# 理解并实现基本的拓扑排序算法：从理论到代码实践

杨其臻

Apr 02, 2025

## 1 引言

在计算机科学中，拓扑排序是一种解决依赖关系问题的关键算法。想象这样一个场景：大学选课时，某些课程需要先修课程。例如，学习「数据结构」前必须先修「程序设计基础」，这种依赖关系构成一个有向无环图（DAG）。拓扑排序的作用正是为这类依赖关系找到一种合理的执行顺序。本文将深入解析拓扑排序的核心原理，并通过 Python 代码实现两种经典算法。

## 2 拓扑排序基础概念

拓扑排序的定义是：对 DAG 的顶点进行线性排序，使得对于任意有向边 $u \to v$，顶点 $u$ 在排序中都出现在顶点 $v$ 之前。例如，若图中存在边 $A \to B$ 和 $B \to C$，则可能的排序之一是 $[A, B, C]$。
拓扑排序有两个关键特性：

- 无环性：若图中存在环（例如 $A \to B \to C \to A$），则无法进行拓扑排序。可通过深度优先搜索（DFS）检测环的存在。
- 不唯一性：同一 DAG 可能有多种有效排序。例如，若图中有两个无依赖关系的节点 $A$ 和 $B$，则 $[A, B]$ 和 $[B, A]$ 均为合法结果。

## 3 拓扑排序算法原理

### 3.1 Kahn 算法（基于入度）

Kahn 算法的核心思想是不断移除入度为 0 的节点，直到所有节点被处理。具体步骤如下：

- 初始化所有节点的入度表。
- 将入度为 0 的节点加入队列。
- 依次处理队列中的节点，将其邻接节点的入度减 1。若邻接节点入度变为 0，则加入队列。
- 若最终处理的节点数等于总节点数，则排序成功；否则说明图中存在环。

该算法依赖队列数据结构，时间复杂度为 $O(V + E)$，其中 $V$ 是节点数，$E$ 是边数。

### 3.2 DFS 后序遍历法

DFS 算法通过深度优先遍历图，并按递归完成时间的逆序得到拓扑排序。具体步骤如下：

- 从任意未访问节点开始递归 DFS。
- 将当前节点标记为已访问。
- 递归处理所有邻接节点。
- 递归结束后将当前节点压入栈中。
- 最终栈顶到栈底的顺序即为拓扑排序结果。

DFS 算法同样具有 $O(V+E)$ 的时间复杂度，但需要额外的栈空间存储结果。

### 3.3 算法对比

1. **Kahn** 算法：显式利用入度信息，适合动态调整入度的场景（如动态图）。
2. **DFS** 算法：代码简洁，但难以处理动态变化的图。

## 4 代码实现（以 Python 为例）

### 4.1 图的表示

使用邻接表表示图，例如节点 0 的邻接节点为 [1, 2]：

```
graph = {
   0: [1, 2],
   1: [3],
   2: [3],
   3: []
}
```

### 4.2 Kahn 算法实现

```
from collections import deque

def topological_sort_kahn(graph, n):
   # 初始化入度表
   in_degree = {i: 0 for i in range(n)}
   for u in graph:
      for v in graph[u]:
         in_degree[v] += 1
```

```
    # 将入度为 0 的节点加入队列
    queue = deque([u for u in in_degree if in_degree[u] == 0])
    result = []

    while queue:
        u = queue.popleft()
        result.append(u)
        # 更新邻接节点的入度
        for v in graph.get(u, []):
            in_degree[v] -= 1
            if in_degree[v] == 0:
                queue.append(v)

    # 检查是否存在环
    if len(result) != n:
        return [] # 存在环
    return result
```

代码解读：

1. in_degree 字典记录每个节点的入度。
2. 队列 queue 维护当前入度为 0 的节点。
3. 每次从队列取出节点后，将其邻接节点的入度减 1。若邻接节点入度变为 0，则加入队列。
4. 最终若结果列表长度不等于节点总数，则说明存在环。

## 4.3 DFS 算法实现

```
def topological_sort_dfs(graph):
    visited = set()
    stack = []

    def dfs(u):
        if u in visited:
            return
        visited.add(u)
        # 递归访问所有邻接节点
        for v in graph.get(u, []):
            dfs(v)
        # 递归结束后压入栈
        stack.append(u)
```

```
    for u in graph:
        if u not in visited:
            dfs(u)
    # 逆序输出栈
    return stack[::-1]
```

代码解读：

1. visited 集合记录已访问的节点。
2. dfs 函数递归访问邻接节点，完成后将当前节点压入栈。
3. 最终栈的逆序即为拓扑排序结果（后进先出的栈结构需要反转）。

## 5 实例演示与测试

假设有以下 DAG：

```
5 → 0 ← 4
↓  ↓  ↓
2 → 3 → 1
```

手动推导：可能的拓扑排序为 [5, 4, 2, 0, 3, 1]。
代码测试：

1. 输入图的邻接表表示：

```
graph = {
    5: [0, 2],
    4: [0, 1],
    2: [3],
    0: [3],
    3: [1],
    1: []
}
n = 6
```

1. 运行 topological_sort_kahn(graph, 6) 应返回长度为 6 的合法排序。
2. 若图中存在环（例如添加边 1 → 5），两种算法均返回空列表。

## 6 复杂度与优化

两种算法的时间复杂度均为 $O(V+E)$，空间复杂度为 $O(V)$。

优化技巧：若需要字典序最小的排序，可将 Kahn 算法中的队列替换为优先队列（最小堆）。

## 7 实际应用场景

- 编译器构建：确定源代码文件的编译顺序。
- 课程安排：解决 LeetCode 210 题「课程表 II」的依赖问题。
- 任务调度：管理具有前后依赖关系的任务执行顺序。

## 8 总结与扩展

拓扑排序是处理依赖关系的核心算法。通过 Kahn 算法和 DFS 算法的对比，可根据实际需求选择实现方式。进一步学习可探索：

1. 强连通分量：使用 Tarjan 算法识别图中的环。
2. 动态拓扑排序：在频繁增删边的场景下维护排序结果。
3. 练习题：LeetCode 207（判断能否完成课程）、310（最小高度树）等。

## 9 参考资源

1.《算法导论》第 22.4 章「拓扑排序」。
2. VisuAlgo 的可视化工具：https://visualgo.net/zh/graphds。