Wrestle Mania

Larry Bush

October 24, 2002

Abstract

This paper presents a generic function to find the 2-coloring of a graph. It is implemented as a C++ template function that uses the Boost Graph Library (BGL). The BGL is a generic library specifically created to solve graph problems[2]. A test program is also presented. This document is a presentation of solutions to the questions presented in [1].

Contents

1 Introduction

The following graph problem is presented in the referenced homework assignment [1].

> "There are two types of professional wrestlers: "good guys" and "bad guys." Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n professional wrestlers and we have a list of r pairs of wrestlers for which there are rivalries. Give an $O(n + r)$-time algorithm that determines whether it is possible to designate some of the wrestlers as good guys and the remainder as bad guys such that each rivalry is between a good guy and a bad guy. If it is possible to perform such a designation, your algorithm should produce it."[1][3]

This problem is a "2-coloring" problem which can be solved using an implementation of the breadth first search algorithm. This abstract solution is presented in 2.

This paper presents an implementation based on that abstract solution to this problem. The solution presented uses a generic function to find the 2-coloring of the graph. The 2-coloring function is implemented as a C++ template function that uses the Boost Graph Library (BGL). As stated above, the BGL is a generic library specifically created to solve graph problems[2].

The solution uses one of the BGL components know as the breadth first visit algorithm. This algorithm is a variation on the breadth first search. The difference between the two is that the breadth first visit does not initialize the verticies as part of the algorithm. Therefore, it can be applied repeatedly to a given graph without reinitializing the status of each vertex. Thus, the results of the application will reflect the culmination of all of the iterative applications of the breadth first visit algorithm to the given graph.

This paper provides a generic solution the the 2-coloring problem. It then applies this solution to the Wrestler Rivalry problem.

A test function is also presented in this paper. The test function takes a graph and label map as its input. The label map is a mapping from verticies to a GOOD or BAD denotation. The mapping should be a 2-coloring. The test function determines if it is indeed a 2-coloring.

This paper presents two programs which use the above functions. The first program solves the Wrestler Rivalry problem using the find two coloring function. That program also runs the test function on the solution and outputs the results. The second program is primarily intended as a test of the find two coloring funtion. This is discussed in 4. The test entails creating a random graph of the appropriate type, running the find two coloring function on it and then testing the result using the find two coloring test function.

The test function determines if the result is a valid 2-coloring of the given graph. The test function does not determine if a valid 2-coloring actually exists, it only tests the given 2-coloring. The test function is only run if the 2-coloring reports that it was able to find a valid solution. This is discussed in 5.

Instructions for compiling and running this program are included in 3.

2 2-Coloring Solution Description

The following is the abstract solution description, given in [1]. It is stated in terms of the original wrestler rivalry formulation.

> "Create an undirected graph G = (V,E) in which the vertices V are the wrestlers names and the edges represent rivalries (i.e.,there is an edge $(u, v) contained in E$ if and only if there is a rivalry between wrestler u and wrestler v). Thus $|V| = n$ and $|E| = r$. Pick a vertex as a source vertex and begin running BFS, labeling the source vertex as "good" and each vertex adjacent to it as "bad." For each such "bad" vertex label each of its adjacent white vertices as "good," and continue in this way, alternating between "good" and "bad" according to whether the distance between the source and the vertex is even or odd. But if a vertex has an adjacent gray or black vertex that is labeled the same as itself, stop with the result that there is no solution. If this first BFS completes, it has successfully labeled all vertices in the connected component containing the chosen source vertex. Repeat this process for each of the connected components; i.e., choose any remaining white vertex as the source (skip the usual initialization loop that initializes all nodes to white). If all of these modified BFSs successfully complete, then the resulting labeling of the vertices as good or bad is a solution; if any of them fails, stop and report there is no solution.
>
> Correctness argument: First, if the algorithm stops and says there is no solution, then there really is none, because for that component of the graph, everything is determined by the initial choice of "good" for the source vertex, and labeling it "bad" instead would lead to exactly the same situation in which the algorithm cannot continue (just with every vertex in the component having the opposite labeling). And choosing a different source vertex in that component wouldnt help since the distances found between any pair of vertices are shortest path distances and are thus independent of the choice of source vertex. Second, if all of the modified BFSs complete, then by the alternation of assignment of good and bad, and the check performed as each adjacent gray or black node is considered, all edges (rivalries) have been considered and there is no rivalry between two good or two bad wrestlers.
>
> Although this algorithm has an extra loop over all vertices in order to process all connected components, the total time is still O(V + E), since there is no overlap between the executions for each connected component. Thus the time is $O(n + r)$, as specified."[1]

