# An Implementation of the VF2 (Sub)Graph Isomorphism Algorithm Using The Boost Graph Library

Flavio De Lorenzi*

November 30, 2012

**Abstract**

This article describes an implementation of the VF2 algorithm, introduced by Cordella et al. for solving the graph isomorphism and graph-subgraph isomorphism problems, using the Boost Graph Library. This implementation includes algorithmic improvements to account for self-loops and works for directed and undirected graphs.

## 1 Introduction

This section briefly outlines the VF2 algorithm[1], following closely [1, 2].

An isomorphism between two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is a bijective mapping $M$ of the vertices of one graph to vertices of the other graph that preserves the edge structure of the graphs. $M$ is said to be a graph-subgraph isomorphism iff $M$ is an isomorphism between $G_1$ and a subgraph of $G_2$.

A matching process between the two graphs $G_1$ and $G_2$ determines the isomorphism mapping $M$ which associates vertices of $G_1$ with vertices of $G_2$ and vice versa. The matching process can be described by means of a state

*E-mail: fdlorenzi@gmail.com

[1]The original code by Pasquale Foggia and collaborators can be obtained from: `http://www.cs.sunysb.edu/~algorith/implement/vflib/implement.shtml`

space representation combined with a depth-first strategy. The details can be found in [1, 2] and references therein.

Cordella et al. give the following high-level description of the matching algorithm:

**procedure** MATCH($s$)
  **if** $M(s)$ covers all the nodes of $G_2$ **then**
    **return** $M(s)$
  **else**
    Compute the set $P(s)$ of the pairs of vertices for inclusion in $M(s)$
    **for all** $(v, w) \in P(s)$ **do**
      **if** $F(s, v, w)$ **then**
        Compute the state $s'$ obtained by adding $(v, w)$ to $M(s)$
        MATCH($s'$)
      **end if**
    **end for**
    Restore data structures
  **end if**
**end procedure**

$M(s)$ is a partial mapping associated with a state $s$, $P(s)$ is the set of all possible pairs of vertices $(v, w)$ to be added to the current state $s$ and $F(s, v, w)$ is a boolean function (called *feasibility function*) used to prune the search tree. If its value is *true* the state $s'$ obtained by adding $(v, w)$ to $s$ is guaranteed to be a partial isomorphism if $s$ is.

To construct $P(s)$ and $F(s, v, w)$ Cordella et al. define the *out-terminal set* as the set of vertices of $G_1$ that are not in $M(s)$ but are successors of vertices in $M(s)$ (connected by out edges), and the *in-terminal set* as the set of vertices that are not in $M(s)$ but are predecessors of vertices in $M(s)$. Analogue sets are defined for $G_2$.

To compute $P(s)$ and $F(s, v, w)$ efficiently, Cordella et al. employ the following data structures:

- Vectors `core_1` and `core_2` whose dimensions correspond to the number of vertices in $G_1$ and $G_2$, respectively. These vectors store the present mapping.

- Vectors `in_1`, `out_1`, `in_2` and `out_2` used to describe the membership to the terminal sets. `in_1` is non-zero for a particular vertex if the vertex is either in the partial mapping $M(s)$ or belongs to the in-terminal

state of $G_1$. The actual value is given by the level of the depth-first search tree at which the vertex was included in the corresponding set.

