



下载APP



10 | 搜索算法：一起来写一个简单的爬虫？

2022-01-01 黄清昊

《算法实战高手课》

课程介绍 >



讲述：黄清昊

时长 17:30 大小 16.03M



你好，我是微扰君。

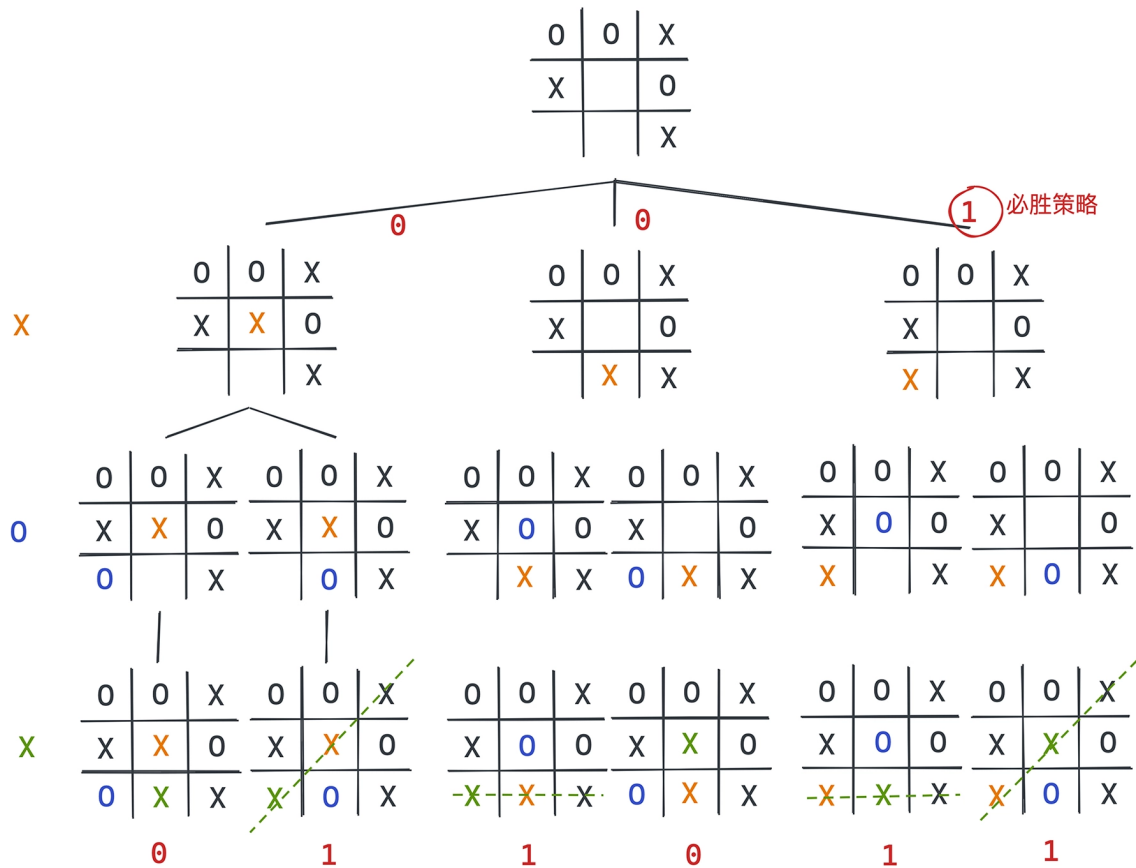
你玩过井字棋的游戏吗？在一个九宫格中，双方轮流用 X 和 O 占领一个格子，某一方的 O 或者 X 三个连成一线时即可获胜。

这样一个简单井字棋的游戏，如果要让你自己写代码实现一个 AI，你会怎么做呢？怎么把博弈过程清晰地表示出来呢？

领资料



实际上，许多博弈类游戏的过程，我们都可以用树来表示。根节点就是棋盘为空的状态，终点就是各个棋下完的状态，这样的树也被称为博弈树。下图是井字棋某个局面 3 步内的树状展示：



一般来说，对弈双方在做的事情，其实就是找到这棵树上对于自己最优的一种落子方式，使得之后的每条路径，自己都有必胜或者必不败的策略。**如果你想要找出一个 AI 策略，最暴力的方式就是直接遍历每一种情况，找到最优的下法，这就是一个典型的搜索问题了。**

