深入理解并实现基本的深度优先搜索（DFS）算法

杨岢瑞

Oct 10, 2025

想象你身处一个巨大的迷宫，没有地图，你该如何系统地探索每一条路径，确保不重不漏？一种最直观的策略便是“一条路走到黑”。遇到岔路时，选择一条路走到底，直到死胡同，然后返回上一个岔路口选择另一条路。这种“不撞南墙不回头”的策略，正是我们今天要深入探讨的深度优先搜索（Depth-First Search，DFS）算法的精髓。本文将带你从本质理解 DFS 的思想，掌握其递归与非递归两种实现方式，并通过经典问题学会如何应用它，从而在算法世界中游刃有余。

1　DFS 核心思想解析

深度优先搜索的核心思想可以用一个形象化的比喻来概括：“一条路走到黑”加上“后悔机制”（即回溯）。具体来说，DFS 在探索图或树结构时，会优先沿着一条路径深入到底，直到无法继续前进，然后回溯到上一个节点，尝试其他未探索的分支。这种策略依赖于栈（Stack）数据结构，无论是隐式的系统调用栈（在递归中）还是显式维护的栈（在迭代中），都利用了栈的“后进先出”特性来实现回溯过程。

DFS 的核心操作可以分解为几个关键步骤。首先，访问当前节点，例如打印节点信息或判断是否达到目标。其次，标记当前节点为已访问，这是防止程序在存在环的图中陷入无限循环的关键步骤。然后，对于当前节点的每一个未被访问的邻居节点，重复上述访问和标记过程，实现深度探索。最后，当所有邻居都被探索完毕时，回溯到上一个节点，继续探索其他分支。整个过程就像在迷宫中系统地尝试每一条路径，确保不会遗漏任何可能性。

2　DFS 的两种实现方式

2.1　递归实现

递归实现是 DFS 最直观和常用的方式，它利用函数调用栈来隐式地管理回溯过程。以下是一个基于图的邻接表表示的递归 DFS 实现框架，使用 Python 风格的伪代码。

```
def dfs_recursive(node, visited):
    # 标记当前节点为已访问
    visited[node] = True
    # 处理当前节点，例如打印节点信息
    print(f"Visiting␣node␣{node}")
    # 遍历当前节点的所有邻居
    for neighbor in graph[node]:
        # 如果邻居未被访问，递归调用 DFS
```

```
        if not visited[neighbor]:
            dfs_recursive(neighbor, visited)
```

在这段代码中，visited 数组用于记录每个节点的访问状态，防止重复访问。递归调用 dfs_recursive 实现了深度优先的探索：每次函数调用都会处理当前节点，然后对其未访问的邻居递归调用自身，从而沿着一条路径深入到底。递归的终止条件隐含在循环中：当所有邻居都已被访问时，函数会自然返回，实现回溯。这种实现方式代码简洁，但需要注意递归深度过大时可能导致栈溢出。

2.2 迭代实现

迭代实现通过显式使用栈来模拟递归过程，避免了递归可能带来的栈溢出问题，并提供了更好的控制力。以下是迭代 DFS 的实现框架。

```
def dfs_iterative(start_node):
    # 初始化栈和已访问集合
    stack = []
    visited = set()
    # 将起始节点压入栈中
    stack.append(start_node)
    while stack: # 当栈不为空时循环
        # 弹出栈顶元素
        node = stack.pop()
        if node not in visited:
            # 标记节点为已访问
            visited.add(node)
            # 处理当前节点
            print(f"Visiting␣node␣{node}")
            # 将未访问的邻居逆序压入栈中
            for neighbor in reversed(graph[node]):
                if neighbor not in visited:
                    stack.append(neighbor)
```

在迭代实现中，栈用于存储待访问的节点，visited 集合记录已访问节点。关键点在于，我们在弹出节点后才检查其访问状态，这是因为同一节点可能被多次压入栈中（例如通过不同路径）。逆序压入邻居节点是为了保持与递归实现一致的遍历顺序，否则由于栈的后进先出特性，遍历顺序可能会颠倒。迭代实现虽然代码稍复杂，但能有效控制内存使用，适用于深度极大的场景。

