An Implementation of Graph Isomorphism Testing

Jeremy G. Siek

December 9, 2001

0.1 Introduction

This paper documents the implementation of the ***isomorphism()*** function of the Boost Graph Library. The implementation was by Jeremy Siek with algorithmic improvements and test code from Douglas Gregor and Brian Osman. The ***isomorphism()*** function answers the question, "are these two graphs equal?" By *equal* we mean the two graphs have the same structure—the vertices and edges are connected in the same way. The mathematical name for this kind of equality is *isomorphism*.

More precisely, an *isomorphism* is a one-to-one mapping of the vertices in one graph to the vertices of another graph such that adjacency is preserved. Another words, given graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, an isomorphism is a function f such that for all pairs of vertices a, b in V_1, edge (a, b) is in E_1 if and only if edge $(f(a), f(b))$ is in E_2.

The graph G_1 is *isomorphic* to G_2 if an isomorphism exists between the two graphs, which we denote by $G_1 \cong G_2$. Both graphs must be the same size, so let $N = |V_1| = |V_2|$.

In the following discussion we will need to use several more notions from graph theory. The graph $G_s = (V_s, E_s)$ is a *subgraph* of graph $G = (V, E)$ if $V_s \subseteq V$ and $E_s \subseteq E$. An *induced subgraph*, denoted by $G[V_s]$, of a graph $G = (V, E)$ consists of the vertices in V_s, which is a subset of V, and every edge (u, v) in E such that both u and v are in V_s. We use the notation $E[V_s]$ to mean the edges in $G[V_s]$.

0.2 Backtracking Search

The algorithm used by the ***isomorphism()*** function is, at first approximation, an exhaustive search implemented via backtracking. The backtracking algorithm is a recursive function. At each stage we will try to extend the match that we have found so far. So suppose that we have already determined that some subgraph of G_1 is isomorphic to a subgraph of G_2. We then try to add a vertex to each subgraph such that the new subgraphs are still isomorphic to one another. At some point we may hit a dead end—there are no vertices that can be added to extend the isomorphic subgraphs. We then backtrack to previous smaller matching subgraphs, and try extending with a different vertex choice. The process ends by either finding a complete mapping between G_1 and G_2 and returning true, or by exhausting all possibilities and returning false.

The problem with the exhaustive backtracking algorithm is that there are $N!$ possible vertex mappings, and $N!$ gets very large as N increases, so we need to prune the search space. We use the pruning techniques described in [1, 2, 3], some of which originated in [4, 5]. Also, the specific backtracking method we use is the one from [1].

We consider the vertices of G_1 for addition to the matched subgraph in a specific order, so assume that the vertices of G_1 are labeled $1, \ldots, N$ according to that order.

As we will see later, a good ordering of the vertices is by DFS discover time. Let $G_1[k]$ denote the subgraph of G_1 induced by the first k vertices, with $G_1[0]$ being an empty graph. We also consider the edges of G_1 in a specific order. We always examine edges in the current subgraph $G_1[k]$ first, that is, edges (u, v) where both $u \leq k$ and $v \leq k$. This ordering of edges can be acheived by sorting each edge (u, v) by lexicographical comparison on the tuple $\langle \max(u, v), u, v \rangle$. Figure 1 shows an example of a graph with the vertices labelled by DFS discover time. The edge ordering for this graph is as follows:

source:	0	1	0	1	3	0	5	6	6
target:	1	2	3	3	2	4	6	4	7

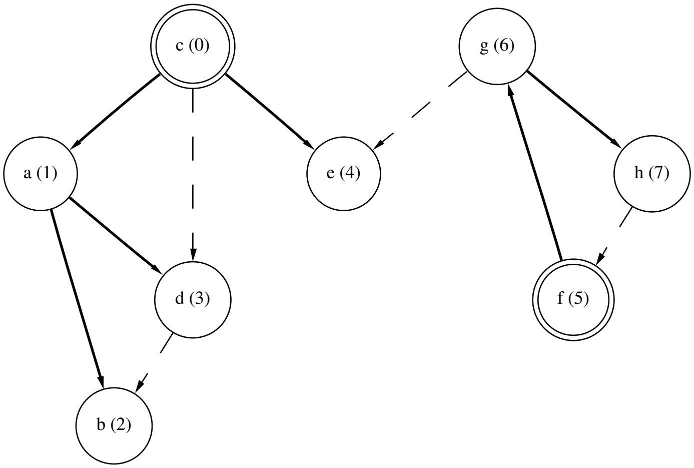

Figure 1: Vertices numbered by DFS discover time. The DFS tree edges are the solid lines. Nodes 0 and 5 are DFS tree root nodes.

