图相关算法

图的遍历

图的拷贝-Leetcode133 克隆图

```
/*
// Definition for a Node.
class Node {
    public int val;
    public List<Node> neighbors;
    public Node() {
        val = 0;
        neighbors = new ArrayList<Node>();
    }
    public Node(int _val) {
        val = _val;
        neighbors = new ArrayList<Node>();
    }
    public Node(int _val, ArrayList<Node> _neighbors) {
        val = _val;
        neighbors = _neighbors;
    }
}
*/

class Solution {
    public Node cloneGraph(Node node) {
        if(node==null){
            return node;
        }
        // 用hashmap来进行拷贝
        HashMap<Node,Node> visited = new HashMap<>();
        // 层序遍历
        Queue<Node> queue = new LinkedList<>();
        queue.offer(node);
        // 克隆第一个
        visited.put(node,new Node(node.val,new ArrayList<>()));

        // 广度优先搜索
        while(!queue.isEmpty()){
            // 取出来
            Node n = queue.poll();
            // 遍历
            for(Node neighbor:n.neighbors){
                if(!visited.containsKey(neighbor)){
                    visited.put(neighbor,new Node(neighbor.val,new ArrayList<>
()));

                    queue.offer(neighbor);
                }
                // 更新
                visited.get(n).neighbors.add(visited.get(neighbor));
```

```
            }
        }
        return visited.get(node);
    }
}
```

欧拉图

给定一个n个点m条边的图，要求从指定的顶点出发，经过所有的边都恰好一次(可以理解为给定起点的【一笔画】问题)，使得路径的字典序最小。

- 通过图中所有边恰好一次且行遍所有顶点的通路称之为欧拉通路；
- 通过图中所有边恰好一次且行遍所有顶点的回路称之为欧拉回路；

Leetcode332 重新安排行程

给定一个机票的字符串二维数组 [from, to]，子数组中的两个成员分别表示飞机出发和降落的机场地点，对该行程进行重新规划排序。所有这些机票都属于一个从 JFK（肯尼迪国际机场）出发的先生，所以该行程必须从 JFK 开始。

提示：

如果存在多种有效的行程，请你按字符自然排序返回最小的行程组合。例如，行程 ["JFK", "LGA"] 与 ["JFK", "LGB"] 相比就更小，排序更靠前
所有的机场都用三个大写字母表示（机场代码）。
假定所有机票至少存在一种合理的行程。
所有的机票必须都用一次 且 只能用一次。

示例 1：

输入：[["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]
输出：["JFK", "MUC", "LHR", "SFO", "SJC"]

```
class Solution {
    Map<String,PriorityQueue<String>> map = new
HashMap<String,PriorityQueue<String>>();
    List<String> itinerary = new LinkedList<String>();
    public List<String> findItinerary(List<List<String>> tickets) {
        for(List<String> ticket:tickets){
            String src = ticket.get(0);
            String dst = ticket.get(1);
            if(!map.containsKey(src)){
                map.put(src,new PriorityQueue<String>());
            }
            map.get(src).offer(dst);
        }
        dfs("JFK");
        Collections.reverse(itinerary);
        return itinerary;
    }
    public void dfs(String curr){
        while(map.containsKey(curr)&&map.get(curr).size()>0){
            String temp = map.get(curr).poll();
            dfs(temp);
        }
```

```
            itinerary.add(curr);
        }
    }
```