3 Compile and Run Instructions

3.1 Compile Instructions

I have included 4 files in this submission. They are “wrestlemania.w”, “makefile”, “input.txt” and “wrestlemania.dat” (the one with a valid solution).

I have included a Makefile that generates a pdf file, 2 cpp files and 2 executable files from this w file. The Makefile uses GNU C++ version 3.2 on CS Solaris Machines to compiles the 2 cpp files.

The included Makefile also runs the executable files. The first executable “wrestlemania.exe” requires an input file of rivalries called wrestlemania.dat. The program directly opens the file. The second executable “find_two_coloring_random_test.exe” requires some input. In particular it requires 2 integers representing the number of verticies and the number of edges in a hypothetical graph. These can be entered by the user through standard in. However, the input.txt file includes a sample parameter, and the Makefile redirects this file to standard in when it runs the test program.

To run the makefile, type “make” while you are within the directory containing all of the files described above.

3.2 Run Instructions

wrestlemania.exe

This program can be run by typing “./wrestlemania.exe”. The executable “wrestlemania.exe” requires an input file of rivalries called wrestlemania.dat. The program directly opens the file. Therefore, this file should reside in the same directory as the executable.

find_two_coloring_random_test.exe

This program can be run by typing “./find_two_coloring_random_test.exe”. The executable “find_two_coloring_random_test.exe” requires some input. In particular it requires 2 integers representing the number of verticies and the number of edges in a hypothetical graph. These can be entered by the user through standard in. However, the input.txt file includes a sample parameter. This can be redirected by typing “./find_two_coloring_random_test.exe ¡ input.txt”.

Both of the above programs can also be run using the makefile by typing “make run”.

4 Wrestlemania Rivalry Graph Program

The wrestle mania rivalry graph program which solves the above stated problem is presented in this section. The program takes a file of Wrestler Rivalries as input. From this data, it constructs a graph of rivalries. It then performs an algorithm on the graph which finds a 2-coloring of the rivalry graph if one is possible. The structure of the program is as follows.

"wrestlemania.cpp" 6 ≡

```
⟨Include Files 7a⟩
⟨Global Declarations 7b⟩
⟨Test Two Labeling Function 12⟩
⟨Label Map Tester Visitor 13a⟩
⟨Label Map Tester Visitor Object Maker 13b⟩
⟨Find Two Labeling Function 8⟩
⟨Find Two Labeling Visitor 10⟩
⟨Find Two Labeling Visitor Object Maker 11⟩
⟨Wrestle Mania Rivalry Main Program 14a⟩
```

The program uses the BGL Breadth First Visit to solve this problem. Therefore, a visitor object must be constructed to control the behavior of this algorithm at certain points. This is defined in the Find Two Labeling Visitor section. The actual Visitor object is instantiated in the Find Two Labeling Visitor Object Maker. The entire algorithm that finds the two labeling (using the BGL Breadth First Visit Algorithm) is defined in the Find Two Labeling Funtion section. The Find Two Labeling Funtion is called from the Wrestle Mania Rivalry Main Program which in turn finds a 2-coloring or the input rivalry graph. This Main Program contains various subsections as explained below.

Once a 2-coloring of the input rivalry graph is determined, the Main Program then tests the solution to see if it is indeed correct. The program also uses the BGL Breadth First Visit to solve this problem. Therefore, a visitor object must be constructed to control the behavior of this algorithm at certain points. This is defined in the Label Map Tester Visitor section. The actual Visitor object is instantiated in the Label Map Tester Visitor Object Maker section. The entire algorithm that Tests the Label Map Visitor (using the BGL Breadth First Visit Algorithm) is defined in the Test Two Labeling Function.