Each step of the backtracking search moves from left to right though the ordered edges. At each step it examines an edge (u, v) of G_1 and decides whether to continue to the left or to go back. There are three cases to consider:

1. $i > k$
2. $i \leq k$ and $j > k$.
3. $i \leq k$ and $j \leq k$.

Case 1: $i > k$**.** i is not in the matched subgraph $G_1[k]$. This situation only happens at the very beginning of the search, or when i is not reachable from any of the vertices in $G_1[k]$. This means that we are finished with $G_1[k]$. We increment k and find match for it amongst any of the eligible vertices in $V_2 - S$. We then proceed to Case 2. It is usually the case that i is equal to the new k, but when there is another DFS root r with no in-edges or out-edges and if $r < i$ then it will be the new k.

Case 2: $i \leq k$ and $j > k$. i is in the matched subgraph $G_1[k]$, but j is not. We are about to increment k to try and grow the matched subgraph to include j. However, first we need to finish verifying that $G_1[k] \cong G_2[S]$. In previous steps we proved that $G_1[k-1] \cong G_2[S - \{f(k)\}]$, so now we just need to verify the extension of the isomorphism to k. At this point we are guaranteed to have seen all the edges to and from vertex k (because the edges are sorted), and in previous steps we have checked that for each edge incident on k in $E_1[k]$ there is a matching edge in $E_2[S]$. However we still need to check the "only if" part of the "if and only if". So we check that for every edge (u, v) incident on $f(k)$ there is $(f^{-1}(u), f^{-1}(v)) \in E_1[k]$. A quick way to verify this is to make sure that the number of edges incident on k in $E_1[k]$ is the same as the number of edges incident on $f(k)$ in $E_2[S]$. We create an edge counter that we increment every time we see an edge incident on k and decrement for each edge incident on $f(k)$. If the counter gets back to zero we know the edges match up.

Once we have verified that $G_1[k] \cong G_2[S]$ we add $f(k)$ to S, increment k, and then try assigning j to any of the eligible vertices in $V_2 - S$. More about what "eligible" means below.

Case 3: $i \leq k$ and $j \leq k$. Both i and j are in $G_1[k]$. We check to make sure that $(f(i), f(j)) \in E_2[S]$ and then proceed to the next edge.

0.2.1 Vertex Invariants

One way to reduce the search space is through the use of *vertex invariants*. The idea is to compute a number for each vertex $i(v)$ such that $i(v) = i(v')$ if there exists some isomorphism f where $f(v) = v'$. Then when we look for a match to some vertex v, only those vertices that have the same vertex invariant number are "eligible". The number of vertices in a graph with the same vertex invariant number i is called the *invariant multiplicity* for i. In this implementation, by default we use the function $i(v) = (|V| + 1) \times \text{out-degree}(v) + \text{in-degree}(v)$, though the user can also supply there own invariant function. The ability of the invariant function to prune the search space varies widely with the type of graph.

The following is the definition of the functor that implements the default vertex invariant. The functor models the AdaptableUnaryFunction concept.

⟨ Degree vertex invariant functor 4 ⟩ ≡

```
template <typename InDegreeMap, typename Graph>
class degree_vertex_invariant
{
    typedef typename graph_traits<Graph>::vertex_descriptor vertex_t;
    typedef typename graph_traits<Graph>::degree_size_type size_type;
  public:
    typedef vertex_t argument_type;
    typedef size_type result_type;

    degree_vertex_invariant(const InDegreeMap& in_degree_map, const Graph& g)
      : m_in_degree_map(in_degree_map), m_g(g) { }

    size_type operator()(vertex_t v) const {
      return (num_vertices(m_g) + 1) * out_degree(v, m_g)
        + get(m_in_degree_map, v);
    }
    // The largest possible vertex invariant number
    size_type max() const {
      return num_vertices(m_g) * num_vertices(m_g) + num_vertices(m_g);
    }
  private:
    InDegreeMap m_in_degree_map;
    const Graph& m_g;
  };
```

0.2.2 Vertex Order

A good choice of the labeling for the vertices (which determines the order in which the subgraph $G_1[k]$ is grown) can also reduce the search space. In the following we discuss two labeling heuristics.

