单5-1中的一般性思路为避免这个问题提供了一种选择，但在我分析这个算法之前，我们先来看看之前那位法国电信工程师是怎么想的。

#### 停止循环遍历的方式

Édouard Lucas在其1891年写的 《*Recreations Mathematiques*》第一卷的引言中是这样描述Trémaux的迷宫遍历算法的：[[15]](#footnote-15)

*如果无论我们从哪一点出发，想要遍历整个迷宫，并且经过所有的通道两次，只要遵循Trémaux所定下的遍历规则，每次进入或离开一个十字路口时留下一个标志就可以了。我们可以将这些规则汇总如下：在可能的情况下，尽量避免重复访问之前走过的十字路口，避免重复走同一条通道，这种态度不仅适用于眼下的情况，也适用于我们每天的生活，不是吗？*

当然，那本书后续还对该方法的细节进行了更多的描述，但方法本身其实是很简单的，重点在于上面这段话的主体思维。例如，我们也可以不用特别在每次进出一个路口时留下标记（比如用粉笔划一道），光凭我们的脏鞋子在地上留下的泥印，就足以追踪自己的足迹（如图5-2所示）。然后Trémaux告诉我们，无论我们一开始朝哪个方向走，当我们走进某个死胡同，或者来到一个已经走过的路口时，就应该回溯回来（以避免陷入环路）。由于同一条通道不能被重复经过两次（一次前进一次回溯）以上，所以当我们*回溯到*某个路口时，就得选择一条没有走过的通道继续走。如果当下没有这样的通道，就要（顺着这些带有单向脚印的通道）一直回溯回去。[[16]](#footnote-16)

这就是所谓的算法。其中还有一个很有趣的现象，即尽管我们*向前*遍历时有若干条通道可以选择，但回溯时就往往只有*一条路*可走。您能明白其中的缘由吗？想要有*两种*（或更多种）回溯路径的话，只有这种可能性：我们在某个路口走向了别的方向，并在不回溯的情况下又走到了它。但对于这种情况，根据之前的规则，我们根本就不该进入这个路口，直接就可以原路返回了。（这也是我们为什么不会在相同方向上经过一条通道两次的原因。）

在这里，我之所以会用到“带泥的鞋”这样的描述，是因为它能清晰地将回溯过程真正地呈现出来。它确实与递归树上的一种遍历非常类似（当然也与左手规则等效）。事实上，如果采用递归式制定的话，Trémaux的算法所做的也就是一种会消耗一些额外内存的树遍历操作而已。因为在这个相应的树结构（我们称之为遍历树）中，我们得记录哪些节点已被访问过，并设置一道墙，以防止我们再次进入这些节点。

清单5-4中所显示的就是一个递归版的Trémaux算法。在这个算法定制中，我们所用的是通常被称之为深度优先搜索的策略，这是最基本（同时也是最重要）的遍历策略之一。[[17]](#footnote-17)

|  |
| --- |
| **请注意：**与清单5-1中walk函数不同的是，我们在这里的循环里不能在G[s]上调用difference方法了，因为s在递归调用中是会发生变化的，而且这样一来我们会很容易地对某些节点作多次访问。 |

**清单5-4.** 递归版的深度优先搜索

def rec\_dfs(G, s, S=None):

if S is None: S = set() # Initialize the history

S.add(s) # We've visited s

for u in G[s]: # Explore neighbors

if u in S: continue # Already visited: Skip

rec\_dfs(G, u, S) # New: Explore recursively

### 继续深入！

这种深度优先搜索（DFS）策略从递归性结构中获得了一些最重要的属性。一旦我们在某个节点上启动了这一操作，我们就得确保自己能在相关操作继续下去之前遍历完其它所有我们所能到达的节点。但正如第4章中所说的那样，任何递归函数都是可以用迭代操作来重写的，一种做法是用我们自己的栈来模拟调用栈。这种基于迭代操作的DFS是非常实用的，它既可以避免调用栈被塞满所带来的问题，也能因此改善算法的某些属性，使其显得更为清晰一些。幸运的是，为了模拟递归遍历，我们在这里只需使用一个栈结构，而不是清单5-1中walk函数中的集合结构。下面我们就来看看迭代版的DFS，如清单5-5所示：

**清单5-5.** 迭代版深度优先搜索

def iter\_dfs(G, s):

S, Q = set(), [] # Visited-set and queue

Q.append(s) # We plan on visiting s

while Q: # Planned nodes left?

u = Q.pop() # Get one

if u in S: continue # Already visited? Skip it

S.add(u) # We've visited it now

Q.extend(G[u]) # Schedule all neighbors

yield u # Report u as visited

除了栈的运用外（这是一种*后进先出（LIFO）*队列，我们可用list的append和pop方法来实现它），我们还对上面这段代码进行了一些调整。例如，由于在原walk函数中，该队列是set类型的，我们无法对相同节点上的多次访问作出任何调度。但一旦我们采用了其它队列结构，那么这就是个问题了。因为在添加一个节点的邻居节点之前，我们已经通过对S成员中相关节点的检查操作（检查该节点是否已经被访问过）解决了这个问题。

为了让这段遍历显得更实用一些，我还在其中添加了一个yield语句，它能使我们按照DFS的顺序来遍历目标图结构中的节点。例如，如果我们在变量G中使用的是图2-3中的图结构，可能就会试出以下结果：

>>> list(iter\_dfs(G, 0))

[0, 5, 7, 6, 2, 3, 4, 1]

