深度优先搜索（DFS）算法

杨其臻

Oct 28, 2025

想象你身处一个迷宫，面前有多条岔路。你没有地图，只有一个目标：找到出口。这时，你有两种策略可以选择。策略一称为广度优先搜索，它要求你把所有岔路口都标记一下，然后一个个路口轮流去探索一步。策略二则截然不同，它让你选择一条路，一头扎到底，直到走到死胡同或者找到出口。如果走不通，就退回上一个岔路口，换另一条没走过的路继续深入。这种策略二正是我们今天要深入探讨的深度优先搜索（DFS）算法的核心思想。DFS 是一种用于遍历或搜索树或图数据结构的经典算法，它在计算机科学中有着广泛的应用。

1　DFS 的核心思想与算法流程

深度优先搜索（DFS）的官方定义是：一种用于遍历或搜索树或图数据结构的算法，它从根节点（或任意节点）开始，在回溯之前尽可能深地探索每一个分支。其核心思想可以概括为三个要素。第一是“不撞南墙不回头”，即沿着一条路径一直向下走，直到无法继续。第二是“回溯”，当无法继续前进时，退回一步（上一个节点），寻找其他未探索的路径。第三是递归与栈，这种“前进”与“回溯”的过程天然适合用递归或显式栈（Stack）来实现。

算法步骤的文字描述如下。首先，从起始节点开始，将其标记为“已访问”。接着，检查当前节点是否为目标节点。如果是，则搜索成功。如果否，则遍历当前节点的所有“未访问”的相邻节点。然后，对每一个相邻节点，递归地执行上述步骤。最后，如果所有相邻节点都已访问且未找到目标，则回溯到上一个节点。这个过程体现了 DFS 的深度优先特性，确保算法优先探索最深的路径。

2　图解 DFS：遍历一棵树

为了直观理解 DFS，我们以遍历一棵简单的二叉树为例。假设二叉树有根节点 A，左子节点 B，右子节点 C，B 有左子节点 D 和右子节点 E，C 有左子节点 F。DFS 的遍历从根节点 A 开始，首先访问 A，然后深入左子树，访问 B，接着继续深入左子树，访问 D。由于 D 没有子节点，算法回溯到 B，然后访问 B 的右子节点 E。回溯到 A 后，再深入右子树，访问 C，然后访问 C 的左子节点 F。整个过程访问顺序为 A、B、D、E、C、F，这体现了前序遍历模式。通过这种文字描述，读者可以感受到 DFS“一路到底”的访问模式，无需依赖图像。

3　DFS 的两种实现方式

3.1　递归实现（最直观）

递归实现利用系统的函数调用栈来隐式地实现回溯，代码简洁直观。以下是一个 Python 示例代码框架。

```python
def dfs_recursive(node, target, visited=None):
    if visited is None:
        visited = set() # 用于记录已访问节点，避免重复访问（图结构尤其重要）

    if node is None or node in visited:
        return None

    # 1. 访问当前节点
    visited.add(node)
    print(f"Visiting node: {node.val}") # 执行访问操作
    if node.val == target:
        return node # 找到目标

    # 2. 递归地访问所有相邻节点（左子树、右子树）
    left_result = dfs_recursive(node.left, target, visited)
    if left_result is not None:
        return left_result

    right_result = dfs_recursive(node.right, target, visited)
    return right_result
```

代码解读：这个递归函数首先检查节点是否为空或已访问，如果是则返回。然后访问当前节点，标记为已访问，并检查是否为目标。如果不是，则递归调用左子树和右子树。递归过程自然地实现了“深入”：当调用左子树时，函数会一直向左深入，直到叶子节点；然后通过返回机制实现“回溯”，回到上一个节点继续探索右子树。使用 visited 集合避免了重复访问，这在图结构中至关重要。

3.2 迭代实现（使用显式栈）

迭代实现使用一个栈数据结构来模拟递归过程，手动管理待访问的节点。以下是一个 Python 示例代码框架。

```python
def dfs_iterative(start_node, target):
    if not start_node:
        return None

    stack = [start_node] # 初始化栈，放入起始节点
    visited = set() # 记录已访问节点

    while stack: # 栈不为空则继续
        node = stack.pop() # 弹出栈顶元素（关键步骤！）

        if node in visited:
```

```
        continue

    # 访问当前节点
    visited.add(node)
    print(f"Visiting node: {node.val}")
    if node.val == target:
        return node # 找到目标

    # !!! 注意：为了保持与递归相同的左子树优先顺序，需要反向压入相邻节点
    # 例如，先压入右子节点，再压入左子节点，这样左子节点会先被弹出
    if node.right:
        stack.append(node.right)
    if node.left:
        stack.append(node.left)

return None # 未找到目标
```

