Cover Automata for Finite Languages

Michaël Cadilhac

Technical Report n°0504, June 2005
revision 681

Abstract. Although regular languages combined with finite automata are widely used and studied, many applications only use finite languages. Cover automata were introduced in Câmpeanu et al. (2001) as an efficient way to represent such languages.

The bold concept is to have an automaton that recognizes not only the given language but also words longer than any word in it. The construction of a minimal deterministic cover automaton of a finite language results in an automaton with fewer states than the minimal deterministic automaton that recognizes strictly the language.

In this technical report, the theory of cover automata is presented, then we focus on the algorithms that compute a minimal deterministic cover automaton.

Résumé. Bien que les langages rationnels vus au travers des automates finis soient très utilisés, beaucoup d'applications n’utilisent au final que les langages finis. Les automates de couverture introduits dans Câmpeanu et al. (2001) sont une façon efficace de représenter ces langages.

Le but est de créer un automate qui reconnaît un langage plus grand que celui d’origine, mais qui permet de le retrouver par seul test de la longueur du mot évalué. L’automate minimal construit à partir de celui-ci aura moins d’état que l’automate minimal de l’automate qui reconnaît strictement le langage.

Dans ce rapport technique, la théorie des automates de couverture sera présentée, puis il y sera détaillé les algorithmes qui permettent le calcul de l’automate minimal de couverture.

Keywords
Automata, Cover Automata, Minimization, Finite Languages.
Copying this document

Copyright © 2005 LRDE.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being just “Copying this document”, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is provided in the file COPYING.DOC.
Introduction

Finite languages are perhaps the most often used but the least studied family of languages in the formal language family hierarchy. In the past century, all the works made in this field were pieces of bigger works that did not focus on it. Only recently, several aspects of finite languages, such as the state complexity and decompositions, have been studied (see for example Campeanu et al. (1999), Yu (1999)).

However, the wide use of finite languages\(^1\) has conduced the research effort to focus on a way to represent finite languages efficiently. This effort has led to the creation of a specific field in the automata theory: cover automata.

First introduced by Câmpeanu et al. (2001), cover automata are an alternative representation of finite languages. The aim was to create an automaton that can, in some way or another, effectively represents a finite language but with fewer states than the corresponding automaton for this language. By “fewer states”, we are referring to the number of states of the minimal deterministic version of each automaton.

Roughly speaking, if \(L\) is a finite language and \(\ell\) the length of the longest word(s) in \(L\), a cover automaton for the language \(L\) accepts all words in \(L\) and possibly additional words of length greater than \(\ell\). We will see in the sequel how this construction can be used to:

- express finite languages as powerfully as with a classical automaton,
- reduce the size of the automaton.

In the first part, some basics of automata theory are introduced, together with a reminder on classical automata minimization. Then, in the second part, cover automata theory is described, and the main properties of these automata are presented. In the third part, the class of algorithms that compute a minimal deterministic finite cover automaton of a language is studied. Last, the final part presents the implementations and performances of these algorithms in the Vaucanson\(^2\) framework.

\(^1\)For instance in lexical analysis or in user interface translations (see Wood and Yu (1998))

\(^2\)Vaucanson: http://vaucanson.irde.epita.fr
Chapter 1

Basics

This chapter presents some basics of automata theory we need in this report. First, some classical notions are presented, then the class of algorithms that minimize an automaton. The following definitions are relative to the use we will have of them (i.e. they are not absolute). Unless specified otherwise, they are taken from Sakarovitch (2003).

This part being only a reminder, the theorem and propositions listed here will not be demonstrated.

1.1 Definitions and notations

Notation 1 (Alphabet and words). If $\Sigma$ is a set of letters, We will write $\Sigma^*$ for the set of words based on the alphabet $\Sigma$. The empty word will be denoted $\varepsilon$. Additionally, we will note $\Sigma^\leq \ell$ the set of words longer than $\ell$ (respectively, $\Sigma^\leq \ell$ the set of words shorter than $\ell$).

Definition 1 (Language and Finite language). A language $L$ on the alphabet $\Sigma$ is a (possibly empty) set of words, that is to say, $L \subseteq \Sigma^*$. A language $L$ is finite if $\text{Card}(L)$ is bounded.

Definition 2 (Automaton). An automaton $A$ is specified by the following elements:

- A non-empty set $Q$ called the set of states of $A$,
- A non-empty finite set $\Sigma$ called the alphabet of $A$,
- Two subsets $I$ and $F$ of $Q$; $I$ being the set of initial states and $F$ being the set of final ones,
- A function $\varphi : Q \times \Sigma^* \rightarrow 2^Q$ called the transition function. Given a state $q$ and a word $w$, $\varphi(q, w)$ represents the set of states which have incoming transitions from $q$ labeled by $w$.

Definition 3 (Language of an automaton). The language recognized by the automaton $A$ is

$$L(A) = \{w \in \Sigma^* \mid \exists q_i \in I, q_f \in F, q_i \xrightarrow{w} q_f\}$$

Definition 4 (Finite automaton). We will say that an automaton is finite if the set $Q$ is finite.

Definition 5 (Real-time automaton). An automaton is real-time if the transition function $\varphi$ is defined as $\varphi : Q \times \Sigma \rightarrow 2^Q$. In other words, all the transitions are labeled by a single letter.

Notation 2 (Automaton). With the notations of Definition 2, an automaton $A$ will be written as a quintuple $A = (Q, \Sigma, \varphi, I, F)$. We will say that $A$ is an automaton over $\Sigma$. 
The sequel of this report will only consider real-time automata.

**Definition 6** (Completeness). An automaton is complete if for all state \( q \) and for all letter \( a \) in the alphabet, there is an outgoing transition from \( q \) labeled by \( a \). An automaton can be completed: a sink state is added on which every missing transitions will go to.

**Definition 7** (Trim). An automaton is trimmed if all its states are reachable from the initial state and can reach a final state. Trimming usually remove the sink state.

