Boosting VAUCANSON's Iterators

Jimmy Ma

Technical Report n^o0820, May 2008
revision 1742

Vaucanson is a generic finite state machine manipulation platform. We have based our genericity on the ability to not only support various types of automata, but also to use different data structures to reprensent them. In its current state, we have various techniques to iterate over sets of transitions, however, none of them is really independent of the data structures. To overcome this problem, we have integrated the design pattern Iterator. Our goal is to assess the improvements given by this method in terms of performance and code writing.

Vaucanson est une bibliothèque générique de manipulation d'automates. Le cœur de sa généricité réside dans le support de types d'automates variés mais aussi sa capacité à s'appuyer sur différentes structures de données. Actuellement, nous avons différentes manières de manipuler des transitions. Cependant, aucune d'entre elles n'est réellement indépendante de la structure de données utilisée. Afin de pallier cela, nous allons nous tourner vers le design pattern Iterator. Nous évaluerons l'impact de ce design pattern sur les performances et sur l'utilisation de la bibliothèque en termes d'écriture d'algorithmes.

Keywords
VAUCANSON, automata, iterators, delta, deltac, deltaf, deltai, implementation, performance, benchmark.

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
jimmy.ma@lrde.epita.fr – http://www.lrde.epita.fr/

Copying this document

Contents

Chapter 1

Introduction

The VAUCANSON project is a finite state machine manipulation platform. Since its early steps of development, its strength relied on its genericity. Its goal is to provide a library that is able to handle automata with various alphabets and weights. Furthermore, it allows the user to easily change the data structure to adapt to the new constraints.

In our latest work, we added a new data structure based on the use of hash tables whereas the former one was based on adjacency lists. Both data structures have different properties which underlined some problems related the VAUCANSON interface.

Theoretically, the new data structure was supposed to outrun dramatically its predecessor. Against all odds, it only gave few improvements. We have traced the performance loss to a lack of freedom in the VAUCANSON application programming interface. For instance, when using VAUCANSON, some data copy are mandatory even though they are not necessary from the user point of view. To solve those problems, we will introduce new iterators in VAUCANSON.

The second chapter exposes the source of the problem and introduces existing techniques to solve that. Then, the third chapter presents our solution. Ultimately, the fourth chapter discusses the results and draws a few conclusions for this work.

Chapter 2

Iterating, a general overview

2.1 Iterate over an automaton

When working with automata, a common and natural desire is to be able to walk through it. Depending on the problems, there are various ways one might want to do it. When the taken path does not matter, it is easier to simply iterate over all the states, but in the other cases, things can be much more complex.

When using VAUCANSON, in order to iterate over an automaton, there exist numerous methods, called the delta functions, that iterate over a set of states or transitions. They can be divided in two groups:

- Those which are filling up a container.
- Those which are a applying functor.

2.1.1 Working with a container

The idea is to fill a container with the states or the transitions that matches the current needs. For instance, if one wants to iterate over all the successors of a given state, the function will put them all into a given container which is specified by the user in the function call.

These methods are defined as follows:

```
1  void delta(Iterator i, State src, DeltaKind k);
2  void delta(Iterator i, State src, Query q, DeltaKind k);
3  void letter_delta(Iterator i, State src, Letter l, DeltaKind k);
4  void spontaneous_delta(Iterator i, State src, DeltaKind k);
```

The `Iterator` taken as an argument is an insertion iterator which is used to fill the container previously created. The `hstate_t` is the state from which the query starts. Finally, the `DeltaKind` argument tells if you want the states or the transitions. In order to get a better idea as to how those methods are used, a few sample codes are available in Appendix A.

The second delta is taking an additional functor parameter `q`which is applied to each successor. If the functor `q` is returning `true`, then the current element will be added to the container. `letter_delta` is filling the container with the states or transitions reachable with the letter `l` from the state `src`. `spontaneous_delta` is performing the same task but with ε-transitions.

Furthermore, we have a last flavor of these types of delta functions, `deltac` which take a container instead of an insertion iterator and behave exactly like all the functions that were previously defined.

```
1  void deltac(Container i, State src, DeltaKind k);
2  void deltac(Container i, State src, Query q, DeltaKind k);
3  void letter_deltac(Container i, State src, Letter l, DeltaKind k);
4  void spontaneous_deltac(Container i, State src, DeltaKind k);
```

Finally, there is one variation of each of these delta functions: the reverse delta. They have the same prototype but with `rdelta` instead of `delta` and they fill in the predecessors or incoming transitions instead.

