<https://www.lintcode.com/problem/605/>

**Description**

Check whether the original sequence org can be uniquely reconstructed from the sequences in seqs. The org sequence is a **permutation** of the integers from 1 to n, with 1≤*n*≤10^4.

Reconstruction means building a shortest common super sequence of the sequences in seqs (i.e., a shortest sequence so that all sequences in seqs are subsequences of it). Determine whether there is only one sequence that can be reconstructed from seqs and it is the org sequence.

**Example**

Example 1:

Input:org = [1,2,3], seqs = [[1,2],[1,3]]

Output: false

Explanation:

[1,2,3] is not the only one sequence that can be reconstructed, because [1,3,2] is also a valid sequence that can be reconstructed.

Example 2:

Input: org = [1,2,3], seqs = [[1,2]]

Output: false

Explanation:

The reconstructed sequence can only be [1,2], can't reconstruct the sequence [1,2,3].

Example 3:

Input: org = [1,2,3], seqs = [[1,2],[1,3],[2,3]]

Output: true

Explanation:

The sequences [1,2], [1,3], and [2,3] can uniquely reconstruct the original sequence [1,2,3].

Example 4:

Input:org = [4,1,5,2,6,3], seqs = [[5,2,6,3],[4,1,5,2]]

Output:true

**Attempt 1: 2022-11-19**

**Solution 1:  Topological Sort (120min)**

public class Solution {

/\*\*

\* @param org: a permutation of the integers from 1 to n

\* @param seqs: a list of sequences

\* @return: true if it can be reconstructed only one or false

\*/

public boolean sequenceReconstruction(int[] org, int[][] seqs) {

// Build graph

Map<Integer, List<Integer>> graph = buildGraph(seqs);

// Find initial indegree based on graph

Map<Integer, Integer> indegree = getIndegree(graph);

// Topological sort to find if unique path can be found from seqs as same as org

List<Integer> tmp = new ArrayList<Integer>();

Queue<Integer> q = new LinkedList<Integer>();

for(int node : indegree.keySet()) {

if(indegree.get(node) == 0) {

q.offer(node);

}

}

while(!q.isEmpty()) {

int size = q.size();

// candidate next node must be only 1 choice, because it require unique path,

// if candidate next node(s) stored on queue more than 1, which means more

// than one unique path, not satisfy requirement

if(size > 1) {

return false;

}

int node = q.poll();

tmp.add(node);

for(int neighbour : graph.get(node)) {

indegree.put(neighbour, indegree.get(neighbour) - 1);

if(indegree.get(neighbour) == 0) {

q.offer(neighbour);

}

}

}

if(tmp.size() != org.length) {

return false;

}

for(int i = 0; i < org.length; i++) {

if(org[i] != tmp.get(i)) {

return false;

}

}

return true;

}

private Map<Integer, List<Integer>> buildGraph(int[][] seqs) {

Map<Integer, List<Integer>> graph = new HashMap<Integer, List<Integer>>();

for(int[] seq : seqs) {

for(int node : seq) {

graph.put(node, new ArrayList<Integer>());

}

}

// Note: seq length may > 2, that's why use for loop

// e.g Input:org = [4,1,5,2,6,3], seqs = [[5,2,6,3],[4,1,5,2]], Output:true

for(int[] seq : seqs) {

for(int i = 0; i < seq.length - 1; i++) {

graph.get(seq[i]).add(seq[i + 1]);

}

}

return graph;

}

private Map<Integer, Integer> getIndegree(Map<Integer, List<Integer>> graph) {

Map<Integer, Integer> indegree = new HashMap<Integer, Integer>();

for(int node : graph.keySet()) {

indegree.put(node, 0);

}

for(int node : graph.keySet()) {

for(int neighbour : graph.get(node)) {

indegree.put(neighbour, indegree.get(neighbour) + 1);

}

}

return indegree;

}

}

Time Complexity: O(n+m)

The overall time complexity of this basic algorithm is O(n+m). The O(n) comes from the number of times that the while loop (and initial for loop) is executed, and the O(m) from the nested for loop.

