Around Hopcroft's Algorithm

Manuel BACLET[1] and Claire PAGETTI[2]

[1] LSV - ENS de Cachan & CNRS - Cachan, France
IRIT - UPS & CNRS - Toulouse, France
baclet@lsv.ens-cachan.fr

[2] ONERA / Cert - Toulouse, France
claire.pagetti@cert.fr

Abstract. In this paper, a reflection is made on an indeterminism inherent to Hopcroft's minimization algorithm: the *splitter* choice. We have implemented two natural policies (FIFO and FILO) for managing the set of splitters for which we obtain the following practical results: the FILO strategy performs better than the FIFO strategy, in the case of a one letter alphabet, the practical complexity in the FILO case never exceeds a linear one and our implementation is more efficient than the minimization algorithm of the FSM tool. This implementation is being integrated in a finite automata library, the Dash library. Thus, we present an efficient manner to manipulate automata by using *canonical* minimal automata.
Keywords: Finite automata, minimization, Hopcroft's algorithm.

1 Introduction

The problem of minimizing a deterministic finite automaton has been widely studied. Finite automata libraries, such as FSM [MPR00], Mona [KM01], etc., include a minimization procedure. State of the art implementations of minimization algorithm is then an important issue for practical efficiency.

Minimization Algorithms For a detailed presentation of the currently known minimization algorithms, the reader is referred to Watson's taxonomy [Wat95].
For a given automaton labeled by the alphabet Σ where Q is the states set and F the final states set, most of minimization algorithms have a $O(|Q|^2)$ complexity and use one of the following two fix point strategies:

(S1) Consider the coarsest partition $\{F, Q/F\}$ and refine this partition until it satisfies some congruence properties;

(S2) Consider the finest partition and gather the equivalent classes.

Among the algorithms using other strategies, the Brzozowski algorithm [Brz62] allows to compute the minimal automaton from a non deterministic automaton in an exponential time.
A linear algorithm exists for complete deterministic automata over a one letter alphabet [PTB85]. Indeed, the problem is equivalent to determining the *coarsest partition* of the states set stable with respect to the transition relation function. Thus, the authors of [PTB85] use the second strategy (S2): the starting partition is the partition with singleton classes and the output is built via a sequence of steps in which two or more classes are merged.

An incremental algorithm has been proposed in [WD03,Wat01]. Unlike the other iterative algorithms, the intermediate results can be used since they consist in partially minimized automata.
The Hopcroft's algorithm proposed in [Hop71] has a theoretical $O(|\Sigma|.|Q|.\log |Q|)$ complexity which is currently the best for a minimization algorithm.

Contribution We propose a new implementation of the Hopcroft algorithm in the OCaml[1] language. We describe this implementation and some heuristics that significantly improve the speed of the practical state-of-the-art Hopcroft's minimization algorithm.
In the Hopcroft's algorithm, at each step a splitter is chosen among a set of classes in order to refine the partition. Every complexity computation leans on the worst case choice. It is the case in [BC04] where the authors exhibit an automata family over a one letter alphabet and a bad strategy that lead to the $O(|Q|.\log|Q|)$ complexity with the Hopcroft's algorithm. This means that there exists a bad strategy in the splitter choice while applying the Hopcroft's algorithm.
Our point of view is that there exists a good strategy in the splitter choice that allows a fast implementation of the Hopcroft's algorithm. This heuristic consists in a FILO strategy in which the most recent class is chosen as the splitter. In practice, this heuristic is powerful. In the case of a one letter alphabet, the practical complexity seems linear, even on the "bad" automata depicted in [BC04].
This implementation is being included in the Dash library (currently developed at the LSV[2]) which is a finite automata library designed to share common connected components between automata. The sharing of common components imposes that two equivalent automata are represented with the same minimal automaton. We thus propose an extension of our implementation, based on the work of [Cou04], to automatically compute a canonical representative.

Outline Section 2 recalls basic definitions and results concerning minimization. We present the Hopcroft's algorithm in section 3. We discuss its complexity and present an open question. In the section 4, the implementation is precisely depicted and the two strategies, FILO and FIFO are detailed. These implementations are then experimented on benchmarks and compared to other softwares in section 5. Finally, in section 6, we detail the efficient representation of automata in the Dash library using *canonical* minimal automata.

