Adapting VAUCANSON algorithms to a simple AUTOMATA interface

Florent D'Halluin

Technical Report n^o0914, January 2010
revision 2086

VAUCANSON is an extensive C++ library for the manipulation of finite state machines. User feedback shows that VAUCANSON is slow and that the library interface for direct automaton manipulation is complex. To enhance VAUCANSON, the development team is making major changes to simplify the interface over the AUTOMATA data structure and reinstate a sane modeling for the underlying implementation. A consequence of these changes is that the algorithms available in the VAUCANSON library must be adapted to the new interface.

Adapting algorithms is an opportunity to study the impact of the recent interface and implementation changes on performance and accessibility. Because the new interface over automata is simpler, possible optimizations are more apparent.

VAUCANSON est une bibliothèque C++ de manipulation d'automates finis. Le feedback utilisateur montre que VAUCANSON est lent et que l'interface de la bibliothèque qui permet de manipuler les automates directement est complexe. Pour améliorer VAUCANSON, l'équipe de développement met en place une interface simplifiée sur la structure de données AUTOMATA et réinstaure une modélisation saine de l'implémentation sous-jacente. Une conséquence de ces changements est que les algorithmes disponibles dans VAUCANSON doivent être adaptés à la nouvelle interface.

L'adaptation de ces algorithmes donne l'opportunité d'étudier l'impact des changements récents sur les performances et l'accessibilité de VAUCANSON. Grâce à la simplicité de la nouvelle interface, les optimisations possibles deviennent plus visibles.

Keywords
Vaucanson, C++, automata, performance, accessibility, interface, optimization

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
d-halluin@lrde.epita.fr – http://publis.lrde.epita.fr/201001-Seminar-DHalluin

Copying this document

Contents

Introduction

Compared to other finite state machine manipulation libraries such as OPENFST (Allauzen et~al., 2007), VAUCANSON (Lombardy et~al., 2004) has important performance issues and a fairly complex interface. To enhance VAUCANSON, major changes were designed early 2009: The generic but heavy data structure used to store automata would be declined in several versions, referred to as *kinds* (Galtier, 2010), an old design that never was implemented. Additionally, the AUTOMATA interface would be reworked.

Since August 2009, the developement of VAUCANSON is focusing on rewriting a new AUTOMATA interface, implementing the LAL *kind* (see Glossary), and adapting algorithms the the new interface.

As of January 6th 2010, the work on this project is not complete. The preliminary results show a large performance gain, but only part of the library is functional, and much still has to be done.

This reports presents the most visible part of the major changes in VAUCANSON: the rewriting of algorithms and the benchmarking results obtained so far. Deep changes in the library are presented in a parallel technical report by Jérôme Galtier (Galtier, 2010).

Since the rewriting of all algorithms in VAUCANSON will be not be complete before a new team takes over the development, much of this report is written as a guide to help the new team continue the development. Because VAUCANSON is currently in an unstable state between versions 1.3 and 2.0, the information presented in this report may become outdated quickly, although the general organization of the algorithms and benchmarks should remain the same.

Part 1 describes how algorithms are written in regards to the ELEMENT design pattern and how to write specialized version of an algorithm for a specific kind of automaton.

Part 2 lists the available opportunities to improve algorithm performance by rewriting part of their code and gives examples for `eval`, `product`, and `determinize`.

Part 3 describes the benchmarking process in VAUCANSON, some considerations about the compilation process, and the results obtained so far in terms of performance.

This report is based on the assumption that the reader is familiar with basic automata theory, C++ programming, and VAUCANSON. The work presented here is previously introduced in technical reports from May 2009 (Galtier, 2009) (D'Halluin, 2009b). Some terms are used in a specific manner throughout this report. They are listed separately at the end of the report, in the Glossary.

Part 1

Algorithm specialization and the AUTOMATA interface

This part presents the ELEMENT design pattern from a practical point of view, as it is used in VAUCANSON, then explains how algorithms are written first in their generic form, then on a specific kind of automaton.

Section 1.1 describes the ELEMENT design pattern from a practical point of view.

Section 1.2 shows how generic algorithms are written in VAUCANSON.

Section 1.3 explains how to write specialized version of an algorithm on a given kind.

The need to rewrite algorithms is a consequence of the major changes in VAUCANSON on the AUTOMATA interface and on the implementation of the *kinds* (Galtier, 2010). The interface changes cause simple modifications of the algorithms, mostly replacing calls to the old functions by calls to the new functions. The implementation of the *kinds*, however, often require bigger changes: in LEGACY (VAUCANSON 1.3, see Glossary), algorithms are generic templated code written only once. Static and dynamic preconditions filter the input automata with respect to the properties they must have. With the *kinds*, each algorithm can have a unique, specific implementation for each *kind*. Some algorithms, such as `eval`, `product` and `determinize`, are only defined on certain *kinds* (LAL), while others (`accessible`) have one generic version that works with all the *kinds*.

VAUCANSON contains 98 functions on automata, including variations of the same core algorithm. Few core algorithms are currently rewritten:

- `eval` (LAL)
- `product` (LAL)
- `determinize` (LAL)
- `realtime` (LAL version only)
- `accessible` (All *kinds*)

Because the major changes in VAUCANSON will not be complete before the development team changes, this chapter is written to serve as a guide for the rewriting of the remaining algorithms. The examples are taken directly from VAUCANSON an illustrate some of the situations encountered.

