Homework # 12
due Monday, December 2, 10:00 PM

In this assignment, you will implement a graph-based ADT called `ProfileGraph`, including "depth-first" and "breath-first" search to find connections between profiles in a graph. Please (re-)read Chapter 14 in the textbook.

1 Concerning the `ProfileGraph` ADT

The `ProfileGraph` ADT will represent a directed graph of user profiles. It will support the following operations:

numNodes() Return the number of nodes (profiles) in the graph.

numEdges() Return the number of edges (relationships) in the graph.

addNode(Profile p) Add a node (not a null or duplicate) to the graph, and return whether a node was added.

addEdge(Profile p1, Profile p2) Add an edge (not a duplicate or self-edge) from p1 to p2 to the graph. Do nothing if either p1 or p2 is invalid. Add p1 and/or p2 to the graph if necessary. Return whether an edge was added.

containsNode(Profile p) Return whether p is in the graph.

containsEdge(Profile p1, Profile p2) Return whether there is an edge from p1 to p2 in the graph.

removeNode(Profile p) Remove the node p from the graph, and all edges to or from p. Return whether a node was removed.

removeEdge(Profile p1, Profile p2) Remove the edge from p1 to p2 from the graph. Return whether an edge was removed.

nodeList() Return a list of all the nodes in the graph.

connectedTo(Profile p1, Profile p2) Return whether p2 is reachable from p1 in the graph.

connectedToAvoiding(Profile p1, Profile p2, Profile... profiles) Return whether p2 is reachable from p1 without using the nodes in profiles.

pathTo(Profile p1, Profile p2) Return the path from p1 to p2 as a list of `Profile`s, or null if there isn't one.

pathToAvoiding(Profile p1, Profile p2, Profile... profiles) Return the path from p1 to p2 without using the nodes in profiles as a list of `Profile`s, or null if there isn't one.

These operations will treat any two `Profile` objects that are equal to one another as representing the same (identical) node. It will never expose any of its internal `Profile` objects, so that mutating a `Profile` will not change the graph.

2 Concerning Search Algorithms

Section 14.3 of the textbook describes two important graph algorithms: "depth-first search" and "breadth-first search." As described on page 731, these algorithms can be used to find a path from one node to another in a graph. In our case, the nodes are profiles and edges are relationships between them.

Depth-first search is implemented using a stack (LIFO) to keep track of seen but unvisited nodes, while breadth-first using a queue (FIFO). Breadth-first will always find a shortest path, while depth-first will often use less memory. In "Artificial Intelligence" (CompSci 422) you can learn about more efficient ways to find the best path.

3 Implementation of the Graph

We will repesent our graph using an edge-list representation. In fact, we already have a class that will be very useful for this. The `ProfileMap` ADT from the previous homework is a map of associations between a `Profile` and a list of `Profile` objects with which it has a relationship. So we will use this class to do most of the work of changing the graph. For example, adding a node to the graph means adding a `Profile` to the map with an empty list. Adding an edge to the graph means adding a `Profile` to the list for some key in the map.

In some cases, there is not a perfect correspondence between methods. For example, `ProfileMap`'s `add` method throws an exception on a null argument, while `ProfileGraph`'s `addNode` method does not. Make sure to handle these cases appropriately.

As stated above, the nodes in the graph should never be exposed, but the underlying map stores its data as `Profile` objects. In the `ProfileMap` the keys were never exposed, but the values were. This meant that a `Profile` could have a relationship with a `Profile` not in the map, and that the map could be modified through operations performed on the value lists. In the `ProfileGraph` this will not be the case. So make sure any object references passed into or returned from object methods on the `ProfileGraph` do not refer to `Profile` objects that are stored in the underlying map.

4 Implementation for Search

The search algorithm will find paths between `Profile`s. A path is a series of connected `Profile`s. We give the path with as a `List` of `Profile` objects;

Four given methods use the private helper method `search`. This method takes a starting and destination `Profile`, a boolean flag to determine BFS or DFS, and an optional series of `Profiles` to be avoided in search. Unlike in the textbook, we break off the search if we find the destination node and return the path from the starting coordinate to the destination. If the destination is never found, we return null.

We provide three useful objects to use during search.

worklist A `LinkedList` of `SearchLink`s. This should hold the nodes we have seen but have not yet visited, as `SearchLink`s, which hold the node itself, as well as which node it was seen from. Storing which node it was seen from is essential for reconstructing the path later. This `LinkedList` can serve as either a stack or queue using its `addFirst` and `addLast` operations, so use it as appropriate depending on the `breadth` flag.

visited A `Map` of `Profile` to `Profile`. When we visit a node, we again want to remember which node we visited it from. However, unlike the `worklist`, we visit each node only once, so we can use a map to represent our backwards links.

avoiding A `Set` of `Profile`s. This will be useful to store the nodes that are to be avoided during search. It will allow to use methods like `avoiding.contains(p)`.

If you follow this design, constructing the path should be easy. At the end of search, the visited map will contain each node that was visited exactly only once as a key, with its previous node as a value. Therefore there will be only one backwards path towards the starting node from any given node.

5 What You Need To Do

Implement all incomplete methods in `ProfileGraph.java`.

6 Files

The git repository has an Eclipse project with the following files:

src/TestInvariant.java Units tests for the new invariant.

src/TestProfileGraph.java Unit tests of the ProfileGraph ADT.

src/TestEfficiency.java Each test should pass within a few seconds.

src/edu/uwm/cs351/ProfileGraph.java Skeleton of ProfileGraph ADT.

src/edu/uwm/cs351/util/ProfileMap.java Solution from previous homework.

src/edu/uwm/cs351/util/Profile.java Solution from previous homework.

src/edu/uwm/cs351/util/Primes.java Solution from previous homework.