Technical Report

Graph Isomorphism Implementation in LEDA 5.1

Author:
Johannes Singler
jsingler@algorithmic-solutions.com

Goal

The goal of this project was to augment LEDA with efficient algorithms in the field of graph isomorphism. There are the problems of testing for *graph isomorphism*, *subgraph isomorphism*, *graph monomorphism* and *graph automorphism*. However, all known algorithms are not guaranteed to run in worst case polynomial time, although graph isomorphism in particular is not even known to be NP-complete. Subgraph isomorphism and graph monomorphism *are* known to be NP-complete, the reduction of CLIQUE to either one is trivial.

All of the problems stated above can be restricted to certain domains, e. g. trees, to make more efficient algorithms possible, but we did not devote any effort to these special cases here.

Problem Definitions

Graph isomorphism is the problem of testing whether two graphs are really the same. Two simple graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ are isomorphic if and only if there exists a bijective (node) mapping $\Phi : V_G \to V_H$ for which $\forall v, w \in V_G : (v, w) \in E_G \Leftrightarrow (\Phi(v), \Phi(w)) \in E_H$ holds true. This mapping also induces a unique mapping between corresponding edges. If the graphs are labelled, the condition can be extended to preserve the node and/or edge labels as well.

Graph automorphism is equivalent to the graph isomorphism between a graph and itself. The cardinality of the automorphism is therefore related to its self-similarity.

Subgraph isomorphism restricts the mapping to a subset of one graph which has at least as many nodes as the other one. Two simple graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ are subgraph isomorphic if and only if there exists an injective (node) mapping $\Phi : V_G \to V_H$ for which $\forall v, w \in V_G : (v, w) \in E_G \Leftrightarrow (\Phi(v), \Phi(w)) \in E_H$ holds true.

Graph monomorphism is a weaker kind of subgraph isomorphism. For $\Phi : V_G \to V_H$,only the condition $\forall v, w \in V_G : (v, w) \in E_G \Rightarrow (\Phi(v), \Phi(w)) \in E_H$ must be fulfilled (implication instead of equivalence).

The *cardinality* of a graph morphism is the number of feasible mappings.

Because of the last variant which is not literally an isomorphism, we often use the term morphism to include this generalization.

Sometimes it is useful to restrict the possible node and edge mappings, i. e. some relation or characteristic function defines which nodes and edges can be matched onto each other respectively. We subsume this with the terms “node compatibility” and “edge compatibility”. An easy example would be that each node and each edge have a label and only nodes or edges that share the same label may be matched.

Algorithm Candidates

A whole lot of algorithms are studied in the literature. The following table summarizes their properties and gives some rationale for selecting two of them for implementation, namely *VF2* and *conauto*.

Algorithm	Time/Best case	Space	Comments	Subgraph Iso?	Directed?
Ullmann [12]	Slow/$\Omega(n^3)$	$O(n^3)$	classic	Yes	Possible
VF2 [1]	Fast	O(n)	straightforward	Yes	Yes
nauty [6]	Faster	$O(n^2)$/O(n)	badly documented	No	Yes
conauto [5]	Depends/$O(n^5)$	$O(n^2)$	uniform behavior	No	Yes
Valiente [13]	Very slow/$\Omega(n^3)$	$O(n^3)$	descriptive	Yes	Yes

There are several very different approaches to the problem. The VF2 algorithm takes a bottom-up approach. It tries to extend an existing mapping of nodes and edges until a full mapping is reached, starting from the empty mapping. This is equivalent to a depth-first search in the tree of all possible permutations where branches that cannot lead to a feasible solution are pruned quite early.

nauty as the very counterpart features a top-down method. A partition of the set of nodes is incrementally refined until no further refinement is possible. Nodes that remain in the same cell are then known to be equivalent (generation of the automorphism group). Based on this so-called *equitable* partition, a canonical labelling is constructed. Two graphs are isomorphic if their canonical labellings coincide.

conauto tries to take the best out of both worlds. Its overall approach resembles nauty's, but it does not try to find the full automorphism group which can be quite costly in some cases. If it is not able to detect quickly whether two nodes are really equivalent, it postpones that decision. When comparing to the second graph it checks all possible mappings using backtracking. Nevertheless, the number of mappings to check is greatly reduced by this precomputation.

While nauty performs best for graphs with high self-similarity (large automorphism group), VF2 is better for irregular graphs. In total, conauto is a good compromise and therefore has a very uniform behaviour. E. g., VF2 takes forever to find the number of automorphisms of a large clique, while nauty and conauto finish this task in a moment. However, enumerating all automorphisms is quite hopeless for all of them since the number grows exponentially in the size of the clique.

Functionality Options

There are many different possible kinds of input, output and morphism type that you might want to have supported by the library. The following table lists the imaginable possibilities for several categories and states to what extent the library supports the feature(s).

Category	Imaginable Possibilities	Supported by Implementation
Graph type	directed, undirected	directed; undirected can easily be reduced to directed by inserting two directed edges per undirected one
Morphism type	graph isomorphism, graph monomorphism, graph automorphism, subgraph isomorphism	VF2: all; conauto: graph isomorphism; graph automorphism can easily be reduced to graph isomorphism
Setting	one-to-one, one-to-many, many-to-many	all; however, even better, specialized algorithms are known for one-to-many and many-to-many
Matching	exact, inexact	exact
Attributes	inherent attributes, compatibility callback function	compatibility callback function
LEDA graph data structure for input	`leda::graph`, `leda::GRAPH<,>`, `leda::static_graph<,>`	all
Result	Boolean test (isomorphic yes/no), first found mapping, all possible mappings in a list, callback for each mapping found	all

Algorithm Implementations

The VF2 Algorithm

The VF2 algorithm can be used for all morphism types. The fastest variant runs at a much higher speed than the original implementation for the problem of graph isomorphism.

First, all variants implemented here copy the data to a static graph data structure which is capable of storing additional information needed for the algorithm during its run and also gives better performance during access than a non-static graph.

Implementation Details

Five variants of the VF2 algorithm exist which document the step-by-step improvement in the programming process. They are called `vf2-simple`, `vf2-better`, `vf2-best`, `vf2-exp` (experimenting). The latest one, vf2-exp was chosen

as the final version and therefore copied to `vf2`. It is used as default when calling a graph morphism algorithm because it is usually the fastest one. In the following, we only describe this latest variant since it contains the most differences to the original implementation.

It is mostly unchanged with respect to control flow and recursion when compared to the original description. However, several subroutines were optimized by using more sophisticated data structure. This improves performance quite a bit.

In the initial step, for both input graphs, the nodes are sorted by out-degree ascending, in-degree ascending. Using this ordering, two static graphs are constructed which are used to hold all additional node and edge information that will be needed by the algorithm. The sorting step is undocumented in [1], but nevertheless used in the original implementation and also gives considerable speedup.