值得一提的是，我们刚才是在一个*有向图*上执行DFS的，而之前讨论的都是该算法在*无向图*上的工作方式。实际上，无论是DFS还是其它遍历算法，都一样可以适用于有向图上。但当在有向图上执行DFS时，我们就不能指望它能探索到一个完整的连通分量了。例如对于图2-3中的那个图结构来说，从a以外的任何一点出发都不能到达a，因为该节点本身不存在入边。

|  |
| --- |
| **提示：**如果想要找出有向图中的连通分量，我们可以先将其简化成对应的无向图，或者直接为图中所有的边添加反向边。这就可以使其同样适用于其它算法。甚至有时候我们都不用构建无向图，只需在使用有向图时直接思考一下其每条边在两个方向上的情况，或许就足够了。 |

我们一样可以将这种思维作用于Trémaux的算法。我们依然按照老方法遍历每一条（有向）通道，但这回我们只被允许沿着这些边的方向*向前走*，*回溯操作*就必须沿其反方向了。

事实上，iter\_dfs函数的结构已经非常接近于我们早先所暗示的通用性图遍历算法了——只需将其队列类型替换一下即可。下面我们就来看看更成熟的遍历算法（如清单5-6所示）。

**清单5-6.** 通用性的图遍历函数

def traverse(G, s, qtype=set):

S, Q = set(), qtype()

Q.add(s)

while Q:

u = Q.pop()

if u in S: continue

S.add(u)

for v in G[u]:

Q.add(v)

yield u

这里采用的默认队列类型是set，这使它更接近于原先的（一般性）walk函数。当然，我们很容易就能将其定义成stack类型（借助于一般队列协议中所定义的pop和add方法），例如像这样：

class stack(list):

add = list.append

对于之前的深度优先测试，我们可以将其替换如下：

>>> list(traverse(G, 0, stack))

[0, 5, 7, 6, 2, 3, 4, 1]

当然，使用各种专用版本的遍历算法也很不错，但我们可以选择用某种大致相同的形式来表现它们。

#### 深度优先的时间戳与拓扑排序（再次）

正如我们之前所提到的，记住并回避之前访问过的节点是我们会不会陷入环路，以及能不能在无环路遍历过程中自然形成一个树结构的关键。根据这些树各自不同的构造方法，它们分别有着不同的名称。在DFS中，它们顾名思义应该叫做深度优先树（或DFS树）。与其它所有的遍历树一样，DFS树的结构也取决于其中节点被访问的顺序。其中，为DFS树所特有的一件事是，在此类树结构中，任意节点u下所有后代节点都将会在u开始被探索到完成回溯操作之间的这段时间被处理。

如果想要让这个属性发挥其作用，我们就需要去了解该算法的具体回溯时间，这对于迭代版本可能会有一点难度。尽管，我们也可以选择扩展一下清单5-5中迭代版的DFS，使其能持续跟踪自己的回溯操作（见习题5-7），但在这里，我们要扩展的是递归版的DFS（清单5-4）。在清单5-6这个版本中，我们为每个节点添加了时间戳，其中一种代表它被探索的时间（即探索开始时，标识为d），而另一种则代表我们回溯到该节点的时间（即探索完成时，标识为f）。

**清单5-6.** 带时间戳的深度优先搜索

def dfs(G, s, d, f, S=None, t=0):

if S is None: S = set() # Initialize the history

d[s] = t; t += 1 # Set discover time

S.add(s) # We've visited s

for u in G[s]: # Explore neighbors

if u in S: continue # Already visited. Skip

t = dfs(G, u, d, f, S, t) # Recurse; update timestamp

f[s] = t; t += 1 # Set finish time

return t # Return timestamp

在这里，参数d和f应该是映射型的（例如字典类型）。根据DFS的属性，每个节点在DFS树中该节点的各个后代节点被探索*之前*被发现（1），以及它们完成处理*之后*被完成探索（2）。（这点我们既可以从算法的递归形式中推导出来，也可以自行用归纳法来证明其正确性。）

该属性所带来的一个直接结果是，我们可以用DFS来进行拓扑排序（我们在第4章中曾讨论过这种排序）。如果我们在DAG上执行了DFS算法，那么我们就可以简单地根据其递减的完成时间来对节点进行排序，这种排序就是拓扑排序。在DFS树中，每个节点*u*都会先于其后代节点被访问，因为其所有后代节点都要经由*u*来访问，也就是说，这些节点是依赖于*u*的。这个例子中，我们了解一个算法的具体工作方式是有用的。我们并不是要先调用带时间戳的DFS，然后再来进行排序，而是要在DFS的*执行过程中*进行直接进行拓扑排序，具体如清单5-7所示。[[18]](#footnote-18)

**清单5-7.** 基于深度优先搜索的拓扑排序

def dfs\_topsort(G):

S, res = set(), [] # History and result

def recurse(u): # Traversal subroutine

if u in S: return # Ignore visited nodes

S.add(u) # Otherwise: Add to history

for v in G[u]:

recurse(v) # Recurse through neighbors

res.append(u) # Finished with u: Append it

for u in G:

recurse(u) # Cover entire graph

res.reverse() # It's all backward so far

return res

对于这段新的拓扑排序算法，有以下几点值得注意。首先，我们明确在for循环中涉及到了所有的节点，以确保整个图结构都能被遍历到。（在习题5-8中，我们会要求您证明这项工作的可行性。）检查曾存在于历史集合（S）中的节点的逻辑现在已经被放入了recurse函数内。也正因为如此，我们也不必在两个函数的for循环中都将其放进去。其次，由于recurse是个内部函数，它可以访问其所在域的对象（这里就是S与res）。其唯一的参数就是其所要遍历的那个起始节点。最后要记住的是，我们所进行的节点排序是*逆序*的，因为其根据的是其探索完成的时间。这也是res列表在被返回之前要进行反转的原因。