**Definition 8** (Deterministic Finite Automaton). A Deterministic Finite Automaton (DFA) is an automaton such that \( \forall q \in Q, \forall l \in \Sigma, |\delta(q, l)| \leq 1 \) and \( |l| = 1 \), i.e. for all states and all letters, it exists at most one outgoing transition labeled by this letter.

**Definition 9** (Non-deterministic Finite Automaton). A Non-deterministic Finite Automaton (NFA) is the general case in the determinism property: determinism and non-determinism are not mutually exclusive concepts, a deterministic automaton is non-deterministic.

The reminder of this document will only consider deterministic automata.

**Definition 10** (Evaluation). An evaluation of a word \( w = (w_1, \ldots, w_n) \) on an automaton \( \mathcal{A} \) is defined recursively on the letters by:

- Let \( Q_k \) be the states reached by the word \( w' = (w_1, \ldots, w_k) \).
- \( Q_{k+1} \) is the states reached with the letter \( w_{k+1} \) from the states of \( Q_k \).

Then the result of the evaluation is true if \( Q_n \neq \emptyset \), false otherwise.

**Notation 3** (Evaluation). The evaluation of a word \( w \) on an automaton \( \mathcal{A} \) will be written \( \text{eval}(w, \mathcal{A}) \). If \( \text{eval}(w, \mathcal{A}) \) is true, then we say that \( w \) is recognized by \( \mathcal{A} \), or that \( w \) is in the language \( L(\mathcal{A}) \) described by \( \mathcal{A} \).

### 1.2 Isomorphism and minimal automata

#### 1.2.1 Preliminaries

First, the reader has to be convicted that there is not only one automaton for a specified language. For instance, these two automata recognize the same words:

![Automata for “a*”](image)

**Figure 1.1: Automata for “a*”**

The question that arises knowing that is: “*Does an automaton have a canonical representation?*”

It is the case with deterministic automata: the minimal deterministic automaton is unique for a given language. For example, the minimal DFA for “a*” is the one on the left on Figure 1.1.
The following definitions and theorems will be needed to understand the algorithms that compute the minimal automaton of a DFA:

**Definition 11** (Isomorphism of automata). Two automata are isomorphic if they are the same without regard to the name of the states.

**Definition 12** (Minimal automaton). An automaton \( \mathcal{A} \) is minimal if for all automaton \( \mathcal{B} \) such that \( \mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B}) \), \( \text{Card}(\mathcal{A}) \leq \text{Card}(\mathcal{B}) \) with \( \text{Card}(\cdot) \) giving the number of states.

**Theorem 1** (Unicity of the minimal automaton). The minimal automaton of a language \( \mathcal{L} \) is unique through automata isomorphism.

**Definition 13** (Equivalency of states). Let \( \mathcal{A} = \langle Q, \Sigma, \varphi, I, F \rangle \), \( (p, q) \in Q^2 \).

Suppose we reached \( p \) and \( q \) with two words \( w_1 \) and \( w_2 \).

We say that:

- \( w_1 \) is equivalent to \( w_2 \) and note \( w_1 \equiv w_2 \) if \( \forall z \in \Sigma^*, w_1 z \in L \iff w_2 z \in L \),

- \( p \) is equivalent to \( q \) and note \( p \equiv q \) if \( w_1 \equiv w_2 \).

The relation \( \equiv \) is an equivalence relation.

Intuitively, \( p \equiv q \) if \( p \) and \( q \) have the same future.

The following theorem establishes the link between equivalency and minimal automata:

**Theorem 2** (Myhill (1957)). A DFA is minimal if and only if \( \forall (p, q) \in Q^2, p \neq q \iff p \equiv q \).

### 1.2.2 On the way to minimization

From the Theorem 2, we could easily deduce a simple algorithm that makes the minimization. Given an oracle that would say if two states are equivalent, we could merge equivalent states (as they have the same future) to obtain an automaton in which no states are equivalent.

For instance, in the following automaton, \( q \) and \( r \) are equivalent:

![Automaton for "(a + b)c"](image)

Figure 1.2: Automaton for “(a + b)c”
Thus, to obtain the minimal automaton for “(a + b)c”, q and r should be merged:

![Minimal automaton for “(a + b)c”](image.png)

1.2.3 Minimization algorithms

In this section, we will work on the automaton \( A = (Q, \Sigma, \varphi, I, F) \).

Minimization algorithms aim at finding the equivalences between the states. They usually work by partitioning the set of states in equivalence classes, that is to say in classes in which states are all equivalent.

The first gross division that could be made is \( Q = F \cup Q \setminus F \). Indeed, we know that the elements of \( F \) could not be equivalent to the elements of \( Q \setminus F \); only a state in \( F \) can lead to a final state on the input \( \varepsilon \).

It does not mean, however, that the states in \( F \) are all equivalent.

The algorithms begin with this first partitioning, then make successive refinements to reach a fixed point. The following theorem assures the equivalence between this task and the minimization.

**Theorem 3 (Myhill (1957)).** For a deterministic finite automaton \( M \), the minimum number of states in any equivalent deterministic finite automaton is the same as the number of equivalence classes of \( M \)'s states.