The current state of the recursion must be stored somewhere, too. To minimize side effects, all the required variables are contained in an algorithm class. Only the values that change during recursion and cannot be backtracked are put onto the call stack.

As the algorithm often refers to sets of nodes only, the iteration on such a set induces an ordering which is not fixed. Hence, there may be little differences in the way different implementations run, e. g. in terms of number of recursive calls which may lead to huge differences in execution time.

Very important are the sets (of nodes) T_1^{in}, T_1^{out}, T_2^{in} and T_2^{out}. The original implementation only stores bit flags for each node which represent its membership in the respective set. This makes the iteration over all nodes of a set taking linear time in the total number of nodes instead of the number of nodes in the respective set. We use a doubly linked list to overcome this problem. Its items, namely the pointers to the predecessor node and to the successor node, are stored with each node in the node data space provided by the static graph data structure. A special head element held in the current recursion contains pointers to the first and the last element of the list. As the recursion goes on, a node that is successfully mapped must be removed from the lists it is member of. This is achieved by changing the pointers in the adjacent nodes to skip over it. However, the item itself remembers its place in the list so that it can be efficiently reinserted during backtracking. Doing so is always safe since the invariant holds that the list will be exactly in the same state as before the recursive call, including all pointers and helper structures.

Also, the algorithm is in need for the smallest element of a node set several times. The ordering is unspecified but must be fixed. Here, we take the very same as in the initial step. Although this might look like a situation to apply a priority queue, experiments showed that it does not improve computation speed. Apparently, the time to update the priority queue dominates the time to find the minimum element. Therefore, the minimum element is determined by traversing the complete respective doubly linked list.

Another subroutine is testing the feasibility of a pair of nodes n_1 and n_2 to be matched. For all edges of n_1, the original algorithm looks for a corresponding edge by searching all edges of n_2 using binary search. However, the LEDA static graph class does not support this kind of fast search. Without it, the search for the corresponding edge would take quadratic time in the number of adjacent edges in total which is inefficient. The following method is better and gives linear complexity:

For n_1, mark each adjacent node that has already been matched. Then, for n_2, check all adjacent nodes whether they have been marked. If one is not marked, the nodes do not form a feasible pair. This procedure can easily be adapted to graph monomorphism and subgraph isomorphism.

The conauto Algorithm

The conauto algorithm was implemented in two variants. The first one, called `conauto-basic`, uses high-level data structures such as linked lists to store the partition of the graph. Its whole data structure is based on the `node`, `edge`, `node_array` and `edge_array` data types from LEDA. However, it is quite slow, probably because of fine-grained memory allocation and release as well as cache-unfriendly pointer jumping. Also, it depends on the graph nodes having increasing indices without many gaps since it allocates memory proportional to `max_node_index()`.

Therefore, a second variant was built, namely `conauto-fast`. Firstly, it copies the given graphs to an internal static graph and associates a step-by-step incrementing index to each node. Thus, all subsequent data structures only contain this node index. The template parameter `ord_t`, which must be some signed integer type, defines the data type these indices are to be stored in. Memory-saving `short` is probably sufficient since the number of nodes rarely exceeds several thousand. The algorithm checks whether `ord_t` is large enough to hold the maximum node index and causes a LEDA error otherwise. Using `short` will usually limit the number of nodes to $2^{15}-1 = 32767$.

The `conauto-fast` version was chosen as the default implementation for the conauto algorithm and therefore copied to `conauto`.

Please note that the conauto algorithm only supports graph isomorphism and graph automorphism testing. It is by design not usable for either testing subgraph isomorphism or graph monomorphism.

Extensions and their Correctness

While the original algorithm and its implementation only test two input graphs for isomorphism, returning a Boolean value, the implementation here was extended so it can also compute the cardinality of the isomorphism and enumerate one or all feasible mappings.

In the original paper, there is only a weak hint about how to achieve this. Hence, we describe it here in more detail. As stated in the paper, the operation of the algorithm defines an ordering on the nodes, namely the order in which they are discarded from the current partition. There are three ways a node can be discarded. Firstly, a node is discarded after having been used as a pivot node. Then, a new cell containing only this one node is appended to a list of discarded cells. Secondly, complete cells are discarded when they are not linked to the rest of the partition any longer. In this case, the whole cell is appended to the list. Thirdly, cells might be remaining when the algorithm stops refining the partition. They are appended to the list as well. In all three cases and for both graphs, the relative order of the cells in the partition is preserved when moving the cells to the list.

Now, it is easy to generate one or all feasible mappings because cells of the same rank in the two lists for the two input graphs correspond to each other. Nodes of corresponding cells can be mapped to each other in every possible combination, yielding a cardinality factor equal to the factorial of the cell size. If only the cardinality itself is requested by the user, the total cardinality is multiplied by this factor. Otherwise, a recursive backtracking algorithm enumerates all feasible mappings.

The most important feature of the conauto algorithm, the search for automorphisms, affects this process, too. When the algorithm finds out that a couple of nodes are equivalent, a backtracking point is removed. However, this would divide the cardinality by the number of equivalent nodes. Therefore, this has to be accounted by the algorithm during generating the first sequence of partitions. Later, the product of all these factors is added to the cardinality in order to achieve the correct result. However, this is not enough when actually generating feasible mappings. In that case, backtracking must be performed anyway. Thus, in that case, the main feature of conauto is mostly useless and the execution time might increase.

Although the extensions to the algorithm were thoroughly tested and compared to the results of other algorithms, their correctness is not formally proven and therefore can not be guaranteed at this time.

General Implementation Details

Precondition

If an empty graph (no nodes) is passed as one of the inputs, the LEDA error handler is called with the message “Emtpy graph.”. Since all the graph morphism types are not defined on empty graphs, it would make no sense to start any computation. Also, the LEDA static graphs do not support empty node sets.

If an algorithm implementation does not support a specific kind of graph morphism (e. g. conauto does not support neither subgraph isomorphism nor graph

monomorphism), the LEDA error handler is called with the message “Algorithm does not support…”.

So far, no meaningful error number was assigned to these errors, we just took “1” for the number.

Precomputation

When using a graph more than once in a graph morphism testing procedure, it can be more efficient to remember the result of some computation steps that are performed for each input graph independently, for later reuse.

This is in particular time-saving for the conauto algorithm which does a great part of its work for each graph independently before finally comparing the results.

The VF2 does not benefit from such a precomputation that much. Still, however, the time for sorting the nodes and constructing the internal static graph can be saved.

All implementations here support independent precomputation for both input graphs which particularly comes in handy when one wants to test all possible pairs of a set of graphs for a certain kind of morphism. There is only one exception: For the VF2 implementations, both inputs must not be the identical prepared graph data structure, i. e. one is not allowed to calculate a graph automorphism using only one prepared graph data structure.