The program structure also includes Global Declarations and a list of "include files."

4.1 Include Files

The code to include the "include files" is shown below. The majority of these files enable the BGL functionality. Others enable input and output as well as convenient string use. The map include file is used to create a unique pair associative container to store the labels for the rivalry 2-coloring.

⟨Include Files 7a⟩ ≡

```
#include <boost/config.hpp>
#include <iostream>
#include <fstream>
#include <string>
#include <boost/tokenizer.hpp>
#include <boost/tuple/tuple.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/visitors.hpp>
#include <boost/graph/breadth_first_search.hpp>
#include <map>
#include <boost/random/uniform_01.hpp>
```

Used in parts 6, 17.

The following section declares for convenience the predominant namespace used in this program. It also defines an enumeration to be used in the 2-coloring map.

4.2 Global Declarations

The following section declares for convenience the predominant namespace used in this program. It also defines an enumeration to be used in the 2-coloring map.

⟨Global Declarations 7b⟩ ≡

```
using namespace boost;
enum ness { GOOD, BAD };
```

Used in parts 6, 17.

4.3 Find Two Labeling Function

This function performs computes a 2-labeling for a given graph. It first gets the color map of the passed in graph then initializes the color of each vertex to white (which is the color of undescovered verticies). It then runs breadth first visit on each vertex.

The algorithm uses a LabelMap (lm) object to store the "ness" of each Vertex. Ness refers to the goodness or badness orientation of the node. Ness is essentially boolean, though defined as and enumeration. lm must be a pointer to a container that supports the accessor operation.

This function also uses a try and catch scenario so that if the function fails, this will catch it and take appropriate action. The function will fail if a valid coloring is not found.

It then returns an empty label map.

⟨Find Two Labeling Function 8⟩ ≡

```
template <typename Graph, typename LabelMap>
void find_two_labeling(Graph& g, LabelMap& lm) {

        typedef typename property_map<Graph, vertex_color_t>::type color_map_t;
        color_map_t color_map = get(vertex_color, g);

        typedef typename property_traits<color_map_t>::value_type ColorValue;
        typedef color_traits<ColorValue> Color;

    typedef typename graph_traits < Graph >::vertex_iterator vi_t;
        vi_t vi, end2;

        //init each color
        for (tie(vi, end2) = vertices(g); vi != end2; ++vi) {
                put(color_map, *vi, Color::white());

        }

        typedef typename graph_traits < Graph >::vertex_descriptor Vertex;

        try     {
                for (tie(vi, end2) = vertices(g); vi != end2; ++vi) {

                        if(  get(color_map, *vi) == Color::white()  ) {
                                Vertex src = *vi;
                                lm[src] = GOOD;

                                breadth_first_visit(g, src,
                                        visitor(record_lm(&lm)));
                        }
                }
        }       catch (int i)
        {
              std::cerr <<"Can't be done!" << std::endl;
        }
}
```

Used in parts 6, 17.

4.4 Find Two Labeling Visitor

The following is a visiter object that has a tree_edge, gray_target, and black_target visitor function. These functions are invoked at specified times during the breadth first visit algorithm.

The tree_edge visitor function is invoked if the edge being examined is a member of the search tree.[2] This function sets the Ness of the vertex to the opposite of the Ness of its respective source vertex.

The gray_target visitor function is invoked if the edge being examined is a cycle edge and the target vertex is colored gray at the time of examination.[2] Basically, this function tests to see if the source and target (itself) vertex Ness are the same. If they are, then the function aborts because then the coloring is not valid and because of the simple nature of 2-coloring we know that no valid 2-coloring exists.

The black_target visitor function is invoked if the edge being examined is a cycle edge, and if the target vertex is colored black at the time of examination.[2] This function also tests to see if the source and target (itself) vertex Ness are the same. Likewise, if they are, then the function aborts because, as stated previously, the coloring is not valid.