该topsort函数会在其执行回溯操作时对各个节点进行某些处理（并将它们追加到其结果列表中）。DFS回溯节点的顺序被称之为*后序*，而第一段中的访问顺序则被称之为*先序*。基于这些时间顺序的处理操作则分别被称之为*先序处理*或*后序处理*。（在习题5-9中，我们将会要求您为DFS中的这类处理操作添加一些通用性钩子函数）

|  |
| --- |
| **节点着色与边线类型** |
| **在描述遍历的过程中，我们实际上是在区分三种类型的节点，它们分别是未知节点、队列中的节点以及已访问过的节点（他们目前有邻居节点在队列中）。对此，有些书中引入了一种叫做节点着色的形式（例如Cormen等人所著的《*Introduction to Algorithms*》的第1章就曾有所提及），这种描述形式在DFS中尤为重要。该方法具体如下：一开始我们会将每个节点都标识为白色；然后将其被发现到完成处理之间的状态标识为灰色；而后则标识为黑色。当然，我们在实现DFS时并不是非得要进行这种分类。但它对于理解这一算法是非常有用的（或者至少它在您阅读到谈及这种着色的文字时有用）。**  **如果具体到Trémaux的算法中来说的话，灰色代表的就是我们已经知道，但却应该要回避的路口；而黑色代表的则是我们已经走过两次（在回溯的时候走到）的那些路口。**  **这些着色方法也可以用来对DFS树上的边线进行分类。当边uv处于被探索状态时，如果节点v为白色，那么这就是一条*树边*（*tree edge*）——也就说它属于遍历树的一部分。而如果节点v为灰色，那么这条边就被称为*后向边*（*back edge*），这在DFS树中是一种用于回到某个先辈节点的边，最后，如果节点v为黑色的话，这条边就有可能是*前向边*（*forward edge*）或*交叉边*（*cross edge*），前向边在遍历树中是一种用于指向某个后代节点的边，而交叉边则是一种用于指向除此之外任意节点的边（也就是除树边、后向边、前向边以外的边）。**  **但请注意，我们在上面的边线分类过程中并没有实际使用任何显式的颜色标签。假设一个节点在其被探索到完成处理之间存在着一个时间间隔，那么后代节点一定会被包含在其先辈节点的时间间隔内，而没有血缘关系的节点之间就不会有这样的重叠区。所以，我们用时间戳就足以判断某一条边是前向边还是后向边了。如果我们的时间戳是单步递增的，那么其子时间区间（即前向边）也将“只存在于”父时间区间内，即便这里使用了颜色标签，我们也还是需要用时间戳来区分出它是前向边还是交叉边。**  **其实，我们可能并不需要分出那么多类型，但它有一个非常重要的功能。即如果我们在某个图中找到了一条后向边，那么该图结构中就包含了一条环路，反之就不会存在环路。（在习题5-10中，我们会要求您证明这一点。）换句话说，这说明我们可以用DFS来检查一个图结构是否属于DAG（或者对于无向图来说，检查它是否是一棵树）。在习题5-11中，我们将会要求您去思考一下其它遍历算法是如何达成这一目标的。** |

### 无限迷宫与最短（不加权）路径问题

直到目前为止，DFS过于积极的行为还没有带来过什么问题。我们可以将该算法放入一个迷宫（即图结构）中，在执行回溯操作之前，我们可以以任由它转向一些其所能转向的方向。但如果这个迷宫非常大的话，这样做就有可能会产生一些问题了。例如，可能我们所要查找的东西（比如某个出口）就在靠近起点的地方；如果DFS设置了一个不同的方向，它返回可能就需要*很久*；还有如果该迷宫是无穷大的话，即使某些其他遍历方法几分钟之内就能找到出口，它也永远不会返回了。当然，无限迷宫的事情可能是有些夸大其词，但它非常接近于一类重要的遍历问题——即在一个既定状态空间内寻找解决方案。

但是，像DFS这种由于过于积极行为造成的迷失问题还不仅仅是巨型图结构。如果我们要找的是从起点到其它所有节点的*最短路径*（现在暂不考虑加权边的情况）的话，DFS也极有可能会给出一个错误答案。下面，我们通过图5-6中的例子来看看DFS所发生的问题，它的积极行为在其到达c点之前会一直持续走着弯路。而如果我们想找的是到达其它所有节点的最短路径的话（如右图所示），我们就需要更加保守一些。为了避免走出弯路，并到达某一个节点，我们得“在后台”一次一步地推进自己的遍历“前沿”。先是一步能访问到的所有节点，然后再是两步能访问到的所有节点，如此类推下去。

（图） （图）

从a点出发的DFS树 从a点出发的SP树

**图5-6.** 一个四节点大小环路的两种遍历形式。其中，DFS树（左边高亮部分）未必会与SP树（shortest path tree，最短路径树，右边高亮部分）一样会包含最短路径。

借着迷宫这个比喻，我们再来看看一个迷宫探索算法。该算法是Øystein（又叫Oystein）Ore在1959年提出的，与Trémaux一样，Ore也会要求我们标记通道的入口和出口。下面，假设我们是从*a*路口出发的。首先，我们会始终沿着一条通道访问迷宫中所有的路口，因而每次回溯都会直接回到起点。如果其中有任何一条通道走进了死胡同，我们都会在返回时留下一个关闭标志，并且任何会引导我们进入已走过路口的通道也会被标记为关闭状态（在其两端）。

在这里，我们想要探索*两*步（也可以说是通道）之遥的所有路口。我们会途经并标记出一条从*a*出发的开放通道；这样它应该会拥有两个标志。假设我们这次最后所在的路口是b点。那么现在就要遍历（和标记）从b点出来的所有开放通道，以确保当它们最终遇到死胡同或已被访问过的路口时能被关闭。这一切完成之后便可原路返回。一旦回到了a点。我们就可以继续其它开放通道的探索了，直到其所有通道都带有了那两个标志。（这两个标志说明了我们已经沿着通道对它完成了两步走的探索。）