Bad Smells

Unfortunately, the macro `forall_out_edges` only works for either regular graphs or static graphs, depending on the header files included. Therefore, we had to define several new macros to make the program run with both graph types. `forall_adj_edges_nonstatic` is just a copy of the original which is not overwritten by the static graph version and therefore still available. `forall_out_edges_graph_t` however works dynamically for both inputs, but assumes the typename `graph_t` to be defined in the context to distinguish between the cases. The code can be found in `forall.h`:

```
#define forall_out_edges_graph_t(g,e,n) \
graph_t::node next_node = g.next_node(n); \
graph_t::edge next_edge; \
if(next_node != g.stop_node()) \
       next_edge = g.first_out_edge(next_node); \
else \
       next_edge = g.stop_edge(); \
for(e = g.first_out_edge(n); e != g.stop_edge() && e != next_edge; e = g.out_succ(e))

#define forall_in_edges_graph_t(g,e,n) \
for(e = g.first_in_edge(n); e != nil; e = g.in_succ(e))
```

In addition, we had to add two convenience functions to `static_graph.h` for compatibility:

```
// compatibility with leda::graph concerning iteration
inline edge first_out_edge(node n) const { return n->first_out_edge(); }
inline edge out_succ(edge e) const { return e->next_out_edge(); }
```

```
inline edge first_in_edge(node n) const { return n->first_in_edge(); }
inline edge in_succ(edge e) const { return e->next_in_edge(); }
```

API

Algorithm Class Hierarchy

Each algorithm implementation is encapsulated in a class. However, the actual algorithm classes are not accessible to the user directly, they must be instantiated through the templated wrapper class `graph_morphism<graph_t, impl>`.

The wrapper class implements the interface `graph_morphism_algorithm<graph_t>` which a wealth of definitions, constants and methods. An excerpt is given here:

Type definitions used in the subsequent prototypes.

```
typedef typename graph_t::node node;
typedef typename graph_t::edge edge;
typedef node_array<node, graph_t> node_morphism;
typedef edge_array<edge, graph_t> edge_morphism;
typedef leda_cmp_base<node> node_compat;
typedef leda_cmp_base<edge> edge_compat;
typedef two_tuple<node_morphism, edge_morphism> morphism;
typedef list<morphism*> morphism_list;
typedef leda_callback_base<morphism> callback;
typedef void* prep_graph;
```

Default node and edge compatibility functions: All nodes and edges are compatible with each other respectively.

```
static leda_cmp_base<node> ALL_NODES_COMPAT;
static leda_cmp_base<edge> ALL_EDGES_COMPAT;
```

Construct a prepared graph data structure for this algorithm implementation.

```
virtual prep_graph prepare_graph(const graph_t& g,
const node_compat& node_comp = ALL_NODES_COMPAT,
const edge_compat& edge_comp = ALL_EDGES_COMPAT) const = 0;
```

Delete prepared graph data structure for this algorithm implementation.

PRECONDITION: The prepared graph data structure must have been constructed by the same algorithm implementation.

```
virtual void delete_prepared_graph(prep_graph pg) const = 0;
```

Statistics: How many recursive calls were needed so far?

```
virtual cardinality_t get_num_calls() = 0;
```

Reset recursive calls counter.

```
virtual void reset_num_calls() = 0;
```

Graph morphism finding methods, described in detail below.

```
virtual bool find_iso(const graph_t& g1, const graph_t& g2,
node_morphism* _node_morph = NULL, edge_morphism* _edge_morph = NULL,
```

```
const node_compat& _node_comp = ALL_NODES_COMPAT,
const edge_compat& _edge_comp = ALL_EDGES_COMPAT) = 0;

virtual cardinality_t cardinality_iso(const graph_t& g1, const graph_t& g2,
const node_compat& _node_comp = ALL_NODES_COMPAT,
const edge_compat& _edge_comp = ALL_EDGES_COMPAT) = 0;

virtual cardinality_t find_all_iso(const graph_t& g1, const graph_t& g2,
list<morphism*>& _isomorphisms,
const node_compat& _node_comp = ALL_NODES_COMPAT,
const edge_compat& _edge_comp = ALL_EDGES_COMPAT) = 0;

virtual cardinality_t enumerate_iso(const graph_t& g1, const graph_t& g2,
leda_callback_base<morphism>& _callback,
const node_compat& _node_comp = ALL_NODES_COMPAT,
const edge_compat& _edge_comp = ALL_EDGES_COMPAT) = 0;
```

Graph morphism double-checking methods, described in detail below.

```
bool is_graph_isomorphism(const graph_t& g1, const graph_t& g2,
node_morphism const* node_morph, edge_morphism const* edge_morph = NULL,
const node_compat& node_comp = ALL_NODES_COMPAT,
const edge_compat& edge_comp = ALL_EDGES_COMPAT);

bool is_subgraph_isomorphism(const graph_t& g1, const graph_t& g2,
node_morphism const* node_morph, edge_morphism const* edge_morph = NULL,
const node_compat& node_comp = ALL_NODES_COMPAT,
const edge_compat& edge_comp = ALL_EDGES_COMPAT);

bool is_graph_monomorphism(const graph_t& g1, const graph_t& g2,
node_morphism const* node_morph, edge_morphism const* edge_morph = NULL,
const node_compat& node_comp = ALL_NODES_COMPAT,
const edge_compat& edge_comp = ALL_EDGES_COMPAT);
```

Semantics of the Main Methods

A method of the algorithm object has to be called to find out about the desired kind of morphism. There are tons of options about the parameters and the desired result. Do you only want to test for existence of a certain kind of morphism? Or do you want to get returned an actual mapping or even all mappings? Is there a restriction on the node and edge mapping?

The `find_` prefix means just testing the existence of a morphism and returning the first found feasible mapping if applicable. `cardinality_` makes the algorithm return the number of possible mappings while the `find_all_` methods append all possible mappings to a list of two-tuples of a node array and an edge array. `enumerate_` is a variant that calls a user-defined callback function each time when a morphism is discovered.

The methods for subgraph isomorphism and graph monomorphism are omitted in the listing. Just replace `iso` by `sub` and `mono` respectively (2 more variants per method). There are no explicit calls for graph automorphism. Just pass the same graph twice to the desired `iso` routine.

Basic Input and Output

All methods take two graphs as their first two parameters. For subgraph isomorphism and graph monomorphism, the first graph is the one of which a subgraph is determined in order to map the second graph onto it. So to gain a positive result, the second graph must contain no more nodes and no more edges than in the first one. Otherwise, no mapping is possible anyway.

