Graph & Topology Sort

Graph

- Definition
  - Graph is very similar to trees
  - Directed v.s. Undirected
- Interview Questions
  - DFS, BFS
  - DAG (Directed Acyclic Graph)
  - Topological sort

Graph Representation

- Graph node

```
GraphNode {
    int val;
    List<GraphNode>
neighbors;
}
```

- Adjacent matrices
  - |V| vertices, then |V|*|V| matrix of 0s and 1s for the graph.
- Adjacent lists
  - |V| vertices, then |V| arraylist, each containing the adjacent vertices.
  - 2|E| space for undirected, and |E| space for directed.

Topological Sort

A **topological sort** (sometimes abbreviated **toposort**) or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.

- Maintain a set of vertices with 0-incoming vertices.
  - Keep selecting 0-incoming vertex, remove all its outcoming edges, add it into the set.
- If all vertices are merged into 0-incoming vertices set, then topological sort result is found.

Course Schedule

There are a total of *n* courses you have to take, labeled from 0 to n - 1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite **pairs**, is it possible for you to finish all courses?

Example:

[1, 0], possible; [[1,0], [0,1]], impossible.

Course Schedule

How would we solve it in real life?

- Draw a directed graph, with prerequisite order.
- Select a course which has no prerequisite, and remove all its prerequisite relationships.
- Repeat the above operation until all courses are scheduled or 0-prerequisite course doesn't exist.
- If there are courses not selected, then the courses cannot be scheduled.

Course Schedule

```
public boolean canFinish(int numCourses, int[][] prerequisites) {
    ArrayList<ArrayList<Integer>> graph = new ArrayList<>();
    for (int i = 0; i < numCourses; i++) {
        graph.add(new ArrayList<Integer>());
    }
    int[] preNum = new int[numCourses];
    for (int i = 0; i < prerequisites.length; i++) {
        graph.get(prerequisites[i][1]).add(prerequisites[i][0]);
        preNum[prerequisites[i][0]]++;
    }

    for (int i = 0; i < numCourses; i++) {
        boolean availableCourse = false;
        for (int j = 0; j < numCourses; j++) {
            if (preNum[j] == 0) {
                for  (int k : graph.get(j)) {
                    preNum[k]--;
                }
                availableCourse = true;
                preNum[j] = -1;
                break;
            }
        }
        if (!availableCourse) {
            return false;
        }
    }
    return true;
}
```

Clone Graph

clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

```
class UndirectedGraphNode {
    int label;
    List<UndirectedGraphNode> neighbors;
    UndirectedGraphNode(int x) { label = x; neighbors = new
            ArrayList<UndirectedGraphNode>(); }
 };
```

Clone Graph

Use BFS/DFS to go through all the nodes

Use Hashmap to remember which nodes has been visited

Clone each node with its label and is neighbors

Clone Graph

```
public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
    if(node == null) return null;
    HashMap<UndirectedGraphNode, UndirectedGraphNode> hm = new HashMap();
    LinkedList<UndirectedGraphNode> queue = new LinkedList();
    UndirectedGraphNode head = new UndirectedGraphNode(node.label);
    hm.put(node, head);
    queue.add(node);

    while(!queue.isEmpty()) {
        UndirectedGraphNode curnode = queue.poll();
        for(UndirectedGraphNode neighbor: curnode.neighbors) {
            if(!hm.containsKey(neighbor)) {
                queue.add(neighbor);
                UndirectedGraphNode newneighbor = new UndirectedGraphNode(neighbor.label);
                hm.put(neighbor, newneighbor);
            }
            hm.get(curnode).neighbors.add(hm.get(neighbor));
        }
    }
    return head;
}
```

Alien Dictionary

There is a new alien language which uses the latin alphabet. However, the order among letters are unknown to you. You receive a list of words from the dictionary, where words are sorted lexicographically by the rules of this new language. Derive the order of letters in this language.

For example,

Given the following words in dictionary,

["wrt", "wrf", "er", "ett", "rftt"]

The correct order is: "wertf".

Alien Dictionary

What do we need to do?

Use the words to get the priority between two letters

Use topology sort to get the information of the order of these letters

Note:

You may assume all letters are in lowercase.

If the order is invalid, return an empty string.

There may be multiple valid order of letters, return any one of them is fine.

Alien Dictionary

```
public String alienOrder(String[] words) {
    Map<Character, Set<Character>> hm = new HashMap<>();
    Set<Character> set = new HashSet<>();
    Queue<Character> queue = new LinkedList<>();
    StringBuilder result = new StringBuilder();
    for (String word: words) {
        char[] wordArray = word.toCharArray();
        for (char c: wordArray) {
            set.add(c);
        }
    }
    for (Character c: set) {
        hm.put(c, new HashSet<Character>());
    }
    for (int i = 0; i < words.length-1; i ++) {
        int minLen = Math.min(words[i].length(), words[i+1].length());
        int j = 0;
        for(j = 0; j < minLen; j ++) {
            if (words[i].charAt(j) != words[i+1].charAt(j)) {
                hm.get(words[i+1].charAt(j)).add(words[i].charAt(j));
                break;
            }
        }
        if (j == minLen && words[i].length() > words[i+1].length()) return "";
    }
```