**Moore's algorithm**

The algorithm of Moore (1956) is based on the following fact:

**Fact 1.** All equivalent states go to equivalent states under all inputs.

The time complexity of the algorithm is in \( O(n^2) \), with \( n = \text{Card}(A) \). Its principle is quite simple:

```plaintext
start with two groups F and Q \setminus F
do |
  for every group |
    for every state in the group |
      find which groups the inputs lead to |
    if there are differences |
      partition the group into sets containing states which go to the |
      same groups under the same inputs |
  } |
while a partitioning has been made in the loop
```
Let us treat an example: we will use the automaton $\mathcal{M}$ that describes the language denoted by the regular expression “$(aa + b)^*ab(bb)^*$”. This automaton is real-time and deterministic, in other words, it matches our requirements:

Our first partition is the split of $Q$ regarding final states:

\[ A = \{q_3, q_8\}, B = \{q_1, q_2, q_4, q_5, q_6, q_7\} \]

We shall now find if the states in these groups go to the same group under inputs $a$ and $b$. It could be seen that the states of group $A$ both go to states in group $B$ under both inputs. However, this is not the case for the group $B$; the following table shows the result of applying the inputs to these states (for instance, the input $b$ leads from $q_7$ to $q_8$, i.e. to $A$)

<table>
<thead>
<tr>
<th>In state:</th>
<th>$q_1$</th>
<th>$q_2$</th>
<th>$q_4$</th>
<th>$q_5$</th>
<th>$q_6$</th>
<th>$q_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$ leads to:</td>
<td>$B$</td>
<td>$B$</td>
<td>$B$</td>
<td>$B$</td>
<td>$B$</td>
<td>$B$</td>
</tr>
<tr>
<td>$b$ leads to:</td>
<td>$B$</td>
<td>$A$</td>
<td>$B$</td>
<td>$B$</td>
<td>$B$</td>
<td>$A$</td>
</tr>
</tbody>
</table>

The input $b$ helps us distinguish between two of the states ($q_2$ and $q_7$) and the rest of the states in the group since it leads to group $A$ for these two instead of group $B$. Thus we should split $B$ into two groups and we now have:

\[ A = \{q_3, q_8\}, B = \{q_1, q_4, q_5, q_6\}, C = \{q_2, q_7\} \]

The next loop in the algorithm shows us that $q_4$ is not equivalent to the rest of group $B$ with input $a$ and we must split again.

Continuing this process until we cannot distinguish between the states in any group by employing input tests, we end up with the following classes:

\[ A = \{q_3, q_8\}, B = \{q_1, q_5, q_6\}, C = \{q_2\}, D = \{q_4\}, E = \{q_7\} \]
Considering the above theoretical definitions and results, we can say that all states in each group are equivalent because they all go to the same groups under the inputs $a$ and $b$.

The minimal automaton for $L(M)$ is then:

![Minimal automaton for $L(M)$](image)

Figure 1.5: Minimal automaton for $L(M)$

**Hopcroft’s algorithm**

The minimization algorithm of *Hopcroft (1971)* is based on the same fact as Moore’s one, but its formulation is changed:

**Fact 2.** On some input and for every group $G$, all the groups should not contain some states that are predecessor of $G$ and some others that are not.

This reversed point of view (i.e. to consider predecessors) together with a trick in enqueueing in the algorithms lead to a time complexity in $O(n \log(n))$. Hopcroft’s algorithm could be written as follows:

```plaintext
initialize $\Pi = (F, Q \setminus F)$, put $F$ in a "To-treat" queue $T$, 
while $T$ is not empty { 
    remove a set $S$ of $T$, 
    for each letter in $\Pi$ { 
        compute $X$: the list of states that go in $S$, 
        $Y$: the list of states that don’t, 
        for each sets $\pi$ in $\Pi$ { 
            if $\pi \cap X \neq \emptyset$ and $\pi \cap Y \neq \emptyset$ { 
                split $\pi$ in $\pi \cap X$ and $\pi \cap Y$, 
                enqueue the smaller one in $T$. 
            } 
        } 
    } 
}
```

Let us apply this algorithm to the automaton $M$ of Figure 1.4. One of the first iteration will consider $S = \{q_2, q_7\}$ with the input letter $b$. In the other set, namely $\{q_0, q_3, q_4, q_5, q_6\}$, the states $\{q_1, q_6\}$ are predecessors of $S$ but the others are not. The first split will be made, and the partition will be:

$\Pi = \{(q_0, q_3, q_5), \{q_2, q_7\}, \{q_1, q_6\}\}$
The last group will be enqueued, being the smaller one in the split, then the loop will continue: the
iteration with $S = \{q_1, q_6\}$ and the input letter $a$ will split $\{q_0, q_3, q_4, q_5\}$, $q_4$ not being a predecessor of $S$. We get:

$$\Pi = \{\{q_0, q_3, q_5\}, \{q_2, q_7\}, \{q_1, q_6\}, \{q_4\}\}$$

Again, the last group will be enqueued and the iteration with $S = \{4\}$ and input letter $a$ will cause an
ultimate split of $\{q_1, q_6\}$. We end up with the same partition as in Moore’s algorithm, that is to say:

$$\Pi = \{\{q_0, q_3, q_5\}, \{q_2, q_7\}, \{q_1\}, \{q_4\}, \{q_6\}\}$$
Chapter 2

Cover automata

In this chapter, we present the theory of cover automata and detail the current results of this field. As one of the most important, we will show that a minimal deterministic cover automaton for a finite language has less or as many states as the minimal deterministic automaton for this language.

The following is mostly taken from Câmpeanu et al. (2001).

2.1 Basic idea

A finite language has a finite number of words; some of them are the longest ones. In the use of these languages, the length \( \ell \) of the longest words is often, if not always, known. An automaton for a finite language \( \mathcal{L} \), whose longest words have size \( \ell \), could be seen as:

- A checker for a word \( w \) to be in \( (\mathcal{L} \cup W), W \subset \Sigma^{>\ell} \).
- A checker for this word \( w \) to have a size less or equal to \( \ell \).

A cover automaton supposed deterministic (DFCA) for a language \( \mathcal{L} \) will not check the length of the input. In other words, it accepts all words of \( \mathcal{L} \) and possibly longer words than the longest ones in \( \mathcal{L} \). If, what we will suppose, \( \ell \) is known\(^1\), the length check will be made before the evaluation in the automaton and we will end up with both functionalities.