# 2 Implementation

The computations of the terminal sets or the addition (deletion) of a pair of vertices to (from) a state are analogous for the two graphs $G_1$ and $G_2$. For example, to add the vertex pair $(v, w)$ with $v \in V_1$ and $w \in V_2$ to vector `core_1` is the same as adding $(w, v)$ to `core_2`. This observation suggests the following improvement to the original VF2 implementation. Instead of implementing a state for $G_1$ and $G_2$ with associated vectors `core_1`, `core_2`, `in_1`, `out_1`, `in_2` and `out_2` directly, we implement a "helper state" class `base_state` associated with a single graph. Class `base_state` then contains `core_`, `in_` and `out_`, and member functions such as *e.g.* `push(const vertex_this_type& v_this, const vertex_other_type& v_other)` to add a vertex pair. The actual state associated with both graphs (implemented in class `state`) can thus be constructed using two "helper states", one for each graph. For instance, the member function `push` to add a pair of vertices to the actual state is obtained as illustrated in the code fragment below:

```
<template<typename Graph1,
          typename Graph2,
          typename IndexMap1,
          typename IndexMap2, .... >
class state
{

    ...

    base_state<Graph1, Graph2, IndexMap1, IndexMap2> state1_;
    base_state<Graph2, Graph1, IndexMap2, IndexMap1> state2_;

public:
    // Add vertex pair to the state
    void push(const vertex1_type& v, const vertex2_type& w)
    {
        state1_.push(v, w);
        state2_.push(w, v);
```

```
    }

    ...

};
```

These classes (`base_state` and `state`) and the non-recursive matching procedure `match` are all members of namespace `boost::detail`.

The functions of the public interface are all defined in namespace `boost` and their documentation will follow in the sections bellow.

## 2.1 Functions for Graph Sub-Graph Isomorphism Testing

```
// Non−named parameter version
template <typename GraphSmall,
          typename GraphLarge,
          typename IndexMapSmall,
          typename IndexMapLarge,
          typename VertexOrderSmall,
          typename EdgeCompatibilityPredicate,
          typename VertexCompatibilityPredicate,
          typename SubGraphIsoMapCallBack>
bool vf2_sub_graph_iso(const GraphSmall& graph_small,
                       const GraphLarge& graph_large,
                       SubGraphIsoMapCallBack user_callback,
                       IndexMapSmall index_map_small,
                       IndexMapLarge index_map_large,
                       const VertexOrderSmall& vertex_order_small,
                       EdgeCompatibilityPredicate edge_comp,
                       VertexCompatibilityPredicate vertex_comp)

// Named parameter interface of vf2_sub_graph_iso
template <typename GraphSmall,
          typename GraphLarge,
          typename VertexOrderSmall,
          typename SubGraphIsoMapCallBack,
          typename Param,
```

```
        typename Tag,
        typename Rest>
bool vf2_sub_graph_iso(const GraphSmall& graph_small,
                       const GraphLarge& graph_large,
                       SubGraphIsoMapCallBack user_callback,
                       const VertexOrderSmall& vertex_order_small,
                       const bgl_named_params<Param, Tag, Rest>&
                           params)

// All default interface for vf2_sub_graph_iso
template <typename GraphSmall,
        typename GraphLarge,
        typename SubGraphIsoMapCallBack>
bool vf2_sub_graph_iso(const GraphSmall& graph_small,
                       const GraphLarge& graph_large,
                       SubGraphIsoMapCallBack user_callback)
```

This algorithm finds all graph-subgraph isomorphism mappings between graphs `graph_small` and `graph_large` and outputs them to `user_callback`. It continues until `user_callback` returns true or the search space has been fully explored.
`EdgeCompatibilityPredicate` and `VertexCompatibilityPredicate` predicates are used to test whether edges and vertices are compatible. By default `always_compatible` is used, which returns true for any pair of vertices or edges.

**Parameters**

IN: `const GraphSmall& graph_small` The (first) smaller graph (fewer vertices) of the pair to be tested for isomorphism. The type `GraphSmall` must be a model of *Vertex List Graph*, *Bidirectional Graph*, *Edge List Graph* and *Adjacency Matrix*.

IN: `const GraphLarge& graph_large` The (second) larger graph to be tested. Type `GraphLarge` must be a model of *Vertex List Graph*, *Bidirectional Graph*, *Edge List Graph* and *Adjacency Matrix*.