Most Constrained First

Consider the most constrained vertices first. That is, examine lower-degree vertices before higher-degree vertices. This reduces the search space because it chops off a trunk before the trunk has a chance to blossom out. We can generalize this to use vertex invariants. We examine vertices with low invariant multiplicity before examining vertices with high invariant multiplicity.

Adjacent First

It only makes sense to examine an edge if one or more of its vertices has been assigned a mapping. This means that we should visit vertices adjacent to those in the current matched subgraph before proceeding.

DFS Order, Starting with Lowest Multiplicity

For this implementation, we combine the above two heuristics in the following way. To implement the “adjacent first” heuristic we apply DFS to the graph, and use the DFS discovery order as our vertex order. To comply with the “most constrained first” heuristic we order the roots of our DFS trees by invariant multiplicity.

0.2.3 Implementation of the *match* function

The ***match*** function implements the recursive backtracking, handling the four cases described in §0.2.

⟨ Match function 6a ⟩ ≡

```
bool match(edge_iter iter, int dfs_num_k)
{
    if (iter != ordered_edges.end()) {
        vertex1_t i = source(*iter, G1), j = target(*iter, G2);
        if (dfs_num[i] > dfs_num_k) {
           ⟨Find a match for the DFS tree root k + 1 6b⟩
        }
        else if (dfs_num[j] > dfs_num_k) {
            ⟨Verify G 1[k] ≅ G 2[S] and then find match for j 7a⟩
        }
        else {
            ⟨Check to see if (f(i), f(j)) ∈ E 2[S] and continue 8b⟩
        }
    } else
        return true;
    return false;
}
```

Now to describe how each of the four cases is implemented.

Case 1: $i \notin G_1[k]$**.** We increment k and try to map it to any of the eligible vertices of $V_2 - S$. After matching the new k we proceed by invoking ***match***. We do not yet move on to the next edge, since we have not yet found a match for edge, or for target j. We reset the edge counter to zero.

⟨ Find a match for the DFS tree root $k + 1$ 6b ⟩ ≡

```
vertex1_t kp1 = dfs_vertices[dfs_num_k + 1];
BGL_FORALL_VERTICES_T(u, G2, Graph2) {
    if (invariant1(kp1) == invariant2(u) && in_S[u] == false) {
        f[kp1] = u;
        in_S[u] = true;
        num_edges_on_k = 0;
```

```
            if (match(iter, dfs_num_k + 1));
                return true;
            in_S[u] = false;
        }
    }
```

Case 2: $i \in G_1[k]$ and $j \notin G_1[k]$. Before we extend the subgraph by incrementing k, we need to finish verifying that $G_1[k]$ and $G_2[S]$ are isomorphic. We decrement the edge counter for every edge incident to $f(k)$ in $G_2[S]$, which should bring the counter back down to zero. If not we return false.

⟨ Verify $G_1[k] \cong G_2[S]$ and then find match for j 7a ⟩ ≡

```
    vertex1_t k = dfs_vertices[dfs_num_k];
    ⟨Count out-edges of f(k) in G 2[S] 7b⟩
    ⟨Count in-edges of f(k) in G 2[S] 7c⟩
    if (num_edges_on_k != 0)
        return false;
    ⟨Find a match for j and continue 8a⟩
```

We decrement the edge counter for every vertex in $Adj[f(k)]$ that is also in S. We call ***count_if*** to do the counting, using ***boost::bind*** to create the predicate functor.

⟨ Count out-edges of $f(k)$ in $G_2[S]$ 7b ⟩ ≡

```
    num_edges_on_k −=
        count_if(adjacent_vertices(f[k], G2), make_indirect_pmap(in_S));
```

Next we iterate through all the vertices in S and for each we decrement the counter for each edge whose target is k.

⟨ Count in-edges of $f(k)$ in $G_2[S]$ 7c ⟩ ≡

```
    for (int jj = 0; jj < dfs_num_k; ++jj) {
        vertex1_t j = dfs_vertices[jj];
        num_edges_on_k −= count(adjacent_vertices(f[j], G2), f[k]);
    }
```

Now that we have finished verifying that $G_1[k] \cong G_2[S]$, we can now consider extending the isomorphism. We need to find a match for j in $V_2 - S$. Since j is adjacent to i, we can further narrow down the search by only considering vertices adjacent to $f(i)$. Also, the vertex must have the same vertex invariant. Once we have a matching vertex v we extend the matching subgraphs by incrementing k and adding v to S, we set $f(j) = v$, and we set the edge counter to 1 (since (i, j) is the first edge incident on our new k). We continue to the next edge by calling ***match***. If that fails we undo the assignment $f(j) = v$.