代码解读：这个迭代版本使用一个栈来存储待访问节点。循环中，每次弹出栈顶节点（后进先出原则），访问它并检查是否为目标。然后，将相邻节点压入栈中，但为了模拟递归的左子树优先顺序，需要先压入右子节点，再压入左子节点。这样，左子节点会在栈顶先被弹出，确保深度优先。栈的 LIFO 特性保证了算法总是优先探索最近添加的节点，从而实现深度优先搜索。这种方法避免了递归可能导致的栈溢出问题，适用于深度较大的场景。

4 从树到图：DFS 的应用扩展与关键点

当将 DFS 应用到图结构时，图的特殊性在于可能存在环。如果不记录已访问节点，算法会在环中无限循环，导致性能问题或死循环。关键挑战在于如何避免重复访问。解决方案是使用一个 visited 集合（或数组）来记录所有已访问过的节点，这在之前的代码中已经体现。代码调整方面，只需将遍历“左右子节点”的逻辑替换为遍历图的“邻接表”或“邻接矩阵”，核心框架不变。例如，在图结构中，相邻节点可能通过列表或字典表示，DFS 会递归或迭代地访问每个邻接节点，同时维护 visited 集合以防止循环。

5 DFS 的经典应用场景

深度优先搜索在多个领域有经典应用。在路径查找中，它可以用于解决迷宫问题或判断图中两点是否连通。拓扑排序是另一个重要应用，用于有向无环图（DAG），解决任务调度、编译顺序等问题。DFS 还能检测图中是否有环：在遍历过程中，如果遇到一个“已访问”且不是父节点的节点，则说明存在环。此外，在无向图中，一次 DFS 可以遍历一个连通分量，用于计算图的连通性。更重要的是，DFS 是回溯算法的基石，解决 N 皇后、数独、组合求和等问题时，核心就是 DFS 加上剪枝策略，通过深度探索和回溯来寻找所有可能解。

6 DFS 的优缺点分析

深度优先搜索有其独特的优点和缺点。优点方面，实现简单，代码简洁，尤其是递归形式，易于理解和编写。对于深度很大但目标在深处的场景，DFS 可能比广度优先搜索（BFS）更快找到解，因为它优先探索深层路径。空间复杂度相对较低，主要取决于递归深度或栈的深度，在最坏情况下为 $O(h)$（树高）或 $O(V)$（图的节点数），其中 h 表示树的高度，V 表示节点数。缺点方面，DFS 不一定找到最短路径，这是 BFS 的优势，因为 DFS 可能先探索一条长路径而忽略更短的选项。如果搜索树深度无限，递归实现可能导致栈溢出，尤其是在编程语言中递归深度有限的情况下。在状态空间巨大且无解的情况下，DFS 可能会陷入“深度陷阱”，性能不佳，因为它会一直深入直到回溯。

回顾 DFS 的核心，我们再次强调其“深度优先”和“回溯”的思想，这使它成为遍历树和图的高效工具。实现上，递归与迭代两种方式各有千秋，但都基于栈的原理。鼓励读者动手实践，例如在 LeetCode 等平台上尝试“二叉树的最大深度”或“路径总和”等题目，以加深对 DFS 的理解和应用。通过不断练习，读者可以掌握如何在不同场景下灵活运用 DFS。

7 附录与思考题

思考题一：如何修改 DFS 算法来记录并输出从起点到目标点的完整路径？这可以通过在递归或迭代过程中维护一个路径栈或列表来实现，每次访问节点时记录路径，回溯时移除节点。思考题二：在迭代实现的 DFS 中，如果希望访问顺序与递归实现完全一致，压栈顺序应该是怎样的？答案是先压入右子节点，再压入左子节点，以确保左子节点先被访问。相关阅读推荐广度优先搜索（BFS）算法，读者可以比较 DFS 与 BFS 的异同，进一步理解搜索策略的选择。