An Implementation of Isomorphism Testing

Jeremy G. Siek

December 3, 2001

0.1 Introduction

This paper documents the implementation of the ***isomorphism()*** function of the Boost Graph Library. This 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 *isomorphic*.

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.

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

In the following discussion we will need to use several 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 $G[V_s]$.

In some places we express a function as a set of pairs, so the set $f = \{\langle a_1, b_1\rangle, \ldots, \langle a_n, b_n\rangle\}$ means $f(a_i) = b_i$ for $i = 1, \ldots, n$.

0.2 Exhaustive 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 return true, or by exhausting all possibilities and returning false.

We are going to consider the vertices of G_1 in a specific order (more about this later), so assume that the vertices of G_1 are labeled $1, \ldots, N$ according to the order that we plan to add them to the subgraph. Let $G_1[k]$ denote the subgraph of G_1 induced by the first k vertices, with $G_1[0]$ being an empty graph. At each stage of the recursion we start with an isomorphism f_{k-1} between $G_1[k-1]$ and a subgraph of G_2, which we denote by $G_2[S]$, so $G_1[k-1] \cong G_2[S]$. The vertex set S is the subset of V_2 that corresponds via f_{k-1} to the first $k-1$ vertices in G_1. We try to extend the isomorphism by finding a vertex $v \in V_2 - S$ that matches with vertex k. If a matching vertex is found, we have a new isomorphism f_k with $G_1[k] \cong G_2[S \cup \{v\}]$.

```
ISOMORPH(k, S, f_{k-1}) ≡
  if (k = |V_1| + 1)
    return true
  for each vertex v ∈ V_2 − S
    if (MATCH(k, v))
      f_k = f_{k-1} ∪ ⟨k, v⟩
      ISOMORPH(k + 1, S ∪ {v}, f_k)
    else
      return false
```

ISOMORPH(0, G_1, $\emptyset$, G_2)

The basic idea of the match operation is to check whether $G_1[k]$ is isomorphic to $G_2[S \cup \{v\}]$. We already know that $G_1[k-1] \cong G_2[S]$ with the mapping f_{k-1}, so all we need to do is verify that the edges in $E_1[k] - E_1[k-1]$ connect vertices that correspond to the vertices connected by the edges in $E_2[S \cup \{v\}] - E_2[S]$. The edges in $E_1[k] - E_1[k-1]$ are all the out-edges (k, j) and in-edges (j, k) of k where j is less than or equal to k according to the ordering. The edges in $E_2[S \cup \{v\}] - E_2[S]$ consists of all the out-edges (v, u) and in-edges (u, v) of v where $u \in S$.

MATCH(k, v) $\equiv$
$out \leftarrow \forall (k,j) \in E_1[k] - E_1[k-1] \Big((v, f(j)) \in E_2[S \cup \{v\}] - E_2[S] \Big)$
$in \leftarrow \forall (j,k) \in E_1[k] - E_1[k-1] \Big((f(j), v) \in E_2[S \cup \{v\}] - E_2[S] \Big)$
return $out \bigwedge in$

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 [**?**, **?**] that originated in [**?**, **?**].

0.3 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, we only need to consider those vertices that have the same vertex invariant number. 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 out-degree of the vertex as the vertex invariant, 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.

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.

Also, we extend the MATCH operation to use the vertex invariants to help rule out vertices.

MATCH-INVAR(k, v) $\equiv$
$out \leftarrow \forall (k,j) \in E_1[k] - E_1[k-1] \Big((v, f(j)) \in E_2[S \cup \{v\}] - E_2[S] \bigwedge i(v) = i(k) \Big)$
$in \leftarrow \forall (j,k) \in E_1[k] - E_1[k-1] \Big((f(j), v) \in E_2[S \cup \{v\}] - E_2[S] \bigwedge i(v) = i(k) \Big)$
return $out \bigwedge in$

0.4 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.

0.4.1 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.

0.4.2 Adjacent First

The MATCH operation only considers edges when the other vertex already has a mapping defined. This means that the MATCH operation can only weed out vertices that are adjacent to vertices that have already been matched. Therefore, when choosing the next vertex to examine, it is desirable to choose one that is adjacent a vertex already in S_1.

0.4.3 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.

0.5 Implementation

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 requirements on type template parameters are described below in the section "Concept checking".

