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 *isom468(v)2-348(a0.909 Tf;(y)84e-to-84*

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 $u \le k$ and $v \le k$. 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 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 k and $j > k$. i

DFS Order, Starting with Lowest Multiplicity

For this implementation, we combine the above two heuristics in the following way. To implement the \adjacent rst" heuristic we apply DFS to the graph, and use the DFS discovery order as our vertex order. To comply with the \most constrained rst" 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 *x*0.2

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

Case 2: $i \in G_1[k]$ and $i \notin G_2[k]$. Before we extend the subgraph by incrementing k, we need to finish verifying that $G_1[k]$ and $G_2[k]$ are isomorphic. We decrement the *h* Find a match for *j* and continue 8a *i*

```
BGL_FORALL_ADJ_T(f[i], v, G2, Graph2)
   if (invariant2(v) == invariant1(j) && in_S[v] == false) f
      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_ FORALL
```

true

```
          typename IndexMap1, typename IndexMap2>
bool isomorphism(const Graph1& G1, const Graph2&
```

```
    hData members for the parameters 14di
    hInternal data structures 15ai
    friend struct compare_multiplicity;
    hInvariant multiplicity comparison functor 12bi
    hDFS visitor to record vertex and edge order 13bi
    hEdge comparison predicate 14bi
public:
    hIsomorphism algorithm constructor 15bi
    hTest isomorphism member function 11ai
private:
    hMatch function 6ai
g;
```

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*

```
std::vector<invar2_value> invar2_array;
BGL_FORALL
```

tree's to be ordered by invariant multiplicity. Therefore we implement the outer-loop of the DFS here and then call *depth_ rst_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.

h Order vertices and edges by DFS 13a

The nal 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 $k = 1; :::; n$.

```
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(  15 09 cm  BT  /F43 9.967 0 Td8f
// and with no claim as to its suitability for any purpose.
#ifndef BOOST_GRAPH_ISOMORPHISM_HPP
#de ne BOOST_GRAPH_ISOMORPHISM_HPP

#include <
```

```
IsoMapping f, IndexMap1 index_
```

g

```
// All defaults interface
template <typename Graph1, typename Graph2>
```

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]