OUT: `SubGraphIsoMapCallBack user_callback` A function object to be called when a graph-subgraph isomorphism has been discovered. The `operator()` must have following form:

**template** <**typename** CorrespondenceMap1To2,
**typename** CorrespondenceMap2To1>
**bool operator**()(CorrespondenceMap1To2 f,
CorrespondenceMap2To1 g) **const**

Both the `CorrespondenceMap1To2` and `CorresondenceMap2To1` types are models of *Readable Property Map* and map equivalent vertices across the two graphs given to `vf2_sub_graph_iso` (or `vf2_graph_iso`). An example is given below.

Returning false from the callback will abort the search immediately. Otherwise, the entire search space will be explored.

IN: `const VertexOrderSmall& vertex_order_small` The ordered vertices of the smaller graph `graph_small`. During the matching process the vertices are examined in the order given by `vertex_order_small`. Type `VertexOrderSmall` must be a model of `ContainerConcept` with value type `graph_traits<GraphSmall>::vertex_descriptor`.
*Default:* The vertices are ordered by multiplicity of in/out degree.

### Named Parameters

IN: `vertex_index1(IndexMapSmall index_map_small)` This maps each vertex to an integer in the range `[0, num_vertices(graph_small))`. Type `IndexMapSmall` must be a model of *Readable Property Map*.
*Default:* `get(vertex_index, graph_small)`

IN: `vertex_index2(IndexMapLarge index_map_large)` This maps each vertex to an integer in the range `[0, num_vertices(graph_large))`. Type `IndexMapLarge` must be a model of *Readable Property Map*.
*Default:* `get(vertex_index, graph_large)`

IN: `edges_equivalent(EdgeCompatibilityPredicate edge_comp)` This function object is used to determine if edges between the two graphs `graph_small` and `graph_large` are compatible.
Type `EdgeCompatiblePredicate` must be a model of *Binary Predicate* and have argument types of `graph_traits<GraphSmall>::edge_descriptor` and `graph_traits<GraphLarge>::edge_descriptor`. A return value of true indicates that the edges are compatible.
*Default:* `always_compatible`

IN: `vertices_equivalent(VertexCompatibilityPredicate vertex_comp)`
This function object is used to determine if vertices between the two graphs `graph_small` and `graph_large` are compatible.
Type `VertexCompatiblePredicate` must be a model of *Binary Predicate* and have argument types of
`graph_traits<GraphSmall>::vertex_descriptor` and
`graph_traits<GraphLarge>::vertex_descriptor`. A return value of true indicates that the vertices are compatible.
*Default:* `always_compatible`

## 2.2 Functions for Isomorphism Testing

Non-named parameter, named-parameter and all default parameter versions of function

vf2_graph_iso(...)

for isomorphism testing take the same parameters as the corresponding functions `vf2_sub_graph_iso`. The algorithm finds all isomorphism mappings between graphs `graph1` and `graph2` and outputs them to `user_callback`. It continues until `user_callback` returns true or the search space has been fully explored. As before, `EdgeCompatibilityPredicate` and `VertexCompatibilityPredicate` predicates are used to test whether edges and vertices are compatible with `always_compatible` as default.

## 2.3 Utility Functions and Structs

**template** <**typename** PropertyMap1,
**typename** PropertyMap2>
property_map_compatible<PropertyMap1, PropertyMap2>
make_property_map_compatible(**const** PropertyMap1 property_map1,
**const** PropertyMap2 property_map2)

Returns a binary predicate function object (`property_map_compatible<PropertyMap1, PropertyMap2>`) that compares vertices or edges between graphs using property maps.