⟨ Isomorphism Function Interface 3a ⟩ ≡

```
template <typename Graph1, typename Graph2,
          typename IndexMapping, typename VertexInvariant,
          typename IndexMap1, typename IndexMap2>
bool isomorphism(const Graph1& g1, const Graph2& g2,
                 IndexMapping f, VertexInvariant invariant,
                 IndexMap1 index_map1, IndexMap2 index_map2)
```

The main outline of the ***isomorphism*** function is as follows. Most of the steps in this function are for setting up the vertex ordering, first ordering the vertices by invariant multiplicity and then by DFS order. The last step is the call to the ***isomorph*** function which starts the backtracking search.

⟨ Isomorphism Function Body 3b ⟩ ≡

```
{
  ⟨Some type definitions and iterator declarations 4a⟩
  ⟨Concept checking 4b⟩
  ⟨Quick return with false if |V 1| ≠ |V 2| 4c⟩
  ⟨Compute vertex invariants 5b⟩
  ⟨Quick return if the graph's invariants do not match 6a⟩
  ⟨Compute invariant multiplicity 6b⟩
```

⟨ *Sort vertices by invariant multiplicity 6c* ⟩
⟨ *Order the vertices by DFS discover time 7a* ⟩
⟨ *Order the edges by DFS discover time 8a* ⟩
⟨ *Invoke recursive* **isomorph** *function 9a* ⟩
}

There are some types that will be used throughout the function, which we create shortened names for here. We will also need vertex iterators for ***g1*** and ***g2*** in several places, so we define them here.

⟨ Some type definitions and iterator declarations 4a ⟩ ≡

```
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;
typename graph_traits<Graph1>::vertex_iterator i1, i1_end;
typename graph_traits<Graph2>::vertex_iterator i2, i2_end;
```

We use the Boost Concept Checking Library to make sure that the type arguments to the function fulfill there requirements. The ***Graph1*** type must be a VertexListGraph and a EdgeListGraph. The ***Graph2*** type must be a VertexListGraph and a BidirectionalGraph. The ***IndexMapping*** type that represents 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 4b ⟩ ≡

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

// Property map requirements
function_requires< ReadWritePropertyMapConcept<IndexMapping, vertex1_t> >();
typedef typename property_traits<IndexMapping>::value_type IndexMappingValue;
BOOST_STATIC_ASSERT((is_same<IndexMappingValue, 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));
```

If there are no vertices in either graph, then they are trivially isomorphic.

⟨ Quick return with false if $|V_1| \neq |V_2|$ 4c ⟩ ≡

```
if (num_vertices(g1) != num_vertices(g2))
  return true;
```

0.5.1 Ordering by Vertex Invariant Multiplicity

The user can supply the vertex invariant function as a function object (the ***invariant*** parameter), but we also define a default which uses the out-degree of a vertex. The following is the definition of the function object for the default vertex invariant. User-defined vertex invariant function objects should follow the same pattern.

⟨ Degree vertex invariant 5a ⟩ ≡

```
struct degree_vertex_invariant {
  template <typename Graph> struct result {
    typedef typename graph_traits<Graph>::degree_size_type type;
  };
  template <typename Graph>
  typename graph_traits<Graph>::degree_size_type
  operator()(typename graph_traits<Graph>::vertex_descriptor v, const Graph& g)
    { return out_degree(v, g); }
};
```

Since the invariant function may be expensive to compute, we pre-compute the invariant numbers for every vertex in the two graphs. The variables ***invar1*** and ***invar2*** are property maps for accessing the stored invariants, which are described next.

⟨ Compute vertex invariants 5b ⟩ ≡

```
  ⟨Setup storage for vertex invariants 5c⟩
for (tie(i1, i1_end) = vertices(g1); i1 != i1_end; ++i1)
  invar1[*i1] = invariant(*i1, g1);
for (tie(i2, i2_end) = vertices(g2); i2 != i2_end; ++i2)
  invar2[*i2] = invariant(*i2, g2);
```

We store the invariants in two vectors, indexed by the vertex indices of the two graphs. We then create property maps for accessing these two vectors in a more convenient fashion (they go directly from vertex to invariant, instead of vertex to index to invariant).

⟨ Setup storage for vertex invariants 5c ⟩ ≡

