深入理解并实现基本的拓扑排序算法

李睿远

Oct 19, 2025

从依赖关系到线性序列，掌握有向无环图（DAG）的排序奥秘。拓扑排序是图论中一个基础而重要的算法，广泛应用于任务调度、依赖管理等场景。本文将系统性地介绍拓扑排序的理论基础、两种主流实现算法（Kahn 算法和基于 DFS 的算法），并通过 Python 代码示例进行详细解析，帮助读者深入理解其核心思想与应用。

在日常生活中，我们经常遇到各种依赖关系场景。例如，在大学课程选修中，学生必须完成《高等数学》后才能学习《数据结构》，而《数据结构》又是《算法分析》的先修课。这种依赖关系构成了一个典型的有向图结构。课程可以被视为「顶点」，先修关系则对应「有向边」，从先修课指向后续课。整个课程体系形成了一张有向图，而一个合理的学习计划就是这张图的一个拓扑序列。拓扑排序的目标正是将这种依赖关系转化为线性序列，确保所有前置条件得到满足。本文将深入探讨拓扑排序的原理与实现，帮助读者掌握这一关键算法。

1　前置知识：图论基础

在深入拓扑排序之前，我们需要了解一些图论基本概念。有向图是由顶点和带有方向的边组成的结构，其中每条边从一个顶点指向另一个顶点。顶点的「入度」是指向该顶点的边的数量，而「出度」是从该顶点指出的边的数量。以课程依赖为例，如果一门课程有多门先修课，其入度就较高；如果它被多门后续课程依赖，则出度较高。拓扑排序的核心前提是图必须为有向无环图（DAG），即图中不能存在任何环。如果图中存在环，例如 A 依赖 B，B 依赖 C，C 又依赖 A，依赖关系会形成死循环，无法得出满足所有依赖的线性序列。因此，拓扑排序仅适用于 DAG，环检测成为算法中的重要环节。

2　拓扑排序的核心思想与算法流程

拓扑排序的目标是生成一个顶点序列，使得对于图中任何一条有向边 $u \to v$，u 在序列中都出现在 v 之前。这确保了所有依赖关系得到遵守。接下来，我们将介绍两种主流算法：Kahn 算法和基于 DFS 的算法。

2.1　Kahn 算法（基于 BFS，贪心思想）

Kahn 算法采用广度优先搜索（BFS）的策略，其直觉是从没有任何依赖（即入度为 0）的顶点开始处理。这些顶点是天然的起点，算法逐步移除它们，并更新依赖关系。详细步骤如下：首先，初始化阶段计算每个顶点的入度，并将所有入度为 0 的顶点加入队列，同时准备一个结果列表。接着，循环处理队列中的顶点：取出一个顶点 u，加入结果列表，然后遍历 u 的所有邻接顶点 v，将 v 的入度减 1（相当于移除边 $u \to v$）。如果 v 的入度变为 0，则将 v 加入队列。最后，检查结果列表中的顶点数是否等于总顶点数；如果相等，排序成功；否则，说明图中存在环，无法完成拓扑排序。这种方法直观易理解，适合动态任务调度场景。

2.2 基于 DFS 的算法

基于深度优先搜索（DFS）的算法利用递归路径反映依赖关系。直觉是当一个顶点的所有后代都被访问完毕后，再将其加入序列，确保它出现在依赖项之后。详细步骤包括：初始化一个栈用于存储结果，并创建一个标记数组跟踪顶点状态（未访问、访问中、已访问）。然后，对每个未访问顶点执行 DFS：标记为访问中，递归访问邻接顶点。如果遇到访问中的顶点，说明发现环，终止过程；否则，当所有邻接顶点处理完毕，标记为已访问并压入栈中。最后，将栈中元素弹出，得到拓扑序列。与 Kahn 算法相比，DFS 算法代码更简洁，但需注意环检测在递归中进行。

3 代码实现（以 Python 为例）

在代码实现中，我们使用邻接表表示图结构。邻接表以字典形式存储，键为顶点，值为该顶点指向的顶点列表。例如，图 `graph = {'A': ['B', 'C'], 'B': ['D'], 'C': ['D'], 'D': []}` 表示顶点 A 指向 B 和 C，B 和 C 指向 D，D 无出边。这种表示法高效且易于遍历。