Each input graph can be passed as a `graph_t` which in is substitutable by `graph`, `GRAPH<,>` or `static_graph<,>`, but also as a precomputed graph morphism data structure (specific for the used algorithm), too. These 3 more variants per method have been omitted in the above interface listing for clarity.

The returned morphism mapping for nodes and edges is stored in a node and an edge array respectively which must be parameterized with the node reference type of the original graph which is passed first, and initialized with the original graph which is passed second. If this additional information is not needed by the caller, a NULL pointer may be passed to prevent its computation. Otherwise, in case a morphism is found, for each node and edge of the second graph, the corresponding one in the first graph is referenced.

All methods except the ones the ones only seeking to find the first morphism, return the cardinality of the found morphism. Since this number can be very large even for small graphs, it is passed in a LEDA `integer` which can handle arbitrary long integer numbers. However, this behavior can be changed by defining `cardinality_t` differently, e. g. to `long`.

In total, there are 48 different methods provided for finding all kinds of graph morphisms with all kinds of inputs and outputs.

Callback Mechanism

When using the `enumerate` call of the algorithm, the passed function will be called back each time a morphism of the desired kind is found. It will be passed a node and an edge array with the found mapping from the second to the first graph, as before. Whether the algorithm continues to look for further mappings depends on the Boolean value returned by this callback function. `true` makes the search terminate instantly. Once more, the design pattern resembles the `leda_cmp_base<>` class. However, in this case, the underlying class is called `leda_callback_base`. The functionality can be incorporated by either passing a C-style function pointer or deriving from the class and overriding the `()`-operator.

Node and Edge Compatibility

The last two parameters most methods deal with node and edge compatibility. If no compatibility functions are passed by the user, all nodes and edges are considered compatible with each other respectively. Technically, this is achieved by passing the predefined functors `ALL_NODES_COMP` and `ALL_EDGES_COMP` by default.

The callback mechanism for checking compatibility of nodes and edges uses the `leda_cmp_base<>` approach whereby additional semantics may be required. Either a functor class derived from `leda_cmp_base<>` or a simple function pointer can be used. In the latter case, the function pointer will be encapsulated by an instance of `leda_cmp_base<>` which is constructed through implicit conversion. The comparison function takes two references to the according node types.

For the VF2 variants, it only is supposed to return 0 for the both nodes being compatible and a non-zero value otherwise. In this case, the node of the first graph is always passed as the first parameter. This is important in case the relation is asymmetric with respect to the graphs.

For the conauto variants, additional properties are claimed. The compatibility function is not only to be an equivalence relation on the set of both graph's nodes, but also has to define a total ordering of non-equivalent ones.

Sample code is given here which bases node compatibility on the standard order of the node data given by the `compare` function.

```
template<typename T>
class identity_compatibility: public leda_cmp_base<node>
{
private:
	typedef node_map<T> node_map;
	const node_map& info1, info2;

public:
	identity_compatibility(const node_map& _info1, node_map& _info2) :
info1(_info1), info2(_info2)
	{
	}

	virtual int operator()(const node& n1, const node& n2) const
	{
		const T* i1, * i2;
		if(info1.get_graph().member(n1))
			i1 = &info1[n1];
		else
			i1 = &info2[n1];

		if(info1.get_graph().member(n2))
			i2 = &info1[n2];
		else
			i2 = &info2[n2];

		return compare(*i1, *i2);
	}
};
```

Double-checking Correctness

For each kind of morphism, methods are provided which double-check the correctness of a mapping. Those start with `is_` and take the mappings for both nodes and edges (optional) as a parameter. Actually, the implementation is not

optimized for performance and also does not depend on the algorithm object. It does not change the state of the algorithm object and is non-static for convenience only. The methods return `true` if the mapping is a morphism of the desired kind and `false` otherwise.

Precomputation

To benefit from precomputed internal data structures, just call the `prepare_graph` method of the respective algorithm object. The returned `prep_graph` value (a anonymous pointer) can be passed instead of the original graph in subsequent isomorphism calls. However, you must always pass an internal structure of the algorithm implementation that you are using to perform the actual computation. The data structures must be destroyed using `delete_prepared_graph` to avoid a memory leak.

Both methods do not change the state of the algorithm object and are non-static for convenience only.

Statistics

The method `get_num_calls()` provides some statistical information, namely the number of recursive calls the algorithm had to perform to solve the problem. The counter must have been reset before by calling `reset_num_calls()`. Since counting degrades performance slightly, it is switched off by a compiler switch by default. You must define `CALL_COUNT` in `graph_morphism_algorithm.h` to make it work.

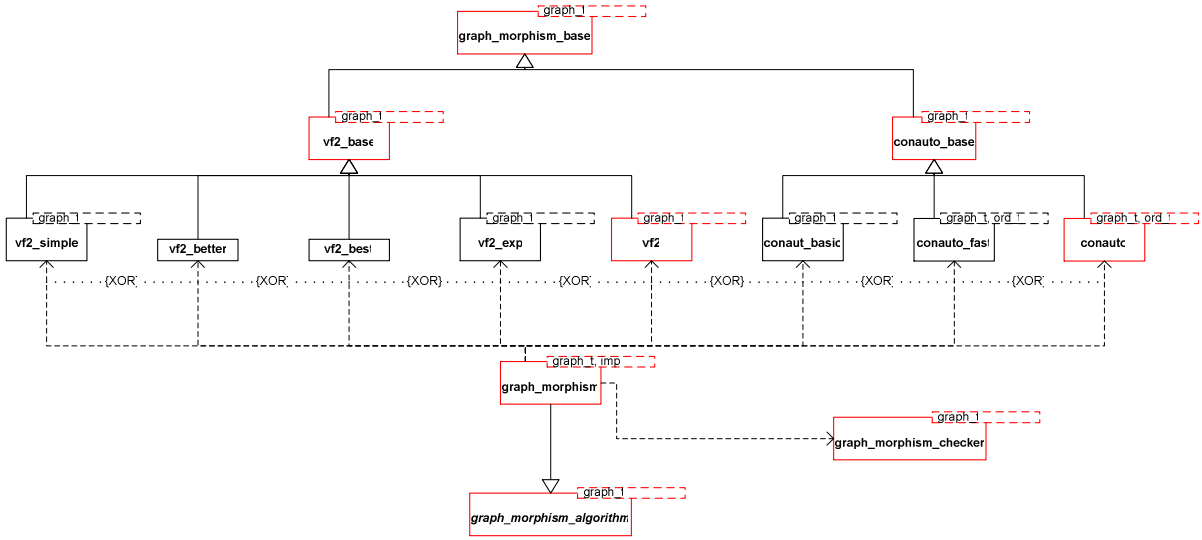
Available Implementations

As mentioned above, the user must instantiate `graph_morphism<impl>` with an optional implementation parameter first, and then call the desired method(s) to run a computation. Since all classes are pretty light-weight, you do not have to worry about the costs of constructing and deleting them.