1.1 The ELEMENT design pattern in VAUCANSON

ELEMENT is a design pattern that separates an object into two entities:

- **Structure:** an instance of a class or type that defines the *element*'s interface and how it can be manipulated. The structure of an *element* is instanciated at run time and can hold dynamic information.
- **Value or implementation:** the data container for the *element*. It contains the object's content or values.

Simply put, an element can be seen as an instance of an object of dynamic type: its structure defines the type, therefore two elements with the same structure have the same dynamic type; its value, or implementation, is the data that is modified in algorithms.

ELEMENT is central to the design of VAUCANSON because in many places there is a need for dynamic type structures. Within algorithms, the most used *element* contains automata: Each automaton is defined on a specific alphabet, which contains a finite set of letters (from `"ab"` to a full ASCII or Unicode set). The alphabet is part of a *monoid*, which, associated with a *semiring*, forms a *series*. A *series* defines the operations on the inner components of an automaton: *letters*, *words*, and *weights*. A series is completed by a *kind* to define the structure of an automaton. *Kinds* describe the type of data that is stored on transitions, from single letters to rational expressions. Automata of different *kinds* have different properties (Galtier, 2009).

An automaton cannot be created without a specific *series* and *kind*, and since all possible alphabets (each defining a different *series*) cannot be statically instanciated during compilation, there is a need for dynamic type structures.

All automata in VAUCANSON are *elements*. Their structure inheritates from `AutomataBase` and their implementation from `AutomataImplBase`. Through static inheritance, more precise types are defined. Within algorithms, the type of implementation does not matter, and in practice, automata are typed as shown in Listing 1.1.

Listing 1.1 – Type definition of an automaton for use in algorithms.

```
1 template<typename Series, typename Kind, typename AutomataImpl>
2 typedef Element<Automata<Series, Kind>, AutomataImpl> automaton_t;
```

1.2 Using templates to write the generic version of an algorithm

There is currently no new algorithm to add to VAUCANSON. However, the current algorithms have to be rewritten to fit with the new interface and *kind* system. All algorithms are composed of two parts:

- **A front-end function** that checks types and preconditions, instanciates empty data structures for storing results, then starts a task in the benchmarking system and calls the algorithm core. This is the function listed in the library's interface, and its prototype must be precisely defined.
- **The algorithm core**, a function or functor that makes all computations and constructions. It may be called by several front-end functions and may assume all parameters correct.

The rewriting process can be divided into 5 steps:

- Clarify the interface.
- Check for preconditions in the front-end function.
- Adapt the algorithm core to the new interface.
- Reorganize and optimize the core of the algorithm.
- Test and benchmark the algorithm.

All algorithms are located in `include/vaucanson/algorithms`. The algorithms included in the VAUCANSON libraries can be listed using the command shown in Listing 1.2.

Listing 1.2 – Listing the algorithms in vaucanson.

```
$ git grep -E ".*NTERFACE:.*[^n]Automaton.*"
List of algorithm function declarations.
INTERFACE: Currently active algorithm.
XNTERFACE: Currently deactivated algorithm.
```

Clarify the interface.

Many algorithms in LEGACY are written using generic templates and do not mention *series* or *kind*. In order to clarify the code, all front-end functions must define types as shown in Listing 1.1.

Listing 1.3 shows the expected prototype of front-end functions.

Listing 1.3 – Prototype of the `eval` algorithm.

```
1 template<typename S, typename K, typename AI>
2 typename Element<Automata<S, K>, AI>::semiring_elt_t
3 eval(const Element<Automata<S, K>, AI>& a,
4      const typename Element<Automata<S, K>, AI>::monoid_elt_t& word);
```

Check for preconditions in the front-end function.

Front-end functions are named `<algorithm>(...)`, while algorithm cores are usually functions named `do_<algorithm>(...)` or functors.

Most preconditions are already in place, but new checks on the automaton *kind* have to be set up for algorithms restricted to certain *kinds*. This is performed by static assertion, as shown in Listing 1.4.

Listing 1.4 – Static assertion on a *kind* of automaton.

```
1 static_assertion_((misc::static_eq<K, labels_are_letters>::value),
2                   eval_is_only_defined_on_labels_are_letters);
```

Adapt the algorithm core to the new interface.

Calls to methods of the AUTOMATA structure can be found within the algorithm code. The definition of the AUTOMATA interface is currently being finalized by Jacques Sakarovitch and Sylvain Lombardy and the most significant methods are already implemented. New methods with significant differences from the LEGACY versions are prefixed with `new_` and should be called when available. Some simple methods, such as `add_state()`, remain unchanged.

The interface guide is not distributed publicly at the moment. Ask the team about it.

Calls that may have to be changed include:

- Operations on transitions: `add_transition()` and similar methods.
- Iterators: use the factories `make_delta_iterator()` and similar methods.
- Loops: loop macros, such as `for_all_states()` remain functional, but if the loop is to be rewritten entirely, use iterators explicitely.

Reorganize and optimize the core of the algorithm.

Many algorithms have stranded or disabled pieces of code that can be cleaned up. Functor algorithms can be changed into functions and vice-versa if it makes the code clearer. In some algorithms, straightforward optimizations will be apparent. For others, Jacques Sakarovitch and Sylvain Lombardy can suggest optimizations.