```
typedef typename VertexInvariant::template result<Graph1>::type
  InvarValue1;
typedef typename VertexInvariant::template result<Graph2>::type
  InvarValue2;
typedef std::vector<InvarValue1> invar_vec1_t;
typedef std::vector<InvarValue2> invar_vec2_t;
invar_vec1_t invar1_vec(num_vertices(g1));
invar_vec2_t invar2_vec(num_vertices(g2));
typedef typename invar_vec1_t::iterator vec1_iter;
typedef typename invar_vec2_t::iterator vec2_iter;
iterator_property_map<vec1_iter, IndexMap1, InvarValue1, InvarValue1&>
  invar1(invar1_vec.begin(), index_map1);
iterator_property_map<vec2_iter, IndexMap2, InvarValue2, InvarValue2&>
  invar2(invar2_vec.begin(), index_map2);
```

As discussed in §0.3, we can quickly rule out the possibility of any isomorphism between two graphs by checking to see if the vertex invariants can match up. We sort both vectors of vertex invariants, and then check to see if they are equal.

⟨ Quick return if the graph's invariants do not match 6a ⟩ ≡

```
{ // check if the graph's invariants do not match
  invar_vec1_t invar1_tmp(invar1_vec);
  invar_vec2_t invar2_tmp(invar2_vec);
  std::sort(invar1_tmp.begin(), invar1_tmp.end());
  std::sort(invar2_tmp.begin(), invar2_tmp.end());
  if (! std::equal(invar1_tmp.begin(), invar1_tmp.end(),
                   invar2_tmp.begin()))
    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.

⟨ Compute invariant multiplicity 6b ⟩ ≡

```
std::vector<std::size_t> invar_mult(num_vertices(g1), 0);
for (tie(i1, i1_end) = vertices(g1); i1 != i1_end; ++i1)
  ++invar_mult[invar1[*i1]];
```

We then order the vertices by their invariant multiplicity. This will allow us to search the more constrained vertices first. Since we will need to know the permutation from the original order to the new order, we do not sort the vertices directly. Instead we sort the vertex indices, creating the ***perm*** array. Once sorted, this array provides a mapping from the new index to the old index. We then use the ***permute*** function to sort the vertices of the graph, which we store in the ***g1_vertices*** vector.

⟨ Sort vertices by invariant multiplicity 6c ⟩ ≡

```
std::vector<size_type> perm;
integer_range<size_type> range(0, num_vertices(g1));
std::copy(range.begin(), range.end(), std::back_inserter(perm));
std::sort(perm.begin(), perm.end(),
          detail::compare_invariant_multiplicity(invar1_vec.begin(),
                                                 invar_mult.begin()));

std::vector<vertex1_t> g1_vertices;
for (tie(i1, i1_end) = vertices(g1); i1 != i1_end; ++i1)
  g1_vertices.push_back(*i1);
permute(g1_vertices.begin(), g1_vertices.end(), perm.begin());
```

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

⟨ Compare multiplicity predicate 6d ⟩ ≡

```
namespace detail {
  template <typename InvarMap, typename MultMap>
  struct compare_invariant_multiplicity_predicate
  {
    compare_invariant_multiplicity_predicate(InvarMap i, MultMap m)
      : m_invar(i), m_mult(m) { }

    template <typename Vertex>
    bool operator()(const Vertex& x, const Vertex& y) const
```

```
    { return m_mult[m_invar[x]] < m_mult[m_invar[x]]; }

  InvarMap m_invar;
  MultMap m_mult;
};
template <typename InvarMap, typename MultMap>
compare_invariant_multiplicity_predicate<InvarMap, MultMap>
compare_invariant_multiplicity(InvarMap i, MultMap m) {
  return compare_invariant_multiplicity_predicate<InvarMap, MultMap>(i,m);
}
} // namespace detail
```

0.5.2 Ordering by DFS Discover Time

To implement the "visit adjacent vertices first" heuristic, we order the vertices according to DFS discover time. We replace the ordering in ***perm*** with the new DFS ordering. Again, we use ***permute*** to sort the vertices of graph ***g1***.