⟨ Find a match for j and continue 8a ⟩ ≡

```
BGL_FORALL_ADJ_T(f[i], v, G2, Graph2)
  if (invariant2(v) == invariant1(j) && in_S[v] == false) {
    f[j] = v;
    in_S[v] = true;
    num_edges_on_k = 1;
  int next_k = std::max(dfs_num_k, std::max(dfs_num[i], dfs_num[j]));
    if (match(next(iter), next_k))
      return true;
    in_S[v] = false;
  }
```

Case 3: both i and j are in $G_1[k]$. Our goal is to check whether $(f(i), f(j)) \in E_2[S]$. We examine the vertices $Adj[f(i)]$ to see if any of them is equal to $f(j)$. If so, then we have a match for the edge (i, j), and can increment the counter for the number of edges incident on k in $E_1[k]$. We continue by calling ***match*** on the next edge.

⟨ Check to see if $(f(i), f(j)) \in E_2[S]$ and continue 8b ⟩ ≡

```
if (any_equal(adjacent_vertices(f[i], G2), f[j])) {
  ++num_edges_on_k;
  if (match(next(iter), dfs_num_k))
      return true;
}
```

0.3 Public Interface

The following is the public interface for the ***isomorphism*** function. The input to the function is the two graphs G_1 and G_2, mappings from the vertices in the graphs to integers (in the range $[0, |V|)$), and a vertex invariant function object. The output of the function is an isomorphism f if there is one. The ***isomorphism*** function returns true if the graphs are isomorphic and false otherwise. The invariant parameters are function objects that compute the vertex invariants for vertices of the two graphs. The ***max_invariant*** parameter is to specify one past the largest integer that a vertex invariant number could be (the invariants numbers are assumed to span from zero to ***max_invariant-1***). The requirements on the template parameters are described below in the "Concept checking" code part.

⟨ Isomorphism function interface 8c ⟩ ≡

```
template <typename Graph1, typename Graph2, typename IsoMapping,
          typename Invariant1, typename Invariant2,
          typename IndexMap1, typename IndexMap2>
bool isomorphism(const Graph1& G1, const Graph2& G2, IsoMapping f,
                 Invariant1 invariant1, Invariant2 invariant2,
                 std::size_t max_invariant,
                 IndexMap1 index_map1, IndexMap2 index_map2)
```

The function body consists of the concept checks followed by a quick check for empty graphs or graphs of different size and then constructs an algorithm object. We then call the ***test_isomorphism*** member function, which runs the algorithm. The reason that we implement the algorithm using a class is that there are a fair number of internal data structures required, and it is easier to make these data members of a class and make each section of the algorithm a member function. This relieves us from the burden of passing lots of arguments to each function, while at the same time avoiding the evils of global variables (non-reentrant, etc.).

⟨ Isomorphism function body 9a ⟩ ≡

```
{
  ⟨Concept checking 10a⟩
  ⟨Quick return based on size 9b⟩
  detail::isomorphism_algo<Graph1, Graph2, IsoMapping, Invariant1,
    Invariant2, IndexMap1, IndexMap2>
    algo(G1, G2, f, invariant1, invariant2, max_invariant,
         index_map1, index_map2);
  return algo.test_isomorphism();
}
```

If there are no vertices in either graph, then they are trivially isomorphic. If the graphs have different numbers of vertices then they are not isomorphic. We could also check the number of edges here, but that would introduce the EdgeListGraph requirement, which we otherwise do not need.

⟨ Quick return based on size 9b ⟩ ≡

```
if (num_vertices(G1) != num_vertices(G2))
  return false;
if (num_vertices(G1) == 0 && num_vertices(G2) == 0)
  return true;
```

We use the Boost Concept Checking Library to make sure that the template arguments fulfill certain requirements. The graph types must model the VertexListGraph and AdjacencyGraph concepts. The vertex invariants must model the AdaptableUnaryFunction concept, with a vertex as their argument and an integer return type. The ***IsoMapping*** type representing the isomorphism f must be a ReadWritePropertyMap that maps from vertices in G_1 to vertices in G_2. The two other index maps are ReadablePropertyMaps from vertices in G_1 and G_2 to unsigned integers.