For VF2, valid implementation parameters are `vf2<graph_t>`, `vf2_simple<graph_t>`, `vf2_better`, `vf2_best`, and `vf2_exp<graph_t>`. They have evolved during different stages of optimization. The default is `vf2` which is at this time equivalent to `vf2_exp`. Those implementations except `vf2` may not support all the advanced features like subgraph isomorphism testing or node and edge compatibility. Particularly, `vf2_better` and `vf2_best` only support `graph` as input. A compile time or LEDA error occurs if unsupported features are requested.

For conauto, there exists `conauto_basic<graph_t>` and `conauto_fast<graph_t, ord_t>` which is equivalent to `conauto<graph_t, ord_t>`. Please note that the conauto implementation of node compatibility is more

restricted compared to the one provided by the VF2 variants. Also, conauto does not support edge compatibility at all.



The VF2 variants derive from `vf2_base<graph_t>`, the conauto variants derive from `conauto_base<graph_t>`. Both these base classes on their part derive from `graph_morphism_base<graph_t>`. They are all parameterized with the type of the input graphs. Please note that it is not possible to mix different types of input graphs.

Sample Code

The following code tests two graphs for isomorphism using the conauto algorithm, whereby for the first graph a precomputed data structure is used and also node compatibility is required. The steps in brackets [] can be omitted when not using precomputation or node compatibility.

1. Include header file.

```
#include <LEDA/graph/graph_morphism.h>
```

2. Declare the input graphs.

```
graph g1, g2;
```

3. In order to use node compatibility, declare associated node maps for the attributes and a corresponding node compatibility function (exemplary, see above for the definition of `identity_compatibility`) .

```
node_map<int> nm1(g1), nm2(g2);
identity_compatibility<int> ic(nm1, nm2);
```

4. Do something useful to build up the graphs and the attributes.

```
/* build up graphs... */
```

5. Instantiate the algorithm object.

```
graph_morphism<graph, conauto<graph> > alg;
```

6. Declare the node and edge mapping arrays.

```
node_array<node> node_mapping(g2);
edge_array<edge> edge_mapping(g2);
```

7. Prepare a graph morphism data structure for the first graph.

```
graph_morphism_algorithm<>::prep_graph pg1 = alg.prepare_graph(g1, ic);
```

8. Find the graph isomorphism.

```
bool isomorphic = alg.find_iso(pg1, g2, &node_mapping, &edge_mapping, ic);
```

9. Delete the prepared graph data structure again.

```
alg.delete_prepared_graph(pg1);
```

The shortest way to just test for isomorphism without using any of the advanced features is this one-liner:

```
bool isomorphic = graph_morphism<>().find_iso(g1, g2);
```

Testing Environment

We implemented a standalone program which tests the functionality of this library, double-checks the result and measures execution time for the own implementation as well as for earlier implementations of algorithms for this problem like the original VF2 implementation [3], Valiente's algorithms [13] and the nauty package [8]. Input can be read from files in the LEDA .gw format[1] and files from the Graph Database [2]. The program does not feature any GUI but is controlled entirely through the command line.

The program takes a number of options for the algorithm(s) to use and the morphism to search for, and either one or two file names or a directory path. If two file names are passed to the program, two graphs are read from them and the desired morphism is searched for. If only one file name is given, the program tries to

[1] However, only the own implementations support this input file type.

reconstruct the second one by the rules of the graph database (replacing the `A` in the extension by a `B`). If a directory is passed, it is assumed to be a part of the graph database. All corresponding graphs in the directory and all subdirectories are matched against each other.

The external algorithms are included through static libraries built by independent projects. They are called VF2, Conauto, Nauty, Valiente.

Option	Effect
-iso|-sub| -mono | -auto	search for graph isomorphism, subgraph isomorphism or graph monomorphism, defaults to -iso
(-[alg])*	algorithm(s) to use, multiple options of this kind allowed, e. g. -conauto-fast -vf2; the external ones are: nauty, vf2-orig, vf2-orig-nosort, valiente, conauto-orig
-map	show all found mappings using node indices
-files	show the names of the processed files
-details	show timing results for each graph size separately
-num_calls	show the number of recursive calls for each algorithm (only in combination with –details)
-first | -card | -all | -enum	check for isomorphism only / determine cardinality of the morphism / compute all feasible mappings and compare them between the algorithms / call a callback function for each discovered morphism; defaults to -all
-ncomp	check the nodes for compatibility[2]
-ecomp	check the edges for compatibility
-micro m	microcensus mode: only process m first graphs/graph pairs for each size (database)
-max n	only consider graphs with at most n nodes
-stress r	run each test r times to test the correctness and efficiency of precomputation

[2] The graphs must be given through .gw files that contain a string associated with every node or edge respectively. For VF2, two nodes or edges are compatible if the string of the second one contains the string of the first one. For conauto, two nodes are compatible if they share the same label.

-exp c	show a warning message if the morphism cardinality does not match the expected number c
-crosscheck	Test all pairs of files in a folder for the desired morphism. This option can be combined with –micro to avoid long execution times.
-cache	Keep precomputed graph morphism data structures in memory for all graphs of the folder and reuse them whenever possible. Only practical in combination with –micro because of the large memory requirements. Particularly useful with –crosscheck and/or –stress.

Example: `GraphIsomorphisms -vf2 –conauto -details iso\` in directory `GraphDatabase\graphs\`.

Compile Time Options

By defining the symbol `USE_STATIC_GRAPHS` in `interface.cpp` to a static graph implementation, e. g. `opposite_graph`, the program is forced to use static graphs for the algorithms' inputs und thus to test the library with static graphs as template parameters. The number defined for `PREP_INPUT` controls the precomputation. If it is equal to `0`, no precomputation is performed, defining it to `1` means preparing the first graph, defining to `2` means preparing only the second graph. Setting `PREP_INPUT` to `3` forces preparation for both graphs and also enables the –cache command line option (see above).

Correctness Tests

We run a lot of tests on the Graph Database to test the correctness of the algorithms. After diverse algorithms have solved the same problem, the results (including the mappings) are firstly checked for plausibility and the tested for equivalency among the results from the different algorithms.

You can find a reasonable set of test `RegressionTest.bat`, a log file of its execution is appended to this report.

Demo Program

A demo program named "gw_isomorphism" features the GraphWin GUI. Graphs can be loaded or constructed graphically. The framework has been extended by a function to load and save files in the GraphDatabase format. However, the integration of the corresponding menu items is quite a hack; it uses submenu indices that may change in the future.

Space Complexity

Although no tight bounds for time complexity can be stated, space complexity is quite easy.

