Homework 6, CSE 232

Due October 8th Note: On most of the problem sets through the semester, I'll put a horizontal line with "Optional" under it. Any problems below this section are encouraged - I think they're interesting and will help you learn the subject - but not necessary to complete in order to get credit for the homework.

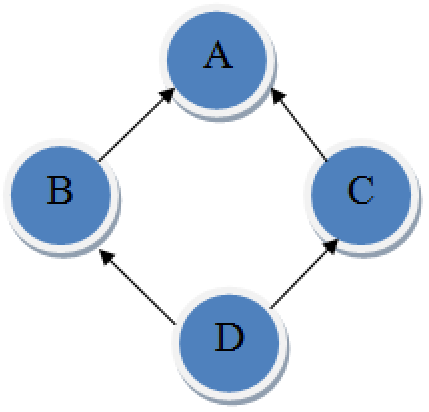
Problem 1

This week's problem is GCJ Round 1C 2012 - Diamond Inheritance.

Here's the problem statement:

> You are asked to help diagnose class diagrams to identify instances of diamond inheritance. The following example class diagram illustrates the property of diamond inheritance. There are four classes: A, B, C and D. An arrow pointing from X to Y indicates that class X inherits from class Y.

```
In [1]: from IPython.display import Image, display
        display(Image(filename='./week6_hwimg.png'))
```



> In this class diagram, D inherits from both B and C, B inherits from A, and C also inherits from A. An inheritance path from X to Y is defined as a sequence of classes X, C1, C2, C3, . . . , Cn, Y where X inherits from C1, Ci inherits from Ci + 1 for 1 ≤ i ≤ n - 1, and Cn inherits from Y. There are two inheritance paths from D to A in the example above. The first path is D, B, A and the second path is D, C, A.
>
> A class diagram is said to contain a diamond inheritance if there exists a pair of classes X and Y such that there are at least two different inheritance paths from X to Y. The above class diagram is a classic example of diamond inheritance. Your task is to determine whether or not a given class diagram contains a diamond inheritance.

Input

> The first line of the input gives the number of test cases, T. T test cases follow, each specifies a class diagram. The first line of each test case gives the number of classes in this diagram, N. The classes are numbered from 1 to N. N lines follow. The ith line starts with a non-negative integer Mi indicating the number of classes that class i inherits from. This is followed by Mi distinct positive integers each from 1 to N representing those classes. You may assume that:
>
> If there is an inheritance path from X to Y then there is no inheritance path from Y to X. A class will never inherit from itself.

Output

For each diagram, output one line containing "Case #x: y", where x is the case number (starting from 1) and y is "Yes" if the class diagram contains a diamond inheritance, "No" otherwise.

Limits

$1 \leq$ T ≤ 50.

$0 \leq$ Mi ≤ 10.

Small dataset

$1 \leq$ N ≤ 50.

Large dataset

$1 \leq$ N $\leq$ 1,000.

a) How can we represent the class inheritance as a graph? What are the nodes of the graph? What are the edges of the graph? We talked about several different ways to classify graphs. Classify the graph of class inheritance according to these distinctions. Is this graph directed or undirected, weighted or unweighted, simple or non-simple, and cyclic or acyclic?

The nodes of the graph will represent the different classes, with edge (X, Y) present in the graph if class X inherits from class Y.

This graph is directed (edges are not symmetric), unweighted (there's no numerical value associated with the edges), simple (there are explicitly no self-loops and multiedges), and acyclic (there are explicitly no inheritance path cycles).

b) Compare the maximum number of vertices and edges in the graph under the Small and Large input limits. How much memory would be required to represent this graph when stored as either an adjacency matrix or an adjacency list in the Large input limits?

The maximum number of vertices are given as 50 in the small limits and 1,000 in the large limits. Each vertex can have at most 10 edges connecting it, for a maximum of 500 edges in the small and 10,000 in the large limits. An adjacency matrix would require V^2 space, while an adjacency list would require $V + E$ space. If each element takes 4 bytes, then the matrix would take 4 million bytes, or about 4 megabytes, while the list would take only 44,000 bytes, or about 44 kilobytes, several orders of magnitude less space.

c) Suppose we pick a starting vertex and try to traverse the graph from that point. How will the vertices we reach be related to the starting vertex? Will this depend on whether we use a breadth-first or a depth-first search?

If we search starting from a particular vertex, all of the vertices we find will represent classes from which the starting class inherits, either directly ($X \rightarrow Y$) or indirectly ($X \rightarrow \ldots \rightarrow Y$). Moreover, we are guaranteed to find *all* of the classes from which the starting class inherits. Both the breadth-first and the depth-first searches will reach the same vertices, just in a different order.

d) How many times will any end vertex Y show up in your search from a particular start vertex X? How does this depend on the inheritance paths between X and Y? Write a solution to the problem using a modified depth-first or breadth-first search and attach it to your homework. Don't forget to run your code on the Small and Large inputs to make sure it works.

Each end vertex Y will show up in a search from X if and only if X inherits from Y. Each time we find Y in our search, that will mark a new path of inheritance between X and Y. That means that if we find at least two separate paths between any pair of vertices, the graph must have diamond inheritance. We can solve this by performing a depth-first search starting from each possible vertex and returning True in our search if we ever find a second inheritance path for any vertex.

```
In []: import sys

       def read_case(infile):
           N = int(infile.readline().strip())
           adjacency_list = []
           for i in range(N):
               adjacency_list.append([])
               line = infile.readline().split()
               for j in range(int(line[0])):
                   adjacency_list[i].append(int(line[j+1]) - 1) #0-index
           return adjacency_list

       def dfs(graph, root):
           stack = [root]
           reached = [False for i in range(len(graph))]
           reached[root] = True
           while stack:
               root = stack.pop()
               for child in graph[root]:
                   if reached[child]:
                       return True
                   reached[child] = True
                   stack.append(child)
           return False

       def is_diamond(graph):
           for i in range(len(graph)):
               if dfs(graph, i):
                   return "Yes"
           return "No"

       infile = open("%s" % sys.argv[1], 'r')
       outfile = open("%s.out" % sys.argv[1][:-3], 'w')
       cases = int(infile.readline().strip())
       for i in range(cases):
           adjacency_list = read_case(infile)
           output = is_diamond(adjacency_list)
           outfile.write("Case #%i: %s\n" % (i+1, output))
           print "Case #%i: %s" % (i+1, output)
       infile.close()
       outfile.close()
```

Optional No optional work this week.