⟨ Concept checking 10a ⟩ ≡

```
// Graph requirements
function_requires< VertexListGraphConcept<Graph1> >();
function_requires< EdgeListGraphConcept<Graph1> >();
function_requires< VertexListGraphConcept<Graph2> >();
function_requires< BidirectionalGraphConcept<Graph2> >();

typedef typename graph_traits<Graph1>::vertex_descriptor vertex1_t;
typedef typename graph_traits<Graph2>::vertex_descriptor vertex2_t;
typedef typename graph_traits<Graph1>::vertices_size_type size_type;

// Vertex invariant requirement
function_requires< AdaptableUnaryFunctionConcept<Invariant1,
  size_type, vertex1_t> >();
function_requires< AdaptableUnaryFunctionConcept<Invariant2,
  size_type, vertex2_t> >();

// Property map requirements
function_requires< ReadWritePropertyMapConcept<IsoMapping, vertex1_t> >();
typedef typename property_traits<IsoMapping>::value_type IsoMappingValue;
BOOST_STATIC_ASSERT((is_same<IsoMappingValue, vertex2_t>::value));

function_requires< ReadablePropertyMapConcept<IndexMap1, vertex1_t> >();
typedef typename property_traits<IndexMap1>::value_type IndexMap1Value;
BOOST_STATIC_ASSERT((is_convertible<IndexMap1Value, size_type>::value));

function_requires< ReadablePropertyMapConcept<IndexMap2, vertex2_t> >();
typedef typename property_traits<IndexMap2>::value_type IndexMap2Value;
BOOST_STATIC_ASSERT((is_convertible<IndexMap2Value, size_type>::value));
```

0.4 Data Structure Setup

The following is the outline of the isomorphism algorithm class. The class is templated on all of the same parameters as the *isomorphism* function, and all of the parameter values are stored in the class as data members, in addition to the internal data structures.

⟨ Isomorphism algorithm class 10b ⟩ ≡

```
template <typename Graph1, typename Graph2, typename IsoMapping,
    typename Invariant1, typename Invariant2,
    typename IndexMap1, typename IndexMap2>
class isomorphism_algo
{
    ⟨Typedefs for commonly used types 14c⟩
    ⟨Data members for the parameters 14d⟩
    ⟨Internal data structures 15a⟩
    friend struct compare_multiplicity;
    ⟨Invariant multiplicity comparison functor 12b⟩
    ⟨DFS visitor to record vertex and edge order 13b⟩
    ⟨Edge comparison predicate 14b⟩
  public:
    ⟨Isomorphism algorithm constructor 15b⟩
    ⟨Test isomorphism member function 11a⟩
  private:
    ⟨Match function 6a⟩
  };
```

The interesting parts of this class are the *test_isomorphism* function and the *match* function. We focus on those in in the following sections, and leave the other parts of the class to the Appendix.

The *test_isomorphism* function does all of the setup required of the algorithm. This consists of sorting the vertices according to invariant multiplicity, and then by DFS order. The edges are then sorted as previously described. The last step of this function is to begin the backtracking search.

⟨ Test isomorphism member function 11a ⟩ ≡

```
  bool test_isomorphism()
  {
    ⟨Quick return if the vertex invariants do not match up 11b⟩
    ⟨Sort vertices according to invariant multiplicity 12a⟩
    ⟨Order vertices and edges by DFS 13a⟩
    ⟨Sort edges according to vertex DFS order 14a⟩

    int dfs_num_k = −1;
    return this−>match(ordered_edges.begin(), dfs_num_k);
  }
```

As a first check to rule out graphs that have no possibility of matching, one can create a list of computed vertex invariant numbers for the vertices in each graph, sort the two lists, and then compare them. If the two lists are different then the two graphs are not isomorphic. If the two lists are the same then the two graphs may be isomorphic.

⟨ Quick return if the vertex invariants do not match up 11b ⟩ ≡

```
  {
    std::vector<invar1_value> invar1_array;
    BGL_FORALL_VERTICES_T(v, G1, Graph1)
      invar1_array.push_back(invariant1(v));
    sort(invar1_array);
    std::vector<invar2_value> invar2_array;
    BGL_FORALL_VERTICES_T(v, G2, Graph2)
      invar2_array.push_back(invariant2(v));
    sort(invar2_array);
    if (! equal(invar1_array, invar2_array))
      return false;
  }
```

Next we compute the invariant multiplicity, the number of vertices with the same invariant number. The ***invar_mult*** vector is indexed by invariant number. We loop through all the vertices in the graph to record the multiplicity. We then order the vertices by their invariant multiplicity. This will allow us to search the more constrained vertices first.