`vf2`'s heap memory requirement is linear in the number of nodes *n* plus the number of edges *m*, i. e. O(n+m). In addition, the implementation needs linear stack space, O(n). Experiments showed that 156 bytes of stack per node (of the smaller graph) are required when compiled using the Microsoft Visual C 7.1 compiler with debug information as well as stack checking disabled.

`conauto` uses much more memory. The sequences of partitions and the adjacency matrix consume an amount of memory proportional to the squared number of nodes, i. e. $O(n^2)$. In the worst case, it also consumes a linear amount of stack space, under same conditions as before 36 bytes per node.

Benchmarks

Setup

All benchmarks were run on a Windows-XP based PC equipped with a 2GHz AMD Athlon 64 processor and 1 GB of RAM. Only the net time for actually executing the algorithm was accounted for, excluding the time for loading and setting up the graphs.

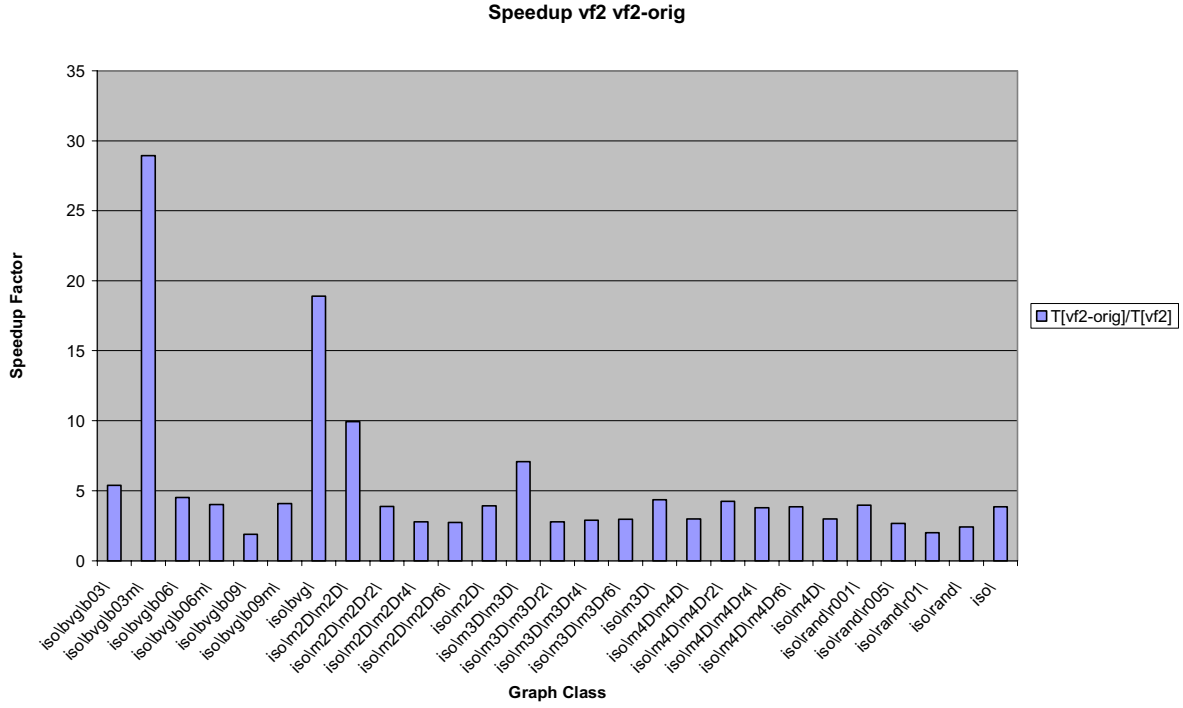
Results

VF2 Family

The tables and diagrams show the speedup of the own VF2 implementation against the original one. The test covered all of the graph isomorphism examples in the GraphDatabase. The algorithms had to compute the cardinality of the isomorphism, which for VF2 effectively means enumerating all possibilities, but not storing them. All values are times measured in seconds.

	vf2	vf2-orig	vf2 speedup
iso\bvg\b03\	0,535403	2,876783	5,37311707
iso\bvg\b03m\	8,304005	240,175857	28,9228941
iso\bvg\b06\	0,603268	2,729346	4,52426782
iso\bvg\b06m\	0,783883	3,154874	4,0246746
iso\bvg\b09\	1,953448	3,696382	1,89223465
iso\bvg\b09m\	1,522252	6,223396	4,08828236
iso\bvg\	13,702259	258,856638	18,8915301
iso\m2D\m2D\	0,489611	4,864756	9,93596141
iso\m2D\m2Dr2\	0,908287	3,524034	3,87986837
iso\m2D\m2Dr4\	1,194296	3,311428	2,77270291
iso\m2D\m2Dr6\	1,270361	3,471949	2,73304124
iso\m2D\	3,862555	15,172167	3,92801319
iso\m3D\m3D\	2,023119	14,288289	7,06250547
iso\m3D\m3Dr2\	1,126221	3,126599	2,77618602
iso\m3D\m3Dr4\	1,235899	3,578689	2,89561607
iso\m3D\m3Dr6\	1,347986	3,998963	2,96662057
iso\m3D\	5,733225	24,992541	4,3592465

iso\m4D\m4D\	209,251542	623,633003	2,98030302
iso\m4D\m4Dr2\	0,69477	2,947433	4,24231472
iso\m4D\m4Dr4\	0,89633	3,38504	3,77655551
iso\m4D\m4Dr6\	1,04443	4,035	3,8633513
iso\m4D\	211,887073	634,000475	2,99216213
iso\rand\r001\	2,271437	9,001015	3,9626963
iso\rand\r005\	6,416066	17,109782	2,66670916
iso\rand\r01\	12,165394	24,43972	2,00895425
iso\rand\	20,852897	50,550517	2,4241484
iso\	256,038008	983,572339	3,84150911
geometric mean			4,1122011



We can conclude that our implementation is about 4 times as fast as the original one for the problem of graph isomorphism. Further test showed that the results are similar for the individual problem sizes. In particular, we were able to eliminate the exceptionally bad execution times of the original implementation for large graphs of the b03m type (modified 3-valent).

For subgraph isomorphism, our implementation is only slightly faster in total.