下面我们将问题扩展到*n*步[[19]](#footnote-19)。于是，我们已经访问了图中*n*-1步远的所有路口，因此从*a*点出发的所有开放通道就会相应地拥有*n*-1个标志。而与*a*点相邻的所有路口（例如我们之前访问过的b点）出发的开放通道则将拥有*n*-2个标志。后续情况以此类推。为了访问从起点到距离为n范围内的所有路口，我们可以直接先移动到a点的所有邻居节点（例如*b*点）上，然后按我们的方法为其加上标志，并且以同样的处理方式访问接下去*n*-1距离内的所有路口（将其可行性设置为归纳前提）。

至此，我们会再次意识到单靠上面这样对局部信息做些记账式的处理会有点麻烦（而且相关的解释也会有点混乱）。然而，就像Trémaux算法与递归版DFS之间存在着很密切的关系一样，Ore的方法也可以被定制成一种更适合于我们计算机科学思维的方法，其结果是一种被称之为*迭代深度的深度优先搜索*的算法，简称IDDFS[[20]](#footnote-20)，它是一种运行深度受到限制的，有限深度递增的DFS算法。

在清单5-8中，我们给出了一个相对简单的IDDFS实现。其中维持了一个名为yielded的全局性集合，该集合由第一时间已被探索过的以及因此而被递交（yield）的节点组成。其内部函数recurse则基本上是一个深度限制为d的递归版DFS算法。如果该限制值为0，就不会有更多的边被递归探索了。反之，该递归调用d-1的深度限制被执行。在iddfs函数中，主循环会经历从0（这里只有访问和递交操作开始的出发点）到len(G)-1（可能的最大深度）的每个深度限制。如果在其到达这一限制深度之前，所有节点就已经被探索完毕了的话，该循环就会提前被跳出。

|  |
| --- |
| **请注意：**如果我们在探索无边界图结构（例如某种无限状态空间）的过程中，要寻找某一特定节点（或某一类节点），就只需要持续尝试更大的深度限制，直至找到自己想要的结果即可。 |

**清单5-8.** 迭代深度的深度优先搜索

def iddfs(G, s):

yielded = set() # Visited for the first time

def recurse(G, s, d, S=None): # Depth-limited DFS

if s not in yielded:

yield s

yielded.add(s)

if d == 0: return # Max depth zero: Backtrack

if S is None: S = set()

S.add(s)

for u in G[s]:

if u in S: continue

for v in recurse(G, u, d-1, S): # Recurse with depth-1

yield v

n = len(G)

for d in range(n): # Try all depths 0..V-1

if len(yielded) == n: break # All nodes seen?

for u in recurse(G, s, d):

yield u

当然，我们对于IDDFS的运行时间是不太能确定的。与DFS不同，该算法通常会多次遍历许多边和节点，所以我们恐怕远远不能确保它是一个线性时间的算法。如果相关图结构本身就是一条路径，而我们又是从该路径的其中一个端点开始执行IDDFS的话，它的运行时间当然是*平方级*的。但这个例子本身就是不正常的；只要我们的遍历树稍微有一点分叉，其大部分节点就都会出现在该树的底层（就像第3章中的淘汰赛制那样），所以对于许多图结构来说，它的运行时间应该是线性级或接近于线性级的。

如果试着在一个简单的图结构上执行该iddfs函数的话，我们会发现其节点的生成顺序是从起点的最近处开始，一直延伸到最远处的。先返回所有距离为*k*的，然后再是距离为*k*+1，依次类推下去。如果它要找的是某实际距离内的节点，我们可以很轻松地在iddfs函数中执行一些额外的记录操作，以便把节点和它的距离一块儿递交。除此之外，另一种方式就是在函数中维护一个距离表（这非常类似于在DFS算法中，我们在节点被探索以及完成处理时所做的操作）。事实上，我们也确实可以在遍历树中用一个字典记录距离，而另一个字典记录各节点的父节点。这样我们才能从这些距离中检索出最短路径。但如果我们目前只专注于路径问题的话，其实倒也不必为了纳入这些额外信息而去修改iddfs函数，我们可以直接构建出*另一种*遍历算法：*广度优先搜索算法*（*breadth-first search，简称BFS*）。

BFS算法的遍历其实要比IDDFS容易得多，我们只需要在一般性遍历框架（见清单5-6）中采用*先进先出*（*first-in first-out*）的队列类型即可。事实上，它与DFS算法唯一显著的区别就是将LIFO替换成了FIFO。结果就是先被访问到的节点会率先完成探索，这使得我们能像在IDDFS算法中那样对图结构进行逐层探索，不过，好处是这次我们不再需要对任何边和节点进行多次访问了，进而也就能恢复对于该算法线性级性能的保证了。[[21]](#footnote-21)

**清单5-9.** 广度优先搜索

def bfs(G, s):

P, Q = {s: None}, deque([s]) # Parents and FIFO queue

while Q:

u = Q.popleft() # Constant-time for deque

for v in G[u]:

if v in P: continue # Already has parent

P[v] = u # Reached from u: u is parent

Q.append(v)

return P

正如您在清单5-9中所看到的那样，bfs函数与清单5-5中的iter\_dfs函数非常类似。我们只是将其中的list换成了deque，而且，这次我们只需在遍历过程中持续跟踪途经节点的父节点（它们都在字典P中）即可，不需要再去记录那些已访问节点了（之前的集合S）。现在，如果您想要获取从a到u的路径的话，只要直接在队列P中“往回倒”就行了。

>>> path = [u]

>>> while P[u] is not None:

... path.append(P[u])

... u = P[u]

...