事实上，这类博弈的游戏要么是先手必不败，要么是后手必不败，所以对全空间的搜索一定可以写出一个无敌 AI 的，对证明感兴趣的同学可以去搜索“策梅洛定理”了解。

如果暴力遍历，有多少种情况呢？相信你也发现了，就是这么一个简单的井字棋小游戏，终局的数量非常多，达到了 255168 种。我们可以这样来简单地估计它，第一步有 9 种下法，第二步有 8 种下法，显然通过排列组合的知识，占满棋盘一共有 $9! = 362880$ 种下法，当然还需要去掉一些中间获胜不应该继续进行对弈的情况。

这本身是一道挺有意思的数学问题，也可以通过写代码更快地计算出来，留给你作为课后作业。当然这个数字还不算天文数字，尚且在计算机的处理范围之内，如果我们稍微把游戏的复杂度提升一下，比如围棋，还能通过暴力搜索的方式得到一个优秀的 AI 吗？

我们知道，围棋盘有 19×19 个落子点，所以刚开始的每一步可能都有接近 361 个选择，那整体的情况可能接近 361！种。这是一个天文数字，在现在的计算机架构下，直接计算和存储这样的问题是不可能的。所以我们**想要写出一个靠谱的围棋 AI，就需要采取一些新的策略，只选择部分分支进行遍历，从中找出一个比较好的方案。**

对于人类而言这个过程就是依靠经验，对于 AI 来说，就是依托于数据，你从 AlphaGO 核心算法的名字“蒙特卡洛搜索树”中，就可以看出来，这本质上还是一个搜索的问题，只不过人类棋手和 AI 都采用了比较高明的搜索策略。

我们今天就不讲那么进阶的内容了，就讲一讲平时常用的广度优先搜索算法 BFS 和深度优先搜索算法 DFS。

它们是两种最常见的暴力搜索算法，在面试中也相当常见，前者的实现需要用到我们之前讲解的队列这一数据结构，后者则是递归思想最常用的场景之一。在工程中它们也发挥着巨大的作用。比如，DFS 在前端开发中 DOM 树相关的操作里就非常常见，我们可以用它来实现对 DOM 树的遍历，从而对比两颗 DOM 树的差异，这就是 React 中虚拟 DOM 树算法的关键点之一。

BFS 和 DFS

BFS 和 DFS，作为两种最暴力、也相当常用的搜索策略，**最大的特点就是无差别地去遍历搜索空间的每一种情况，因此但凡是可以抽象成图上的问题，基本上都可以考虑用 BFS、DFS 去做。**只不过效率可能不是最优的，所以我们也常常称之为暴力搜索算法，在各大刷题网站题解区中，你应该常常能见到“暴搜”这样的关键词，说的一般就是 DFS 和 BFS 这两种算法。

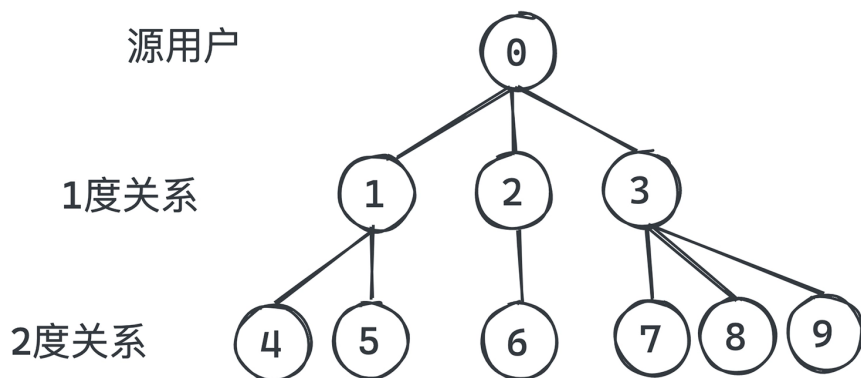
所以，在爬虫这样本来就需要无差别遍历全部空间的场景下可以说是非常合适的了。至于 DFS 和 BFS 具体选择哪一种，我们可以结合一个具体的爬虫场景来分析。