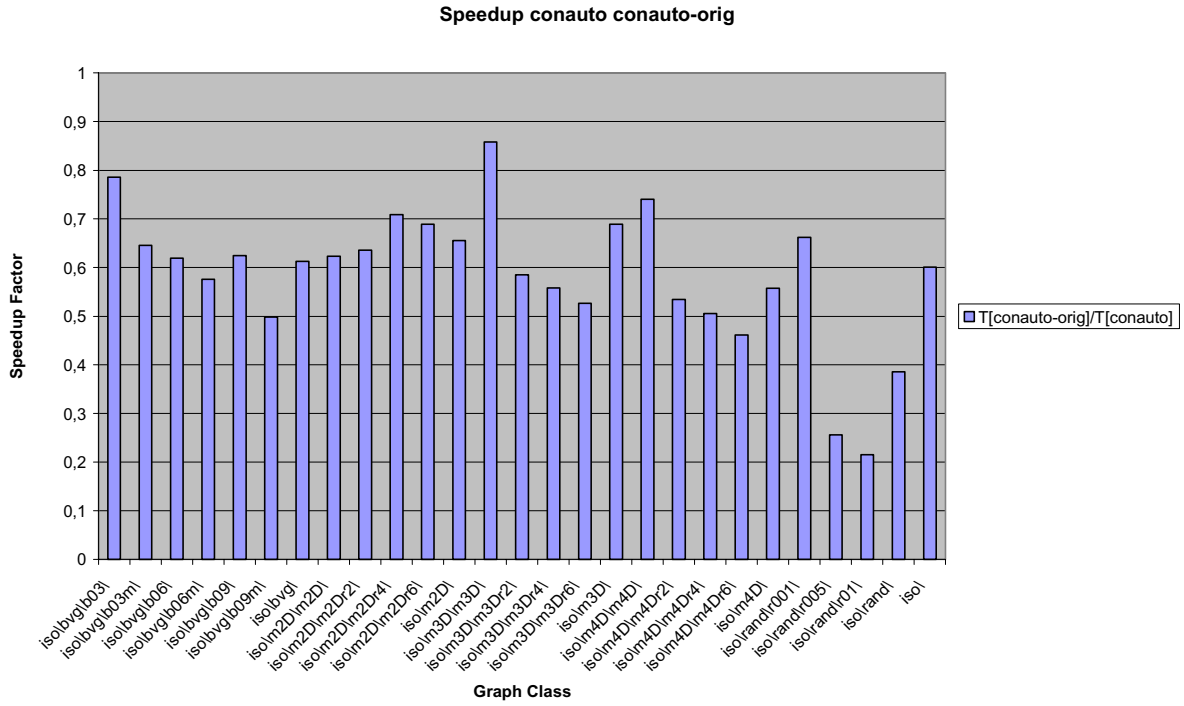
Even worse, for graph monomorphism, singular examples from the GraphDatabase (from the class random graphs with 10% edge probability) make our implementation take very long time to finish. This behavior disappears when abandoning the sorting by degree in the initial step which improves all other cases significantly. Apparently, there must be some examples in the database that represent a worst case for specifically our implementation. The implementation vf2_exp contains a special

sorting routine where nodes are clustered by the same degrees and the clusters are then sorted by size. This eases this case, but makes graph isomorphism taking much longer time on the opposite. Because of the time requirement, we only could test with instances of small graphs. It looks like moving to larger graphs make all algorithms take an impossibly long time anyway.

conauto Family

Testing was performed similarly to the VF2 case. The only difference was that the algorithms had to *test* for isomorphism only. This is because the original implementation can neither count nor enumerate the possible mappings.

	conauto	conauto-orig	conauto speedup
iso\bvg\b03\	10,974554	8,623087	0,78573462
iso\bvg\b03m\	14,412126	9,297717	0,64513154
iso\bvg\b06\	19,071365	11,812264	0,61937171
iso\bvg\b06m\	19,760526	11,376455	0,5757162
iso\bvg\b09\	19,095762	11,92287	0,62437257
iso\bvg\b09m\	17,687889	8,810201	0,49809228
iso\bvg\	101,002222	61,842594	0,61228944
iso\m2D\m2D\	54,457727	33,95911	0,62358662
iso\m2D\m2Dr2\	21,970786	13,968351	0,63576929
iso\m2D\m2Dr4\	23,299137	16,519131	0,70900184
iso\m2D\m2Dr6\	25,535138	17,598253	0,68917791
iso\m2D\	125,262787	82,044845	0,65498179
iso\m3D\m3D\	47,869859	41,082517	0,85821262
iso\m3D\m3Dr2\	18,396495	10,758553	0,58481537
iso\m3D\m3Dr4\	19,827667	11,070814	0,55835182
iso\m3D\m3Dr6\	22,253122	11,713282	0,52636578
iso\m3D\	108,347144	74,625166	0,68875988
iso\m4D\m4D\	39,786547	29,458302	0,74040861
iso\m4D\m4Dr2\	35,369033	18,89187	0,53413589
iso\m4D\m4Dr4\	41,524554	20,999195	0,50570549
iso\m4D\m4Dr6\	44,928156	20,722194	0,46122957
iso\m4D\	161,60829	90,07156	0,55734492
iso\rand\r001\	17,065507	11,302994	0,66232981
iso\rand\r005\	14,589887	3,738765	0,2562573
iso\rand\r01\	16,620824	3,582802	0,21556103
iso\rand\	48,276218	18,624561	0,38579163
iso\	544,496661	327,208726	0,60093798
geometric mean			0,56410127



Apparently, our implementation is slower than the original one, approximately half as fast. This is probably due to abstaining from bit vector manipulation and also to virtual function calls to comparator functors during sorting.

The performance degrades even more when asking for an enumeration of all possible mappings. But this is not very astonishing, and the original implementation is not capable of doing that at all.

Directory and File Structure of the Project

The implementation was done using Microsoft Visual Studio.net 2003 (7.1). A project called "GraphIsomorphisms" references all necessary files. A CVS repository was used as well and can be helpful when having to reproduce the development process. The files are located in the CVS directory.

In fact, the project folder contains more files than absolutely necessary for the project. This is because several files became obsolete during the process but should not be lost from the CVS. Also, the folder contains several files that form the test program and are not needed for the library functionality itself. Therefore, the *solution* "GraphIsomorphisms" contains two projects, GraphMorphismTest and GraphMorphismLibrary. The latter includes only the files that implement the raw functionality, without test program. However, for the final version to be integrated into LEDA, one would probably discard the premature implementations of the algorithms, and stick with the final ones `vf2` and `conauto` only.

The following table describes the purpose of each file.

