CSCI-311 Assignment 11 BFS, DFS

Download Lab12.tar and untar it using the command: tar -xvf Lab12.tar

You will implement BFS and DFS on a directed graph. First, you will write class Graph. In addition, you will implement some member functions that solve specific problems on a directed graph.

Step 1. Write graph.h and graph.cpp files where you will declare and define class Graph.

Class Graph will have a default constructor that will read input from the standard input (keyboard) using ***cin.*** The input will be given in this format:

```
N M
1 3
4 5
0 2
3 5
2 3
```

Where N is an integer, the total number of vertices (nodes) in a graph and M is an integer, the total number of edges in the graph.
Here are the data members of class Graph:

```
vector< vector<int> > Adj; //adjacency lists of the graph
vector< int > distance; //for BFS
vector<int> parents; //for BFS and DFS
vector<char> colors; //for DFS
vector<TimeStamp> stamps; //for DFS
int size; //total vertices
```

Your program will read input integer N and will resize *Adj* to size N. Then your program will read M, and will run *for* loop to read in M pairs (u, v) and will push *v* into *Adj*[u]. The default constructor also needs to resize the rest of vectors to size N. **This is important**: after the default constructor has been called, all vectors of the class must have size N.

You are given another class ***TimeStamp***, with the following declaration in ***timestamp.h*** file:

```
class TimeStamp{
            public:
            TimeStamp(): d(0), f(0){};
            int d; //discovery time
            int f; //finish time
}
```

Step 2. Implement BFS and DFS and other member functions for class Graph.

Member function	**Description**	**Test files**
`void printGraph();`	This function prints the adjacency lists in this format u: v1 v2 v3 where u is the vertex whose Adjacency list is printed, and vertices v1, v2 and v3 are the vertices in the Adjacency list of u. There is a colon after u and then space after the colon, and space after each vertex in Adj[u], and then ***endl***	t00
`void printNeighbors(int u);`	This function takes an integer *u* as a parameter, *u* is a vertex in the graph. The function prints neighbors of *u* (i.e. vertices *v* such that there is an edge from *u* to *v* in the graph) in the format: <vertex><space>…<vertex><space><endl>	t01
`bool isNeighbor(int u, int v);`	This function takes two vertices as parameters, *u* and *v*, and returns *true* if there is an edge from *u* to *v* in the graph.	t02

void bfs(int s)	This function implements BFS algorithm. The parameter *s* is
	the source, from which BFS starts running.
	It initializes *distance* array to INT_MAX (defined in <*climits*>, so you need to include <*climits*>)
	It initializes *parents*[i] to i.
	It uses <queue> of the standard library (include it).
	To use queue, just do:
	queue<int> aq; //aq is the name of the queue
	Uses pseudocode from lecture notes.
	bfs will print out a node as it pops the node from the queue.
	cout << node << " " ;
	After queue is empty, then print out ***endl***
		t03

void dfs() void dfsVisit(int u, int &atime)	This function implements DFS algorithm. dfs() initializes arrays *colors, parents, stamps,* and contains the main *for* loop, from which *dfsVisit* is called on each node whose color is White. Parameter *u* of *dfsVisit* is the current node, and parameter *atime* is the current time stamp used. DFS will print out a node inside dfsVisit before processing Adj[u]: cout << u << " " ; After the main *for* loop, print out ***endl***	t04
void printPath(int v) *You may have a helper function in addition*	This function must be called after bfs is called, so that parents array has been calculated by the time printPath is called.	t05

	Parameter *v* is the node from which we start backtracking the path from the source *s* to *v.* Path is printed in the format: <node><space>…<node><space><endl>	
void printLevels(*int s*)	This function uses code of bfs, but modifies it slightly by using queue< pair<int, int> > where *first* in pair is a node, and *second* is the distance from the source to this node. Include <utility> to use *C++ pair.* When enqueueing a node into the queue, make a pair: pair<int, int>(node, dist) and push it into the queue. Nodes at the same levels will be in consecutive order inside the queue. You will need to maintain a variable, current level being printed, and when the next popped node from the queue has a different distance than the current level, then reset this variable to the new level, print out ***endl*** after the last level and only then print the node on the new line.	t06
bool isCycle() bool isCycleVisit(int u, int & atime)	This function uses code of *dfs* and *dfs_visit*, but it will modify this code. Whenever a node *v* in Adj[u] is found such that *v* is Grey, this means that *v* is an ancestor of *u* in the DFS-tree, and that there exists the path from *v* to *u*, and edge (u, v) concludes the cycle consisting of this path and edge (u, v). At this point (Grey vertex has been discovered), return *true.* Stop traversal after *true* has been returned. If the main *for* loop has been finished, and every time *isCycleVisit* has been called, it returned *false,* then return *false.* This function does not print anything, just returns *Boolean.*	t07-t10
Test t11 tests all the functions above		t11

Submission: Submit graph.h and graph.cpp on turnin to Assignment11.
Grading: If your program does not compile, you will receive 0 (no partial credit). Otherwise, grading will be done according to the table below:

Function	Tests	Points for these tests
printGraph	t00	0
printNeighbors	t01	0
isNeighbor	t02	0
Bfs	t03	10
Dfs	t04	10
printPath	t05	10
printLevels	t06	25
isCycle	t07-t10	25
All	t11	20