2.1.2 Working with a functor

An alternative solution to the container exists: `deltaf`. Its use is similar to what was described above:

```
1  void deltaf(Functor i, State src, DeltaKind k);
2  void deltaf(Functor i, State src, Query q, DeltaKind k);
3  void letter_deltaf(Functor i, State src, Letter l, DeltaKind k);
4  void spontaneous_deltaf(Functor i, State src, DeltaKind k);
```

They also exists with the reverse flavor. The main difference is the use of a functor. It applies the functor to each element that matches the request. For instance, all the successor states for a basic delta. The targeted optimization was the capacity to stop the iterations when the functor returned `true`. The `deltaf` can be compared to the `std::for_each` provided by the C++ Standard Template Library or STL (**??**).

2.1.3 Genericity issues

At first sight, these delta functions seemed to be working pretty well. However, with the new data structure `bmig` (Lazzara, 2008), some performance issues have been pointed out.

Before digging further, let's have a short reminder of the data structures used in VAUCANSON. On the one hand, we have a first data structure, called `listg`, which is a graph representation based on the adjacency lists. To be more precise, it uses the lists provided by the STL. The successors and the predecessors of a node are stored in two proper lists which are used in combination with a list of states. On the other hand, we have `bmig` where the goal is to lighten the use of multiple lists by using a single hash table. In this implementation, we are using the Boost Multi-index Container Library (Munoz, 2003). It provides an efficient implementation of hash tables with the specificity of supporting several sub-indexes on the stored items. In our case, we are keeping a single hash table with all the transitions of the automaton and we have defined sub-indexes on the successors and predecessors of each states. This way, `bmig` have embedded sub-containers on successors and predecessors.

The root of the problem lies in the constraints imposed by the use of the deltas. The trend of the algorithm writing style is to call `deltac`. This disables the main advantage of `bmig`: instead of using the existing containers provided by the hash table, it has became mandatory to, first, copy all the states or transitions into a container before any further operations.

`deltaf` can avoid this cost. It has, however, its own step back with regards to the code writing. The functors usually requires the algorithms to be more fragmented. Thereby, using

VAUCANSON becomes less intuitive because pulling the loops out of the body of the algorithm often makes the code less readable.

Therefore, we are seeking a solution which gives a wide range of action to the implementation of the data structure. Furthermore, this answer has to enhance or at least not to deteriorate the readability of the algorithms.

2.2 Iterators

In order to avoid unnecessary copy of data, the intuitive answer would be not to copy it, and to directly work on what is inside the automaton. A first way of implementing it would be to rely on the containers offered by the hash tables. However, it is not the best way, because with the other data structure `listg`, we do not have any equivalent. Hence, it may imply having to add significant cost to `listg` since wrapping virtual containers can often be a tricky task.

Another solution would be to simulate the existence of this container with the support of some wrapper. In our particular case, we are not interested by the container itself but rather by its items. We will, then, look into how we can directly provide iterators without having to build additional substructures.

Before making a decision on how we are going to implement our new iterators, let's have a look at the existing techniques. There is two main existing way of defining iterators: control-based or object-based. In addition, we will sub-categorize the object ones in two groups: the container-dependant ones and the self-sufficient ones.

2.2.1 The control-based style

Many languages, such as Python, Ruby or even shell scripting, provide control-based iterators. They are neither functions nor objects but keywords part of the language itself. The grammar of the language can also assist the implementation of methods to simulate this type of writing.

In Python (Guido van Rossum, 2003), the `for` keyword can implicitly act as an iterator over container.

```
for Value in List:
  print Value
```

In Ruby (Thomas and Hunt, 2001), containers have methods such as `each`, or derived from it, to iterate over its items.

```
list.each { |val|
  puts val
}
```

In the first C++ standards, the grammar did not provide any similar constructions. However, using external libraries, such as `BOOST_FOREACH` (Niebler, 2004), can give the same results.

```
BOOST_FOREACH(Type i, container)
  std::cout << i << std::endl;
```

This will be improved in the new C++0x standard (Becker, 2008) as this syntax will become a built-in feature of the C++ under the `for` keyword:

```
for (Type& i : container)
  std::cout << i << std::endl;
```

2.2.2 The container-dependant style.

In some object languages, there are some iterator objects that help iterating over a container. For instance, in C++, every container of the STL has its `iterator` which works as follows:

```
Container c;
for (Container::iterator i = c.begin(); i != c.end(); ++i)
  std::cout << *i << std::endl;
```

Note that the stopping condition is an iterator given by the container. The use of those iterators depends on the container which is why we called them container-dependant. Also, most of the information are held by the containers. It is similar to the pointer way of working with an array in C. The iterator points to an item, it can move all along and it is dereferencable to access the underlying element.