>>> path.reverse()

当然，我们可以自由选择在DFS中使用这种父节点字典，也可以在BFS中选择用yield来迭代遍历的节点集。在习题5-13中，我们将会让您通过修改这段代码来找出相关的距离（而不是路径）

|  |
| --- |
| **提示：**浏览网页是一种将DFS和BFS可视化的方式。当我们接连点过一些链接后，通过“后退（Back）”按钮回到某个自己曾到过的页面时，我们使用的就是DFS。在这里，回溯操作就与“撤销”按钮的作用有些类似了。而BFS则更像是我们用新窗口（或标签页）在后台打开每一个链接，然后在看完各页面后依次关闭其窗口。 |

只有在一种情景中，IDDFS的情况是好于BFS的：即当搜索目标是一个巨大的树结构（或某种“形态结构”类似于树的状态空间）时。由于该结构不存在环路问题，我们不用记录自己访问过的节点，这也就意味着IDDFS算法只需要存储从起点出发的单条路径即可[[22]](#footnote-22)。而在另一方面，BFS则必须要在内存（即队列）中持有全体的前沿节点，并且只要该树上带有一些分叉，随着与根节点之间距离的增大，前沿节点就会指数式地增长。换句话说，在上述这些情况下，IDDFS的内存用量更为节省，同时又很少出现或者根本就不出现渐进性的性能下降。

|  |
| --- |
| **“黑盒子”专栏之；deque** |
| **我们之前曾几次简单提到过，Python的list类型能胜任stack的角色（一种LIFO的queue），但并不胜任（FIFO）queue的角色。尽管它的append操作时间是常量级的（至少对于许多这种append操作的时间求平均值是如此），但其从前端pop（或insert）的操作时间则是线性级的。而在BFS这类算法中，我们要使用的是一种*两端式队列*（*double-ended queue*, 或*deque*）。所以，这种队列通常都是用链表（其前后端的追加及pop都属于常数级操作）或者所谓的环状缓冲区来实现的——后者（即circular buffers）是一种始终对其第一个（首端）和最后一个（尾端）元素的位置（下标）保持跟踪的数组，如果该数组首尾两端有任何一个元素越过了其尾端，我们都会将其“导向”另一端，并用取模（%）运算符计算出其实际索引位置（故而我们说它是环状的）。如果该数组已经完全被填满了，那我们就需要为其重新分配一个更大的容量，就像我们在动态列表中所做的那样（详见第2章中相关的“黑盒子”专栏）。**  **幸运的是，Python本身在其标准库的collections模块中就提供了deque类。该类除了在*右端*所执行的append、extend、pop这些方法外，它在*左端*也相应提供了一组名为appendleft、extendleft、popleft的方法。在内部，deque类的实现是一个块空间的双向链表，其中每个独立元素都是一个数组。尽管它与纯独立元素组成的链表的效率近乎相等，但这样做能降低开销并且在实践中更为高效。例如，如果它是一个普通的链表，表达式d[k]将要求我们遍历队列d中的前k个元素。然而，如果deque对象中的每个块空间中都有b个元素，那么我们就只需要遍历k//b块空间就够了。** |

### 强连通分量

在DFS、IDDFS、BFS这些遍历算法在其各自领域发挥其作用的同时，我们之前也提到过，在一些*别的*算法中，遍历扮演了其底层结构的角色。在未来的几章中，我们还会继续关注这些角色，但在本章结尾，我们要向您介绍一个经典问题——一个有相当难度的问题，但只要您对基本的遍历有一些理解，应该就能很优雅地解决掉这个问题。

该问题就是要找出*强连通分量*（*strongly connected components*，SCCs），有时也直接简称为*强分量*（*strong components*）。SCCs是一种有向化的连通分量，我们在本章开头部分已经介绍过后者的寻找方法。在忽略目标图的边方向（或其本身就是一个无向图）的情况下，一个图结构的连通分量应该就是能让它里面的所有节点能彼此到达的最大子图。而要想找出*强*连通分量的话，我们就需要对这些边的方向进行跟踪了，所以SCCs应该是一个能让有向路径上所有节点能彼此到达的最大子图。

（图）

**图5-7.** 一张有向图中的三个SCC（高亮部分）：A、B、C。

下面，我们来仔细思考一下图5-7中的这个图结构。它与本章开头的那张图（图5-1）非常类似：即尽管图中存在着一些额外的边，但构成这个新结构的SCC节点与原无向图中的连通分量是相同的。而且正如上图所示，在这些强分量（高亮部分）中，所有节点间都是可以彼此到达的，但如果我们试着再往其中加入任何一个节点，这一属性就不成立了。

现在，假设我们要在该图结构上执行DFS算法（为了确保能涵盖到整个图结构，该算法可能会选择几个不同的遍历起点），您会如何评估遍历完强分量A和B中所有节点所需的时间呢？如您所见，图中从A到B有一条边可走，但从B到A却没有路径可走。这当然会影响遍历的完成时间。至少我们可以肯定，A将会晚于B完成遍历，也就是说，A的最后完成时间一定比B的最后完成时间要晚一些。如果我们仔细观察一下图5-7中的情况，其中的缘由是显而易见的。如果遍历从是B中开始的，那么我们是不可能到达A的，所以B的遍历可能甚至在A的遍历*开始*（更不用说*完成*了）之前就已经完成了。但如果遍历从A开始的话，我们很清楚自己不会被局限在A内部（其中的任何节点都能彼此到达），所以在其完成遍历之前，我们还是会*先*移动到B以及其它需要遍历的区域（在这里就是C），待这一切完成之后才会回溯到A。

事实上，在一般情况下，只要任何一个强分量X中有一条边通往另一个强分量Y，X中最后完成遍历的时间就一定会晚于Y中最后完成的时间。其证明过程与我们之前那个例子是相同的（请参考习题5-16）。根据这些事实我们可以得出以下的结论：我们无法采用从B到A的遍历方式——事实上，SCC的工作方式原本就是如此，因为这些SCC合起来就构成了DAG！因而，只要X中有一条边通往Y，Y中就不可能会有一条边能通往X。