⟨Find Two Labeling Visitor 10⟩ ≡

```
template < typename     TwoColoringMap >
class lm_recorder:public default_bfs_visitor {
public:
        lm_recorder(TwoColoringMap tcm):d(tcm) {}

        template < typename     Edge, typename Graph >
                void tree_edge(Edge     e, const Graph & g)     const {

                typedef typename property_map<Graph, vertex_color_t>::type color_map_t;
                typename graph_traits < Graph >::vertex_descriptor
                        u =     source(e, g), v = target(e,     g);
                        if ( (*d)[u]== GOOD     ) {
                                (*d)[v] = BAD;
                        } else {
                                (*d)[v] = GOOD;
                        }
                }

        template < typename     Edge, typename Graph >
        void gray_target(Edge   e, const Graph & g)     const {

                typedef typename property_map<Graph, vertex_color_t>::type color_map_t;
                typename graph_traits < Graph >::vertex_descriptor
                        u =     source(e, g), v = target(e,     g);


                if(     (*d)[v] == (*d)[u] ) {
                        (*d).clear();
                        throw (1);
                }
        }

        template < typename     Edge, typename Graph >
        void black_target(Edge  e, const Graph & g)     const {

                typedef typename property_map<Graph, vertex_color_t>::type color_map_t;
                typename graph_traits < Graph >::vertex_descriptor
                        u =     source(e, g), v = target(e,     g);

                if(     (*d)[v] == (*d)[u] ) {
                        (*d).clear();
                        throw (1);
                }
        }

private:
        TwoColoringMap d;
};
```

Used in parts 6, 17.

4.5 Find Two Labeling Visitor Object Maker

The purpose of this function is to instantiate and return an instance of an lm_recorder object.

⟨Find Two Labeling Visitor Object Maker 11⟩ ≡

```
template < typename TwoColoringMap >
lm_recorder < TwoColoringMap > record_lm(TwoColoringMap d)
{
        return lm_recorder < TwoColoringMap > (d);
}
```

Used in parts 6, 17.

4.6 Test Two Labeling Function

This function performs a test on the label map to determine if it is a correct labeling. This algorithm runs breadth first visit on each vertex in the graph. Since this algorithm uses the examine_edge visitor function, it will inspect each out edge regardless of the color. Therefore, it is not necessary it initialize the vertex internal colors (those used by breadth first visit) to white.

The algorithm uses a LabelMap (lm) object to store and retrieve the "Ness" of each Vertex. This particular function only tests the Ness, therefore, it doesn't change the Ness. Ness refers to the goodness or badness orientation of the vertex. Ness is essentially boolean, though defined as and enumeration. lm must be a pointer to a container that supports the accessor operation.

This uses a try and catch scenario so that if the function fails, this will catch it and take appropriate action. The function called will fail if the coloring it is testing is invalid.

⟨Test Two Labeling Function 12⟩ ≡

```
template <typename Graph, typename LabelMap>
void test_two_labeling(Graph& g, LabelMap& lm) {

        typedef typename property_map<Graph, vertex_color_t>::type color_map_t;
        color_map_t color_map = get(vertex_color, g);

    typedef typename graph_traits < Graph >::vertex_iterator vi_t;
        vi_t vi, end2;

        typedef typename graph_traits < Graph >::vertex_descriptor Vertex;

                try
                {
                        for (tie(vi, end2) = vertices(g); vi != end2; ++vi) {
                                Vertex src = *vi;
                                breadth_first_visit(g, src,     visitor(test_lm(&lm)));
                        }
                        std::cout << std::endl << "Result passes the acceptance test." << std::e
                }
                catch ( int str )
                {
                        std::cerr << "Failed acceptance test!" << std::endl;
                }
}
```

Used in parts 6, 17.

4.7 Label Map Tester Visitor

The following is a visiter object that has an examine_edge visitor function. This function is invoked on every out edge of each vertex after it is discovered. The function tests to see if the Ness of the source vertex is the same as the Ness of the target vector. If it is, then the coloring is not valid.