Keep in mind that the performance of an algorithms is limited by a few (often one) bottlenecks, and there is no need to optimize lines of code that only have a small influence on performance. Keeping the code clear and readable is often preferable to a small but confusing tweak. Part 2 details the cause of some bottlenecks encountered in VAUCANSON and how to optimize them.

Test and benchmark the algorithm.

As of now, testing algorithms is difficult because the working version of VAUCANSON is not in a stable state, some algorithms used in preconditions are missing, and the test suite is too complex to be used for small changes.

The most practical way to test algorithms is currently to use TAF-KIT, available for LAL for the contexts `lal_boolean_automaton` and `lal_z_automaton`. Basic tests are performed as follows:

- Set up two clones of the VAUCANSON repository, one on `exp/algos`, the other one first on `yavgui`, compile YAVGUI, then switch to `exp/libbench` and compile TAF-KIT.
- Activate the algorithm in the library by changing `XNTERFACE` into `INTERFACE` at the top of the header (only for `Automaton`, not `GenAutomaton`).
- Compile the library needed for the TAF-KIT version you want to test on, then compile TAF-KIT.
- Run algorithms using TAF-KIT in both repository clone.
- Visualize and compare results using YAVGUI in `<build dir>/yavgui/src`.
- Add relevant tests in the TAF-KIT test scripts (`taf-kit/tests`).

The benchmarking system is described in Part 3. Benchmarks are available for the most important algorithms, but a LAL version will sometimes have to be written:

- Duplicate the code from a `*bench.hh` file
- Change the context to `lal_boolean_automaton` or similar, then modify the benchmark information and output directory to reflect the change.
- Run `src/bench/generate_bench.sh all` to generate the corresponding `*bench.cc` file.
- Compile the new benchmark in the build dir.

Follow the process described in Part 3 to execute the benchmark.

1.3 Using template specialization to write an algorithm on a specific kind of automaton

Some algorithms operate only on a specific kind of automaton (`eval`, `product`, `determinize`), or have different implementations for different kinds (`realtime`). A separate algorithm core is written for each specific version of an algorithm. The front-end function remains the same and performs a dispatch. In effect, one version of the front-end function is instanciated through the template mechanism for each kind of automaton, therefore the dispatch is static.

Specific algorithm cores are templates specialized on a given *kind*. The only difference with the generic algorithm core is that the *kind* parameter is explicitely named. Listing 1.5 shows the implementation of `realtime` on LAL automata, which is a no-op since the automaton has no epsilon transitions and all transitions are labeled by letters.

Listing 1.5 – Specialization of `realtime` on LAL.

```
1  // Specialization on LAL
2  template<typename S, typename AI>
3  void
4  do_realtime_here(const Automata<S, labels_are_letters>&,
5                   Element<Automata<S, labels_are_letters>, AI>&,
6                   misc::direction_type)
7  {
8    // A LAL automata is realtime
9    return;
10 }
```

Part 2

Optimizing algorithms

This part presents examples of optimizations made during the major changes in VAUCANSON. Most of these optimizations can be applied on algorithms working with LAL automata.

Section 2.1 explains the consequences of changes in the automaton data structures and the resulting performance gain.

Section 2.2 shows modifications performed on algorithms and how they improve performance.

2.1 Design level

The major changes in VAUCANSON started as an effort to improve the performance of the library. Compared to its main competitor, OPENFST, VAUCANSON (LEGACY) has severe performance issues: `determinize` is 6 times slower than the OPENFST version (D'Halluin, 2009b). Compared with versions from other libraries, `eval` has a higher complexity.

While all causes of such performance issues in LEGACY are not clear, the high memory consumption of the generic automaton structure was identified as a possible cause: In order to maintain a high degree of genericity, all automata in LEGACY store maps of words and weights on every transition. In practice, many automata and algorithms only require a letter and a weight on each transition.

The goal of the main optimization at the design level is thus to allow lightweight data structures to be defined and used whenever applicable. This was formalized as a list of *kinds*, each kind defining specific properties on automata that could allow data structures and algorithms to be simplified (Galtier, 2009).

The algorithm `accessible` benefits directly from the design changes. It uses iterators on successors to select all the states reachable from all initial states. While it does not perform any operations on transition labels, a simpler data structure still yields a 20% gain in execution time on large automata (see benchmarks in Part 3, Figure 3.8).

2.2 Algorithm level

At algorithm level, three main optimizations are to be considered:

- Moving costly calls outside of loops.
- Reducing the use of series.
- Using optimized data structures.

Moving costly calls outside of loops.

In several algorithms, unnecessary computations are made within loops while they could advantageously be moved outside of loops. Listing 2.1 is a code excerpt from VAUCANSON illustrating a recurrent situation in algorithms. Listing 2.2 shows a more efficient way to achieve the same result, as the construction of a `monoid` is moved outside the loop. Note that these cases are not always bottlenecks, i.e. will not result in a huge boost of performance, but in the past, the correction of a similar errors in `quotient` resulted in a 40% gain in execution time (D'Halluin, 2009b).

Listing 2.1 – Expensive loop in `determinize`, before optimization.

```
1 for_all_const_ (subset_t, j, s)
2 {
3   for (delta_iterator t = input.make_delta_iterator(*j);
4        ! t.done(); t.next())
5   {
6     monoid_elt_t w(input.series_of(*t).structure().monoid(), *e);
7     // Work
8   }
9 }
```

Listing 2.2 – Optimized loop in `determinize`.

```
1  monoid_t monoid(input.series().structure().monoid());
2  for_all_const_ (subset_t, j, s)
3  {
4    for (delta_iterator t = input.make_delta_iterator(*j);
5         ! t.done(); t.next())
6    {
7      monoid_elt_t w(monoid, *e);
8      // Work
9    }
10 }
```

Reducing the use of series.