Although there is no way to calculate how many times the inner loop will be executed on any one iteration of the outer loop, it will only be executed once for each successor of each member, which means that the total number of times that it will be executed is the total number of successors of all the members -- or the total number of relations.

Space Complexity: O(n+m)

Space complexity is also O(n+m). The O(n) component comes from the predecessor count information stored for each member, and the maximum length of the auxiliary queue. The O(m) comes from storing the successors for each member; once again, the total number of successors is the number of relations, so O(m).

**Refer to**

<https://www.lintcode.com/problem/605/solution/17390>

最后要是唯一的topological order，有三个条件：

a）Topological Sorting所输出的数组 == org

b）seq里面的node数目要等于org里面的node数目

c）顶点到末端距离 == org的长度

**Basic solution refer to**

<https://www.lintcode.com/problem/605/solution/18253>

public class Solution {

/\*\*

\* @param org a permutation of the integers from 1 to n

\* @param seqs a list of sequences

\* @return true if it can be reconstructed only one or false

\*/

public boolean sequenceReconstruction(int[] org, int[][] seqs) {

// Write your code here

// build graph

Map<Integer, ArrayList<Integer>> graph = getGraph(seqs);

/\*\*

\* we need to build graph from seqs!! cause we are using seqs to try to build org

\*

\*/

// build indegree

Map<Integer, Integer> indegree = getIndegree(graph);

// get zero indegree, offer in queue

Queue<Integer> queue = new LinkedList<>();

for (int node : graph.keySet()) {

if (indegree.get(node) == 0) {

queue.offer(node);

}

}

// we shoud have a container to compare it to org

List<Integer> sequence = new ArrayList<>();

// bfs

while (!queue.isEmpty()) {

int size = queue.size();

if (size > 1) {

return false;

}

int node = queue.poll();

sequence.add(node);

for (int neib : graph.get(node)) {

indegree.put(neib, indegree.get(neib) - 1);

if (indegree.get(neib) == 0) {

queue.offer(neib);

}

}

}

if (sequence.size() != org.length) { // first compare length, to prevent egde cases.

return false;

}

for (int i = 0; i < org.length; i++) {

if (sequence.get(i) != org[i]) {

return false;

}

}

return true;

}

private Map<Integer, ArrayList<Integer>> getGraph(int[][] seqs) {

Map<Integer, ArrayList<Integer>> graph = new HashMap<Integer, ArrayList<Integer>>();

for (int[] path : seqs) {

for (int node : path) {

graph.put(node, new ArrayList<Integer>());

}

}

for (int[] path : seqs) {

for (int i = 0; i < path.length - 1; i++) {

graph.get(path[i]).add(path[i+1]);

}

}

return graph;

}

private Map<Integer, Integer> getIndegree(Map<Integer, ArrayList<Integer>> graph) {

Map<Integer, Integer> indegree = new HashMap<Integer, Integer>();

for (int node : graph.keySet()) {

indegree.put(node, 0);

}

for (int node : graph.keySet()) {

for (int neib : graph.get(node)) {

indegree.put(neib, indegree.get(neib) + 1);

}

}

return indegree;

}

}

**Solution 2:  Topological Sort with comparison on the fly (10min)**

public class Solution {

/\*\*

\* @param org: a permutation of the integers from 1 to n

\* @param seqs: a list of sequences

\* @return: true if it can be reconstructed only one or false

\*/

public boolean sequenceReconstruction(int[] org, int[][] seqs) {

// Build graph

Map<Integer, List<Integer>> graph = buildGraph(seqs);

// Find initial indegree based on graph

Map<Integer, Integer> indegree = getIndegree(graph);

// Topological sort to find if unique path can be found from seqs as same as org

List<Integer> tmp = new ArrayList<Integer>();

Queue<Integer> q = new LinkedList<Integer>();

for(int node : indegree.keySet()) {

if(indegree.get(node) == 0) {

q.offer(node);

}

}

// Add variable 'level'(used in 'org' as index) to indicate if unique path

// is identical to 'org' during build process or not

int level = 0;

while(!q.isEmpty()) {

if(level >= org.length) {

return false;