⟨ Order the vertices by DFS discover time 7a ⟩ ≡

```
{
  perm.clear();
  ⟨Compute DFS discover times 7b⟩
  g1_vertices.clear();
  for (tie(i1, i1_end) = vertices(g1); i1 != i1_end; ++i1)
    g1_vertices.push_back(*i1);
  permute(g1_vertices.begin(), g1_vertices.end(), perm.begin());
}
```

We implement the outer-loop of the DFS here, instead of calling the ***depth_first_search*** function, because we want the roots of the DFS tree's to be ordered by invariant multiplicity. We call ***depth_first_visit*** to implement the recursive portion of the DFS. The ***record_dfs_order*** adapts the DFS to record the order in which DFS discovers the vertices.

⟨ Compute DFS discover times 7b ⟩ ≡

```
std::vector<default_color_type> color_vec(num_vertices(g1));
for (typename std::vector<vertex1_t>::iterator ui = g1_vertices.begin();
     ui != g1_vertices.end(); ++ui) {
  if (color_vec[get(index_map1, *ui)]
      == color_traits<default_color_type>::white()) {
    depth_first_visit
      (g1, *ui, detail::record_dfs_order<Graph1, IndexMap1>(perm,
                                                            index_map1),
       make_iterator_property_map(&color_vec[0], index_map1,
                                  color_vec[0]));
  }
}
```

The definition of the ***record_dfs_order*** visitor class is as follows. The index of each vertex is recorded in the ***dfs_order*** vector (which is the ***perm*** vector) in the ***discover_vertex*** event point.

⟨ Record DFS ordering visitor 7c ⟩ ≡

```
namespace detail {
  template <typename Graph1, typename IndexMap1>
  struct record_dfs_order : public default_dfs_visitor {
    typedef typename graph_traits<Graph1>::vertices_size_type size_type;
    typedef typename graph_traits<Graph1>::vertex_descriptor vertex;

    record_dfs_order(std::vector<size_type>& dfs_order, IndexMap1 index)
      : dfs_order(dfs_order), index(index) { }

    void discover_vertex(vertex v, const Graph1& g) const {
      dfs_order.push_back(get(index, v));
    }
    std::vector<size_type>& dfs_order;
    IndexMap1 index;
  };
} // namespace detail
```

In the MATCH operation, we need to examine all the edges in the set $E_1[k] - E_1[k-1]$. That is, we need to loop through all the edges of the form (k, j) or (j, k) where $j \leq k$. To do this efficiently, we create an array of all the edges in G_1 that has been sorted so that $E_1[k] - E_1[k-1]$ forms a contiguous range. To each edge $e = (u, v)$ we assign the number $\max(u, v)$, and then sort the edges by this number. All the edges $(u, v) \in E_1[k] - E_1[k-1]$ can then be identified because $\max(u, v) = k$. The following code creates an array of edges and then sorts them. The ***edge_ordering_fun*** function object is described next.

⟨ Order the edges by DFS discover time 8a ⟩ ≡

```
typedef typename graph_traits<Graph1>::edge_descriptor edge1_t;
std::vector<edge1_t> edge_set;
std::copy(edges(g1).first, edges(g1).second, std::back_inserter(edge_set));

std::sort(edge_set.begin(), edge_set.end(),
          detail::edge_ordering
          (make_iterator_property_map(perm.begin(), index_map1, perm[0]), g1));
```

The ***edge_num*** function computes the ordering number for an edge, which for edge $e = (u, v)$ is $\max(u, v)$. The ***edge_ordering_fun*** function object simply returns comparison of two edge's ordering numbers.

⟨ Isomorph edge ordering predicate 8b ⟩ ≡

```
namespace detail {

  template <typename VertexIndexMap, typename Graph>
  std::size_t edge_num(const typename graph_traits<Graph>::edge_descriptor e,
                       VertexIndexMap index_map, const Graph& g) {
    return std::max(get(index_map, source(e, g)), get(index_map, target(e, g)));
  }

  template <typename VertexIndexMap, typename Graph>
  class edge_ordering_fun {
  public:
    edge_ordering_fun(VertexIndexMap vip, const Graph& g)
      : m_index_map(vip), m_g(g) { }
    template <typename Edge>
    bool operator()(const Edge& e1, const Edge& e2) const {
```

```
    return edge_num(e1, m_index_map, m_g) < edge_num(e2, m_index_map, m_g);
  }
  VertexIndexMap m_index_map;
  const Graph& m_g;
};
template <class VertexIndexMap, class G>
inline edge_ordering_fun<VertexIndexMap,G>
edge_ordering(VertexIndexMap vip, const G& g)
{
  return edge_ordering_fun<VertexIndexMap,G>(vip, g);
}
} // namespace detail
```