如果让你手写一个爬虫，从豆瓣上爬取一个用户关注的所有用户，是不是很简单？只要直接遍历某个用户的关注者列表就可以了，除了需要处理一些鉴权和页面解析的问题，没有什么复杂的地方。

那我们升级一下挑战，爬取这个用户关注的人的所有关注的人，也就是和这个用户有二度关系的所有用户，你要怎么实现呢？如果不是二度，而是让你查找三度关系，也就是找出

需要三跳的所有用户，你的代码能否很简单地通过配置就完成这件事呢？

这其实就是一个非常适合用 DFS 和 BFS 解决的问题，因为它天然就是一个需要无差别遍历所有图上节点的问题。



你可以把豆瓣用户看成节点，用户之间的关注关系就是边，它们一起构成了一个复杂的社交网络。相信你也听过社交网络中的“六度分割理论”，说的就是世界上任何一个人和你之间的距离不会超过 6 度，描述了社交网络的小世界特性。这种网络关系也是许多人在研究的。

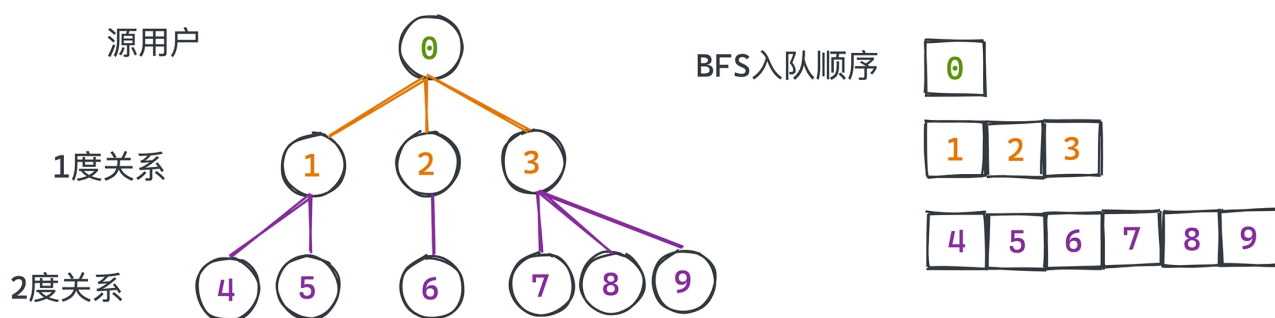
BFS 实现思路

好我们先来使用 BFS 解决这一问题。我们优先去遍历所有到源用户距离为一度的用户，然后再遍历这些用户的邻居，用层层深入的方式进行搜索。广度优先搜索，其实是一个很直观的定义，我们把对应到图上的搜索顺序画出来，就很清晰了。

看简化的情况，假设我们搜索的源用户为图的 0 节点，一度关系包含 3 个节点，二度关系包含了 6 个节点，每条连边都是一个关注关系。那我们基于广度优先搜索策略遍历时，就会按照标号顺序进行遍历。

为了在代码中实现这样的遍历效果，求出所有和 0 节点构成两度以内关系的用户，我们就需要借助之前学习的“队列”了。

因为广度优先搜索是由源点向外逐层推进的，每遍历下一层的时候，我们都需要用到上一层的节点，所以我们需要一个容器记录每一层的元素，并依次遍历，先进先出的队列就可以帮助我们很好地解决这个问题。



我们首先把遍历的初始节点加入 queue 中，然后循环读取 queue 中的元素，每次读出一个元素，就把它的所有相邻节点都放入新队列中，直到目前队列为空，就代表我们遍历完了所有的元素。队列 FIFO 的特性保证了，下一层的元素一定会比上层的元素更晚出现。

当然我们经常需要设置自己的搜索退出条件，比如在最短路径问题中，我们并不需要遍历所有的路径，当搜索到终点的时候其实就可以退出了；在我们的例子中，我们的退出条件也不是遍历完整个社交网络，遍历到第二度关系就可以结束了，因此我们还需要在代码中记录当前遍历的层数。