}

int size = q.size();

// candidate next node must be only 1 choice, because it require unique path,

// if candidate next node(s) stored on queue more than 1, which means more

// than one unique path, not satisfy requirement

if(size > 1) {

return false;

}

int node = q.poll();

// If current node on unique path is different than org[level] (level equal to

// index start with 0 usage), no path avaiale

if(node != org[level]) {

return false;

}

tmp.add(node);

level++;

for(int neighbour : graph.get(node)) {

indegree.put(neighbour, indegree.get(neighbour) - 1);

if(indegree.get(neighbour) == 0) {

q.offer(neighbour);

}

}

}

// After all the unique path length(indicate by 'level') have to equal to 'org',

// otherwise even all nodes on unique path are identical to 'org', it may miss

// later section in 'org'

if(level != org.length) {

return false;

}

return true;

}

private Map<Integer, List<Integer>> buildGraph(int[][] seqs) {

Map<Integer, List<Integer>> graph = new HashMap<Integer, List<Integer>>();

for(int[] seq : seqs) {

for(int node : seq) {

graph.put(node, new ArrayList<Integer>());

}

}

// Note: seq length may > 2, that's why use for loop

// e.g Input:org = [4,1,5,2,6,3], seqs = [[5,2,6,3],[4,1,5,2]], Output:true

for(int[] seq : seqs) {

for(int i = 0; i < seq.length - 1; i++) {

graph.get(seq[i]).add(seq[i + 1]);

}

}

return graph;

}

private Map<Integer, Integer> getIndegree(Map<Integer, List<Integer>> graph) {

Map<Integer, Integer> indegree = new HashMap<Integer, Integer>();

for(int node : graph.keySet()) {

indegree.put(node, 0);

}

for(int node : graph.keySet()) {

for(int neighbour : graph.get(node)) {

indegree.put(neighbour, indegree.get(neighbour) + 1);

}

}

return indegree;

}

}

Time Complexity: O(n+m)

The overall time complexity of this basic algorithm is O(n+m). The O(n) comes from the number of times that the while loop (and initial for loop) is executed, and the O(m) from the nested for loop.

Although there is no way to calculate how many times the inner loop will be executed on any one iteration of the outer loop, it will only be executed once for each successor of each member, which means that the total number of times that it will be executed is the total number of successors of all the members -- or the total number of relations.

Space Complexity: O(n+m)

Space complexity is also O(n+m). The O(n) component comes from the predecessor count information stored for each member, and the maximum length of the auxiliary queue. The O(m) comes from storing the successors for each member; once again, the total number of successors is the number of relations, so O(m).

**More efficient solution refer to**

<https://www.lintcode.com/problem/605/solution/16613>

**和九章答案有一些细节的不同，一边build map一边检查，貌似更高效一些**

public class Solution {

public boolean sequenceReconstruction(int[] org, int[][] seqs) {

// write your code here

Map<Integer, List<Integer>> graph = new HashMap<>();

Map<Integer, Integer> indegree = new HashMap<>();

// build up the map

for (int[] seq : seqs) {

for (int i = 0; i < seq.length; i++) {

graph.putIfAbsent(seq[i], new ArrayList<Integer>());

indegree.putIfAbsent(seq[i], 0);

if (i > 0) {

graph.get(seq[i - 1]).add(seq[i]);

indegree.put(seq[i], indegree.get(seq[i]) + 1);

}

}

}

if (org.length != indegree.size()) {

return false;

}

// classical BFS

Queue<Integer> queue = new LinkedList<>();

for (int key : indegree.keySet()) {

if (indegree.get(key) == 0) {

queue.add(key);

}

}

int index = 0;

while (queue.size() == 1) {

int cur = queue.poll();

if (org[index++] != cur) {

return false;

}

for (int neighbor : graph.get(cur)) {

indegree.put(neighbor, indegree.get(neighbor) - 1);

if (indegree.get(neighbor) == 0) {

queue.add(neighbor);

}

}

}

return index == org.length;

}

}