对于图5-7中的高亮部分，如果我们将它们视为单一“超级节点”的话（并保留这三部分之间原有的边线），最终会得到这样一个图结构（我们称其为SCC图）：

（图）

这显然也是个DAG，但为什么这种SCC图中*始终会*没有环路呢？因为只要我们假设SCC图中存在环路。那么，这就意味着我们可以在两个SCC之间反复来回，您能看出其中存在的问题吗？是的，没错，这就等于第一个SCC中的每个节点都可以到达第二个SCC中的每个节点，反之亦然。这样的话，事实上该环路上的所有SCC就可以被合并成一个*单一*的SCC。这显然与我们最初的假设是矛盾的。

现在，让我们将该图中所有的边线都翻转一下。这样做并不会影响这些节点所属的SCC（具体见习题5-15），但会影响SCC图本身。具体到上面的例子来说，就是现在A中没有出路了。所以如果我们还是在遍历完A后，再在B中开始新一轮遍历的话，是不可能从中脱身的，这样就只剩下C了。而且……等一下，我们刚才似乎已经完成强分量的查找了，不是吗？如果纯粹从一般性思维运用来说，我们总得需要先从原图结构中没有任何入边的SCC（也就是翻转后没有出边的那个SCC）入手。基本上，我们正在查找的就是该SCC图的拓扑排序中的第一个SCC（然后才是第二个，并依次找下去）。下面请大家回顾一下，根据我们最初对DFS的推理：起点的遍历处理将会在*最后一刻完成*。事实上，如果我们根据其最后完成时间的递减顺序来选择其最终遍历的起点的话，同时也就等于确保了一个SCC的完全探索，因为这些被翻转的边线已经阻断了我们移往下一个SCC的路径。

要完全跟上这个推理过程确实会有一点难度，但这并不等于其主要思维都很难。如果A中有一条边通往B，那么A的遍历（最终的）完成时间将会晚于B。如果我们是根据最终完成时间的递减顺序来选择（第二次的）遍历起点的话，这就意味着我们要先于B访问A。而如果在此刻*翻转*图中所有边线的话，虽然我们依然可以遍历A中所有节点，但再也不能由此移动到B中了，这使得我们一次只能单独探索一个SCC。

下面我们来对该算法做个概述。需要注意的是，这里将不再需要“手动式”运用DFS算法，并根据完成时间来对相关节点进行反向排序了，我会直接调用dfs\_topsort函数，这才是我们该做的工作。[[23]](#footnote-23)

1. 在目标图结构上执行dfs\_topsort函数，并产生一个序列seq。
2. 翻转图中所有的边线。
3. 从seq中（根据顺序）选择一个起点进行完全遍历。

其具体实现如清单5-10所示：

**清单5-10.** Kosaraju的查找强连通分量算法

def tr(G): # Transpose (rev. edges of) G

GT = {}

for u in G: GT[u] = set() # Get all the nodes in there

for u in G:

for v in G[u]:

GT[v].add(u) # Add all reverse edges

return GT

def scc(G):

GT = tr(G) # Get the transposed graph

sccs, seen = [], set()

for u in dfs\_topsort(G): # DFS starting points

if u in seen: continue # Ignore covered nodes

C = walk(GT, u, seen) # Don't go "backward" (seen)

seen.update(C) # We've now seen C

sccs.append(C) # Another SCC found

return sccs

如果我们尝试在图5-7上运行一下上面的scc函数，应该就会得到{*a*, *b*, *c*, *d*}、{*e*, *f*, *g*}和{*i*, *h*}这三个集合[[24]](#footnote-24)。另外，在调用walk函数时还需要注意一件事，我们这次启用了该函数的S参数，使其能回避掉之前的SCC。因为图中所有的边都是指向后方的，如果不加以禁止的话，它们都太容易成为遍历起点了。

|  |
| --- |
| **请注意：**有一个念头似乎很有吸引力，那就是我们选择不调用tr(G)来翻转图中所有边线，而改成在dfs\_topsort函数返回时将其序列按逆序排列（也就是按照完成时间的递增原则，而不是递减原则来选择遍历起点），但这显然是不能工作的（在习题5-17中，我们会要求您证明这一结论）。 |

|  |
| --- |
| **目标与剪枝** |
| **本章所描述的遍历算法基本上都是用来访问它们所能到达的每个节点的。但有时候，我们其实只是需要查找某个特定节点（或某一类节点），并且最好能尽可能多地忽略掉图中的细节。对此类操作，我们称之为*目标导向型*（*goal-directed*）搜索，而对于这种搜索带来的，遍历过程中忽略潜在子树的行为，我们称之为*剪枝*（*pruning*）。例如，如果已知自己要找的节点位于起点的k步以内的话，那我们只需要在执行遍历算法时将其深度限制设定为k即可，这就是一种剪枝形式。同样地，二分搜索或搜索树的搜索算法（我们将在第6章中具体讨论它们）也都要用到剪枝。相较于整个搜索树的遍历来说，我们在这里只需要访问可能存在被查找值的那部分子树就可以了。而根据这些树的构造，我们往往每进一步都能丢弃掉其同级的大部分子树，并以此衍生出了高效算法。**  **对自身目标的充分了解能有助于我们选出最有前途的优先方向（这也就是所谓的*最优先搜索*）。第9章将会讨论的A\*算法就是其中一个例子。如果您此刻正在某个解决方案空间中搜索，其实您也可以先考虑一下如何评估出其中最有*前途*的方向（也就是看看跟着这条边走下去能得到的最佳解决方案有多好）。通过忽略一些对目前最佳方案的改善没有帮助的边线，我们可以大大地提升相关算法的速度。这种方法叫做*分支定界法*（*branch and bound*），我们将会在第11章中对此加以具体讨论。** |