In the future, it will be possible to simulate the control-based style with iterators. Actually, we have the `std::for_each` construction which applies a function or a functor to all the items that lies between a pair of iterators. In the forth coming C++0x standard, we will be able to use lambda functions instead of functor for instance. This solves the readability issue introduced earlier with regards to the use of `deltaf`. With the lambda functions, the writing style is close to the control-based style.

```
Container c;

// In the old C++
std::for_each(c.begin(), c.end(), Function);

// Using C++0x and lambda functions
std::for_each(c.begin(), c.end(), [](Type i) {
  std::cout << i << std::endl
});
```

2.2.3 The self-sufficient style

There is a variant to the container-dependant iterators which is, for instance, the one used in Java. The objective is to provide a way to access the elements of an aggregated object sequentially without exposing its underlying representation (Gamma et al., 1995).

```
Iterator i = list.iterator();
while (i.hasNext())
  System.out.println(i.next());
```

The main difference with the container-dependant is that the iterator is self-sufficient with regards to the loop part. It is as if the iterator knew its own container and therefore, it is able to stop when it reaches its end. It is even more powerful, it could compute its own successors on the fly and therefore simulate a container. It can be seen as a getter on the next item more than an iterator.

2.2.4 Additional flavors

The three types previously introduced were designed to iterate over sets of elements. Based on those, others have been created to be able to perform additional actions during the iteration

process. Two variations have drawn our attention: `OutputIterator` and `EraseIterator`.

The `OutputIterator`, also known as insertion iterators, are capable of filling up a container. Their task is to iterate over a container while inserting items at the end of it. For instance, the following code will insert integers ranging from 1 to 5 into a list.

```
std::list<int>  list;
std::insert_iterator<std::list<int> > it(list, list.begin());
for (int i = 1; i <= 5; ++i)
  *it++ = i;
```

These are also the iterators used with the `delta` function presented in subsection 2.1.1.

The `EraseIterator` are capable of erasing item during the iteration. They are handy in a sense that they are aware of the constraints of the container with regards to the suppression process. For instance, in the STL list, suppressing the current item invalidate the current iterator, however, the use of a `EraseIterator` can solve this problem since it knows how to be able to keep tracks on its iterations.

Chapter 3

Integrating new iterators

It has been established that we needed to add new iterating techniques to VAUCANSON. This chapter presents how we are going to integrate the new iterators into VAUCANSON.

3.1 Directions

3.1.1 Choosing the iterator flavor

As we have seen, different concepts of iterators have already been worked out. We now have to choose the one that is best fitted to our needs.

The control-based iterators seems to be very handy to use, providing ease of code writing. They are, however, difficult to integrate in a C++ environment because of syntactic constraints. Heavy use of macro definition is generally needed to achieve this type of syntax if one do not want to rely on the `BOOST_FOREACH` library. The control-based writing style is what we aim to but we will reach it later on with some syntactic sugar.

The container-dependant style is the common flavor used in C++ in general. We are looking for a different solution since it is currently the source of our performance leak. If we want to stay in this direction, we have to work on some methods to help those iterators. The solution we are seeking is capable of avoiding data copy, so it has to be able to manipulate the internal data of the data structure.

We will finally choose the self-sufficient style since it seems to have more possibilities. Indeed, if the container does not physically exist, it can, then, be built or simulated on the fly. Therefore, we can avoid data copy and furthermore, it seems to be the most user-friendly solution. This last statement should be more detailed later on in this report. In the end, these iterators might even evolve into the erase iterators that were described in subsection 2.2.4.

3.1.2 Choosing the initialization method

The FACTORY pattern

Since the beginning, we aimed at keeping the VAUCANSON library as static as possible. Therefore, we do not want an abstract `Iterator` class but one that is templated by the needs. The templated part is however hidden underneath the API and the user only manipulates the instances that has been instanciated in his stead.

The first and intuitive idea to initialize these iterators was to add a method to the automaton to get an iterator. This way, the automaton can be seen as an iterator `Factory`. It turned out to be quite heavy to use.

The problem is related to the wish of keeping the code as static as possible. Without declaring an iterator abstraction, it was mandatory to specify most of the needs in the type declaration and then repeat most the them in the function call.

```
delta_state_iterator i = aut.deltai(state, delta_kind::states());
```

From the type definition, we know that an iterator on the successors is needed. In the function call, we have to either repeat explicitly that we want the accessible (`deltai`) states (`delta_kind::states()`).

The constructor method

To avoid the redundancy caused by the FACTORY pattern, we chose to initialize the iterator within its constructor.

```
delta_state_iterator i(automaton, state);
delta_transition_iterator i(automaton, state);
```

This is pretty intuitive, first, in the declaration of the iterator, we specify what type of iterator we want, and then, in its constructor, on which automaton and starting from which state we want to iterate.