For instance, the following automaton is a cover automaton for \( \mathcal{L} = \{a, b, aa, aaa, bab\} \):

![Figure 2.1: Cover automaton for \( \mathcal{L} = \{a, b, aa, aaa, bab\}, \ell = 3 \) ](image)

\(^1\)It can be computed in a time linear in \( |\mathcal{L}| \).
Obviously, this automaton does not recognize strictly $L$, for example, the word “baba” is accepted here. However, the longest words have a length of $\ell = 3$, and “baba” is longer than that. Therefore, the length check failing, we can say that “baba” is not in $L$. It follows this formal definition:

**Definition 14** (Cover automaton). A deterministic finite cover automaton for a finite language $L_f$ of longest words of length $\ell$, is a DFA $A$ such that $L(A) \cap \Sigma^\leq \ell = L_f$. We say that $L(A)$ is a cover language of $L_f$.

The sequel of this report will focus on the cover minimization of an automaton, that is to say the building of a minimal deterministic cover automaton (MDFCA) from a DFA.

### 2.2 On the similarity of states

As we saw in Section 1.2, the construction of a minimal automaton is based on the $\equiv$ equivalence relation (Definition 13): if $p \equiv q$, $p$ and $q$ have the same future. In the context of cover automata, we need another important relation:

**Definition 15** (Similarity of states). Let $A = \langle Q, \Sigma, \varphi, I, F \rangle$, $(p, q) \in Q^2$. Suppose we reached $p$ and $q$ with, respectively, two words $w_1$ and $w_2$.

We say that:

- $w_1$ is similar to $w_2$ and note $w_1 \sim w_2$ if $\forall z \in \Sigma^*$ such that $|x \cdot z| \leq \ell \land |y \cdot z| \leq \ell, xz \in L \iff yz \in L$,
- $p$ is similar to $q$ and note $p \sim q$ if $w_1$ and $w_2$ are the shortest paths to respectively $p$ and $q$, and $w_1 \sim w_2$.

The defined $\sim$ relation is reflexive, symmetric but *not* transitive.

Intuitively, $p \sim q$ if $p$ and $q$ have the same bounded future.

For instance, in Figure 2.1, we have “bab” $\sim$ “a”, “a” $\sim$ “bab” but “a” $\not\sim$ “b”, so $p \not\sim q$.

**Definition 16** (Gap). The function $\text{gap} : Q \times Q \to \mathbb{N}$ computes the length of the shortest word that shows that its two arguments are dissimilar (i.e. a word that leads to a final state from one state and that does not from the other one). For convenience, if they are similar, the function returns $\ell$.

For instance, from the following automaton:

![Automaton for “(a + b)a”](image)

**Figure 2.2: Automaton for “(a + b)a”**
The gap function can be computed as follows:

<table>
<thead>
<tr>
<th>gap</th>
<th>q</th>
<th>r</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>$\ell$</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>r</td>
<td>$\ell$</td>
<td>$\ell$</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>$\ell$</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>$\ell$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For example, $\text{gap}(r, q)$ is 1 because the shortest word that shows a dissimilarity between $r$ and $q$ is “a”.

**Definition 17 (Level).** For $i$ the initial state of the automaton $\mathcal{A} = (Q, \Sigma, \varphi, I, F)$, we define

$$\forall q \in Q, \text{level}(q) = \min \{|w|, \ i \sim^w q\}$$

Intuitively, $\text{level}(q)$ is the word that represents the shortest path to $q$.

### 2.3 Minimal DFCA

**Definition 18 (Minimal DFCA).** An automaton $\mathcal{A}$ is a minimal DFCA for $L$ if for all automata $\mathcal{B}$ such that $\mathcal{B}$ is a cover automaton of $L$, $\text{Card}(\mathcal{A}) \leq \text{Card}(\mathcal{B})$.

The main theorem in the field of cover minimization is the equivalent of Theorem 2 but with the $\sim$ relation:

**Theorem 4 (Câmpeanu et al. (2001)).** A DFCA $\mathcal{A}$ is minimal if and only if $\forall (p, q) \in Q^2, p \neq q \Rightarrow p \sim q$.

A strong link can be made with the gap function: the computation of this function leads to the knowledge of similar classes: if $\text{gap}(r, s)$ is known and valued as $\ell$, then $r$ and $s$ are similar. Therefore:

**Fact 3.** An algorithm that computes the gap function would cover minimize the automaton.

It could be noticed that the $\equiv$ relation is more restrictive than $\sim$. In other words, $p \equiv q \Rightarrow p \sim q$; this implies that the similar classes are larger or as large as the equivalence classes. The following fact is then deduced:

**Fact 4.** The minimal DFCA for $L$ is smaller than the minimal DFA for $L$.

As a minimal DFA was first defined to be a canonical representation of a language, one can wonder if a minimal DFCA is still unique, this property being really useful when, for instance, comparing automata.

Alas, it is not the case, due to the fact that the similarity relation is not an equivalence relation. For example, the following automata are minimal DFCA for the language $L = \{a, ab, ba, aba, abb, baa, bab\}$:

![Figure 2.3: Two distinct minimal DFCA for $L = \{a, ab, ba, aba, abb, baa, bab\}$](image)
Some questions may arise, first, "Why cannot we minimize through classical minimization the minimal DFCA in order to have a canonical form?"

The answer is simple: minimal DFCA are minimal DFA. But the minimal DFCA $A$ is for a language $L_f$ whereas the view of $A$ as a minimal DFA would be for the language $L(A)$, and these languages are different (see Definition 14).

Another question could be "How many distinct automata can an algorithm that makes a minimization through similarity classes yield?"

The following theorem answers this question:

**Theorem 5** ( Câmpeanu and Paun (2002)). The number of minimal DFCA that can be obtained from a given minimal DFA with $n$ states by merging the similar states in the given DFA is upper bounded by

$$\frac{k_0!}{(2k_0 - n + 1)!}, \text{ with } k_0 = \left\lfloor \frac{4n - 9 + \sqrt{8n + 1}}{8} \right\rfloor$$

Moreover, this bound is reached, i.e. for any given positive integer $n$ we can find a minimal DFA with $n$ states, which has the number of minimal DFCA obtained by merging similar states equal to this maximum.