⟨Label Map Tester Visitor 13a⟩ ≡

```
template < typename TwoColoringMap >
class lm_tester:public default_bfs_visitor {
public:
        lm_tester(TwoColoringMap tcm):d(tcm) {}

        template < typename Edge, typename Graph >
                void examine_edge(Edge e, const Graph & g) const {

                typedef typename property_map<Graph, vertex_color_t>::type color_map_t;
                typename graph_traits < Graph >::vertex_descriptor
                        u = source(e, g), v = target(e, g);

                typedef typename property_traits<color_map_t>::value_type ColorValue;
                typedef color_traits<ColorValue> Color;

                if( (*d)[v] == (*d)[u] ) {
                        throw (1);
                        } else {}
                }

private:
        TwoColoringMap d;

};
```
Used in parts 6, 17.

4.8 Label Map Tester Visitor Object Maker

The purpose of this function is to instantiate and return an instance of an lm_tester object.

⟨Label Map Tester Visitor Object Maker 13b⟩ ≡

```
template < typename TwoColoringMap >
lm_tester < TwoColoringMap > test_lm(TwoColoringMap d)
{
        return lm_tester < TwoColoringMap > (d);
}
```
Used in parts 6, 17.

4.9 Wrestle Mania Rivalry Main Program

Finally, we get to the program that uses all of the above objects and functions. This main section of the Wrestle Mania Rivalry Program has 5 important functions. First, it declares a bunch of typedefs for the composite types we will need to use. Then it opens our data file "wrestlemanie.dat" and loads the rivaries into a graph. It also outputs the rivalries to the screen in the process. It than runs the find two labeling function on the graph (and the associated label map that was also created in the declarations section). It then outputs the results if there is a valid coloring found. Once that is done, it runs a test on the coloring to make sure that it is a valid coloring.

⟨Wrestle Mania Rivalry Main Program 14a⟩ ≡

```
int main()
{
        ⟨Typedefs and Declarations 14b⟩
        ⟨Get Data File 15⟩
        ⟨Find Two Labeling 16a⟩
        ⟨Output Good and Bad Guys 16b⟩
        ⟨Test The Solution 16c⟩

        return 0;
}
```

Used in part 6.

4.10 Typedefs and Declarations

This section defines types for the composite types we will be using. It also uses declares some objects.

⟨Typedefs and Declarations 14b⟩ ≡

```
        // Typedefs
        typedef adjacency_list<vecS, vecS, undirectedS, property<vertex_name_t,
                std::string, property<vertex_color_t, default_color_type> > > Graph;
        typedef graph_traits < Graph >::vertex_descriptor Vertex;
        typedef std::map < Vertex, ness > lm_t;

        Graph g;
        lm_t lm;
        bool good_flag1 = 1, good_flag2 = 1;
```

Used in parts 14a, 18a.

4.11 Get Data File

This section of the program opens the data file, and inputs the each rivalry into a graph. A rivalry consists of the names of two wrestlers. Each wrestler will be inserted into the graph as a vertex. If a given wrestler is already in the graph, then it will not be re-inserted. The two verticies will be connected by an edge to depict a rivalry. These rivalrys will be output to standard out.

⟨Get Data File 15⟩ ≡

```
std::ifstream datafile("./wrestlemania.dat");
if (!datafile) {
        std::cerr << "No ./wrestlemania.dat file" << std::endl;
        return EXIT_FAILURE;
}

typedef property_map < Graph, vertex_name_t >::type wrestler_name_map_t;
wrestler_name_map_t wrestler_name = get(vertex_name, g);

typedef std::map < std::string, Vertex > NameVertexMap;
NameVertexMap wrestlers;

for (std::string line; std::getline(datafile, line);) {
        char_separator<char> sep("|", "");
        tokenizer<char_separator<char> > line_toks(line, sep);
        tokenizer<char_separator<char> >::iterator i = line_toks.begin();
        std::string wrestlers_name = *i++;
        std::cout << wrestlers_name << std::endl;
        NameVertexMap::iterator pos;
        bool inserted;
        Vertex u, v;
        tie(pos, inserted) = wrestlers.insert(std::make_pair(wrestlers_name, Vertex()));
        if (inserted) {
                u = add_vertex(g);
                wrestler_name[u] = wrestlers_name;
                pos->second = u;
        } else
                u = pos->second;

        tie(pos, inserted) = wrestlers.insert(std::make_pair(*i, Vertex()));
        if (inserted) {
                v = add_vertex(g);
                wrestler_name[v] = *i;
                pos->second = v;
        } else
                v = pos->second;
        std::cout << "   has a rivalry with\n      " << *i << std::endl << std::endl;

        graph_traits < Graph >::edge_descriptor e;
        tie(e, inserted) = add_edge(u, v, g);
}
```

Used in part 14a.