File Name	Description	Obsolete	Testing Env.	Library	Library Final
BinaryGraphReader.h BinaryGraphReader.cpp	reads in the GraphDatabase format		*		
GraphIsomorphisms.cpp	main test program		*		
graph_isomorphism.h	renamed to graph_morphism.h	*			
graph_morphism.h	main header file			*	*
graph_morphism_algorithm.h	main interface			*	*
graph_morphism_checker.h	correctness checkers			*	*
interface.cpp interface.h	wrapper for the own library and external libraries		*		
IOException.h	helper class for testing		*		
PerfTimer.cpp PerfTimer.h	fine-grained timer for Windows		*		
_bounded_ordered_partition.h	helper class for conauto/conauto-fast			*	*
_bounded_ordered_partition.cpp	empty file	*			
_callback_base.h	interface for the callback functionality			*	*
_compose.h	composition class (derives from two classes)			*	*
_conauto.h	conauto final implementation			*	*
_conauto_base.h _conauto_base.cpp	base class for all conauto implementations			*	*
_conauto_basic.h _conauto_basic.cpp	conauto-basic implementation			*	
_conauto_fast.cpp	empty file	*			
_conauto_fast.h	conauto_fast implementation			*	
_conauto_types.h	content moved to _ext_adjacency_matrix.h	*			
_equiv_base.h	class replaced by	*			

File Name	Description	Obsolete	Testing Env.	Library	Library Final
	leda_cmp_base<>				
_ext_adjacency_matrix.h _ext_adjacency_matrix.cpp	helper class for conauto_basic			*	
_ext_bounded_adjacency_matrix.h	helper class for conauto/conauto-fast			*	*
_ext_bounded_adjacency_matrix.cpp	empty file	*			
_forall.h	extended graph iteration			*	*
_graph_isomorphism.cpp	old API	*			
_graph_morphism_base.h	base class for all algorithms			*	*
_ll_item.h	helper class for vf2_better, vf2_best			*	
_morphism_base.h _morphism_base.cpp	renamed to graph_morphism_base.*	*			
_node_comparator.h	helper class for all VF2 algorithms			*	*
_node_disjoint_set.h	helper class for conauto-basic			*	
_node_disjoint_set.cpp	empty file	*			
_node_ordered_partition.h	helper class for conauto-basic			*	
_node_ordered_partition.cpp	empty file	*			
_vf2.h	final VF2 implementation			*	*
_vf2.cpp	empty file	*			
_vf2_base.h	base class for all VF2 implementations			*	*
_vf2_best.cpp _vf2_best.h	vf2-best implementation			*	
_vf2_better.cpp _vf2_better.h	vf2-better implementation			*	
_vf2_exp.h	vf2-exp implementation			*	

File Name	Description	Obsolete	Testing Env.	Library	Library Final
_vf2_exp.cpp	empty file	*			
_vf2_simple.h	vf2-simple implementation			*	
_vf2_simple.cpp	empty file	*			
GraphMorphismLibrary.vcproj	library build project			*	*
GraphMorphismTest.vcproj	test project				

All files that are not supposed to be included by the user are prefixed with an underscore.

Compiler Compatibility

The library was developed using the Microsoft Visual C 7.1 compiler. The final version was also adapted to compile with GCC 3.4. Thus, the code is likely to be fully ISO-C++ compliant.

Source Code Documentation

In order to be able to generate documentation files automatically using Lman, some machine processable comments were added to `graph_morphism_algorithm.h` and `graph_morphism.h`.

Possible Improvements

- Extend the algorithms to non-simple graphs. Work-around: Use edge compatibility functions which compare multiplicity.
- Partition the graphs into connected components first and run the algorithm for pairs of these whereupon each of the two belongs to a different graph.
- Avoid copying the input graphs in `vf2` which takes considerable time. However, one must still sort the nodes. Would it be allowed to change the order of the nodes of input graph?
- Speed up the conauto implementation by avoiding the LEDA sorting routines and therefrom the comparison functors which involve virtual function calls.
- Implement another improvement found in the original implementation but not documented in the paper: Build the initial sequence of partitions for both graphs (by precomputation where applicable) and estimate the backtracking

complexity for each of them. Then, choose the graph with less complexity as the first graph.

- Prove the correctness of the conauto extensions, i. e. the calculation of the cardinality, the generation of mappings, and the support of node compatibility.
- Combine any graph isomorphism algorithm implementations in the following way: All algorithms start solving the problem in parallel, for example through (user-level) multi-threading. Then stop the whole computation when one of them is finished. This approach could avoid the possibly extremely long execution time for some algorithm while preserving its good performance in the easy cases, thereby reducing the average performance only by a constant factor. However, there is the problem of implementing this platform-independently.
- Easy variant of the latter: Integrate the fast initial check for compatibility by conauto into the VF2 implementation. This may rule out non-isomorphic graphs quickly, but those are simple cases anyway.
- Add a reimplementation of nauty. A possible improvement could be the usage of a generalized degree as in conauto (0 = not connected, 1 = connected incoming, 2 = connected outgoing, 3 = connected both ways).
- Test the performance of all algorithms on non-isomorphic, but "similar" graphs. Since such hard examples are difficult to construct, most scientific literature avoid this topic, too. However, the conauto inventor appears to be interested in those [5].
- Improve the VF2 performance for graph monomorphism. First one would have to analyze why certain random graphs make the algorithm take such a long time and why this does not apply to the original implementation.

References

[1] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In Proceedings of the 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Ischia (Italy), May 2001.

[2] P. Foggia, C. Sansone, and M.Vento. A database of graphs for isomorphism and sub-graph isomorphism benchmarking. In Proceedings of the 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Ischia (Italy), May 2001.

[3] P. Foggia, C. Sansone (carlosan@unina.it), and M. Vento. A performance comparison of five algorithms for graph isomorphism. In Proceedings of the 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Ischia (Italy), May 2001.

[4] Martin Fürer. A counterexample in graph isomorphism testing. Tech. Rep. CS-87-36, Department of Computer Science, The Pennsylvania State University, University Park, Penna., 1987.

[5] José Luis López-Presa (jllopez@diatel.upm.es), Antonio Fernández. Graph Isomorphism Testing Without Full Automorphism Group Computation. Informes Técnicos de miembros del GSyC, Volume IV (2004), No 3 (TR-GSYC-2004-3).

[6] Brendan D. McKay. Practical graph isomorphism. Congressus Numerantium, 30:45-87, 1981.

[7] Brendan D. McKay. nauty user's guide (version 1.5). Technical report, Computer Science Department, Australian National University, 1990.

[8] Brendan D. McKay. The nauty page, March 2004. http://cs.anu.edu.au/~bdm/nauty/.

[9] Sven Reichard. Strongly regular graphs, May 2000. http://www.math.udel.edu/~reichard/srg_new/index.html.

[10] Douglas C. Schmidt, Larry E. Druffel. A Fast Backtracking Algorithm to Test Directed Graphs for Isomorphism Using Distance Matrices. Journal of the ACM, Volume 23, No 3, July 1976, pp. 433-445.

[11] SIVALab. The graph database, May 2003. http://amalfi.dis.unina.it/graph.

[12] J. R. Ullmann. An algorithm for subgraph isomorphism. Journal of the ACM, 23(1):31-42, January 1976.

[13] Gabriel Valiente. Algorithms on Trees and Graphs. Springer-Verlag 2002. ISBN 3-540-43550-6.