图的拓扑排序

Leetcode310 最小高度树

树是一个无向图，其中任何两个顶点只通过一条路径连接。 换句话说，一个任何没有简单环路的连通图都是一棵树。

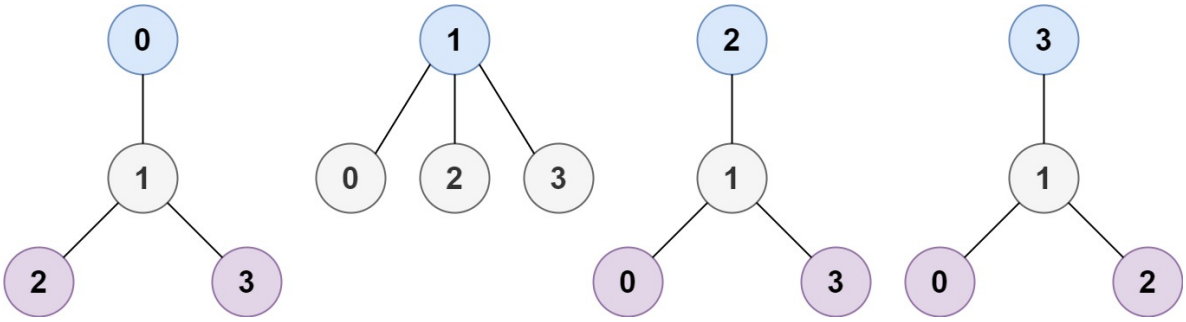
给你一棵包含 `n` 个节点的树，标记为 `0` 到 `n - 1` 。给定数字 `n` 和一个有 `n - 1` 条无向边的 `edges` 列表（每一个边都是一对标签），其中 `edges[i] = [ai, bi]` 表示树中节点 `ai` 和 `bi` 之间存在一条无向边。

可选择树中任何一个节点作为根。当选择节点 `x` 作为根节点时，设结果树的高度为 `h` 。在所有可能的树中，具有最小高度的树（即，`min(h)`）被称为 **最小高度树** 。

请你找到所有的 **最小高度树** 并按 **任意顺序** 返回它们的根节点标签列表。

树的 **高度** 是指根节点和叶子节点之间最长向下路径上边的数量。

示例 1：




```
输入：n = 4, edges = [[1,0],[1,2],[1,3]]
输出：[1]
解释：如图所示，当根是标签为 1 的节点时，树的高度是 1 ，这是唯一的最小高度树
```

```
class Solution {
    public List<Integer> findMinHeightTrees(int n, int[][] edges) {
        // 最终的结果
        List<Integer> res = new ArrayList<>();
        if(n<=2){
            for(int i=0;i<n;i++){
                res.add(i);
            }
            return res;
        }
        //BFS得到最后的度为1的
        // 构建无向图
        List<List<Integer>> graph = new ArrayList<>();
        int[] indeg = new int[n];
```

```
        // 初始化
        for(int i=0;i<n;i++){
            graph.add(new ArrayList<>());
        }
        // 初始化赋值
        for(int[] edge:edges){
            graph.get(edge[0]).add(edge[1]);
            graph.get(edge[1]).add(edge[0]);
            indeg[edge[0]]++;
            indeg[edge[1]]++;
        }
        // 层次遍历
        Queue<Integer> queue = new LinkedList<>();
        for(int i=0;i<n;i++){
            if(indeg[i]==1){
                queue.offer(i);
            }
        }
        while(!queue.isEmpty()){
            int size = queue.size();
            res = new ArrayList<>();
            for(int i=0;i<size;i++){
                int curr  = queue.poll();
                res.add(curr);
                // 对其邻居遍历
                for(Integer next:graph.get(curr)){
                    indeg[next]--;
                    if(indeg[next]==1){
                        queue.offer(next);
                    }
                }
            }
        }
        return res;


    }
}
```

Leetcode207 课程表

你这个学期必须选修 numCourses 门课程，记为 0 到 numCourses - 1 。

在选修某些课程之前需要一些先修课程。 先修课程按数组 prerequisites 给出，其中 prerequisites[i] = [ai, bi] ，表示如果要学习课程 ai 则 必须 先学习课程 bi 。

例如，先修课程对 [0, 1] 表示：想要学习课程 0 ，你需要先完成课程 1 。
请你判断是否可能完成所有课程的学习？如果可以，返回 true ；否则，返回 fa