⟨ Sort vertices according to invariant multiplicity 12a ⟩ ≡

```
  std::vector<vertex1_t> V_mult;
  BGL_FORALL_VERTICES_T(v, G1, Graph1)
    V_mult.push_back(v);
  {
    std::vector<size_type> multiplicity(max_invariant, 0);
    BGL_FORALL_VERTICES_T(v, G1, Graph1)
      ++multiplicity[invariant1(v)];
    sort(V_mult, compare_multiplicity(invariant1, &multiplicity[0]));
  }
```

The definition of the ***compare_multiplicity*** predicate is shown below. This predicate provides the glue that binds ***std::sort*** to our current purpose.

⟨ Invariant multiplicity comparison functor 12b ⟩ ≡

```
  struct compare_multiplicity
  {
    compare_multiplicity(Invariant1 invariant1, size_type* multiplicity)
      : invariant1(invariant1), multiplicity(multiplicity) { }
    bool operator()(const vertex1_t& x, const vertex1_t& y) const {
      return multiplicity[invariant1(x)] < multiplicity[invariant1(y)];
    }
    Invariant1 invariant1;
    size_type* multiplicity;
  };
```

0.4.1 Ordering by DFS Discover Time

Next we order the vertices and edges by DFS discover time. We would normally call the BGL ***depth_first_search*** function to do this, but we want the roots of the DFS tree's to be ordered by invariant multiplicity. Therefore we implement the outer-loop of the DFS here and then call ***depth_first_visit*** to handle the recursive portion of the DFS. The ***record_dfs_order*** adapts the DFS to record the ordering, storing the results in in the ***dfs_vertices*** and ***ordered_edges*** arrays. We then create the ***dfs_num*** array which provides a mapping from vertex to DFS number.

⟨ Order vertices and edges by DFS 13a ⟩ ≡

```
std::vector<default_color_type> color_vec(num_vertices(G1));
safe_iterator_property_map<std::vector<default_color_type>::iterator, IndexMap1>
    color_map(color_vec.begin(), color_vec.size(), index_map1);
record_dfs_order dfs_visitor(dfs_vertices, ordered_edges);
typedef color_traits<default_color_type> Color;
for (vertex_iter u = V_mult.begin(); u != V_mult.end(); ++u) {
    if (color_map[*u] == Color::white()) {
        dfs_visitor.start_vertex(*u, G1);
        depth_first_visit(G1, *u, dfs_visitor, color_map);
    }
}
// Create the dfs_num array and dfs_num_map
dfs_num_vec.resize(num_vertices(G1));
dfs_num = make_safe_iterator_property_map(dfs_num_vec.begin(),
                          dfs_num_vec.size(), index_map1);
size_type n = 0;
for (vertex_iter v = dfs_vertices.begin(); v != dfs_vertices.end(); ++v)
    dfs_num[*v] = n++;
```

The definition of the ***record_dfs_order*** visitor class is as follows.

⟨ DFS visitor to record vertex and edge order 13b ⟩ ≡

```
struct record_dfs_order : default_dfs_visitor
{
    record_dfs_order(std::vector<vertex1_t>& v, std::vector<edge1_t>& e)
        : vertices(v), edges(e) { }

    void discover_vertex(vertex1_t v, const Graph1&) const {
        vertices.push_back(v);
    }
    void examine_edge(edge1_t e, const Graph1& G1) const {
        edges.push_back(e);
    }
    std::vector<vertex1_t>& vertices;
    std::vector<edge1_t>& edges;
};
```

The final stage of the setup is to reorder the edges so that all edges belonging to $G_1[k]$ appear before any edges not in $G_1[k]$, for $k = 1, ..., n$.

⟨ Sort edges according to vertex DFS order 14a ⟩ ≡

```
sort(ordered_edges, edge_cmp(G1, dfs_num));
```

The edge comparison function object is defined as follows.