On LAL automata, series seldom have to be constructed, since each transition is labeled with only one letter. Tests such as the one listed in Listing 2.3 can be rewritten as in Listing 2.4, which is more efficient.

Listing 2.3 – Expensive test in `determinize`, before optimization.

```
1 monoid_elt_t w(monoid, *e);
2 if (input.series_of(*t).get(w) != zero)
3 {
4   // Work
5 }
```

Listing 2.4 – Optimized test in `determinize`.

```
1 if (t.label() == *e && t.weight() != zero)
2 {
3   // Work
4 }
```

In some cases, iterations on the support of a series can be completely avoided. In product, when series are stored of transitions, a costly test is performed on pairs of transitions. When only letters are stored on transitions, this test becomes much simpler. As a result, `product` is 65% faster on LAL automata than on series automata (Part 3, Figure 3.6) and `determinize` is 60% faster (Part 3, Figure 3.7), although some of the gain comes from lighter automata structures, as discussed in Section 2.1.

Using optimized data structures.

In some cases, data structures used in algorithms to store temporary results are not optimal. In LEGACY `eval`, a vector is used to store temporary weigths. The size of this vector is equal to n the number of states of the automaton on which a word is evaluated. The vector is traversed m times, with m the size of the word, resulting in a complexity of $O(n \times m)$.

In many cases, and especially when the automata on which `eval` is called is deterministic, the vector used to store weights is sparse. Replacing it with a map is more efficient, and the complexity in practice becomes $O(m)$ on determinisitic automata and $O(a \times log(a) \times m)$, with $a << n$ (on automata with a transition between any two states, $a = n$, but automata with many states rarely have this property).

The optimized version of `eval` has a linear complexity on deterministic automata and is much more efficient than the LEGACY version (Part 3, Figure 3.5).

Part 3

Benchmarks and version comparison

This part contains the measure of the performance impact of the changes made to VAUCANSON.

Section 3.1 describes the benchmarking process and how to reproduce the results presented in the report.

Section 3.2 presents some considerations related to the use of `gcc` and optimization flags.

Section 3.3 gathers the benchmarks related to the rewriting of VAUCANSON's structure, interface and algorithms.

3.1 Benchmarking in VAUCANSON

This section describes the architecture of the benchmarking system and provides examples on how to use it to evaluate the impact of changes in VAUCANSON.

The benchmarking system is located in `src/bench`. The file `src/bench/README` contains information that is not covered in this section. The benchmarks are available in the branch `exp/libbench` (before the major changes) and `exp/algos` (during the major changes).

3.1.1 Organization

The benchmarking system is centered around a collection of programs that run the most significant algorithms in VAUCANSON on a set of predefined input automata. These programs are generated from the cource code files in `src/bench/<dir>` where `<dir>` is the name of an algorithm.

Each directory contains one or several `*bench.hh` files that contain the benchmark code equipped with CBS macros. For each `*bench.hh` file, a corresponding `*bench.cc` file is generated during the bootstrap step or by calling `./generate_bench.sh all` from `src/bench`.

The current benchmarks cover the 8 most important algorithms in VAUCANSON. Until all

the rewriting is done, there is no need to add benchmarks on other algorithms. For more information on how to write benchmarks, see the description of CBS (D'Halluin, 2009b): http://lrde.epita.fr/~d-halluin/include/files/csi-techrep/200905-benchmarking.pdf.

Each benchmark program is run several times for increasing values of a parameter n, which in most cases refers to the size of the automaton used during the run. In each directory the file `Makefile.bench` shows the default range used for n so that all the runs of a benchmark take a few minutes or less on a 2GHz Intel Centrino with 1GB of RAM.

3.1.2 Input automata

Each benchmark is run on a predefined automaton. The description of each predefined automaton is located in `src/bench/common/README_AUTOMATA`. The most used predefined automata are:

- **aut_ladybird:** non-deterministic automaton on the 'abc' alphabet, has n states in its original form and 2^n states once determinized. This is the most used benchmarking automaton.
- **aut_dnk:** deterministic automaton with n states and k (typically 17) transitions for each state. For the state i, the j^{th} transition goes to the state $i+j$ and is labeled by the j^{th} letter of the input alphabet (the letters a-z). This automata can be used as an alternative reference for tests on a deterministic automaton when neither `determinized` nor `product` are available. In the eval benchmark, the automaton is constructed with 2^n states instead of n.
- **aut_b:** A simple automaton that counts the number of 'b' in words that are evaluated. Benchmarks using this automaton take the n^{th} power of the automaton.

3.1.3 Basic use

In the branch `exp/libbench`, all benchmarks can be run sequentially by using `make bench` from the build dir or separately by using `make bench` in `<build dir>/src/bench/<dir>`.

In the branch `exp/algos`, only part of the benchmarks are currently available (those on LAL automata). They have to be compiled and run one by one. Listing 3.1 shows how to compile and run a benchmark.