递归实现和迭代实现在可读性、性能和适用场景上各有优劣。递归实现代码简洁，更符合 DFS 的思维模型，但在深度过大时可能引发栈溢出。迭代实现通过显式栈避免了这一问题，且便于控制执行流程，但代码可读性稍低。在实际应用中，如果图深度可控，递归实现是首选；对于深度极大或需要精细控制栈的场景，迭代实现更为合适。无论哪种方式，核心都是利用栈的后进先出特性来实现深度优先和回溯。

3 DFS 的实战应用

3.1 应用一：二叉树的前序遍历

二叉树的前序遍历是 DFS 在树结构上的直接体现，访问顺序为“根节点 -> 左子树 -> 右子树”。以下分别用递归和迭代实现前序遍历。

递归实现代码如下：

```
def preorder_recursive(root):
    if root is None:
        return
    # 访问根节点
    print(root.val)
    # 递归遍历左子树
    preorder_recursive(root.left)
    # 递归遍历右子树
    preorder_recursive(root.right)
```

在这段代码中，我们首先处理当前节点（根节点），然后递归处理左子树和右子树，这正符合 DFS 的深度优先思想。递归调用栈确保了回溯的正确性。

迭代实现代码如下：

```
def preorder_iterative(root):
    if root is None:
        return
    stack = [root]
    while stack:
        node = stack.pop()
        # 访问当前节点
        print(node.val)
        # 先将右子节点压栈，再将左子节点压栈，以确保左子树先被处理
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)
```

迭代实现使用栈来模拟递归过程。由于栈的后进先出特性，我们先将右子节点压栈，再将左子节点压栈，这样左子节点会先被弹出和处理，实现了前序遍历的顺序。这种方法展示了 DFS 如何通过栈管理遍历路径。

3.2 应用二：查找路径

在图中查找从节点 A 到节点 B 的路径是 DFS 的经典应用。我们可以通过 DFS 探索所有可能路径，并记录访问顺序。以下是一个查找路径的递归实现示例。

```
def find_path_dfs(start, target, visited, path):
    # 标记当前节点为已访问
    visited[start] = True
    # 将当前节点加入路径
    path.append(start)
    if start == target:
        # 找到目标，返回路径
        return path.copy()
    # 遍历邻居节点
    for neighbor in graph[start]:
        if not visited[neighbor]:
            # 递归搜索
            result = find_path_dfs(neighbor, target, visited, path)
            if result:
                return result
    # 回溯：从路径中移除当前节点
    path.pop()
    return None
```

在这个实现中，我们使用 visited 数组避免重复访问，path 列表记录当前路径。当找到目标节点时，返回路径副本；否则，在回溯时从路径中移除当前节点。这体现了 DFS 的回溯机制，但注意 DFS 找到的路径不一定是最短路径，因为它优先深度探索。

4 DFS 的复杂度分析与常见陷阱

深度优先搜索的时间复杂度通常为 $O(V+E)$，其中 V 是顶点数，E 是边数。这是因为每个节点和边最多被访问一次。空间复杂度主要取决于 visited 数据结构和栈的使用，最坏情况下为 $O(V)$，例如当图呈链状结构时，栈可能需要存储所有节点。

在实际实现中，常见的陷阱包括忘记标记节点为已访问，这会导致在存在环的图中陷入无限循环；在迭代实现中，在错误的位置标记已访问可能导致节点被重复处理；此外，混淆 DFS 与广度优先搜索（BFS）的适用场景也是一个常见错误，例如 DFS 不适合求解非加权图的最短路径问题。为了避免这些陷阱，务必确保在访问节点时立即标记，并根据问题特性选择合适的算法。

深度优先搜索的核心在于其“深度优先”和“回溯”思想，以及栈数据结构的巧妙运用。DFS 的优点包括实现简单、空间效率较高，并且是许多高级算法（如回溯和 Tarjan 算法）的基础；缺点则是可能无法找到最优解（如最短路径），且递归实现有深度限制。

进一步思考，DFS 与回溯算法密切相关，回溯本质上是 DFS 加上剪枝优化。与 BFS 相比，DFS 更适用于探索所有可能解或路径存在的场景，而 BFS 则擅长寻找最短路径。鼓励读者动手实现代码，并尝试应用 DFS 解决更复杂的问题，如数独或 N 皇后问题，从而深化对算法的理解。通过不断实践，你将能灵活运用 DFS 应对各种挑战。