**struct** always_compatible

A binary function object that returns true for any pair of items.

```
template <typename Graph1,
          typename Graph2>
struct vf2_print_callback
```

Callback function object that prints out the correspondences between vertices of `Graph1` and `Graph2`. The constructor takes the two graphs $G_1$ and $G_2$ and an optional `bool` parameter as arguments. If the latter is set to `true`, the callback function will verify the mapping before outputting it to standard output.

```
// Verifies a graph (sub)graph isomorphism map
template<typename Graph1,
         typename Graph2,
         typename CorresponenceMap1To2,
         typename EdgeCompatibilityPredicate,
         typename VertexCompatibilityPredicate>
inline bool verify_vf2_sub_graph_iso(const Graph1& graph1,
                                     const Graph2& graph2,
                                     const CorresponenceMap1To2 f,
                                     EdgeCompatibilityPredicate
                                         edge_comp,
                                     VertexCompatibilityPredicate
                                         vertex_comp)

// Variant of verify_sub_graph_iso with all default parameters
template<typename Graph1,
         typename Graph2,
         typename CorresponenceMap1To2>
inline bool verify_vf2_sub_graph_iso(const Graph1& graph1,
                                     const Graph2& graph2,
                                     const CorresponenceMap1To2 f)
```

This function can be used to verify a (sub)graph isomorphism mapping *f*. The parameters are akin to function `vf2_sub_graph_iso` (`vf2_graph_iso`).

## 2.4 Complexity

Spatial and time complexity are given in [2]. The spatial complexity of VF2 is of order $O(V)$, where $V$ is the (maximum) number of vertices of the two graphs. Time complexity is $O(V^2)$ in the best case and $O(V!V)$ in the worst case.

## 2.5 A Graph Sub-Graph Isomorphism Example

In the example below, a small graph `graph1` and a larger graph `graph2` are defined.
`vf2_sub_graph_iso` computes all the mappings between the two graphs and outputs them via `callback`.

**typedef** adjacency_list<vecS, vecS, bidirectionalS> graph_type;

*// Build graph1*
**int** num_vertices1 = 8; graph_type graph1(num_vertices1);
add_edge(0, 6, graph1); add_edge(0, 7, graph1);
add_edge(1, 5, graph1); add_edge(1, 7, graph1);
add_edge(2, 4, graph1); add_edge(2, 5, graph1); add_edge(2, 6, graph1);
add_edge(3, 4, graph1);

*// Build graph2*
**int** num_vertices2 = 9; graph_type graph2(num_vertices2);
add_edge(0, 6, graph2); add_edge(0, 8, graph2);
add_edge(1, 5, graph2); add_edge(1, 7, graph2);
add_edge(2, 4, graph2); add_edge(2, 7, graph2); add_edge(2, 8, graph2);
add_edge(3, 4, graph2); add_edge(3, 5, graph2); add_edge(3, 6, graph2);

*// true instructs callback to verify a map using*
*// verify_vf2_sub_graph_iso*
vf2_print_callback<graph_type, graph_type> callback(graph1, graph2,
    **true**);

**bool** ret = vf2_sub_graph_iso(graph1, graph2, callback);

# A Testing

Also included are `vf2_sub_graph_iso_gviz_example.cpp` and a Scilab (`http://www.scilab.org/`) script `vf2_random_graphs.sce` for testing the implementation. The script generates pairs of simple graphs of (possibly) different size, such that there exists at least one (sub)graph isomorphism mapping between the two graphs. The graphs are written to files `graph_small.dot`

and `graph_large.dot` using the Graphviz *DOT* language (`http://www.graphviz.org`). The following parameters can be used to control the output:

- `nbig` Dimension of the large adjacency matrix
- `nsmall` Dimension of the small adjacency matrix
- `density` Density of the non-zero entries (of an initial square matrix with dimension `nbig`)
- `directed` If set to one, a pair of directed graphs is generated, otherwise undirected graphs are produced.
- `loops` If set to one, self-loops are allowed, otherwise self-loops are excluded.

The generated dot-files specifying the graphs can be given as command line arguments to the executable test program, which uses boost's GraphViz input parser to build the graphs. The graphs are then tested for (sub)graph isomorphism. The isomorphism mappings are verified and written to standard output.

To build the test executable, you will need to build and link against the "boost_graph" and "boost_regex" libraries, *cf.* also `read_graphviz`.

# References


[1] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "An improved algorithm for matching large graphs," *In: 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen*, pp. 149–159, 2001.

[2] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 10, pp. 1367–1372, 2004