This is a penalty one has to deal with when working with cover automata. Another penalty is the following: when one considers a minimal automaton, the sink state is usually not represented. However, a minimization algorithm should, in theory, return a complete automaton. A drawback of cover automata is that the sink state could be merged with another state, leading to the creation of transitions that a trimmed automaton would not have:

![Figure 2.4: Merge of the sink state](image-url)
Chapter 3

Cover minimization

This chapter presents two algorithms that make cover minimization, that is to say, transform a DFA $A$ into a minimal DFCA for $L(A)$. The historically first algorithm presented in the original paper (Câmpeanu et al. (2001)), with a time complexity in $O(n^4)$, has a messy theoretical study; thus, we will not treat it, as the original authors published in the same breath of the first paper another one that details a $O(n^2)$ algorithm.

3.1 A $O(n^2)$ algorithm

This algorithm has been presented by Paun et al. (2001) and is a significant improvement of the previous one of Câmpeanu et al. (2001). As seen in Fact 3, computing the gap function is a way to compute the minimal DFCA. It is proposed, as in the very first algorithm, to compute efficiently this function.

3.1.1 Definitions

We assume that $A = (Q, \Sigma, \varphi, I, F)$ is a DFA accepting a finite language $L$ whose longest words have length $\ell$. $A$ is complete and without any useless state except the sink state.

One may refer to Paun et al. (2001) for the proofs of the following lemmas and theorems.

Definition 19 (Range). For $p, q \in Q$ and $p \neq q$, we define

$$\text{range}(p, q) = \ell - \max(\text{level}(p), \text{level}(q))$$

Intuitively, if $w_p$ and $w_q$ are the shortest words that lead to, respectively, $p$ and $q$, range$(p, q)$ is the maximum length of a word $w$ such that $|w_p| \leq \ell$ and $|w_q| \leq \ell$.

Definition 20 (State failure). Let $p, q \in Q$ and $z \in \Sigma^*$. We say that $p$ and $q$ fail on $z$ if $\varphi(p, z) \in F$ and $\varphi(q, z) \in Q \setminus F$ or vice versa, and $|z| \leq \text{range}(p, q)$.

Theorem 6. $p \neq q$ if and only if there exists $z \in \Sigma^*$ such that $p$ and $q$ fail on $z$.

With those definitions, gap can be re-defined as follows:

Definition 21 (Gap). If $p \neq q$,

$$\text{gap}(p, q) = \min\{|z| \text{ such that } p \text{ and } q \text{ fail on } z\}$$
Theorem 7. 1. Let \( d \) be the sink state of \( \mathcal{A} \). If \( \text{level}(d) > \ell \), then \( \forall q \in Q \setminus \{d\}, d \sim q \). If \( \text{level}(d) \leq \ell \), then \( \forall f \in F, d + f \) and \( \text{gap}(d, f) = 0 \).

2. If \( p \in F \) and \( q \in Q \setminus F \setminus \{d\} \) then \( p + q \) and \( \text{gap}(p, q) = 0 \).

Lemma 1. Let \( p, q \in Q^2 \), \( p \neq q \) and \( r = \varphi(p, a) \), \( t = \varphi(q, a) \), for some \( a \in \Sigma \), then \( \text{range}(p, q) \leq \text{range}(r, t) + 1 \).

The following theorems are the crux of the algorithm:

Theorem 8 (Paun et al. 2001). Let \( p \) and \( q \) be two states such that either \( p, q \in F \) or \( p, q \in Q \setminus F \). Then \( p + q \) if and only if there exists \( a \in \Sigma \) such that \( \varphi(p, a) = r \) and \( \varphi(q, a) = t \), \( r + t \), and

\[
\text{gap}(r, t) + 1 \leq \text{range}(p, q)
\]

Theorem 9 (Paun et al. 2001). If \( p + q \) such that \( p, q \in F \) or \( p, q \in Q \setminus F \), then

\[
\text{gap}(p, q) = \min\{\text{gap}(r, t) + 1 \mid \varphi(p, a) = r \text{ and } \varphi(q, a) = t, \text{ for } a \in \Sigma, r + t, \text{ and } \text{gap}(r, t) + 1 \leq \text{range}(p, q)\}
\]

3.1.2 Algorithm

The presented algorithm assumes that its input automaton is ordered, which means that the states are numbered from 0 to \( n \), and that the last one is the sink state.

The Theorem 9 naturally leads to the following algorithm:

\[
\begin{align*}
\text{Input:} & \quad \text{An ordered, reduced and complete DFA } \mathcal{A} = (Q, \Sigma, \varphi, I, F), \text{ with } n + 1 \text{ states, which accepts a finite language } \mathcal{L} \text{ whose longest words havelength } \ell \\
\text{Output:} & \quad \text{gap}(i, j) \text{ for each pair } i, j \in Q \text{ and } i < j.
\end{align*}
\]

\textbf{Algorithm:}

1. \textbf{for} each \( i \in Q \) \textbf{do}
   \begin{align*}
   & \text{compute } \text{level}(i) \\
   \end{align*}

2. \textbf{for} \( i = 0 \) to \( n - 1 \) \textbf{do}
   \begin{align*}
   & \text{if } \text{level}(n) \leq \ell \text{ then}
   & \quad \text{for each } i \in F \text{ do}
   & \quad \quad \text{gap}(i, n) = 0
   & \quad \text{else}
   & \quad \quad \text{gap}(i, n) = \ell
   & \end{align*}

3. \textbf{for} each pair \( i, j \in Q \setminus \{n\} \) such that \( i < j \) \textbf{do}
   \begin{align*}
   & \text{if } i \in F \text{ and } j \in Q \setminus F \text{ or vice versa}
   & \quad \text{gap}(i, j) = 0
   & \text{else}
   & \quad \quad \text{gap}(i, j) = \ell
   & \end{align*}