3.2 The delta iterator specifications

3.2.1 The main types

Four new types are now available through the VAUCANSON API:

- `delta_state_iterator`.
- `delta_transition_iterator`.
- `rdelta_state_iterator`.
- `rdelta_transition_iterator`.

The task of each iterator is defined in its name:

- `delta` or `rdelta` specify whether we want to iterate over successors or predecessors.
- `state` or `transition` specify if we are interested in the accessible states or in the transitions that leads to them.

In order to keep things clear, the notion of reverse iterators in VAUCANSON is not related to the common notions of forward and backward iterators. The normaliterator iterates over the successors of a state and the reverse iterator iterates over its predecessors. We do not have the forward and backward iterators because they are not useful when it comes to working with automata.

The equivalent to `letter_delta` and `spontaneous_delta` are not yet implemented. Further details are given later on in this report, see subsection 4.2.1.

3.2.2 The interface specifications

Each of the iterators previously defined implements the following interface:

```
struct DeltaConstIterator
{
  DeltaConstIterator(const Automaton&, Automaton::hstate_t);

  void      next();
  bool      done() const;
  data_type operator*() const;
};
```

- The constructor initializes the iterator.
- `void next()`: step to the next item.
- `bool done() const`: return `true` if there is no more item to iterate over.
- `data_type operator*() const`: return the current item. `data_type` is a `hstate_t` if we have a state iterator, or a `htransition_t` if we have a transition iterator.

We have chosen to use explicit methods and not overriding the `operator++()` because it avoid the temptation of using some post and pre incrementation operators which in our case, do not always make sense. Furthermore, the notations `next` and `done` are used in Java for instance. It is also similar to the API of OpenFst (GoogleResearch and NYU, 2007) which is the weighted finite-state transducer library developed at Google.

Depending on the implementation, their properties are not exactly the same. On `listg`, the iterator is valid as long as the current item (state or transition) exists. On `bmig`, the iterator remains valid as long as there is no modification upon the automaton. To ensure code robustness, there should not be any modification on the automaton while using the delta iterators. The erase iterators have not been implemented yet, further details are available in subsection 4.2.2.

3.2.3 In real life

The control-based flavor, as we introduced it earlier, presents some advantages related to the code writing part and its readability. We provide sufficient syntactic sugar to write algorithms using a control-based style.

```
for_all_successor_states(i, automaton, state);
for_all_successor_transitions(i, automaton, state);
for_all_predecessor_states(i, automaton, state);
for_all_predecessor_transitions(i, automaton, state);
```

For example, to print out all the successors of a given state, the sample code would be:

```
for_all_successor_states(i, automaton, state)
{
  std::cout << *i << std::endl;
}
```

This way, the use of the iterators is totally transparent to the user and he can therefore enjoy the control-based style. Note the syntax here is quite similar to the `BOOST_FOREACH`, however, we have chosen to use our own to have explicit names to enhance the code readability.

Chapter 4

Performances and drawbacks

4.1 Performance analysis

In order to assess the improvements brought by the new iterators, we will conduct a short series of benchmarking.

4.1.1 Evaluation protocol

The test is pretty basic, first, we build an automaton with n states and with one transition between each state including loops. Therefore, we have an automaton with n states and n^2 transitions. Then, we iterate over each transition starting from each state.

The iterating part can be described as follows:

```
for_all_states(state, automaton)
  for_all_successors(i, automaton, *state)
    ;
```

This is a basic state querying or transition querying benchmark. For each state, we access all of its successors. More detailed code is available in Appendix A. Additionally, from Appendix A, it is clear that the new iterators can greatly improve the code readability.

We will not benchmark the memory consumption since the purpose of the report is to improve the computation time of VAUCANSON. Basic tests have been run to ensure that there is no loss in this part but it will not be detailed in this report.

The computation time benchmarks were run on a Intel® Core 2 Duo™ T7300, 2.0Ghz with 2Gb of DDR2. The compiler used is GCC version 4.2.3, the GNU Compiler with the following compilation flags: `-O3 -finline-limit-1500 -DNDEBUG`. The time values were retrieved using the benchmarking tools that are built into VAUCANSON. They consist of marking down ticks at some user defined watchpoints and comparing them at the end of the execution.