We are now ready to enter the main part of the algorithm, the backtracking search implemented by the ***isomorph*** function (which corresponds to the ISOMORPH algorithm). The set S is not represented directly; instead we represent $V_2 - S$. Initially $S = \emptyset$ so $V_2 - S = V_2$. We use the permuted indices for the vertices of graph ***g1***. We represent $V_2 - S$ with a bitset. We use ***std::vector*** instead of ***boost::dyn_bitset*** for speed instead of space.

⟨ Invoke recursive ***isomorph*** function 9a ⟩ ≡

```
std::vector<char> not_in_S_vec(num_vertices(g2), true);
iterator_property_map<char*, IndexMap2, char, char&>
  not_in_S(&not_in_S_vec[0], index_map2);

return detail::isomorph(g1_vertices.begin(), g1_vertices.end(),
        edge_set.begin(), edge_set.end(), g1, g2,
        make_iterator_property_map(perm.begin(), index_map1, perm[0]),
        index_map2, f, invar1, invar2, not_in_S);
```

0.5.3 Implementation of ISOMORPH

The ISOMORPH algorithm is implemented with the ***isomorph*** function. The vertices of G_1 are searched in the order specified by the iterator range ***[k_iter,last)***. The function returns true if a isomorphism is found between the vertices of G_1 in ***[k_iter,last)*** and the vertices of G_2 in ***not_in_S***. The mapping is recorded in the parameter ***f***.

⟨ Signature for the recursive isomorph function 9b ⟩ ≡

```
template <class VertexIter, class EdgeIter, class Graph1, class Graph2,
  class IndexMap1, class IndexMap2, class IndexMapping,
  class Invar1, class Invar2, class Set>
bool isomorph(VertexIter k_iter, VertexIter last,
              EdgeIter edge_iter, EdgeIter edge_iter_end,
              const Graph1& g1, const Graph2& g2,
              IndexMap1 index_map1,
              IndexMap2 index_map2,
              IndexMapping f, Invar1 invar1, Invar2 invar2,
              const Set& not_in_S)
```

The steps for this function are as follows.

⟨ Body of the isomorph function 9c ⟩ ≡

```
{
  ⟨Some typedefs and variable declarations 10a⟩
  ⟨Return true if matching is complete 10b⟩
  ⟨Create a copy of f k − 1 which will become f k 13a⟩
  ⟨Compute M, the potential matches for k 10c⟩
  ⟨Invoke isomorph for each vertex in M 13b⟩
}
```

Here we create short names for some often-used types and declare some variables.

⟨ Some typedefs and variable declarations 10a ⟩ ≡

```
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;

vertex1_t k = *k_iter;
```

We have completed creating an isomorphism if ***k_iter*** == ***last***.

⟨ Return true if matching is complete 10b ⟩ ≡

```
if (k_iter == last)
  return true;
```

In the pseudo-code for ISOMORPH, we iterate through each vertex in $v \in V_2 - S$ and check if k and v can match. A more efficient approach is to directly iterate through the potential matches for k, for this often is many fewer vertices than $V_2 - S$. Let M be the set of potential matches for k. M consists of all the vertices $v \in V_2 - S$ such that if (k, j) or $(j, k) \in E_1[k] - E_1[k-1]$ then $(v, f(j)$ or $(f(j), v) \in E_2$ with $i(v) = i(k)$. Note that this means if there are no edges in $E_1[k] - E_1[k-1]$ then $M = V_2 - S$. In the case where there are edges in $E_1[k] - E_1[k-1]$ we break the computation of M into two parts, computing out sets which are vertices that can match according to an out-edge of k, and computing in sets which are vertices that can match according to an in-edge of k.

The implementation consists of a loop through the edges of $E_1[k] - E_1[k-1]$. The straightforward implementation would initialize $M \leftarrow V_2 - S$, and then intersect M with the out or in set for each edge. However, to reduce the cost of the intersection operation, we start with $M \leftarrow \emptyset$, and on the first iteration of the loop we do $M \leftarrow out$ or $M \leftarrow in$ instead of an intersection operation.

