Topological Sorting拓扑排序

入度与出度

在介绍算法之前，我们先介绍图论中的一个基本概念，入度和出度，英文为 in-degree & out-degree。

在有向图中，如果存在一条有向边 A-->B，那么我们认为这条边为 A 增加了一个出度，为 B 增加了一个入度。

算法流程

拓扑排序的算法是典型的宽度优先搜索算法，其大致流程如下：

1. 统计所有点的入度，并初始化拓扑序列为空。
2. 将所有入度为 0 的点，也就是那些没有任何依赖的点，放到宽度优先搜索的队列中
3. 将队列中的点一个一个的释放出来，放到拓扑序列中，每次释放出某个点 A 的时候，就访问 A 的相邻点（所有A指向的点），并把这些点的入度减去 1。
4. 如果发现某个点的入度被减去 1 之后变成了 0，则放入队列中。
5. 直到队列为空时，算法结束，

深度优先搜索的拓扑排序

深度优先搜索也可以做拓扑排序，不过因为不容易理解，也并不推荐作为拓扑排序的主流算法。

拓扑排序程序模版：

```
public class Solution {
    /**
     * @param graph: A list of Directed graph node
     * @return: Any topological order for the given graph.
     */
    public ArrayList<DirectedGraphNode> topSort(ArrayList<DirectedGraphNode> graph) {
        // map 用来存储所有节点的入度，这里主要统计各个点的入度
        HashMap<DirectedGraphNode, Integer> map = new HashMap();
        for (DirectedGraphNode node : graph) {
            for (DirectedGraphNode neighbor : node.neighbors) {
                if (map.containsKey(neighbor)) {
                    map.put(neighbor, map.get(neighbor) + 1);
                } else {
                    map.put(neighbor, 1);
                }
            }
        }

        // 初始化拓扑序列为空
        ArrayList<DirectedGraphNode> result = new ArrayList<DirectedGraphNode>();

        // 把所有入度为0的点，放到BFS专用的队列中
        Queue<DirectedGraphNode> q = new LinkedList<DirectedGraphNode>();
        for (DirectedGraphNode node : graph) {
            if (!map.containsKey(node)) {
                q.offer(node);
                result.add(node);
            }
        }

        // 每次从队列中拿出一个点放到拓扑序列里，并将该点指向的所有点的入度减1
        while (!q.isEmpty()) {
            DirectedGraphNode node = q.poll();
            for (DirectedGraphNode n : node.neighbors) {
                map.put(n, map.get(n) - 1);
                // 减去1之后入度变为0的点，也放入队列
                if (map.get(n) == 0) {
                    result.add(n);
                    q.offer(n);
                }
            }
        }

        return result;
    }
}
```

宽度优先搜索

- 需要分层遍历
- 不需要分层遍历

不需要分层遍历

```
// T 指代任何你希望存储的类型
Queue<T> queue = new LinkedList<>();
Set<T> set = new HashSet<>();

set.add(start);
queue.offer(start);
while (!queue.isEmpty()) {
    T head = queue.poll();
    for (T neighbor : head.neighbors) {
        if (!set.contains(neighbor)) {
            set.add(neighbor);
            queue.offer(neighbor);
        }
    }
}
```

需要分层

```
Queue<T> queue = new LinkedList<>();
Set<T> set = new HashSet<>();

set.add(start);
queue.offer(start);
while (!queue.isEmpty()) {
    int size = queue.size();
    for (int i = 0; i < size; i++) {
        T head = queue.poll();
        for (T neighbor : head.neighbors) {
            if (!set.contains(neighbor)) {
                set.add(neighbor);
                queue.offer(neighbor);
            }
        }
    }
}
```

Practise:

- 69. Binary Tree Level Order Traversal

```
public class Solution {
    /**
     * @param root: A Tree
     * @return: Level order a list of lists of integer
     */
    public List<List<Integer>> levelOrder(TreeNode root) {
        // write your code here
        List<List<Integer>> ans = new ArrayList<>();
        if(root == null){
            return ans;
        }
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        while(!queue.isEmpty()){
            List<Integer> level = new ArrayList<>();//key point !!!
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                TreeNode head = queue.poll();
                if(head.left != null){
                    queue.add(head.left);
                }
                if(head.right != null){
                    queue.add(head.right);
                }
                level.add(head.val);
            }
            ans.add(level);
        }
        return ans;
    }

}
```

-