4.1.2 The iterator implementation on `bmig`

The results of the state querying (Figure 4.1) are good. The computation time of `delta` and `deltac` are almost the same which is what we expected since the implementation of `deltac` uses `delta`. `deltaf` is faster than the other two, it seems natural since it does not have to build a temporary structure. Finally, the quickest of all is `deltai`. It was expected because of how the `deltaf` are implemented. From how the dispatch is done in VAUCANSON, it has to

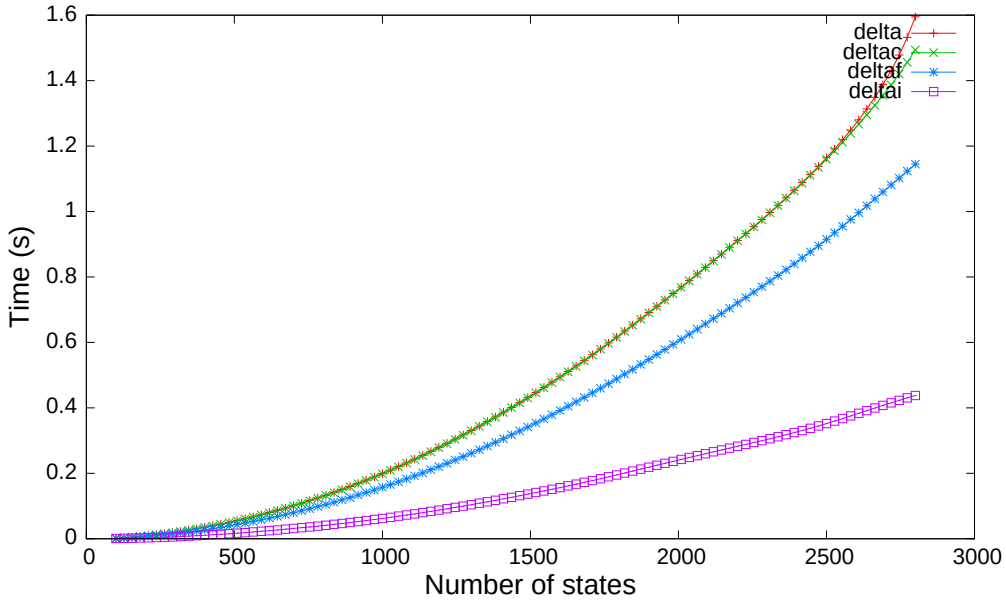

Figure 4.1: Querying states on `bmig`.

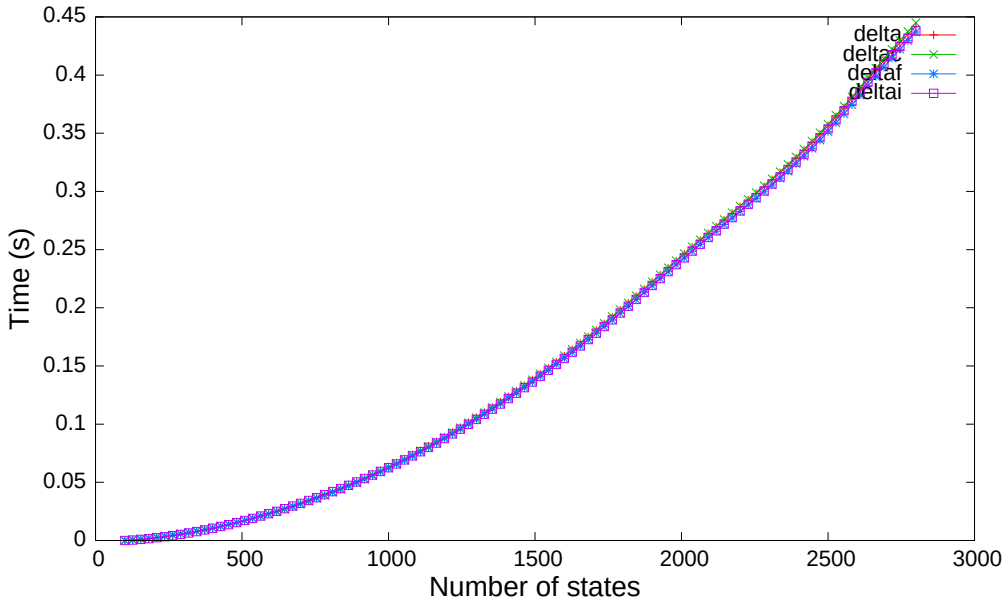

Figure 4.2: Querying transitions on `bmig`.