⟨ Compute M, the potential matches for k 10c ⟩ ≡

```
std::vector<vertex2_t> potential_matches;
bool some_edges = false;

for (; edge_iter != edge_iter_end; ++edge_iter) {
  if (get(index_map1, k) != edge_num(*edge_iter, index_map1, g1))
    break;
  if (k == source(*edge_iter, g1)) { // (k,j)
    ⟨Compute the out set 11a⟩
    if (some_edges == false) {
      ⟨Perform M ← out 11b⟩
    } else {
      ⟨Perform M ← M ∩ out 11c⟩
    }
    some_edges = true;
```

```
  } else { // (j,k)
    ⟨Compute the in set 12a⟩
    if (some_edges == false) {
      ⟨Perform M ← in 12b⟩
    } else {
      ⟨Perform M ← M ∩ in 12c⟩
    }
    some_edges = true;
  }
  if (potential_matches.empty())
    break;
} // for edge_iter
if (some_edges == false) {
  ⟨Perform M ← V 2 − S 12d⟩
}
```

To compute the out set, we iterate through the out-edges (k, j) of k, and for each j we iterate through the in-edges $(v, f(j))$ of $f(j)$, putting all of the v's in out that have the same vertex invariant as k, and which are in $V_2 - S$. Figure 1 depicts the computation of the out set. The implementation is as follows.

⟨ Compute the out set 11a ⟩ ≡

```
vertex1_t j = target(*edge_iter, g1);
std::vector<vertex2_t> out;
typename graph_traits<Graph2>::in_edge_iterator ei, ei_end;
for (tie(ei, ei_end) = in_edges(get(f, j), g2); ei != ei_end; ++ei) {
  vertex2_t v = source(*ei, g2); // (v,f[j])
  if (invar1[k] == invar2[v] && not_in_S[v])
    out.push_back(v);
}
```

Here initialize M with the out set. Since we are representing sets with sorted vectors, we sort ***out*** before copying to ***potential_matches***.

⟨ Perform $M \leftarrow out$ 11b ⟩ ≡

```
indirect_cmp<IndexMap2,std::less<std::size_t> > cmp(index_map2);
std::sort(out.begin(), out.end(), cmp);
std::copy(out.begin(), out.end(), std::back_inserter(potential_matches));
```

We use ***std::set_intersection*** to implement $M \leftarrow M \cap out$. Since there is no version of ***std::set_intersection*** that works in-place, we create a temporary for the result and then swap.

⟨ Perform $M \leftarrow M \cap out$ 11c ⟩ ≡

```
indirect_cmp<IndexMap2,std::less<std::size_t> > cmp(index_map2);
std::sort(out.begin(), out.end(), cmp);
std::vector<vertex2_t> tmp_matches;
std::set_intersection(out.begin(), out.end(),
                      potential_matches.begin(), potential_matches.end(),
                      std::back_inserter(tmp_matches), cmp);
std::swap(potential_matches, tmp_matches);
```

The in set is is constructed by iterating through the in-edges (j, k) of k, and for each j we iterate through the out-edges $(f(j), v)$ of $f(j)$. We put all of the v's in in that have the same vertex invariant as k, and

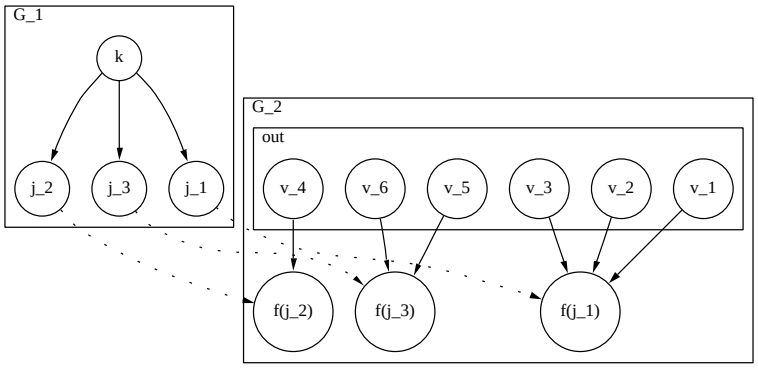

Figure 1: Computing the out set.

which are in $V_2 - S$. Figure 2 depicts the computation of the in set. The following code computes the in set.