⟨ Edge comparison predicate 14b ⟩ ≡

```
struct edge_cmp {
  edge_cmp(const Graph1& G1, DFSNumMap dfs_num)
    : G1(G1), dfs_num(dfs_num) { }
  bool operator()(const edge1_t& e1, const edge1_t& e2) const {
    using namespace std;
    vertex1_t u1 = dfs_num[source(e1,G1)], v1 = dfs_num[target(e1,G1)];
    vertex1_t u2 = dfs_num[source(e2,G1)], v2 = dfs_num[target(e2,G1)];
    int m1 = max(u1, v1);
    int m2 = max(u2, v2);
    // lexicographical comparison
    return make_pair(m1, make_pair(u1, v1))
      < make_pair(m2, make_pair(u2, v2));
  }
  const Graph1& G1;
  DFSNumMap dfs_num;
};
```

0.5 Appendix

⟨ Typedefs for commonly used types 14c ⟩ ≡

```
typedef typename graph_traits<Graph1>::vertex_descriptor vertex1_t;
typedef typename graph_traits<Graph2>::vertex_descriptor vertex2_t;
typedef typename graph_traits<Graph1>::edge_descriptor edge1_t;
typedef typename graph_traits<Graph1>::vertices_size_type size_type;
typedef typename Invariant1::result_type invar1_value;
typedef typename Invariant2::result_type invar2_value;
```

⟨ Data members for the parameters 14d ⟩ ≡

```
const Graph1& G1;
const Graph2& G2;
IsoMapping f;
Invariant1 invariant1;
Invariant2 invariant2;
```

```
std::size_t max_invariant;
IndexMap1 index_map1;
IndexMap2 index_map2;
```

⟨ Internal data structures 15a ⟩ ≡

```
std::vector<vertex1_t> dfs_vertices;
typedef std::vector<vertex1_t>::iterator vertex_iter;
std::vector<int> dfs_num_vec;
typedef safe_iterator_property_map<typename std::vector<int>::iterator, IndexMap1>
DFSNumMap dfs_num;
std::vector<edge1_t> ordered_edges;
typedef std::vector<edge1_t>::iterator edge_iter;

std::vector<char> in_S_vec;
typedef safe_iterator_property_map<typename std::vector<char>::iterator,
    IndexMap2> InSMap;
InSMap in_S;

int num_edges_on_k;
```

⟨ Isomorphism algorithm constructor 15b ⟩ ≡

```
isomorphism_algo(const Graph1& G1, const Graph2& G2, IsoMapping f,
                 Invariant1 invariant1, Invariant2 invariant2, std::size_t max_invariant,
                 IndexMap1 index_map1, IndexMap2 index_map2)
    : G1(G1), G2(G2), f(f), invariant1(invariant1), invariant2(invariant2),
      max_invariant(max_invariant),
      index_map1(index_map1), index_map2(index_map2)
{
    in_S_vec.resize(num_vertices(G1));
    in_S = make_safe_iterator_property_map
        (in_S_vec.begin(), in_S_vec.size(), index_map2);
}
```