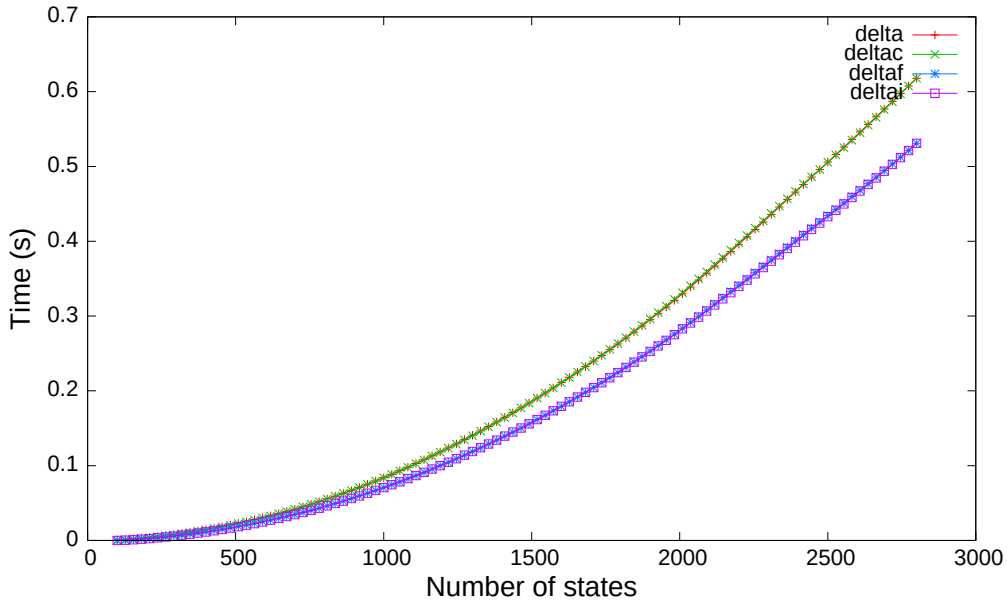

Figure 4.3: Querying states on `listg`.

use an intermediate functor in order to evaluate which transitions to apply the given functor to. `deltaf` was outrun because of this additional cost.

On the contrary, the results of the transition querying (Figure 4.2) are quite surprising. We had expected gains to be similar to the previous benchmark. However, the computation times are almost the same. If we have a very close look at their scores, the ranking remains the same but the difference is insignificant. The answer to this question is yet to be discovered.

Based on those two tests, the new iterating functionality seems more stable independently of the type of query and could therefore be used without any concerns or worries.

4.1.3 The iterator implementation on `listg`

The state querying on `listg` (Figure 4.3) is comparable to the one on `bmig`. For the same reason, we can group together `delta` and `deltac`. But in this case, `deltaf` and `deltai` have the same results. This was expected since we did not have the built in sub container that we had with `bmig`. Therefore, the execution time is pretty much the same in both cases.

Once more, the results of the transitions querying (Figure 4.4) are unexpected. The good thing is that both implementation seems to behave similarly on this matter.

The outcome here remains the same, using `deltai` seems to be a steady decision.

4.1.4 Comparing `listg` and `bmig`

From subsection 4.1.2 and subsection 4.1.3, we can steadily asses `deltai` is have the best time performances of the all the four existing deltas. Hence, we will compare its performance on both data structures.

One of the main drawbacks from the integration of `bmig` into VAUCANSON is that the computation time were unexpected (Lazzara and Ma, 2007). We expected `bmig` to be always faster

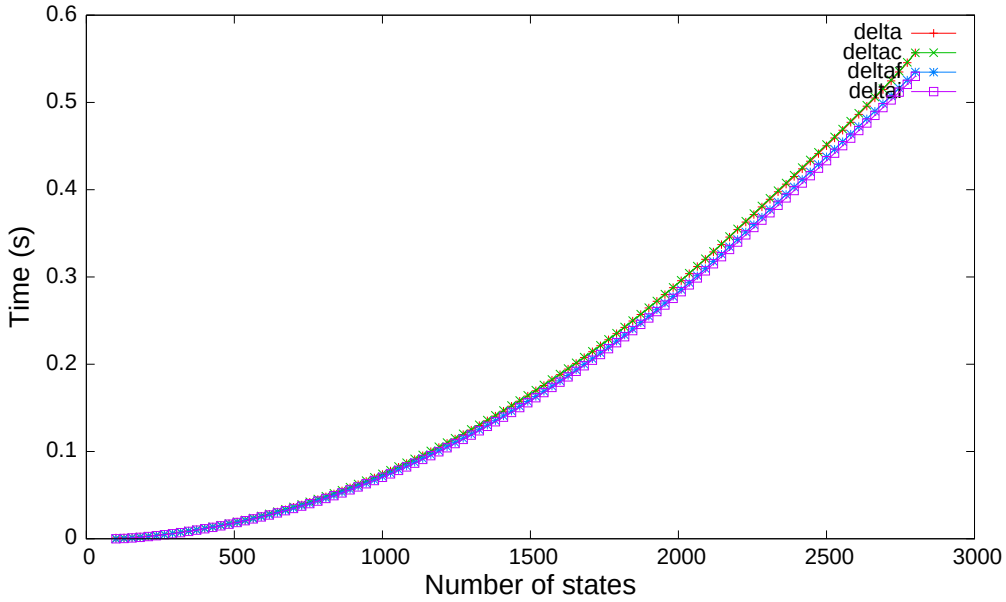

Figure 4.4: Querying transitions on `listg`.