4.12 Find Two Labeling

This section of the program calls the find_two_labeling function.

⟨Find Two Labeling 16a⟩ ≡

```
                              find_two_labeling(g, lm);
```

Used in parts 14a, 18a.

4.13 Output Good and Bad Guys

This section outputs the results of the coloring found. The results are only output if a valid coloring is found.

⟨Output Good and Bad Guys 16b⟩ ≡

```
        if (lm.begin() != lm.end()) {
                graph_traits < Graph >::vertex_iterator i, end;
                std::cout<<"\n\nGood guys:\n";
                for (tie(i, end) = vertices(g); i != end; ++i) {
                        if(lm[*i] == 0)
                                std::cout << "    " << wrestler_name[*i] << std::endl;
                }
                std::cout<<"\nBad guys:\n";
                for (tie(i, end) = vertices(g); i != end; ++i) {
                        if(lm[*i] == 1)
                                std::cout << "    " <<wrestler_name[*i] << std::endl;
                }
        }
```

Used in part 14a.

4.14 Test The Solution

This section calls the test on the coloring that was found.

⟨Test The Solution 16c⟩ ≡

```
        if (lm.begin() != lm.end()) {
                        test_two_labeling(g, lm);
        } else {
                std::cout << "No valid solution exists to test." << std::endl;
        }
        std::cout<< std::endl<< std::endl<<"**  Program Finished  **"<< std::endl;
```

Used in parts 14a, 18a.

5 Find Two Coloring Random Test Program

The Find Two Coloring Random Test Program creates a random graph to the user's dimensions, finds a two coloring of it, if possible, then tests that two coloring to see if it is valid.

The structure of the program is as follows:

"find_two_coloring_random_test.cpp" 17 ≡

```
⟨Include Files 7a⟩
⟨Global Declarations 7b⟩
⟨Test Two Labeling Function 12⟩
⟨Label Map Tester Visitor 13a⟩
⟨Label Map Tester Visitor Object Maker 13b⟩
⟨Find Two Labeling Function 8⟩
⟨Find Two Labeling Visitor 10⟩
⟨Find Two Labeling Visitor Object Maker 11⟩
⟨Two Coloring Test Main Program 18a⟩
```

The program uses the BGL Breadth First Visit to solve this problem. Therefore, a visitor object must be constructed to control the behavior of this algorithm at certain points. This is defined in the Find Two Labeling Visitor section (above). The actual Visitor object is instantiated in the Find Two Labeling Visitor Object Maker (above). The entire algorithm that finds the two labeling (using the BGL Breadth First Visit Algorithm) is defined in the Find Two Labeling Function section. The Find Two Labeling Funtion is called from the Two Coloring Test Main Program section. This Main Program contains various subsections as explained below.

Once a 2-coloring of the random graph is determined, the Main Program then tests the solution to see if it is indeed correct. The program also uses the BGL Breadth First Visit to solve this problem. Therefore, a visitor object must be constructed to control the behavior of this algorithm at certain points. This is defined in the Label Map Tester Visitor section. The actual Visitor object is instantiated in the Label Map Tester Visitor Object Maker section. The entire algorithm that Tests the Label Map Visitor (using the BGL Breadth First Visit Algorithm) is defined in the Test Two Labeling Function.

5.1 Two Coloring Test Main Program

Most of the functions and objects of this test program were defined as part of the Rivalries application. However, the “main” section of the program is different.

This section performs several functions that are described in the various section.

This main section has 5 important functions. First, it declares a bunch of typedefs for the composite types we will need to use. Then it gets user input to define the graph dimensions. It then runs the find two labeling function on the graph (and the associated label map that was also created in the declarations section). Once that is done, it runs a test on the coloring to make sure that it is a valid coloring.

