Sample Solution to Quiz 3

Name: _____

(10 points) Consistency of constraints

The following problem occurs in program analysis. For a set of n variables $x_1, \ldots, x_n$, you are given a set of m constraints, of one of two forms: *equality*, of the form $x_i = x_j$; *strict inequality*, of the form $x_i > x_j$. The goal of this problem is to determine whether a given set of constraints can be satisfied.

For example, the following set of constraints can be satisfied.

$$x_1 = x_2; x_1 = x_3; x_2 > x_4; x_3 > x_5$$

One can set $x_1 = x_2 = x_3 = 1$, $x_4 = 0$, and $x_5 = 0$.

The following set of constraints, however, cannot be satisfied.

$$x_1 = x_2; x_1 = x_3; x_2 > x_4; x_4 > x_5; x_5 > x_3$$

Give an algorithm that takes as input m constraints over n variables and determines whether the constraints can be satisfied. Your algorithm only needs to return a "yes/no" answer ("yes" if the constraints can be satisfied; and "no" otherwise).

State the running time of your algorithm. You do not need to prove the correctness of the algorithm. The more efficient your algorithm is, in terms of its worst case running time as a function of n and m, the more credit you will get.

Answer: One can give several linear-time algorithms for this problem.

Algorithm 1

Consider the n variables as vertices and the m constraints as edges in a directed graph, in which we add edge $x_i \to x_j$ and $x_j \to x_j$ for each equality constraint $x_i = x_j$, and the edge $x_i \to x_j$ for each strict inequality constraint $x_i > x_j$. We run the linear-time algorithm to determine the strongly connected components. If we find two vertices x_i and x_j in the same strongly connected component with a strict inequality constraint $x_i > x_j$, then the set of constraints is not consistent; otherwise, the set is consistent. This is clearly a linear-time algorithm.

Algorithm 2

Here is another algorithm. We can consider the n variables as vertices and the m constraints as edges in the graph, in which the edge between two *equality* nodes is an undirected edge, and the

edge between two *strict inequality* nodes is a directed edge. So we have a graph containing both undirected and directed edges.

1) We can merge the *equality* nodes into a new node, and add this node into the graph. And keep those *strict inequality* nodes in the graph. In this way we get a directed graph.

2) So the problem becomes to find out whether the new graph is cyclic.

   a) We'll do a DFS on the new graph to find whether there is a cycle, marking vertices along the way, i.e. you should maintain a global array to record whether this node is visited or not.

   b) Initially all the nodes are not visited. We start from the first node in the array and begins a DFS traverse. If you hit a vertex that you have already marked, then you have a cycle; else we continue from the next unvisited node in the array, until the end of array.

   c) If there is no cycle in the new graph, it returns "yes" which means the constraints can be satisfied; and "no" otherwise.

   Note: Here we don't recommend to use BFS to find a cycle, since it needs extra effort to maintain the visited nodes during the recursion. Also if you use topological sorting, for its original version to work, the graph must be a DAG(directed acyclic graph). You should have to adapt it to solve the cyclic graph.

In the worst case, step 1 takes $O(m)$ time, and step 2 takes $O(n)$ time. Thus totally the running time will $O(m+n)$.