⟨ *isomorphism.hpp* 15c ⟩ ≡

```
// Copyright (C) 2001 Jeremy Siek, Doug Gregor, Brian Osman
//
// Permission to copy, use, sell and distribute this software is granted
// provided this copyright notice appears in all copies.
// Permission to modify the code and to distribute modified code is granted
// provided this copyright notice appears in all copies, and a notice
// that the code was modified is included with the copyright notice.
//
// This software is provided “as is” without express or implied warranty,
// and with no claim as to its suitability for any purpose.
#ifndef BOOST_GRAPH_ISOMORPHISM_HPP
#define BOOST_GRAPH_ISOMORPHISM_HPP

#include <utility>
#include <vector>
#include <iterator>
#include <algorithm>
#include <boost/graph/iteration_macros.hpp>
#include <boost/graph/depth_first_search.hpp>
#include <boost/utility.hpp>
#include <boost/algorithm.hpp>
#include <boost/pending/indirect_cmp.hpp> // for make_indirect_pmap

namespace boost {

namespace detail {

 ⟨Isomorphism algorithm class 10b⟩

template <typename Graph, typename InDegreeMap>
void compute_in_degree(const Graph& g, InDegreeMap in_degree_map)
{
    BGL_FORALL_VERTICES_T(v, g, Graph)
        put(in_degree_map, v, 0);

    BGL_FORALL_VERTICES_T(u, g, Graph)
      BGL_FORALL_ADJ_T(u, v, g, Graph)
        put(in_degree_map, v, get(in_degree_map, v) + 1);
}

} // namespace detail


 ⟨Degree vertex invariant functor 4⟩

 ⟨Isomorphism function interface 8c⟩
 ⟨Isomorphism function body 9a⟩

namespace detail {

template <typename Graph1, typename Graph2,
          typename IsoMapping,
          typename IndexMap1, typename IndexMap2,
          typename P, typename T, typename R>
bool isomorphism_impl(const Graph1& G1, const Graph2& G2,
                     IsoMapping f, IndexMap1 index_map1, IndexMap2 index_map2,
                     const bgl_named_params<P,T,R>& params)
{
  std::vector<std::size_t> in_degree1_vec(num_vertices(G1));
  typedef safe_iterator_property_map<std::vector<std::size_t>::iterator, IndexMap1> In
  InDeg1 in_degree1(in_degree1_vec.begin(), in_degree1_vec.size(), index_map1);
  compute_in_degree(G1, in_degree1);

  std::vector<std::size_t> in_degree2_vec(num_vertices(G2));
  typedef safe_iterator_property_map<std::vector<std::size_t>::iterator, IndexMap2> In
  InDeg2 in_degree2(in_degree2_vec.begin(), in_degree2_vec.size(), index_map2);
  compute_in_degree(G2, in_degree2);

  degree_vertex_invariant<InDeg1, Graph1> invariant1(in_degree1, G1);
  degree_vertex_invariant<InDeg2, Graph2> invariant2(in_degree2, G2);

  return isomorphism(G1, G2, f,
        choose_param(get_param(params, vertex_invariant1_t()), invariant1),
        choose_param(get_param(params, vertex_invariant2_t()), invariant2),
        choose_param(get_param(params, vertex_max_invariant_t()), invariant2.max()
        index_map1, index_map2
        );
}

} // namespace detail


// Named parameter interface
template <typename Graph1, typename Graph2, class P, class T, class R>
bool isomorphism(const Graph1& g1,
                 const Graph2& g2,
                 const bgl_named_params<P,T,R>& params)
{
  typedef typename graph_traits<Graph2>::vertex_descriptor vertex2_t;
  typename std::vector<vertex2_t>::size_type n = num_vertices(g1);
  std::vector<vertex2_t> f(n);
  return detail::isomorphism_impl
    (g1, g2,
     choose_param(get_param(params, vertex_isomorphism_t()),
          make_safe_iterator_property_map(f.begin(), f.size(),
                  choose_const_pmap(get_param(params, vertex_index1),
                                    g1, vertex_index), vertex2_t())),
     choose_const_pmap(get_param(params, vertex_index1), g1, vertex_index),
     choose_const_pmap(get_param(params, vertex_index2), g2, vertex_index),
     params
     );
```

```
}

// All defaults interface
template <typename Graph1, typename Graph2>
bool isomorphism(const Graph1& g1, const Graph2& g2)
{
  return isomorphism(g1, g2,
    bgl_named_params<int, buffer_param_t>(0));// bogus named param
}


// Verify that the given mapping iso_map from the vertices of g1 to the
// vertices of g2 describes an isomorphism.
// Note: this could be made much faster by specializing based on the graph
// concepts modeled, but since we're verifying an O(n^(lg n)) algorithm,
// O(n^4) won't hurt us.
template<typename Graph1, typename Graph2, typename IsoMap>
inline bool verify_isomorphism(const Graph1& g1, const Graph2& g2, IsoMap iso_map)
{
  if (num_vertices(g1) != num_vertices(g2) || num_edges(g1) != num_edges(g2))
    return false;

  for (typename graph_traits<Graph1>::edge_iterator e1 = edges(g1).first;
       e1 != edges(g1).second; ++e1) {
    bool found_edge = false;
    for (typename graph_traits<Graph2>::edge_iterator e2 = edges(g2).first;
         e2 != edges(g2).second && !found_edge; ++e2) {
      if (source(*e2, g2) == get(iso_map, source(*e1, g1)) &&
          target(*e2, g2) == get(iso_map, target(*e1, g1))) {
        found_edge = true;
      }
    }

    if (!found_edge)
      return false;
  }

  return true;
}

} // namespace boost

#include <boost/graph/iteration_macros_undef.hpp>

#endif // BOOST_GRAPH_ISOMORPHISM_HPP
```

Bibliography

[1] N. Deo, J. M. Davis, and R. E. Lord. A new algorithm for digraph isomorphism. *BIT*, 17:16–30, 1977.

[2] S. Fortin. Graph isomorphism problem. Technical Report 96-20, University of Alberta, Edomonton, Alberta, Canada, 1996.

[3] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice Hall, 1977.

[4] E. Sussenguth. A graph theoretic algorithm for matching chemical structure. *J. Chem. Doc.*, 5:36–43, 1965.

[5] S. H. Unger. Git—a heuristic program for testing pairs of directed line graphs for isomorphism. *Comm. ACM*, 7:26–34, 1964.