示例 1：

输入：numCourses = 2, prerequisites = [[1,0]]
输出：true
解释：总共有 2 门课程。学习课程 1 之前，你需要完成课程 0 。这是可能的。

```
class Solution {
```

```
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        // 课程表存在依赖的先后关系AOV网
        // 构造一个有向图以及度
        // 记录入度
        int[] indeg = new int[numCourses];
        // 邻接表
        List<List<Integer>> edges = new ArrayList<>();
        for(int i=0;i<numCourses;i++){
            edges.add(new ArrayList<>());
        }
        // 开始初始化赋值
        for(int[] prerequisite:prerequisites){
            edges.get(prerequisite[1]).add(prerequisite[0]);
            indeg[prerequisite[0]]++;
        }
        // 拓扑排序
        Queue<Integer> queue = new LinkedList<>();
        // 度为0的先入对
        for(int i=0;i<numCourses;i++){
            if(indeg[i]==0){
                queue.offer(i);
            }
        }
        while(!queue.isEmpty()){
            int u = queue.poll();
            numCourses--;
            // 继续遍历
            for(int v:edges.get(u)){
                // 度减少
                indeg[v]--;
                // 如果为0
                if(indeg[v]==0){
                    // 入
                    queue.offer(v);
                }
            }
        }

        return numCourses==0;
    }
}
```

Leetcode210 课程表II

在你总共有 n 门课需要选，记为 0 到 n-1。

在选修某些课程之前需要一些先修课程。 例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示他们: [0,1]

给定课程总量以及它们的先决条件，返回你为了学完所有课程所安排的学习顺序。

可能会有多个正确的顺序，你只要返回一种就可以了。如果不可能完成所有课程，返回一个空数组。

示例 1:

输入: 2, [[1,0]]
输出: [0,1]
解释: 总共有 2 门课程。要学习课程 1，你需要先完成课程 0。因此，正确的课程顺序为 [0,1] 。

```java
class Solution {
    public int[] findOrder(int numCourses, int[][] prerequisites) {
        // 构造图
        int[] indeg = new int[numCourses];
        List<List<Integer>> edges = new ArrayList<>();
        for(int i=0;i<numCourses;i++){
            edges.add(new ArrayList());
        }
        // 开始构造一个图
        for(int[] prerequisite:prerequisites){
            edges.get(prerequisite[1]).add(prerequisite[0]);
            indeg[prerequisite[0]]++;
        }
        // 拓扑排序
        Queue<Integer> queue = new LinkedList<>();
        // 度为0先进入
        for(int i=0;i<numCourses;i++){
            if(indeg[i]==0){
                queue.offer(i);
            }
        }
        // 结果存储
        int[] res = new int[numCourses];
        int index = 0;
        while(!queue.isEmpty()){
            int u = queue.poll();
            numCourses--;
            res[index++] = u;

            for(int v:edges.get(u)){
                indeg[v]--;
                if(indeg[v]==0){
                    queue.offer(v);
                }
            }
        }
        if(numCourses==0){
            return res;
        }else{
            return new int[0];
        }
    }
}
```

图的最小生成树

图的最短路径

Floyd弗洛伊德算法

Leetcode1462 课程表IV

你总共需要上 n 门课，课程编号依次为 0 到 n-1 。

有的课会有直接的先修课程，比如如果想上课程 0 ，你必须先上课程 1 ，那么会以 [1,0] 数对的形式给出先修课程数对。

给你课程总数 n 和一个直接先修课程数对列表 prerequisite 和一个查询对列表 queries 。

对于每个查询对 queries[i] ，请判断 queries[i][0] 是否是 queries[i][1] 的先修课程。

请返回一个布尔值列表，列表中每个元素依次分别对应 queries 每个查询对的判断结果。

注意：如果课程 a 是课程 b 的先修课程且课程 b 是课程 c 的先修课程，那么课程 a 也是课程 c 的先修课程。

示例 1：


```
class Solution {
    public List<Boolean> checkIfPrerequisite(int n, int[][] prerequisites, int[]
[] queries) {
        // floyd算法来计算 该结点到其他结点的距离
        boolean[][] G = new boolean[n][n];
        // 初始化
        for(int i=0;i<prerequisites.length;i++){
            G[prerequisites[i][0]][prerequisites[i][1]] = true;
        }
        // 以及其余的
        for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                for(int k=0;k<n;k++){
                    if(G[j][i]&&G[i][k]){
```

```
                        G[j][k] = true;
                    }
                }
            }
        }
        // 找到结果了
        List<Boolean> res = new ArrayList<>();
        for(int i=0;i<queries.length;i++){
            res.add(G[queries[i][0]][queries[i][1]]);
        }
        return res;
    }
}
```