Listing 3.1 – Compiling and running benchmarks.

```
$ pwd
/home/fre/work/vaucanson/vaucanson/_build-listg/src/bench/eval
$ mkdir aut_dnk_lal
$ make eval_aut_dnk_lal_bench
$ for n in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15; do
 ./eval_aut_dnk_lal_bench $n; done;
```

When benchmarks are run, results are printed on the standard output and written to files in separate subdirectories (one per benchmark). The output files can be processed to extract parameters and results in GNUPLOT format, allowing plots to be generated easily. Listing 3.2 is an example of gnuplot script that gives an appearance similar to the benchmark plots is this report. Listing 3.3 shows how to extract the benchmark data and plot the results. Figure 3.5 is the resulting image file.

Listing 3.2 – Example of GNUPLOT script file.

```
$ cat eval_dnk.plot
set term postscript eps enhanced
set output "eval_dnk.eps"
set size 0.7,0.7

set title "Benchmark for eval (Automaton D_n_k)"
set xlabel "Automaton complexity"
set ylabel "CPU time (ms)"
set key top left
plot 'eval_dnk_map.data' using 1:2 title "Map implementation" smooth
   unique  w linespoints pt 5 lc 2 lw 2 lt 1, \
     'eval_dnk_vector.data' using 1:2 title "Vector implementation"
        smooth unique w linespoints pt 5 lc 1 lw 2 lt 1
```

Listing 3.3 – Plotting result data.

```
$ pwd
/home/fre/work/vaucanson/vaucanson/_build-listg/src/bench/eval
$ mkdir aut_dnk_lal
 Run benchmark on aut_dnk_lal (vector version)
$ mv aut_dnk_lal aut_dnk_lal_vector
$ mkdir aut_dnk_lal
 Run benchmark on aut_dnk_lal (map version)
$ mv aut_dnk_lal aut_dnk_lal_map
$ ../../../../cbs/bin/plot.pl -p "_n_" -r "time" -d aut_dnk_lal_map
> eval_dnk_map.data
$ ../../../../cbs/bin/plot.pl -p "_n_" -r "time" -d
   aut_dnk_lal_vector
> eval_dnk_vector.data
$ gnuplot eval_dnk.plot
$ display eval_dnk.eps&
```

3.2 Compilers and optimization flags

This section shows some of the effect of compilation flags for different versions of `gcc` on the performance of algorithms.

3.2.1 Recommendation for default flags

VAUCANSON has default optimization flags set to `-g -O2`, which is the Autotools default value. With `-O2`, functions are not inlined without the `inline` keyword. Since VAUCANSON relies heavily on wrappers and on the construction of small objects, and the `inline` keyword is rarely used in the library, this leads to a significant difference in performance between algorithms compiled with `-O2` and algorithms compiled with `-O3`. Figure 3.6 and Figure 3.7 illustrate this point, with performance gains of up to 30% in the `-O3` version.

Therefore, it is recommended to set the default flags in VAUCANSON to `-O3`. Should the development of new features require the use of other flags, these can be set during the compilation step by adding `CXXFLAGS="-g -O2"` when calling `configure`.

The negative impact on compilation time of using the `-O3` flag was not measured precisely, but was not perceived when compiling small parts of the library separately.

Note that the benchmarks made on LEGACY (D'Halluin, 2009b) used the default `-g -O2` flags. They were run again with `-O3` and showed the same performance gain as the examples in Figure 3.6 and Figure 3.7. E-mail `florent.dhalluin@gmail.com` for details. These benchmarks can also be reproduced on `seattle` from the `exp/libbench` branch.

3.2.2 Other observations

Inlining plays such an important role in the performance of compiled code in VAUCANSON that execution can take twice as long if automatic inlining (when using `-O3`) is improperly done. Jérôme Galtier studied the effect of inlining on code performance and this subsection presents some of his results. The benchmarks are based on `eval` (vector version), on `aut_dnk` with $n = 2^{17}$ and $k = 17$.

Previous benchmarks showed a significant performance difference between LISTG and BMIG (D'Halluin, 2009b), but the cause of this gap remained unclear. Figure 3.4 shows that the compiler version and automatic inlining efficiency may have a more significant influence on BMIG than on LISTG.

Also, using CBS to profile algorithms (D'Halluin, 2009b) has an overhead cost. This cost varies on the version of `gcc` and compilation flags used. For some versions on `gcc`, inlining plays an important role in the overhead cost because the calls to the benchmark measures are not inlined as they should be. Figure 3.4 shows lower performance when CBS is used, but with optimized inlining, the cost of the measures becomes low (< 5%, while in some cases it could reach 150%).

Note that in this context, using CBS refers to equipping parts of the algorithms, including loops, with calls to the CBS timer. In the benchmarks, calls to CBS are only wrapped around

algorithms, at an unsignificant cost.

Profile-Guided Optimization (PGO)

Older versions of `gcc` produce less optimized, slower code than more recent versions. Indeed, automatic inlining appears to be less efficient with old versions of `gcc`. To optimize inlining during compilation, the profile of function calls can be generated beforehand using the `gcc` option `--fprofile-generate` and used with `--fprofile-use`. This is refered to as Profile-Guided Optimization (PGO) and has a positive influence on performance (Figure 3.4).

With PGO enabled, LISTG and BMIG have similar performance on this test.