另外，有时候需要对插入队列中的元素做一些判重，防止重复的搜索。

在搜索最短路径或者求几度关系所有用户的情况下就很有用，因为重复的节点已经没有必要再搜索了；如果不这样做，甚至可能导致你的搜索永远无法结束，比如在图有环的情况

下。

实现


下面我们试着写一下，就用爬虫常用的脚本语言 Python 来实现这次的代码。

解释一下几个关键的变量。

degree 就是广度优先搜索中我们用于记录搜索层数的变量。为了每一次出队的时候，把一层的元素全部出队，在遍历中我们采用了两层 while 循环，这样内层循环结束后，我们就可以保证当前层的元素已经全部被访问，也可以将 degree 进行自增操作。

之所以开了一个新的变量 next_degree_urls 也是同样的道理，当前层的邻居不能干扰到这一层的出队操作，所以我们将邻居们放到一个新的队列中；内层循环结束后，再将已经为空的队列更新为 next_degree_urls。

而类型为 set 的变量 res，除了记录所有的用户主页，也起到了判重的作用；如果已经出现在 res 集中，说明我们已经遍历过这个用户主页了，不需要再遍历一次了，所以在循环中直接通过continue跳过。

 复制代码

```
1      # 这里我们需要一个合适的 douban html parser
2      def crawl(self, startUrl: str) -> List[str]:
3          urls = deque()
4          urls.append(startUrl)
5          res = set()
6
7          degree = 0
8          N = 2
9
10         while urls:
11             # 遍历层数超过N层，停止遍历
12             if degree > N: break
13             # 用于记录下一层的节点
14             next_degree_urls = deque()
15             # 遍历当前层
16             while urls:
17                 u = urls.popleft()
18                 if u in res: continue
19                 for url in doubanHtmlParser.getFollowings(u):
20                     next_degree_urls.append(url)
21                 res.add(u)
```

```
22
23         urls = next_degree_urls
24         # 当前层元素全部出队；进入下一层遍历，记录遍历层数的变量加1
25         degree = degree + 1
```

当然，既然是爬虫，我们肯定是需要对网页进行解析的，就用一个 `getFollowing` 函数表示这个过程，做的事情就是输入一个用户主页的 URL，返回该用户关注的其他好友的个人主页。

大致实现思路就是，通过网络请求库获取指定 URL 的 HTML 文本信息，从中解析出表示用户关注好友列表的部分，一般列表中的每个元素都会指向该好友的个人主页，我们把相关的 href 标签里的 URL 解析出来即可。

好了，这样我们就用基于队列的广度优先搜索策略完成了一个简易的爬虫，感兴趣你可以自己补全 HTML 解析器的代码，完整实现一下。

DFS 实现思路

再来用 DFS 解决这个问题。深度优先搜索，就不会再像广度优先搜索那样严格由内而外逐层推进了，它和棋手下棋的思路其实会更像一点，我们就用下棋举个例子。

假设下棋的时候，当前局面可以有若干个落子点，棋手一般会先顺着其中一个落子点在脑海中模拟若干步，发现某一步不行，我们回溯到分叉点，再看一下其他选择；最终遍历完当前选择的落子点的各种局面之后，再依次进行其他落子点的判断，直到选出一种比较优的策略。

画成图对比一下，你会更直观地感受到两者的区别，同样用刚刚假想的豆瓣用户关注关系图来举例：



极客时间

上图的数字，表示深度优先搜索在同样的关系图中的遍历顺序；可以看到相比于 BFS 的逐层推进，在 DFS 中，是一条条分支顺次遍历到终点再进行下一种尝试的，这也是深度优先搜索命名的由来。

实现

这种遍历方式也天然符合回溯法的适用场景，所以常规做法就是通过递归来实现。写成代码如下，关键就是 11-13 行的 for 循环，递归地对当前节点的每个子节点进行同样的 DFS 过程，细节你可以参考代码中的注释来理解，网页解析的逻辑和前面所说的是一样的：

复制代码

```
1  # 结果集 用于存放所有N度关系以内的用户
2  res = set()
3  N = 2 # 记录找N度关系以内的所有用户；N=2即找2度关系以内的用户
4  def crawl(startUrl, degree):
5      # 如果已经超过N度关系，我们不用继续遍历
6      if degree > N : return
7      # 如果已经搜索过，我们也不用继续搜索
8      if startUrl in res : return
9      # 将当前搜索的用户主页加入结果集
10     res.add(startUrl)
11     for url in doubanHtmlParser.getFollowings(startUrl):
12         # 遍历关注的所有用户，注意需要将度数增加1
13         crawl(url, degree+1)
```


