```
#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:
      std::vector<std::list<int>> adj_list;

  };
```