Compiler version	Use of CBS	PGO	Time (BMIG)	Time (LISTG)
g++ 4.4	×	×	26s	24s
		×	25s	23s
g++ 4.4	×		60s	57s
			60s	57s
g++ 4.3	×		65s	62s
			25s	25s
g++ 4.2	×		58s	71s
			67s	24s

Figure 3.4 – Comparison of compilation options on the performance of `eval`.

3.3 Results

This section contains the benchmark results run on VAUCANSON's algorithms at several stages of the rewriting process. They illustrate the gain of performance exhibited in the report.

The benchmarks were run on an Intel Centrino 2GHz laptop with 1GB of RAM. More benchmarks were run on different hardware configurations (the *seattle* machine at LRDE and a desktop Intel QuadCore) prior to the major rewriting of VAUCANSON (D'Halluin, 2009b). They are to be run again once VAUCANSON reaches a stable state (see `/work/d-halluin/vaucanson` on *seattle*).

Figure 3.5 through Figure 3.8 were generated using the method presented in Section 3.1. To get the complete benchmark output data, e-mail `florent.dhalluin@gmail.com`.

3.3.1 Eval

Figure 3.5 shows benchmark runs for `eval` on the automaton `aut_dnk`. As detailed in Section 2.2, using a vector to store the intermediate weights during the evaluation of a word on an automaton is in numerous cases very expensive compared to using a map. The map implementation lowers the complexity of the algorithm and the benchmark shows an important performance gain as the input automaton and word grow larger. Note that in the benchmark runs, the automaton has 2^n states and the input word 2^n letters.

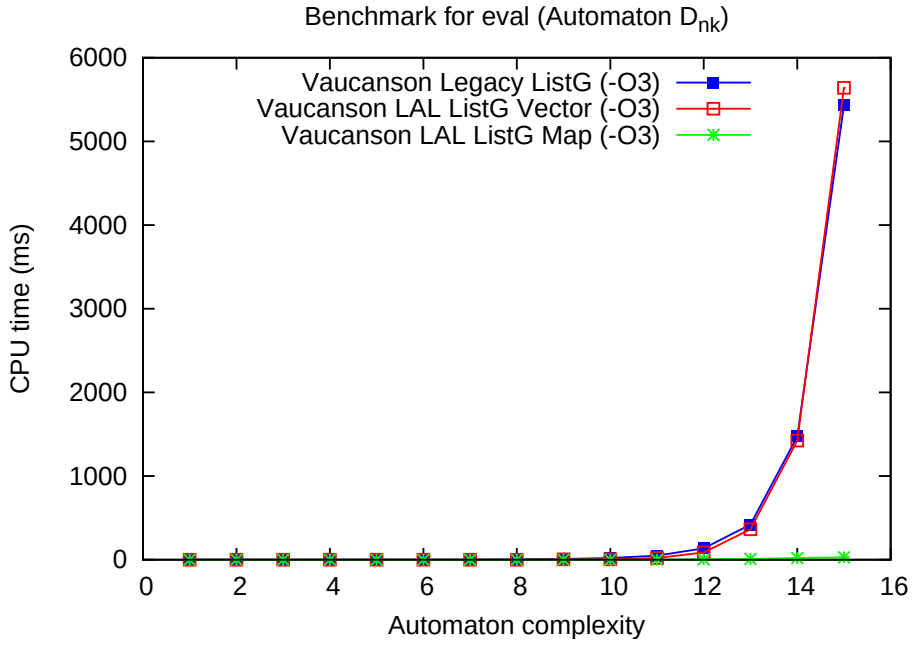

Figure 3.5 – `eval` benchmark.

3.3.2 Product

Figure 3.6 shows benchmarks runs for `product` on the automaton `B1` in the boolean context. While the optimization flags used have an influence on the algorithm's performance and some code optimizations were made, the most significant gain comes from the lightweight LAL structures.

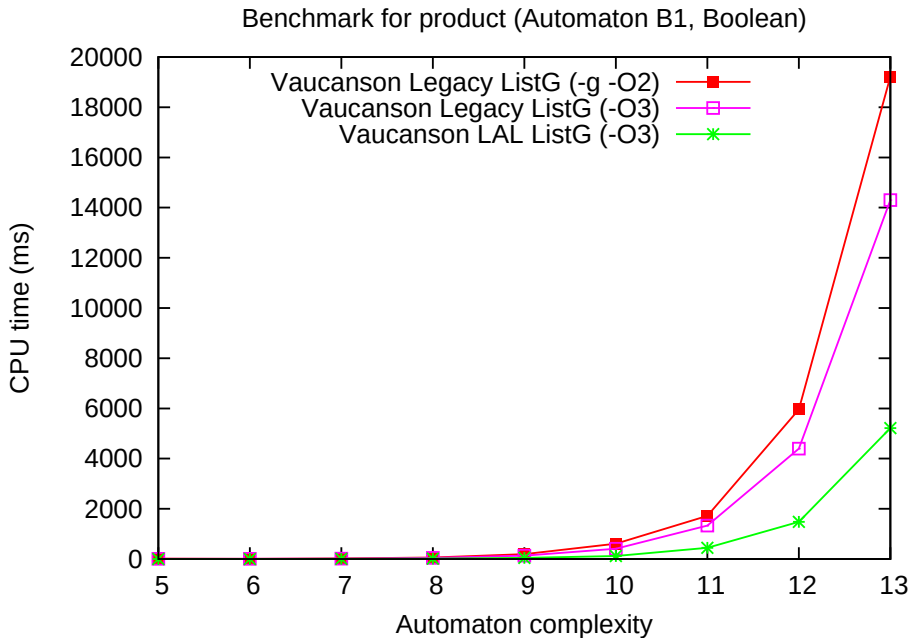

Figure 3.6 – `product` benchmark.

