/\*

\* assign7\_test.cpp

\* Assignment 7 test code.

\*/

#include <iostream>

#include <vector>

using std::cout;

using std::endl;

using std::vector;

#include "graph.hpp"

// Quick way to check if condition is false; if so, print message and return false

#define check(cond,msg) {if(!(cond)) { std::cout << "FAILED: " << msg << std::endl; return false; }}

// Test core graph functionality

bool test\_core\_graph() {

cout << "Testing basic functionality...\n";

graph g\_empty{0};

check(g\_empty.count\_nodes() == 0, "Empty graph should have 0 nodes");

check(g\_empty.count\_edges() == 0, "Empty graph should have 0 edges");

graph g{10};

check(g.count\_nodes() == 10, "10-node graph does not have 10 nodes");

check(g.count\_edges() == 0, "10-node graph should not have any edges");

// Try adding a normal edge

g.add\_edge(1,2);

check(g.has\_edge(1,2), "Edge 1 -> 2 added but does not exist");

check(g.count\_edges() == 1, "Edge count should be 1 after adding edge");

g.add\_edge(1,1);

check(!g.has\_edge(1,1), "Self edges should not be created");

check(g.count\_edges() == 1, "Edge count should be unchanged after adding invalid edge");

// Try to add a duplicate edge

g.add\_edge(1,2);

check(g.count\_edges() == 1, "Edge count should be unchaged after adding duplicate edge");

return true;

}

// Test constructing the sample graph

bool test\_sample\_graph() {

graph g{10};

cout << "Building sample graph...\n";

g.add\_edge(0,2);

g.add\_edge(1,0);

g.add\_edge(2,1);

g.add\_edge(3,1);

g.add\_edge(3,2);

g.add\_edge(3,4);

g.add\_edge(4,5);

g.add\_edge(4,7);

g.add\_edge(5,6);

g.add\_edge(6,5);

g.add\_edge(7,3);

g.add\_edge(8,7);

g.add\_edge(8,9);

g.add\_edge(9,7);

check(g.count\_edges() == 14, "Sample graph should have 14 edges in total");

cout << "Checking BFS... ";

// Check BFS distances.

// This array defines, for every possible starting node and other node,

// what the distance should be

const int IM = INT\_MAX;

int bfs\_dist[10][10] = {

// 0 1 2 3 4 5 6 7 8 9

{0, 2, 1, IM, IM, IM, IM, IM, IM, IM}, // 0

{1, 0, 2, IM, IM, IM, IM, IM, IM, IM}, // 1

{2, 1, 0, IM, IM, IM, IM, IM, IM, IM}, // 2

{2, 1, 1, 0, 1, 2, 3, 2, IM, IM}, // 3

{4, 3, 3, 2, 0, 1, 2, 1, IM, IM}, // 4

{IM, IM, IM, IM, IM, 0, 1, IM, IM, IM}, // 5

{IM, IM, IM, IM, IM, 1, 0, IM, IM, IM}, // 6

{3, 2, 2, 1, 2, 3, 4, 0, IM, IM}, // 7

{4, 3, 3, 2, 3, 4, 5, 1, 0, 1}, // 8

{4, 3, 3, 2, 3, 4, 5, 1, IM, 0}, // 9

};

for(int a = 0; a < 10; ++a) {

cout << a << " ";

vector<int> ds = g.bfs(a);

for(int b = 0; b < 10; ++b) {

check(bfs\_dist[a][b] == ds.at(b),

"BFS distance from " << a << " to " << b << " is wrong " <<

"(should be " << bfs\_dist[a][b] << " but got " << ds.at(b) << ")");

}

}

cout << endl;

cout << "Checking connectivity... ";

// To check connectivity, we reuse the BFS distances. Two nodes should be

// connected if their distance is finite.

for(int a = 0; a < 10; ++a) {

cout << a << " ";

for(int b = 0; b < 10; ++b)

check(g.is\_connected(a,b) == (bfs\_dist[a][b] < IM),

"Nodes " << a << " and " << b << " should be connected");

}

cout << endl;

return true;

}

int main() {

std::cout << "---- Starting graph tests ----" << std::endl;

if(test\_core\_graph() && test\_sample\_graph()) {

std::cout << "---- All tests passed successfully ----" << std::endl;

return 0;

}

else

return 1;

}