3.1 Kahn 算法实现

以下是 Kahn 算法的 Python 实现代码。我们使用 `collections.deque` 作为队列，以提高效率。

```
from collections import deque

def topological_sort_kahn(graph):
    # 初始化入度表：为每个顶点设置初始入度为 0
    in_degree = {node: 0 for node in graph}
    # 遍历图，计算每个顶点的实际入度
    for node in graph:
        for neighbor in graph[node]:
            in_degree[neighbor] += 1

    # 创建队列，将所有入度为 0 的顶点加入
    queue = deque([node for node in in_degree if in_degree[node] == 0])
    topo_order = [] # 存储拓扑序列的结果列表

    # 循环处理队列中的顶点
    while queue:
        u = queue.popleft() # 取出一个入度为 0 的顶点
        topo_order.append(u) # 加入结果序列
        # 遍历 u 的所有邻接顶点 v
        for v in graph[u]:
            in_degree[v] -= 1 # 减少 v 的入度，相当于移除边 u->v
```

```
            if in_degree[v] == 0: # 如果 v 的入度变为 0，加入队列
                queue.append(v)

    # 检查是否有环：如果结果序列长度不等于顶点数，则存在环
    if len(topo_order) == len(graph):
        return topo_order
    else:
        return [] # 返回空列表表示有环，也可抛出异常

# 调用示例
graph = {'A': ['B', 'C'], 'B': ['D'], 'C': ['D'], 'D': []}
print(topological_sort_kahn(graph)) # 输出可能为 ['A', 'B', 'C', 'D'] 或 ['A', 'C', 'B
    ↪ ', 'D']
```

在这段代码中，首先初始化入度表，通过遍历图的边来填充每个顶点的入度值。然后，使用队列处理入度为 0 的顶点，逐步更新邻接顶点的入度。循环结束后，通过比较结果序列长度与顶点总数来判断是否存在环。该实现的时间复杂度为 $O(V+E)$，其中 V 是顶点数，E 是边数，因为每个顶点和边只被处理一次。

3.2 DFS 算法实现

基于 DFS 的拓扑排序算法代码如下。它使用递归和栈来管理顶点状态。

```
def topological_sort_dfs(graph):
    visited = {} # 标记字典：0 未访问，1 访问中，2 已访问
    stack = [] # 栈用于存储拓扑序列

    # 初始化所有顶点为未访问状态
    for node in graph:
        visited[node] = 0

    def dfs(node):
        if visited[node] == 1: # 如果遇到访问中的顶点，说明有环
            raise ValueError("图中存在环，无法进行拓扑排序！")
        if visited[node] == 2: # 已访问顶点，直接返回
            return
        visited[node] = 1 # 标记为访问中
        # 递归访问所有邻接顶点
        for neighbor in graph[node]:
            dfs(neighbor)
        visited[node] = 2 # 标记为已访问
        stack.append(node) # 顶点处理完毕后压入栈
```

```
    # 对每个未访问顶点执行 DFS
    for node in graph:
        if visited[node] == 0:
            dfs(node)

    return stack[::-1] # 反转栈，得到拓扑序列

# 调用示例
graph = {'A': ['B', 'C'], 'B': ['D'], 'C': ['D'], 'D': []}
print(topological_sort_dfs(graph)) # 输出可能为 ['A', 'B', 'C', 'D'] 或 ['A', 'C', 'B',
    ↪ 'D']
```

这段代码通过 DFS 递归遍历图，使用 `visited` 字典跟踪顶点状态。在递归过程中，如果发现顶点处于访问中状态，则抛出环异常；否则，处理完所有邻接顶点后压入栈。最后，反转栈得到拓扑序列。该实现同样具有 $O(V+E)$ 的时间复杂度，但空间复杂度取决于递归深度。

4 算法分析与比较

拓扑排序的两种算法在时间复杂度和空间复杂度上相似，时间复杂度均为 $O(V+E)$，因为每个顶点和边只被访问一次。空间复杂度方面，Kahn 算法需要存储入度表和队列，通常为 $O(V)$；DFS 算法则需要递归栈空间，最坏情况下也为 $O(V)$。在选择算法时，Kahn 算法更直观，易于实现环检测，适合需要动态知道当前可执行任务的场景，如任务调度器。而 DFS 算法代码更简洁，如果需要求所有可能的拓扑序列，DFS 更容易通过回溯实现。实际应用中，应根据具体需求选择算法，例如在依赖管理系统中，Kahn 算法可能更实用。

5 实际应用场景

拓扑排序在多个领域有广泛应用。在编译过程中，它用于管理模块依赖关系，确保编译顺序正确；在任务调度与项目管理中，它帮助确定任务执行的先后顺序；软件包管理器如 apt 或 npm 利用拓扑排序解决库依赖的安装顺序；数据流系统中，它用于计算有依赖关系的单元格，例如 Excel 中的公式求值；此外，课程安排、工作流设计等场景也依赖拓扑排序来优化流程。这些应用凸显了拓扑排序在解决依赖问题中的重要性。

拓扑排序是针对有向无环图的关键算法，其核心在于将依赖关系转化为线性序列。本文介绍了 Kahn 算法和基于 DFS 的算法，两者均能高效实现拓扑排序，但各有适用场景。环检测是算法中不可忽视的环节，确保图的合理性。通过代码实现和详细解读，读者可以动手实践，进一步解决实际问题，如 LeetCode 上的课程表题目。掌握拓扑排序不仅提升算法能力，还为处理复杂依赖关系提供实用工具。