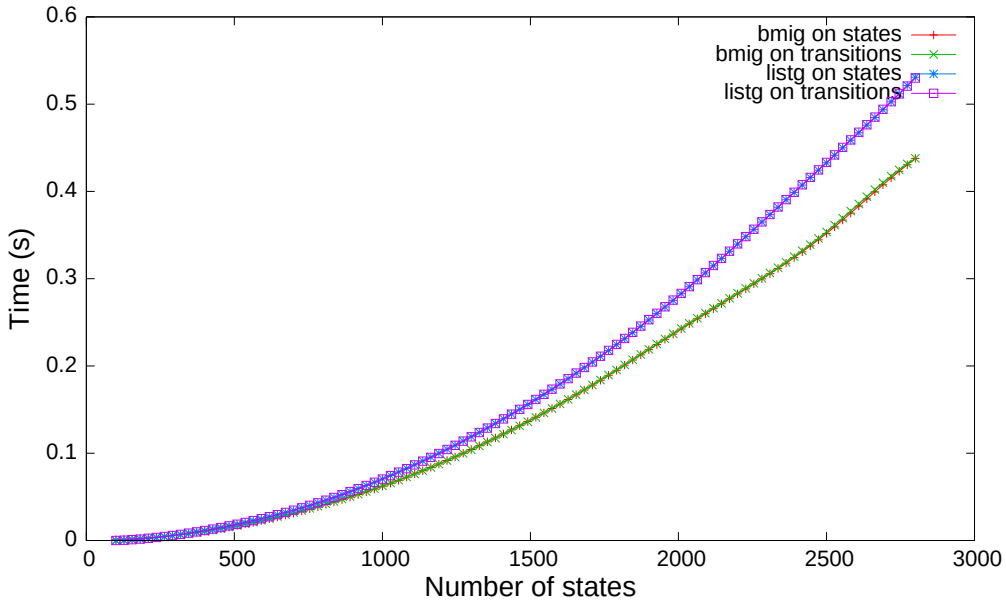

Figure 4.5: Comparing the implementation of the delta iterators on `bmig` and `listg`.

than `listg`. However, on some benchmarking tests, it happened to be slower. By looking at Figure 4.5, we can claim that `bmig` might, finally, always have better performances than `listg`.

This is yet to be confirmed on real case benchmarking, for instance on algorithms that exists in VAUCANSON. Those benchmarks are not yet available because the current iterators are not sufficient to run an algorithm. We need to first provide the equivalent to the `letter_delta` and the `spontaneous_delta` in order to evaluate the real case performances. More details are given in the next section of this report.

4.2 Drawbacks

So far, the new iterators of VAUCANSON have better computation time performance and improves significantly the code readability. There are however a few drawbacks from our present works.

4.2.1 The labels

In the implementation of the `letter_delta` and `spontaneous_delta`, we came across unexpected complications: the representation of the labels.

The labels in VAUCANSON, a quick overview

There are four different types of labels in VAUCANSON:

- `labels_are_letters`, the transitions are labeled by letters.
- `labels_are_words`, the transitions are labeled by words.
- `labels_are_series`, the transitions are labeled by series.
- `labels_are_couples`, the transitions are couples which can be any combination among letters, words or series.

In addition, two variations are available when the label are letters or words. There are either multiple transitions between two states or only one labeled by a polynomial, see Figure 4.6. An automaton can be in either form as long as all of its transitions follows the same rule.

Figure 4.6: Two equivalent automata with and without polynomial as transitions.

Label related issues

It is often unclear as to which type of label we have, especially when writing an algorithm. The issue here is that polynomial and multiple transitions are not handled the same way, and therefore have great impact on the code writing. Some wrappers are necessary but it is not yet clear as to where they are needed, if it is in the iterators or somewhere else.

The problem that it underlines is how to hand in the data to the user. The solution we want to avoid is having to specialize each algorithms according the type of label. Another part is that

Figure 4.7: Multiple label issues. The left automaton is wrong, the one on the right is the expected $\mathbb{Z}$-automaton.

some graphs specifications are still unclear at the moment. For instance, we definitely do not want two transitions between a same state to have the same label (Figure 4.7). In this case, we must have a single transition. The expected behaviour would be that the automaton automatically merges those transitions. This still has to be unified in the data structures. For what is sure, this has to be solved before having the `letter_delta` iterators and the `spontaneous_delta` iterators implemented.

4.2.2 The erase iterators

Another goal was to provide iterator capable of deleting the current item. The problem in this part is how to erase an item without invalidating the current iteration. This is highly entangled with the properties of the underlying data structure.