### 本章小结

在本章，我们首先向您展示了一些在图结构中移动的基本方法，其中既包括有向图也包括无向图。这种遍历思维（直接或从概念上）为本书后面章节中的许多算法，以及我们今后可能遇到的其它算法奠定了基础。接着，我们列举了几个迷宫遍历的算法实例，譬如由Trémaux或Ore提出的算法，当然，这些算法的大体上是作为构建对机器友好的方法的出发点而提供的。通常情况下，一个图结构的遍历过程主要包括：维护一个用来存放待探索节点的to-do列表（一种队列），并从中去除我们已访问过的节点。最初，该列表中只有遍历的起点，在之后的每一步，我们都会访问（并去除）其中一个节点，并将其邻居节点加入到该列表中。而在该列表中，项目的（调度）顺序则很大程度上决定了我们所实现的遍历类型，例如，如果您采用了LIFO队列（即stack类型），那么执行的就是深度优先搜索（DFS），而要是我们采用的是FIFO队列，执行的就是广度优先搜索（BFS）了。其中，DFS是一个相对直接的递归式遍历，假设我们找出了各个节点从被探索到完成遍历的时间，那么后代节点所用的时间就一定落在其先辈节点所用的时间之内。而BFS则具有被用来在一个节点到另一个节点之间寻找（未加权）最短路径的作用。另外，我们还介绍了一种DFS变体，称为*迭代深度的DFS*，虽然它也具有BFS的作用，但其在大型树结构搜索中所起到的作用会更大一些，譬如我们将会在第11章中讨论状态空间时用到这种算法。

如果相关图结构是由若干个连通分量组成的，我们可能需要针对各分量分别重启遍历程序。要做到这一点，我们可以对图中所有节点进行枚举，跳过那些已访问过的节点，并从别处启动一次遍历程序。而在有向图中，这种方式即使在图结构整体连通的情况下有可能仍是必要的，因为图中一些边的方向可能会成为我们到达某些节点的阻碍。为了找出一个有向图中的强连通分量（在这种分量中，图中的路径可以支持其中任何两个节点之间的彼此到达），我们可能需要经历一个相对复杂的过程。我们这里讨论的是Kosaraju算法，它会先找出各节点最后完成遍历的时间，然后根据其完成时间的递减顺序来选择遍历的起点，在目标图的*翻转图*（即翻转图中所有边的方向）中进行遍历。

### 如果您感兴趣……

如果您喜欢遍历，也不用担心，我们很快就要去做更多的遍历了。另外，您也可以通过Cormen等人的书（请参考第1章的“参考资料”部分）来了解更多关于DFS、BFS以及SCC算法的细节问题。如果您对于查找强连通分量也有兴趣的话，可以去参考一下本章“参考资料”部分所列出的Tarjan与Gabow（或者更确切地说，是Cheriyan-Mehlhorn/Gabow）各自提出的算法。

### 练习题

* 1. 在清单5-2的components函数中，seen节点集合是一次性更新整个分量的。但这里的另一个选项是在walk函数中逐个将节点添加进去。您觉得这两种选项的区别在哪里？还是它们根本没有什么不同？
  2. 在面对一个每个节点的度（degree）都是偶数的图结构时，您会用何种方式寻找其中的欧拉环路？
  3. 如果在一个有向图中，每个节点的入度与出度是相同的，我们就会找到一条*有向*的欧拉环路。这其中的缘由是什么？您如何看待它与Trémaux算法之间的关联？
  4. 在图像处理中，有一个被称之为*色彩填充*（*flood fill*）的基本操作，它会在某一区域内填充一种单色。在一些绘图应用程序（譬如GIMP或Adobe Photoshop）中，它往往就是油漆桶这样的工具，您觉得这种填充操作应该要如何实现呢？
  5. 在希腊神话中，Ariadne在帮助Theseus战胜了牛头人之后，给了他一个羊毛线球，以便让他能逃离迷宫。但如果Theseus在他来的路上，进迷宫之前忘记了桩住线头，而且只在他彻底迷路时才想起这个球——那么，接下来他该如何使用这个线球呢？
  6. 在递归版的DFS中，回溯操作会发生在我们从一个递归调用返回的时候。但迭代版的DFS中的回溯操作又是在何处发生的呢？
  7. 请写一个非递归版的，且能确定“处理完成”时间的DFS算法。
  8. 在（清单5-7的）dfs\_topsort函数中，每个节点上都会启动一次递归版的DFS（尽管它会在遇到已访问节点时立刻被终止）。那么我们要如何确定这是一次有效的拓扑排序呢？它即使在任意选择排序起点的情况下也是有效的吗？
  9. 写一个有钩子（hook，即可覆写函数）版本的DFS算法，使用户能实现自定义的前序处理和后序处理。
  10. 请证明当（且仅当）DFS没有找到后向边时，被遍历的图结构中是不存在环路的。
  11. 如果您打算用其它遍历算法而不是DFS来查找一个有向图中的环路时，您会面对什么挑战？为什么在无向图中不需要面对这些挑战？
  12. 如果我们在无向图中执行DFS算法，我们将不会遇到任何前向边或交叉边。这是为什么？
  13. 请写出一个版本的BFS，该版本要找的是起点到各个节点的距离，而不是实际路径。
  14. 正如我们在第4章中所提到的，如果我们能将目标图中的全体节点分成两组，而且没有任何一对邻居节点存在于同一个组中，该图就被称为二分图。这个问题的另一种思考方式是将目标图中的节点分别涂上黑色或白色，没有任何一对邻居节点的颜色是相同的。请展示对于任意的无向图，您是如何找到这种二分法（或者双色法）的，当然，前提是确实存在这样一种划分。
  15. 如果我们翻转一个有向图中的所有边线，其中的强连通分量将保持原样。为什么？
  16. 假设X、Y是同一个图结构G中的两个强连通分量，并且从X到Y至少存在着一条边。如果我们要在G上运行DFS算法（必要时可以重启调用，直至图中所有节点都被访问到为止），那么X中最终完成遍历的时间一定会晚于Y中的最终完成时间。这是为什么？
  17. 在Kosaraju算法中，我们是按照（由DFS在初始阶段得出的）最后完成时间的递减顺序来选择最终遍历的起点的，并且我们是在翻转图（即翻转所有的边）中执行遍历操作的。为什么我们不能在原图中用其完成时间的递增顺序来做这件事？