3.3.3 Determinize

Figure 3.7 shows benchmark runs for `determinize` on the Ladybird automaton. When using the best optimization flags available with the LAL structures, the performance of `determinize` is close to the OpenFst version. Note that during a benchmark run on OpenFst, loading and saving the input automaton from the hard disk in included in the execution time, while the Vaucanson benchmarks only measure the algorithm execution time. Nevertheless, the performance gain obtained through the rewriting process is significant.

Most of the gain comes from the LAL structures and from the removal of *series*, as the rewriting was straightforward, with some cleaning up but no other code optimization.

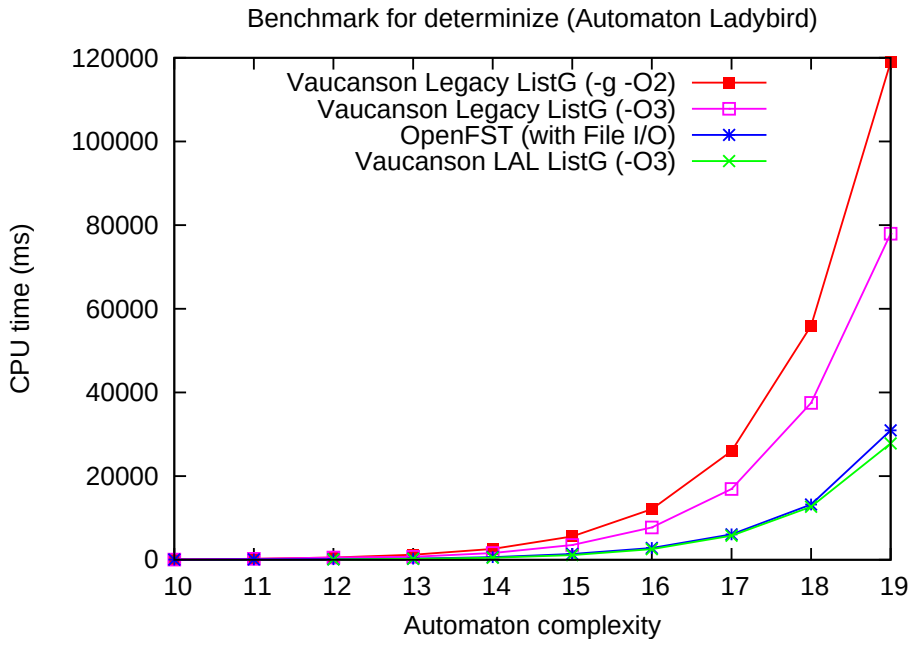

Figure 3.7 – `determinize` benchmark.

3.3.4 Accessible

Figure 3.8 shows benchmark runs for `accessible` on a automaton with a transition between any two states (complete graph with n vertices). The `accessible` algorithm is a pure graph function, which means it only depends on the underlying graph implementation. Even so, there is a small performance gain with the LAL implementation. The prototype of the algorithm was cleaned up but the algorithm core was not changed in any other way.

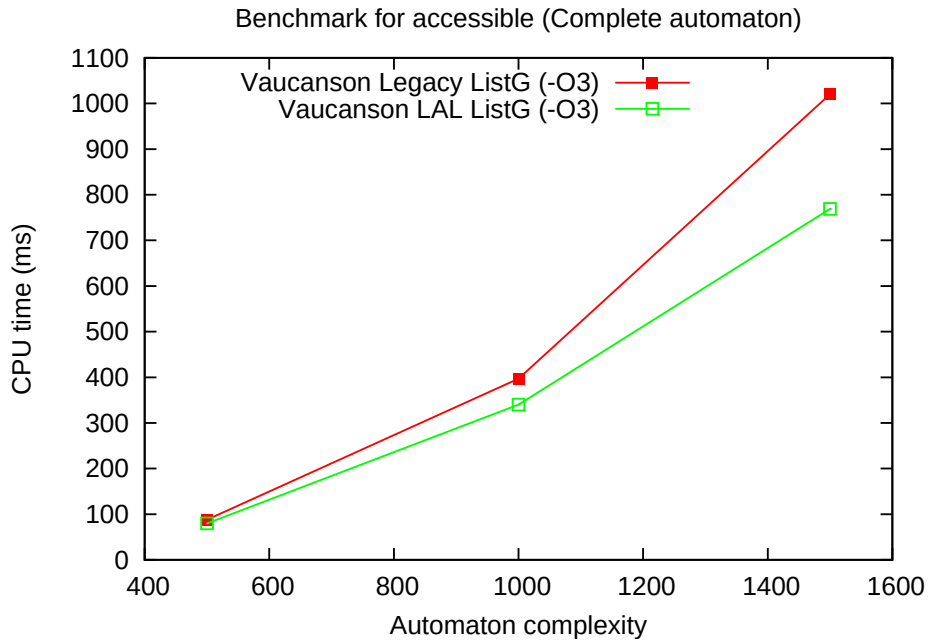

Figure 3.8 – `accessible` benchmark.