2 Minimal Automaton

In this section, we recall some basic notions and terminology on finite automata and regular languages. For a complete theory, one can refer to [BBC92,HU79]. In the sequel, Σ is a non empty finite alphabet.

Definition 1. *A deterministic, complete and* finite automaton[3] *over Σ is a tuple (Q, q_0, T, F) where:*

[1] http://caml.inria.fr/index.en.html
[2] http://www.lsv.ens-cachan.fr
[3] Since we only consider deterministic, complete and finite automata, we use the shortcut finite automaton.

– Q *is a finite set of states;*
– $q_0 \in Q$ *is the initial state;*
– $T : Q \times \Sigma \to Q$ *is the* transition function*;*
– $F \subseteq Q$ *is the set of final states.*

A rational language is associated to any finite automaton: it consists in the set of letter sequences which label paths from the initial state to a final state. We suppose that every considered automaton is *reachable*, i.e. any state of the automaton is reachable from the initial state. The automata theory ensures [Ner58,BBC92] that any rational language is recognized by a unique finite automaton (up to an isomorphism) with a minimal number of states. This automaton is the *minimal automaton* associated with the language.
For a given deterministic, complete and finite automaton, the equivalent (in term of language recognized) minimal automaton can be obtained by defining a congruence relation on the initial automaton's states, i.e. an equivalence relation which is stable with the transition function:

$$q \sim q' \implies \forall a \in \Sigma,\ T(q,a) \sim T(q',a)$$

If $\sim_m$ is the coarsest congruence such that

$$q \sim_m q' \implies (q,q') \in F^2 \text{ or } (q,q') \in (Q/F)^2,$$

we have the following result:

Proposition 1. *The finite automaton* $(Q/\sim_m, \overline{q_0}, T', F')$ *where*
– $\overline{q_0}$ *is the* q_0 *class up to* $\sim_m$*;*
– $T'(a,\overline{q}) = \overline{T(a,q)}$*;*
– $\overline{q} \in F' \Leftrightarrow q \in F$*;*
is the minimal automaton associated to the automaton (Q, q_0, T, F).

Given a finite automaton, this proposition allows to compute the associated minimal automaton by simply computing the equivalence relation. In the next section, we introduce an efficient algorithm for computing this equivalence.

3 Hopcroft's Algorithm

The Hopcroft's algorithm [Hop71] is detailled in Algorithm 1. It has a theoretical $O(|\Sigma|.|Q|.\log|Q|)$ complexity. The main principle consists in refining the coarsest partition until finding a stable partition (strategy S1). The initial partition is $\{F, Q \setminus F\}$ and each step of the algorithm consists in splitting the classes for which the stability constraint is not satisfied.
$\mathcal{P}$ is the current partition and L contains the elements of the partition to be treated. The set C is called the *splitter*.

1. The function `split` covers all the classes in $\mathcal{P}$ whose image by the transition function meets the splitter and determines the refined classes. Its implementation will be precise latter.
2. `SPLIT` has three arguments and decomposes the second argument (a subset) into two subsets depending on the splitter (the first argument) and the transitions labeled by the third argument. More precisely:

$$(B \cap T^{-1}(C,a), B \cap^c T^{-1}(C,a)) = \texttt{SPLIT}(B,C,a);$$

3. `ADD` has two arguments and adds a new subset in a set of subsets;
4. `EXTRACT` is the choice function on which we act to define the strategies we studied.

In the next section, we detail our implementation.