4. \textbf{for} \( i = n - 2 \) down to 0 \textbf{do}
   \begin{align*}
   & \textbf{for} \ j = n \text{ down to } i + 1 \textbf{ do}
   & \quad \textbf{for} \ a \in \Sigma \textbf{ do}
   & \quad \quad \text{let } i' = \varphi(i, a) \text{ and } j' = \varphi(j, a)
   & \quad \quad \text{if } i' \neq j'
   & \quad \quad \quad \text{g = if } (i' < j') \text{ then } \text{gap}(i', j') \text{ else } \text{gap}(j', i')
   & \quad \quad \text{if } g + 1 \leq \text{range}(i, j) \textbf{ then}
   & \quad \quad \quad \text{gap}(i, j) = \min(\text{gap}(i, j), g + 1)
   & \end{align*}
The time complexity is established as follows:

- Each of Step 1 and Step 2 is $O(n)$,
- Step 3 takes $O(n^2)$ iterations,
- Step 4 has two nested loops, each of which has $O(n)$ iterations. Each inner iteration is $O(|\Sigma|)$, where $|\Sigma|$ is a constant.

Therefore, this algorithm that computes the gap function is in $O(n^2)$.

3.1.3 Example

Let $\mathcal{A} = (Q, \Sigma, \varphi, i, F)$ be the automaton denoted by Figure 3.1. Clearly, $L(\mathcal{A}) = \{abc, ababc, abababc\}$ and $\ell = 7$.

We suppose the existence of a sink state $q_9$, not represented here for the sake of clarity.

We follow the algorithm of Section 3.1.2 that computes the gap function. At Step 1, level($q$) is calculated for each $q \in Q$:

<table>
<thead>
<tr>
<th>State</th>
<th>$q_1$</th>
<th>$q_2$</th>
<th>$q_3$</th>
<th>$q_4$</th>
<th>$q_5$</th>
<th>$q_6$</th>
<th>$q_7$</th>
<th>$q_8$</th>
<th>$q_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

After Step 2, Step 3 and Step 4, we have gap($q_i, q_j$) for each $1 \leq i \leq 8, 2 \leq j \leq 9$ and $i < j$ as follows:

<table>
<thead>
<tr>
<th>$q_1$</th>
<th>$q_2$</th>
<th>$q_3$</th>
<th>$q_4$</th>
<th>$q_5$</th>
<th>$q_6$</th>
<th>$q_7$</th>
<th>$q_8$</th>
<th>$q_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>$q_2$</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>$q_3$</td>
<td></td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$q_4$</td>
<td></td>
<td></td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>$q_5$</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$q_6$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>$q_7$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$q_8$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
In the above table, the states are equal if $\text{gap}(q_i, q_j) = 7$, $ell$ being 7. The merging of similar states results in the following minimal DFCA for $\mathcal{L}(\mathcal{A})$ (the sink state is still not represented):

\[
\begin{align*}
\{q_1\} & \xrightarrow{a} \{q_2, q_3, q_6\} & \xrightarrow{b} \{q_3, q_5, q_7\} & \xrightarrow{c} \{q_8\} \\
\{q_1\} & \xrightarrow{a} \{q_2, q_3, q_6\}
\end{align*}
\]

Figure 3.2: MDFCA for $\mathcal{L}(\mathcal{A}) = \{abc, ababc, abababc\}$

### 3.2 A $O(n \log(n))$ algorithm

This algorithm, which has been presented by Körner (2003), is a major and the last improvement in cover minimization. Inspired by the algorithm of Hopcroft (1971), it is unlikely that an algorithm with a smaller time complexity will be found as it equals the better one for classical minimization.

#### 3.2.1 Algorithm

Basically, it uses Hopcroft's algorithm with regard to a bounded evaluation of each states.

The algorithm can be expressed as the following to emphasize the similarities with Hopcroft's algorithm:

```plaintext
initialize \( \Pi = \{F, Q \setminus F\} \),
put \((F, 0)\) in a "To_treat" queue \( T \),
while \( T \) is not empty {
    remove a set \((S, k)\) of \( T \),
    for each letter \( l \) in \( \Pi \) {
        compute \( X \): the list of states that go in \( S \),
        \( Y \): the list of states that don't,
        with \( \text{level}(p) + k \leq \ell \)
        for each sets \( \pi \) in \( \Pi \) {
            if \( \pi \cap X \neq \emptyset \) and \( \pi \cap Y \neq \emptyset \) {
                split \( \pi \) in \( \pi \cap X \) and \( \pi \cap Y \),
                enqueue the smaller one in \( T \) with \( k + 1 \).
            }
        }
    }
}
```
The modifications from the original algorithm are the following:

- **l.2:** The final states are enqueued together with a 0; this number represents the distance from the states enqueued to a final state,
- **l.4:** $k$, this distance, is removed with the set of states,
- **l.8:** The states are considered with regard to their level and the distance from them to a final state ($k$); it results in considering a state if and only if the length of the word that led to it is less than $\ell$.
- **l.12:** The new set is enqueued with an incremented distance to the final states.

As the computing of the level function is in $O(n)$, the whole algorithm is still in $O(n \log(n))$.

### 3.2.2 Example

This example considers the following automaton $B = \langle Q, \Sigma, \varphi, i, F \rangle$ which recognizes the language $\mathcal{L}(B) = \{bc, babc\}$ with $\ell = 4$.

We suppose the existence of a sink state $q_6$ not represented in the figure.

![Image of DFA for $\mathcal{L}(B) = \{bc, babc\}$](image)

One of the first iteration will consider $S = \{q_3\}$, $k = 0$, with the input letter $c$. The set $\{q_1, q_2, q_3, q_4, q_6\}$ will be split into $\{q_2, q_4\}$ and $\{q_1, q_3, q_6\}$, and the latter will be enqueued together with $k + 1 = 1$.