⟨ Compute the in set 12a ⟩ ≡

```
vertex1_t j = source(*edge_iter, g1);
std::vector<vertex2_t> in;
typename graph_traits<Graph2>::out_edge_iterator ei, ei_end;
for (tie(ei, ei_end) = out_edges(get(f, j), g2); ei != ei_end; ++ei) {
  vertex2_t v = target(*ei, g2); // (f[j],v)
  if (invar1[k] == invar2[v] && not_in_S[v])
    in.push_back(v);
}
```

Here initialize M with the in set. Since we are representing sets with sorted vectors, we sort ***in*** before copying to ***potential_matches***.

⟨ Perform $M \leftarrow in$ 12b ⟩ ≡

```
indirect_cmp<IndexMap2,std::less<std::size_t> > cmp(index_map2);
std::sort(in.begin(), in.end(), cmp);
std::copy(in.begin(), in.end(), std::back_inserter(potential_matches));
```

Again we use ***std::set_intersection*** on sorted vectors to implement $M \leftarrow M \cap in$.

⟨ Perform $M \leftarrow M \cap in$ 12c ⟩ ≡

```
indirect_cmp<IndexMap2, std::less<std::size_t> > cmp(index_map2);
std::sort(in.begin(), in.end(), cmp);
std::vector<vertex2_t> tmp_matches;
std::set_intersection(in.begin(), in.end(),
                      potential_matches.begin(), potential_matches.end(),
                      std::back_inserter(tmp_matches), cmp);
std::swap(potential_matches, tmp_matches);
```

In the case where there were no edges in $E_1[k] - E_1[k-1]$, then $M = V_2 - S$, so here we insert all the vertices from V_2 that are not in S.

⟨ Perform $M \leftarrow V_2 - S$ 12d ⟩ ≡

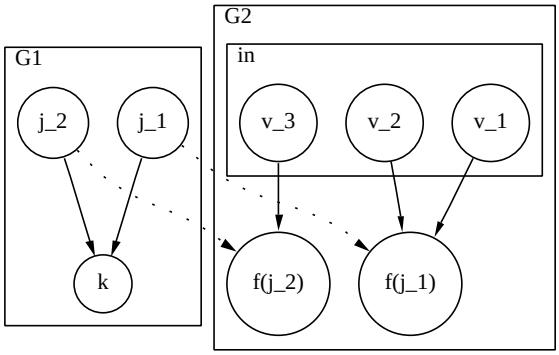

Figure 2: Computing the in set.

```
typename graph_traits<Graph2>::vertex_iterator vi, vi_end;
for (tie(vi, vi_end) = vertices(g2); vi != vi_end; ++vi)
  if (not_in_S[*vi])
    potential_matches.push_back(*vi);
```

For each vertex v in the potential matches M, we will create an extended isomorphism $f_k = f_{k-1} \cup \langle k, v \rangle$. First we create a local copy of f_{k-1}.

⟨ Create a copy of f_{k-1} which will become f_k 13a ⟩ ≡

```
std::vector<vertex2_t> my_f_vec(num_vertices(g1));
typedef typename std::vector<vertex2_t>::iterator vec_iter;
iterator_property_map<vec_iter,      IndexMap1, vertex2_t, vertex2_t&>
  my_f(my_f_vec.begin(), index_map1);

typename graph_traits<Graph1>::vertex_iterator i1, i1_end;
for (tie(i1, i1_end) = vertices(g1); i1 != i1_end; ++i1)
  my_f[*i1] = get(f, *i1);
```

Next we enter the loop through every vertex v in M, and extend the isomorphism with $\langle k, v \rangle$. We then update the set S (by removing v from $V_2 - S$) and make the recursive call to ***isomorph***. If ***isomorph*** returns successfully, we have found an isomorphism for the complete graph, so we copy our local mapping into the mapping from the previous calling function.

