Assignment 7: Graphs and breadth-first search

In this assignment, you will construct an adjacency-list implementation of a directed, unweighted graph, and then use it to implement the breadth-first and depth-first search algorithms.

The Adjacency-List Graph Representation

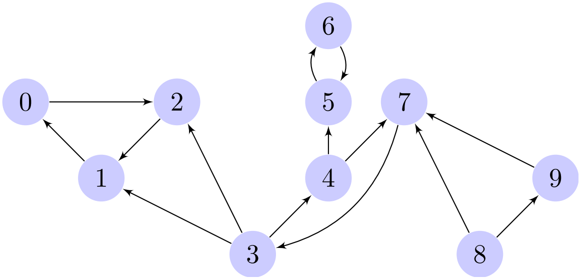
Write an implementation of the *adjacency list* graph representation. You may assume that nodes are integers which start at `0`. You should use the following graph class:

```
#pragma once
/*
 * graph.hpp
 * Adjacency-list graph implementation
 */
#include <climits> // For INT_MAX, INT_MIN
#include <list>
#include <vector>

class graph {
  public:
    /* graph(n)
       Construct a graph with n nodes and no edges, initially.
    */
    graph(int n);

    /* add_edge(a,b)
       Add an edge from node a to node b. Note that self edges are not allowed,
       so attempting add_edge(a,a) should be ignored. Similarly, this is not
       a multigraph, so if an edge a -> b already exists, a second one should
       be ignored.
       Should run in O(E) time in the worst case.
    */
    void add_edge(int a, int b);

    /* has_edge(a,b)
       Returns true if there is an edge from a to b. Should return false if
       either a or b is out-of-range (< 0 or >= count_nodes()).
       Should run in O(E) time.
    */
    bool has_edge(int a, int b);

    /* count_nodes()
       Returns the number of nodes in this graph.
       Should run in O(1) time
    */
    int count_nodes();

    /* count_edges()
       Returns the total number of edges in this graph.
       Should run in O(E) time.
    */
    int count_edges();
    /* count_edges(n)
       Returns the number of outbound edges from node n.
       Should run in O(E) time
    */
    int count_edges(int n);

    /* bfs(n)
       Perform a breadth-first search, starting at node n, and returning a
       vector that gives the distance to every other node. (If a node is
       unreachable from n, then set its distance to INT_MAX.)
       Should run in O(E + N) time.
    */
    std::vector<int> bfs(int n);

    /* is_connected(a,b)
       Returns true if a path exists from node a to b.
       Should run in O(E + N) time.
    */
    bool is_connected(int a, int b);

  private:

    // Add any private data/function members you need.

};
```

You should save the above class definition into the file `graph.hpp`.

Feel free to use the standard library `queue` class for implementing the BFS, and to use the standard library list classes (`list` for doubly-linked, `forward_list` for singly-linked) if you want.

When you compile, link with `assign7_test.cpp`. The test runner will test the core functionality of your class, and will then test both the BFS and DFS on the following graph:



although your graph should be able to work with *any* directed graph.

(You can also find both files on the server, under `/usr/local/class/src/`.)

Submission

Save your work in a directory on the server named `cs133s/assign7/`.

Andrew Clifton, Computer Science
instructor
aclifton@fullcoll.edu