$L := \emptyset$
if $|F| < |Q/F|$ **then**
  $C_0 := Q/F; C_1 := F; \texttt{ADD}(C_1, L)$
else
  $C_1 := Q/F; C_0 := F; \texttt{ADD}(C_1, L)$
end if
$\mathcal{P} := \{C_0, C_1\}$;
while $L \neq \emptyset$ {The while loop corresponds to the Cut procedure} **do**
  let $C = \texttt{EXTRACT}(L)$ in
  for all $a \in \Sigma$ **do**
    for all $B \in \mathcal{P}$ {The forall loop corresponds to `split` (C, a) procedure} **do**
      let $(B', B'') = \texttt{SPLIT}(B, C, a)$
      if $|B'| < |B''|$ **then**
        $B := B''; \texttt{ADD}(B', \mathcal{P}); \texttt{ADD}(B', L)$
      else
        $B := B'; \texttt{ADD}(B'', \mathcal{P}); \texttt{ADD}(B'', L)$;
      end if
    end for
  end for
end while

Algorithm 1: Hopcroft's Algorithm

4 Implementation

4.1 Automata Representation

In our implementation, automata states are represented by integers: the states of an automaton $\mathcal{A}$ are numbered from 0 to $|Q| - 1$ and the following data structures will be used:

- the initial state is represented by an integer,
- the final states are represented by a boolean array of size $|Q|$,
- the transition function is represented by an array of integer array:

$$transition.(i).(a) = j \Leftrightarrow T(i, a) = j$$

4.2 Data Structures

The current partition is represented by an integer array *partition* of size $|Q|$ and an integer couple array *class_indices*.
To each class B of the current partition, the array *class_indices* maps the integer couple (l, h) so that elements of class B are the elements of the array *partition* whose indices are between l and h. During the execution of our implementation, elements in a class always have consecutive indices in the array *partition*.
When a class B is split in B' and B'', the elements in *partition* with indices between l and h are permuted so that elements of class B' have indices between l and h' and elements of B'' have indices between $l' = (h' + 1)$ and h. An integer array *class* is used to quickly find the class of an element.

In order to find efficiently the index of an element, an integer array *partition*$^{-1}$ is held up to date. It has the following property:

$$\textit{partition}.(i) = j \Leftrightarrow \textit{partition}^{-1}.(j) = i.$$

The *partition* array represents a one-to-one mapping over the integers between 0 and $|Q| - 1$ and the array *partition*$^{-1}$ represents the inverse mapping.
In order to realize the split operation efficiently, an integer list array *transition*$^{-1}$ is used to decide which classes have to be (possibly) split. It represents the inverse of the transition function:

$$i \in \textit{transition}^{-1}.(a).(j) \Leftrightarrow T(i, a) = j$$

A *pointer_array* is used to decide whether a class needs to be split: if B is a class with indices h and l in *class_indices* and if *pointer_array*$.(B) \neq (h - 1)$, then B needs to be split in B' and B'', with respective indices (l, h') and $(h' + 1, h)$, where $h' =$ *pointer_array*$.(B)$. At the beginning of each iteration of the algorithm, *pointer_array* associates to each class B the upper index h associated to B, if *class_indices*$.(B) = (l, h)$.
In our implementation, the two initial classes are numbered 0 and 1. Then, the created classes during the algorithm execution are numbered with increasing indices above 2.

4.3 L's Implementation

There are two natural choices for implementing the L object. By natural, we mean that there is no other simple choice that allows to carry out the ADD and EXTRACT operations in constant time [Knu01]. For instance, always choosing the class of L with the smallest size needs important resources and leads to a loss of performance in practice.

FIFO Strategy For this strategy, the classes are treated in their appearance order. If *classes_number* refers to the number of known classes and *next_class* indicates the next splitter, the algorithm 1 while loop, where the function `incr` increments an integer pointer, becomes Algorithm 2.

while *next_class* $\leqslant$ *classes_number* **do**
  let $C =$*next_class*
  for all $a \in \Sigma$ **do**
    `split` (C, a)
  end for
  `incr` *next_class*
end while

Algorithm 2: FIFO Cut Procedure

FILO Strategy For this strategy, the chosen splitter is the most recent class of the splitter set. The L object is then represented by a list: additions and deletions then occur on the top of the list. The Algorithm 1 while loop becomes Algorithm 3.

while $L \neq \emptyset$ **do**
  let $C = head(L)$ in $remove\_head(L)$;
  for all $a \in \Sigma$ **do**
    `split` (C, a)
  end for
end while

Algorithm 3: FILO Cut Procedure

4.4 `split` Function Implementation

Let C be the splitter and a a letter, the `split` function acts in two steps:

1. First, the set $T^{-1}(C, a)$ is considered and the elements of the array *partition* are permuted so that each class B is transformed into:

$$\boxed{B \cap T^{-1}(C,a)}\boxed{B \cap {}^{\complement} (T^{-1}(C,a))}$$

   where ${}^{\complement}A$ denotes the complementary of A in Q.
   Moreover, a list *visited_classes* which stores the encountered classes is also computed. A particular care must be taken when $C \cap T^{-1}(C, a) \neq \emptyset$.
2. For every B in *visited_classes*, we determine if B is refined by C: B is refined if, and only if $B \cap {}^{\complement}(T^{-1}(C, a)) \neq \emptyset$. (Since B was encountered in the first step, $B \cap T^{-1}(C, a) \neq \emptyset$.) If it is the case, a new class with the smaller part of B is created and added to L, otherwise nothing is done.

Due to a lack of space, the details of the `split` procedure can be found in [BP06]. We have presented the implementation, thus the contiguous question is its complexity and its efficiency. We only have actually partial results that we present in the following.

4.5 Complexity

We do not go into the details of the Hopcroft's algorithm complexity computation. The reader is referred to [BBC92,Knu01] for instance.

Theoretical Upper Bound The computation of the time complexity preponderant term is realized by bounding the sum, denoted by S, of the lists size of T^{-1} covered during the execution. It can be shown that $S \leqslant |\Sigma|.(\log_2 |Q|).|Q|$.

Reachability of the Upper Bound For the case of one letter alphabet, the authors of [BC04] construct an automata family and a splitter choice so that the bound $O(|Q| \log |Q|)$ is reached. Their strategy consists in choosing at each step a splitter that does not refine classes in L (if possible).

Conjecture There are open questions: does there exists a static strategy such that for every automaton the complexity is linear? And if it does, is it the case of the FILO strategy?

We have not found yet any way to compute this complexity. We only have practical results that we develop in the next section. For each automaton, a tree derivation can be constructed as proposed in [Knu01] for representing the program execution. It is a binary tree such that a node is a set of states and each son is a subset of the root such that the two sons form a partition. We add three colors: black when a set belongs to L and has never been modified, blue when a set belongs to L and has been refined, green when the set does not belong to L. A cost function can be associated to each node to compute the complexity. The idea is that the smaller the splitter is the smaller the cost function is locally, but this does not ensure that the minimum is global.

In order to obtain precise practical results of our two implementations behavior, our programs also compute the value of S.

5 Experiments

We realized many experiments on different automata families and we implemented several automata generators:

- a random automata family over a one letter alphabet with a number of states between 40 and 4.10^6;
- the automata family over a one letter alphabet constructed from the de Bruijn words given in [BC04];
- a random automata family over a two letters alphabet with a number of states between 40 and 4.10^6;
- a particular automata family over a four letters alphabet developped in previous works [BPP04] to model hardware signal processing components.

These experimentations allowed to compare the practical performances of our program with those of the Finite-State Machine Library FSM [MPR00].

5.1 One Letter Alphabet Results

Random one letter The random automata generation over a one letter alphabet is a simple problem since the topology of a reachable finite automaton over this particular alphabet has the particular structure of a *frying pan*.

The diagram given on the left of Figure 1 depicts our experimentations results: for any fixed size minimized automaton, we represent $max\{S/(|Q|.|\Sigma|)\}$ in function of $|Q|$, which corresponds to the worst case. The scale of the abscissa axis is logarithmic: a logarithmic curve will be represented by a line. We notice that the FIFO strategy is a linear function and it means that we obtain a bad complexity around $|Q| \log |Q|$.

On the opposite, the curve associated to the FILO strategy is always below the constant 3 and seems to converge towards the constant 2. This suggests a linear complexity for the one letter case.