Conclusion

As of January 6th 2010, VAUCANSON is still in an unstable state between two major versions. This report is written from the point of view of the development team, as a short guide to the work in progress. It focuses on the visible part of the major changes ongoing in VAUCANSON, algorithm rewriting and benchmarks.

Adapting VAUCANSON algorithms to a simple AUTOMATA interface is a task on several levels. Algorithms operating on the graph structure require no change in the core algorithm; only the prototype of the front-end function should be adapted to the preferred way of declaring ELEMENT types. Algorithms that manipulate *series* must be adapted to the new automaton *kinds*: most of them can be rewritten to directly access words and weights, which improves their execution time by up to 60%. The complexity of some algorithms may be improved by replacing temporary data structure with more efficient ones: the complexity of `eval` improved from $O(m \times n)$ ($m = word\ length$ and $n = automaton\ states$) to $O(m)$ on deterministic automata by using a map to store weights instead of a vector.

This adaptation and optimization of algorithms takes longer than expected because tests are difficult to run. Benchmarks are also not thorough: they should be performed on more input automata in order to give a better picture of the performance of VAUCANSON. As of January 6th 2010, only 5 of the 30 to 40 algorithms in VAUCANSON are fully adapted to the *kind* system, with several more rewritten but not tested thorougly.

Nevertheless, the results obtained so far are good: algorithms that take advantage of the lightweight data structures for the LAL *kind* (`eval`, `product`, `determinize`) run 60% faster, and even pure graph algorithms (`accessible`) run 20% faster. Many opportunities to optimize algorithm code were identified and more should appear in the remaining 20 to 30 different core algorithms that have yet to be rewritten.

Observations about compiler versions and compilation options shed some light on the importance of inlining in VAUCANSON. Automatic inlining, as performed when specifying the `-O3` compilation flag, may improve performance by up to 30%. Better heuristics for automatic inlining are available through Profile Guided Optimization, although its impact on performance was only measured on one specific case that gave positive results of limited scope.

The results obtained so far show that the performance of VAUCANSON can be improved while maintaining genericity: efficient data structures and algorithms can be designed for automata that verify specific properties, while all other automata can still be manipulated through expensive but less restrictive data structures and algorithms.

Glossary

Branch.
Branch on the VAUCANSON git repository. `master` contains LEGACY (with an outdated benchmarking system). `exp/libbench` contains LEGACY and the latest benchmarking system. `exp/algos` contains the current latest state of the rewriting process. `exp/kinds` and `exp/interface` contain different parts of the rewriting process already integrated into `exp/algos` (double check this). The other branches are not related to the rewriting process. `yavgui` contains the C++ GUI for VAUCANSON.

CBS.
CBS is a benchmarking library developed for VAUCANSON. It is used to measure execution time and memory usage, and to present benchmarks in a readable form (D'Halluin, 2009b).

Current version, currently.
Refers to the state of the branch `exp/algos` at the time this report is written (January 6th 2010). Most of the work is still in progress and VAUCANSON is in an unstable state.

***Kind*.**
Refers to the set of properties of an automaton (restrictions on the labels and weigths stored on transitions, behavior of some algorithms). Four *kinds* are defined (LAL, LAA, LAW, LAS), but currently, only LAL is available. See the definition of the kinds (Galtier, 2009), and the glossary entry for LAL.

LAL.
The only new *kind* of automaton currently implemented. Labels are stored using the type `char`, while weights remain dependent on the semiring on which that automaton is defined (Galtier, 2010).

LEGACY**.**
The implementation of VAUCANSON before the major changes to the interface, the implementation of the LAL *kind*, and the rewriting of algorithms.

Major changes.

Refers to the changes in VAUCANSON designed early 2009 and worked on since July 2009. They include the implementation of the *kinds*, the interface changes and the rewriting of algorithms in progress since LEGACY.

Index

List of Figures

List of Listings

References

Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., and Mohri, M. (2007). OpenFst: A general and efficient weighted finite-state transducer library. In *Proceedings of the Ninth International Conference on Implementation and Application of Automata, (CIAA 2007)*, volume 4783 of *Lecture Notes in Computer Science*, pages 11–23. Springer. http://www.openfst.org.

D'Halluin, F. (2009a). Adapting vaucanson algorithms to a simpler interface. https://www.lrde.org/cgi-bin/twiki/view/Publications/201001-Seminar-DHalluin.

D'Halluin, F. (2009b). Benchmarking vaucanson and large c++ libraries with cbs. https://www.lrde.org/cgi-bin/twiki/view/Publications/200905-Seminar-DHalluin.

Galtier, J. (2009). Remedial treatment for vaucanson: an enhanced automaton concept. http://lrde.org/cgi-bin/twiki/view/Publications/200905-Seminar-Galtier.

Galtier, J. (2010). Adapting the data structures of Vaucanson to the concept of kind and a new interface. Technical report, EPITA Research and Development Laboratory (LRDE). https://lrde.org.

Lazzara, G. (2006). Automata and performances. https://www.lrde.org/cgi-bin/twiki/view/Publications/200607-Seminar-Lazzara.

Lombardy, S., Régis-Gianas, Y., and Sakarovitch, J. (2004). Introducing Vaucanson. *Theoretical Computer Science*, 328:77–96.

Sakarovitch, J. (2003). *Éléments de théorie des automates*.

VAUCANSON Group (2008). VAUCANSON home page. http://vaucanson.lrde.epita.fr/.