Solution Review: Problem Challenge 1

We'll cover the following

Reconstructing a Sequence (hard)

Given a sequence `originalSeq` and an array of sequences, write a method to find if `originalSeq` can be uniquely reconstructed from the array of sequences.

Unique reconstruction means that we need to find if `originalSeq` is the only sequence such that all sequences in the array are subsequences of it.

Example 1:

```
Input: originalSeq: [1, 2, 3, 4], seqs: [[1, 2], [2, 3], [3, 4]]
Output: true
Explanation: The sequences [1, 2], [2, 3], and [3, 4] can uniquely reconstruct
[1, 2, 3, 4], in other words, all the given sequences uniquely define the order of
 numbers
in the 'originalSeq'.
```

Example 2:

```
Input: originalSeq: [1, 2, 3, 4], seqs: [[1, 2], [2, 3], [2, 4]]
Output: false
Explanation: The sequences [1, 2], [2, 3], and [2, 4] cannot uniquely reconstruct
[1, 2, 3, 4]. There are two possible sequences we can construct from the given seq
uences:
1) [1, 2, 3, 4]
2) [1, 2, 4, 3]
```

Example 3:

```
Input: originalSeq: [3, 1, 4, 2, 5], seqs: [[3, 1, 5], [1, 4, 2, 5]]
Output: true
Explanation: The sequences [3, 1, 5] and [1, 4, 2, 5] can uniquely reconstruct
[3, 1, 4, 2, 5].
```

Solution

Since each sequence in the given array defines the ordering of some numbers, we need to combine all these ordering rules to find two things:

1. Is it possible to construct the `originalSeq` from all these rules?
2. Are these ordering rules not sufficient enough to define the unique ordering of all the numbers in the `originalSeq`? In other words, can these rules result in more than one sequence?

Take Example-1:

```
originalSeq: [1, 2, 3, 4], seqs:[[1, 2], [2, 3], [3, 4]]
```

The first sequence tells us that '1' comes before '2'; the second sequence tells us that '2' comes before '3'; the third sequence tells us that '3' comes before '4'. Combining all these sequences will result in a unique sequence: [1, 2, 3, 4].

The above explanation tells us that we are actually asked to find the topological ordering of all the numbers and also to verify that there is only one topological ordering of the numbers possible from the given array of the sequences.

This makes the current problem similar to Tasks Scheduling Order (https://www.educative.io/collection/page/5668639101419520/5671464854355968/5066018374287360/) with two differences:

1. We need to build the graph of the numbers by comparing each pair of numbers in the given array of sequences.
2. We must perform the topological sort for the graph to determine two things:
   - Can the topological ordering construct the `originalSeq`?
   - That there is only one topological ordering of the numbers possible. This can be confirmed if we do not have more than one source at any time while finding the topological ordering of numbers.

Code

Here is what our algorithm will look like (only the highlighted lines have changed):

Java | Python3 | C++ | JS

```
from collections import deque


def can_construct(originalSeq, sequences):
  sortedOrder = []
  if len(originalSeq) <= 0:
    return False

  # a. Initialize the graph
  inDegree = {}  # count of incoming edges
  graph = {}  # adjacency list graph
  for sequence in sequences:
```

```
    for num in sequence:
      inDegree[num] = 0
      graph[num] = []

  # b. Build the graph
  for sequence in sequences:
    for i in range(1, len(sequence)):
      parent, child = sequence[i - 1], sequence[i]
      graph[parent].append(child)
      inDegree[child] += 1

  # if we don't have ordering rules for all the numbers we'll not able to uniquely constru
  if len(inDegree) != len(originalSeq):
    return False

  # c. Find all sources i.e., all vertices with 0 in-degrees
  sources = deque()
  for key in inDegree:
    if inDegree[key] == 0:
      sources.append(key)

  # d. For each source, add it to the sortedOrder and subtract one from all of its childre
  # if a child's in-degree becomes zero, add it to the sources queue
  while sources:
    if len(sources) > 1:
      return False  # more than one sources mean, there is more than one way to reconstruc
    if originalSeq[len(sortedOrder)] != sources[0]:
      # the next source(or number) is different from the original sequence
      return False

    vertex = sources.popleft()
    sortedOrder.append(vertex)
    for child in graph[vertex]:  # get the node's children to decrement their in-degrees
      inDegree[child] -= 1
      if inDegree[child] == 0:
        sources.append(child)

  # if sortedOrder's size is not equal to original sequence's size, there is no unique way
  return len(sortedOrder) == len(originalSeq)


def main():
  print("Can construct: " +
        str(can_construct([1, 2, 3, 4], [[1, 2], [2, 3], [3, 4]])))
  print("Can construct: " +
        str(can_construct([1, 2, 3, 4], [[1, 2], [2, 3], [2, 4]])))
  print("Can construct: " +
        str(can_construct([3, 1, 4, 2, 5], [[3, 1, 5], [1, 4, 2, 5]])))


main()
```

Time complexity #

In step ‘d’, each number can become a source only once and each edge (a rule) will be accessed and removed once. Therefore, the time complexity of the above algorithm will be $O(V + E)$, where ‘V’ is the count of distinct numbers and ‘E’ is the total number of the rules. Since, at most, each pair of numbers can give us one rule, we can conclude that the upper bound for the rules is $O(N)$ where 'N' is the count of numbers in all sequences. So, we can say that the time complexity of our algorithm is $O(V + N)$.

Space complexity

The space complexity will be $O(V + N)$, since we are storing all of the rules for each number in an adjacency list.

← Back

Problem Challenge 1

Next →

Problem Challenge 2

Completed