⟨ Invoke isomorph for each vertex in M 13b ⟩ ≡

```
for (std::size_t j = 0; j < potential_matches.size(); ++j) {
  my_f[k] = potential_matches[j];
  ⟨Perform S' = S − {v} 14a⟩
  if (isomorph(boost::next(k_iter), last, edge_iter, edge_iter_end, g1, g2,
               index_map1, index_map2,
               my_f, invar1, invar2, my_not_in_S)) {
    for (tie(i1, i1_end) = vertices(g1); i1 != i1_end; ++i1)
      put(f, *i1, my_f[*i1]);
    return true;
  }
}
return false;
```

We need to create the new set $S' = S - \{v\}$, which will be the S for the next invocation to ***isomorph***. As before, we represent $V_2 - S'$ instead of S' and use a bitset.

⟨ Perform $S' = S - \{v\}$ 14a ⟩ ≡

```
std::vector<char> my_not_in_S_vec(num_vertices(g2));
iterator_property_map<char*, IndexMap2, char, char&>
  my_not_in_S(&my_not_in_S_vec[0], index_map2);
typename graph_traits<Graph2>::vertex_iterator vi, vi_end;
for (tie(vi, vi_end) = vertices(g2); vi != vi_end; ++vi)
  my_not_in_S[*vi] = not_in_S[*vi];;
my_not_in_S[potential_matches[j]] = false;
```

0.6 Appendix

Here we output the header file ***isomorphism.hpp***. We add a copyright statement, include some files, and then pull the top-level code parts into namespace ***boost***.

⟨ ***isomorphism.hpp*** 14b ⟩ ≡

```
// (C) Copyright Jeremy Siek 2001. Permission to copy, use, modify,
// sell and distribute this software is granted provided this
// copyright notice appears in all copies. This software is provided
// "as is" without express or implied warranty, and with no claim as
// to its suitability for any purpose.

// See http://www.boost.org/libs/graph/doc/isomorphism-impl.pdf
// for a description of the implementation of the isomorphism function
// defined in this header file.

#ifndef BOOST_GRAPH_ISOMORPHISM_HPP
#define BOOST_GRAPH_ISOMORPHISM_HPP

#include <algorithm>
#include <boost/graph/detail/set_adaptor.hpp>
#include <boost/pending/indirect_cmp.hpp>
#include <boost/graph/detail/permutation.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/graph/graph_concepts.hpp>
#include <boost/property_map.hpp>
#include <boost/pending/integer_range.hpp>
#include <boost/limits.hpp>
#include <boost/static_assert.hpp>
#include <boost/graph/depth_first_search.hpp>

namespace boost {

  ⟨Degree vertex invariant 5a⟩

  namespace detail {
    ⟨Signature for the recursive isomorph function 9b⟩
    ⟨Body of the isomorph function 9c⟩
  } // namespace detail
```

⟨*Record DFS ordering visitor 7c*⟩
⟨*Compare multiplicity predicate 6d*⟩
⟨*Isomorph edge ordering predicate 8b*⟩

⟨*Isomorphism Function Interface 3a*⟩
⟨*Isomorphism Function Body 3b*⟩

```
// 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 = is_default_param(get_param(params, vertex_isomorphism_t()))
      ? num_vertices(g1) : 1;
  std::vector<vertex2_t> f(n);
  vertex2_t x;
  degree_vertex_invariant default_invar;
  return isomorphism
    (g1, g2,
     choose_param
     (get_param(params, vertex_isomorphism_t()),
      make_iterator_property_map
      (f.begin(),
       choose_const_pmap(get_param(params, vertex_index1),
                         g1, vertex_index), x)),
     choose_param(get_param(params, vertex_invariant_t()),
                  default_invar),
     choose_const_pmap(get_param(params, vertex_index1),
                       g1, vertex_index),
     choose_const_pmap(get_param(params, vertex_index2),
                       g2, vertex_index)
     );
}

// All defaults interface
template <typename Graph1, typename Graph2>
bool isomorphism(const Graph1& g1, const Graph2& g2)
{
  typedef typename graph_traits<Graph1>::vertices_size_type size_type;
  typedef typename graph_traits<Graph2>::vertex_descriptor vertex2_t;
  std::vector<vertex2_t> f(num_vertices(g1));
  vertex2_t x;
  degree_vertex_invariant invariant;
  return isomorphism
    (g1, g2,
     make_iterator_property_map(f.begin(), get(vertex_index, g1), x),
     invariant,
     get(vertex_index, g1),
     get(vertex_index, g2)
```

```
        );
    }

} // namespace boost

#endif // BOOST_GRAPH_ISOMORPHISM_HPP
```