DFS 的代码看起来明显要简短很多，这就是递归的威力。通过对自身的调用，很多时候，我们可以让代码变得非常简单。

时空复杂度

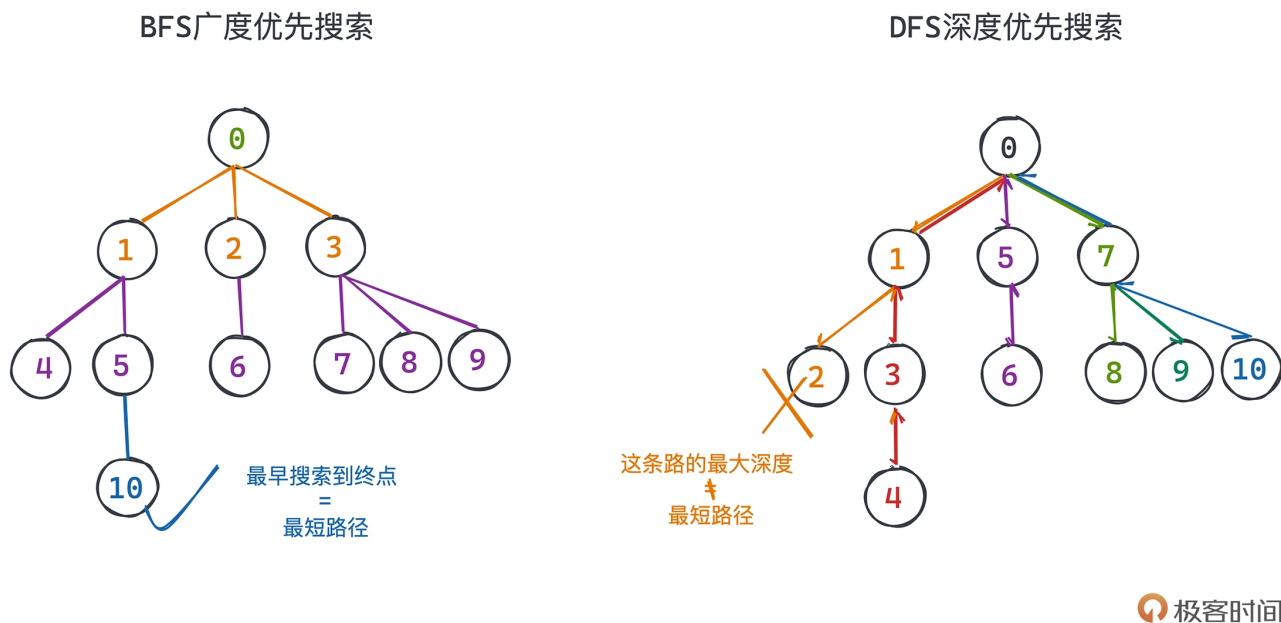
两种方法的时空复杂度都非常容易分析。

先说时间复杂度。我们做的事情就是遍历一次网络或者图中的每个节点，因为借助结果集判重后，即使在图中多次出现的节点，也只会入队一次或者被递归一次。那么假设图中有 V 个顶点， E 条边。

在 BFS 中，图中所有顶点入队一次、出队一次，每条边都会在边起点出队的时候被遍历一次，所以整体的复杂度为 $O(V+E)$ 。而在实际的复杂网络中， E 一般是远大于 V 的，所以可以近似认为复杂度为 $O(E)$ 。

在 DFS 中，是从起点出发递归遍历图，通过结果集判重，保证重复的节点不会被递归两次，从而每条边只会被遍历一次，整体复杂度为 $O(E)$ 。

所以时间复杂度上两者没有太大的差别。但是在一些求最短路径的场景下，比如求从当前位置走出迷宫的最短路径，就会有一定差异。这是因为 BFS 是从内到外逐层搜索，所以最早搜索到终点的时候，就对应了最短路径；因此找到终点我们就可以提前结束搜索过程。