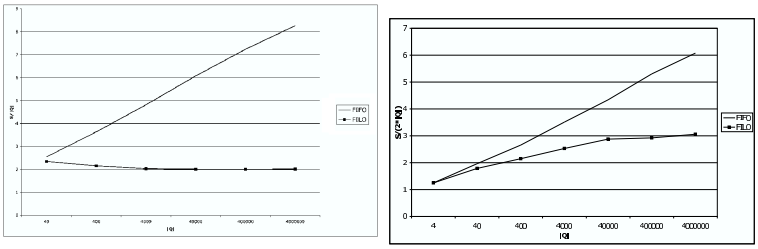

Fig. 1. One letter and two letters alphabet experimental results

De Bruijn's Words Automata We depict briefly the de Bruijn's words. A n-de Bruijn's word w over Σ is a word of minimal size such that for every word v of size n, v is a contiguous sub-word of w^2: $\forall v$, $|v| = n$, $\exists u_1, u_2$ such that $w = u_1.v.u_2$. For instance, the word $\omega = 11101000$ is 3-de Bruijn's word.
In [BC04], the authors use these words to construct one letter automata with a circular shape: if 1 and 0 are the letters, if $w = w_0.w_2...w_{n-1}$ is the de Bruijn's word, the associated automaton has n states $\{0, \cdots, n-1\}$, the transition function is $T : i \mapsto (i+1) \mod n$ and a state i is final if, and only if $w_i = 1$.
For the word $\omega = 11101000$, the automaton is depicted in Figure 2.

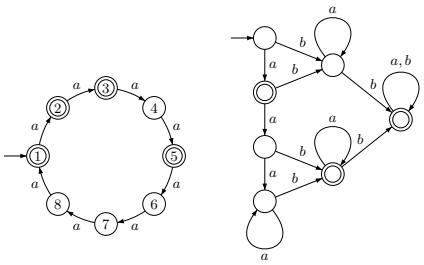

Fig. 2. Automaton for $\omega = 11101000$ and a two letter automaton

The experiments are made over automata associated to de Bruijn's word of size between 3 and 21. The results are identical to those of the previous subsection.

5.2 Two Letters Alphabet Results

The topology of a two letters alphabet is more complex than the case of one letter. We thus have chosen a particular family with the shape of a binary tree given in Figure 2.
Such automata with various number of states, which are final with a probability p, were minimized and the worst case results are given on the right side of Figure 1.

5.3 FSM Minimization Comparison

FSM [MPR00] is a powerful and performant finite-state machine library. It is able to manipulate large size automata and transducers. In particular, it holds a minimization implementation whose code is not known. We compared the library with our implementations on the benchmarks previously depicted. Our implementation is always faster, the efficiency varies from 4 to 15 times faster.
The test automata for four letters we have chosen are signal processing components studied in some verification process. Their interest is that they are realistic examples and that it is easy to generate many automata with the same structure but with different sizes. Their precise description can be found in [BPP04]. A small library was written to handle these models which are specified in a functional way, so that their descriptions are very close to the automaton definition.
Two families of automata were studied and the results are given in Figure 3.

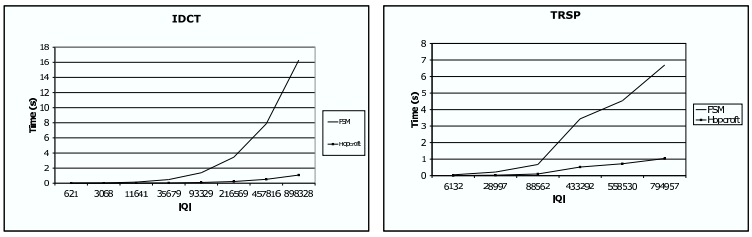

Fig. 3. Comparison with FSM

6 Canonical Minimal Automaton

When manipulating huge automata, a software decomposes each automaton in its connected components and stores the minimal automaton associated to each component. In order to reduce the resources and memory, if two components recognize the same language, it would be interesting to store it only once.
In the previous sections, we detailed minimization procedures that compute two isomorphic automata. We can refine this result and impose a states encoding so that we can compute an identical minimal automaton, which is called *canonical minimal automaton*. Thus, let $\mathcal{A}$ be an automaton, we denote by $\mathcal{A}_c$ the canonical minimal automaton (w.r.t. a particular states encoding). We have:

$$\forall \mathcal{A}, \mathcal{B}, \mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B}) \implies \mathcal{A}_c = \mathcal{B}_c.$$

