CSC 225 - SPRING 2016
ALGORITHMS AND DATA STRUCTURES I
PROGRAMMING ASSIGNMENT 4
UNIVERSITY OF VICTORIA

1 Programming Assignment

Given an adjacency matrix for graph G, your assignment is to determine whether G is strongly connected. If G is strongly connected, return true. Otherwise, return false. Your algorithm should run in $O(n^2)$ time.

Input: An adjacency matrix of directed graph G.
Output: A boolean value (true or false).

An undirected graph is considered *connected* when there exists a path between every pair of nodes in the graph, that is for all vertices $v, w \in G$ there exists a path between v and w. Connectivity can be confirmed through a DSF traversal. A directed graph is considered *weakly connected* if the underlying undirected graph is connected and *strongly connected* if there exists a directed path between all vertices. Specifically, for all vertices $v, w \in G$ there must exists a path from v to w and from w to v.

A DFS traversal on a directed graph will only check the 'outbound' paths from a node v not the inbound, meaning one DFS traversal is not sufficient for testing connectivity on a directed graph. There is $O(n^3)$ algorithm to confirm strong connectivity which runs a DFS traversal from every node on the graph, however a faster algorithm exists.

Not every unique path must be confirmed individually, if there exists a path from w to v and a path from v to u, then a wu-path runs through v. If G is strongly connected, there must exist a wv-path and a vu-path for all $w, v, u \in G$, therefore strong connectivity can be tested by simply verifying that all vertices in G, and in the reverse of G, G^T, are visited by traversals rooted in an arbitrary starting point v. The spanning tree generated by DFS (or BFS) on G rooted at a starting vertex $v \in G$ gives a set of outbound paths from v to other vertices $w \in G$. A set of inbound paths from other vertices to v can be obtained by running DFS on the graph G^T, a graph constructed by reversing the direction of all arcs in G. This graph is called the transpose graph of G because the adjacency matrix of G^T is the transpose of the adjacency matrix of G. Pseudocode for this algorithm is given below.

A Java template has been provided containing an empty function `strongly_Connected`, which takes the adjacency matrix as its only argument. Your task is to write the body of the `strongly_Connected` function. You must use the provided Java template as the basis of your submission, and put your implementation inside the `strongly_Connected` function in the template. You may not change the name, return type or parameters of the `strongly_Connected` function. You may add additional functions as needed. You are not permitted to change any aspect of the

`strongly_Connected` class (including adding, removing, or renaming its contents).

The `main` function in the template contains code to help you test your implementation by entering test data or reading it from a file. You may modify the `main` function, but only the contents of the `strongly_Connected` function (and any functions you have added) will be marked, since the `main` function will be deleted before marking begins. Please read through the comments in the template file before starting.

2 Test Datasets

A set of input files containing test data are available on conneX. You should ensure that your implementation gives the correct answer on these test files before submitting. It may also be helpful to test your implementation on a variety of other inputs, since the posted data may not cover all possible cases.

3 Evaluation Criteria

The programming assignment will be marked out of 40, based on a combination of automated testing (using large test arrays similar to the ones posted on conneX) and human inspection.

Score	**Description**
0 - 8	Submission does not compile or does not conform to the provided template.
9 - 17	The implemented algrorithm does not test connectivity or is substantially inaccurate on the tested inputs.
17 - 25	The implemented algorithm tests weak connectivity or is substantially inaccurate on the tested inputs.
25 - 33	The implemented algorithm is correct and has a $O(n^3)$ running time.
33 - 40	The implemented algorithm is correct and has a $O(n^2)$ running time.

To be properly tested, every submission must compile correctly as submitted, and must be based on the provided template. **If your submission does not compile for any reason (even trivial mistakes like typos), or was not based on the template, it will receive at most 8 out of 40.** The best way to make sure your submission is correct is to download it from conneX after submitting and test it. You are not permitted to revise your submission after the due date, and late submissions will not be accepted, so you should ensure that you have submitted the correct version of your code before the due date. conneX will allow you to change your submission before the due date if you notice a mistake. After submitting your assignment, conneX will automatically send you a confirmation email. If you do not receive such an email, your submission was not received. If you have problems with the submission process, send an email to the instructor **before** the due date.

Algorithm 1 Test Whether a Directed Graph G is Strongly Connected

```
procedure DFS(G, covered, current)
    covered[current] ← true
    totalCovered ← 1
    for each outbound neighbor v of current do
        if covered[v] = true then
            {v has already been visited}
            Continue
        end if
        totalCovered ← totalCovered + DFS(G, covered, v)
    end for
    return totalCovered
end procedure

procedure DFSTranpose(G, covered, current)
    covered[current] ← true
    totalCovered ← 1
    for each inbound neighbor v of current do
        if covered[v] = true then
            {v has already been visited}
            Continue
        end if
        totalCovered ← totalCovered + DFSTranspose(G, covered, v)
    end for
    return totalCovered
end procedure

procedure StrongConnectivityTest(G, covered, current)
    n ← |G|
    covered ← Array of size n, initialized to false
    v ← An arbitrary vertex of G
    {Traverse arcs from v in G}
    if DFS(G, covered, v) < n then
        return false
    end if
    Reset every value of covered to false
    {Traverse arcs from v in G^T}
    if DFS(G^T, covered, v) < n then
        return false
    end if
    return true
end procedure
```