而在 DFS 中，由于我们优先遍历每条路径的最大深度，即使找到了终点也只能说明找到了一条路径，这并不能保证这条路径是最短的，所以哪怕找到终点也不能结束搜索过程，需要遍历完整个搜索空间，找到所有可能的路径之后再从中选择一条最短的。

在搜索空间很大，但已知搜索路径不会特别长的情况下，DFS 可能会比 BFS 要慢很多，所以你要根据实际情况选择一种合适的算法。

当然在今天豆瓣爬虫的场景下，我们需要的就是遍历整个空间找到所有构成 N 度关系的用户，所以两种算法的时间复杂度其实没什么区别。

空间复杂度

再来看空间复杂度。

在 BFS 中，主要的空间复杂度就是 queue 和 res 所占用的大小。那这里的，res 本身并不是所有的 BFS 场景下都会需要的，因为我们并不一定需要返回所有遍历过的节点，可能只需要记录一个距离之类的值。

但是，在大部分的 BFS 下，我们是不希望重复遍历节点的，所以仍然需要一个类似于 res 的集合去标记所有经过的点。它所需要的最大空间和图中总结点数量 V 一致。而 queue

存储的就是图上的节点，其所占用的空间最大也不会超过 V 。所以 BFS 的空间复杂度是 $O(V)$ 。

在 DFS 中，所消耗的内存同样主要与 res 相关。因为虽然相比于 BFS，我们少了 queue 的内存消耗，但是也多了隐含递归中调用栈所消耗的空间。由于调用栈最多就是描述一条经过了所有节点的路径，其最大空间大小也不会超过顶点数量 V 。因而 DFS 的空间复杂度同样是 $O(V)$ 。

总结

作为两个相当常用的暴力搜索算法，BFS 和 DFS 比较适合用来解决图规模不大，或者本身就需要无差别遍历搜索空间的每一种情况的问题；这两者的时间空间复杂度是相当的。

而至于 DFS 和 BFS 具体选择哪一种，我也总结出一些自己的经验，供你参考。

BFS 因为是由内向外地毯式地搜索，所以首次搜索到目标位置的时候一定是源点到目标位置的最短路径，所以求最短路径类的问题往往可以用 BFS 解决。当然，这里的“最短路径”是有条件的，只有在图中所有边权重相等时首次搜索到的才是最短路径；另一类边权重不等的图上的最短路径求解问题我们之后会单独讲解。

而 DFS 实现起来比 BFS 更简单，且由于递归栈的存在，让我们可以很方便地在递归函数的参数中记录路径，所以需要输出路径的题目用 DFS 会比较合适。毕竟想用 BFS 实现相同的路径记录，除了需要在 queue 中记录节点，还需要关联到此节点的路径才可以，占用的空间比 DFS 高得多。

一般情况下我们都可以优先使用 DFS 实现，但这完全建立在我个人觉得 DFS 写起来更简单的前提下。而在需要求解路径本身的问题中，强烈建议你采用 DFS 作为搜索算法的实现。

课后作业

最后，我再给你留三个小作业。

1. 既然说是可以用 DFS 或者 BFS 写一个爬虫，希望你尝试补充一下爬虫中解析 HTML 和 HTTP 请求的逻辑。或者动手写一个你自己想写的爬虫感受一下，体会搜索算法在实战

- 中的应用。
2. 在搜索 N 度关系的所有用户时，如果我们希望同时把源用户和这些用户的关注关系记录下来，比如 A->B->C 就表示 A 关注了 B，B 关注了 C；更广泛地意义上来说就是让你记录搜索过程中的路径。你会怎么实现这个逻辑呢？
3. 前面提到的井字棋中，尝试用代码计算一共有多少种最终合法的局面呢？

欢迎你在留言区与我讨论。如果有收获也欢迎你转发给身边的朋友，邀他一起学习。我们下节课见~

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 09 | 二分：如何高效查询Kafka中的消息？

下一篇 11 | 字符串匹配：如何实现最快的grep工具

精选留言 (2)

写留言

 Daneil
2022-01-01
老师新年快乐！



 qinsi
2022-01-01
2. 力扣126
3. 力扣794