⟨Two Coloring Test Main Program 18a⟩ ≡

```
int main()
{
        ⟨Typedefs and Declarations 14b⟩

        ⟨Get User Input (Graph Parameters) 18b⟩
        ⟨Construct a Random Graph 19⟩
        ⟨Find Two Labeling 16a⟩
        ⟨Test The Solution 16c⟩

        return 0;
}
```

Used in part 17.

5.2 Get User Input (Graph Parameters)

This section gets input from the user to create a graph. The uses is asked to define the number of verticies in the graph along with the number of edges. The later section will create a graph to those dimensions.

⟨Get User Input (Graph Parameters) 18b⟩ ≡

```
        int N,E;
        std::cout<< "This program makes a random graph, and creates a "
        <<"two-coloring of it.\nPlease enter the number of verticies "
        <<"you would like in the graph: ";
        std::cin >> N;
        std::cout<< "\nPlease enter the number of Edges you would like "
        <<"in the graph \n(remember, the more edges the less likely the "
        <<"graph is to be 2-colorable): ";
        std::cin >> E;
```

Used in part 18a.

5.3 Construct a Random Graph

This section of code creates a random graph. It uses the random graph generator supplied by the boost libraries along with the random number generator to seed it.

⟨Construct a Random Graph 19⟩ ≡

```
const unsigned int randseed = 29u;

typedef mt19937 RandGen;
RandGen gen1;
gen1.seed(31u);

std::cout << "\n\nGenerating a random graph with " << N << " vertices"
        << " and " << E << " edges." << std::endl;
generate_random_graph(g, N, E, gen1);
```

Used in part 18a.

6 Summary

The above program uses three specific concept models in its implementation. They are the models of the VertexListGraph concept, the IncidenceGraph concept and the UniquePairAssociativeContainer concept.

The VertexListGraph concept is a refinement of the Graph concept. Basically, it requires the ability to traversal of all the vertices in the graph efficiently.

The IncidenceGraph concept provides an interface for efficient access to the out edges of each vertex in the graph.

Both of these qualities are important for the above program because the breadth first search algorithm uses the out edge iterators and the calling functions use the efficient traversal quality.

The UniquePairAssociativeContainer concept is a combination of the Associative Container, the Unique Associative Container, and the Pair Associative Container concepts.

An Associative Container is a variable-sized Container that supports efficient retrieval of elements (values) based on keys. The Unique Associative Container is an Associative Container with the property that each key in the container is unique (i.e. no two elements in a Unique Associative Container have the same key). A Pair Associative Container is an Associative Container that associates a key with some other object.[4]

In sum, a map is the appropriate model to use. It is associative (key based), unique (not a multi-map) and pair based (the internal structure is a pair).

The efficiency of this algorithm is $O(V + E)$. In other words it is linear in relationship to the number of verticies and edges. The function that loops through and calls the find coloring function would give you the impression that there is some exponential computation going on. However, the breadth first visit algorithm calls the breadth first search on the graph from different vertices, without resetting the color of the verticies. Therefore, the algorithm stops short, on successive loops, of duplicating work by not overlapping where the previous loop already went. It does this by noting the color of the verticies.

The above program performs in accordance with the specifications. It returns the appropriate result on the given sample graphs. The test function returns the same result as the initial find two coloring function as expected. There are no apparent problems.

References

[1] D. R. Musser, "Homework 4," *Generic Software Design Course Website*, http://www.cs.rpi.edu/~musser/gsd/quartiles/quartiles.pdf (October 2002). (document), 1, 2

[2] Jeremy Siek, Lie-Quan Lee, Andrew Lumsdaine, "Boost Graph Library, The: User Guide and Reference Manual" *Addison-Wesley*, http://www.boost.org/libs/graph/doc/index.html (October 2000 - 2001). (document), 1, 4.4

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms, Second Edition," *MIT Press*, p. 539 (2001). 1

[4] Silicon Graphics, Inc., "Standard Template Library Programmer's Guide," *Silicon Graphics, Inc. Website*, http://www.sgi.com/tech/stl/nth_element.html (1993-2002). 6