Finding the canonical minimal automaton is rather straightforward: once the classical Hopcroft's algorithm has been applied, the states are renamed by a procedure of complexity $O(|\Sigma|.|Q|)$ with numbers in $\{0, \cdots, n-1\}$. The idea is roughly the following:

– the letters are ordered,
– the initial state has the number 0,
– if the state reached from 0 by a is not the initial state, then it is denoted 1 and is stored in a stack. Every successor of 0 is treated in the same way,
– every successor of the state in the top of the stack is numbered like the successors of 0 (note that if a state is already visited, nothing is done),
– the procedure ends when the stack is empty.

The previous solution can be applied when there is an initial state which is not the case when dealing with connected components. The author of [Cou04] proposes two techniques, which he did not prove formally, to find a canonical representative:

– a *sorted* Hopcroft's algorithm that needs to be executed twice;
– a *sorted* Hopcroft's algorithm with a static storage policy that computes directly the solution.

We prove in this paper that these solutions are correct. We included them in our FILO implementation so that we could experiment this approach on different automata families. Due to a lack of space, proves of propositions 2 and 3 can be found in [BP06].

6.1 Two-pass Solution

In this case, the idea is to sort the encountered classes list *visited_classes* handled in the split procedure. Thus, this procedure is modified by adding a sorting algorithm:
`visited_classes:=sort(visited_classes)`.
With this modification, the canonical minimal automaton is obtained by applying twice the FILO strategy. Let us denote by $\mathtt{min}_2$ this new implementation.

Proposition 2. *Let $\mathcal{A}$ and $\mathcal{B}$ two automata such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$, then*

$$\mathtt{min}_2 \circ \mathtt{min}_2(\mathcal{A}) = \mathtt{min}_2 \circ \mathtt{min}_2(\mathcal{B}).$$

The complexity is not easy to estimate. Nevertheless, on practical analyses, the sorting does not increase that much the execution time: the number of classes is often small.

6.2 One-pass Solution

The sorted Hopcroft's algorithm can be improved in the following way:

– when a class B is split in $\{B \cap T^{-1}(C,a), B \cap \complement(T^{-1}(C,a))\}$, we add to L either $B \cap T^{-1}(C,a)$, or $B \cap \complement(T^{-1}(C,a))$. The author of [Cou04] proposes to impose statically that one of these sets is always chosen. In our experiments, we found that the set $B \cap T^{-1}(C,a)$ is more profitable,
– during the initialization of L, again a static choice of the final set or the non-final states is done.

Note that there does not exist a good static strategy for the initialization step: indeed, if for $\mathcal{A} = (Q, q_0, T, F)$ it is more efficient to choose F, then for $\mathcal{A}' = (Q, q_0, T, Q \setminus F)$ the choice of $Q \setminus F$ is more advantageous.
Note that the modifications do not act on the correction but only on the complexity which could be increased.
Let us denote this new algorithm $\mathtt{min}_1$. It computes a canonical minimal automaton:

Proposition 3. *Let $\mathcal{A}$ and $\mathcal{B}$ be two automata such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$, then*

$$\mathtt{min}_1(\mathcal{A}) = \mathtt{min}_1(\mathcal{B}).$$

6.3 Practical Results

We apply the two sorted Hopcroft's algorithm on the automata families depicted in the section 5. The two-pass solution goes practically twice slower than the one-pass solution if the initialization step is good. At the opposite, if the initialization is badly realized, the complexity of the one-pass solution increases a lot. These practical results are presented in Figure 4. Since the initialization step of the one-