An other iteration will then consider $S = \{q_2, q_4\}$, $k = 1$ and the input letter $b$. The set $\{q_1, q_3, q_6\}$ is split into $\{q_1, q_3\}$ and $\{q_6\}$, and $\{q_6\}$, $k + 1 = 2$ will be enqueued.

If we now consider $S = \{q_6\}$, $k = 2$ and the input letter $a$, the list of states going in $S$ is $X = \{q_1, q_3, q_4, q_5, q_6\}$. This $X$ would split the set $\{q_2, q_4\}$ if $q_4$ was in the actual $X$: indeed, level($q_4$) + ($k = 2$) = 5 $\geq \ell$, so $q_4$ is not in $X$.

The algorithm will not produce anymore split, and the resulting automaton is the following:

![Image of MDFCA for $\mathcal{L}(B) = \{bc, babc\}$](image)
Chapter 4

Implementation in the VAUCANSON framework

The two algorithms of Section 3.1 and Section 3.2 have been implemented using the VAUCANSON framework. In this chapter, we present those implementations, together with a study of their respective performances.

4.1 Implementations

4.1.1 The $O(n^2)$ algorithm

The implementation is quite straightforward, it uses only basic functionalities of VAUCANSON. The “Step”s in comment refer to the ones of Section 3.1.

```c++
void compute_gaps (const automaton_t& aut,
const unsigned l,
gaps_t& gaps,
vstates_t& states,
levels_t& levels)
{
    const alphabet_t& alphabet = aut.series().monoid().alphabet();

    // Step 1.
    for_each_initial_state (istate, aut)
        compute_levels (aut, *istate, 0, levels);

    // Get the sink state
    hstate_t sink = get_sink_state (aut);

    // Compute $Q - sink$
    states.clear ();
    for_each_state (istate, aut)
        if (*istate != sink)
            states.push_back (*istate);
```
// Step 2.
for_all (vstates_t, istate, states)
gaps[*istate][sink] = 1;

if (levels[sink] <= 1)
    for_each_final_state (istate, aut)
        gaps[*istate][sink] = 0;

// Step 3.
for_all (vstates_t, i, states)
    for_all (vstates_t, j, states) {
        if (*i < *j)
            continue;
        if ((aut.is_final (*i) and not aut.is_final (*j)) or
            (not aut.is_final (*i) and aut.is_final (*j)))
            gaps[*i][*j] = 0;
        else
            gaps[*i][*j] = 1;
    }

// Now add the sink state at the end of the container.
states.push_back (sink);

// Step 4.
delta_t delta;

vstates_t::reverse_iterator i = states.rbegin();
for (++i, ++i; i != states.rend(); ++i) {
    for (vstates_t::reverse_iterator j = states.rbegin();
        j != i; ++j)
        for_each_letter (iletter, alphabet) {
            delta.clear();
            aut.letter_deltac (delta, *i, *iletter, delta_kind::states ());
            hstate_t ip = *(delta.begin ());
            delta.clear ();
            aut.letter_deltac (delta, *j, *iletter, delta_kind::states ());
            hstate_t jp = *(delta.begin ());

            if (ip != jp) {
                unsigned g = (ip < jp) ? gaps[ip][jp] : gaps[jp][ip];
                if (g + 1 <= range (*i, *j)) {
                    gaps[*i][*j] = std::min (gaps[*i][*i], g + 1);
                }
            }
        }
    }
}
4.1.2 The $O(n \log(n))$ algorithm

This implementation of the algorithm presented in Section 3.2 computes similarity states decompositions (SSD). Only the relevant portions of code are kept here. The comments are put to help the reader know where in algorithm of Section 3.2 we are.

Thanks to VAUCANSON, this implementation is a great improvement of the original 400 lines implementation presented in Körner (2003).

```c
void compute_ssd (automaton_t& aut,
    unsigned l,
    levels_t& levels,
    ssds_t& Qs)
{
    // Compute level(q) for all q <= Q
    levels.clear();
    for_each_initial_state (istate, aut)
        compute_levels (aut, *istate, 0, levels);

    // Q(0) = Q \ F; Q(1) = F
    for_each_state (istate, aut)
        if (not aut.is_final (*istate))
            Qs[0].insert (*istate);
        else
            Qs[1].insert (*istate);

    unsigned r = 2; // ssds index.
    ssd_queue_t T;  // FIFO queue for splitting.
    hstates_t X;
    hstates_t Y;
    delta_t delta;
    const alphabet_t& alphabet = aut.series().monoid().alphabet();

    // Initialize T with (F, 0)
    T.push (ssd_pair_t (hstates_t (), 0));
    for_all (hstates_t, istate, Qs[1])
        T.front().first.insert (*istate);

    // Main loop
    while (not T.empty()) {
        // First element of T is T.front()
        ssd_pair_t S_k = T.front();
        T.pop();

        for_each_letter (iletter, alphabet) {
            // X = \{ p | delta(p, *iletter) <= S and level(p) + k < l\}
```
4.1 Implementations

// Y = \{ p \mid \text{delta}(p, *i\text{lette}r) \text{ not} \leq S \text{ and } \text{level}(p) + k < l \}
X.clear();
Y.clear();
for_each_state(istate, aut) {
    delta.clear();
    aut.letter_deltac(delta, *istate, *iletter,
        delta_kind::states());
    // w.r.t level and k.
    if (levels[*istate] + S_k.second < l) {
        bool b_is_in_S = true;
        for_all(delta_t, isucc, delta)
            if (S_k.first.find(*isucc) == S_k.first.end()) {
                b_is_in_S = false;
                break;
            }
        if (b_is_in_S)
            X.insert(*istate);
        else
            Y.insert(*istate);
    }
}

for (int i = r - 1; i >= 0; --i) {
    hstates_t Qi_inter_X, Qi_minus_Qi_inter_X;
    bool b_Qi_inter_Y_empty = true;
    const unsigned n_Qi_size = Qs[i].size();
    // Compute Qi_inter_X, Qi minus X and Qi_inter Y
    for_all(hstates_t, istate, Qs[i]) {
        if (X.find(*istate) != X.end())
            Qi_inter_X.insert(*istate);
        else
            Qi_minus_Qi_inter_X.insert(*istate);
        if (Y.find(*istate) != Y.end())
            b_Qi_inter_Y_empty = false;
    }