In `listg`, which is based on the Standard Template Library of the C++, the answer is quite easy. The deletion of an item invalidates the iterators which are pointing toward it. Therefore, delaying the deletion can easily solve this problem. Whereas in the multi indexes used in `bmig`, the deletion of an item makes all the iterators on the sub-indexes unsafe. Some leads are currently being investigated, however, as long as nothing is sure, this feature will not be available directly.

Chapter 5

Conclusion

The iterators presented in this report are promising. In terms of performance, they enable better use of the underlying hash table and its sub indexes. We can finally expect `bmig` to outrun `listg`. Up to now, we have successfully improved the computation time and the code readability of VAUCANSON with those new iterators.

However, the work on that part is yet to be finished. The next task will be to correct the labeling of the transitions to complete the set of iterating methods.

Regarding the readability, we have now released the constraints of intermediate types and the use of the VAUCANSON library is therefore lighter.

This also improves the genericity of VAUCANSON with regards to the performance when using different data structures. Those iterators have the best performance on both implementations and allows algorithm writing without having to care about the structure used.

Finally, a last step would be the integrate the erase iterators to provide easy state or transition deletion.

Appendix A

Using the delta functions

In this section, we present sample codes of the different workaround with the delta functions of VAUCANSON. The examples described below correspond to the code used in the benchmarks of subsection 4.1.1.

```
automaton_t a;
AUTOMATON_TYPES_EXACT(automaton_t);
```

This is a prefix to all the samples below. We have an automaton `a` and the second line is a macro definition that exports useful types to the top-level. Finally, we assume that we are in the namespaces `std` and `vcsn` to lighten the samples.

A.1 `delta`

```
// Declare a container that will be filled by delta.
vector<hstate_t> delta_container;

for_all_states(s, a)
{
  // The insert iterator used to fill the delta_container.
  insert_iterator<vector<hstate_t> >i(delta_container,
                                      delta_container.begin());

  a.delta(i, *s, delta_kind::states());
  for (vector<hstate_t>::iterator i = delta_container.begin();
       i != delta_container.end(); ++i)
    /* Do something here if needed */;

  delta_container.clear();
}
```

With the use of syntactic sugar, we get the following:

```
vector<hstate_t> delta_container;

for_all_states(s, a)
{
  insert_iterator<vector<hstate_t> >i(delta_container,
                                      delta_container.begin());

  a.delta(i, *s, delta_kind::states());
  for_all_iterator(vector<hstate_t>, i, delta_container)
    /* Do something here if needed */;

  delta_container.clear();
}
```

A.2 deltac

```
// Declare a container that will be filled by deltac.
vector<hstate_t> delta_container;

for_all_states(s, a)
{
  a.deltac(delta_container, *s, delta_kind::states());
  for (vector<hstate_t>::iterator i = delta_container.begin();
       i != delta_container.end(); ++i)
    /* Do something here if needed */;

  delta_container.clear();
}
```

With the use of syntactic sugar, we get the following:

```
vector<hstate_t> delta_container;

for_all_states(s, a)
{
  a.deltac(delta_container, *s, delta_kind::states());
  for_all_iterator(vector<hstate_t>, i, delta_container)
    /* Do something here if needed */;

  delta_container.clear();
}
```

A.3 deltaf

```
// Declare a functor to use with deltaf.
struct Functor
{
  void operator()(hstate_t) {
    /* Do something here if needed */
  }
};

// Back to the function implementation.
for_all_states(s, a)
{
  // The functor used by deltaf.
  Functor f;
  a.deltaf(f, *s, delta_kind::states());
}
```

A.4 deltai

```
for_all_states(s, a)
{
  // The use of delta iterator.
  for (delta_state_iterator i(a.value(), *s); ! i.done(); i.next())
    /* Do something here if needed */;
}
```

With the use of syntactic sugar, we get the following:

```
for_all_states(s, a)
  for_all_successor_states(i, a.value(), *s)
    /* Do something here if needed */;
```

Appendix B

Bibliography

Becker, P. (2008). *Working Draft, Standard for Programming Language C++*.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY.

GoogleResearch and NYU (2007). OpenFst.

Guido van Rossum (2003). *Python Reference Manual*. Python Software Foundation.

Lazzara, G. (2008). Boosting Vaucanson's genericity. Technical report, EPITA Research and Development Laboratory (LRDE).

Lazzara, G. and Ma, J. (2007). Boosting Vaucanson. Technical report, EPITA Research and Development Laboratory (LRDE).

Munoz, J. M. L. (2003). Boost multi-index containers library.

Niebler, E. (2004). Boost foreach.

Thomas, D. and Hunt, A. (2001). *Programming Ruby - The Pragmatic Programmer's Guide*. The Pragmatic Programmers.