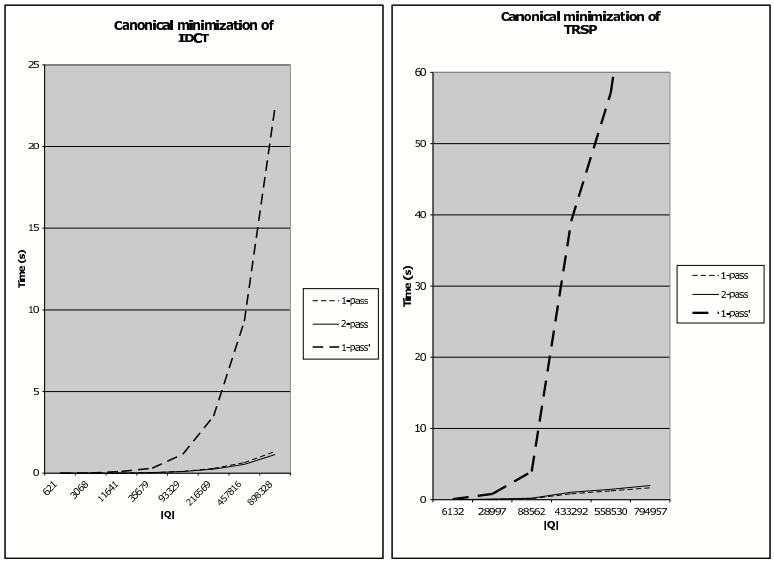

Fig. 4. Canonical Minimization

pass algorithm is too sensitive, the two-pass algorithm is included in the Dash library.

7 Conclusion

We have presented a detailed implementation of the Hopcroft's algorithm which is very efficient in practice. This implementation is being included in the Dash library. The way the set of splitters is handled is a crucial point for the efficiency and we studied two natural implementations. On very large scale experiments, we discover that the FILO strategy performs better and could lead to a minimization procedure that could be linear in the size of the automaton in the case of a one letter alphabet. In future works, we plan to find a proof of this belief.
In the second part of the paper, we investigate how the original algorithm can be modified in order to obtain a canonical minimal automaton associated to a rational language. Two solutions are proved correct and experimental results are depicted that show that, most of the time, one solution is better than the other.

References

[BBC92] D. Beauquier, J. Berstel, and P. Chrétienne. *Éléments d'Algorithmique*. Masson, 1992.

[BC04] J. Berstel and O. Carton. On the complexity of Hopcroft's state minimization algorithm. In *Proc. 9th Conference on Implementation and Application of Automata (CIAA'04)*, volume 3317 of *Lecture Notes in Computer Science*, pages 35–44. Springer, 2004.

[BP06] M. Baclet and C. Pagetti. Around Hopcroft's Algorithm. Technical Report LSV-06-12, Laboratoire Spécification et Vérification, ENS de Cachan, France, 2006.

[BPP04] M. Baclet, R. Pacalet, and A. Petit. Register transfer level simulation. Technical Report LSV-04-10, Laboratoire Spécification et Vérification, ENS de Cachan, France, 2004.

[Brz62] J. A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Mathematical Theory of Automata*, volume 12 of *MRI Symposia Series*, pages 529–561. Polytechnic Press, 1962.

[Cou04] J.-M. Couvreur. A BDD-like implementation of an automata package. In *Proc. 9th International Conference on Implementation and Application of Automata (CIAA'04)*, volume 3317 of *Lecture Notes in Computer Science*, pages 310–311. Springer, 2004.

[Hop71] J. E. Hopcroft. An n log n algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[KM01] N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.

[Knu01] T. Knuutila. Re-describing an algorithm by Hopcroft. *Theoretical Computer Science*, 250(1-2):333–363, 2001.

[MPR00] M. Mohri, F. Pereira, and M. Riley. The design principles of a weighted finite-state transducer library. *Theoretical Computer Science*, 231(1):17–32, 2000.

[Ner58] A. Nerode. Linear automaton transformation. In *Proc. American Mathematical Society*, volume 9, pages 541–544, 1958.

[PTB85] R. Paige, R. E. Tarjan, and R. Bonic. A linear time solution to the single function coarsest partition problem. *Theoretical Computer Science*, 40:67–84, 1985.

[Wat95] B. W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Eindhoven University of Technology, the Netherlands, 1995.

[Wat01] B. W. Watson. An incremental DFA minimization algorithm. In *Proc. 3rd International Workshop on Finite-State Methods and Natural Language Processing (FSMNLP'01)*, 2001.

[WD03] B. W. Watson and J. Daciuk. An efficient incremental DFA minimization algorithm. *Natural Language Engineering*, 9(1):49–64, 2003.