Alien Dictionary

```
for(Map.Entry<Character, Set<Character>> entry: hm.entrySet()) {
    if (entry.getValue().isEmpty()) {
        queue.add(entry.getKey());
        result.append(entry.getKey());
    }
}
while(!queue.isEmpty()) {
        Character c = queue.poll();
        for(Map.Entry<Character, Set<Character>> entry: hm.entrySet()) {
            if (entry.getValue().contains(c)) {
                entry.getValue().remove(c);
                if (entry.getValue().isEmpty()) {
                    queue.add(entry.getKey());
                    result.append(entry.getKey());
                }
            }
        }
    }
    String finalResult = result.toString();
    if (finalResult.length() == set.size()) {
        return finalResult;
    } else {
        return "";
    }
}
```

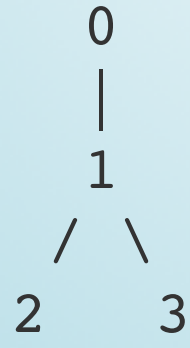
Minimum Height Trees

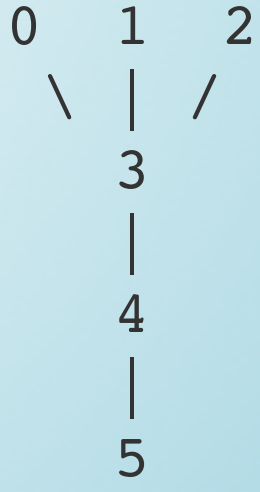
For a undirected graph with tree characteristics, we can choose any node as the root. The result graph is then a rooted tree. Among all possible rooted trees, those with minimum height are called minimum height trees (MHTs). Given such a graph, write a function to find all the MHTs and return a list of their root labels.

Examples

Given n = 4, edges = [[1, 0], [1, 2], [1, 3]], return [1].

Given n = 6, edges = [[0, 3], [1, 3], [2, 3], [4, 3], [5, 4]], return [3, 4]



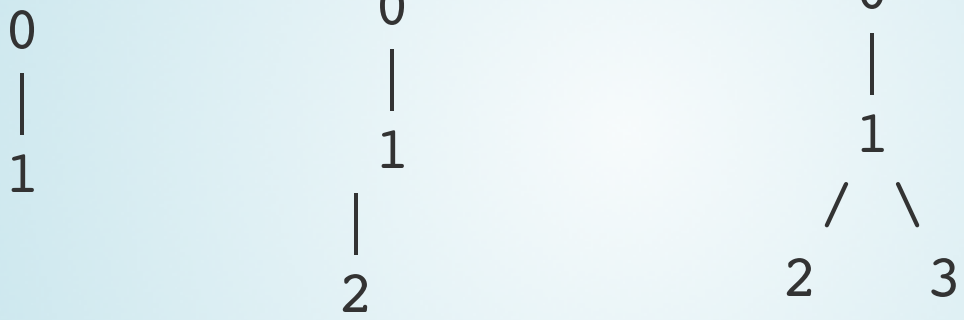


Minimum Height Trees

- Calculate the height of trees for every node
- O(|V| * |V|)

Minimum Height Trees

How many Minimum Height Trees can a tree have at most?



At most 2!

Minimum Height Trees

How many Minimum Height Trees can a tree have at most?

We can use a proof by Contradiction

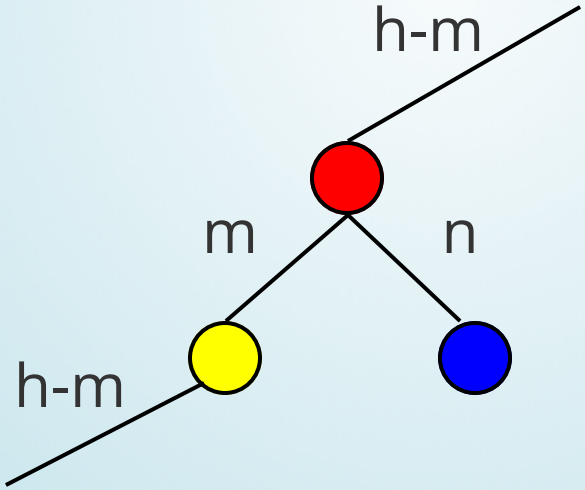
If there are more than 2 nodes that are the roots.

We just choose 3 of them. Then these 3 must become a circle somehow. It contradicts with the definition of tree

Minimum Height Trees

How many Minimum Height Trees can a tree have at most?

We can use a proof by Contradiction



Minimum Height Trees

- Remove all the leaves
- Update the graph
- Keep the process until at most two leaves are left.

```java
public List<Integer> findMinHeightTrees(int n, int[][] edges) {
    List<Integer> result = new ArrayList<>();
    if (n < 1) return result;
    ArrayList<HashSet<Integer>> graph = new ArrayList<>(n);
    for (int i = 0; i < n; i++) {
        graph.add(new HashSet<>());
    }
    for (int i = 0; i < edges.length; i++) {
        graph.get(edges[i][0]).add(edges[i][1]);
        graph.get(edges[i][1]).add(edges[i][0]);
    }

    Queue<Integer> leaves = new LinkedList<>();
    for (int i = 0; i < n; i++) {
        if (graph.get(i).size() <= 1) {
            leaves.add(i);
        }
    }
    while (n > 2) {
        n -= leaves.size();
        Queue<Integer> newLeaves = new LinkedList<>();
        while (!leaves.isEmpty()) {
            Integer leaf = leaves.poll();
            Integer neighbor = graph.get(leaf).iterator().next();
            graph.get(neighbor).remove(leaf);
            if (graph.get(neighbor).size() == 1) {
                newLeaves.add(neighbor);
            }
        }
        leaves = newLeaves;
    }
    return new ArrayList<>(leaves);
}
```

Homework

Evaluate Division

Clone Graph (DFS)

Course Schedule II