### 参考资料

* Cheriyan, J. and Mehlhorn, K. (1996). Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15(6):521-549.
* Littlewood, D. E. (1949). The Skeleton Key of Mathematics: A Simple Account of Complex Algebraic Theories. Hutchinson & Company, Limited.
* Lucas, É. (1891). *Récréations Mathématiques*, volume 1. Gauthier-Villars et fils, Imprimeurs-Libraires, second edition. [Available online at [http://archive.org]](http://archive.org)
* Lucas, É. (1896). *Récréations Mathématiques*, volume 2. Gauthier-Villars et fils, Imprimeurs-Libraires, second edition. [Available online at [http://archive.org]](http://archive.org)
* Ore, O. (1959). An excursion into labyrinths. *Mathematics Teacher*, 52:367-370.
* Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2): 146-160.

1. 本章标题“偷师”自Dudley Ernest Littlewood的著作：《*The Skeleton Key of Mathematics*》。 [↑](#footnote-ref-1)
2. 译者注：《龙与地下城》（Dungeons & Dragons）是一款奇幻背景的角色扮演游戏，而且还是世界上第一款被商业化的桌上型角色扮演游戏。 [↑](#footnote-ref-2)
3. 如果您不是一个游戏玩家，也可以把以下场景想象成办公楼、梦想中的家，或其它任何合您心意的地方。 [↑](#footnote-ref-3)
4. 往后，我会将dict当作是邻接集的默认表示类型。当然了，许多算法用了第2章中介绍的其他表示类型也一样可以工作得很好，将一个算法中的邻接集类型重写成其它不同的表示形式也不是一件太难的事。 [↑](#footnote-ref-4)
5. 在本章，除了IDDFS（偶尔例外），几乎所有遍历算法的运行时间都是如此。 [↑](#footnote-ref-5)
6. 译者注：该书在维基百科中文版中被译为《娱乐数学》，其中收录了不少数学名题，共4卷，原版应为法语书。译者在翻译本书时尚未发现有中译本。 [↑](#footnote-ref-6)
7. 译者注：莱昂哈德·欧拉（1707年4月15日－1783年9月18日）是一位瑞士数学家和物理学家，近代数学先驱之一，他一生大部分时间在俄国和普鲁士度过。 [↑](#footnote-ref-7)
8. 译者注：由于这些都是具有固定译名的图论术语，所以这里就直接使用Euler的中文译名了。 [↑](#footnote-ref-8)
9. 译者注：其实，哈密顿爵士的主要成就来自物理方面，他重新表述了牛顿力学，为后来的量子力学奠定了重要的基础。 [↑](#footnote-ref-9)
10. 嗨，牛顿在苹果树下的故事也是虚构的！ [↑](#footnote-ref-10)
11. 从a点开始跟踪您的路径的话，您会得到以下节点序列：a、b、c、d、e、f、g、h、d、c、i、j、i、k、i、c、b、l、b、a。 [↑](#footnote-ref-11)
12. 当然了，如果您面对的是一个真实世界中的迷宫，这个递归版本很难有实际用处。 [↑](#footnote-ref-12)
13. 而且这样下去，一个洞穴探险家可能会变成穴居人。 [↑](#footnote-ref-13)
14. 其实即使在夜晚散步，人们也一样有可能会走出环路来。根据美国陆军的研究，人们通常会因为某种原因更喜欢往南走。当然了，如何您执意要进行一次完全遍历，这种策略恐怕帮不上什么忙。 [↑](#footnote-ref-14)
15. 当然，这只是我个人的翻译（译者注：该书原文为法语）。 [↑](#footnote-ref-15)
16. 即使您的鞋子很干净，只要有办法在各个入口和出口留下清晰的标志（譬如用粉笔），我们一样可以完成这个流程。这里的重点是当我们进入旧路口并马上开始回溯时，要留下两个标志。 [↑](#footnote-ref-16)
17. 事实上，术语“回溯”被用作递归遍历与深度优先搜索的同义词。 [↑](#footnote-ref-17)
18. dfs\_topsort函数也可以按照遍历完成时间的递减顺序来对通用图结构中的节点进行排序，本章稍后讨论强连通分量时会用到这一功能。 [↑](#footnote-ref-18)
19. 换句话说，下面将用归纳法来思考。 [↑](#footnote-ref-19)
20. IDDFS与Ore的方法并不是完全等效的，因为它不会用与其相同的方式来标志相关边线的关闭状态。要向它增加这种标志是可能的，通常会被认为是一种剪枝形式，这方面我们稍后会详细讨论。 [↑](#footnote-ref-20)
21. 从另一方面来说，这也意味着我们可能会以一种在真实迷宫中无法实现的方式在节点之间来回跳跃。 [↑](#footnote-ref-21)
22. 要省下这些内存空间，我们必须从代码中去掉集合S。由于被遍历的对象是一棵树的缘故，这样做不会带来任何麻烦（那种遍历环路的麻烦）。 [↑](#footnote-ref-22)
23. 这有点是像是在作弊，因为我们是在一个非DAG上执行拓扑排序的。但是，这里的想法只是要获取各节点完成遍历时间的递减顺序，而这正是dfs\_topsort函数所在做的事，并且会在线性时间内完成。 [↑](#footnote-ref-23)
24. 事实上，walk函数返回的是各个强分量的遍历树。 [↑](#footnote-ref-24)