    // if Qi_inter X != 0 and Qi_inter Y != 0
    if (not Qi_inter_X.empty() and not b_Qi_inter_Y_empty) {
        // Z is Qi_inter_X
        // If |Z| <= |Qi \setminus Z|
        if (Qi_inter_X.size() <= Qi_minus_Qi_inter_X.size()) {
            Qs[r] = Qi_inter_X;
            Qs[i] = Qi_minus_Qi_inter_X;
        } else {
            Qs[r] = Qi_minus_Qi_inter_X;
            Qs[i] = Qi_inter_X;
        }
        assert(Qs[i].size() + Qs[r].size() == n_Qi_size);
        T.push(ssd_pair_t(Qs[r], S_k.second + 1));
        r += 1;
    }
}}}}
4.2 Performances

At the moment this report is written, performance tests are not deeply made. A first result is the following:

Protocol:

- Let $\mathcal{L}$ be a language composed of random words over $\Sigma$, with $\Sigma = \{a, b\}$.
- Let $\mathcal{A}$ be a deterministic automaton that recognizes $\mathcal{L}$.
- We will compute the minimal automaton for $\mathcal{A}$, and a minimal cover automaton for it.

The results are the following:

| Cardinal | Time (s) | | | |
|----------|----------|----------|----------|----------|----------|
|          | $\mathcal{A}$ | Words | Min. DFA | Min. DFCA | Hopcroft | $O(n^2)$ | $O(n\log(n))$ |
| 55       |           | 20     | 37       | 30        | 0.01      | 0.01      | 0.02       |
| 412      |           | 40     | 172      | 140       | 0.1       | 0.9       | 0.5        |
| 963      |           | 60     | 498      | 440       | 0.2       | 3.0       | 1.1        |
| 1418     |           | 80     | 742      | 698       | 0.5       | 6.4       | 3.1        |
| 2437     |           | 100    | 1481     | 1323      | 0.9       | 34.5      | 9.2        |

More tests and comments on performances are to be made in the next few months. Câmpeanu et al. plan to make their tests too, but the best protocol to do so is not really defined at this time: real-world applications tend to not give good results and we have to focus on specific uses of cover automata.
Conclusion

In this technical report, we presented the theory of cover automata, an efficient way to represent finite languages. This field try to fulfill a real need from the world of language processing, which commonly uses finite languages.

Cover automata are useful thanks to algorithms that “cover minimize” automata, giving an automaton with the same language of the input one, modulo the size of the words recognized.

The best algorithm known is in $O(n \log(n))$ and is inspired by Hopcroft’s algorithm, which is the most performant algorithm for classical minimization. It is rather unlikely that this bound would be improved, and it is assumed to be tight for both minimizations.

Ongoing researches in this field are focused on the test of performance: scientists are interested in knowing in which cases cover automata can significantly reduce the number of states of an automaton.
Bibliography


Câmpeanu, C. and Paun, A. (2002). The number of similarity relations and the number of minimal deterministic finite cover automata. In CIAA, pages 67–76.


## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Automata for “a*”</td>
<td>6</td>
</tr>
<tr>
<td>1.2</td>
<td>Automaton for “(a + b)c”</td>
<td>7</td>
</tr>
<tr>
<td>1.3</td>
<td>Minimal automaton for “(a + b)c”</td>
<td>8</td>
</tr>
<tr>
<td>1.4</td>
<td>The automaton $M$</td>
<td>9</td>
</tr>
<tr>
<td>1.5</td>
<td>Minimal automaton for $L(M)$</td>
<td>10</td>
</tr>
<tr>
<td>2.1</td>
<td>Cover automaton for $L = {a, b, aa, aaa, bab}$, $\ell = 3$</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Automaton for “(a + b)a”</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Two distinct minimal DFCA for $L = {a, ab, ba, aba, abb, baa, bab}$</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>Merge of the sink state</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>A DFA for $L(\mathcal{A}) = {abc, ababc, abababc}$</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>MDFCA for $L(\mathcal{A}) = {abc, ababc, abababc}$</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>A DFA for $L(\mathcal{B}) = {bc, babc}$</td>
<td>20</td>
</tr>
<tr>
<td>3.4</td>
<td>MDFCA for $L(\mathcal{B}) = {bc, babc}$</td>
<td>20</td>
</tr>
</tbody>
</table>
Index

Alphabet, 5
Automaton, 5, 5
   complete –, 6
   cover –, see DFCA
deterministic finite –, see DFA
finite –, 5
isomorphism of –, 7
minimal –, see MDFA
minimal deterministic –, see MDFA
minimal deterministic cover –, see MDFCA
non-deterministic finite –, see NFA
   real-time –, 5

Complexity, 18
Cover automata, 4
Cover minimization, 13, 14, 16, 19

DFA, 6, 13
DFCA, 12, 13, 14

Equivalency, 7
Evaluation, 6

Gap, 13, 16
gap, 18

Hopcroft’s algorithm, 19
Hopcroft’s algorithm, 10

Language, 5
   cover, 13
   finite, 5, 16
   of an automaton, 5
Level, 14

MDFA, 6, 7, 8, 13
MDFCA, 13, 14, 14
Minimization, 8, 10
   cover –, see Cover minimization
Moore’s algorithm, 8

NFA, 6

Range, 16

Similarity, 13
State failure, 16

Trim, 6

Vaucanson, 21, 23

Words, 5