Topological Sort

CS 223

1 Introduction

For this project you'll be using an implementation of a graph structure to generate a topological ordering of the graph. I am providing a simple graph implementation for you, although your code should work with anything that correctly implements the `Graph` abstract class.

Section 2 describes how topological sort works. Section 3 covers the requirements for your implementation. Submission details are covered in Section 4.

2 Topological Sort

Topological sort operates on directed graphs. Within the graph, an edge from v to u means that u depends on v. That is, we must complete whatever v represents before u can be completed. (For example, if we consider course prerequisites, CS 223 depends on CS 122, so there would be an edge from 122 to 223.) The goal of topological sort is to produce an ordering of the vertices so that for every edge (v, u), v precedes u in the ordering. (So, 122 would come before 223.)

2.1 The Algorithm

```
TopoSort(g)                                  ▷ g is some reasonable encoding of a digraph
  inbound ← new array of size |g.V|          ▷ Create an array of inbound edge counts
  for v ∈ g.V do                             ▷ For every vertex v in the graph
    for u ∈ v.adj() do                       ▷ For every u adjacent to v
      inbound[u] ← inbound[u] + 1            ▷ Increment u's inbound count
  for v ∈ g.V do
    if inbound[v] = 0 then
      add v to a queue                       ▷ Enqueue vertices with no inbound edges
  while queue is not empty do
    Dequeue vertex v
    Add v to ordering
    for u ∈ v.adj() do                       ▷ For every u adjacent to v
      inbound[u] ← inbound[u] − 1            ▷ Decrement u's inbound count
      if inbound[u] = 0 then                 ▷ If all dependencies are satisfied
        add u to the queue
  if All vertices are in ordering then return ordering
  else return null
```

3 Implementation

You will be implementing topological sort as a method in the `TopoSort` class. The type signature must be:

```
public static Iterable<Integer> sort(Graph g) {

}
```

As a result, you would call this method by doing something like this:

```
Iterable<Integer> order = TopoSort.sort(g);
```

This approach is used to allow us to sort any graph data structure that extends the Graph abstract class, rather than needing to make topological sorting part of the graph itself.

Your implementation must accept a single parameter of type Graph, and return either a topological ordering (via an iterable type) or null if an ordering is impossible. Specifically, you are to return null in three cases:

1. `null` is passed into your function
2. `g` is a graph, but it contains no vertices
3. `g` contains a cycle and is therefore not topologically orderable.

You may assume that there is at most one edge between any pair of vertices. Loops are permitted in the input graph, so be careful of them.

3.1 Hint: `LinkedLists`

Java provides a `LinkedList` class which has two notable features. First, it can be used as a queue by calling its `.add()` and `.remove()` methods. Second, `LinkedList`s are `Iterable`, and are therefore suitable for returning from the `.sort()` method.

3.2 Hint: Java's foreach Construct

Java provides a variant of the `for` loop which iterates over `Iterable` objects for you, giving back each item in the object in turn. For example:

```
for(int u: g.adj(v)) {
  // runs once for each u adjacent to v
}
```

4 Submitting your solution

You will be submitted exactly one file for this project: `TopoSort.java`. Once you have implemented this project to your satisfaction, submit the file, uncompressed, to Autolab. Your submission will be passed through automatic testing, so you should get feedback on functionality quickly.